Einführung in die Informatik

Universität Stuttgart, Studienjahr 2005/06

Gliederung der Grundvorlesung

1. ~~Einführung in die Sprache Ada95~~
2. ~~Algorithmen und Sprachen~~
3. ~~Daten und ihre Strukturierung~~
4. ~~Begriffe der Programmierung~~
5. ~~Abstrakte Datentypen~~ Zurückgestellt
6. ~~Komplexität von Algorithmen und Programmen~~
7. ~~Semantik von Programmen~~
8. Suchen
9. Hashing
10. Sortieren
11. Graphalgorithmen
12. Speicherverwaltung

Stand: 1.5.06

8. Suchen / Vollversion

8.1 Suchen in sequentiellen Strukturen

8.2 Bäume und (binäre) Suchbäume

8.3 Optimale Suchbäume

8.4 Balancierte Bäume (insbesondere AVL-Bäume)

8.5 B-Bäume

8.6 Digitale Suchbäume (Tries)

8.7 Datenstrukturen mit Historie

Anhang: 8.8 Weitere Definitionen zu Graphen

8.9 Sonstiges

Ziel dieses 8. Kapitels:

Die meisten Probleme werden mit Mengen beschrieben und die Lösungsalgorithmen arbeiten auf Mengen. Dieses Kapitel stellt Ihnen einige Datenstrukturen (Felder, Bitvektoren, Suchbäume, optimale Suchbäume, AVL-, B- und digitale Bäume, Tries) vor, um Mengen darzustellen.

Am Ende sollen Sie gelernt haben, welche dieser Strukturen welche Eigenschaften besitzen. So sind (binäre) Suchbäume im Mittel gut zum Speichern und Wiederfinden von Elementen geeignet, will man jedoch eine logarithmische Such-, Einfüge- und Löschzeiten garantieren, so muss man spezielle Bäume verwenden (z.B. die höhenbalancierten Bäume). Deren Vor- und Nachteile lernen Sie soweit kennen, dass Sie in konkreten Anwendungen entscheiden können, welche Datenstruktur die günstigste ist.

Vorbemerkungen: Grundaufgabe des Suchens

Gegeben: Menge $A = \{a_1, ..., a_n\} \subseteq B$ sowie ein Element $b \in B$.

Realisiere A in einer geeigneten Datenstruktur, so dass die folgenden drei Operationen "effizient" durchgeführt werden:

- Entscheide, ob b in A liegt (und gib ggf. an, wo). **FIND**
- Füge b in A ein. **INSERT**
- Entferne b aus A. **DELETE**

Statt des meist sehr umfangreichen Elements b betrachten wir nur eine Komponente s von b, durch die b eindeutig bestimmt ist. Dieses s nennen wir "Suchelement" oder meistens "**Schlüssel**" (englisch: "**key**").

Weitere Operationen sind denkbar, zum Beispiel:

Weitere Aufgaben: Wähle eine Datenstruktur so, dass alle oder einige der folgenden Tätigkeiten für zwei Mengen $A_1, A_2 \subseteq B$ effizient durchführbar sind.

- Vereinige A_1 und A_2.	UNION
- Schneide A_1 und A_2.	INTERSECTION
- Bilde das Komplement $B \setminus A_1$.	Complement
- Entscheide, ob A_1 leer ist.	EMPTINESS
- Entscheide, ob $A_1 = A_2$ ist.	EQUALITY
- Entscheide, ob $A_1 \subseteq A_2$ ist.	SUBSET
- Entscheide, ob $A_1 \cap A_2$ leer ist.	Empty Intersection

Wir beschränken uns in diesem Kapitel aber auf die drei Operationen FIND, INSERT und DELETE.

Annahme: Die Mengen haben keine Struktur (insbesondere sind sie nicht geordnet). In diesem Fall muss man die die Elemente der Menge "wie sie kommen" in ein Feld oder eine Liste einfügen. Dies kennen wir bereits (vgl. 3.5).

Worst-Case-Aufwand der drei Operationen der Grundaufgabe mit Listen, wenn die Menge A genau n Elemente enthält:

FIND: Durchlauf durch die Auflistung, also O(n).

INSERT: Füge das neue Element am Anfang ein: O(1). (Elemente treten dann evtl. mehrfach in der Struktur auf. Will man dies nicht, dann: O(n).)

DELETE: Zunächst das Element finden, dann aus der Auflistung entfernen: O(n).

Hinweis: Ist die Gesamtmenge B klein, so lohnen sich Bitvektoren, siehe unten; hierfür muss man B in irgendeiner Weise anordnen.

Wir nehmen im Folgenden stets an, dass die zugrunde liegenden Mengen angeordnet sind.

Grund: Alle Elemente werden im Rechner durch eine Folge von Nullen und Einsen dargestellt. Dies impliziert eine (lexikografische) Anordnung, die wir stets ausnutzen können.

[Auch können wir eine Menge ohne Anordnung stets in irgendeiner beliebigen Weise ordnen. Wurde eine solche Ordnung festgelegt, so darf sie anschließend nicht mehr verändert werden.]

8.1 Suchen in "flachen" Strukturen

Flache oder sequentielle Strukturen sind üblicherweise
- (eindimensionale) Felder,
- Listen,
- Bitvektoren,

wobei egal ist, ob die Felder linear oder zyklisch sind und ob die Listen zusätzlich einfach oder doppelt verkettet werden.

In eindimensionalen **Feldern** sucht man nach einem Element s, indem man das Feld linear durchläuft. Ist das Feld geordnet, so kann man eine binäre Suche (Intervallschachtelung, siehe 6.5.1) durchführen. **Listen** werden prinzipiell von vorne nach hinten oder von hinten nach vorne durchsucht. Im Falle zyklischer Strukturen muss man sich mit einem Index oder einem Zeiger merken, ab wo die Suche begonnen wurde.

Struktur 1: Das array. *Durchsuchen eines Feldes:*

```
s: element; i: Integer; A: array (1..n) of element;
....; i:=1;
while i <= n and then A(i) /= s loop i := i+1; end loop;
if i <= n then < das gesuchte Element s steht an der Position i >
else  < s ist nicht im Feld A vorhanden > end if; ...
```

Mit einem Stopp-Element kann man eine Abfrage sparen:

```
s: element; i, n: Positive; A: array (1..n+1) of element;
....; i:=1; A(n+1) := s;        -- Stopp-Element setzen!
while A(i) /= s loop i := i+1; end loop;
if i <= n then < das gesuchte Element s steht an der Position i >
else  < s ist nicht im Feld A vorhanden > end if; ...
```

Struktur 2: Die Liste. *Durchsuchen einer linearen Liste:*

Suche s in einer Liste, auf die die Variable Anker zeigt:

```
p := Anker;
while p /= null and then p.inhalt /= s loop
                    p := p.next;  end loop;
if p = null then < s ist nicht in der Liste enthalten >
else < p verweist auf das erste Element mit Inhalt s > end if;
```

Aufwand:

Zeit: Die lineare Suche erfordert O(n) Vergleichsschritte, dauert also relativ lange. Im Falle binärer Suche (geordnetes Feld) benötigt man maximal O(log(n)) Vergleiche. Eine genaue Analyse hierzu erfolgte bereits in Abschnitt 6.5.1, wobei dort zwei Programme im Detail untersucht wurden.
Platz: Es sind nur wenige Speicherplätze zusätzlich erforderlich.

Wie sieht es mit den Operationen der Grundaufgabe aus?

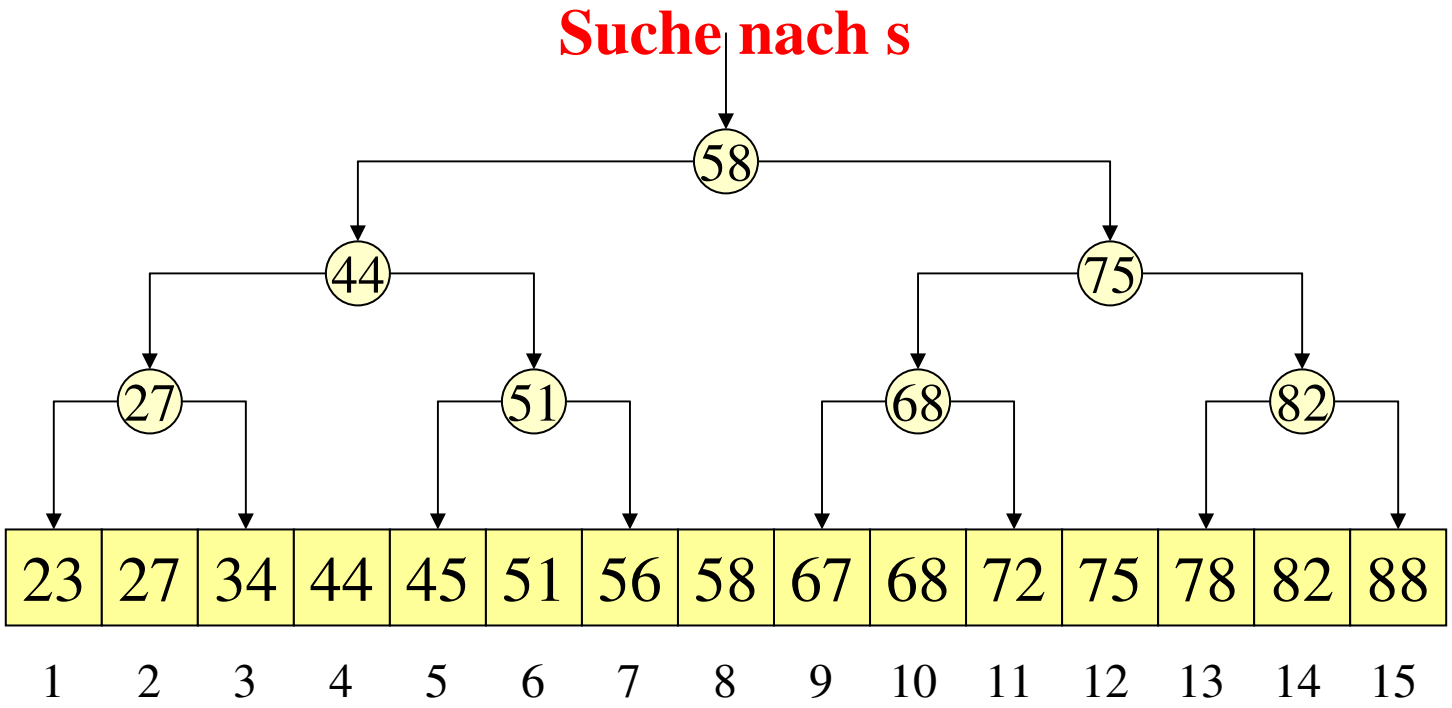
Wir wählen ein array als Datenstruktur, in das die Menge der Größe nach geordnet eingetragen wird. Dann

- dauert FIND nur O(log(n)) Schritte, sofern man ein geordnetes array verwendet und hierauf mit der Intervall-schachtelung sucht,
- dauert INSERT aber O(n) Schritte, da beim Einfügen alle größeren Elemente um eine Komponente nach hinten verschoben werden müssen,
- dauert DELETE ebenfalls O(n) Schritte, da beim Löschen alle größeren Elemente um eine Komponente nach vorne verschoben werden müssen.

Darstellung als Liste: Ähnlich wie beim array, alles dauert hier O(n) Schritte, sofern man beim Einfügen keine doppelten Elemente zulässt (Details selbst ausführen).

Erinnerung an Abschnitt 6.5.1, Darstellung durch ein array:

Hier ist $n=15=2^4-1$:



Die Tiefe dieses Index-Baums, der über dem array liegt, beträgt hier 4, so dass man spätestens nach 4 Schritten (allgemein: spätestens nach log(n) Schritten) die Suche beenden kann.

Lässt sich eine der Operationen noch beschleunigen? Unter der folgenden Bedingung, ja, für die Operation FIND.

Beachte: Bei der Intervallschachtelung (= binäre Suche) halbieren wir jeweils das gesamte noch verbleibende Feld A(links..rechts). Als Teilungsindex "Mitte" wählen wir: Mitte = (rechts + links) / 2 = links + (rechts - links) / 2.

Verbesserung: Kann man mit den Schlüsseln "rechnen" und sind die Schlüssel recht gleichmäßig über den Indexbereich verteilt, so kann man den ungefähren Bereich, wo ein Schlüssel s im Feld A(links..rechts) liegen muss, genauer angeben durch den folgenden Teilungsindex

$$p = \text{links} + \frac{s - A(\text{links})}{A(\text{rechts}) - A(\text{links})} (\text{rechts} - \text{links}).$$

So geht man beispielsweise beim Suchen in einem Lexikon vor.

Man kann zeigen, dass mit dieser "Interpolationssuche" die Operation FIND bei einem geordneten Feld nur noch den uniformen Zeitaufwand O(log(log(n))) benötigt. Falls die Schlüssel gleichverteilt sind, so braucht man für den gesamten Suchprozess nur mit $1 \cdot \log(\log(n)) + 1$ Schritten zu rechnen.

Da log(log(n)) eine sehr schwach wachsende Funktion ist, sollte man die Interpolationssuche einsetzen, wo immer es möglich ist. (In der Praxis liegt log(log(n)) fast immer unterhalb von 10.)

Struktur 3: Darstellung der Menge durch Bitvektoren

Da $A = \{a_1, ..., a_n\} \subseteq B = \{b_1, ..., b_r\}$ eine Teilmenge ist (mit $a_i \leq a_j$ und $b_i \leq b_j$ für $i < j$), kann man A auch durch einen Bitvektor $x = (x_1, ..., x_r)$, mit $x_i \in \{0, 1\}$, der Länge r darstellen, wobei für alle i gilt: $x_i = 1 \Leftrightarrow b_i \in A$.

Wird nach dem Element $s \in B$ gesucht und kennt man die Nummer, die s in der Anordnung von B trägt (also dasjenige i mit $s = b_i$), dann gilt für eine Teilmenge A mit Bitvektor x:

FIND: $s = b_i \in A \Leftrightarrow x_i = 1$.

INSERT: Setze $x_i := 1$.

DELETE: Setze $x_i := 0$.

Im uniformen Komplexitätsmaß läuft dann alles in O(1), also in konstanter Zeit ab!

Dennoch verwendet man diese Darstellung mit Bitvektoren nur selten, weil in der Praxis B meist sehr groß (wenn nicht sogar unendlich groß) ist. Auch benötigen alle Operationen der "weiteren Aufgaben" (UNION, INTERSECTION usw.) den Aufwand O(r), während man in der Praxis höchstens auf O(n) oder $O(n^2)$ kommen darf (n = Größe der betrachteten Teilmenge von B, während $r = |B|$ ist). Oft lässt sich auch das r gar nicht bestimmen, z.B. wenn man die Menge aller Namen zugrunde legt.

Ist dagegen die Kardinalität $r = |B|$ relativ klein, dann sind Bitvektoren eine geeignete und vor allem eine leicht zu implementierende Darstellung für Mengen; auch die üblichen Mengenoperationen wie Vereinigung, Durchschnitt usw. lassen sich hiermit leicht realisieren. (Selbst durchdenken!)

8.2 Bäume und (binäre) Suchbäume

Wiederholen Sie bitte die Abschnitte 2.6 und 3.7 über Bäume (bzw. über Ableitungsbäume in kontextfreien Grammatiken/ BNF). Dort wurden die Begriffe Baum, Wurzel, Vorgänger, direkter Vorgänger, Blatt, Höhe usw. erläutert.

Wir stellen diese Begriffe nochmals auf den folgenden Folien zusammen. Wir setzen voraus, dass die Begriffe ungerichteter und gerichteter Graph, Nachbar (im ungerichteten Fall) bzw. Vorgänger und Nachfolger (im gerichteten Fall), Weg in einem Graphen, Kreis (oder Zyklus), azyklischer Graph, Zusammenhang und Zusammenhangskomponente bekannt sind. Die auf Graphen bezogenen Definitionen finden Sie in Abschnitt 3.8. Weiterführende Definitionen sind in Abschnitt 8.8 zusammengefasst.

Definition 8.2.1: Es sei $G=(V,E)$ ein Graph, $|V|=n\geq 0$.

(1) Ein Knoten $w\in V$ heißt Wurzel von G, wenn es von w zu *jedem* Knoten des Graphen einen Weg gibt (im gerichteten Fall muss der Weg natürlich auch gerichtet sein).

(2) Der ungerichtete Graph G = (V, E) heißt Baum, wenn er azyklisch und zusammenhängend ist (insbesondere ist dann jeder Knoten des Baums auch Wurzel).

(3) Der gerichtete Graph G = (V, E) heißt Baum, wenn er eine Wurzel w besitzt, die keinen Vorgänger hat, und jeder Knoten ungleich der Wurzel genau einen Vorgänger besitzt, d.h., zu jedem $x\in V$, $x\neq w$ gibt es genau einen Knoten y mit $(y,x)\in E$, und es gibt kein $u\in V$ mit $(u,w)\in E$.

(4) Ein Graph heißt Wald, wenn alle seine (schwachen) Zusammenhangskomponenten Bäume sind.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(a) Es sei $G=(V,E)$ ein ungerichteter Baum. Wähle irgendeinen Knoten w als Wurzel aus und ersetze jede ungerichtete Kante $\{x,y\}$ durch die gerichtete Kante (x,y), wobei x näher an der Wurzel liegt als y, d.h., die Richtung zeigt stets von der Wurzel weg. So erhält man aus G in eindeutiger Weise den gerichteten Baum $G'=(V,E')$ mit Wurzel w.
Anwendung für die Programmierung: Man kann einen Baum stets als gerichteten Baum auffassen/implementieren.

(b) Es sei $G'=(V,E')$ ein gerichteter Baum mit Wurzel w. Ersetze jede gerichtete Kante (x,y) durch die ungerichtete Kante $\{x,y\}$, so erhält man aus G' in eindeutiger Weise den ungerichteten Baum $G=(V,E)$.

Folgerung (a) begründet, warum wir Bäume mit Hilfe von Zeigern darstellen. Fügt man für jeden Knoten noch einen Inhalt hinzu, so erhält man folgende Ada-Darstellung; hierbei ist MaxG der maximale Ausgangsgrad des Baums:

```
MaxG: constant Positive := ...;
type Grad is 1..MaxG;
type Element is ...;
type Baum;  type Ref_Baum is access Baum;
type Baum (ausgangsgrad: Grad := MaxG) is record
        Inhalt: Element;
        Nachf: array (1..ausgangsgrad) of Ref_Baum;
end record;
```

Hier hat jeder Knoten mindestens einen Nachfolger. Knoten ohne Nachfolger müssen daher einen null-Zeiger erhalten.

Folgerung (Fortsetzung):

(c) In einem ungerichteten Baum gibt es von jedem Knoten zu jedem Knoten *genau* einen doppelpunktfreien Weg.

(d) In jedem gerichteten Baum gibt es zu jedem Knoten $x \in V$ *genau* einen Weg von der Wurzel w nach x.

(e) Wenn es in einem gerichteten Baum einen Weg vom Knoten x zum Knoten y gibt, dann führt der Weg von der Wurzel w nach y über den Knoten x.

Definition 8.2.2: Es sei $G=(V,E)$ ein Graph.

(1) Zu jedem Knoten x eines ungerichteten Graphen $G=(V,E)$ heißt $N(x)=\{y \mid \{x,y\} \in E\}$ die Menge der *Nachbarn* von x. Ihre Anzahl $|N(x)|$ heißt *Grad des Knotens* x. Der maximale Knotengrad heißt *Grad des Graphen* G.

(2) Zu jedem Knoten x eines gerichteten Graphen $G=(V,E)$ heißt $Vor(x)=\{y \mid (y,x) \in E\}$ die Menge der *Vorgänger* von x und $Nach(x)=\{y \mid (x,y) \in E\}$ die Menge der *Nachfolger* von x. Ihre Anzahlen $|Vor(x)|$ bzw. $|Nach(x)|$ heißen *Eingangsgrad* bzw. *Ausgangsgrad des Knotens* x. Der jeweils maximale Grad heißt *Eingangsgrad* bzw. *Ausgangsgrad des Graphen* G.

Definition 8.2.2 (Fortsetzung): Es sei $G=(V,E)$ ein Baum.

(3) Ist G ein gerichteter Baum mit Wurzel w, so ist $|Vor(x)|=1$ für alle Knoten $x \neq w$ und $|Vor(w)|=0$. Der eindeutig bestimmte Knoten vorg(x) = $y \in Vor(x)$ heißt direkter Vorgänger oder Elternknoten oder Vaterknoten von x. Jeder Knoten, der auf dem Weg von der Wurzel w nach x liegt, heißt Vorfahr von x (aber nicht x selbst). Verschiedene Knoten, die den gleichen direkten Vorgänger besitzen, heißen Geschwister oder Brüder. Jeder Knoten $x \in Nach(y)$ heißt direkter Nachfolger oder Kind oder Sohn von x.

(4) Gerichteter Baum: Ein Knoten x mit $x \neq w$ und mit $|Nach(x)|=0$ heißt Blatt des Baums.
Ungerichteter Baum: Ein Knoten x mit $x \neq w$ und mit $|N(x)|=1$ heißt Blatt des Baums.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(f) Es sei $G=(V,E)$ ein Baum mit Wurzel w. Der von einem Knoten x in G erzeugte Unterbaum (oder Teilbaum) ist $G_x = (V_x, E_x)$ mit
$V_x = \{y \mid$ jeder doppelpunktfreie Weg von w nach y führt über x$\}$,
$E_x = E|_{V_x}$ (= alle Kanten zwischen Knoten aus V_x).
Offenbar ist G_x ein Baum mit Wurzel x. Beachte, dass G_x nicht leer ist, da stets $x \in G_x$ gilt. Speziell ist $G_w = G$.

(g) Wenn es in einem gerichteten Baum einen Weg vom Knoten x zum Knoten y gibt, so liegt y in dem von x erzeugten Unterbaum.

Folgerung: Überzeugen Sie sich von folgenden Aussagen:

(h) Wenn G ein Baum mit n Knoten ist, so besitzt G genau n-1 Kanten (für n>0).

(i) Jeder Baum lässt sich mit zwei Farben färben, d.h., es gibt eine Abbildung f: $V \to \{1,2\}$ mit $f(x) \neq f(y)$ für alle Kanten {x,y} bzw. (x,y).

Bäume sind 2-färbbar. Definition hierzu: Sei $k \in \mathbb{N}$.
Ein beliebiger Graph G=(V,E) lässt sich mit k Farben färben (man sagt auch, G ist k-färbbar), wenn eine Abbildung

$$f: V \to \{1,2,...,k\}$$

existiert mit $f(x) \neq f(y)$ für alle Kanten {x,y} bzw. (x,y). Die minimale Zahl k, so dass sich G mit k Farben färben lässt, heißt Färbungszahl von G; sie zu bestimmen, heißt "Färbungsproblem". Diese Zahl lässt sich nach heutiger Kenntnis für beliebige Graphen nur mit großem Zeitaufwand berechnen.

Definition 8.2.3: Es sei $G=(V,E)$ ein Baum.

(1) Ist für jeden Knoten x die Menge N(x) bzw. im gerichteten Fall die Menge Nach(x) geordnet (d.h., die Knoten y_i der Menge N(x) bzw. Nach(x) sind angeordnet: $y_1 < y_2 < ... < y_k$), dann heißt G ein geordneter Baum.

(2) Es sei $k \in \mathbb{N}$ eine positive Zahl. Sei $G=(V,E)$ ein Baum mit $0 \notin V$. Dieser Baum zusammen mit einer Abbildung
$\nu: V \times \{1,...,k\} \to V \cup \{0\}$, so dass für alle $x \in V$ gilt:
$\text{Nach}(x) \subseteq \{\nu(x,i) \mid i = 1,...,k\}$,
aus $\nu(x,i) \neq 0$, $\nu(x,j) \neq 0$ und $i \neq j$ folgt $\nu(x,i) \neq \nu(x,j)$,
heißt k-närer Baum.
(Die direkten Nachfolger stehen also in einem k-stelligen Vektor, wobei Komponenten mit leerem Unterbaum - durch 0 bezeichnet - auftreten dürfen/müssen, sofern der Ausgangsgrad von x kleiner als k ist.)

Speziell: Im Fall $k=2$ heißt der Baum binär oder Binärbaum.

Definition 8.2.4: Es sei $G=(V,E)$ ein Baum mit Wurzel w.

(1) Die Anzahl der Knoten in einem längsten Weg von der Wurzel zu einem Blatt heißt die Tiefe des Baumes G. (Dies ist also die Länge des längsten Weges im Baum, der von w ausgeht, plus 1.)

(2) Zu jedem Baum mit Wurzel w gehört die Levelfunktion level: $V \to \mathbb{N}_0$, rekursiv definiert durch

level(w) = 1 und

level(x) = level(vorg(x)) + 1 für $x \neq w$.

Hinweis: Das maximale Level kann offenbar nur von einem Blatt angenommen werden. Die Tiefe des Baumes ist das maximale Level eines Knotens x im Baum:

Tiefe von G = $\text{Max}\{\text{level}(x) \mid x \in V\}$.

Weiterhin wird auch der leere Baum mit Tiefe 0 zugelassen.

Hinweis: Statt "Tiefe" verwendet man auch das Wort Höhe. Achten Sie in der Literatur genau auf die Definition; teilweise werden auch die um 1 verringerten Werte als Höhe oder Tiefe definiert.

Wir haben die Bäume "wie üblich" definiert. Allerdings haben wir nichts über den Grenzfall $V = \emptyset$ gesagt. Diesen Fall wollen wir zulassen, da wir es regelmäßig mit leeren Unterbäumen zu tun haben werden.
Die rekursive Definition von Bäumen umfasst diesen Fall unmittelbar. Der Vollständigkeit halber definieren wir daher Bäume nochmals auf diese Weise.

8.2.5: Rekursive Definition

Man kann k-näre Bäume leicht rekursiv definieren; sei $k \in \mathbb{N}$:

1) Die leere Menge ist ein Baum.

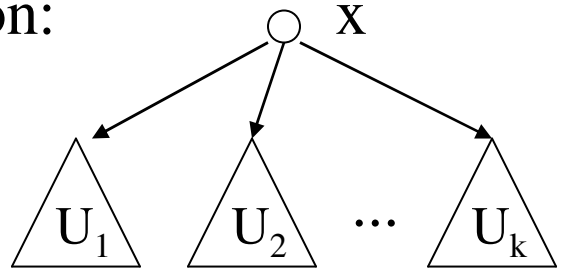
2) Wenn x ein Knoten und $U = \{U_1, U_2, ..., U_k\}$ eine geordnete Menge von k Bäumen ist, so ist auch $x(U)$ ein Baum.

x bildet die Wurzel des Baums $x(U)$, die Elemente von U sind Unterbäume oder Teilbäume im Baum $x(U)$.

Skizze: Leerer Baum: $\varnothing$ Tiefe dieses Baumes = 0

gerichtet oder ungerichtet

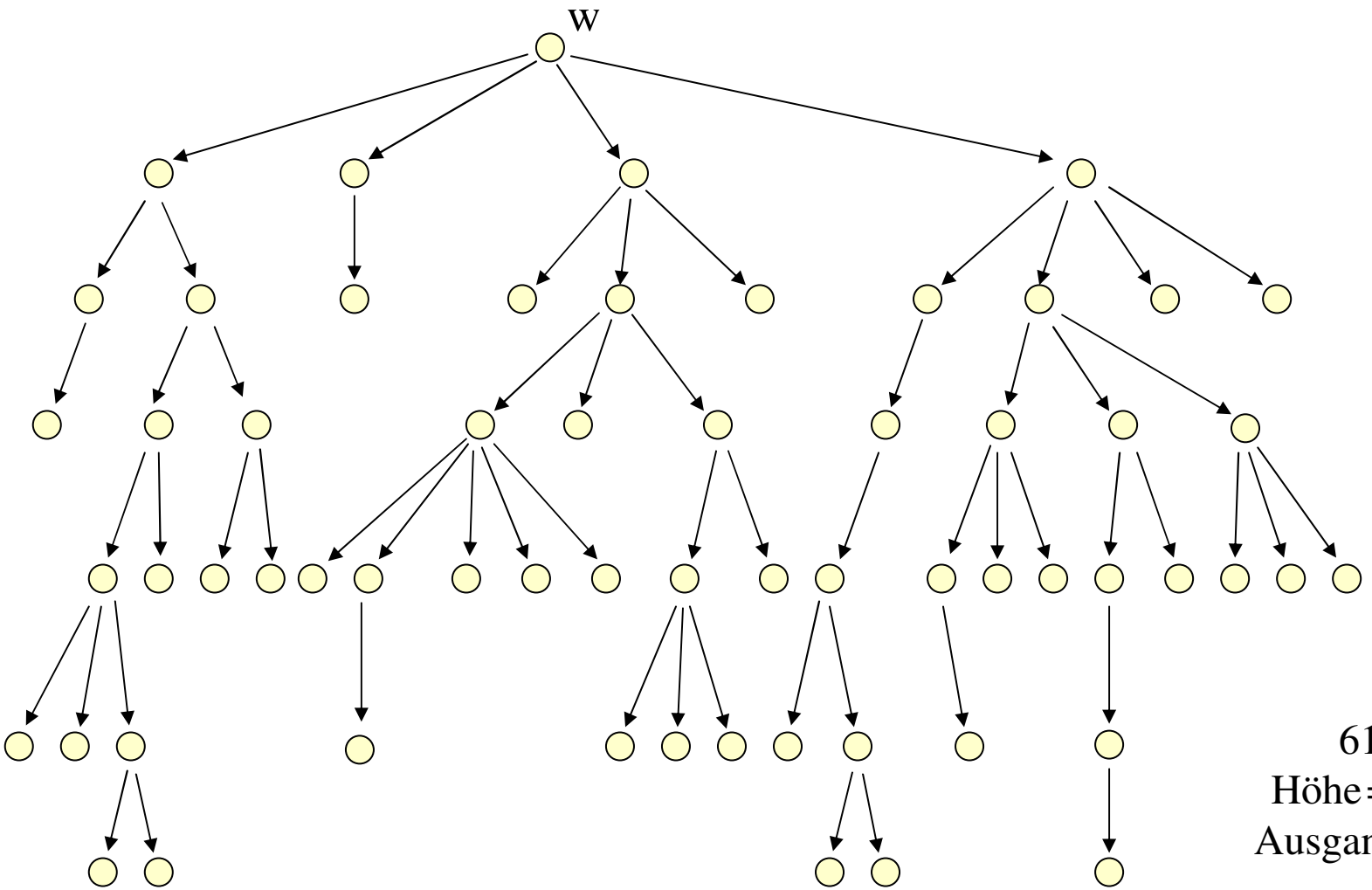
Rekursion:



Tiefe dieses Baumes = Max (Tiefe eines U_i) + 1.

Die k Nachfolger von x sind hier geordnet.

Beispiel für einen "beliebigen geordneten Baum":



61 Knoten,
Höhe = Tiefe 7,
Ausgangsgrad 5

Spezialfall: Binäre Bäume

Binäre Bäume sind also gerichtete und geordnete Bäume, bei denen jeder Knoten genau einen linken und einen rechten Nachfolger besitzt (diese Nachfolger können auch leer sein; *wichtig ist, dass der linke und der rechte Nachfolger stets unterschieden werden*).

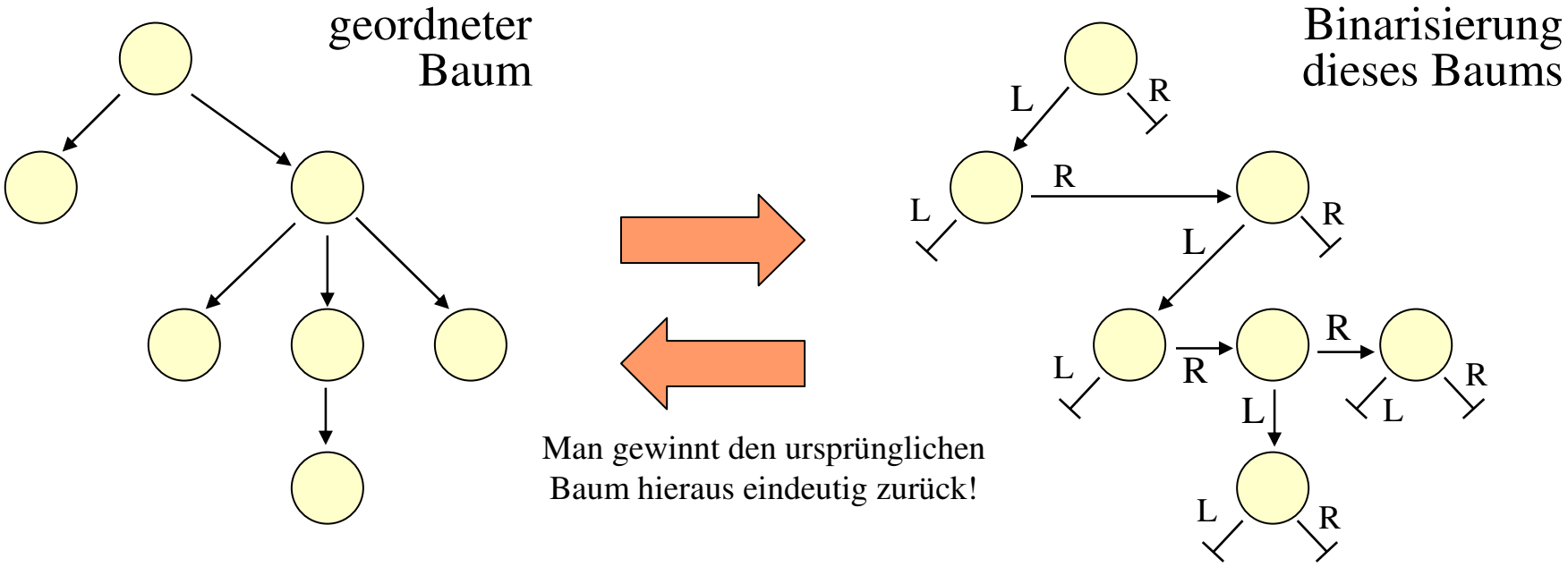
Die Darstellung binärer Bäume in Ada lautet bekanntlich (hier: mit dem Inhalt vom Typ Integer):

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        Inhalt: Integer;
        L, R: Ref_BinBaum;
end record;
```

8.2.6 Binarisierung von beliebigen geordneten Bäumen

Jeder geordnete Baum lässt sich eindeutig in einen binären Baum umwandeln, indem

- der linke Zeiger L stets auf das erste Kind und
- der rechte Zeiger R stets auf den nächsten Geschwisterknoten zeigt.



Folgerung aus dieser eindeutigen Umwandlung:

Es sei C_n die Anzahl aller binärer Bäume mit n Knoten.

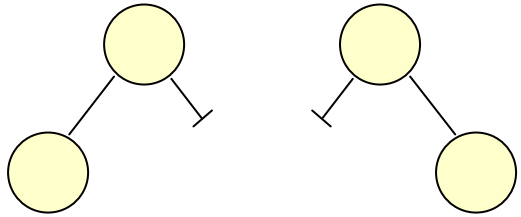
Es sei B_n die Anzahl aller geordneter Bäume mit n Knoten.

Dann gilt: $B_n = C_{n-1}$ für alle $n > 0$.

Vielleicht stutzen Sie hier, weil die binären Bäume doch eigentlich eine Teilmenge der geordneten Bäume sein müssten?! *Dies stimmt aber nicht,* weil geordnete Bäume nicht zwischen linkem und rechtem Unterbaum unterscheiden, sondern nur die Reihenfolge notieren. Z. B.:

nur ein geordneter Baum mit nur einem Nachfolger:

aber zwei binäre Bäume mit nur einem Nachfolger:



Hinweis: Eine zu unserer Binarisierung verwandte Darstellung ist die Ordnerhierachie in einem Betriebssystem; dort listet man untereinaner die Geschwisterknoten und nach rechts abzweigend die Kinderknoten auf.

Zentrale Frage nun:

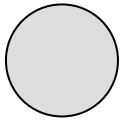
Wie viele binäre Bäume mit n Knoten gibt es?

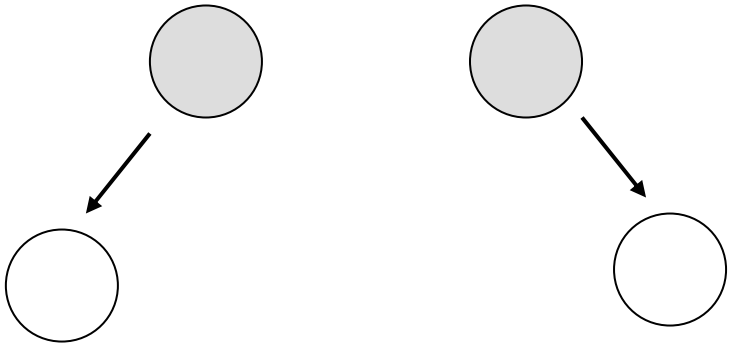
Das heißt: Man berechne C_n.

Wir werden für C_n eine geschlossene Formel angeben. Zunächst berechnen wir C_0, C_1, ..., C_4 durch Aufzählen aller zugehöriger Binärbäume.

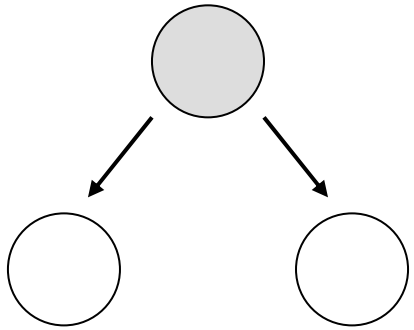
Wir listen alle binären Bäume mit höchstens 4 Knoten auf. Die Wurzel des binären Baums ist hier grau dargestellt. Die leeren Zeiger wurden weggelassen.

n = 0 <leerer Baum> $C_0 = 1$

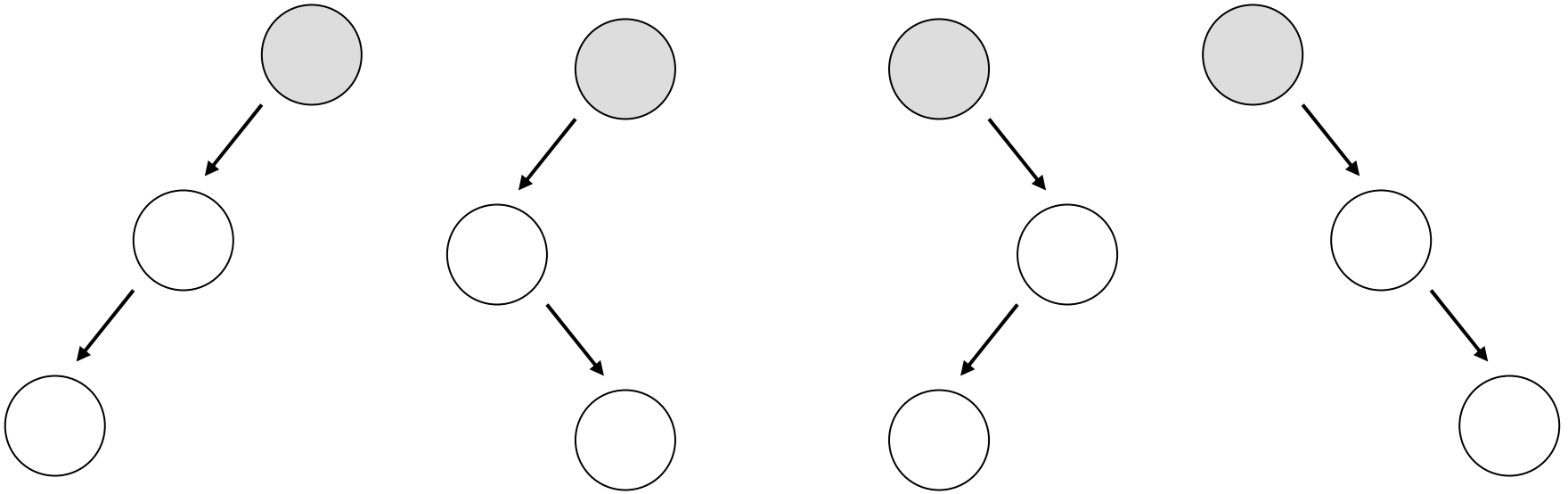
n = 1  $C_1 = 1$

n = 2  $C_2 = 2$

n = 3

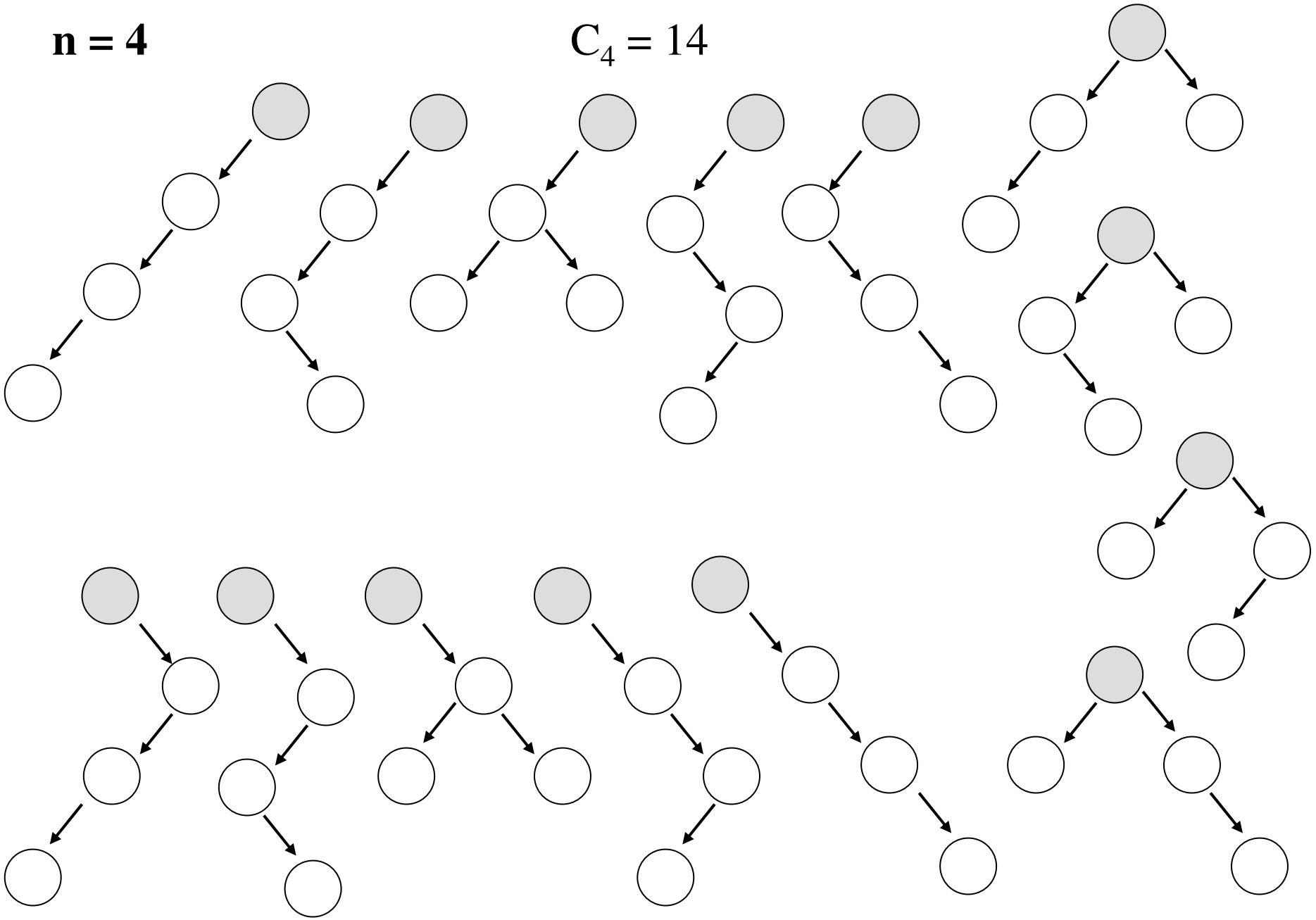


$C_3 = 5$



n = 4

$C_4 = 14$



Durch Ausprobieren erhält man die Werte:

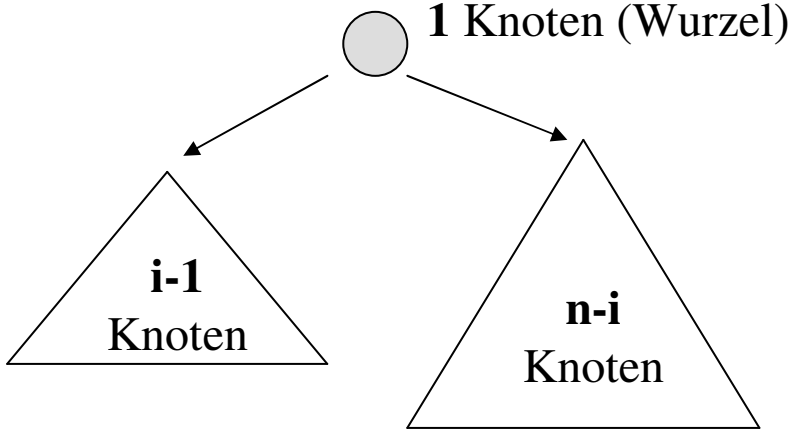
n	Anzahl
0	1
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430

Für C_n, die Anzahl der Binärbäume mit n Knoten, gilt die Rekursionsformel:

$$C_0 = 1, \text{ und für alle } n \geq 1:$$

$$C_n = \sum_{i=1}^{n} C_{i-1} \cdot C_{n-i}$$

wegen:



Erinnerung aus der Mathematik: Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!} = \frac{n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot(n-k+1)}{1\cdot 2\cdot 3\cdot\ldots\cdot k}$$

Bedeutung für die Anwendung:
Es gibt genau " n über k " Möglichkeiten, um k verschiedene Dinge auf n Bereiche zu verteilen bzw. um k Dinge aus n verschiedenen Dingen (ohne Zurücklegen und Wiederholungen) auszuwählen.
Beispiel: Lotto 6 aus 49: Es gibt genau " 49 über 6 " = (49·48·47·46·45·44)/(1·2·3·4·5·6) = 13 983 816 Möglichkeiten, aus 49 Zahlen 6 Zahlen auszuwählen, sofern die Reihenfolge des Auswählens keine Rolle spielt.

8.2.7 Satz "Catalansche Zahlen"

$$\mathbf{C_n = \frac{1}{n+1} \binom{2n}{n}}$$

Näherung: $C_n \approx \frac{4^n}{(n+1)\sqrt{\pi \cdot n}}$ für n>0

Wir beweisen nur den Satz. Die Näherung ergibt sich aus dem Satz unter Verwendung der Stirlingschen Formel für die Fakultät (8.8.8).

Wir zeigen zunächst: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen. Z.B. gibt es genau 5 korrekte Klammerungmöglichkeiten für n = 3:
((())), (()()), (())(), ()(()), ()()().

Behauptung: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen:

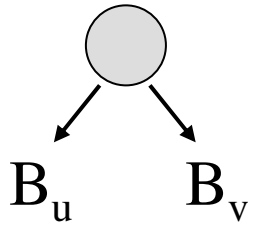
((( ))) , (( ) ( )) , (( )) ( ) , ( ) (( )) , ( ) ( ) ( ) .

Wir geben diesen Zusammenhang präzise an:

1. Jeder korrekt geklammerte Ausdruck fängt mit "(" an.
2. Es gibt zu dieser "(" genau eine zugehörige ")", nämlich die erste "Klammer zu", bei der die Anzahl der "Klammer auf" und "Klammer zu" gleich sind (von links nach rechts gezählt).
3. Also hat jeder Klammerausdruck die Form (u)v, wobei sowohl u als auch v korrekt geklammerte Ausdrücke sind. u und v sind eindeutig bestimmt. (u und v können leer sein.)

4. Ordne dem leeren Wort den leeren Binärbaum zu.

5. Ordne dann rekursiv dem Ausdruck **(**u**)**v folgenden Baum zu:



wobei B_u der zu u und B_v der zu v gehörige Baum ist.

Umgekehrt gewinnt man aus diesem Baum den Ausdruck **(**u**)**v.

Auf diese Weise lässt sich jedem korrekt geklammerten Ausdruck umkehrbar eindeutig ein binärer Baum zuordnen.

Folglich ist auch die Anzahl der korrekten Klammerungen aus n Klammerpaaren gleich C_n.

Damit ist die Behauptung bewiesen.

[Für u=v=ε wird **()** also der einknotige Baum zugeordnet:]

Wie viele korrekt geklammerte Ausdrücke gibt es?

Man muss n "Klammer auf" auf 2n Positionen verteilen.

Hiervon gibt es genau $\binom{2n}{n}$ Möglichkeiten.

Von dieser Anzahl muss man die abziehen, die zu keinen korrekten Klammerungen führen. Diese besitzen eine erste Position, bis zu der mehr "Klammer zu" als "Klammer auf" stehen; sie haben also die Form:

x) y

wobei x korrekt geklammert ist und y genau eine "Klammer auf" mehr besitzt als "Klammer zu".

Ersetze nun in y jede "(" durch ")" und umgekehrt. So möge die Klammerfolge y' entstehen. Für x) y' gilt dann:

x möge k Klammerpaare besitzen ($0 \leq k \leq n$).
Dann besitzt y n-k "Klammer auf" und n-k-1 "Klammer zu".
Dann besitzt y' n-k-1 "Klammer auf" und n-k "Klammer zu".
Also besitzt x)y' n-1 "Klammer auf" und n+1 "Klammer zu".

Weiterhin gilt: Geht man von zwei verschiedenen unkorrekten Klammerungen aus, so erhält man auch zwei verschiedene Ausdrücke der Form x)y'. Wäre nämlich $x)y' = x_1)y_1'$, dann muss $x=x_1$ sein, da x und x_1 beide korrekt geklammert sind und ")" an der ersten Position steht, an der die Anzahl der "Klammer zu" die Anzahl der "Klammer auf" übersteigt. Ebenso muss dann $y' = y_1'$ sein, da die Längen der beiden Ausdrücke gleich lang, nämlich 2n, sind, und folglich gilt auch $y = y_1$.

Wir haben also gezeigt: Jeder unkorrekten Klammerung von n Klammerpaaren lässt sich eindeutig eine Folge von n-1 "Klammer auf" und n+1 "Klammer zu" zuordnen.

Die Umkehrung gilt aber auch:

Wenn eine Folge aus n-1 "Klammer auf" und n+1 "Klammer zu" gegeben ist, so muss es genau eine erste Stelle geben, an der die Zahl der "Klammer zu" die Zahl der "Klammer auf" erstmals übersteigt. Die Folge hat also die Form x)y' , wobei in x überall die Anzahl der "Klammer auf" größer oder gleich der Anzahl der "Klammer zu" bis zu dieser Stelle ist. x ist also ein korrekt geklammerter Ausdruck. In y' gibt es dann eine "Klammer auf" weniger, als es "Klammer zu" gibt. Wandle nun y' in ein y um, indem jede "(" durch ")" ersetzt wird und umgekehrt. So erhält man eine Folge x)y, die gleich viele "Klammer auf" und "Klammer zu" besitzt und die nicht korrekt geklammert ist.

Diese Zuordnung ist offenbar ebenfalls eindeutig.

Daher gilt:

Jeder unkorrekten Klammerung von n Klammerpaaren lässt sich umkehrbar eindeutig eine Folge von n-1 "Klammer auf" und n+1 "Klammer zu" zuordnen. Hieraus folgt:

Die Anzahl der unkorrekten Klammerungen mit n Klammerpaaren ist gleich der Anzahl der Folgen von n-1 "Klammer auf" und n+1 "Klammer zu".

Deren Anzahl ist aber $\binom{2n}{n-1}$

Wir haben also gezeigt:

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n} .$$

Damit ist Satz 8.2.7 bewiesen. ■

Folgerung 1:

$$C_{n+1} = C_n \cdot \frac{4n+2}{n+2} = 4 \cdot C_n \cdot \left(1 - \frac{6}{n+2}\right)$$

Dieser Formel kann man zum einen das exponentielle Wachstum entnehmen, das in der Näherungsformel ausgedrückt wird. Zum anderen lassen sich hiermit die Catalanschen Zahlen, ausgehend von $C_0 = 1$, leicht iterativ berechnen.

Hinweis: Aus dem Satz lässt sich sofort schließen:

Der Binomialkoeffizient $\binom{2n}{n}$ ist stets durch n+1 teilbar.

(Unter welchen Bedingungen auch durch n+2?

Man erhält oft solche "nebensächlichen" Resultate.)

Folgerung 2 aus diesem Satz:

Wir werden anstreben, Mengen in geordneten oder in binären Bäumen zu speichern. Will man n Elemente auf diese Weise ablegen, so gibt es C_n verschiedene Bäume, die hierfür in Frage kommen. Deren Anzahl wächst größenordnungsmäßig wie 4^n, so dass man "geeignete" Bäume in der Praxis *nicht durch Ausprobieren* aller Möglichkeiten finden kann!

Vielmehr folgern wir aus Satz 8.2.7:
Wir müssen "intelligente" Verfahren entwickeln, um spezielle Bäume, die für gewisse Anwendungsprobleme nützlich sind, aufzuspüren. Wie solche speziellen Bäume aussehen können und welche "intelligenten" Verfahren es für ihre Bearbeitung gibt, wird im weiteren Verlauf dieses Kapitels 8 betrachtet.

Erinnerung an Abschnitt 3.7
(Einige Details sind im Anhang 8.8.8 aufgeführt.)

Ein binärer Baum kann preorder, inorder oder postorder in linearer Zeit durchlaufen werden, siehe nächste Folie (L ist der Verweis zum linken Unterbaum, R der zum rechten).

Man kann eine Folge sortieren, indem man ihre Elemente nacheinander in einen Binärbaum einfügt und diesen am Ende inorder ausgibt (siehe 3.7.7).

Hierzu muss jeder Knoten einen "Inhalt" erhalten und die Elemente müssen in die Knoten "richtig" eingeordnet werden. Solch einen Baum nannten wir einen "Suchbaum".

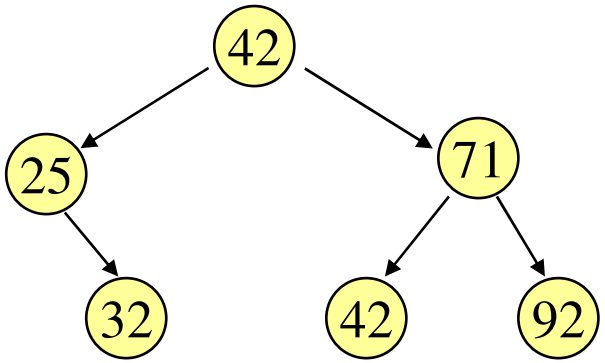
```
procedure Preorder (K: Ref_BinBaum) is
begin  if K /= null then  < bearbeite den Knoten K >;
                          Preorder (K.L);
                          Preorder (K.R);
       end if;
end Preorder;
```

```
procedure Inorder (K: Ref_BinBaum) is
begin  if K /= null then  Inorder (K.L);
                          < bearbeite den Knoten K >;
                          Inorder (K.R);
       end if;
end Inorder;
```

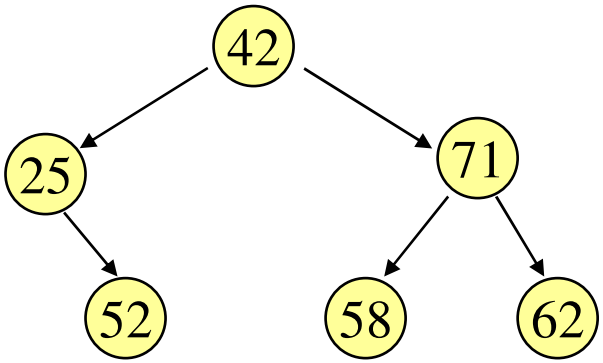
```
procedure Postorder (K: Ref_BinBaum) is
begin  if K /= null then  Postorder (K.L);
                          Postorder (K.R);
                          < bearbeite den Knoten K >;
       end if;
end Postorder;
```

8.2.8 Definition "Suchbaum"

Ein binärer Baum, dessen Inhalts-Datentyp geordnet ist (z.B. ganze Zahlen), heißt binärer Suchbaum, wenn für jeden Knoten u gilt: Alle Inhalte von Knoten im linken Unterbaum von u sind echt kleiner als der Inhalt von u und alle Inhalte von Knoten im rechten Unterbaum von u sind größer oder gleich dem Inhalt von u.



Dies ist ein Suchbaum



Dies ist kein Suchbaum

8.2.9 Suchbäume zu einer festen Folge

Es sei $a_1, a_2, ..., a_n$ eine sortierte Folge von n ganzen Zahlen und es sei B ein Suchbaum mit dem Inhalts-Datentyp Integer. Dann kann man die Zahlen a_i auf genau eine Art so in die Knoten von B legen, dass der Inorder-Durchlauf von B die sortierte Folge $a_1, a_2, ..., a_n$ ergibt.

Dies ist klar: Man durchlaufe B inorder und ordne dem i-ten Knoten bei dieser Besuchsreihenfolge die Zahl a_i zu.

Da es exponentiell viele (genauer: C_n) binäre Bäume mit n Knoten gibt, muss man den für eine gegebene Fragestellung "besten Baum" mit guten Algorithmen ermitteln.

Wir untersuchen zunächst Binärbäume (in Form von Suchbäumen) und gehen ab Kapitel 8.3 zu spezielleren Bäumen über.

8.2.10 Binäre Bäume und die Grundaufgaben

Gegeben sei eine geordnete Menge. Hierfür werden wir die ganzen Zahlen verwenden.
Eine Teilmenge oder eine Folge solcher Elemente (d.h., es können Elemente auch mehrfach vorkommen!) soll in einem Binärbaum verwaltet werden. Wir berechnen zu konkreten Verfahren für das Suchen (FIND), das Einfügen (INSERT) und das Löschen (DELETE) deren Aufwand im schlechtesten Fall und im Durchschnitt (im worst case und im average case). Die Datenstruktur für Binärbäume ist:

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        Inhalt: Integer;
        L, R: Ref_BinBaum;
end record;
```

Für alle Suchbäume (auch für die später vorzustellenden AVL- und B-Bäume) und alle Elemente s gilt:

Suchen (FIND): Durchlaufe einen Pfad von der Wurzel abwärts zu einem Blatt, wobei man entweder s findet oder feststellt, dass s im Baum nicht vorkommt.

Einfügen (INSERT): Gehe so vor, als ob s gefunden werden soll; hierbei gelangt man schließlich an einen leeren Verweis; hänge genau hier einen neuen Knoten mit dem Element s an.

Löschen (DELETE): Suche den Knoten u mit Inhalt s. Falls dieser Knoten keinen oder nur einen Nachfolger besitzt, kann man ihn leicht löschen. Falls er mehr Nachfolger hat, so suche den Knoten v des Inorder-Vorgängers oder -Nachfolgers s' von s, überschreibe s in u mit s' und entferne v geeignet.

8.2.10.a Suchen in einem Binärbaum

Der Binärbaum ist durch den Zeiger "Anker" auf seine Wurzel gegeben. Wir formulieren die Prozedur *Suche*, die zu dem Zeiger Anker und dem zu suchenden Element s einen Verweis q auf den Knoten zurückgibt, dessen Inhalt s ist. Ist s nicht im Binärbaum enthalten, wird der Verweis null zurückgegeben.

```
procedure Suche (Anker: in Ref_BinBaum; s: in Integer;
                                          q: out Ref_BinBaum) is
begin  q := Anker;
    while q /= null loop
        if q.Inhalt = s then return;
        elsif q.Inhalt > s then q := q.L; else q := q.R; end if;
    end loop;
end Suche;
```

An die Definition des Binärbaums besser angepasst ist die *rekursive Darstellung:*

```
procedure SucheRek (p: in Ref_BinBaum; s: in Integer;
                                        q: out Ref_BinBaum) is
begin
    if p = null  then  q := null;
    else
       if p.Inhalt > s then SucheRek(p.L, s, q);
       elsif p.Inhalt < s then SucheRek(p.R, s, q);
       else q := p;
       end if;
    end if;
end SucheRek;
```

Verwendung dieser Prozedur:

```
SucheRek(Anker, Schlüssel, Ergebniszeiger);
if Ergebniszeiger = null then .... else ... end if;
```

8.2.10.b Einfügen in einen Binärbaum

Prinzipiell werden bei binären Suchbäumen die neuen Elemente als neues Blatt in den Baum eingetragen.

```
procedure Einfügen (Anker: in out Ref_BinBaum; s: Integer) is
    p, q: Ref_BinBaum := Anker;
begin
    if  p = null  then  Anker := new BinBaum'(s, null, null);
    else
        while p /= null loop  q := p;
            if p.Inhalt > s then p := p.L; else p := p.R; end if;
        end loop;
        if q.Inhalt > s then q.L := new BinBaum'(s, null, null);
        else q.R := new BinBaum'(s, null, null); end if;
    end if;
end Einfügen;
```

Aufgabe: Schreiben Sie für diese Operation eine rekursive Prozedur.

8.2.10.c Löschen in einem Binärbaum

Der Schlüssel s soll gelöscht werden. Die Suche ergibt, dass s im Knoten u steht. Hat u keinen Nachfolger, so wird u einfach gelöscht und der Verweis vom Vorgängerknoten von u wird auf null gesetzt.

Hat u genau einen Nachfolger, so wird u gelöscht und der Verweis des Vorgängerknotens auf u wird auf den einzigen Nachfolger von u gesetzt.

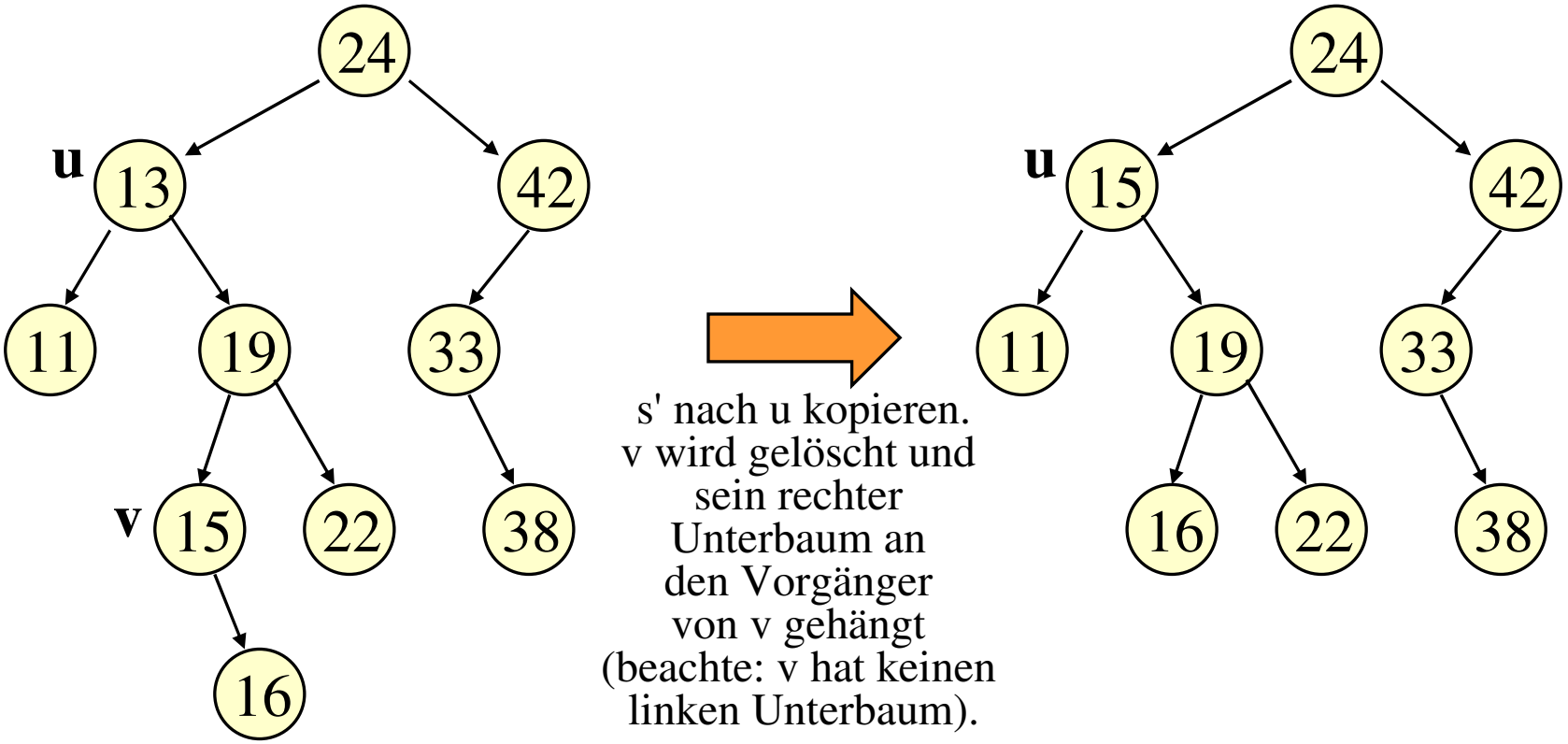
Hat u zwei Nachfolger, dann kann u nicht einfach gelöscht werden. Vielmehr ersetzt man den Inhalt von u durch einen Schlüssel s', der in einem Knoten v mit höchstens einem Nachfolger steht und löscht dann v wie oben angegeben. Damit der Baum ein Suchbaum bleibt, muss s' in der sortierten Reihenfolge aller Inhalte des Baums unmittelbar vor oder unmittelbar nach s stehen.

s' möge bzgl. der Sortierung der unmittelbar folgende Schlüssel von s sein. Da die sortierte Reihenfolge durch einen inorder-Durchlauf erreicht wird, nennt man s' und seinen Knoten v den "Inorder-Nachfolger" von s bzw. von u. s steht in dem Knoten v, der in der inorder-Reihenfolge auf u folgt: v ist der linkeste Knoten im rechten Unterbaum von u. Man geht also zum rechten Nachfolger von u und folgt dann immer dem linken Verweis, bis dieser null ist.

In den Knoten u schreibt man nun s' und löscht den Knoten v, wobei dessen eventueller rechter Unterbaum an den Vorgänger von v gehängt wird.

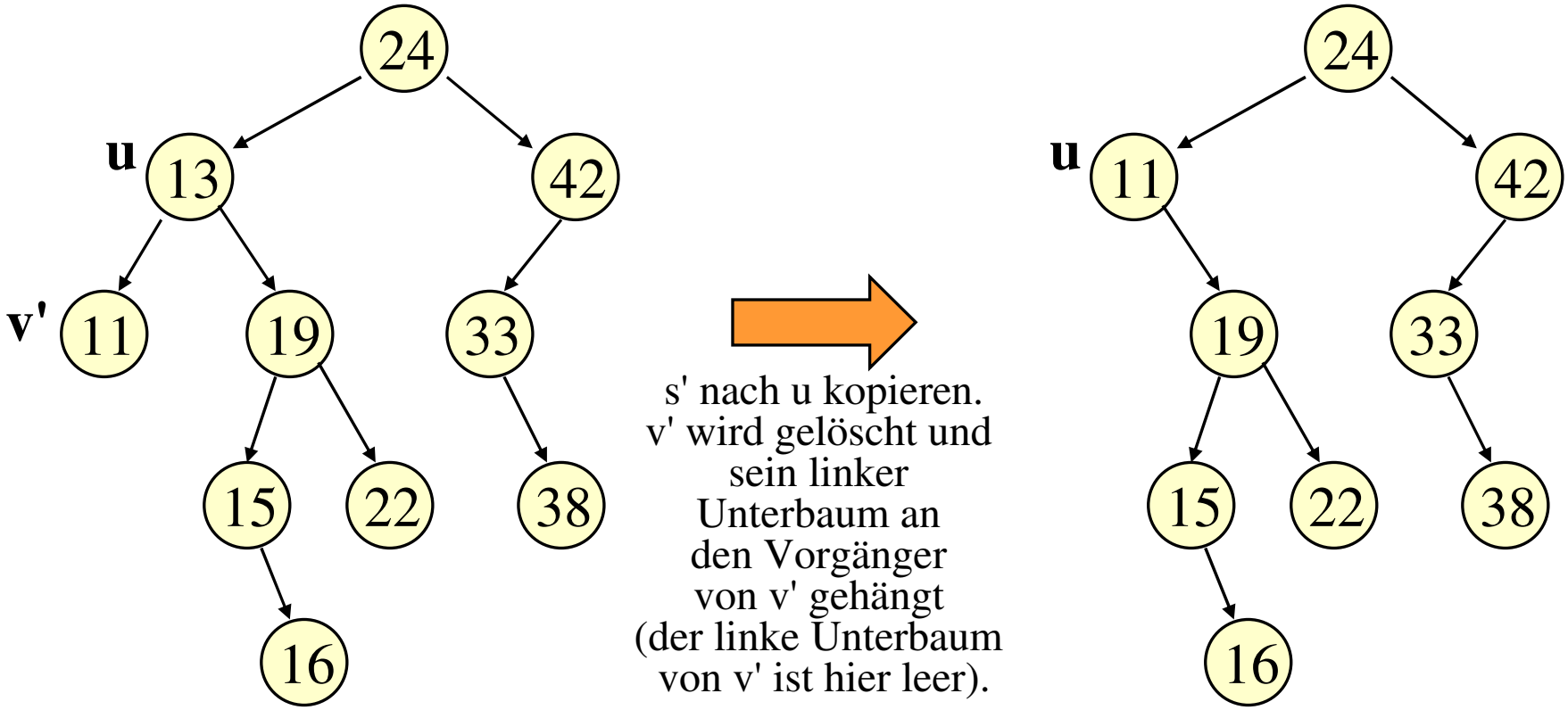
Man kann auch den Schlüssel s' unmittelbar vor s wählen, den sog. "Inorder-Vorgänger": Er steht im Knoten v', der der rechteste Knoten im linken Unterbaum von u ist.
In der Praxis entscheidet man sich jedes Mal zufällig, ob man den Inorder-Vorgänger oder den Inorder-Nachfolger für das Löschen heranzieht.

Skizze: Lösche s=13. s steht im Knoten **u**. **u** hat zwei Kinder.



Der Inorder-Nachfolger-Knoten **v** von **u** hat hier den Inhalt s'=15.

Man hätte auch den Inorder-Vorgänger-Knoten v' nehmen können:



Der Inorder-Vorgänger-Knoten **v'** von **u** hat hier den Inhalt s'=11.

Löschen in einem Binärbaum (über den Inorder-Nachfolger)

```
procedure Löschen (Anker: in Ref_BinBaum; s: in Integer) is
u, v: Ref_BinBaum := null;
begin  Suche (Anker, s, u);   -- siehe 8.2.10a; falls u=null ist, nichts tun
    if  u /= null  then
        if (u.L = null) or (u.R = null) then
            < lösche u, hänge Unterbaum um, sofern einer existiert > ;
        else                              -- Suche den Inorder-Nachfolgerknoten v
            v := u.R;
            while  v.L /= null  loop  v := v.L; end loop;
            u.Inhalt := v.Inhalt;              -- s' wird nach u kopiert
            < lösche v, hänge Unterbaum um, sofern einer existiert > ;
        end if;                   -- Hier ist noch ein Problem: Für < lösche u >
    end if;                       -- und < lösche v > muss man den jeweiligen
end Löschen;                      -- Vorgänger von u bzw. v kennen. Also muss
                                  -- die Prozedur Löschen modifiziert werden.
```

Hinweis zur Realisierung: Man lässt einen Zeiger "vorg" mitlaufen, der auf den zuvor betrachteten (Vorgänger-) Knoten zeigt:

```
procedure Löschen (Anker: in Ref_BinBaum; s: in Integer) is
u, v, vorg: Ref_BinBaum := null; links: Boolean;
begin  "Suche(Anker, s, u, vorg, links);"
    -- dies ist neu zu programmieren: vorg zeigt auf den Elternknoten von u
    -- (sofern vorhanden) und links ist true, falls u linkes Kind von vorg ist
    if  u /= null  then
        if (u.L = null) or (u.R = null) then  < lösche u, ... > ;
        else                              -- Suche den Inorder-Nachfolgerknoten v
            v := u.R; vorg := u;
            while  v.L /= null  loop  vorg := v; v := v.L; end loop;
            u.Inhalt := v.Inhalt;          -- s' wird nach u kopiert
            < lösche v > ;
        end if;
    end if;
end Löschen;
```

Bedeutet < lösche v > nun einfach:

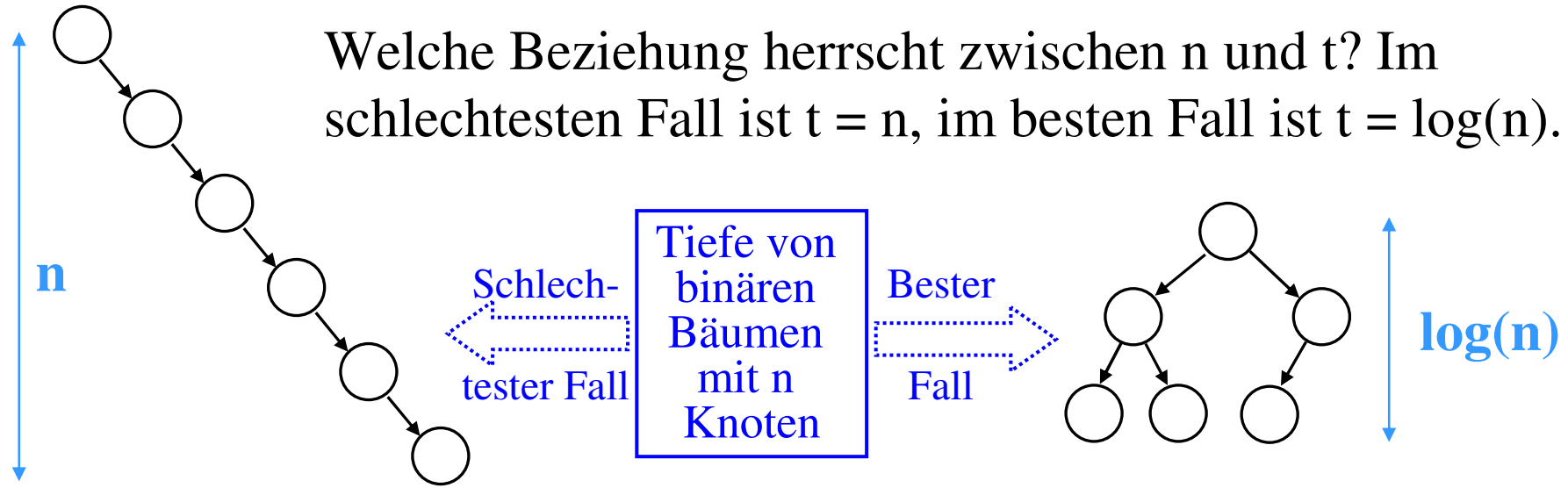
vorg.L := v.R; ? (Vorsicht: Fehler!)

Wie müssen < lösche u > und Suche programmiert werden?? Selbst lösen!

8.2.10.d: Welchen Aufwand erfordern die Operationen Suchen, Einfügen und Löschen, wenn der Suchbaum n Knoten besitzt?

Man misst diesen Aufwand meist in der Anzahl der Vergleiche, die erforderlich sind, um die Operation durchzuführen.

Suchen, Einfügen und Löschen: Im schlechtesten Fall benötigt man jeweils t Vergleiche, wobei t die Tiefe des Baumes mit n Knoten ist; zur Definition "Tiefe" siehe 8.2.4 (1).



Welche Beziehung herrscht zwischen n und t? Im schlechtesten Fall ist t = n, im besten Fall ist t = log(n).

Ein Baum kann also zu einer Liste "entarten" und man braucht dann entsprechend viele Vergleiche.

Besonders günstig ist dagegen ein Baum, bei dem jeder Knoten höchstens das Level log(n+1) besitzt; zu "Level" siehe 8.2.4 (2).

Was genau ist hier "log" (der Logarithmus zur Basis 2)? Wir benötigen ihn als eine Funktion zwischen natürlichen Zahlen und modifizieren daher die üblicherweise über den positiven reellen Zahlen definierte Logarithmusfunktion wie folgt:

Definition 8.2.11: Diskreter Zweierlogarithmus. Wenn nicht anders vermerkt sei in Zukunft log: $\mathbb{N} \to \mathbb{N}_0$ definiert durch
$\log(1) = 0$, und für $n \geq 2$:
$\log(n) = k$ für das eindeutig bestimmte k mit $2^{k-1} < n \leq 2^k$.

Es gilt dann also log(2)=1, log(3)=log(4)=2, log(5)=3 usw. Vom üblichen Logarithmus weicht dieses log höchstens um 1.0 ab.

Folgerung 8.2.12:
log sei wie in 8.2.11 definiert, dann gilt für die Tiefe t jedes binären Baums mit n Knoten $\log(n+1) \leq t \leq n$ für alle $n \geq 0$.

Beweis: $t \leq n$ ist klar, da jeder doppelpunktfreie Weg in einem Baum mit n Knoten höchstens n Knoten besitzen kann.

Ein binärer Baum der Tiefe k kann höchstens 2^k-1 Knoten haben, wie man durch Induktion leicht sieht:
Für k=1 ist dies richtig, und wenn Bäume der Tiefe k höchstens 2^k-1 Knoten enthalten, dann kann ein Baum der Tiefe k+1 höchstens "Wurzel plus zwei Unterbäume der Tiefe k", also $1+2^k-1+2^k-1=2^{k+1}-1$ Knoten besitzen.
Folglich ist $n \leq 2^t-1$, also $\log(n+1) \leq \log(2^t) = t$.
(Für den Fall n=0 trifft die Aussage ebenfalls zu.) ■

Hinweis: In der Ungleichung $\log(n+1) \leq t \leq n$ für alle $n \geq 0$ werden beide Grenzen angenommen, d.h., es gibt Bäume mit n Knoten, deren Tiefe n ist, und es gibt Bäume mit n Knoten, deren Tiefe $\log(n+1)$ ist.

Übungsaufgaben:

a) Berechnen Sie, wie viele verschiedene binäre Bäume mit n Knoten es gibt, die genau die Tiefe n besitzen.

b) Versuchen Sie zu berechnen, wie viele verschiedene binäre Bäume mit n Knoten es gibt, die genau die Tiefe $\log(n+1)$ besitzen.

c) Berechnen Sie die "mittlere Suchzeit im besten Fall" für den Fall $n = 2^k-1$, d.h., summieren Sie für einen Baum mit diesen n Knoten und minimaler Tiefe alle Level seiner Knoten auf und dividieren diesen Wert durch n. Machen Sie das Gleiche für den schlechtesten Fall.

Zur Komplexität der Grundoperationen Suchen, Einfügen und Löschen in binären Suchbäumen:

Jede Operation benötigt im Mittel O(log(n)) Schritte.

Beim Einfügen muss man sich hierbei auf die Folge von zu speichernden Zeichen (und nicht auf die Suchbäume) beziehen.

Diese Aussagen werden im Folgenden genauer beleuchtet.

Für die Praxis ist die "mittlere Suchzeit" eines binären Baums wichtig. Hier gibt es zwei verschiedene Ansätze. Ansatz 1:

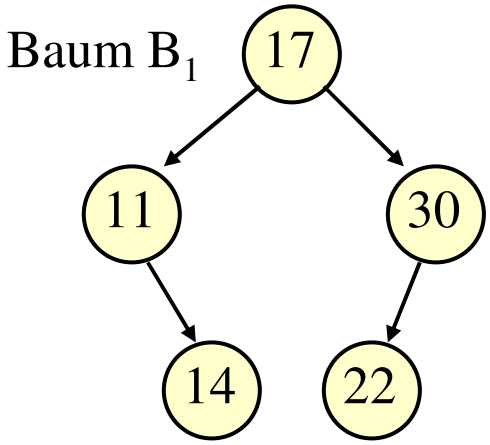
Definition 8.2.13.a: Man betrachte alle C_n binären Bäume mit n Knoten ($n > 0$). Für jeden Baum B mit n Knoten $v_1, ..., v_n$ berechne man das mittlere Level oder die mittlere Suchzeit *ml*:

$$ml(B) = \frac{1}{n} \sum_{i=1}^{n} level(v_i)$$

und ermittle hiermit das mittlere Level ML_n aller binären Bäume mit n Knoten:

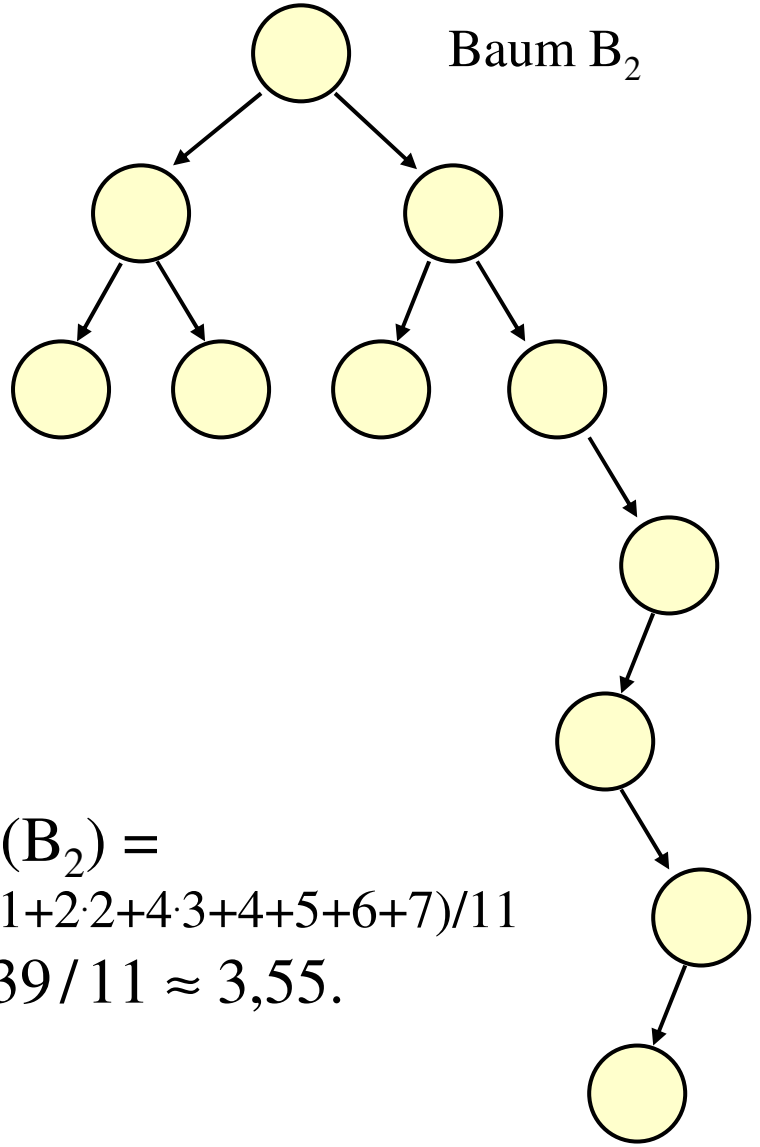
$$ML_n = \frac{1}{C_n} \sum_{\substack{B \text{ ist binärer} \\ \text{Baum mit n} \\ \text{Knoten}}} ml(B)$$

Zwei Binärbäume als Beispiele für ml:



$ml(B_1)$
$= (1+2+2+3+3)/5$
$= 11/5 = 2{,}2$

Man sieht, dass *ml* von den Inhalten in den Knoten unabhängig ist.



$ml(B_2) =$
$= (1+2\cdot2+4\cdot3+4+5+6+7)/11$
$= 39/11 \approx 3{,}55.$

Machen Sie sich die Definition von ML_n genau klar: Unter der Annahme, dass alle C_n binären Bäume mit n Knoten gleichwahrscheinlich sind, ist ML_n deren mittlere Suchzeit.

Beispiel (selbst nachrechnen, vgl. die Auflistung der binären Bäume in 8.2.6 für $n = 1, ..., 4$):

$ML_1 = 1/1 = 1$

$ML_2 = (1/2)\cdot((1+2)/2 + (1+2)/2) = 1{,}5$

$ML_3 = (1/5)\cdot((1+2+2)/3 + 4\cdot(1+2+3)/3) = 29/15 = 1{,}9333...$

$ML_4 = (1/14)\cdot(80+32+18)/4 = 130/56 = 2{,}3214...$

$ML_5 = (1/42)\cdot(562/5) = 2{,}67619...$

Man kann beweisen, dass ML_n mit $O(\sqrt{n})$ wächst.
(Der Beweis ist relativ aufwändig, siehe Literatur.)

Wir kommen zum Ansatz 2:
Definition 8.2.13.b: Gesucht wird die mittlere Suchzeit aller Bäume, die zu allen Folgen von n Elementen gehören. Das heißt: Man gehe von den n! Folgen mit n Elementen aus, füge jede Folge entsprechend ihrer Reihenfolge in einen binären Suchbaum ein und frage dann nach der mittleren Suchdauer dieser n! Bäume.

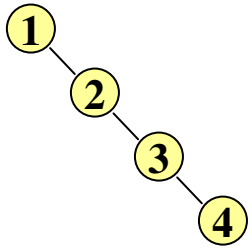
Es sei also $a = a_1 a_2 \ldots a_n$ eine Folge aus n Elementen. Man baue hieraus den binären Suchbaum B_a auf, indem man den Algorithmus 8.2.10.b auf $a_1, a_2, \ldots, a_n$ (und danach auf alle Permutationen) anwendet. Dann sei die mittlere Suchzeit *MS*:

$$MS_n = \frac{1}{n!} \sum_{\substack{\pi(a) \text{ ist eine} \\ \text{Permutation} \\ \text{der Folge a}}} ml(B_{\pi(a)})$$

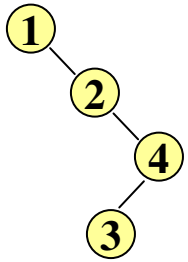
Die Folge a ist irrelevant. Wichtig ist nur, dass sie aus n Elementen besteht.

Beispiel für n = 4: Es sei a = 1 2 3 4. Neben jede Permutation $\pi(a)$ wird der zugehörige Baum $B_{\pi(a)}$ gezeichnet:

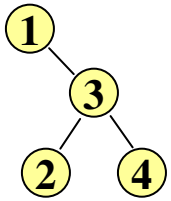
1 2 3 4



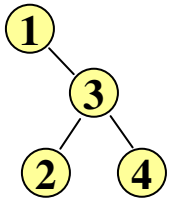
1 2 4 3



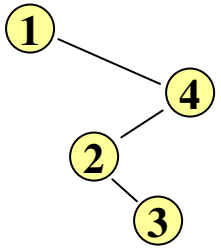
1 3 2 4



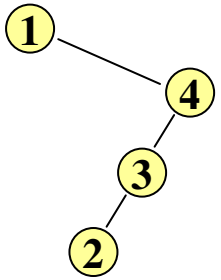
1 3 4 2



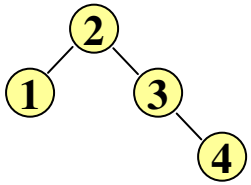
1 4 2 3



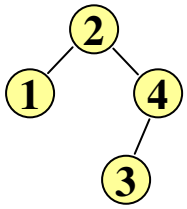
1 4 3 2



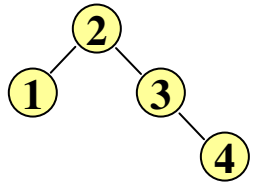
2 1 3 4



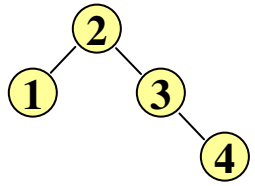
2 1 4 3



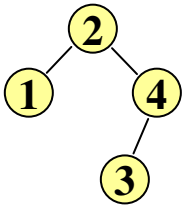
2 3 1 4



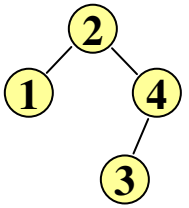
2 3 4 1



2 4 1 3



2 4 3 1



Die Bäume zu den restlichen 12 Permutationen 3 1 2 4, 3 1 4 2, 3 2 1 4, 3 2 4 1, 3 4 1 2, 3 4 2 1, 4 1 2 3, 4 1 3 2, 4 2 1 3, 4 2 3 1, 4 3 1 2 und 4 3 2 1 mögen Sie selbst zusammenstellen.

Es lässt sich nun MS_4 hieraus leicht berechnen:

$$MS_4 = (1/24)\ (10/4 + 10/4 + 9/4 + 9/4 + 10/4 + 10/4 + 8/4 + 8/4 + 8/4 + 8/4 + 8/4 + 8/4 + \ldots\ )$$
$$= 53/24 = 2{,}20833\ldots$$

MS_4 ist kleiner als ML_4. Dies ist kein Zufall, sondern es gilt stets $MS_n \leq ML_n$.

Übungsaufgabe: Diese Aussage kann man anschaulich leicht einsehen; überlegen Sie sich, wie. (Notfalls finden Sie Hinweise in 8.7.5 und in 8.7.7.)

Problem: Mit welcher mittleren Suchzeit MS_n muss man rechnen, wenn man einen Binärbaum aus einer zufälligen Folge mit n Elementen erzeugt?

Bevor Sie weiterlesen: Versuchen Sie eine Rekursionsformel aufzustellen, um diese Suchzeit zu beschreiben. Im Folgenden wird eine Formel und die Herleitung der Lösung angegeben.

Lösungsansatz: Wir betrachten nur Binärbäume. Es sei F(n) die zu berechnende mittlere Suchzeit *für alle n Knoten zusammen*. Die mittlere Suchdauer ist dann $MS_n = F(n)/n$. Es gilt F(0)=0 und F(1)=1.

Betrachte einen Binärbaum mit n > 1 Knoten. Dann gibt es eine Wurzel, die einen linken Unterbaum mit i-1 Knoten und einen rechten Unterbaum mit n-i Knoten besitzt für ein i mit $1 \le i \le n$. Ist nun jedes i gleichwahrscheinlich (und dies darf man beim zufälligen Aufbauen annehmen), dann erhält man

$$F(n) = (1/n) \cdot \big((F(0) + F(n-1) + \text{Erhöhung der Pfadlängen}) + (F(1) + F(n-2) + \text{Erhöhung der Pfadlängen}) + \ldots (F(n-1) + F(0) + \text{Erhöhung der Pfadlängen}) \big)$$

Die "Erhöhung der Pfadlängen" berücksichtigt die Verlängerung aller Pfade durch die Wurzel. Diese Erhöhung ist aber für *jeden* Knoten 1, d.h. "Erhöhung der Pfadlängen" = n.

Somit erhalten wir die Formeln: $F(0) = 0$ und $F(1) = 1$ und

$$F(n) = (1/n) \cdot \big((F(0) + F(n-1) + n) + (F(1) + F(n-2) + n) + \ldots \\ (F(n-1) + F(0) + n) \big)$$
$$= (1/n) \cdot 2 \cdot (F(0) + F(1) + F(2) + \ldots + F(n-1)) + n$$

Folglich gilt

$$F(n-1) = (1/(n-1)) \cdot 2 \cdot (F(0) + F(1) + F(2) + \ldots + F(n-2)) + n-1$$
$$= (n/(n-1)) \cdot (1/n) \cdot 2 \cdot \big(F(0) + F(1) + F(2) + \ldots + F(n-2)\big) + n-1$$

Den Wert für

$$(1/n) \cdot 2 \cdot \big(F(0) + F(1) + F(2) + \ldots + F(n-2)\big) = (n-1)/n \cdot (F(n-1) - n+1)$$

setzen wir nun oben ein und erhalten die Rekursionsformel:

$$F(n) = (1/n) \cdot 2 \cdot F(n-1) + (n-1)/n \cdot (F(n-1) - n+1) + n$$
$$= (n+1)/n \cdot (F(n-1) - (n-1)^2/n + n = (n+1)/n \cdot F(n-1) + (2n-1)/n$$

Für Interessierte: Wie findet man Lösungen für solche Gleichungen? Siehe hierzu 8.9.1.

Hiermit kann man die mittlere Suchzeit F(n)/n, die man für zufällig aufgebaute Binärbäume erwarten muss, bereits berechnen:

$F(2)/2 = (3/2 \cdot F(1) + 3/2)/2 = 3/2 = 1{,}5$
$F(3)/3 = (4/3 \cdot F(2) + 5/3)/3 = 17/9 = 1{,}8888...$
$F(4)/4 = (5/4 \cdot F(3) + 7/4)/4 = 106/48 = 2{,}20833...$

Um eine geschlossene Formel für die exakte Lösung zu erhalten, probiert man einige Umformungen aus, zum Beispiel vereinfacht man die Formel durch Division durch (n+1); anschließend setzt man F(n-1) und danach F(n-2), F(n-3) usw. ein, bis man eine geschlossene Darstellung erhält.
$F(n) = (n+1)/n \cdot F(n-1) + (2n-1)/n$ wird also umgewandelt in

$$\frac{F(n)}{n+1} = \frac{2n-1}{n \cdot (n+1)} + \frac{F(n-1)}{n}$$

Wir ersetzen F(n-1)/n nun mit dieser Formel:

$$\frac{F(n)}{n+1} = \frac{2n-1}{n\cdot(n+1)} + \frac{F(n-1)}{n}$$

$$= \frac{2n-1}{n\cdot(n+1)} + \frac{2n-3}{(n-1)\cdot n} + \frac{F(n-2)}{n-1}$$

$$= \frac{2n-1}{n\cdot(n+1)} + \frac{2n-3}{(n-1)\cdot n} + \frac{2n-5}{(n-2)\cdot(n-1)} + \frac{F(n-3)}{n-2}$$

usw. So erhält man folgende Summenformel (F(0)=0):

$$\frac{F(n)}{n+1} = \sum_{i=0}^{n-1} \frac{2(n-i)-1}{(n-i)\cdot(n-i+1)} + \frac{F(0)}{1}$$

$$= \sum_{i=0}^{n-1} \frac{2}{(n-i+1)} - \sum_{i=0}^{n-1} \frac{1}{(n-i)\cdot(n-i+1)}$$

$$\frac{F(n)}{n+1} = \sum_{i=0}^{n-1} \frac{2}{(n-i+1)} - \sum_{i=0}^{n-1} \frac{1}{(n-i)\cdot(n-i+1)}$$

$$= 2 \cdot \sum_{i=2}^{n+1} \frac{1}{i} - \sum_{i=1}^{n} \frac{1}{i\cdot(i+1)}$$

$$= 2 \cdot \left(H(n+1) - 1\right) - \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right)$$

$$= 2 \cdot H(n+1) - 2 - 1 + \frac{1}{n+1} \qquad \text{mit } H(n) = \sum_{i=1}^{n} \frac{1}{i}$$

$$F(n) = 2\cdot(n+1)\cdot H(n+1) - 3\cdot(n+1) + 1 = 2\cdot(n+1)\cdot H(n) - 3\cdot n$$

Definition 8.2.14: H(n) heißt "harmonische Funktion".
Hierbei wird $H(0) = 0$ gesetzt. (Illustration: siehe 1.5 und 8.9.2.)

Lösung: $F(n) = 2 \cdot (n+1) \cdot H(n) - 3 \cdot n$

Aus der Analysis ist bekannt: $H(n) - \ln(n) \longrightarrow \gamma$ für $n \to \infty$.

ln ist der natürliche Logarithmus (zur Basis e = 2,7182818284...),

$\gamma = 0{,}5772156649...$ ist die Eulersche Konstante. Einsetzen ergibt:

$$F(n) \approx 2 \cdot (n+1) \cdot (\ln(n) + \gamma) - 3 \cdot n$$

(Hier ist log der reellwertige Zweierlogarithmus.)

$$\approx 2 \cdot (n+1) \cdot \log(n)/\log(e) + 2 \cdot (n+1) \cdot \gamma - 3 \cdot n$$

$$\approx 1{,}3863 \cdot n \cdot \log(n) - 1{,}8456 \cdot n + O(\log(n)).$$ *Bilde nun F(n)/n.*

Satz 8.2.15: Die mittlere Suchzeit MS_n

Die mittlere Suchzeit in einem zufällig aufgebauten Binärbaum mit n Knoten beträgt $1{,}3863 \cdot \log(n) - 1{,}8456$. Sie ist um rund 39% schlechter als die mittlere Suchzeit im besten Fall.

(Zu letzterer siehe Hinweis in 8.2.12; sie liegt bei $\log(n+1) - 1$.)

8.2.16: Zeitkomplexität des Sortieralgorithmus Baumsortieren ($\Rightarrow$ 3.7.7): Sortiere n Elemente, indem sie nacheinander in einen Suchbaum eingefügt und anschließend mit einem Inorder-Durchlauf ausgelesen werden.

Nach Satz 8.2.15 benötigt dieses Verfahren im Mittel (average case) $1{,}3863 \cdot n \cdot \log(n) + O(n)$ Schritte. Im schlechtesten Fall kann allerdings ein zu einer Liste entarteter Suchbaum entstehen, so dass das Verfahren im worst case $O(n^2)$ Schritte braucht.

Hinweis: Da Quicksort (siehe 7.3.3) ein Verfahren ist, welches zufällig einen binären Suchbaum erzeugt (das erste Pivot-Element wird die Wurzel dieses Baumes, danach rekursiv links und rechts weitermachen), ist $1{,}3863 \cdot n \cdot \log(n) - 1{,}8456 \cdot n + O(\log(n))$ zugleich die mittlere Zeitkomplexität für Quicksort (wir kommen hierauf in Kapitel 10 zurück).

Haben wir nun wirklich bewiesen, dass beim Einfügen und Löschen stets Binärbäume entstehen, deren mittlere Tiefe $O(\log(n))$ ist? Nein, dies ist nicht der Fall, weil mit unserem Verfahren beim Löschen *keine* gleichwahrscheinliche Auswahl des zu löschenden Knotens erfolgt. Denn es wurde bei uns stets der "Inorder-Nachfolger" ausgewählt, also ein Knoten, der immer im rechten Unterbaum liegt!

Daher ist nach einer längeren Folge von Einfüge- und Löschoperationen damit zu rechnen, dass immer mehr Knoten mit großen Inhalten nach oben wandern und somit die entstehenden Binärbäume ständig "links-lastiger" werden! (Umgekehrt würde bei ständiger Wahl des Inorder-Vorgängers ein immer rechtslastigerer Baum entstehen.)

Dies tritt in der Praxis auch tatsächlich ein. Es konnte sogar theoretisch gezeigt werden, dass nach etwa n^2 Einfüge- und Löschoperationen die mittlere Suchzeit bereits in der Größenordnung von "Wurzel(n)", also weit über $1{,}3863 \log(n)$ liegt. In der Praxis wählt man daher beim Löschen zufällig den Inorder-Vorgänger oder den Inorder-Nachfoger aus, um diesem Effekt der "Links-Rechts-Lastigkeit" entgegen zu wirken. (Zu weiteren analytischen Aussagen siehe das Buch von Ottmann und Widmayer, dort im Abschnitt 5.1.3.)

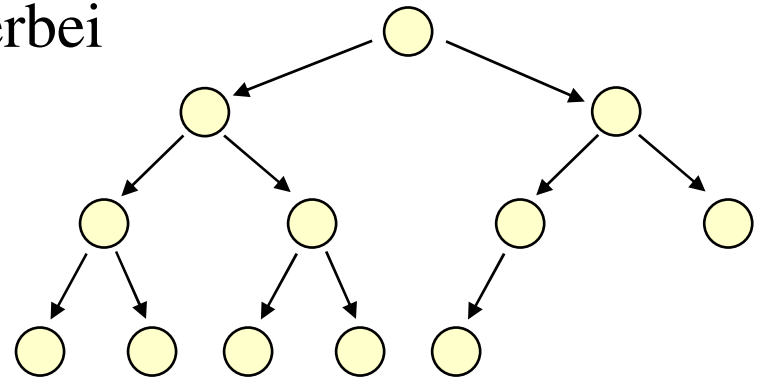
Kann man den Suchbaum so einschränken, dass Entartungen möglichst nicht eintreten können und auch im schlechtesten Fall nur $O(n \cdot \log(n))$ Schritte benötigt werden? Ja, siehe später AVL-Bäume.

Doch zunächst wollen wir uns den "besten" Suchbäumen zuwenden.

8.3 Optimale Suchbäume

Eine geordnete Folge von Elementen a_1, a_2, a_3, ..., a_n (mit $a_i \leq a_j$ für $i < j$) wird oft als Binärbaum gespeichert. Unter den C_n möglichen Binärbäumen wird man sich denjenigen auswählen, in dem man jedes Element möglichst schnell finden kann. Wird nach allen Elementen mit gleicher Wahrscheinlichkeit gesucht, so wird man einen möglichst gleichförmigen, einen sog. "ausgeglichenen" Baum nehmen, vgl. 8.4.11.

Ausgehend von der Wurzel wird hierbei die nächste Knotenschicht von links aufgefüllt, bis n Knoten vorhanden sind. Bis auf Verschiebungen von Knoten in der untersten Schicht sind diese Bäume eindeutig.



Einschub:
Man kann diese "ausgeglichenen" Bäume auch formal definieren. Im Vorgriff auf Abschnitt 8.4 sei diese Definition bereits hier angegeben,da wir sie in den Übungen diskutieren wollen:
Vorgriff auf die Definition 8.4.11:
Ein nicht-leerer, k-närer Baum [vgl. Definition 8.2.3] heißt ausgeglichener Baum, wenn es eine natürliche Zahl r gibt, so dass jeder Knoten, der mindestens einen null-Zeiger enthält, das Level r-1 oder r besitzt und es mindestens ein Blatt mit Level r gibt. (Diese Zahl r ist dann zugleich die Tiefe des ausgeglichenen Baums.)

Solche ausgeglichenen Bäume garantieren eine Suchzeit mit log(n)+1 Vergleichen, wobei n die Zahl der Knoten des Baumes ist. Liegen jedoch Informationen über die Häufigkeiten, mit denen auf die Knoten zugegriffen wird, vor, so kann man diese Suchzeit oft noch verringern.

Wir nehmen nun an, dass nach jedem Element a_i im Baum mit der Wahrscheinlichkeit (oder der Häufigkeit) p_i gesucht wird. Dann werden selbstverständlich *den* Binärbaum auswählen, für den die mittlere Suchdauer (= Summe der Suchzeiten gewichtet mit den Häufigkeiten) minimal ist (vgl. 8.2.13 a und b).

Definition 8.3.1: Gegeben sind eine geordnete Folge von n Elementen $(a_1, a_2, a_3, ..., a_n)$, mit $a_i \leq a_j$ für $i < j$, mit zugehörigen Häufigkeiten oder Wahrscheinlichkeiten $p_1, p_2, p_3, ..., p_n$ (insbesondere sind alle $p_i \geq 0$) sowie ein binärer Suchbaum B mit n Knoten $v_1, ..., v_n$, wobei a_i der Inhalt des Knotens v_i ist.

Die gewichtete mittlere Suchdauer S(B) von B ist definiert als

$$S(B) = \sum_{i=1}^{n} p_i \cdot \text{level}(v_i) \; .$$

Man kann in dieser Definition die Werte p_i als Häufigkeiten verwenden: Man führt p Anfragen durch und zählt hierbei, wie oft nach jedem Element a_i gefragt wurde; diese Anzahl sei jeweils p_i. Es gilt $p_1 + p_2 + p_3 + ... + p_n = p$.

Man kann in der Definition zum einen diese Werte p_i verwenden, man kann aber auch die relativen Häufigkeiten p_i/p benutzen; diese sind eine Näherung für die Wahrscheinlichkeit, dass nach dem i-ten Element a_i gesucht wird. Da wir später die Suchzeiten in Unterbäumen betrachten werden, brauchen wir von den Werten p_i nur zu wissen, dass $p_i \geq 0$ ist.

Das folgende Lösungsverfahren arbeitet sowohl mit Häufigkeiten als auch mit Wahrscheinlichkeiten.

Hinweis: Die in 8.2.13.a bereits definierte mittlere Suchzeit *ml*(B) eines Baumes B ist die gewichtete mittlere Suchdauer für den Fall, dass alle Elemente die gleiche Wahrscheinlichkeit $p_i = 1/n$ besitzen.

Hat man die Elemente $a_1, a_2, a_3, ..., a_n$ in einem Suchbaum B abgelegt und wird mit den Wahrscheinlichkeiten p_i nach ihnen gesucht, so gibt S(B) die zu erwartende Anzahl der Vergleiche an, um irgendein vorgegebenes Element a_j zu finden.

Hierbei suchen wir zunächst nur nach Elementen a_j, die in der ursprünglichen Folge $(a_1, a_2, a_3, ..., a_n)$ vorkommen. Wird auch nach Elementen gesucht, die nicht zu den a_i gehören, so muss man die Bäume leicht abändern; dies wird am Ende dieses Abschnitts 8.3 im Hinweis 4 erläutert.

Bevor wir die Definition eines optimalen Suchbaums angeben, erinnern wir an 8.2.9:

Gegeben sei eine sortierte Folge $a_1, a_2, a_3, ..., a_n$. Dann gibt es zu jedem binären Baum B mit n Knoten $v_1, v_2, v_3, ..., v_n$ genau eine Bijektion $f: \{a_1, a_2, a_3, ..., a_n\} \rightarrow \{v_1, v_2, v_3, ..., v_n\}$ der Folgenelemente a_i zu den Knoten v_j, so dass B hierdurch zu einem Suchbaum wird und der Inorder-Durchlauf von B die sortierte Folge $a_1, a_2, a_3, ..., a_n$ liefert. (Hierbei ist $f^{-1}(v_i)$ der Inhalt des Knotens v_i.)

Jeder solche binäre Baum B mit der Zuordnung f heißt ein zur Folge $a_1, a_2, a_3, ..., a_n$ gehörender Suchbaum (mit dem Inhalt f).

Ein Suchbaum, dessen mittlere Suchdauer unter allen zur Folge gehörenden Suchbäumen minimal ist, heißt optimal. Formal:

Definition 8.3.2: Gegeben sei eine geordnete Folge von n Elementen $a_1, a_2, a_3, ..., a_n$ (mit $a_i \leq a_j$ für $i < j$) mit ihren Wahrscheinlichkeiten $p_1, p_2, p_3, ..., p_n$ ($p_i \geq 0$ und $p_1+p_2+...+p_n = 1$) oder Häufigkeiten $p_1, p_2, p_3, ..., p_n$ ($p_i \geq 0$).

Ein Suchbaum B mit n Knoten $v_1, ..., v_n$, wobei a_i der Inhalt des Knotens v_i ist, heißt optimaler Suchbaum (zur Folge $a_1, a_2, ..., a_n$ mit ihren Wahrscheinlichkeiten bzw. Häufigkeiten), wenn für alle zur Folge $a_1, a_2, a_3, ..., a_n$ gehörenden Suchbäume B' gilt: $S(B) \leq S(B')$.
In dieser Definition sind nicht die konkreten Elemente a_i, sondern nur deren Wahrscheinlichkeiten bzw. Häufigkeiten p_i von Bedeutung.

Die Aufgabe lautet nun, zu n und $p_1, p_2, p_3, ..., p_n$ einen optimalen Suchbaum zu konstruieren. Da es C_n mögliche Suchbäume gibt (Satz 8.2.7), dauert das systematische Durchprobieren viel zu lange.

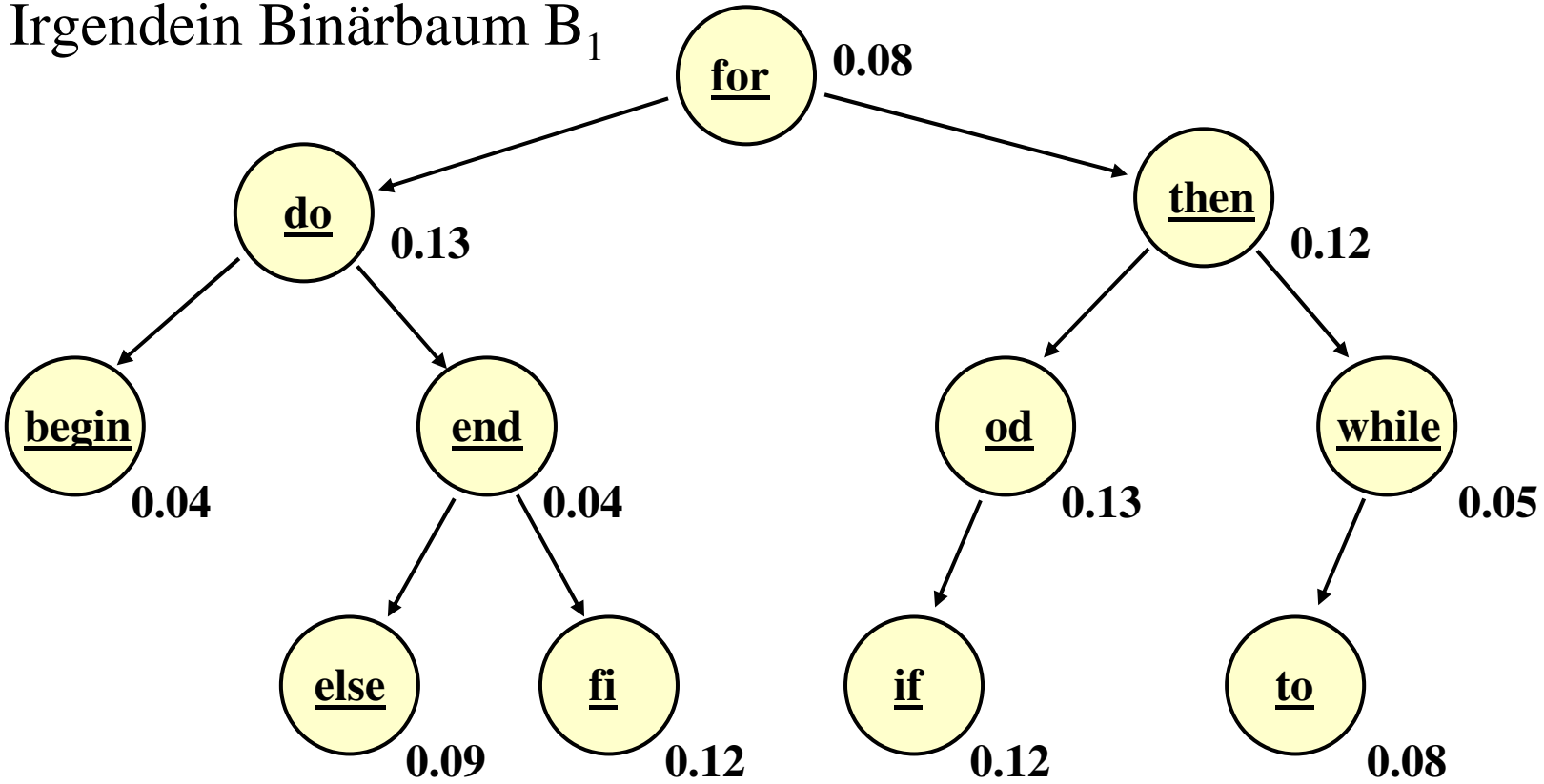
8.3.3 Beispiel

Als Beispiel betrachten wir einen Compiler, der ein Programm übersetzen soll. Hierzu muss er zunächst die Schlüsselwörter der Sprache erkennen. Diese Wörter treten mit gewissen Wahrscheinlichkeiten in einem Programm auf und wir nehmen an, wir hätten diese Wahrscheinlichkeiten gemessen. Wir verwenden hier die Folge aus n=11 Wörter begin, do, else, end, fi, for, if, od, then, to, while. Ihre Wahrscheinlichkeiten seien

begin: 0.04, do: 0.13, else: 0.09, end: 0.04, fi: 0.12, for: 0.08, if: 0.12, od: 0.13, then: 0.12, to: 0.08, while: 0.08.

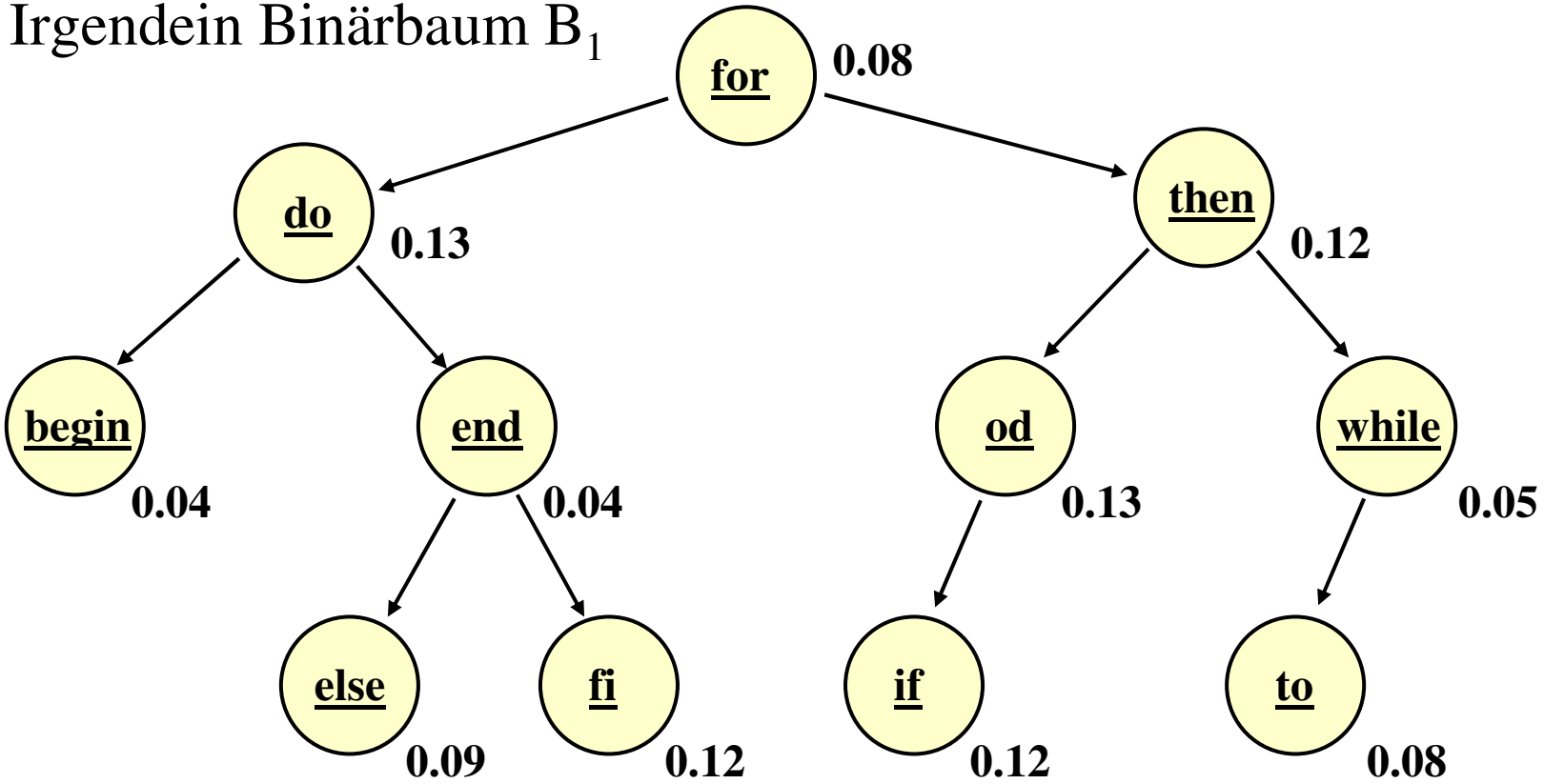
Wir fügen die 11 Folgenelemente zunächst in einen beliebigen Binärbaum B_1 ein und berechnen dessen gewichtete mittlere Suchdauer $S(B_1)$.

Irgendein Binärbaum B_1



Schlüsselwörter eintragen (inorder-Durchlauf)
Wahrscheinlichkeiten hinzufügen
Gewichtete mittlere Suchdauer nun ausrechnen (bitte selbst durchführen, dann erst zur nächsten Folie klicken).

Irgendein Binärbaum B_1

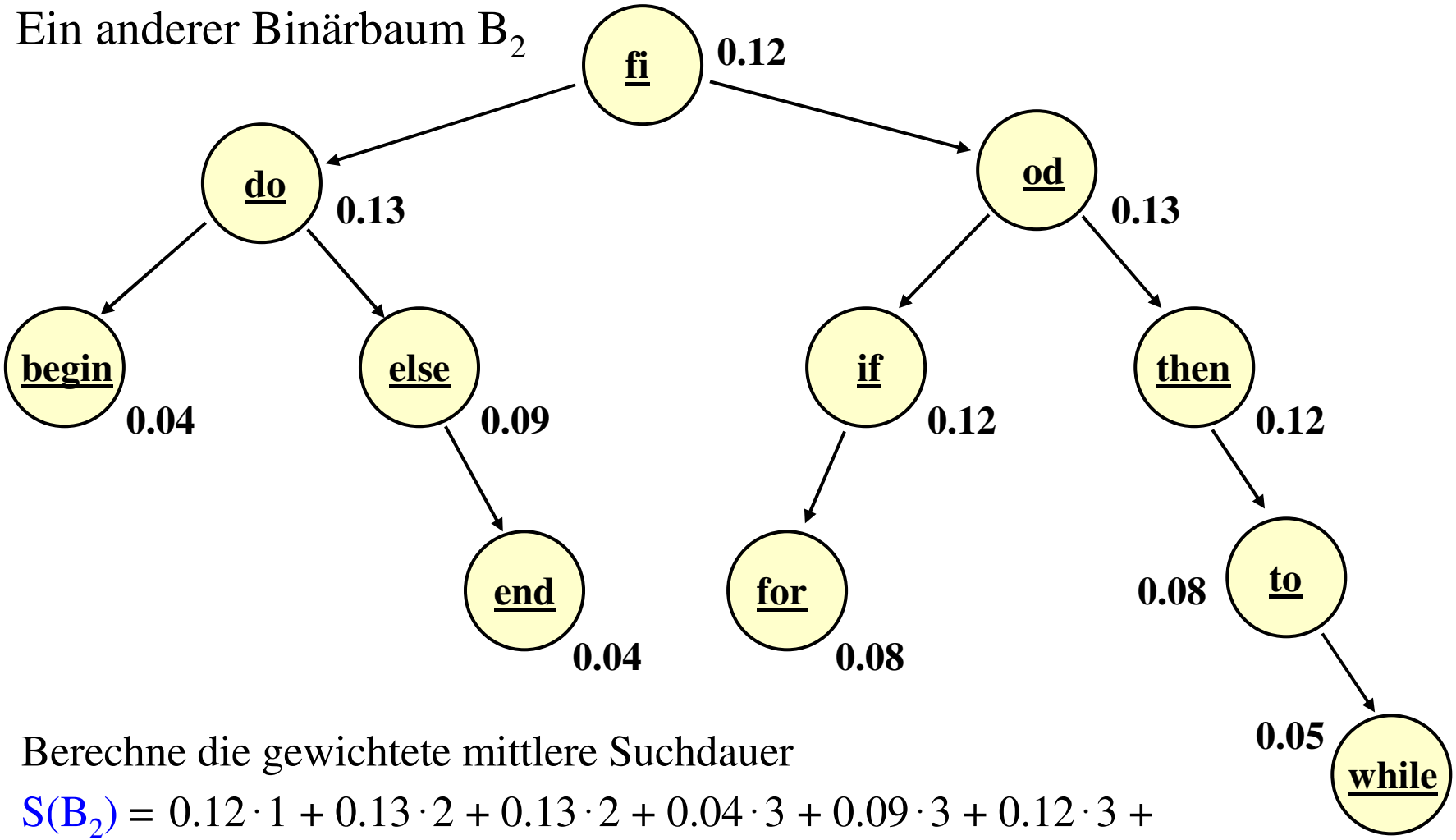


Berechne die gewichtete mittlere Suchdauer

$S(B_1) = 0.08 \cdot 1 + 0.13 \cdot 2 + 0.12 \cdot 2 + 0.04 \cdot 3 + 0.04 \cdot 3 + 0.13 \cdot 3 + 0.05 \cdot 3 +$
$0.09 \cdot 4 + 0.12 \cdot 4 + 0.12 \cdot 4 + 0.08 \cdot 4 = 3.00$

Konstruieren Sie nun einen Suchbaum mit kleinerem S(B). [Man sieht sofort, dass man to ein Level hinauf und while eines hinab schieben sollte, usw.]

Ein anderer Binärbaum B_2

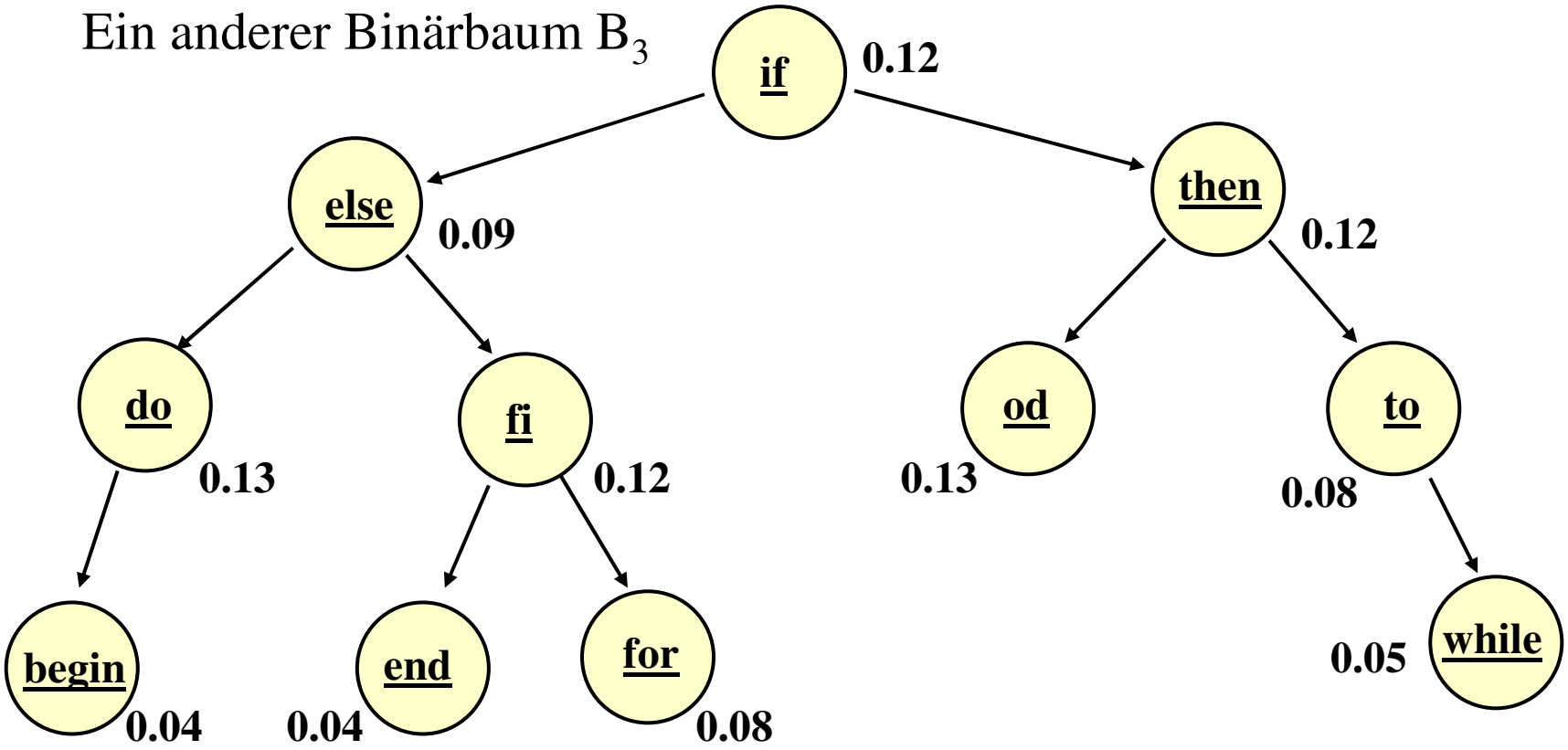


Berechne die gewichtete mittlere Suchdauer

$S(B_2) = 0.12 \cdot 1 + 0.13 \cdot 2 + 0.13 \cdot 2 + 0.04 \cdot 3 + 0.09 \cdot 3 + 0.12 \cdot 3 +$
$0.12 \cdot 3 + 0.04 \cdot 4 + 0.08 \cdot 4 + 0.08 \cdot 4 + 0.05 \cdot 5 = 2.80$

Konstruieren Sie nun weitere Suchbäume mit kleinerem S(B). Versuchen Sie, den optimalen Suchbaum zu finden. Erkennen Sie hierbei ein Verfahren?

Ein anderer Binärbaum B_3



Berechnen Sie selbst die gewichtete mittlere Suchdauer dieses Suchbaums. Ist dieser Baum B_3 besser als B_2? Gibt es bessere? Welche? Sollte man das mittlere Element der Folge (hier: for) möglichst in die Wurzel setzen?

■

Hilfssatz 8.3.4:

Jeder Unterbaum eines optimalen Suchbaums ist ebenfalls ein optimaler Suchbaum.

Beweis: Wäre der Unterbaum U eines optimalen Suchbaums B nicht optimal, so gäbe es einen anderen Suchbaum U', der bzgl. der in U enthaltenen Elemente optimal ist, für den insbesondere $S(U') < S(U)$ gilt. Tausche dann im Baum B den Unterbaum U gegen den Unterbaum U' aus, wodurch der Baum B' entsteht. Es gilt dann ($a_i \in U$ bezeichnen die Elemente, die im Unterbaum U liegen; x+1 sei das Level der Wurzel von U in B; in U und U' liegen natürlich die gleichen Elemente):

$$S(B) = \sum_{i=1}^{n} p_i \cdot level(v_i) = \sum_{a_i \in B-U} p_i \cdot level(v_i) + \sum_{a_i \in U} p_i \cdot level(v_i)$$

$$S(B) = \sum_{a_i \in B-U} p_i \cdot level(v_i) + \overbrace{\sum_{a_i \in U} p_i \cdot (level(v_i) - x)}^{S(U)} + \sum_{a_i \in U} p_i \cdot x$$

Hierbei ist x+1 das Level der Wurzel des Unterbaums U in B; dies ist zugleich das Level der Wurzel des Unterbaums U' in B'.

$$S(B) = \sum_{a_i \in B-U} p_i \cdot level(v_i) + S(U) + \sum_{a_i \in U} p_i \cdot x$$

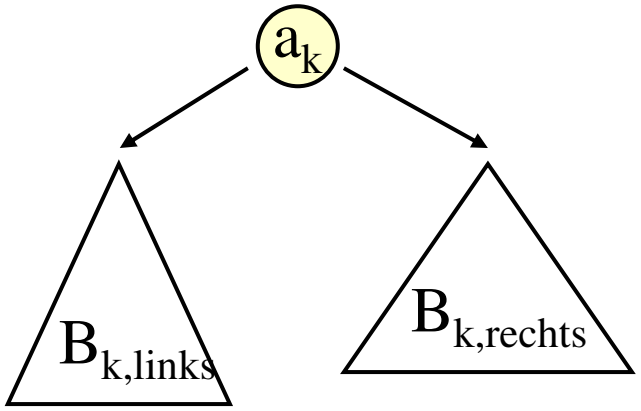
$$> \sum_{a_i \in B'-U'} p_i \cdot level(v'_i) + S(U') + \sum_{a_i \in U'} p_i \cdot x$$

$$= \sum_{a_i \in B'-U'} p_i \cdot level(v'_i) + \sum_{a_i \in U'} p_i \cdot (level(v'_i) - x) + \sum_{a_i \in U'} p_i \cdot x = S(B')$$

Also war B kein optimaler Suchbaum, im Widerspruch zur Voraussetzung. Folglich muss U optimal gewesen sein. ■

Hieraus folgt, dass man einen optimalen Suchbaum schrittweise aus seinen (optimalen) Unterbäumen aufbauen kann.

Um den optimalen Suchbaum für $a_i, a_{i+1}, ..., a_{j-1}, a_j$ zu finden, prüft man alle Paare von Unterbäumen für $a_i, a_{i+1}, ..., a_{k-1}$ und $a_{k+1}, a_{i+2}, ..., a_j$ durch und wählt die Kombination aus, deren Summe den kleinsten Wert ergibt. Skizze hierzu:



Der beste Fall liegt vor, wenn $S(B_{k,links}) + S(B_{k,rechts})$ minimal ist. Ein solches k ist also zwischen i und j zu suchen.

Dieser Baum enthält genau die Elemente $a_i, a_{i+1}, ..., a_{j-1}, a_j$ und es ist $i \leq k \leq j$.

8.3.5 Bezeichnungen und Formeln: Gegeben seien n und die Häufigkeiten P[1], P[2], ..., P[n]. Sei $1 \le i \le j \le n$.

Es sei G[i,j] = P[i] + P[i+1] + ... + P[j] (man bezeichnet diesen Wert auch als das "Gewicht" der Teilfolge a_i bis a_j).

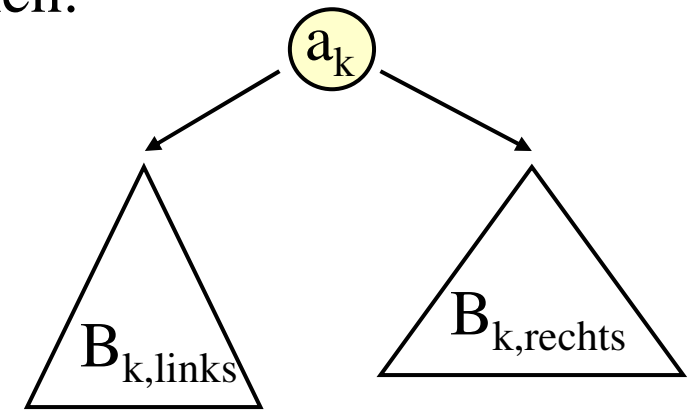
Mit S[i,j] bezeichnen wir die gewichtete mittlere Suchdauer für einen optimalen Suchbaum für die Elemente $a_i, a_{i+1}, ..., a_{j-1}, a_j$.

S[i,j] lässt sich leicht rekursiv berechnen:

S[i,i] = P[i] für i = 1, 2, ..., n.

Die Formel für S[i,j] liest man leicht aus der nebenstehenden Skizze ab.

S[i,j] ist S[i,k-1] + S[k+1,j] plus der Erhöhung aller Level um 1 (dieser Summand ist genau G[i,j]):



$$S[i,j] = \text{Min}\{ S[i,k-1] + S[k+1,j] \mid i \le k \le j \} + G[i,j] \quad \text{für } i < j.$$

Den Wert S[i,j] kann man berechnen, wenn man alle Werte S[r,s] mit s-r < j-i kennt. Setze also diff:=j-i und berechne für diff=0,1,2,...,n-1 alle Werte S[i,i+diff].

Für diff=0 setze S[i,i] =P[i]. Für diff > 0 verwende die Rekursionsformel für S[i,j] mit j=i+diff.

Zeitaufwand: Das eigentliche Verfahren benutzt drei ineinander geschachtelte Schleifen:

```
for diff in 1..n-1 loop
    for i in 1..n-diff loop
        j := i + diff;
        for k in i..j loop berechne das Minimum aller Werte
            S[i,k-1]+S[k+1,j]; end loop; setze S[i,j] entsprechend;
    end loop;
end loop;
```

Das gesuchte Ergebnis S(B) steht am Ende in S[1,n]. Da alle Schleifen linear von n abhängen, beträgt der Aufwand $\Theta(n^3)$.

Erläuterungen zum Programm:

Wir verwenden G[i,j] und S[i,j] wie angegeben, wobei wir rund 50% des Speicherplatzes verschwenden, da wir diese Werte nur für $i \leq j$ brauchen.
Weiterhin berechnen wir in diesem Programm auch die Wurzeln der Unterbäume zu a_i bis a_j. Diese legen wir in einem Feld R ab, wobei gilt

R[i,j] = k $\Leftrightarrow$ a_k steht in der Wurzel des optimalen Suchbaums von $a_i, a_{i+1}, ..., a_j$. (Dieses k wird im Programm in der innersten Schleife als k_min berechnet.)

Mit den R[i,j] kann man am Ende den optimalen Suchbaum re-konstruieren (dies programmieren wir aber nicht aus).
Die Minimumbildung erfolgt wie üblich, indem man alle Werte durchprobiert und das aktuelle Minimum speichert.
Die Werte G[i,j] kann man zu Beginn des Programms berechnen; wir führen dies jedoch in der mittleren Schleife durch.

8.3.6: Programm für einen optimalen Suchbaum in Zeit $\Theta(n^3)$

Global gegeben seien: Die Zahl n und n Häufigkeiten P[1], ..., P[n].

Ergebnis ist der Wert S[1,n] = die Suchdauer eines optimalen Suchbaums zu diesen Wahrscheinlichkeiten. Aus den Wurzeln R[i,j] der Teilbäume kann man den optimalen Suchbaum rekonstruieren. (Wie? Selbst hinzu programmieren!)

```
var  i, j, k, k_min, diff: natural;  min: real;
     S, G: array [1..n+1, 1..n] of real; R: array [1..n, 1..n] of natural;
begin  for i:=1 to n do
          S[i+1,i] := 0.0; S[i,i] := P[i]; G[i,i] := P[i]; R[i,i]:=i od;
       for diff:=1 to n-1 do
          for i:=1 to n-diff do
             j := i + diff; G[i,j] := G[i,j-1] + P[j]; min := S[i+1,j]; k_min := i;
             for k:=i+1 to j do
                if  S[i,k-1] + S[k+1,j] < min  then
                   min := S[i,k-1] + S[k+1,j]; k_min := k  fi  od;
          S[i,j] := min + G[i,j]; R[i,j] := k_min
       od od   -- Die gewichtete mittlere Suchdauer eines optimalen Suchbaums steht nun
end             -- in S[1,n]; der Suchbaum selbst kann aus den R[i,j] konstruiert werden.
```

8.3.7: Hinweise

Hinweis 1: Dieses ist ein "garantiertes" n^3-Verfahren. In der Praxis ist es deshalb nur für kleinere Werte von n einsetzbar.
Hinweis 2: Das Suchen (FIND) lässt sich optimal schnell durchführen. Dagegen muss man beim Einfügen (INSERT) und beim Löschen (DELETE) den Baum neu aufbauen. Daher setzt man optimale Suchbäume nur dann ein, wenn der Datenbestand sich über längere Zeiträume nicht ändert. Beispiele hierfür sind Lexika oder die Erkennung von Schlüsselwörtern und anderen Textteilen durch einen Compiler.
In zeitkritischen Anwendungen kann man eine Doppel-Strategie verfolgen: Der "große Datenbestand" ist in einem optimalen Suchbaum gespeichert, die (seltenen) Neueintragungen speichert man in einem gesonderten Binärbaum, bis dieser eine gewisse Größe erreicht hat; dann baut man aus den beiden Bäumen einen neuen optimalen Suchbaum auf.

Hinweis 3: Man kann nachweisen, dass die Wurzel des jeweils zu konstruierenden Unterbaums innerhalb bestimmter Grenzen liegen muss, die von den im letzten Durchlauf konstruierten Wurzeln bestimmt werden, genauer:
Es gilt stets $R[i,j-1] \leq R[i,j] \leq R[i+1,j]$. Dies nennt man die "Monotonie der Wurzeln".
Den Beweis finden Sie in Lehrbüchern oder in weiterführenden Vorlesungen (siehe "Effiziente Algorithmen" $(EA)^2$).
Mit dieser Eigenschaft lässt sich obiges Verfahren zu einem $\Theta(n^2)$-Verfahren beschleunigen. (Selbst durchdenken. Obiges Programm muss nur leicht modifiziert werden.)
Aber auch diese Beschleunigung reicht für die Praxis nicht aus, wenn sich der Datenbestand und/oder die Häufigkeiten oft ändern. In solchen Fällen verwendet man in der Praxis dann meist AVL-Bäume oder B-Bäume (siehe 8.4 und 8.5).

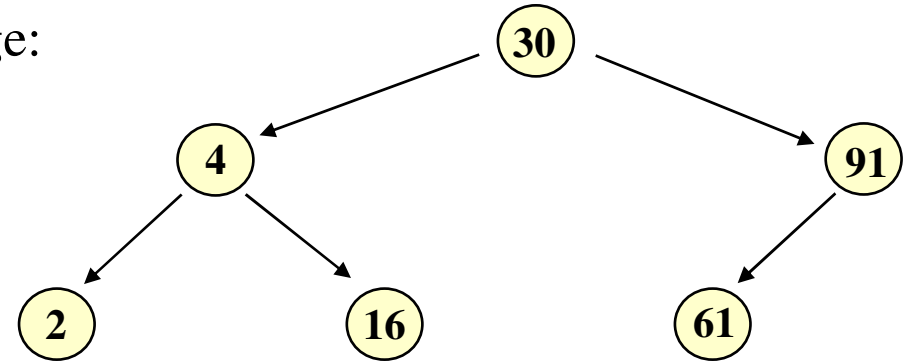
Hinweis 4: Zum Abschluss erläutern wir kurz, wie man dieses Verfahren auf den Fall der erfolglosen Suche erweitert.

Sucht man nach einem Element, das nicht in der Folge $a_1, a_2, a_3, ..., a_n$ vorkommt, so endet die Suche bei einem der n+1 null-Zeiger des Suchbaums.

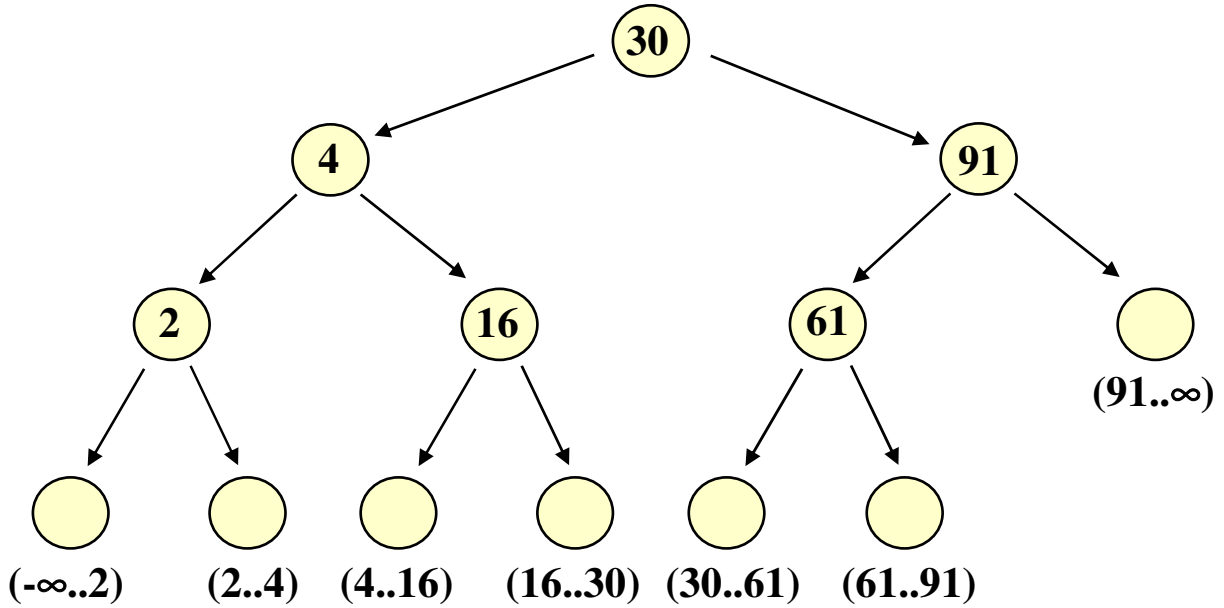
Man ersetzt nun diese null-Zeiger durch Knoten, deren Inhalt alle die Elemente bilden, mit denen man im Suchbaum hierhin gelangt. Ihre Inhalte sind daher offene Intervalle der Form $(i..j) = \{r \mid i < r < j\}$.

Jetzt muss man den Intervallen ebenfalls Häufigkeiten zuordnen, mit denen nach einem Element in ihnen gesucht wird. Auf diese Weise kann man einen optimalen Suchbaum auch für den Fall der "erfolglosen Suche" mit dem gleichen Verfahren konstruieren.

Beispiel: Gegebene Folge:
2, 4, 16, 30, 61, 91
und ein zugehöriger
Suchbaum.



Füge an den sieben null-Zeigern neue Blätter an, die die nicht enthaltenen Elemente repräsentieren; diese neuen Blätter werden mit den offenen Intervallen (i..j) beschriftet.



8.4 Balancierte Bäume, AVL-Bäume

Unter der Balance eines Knotens in einem Binärbaum versteht man ein *Verhältnis*, in dem seine beiden Unterbäume zueinander stehen.

Dieses „Verhältnis“ kann sehr verschieden festgelegt werden. Üblich sind zwei Festlegungen:

- Die *Gewichts-Balance* von u gibt die Knotenanzahl eines Unterbaums relativ zum gesamten Unterbaum an.
- Die *Höhen-Balance* von u gibt die Differenz der Höhen (bzw. Tiefen) der beiden Unterbäume von u an.

Entscheidend ist: Liegt die Balance in gewissen Bereichen, dann ist die Tiefe des Baums in O(log(n)). Dadurch kann die Suchzeit stets logarithmisch beschränkt werden.

Definition 8.4.1:

Es sei u ein Knoten und es seien UB_{links} und UB_{rechts} seine beiden Unterbäume. Es bezeichnen

$|V_{UB_{links}}|$ die *Anzahl der Knoten* im linken Unterbaum und

$|V_{UB_{rechts}}|$ die *Anzahl der Knoten* im rechten Unterbaum von u.

$|V_{UB_{links}}| + |V_{UB_{rechts}}|$ ist dann die Anzahl aller Knoten, die sich unterhalb des Knotens u befinden.

Definition 8.4.1 (Fortsetzung):

Es sei α eine Zahl aus dem reellen Intervall $(0, ½]$. Ein Knoten u eines Binärbaums heißt α-gewichtsbalanciert, wenn gilt

$$\alpha \leq \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2} \leq 1 - \alpha$$

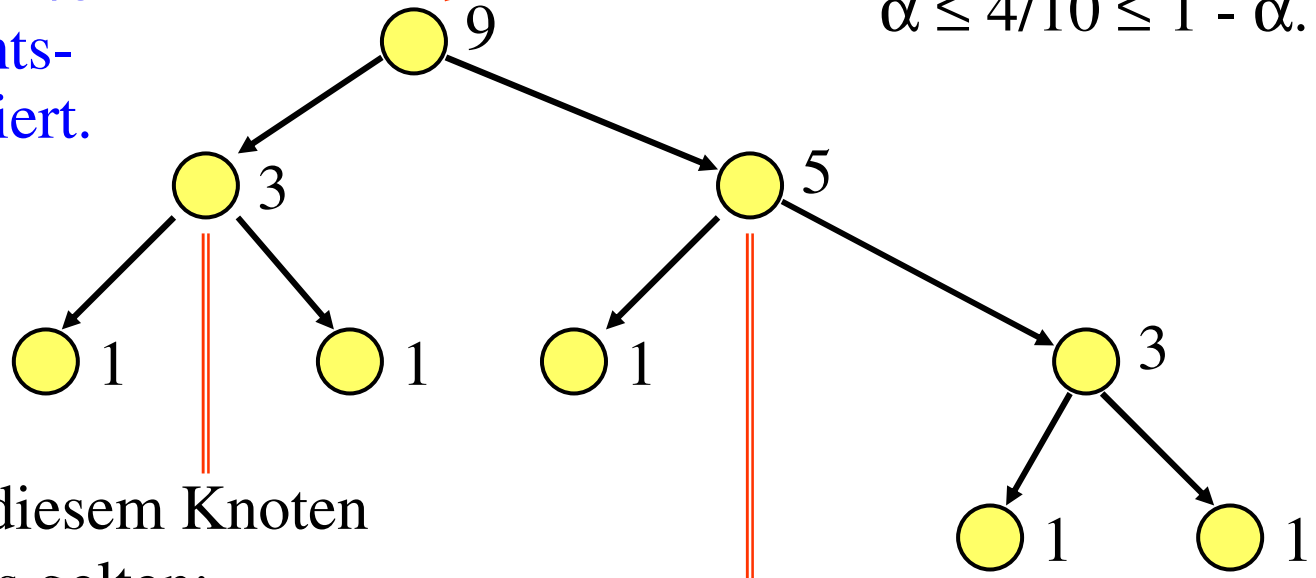
Den Ausdruck $\beta(u) = \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2}$

nennt man auch die Gewichts- oder Wurzelbalance von u.

Die Anzahlen der Knoten in den Unterbäumen werden hier jeweils um 1 erhöht, weil die Unterbäume leer sein können. Wenn ein Knoten α-gewichtsbalanciert ist und $0 \leq \alpha' \leq \alpha \leq ½$ gilt, dann ist der Knoten auch α'-gewichtsbalanciert.

Beispiel 8.4.2: An jedem Knoten sei die Zahl der Knoten in dem Unterbaum, dessen Wurzel er ist, angegeben. Bestimme für folgenden Baum ein geeignetes α.

Dieser Baum ist also 1/3-gewichts-balanciert.



Am Wurzelknoten muss gelten $\alpha \leq 4/10 \leq 1 - \alpha$.

An diesem Knoten muss gelten: $\alpha \leq 2/4 \leq 1 - \alpha$.

An diesem Knoten muss gelten: $\alpha \leq 2/6 \leq 1 - \alpha$.

Blätter sind stets 1/2-gewichts-balanciert.

Folgerung 8.4.3: Symmetrie der Balance

Wenn

$$\alpha \leq \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2} \leq 1 - \alpha$$

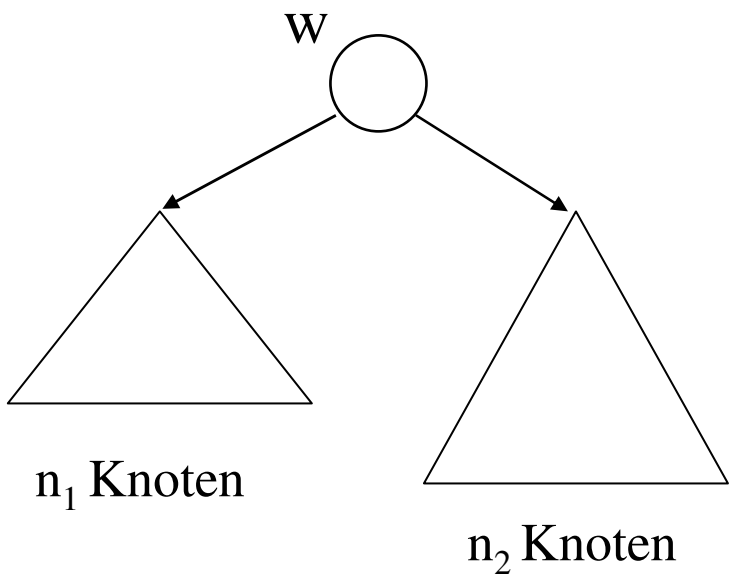
gilt, dann gilt auch

$$\alpha \leq \frac{|V_{UB_{rechts}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2} \leq 1 - \alpha$$

Der Beweis ist einfach: Seien $|V_{UB_{links}}| = n_1$ und $|V_{UB_{rechts}}| = n_2$, dann folgt die Aussage folgt sofort aus

$\alpha \leq (n_1+1)/(n_1+n_2+2) = 1 - (n_2+1)/(n_1+n_2+2) \leq 1 - \alpha.$ ■

Betrachte einen Binärbaum B mit n Knoten und der Wurzel w:



Die Gesamtzahl der Knoten im Baum ist $n = n_1 + n_2 + 1$.

Für α-gewichtsbalancierte Bäume B gilt dann an der Wurzel w für i = 1, 2:

$$\alpha \le \frac{n_i + 1}{n + 1} \le 1 - \alpha, \text{ d.h., } n_i \le (1 - \alpha)\cdot(n + 1) - 1 = (1 - \alpha)\cdot n - \alpha$$

Dies gilt auch für das nächste Level, d.h.:

Anzahl der Knoten in einem Unterbaum zwei Level tiefer

$\leq (1 - \alpha) \cdot n_i - \alpha$

$\leq (1 - \alpha) \cdot ((1 - \alpha) \cdot n - \alpha) - \alpha = (1 - \alpha)^2 \cdot n - \alpha \cdot ((1 - \alpha) + 1)$

Analog weiter einsetzen! Dies ergibt (beachte $\alpha > 0$):

Anzahl der Knoten in einem Unterbaum k Level tiefer

$\leq (1 - \alpha)^k \cdot n - \alpha \cdot ((1 - \alpha)^{k-1} + (1 - \alpha)^{k-2} + (1 - \alpha)^1 + (1 - \alpha)^0)$

$= (1 - \alpha)^k \cdot n - \alpha \cdot (1 - (1 - \alpha)^k) / (1 - (1 - \alpha))$

$= (1 - \alpha)^k \cdot n - (1 - (1 - \alpha)^k)$

$< (1 - \alpha)^k \cdot n$

Wenn $(1 - \alpha)^k \cdot n < 2$ geworden ist, gibt es höchstens noch einen Knoten und dieser muss dann ein Blatt sein.

Aus $(1-\alpha)^k \cdot n < 2$ folgt mit $z := 1/(1-\alpha)$:

$n < 2 \cdot z^k$, d.h., $\log(n/2) < k \cdot \log(z) = -k \cdot \log(1-\alpha)$

Suche das kleinste k, für das diese Ungleichung zutrifft:

$k = 1 - (\log(n) - 1) / \log(1-\alpha) \in O(\log(n))$.

Dieses (reelle) k ist eine obere Schranke für die Tiefe des Baums (minus 1). Somit haben wir gezeigt:

Satz 8.4.4

Die Tiefe eines α-gewichtsbalancierten Baums mit n Knoten ist höchstens $2 - (\log(n) - 1)/\log(1-\alpha)$,

d.h., die Tiefe ist stets von der Größenordnung $O(\log(n))$.

Je mehr α sich der Zahl 0.5 nähert, um so mehr nähert sich die Tiefe dem minimalen Wert $\lceil \log(n+1) \rceil$ (vgl. diskreter Zweierlogarithmus 8.2.11 und Folgerung 8.2.12).

Kann man die Operationen FIND, INSERT und DELETE auf solchen α-gewichtsbalancierten Bäumen so ausführen, dass diese Operationen schnell durchführbar sind (z.B. in O(log(n)) Schritten) und dass nach Ausführung jeder Operation der entstandene Baum wieder ein α-gewichtsbalancierter Baum ist?

Ja, das geht tatsächlich.

Wir führen dies hier jedoch nicht durch, sondern verweisen auf die Literatur.

An Stelle der gewichtsbalancierten Bäume untersuchen wir die höhenbalancierten Bäume genauer, die besonders angenehme Eigenschaften haben.

Für das Folgende beachten Sie, dass hier "Höhe" und "Tiefe" synonym verwendet werden (8.2.4). T bezeichnet die Tiefe.

Definition 8.4.5:

Ein binärer Baum heißt **AVL-Baum** (oder **höhenbalancierter** Baum), wenn für jeden Knoten u gilt:

$T(UB_{rechts}) - T(UB_{links}) =$

("Höhe" des rechten Unterbaums von u **-**

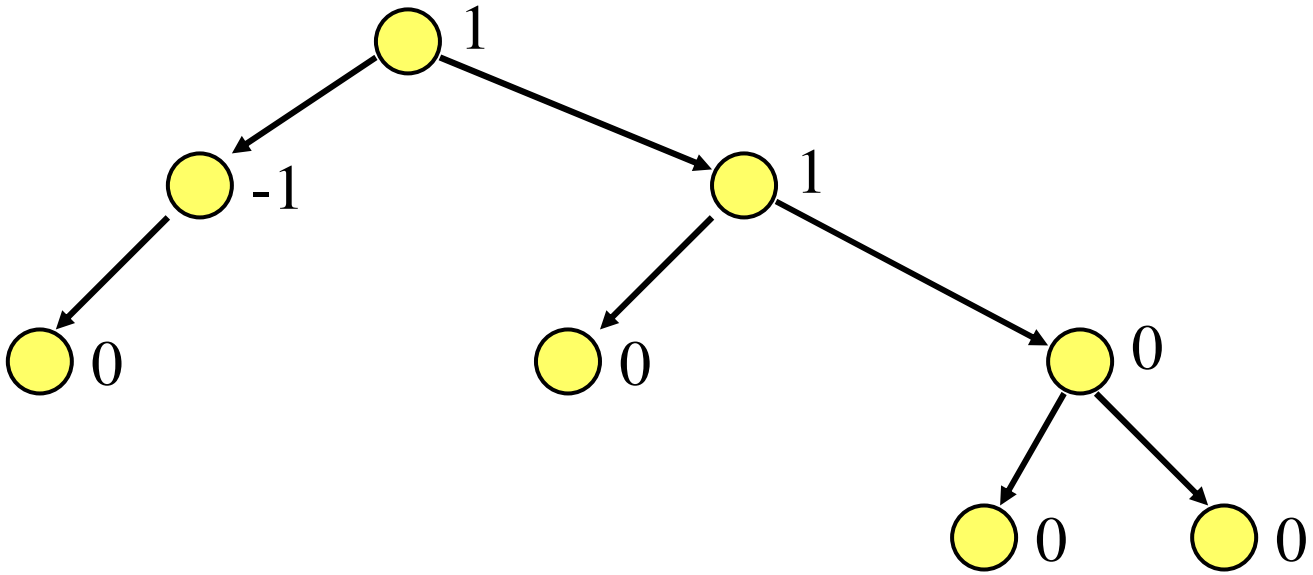
"Höhe" des linken Unterbaums von u) $\in \{-1, 0, 1\}$,

d.h., für jeden Knoten unterscheiden sich die Höhen (= Tiefen) seiner Unterbäume höchstens um 1.

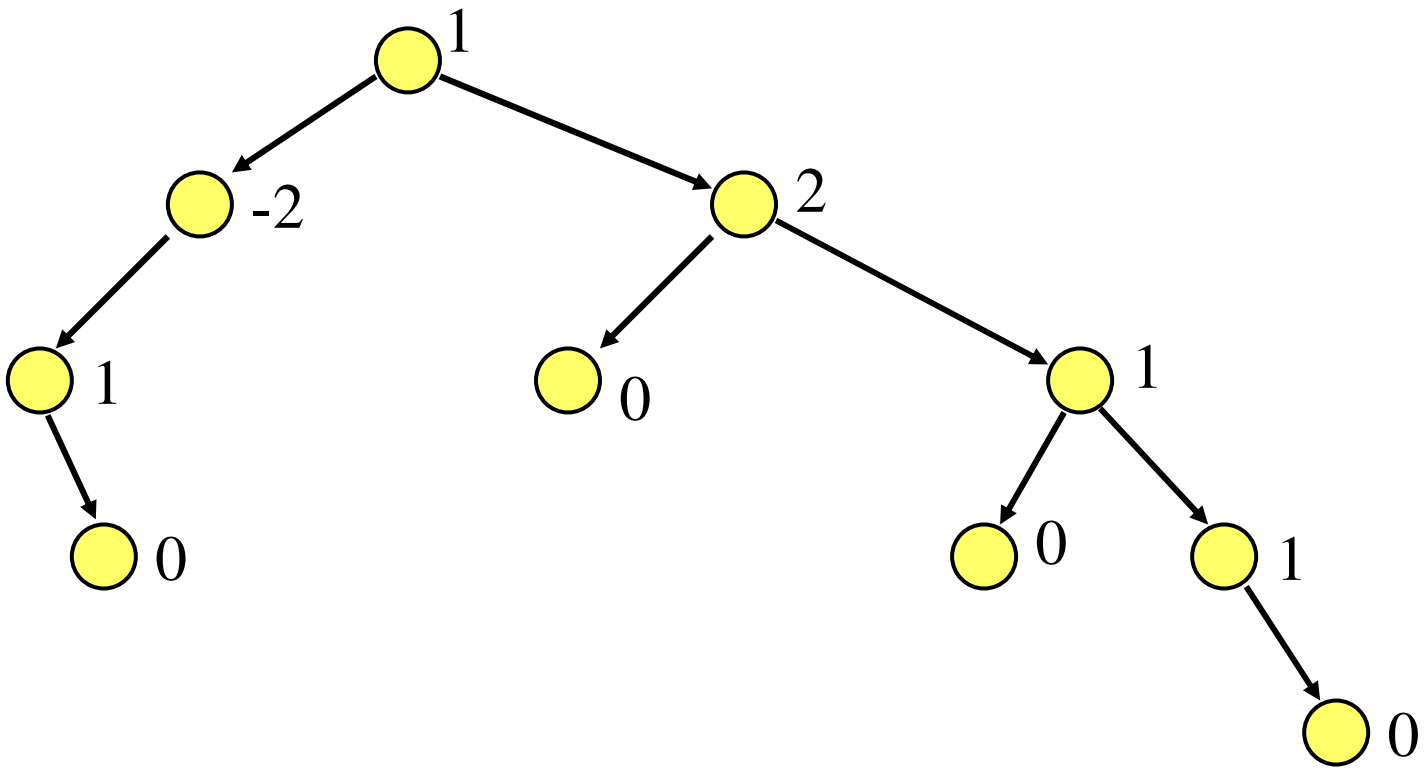
$T(UB_{rechts}) - T(UB_{links})$ heißt die Höhenbalance von u, oft auch Balancefaktor oder kurz Balance von u genannt.

AVL-Bäume wurden benannt nach ihren beiden Erfindern Adelson-Velski und Landis.

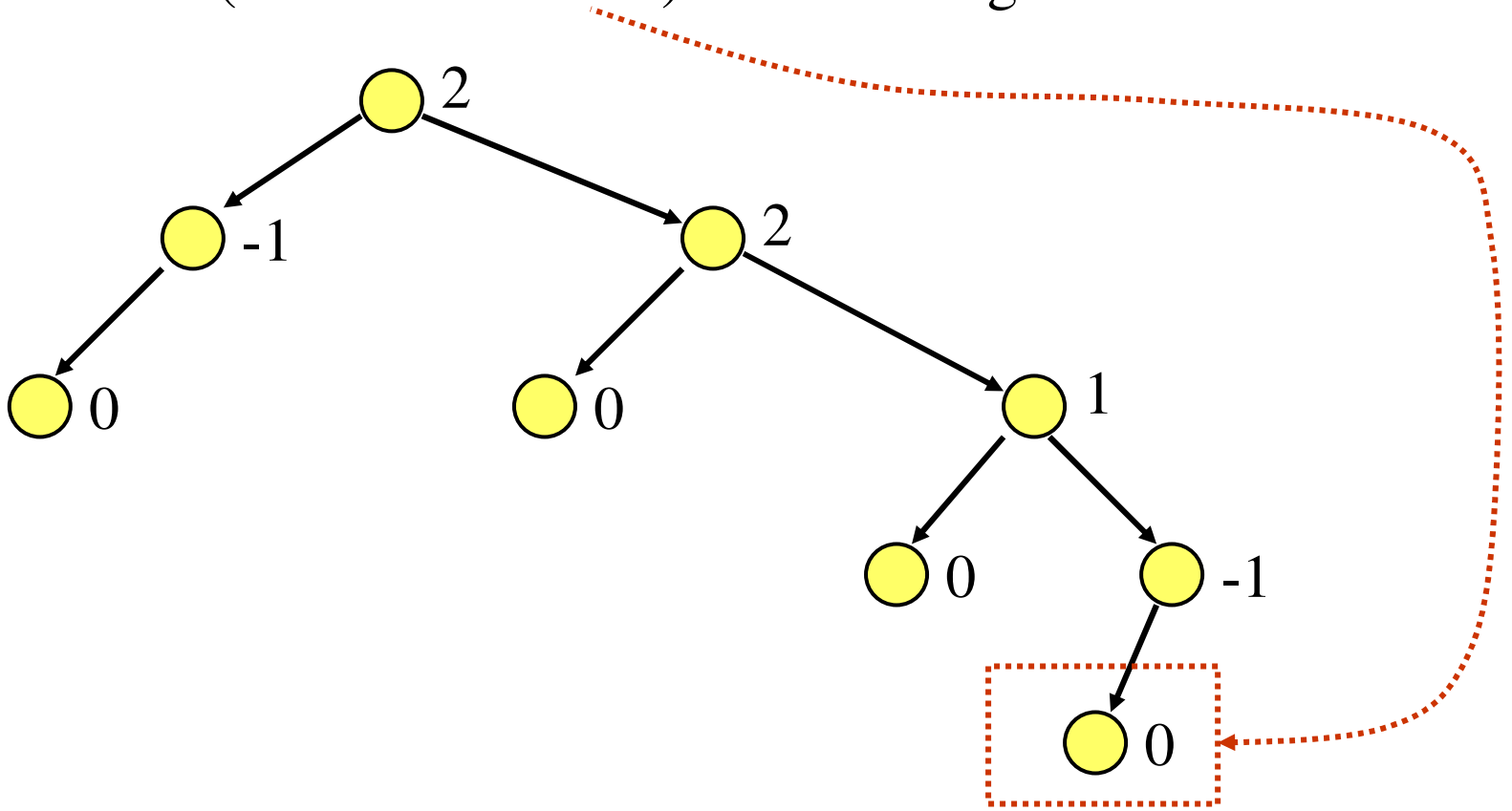
Beispiel: Der bereits im Beispiel 8.4.2 betrachtete Baum ist höhenbalanciert, d.h. ein AVL-Baum. Wir tragen die Höhenbalancen neben jedem Knoten ein:



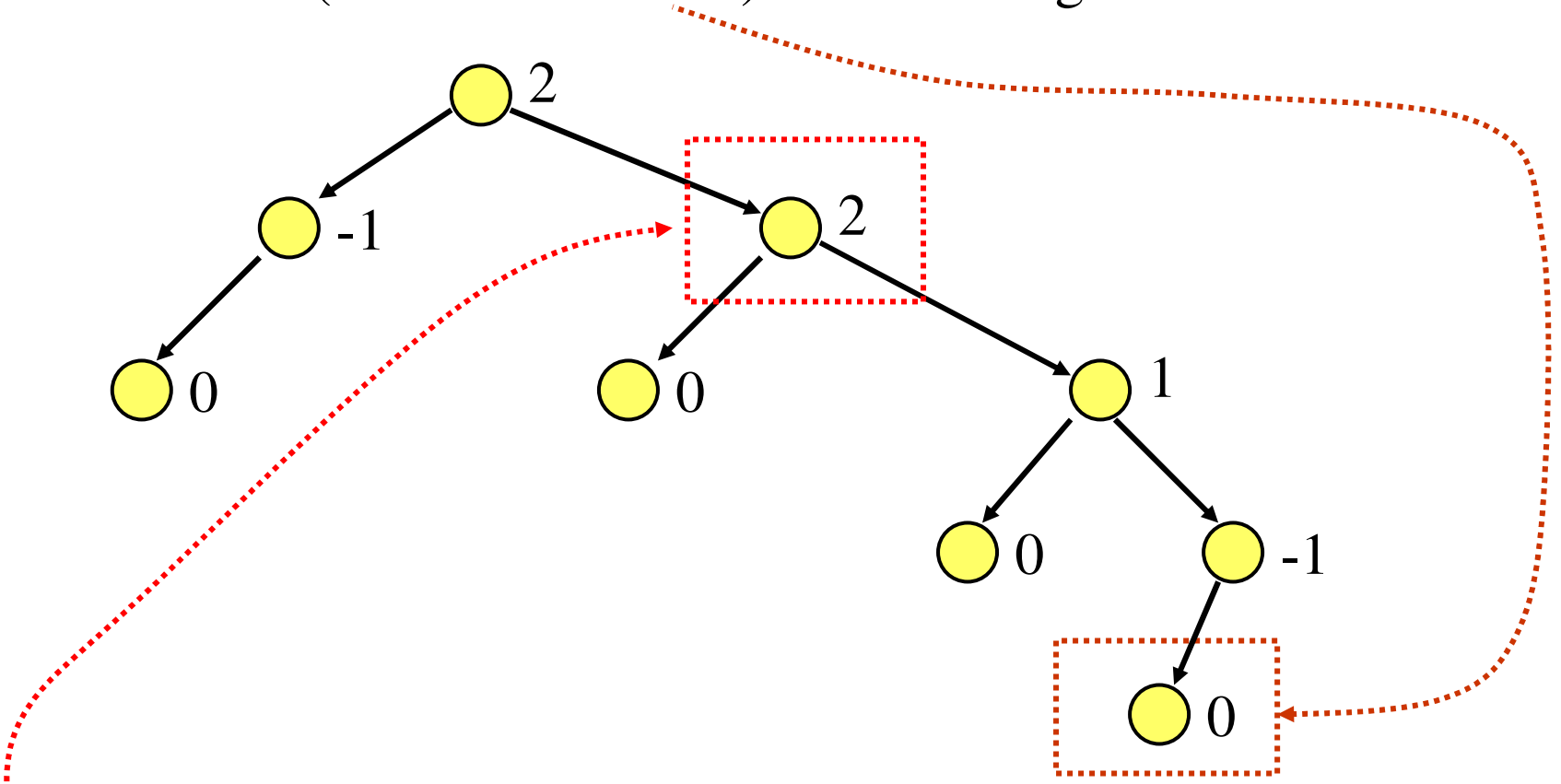
Beispiel: Dagegen ist folgender Baum *nicht* höhenbalanciert:



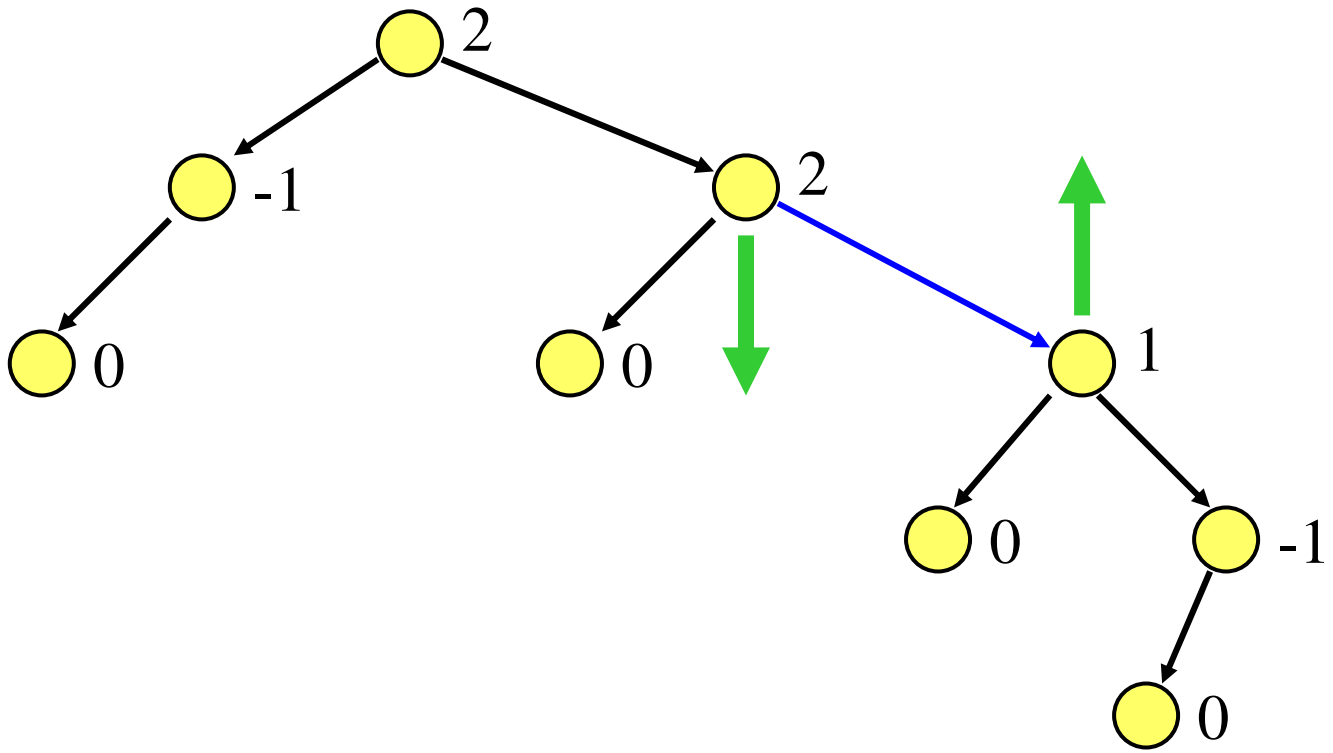
Betrachte nun einen AVL-Baum, in dem nur wegen eines Knotens (hier: unten rechts) die AVL-Eigenschaft verletzt ist:



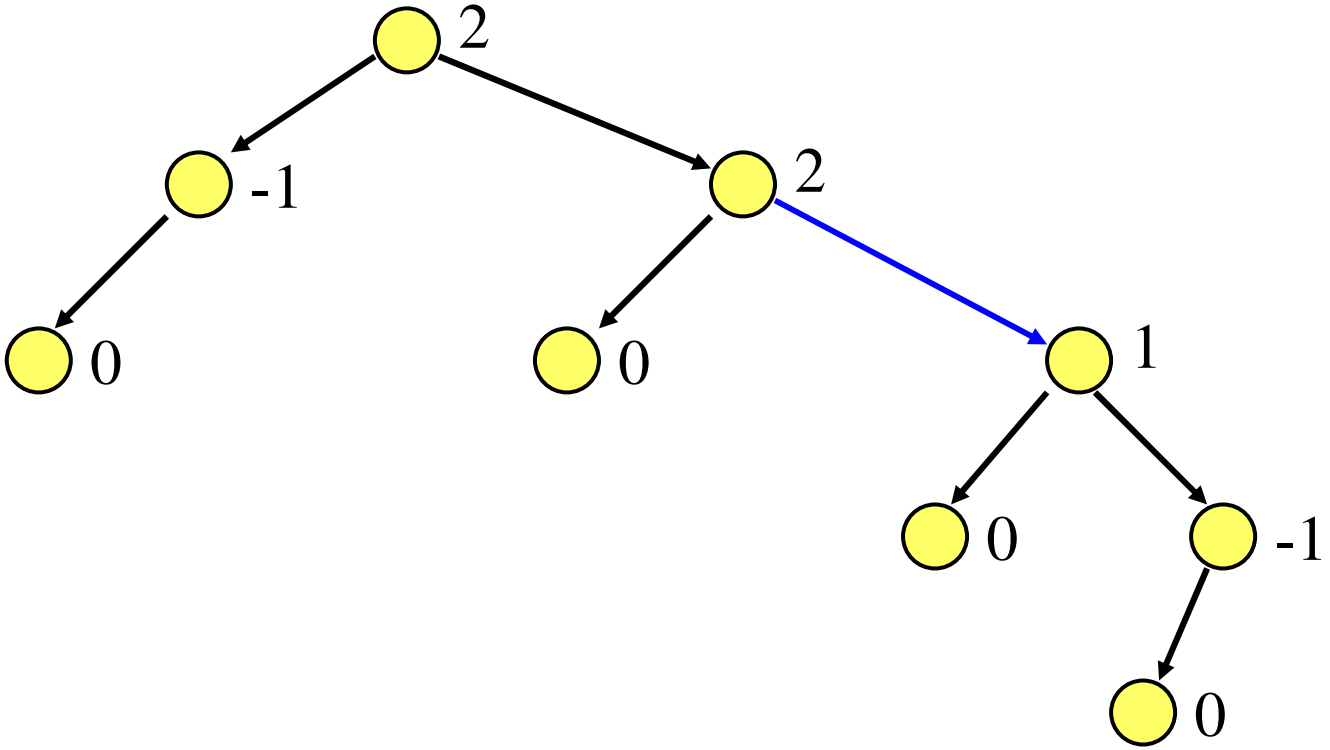
Betrachte nun einen AVL-Baum, in dem nur wegen eines Knotens (hier: unten rechts) die AVL-Eigenschaft verletzt ist:



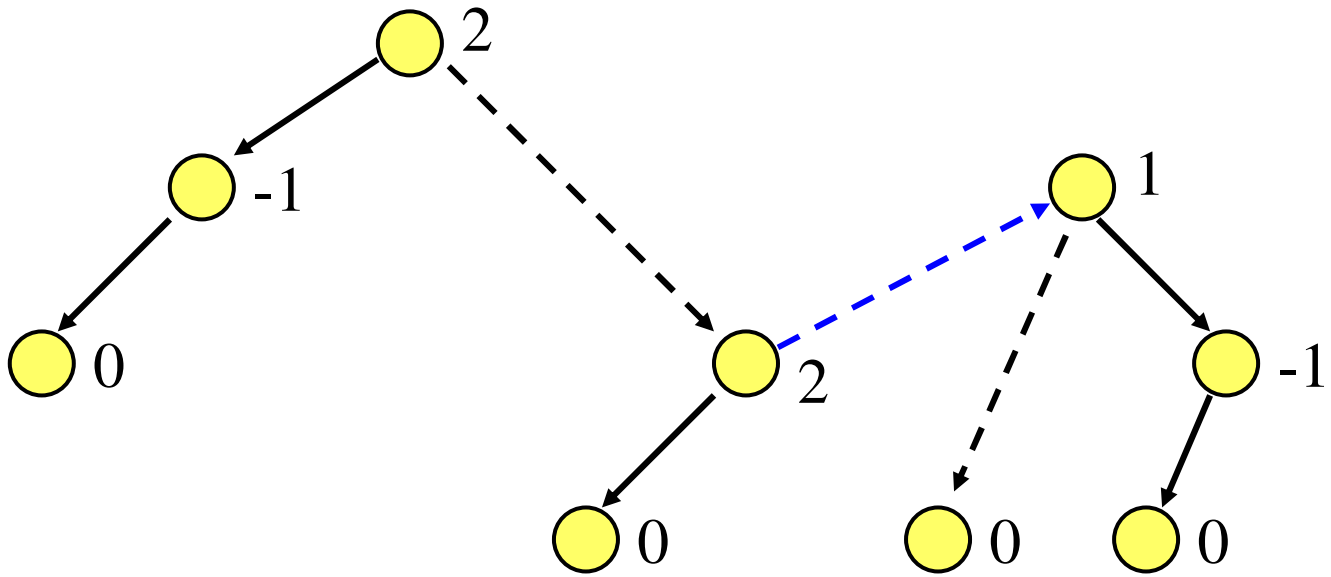
Dann kann man durch eine leichte Korrektur an der untersten Verletzungs-Position die AVL-Eigenschaft wieder herstellen:



Dies nennt man eine "Links-Rotation" (= Drehung der blauen Kante nach links, also gegen den Uhrzeigersinn).

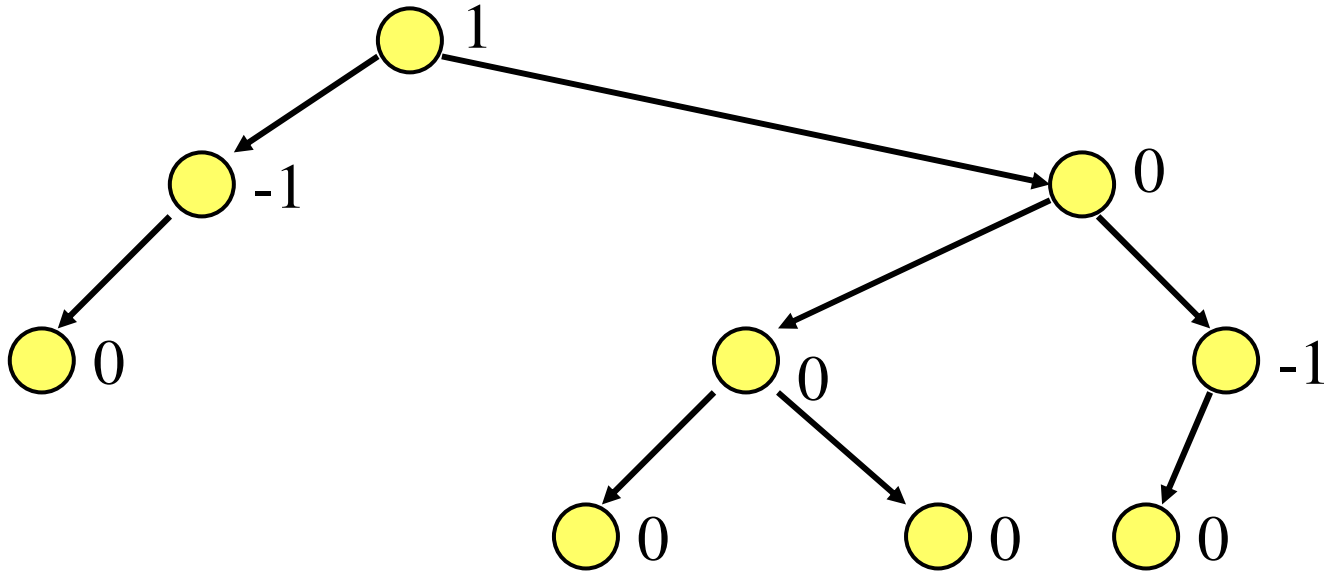

2
-1
2
0
0
1
0
-1
0

Nur die drei gestrichelten Kanten werden verzerrt.



Nun muss man die drei gestrichelten Kanten neu orientieren, wobei die Suchbaum-Eigenschaft erhalten bleiben muss,

und schon liegt wieder ein AVL-Baum vor:



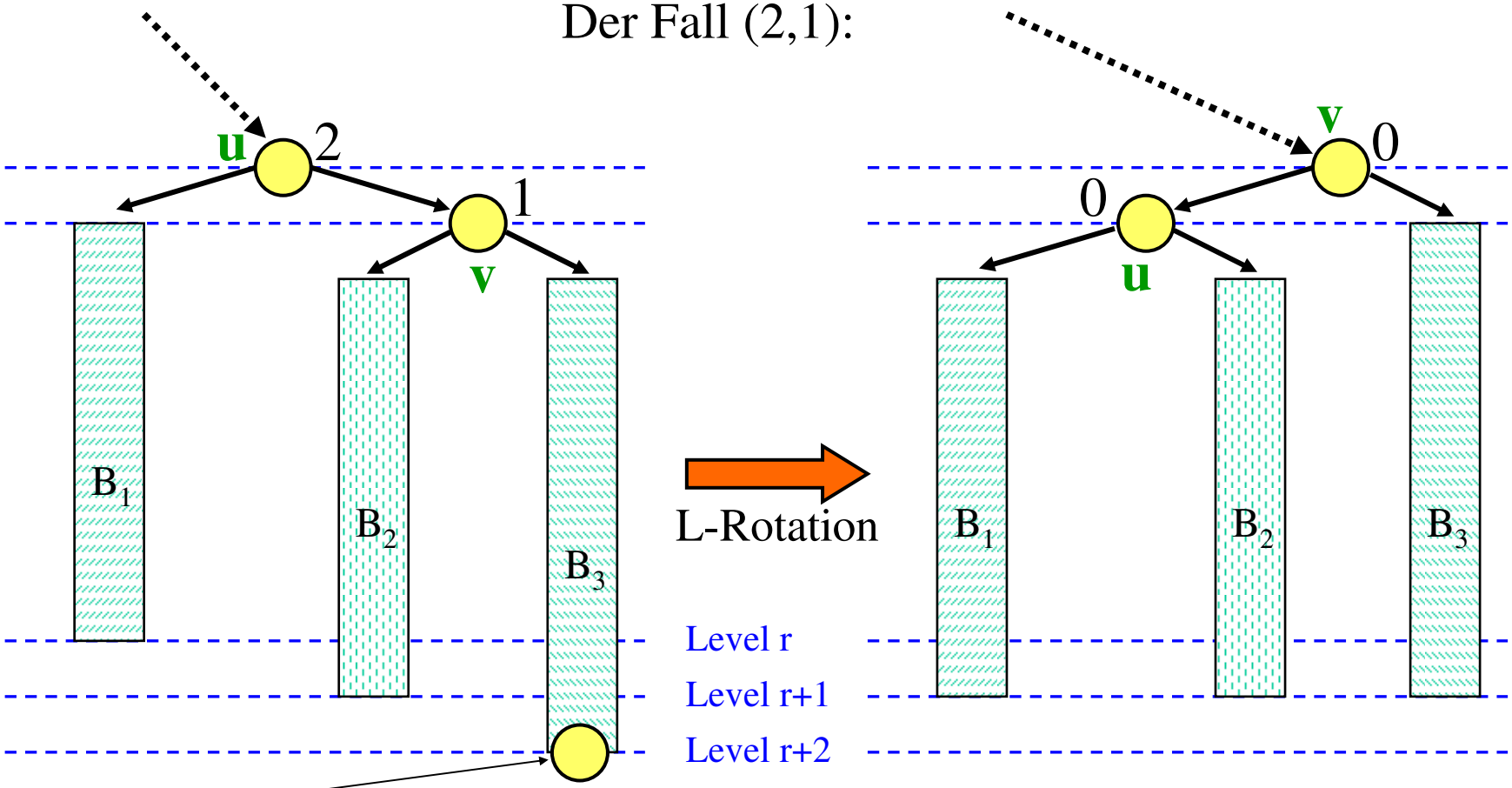
Durch Ausprobieren stellt man fest, dass man mit vier Rotationsarten auskommt, um alle solche Fälle zu korrigieren.

Wir betrachten das Einfügen in einen AVL-Baum. Dies geschieht wie bei einem beliebigen Suchbaum stets in einem Blatt. Dabei kann durch den neu eingefügten Knoten die AVL-Eigenschaft der Höhenbalancierung verletzt werden.

In diesem Fall führen wir eine Rotation an dem untersten Knoten, der nicht mehr höhenbalanciert ist, durch, Dadurch wird die AVL-Eigenschaft wieder hergestellt (siehe unten).

Wir untersuchen nun die möglichen Fälle, die zu einer Verletzung führen. Da durch einen Knoten das Level höchstens um 1 steigen kann, muss die unterste Verletzung an einem Knoten u in einer Balance 2 oder -2 bestehen. Dann muss aber der direkte Nachfolgeknoten von u, der auf dem Weg zum eingefügten Blatt liegt, jetzt 1 oder -1 geworden sein (wäre er 0, hätte sich die Tiefe dieses Unterbaums nicht geändert und damit hätte sich auch nicht die Balance von u ändern können). Somit gibt es die vier Fälle (2,1), (2,-1), (-2,1) und (-2,-1).

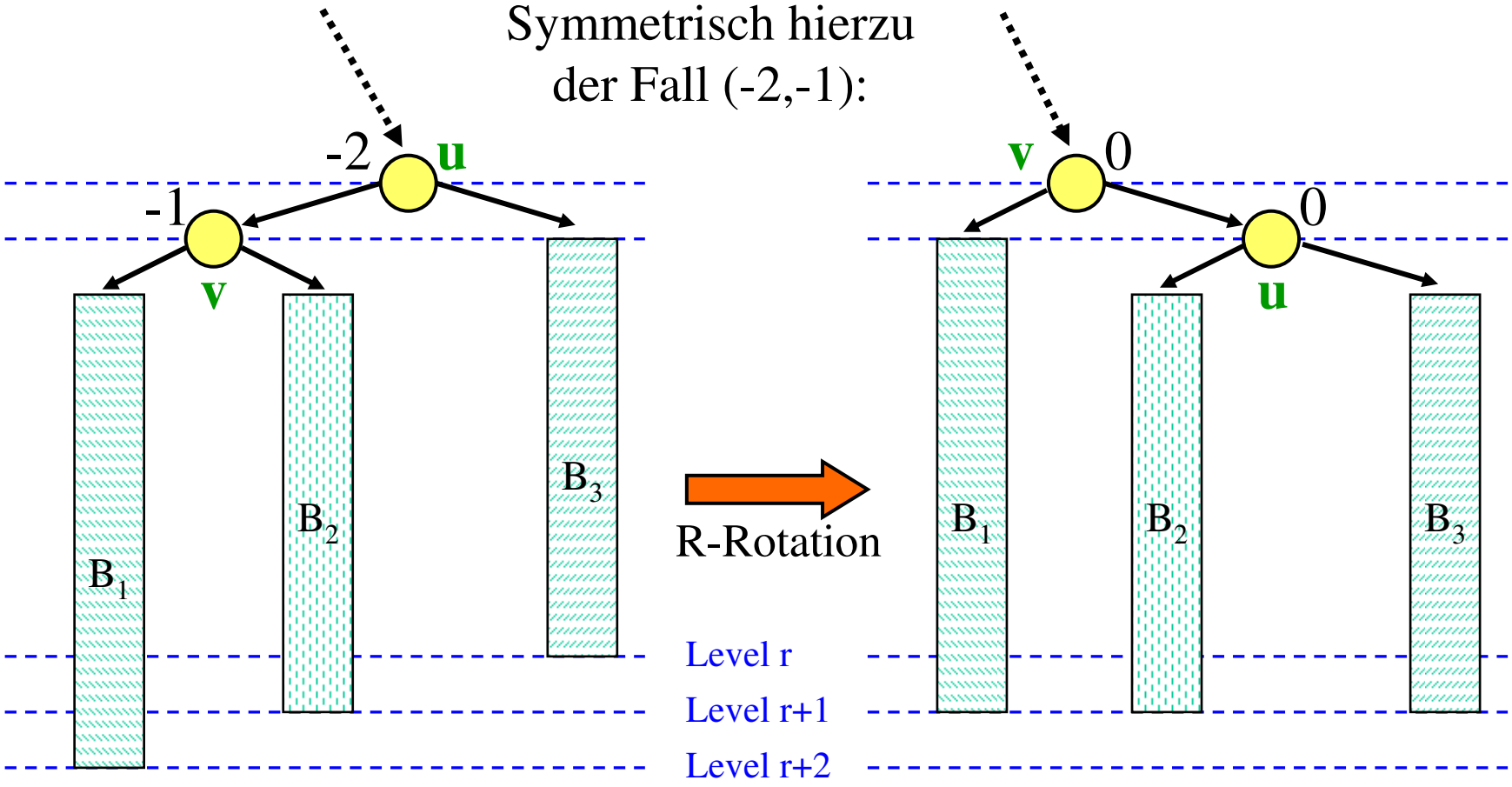
Der Fall (2,1):



Dieses Blatt wurde eingefügt, wodurch die neuen Balancen 2 und 1 bei u und v entstanden.

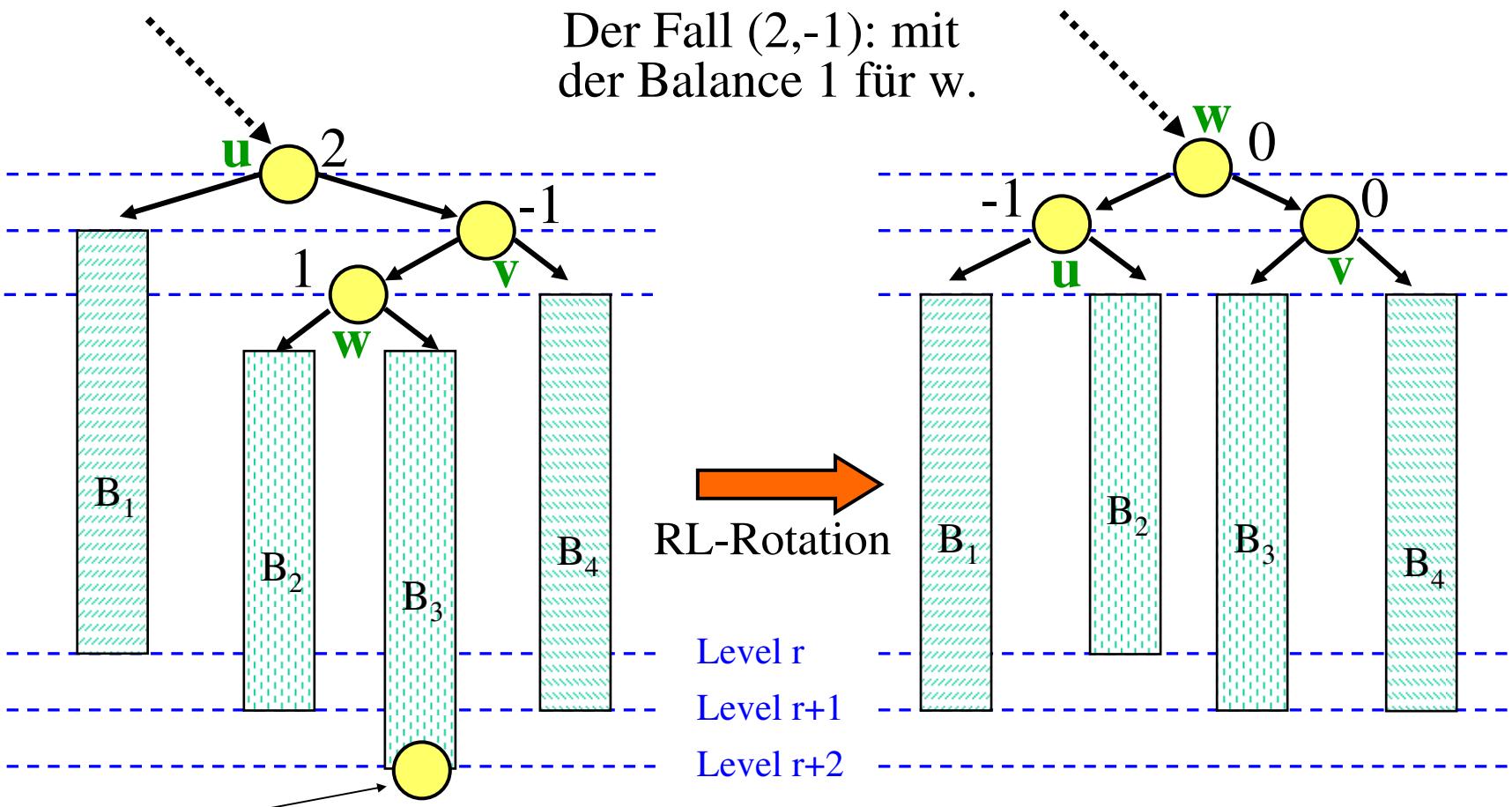
Situation an der untersten Verletzungsstelle **u**—**v**: Balancen 2 und 1.

⇒ Links-Rotation durchführen (3 Zeiger umhängen, Balancen 0 und 0).



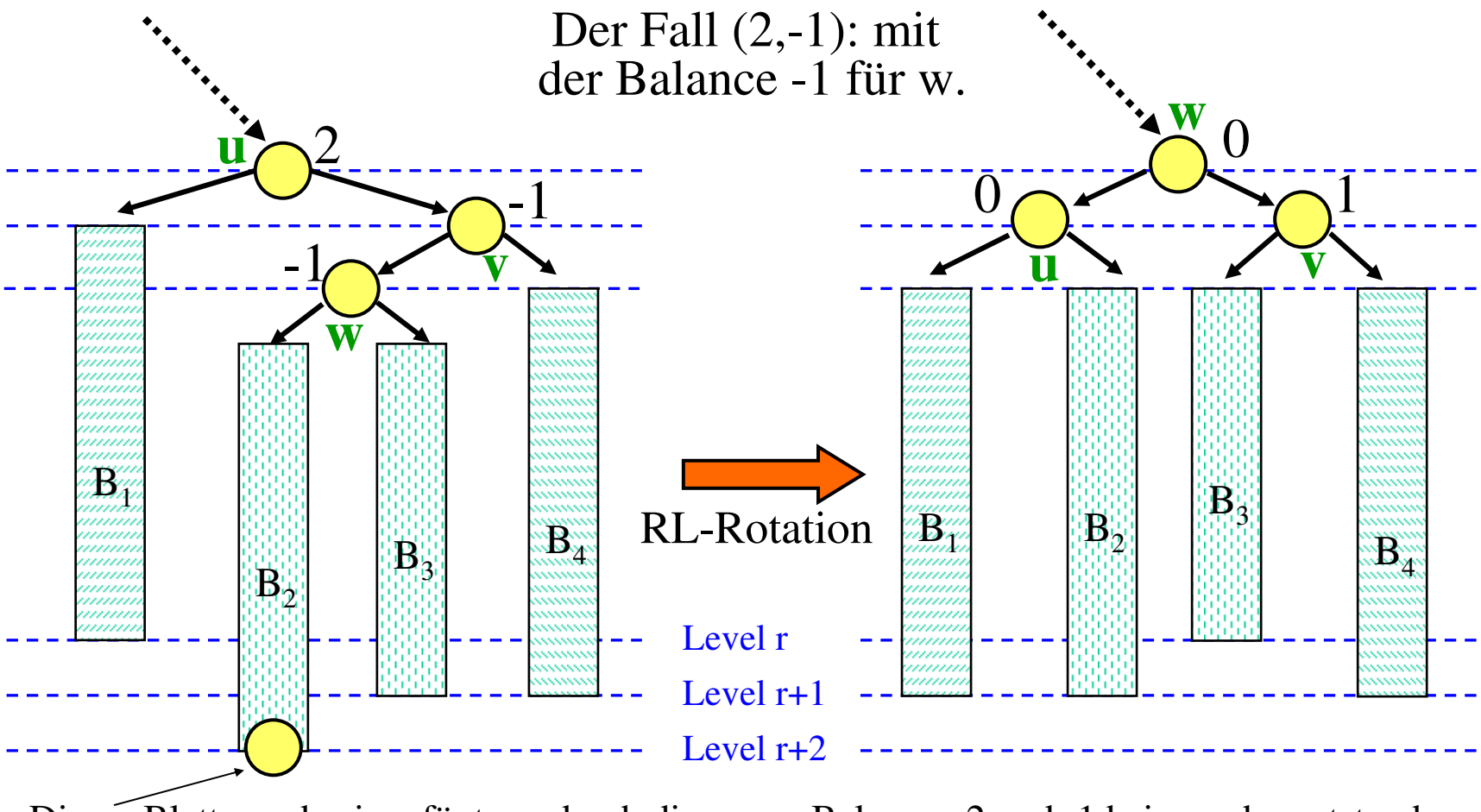
Situation an der untersten Verletzungsstelle **u** — **v**: Balancen -2 und -1.

⇒ Rechts-Rotation durchführen (3 Zeiger umhängen, Balancen 0 und 0).



Situation an der untersten Verletzungsstelle **u — v**: Balancen 2 und -1.

⇒ Rechts-Links-Rotation durchführen (4 Zeiger umhängen).



Situation an der untersten Verletzungsstelle **u — v**: Balancen 2 und -1.

$\Rightarrow$ ebenfalls Rechts-Links-Rotation durchführen (4 Zeiger umhängen).

Der Fall (-2,1) ist symmetrisch und wird mit einer Links-Rechts-Rotation (LR-Rotation) gelöst.

Zusammenfassung zum Einfügen eines Elements s:

- Füge ein neues Blatt mit Inhalt s und Höhenbalance 0 in den Suchbaum ein, siehe 8.2.10 b.
- Setze von diesem Blatt aufsteigend die Höhenbalancen (eventuell bis zur Wurzel hinauf) neu.
- Wird dabei einmal die neue Höhenbalance 0 eingetragen oder wurde die Balance der Wurzel bearbeitet, so ist man fertig.
- Wird dabei einmal der Wert 2 oder -2 angenommen, so führe man die zugehörige Rotation durch; ebenfalls fertig.

Beachten Sie:

(1) Bei jedem Einfügen muss *höchstens eine Rotation* durchgeführt werden, um die AVL-Eigenschaft wieder herzustellen. (Beim Löschen gilt dies nicht, siehe unten.)

(2) Die "Rotation" ist eine Drehung um den Mittelpunkt einer Kante. Logisch betrachtet findet dabei eine vertikale Verschiebung von Knoten statt (sonst würde die Suchbaum-Eigenschaft verletzt werden).

Genauere Nachfrage:
Wie wird die Höhenbalance B(...) neu berechnet? Dieser Algorithmus wird auf der folgenden Folie vorgestellt.

Füge ein Blatt **v** mit Inhalt s und Höhenbalance B(**v**)=0 ein;
u := direkter Vorgängerknoten von **v**; Abbruch:=false;

```
while u ≠ null and not Abbruch loop
   if v ist linkes Kind von u then
       if  B(u) = 1  then B(u) := 0; Abbruch:=true;
       elsif  B(u) = 0  then B(u) := -1;
       else  "führe R/LR-Rotation durch;" Abbruch:=true;  end if;
   else              -- v ist rechtes Kind von u
       if  B(u) = -1  then B(u) := 0; Abbruch:=true;
       elsif  B(u) = 0  then B(u) := 1;
       else  "führe L/RL-Rotation durch;" Abbruch:=true;  end if;
   end if;
   if not Abbruch then  v := u;
         u := direkter Vorgänger von u;  end if;
end loop;
```

Das Löschen in einem AVL-Baum erfolgt zunächst wie beim Suchbaum: Finde den Knoten x mit Inhalt s. Hat x weniger als zwei Nachfolger, so lösche x und setze den Zeiger, der vom direkten Vorgänger auf x zeigte, auf den eventuell vorhandenen Nachfolger von x.
Hat x zwei Nachfolger, so ersetze s durch den Inorder-Nachfolger s', lösche den Knoten y, in dem s' stand, und biege die Kante, die auf y zeigte, auf den rechten Nachfolger von y (eventuell ist dieser null) um.

Ausgehend vom Vorgänger von x bzw. y müssen (zur Wurzel hin aufsteigend) die Höhenbalancen neu gesetzt werden. Im Gegensatz zum Einfügen, können nun mehrere Rotationen notwendig sein, bis die Wurzel erreicht ist oder bis man aus einem anderen Grund abbrechen kann. Der Algorithmus zur Berechnung der Höhenbalancen lautet ab dem Löschen, beginnend mit dem direkten Vorgänger des gelöschten Knotens (im Programm heißt der gelöschte Knoten "v"):

u := direkter Vorgängerknoten von **v**; Abbruch:=false;
u.Links := **v**.Rechts; -- Knoten **v** wird hierdurch unerreichbar
while **u** ≠ null and not Abbruch loop
  if **v** ist linkes Kind von **u** then
    if B(**u**) = -1 then B(**u**):=0; **v**:=**u**; **u**:=direkter Vorgänger von **u**;
    elsif B(**u**) = 0 then B(**u**) := 1; Abbruch := true;
    else *führe in Abhängigkeit von* B(v)∈*{-1,0,1} eine*
    Rotation durch; **v**:=**u**; **u**:=direkter Vorgänger von **u**; end if;
  else -- **v** ist rechtes Kind von **u**
    if B(**u**) = 1 then B(**u**):=0; **v**:=**u**; **u**:=direkter Vorgänger von **u**;
    elsif B(**u**) = 0 then B(**u**) := -1; Abbruch := true;
    else *führe in Abhängigkeit von* B(v)∈*{-1,0,1} eine*
    Rotation durch; **v**:=**u**; **u**:=direkter Vorgänger von **u**; end if;
  end if;
end loop;

Aufgabe: Führen Sie die kursiven Teile des Algorithmus im Detail aus.

Zusammenfassung:

FIND: Wie beim Suchbaum.

INSERT: Wie beim Suchbaum, aber die Balancen längs des Einfügepfades aufsteigend korrigieren. Dabei ist höchstens eine Rotation erforderlich.

DELETE: Wie beim Suchbaum, aber aufsteigend vom Inorder-Nachfolger- (oder -Vorgänger-) Knoten entlang des Pfads zur Wurzel die Höhenbalancen (evtl. mittels Rotationen) ändern.

8.4.6 Ergebnis: Jede der drei Operationen erfordert auch im schlechtesten Fall höchstens O(log(n)) Schritte.

Der Grund hierfür: Ein AVL-Baum mit n Knoten kann höchstens die Tiefe $1{,}4404 \cdot \log(n)$ besitzen.

Diesen Satz beweisen wir im Folgenden.

Aufgabe: Man stelle fest, wie viele Knoten in einem AVL-Baum der Tiefe t maximal und minimal liegen können.

Maximal können es 2^t-1 Knoten sein (klar, 8.2.12).

Sei m_t die Anzahl der Knoten, die sich mindestens in einem AVL-Baum der Tiefe t befinden müssen, dann gilt: $m_0=0$, $m_1=1$ und für alle $t \geq 2$: $m_t = 1 + m_{t-1} + m_{t-2}$. (Wurzel plus linker Unterbaum und rechter Unterbaum, die Folge der Zahlen ist $0, 1, 2, 4, 7, 12, 20, \ldots$).

Die Lösung der Gleichung lautet: $m_t = F_{t+2} - 1$, wobei F_t die t-te Fibonacci-zahl ist. Dies ist durch Einsetzen in die Gleichung leicht nachzuweisen.

8.4.7 Die *Fibonaccizahlen* sind definiert durch:

$F_0=0$, $F_1=1$, $F_k = F_{k-1} + F_{k-2}$ für alle $k \geq 2$ (Folge: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$).

Man nehme an, F_k erfüllt eine Gleichung der Form $F_k = a \cdot x^k$, dann muss $x^k = x^{k-1} + x^{k-2}$ gelten, d.h., man muss die Gleichung $x^2 = x + 1$ lösen. Deren Lösungen sind c_1 und c_2 (siehe nächste Folie).

Da auch deren Linearkombinationen Lösungen darstellen, erhält man für die Fibonaccizahlen die Formel $F_k = a_1 \cdot c_1^k + a_2 \cdot c_2^k$. Mit den Anfangsbedingungen $F_0=0$ und $F_1=1$ ergibt sich $a_1 = -a_2 = f$, woraus man $F_k = f \cdot (c_1^k - c_2^k)$ erhält.

Wegen $|c_2| < 1$ ist F_k stets die nächste natürliche Zahl zu $f \cdot c_1^k$.

Einschub: Fibonaccizahlen

$F_0 = 0,\ F_1 = 1$ und $F_k = F_{k-1} + F_{k-2}$ für alle $k \geq 2$.

$$F_k = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right) =: f \cdot (c_1^k - c_2^k)$$

mit $c_1 = \left(\frac{1+\sqrt{5}}{2} \right) \approx 1.618035,$

$c_2 = \left(\frac{1-\sqrt{5}}{2} \right) \approx -0.618035,$ $f = \frac{1}{\sqrt{5}} \approx 0.447213$

und F_k = nächste natürliche Zahl zur Zahl $f \cdot c_1^k$.

Sei also ein AVL-Baum mit n Knoten der Tiefe t gegeben.

$$n \geq m_t = F_{t+2} - 1 \approx f \cdot c_1^{t+2} - 1$$
$$\approx 0.447213 \cdot 1.618035 \cdot 1.618035 \cdot 1.618035^t - 1$$
$$\approx 1.17082 \cdot 1.618035^t - 1$$

Es folgt mit $1/\log(1.618035) \approx 1.4404$:

$$t \leq \log((n+1)/1.17082) / \log(1.618035)$$
$$\approx 1.4404 \cdot \log(n+1) - \log(1.17082) / \log(1.618035)$$
$$\approx 1.4404 \cdot \log(n)$$ [Diese Abschätzung ist sehr genau.]

Satz 8.4.8: Ein AVL-Baum mit n Knoten besitzt höchstens die Tiefe $1.4404 \cdot \log(n)$.

Hinweis: Messungen ergaben, dass diese maximale Tiefe in der Praxis fast nie auftritt und sich die Tiefe der AVL-Bäume meist nur sehr wenig von log(n) unterscheidet. Aber: Der Fall der Tiefe $\approx 1.4404 \cdot \log(n)$ kann auftreten, siehe im Folgenden.

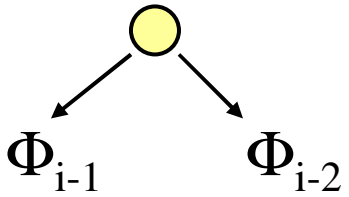
8.4.9 Fibonacci-Bäume

Φ_0 ist der leere Baum, Φ_1 ist der einknotige Baum.

Definition:

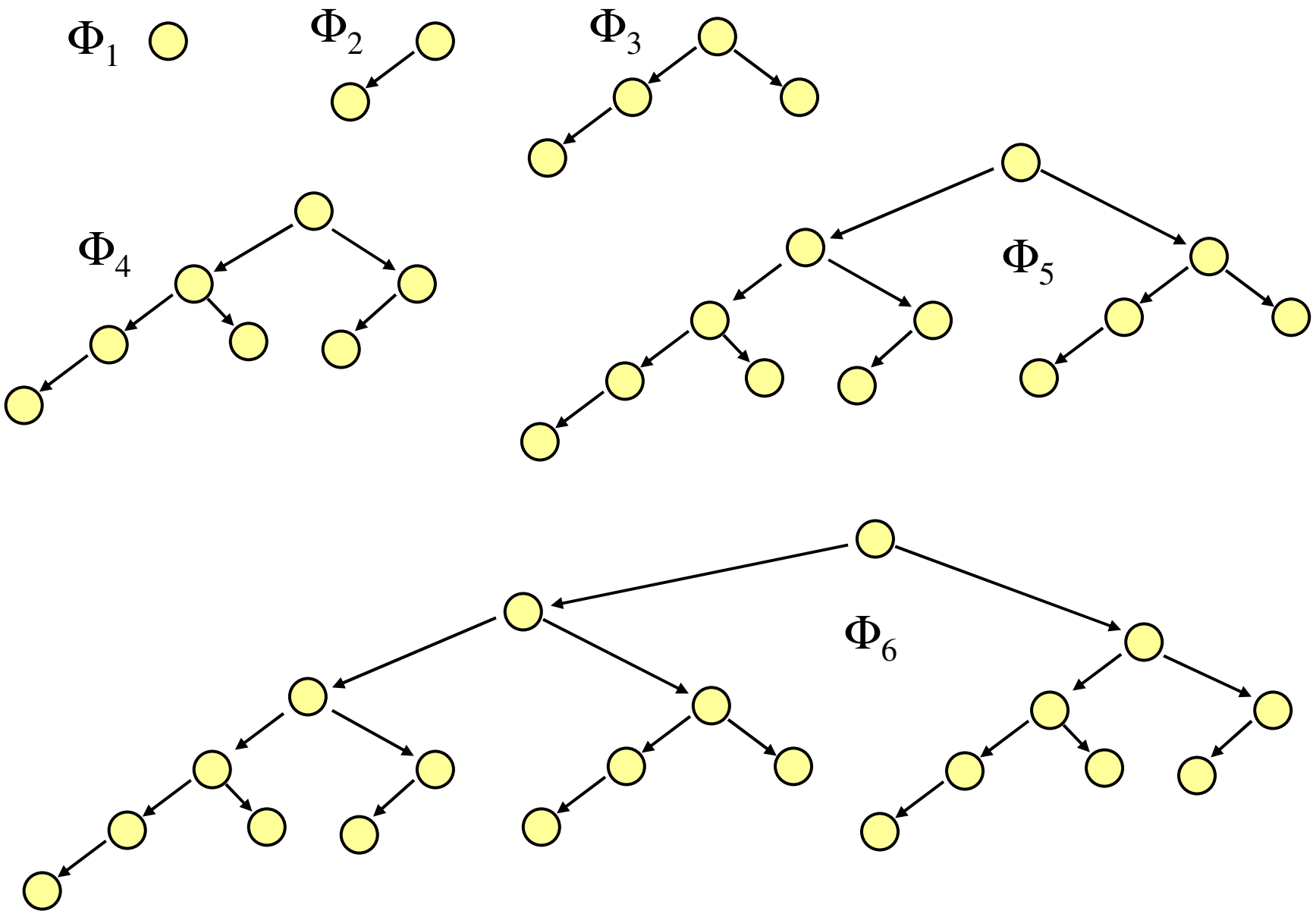
(1) Φ_0 und Φ_1 sind die Fibonaccibäume der Ordnung 0 und 1.

(2) Wenn Φ_{i-1} der Fibonaccibaum der Ordnung i-1 und Φ_{i-2} der Fibonaccibaum der Ordnung i-2 sind ($i \geq 2$), dann ist



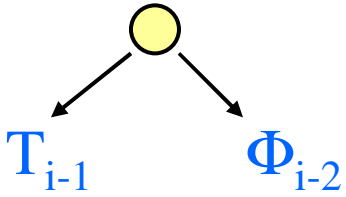
der Fibonaccibaum Φ_i der Ordnung i.

Der Fibonaccibaum der Ordnung i ist der "dünnste" AVL-Baum der Tiefe i, also der AVL-Baum mit minimal vielen Knoten zu gegebener Tiefe i (hieraus folgt auch 8.4.6). Der Baum Φ_i besitzt genau $m_i = F_{i+2} - 1$ Knoten.


Φ_1
Φ_2
Φ_3
Φ_4
Φ_5
Φ_6

8.4.10 Ist jeder AVL-Baum zugleich gewichtsbalanciert?

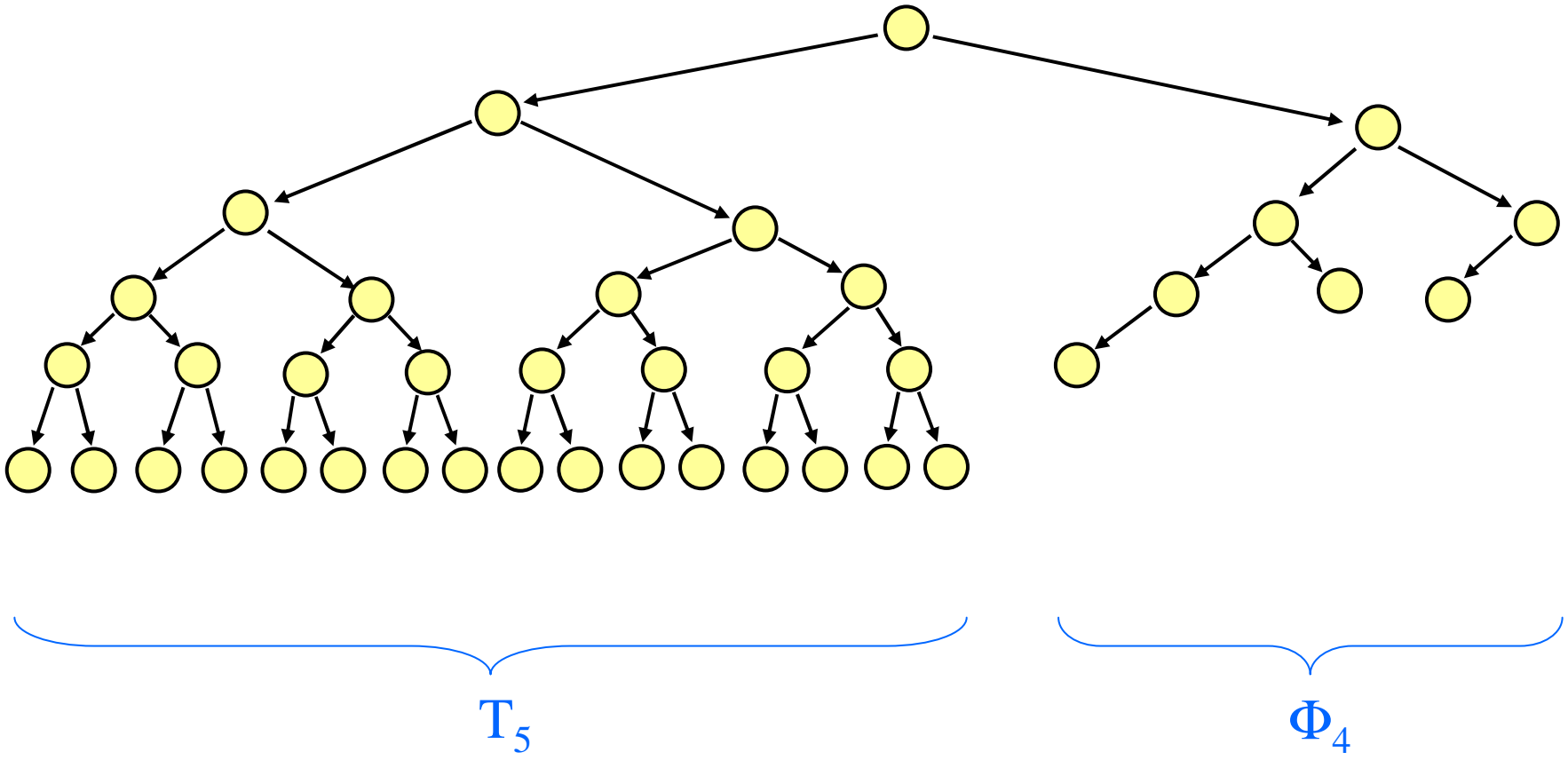
Nein! Betrachte hierzu folgenden AVL-Baum der Tiefe i:



wobei T_{i-1} der AVL-Baum der Tiefe i-1 mit maximal vielen Knoten ist. T_{i-1} besitzt $2^{i-1}-1$ und Φ_{i-2} besitzt $m_{i-2} = F_i - 1$ Knoten. (Ein Beispiel steht auf der nächsten Folie.)

Der Quotient $2^{i-1}/F_i$ wächst exponentiell in i. Daraus folgt, dass es für jede vorgegebene reelle Zahl $\alpha \in (0, ½]$ AVL-Bäume gibt, die *nicht* α-gewichtsbalanciert (siehe Definition 8.4.1) sind.

Der folgende Baum ist ein AVL-Baum. Man erkennt, dass seine Tiefe durch log(n)+2 beschränkt ist; seine 39 Knoten ($i = 6$ und $n = 39 = 1 + 2^5\text{-}1 + 8\text{-}1$) sind jedoch sehr unterschiedlich auf die beiden Unterbäume der Wurzel verteilt.



Abschließender Hinweis: Es werden häufiger Bäume, die sehr gleichverzweigt sind und nur Blätter auf dem Level log(n) oder eins weniger besitzen, verwendet. Ihnen wollen wir einen Namen geben, nämlich "ausgeglichene" Bäume, vgl. Anfang von Abschnitt 8.3:

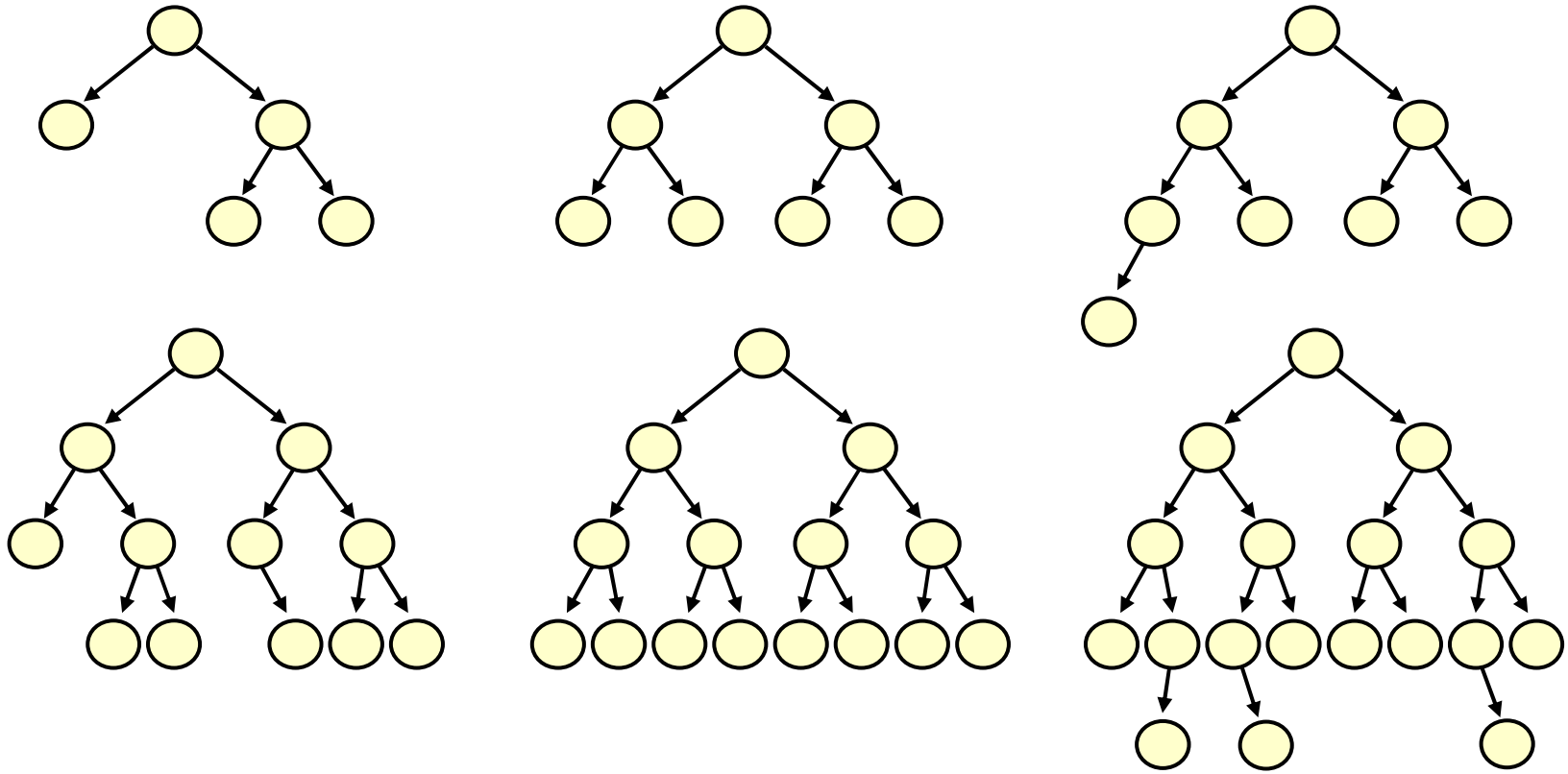
Definition 8.4.11:

Ein nicht-leerer, k-närer Baum (siehe Definition 8.2.3) heißt ausgeglichener Baum (der Tiefe r), wenn es eine natürliche Zahl r gibt, so dass jeder Knoten, der mindestens einen null-Zeiger enthält, das Level r-1 oder r besitzt und wenn es mindestens ein Blatt der Tiefe r gibt.

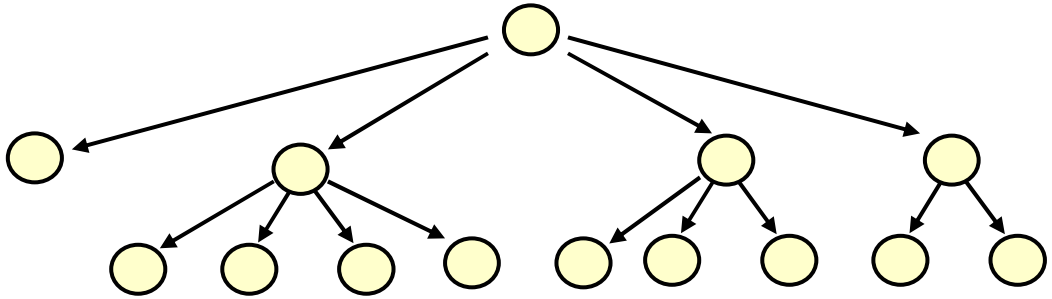
Man kann diese Definition leicht von k-nären auf beliebige geordnete Bäume übertragen. Die B-Bäume des nächsten Abschnitts 8.5 bilden solch ein Beispiel.

Sofern es Knoten mit null-Zeigern in verschiedenen Leveln gibt, ist in einem ausgeglichenen Bäumen das Level r-1 komplett mit Knoten besetzt. Überschüssige Knoten können sich dann nur noch auf dem nächsten Level r befinden.

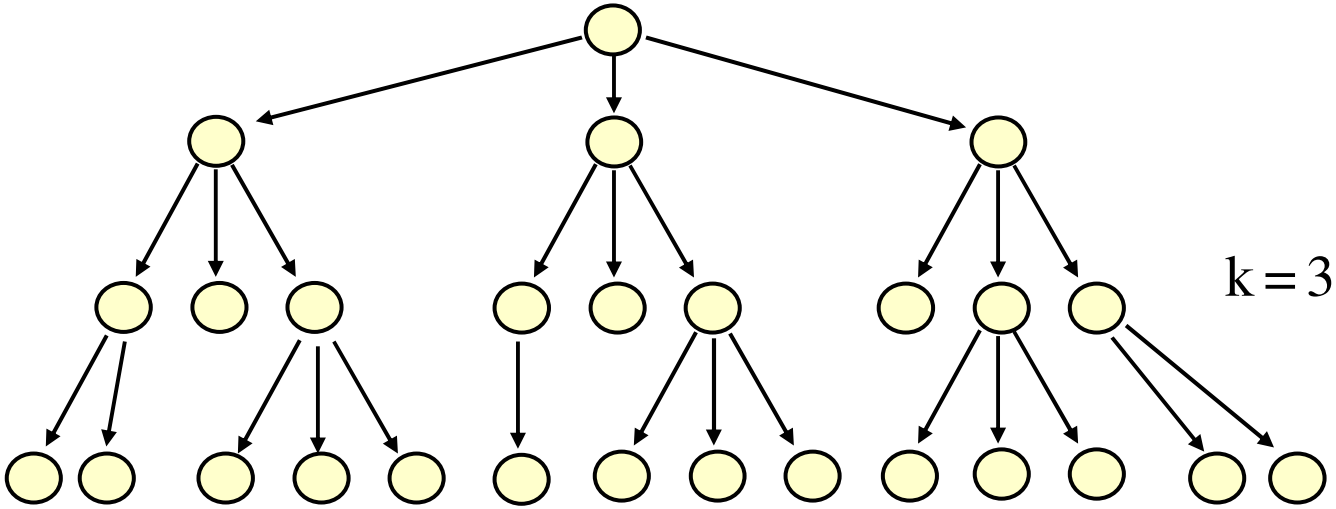
Beispiele für $k=2$ (also binäre Bäume) und für $r=3,4$ und 5:



Beispiele für k > 2 (null-Zeiger wurden nicht eingetragen, wodurch zu jeder Skizze mehrere k-näre Bäume gehören können):



$k = 4, r = 3$



$k = 3, r = 4$

8.5 B-Bäume (für externe Speicherung)

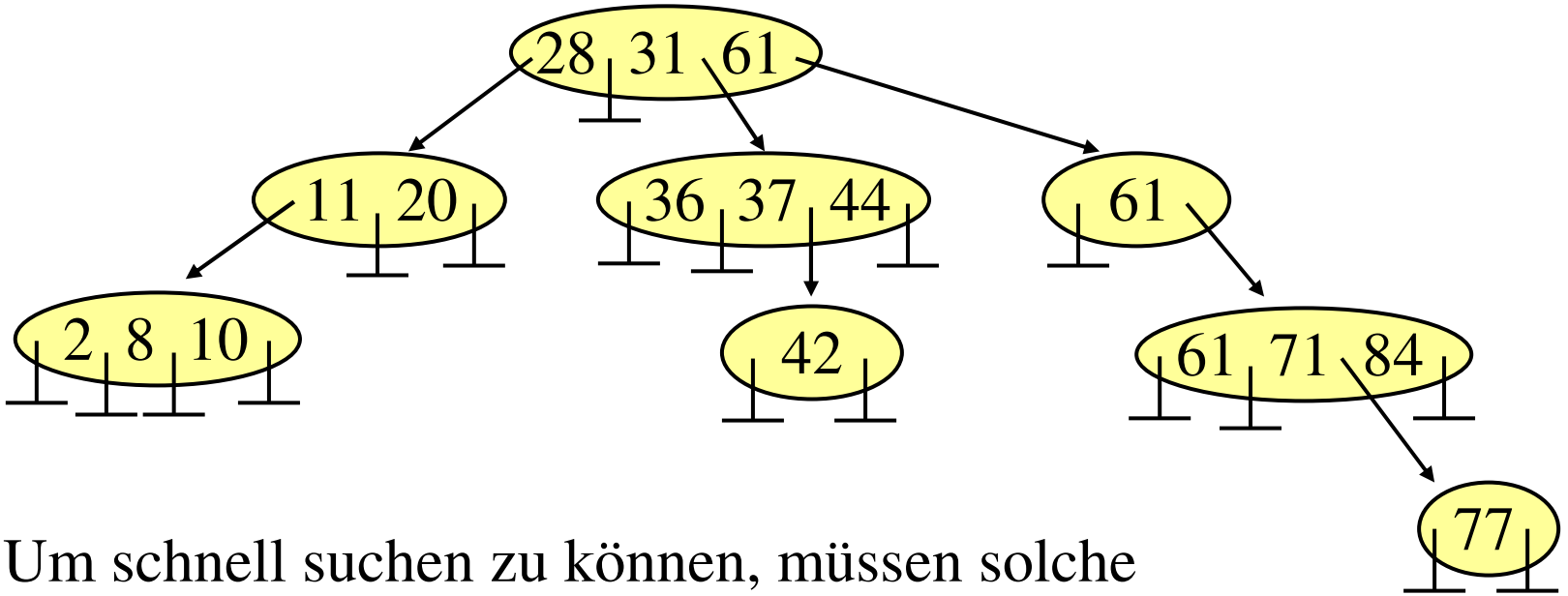
AVL-Bäume eignen sich gut als Darstellung von Mengen, wenn sich die gesamte Menge im Hauptspeicher befindet. Liegen die Elemente dagegen auf einem Hintergrundspeicher, so muss man bei jedem Übergang zu einem Kindknoten auf diesen externen Speicher zugreifen. Heute ist der Hintergrundspeicher meist eine Festplatte mit einer Zugriffszeit von im Mittel 5 Millisekunden. Hat man einen AVL-Baum der Tiefe 20 (dies entspricht einer Menge mit 1 Million Elementen), so werden alleine 100 Millisekunden für den Zugriff benötigt, während die durchgeführten 20 Vergleiche weniger als eine Millisekunde erfordern. Der Rechner verbringt seine Zeit daher zu über 99% mit Warten und ist nach rund 101 Millisekunden mit der Suche fertig.

Ziel ist daher eine Datenstruktur, die mit möglichst wenigen externen Zugriffen die gesuchten Daten findet bzw. einfügt bzw. löscht.

Nahe liegend ist eine gute Mischung aus baumartiger und sequenzieller Suche: Man holt bei jedem Zugriff - sagen wir - 500 Werte in den Hauptspeicher und grenzt den Bereich, wo der gesuchte Schlüssel zu finden ist, nicht wie beim AVL-Baum um den Faktor 2, sondern um den Faktor 500 ein.

Bei einem Datenbestand von 1 Million Werten sind dann nur noch drei Zugriffe erforderlich. Zusammen mit dem (internen) Durchlaufen der 500 Werte braucht man jetzt nur noch eine Gesamtlaufzeit von unter 20 Millisekunden, wobei 15 Millisekunden aus Warten bestehen.

Idee: Wir betrachten nun also Bäume, in deren Knoten sich nicht nur *ein* Schlüssel, sondern ein sortiertes k-Tupel von Schlüsseln befindet. Um effizient suchen zu können, durchläuft man dieses k-Tupel und folgt je nachdem, zwischen welchen Elementen der gesuchte Schlüssel liegt, einem Zeiger in den nächsten Unterbaum. Beispiel:



Um schnell suchen zu können, müssen solche Bäume folgende Suchbaumeigenschaft erfüllen.

Definition 8.5.1: Allgemeine Suchbaumeigenschaft

Gegeben sei ein geordneter Baum. In jedem Knoten steht ein nicht-leeres sortiertes Tupel von Schlüsseln einer geordneten Menge. Für alle Knoten u muss gelten:
Sind $s_1, s_2, ..., s_k$ die sortierten Schlüssel von u ($s_1 \leq s_2 \leq ... \leq s_k$), so besitzt u genau k+1 Kinder und es gilt:
Alle Schlüssel im Unterbaum des ersten Kindes sind kleiner als s_1, alle Schlüssel im Unterbaum des zweiten Kindes sind größer oder gleich s_1 und kleiner als s_2, alle Schlüssel im Unterbaum des i-ten Kindes sind größer oder gleich s_{i-1} und kleiner als s_i (für $i = 2, 3, ..., k$), und alle Schlüssel im Unterbaum des (k+1)-ten Kindes sind größer oder gleich s_k.
Ein geordneter Baum mit dieser Eigenschaft heißt (allgemeiner) Suchbaum.

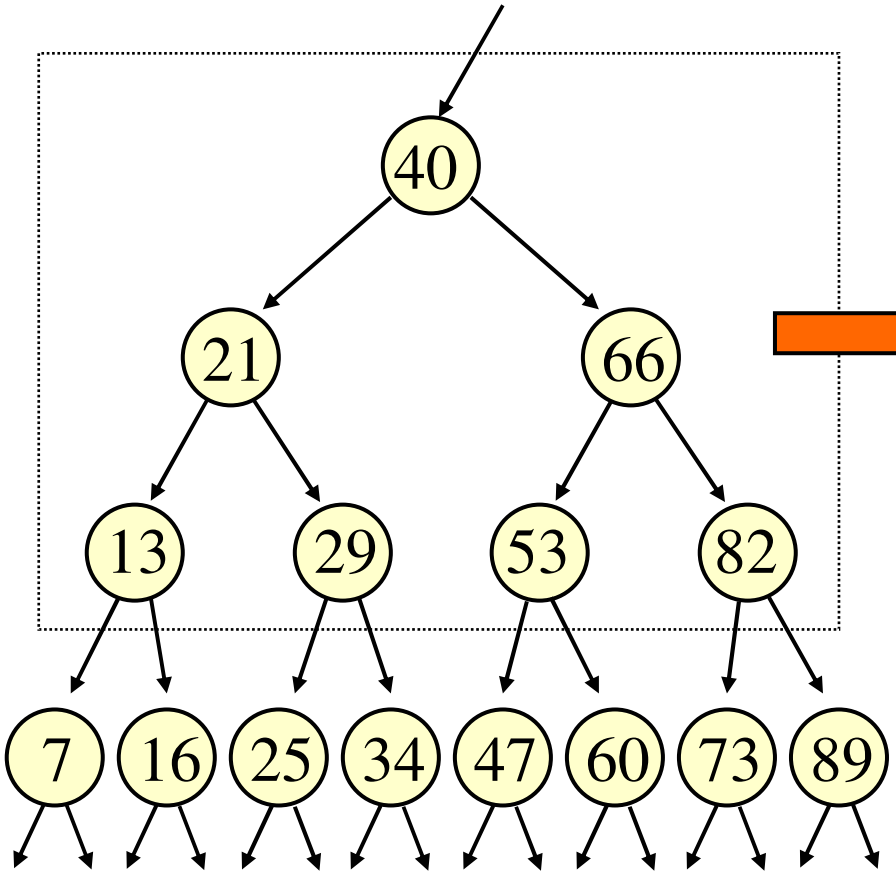
Hinweis: Wenn gleiche Schlüssel zugelassen sind, so wird man diese wie üblich in dem rechten Unterbaum zu einem Schlüssel ablegen (siehe Zahl 61 in der Abbildung auf der vorletzten Folie). Bei den nun zu definierenden B-Bäumen kann man diese Eigenschaft jedoch nicht garantieren, vielmehr können durch Einfüge- oder Löschoperationen gleiche Schlüssel auch in den linken Unterbaum verschoben werden. Wir werden diesen Fall daher verbieten und ihn am Ende des Abschnitts gesondert betrachten.
In Anwendungsfällen informiere man sich zuvor genau, ob dort gleiche Schlüssel zugelassen sind und wie sie behandelt werden.

Definition 8.5.2:

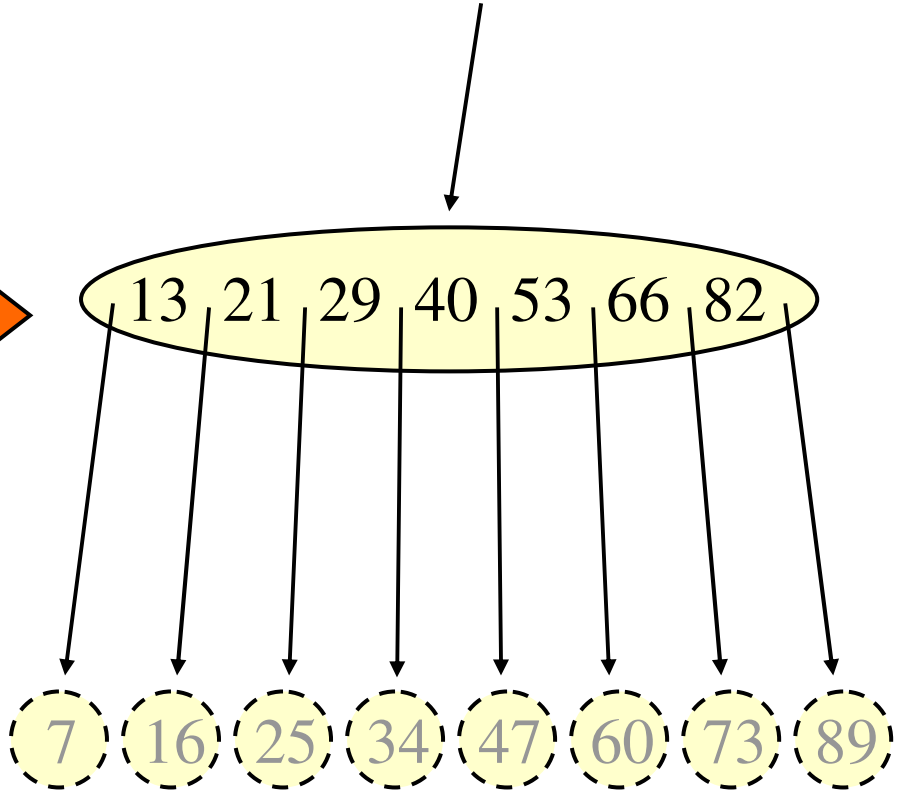
Es sei $m \geq 1$ eine natürliche Zahl. Ein geordneter nicht-leerer Baum, in dem jeder Knoten eine sortierte Folge von Schlüsseln aus einer geordneten Menge enthält, heißt **B-Baum** der Ordnung m, wenn für alle Knoten gilt:

1. Jeder Knoten enthält höchstens 2m Schlüssel.
2. Die Wurzel enthält mindestens einen Schlüssel.
3. Jeder andere Knoten enthält mindestens m Schlüssel.
4. Ein Knoten mit k Schlüsseln besitzt entweder genau k+1 Kinder oder kein Kind.
5. Alle Blätter (= Knoten ohne Kinder) besitzen das gleiche Level.
6. B erfüllt die allgemeine Suchbaumeigenschaft.

***Hinweis:* Wir werden in diesem Abschnitt 8.5 nur B-Bäume mit Inhalten behandeln, bei denen alle Schlüssel verschieden sind!**



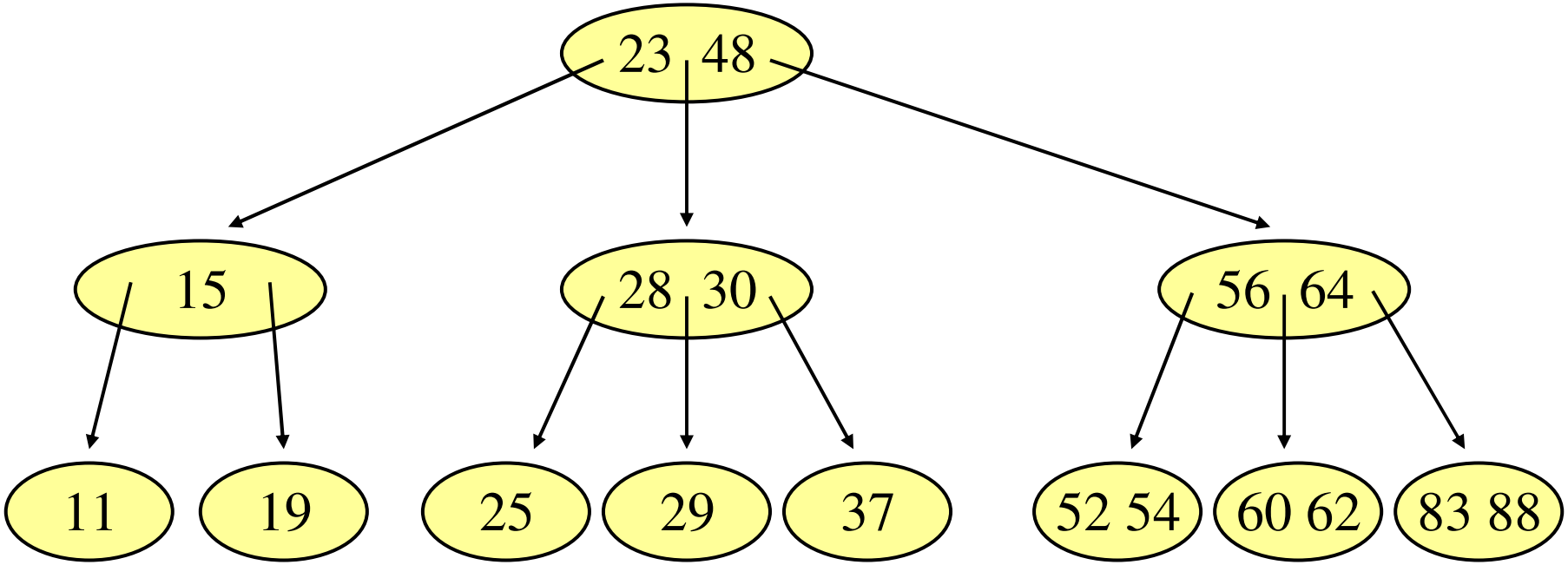
Ausschnitt aus einem binären Suchbaum.



Zusammenfassung zu einem B-Baum-Knoten, dessen Inhalt linear (oder mit Intervall-schachtelung) durchlaufen wird.

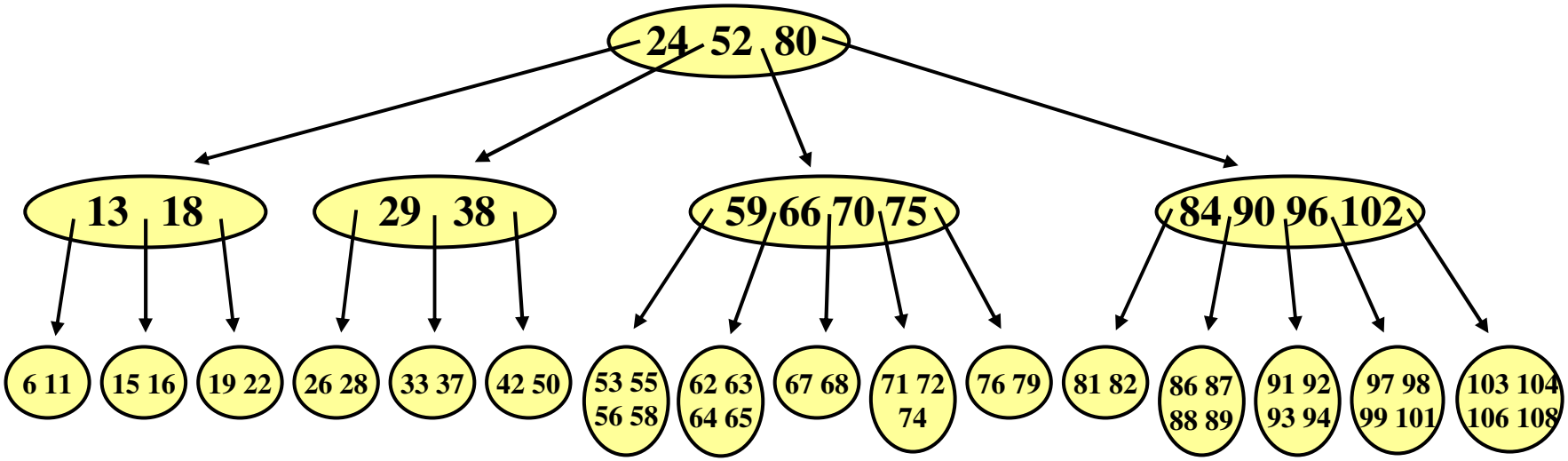
Beispiel für einen B-Baum der Ordnung 1

Man beachte, dass alle Blätter das gleiche Level besitzen.



Hinweis: B-Bäume der Ordnung 1 heißen auch 2-3-Bäume.

Beispiel für einen B-Baum der Ordnung 2:



Übliche Ordnungen liegen in der Praxis zwischen 128 und 4096.
(Begründen Sie diese Größenordnung!)

Tiefe eines B-Baums:
Da jeder Knoten (außer der Wurzel) mindestens m+1 Kinder besitzt, muss die Tiefe bei n Schlüsseln kleiner als $\log_{m+1}(n)+1$ sein. Andererseits kann sie nicht weniger als $\log_{2m+1}(n)$ betragen.

Hinweis: Die Terminologie ist in der Literatur unterschiedlich. Oft verwendet man auch "2m" oder "2m+1" als die Ordnung des B-Baums. Vergewissern Sie sich daher bei B-Bäumen stets über die zugrunde liegenden Definitionen.

Die Operationen auf B-Bäumen verändern die Struktur durch zwei Maßnahmen: Aufspalten (Splitten) eines Knotens in zwei Knoten und Verschmelzen zweier Knoten zu einem Knoten.

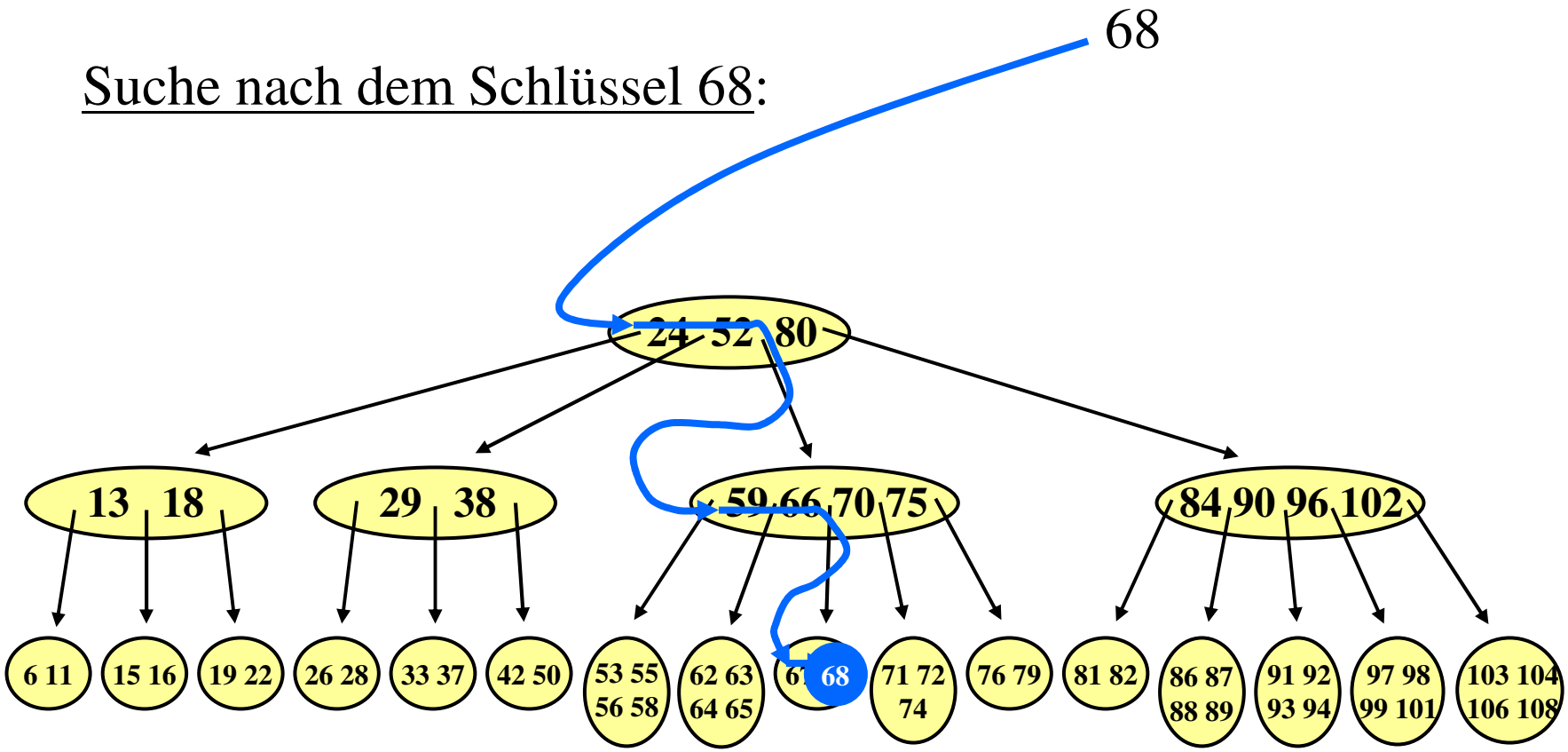
In vielen Anwendungen speichert man die Daten zwar im B-Baum, möchte die eigentlichen Inhalte beim Löschen und bei manchen Implementierungen auch beim Einfügen aber nicht im Baum verschieben. Dann legt man alle Daten in die Blätter des Baums und errichtet hierüber einen Baum aus Zeigern. Da ein B-Baum nur an der Wurzel wächst und schrumpft (siehe später), wird eine solche Struktur bei B-Bäumen automatisch aufrecht erhalten.
Ein B-Baum, bei dem alle Daten nur in den Blättern liegen, nennt man B*-Baum.
(Solche Bäume können auch Effizienzvorteile haben, siehe Analyse von Search2 in 6.5.1.)

Wir betrachten nun die drei Operationen Suchen, Einfügen und Löschen.

8.5.3 Suchen in einem B-Baum

Das Suchen erfolgt wie oben angegeben: Man durchlaufe das sortierte Tupel des Knotens und folge dem "richtigen" Zeiger entsprechend der Suchbaumeigenschaft. Der Zeitaufwand ist proportional zur Tiefe des Baums. Man muss jedoch noch die sortierten Listen der Schlüssel in jedem betrachteten Knoten durchlaufen. Dies kostet mindestens m und höchstens 2m Vergleiche (durch Intervallschachtelung kommt man auch mit log(2m) Vergleichen aus, wenn die Schlüssel in einem array abgelegt sind). Insgesamt muss man daher mit bis zu $2m \cdot \log_{m+1}(n)$ Vergleichen rechnen, je nach Implementierung weniger, vgl. nächste Folie.

Suche nach dem Schlüssel 68:

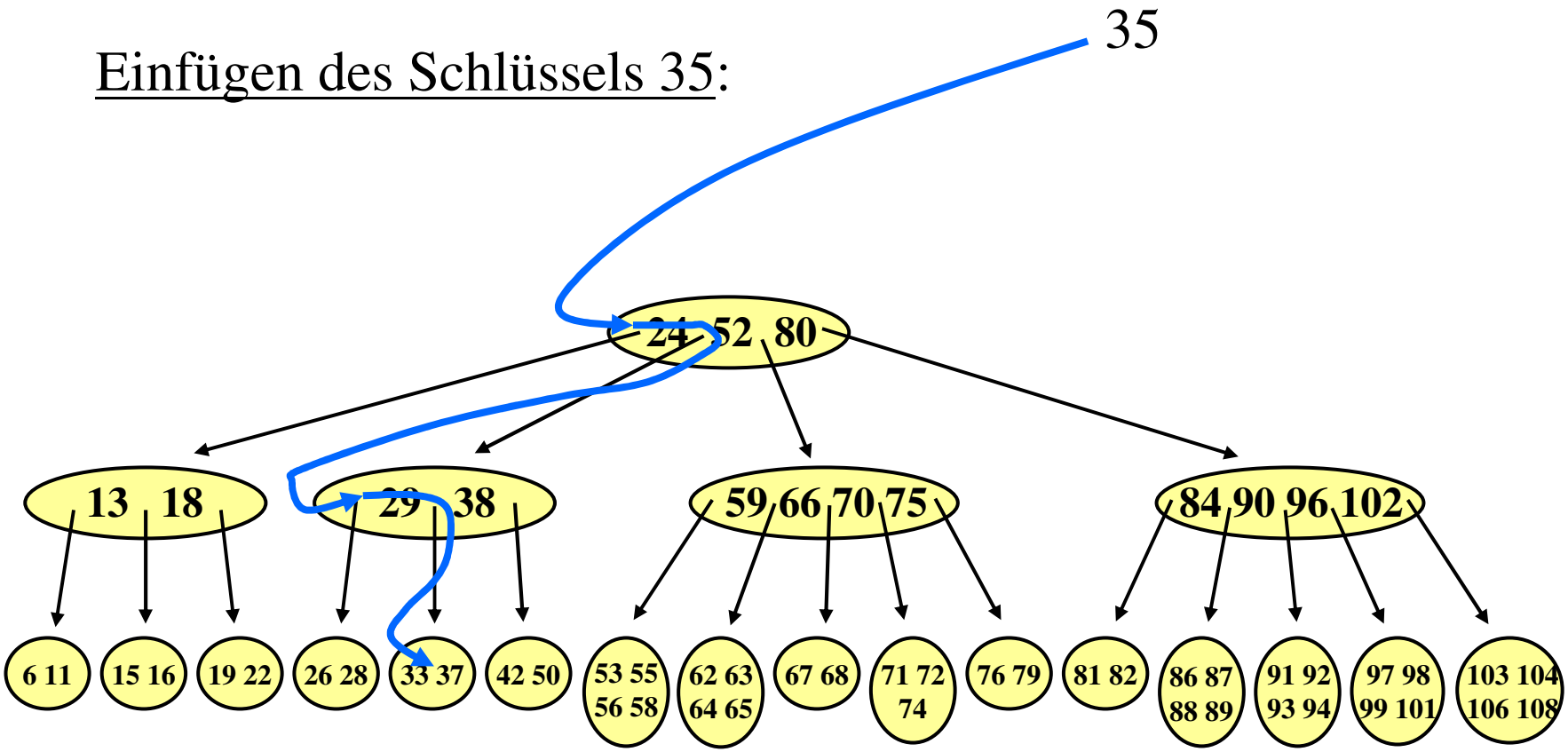


Für n Elemente im Baum gilt offenbar:
Zahl der Vergleiche ≤ 2m · Tiefe des Baumes ≤ $2m \cdot \log_{m+1}(n)$.
Für m = 512 ist dies beispielsweise für eine Wertemenge bis 134 Millionen Elemente durch 6m = 3072 Vergleiche und 3 Zugriffe auf den externen Speicher beschränkt. (Durch Intervallschachtelung bei der Suche in der Schlüsselmenge jedes Knotens reduziert sich die Zahl der Vergleiche auf $3 \cdot \log(2m) = 27$.)

8.5.4 Einfügen in einen B-Baum

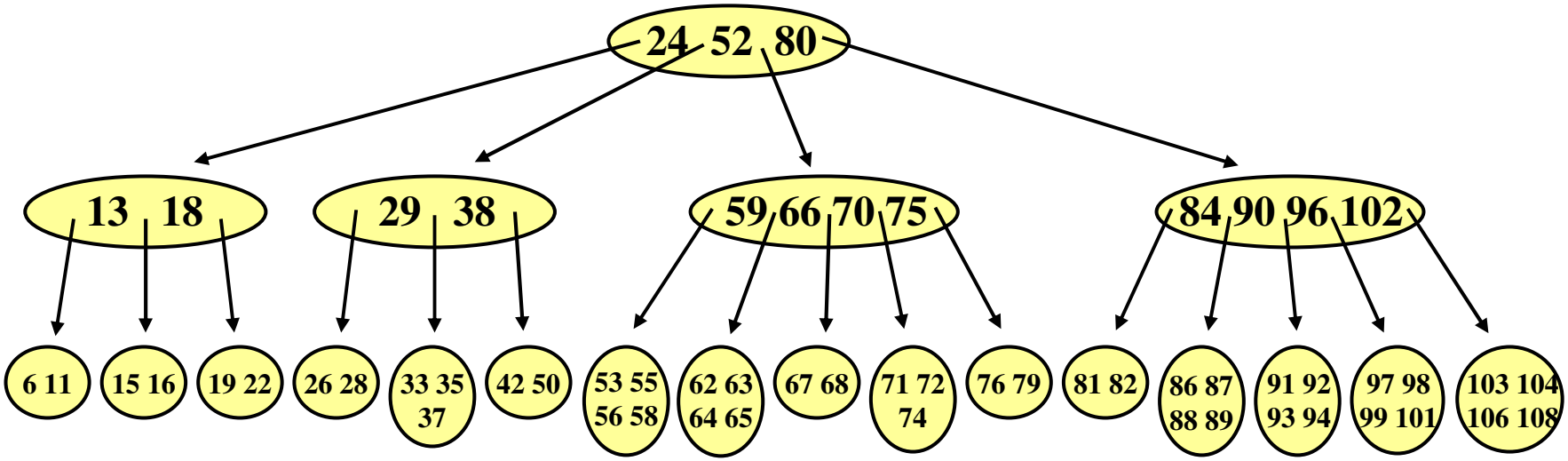
Beim Einfügen ermittelt man zunächst das Blatt, in das der neue Schlüssel s einzutragen ist. Befinden sich weniger als 2m Schlüssel in diesem Blatt, so füge man den neuen Schlüssel s sortiert ein. Anderenfalls liegt ein Überlauf im Knoten vor und das Blatt muss in zwei Blätter mit je m Schlüsseln aufgespalten werden, wobei der mittelste der 2m+1 Schlüssel an den Elternknoten weitergereicht und dort eingetragen wird. Eventuell muss nun auch der Elternknoten aufgespalten werden usw. Hierbei kann von unten nach oben ein Aufspalten bis zur Wurzel hin erfolgen.
Wird die Wurzel aufgespalten, so wird eine neue Wurzel mit genau einem Schlüssel erzeugt.
Das Vorgehen wird in den folgenden Beispielen illustriert.
Der Aufwand ist wie bei der Suche, jedoch muss ggf. der Pfad bis zur Wurzel zurückverfolgt werden. Hinzu kommt das Einsortieren der Schlüssel in die Knoten.

Einfügen des Schlüssels 35:

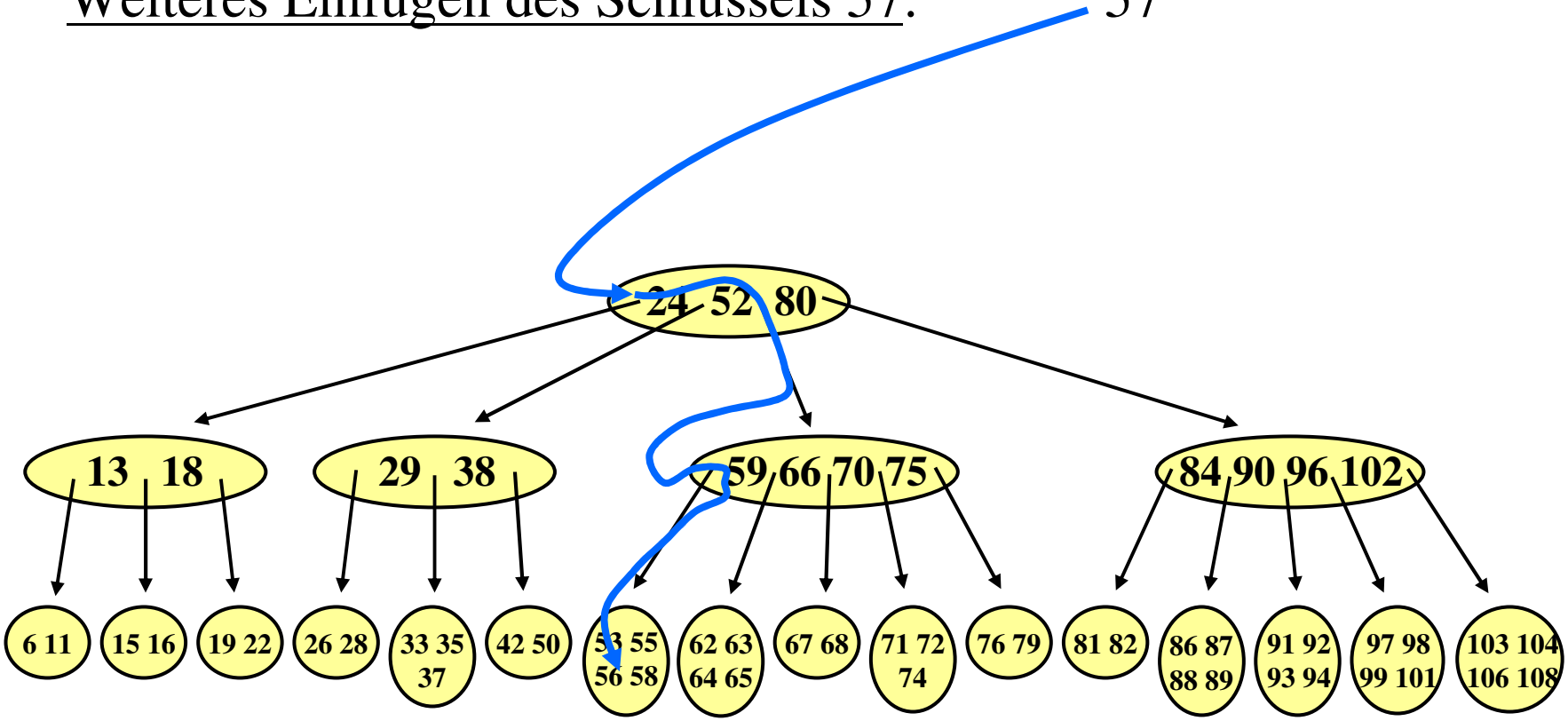


Es ist noch Platz, also wird der Schlüssel 35 sortiert hier eingetragen.

Schlüssels 35 ist nun eingefügt:

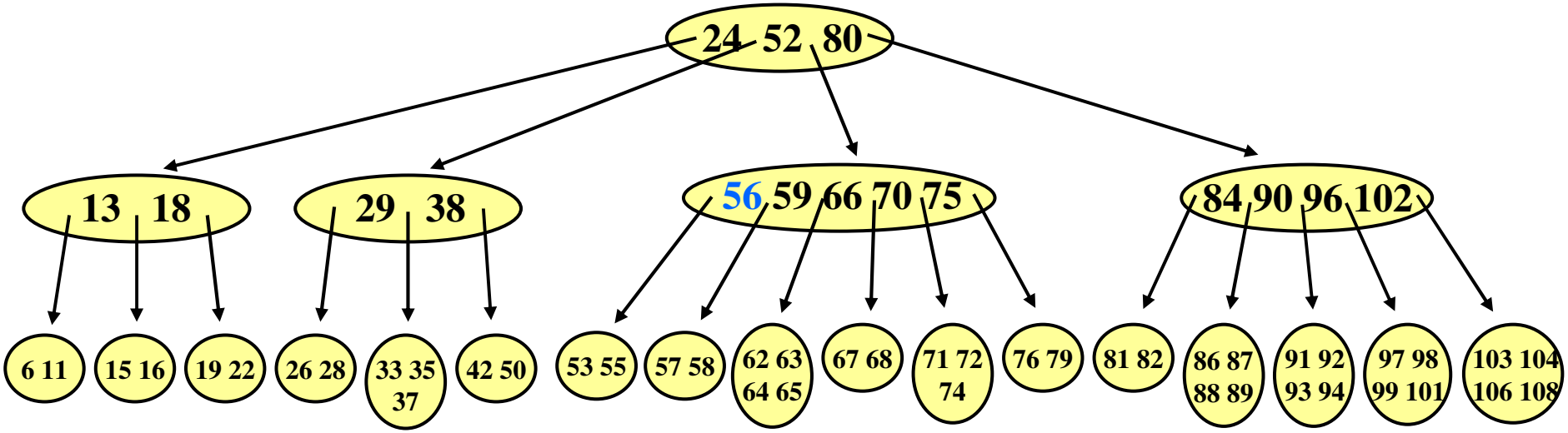


Weiteres Einfügen des Schlüssels 57:



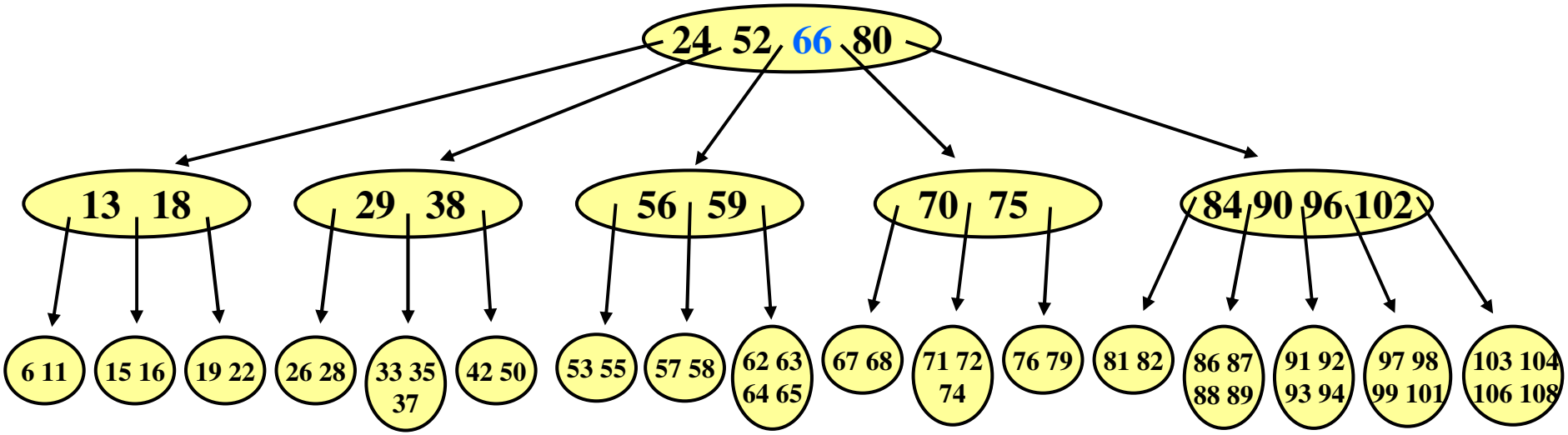
Beim Einfügen der 57 läuft der Knoten über, da er jetzt fünf Schlüssel enthält. Also wird er in zwei Knoten aufgespalten mit den Inhalten "53 55" bzw. "57 58". Der mittlere Wert "56" wird zum Elternknoten hinauf gereicht.

Weiteres Einfügen des Schlüssels 57:



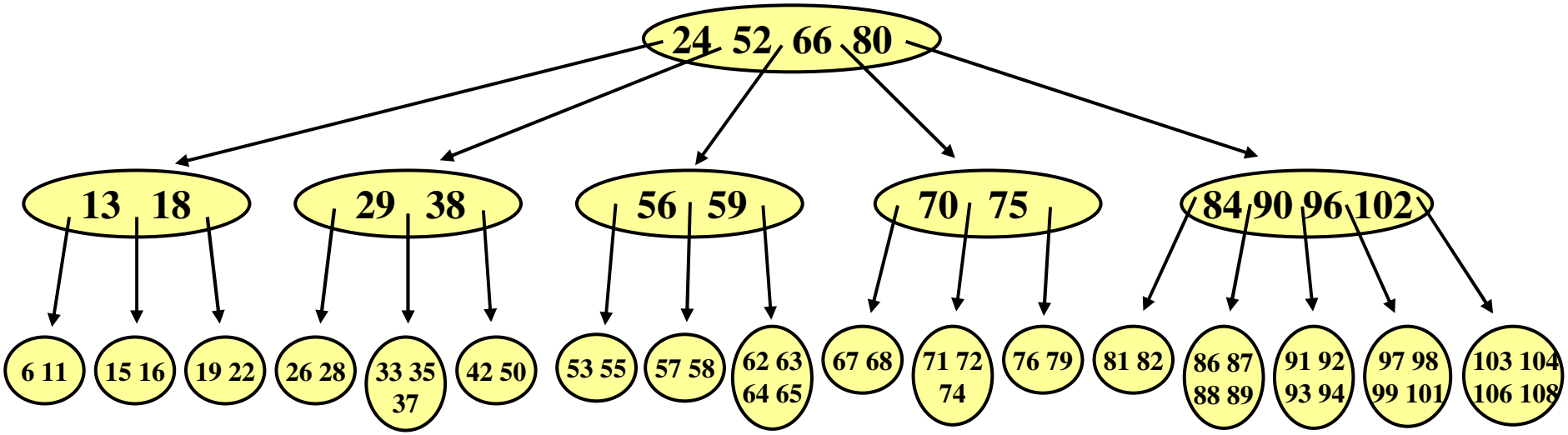
Beim Einfügen der 56 läuft aber nun der Elternknoten über. Also wird er in zwei Knoten aufgespalten mit den Inhalten "56 59" bzw. "70 75". Der mittlere Wert "66" wird zu seinem Elternknoten (dies ist hier die Wurzel) hinauf gereicht.

Ergebnis des Einfügens von Schlüssel 57:



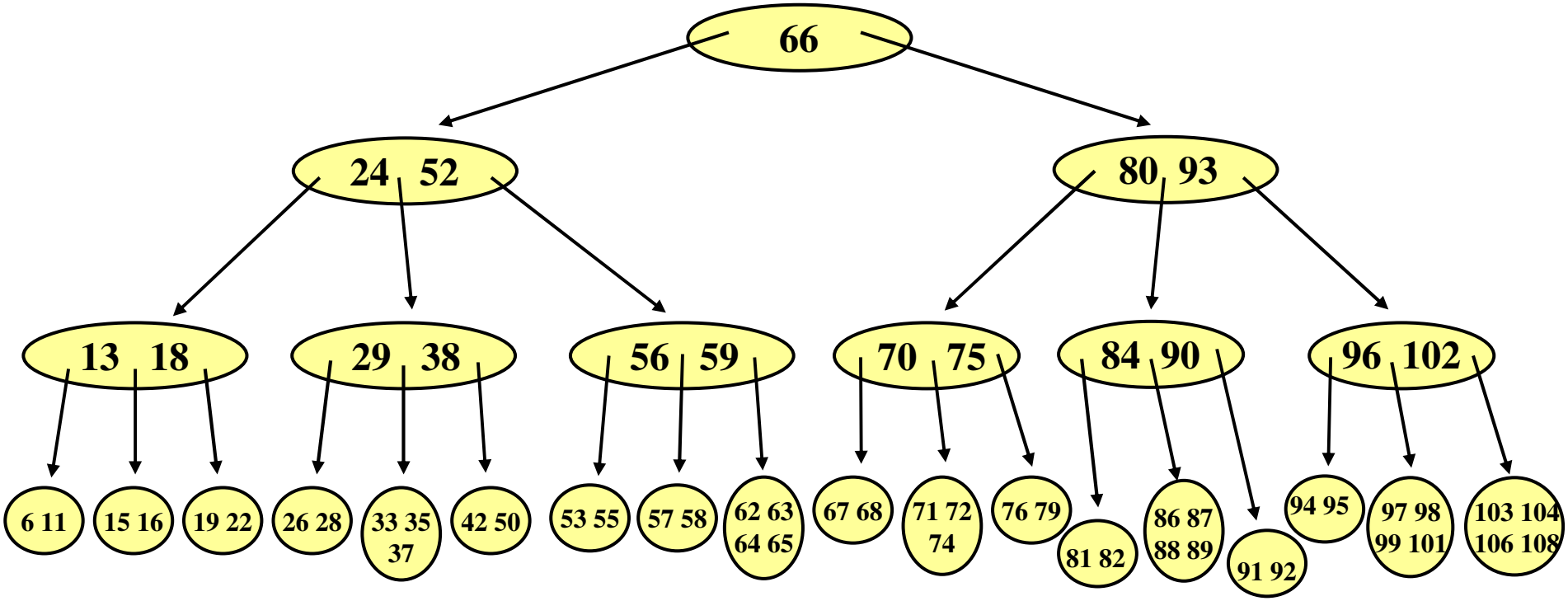
Die Wurzel darf (wie jeder Knoten in einem B-Baum der Ordnung 2) bis zu 4 Schlüssel besitzen, folglich ist dieser Baum das Ergebnis, wenn der Schlüssel 57 eingefügt wurde.

Weiteres Einfügen des Schlüssels 95:



Die 95 muss in den mit "91 92 93 94" beschrifteten Knoten eingetragen werden. Dieser läuft über und wird in zwei Knoten mit den Inhalten "91 92" und "94 95" aufgespalten; der Schlüssel "93" wird nach oben gereicht. Der Elternknoten wird auch aufgepalten und reicht den Schlüssel "93" weiter nach oben. Auch die Wurzel läuft nun über, wird gespalten und reicht den Schlüssel 66 weiter. Dieser wird neue Wurzel des Baumes. Man erhält also folgenden B-Baum der Ordnung 2 (mit einer um 1 größeren Tiefe):

Ergebnis nach Einfügen der Schlüssel 35, 57 und 95:



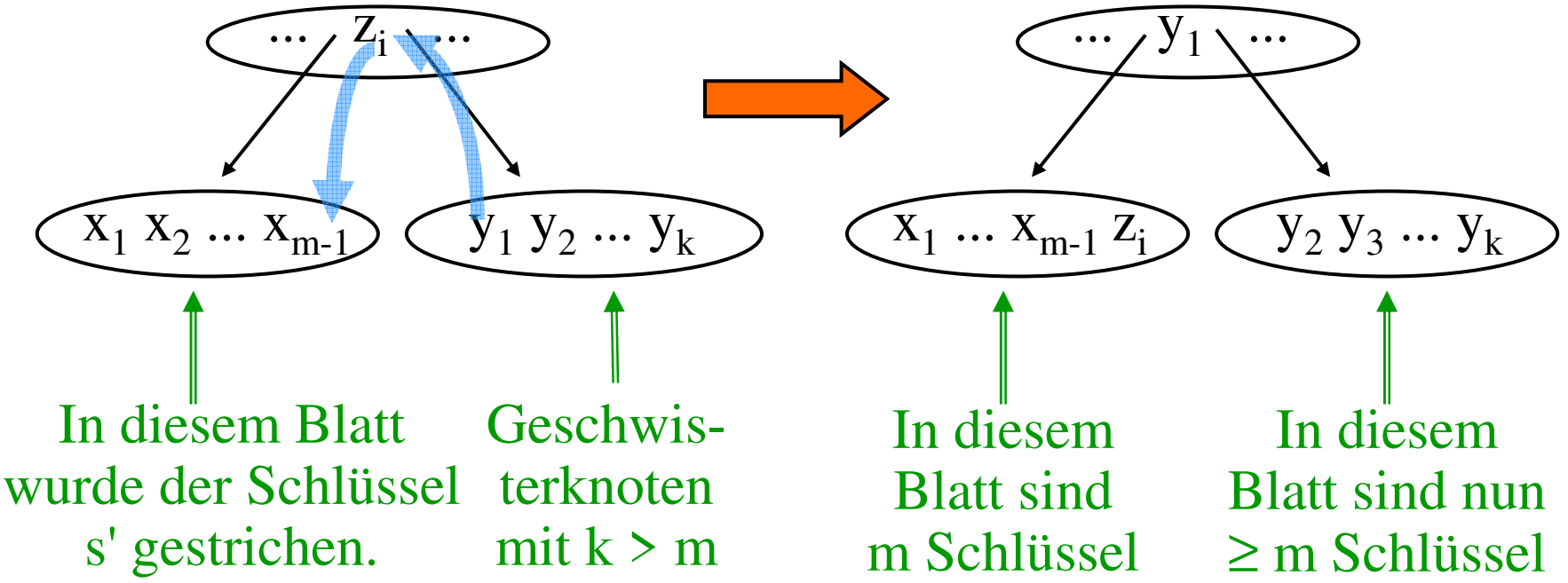
Man beachte, dass die Blätter hierbei stets auf dem gleichen Level liegen. Ein B-Baum kann nur an der Wurzel wachsen (und schrumpfen), während AVL-Bäume an den Blättern wachsen.

8.5.5 Löschen in einen B-Baum

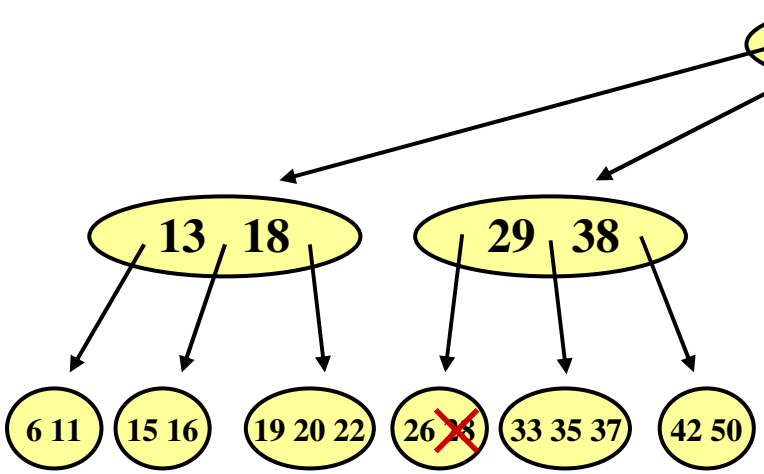
Beim Löschen wird zunächst der Knoten, in dem der gesuchte Schlüssel s steht, ermittelt. Ist dieser Knoten kein Blatt, so wird der Schlüssel s durch seinen Inorder-Nachfolger s' ersetzt (Erinnerung: wie findet man diesen? Siehe 8.2.10 c). Da der Inorder-Nachfolger in einem Blatt steht, muss also auf jeden Fall ein Schlüssel s' (oder s) im Blatt gelöscht werden.

Fall 1: Besitzt das Blatt mindestens m+1 Schlüssel, so wird der Schlüssel s' gelöscht und man ist fertig.
Fall 2: Besitzt das Blatt genau m Schlüssel (es liegt hier ein "Unterlauf" vor), so prüft man, ob ein Geschwisterknoten mit mindestens m+1 Schlüsseln existiert. Ist dies der Fall, so führt man einen Ausgleich unter Verwendung des zugehörigen Schlüssels des Elternknotens durch. Hierbei genügt eine Verschiebung von 2 Schlüsseln.

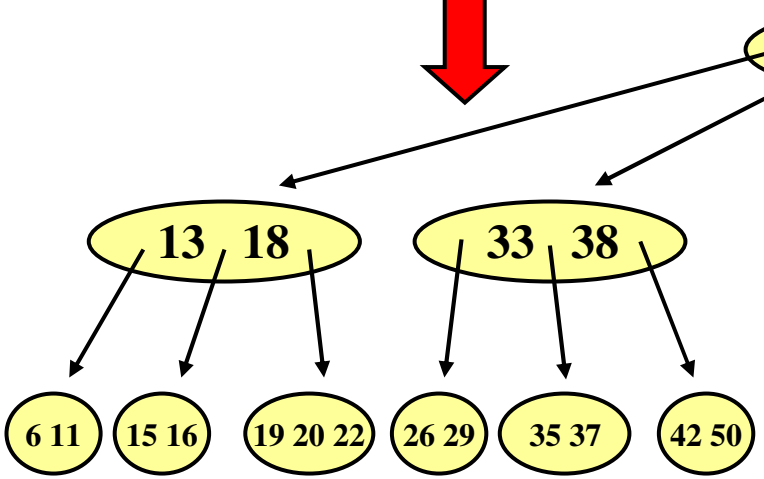
Genauer: Das Blatt besitzt nach dem Löschen von s' nur m-1 Schlüssel, aber ein Geschwisterknoten besitzt mindestens m+1 Schlüssel. Dann wird der zugehörige Schüssel des Eltern-knotens in das Blatt geschoben und an seine Stelle tritt der nächstgelegene Schlüssel aus dem Geschwisterknoten. Skizze:



Beispiel zu Fall 2: Lösche den Schlüssel 28

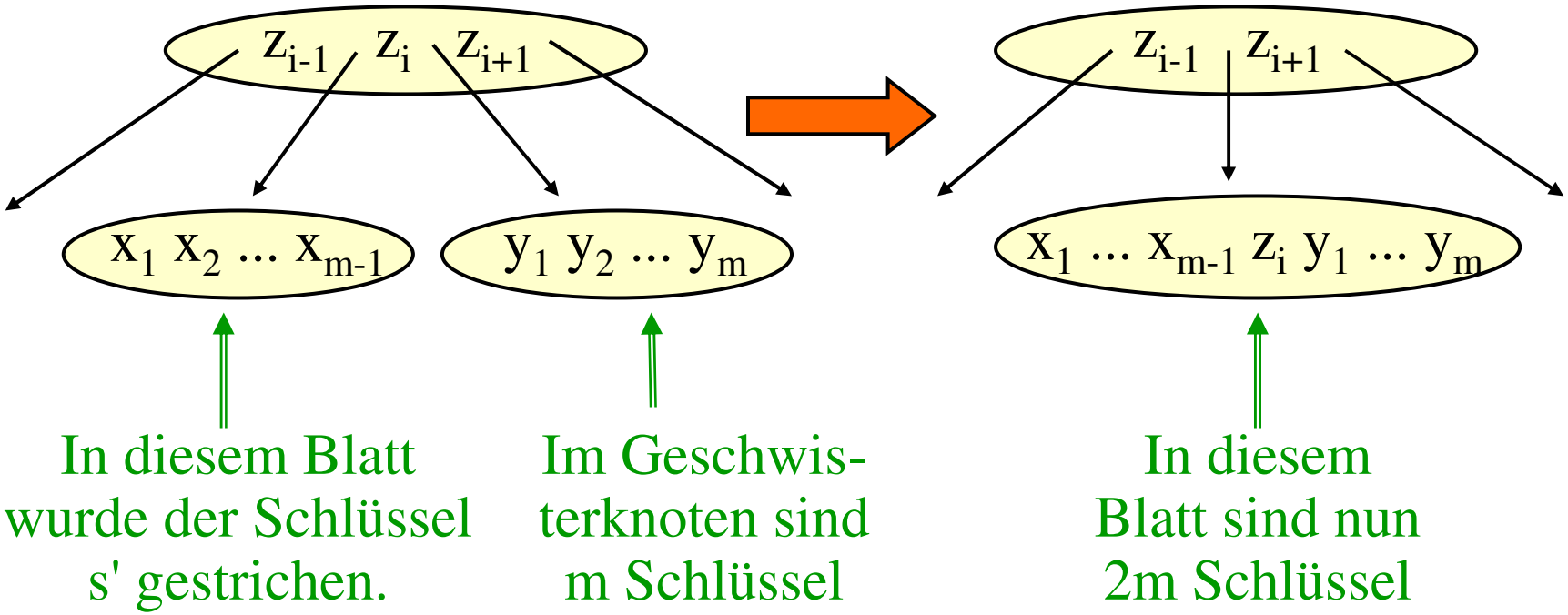


Die sortierte Folge 26 29 33 35 37 wird um eins nach links geschoben unter Benutzung des Elternknotens.



Man beachte, dass der Knoten mit dem Inhalt "19 20 22" nicht zum Ausgleich herangezogen werden kann, da er kein Geschwisterknoten ist.

Fall 3: Das Blatt besitzt m Schlüssel und die Geschwisterknoten besitzen ebenfalls m Schlüssel. Dann wird das Blatt mit einem Geschwisterknoten verschmolzen unter Einbeziehung des zugehörigen Schlüssels im Elternknoten (falls das Blatt das rechteste Kind des Elternknotens ist, so nimm den links daneben liegenden Geschwisterknoten, anderenfalls kann man stets den rechts daneben liegenden nehmen):



Fall 3 (Fortsetzung): In diesem Fall wird die Zahl der Schlüssel im Elternknoten verringert. Wende daher rekursiv auf den Elternknoten die Fälle 1 bis 3 an.
Hierbei kann es geschehen, dass jedes Mal Fall 3 auftritt und schließlich ein Schlüssel aus der Wurzel entfernt wird.
Wenn die Wurzel hierbei noch mindestens einen Schlüssel behält, so ist man fertig. Falls die Wurzel aber nur einen Schlüssel besaß, so wird die Wurzel leer und der einzige, neu entstandene verschmolzene Knoten unter ihr wird zur neuen Wurzel des Baumes. Genau in diesem Fall wird die Tiefe des Baumes um 1 verringert.

Da der B-Baum höchstens die Tiefe $\log_{m+1}(n)+1$ besitzt, erfordert also auch das Löschen nur O(log(n)) Schritte.

8.5.6 Speicherplatzausnutzung durch B-Bäume

Ein B-Baum ist stets zu mindestens 50% gefüllt. Wie viel Speicherplatz wird aber tatsächlich ausgenutzt? Man wird 75% erwarten, jedoch zeigt eine theoretische Analyse, dass es im Mittel ca. 69% sind.
In der Praxis wurde durch Messungen festgestellt, dass B-Bäume meist nur zu zwei Drittel (knapp 67%) gefüllt sind.
Man sollte daher das Einfügen von Schlüsseln so implementieren, dass das Aufspalten von Knoten möglichst lange vermieden wird (z.B., indem man die Geschwisterknoten einbezieht, wie es beim Löschen geschieht). Beim Löschen sollte man zufällig bestimmen, ob man beim Löschen in inneren Knoten den Inorder-Nachfolger oder -Vorgänger verwendet.
Selbst nachdenken!

8.5.7 Zur Implementierung:

Für die Implementierung kann man in jedem Knoten ein

array [1..2*m] für die Schlüssel, ein
array [1..2*m] für die zugehörigen Inhalte und ein
array [0..2*m] für die Zeiger auf die Kinder

mitführen. Zusätzlich ist eine natürliche Zahl "Anzahl" zu speichern, die die aktuelle Zahl der Schlüssel angibt, sowie eine Boolesche Variable für die Eigenschaft "Blatt".

Dies führt zu folgender Datentypdefinition :

....

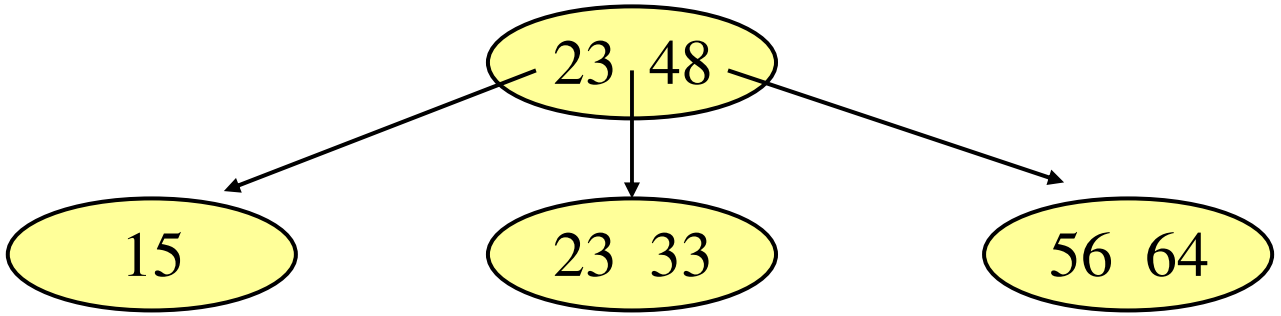
(selbst einsetzen!).

Alternative: In der Praxis nennt man einen B-Baum-Knoten auch "Seite". Zeiger werden oft durch Ref_ oder durch Ptr_ (von "pointer") bezeichnet. m sei hier als Konstante vorgegeben. In einem Knoten stehen bis zu 2m Schlüssel mit den durch die Schlüssel eindeutig charakterisierten Inhalten. SchlüsselTyp sei der Typ der Schlüssel, InhaltTyp sei der Typ der Inhalte. Einen Schlüssel, einen Inhalt und einen Zeiger auf den Knoten mit den nächstgrößeren Schlüsseln fassen wir als "Item" zusammen. Dann bilden bis zu 2m Items plus ein Zeiger auf das linkeste Kind den gesamten Knoten. Eine mögliche Datentypdefinition für B-Bäume lautet daher:

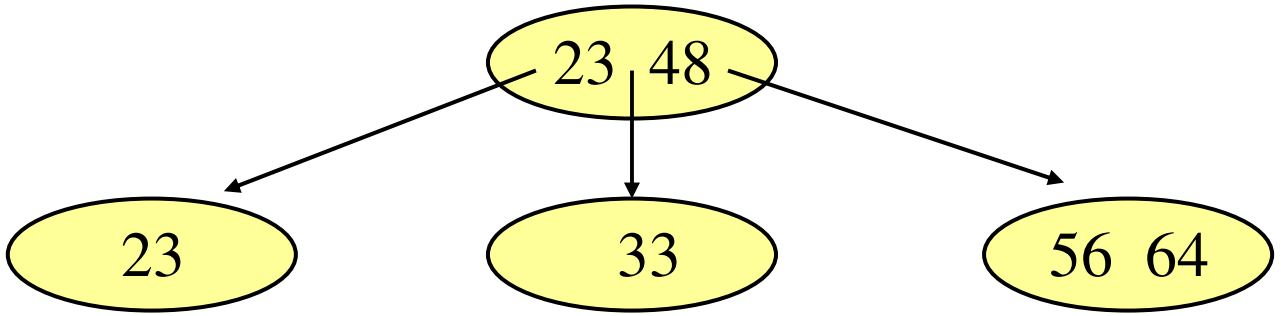
```
type Seite;
type Ptr_Seite is access Seite;
type Item is  record Schlüssel: SchlüsselTyp;
                     Inhalt: InhaltTyp;
                     Zeiger: Ptr_Seite;
                   end record;
type Seite is record Anzahl: Natural;
                     Blatt: Boolean;
                     linkeSeite: Ptr_Seite;
                     Items: array (1..2*m) of Item;
                   end record;
```

8.5.8 Gleiche Schlüssel

Wenn gleiche Schlüssel auftreten dürfen, dann lässt sich die allgemeine Suchbaumeigenschaft mit den bisher vorgestellten Operationen nicht gewährleisten. Beispiel: Betrachte den folgenden B-Baum der Ordnung m=1:

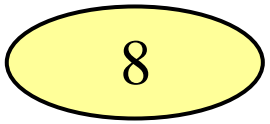


Wird nun "15" gelöscht, so wird vom Knoten "23 33" der linke Schlüssel "23" in den Elternknoten und dessen linker Schlüssel "23" in das Blatt geschoben. Ergebnis:

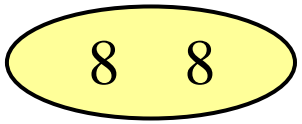


Da "23" nun links von "23" des Elternknotens steht, ist die allgemeine Suchbaumeigenschaft verletzt.

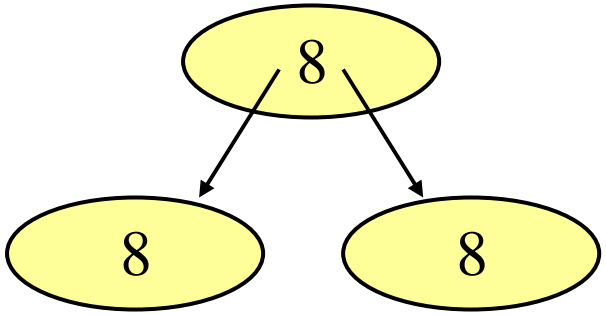
Dies lässt sich bei B-Bäumen prinzipiell nicht verhindern. Fügt man in einen B-Baum nacheinander den gleichen Schlüssel (z.B. 8) ein, so tritt der Elternschlüssel zwangsläufig sowohl im linken wie im rechten Unterbaum auf (die Ordnung sei hier erneut 1):



B-Baum
nach Einfügen
der ersten "8"



B-Baum
nach Einfügen
der zweiten "8"



B-Baum
nach Einfügen
der dritten "8"

Wir müssen also die Suchbaumeigenschaft abschwächen:

Definition: Abgeschwächte Suchbaumeigenschaft

Gegeben sei ein geordneter Baum. In jedem Knoten steht ein nicht-leeres sortiertes Tupel von Schlüsseln einer geordneten Menge. Für alle Knoten u muss gelten:
Sind $s_1, s_2, ..., s_k$ die sortierten Schlüssel von u ($s_1 \leq s_2 \leq ... \leq s_k$), so besitzt u genau k+1 Kinder und es gilt:
Alle Schlüssel im Unterbaum des ersten Kindes sind kleiner als s_1, alle Schlüssel im Unterbaum des zweiten Kindes sind größer oder gleich s_1 und kleiner als s_2, alle Schlüssel im Unterbaum des i-ten Kindes sind größer oder gleich s_{i-1} und kleiner als s_i (für $i = 2, 3, ..., k$), und alle Schlüssel im Unterbaum des (k+1)-ten Kindes sind größer oder gleich s_k.

Einen geordneten Baum mit dieser Eigenschaft bezeichnen wir ebenfalls als (allgemeinen) Suchbaum.

Diese Definition erzwingt, dass zu jedem Schlüssel (genauer: zu jedem Item, siehe 8.5.7) eine Boolesche Variable mitgeführt wird, die angibt, ob sich im linken Unterbaum ebenfalls gleiche Schlüssel befinden. Diese Variable ist beim erstmaligen Auftreten eines Schlüssels false. Sie wird in einem Elternknoten auf true gesetzt, wenn dieser Knoten beim Einfügen aufgespalten wurde und hierbei der Schlüssel in den linken Unterbaum verschoben und durch den gleichen Schlüssel ersetzt wurde oder wenn beim Löschen ein Ausgleich über den Elternknoten mit einem gleichen Schlüssel erfolgte. Andererseits muss sie auf false zurückgesetzt werden, falls alle gleichen Schlüssel im linken Unterbaum gelöscht wurden.
Kriterium: Diese Boolesche Variable muss zu einem konkreten Schlüssel s genau dann den Wert true besitzen, wenn s nicht in einem Blatt steht und wenn s gleich seinem Inorder-Vorgänger ist.

Hierdurch müssen nun alle Algorithmen für die Operationen modifiziert werden. Bei der vollständigen Suche nach allen k Vorkommen eines Schlüssels können nun bis zu $O(k \cdot 2m \cdot \log_{m+1}(n))$ Vergleiche notwendig werden. Auch beim Einfügen und Löschen erhöht sich die Zahl der Schritte.

Man kann dies vermeiden zum Beispiel durch:

- Man speichere jeden Schlüssel im B-Baum nur einmal und lege für jeden Schlüssel eine lineare Liste an, auf deren Anfang von B-Baum-Knoten verwiesen wird.
- Man verwende B*-Bäume, bei denen die unterste Schicht, in der alle Schlüssel und Inhalte stehen, doppelt verkettet ist.

Über Details selbst nachdenken.