3.8 Bäume

Definition des Baumes

Ein **Baum** besteht aus einer nichtleeren Menge von **Knoten** und einer Menge von **Kanten**. Jede Kante verbindet genau zwei Knoten.

Die **Knoten** (nodes) enthalten Datenelemente.

Die **Kanten** (edges) beschreiben Beziehungen zwischen den Datenelementen.

Die hierarchisch höchste Ebene hat genau einen Knoten, die **Wurzel** des Baumes (root).

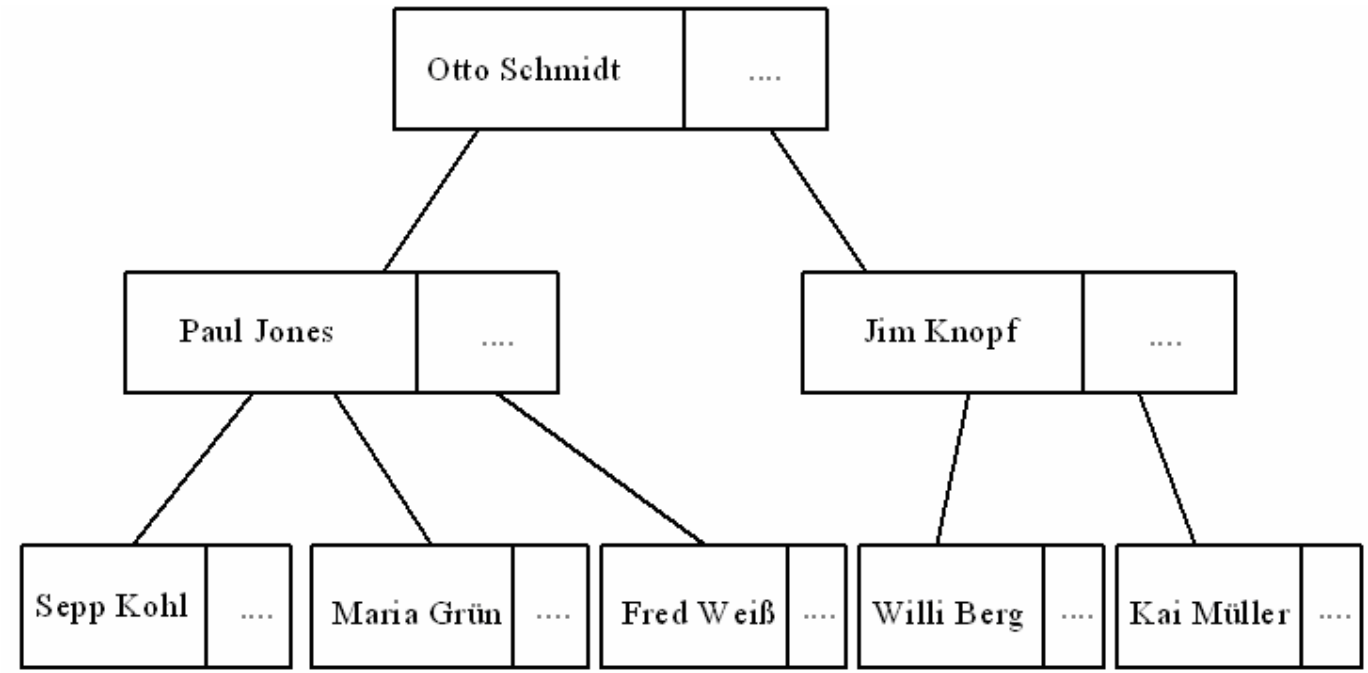
Die Knoten auf der hierarchisch tiefsten Ebene heißen **Blätter** des Baumes (leaves).

Ein **Pfad** ist eine Liste von unterschiedlichen Knoten, in der aufeinander folgende Knoten durch Kanten verbunden sind.

Zwischen der Wurzel des Baumes und jedem anderen Knoten gibt es nur genau einen Pfad.

Baum-Beispiel 1

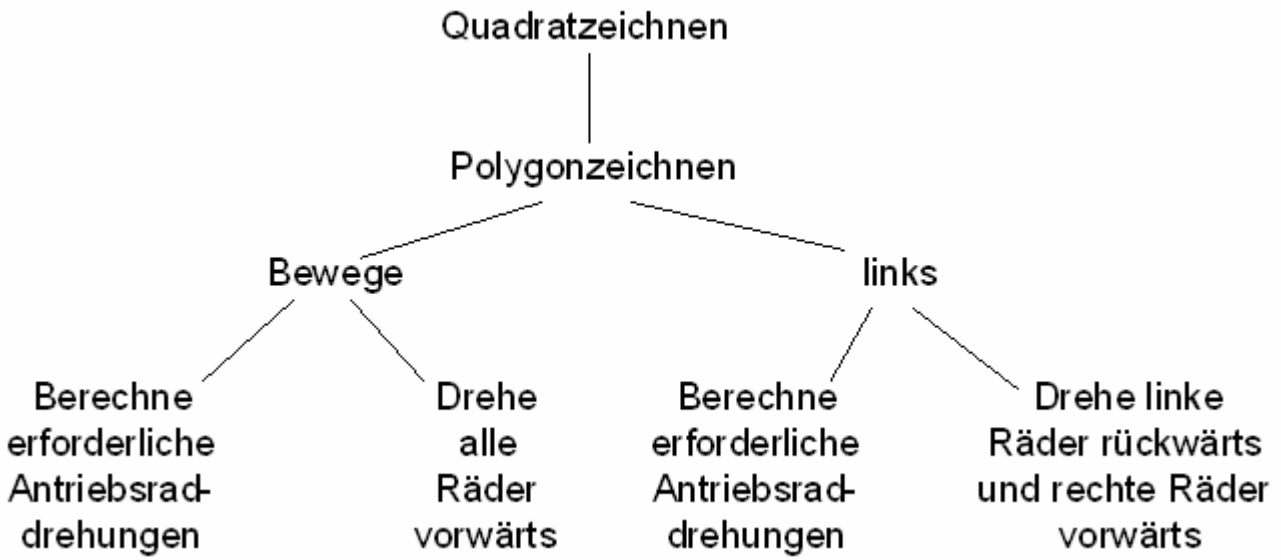
Organisationsstruktur eines Unternehmens



Die **Knoten** sind Datensätze, die jeweils einen Angestellten bezeichnen. Die **Kanten** bezeichnen die Vorgesetztenhierarchie.

Baum-Beispiel 2

Baumstruktur der Module eines Softwaresystems

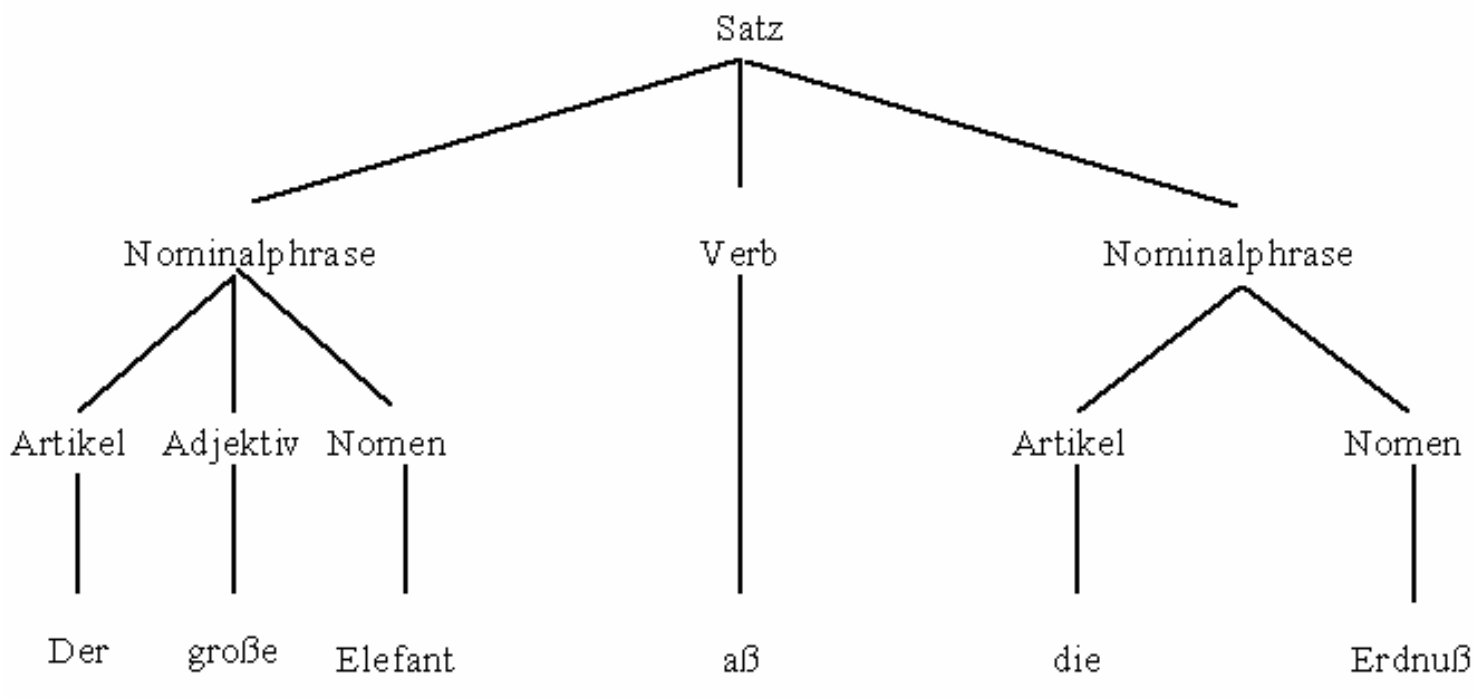


Die **Knoten** sind die Module.

Die **Kanten** beschreiben die Aufrufhierarchie.

Baum-Beispiel 3

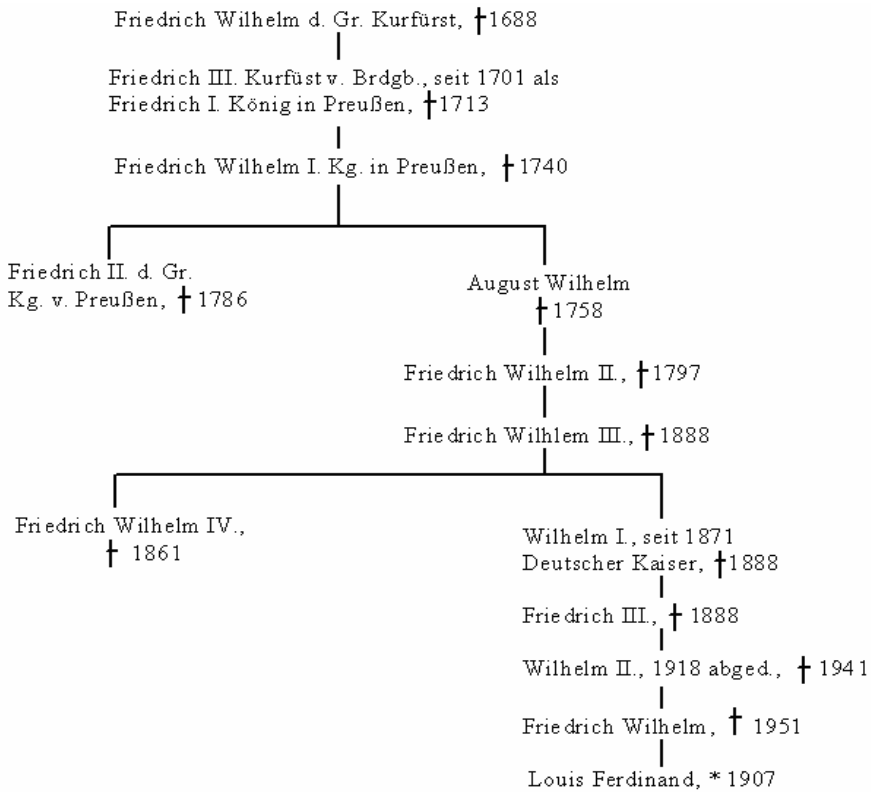
Struktur eines deutschen Satzes



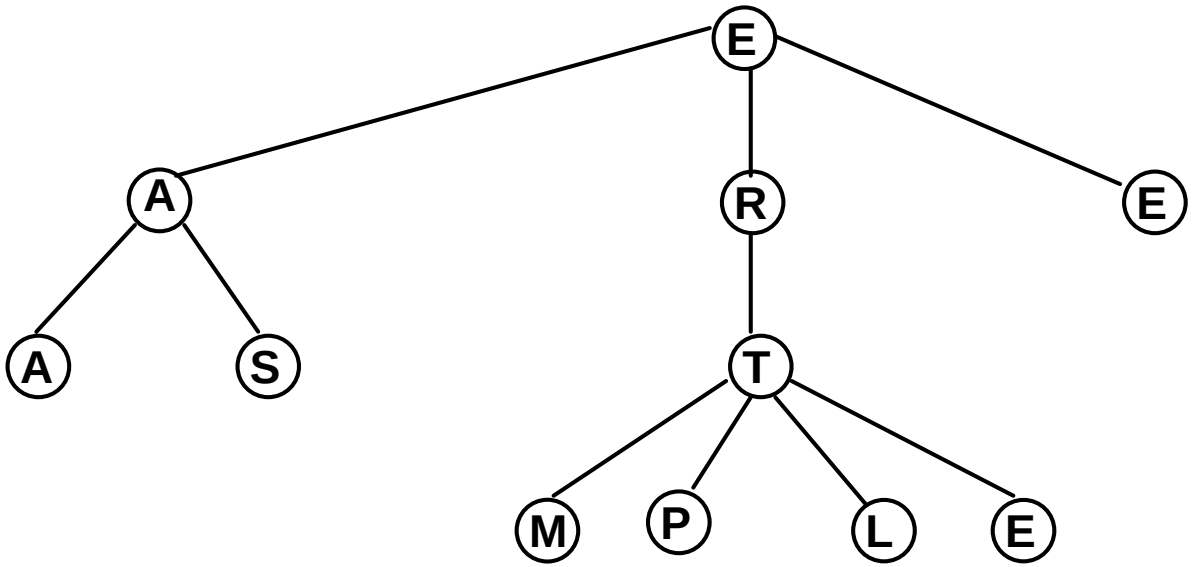
Baum-Beispiel 4

In vielen Bäumen ist die horizontale Anordnung der Knoten einer Ebene signifikant (d.h. eine Vertauschung ändert die Bedeutung). Ein Beispiel ist der Stammbaum:

Hohenzollern



Baum-Beispiel 5



Weitere Baum-Terminologie (1)

- Jeder Knoten (außer der Wurzel) hat genau einen **Elternknoten (Vaterknoten)**.
- Die mit ihm verbundenen Knoten sind seine **Kinder** oder **Söhne**.
- Ein Knoten, der keine Söhne hat, heißt **Blattknoten**.
- Jeder Knoten ist die Wurzel eines **Unterbaumes**.
- Ein Baum heißt vom **Rang** *d*, wenn jeder Knoten außer den Blattknoten maximal *d* Söhne hat.
- Jedem Knoten kann ein **Ebene** zugeordnet werden. Die Ebene ist definiert als die Anzahl der Knoten auf dem Pfad zur Wurzel.
- Die **Höhe** eines Baumes ist Anzahl der Ebenen.
- Die **Pfadlänge** eines Baumes ist die Summe aller Pfadlängen von einem Knoten zur Wurzel.

Weitere Baum-Terminologie (2)

- Ein **vollständiger** Baum ist ein Baum, bei dem alle Ebenen bis auf die unterste vollständig besetzt sind.

Anmerkung

In der Literatur findet sich auch eine abweichende Definition von *Rang:* Ein Baum heißt *vom Rang d*, wenn jeder Knoten außer den Blattknoten **genau** *d* Söhne hat.

Programmtechnische Darstellung von Bäumen

Problem: Jeder Knoten kann eine andere Anzahl von Kindern haben, maximal *d*. Deshalb können die Referenzen (Zeiger) auf die Kinder nicht effizient im Elternknoten gespeichert werden.

Lösung 1: Parent-link-Darstellung

- In jedem Knoten wird nur der Zeiger auf seinen Elternknoten gespeichert.
- Ist in der Regel nur sinnvoll, wenn der Baum nur von unten nach oben durchlaufen werden soll.

Lösung 2: Listendarstellung der Kinder

- Jeder Knoten hat zwei Zeiger:
  - einen zu seinem ganz linken Kind, falls es ein solches gibt
  - einen zu seinem rechten Bruder oder zum Elternknoten, falls er keinen rechten Bruder hat

Binärbäume

Definition

Ein Baum heißt **Binärbaum** genau dann, wenn er vom Rang 2 ist, d.h. wenn jeder Elternknoten maximal zwei Söhne hat.

Anmerkung

Im Gegensatz zu allgemeinen Bäumen sind Binärbäume maschinenintern leicht darzustellen: Es werden pro Knoten zwei Referenzen (bzw. Zeiger) fest eingerichtet.

Ordnungen von Bäumen

Eine **Ordnung** ist eine Abbildung eines Baumes auf eine lineare Struktur ("Plattklopfen des Baumes").

Man bildet eine eindeutigen Knotenfolge derart, dass jeder Knoten genau einmal darin vorkommt.

Bei Binärbäumen unterscheidet man vier wichtige Ordnungen:

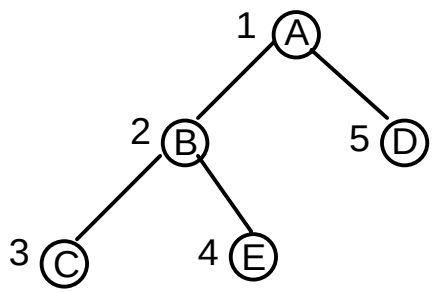
- Preorder
- Inorder
- Postorder
- Level-Order

Die “order“ gibt an, an welcher Stelle die Wurzel im Bezug zum linken und rechten Unterbaum vorkommen soll. Die Definition der “order“ ist rekursiv.

Preorder

Regel:

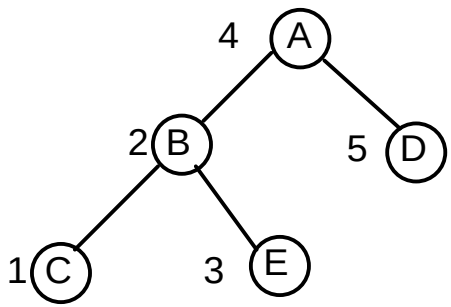
1. Wurzel
2. Linker Unterbaum
3. Rechter Unterbaum



Inorder

Regel:

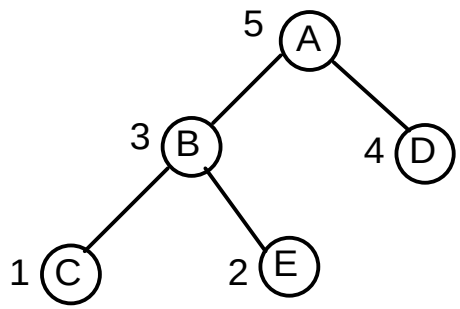
1. Linker Unterbaum
2. Wurzel
3. Rechter Unterbaum



Postorder

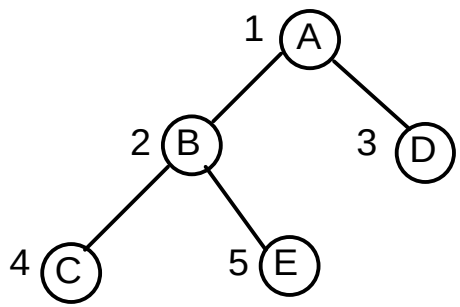
Regel:

1. Linker Unterbaum
2. Rechter Unterbaum
3. Wurzel



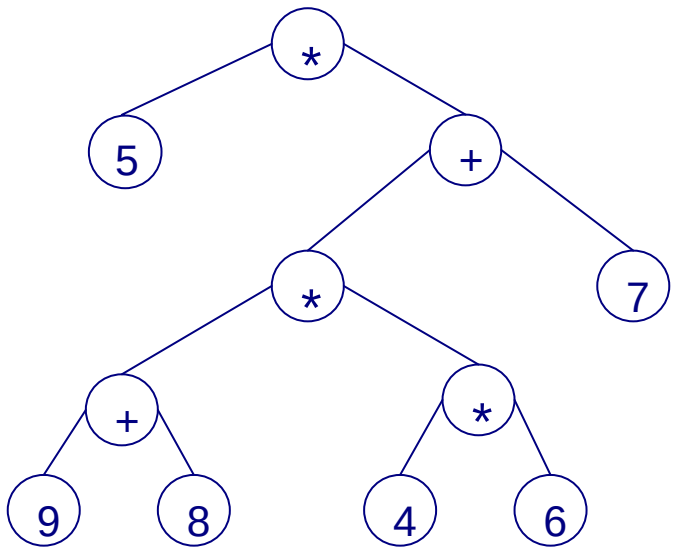
Level-Order

Regel:
1. Alle Knoten einer Ebene werden von links nach rechts besucht.
2. Alle Ebenen werden von oben nach unten behandelt.



Binärbaum, Beispiel 1: Operatorbaum

5 * (((9 + 8) * (4 * 6)) + 7)



Der Operatorbaum dient zur maschineninternen Darstellung von arithmetischen Ausdrücken. Eine Abarbeitung des Baumes in Postorder berechnet den Wert des Ausdrucks.

Beispiel 2: Binärbaum für Namen (1)

Jochen, Karin, Sepp, Franz, Bernd, Jim, Maria

Regel für den Aufbau des Baumes:

- Name kleiner als der aktuelle Knoten $\Rightarrow$ wird linker Sohn
- Name größer als der aktuelle Knoten $\Rightarrow$ wird rechter Sohn

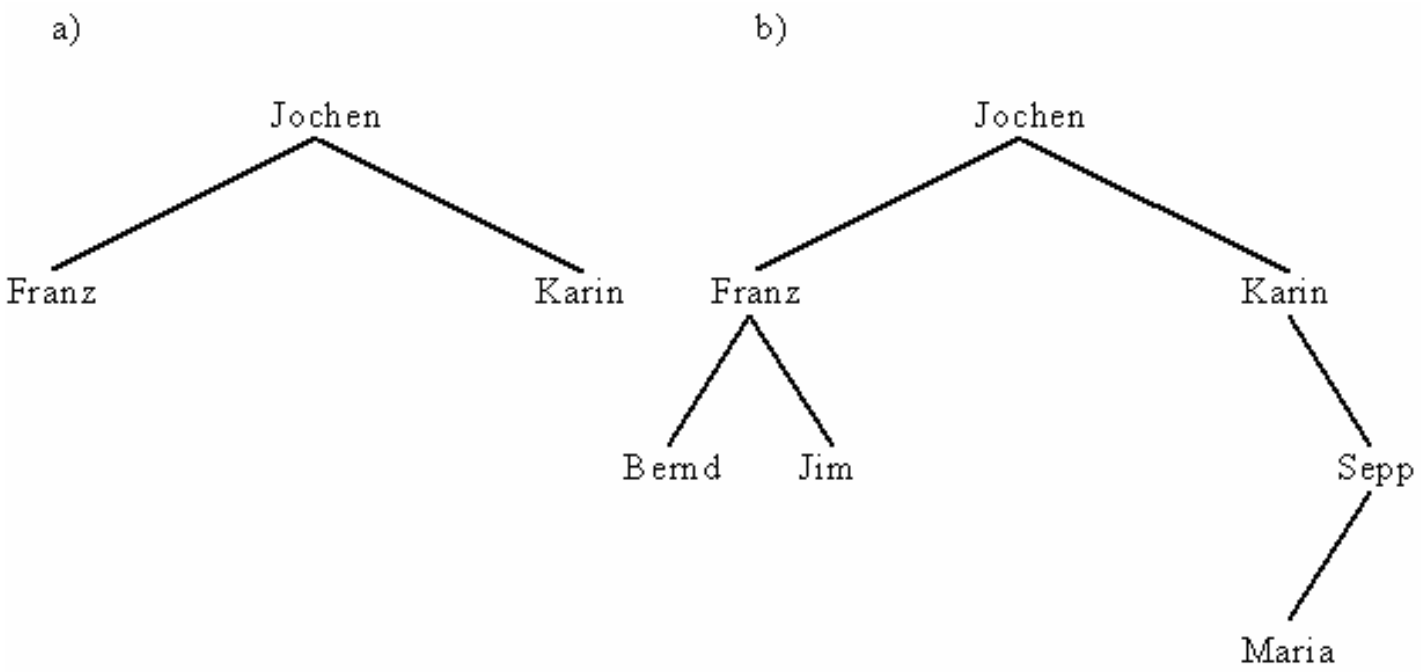
Regel für die Ausgabe ("**Tree Traversal**"):

linker Unterbaum, Wurzel, rechter Unterbaum

(in-order traversal)

Binärbaum für Namen (2)

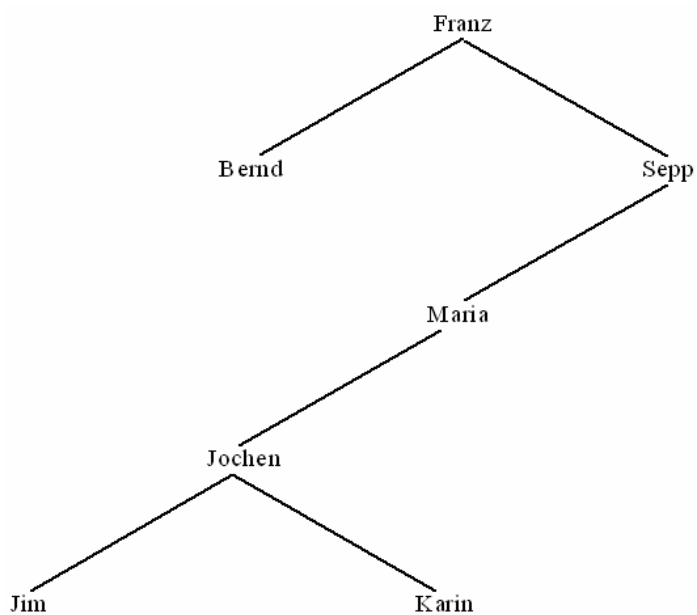
Zwei alphabetisch sortierte Binärbäume



Binärbaum für Namen (3)

Eine Eingabe der ursprünglichen Liste in anderer Reihenfolge führt zu einem anderen Binärbaum, der aber ebenfalls die korrekte Sortierfolge wiedergibt:

Franz, Bernd, Sepp, Maria, Jochen, Karin, Jim



Algorithmus "Sortiere durch Binärbaum" (1)

a) Grundidee

Baum-bilden

Baum-ausgeben

b) Erste Verfeinerung

Sei L die Eingabeliste, B der Binärbaum

Modul *Baum-bilden*(*L, B*)

// Baut aus der Eingabeliste L mit Namen einen sortierten Binärbaum B auf

Beginne mit dem Anfang von L

Solange *L nicht erschöpft ist* **führe aus**

Füge nächsten Namen in B ein

Algorithmus "Sortiere durch Binärbaum" (2)

Modul *Baum-ausgeben*(*B*)

// Gibt alle Knoten eines Binärbaumes B nach der Regel "links-vor-rechts"
// aus."

Falls *B nicht leer ist*

dann *Baum-ausgeben(linker Teilbaum von B)*

Schreibe Namen in der Wurzel von B nieder

Baum-ausgeben(rechter Teilbaum von B)

Modul *Sortiere(L)*

// Sortiert eine Liste L in alphabetischer Ordnung

Beginne mit einem leeren Baum, der mit B bezeichnet wird

Baum-bilden(L,B)

Baum-ausgeben(B)

Algorithmus "Sortiere durch Binärbaum" (3)

c) Zweite Verfeinerung

Der Modul Baum-bilden(L, B) wird durch Rekursion verfeinert:

Modul *Name-einfügen*(*Name, B*)

// Fügt *Name* in den sortierten Binärbaum B ein

Falls B *leer ist*

dann *erzeuge einen neuen Teilbaum mit Name als Wurzel*

sonst **falls** *Name vor dem Namen in Wurzel von B liegt*

dann *Name-einfügen(Name, linker Teilb. von B)*

sonst Name-*einfügen(Name, rechter Teilb. von B)*

Beachte: B ist hier Eingabe- und Ausgabeparameter.

Algorithmus "Sortiere durch Binärbaum" (4)

Nunmehr kann der Modul Baum-bilden geschrieben werden als:

Modul *Baum-bilden*(*L, B*)

// Baut aus der Eingabeliste L mit Namen einen sortierten Binärbaum B auf

Beginne mit dem Anfang von L

Solange *L nicht erschöpft ist* **führe aus**

Name-einfügen(nächster Name, B)

Algorithmus “Suche-im-Binärbaum“

Suche im Binärbaum nach einem Knoten, dessen Schlüssel gleich dem Suchschlüssel ist. Suche rekursiv.

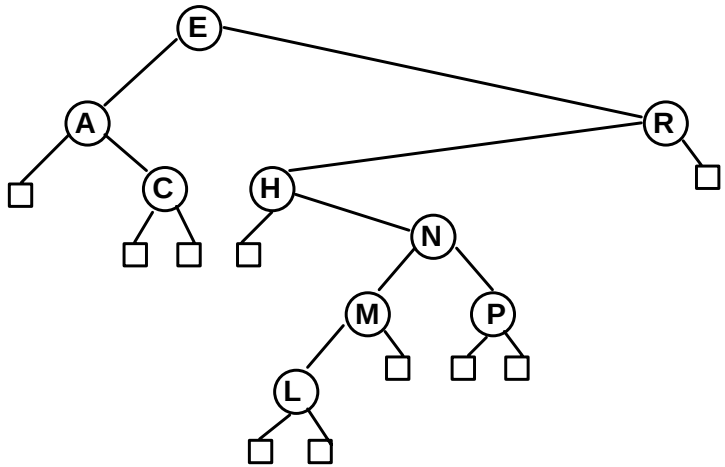
```
Beginne mit der Wurzel
Falls Suchschlüssel < Schlüssel des aktuellen Knotens:
      Suche im linken Teilbaum weiter
Falls Suchschlüssel = Schlüssel des aktuellen Knotens:
      fertig
Falls Suchschlüssel > Schlüssel des aktuellen Knotens:
      Suche im rechten Teilbaum weiter
```

Einfügen eines neuen Knotens in einen Binärbaum

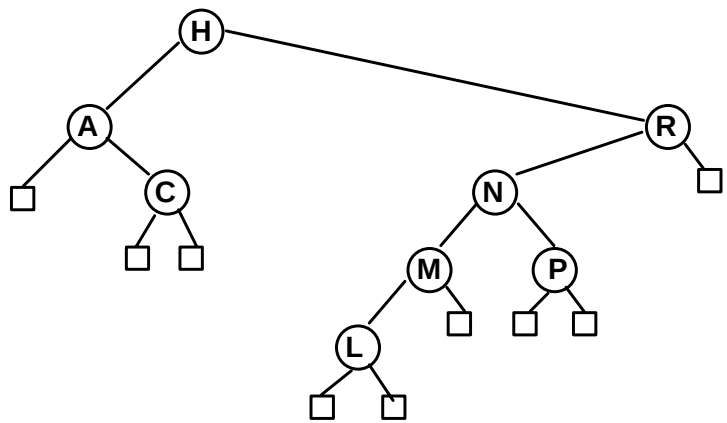
- Suche den Schlüssel des neuen Knotens (erfolglos) im Baum. Das bestimmt die Einfügestelle, d.h. den werdenden Vater.
- Setze den Zeiger des werdenden Vaters auf den neuen Knoten. Dieser wird stets ein neues Blatt sein.

Löschen im Binärbaum: leichte Fälle, schwere Fälle

Leicht: Löschen von L, A.



Schwer: Löschen von E



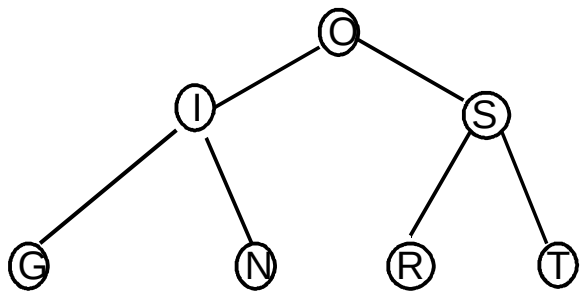
Löschen eines Knotens in einem binären Suchbaum

- Knoten hat keine Kinder $\Rightarrow$ entferne ihn
- Knoten hat nur ein Kind $\Rightarrow$ ersetze ihn durch sein Kind
- Knoten hat zwei Kinder, von denen eines selbst keine Kinder hat
  $\Rightarrow$ ersetze ihn durch dieses Kind
- Knoten hat zwei Kinder, die selbst beide ebenfalls Kinder haben
  $\Rightarrow$ ersetze ihn durch den Knoten mit dem nächst höheren Schlüssel aus dem rechten Teilbaum. Dazu muss in der Regel der rechte Teilbaum umorganisiert werden!

Beispiele zur Gestalt von Binärbäumen (1)

Schlüsselmenge: SORTING

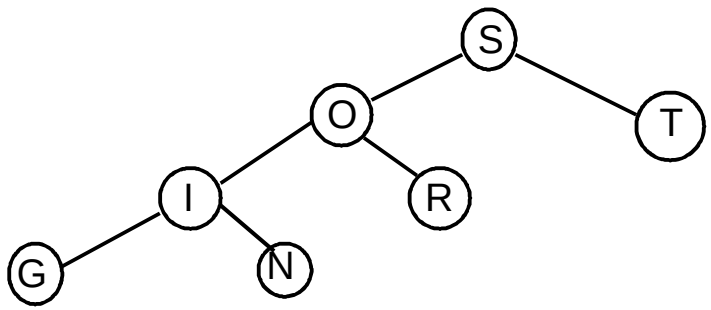
1. Vollständiger Binärbaum



Mittlere Anzahl von Vergleichen: 17/7 = 2,428 im Mittel

Beispiele zur Gestalt von Binärbäumen (2)

2. Ein natürlich gewachsener Binärbaum



1x 1 Vergleich

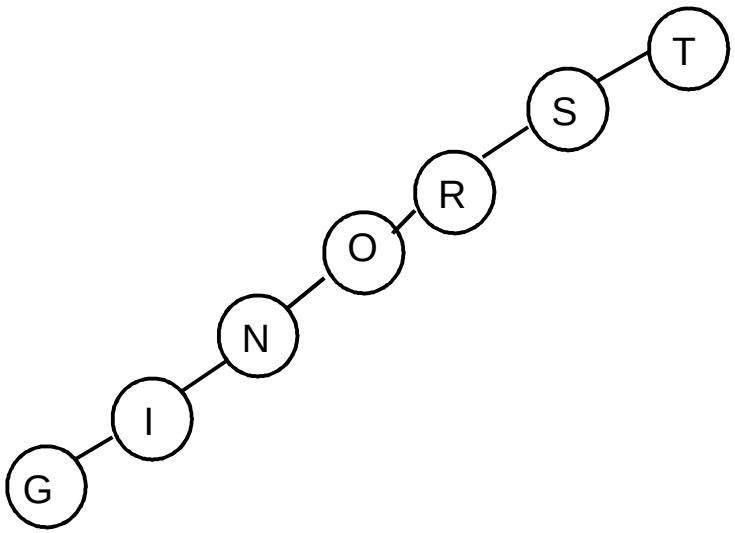
2x 2 Vergleiche

2x 3 Vergleiche

2x 4 Vergleiche ergibt 19/7 = 2,714 im Mittel

Beispiele zur Gestalt von Binärbäumen (3)

3. Der ungünstigste Fall



1x	1 Vergleich
1x	2 Vergleiche
1x	3 Vergleiche
1x	4 Vergleiche
1x	5 Vergleiche
1x	6 Vergleiche
1x	7 Vergleiche
28/7	= 4 im Mittel

"Gute" und "schlechte" Binärbäume

Für jeden Knoten im Baum ist die Anzahl der Vergleiche, die zum Finden benötigt wird, gleich seinem Abstand von der Wurzel. Die Summe dieser Abstände über alle Knoten heißt die **innere Pfadlänge** des Baumes.

Die **mittlere Pfadlänge** in einem Baum ergibt sich aus

(innere Pfadlänge)/*N*.

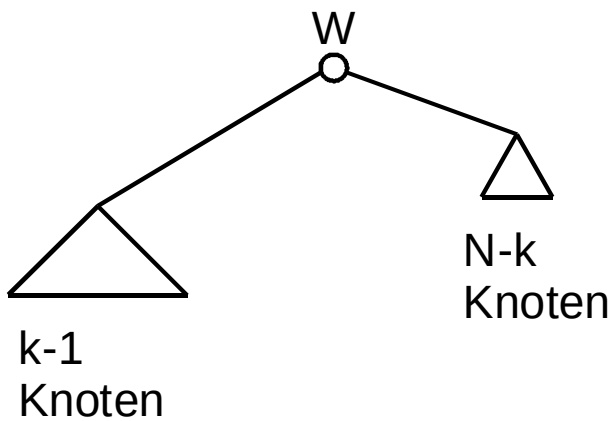
Sie kann rekursiv berechnet werden als

C_1=1 für einen Baum mit einem Knoten

$$C_N = N + \frac{1}{N} \sum_{1 \le k \le N} (C_{k-1} + C_{N-k})$$

Herleitung der mittleren Pfadlänge

Der betrachtete Baum aus *N* Knoten besteht aus der Wurzel, deren Schlüssel zuerst eingefügt wurde und der *k*-größte des Baumes ist, sowie dem linken und dem rechten Unterbaum der Wurzel, die *k*-1 bzw. *N*-*k* Knoten haben:



Die Wurzel fügt zu jeder anderen Pfadlänge 1 hinzu, für *N*-1 Restknoten also *N*-1. Bei zufällig eingefügten Schlüsseln ist die Wahrscheinlichkeit für jedes *k* gleich 1/*N*. Dazu kommt noch ein einziger Vergleich, falls die Wurzel selbst gesucht wird.

Mittlere Suchpfadlänge im vollständigen Binärbaum (1)

Ein vollständiger Binärbaum der Höhe *h* hat *N* = 2h-1 Knoten. Die mittlere Suchpfadlänge ist wieder die innere Pfadlänge dividiert durch *N*. Es kommen im Baum vor:

1x	1 Vergleich
2x	2 Vergleiche
4x	3 Vergleiche
8x	4 Vergleiche

Mittlere Suchpfadlänge im vollständigen Binärbaum (2)

Insgesamt ist die mittlere Pfadlänge also:

$$C_{N\min} = \frac{1}{N}\sum_{i=0}^{h-1}(i+1)*2^i ; \text{mit } N = 2^h - 1$$

$$= \frac{1}{2^h - 1}\left[(h-1)*2^h + 1\right]$$

Wegen h = $\log_2$ (N+1) ist also:

$$C_{N\min} = \frac{1}{2^h - 1} * \left[(h-1)(2^h - 1) + h\right]$$

$$= \log_2(N+1) + \frac{\log_2(N+1)}{N} - 1$$

$C_{N\min}$ ist also in der Größenordnung von $\log_2$ (N)

Vergleich "natürlicher" Binärbaum und vollständiger Binärbaum

Die mittlere Suchpfadlänge im vollständigen Binärbaum ist um ca. 38% kürzer als im zufällig gewachsenen Binärbaum.

"Entartete" Binärbäume können noch wesentlich längere Suchpfade haben. Es sind dann bis zu N Vergleiche erforderlich.

Problem 1

Es ist nicht immer $N = 2^i-1$, daher kann nicht immer ein vollständiger Binärbaum aufgebaut werden.

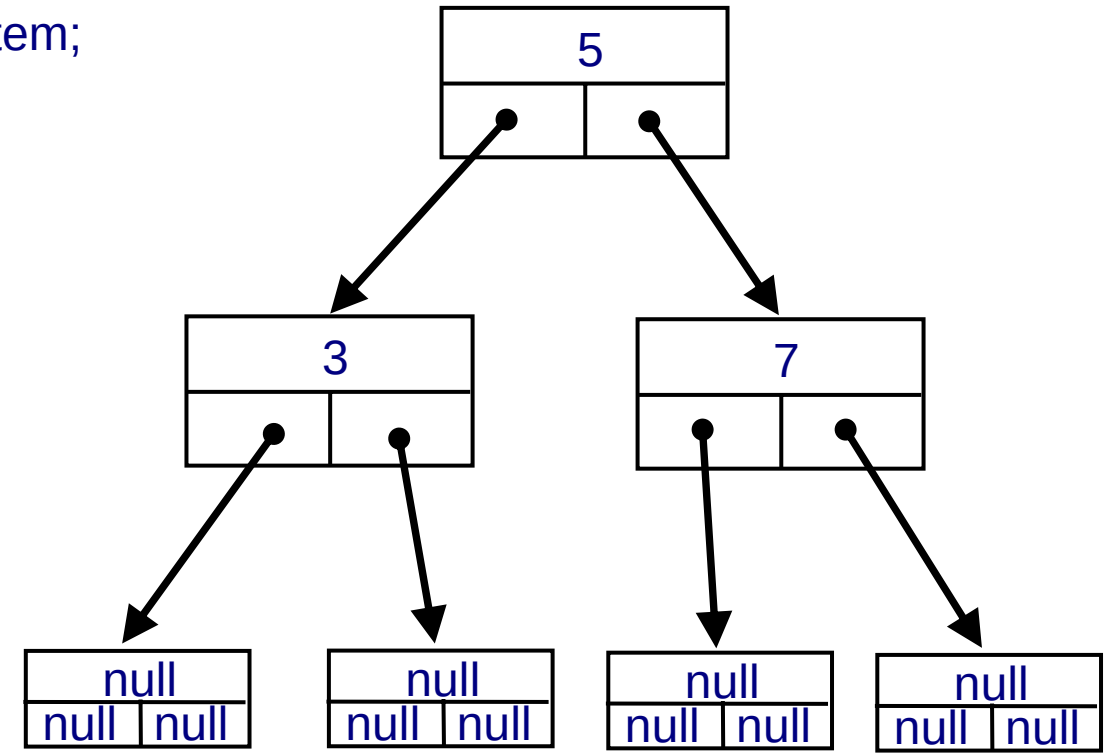
Problem 2

Eine vollständige **Balancierung** des Baumes erfordert **erheblichen** Aufwand bei Änderungsoperationen.

Man definiert daher spezielle Varianten von Binärbäumen, die einen guten Kompromiss zwischen effizientem zugriff beim Lesen und akzeptablem Aufwand beim Ändern darstellen. Solche Bäume sind Thema der weiterführenden Vorlesung „Algorithmen und Datenstrukturen“.

Binärbaum in Java

```
public class BinTree {
    private Comparable item;
    private BinTree left;
    private BinTree right;
    // …
}
```



Binärbaum in Java: Einfügen

```
public Comparable insert(Comparable elem) {
        if (item == null) {
            item = elem;
            left = new BinTree();
            right = new BinTree();
        } else
            ((item.compareTo(elem) > 0) ? left : right).insert(elem);
        return elem;
}
```

Binärbaum in Java: Suchen

```
public Comparable search(Comparable elem) {
        if (item == null)
            return null;
        int res = item.compareTo(elem);
        return (res == 0) ?
            item : ((res > 0) ? left : right).search(elem);
}

public String toString() {
        return (item == null) ? "" : (left.toString() + item.toString()
                                        + " " + right.toString());
}
```

Mehrwegbäume

Motivation

Die Länge eines Suchpfades im Baum ist proportional zur Höhe. Verwendet man statt Binärbäumen **Mehrwegbäume**, also Bäume von Rang $m > 2$, so ist die Höhe bei gleicher Knotenzahl N geringer und damit die Länge der Suchpfade kürzer. Dies ist bei der Speicherung auf externen Speichern (Magnetplatten) besonders wichtig, da so die Anzahl der Plattenzugriffe minimiert werden kann.

Beispiel

Ein Binärbaum für 1000 Knoten hat 10 Ebenen, ein Baum vom Rang 10 für 1000 Knoten hat 4 Ebenen.

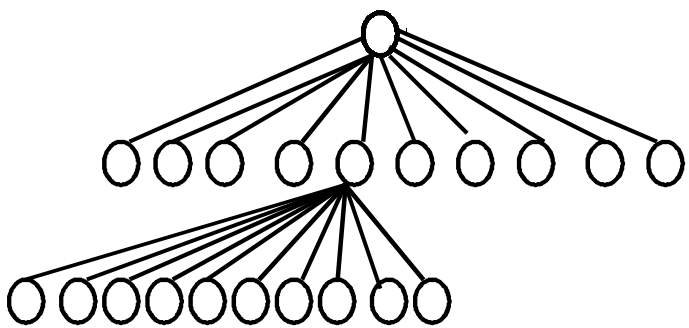
Suchbaum vom Rang 10

In jedem Knoten des Suchbaumes speichert man eine Liste der Form $<z_1, s_1,z_2,s_2, \ldots s_{m-1},z_m>$, wobei die z_i Zeiger (Verweise) sind und die s_i Suchschlüssel. Wie beim Binärbaum verweisen die linken Zeiger jeweils auf kleinere Werte, die rechten Zeiger auf größere Werte.

Im folgenden werden wir stets annehmen, dass alle gespeicherten Schlüssel voneinander verschieden sind (es gibt keine Duplikate).

Beispiel

Suchbaum vom Rang 10. Jeder Knoten speichert 9 Schlüssel und 10 Zeiger.



B-Bäume

Definition

Ein **B-Baum** vom Rang m ($m > 2$) ist ein Mehrwegbaum mit folgenden Eigenschaften:

1. Alle Blätter haben die gleiche Tiefe.
2. Jeder Knoten mit i Söhnen hat i -1 Schlüssel.
3. Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat wenigstens $\lceil m/2 \rceil$ Söhne.
4. Die Wurzel hat wenigstens 2 Söhne.
5. Jeder Knoten hat höchstens m Söhne.

Anzahl der Schlüssel in einem B-Baum (1)

Um die Anzahl der in einem B-Baum mit gegebener Höhe *h* gespeicherten Schlüssel abzuschätzen, genügt es also, die Anzahl seiner Blätter abzuschätzen.

Ein B-Baum der Ordnung *m* mit gegebener Höhe *h* hat die **minimale** Blattzahl, wenn seine Wurzel nur 2 und jeder andere innere Knoten nur *m*/2 Söhne hat. Daher ist die minimale Blattzahl

$$N_{\min} = 2 * \left\lceil \frac{m}{2} \right\rceil^{h-1}$$

Anzahl der Schlüssel in einem B-Baum (2)

Die Blattzahl wird **maximal**, wenn jeder innere Knoten die maximal mögliche Anzahl *m* von Söhnen hat. Daher ist die maximale Blattzahl

$$N_{\max} = m^h$$

Ist umgekehrt ein B-Baum mit *N* Schlüsseln gegeben, so hat er (*N*+1) Blätter, was man durch Induktion zeigen kann. Hat der Baum die Höhe *h*, so muss gelten:

$$N_{\min} = 2 * \left\lceil \frac{m}{2} \right\rceil^{h-1} \leq (N+1) \leq m^h = N_{\max}$$

Anzahl der Schlüssel in einem B-Baum (3)

Für die Höhe *h* ergibt sich:

$$h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right) \quad \text{und} \quad h \geq 1 + \log_m (N+1)$$

Wir haben also wieder die für balancierte Bäume typische Eigenschaft, dass die Höhe eines B-Baumes logarithmisch in der Anzahl der gespeicherten Schlüssel beschränkt ist. Da die Ordnung *m* eines B-Baumes in der Praxis etwa bei 100 bis 200 liegt, sind B-Bäume besonders niedrig.

Ist etwa *m* = 199, so haben B-Bäume mit bis zu 1999999 Schlüsseln höchstens die Höhe 4.

B-Bäume und externe Speicher

Besonders häufig werden B-Bäume zur Organisation von großen Datenmengen auf externen Speichern eingesetzt, typischerweise auf Magnetplatten. Die Schlüssel im Baum entsprechen Datenfeldern der Anwendung, zum Beispiel dem Namensfeld eines Datensatzes “Personalstammsatz“.

Ein Knoten im Baum entspricht einer Seite auf der Platte (= Block fester Größe; Einheit des Transfers im Dateisystem; typischerweise 512 Byte bis 4 Mbyte). Jeder Zugriff auf einen Knoten des Baumes entspricht also einem Ein-Ausgabe-Vorgang auf der Magnetplatte.

Suche in einem B-Baum

Der Algorithmus zur Suche eines Schlüssels in einem B-Baum ergibt sich durch sinngemäße Erweiterung des Suchalgorithmus für Binärbäume.

Die Suche **innerhalb eines Knotens** im B-Baum kann sequentiell erfolgen, aber auch durch binäre Suche, da die Schlüsselfelder sortiert sind und in der Regel feste Länge haben. Im Vergleich zu den Kosten eines Platten-zugriffs sind diese internen Suchkosten vernachlässigbar.

Einfügen in einen B-Baum

1. Suche im Baum nach dem neu einzufügenden Schlüssel. Die Suche endet erfolglos bei einem Blattknoten.
2. Falls der Knoten noch nicht die maximale Anzahl von *m*-1 Schlüsseln hat, füge den neuen Schlüssel ein.
3. Falls der Knoten bereits voll ist, zerteile ihn in zwei Knoten und ziehe den Teilerschlüssel in den Vater des aktuellen Knotens hoch. Die beiden entstandenen Teilknoten haben je (*m*-1)/2 Schlüssel. Füge den neuen Schlüssel dann an der richtigen Stelle ein.
4. Falls der Teilerschlüssel in den Vater hochgezogen wurde, mache weiter mit Schritt 2 für diesen Vater (rekursiv).

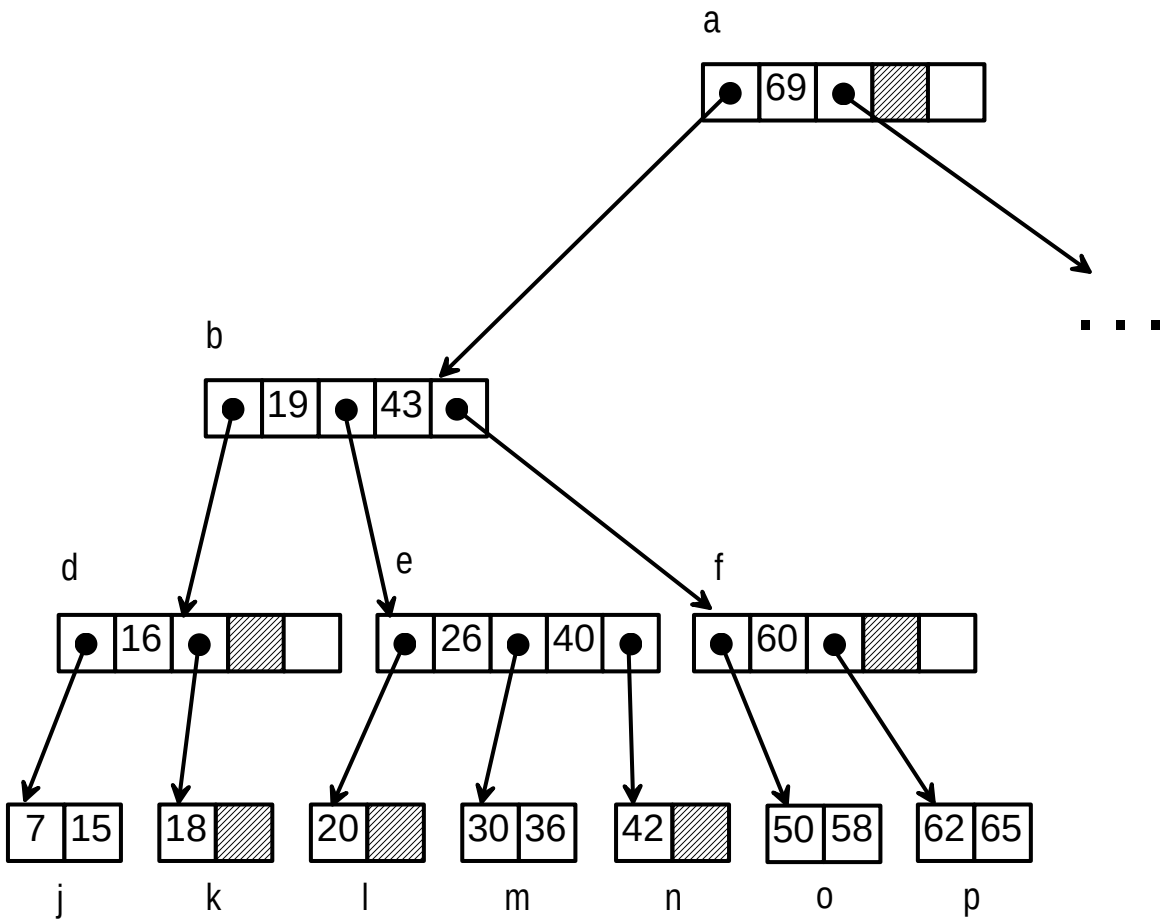
Löschen aus einem B-Baum

Zum Entfernen eines Schlüssels aus einem B-Baum der Ordnung *m* geht man umgekehrt vor. Man sucht den Schlüssel im Baum, entfernt ihn und verschmilzt gegebenenfalls einen Knoten mit einem Bruder, wenn er nach Entfernen eines Schlüssels "unterläuft", also weniger als *m*/2-1 Schlüssel enthält.

Ein Unterlauf der Wurzel, die ja nur einen Schlüssel gespeichert haben muss, bedeutet natürlich, dass die Wurzel keinen Schlüssel mehr speichert und nur noch einen einzigen Sohn hat. Man kann dann die Wurzel entfernen und den einzigen Sohn zur neuen Wurzel machen. Der Baum ist um eine Ebene niedriger geworden.

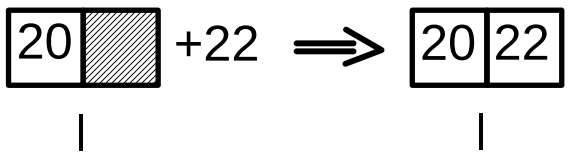
B-Baum-Beispiel

Gegeben sei ein B-Baum des Ranges 3 wie folgt:

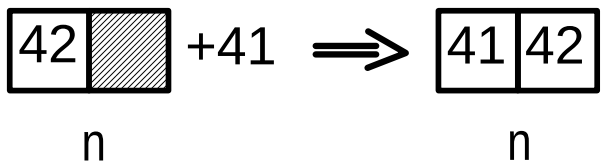


Einfügen von Schlüsseln in den B-Baum (1)

Einfügen des Schlüssels 22

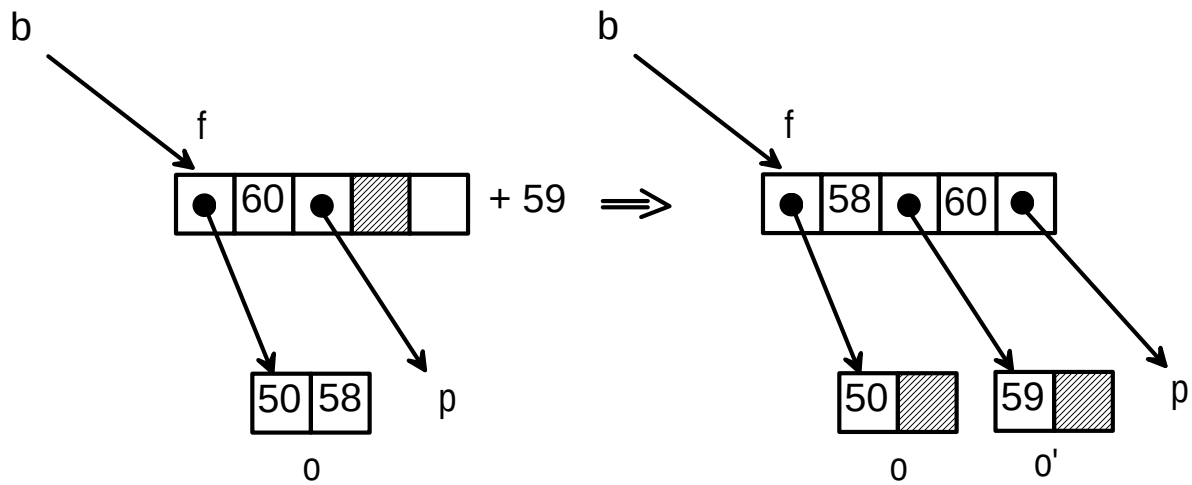


Einfügen des Schlüssels 41



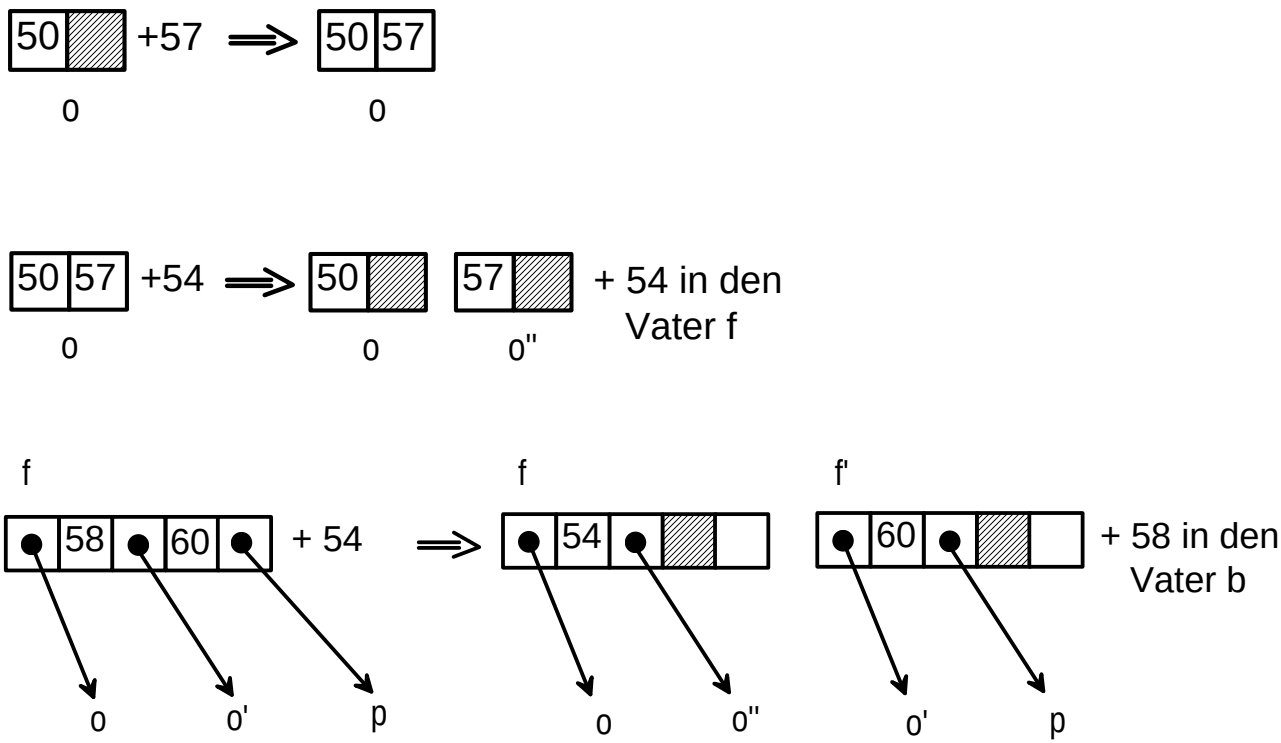
Einfügen von Schlüsseln in den B-Baum (2)

Einfügen der Schlüssels 59

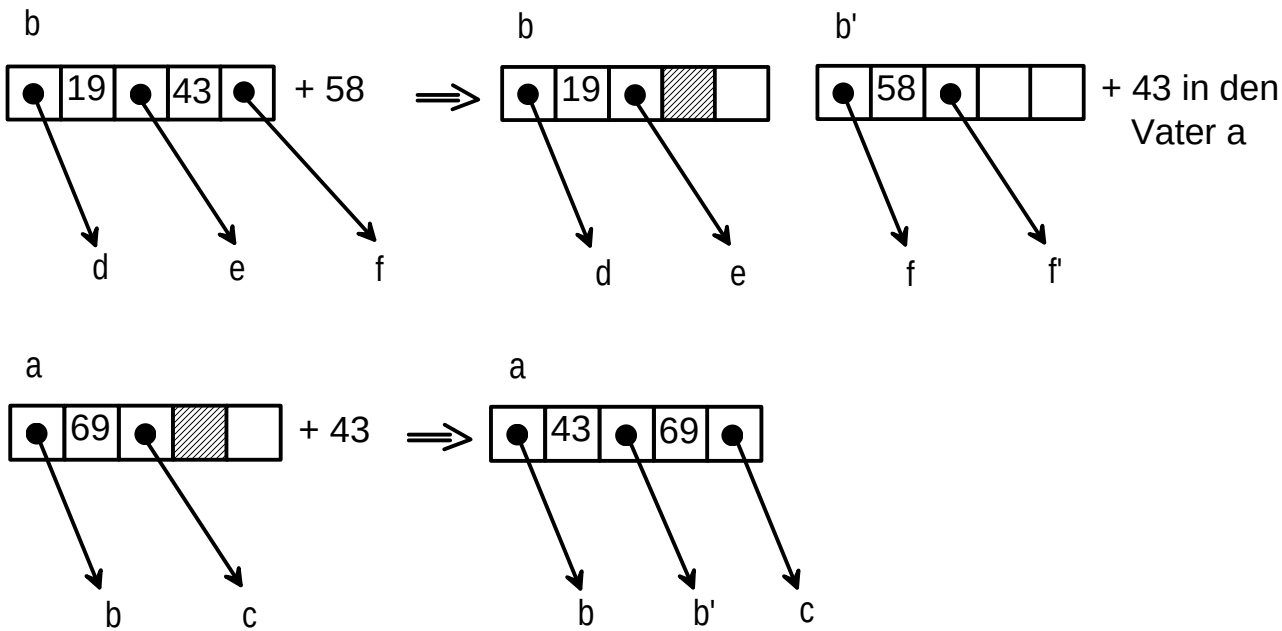


Einfügen von Schlüsseln in den B-Baum (3)

Einfügen des Schlüssels 57



Einfügen von Schlüsseln in den B-Baum (4)

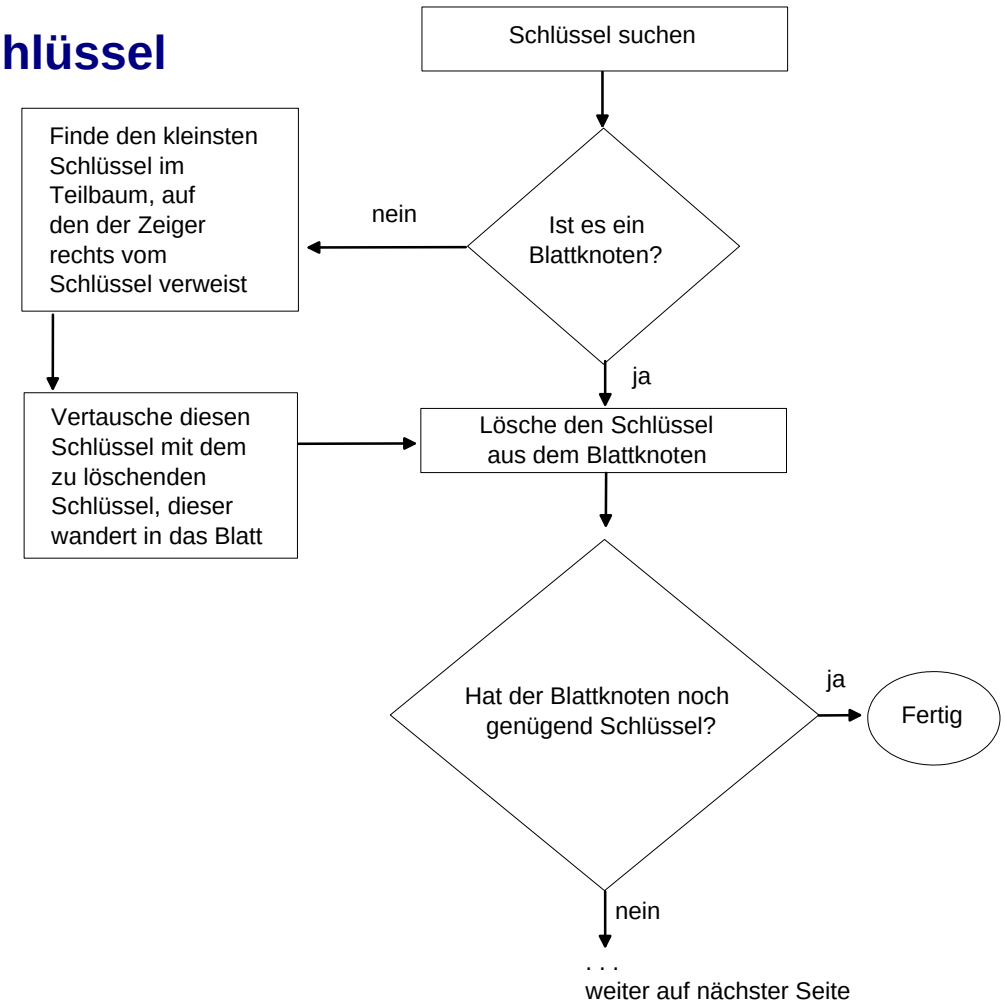


Übungsaufgabe

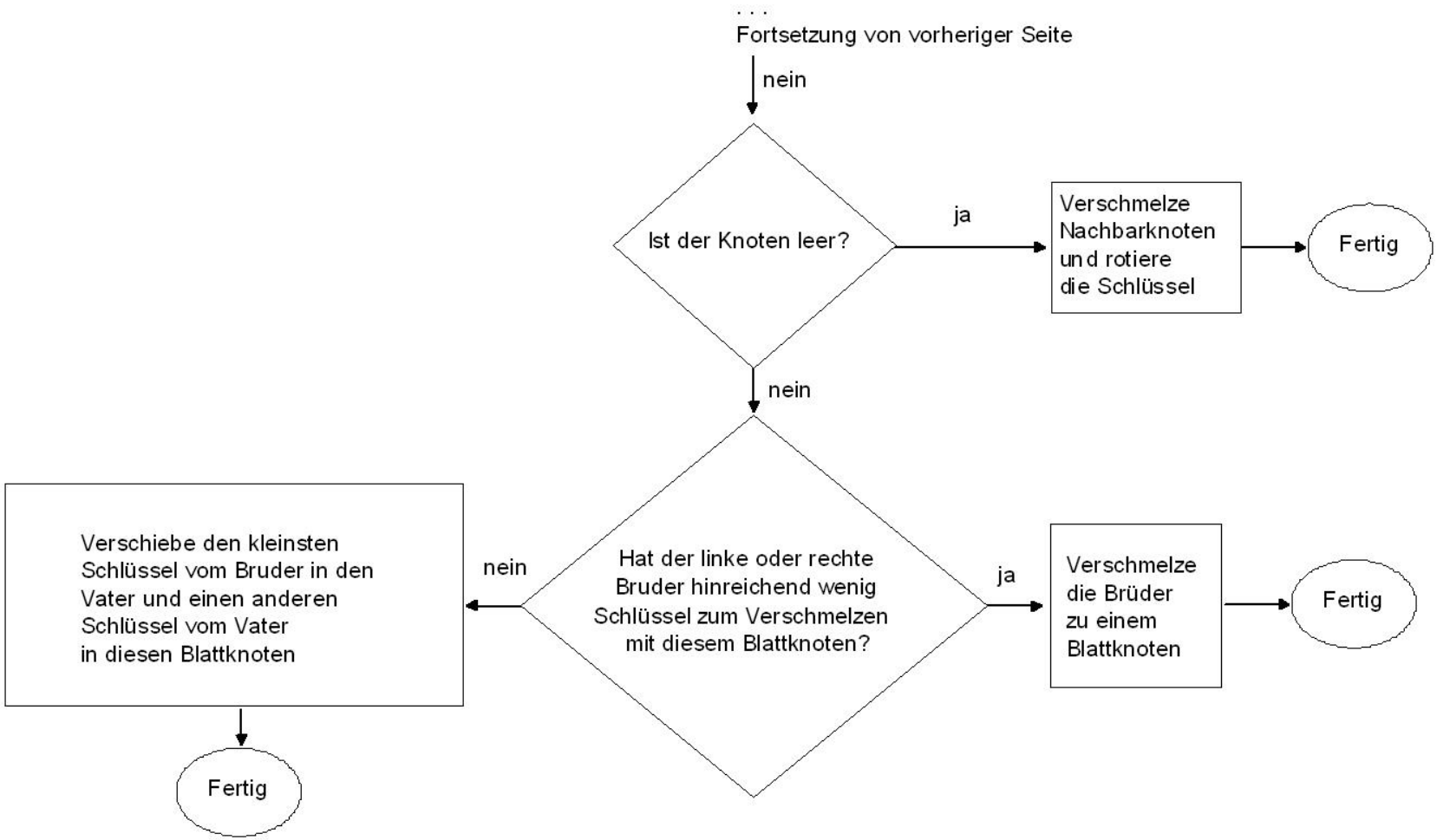
Fügen Sie weiterhin die Schlüssel 33, 75, 124, 122 und 123 in den B-Baum ein!

Löschen von Schlüsseln aus dem B-Baum (1)

Algorithmus Lösche-Schlüssel

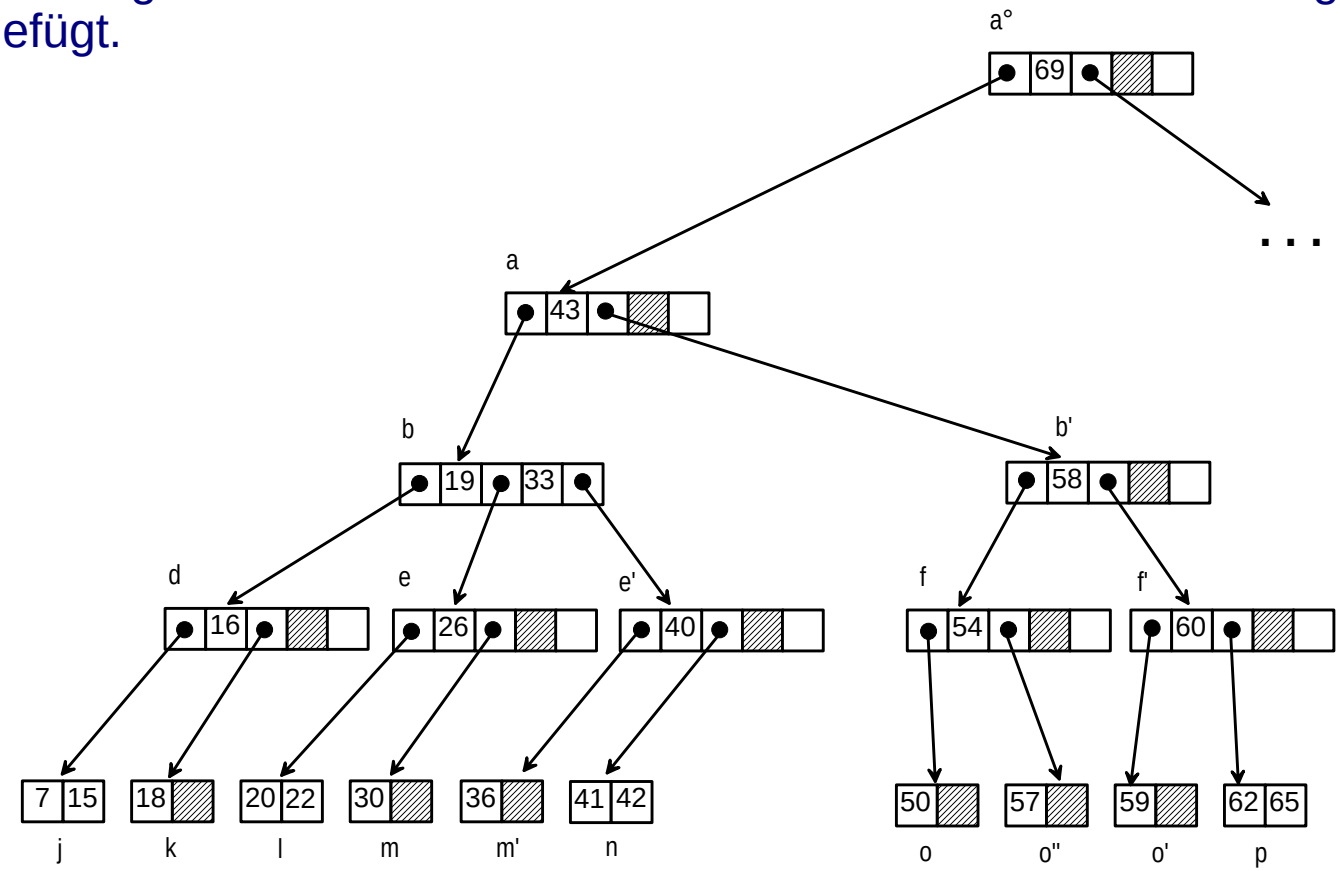


Löschen von Schlüsseln aus dem B-Baum (2)



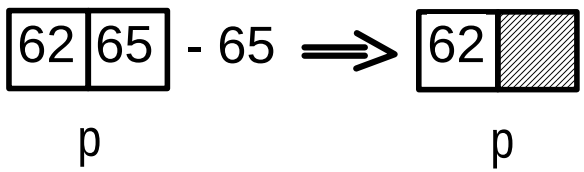
Lösch-Beispiel (1)

Anmerkung: Es wurden inzwischen auch die Schlüssel der Übungsaufgabe eingefügt.

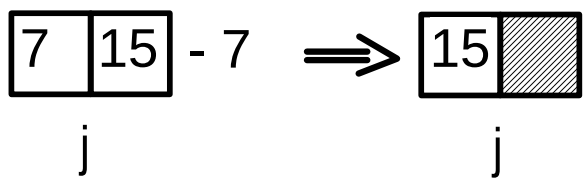


Lösch-Beispiel (2)

Löschen des Schlüssels 65

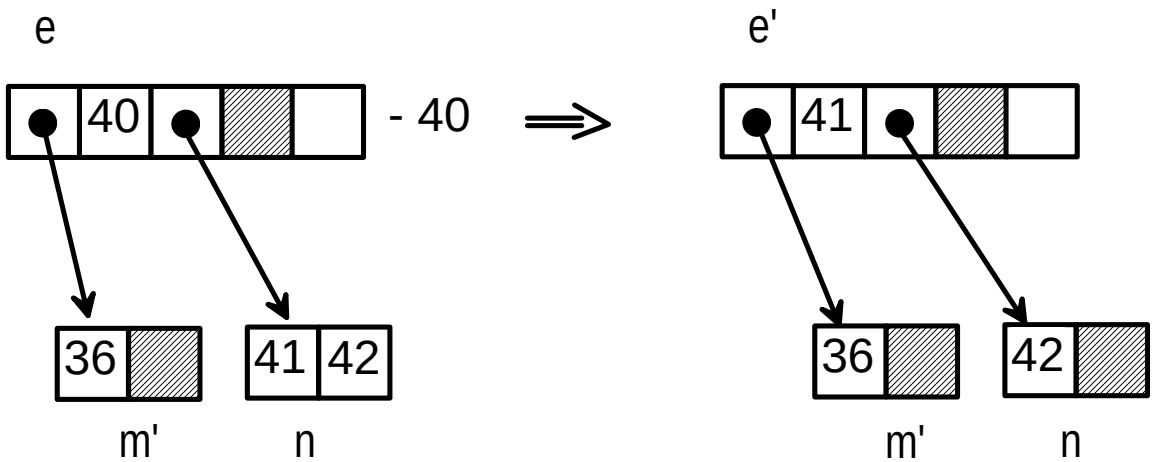


Löschen des Schlüssels 7



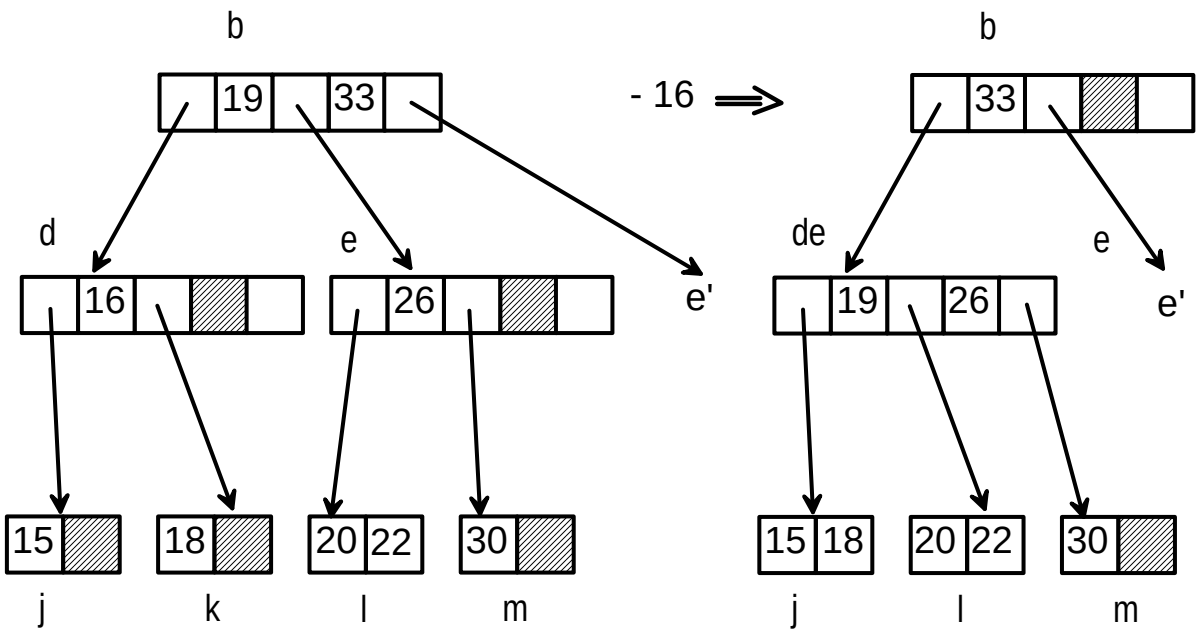
Lösch-Beispiel (3)

Löschen des Schlüssels 40



Lösch-Beispiel (4)

Löschen des Schlüssels 16



B-Bäume, Komplexität der Operationen (1)

Ein B-Baum der Ordnung *m* mit *N* Schlüsseln hat maximal die Höhe

$$\log_{\lceil \frac{m}{2} \rceil}(N+1)$$

Also gilt:

- Die **Suche** berührt je einen Knoten pro Ebene, im schlimmsten Fall also $\log_{\lceil \frac{m}{2} \rceil}(N+1)$ Knoten.

- Das **Einfügen** muss maximal pro Ebene ein Splitting durchführen; zur Suche kommen also im schlimmsten Fall nochmals größenordnungsmäßig $\log_{\lceil \frac{m}{2} \rceil}(N+1)$ Split-Operationen hinzu.

B-Bäume, Komplexität der Operationen (2)

- Das **Löschen** muss zusätzlich zur Suche maximal in der Größenordnung von $\log_{\lceil \frac{m}{2} \rceil}(N+1)$ Verschmelzungen durchführen.

Fazit

Alle Operationen im B-Baum erfordern größenordnungsmäßig $\log_{\lceil \frac{m}{2} \rceil}(N+1)$ Operationen!

Speicherplatzausnutzung von B-Bäumen (1)

Erwartungswerte für die in einem Knoten gespeicherte Schlüsselzahl eines B-Baumes vom Rang *m* und somit für die **Speicherplatzausnutzung** kann man analytisch berechnen.

Es ergibt sich, dass man (unabhängig von *m*) eine Speicherplatzausnutzung von ln2 = 69% erwarten kann, wenn man eine zufällig gewählte Folge von *N* Schlüsseln in den anfangs leeren B-Baum des Ranges *m* einfügt; die Knoten des entstehenden B-Baumes sind also nur zu gut 2/3 gefüllt.

Fügt man Schlüssel in auf- oder absteigend sortierter Reihenfolge in den anfangs leeren B-Baum ein, entstehen B-Bäume mit besonders schlechter Speicherplatzausnutzung.

Die Knoten sind (in allen Fällen, in denen $N = 2 * \left\lceil \frac{m}{2} \right\rceil^h$ ist) *minimal* gefüllt, d.h. die Wurzel hat nur einen und jeder andere innere Knoten nur $\left\lceil \frac{m}{2} \right\rceil - 1$ Schlüssel.

Speicherplatzausnutzung von B-Bäumen (2)

Es gibt verschiedene Vorschläge in der Literatur, diese schlechte Speicherplatzausnutzung von B-Bäumen zu vermeiden. Man kann zunächst die unmittelbaren oder gar alle Brüder eines überlaufenden Knotens daraufhin untersuchen, ob man ihnen nicht Schlüssel abgeben kann, bevor man den Knoten teilt und den mittleren Schlüssel und damit eventuell auch das Überlaufproblem auf das nächst höhere Niveau verschiebt.

Andere Vorschläge zielen darauf ab, für eine Folge bereits sortierter Schlüssel B-Bäume nicht durch wiederholtes Einfügen in den anfangs leeren Baum zu erzeugen, sondern möglichst optimale Anfangsstrukturen zu erzeugen in der Hoffnung, dass nachfolgende Einfügungen oder Entfernungen von Schlüsseln den Baum höchstens allmählich, d.h. für eine sehr große Zahl solcher Operationen, stärker vom Optimum abweichen lassen.