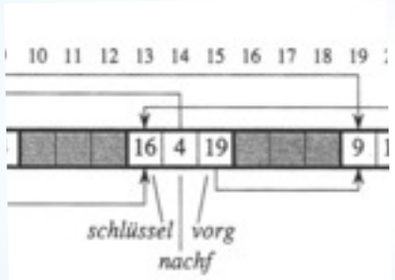
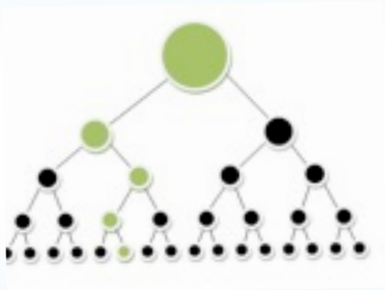


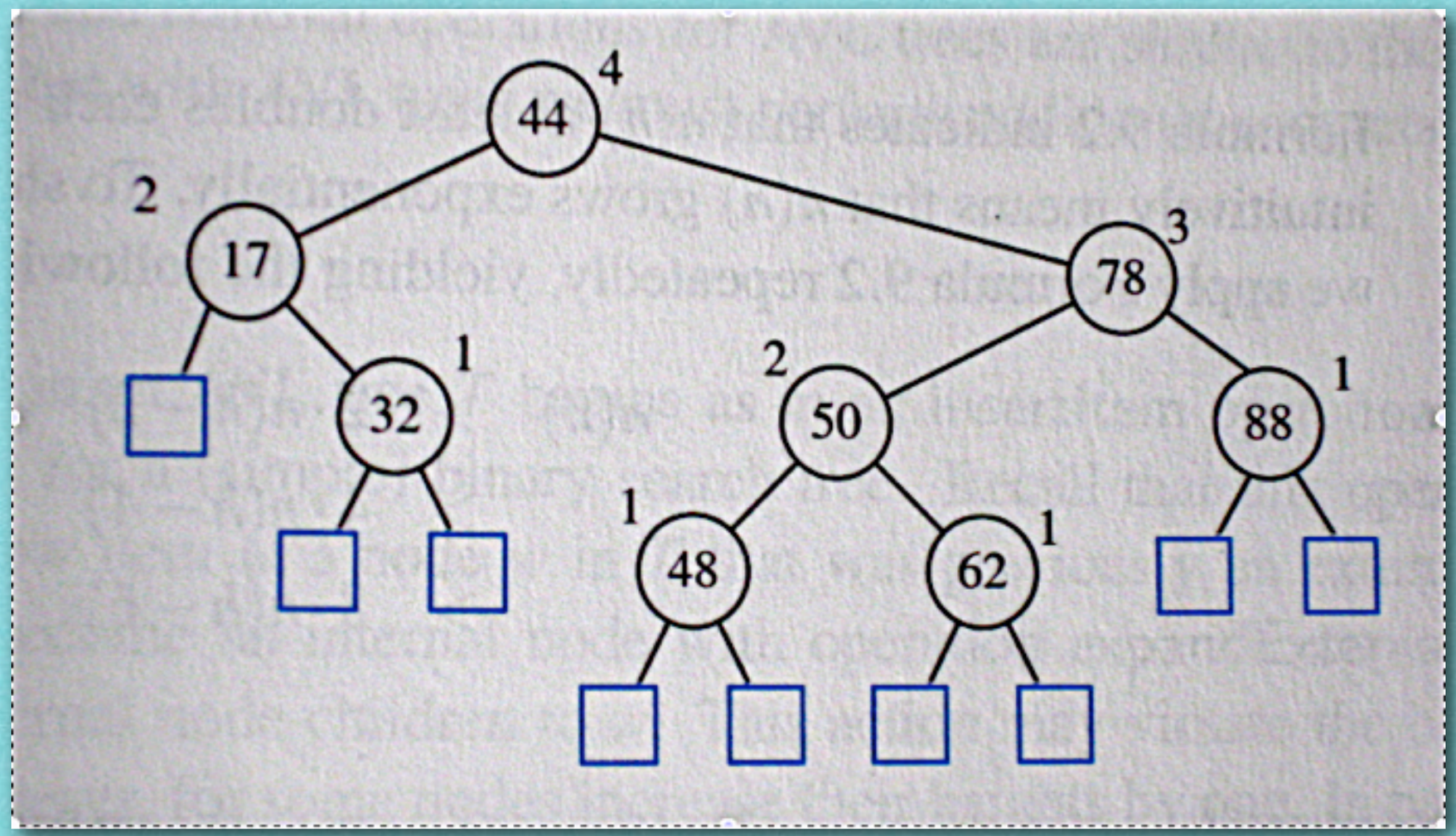


Kapitel 4.8-4.11: Andere dynamische Datenstrukturen

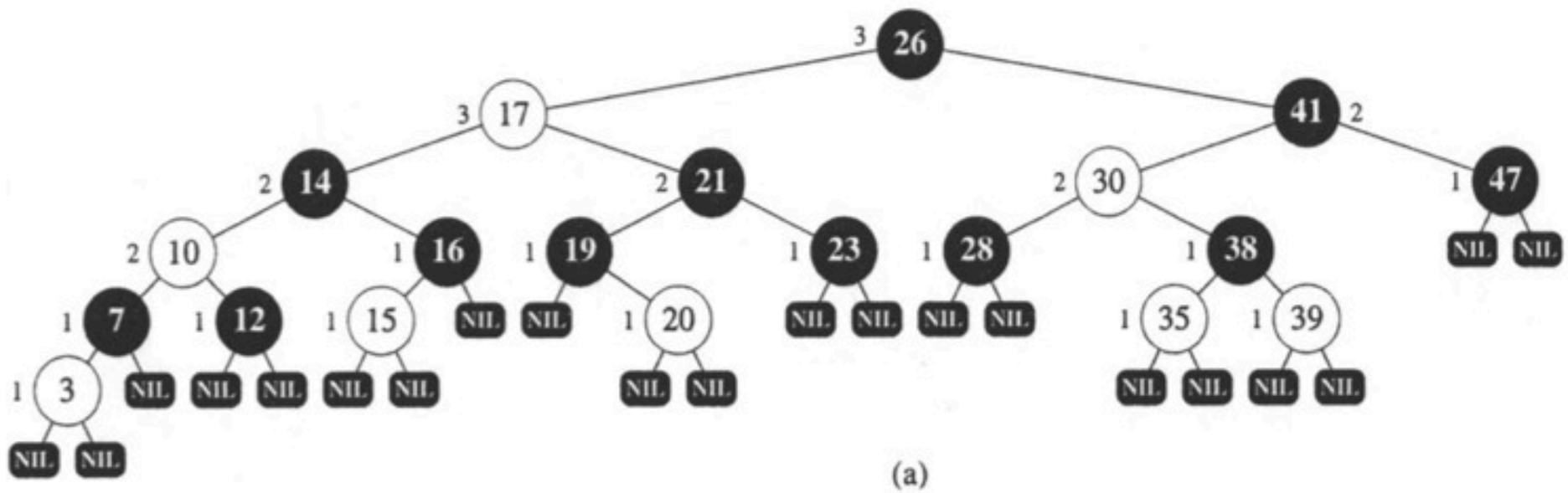
Algorithmen und Datenstrukturen
WS 2012/13

Prof. Dr. Sándor Fekete

4.6 AVL-Bäume



4.8 Rot-Schwarz-Bäume



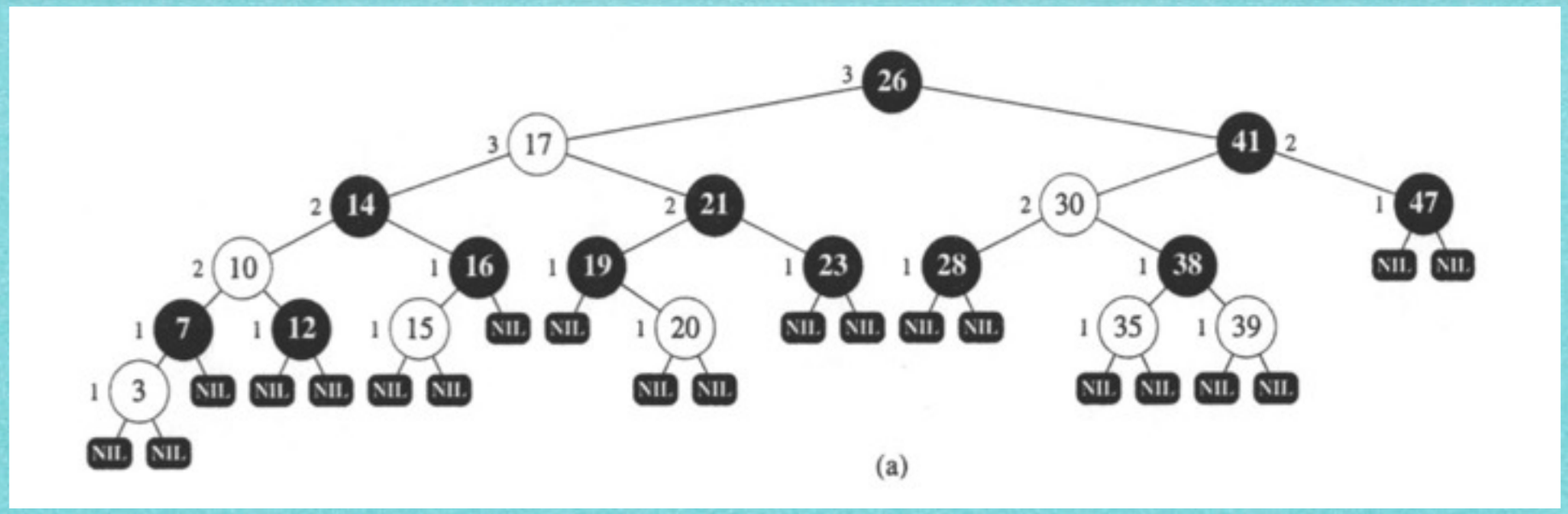
(a)

Idee: Verwende "Farben", um den Baum vertikal zu strukturieren und zu balancieren!

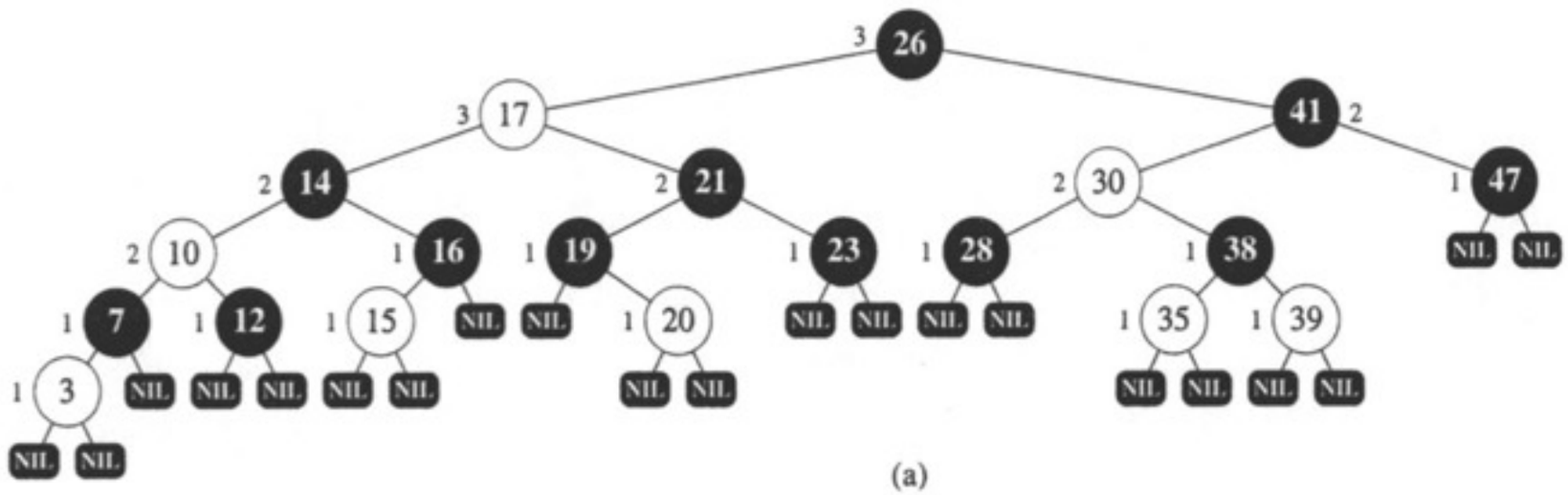
4.8 Rot-Schwarz-Bäume

<u>Definition 4.12 (Rot-Schwarz-Baum)</u>
Ein binärer Suchbaum heißt Rot-Schwarz-Baum, wenn er folgende Eigenschaften erfüllt:

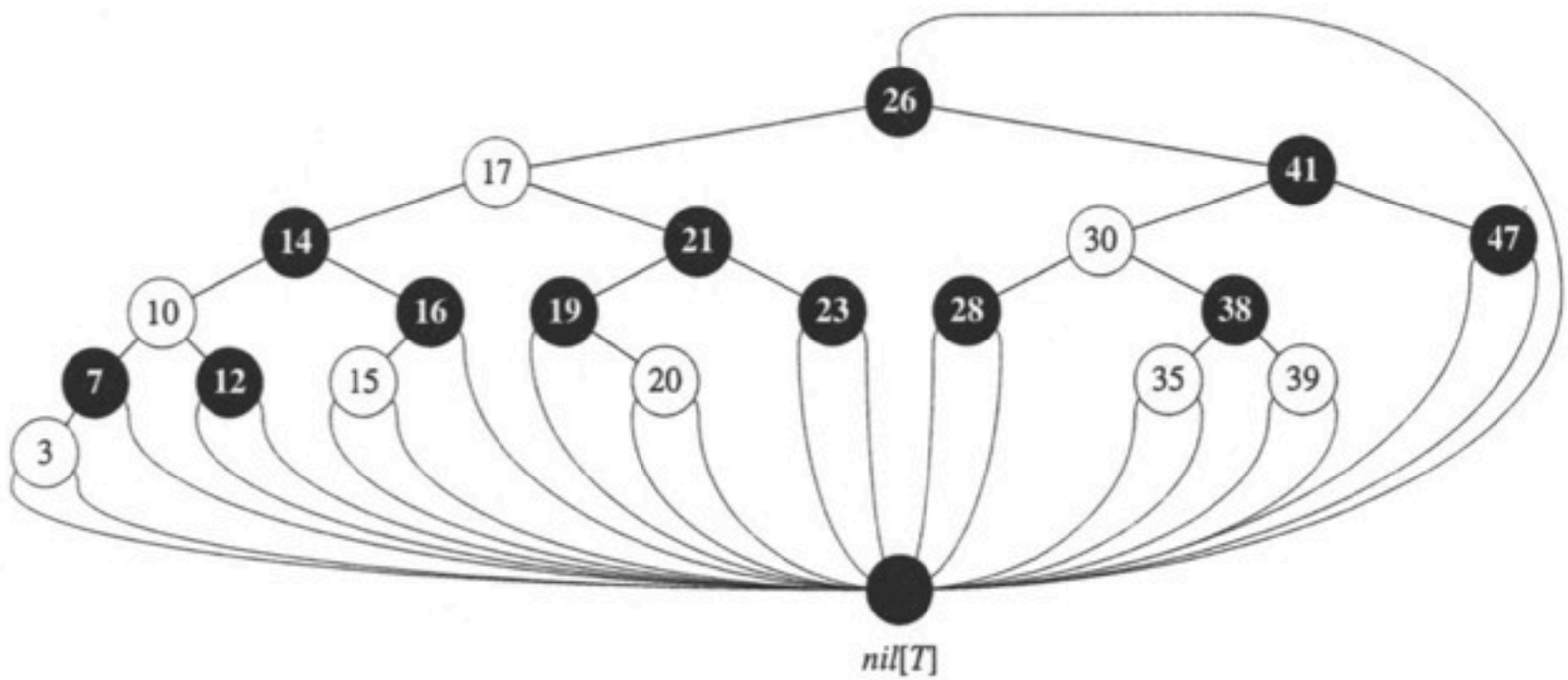
1. ***Jeder Knoten ist rot oder schwarz.***
2. ***Die Wurzel ist schwarz.***
3. ***Jedes Blatt (NIL) ist schwarz.***
4. ***Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.***
5. ***Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten.***



4.8 Rot-Schwarz-Bäume

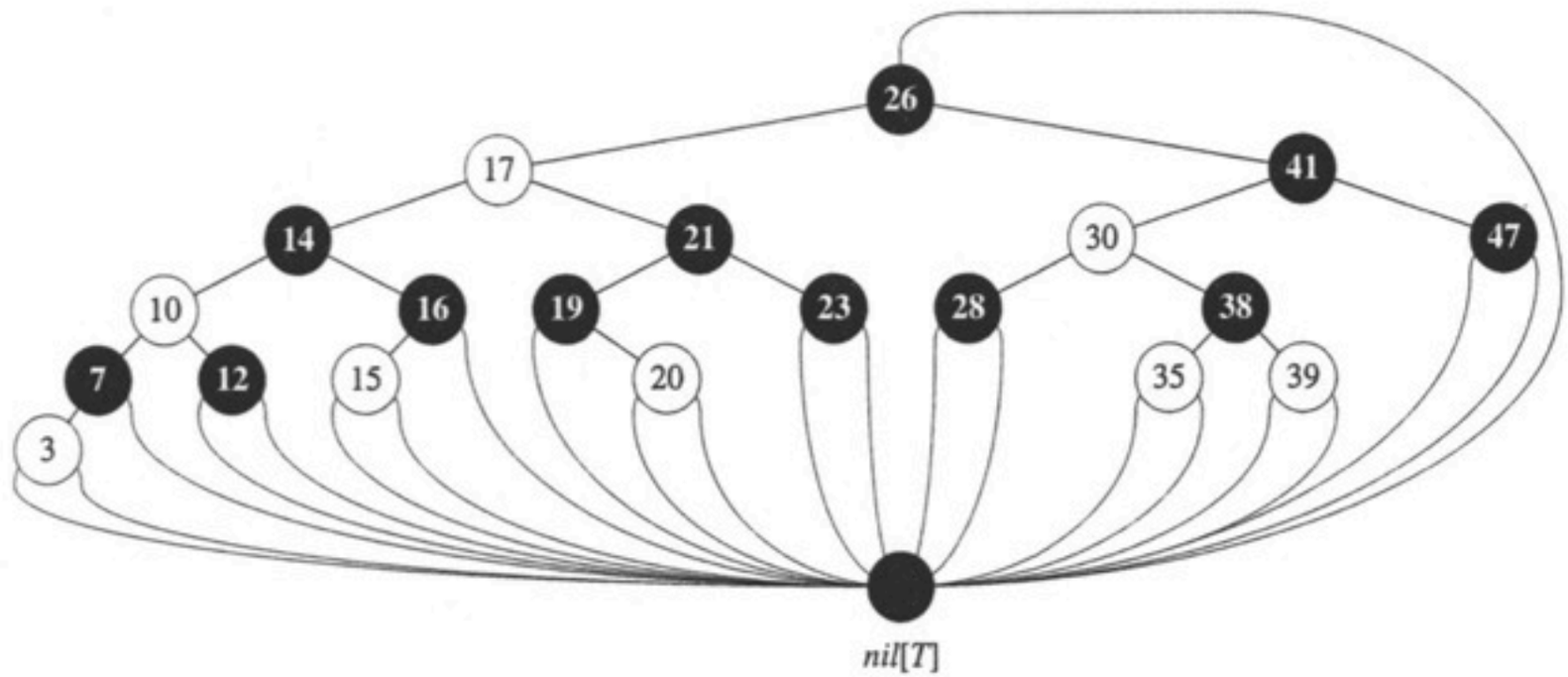


(a)

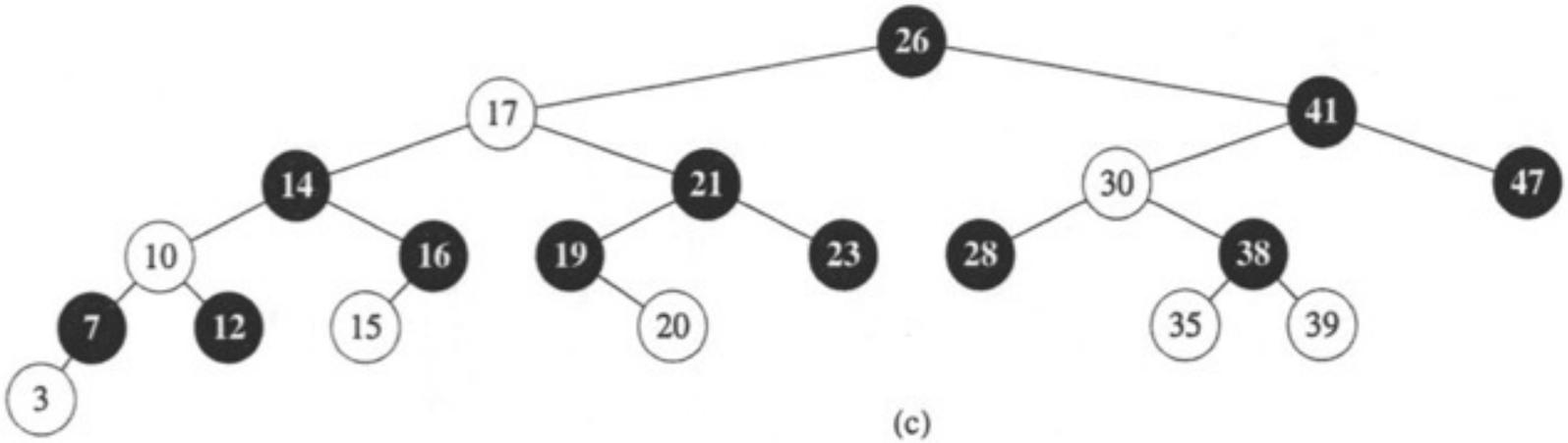


(b)

4.8 Rot-Schwarz-Bäume

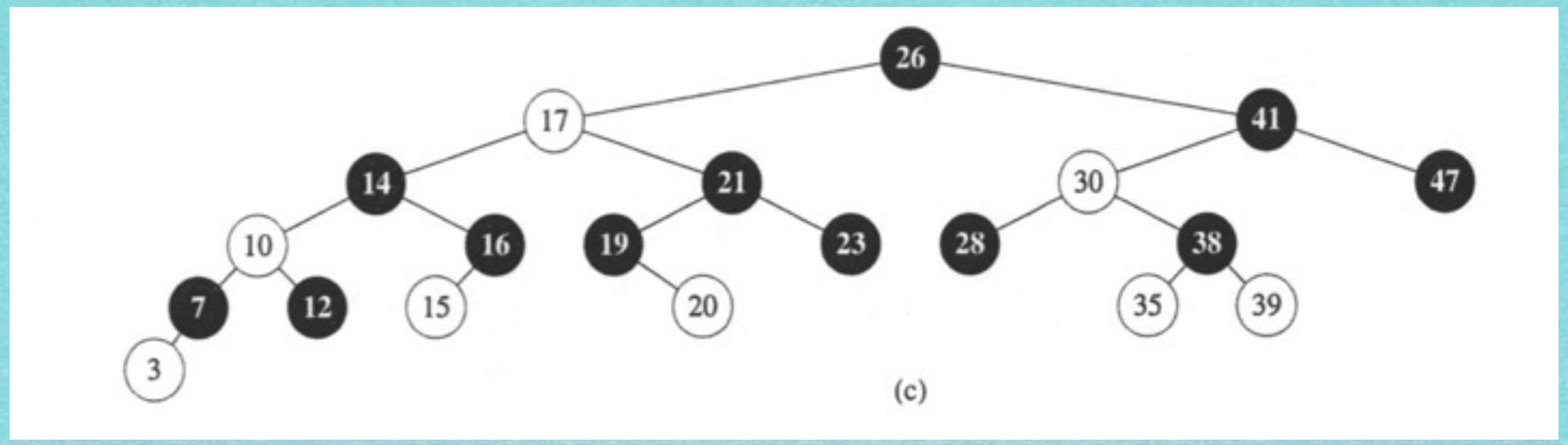


(b)



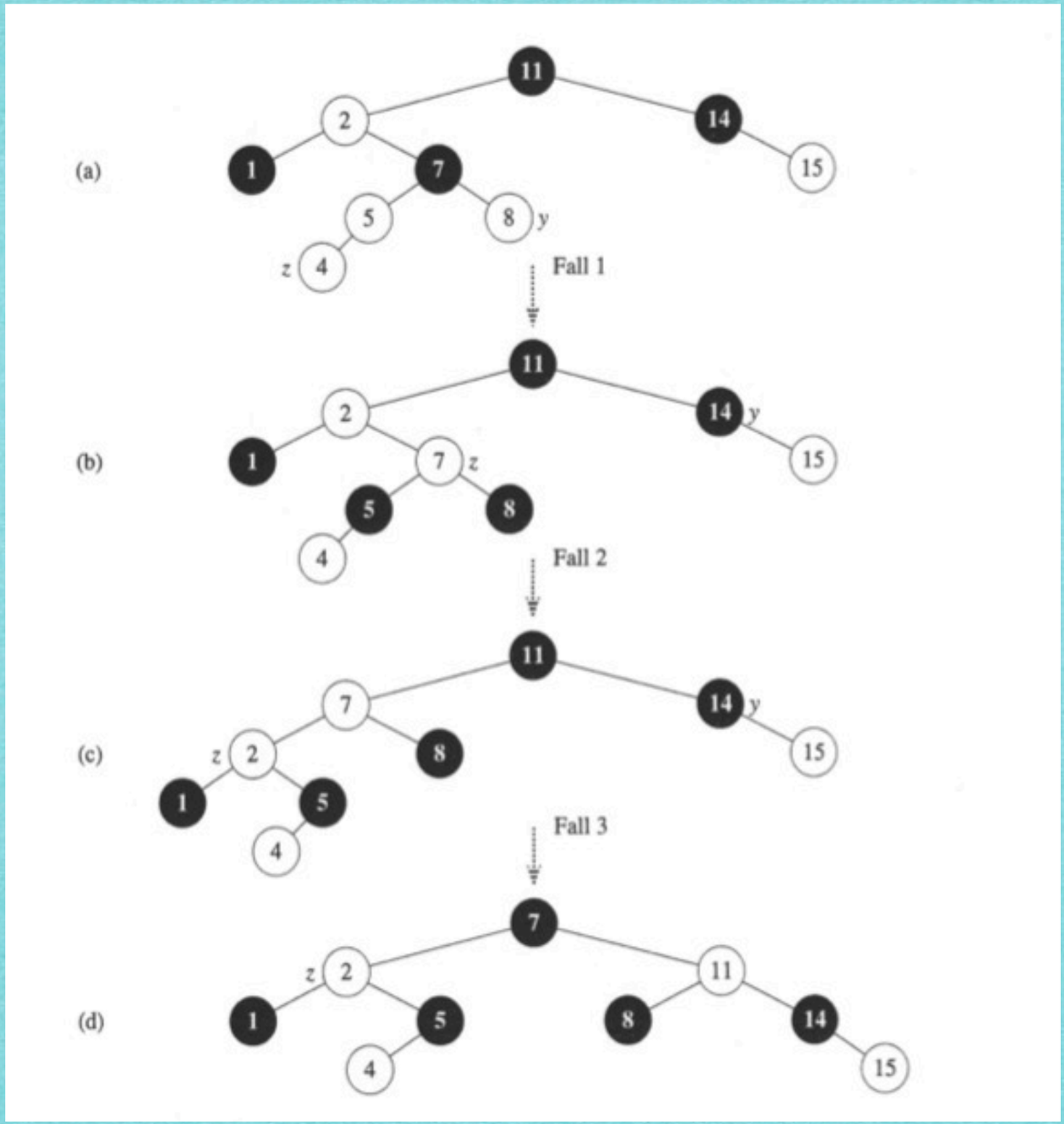
(c)

4.8.1 Rot-Schwarz-Bäume

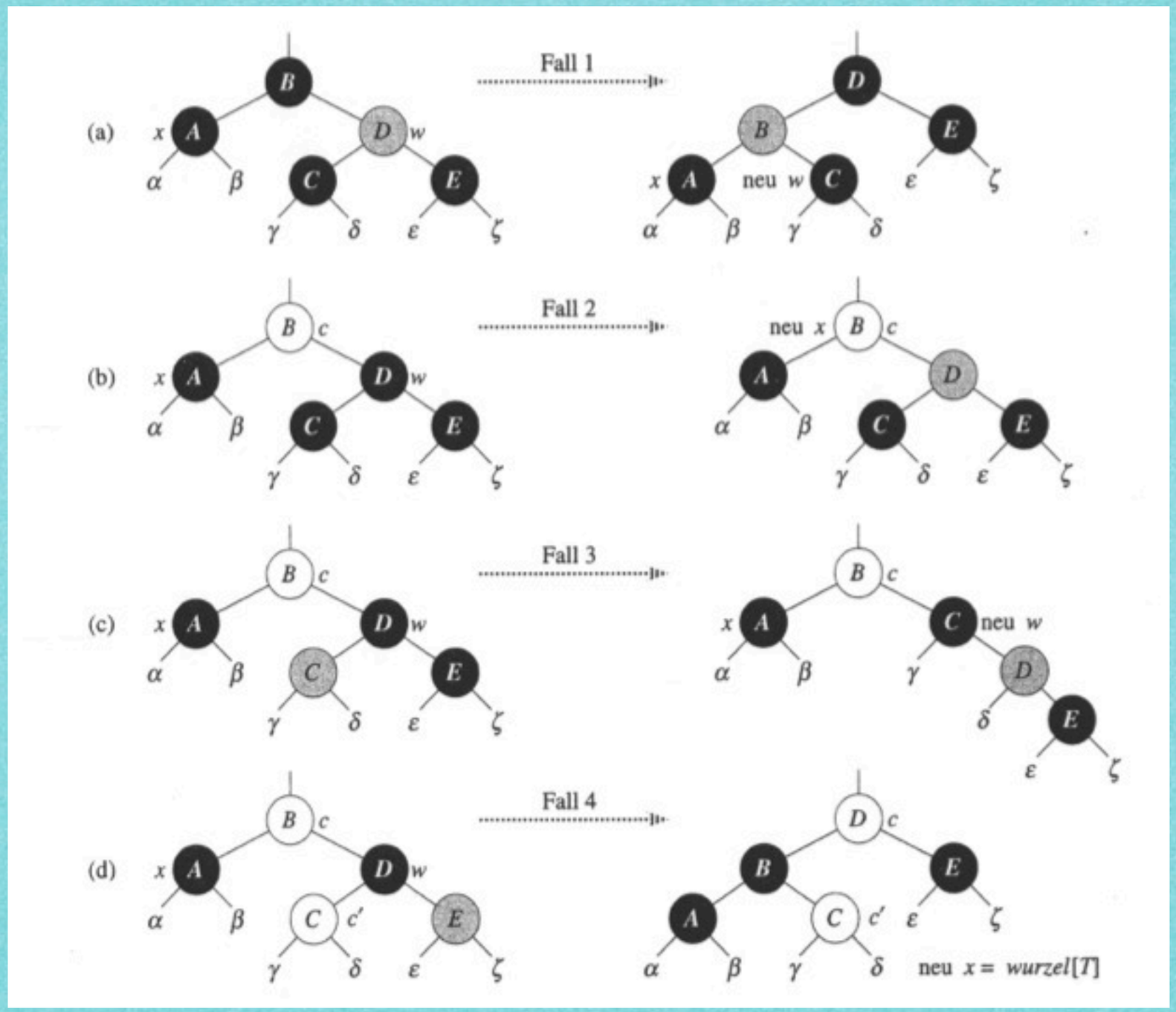


<u>Satz 4.13</u>
Ein Rot-Schwarz-Baum mit n Knoten hat Höhe O(log n).

4.8 Rot-Schwarz-Bäume: Einfügen

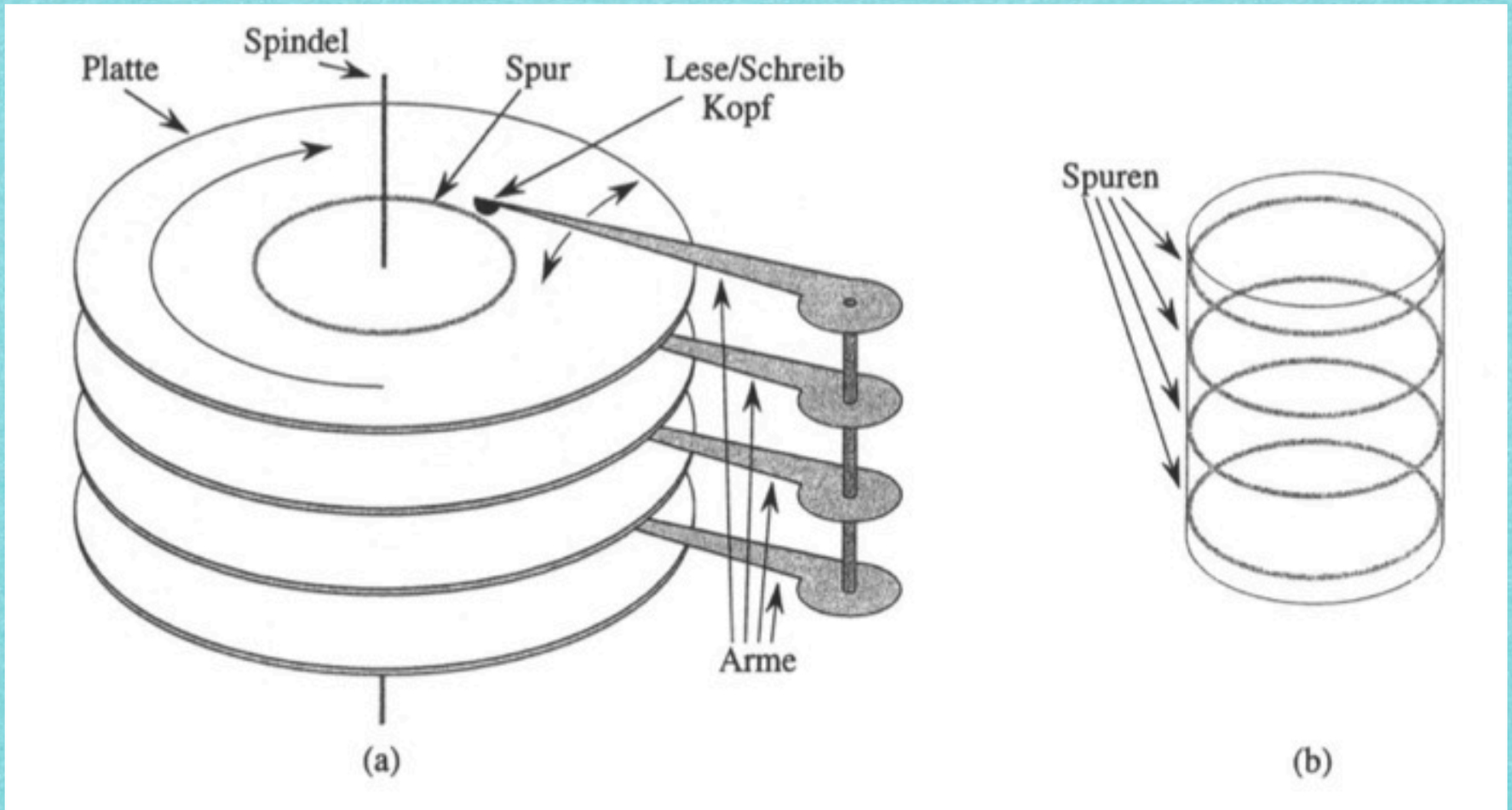


4.8 Rot-Schwarz-Bäume: Löschen



<u>Satz 4.14</u>
Ein Rot-Schwarz-Baum benötigt O(log n) für dynamische Operationen auf Datenmengen mit n Objekten - d.h. genauso lange wie ein AVL-Baum.

4.9 B-Bäume



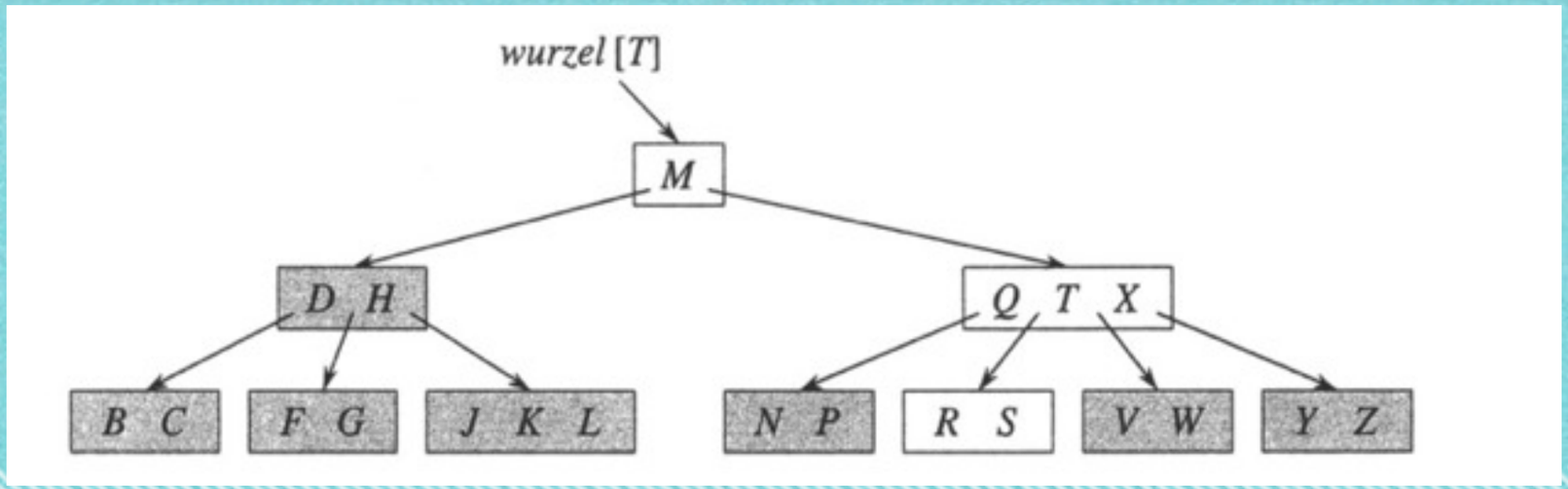
Kontext: Speicherhierarchien mit unterschiedlich schnellen Zugriffszeiten

4.9 B-Bäume

Definition 4.13 (B-Baum)

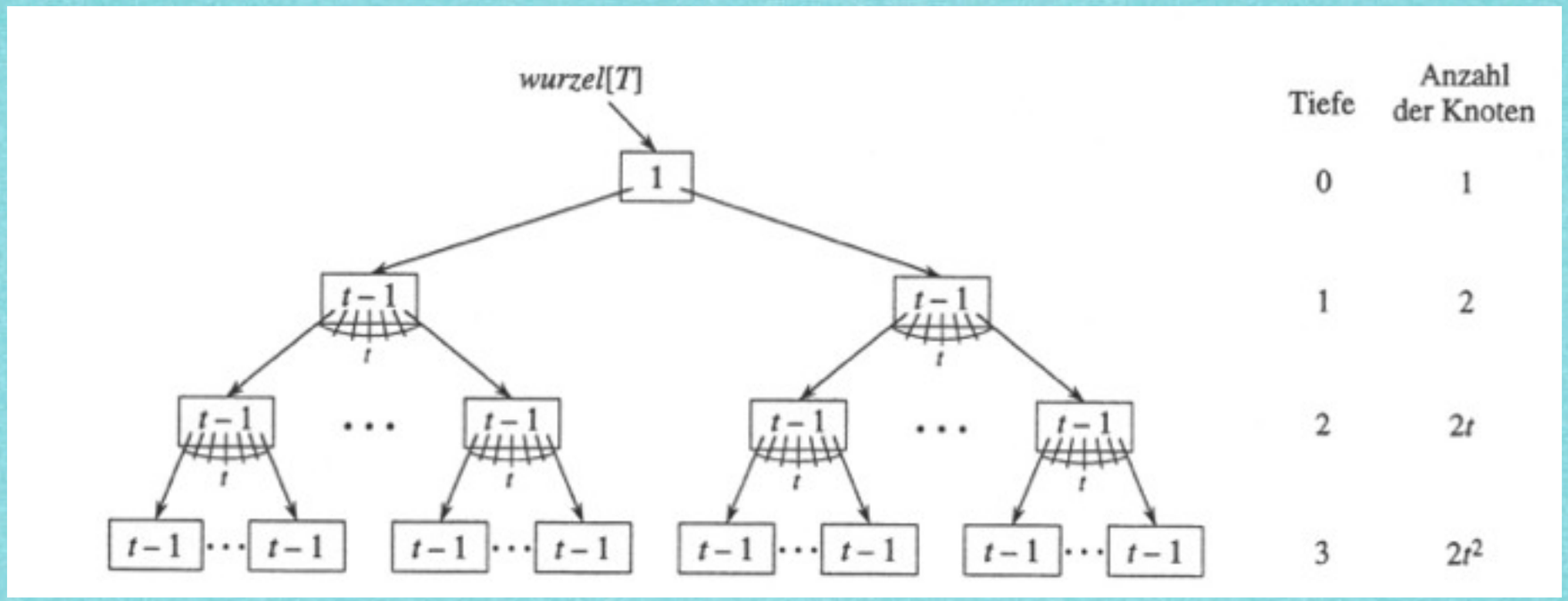
Ein B-Baum ist ein gerichteter Baum, der die folgenden Eigenschaften hat:

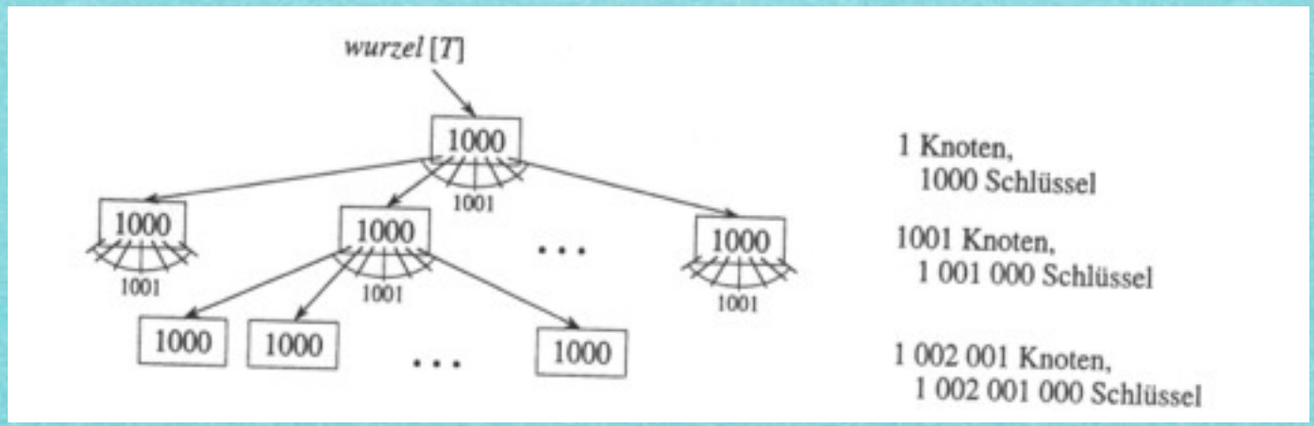
1. ***Jeder Knoten hat die folgenden Attribute:***
   a. ***die Anzahl n[x] der in x gespeicherten Schlüssel,***
   b. ***die sortiert gespeicherten Schlüssel,***
   c. ***ein Boolscher Wert, der anzeigt, ob x ein Blatt ist.***
2. ***Jeder innere Knoten enthält n[x]+1 Zeiger auf seine Kinder.***
3. ***Die Schlüssel unterteilen die darunter stehenden Teilbäume nach Größe.***
4. ***Alle Blätter haben gleiche Tiefe.***
5. ***Jeder Knoten hat Mindest- und Maximalzahlen von Schlüsseln:***
   a. ***Jeder Knoten außer der Wurzel hat mindestens t-1 Schlüssel. (Also hat jeder innere Knoten mindestens t Kinder.)***
   b. ***Jeder Knoten hat höchstens 2t-1 Schlüssel, also höchstens 2t Kinder.***



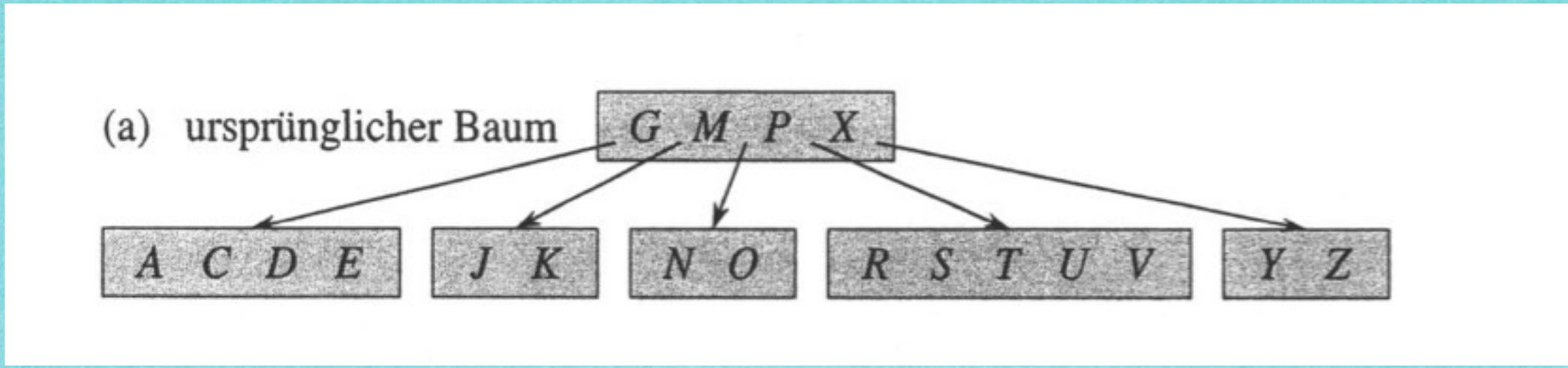
t=3

4.9 B-Bäume

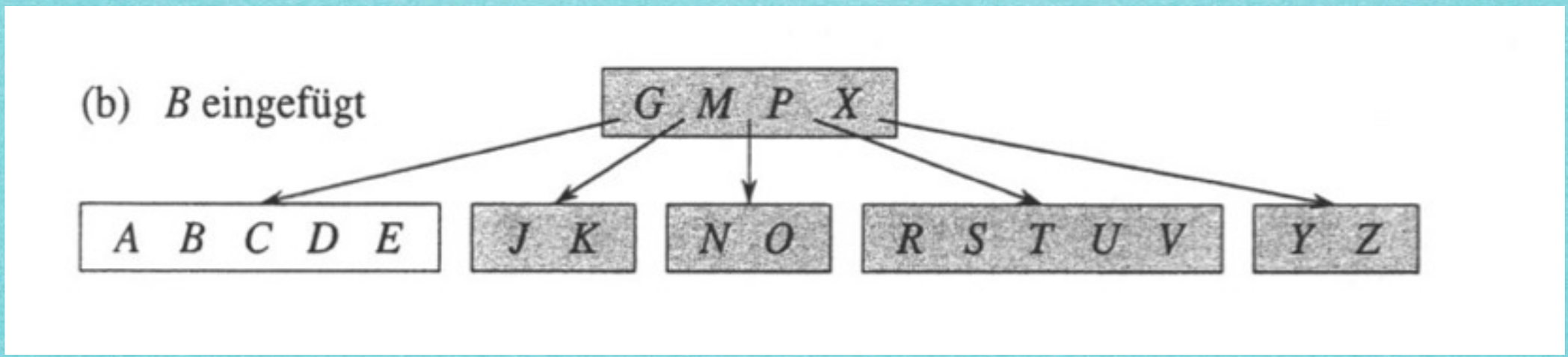

wurzel[T]
1
t − 1
t
Tiefe
Anzahl der Knoten
0 1
1 2
2 2t
3 2t²


wurzel [T]
1000
1001
1 Knoten,
1000 Schlüssel
1001 Knoten,
1 001 000 Schlüssel
1 002 001 Knoten,
1 002 001 000 Schlüssel

4.9 B-Bäume: Einfügen

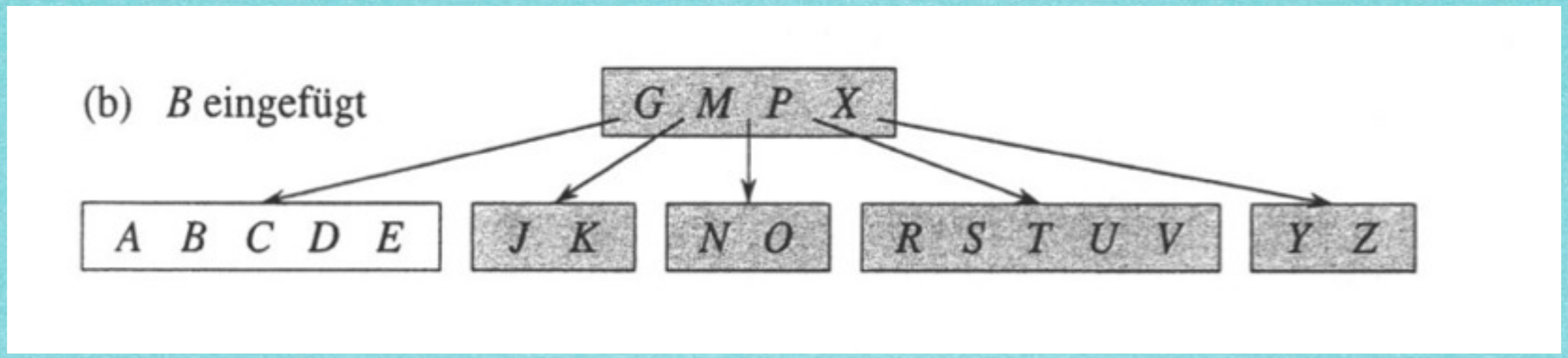


Füge "B" ein!

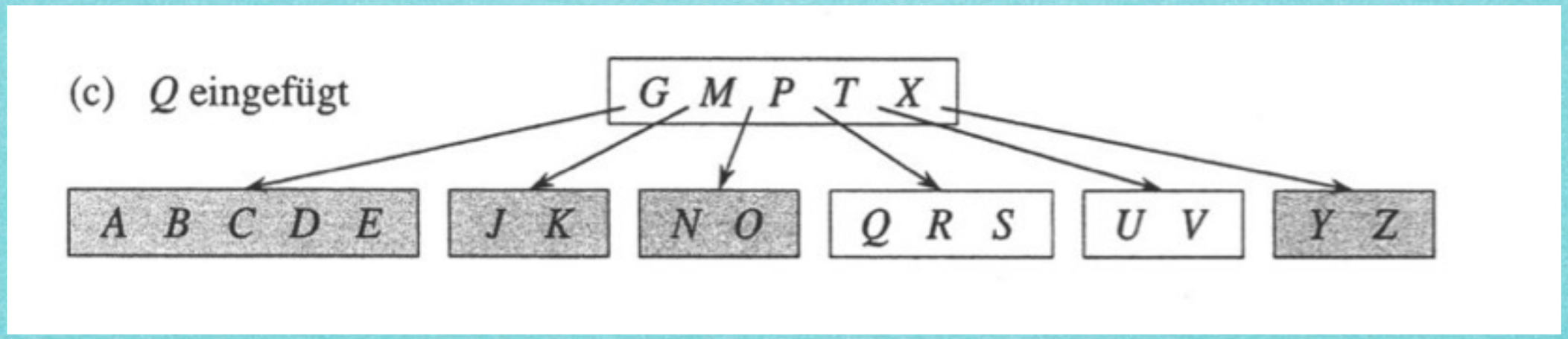


Füge "Q" ein!

4.9 B-Bäume: Einfügen

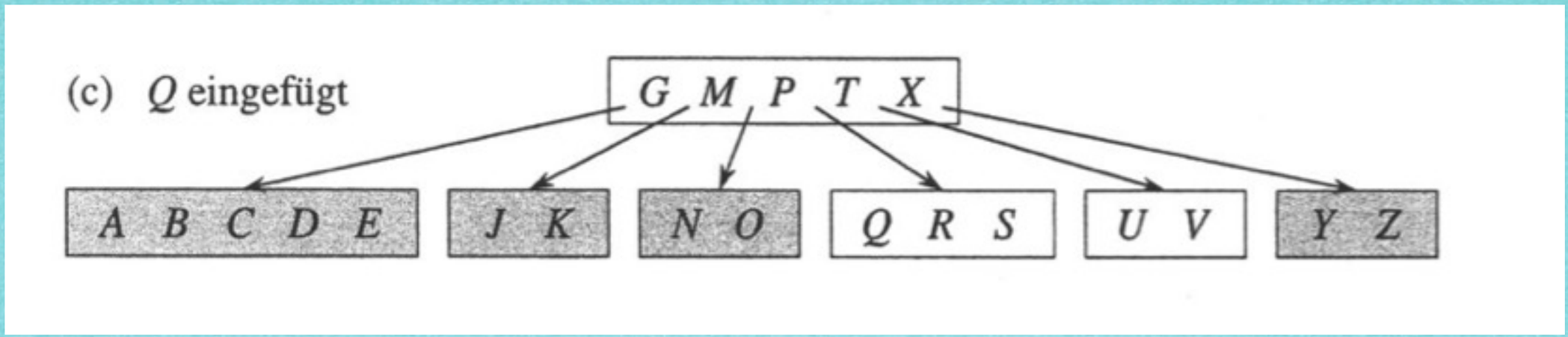


Füge "Q" ein!

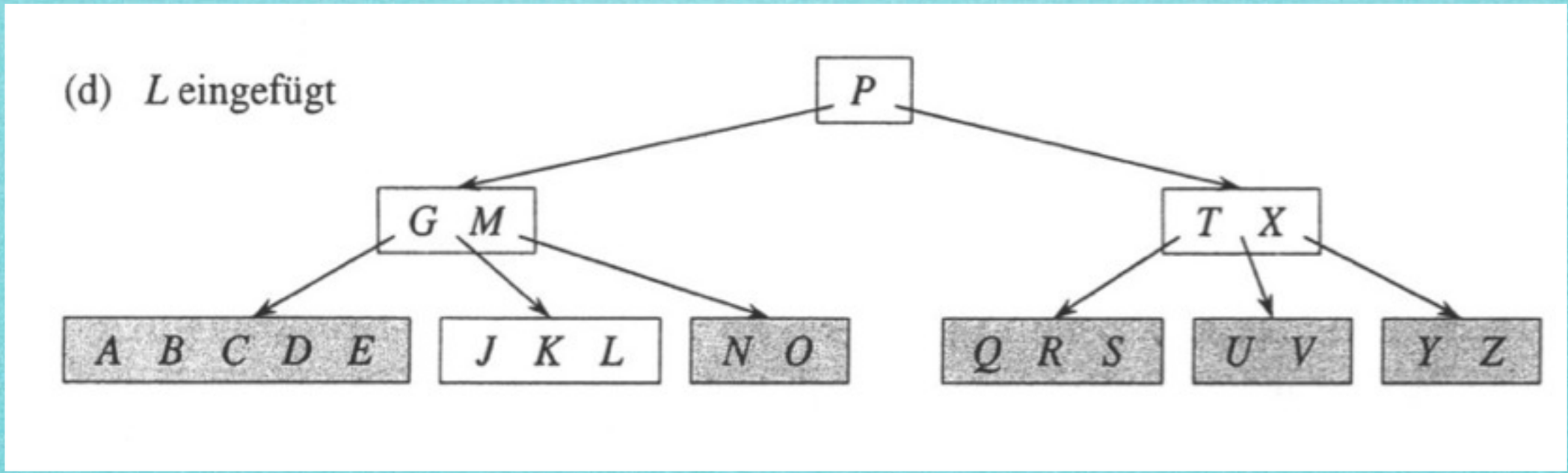


Füge "L" ein!

4.9 B-Bäume: Einfügen

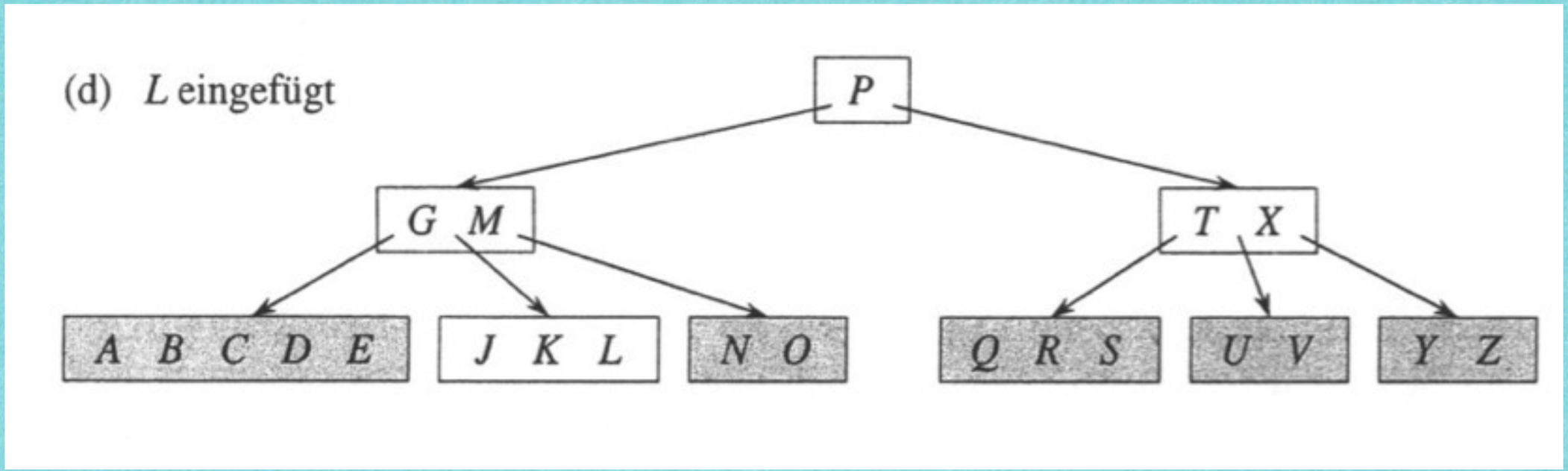


Füge "L" ein!

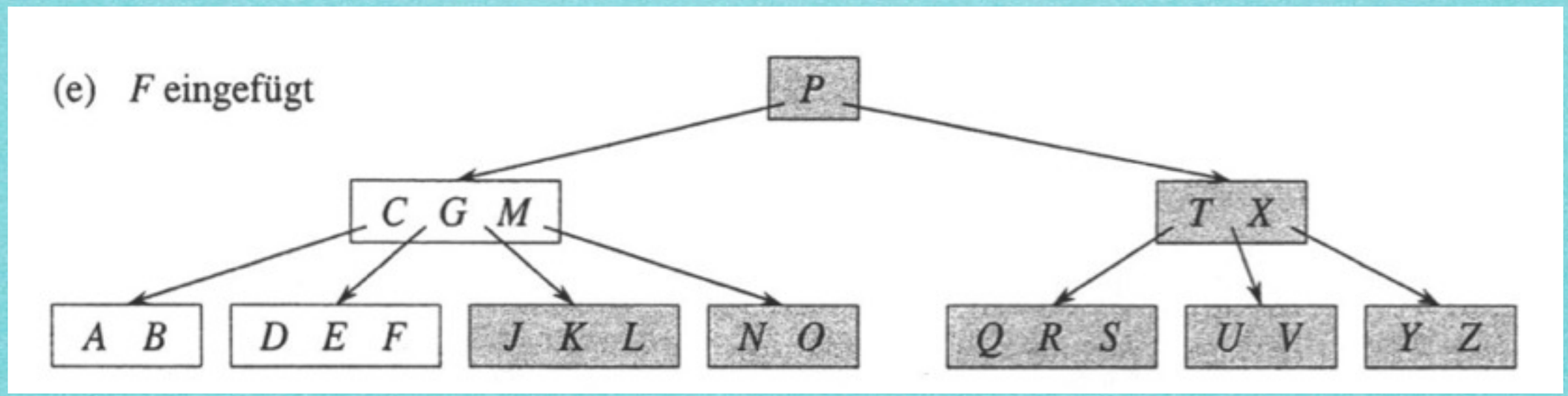


Füge "F" ein!

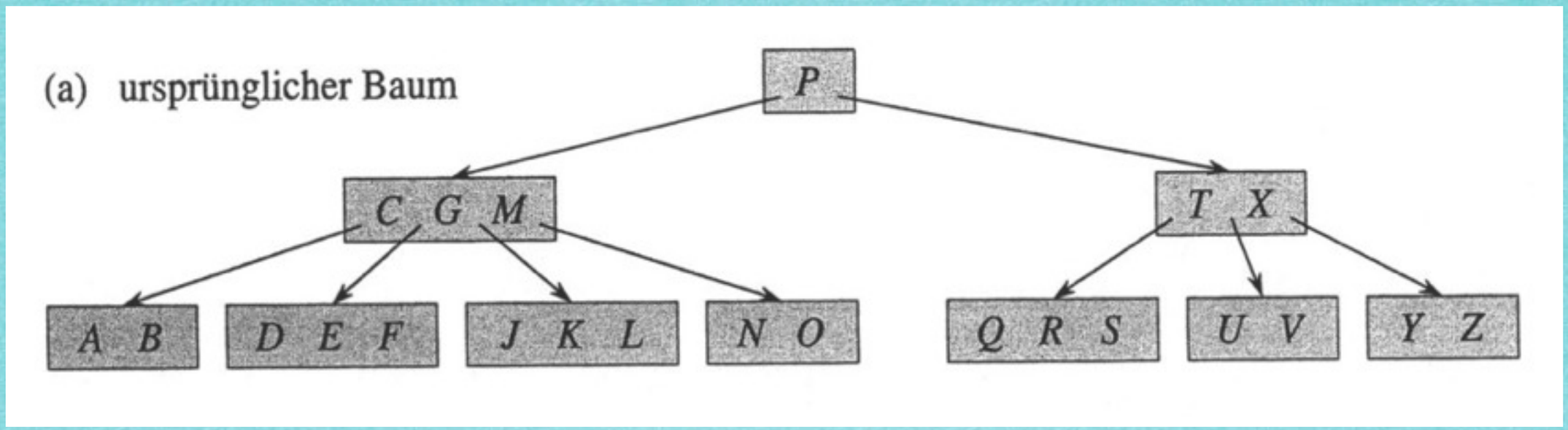
4.9 B-Bäume: Einfügen



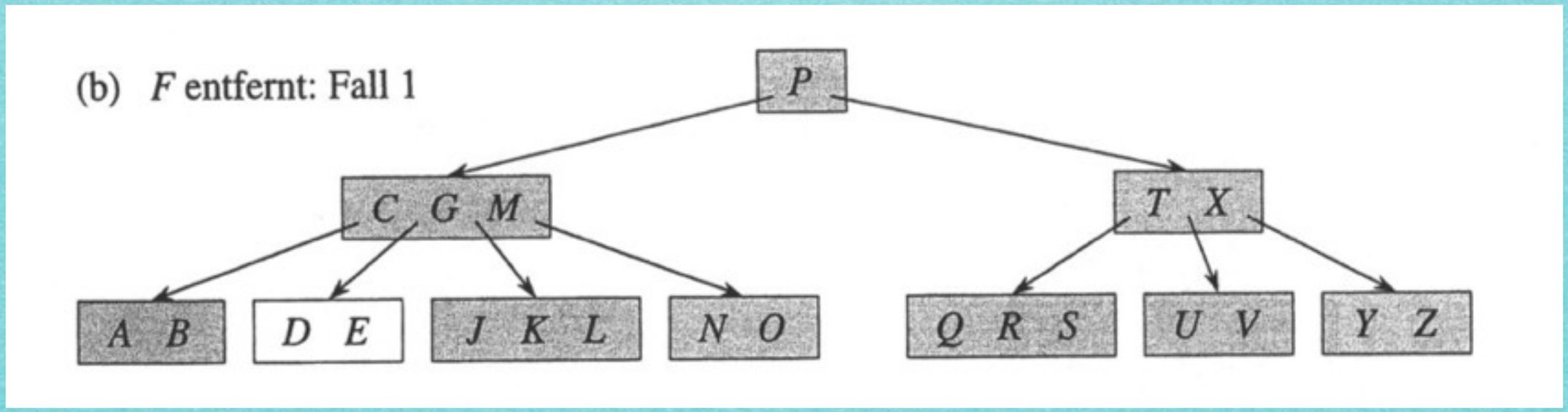
Füge "F" ein!



4.9 B-Bäume: Löschen

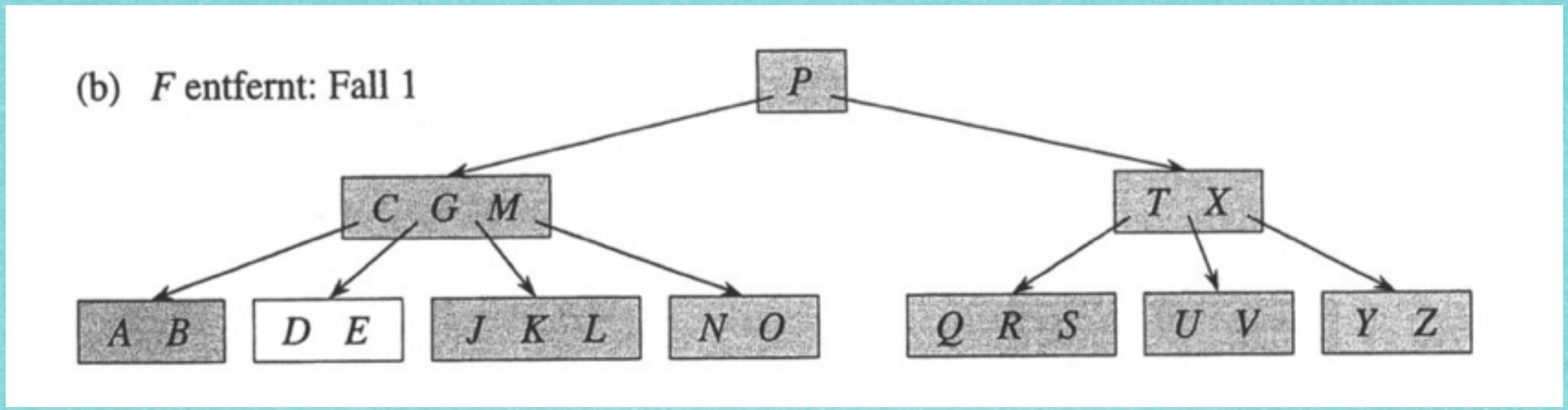


Lösche "F"!

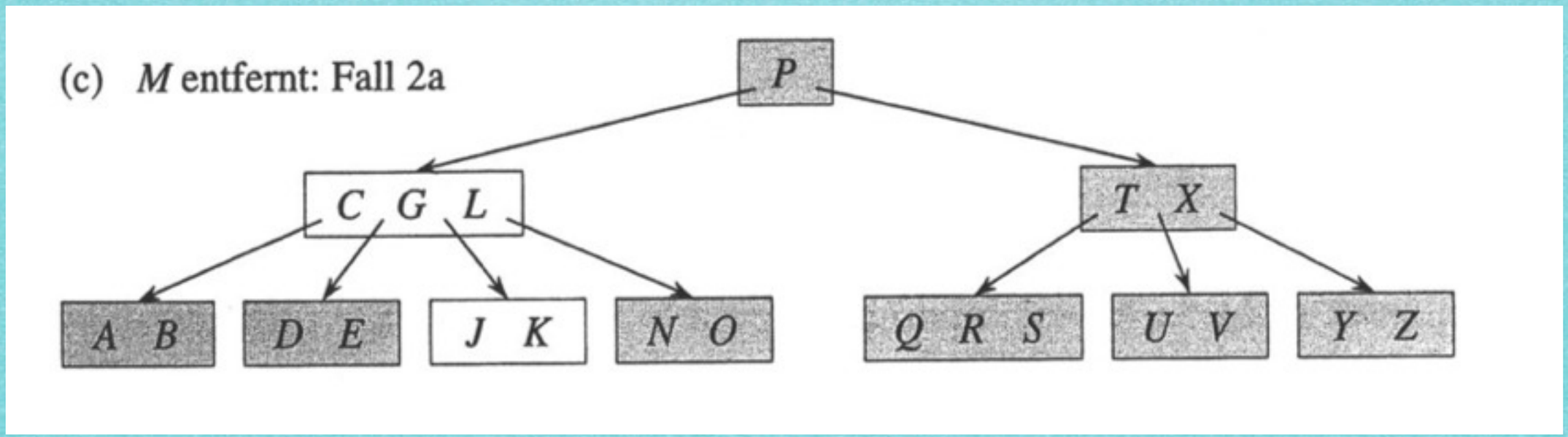


Lösche "M"!

4.9 B-Bäume: Löschen

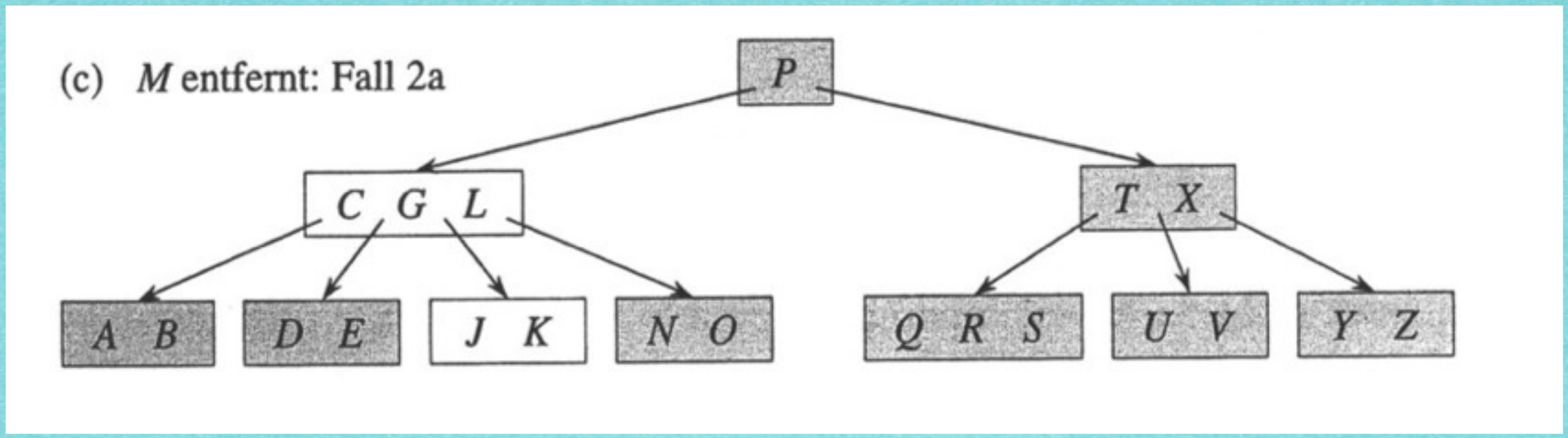


Lösche "M"!

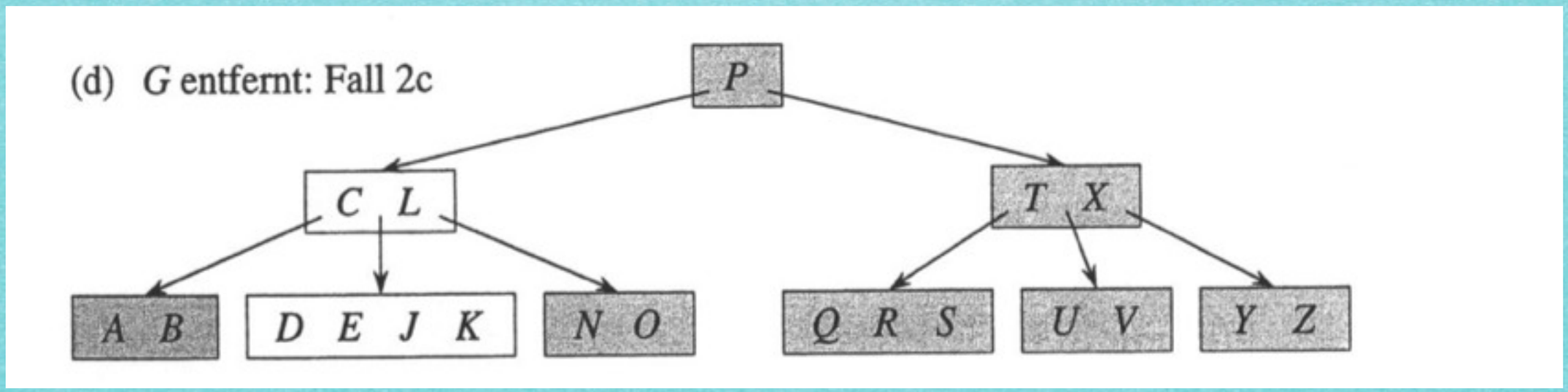


Lösche "G"!

4.9 B-Bäume: Löschen

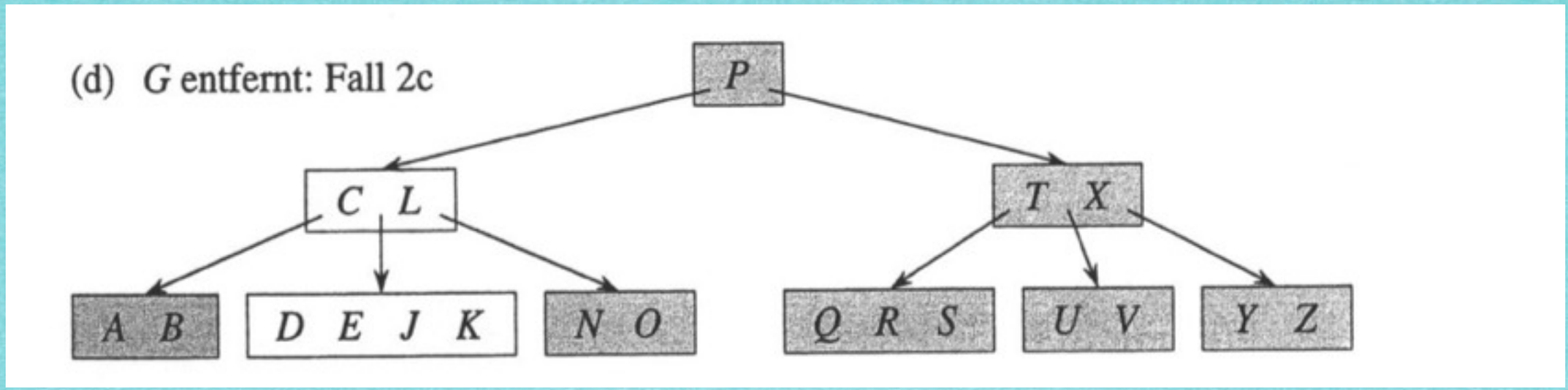


Lösche "G"!

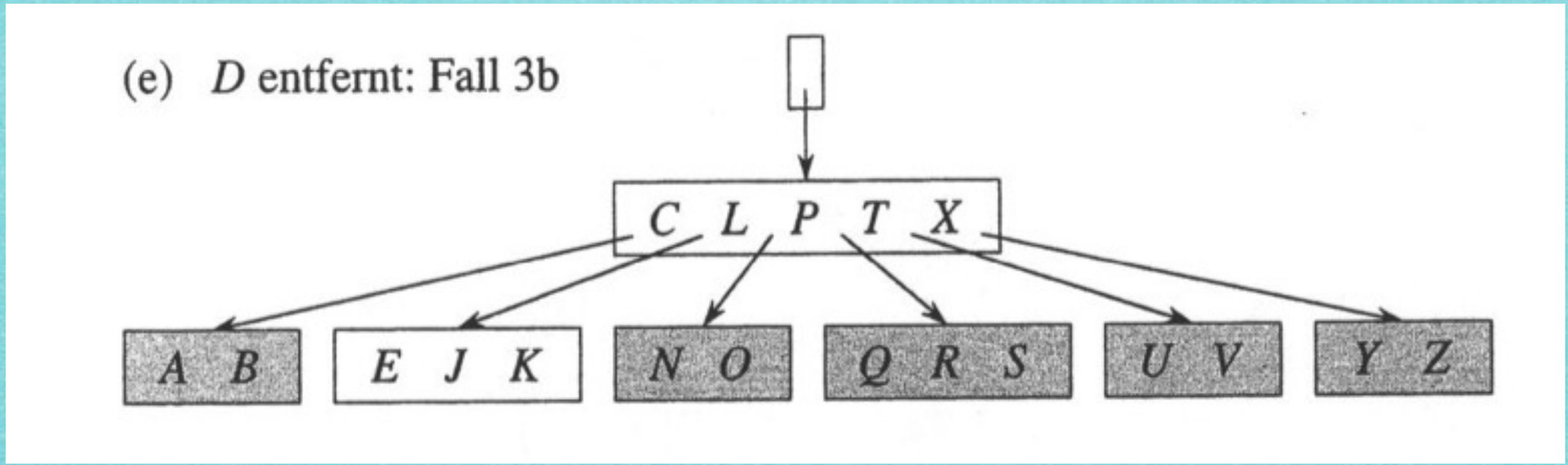


Lösche "D"!

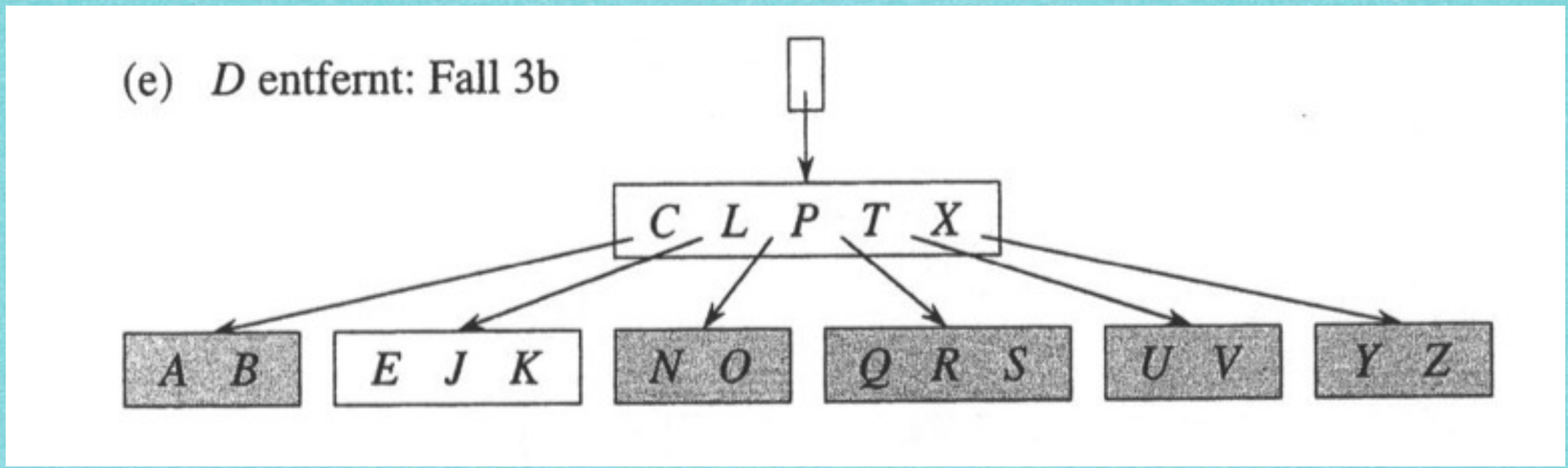
4.9 B-Bäume: Löschen

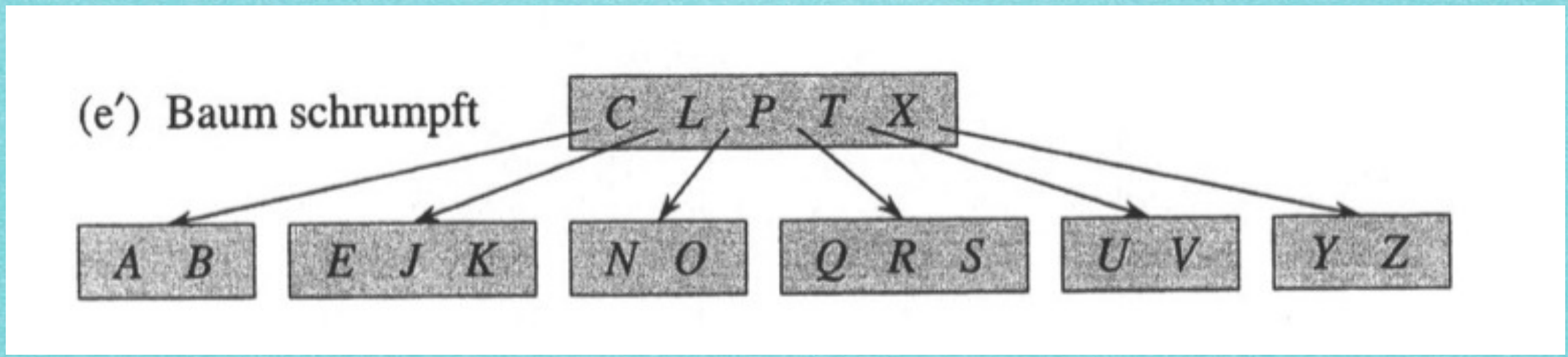


Lösche "D"!



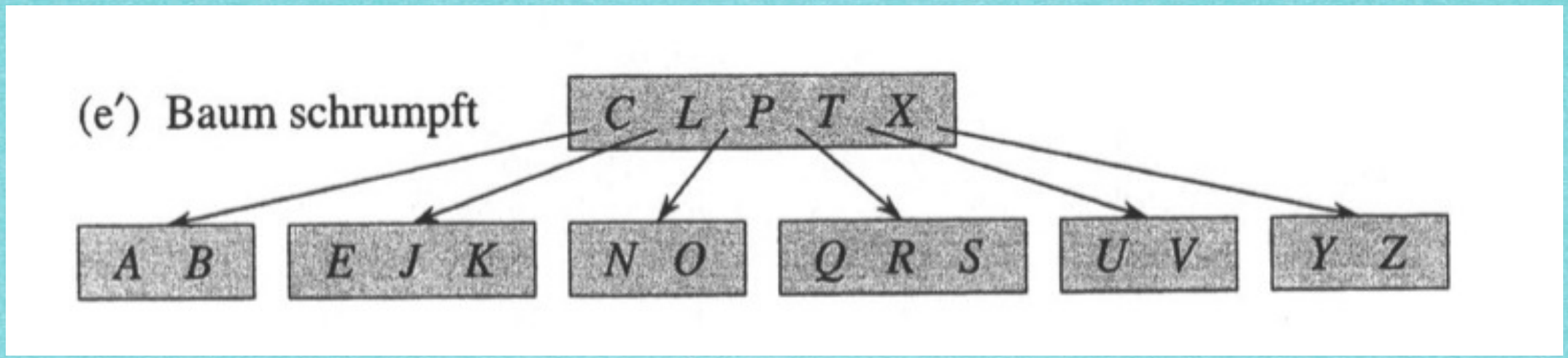
4.9 B-Bäume: Löschen



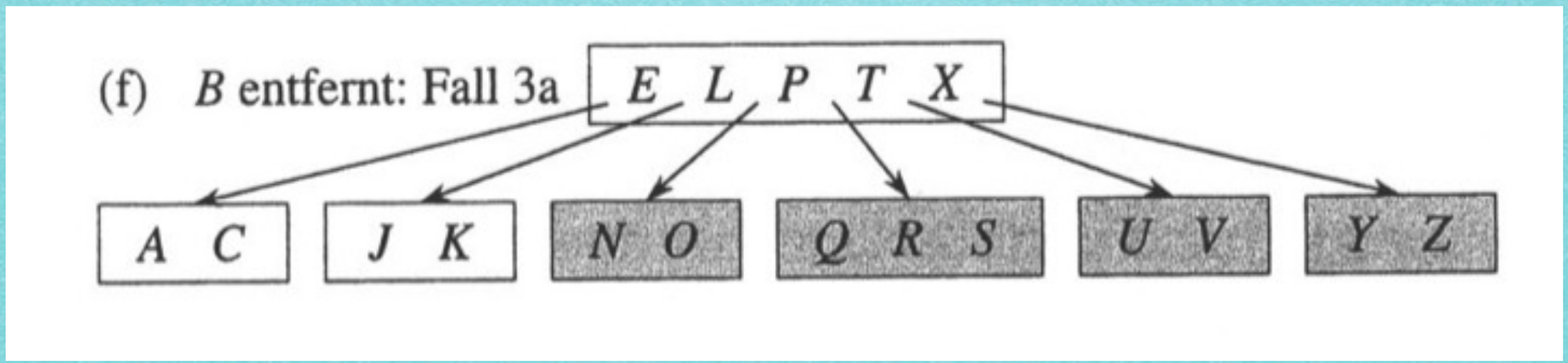


Lösche "B"!

4.9 B-Bäume: Löschen

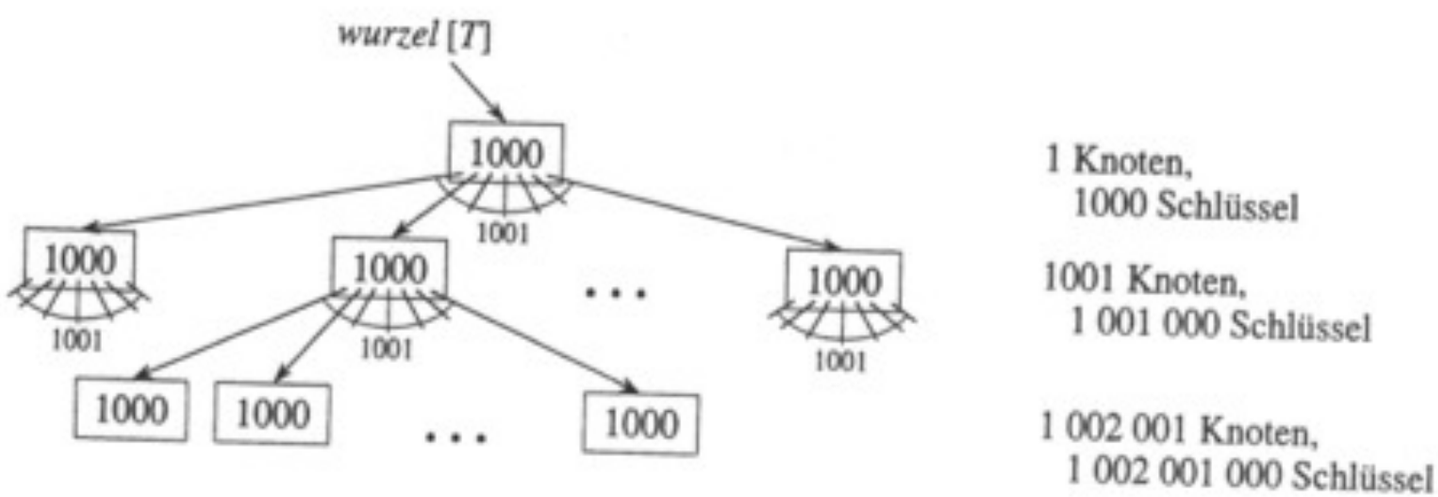


Lösche "B"!

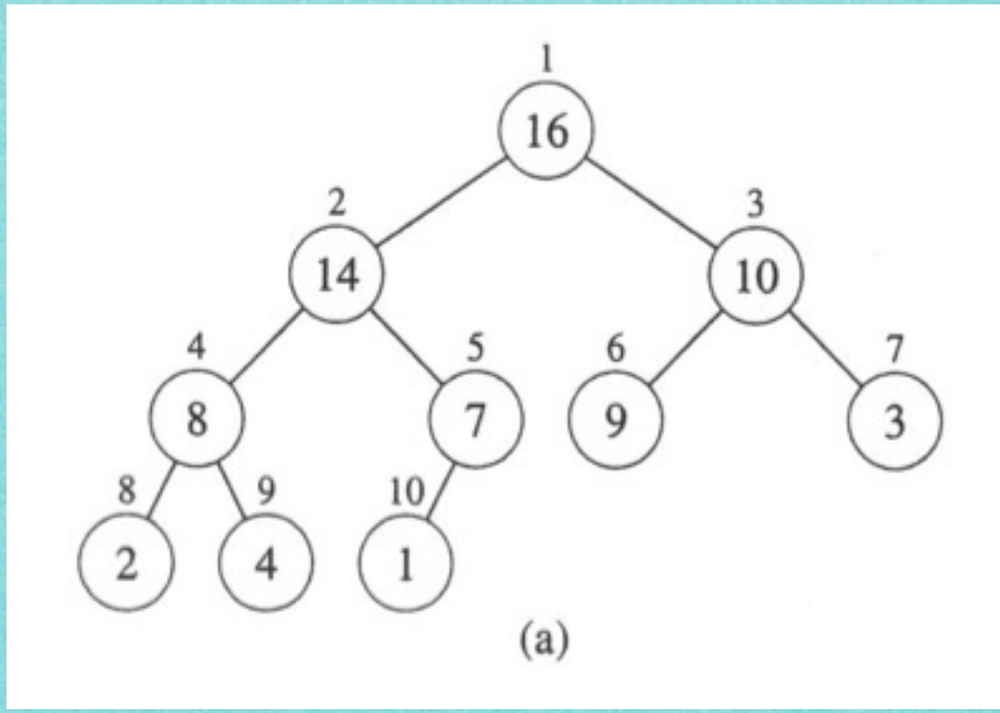


4.9 B-Bäume

<u>Satz 4.14</u>
Ein B-Baum der Höhe h benötigt zur dynamischen Datenverwaltung im schlimmsten Falle O(h) Plattenoperationen und $O(t \log_t n)$ CPU-Zeit.



4.10 Heaps



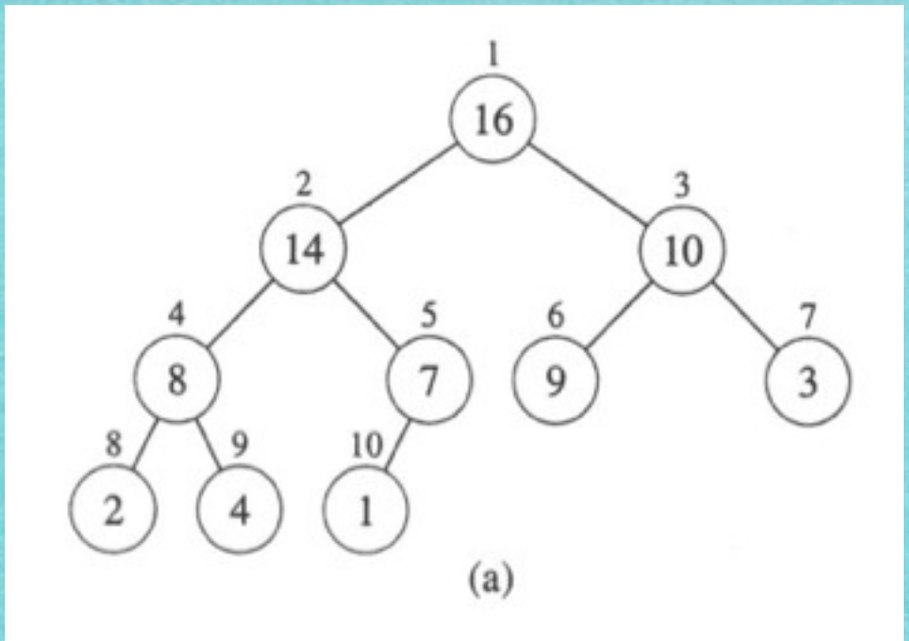
Idee: Ordne Baum so, dass
größere Elemente immer oben stehen!

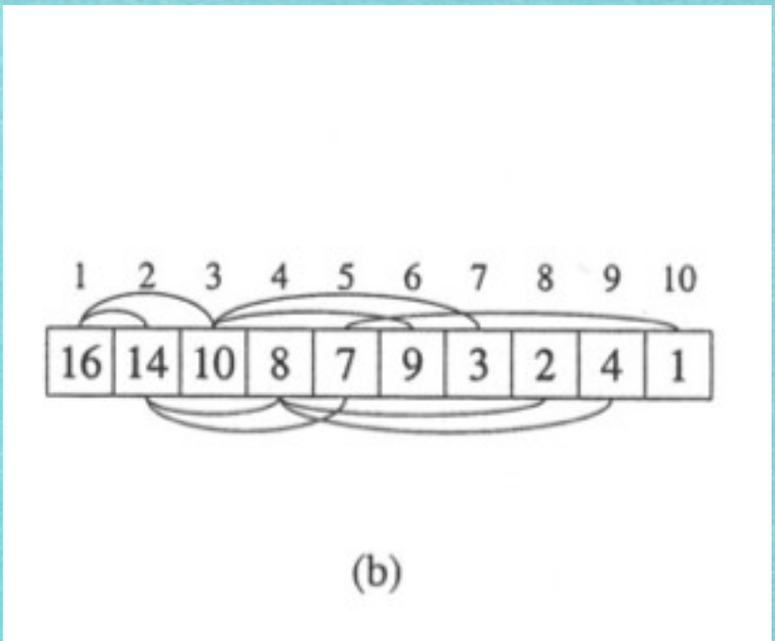
4.10 Heaps

<u>Definition 4.14 (Heap)</u>
Ein gerichteter binärer Baum heißt *binärer Max-Heap*, wenn er folgende Eigenschaften erfüllt:

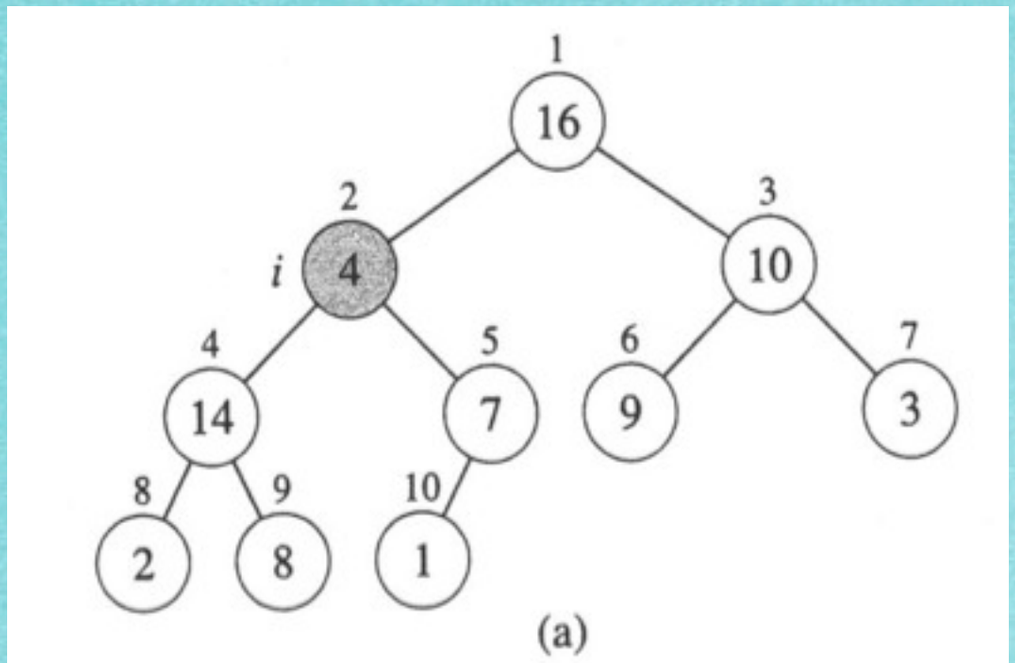
1. ***Jeder Knoten hat einen Schlüssel.***
2. ***Ist h die maximale Distanz von der Wurzel, dann haben alle Ebenen $i<h$ genau 2^i Knoten.***
3. ***Auf Ebene h sind die linken $n - 2^h + 1$ Positionen besetzt.***
4. ***Der Schlüssel jedes Knotens ist mindestens so groß wie die seiner Kinder.***

In einem Min-Heap sind Schlüssel höchstens so groß wie die von Kinderknoten.

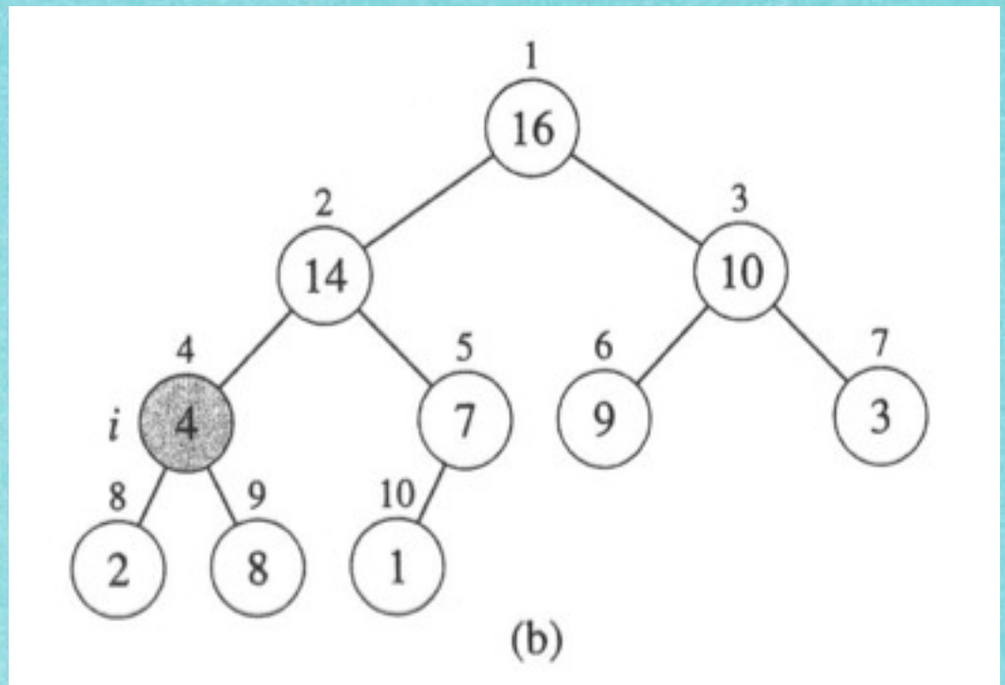




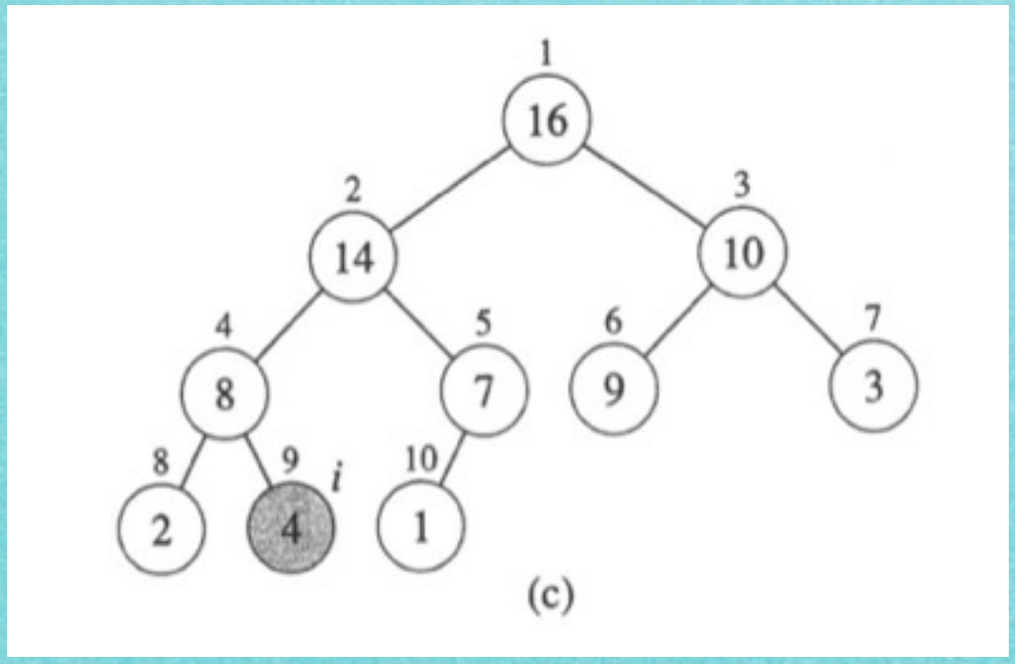
4.10 Heaps: Umordnen



(a)



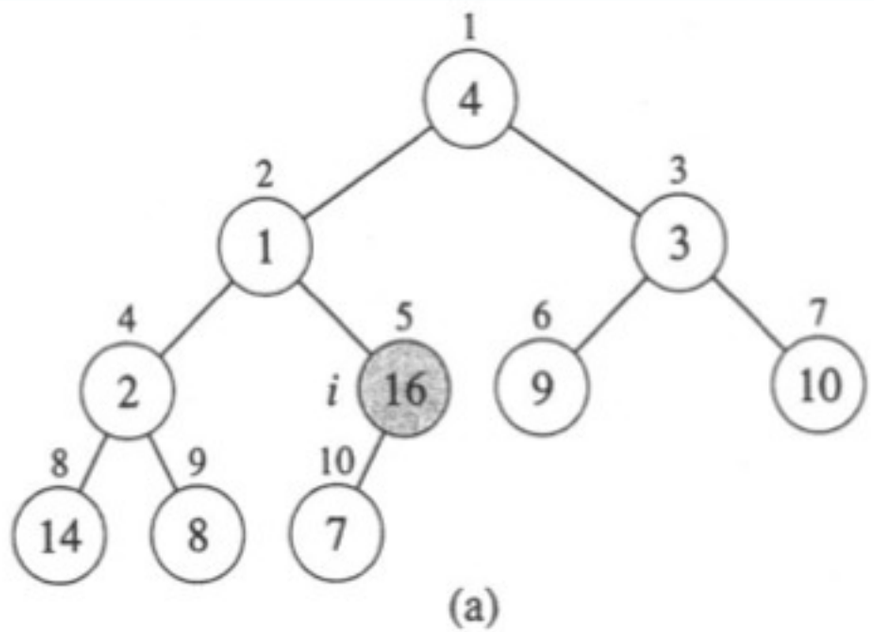
(b)



(c)

```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-größe[A] und A[l] > A[i]
    then maximum ← l
    else maximum ← i
if r ≤ heap-größe[A] und A[r] > A[maximum]
    then maximum ← r
if maximum ≠ i
    then vertausche A[i] ↔ A[maximum]
         MAX-HEAPIFY(A, maximum)
```

4.10 Heaps: Bauen



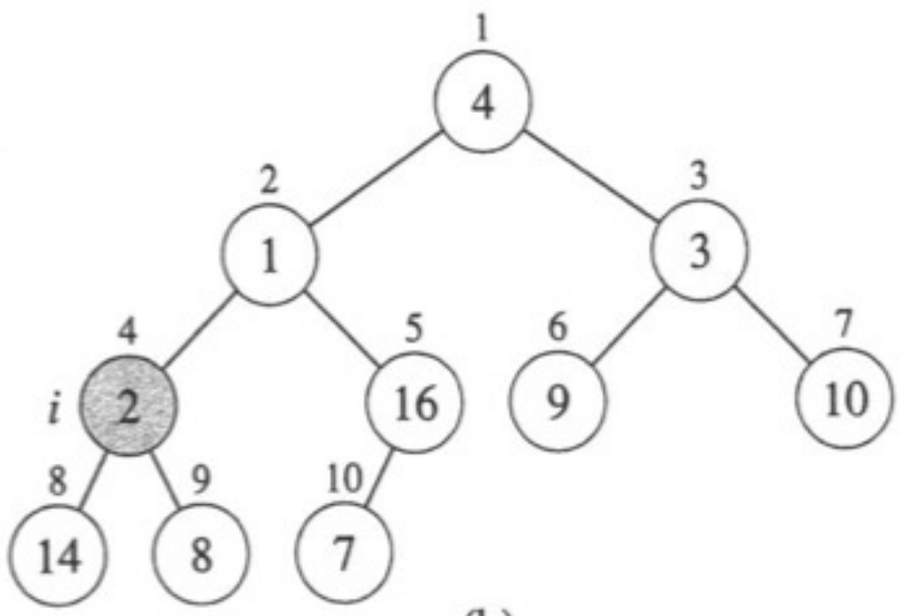
(a)

A	4	1	3	2	16	9	10	14	8	7

BUILD-MAX-HEAP(A)

```
heap-größe[A] ← länge[A]
for i ← ⌊länge[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

4.10 Heaps: Bauen



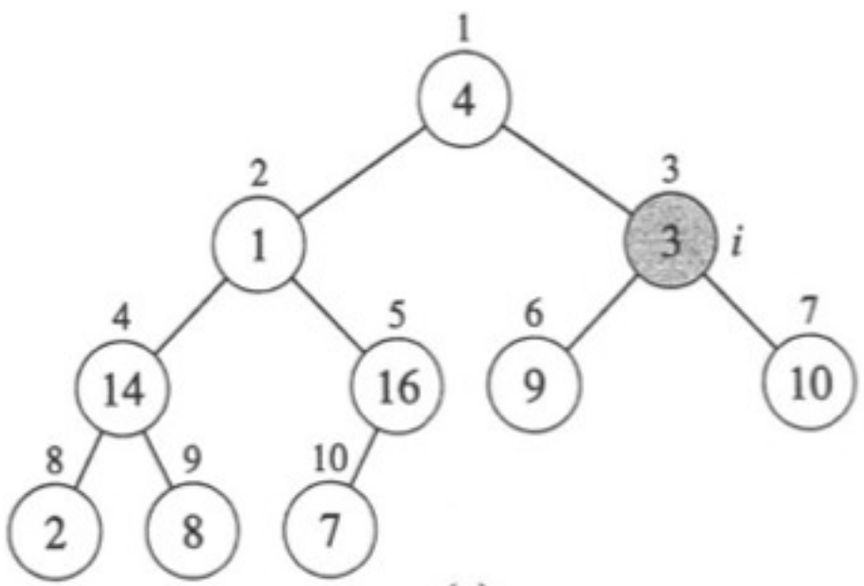
(b)

A	4	1	3	2	16	9	10	14	8	7

BUILD-MAX-HEAP(A)

```
heap-größe[A] ← länge[A]
for i ← ⌊länge[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

4.10 Heaps: Bauen



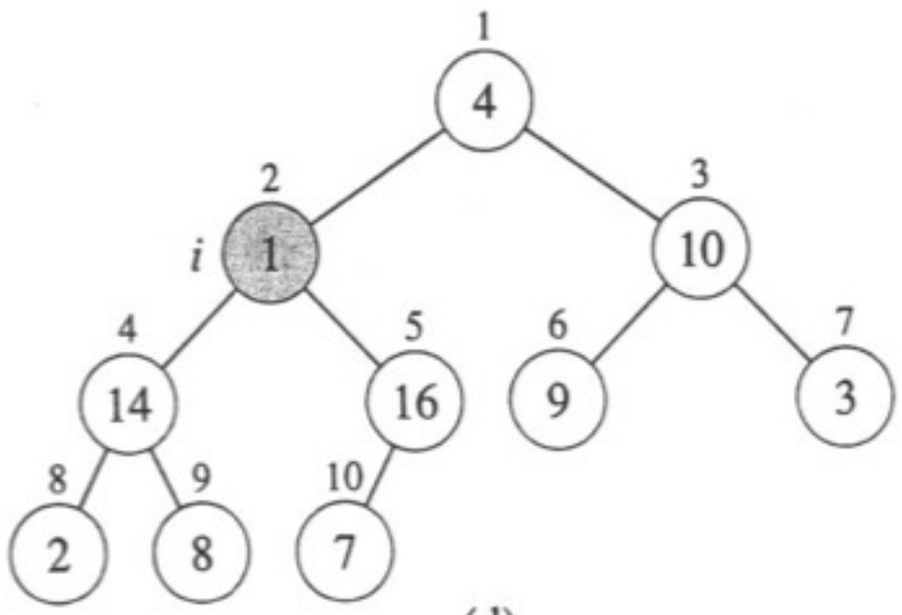
(c)

A	4	1	3	2	16	9	10	14	8	7

BUILD-MAX-HEAP(A)

```
heap-größe[A] ← länge[A]
for i ← ⌊länge[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

4.10 Heaps: Bauen

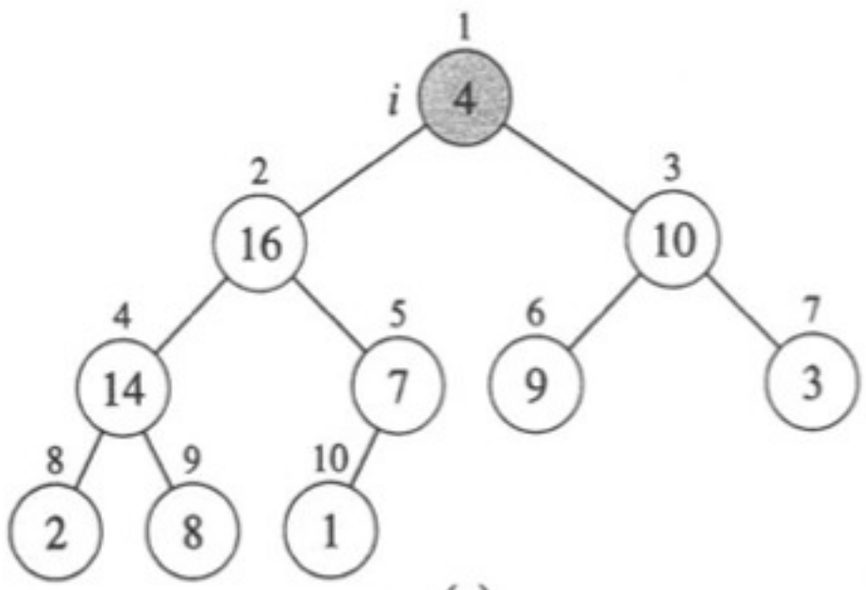


(d)

A	4	1	3	2	16	9	10	14	8	7

```
BUILD-MAX-HEAP(A)
heap-größe[A] ← länge[A]
for i ← ⌊länge[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

4.10 Heaps: Bauen



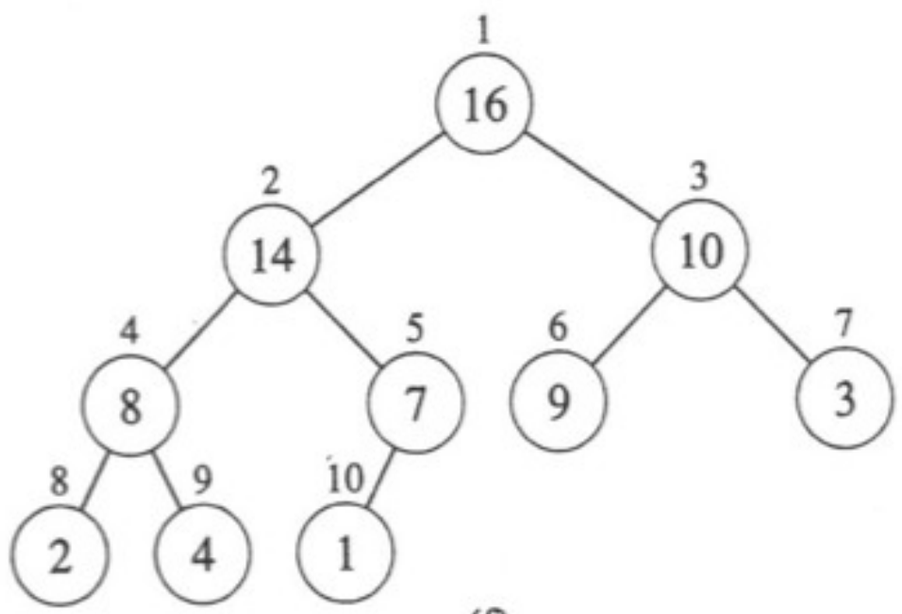
(e)

A	4	1	3	2	16	9	10	14	8	7

BUILD-MAX-HEAP(A)

```
heap-größe[A] ← länge[A]
for i ← ⌊länge[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

4.10 Heaps: Bauen



(f)

A	4	1	3	2	16	9	10	14	8	7

BUILD-MAX-HEAP(A)

```
heap-größe[A] ← länge[A]
for i ← ⌊länge[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

4.10 Heaps: Bauen

Da man O(log n) Operationen pro Level benötigt, ist klar, dass ein Max-Heap in O(n log n) gebaut werden kann.
Da aber mehr Knoten auf niedrigerem Niveau sind, was weniger Arbeit erfordert, kann man diese Abschätzung noch verbessern:

<u>Satz 4.14</u>
Ein Max-Heap mit n Knoten kann in O(n) gebaut werden.

4.11 Andere Strukturen

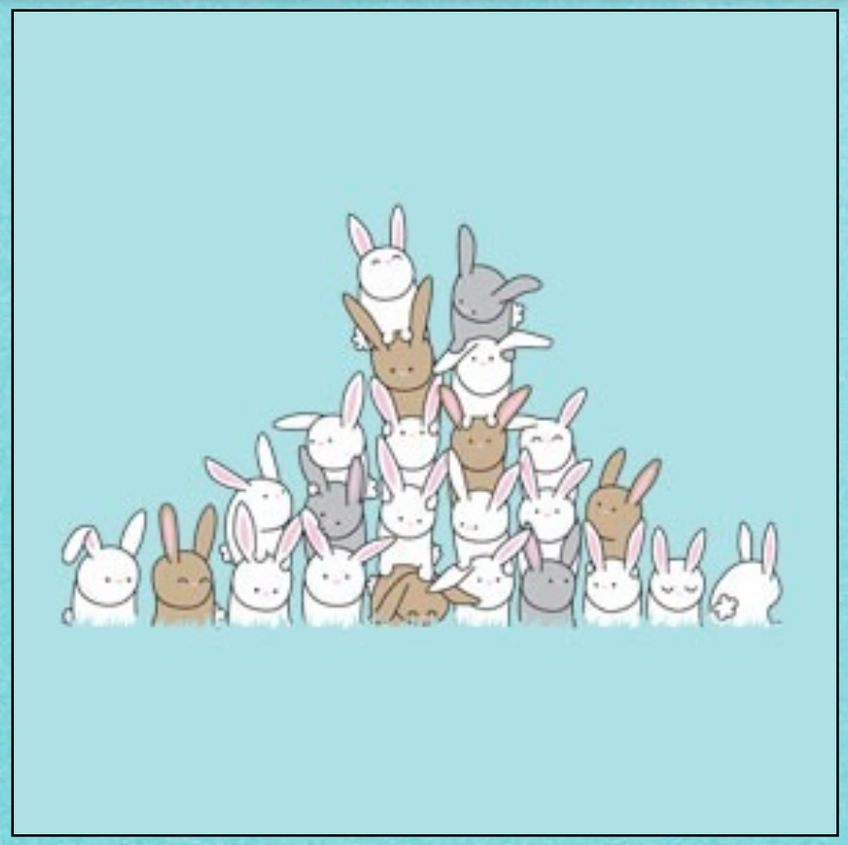
4.11.1 Fibonacci-Heaps:
Heap-Struktur mit sehr schneller *amortisierter* Zugriffszeit.

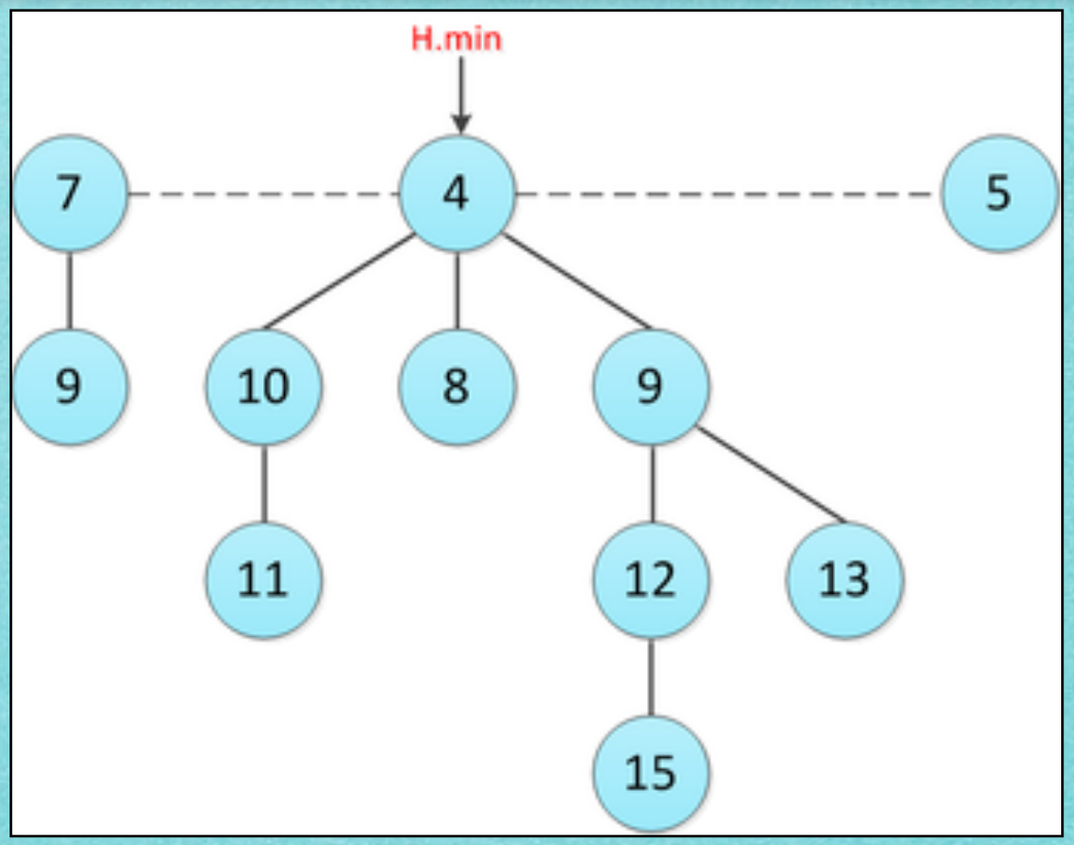
Common Operations	*Effect*	Unsorted Linked List	Self-balancing binary search tree	Binary heap	Binomial heap	Fibonacci heap
insert(data,key)	Adds *data* to the queue, tagged with *key*	O(1)	O(log n)	O(log n)	O(log n)	O(1)
findMin() -> key,data	Returns *key,data* corresponding to min-value *key*	O(n)	O(log n) or O(1) (**)	O(1)	O(log n)[4]	O(1)[1]
deleteMin()	Deletes *data* corresponding to min-value *key*	O(n)	O(log n)	O(log n)	O(log n)	O(log n)*
delete(node)	Deletes *data* corresponding to given *key*, given a pointer to the node being deleted	O(1)	O(log n)	O(log n)	O(log n)	O(log n)*
decreaseKey(node)	Decreases the *key* of a node, given a pointer to the node being modified	O(1)	O(log n)	O(log n)	O(log n)	O(1)*
merge(heap1,heap2) -> heap3	Merges two heaps into a third	O(1)	O(m log(n+m))	O(m + n)	O(log n)***	O(1)

(*)Amortized time
(**)With trivial modification to store an additional pointer to the minimum element
(***)Where n is the size of the larger heap

4.11 Andere Strukturen

4.11.1 Fibonacci-Heaps:
Heap-Struktur mit sehr schneller *amortisierter* Zugriffszeit.





4.11 Andere Strukturen

4.11.2 Cache-Oblivious B-Trees:
Umgang mit großen Datenmengen bei unbekannter Cache-Größe

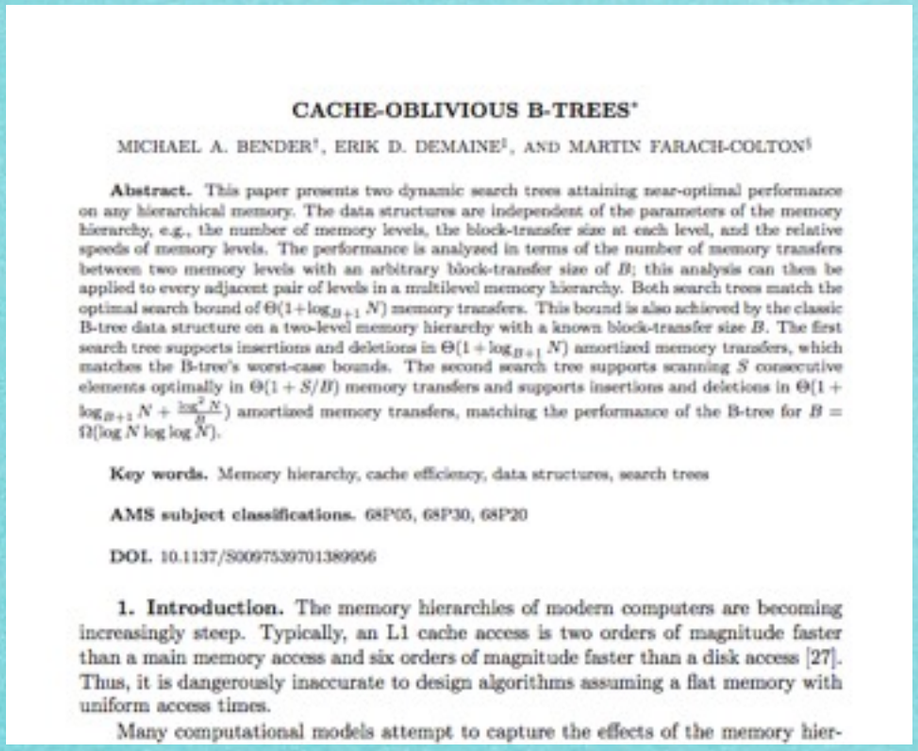
Michael Bender

Erik Demaine

Martin
Farach-Colton

4.11 Andere Strukturen

4.11.2 Cache-Oblivious B-Trees: Umgang mit großen Datenmengen bei unbekannter Cache-Größe



CACHE-OBLIVIOUS B-TREES*

MICHAEL A. BENDER†, ERIK D. DEMAINE‡, AND MARTIN FARACH-COLTON§

Abstract. This paper presents two dynamic search trees attaining near-optimal performance on any hierarchical memory. The data structures are independent of the parameters of the memory hierarchy, e.g., the number of memory levels, the block-transfer size at each level, and the relative speeds of memory levels. The performance is analyzed in terms of the number of memory transfers between two memory levels with an arbitrary block-transfer size of B; this analysis can then be applied to every adjacent pair of levels in a multilevel memory hierarchy. Both search trees match the optimal search bound of $\Theta(1+\log_{B+1} N)$ memory transfers. This bound is also achieved by the classic B-tree data structure on a two-level memory hierarchy with a known block-transfer size B. The first search tree supports insertions and deletions in $\Theta(1+\log_{B+1} N)$ amortized memory transfers, which matches the B-tree's worst-case bounds. The second search tree supports scanning S consecutive elements optimally in $\Theta(1+S/B)$ memory transfers and supports insertions and deletions in $\Theta(1+\log_{B+1} N+\frac{\log^2 N}{B})$ amortized memory transfers, matching the performance of the B-tree for $B=\Omega(\log N \log\log N)$.

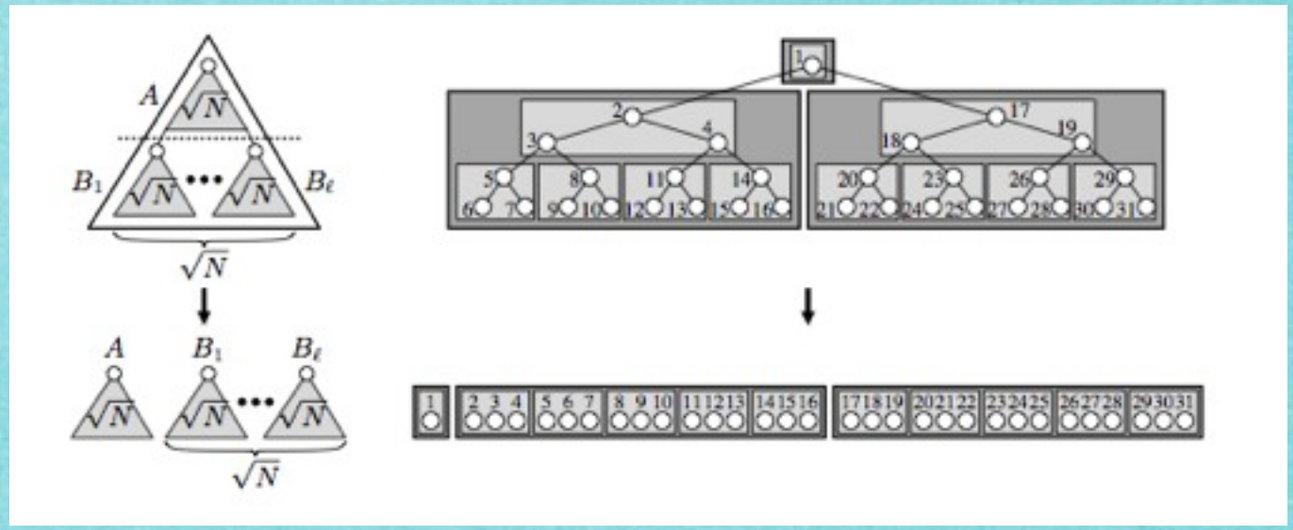
Key words. Memory hierarchy, cache efficiency, data structures, search trees

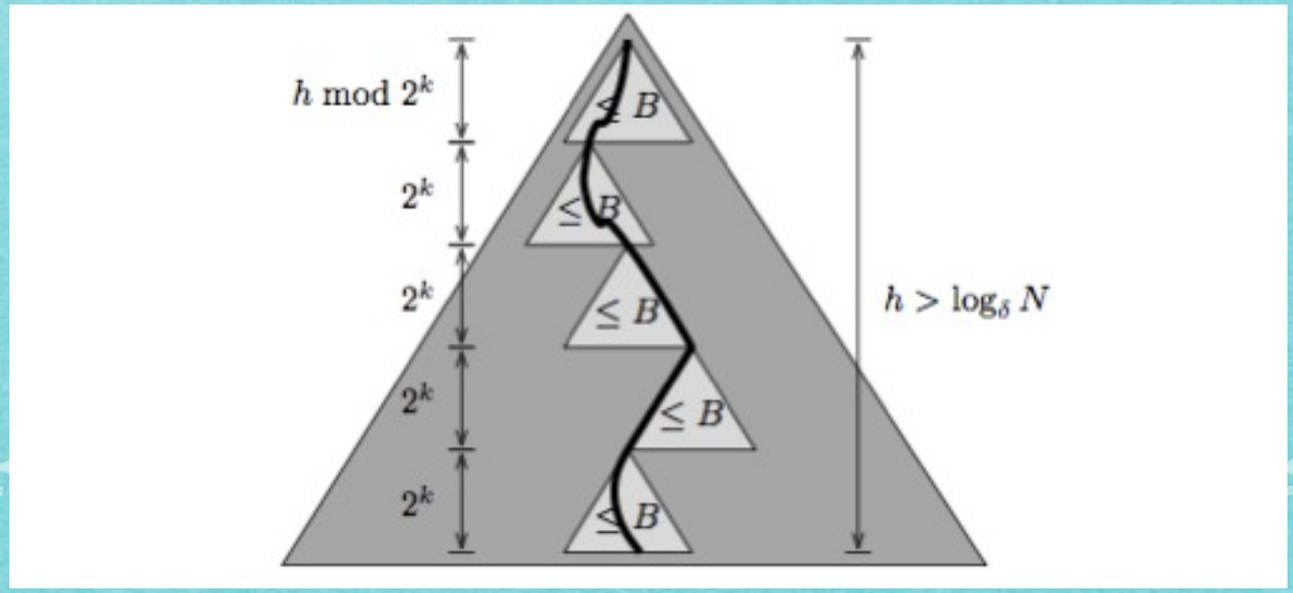
AMS subject classifications. 68P05, 68P30, 68P20

DOI. 10.1137/S0097539701389956

1. Introduction. The memory hierarchies of modern computers are becoming increasingly steep. Typically, an L1 cache access is two orders of magnitude faster than a main memory access and six orders of magnitude faster than a disk access [27]. Thus, it is dangerously inaccurate to design algorithms assuming a flat memory with uniform access times.

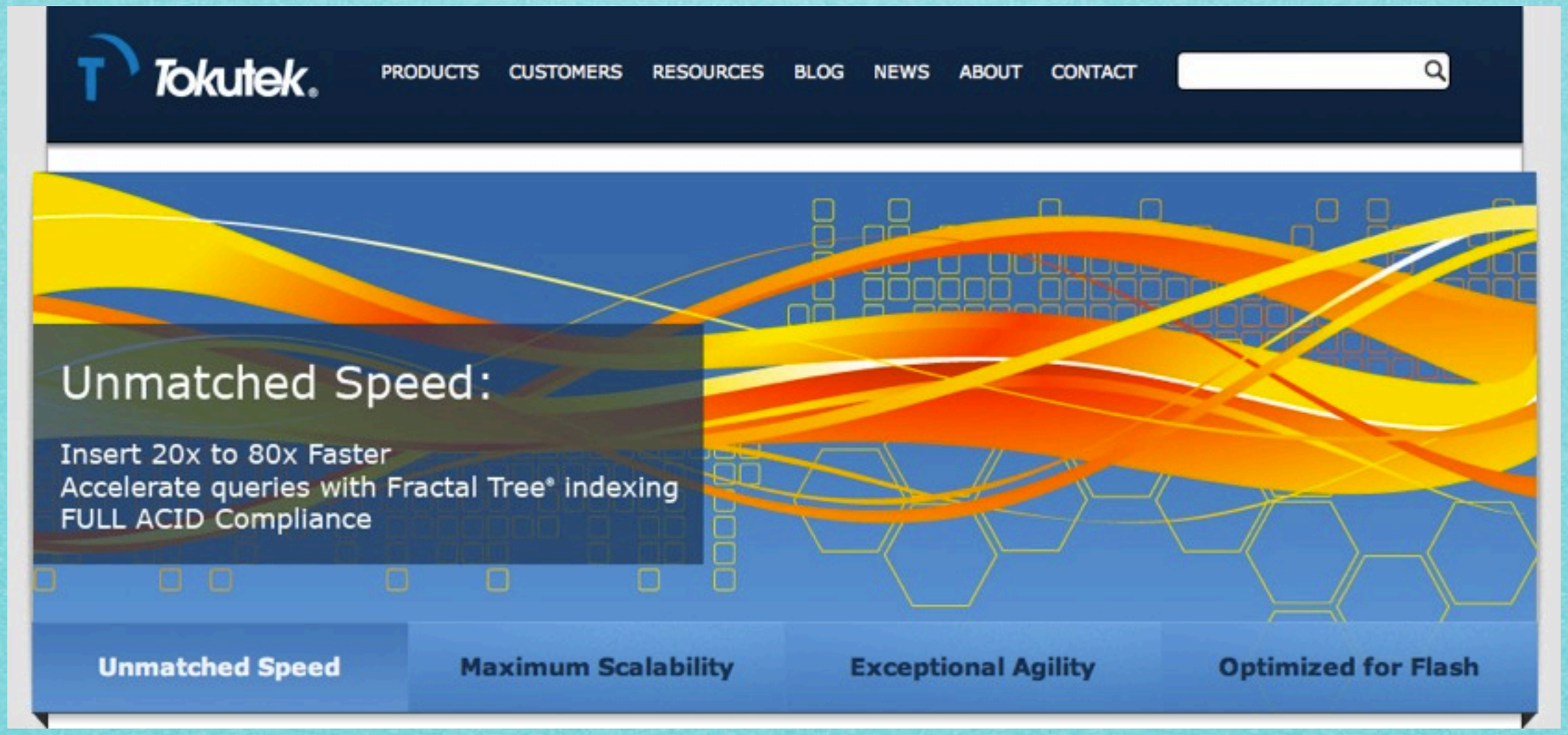
Many computational models attempt to capture the effects of the memory hier-



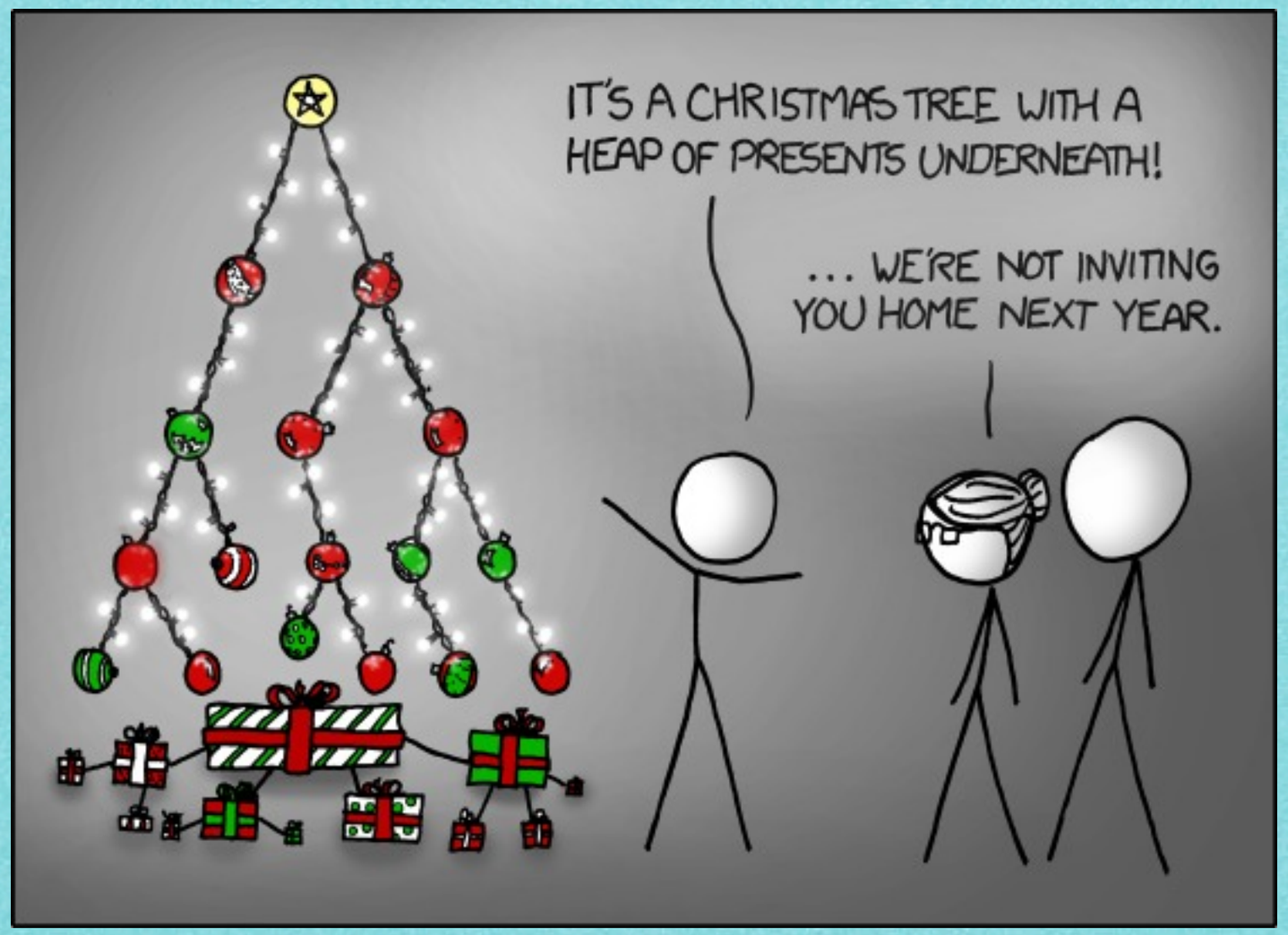


4.11 Andere Strukturen

4.11.2 Cache-Oblivious B-Trees:
Umgang mit großen Datenmengen bei unbekannter Cache-Größe



Zusammenfassung Kapitel 4!



Ende von Kapitel 4!

s.fekete@tu-bs.de