Vorlesung Datenbanksysteme vom 24.10.2007

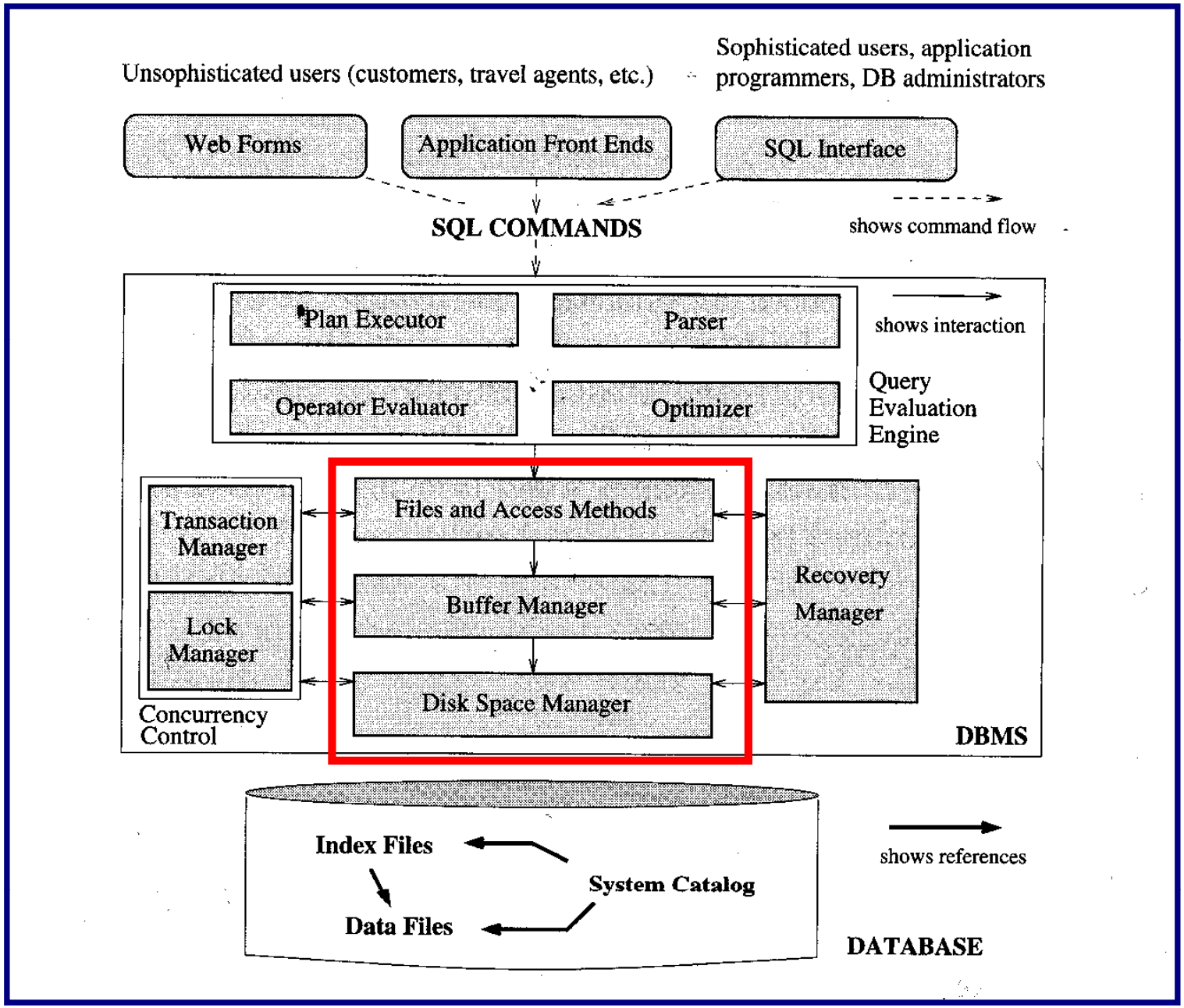
Physische Datenorganisation

- Architektur eines DBMS
- Speicherhierarchie
- Hintergrundspeicher / RAID
- Index-Verfahren
- Ballung (Clustering)
- „beste“ Zugriffsmethode

Architektur eines DBMS

- Wichtigste SW-Komponenten eines DBMS
- SW-Komponenten der physischen Speicherorganisation

Architektur eines DBMS



Disk Space Manager:

- Verwaltet die Daten auf der Platte
- Die höheren Schichten des DBMS sehen keine HW-Details sondern nur noch eine Sammlung von Seiten
- Disk Space Manager stellt Routinen bereit für
  - Allokieren / Deallokieren von Seiten
  - Lesen / Schreiben einer Seite

Bemerkung:

- 1 Seite entspricht einem Block auf der Platte
  $\Rightarrow$ Lesen/Schreiben einer Seite in einem I/O-Vorgang
- Die meisten DB-Systeme haben eigenes Disk Management (und erweitern nicht bloß die File System Funktionen des OS)
  $\Rightarrow$ flexibler, größere OS-Unabhängigkeit

Buffer Manager:

- Verwaltet den Datenbankpuffer (buffer pool) im Hauptspeicher
- Bearbeitung von Daten kann immer nur im Hauptspeicher (und nicht direkt auf der Platte) geschehen. $\Rightarrow$ Seiten müssen von der Platte in den Datenbankpuffer gelesen und später wieder auf Platte zurück geschrieben werden.
- Buffer Manager ist für das Einlesen und Auslagern von Seiten zw. Datenbankpuffer und Platte verantwortlich. Die anderen Schichten des DBMS fordern einfach eine Seite an.
- Buffer Manager speichert zu jeder Seite im Puffer Information:
  - ob die Seite noch verwendet wird: in diesem Fall darf die Seite nicht durch andere Seiten überschrieben werden.
  - ob die Seite geändert wurde: nur in diesem Fall ist Zurückschreiben auf Platte nötig

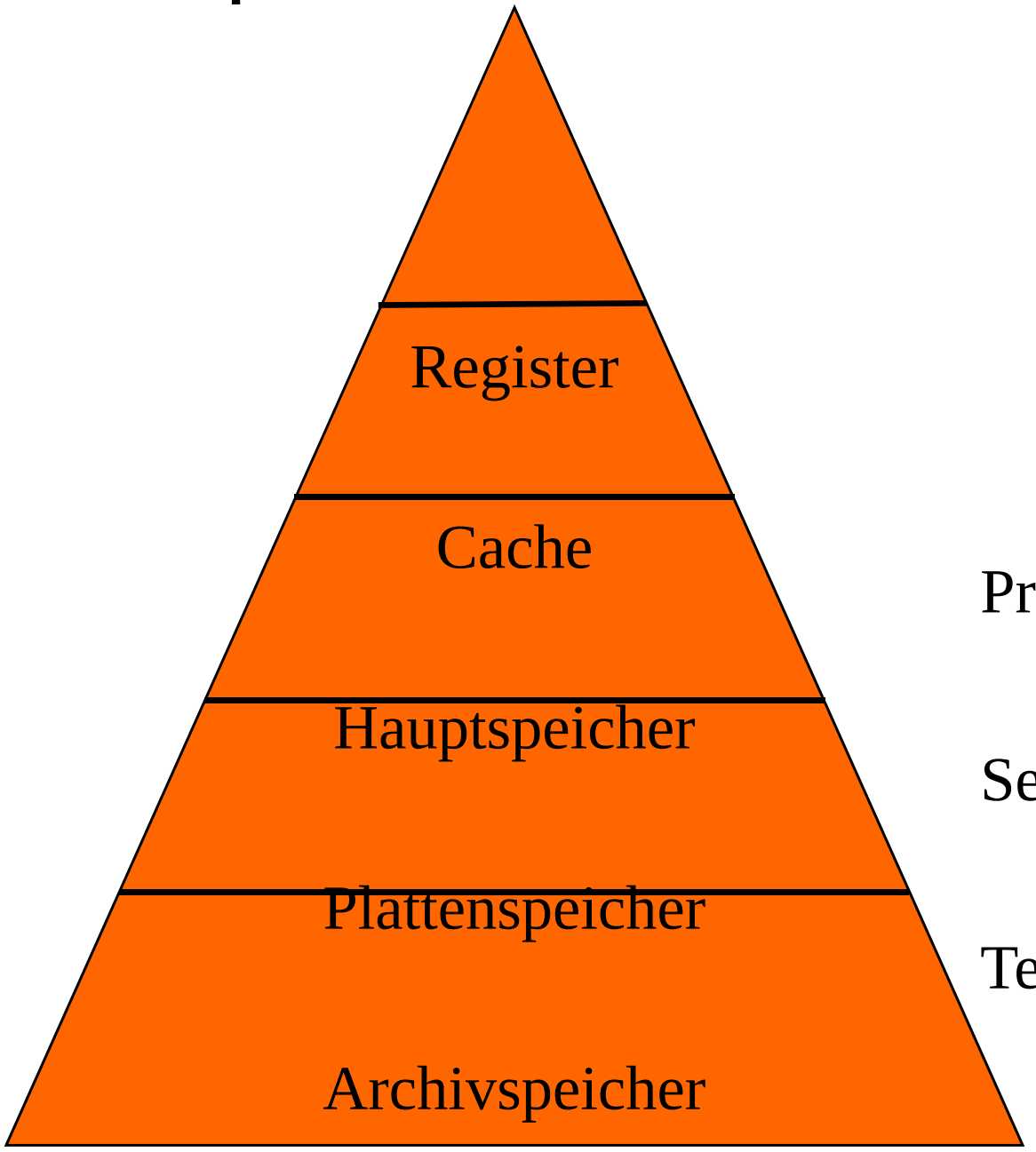
Files and Access Methods Layer:

- Die Zeilen einer DB-Tabelle (= Tupeln einer Relation) haben im allgemeinen nicht auf einer einzelnen Seite Platz. Logisch zusammengehörige Seiten werden als "File" verwaltet.
- Zwei Hauptaufgaben dieses Software Layers:
  1. Verwaltung der Seiten eines Files (entspricht üblicherweise einer Tabelle):
     - "Heap file": Zufällige Verteilung der Tupeln auf die Seiten
     - "Index file": Spezielle Verteilung der Tupeln auf die Seiten zwecks Optimierung bestimmter Zugriffe.
  2. Verwaltung der Tupeln innerhalb einer einzelnen Seite:
     - Seitenformat: Tupeln mit fixer oder variabler Länge
- Zugriff der anderen SW-Schichten auf die einzelnen Tupeln: mit TID (Tupelidentifikator) (engl.: RID: record ID), d.h. Seiten-ID + Speicherplatz innerhalb der Seite

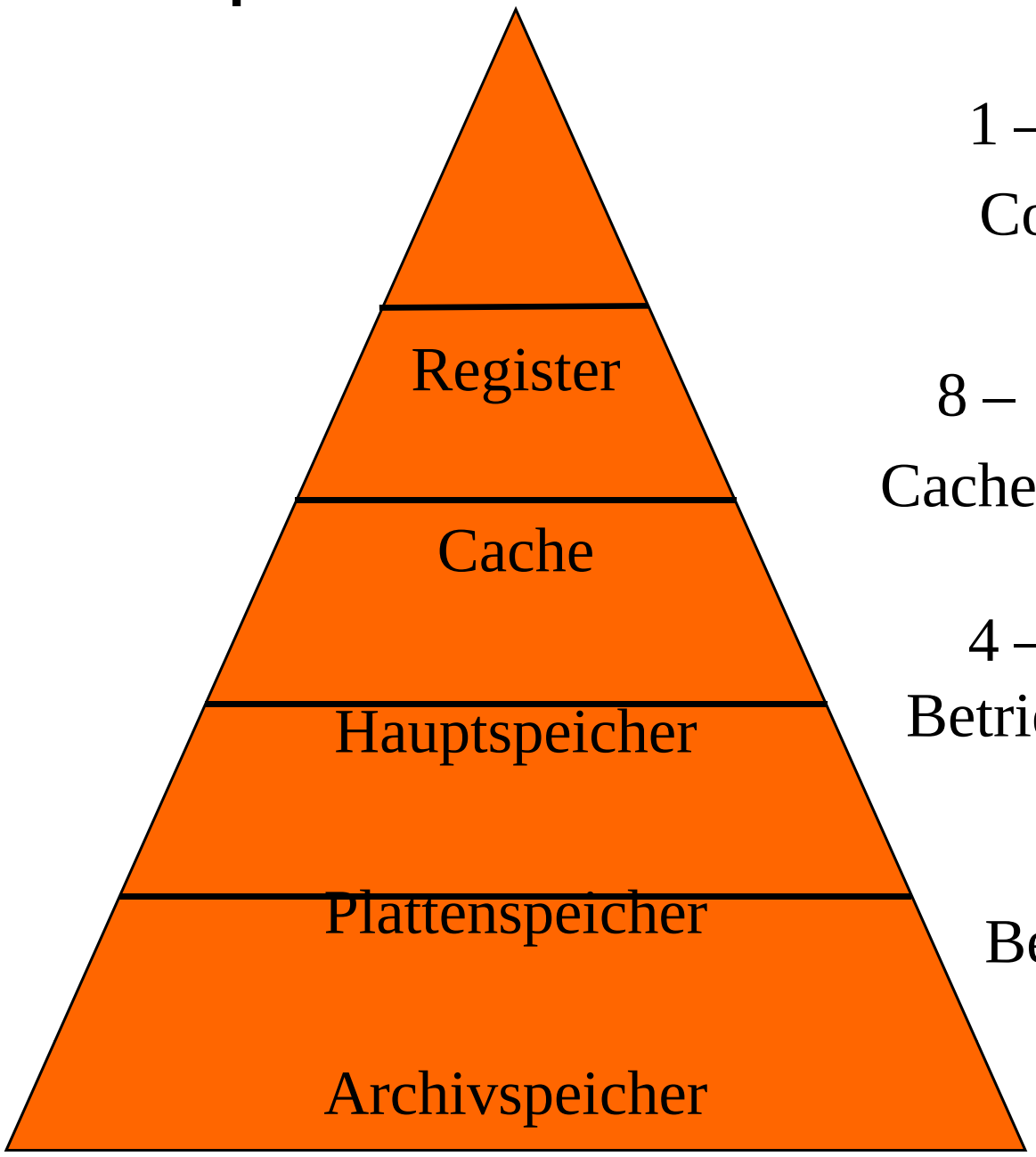
Speicherhierarchie

- Primärer / sekundärer / tertiärer Speicher
- Zugriffszeiten
- Puffer-Verwaltung
- Adressierung von Tupeln

Überblick: Speicherhierarchie



Überblick: Speicherhierarchie



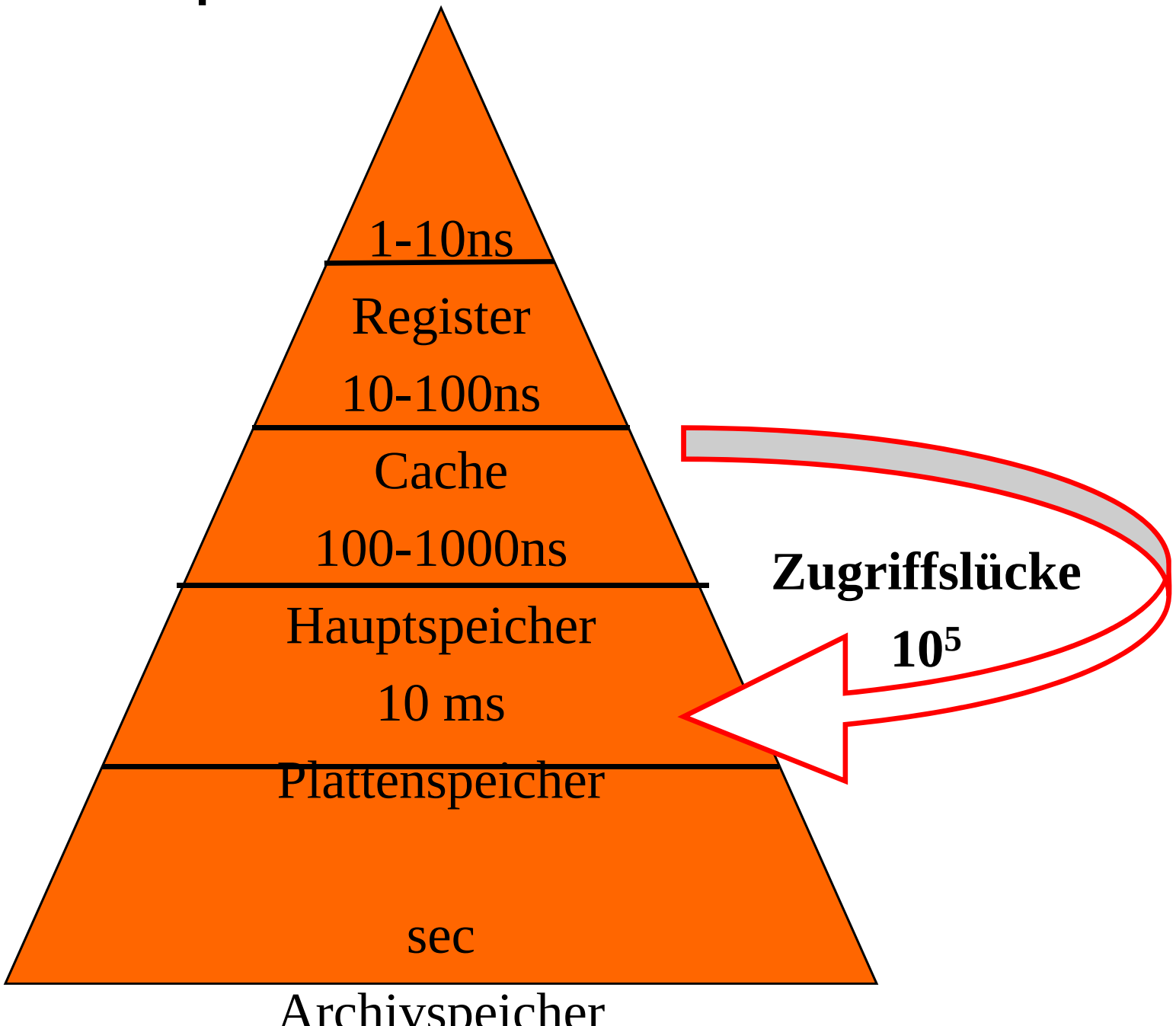
1 – 8 Byte
Compiler

8 – 128 Byte
Cache-Controller

4 – 64 KB
Betriebssystem

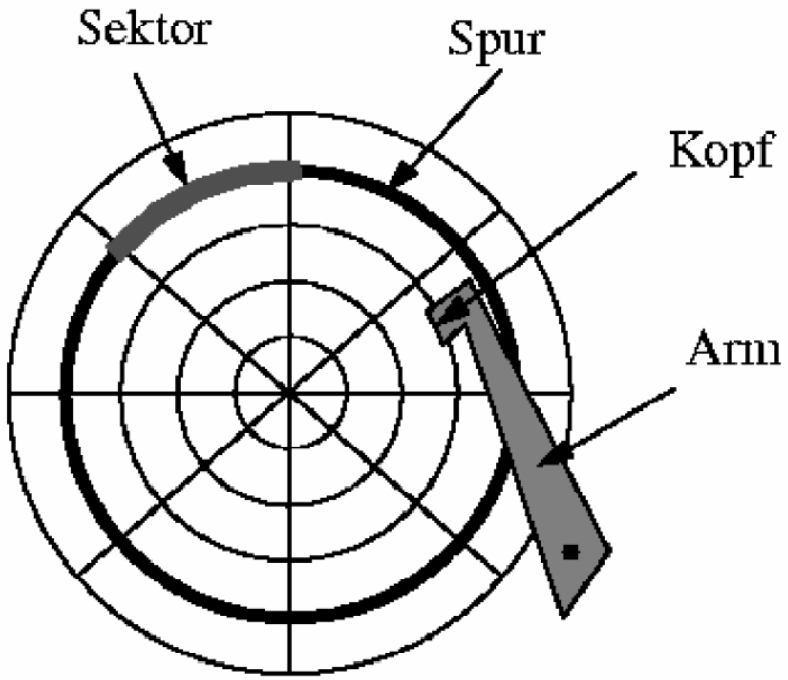
Benutzer

Überblick: Speicherhierarchie



Lesen von Daten von der Platte

- Seek Time: Arm auf gewünschte Spur positionieren: ➔ ca. 5ms
- Latenzzeit (= "rotational delay"): Warten, bis sich der gesuchte Block am Kopf vorbeibewegt (durchschnittlich ½ Plattenumdrehung; 10000 Umdrehungen / min) ➔ ca. 3ms
- Transfer von der Platte zum Hauptspeicher: (100 Mb /s) ➔ ca. 15 MB/s



Bemerkung:

- Sektor: physikalische Größe der Platte, meist 512B
- Block: kleinste Lese/Schreib-Einheit (= mehrere Sektoren), z.B.: 4kB, 8kB

Random versus Chained IO

Beispiel: 1000 Blöcke à 4KB sind zu lesen

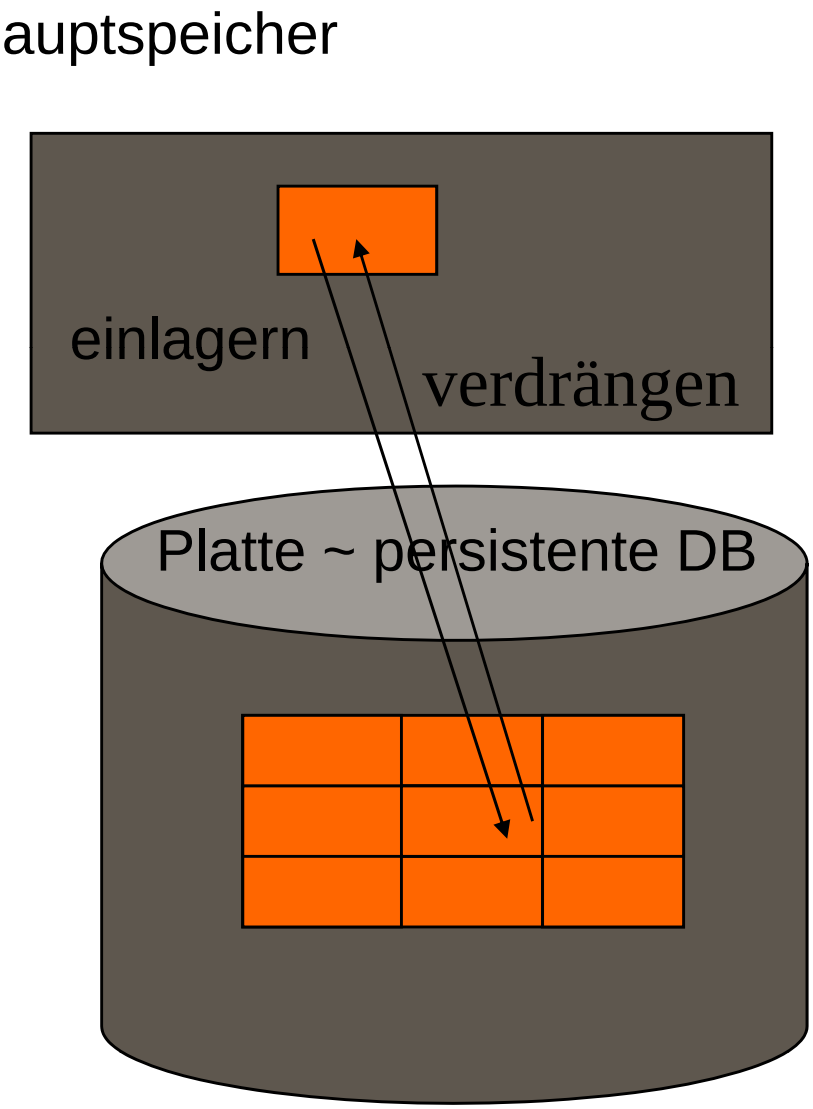
- Random I/O
  - Jedes Mal Arm positionieren
  - Jedes Mal Latenzzeit
  - ➔ 1000 * (5 ms + 3 ms) + Transferzeit von 4 MB
  - ➔ ca. 8000 ms + 300ms → 8s
- Chained I/O
  - Ein Mal positionieren, dann "von der Platte kratzen"
  - ➔ 5 ms + 3ms + Transferzeit von 4 MB
  - ➔ ca. 8ms + 300 ms → 1/3 s

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

Datenbankpuffer-Verwaltung

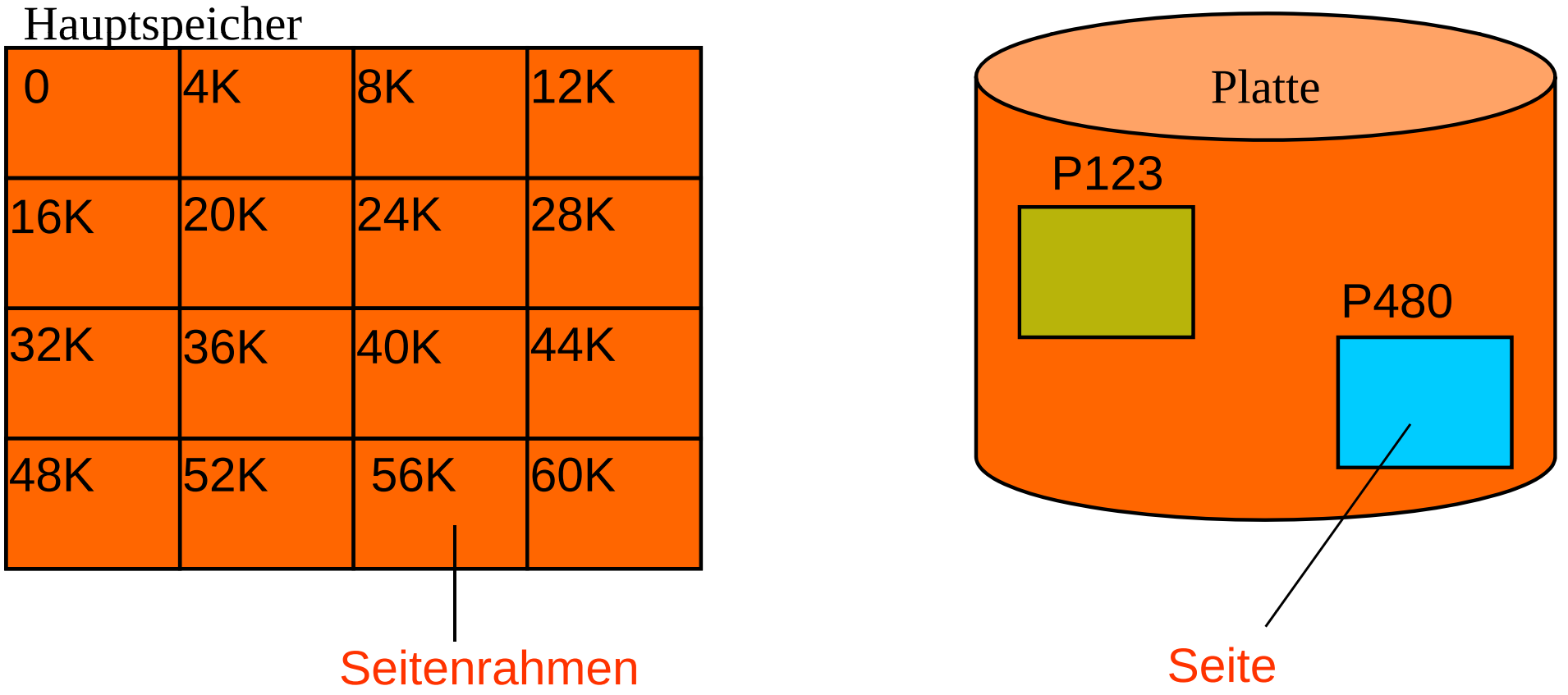
Bemerkung:

- DBMS greift nie direkt auf eine Seite (= Block) des Hintergrundspeichers zu
- Seite muss zuerst in den Hauptspeicher (Datenbank-puffer) gelesen werden.

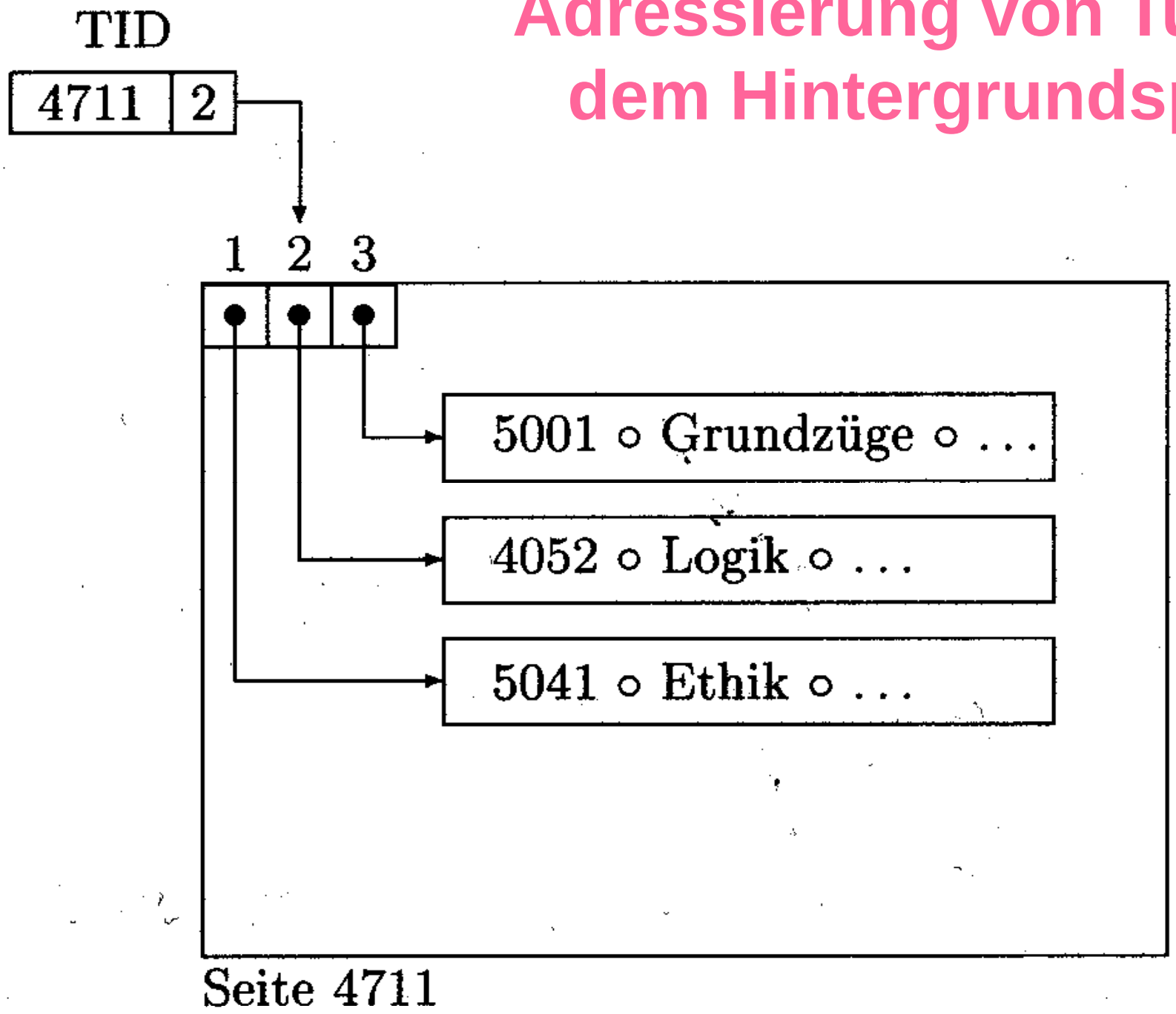


Ein- und Auslagern von Seiten

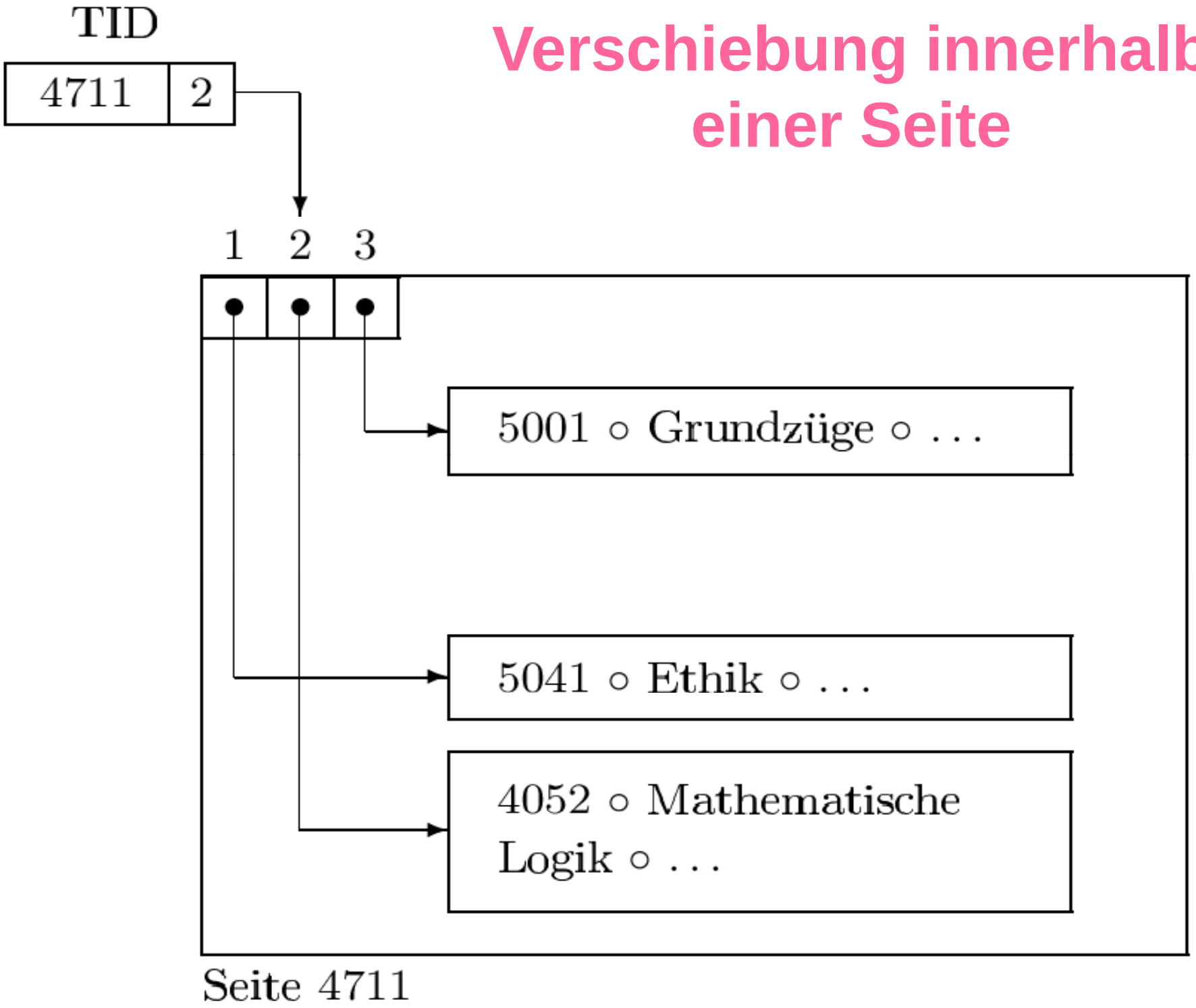
- DB-Puffer ist in Seitenrahmen gleicher Größe aufgeteilt.
- Ein Rahmen kann eine Seite aufnehmen.
- "Überzählige" Seiten werden auf die Platte ausgelagert.



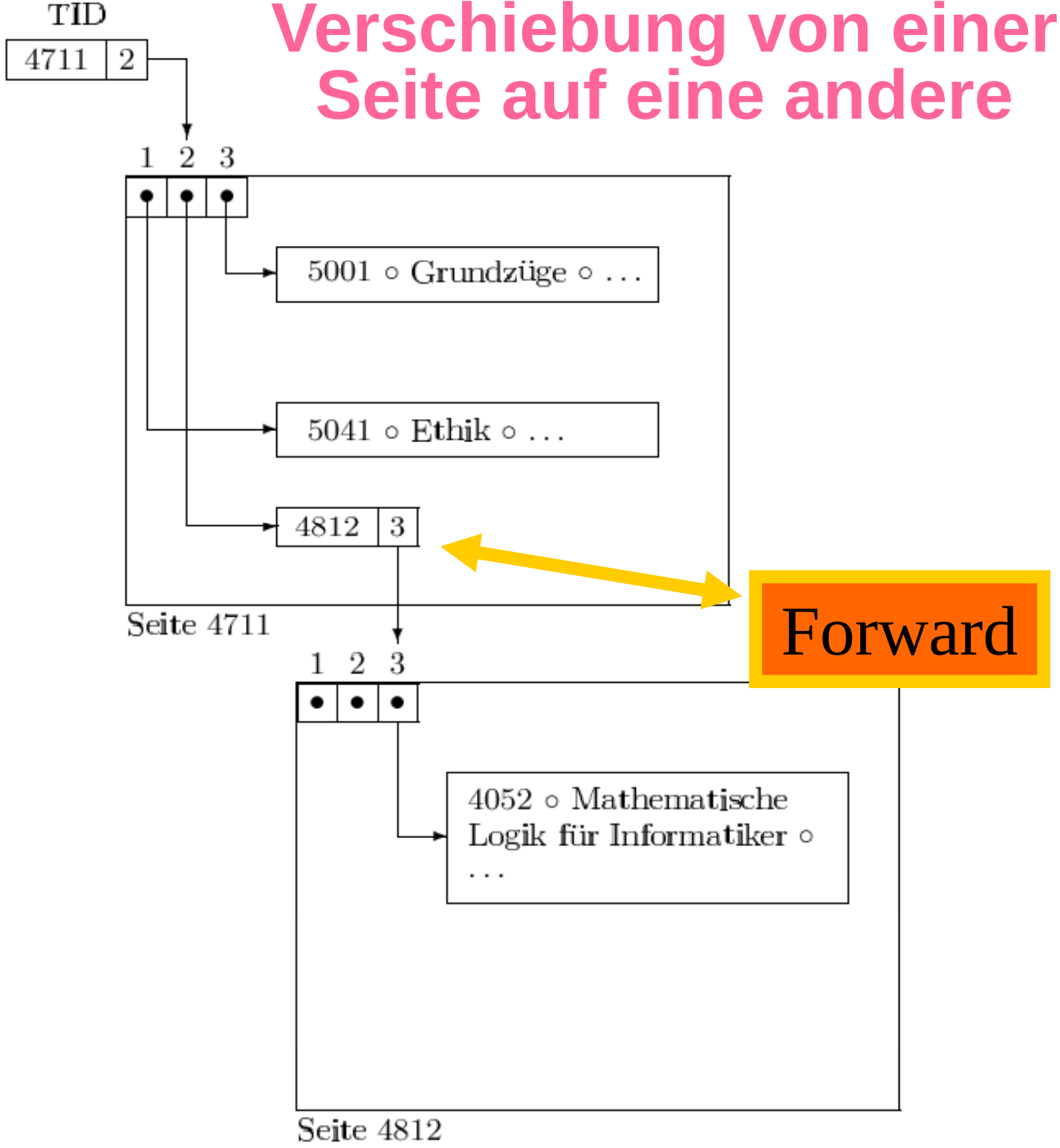
Adressierung von Tupeln auf dem Hintergrundspeicher



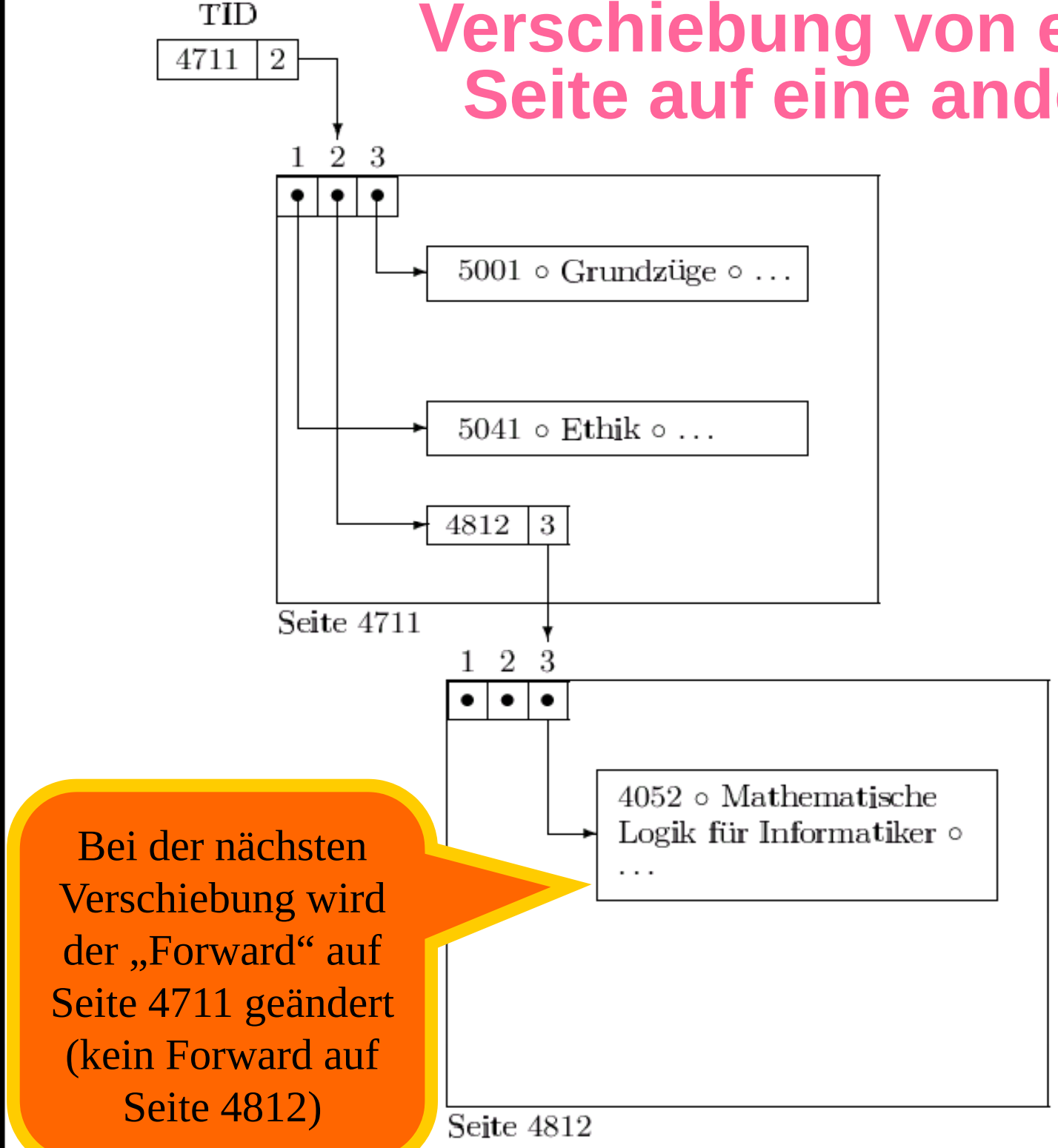
Verschiebung innerhalb einer Seite



Verschiebung von einer Seite auf eine andere



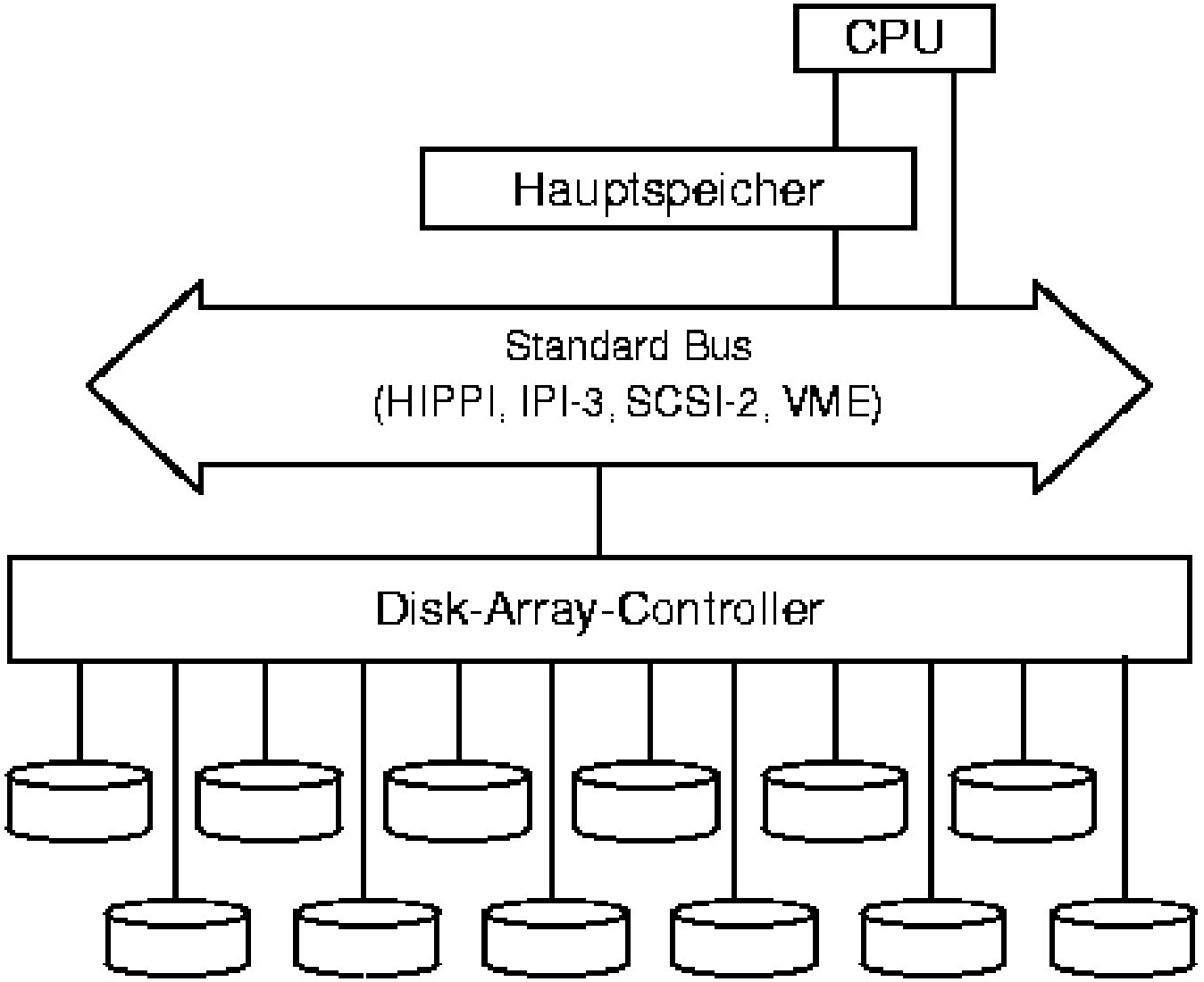
Verschiebung von einer Seite auf eine andere



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

Hintergrundspeicher / RAID

Disk Arrays ➔ RAID-Systeme



RAID

Abkürzung:

- ursprünglich: redundant array of *inexpensive* disks
- mittlerweile: redundant array of *independent* disks

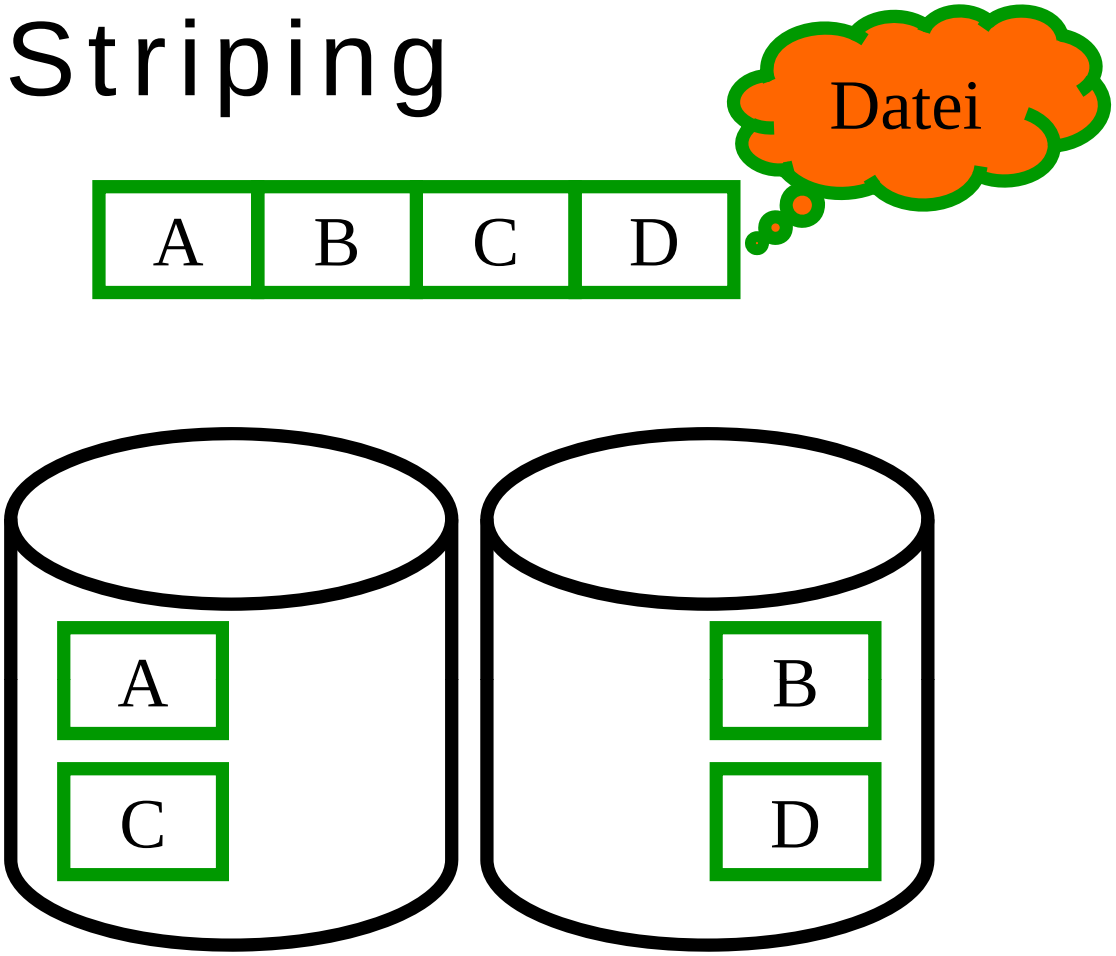
Idee: mehrere kleine Platten an Stelle von einer großen Platte

- "inexpensive disks" (ursprüngliche Idee von RAID):
  - Kleine Platten sind wesentlich billiger als große Platten.
  - (nicht zuletzt wegen Erfolg von RAID): ausreichend große Platten werden mittlerweile nicht einmal mehr hergestellt.
- "independent disks":
  - Parallele Schreib/Lese-Zugriffe auf die Platten möglich
  - Nach außen: über RAID-Controller sieht Speicherarray logisch wie eine einzige, große Platte aus.

RAID Levels

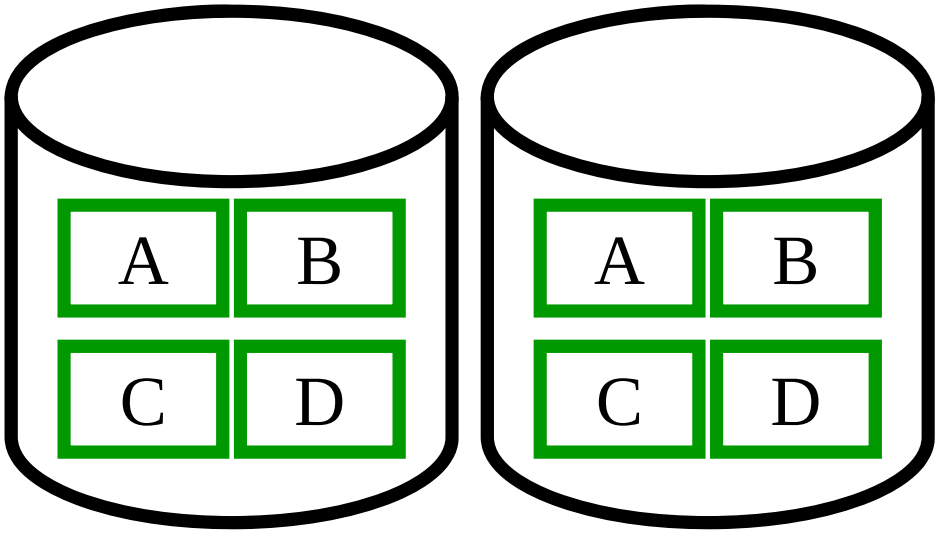
- Es gibt 8 RAID-Levels (für unterschiedliche Zielsetzungen):
  - RAID 0, 1, 0+1 (auch RAID 10 genannt), 2, 3, 4, 5, 6
- "Striping" (= Verteilung der Daten auf mehrere Platten):
  - ermöglicht parallelen Schreib/Lese-Zugriff
  - Problem: Ausfallswahrscheinlichkeit steigt mit der Anzahl der Platten
- Redundanz :
  - "Mirroring" (= mehrfaches Abspeichern der Daten): ermöglicht parallelen Lese-Zugriff
  - Abspeichern von Fehlererkennungs- und Korrekturcodes: ermöglicht Rekonstruktion der Daten bei Ausfall einer Platte.

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...
- Datenverlust wird immer wahrscheinlicher, je mehr Platten man verwendet (Stripingbreite = Anzahl der Platten, hier 2)

RAID 1: Spiegelung (mirroring)



- Datensicherheit: durch Redundanz aller Daten
- 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

Zusammenfassung: RAID Levels

- RAID 0: Striping (blockweise)
- RAID 1: Mirroring
- RAID 0+1 (= RAID 10): kombiniert Striping und Mirroring

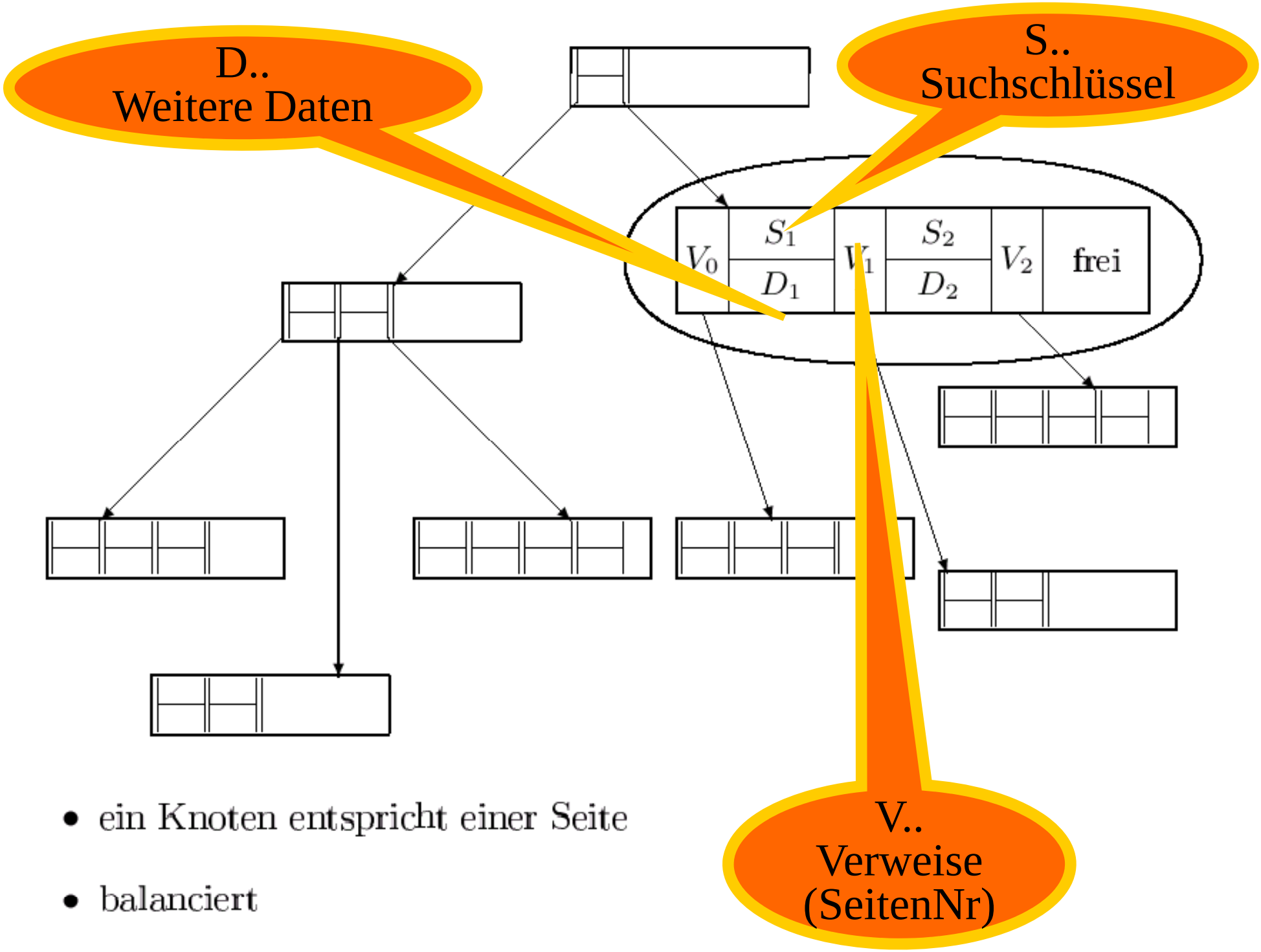
- RAID 2, 3, 4, 5, 6: unterschiedliche Kombinationen von Ideen:
  - Striping auf Bit (oder Byte)-Ebene bzw. blockweise
  - Umfang des Fehlererkennungs- und Korrekturcodes: Rekonstruktion der Daten bei Ausfall einer einzelnen Platte bzw. auch bei Ausfall von 2 Platten möglich
  - Speicherort des Fehlererkennungs- und Korrekturcodes: auf alle Laufwerke verteilt bzw. auf zusätzlicher Platte

Index-Verfahren

- B-Bäume
- B^+-Bäume
- Hashing
- R-Bäume

B-Bäume

- Problemstellung:
  - "Normale" Binär-Bäume (wie AVL-Bäume, Rot/Schwarz-Bäume) sind gut geeignet für den Hauptspeicher.
  - Bei Datenbanken benötigt man ein Verfahren, das auf die Seitengröße des Hintergrundspeichers abgestimmt ist.
  - Lösung: B-Bäume (oder B^+-Bäume)
- Idee von B-Bäumen (und B^+-Bäumen):
  - Die Knotengröße entspricht der Seitengröße.
  - Der Verzweigungsgrad des Baums hängt davon ab, wie viele Einträge auf einer Seite Platz haben.
  - Durch entsprechende Algorithmen für das Einfügen und Löschen von Daten wird garantiert, dass der Baum immer ausbalanciert ist => Logarithmische Zugriffszeiten.



- 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 $S_1, \ldots, S_n$ die Schlüssel eines Knotens mit $n+1$ Kindern. $V_0, V_1, \ldots, V_n$ seien die Verweise auf diese Kinder. Dann gilt:

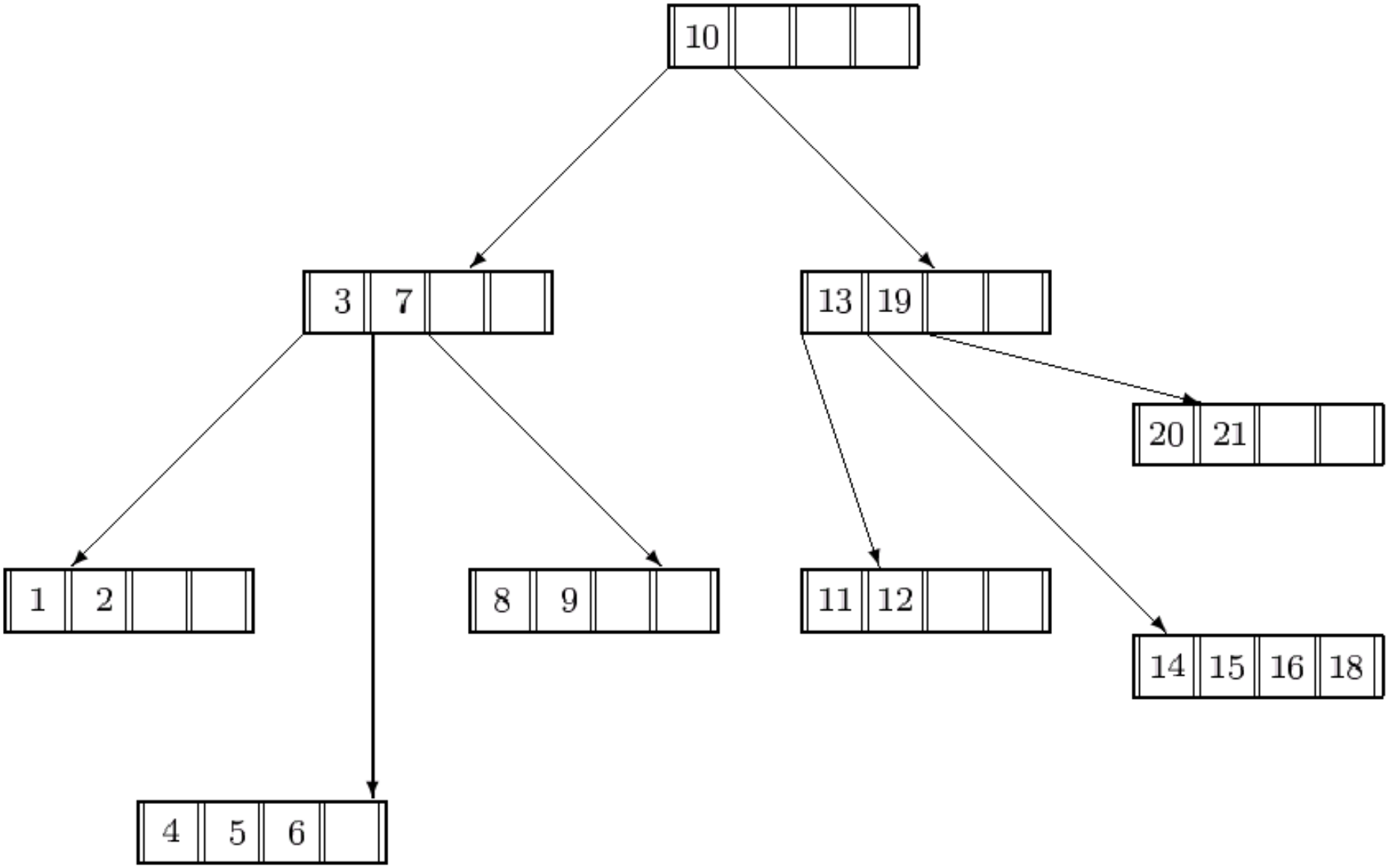
   (a) V_0 weist auf den Teilbaum mit Schlüsseln kleiner als S_1.

   (b) V_i $(i = 1, \ldots, n-1)$ 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



Einfügen eines neuen Objekts (Datensatz) in einen B-Baum

1. Führe eine Suche nach dem Schlüssel durch; diese endet (scheitert) an der Einfügestelle.

2. Füge den Schlüssel dort ein.

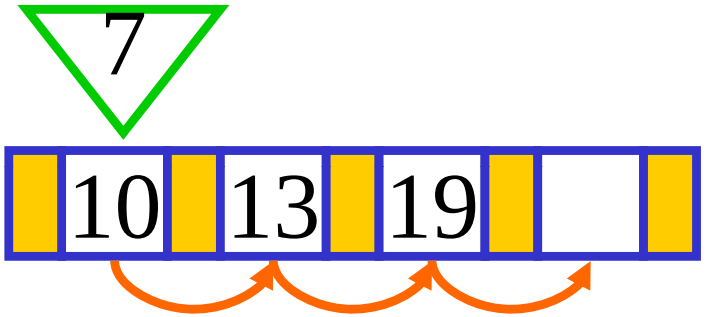
3. Ist der Knoten überfüllt, teile ihn
   - Lege einen neuen Knoten an und belege ihn mit den Schlüsseln, die rechts vom mittleren Eintrag des überfüllten Knotens liegen.
   - Füge den mittleren Eintrag im Vaterknoten des überfüllten Knotens ein.
   - Verbinde den Verweis rechts des neuen Eintrags im 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.

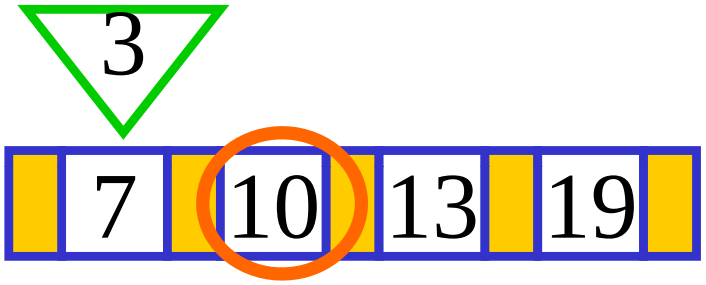
Beispiel 1

- B-Baum vom Grad k = 2
- Ausgangssituation: 1 Knoten mit Schlüssel-Werten 10, 13, 19
- Sukzessiver Aufbau des Baums durch das **Einfügen neuer Schlüssel-Werte**: 7, 3, 1, 2, 4

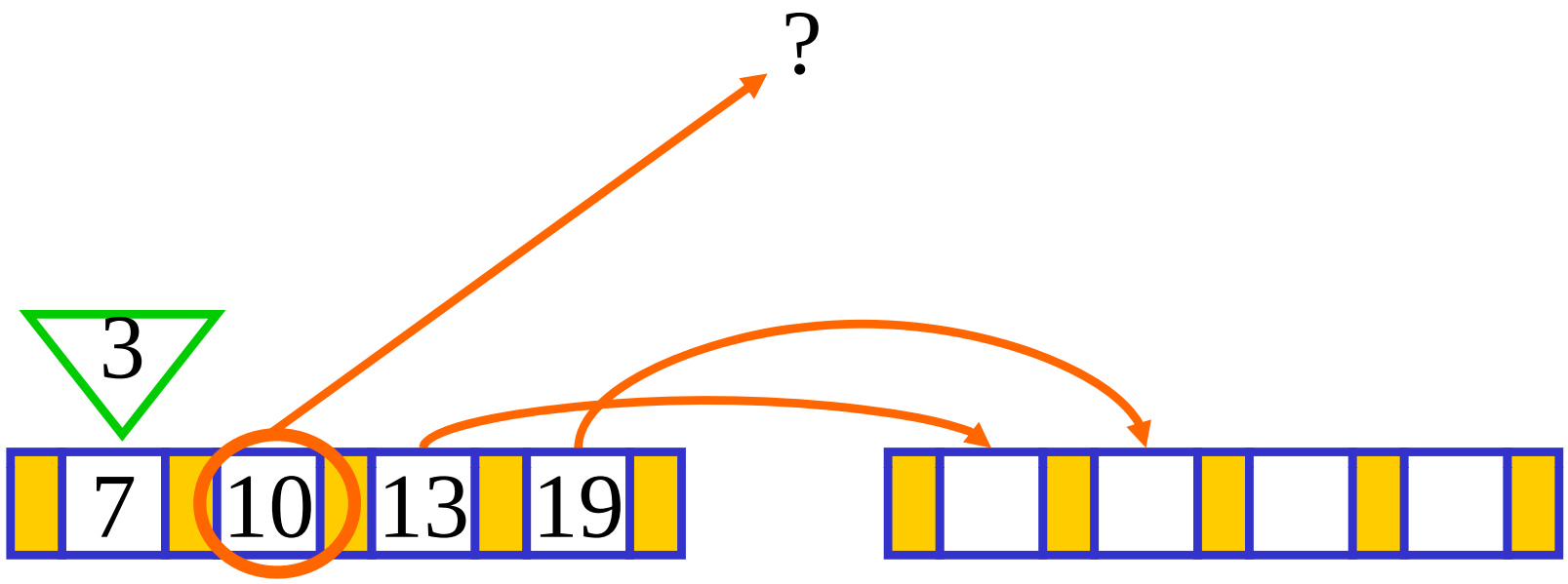
Beispiel 1



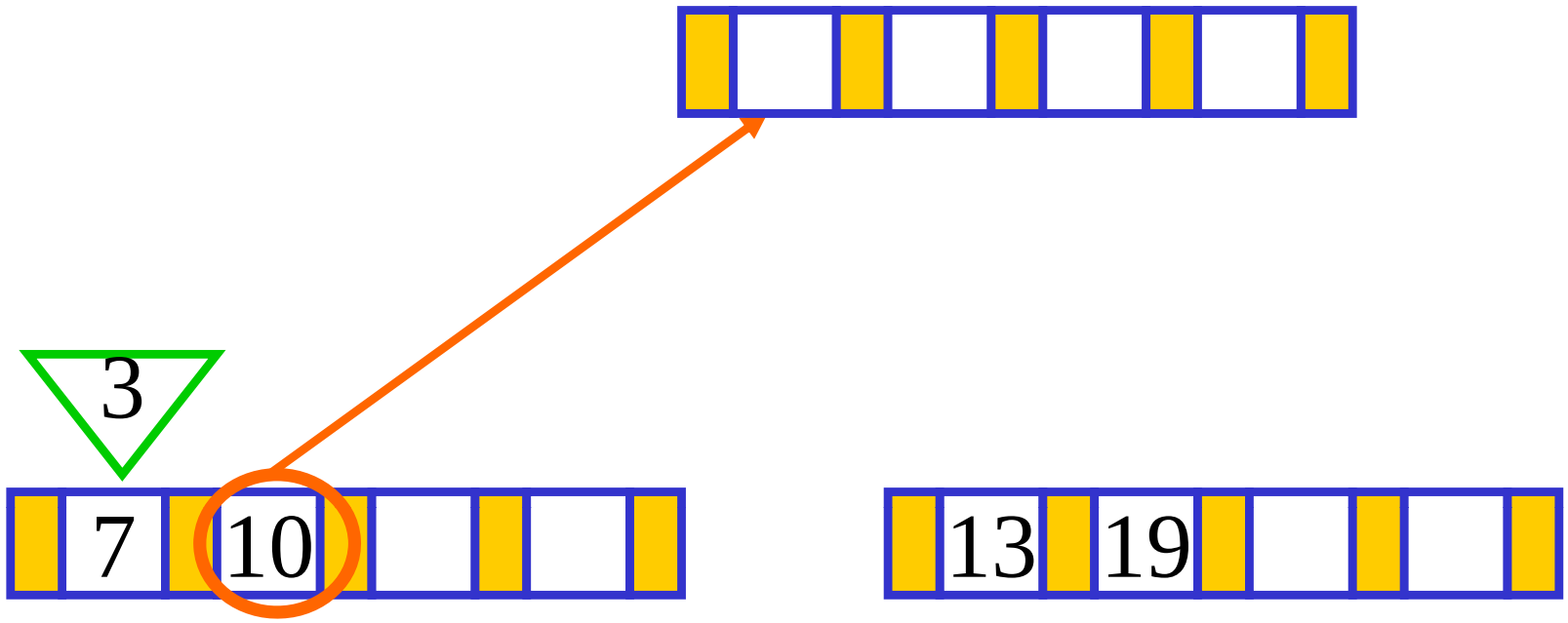
Beispiel 1



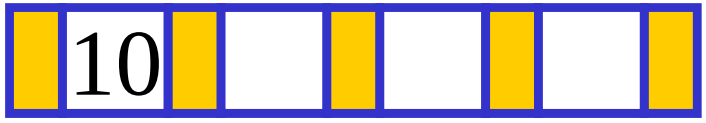
Beispiel 1

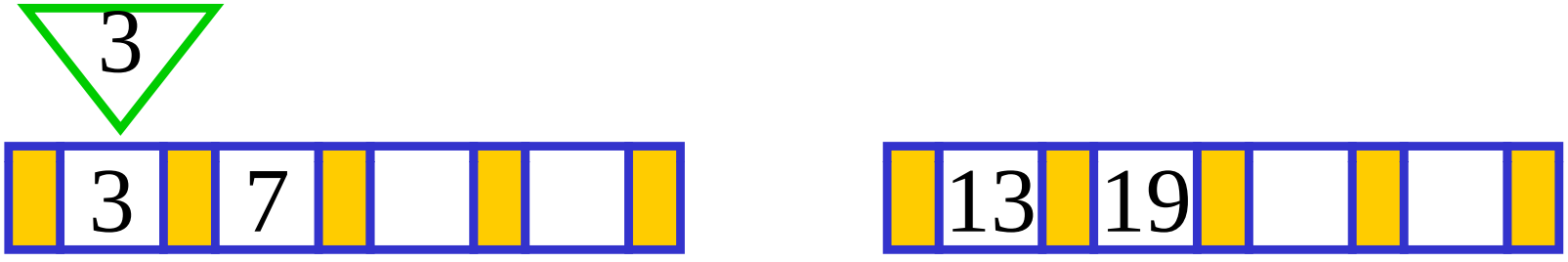


Beispiel 1

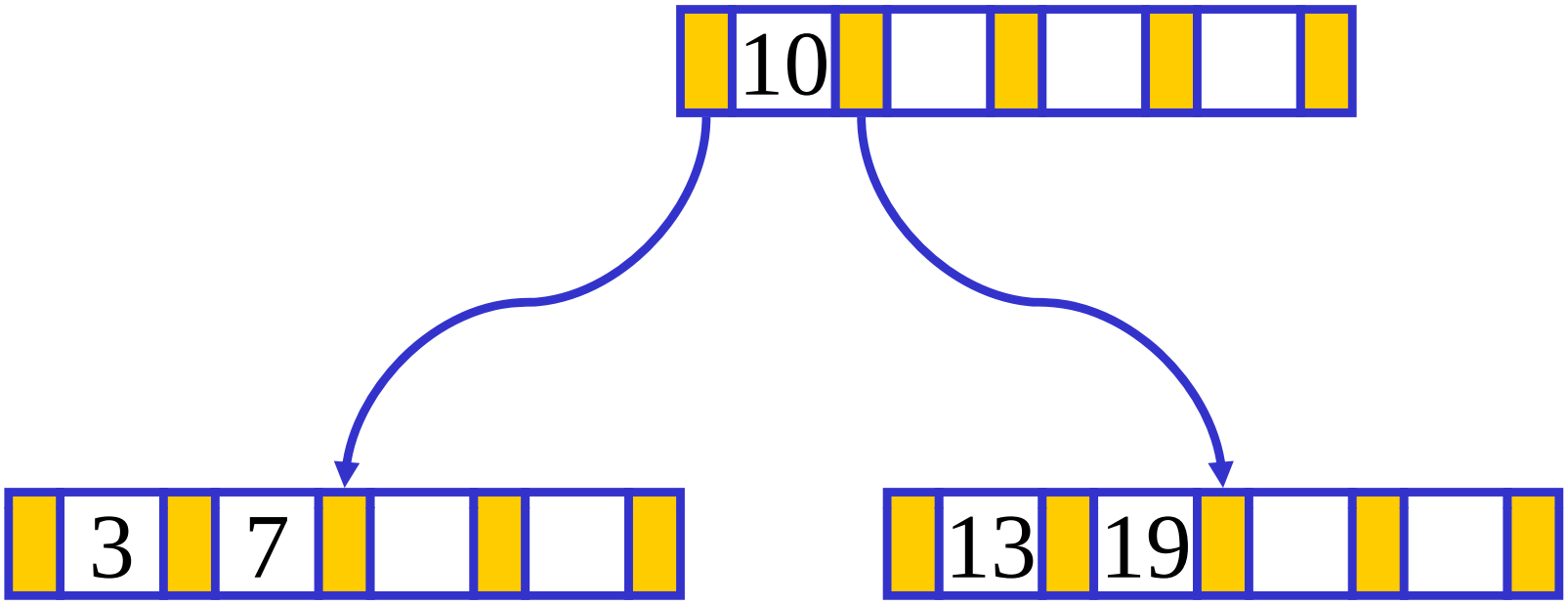


Beispiel 1

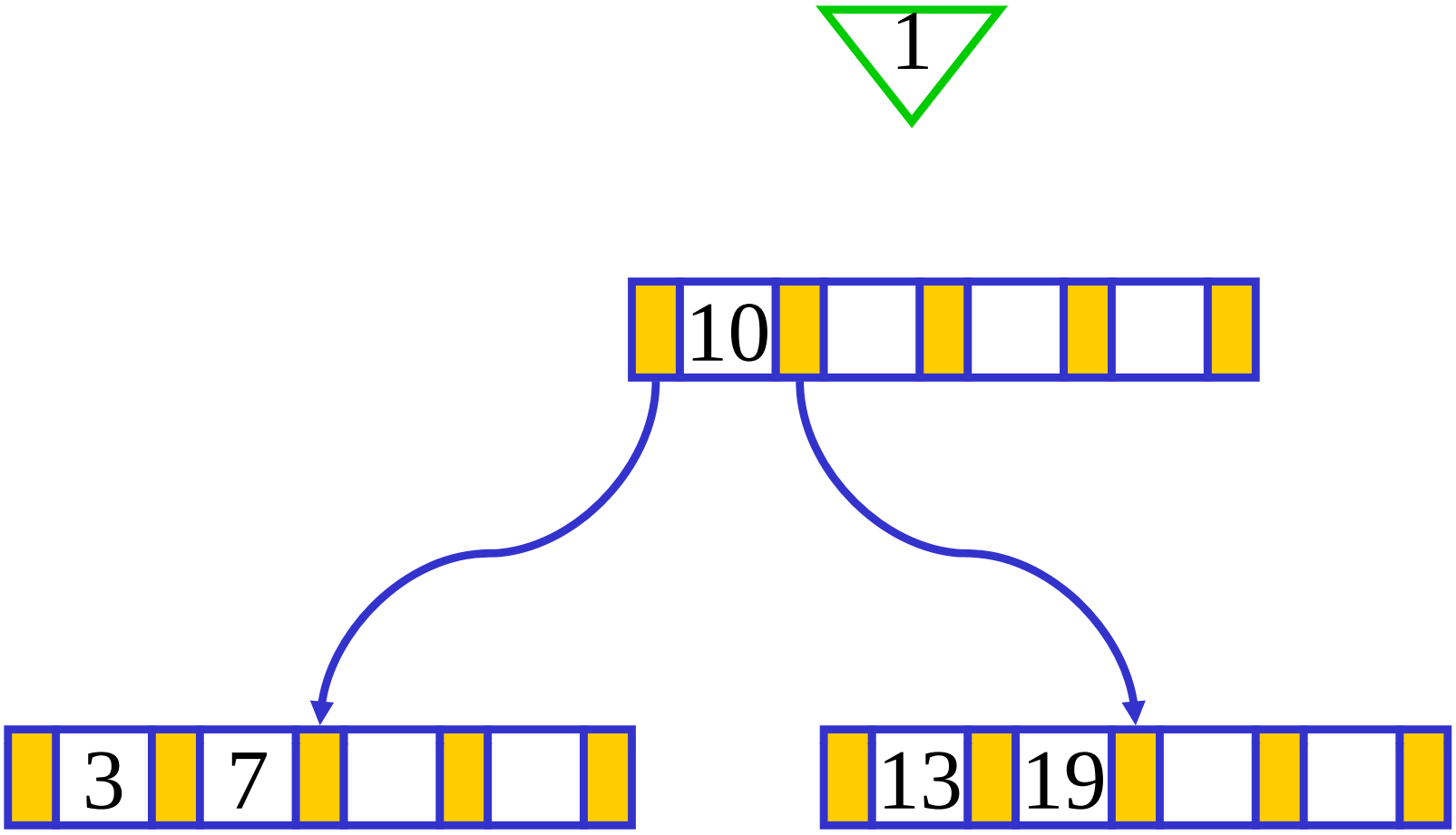




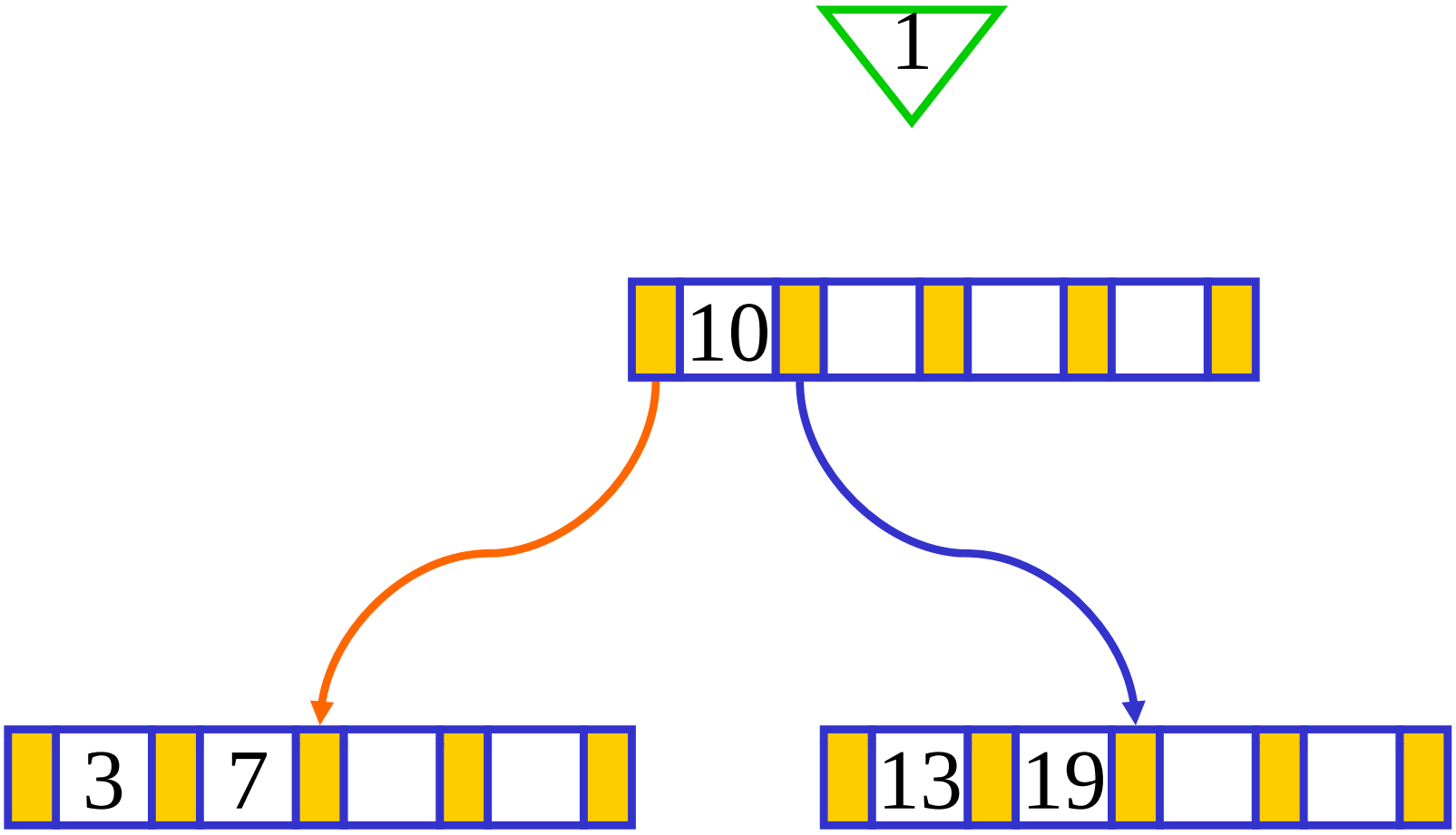
Beispiel 1



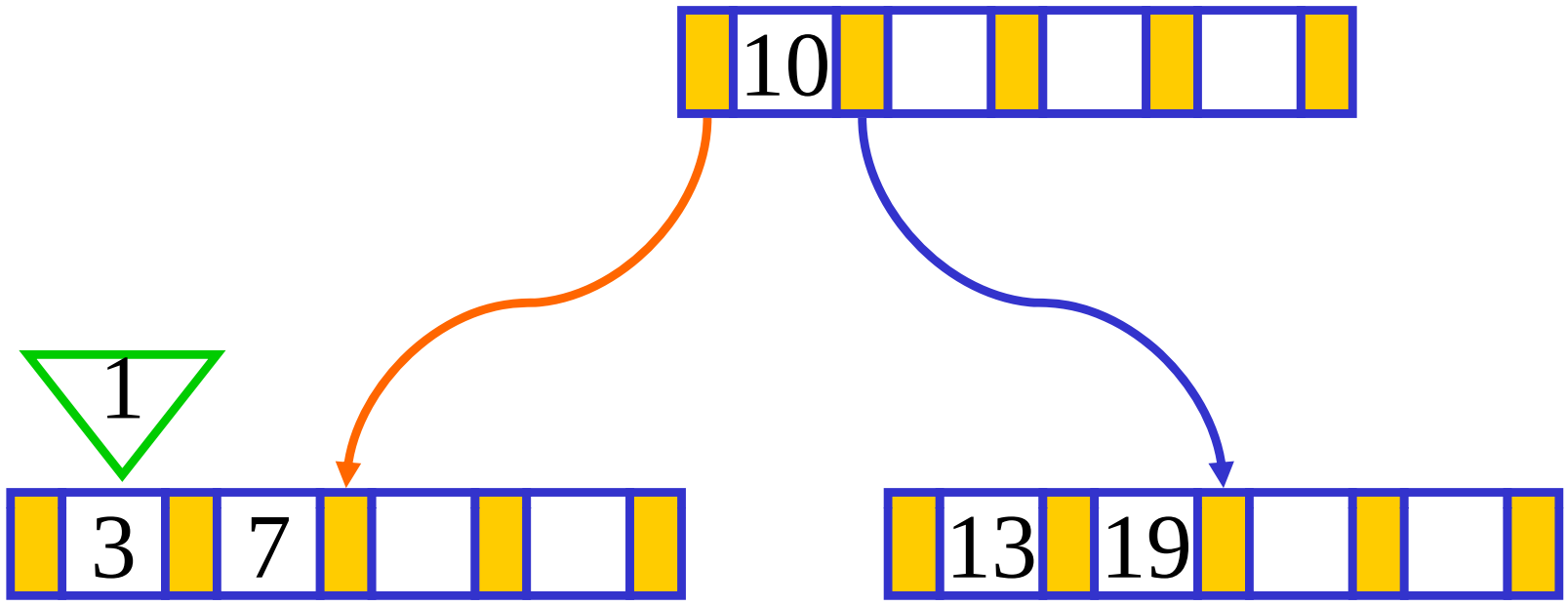
Beispiel 1



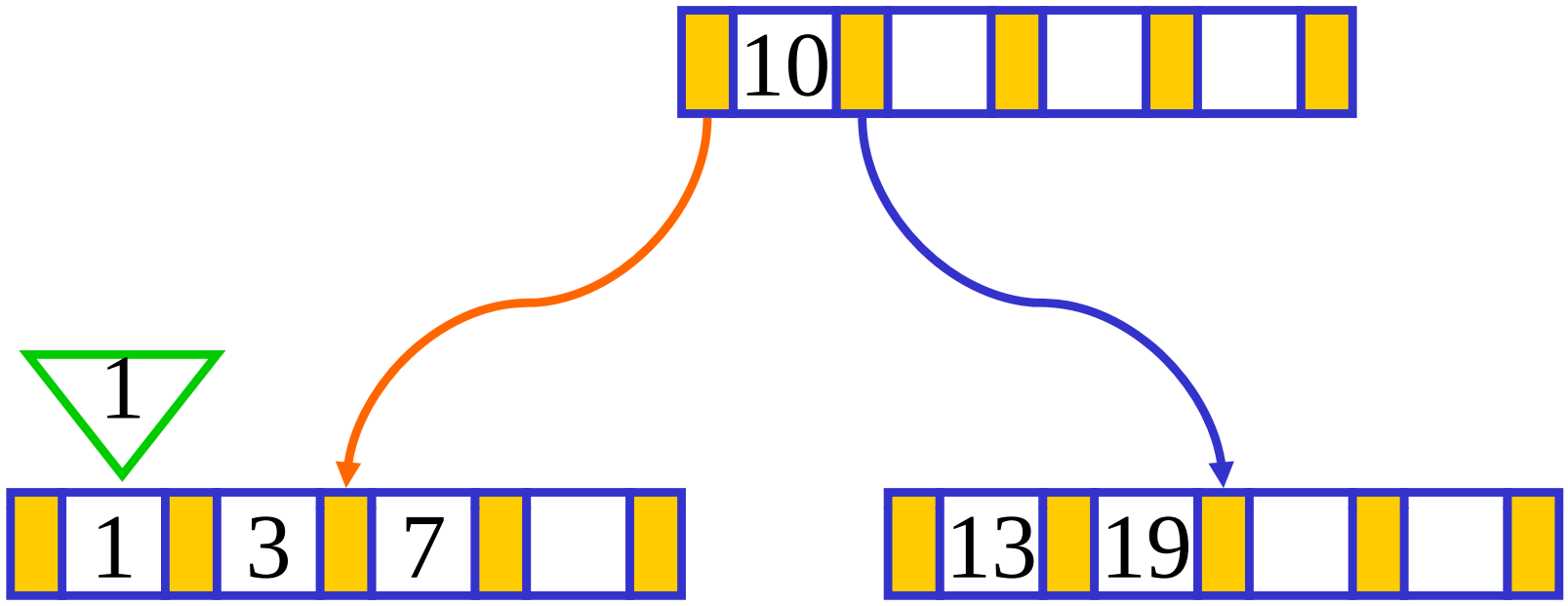
Beispiel 1



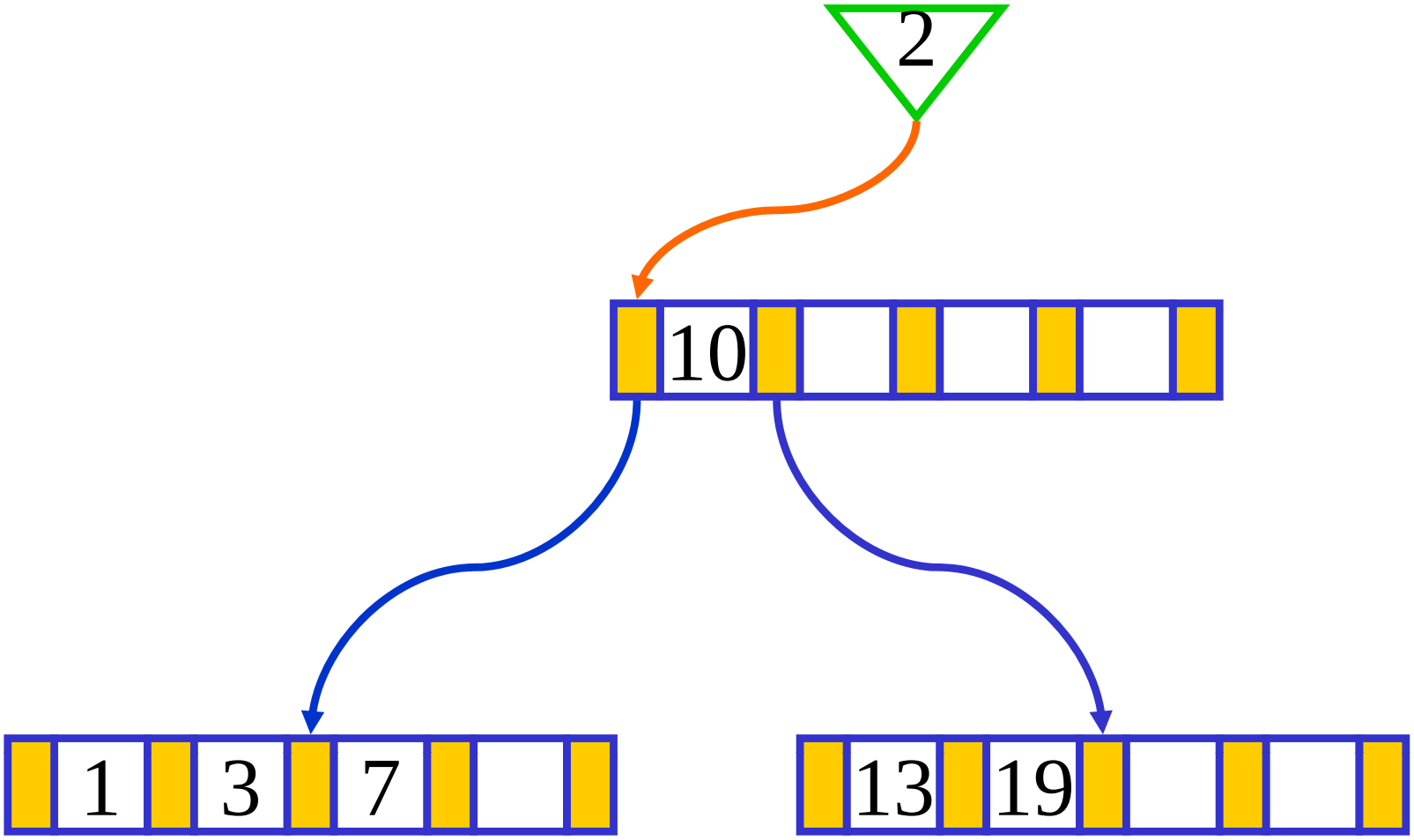
Beispiel 1



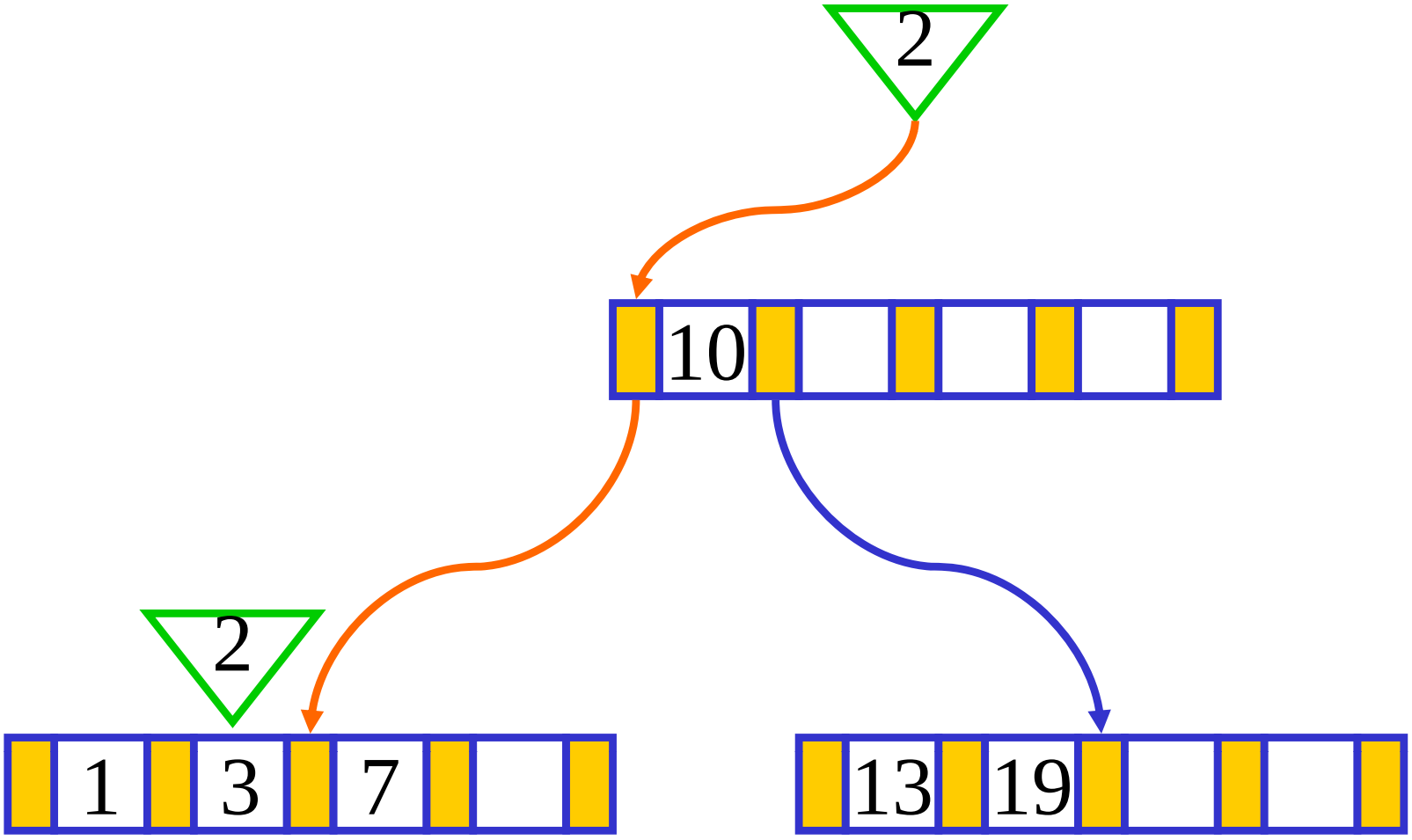
Beispiel 1



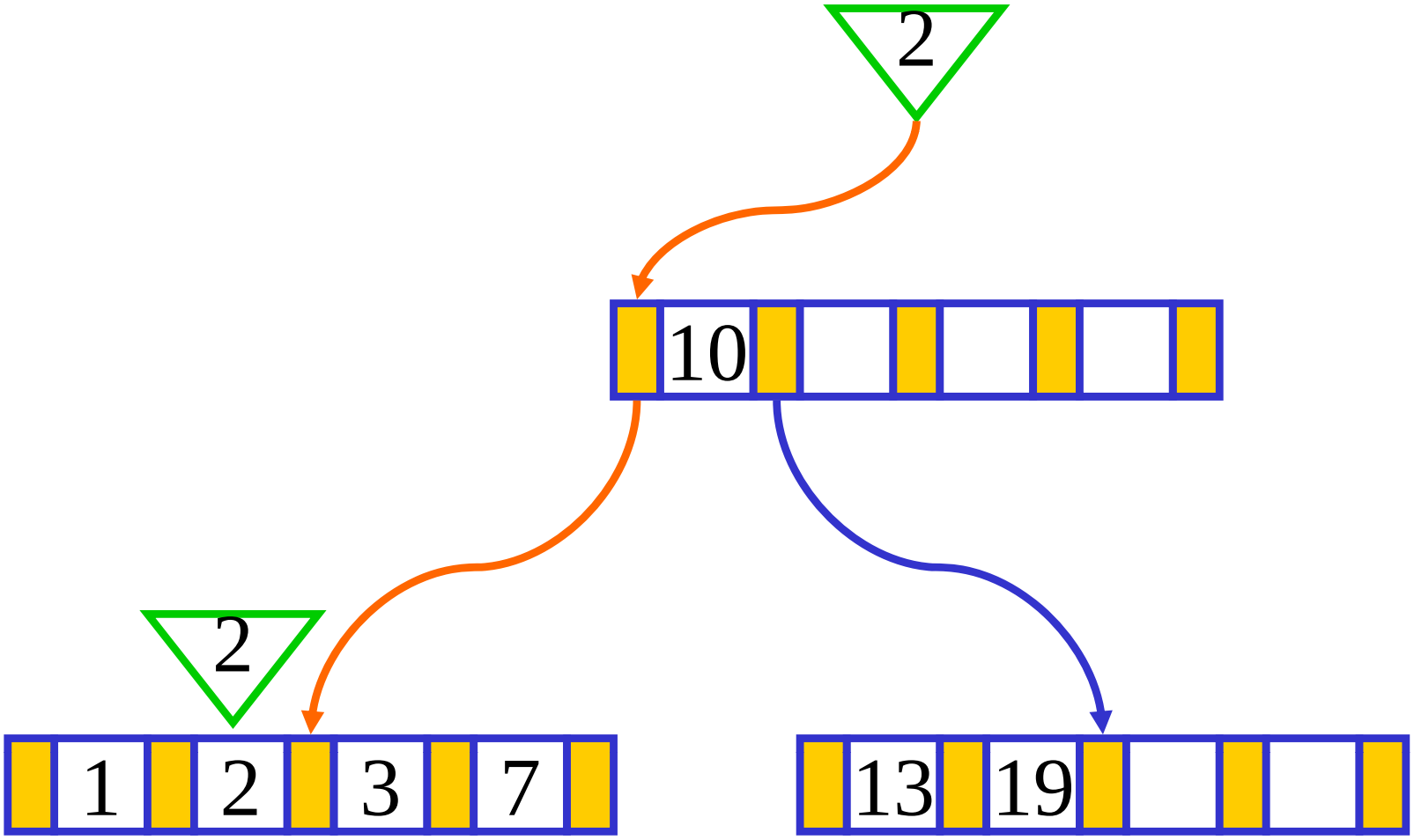
Beispiel 1



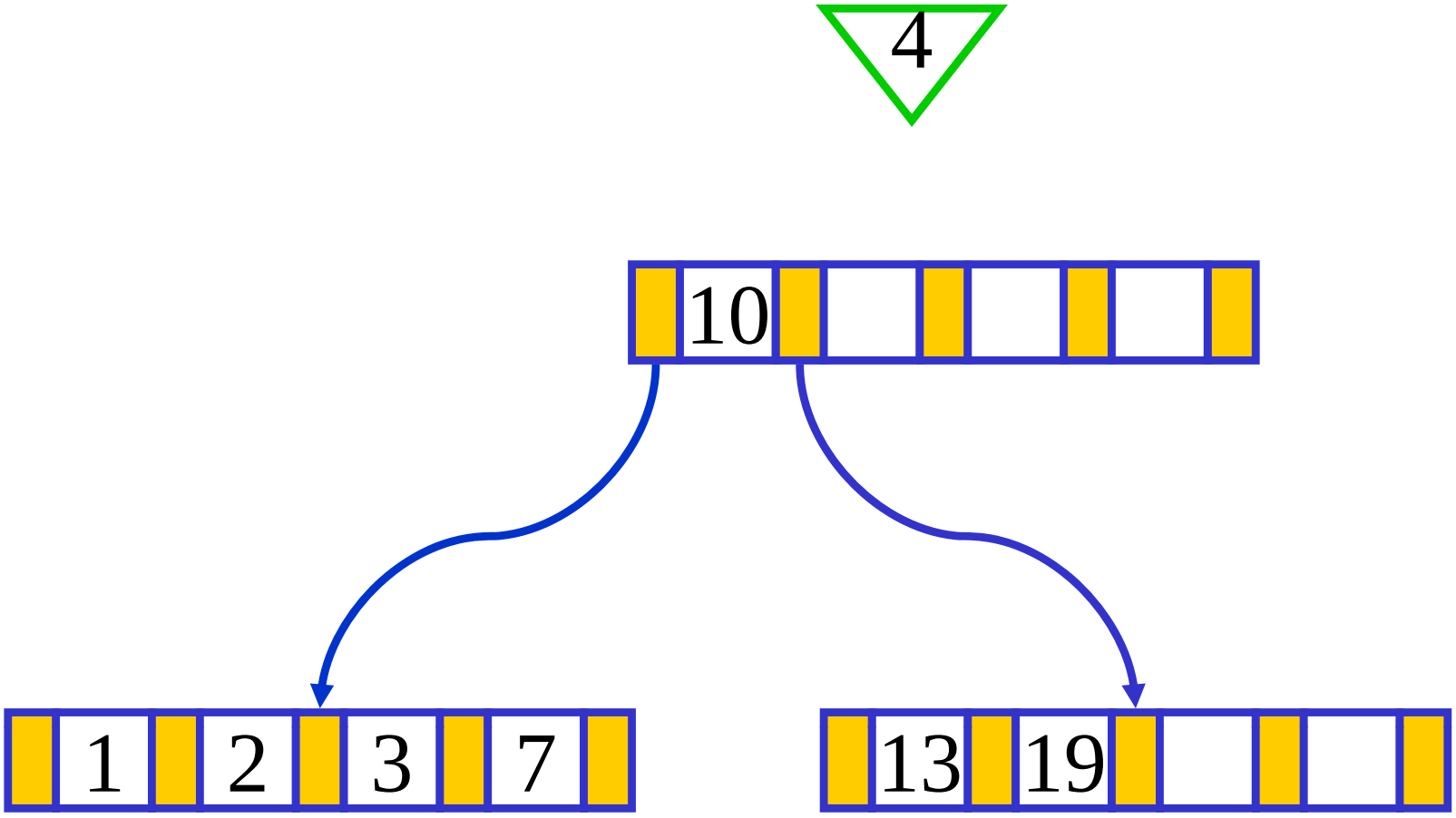
Beispiel 1



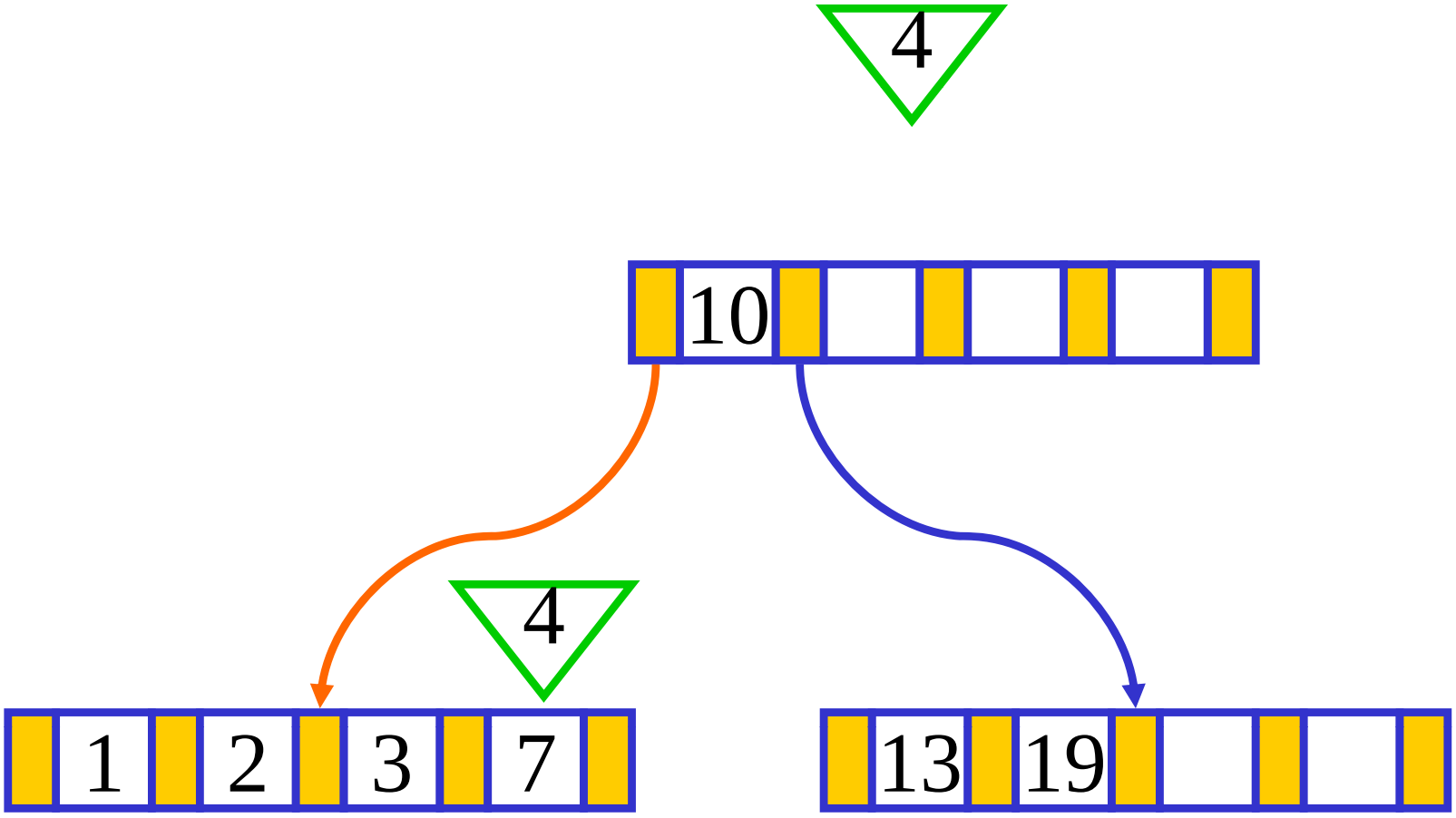
Beispiel 1



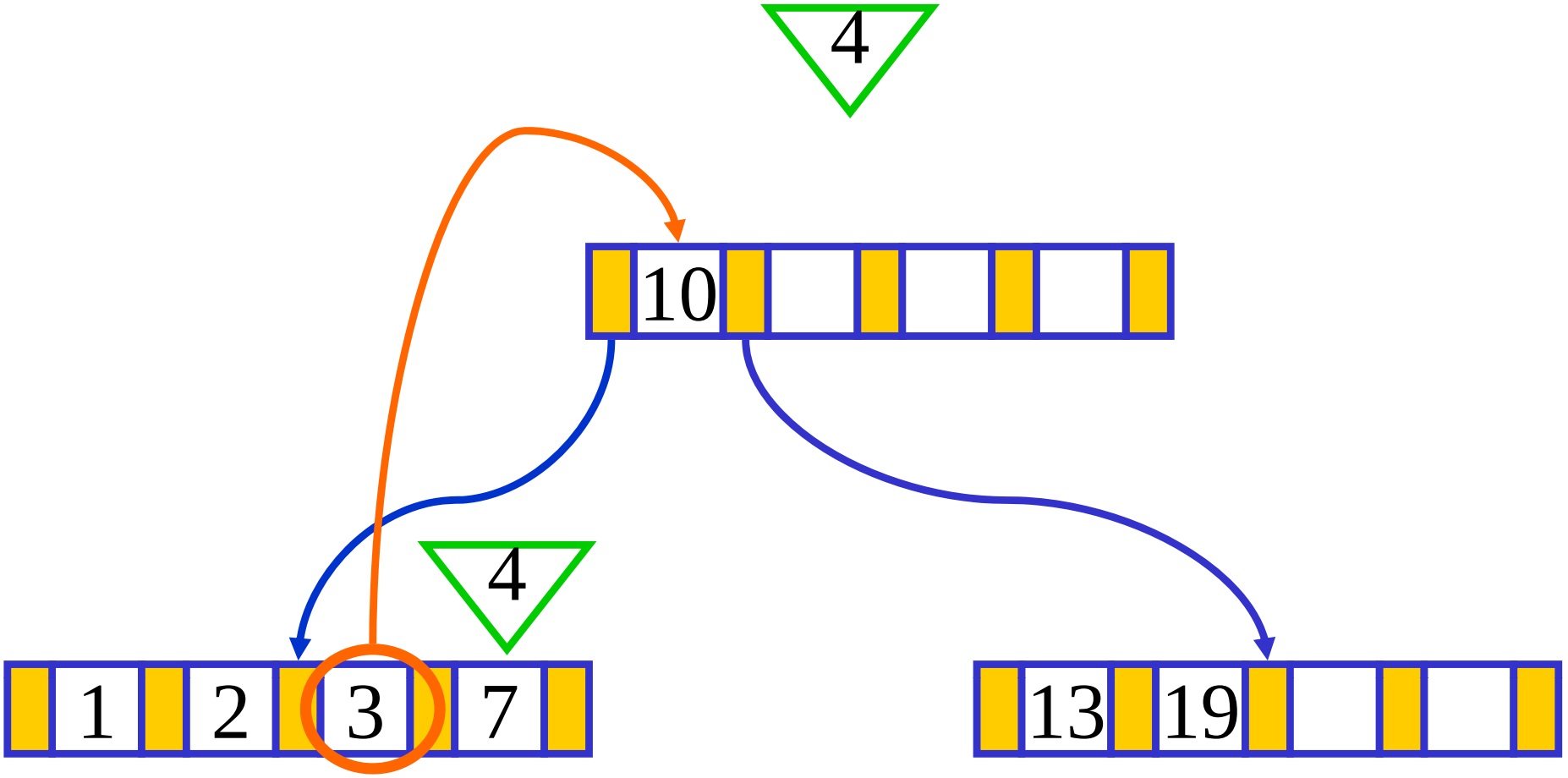
Beispiel 1



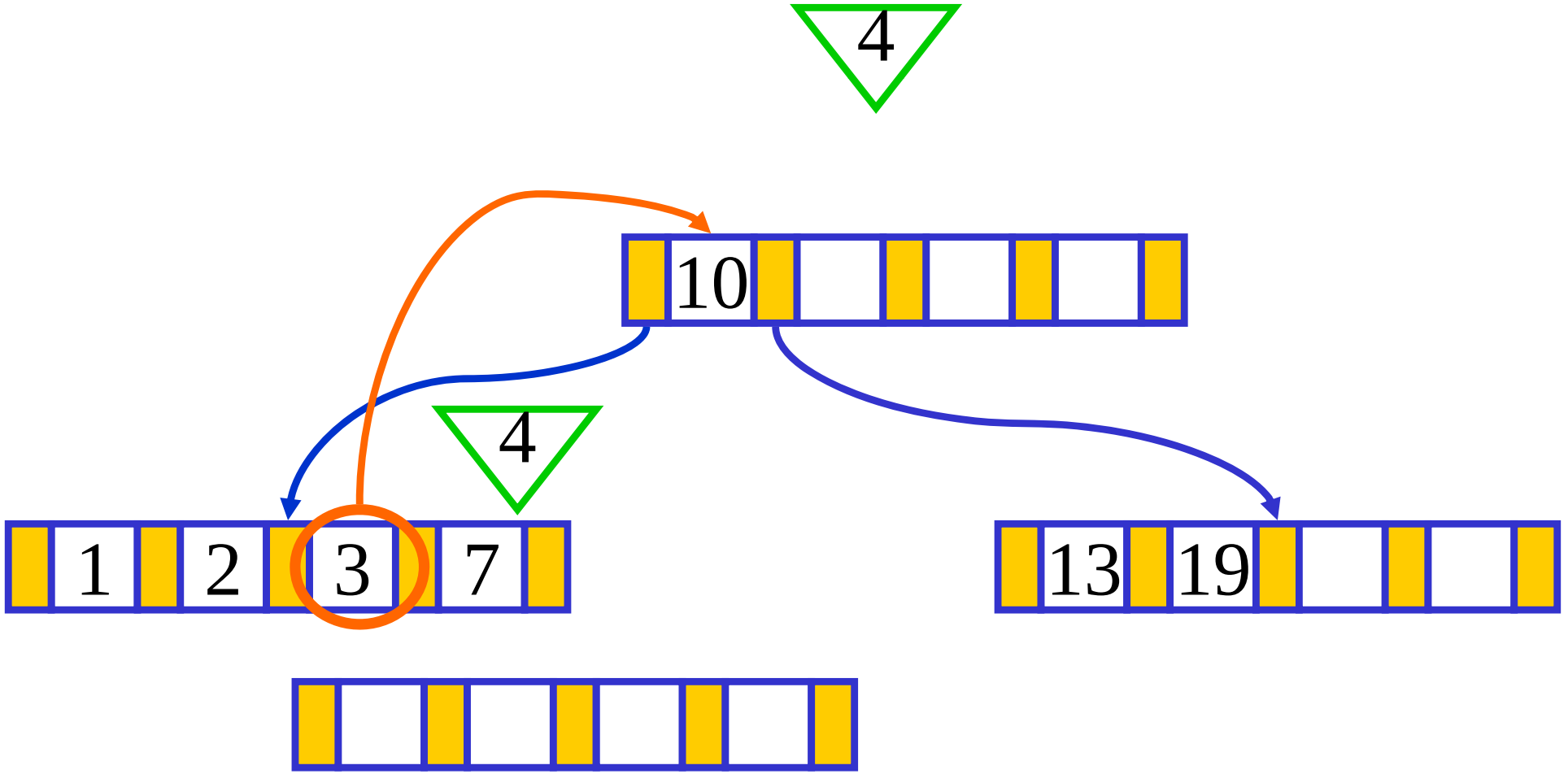
Beispiel 1



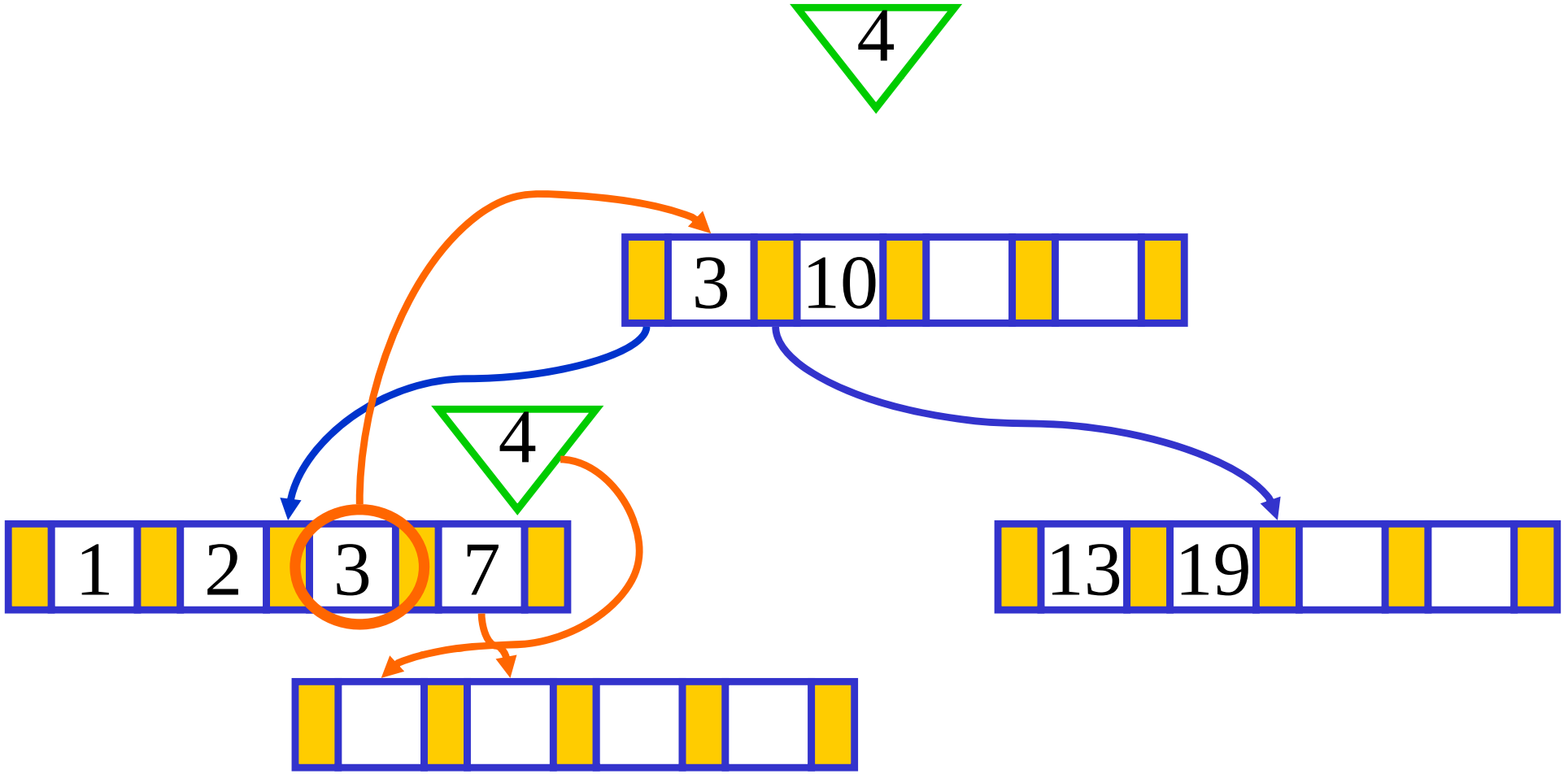
Beispiel 1



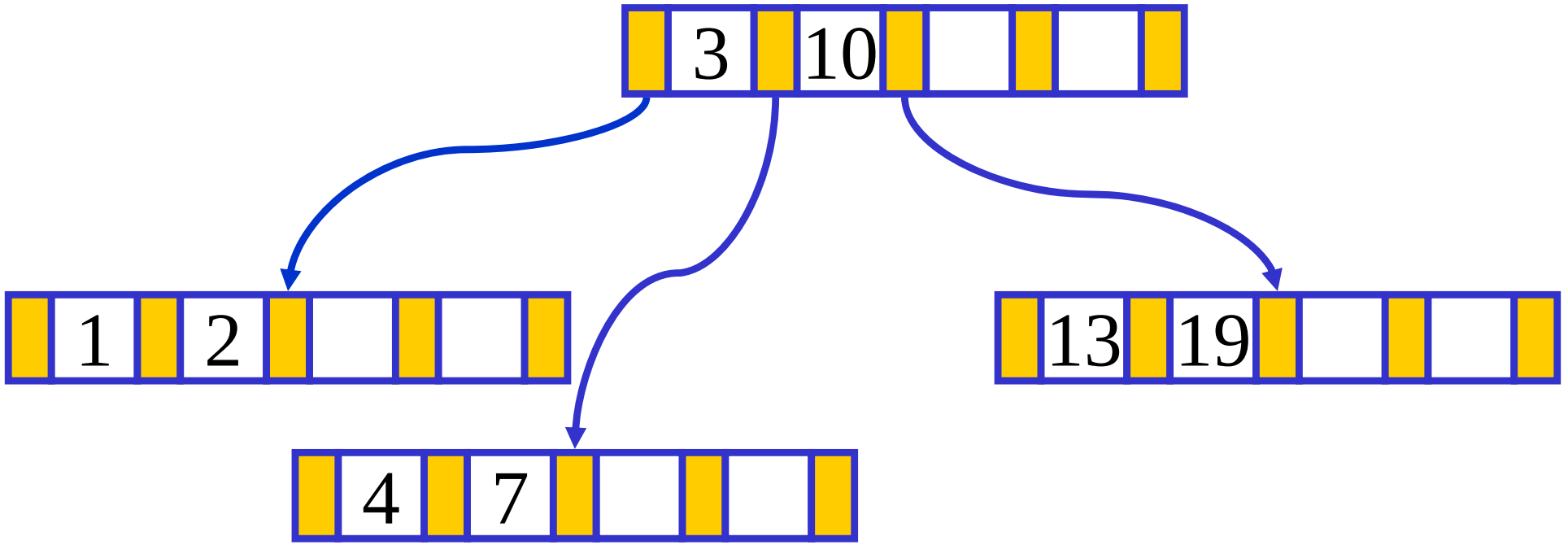
Beispiel 1



Beispiel 1



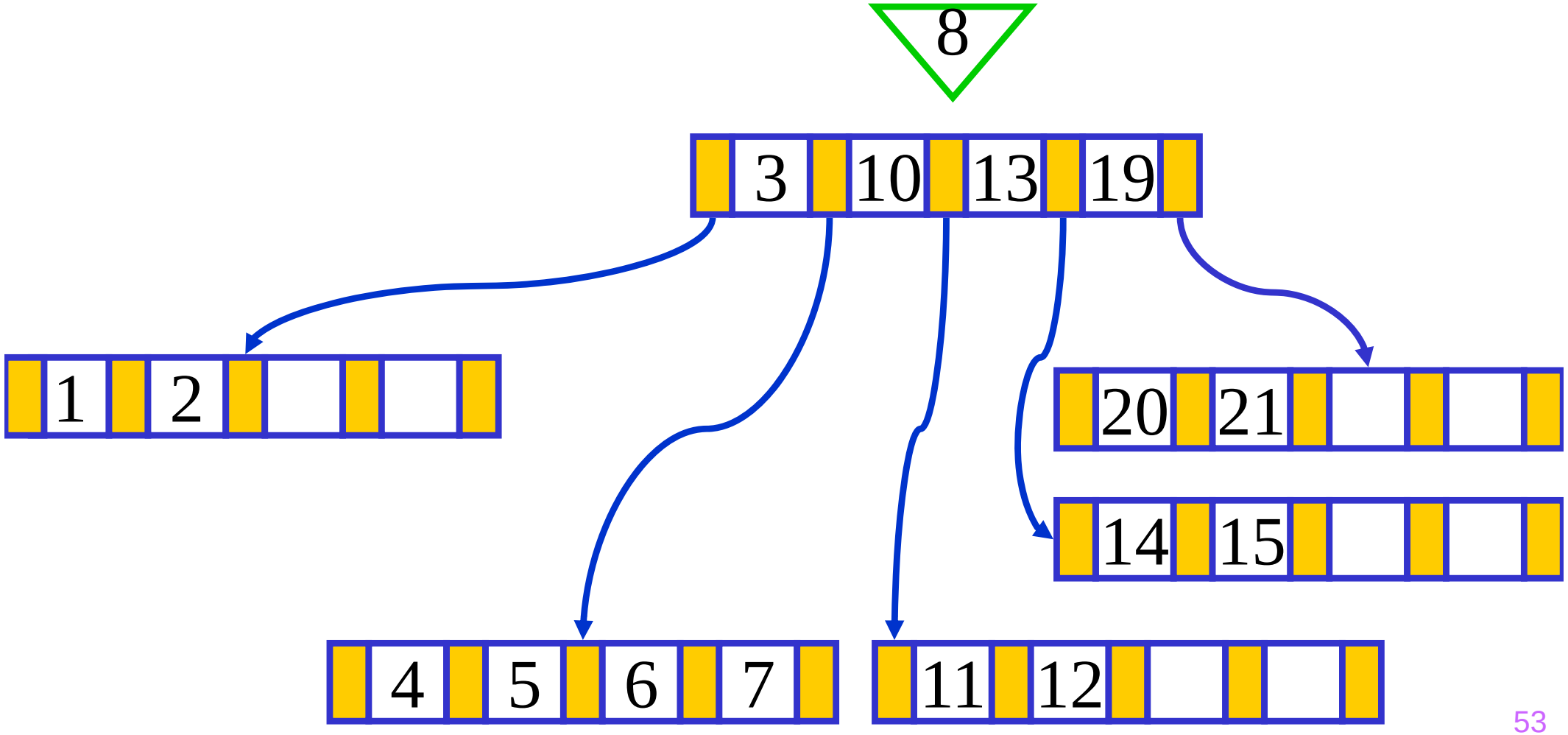
Beispiel 1



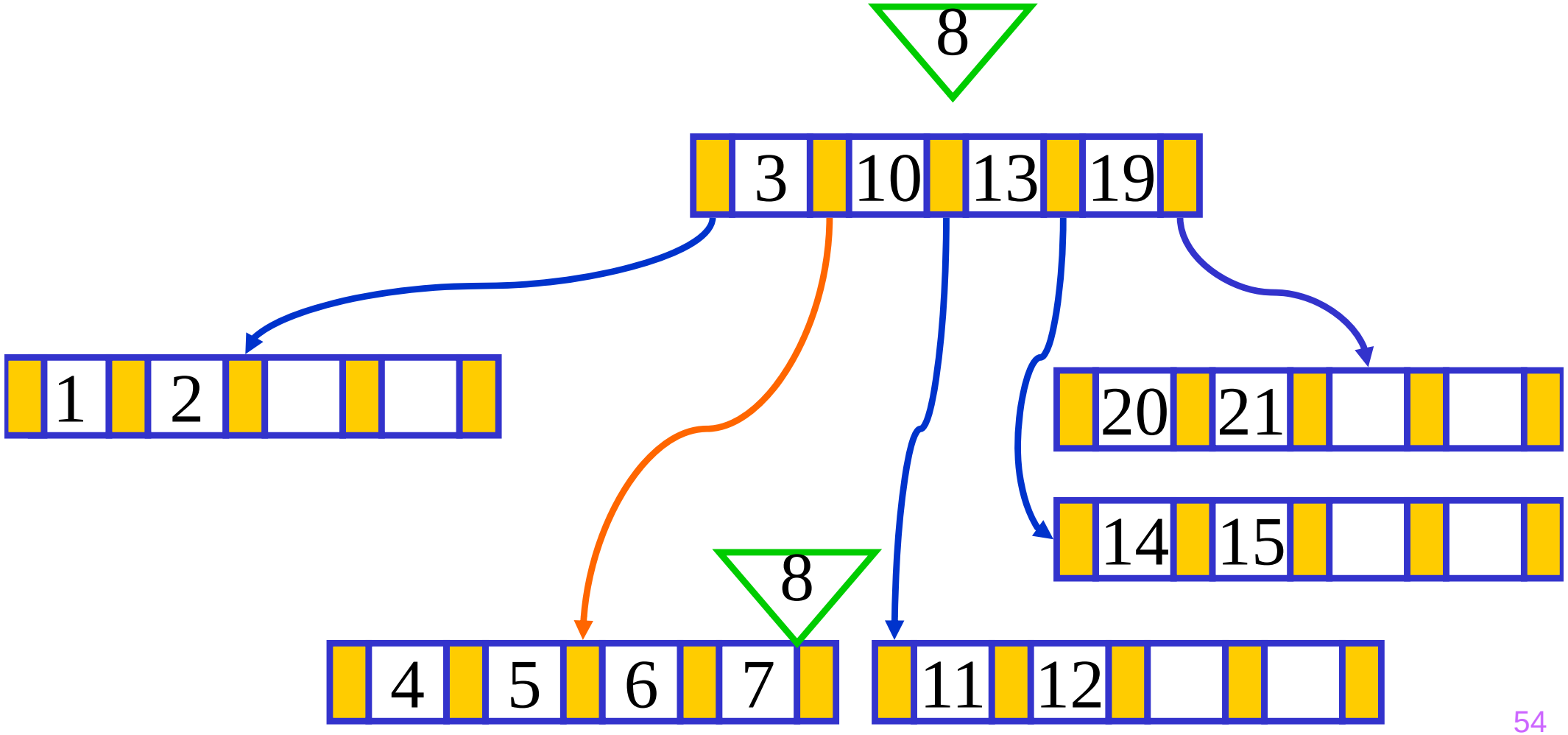
Beispiel 2

- B-Baum vom Grad k = 2
- Ausgangssituation:
  - Baum der Tiefe 2
  - Die Wurzel hat bereits den maximalen Füllstand erreicht
- **Einfügen eines weiteren Knotens** führt zur Teilung eines Blattknotens
- Teilung des Blattknotens führt nun zur **Teilung der Wurzel**
- Resultat: Baum der Tiefe 3

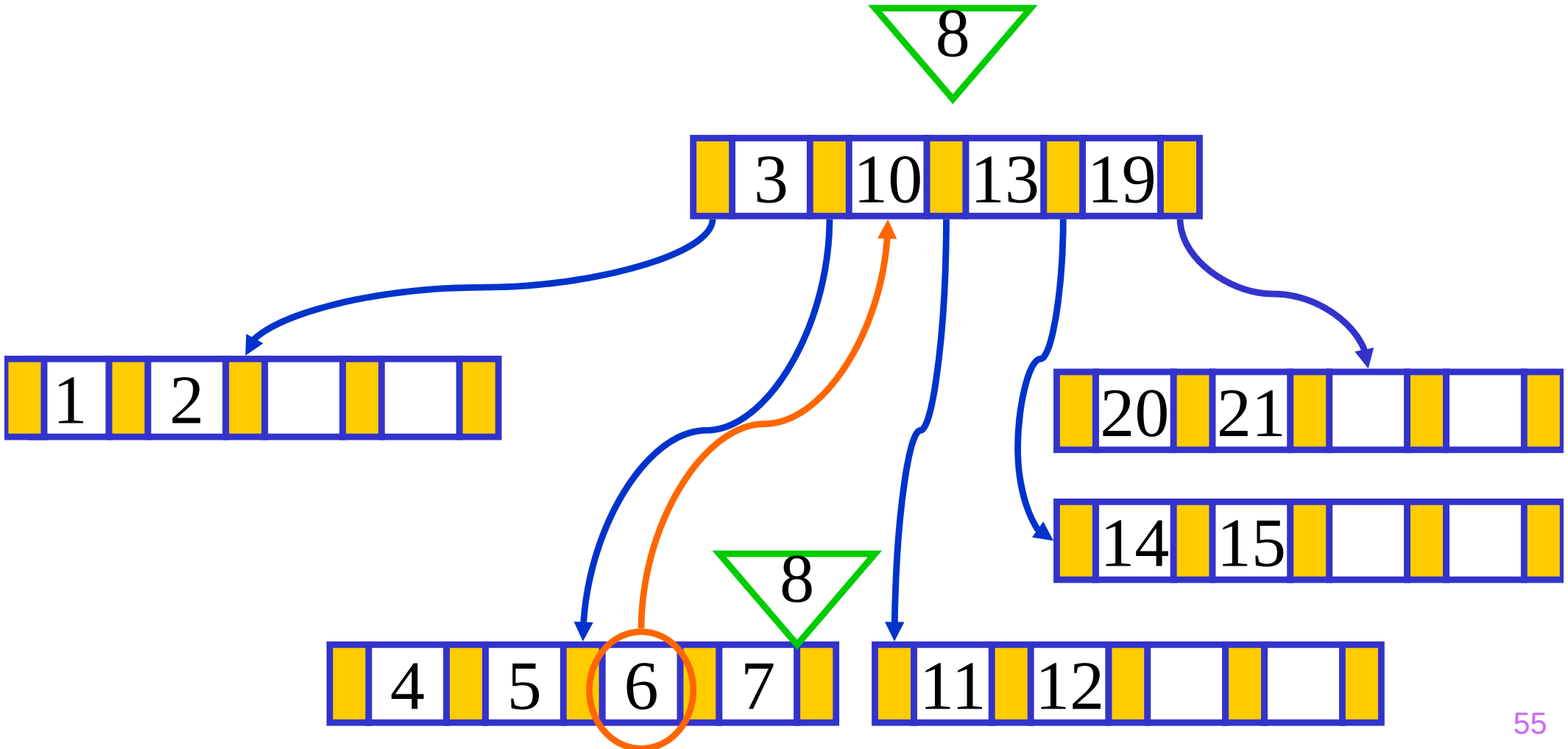
Beispiel 2



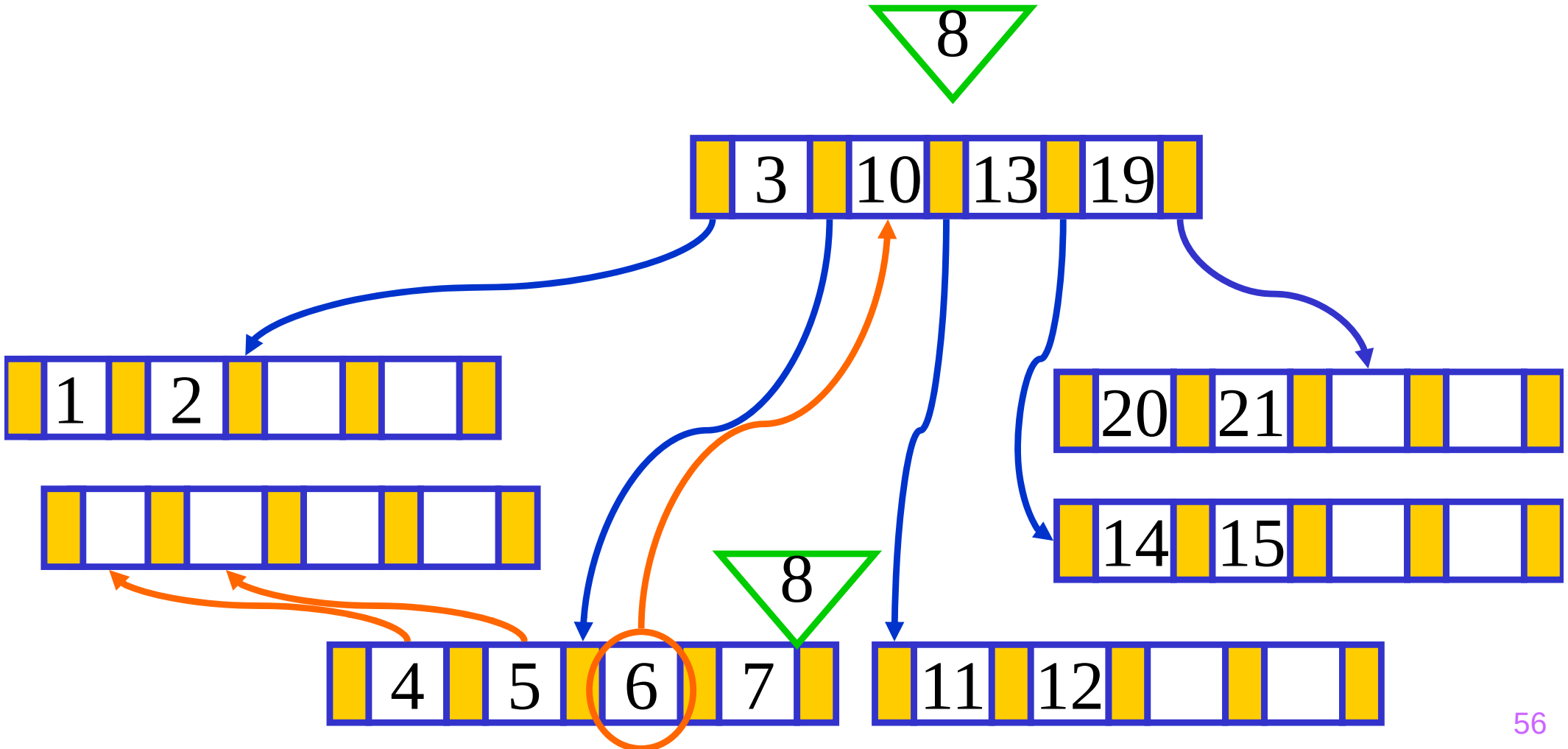
Beispiel 2



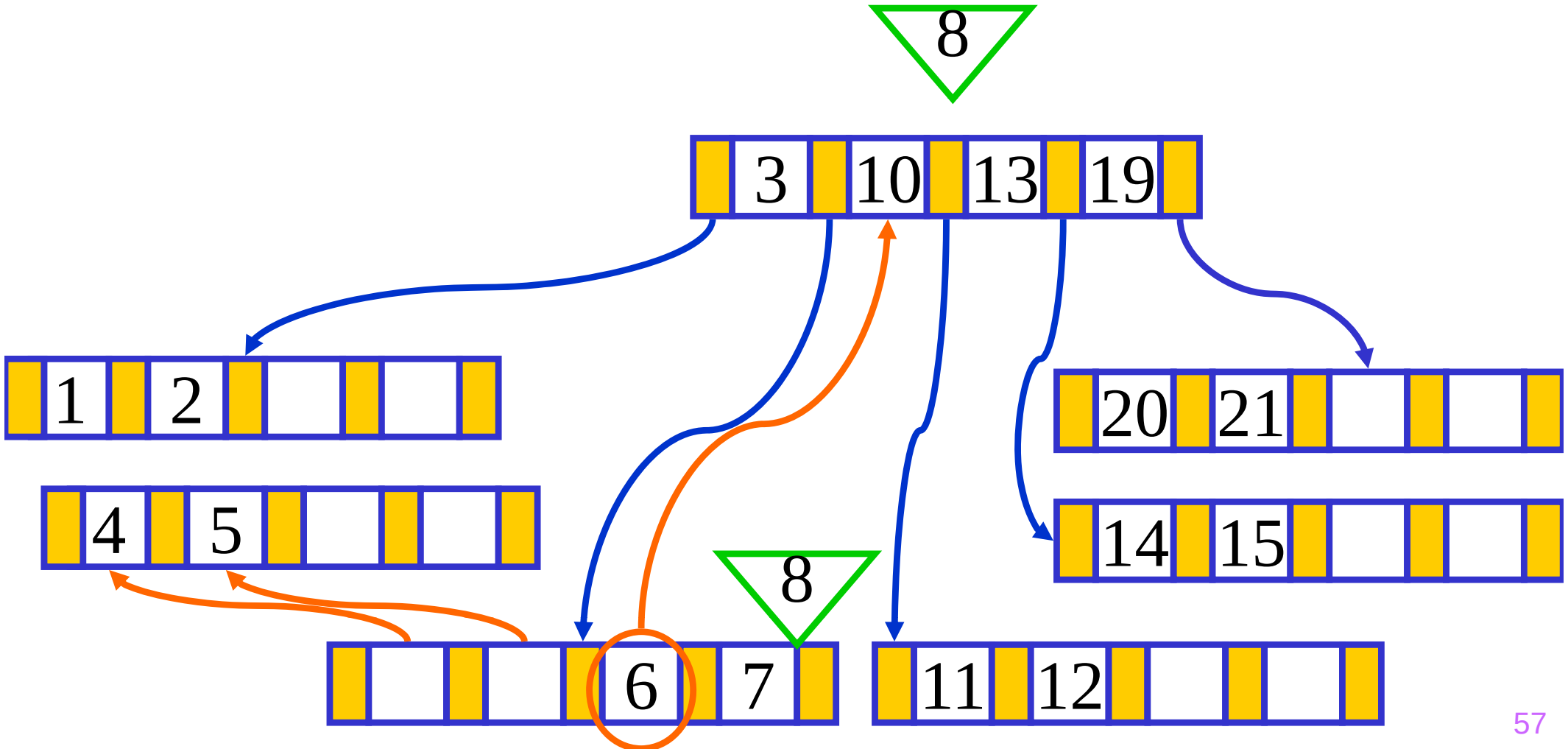
Beispiel 2



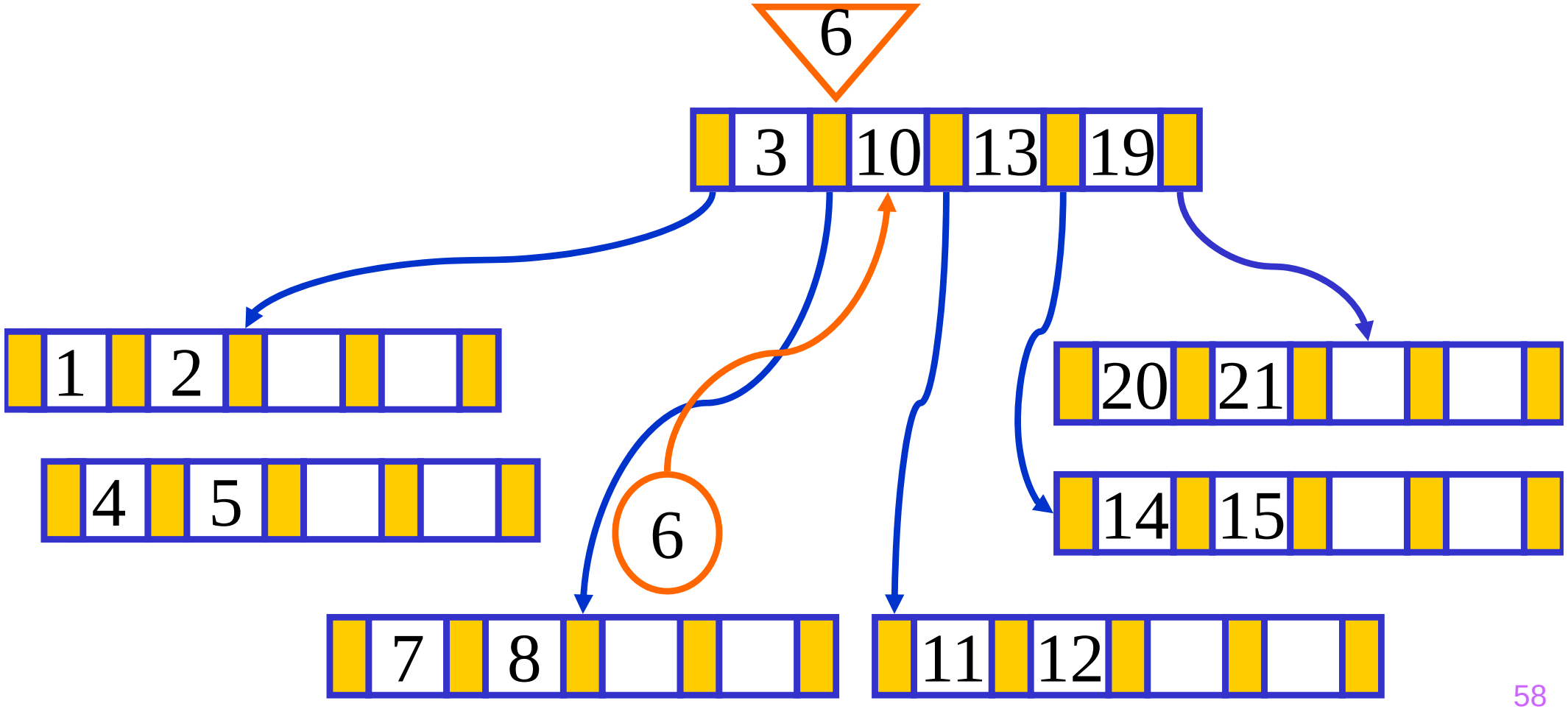
Beispiel 2



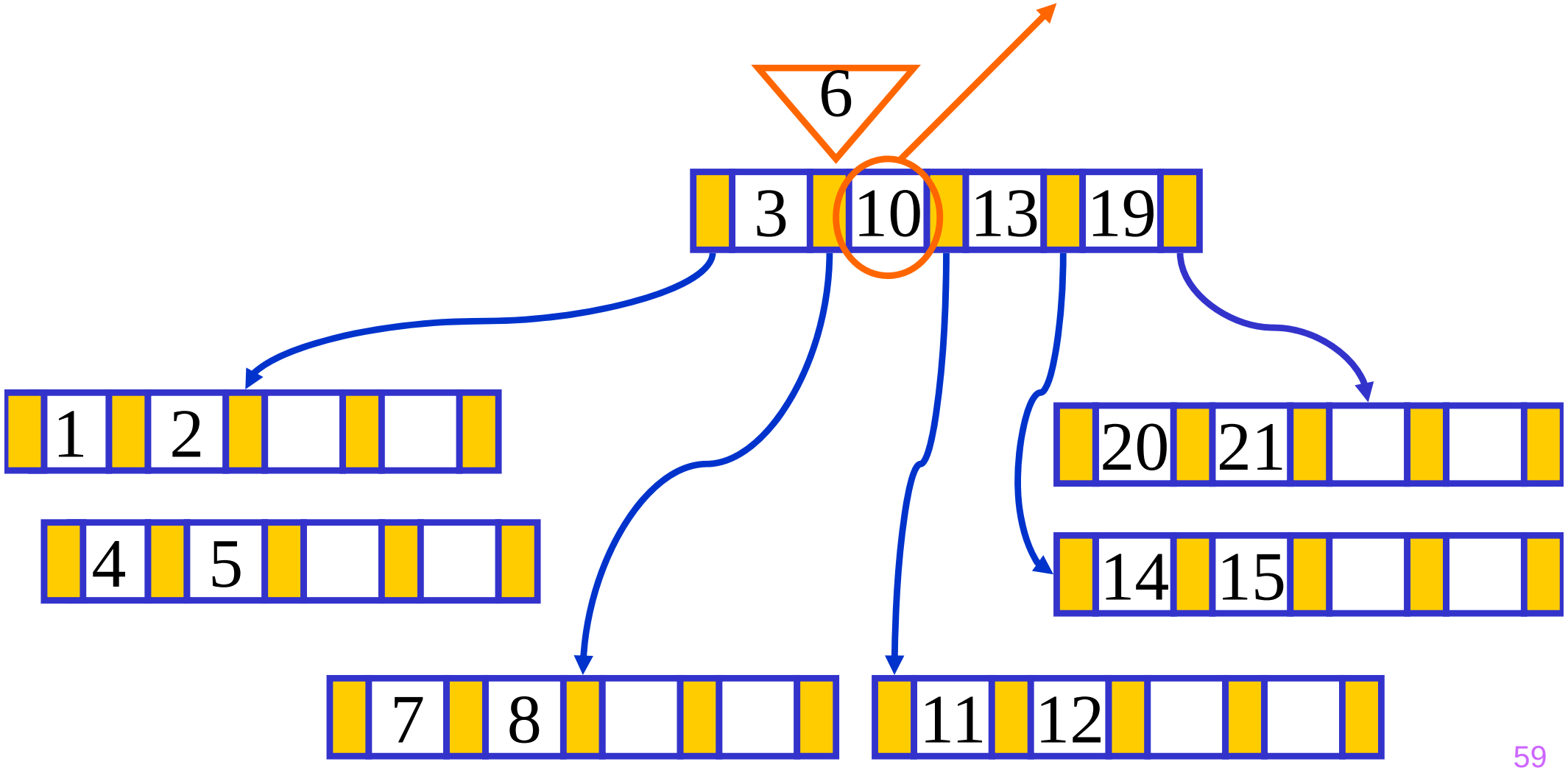
Beispiel 2



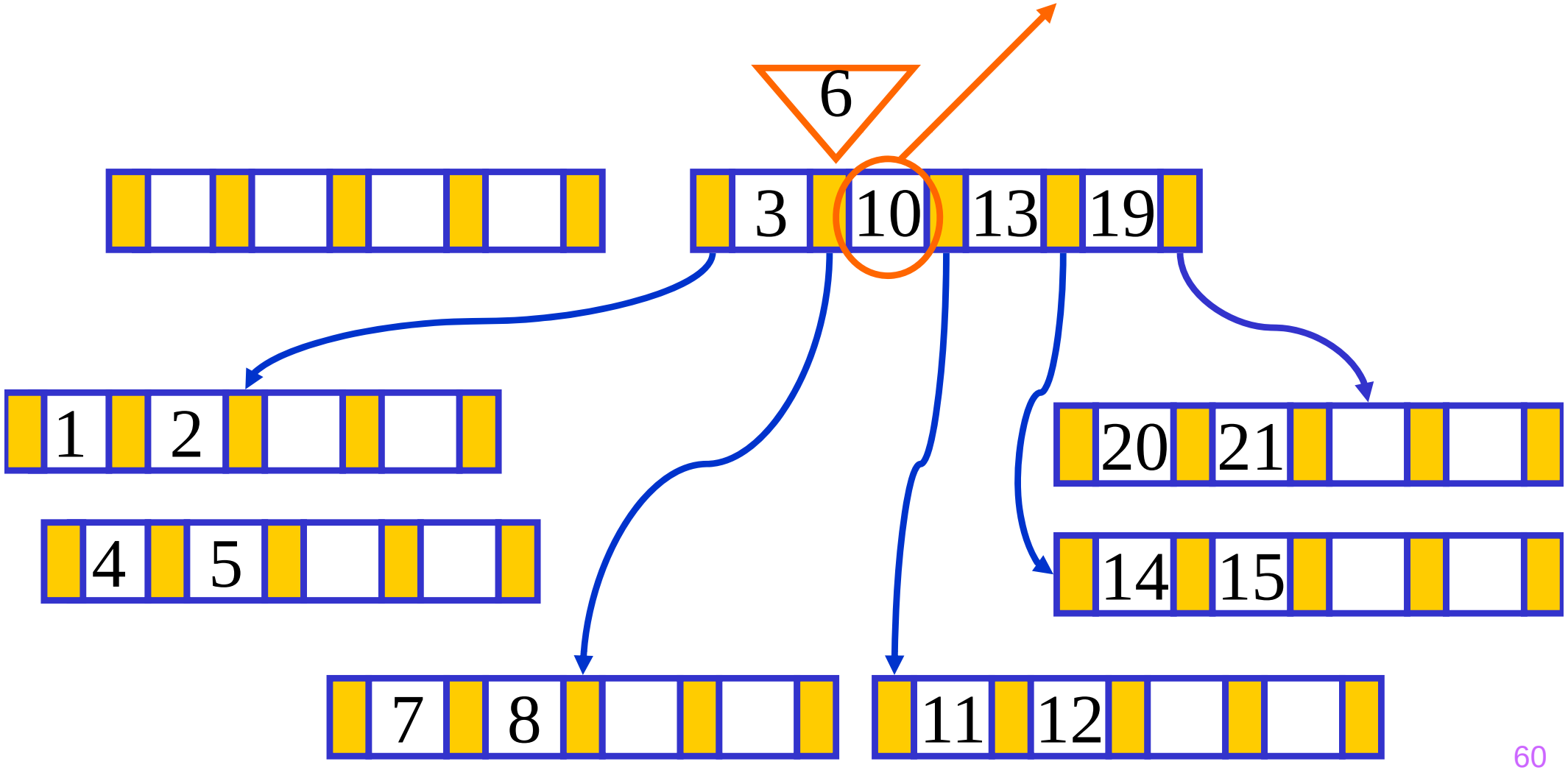
Beispiel 2



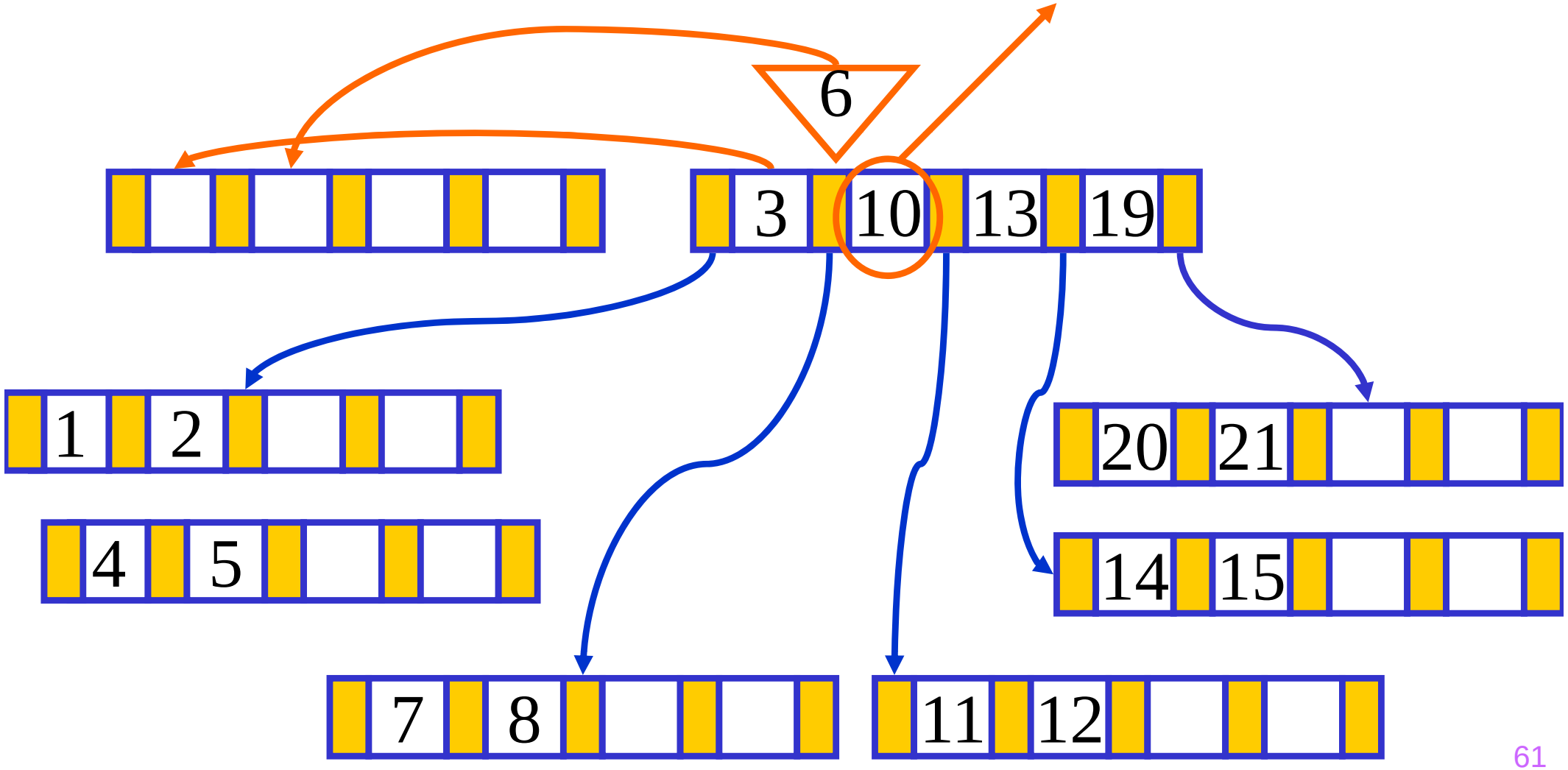
Beispiel 2



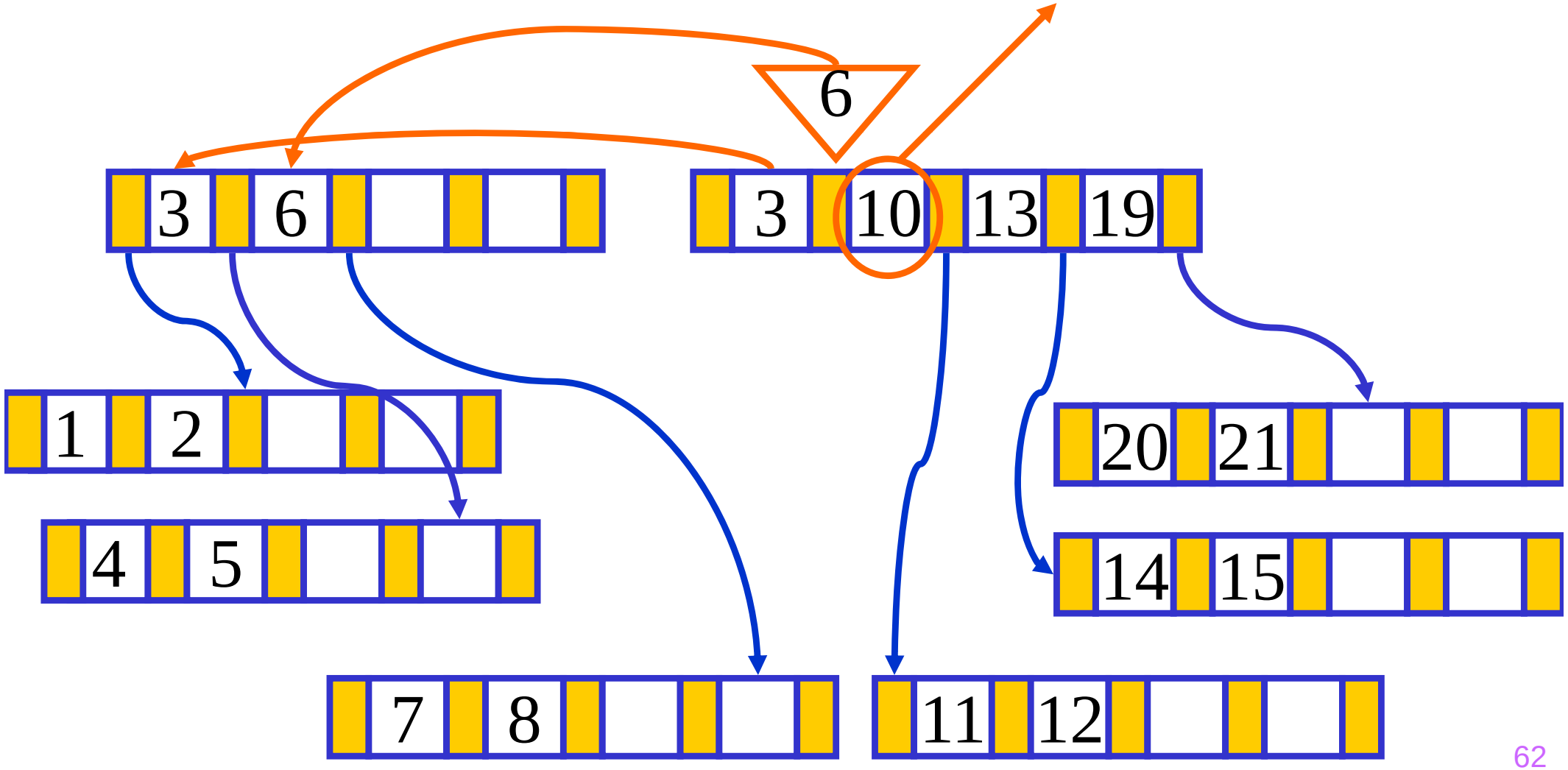
Beispiel 2



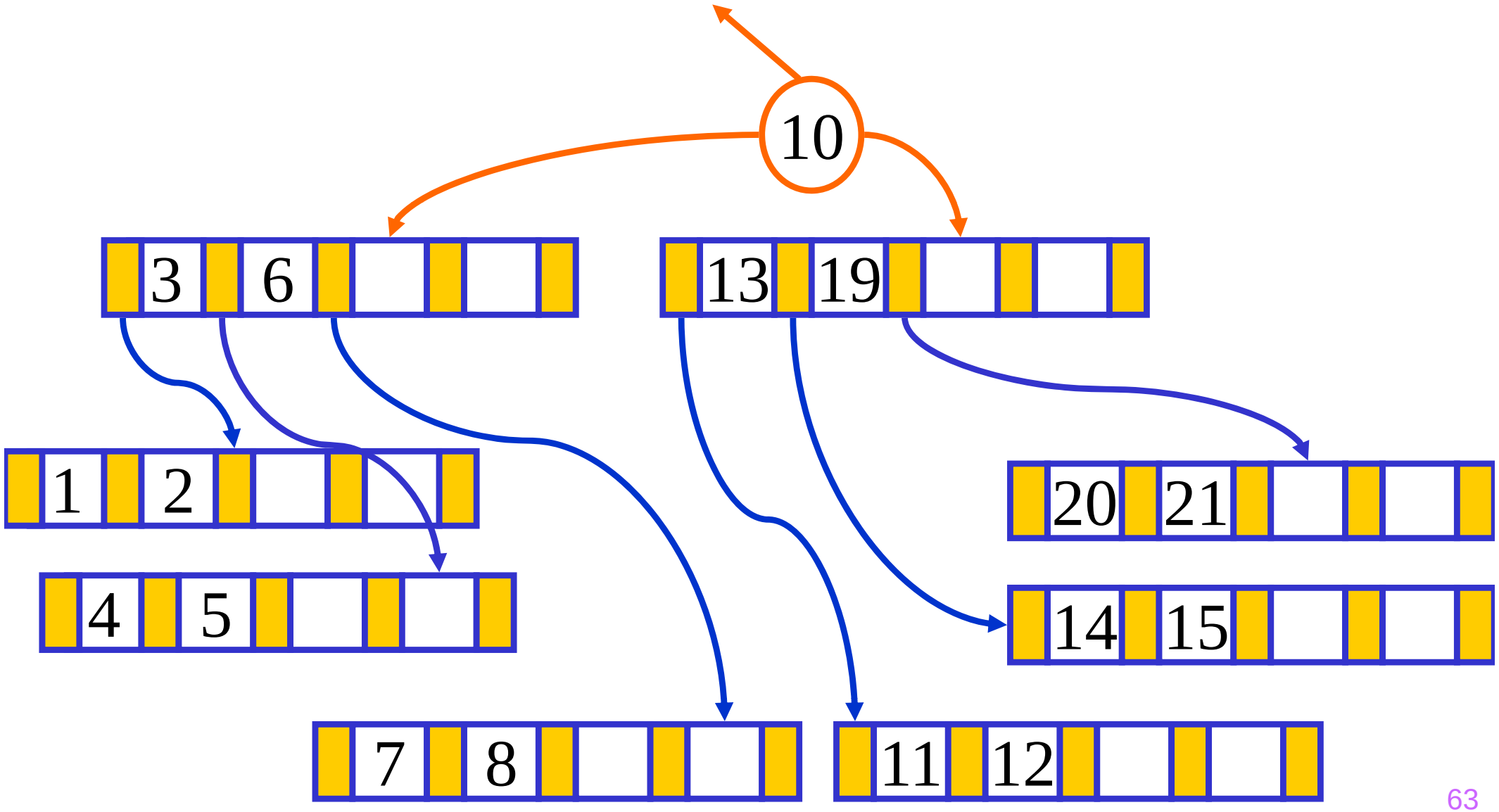
Beispiel 2



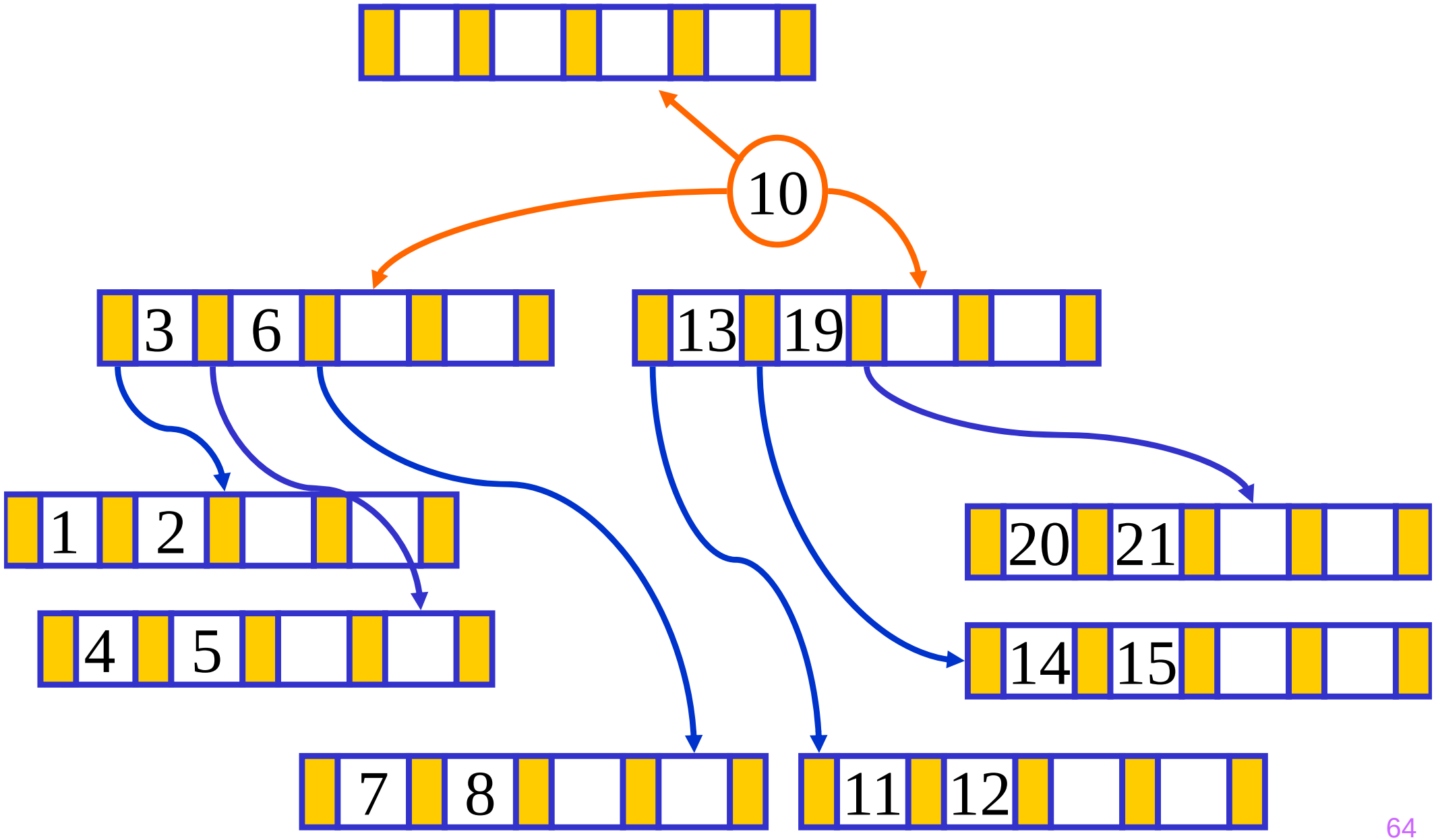
Beispiel 2



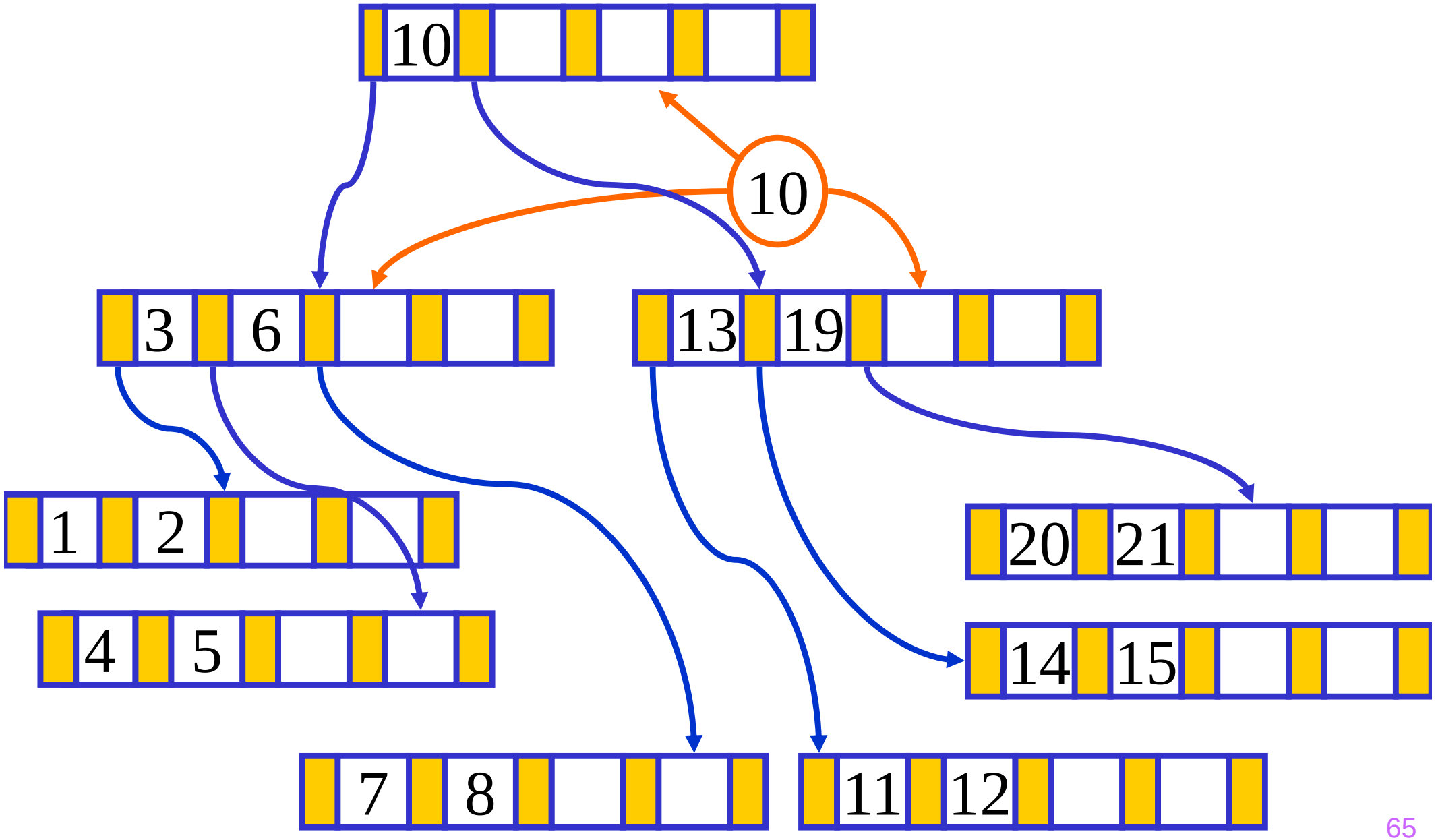
Beispiel 2



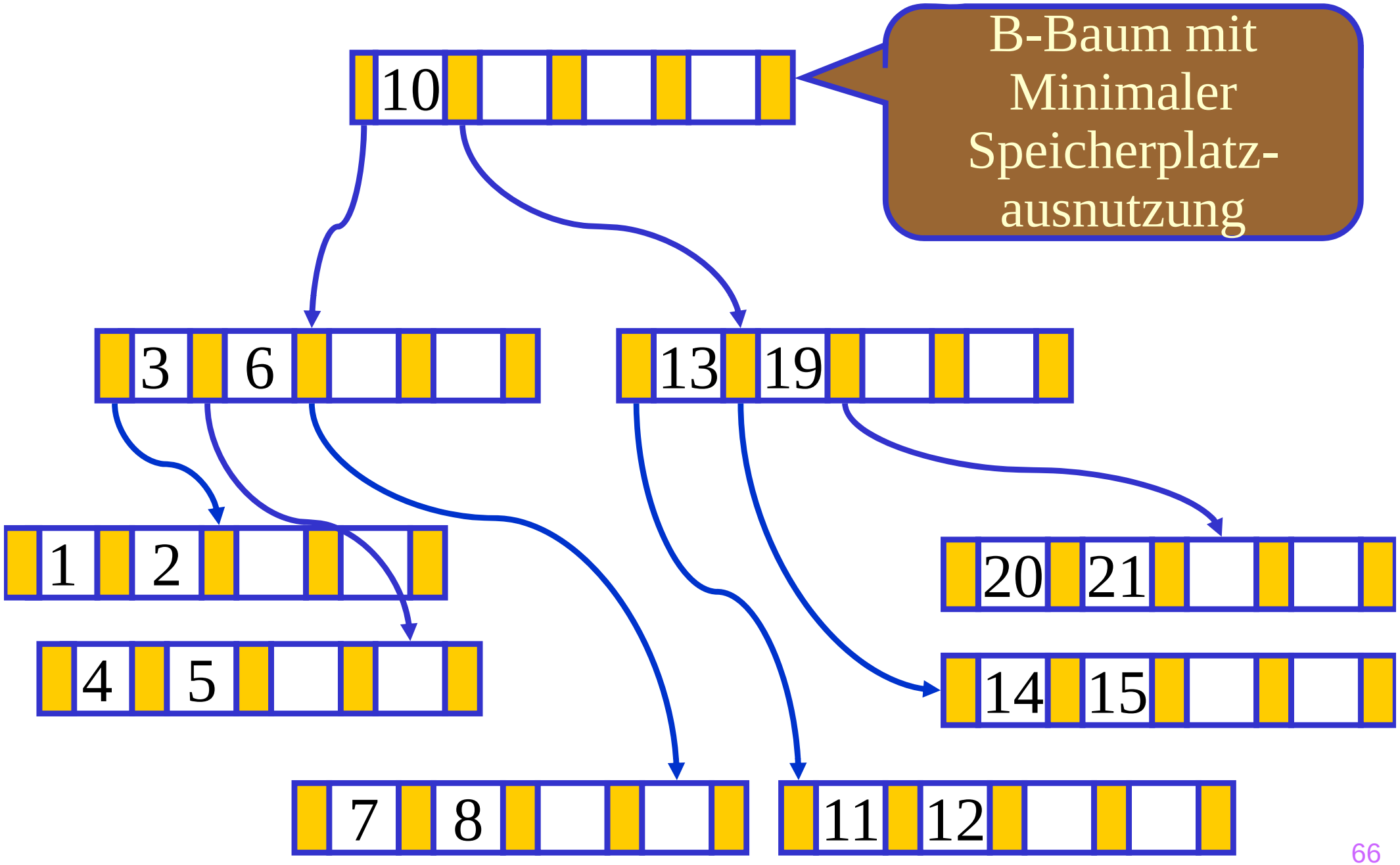
Beispiel 2



Beispiel 2



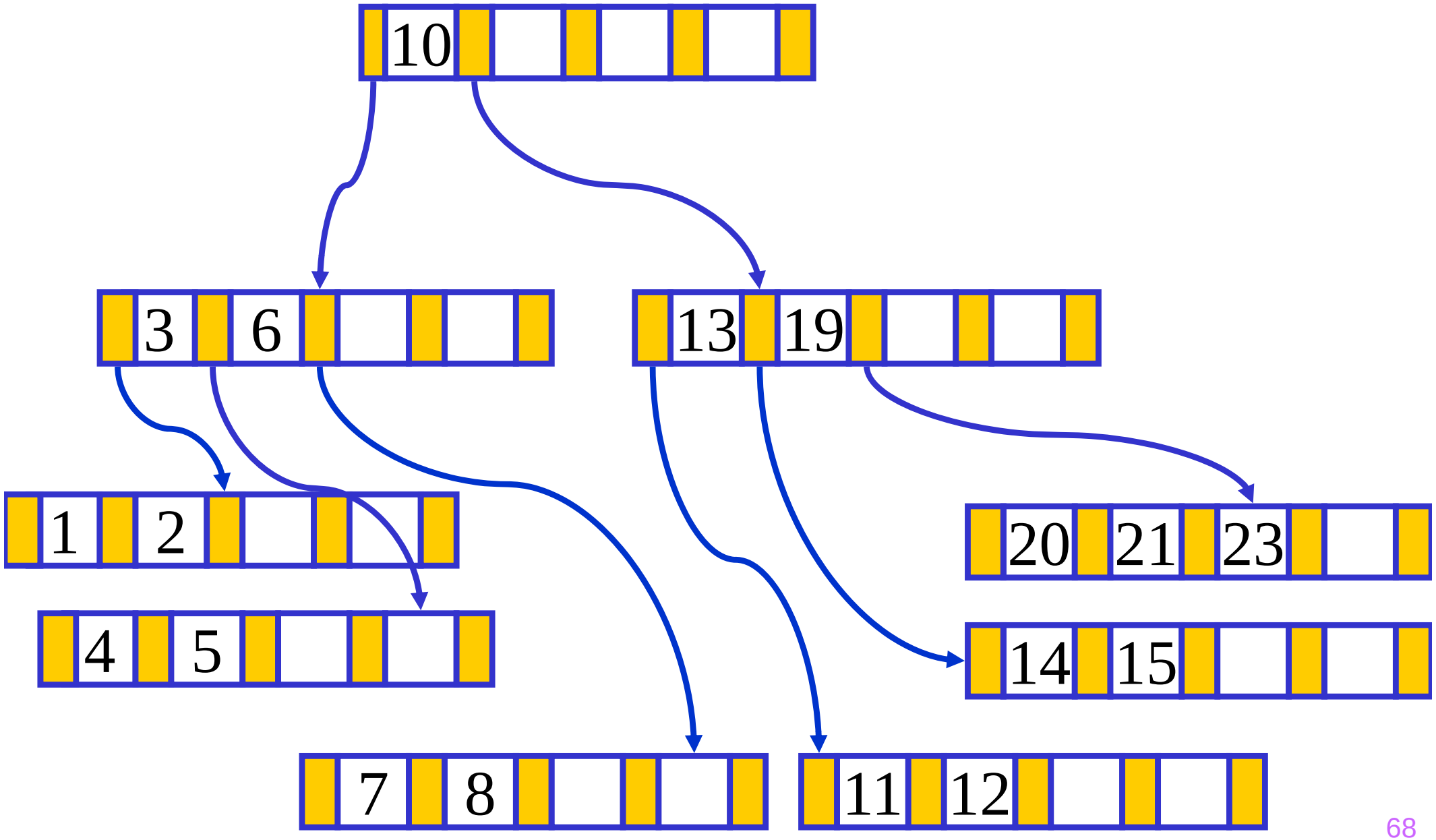
Beispiel 2


10
B-Baum mit Minimaler Speicherplatz-ausnutzung
3
6
13
19
1
2
4
5
7
8
11
12
20
21
14
15

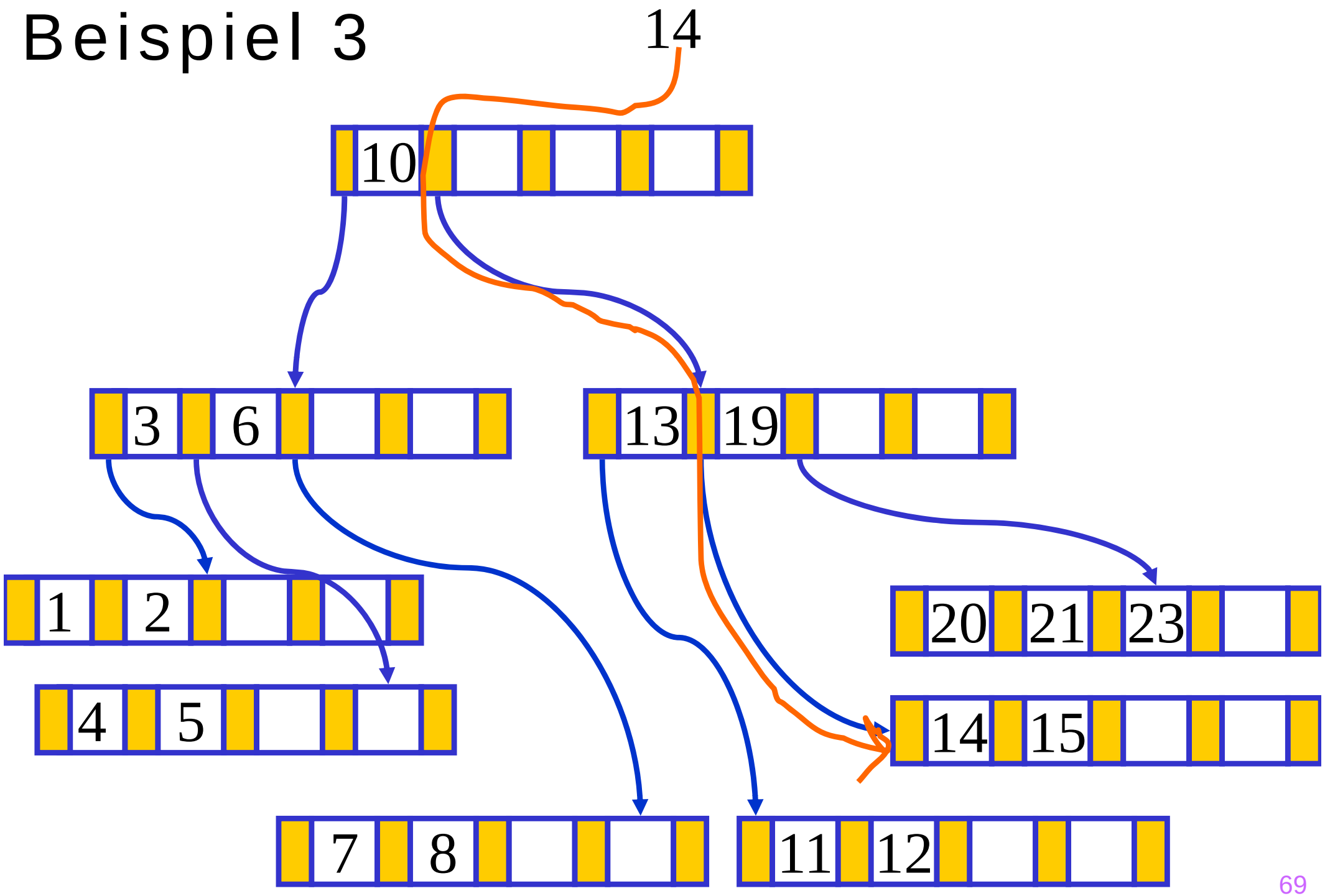
Beispiel 3

- B-Baum vom Grad k = 2
- Ausgangssituation: Einige Knoten haben minimalen Füllstand
- Löschen von Schlüsselwerten führt zu **Unterlauf**
  $\Rightarrow$ **Ausgleich mit einem Nachbarknoten** erforderlich.

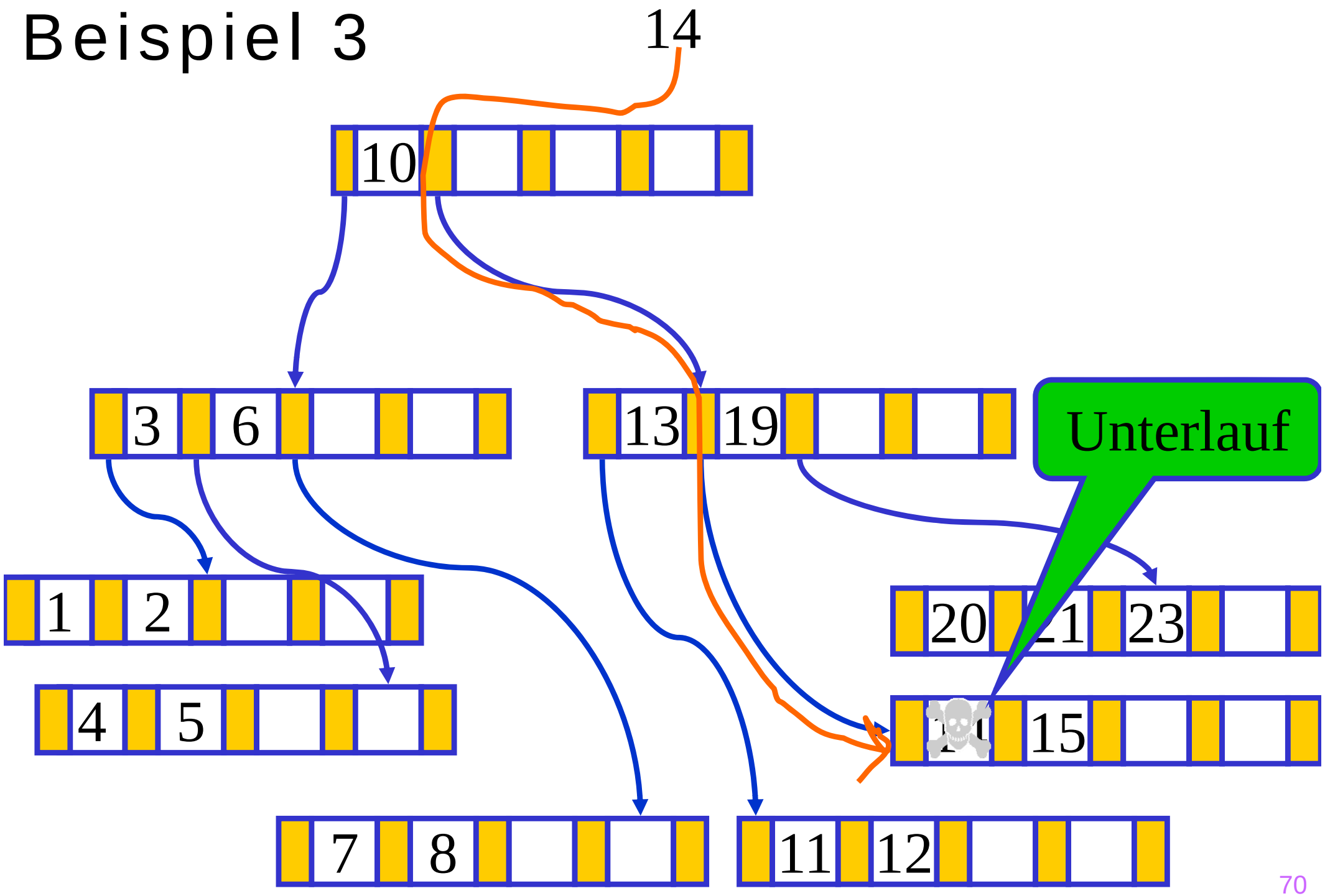
Beispiel 3



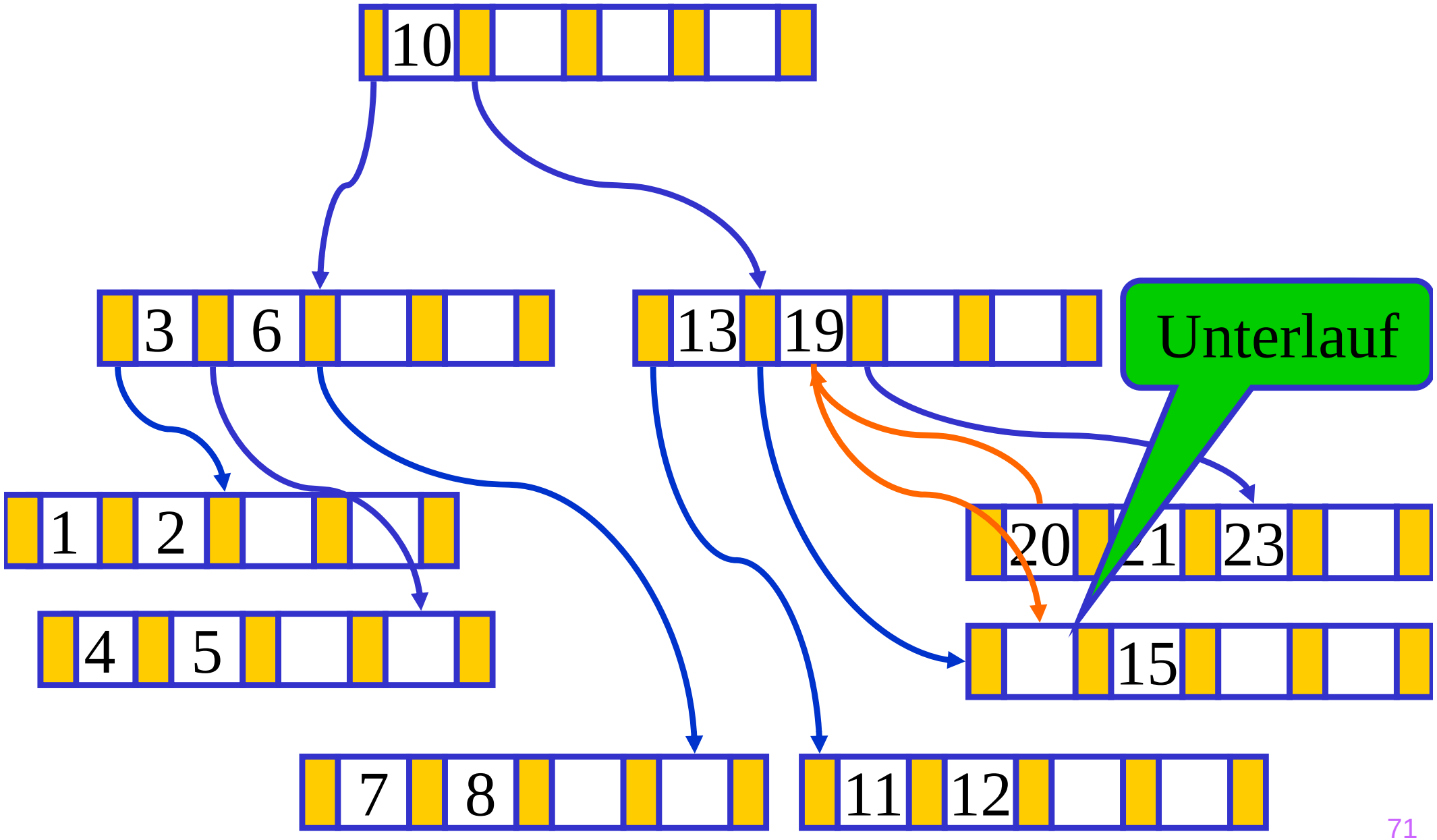
Beispiel 3



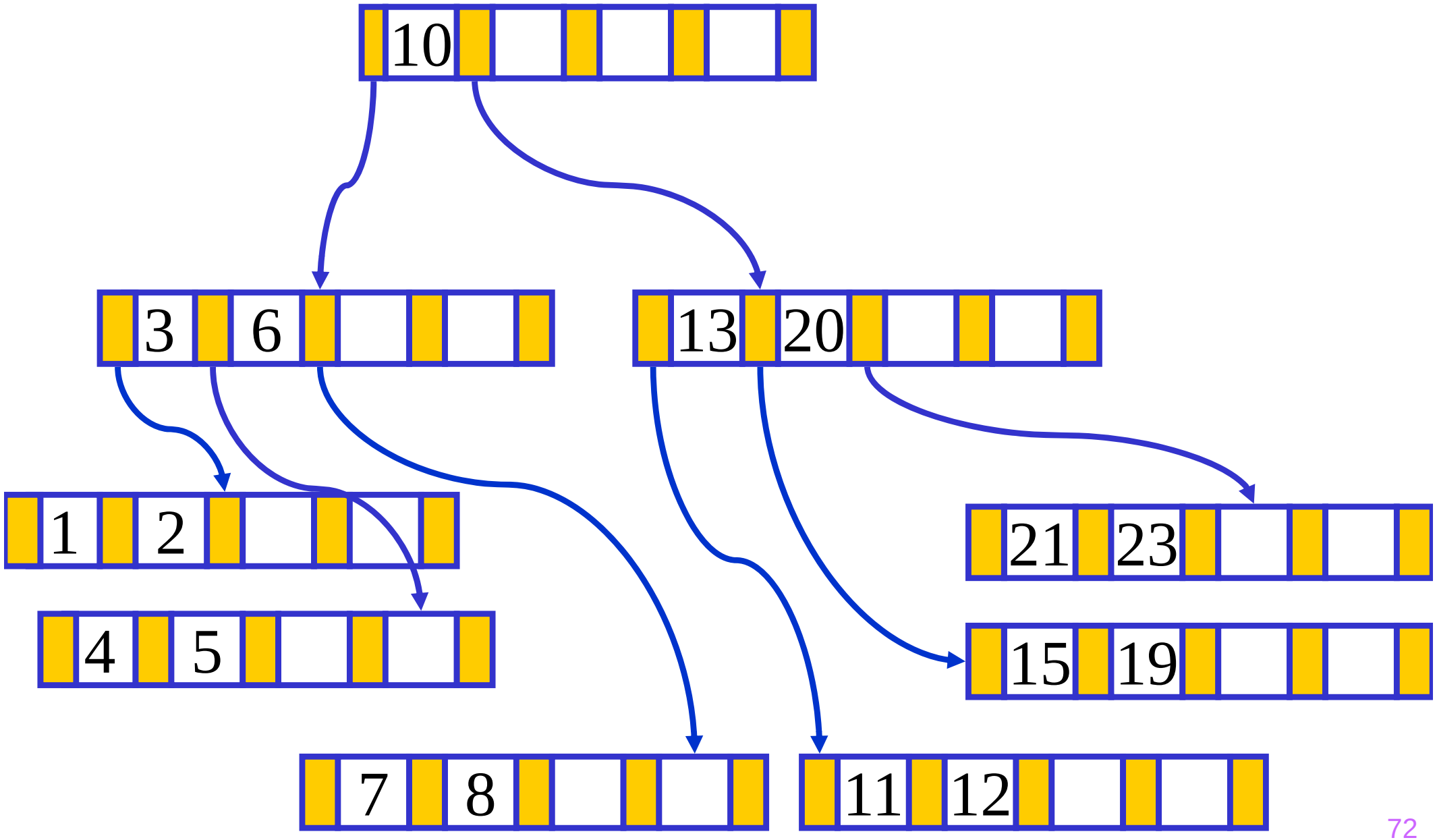
Beispiel 3



Beispiel 3



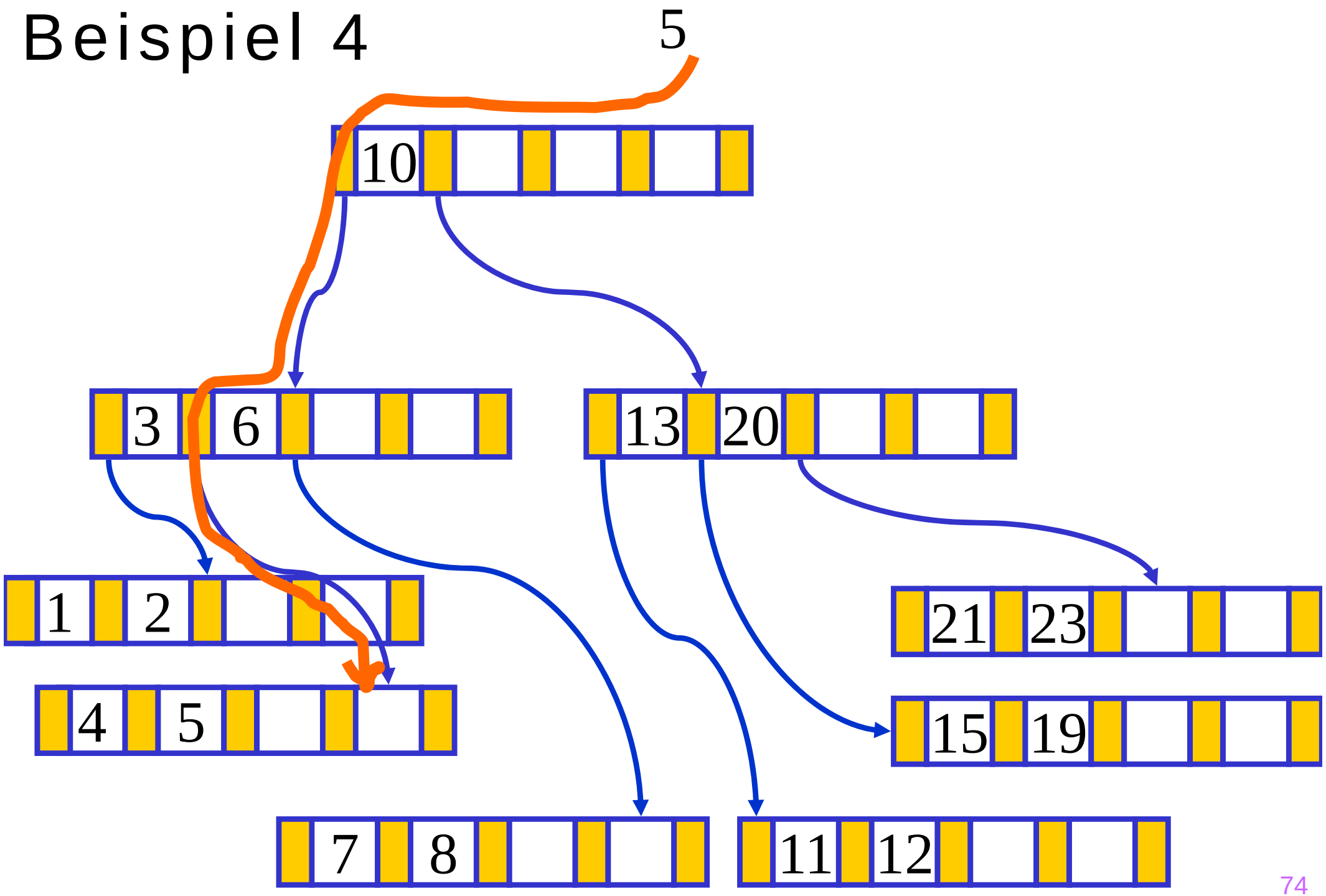
Beispiel 3



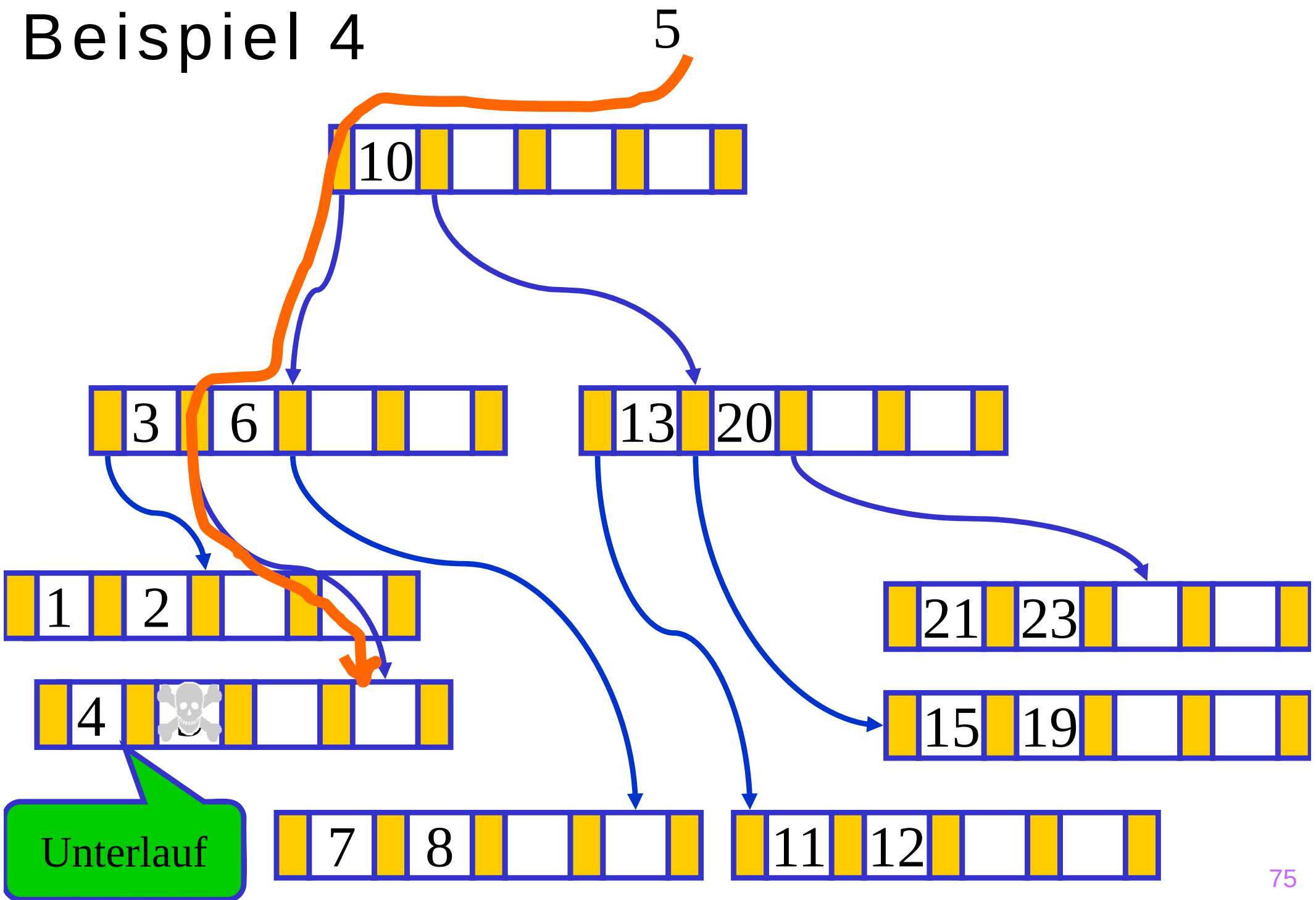
Beispiel 4

- B-Baum vom Grad k = 2
- Ausgangssituation:
  - Baum der Tiefe 3
  - Einige Knoten haben minimalen Füllstand
- **Löschen** von Schlüsselwerten führt zu Unterlauf
- Da die Nachbarknoten ebenfalls minimalen Füllstand haben, müssen **2 Knoten verschmolzen** werden.
- Durch das Verschmelzen kann es zu einem **Unterlauf des Vaterknotens** kommen. $\Rightarrow$ Vorgang wird wiederholt.
- Resultat: Baum der Tiefe 2

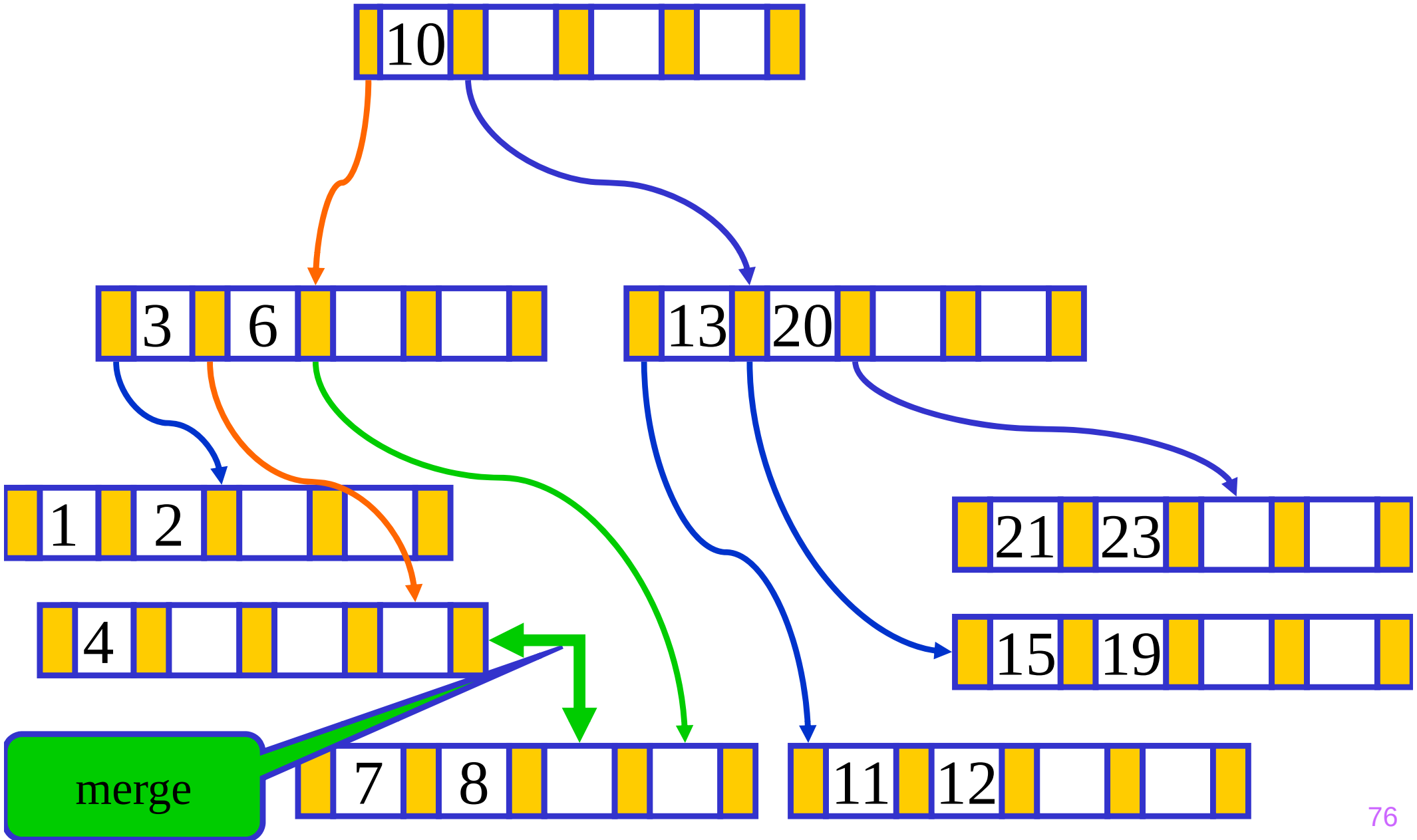
Beispiel 4



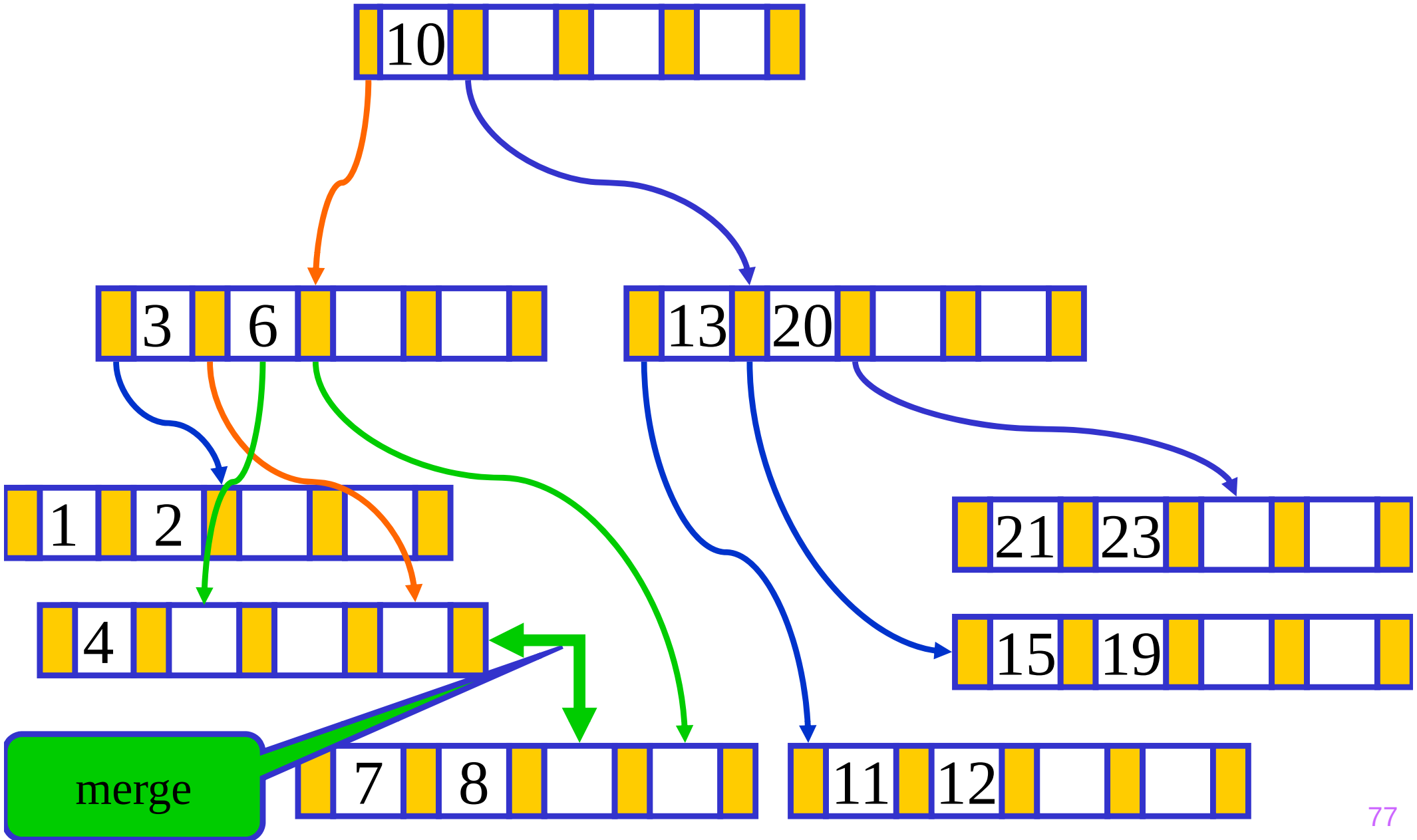
Beispiel 4



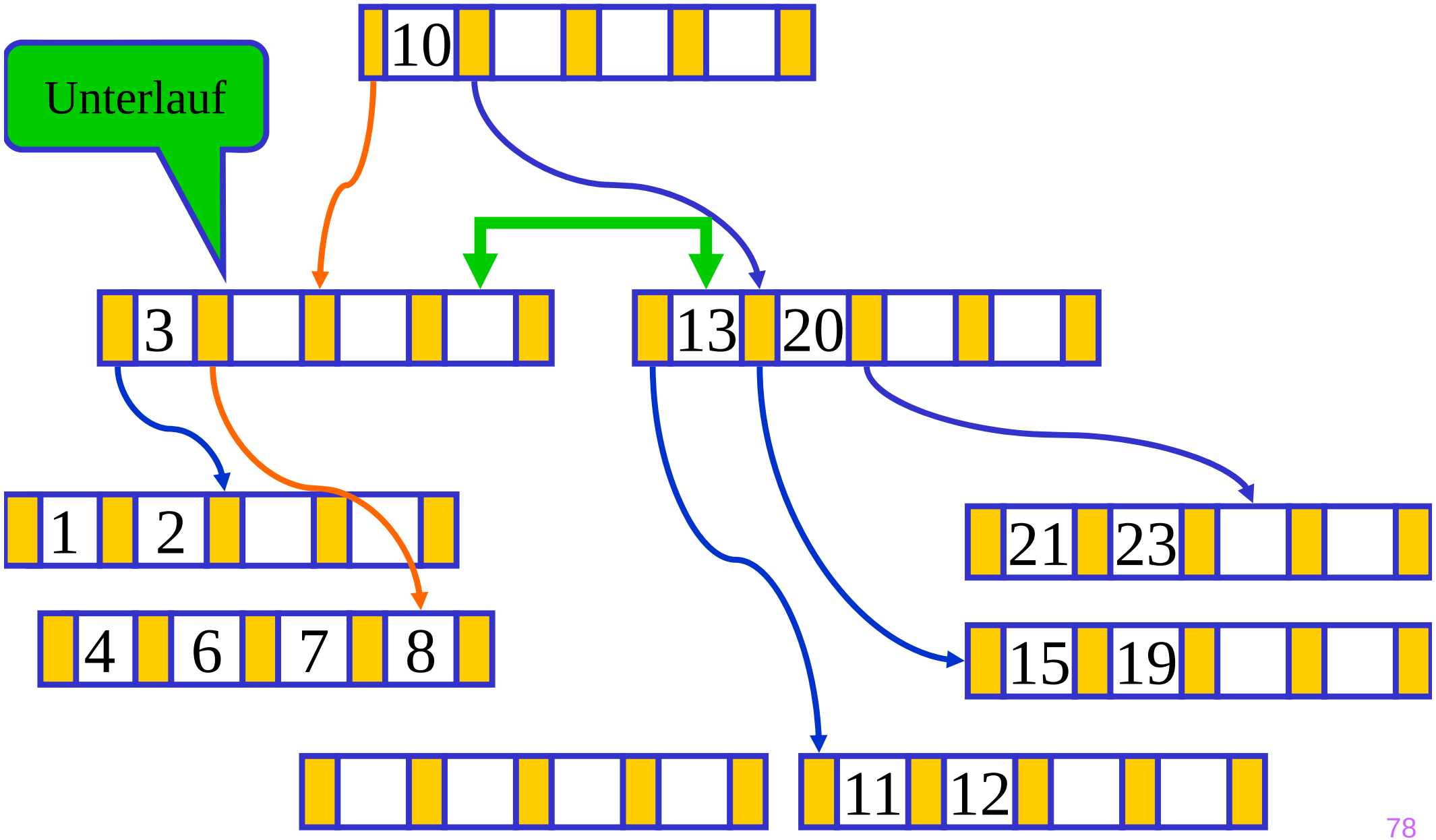
Beispiel 4



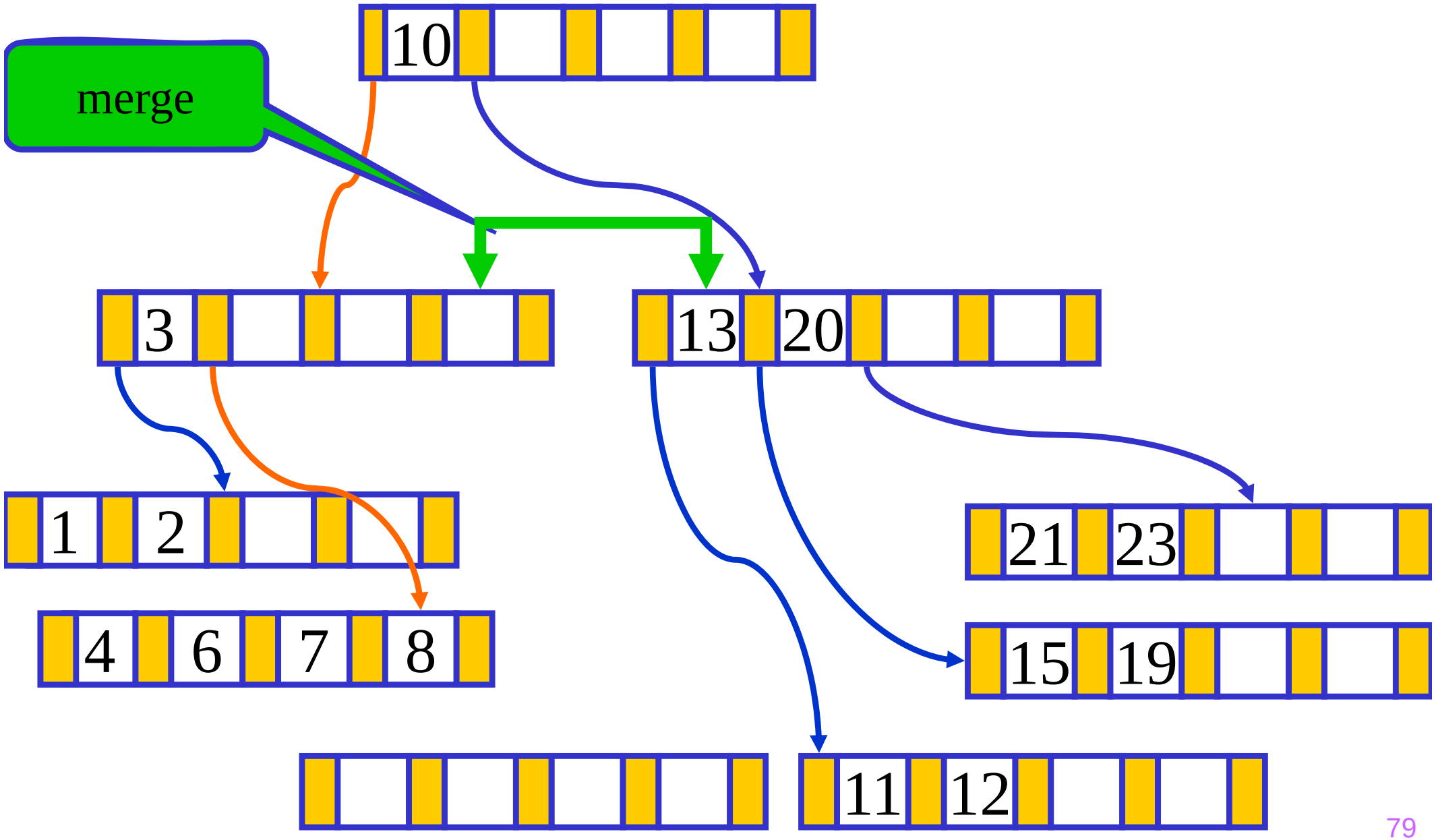
Beispiel 4



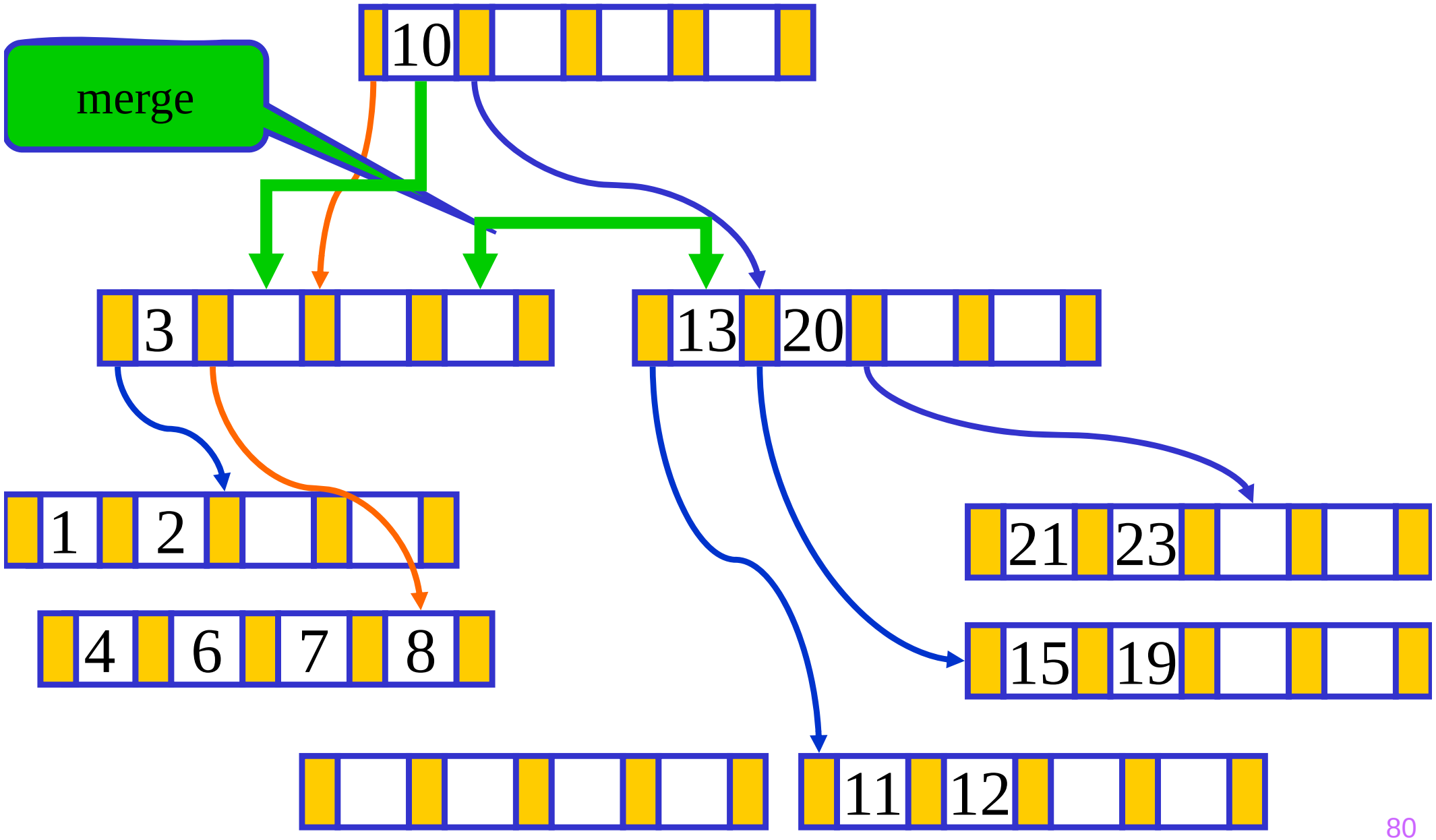
Beispiel 4



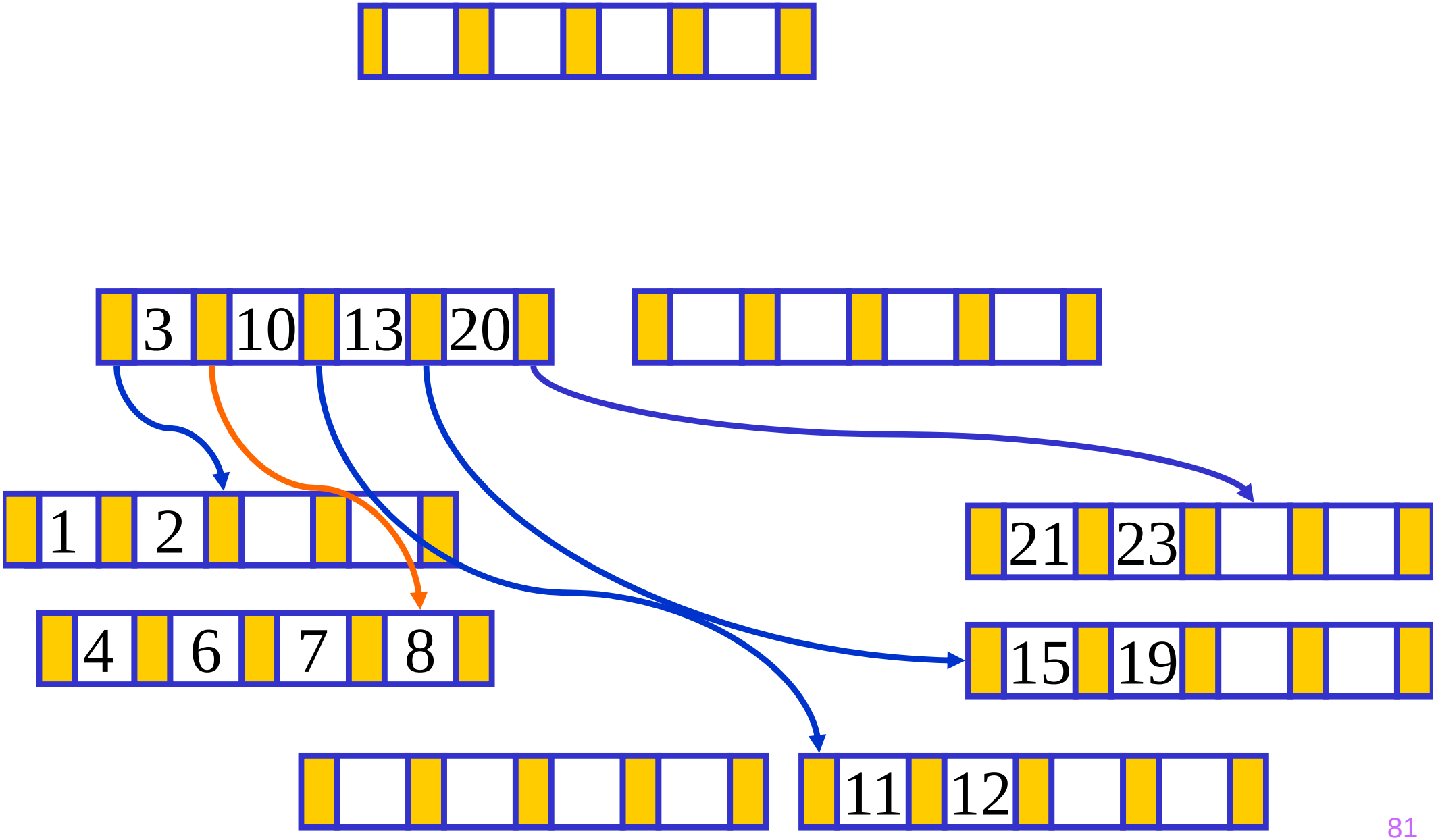
Beispiel 4



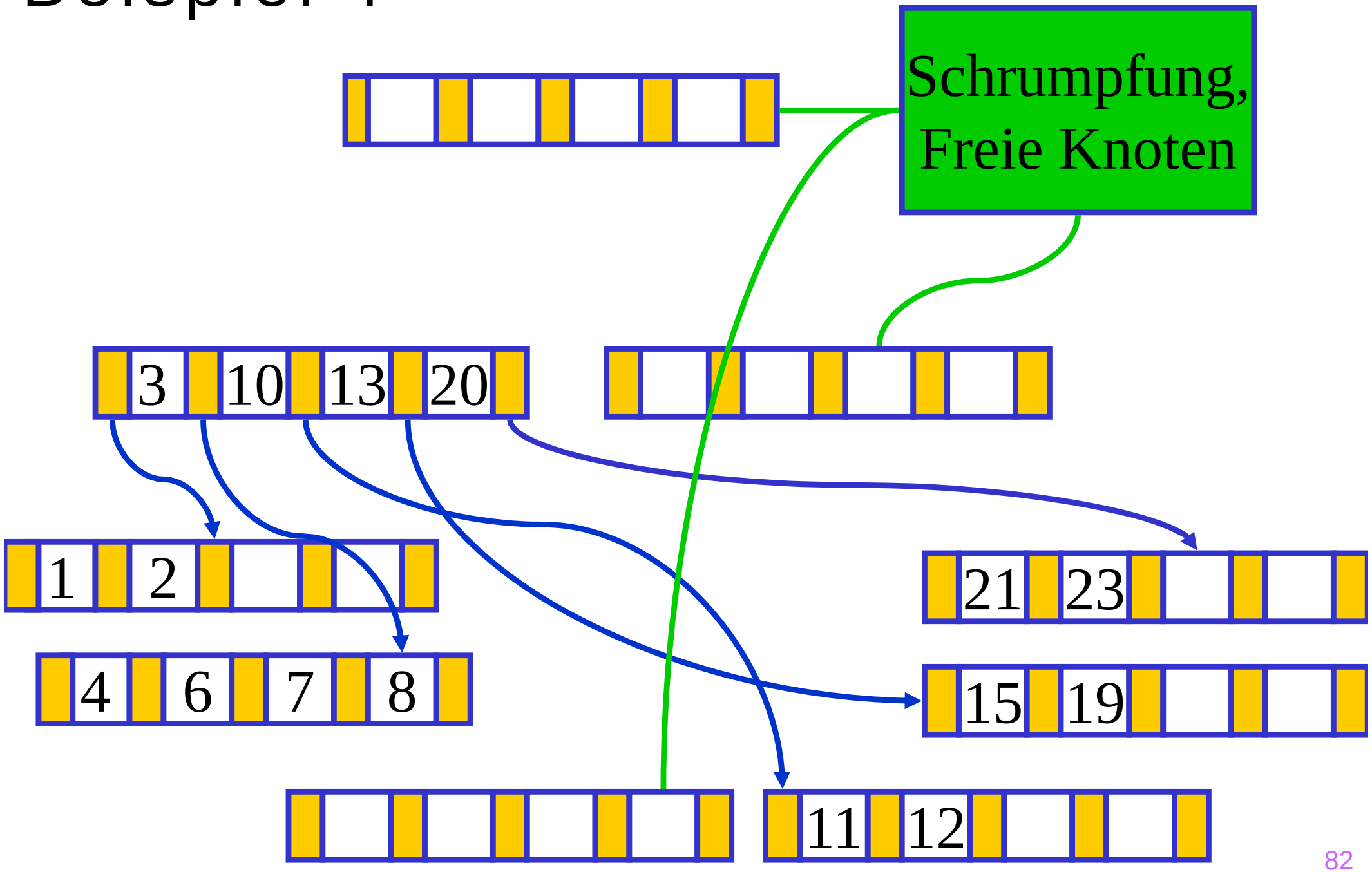
Beispiel 4



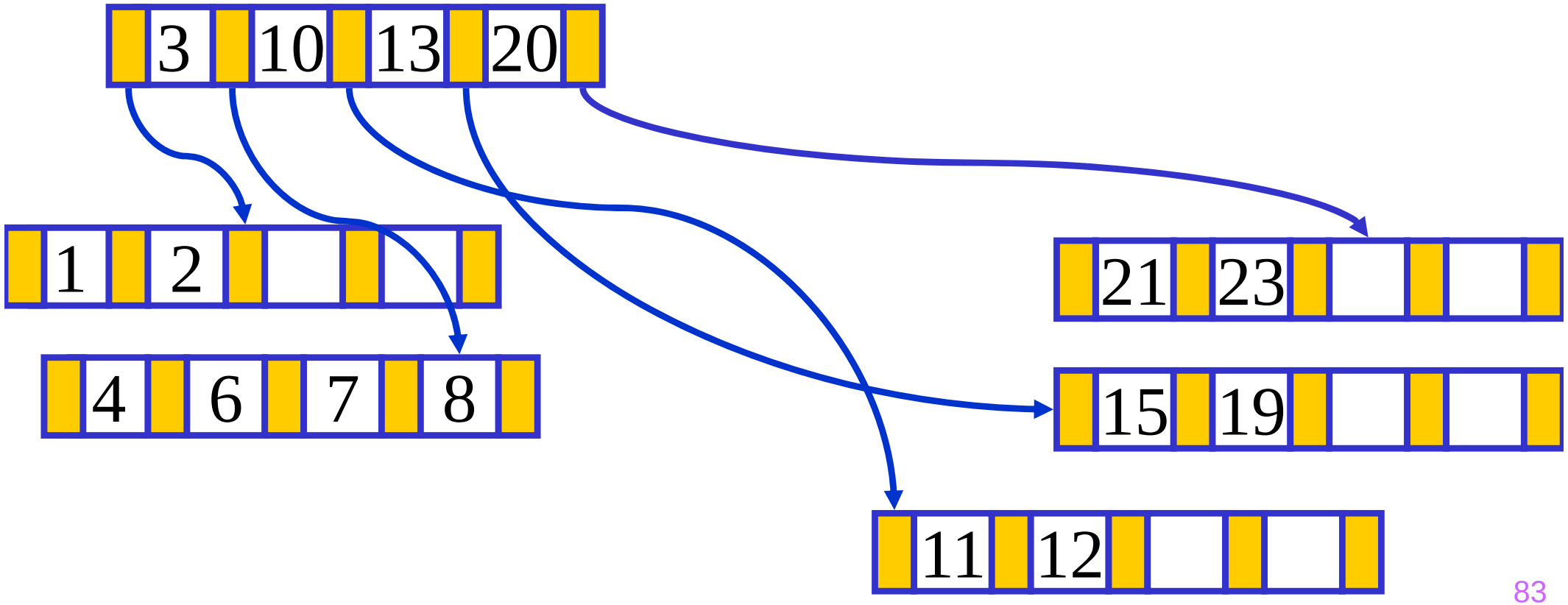
Beispiel 4



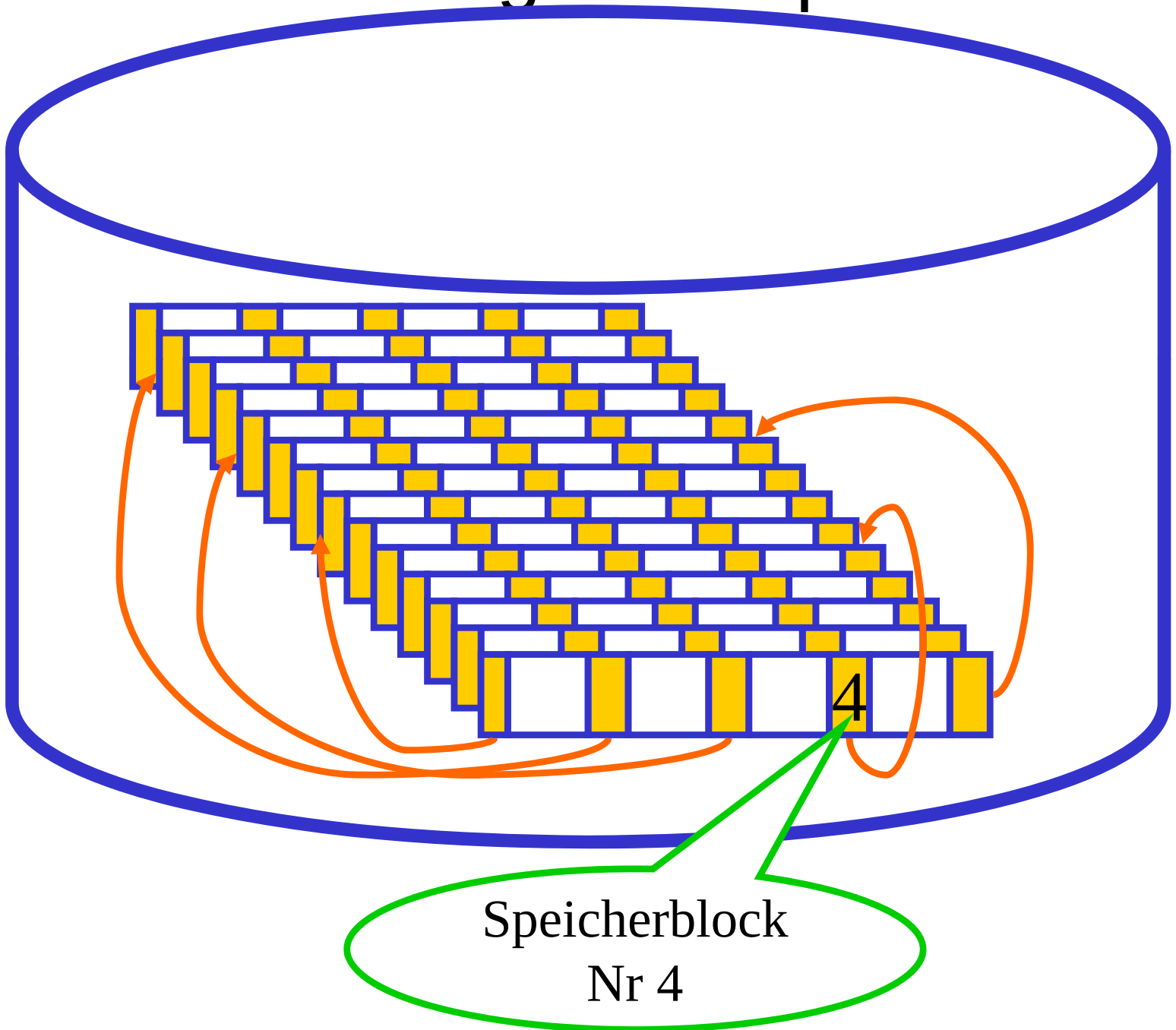
Beispiel 4



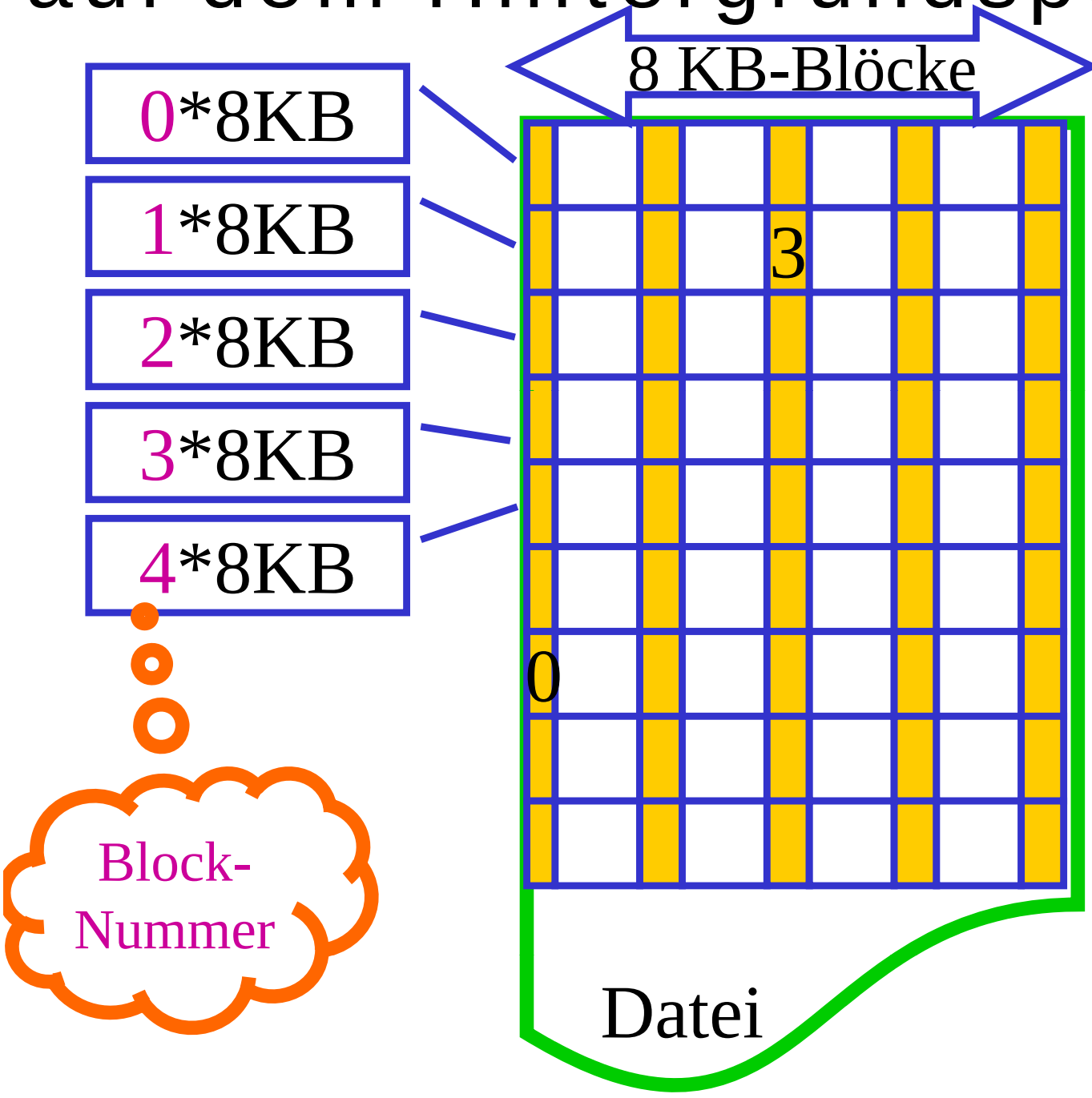
Beispiel 4



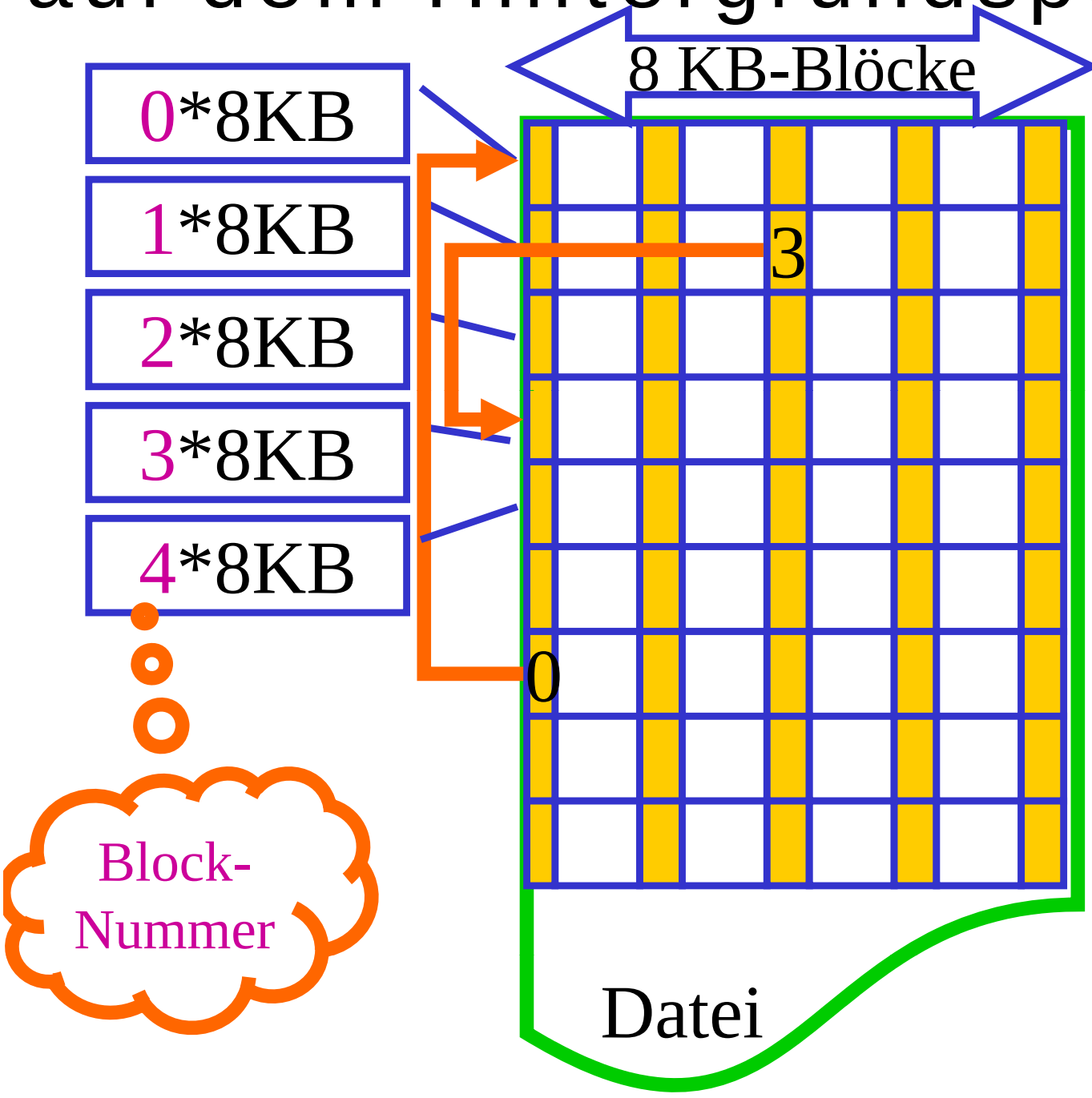
Speicherstruktur eines B-Baums auf dem Hintergrundspeicher



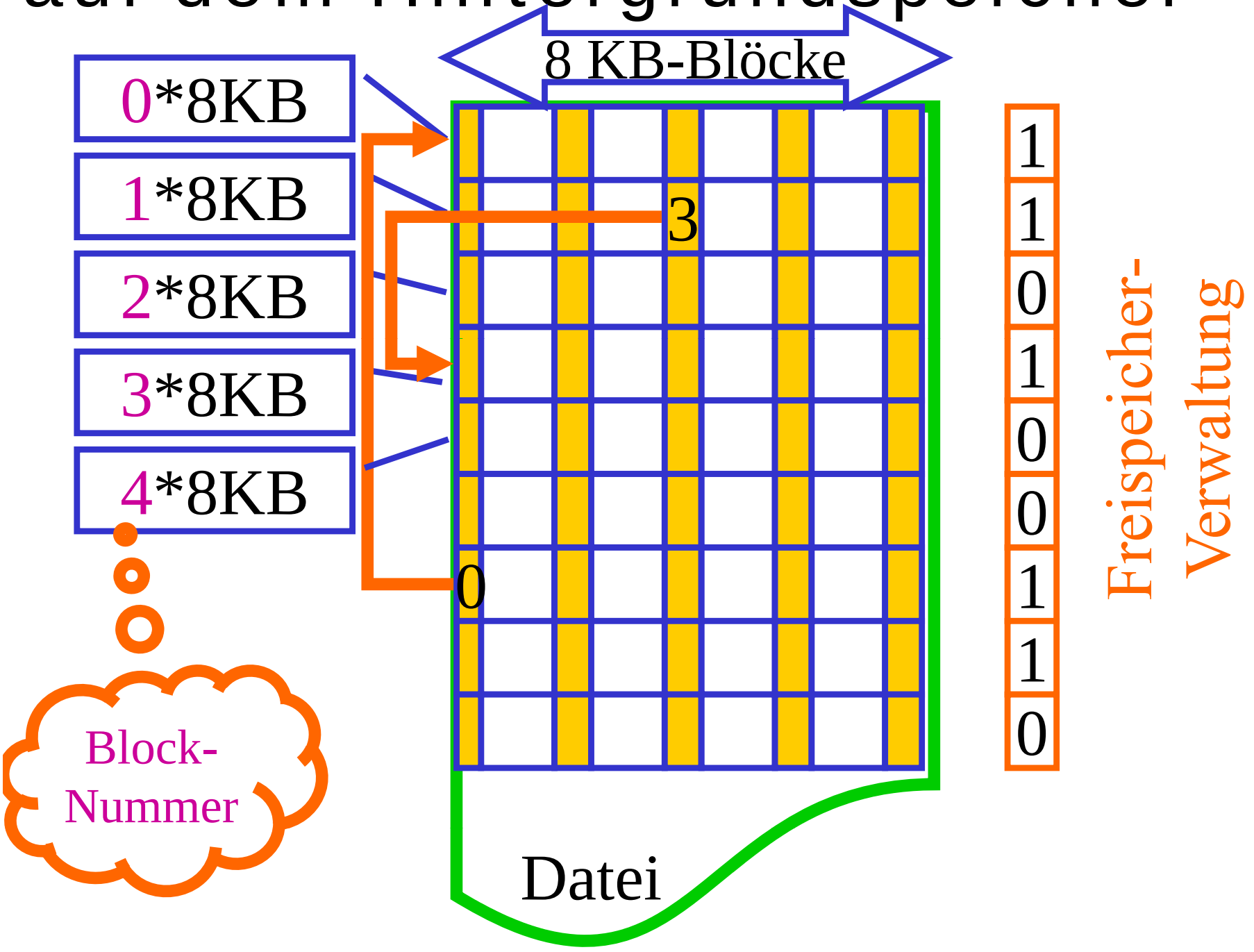
Speicherstruktur eines B-Baums auf dem Hintergrundspeicher



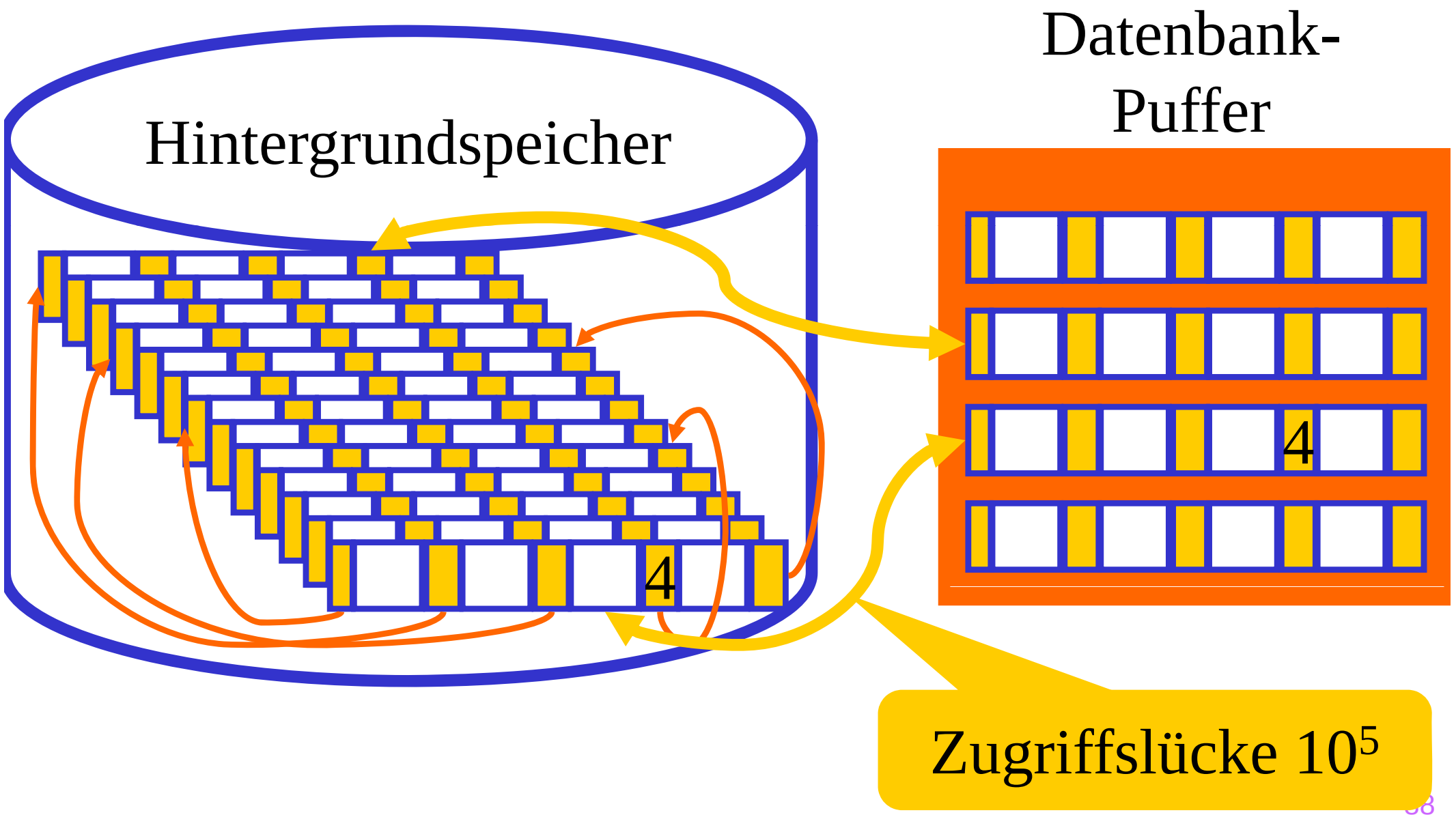
Speicherstruktur eines B-Baums auf dem Hintergrundspeicher



Speicherstruktur eines B-Baums auf dem Hintergrundspeicher



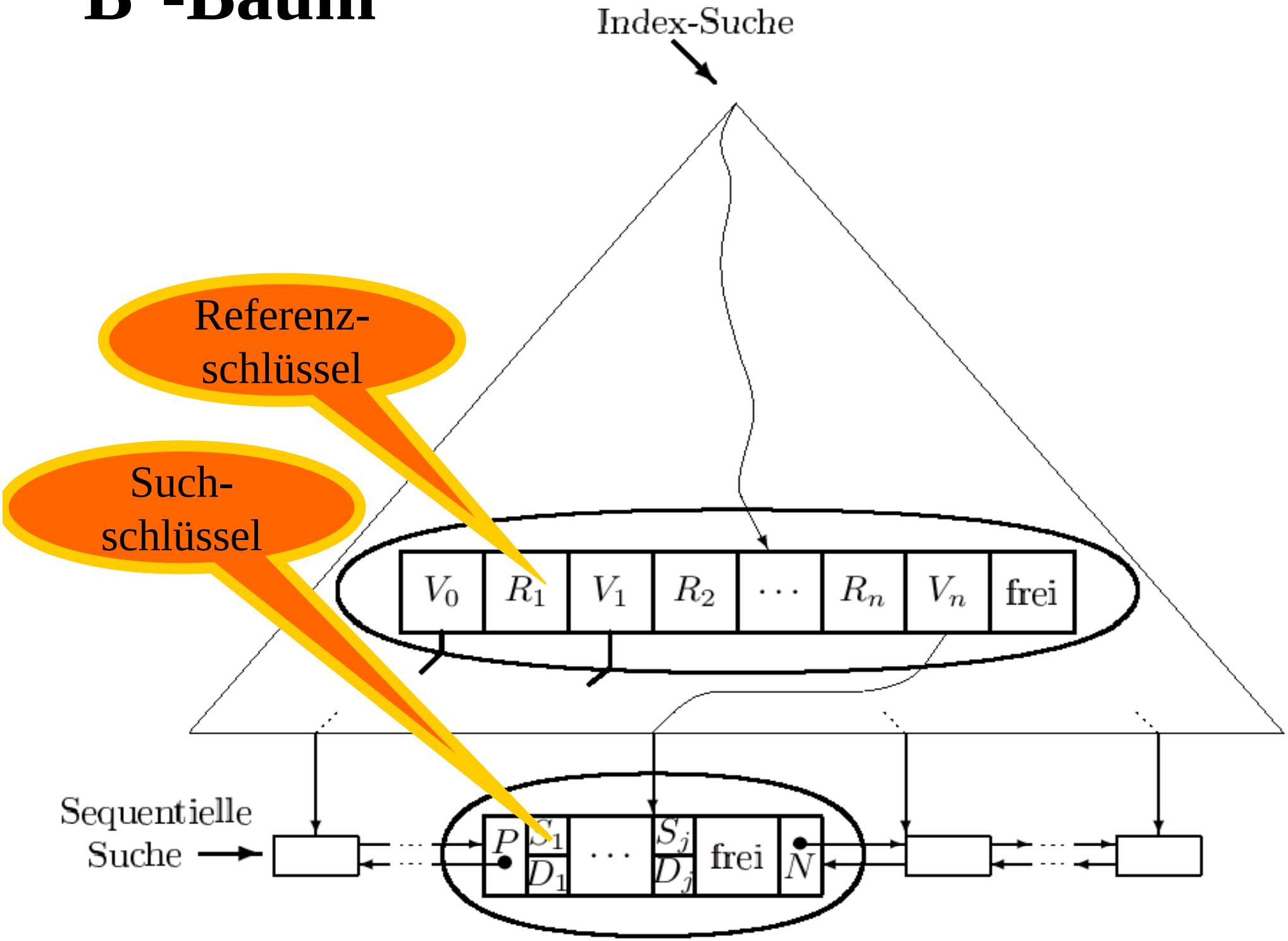
Zusammenspiel:
Hintergrundspeicher -- Hauptspeicher



B^+-Bäume

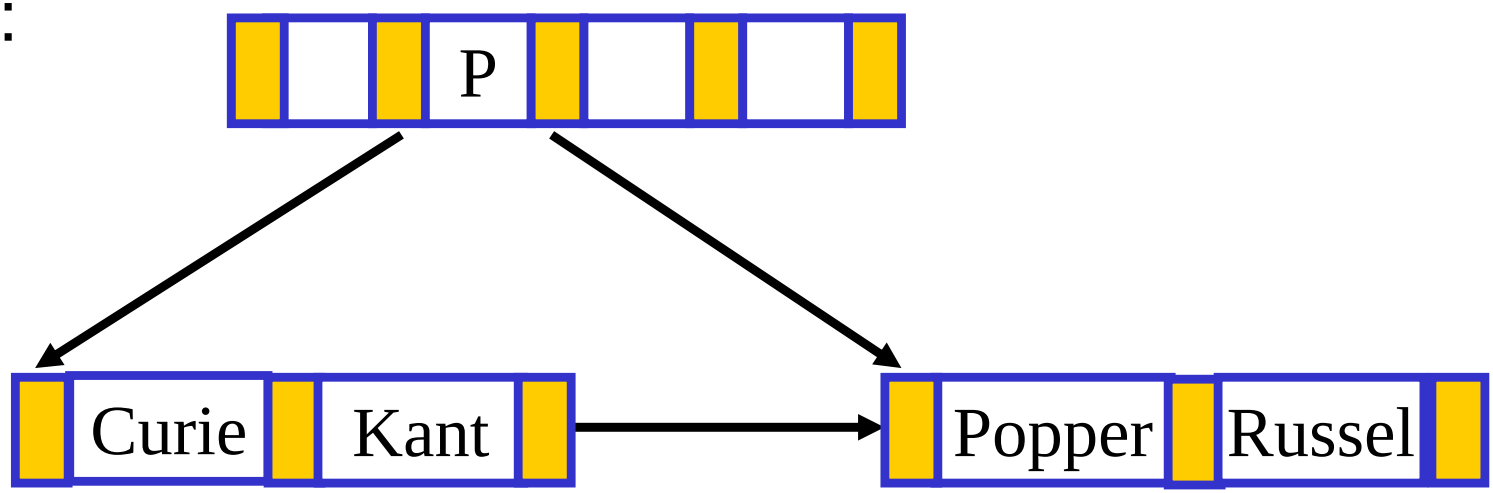
- Ziel bei B-Bäumen: geringe Tiefe. Dafür benötigt man hohen Verzweigungsgrad (d.h.: viele Kinder pro Knoten)
- Verzweigungsgrad wird bestimmt durch die Anzahl der Datensätze in einem Knoten
- Idee von B^+-Bäumen:
  - Daten werden ausschließlich in den Blättern gespeichert
  - Die inneren Knoten dienen nur noch der Navigation, d.h.: Sie enthalten Referenzschlüssel und Verweise (aber keine weiteren Daten).
  - Da Referenzschlüssel im allgemeinen weniger Platz brauchen als vollst. Datensätze, erhöht sich der Verzweigungsgrad.
  - Zusätzlicher Vorteil: Durch Verkettung der Blattknoten lassen sich Bereichsanfragen effizient beantworten.

B$^+$-Baum


Index-Suche
Referenz-
schlüssel
Such-
schlüssel
V_0 R_1 V_1 R_2 $\cdots$ R_n V_n frei
Sequentielle
Suche
P S_1 D_1 $\cdots$ S_j D_j frei N

Präfix B$^+$-Bäume

- Idee:
  - An den inneren Knoten werden nur Referenzschlüssel benötigt.
  - Die Werte der Referenzschlüssel dienen nur der Navigation und müssen nicht unbedingt in den Daten existieren.
- Beispiel:



- Anforderung an Referenzschlüssel R: Kant < $R \leq$ Popper

  Bemerkung: Kleiner Fehler im Buch: dort steht, dass links die Schlüssel $\leq R$ sein müssen und nicht $< R$.

Idee von Hashing

- Direkte Abbildung von Werten eines Such-Schlüssels auf einen bestimmten Speicherbereich.
- Aufteilung des Speichers in "**Buckets**".
- Jedes Bucket besteht aus einer primären Seite sowie möglicherweise 1 oder mehreren Überlaufseiten.
- Mittels **Hashfunktion** h: S $\rightarrow$ B wird jedem Schlüsselwert eindeutig eine Bucket-Nummer zugeordnet.
- Gebräuchlichste Hashfunktionen:
  - h(x) = x mod N (N = Anzahl der Buckets)
  - oder: h(x) = (a*x + b) mod N
    (für geeignete Wahl von Konstanten a und b)
- "Gute Hashfunktion": möglichst gleichmäßige Verteilung der möglichen Schlüsselwerte auf die Buckets.

Statisches Hashing

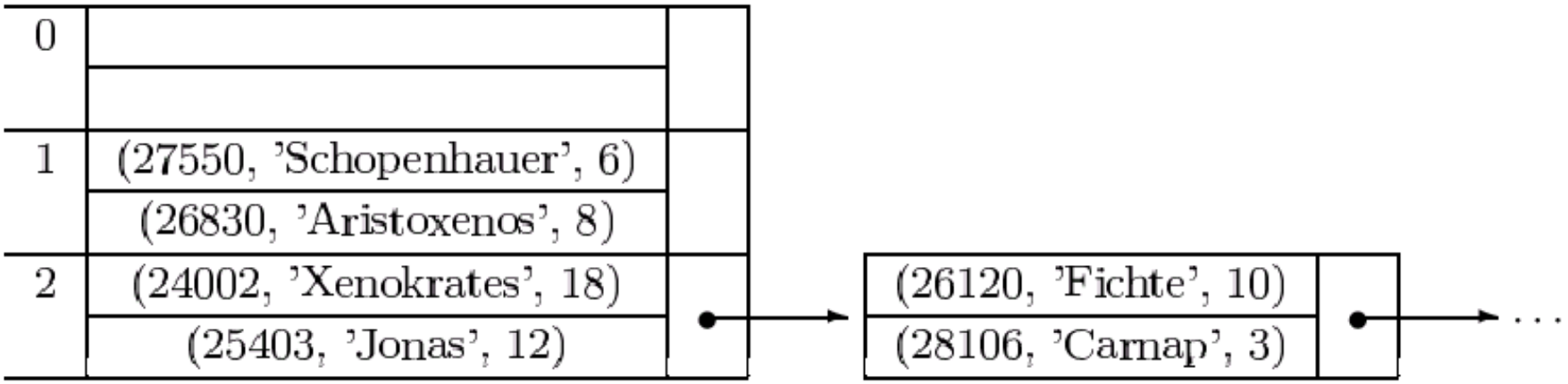
- A priori Allokation des Speichers, d.h.: jedes Bucket besteht exakt aus 1 Seite (= primäre Seite).
- Bei Überlauf einer Seite wird eine weitere Seite zum Bucket hinzugefügt (= Überlaufseite).
- Mit der Zeit können sehr viele Überlaufseiten entstehen
  $\Rightarrow$ Suchen innerhalb eines Buckets wird teuer.
- Lösung 1: Nachträgliche Vergrößerung der Hashtabelle
  - Rehashing der Einträge
  - Hashfunktion h(...) = ... mod N wird ersetzt durch h(...) = ... mod M (mit M > N)
  - In Datenbankanwendungen: viele GB $\Rightarrow$ sehr teuer
- Lösung 2: Erweiterbares Hashing

Statisches Hashing

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

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

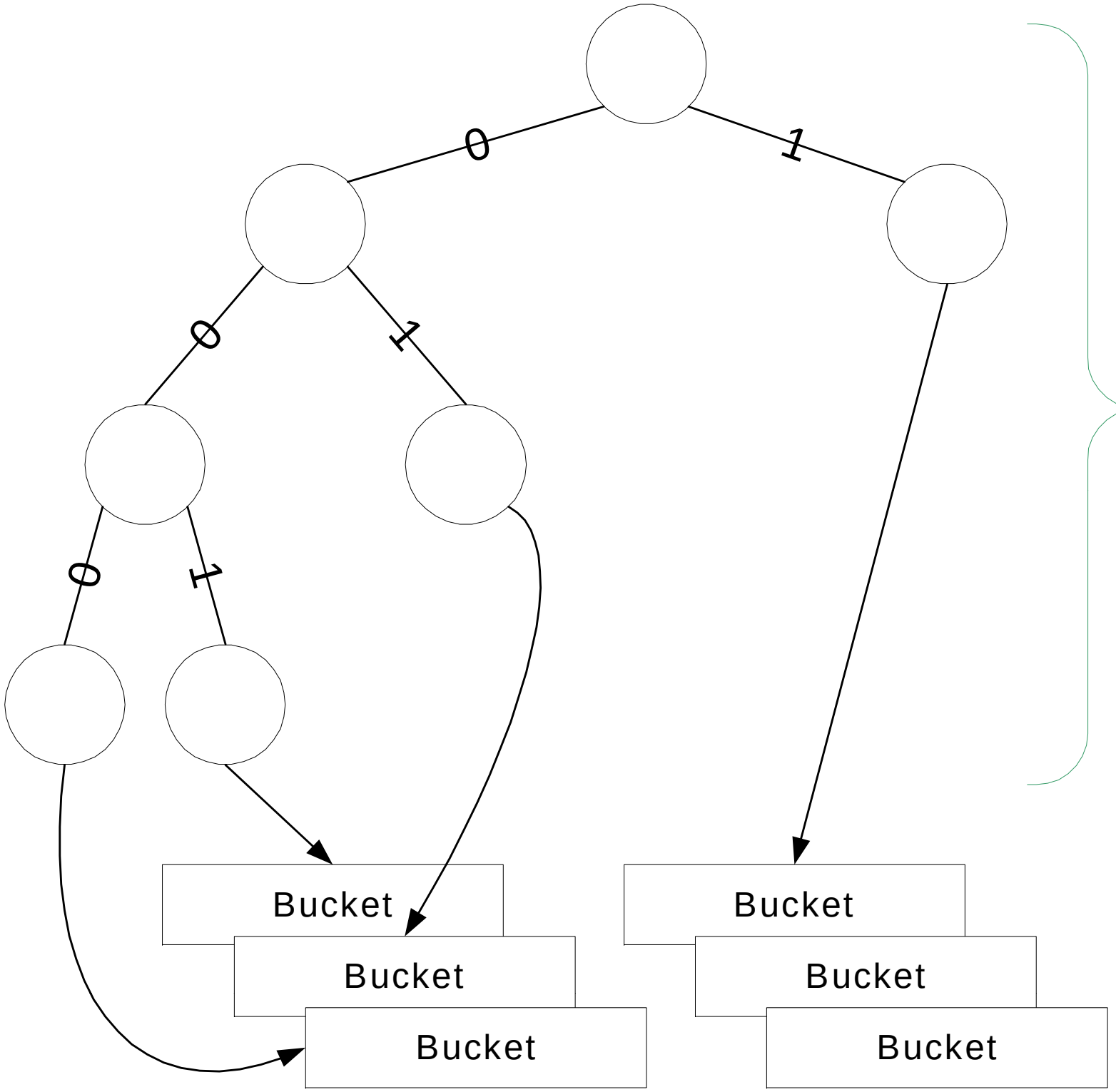
- Kollisionsbehandlung



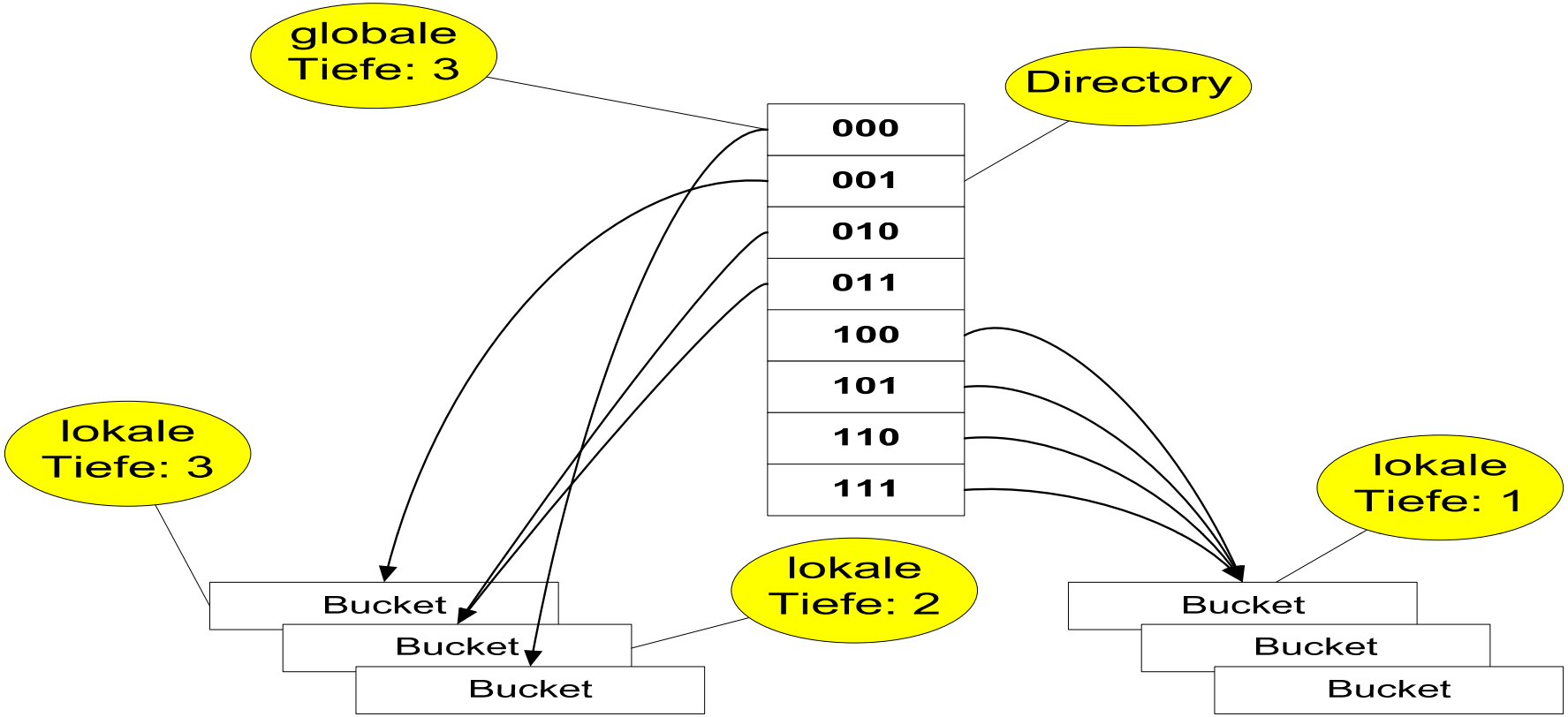
$\Longrightarrow$ ineffizient bei nicht vorhersehbarer Datenmenge

Erweiterbares Hashing

- zusätzliche Indirektion über ein Directory
- Directory enthält einen Zeiger (Seiten-Nr) des Hash-Bucket
- Dynamisches Wachsen und Schrumpfen ist möglich (ohne Überlaufseiten).
- Der Zugriff auf das Directory erfolgt über einen binären Hashcode (d.h.: Strings aus 0 und 1).
- Der Zugriff auf die Buckets erfolgt im Stil eines (i.a. nicht ausbalancierten) binären Entscheidungsbaums (= "Trie"). Jeder Pfad im Trie entspricht einem String aus 0 und 1.



binärer Trie, Entschei-dungs-baum Directory


globale
Tiefe: 3
Directory
000
001
010
011
100
101
110
111
lokale
Tiefe: 3
lokale
Tiefe: 2
lokale
Tiefe: 1
Bucket
Bucket
Bucket
Bucket
Bucket
Bucket

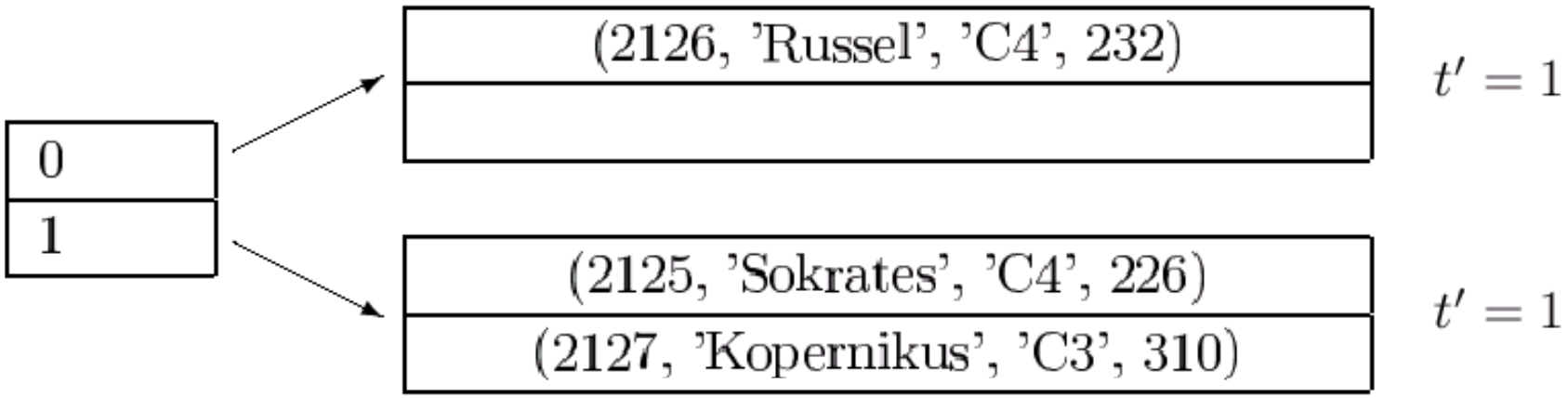
Hashfunktion für erweiterbares Hashing

- 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 Hash-Wertes (Bitstrings) benötigt.
- Wenn die Hashtabelle wächst, wird aber sukzessive ein längerer Präfix benötigt. $\Rightarrow$ Das Directory wird jeweils verdoppelt.
- **Globale Tiefe** (des Directory): Momentan verwendete Anzahl der Bits in den Hashwerten des Directory (= max. Länge der Pfade des binären Trie).
- **Lokale Tiefe** (jedes einzelnen Bucket): Länge des Pfades (im binären Trie) der auf dieses Bucket zeigt.

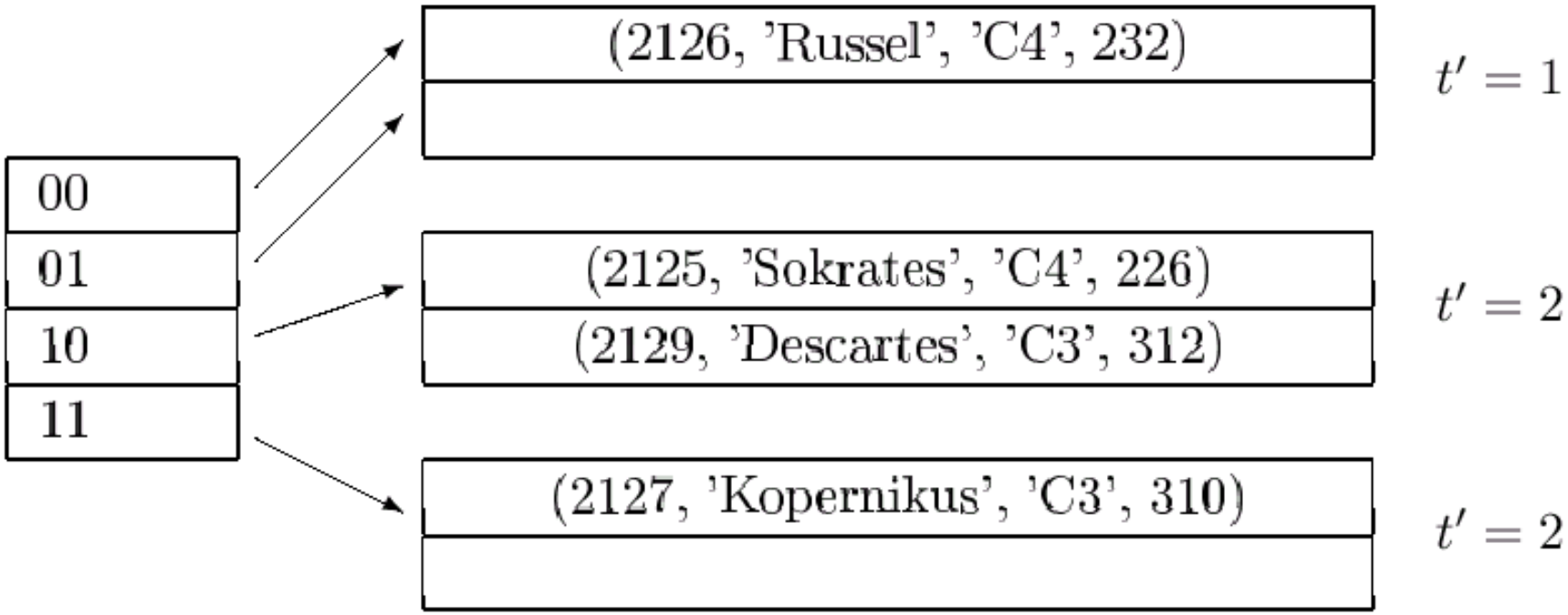
Demonstration des erweiterbaren Hashings

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

Beispiel-Hashfunktion: gespiegelte PersNr



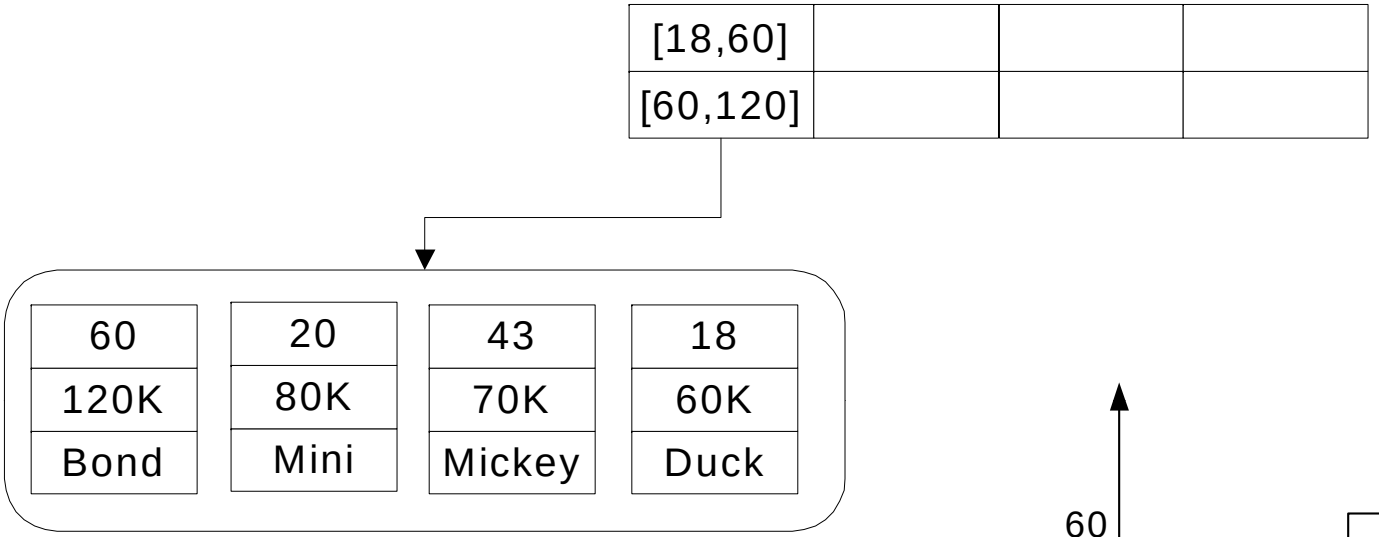
x	$h(x)$	
	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001

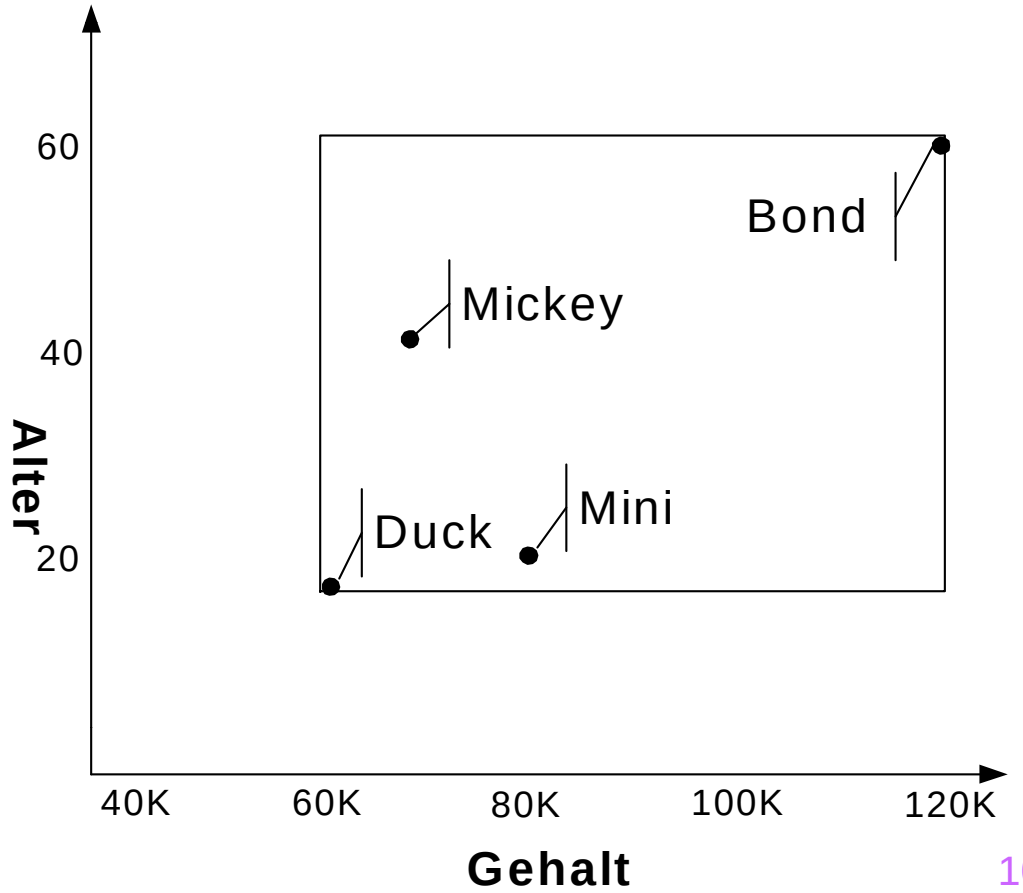


R-Bäume

- Ursprüngliche Motivation: Indexe für geometrische Daten
- Allgemeine Verwendungsmöglichkeit: Indexierung von mehreren Attributen (= Dimensionen)
- R-Baum = balancierter Baum. Die inneren Knoten dienen nur der Navigation; Daten nur an den Blättern (wie B^+-Baum).
- Einträge eines inneren Knoten:
  - Anstelle eines Referenzschlüssels enthält jeder Eintrag die Beschreibung einer n-dimensionalen "Box".
  - Verweis auf einen Nachfolger-Knoten. Alle Datenpunkte bzw. alle Boxes der Nachfolger müssen innerhalb der Box des Vorgängers liegen.

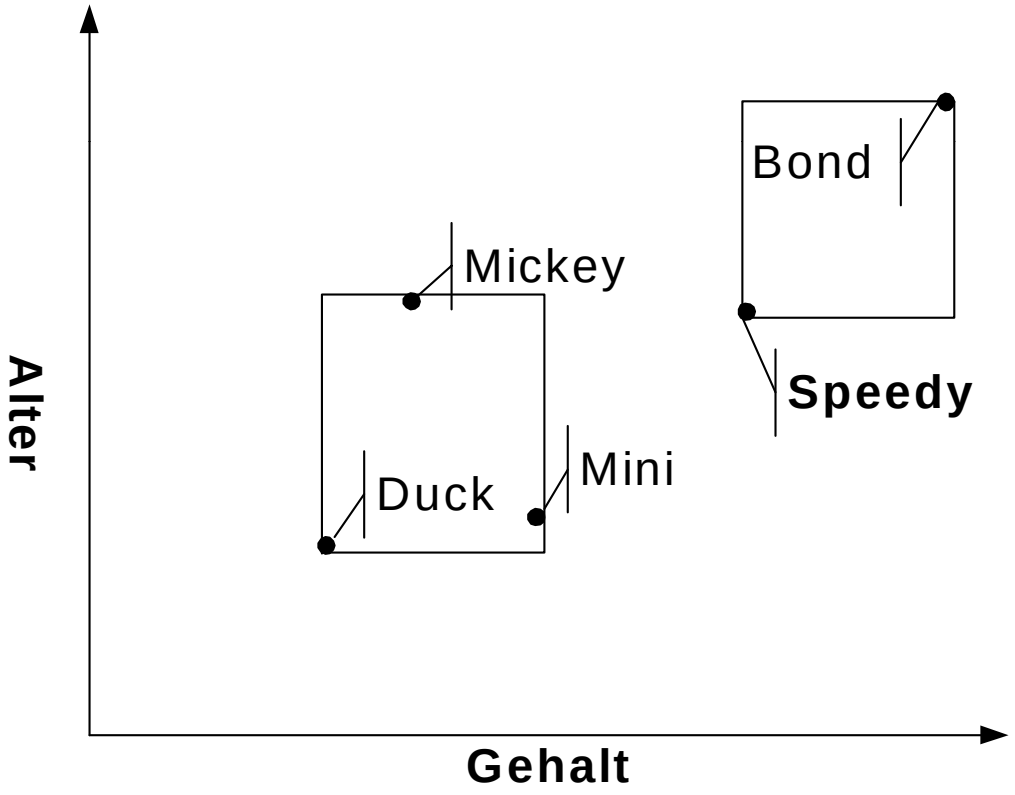
R-Baum: Urvater der baum-strukturierten mehrdimensionalen Zugriffsstrukturen



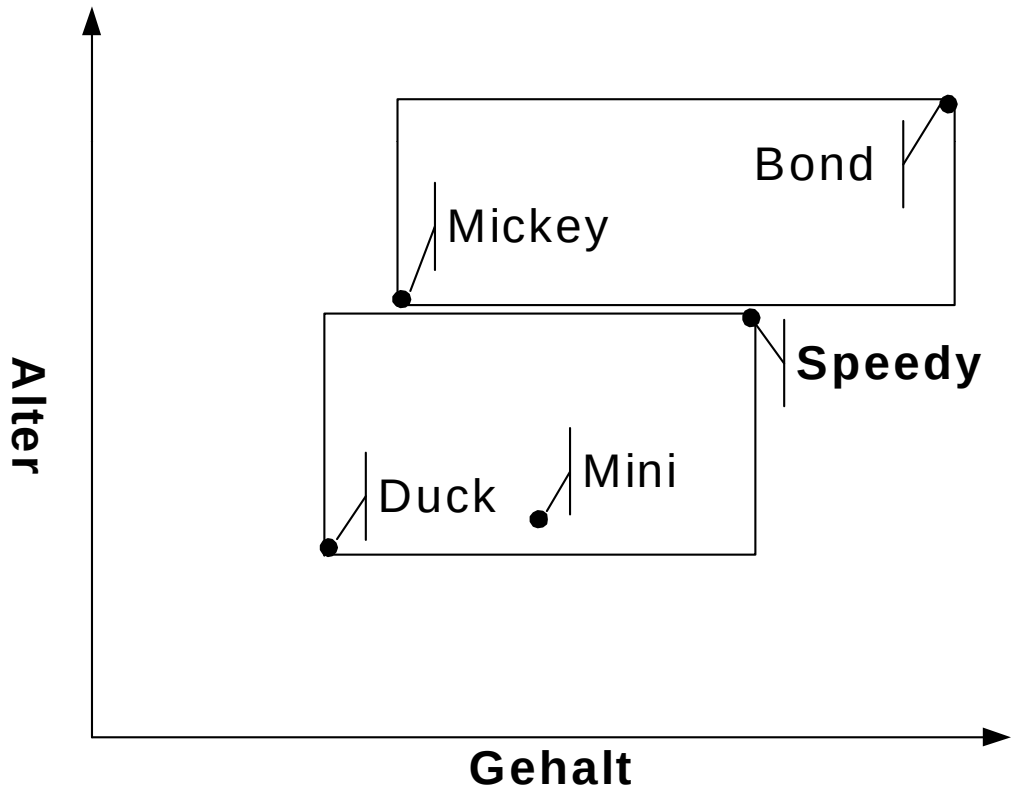


Gute vs. schlechte Partitionierung

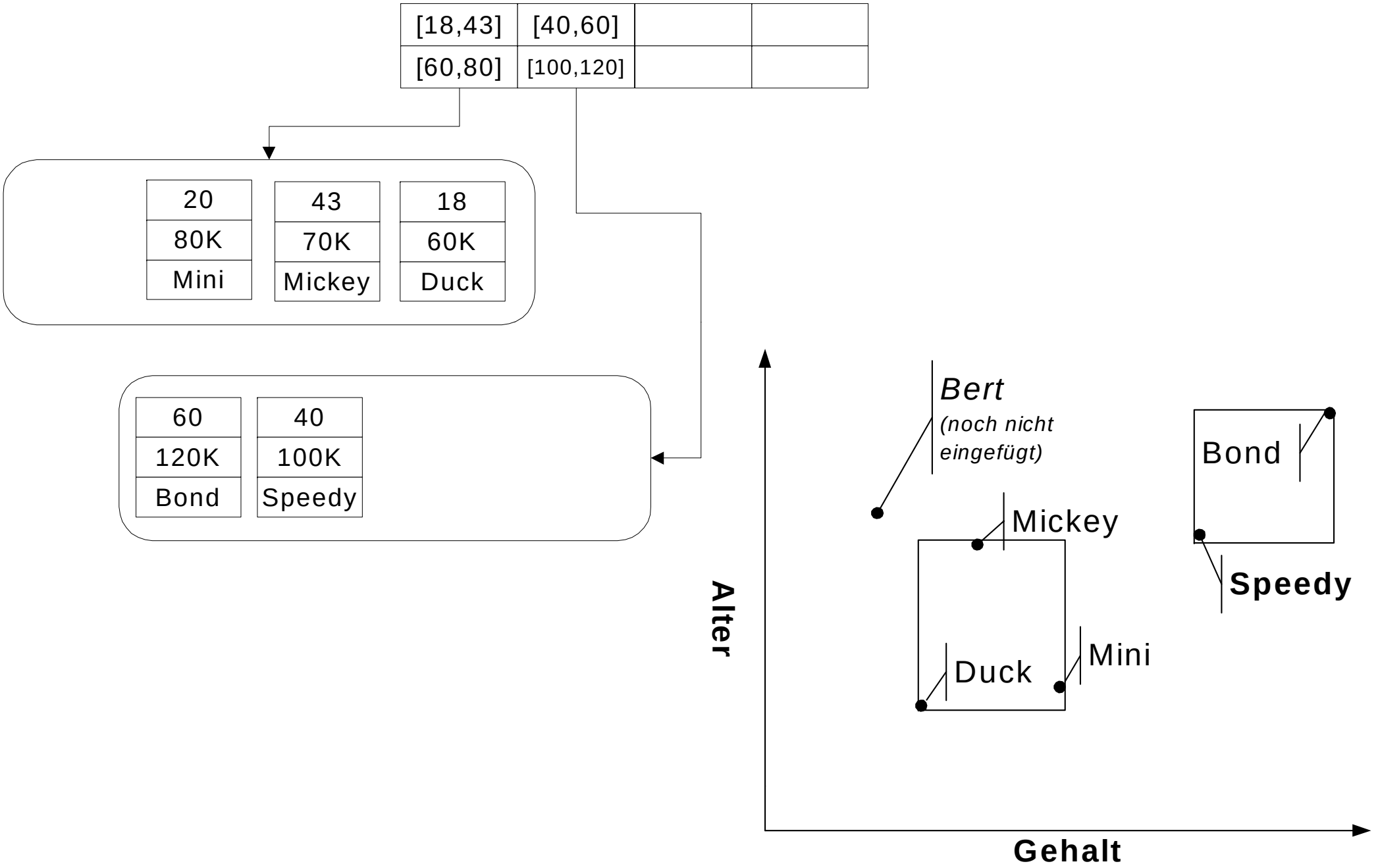
gute Partitionierung



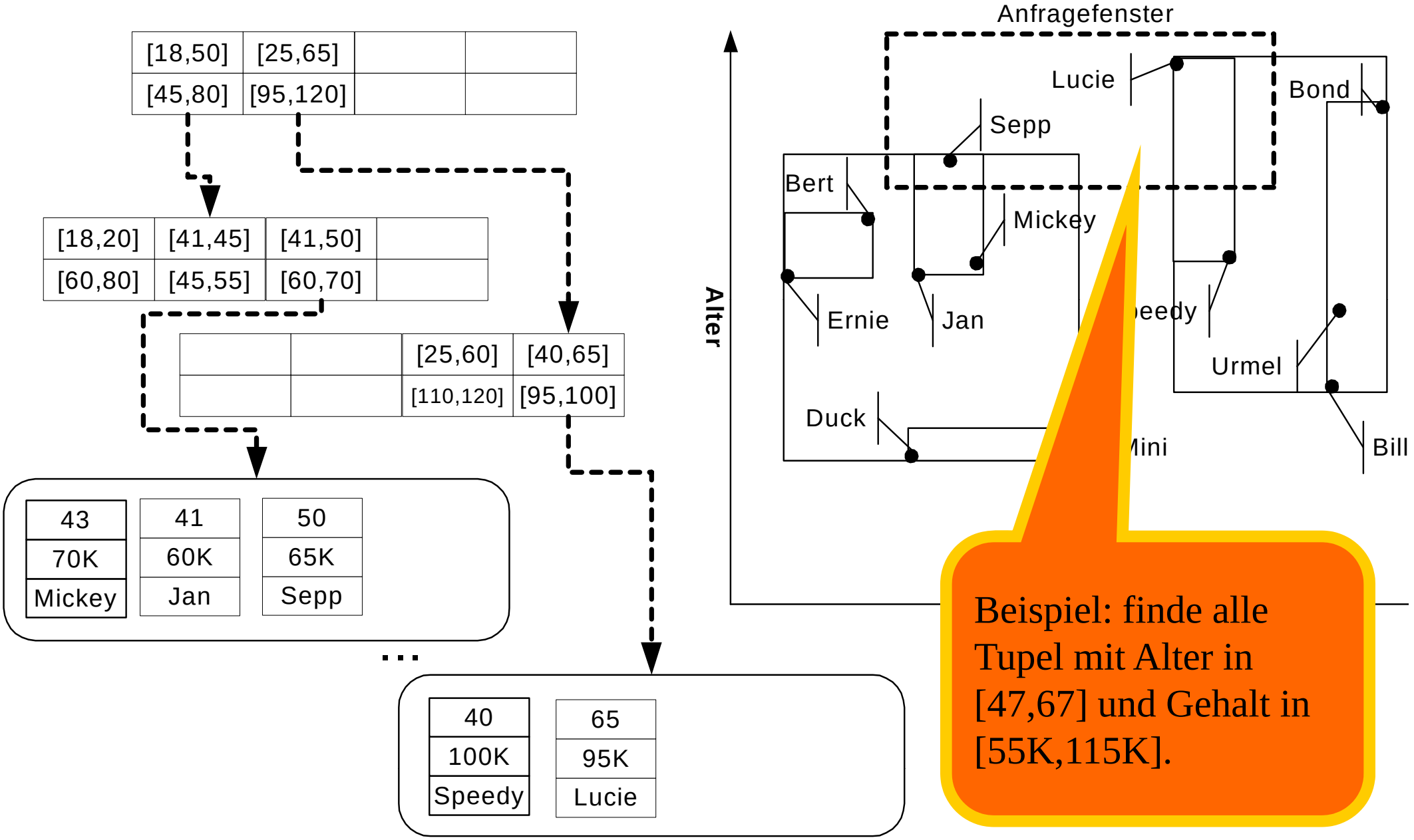
schlechte Partitionierung



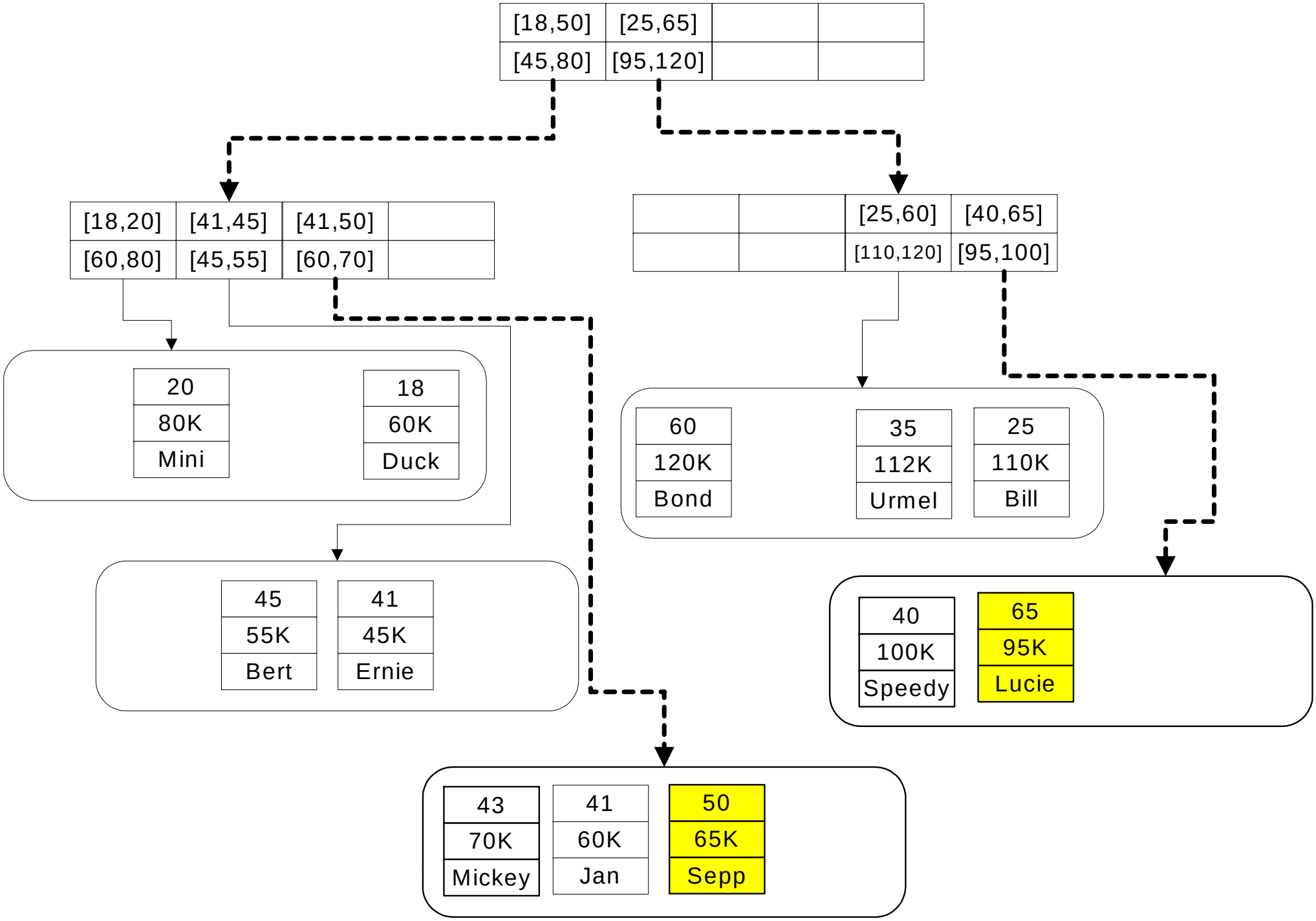
Nächste Phase in der Entstehungsgeschichte des R-Baums



Bereichsanfragen auf dem R-Baum



Beispiel: finde alle Tupel mit Alter in [47,67] und Gehalt in [55K,115K].

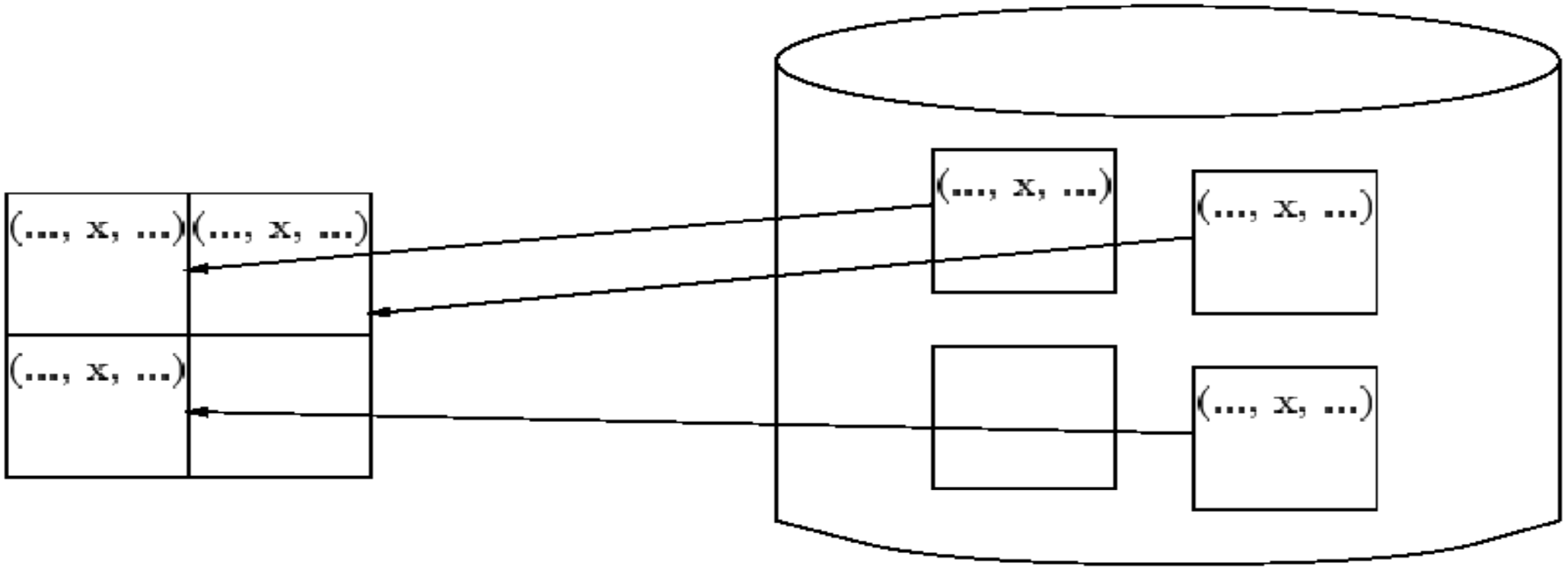

[18,50]
[25,65]
[45,80]
[95,120]
[18,20]
[41,45]
[41,50]
[60,80]
[45,55]
[60,70]
[25,60]
[40,65]
[110,120]
[95,100]
20
80K
Mini
18
60K
Duck
60
120K
Bond
35
112K
Urmel
25
110K
Bill
45
55K
Bert
41
45K
Ernie
40
100K
Speedy
65
95K
Lucie
43
70K
Mickey
41
60K
Jan
50
65K
Sepp

Ballung

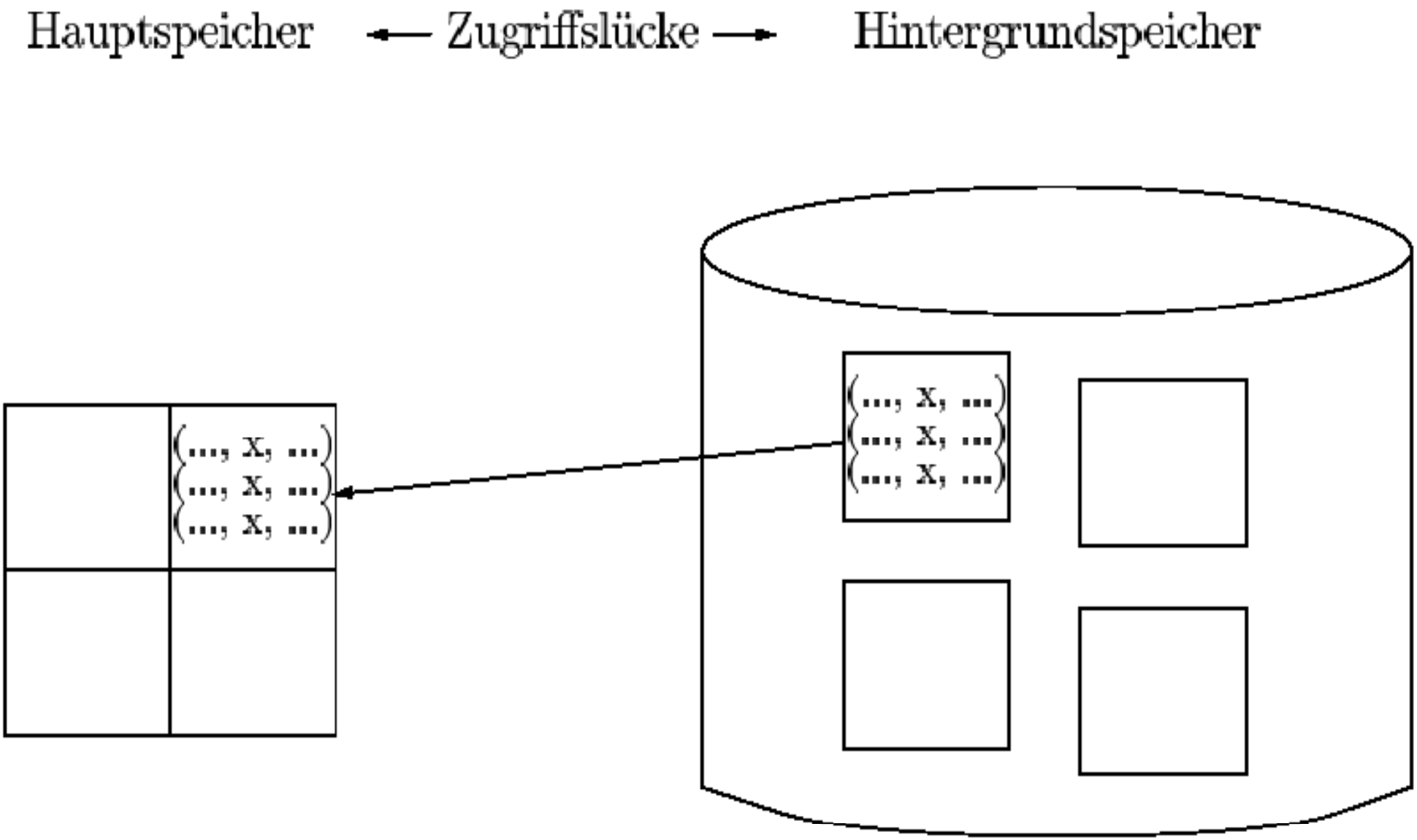
Logisch verwandte Daten liegen am Hintergrundspeicher nahe beieinander

Objektballung / Clustering logisch verwandter Daten

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



Hauptspeicher ← Zugriffslücke → Hintergrundspeicher


Hauptspeicher
Zugriffslücke
Hintergrundspeicher
(..., x, ...)
(..., x, ...)
(..., x, ...)
(..., x, ...)
(..., x, ...)
(..., x, ...)

Seite P_i

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

Seite P_{i+1}

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

⋮

Indexe und Ballung

- Geballter Index = Index, bei dem die Anordnung der Daten-Tupel der Ordnung der Einträge im Index entspricht.
- Beispiel: Angenommen, die Zeilen der Tabelle Student sind auf der Platte nach MatrNr sortiert.
  - Index für Attribut MatrNr: geballt
  - Index für Attribut Semester: nicht geballt
- Nutzen der Ballung bei unterschiedlichen Anfragen:
  - Punktanfrage (exact match): Ballung irrelevant
  - Bereichsanfrage (range selection): Ballung entscheidend

    Beispiel: Select * From Studenten Where Semester > 6;
    bei geballtem Index: Suche ersten Treffer (mittels Index) und durchlaufe ab hier alle Datensätze, solange die Bedingung erfüllt ist.

Index-Einträge und Daten

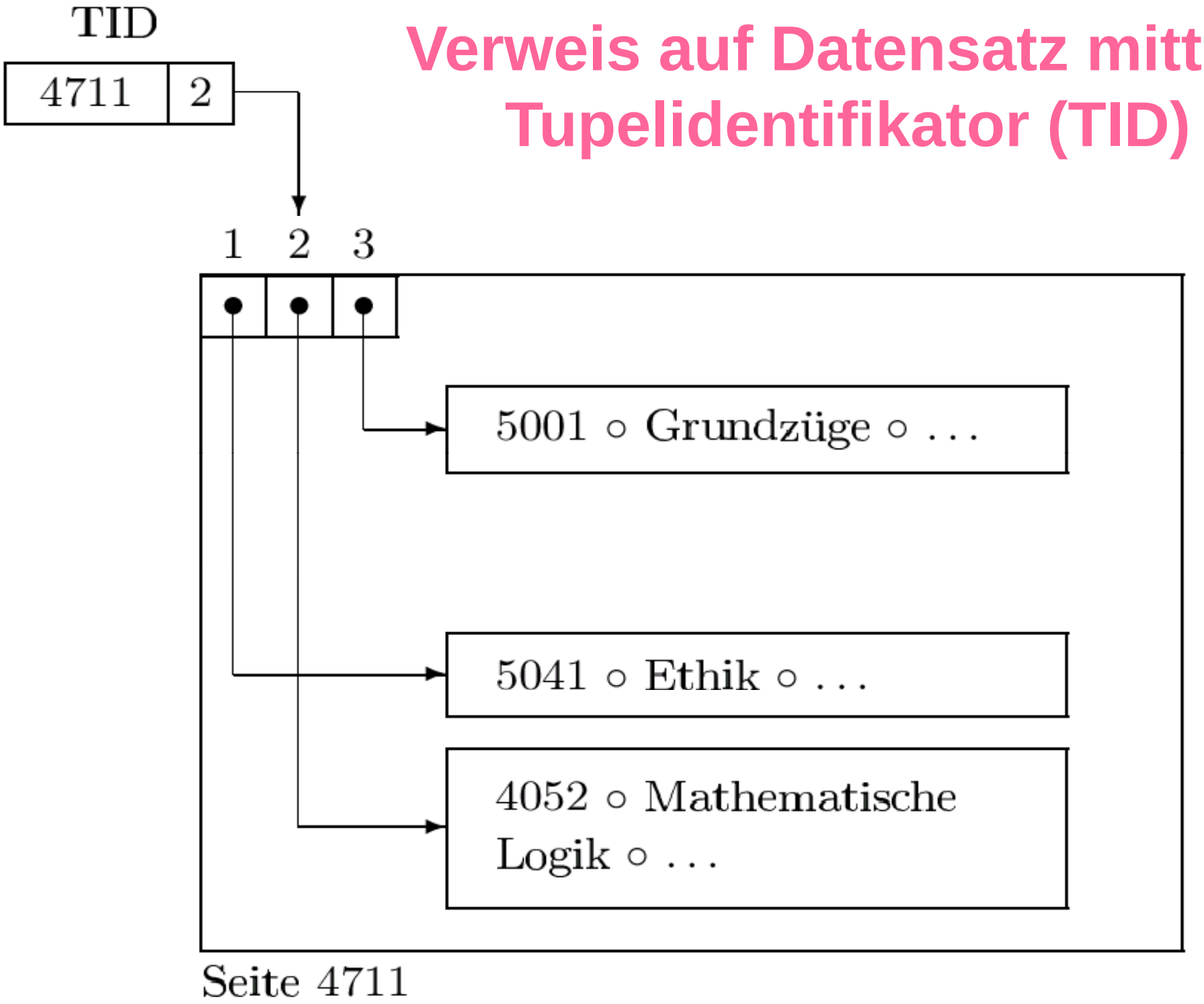
Zwei Alternativen:

1. Der Index enthält (neben der Steuerinformation des Index) die Daten selbst, z.B.:
   - B^+-Index: Zeilen der Tabelle stehen in den Blattknoten
   - Hash-Index: Zeilen der Tabelle stehen in den Buckets
2. Der Index enthält nur Verweise auf die Daten, d.h.: TIDs

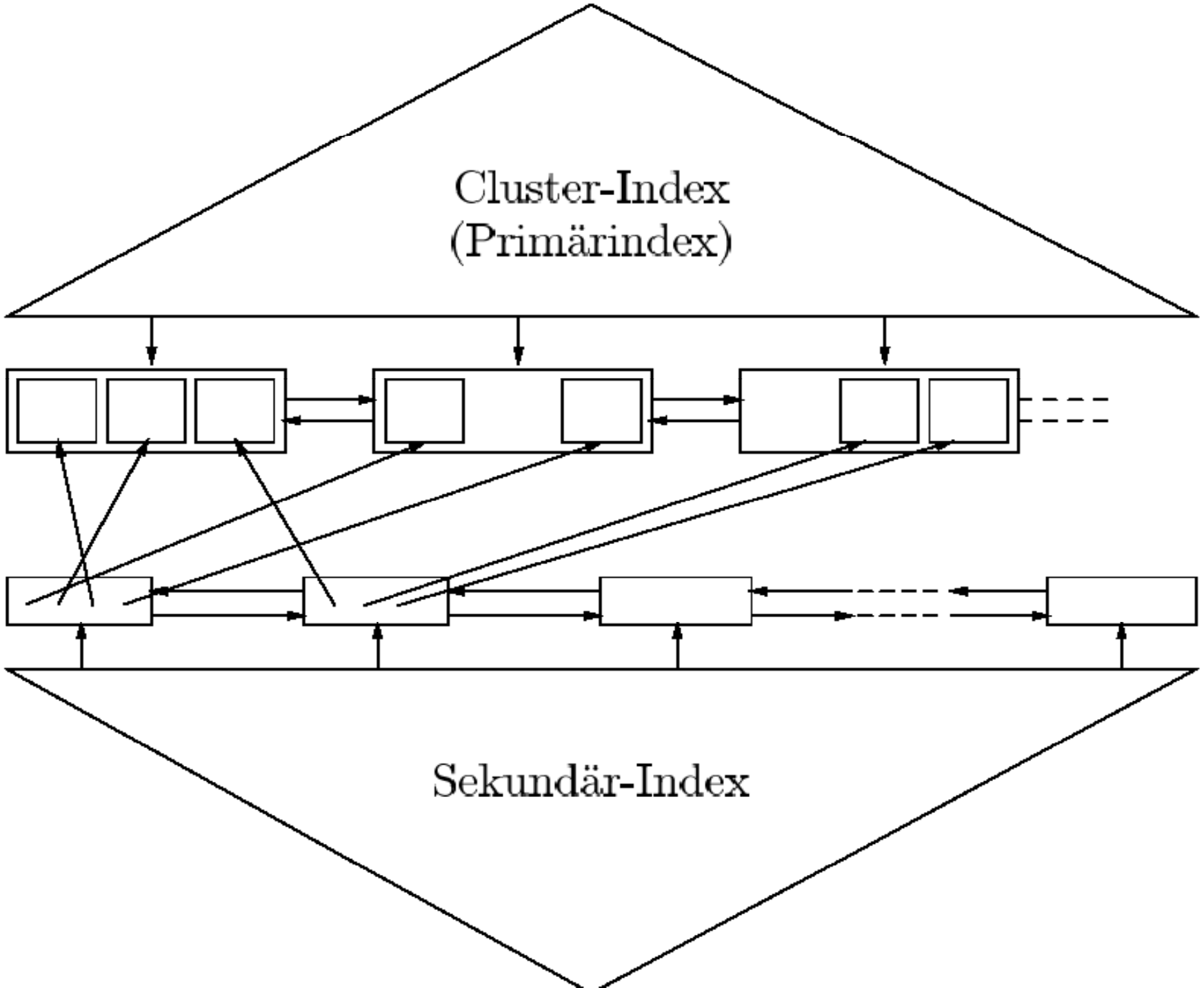
Auswirkung auf Ballung:

- Index mit Alternative 1 ist immer geballt.
- Index mit Alternative 2 kann geballt sein (falls die Zeilen der Tabelle entsprechend dem Suchschlüssel sortiert sind)

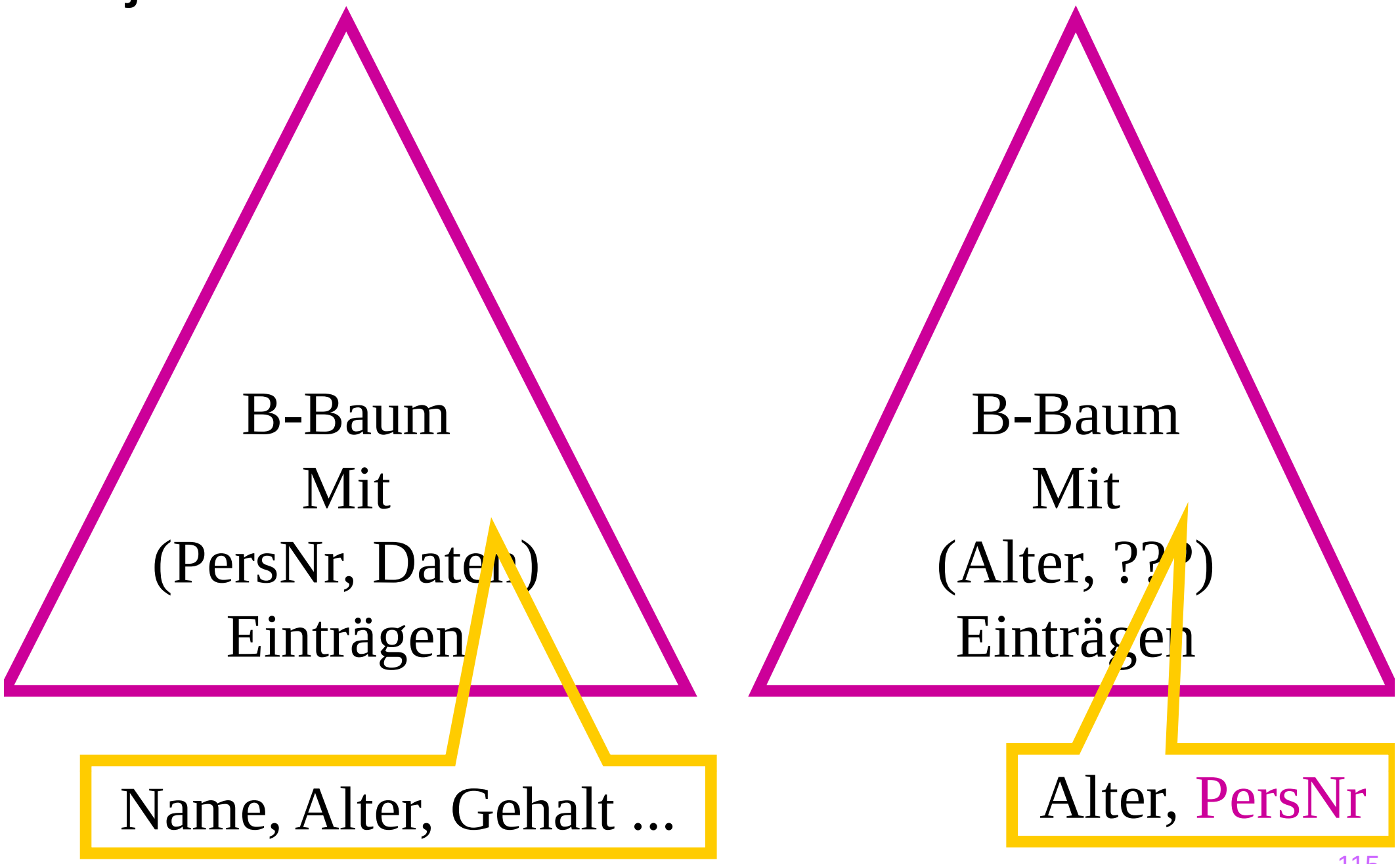
Verweis auf Datensatz mittels Tupelidentifikator (TID)



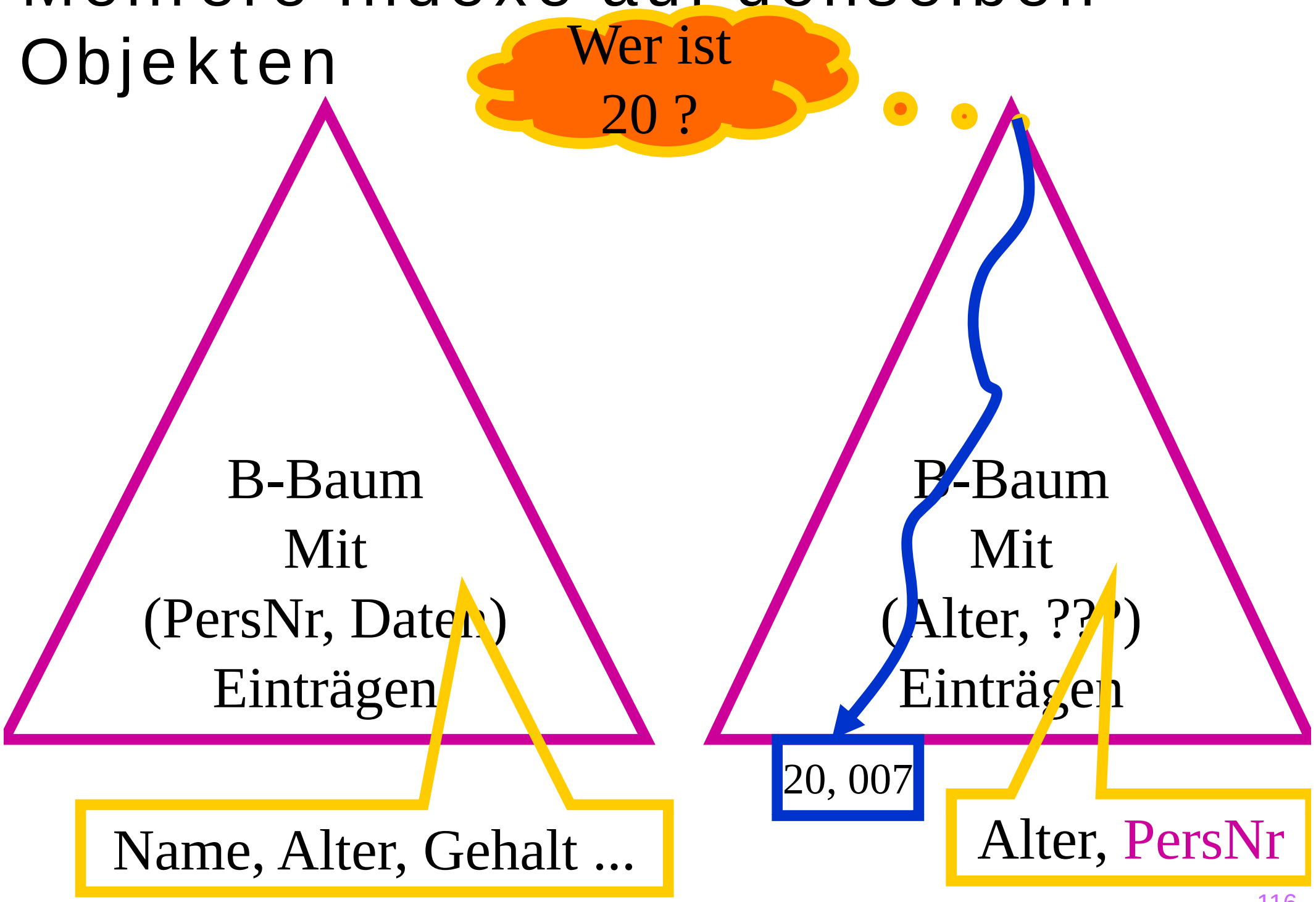
Indexe und Ballung



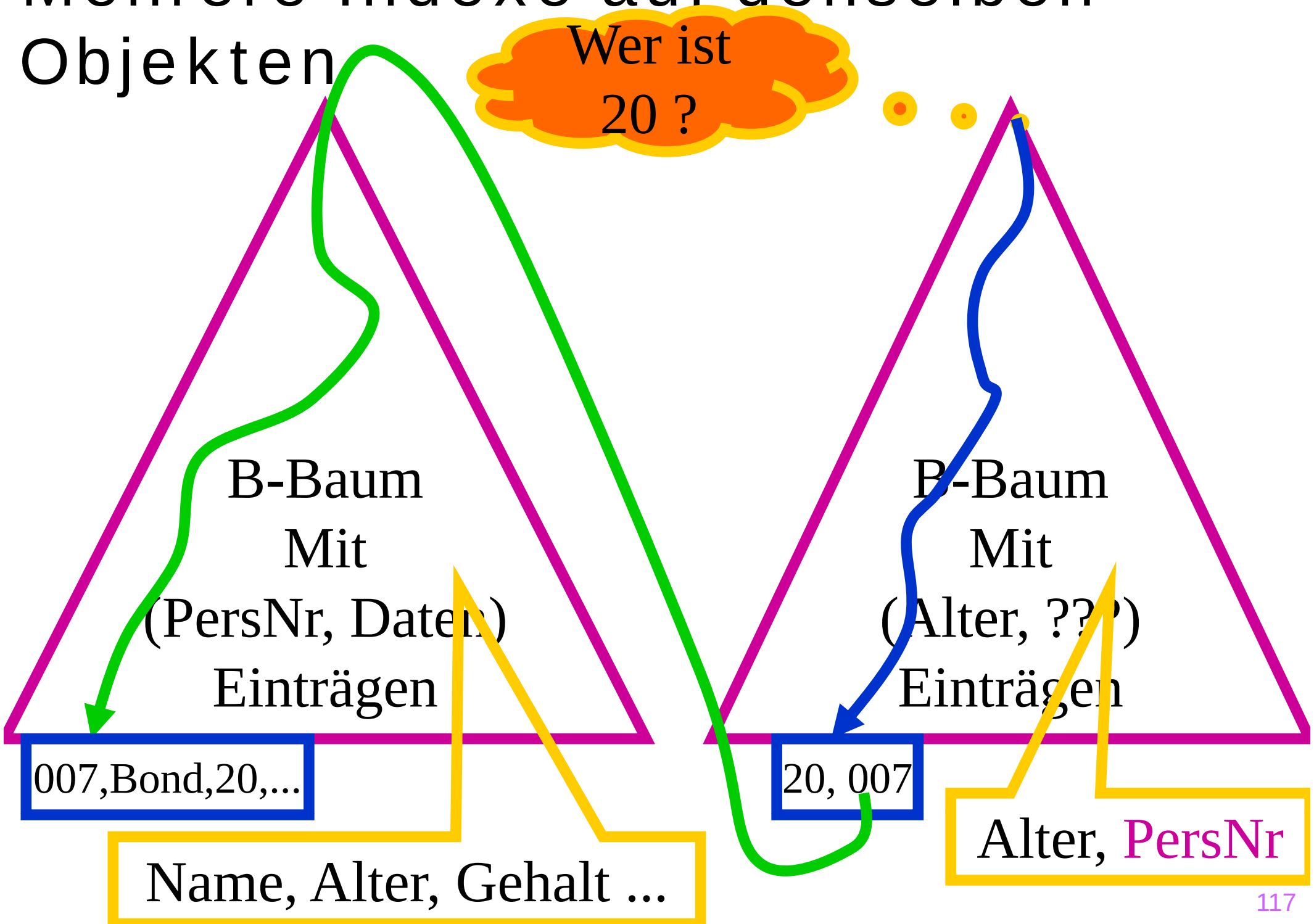
Mehrere Indexe auf denselben Objekten



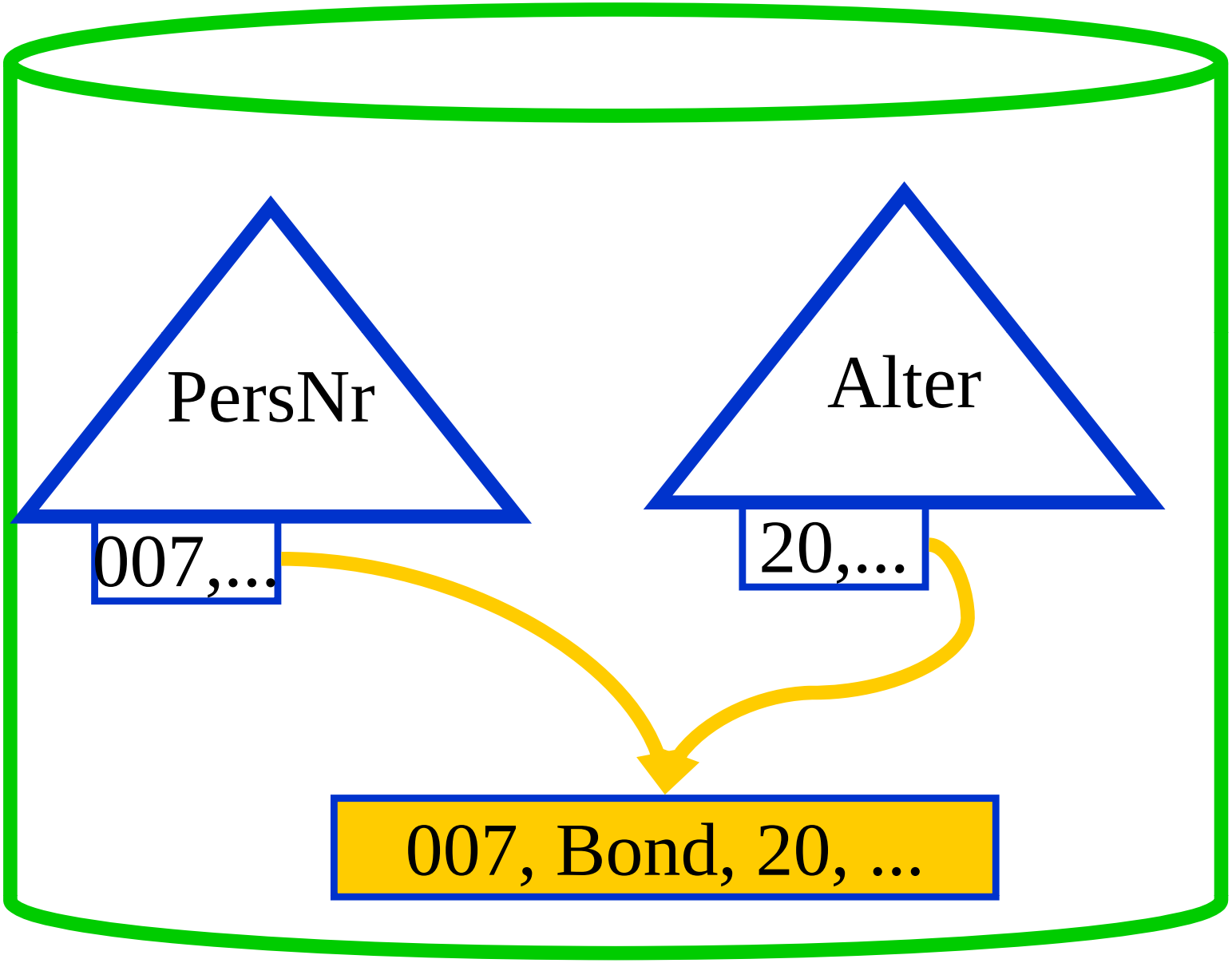
Mehrere Indexe auf denselben Objekten



Mehrere Indexe auf denselben Objekten



Eine andere Möglichkeit: Referenzierung über Speicheradressen



„Beste“ Zugriffsmethode

Beobachtung:
„Beste“ Methode hängt vom Anwendungsfall ab.

Anwendungsfall

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

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

„Beste“ Methode

- Die wichtigsten Zugriffsmethoden:
  - Heap File (ungeordnet bzw. kein "passender" Index)
  - sortierte Datei
  - geballter Index
  - nicht geballter Index
- Die wichtigsten Anwendungsfälle:
  - Durchlaufen des gesamten File (scan)
  - Punktanfrage (exact match)
  - Bereichsanfrage (range selection)
  - Einfügen (insert)
  - Löschen (delete)

„Beste“ Methode

- "Beste" Methode hängt vom Anwendungsfall ab, d.h.: Es gibt keine Methode, die immer besser ist als die anderen, z.B.:
  - Bei Punktanfragen ist Hash-Index im allgemeinen etwas schneller als ein B^+-Baum.
  - Bei Bereichsanfragen ist Hash-Index wesentlich schlechter als ein B^+-Baum.
  - Bei einer Bereichsanfrage mit vielen Treffern wird auch ein nicht geballter B^+-Baum schlechter (wegen Random I/O).
  - Beim Einfügen ist ein ungeordnetes File am schnellsten.
- Gutes Verhalten "in allen Lebenslagen": B^+-Baum

Indexe in SQL

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

```
drop index SemsterInd
```