

Kap. 4.4: B-Bäume  
Kap. 4.5: Skiplisten

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

14. VO

23. Mai 2006

# Motivation

„Warum soll ich heute hier bleiben?“

Heute besprechen wir Skiplisten!

„Was ist daran Besonderes?“

Einfacher und schneller als alles bisherige!

# Überblick

- Wiederholung Insert/Delete in B-Bäume

s. vorherige Vorlesung

- B<sup>+</sup>-Bäume

- Einführung von Skiplisten

- Suchen/Einfügen/Entfernen in Skiplisten

# Entfernen in B-Bäumen

- (1) Den zu entfernenden Schlüssel  $x$  suchen
- (2) Falls  $x$  in einem Blatt ist: einfach entfernen
- (3) Sonst: Suche direkten Vorgänger (dieser ist Blatt), überschreibe  $x$  mit dessen Schlüssel und entferne das Blatt.
- (4) Falls dieses Blatt  $\beta$  nun zu wenige Schlüssel besitzt, dann entweder (5) oder (6)
- (5) Reparatur A: Rotation:** Falls ein Geschwisterblatt von  $\beta$  genug Schlüssel enthält, dann wird ein Schlüssel daraus verwendet um die Größe von  $\beta$  wiederherzustellen.
- (6) Reparatur B: Verschmelzen:** Sei  $\gamma$  ein Geschwisterblatt von  $\beta$ . Wir verschmelzen  $\gamma$ ,  $\beta$  und ihren Trennschlüssel in einen neuen großen Knoten
- (7) Setze die Reparatur rekursiv fort (evtl. nach (6) nötig)

# Diskussion zum Entfernen (1)

- Warum ist Reparatur B korrekt?
- Wir wissen, dass
  - $\beta$  nur  $\lceil m/2 \rceil - 2$  Schlüssel enthält
  - beide Geschwister (bzw.  $\gamma$ ) von  $\beta$  nur  $\lceil m/2 \rceil - 1$  Schlüssel enthalten
  - damit: der neue Knoten höchstens  $m-1$  Schlüssel besitzt.

# Analyse von Delete(r,s)

Laufzeit:  $\Theta(\text{Höhe des Baums}) = \Theta(\log n)$

# Diskussion zum Entfernen (2)

- Alternative zu **Reparatur B** wäre eine erweiterte Rotation:
- Falls ein Geschwister  $\gamma'$  von einem Geschwisterknoten  $\gamma$  von  $\beta$  genügend Schlüssel besitzen würde, wäre eine doppelte Rotation denkbar: zunächst von  $\gamma'$  nach  $\gamma$ , danach von  $\gamma$  nach  $\beta$ .
- Also wäre es denkbar solange alle Nachbarn abzusuchen, bis ein Knoten gefunden wird.
- Die Zeit hierfür wäre  $O(m)$ .
- Aber: **Reparatur B** ist besser, weil Zeit:  $O(\log n)$ , und in der Praxis  $\log n < m$

# Varianten zum Einfügen/Entfernen

- Wir durchlaufen für diese Operationen den Baum einmal von oben nach unten, einmal von unten nach oben.
- Es existieren auch Alternativen bei denen beim Einfügen und Entfernen der Baum jeweils nur einmal von oben nach unten durchwandert wird:
- Hierbei werden die besuchten Knoten des Baumes gleich „auf Verdacht“ gespalten (falls sie groß sind) bzw. verschmolzen (falls sie klein sind).

# Varianten von B-Bäumen

- Alternative Realisierungen garantieren jeweils eine  $2/3$  Füllung der Knoten.
- oder erweitern den Baum durch Zeiger zwischen den Geschwistern.
- oder akkumulieren „falsche“ Balancierungen auf und reparieren diese später gemeinsam.
- Es gibt immer wieder neue gute Einfälle!
- Auch in der praktischen Umsetzung gibt es immer wieder Neues!

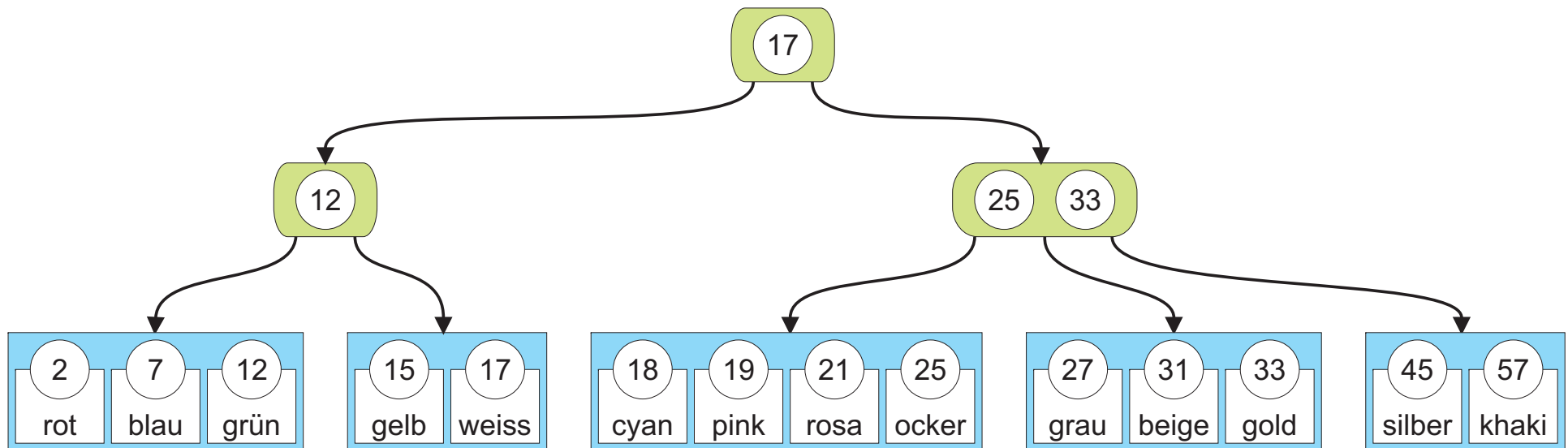
# Spezielle B-Bäume

- 2-3 Baum = B-Baum der Ordnung 3  
[Hopcroft '70]
- 2-3-4 Baum = B-Baum der Ordnung 4
- Rot-Schwarz-Baum = 2-3-4 Baum in dem große Knoten durch binäre Teilbäume simuliert werden

# B<sup>+</sup>-Bäume

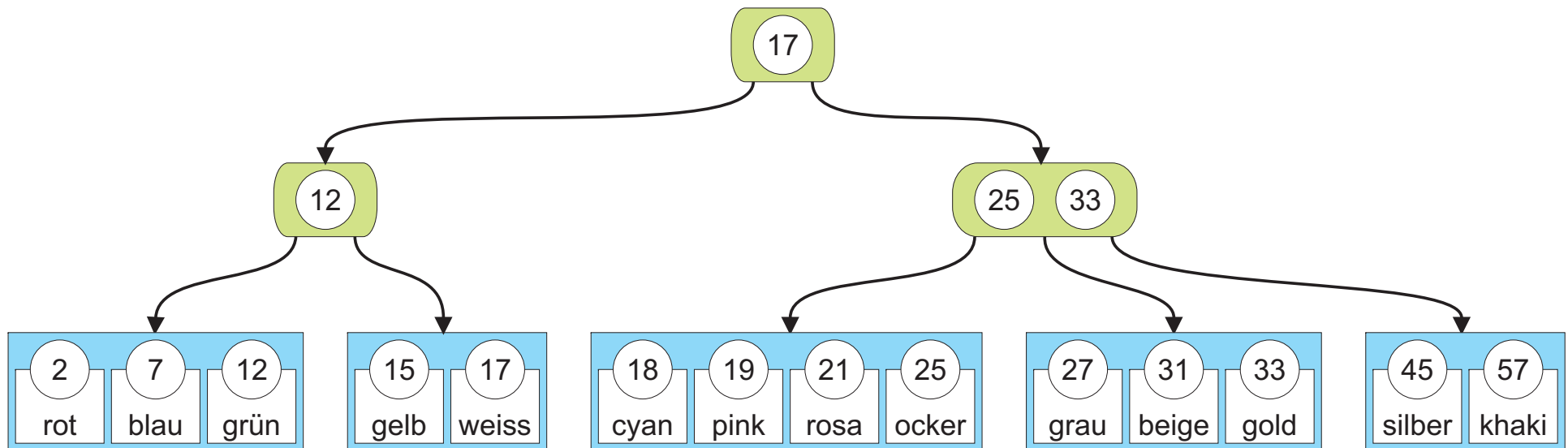
In Literatur auch: „B\*-Bäume“

- Im Wesentlichen B-Bäume bei denen die Datensätze nur in den Blättern stehen.
- Die inneren Knoten enthalten ausschliesslich Schlüssel (und Zeiger) zur Suchsteuerung.



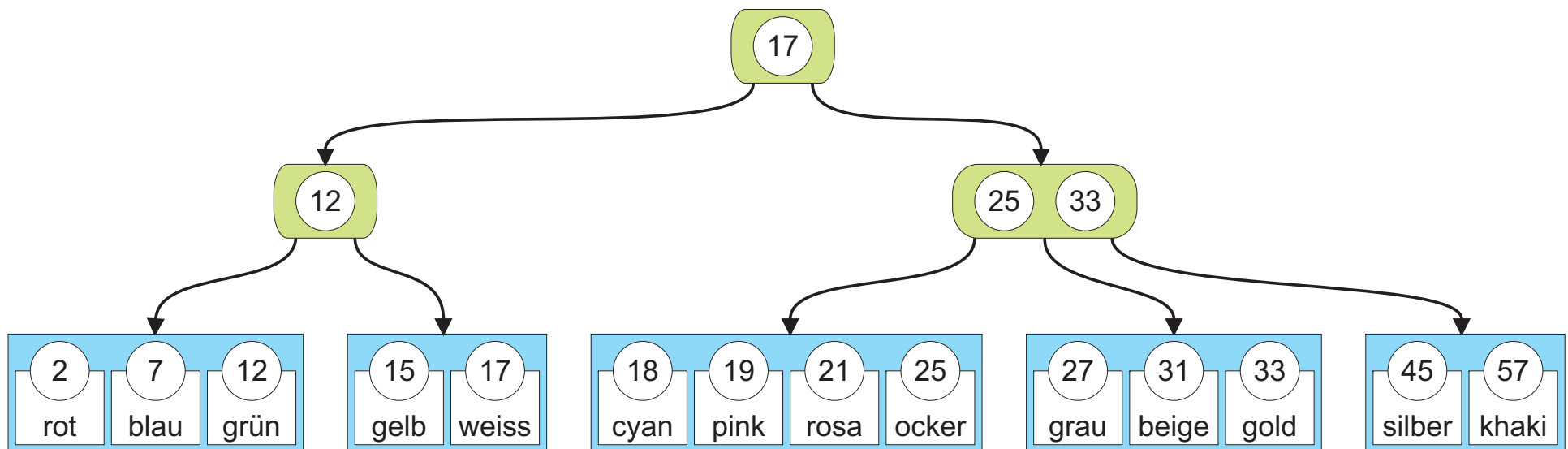
# B<sup>+</sup>-Bäume

- Vorteil: Ein innerer Knoten kann mehr Schlüssel aufnehmen.
- Dadurch wird Verzweigungsgrad erhöht und die Höhe nimmt ab.



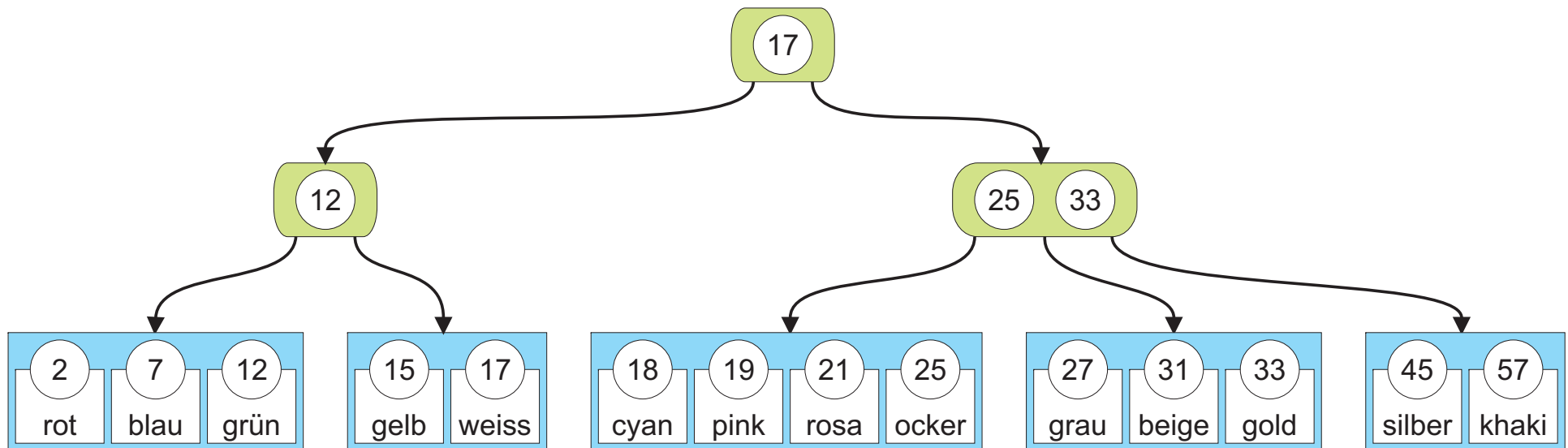
# B<sup>+</sup>-Baum der. Ordnung (m,m<sup>+</sup>)

- Die inneren Knoten besitzen zwischen  $\lceil m/2 \rceil - 1$  und  $m - 1$  viele Schlüssel.
- In jedem Blatt sind zwischen  $\lceil m^+/2 \rceil - 1$  und  $m^+ - 1$  viele Schlüssel-Daten-Paare gespeichert.
- Dies führt zu besseren Externspeicherbedingungen.



# Operationen in B<sup>+</sup>-Bäumen

- Suchen, Einfügen, Entfernen: sehr ähnlich wie in B-Bäumen.

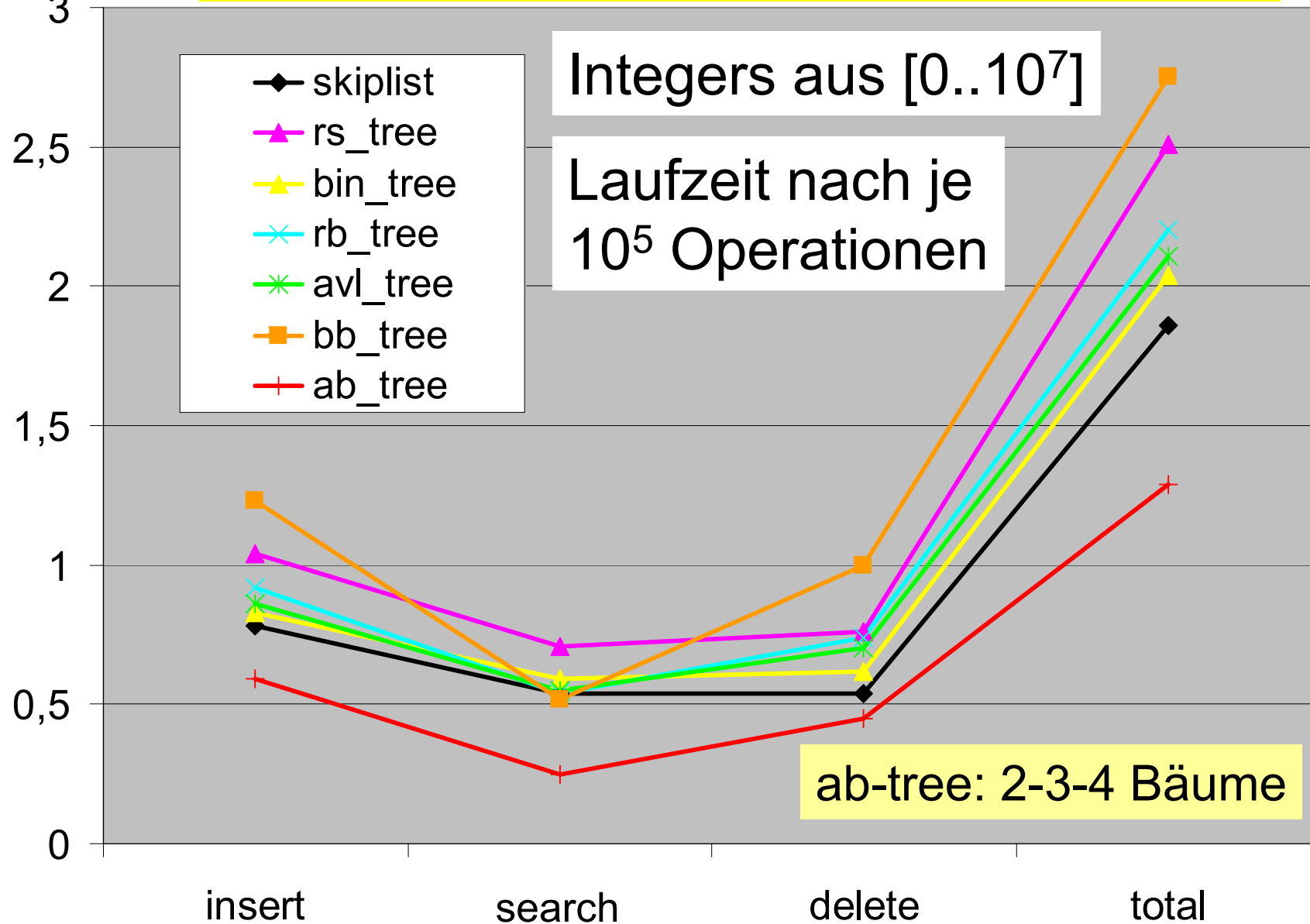


## Kap. 4.6: Dictionary- Realisierungen in der Praxis

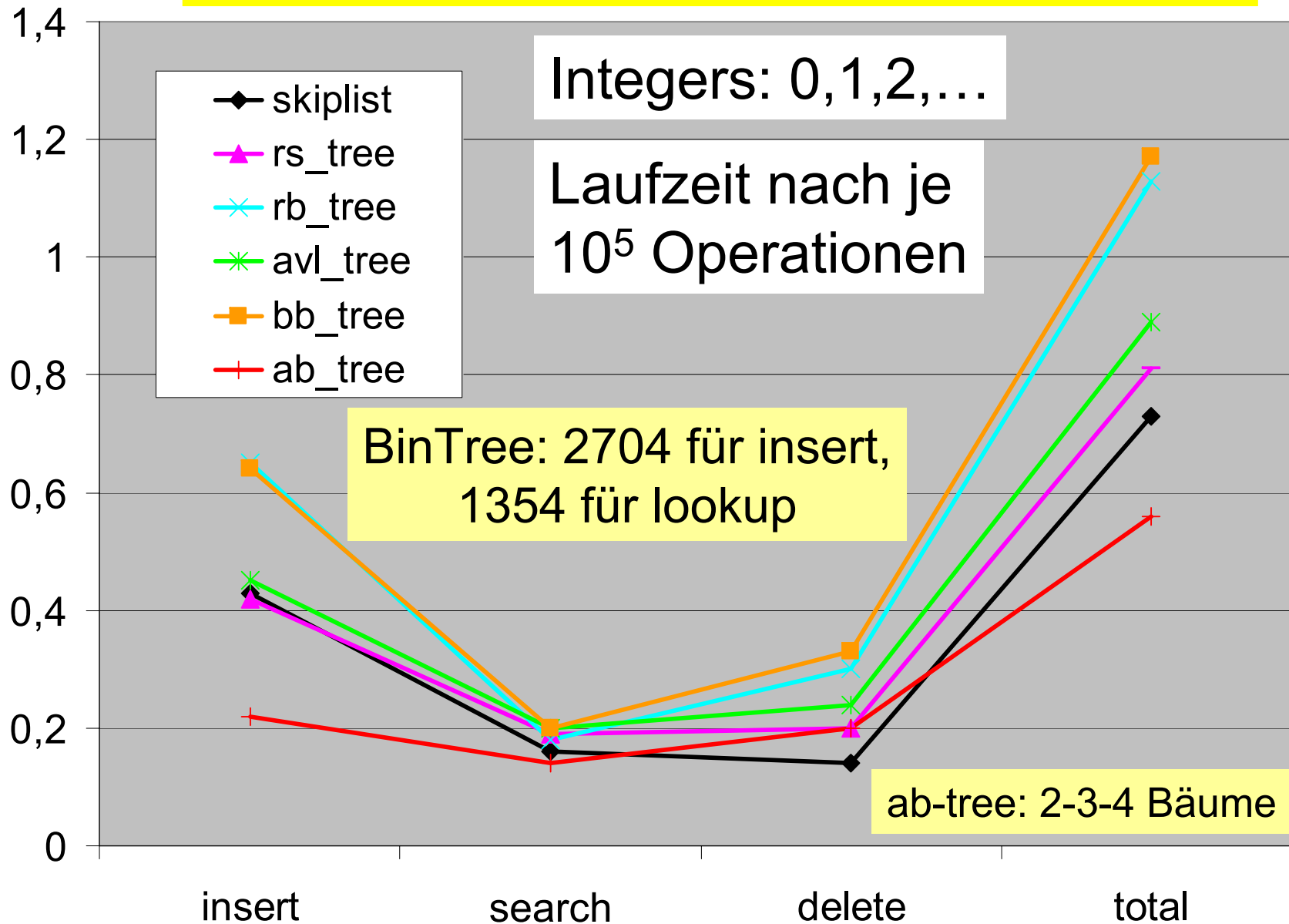
Quelle: K. Mehlhorn, S. Näher:

LEDA: A platform for combinatorial and geometric computing,  
Cambridge University Press, 1999, S. 127

# Zufällige Eingabefolge



# Sortierte Eingabefolge

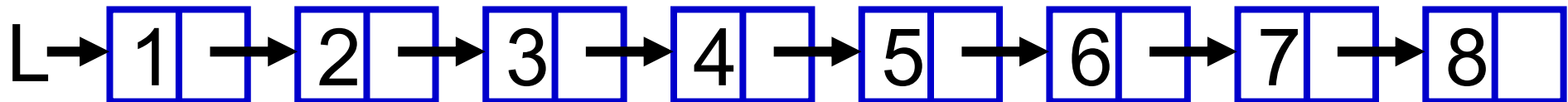


# Kap. 4.5: Skiplisten

# Einführung von Skiplisten

- **Idee:** Simulation von Binary Search auf einfach verketteten Listen
- dies geschieht durch Hinzufügen mehrerer Zeiger auf Nachfolger
- Verallgemeinerung von einfach verketteten Listen

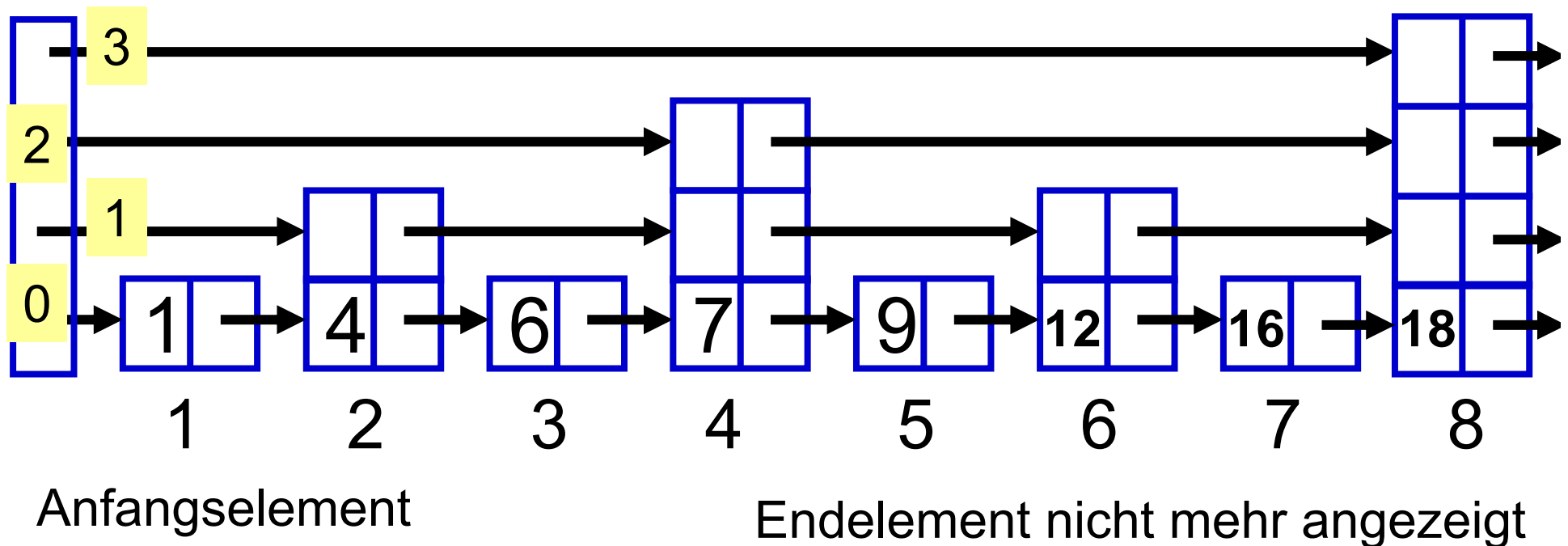
Einfach verkettete Liste:



# Perfekte Skiplisten

- **Idee:** Simulation von Binary Search auf einfach verketteten Listen

Niveau  $i$  des Zeigers



# Perfekte Skiplisten

- Elemente  $c$  der Liste besitzen Höhe  $h(c)$
- jedes Element  $c$  enthält einen Schlüssel und  $h(c)+1$  Zeiger auf nachfolgende Listenelemente (LE)
- Anfang und Ende der Liste: je ein Pseudo-Element ohne Daten, gleiche Höhe wie maximales Element in der Liste; Schlüssel des Endelements ist größer als alle anderen
- Zeiger  $i$  für  $0 \leq i \leq h(c)$  zeigt auf Element, das  $2^i$  Positionen hinter LE  $c$  steht oder auf Endelement

# Suchen in Skiplisten

Sei  $s$  der zu suchende Schlüssel

- (1) Folge dem höchsten Zeiger des Anfangselements:  
der Schlüssel sei  $s'$
- (2) Solange  $s' < s$ : folge dem Zeiger auf gleichem  
Niveau zu Nachfolgeelement  $\rightarrow$  neues  $s'$
- (3) Solange  $s' > s$ : Reduziere das Niveau um 1: folge  
diesem Zeiger zu Nachfolger  $\rightarrow$  neues  $s'$
- (4) Gehe zu (2)
- (5) Falls  $s == s'$ : STOP:  $s$  gefunden
- (6) Sonst: wir sind auf Niveau 0 angekommen und  
haben  $s$  nicht gefunden

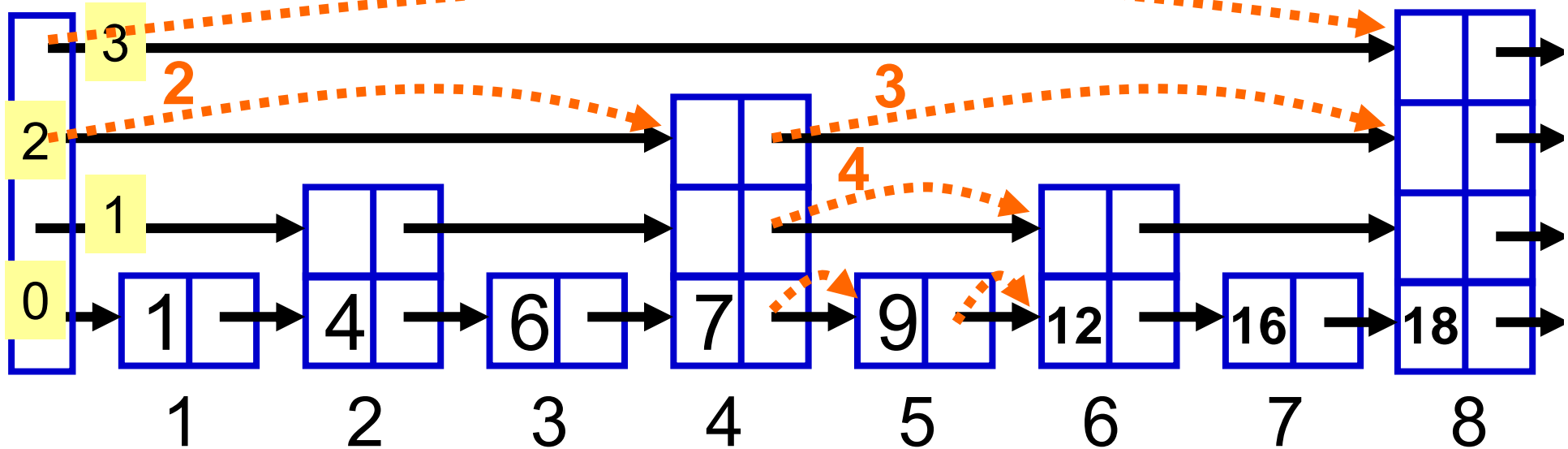
# Suche in Skiplisten

Suche nach 12:

Niveau i des Zeigers

Vergleiche

1



Anfangselement

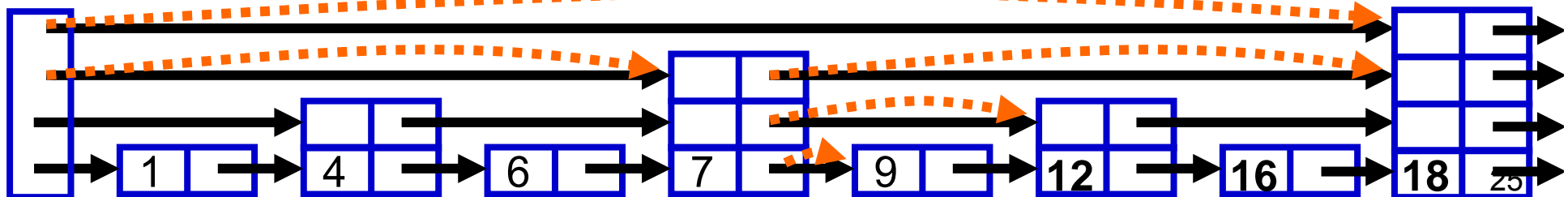
Endelement nicht mehr angezeigt

# Datenstruktur einer Skipliste

- `c.height`: Höhe von Listenelement `c`
- `c.next[i]`: Nachfolger auf Niveau `i`
- `L.head`: Anfangselement der Liste
- `L.height`: Höhe des Anfangselements

# SEARCH(L,x)

- (1)  $p = L.head$
- (2) **for**  $i := L.height$  to 0 **do** {
- (3)     **while**  $p.next[i].key < x$  **do**
- (4)          $p := p.next[i]$
- (5)     }
- (6)  $p := p.next[0]$
- (7) **if**  $p.key == x$  **then**
- (8)     return  $p$
- (9) **else** return nil



# Größe der Perfekten Skipliste

Sei  $n=2^l$ . Die Höhe ist dann also  $l$ .

**Lemma:** Die Anzahl der Zeiger in einer perfekten Skipliste ist  $2n + \log n + 1 = O(n)$ .

**Denn:** Die Anzahl aller Zeiger, die nicht von Anfangselement ausgehen:

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = \sum_{i=0}^l \frac{n}{2^i} = 2n$$

Die Anzahl aller Zeiger, die vom Anfangselement ausgehen:  $l+1=\log n+1$

# Analyse der Suchzeit

**Beobachtung:** Auf jedem Niveau gibt es höchstens zwei Vergleiche.

**Denn:** Entweder der Schlüssel ist zu groß, dann wird in ein kleineres Niveau abgestiegen. (Es wird nie wieder aufgestiegen.)

Oder der Schlüssel ist zu klein, dann wird ein Vergleich ausgeführt. Aber wir wissen: nach zwei Vergleichen auf dem gleichen Niveau sind wir wieder dort, wo wir schon auf höherem Niveau verglichen haben.

**Lemma:** Die Suchzeit in einer perfekten Skipliste ist durch  $O(\log n)$  beschränkt.

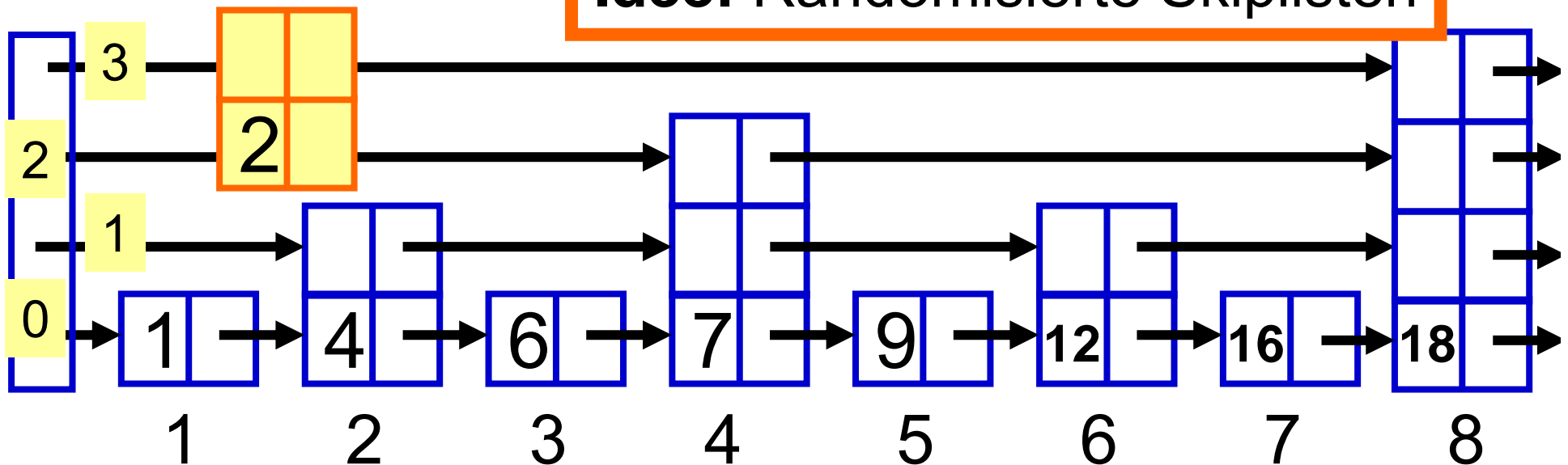
# Einfügen und Entfernen in Perfekten Skiplisten

Einfügen von 2:

Niveau i des Zeigers

Hierzu wäre eine komplette Reorganisation mit Laufzeit  $\Omega(n)$  notwendig!!!

**Idee:** Randomisierte Skiplisten

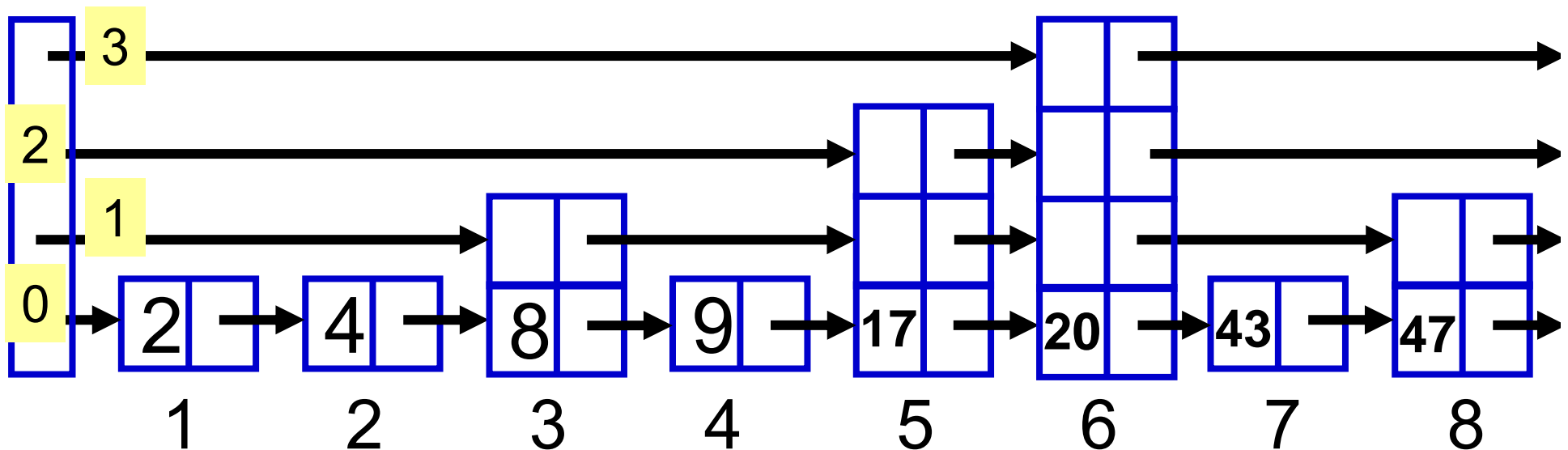


# Randomisierte Skiplisten

- **Idee:** Die Verteilung der Höhen in der Skipliste entspricht ungefähr der Verteilung in einer perfekten Skipliste.

Wie erreicht man diese Verteilung?

Niveau  $i$  des Zeigers



# Randomisierte Skiplisten

- Elemente  $c$  der Liste besitzen Höhe  $h(c)$
- Diese Höhe wird zufällig bestimmt, so dass die Verteilung derjenigen einer perfekten Skipliste entspricht.
- Jedes Element  $c$  enthält einen Schlüssel und  $h(c)+1$  Zeiger auf nachfolgende Listenelemente
- Anfang und Ende der Liste: je ein Pseudo-Element ohne Daten, gleiche Höhe wie maximales Element in der Liste; Schlüssel des Endelements ist größer als alle anderen

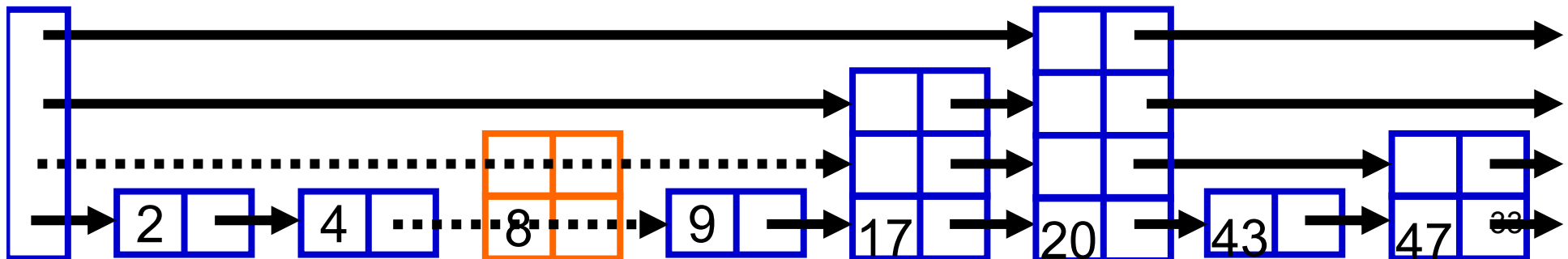
# Einfügen in randomisierte Skipliste

- (1) Suche die Position, an der eingefügt werden soll
- (2) Füge dort einen neuen Container ein, und bestimme dessen Höhe durch folgendes Zufallsexperiment:
  - (3) Setze die Höhe = 0.
  - (4) Wiederhole:
    - (5) Münzwurf: Falls „Zahl“, dann:
      - (6) Höhe=Höhe+1
    - (7) Solange bis „Kopf“ erscheint.
  - (8) Aufspalten aller Zeiger die „durch“ neuen Container laufen



# INSERT(L,x)

- (1)  $p = L.head$
- (2) **for**  $i := L.height$  to 0 **do** {
- (3)     **while**  $p.next[i].key < x$  **do**
- (4)          $p := p.next[i]$
- (5)      $update[i] = p$
- (6)     }
- (7)  $p := p.next[0]$
- (8) **if**  $p.key == x$  **then** // Schlüssel x schon da
- (9) **else**



# INSERT(L,x) ff.

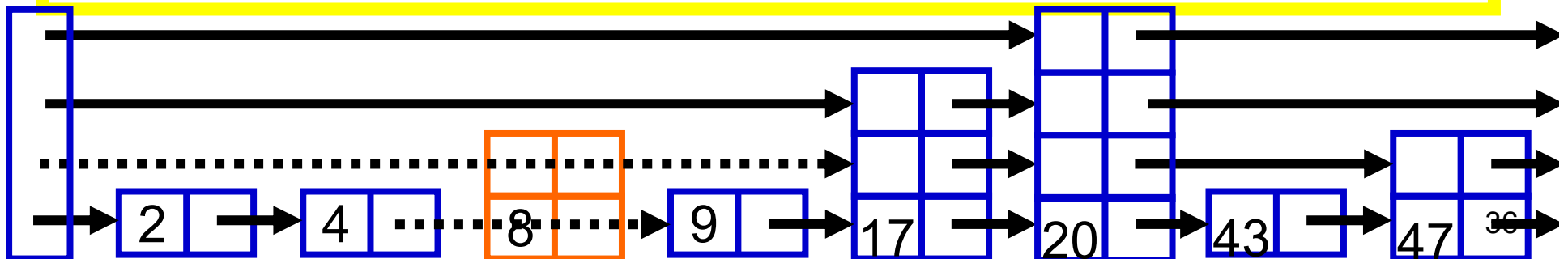
```
(9) else
(10) newheight:=randomheight()
(11) if newheight > L.height then {
(12)   for i:=L.height+1 to newheight do {
(13)     update[i]:=L.head }
(14)   L.height:=newheight
(15) }
(16) p:=newContainer(x,newheight)
(17) for i:=0 to newheight do
(18)   p.next[i]:=update[i].next[i]
(19)   update[i].next[i]:=p
(20)} }
```

# Entfernen aus randomisierter Skipliste

- analog wie Einfügen

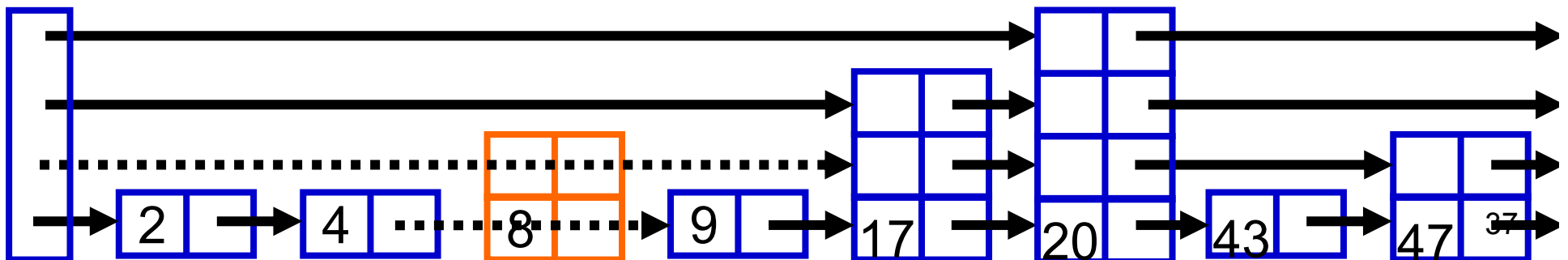
# DELETE(L,x)

- (1)  $p = L.head$
- (2) **for**  $i := L.height$  to 0 **do** {
- (3)     **while**  $p.next[i].key < x$  **do**
- (4)          $p := p.next[i]$
- (5)      $update[i] = p$
- (6) }
- (7)  $p := p.next[0]$
- (8) **if**  $p.key == x$  **then**



# DELETE(L,x) ff.

```
(8) if p.key == x then
(9)   for i:=0 to p.height do
(10)    update[i].next[i]:=p.next[i]
(11)   while L.height ≥ 1 and
        L.head.next[L.height].key == ∞ do
(12)    L.height := L.height-1
(13) }
```



# Eigenschaften randomisierter Skiplisten

**Beobachtung:** Skiplisten besitzen kein Gedächtnis, d.h. Elemente, die eingefügt und wieder entfernt wurden haben keinen Einfluss auf das Aussehen der aktuellen Skipliste.

Dies ist bei den anderen Datenstrukturen, wie z.B. den binären Suchbäumen anders.

# Analyse randomisierter Skiplisten

**Beobachtung:** Wir können keine Worst Case Schranke garantieren, denn theoretisch ist es möglich, dass ein Listenelement eine beliebig hohe Höhe erhält.  
Dies ist jedoch „unwahrscheinlich“.

**Lösung:** Wir analysieren die Erwartungswerte und zeigen, dass große Abweichungen von diesen Werten extrem unwahrscheinlich sind.

nächstes Mal 😊