Vorlesung Implementierung von Datenbanksystemen

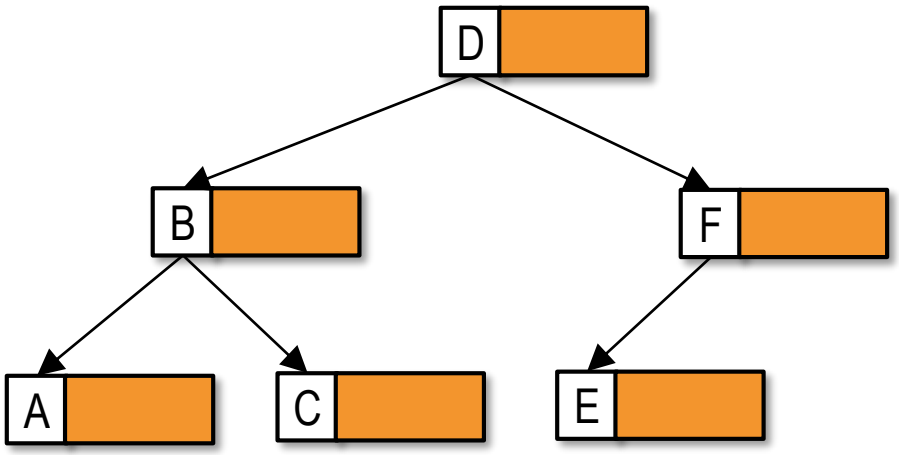
5. Schlüsselzugriff – Teil 2

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

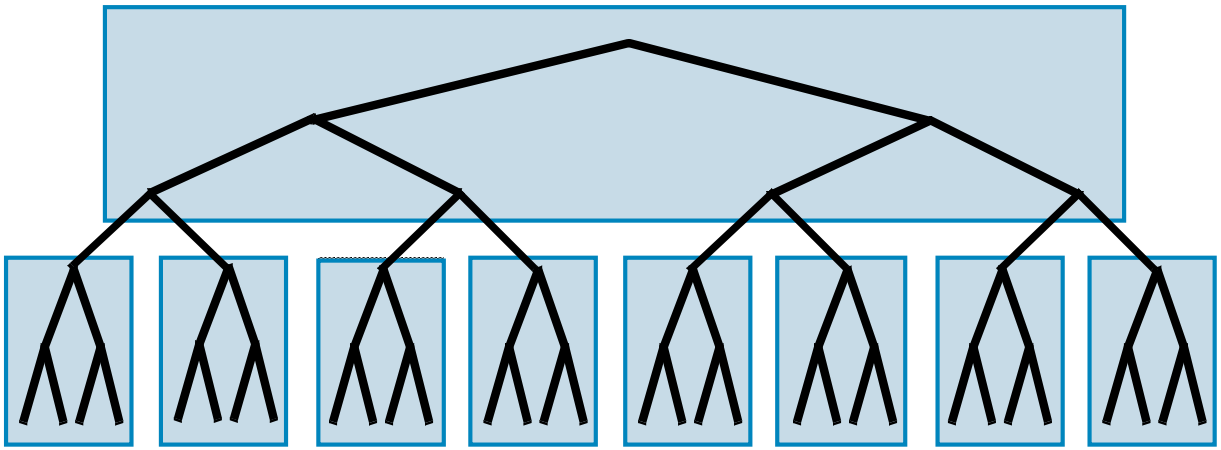
- **Ausgangspunkt: Binäre Such-Bäume (balanciert)**



- Entwickelt für Hauptspeicher
- Prinzipiell machbar auch für Sätze in Dateien:
  Satzadressen als Zeiger
- Dann aber unzureichend:
  Zu viele Blockzugriffe beim Abstieg durch den Baum (einer pro Stufe)

B-Bäume (2)

- **Idee (Bayer und McCreight 1972):**
  - Zusammenfassung ganz bestimmter Sätze in einem Block



  - Mehrweg-Baum,
    bei dem jeder Knoten genau einem Block entspricht
- **Das Ergebnis heißt B-Baum.**
  - B steht für "Block", sagt der Erfinder Rudolf **B**ayer …

Aufbau eines B-Baum-Knotens

n = Anzahl der verwendeten Einträge, **k** ≤ n ≤ 2k (bzw. in der Wurzel 1 ≤ n ≤ 2k)

(Ki, Di, Pi) bilden zusammen einen **Eintrag**

Ki = Schlüsselwert

Di = Datensatz

Pi = Zeiger auf den Nachfolgeknoten (= dessen Blocknummer)

Einträge nach Schlüsselwert aufsteigend sortiert

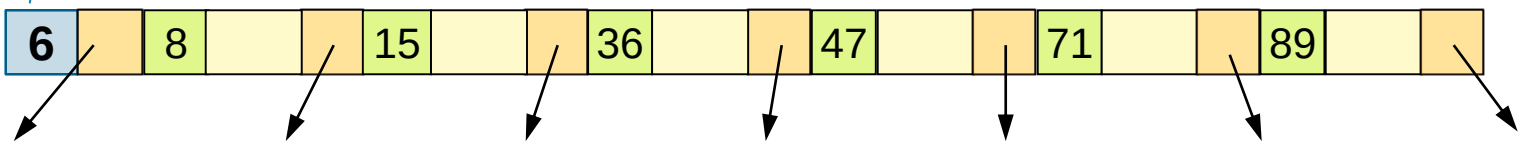
- Zugleich der Inhalt eines Blocks (Wieder ein neuer Blocktyp!)

- **Bedeutung:**
  - Alle Schlüsselwerte im Unterbaum, auf den P0 zeigt, sind kleiner als K1 oder gleich K1
  - Alle Schlüsselwerte im Unterbaum, auf den Pi zeigt (0 < i < n), sind größer als Ki und kleiner oder gleich Ki+1
  - Alle Schlüsselwerte im Unterbaum von Pn sind größer als Kn

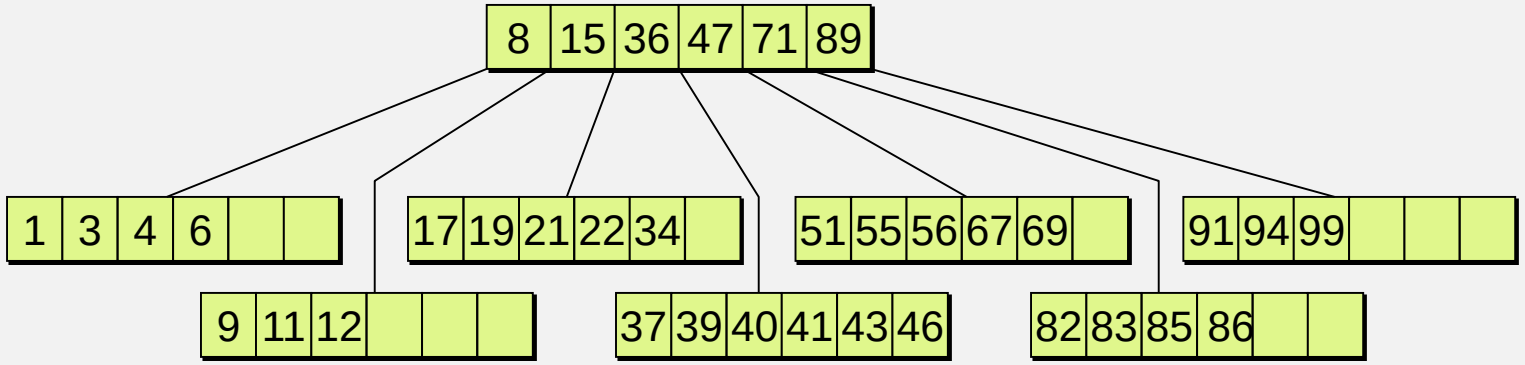
B-Baum: Beispiel mit k = 3

Aufbau eines Knotens: maximal 6 Einträge

Anzahl der tatsächlichen Einträge in diesem Knoten

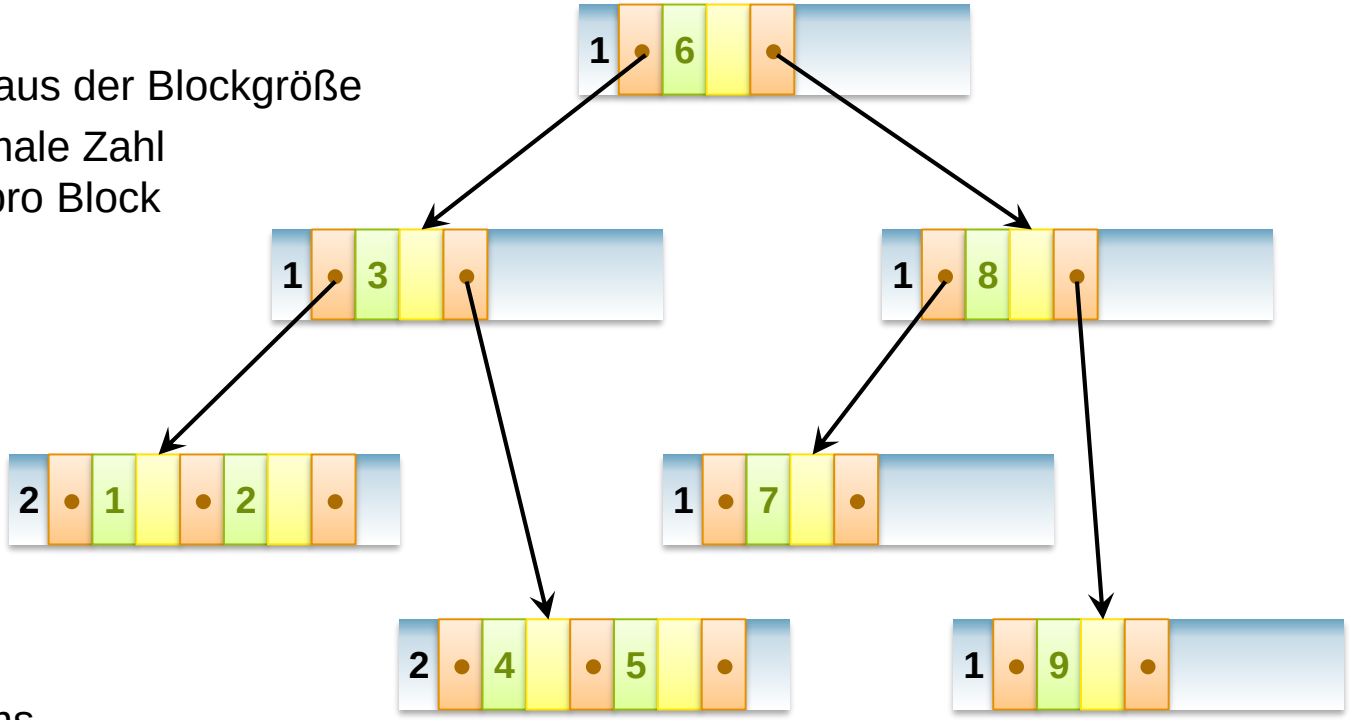


B-Baum der Höhe 2



B-Baum-Parameter

- **k**
  - Errechnet sich aus der Blockgröße
  - 2k ist die maximale Zahl von Einträgen pro Block



- **h**
  - Höhe des Baums
    - Anzahl der Kanten von der Wurzel bis zum Blatt plus 1, also Anzahl der Ebenen
  - Ergibt sich aus:
    Anzahl der gespeicherten Datenelemente und Einfügereihenfolge

Im Beispiel:
k = 1, h = 3

B-Baum-Eigenschaften

- **Jeder Pfad**
  - vom Wurzelknoten zu einem der Blattknoten hat **dieselbe Länge h–1.**
    - Baum ist perfekt balanciert.
- **Jeder Knoten**
  - mit Ausnahme des Wurzelknotens und der Blattknoten hat **mindestens k+1 Nachfolger.**
    - Jeder Block also mindestens halb voll: Speicherplatz-Ausnutzung > 50 % (bei jeder Zahl von Sätzen).
- **Der Wurzelknoten**
  - ist entweder ein Blattknoten oder hat mindestens 2 Nachfolger.
- **Jeder Knoten**
  - hat **höchstens 2k+1 Nachfolger.**
    - Ergibt sich aus der Block-Größe: Dann ist der Block voll.

Suche im B-Baum

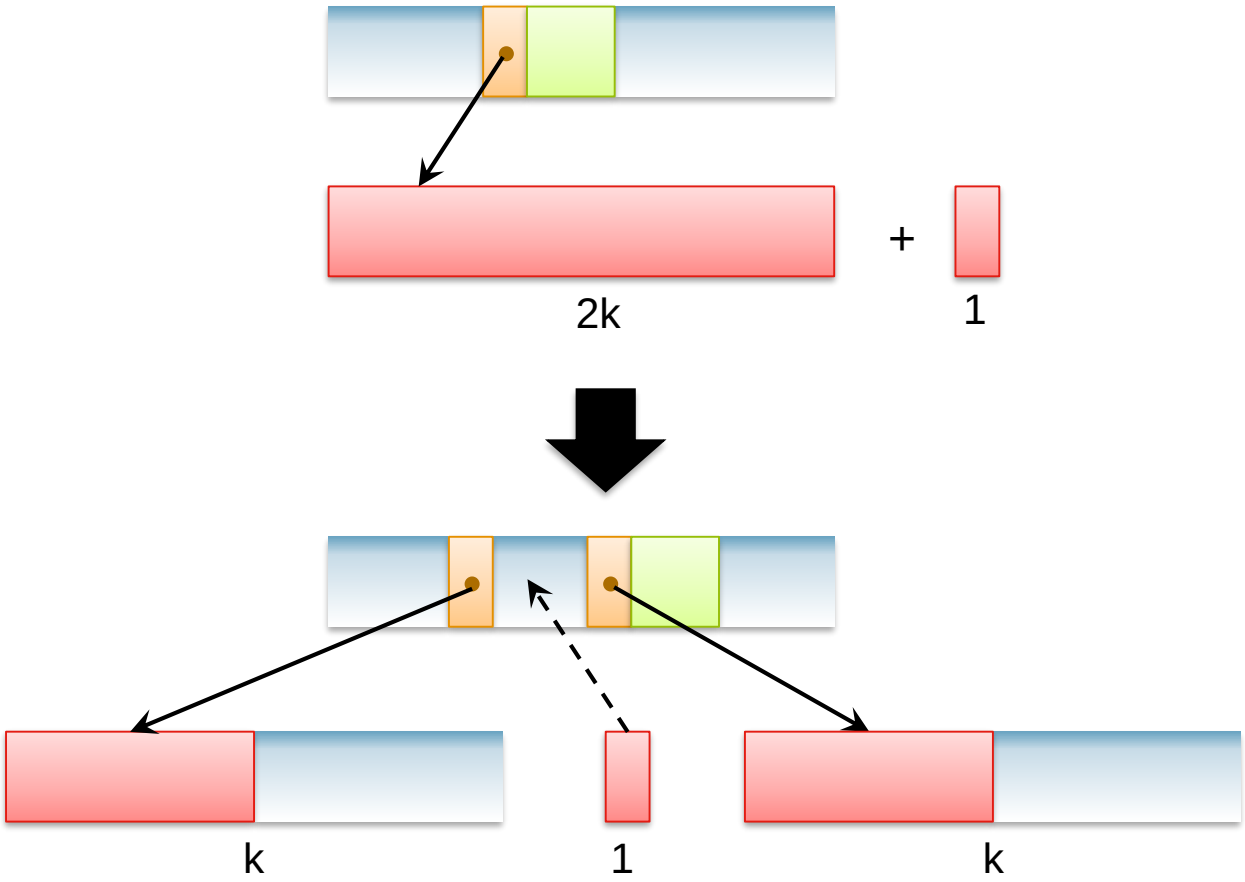
- **Beginnend mit dem Wurzelknoten,
wird ein Knoten jeweils von links nach rechts durchsucht:**
  (1) Stimmt Ki mit dem gesuchten Schlüsselwert überein,
  ist der Satz gefunden.
  (2) Ist Ki größer als der gesuchte Wert,
  wird die Suche in der Wurzel des an Pi–1 hängenden Unterbaums
  fortgesetzt.
  (3) Ist Ki kleiner als der gesuchte Wert,
  wird der Vergleich mit Ki+1 wiederholt.
  (4) Ist auch Kn noch kleiner als der gesuchte Wert,
  wird die Suche im Unterbaum von Pn fortgesetzt.
- **Falls weiterer Abstieg in Unterbaum (über Pi–1 in (2) oder Pn in (4)) nicht möglich (d.h. Blattknoten):**
  - Suche abbrechen,
  kein Satz mit gewünschtem Schlüsselwert vorhanden

Einfügen eines Satzes im B-Baum

- **Eingefügt wird nur in Blattknoten!**
  - D.h. zunächst Abstieg durch den Baum wie bei der Suche
  - Im so gefundenen Blattknoten
    Satz entsprechend der Sortierreihenfolge einfügen

- **Sonderfall: Blattknoten schon voll (enthält 2k Sätze)**
  - **Splitt:** neuen Blattknoten erzeugen
  - Die 2k+1 Sätze (in Sortierordnung!) halbe-halbe aufteilen
    zwischen altem und neuem Blattknoten:
    - Die ersten k Sätze in den ersten (linken) Block
    - Die letzten k Sätze in den zweiten (rechten) Block
    - Den mittleren (k+1-ten) Satz als neuen "Diskriminator",
      d.h. als Verzweigungsinformation bei der Suche, in den Knoten eine Stufe höher einfügen, der auf den Blattknoten verweist (zusammen mit einem Verweis auf den neuen Blattknoten)

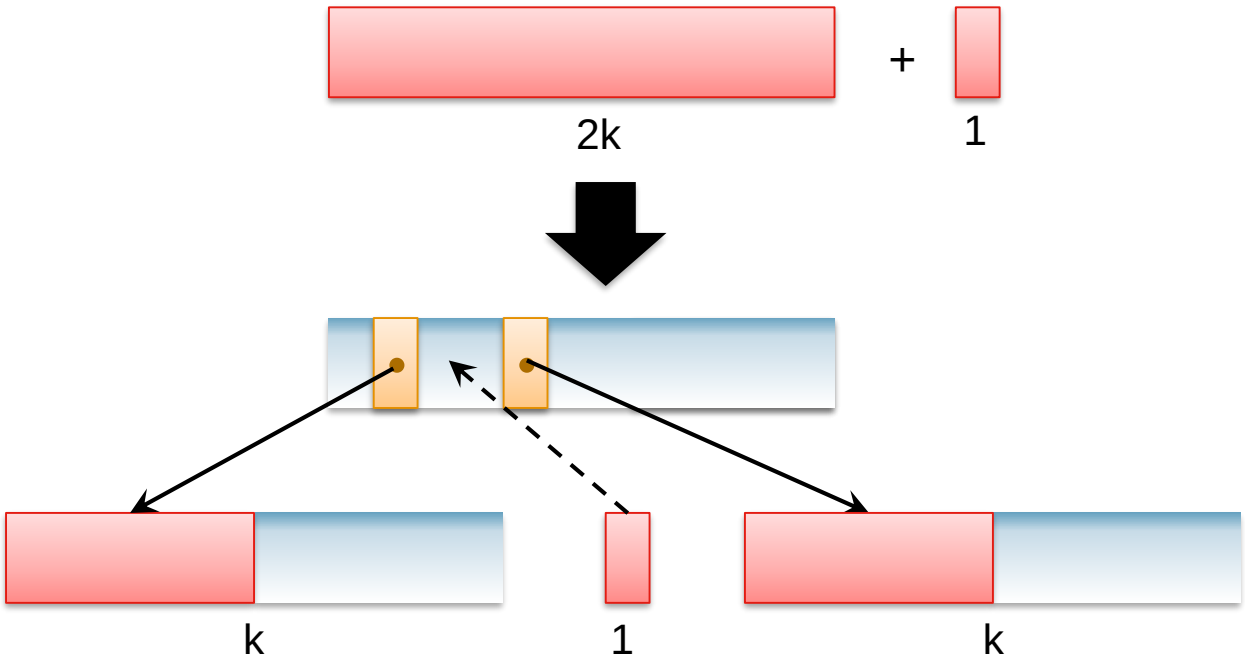
Einfügen B-Baum (2)



- Falls auch der übergeordnete Knoten voll:
  - Splitt auf dieser Ebene wiederholen

Einfügen B-Baum (3)

- **Weiterer Sonderfall: Splitt des Wurzelknotens**
  - Erzeugung von *zwei* neuen Knoten:



  - Dann (und nur dann!) wächst die Höhe des Baums um 1.
    - (Man sagt bildhaft: Der Baum "reißt von unten nach oben auf".)

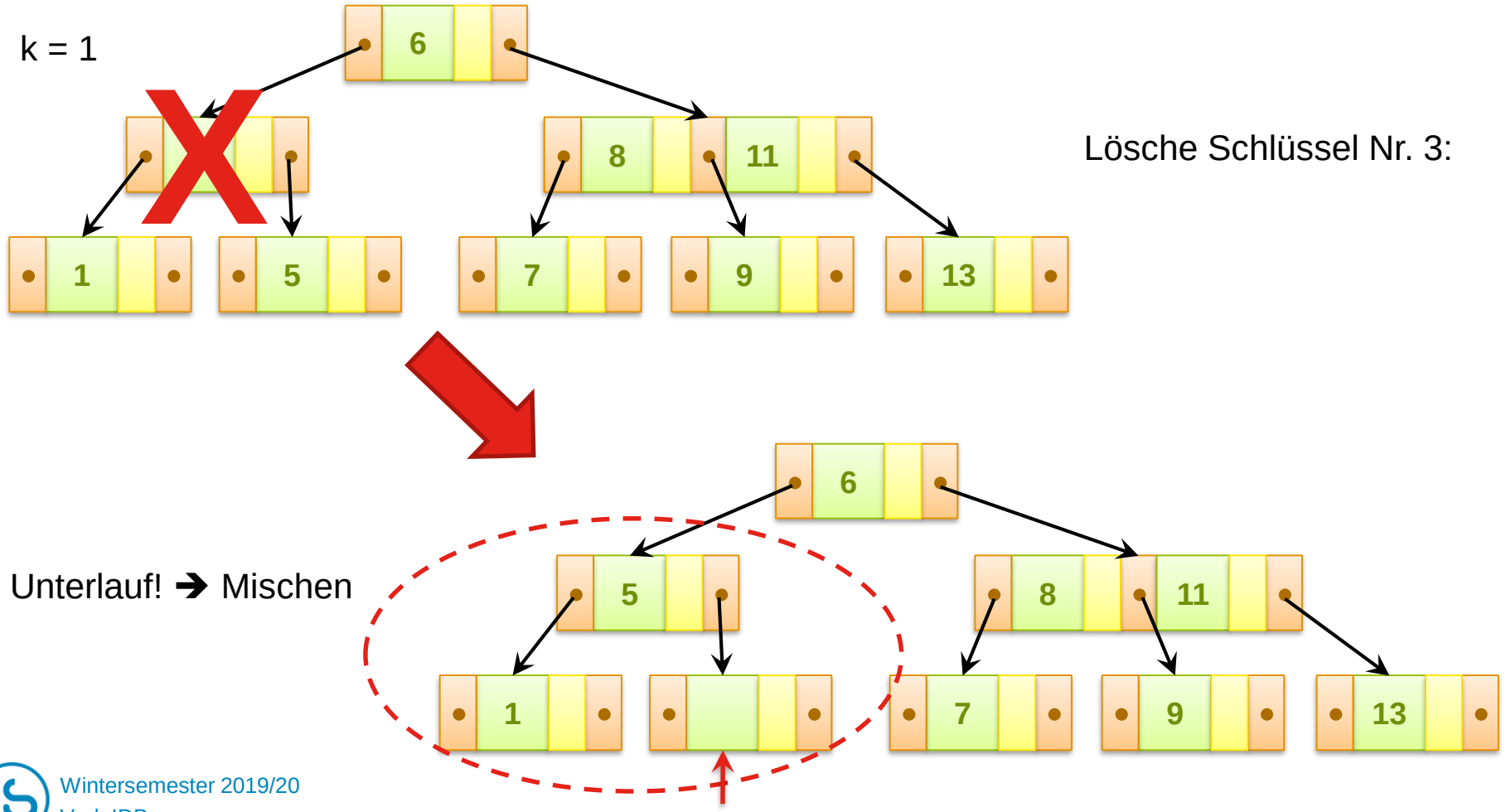
Einfügen B-Baum (4)

- **Dynamische Reorganisation**
  - Kein Entladen und Neuladen erforderlich
  - Baum immer balanciert

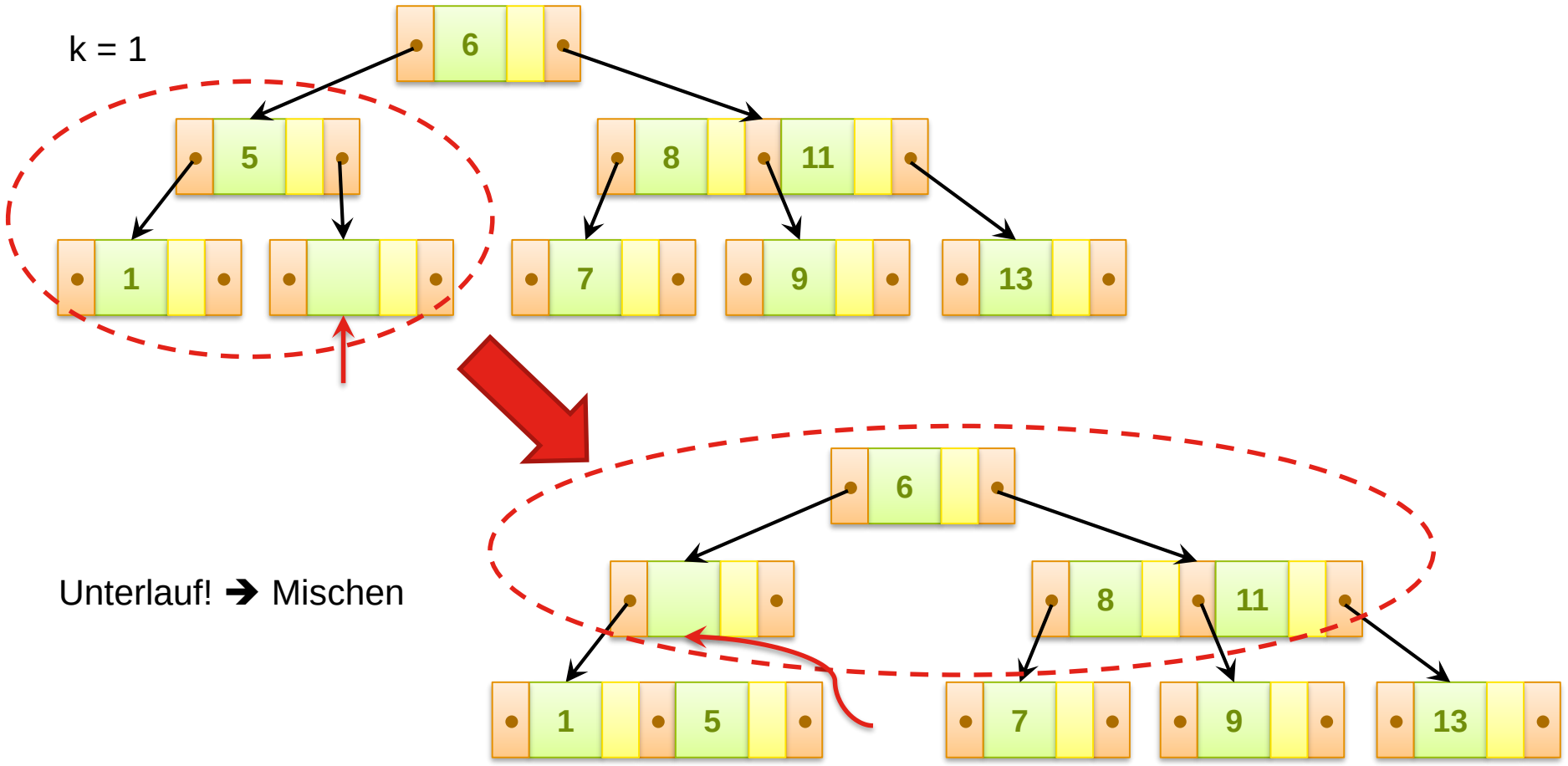
- **Speicherplatzausnutzung:**
  - Jeder Knoten (bis auf die Wurzel)
    ist immer mindestens halb voll,
    d.h. Speicherausnutzung garantiert $\geq 50\ \%$
  - Bei zufälliger und gleichverteilter Einfügung ergibt sich eine Speicherausnutzung von ln 2, also rund 70 %

Löschen im B-Baum

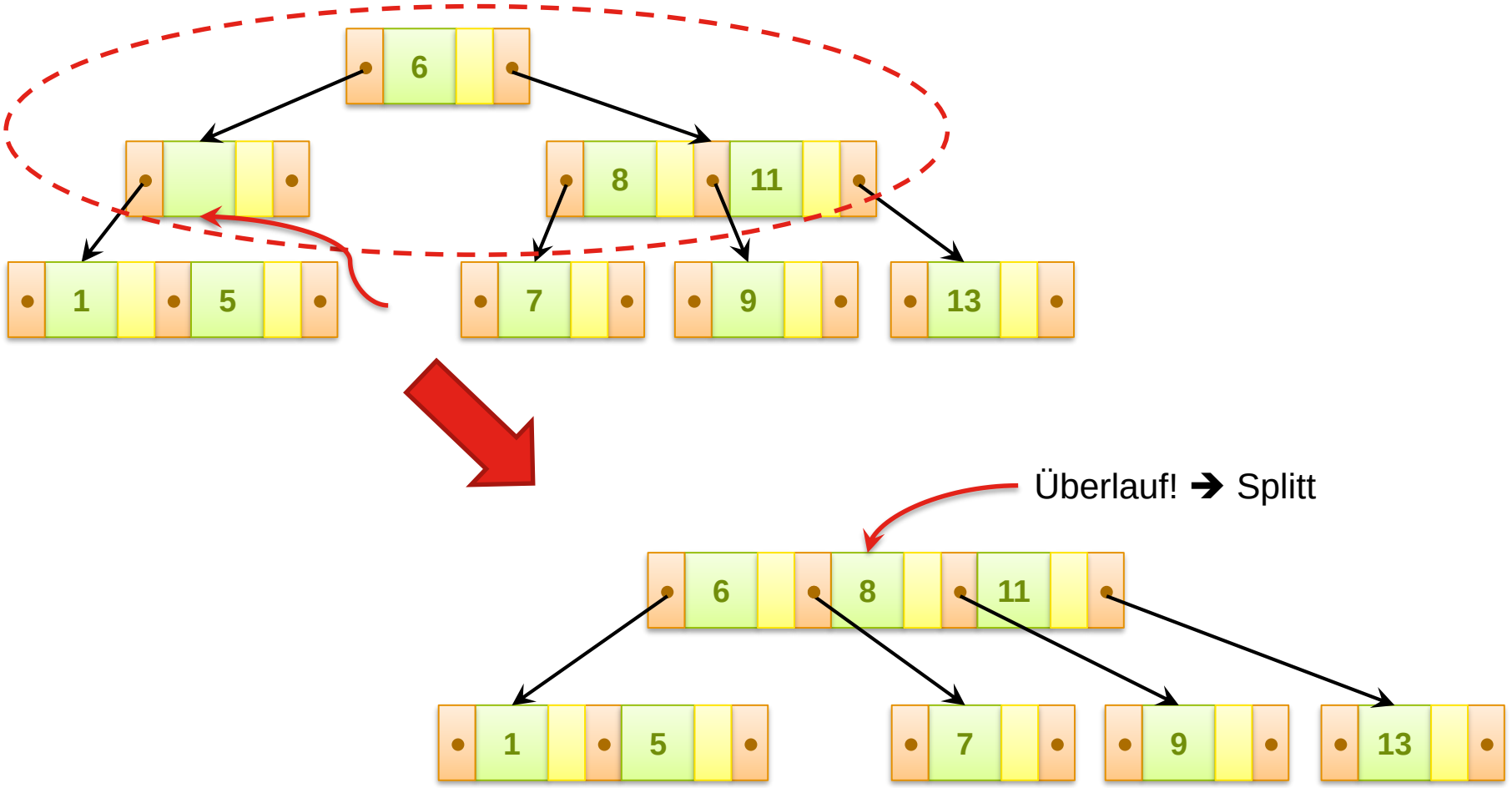
- **... erstmal am Beispiel !!**



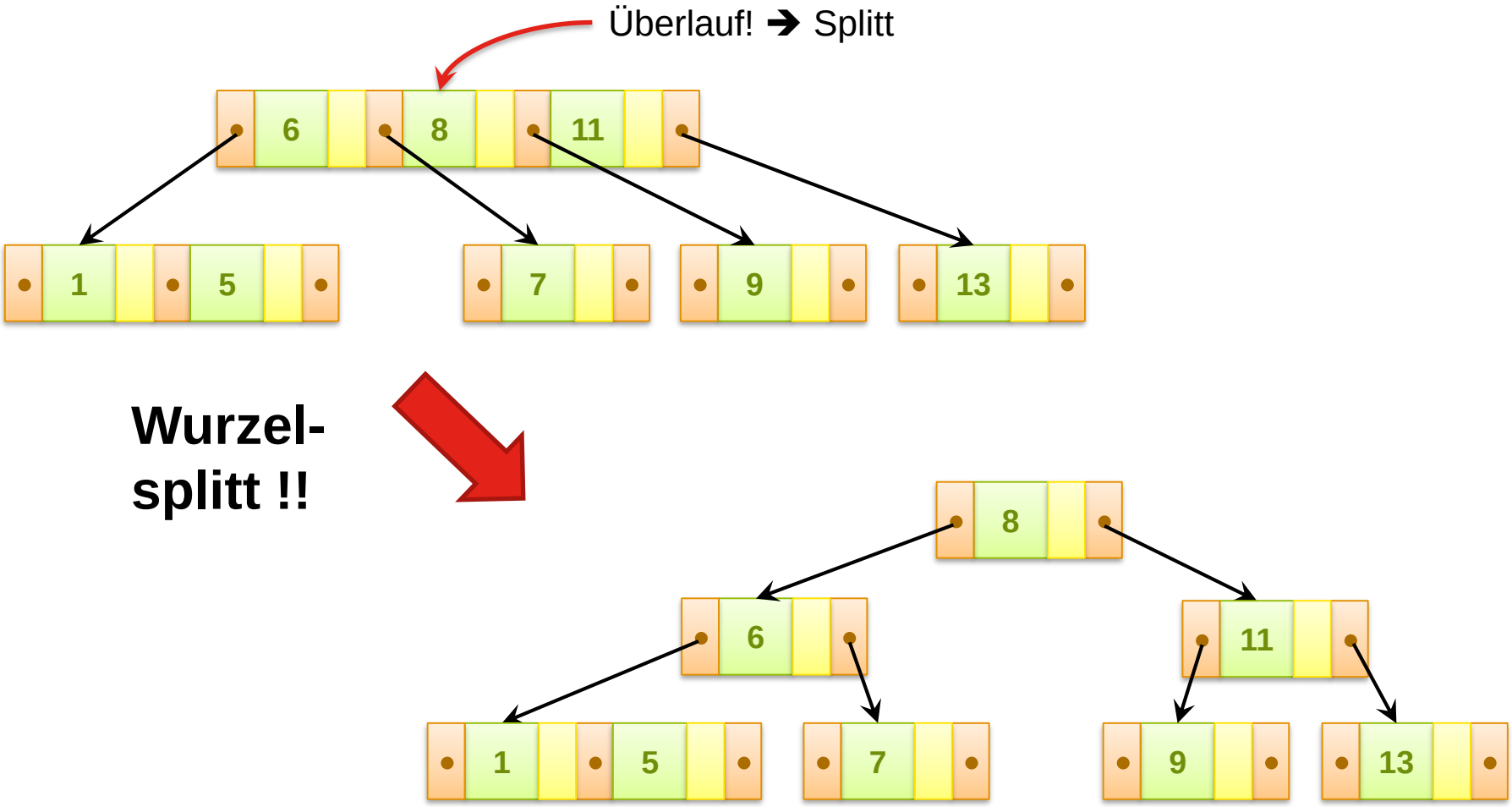
Löschen im B-Baum (2)



Löschen im B-Baum (3)



Löschen im B-Baum (4)



Löschvorgang – algorithmisch

- **War nur Beispiel – es gibt verschiedene Algorithmen**
  - Such den Knoten, in dem der **zu löschende Schlüssel S** liegt.
  - Falls Schlüssel S **im Blattknoten**,
    dann lösch S dort
    und behandle ggf. entstehenden Unterlauf.
  - Falls Schlüssel S **im innerem Knoten**,
    dann untersuch linken und rechten Unterbaum von S:
    - Betrachte Blattknoten mit direktem Vorgänger S' von S
      und Blattknoten mit direktem Nachfolger S'' von S.
    - Wähl den aus, der mehr Elemente hat.
      Falls beide gleich viele Elemente haben, wähl zufällig einen aus.
    - Ersetz den zu löschenden Schlüssel S
      durch S' bzw. S'' aus dem gewählten Blattknoten.
    - Lösch S' bzw. S'' im gewählten Blattknoten
      und behandle ggf. entstehenden Unterlauf.

Anmerkungen zum Unterlauf

- **Anmerkungen**
  - Ein endgültiger Unterlauf entsteht bei obigem Algorithmus erst auf Blattebene!
  - Unterlaufbehandlung wird durch Mischen des Unterlaufknotens mit seinem Nachbarknoten und dem darüber liegenden Diskriminator durchgeführt.
    - Sozusagen Splitt rückwärts ausführen
  - Wurde einmal mit dem Mischen auf Blattebene begonnen, so setzt sich dieses evtl. nach oben hin fort.
  - Das Mischen auf Blattebene wird so lange weitergeführt, bis kein Unterlauf mehr existiert oder die Wurzel erreicht ist.
  - Wird die Wurzel erreicht, kann der Baum in der Höhe um 1 schrumpfen. Beim Mischen kann es auch wieder zu einem Überlauf kommen. In diesem Fall muss wieder gesplittet werden.

Variante: B*-Baum

(Bisweilen auch B+-Baum genannt)

- **Eigenschaften und Unterschiede zum B-Baum**
  - Alle Sätze werden in den **Blattknoten** abgelegt.
  - **Innere Knoten** enthalten nur noch Verzweigungsinformation, keine Daten.

  - Aufbau von B*-Baum-Knoten:

Innerer Knoten

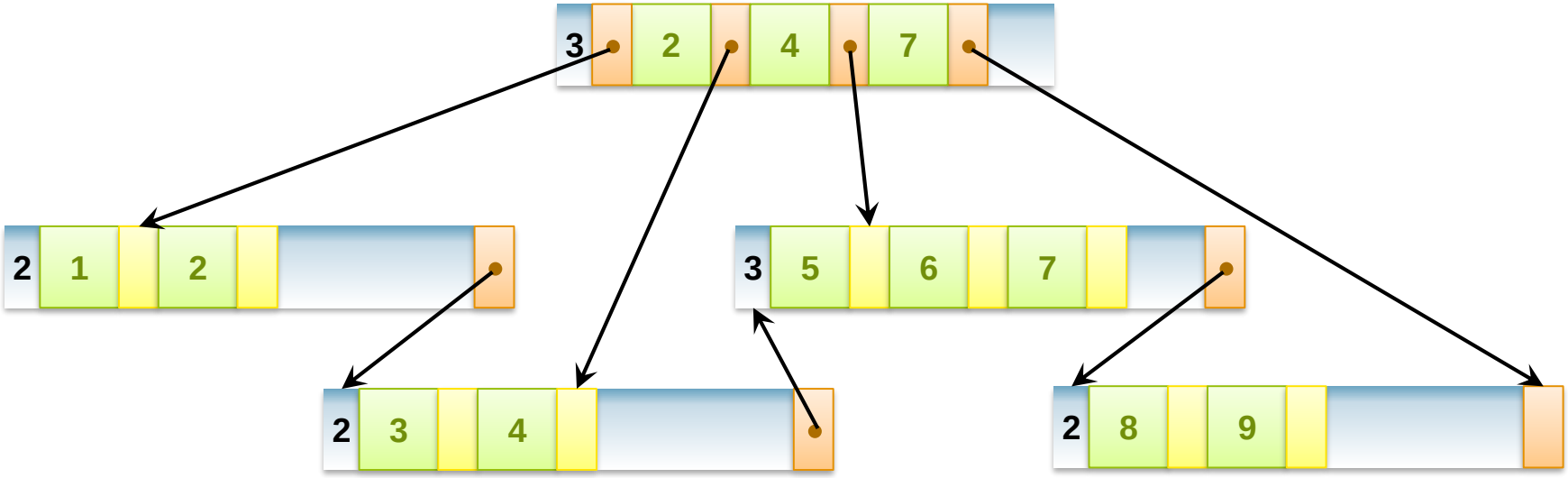
P_i = Zeiger auf Knoten, R_i = Referenzschlüssel, $k \leq m \leq 2k$

Blatt-Knoten

V = Vorgänger-Zeiger, N = Nachfolger-Zeiger, $k^* \leq j \leq 2k^*$

Beispiel eines B*-Baums

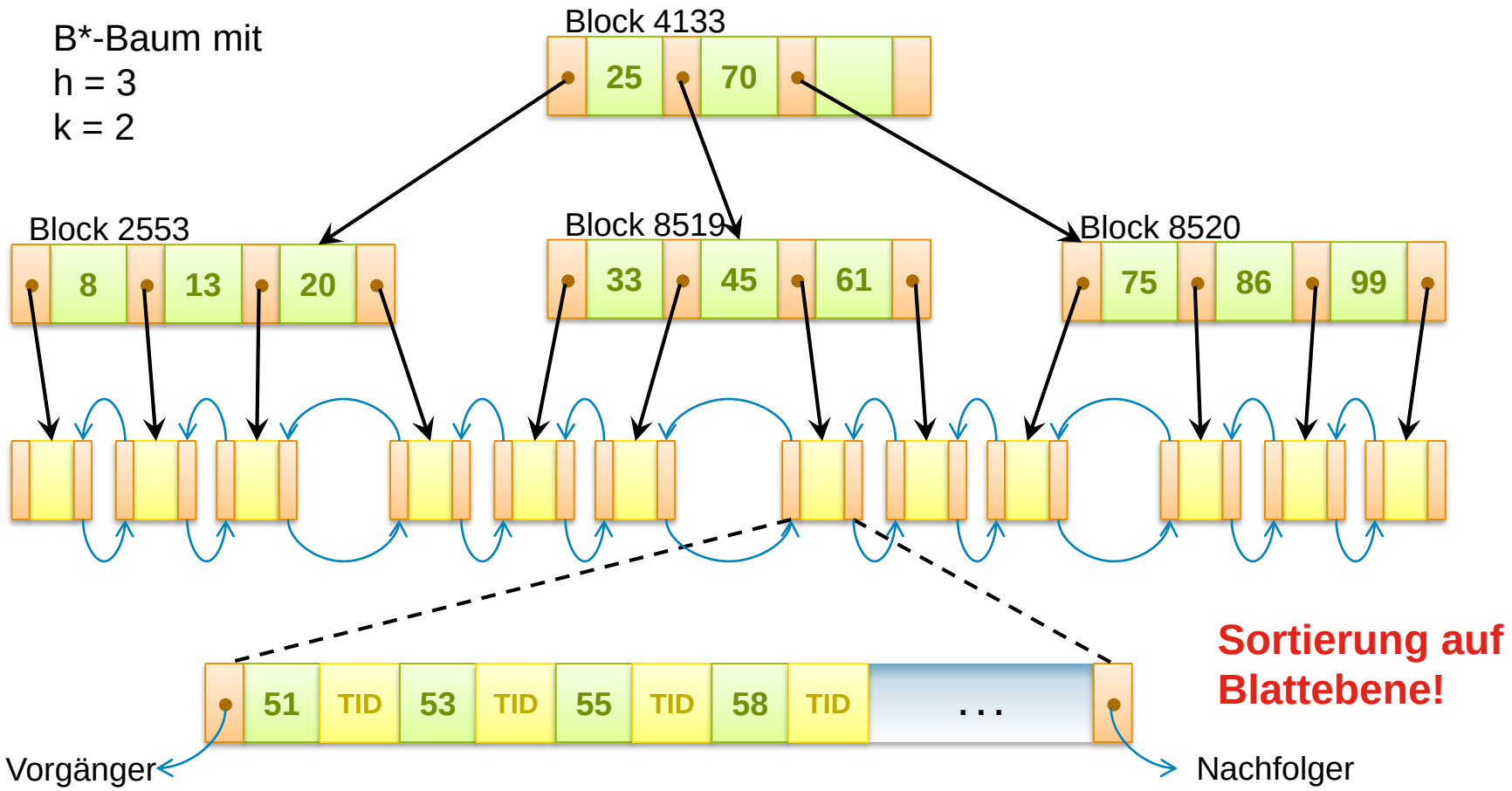
- Ohne Vorgänger-Zeiger



- **Sortierung der Sätze auf Blattebene!**

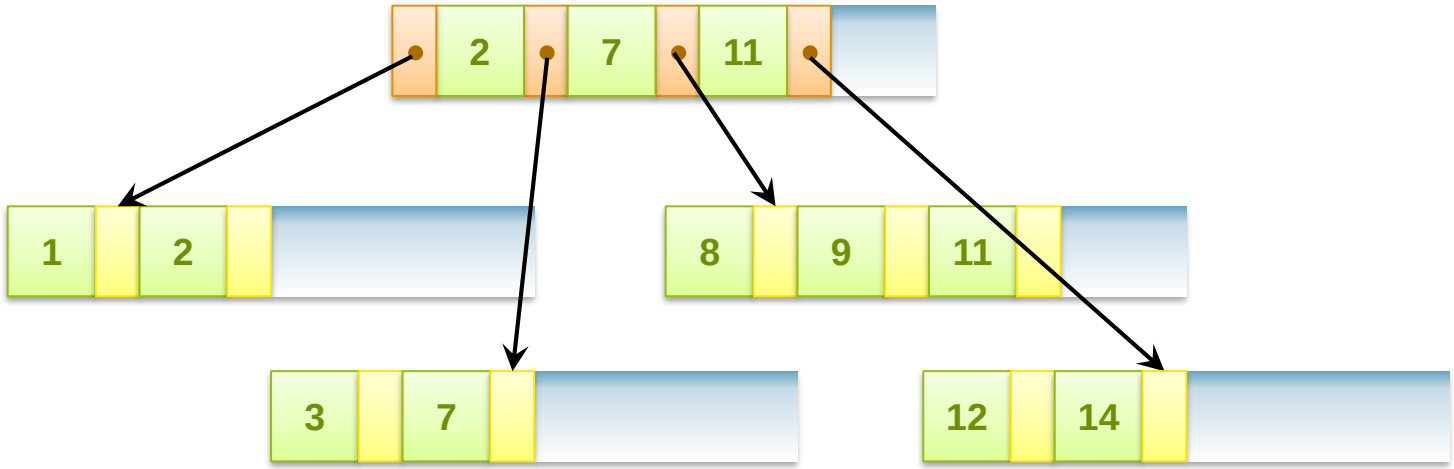
Weiteres Beispiel eines B*-Baums

- Etwas umfangreicher:

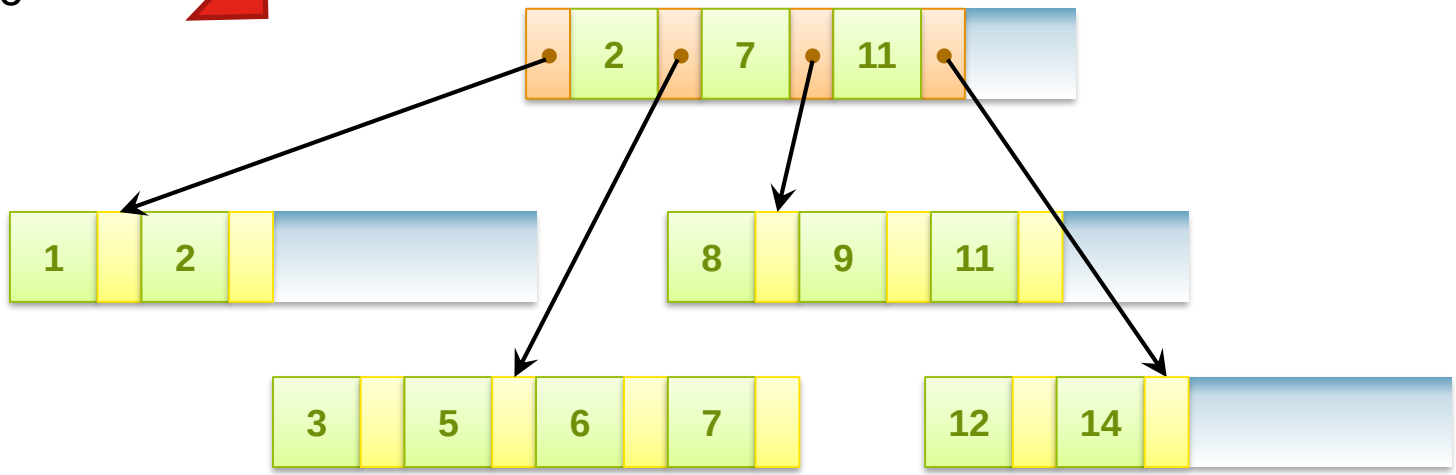


Einfügen im B*-Baum

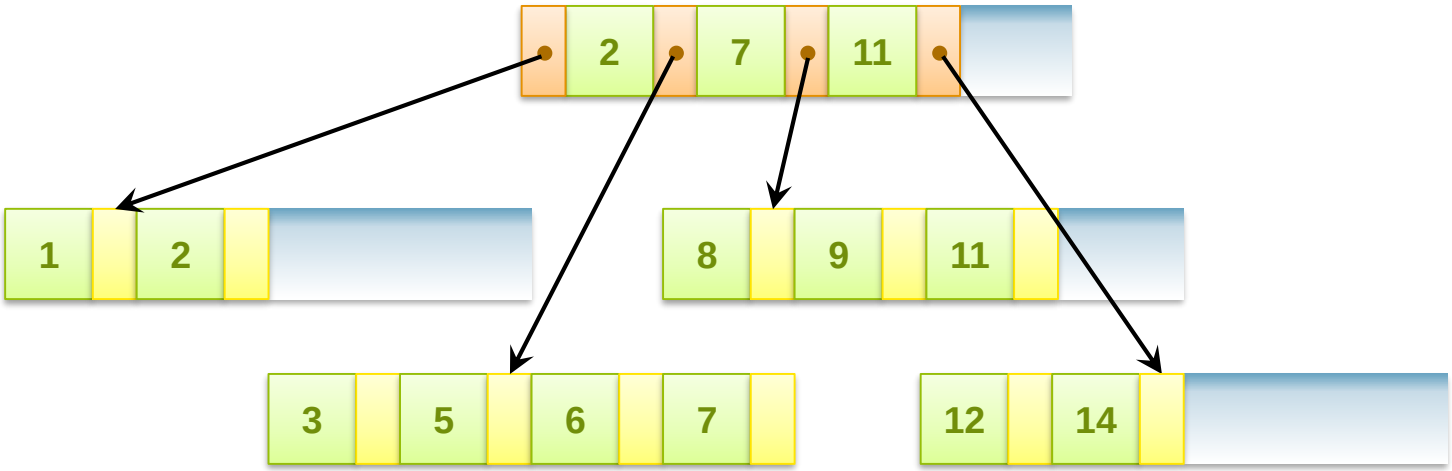
- ... am Beispiel



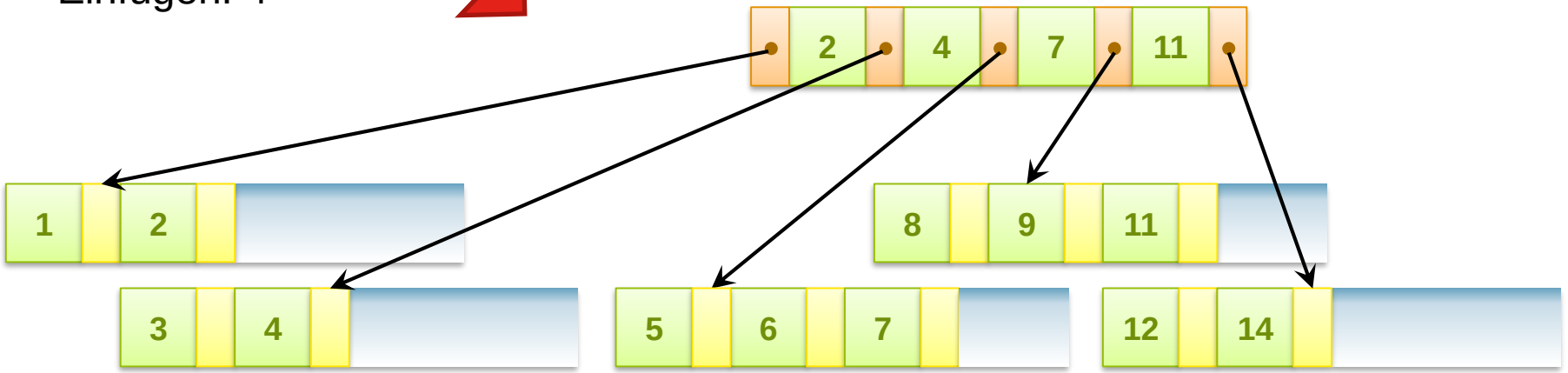
Einfügen: 5, 6



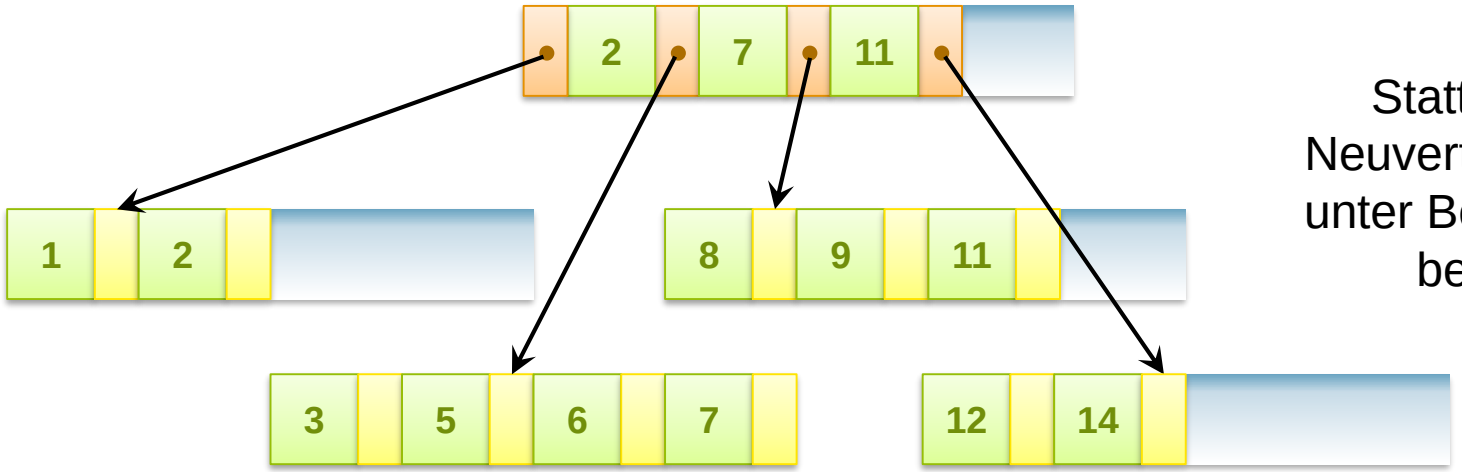
Splitt im B*-Baum



Einfügen: 4

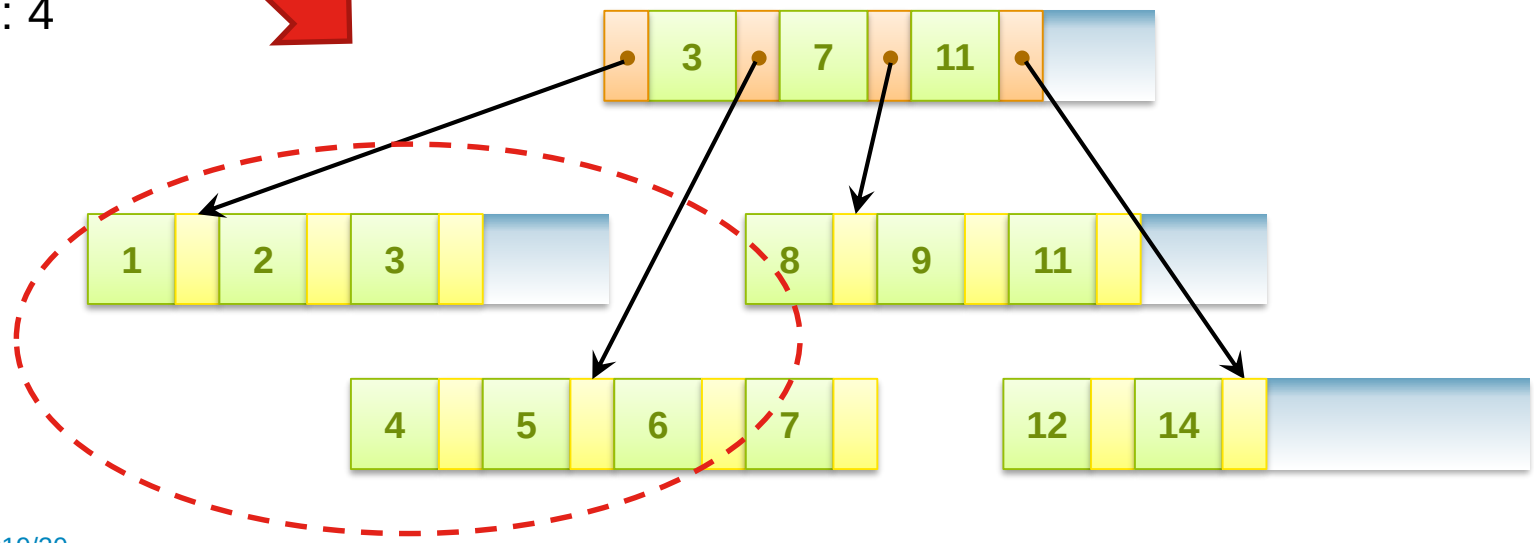


Variante beim Splitt im B*-Baum

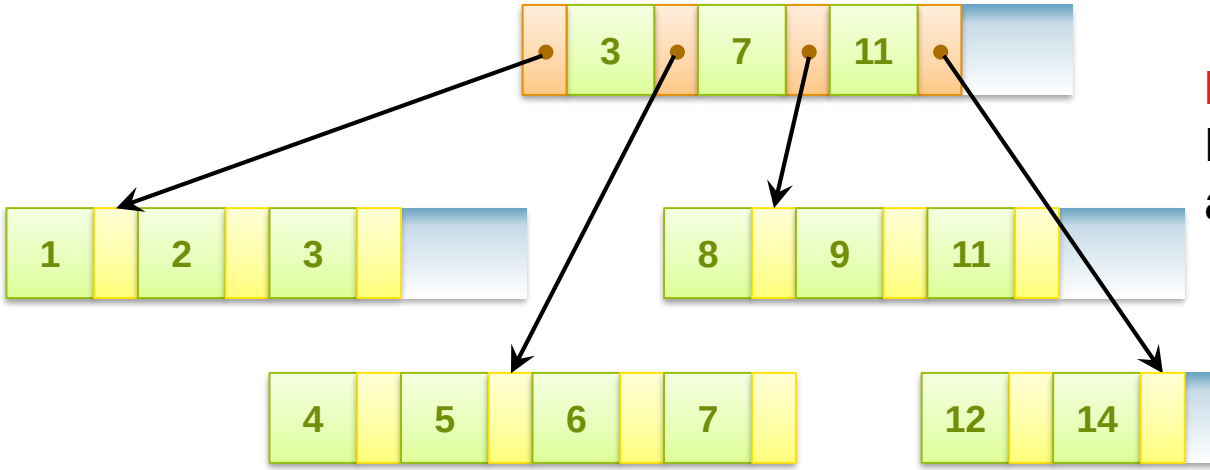


Statt Splitt bei Überlauf, Neuverteilung der Einträge unter Berücksichtigung der benachbarten Knoten

Einfügen: 4

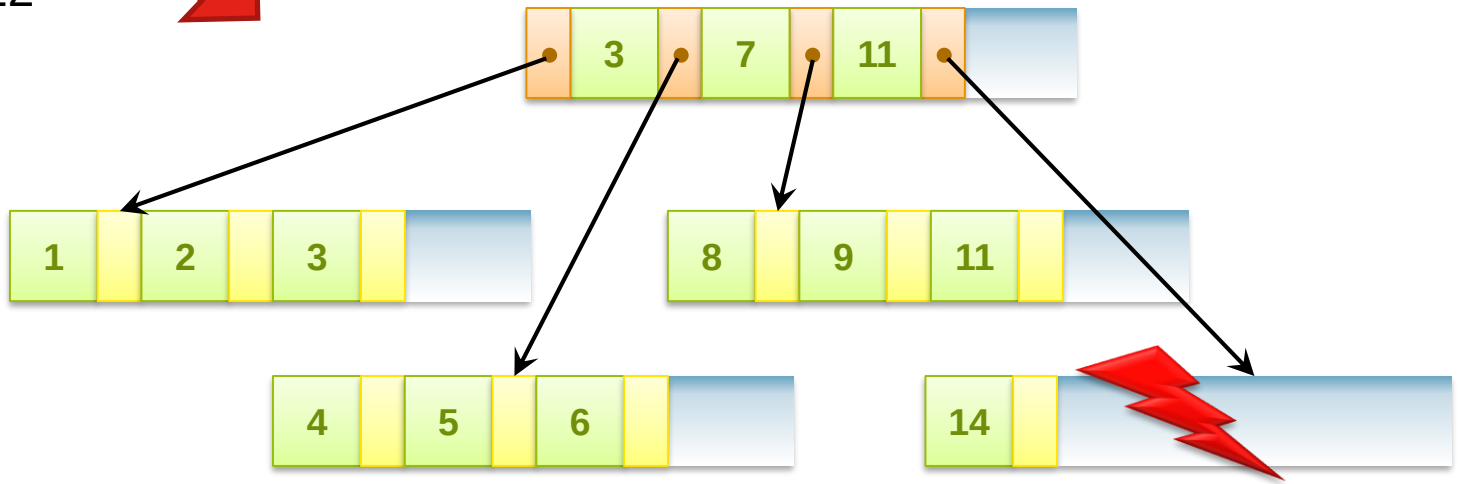


Löschen im B*-Baum



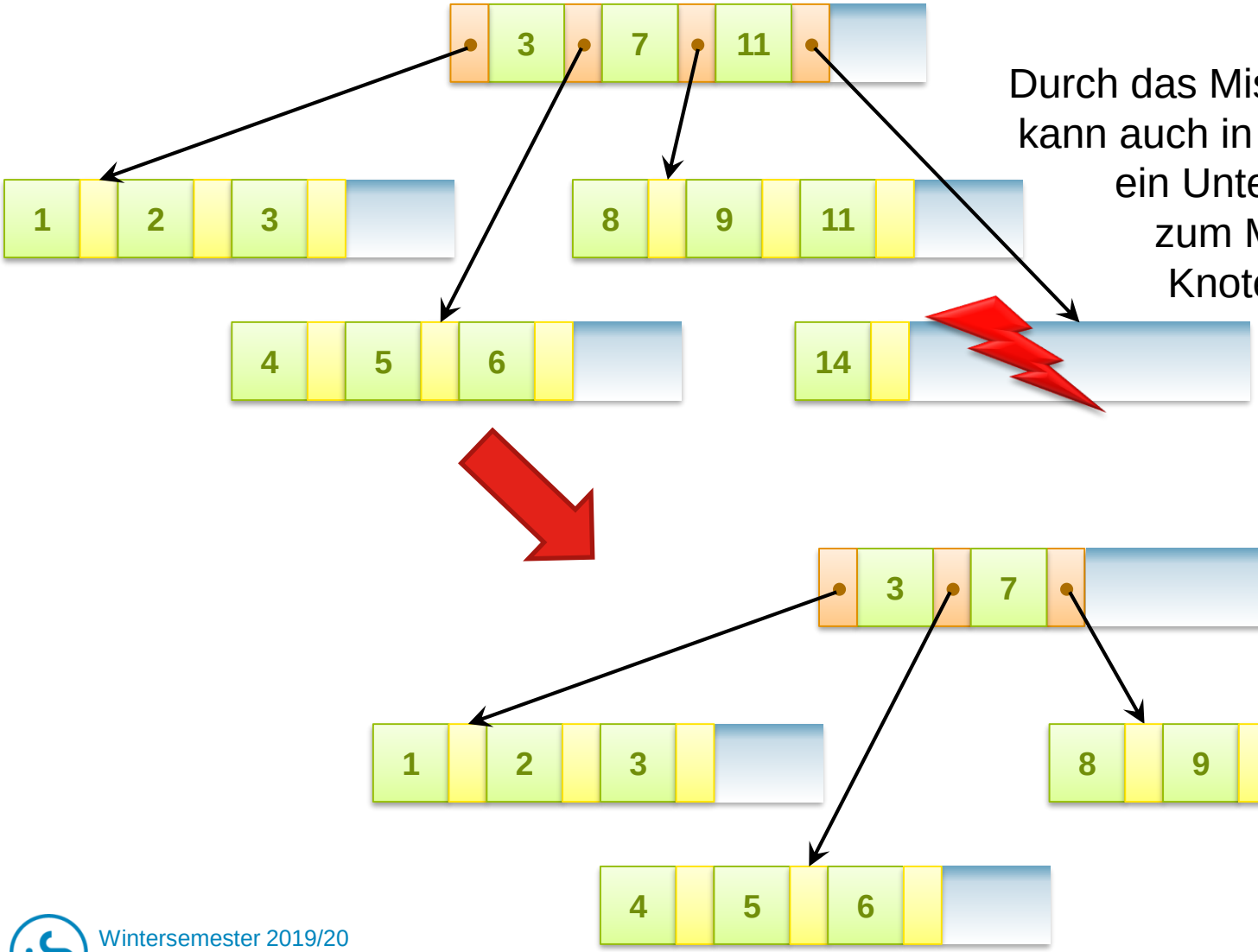
Merke:
Beim Löschen von "7" bleibt "7" als Diskriminator erhalten

Löschen: 7, 12



Unterlauf!!

Löschen im B*-Baum (2)



Durch das Mischen zweier Blätter kann auch in den inneren Knoten ein Unterlauf entstehen, der zum Mischen von inneren Knoten führt – kann sich analog zum Splitt bis zur Wurzel fortpflanzen.

Löschen von Sätzen aus B*-Baum

- **1. Such den zu löschenden Eintrag im Baum.**
- **2. Entsteht durch das Löschen ein Unterlauf? (#Einträge < k?)**
  - **NEIN**
    - Entfern den Satz aus dem Blatt.
      (Eine Aktualisierung des Diskriminators im Vaterknoten ist nicht erforderlich!)
  - **JA**
    - Prüf das Blatt zusammen mit einem Nachbarknoten:
    - Ist die Summe der Einträge in beiden Knoten größer als 2k?
    - **NEIN**
      - Misch beide Blätter zu einem Blatt zusammen.
      - Falls dabei ein Unterlauf im Vaterknoten entsteht: Misch die inneren Knoten analog.
    - **JA**
      - Teil die Sätze neu auf beide Knoten auf, so dass ein Knoten jeweils die Hälfte der Sätze aufnimmt.
      - Der Diskriminator im Vaterknoten ist entsprechend zu aktualisieren.

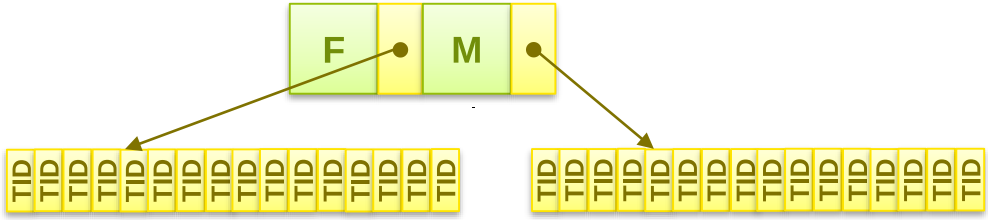
Vergleich B- und B*-Baum

B-Baum	B*-Baum
Keine Redundanz	Schlüsselwerte teilweise redundant gespeichert
Lesen aller Sätze sortiert nach Schlüsselwert nur mit Verwaltung eines Stacks der max. Tiefe = Baumhöhe h	Kette der Blattknoten liefert alle Sätze nach Schlüsselwert sortiert.
Bei Einbettung der Datensätze geringe Verzweigungszahl ("Grad" oder "fan-out"), daher größere Höhe	Hohe Verzweigung in der inneren Knoten, daher geringere Höhe
Einige wenige Sätze (die in der Wurzel) werden mit einem Blockzugriff gefunden.	Für alle Sätze müssen h Blöcke gelesen werden.
	Schlüsselwerte in den inneren Knoten müssen nicht in den Datensätzen vorkommen (Optimierung beim Löschen von Sätzen).

B- und B*-Baum: Verfeinerungen

- **Die Di in den Knoten können sein:**
  - ein Datensatz
  - eine Liste von Datensätzen
    - Alle die mit gleichem Schlüsselwert
  - eine Satzadresse
    - Verweis auf Datensatz, der nach anderen Kriterien abgespeichert ist
  - eine Liste von Satzadressen
    - U.U. sogar von Sätzen verschiedenen Typs …
- **Die Di können variabel lang sein …**
  - Sowieso
  - Wie groß ist dann k?
  - Alles wird komplizierter, aber es geht – siehe Literatur

- **Problem**
  - Am Beispiel: B-Baum auf Geschlecht bei Kundendatei mit 100.000 Sätzen resultiert in zwei Listen mit jeweils ca. 50.000 Einträgen



  - **Anfrage nach allen weiblichen Kunden erfordert 50.000 einzelne Block-Zugriffe** $\Rightarrow$ **Sequenzieller Zugriff ist um Längen schneller**
- **Folgerung**
  - B-Bäume (und auch Hashing) sinnvoll für Suchschlüssel mit hoher "**Selektivität**" (also geringem Anteil passender Sätze an allen Sätzen einer Datei)
  - Faustregel:
    - Grenztrefferrate liegt bei ca. 5%.
    - Höhere Trefferraten lohnen bereits den Aufwand für einen Indexzugriff nicht mehr.

Grundlegende Idee der Bitmap-Indexstruktur

- **Idee**
  - In die Jahre gekommen ...
    - Eingesetzt schon in den 60er Jahren in Model 204 der Computer Corporation of America
  - Legt **für jeden Schlüsselwert eine Bitliste** an.
  - Jedem Satz der Datei ist ein Bit in der Bitliste zugeordnet.
    - Dafür notwendig: beliebige, aber feste Reihenfolge der Sätze
  - Bitwert 1 heißt:
    Der Schlüssel hat im Satz den Wert, zu dem die Liste gehört;
    0 heißt:
    Er hat einen anderen Wert.

ID	Geschlecht	Wohnort	Alter
1	M	Erlangen	jung
2	M	Erlangen	mittel
3	F	Forchheim	alt
4	M	Eckental	alt
5	F	Erlangen	jung
6	F	Erlangen	jung
7	M	Bamberg	mittel
8	F	Höchstadt	mittel
9	F	Forchheim	jung
10	M	Erlangen	Jung

Geschlecht

F	M
0	1
0	1
1	0
0	1
1	0
1	0
0	1
1	0
1	0
0	1

Größe von Bitmap-Indexstrukturen

- **Indexgröße: (Anzahl der Werte) × (Anzahl der Sätze) Bits**
  - Beispiel:
    - Schlüssel Geschlecht mit zwei Werten in Datei mit 10.000 Sätzen
    - Bitmap: 2 × 10.000 Bits = 20.000 Bits = 2.500 Bytes
    - Ggf. noch TID-Liste für Reihenfolge: 4 × 10.000 Bytes = 40.000 Bytes
- **Eigenschaften**
  - Wächst mit der Anzahl der möglichen Werte
  - Besonders interessant bis zu ca. 500 verschiedenen Werten
  - Bei kleinen Wertigkeiten (z.B. Geschlecht) nur sinnvoll, wenn entsprechender Schlüssel oft in **Konjunktionen** mit anderen indizierten Attributen auftritt (z.B. Geschlecht und Wohnort)
- **Nochmal Indexgröße:**
  - Nicht so problematisch, da gerade bei höherwertigen Schlüsseln die Bitmaps sehr dünn besetzt und Kompressionsverfahren (z.B. RLE) sehr gut einsetzbar sind.

Konjunktionen auf Bitmap-Indexstrukturen

- **Hauptvorteil von Bitmap-Indexen**
  - Einfache und effiziente logische Verknüpfbarkeit
  - Beispiel: Bitmaps B1 und B2 in Konjunktion

```
for ( i=0; i < B1.length; i++ )
  B = B1[i] & B2[i];
```

F		FO		jung		
0		0		1		0
0		0		0		0
1		1		0		0
0		0		0		0
1	AND	0	AND	1	=	0
1		0		1		0
0		0		0		0
1		0		0		0
1		1		1		1
0		0		1		0

- **Beispiel "junge Frauen aus Forchheim"**
  - Selektivitätsfaktor allg.: $1/2 \times 1/5 \times 1/3 = 1/30$
  - Annahme: 10.000 Sätze mit je 200 Bytes Länge (ca. 10 Sätze pro Block bei 2KB-Blöcken)
    - Sequenzieller Zugriff: 1.000 Blöcke
    - Bitmap-Zugriff: $10.000/30 \approx 334$ Sätze und damit Blöcke (worst case)

5.3 Zugriffsoperationen B-Baum / B*-Baum / Bitmap

- **Genau die gleichen wie bei Gestreuter Speicherung!**
  (Siehe oben)

```
void KeyedRecordFile::insert ( char *RecordBuffer,
  int RecordLength, char *KeyValue );
char *KeyedRecordFile::read ( char KeyValue,
  int *RecordLength )
void KeyedRecordFile::modify-key ( char *OldKeyValue,
  char *NewKeyValue )
```

  usw.

- **D.h. entscheidend ist Zugriff über einen Schlüssel**
  - Nicht die Realisierung über Hashing oder B-Baum oder Bitmap
- **Wieder eine weitere Art von Datenunabhängigkeit,**
  - **Meist als** Datenstruktur- **oder** Speicherungsstruktur-Unabhängigkeit **bezeichnet**

5.4 Primär- und Sekundär-Organisation

- **Primär-Organisation**
  - bestimmt Speicherung der Sätze selbst:
    - Entscheidet darüber, in welchem Block ein Satz abzulegen ist
  - kann sequenziell, direkt oder über Schlüssel sein
- **Sekundär-Organisation**
  - verweist nur auf die Sätze,
    die nach beliebigen anderen Kriterien abgespeichert wurden
    (in einer Primär-Organisation)
  - ist nur möglich, wenn die Primärorganisation den Direktzugriff auf einen einzelnen Satz unterstützt
    - Satzadresse oder eindeutiger Schlüssel, "**Satzverweis**"
- **D.h. genau eine Primär-Organisation pro Satzmenge, aber mehrere Sekundär-Organisationen möglich**
  - (Invertierungen, "Indexe")

Primär- und Sekundär-Organisation (2)

- **B-Baum/B*-Baum als Sekundär-Organisation**
  - Oben bereits dargestellt:
    In Di steht anstelle eines Satzes nur ein Satzverweis.

- **Auch Gestreute Speicherung als Sekundär-Organisation einsetzbar:**
  - In den Buckets dann nur (Schlüsselwert, Satzverweis)-Paare
  - Dann allerdings mindestens zwei Blockzugriffe beim Lesen
    - Sätze dafür unabhängig speicherbar

Organisation einer Menge von Sätzen

- **Aus der Sicht des Benutzers einer Datei:**
  - Unter den ***n*** Feldern der Sätze sind:
    - ***p*** $\in$ {0, 1} Felder, die für die Primär-Organisation benutzt werden
    - ***k*** $\in$ {0, 1, ... , *n*} Felder, über denen jeweils eine Sekundär-Organisation verwaltet wird
    - $n - k - p$ Felder, über denen kein Index verwaltet wird
- **Realisierung:**
  - Eine Datei für die Sätze selbst
    - Direktzugriff ($p = 0$) oder Schlüsselzugriff ($p = 1$)
  - Für jede Sekundär-Organisation eine eigene Datei
    - Schlüsselzugriff
    - Als "Sätze" nur Satzverweise gespeichert (mit genau dem Schlüsselwert, der in dem adressierten Satz zum Index gehört – Konsistenzregel!)
  - D.h. ***k* + 1 Dateien** für eine "Menge von Sätzen"

Einfügen und Suchen von Sätzen

- **Einfügen:**
  1. In die Datei der Sätze
     - Liefert Satzverweis
  2. In jede Index-Datei
     - das Paar (Schlüsselwert, Satzverweis)
- **Suchen:**
  1. **read** in der Datei der Sätze
  2. Über Index:
     - **read** im Index, liefert Satzverweise
     - In einer Schleife über alle Treffer: **read** in der Satzdatei
  3. Über nicht "indexierte" Felder:
     - **read-first** und Schleife mit **read-next** in der Datei der Sätze (Scan)

Eindeutige Schlüssel

- **Primärschlüssel**
  - Schlüssel, bei dem jeder Wert *in höchstens einem Satz* vorkommen darf
    - Kontonummer, Kundennummer, Auftragsnummer, Bibliothekssignatur u.v.a.
  - Eindeutigkeit kann bei Benutzung einer Schlüsselzugriffs-Datei (primär oder sekundär) einfach überprüft werden – sonst nur durch Sortierung!
  - **Sekundärschlüssel**: nicht eindeutig, darf in mehreren Sätzen gleich sein
- **Deshalb in Datenverwaltungssystemen oft zwei Varianten von jeder Satzorganisation verfügbar:**
  - Mit Duplikaten (Sekundärschl.) – ohne Duplikate (Primärschl.)
  - Falls ohne Duplikate:
    - Operation **read** liefert nur noch einen Satz, nicht ein Feld von Sätzen
    - Neuer Fehlerfall bei **insert**: Schlüsselwert schon vorhanden
- **Vorsicht:**
  - Index über Primärschlüssel muss nicht die Primär-Organisation sein!
  - Analog Sekundärschlüssel – Sekundär-Organisation

Weitere Möglichkeiten

- **Domänen-orientierte Indexstruktur**
  - Schlüssel mehrerer Satzmengen zusammen in einer Struktur
  - Schneller Zugriff bei Verknüpfungen (Join, siehe unten)
- **Mehrdimensionale Indexstrukturen**
  - Sätze über zwei oder mehr Schlüssel gleichzeitig zugreifbar
  - Soll auch helfen, wenn nicht alle davon als Suchschlüssel verwendet werden (partial-match queries)
  - Quadranten-Baum, Mehrschlüssel-Hashing, k-d-Baum, UB-Baum, Gridfile, R-Baum, …
    - Letztes Kapitel der Vorlesung "Multimedia-Datenbanken"

Resultierende Schicht