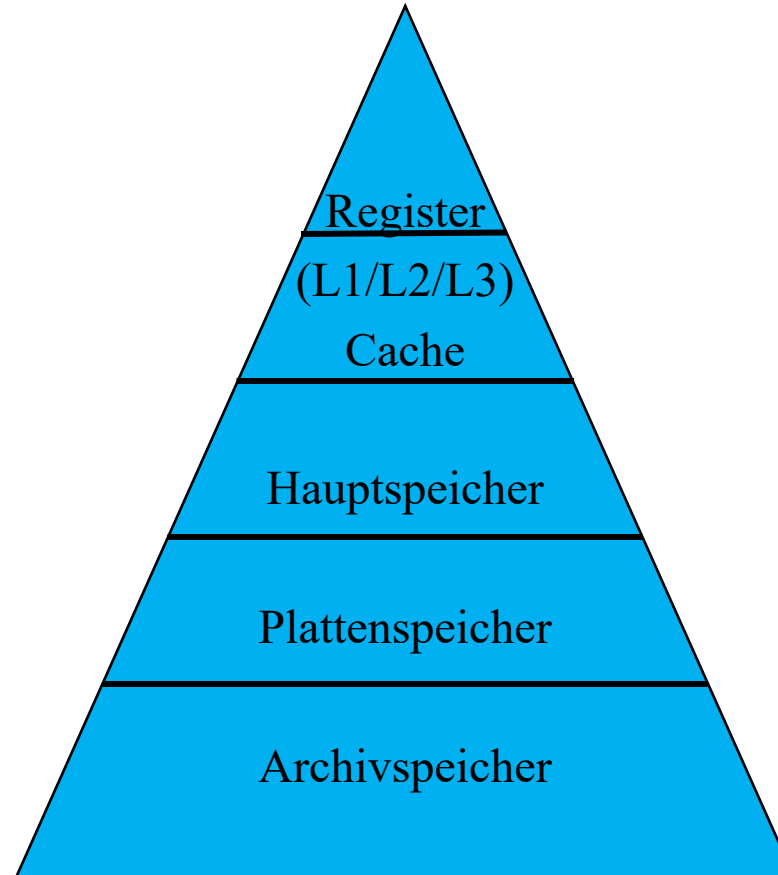


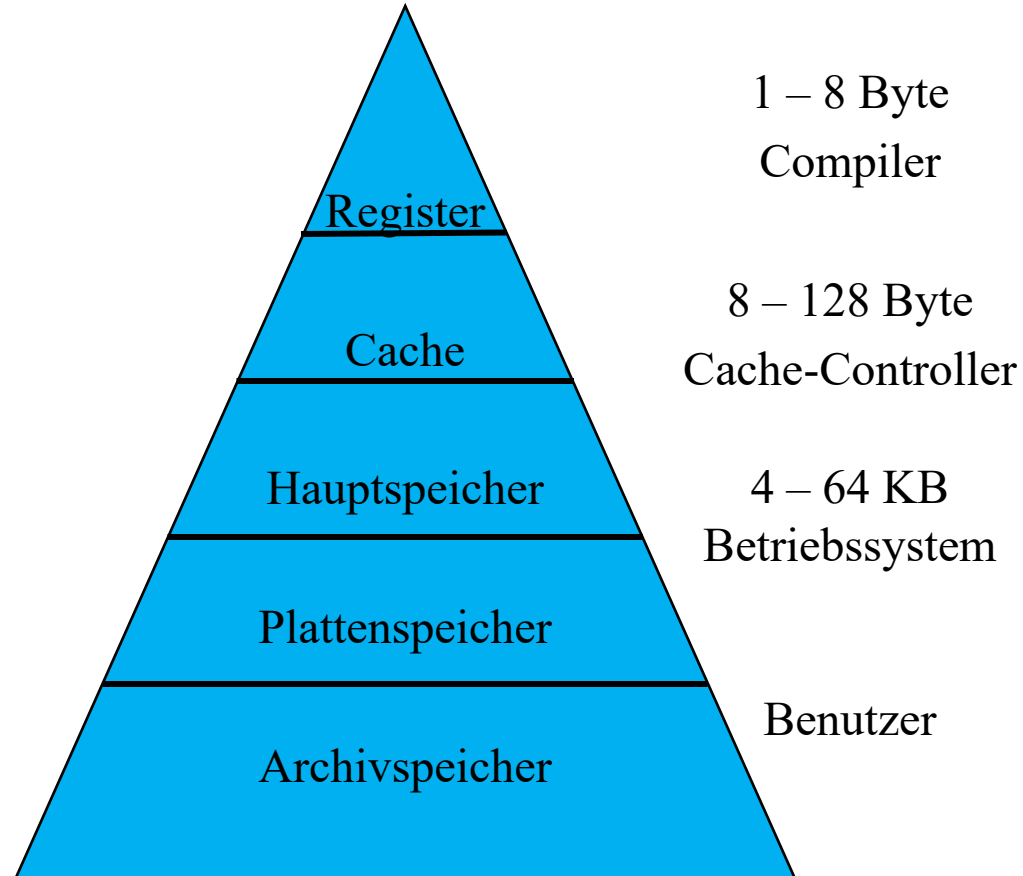
# Kapitel 7

## Physische Datenorganisation

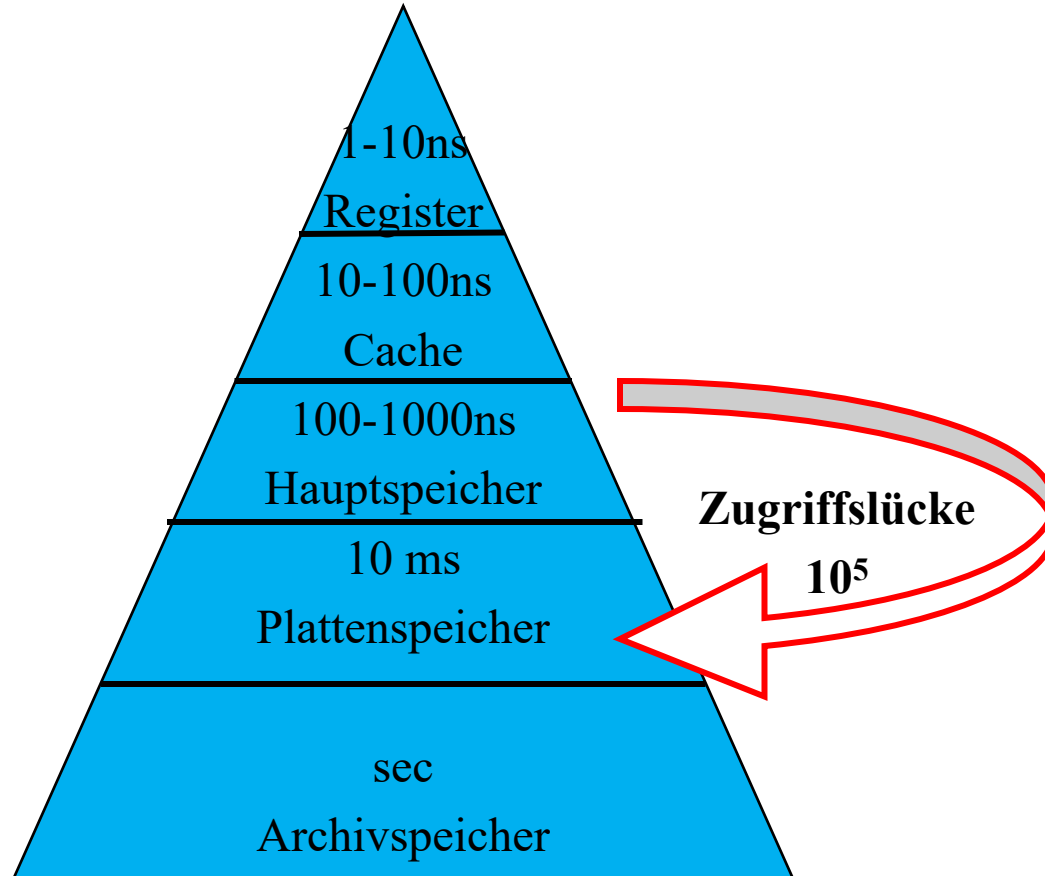
---

- **Speicherhierarchie**
- **Hintergrundspeicher / RAID**
- **Speicherstrukturen**
- **B-Bäume**
- **Hashing**
- **R-Bäume**

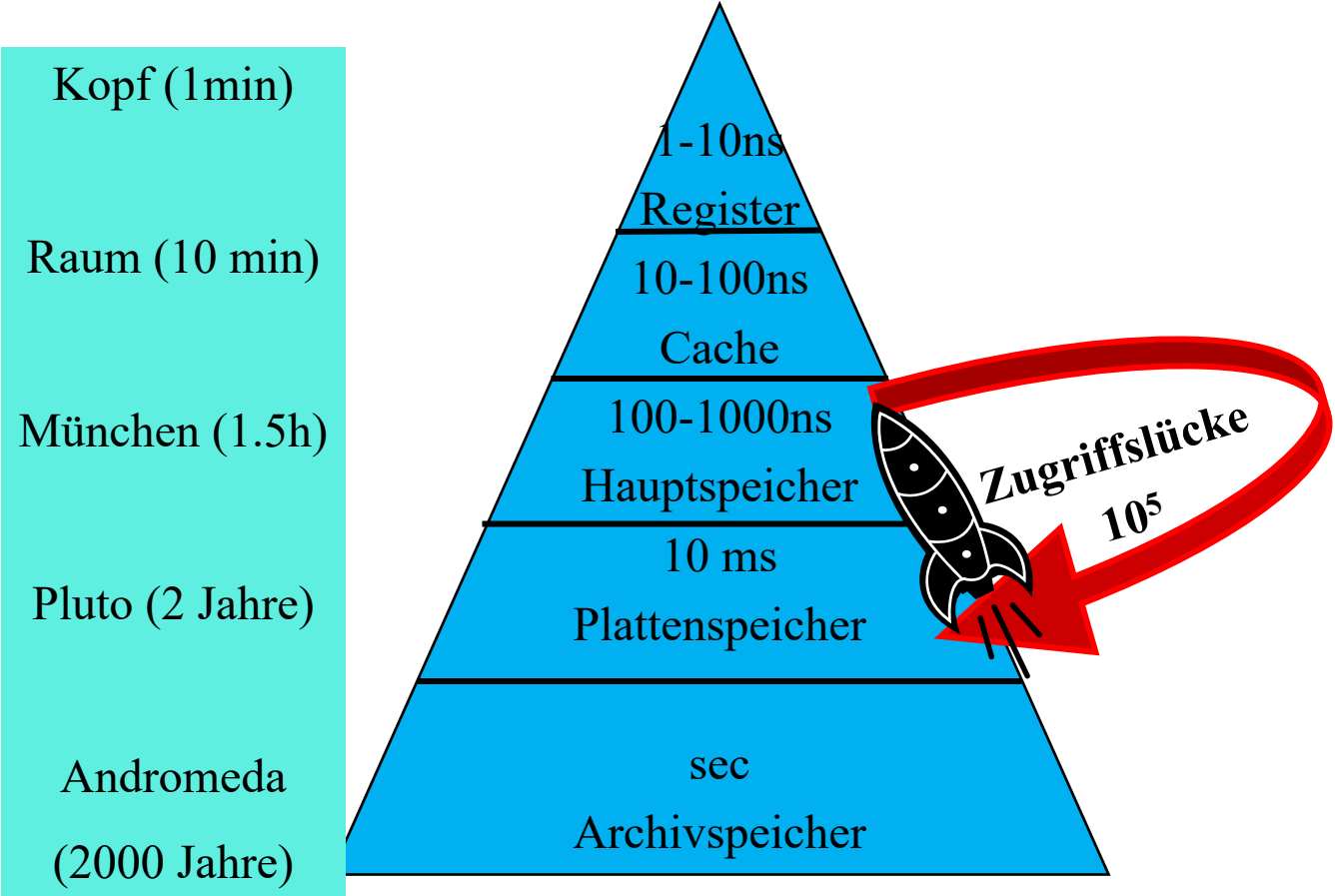


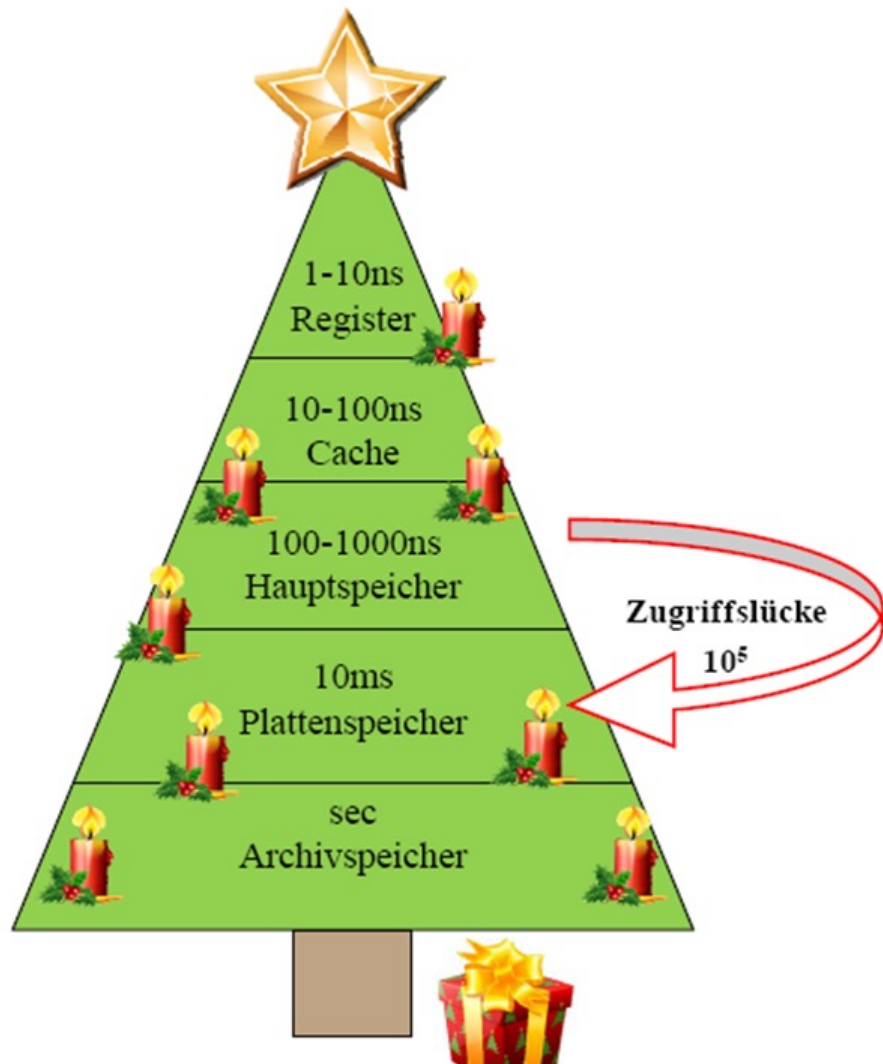


# Überblick: Speicherhierarchie



# Überblick: Speicherhierarchie





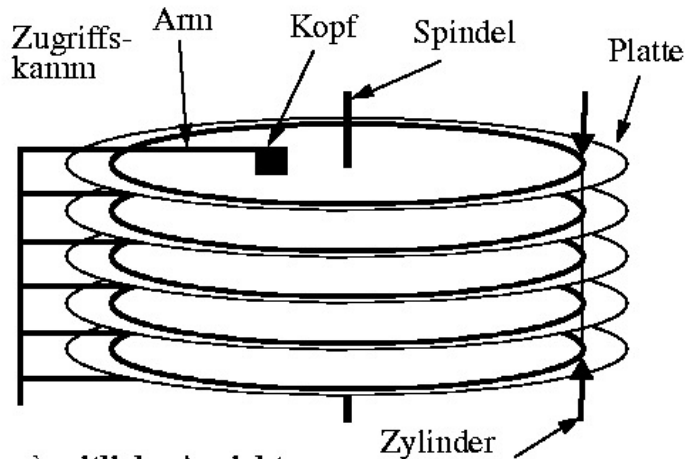
# Magnetplattenspeicher

## Aufbau

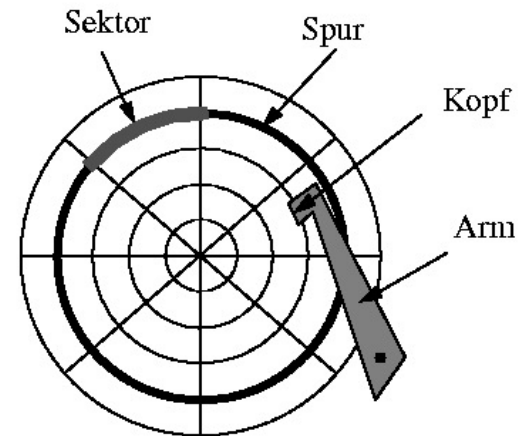
- mehrere gleichförmig rotierende Platten, für jede Plattenoberfläche ein Schreib-/Lesekopf
- jede Plattenoberfläche ist eingeteilt in Spuren
- die Spuren sind formatiert als Sektoren fester Größe (Slots)
- Sektoren (typischerweise 1 - 8 KB) sind die kleinste Schreib-/Leseinheit auf einer Platte

## Adressierung

- Zylindernummer, Spurnummer, Sektornummer
- jeder Sektor speichert selbstkorrigierende Fehlercodes; bei nicht behebbaren Fehlern erfolgt automatische Abbildung auf Ersatzsektoren



a) seitliche Ansicht



b) Draufsicht

# Lesen von Daten von der Platte

Seek Time: Arm positionieren

- 5ms

Latenzzeit:  $\frac{1}{2}$  Plattenumdrehung (im Durchschnitt)

- 10000 Umdrehungen / Minute
- → Ca 3ms

Transfer von der Platte zum Hauptspeicher

- 100 Mb /s → 15 MB/s



# Random versus Chained IO

1000 Blöcke à 4KB sind zu lesen

Random I/O

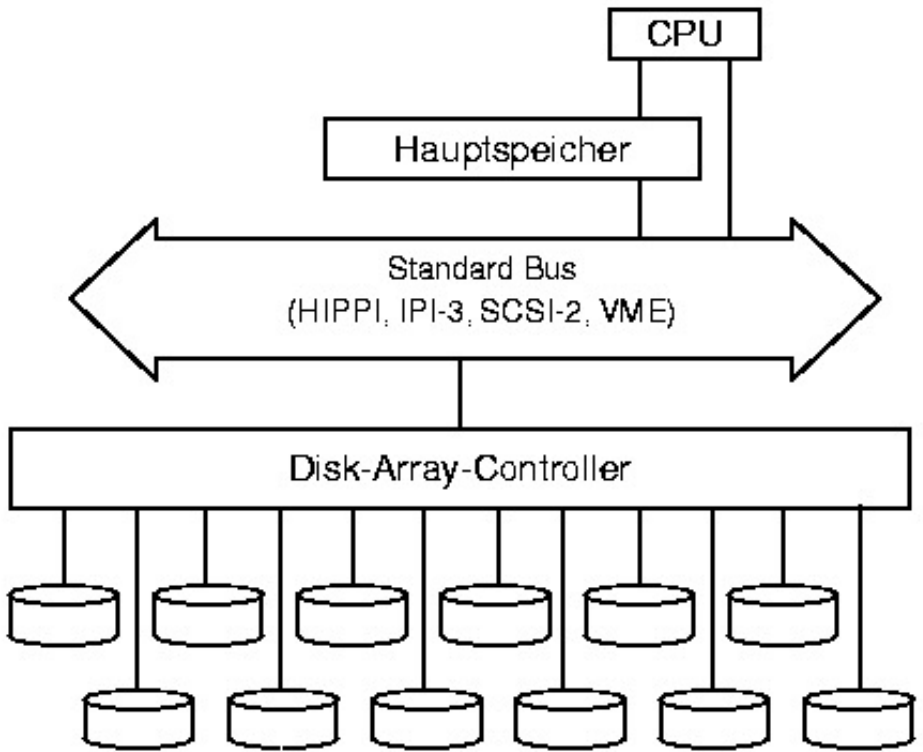
- Jedesmal Arm positionieren
- Jedesmal Latenzzeit
- →  $1000 * (5 \text{ ms} + 3 \text{ ms}) + \text{Transferzeit von 4 MB}$
- →  $> 8000 \text{ ms} + 300\text{ms} \rightarrow 8\text{s}$

Chained IO

- Einmal positionieren, dann „von der Platte kratzen“
- →  $5 \text{ ms} + 3\text{ms} + \text{Transferzeit von 4 MB}$
- →  $8\text{ms} + 300 \text{ ms} \rightarrow 1/3 \text{ s}$

Also ist chained IO **ein bis zwei Größenordnungen schneller** als random IO in Datenbank-Algorithmen unbedingt beachten !

# Disk Arrays → RAID-Systeme



*„The Problem with Many Small Disks: Many Small Faults“*

Disk-Array mit N Platten: ohne Fehlertoleranzmechanismen N-fach erhöhte Ausfallwahrscheinlichkeit

⇒ System ist unbrauchbar

Begriffe

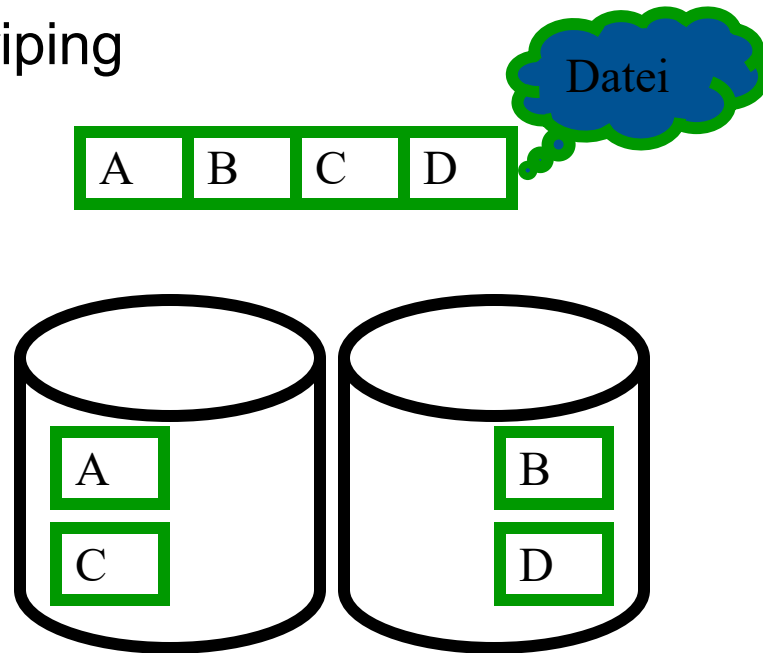
- Mean Time To Failure (MTTF): Erwartungswert für die Zeit (von der Inbetriebnahme) bis zum Ausfall einer Platte
- Mean Time To Repair (MTTR): Erwartungswert für die Zeit zur Ersetzung der Platte und der Rekonstruktion der Daten
- Mean Time To Data Loss (MTTDL): Erwartungswert für die Zeit bis zu einem nicht-maskierbaren Fehler

Disk-Array mit N Platten ohne Fehlertoleranzmechanismen:  $MTTDL = MTTF/N$

Der Schlüssel zur Fehlertoleranz ist Redundanz => Redundant Arrays of Independent Disks (RAID)

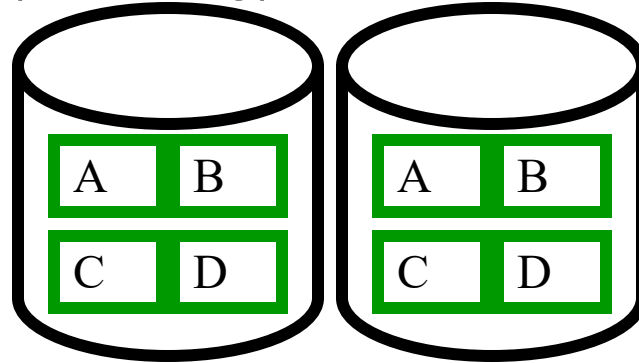
- Durch Replikation der Daten (z.B. Spiegelplatten) – RAID1
- Durch zusätzlich zu den Daten gespeicherte Error-Correcting-Codes (ECCS), z.B. Paritätsbits (RAID-4, RAID-5)

# RAID 0: Striping



Lastbalancierung wenn alle Blöcke mit gleicher Häufigkeit gelesen/geschrieben werden  
Doppelte Bandbreite beim sequentiellen Lesen der Datei bestehend aus den Blöcken ABCD...  
Aber: Datenverlust wird immer wahrscheinlicher, je mehr Platten man verwendet (Stripingbreite = Anzahl der Platten, hier 2)

# RAID 1: Spiegelung (mirroring)



Datensicherheit: durch Redundanz aller Daten (Engl. mirror)

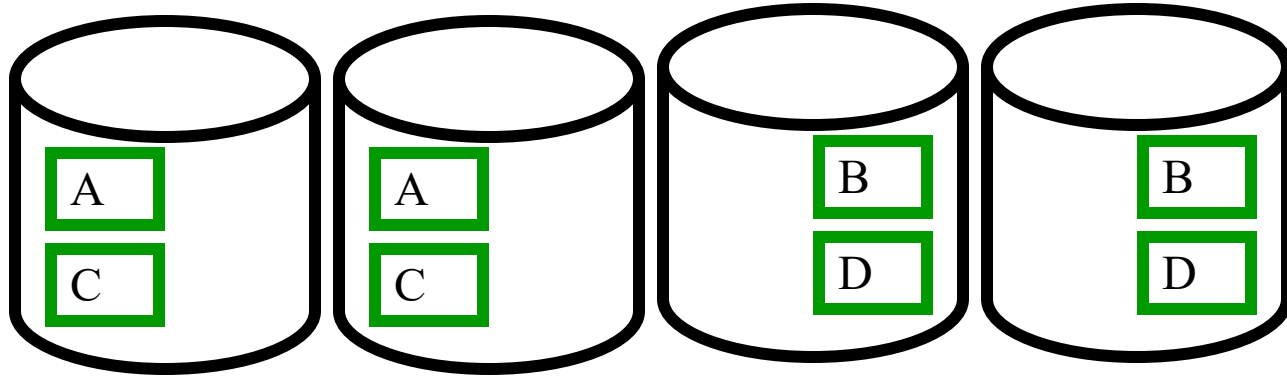
Doppelter Speicherbedarf

Lastbalancierung beim Lesen: z.B. kann Block A von der linken oder der rechten Platte gelesen werden

Aber beim Schreiben müssen beide Kopien geschrieben werden

- Kann aber parallel geschehen
- Dauert also nicht doppelt so lange wie das Schreiben nur eines Blocks

# RAID 0+1: Striping und Spiegelung



Kombiniert RAID 0 und RAID 1

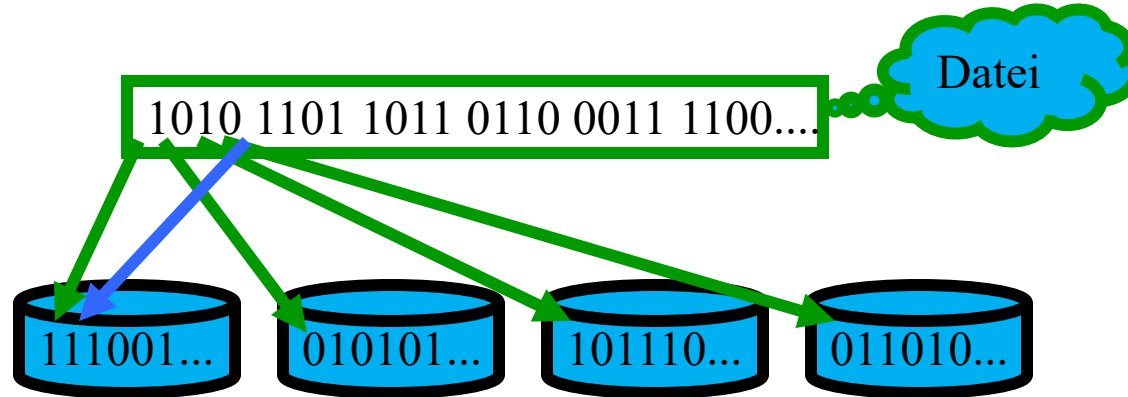
Immer noch doppelter Speicherbedarf

Zusätzlich zu RAID 1 erzielt man hierbei auch eine höhere Bandbreite beim Lesen der gesamten Datei ABCD....

Wird manchmal auch als RAID 10 bezeichnet

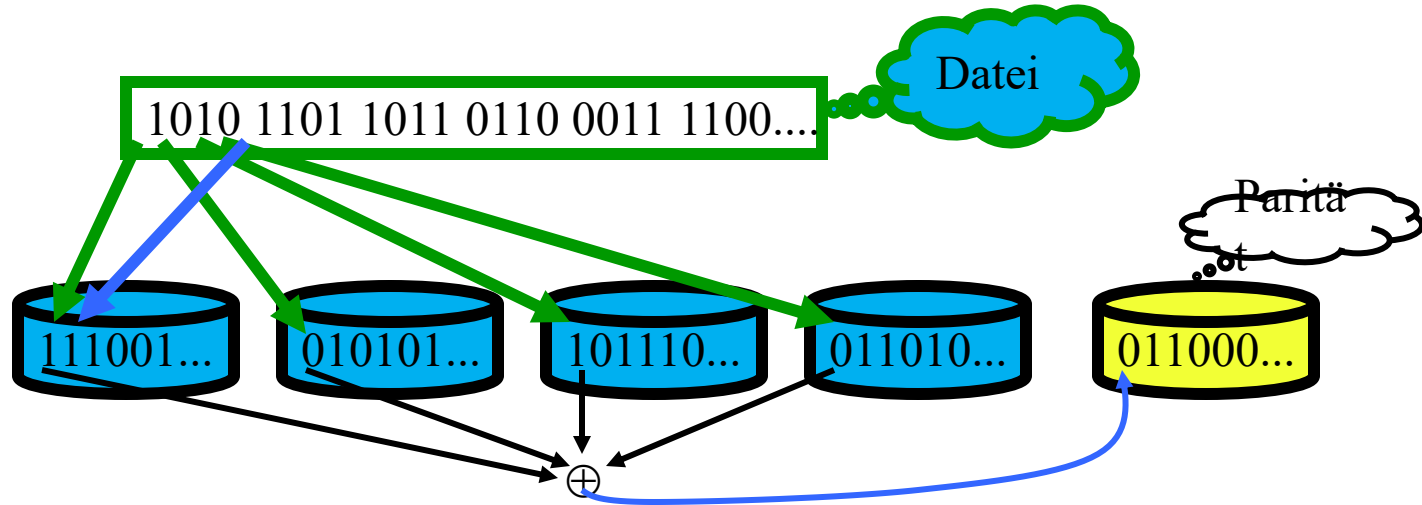
# RAID 2: Striping auf Bit-Ebene

Anstatt ganzer Blöcke, wie bei RAID 0 und RAID 0+1, wird das Striping auf Bit- (oder Byte-) Ebene durchgeführt



Es werden zusätzlich auf einer Platte noch Fehlererkennungs- und Korrekturcodes gespeichert  
In der Praxis nicht eingesetzt, da Platten sowieso schon Fehlererkennungscode verwalten

# RAID 3: Striping auf Bit-Ebene, zusätzliche Platte für Paritätsinfo



Das Striping wird auf Bit- (oder Byte-) Ebene durchgeführt  
Es wird auf einer Platte noch die Parität der anderen Platten gespeichert.

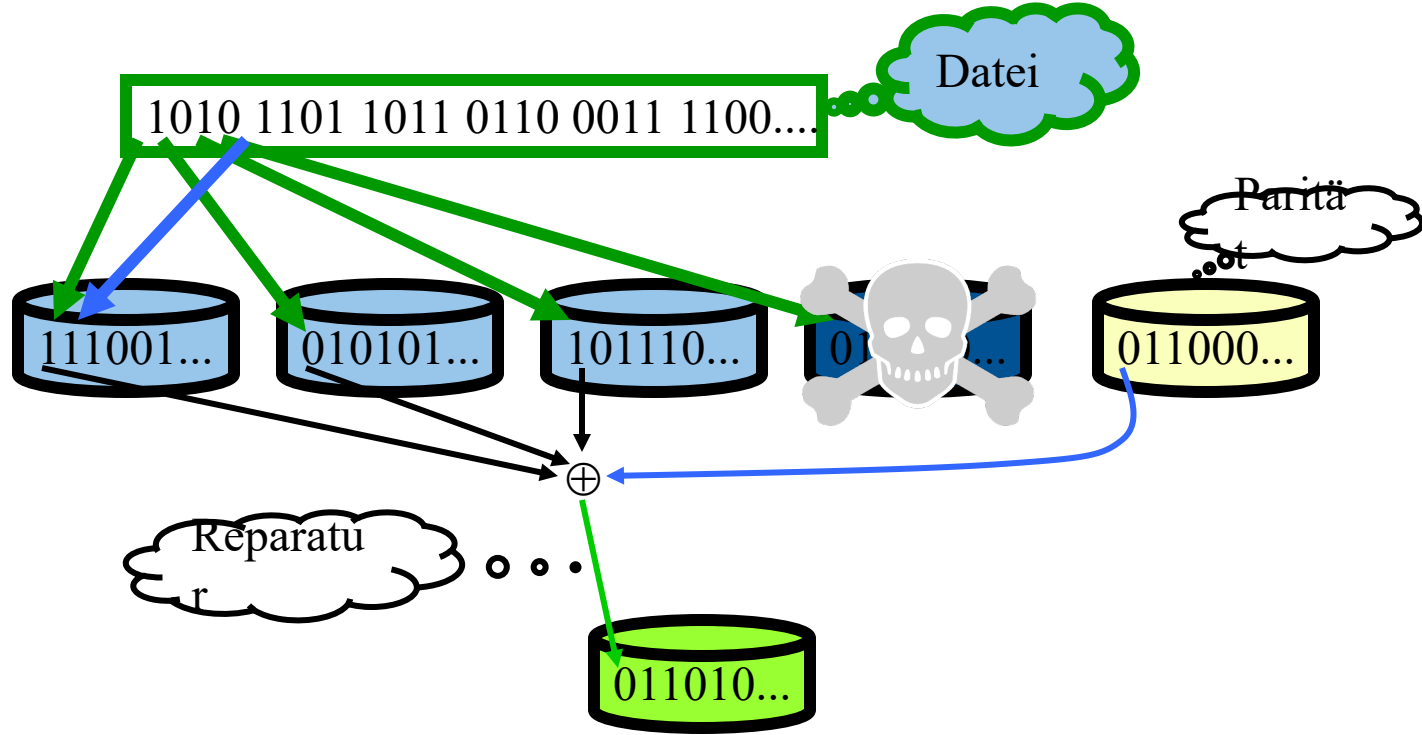
Parität = bit-weise xor  $\oplus$

Dadurch ist der Ausfall einer Platte zu kompensieren  
Das Lesen eines Blocks erfordert den Zugriff auf alle Platten

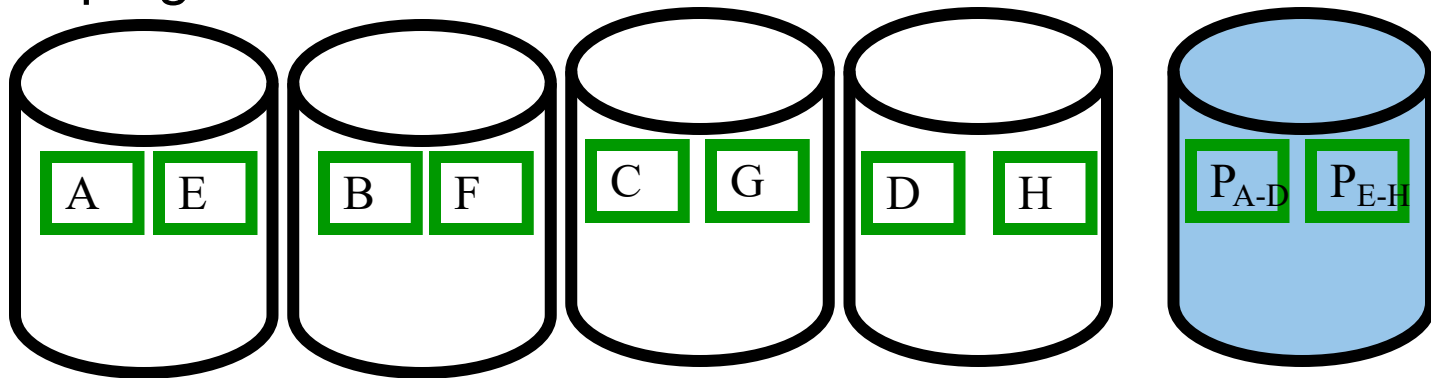
- Verschwendung von Schreib/Leseköpfen
- Alle marschieren synchron



# RAID 3: Plattenausfall



## RAID 4: Striping von Blöcken



Bessere Lastbalancierung als bei RAID 3

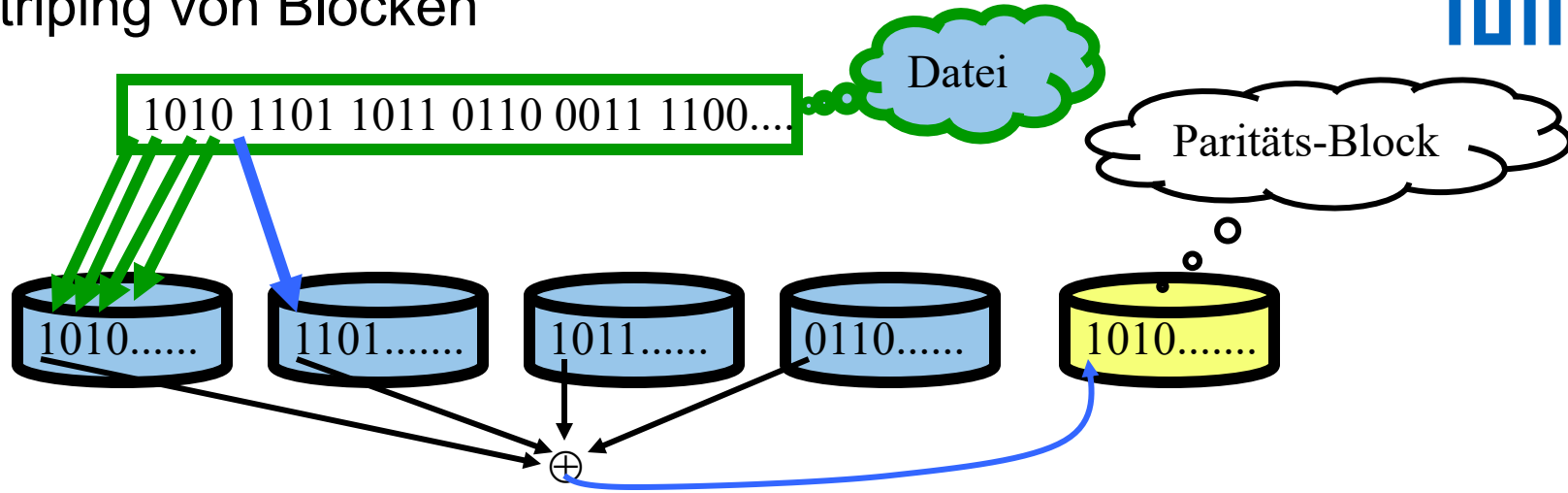
Flaschenhals bildet die Paritätsplatte

Bei jedem Schreiben muss darauf zugegriffen werden

- Bei Modifikation von Block A zu A' wird die Parität P<sub>A-D</sub> wie folgt neu berechnet:
- $P'_{A-D} := P_{A-D} \oplus A \oplus A'$

D.h. bei einer Änderung von Block A muss der alte Zustand von A und der alte Paritätsblock gelesen werden und der neue Paritätsblock und der neue Block A' geschrieben werden

## RAID 4: Striping von Blöcken



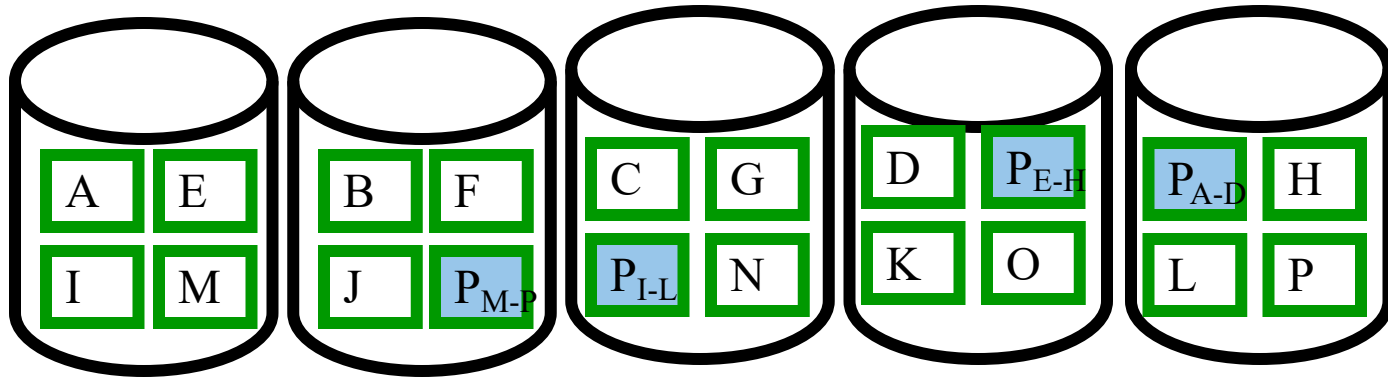
Flaschenhals bildet die Paritätsplatte

Bei jedem Schreiben muss darauf zugegriffen werden

- Bei Modifikation von Block A zu A' wird die Parität  $P_{A-D}$  wie folgt neu berechnet:
- $P'_{A-D} := P_{A-D} \oplus A \oplus A'$

D.h. bei einer Änderung von Block A muss der alte Zustand von A und der alte Paritätsblock gelesen werden und der neue Paritätsblock und der neue Block A' geschrieben werden

# RAID 5: Striping von Blöcken, Verteilung der Paritätsblöcke



Bessere Lastbalancierung als bei RAID 4

die Paritätsplatte bildet jetzt keinen Flaschenhals mehr

Wird in der Praxis häufig eingesetzt

Guter Ausgleich zwischen Platzbedarf und Leistungsfähigkeit

# Bewertung der Parallelität bei RAID

RAID 0

- ?

RAID 1

- ?

RAID 0+1

- ?

RAID 3

- ?

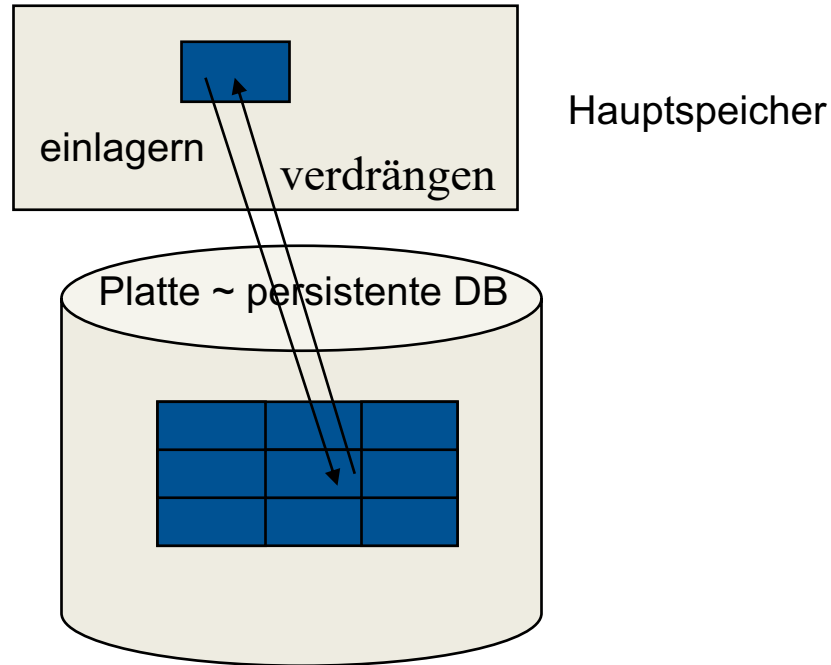
RAID 4

- ?

RAID 5

- ?

# Systempuffer-Verwaltung

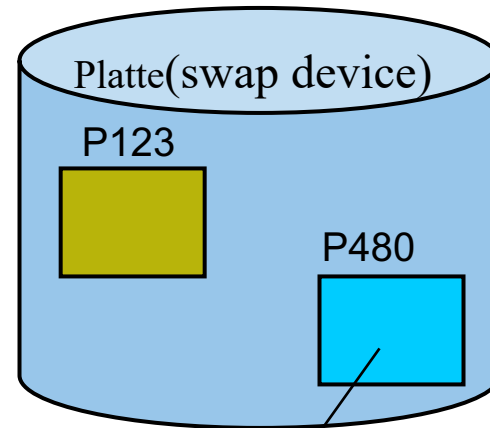


# Ein- und Auslagern von Seiten

- Systempuffer ist in Seitenrahmen gleicher Größe aufgeteilt
- Ein Rahmen kann eine Seite aufnehmen
- „Überzählige“ Seiten werden auf die Platte ausgelagert

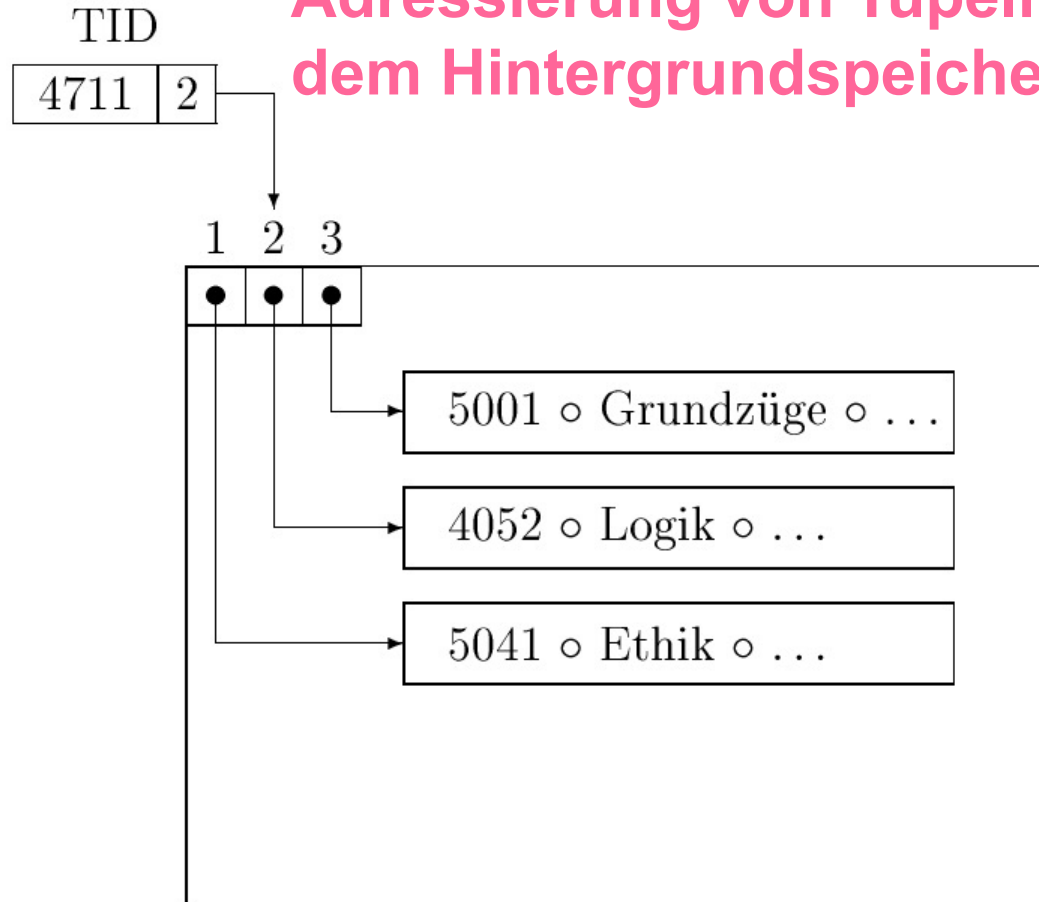
0	4K	8K	12K
16K	20K	24K	28K
32K	36K	40K	44K
48K	52K	56K	60K

Seitenrahmen



Seite

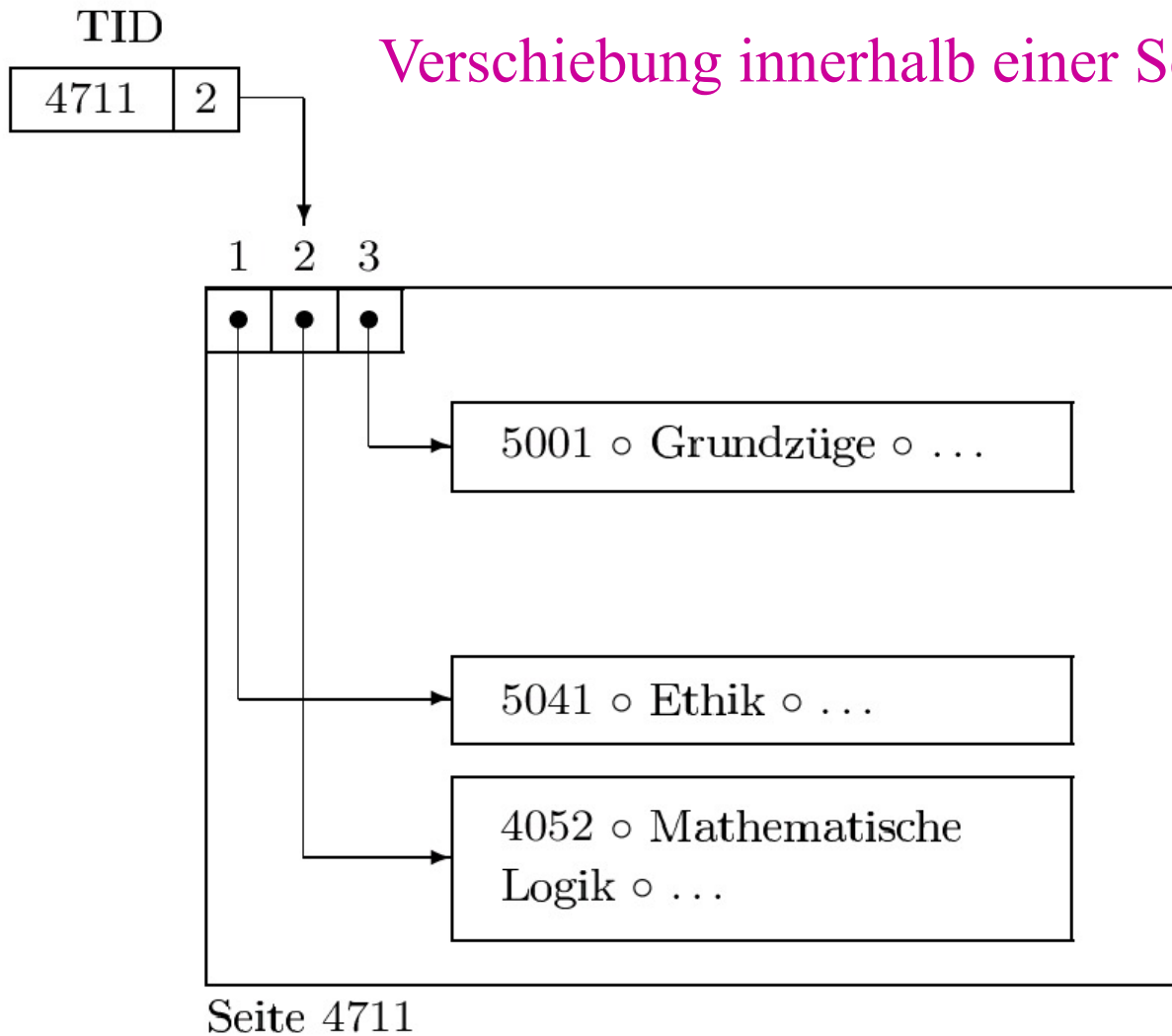
# Adressierung von Tupeln auf dem Hintergrundspeicher



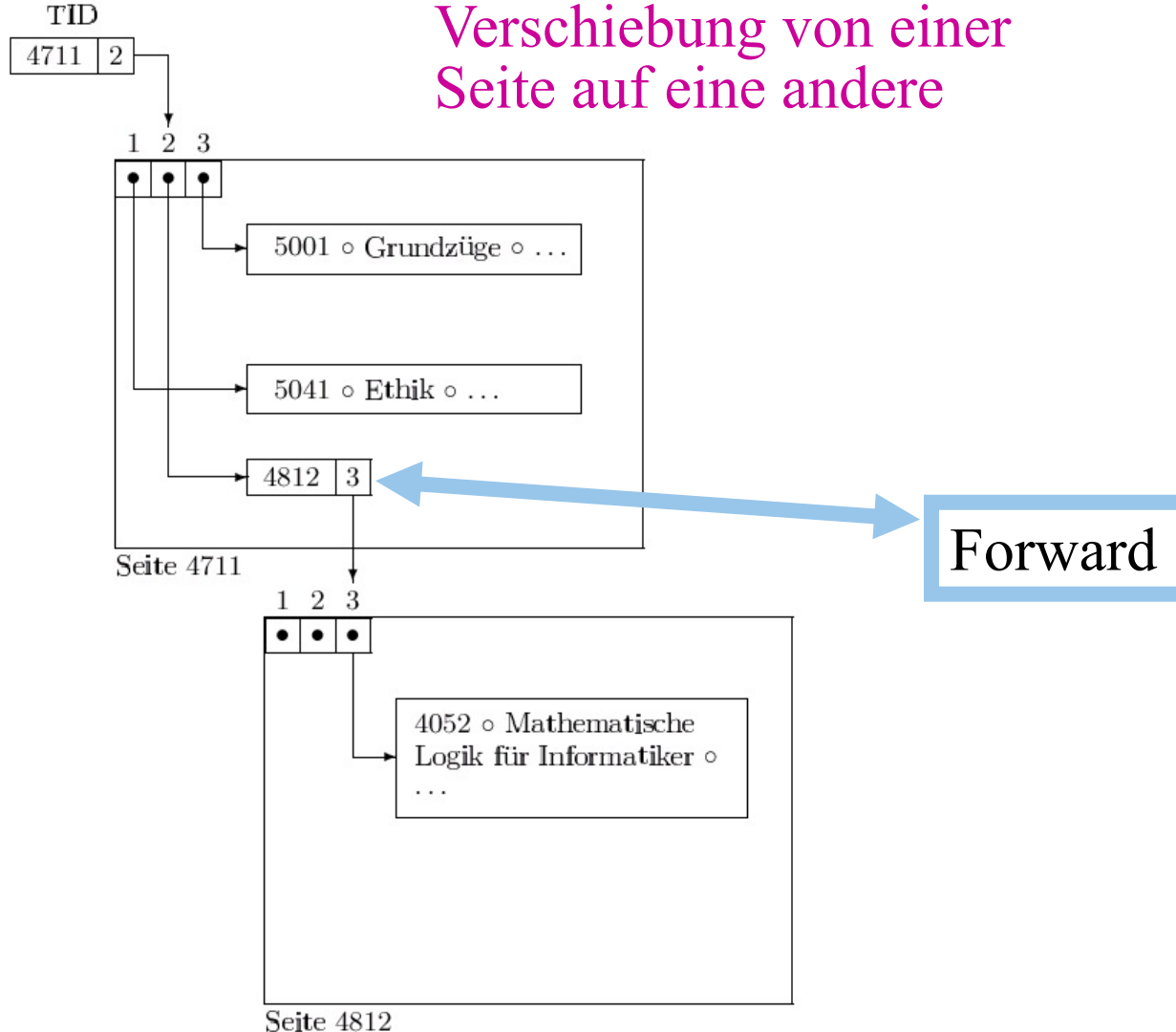
Seite 4711



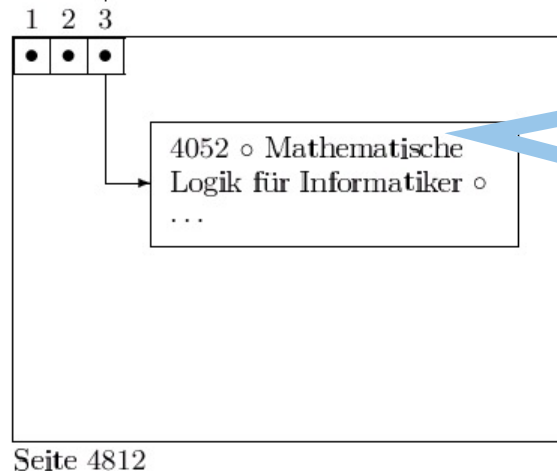
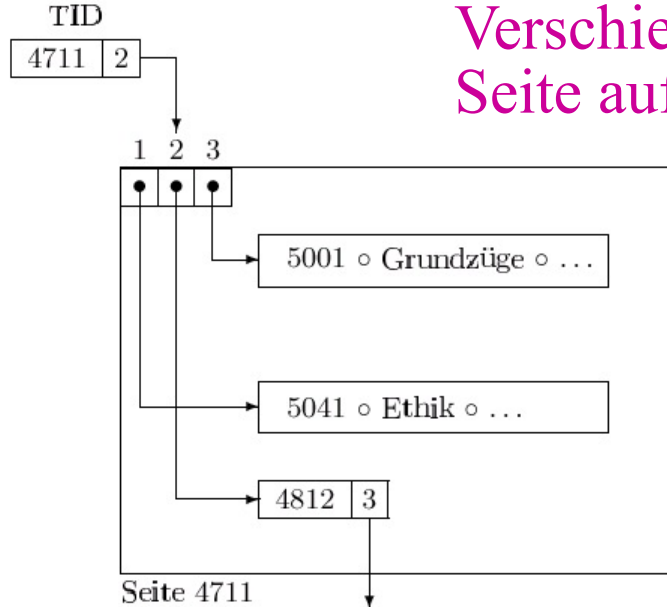
## Verschiebung innerhalb einer Seite



# Verschiebung von einer Seite auf eine andere



# Verschiebung von einer Seite auf eine andere



Bei der nächsten Verschiebung wird der „Forward“ auf Seite 4711 geändert (kein Forward auf Seite 4812)

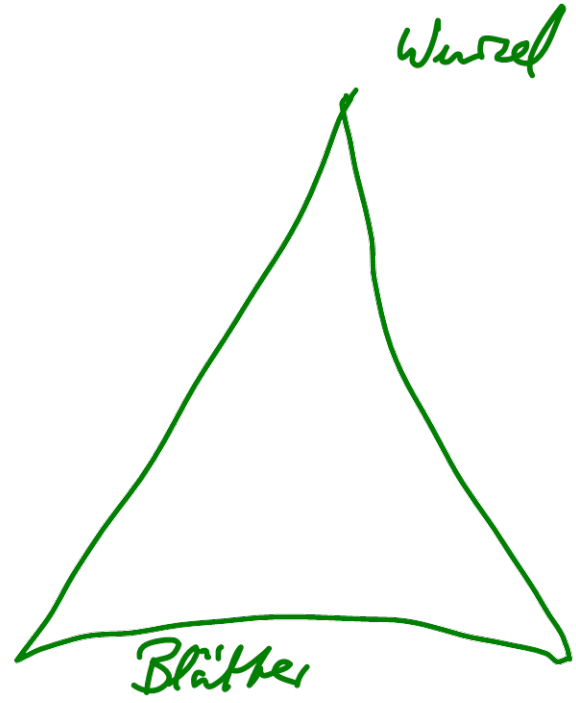
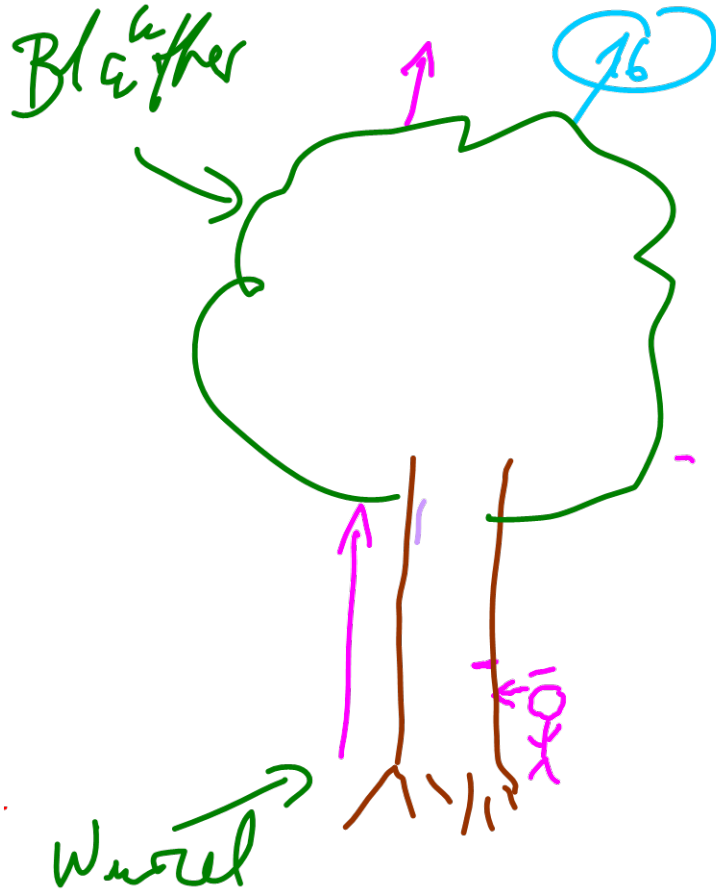
# B-Bäume

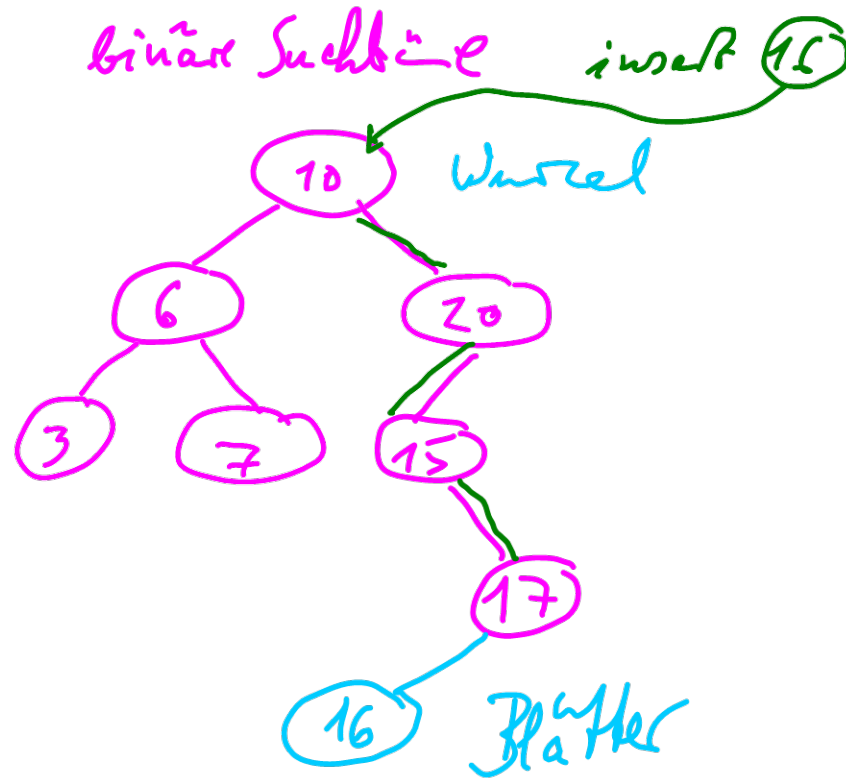
---

**Balancierte Mehrwege-Suchbäume**

**Für den Hintergrundspeicher**

**Erfunden von Rudolph Bayer (TUM Professor Emeritus)**



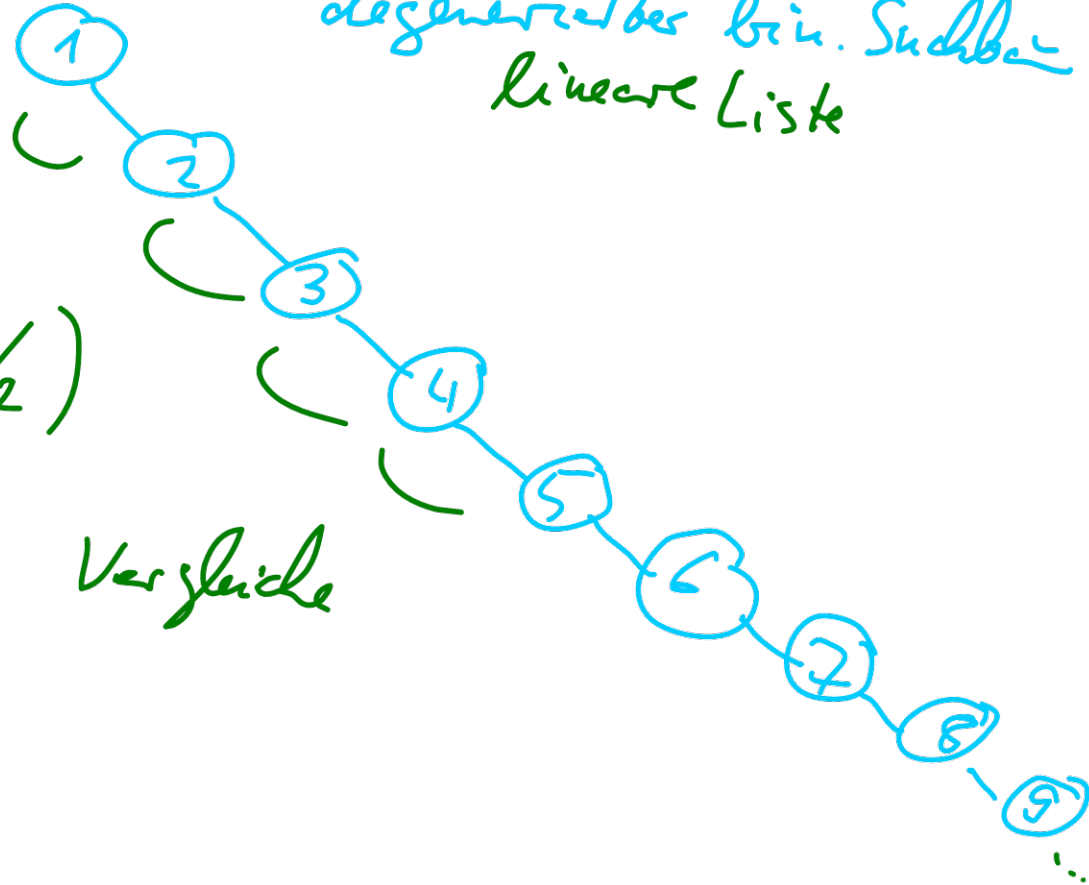


1 2 3 4 5 6 7 8 9 ...  $\overset{\circ}{\circ}$  ...  $10^{10}$   
 ↑

degeneriertes bin. Suchbaum  
 lineare Liste

~~$\Theta(N/2)$~~

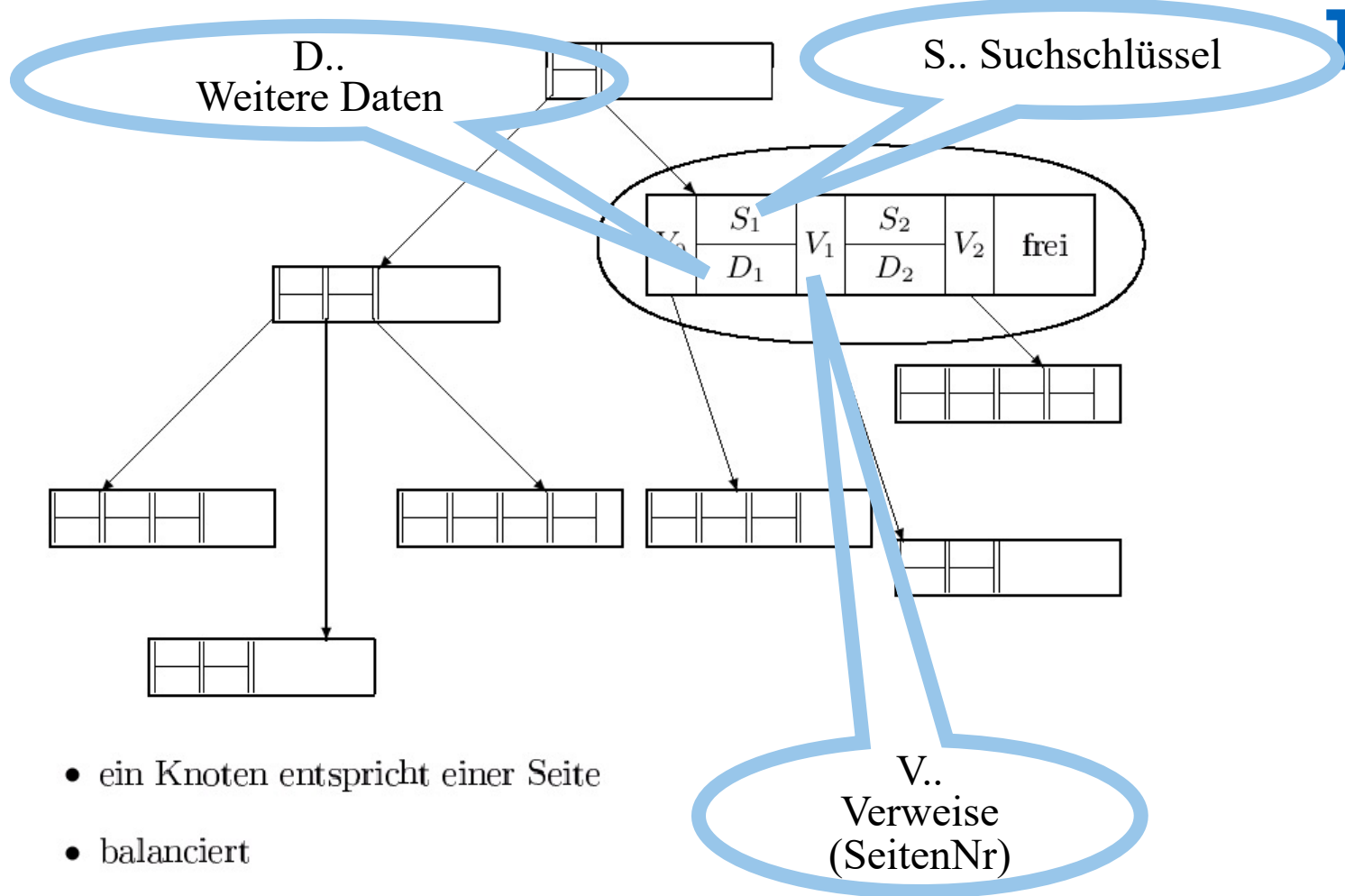
$10^{10}/2$  Vergleiche



# (Handschriftliche) Demo über Binäre Suchbäume





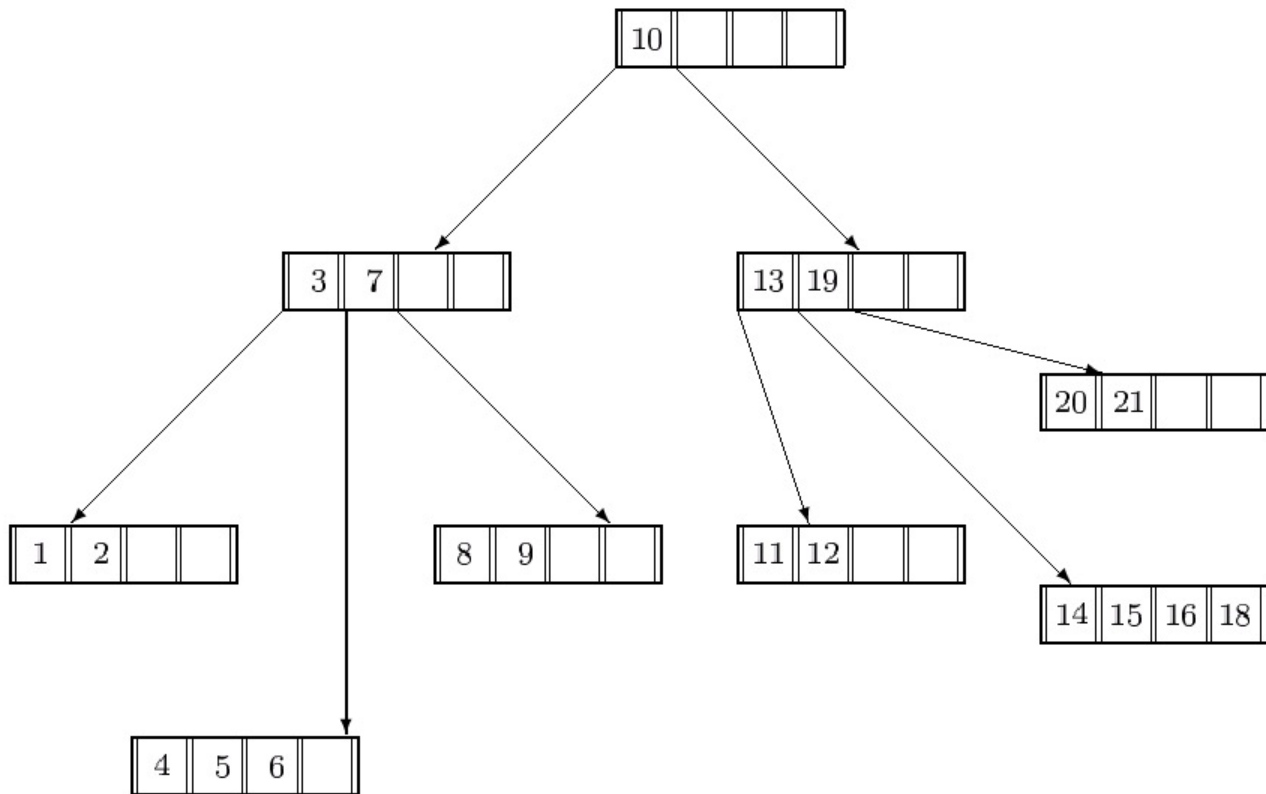


- ein Knoten entspricht einer Seite
- balanciert
- garantierte Auslastung  $\geq 50\%$

B-Baum von Grad  $k$ :

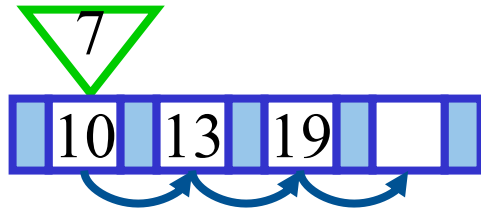
1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge.
2. Jeder Knoten außer der Wurzel hat mindestens  $k$  und höchstens  $2k$  Einträge. Die Wurzel hat höchstens  $2k$  Einträge. Die Einträge werden in allen Knoten sortiert gehalten.
3. Alle Knoten mit  $n$  Einträgen, außer den Blättern, haben  $n + 1$  Kinder.
4. Seien  $S_1, \dots, S_n$  die Schlüssel eines Knotens mit  $n + 1$  Kindern.  $V_0, V_1, \dots, V_n$  seien die Verweise auf diese Kinder. Dann gilt:
  - (a)  $V_0$  weist auf den Teilbaum mit Schlüsseln kleiner als  $S_1$ .
  - (b)  $V_i$  ( $i = 1, \dots, n - 1$ ) weist auf den Teilbaum, dessen Schlüssel zwischen  $S_i$  und  $S_{i+1}$  liegen.
  - (c)  $V_n$  weist auf den Teilbaum mit Schlüsseln größer als  $S_n$ .
  - (d) In den Blattknoten sind die Zeiger nicht definiert.

# Ein Beispielbaum

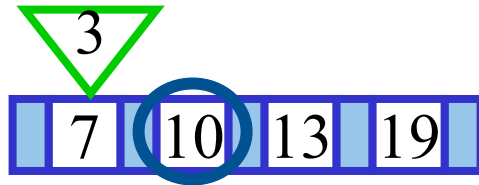


1. Führe eine Suche nach dem Schlüssel durch; diese endet (scheitert) an der Einfügestelle.
2. Füge den Schlüssel dort ein.
3. Ist der Knoten überfüllt, teile ihn
  - Lege einen neuen Knoten an und belege ihn mit den Schlüsseln, die rechts vom mittleren Eintrag des überfüllten Knotens liegen.
  - Füge den mittleren Eintrag im Vaterknoten des überfüllten Knotens ein.
  - Verbinde den Verweis rechts des neuen Eintrags im Vaterknoten mit dem neuen Knoten
4. Ist der Vaterknoten jetzt überfüllt?
  - Handelt es sich um die Wurzel, so lege eine neue Wurzel an.
  - Wiederhole Schritt 3 mit dem Vaterknoten.

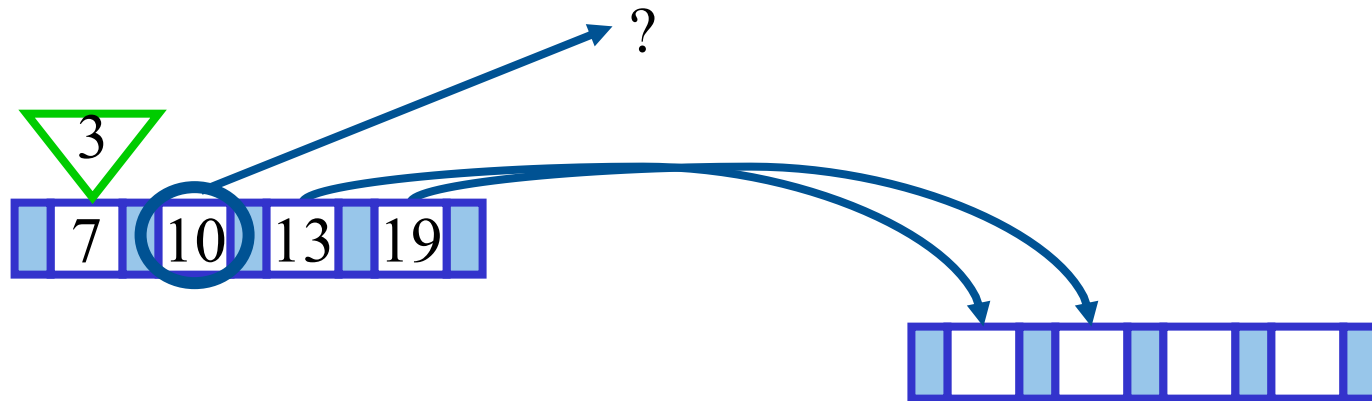
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

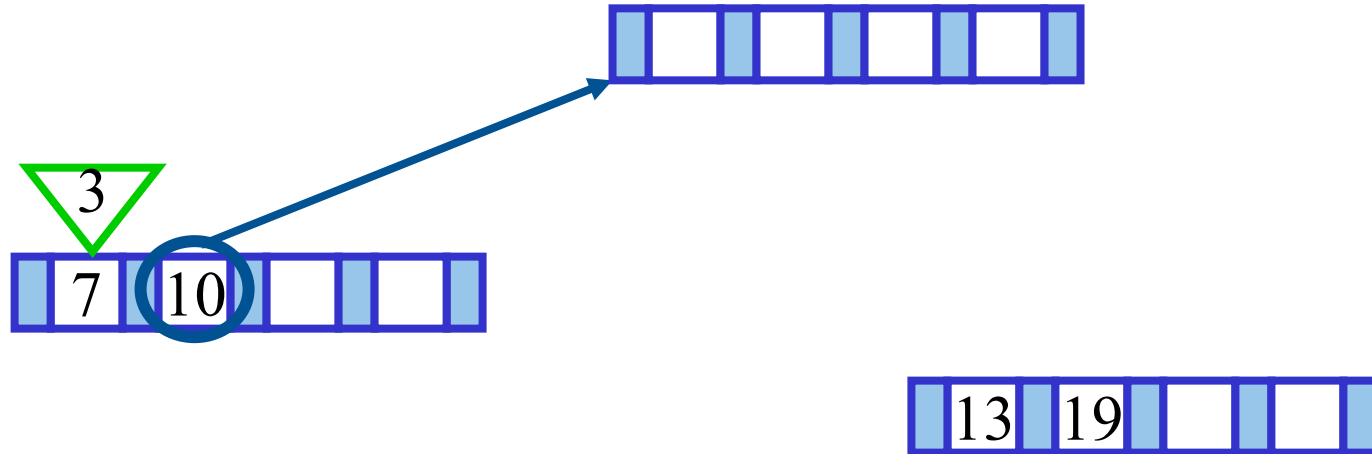


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

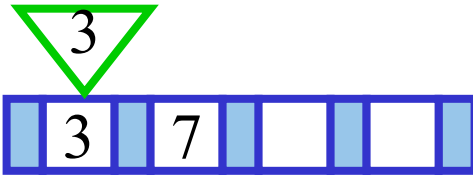




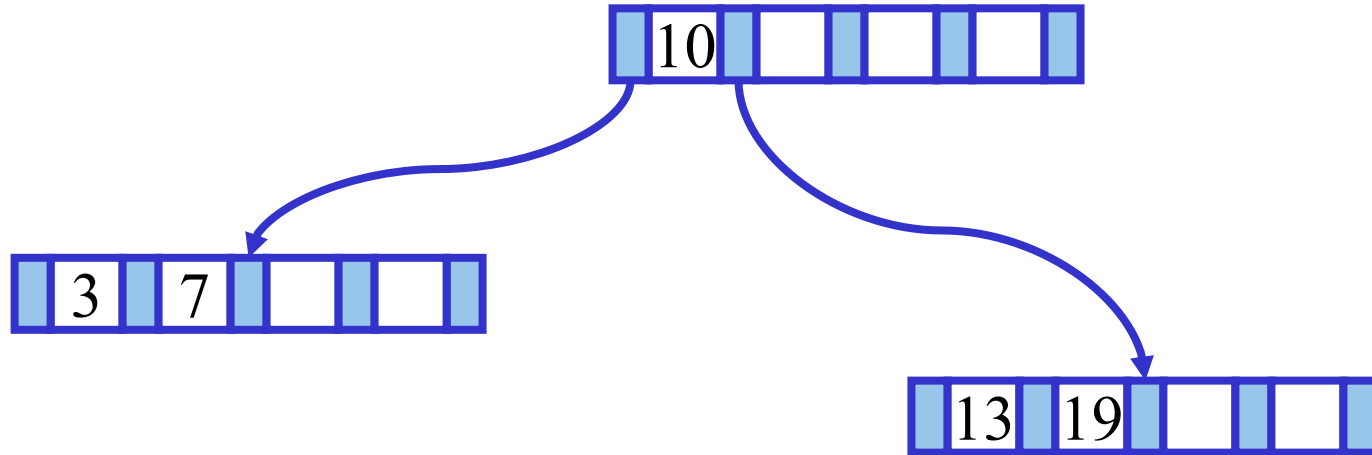
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



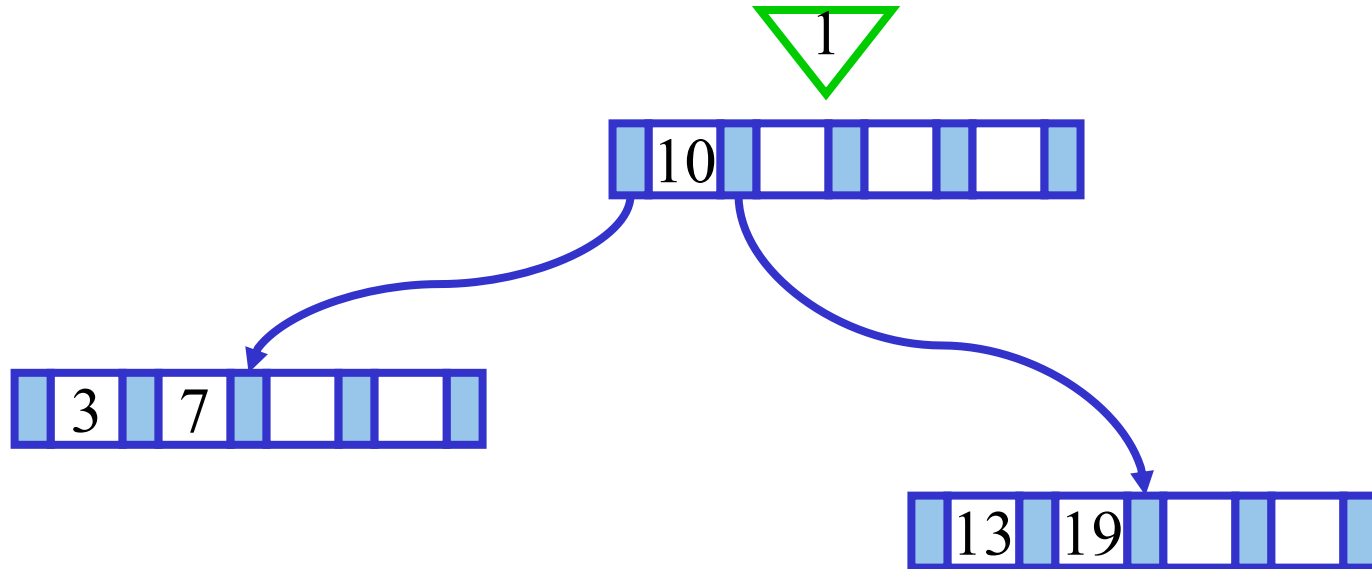
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



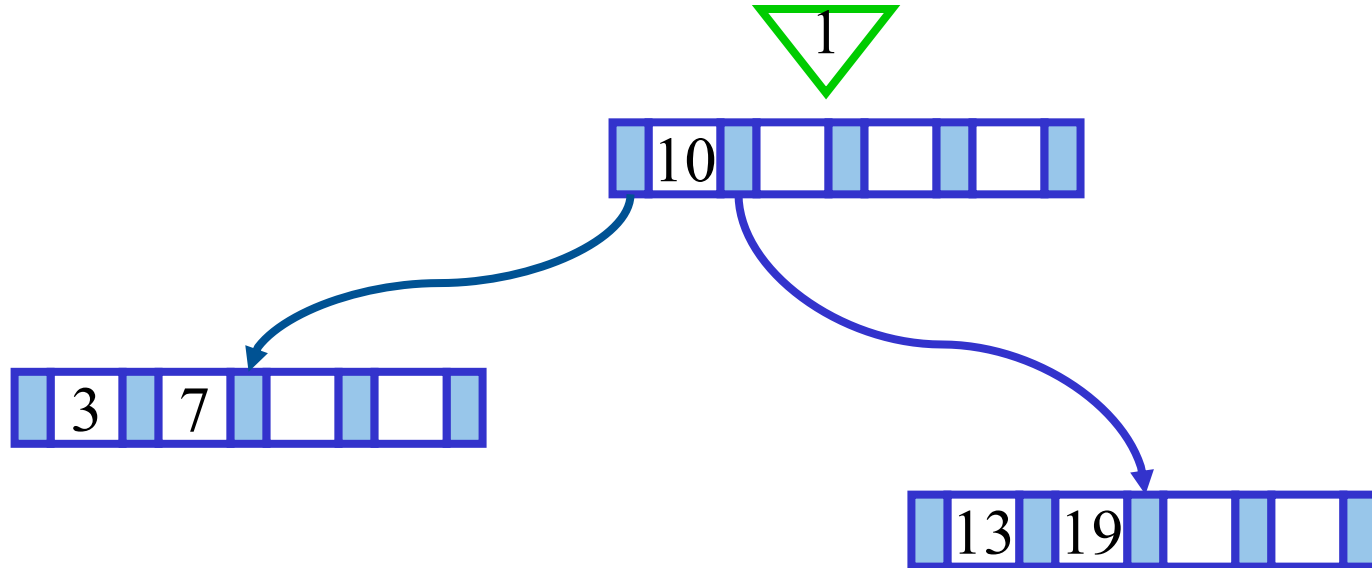
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



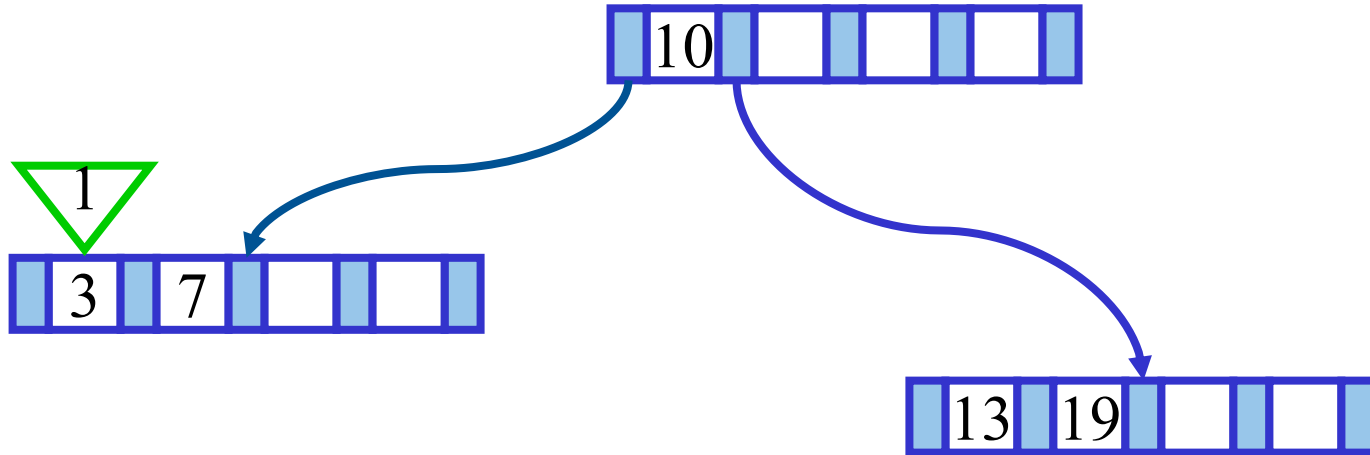
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



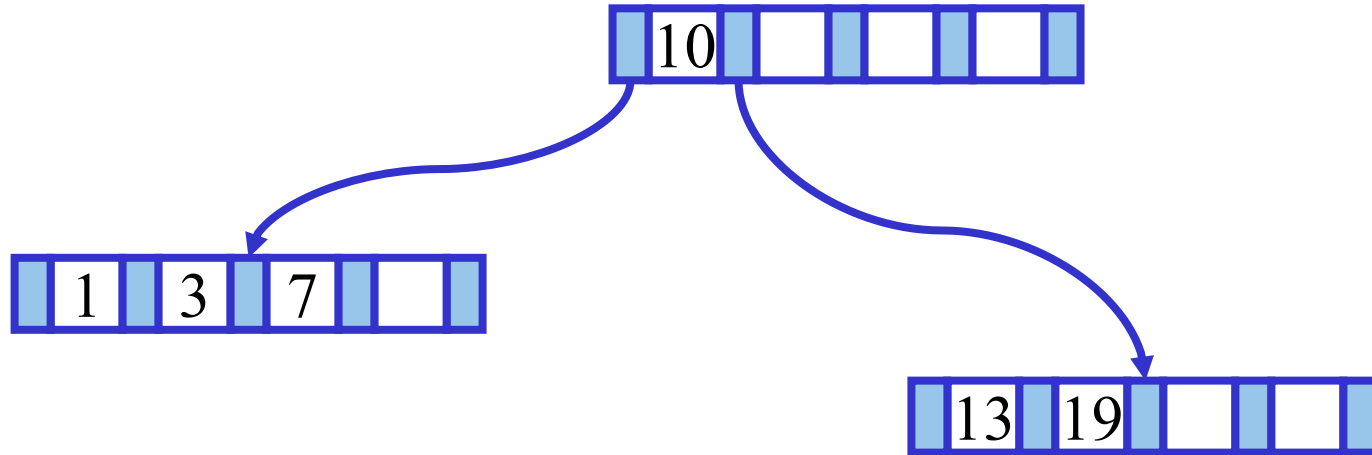
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



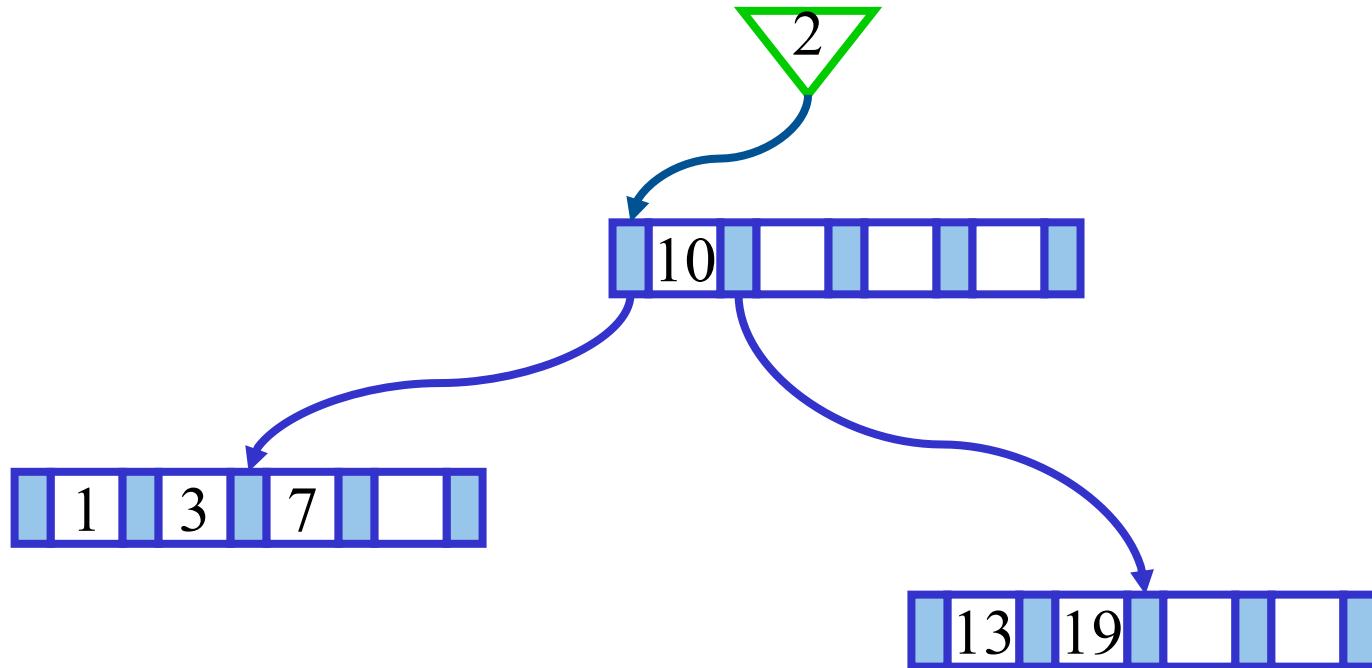
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

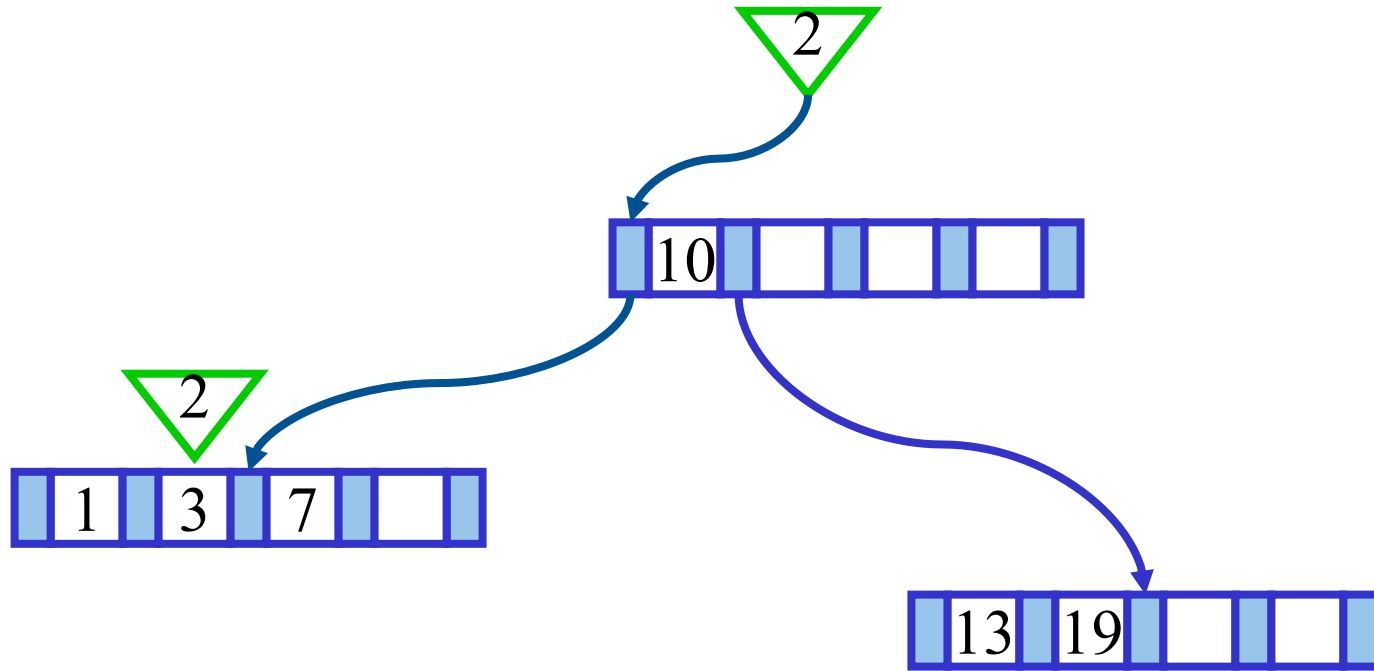


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

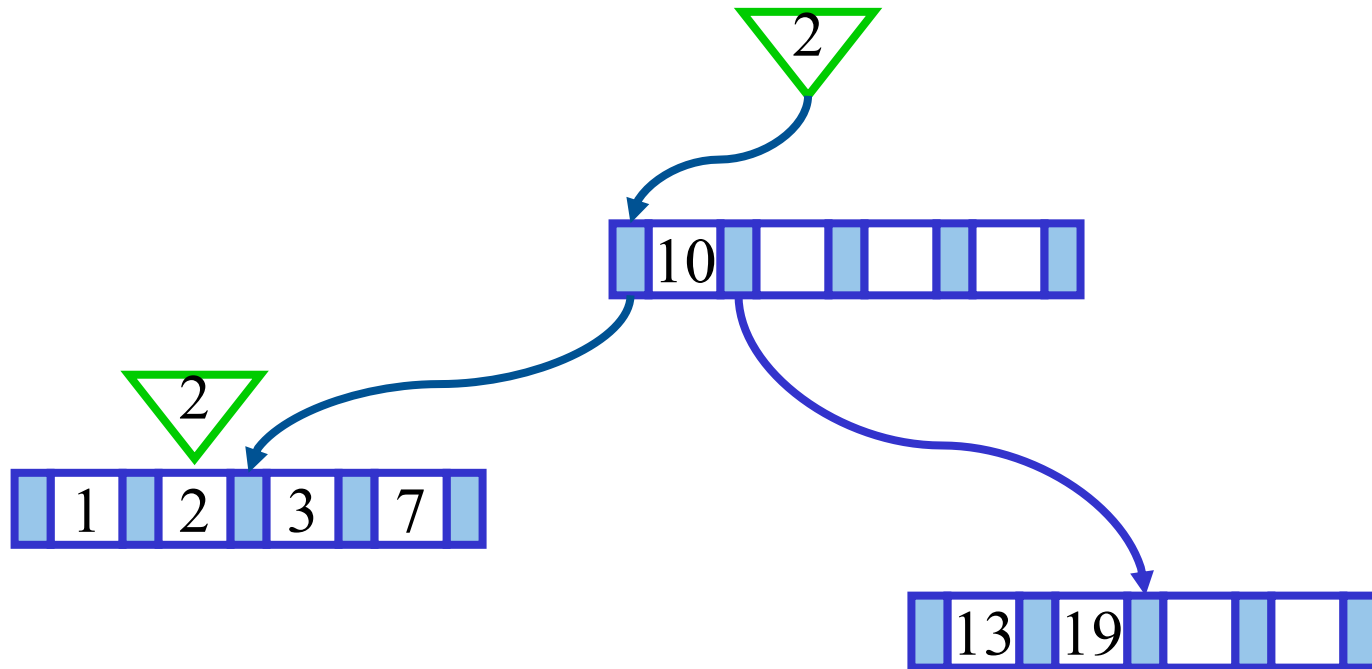




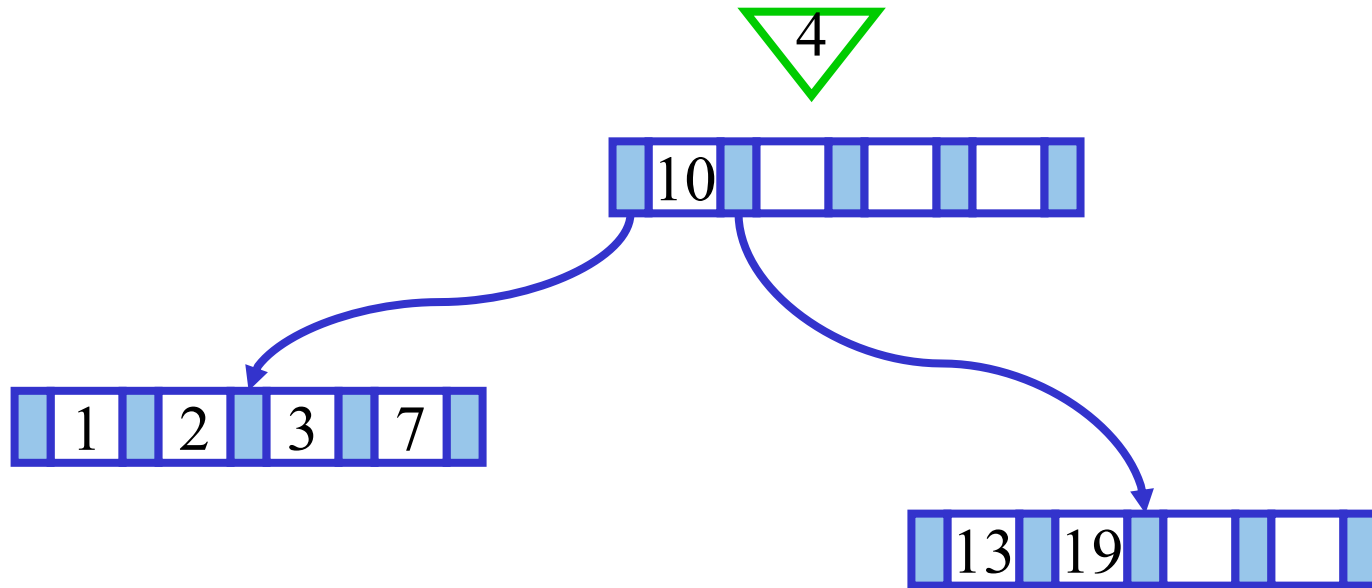
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



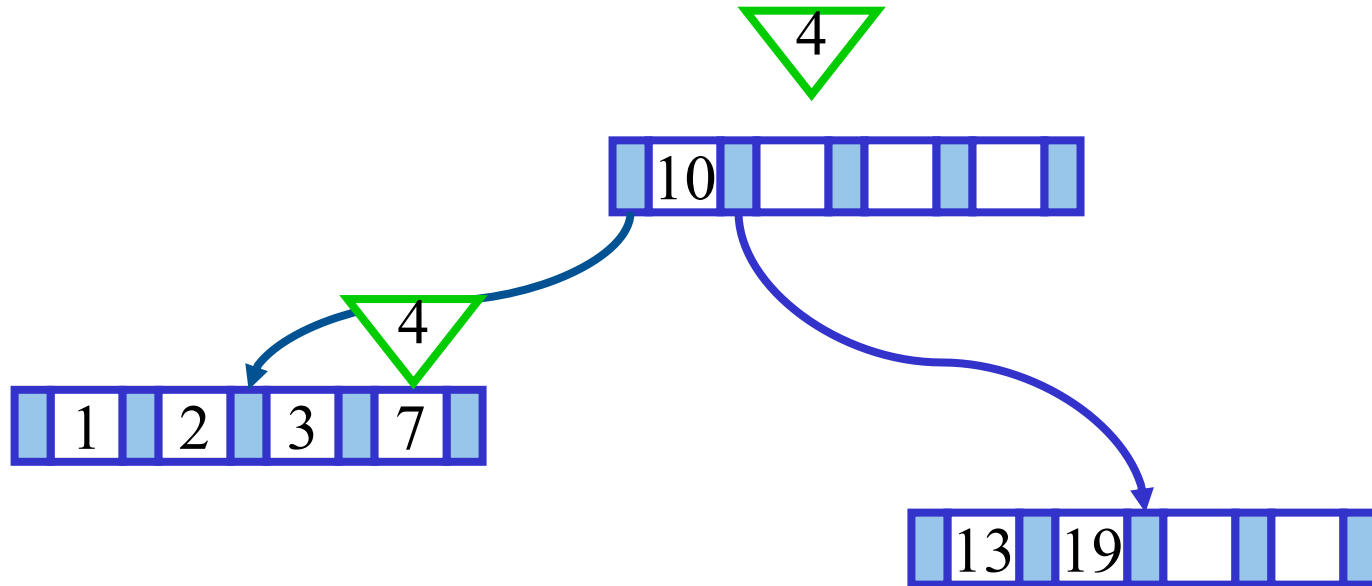
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

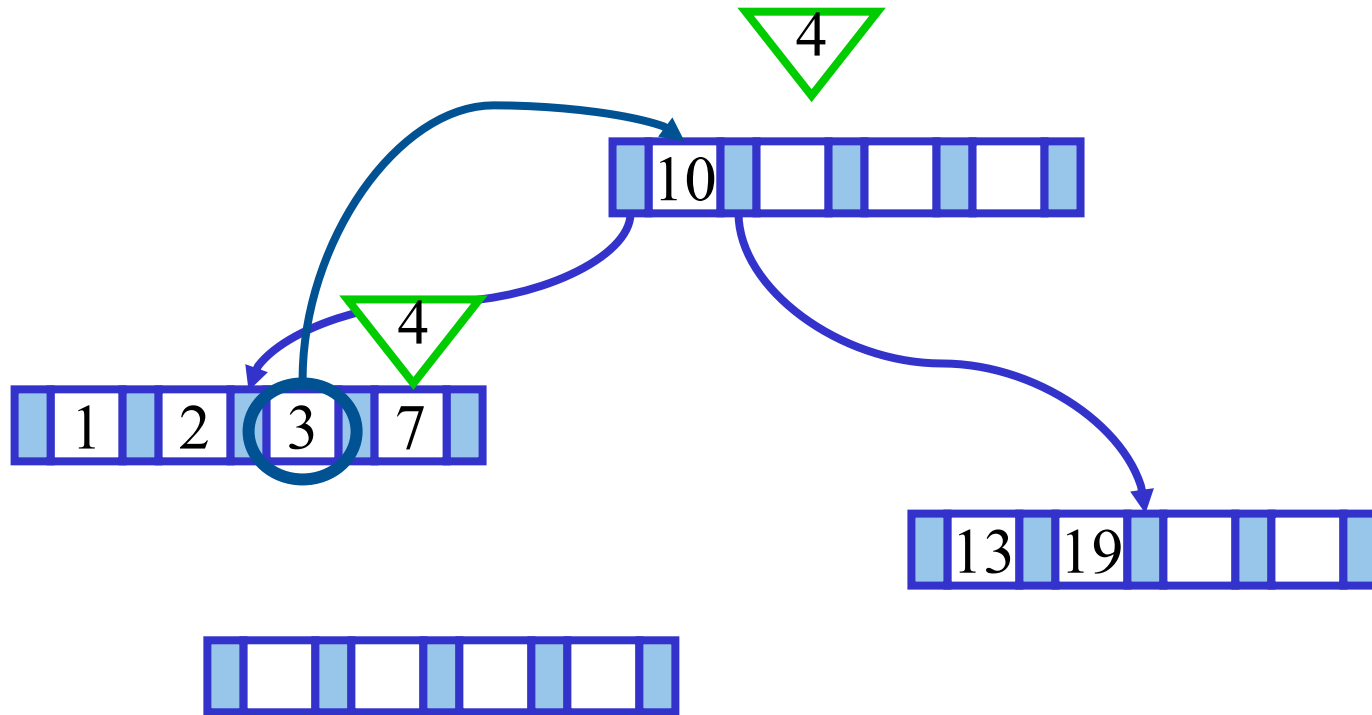


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

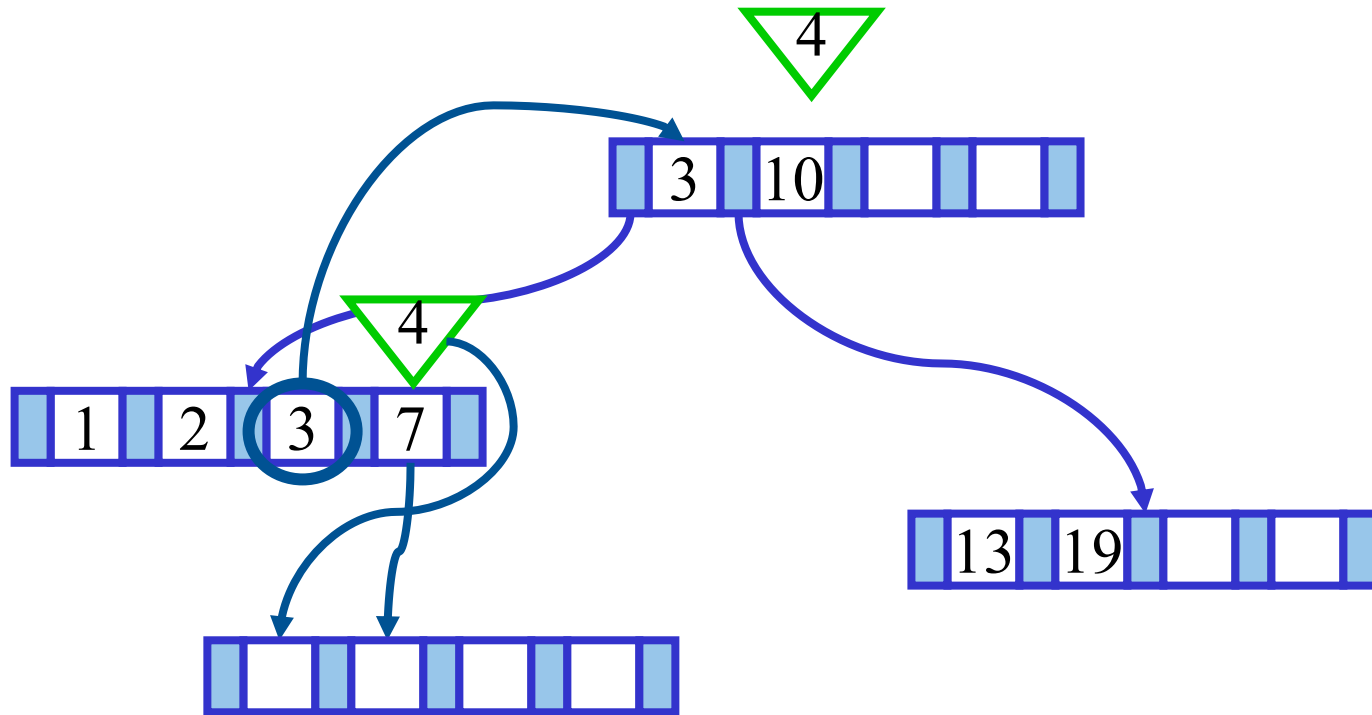




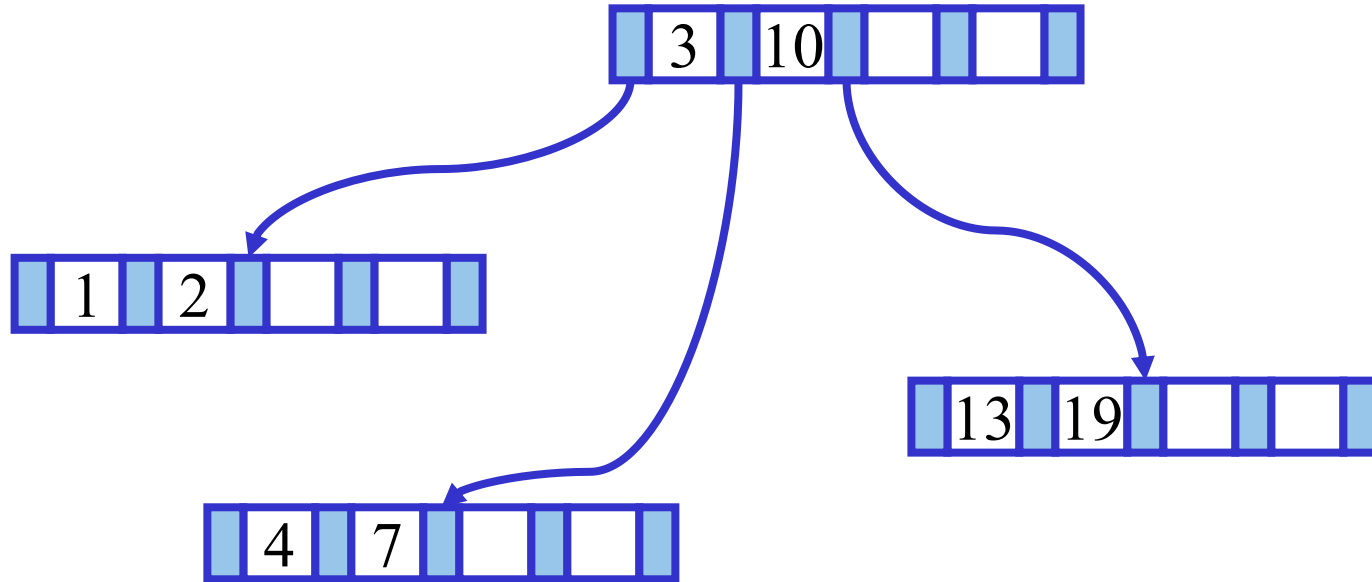
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

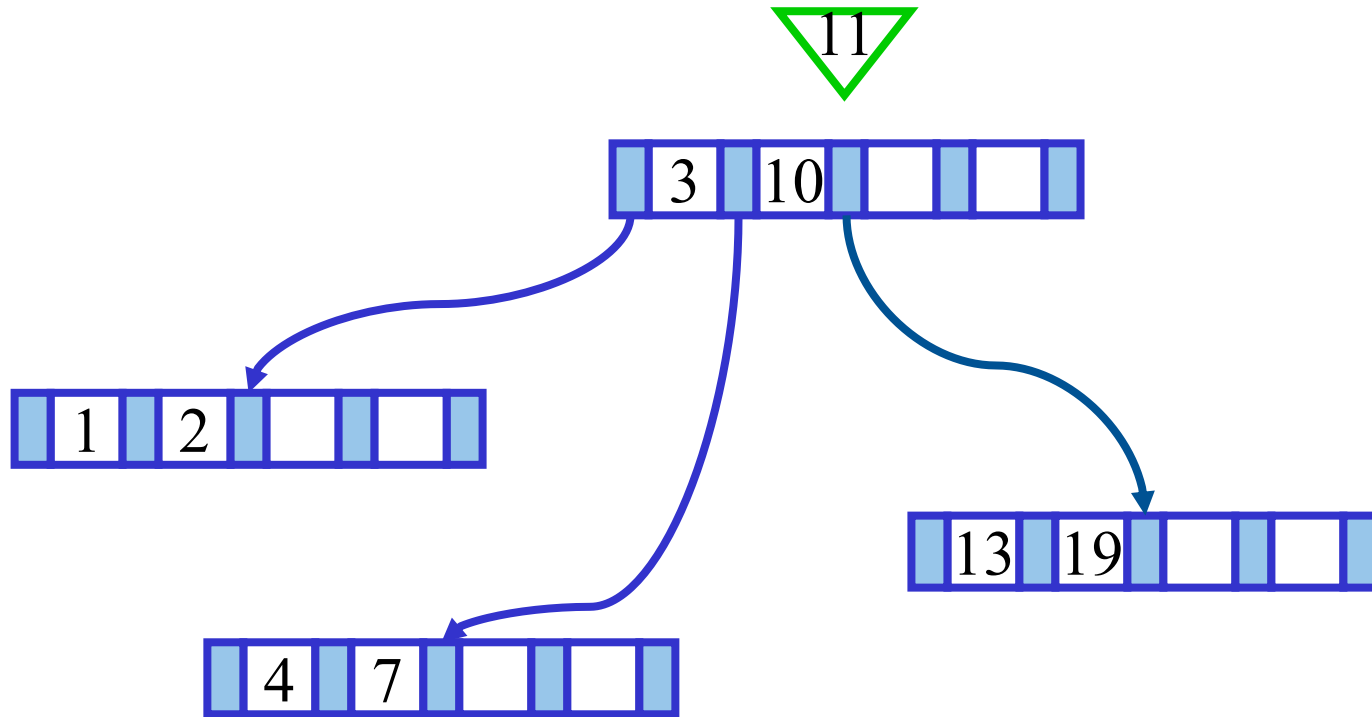


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

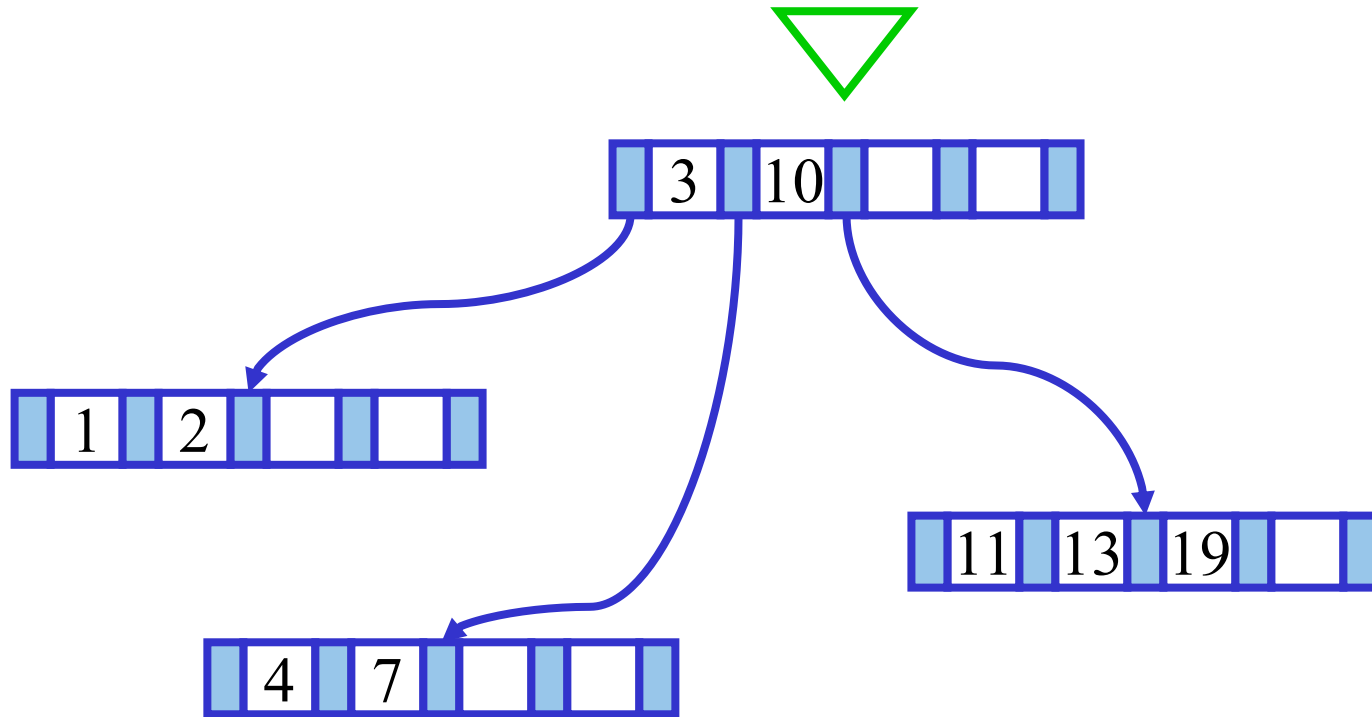




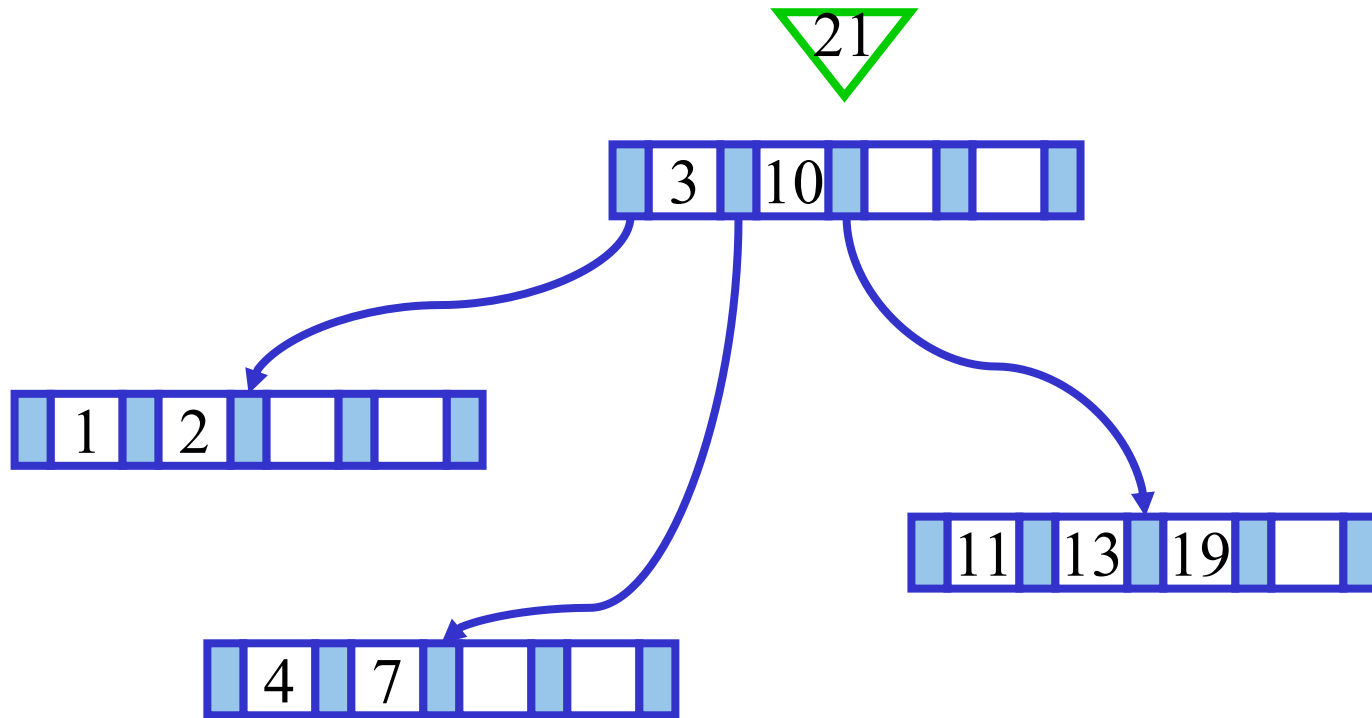
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



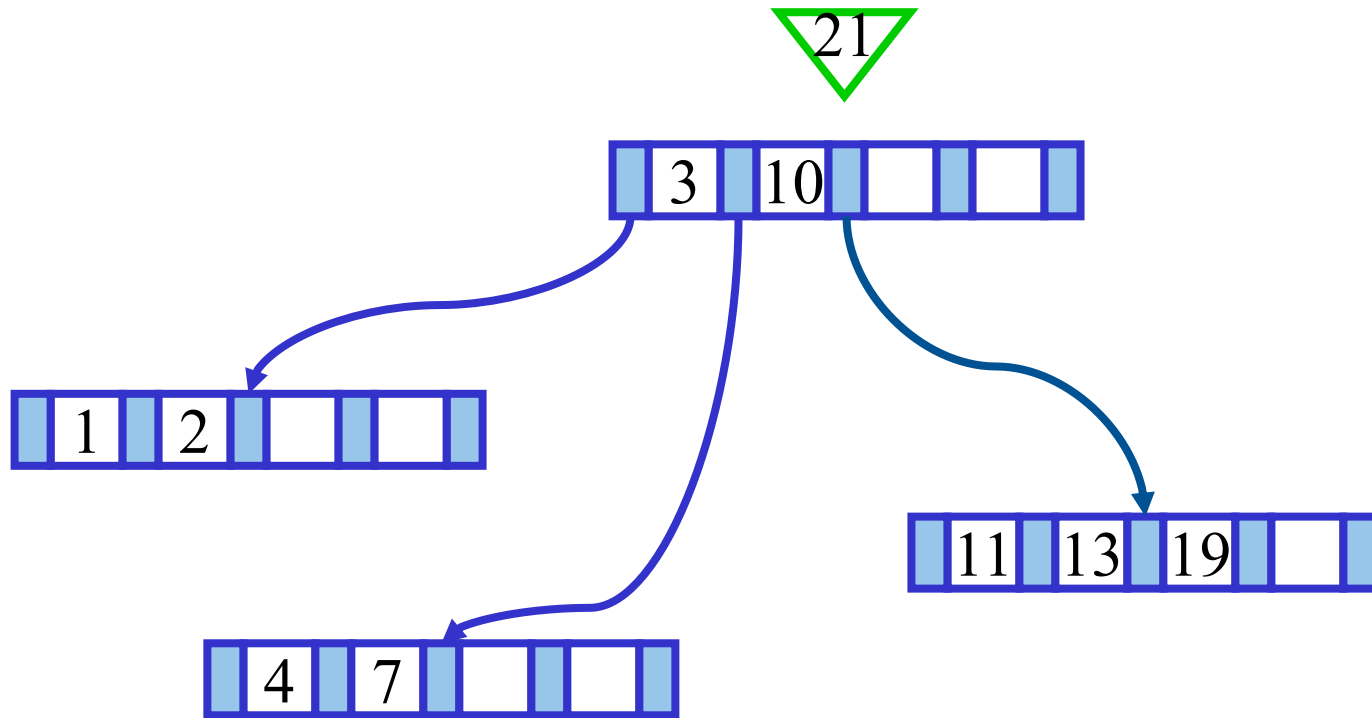
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



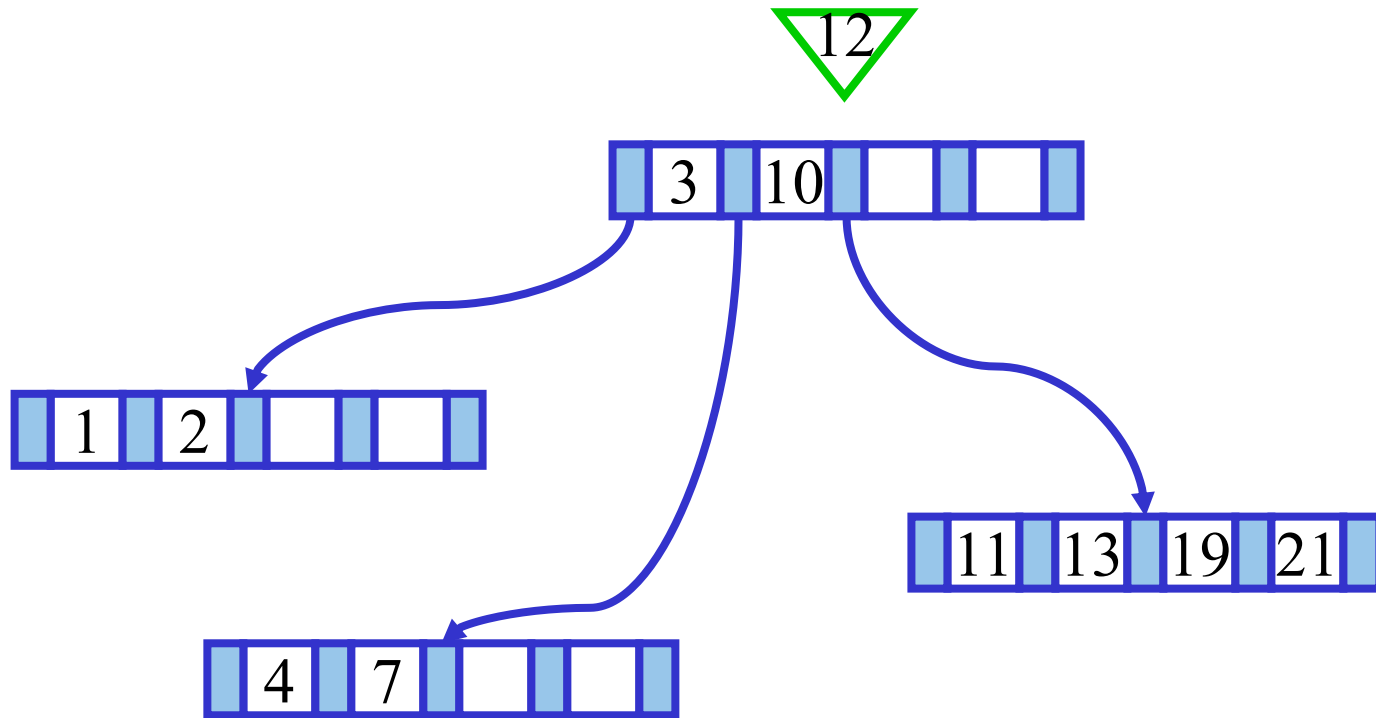
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



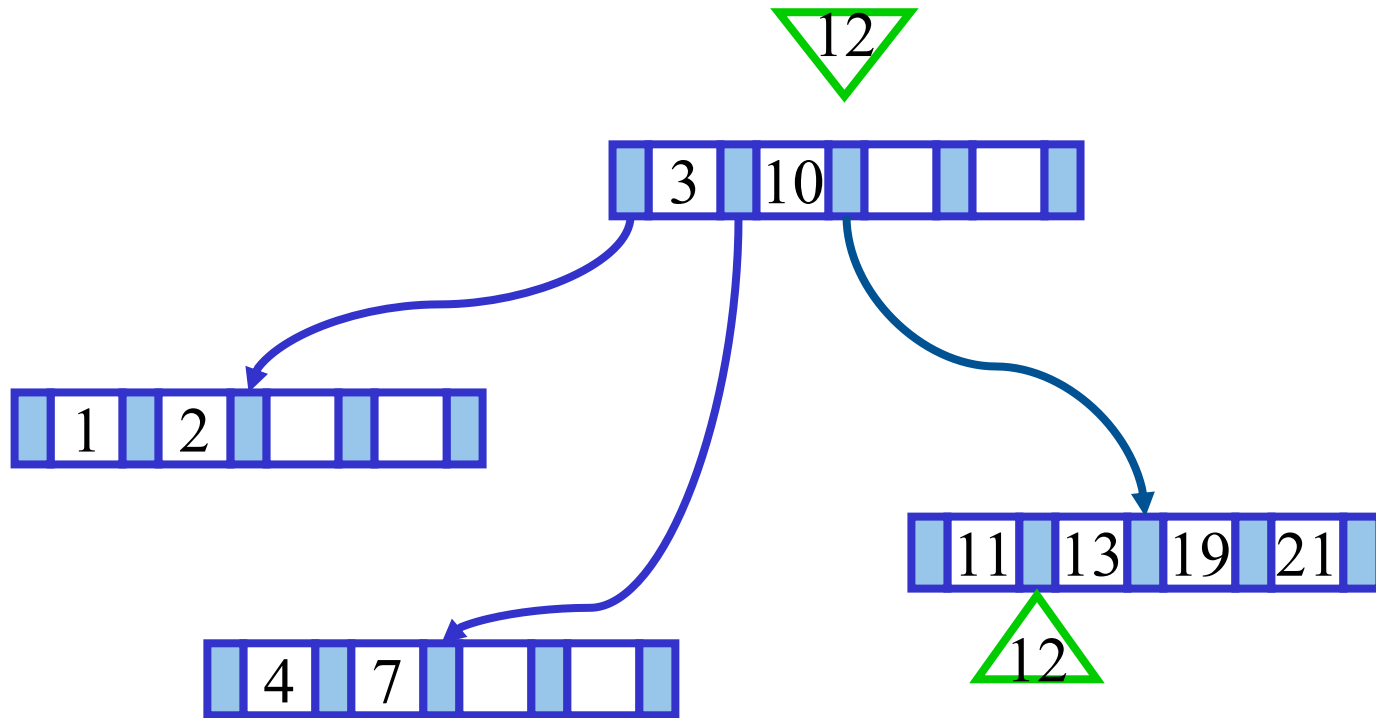
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



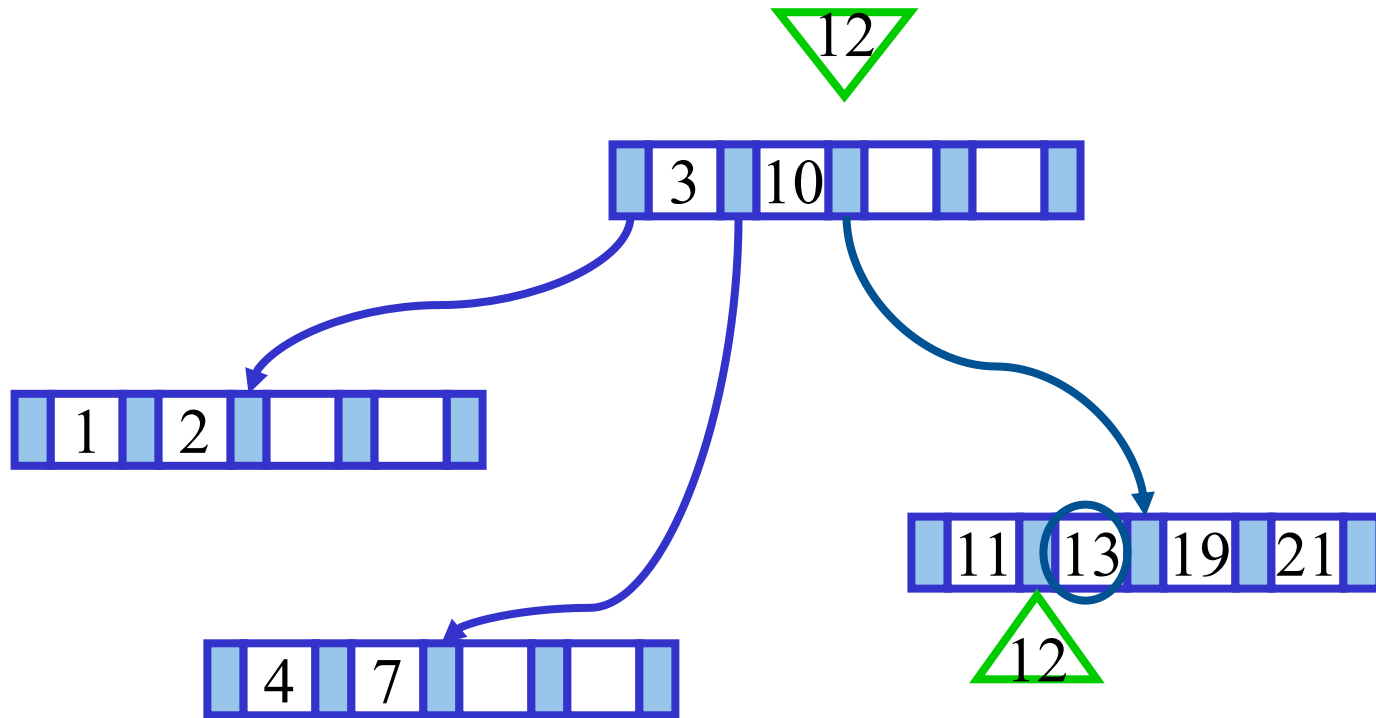
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



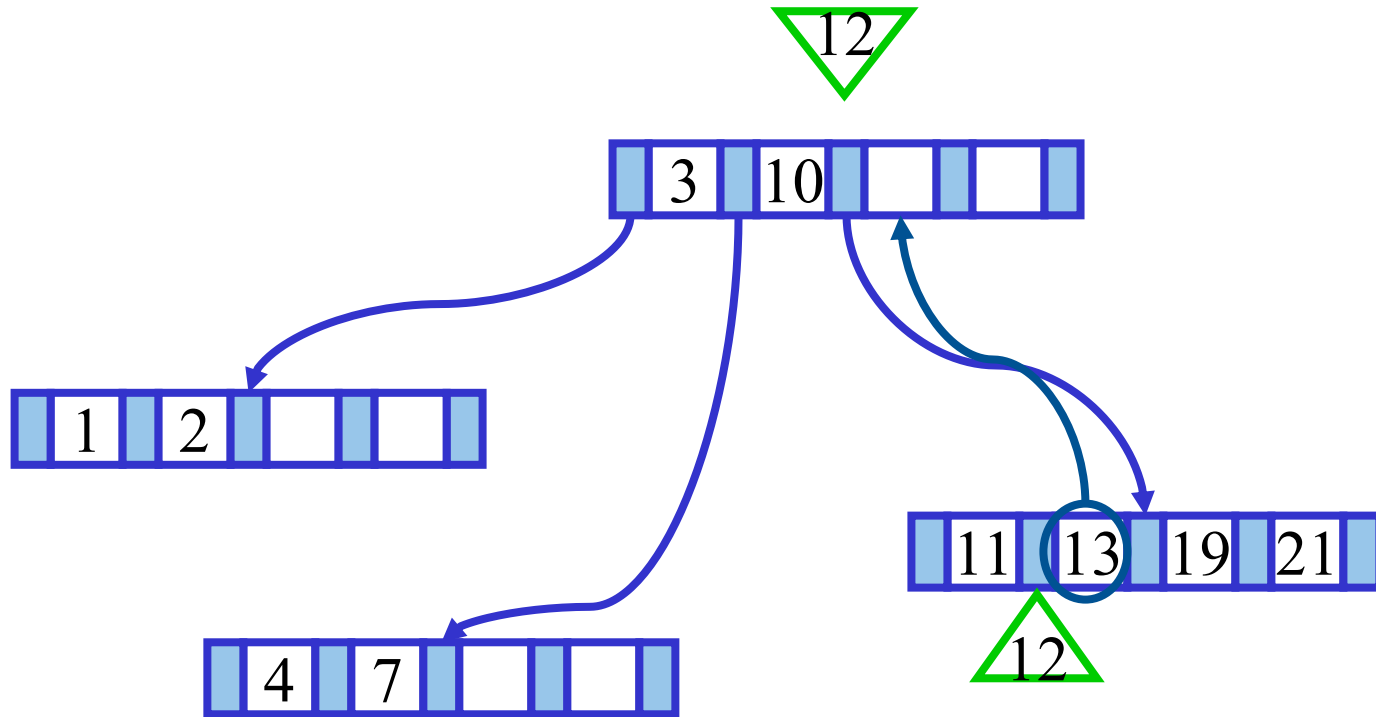
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

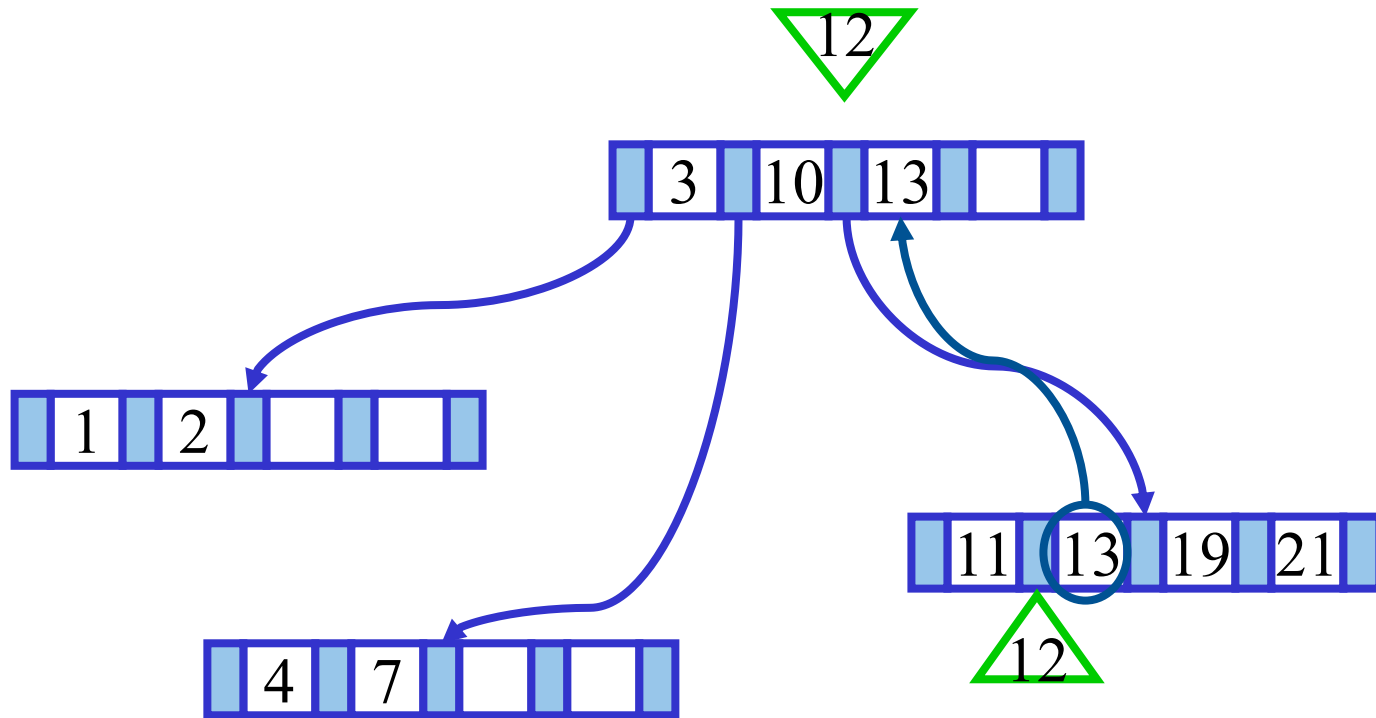


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

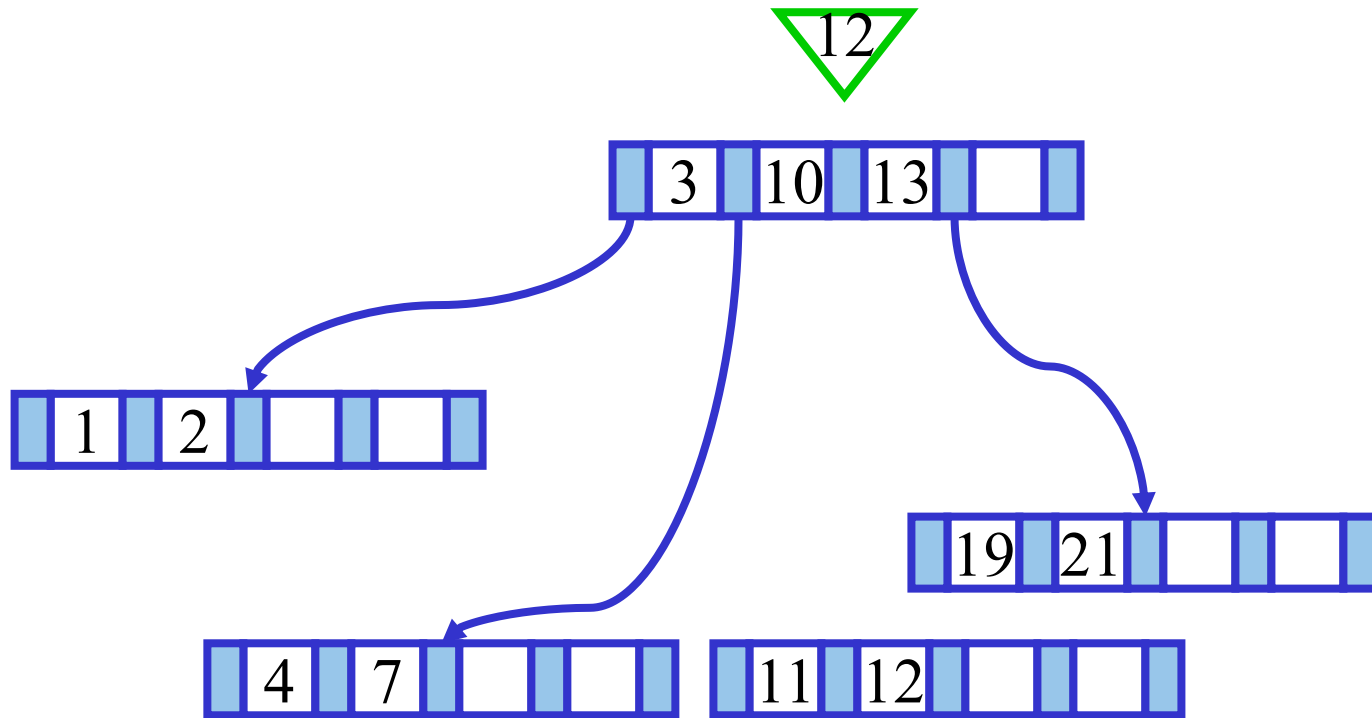




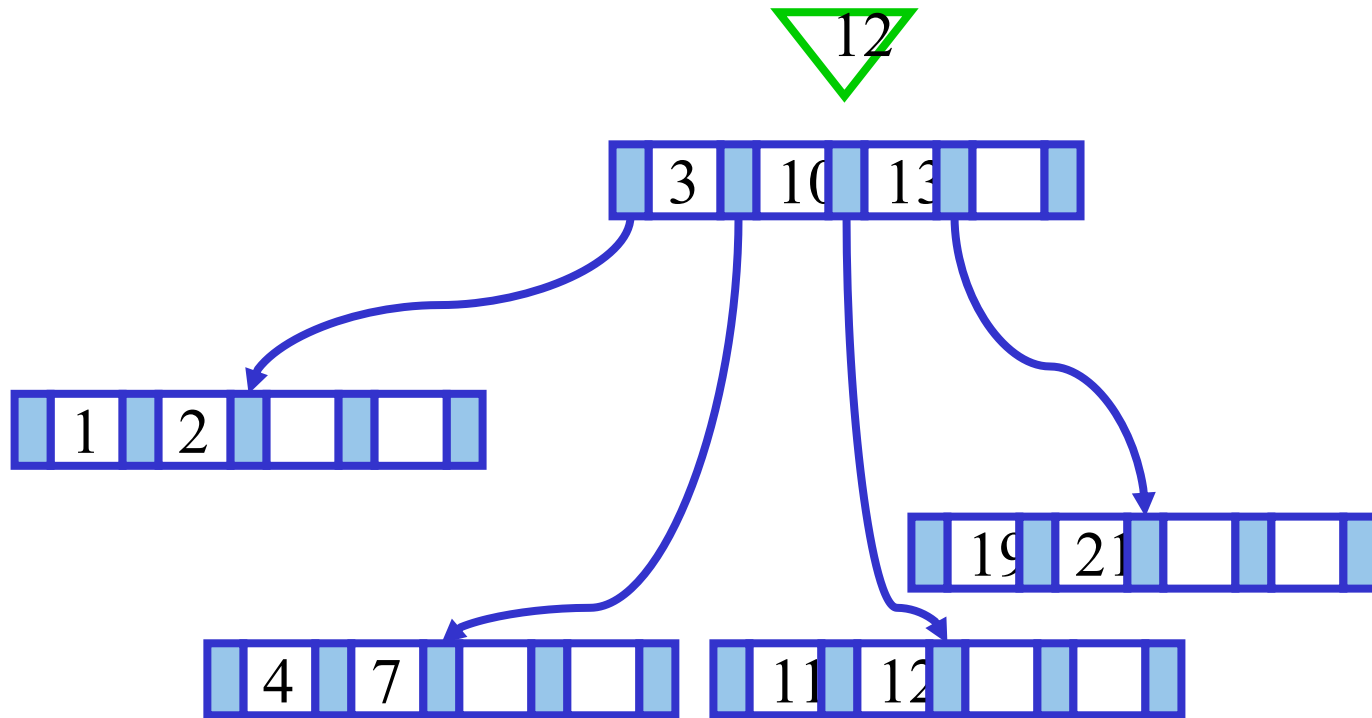
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



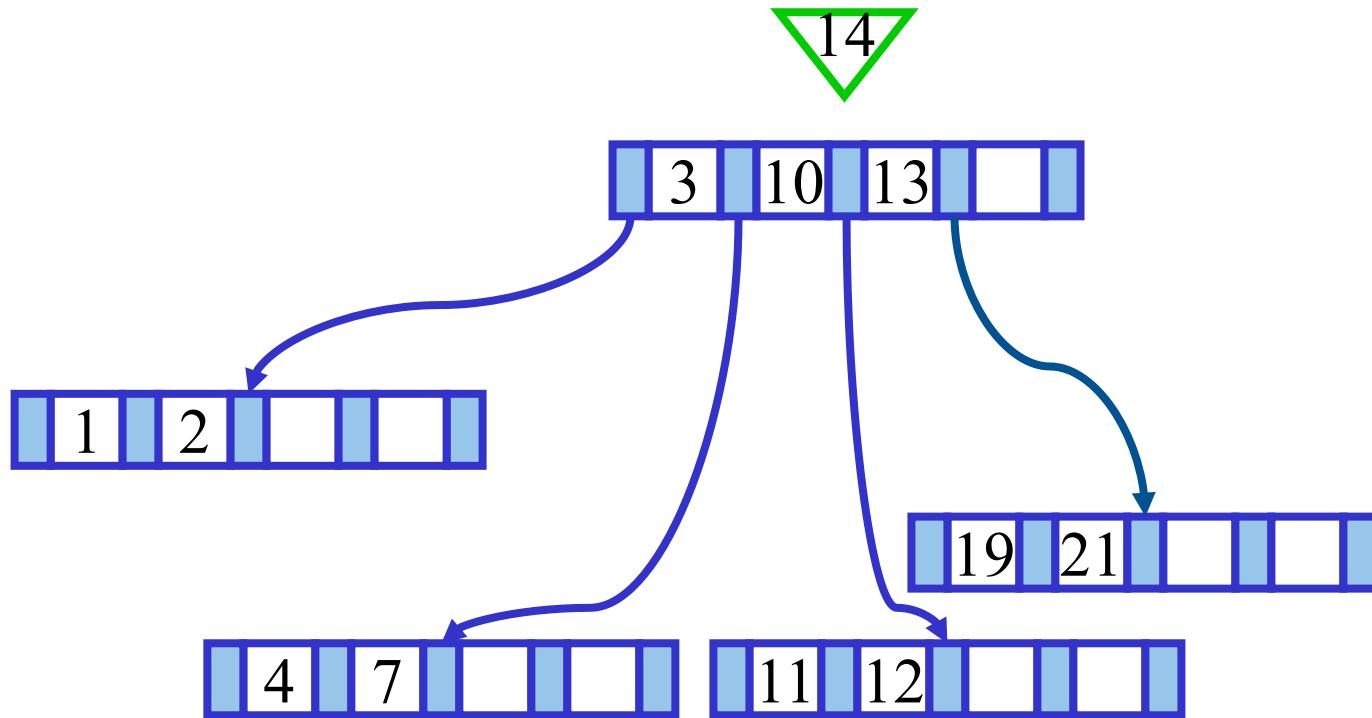
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



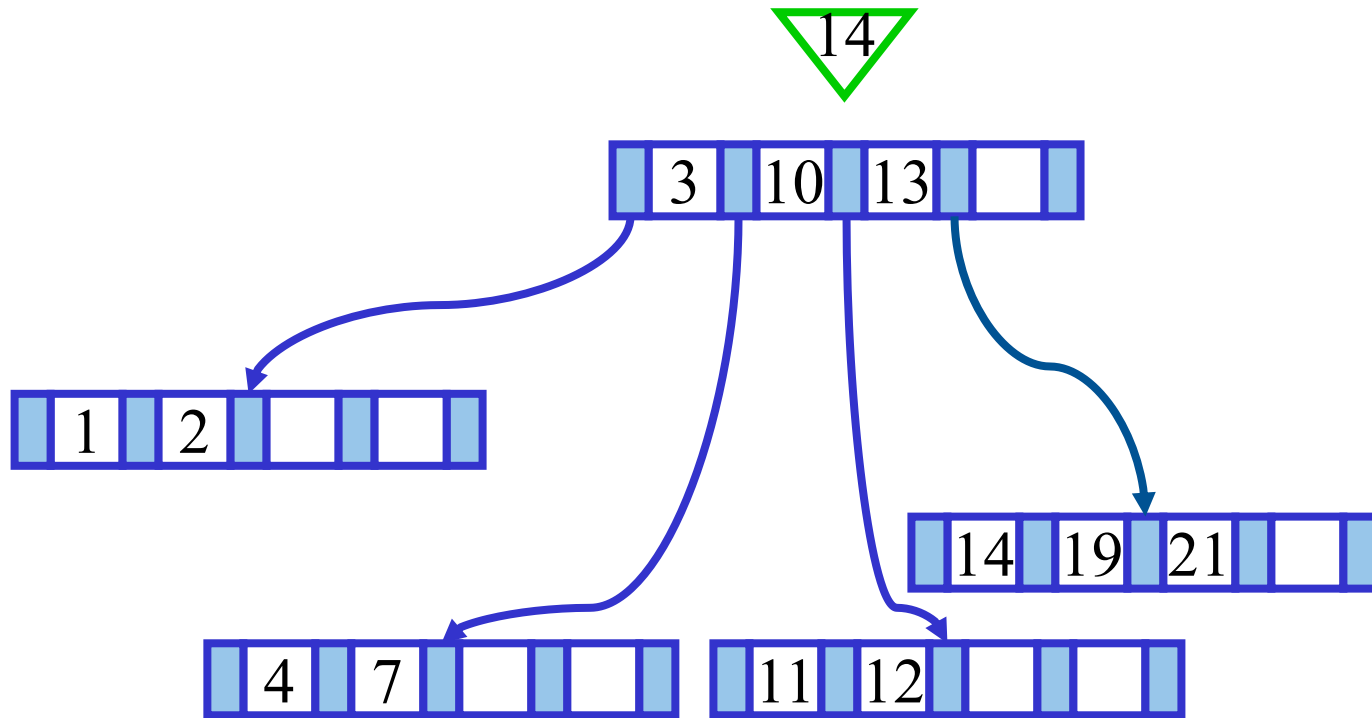
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



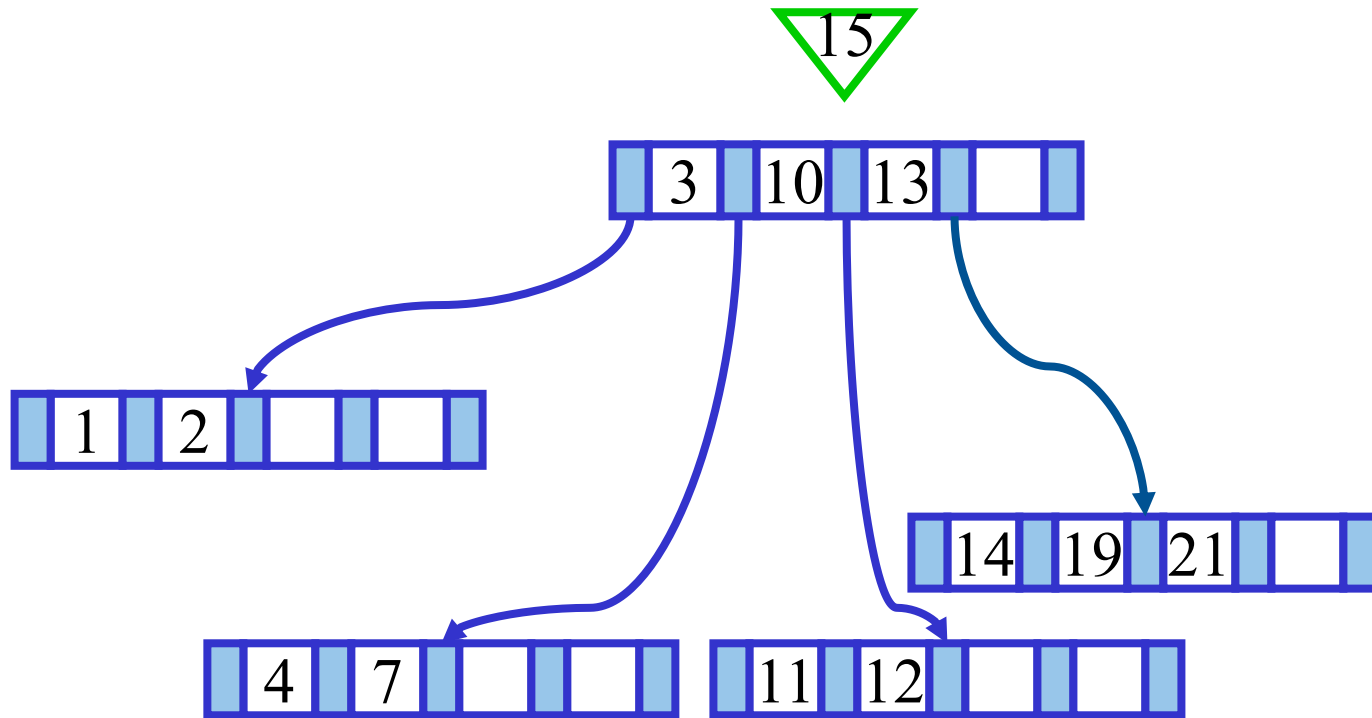
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



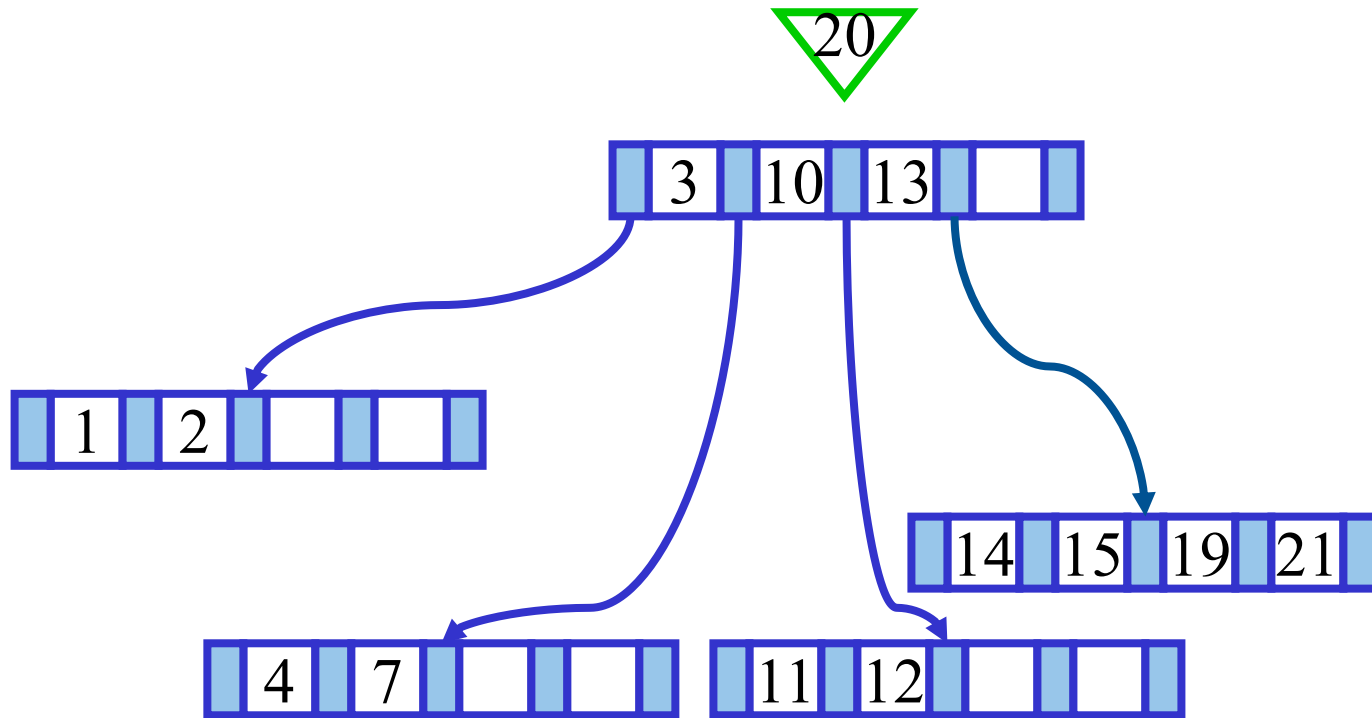
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



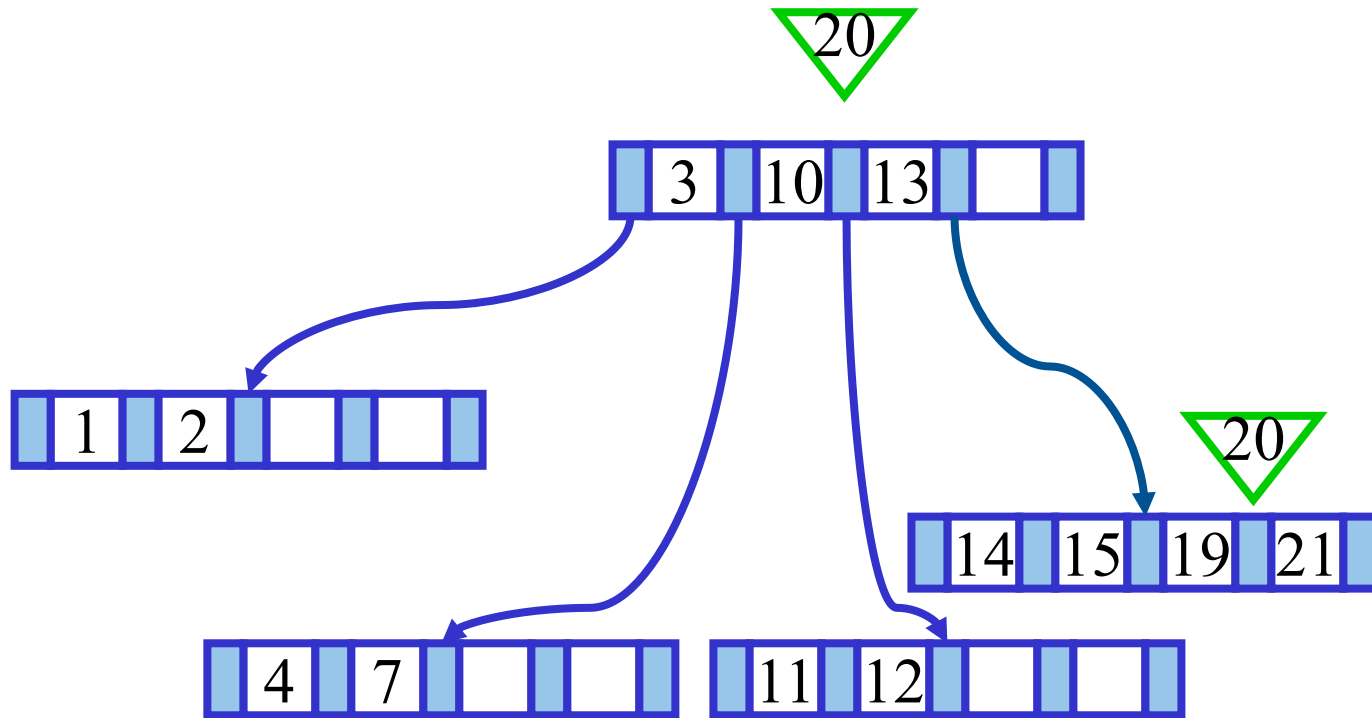
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

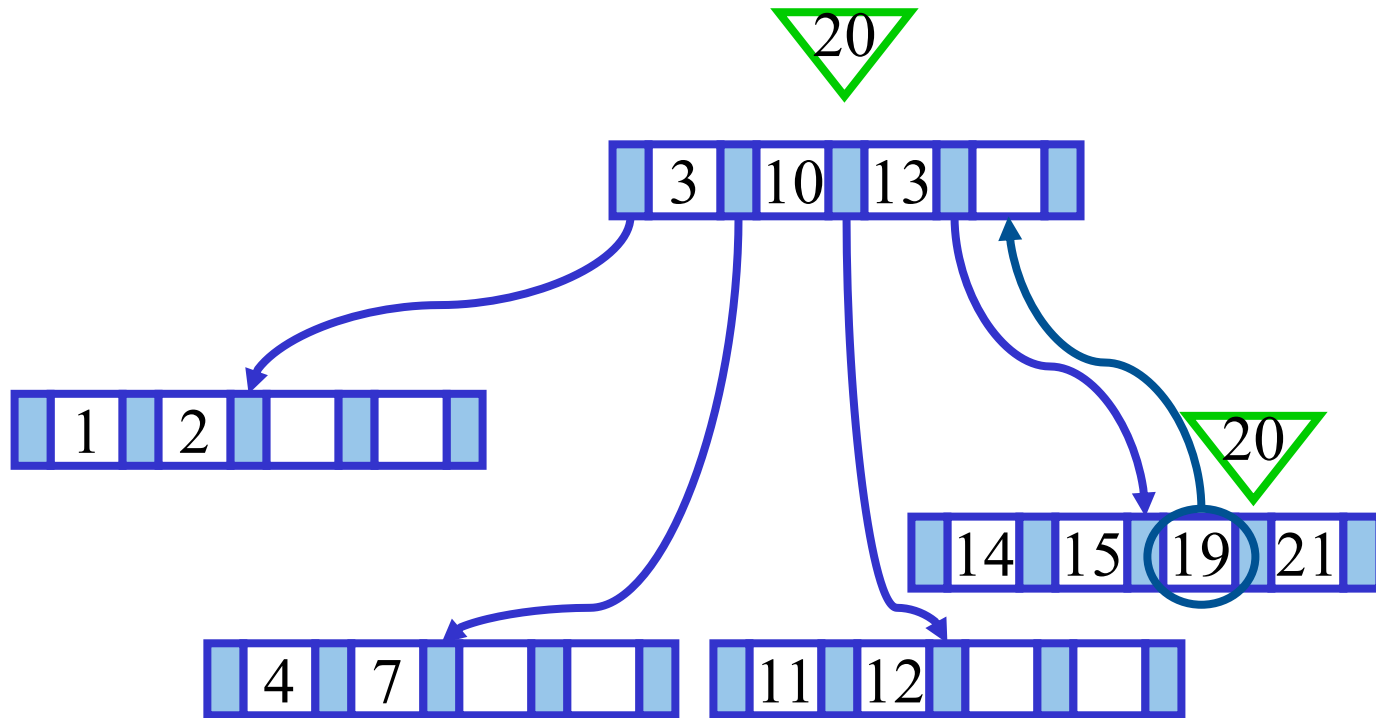


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

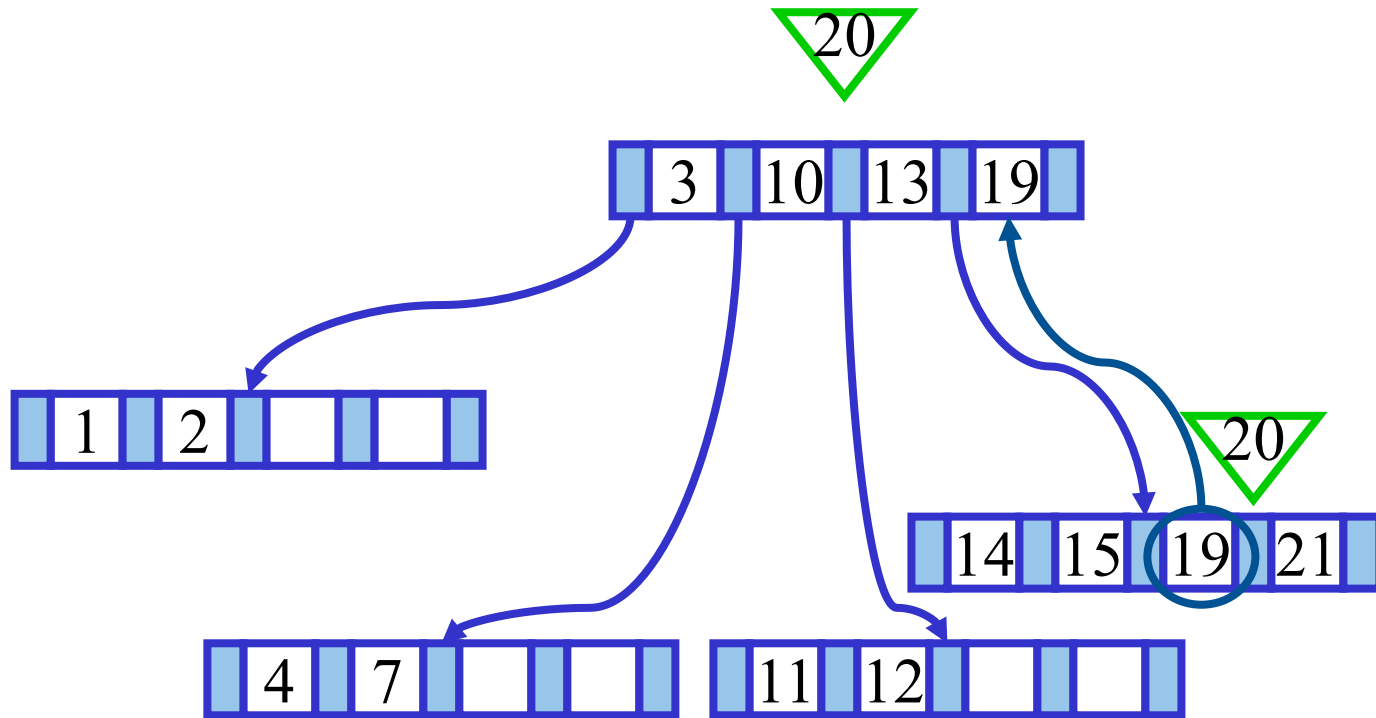




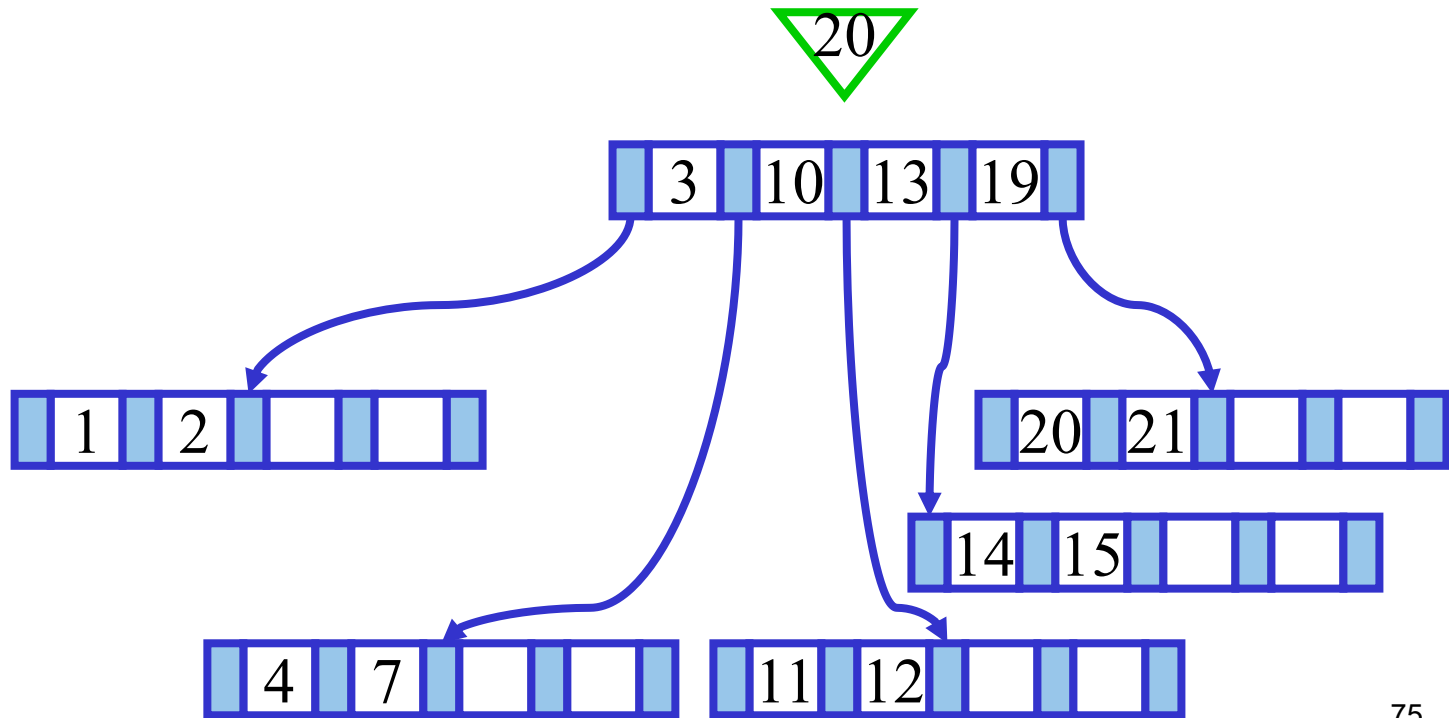
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



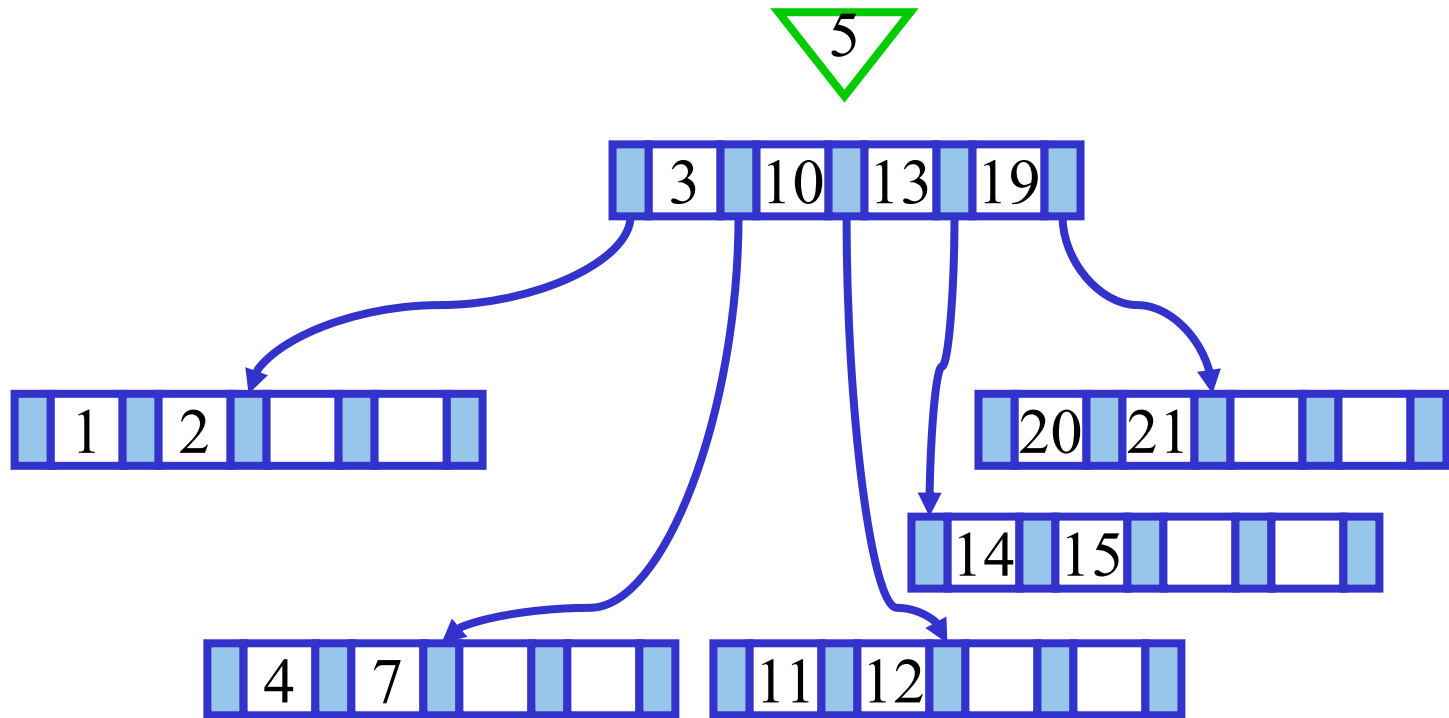
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



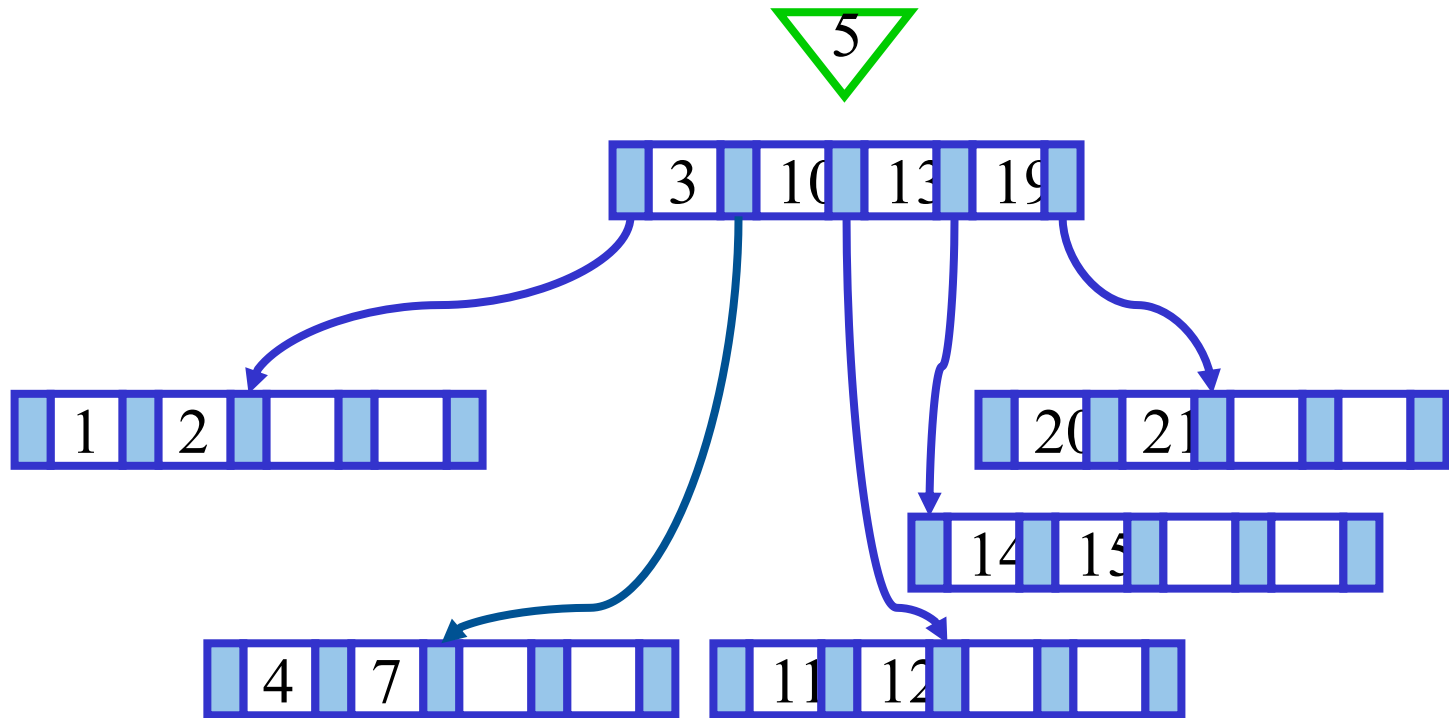
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



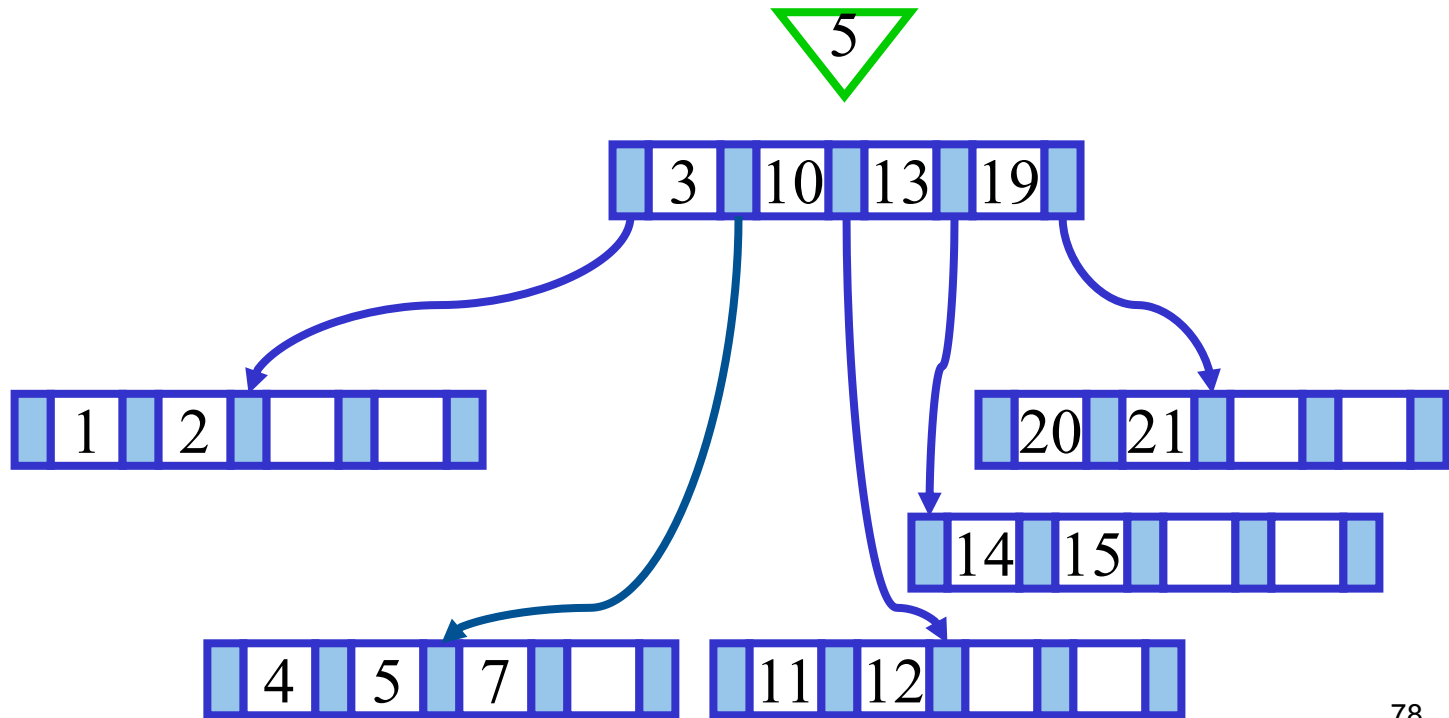
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



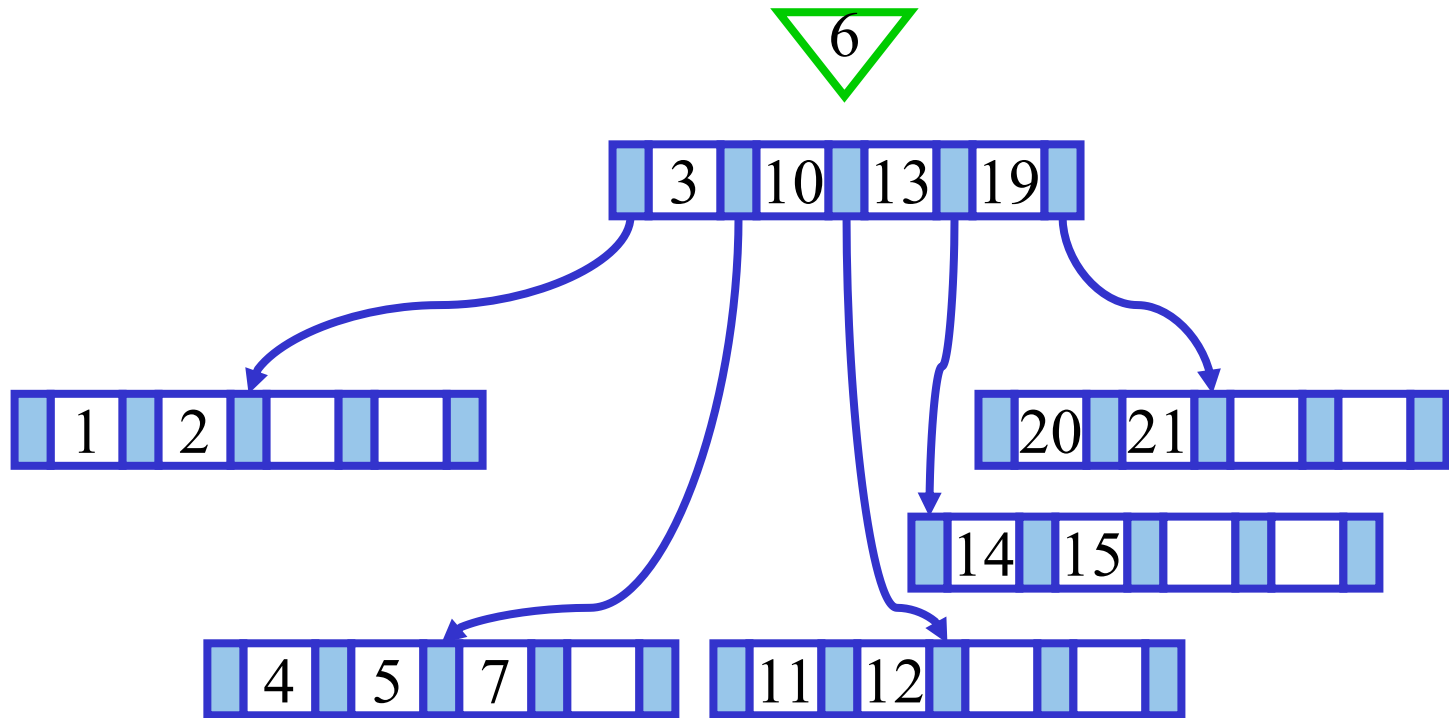
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



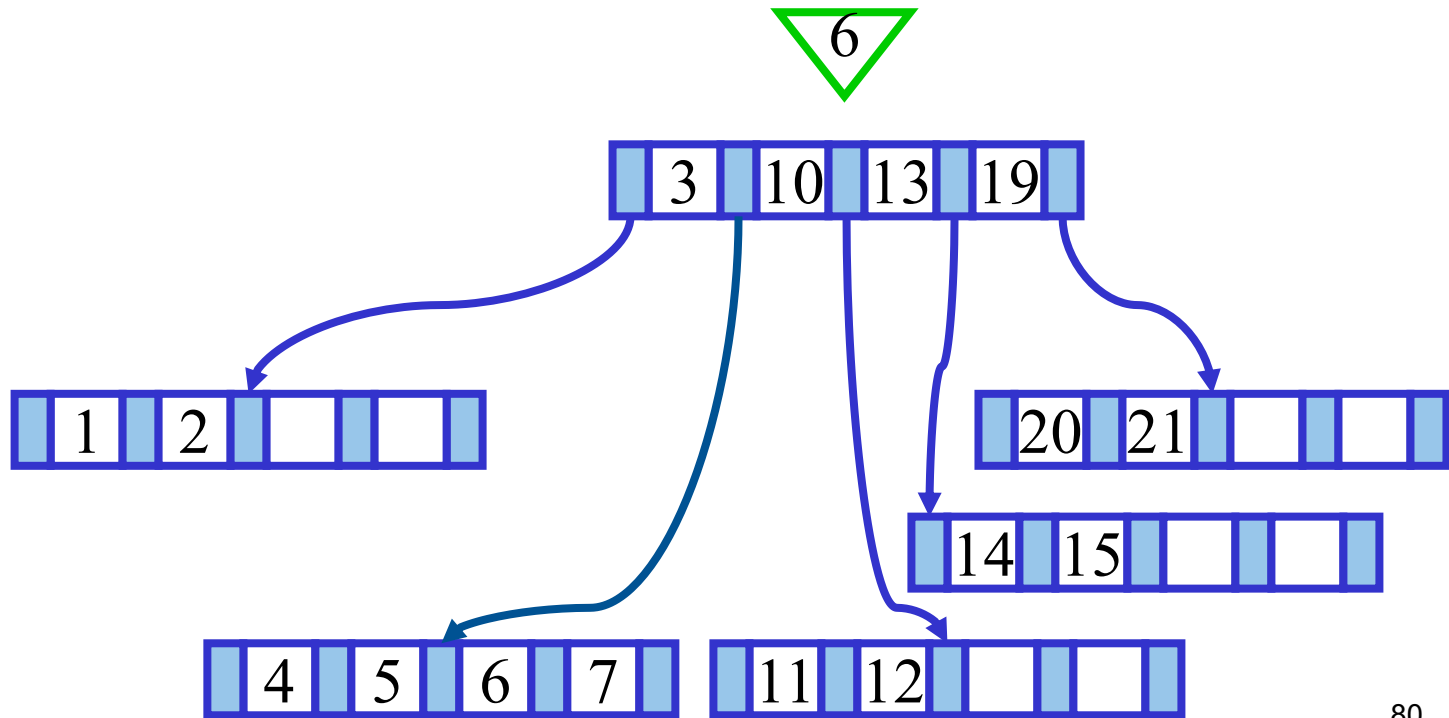
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



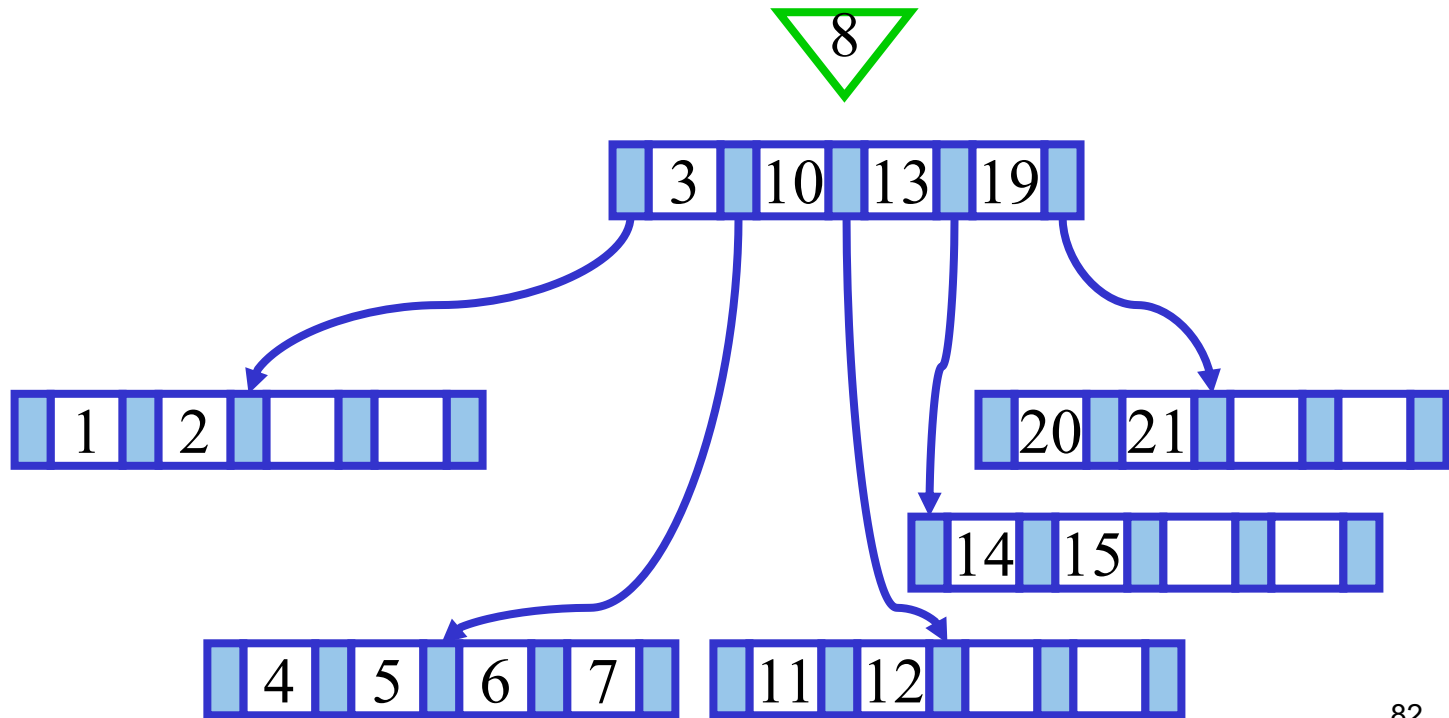
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



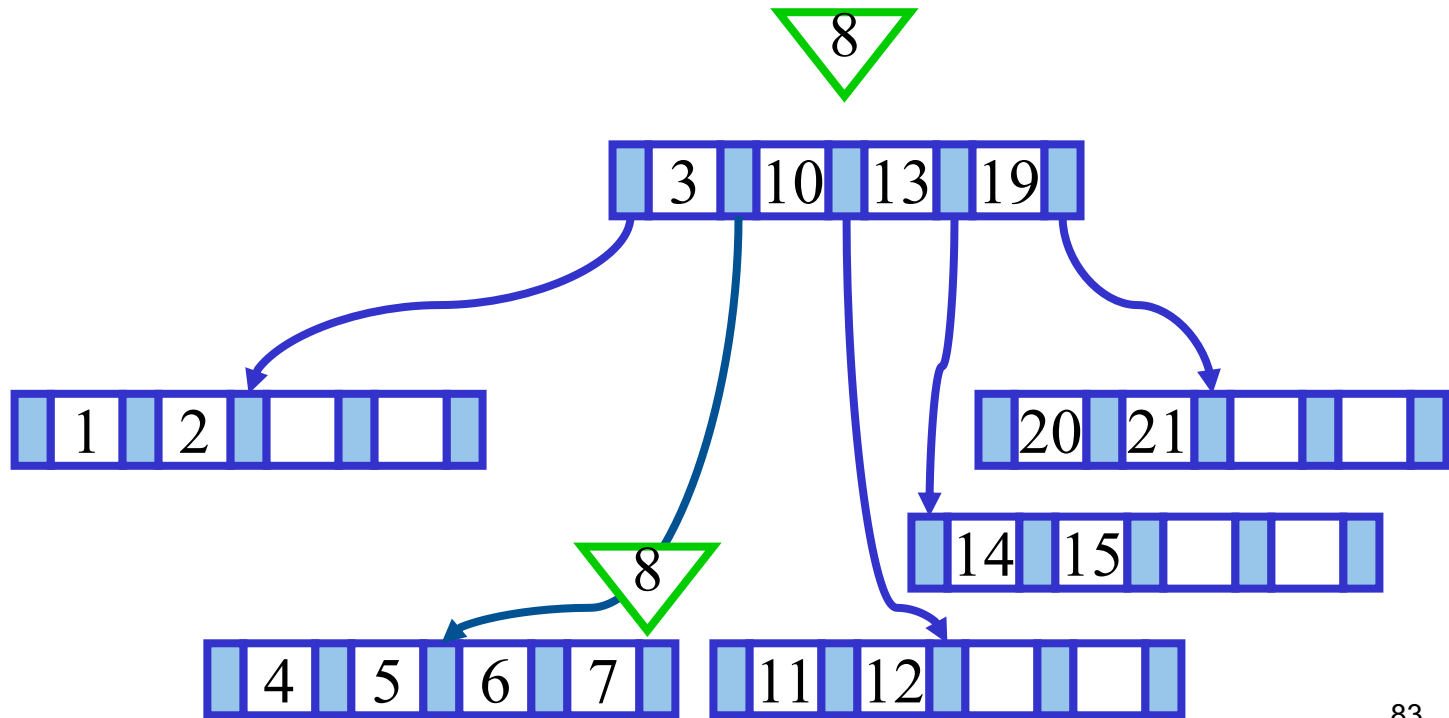




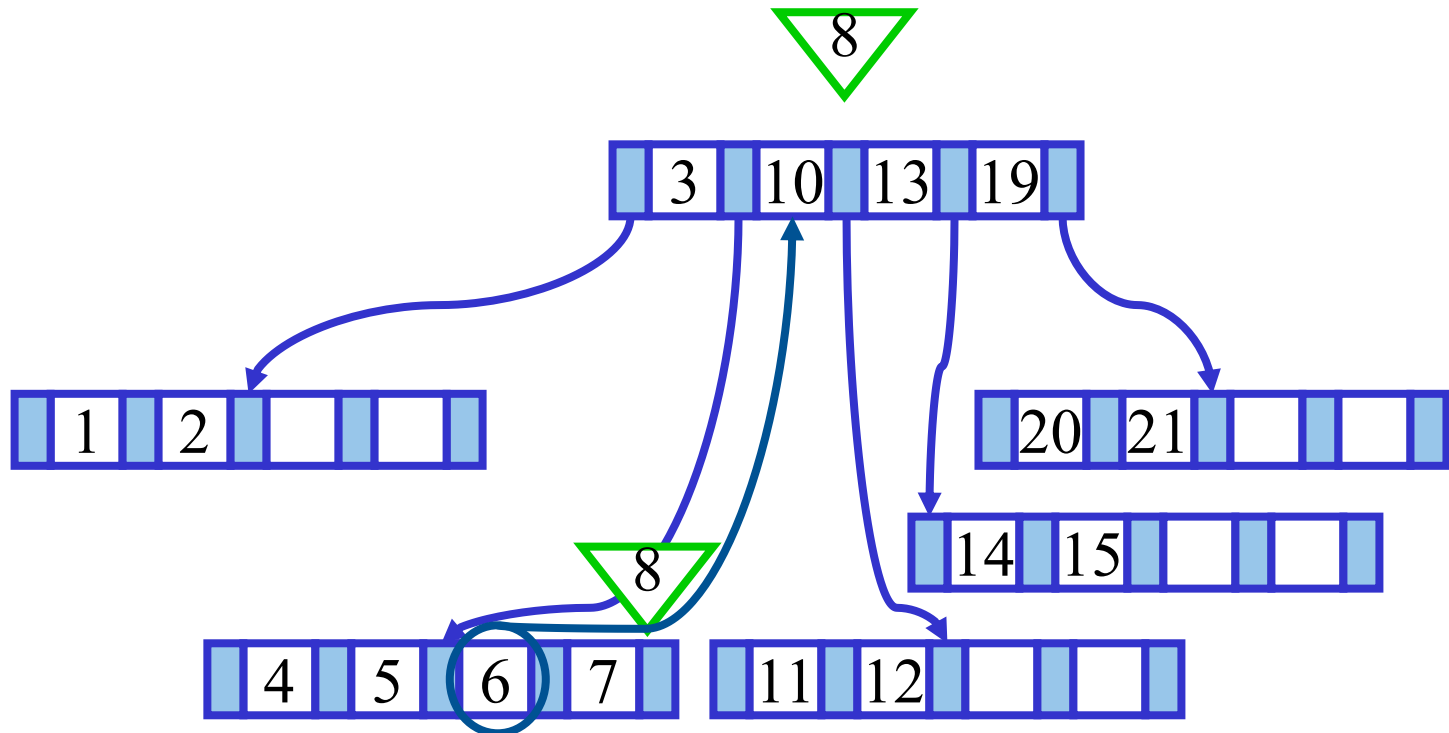
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



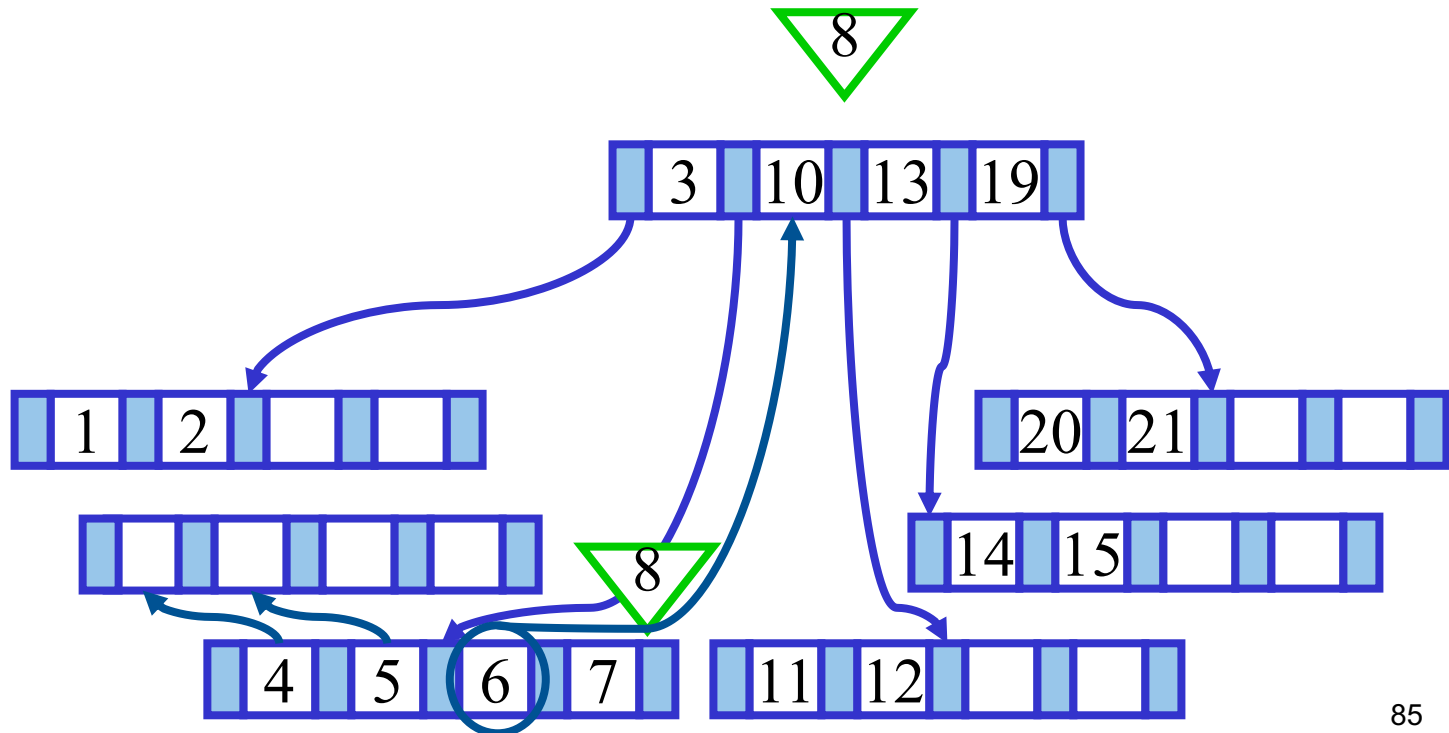
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



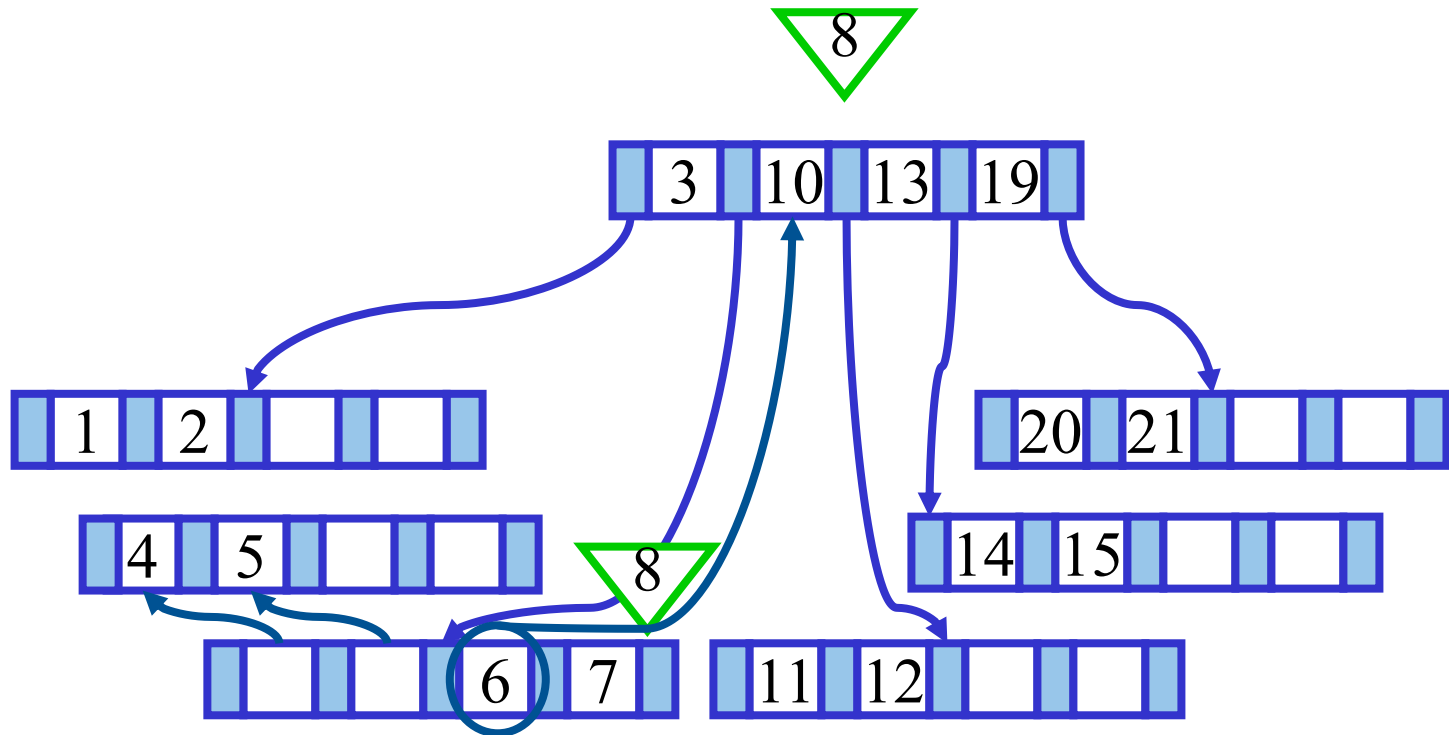
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



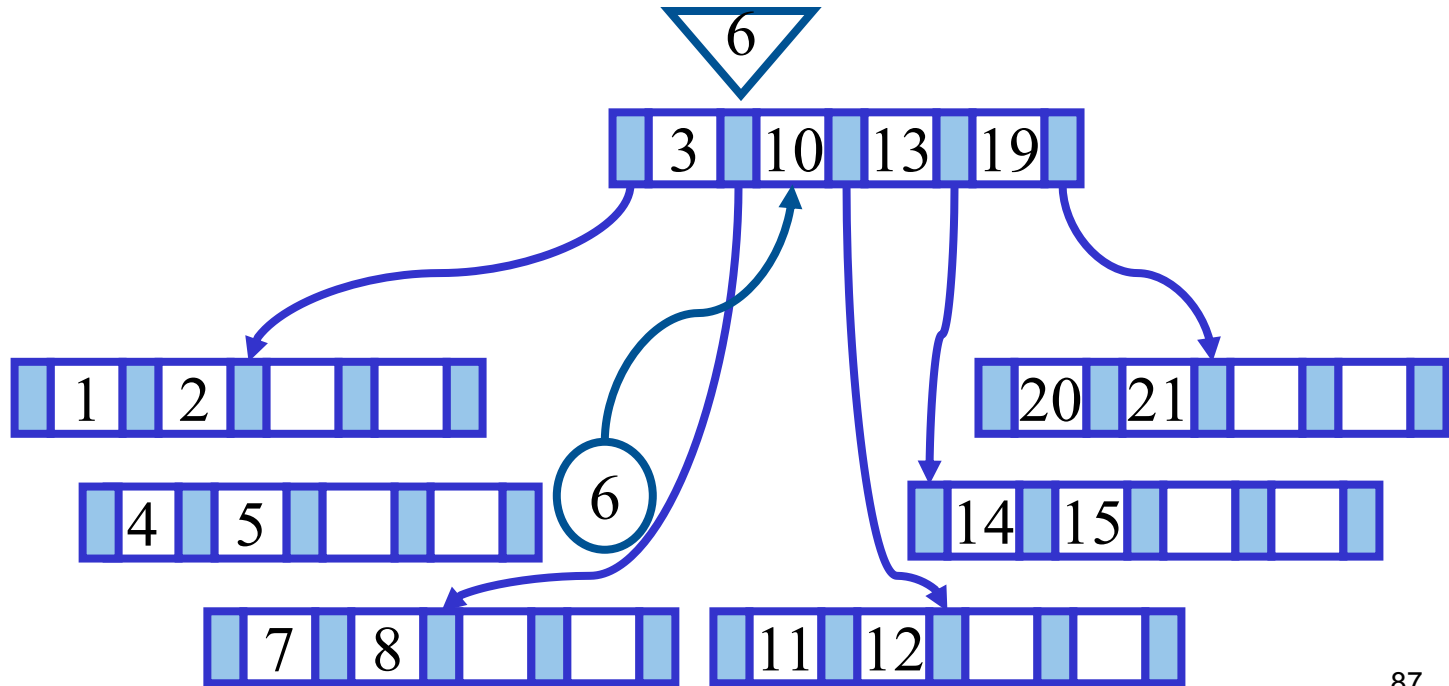
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



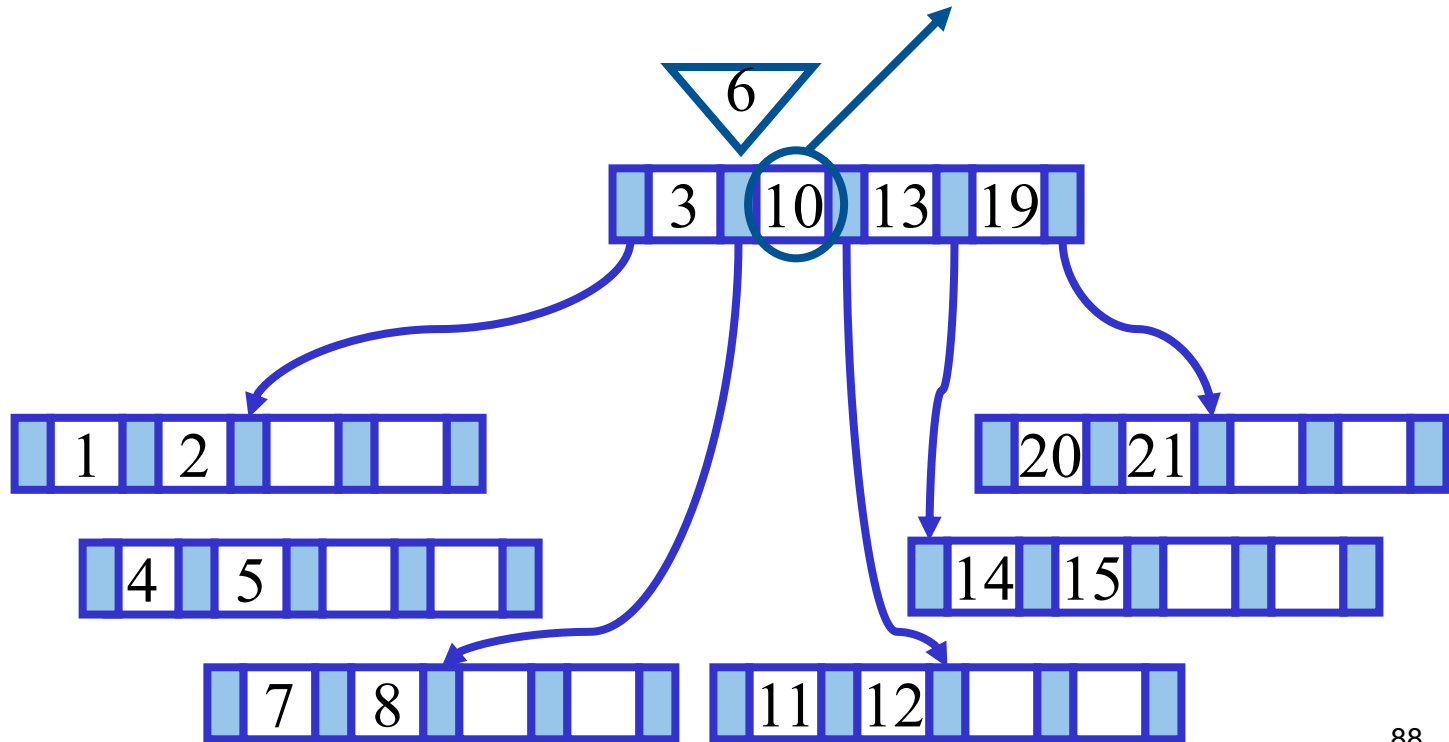
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

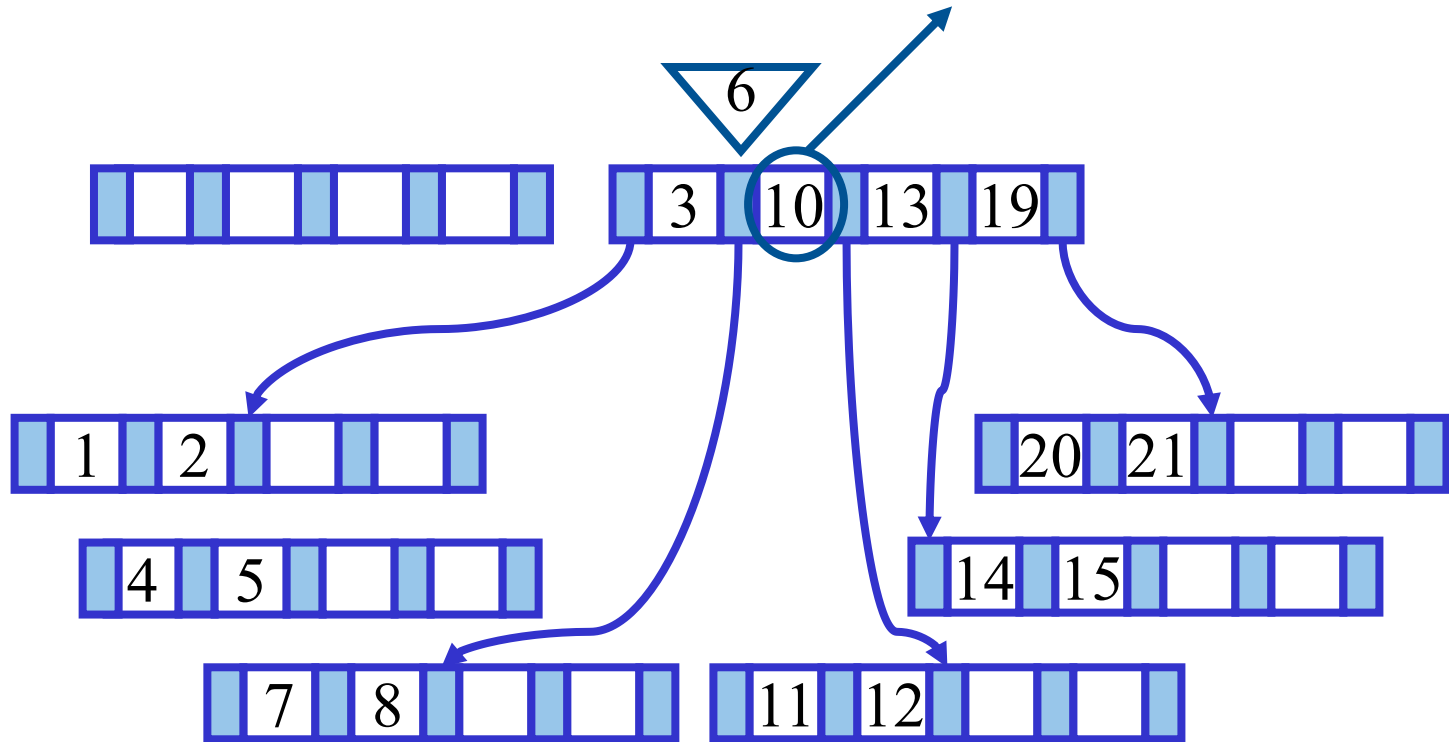


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

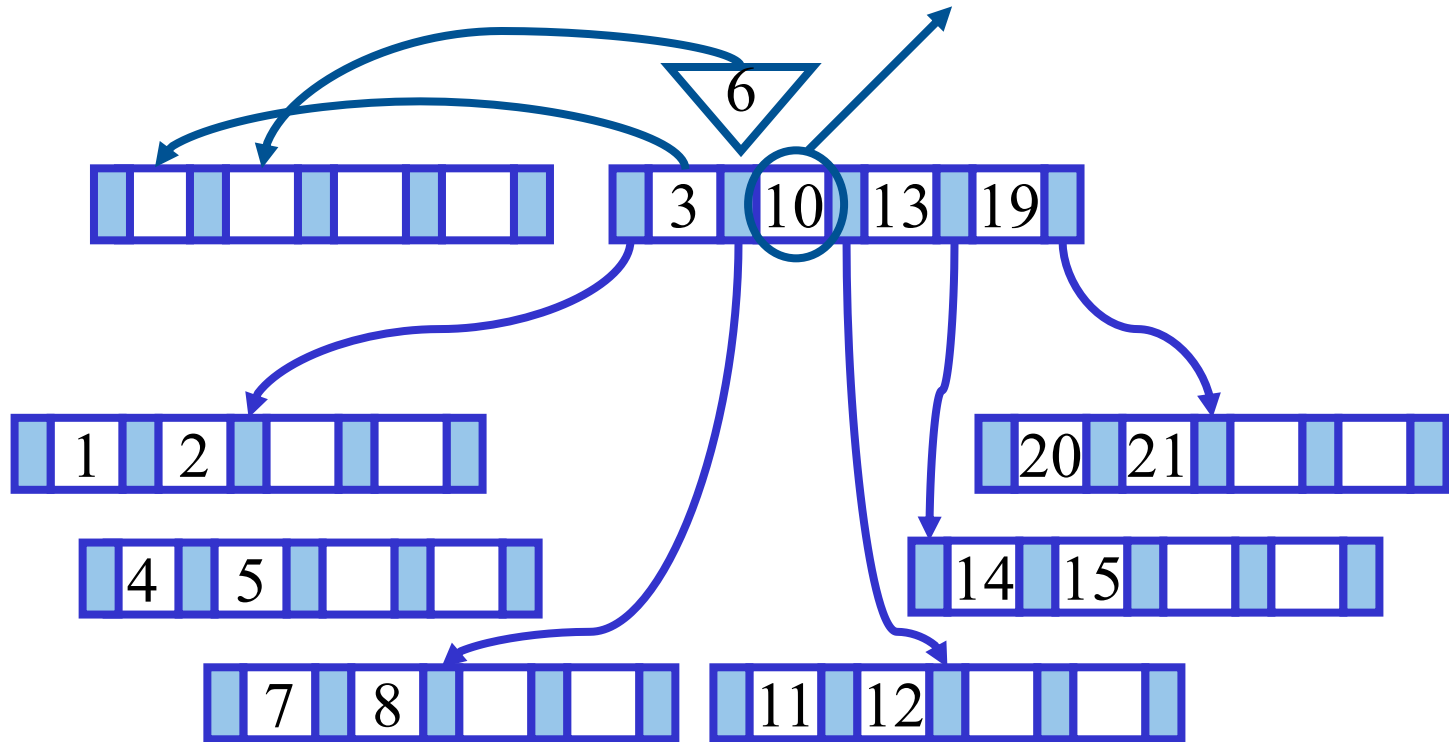




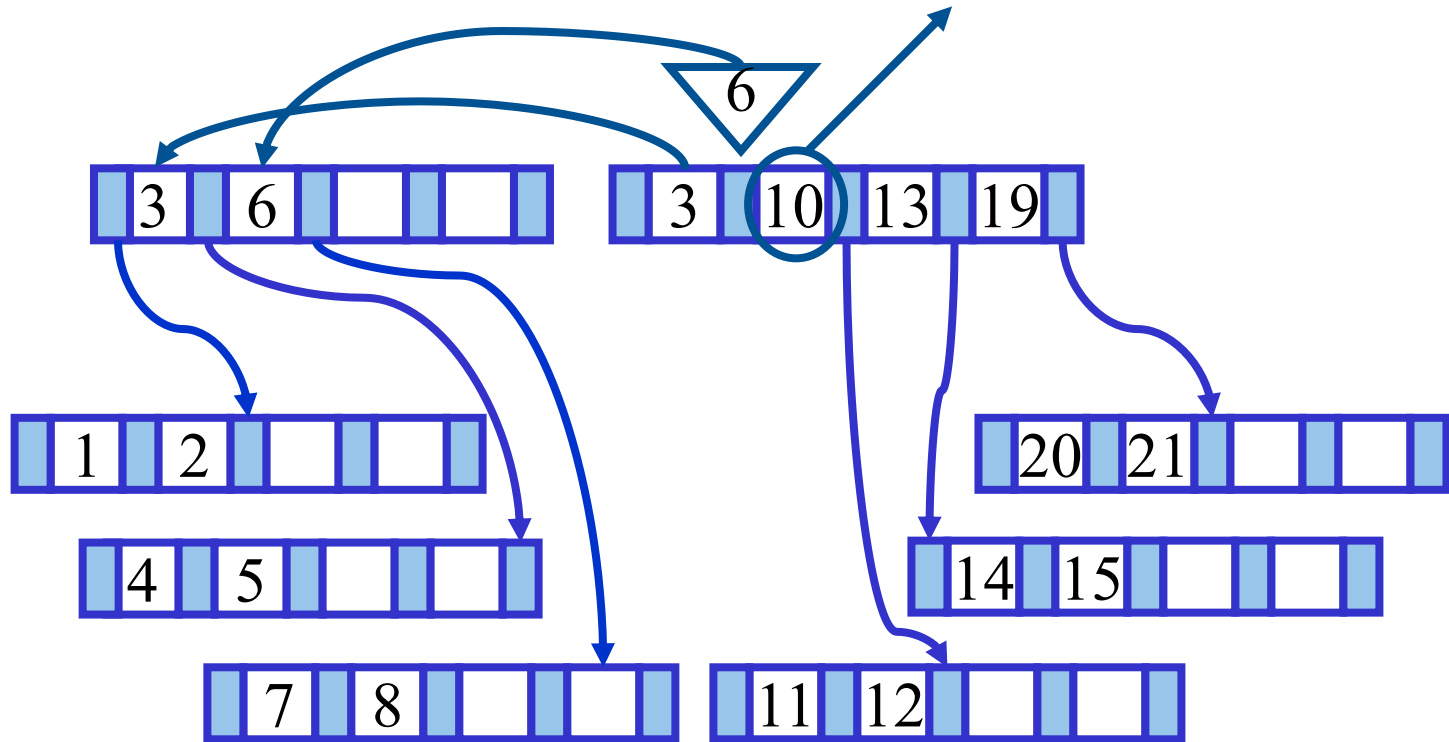
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



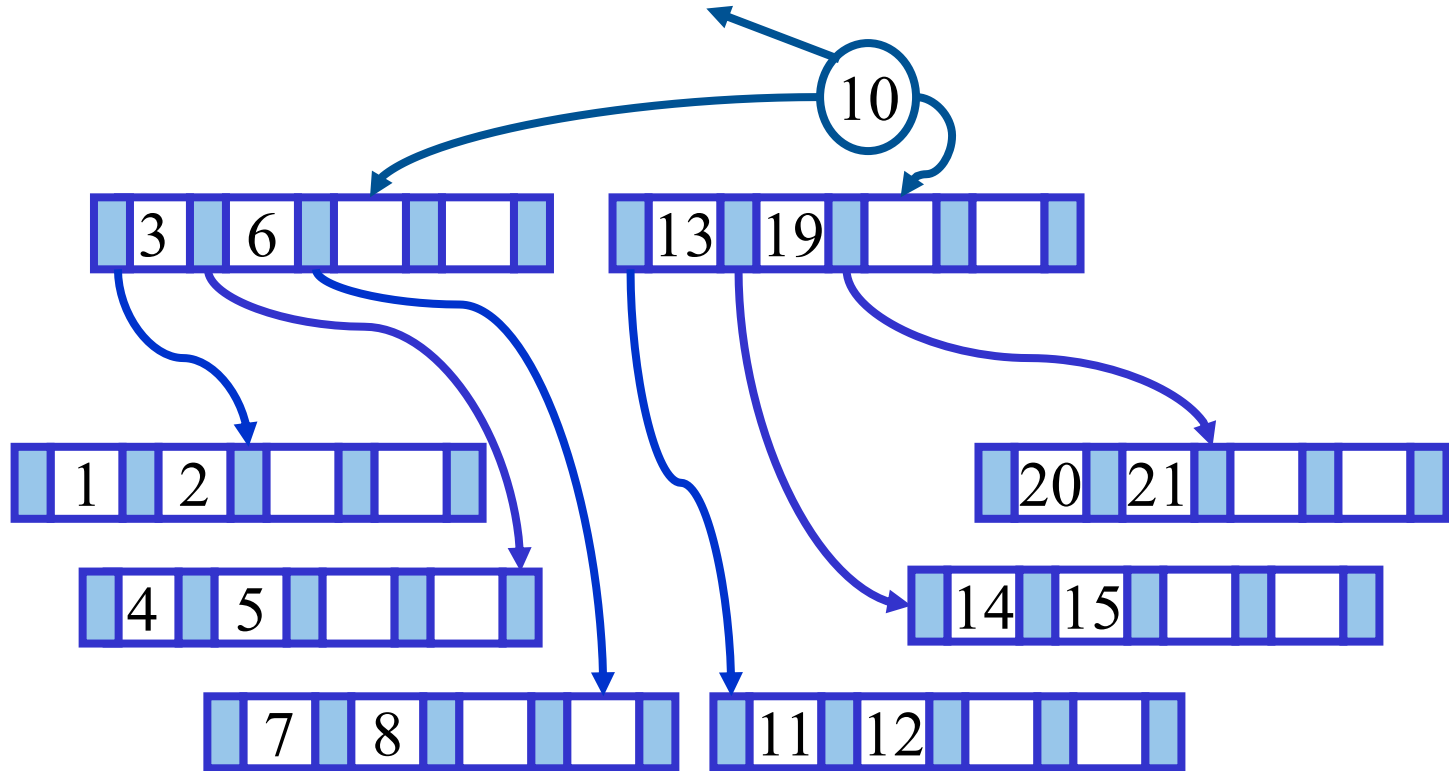
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



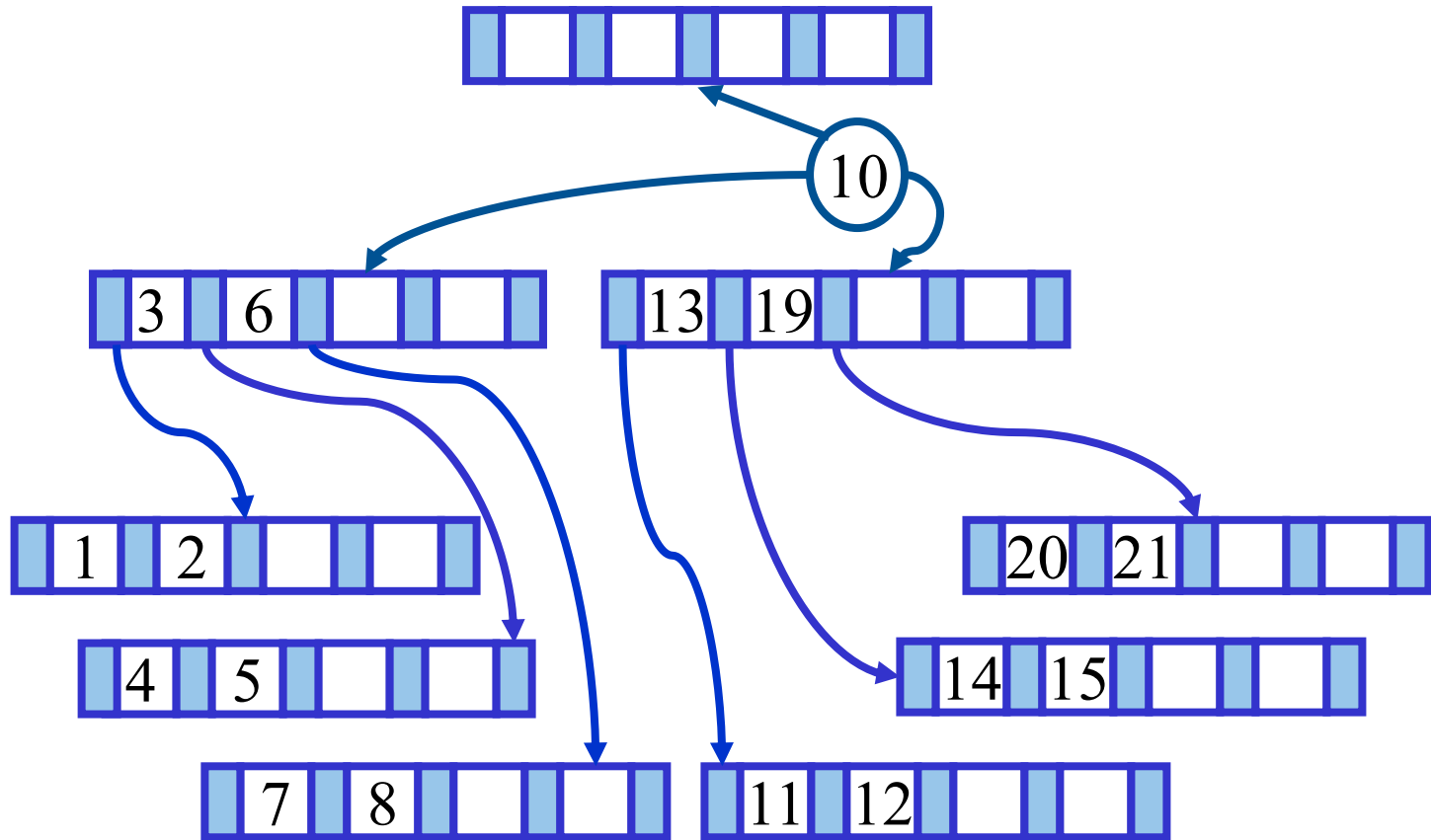
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



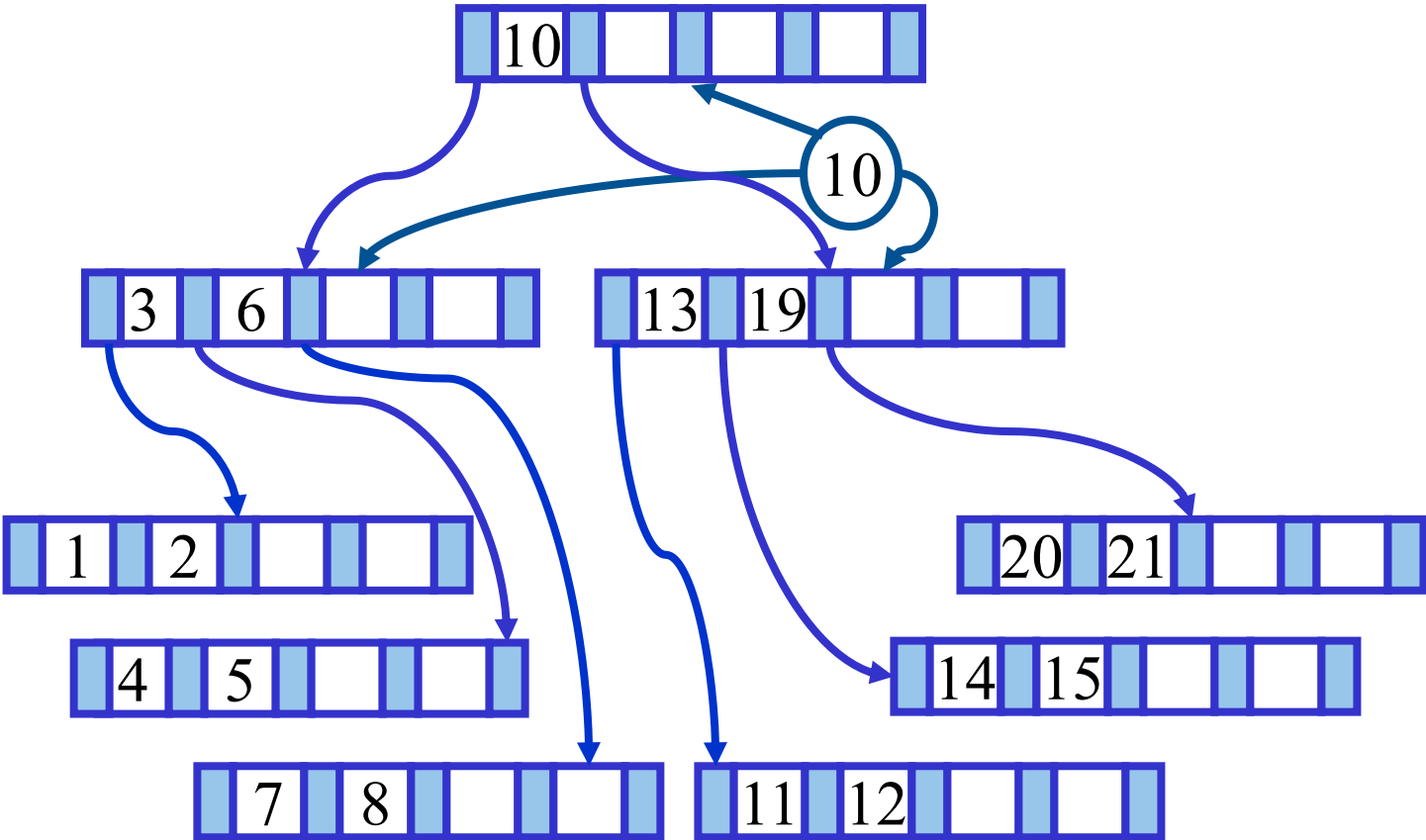
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



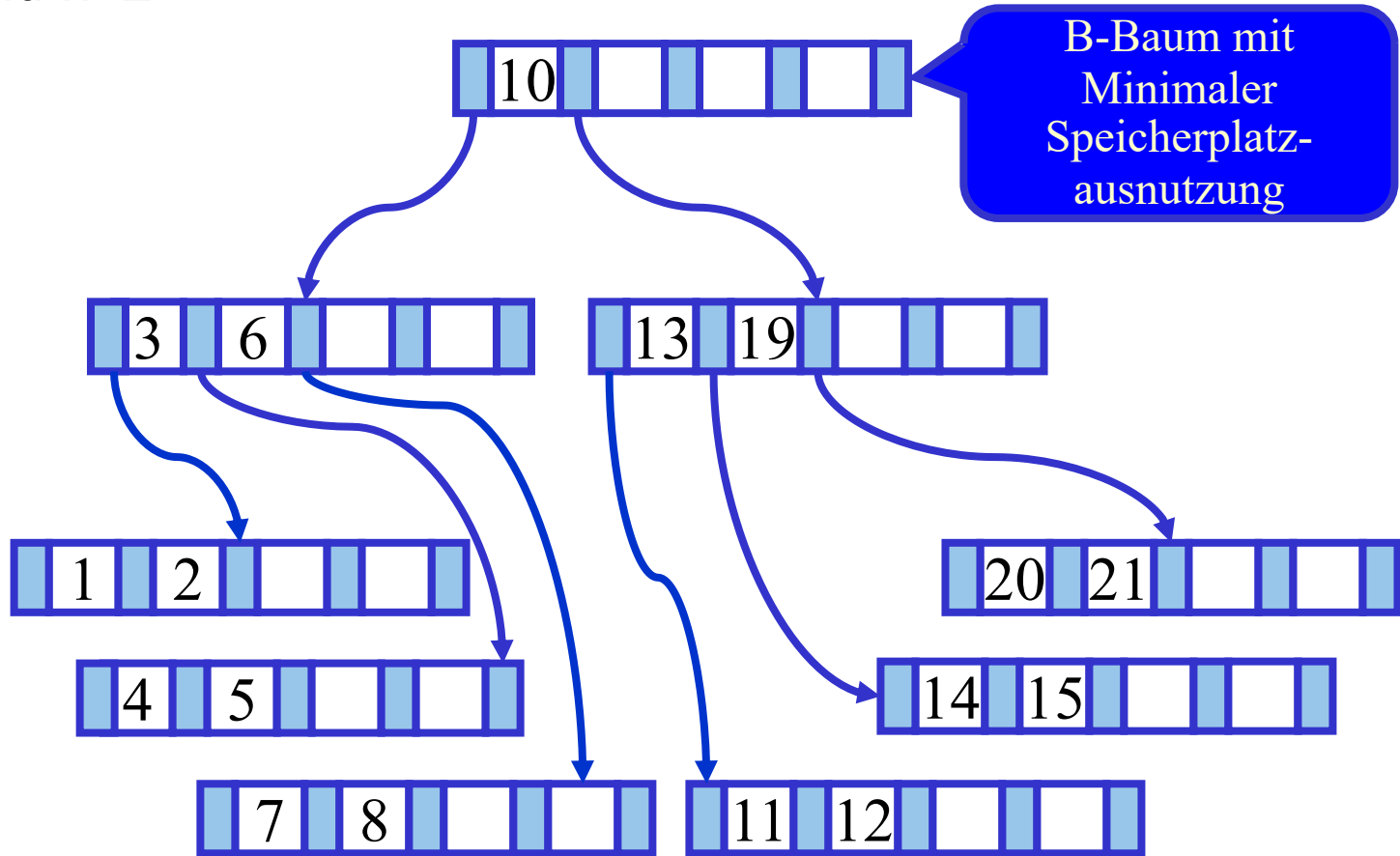
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



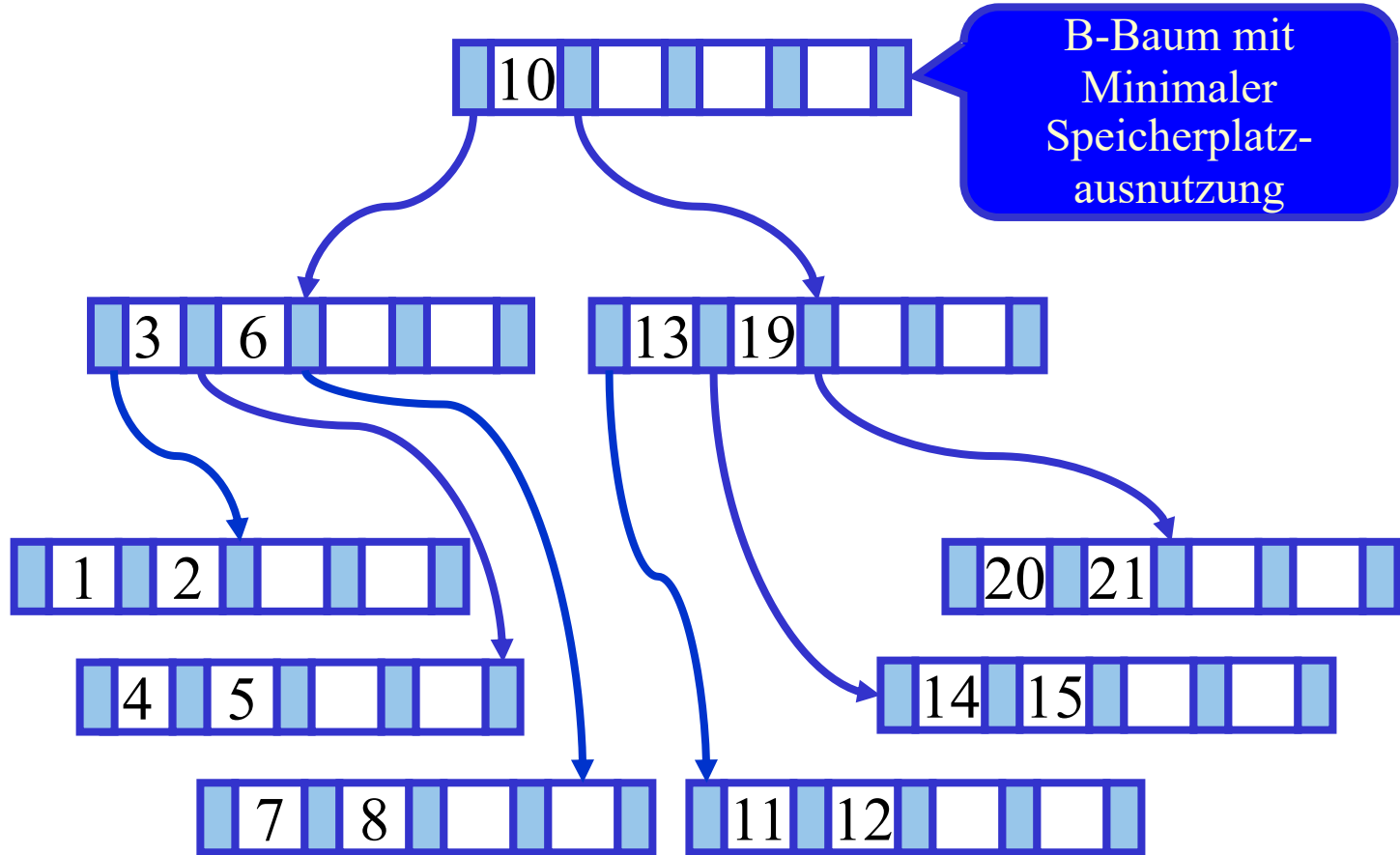
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



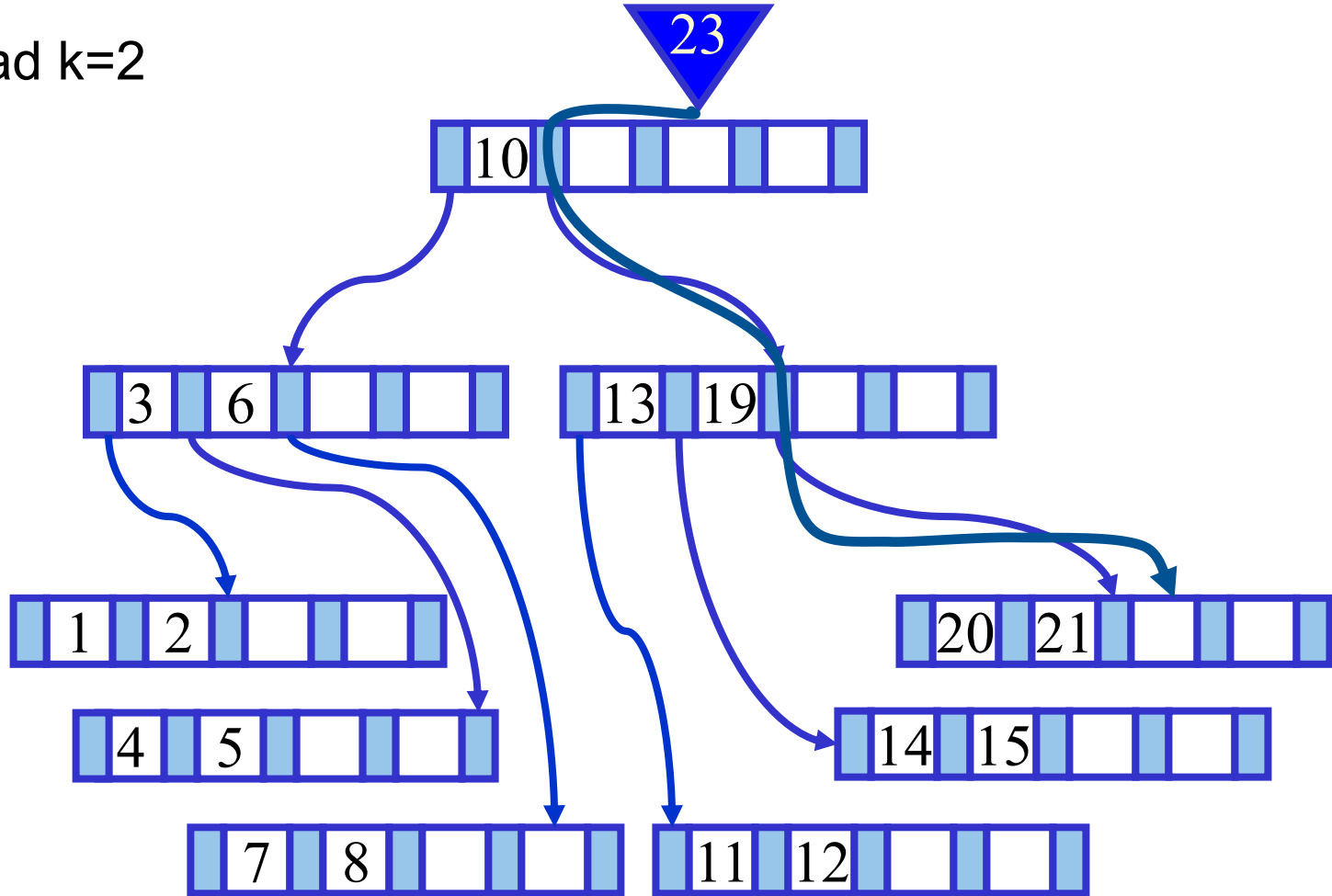
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



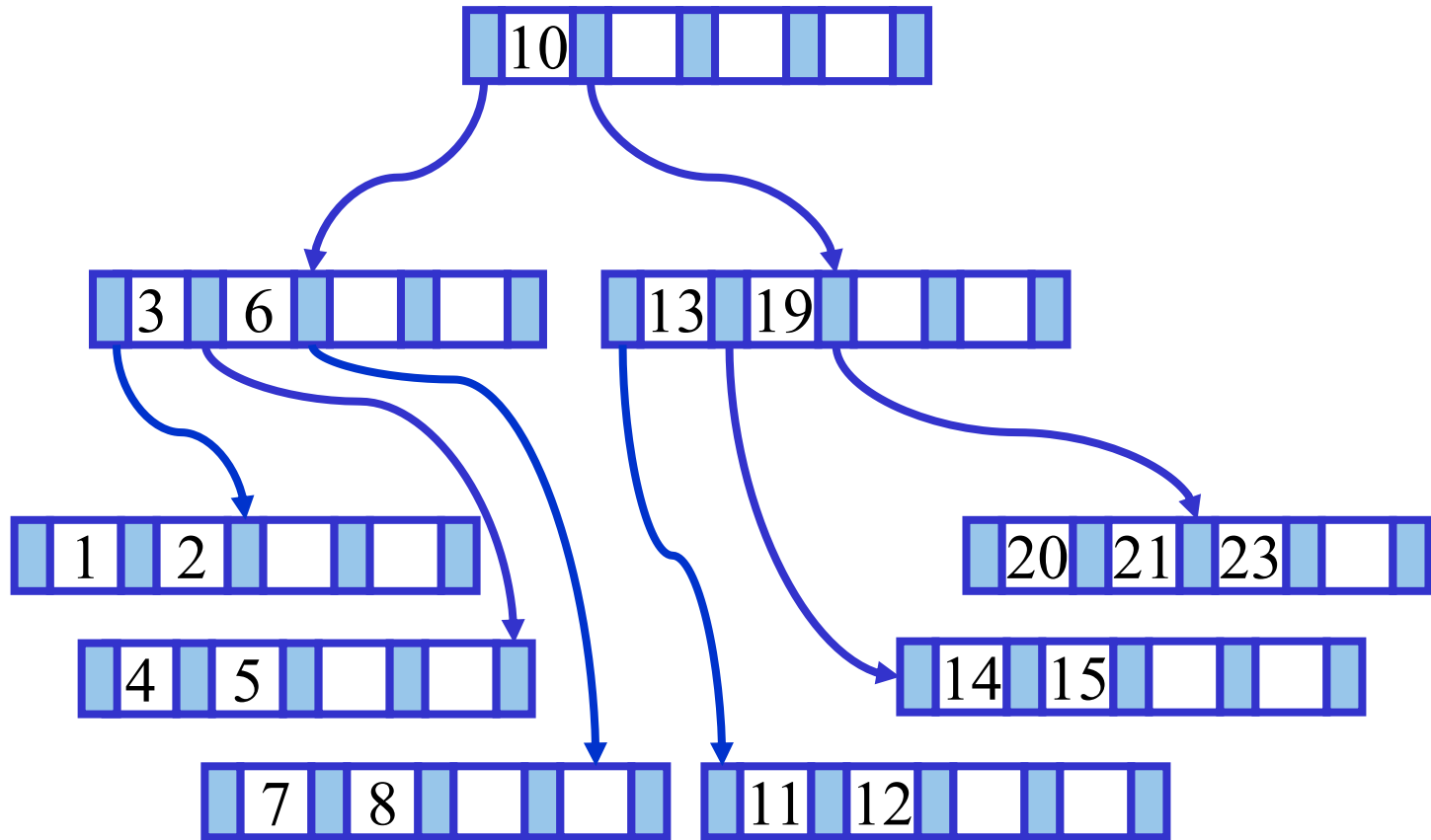




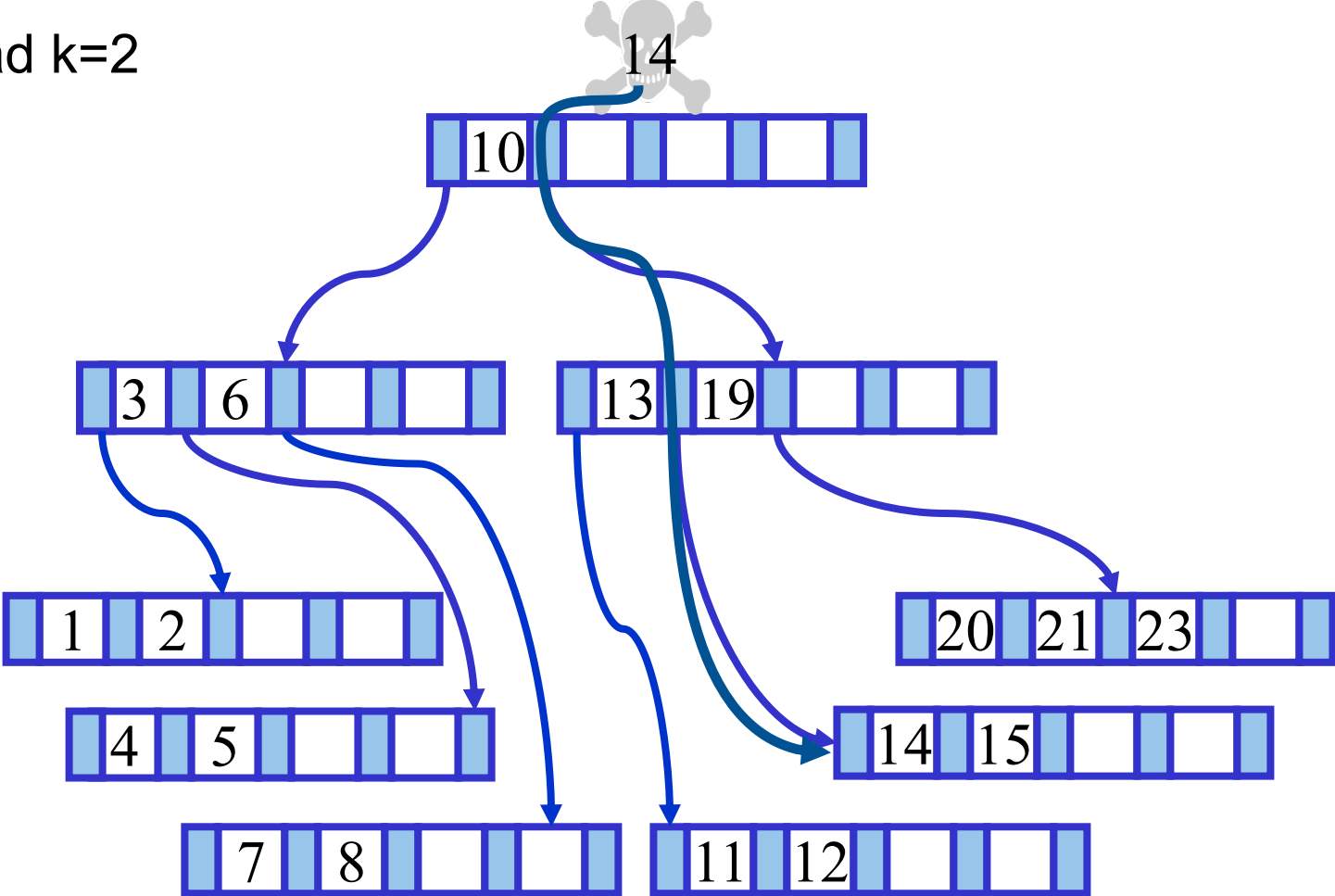
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



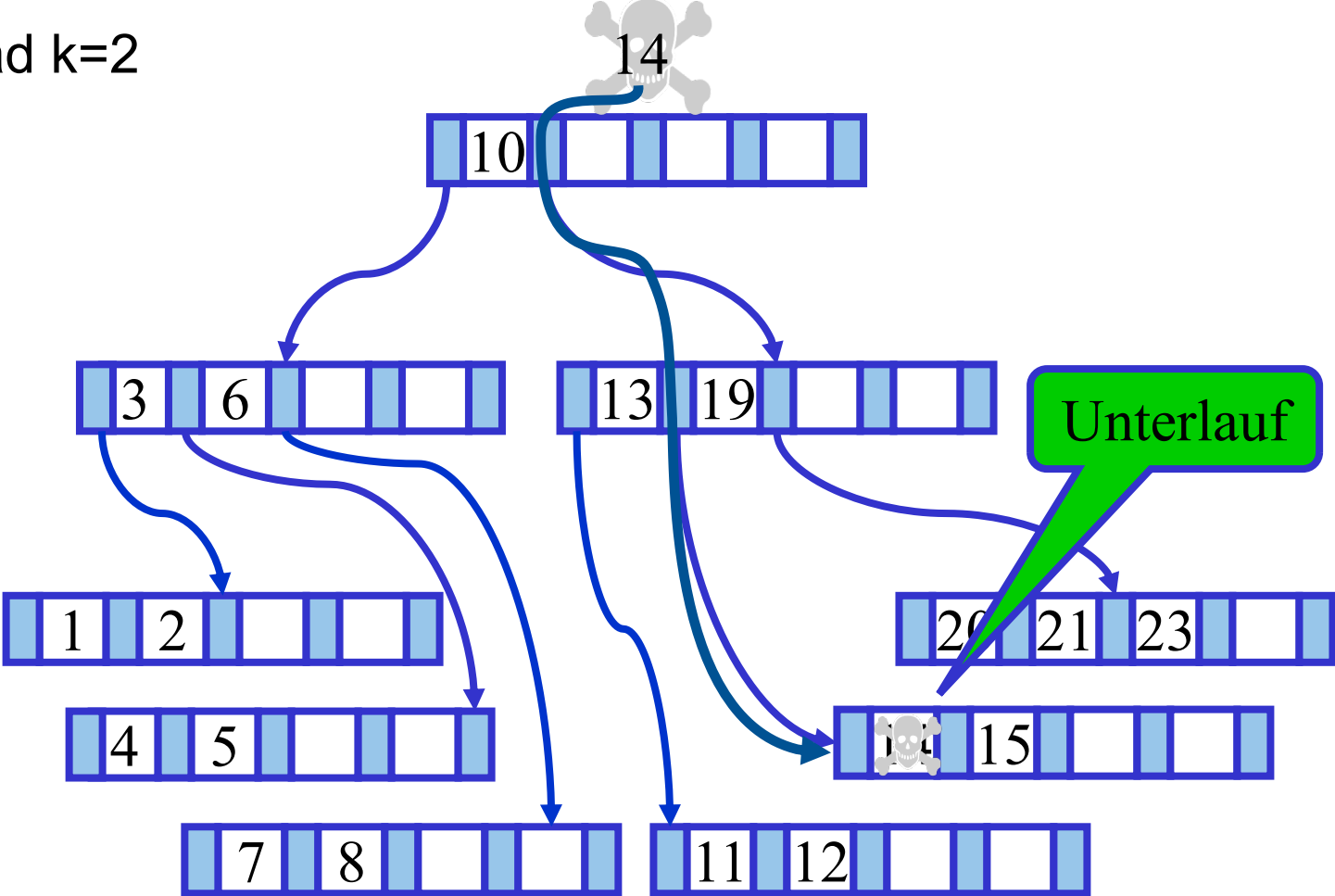
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



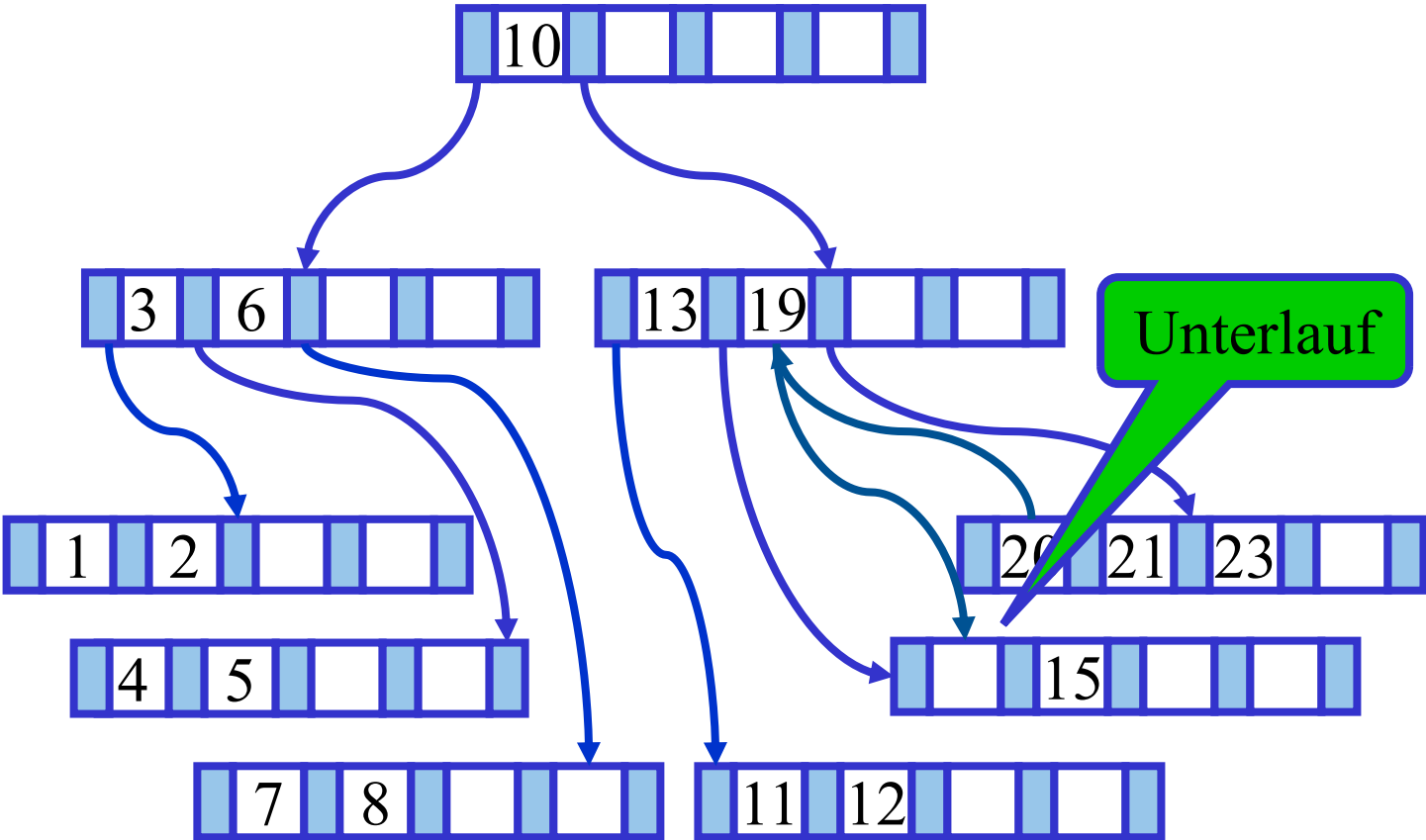
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



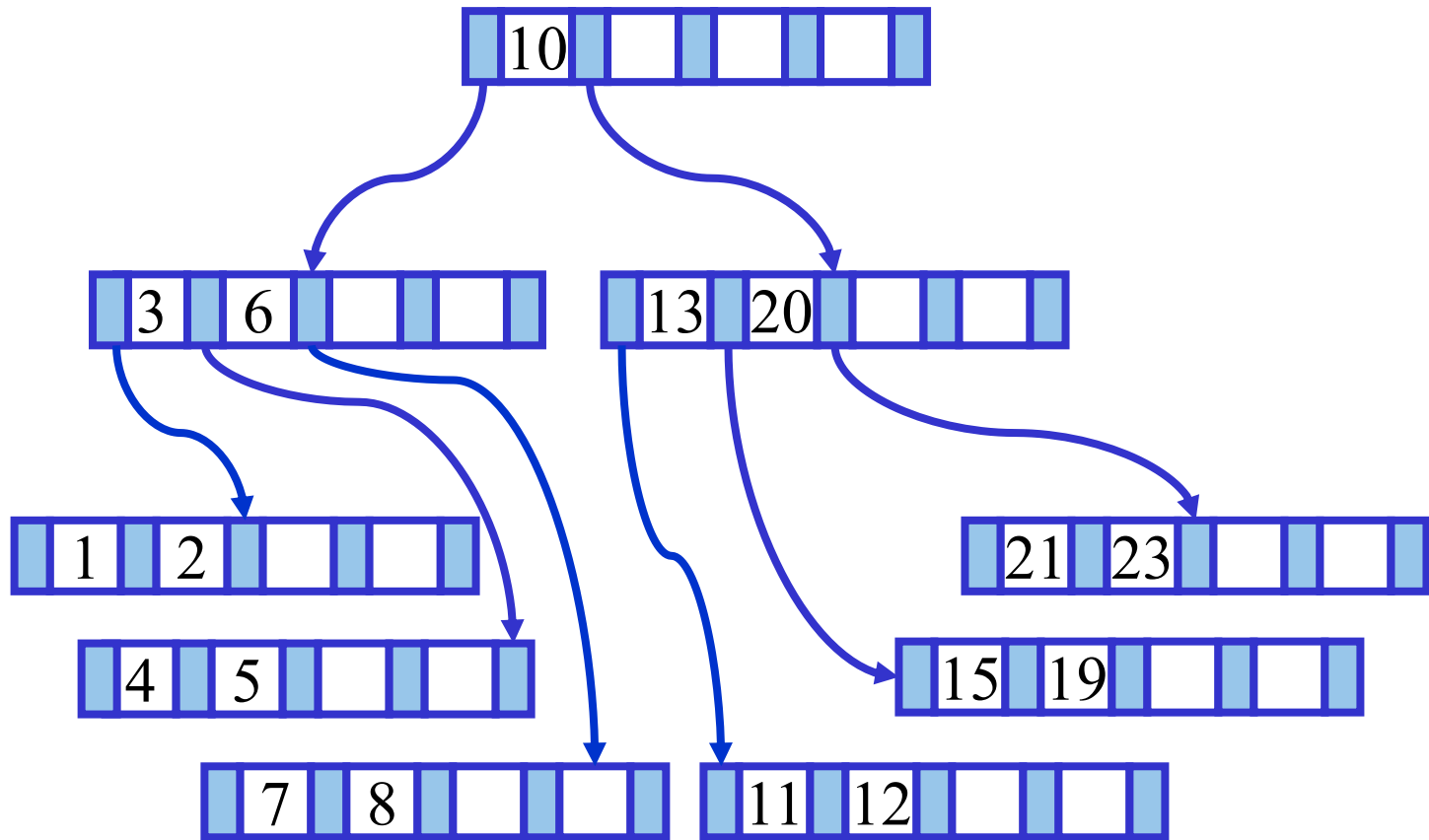
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



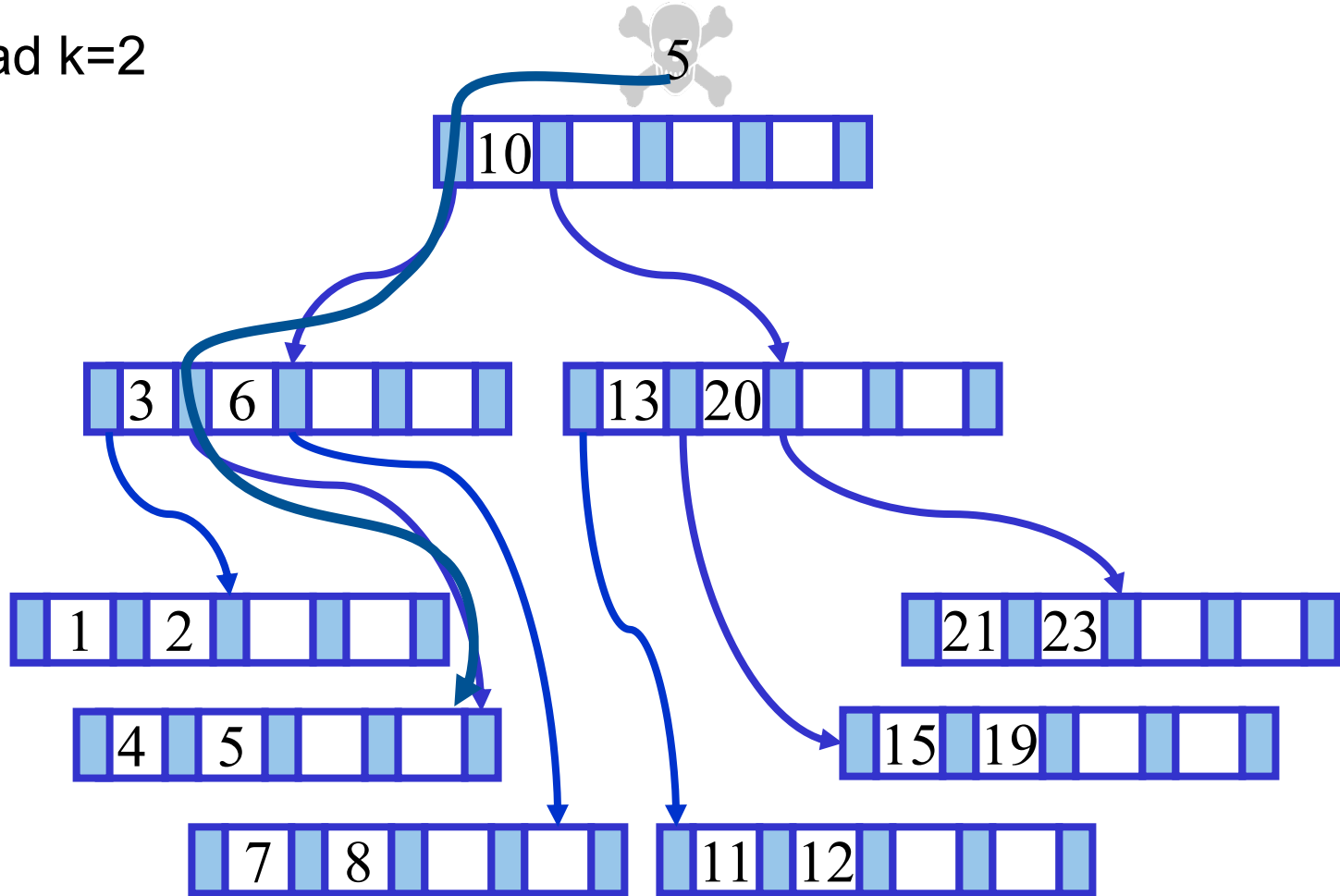
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

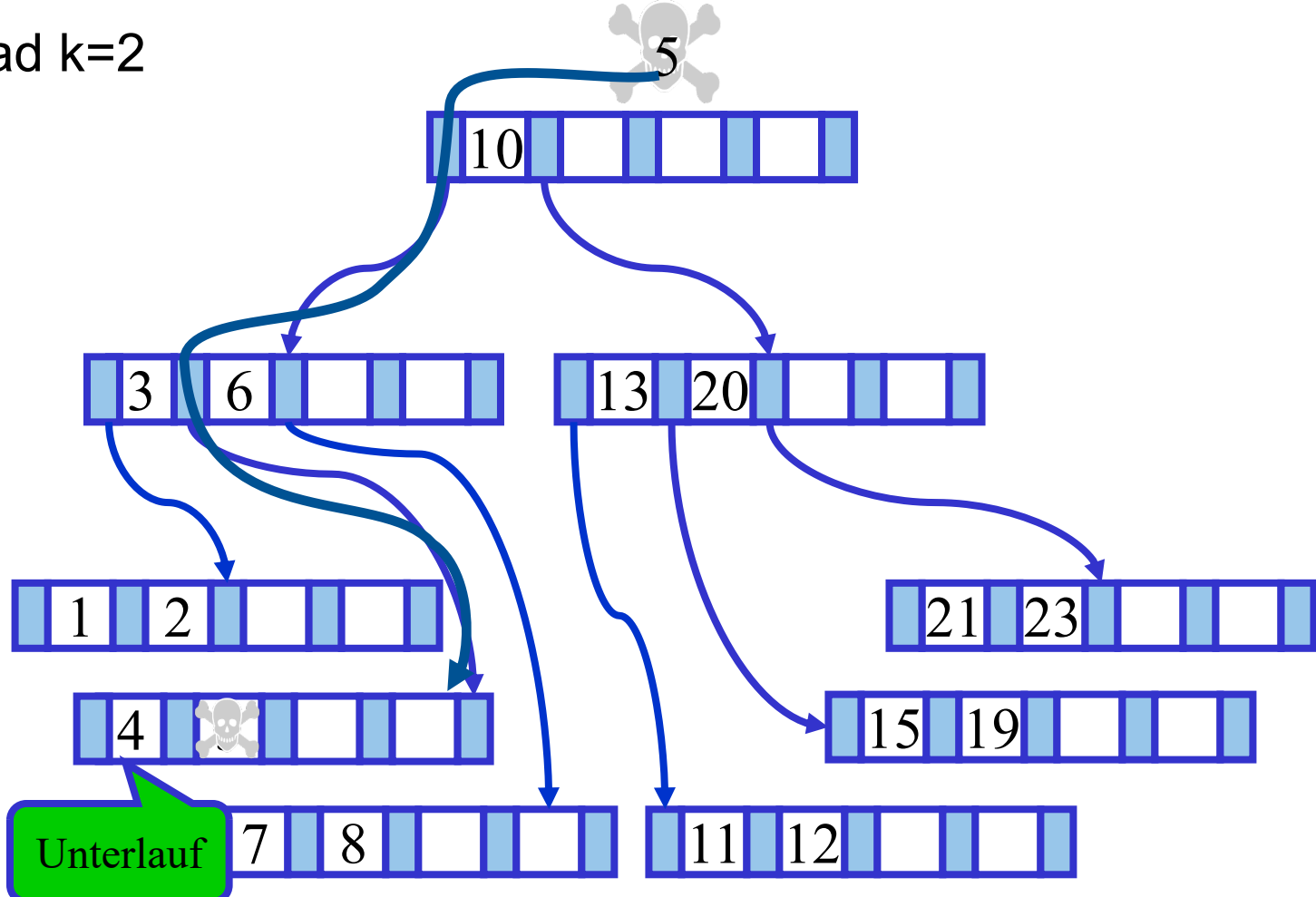


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

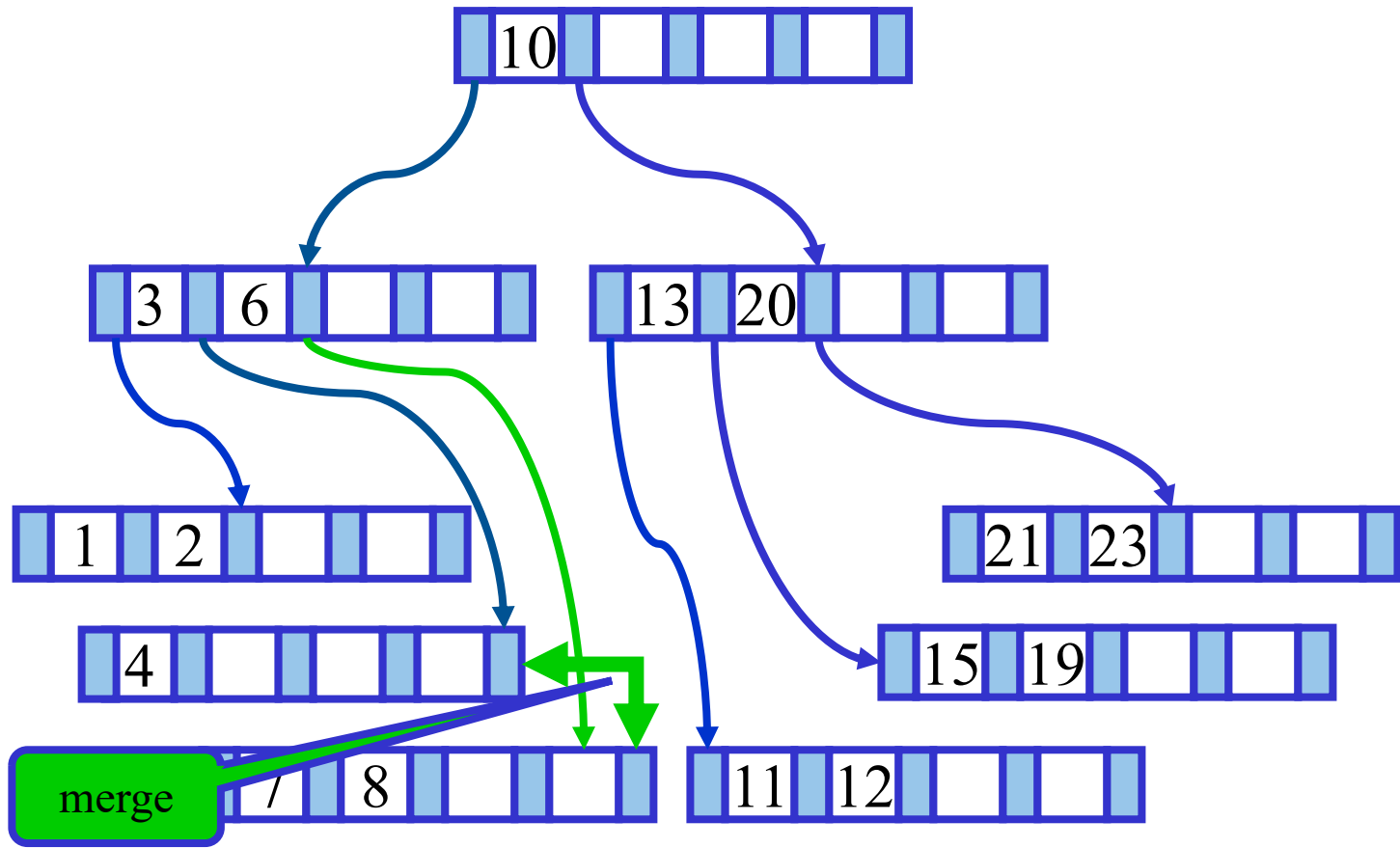




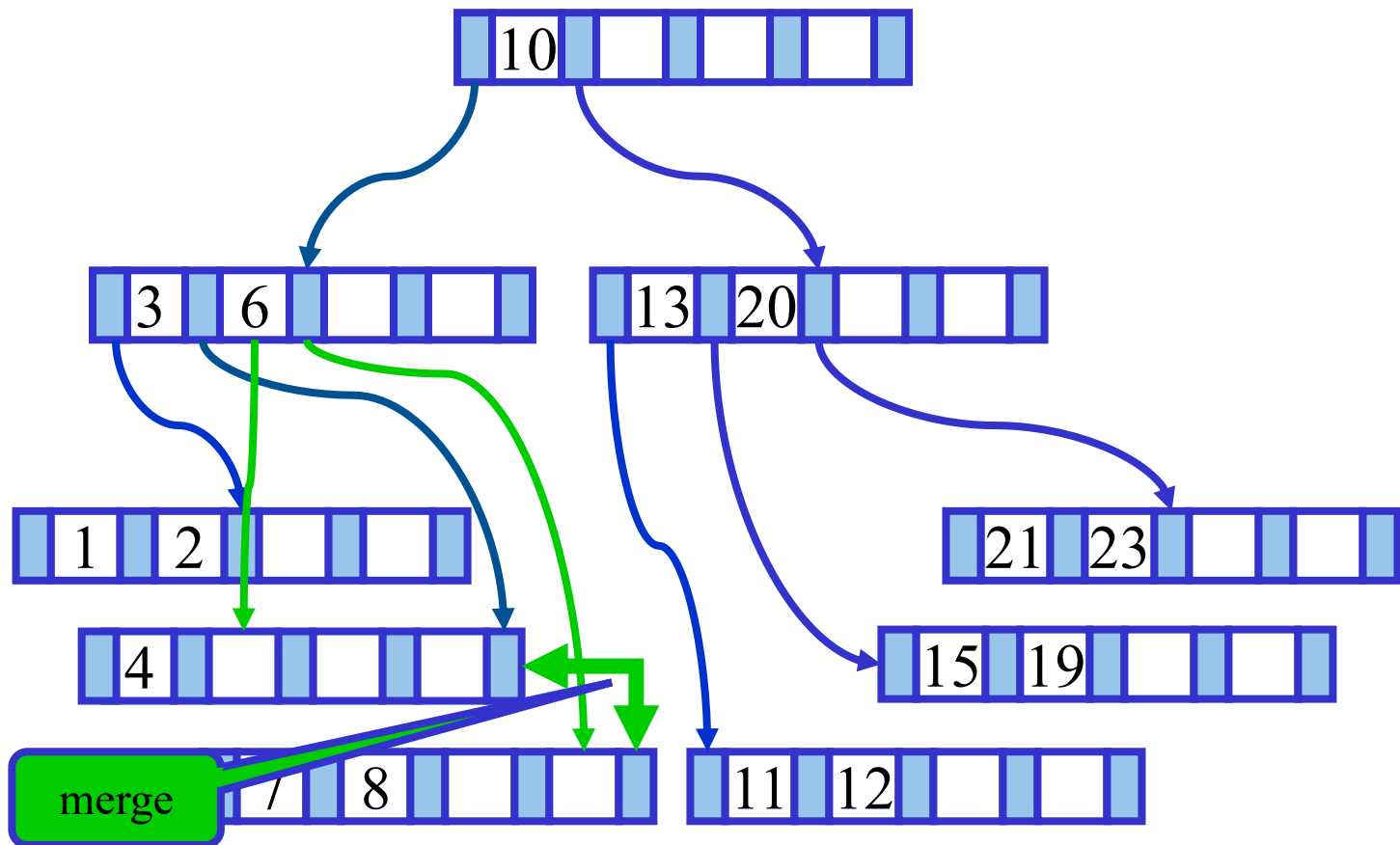
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



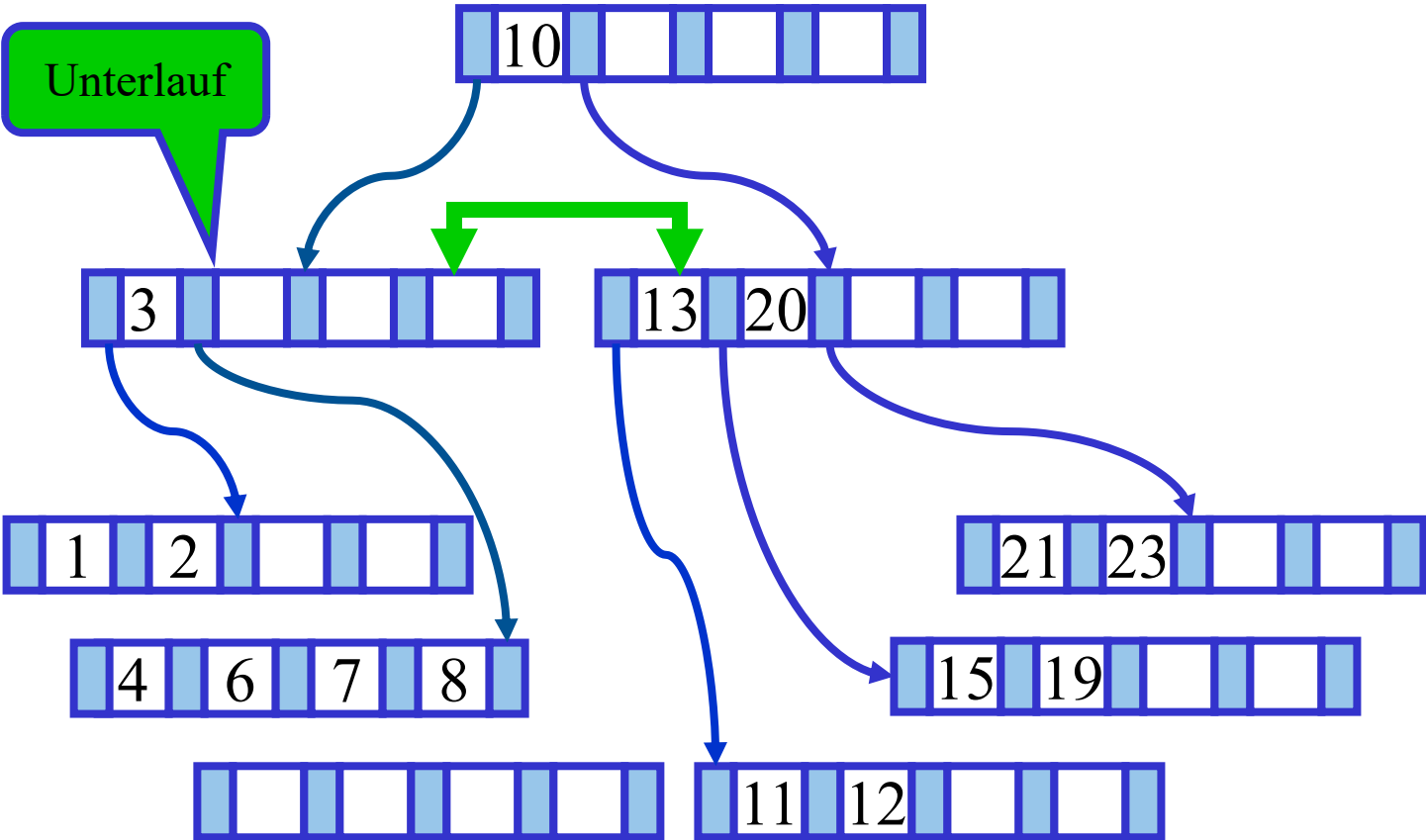
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



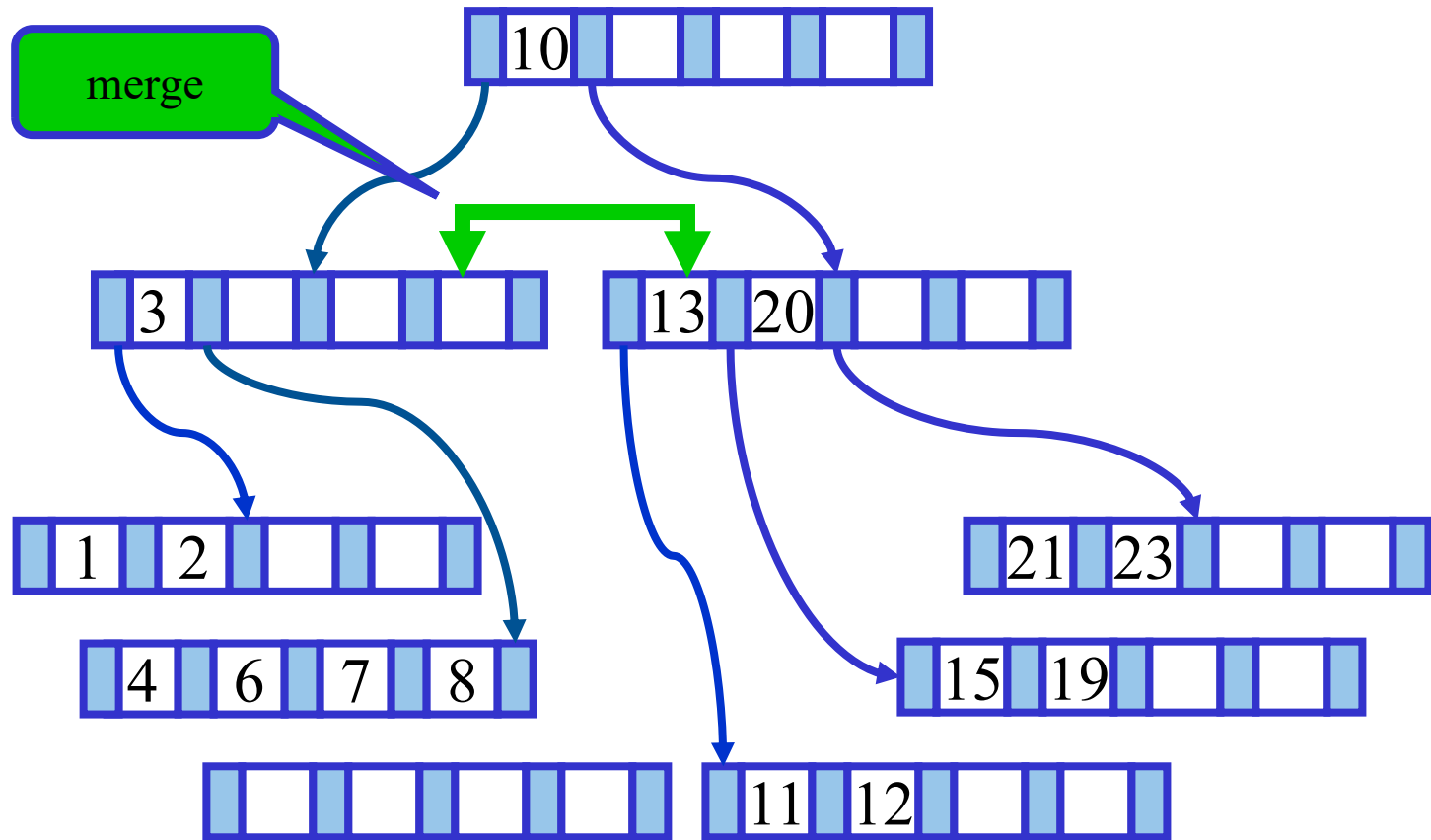
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



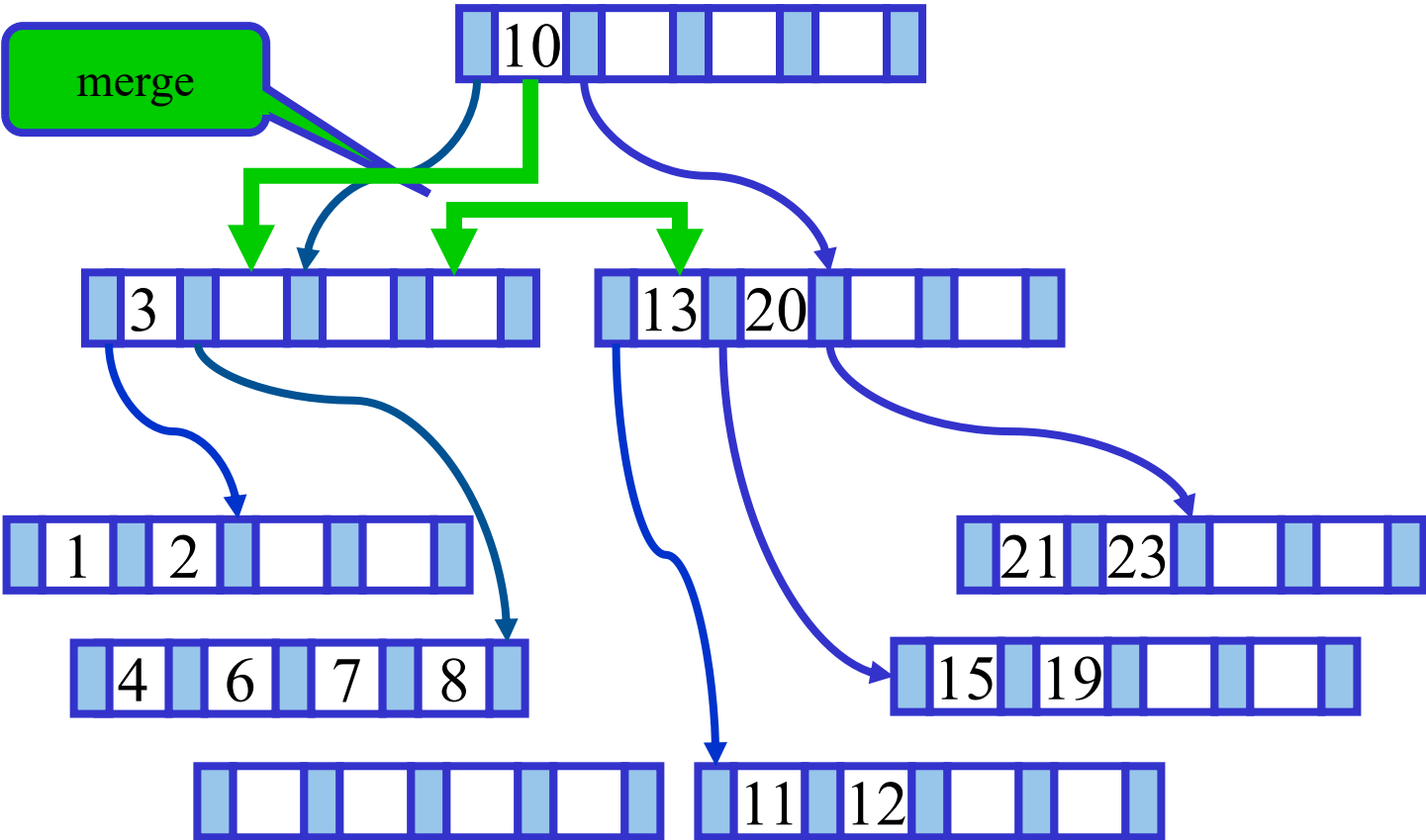
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



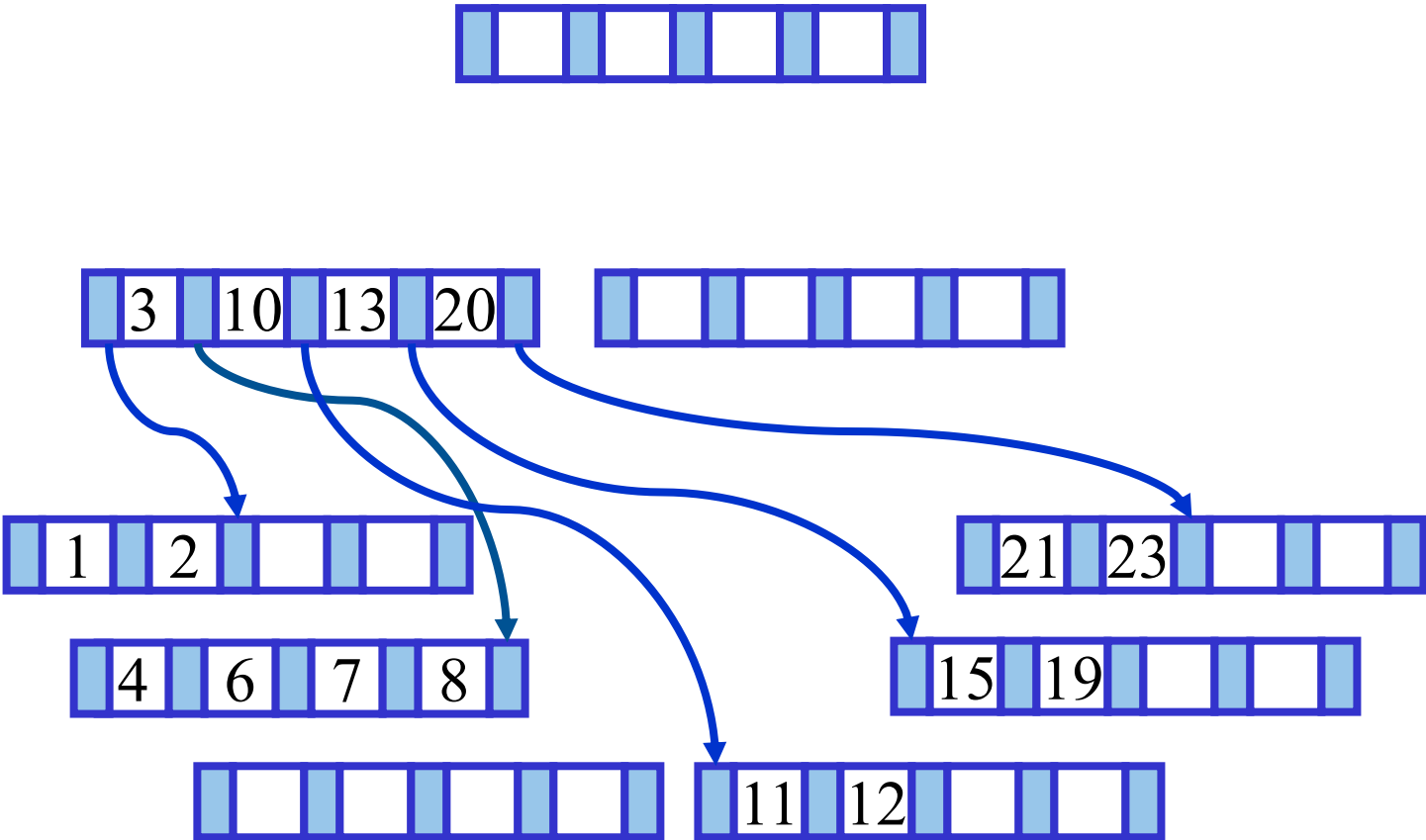
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



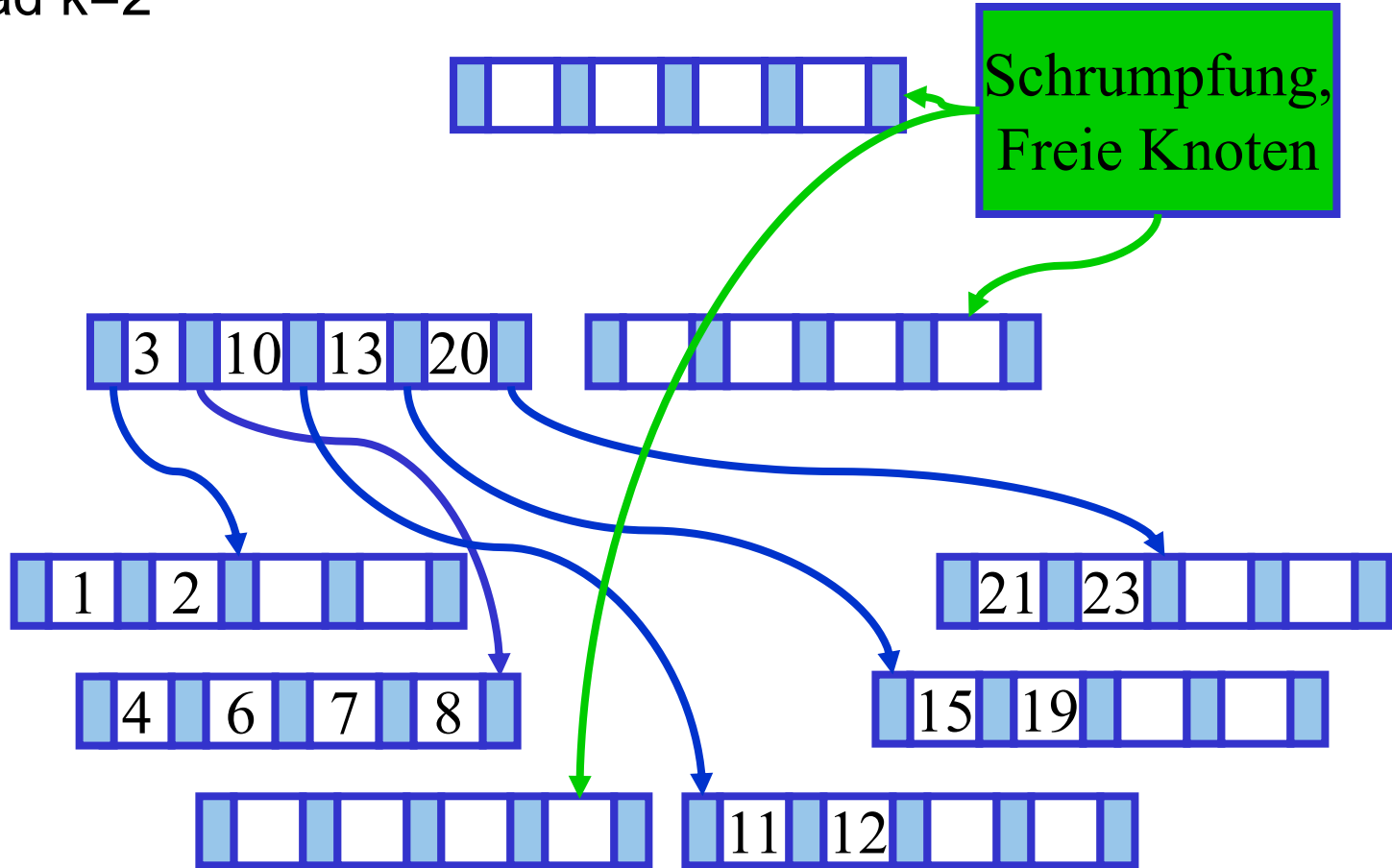
# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$

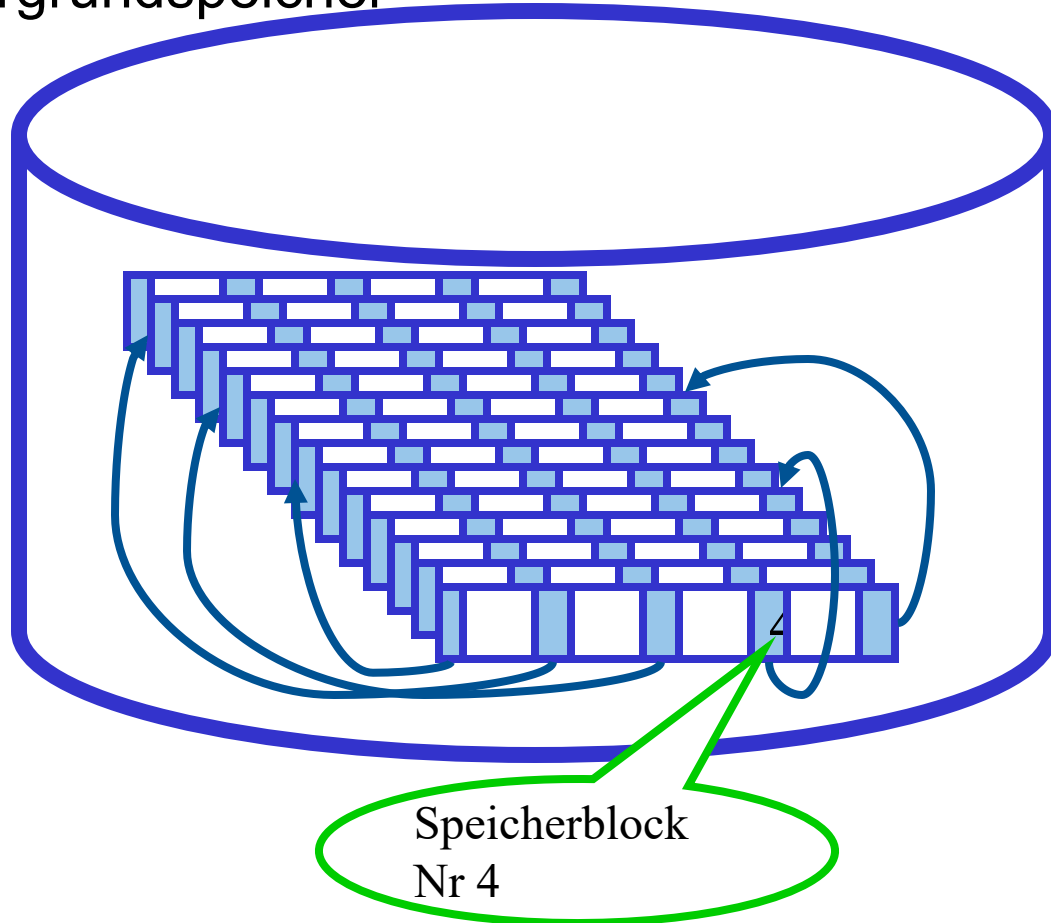


# Sukzessiver Aufbau eines B-Baums vom Grad $k=2$



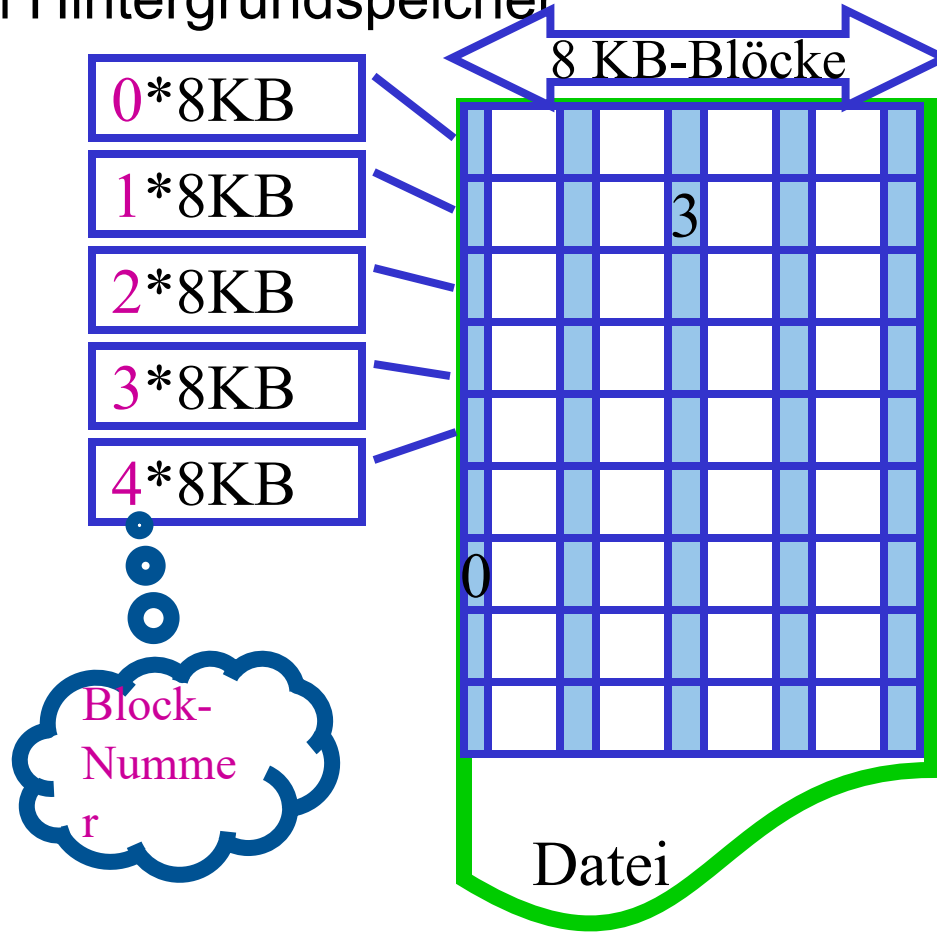


# Speicherstruktur eines B-Baums auf dem Hintergrundspeicher



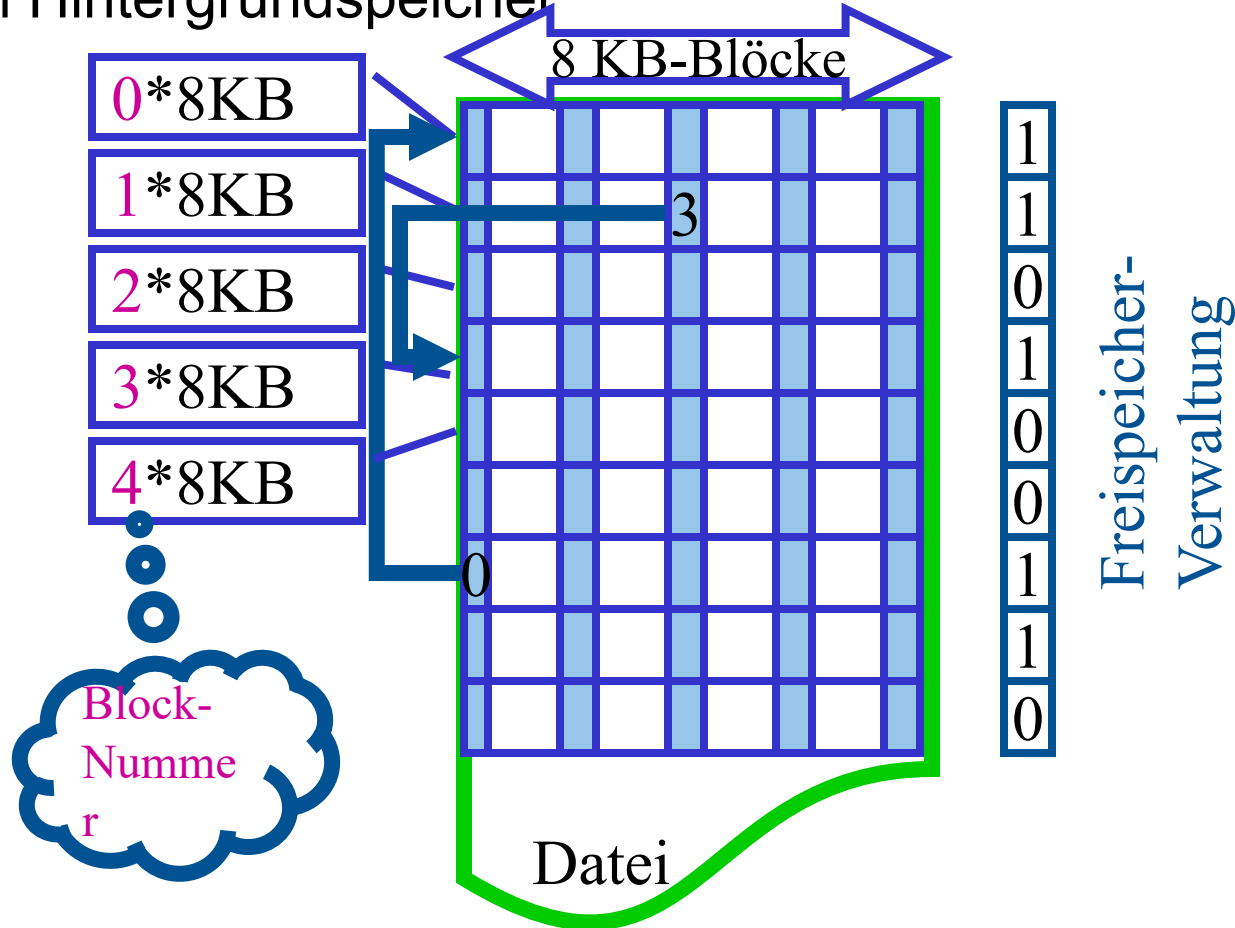
Speicherblock  
Nr 4

# Speicherstruktur eines B-Baums auf dem Hintergrundspeicher



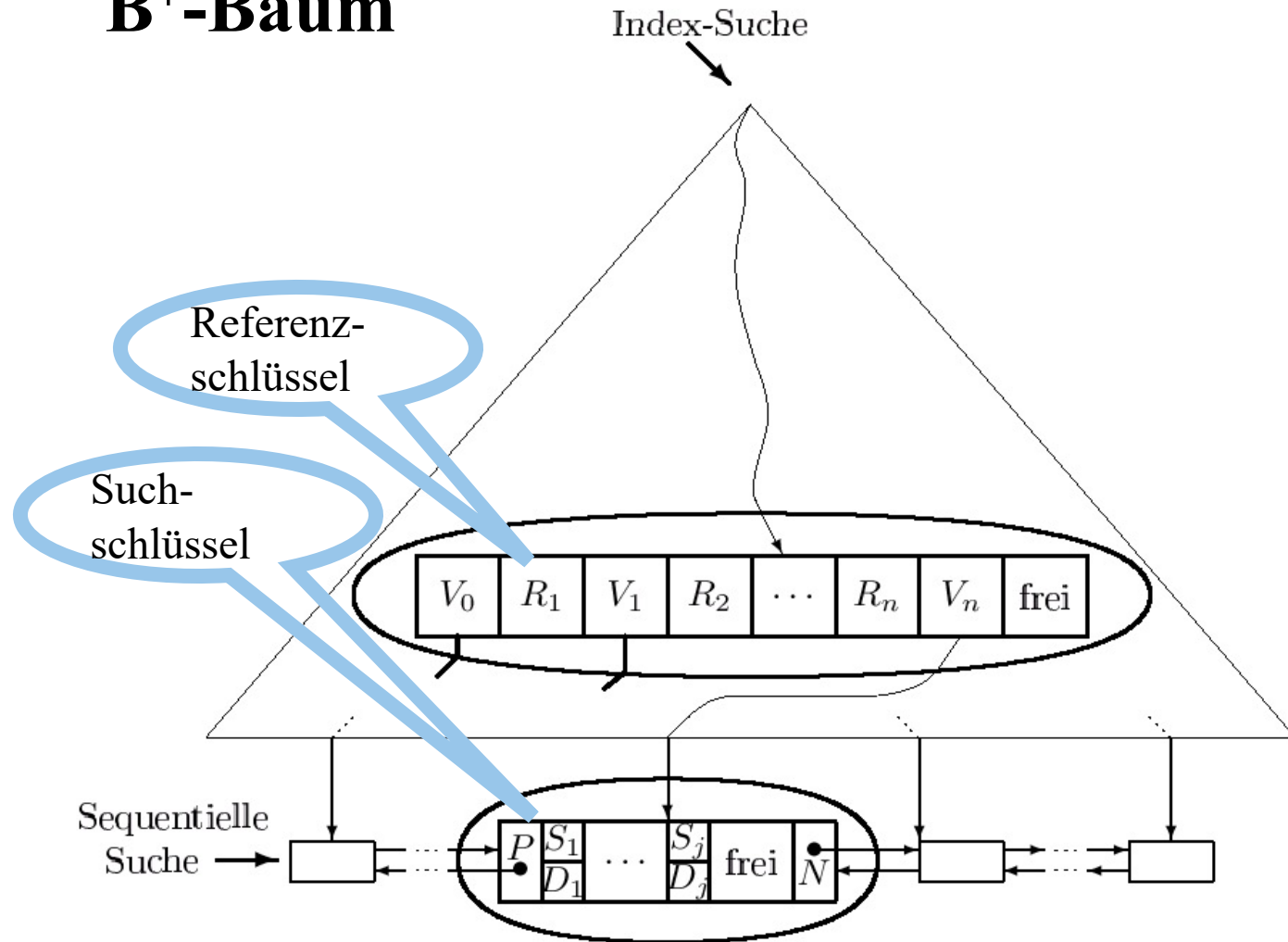


# Speicherstruktur eines B-Baums auf dem Hintergrundspeicher





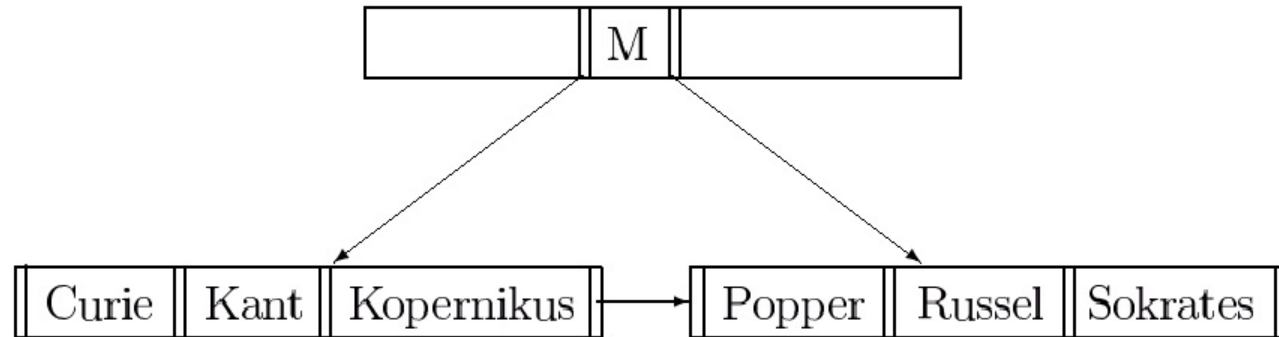
# B<sup>+</sup>-Baum



Ein  $B^+$ -Baum vom Typ  $(k, k^*)$  hat also folgende Eigenschaften:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge.
2. Jeder Knoten – außer Wurzeln und Blättern – hat mindestens  $k$  und höchstens  $2k$  Einträge. Blätter haben mindestens  $k^*$  und höchstens  $2k^*$  Einträge. Die Wurzel hat entweder maximal  $2k$  Einträge, oder sie ist ein Blatt mit maximal  $2k^*$  Einträgen.
3. Jeder Knoten mit  $n$  Einträgen, außer den Blättern, hat  $n + 1$  Kinder.
4. Seien  $R_1, \dots, R_n$  die Referenzschlüssel eines inneren Knotens (d.h. auch der Wurzel) mit  $n + 1$  Kindern. Seien  $V_0, V_1, \dots, V_n$  die Verweise auf diese Kinder.
  - (a)  $V_0$  verweist auf den Teilbaum mit Schlüsseln kleiner oder gleich  $R_1$ .
  - (b)  $V_i$  ( $i = 1, \dots, n - 1$ ) verweist auf den Teilbaum, dessen Schlüssel zwischen  $R_i$  und  $R_{i+1}$  liegen (einschließlich  $R_{i+1}$ ).
  - (c)  $V_n$  verweist auf den Teilbaum mit Schlüsseln größer als  $R_n$ .

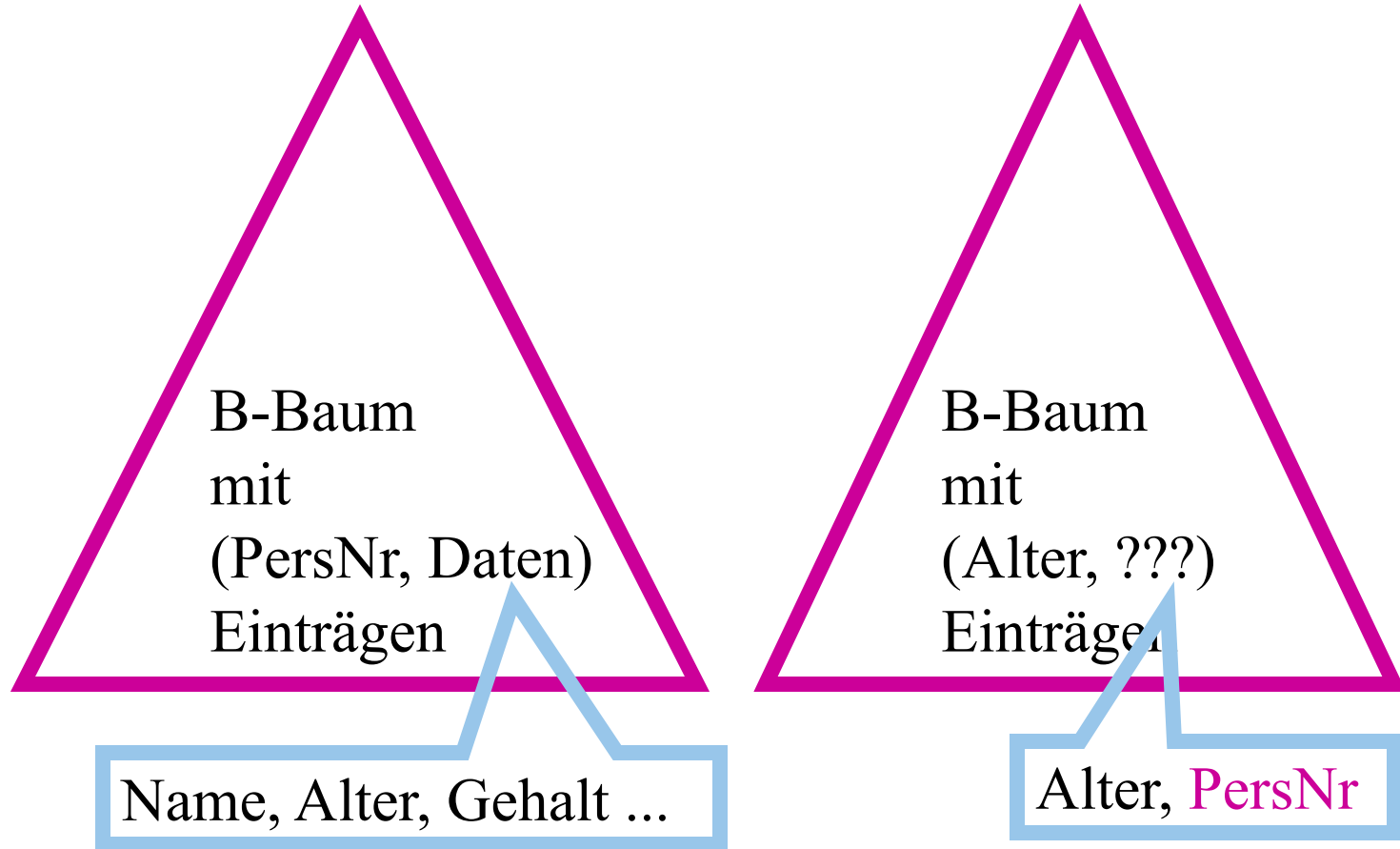
- Es werden nur *Referenzschlüssel* benötigt.



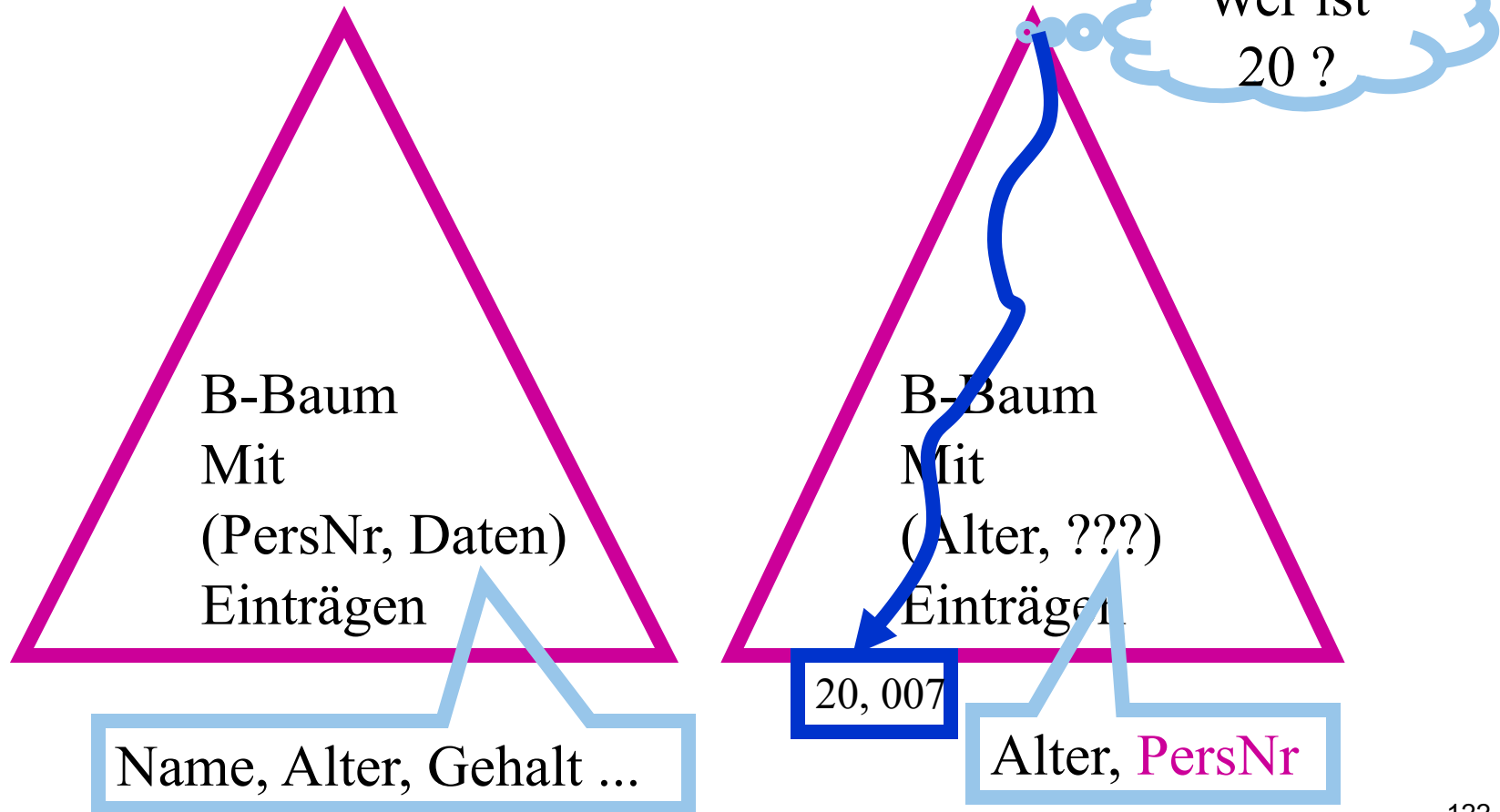
- beliebiger Referenzschlüssel  $R$  mit  
 $\text{Kopernikus} \leq R < \text{Popper}$



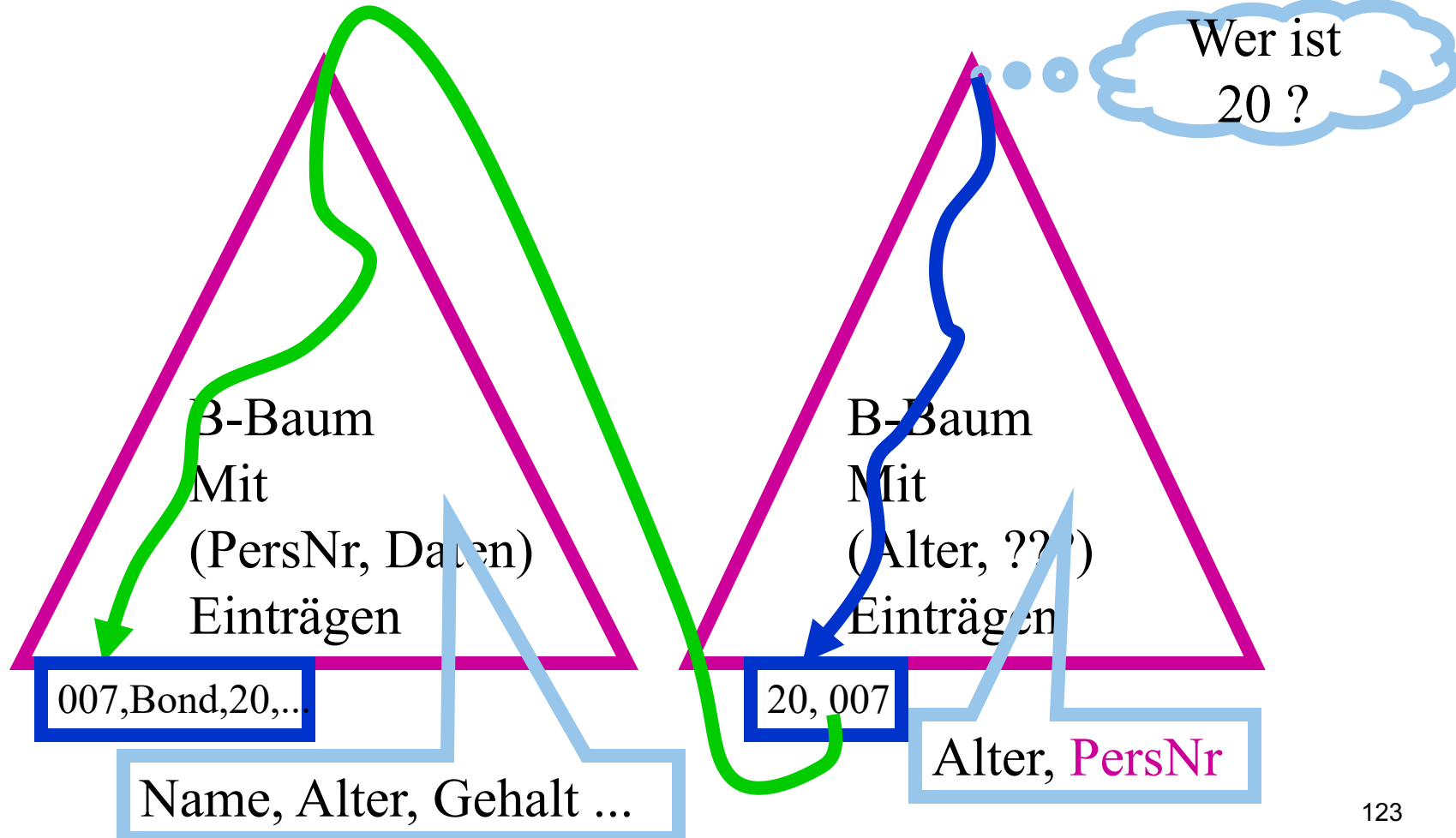
# Mehrere Indexe auf denselben Objekten



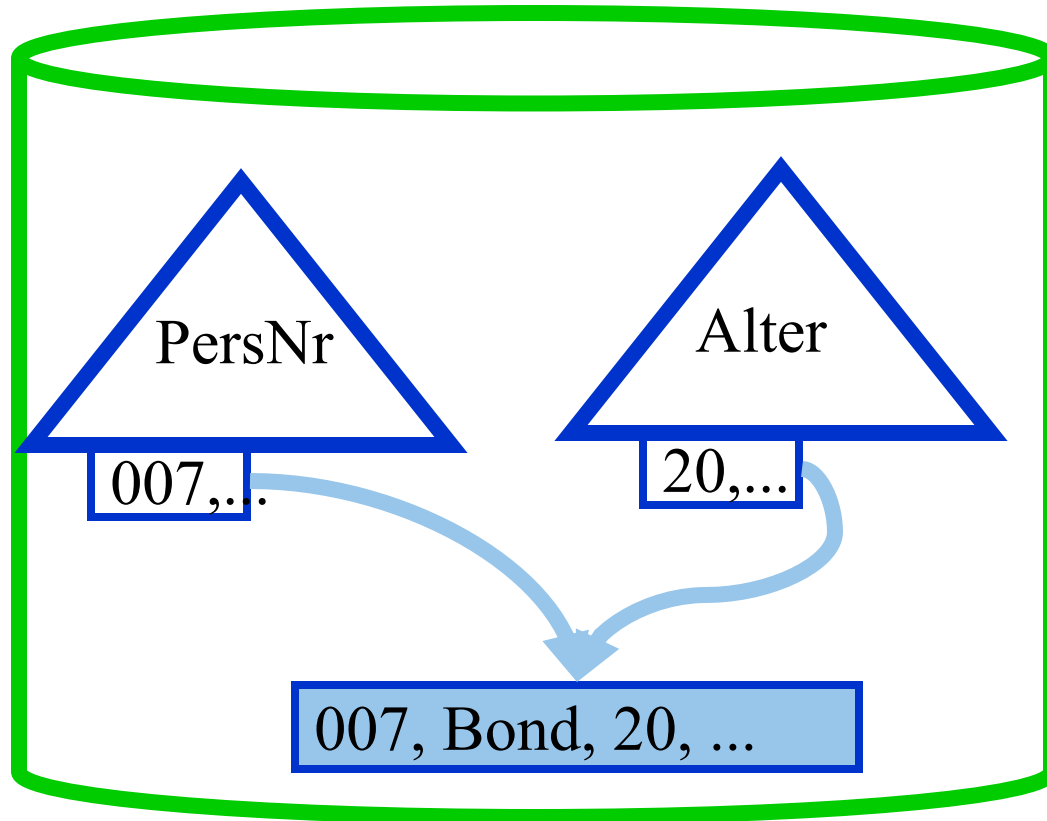
## Mehrere Indexe auf denselben Objekten



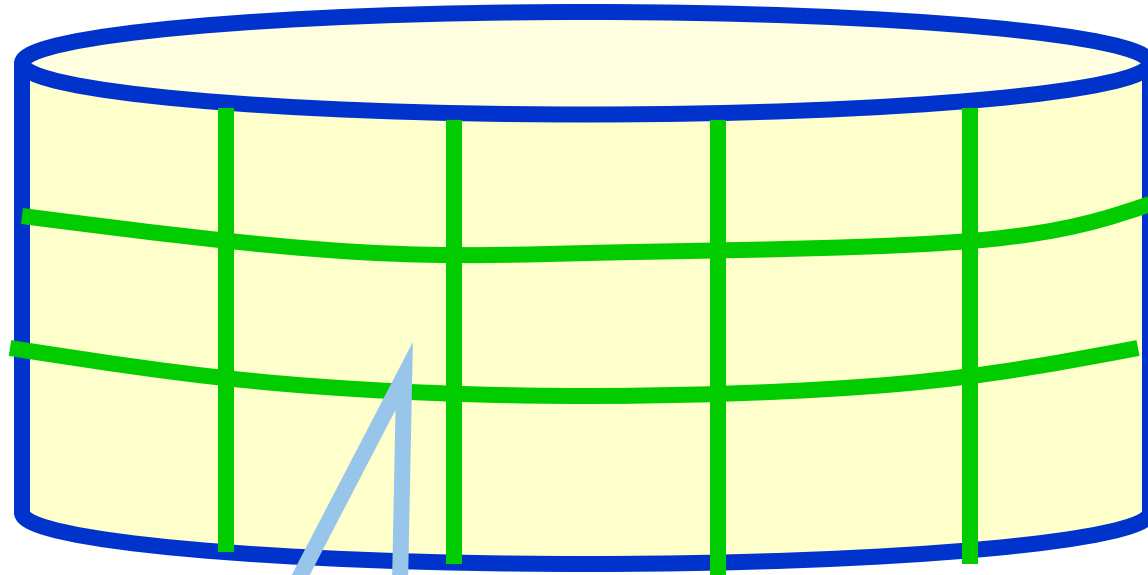
## Mehrere Indexe auf denselben Objekten



# Eine andere Möglichkeit: Referenzierung über Speicheradressen

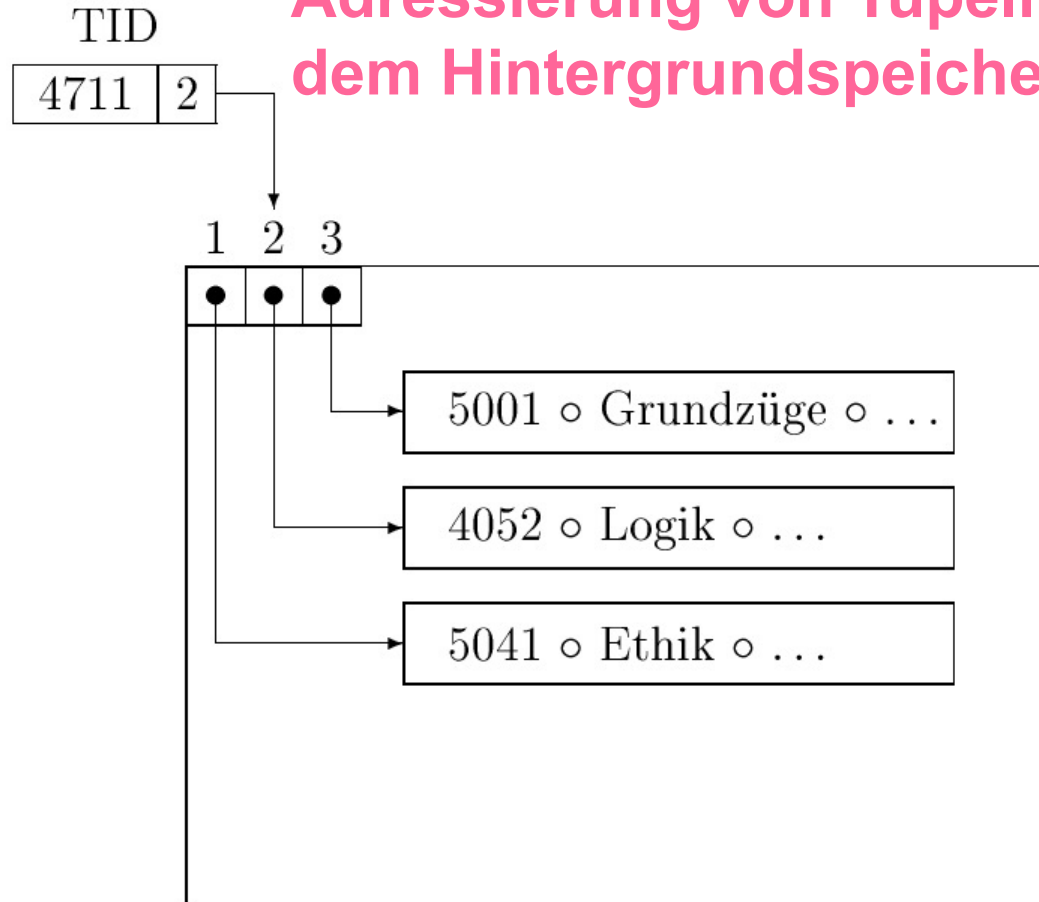


# Realisierungstechnik für Hintergrundspeicher-Adressen



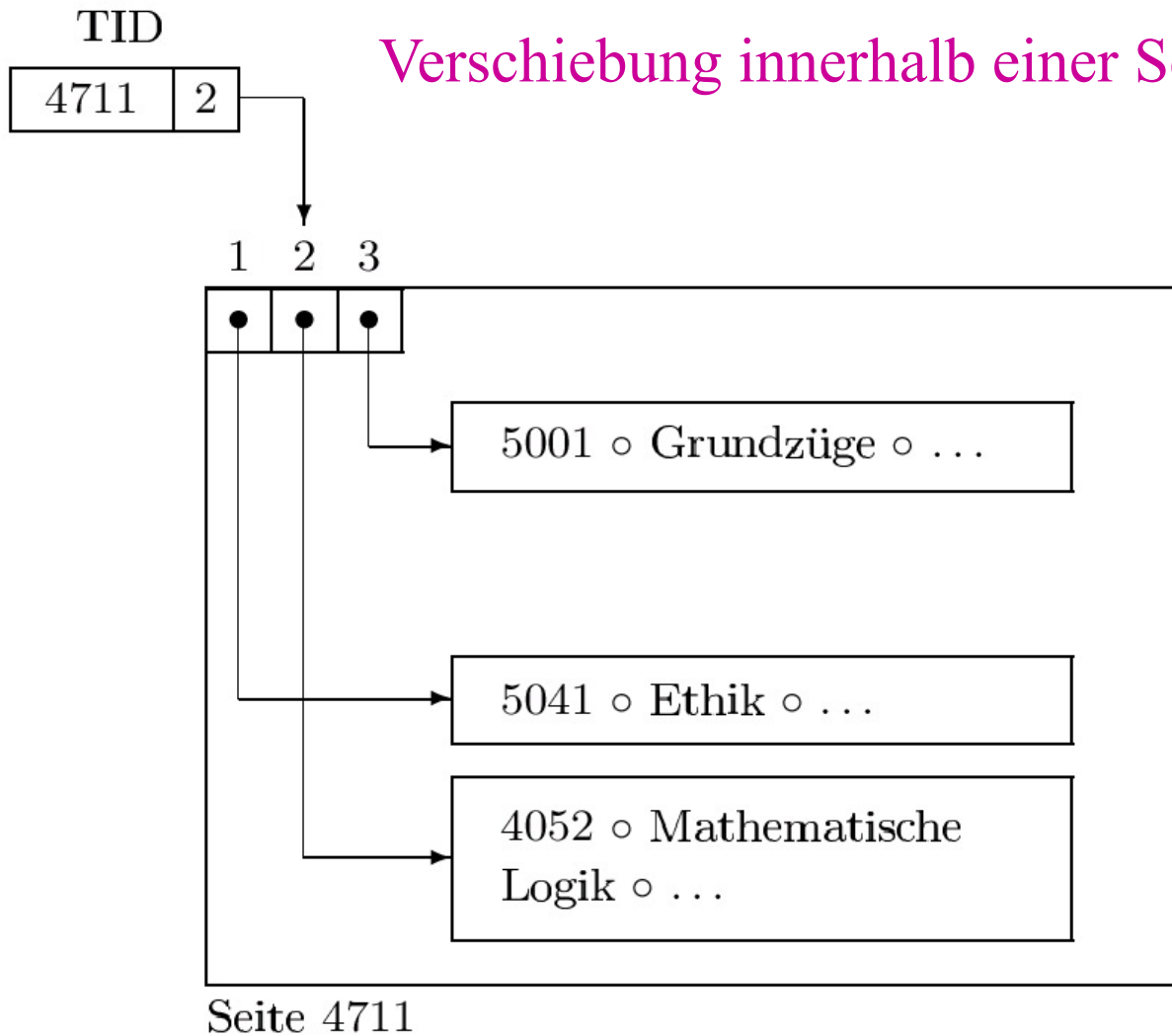
Seiten / Blöcke  
(ca 8 KB)

# Adressierung von Tupeln auf dem Hintergrundspeicher

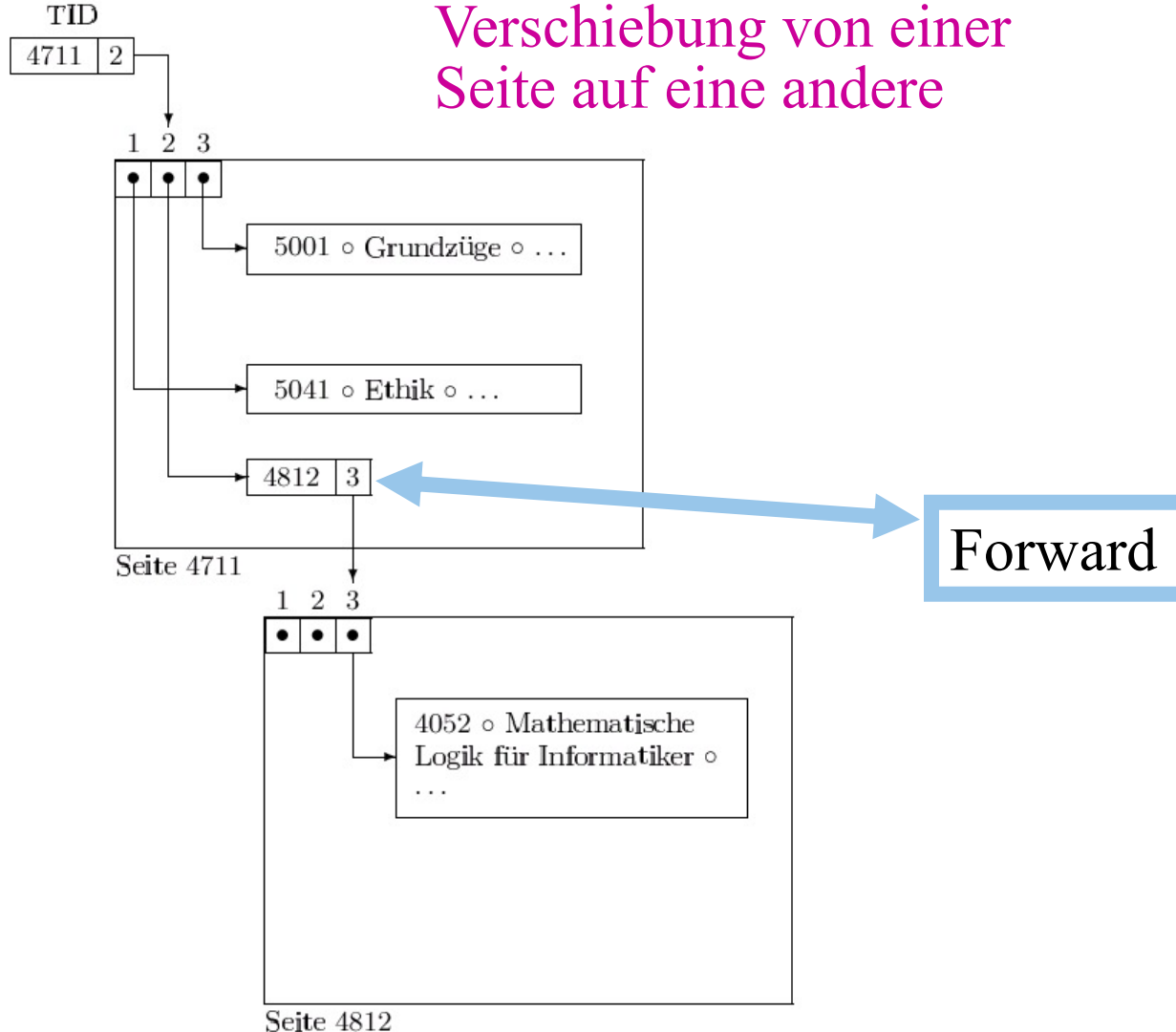


Seite 4711

## Verschiebung innerhalb einer Seite

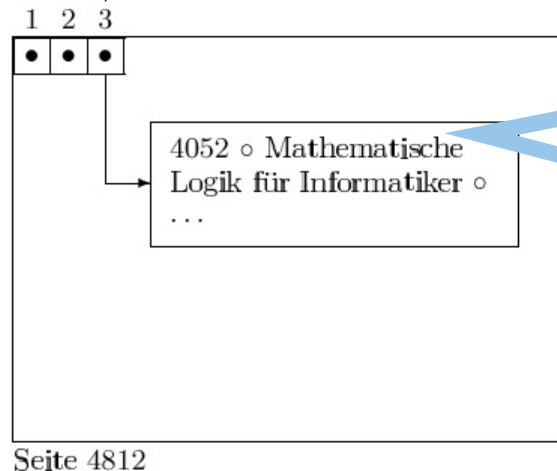
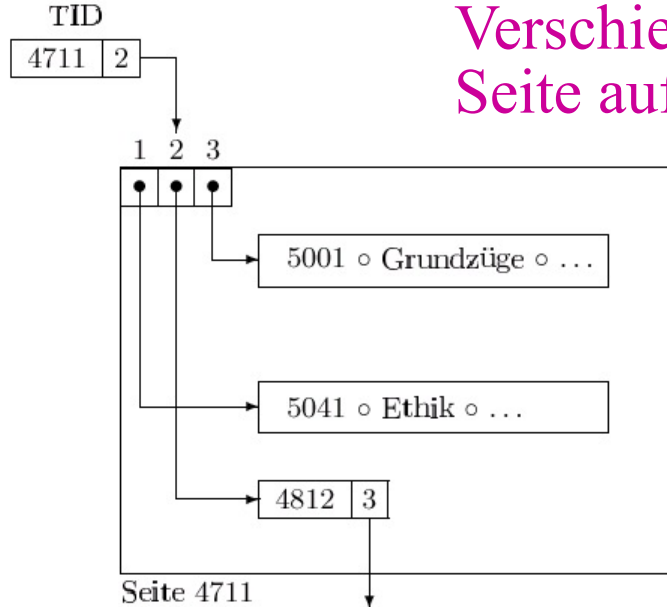


# Verschiebung von einer Seite auf eine andere





# Verschiebung von einer Seite auf eine andere



Bei der nächsten Verschiebung wird der „Forward“ auf Seite 4711 geändert (kein Forward auf Seite 4812)

# „Statische“ Hashtabellen

À priori Allokation des Speichers

Nachträgliche Vergrößerung der Hashtabelle ist „teuer“

- Hashfunktion  $h(\dots) = \dots \bmod N$
- Rehashing der Einträge
  - $h(\dots) = \dots \bmod M$
- In Datenbankanwendungen viele GB

Erweiterbares Hashing

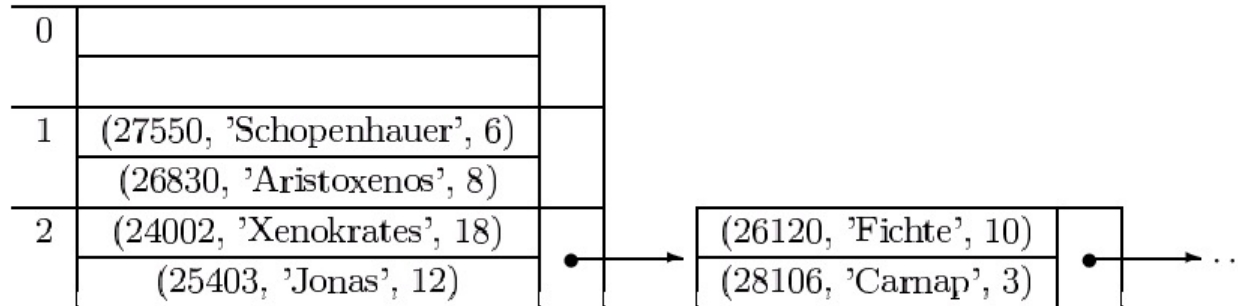
- Zusätzliche Indirektion über ein Directory
- Ein zusätzlicher Zugriff auf ein Directory, das den Zeiger (Verweis, BlockNr) des Hash-Bucket enthält
- Dynamisches Wachsen (und Schrumpfen) ist möglich
- Der Zugriff auf das Directory erfolgt über einen binären Hashcode



- Hashfunktion  $h(x) = x \bmod 3$

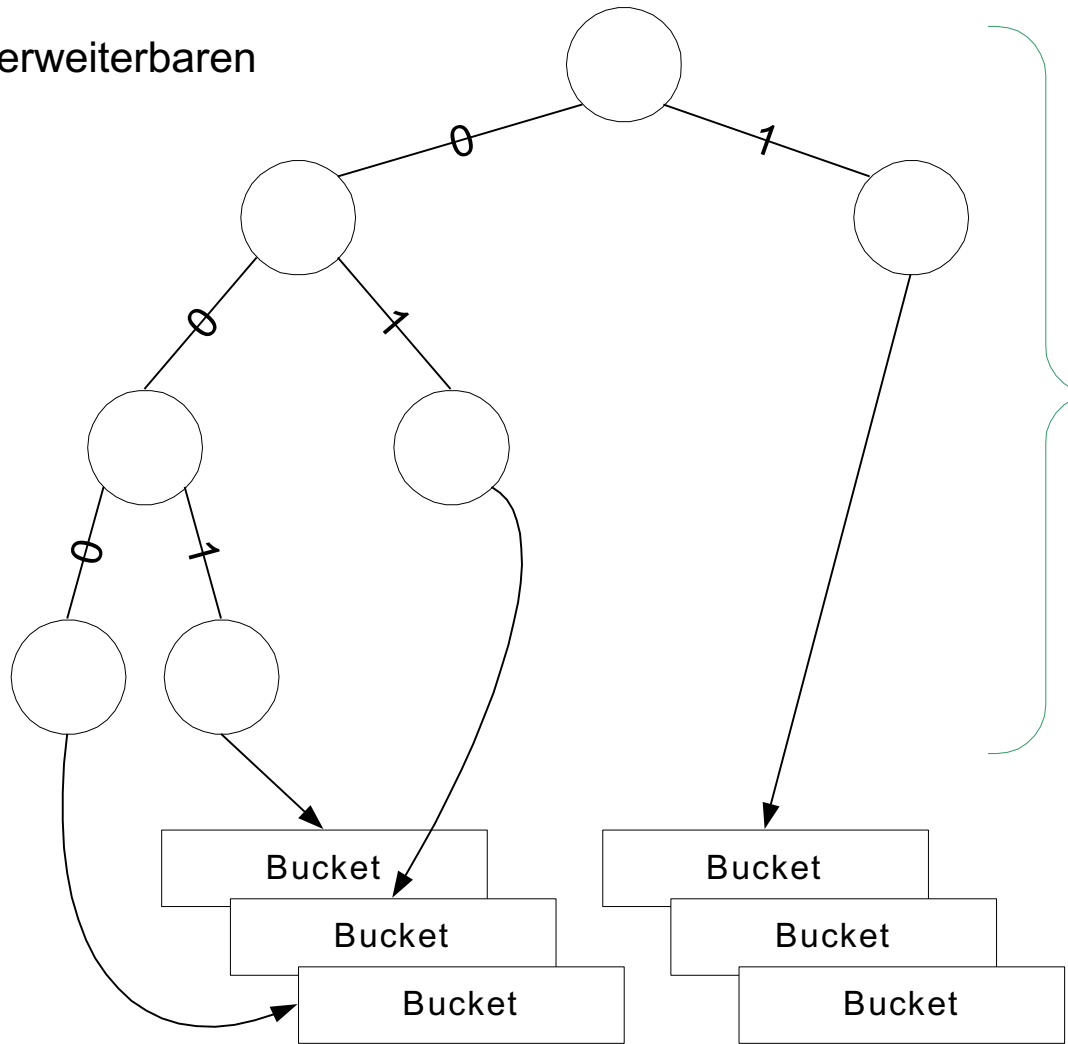
0	
1	(27550, 'Schopenhauer', 6)
2	(24002, 'Xenokrates', 18) (25403, 'Jonas', 12)

- Kollisionsbehandlung



⇒ ineffizient bei nicht vorhersehbarer Datenmenge

# Grundidee des erweiterbaren Hashings



binärer  
Trie,  
Entscheidungs-  
baum  
Directory

# Hashfunktion für erweiterbares Hashing

$h$ : Schlüsselmenge  $\rightarrow \{0,1\}^*$

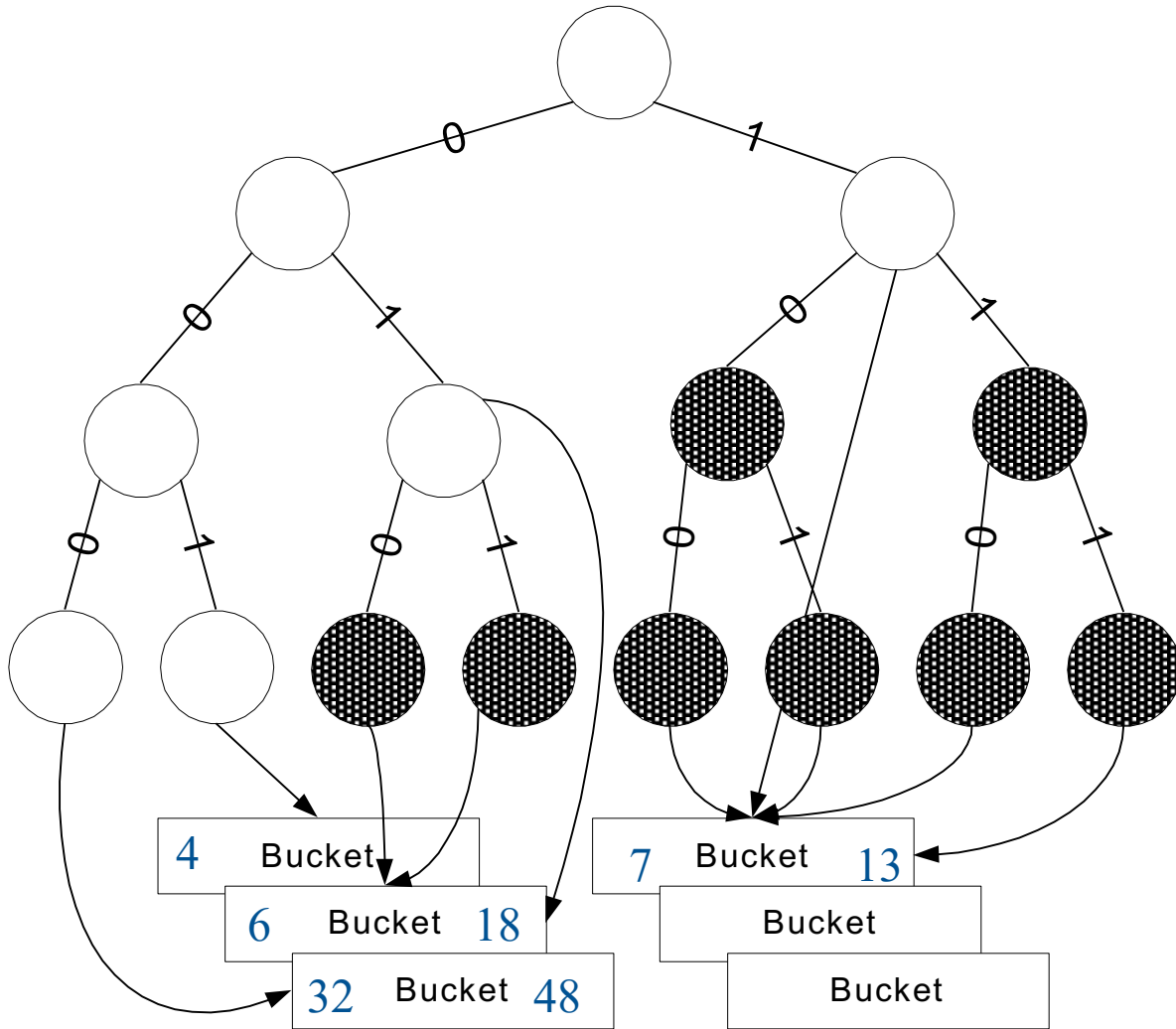
Der Bitstring muss lang genug sein, um alle Objekte auf ihre Buckets abbilden zu können

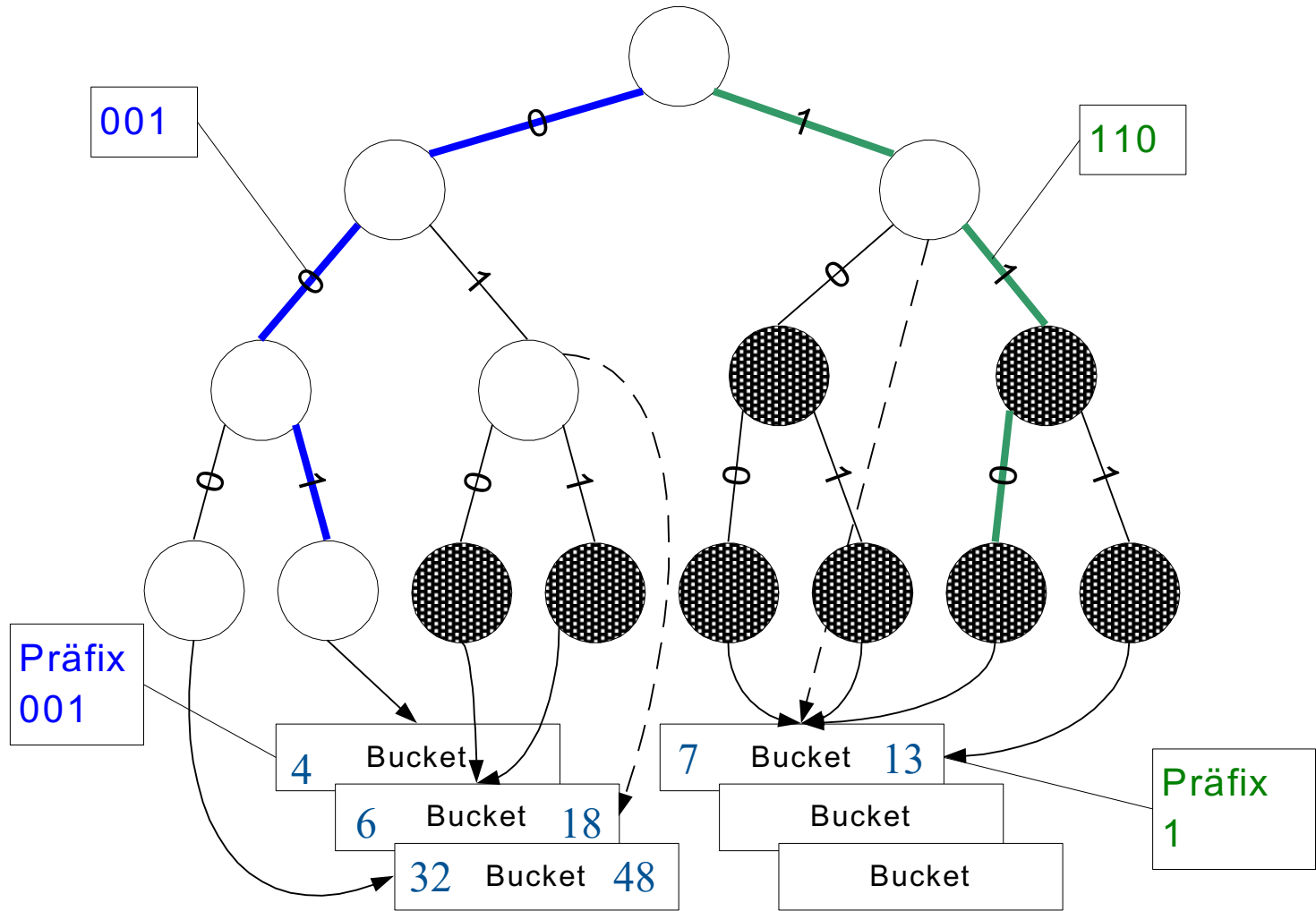
Anfangs wird nur ein (kurzer) Präfix des Hashwertes (Bitstrings) benötigt

Wenn die Hashtabelle wächst wird aber sukzessive ein längerer Präfix benötigt

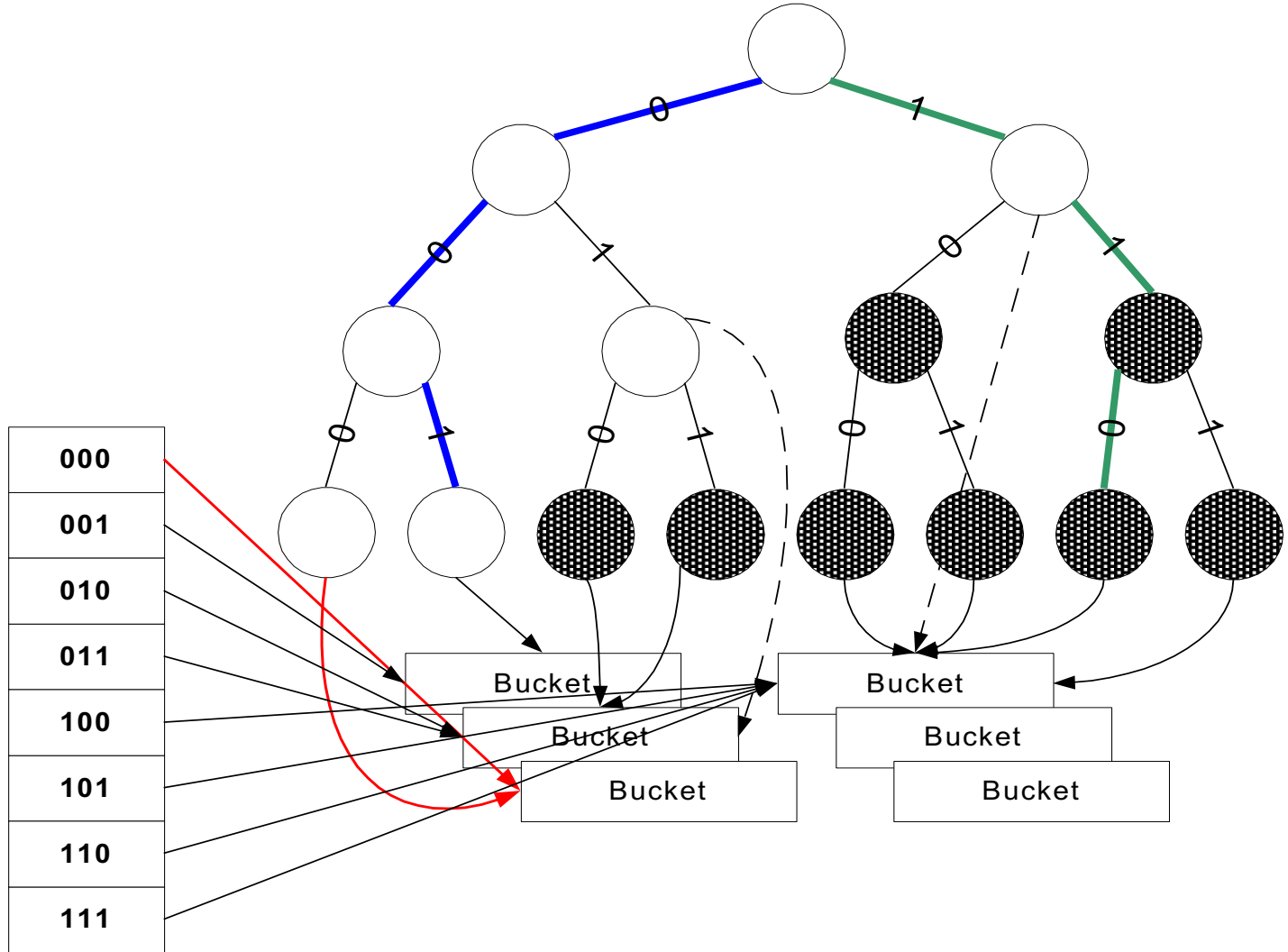
Beispiel-Hashfunktion: gespiegelte binäre PersNr

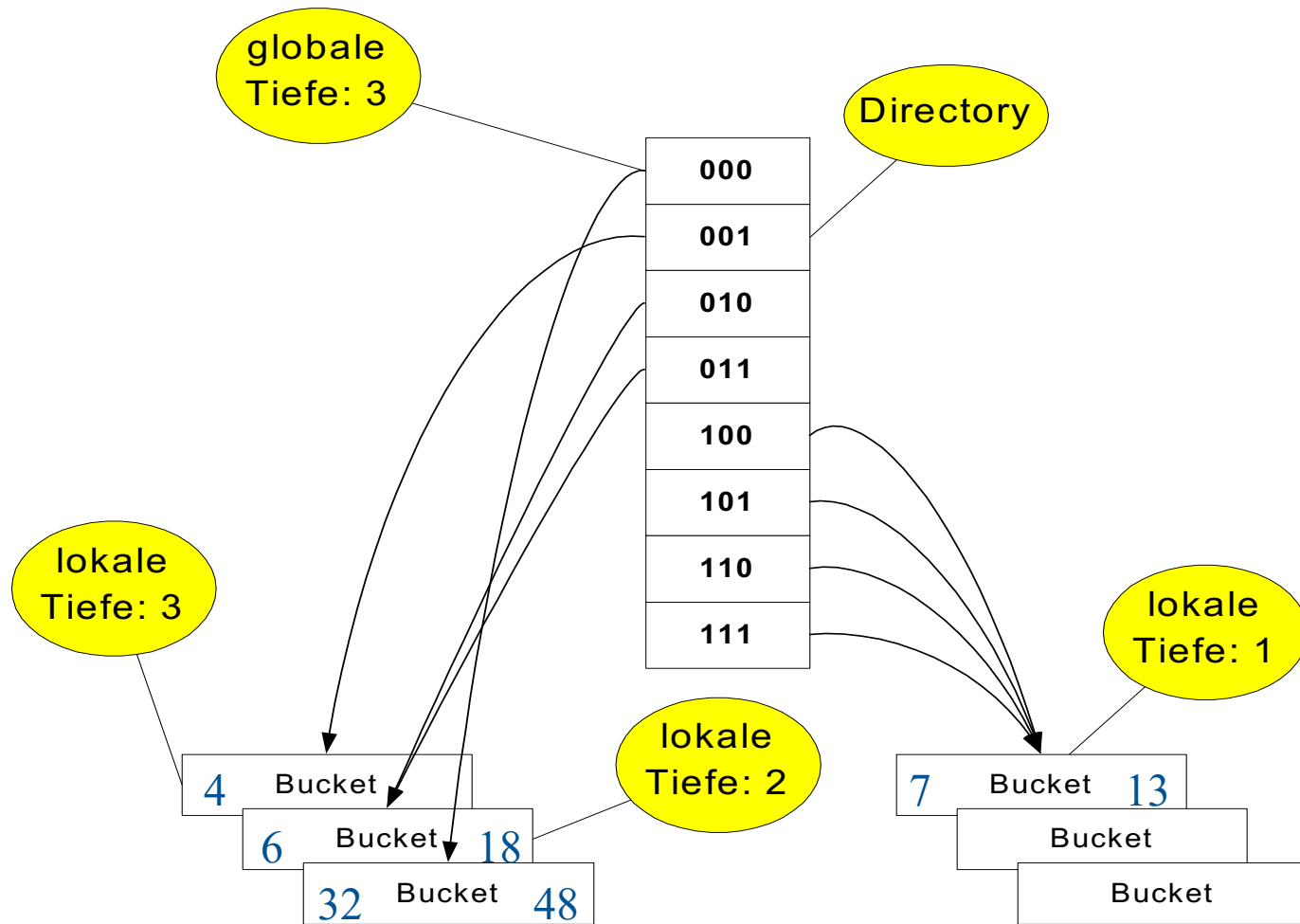
- $h(004) = 001000000\dots$  (4=0..0100)
- $h(006) = 011000000\dots$  (6=0..0110)
- $h(007) = 111000000\dots$  (7 =0..0111)
- $h(013) = 101100000\dots$  (13 =0..01101)
- $h(018) = 0100100000\dots$  (18 =0..010010)
- $h(032) = 000001000\dots$  (32 =0..0100000)
- $H(048) = 000011000\dots$  (48 =0..0110000)

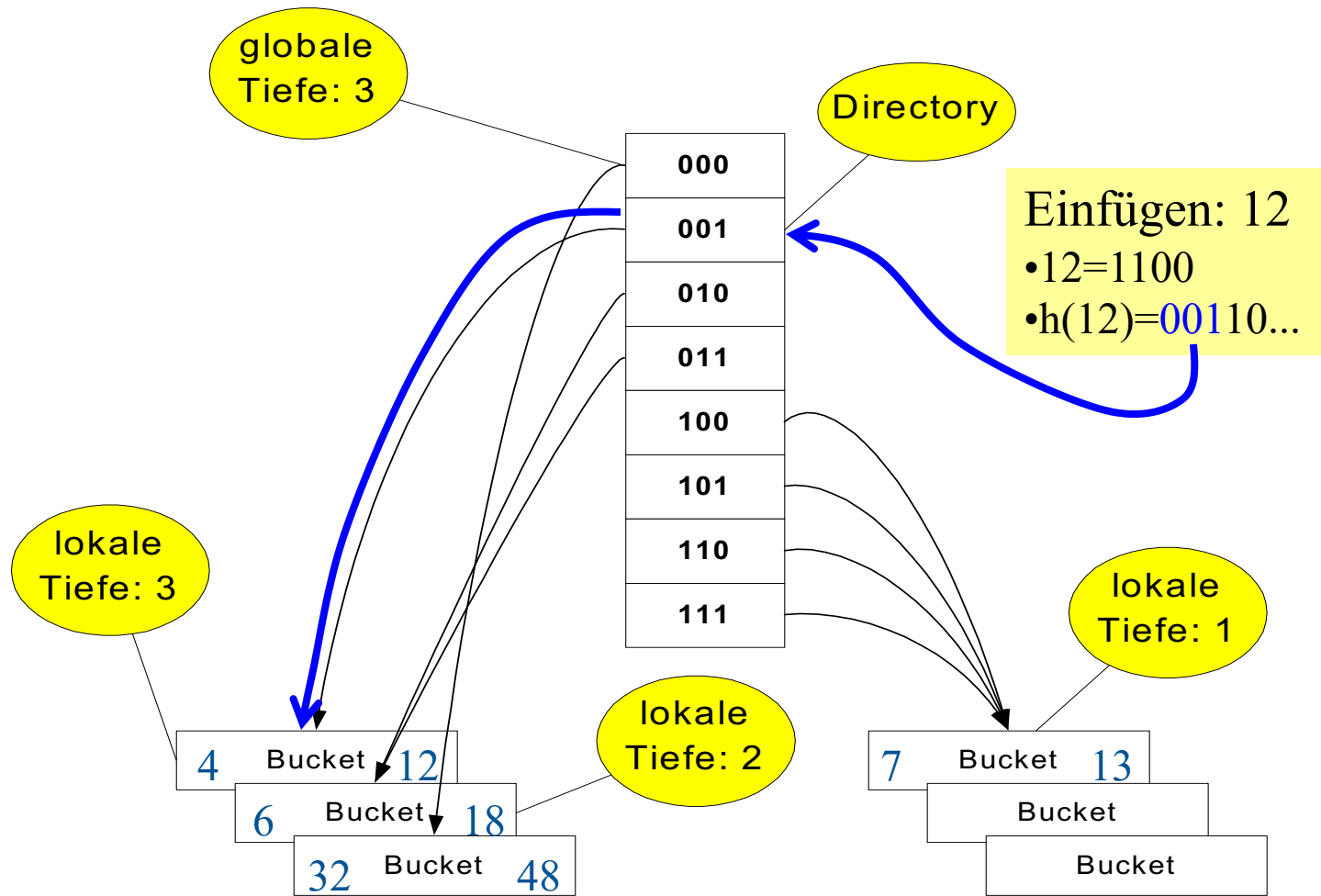


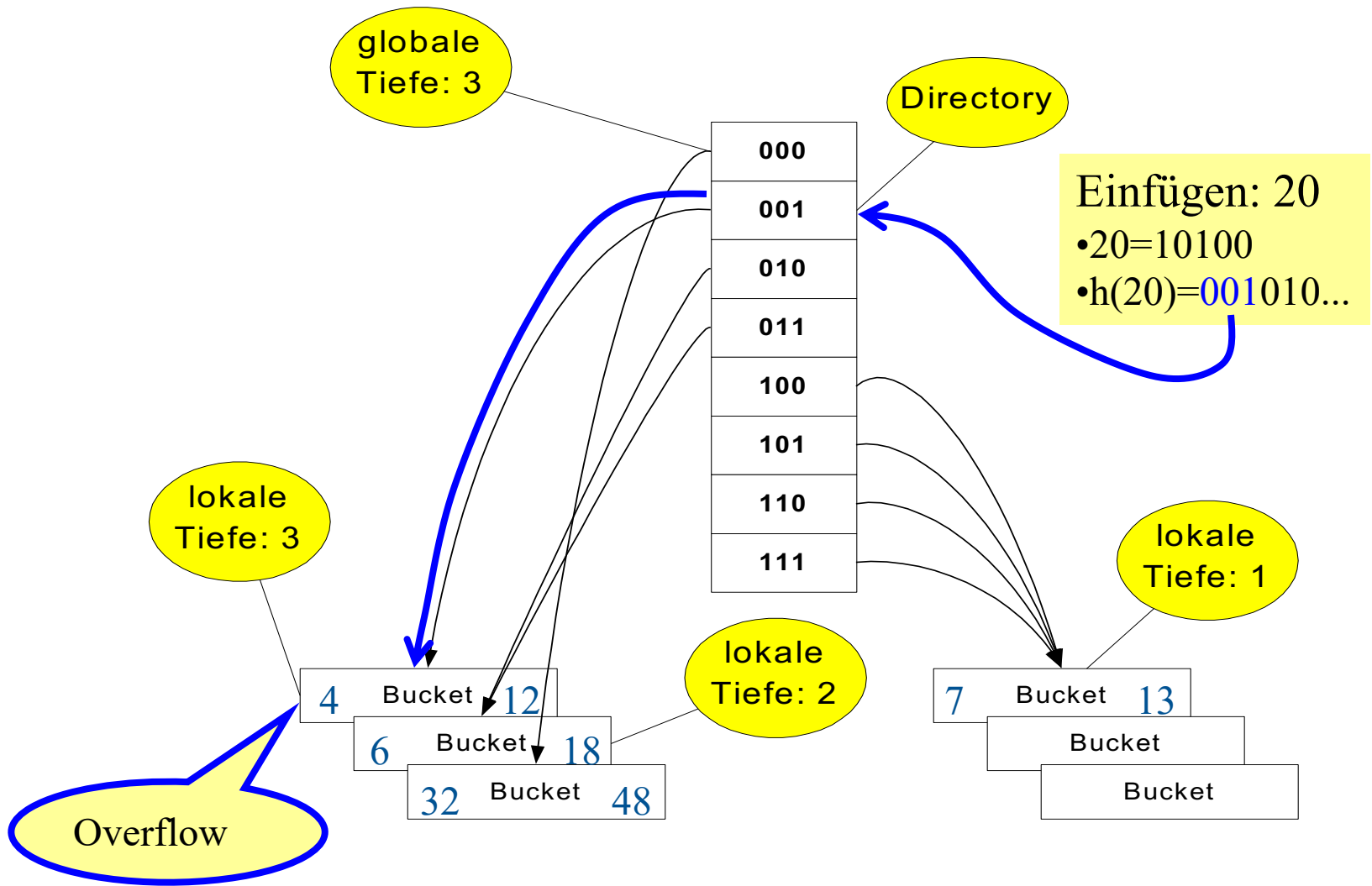


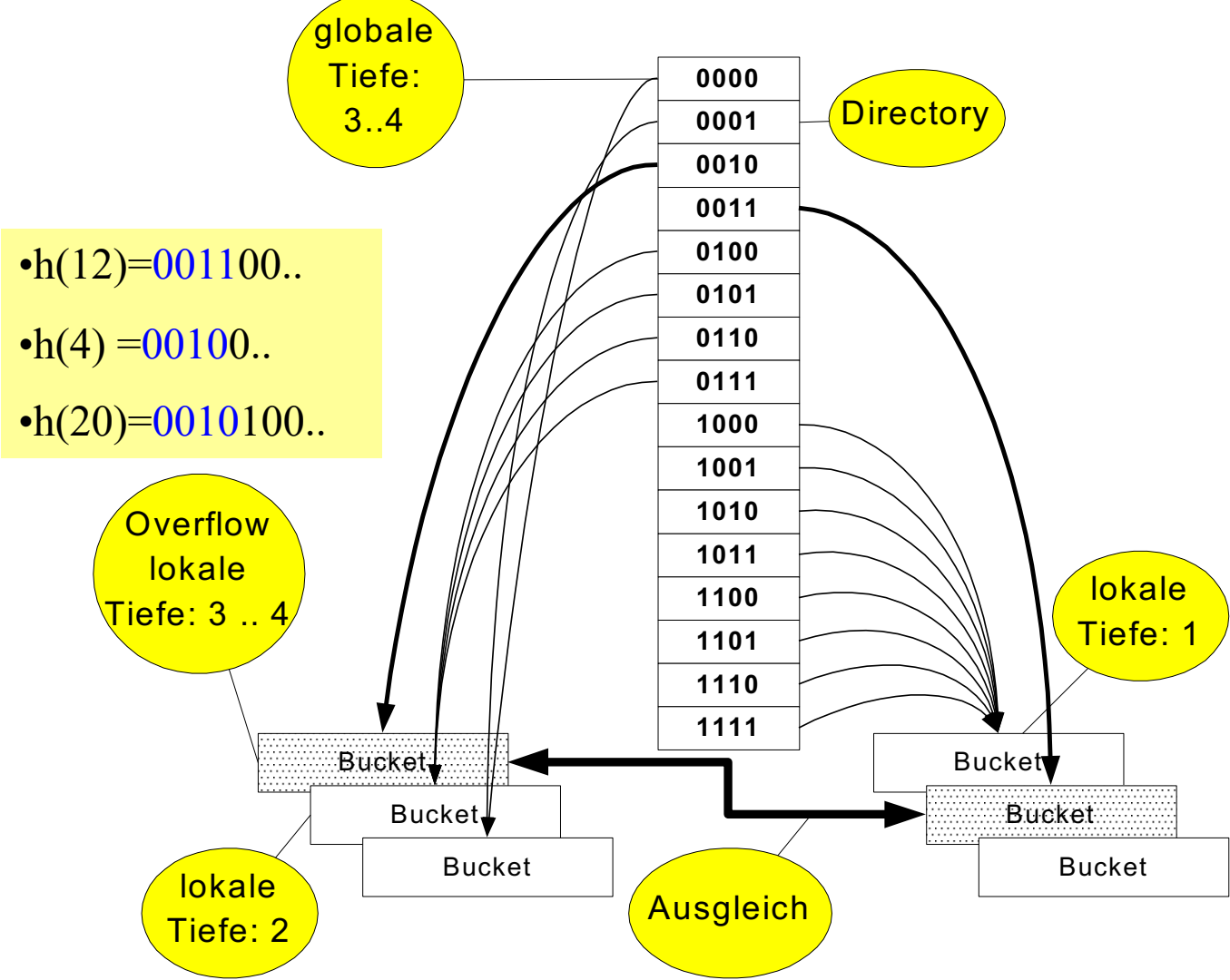




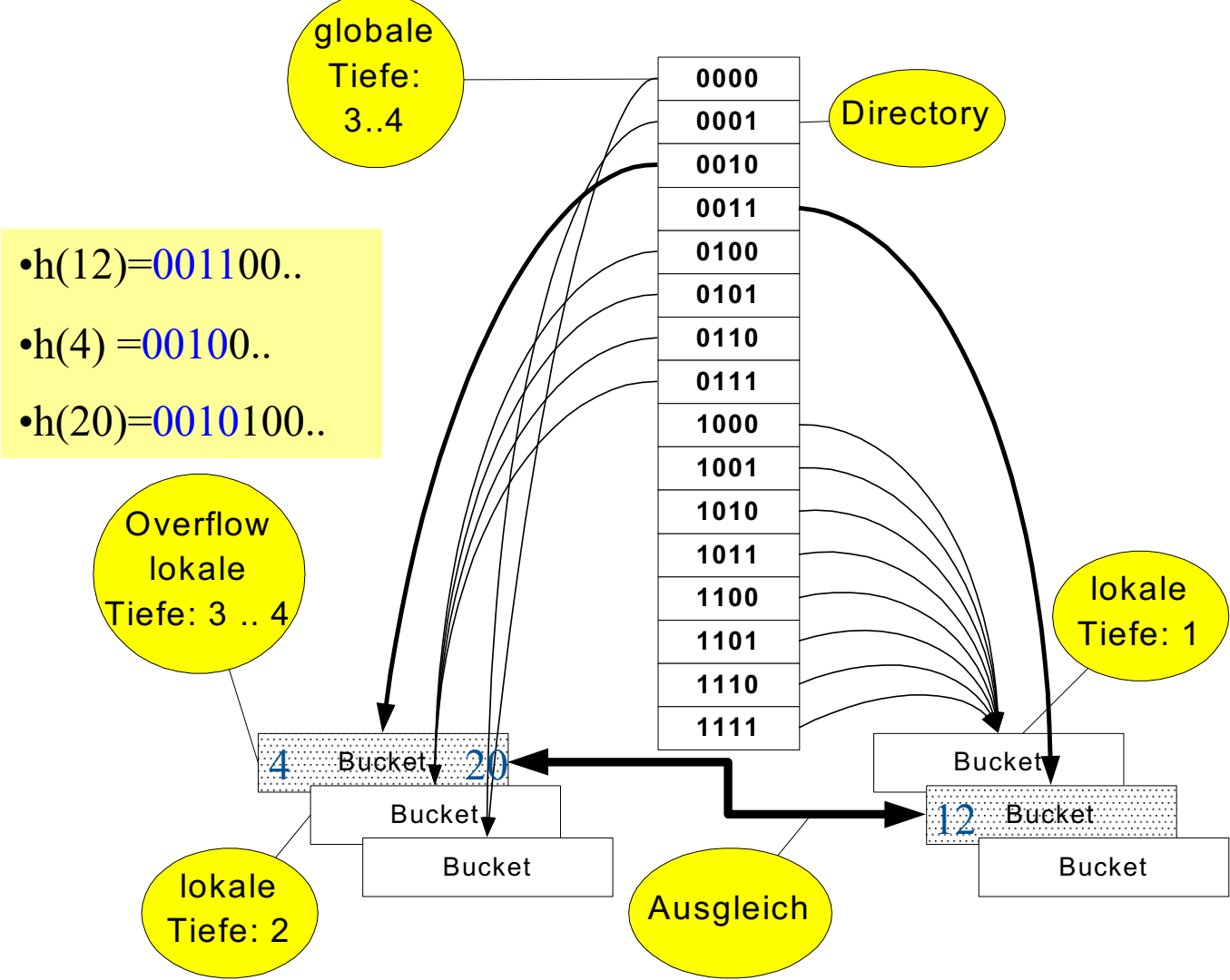


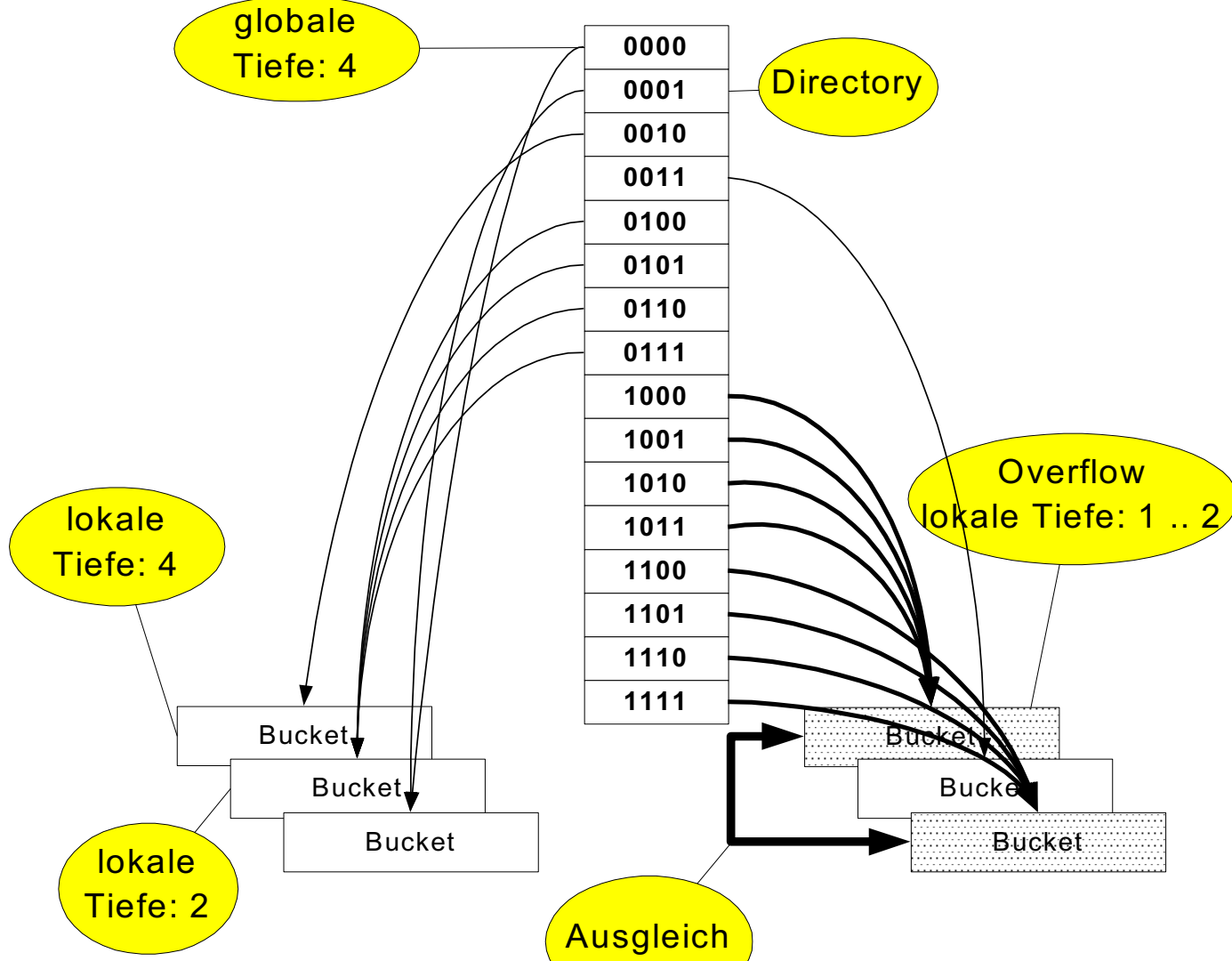






•h(12)=001100..  
 •h(4) =00100..  
 •h(20)=0010100..

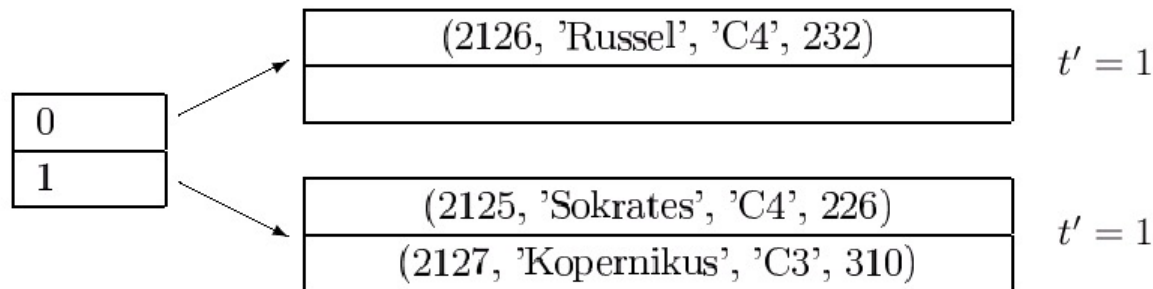




## Demonstration des erweiterbaren Hashings

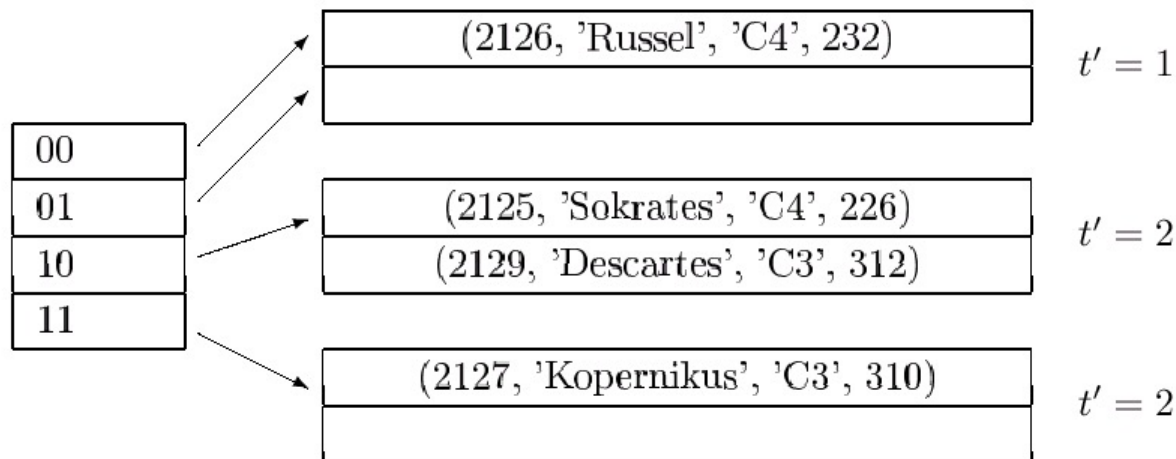
---

$x$	$h(x)$	
	$d$	$p$
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001





$x$	$h(x)$	
	$d$	$p$
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001





# Mehrdimensionale Datenstrukturen

Wertbasierter Zugriff auf der Grundlage mehrerer Attribute, dies einzeln oder in beliebigen Kombinationen.

Typische Anforderungen aus CAD, VLSI-Entwurf, Kartographie,...

Anfragen decken den Bereich ab zwischen

- mehrdimensionalem Punktzugriff (EMQ) und
- mehrdimensionalen Bereichsanfragen (RQ)

Lösung mit eindimensionalen Indexen

- erfordert konjunktive Zerlegung der Anfrage in Einattributanfragen und Schnittmengenbildung
- bedingt hohe Speicherredundanz

Problemstellung:

- Mehrdimensionale Nachbarschaftsverhältnisse

# Grundlagen mehrdimensionaler Datenstrukturen

Wertebereiche  $D_0, \dots, D_{k-1}$ :

alle  $D_i$  sind endlich, linear geordnet und besitzen kleinstes ( $-\infty_i$ ) und größtes ( $\infty_i$ ) Element

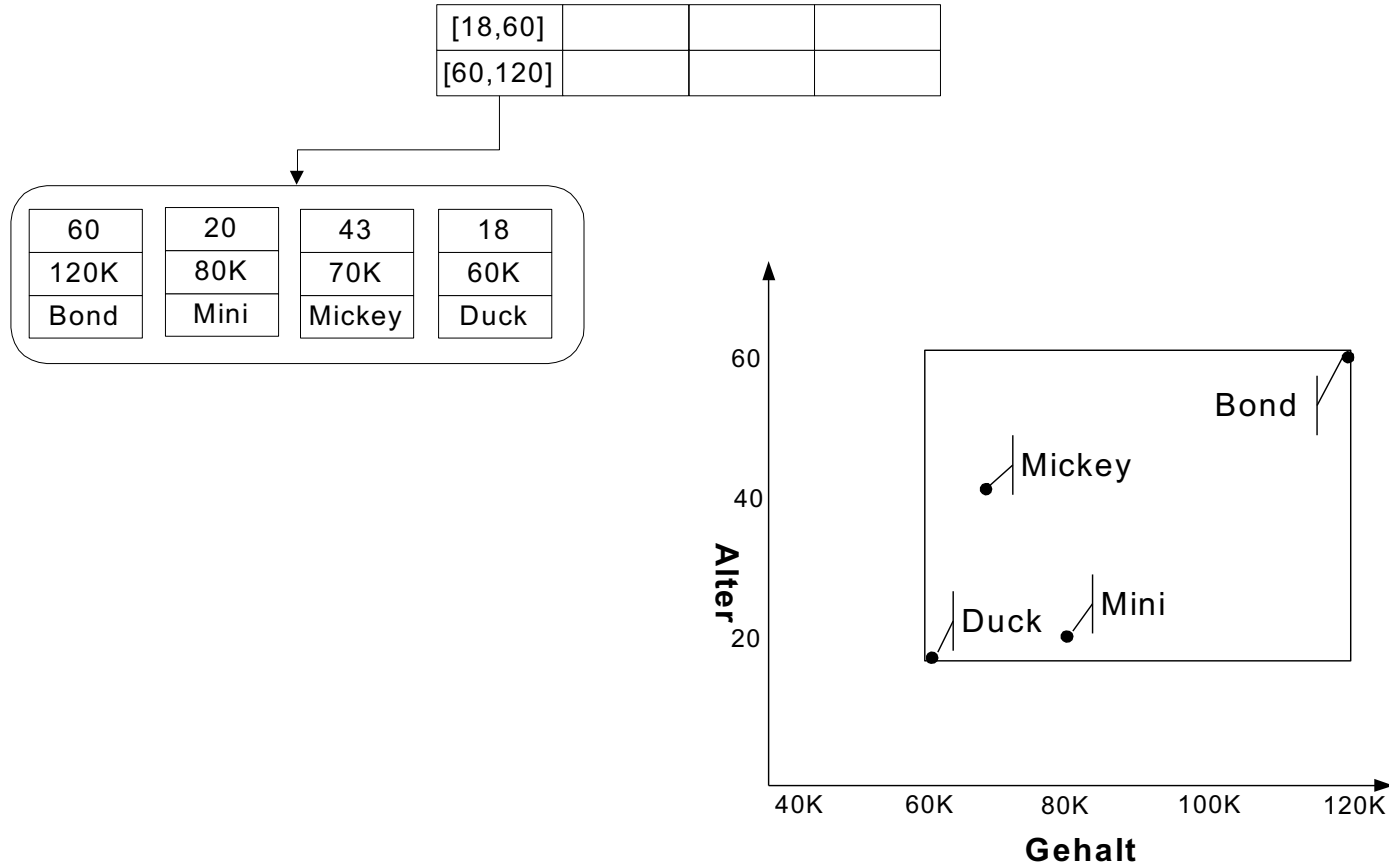
Datenraum  $\mathbf{D} = D_0 \times \dots \times D_{k-1}$

$k$ -dimensionaler Schlüssel entspricht Punkt im Datenraum  $p \in \mathbf{D}$

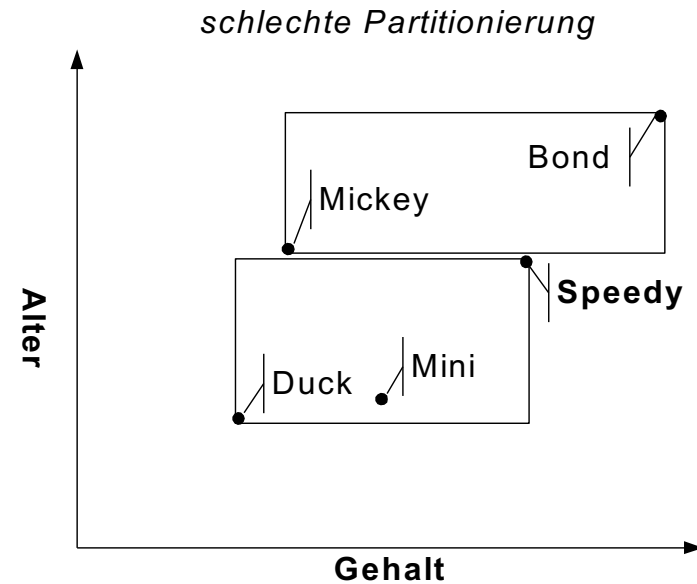
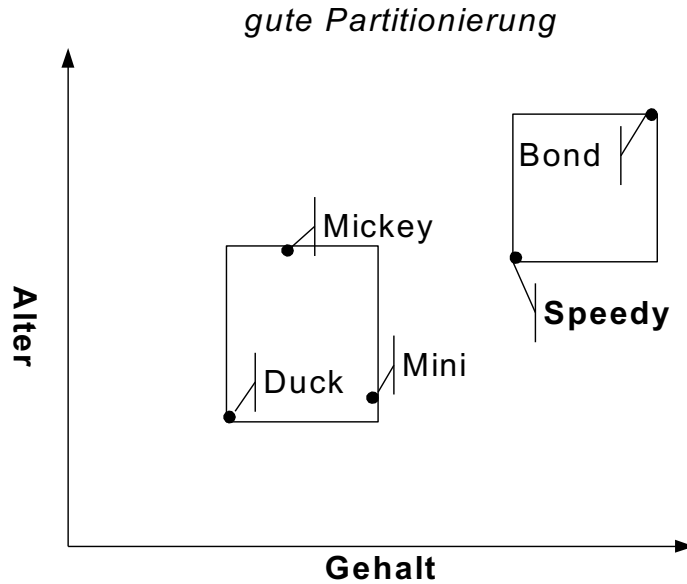
# Grundlagen mehrdimensionaler Datenstrukturen

1. Exact Match Query
  - spezifiziert Suchwert für jede Dimension  $D_i$
2. Partial Match Query
  - spezifiziert Suchwert für einen Teil der Dimensionen
3. Range Query
  - spezifiziert ein Suchintervall  $[ug_i, og_i]$  für alle Dimensionen
4. Partial Range Query
  - spezifiziert ein Suchintervall für einen Teil der Dimensionen

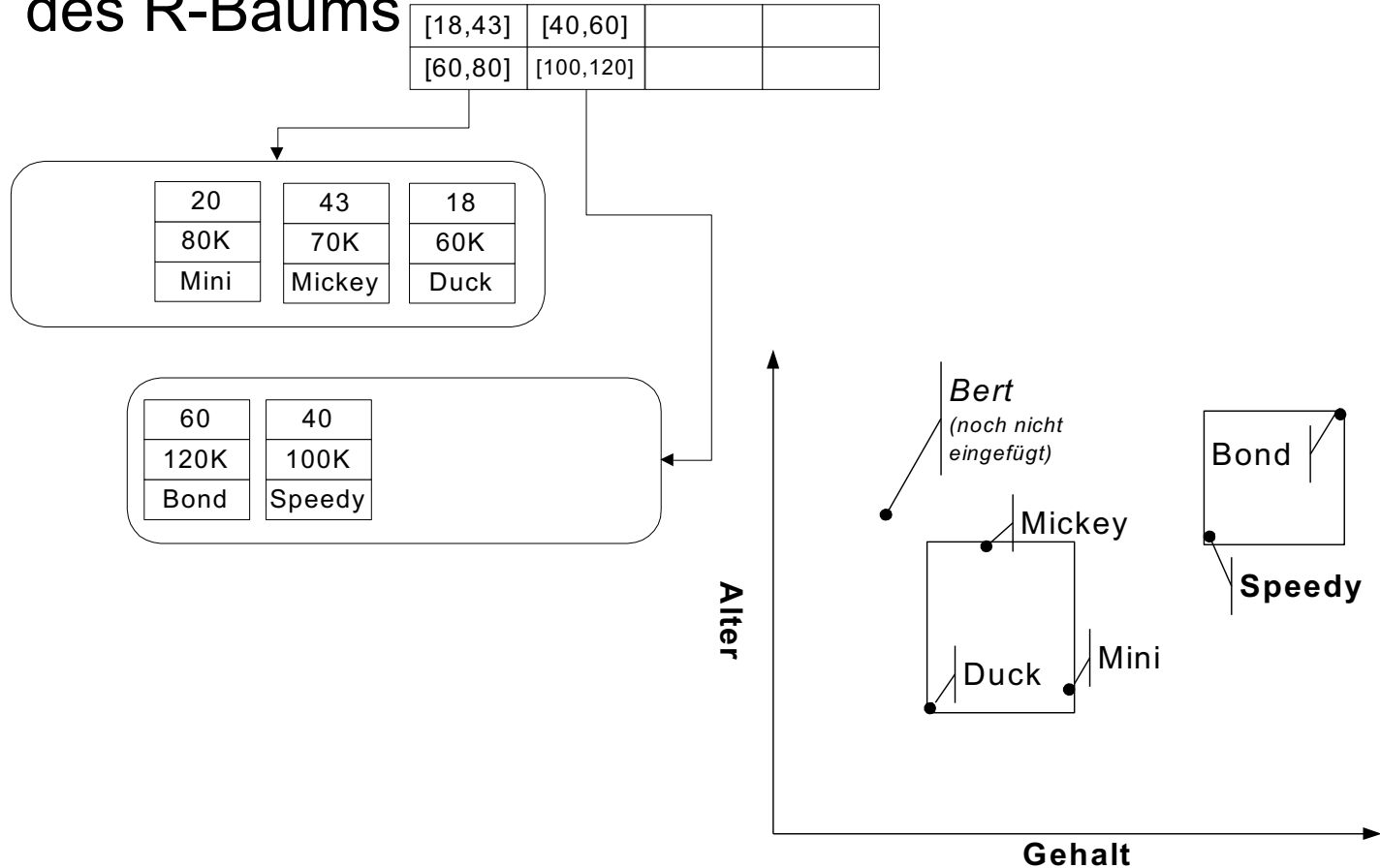
# R-Baum: Urvater der baum-strukturierten mehrdimensionalen Zugriffsstrukturen



# Gute versus schlechte Partitionierung

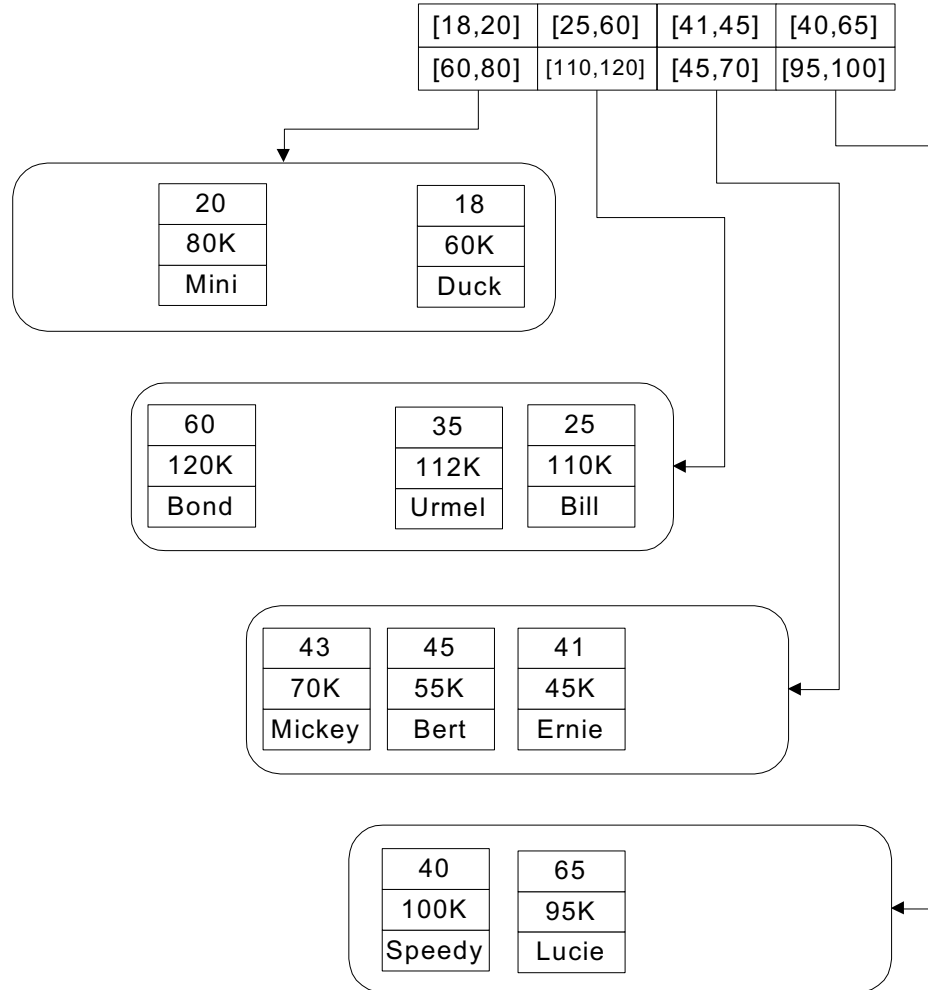


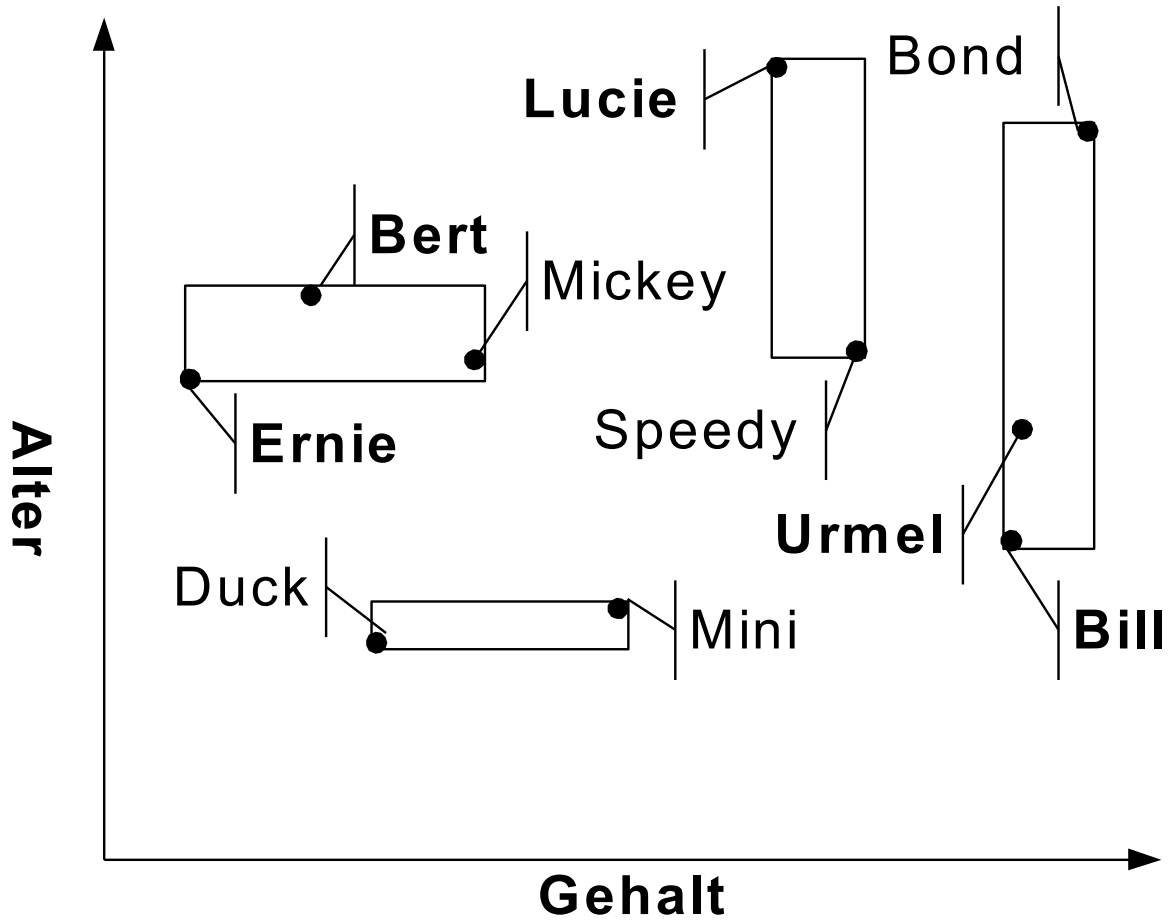
# Nächste Phase in der Entstehungsgeschichte des R-Baums



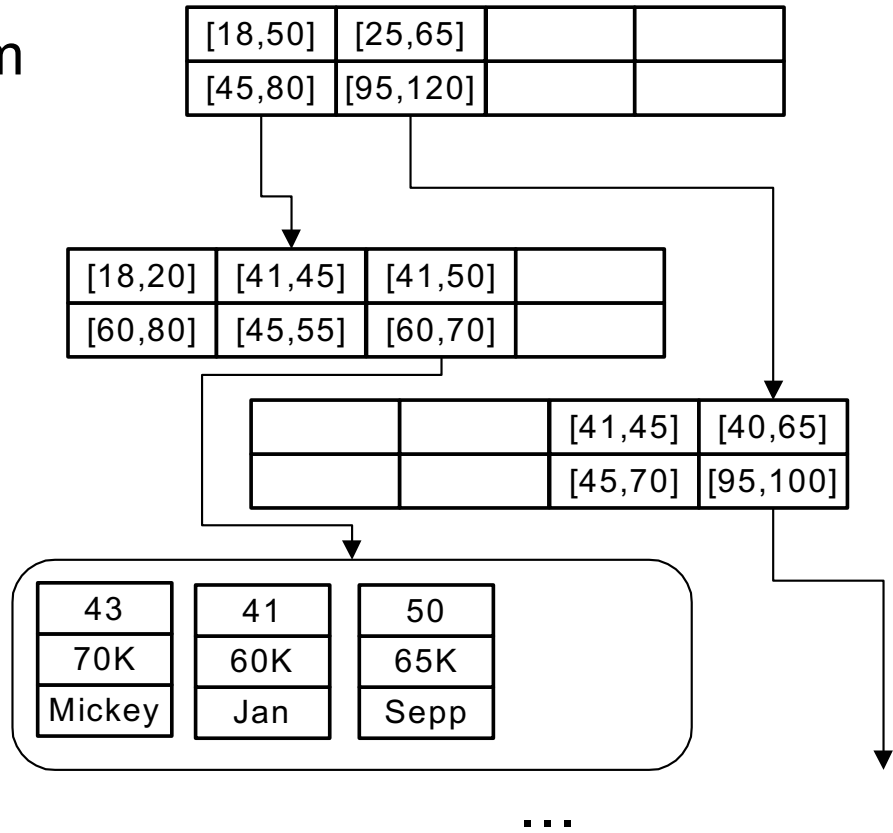


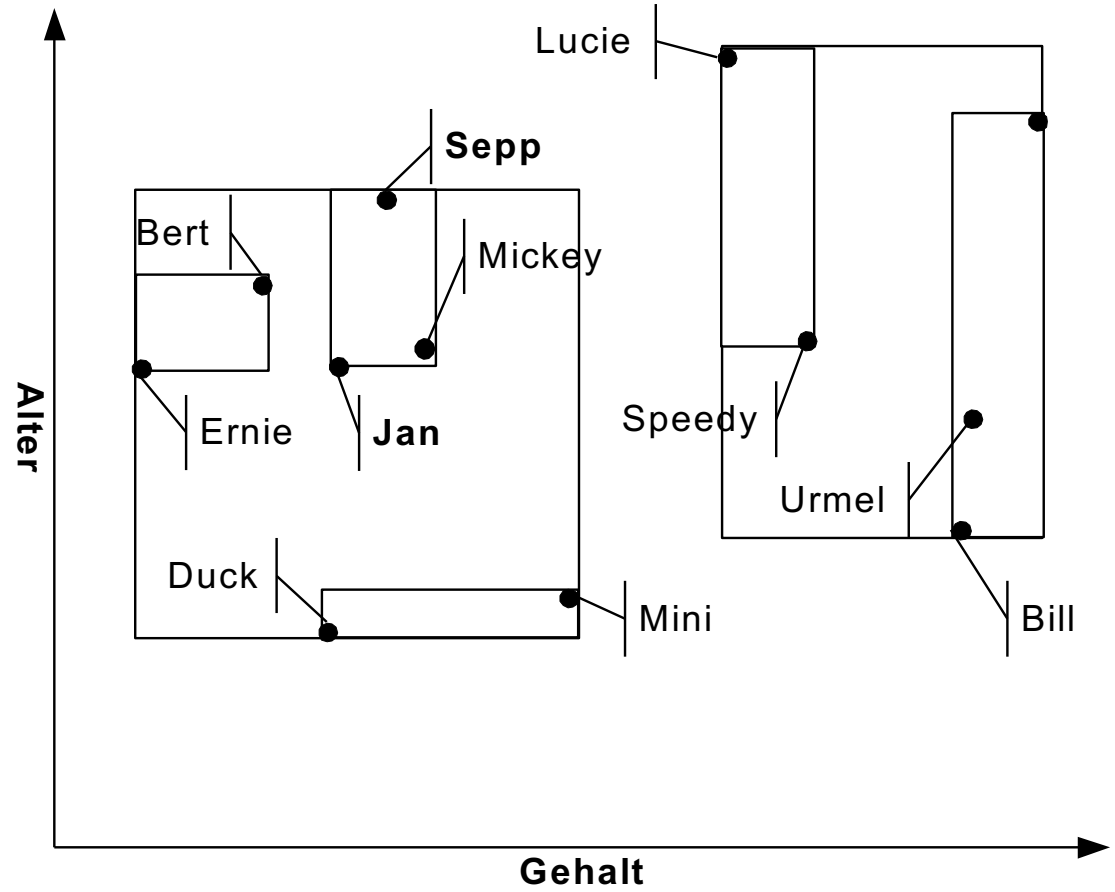
# Nächste Phase



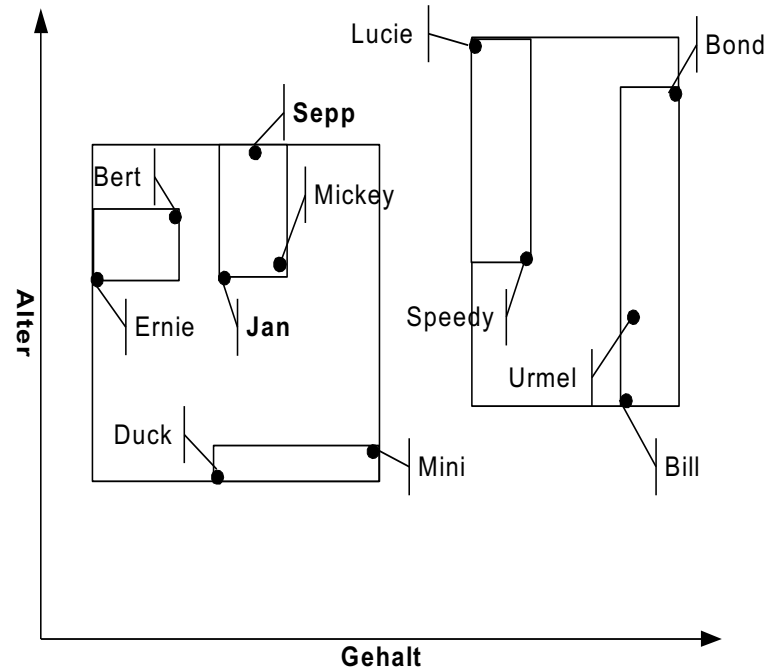
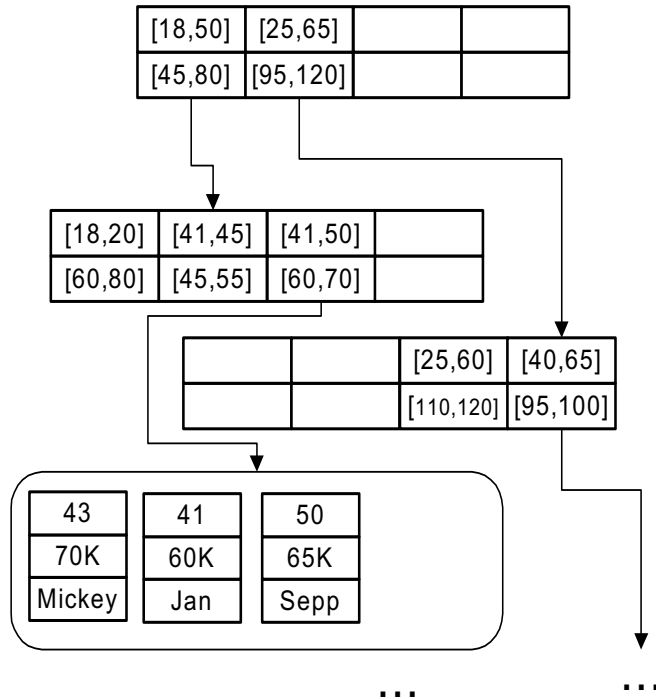


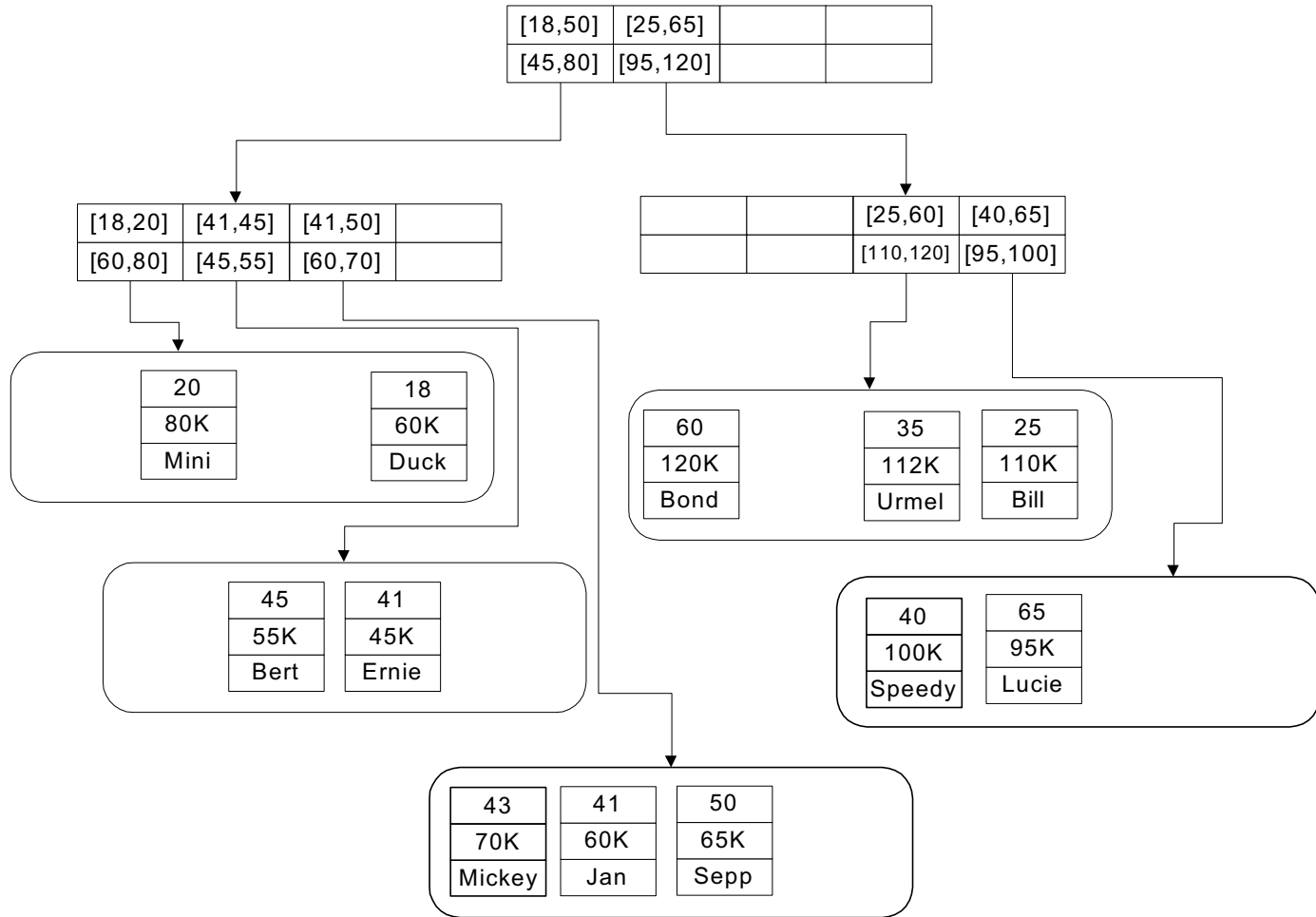
# Wachsen des Baums: nach oben – wie im B-Baum



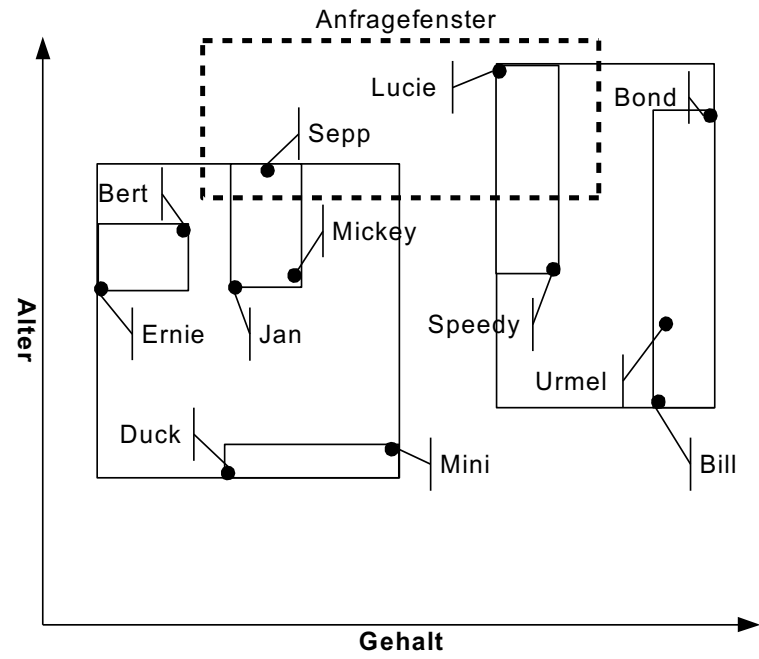
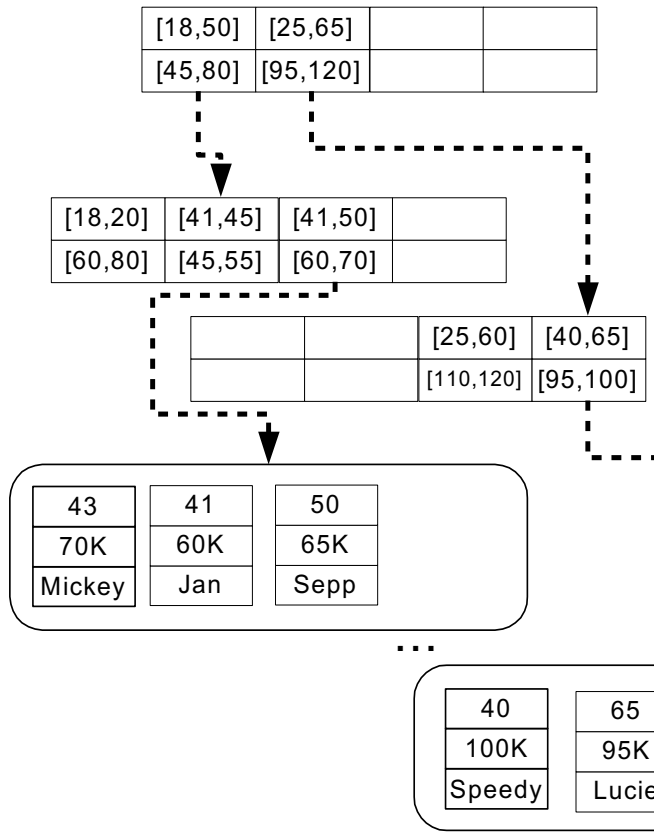


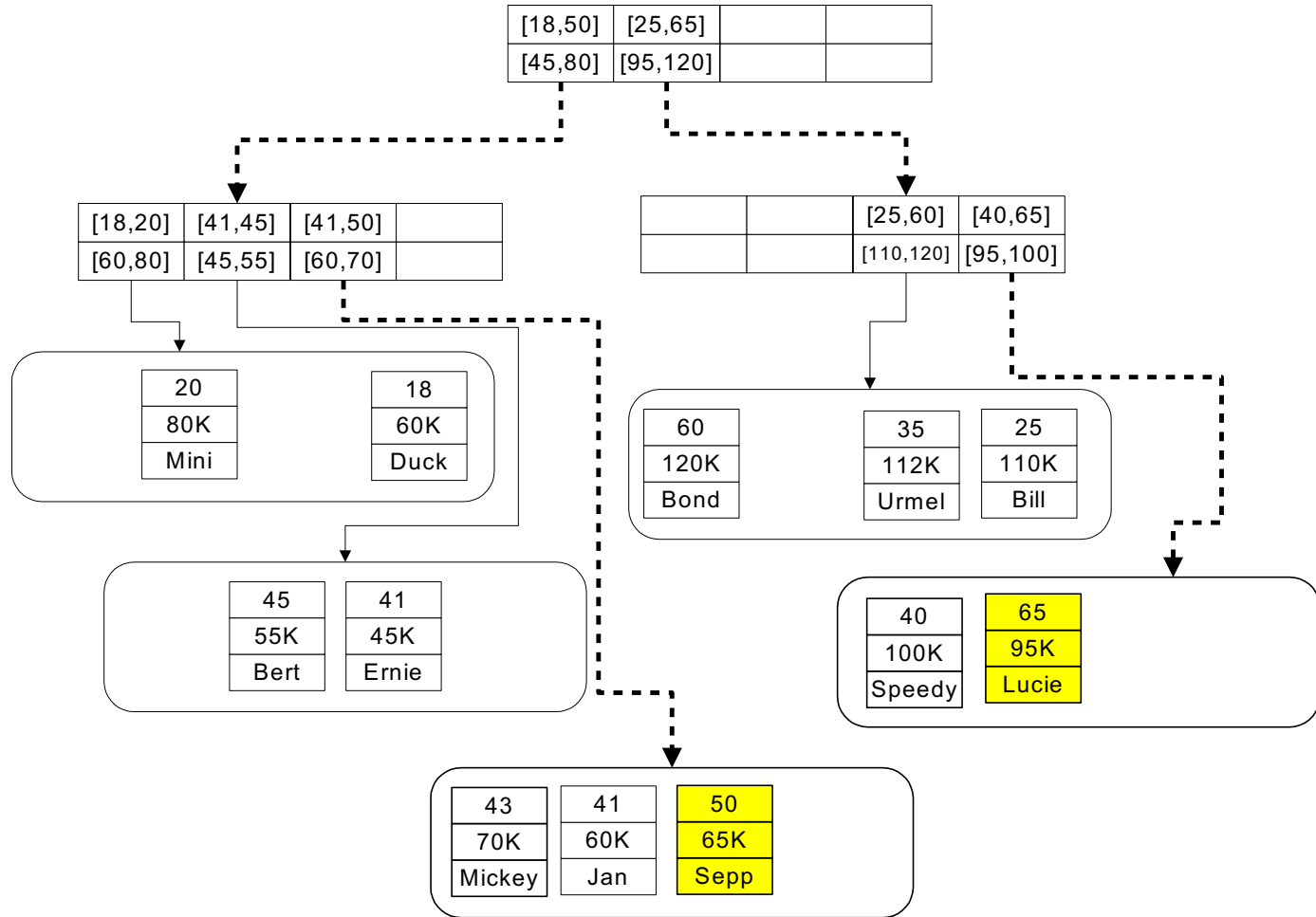
# Datenraum und Speicherstruktur – Überblick



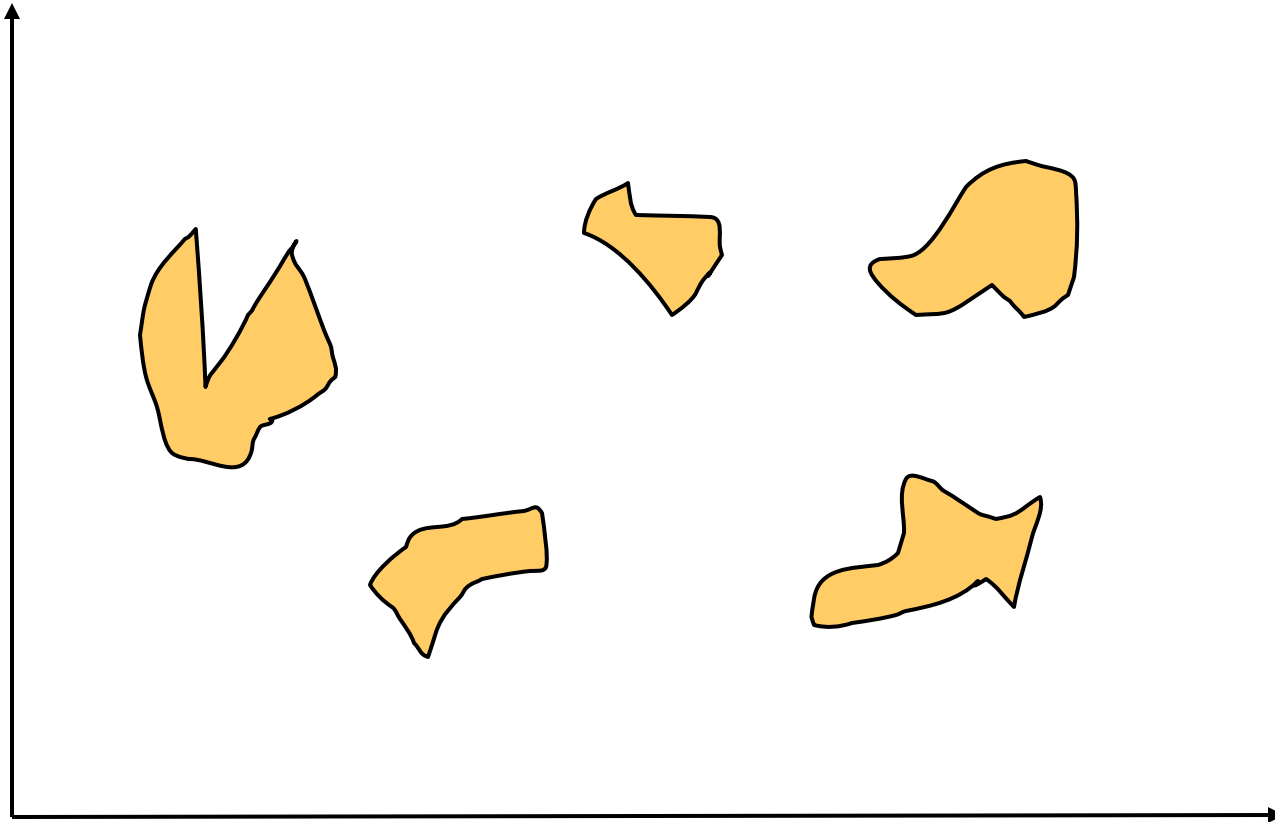


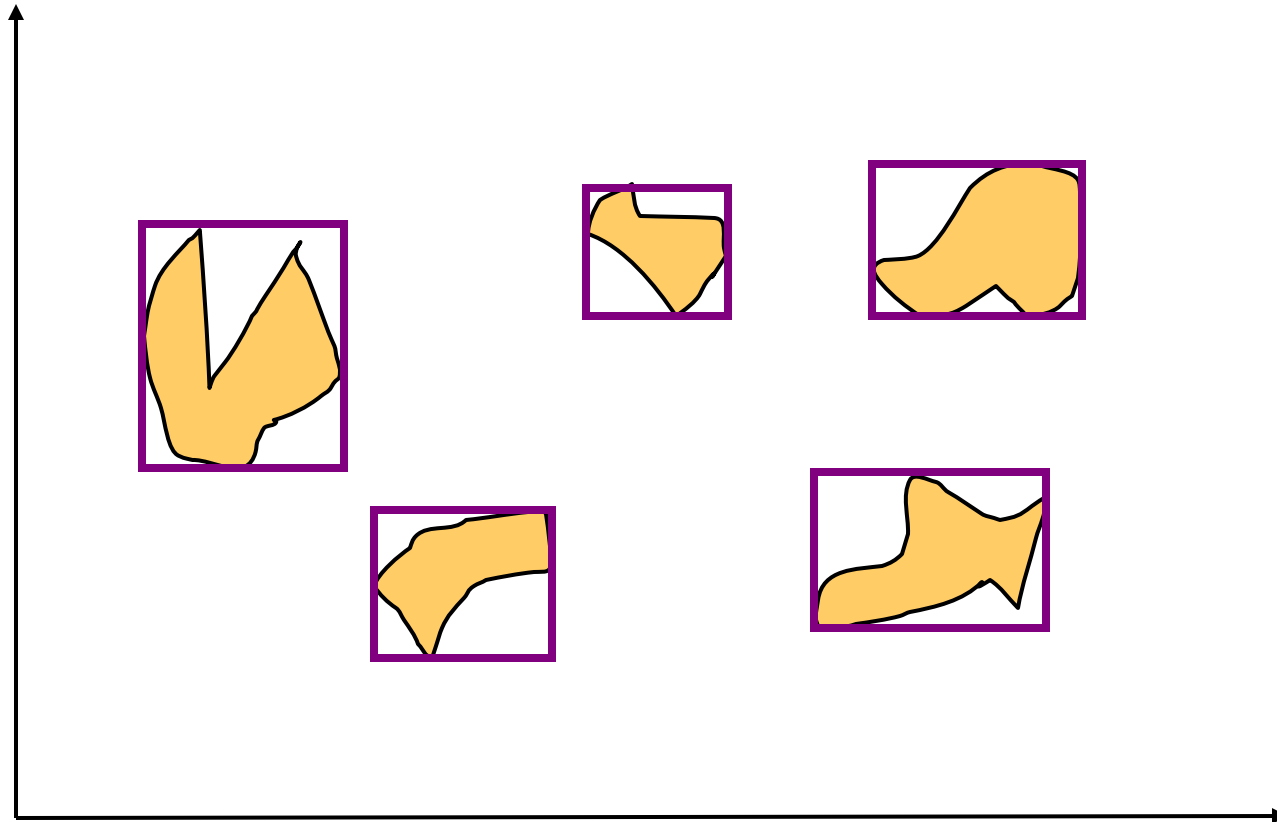
# Bereichsanfragen auf dem R-Baum

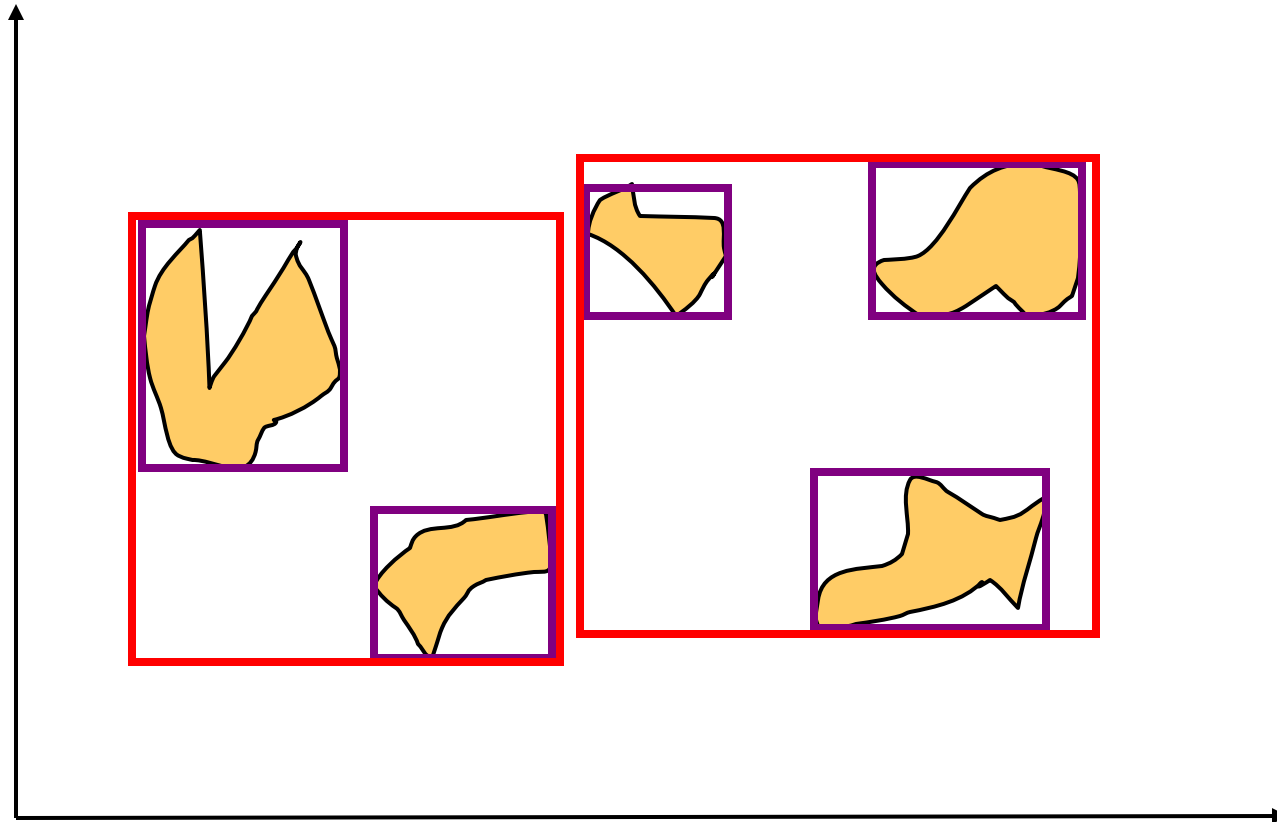






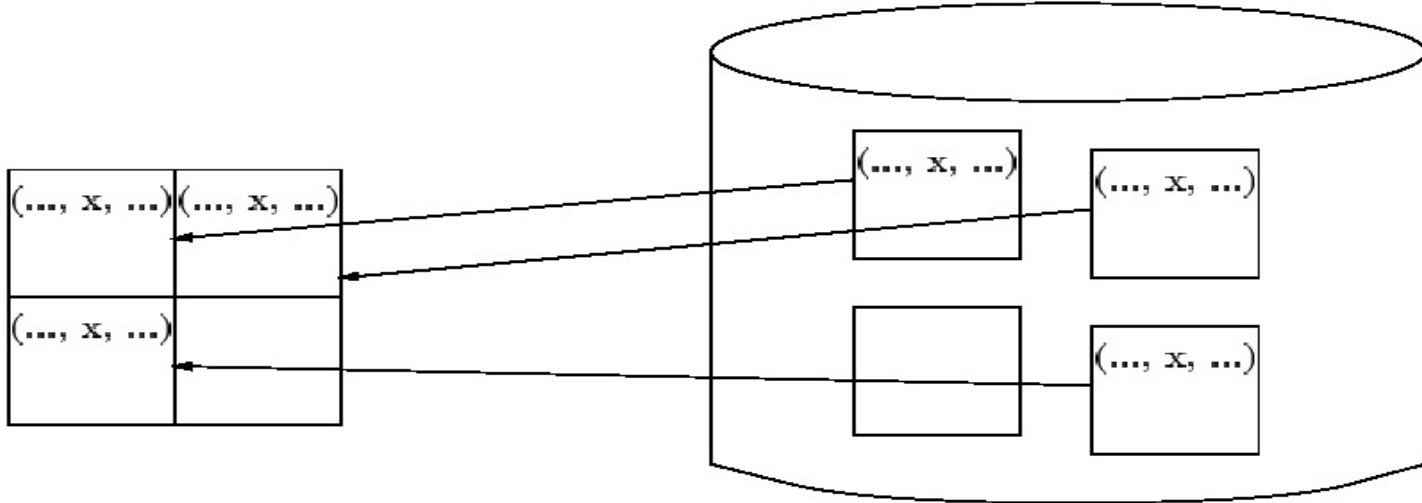






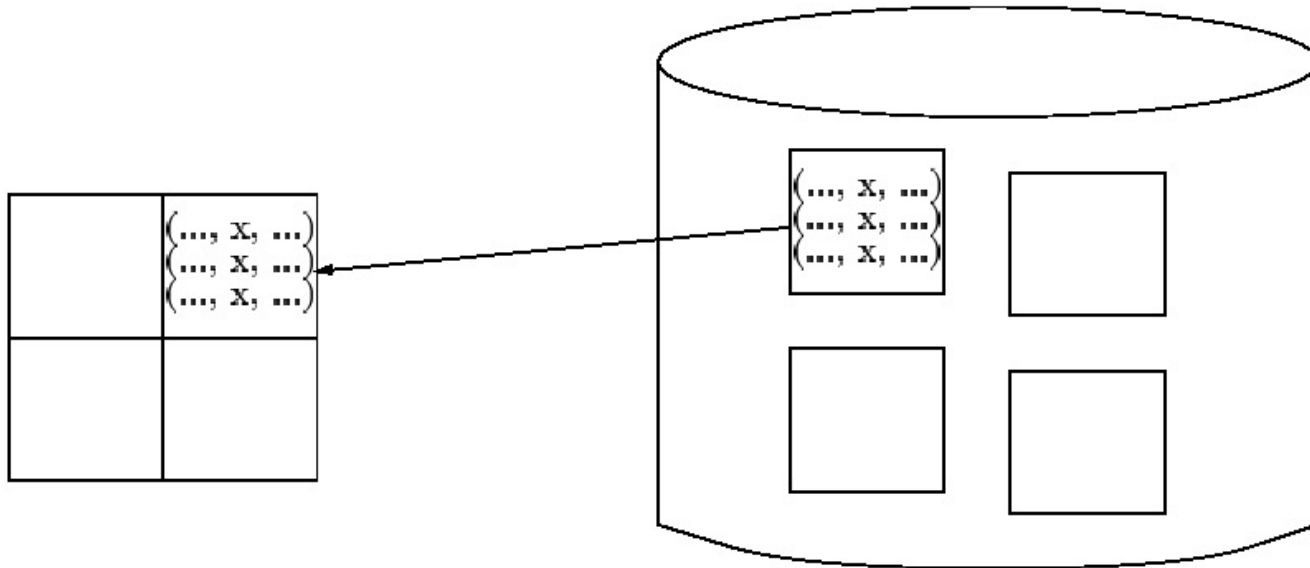
# Objektballung / Clustering logisch verwandter Daten

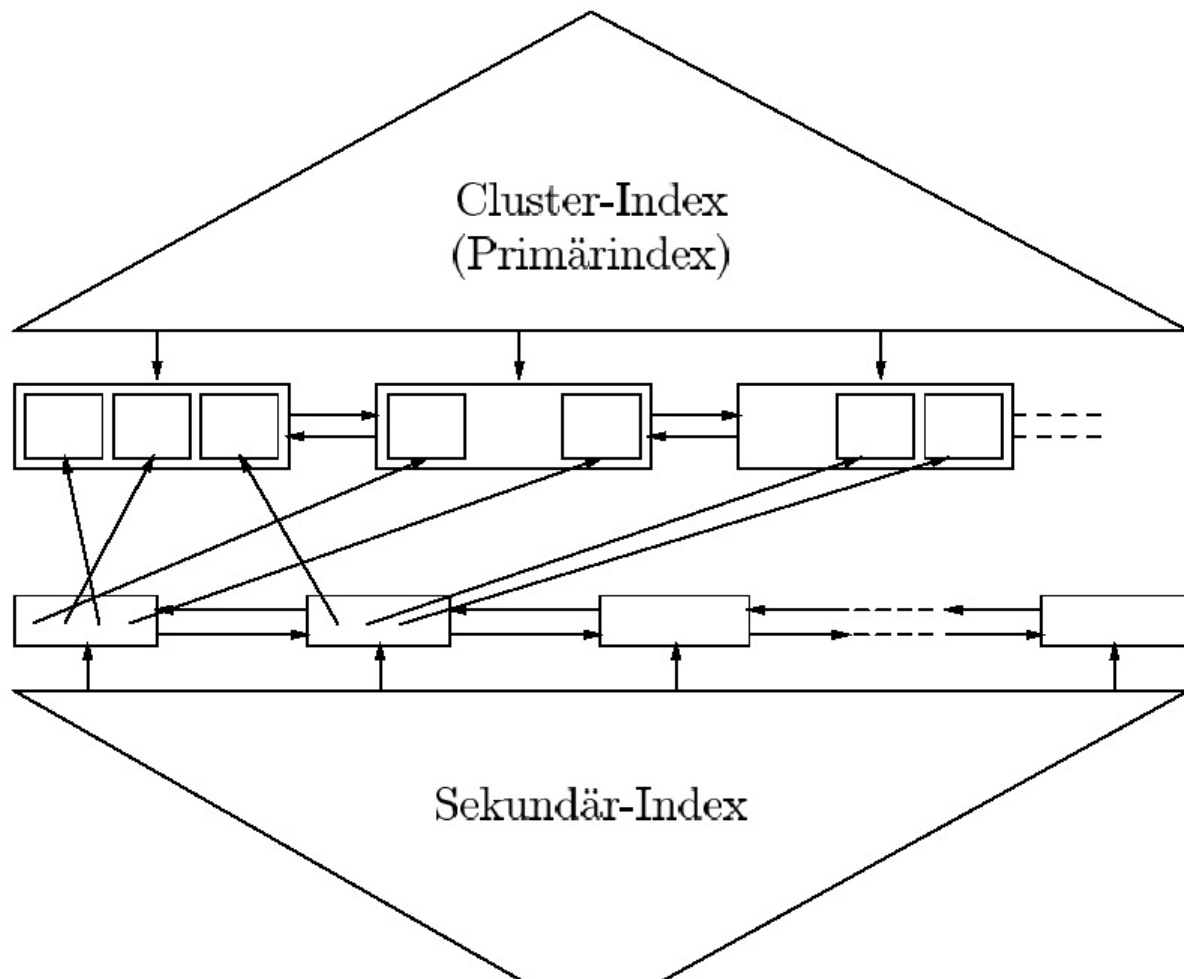
```
select *  
from R  
where A = x;
```



Hauptspeicher ← Zugriffslücke → Hintergrundspeicher

Hauptspeicher ← Zugriffslücke → Hintergrundspeicher





„verzahnte“ Ballung bei  
1:N Beziehungen

Seite  $P_i$ 

2125	o Sokrates	o C4	o 226	•
5041	o Ethik	o 4	o 2125	•
5049	o Mäeutik	o 2	o 2125	•
4052	o Logik	o 4	o 2125	•
2126	o Russel	o C4	o 232	•
5043	o Erkenntnistheorie	o 3	o 2126	•
5052	o Wissenschaftstheorie	o 3	o 2126	•
5216	o Bioethik	o 2	o 2126	•

Seite  $P_{i+1}$ 

2133	o Popper	o C3	o 52	•
5259	o Der Wiener Kreis	o 2	o 2133	•
2134	o Augustinus	o C3	o 309	•
5022	o Glaube und Wissen	o 2	o 2134	•
2137	o Kant	o C4	o 7	•
5001	o Grundzüge	o 4	o 2137	•
4630	o Die 3 Kritiken	o 4	o 2137	•

⋮

```
Select Name  
From Professoren  
Where PersNr = 2136
```

```
Select Name  
From Professoren  
Where Gehalt >= 90000 and Gehalt <= 100000
```



```
Create index SemesterInd  
on Studenten  
(Semester)
```

```
drop index SemesterInd
```

