

Einführung in die Informatik II

Universität Stuttgart, Studienjahr 2007

Gliederung der Grundvorlesung

8. Suchen

9. Hashing

10. Sortieren

11. Graphalgorithmen

12. Speicherverwaltung

8. Suchen

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 Löschenzeiten 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 $A = \{a_1, \dots, a_n\} \subseteq B$ 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 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  $\neq$  null and then p.inhalt  $\neq$  s loop  
      p := p.next; end loop;  
if p = null then  $\langle s$  ist nicht in der Liste enthalten  $\rangle$   
else  $\langle p$  verweist auf das erste Element mit Inhalt  $s \rangle$  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?

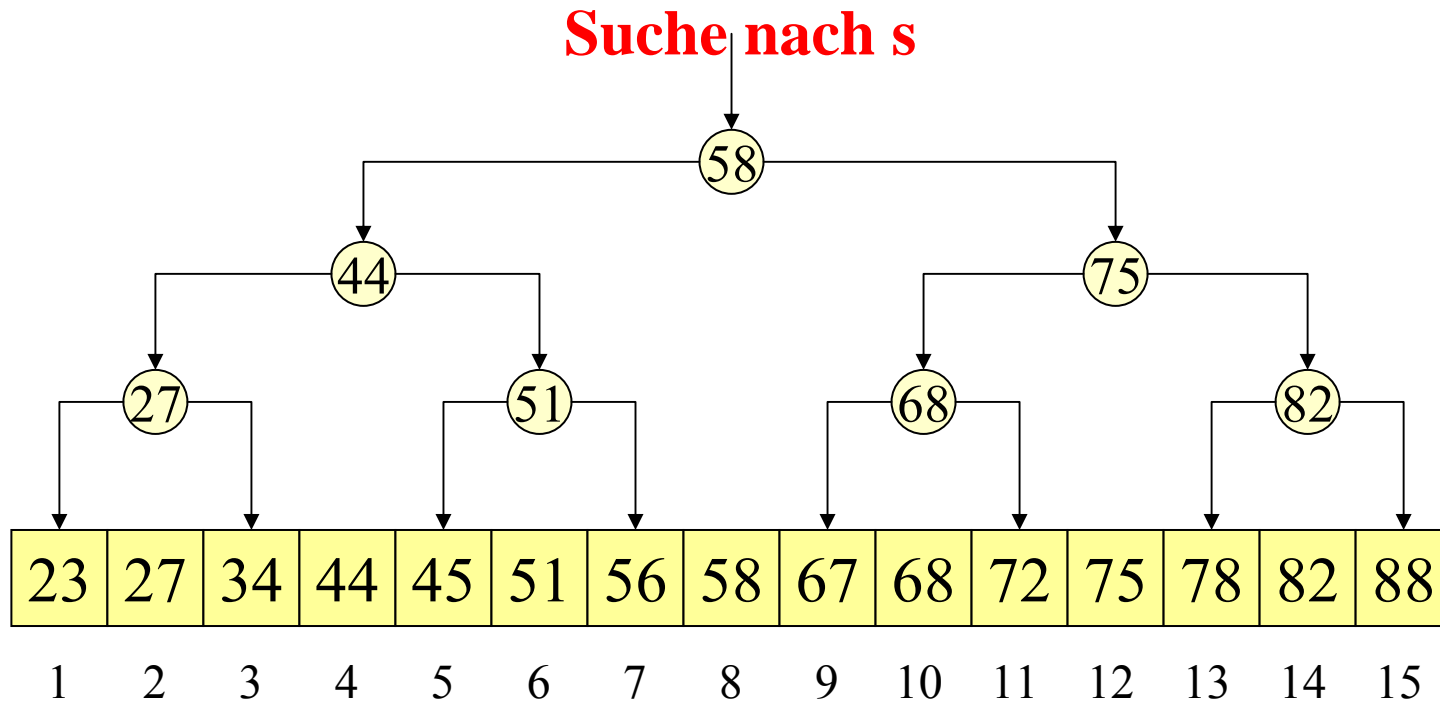
Wir wählen ein array als Datenstruktur, in das die Menge der Größe nach geordnet eingetragen wird. Dann

- dauert FIND nur $O(\log(n))$ Schritte, sofern man ein geordnetes array verwendet und hierauf mit der Intervallschachtelung 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 $A(\text{links}..\text{rechts})$. Als Teilungsindex "Mitte" wählen wir:
 $\text{Mitte} = (\text{rechts} + \text{links}) / 2 = \text{links} + (\text{rechts} - \text{links}) / 2$.

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 $A(\text{links}..\text{rechts})$ 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 $A = \{a_1, \dots, a_n\} \subseteq B = \{b_1, \dots, b_r\}$ 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 $x = (x_1, \dots, x_r)$, 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
 $|\text{Vor}(x)|=1$ für alle Knoten $x \neq w$ und $|\text{Vor}(w)|=0$. Der
eindeutig bestimmte Knoten $\text{vorg}(x) = y \in \text{Vor}(x)$ 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 $x \in \text{Nach}(y)$ heißt **direkter Nachfolger** oder
Kind oder **Sohn** von x .
- (4) Gerichteter Baum: Ein Knoten x mit $x \neq w$ und mit
 $|\text{Nach}(x)|=0$ heißt **Blatt** des Baums.
Ungerichteter Baum: Ein Knoten x mit $x \neq w$ und mit
 $|\text{N}(x)|=1$ 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
- $$V_x = \{y \mid \text{jeder doppeltpunktfreie Weg von } w \text{ nach } y \text{ führt über } x\},$$
- $$E_x = E|_{V_x} \text{ (= alle Kanten zwischen Knoten aus } V_x).$$
- 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 $f: V \rightarrow \{1,2\}$ 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

$$f: V \rightarrow \{1,2,\dots,k\}$$

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 $\text{Nach}(x)$ geordnet (d.h., die Knoten y_i der Menge $N(x)$ bzw. $\text{Nach}(x)$ sind angeordnet: $y_1 < y_2 < \dots < y_k$), 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 $v: V \times \{1, \dots, k\} \rightarrow V \cup \{0\}$, so dass für alle $x \in V$ gilt:

$$\text{Nach}(x) \subseteq \{v(x, i) \mid i = 1, \dots, k\},$$

aus $v(x, i) \neq 0$, $v(x, j) \neq 0$ und $i \neq j$ folgt $v(x, i) \neq v(x, j)$,

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 $\text{level}: V \rightarrow \mathbb{N}_0$, rekursiv definiert durch
$$\begin{aligned}\text{level}(w) &= 1 \text{ und} \\ \text{level}(x) &= \text{level}(\text{vorg}(x)) + 1 \quad \text{für } x \neq w.\end{aligned}$$

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

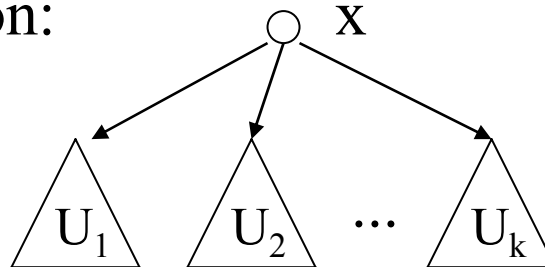
x bildet die Wurzel des Baums $x(U)$, die Elemente von U sind Unterbäume oder Teilbäume im Baum $x(U)$.

Skizze: Leerer Baum: \emptyset

Tiefe dieses Baumes = 0

gerichtet
oder
ungerichtet

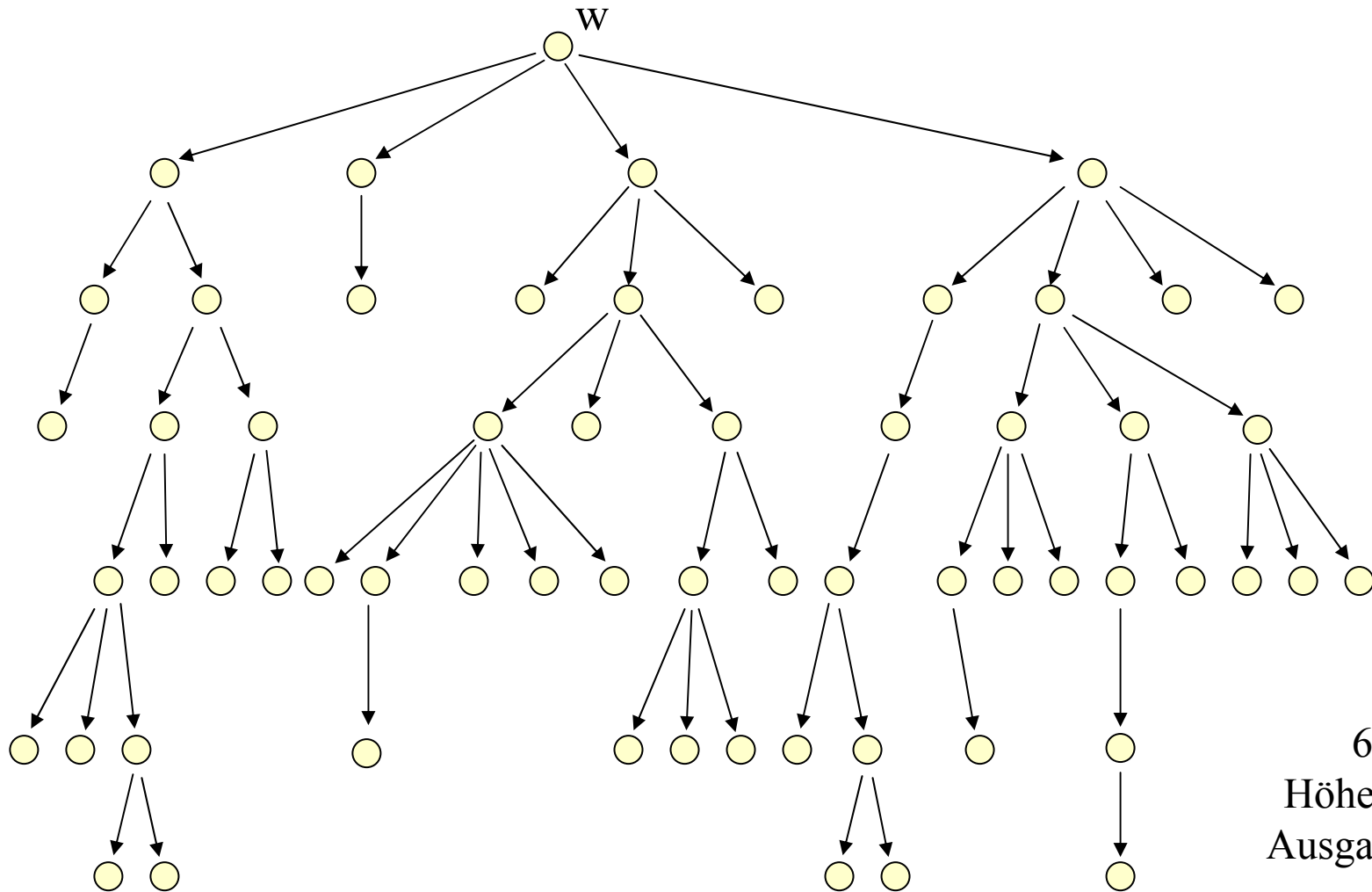
Rekursion:



Tiefe dieses Baumes =
 $\text{Max}(\text{Tiefe eines } U_i) + 1$.

Die k Nachfolger von
 x sind hier geordnet.

Beispiel für einen "beliebigen geordneten Baum":



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*).

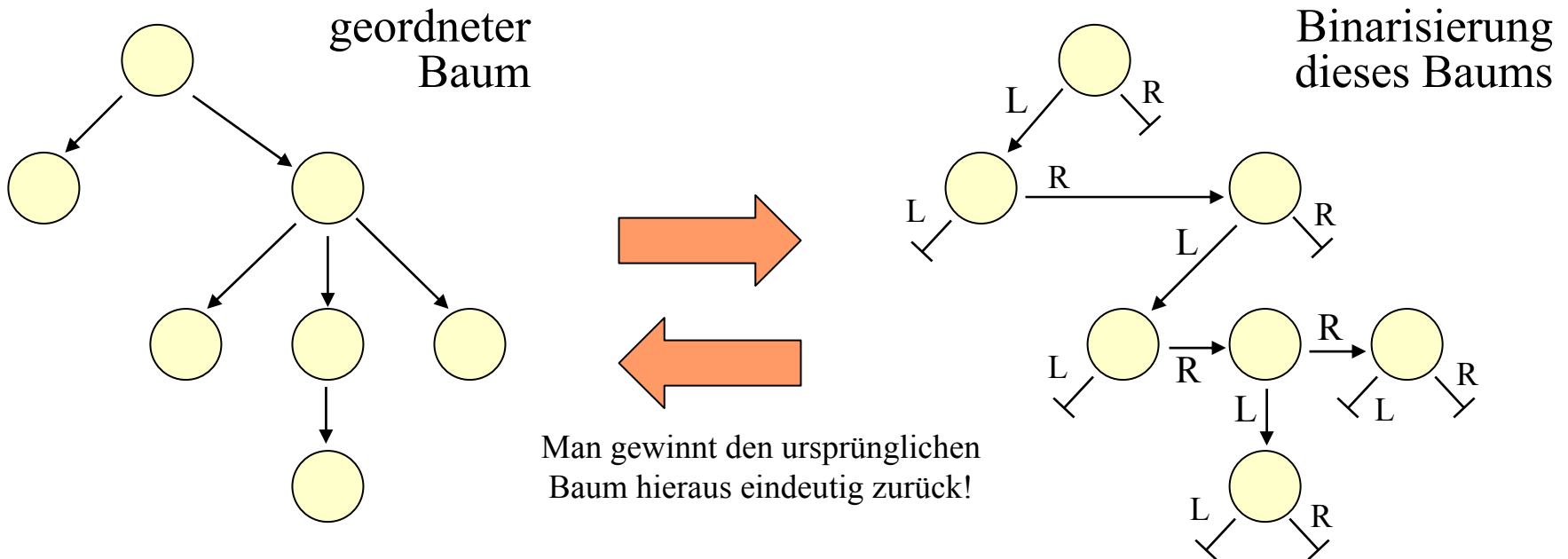
Die Darstellung binärer Bäume in Ada lautet bekanntlich (hier: mit dem Inhalt vom Typ Integer):

```
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.



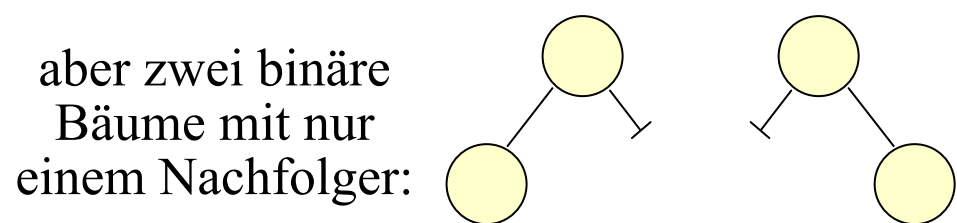
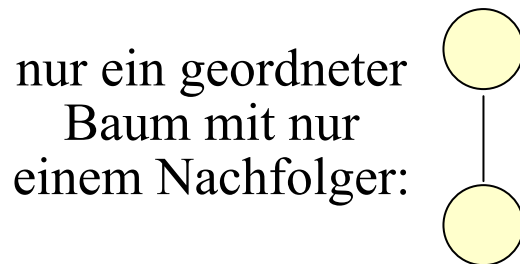
Folgerung aus dieser eindeutigen Umwandlung:

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.:



Hinweis: Eine zu unserer Binarisierung verwandte Darstellung ist die Ordnerhierarchie in einem Betriebssystem; dort listet man untereinander 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, \dots, C_4 durch Aufzählen aller zugehöriger Binärbäume.

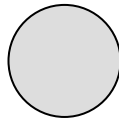
Wir listen alle binären Bäume mit höchstens 4 Knoten auf. Die Wurzel des binären Baums ist hier grau dargestellt. Die leeren Zeiger wurden weggelassen.

n = 0

<leerer Baum>

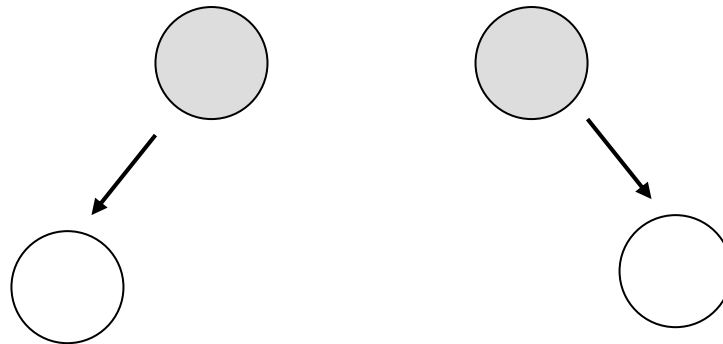
$C_0 = 1$

n = 1



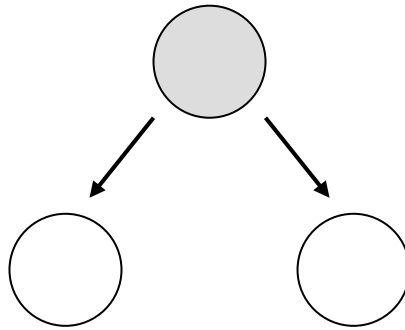
$C_1 = 1$

n = 2

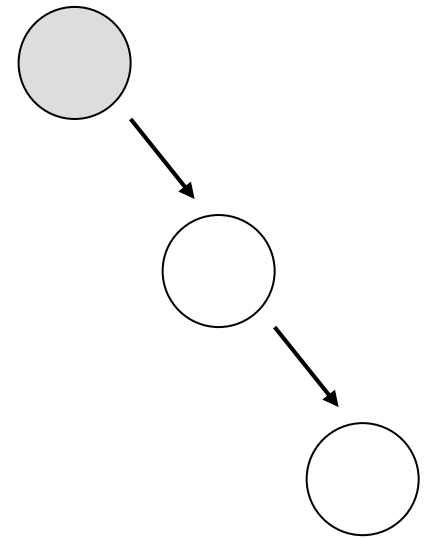
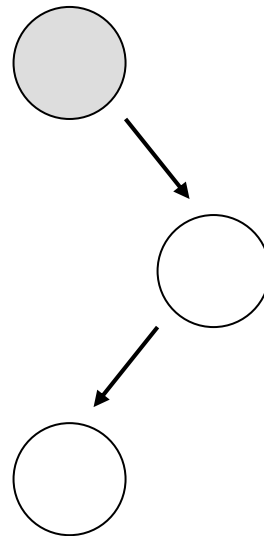
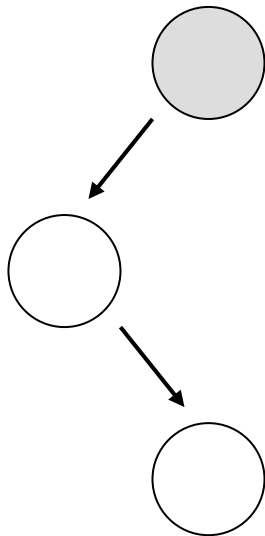
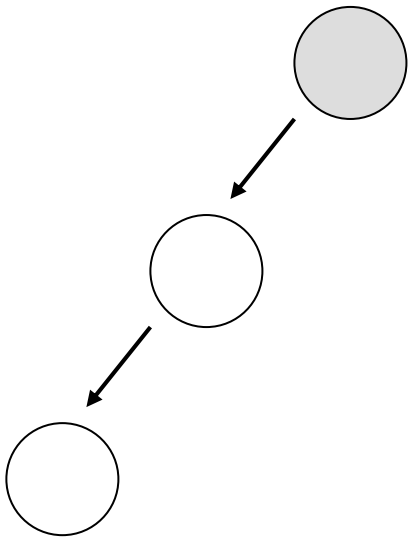


$C_2 = 2$

$n = 3$

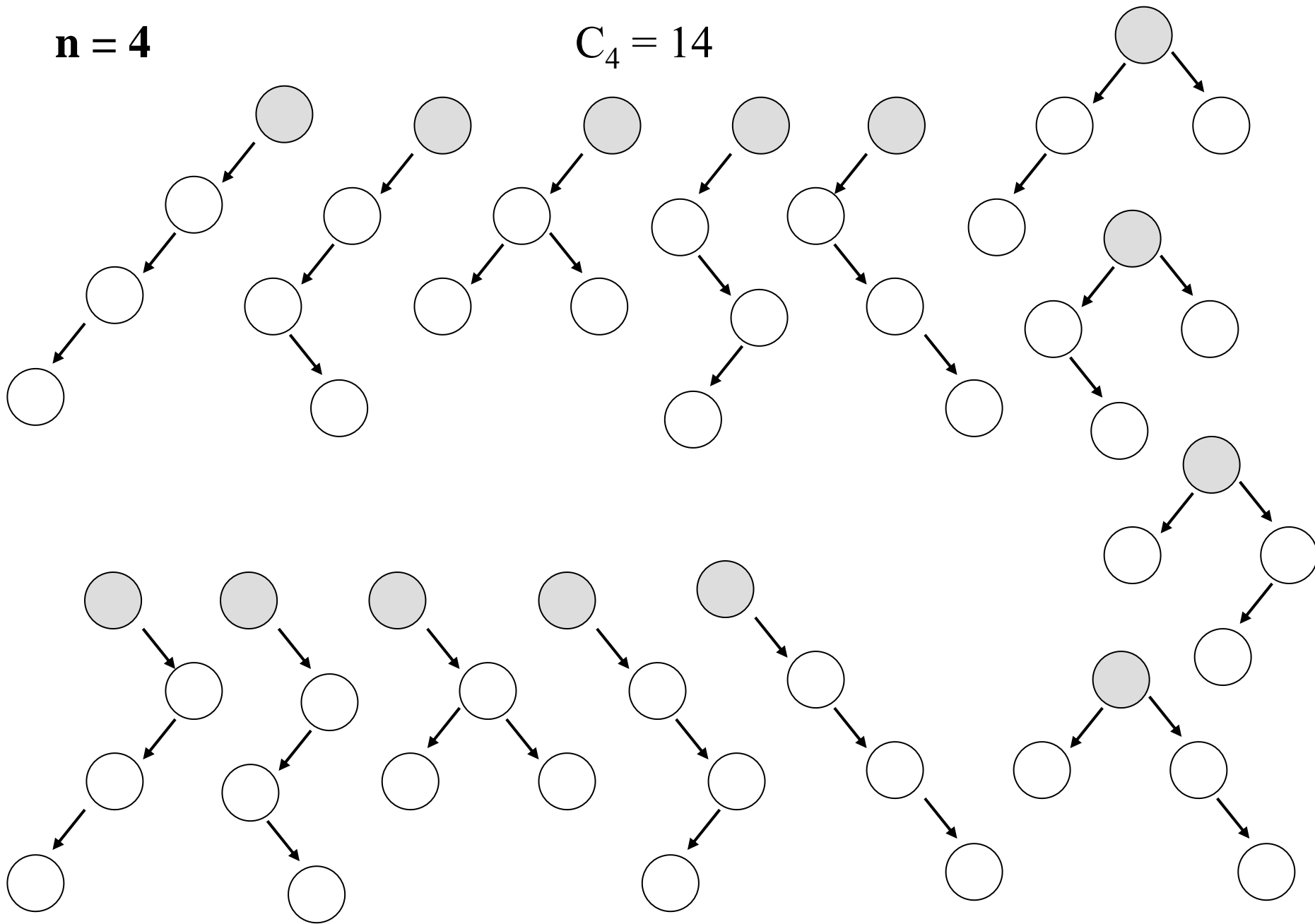


$C_3 = 5$



n = 4

$C_4 = 14$



Durch Ausprobieren
erhält man die Werte:

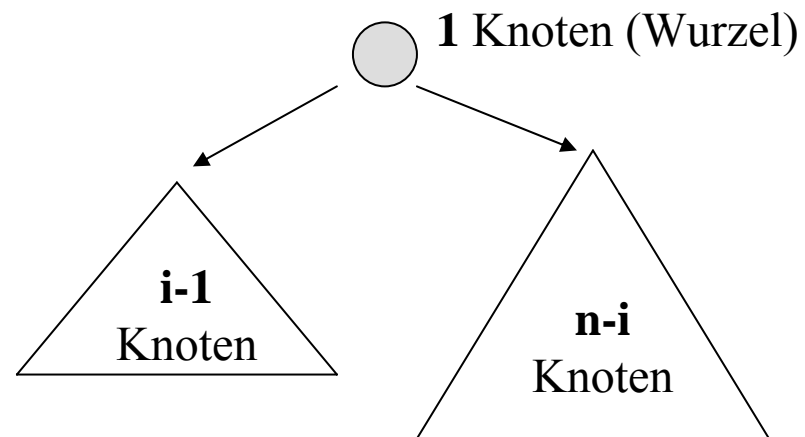
n	Anzahl
0	1
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430

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

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

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 " = $(49 \cdot 48 \cdot 47 \cdot 46 \cdot 45 \cdot 44) / (1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6) = 13\,983\,816$ Möglichkeiten, aus 49 Zahlen 6 Zahlen auszuwählen, sofern die Reihenfolge des Auswählens keine Rolle spielt.

8.2.7 Satz "Catalansche Zahlen"

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

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 Klammerungsmöglichkeiten für $n = 3$:
 $((()))$, $((())())$, $(())(())$, $(())(())$, $(())(())$.

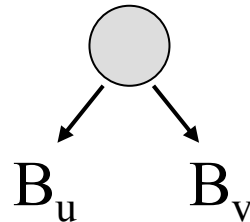
Behauptung: Die Anzahl C_n der Binärbäume ist gleich der Anzahl der Möglichkeiten, um n „Klammer auf“ und n „Klammer zu“ wie in korrekt geklammerten Ausdrücken aneinander zu reihen:

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

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 Klammersausdruck 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 $u=v=\varepsilon$ 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".

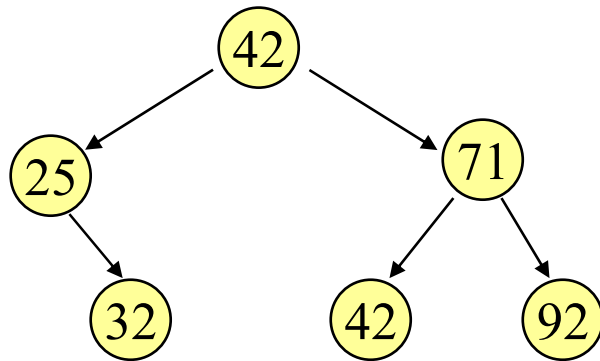
```
procedure Preorder (K: Ref_BinBaum) is
begin if K /= null then < bearbeite den Knoten K >;
      Preorder (K.L);
      Preorder (K.R);
    end if;
end Preorder;
```

```
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;
```

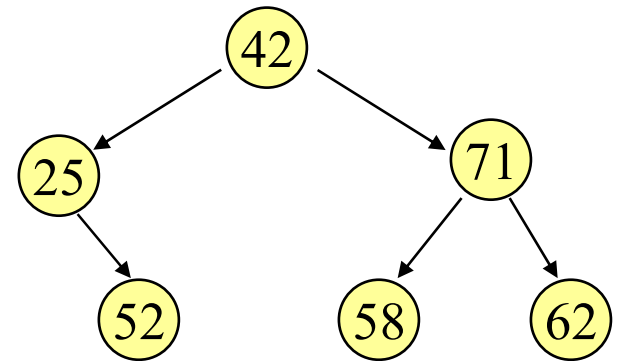
```
procedure Postorder (K: Ref_BinBaum) is
begin if K /= null then Postorder (K.L);
      Postorder (K.R);
      < bearbeite den Knoten K >;
    end if;
end Postorder;
```

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, \dots, 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, \dots, 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.

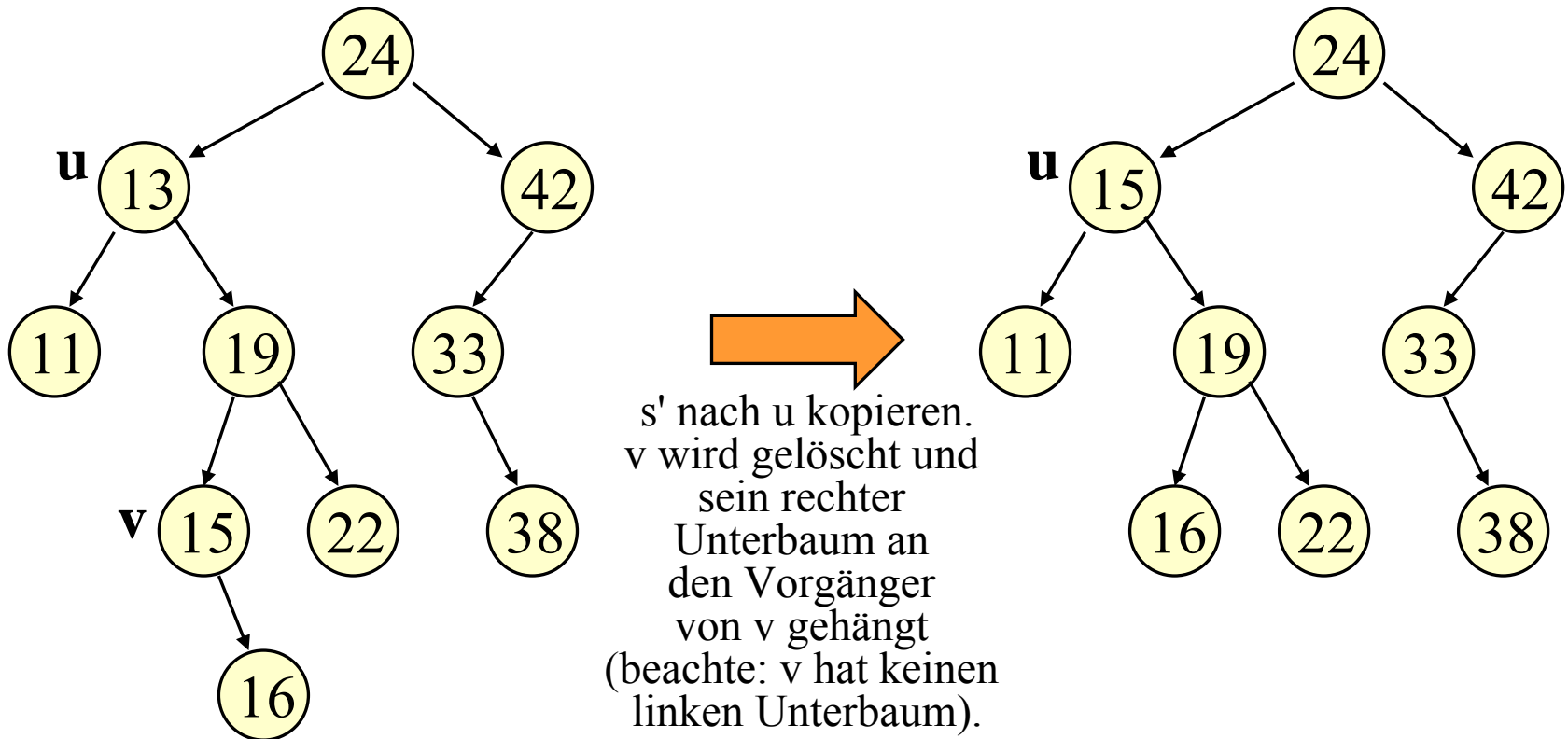
s' möge bzgl. der Sortierung der unmittelbar folgende Schlüssel von s sein. Da die sortierte Reihenfolge durch einen inorder-Durchlauf erreicht wird, nennt man s' und seinen Knoten v den "**Inorder-Nachfolger**" von s bzw. von u. s steht in dem Knoten v, der in der inorder-Reihenfolge auf u folgt: v 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.

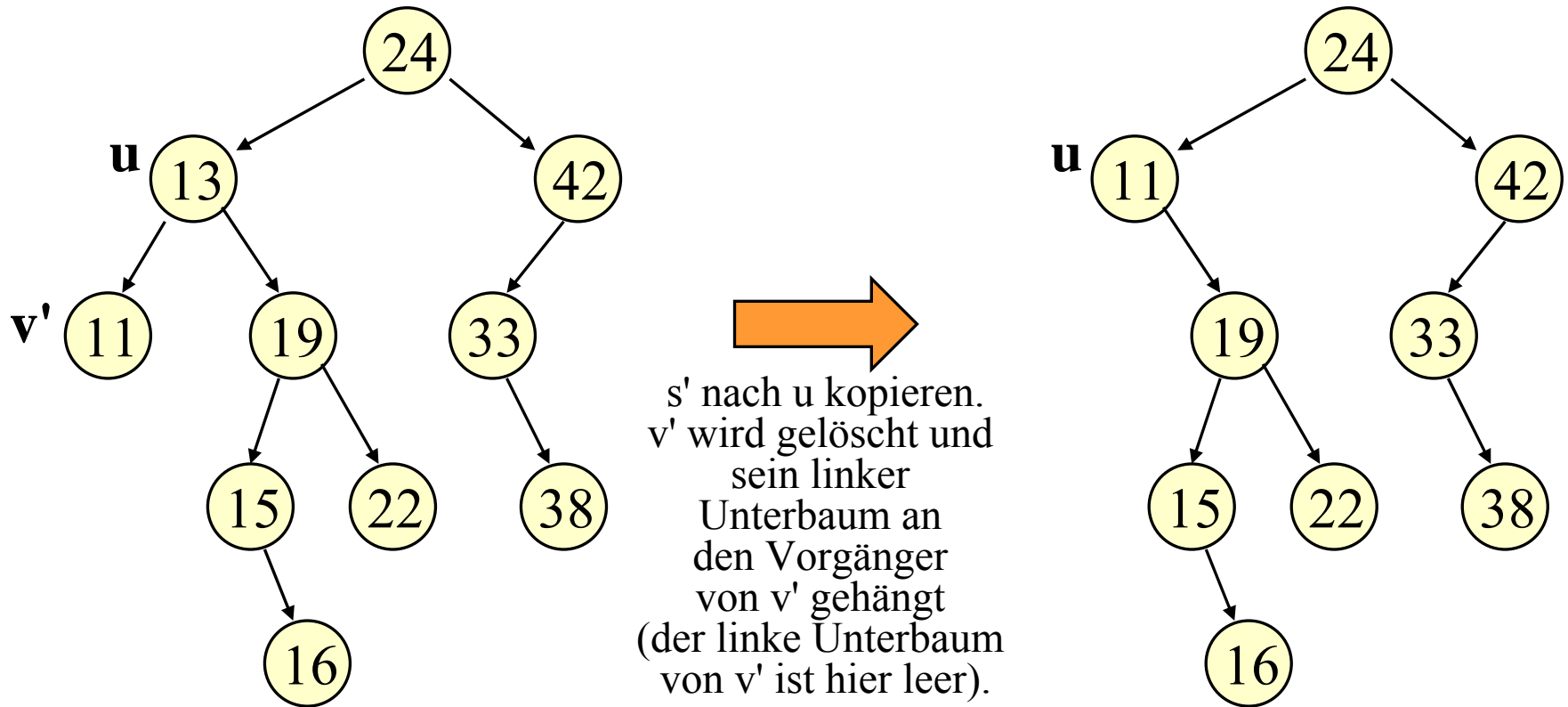
In der Praxis entscheidet man sich jedes Mal zufällig, ob man den Inorder-Vorgänger oder den Inorder-Nachfolger für das Löschen heranzieht.

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:

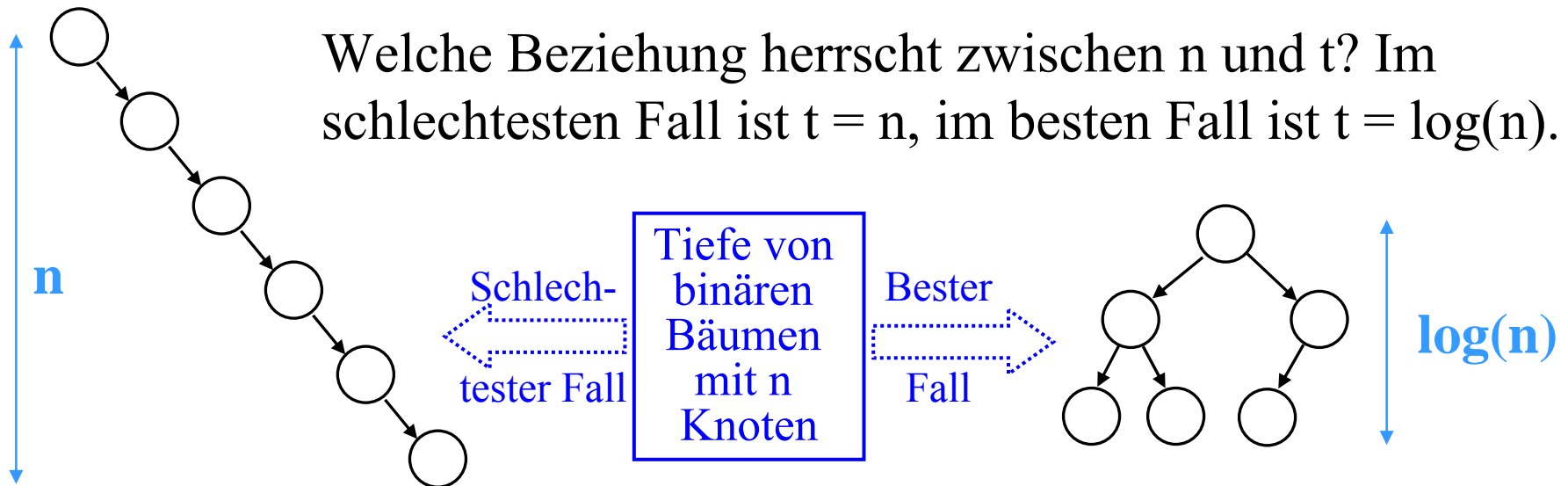
```
procedure Löschen (Anker: in Ref_BinBaum; s: in Integer) is  
u, v, vorg: Ref_BinBaum := null; links: Boolean;  
begin "Suche(Anker, s, u, vorg, links);"  
    -- dies ist neu zu programmieren: vorg zeigt auf den Elternknoten von u  
    -- (sofern vorhanden) und links ist true, falls u linkes Kind 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).



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 $\log: \mathbb{N} \rightarrow \mathbb{N}_0$ 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 doppelunktfreie 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:

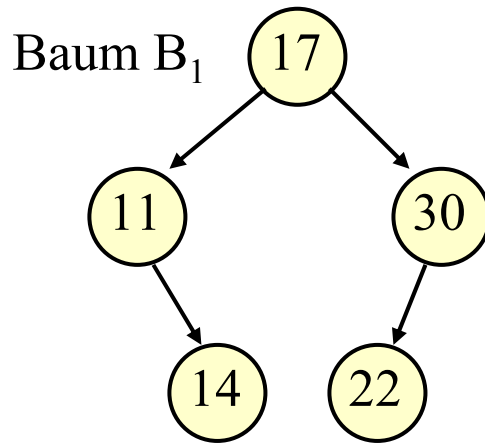
Definition 8.2.13.a: Man betrachte alle C_n binären Bäume mit n Knoten ($n > 0$). Für jeden Baum B mit n Knoten v_1, \dots, v_n berechne man das **mittlere Level** oder die **mittlere Suchzeit** ml :

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

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

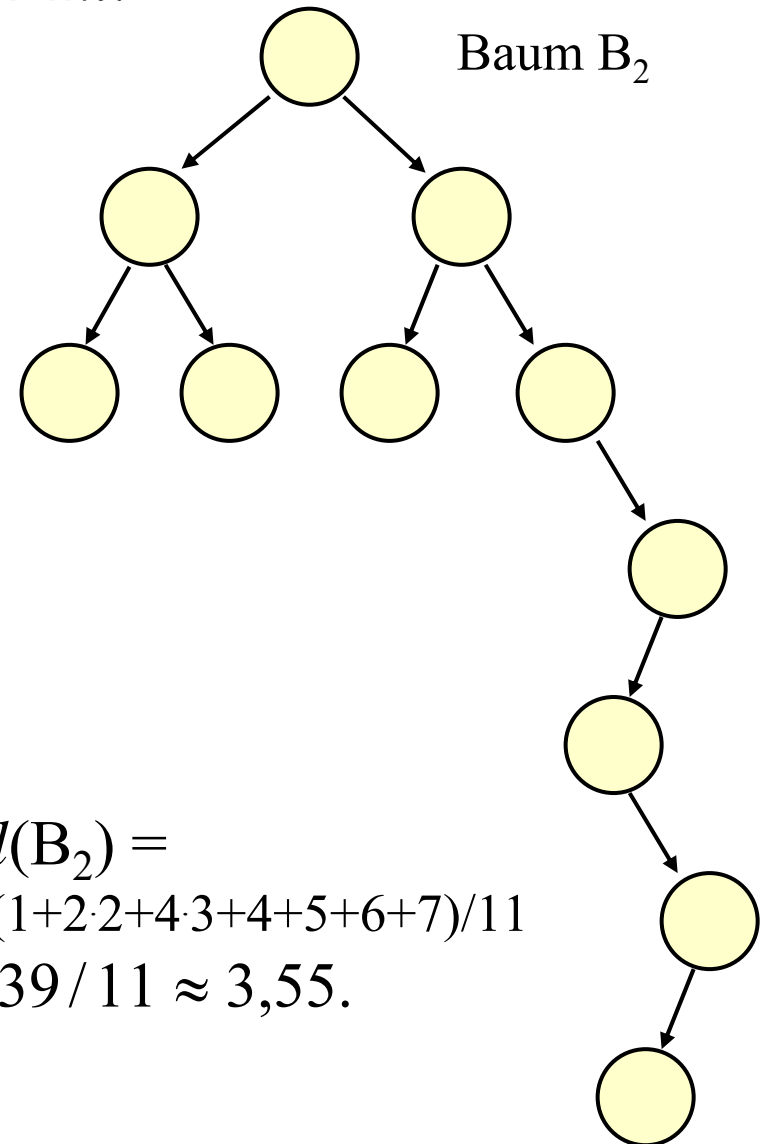
$$ML_n = \frac{1}{C_n} \sum_{\substack{B \text{ ist binärer} \\ \text{Baum mit } n \\ \text{Knoten}}} ml(B)$$

Zwei Binärbäume als Beispiele für ml :



$$\begin{aligned} ml(B_1) &= (1+2+2+3+3)/5 \\ &= 11/5 = 2,2 \end{aligned}$$

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



$$\begin{aligned} ml(B_2) &= \\ &= (1+2 \cdot 2+4 \cdot 3+4+5+6+7)/11 \\ &= 39/11 \approx 3,55. \end{aligned}$$

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 $n = 1, \dots, 4$):

$$ML_1 = 1 / 1 = 1$$

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

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

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

$$ML_5 = (1/42) \cdot (562/5) = 2,67619\dots$$

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

Wir kommen zum Ansatz 2:

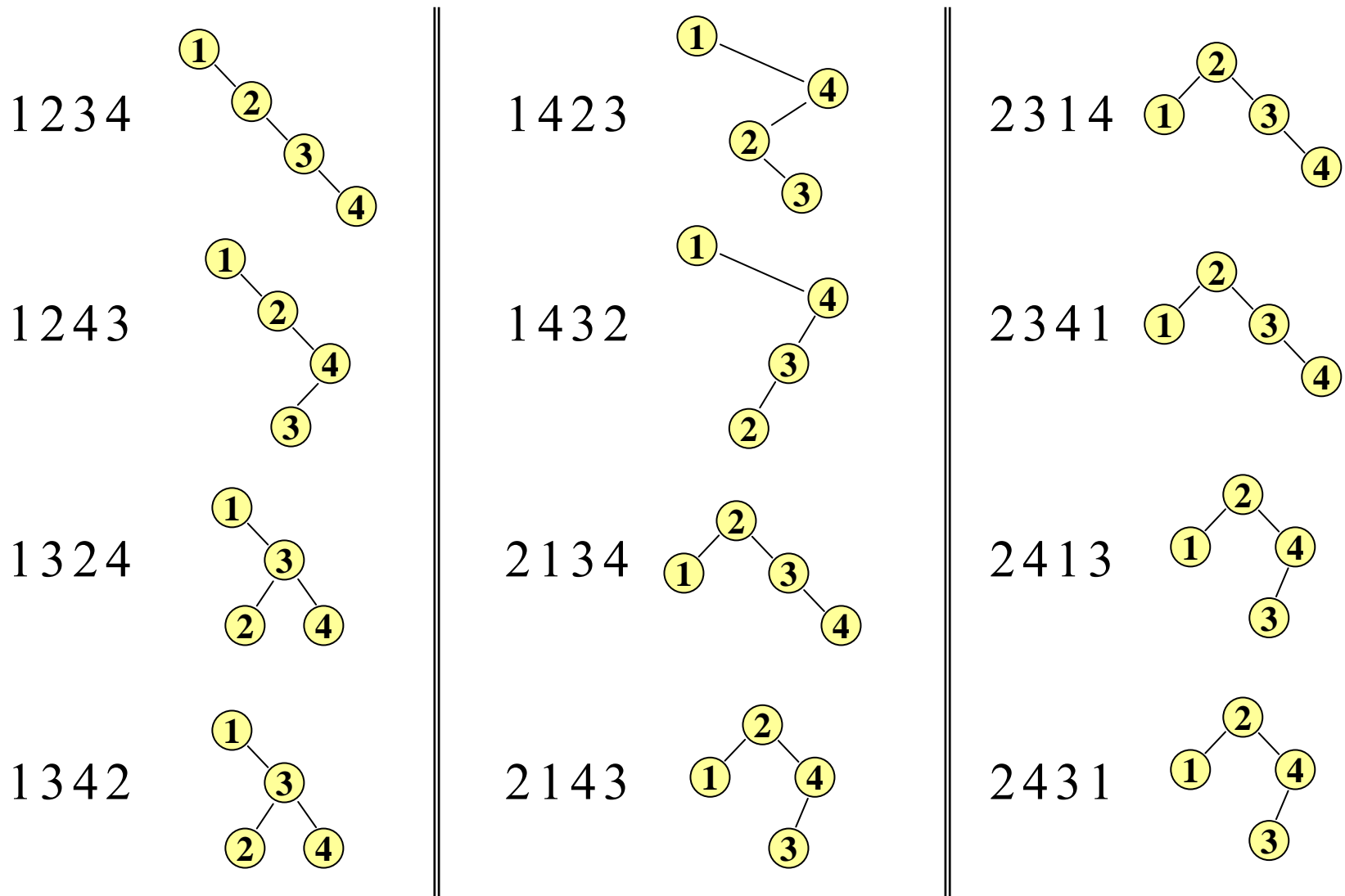
Definition 8.2.13.b: Gesucht wird die mittlere Suchzeit aller Bäume, die zu allen Folgen von n Elementen gehören. Das heißt: Man gehe von den $n!$ Folgen mit n Elementen aus, füge jede Folge entsprechend ihrer Reihenfolge in einen binären Suchbaum ein und frage dann nach der mittleren Suchdauer dieser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

Es sei also $a = a_1 a_2 \dots a_n$ 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, \dots, 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:



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:

$$\begin{aligned} MS_4 &= (1/24) (10/4 + 10/4 + 9/4 + 9/4 + 10/4 + 10/4 \\ &\quad + 8/4 + 8/4 + 8/4 + 8/4 + 8/4 + 8/4 + \dots) \\ &= 53/24 = 2,20833\dots \end{aligned}$$

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 $1 \leq i \leq n$. Ist nun jedes i gleichwahrscheinlich (und dies darf man beim zufälligen Aufbauen annehmen), dann erhält man

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

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

$$\begin{aligned} F(n) &= (1/n) \cdot ((F(0) + F(n-1) + n) + (F(1) + F(n-2) + n) + \dots \\ &\quad (F(n-1) + F(0) + n)) \\ &= (1/n) \cdot 2 \cdot (F(0) + F(1) + F(2) + \dots + F(n-1)) + n \end{aligned}$$

Folglich gilt

$$\begin{aligned} F(n-1) &= (1/(n-1)) \cdot 2 \cdot (F(0) + F(1) + F(2) + \dots + F(n-2)) + n-1 \\ &= (n/(n-1)) \cdot (1/n) \cdot 2 \cdot (F(0) + F(1) + F(2) + \dots + F(n-2)) + n-1 \end{aligned}$$

Den Wert für

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

setzen wir nun oben ein und erhalten die Rekursionsformel:

$$\begin{aligned} F(n) &= (1/n) \cdot 2 \cdot F(n-1) + (n-1)/n \cdot (F(n-1) - n+1) + n \\ &= (n+1)/n \cdot (F(n-1) - (n-1)^2/n + n) = (n+1)/n \cdot F(n-1) + (2n-1)/n \end{aligned}$$

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:

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

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

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

$$\begin{aligned}
\frac{F(n)}{n+1} &= \sum_{i=0}^{n-1} \frac{2}{(n-i+1)} - \sum_{i=0}^{n-1} \frac{1}{(n-i) \cdot (n-i+1)} \\
&= 2 \cdot \sum_{i=2}^{n+1} \frac{1}{i} - \sum_{i=1}^n \frac{1}{i \cdot (i+1)} \\
&= 2 \cdot (H(n+1) - 1) - \sum_{i=1}^n \left(\frac{1}{i} - \frac{1}{i+1} \right) \\
&= 2 \cdot H(n+1) - 2 - 1 + \frac{1}{n+1} \quad \text{mit } H(n) = \sum_{i=1}^n \frac{1}{i}
\end{aligned}$$

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

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: $H(n) - \ln(n) \rightarrow \gamma$ für $n \rightarrow \infty$.

\ln ist der natürliche Logarithmus (zur Basis $e = 2,7182818284\dots$),
 $\gamma = 0,5772156649\dots$ ist die Eulersche Konstante. Einsetzen ergibt:

$$\begin{aligned} F(n) &\approx 2 \cdot (n+1) \cdot (\ln(n) + \gamma) - 3 \cdot n && \text{(Hier ist log der reellwertige Zweierlogarithmus.)} \\ &\approx 2 \cdot (n+1) \cdot \log(n)/\log(e) + 2 \cdot (n+1) \cdot \gamma - 3 \cdot n \\ &\approx 1,3863 \cdot n \cdot \log(n) - 1,8456 \cdot n + O(\log(n)). \text{ Bilde nun } F(n)/n. \end{aligned}$$

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 Zeitkom-
plexität 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-Nachfolger 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.)

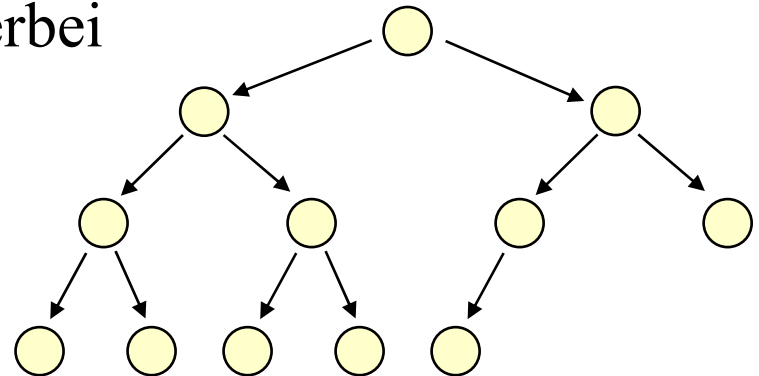
Kann man den Suchbaum so einschränken, dass Entartungen möglichst nicht eintreten können und auch im schlechtesten Fall nur $O(n \cdot \log(n))$ Schritte benötigt werden? Ja, siehe später AVL-Bäume.

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

8.3 Optimale Suchbäume

Eine geordnete Folge von Elementen $a_1, a_2, a_3, \dots, a_n$ (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 $(a_1, a_2, a_3, \dots, a_n)$, mit $a_i \leq a_j$ für $i < j$, mit zugehörigen Häufigkeiten oder Wahrscheinlichkeiten $p_1, p_2, p_3, \dots, p_n$ (insbesondere sind alle $p_i \geq 0$) sowie ein binärer Suchbaum B mit n Knoten v_1, \dots, 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 $p_1 + p_2 + p_3 + \dots + p_n = p$.

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 $a_1, a_2, a_3, \dots, a_n$ 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 $(a_1, a_2, a_3, \dots, a_n)$ 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 $a_1, a_2, a_3, \dots, a_n$. Dann gibt es zu jedem binären Baum B mit n Knoten $v_1, v_2, v_3, \dots, v_n$ genau eine Bijektion $f: \{a_1, a_2, a_3, \dots, a_n\} \rightarrow \{v_1, v_2, v_3, \dots, v_n\}$ der Folgelemente a_i zu den Knoten v_j , so dass B hierdurch zu einem Suchbaum wird und der Inorder-Durchlauf von B die sortierte Folge $a_1, a_2, a_3, \dots, a_n$ 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 $a_1, a_2, a_3, \dots, a_n$ 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 $a_1, a_2, a_3, \dots, a_n$ (mit $a_i \leq a_j$ für $i < j$) mit ihren Wahrscheinlichkeiten $p_1, p_2, p_3, \dots, p_n$ ($p_i \geq 0$ und $p_1 + p_2 + \dots + p_n = 1$) oder Häufigkeiten $p_1, p_2, p_3, \dots, p_n$ ($p_i \geq 0$).

Ein Suchbaum B mit n Knoten v_1, \dots, v_n , wobei a_i der Inhalt des Knotens v_i ist, heißt optimaler Suchbaum (zur Folge a_1, a_2, \dots, a_n mit ihren Wahrscheinlichkeiten bzw. Häufigkeiten), wenn für alle zur Folge $a_1, a_2, a_3, \dots, a_n$ 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 $p_1, p_2, p_3, \dots, p_n$ einen optimalen Suchbaum zu konstruieren. Da es C_n mögliche Suchbäume gibt (Satz 8.2.7), dauert das systematische Durchprobieren viel zu lange.

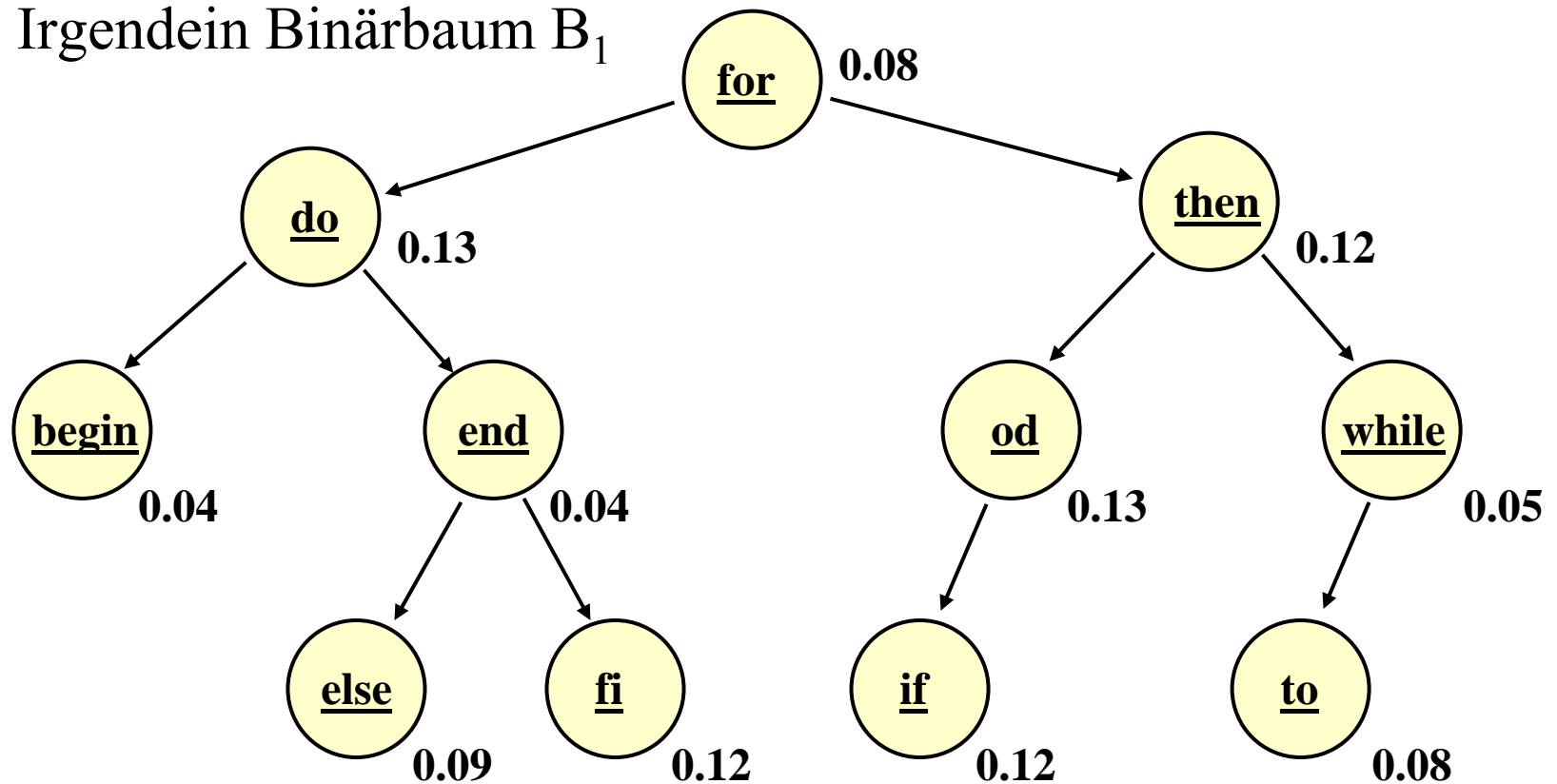
8.3.3 Beispiel

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 Folgeelemente zunächst in einen beliebigen Binärbaum B_1 ein und berechnen dessen gewichtete mittlere Suchdauer $S(B_1)$.

Irgendein Binärbaum B_1

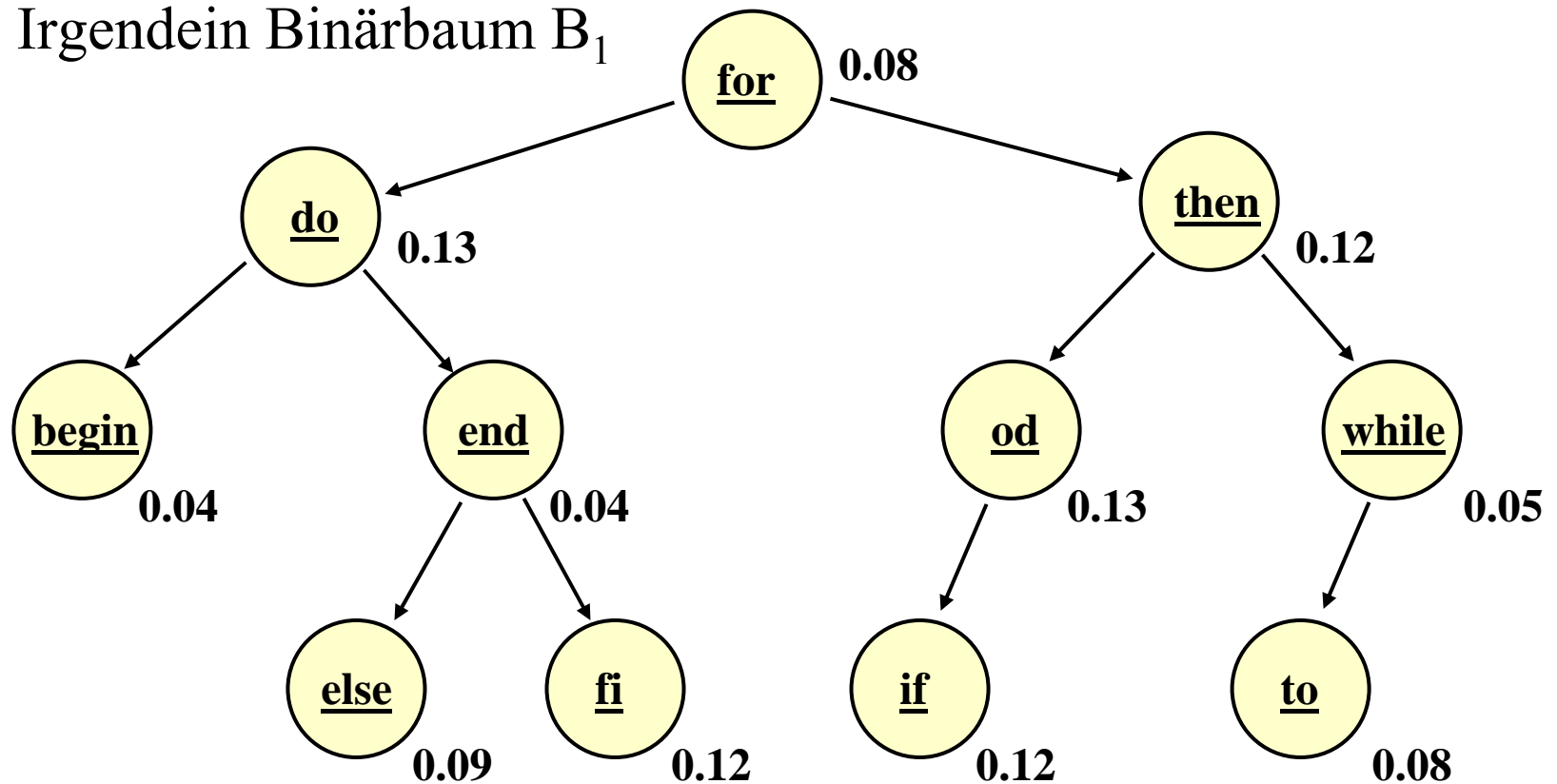


Schlüsselwörter eintragen (inorder-Durchlauf)

Wahrscheinlichkeiten hinzufügen

Gewichtete mittlere Suchdauer nun ausrechnen (bitte selbst durchführen, dann erst zur nächsten Folie klicken).

Irgendein Binärbaum B_1

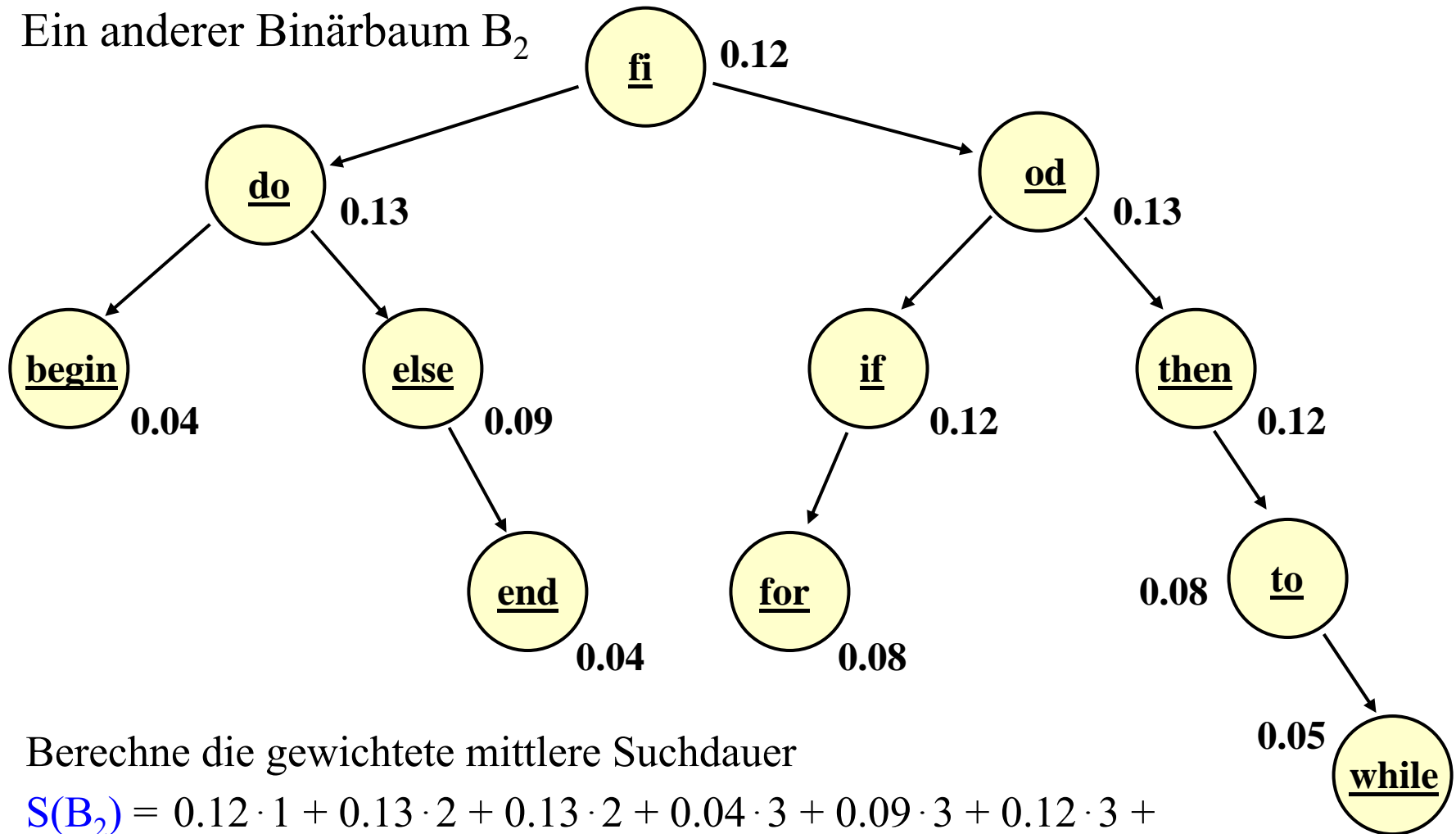


Berechne die gewichtete mittlere Suchdauer

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

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

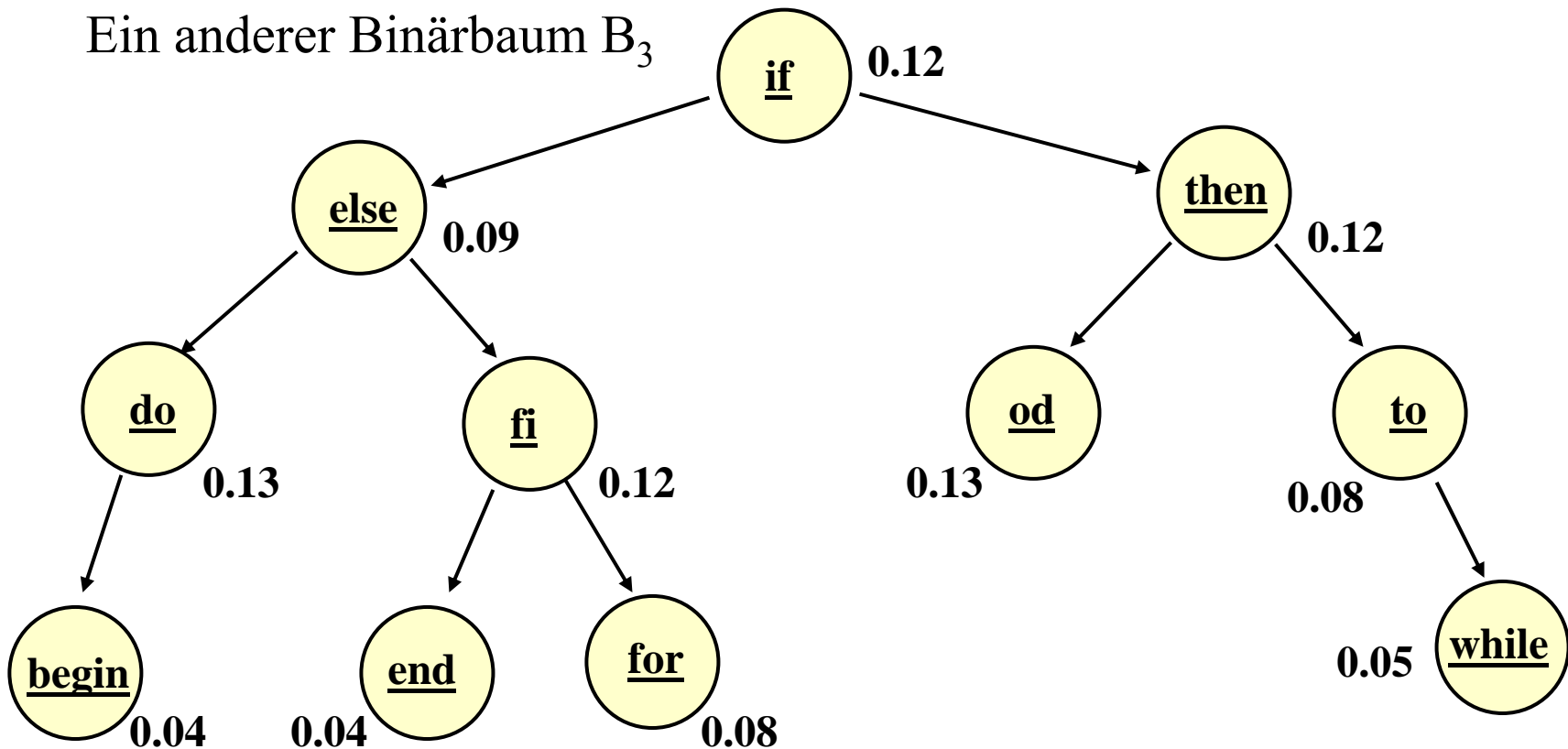


Berechne die gewichtete mittlere Suchdauer

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

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):

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

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

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' .

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

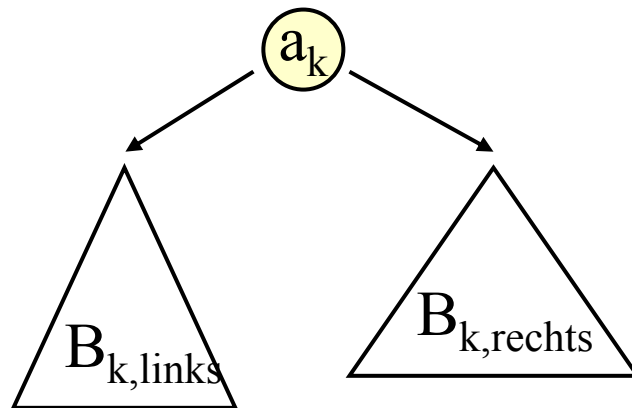
$$> \sum_{a_i \in B'-U'} p_i \cdot \text{level}(v'_i) + S(U') + \sum_{a_i \in U'} p_i \cdot x$$

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

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 $a_i, a_{i+1}, \dots, a_{j-1}, a_j$ zu finden, prüft man alle Paare von Unterbäumen für $a_i, a_{i+1}, \dots, a_{k-1}$ und $a_{k+1}, a_{i+2}, \dots, a_j$ 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 $a_i, a_{i+1}, \dots, a_{j-1}, a_j$ und es ist $i \leq k \leq j$.

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

Es sei $G[i,j] = P[i] + P[i+1] + \dots + P[j]$ (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 $a_i, a_{i+1}, \dots, a_{j-1}, a_j$.

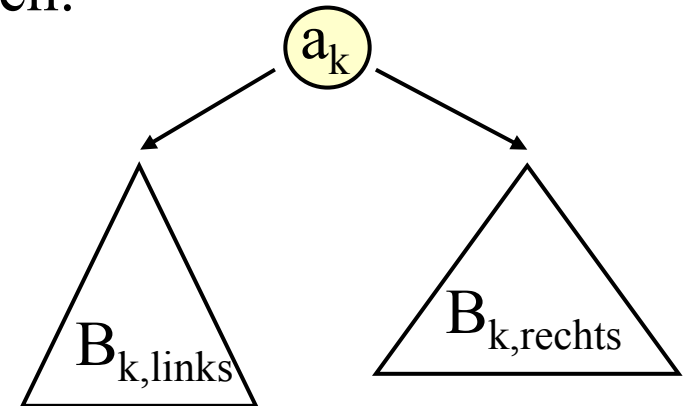
$S[i,j]$ lässt sich leicht rekursiv berechnen:

$S[i,i] = P[i]$ für $i = 1, 2, \dots, n$.

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 $G[i,j]$):

$S[i,j] = \text{Min} \{ S[i,k-1] + S[k+1,j] \mid i \leq k \leq j \} + G[i,j]$ 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 $\text{diff} := j-i$ und berechne für $\text{diff} = 0, 1, 2, \dots, n-1$ alle Werte $S[i, i+\text{diff}]$.

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

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}, \dots, 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 $P[1], \dots, P[n]$.

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) Neueintragen 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).

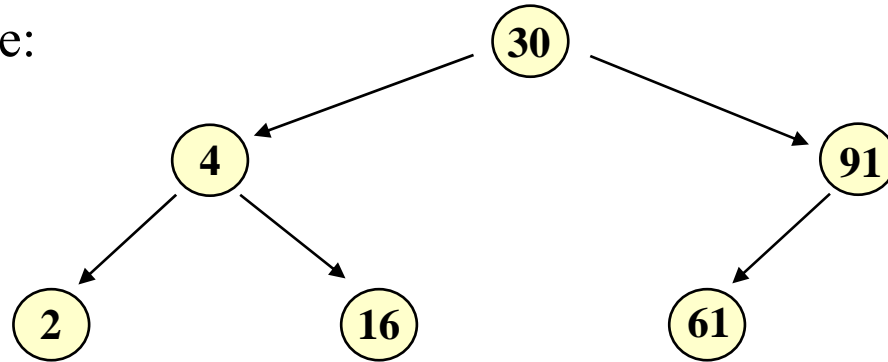
Hinweis 4: Zum Abschluss erläutern wir kurz, wie man dieses Verfahren auf den **Fall der erfolglosen Suche** erweitert.

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

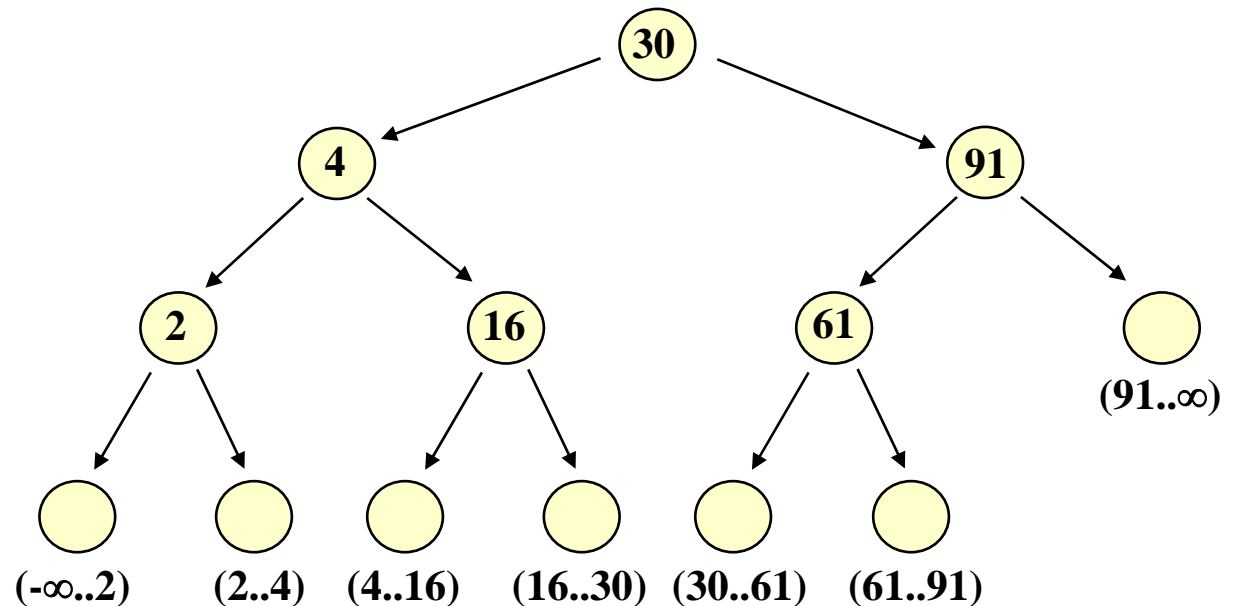
Man ersetzt nun diese null-Zeiger durch Knoten, deren Inhalt alle die Elemente bilden, mit denen man im Suchbaum hierhin gelangt. Ihre Inhalte sind daher offene Intervalle der Form $(i..j) = \{r \mid i < r < j\}$.

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 repräsen-
tieren; diese neuen
Blätter werden mit
den offenen In-
tervallen (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

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

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

$|V_{UB_{\text{links}}}| + |V_{UB_{\text{rechts}}}|$ 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 $(0, \frac{1}{2}]$. Ein Knoten u eines Binärbaums heißt α -gewichtsbalanciert, wenn gilt

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

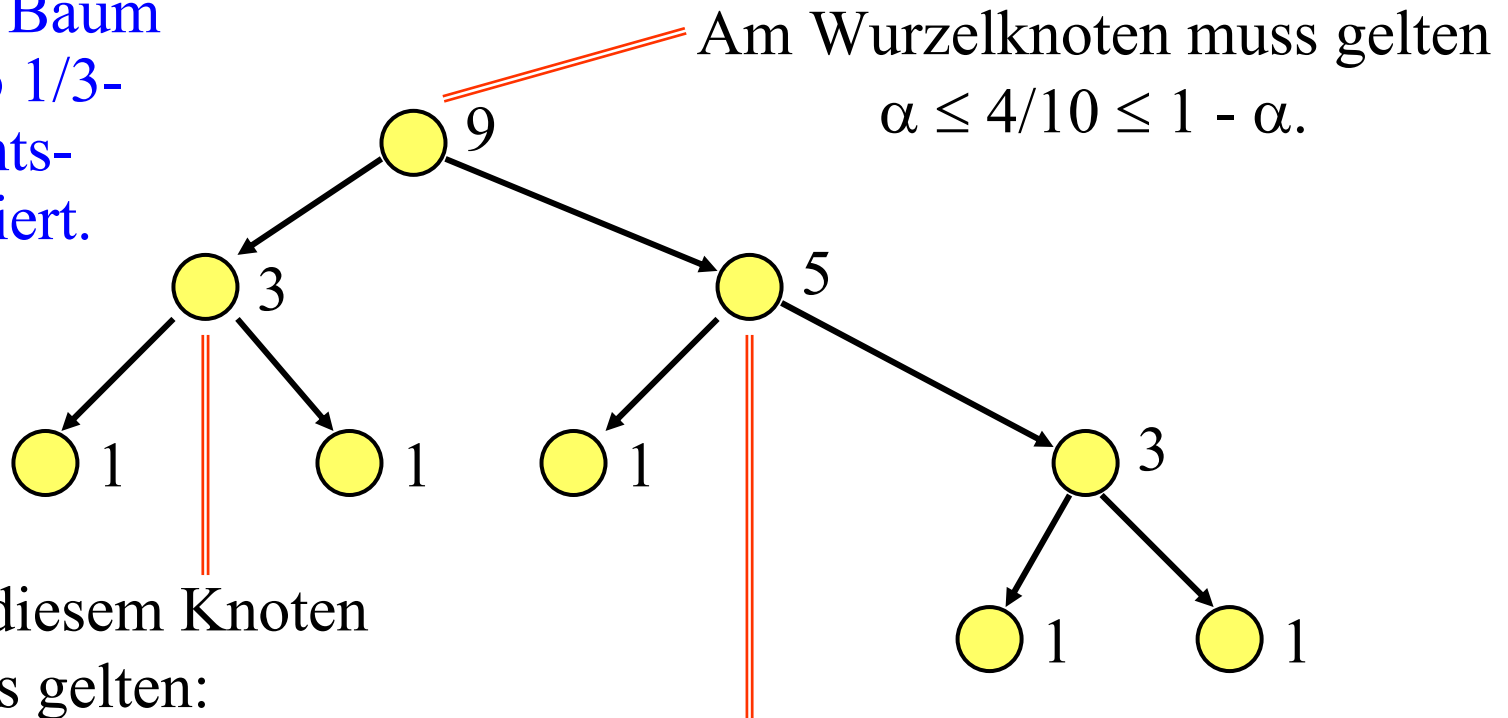
Den Ausdruck $\beta(u) = \frac{|V_{UB_{links}}| + 1}{|V_{UB_{links}}| + |V_{UB_{rechts}}| + 2}$

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 $0 \leq \alpha' \leq \alpha \leq \frac{1}{2}$ 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.



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

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

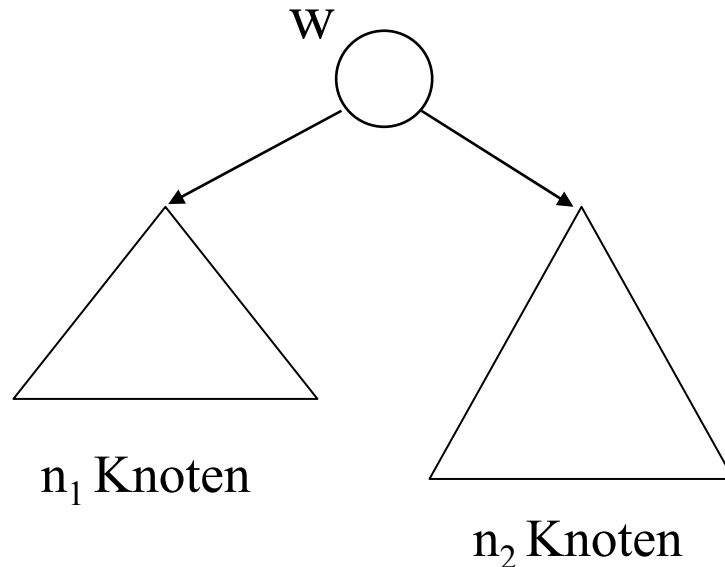
gilt, dann gilt auch

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

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

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

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 $i = 1, 2$:

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

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:

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

("Höhe" des rechten Unterbaums von u -

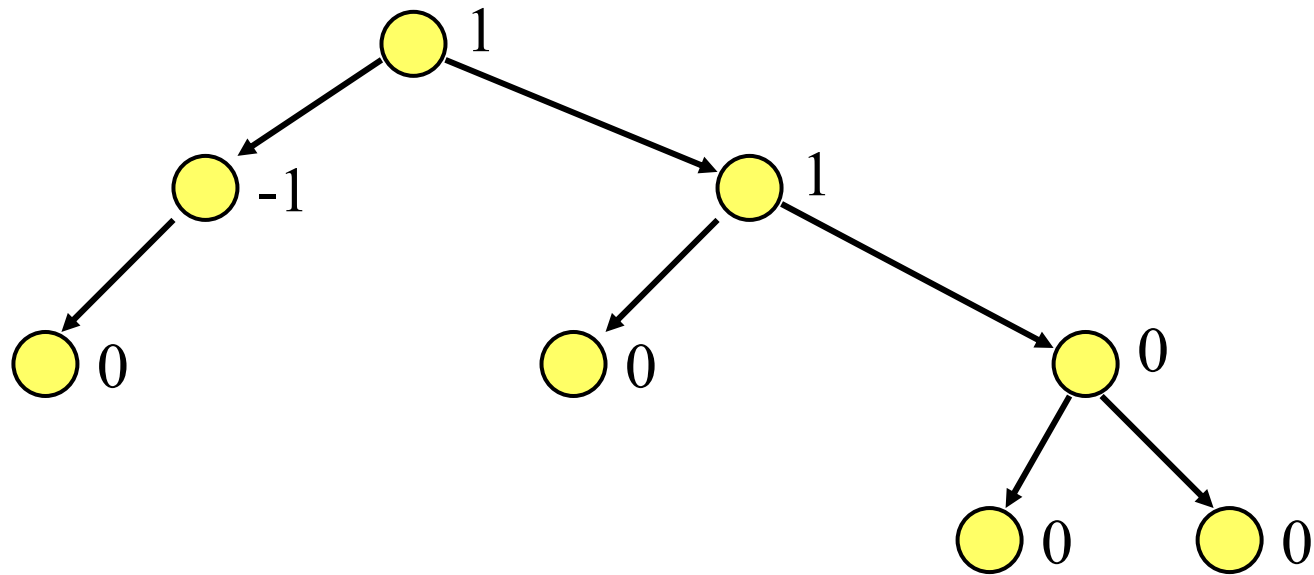
"Höhe" des linken Unterbaums von u) $\in \{-1, 0, 1\}$,

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

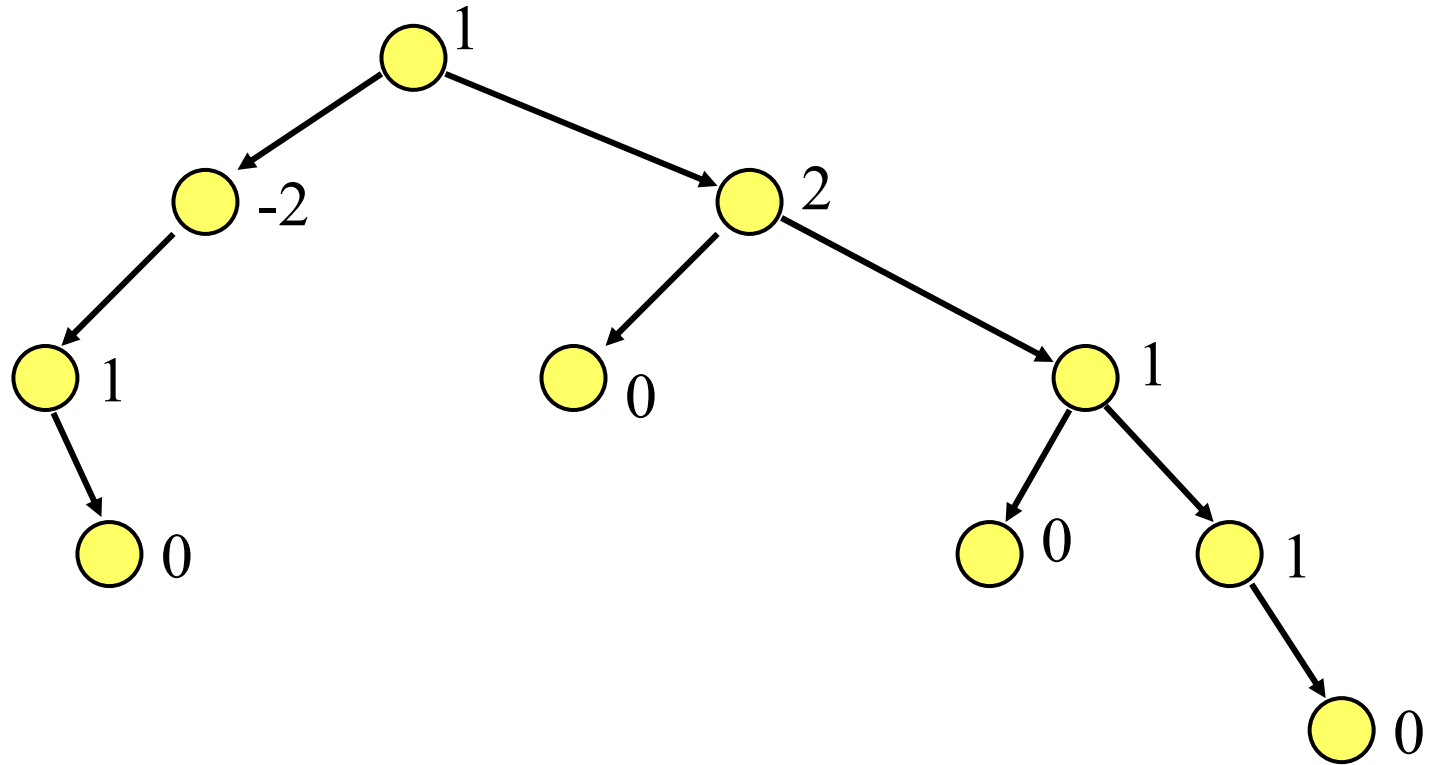
$T(UB_{\text{rechts}}) - T(UB_{\text{links}})$ heißt die **Höhenbalance** von u, oft auch **Balancefaktor** oder kurz **Balance** von u genannt.

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

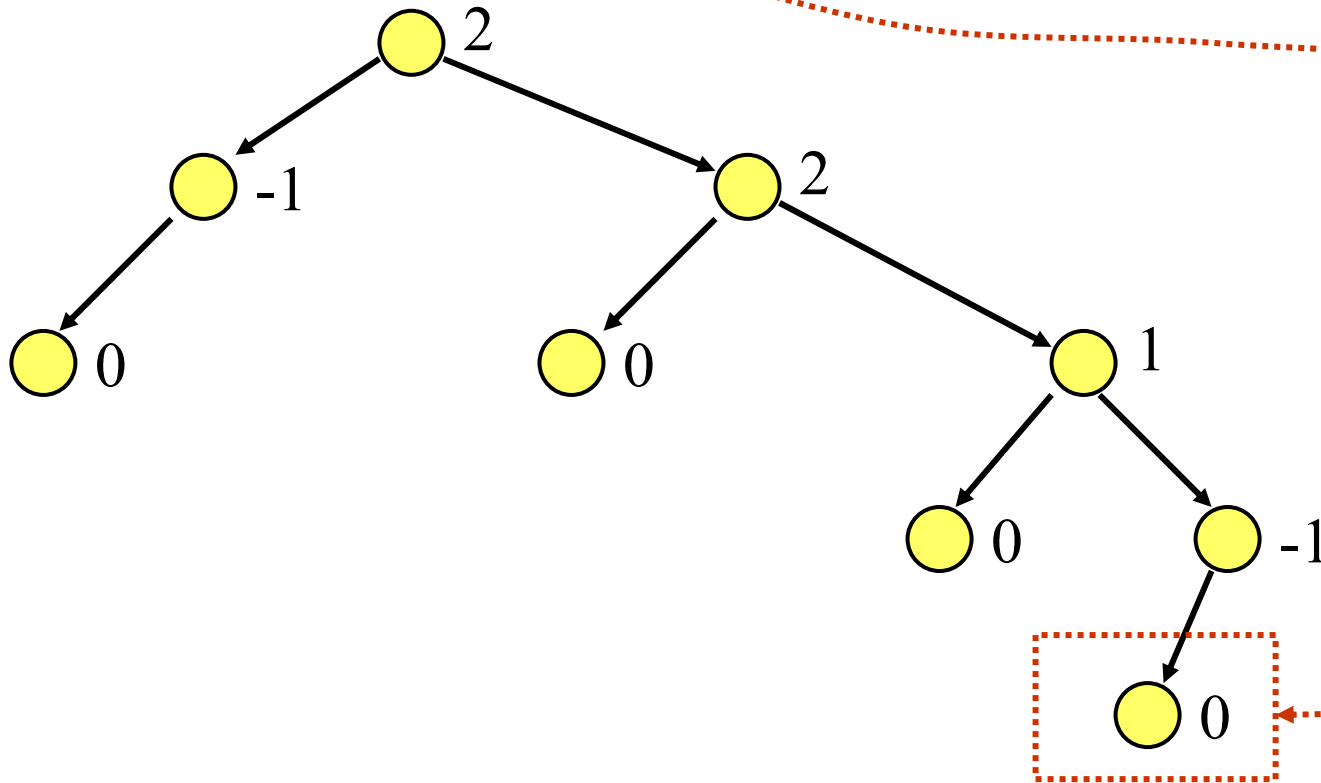
Beispiel: Der bereits im Beispiel 8.4.2 betrachtete Baum ist höhenbalanciert, d.h. ein AVL-Baum. Wir tragen die Höhenbalancen neben jedem Knoten ein:



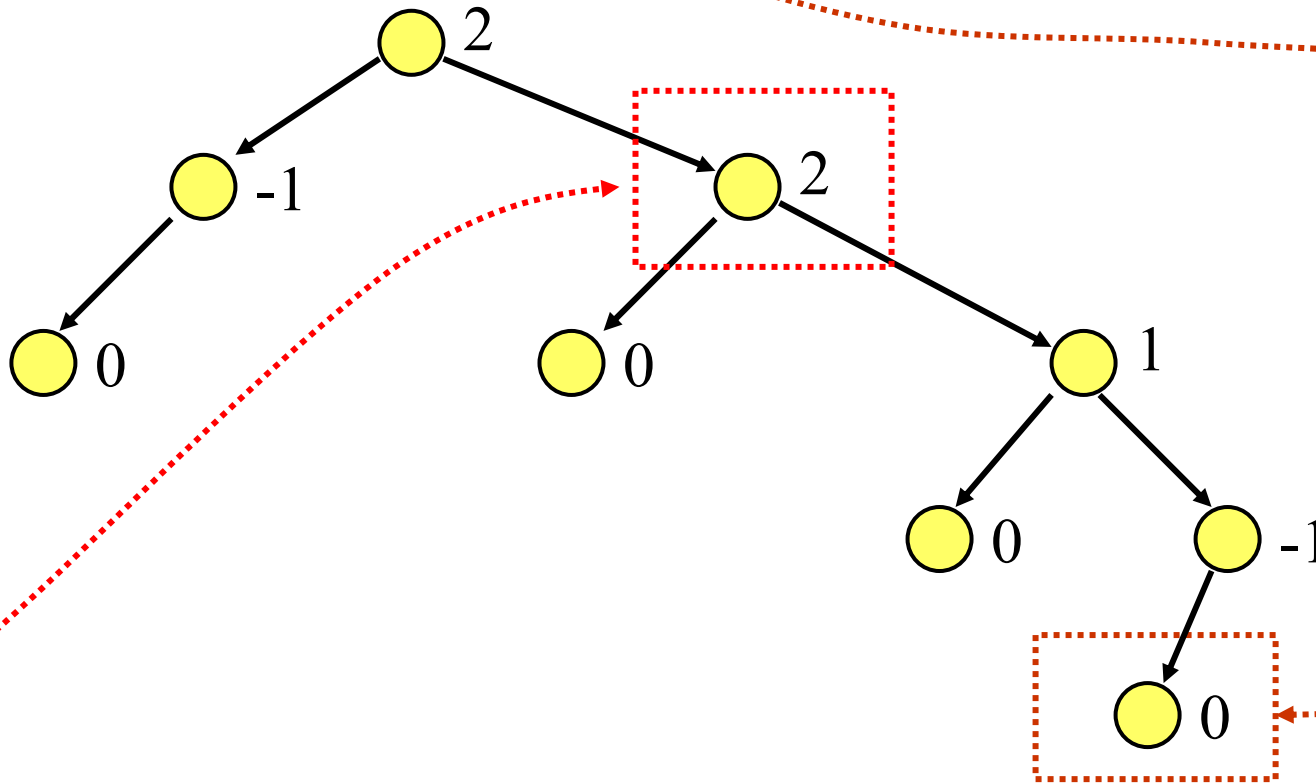
Beispiel: Dagegen ist folgender Baum *nicht* höhenbalanciert:



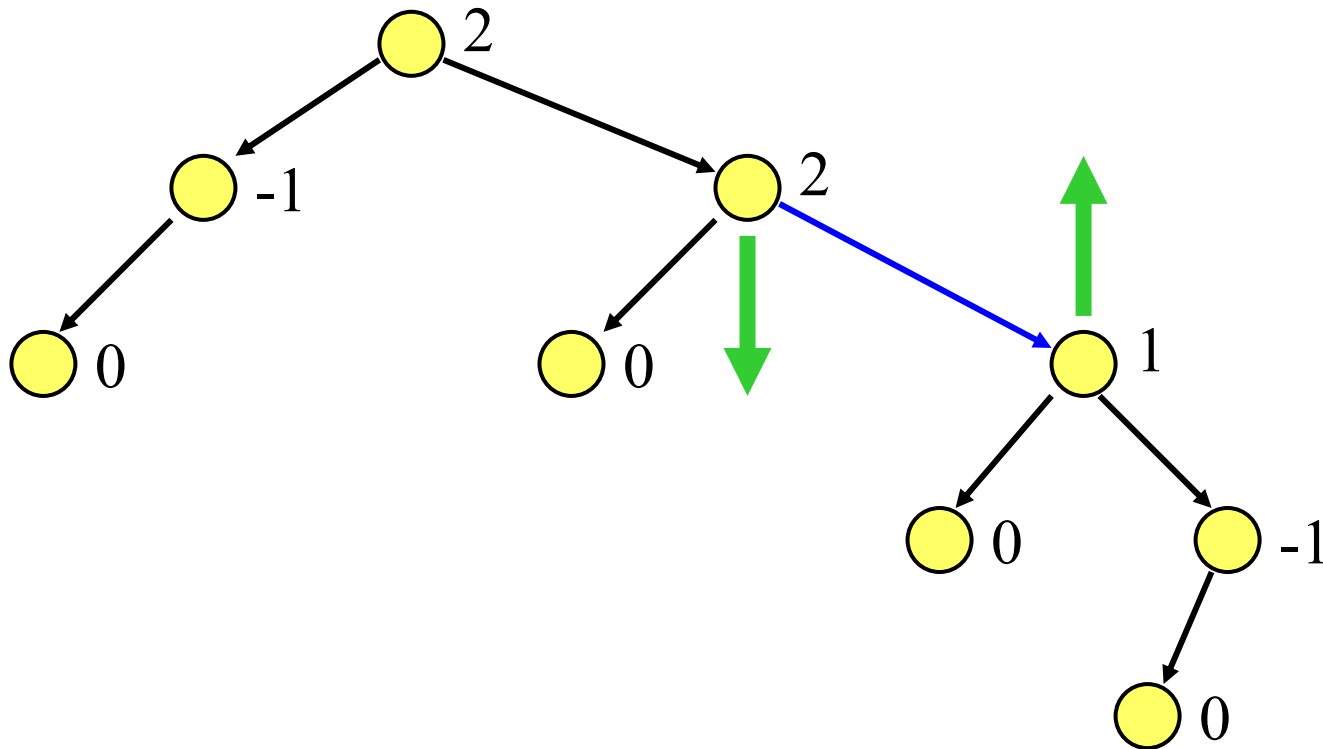
Betrachte nun einen AVL-Baum, in dem nur wegen eines Knotens (hier: unten rechts) die AVL-Eigenschaft verletzt ist:



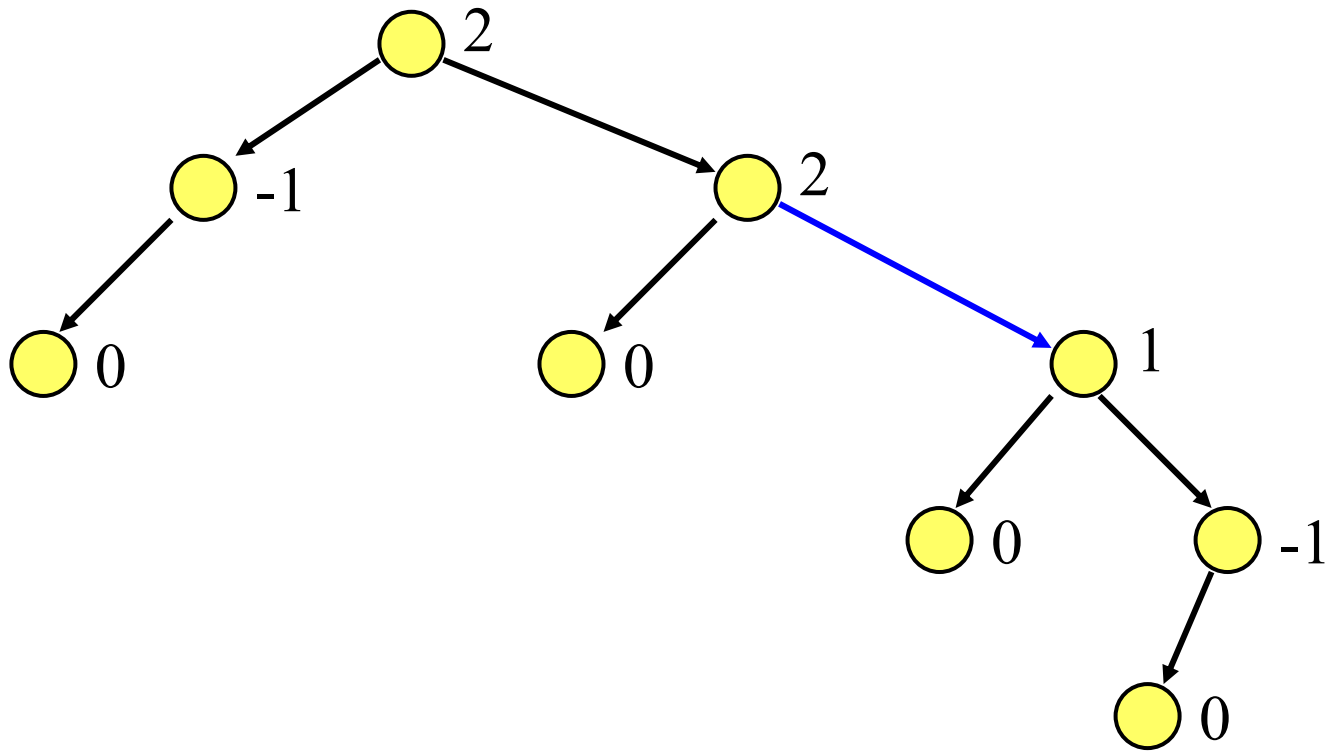
Betrachte nun einen AVL-Baum, in dem nur wegen eines Knotens (hier: unten rechts) die AVL-Eigenschaft verletzt ist:



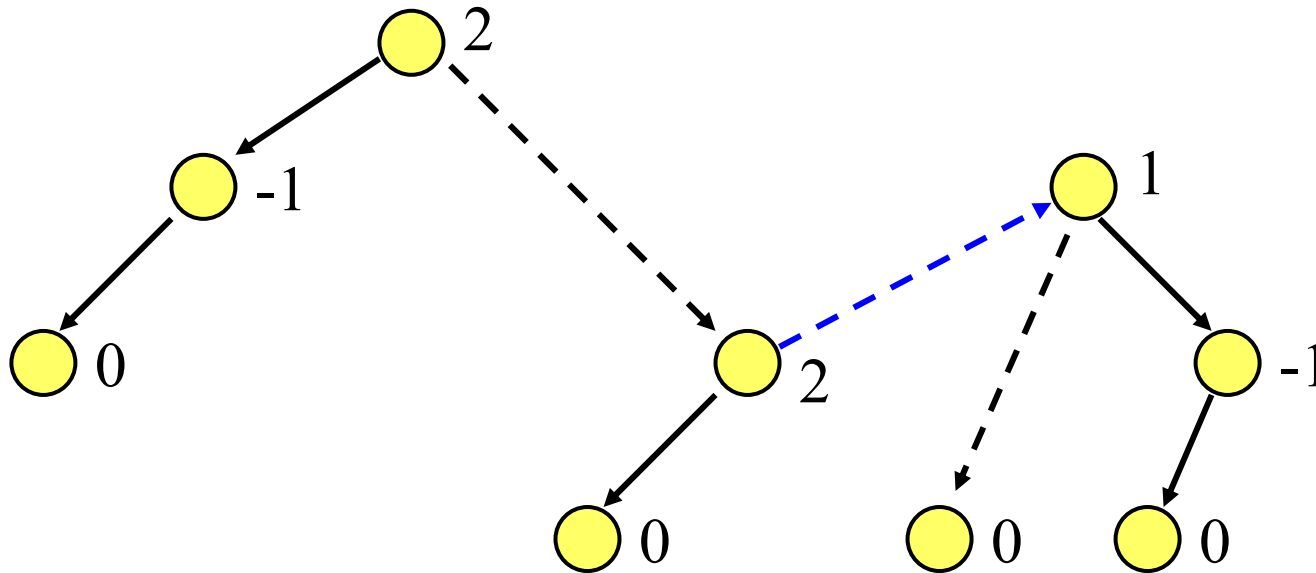
Dann kann man durch eine leichte Korrektur an der untersten Verletzungs-Position die AVL-Eigenschaft wieder herstellen:



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

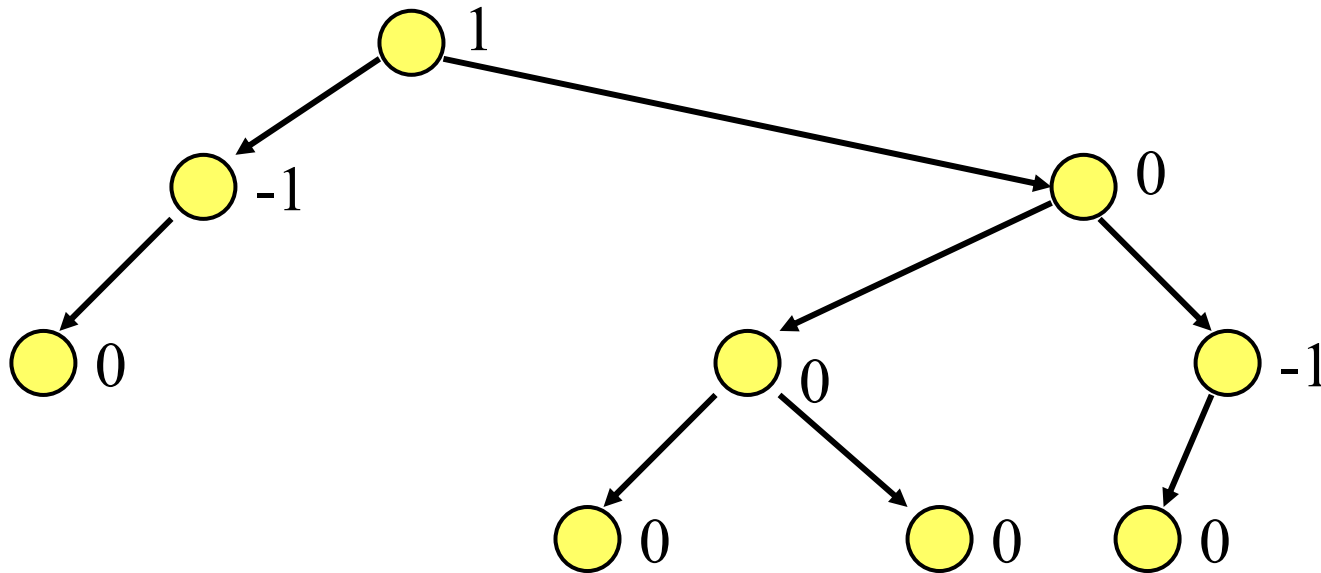


Nur die drei gestrichelten Kanten werden verzerrt.



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

und schon liegt wieder ein AVL-Baum vor:



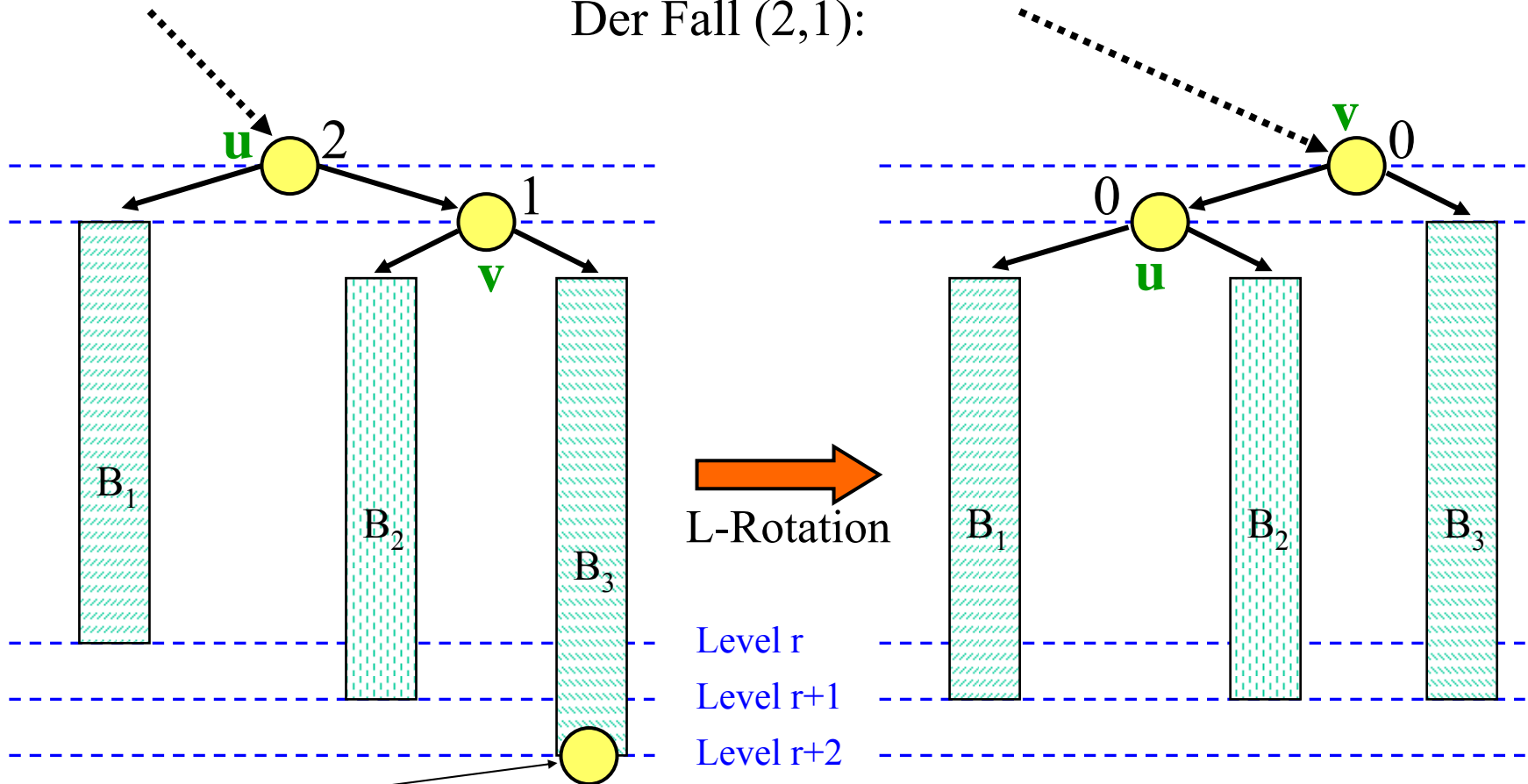
Durch Ausprobieren stellt man fest, dass man mit vier Rotationsarten auskommt, um alle solche Fälle zu korrigieren.

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):

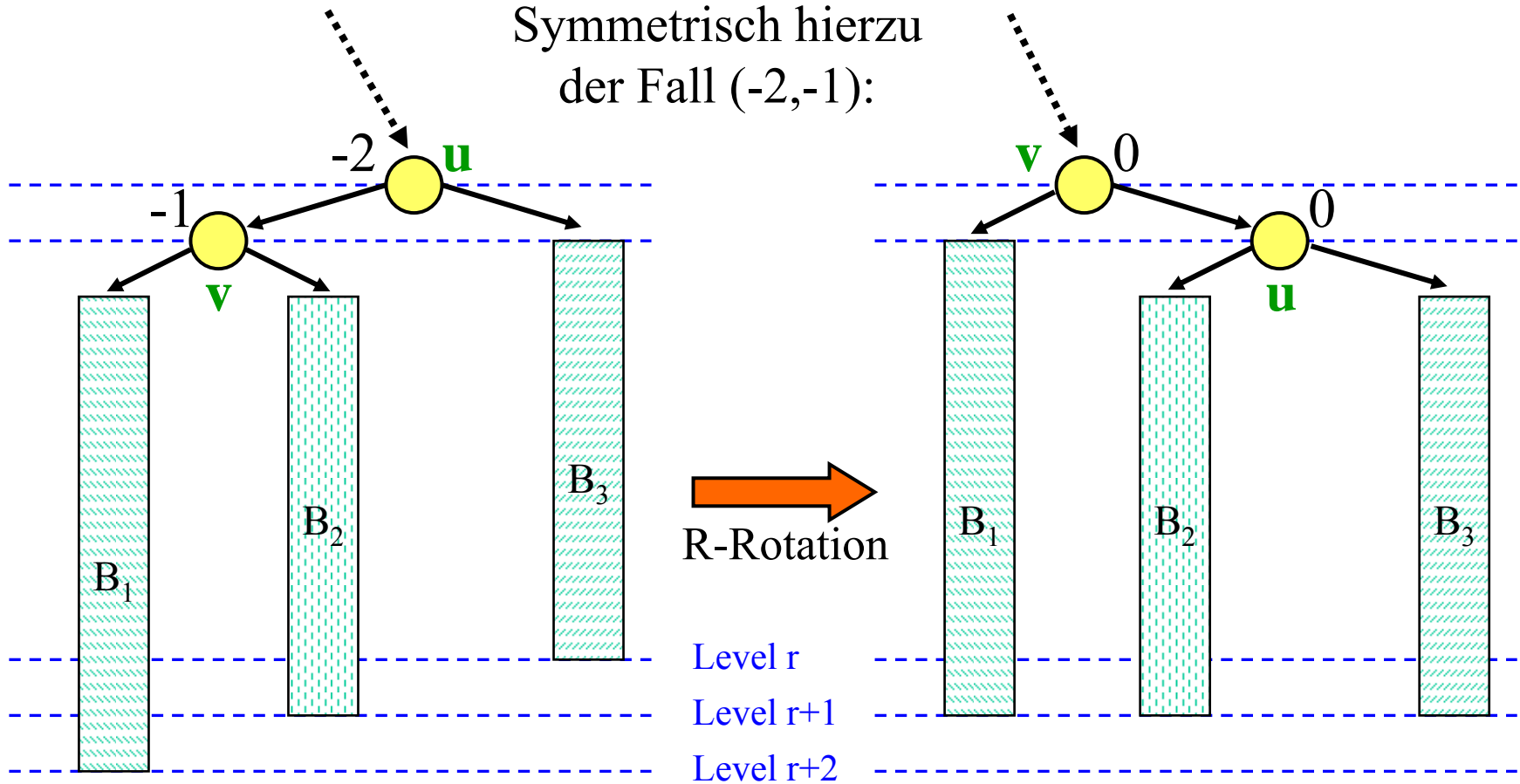


Dieses Blatt wurde eingefügt, wodurch die neuen Balancen 2 und 1 bei u und v entstanden.

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

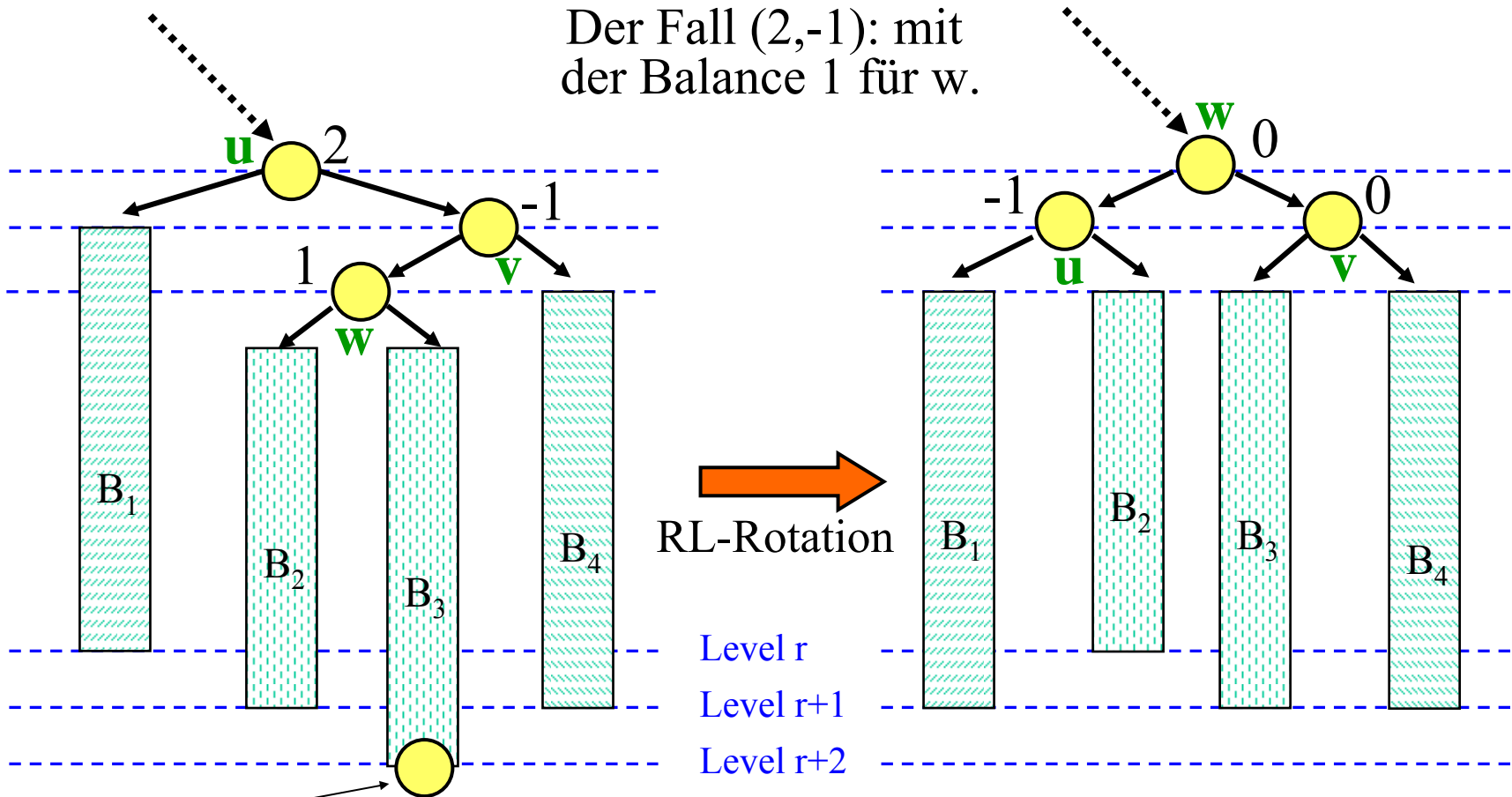
⇒ Links-Rotation durchführen (3 Zeiger umhängen, Balancen 0 und 0).

Symmetrisch hierzu
der Fall $(-2, -1)$:



Situation an der untersten Verletzungsstelle $\mathbf{u} \text{ — } \mathbf{v}$: Balancen -2 und -1 .
 \Rightarrow Rechts-Rotation durchführen (3 Zeiger umhängen, Balancen 0 und 0).

Der Fall (2,-1): mit der Balance 1 für w.

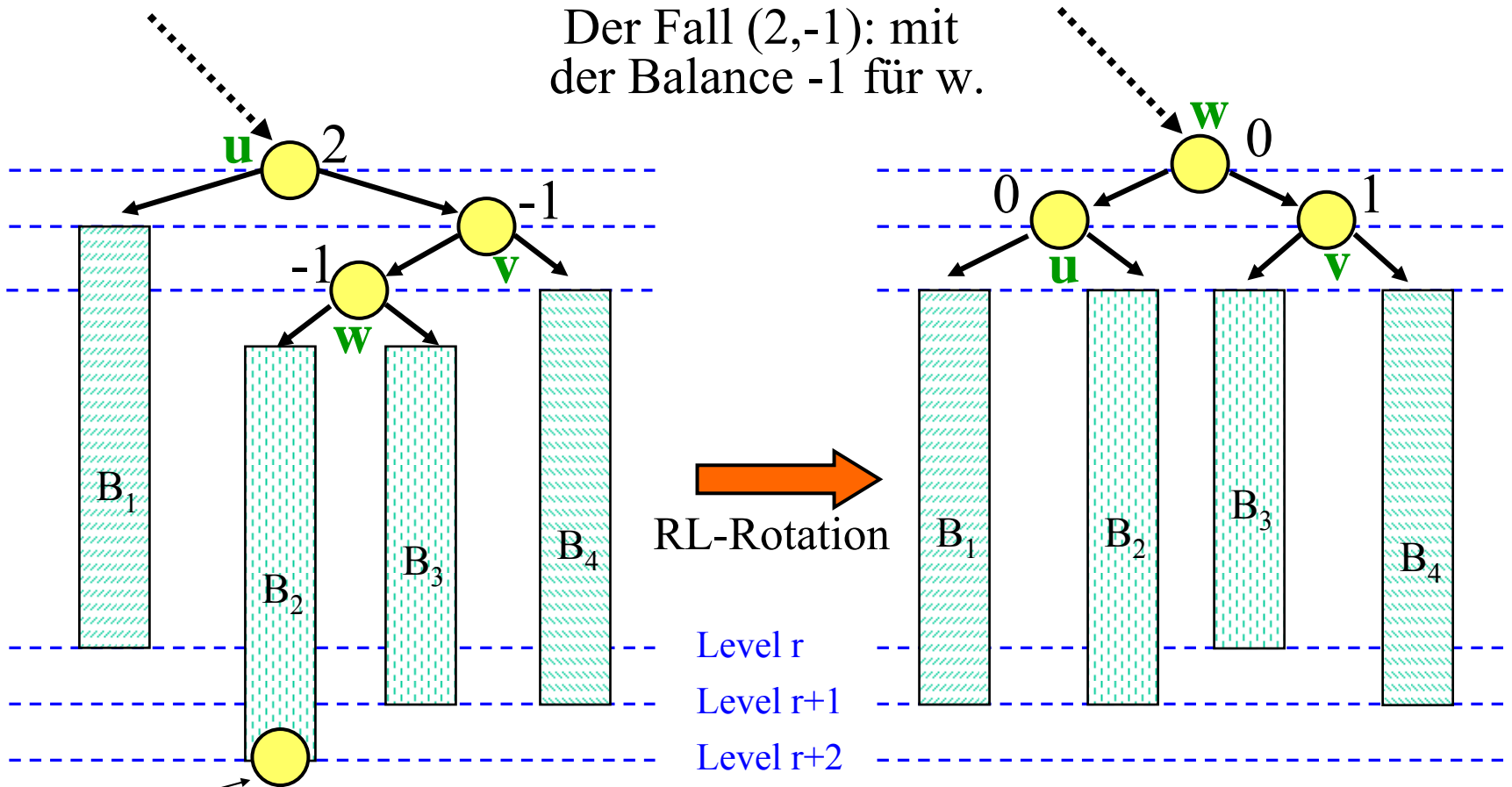


Dieses Blatt wurde eingefügt, wodurch die neuen Balancen 2 und -1 bei u und v entstanden.

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

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

Der Fall (2,-1): mit
der Balance -1 für w.



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.
 \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 $B(\dots)$ neu berechnet? Dieser Algorithmus wird auf der folgenden Folie vorgestellt.

```

Füge ein Blatt v mit Inhalt s und Höhenbalance  $B(\mathbf{v})=0$  ein;
u := direkter Vorgängerknoten von v; Abbruch:=false;
while u ≠ null and not Abbruch loop
    if v ist linkes Kind von u then
        if  $B(\mathbf{u}) = 1$  then  $B(\mathbf{u}) := 0$ ; Abbruch:=true;
        elsif  $B(\mathbf{u}) = 0$  then  $B(\mathbf{u}) := -1$ ;
        else “führe R/LR-Rotation durch;” Abbruch:=true; end if;
    else          -- v ist rechtes Kind von u
        if  $B(\mathbf{u}) = -1$  then  $B(\mathbf{u}) := 0$ ; Abbruch:=true;
        elsif  $B(\mathbf{u}) = 0$  then  $B(\mathbf{u}) := 1$ ;
        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"):

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

Die Lösung der Gleichung lautet: $m_t = F_{t+2} - 1$, wobei F_t die t -te Fibonaccizahl 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, ...).

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 $F_k = f(c_1^k - c_2^k)$ erhält. Wegen $|c_2| < 1$ ist F_k stets die nächste natürliche Zahl zu $f c_1^k$.

Einschub: [Fibonaccizahlen](#)

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

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

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

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

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.

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

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

$$\begin{aligned} t &\leq \log((n+1)/1.17082) / \log(1.618035) \\ &\approx 1.4404 \cdot \log(n+1) - \log(1.17082) / \log(1.618035) \\ &\approx 1.4404 \cdot \log(n) \text{ [Diese Abschätzung ist sehr genau.]} \end{aligned}$$

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

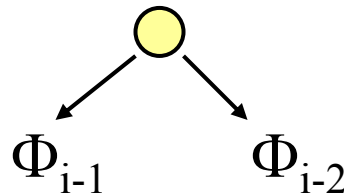
Hinweis: Messungen ergaben, dass diese maximale Tiefe in der Praxis fast nie auftritt und sich die Tiefe der AVL-Bäume meist nur sehr wenig von $\log(n)$ unterscheidet. Aber: Der Fall der Tiefe $\approx 1.4404 \cdot \log(n)$ kann auftreten, siehe im Folgenden.

8.4.9 Fibonacci-Bäume

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

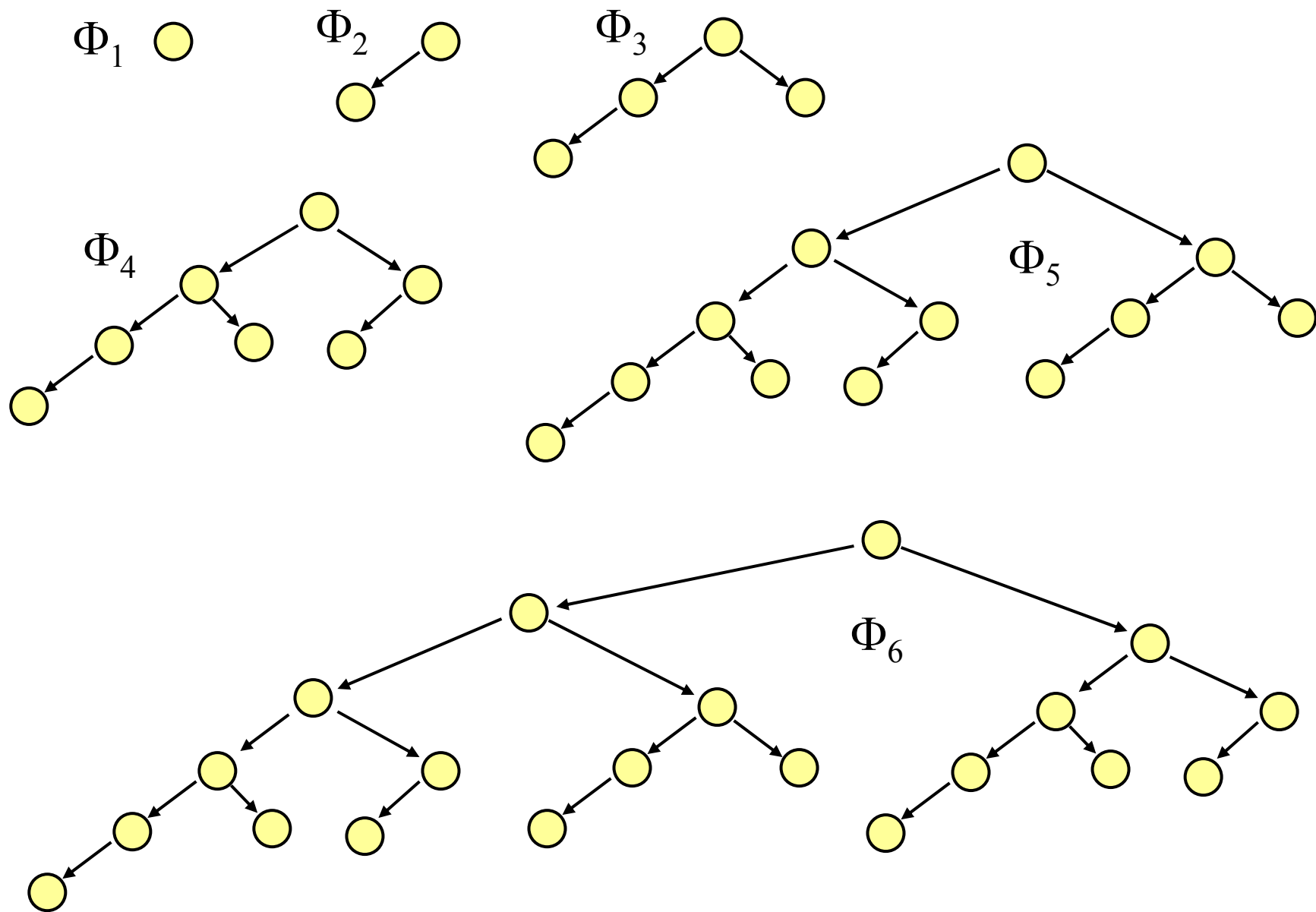
Definition:

- (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



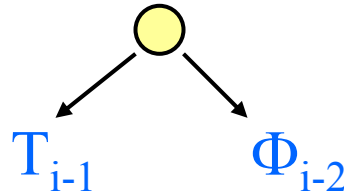
der Fibonaccibaum Φ_i der Ordnung i .

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.



8.4.10 Ist jeder AVL-Baum zugleich gewichtsbalanciert?

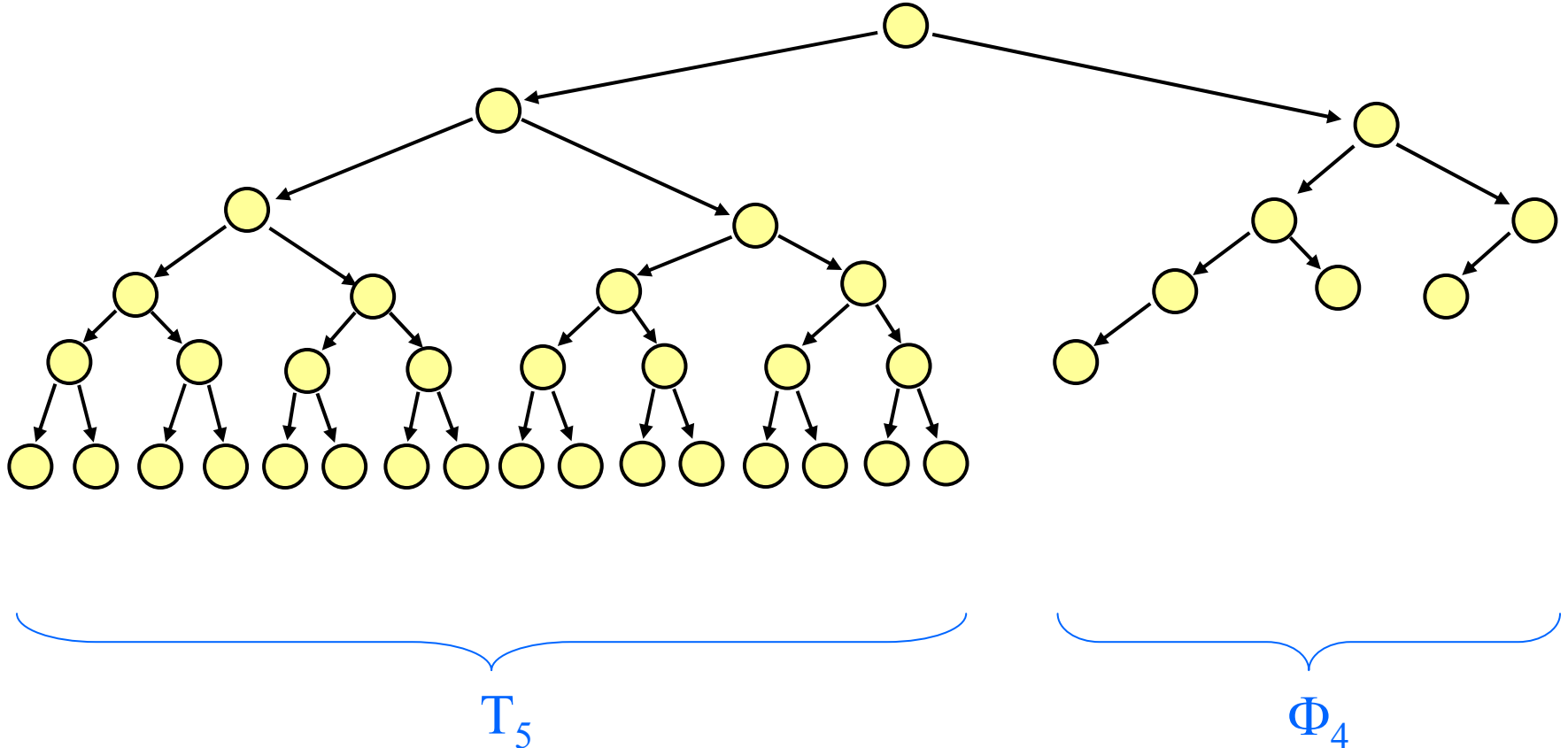
Nein! Betrachte hierzu folgenden AVL-Baum der Tiefe i :



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 $\alpha \in (0, \frac{1}{2}]$ 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 $n = 39 = 1 + 2^5 - 1 + 8 - 1$) 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:

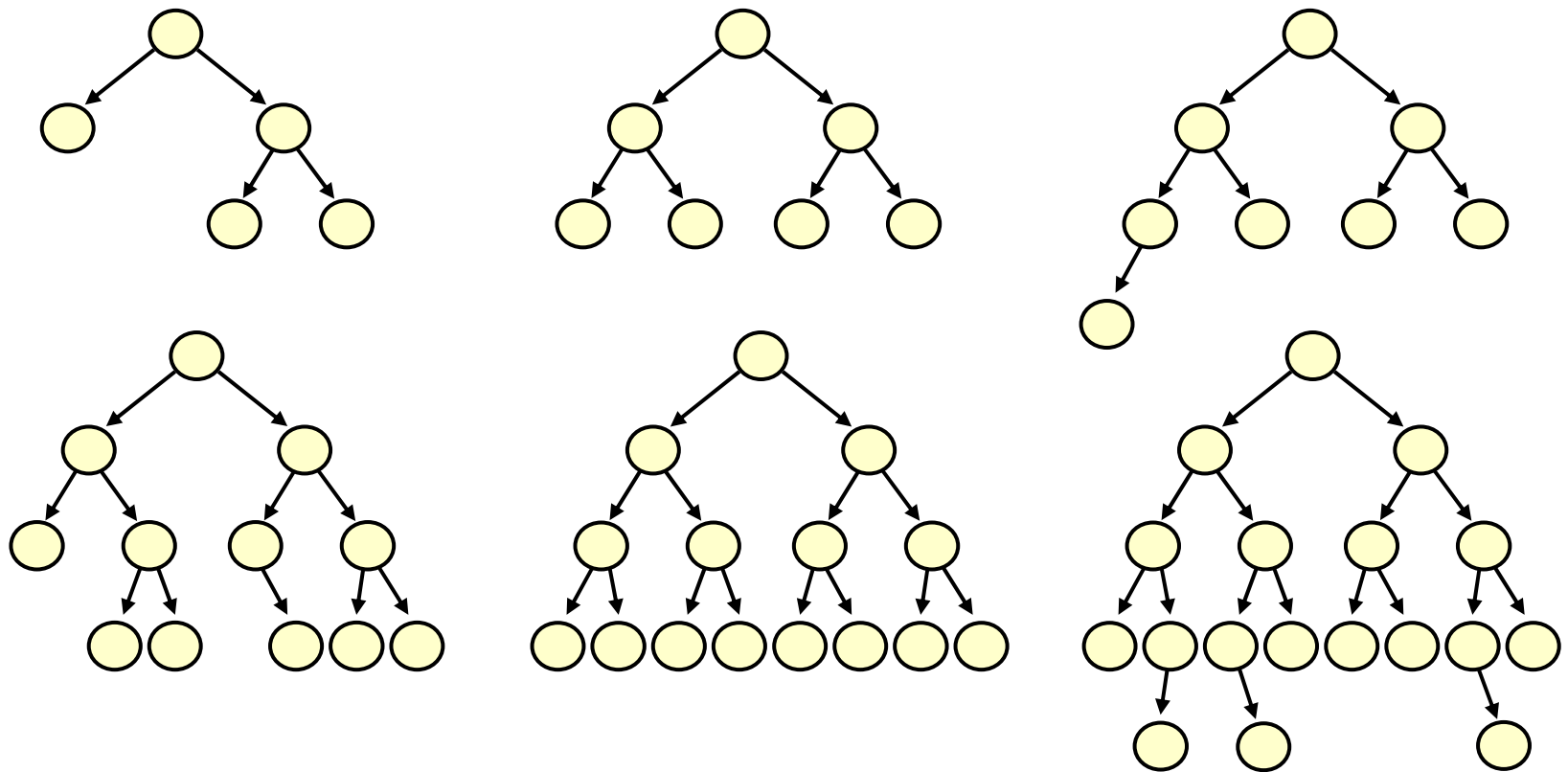
Definition 8.4.11:

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.

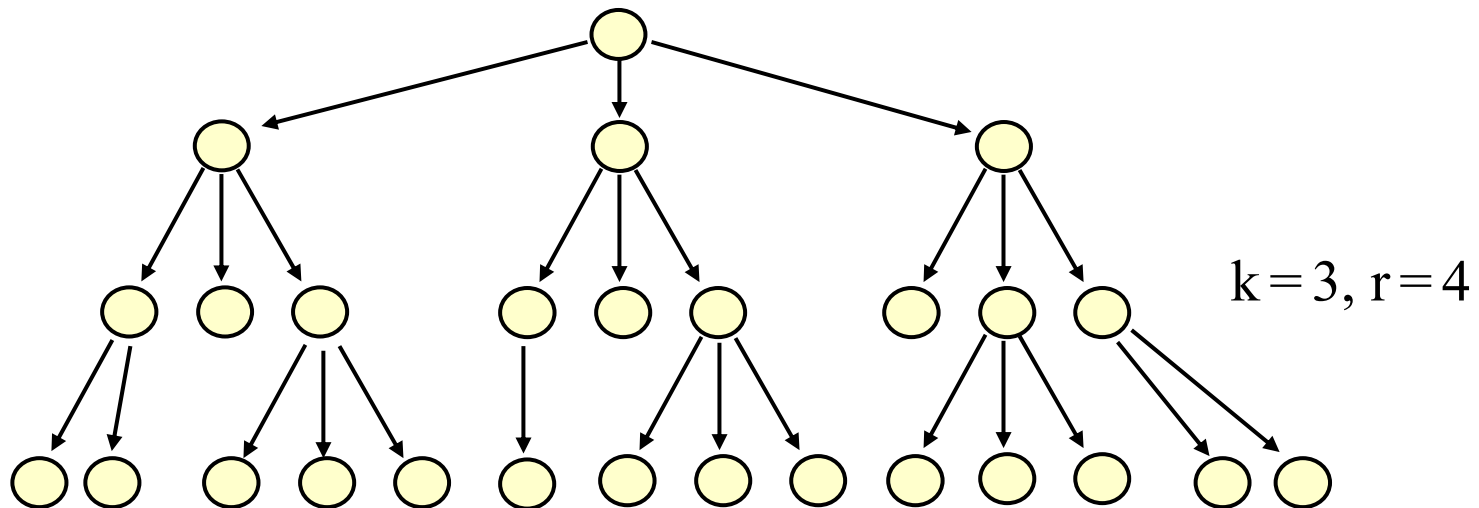
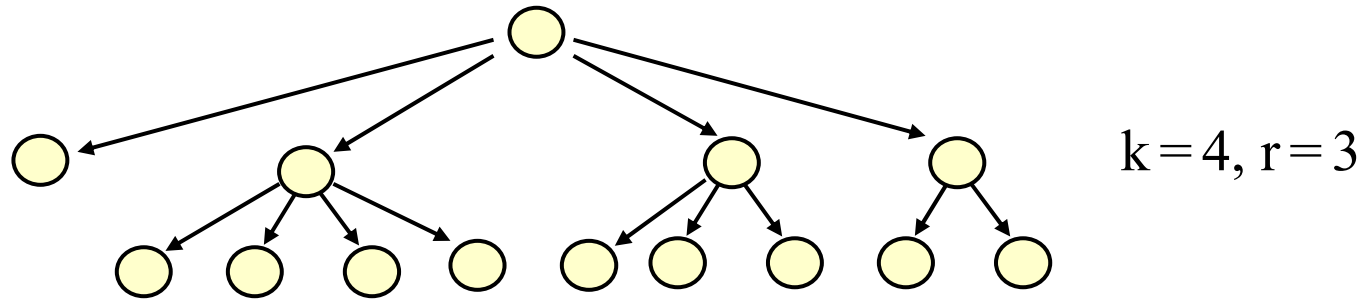
Man kann diese Definition leicht von k -nären auf beliebige geordnete Bäume übertragen. Die B-Bäume des nächsten Abschnitts 8.5 bilden solch ein Beispiel.

Sofern es Knoten mit null-Zeigern in verschiedenen Levels 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):



8.5 B-Bäume (für externe Speicherung)

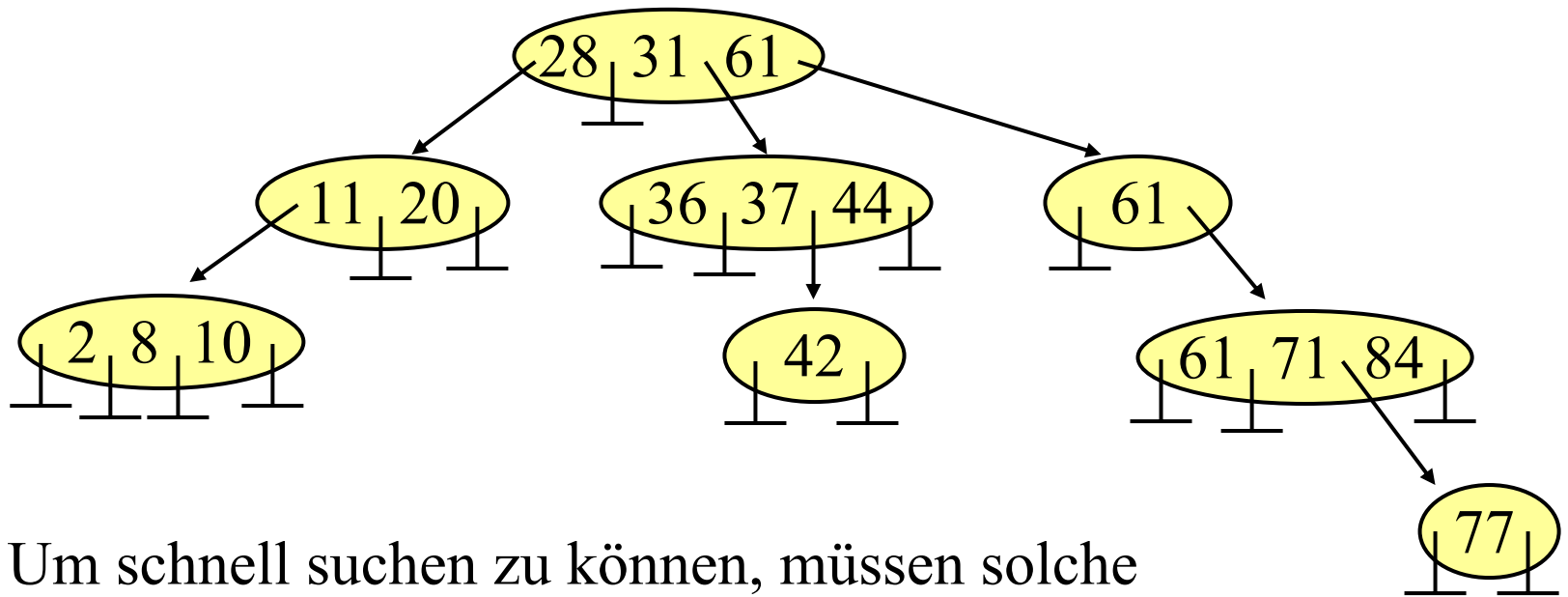
AVL-Bäume eignen sich gut als Darstellung von Mengen, wenn sich die gesamte Menge im Hauptspeicher befindet. Liegen die Elemente dagegen auf einem Hintergrundspeicher, so muss man bei jedem Übergang zu einem Kindknoten auf diesen externen Speicher zugreifen. Heute ist der Hintergrundspeicher meist eine 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, \dots, s_k die sortierten Schlüssel von u ($s_1 \leq s_2 \leq \dots \leq s_k$), 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 $i = 2, 3, \dots, k$), 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 Löschooperationen gleiche Schlüssel auch in den linken Unterbaum verschoben werden. Wir werden diesen Fall daher verbieten und ihn am Ende des Abschnitts gesondert betrachten.

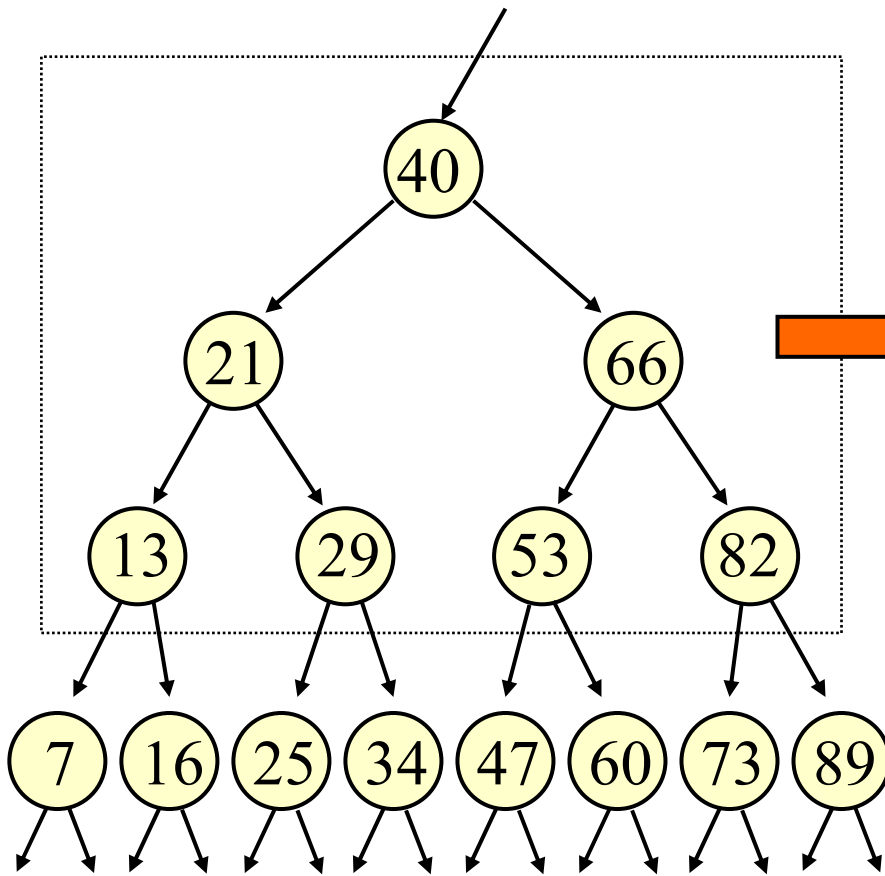
In Anwendungsfällen informiere man sich zuvor genau, ob dort gleiche Schlüssel zugelassen sind und wie sie behandelt werden.

Definition 8.5.2:

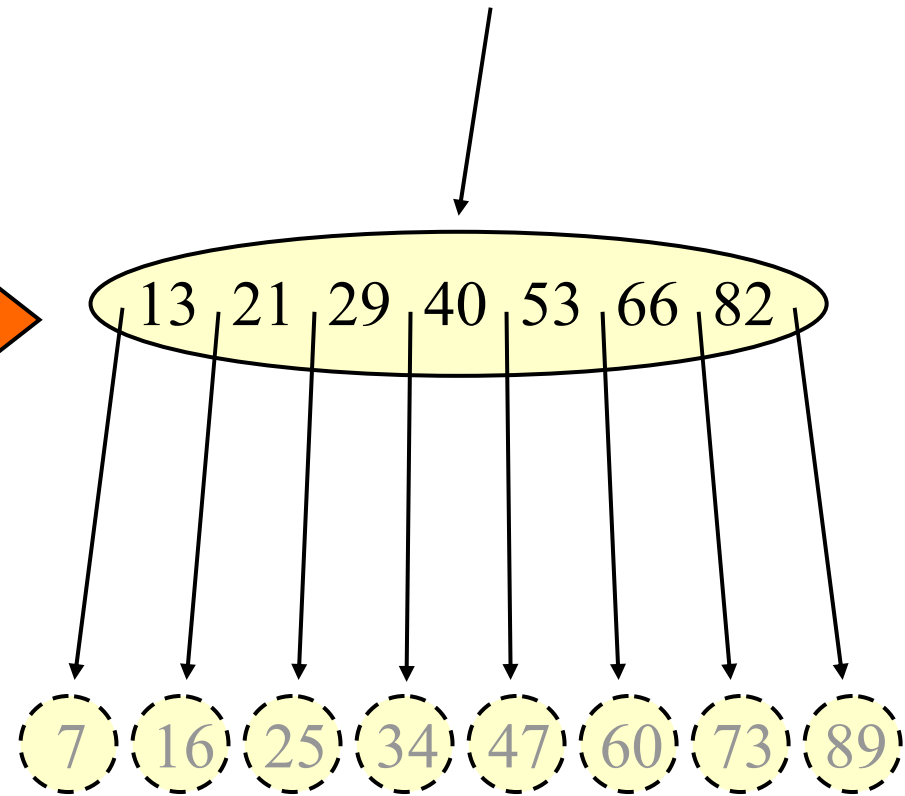
Es sei $m \geq 1$ eine natürliche Zahl. Ein geordneter nicht-leerer Baum, in dem jeder Knoten eine sortierte Folge von Schlüsseln aus einer geordneten Menge enthält, heißt **B-Baum der Ordnung m** , wenn für alle Knoten gilt:

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!



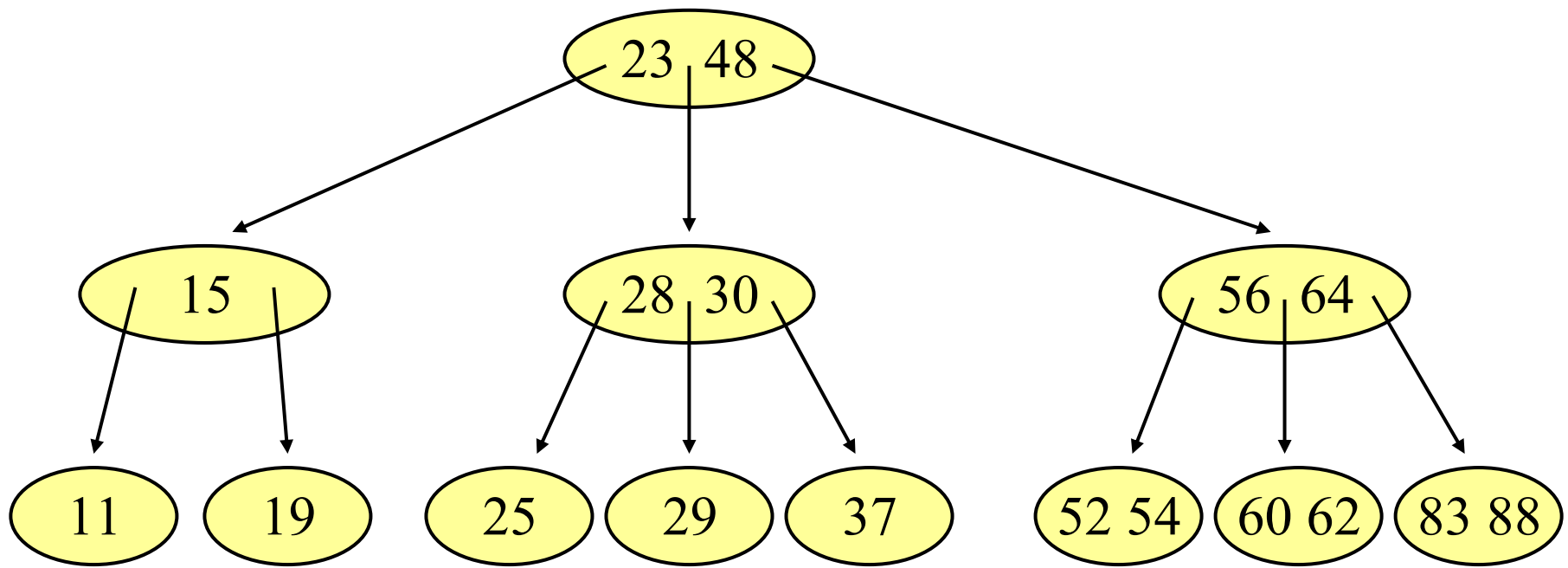
Ausschnitt aus einem
binären Suchbaum.



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

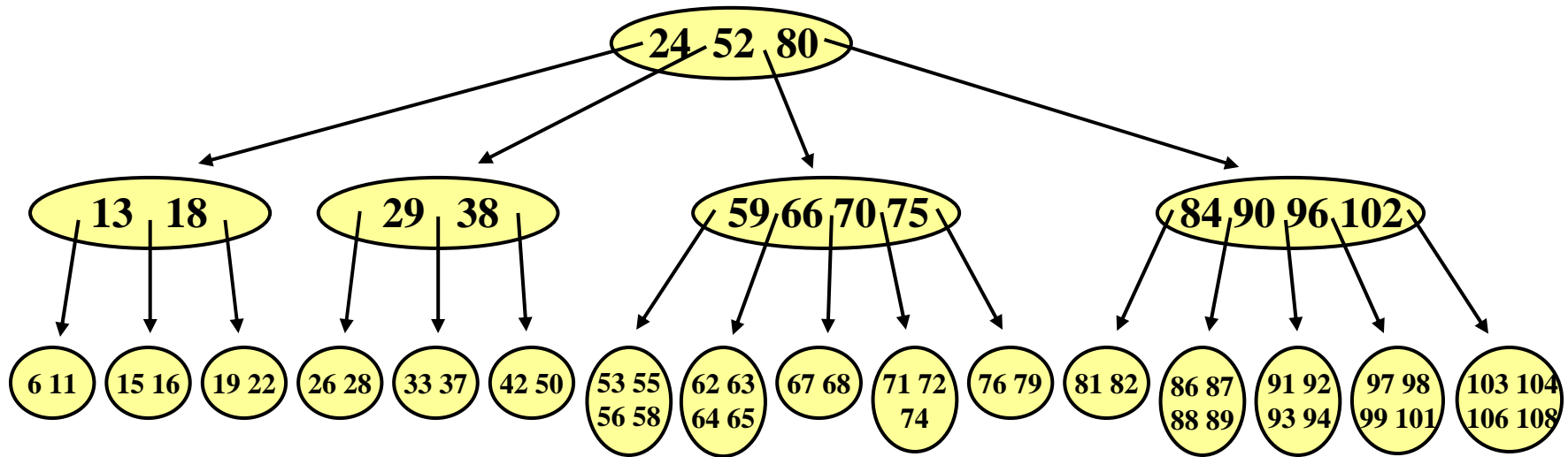
Beispiel für einen B-Baum der Ordnung 1

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**.

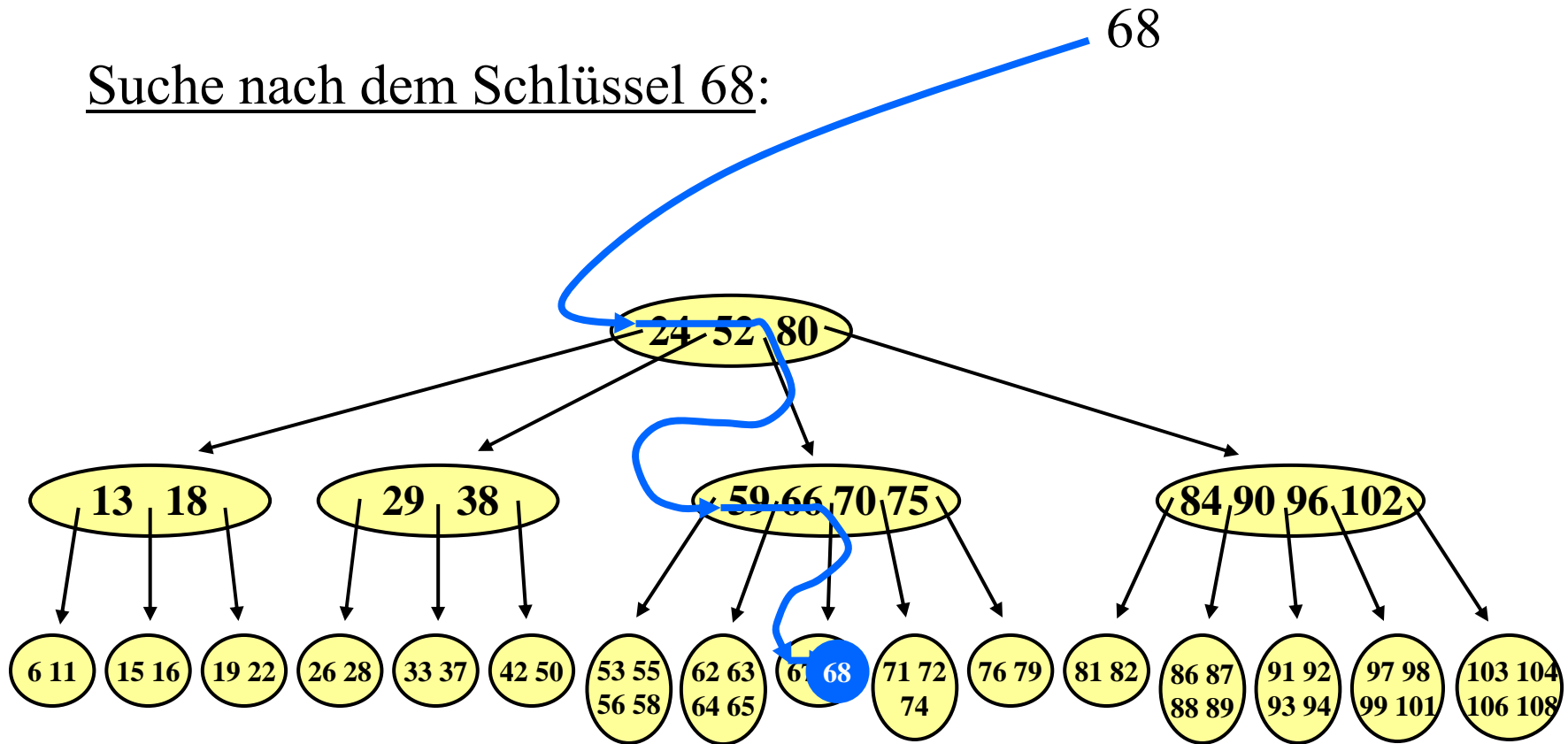
(Solche Bäume können auch Effizienzvorteile haben, siehe Analyse von Search2 in 6.5.1.)

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 $\leq 2m \cdot \text{Tiefe des Baumes} \leq 2m \cdot \log_{m+1}(n)$.

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 $3 \cdot \log(2m) = 27$.)

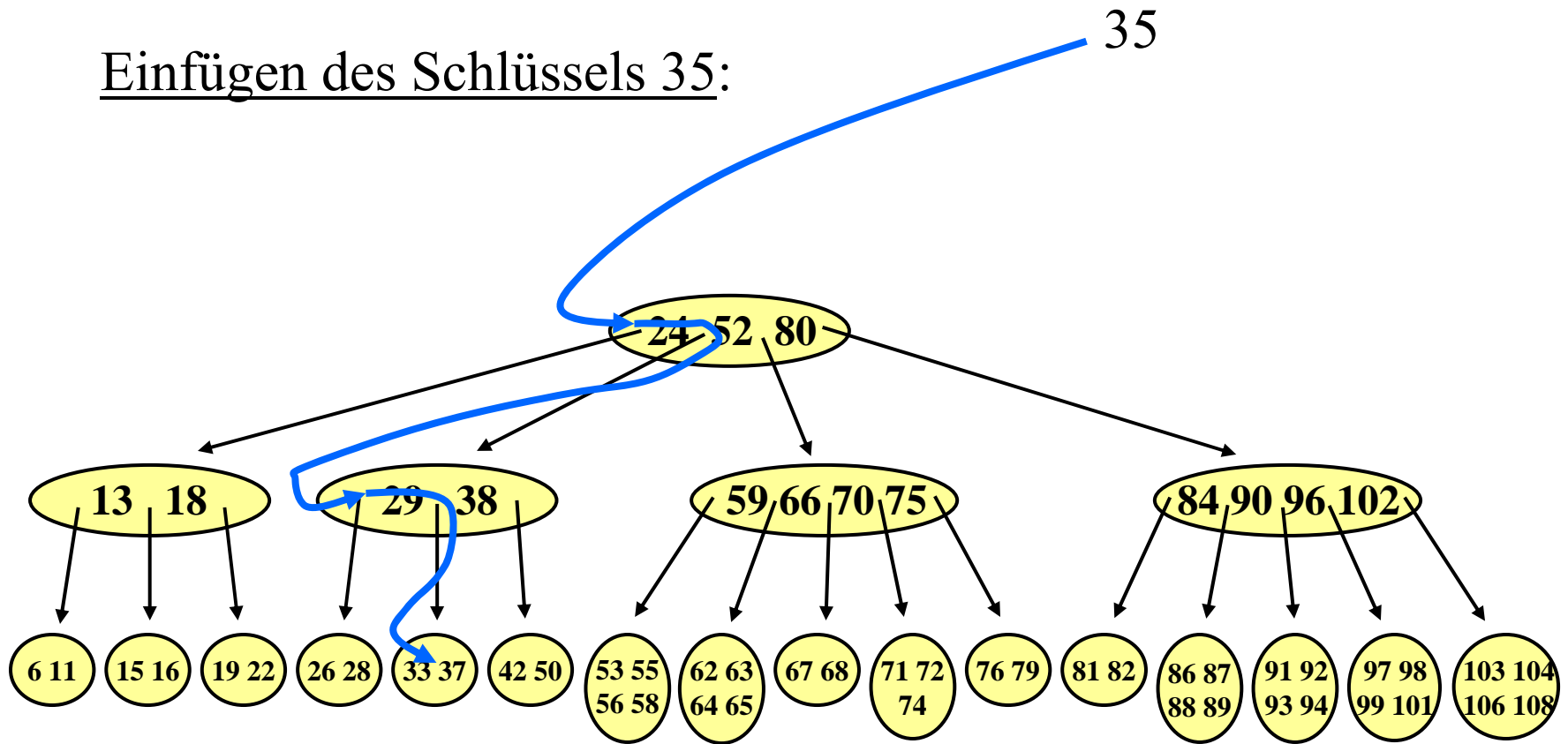
8.5.4 Einfügen in einen B-Baum

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.

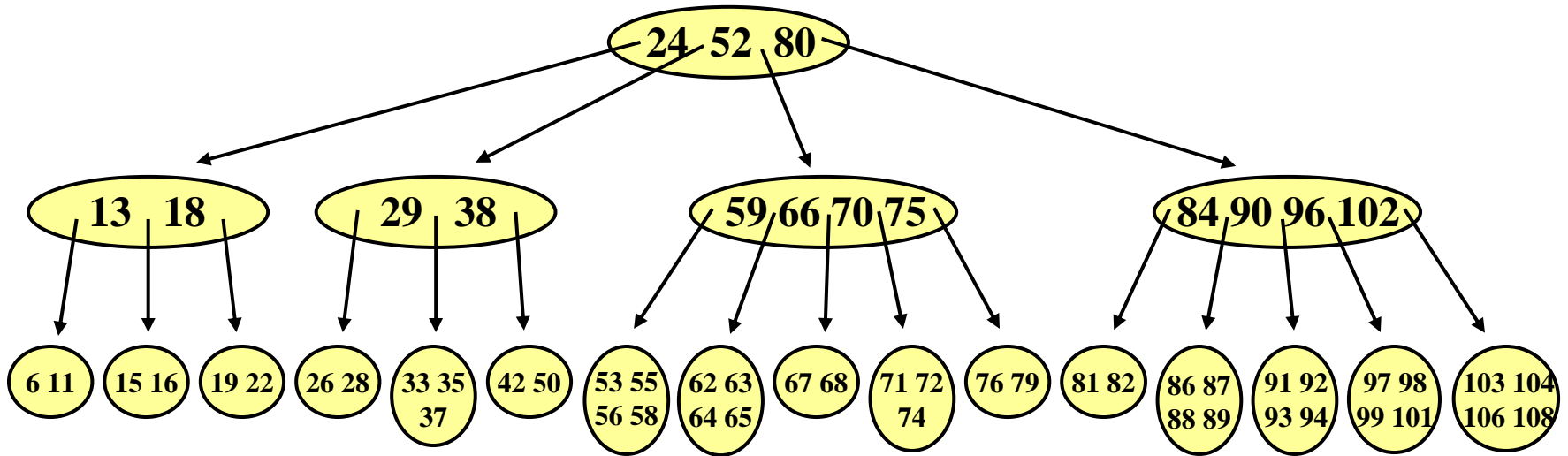
Das Vorgehen wird in den folgenden Beispielen illustriert. Der Aufwand ist wie bei der Suche, jedoch muss ggf. der Pfad bis zur Wurzel zurückverfolgt werden. Hinzu kommt das Einsortieren der Schlüssel in die Knoten.

Einfügen des Schlüssels 35:

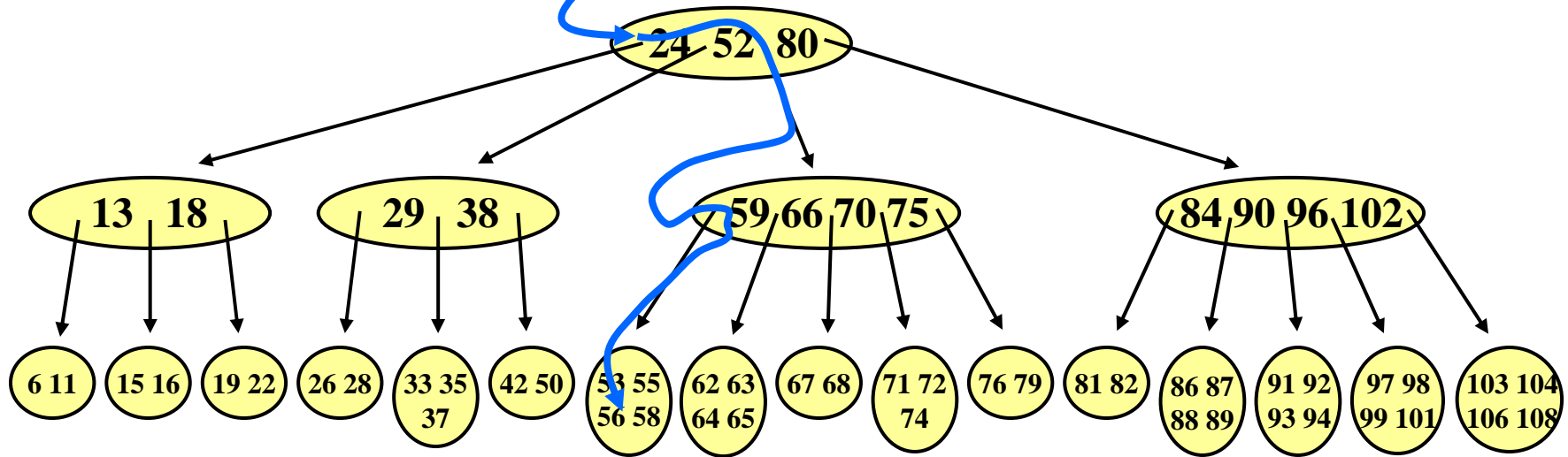


Es ist noch Platz, also wird der Schlüssel 35 sortiert hier eingetragen.

Schlüssels 35 ist nun eingefügt:

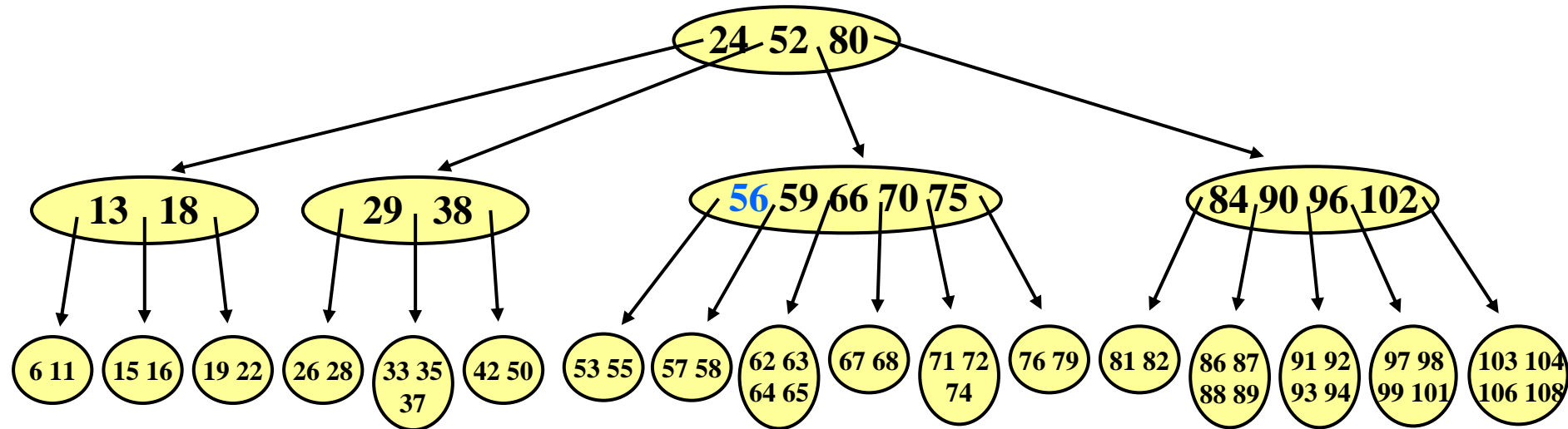


Weiteres Einfügen des Schlüssels 57: 57



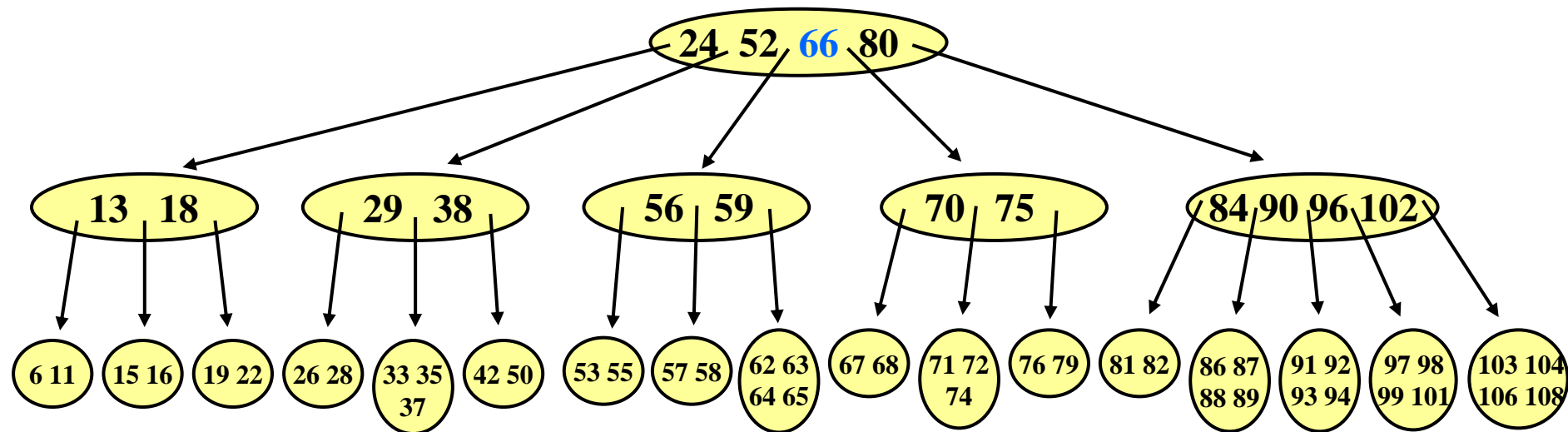
Beim Einfügen der 57 läuft der Knoten über, da er jetzt fünf Schlüssel enthält. Also wird er in zwei Knoten aufgespalten mit den Inhalten "53 55" bzw. "57 58". Der mittlere Wert "56" wird zum Elternknoten hinauf gereicht.

Weiteres Einfügen des Schlüssels 57:



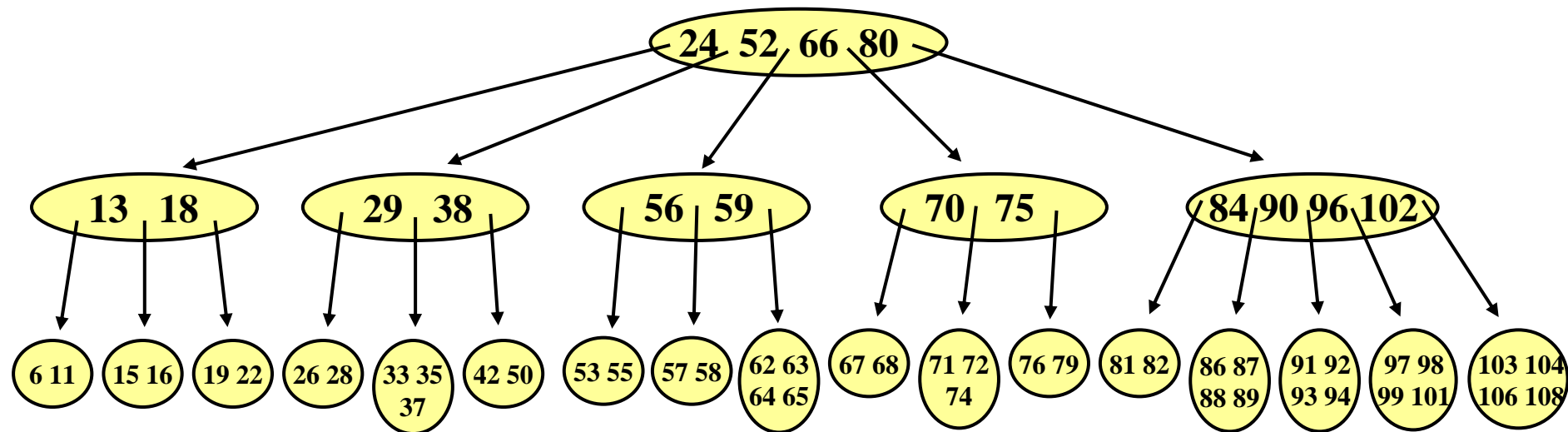
Beim Einfügen der 56 läuft aber nun der Elternknoten über. Also wird er in zwei Knoten aufgespalten mit den Inhalten "56 59" bzw. "70 75". Der mittlere Wert "66" wird zu seinem Elternknoten (dies ist hier die Wurzel) hinauf gereicht.

Ergebnis des Einfügens von Schlüssel 57:



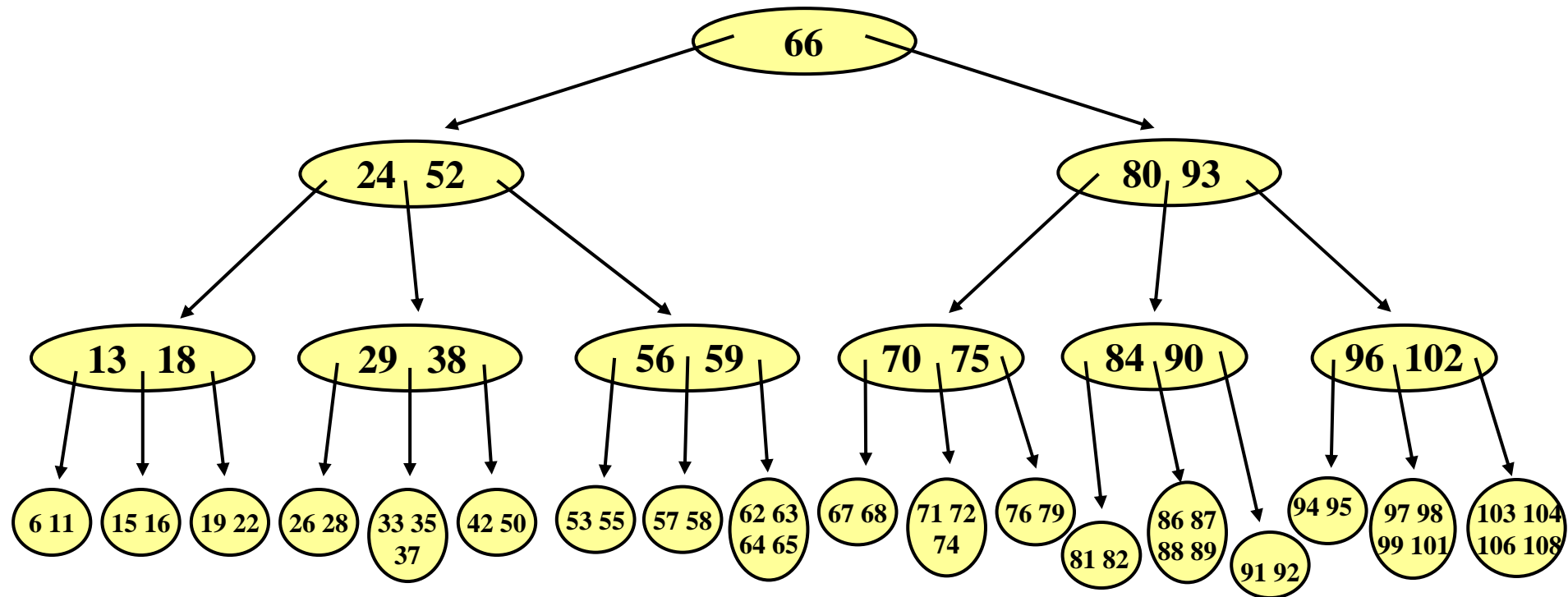
Die Wurzel darf (wie jeder Knoten in einem B-Baum der Ordnung 2) bis zu 4 Schlüssel besitzen, folglich ist dieser Baum das Ergebnis, wenn der Schlüssel 57 eingefügt wurde.

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 aufgespalten 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.

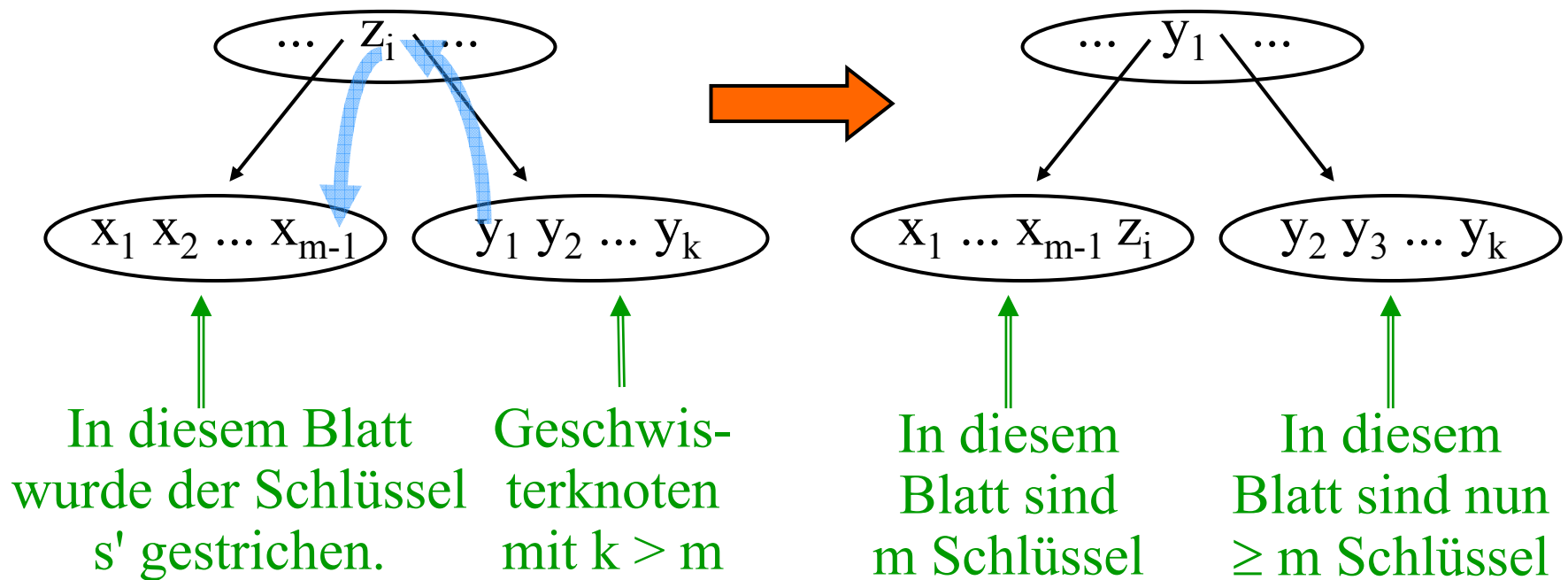
8.5.5 Löschen in einen B-Baum

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.

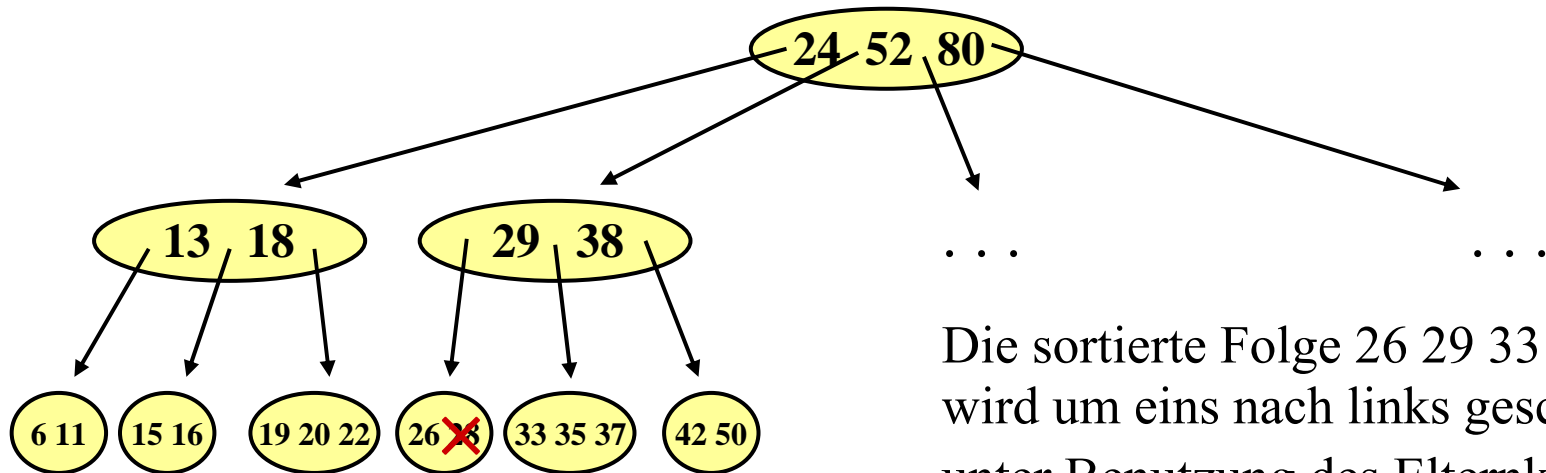
Fall 1: Besitzt das Blatt mindestens $m+1$ Schlüssel, so wird der Schlüssel s' gelöscht und man ist fertig.

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.

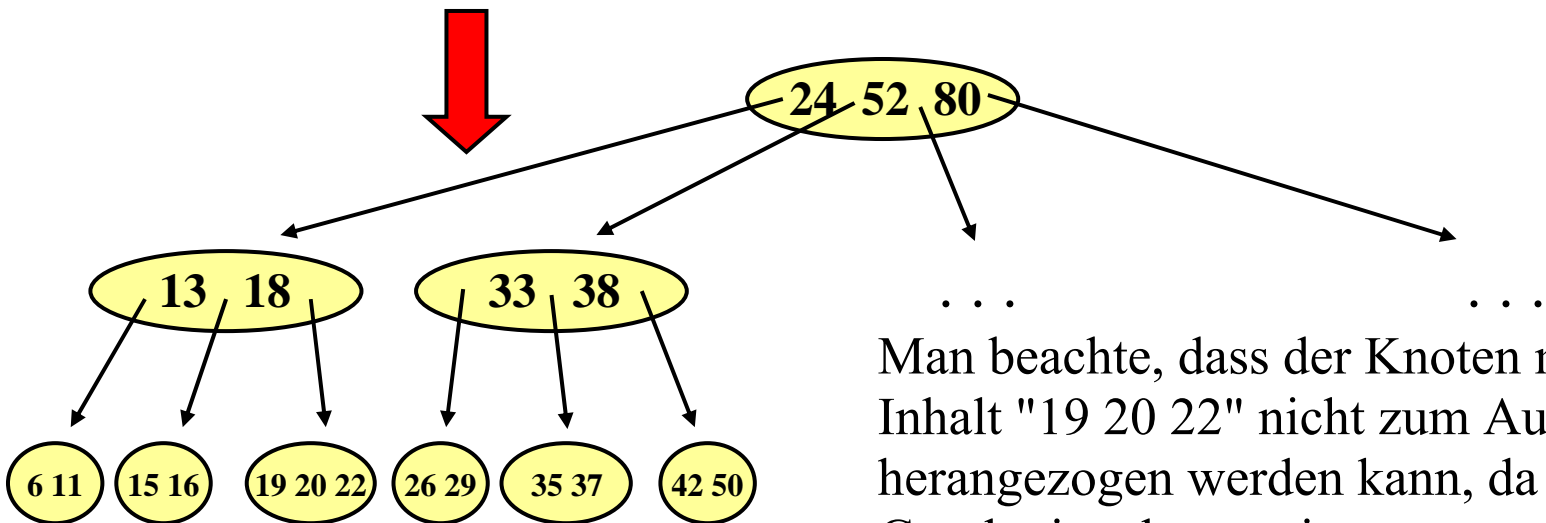
Genauer: Das Blatt besitzt nach dem Löschen von s' nur $m-1$ Schlüssel, aber ein Geschwisterknoten besitzt mindestens $m+1$ Schlüssel. Dann wird der zugehörige Schlüssel des Elternknotens in das Blatt geschoben und an seine Stelle tritt der nächstgelegene Schlüssel aus dem Geschwisterknoten. Skizze:



Beispiel zu Fall 2: Lösche den Schlüssel 28

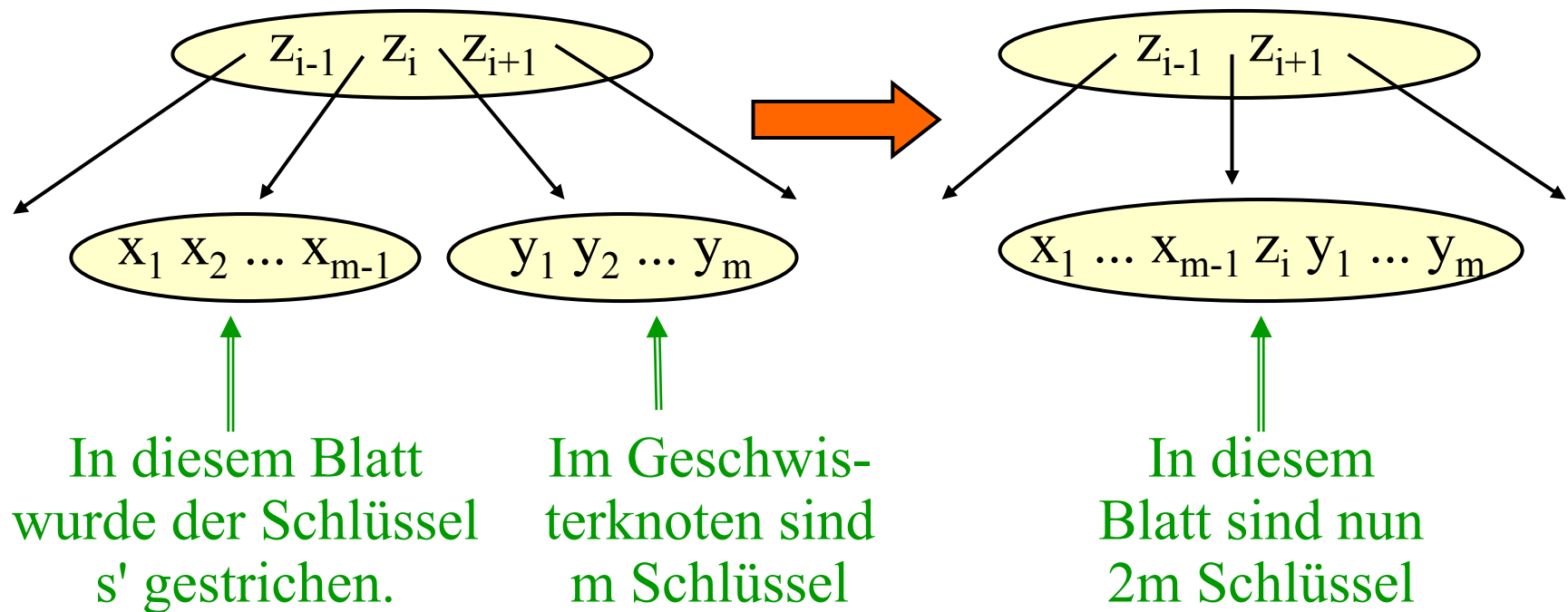


Die sortierte Folge 26 29 33 35 37 wird um eins nach links geschoben unter Benutzung des Elternknotens.



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 :

....

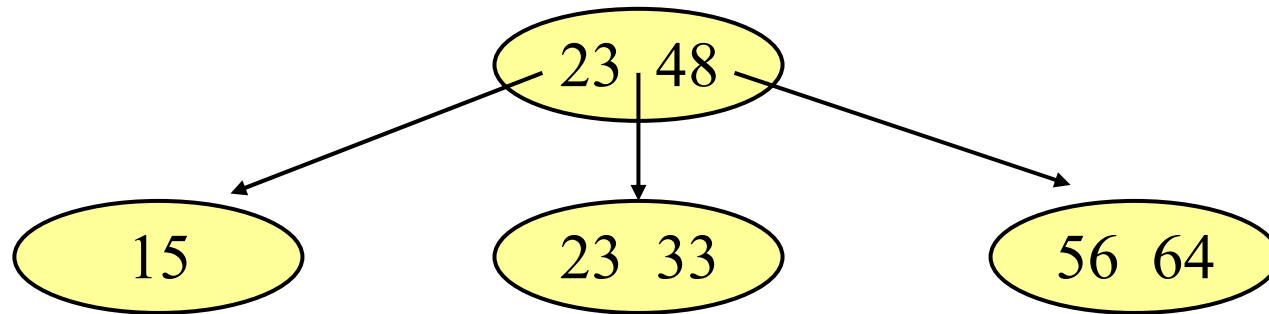
(selbst einsetzen!).

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:

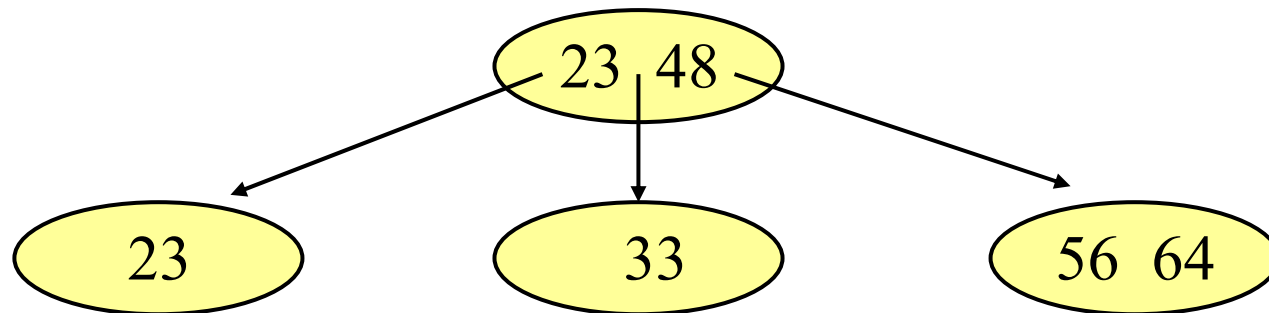
```
type Seite;  
type Ptr_Seite is access Seite;  
type Item is record Schlüssel: SchlüsselTyp;  
    Inhalt: InhaltTyp;  
    Zeiger: Ptr_Seite;  
end record;  
type Seite is record Anzahl: Natural;  
    Blatt: Boolean;  
    linkeSeite: Ptr_Seite;  
    Items: array (1..2*m) of Item;  
end record;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

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 $m=1$:

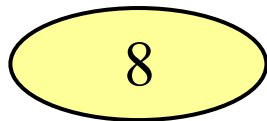


Wird nun "15" gelöscht, so wird vom Knoten "23 33" der linke Schlüssel "23" in den Elternknoten und dessen linker Schlüssel "33" in das Blatt geschoben. Ergebnis:

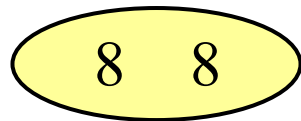


Da "23" nun links von "23" des Elternknotens steht, ist die allgemeine Suchbaumeigenschaft verletzt.

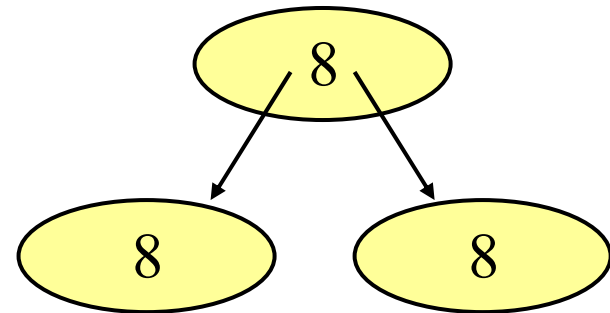
Dies lässt sich bei B-Bäumen prinzipiell nicht verhindern. Fügt man in einen B-Baum nacheinander den gleichen Schlüssel (z.B. 8) ein, so tritt der Elternschlüssel zwangsläufig sowohl im linken wie im rechten Unterbaum auf (die Ordnung sei hier erneut 1):



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, \dots, s_k die sortierten Schlüssel von u ($s_1 \leq s_2 \leq \dots \leq s_k$), 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 $i = 2, 3, \dots, k$), 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 $O(k \cdot 2^m \cdot \log_{m+1}(n))$ Vergleiche notwendig werden. Auch beim Einfügen und Löschen erhöht sich die Zahl der Schritte.

Man kann dies vermeiden zum Beispiel durch:

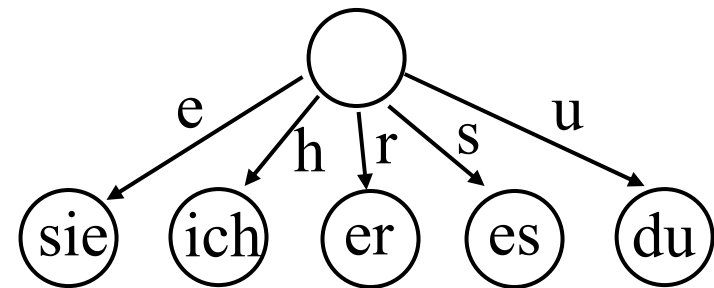
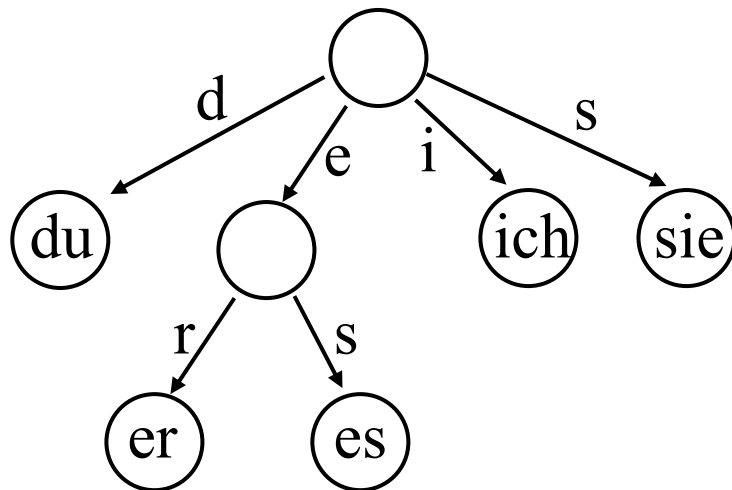
- Man speichere jeden Schlüssel im B-Baum nur einmal und lege für jeden Schlüssel eine lineare Liste an, auf deren Anfang von B-Baum-Knoten verwiesen wird.
- Man verwende B*-Bäume, bei denen die unterste Schicht, in der alle Schlüssel und Inhalte stehen, doppelt verkettet ist.

Über Details selbst nachdenken.

8.6 Digitale Suchbäume

Sind Schlüssel Wörter über einem Alphabet (und das sind sie meistens), so kann man sie zeichenweise (digit per digit, also "digital") lesen und hiermit gleichzeitig einen Baum durchlaufen, entweder vollständig oder bis die restliche Zeichenfolge eindeutig auf den Schlüssel schließen lässt.

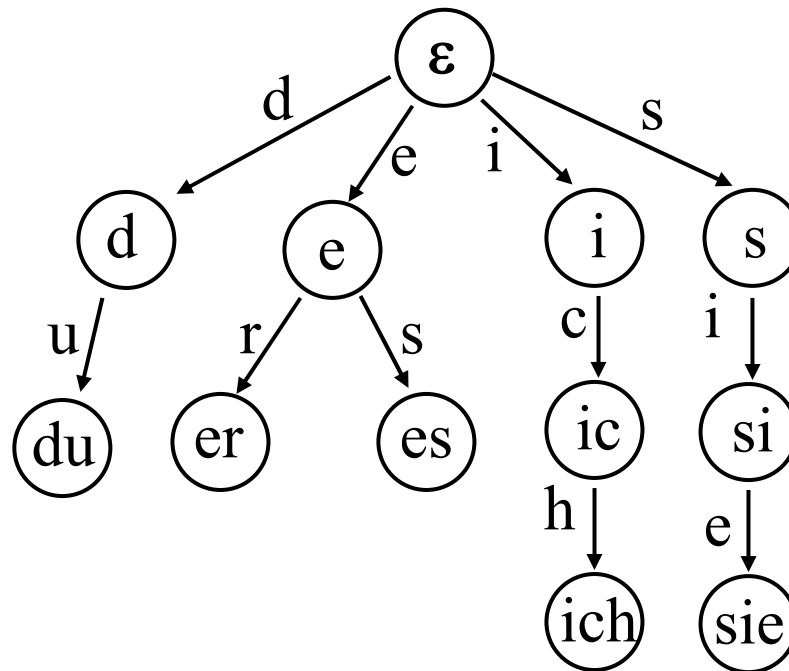
Beispiel: Die Schlüssel lauten "ich", "du", "er", "sie", "es".



Durchlaufen der Schlüssel von vorne (links) und hinten (rechts).

Baum mit vollständigen Pfaden (von vorne nach hinten durchlaufener Schlüssel):

Die Schlüssel lauten wieder "ich", "du", "er", "sie", "es".

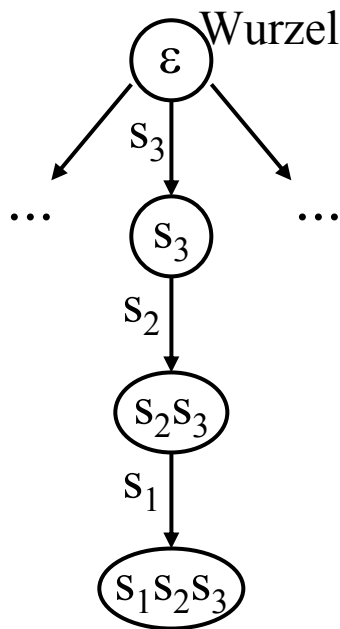


Einen solchen geordneten digitalen Suchbaum nennt man einen Trie.

Meist fasst man die Schlüssel s als Zahl zur Basis m auf, also $s \in \Sigma^* = \{0, 1, \dots, m-1\}^*$; m ist die Zahl der Alphabetzeichen. Betrachte $s = s_1 s_2 \dots s_k$, dann erhält man den Knoten, der zu s gehört, indem man die Zeichen von hinten nach vorne mittels Modulo-Bildung berechnet:

```
z := "Wurzel des Baums"; zahl := s;  
for i := 1 to k do                                -- oder: while zahl > 0 do ...  
    x := zahl mod m; zahl := zahl div m;  
    z := "x-tes Kind von z" od;  
"Untersuche den Inhalt von z"
```

Der Pfad von der Wurzel zu einem Knoten mit dem Schlüssel $s_1s_2\dots s_k$ ist mit genau diesen Zeichen s_i markiert. Dabei kann man Abkürzungen vornehmen, sofern die weitere Zeichenfolge eindeutig (= eine Kantenfolge ohne Abzweigungen) ist. *Wir lesen hier die Schlüssel von hinten, da sie auch in dieser Reihenfolge berechnet werden!*



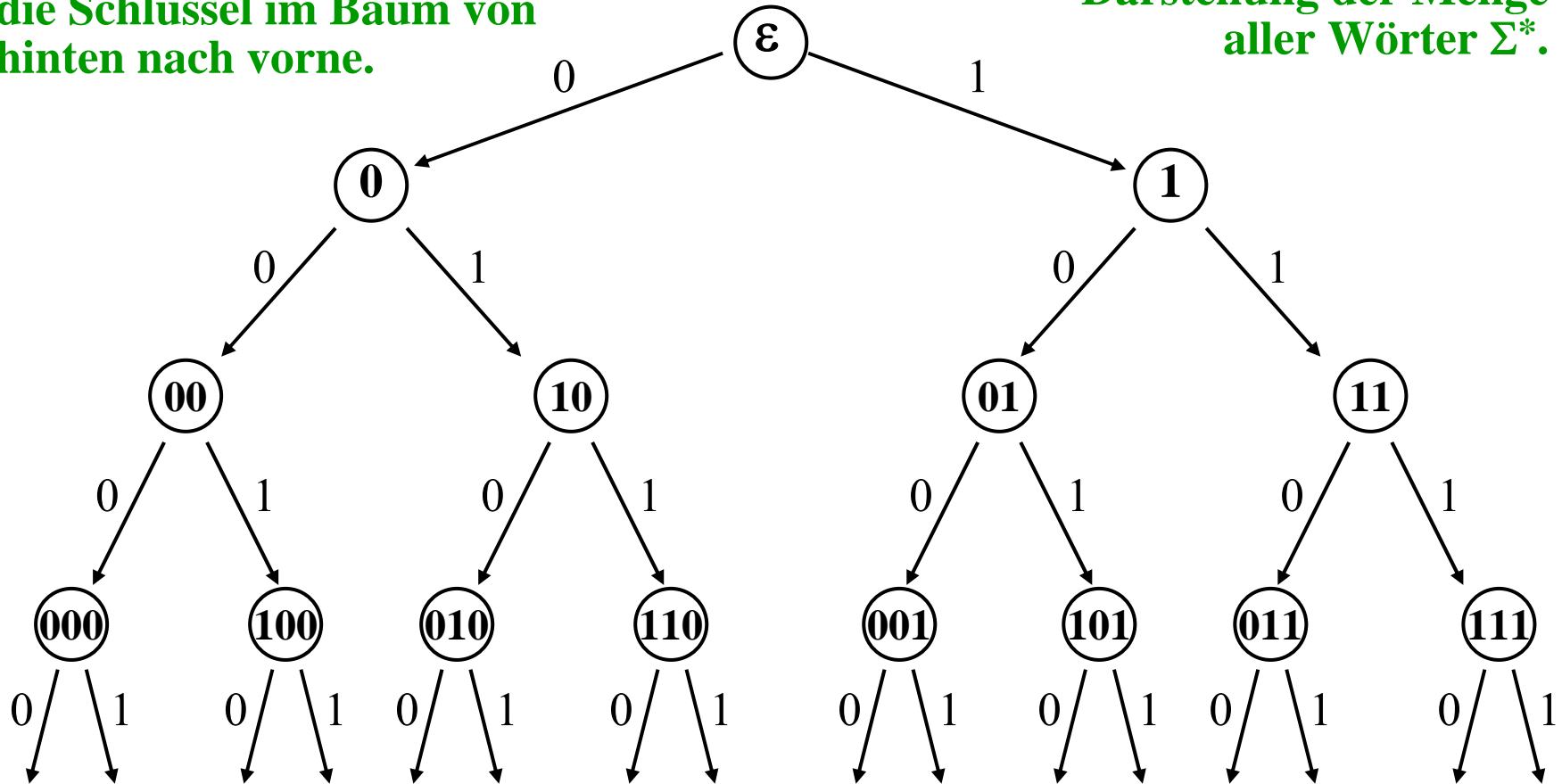
Viele Knoten dienen nur als Verzweigungsknoten zu den Knoten der Schlüssel.

Prinzipiell muss man in jedem Knoten mit so vielen Söhnen rechnen, wie es Zeichen im Alphabet gibt. Z.B.: Buchstaben und Ziffern für Bezeichner oder alle Nicht-Steuerzeichen des ASCII-Alphabets (62 bzw. 96 Elemente).

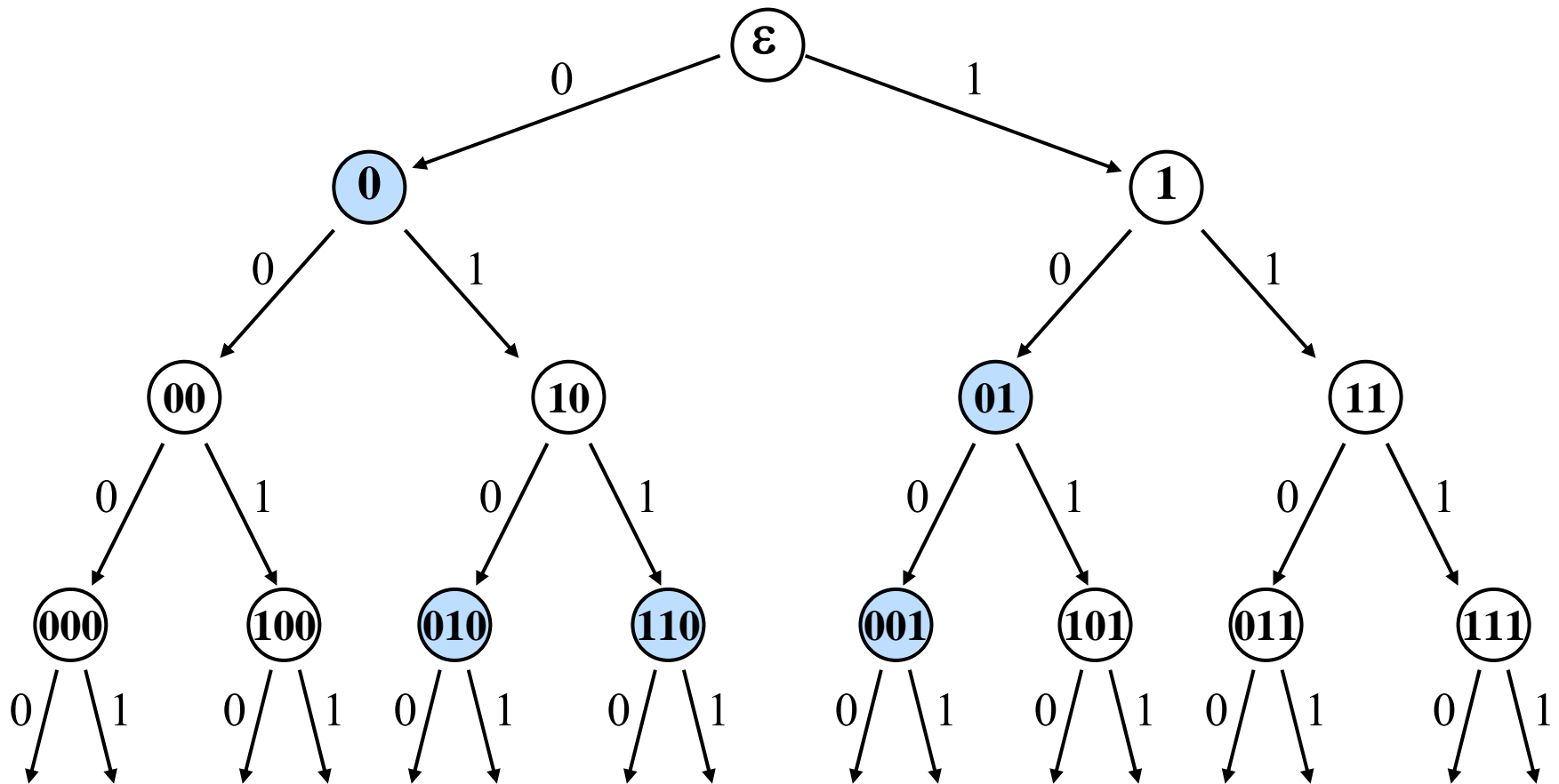
Am Beispiel des Alphabets $\Sigma = \{0, 1\}$ lässt sich dies gut demonstrieren.

**Beachte: Wir speichern hier
die Schlüssel im Baum von
hinten nach vorne.**

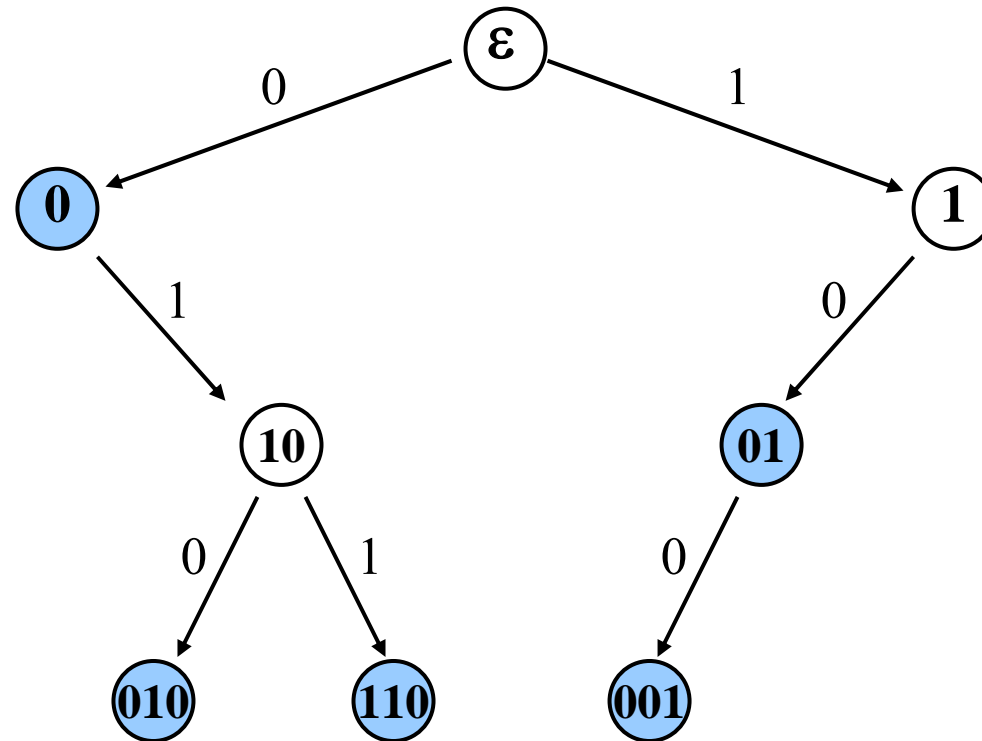
**Darstellung der Menge
aller Wörter Σ^* .**



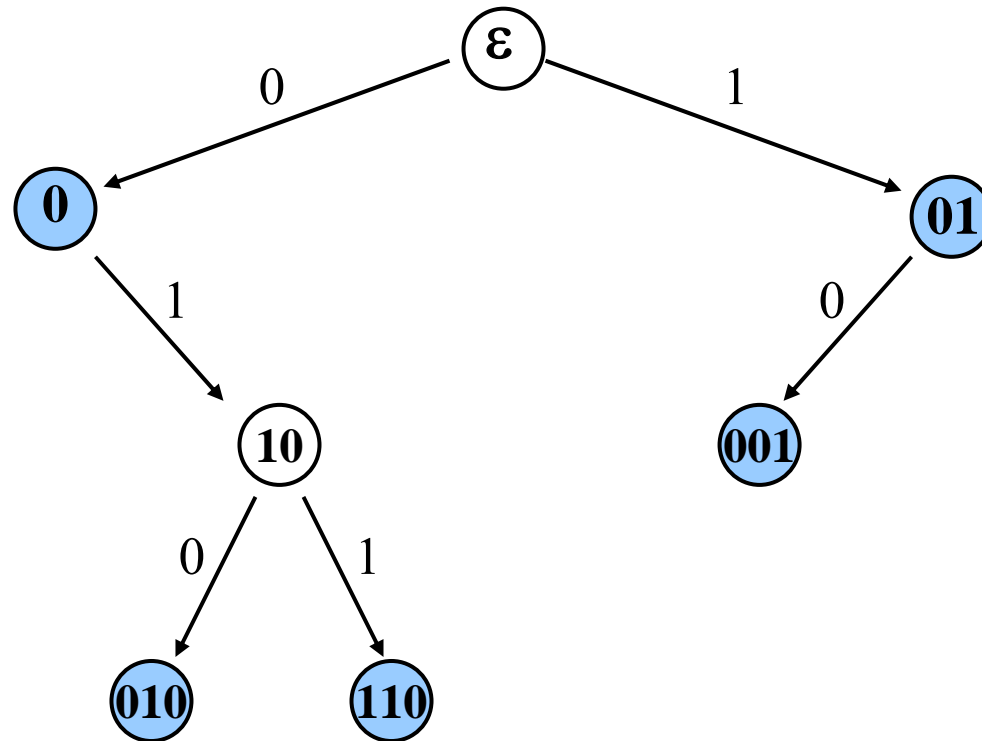
Jeder Knoten entspricht der Beschriftung auf dem Pfad von der Wurzel zu ihm. 0 = linkes Kind, 1 = rechtes Kind.



Die Menge $\{0, 01, 010, 110, 001\}$ im unendlichen Baum.



Die Menge $\{0, 01, 010, 110, 001\}$ als digitaler Baum.



Etwas verkürzter Baum für die Menge $\{0, 01, 010, 110, 001\}$.
Beim Suchen und Einfügen muss man dann ab dem aktuell im Knoten stehenden Schlüssel weiter vergleichen.

Digitaler Suchbaum über $\{0,1\}$:

Ansatz 1: Gegeben sind Schlüssel als 0-1-Folgen. Für jeden solchen Schlüssel durchlaufe man den (binären) 0-1-Baum entsprechend der Binärfolge des Schlüssels und platziere ihn genau in dem zum Schlüssel gehörenden Knoten.

FIND, INSERT und DELETE verhalten sich nun wie bei binären Suchbäumen.

Vorteile dieser Struktur vor allem dann, wenn man auf Maschinenebene programmiert oder die Binärdarstellung von Schlüsseln vorliegt.

Nachteil: Dies kostet oft viel Speicherplatz.

Digitaler Suchbaum über $\{0,1\}$:

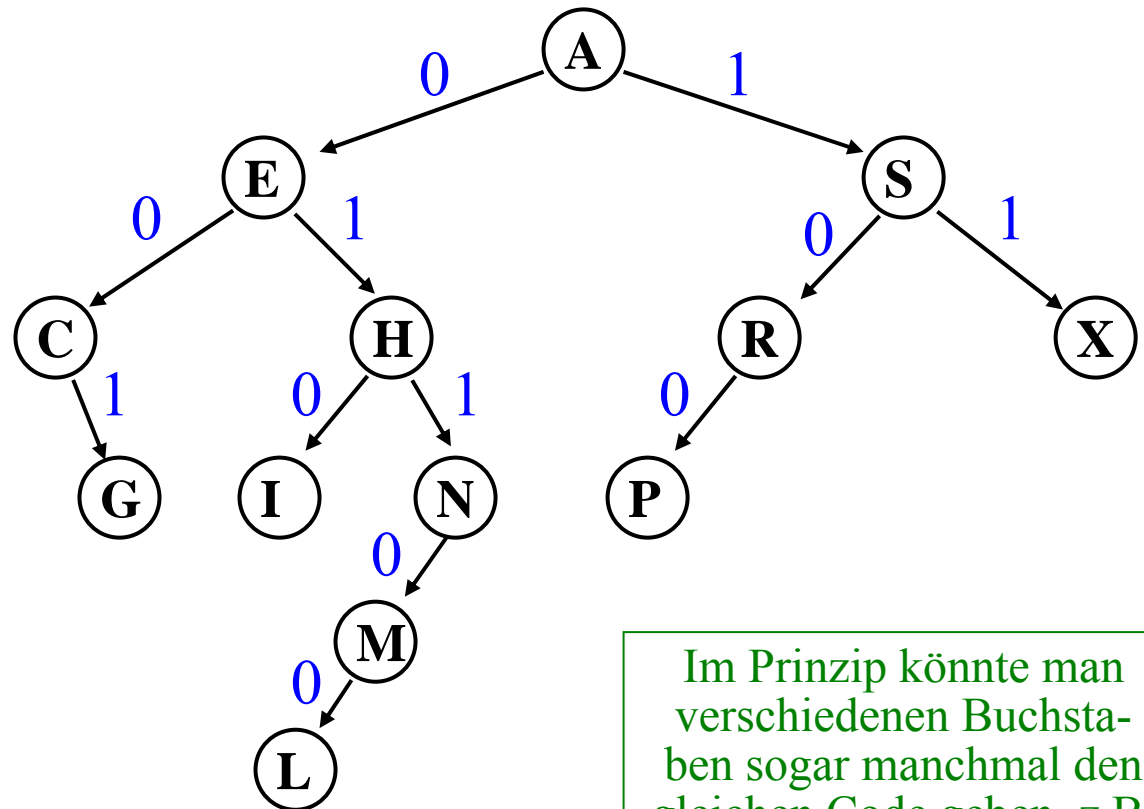
Ansatz 2: Verkürzung der Pfade um lineare Ketten. Gegeben sind Schlüssel als 0-1-Folgen. Für jeden solchen (stets vorwärts- oder stets rückwärts gelesenen) Schlüssel durchlaufe man den 0-1-Baum und platziere den Schlüssel auf den ersten freien Knoten.

FIND, INSERT und DELETE wie bei binären Suchbäumen, *jedoch muss in jedem Knoten auf Gleichheit während des Durchlaufs geprüft werden.*

Digitale Suchbäume über beliebigen Alphabeten ergeben sich als nahe liegende Verallgemeinerung.

Übliches Vorgehen: Codiert man Zeichen eines beliebigen Alphabets binär, so kann man entsprechend der Reihenfolge der Schlüssel mit ihnen einen Binärbaum aufbauen und jeden Schlüssel in das jeweils erreichte Blatt legen. Beispiel (aus der Plödereder-Vorlesung 2001) *diesmal von vorne nach hinten*:

A: 00001
S: 10011
E: 00101
R: 10010
C: 00011
H: 01000
I: 01001
N: 01110
G: 00111
X: 11000
M: 01101
P: 10000
L: 01100



Im Prinzip könnte man verschiedenen Buchstaben sogar manchmal den gleichen Code geben, z.B. $P \leftrightarrow 10010 (\leftrightarrow R)$.

In der Praxis verwendet man ein Endezeichen für Schlüssel, z.B. '#'.

Die Schlüssel 1, 10 und 101 werden also als
1#, 10# und 101# dargestellt,
wenn man einen digitalen Baum aufbaut.

Die Implementierung dieser Ideen sollte nun klar sein (?).
Hinweise zur möglichen Datenstruktur für digitale Bäume
über $\{0,1\}$ und zum Suchen und Aufbauen finden Sie auf den
nächsten Folien. Das Löschen in einem digitalen Baum ist je
nach Ansatz unterschiedlich aufwändig.

```

Maxlaenge: constant Positive := ... ;
type Stelle is (0,1,#);
type Schluesstyp is array (1..Maxlaenge) of Stelle;
type Knotentyp;
type Baumtyp is access Knotentyp;
type Knotentyp is record
    Schluessel: Schluesstyp := (others => #);
    L, R: Baumtyp;
end record;

```

Für eine natürliche Zahl (= Schlüssel) s sei
 $\text{bit}(s,k) = \text{if } s < 2^k \text{ then } k\text{-tes Bit von hinten in der Binärdarstellung von } s \text{ else } \# \text{ fi.}$
 $\text{bit}(s,1)$ ist also die letzte Binärstelle von s , $\text{bit}(s,2)$ die vorletzte usw. Vorne wird mit '#' aufgefüllt (nicht mit 0).

Suche oder Einfügen für Schlüssel s (nach Ansatz 2)

```
h1, h2: Baumtyp; i: Natural := Maxlaenge; Code: Schluesseltyp;  
begin h1 := "Verweis auf den digitalen Suchbaum"; h2 := null;  
    "wandle  $s$  mittels  $\text{bit}(s, k)$  für  $k = 1, 2, \dots, \text{Maxlaenge}$  in  
    Schluesseltyp um und weise dies Code zu";  
    while h1  $\neq$  null and then h1.Schluessel  $\neq$  Code loop  
        h2 := h1;  
        if Code(i) = 0 then h1 := h1.L;  
        elsif Code(i) = 1 then h1 := h1.R;  
        else h1 := null; end if;  
        i := i - 1;  
    end loop;  
    if h1 = null then "s nicht im Baum; h2 verweist auf den Knoten,  
        an den ein Knoten mit Schlüssel  $s$  angehängt werden kann"  
    else "h1 verweist auf Knoten mit dem Schlüssel  $s$ " end if;  
end;
```

**Man hört hierbei
auch auf, wenn '#'
erreicht wird.**

8.7 Datenstrukturen mit Historie

Dieser Abschnitt entstand nach einem Vortrag von Prof. Thomas Ottmann (Universität Freiburg) am 25.6.2004.

Entwicklungen und Vorgänge unterliegen der Zeit. So gibt es auch beim Aufbau einer konkreten Datenstruktur stets eine Historie, also einen zeitlichen Ablauf, in dem diese Struktur entsteht.

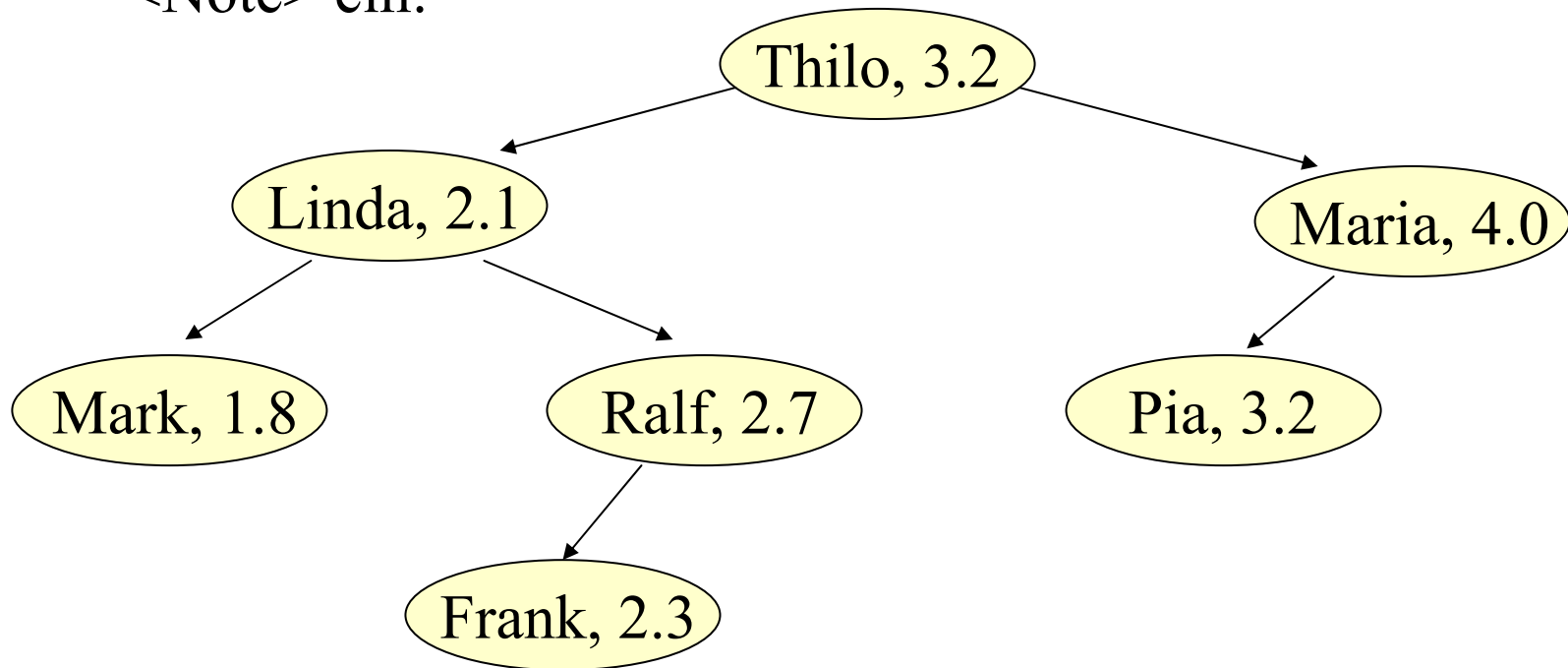
Wenn wir uns die zeitliche Reihenfolge merken möchten, so müssen wir die Datenstruktur und die verwendeten Operationen kennen. In diesem Kapitel 8 würden wir geordnete Bäume und die Operationen **FIND**, **INSERT** und **DELETE** wählen.

Wir betrachten ein Beispiel:

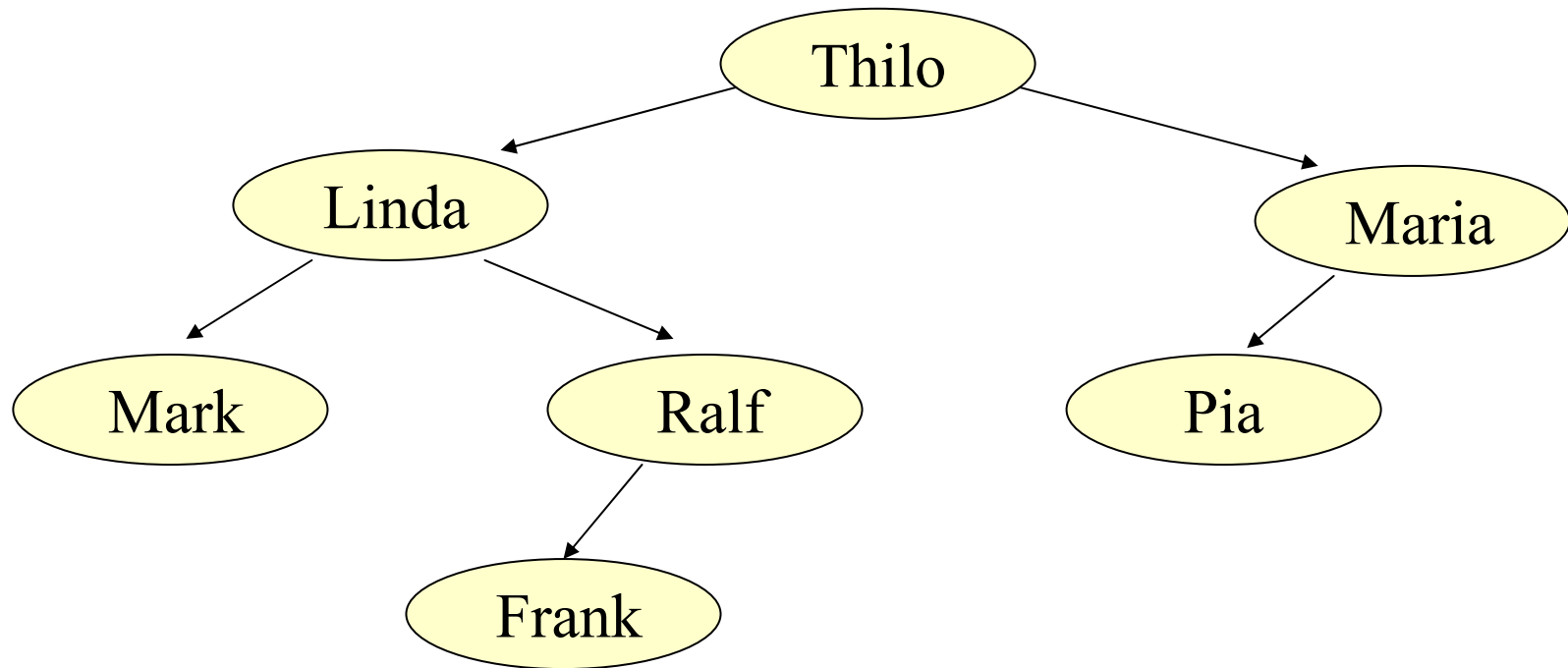
8.7.1 Beispiel "Teilnehmer und Ergebnisse einer Klausur":

Die Datenstruktur sei ein binärer Suchbaum. Die verwendeten Daten "(<Name>,<Note>)" bilden die Menge $\{(Tilo, 3.2), (Linda, 2.1), (Ralf, 2.7), (Maria, 4.0), (Mark, 1.8), (Frank, 2.3), (Pia, 3.2)\}$.

Diese 7 Elemente tragen wir in dieser Reihenfolge mittels **INSERT** in einen binären Suchbaum bzgl. des Schlüssels <Note> ein:

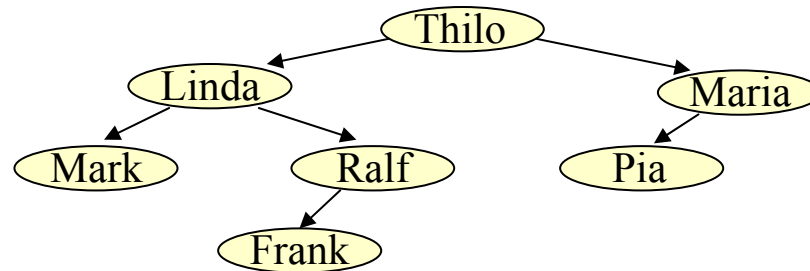


Wir möchten nun wissen, wer an der Klausur teilgenommen hat. Hierzu lassen wir die Noten weg und übermitteln den verbliebenen Baum.



Können wir diesem fertigen Produkt, also dem binären Baum, mehr als nur die Teilnehmer ansehen?

Ja!



Erstens erkennen wir die Rangordnung der Klausurergebnisse mit einem Inorder-Durchlauf:

Mark, Linda, Frank, Ralf, Thilo, Pia, Maria.

Zweitens liefert uns der Baum eine Halbordnung der Eintragsreihenfolge: Jeder Pfad von der Wurzel zu einem Blatt gibt die relative Reihenfolge beim Einfügen an.

So müssen Linda vor Ralf und Frank, Maria vor Pia, Linda vor Mark usw. eingefügt worden sein.

Folgerung: Die Datenstruktur speichert also nicht nur irgendwelche Fakten, sondern sie merkt sich zugleich Ausschnitte aus ihrer Historie und gewisse zusätzliche Inhalte.

Im hier betrachteten Fall sind die zusätzlichen Informationen nichts anderes als Relationen auf der Menge der Daten.

Idee und intuitive Formulierung: Es sei M die Menge der Daten, die in einer Datenstruktur D gespeichert sind. Eine Relation $R \subseteq M^k = M \times M \times \dots \times M$ heißt *mit D verträglich*, wenn jede Beziehung $(m_1, m_2, \dots, m_k) \in R$ aus der Datenstruktur D gefolgert werden kann.

Diese Formulierung ist nicht so genau, dass wir sie programmieren könnten. Es fehlt die Präzisierung, was es bedeutet, dass eine Beziehung aus einer Datenstruktur gefolgert werden kann. Hierzu müssten wir eine formale Logik über den Datenstrukturen definieren. Siehe hierzu Vorlesungen aus dem Gebiet "sichere und zuverlässige Systeme". Wir verfolgen daher eine andere Idee.

Eine Datenstruktur d_i entsteht aus der Menge der Daten M , indem zulässige Operationen auf Elemente aus M und die bis dahin gewonnene Datenstruktur d_{i-1} angewendet werden. Anfangs ist die Datenstruktur leer: $d_0 = \text{'empty'}$.

Definition 8.7.2

- Es sei M eine Menge. Es sei D eine Menge von Datenstrukturen, deren Inhalte aus M seien. Es sei 'empty' eine spezielle Datenstruktur aus D (die "leere Struktur").
- Es sei Ω eine endliche Menge von Operationen.
- Der Einfachheit halber nehmen wir hier an, dass für jedes $\omega \in \Omega$ gilt: $\omega: M \times D \rightarrow D$.

Eine Folge von Operationen und Elementen

$h = (\omega_1, m_1), (\omega_2, m_2), \dots, (\omega_r, m_r)$ heißt eine **Historie** der Länge r von $d = \omega_r(m_r, \dots (\omega_2(m_2, \omega_1(m_1, \text{empty}))) \dots) \in D$.

Wir vollziehen also den Aufbau der Datenstruktur schrittweise nach. Wenn wir nacheinander die Operationen $\omega_1, \omega_2, \dots, \omega_r$ mit den Elementen m_1, m_2, \dots, m_r durchführen, so erhalten wir nacheinander die Datenstrukturen $d_0, d_1, d_2, \dots, d_r$:

$d_0 = \text{'empty'}$, $d_1 = \omega_1(m_1, \text{empty})$, $d_2 = \omega_2(m_2, d_1)$,
 $d_3 = \omega_3(m_3, d_2)$, \dots $d_i = \omega_i(m_i, d_{i-1})$ für alle $i > 0, \dots$,
und am Ende:

$d_r = \omega_r(m_r, d_{r-1}) = \omega_r(m_r, \dots (\omega_2(m_2, \omega_1(m_1, \text{empty}))) \dots)$.

Frage 1: Wenn man d_r kennt, was lässt sich dann über dessen Historie $(\omega_1, m_1), (\omega_2, m_2), \dots, (\omega_r, m_r)$ aussagen?

Frage 2: Wieviele verschiedene Historien (der Länge r oder $\leq r$) kann eine Datenstruktur $d \in D$ haben?

Frage 3: Was muss man tun, um aus d stets auf die Historie schließen zu können?

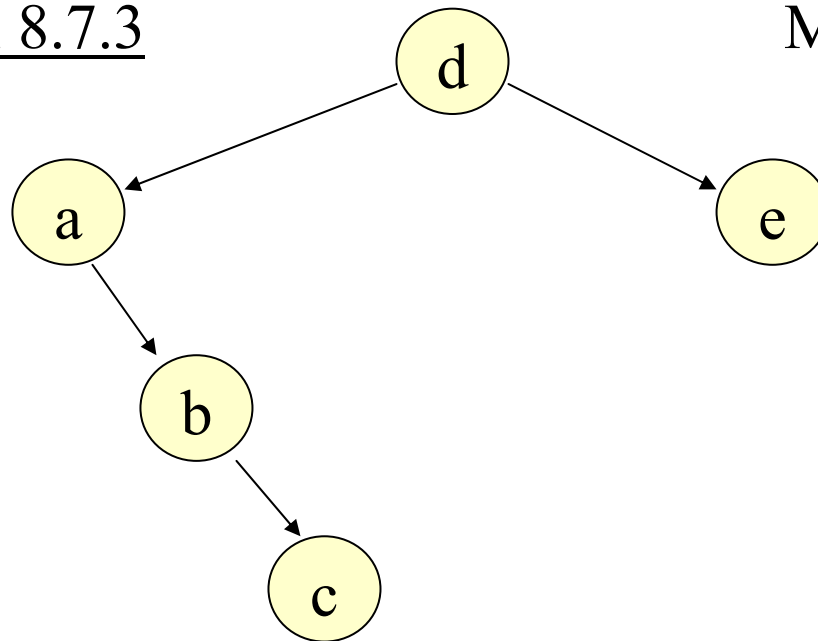
Dies muss man an Beispielen erläutern. Wir wählen als Menge M die ganzen Zahlen \mathbb{Z} und als Menge D die Menge der binären (Such-) Bäume mit ganzen Zahlen als Inhalten. 'empty' sei der leere Baum.

Als endliche Menge der Operationen Ω wählen wir $\{\text{FIND}, \text{INSERT}, \text{DELETE}\}$. Für jedes $\omega \in \Omega$ ist $\omega: M \times D \rightarrow D$ klar:

- im Falle von FIND nehmen wir die Identität,
- im Falle von INSERT wird das Element als Blatt eingefügt,
- im Falle von DELETE wird über den Inorder-Nachfolger gelöscht, vgl. 8.2.10.

Wir betrachten nun einen binären Baum und fragen danach, welche Historie zu ihm gehört. Dies ist im Allgemeinen nicht eindeutig. Daher interessiert uns die Anzahl aller Historien, die ein solcher Baum haben kann.

Beispiel 8.7.3



$M = \{a, b, c, d, e\}$

Dieser Baum besitzt genau 4 Historien, wenn man nur die Operation IN (= INSERT) verwenden darf.

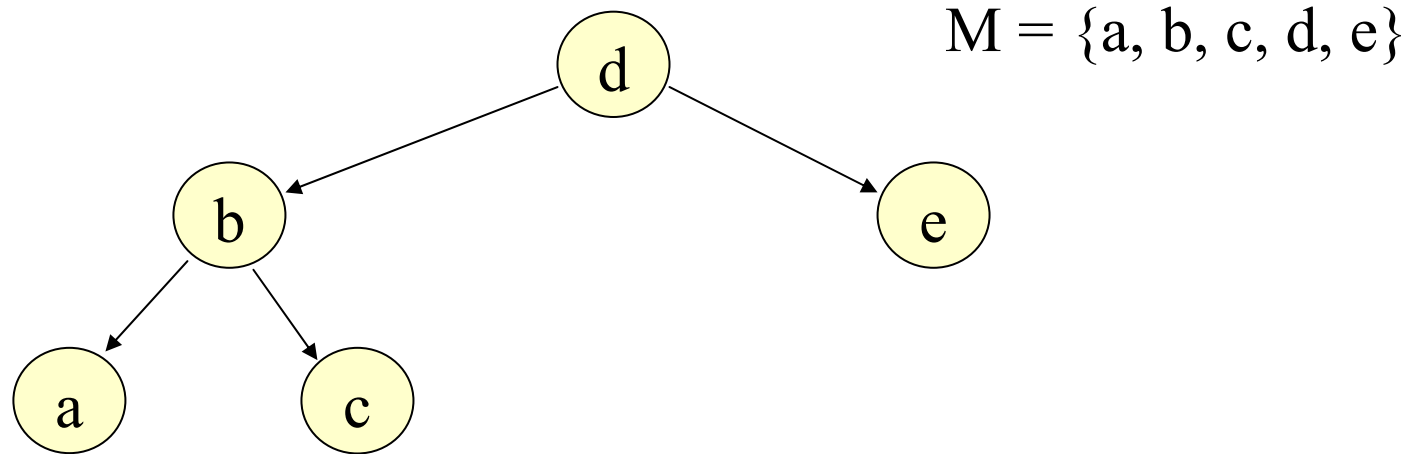
Beispiele:

Historie 1: (IN,d), (IN,a), (IN,b), (IN,c), (IN,e)

Historie 2: (IN,d), (IN,a), (IN,e), (IN,b), (IN,c)

Hinweis: Die Anzahl der Historien, die *nur* die INSERT-Operation verwenden, ist gleich der Zahl der topologischen Sortierungen des Baums (hierzu vgl. Anhang 8.8.2).

Historie 3: (IN,c), (IN,d), (IN,e), (DEL,c), (IN,a), (IN,s), (IN,c)



Historie 1: (IN,d), (IN,b), (IN,a), (IN,c), (IN,e)

Historie 2: (IN,d), (IN,b), (IN,e), (IN,c), (IN,a) usw.

Dieser Baum besitzt genau 8 Historien, wenn man nur die Operation IN verwenden darf. Seine Entstehung ist also "unbestimmter" als die des Baums auf der vorherigen Folie. Unbestimmtheit bezeichnet man oft als Entropie.

Definition 8.7.4:

Es seien M , D , Ω und $\omega: M \times D \rightarrow D$ (für $\omega \in \Omega$) wie in Def. 8.7.2.

Für eine natürliche Zahl r und eine Struktur $d \in D$ sei die r-Entropie von d die Anzahl der Historien der Länge r , die zu d gehören.

Sofern es zu d nur endlich viele Historien gibt, bezeichnet man die Anzahl aller Historien als die Entropie von d .

Hinweis: Die Strukturen mit hoher Entropie sind also zugleich die "vergesslichen Strukturen", in denen die eigene Entstehung nur teilweise notiert werden kann.

Fallstudie 8.7.5:

Es seien $M = \mathbb{Z}$ die Menge der ganzen Zahlen, D = die Menge aller binären (Such-) Bäume (mit Inhalten aus \mathbb{Z} , \Rightarrow 8.2.10), $\Omega = \{\text{IN}\}$ eine einelementige Menge und $\text{IN}: \mathbb{Z} \times D \rightarrow D$ die übliche Einfügeoperation INSERT, die durch die Prozedur procedure Einfügen (Anker: in out Ref_BinBaum; s: Integer) in 8.2.10.b exakt definiert ist (der neue Schlüssel $s \in \mathbb{Z}$ wird hierbei in ein neues Blatt eingefügt).

Gegeben sei eine natürliche Zahl n und die Menge der ersten n natürlichen Zahlen $\underline{n} := \{0, 1, 2, \dots, n-1\} \subset \mathbb{Z}$. Betrachte die Menge D_n alle binären Bäume mit genau den n verschiedenen Inhalten $0, 1, 2, \dots, n-1$. Dann bildet die Menge $S_n = \{i_1 i_2 \dots i_n \mid i_j \in \underline{n}, i_j \neq i_k \text{ für } j \neq k\}$ (= die Menge aller Anordnungen der n ersten natürlichen Zahlen) genau die Menge der Historien aller dieser binären Bäume.

Jeder Anordnung $i_1 i_2 \dots i_n$ ist genau ein binärer (Such-) Baum zugeordnet, der durch Einfügen hieraus entsteht und zu dessen Historie sie gehört. Es gibt also eine (surjektive) Abbildung $\varphi: S_n \rightarrow D_n$.

Formal genauer: Die Historie $h = (IN, i_1), (IN, i_2), \dots, (IN, i_n)$, die wir mit der Folge $i_1 i_2 \dots i_n$ identifizieren (mit $i_j \neq i_k$ für $j \neq k$), liefert genau einen binären Baum $\varphi(h)$.

Für die inverse Abbildung $\varphi^{-1}: D_n \rightarrow 2^{S_n}$ gilt dann:

$|\varphi^{-1}(d)|$ ist die Entropie des binären Baums $d \in D_n$.

($\varphi^{-1}(d) = \{ i_1 i_2 \dots i_n \mid \varphi(i_1 i_2 \dots i_n) = d \} \subset S_n$.)

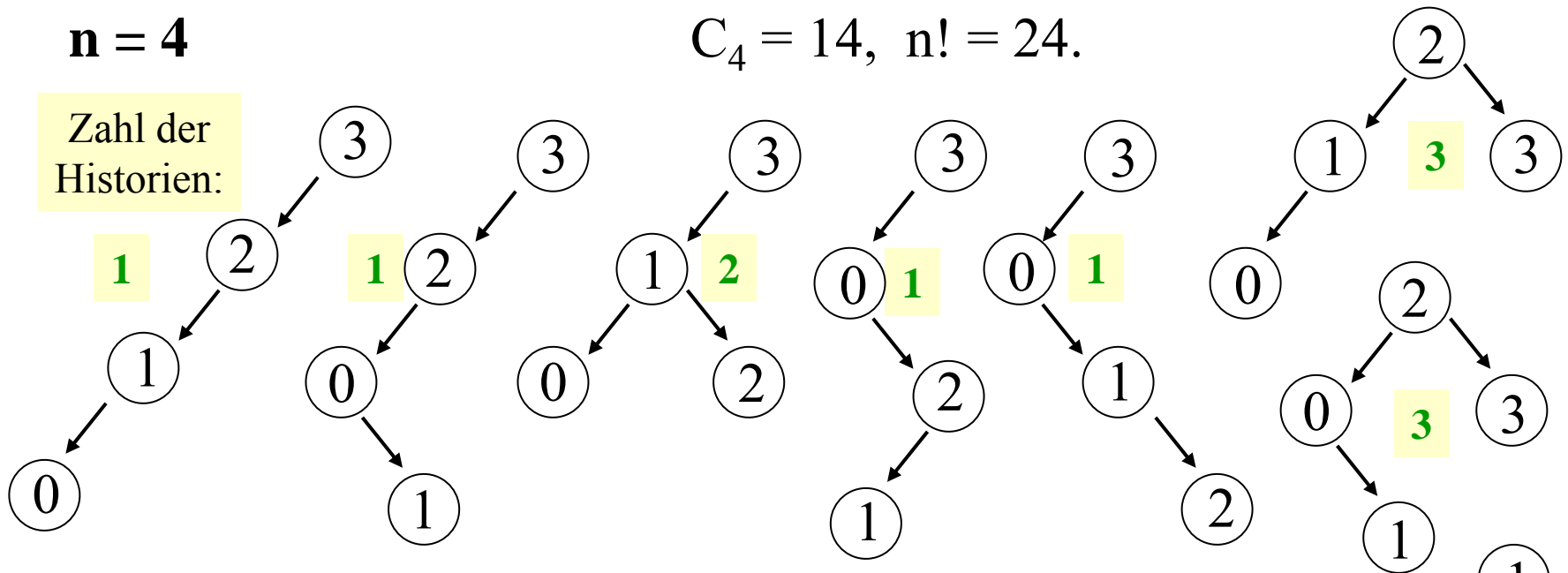
Faustregel: Je gleichverzweigter der Baum d ist, umso größer ist seine Entropie.

Dies beleuchten wir zunächst am Beispiel $n=4$.

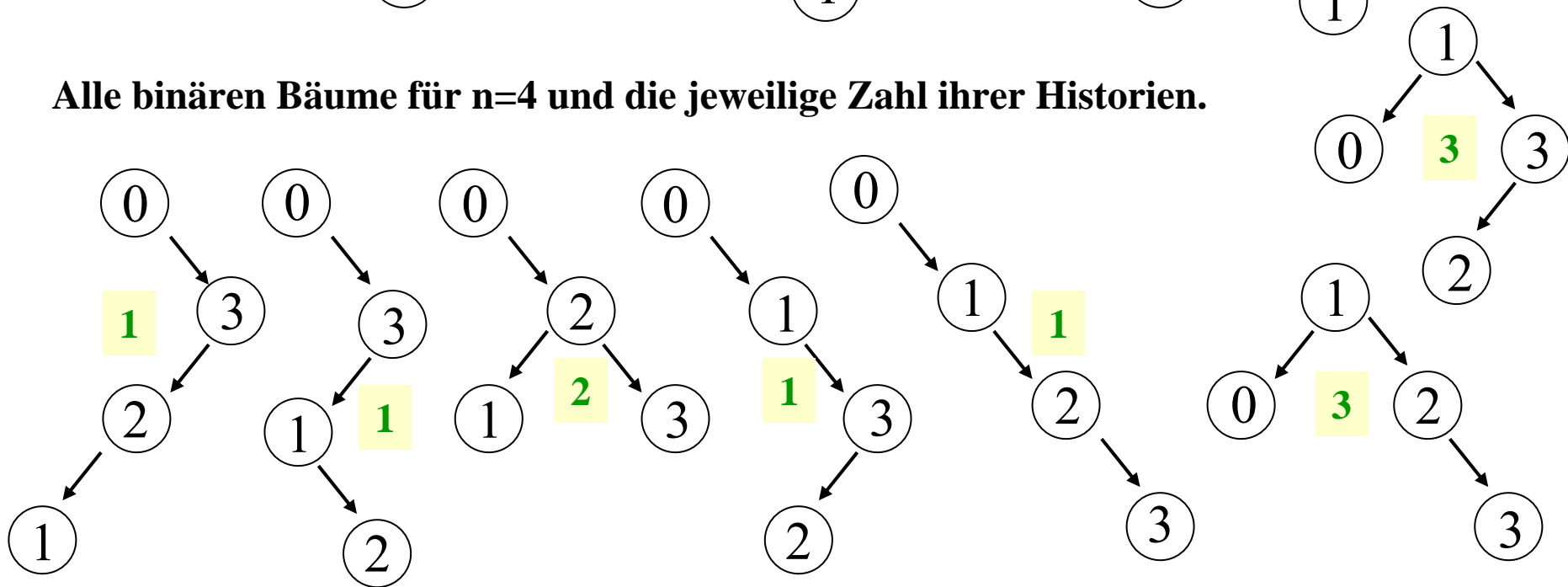
n = 4

$C_4 = 14$, $n! = 24$.

Zahl der
Historien:



Alle binären Bäume für n=4 und die jeweilige Zahl ihrer Historien.



Wie viele Historien kann ein binärer Baum mit n Knoten besitzen?

Fall 1: Der Baum hat die Tiefe n . Er bildet dann eine lineare Liste, die nur genau eine Historie besitzen kann, nämlich die Folge der Zahlen von der Wurzel zum einzigen Blatt.

Fall 2: Für $n=2^k-1$ kann man den vollständig ausgeglichenen Baum mit genau 2^{k-1} Blättern betrachten. Es sei E_k die Zahl der Historien eines solchen Baumes mit 2^k-1 Knoten. Dann gilt:

$$E_1 = 1 \quad \text{und}$$

$$E_k = \binom{2^k-2}{2^{k-1}-1} \cdot E_{k-1} \cdot E_{k-1} \quad \text{für } k > 1.$$

$$E_1 = 1 \text{ und}$$

$$E_k = \binom{2^k - 2}{2^{k-1} - 1} \cdot E_{k-1} \cdot E_{k-1} \text{ für } k > 1.$$

Begründung: $E_1 = 1$ ist klar, weil es nur einen Knoten gibt. Es liege nun ein binärer Baum mit $2^k - 1$ Knoten $k > 1$ vor, dessen Wurzel den Inhalt z besitzt.

Dann muss z am Anfang der Historie stehen und man kann für die $2^{k-1} - 1$ Inhalte des linken Unterbaums der Wurzel irgendeine beliebige Teilmenge der $2^k - 2$ Plätze in der Historie festlegen. Dies sind " $(2^k - 2)$ über $(2^{k-1} - 1)$ " Möglichkeiten. Auf diesen $2^{k-1} - 1$ Plätzen gibt es E_{k-1} Möglichkeiten der Zuordnung zu den Inhalten $< z$ und ebenso viele Möglichkeiten für die Zuordnung der Inhalte $> z$ zu den restlichen $2^{k-1} - 1$ Plätzen. Diese Auswahlen kann man unabhängig voneinander treffen, woraus sich die Rekursionsformel für E_k ergibt.

E_k ist eine schnell wachsende Funktion. Die ersten Werte:

$$k = 1 \text{ und } n = 1: E_1 = 1; \quad n! = 1.$$

$$k = 2 \text{ und } n = 3: E_2 = 2; \quad n! = 6.$$

$$k = 3 \text{ und } n = 7: E_3 = 80; \quad n! = 5040.$$

$$k = 4 \text{ und } n = 15: E_4 = 21.964.800; \quad n! = 1.307.674.368.000.$$

$$E_5 \text{ ist ungefähr } 7,5 \cdot 10^{22} \text{ (mit zugehörigem } n! = 31! \approx 8,2 \cdot 10^{33}).$$

Vergleichswert (vgl. 8.2.13): Der durchschnittliche Wert für die Entropie eines binären Baumes mit n Knoten beträgt $n!/C_n$. Für $n = 2^k - 1$ muss der Wert E_k deutlich über diesem Wert liegen.

Für $k = 5$ und $n = 31$ ist $n!/C_n \approx 5,6118 \cdot 10^{17}$, d.h., im Durchschnitt hat jeder binäre Baum mit 31 Knoten rund $5,6118 \cdot 10^{17}$ Historien. Der Wert für E_5 ist um etwas mehr als das 10^5 -fache größer, was durchaus plausibel für diesen Wert von n erscheint.

Für $k > 2$ gilt: $(E_k)^2 > (2^k - 1)!$, wie man mit Induktion unter Verwendung der Stirlingschen Formel beweist.

Aufgabe für Fortgeschrittenere und/oder an der Theorie Interessierte: Untersuchen Sie die Funktionswerte E_k selbst weiter und versuchen Sie, eine genauere Abschätzung für diese Werte zu finden.

Anmerkung für Fortgeschrittenere: Die Zahl der Historien zu einem binären Baum ist gleich der Zahl der topologischen Sortierungen dieses Baums (vgl. 8.8).

Allgemeine Aussagen:

Jeder binäre Baum mit n Knoten, der eine lineare Liste ist (dessen Tiefe also genau n ist), besitzt genau eine Historie.

Ein binärer Baum mit $2^k - 1$ Knoten besitzt höchstens E_k Historien.

Alle binären Bäume mit n Knoten zusammen besitzen genau $|S_n| = n!$ Historien.

Es gibt genau C_n verschiedene binäre Bäume mit n Knoten (Satz 8.2.7). Im Durchschnitt entfallen somit $n! / C_n$ Historien auf jeden binären Baum. Diese Zahl wächst mindestens mit $(n/11)^n$, wie auf der folgenden Folie bewiesen wird.

Es gilt natürlich stets $E_k \geq (2^k - 1)! / C_{2^k - 1}$. (Beweis? Selbst versuchen.)

8.7.6: Mittlere Entropie bei binären Bäumen:

Die Zahl der Historien wächst sehr viel schneller als die Zahl der binären Bäume. Genauer:

Mit der Stirlingschen Formel ($e \approx 2,718\ 281\ 828\ 459\ 045$)

$$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \quad \text{erhält man:}$$

$$\begin{aligned} \frac{n!}{C_n} &= \frac{n!}{\frac{1}{n+1} \binom{2n}{n}} \approx \dots = \pi \sqrt{2} \cdot n \cdot (n+1) \cdot \left(\frac{n}{4e}\right)^n \\ &\approx 4.442882938 \cdot n \cdot (n+1) \cdot \left(\frac{n}{10.873127314}\right)^n \end{aligned}$$

Diese Näherung ist auch für kleine Werte von n recht genau.

8.7.7: Mittlere Suchdauer und mittlere Tiefe binärer Bäume

Mittleres Level eines Knotens in einem binären Baum, wobei jeder binäre Baum mit seiner Entropie gewichtet wird

= Mittelwert der Suche in binären Bäumen bezogen auf alle möglichen Eingabefolgen aus n verschiedenen Elementen

= (Summe über das Level aller Knoten von allen binären Bäumen mit n Knoten mal ihrer Entropie) / ($n \cdot n!$)

$\approx 1,3863 \cdot \log(n) - 1,8456$ ($= MS_n$, siehe 8.2.15).

Da sich die mittlere Suche auf alle möglichen Eingabefolgen bezieht, wurde hierbei also jeder Baum so oft gezählt, wie er Eingabefolgen als Historie besitzt.

Daher ist der Wert $1,3863 \cdot \log(n) - 1,8456$ nicht das mittlere Level eines Knotens ML_n , wenn man zufällig einen binären Baum und in ihm zufällig einen Knoten auswählt.

Wir beenden hiermit den Exkurs über die Entropie von Bäumen, die sich auf andere Datenstrukturen übertragen lässt. Die Untersuchung ergab zugleich eine Klärung über die auf Folgen (Historien) bezogene mittlere Suchdauer bei binären Bäumen, die deutlich kleiner ist als das mittlere Level eines beliebigen Knotens in einem beliebigen binären Baum.

Interessanterweise sind AVL-Bäume sowie gewichts-balancierte Bäume Strukturen mit einer hohen Entropie. Bei einer hohen Entropie erreicht man im Mittel nach der Anwendung einer Operation schneller eine äquivalente Struktur, als wenn eine geringe Entropie vorliegt. Daher lassen sich AVL-Bäume und andere entropiereiche Strukturen leicht (mit $O(\log(n))$ als Aufwand) beim Ein- oder Ausfügen in eine gleichartige Struktur überführen, während beispielsweise lineare Listen beim Ein- oder Ausfügen einen Aufwand von $O(n)$ erfordern.

Man kann nun je nach Anwendungsproblem nach Strukturen suchen, die ihre eigene Historie mitspeichern oder rasch vergessen und unter dieser Nebenbedingung sehr effizient bearbeitet werden können. Hier gibt es noch viel Unbekanntes zu entdecken.

Anhang zu Kapitel 8:

8.8 Weitere Definitionen zu Graphen

8.9 Sonstiges

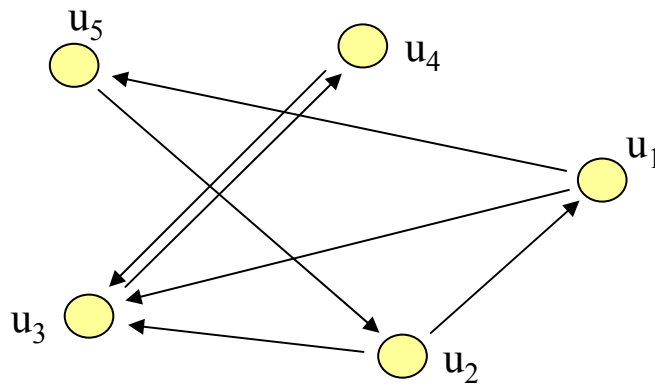
8.8 Weitere Definitionen zu Graphen

Graphen wurden in Abschnitt 3.8 auf 36 Folien ausführlich erläutert. Gehen Sie bitte jene Folien nochmals genau durch. Die Begriffe gerichteter und ungerichteter Graph, adjazent bzw. benachbart, inzident, (induzierter) Teilgraph, Grad, Weg, Kreis, Länge von Wegen, erreichbar, azyklisch, starke und schwache Zusammenhangskomponente, Adjazenzmatrix, Adjazenzliste (mit zugehöriger Datenstruktur), transitive Hülle und Graphdurchlauf sollten Ihnen gut vertraut sein.

Auf den folgenden Seiten ergänzen wir jene Definitionen um Begriffe, die zum Teil in diesem Kapitel aufgetreten sind. Wir werden sie in den Übungen vertiefen. Hiermit werden zugleich die Graphalgorithmen in Kapitel 11 vorbereitet.

8.8.1 Erinnerung zur Darstellung von Graphen:

Graphen kann man durch ihre *Adjazenzmatrix* A , durch *Adjazenzlisten* oder durch *Inzidenzlisten* darstellen (siehe 3.8.5 g).



Diesen Graphen stellen wir als Adjazenzmatrix und als Adjazenzliste dar.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Adjazenzmatrix A

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

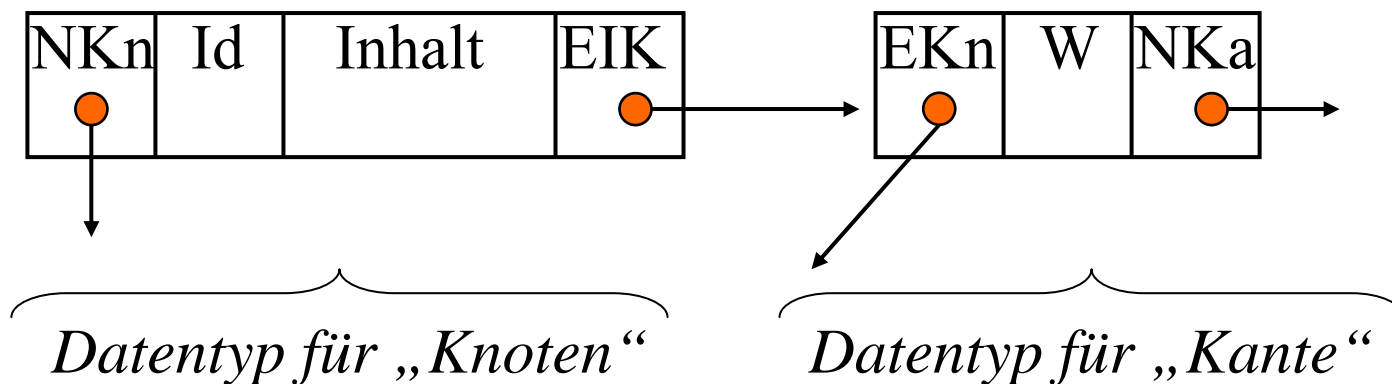
Erweiterte
Adjazenzmatrix A'

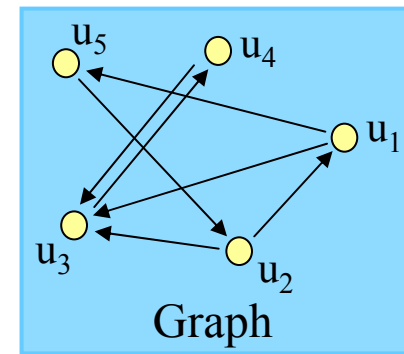
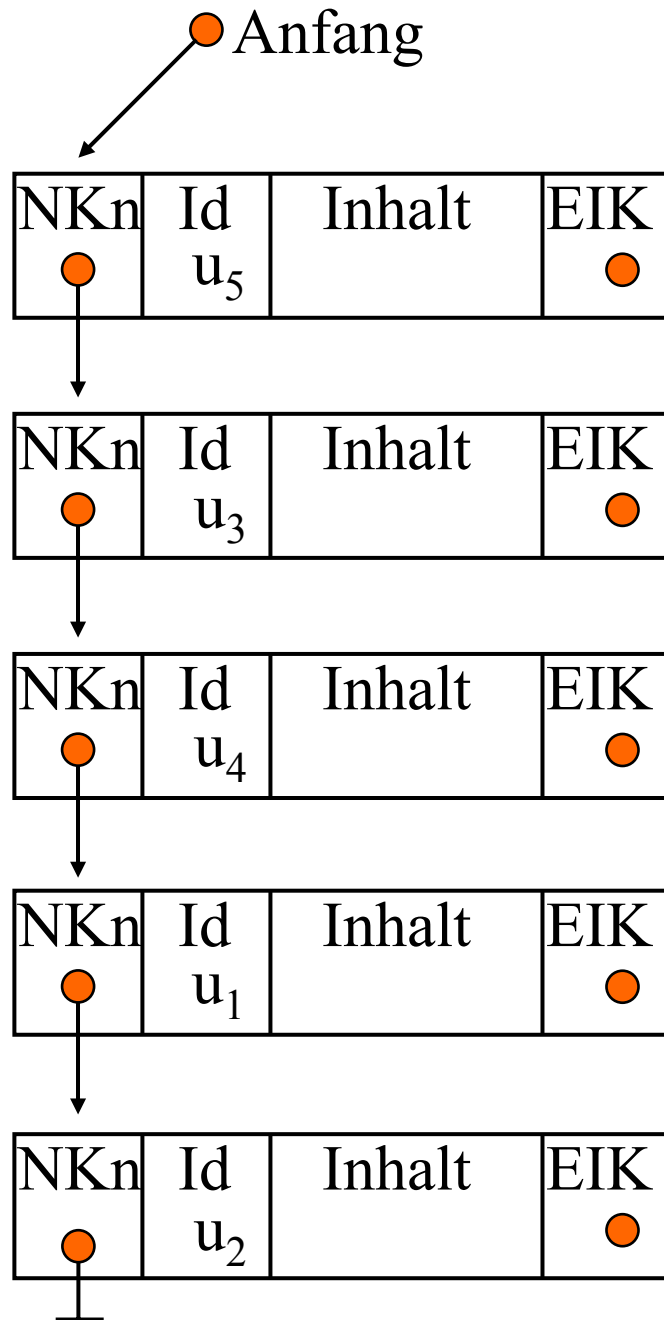
Darstellung als Adjazenzliste (siehe 3.8.6)

Jeder *Knoten* erhält einen Identifikator *Id* (z.B. einen Bezeichner oder eine natürliche Zahl) und einen Inhalt. Die Knoten werden in einer Liste zusammengefasst und besitzen neben *Id* und Inhalt weitere Komponenten:

- Verweis auf den **N**ächsten **K**noten in der Liste "NK_n",
- Verweis auf die **E**rste **I**nzidente **K**ante "EIK".

Jede von einem Knoten ausgehende *Kante* muss enthalten: den "Endknoten der Kante" (EK_n), ihren Wert *W* ("weight") und einen Verweis auf die nächste Kante "NK_a".

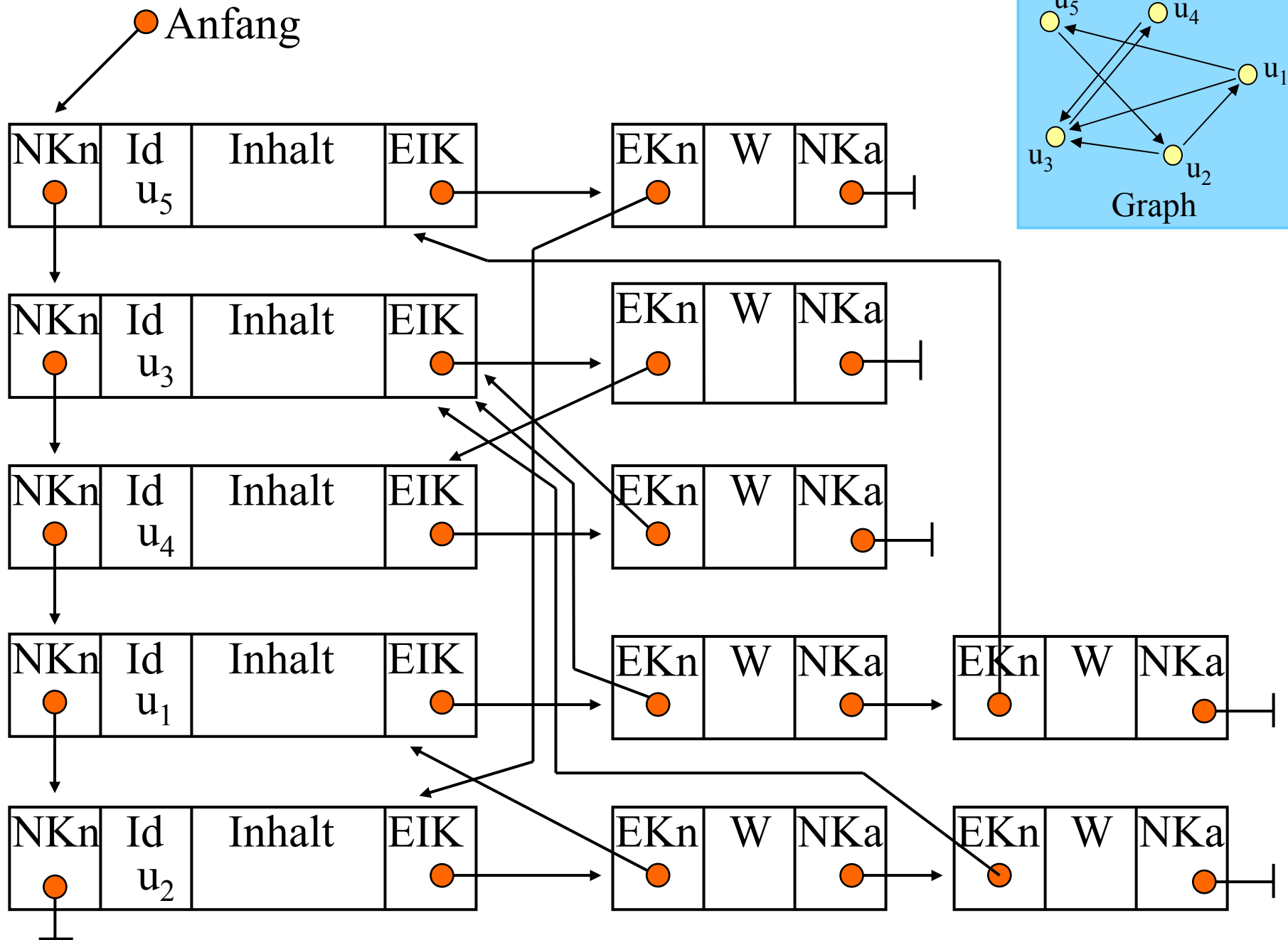




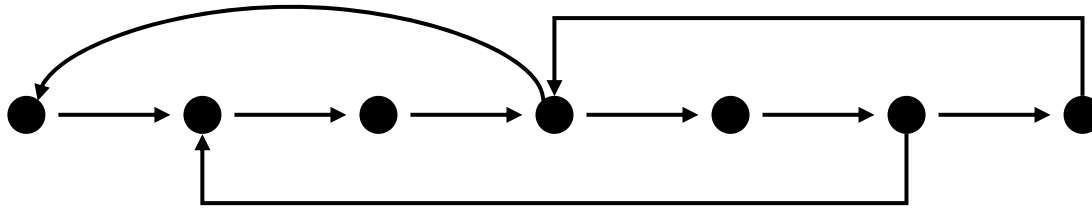
Zunächst werden die Knoten in beliebiger Reihenfolge in einer Liste gespeichert (siehe links).

Danach (siehe nächste Folie) werden die Kanten über die EIK-Listen eingetragen.

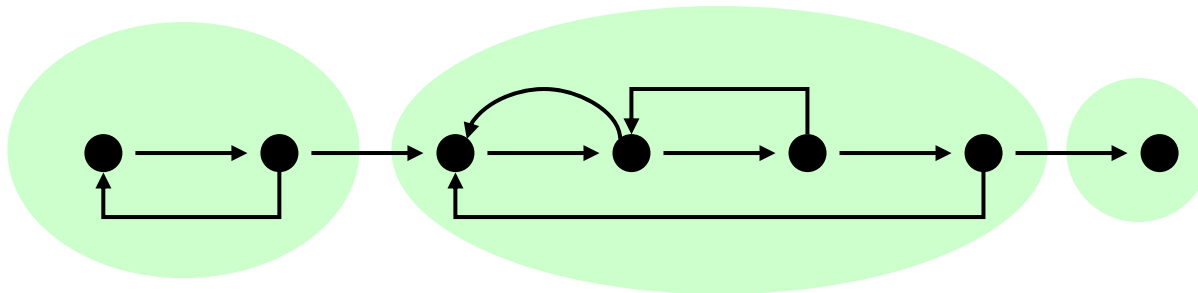
Wir fügen hier keine Werte für die Komponenten „Inhalt“ und „W“ ein.



Erinnerung: Zusammenhang im gerichteten Fall



Dieser Graph ist stark zusammenhängend.



Dieser Graph besitzt drei starke Zusammenhangskomponenten.

Vgl. hierzu 3.8.14.

Definition 8.8.2: Topologische Sortierung von DAGs

Zu einem Graphen $G=(V, E)$ heißt $G^*=(V, E^*)$ die transitive Hülle, wenn im ungerichteten Fall gilt

$$E^* = \{ \{u,v\} \mid u \neq v \text{ und es gibt einen Weg von } u \text{ nach } v \text{ in } G \}$$

bzw. im gerichteten Fall gilt

$$E^* = \{ (u,v) \mid u \neq v \text{ und es gibt einen Weg von } u \text{ nach } v \text{ in } G \}.$$

Stets gilt natürlich $E \subseteq E^*$.

Es sei $G=(V,E)$ gerichtet. Eine Abbildung $\text{ord}: V \rightarrow \mathbb{N}$ mit

$$\forall u, v \in V \text{ mit } u \neq v \text{ gilt: } (u,v) \in E^* \Rightarrow \text{ord}(u) < \text{ord}(v)$$

heißt topologische Sortierung von G .

Man ordnet also die Knoten so an, dass jeder von u aus erreichbare Knoten eine höhere Nummer als u bekommt.

Genau jeder azyklische gerichtete Graph („DAG“) besitzt eine topologische Sortierung; sie ist nicht eindeutig (selbst am Beispiel klar machen!).

Definition 8.8.3: Markierte oder gewichtete Graphen

In Anwendungen sind Graphen meist "markiert" oder "gewichtet", d.h., ihre Knoten und/oder ihre Kanten sind mit Werten ("Markierung" oder "Gewicht") aus einer Wertemenge W bzw. W' versehen: $\mu: V \rightarrow W$ und $\delta: E \rightarrow W'$.

Man schreibt $G=(V, E, \mu)$ bzw. $G=(V, E, \delta)$ bzw. $G=(V, E, \mu, \delta)$. Meist sind die Markierungen ganze oder reelle Zahlen. Oft hat man mehr als nur zwei solche Abbildungen.

Kantenmarkierungen $\delta: E \rightarrow \mathbb{R}^{\geq 0}$ mit
 $\mathbb{R}^{\geq 0}$ = Menge der nichtnegativen reellen Zahlen
bezeichnet man auch als Entfernungen.

[Die Summe aller Entfernungen nennt man das Gewicht des Graphens, siehe unten bei "minimaler Spannbaum".]

Definition 8.8.4: Länge von Wegen bzgl. δ , Abstand

Sei $G=(V, E, \delta)$ ein Graph. Die reellwertige Abbildung $\delta: E \rightarrow \mathbb{R}$ wird auf die Menge der Wege fortgesetzt durch

$$\delta((u_0, u_1, \dots, u_k)) := \delta((u_0, u_1)) + \delta((u_1, u_2)) + \dots + \delta((u_{k-1}, u_k)).$$

Dieser Wert heißt die Länge des Weges (u_0, u_1, \dots, u_k) .

Sei $\delta: E \rightarrow \mathbb{R}^{\geq 0}$ (= Menge der nichtnegativen reellen Zahlen), dann bezeichnet man für den Graphen $G=(V, E, \delta)$ die (kürzeste Entfernung) oder den Abstand oder die Distanz eines Knotens u zum Knoten v den Wert $\delta: V \times V \rightarrow \mathbb{R}^{\geq 0} \cup \{\infty\}$ mit

$$\delta(u, v) = 0, \text{ für } u=v,$$

$$\delta(u, v) = \text{Min } \{ \delta((u_0, u_1, \dots, u_k)) \mid u=u_0, v=u_k \}, \text{ sofern es mindestens einen Weg von } u \text{ nach } v \text{ gibt,}$$

$$\delta(u, v) = \infty, \text{ falls es keinen Weg von } u \text{ nach } v \text{ gibt.}$$

Statt $\delta(u, v)$ schreibt man oft $\text{dist}(u, v)$ oder $d(u, v)$.

Die maximale Distanz in einem Graphen bezeichnet man auch als den Durchmesser des Graphen.

8.8.5: Kürzeste Wege

Die Aufgabe, den Abstand $\text{dist}(u,v)$ vom Knoten u zum Knoten v und einen Weg von u nach v mit der Länge $\text{dist}(u,v)$ zu ermitteln, bezeichnet man als das Kürzeste-Wege-Problem.

Hierbei unterscheidet man die Probleme:

SSSP = single source shortest paths

= alle kürzesten Wege, die von einem gegebenen Knoten zu jedem anderen Knoten führen,

SPSP = single pair shortest path

= kürzester Weg zwischen zwei gegebenen Knoten u und v

APSP = all pair shortest paths

= alle kürzesten Wege zwischen allen Paaren (u,v) von Knoten

Definition 8.8.6: (minimaler) Spannbaum, Gewicht

Sei $G=(V, E)$ ein zusammenhängender gerichteter oder ungerichteter Graph. Ein (gerichteter bzw. ungerichteter) Teilgraph $B=(V, E_B)$, der ein Baum ist, heißt Spannbaum oder aufspannender oder spannender Baum von G (beachte: in B kommen alle Knoten des Graphens vor).

Es sei $G=(V, E, \delta)$ ein Kanten-markierter Graph mit $\delta: E \rightarrow \mathbb{R}$. Die Summe aller seiner Entfernungen $\delta(G)$

$$\delta(G) := \sum_{e \in E} \delta(e)$$

heißt das Gewicht des Graphen G .

Ein Spannbaum $B=(V, E_B, \delta)$ des Graphen $G=(V, E, \delta)$ heißt minimaler Spannbaum (engl.: minimal spanning tree) von G , wenn $\delta(B) \leq \delta(B')$ für alle Spannbäume B' von G gilt.

Definition 8.8.7: Hamiltonsche und Eulersche Wege

Ein Weg $(u_0, u_1, \dots, u_{n-1})$ in einem Graphen G mit n Knoten heißt Hamiltonscher Weg, wenn er jeden Knoten genau einmal enthält. Ein Weg $(u_0, u_1, \dots, u_{n-1}, u_0)$ heißt Hamiltonscher Kreis, wenn $(u_0, u_1, \dots, u_{n-1})$ ein Hamiltonscher Weg ist.

Ein Weg (u_0, u_1, \dots, u_k) heißt Eulerscher Weg, wenn jede Kante des Graphen genau einmal in ihm vorkommt. Er heißt Eulerscher Kreis, wenn zusätzlich $u_k = u_0$ ist.

Die Bestimmung Hamiltonscher Wege ist schwierig, diejenige Eulerscher Wege dagegen leicht, siehe unten.

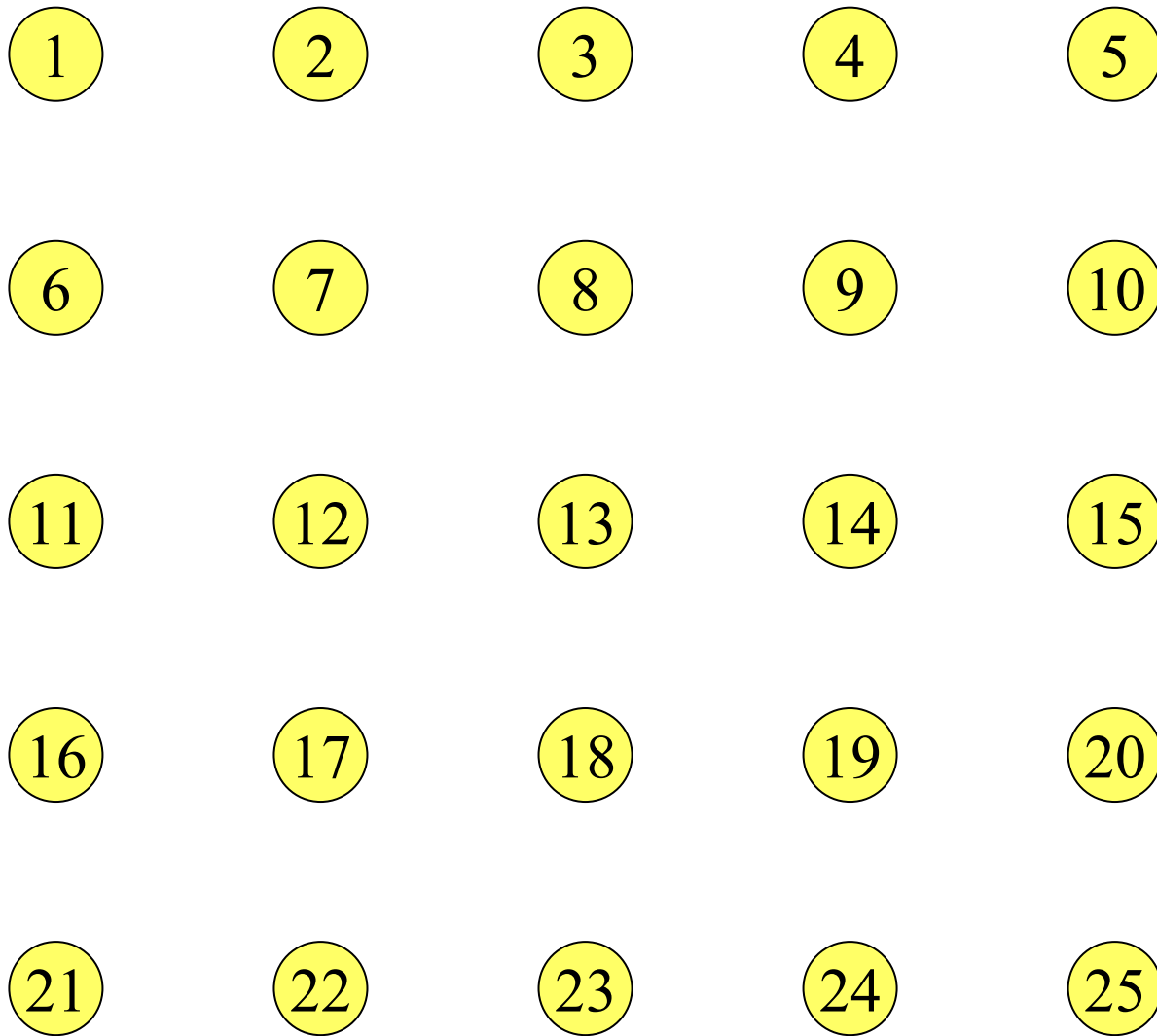
8.8.8 Einige Fragen und ihre Lösungen.

Wir wollen folgende Aussagen erläutern.

1. Die Zahl der verschiedenen (doppelpunktfreien) Wege zwischen zwei festen Knoten u und v in einem Graphen mit n Knoten und höchstens $2n$ Kanten kann bereits exponentiell wachsen. Wir geben ein Beispiel mit mehr als $4^{\sqrt{n}}$ solchen Wegen an („Gitter“).

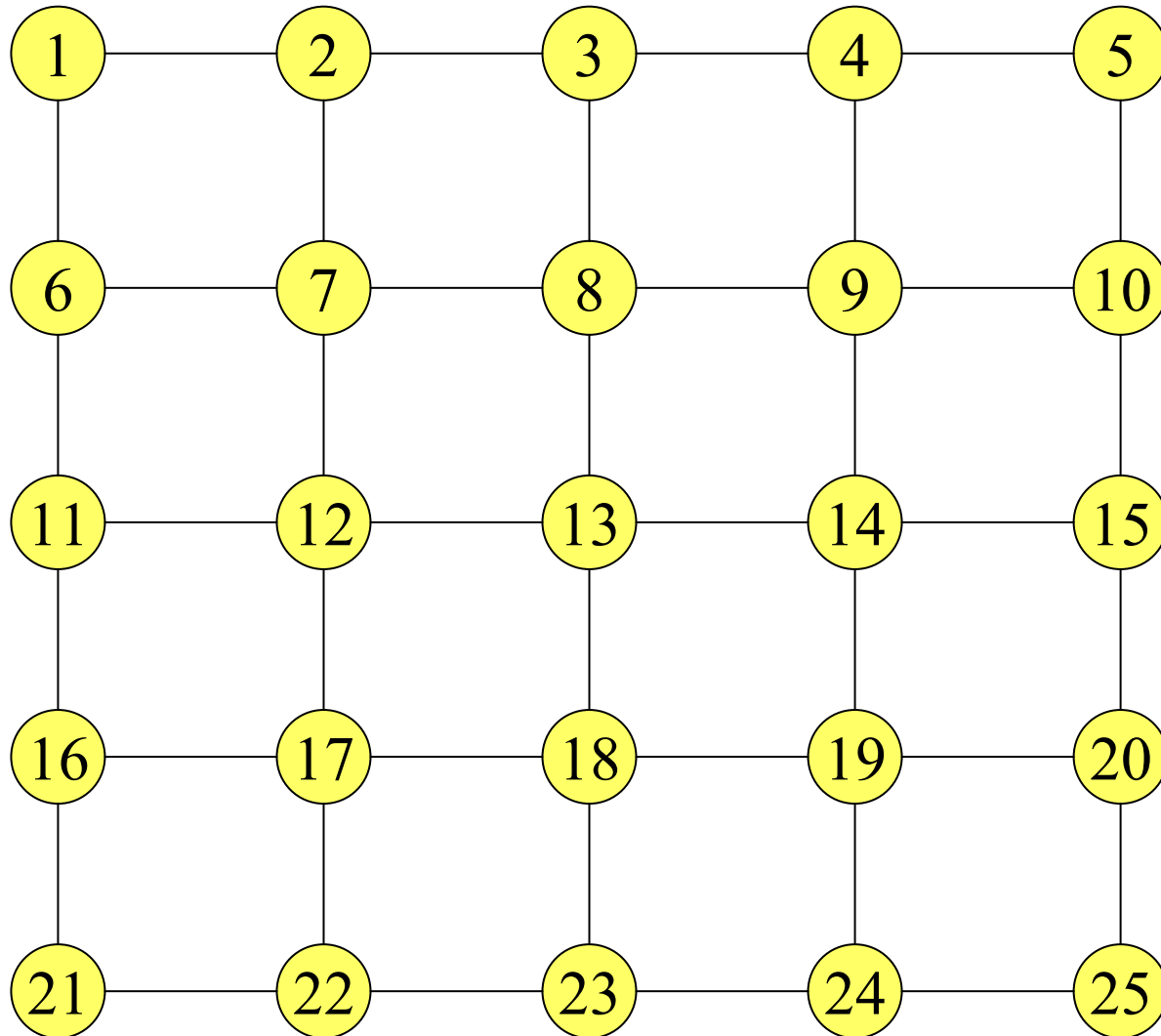
2. Hamiltonsche Wege/Kreise sind nicht leicht zu finden.
(Manche von Ihnen werden später beweisen, dass dieses Problem „NP-hart“ ist und daher nach heutiger Meinung nur mit exponentiellem Aufwand gelöst werden kann.)

3. Eulersche Wege/Kreise sind dagegen leicht zu finden.
(Wir erläutern dies nur an einem Beispiel. Den Beweis des Satzes sollten Sie selbst versuchen oder in einem Lehrbuch nachlesen.)



Wir betrachten im Folgenden diese Knotenmenge $\{1, 2, 3, \dots, 25\}$.

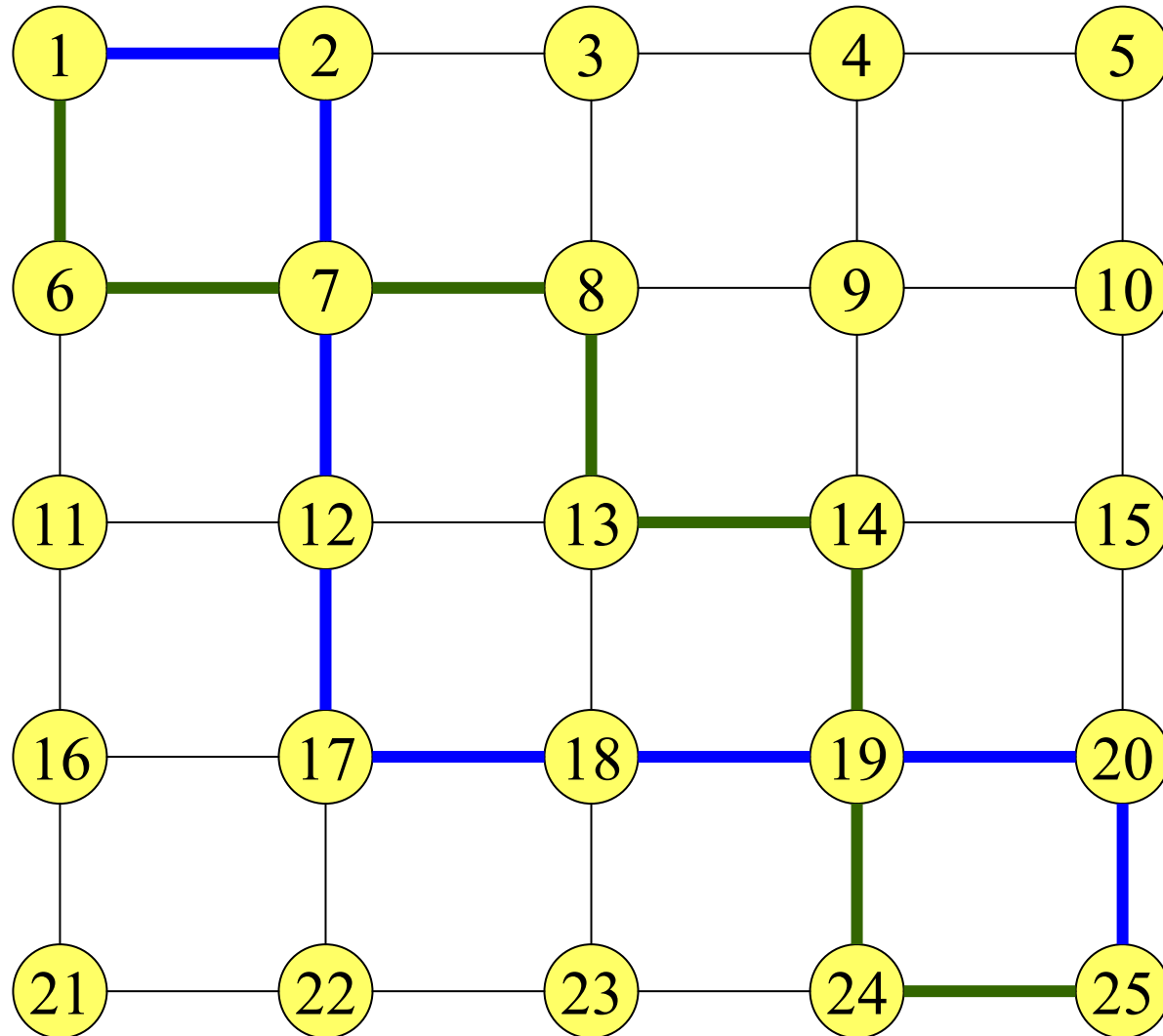
zu 1. Berechne die Anzahl von Wegen zwischen zwei Knoten



Als
Beispiel
wählen
wir ein
ungerich-
tetes
„Gitter“.

Knoten:
25.
Kanten:
40.

Wie viele verschiedene Wege gibt es von Knoten 1 nach Knoten 25?



Knoten:
25.
Kanten:
40.

Es sind *mindestens*

$$\binom{10}{5} = (10 \cdot 9 \cdot 8 \cdot 7 \cdot 6) / (1 \cdot 2 \cdot 3 \cdot 4 \cdot 5) = 252 \text{ Wege.}$$

Allgemein: Wenn ein $k \times k$ -Gitter mit $k^2 = n$ vielen Knoten gegeben ist, so gibt es mindestens $\binom{2k}{k}$ doppelpunktfreie Wege zwischen den beiden Knoten 1 und k^2 .

Warum? Man kann genau k -mal eine Kante nach rechts („r“) und genau k -mal eine Kante nach unten („u“) gehen. Die Reihenfolge ist beliebig. Dies gibt eine Folge der Länge $2k$ bestehend aus je k Buchstaben „r“ und „u“. Man kann aus den $2k$ Positionen beliebig k Stellen für den Buchstaben „r“ auswählen (auf die restlichen Stellen kommt dann der Buchstabe „u“). Man erhält genau „ $2k$ über k “ Möglichkeiten. Dies ist aber nur ein Bruchteil der tatsächlichen Möglichkeiten, wie man sich leicht überlegt.

Man beachte: Die Zahl der Wege wächst exponentiell mit n . Denn es ist:

$$\binom{2k}{k} = \frac{(2k)!}{k! \cdot k!}$$

Mit der Stirlingschen Formel für die Fakultät

$$k! \approx \left(\frac{k}{e}\right)^k \cdot \sqrt{2 \cdot \pi \cdot k}$$

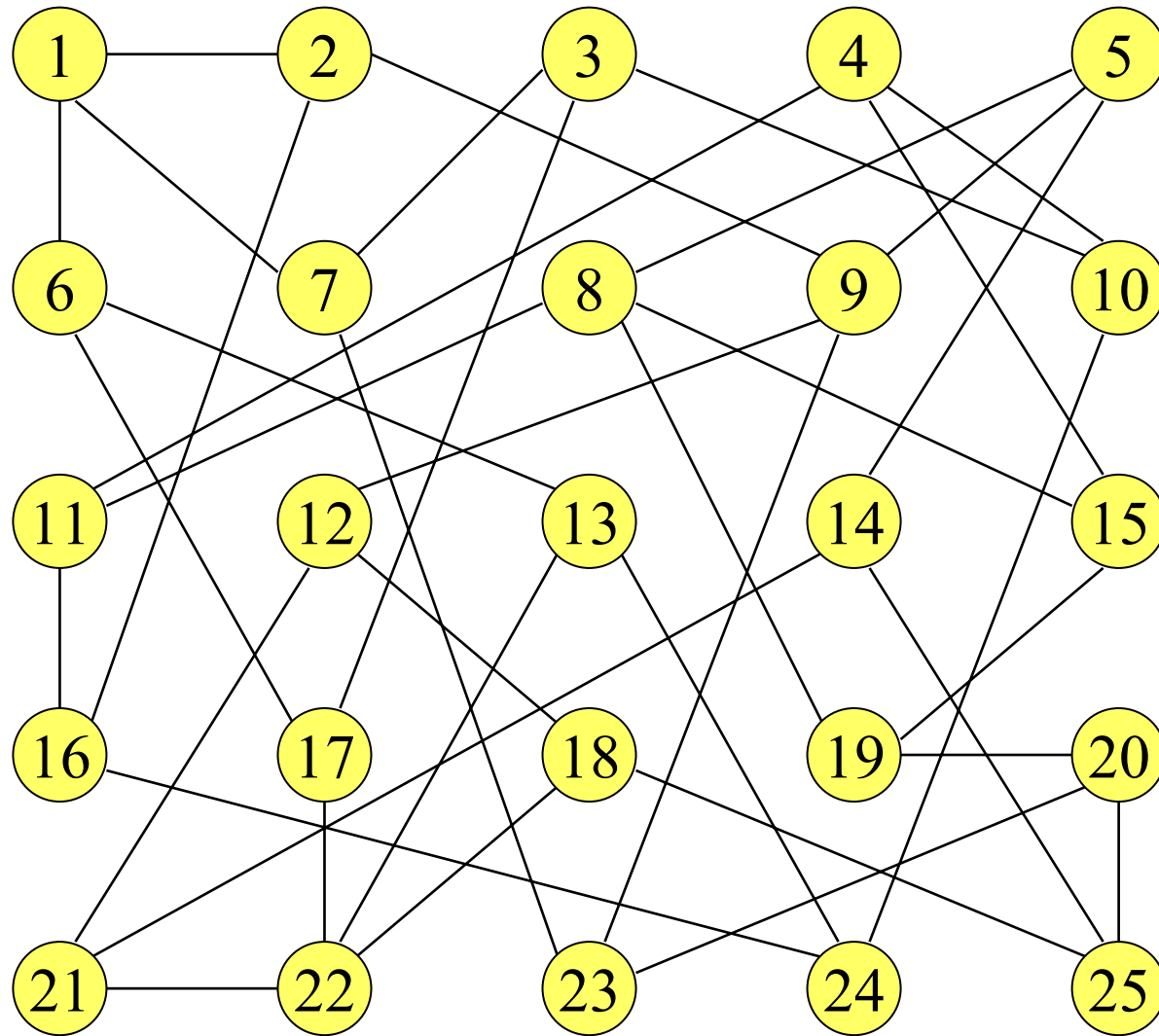
mit $e = 2,71828182845904\dots$
und $\pi = 3,14159265358979\dots$

folgt hieraus durch Einsetzen und Ausrechnen:

$$\binom{2k}{k} \approx \frac{4^k}{\sqrt{\pi \cdot k}}$$

mit $k^2 = n$.

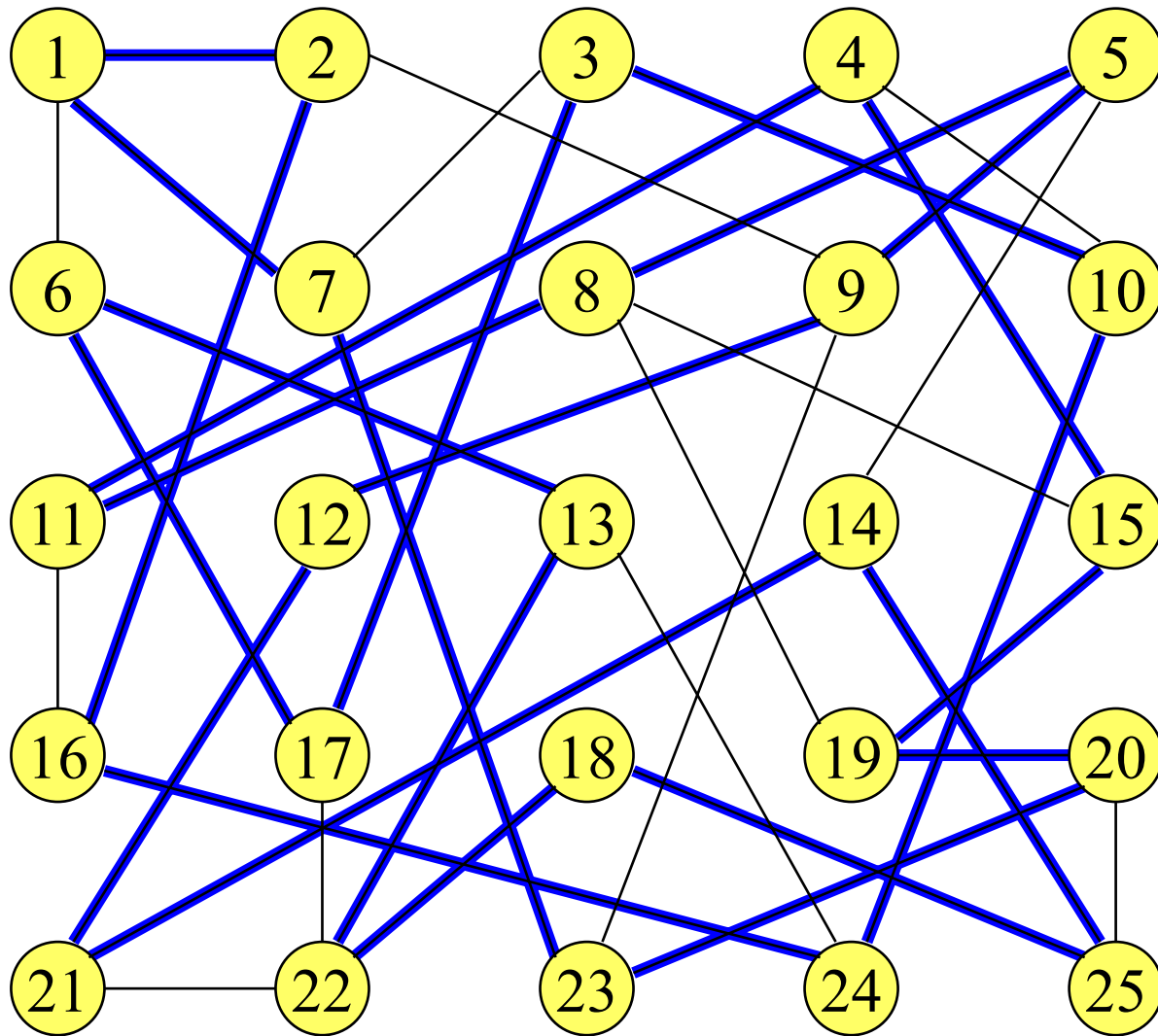
zu 2. Finde einen Weg, der jeden Knoten genau einmal besucht



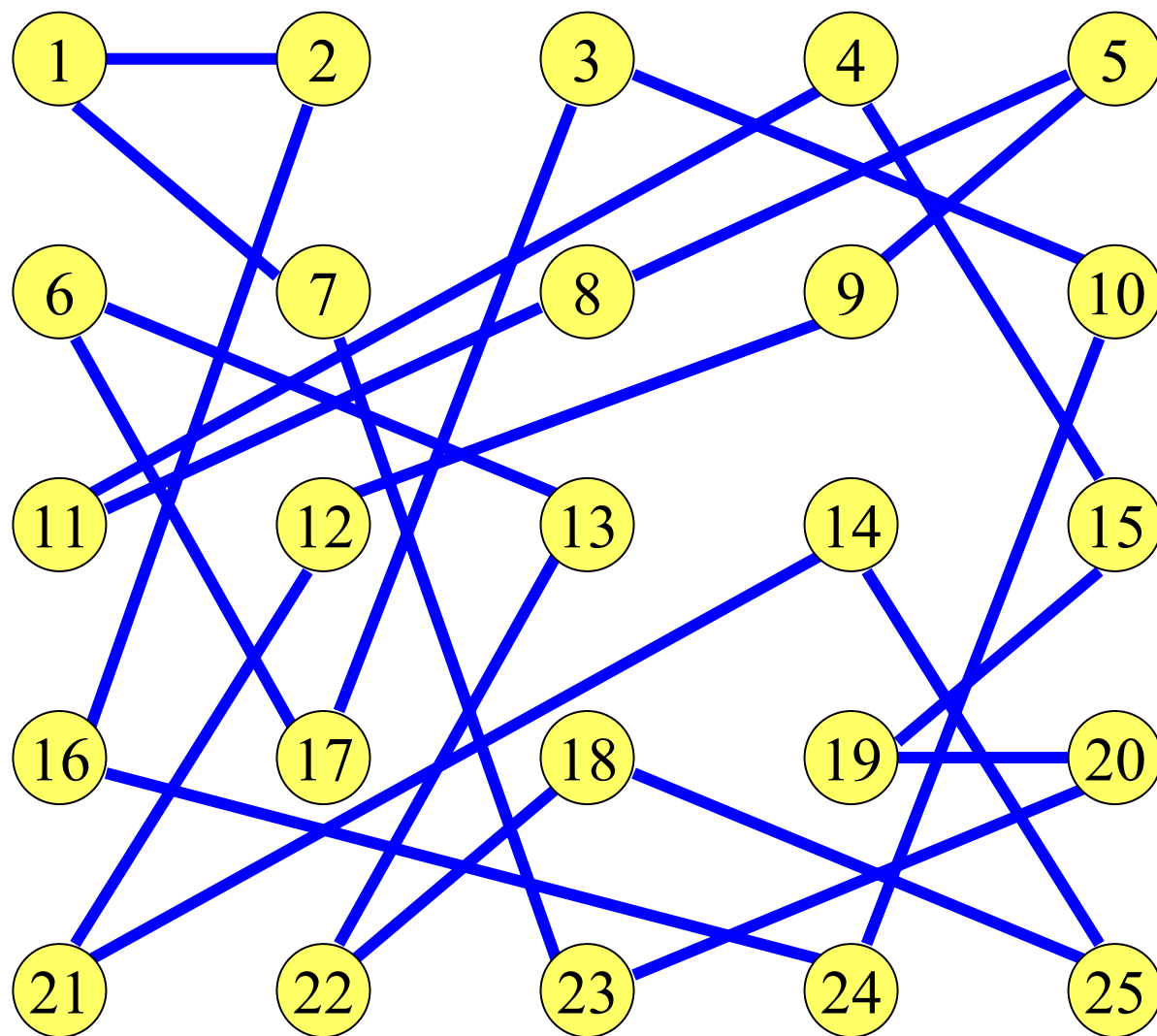
Knoten:
25

Kanten:
39

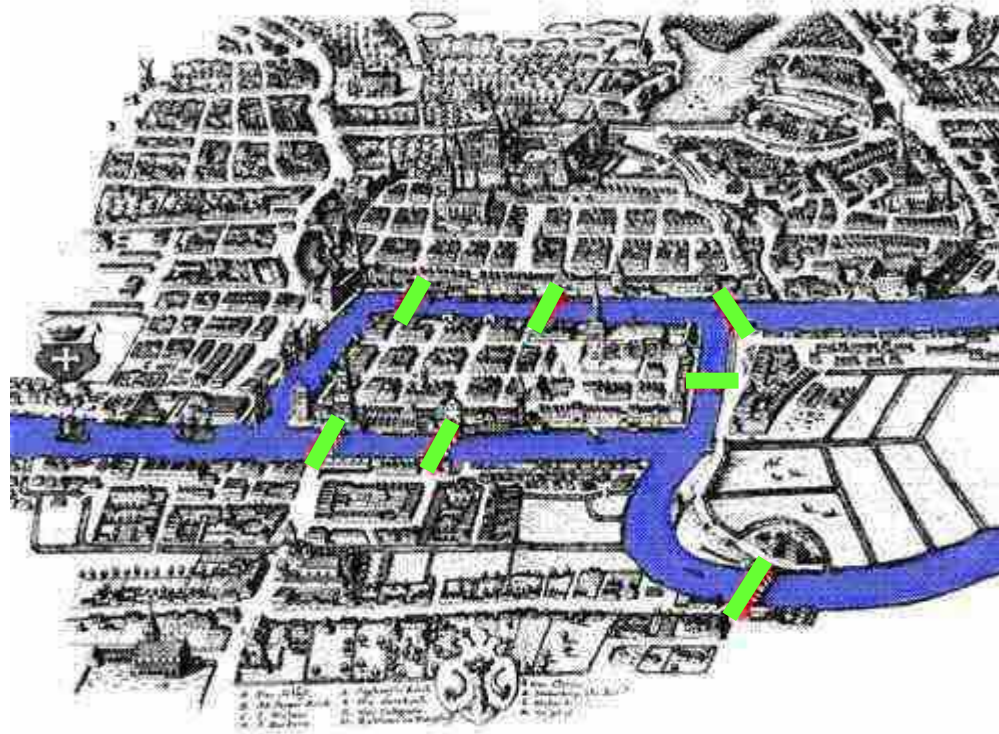
Gibt es in diesem Graphen einen Hamiltonschen Kreis? **Ja.**



Kreis: 1, 2, 16, 24, 10, 3, 17, 6, 13, 22, 18, 25, 14, 21, 12, 9, 5, 8, 11, 4, 15, 19, 20, 23, 7, 1

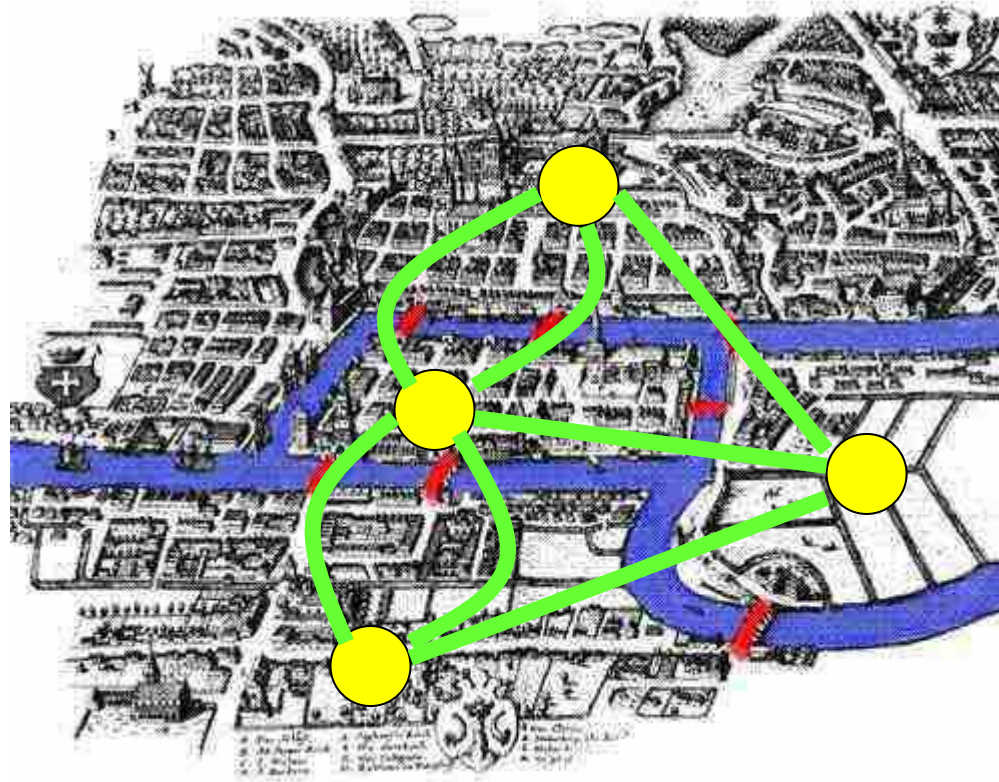


zu 3. Eulersche Kreise: Das Königsberger Brückenproblem (1736)

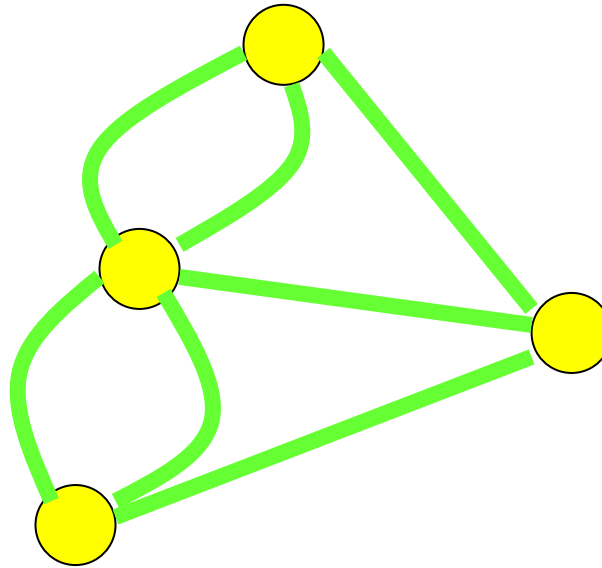


Kann man einen Rundgang durch Königsberg machen, so dass man jede der 7 Brücken genau einmal überquert und am Ende wieder am Startpunkt ankommt? Dieses Problem gilt als der Beginn der Graphentheorie.

Konstruiere hierzu einen Graphen (mit Mehrfachkanten)

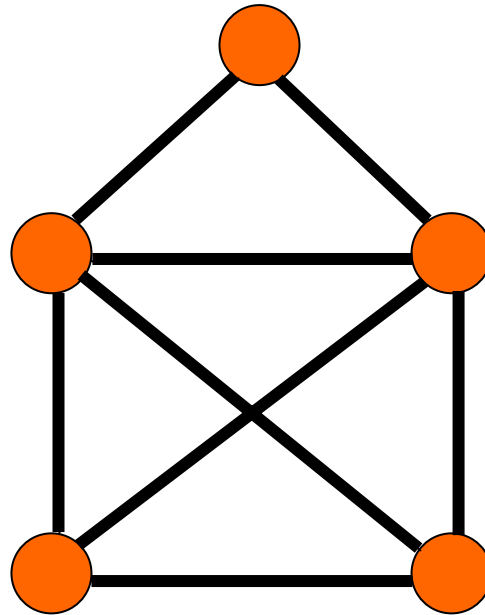


Satz: Ein zusammenhängender ungerichteter Graph besitzt genau dann einen Eulerschen Kreis, wenn alle seine Knoten einen geraden Grad haben.



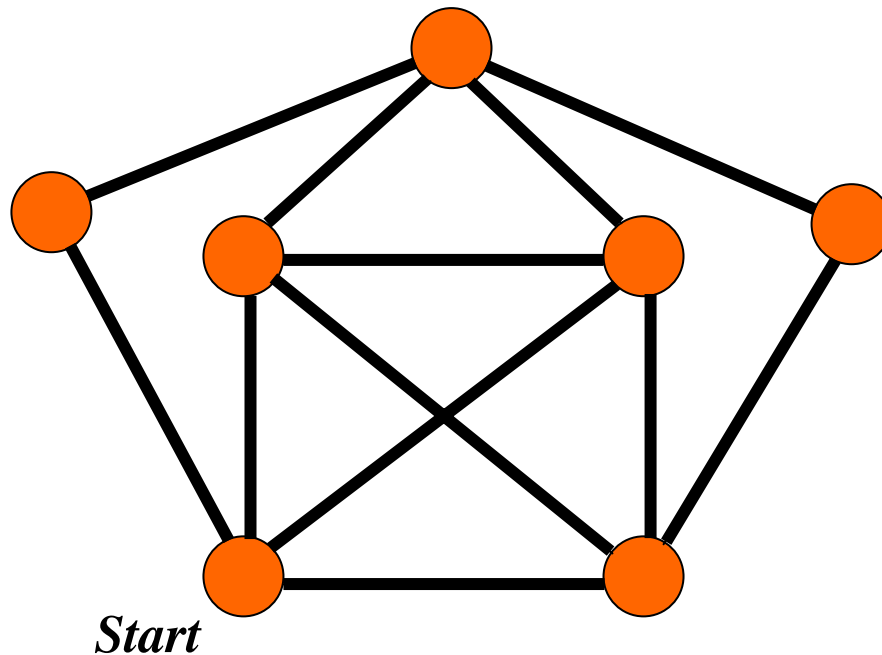
Eulers Beweis ergab: Es gibt keinen Rundgang durch Königsberg, auf dem jede Brücke genau einmal benutzt wird.

Zusatz: Haben genau zwei Knoten einen ungeraden Grad, so hat der Graph einen Eulerschen Weg, aber keinen Eulerschen Kreis.



Beispiel: "Dies ist das Haus des Nikolaus".

Durch Hinzunahme eines oder zweier Knoten mit geeigneten Kanten erhalten wir hieraus einen Graphen mit einem Eulerschen Kreis.



8.8.9 Durchsuchen eines Graphens

Gesucht ist also ein Algorithmus, der an irgendeinem Knoten u beginnt und dann alle von u aus erreichbaren Knoten und Kanten besucht.

Im ungerichteten Fall wird hierbei genau die zu u gehörige Zusammenhangskomponente $G(u) = (Z(u), E'(u))$ durchlaufen, siehe 3.8.9. Im gerichteten Fall wird der von u aus erreichbare Teil der schwachen Zusammenhangskomponente (speziell: alle Knoten v mit $(u, v) \in E^*$, siehe transitive Hülle) besucht. Werden auf diese Weise nicht alle Knoten des Graphens erreicht, so muss man dieses Vorgehen mit irgendeinem bisher noch nicht besuchten Knoten fortsetzen. Vergleiche hierzu 3.8.11 und 12.

Wir fassen das bereits Bekannte zusammen und ergänzen es. Zum Durchlauf von Bäumen siehe Abschnitt 3.7.6.

Ausgehend von einem Knoten u werden die Knoten und Kanten meist nach zwei Strategien aufgesucht:

Tiefensuche (**DFS** = depth first search): Erreicht man einen Knoten v , so wird ab hier rekursiv weiter gesucht, indem man allen von v ausgehenden Kanten folgt. Stößt man hierbei auf einen bereits besuchten Knoten, so wird in dieser Richtung nicht weiter gesucht (Abbruch der Rekursion).

Breitensuche (**BFS** = breadth first search): Man durchläuft den Graphen, ausgehend von u , schalenförmig gemäß des Abstandes, d.h., man besucht zunächst alle Knoten v mit dem Abstand $\text{dist}(u,v) = 1$, dann alle Knoten v mit dem Abstand $\text{dist}(u,v) = 2$ usw.

Tiefensuche (**DFS** = depth first search) umgangssprachlich.

Gerichteter Fall :

```
procedure besuche(u) is  
begin  "bearbeite den Knoten u;"  markiere u als besucht;  
      for alle v aus der Menge der Nachfolger S(u) loop  
        if v noch nicht besucht then besuche(v); end if;  
      end loop;  
end besuche;
```

Im ungerichteten Fall ersetzt man die Menge S(u) durch die Menge der Nachbarn N(u), siehe 3.8.5e.

Wir betrachten im Folgenden nur den gerichteten Fall.
Die Tiefensuche läuft im Prinzip wie im Algorithmus GD aus 3.8.7 ab. Zur Datenstruktur siehe oben bzw. 3.8.6.

Rekursives Ada-Programm zur *Tiefensuche*, gerichteter Fall.

```
procedure DFS (Anfang: in NextKnoten) is  
p: NextKnoten;                                -- Datentypen siehe am Ende von 3.8.6  
procedure besuche (u: in NextKnoten) is  
    edge: NextKante; v: NextKnoten;  
    begin u.Besucht := true; edge := u.EIK;  
        while edge /= null loop v := edge.EKn;  
            if not v.Besucht then besuche(v); end if;  
            edge := edge.NKa; end loop;  
    end besuche;  
begin p := Anfang;  
    while p /= null loop p.Besucht := false; p:=p.NKn; end loop;  
    p := Anfang;  
    while p /= null loop if not p.Besucht then besuche(p); end if;  
        p := p.NKn; end loop;  
end DFS;
```

Iteratives Programm zur *Tiefensuche*, gerichteter Fall. Zu den Kellern vgl. Abschnitte 4.3.3 und 4.4.3 (generisches Paket "Stack" sei hier sichtbar).

```
procedure DFS (Anfang: in NextKnoten) is  
p, v: NextKnoten; KK: new Stack (item => NextKnoten); edge: NextKante;  
begin p := Anfang;  
  while p /= null loop p.Besucht := false; p:=p.NKn; end loop;  
  p := Anfang ;  
  while p /= null loop  
    if not p.Besucht then Push(p, KK);  
      while not Iseempty(KK) loop  
        u := Top(KK); Pop(KK); u.Besucht := true; edge := u.EIK;  
        while edge /= null loop v := edge.EKn;  
          if not v.Besucht then push(v, KK); end if;  
          edge := edge.NKa;  
        end loop;  
      end loop;  
    end if;  
    p := p.NKn;  
  end loop;  
end DFS;
```


Iteratives Programm zur Breitensuche, gerichteter Fall: Wie Tiefensuche, aber ersetze den Keller durch eine Schlange! (blau = Unterschied zu DFS)

```
procedure BFS (Anfang: in NextKnoten) is  
  p, v: NextKnoten; "KK: new Schlange (NextKnoten);" edge: NextKante;  
  begin p := Anfang;  
    while p /= null loop p.Besucht := false; p:=p.NKn; end loop;  
    p := Anfang ;  
    while p /= null loop  
      if not p.Besucht then Enter (p,KK);  
        while not Iempty(KK) loop  
          u := First(KK); Remove(KK); u.Besucht := true; edge := u.EIK;  
          while edge /= null loop v := edge.EKn;  
            if not v.Besucht then Enter(v,KK); end if;  
            edge := edge.NKa;  
          end loop;  
        end loop;  
      end if;  
      p := p.NKn;  
    end loop;  
end BFS;
```

Tiefensuche mit der Adjazenzmatrix A als Darstellung:

```
procedure DFS_Adj is                                     -- n ist hier global
A: array (0..n-1, 0..n-1) of Integer;
Besucht: array(0..n-1) of Boolean;
procedure besuche (j: in 0..n-1) is
    begin Besucht(j) := true;
        for k in 0..n-1 loop
            if A(j,k) /= 0 and not Besucht(k)
                then besuche (k); end if;
        end loop;
    end besuche;
begin ...           -- die Adjazenzmatrix A möge aufgebaut worden sein
    for i in 0..n-1 loop Besucht(i) := false; end loop;
    for i in 0..n-1 loop
        if not Besucht(i) then besuche (i); end if; end loop;
end DFS_Adj;
```

Zur Zeitkomplexität:

Adjazenzlistendarstellung: Jeder Durchlauf besucht jede Kante genau einmal. Jeder Knoten wird so oft besucht, wie Kanten auf ihn verweisen, mindestens aber einmal. Also ist die Zeitkomplexität linear $O(n+m)$.

Adjazenzmatrix: Da eine Matrix angelegt werden muss, beträgt der Aufbau des Graphens $\Theta(n^2)$ Schritte, auch wenn er nur relativ wenige Kanten besitzt. In der Prozedur wird bei jedem Aufruf von “besuche” die for-k-Schleife n Mal durchlaufen; weil “besuche” mindestens n Mal aufgerufen werden muss, ergibt sich eine Zeitkomplexität von $O(n^2)$.

Überlegen Sie, wie viele Schritte die Verfahren für einen vollständigen Graphen K_n und für einen Graphen mit n isolierten Knoten (also ein Graph ohne Kanten) benötigen.

8.9 Sonstiges

8.9.1: Anregung für analytisch Interessierte

Wie kann man einer Rekursionsgleichung ansehen, welche Lösungen sie hat? (siehe Herleitung von Satz 8.2.15)

Das lässt sich nicht pauschal beantworten. Denn schließlich ist dieses Problem nicht entscheidbar. Manchmal reicht es aber bereits, wenn man die Gleichung umformt (erweitern, einsetzen, in irgendeinem Sinne vereinfachen usw.).

Oft ist es hilfreich, die "Ableitung" zu betrachten, also $F(n) - F(n-1)$, oder die zweite Ableitung

$$(F(n) - F(n-1)) - (F(n-1) - F(n-2)) = F(n) - 2 F(n-1) + F(n-2).$$

Dies liefert einen Hinweis, wie die Lösung aussehen könnte, und man kann dann mit einem Lösungsansatz, den man in die Gleichung einsetzt, versuchen, eine Lösung aufzuspüren.

Wir betrachten als Beispiel die Rekursionsformel für $F(n)$:

$F(n) = (n+1)/n \cdot F(n-1) + (2n-1)/n$. Einfaches Umformen liefert:

$F(n) = F(n-1) + 1/n \cdot F(n-1) + (2n-1)/n$, d.h.:

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

Hier sieht man wenig, weil der Wert $F(n-1)$ noch auf der rechten Seite stehen geblieben ist.

Daher versuchen wir nun, die zweite Ableitung zu berechnen.

Aus obiger Formel für $F(n) - F(n-1)$ folgt:

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

Man subtrahiere die letzte Formel von der davor:

$(F(n) - F(n-1)) - (F(n-1) - F(n-2))$

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

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

$1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2)$ erinnert nun an die Rekursionsformel, denn dort steht (ersetze n durch $n+1$):

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

Also gilt:

$$1/n \cdot F(n-1) = 1/(n-1) \cdot F(n-2) + (2n-3)/(n \cdot (n-1)) \quad \text{bzw.}$$

$$1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) = (2n-3)/(n \cdot (n-1)) .$$

Dies setzt man oben ein:

$$\begin{aligned} & (F(n) - F(n-1)) - (F(n-1) - F(n-2)) \\ &= 1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) + 1/(n \cdot (n-1)) \\ &= (2n-3)/(n \cdot (n-1)) + 1/(n \cdot (n-1)) \\ &= (2n-2)/(n \cdot (n-1)) = 2/n \end{aligned}$$

Es entsteht ein überraschend einfacher Ausdruck. Nun erinnern wir uns an die Stammfunktionen aus der Analysis: $1/n$ ist die Ableitung des Logarithmus $\ln(n)$. Also müsste die "Ableitung" $\ln(n)$ und damit die Funktion F von der Form $n \cdot \ln(n)$ sein. Da keine kontinuierliche Funktion herauskommen wird, sollte man $H(n)$ anstelle von $\ln(n)$ nehmen, d.h., das Ergebnis könnte die Form $F(n) = a \cdot n \cdot H(n) + b \cdot n + c$ (mit Konstanten a, b, c) haben.

Dies setzt man nun in die ursprüngliche Gleichung

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

$$a \cdot n \cdot H(n) + b \cdot n + c$$

$$= (n+1)/n \cdot (a \cdot (n-1) \cdot H(n-1) + b \cdot (n-1) + c) + (2n-1)/n$$

Multiplikation mit n liefert:

$$a \cdot n^2 \cdot H(n) + b \cdot n^2 + c \cdot n$$

$$= (n+1) \cdot a \cdot (n-1) \cdot H(n-1) + b \cdot (n^2-1) + c \cdot (n+1) + (2n-1), \quad \text{also}$$

$$a \cdot n^2 \cdot H(n-1) + a \cdot n = a \cdot (n^2-1) \cdot H(n-1) - b + c + (2n-1)$$

$$a \cdot H(n-1) + a \cdot n = c - b + (2n-1).$$

Man sieht, dass diese Gleichung für Konstanten a , b und c nicht zu erfüllen ist. Man braucht offenbar im Ansatz ein weiteres Glied $d \cdot H(n)$. Neuer Ansatz:

$$F(n) = a \cdot n \cdot H(n) + b \cdot n + c + d \cdot H(n).$$

Wegen der Nebenbedingung $F(0)=0$ muss $c=0$ sein. Erneut einsetzen:

$$a \cdot n \cdot H(n) + b \cdot n + d \cdot H(n) \\ = (n+1)/n \cdot (a \cdot (n-1) \cdot H(n-1) + b \cdot (n-1) + d \cdot H(n-1)) + (2n-1)/n$$

Multiplikation mit n liefert:

$$a \cdot n^2 \cdot H(n) + b \cdot n^2 + d \cdot n \cdot H(n) \\ = a \cdot n^2 \cdot H(n-1) + a \cdot n + b \cdot n^2 + d \cdot n \cdot H(n) \\ = (n+1) \cdot a \cdot (n-1) \cdot H(n-1) + b \cdot (n^2-1) + d \cdot (n+1) \cdot H(n-1) + (2n-1) \\ = a \cdot (n^2-1) \cdot H(n-1) + b \cdot (n^2-1) + d \cdot n \cdot H(n-1) + d \cdot H(n-1) + (2n-1).$$

Umformen:

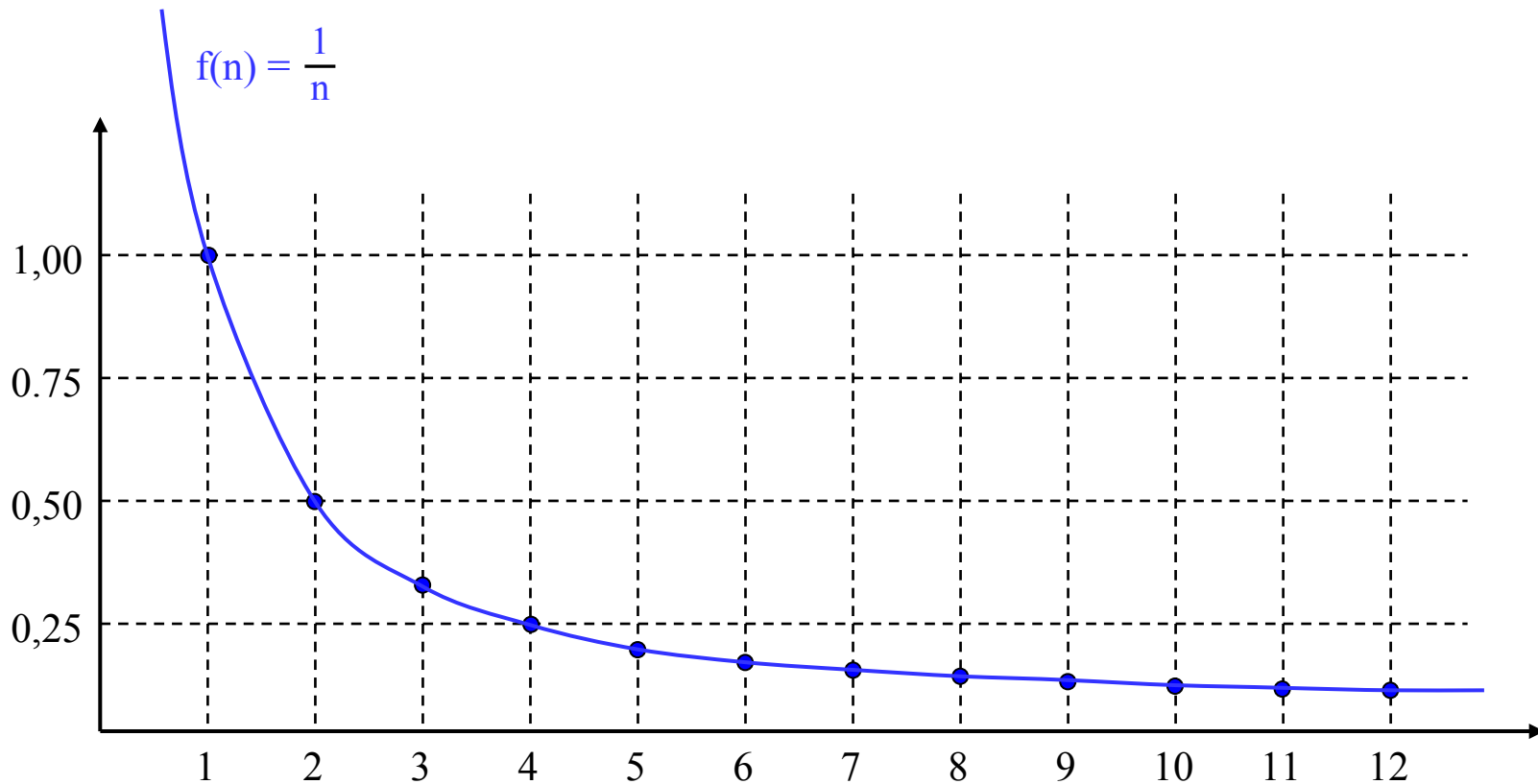
$$a \cdot H(n-1) + a \cdot n + b \cdot n^2 + d \cdot n \cdot H(n) \\ = b \cdot (n^2-1) + d \cdot n \cdot H(n-1) + d \cdot H(n-1) + (2n-1) \quad \text{wird zu} \\ a \cdot H(n-1) + a \cdot n + d = -b + d \cdot H(n-1) + (2n-1).$$

Wenn dies lösbar ist, so muss $a = d$ und $a = 2$ sein; es verbleibt:

$2 = -b - 1$, d.h., $b = -3$. Somit haben wir eine Lösung gefunden:

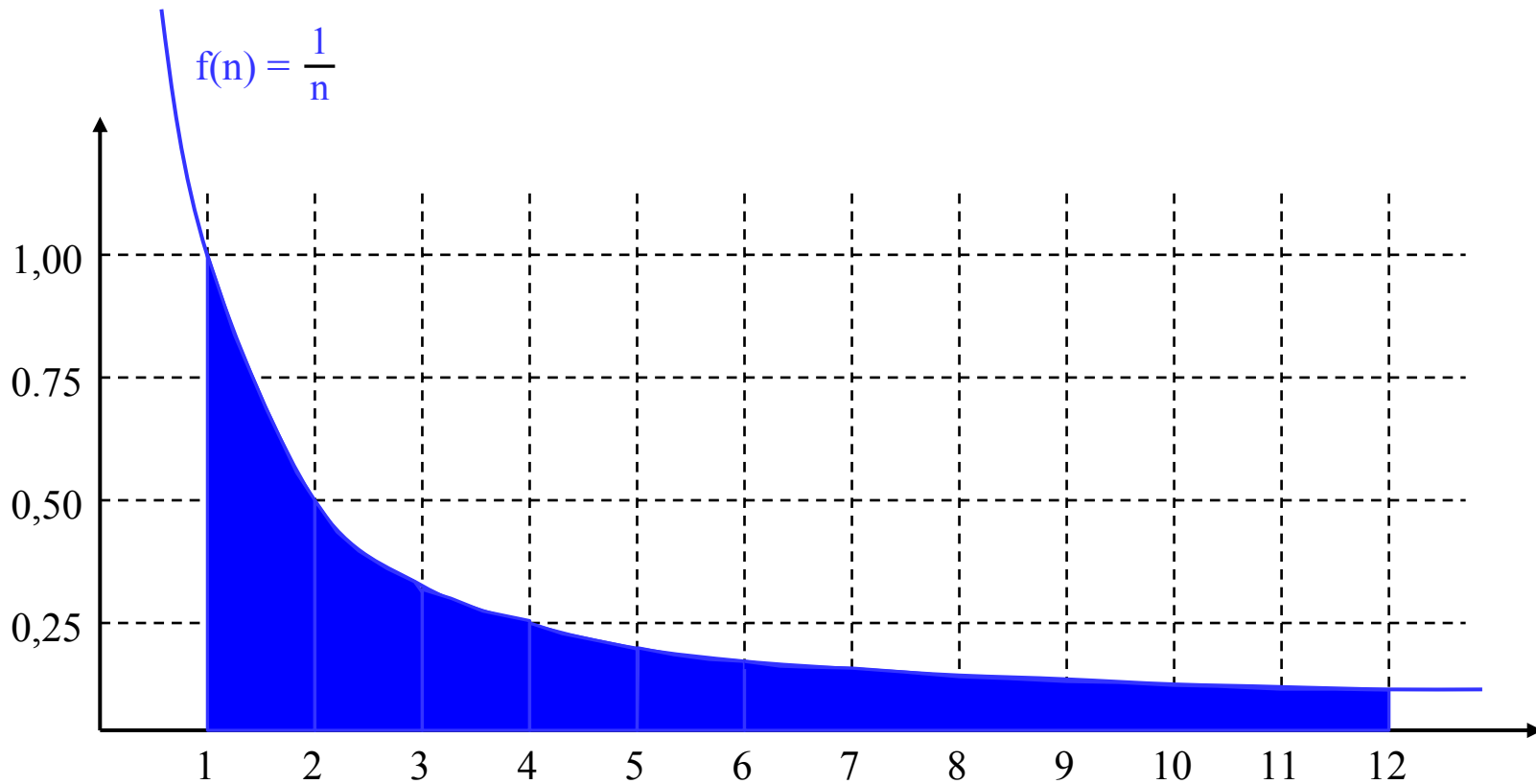
$F(n) = 2 \cdot n \cdot H(n) - 3 \cdot n + 2 \cdot H(n)$, also die gleiche Lösung wie im Beweis zu Satz 8.2.15.

8.9.2: Veranschaulichung der harmonischen Funktion 8.2.14



Eine ausführlichere Untersuchung steht in Abschnitt 1.5.

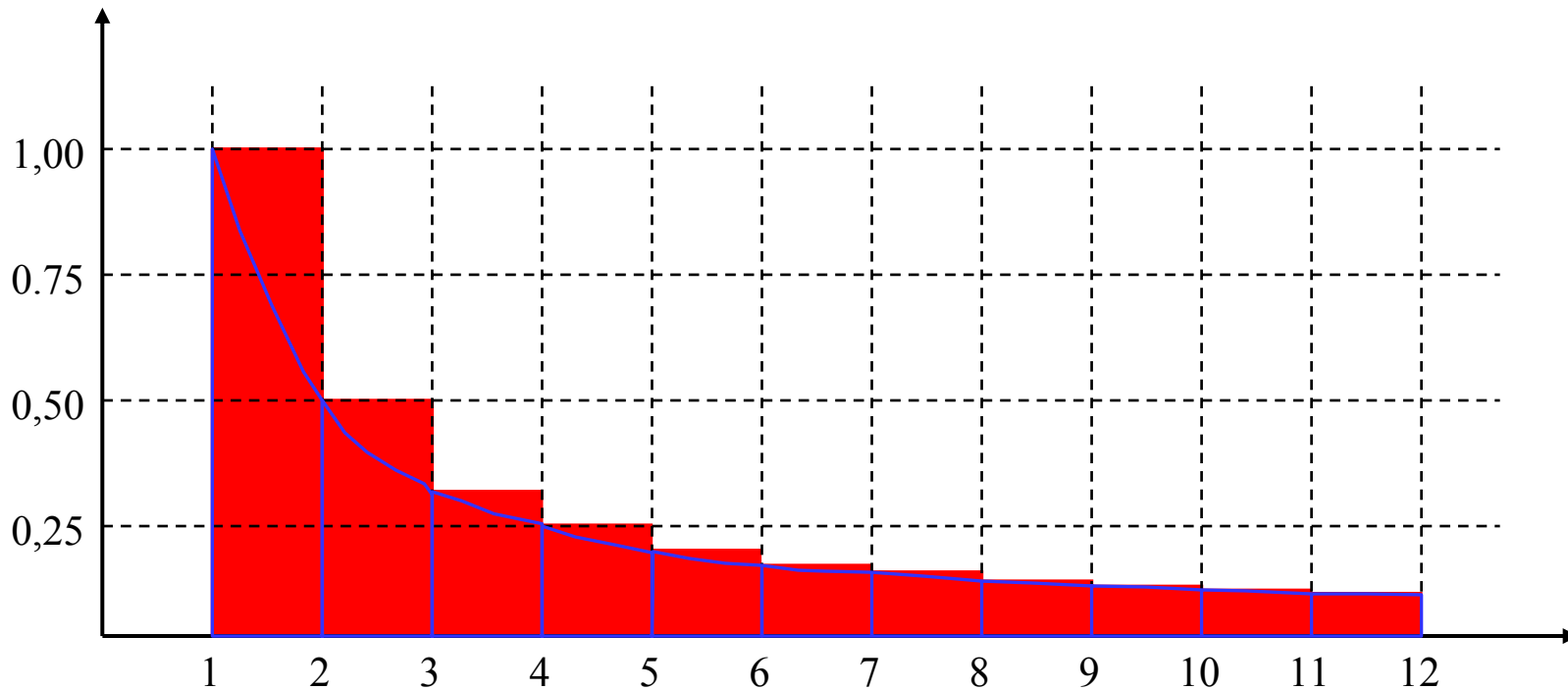
Abschätzung der harmonischen Funktion



Der natürliche Logarithmus $\ln(n)$ ist die Fläche unterhalb der Kurve von 1 bis n .

Abschätzung der harmonischen Funktion

$$f(n) = \frac{1}{n}$$



Der natürliche Logarithmus $\ln(n)$ ist die Fläche unterhalb der Kurve von 1 bis n.

Die harmonische Funktion $H(n-1)$ ist durch die rote Fläche von 1 bis n gegeben.

Man sieht: $H(n) - 1 \leq \ln(n) \leq H(n-1)$ für alle $n \geq 1$. Die kleinen über der blauen Kurve liegenden roten Flächenstücke summieren sich zur Eulerschen Konstanten $\gamma = 0,5772156649\dots$ auf.

Einführung in die Informatik II

Universität Stuttgart, Studienjahr 2007

Gliederung der Grundvorlesung

~~8. Suchen~~

9. Hashing

10. Sortieren

11. Graphalgorithmen

12. Speicherverwaltung

9. Hashing

- 9.1 Einführung (am Beispiel)
- 9.2 Hashfunktionen
- 9.3 Techniken beim Hashing
- 9.4 Analyse von Hashverfahren
- 9.5 Rehashing
- 9.6. Beispiel

Ziel dieses 9. Kapitels:

Beim Hashing werden die Elemente einer Menge nicht wie bei einem Suchbaum sortiert und durch Vergleiche wiedergefunden, sondern man berechnet mit Hilfe einer "Hashfunktion" aus dem Schlüssel einen Index (eine "Adresse"), unter dem der Schlüssel in einer "Hash"-Tabelle abgelegt wird.

In diesem Kapitel lernen Sie, welche Eigenschaften solch eine Hashfunktion besitzen muss, welche Funktionen in der Praxis eingesetzt werden, wie man durch Freihalten von Speicherplatz im Mittel eine konstante Such- und Einfügezeit erreicht, wie man das Löschen effizient behandelt und wie man den Speicherbereich dynamisch vergrößern kann. Zugleich werden Ihnen die hierfür benötigten Parameter (Tabellengröße, Auslastungsgrad, Kollisionsstrategien, Zyklenlänge) und ihre Bedeutung für Anwendungen vermittelt.

9.1 Einführung

Grundidee: Schlüssel sollen durch einen einzigen Zugriff auf eine Tabelle (mit maximal p Einträgen) gefunden werden.

Gegeben sei eine Menge möglicher Schlüssel S und die Tabellengröße, dies ist eine natürliche Zahl $p \ll |S|$.

Es ist eine Abbildung $f: S \rightarrow \{0, 1, \dots, p-1\}$ zu konstruieren, sodass es in einer zufällig ausgewählten n -elementigen Teilmenge $B = \{b_1, \dots, b_n\} \subseteq S$ (mit $n \leq p$) im Mittel nur wenige Elemente $b_i \neq b_j$ mit $f(b_i) = f(b_j)$ gibt.

Die drei Operationen Suchen, Einfügen und Löschen (siehe Anfang von Kap. 8) müssen sehr "effizient" realisiert werden:

- Entscheide, ob $s \in S$ in B liegt (und gib an, wo). **FIND**
- Füge einen Schlüssel s in B ein. **INSERT**
- Entferne einen Schlüssel s aus B . **DELETE**

Für die Abbildung $f: S \rightarrow \{0, 1, \dots, p-1\}$ müssen wir daher mindestens folgendes fordern:

- Sie muss surjektiv sein.
- Sie muss "gleichverteilt" sein, d.h., für jedes $0 \leq m < p$ sollte die Menge $S_m = \{s \in S \mid f(s)=m\}$ ungefähr $|S|/p$ Elemente enthalten.
- Sie muss schnell berechnet werden können.

Solch eine Abbildung f heißt **Schlüsseltransformation** oder **Hashfunktion**. (Genaueres siehe 9.4.1.)

Diese Funktion verstreut die möglichen Schlüssel über den Indexbereich $\{0, 1, \dots, p-1\}$. Daher der Name:

Hashing = (über eine Tabelle) gestreute Speicherung

Nehmen wir an, wir hätten eine solche Abbildung
 $f: S \rightarrow \{0, 1, \dots, p-1\}$, dann werden wir zur Speicherung
von Teilmengen A von S ein Feld deklarieren:

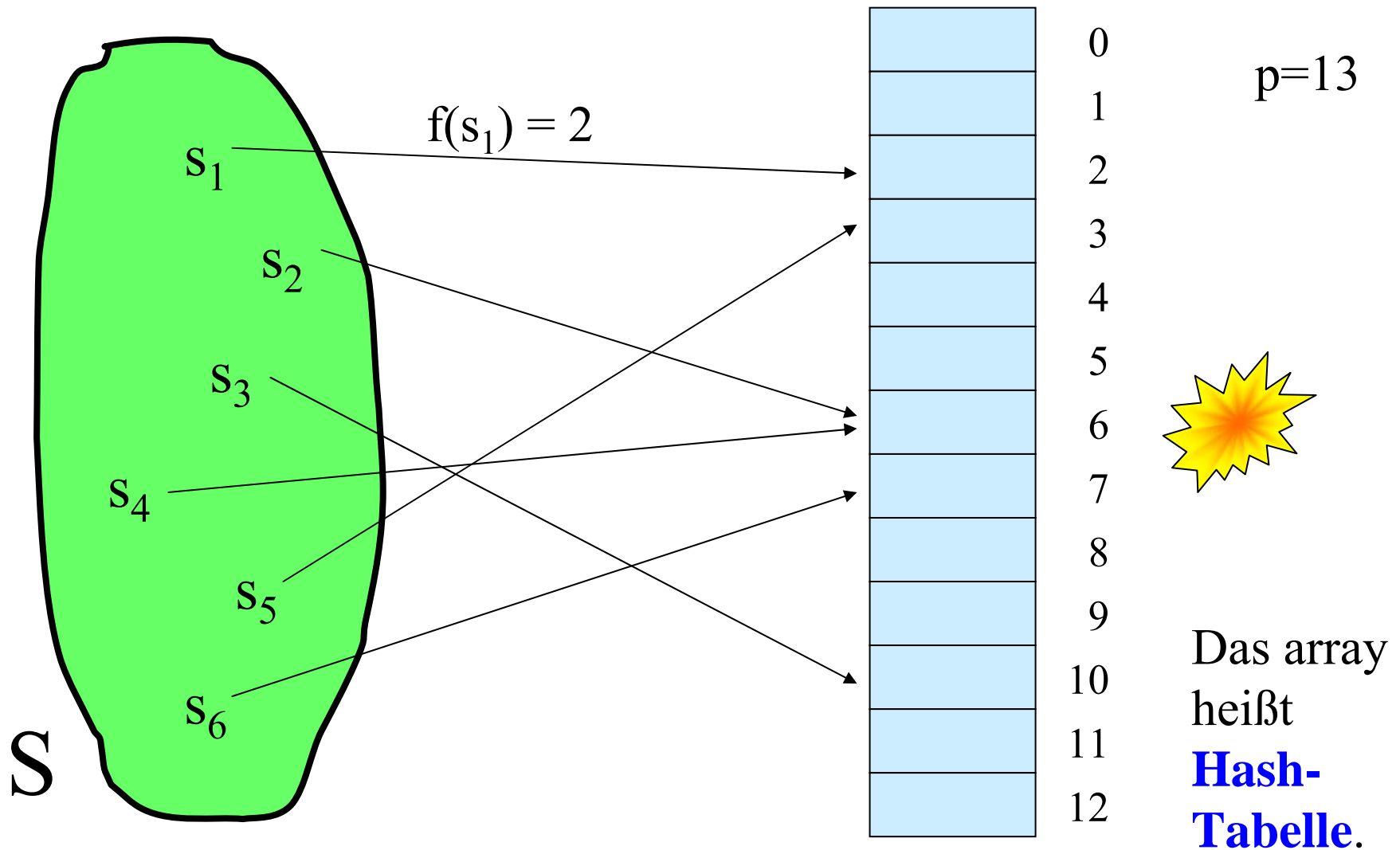
A : array (0.. $p-1$) of <Datentyp zur Menge S >

Jedes Element $s \in S$ wird unter der Adresse $f(s)$ gespeichert,
d.h., nach dem Speichern sollte $A(f(s)) = s$ gelten.

Um festzustellen, ob ein Schlüssel s in der jeweiligen
Teilmenge liegt, braucht man nur zu prüfen, was in $A(f(s))$
steht. Doch es entstehen Probleme, wenn in der konkreten
Teilmenge B mehrere Elemente mit gleichem f -Wert
("Kollisionen") enthalten sind.

Wie sieht es mit den Operationen INSERT und DELETE
aus? Wir schauen uns zunächst eine Skizze und dann ein
Beispiel an.

Folgende 6 Elemente s_1 bis s_6 sollen gespeichert werden:



Hier ist $f(s_2) = f(s_4) = 6$. Was nun?

Beispiel "modulo p"

$S = \Sigma^*$ = die Menge aller Folgen über einem t -elementigen Alphabet $\Sigma = \{\alpha_0, \alpha_1, \dots, \alpha_{t-1}\}$.

Weiterhin sei p eine natürliche Zahl, $p > 1$.

Eine nahe liegende Codierung $\varphi: \Sigma \rightarrow \{0, 1, \dots, t-1\}$ ist $\varphi(\alpha_i) = i$. Als Abbildung $f: \Sigma^* \rightarrow \{0, 1, \dots, p-1\}$ kann man dann die Codierung eines Anfangsworts der Länge q wählen (für ein q mit $0 < q \leq r$) oder Teile davon:

$$f(\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_r}) = \left(\sum_{j=1}^q \varphi(\alpha_{i_j}) \right) \underline{\text{mod}} p.$$

Wir demonstrieren dies am lateinischen Alphabet, wobei wir nur die großen Buchstaben **A, B, C, ...** verwenden. Als Codierung φ wählen wir die Position des Buchstabens im Alphabet, und als Menge B die Menge **A** der Monatsnamen:

a	$\varphi(a)$	a	$\varphi(a)$	a	$\varphi(a)$
A	1	J	10	S	19
B	2	K	11	T	20
C	3	L	12	U	21
D	4	M	13	V	22
E	5	N	14	W	23
F	6	O	15	X	24
G	7	P	16	Y	25
H	8	Q	17	Z	26
I	9	R	18		

Abzubildende
Menge **A** :

A = {**JANUAR,**
FEBRUAR,
MAERZ,
APRIL, MAI,
JUNI, JULI,
AUGUST,
SEPTEMBER,
OKTOBER,
NOVEMBER,
DEZEMBER}.

Wir berechnen nun $f(\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_r}) = \left(\sum_{j=1}^q \varphi(\alpha_{i_j}) \right) \underline{\text{mod}} p.$

für alle Wörter aus **A**. Zum Beispiel muss man für das Wort **JANUAR** als erstes die zu q gehörende Summe

$$\varphi(\mathbf{J}) = 10 \quad (\text{im Falle } q=1),$$

$$\varphi(\mathbf{J}) + \varphi(\mathbf{A}) = 10 + 1 = 11 \quad (\text{im Falle } q=2),$$

$$\varphi(\mathbf{J}) + \varphi(\mathbf{A}) + \varphi(\mathbf{N}) = 10 + 1 + 14 = 25 \quad (\text{im Falle } q=3),$$

$$\varphi(\mathbf{J}) + \varphi(\mathbf{A}) + \varphi(\mathbf{N}) + \varphi(\mathbf{U}) = 46 \quad (\text{im Falle } q=4) \text{ usw.}$$

ermitteln. Wir listen zunächst diese Summen in der folgenden Tabelle für verschiedene q auf; hierbei kann man auch Buchstaben weglassen und z.B. nur den ersten und dritten Buchstaben betrachten ("1.+3."); später müssen wir diese Werte modulo p (siehe übernächste Tabelle, die nur drei der sechs Spalten aus der anderen Tabelle benutzt) nehmen.

Wir erhalten für $q = 1, 2, 3, 4$ und für "1. und 3.", "2. und 3." die Werte:

Monatsname	q=1	q=2	q=3	q=4	1.+3.	2.+3.
JANUAR	10	11	25	46	24	15
FEBRUAR	6	11	13	31	8	7
MAERZ	13	14	19	37	18	6
APRIL	1	17	35	44	19	34
MAI	13	14	23	23	22	10
JUNI	10	31	45	54	24	35
JULI	10	31	43	52	22	33
AUGUST	1	22	29	50	8	28
SEPTEMBER	19	24	40	60	35	21
OKTOBER	15	26	46	61	35	31
NOVEMBER	14	29	51	56	36	37
DEZEMBER	4	9	9	14	4	5

Wir verwenden nur die Spalten "q=2", "q=3" und "2.+3.", wählen als p die Zahlen 17 und 22 und erhalten:

Monatsname	q = 2 p=17	q = 3 p=17	2.+3. p=17	q = 2 p=22	q = 3 p=22	2.+3. p=22
JANUAR	11	8	15	11	3	15
FEBRUAR	11	13	7	11	13	7
MAERZ	14	2	6	14	19	6
APRIL	0	1	0	17	13	12
MAI	14	6	10	14	1	10
JUNI	14	11	1	9	1	13
JULI	14	9	16	9	21	11
AUGUST	5	12	11	0	7	6
SEPTEMBER	7	6	4	2	18	21
OKTOBER	9	12	14	4	2	9
NOVEMBER	12	0	3	7	7	15
DEZEMBER	9	9	5	9	9	5

In der Tabellen haben wir bereits die genannte Modifikation für f benutzt, indem wir eine Teilmenge der Indizes $\{1, 2, \dots, r\}$ ausgewählt und die zugehörigen Buchstabenwerte aufsummiert haben. In der Tabelle auf den vorherigen Folien sind dies die Teilmengen

$\{1, 3\}$, bezeichnet durch 1.+3. sowie
 $\{2, 3\}$, bezeichnet durch 2.+3.

Die Abbildungen, die in den Spalten angegeben sind, sind untereinander nicht "besser" oder "schlechter", sondern sie sind nur von unterschiedlicher Qualität für unsere spezielle Menge **A** der Monatsnamen. Wir wählen nun irgendeine dieser Funktionen und fügen mit ihr die Monatsnamen in eine Tabelle (= in ein array A = in eine "Hashtabelle" A) mit p Komponenten ein.

Für die Abbildung wählen wir willkürlich $q=2$ und $p=22$ und berechnen also hier die Hashfunktion

$$f(\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_r}) = \left(\varphi(\alpha_{i_1}) + \varphi(\alpha_{i_2}) \right) \underline{\text{mod}} 22.$$

Zum Beispiel ist dann $f(\text{JANUAR}) = (10 + 1) \underline{\text{mod}} 22 = 11$ und $f(\text{OKTOBER}) = (15 + 11) \underline{\text{mod}} 22 = 4$. Alle Werte dieser Abbildung finden Sie in der entsprechenden Spalte für $q=2$ und $p=22$ auf der vorletzten Folie.

Die Monatsnamen tragen wir in ihrer jahreszeitlichen Reihenfolge nacheinander in das Feld A ein. Die Menge $S = \Sigma^*$ ist unendlich; doch nehmen wir hier an, dass die tatsächlich benutzten Wörter der Menge S höchstens die Länge 20 haben (kürzere Wörter werden durch Zwischenräume, deren φ -Wert 0 sei, aufgefüllt) und deklarieren daher die Hashtabelle:

A : array (0.. $p-1$) of String(20);

Es wird sicher auch folgender Fall auftreten:

Wir möchten einen Schlüssel s mit Hashwert $f(s) = k$ in die array-Komponente $A(k)$ eintragen; dort befindet sich jedoch bereits ein Schlüssel (es tritt ein "Konflikt" ein). Wir werden verschiedene Konfliktstrategien kennen lernen. Die einfachste ist sicherlich: Lege den Schlüssel s in die Komponente $A(k+1)$; ist diese ebenfalls besetzt, so versuche es mit $A(k+2)$ usw., wobei man von der Komponente $A(p-1)$ nach $A(0)$ übergeht (das Feld wird also als zyklisch aufgefasst).

Diese Vorgehensweise wird nun auf den nächsten Folien demonstriert; hierbei tritt bereits beim zweiten Schlüssel ein Konflikt auf.

Ein weiteres Beispiel finden Sie in Abschnitt 9.6.

A	0	
	1	
	2	
	3	
	4	
	5	
	6	
	7	
	8	
	9	
	10	
	11	JANUAR
	12	FEBRUAR
	13	
	14	MAERZ
	15	MAI
	16	
	17	APRIL
	18	
	19	
	20	
	21	

Füge das Wort **JANUAR**
mit $f(\text{JANUAR}) = 11$ ein:

Füge das Wort **FEBRUAR**
mit $f(\text{FEBRUAR}) = 11$ ein:

Konflikt! Verschiebe
FEBRUAR um einen
Platz nach hinten.

Füge das Wort **MAERZ**
mit $f(\text{MAERZ}) = 14$ ein:

Füge das Wort **APRIL** mit
 $f(\text{APRIL}) = 17$ ein:

Füge das Wort **MAI** mit
 $f(\text{MAI}) = 14$ ein:

Konflikt! Verschiebe
MAI um einen Platz
nach hinten.

A 0	AUGUST
1	
2	SEPTEMBER
3	
4	OKTOBER
5	
6	
7	NOVEMBER
8	
9	JUNI
10	JULI
11	JANUAR
12	FEBRUAR
13	DEZEMBER
14	MAERZ
15	MAI
16	
17	APRIL
18	
19	
20	
21	

Füge nun weiterhin die Wörter JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, **DEZEMBER** ein.

Die zugehörigen f-Werte lauten: 9, 9, 0, 2, 4, 7, 9.

Es entstehen erneut **Konflikte** bei JUNI, JULI und DEZEMBER. JULI muss um einen, DEZEMBER um zwei Plätze verschoben werden. Dabei entsteht ein Konflikt mit JANUAR, d.h., man muss DEZEMBER bis Platz 13 verschieben.



A	0	AUGUST
	1	
	2	SEPTEMBER
	3	
	4	OKTOBER
	5	
	6	
	7	NOVEMBER
	8	
	9	JUNI
	10	JULI
	11	JANUAR
	12	FEBRUAR
	13	DEZEMBER
	14	MAERZ
	15	MAI
	16	
	17	APRIL
	18	
	19	
	20	
	21	

Dies ist die Hashtabelle nach Einfügen der 12 Schlüssel.

Suchen:

Gesucht wird **APRIL**. Es ist $f(\text{APRIL}) = 17$. Man prüft, ob $A(17) = \text{APRIL}$ ist. Dies trifft zu, also ist **APRIL** in der Menge.

Gesucht wird **JULI**. Es ist $f(\text{JULI}) = 9$. Man prüft, ob $A(9) = \text{JULI}$ ist. Dies trifft nicht zu. Da $A(9)$ besetzt ist, könnte **JULI** durch einen Konflikt verschoben worden sein, also prüft man, ob $A(10) = \text{JULI}$ ist. Dies trifft zu, also ist **JULI** in der Menge.

A	0	AUGUST
	1	
	2	SEPTEMBER
	3	
	4	OKTOBER
	5	
	6	
	7	NOVEMBER
	8	
	9	JUNI
	10	JULI
	11	JANUAR
	12	FEBRUAR
	13	DEZEMBER
	14	MAERZ
	15	MAI
	16	
	17	APRIL
	18	
	19	
	20	
	21	

Gesucht wird **DEZEMBER**. Es ist $f(\text{DEZEMBER}) = 9$. Man prüft, ob $A(9) = \text{DEZEMBER}$ ist, dann für $A(10)$ usw. bis man entweder auf **DEZEMBER** oder auf einen leeren Eintrag trifft.

Gesucht wird **JURA**. Es ist $f(\text{JURA}) = 9$. Man prüft, ob $A(9) = \text{JURA}$ ist. Dies trifft nicht zu, also geht man zu $A(10)$ usw., bis man auf den leeren Eintrag $A(16)$ trifft, dort bricht die Suche ab und **JURA** ist nicht in der Menge der Monatsnamen.

Wie löscht man? (Später!)

Wie viele Vergleiche braucht man, um einen Schlüssel zu finden, der in der Menge liegt?

JANUAR: 1 Vergleich

FEBRUAR: 2 Vergleiche

MAERZ: 1 Vergleich

APRIL: 1 Vergleich

MAI: 2 Vergleiche

JUNI: 1 Vergleich

JULI: 2 Vergleiche

AUGUST: 1 Vergleich

SEPTEMBER: 1 Vergleich

OKTOBER: 1 Vergleich

NOVEMBER: 1 Vergleich

DEZEMBER: 5 Vergleiche

insgesamt 19 Vergleiche

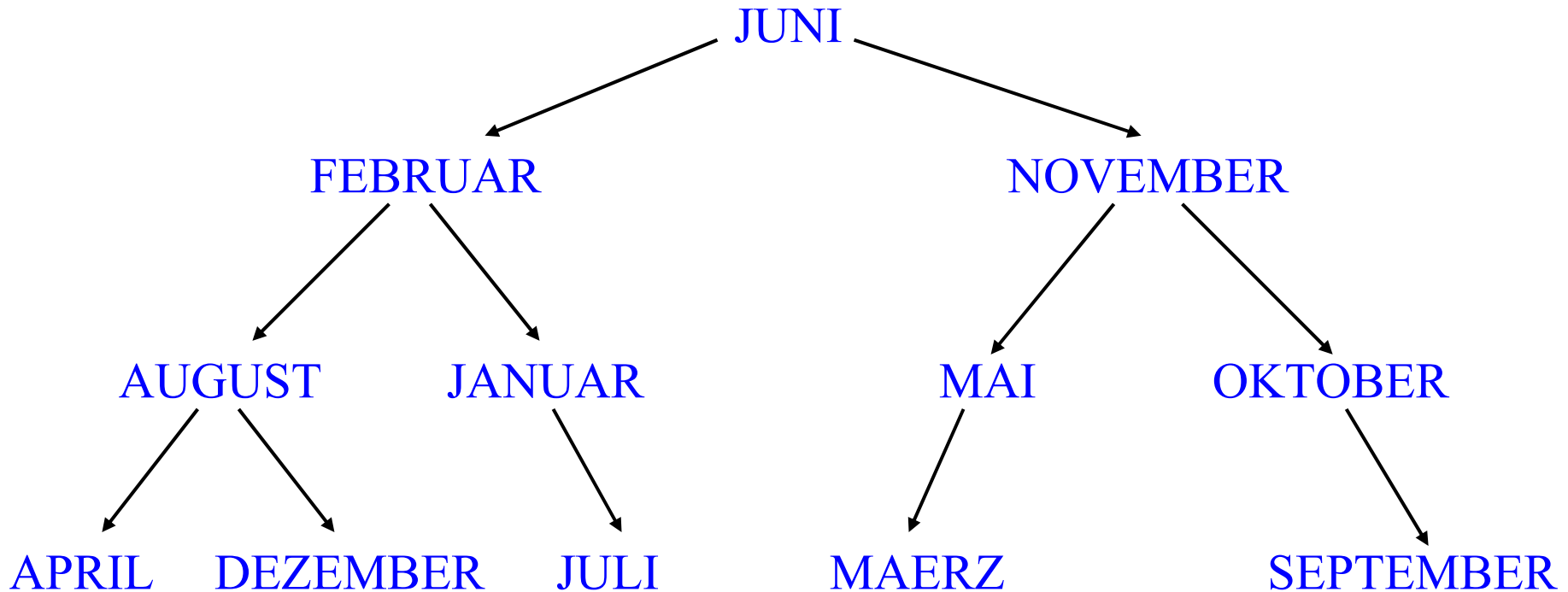
Im Mittel braucht man
also
 $19/12 \approx \mathbf{1,6 \text{ Vergleiche}}$,
falls der Schlüssel in
der Menge ist
(erfolgreiche Suche)

Wie viele Vergleiche braucht man für einen Schlüssel, der **nicht** in der Menge liegt? Gehe jede Komponente des Feldes hierzu durch (f soll gleichverteilt sein, siehe Anfang von Kapitel 9.1):

0:	2 Vergleiche	11:	6 Vergleiche
1:	1 Vergleich	12:	5 Vergleiche
2:	2 Vergleiche	13:	4 Vergleiche
3:	1 Vergleich	14:	3 Vergleiche
4:	2 Vergleiche	15:	2 Vergleiche
5:	1 Vergleich	16:	1 Vergleich
6:	1 Vergleich	17:	2 Vergleiche
7:	2 Vergleiche	18:	1 Vergleich
8:	1 Vergleich	19:	1 Vergleich
9:	8 Vergleiche	20:	1 Vergleich
10:	7 Vergleiche	21:	1 Vergleich
Gesamt:		<hr/> 55 Vergleiche	

Im Mittel braucht man also $55/22 \approx \mathbf{2,5 \text{ Vergleiche}}$, falls der gesuchte Name **nicht** in der Menge ist (erfolglose Suche).

Vergleich mit einem ausgeglichenen Suchbaum (siehe 8.4):



Mittlere Anzahl der Vergleiche für Elemente, die in der Menge sind: $(1+2+2+3+3+3+3+4+4+4+4+4)/12 \approx \mathbf{3,1 \text{ Vergleiche}}$.

Falls das Element **nicht** in der Menge ist (13 null-Zeiger):
im Mittel $49/13 \approx \mathbf{3,8 \text{ Vergleiche}}$.

Wir vernachlässigen hier, dass man an jedem Knoten eigentlich zwei Vergleiche durchführt: auf "Gleichheit" und auf "Größer".

Zeitbedarf: Die Hashtabelle ist deutlich günstiger. Man muss aber die Berechnung der Abbildung f hinzu zählen, die allerdings nur einmal je Wort durchgeführt wird.

Speicherplatz: Wir benötigen 22 statt 12 Bereiche für die Elemente der Schlüsselmenge S . Dafür sparen wir die Zeiger des Suchbaums. Es hängt also vom Platzbedarf ab, den jedes Element aus S braucht, um abschätzen zu können, ob sich diese Tabellendarstellung mit der Abbildung f lohnt.

Sie ahnen es schon: Hashtabellen sind in der Regel deutlich günstiger als Suchbäume. Allerdings darf man die Tabelle nicht zu sehr füllen, da dann die Suchzeiten, insbesondere für Wörter, die *nicht* in der Tabelle sind, stark anwachsen. Erfahrungswert: Mindestens **20%** der Plätze sollten ständig frei bleiben (vgl. Abschnitt 9.4).

9.2 Hashfunktionen

9.2.1 Aufgabe:

n Elemente einer (sehr großen) Menge S sollen in einem Feld array $(0..p-1)$ of ... gesucht und dort in irgendeiner Reihenfolge eingefügt und gelöscht werden können.

Es sei $|S| > p$ (sonst ist die Aufgabe ohne Konflikte durch irgendeine injektive Zuordnung zu lösen).

Benutze hierfür eine Funktion $f: S \rightarrow \{0, 1, \dots, p-1\}$, genannt *Hashfunktion*, die

- surjektiv ist (d.h., jede Zahl von 0 bis $p-1$ tritt als Bild auf),
- die die Elemente von B möglichst gleichmäßig über die Zahlen von 0 bis $p-1$ verteilt und
- die schnell berechnet werden kann.

In der Praxis verwendet man meist das

9.2.2 Divisionsverfahren

1. Fasse den gegebenen Schlüssel s als Zahl auf (jedes Datum ist binär dargestellt und kann daher prinzipiell als Zahl aufgefasst werden).
2. Bilde den Rest der Division durch die Zahl p
$$f(s) = s \bmod p.$$

Diese Restbildung hatten wir in 9.1 benutzt.

(p wählt man meist als Primzahl, um Abhängigkeiten bei der Modulo-Bildung zu verringern; bei quadratischer Kollisionsstrategie, siehe unten, maximiert man hierdurch die Zykellänge.)

Ein anderes Verfahren verwendet eine "möglichst irrationale" Zahl z zwischen 0 und 1.

9.2.3 Multiplikationsverfahren:

(p ist die Größe der Hashtabelle)

1. Fasse wiederum den gegebenen Schlüssel s als Zahl auf.
2. Multipliziere diese Zahl mit z und betrachte nur den Nachkommateil, d.h., die Ziffernfolge nach dem Dezimalpunkt:
$$g(s) = s \cdot z - \lfloor s \cdot z \rfloor$$

(dies ist eine reelle Zahl größer gleich 0 und kleiner 1).
3. Erweitere dies auf das Intervall $[0..p)$ und bilde den ganzzahligen Anteil:

$$f(s) = \lfloor p \cdot g(s) \rfloor.$$

Beispiel: Seien $p = 22$ und $z = 0,624551$.

Dann gilt für $s = 34$: $s \cdot z = 21,234734$, $g(s) = 0,234734$.

$f(s) = \text{ganzzahliger Anteil von } p \cdot g(s) = \lfloor 5,164148 \rfloor = 5$.

Hinweise:

1. Die Zahl $|c_2| = 0.618\ 033\ 988\ 749\ 894\ 848\ 204 \dots$ gilt als gut geeignete Zahl z (zu c_2 siehe Fibonaccizahlen, 8.4.7).
2. Es kann auch $g(s) = \lceil s \cdot z \rceil - s \cdot z$ benutzt werden.

9.2.4: Wenn Zeichenfolgen als Schlüsselmenge $S = \Sigma^*$ vorliegen, wählt man gerne ein Teilfolgenverfahren (hier bzgl. der Division vorgestellt; analog: bzgl. der Multiplikation):

1. Codiere die Buchstaben: $\varphi: \Sigma \rightarrow \{0, 1, \dots, t-1\}$, z.B. ASCII.
2. Wähle fest eine Teilfolge $i_1 i_2 \dots i_q$.
3. Wähle als Hashfunktion $f: \Sigma^* \rightarrow \{0, 1, \dots, p-1\}$

$$f(\alpha_1 \alpha_2 \dots \alpha_r) = \left(\sum_{j=1}^q \varphi(\alpha_{i_j}) \right) \underline{\text{mod}} \ p$$

oder verwende allgemein eine gewichtete Summe mit irgendwelchen speziell gewählten Zahlen x_1, x_2, \dots, x_q :

$$f(\alpha_1 \alpha_2 \dots \alpha_r) = \left(\sum_{j=1}^q x_j \cdot \varphi(\alpha_{i_j}) \right) \underline{\text{mod}} \ p.$$

Definition 9.2.5: Eine Hashfunktion $f: S \rightarrow \{0, 1, \dots, p-1\}$ heißt perfekt bzgl. einer Menge $B \subseteq S$ (mit $|B| \leq p$) von Elementen, wenn f auf der Menge B injektiv ist, wenn also für alle Elemente $b_i \neq b_j$ aus B stets $f(b_i) \neq f(b_j)$ gilt.

Wenn man einen unveränderlichen Datenbestand hat (etwa gewisse Wörter in einem Lexikon oder die reservierten Wörter einer Programmiersprache), so lohnt es sich, eine Hashtabelle mit einer perfekten Hashfunktion einzusetzen, da dann die Entscheidung, ob ein Element b in der Tabelle vorkommt, durch eine Berechnung $f(b)$ und einen weiteren Vergleich getroffen werden kann.

Durch Ausprobieren lassen sich solche perfekten Funktionen finden. Siehe Spalte 2.+3. und $p=17$ auf Folie 12. Suchen Sie z.B. eine für $\mathbf{A} = \{\text{JANUAR}, \dots, \text{DEZEMBER}\}$ und $p=15$. Blättern Sie erst danach zur nächsten Folie weiter.

Eine Lösung für $\mathbf{A} = \{\text{JANUAR}, \dots, \text{DEZEMBER}\}$ und $p=15$ lautet:

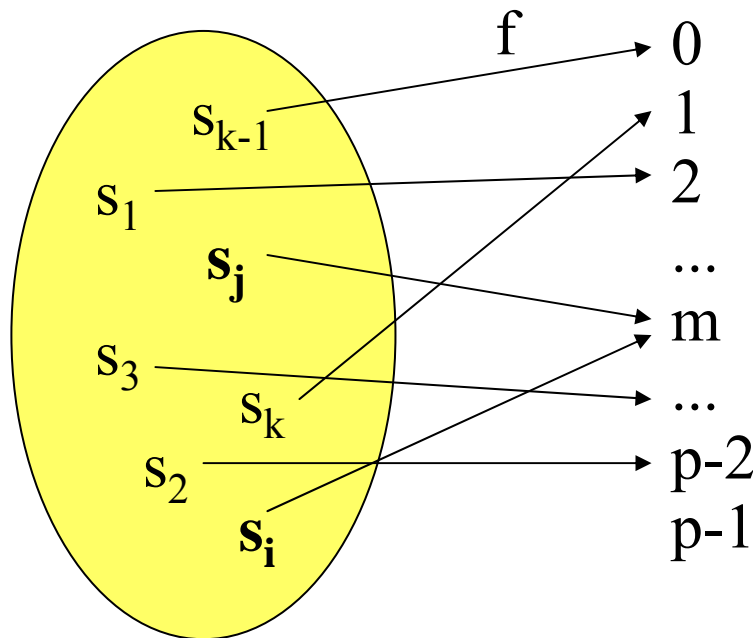
$$f(\alpha_1 \alpha_2 \dots \alpha_r) = (7 \varphi(\alpha_1) + 5 \varphi(\alpha_2) + 2 \varphi(\alpha_3)) \pmod{15}.$$

Dieses f ist
tatsächlich
injektiv:

JANUAR	13
FEBRUAR	11
MAERZ	1
APRIL	3
MAI	9
JUNI	8
JULI	4
AUGUST	6
SEPTEMBER	10
OKTOBER	5
NOVEMBER	7
DEZEMBER	0

9.2.6: Wahrscheinlichkeit für einen Konflikt.

In eine Tabelle von p Plätzen sollen nun k Elemente aus S mit Hilfe einer Hashfunktion $f: S \rightarrow \{0, 1, \dots, p-1\}$ eingetragen werden. Eine Hashfunktion f soll die Elemente aus S möglichst gleichmäßig auf die p Zahlen abbilden. Wie groß ist die Wahrscheinlichkeit, dass unter k verschiedenen Elementen mindestens zwei Elemente s_i und s_j sind mit $f(s_i) = f(s_j)$?



Berechne die Wahrscheinlichkeit, dass unter k verschiedenen Elementen mindestens zwei Elemente s_i und s_j sind mit $f(s_i) = f(s_j)$. Dies ist 1 minus der Wahrscheinlichkeit, dass alle k Elemente auf verschiedene Werte abgebildet werden:

$$1 - \left(1 - \frac{1}{p}\right) \cdot \left(1 - \frac{2}{p}\right) \cdot \dots \cdot \left(1 - \frac{k-1}{p}\right)$$

$$\approx 1 - \prod_{i=1}^{k-1} e^{-\frac{i}{p}} = 1 - e^{-\frac{k(k-1)}{2p}}$$

$$\text{Beachte hierbei: } (1 - i/p) \approx e^{-\frac{i}{p}} = 1 - \frac{i}{p} + \frac{\left[\frac{i}{p}\right]^2}{2!} - \frac{\left[\frac{i}{p}\right]^3}{3!} + \dots$$

Wann beträgt die Wahrscheinlichkeit 50%, dass mindestens zwei Schlüssel auf den gleichen Wert abgebildet werden?

$$1 - e^{-\frac{k(k-1)}{2p}} = \frac{1}{2} \quad \text{liegt vor bei}$$

$$\ln\left(\frac{1}{2}\right) = -\frac{k(k-1)}{2p}, \text{ d.h., es gilt ungefähr}$$

$$k \approx \sqrt{p \cdot 2 \cdot \ln(2)} \quad \text{mit } 2 \cdot \ln(2) \approx 1,386 \text{ und } \sqrt{2 \cdot \ln(2)} \approx 1,1777.$$

Satz 9.2.7

Trägt man gleichverteilte Schlüssel nacheinander in eine Hashtabelle der Größe p ein, so muss man nach $1,1777 \cdot \sqrt{p}$ Schritten damit rechnen, dass "Kollisionen" eingetreten sind, dass also zwei verschiedene Schlüssel auf den gleichen Platz eingetragen werden wollen.

Hinweis:

Diese Aussage gilt natürlich nicht nur für Hashtabellen.

Bekannt ist das "**Geburtstagsparadoxon**": Wie groß ist die Wahrscheinlichkeit, dass sich unter k Personen mindestens zwei Personen mit gleichem Geburtstag befinden?

Da hier $p=365$ (oder 366) und $2p = 730$ ist, lautet die Antwort:

$$\approx 1 - e^{-\frac{k(k-1)}{730}}$$

Soll die Wahrscheinlichkeit 50% sein, so ist k so zu wählen, dass

$$1/2 = e^{-\frac{k(k-1)}{730}}$$

gilt, d.h., $k \approx 1,1777 \cdot 19,105 \approx 22,5$. Wenn also nur 23 Personen zusammen sind, so ist die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, bereits über 50%.

9.2.8: Wie häufig werden Kollisionen auftreten?

Hierzu wählen wir zufällig k Schlüssel s_1, s_2, \dots, s_k und betrachten die Folge der Hashwerte $(f(s_1), f(s_2), \dots, f(s_k))$. Wie groß ist die Wahrscheinlichkeit, dass ein Wert j in dieser Folge nicht auftritt ($0 \leq j \leq p-1$)?

Wegen der angenommenen Eigenschaften der Funktion h sollte jeder Index j mit gleicher Wahrscheinlichkeit $1/p$ auftreten. Folglich tritt j bei k Berechnungen mit der Wahrscheinlichkeit $(1-1/p)^k$ *nicht* auf, d.h., mit der Wahrscheinlichkeit $1 - (1-1/p)^k$ kommt j mindestens einmal vor. Es gilt:

$$(1-1/p)^k \approx e^{-\frac{k}{p}} = e^{-\lambda}$$

mit $\lambda = k/p = \text{"Auslastungsgrad"}$ der Tabelle.

Jeder Index wird also ungefähr mit der Wahrscheinlichkeit $1-e^{-\lambda}$ vorkommen. Ist $k=p/2$, d.h., $\lambda=1/2$, so werden ungefähr $p \cdot (1-e^{-\lambda}) = p \cdot (1-e^{-1/2}) \approx p \cdot 0,3905$ verschiedene Indizes auftreten; die verbleibenden $p/2 - p \cdot 0,3905 = 0,1095 \cdot p$ Berechnungen führen also zu Konflikten bereits bei der Berechnung der Hashfunktion (sog. Primärkollisionen; es kommen noch die Konflikte hinzu, die durch das Verschieben der Schlüssel in der Hashtabelle zusätzlich entstehen, siehe später 9.3.6).

Ist $\lambda=1$, so wird jeder Index mit der Wahrscheinlichkeit $(1-e^{-1}) \approx 0,63212...$ besucht. Fügt man also p Elemente nacheinander in eine Hashtabelle der Größe p ein, so werden hierbei nur rund 63,2% verschiedene Tabellen-Indizes beim Ausrechnen der Hashfunktion berechnet. Es treten also $0,3689 \cdot p$ Primär-Kollisionen auf.

9.3 Techniken beim Hashing

Man unterscheidet beim Hashing zwei Techniken: das geschlossene Hashing mit angefügten Überlaufbereichen und das offene Hashing, das alle Schlüssel im vorgegebenen Array unterbringt.

9.3.1 Geschlossenes Hashing

Beim *geschlossenen* Hashing lässt man keine Korrekturen des Hashwertes zu, sondern die Hash-Funktion führt zu einem Index, über den man zu einer Datenstruktur gelangt, an der sich der gesuchte Schlüssel befindet, befinden müsste oder an der er einzufügen ist. In der Regel werden alle gespeicherten Schlüssel, die den gleichen Hash-Wert besitzen, in eine lineare Liste oder in einen Suchbaum eingefügt.

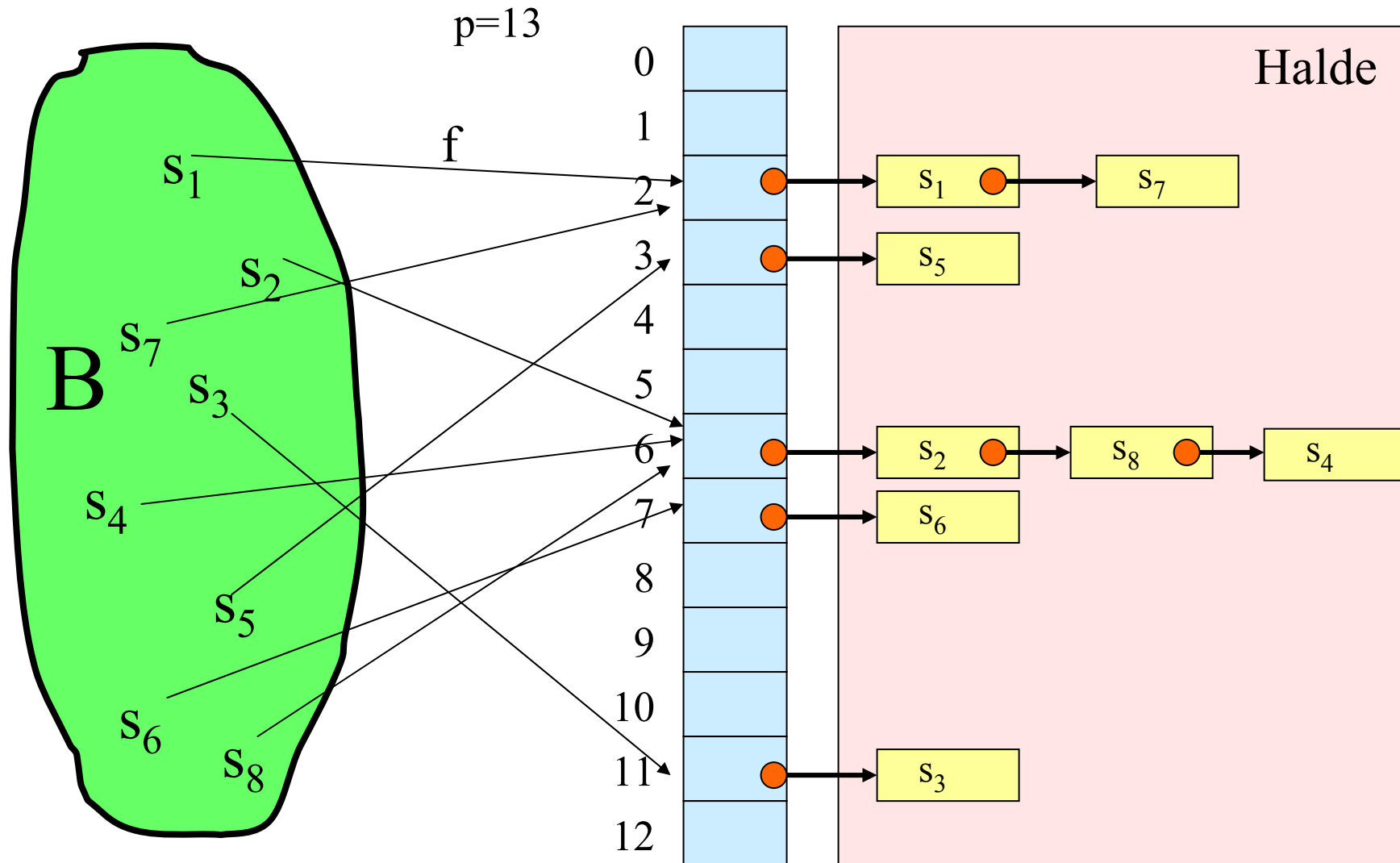
Als Beispiel betrachten wir auf der nächsten Folie den Fall, dass alle Schlüssel, die den gleichen f-Wert haben, in einer linearen Liste gespeichert werden. In der Hashtabelle A steht an der Stelle $A(i)$ der Zeiger auf die Liste der Schlüssel s mit $f(s)=i$.

Auf diese Weise entstehen keine Kollisionen in der Hashtabelle, sondern dieses Problem wird in die Verwaltung der p Listen verlagert.

(Hinweis: In der Praxis ist dies die "Haldenverwaltung".)

Die Hashtabelle ist somit nur eine Zugriffsstruktur für viele gleichartige Speicherstrukturen, die relativ klein gehalten werden können. Man spricht auch von externen Hashtabellen oder von Hashing mit externer Kollision.

Skizze: "Externe" Hashtabelle



Überlaufprobleme müssen mit der Haldenverwaltung gelöst werden!

Zeitaufwand beim geschlossenen Hashing: Das Suchen ("FIND") erfordert einen Zugriff auf das Feld $A(f(s))$ und anschließend muss die jeweilige Datenstruktur durchsucht werden.

Auch Einfügen ("INSERT") und Löschen ("DELETE") laufen bis auf den ersten Zugriff wie bei den zugrunde liegenden Datenstrukturen ab.

Vorteil des geschlossenen Hashings: Man muss keine feste obere Grenze für die Menge B der Schlüssel vorgeben. Vor allem, wenn viel gelöscht wird, kann man die schnellen Algorithmen der jeweiligen Datenstrukturen einsetzen. Insbesondere bei großen Datenbeständen lohnen sich Suchbäume.

Nachteil: Das Verfahren hängt von der Verwaltung der Listen (Zugriffszeiten, Garbage Collection) ab. (Offenes Hashing ist meist effizienter, siehe im Folgenden).

9.3.2 Offenes Hashing

Beim *offenen* Hashing wird der tatsächliche Index, unter dem der Schlüssel s später steht, erst mit dem Eintragen, also ggf. nach dem Durchlaufen einer Kollisionsstrategie, bestimmt. In diesem Fall befinden sich alle Schlüssel im Speicherbereich, den der Indexraum vorgibt (also im array $(0..p-1)$), und es gibt keine Überlaufbereiche.

Die Bezeichnungen „offen“ und „geschlossen“ sind anfangs verwirrend, da sie aus der Indexzuordnung abgeleitet wurden und sich nicht auf die Struktur des Speicherbereichs beziehen. Der Index bleibt also nach der Berechnung des Hashwertes "noch offen" und wird erst festgelegt, wenn ein freier Platz gefunden wurde.

Das Suchen geht in der Regel schnell, sofern mindestens 20% der Plätze des array frei gehalten werden.

Das Einfügen ist nicht schwierig.

Problem: Löschen.

9.3.3 Datentypen für das offene Hashing festlegen

(die Booleschen Werte brauchen wir erst später; in der Praxis speichert man den Hashwert $f(s)$ des Schlüssels s zusätzlich ab; in der Regel lässt man die Komponente "Inhalt" weg, wenn der Schlüssel auf den eigentlichen Speicherort verweist):

type Eintragtyp is record

belegt: Boolean; geloescht: Boolean;

kollision: Boolean; behandelt: Boolean;

Schluessel: Schluesseltyp;

Inhalt: Inhalttyp;

end record;

type Hashtabelle is array(0.. $p-1$) of Eintragtyp;

FIND: Der Suchalgorithmus lautet dann: ...

INSERT: Der Einfügealgorithmus lautet dann: ...

9.3.4 Einfüge-Algorithmus (für offenes Hashing)

```
A: Hashtabelle; i, j: Integer;           -- p sei global bekannt
k: Integer := 0;                         -- k gibt die Anzahl der Schlüssel in A an
for i in 0..p-1 loop A(i).besetzt:=false; A(i).kollision:=false;
    A(i).geloescht:=false; A(i).behandelt:=false; end loop;
while "es gibt noch einen einzutragenden Schlüssel s" loop
    if k < p then
        k := k+1; j := f(s);             -- j ist der Index in A für s
        if A(j).geloescht or not A(j).besetzt then A(j).besetzt := true;
            A(j).Schluessel := s; A(j).Inhalt := ...;
        else A(j).kollision := true; "Starte eine Kollisionsstrategie";
        end if;
    else "Tabelle A ist voll, starte eine Erweiterungsstrategie für A";
    end if;
end loop;
```

9.3.5 Das Suchen erfolgt ähnlich:

Um einen Eintrag mit dem Schlüssel s zu finden, berechne $f(s)$ und prüfe, ob in $A(f(s))$ ein Eintrag mit dem Schlüssel s steht. Falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe $A(f(s)).\text{kollision}$. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls berechne mit der verwendeten Kollisionsstrategie (s.u.) einen neuen Platz j und prüfe erneut, ob der Schlüssel s gleich $A(j).\text{Schlüssel}$ ist; falls ja, ist die Suche erfolgreich beendet, falls nein, prüfe den Wert von $A(j).\text{kollision}$. Ist dieser Wert false, dann ist die Suche erfolglos beendet, anderenfalls berechne mit der verwendeten Kollisionsstrategie einen neuen Platz j usw.

Wir wenden uns nun den Kollisionen und ihrer Behandlung zu.

Definition 9.3.6: Sei $f: S \rightarrow \{0, 1, \dots, p-1\}$ eine Hashfunktion.

Gilt $f(s) = f(s')$ für zwei einzufügende Schlüssel s und s' , so spricht man von einer Primärkollision. In diesem Fall muss der zweite Schlüssel s' an einer Stelle $A(i)$ gespeichert werden, für die $i \neq f(s')$ gilt.

Ist $f(s') = i$ und befindet sich auf dem Platz $A(i)$ ein Schlüssel s'' mit $f(s'') \neq i$, so spricht man von einer Sekundärkollision, d.h., die erste Kollision beim Eintragen von s' wird durch einen Schlüssel s'' verursacht, der vom Hashwert her nicht an die Position i gehört und selbst durch Kollision hierhin gelangt ist.

Wenn man eine Strategie zur Behandlung von Kollisionen festlegt, so kann man sich gegen die Primärkollisionen kaum wehren, aber man kann versuchen, die Sekundärkollisionen klein zu halten.

Beispiel: Sei

$\mathbf{A} = \{\text{JANUAR, FEBRUAR, MÄRZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER}\}$ mit

$f(\alpha_1 \alpha_2 \dots \alpha_r) = (\varphi(\alpha_1) + \varphi(\alpha_2)) \bmod 14$. Drei Schlüssel werden in der Reihenfolge JANUAR, FEBRUAR, OKTOBER eingegeben. Es gilt:

$f(\text{JANUAR}) = 11$, $f(\text{FEBRUAR}) = 11$, $f(\text{OKTOBER}) = 12$.

Wir tragen JANUAR in der Komponente A(11) ein.

Der Schlüssel FEBRUAR führt zu einer Primärkollision. Die Strategie möge lauten: *Gehe von Platz j zum nächsten Platz $j+1$* . Dann wird FEBRUAR in dem Platz A(12) gespeichert.

Der Schlüssel OKTOBER gehört in den Platz A(12), doch hier steht ein Schlüssel, der dort nicht hingehört, sondern durch eine Kollision hierhin verschoben wurde. Folglich führt OKTOBER zu einer Sekundärkollision. (OKTOBER wird dann auf Platz A(13) eingetragen.)

Definition 9.3.7: Kollisionsstrategien ("Sondieren")

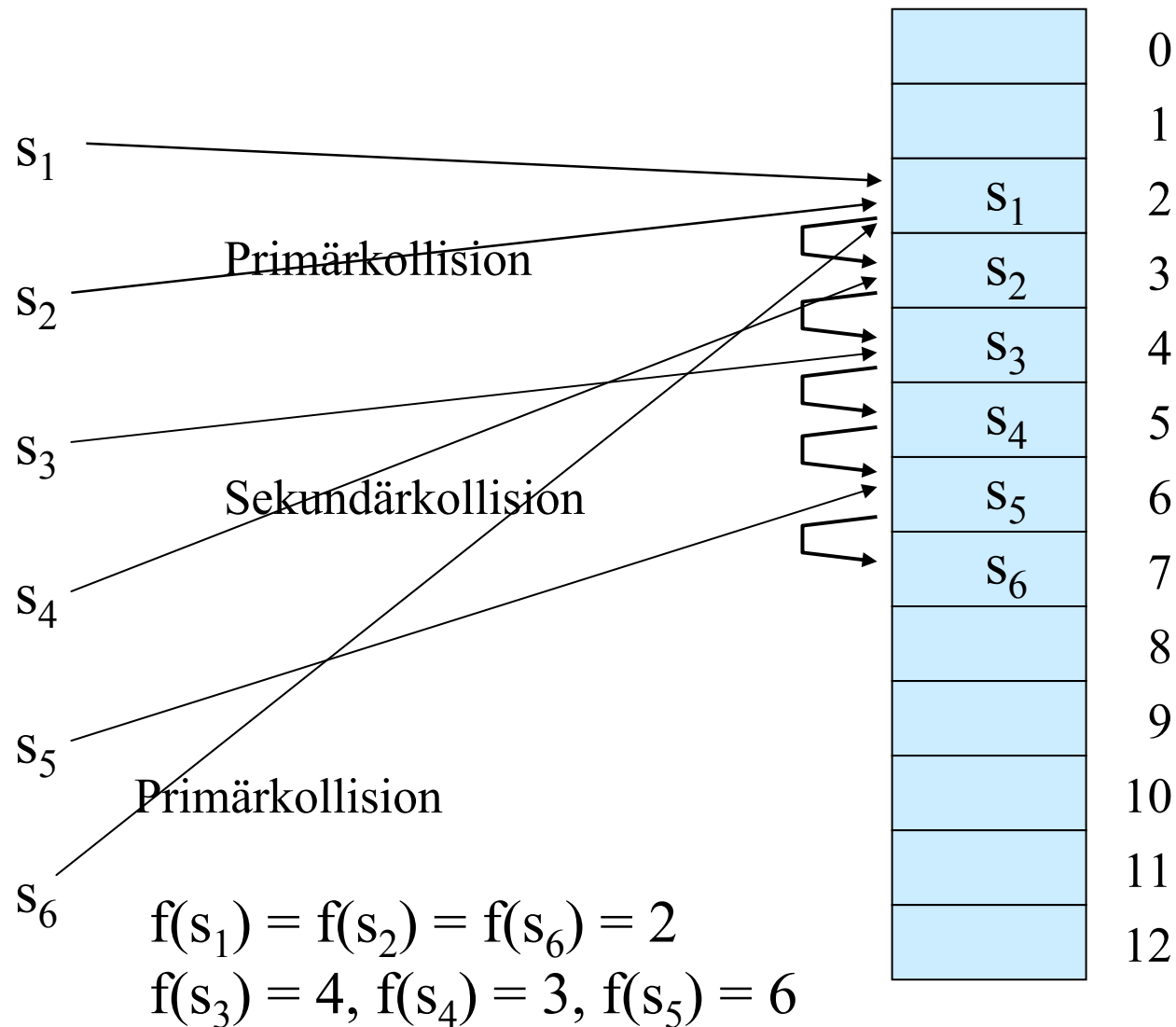
Der Schlüssel s soll stets unter dem Index $f(s)$ gespeichert werden. Ist diese Komponente bereits besetzt, so versuche man, den Schlüssel s unter dem Index $G(s,i)$ einzufügen. Es sei i die Zahl der versuchten Zugriffe. c ist eine fest gewählte Konstante mit $\text{ggT}(c,p)=1$; man kann stets $c=1$ wählen.

$G(s,i) = (f(s)+i \cdot c) \bmod p$ heißt "lineare Fortschaltung" oder "**lineares Sondieren**" oder "Lineares Hashing".

$G(s,i) = (f(s)+i^2) \bmod p$ heißt "quadratische Fortschaltung" oder "**quadratisches Sondieren**".

Beispielskizze: Lineares Sondieren mit $c=1$

$p=13$



Das lineare Sondieren ist ein leicht zu implementierendes Verfahren, das sich in der Praxis bewährt hat. *Nachteil* ist die sog. "**Clusterbildung**", die durch Sekundärkollisionen hervorgerufen wird (siehe auch das vorige Beispiel):

Die Wahrscheinlichkeit, auf eine bereits durchlaufene Kollisionkette zu stoßen, wird im Laufe der Zeit größer als die Wahrscheinlichkeit, auf Anhieb einen freien Platz zu finden. Dadurch müssen die Kollisionketten immer nachvollzogen werden: Es bilden sich sog. Cluster, siehe Erläuterung unten.

Der *Vorteil* des linearen Sondierens liegt darin, dass man bei dieser Kollisionsstrategie Werte aus der Hashtabelle wieder löschen kann. (Selbst überlegen und Übungen. Das prinzipielle Vorgehen wird unten erläutert.)

Das quadratische Sondieren ist ebenfalls leicht zu implementieren. Sein *Vorteil* ist, dass die Sekundärkollisionen und damit die Clusterbildung reduziert werden. (Bei Primärkollisionen müssen natürlich alle Versuche erneut nachvollzogen werden.)

Der *Nachteil* des quadratischen Sondierens besteht darin, dass der Aufwand, um Werte aus der Hashtabelle wieder zu löschen, unzumutbar groß ist. Man markiert daher die gelöschten Elemente als "gelöscht", belässt sie aber weiter in der Tabelle und entfernt alle gelöschten Elemente erst nach einiger Zeit gemeinsam (siehe unten 9.5).

Um nicht in kurze Zyklen bei der Kollisionsstrategie zu gelangen, muss man beim quadratischen Sondieren unbedingt verlangen, dass die Größe der Hashtabelle p eine Primzahl ist. Dies wird in Satz 9.3.10 begründet.

9.3.8 Clusterbildung bei linearem Sondieren

0	
1	
2	
3	s_1
4	s_2
5	s_3
6	s_4
7	s_5
8	
9	
10	
11	
12	

Es möge die nebenstehende Situation mit dem Cluster $A(3)$ bis $A(7)$ entstanden sein.

Es soll nun ein weiterer Schlüssel s eingefügt werden. Ist $f(s)$ einer der Werte 3, 4, 5, 6 oder 7, so wird s bei linearem Sondieren mit $c=1$ in $A(8)$ gespeichert.

Die Wahrscheinlichkeit, dass im nächsten Schritt $A(8)$ belegt wird, ist daher $6/13$, während für jeden anderen Platz nur die Wahrscheinlichkeit $1/13$ gilt.

Cluster haben also eine hohe Wahrscheinlichkeit, sich zu vergrößern. Genau dieser Effekt wird in der Praxis beobachtet.

Die Clusterbildungen beruhen auf den Sekundärkollisionen.
Diese werden beim quadratischen Sondieren reduziert.

Als Beispiel betrachten wir erneut $\mathbf{A} = \{\text{JANUAR, FEBRUAR, MAERZ, APRIL, MAI, JUNI, JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DEZEMBER}\}$ mit $p=22$.

Als Abbildung verwenden wir dieses Mal die Hashfunktion
 $f(\alpha_1 \alpha_2 \dots \alpha_r) = (2\varphi(\alpha_1) + \varphi(\alpha_2)) \bmod 17$.

A

0	FEBRUAR
1	APRIL
2	
3	JANUAR
4	
5	
6	AUGUST
7	JUNI
8	JULI
9	SEPTEMBER
10	MAERZ
11	MAI
12	
13	NOVEMBER
14	DEZEMBER
15	
16	OKTOBER

Quadratisches Sondieren:
Wir fügen die Wörter ein
JANUAR, FEBRUAR,
MAERZ, APRIL, MAI,
JUNI, JULI, AUGUST,
SEPTEMBER, OKTOBER,
NOVEMBER, DEZEMBER .

Die zugehörigen f-Werte
lauten: 4, 0, 10, 1, 10, 7, 7,
6, 9, 7, 9, 13.

Tragen Sie die Wörter in
die zunächst leere Tabelle
ein. Es ergibt sich am Ende
die nebenstehende Tabelle.

9.3.9: Länge von Zyklen bei Kollisionsstrategien

Wir müssen uns nun überzeugen, dass bei den Kollisionsstrategien keine zu kurzen Zyklen durchlaufen werden. Beim linearen Sondieren ist dies gewährleistet: Wenn c und p teilerfremd sind ($\text{ggT}(c,p)=1$), dann durchläuft die Folge der Zahlen $(f(s)+i \cdot c) \bmod p$ (für $i=0, 1, 2, \dots$) alle Zahlen von 0 bis $p-1$, bevor eine Zahl erneut auftritt.

Wir wollen nun zeigen, dass beim quadratischen Sondieren keine "kurzen" Zyklen auftreten, sofern p eine Primzahl ist. (Wir nehmen hier an, dass $p > 2$ und somit ungerade ist.)

Wir fragen daher: Wann tritt in der Folge der Zahlen
 $(f(s)+i^2) \bmod p$ für $i=0, 1, 2, 3, \dots$
erstmal eine Zahl wieder auf?

Wenn eine Zahl erneut auftritt, so muss es zwei Zahlen i und j geben mit $i \neq j$, $i \geq 0$, $j \geq 0$ und

$$(f(s)+i^2) \bmod p = (f(s)+j^2) \bmod p,$$

$$\text{d.h. } (i^2-j^2) \bmod p = (i+j) \cdot (i-j) \bmod p = 0.$$

Wenn p eine Primzahl ist, dann muss $(i-j)$ oder $(i+j)$ durch p teilbar sein. Wir nehmen an, dass wir höchstens p Mal das quadratische Sondieren durchführen, d.h., dass $0 \leq i \leq p-1$ und $0 \leq j \leq p-1$ gelten. Dann ist $-p < (i-j) < p$ und wegen $i \neq j$ kann daher p nicht $(i-j)$ teilen. Also muss p die Zahl $(i+j)$ teilen. Das geht aber nur, wenn mindestens eine der beiden Zahlen größer als die Hälfte von $p+1$ ist. Also gilt:

[Satz 9.3.10](#) (Zykluslänge beim quadratischen Sondieren)

Beim quadratischen Sondieren kann frühestens nach $(p+1)/2$ Schritten eine Zahl erneut auftreten, sofern p eine Primzahl ist.

Für $i = (p+1)/2$ und $j = (p-1)/2$ ist $(i+j) \cdot (i-j) \pmod p = 0$, da $i+j=p$ und $i-j=1$ gilt. Die in Satz 9.3.10 genannte Länge eines Zyklus von $(p+1)/2$ Schritten tritt somit für *alle* Zahlen (auch für die Primzahlen) auf.

Hinweis: Bei einer Primzahl p kommen die "komplementären Zahlen" $(-i^2) \pmod p$ beim quadratischen Sondieren nicht vor. Modifiziert man daher das quadratische Sondieren so, dass zwischen $i^2 \pmod p$ und $(i+1)^2 \pmod p$ immer $(-i^2) \pmod p$ eingeschoben wird, so erreicht man die volle Zykluslänge p .

Übungsaufgabe:

Beweisen Sie diese Aussage und schreiben Sie eine Prozedur für diese Vorgehensweise.

Tritt beim linearen oder beim quadratischen Sondieren eine Primärkollision (= zwei verschiedene Schlüssel haben den gleichen Hashwert) auf, so wird für das Einfügen des jeweils letzten Schlüssels die gesamte Kette der Kollisionen, die die früheren Schlüssel mit gleichem Hashwert durchlaufen haben, ebenfalls durchlaufen.

Will man diesen Effekt vermeiden, so muss man eine zweite Hashfunktion g hinzunehmen, die möglichst unabhängig von f ist, d.h., für f und g sollte auf jeden Fall gelten:

$S_{m,n} = \{s \in S \mid f(s)=m \text{ und } g(s)=n\}$ enthält für alle m und n ungefähr $|S|/p^2$ Elemente.

Dies führt zu "Doppel-Hash"-Kollisionsverfahren:

Definition 9.3.11: Kollisionsstrategien (Fortsetzung)

Es seien f und g zwei unterschiedliche Hashfunktionen. Sei i die Zahl der Zugriffe (beginnend mit $i=0$). Die Kollisionsstrategie

$D(s,i) = (f(s) + i \cdot g(s)) \bmod p$ heißt "**Doppel-Hash-Verfahren**".

Es seien $f_1, f_2, f_3, f_4, \dots$ eine Folge von möglichst unterschiedlichen Hashfunktionen. Die Kollisionsstrategie

$M(s,i) = f_i(s)$ heißt "**Multi-Hash-Verfahren**".

Hinweis: In der Praxis hat man mit Doppel-Hash-Strategien gute Erfahrungen gemacht.

9.3.12 Löschen in Hashtabellen (DELETE)

Dieses bildet das Hauptproblem beim offenen Hashing.

Der einfachste Weg ist es, das Löschen durch Setzen eines Booleschen Wertes zu realisieren: Wenn der Eintrag mit dem Schlüssel s gelöscht werden soll, so suche man seine Position j auf und setze $A(j).geloescht := \text{true}$.

Beim Einfügen behandelt man dieses Feld $A(j)$ dann wie einen freien Platz (nicht aber beim Suchen).

Nachteil: Wenn oft gelöscht wird, dann ist die Tabelle schnell voll und muss mit gewissem Aufwand reorganisiert werden, vgl. Abschnitt 9.5. Dennoch ist dieses Vorgehen in der Praxis gut einsetzbar.

Hat man sich jedoch für das lineare Sondieren entschieden, dann kann man das Löschen korrekt durchführen: Man sucht den Eintrag $A(j)$ mit dem zu löschenden Element auf und geht dann die Einträge $A(j+c)$, $A(j+2c)$, $A(j+3c)$ solange durch, bis man auf eine freie Komponente stößt. In dieser Kette kopiert man alle Einträge um c , $2c$, $3c$ usw. Plätze zurück, aber niemals über die Komponente k hinaus mit $f(s)=k$.

Details: selbst überlegen! Siehe auch Übungen.

9.4 Analyse der Hashverfahren

Beim linearen Sondieren steigt die Zeit, die man für das Einfügen benötigt, wegen der Cluster mit steigendem Grad der Auslastung (d.h., wenn sich die Anzahl k der eingetragenen Schlüssel der Zahl p der Plätze in der Tabelle nähert) überproportional an. Beim quadratischen Sondieren tritt dies nicht so stark hervor. Beim Doppel-Hash-Verfahren noch weniger. (Dafür wird das Löschen jedes Mal schwieriger.)

Welche theoretischen Ergebnisse gibt es zur Analyse der Laufzeiten beim Suchen und Einfügen?

Für die Beweise benötigt man einige Annahmen. Diese fordern meist die Gleichverteilung der Schlüssel und die Unabhängigkeit von Ereignissen.

9.4.1 Annahmen:

1. Die Hashfunktion f ist gleichverteilt über die Schlüsselmenge, sie bevorzugt oder benachteiligt dort keine Bereiche.
2. Jeder Schlüssel ist beim Suchen und beim Einfügen gleichwahrscheinlich.
3. Erfolgt beim Einfügen eines Schlüssels eine Kollision, so werden bis zu dessen Eintrag auf einen freien Platz nur paarweise verschiedene Plätze besucht.

Wie lange dauert es unter diesen Annahmen im Mittel, einen Schlüssel in eine Hashtabelle einzufügen, in der bereits k von p Plätzen belegt sind?

Setze

w_i = Wahrscheinlichkeit dafür, dass für dieses Einfügen genau i Vergleiche durchgeführt werden ($1 \leq i \leq k+1$).

Wegen der Annahmen gilt: Mit der Wahrscheinlichkeit k/p trifft man beim ersten Mal auf einen belegten Platz, mit der Wahrscheinlichkeit $(k-1)/(p-1)$ beim zweiten Mal, mit $(k-2)/(p-2)$ beim dritten Mal usw. So erhalten wir die Formeln:

$$w_1 = 1 - k/p$$

$$w_2 = (k/p) \cdot (1 - (k-1)/(p-1)) , \text{ allgemein:}$$

$$w_i = (k/p) \cdot (k-1)/(p-1) \cdot (k-2)/(p-2) \cdot \dots \cdot (k-i+2)/(p-i+2) \cdot (1 - (k-i+1)/(p-i+1))$$

$$= \frac{k \cdot (k-1) \cdot (k-2) \cdot \dots \cdot (k-i+2)}{p \cdot (p-1) \cdot (p-2) \cdot \dots \cdot (p-i+2)} \left(1 - \frac{k-i+1}{p-i+1}\right)$$

Dann lautet die mittlere Zahl der Vergleiche E_{k+1} beim Einfügen eines $(k+1)$ -ten Schlüssels in eine Hashtabelle der Größe p :

$$E_{k+1} = \sum_{i=1}^{k+1} i \cdot w_i = \dots = \frac{p+1}{p+1-k} = \frac{1}{1-\lambda} \quad \text{mit } \lambda = k/(p+1)$$

$$\lambda \approx \text{"Auslastungsgrad"} \quad k/p$$

(Dieses Ergebnis lässt sich nicht allzu schwer herleiten. Versuchen Sie es selbst einmal.)

Satz 9.4.2 Unter den Annahmen 9.4.1 gilt:

Um den $(k+1)$ -ten Schlüssel in eine Hashtabelle der Größe p einzufügen, werden im Mittel $\frac{p+1}{p+1-k} = \frac{1}{1-\lambda}$ Vergleiche (hier: mit $\lambda = k/(p+1)$) benötigt.

Einige Funktionswerte für $E_{k+1} = \frac{p+1}{p+1-k}$

λ	E_{k+1}
0,1	1,11
0,2	1,25
0,3	1,43
0,4	1,67

λ	E_{k+1}
0,5	2,00
0,6	2,50
0,7	3,33
0,8	5,00

λ	E_{k+1}
0,85	6,67
0,88	8,33
0,90	10,00
0,95	20,00

9.4.3: Wie lange dauert bei "idealen Hashfunktionen" die *erfolgreiche Suche* im Mittel und wie lange die *erfolglose Suche*, wenn k der p Plätze belegt sind?

Die erfolglose Suche entspricht dem Einfügen eines neuen Schlüssels; sie benötigt also E_{k+1} Vergleiche.

Es sei S_k die Zahl der erforderlichen Vergleiche, um einen Schlüssel zu finden, der in einer Hashtabelle der Größe p mit dem Auslastungsgrad k/p steht.

Die erfolgreiche Suche kann man dann durch folgende Formel beschreiben:

$$\mathbf{S_k} = \frac{1}{k} \sum_{i=0}^{k-1} \mathbf{E_{i+1}} ,$$

denn der gesuchte Schlüssel muss in einem der Schritte 1, 2, 3, ..., k in die Tabelle eingefügt worden sein, und nach der Annahme 2 können wir den Mittelwert der Zahl der Vergleiche nehmen. Durch Auswerten dieser Formel erhält man:

$$\mathbf{S_k} \approx \frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-\lambda}\right)$$

(Dieses Ergebnis soll evtl. in den Übungen ausgerechnet werden.)

Satz 9.4.4 Unter den Annahmen 9.4.1 gilt:

Für die erfolgreiche Suche nach einem Schlüssel in einer

Hashtabelle der Größe p werden im Mittel $S_k \approx \frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-\lambda}\right)$

Vergleiche (hier: mit $\lambda = k/(p+1)$) benötigt.

Man beachte, dass $\frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-\lambda}\right) \leq \frac{1}{1-\lambda}$ ist wegen

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \geq 1 + x \quad \text{mit} \quad x = \frac{\lambda}{1-\lambda} \quad \text{für} \quad x \geq 0, \text{ also } 1 > \lambda \geq 0.$$

Experimente haben ergeben, dass Doppel-Hash-Verfahren recht gut den Wert S_k annähern. Wie wirken sich Kollisionen aus?

Es sei $Slin_k$ die *mittlere Suchzeit für die erfolgreiche Suche* bei der linearen Kollisionsstrategie, dann kann man mit einigem Aufwand beweisen (ohne nähere Erläuterung hier):

$$Slin_k \approx \frac{1 - \frac{\lambda}{2}}{1 - \lambda}$$

Einige Werte zu S_k und $Slin_k$ mit $\lambda = k/(p+1)$:

λ	S_k	$Slin_k$
0,50	1,39	1,50
0,75	1,85	2,50
0,80	2,01	3,00
0,90	2,56	5,50
0,95	3,15	10,50
0,99	4,65	50,50

Für die Praxis, die meist mit linearem Sondieren arbeitet, folgt hieraus: Man begrenze den Auslastungsgrad möglichst auf 80%.



9.5 Rehashing

Was muss man tun, wenn der Auslastungsgrad über 80% hinausgeht oder gar den Wert 1 erreicht? Dann muss man die Hashtabelle verlängern, also p durch eine Zahl $p' > p$ ersetzen, hierfür eine neue Hashfunktion festlegen und die neue Hashtabelle aus der alten Tabelle, in der die Schlüssel in den Plätzen von 0 bis $p-1$ standen, errechnen.

Diesen Vorgang der Umorganisation innerhalb der bestehenden Hashtabelle bezeichnen wir als "**Rehashing**". Dieses Verfahren wird auch verwendet, wenn man Schlüssel, statt sie zu löschen, nur als "gelöscht" markiert, wodurch im Laufe der Zeit der Auslastungsgrad zu groß und eine Umorganisation mit dem gleichen p notwendig wird (Rehashing mit $p=p'$).

9.5.1 Beispiel für $p = 7$ und $p' = 13$

	0
	1
MAE	2
JAN	3
FEB	4
APR	5
MAI	6

$p=7$

Wörter:

JAN, FEB, MAE, APR, MAI.

Alte Hashfunktion (lin. Sond.):

$$f(\alpha_1 \alpha_2 \dots \alpha_r) = (\varphi(\alpha_1) + 2 \varphi(\alpha_3)) \bmod 7.$$



0	
1	
2	
3	
4	
5	MAE
6	APR
7	
8	FEB
9	MAI
10	
11	JAN
12	

$p'=13$

Alle Wörter müssen übertragen werden.

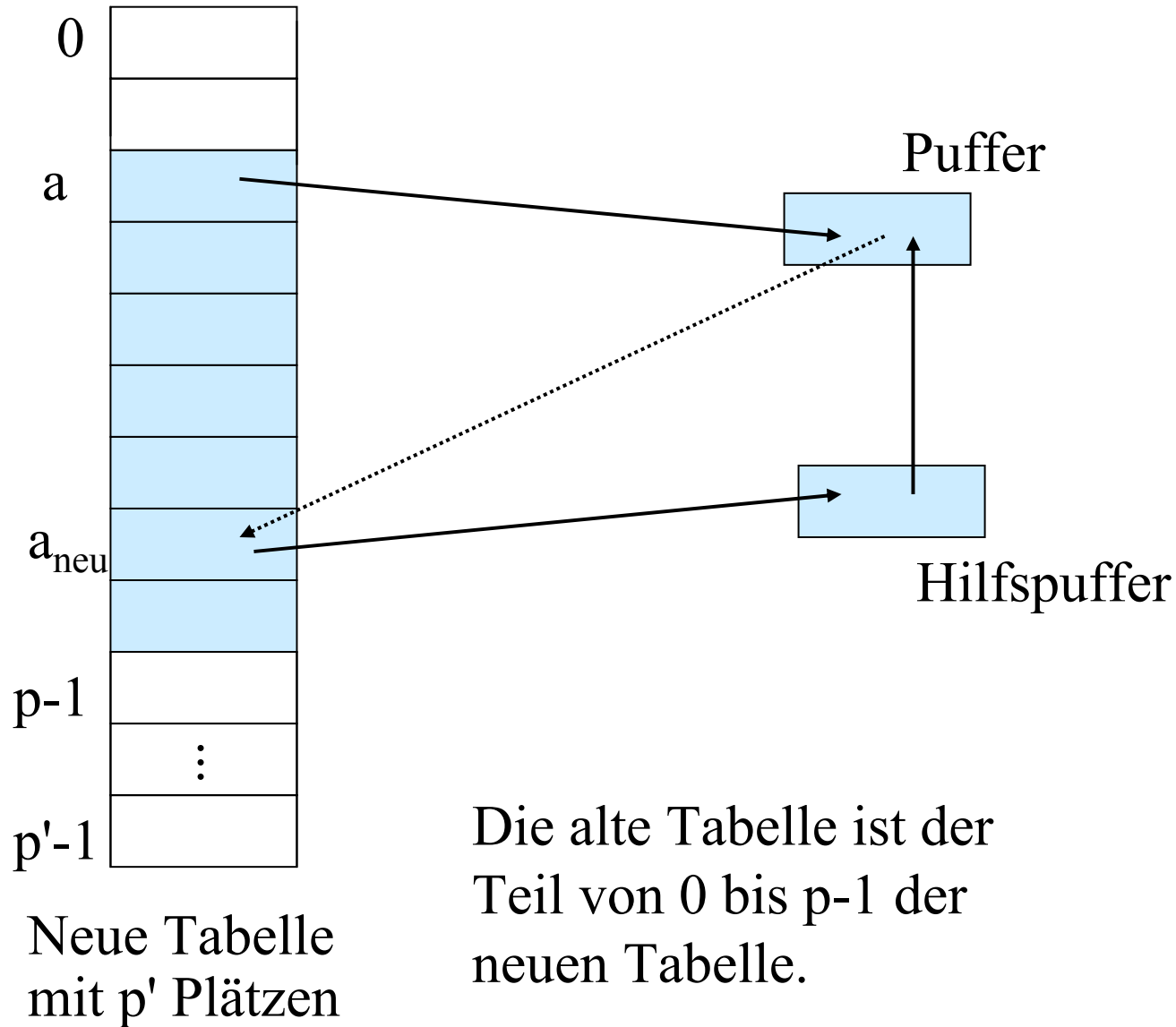
Neue Hashfunktion:
 $f'(\alpha_1 \alpha_2 \dots \alpha_r) = (\varphi(\alpha_1) + \varphi(\alpha_3)) \bmod 13.$
 Lineares Sondieren.

Von oben nach unten durchgehen und dabei umsortieren!
 Wir fangen also mit MAE an, dann JAN usw.

In diesem Beispiel haben wir die Wörter JAN, FEB, MAE, APR, MAI in dieser Reihenfolge in die neue Tabelle mit $p'=13$ eingetragen. Dies entspricht aber nicht dem gewünschten "Rehashing", weil beispielsweise $f'(\text{MAE}) = 5$ ist, aber auf Platz 5 steht APR und dieses Wort würde hierbei überschrieben werden. Faktisch haben wir also nicht *innerhalb* der neuen Tabelle umorganisiert, sondern wir haben neben die alte Tabelle mit $p=7$ Plätzen eine neue Tabelle mit $p'=13$ Plätzen gelegt und die Wörter dorthin entsprechend der neuen Hashfunktion f' umgespeichert. Wir brauchten also insgesamt $p+p'=20$ Plätze.

Unser Rehash-Verfahren soll jedoch auf der verlängerten Ausgangstabelle arbeiten, also mit insgesamt p' Plätzen auskommen. Lösung dieses Problems:

9.5.2 Rehashing: Durchlaufe die neue Tabelle von 0 bis $p-1$:



Man kommt also mit zwei Zusatzvariablen "Puffer" und "Hilfspuffer" aus, in denen die gerade betrachteten oder die weiter zu verschiebenden Elemente zwischengespeichert werden. Nun zur Programmierung:

Erinnerung: (siehe 9.3.1)

type Eintragtyp is record

belegt: Boolean; geloescht: Boolean;

kollision: Boolean; behandelt: Boolean;

Schluessel: Schluesseltyp;

Inhalt: Inhalttyp;

end record;

type Hashtabelle is array(0..p-1) of Eintragtyp;

A: Hashtabelle;

Programm für das Rehashing

Bevor die Hashtabelle erstmals benutzt wird, wurde gesetzt:

```
for j in 0..p-1 loop  A(j).belegt:=false; A(j).kollision:=false;  
                        A(j).geloescht:=false; A(j).behandelt:=false; end loop;
```

Während der Verwendung der Tabelle A wurden A(j).besetzt, A(j).kollision und A(j).geloescht eventuell verändert.

Erforderliche Variablen:

Puffer, Hilfspuffer: Eintragtyp;

Berechne p' als neue Größe der Hashtabelle A. Diese besitzt dann also die Grenzen von 0 bis $p'-1$.

Die verwendete Hashfunktion f' möge zwei Parameter haben: den Schlüssel und die Anzahl i der Zugriffe (= die Anzahl der bisherigen Kollisionen).

Vorgehensweise: Führe (1) bis (3) von $a=0$ bis $a=p-1$ durch.

- (1) *$A(a).belegt$ and not $A(a).gelöscht$ and not $A(a).behandelt$* : kopiere $A(a)$ in den Puffer und setze $A(a).belegt$ auf false.
- (2) Berechne in diesem Fall mit der neuen Hashfunktion f' den neuen Index a_{neu} , wohin das Element des Puffers hingehört.

- (3) Unterscheide hierzu folgende Fälle:
not $A(a_{neu}).belegt$ or $A(a_{neu}).geloescht$: Kopiere den Puffer nach $A(a_{neu})$; setze $A(a_{neu}).belegt$ und $A(a_{neu}).behandelt$ auf true. Erhöhe a . Weiter bei (1).

not $A(a_{neu}).behandelt$ and $A(a_{neu}).belegt$: Kopiere $A(a_{neu})$ in den Hilfspuffer; kopiere den Puffer nach $A(a_{neu})$; setze $A(a_{neu}).behandelt$ und $A(a_{neu}).belegt$ auf true. Kopiere dann den Hilfspuffer in den Puffer. Weiter bei (2).

Sonst, d.h.: *$A(a_{neu}).behandelt$* : Setze $A(a_{neu}).kollision := true$, berechne den Index a_{neu} neu, weiter bei (3).

Programmstück in Ada zum Rehashing

-- a durchläuft die Adressen von 0 bis p-1,
-- i zählt die Zahl der auftretenden neuen Kollisionen,
-- aneu gibt die Adresse an, wohin der Eintrag in der neuen
-- Tabelle gehört.

for a in 0..p-1 loop

if A(a).geloescht then "lösche den Eintrag A(a)"

elsif A(a).belegt and not A(a).behandelt then

 Puffer := A(a); Puffer.belegt := true;

 A(a).belegt := false;

 i := 0;

-- nun f' auf Puffer anwenden und Adresse aneu berechnen,
-- auf Kollisionen mit schon behandelten Einträgen achten.

(Programmstück in Ada zum Rehashing, Fortsetzung)

```
while Puffer.belegt loop  
    aneu := f'(Puffer.schluessel, i);  i := i+1;  
    if not A(aneu).belegt or A(aneu).geloescht then  
        A(aneu) := Puffer;  
        A(aneu).geloescht := false; A(aneu).belegt := true;  
        A(aneu).behandelt := true; A(aneu).kollision:=false;  
        Puffer.belegt:= false;  
    elsif not A(aneu).behandelt then  
        Hilfspuffer := A(aneu); Hilfspuffer.belegt := true;  
        A(aneu) := Puffer; A(aneu).behandelt := true;  
        Puffer := Hilfspuffer;  
        i := 0;  
    else A(aneu).kollision := true; end if;  
    end loop;  
end if; end loop a;
```

Zeitaufwand hierfür? (Selbst durchdenken.)

Man kennt mittlerweile viele theoretische Resultate über das Hashing. Schauen Sie in der Literatur nach.

9.6 Noch ein Beispiel

Auf den folgenden Folien ist nochmals ein Beispiel für das lineare Sondieren angegeben, wobei die zusätzlichen Booleschen Werte miteingetragen sind. Setzen Sie dieses Beispiel fort, indem Sie Elemente wieder explizit löschen bzw. ein Rehashing mit größerem p' durchführen. Machen Sie sich hieran die Schwierigkeiten beim quadratischen Sondieren klar, insbesondere das Problem eines auftretenden unendlichen Zyklus.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							

$$p = 13$$

A = {JANUAR,
FEBRUAR,
MAERZ,
APRIL, MAI,
JUNI, JULI,
AUGUST,
SEPTEMBER,
OKTOBER,
NOVEMBER,
DEZEMBER},

Hashfunktion f :
Nummer des ersten
plus Nummer des
zweiten Buchstabens,
modulo 13.

JANUAR	-	-	-	-	11	?
FEBRUAR	-	-	-	-	11	?
MAERZ	-	-	-	-	1	?
APRIL	-	-	-	-	4	?
MAI	-	-	-	-	1	?
JUNI	-	-	-	-	5	?
JULI	-	-	-	-	5	?
AUGUST	-	-	-	-	9	?
SEPTEMBER	-	-	-	-	11	?
OKTOBER	-	-	-	-	0	?
NOVEMBER	-	-	-	-	3	?
DEZEMBER	-	-	-	-	9	?

- = false
+ = true

Hashfunktion f : Nummer des ersten plus Nummer des zweiten Buchstabens, modulo 13.

Nun tragen wir diese Werte nacheinander in die Hashtabelle mit $p = 13$ Plätzen ein, lineare Kollisionstrategie, $c = 1$. Das Feld für "Inhalt" interessiert nicht und wird stets "?" gesetzt.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11	JANUAR	+	-	-	-	11	?
12							

FEBRUAR

erzeugt eine Kollision
am Platz 11 und wird
dann im Platz 12
eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1	MAERZ	+	-	-	-	1	?
2							
3							
4	APRIL	+	-	-	-	4	?
5							
6							
7							
8							
9							
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	-	-	11	?

MAI

erzeugt eine Kollision
am Platz 1 und wird
dann im Platz 2
eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	-	-	1	?
3							
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	-	-	5	?
6							
7							
8							
9							
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	-	-	11	?

JULI

erzeugt eine Kollision
am Platz 5 und wird
dann im Platz 6
eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0							
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	-	-	1	?
3							
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	-	-	5	?
7							
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	-	-	11	?

SEPTEMBER

erzeugt eine Kollision
am Platz 11 und am
Platz 12 und wird dann
im Platz 0 eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	-	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	-	-	1	?
3							
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	-	-	5	?
7							
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

OKTOBER

erzeugt eine Kollision
am Platz 0, 1 und 2 und
wird dann im Platz 3
eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	+	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	+	-	1	?
3	OKTOBER	+	-	-	-	0	?
4	APRIL	+	-	-	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	-	-	5	?
7							
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

NOVEMBER

erzeugt eine Kollision
am Platz 3, 4, 5 und 6
wird dann im Platz 7
eingetragen.

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	+	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	+	-	1	?
3	OKTOBER	+	-	+	-	0	?
4	APRIL	+	-	+	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	+	-	5	?
7	NOVEMBER	+	-	-	-	3	?
8							
9	AUGUST	+	-	-	-	9	?
10							
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

DEZEMBER

erzeugt eine Kollision
am Platz 9 wird dann
im Platz 10 eingetragen.

Ergebnistabelle

	Schlüssel	belegt	gelöscht	Kollision	behandelt	Hashwert	Inhalt
0	SEPTEMBER	+	-	+	-	11	?
1	MAERZ	+	-	+	-	1	?
2	MAI	+	-	+	-	1	?
3	OKTOBER	+	-	+	-	0	?
4	APRIL	+	-	+	-	4	?
5	JUNI	+	-	+	-	5	?
6	JULI	+	-	+	-	5	?
7	NOVEMBER	+	-	-	-	3	?
8							
9	AUGUST	+	-	+	-	9	?
10	DEZEMBER	+	-	-	-	9	?
11	JANUAR	+	-	+	-	11	?
12	FEBRUAR	+	-	+	-	11	?

Überlegen Sie sich:

Wie sähe eine "optimale" Reihenfolge der Eintragungen aus, so dass in der Ergebnistabelle minimal viele Kollisions-Bits auf true gesetzt sind.

Welche Tabelle erhält man bei einer quadratischen Kollisionsstrategie?

Berechnen Sie die mittlere Zugriffszeit bei den verschiedenen Tabellen. Trifft es hier zu, dass die quadratische Strategie besser als die lineare ist?

Hinweis: Schön wäre es, wenn auch die folgenden Operationen leicht ausführbar wären.

- | | |
|--|---------------------|
| - Gib die Elemente von B_1 geordnet aus. | SORT |
| - Vereinige B_1 und B_2 . | UNION |
| - Bilde den Durchschnitt von B_1 und B_2 . | INTERSECTION |
| - Entscheide, ob B_1 leer ist. | EMPTINESS |
| - Entscheide, ob $B_1 = B_2$ ist. | EQUALITY |
| - Entscheide, ob $B_1 \subseteq B_2$ ist. | SUBSET |

Überlegen Sie sich, welche dieser Operationen mit Hilfe des Hashings mit welchem Aufwand (Zeit und Platz) durchgeführt werden können. Welche Zusatzinformationen sollte der Datentyp "Hashtabelle über Schlüsseldatentyp" besitzen und wie würde er in Ada95 spezifiziert und implementiert?

Einführung in die Informatik II

Universität Stuttgart, Studienjahr 2007

Gliederung der Grundvorlesung

~~8. Suchen~~

~~9. Hashing~~

10. Sortieren

11. Graphalgorithmen

12. Speicherverwaltung

10. Sortieren

10.1 Überblick

10.2 Sortieren durch Austauschen

10.3 Sortieren durch Einfügen

10.4 Sortieren durch Aussuchen/Auswählen

10.5 Mischen (meist für externes Sortieren)

10.6 Streuen und Sammeln

10.7 Paralleles Sortieren

Ziele des 10. Kapitels:

Um Dinge schneller wiederfinden zu können oder um den Überblick zu bewahren, legt man Dokumente in geordneter Form ab. Hierzu muss der Datenbestand sortiert werden.

In diesem Kapitel lernen Sie, welche Sortierverfahren es gibt und welche Eigenschaften, insbesondere welche Komplexität sie besitzen. Da man bei sequenziellem Vorgehen mindestens $n \cdot \log(n)$ Vergleiche benötigt, werden auch deutlich schnellere parallel arbeitende Sortierverfahren vorgestellt, die allerdings recht viele Bausteine erfordern.

Zugleich wiederholen wir aus Kapitel 7 den Nachweis der Korrektheit für einige dieser Verfahren (vor allem in den Übungen).

10.1 Überblick

Suchen und Sortieren machen einen Großteil aller Verwaltungstätigkeiten im Rechner aus, wo Daten ständig abgelegt und schnell wiedergefunden werden müssen. Das Sortieren nutzt die Anordnung der Schlüsselmenge direkt aus und ermöglicht ein schnelles Auffinden: $O(\log(n))$ durch Intervallsuche oder $O(\log(\log(n)))$ durch Interpolationssuche, siehe Abschnitt 6.5.1 und Abschnitt 8.1.

Wir klären als erstes den Begriff "Sortieren" und leiten dann eine *untere* Schranke für die Zahl der Vergleiche her, die bei einem Sortierverfahren, das ausschließlich auf Vergleichen basiert, erforderlich ist. Sodann geben wir einen Überblick über gängige Sortierverfahren und deren Komplexität.

Definition 10.1.1: Sortierte Folgen

Gegeben sei eine endliche oder unendliche Menge mit totaler Ordnung $A = \{a_1, \dots, a_s\}$ oder $A = \{a_1, a_2, a_3, a_4, \dots\}$ mit $a_1 < a_2 < a_3 < a_4 < \dots$.

- (1) Eine Folge $v = v_1 v_2 \dots v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt (aufsteigend) geordnet genau dann, wenn gilt $v_1 \leq v_2 \leq \dots \leq v_n$.
(Speziell ist jede leere oder einelementige Folge geordnet.)
- (2) Eine Folge $v = v_1 v_2 \dots v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt invers geordnet oder absteigend geordnet $\Leftrightarrow v_n \leq v_{n-1} \leq \dots \leq v_1$.
(Speziell ist jede leere oder einelementige Folge invers geordnet.)

Die Sortieraufgabe lautet: Ordne eine Folge von n Elementen so um, dass sie sortiert ist. Wir präzisieren dies nun.

Definition 10.1.2: Permutationen

Es sei n eine natürliche Zahl.

Eine bijektive Abbildung $\pi: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ heißt Permutation der Ordnung n .

Erinnerung:

bijektiv = injektiv und surjektiv, d.h.,

injektiv: für je zwei Elementen $i \neq j$ gilt $\pi(i) \neq \pi(j)$ und

surjektiv: zu jedem j existiert ein i mit $\pi(i) = j$.

Eine Permutation ist also nur eine Umordnung der n Elemente.

Hinweis 1: Bekanntlich gibt es genau $n!$ verschiedene Permutationen der Ordnung n .

Hinweis 2: Da π bijektiv ist, existiert auch die inverse Abbildung $\pi^{-1}: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$, definiert durch $[\pi^{-1}(i) = j \Leftrightarrow \pi(j) = i]$. π^{-1} ist ebenfalls eine Permutation.

Definition 10.1.3: Die Sortieraufgabe lautet:

Finde zu einer beliebigen Folge $v = v_1 v_2 \dots v_n \in A^*$ eine Permutation π der Ordnung n mit: $v_{\pi(1)} v_{\pi(2)} \dots v_{\pi(n)}$ ist sortiert (oder invers sortiert, je nach Anwendung).

Ein Algorithmus, welcher $v_1 v_2 \dots v_n$ in die sortierte Folge $v_{\pi(1)} v_{\pi(2)} \dots v_{\pi(n)}$ überführt, heißt Sortierverfahren. Hierbei wird das i -te Element v_i der Folge an die Position $\pi^{-1}(i)$ gesetzt.

Anschaulich gesprochen konstruieren Sortierverfahren die Permutation π^{-1} . Denn man gibt an, *wohin* ein Element der Folge verschoben werden muss (und nicht, *woher* es gekommen ist). Da mit π auch π^{-1} eine Permutation ist, ist es für den Formalismus egal, ob man das Sortieren über π oder über π^{-1} definiert. Für uns Menschen spielt dies aber eine wesentliche Rolle und Sie sollten sich bei konkreten Problemen stets im Klaren darüber sein, ob π oder π^{-1} berechnet wird!

Beispiel zum Formalismus:

Es sei $5, 8, 2, 5, 1, 8$ die gegebene Folge $v_1 v_2 \dots v_n \in A^*$ mit $A = \{1, 2, 5, 8\}$, $n=6$ sowie $v_1=5, v_2=8, v_3=2, v_4=5, v_5=1$ und $v_6=8$.

Die geordnete Folge $v_{\pi(1)} v_{\pi(2)} \dots v_{\pi(n)}$ lautet $1, 2, 5, 5, 8, 8$; man kann also π folgendermaßen festlegen:

$\pi(1)=5, \pi(2)=3, \pi(3)=1, \pi(4)=4, \pi(5)=2, \pi(6)=6$;
denn dann gilt

$$v_{\pi(1)} v_{\pi(2)} v_{\pi(3)} v_{\pi(4)} v_{\pi(5)} v_{\pi(6)} = v_5 v_3 v_1 v_4 v_2 v_6 = 1, 2, 5, 5, 8, 8.$$

(Die Kommata rechts müssen Sie sich wegdenken. Wir schreiben in Beispielen die Folgen der Deutlichkeit halber oft mit dem besser sichtbaren Trennzeichen "Komma" auf; aus Sicht der Definition ist dies aber nicht notwendig.)

$\pi^{-1}(1)=3, \pi^{-1}(2)=5, \pi^{-1}(3)=2, \pi^{-1}(4)=4, \pi^{-1}(5)=1, \pi^{-1}(6)=6$ gibt an, wohin das i -te Element der Folge verschoben wird.



Meist hängt ein Sortieralgorithmus ab von Fragen wie:

- Wie sind die Daten gegeben, zu welchen Mengen gehören sie?
- Wo liegen die Daten? (Hauptspeicher, Platten, Bänder, ...?)
- Zulässige Operationen? (Vergleichen, vertauschen ...?)
- Gibt es Elemente mit gleichem Schlüssel? (\Rightarrow Stabilität.)
- Nur Zugriffsstruktur oder Gesamtdaten sortieren?
- Darf man zusätzlichen Speicher verwenden oder nicht?
- Sequentielles, paralleles, verteiltes Sortieren?
- Effizienz im Mittel oder auch im schlimmsten Fall?
- Erkennen oder Beachten von Vorsortierungen?

10.1.4: Jedes Element v_i der zu sortierenden Folge $v = v_1 v_2 \dots v_n$ ist in der Praxis meist ein umfangreiches Dokument. Die Folge wird in der Regel nur nach *einem* Ordnungskriterium sortiert.

Fall 1: Dieses Kriterium wird durch eine Funktion $g: A \rightarrow M$ beschrieben, wobei M eine geordnete Menge ist (A braucht in diesem Fall gar nicht sortierbar zu sein). $g(v_i) = k_i$ heißt dann der Schlüssel von v_i , der in der Regel im record v_i enthalten ist. (Wir können also stets Fall 2 annehmen.)

Fall 2: Das Ordnungskriterium bezieht sich auf eine oder mehrere Komponenten des Dokuments. Den Vektor dieser Komponenten, nach denen zu sortieren ist, bezeichnen wir als Schlüssel. Man verlangt oft, dass dieser ein Element eindeutig beschreibt, doch lassen wir hier auch verschiedene Elemente mit gleichem Schlüssel zu.

In der Regel sortiert man nur die Schlüssel einer Folge.

Beispiel: Folgende Folge aus Name und Alter

(Beier,23), (Zahn,40), (Fuhr,30), (Horn,41), (Beier,30), (Horn,17)
wird zur Folge

(Beier,23), (Beier,30), (Fuhr,30), (Horn,41), (Horn,17), (Zahn,40),
sofern nach dem Namen sortiert wird. Selbstverständlich hat
man bei *gleichen Schlüsseln* eine Wahlfreiheit. Statt dieser
Reihenfolge hätte man auch die Folge

(Beier,30), (Beier,23), (Fuhr,30), (Horn,17), (Horn,41), (Zahn,40)
als korrektes Ergebnis erhalten können. Die erste Folge hat
den Vorteil, dass dort die relative Anordnung von Elementen
mit gleichem Schlüssel erhalten blieb (ein solches Verfahren
nennt man "stabil").

Dagegen hätte man die Folge

(Horn,17), (Beier,23), (Beier,30), (Fuhr,30), (Zahn,40), (Horn,41),
erhalten können, falls nach dem Alter sortiert wird.



Wird also eine Folge v nach mehreren Komponenten (Schlüsseln) nacheinander geordnet, so kann man die Folge zunächst nach dem am wenigsten relevanten Kriterium (im Beispiel: nach dem Alter) und dann schrittweise nach dem nächstwichtigeren Kriterium sortieren. Hierfür muss ein Sortierv Verfahren "stabil" sein.

Definition 10.1.5:

Ein Sortierv Verfahren $Sort$ heißt stabil, wenn $Sort$ die Reihenfolge von Elementen mit gleichem Schlüssel nicht verändert, d.h.:

Wenn $Sort(v_1 v_2 \dots v_n) = v_{\pi(1)} v_{\pi(2)} \dots v_{\pi(n)}$ ist, so gilt für alle $i < j$ mit $v_i = v_j$ stets $\pi^{-1}(i) < \pi^{-1}(j)$ [d.h., stand in der ursprünglichen Folge v_i vor v_j , so gilt dies auch für die sortierte Folge, vgl. 10.1.3].

$Sort$ heißt invers stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von $Sort$ gespiegelt wird.

Beispiel: Ein stabiles Sortierverfahren wird aus der Folge
(Beier,23), (Zahn,40), (Fuhr,30), (Horn,41), (Beier,30), (Horn,17)
beim Sortieren nach dem Alter die Folge
(Horn,17), (Beier,23), (Fuhr,30), (Beier,30), (Zahn,40), (Horn,41)
liefern, und das anschließende Sortieren nach dem Namen
ergibt:

(Beier,23), (Beier,30), (Fuhr,30), (Horn,17), (Horn,41), (Zahn,40).

Wir erhalten also die Reihenfolge, die wir erwarten würden:
Die Liste ist nach dem Namen sortiert und gleiche Namen
sind aufsteigend nach dem Alter angeordnet.

Prüfen Sie Definition 10.1.5 nach, indem Sie die Permutation
 π präzise angeben.



Wir haben bereits einige Sortiervverfahren kennen gelernt:

In 4.4.4 und 7.3.3: Sortieren durch Austauschen benachbarter, falsch stehender Elemente (*Bubble Sort*).

In 3.7.7 und 8.2.16: *Baumsortieren*, indem die Glieder der Folge nacheinander in einen binären Suchbaum eingefügt und anschließend in inorder-Reihenfolge ausgelesen werden.

In 7.3.3, 7.3.4 und 8.2.16: *Quicksort*. Man wählt ein "Pivot"-Element p aus und spaltet die in einem array gespeicherte Folge in zwei Teilfolgen, von denen alle Elemente der ersten Teilfolge kleiner oder gleich p und alle Elemente der zweiten Teilfolge größer oder gleich p sind. Danach: rekursiv weiter mit beiden Teilfolgen, sofern die jeweilige Teilfolge noch mindestens zwei Elemente besitzt.

Frage an Sie: Welche dieser drei Verfahren sind stabil?

Sortieren erfordert in der Praxis viele Umspeicherungsoperationen. Sind die Elemente sehr groß, so kostet dies viel Zeit. Man zieht daher die Schlüssel und den Verweis auf den jeweiligen Datensatz heraus und sortiert nur diese Schlüssel-Verweis-Tabelle. Wir werden also unseren Sortiervorgang auf die Elemente des Datentyps

```
record key: <Typ des Schlüssels>;  
          zeiger: <Zeigertyp auf den Elementtyp>;  
end record
```

zugrunde legen. Es genügt, sich nur auf die Sortierung der Schlüssel zu beschränken.

Definition 10.1.6:

Für eine Permutation $\pi: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ heißt

$$\mathfrak{I}(\pi) = \left| \{ (i, j) \mid i < j \text{ und } \pi(i) > \pi(j) \} \right|$$

die Inversionszahl (oder der Fehlstand) von π .

Analog: Für eine Folge $v = v_1 v_2 \dots v_n \in A^*$ (A sei eine geordnete

Menge) heißt $\mathfrak{I}(v) = \left| \{ (i, j) \mid i < j \text{ und } v_i > v_j \} \right|$

die Inversionszahl (oder der Fehlstand) von v .

Wegen $\mathfrak{I}(v_1 v_2 \dots v_n) + \mathfrak{I}(v_n v_{n-1} \dots v_1) = \left| \{ (i, j) \mid i < j \} \right|$ gilt der

Hilfssatz 10.1.7: $\mathfrak{I}(v_1 v_2 \dots v_n) + \mathfrak{I}(v_n v_{n-1} \dots v_1) = \frac{1}{2} \cdot n \cdot (n-1)$.

Wegen $\mathfrak{I}(1 \ 2 \ 3 \ \dots \ n) = 0$ folgt $\mathfrak{I}(n \ n-1 \ \dots \ 2 \ 1) = \frac{1}{2} \cdot n \cdot (n-1)$.

Zeigen Sie: $\mathfrak{I}(\pi) = \mathfrak{I}(\pi^{-1})$ für alle Permutationen π .

Begriffsbeschreibung 10.1.8:

Ein Sortiervorgehen *Sort* heit ordnungsvertrglich \Leftrightarrow
Je geordneter die zu sortierende Folge bereits ist,
umso schneller arbeitet *Sort*.

Genauer:

Wenn *Sort* fr zwei Folgen $v = v_1 v_2 \dots v_n$ und $w = w_1 w_2 \dots w_n$ die Permutationen π_1 und π_2 realisiert und wenn die Inversionszahl von π_1 kleiner als die von π_2 ist, dann bentigt *Sort* zum Sortieren von v nicht mehr Zeit als zum Sortieren von w .

Hinweis: Eigentlich realisiert *Sort* die Permutationen π_1^{-1} und π_2^{-1} , siehe 10.1.3. Weil $\mathfrak{I}(\pi) = \mathfrak{I}(\pi^{-1})$ fr alle Permutationen π gilt und weil im Folgenden nur die Absolutbetrge untersucht werden, kann man stattdessen auch π_1 und π_2 betrachten.

Wir betrachten die Operation "*benachbartes Vertauschen*".
Diese Operation überführt eine Folge

in die Folge

$$\begin{array}{ccccccccccc} V_1 & V_2 & \dots & V_{i-1} & V_i & V_{i+1} & V_{i+2} & \dots & V_n \\ V_1 & V_2 & \dots & V_{i-1} & V_{i+1} & V_i & V_{i+2} & \dots & V_n \end{array} \quad (\text{für ein } 0 < i < n).$$

Hierfür gilt:

$$\left| \mathfrak{J}(V_1 V_2 \dots V_{i-1} V_i V_{i+1} V_{i+2} \dots V_n) - \mathfrak{J}(V_1 V_2 \dots V_{i-1} V_{i+1} V_i V_{i+2} \dots V_n) \right| = 1.$$

Wegen 10.1.7 haben wir somit gezeigt:

Hilfssatz 10.1.9:

Ein Sortierverfahren, das ausschließlich mit der Operation "*benachbartes Vertauschen*" arbeitet, benötigt im worst case mindestens $\frac{1}{2} \cdot n \cdot (n-1)$ Schritte.

Wir betrachten die Operation "*beliebiges Vertauschen*". Diese überführt eine Folge

(für $1 \leq i \leq j \leq n$)
in die Folge

$$\begin{array}{ccccccccccc} & & & & & & \text{↩} & & \text{↪} & & \\ v_1 & v_2 & \dots & v_{i-1} & v_i & v_{i+1} & \dots & v_{j-1} & v_j & v_{j+1} & \dots & v_n \\ v_1 & v_2 & \dots & v_{i-1} & v_j & v_{i+1} & \dots & v_{j-1} & v_i & v_{j+1} & \dots & v_n \end{array}$$

Hierfür gilt:

$$0 \leq \left| \mathfrak{I}(v_1 \dots v_i \dots v_j \dots v_n) - \mathfrak{I}(v_1 \dots v_j \dots v_i \dots v_n) \right| \leq 2(j-i)-1 \leq 2n-3,$$

wobei der größte Wert $2n-3$ nur erreicht wird, wenn das kleinste Element am Ende und das größte am Anfang stand und genau diese beiden vertauscht wurden. Es folgt:

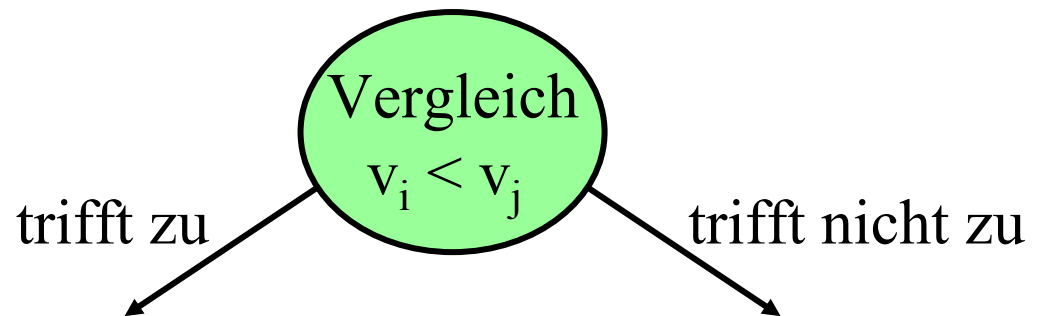
Hilfssatz 10.1.10: Ein Sortiervverfahren, das ausschließlich mit der Operation "Vertauschen" arbeitet, benötigt im worst case mindestens $n \cdot (n-1) / (4n-6) > \frac{1}{4} \cdot n$ Schritte.

(Man kann diese Schranke noch verschärfen. Versuchen Sie, die bestmögliche untere Schranke zu finden.)

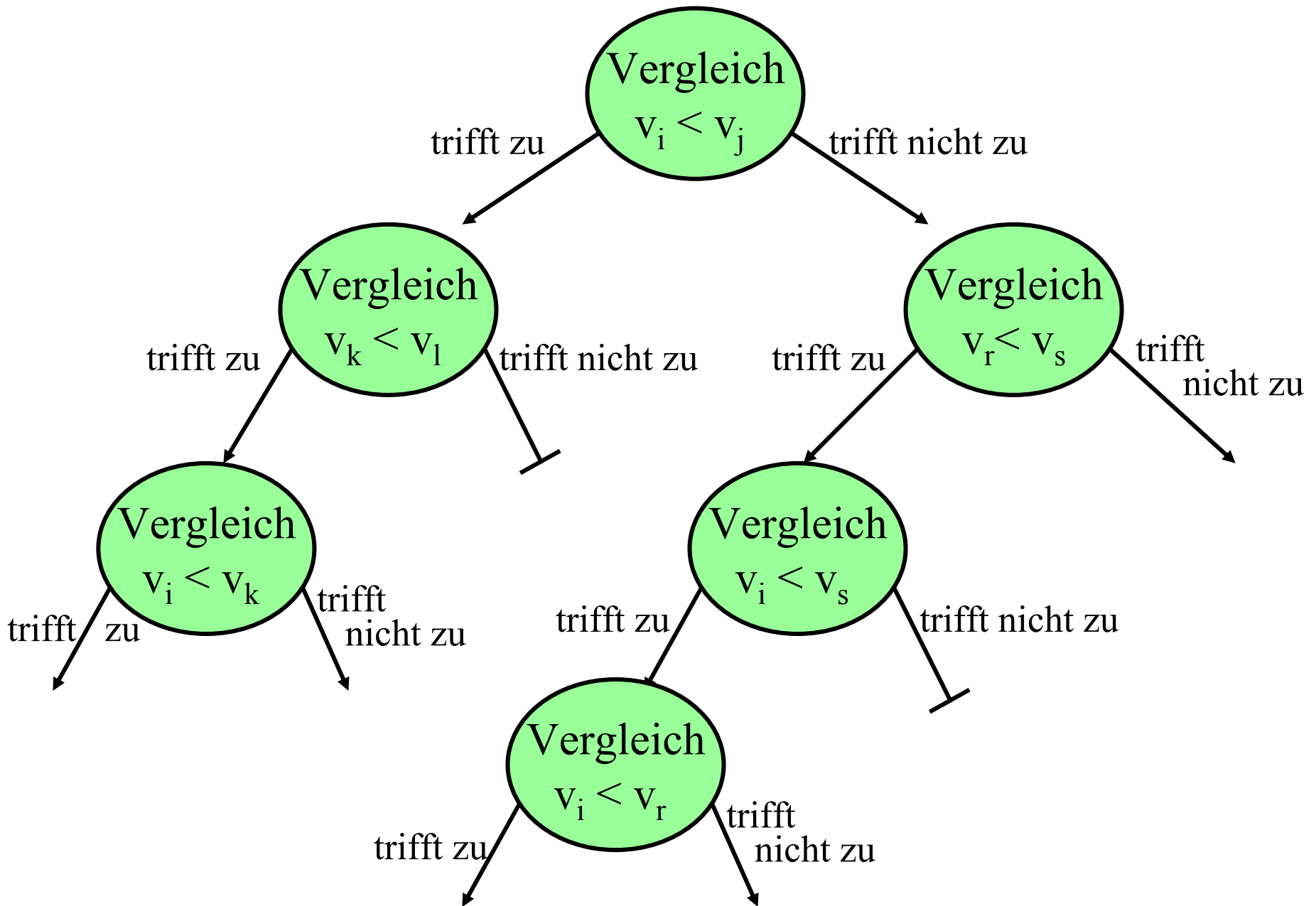
In der Regel weiß man nicht, ob man zwei Elemente einer Folge vertauschen soll. Diese Entscheidung wird durch einen Vergleich getroffen: Falls $v_i < v_j$ und $i > j$ ist, dann vertauscht man diese beiden Elemente in der Folge.

Wird das Vertauschen durch vorher gehende Vergleiche gesteuert, so dauert das Sortiervverfahren mindestens so lange wie die Anzahl der hierfür erforderlichen Vergleiche.

Eine Folge von Vergleichen bildet einen binären Baum, der aus den Knoten und Kanten

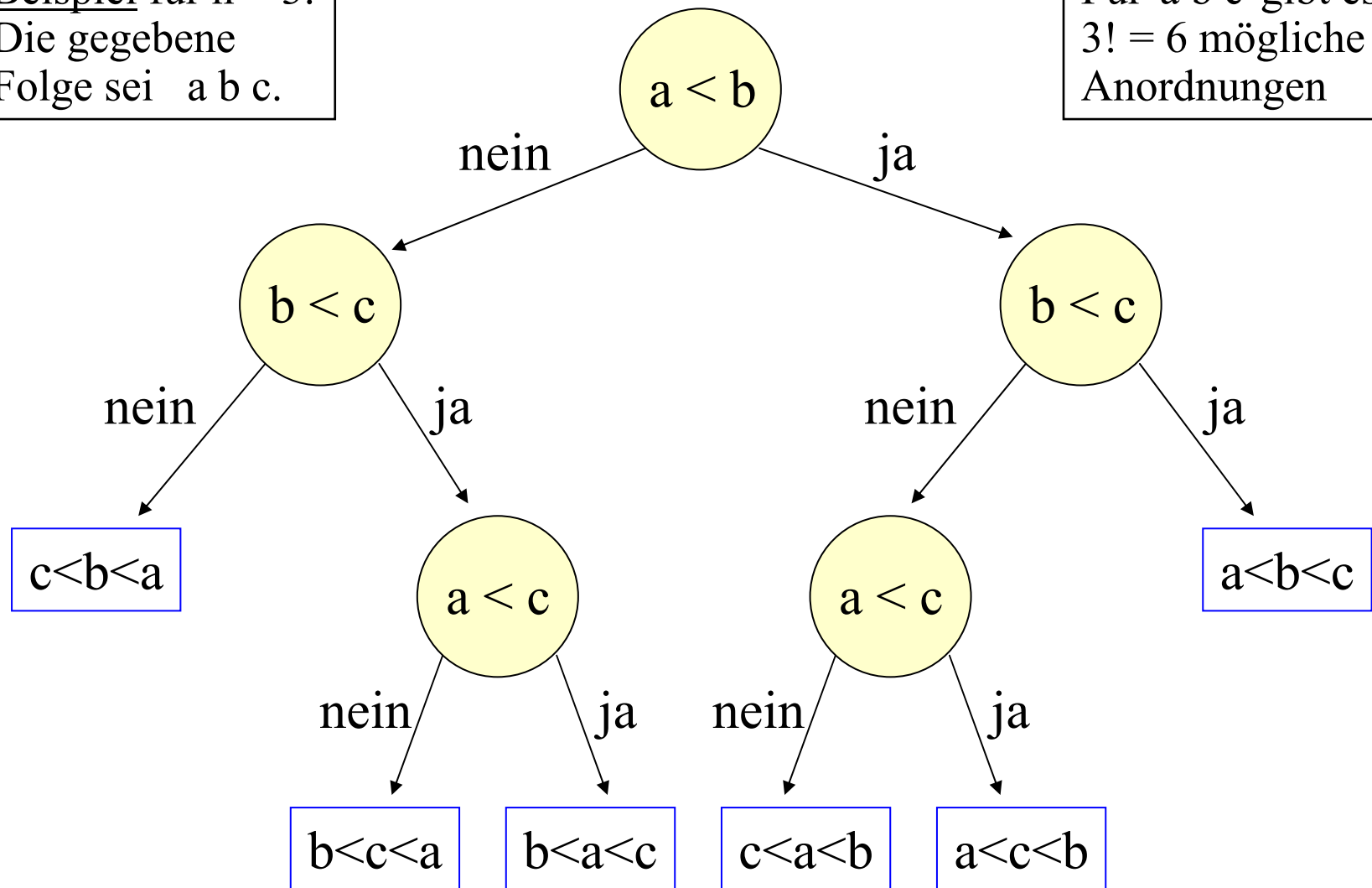


aufgebaut ist.



Beispiel für $n = 3$.
Die gegebene
Folge sei $a b c$.

Für $a b c$ gibt es
 $3! = 6$ mögliche
Anordnungen



Die sortierte Folge ist jeweils blau umrandet als Blatt angegeben.

Hat man alle Informationen zum Sortieren gewonnen, dann stellen genau die null-Zeiger in diesem Baum (= alle blau umrandeten Aussagen in obigem Beispiel) die Permutationen dar, die zur Sortierung gehören. Da es $n!$ Permutationen der Ordnung n gibt, muss der "Baum der Vergleiche" daher mindestens $n!$ null-Zeiger besitzen.

Ein binärer Baum mit $m-1$ Knoten besitzt genau m null-Zeiger. Also muss der Baum der Vergleiche

mindestens $n!-1$

Knoten besitzen.

Die Länge des längsten Weges, also die Tiefe dieses Baums gibt die Zahl der erforderlichen Vergleiche im worst case an. Die Tiefe eines binären Baums mit k Knoten ist aber mindestens $\log(k+1)$, siehe Folgerung 8.2.12. Daher gilt:

Satz 10.1.11: Ein Sortiervverfahren, das ausschließlich auf Vergleichen zweier Elemente beruht, benötigt im worst case mindestens $\log(n!) \approx n \cdot \log(n) - 1,4404 \cdot n + O(\log(n))$ Schritte.

Hinweis: Nach der *Stirlingschen Formel* gibt es zu jedem n ein d mit $0 < d < 1$, so dass gilt (mit $e = 2,718281828459\dots$):

$$n! = \left[\frac{n}{e} \right]^n \sqrt{2\pi n} e^{\frac{1}{12}d}$$

Durch Logarithmieren erhält man hieraus:

$$\log(n!) \approx n \cdot \log(n) - n \cdot \log(e) \approx n \cdot \log(n) - 1,4404 \cdot n$$

und von $2\pi n$ verbleibt noch ein additiver Anteil $O(\log(n))$.

10.1.12 Überblick über die üblichen Sortiermethoden:

Austauschen:

- a. Benachbartes Austauschen (Bubble sort, Shaker sort)
- b. Shellsort
- c. Quicksort

Einfügen:

- a. Einfügen in Listen (Insertion sort)
- b. Baumsortieren (mit binären Bäumen, AVL-Bäumen, ...)
- c. Fachverteilen (und radix exchange)

Aussuchen / Auswählen:

- a. Minimumsuche (minimum sort)
- b. Heapsort (normal, bottom up, ultimativ)

Mischen:

Merge sort und diverse Varianten

Streuen und Sammeln (bucket sort)

Sortiermethoden	Zeitaufwand	zusätzl. Platz
Austauschen a. Benachb. Austauschen b. Shellsort c. Quicksort (im Mittel !)	$\frac{1}{2} \cdot n^2$ $O(n^{\frac{3}{2}})$ $1,3863 \cdot n \cdot \log(n)$	konstant $\leq \log(n)$ $2 \log(n)$
Einfügen a. Einfügen in Listen b. Baumsortieren (AVL-Bäume) c. Fachverteilen (im Mittel)	$\frac{1}{2} \cdot n^2$ $\leq 1,4404 \cdot n \cdot \log(n)$ $O(n \cdot \log(n))$	konstant $O(n)$ $O(n)$
Aussuchen / Auswählen a. Minimumsuche b. Heapsort c. Bottom-up Heapsort	$\frac{1}{2} \cdot n^2$ $\approx 2n \cdot \log(n)$ $n \cdot \log(n) + O(n)$	konstant konstant konstant
Mischen Verschmelzen (Merge sort)	$O(n \cdot \log(n))$	n
Streuen und Sammeln	$O(n)$	$O(n)$

10.2 Sortieren durch Austauschen

Die beiden wichtigsten Vertreter **Bubble Sort** und **Quicksort** wurden bereits vorgestellt; sie sind auf den nächsten Folien nochmals als Programm ausformuliert.

Aufwandsabschätzung für Bubble Sort:

Platzbedarf: konstant (3 zusätzliche Variablen)

Zeitbedarf: worst case: $\frac{1}{2} \cdot n \cdot (n-1)$ Vergleiche,
best case: n , falls das Feld bereits sortiert ist,
im Mittel sind $\frac{1}{4} \cdot n \cdot (n-1)$ Vergleiche zu erwarten (**wirklich?**).

Man kann das Feld abwechselnd aufwärts und abwärts durchlaufen (dies nennt man **Shaker Sort**). Diese Variante bringt aber in der Praxis keine Verbesserung des Laufzeitverhaltens.

10.2.1 Bubble Sort für ein Integer-Feld A mit dem Indextyp 1..n
(das Feld A sei vom Feld-Typ Vektor):

procedure BubbleSort (A: in out Vektor) is

Weiter: Boolean := True; H: Integer;

begin

while Weiter loop

Weiter := False;

for i in 1..n-1 loop

if A(i) > A(i+1) then Weiter := True;

H := A(i); A(i) := A(i+1); A(i+1) := H;

end if;

end loop;

end loop;

end BubbleSort;

Beachte: Durch die Variable
"Weiter" wird Bubble Sort
ordnungsverträglich.

Oft programmiert man Buble Sort auch "abwärts", also:

for i in revers 1..n-1 loop ...

10.2.2 Quicksort: Aus 7.3.4 übernehmen wir mit den Datentypen *type Index is 1..n; ... A: array (Index) of Integer; ...* das Programm:

```
procedure Quicksort(L, R: Index) is      -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;              -- p wird das Pivot-Element
begin
  if L < R then
    i := L; j := R; p := A((L+R)/2);      -- man kann p auch anders wählen
    while i <= j loop                    -- die Indizes i und j laufen aufeinander zu
      while A(i) < p loop i := i+1; end loop;
      while A(j) > p loop j := j-1; end loop;
      if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
        i := i+1; j := j-1; end if;
      end loop;                          -- auch bei Gleichheit A(i)=p oder A(j)=p vertauschen!
      if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
      else Quicksort(i, R); Quicksort(L, j); end if; -- Vorsicht; siehe unten!
    end if;
  end Quicksort;

... Quicksort(1,n); ...                  -- Aufruf des Sortierverfahrens
```

10.2.3 Platzbedarf von Quicksort

In 7.3.4 wird erläutert, warum mit einem hohen Platzbedarf bei der rekursiven Formulierung zu rechnen ist. Man muss das Quicksort-Programm so abändern, dass man den Stack für die Rekursion selbst verwaltet und nutzlos gewordene Information sofort entfernt und nicht mehr im Stack stehen lässt. Es ist klar, dass man auf diese Weise die Tiefe des Stacks auf $\log(n)$ Paare beschränken kann.

Aufgabe: Versuchen Sie, eine Lösung für dieses Problem anzugeben, d.h., Sie sollen die Prozedur Quicksort so abwandeln, dass die rekursive Tiefe des Aufruf-Stacks durch $2 \cdot \log(n)$ beschränkt bleibt.

(Hinweis: Eine Lösung finden Sie im Buch Ottmann/Widmaier.)

10.2.4 Zeitbedarf von Quicksort (siehe 8.2.16):

Zeitbedarf: worst case: $\approx \frac{1}{2} \cdot n^2$ Vergleiche, wenn das Pivot-Element stets das kleinste oder das größte Element des Teilfelds ist.

best case: $n \cdot \log(n)$, wenn das Pivot-Element stets das mittelste der Elemente des Teilfelds ("Median") ist.

average case: Es ist mit $1,3863 \cdot n \cdot \log(n) - 1,8456 \cdot n$ Vergleichen im Mittel zu rechnen. Begründung:

Man kann das Aufteilen des Feldes in zwei Teilfelder mit dem Aufbau eines binären Suchbaums vergleichen, wobei das Pivot-Element in die Wurzel kommt und aus den beiden Teilfeldern rekursiv der linke und rechte Unterbaum aufgebaut werden.

Quicksort verhält sich daher im Mittel genau wie das Baum-sortieren mit binären Suchbäumen (Satz 8.2.15). Die formalen Berechnungen liefern das gleiche Ergebnis, siehe Lehrbücher.

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Für den Zeitbedarf ist die "gute Wahl" des Pivot-Elements p entscheidend. Gebräuchlich ist folgende Variante: Statt das Element p irgendwie fest zu wählen (z.B.: $p := A((L+R)/2)$ oder $p := A(L)$ oder ...) nimmt man das mittlere von drei Elementen, z.B. von $A(L)$, $A(R)$ und $A((L+R)/2)$.

Mit dieser "Mittelwert aus 3"-Variante erhält man einen deutlich besseren mittleren Zeitbedarf, nämlich im Mittel höchstens

$$1.188 \cdot n \cdot \log(n) - 2,255 \cdot n \text{ Vergleiche.}$$

Diese Variante wird daher gern in der Praxis verwendet.

Aufgabe: Erweitern Sie unsere Prozedur um diese Variante. Machen Sie Messungen mit zufällig erzeugten Daten und verifizieren Sie hiermit den Laufzeitgewinn, der ab einem gewissen n eintritt. (Bestimmen Sie dieses n experimentell.)

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Eine andere Variante besteht darin, die Zerlegung in drei Teilfelder vorzunehmen (sog. "Dreiwege-Split"):

Alle Elemente in $A(L..j)$ sind echt kleiner als p ,
alle Elemente in $A(i..R)$ sind echt größer als p und
alle Elemente in $A(j+1..i-1)$ sind gleich p , wobei dieser Bereich nie leer ist. Da man ihn nicht bei der Rekursion berücksichtigen muss, verringert sich die Rekursionstiefe und damit die Laufzeit. Treten in einer Folge viele Schlüssel mehrfach auf, so lässt sich hierdurch das Sortieren beschleunigen.

Aufgabe: Erweitern Sie unsere Prozedur auch um diese Variante. Die Schwierigkeit liegt darin, alle Schlüssel, die gleich p sind, ohne Zusatzaufwand in der Mitte des zu sortierenden Teilfelds zu platzieren. Auch hier sollten Sie dann Messungen mit zufällig erzeugten Daten vornehmen und prüfen, ab welchem Prozentsatz gleicher Schlüssel sich dieses Vorgehen lohnt.

Hinweise:

Historisch gesehen gibt es weitere Sortiervverfahren, die auf dem Austauschen beruhen. Am bekanntesten ist **Shellsort**, bei dem die Folge in äquidistante Teilfolgen unterteilt wird und als erste Phase in diesen eine Sortierung erfolgt (z.B. ein Bubble Sort Durchlauf). Die äquidistanten Abstände werden dann für eine zweite Phase verringert und diese Verringerung wird fortgesetzt, bis der Abstand gleich 1 ist, d.h. eine Sortierung der bis dahin entstandenen schon gut vorsortierten Folge stattfindet. Dieses Vorgehen setzt in den einzelnen Phasen ordnungsverträgliche Sortiervverfahren voraus.

Denken Sie sich selbst "schnelle" heuristische Verfahren aus. Der Fantasie sind hier kaum Grenzen gesetzt. Führen Sie aber auf jeden Fall Messungen durch (denn fast nie erfüllen sich die erhofften Beschleunigungen).

10.3 Sortieren durch Einfügen

Wenn die Elemente einer Folge durch Zeiger (sortierte Listen, Suchbäume) dargestellt werden, so wird man die einzelnen Elemente der Folge nacheinander in diese Struktur einfügen und dabei die Struktur stets wiederherstellen.

Einfachster Fall: Einfügen in eine sortierte Liste.

Eine Liste heißt sortiert, wenn für alle Elemente der Liste gilt: $p.\text{Inhalt} \leq p.\text{next}.\text{Inhalt}$. Hierbei ist p ein Zeiger, der auf das jeweilige Element der Liste verweist.

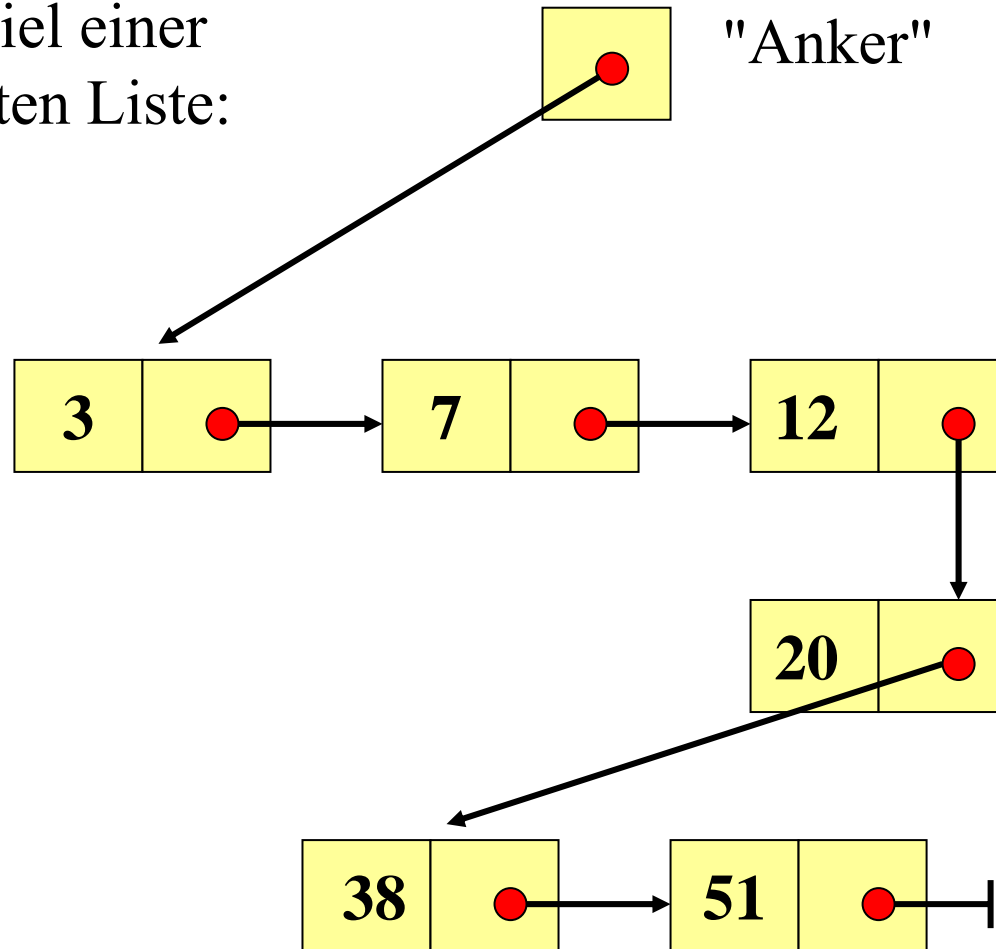
Erinnerung: Datenstruktur für eine Liste (vgl. z. B. 3.5.2.0):

type Zelle;

type Ref_Zelle is access Zelle;

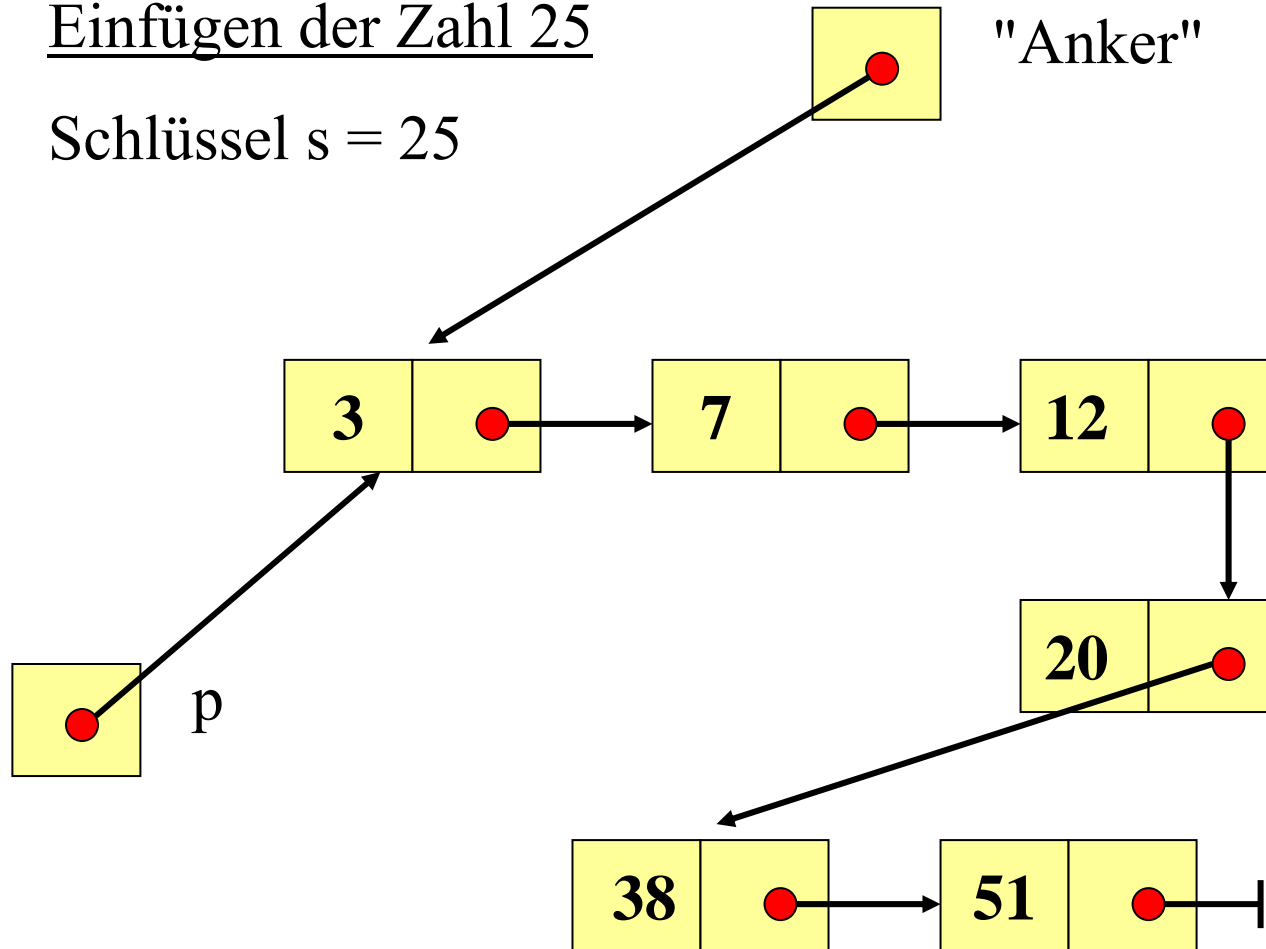
type Zelle is record Inhalt: Integer; Next: Ref_Zelle; end record;

Beispiel einer
sortierten Liste:



Einfügen der Zahl 25

Schlüssel $s = 25$

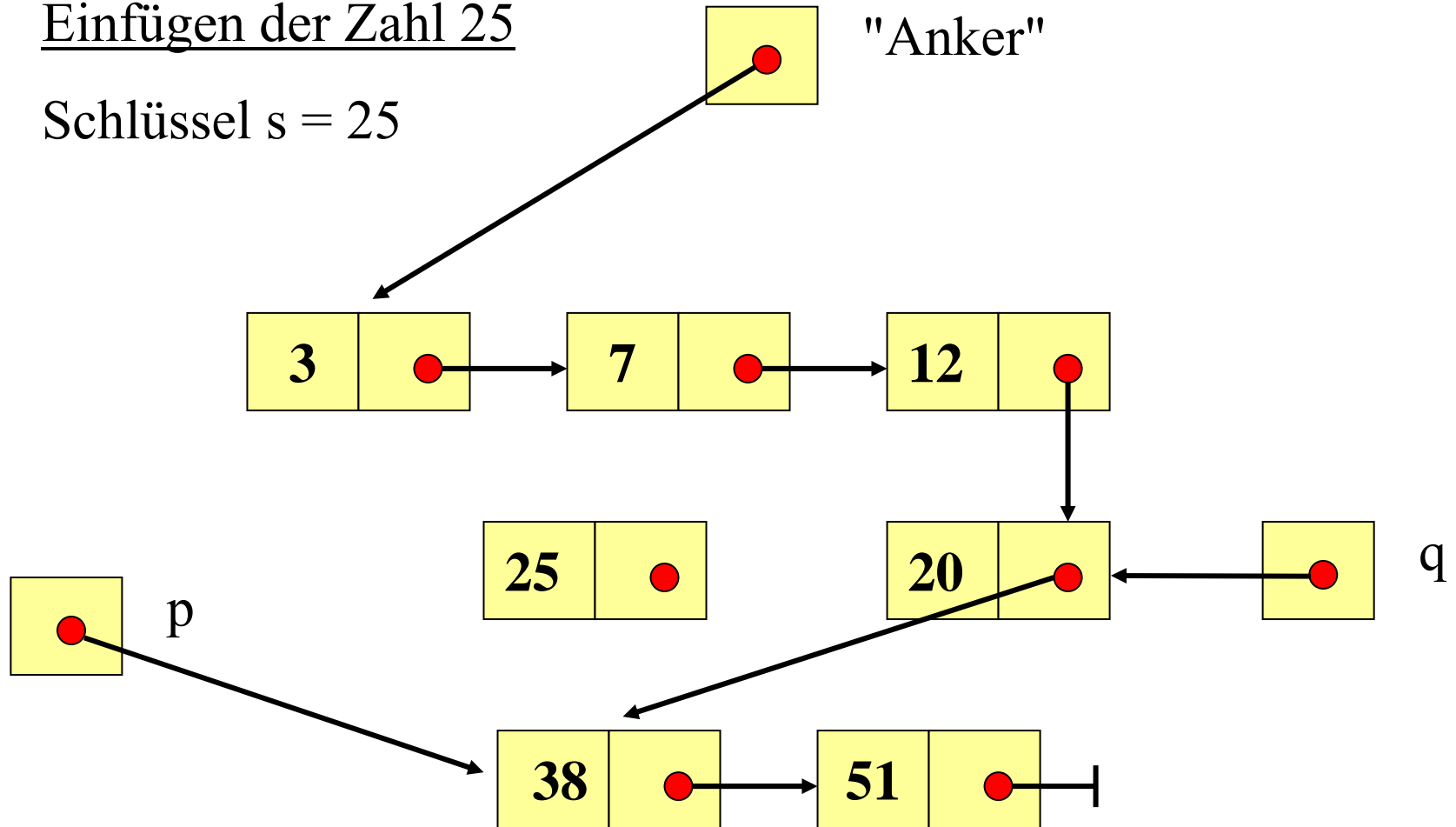


$p := \text{Anker};$

while $p \neq \text{null}$ and then $s > p.\text{Inhalt}$ loop $p := p.\text{Next};$ end loop;

Einfügen der Zahl 25

Schlüssel $s = 25$

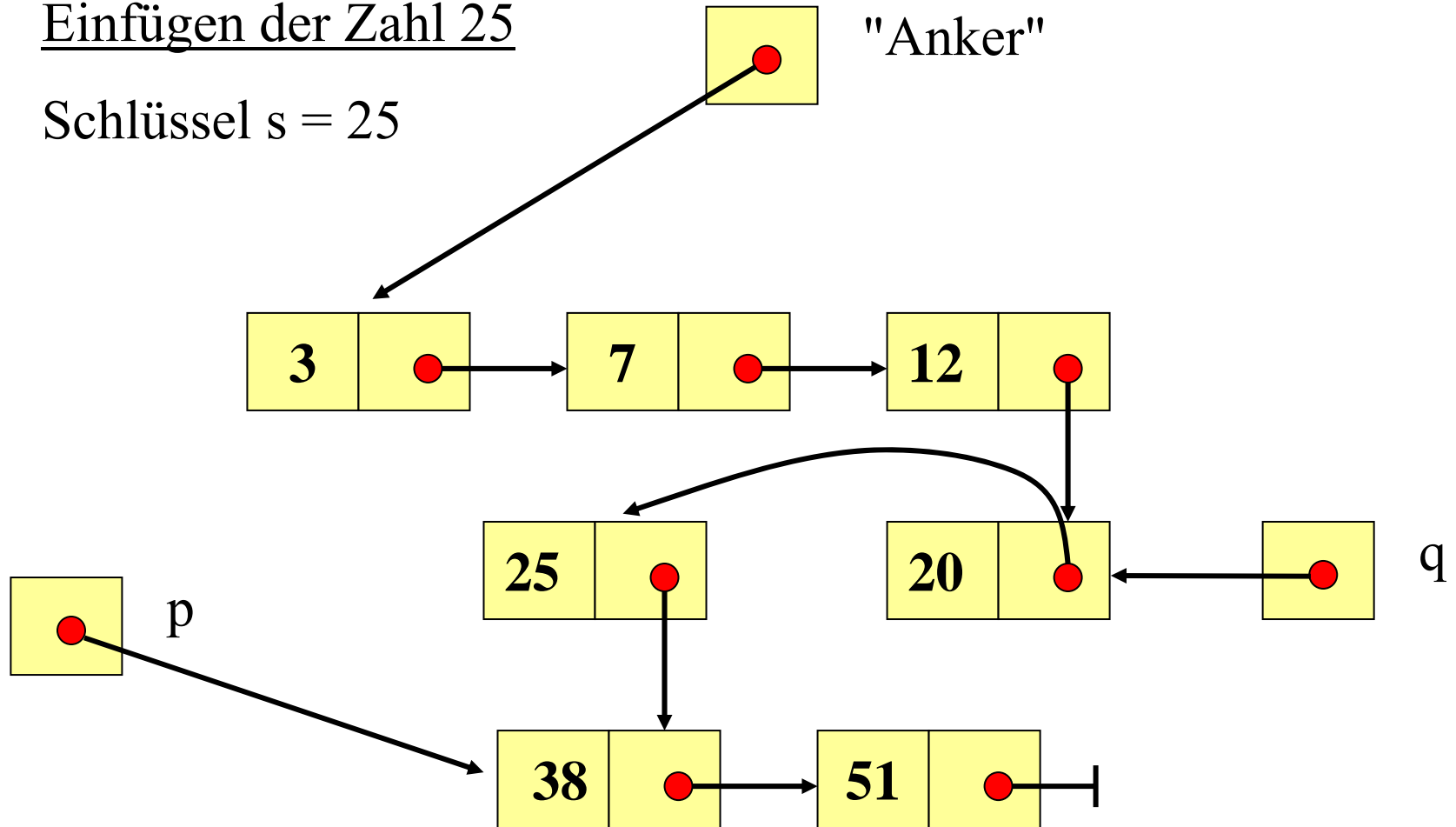


$p := \text{Anker};$

while $p \neq \text{null}$ and then $s > p.\text{Inhalt}$ loop $p := p.\text{Next};$ end loop;

Einfügen der Zahl 25

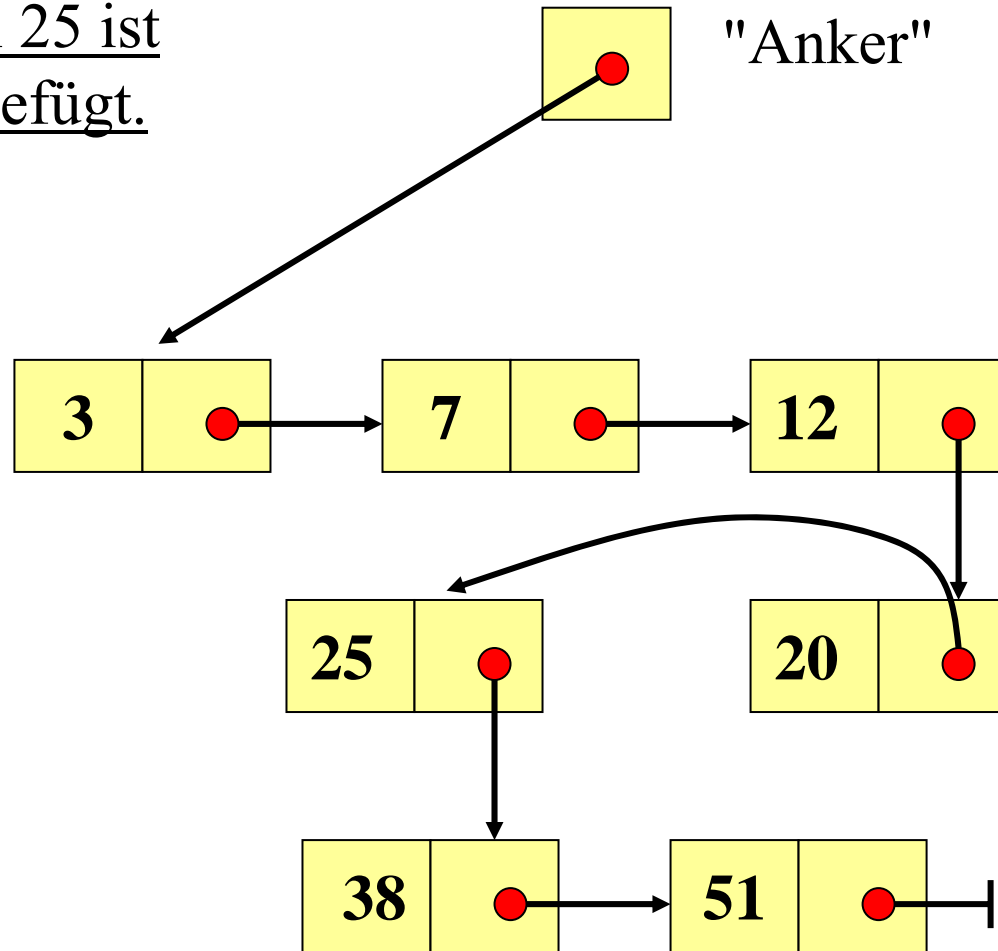
Schlüssel $s = 25$



$p := \text{Anker};$

while $p \neq \text{null}$ and then $s > p.\text{Inhalt}$ loop $p := p.\text{Next};$ end loop;

Die Zahl 25 ist
nun eingefügt.



Prozedur zum Einfügen eines Elementes in eine sortierte Liste:

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is  
  p, q: Ref_Zelle;  
begin  p := Anker; q := null;  
        while p /= null and then s > p.Inhalt loop  
          q := p; p := p.Next; end loop;  
        if q = null then  
          if p = null then Anker := new Zelle'(s, null);  
          else Anker := new Zelle'(s, Anker); end if;  
        else q.Next := new Zelle'(s, p); end if;  
end Einf;
```


Diese Prozedur kann man vereinfachen zu (bitte selbst nachprüfen!):

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is  
  p, q: Ref_Zelle;  
begin  p := Anker; q := new Zelle'(s, Anker);  
        while p /= null and then s > p.Inhalt loop  
          q := p; p := p.Next; end loop;  
        q.Next := new Zelle'(s, p);  
end Einf;
```

10.3.1 Sortieren von n Elementen durch Einfügen in eine Liste:

```
while not End_of_File loop Get(v); Einf(Anker, v); end loop;
```

Zahl der Vergleiche im Mittel und im schlechtesten Fall: $O(n^2)$.

10.3.2 *Sortieren durch Einfügen in einen Baum*: siehe 8.2.16.

Wenn man beliebige binäre Suchbäume verwendet, so können diese im schlechtesten Fall zu einer Liste entarten und daher beträgt im worst case die Zeitkomplexität $O(n^2)$.

Benutzt man aber anstelle eines beliebigen Suchbaums AVL-Bäume, so ist deren Höhe durch $1.4404 \cdot n \cdot \log(n)$ nach Satz 8.4.8 beschränkt. Daher ist die Anzahl der Vergleiche für das Sortieren mit Bäumen auch im schlechtesten Fall durch $1.4404 \cdot n \cdot \log(n) + O(n)$ beschränkt.

In der Praxis kann man bei der Verwendung von beliebigen Suchbäumen im Mittel mit $1.3863 \cdot n \cdot \log(n) + O(n)$, bei der Verwendung von AVL-Bäumen im Mittel mit $n \cdot \log(n) + O(n)$ rechnen (siehe Satz 8.2.15 und Hinweis nach Satz 8.4.8).

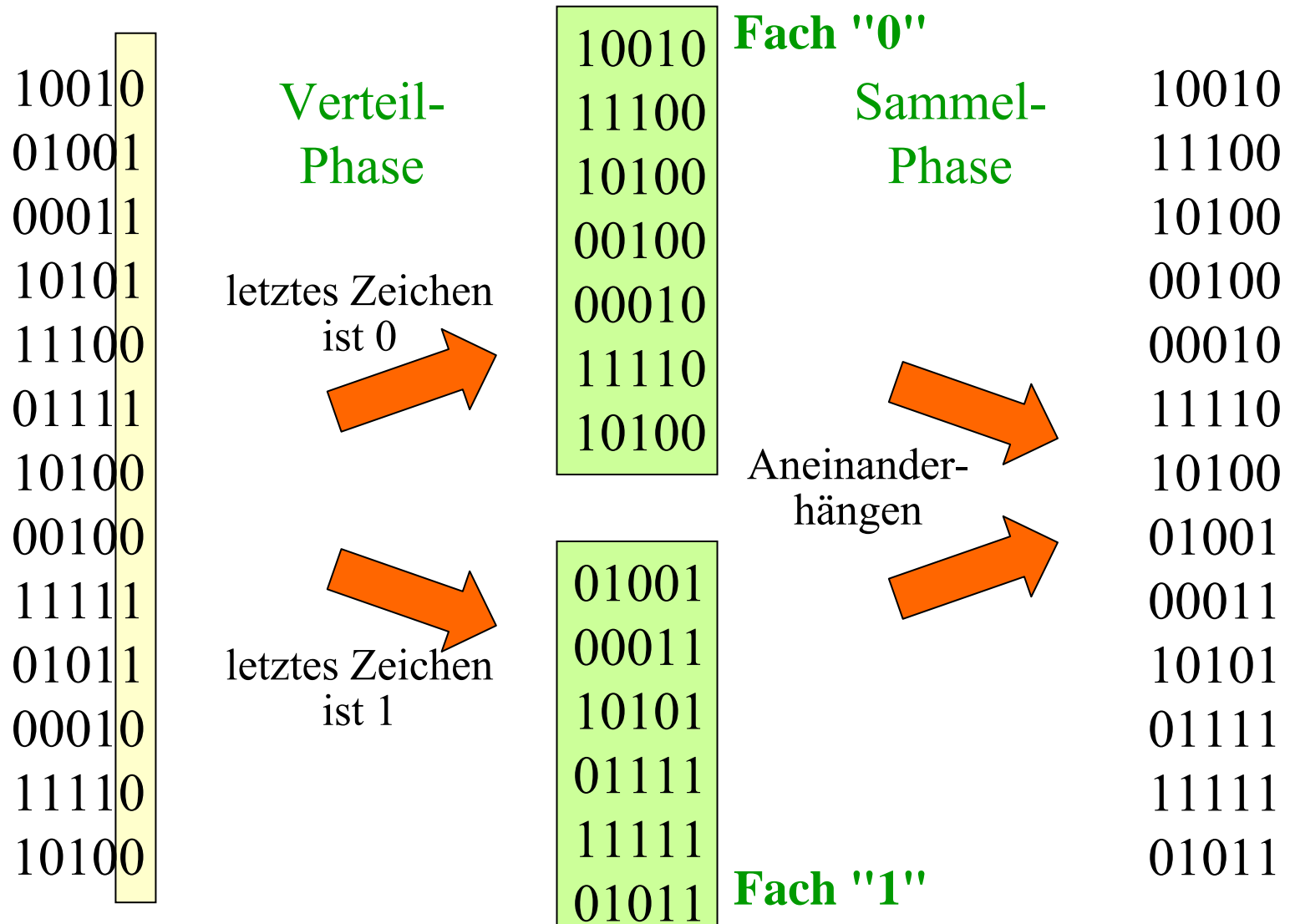
10.3.3 *Sortieren durch Fachverteilen*

Sind die Schlüssel Wörter über einem endlichen Alphabet $A = \{a_1, a_2, \dots, a_s\}$, kann man die zu sortierenden Elemente zunächst bzgl. des letzten Zeichens an s Listen anfügen. Diese Listen hängt man aneinander und fügt nun alle Elemente bzgl. des vorletzten Zeichens an s Listen an usw. Liegt die Länge jedes Schlüssels in der Größenordnung von $\log(n)$, dann ergibt sich ein $O(n \cdot \log(n))$ -Sortiervverfahren.

Das Anhängen an die s Listen entspricht dem Ablegen in s verschiedene Fächer, weshalb dieses Verfahren als "Fachverteilen" bezeichnet wird.

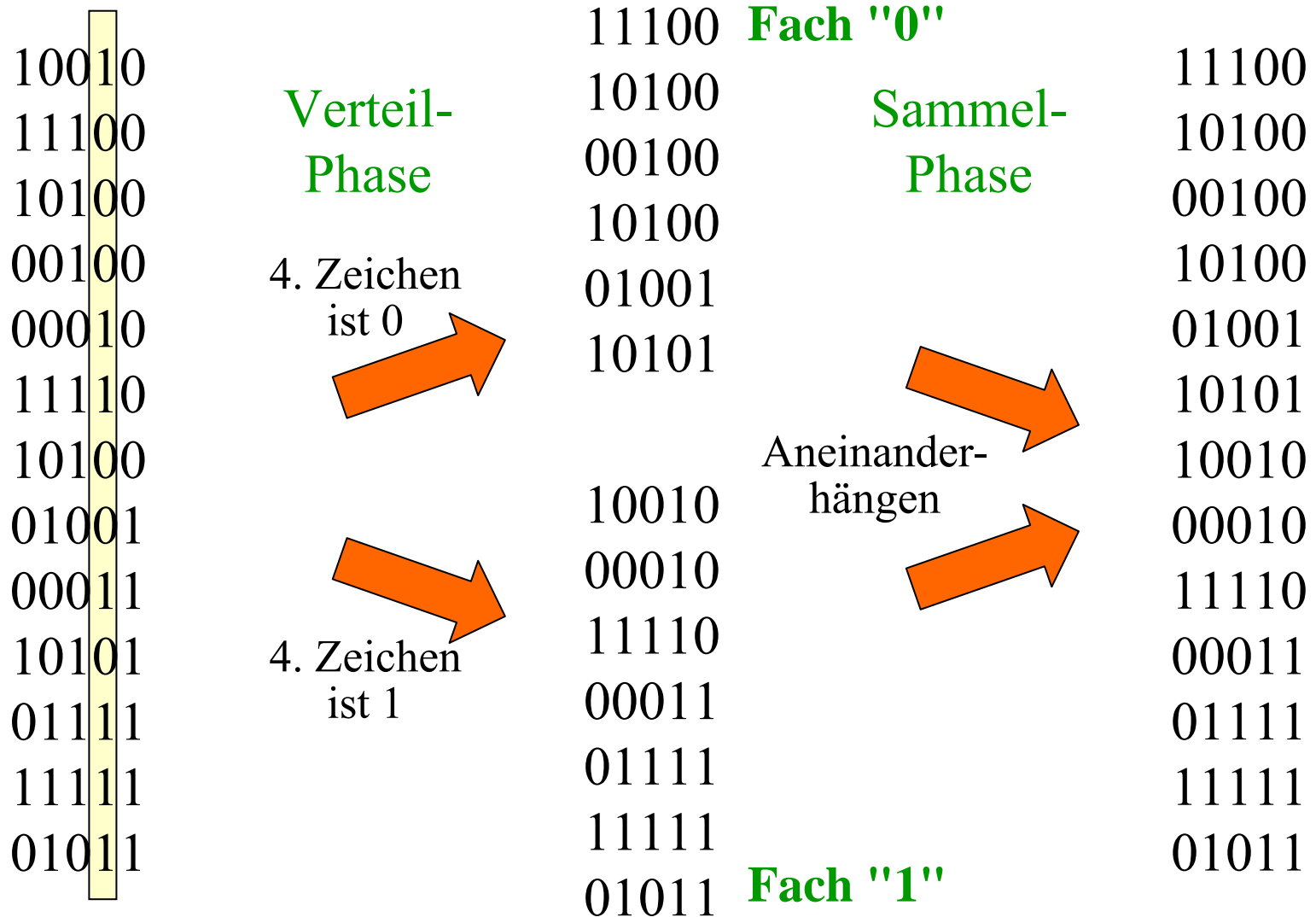
Wir erläutern das Verfahren nur an einem Beispiel mit $s=2$. Die Programmierung ist nicht schwierig.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



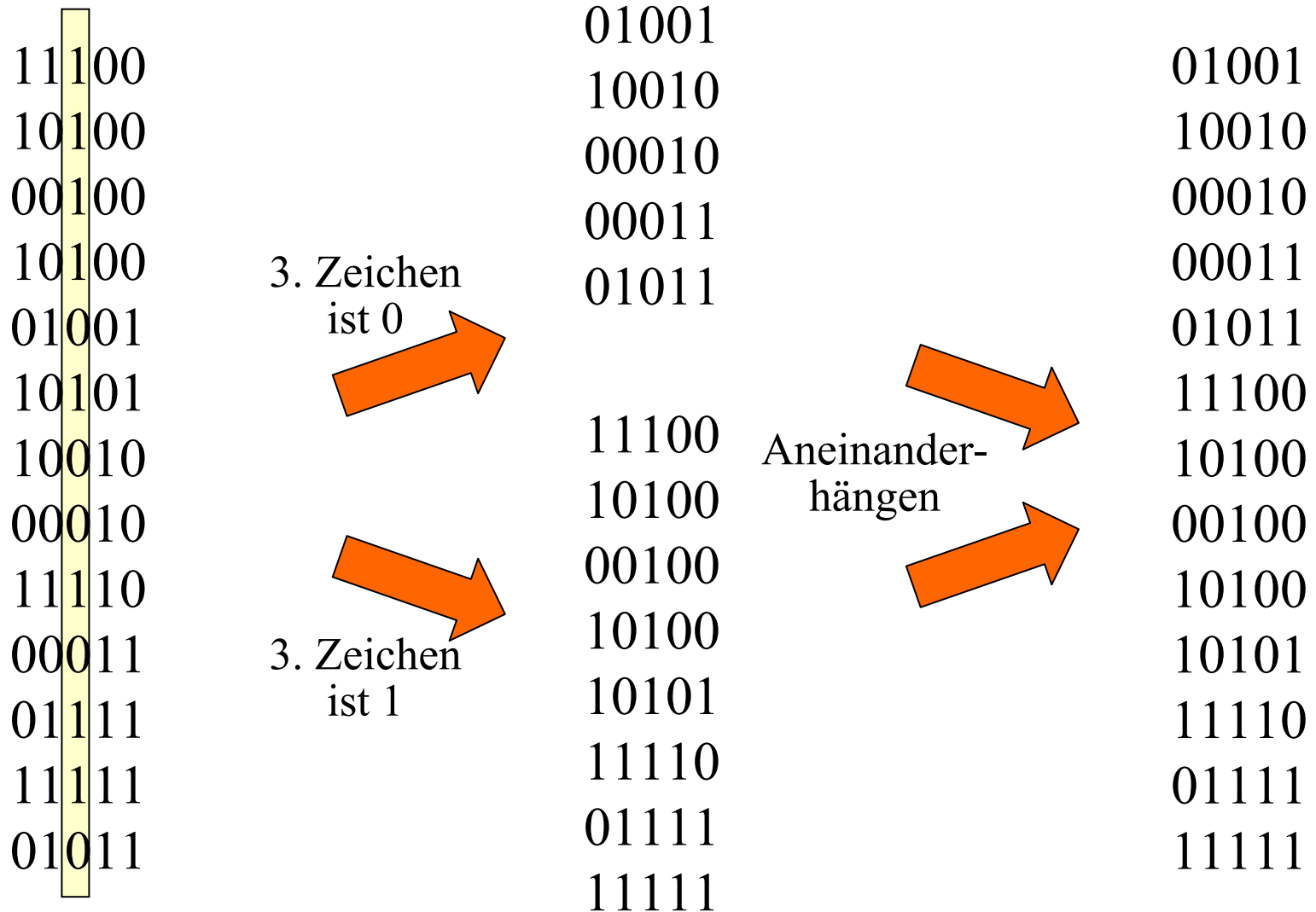
Nun iterativ weiter durch alle Stellen der Schlüssel.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5

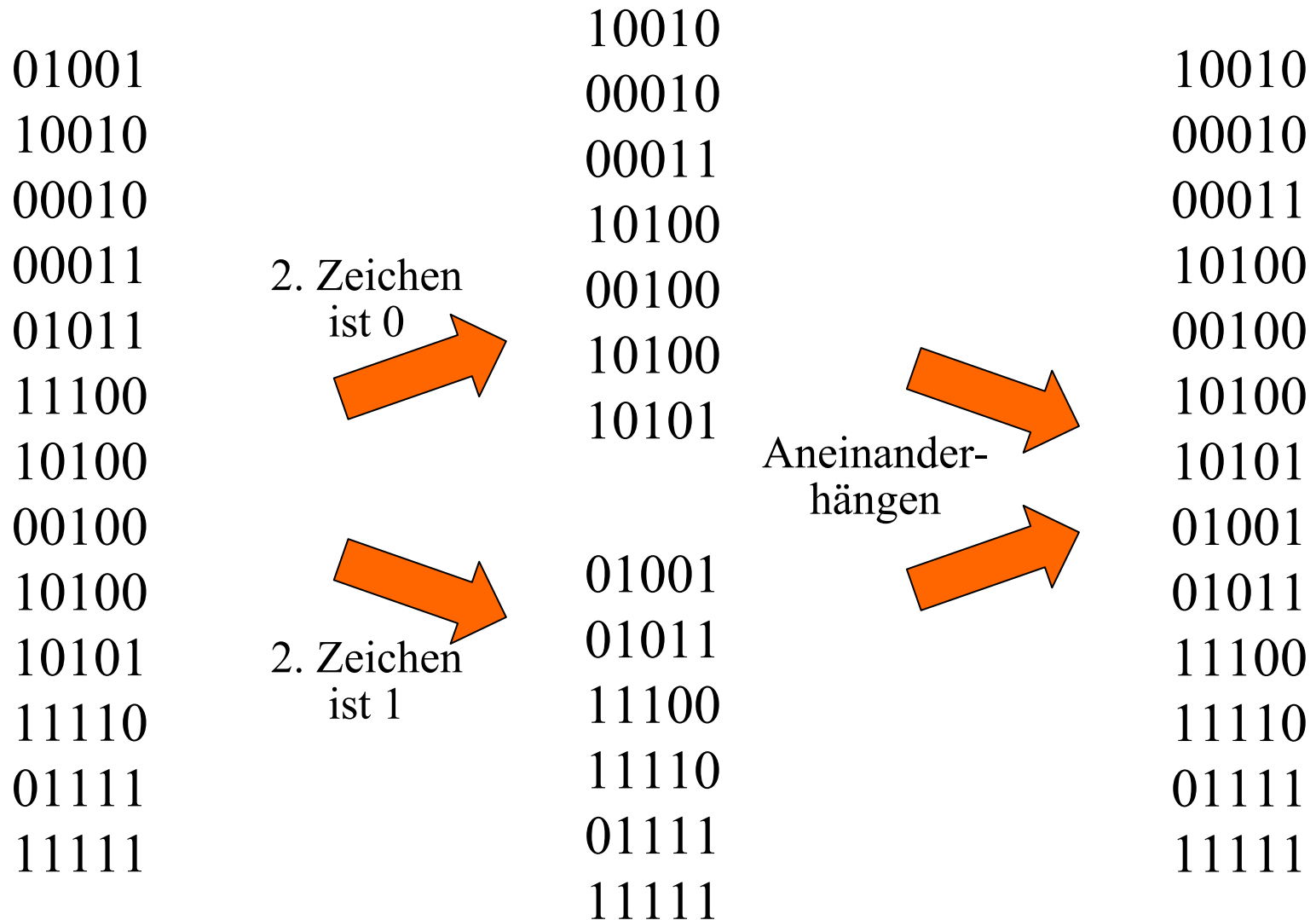


Nun iterativ weiter durch alle Stellen der Schlüssel.

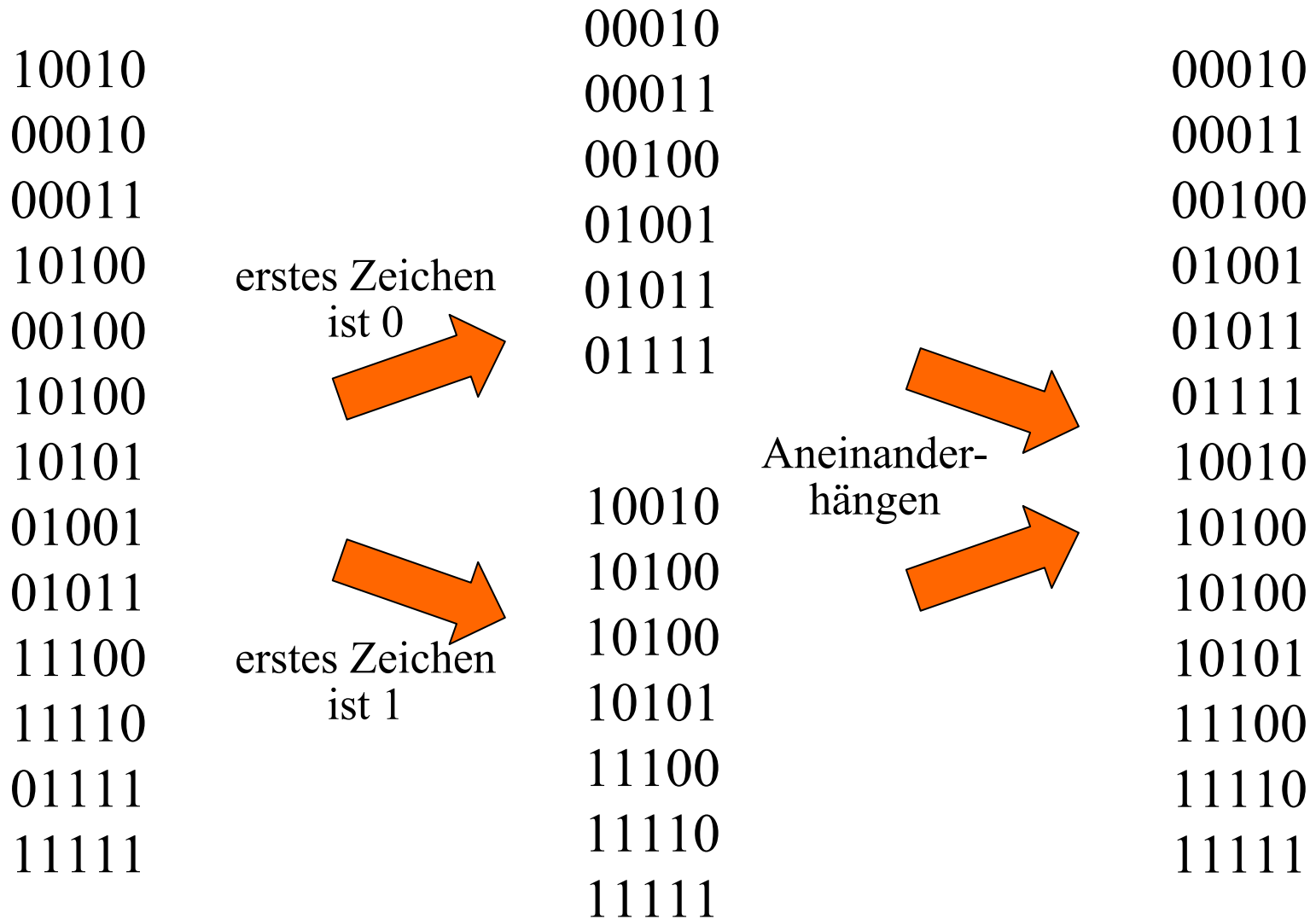
Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Ergebnis: Sortierte Folge.

Man beachte: Die Sortierung jeder Phase muss stabil sein. Hat man s Zeichen (z.B. $s=26$ für das Alphabet oder $s=128$ für den ASCII-Zeichensatz), dann muss man s solche "Fächer" bereitstellen. In der Regel organisiert man die Fächer als Listen, an die man hinten (in einem Schritt) den jeweils nächsten gelesenen Schlüssel anhängt; danach werden die s Listen (in s Schritten) aneinandergehängt und die nächste Verteilphase kann beginnen.

Das Sortieren erfordert $s \cdot n$ Vergleiche. Es eignet sich besonders gut für das Sortieren von binärer Information (z.B. in der Systemprogrammierung) und in allen Fällen, in denen $s \leq \log(n)$ ist. Nachteilig ist, dass man in der Regel das Doppelte an Speicherplatz benötigt. Man kann aber auch einen Austausch wie bei Quicksort vornehmen, so dass die 0-en oben und die 1-en unten zu stehen kommen (Vorsicht wegen der erforderlichen Stabilität!); im Falle $s > 2$ bietet sich auch ein bucket sort an, siehe 10.6.

10.4 Sortieren durch Aussuchen/Auswählen

Vorgehen:

Wähle das kleinste Element aus, stelle es an die erste Stelle und mache genauso mit den restlichen Elementen weiter.

10.4.1 Sortieren durch "**Minimum sortieren**":

for i in 1..n-1 loop

 min := A(i); pos := i;

 -- finde das kleinste Element von A(i) bis A(n)

for j in i+1..n loop

if A(j) < min then min:=A(j); pos := j; end if;

end loop;

 -- das kleinste Element steht an Position pos

 A(pos) := A(i); A(i) := min;

 -- nun steht das kleinste Element an Position i

end loop;

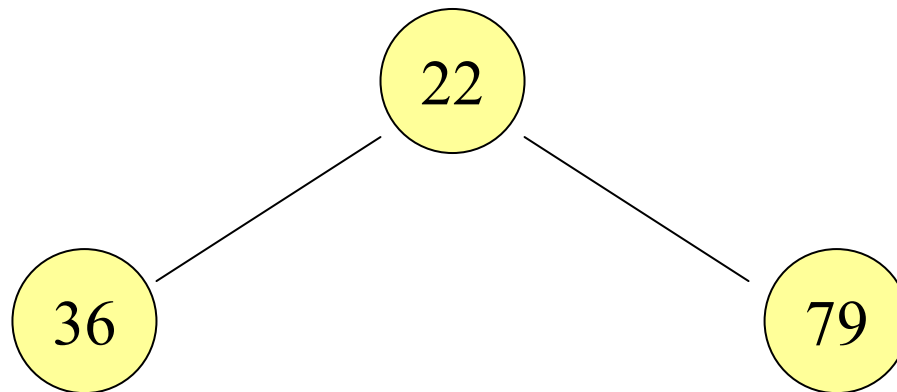
Zahl der Vergleiche stets $\frac{1}{2} \cdot n \cdot (n-1)$ Schritte: $\Theta(n^2)$

Platzaufwand 4 zusätzliche Speicherplätze: $O(1)$

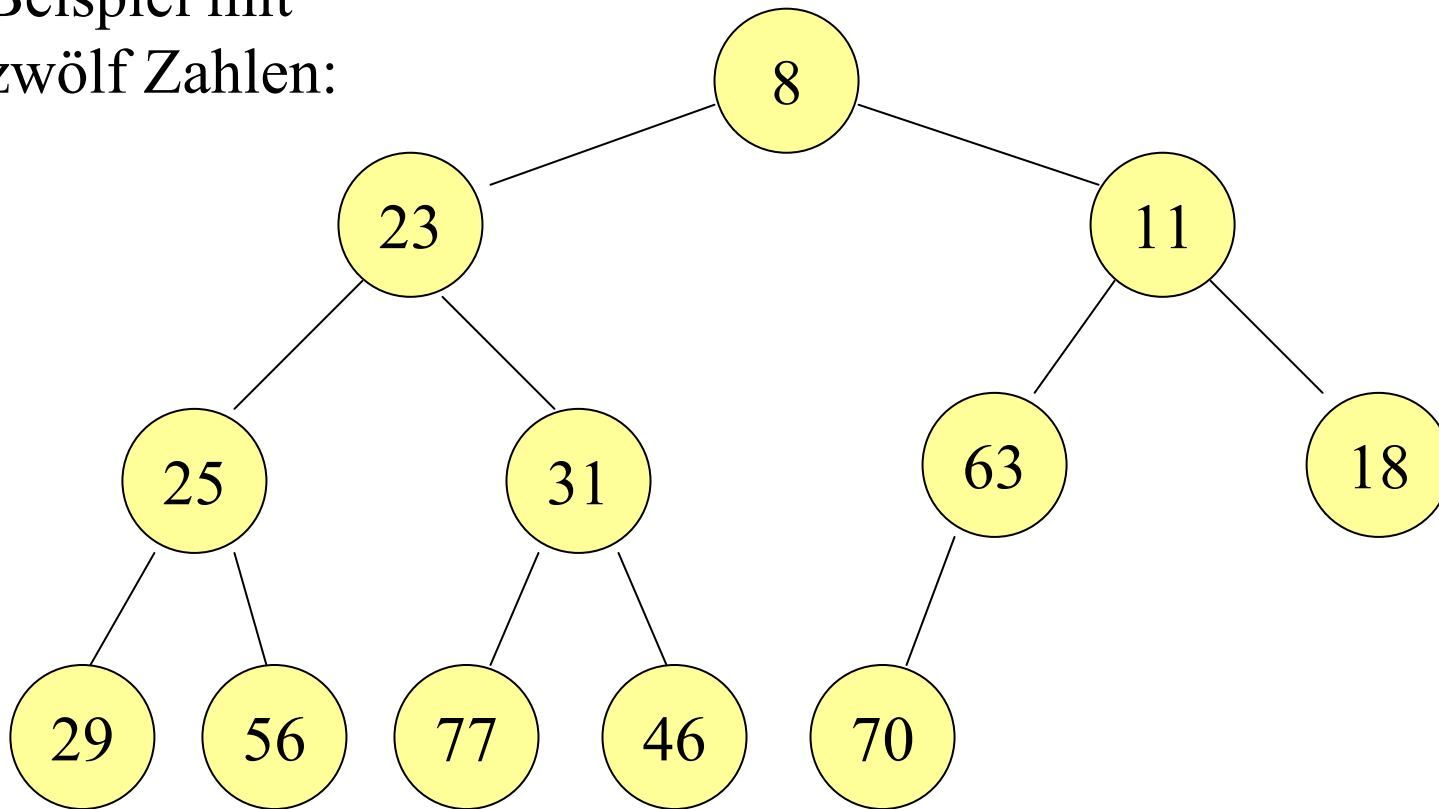
Überlegung:

Könnte man die Information "Minimum sein" besser anordnen?

Ja, in einem binären Baum. Betrachte einen Knoten mit zwei Nachfolgern. Schreibe in den Elternknoten das Minimum der drei Knoten.



Beispiel mit
zwölf Zahlen:



Als Feld levelweise aufgeschrieben:

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Besonderheit dieses Baums: Auf jedem Pfad von der Wurzel zu einem Blatt sind die Elemente aufsteigend geordnet.

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Die Bedingung "Der Inhalt eines Knotens ist nicht größer als der Inhalt jedes Nachfolgeknotens" lässt sich präzisieren durch $A(i) \leq A(2i)$ und $A(i) \leq A(2i+1)$. Folgen oder Felder mit dieser Eigenschaft nennen wir "Heap" (meist übersetzt mit "Haufen", es handelt sich aber um gut geordnete Haufen; sie haben nichts mit der Halde zu tun, die die mit new eingeführten Daten aufnimmt und die im Englischen ebenfalls "heap" heißt).

Definition 10.4.2: Heap-Eigenschaft

Eine Folge oder ein array $A(1), A(2), A(3), \dots, A(n)$ heißt ein (**aufsteigender**) **Heap**, wenn für jedes i gilt:

$$A(i) \leq A(2i) \text{ und } A(i) \leq A(2i+1),$$

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen $2i$ bzw. $2i+1$ nicht größer als n sind.

Eine Folge oder ein array $A(1), A(2), A(3), \dots, A(n)$ heißt ein **absteigender Heap**, wenn für jedes i gilt:

$$A(i) \geq A(2i) \text{ und } A(i) \geq A(2i+1),$$

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen $2i$ bzw. $2i+1$ nicht größer als n sind.

10.4.3 Heapsort:

Gegeben sei ein Feld $A(1), A(2), A(3), \dots, A(n)$ mit Elementen aus einer geordneten Menge.

1. Wandle dieses Feld in einen absteigenden Heap um, so dass anschließend gilt: $A(i) \geq A(2i)$ und $A(i) \geq A(2i+1)$ für alle i (sofern $2i \leq n$ bzw. $2i+1 \leq n$ ist).
2. Für j von n abwärts bis 2 wiederhole:
Vertausche $A(1)$ und $A(j)$.
(Nun verletzt $A(1)$ in der Regel die Heapeigenschaft.)
Wandle das Feld $A(1..j-1)$ ausgehend von der Wurzel so um, dass wieder ein absteigender Heap entsteht.

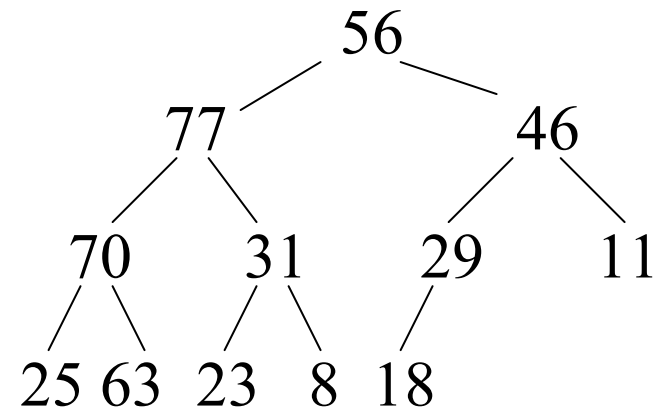
Im Folgenden beschreiben wir das Umwandeln in einen Heap (1.) und die Wiederherstellung der Heap-Eigenschaft (2.).

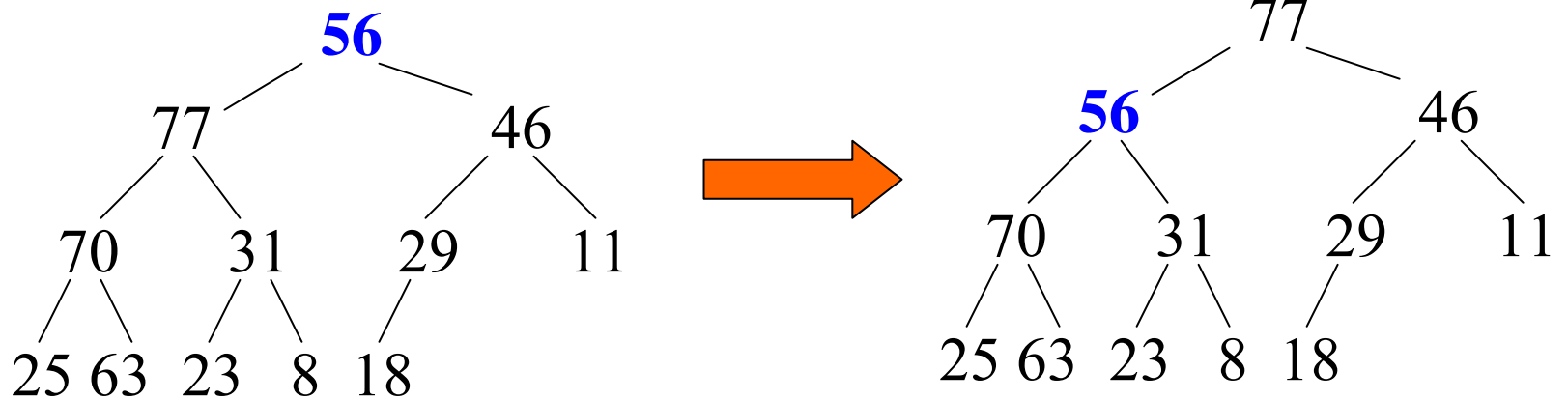
Die zentrale Prozedur ist die Herstellung der Heap-Eigenschaft in dem Teil des Feldes A, das mit dem Index "links" beginnt und mit dem Index "rechts" endet, unter der Annahme, dass höchstens beim Index "links" die Heap-Eigenschaft verletzt ist.

Hierfür vergleicht man den Inhalt des Elements A(links) mit den Inhalten der beiden Nachfolgeknoten und lässt gegebenenfalls den Inhalt von A(links) durch Vertauschen mit dem kleineren der beiden Nachfolger-Inhalten "**absinken**".

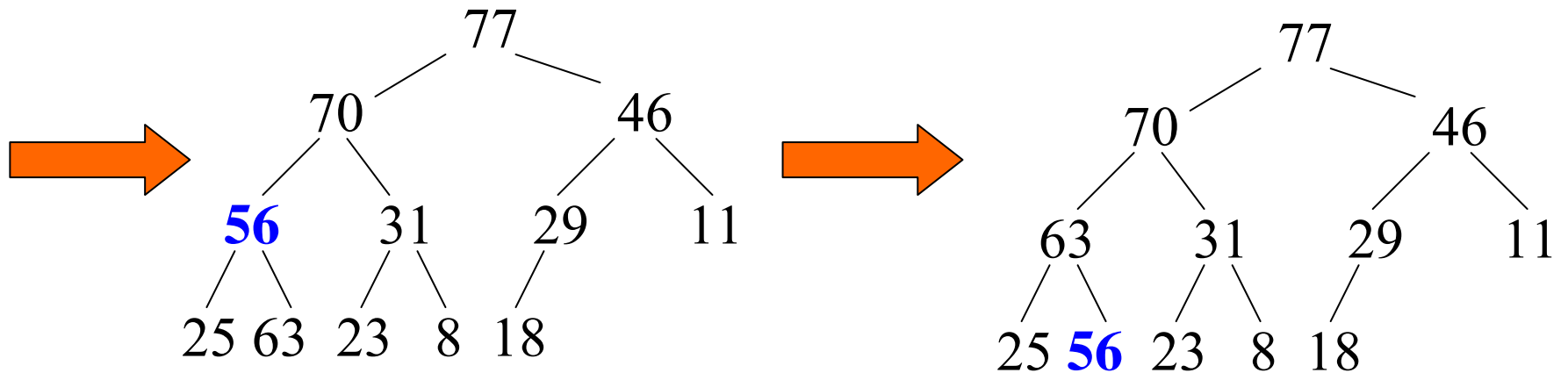
Betrachte ein Beispiel (mit links = 1 und rechts = 12) :

Die Heap-Eigenschaft ist hier nur bei "56" verletzt.





Vorgehen: Vergleiche 56 mit 77 und 46. Das Maximum ist 77, daher werden 77 und 56 vertauscht und bei 56 weitergemacht.



10.4.4: Prozedur für das Absinken

```
procedure sink (links, rechts: 1..n) is                                -- A und n sind global
i, j: Natural; weiter: Boolean:=true; v: <Elementtyp>;
begin v := A(links); i := links; j := i+i;
      while (j <= rechts) and weiter loop
        if j = rechts then
          if A(j) > v then A(i):=A(j); i:=j; end if;
          weiter:=false;
        elsif A(j) < A(j+1) then
          if v < A(j+1) then A(i) := A(j+1); i := j+1;
          else weiter:=false; end if;
        else   if v < A(j) then A(i) := A(j); i := j;
          else weiter:=false; end if;
        end if;
        j := i+i;
      end loop;
      A(i):=v;
end sink;
```

-- v wird erst am Ende explizit eingetragen.
-- i gibt am Ende die aktuelle Position an, an
-- der v einzufügen ist.
-- Im Inneren der Schleife werden bis zu **zwei**
-- **Vergleiche** zwischen Elementen durchgeführt.

10.4.5: Prozedur für Heapsort

```
procedure heapsort is                                     -- A und n sind global
  procedure sink ... begin .... end sink;                 -- siehe oben 10.4.4
  h: Natural; x: <Elementtyp>;
  begin
    h := n div 2;                                           -- wandle A in einen Heap um
    for k in reverse 1..h loop sink (k, n); end loop;
    for k in reverse 2..n loop
      x := A(1); A(1) := A(k); A(k) := x; -- vertausche A(1) und A(k)
      sink (1, k-1); end loop;                             -- Wurzel absinken lassen
  end heapsort;
```

Hinweis: Es lassen sich noch einige Umspeicherungen vermeiden, indem man das Wechselspiel zwischen v (in 'sink') und x optimiert. Dies ändert aber nichts an der Zahl der (Element-) Vergleiche.

Wie viele Vergleiche benötigt Heapsort?

1. Aufbau des Heaps (for k in reverse 1..h loop sink(k, n); end loop);

Für k von h bis h/2: maximal 2 Vergleiche

für k von h/2 bis h/4: maximal 4 Vergleiche

für k von h/4 bis h/8: maximal 6 Vergleiche

für k von $h/2^{i-1}$ bis $h/2^i$ maximal $2i$ Vergleiche ($i=1, 2, \dots, \log(n)$)

Aufsummieren ergibt **maximal $2 \cdot n$ Vergleiche** (beachte $h = n/2$):

$$2 \cdot h/2 + 4 \cdot h/4 + 6 \cdot h/8 + 8 \cdot h/16 + \dots + 2 \cdot \log(n) \cdot 1$$

$$= 2h \cdot (2 - 2 \cdot (\log(n)+1)/n) = 2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n \text{ Vergleiche.}$$

Der Aufbau des Heaps erfolgt also in linearer Zeit.

Dies beweist man genauso wie die entsprechende Formel $\sum_{j=1}^k j \cdot 2^{j-1}$
in Abschnitt 6.5.1. Es gilt:

$$1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + \dots + m/2^m = 2 - (m+1)/2^{(m-1)}.$$

2. Sortierphase (for k in reverse 2..n loop ... sink(1, k-1); end loop;))

Für k von n bis n/2: maximal $2 \cdot \log(n)$ Vergleich
für k von n/2 bis n/4: maximal $2 \cdot \log(n) - 1$ Vergleiche
für k von n/4 bis n/8: maximal $2 \cdot \log(n) - 2$ Vergleiche
für k von $n/2^{i-1}$ bis $n/2^i$ maximal $2 \cdot i$ Vergleiche ($i=1, 2, \dots, \log(n)$)

Aufsummieren ergibt **maximal $2 \cdot n \cdot \log(n)$ Vergleiche**: Sei $n > 1$:

$$\begin{aligned} & \log(n) \cdot n + (\log(n)-1) \cdot n/2 + (\log(n)-2) \cdot n/4 + (\log(n)-3) \cdot n/8 + \dots + 2 = \\ & 2 \cdot n \cdot (\log(n)/2 + \log(n)/4 + \dots + 1/2^{\log(n)} - 1/4 - 2/8 - 3/16 - \dots - (\log(n)-1)/2^{\log(n)}) = \\ & 2 \cdot n \cdot \log(n) \cdot (1 - 1/2^{\log(n)}) - n \cdot (1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + \dots + (\log(n)-1)/2^{\log(n)-1}) = \\ & 2 \cdot n \cdot \log(n) - 2 \cdot \log(n) - n \cdot (2 - \log(n)/2^{\log(n)-2}) = 2 \cdot n \cdot \log(n) - 2 \cdot n + 2 \cdot \log(n) \\ & < 2 \cdot n \cdot \log(n) \end{aligned}$$

(Wir benutzten hier erneut die Formel von der vorherigen Folie ganz unten.)

Insgesamt ergeben sich für Heapsort im worst case maximal
 $(2 \cdot n - 2 \cdot \log(n) - 2) + (2 \cdot n \cdot \log(n) - 2 \cdot n + 2 \cdot \log(n)) =$
 $2 \cdot n \cdot \log(n) - 2$ Vergleiche (für $n > 1$).

Im best case kann $n \cdot \log(n)$ erreicht werden, da durch spezielle Folgen das Absinken beschränkt werden kann. Der Mittelwert allerdings wird ebenfalls dicht bei $2 \cdot n \cdot \log(n)$ liegen, da Heapsort keine Vorsortierungen oder günstigen Konstellationen ausnutzt und da das in die Wurzel vertauschte Element klein ist, also meist weit nach unten im Baum absinkt. Daher folgt:

Satz 10.4.6:

Dieses normale Heapsort (aus dem Jahre 1962) benötigt im schlechtesten Fall höchstens $2 \cdot n \cdot \log(n)$ Vergleiche (für $n > 1$). (Im besten Fall kann man höchstens den Faktor 2 sparen. Der average case liegt recht nahe beim schlechtesten Fall.)

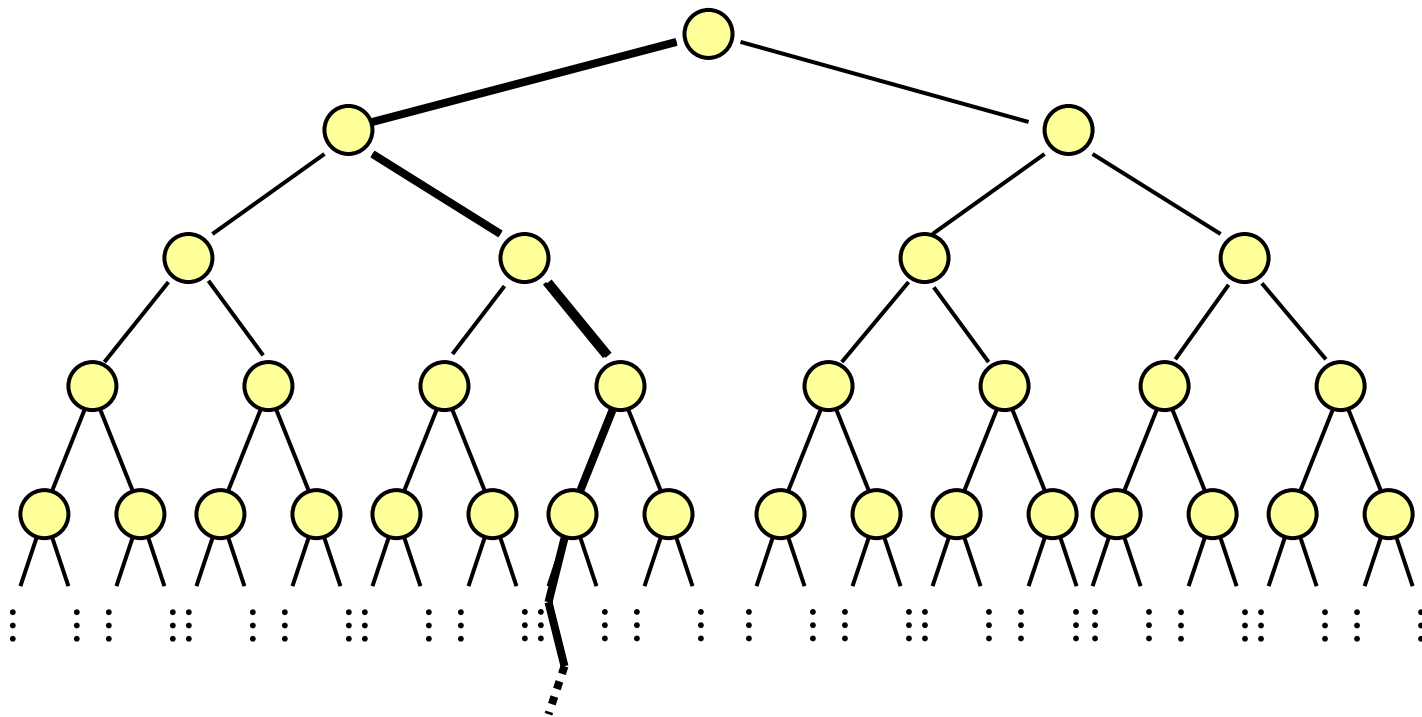
Geht es nicht doch noch besser?

Ja, man kann den Faktor 2 noch verkleinern (allerdings kann er nicht kleiner als "1" werden, wie in Satz 10.1.11 gezeigt wurde.)

Beobachtung: Beim eigentlichen Sortieren wird das letzte Element mit dem ersten vertauscht. Die Prozedur "sink" wird dieses letzte Element in der Regel sehr weit absenken, da es ja zu den leichten Elementen gehört hat, die sich ganz unten im Baum befinden. Man sollte daher das Element nicht von oben nach unten absenken, sondern es von unten nach oben (also "bottom-up") aufsteigen lassen. Hierzu muss man aber die Stelle, an der es einzufügen ist, kennen. Genau diese Stelle ermittelt man durch die Berechnung des "Einsinkpfads".

10.4.7 Einsinkpfad: Dies ist der Weg, den ein Element, das in die Wurzel gesetzt wurde, nehmen muss, damit die Heap-Eigenschaft wiederhergestellt wird (vgl. 10.4.4).

Dieser Pfad ist unabhängig vom einzusortierenden Element. Er endet in einem Blatt des Baumes. Es genügt, den Index dieses Blattes zu bestimmen.



Ermittlung des Einsinkpfads:

starte mit der Wurzel;

while noch nicht Blatt erreicht loop

 gehe zum Nachfolgeknoten mit dem größeren Inhalt;

end loop;

Bei der Darstellung mit Feldern muss man nur den Index j des letzten Elements auf dem Einsinkpfad kennen. Die anderen Knoten auf diesem Pfad besitzen die Indizes $j \div 2$, $(j \div 2) \div 2$, $((j \div 2) \div 2) \div 2$, ..., 1. (In binärer Darstellung muss man also nur die letzte Ziffer streichen.)

Die folgende Funktion berechnet den Index j des letzten Elements des Einsinkpfads, wobei man den Fall, dass der letzte Knoten keinen Geschwisterknoten besitzt, gesondert berücksichtigt ("if $m = \text{rechts}$ then ...").

Ermittlung des Einsinkpfads:

```
function einsinkpfad (rechts: 1..n) return 1..n is  
  j: 1..n := 1; m: Natural := 2;  
  begin  
    while m < rechts loop  
      if A(m) < A(m+1) then j := m+1; else j:= m; end if;  
      m := j+j;  
    end loop;  
    if m = rechts then j := rechts; end if;  
    return j;  
  end;
```

10.4.8 Bottom-up-Heapsort: (Carlsson 1987, Wegener 1993)

1. Ermittle den Index j des letzten Knotens auf dem Einsinkpfad.
2. Suche von j aus rückwärts entlang des Einsinkpfads die Stelle, wo das einzusortierende Element hingehört.
3. Füge es dort ein und schiebe alle darüber stehenden Elemente entlang des Einsinkpfads um eine Position in Richtung zur Wurzel.

Für die Programmierung benutzen wir die obige Funktion "einsinkpfad", die wir jedoch direkt in den Algorithmus integrieren. Weiterhin führen wir die Verschiebung von Punkt 3 bereits beim Berechnen von j durch, da man in der Regel den Einsinkpfad nur wenige Schritte zurücklaufen muss und hierdurch im Mittel ein doppeltes Durchlaufen vermieden wird.

```

procedure bottomupheapsort is                                -- A und n sind global
procedure sink ... begin .... end sink;                       -- wie früher
h, j, m: Natural; x: <Elementtyp>;
begin h := n div 2;                                           -- wandle A in einen Heap um
    for k in reverse 1..h loop sink (k, n); end loop;
    for k in reverse 2..n loop                                  -- x = A(k) einsinken lassen
        x := A(k); A(k) := A(1);                                -- rette A(1) nach A(k)
        j := 1; m:= 2;                                           -- Suche den Index j
        while m < k-1 loop
            if A(m) < A(m+1) then A(j) := A(m+1); j := m+1;
            else A(j) := A(m); j:= m; end if;
            m := j+j;
        end loop;
        if m = k-1 then A(j) := A(k-1); j := k-1; end if;

-- Nun ist der Index j (=Ende des Einsinkpfads) bekannt und alle Inhalte auf dem
-- Einsinkpfad sind um eine Position in Richtung der Wurzel verschoben worden.

```

-- Die Elemente des Einsinkpfads müssen nun zurückgeschoben werden,
-- solange die Stelle, an die x gehört, noch nicht erreicht ist.

```
while (j > 1) and then (A(j) < x) loop  
    i := j div 2; A(j) := A(i); j := i;  
end loop;
```

-- Die Stelle j, an die x gehört, ist nun erreicht.

```
    A(j) := x;  
    end loop k;  
end bottomupheapsort;
```

Wie viele Vergleiche benötigt Bottom-up-Heapsort?

1. Aufbau des Heaps: Genauso wie beim normalen Heapsort maximal $2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n$ Vergleiche.

2. Sortierphase

Hier benötigt man maximal für jedes k so viele Vergleiche, wie die doppelte Länge des Einsinkpfads ist, also rund $2 \cdot \log(k)$.

Dies führt zunächst nur auf genau die gleiche Abschätzung wie beim normalen Heapsort.

Aber: In den meisten Fällen wird man bereits nach etwas mehr als $\log(k)$ Vergleichen fertig sein.

Experimente bestätigen, dass der Faktor "2" vom normalen Heapsort im Mittel auf "1" sinkt; dies lässt sich auch beweisen. Man kann aber Folgen konstruieren, bei denen man nur auf den Faktor 1,5 kommt. Daher gilt im worst case nur:

Bottom-up-Heapsort benötigt im worst case maximal $1,5 \cdot n \cdot \log(n)$ Vergleiche. Dieser Fall tritt aber fast nie auf, so dass man in der Praxis von $n \cdot \log(n) + O(n)$ Vergleichen ausgehen kann. Carlsson konnte zeigen, dass im Mittel höchstens $n \cdot \log(n) + 0.67n$ Vergleiche benötigt werden.

10.4.9 Satz: Bottom-up-Heapsort benötigt
im schlimmsten Fall höchstens $1,5 \cdot n \cdot \log(n)$,
im Mittel höchstens $n \cdot \log(n) + 0.67 \cdot n$ Vergleiche.

Beachte: Heapsort und seine Varianten sind
garantierte $n \cdot \log(n)$ - Verfahren.

Hinweis: Es gibt weitere Varianten, z.B. von McDiarmid and Reed 1998 oder von Katajainen 1998. Ziel ist es, die untere theoretische Schranke von Satz 10.1.11 zu erreichen. Der Faktor 1 (statt 1,5) wurde mit dem "ultimativen" Heapsort erreicht, das aber (wegen eines zu hohen linearen Anteils) bisher noch ungeeignet ist (siehe Algorithmen-Vorlesung).

Diskussion dieses Ergebnisses:

Mit Bottomup-Heapsort haben wir ein Verfahren gefunden, das schneller als Quicksort sein müsste. Dies trifft vor allem dann zu, wenn im Algorithmus der Vergleich zwischen zwei Elementen, also die Abfragen " $A(m) < A(m+1)$ " und " $A(j) < x$ ", viel mehr Zeit kosten als alle anderen elementaren Anweisungen; denn wir haben ja nur die Anzahl dieser Vergleiche gezählt. Diese Annahme gilt sicher, wenn die Schlüssel besonders lang (z.B. lange Wörter) sind.

Sind die Schlüssel jedoch Zahlen vom Typ Integer, so dauert der Vergleich nicht länger als eine Zuweisung der Form " $A(j) := A(i)$ ". Vor allem die Zugriffe auf den Hauptspeicher kommen nun zum Tragen. Bei Quicksort erfordert das Umspeichern 4 Zugriffe, da zwei Werte ausgetauscht werden, bei Heapsort braucht man nur 2, da ein Wert verschoben wird. Allerdings werden bei Heapsort bei jedem Absinken mindestens $\log(k)$, insgesamt also ungefähr $n \cdot \log(n)$ Verschiebungen ausgeführt, während Quicksort mit weniger als $\frac{1}{2} \cdot n \cdot \log(n)$ Vertauschungen auskommt. Insgesamt ist bei Heapsort also mit $2 \cdot n \cdot \log(n)$, bei Quicksort dagegen mit weniger als $2 \cdot n \cdot \log(n)$ Zugriffen auf den Hauptspeicher zu rechnen. Daher wird Quicksort in der Praxis trotzdem oft etwas schneller als Heapsort sein.

Auch die weiteren elementaren Anweisungen spielen eventuell noch eine Rolle. Für eine genaue (nicht uniforme) Abschätzung muss man nun die Implementierung und die Bearbeitungszeiten, die der verwendete Computer für die verschiedenen Anweisungen benötigt, kennen. Erfahrene Informatiker(innen) können dies gut einschätzen, oder sie ermitteln - aufbauend auf den theoretischen Resultaten - die tatsächlichen Laufzeiten mit Hilfe von Testläufen oder aus Statistiken des Betriebssystems.

10.5 Mischen

Die bisherigen Sortierverfahren haben einzelne Elemente verglichen, die oft weit voneinander entfernt in der zu sortierenden Folge standen. Sie sind daher für riesige Datenbestände, die auf Hintergrundspeichern stehen, nur bedingt einsetzbar.

Daten werden von Hintergrundspeichern "stromartig" eingelesen, vergleichbar dem Lesen von Magnetbändern. Deshalb muss man immer möglichst viele Daten, die zusammenhängend verglichen und sortiert werden können, zusammenfassen und verarbeiten.

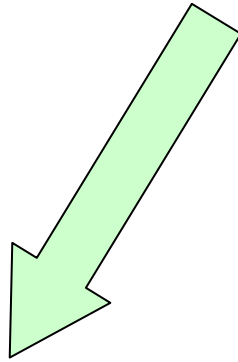
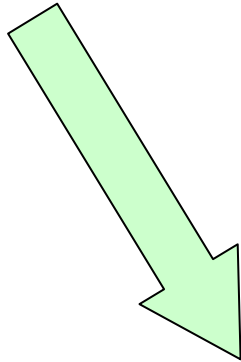
Hierfür ist vor allem das Zusammen-Mischen ("Verschmelzen") zweier bereits sortierter Datenströme geeignet.

(Hinweis: Ob auf " \leq " oder auf " $<$ " abgefragt wird, ist wichtig, sofern man Elemente innerhalb eines Feldes vergleicht. Fragt man auf " $<$ " ab, so ist das "Mischen" nicht stabil.)

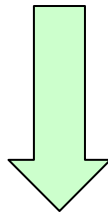
Feld A1
3 12 16 19 30 33 37

Feld A2
8 18 23 25 26 28 30

**Verschmelzen
zweier Folgen**



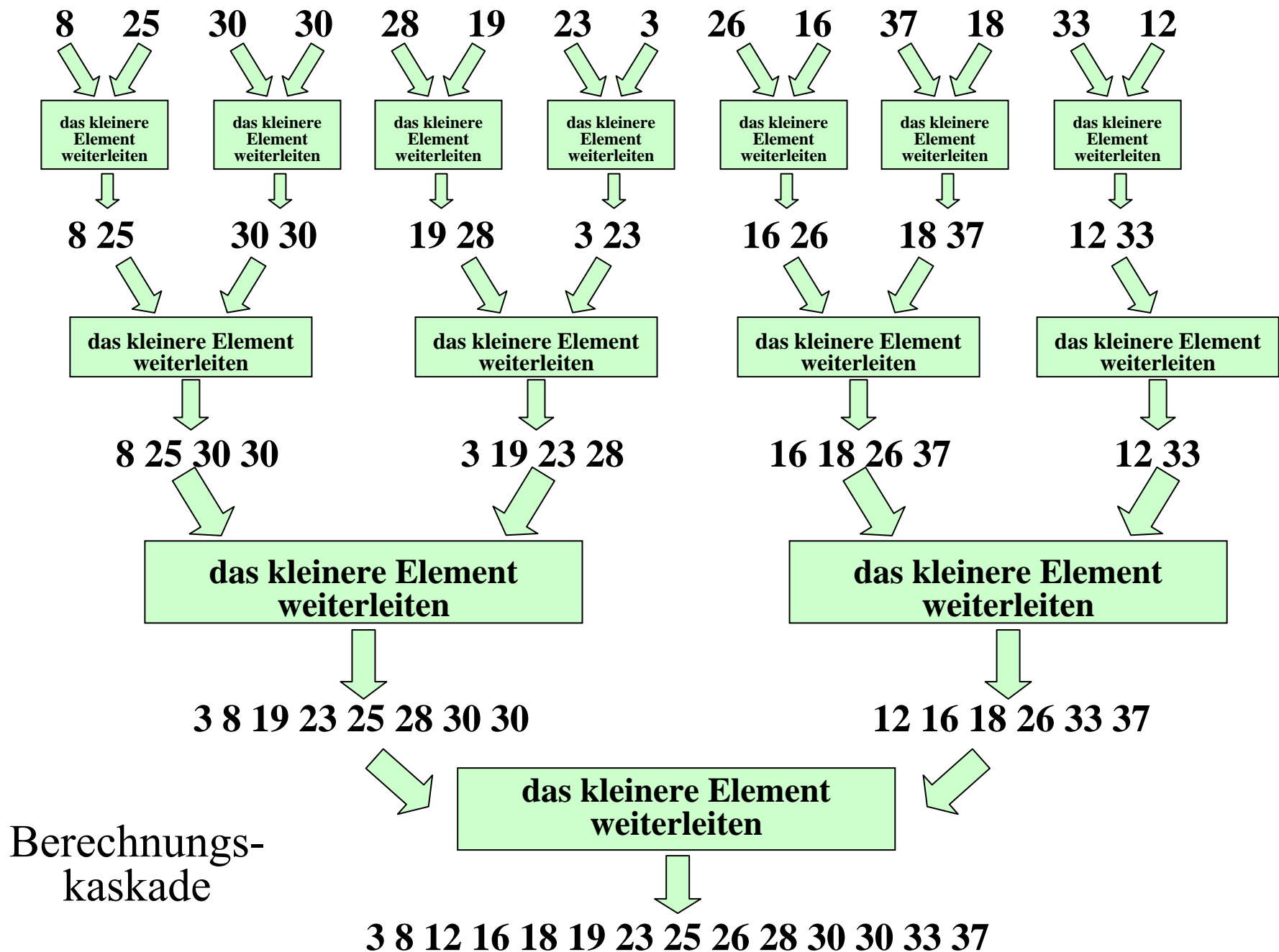
das kleinere Element weiterleiten



3 8 12 16 18 19 23 25 26 28 30 30 33 37
Feld B

```
i:= 1; j := 1; k := 0;  
while i ≤ n and j ≤ n loop  
  k := k+1;  
  if A1(i) ≤ A2(j)  
  then B(k) := A1(i); i := i+1;  
  else B(k) := A2(j); j := j+1;  
  end if;  
end loop;  
"Füge den Rest an B an"
```

Der gesamte Sortierprozess ist auf der nächsten Folie dargestellt.



10.5.1: Verschmelzen zweier Folgen vom Typ Vektor

```
procedure Verschmelzen (A1, A2: in Vektor; B: in out Vektor;  
                        LA1, RA1, LA2, RA2, LB: in Integer;  
                        RB: out Integer) is  
  
i, j, k: Integer;  
begin i:= LA1; j := LA2; k := LB-1;  
  while i ≤ RA1 and j ≤ RA2 loop  
    k := k+1;  
    if A1(i) ≤ A2(j) then B(k) := A1(i); i := i+1;  
      else B(k) := A2(j); j := j+1; end if;  
  end loop;                                     -- nun muss der Rest einer Folge noch angefügt werden  
  if i ≤ RA1 then  
    for m in i..RA1 loop k:=k+1; B(k):=A1(m); end loop;  
  else for m in j..RA2 loop k:=k+1; B(k):=A2(m); end loop; end if;  
  RB := k;                                       -- RB dient nur für Kontrollzwecke, kann entfallen  
end Verschmelzen;
```

10.5.2: Sortieren durch Mischen

Es soll ein Feld $A(1..n)$ durch Mischen sortiert werden. Zuerst die "*Bottom-Up-Denkweise*", die zu einem Iterationsverfahren führt: Man verschmilzt zunächst je zwei Folgen der Länge 1 zur sortierten Folgen der Länge 2 (man muss hierfür $n/2$ mal "Verschmelzen" aufrufen), dann verschmilzt man je zwei sortierte Folgen der Länge 2 zu sortierten Folgen der Länge 4 (hierfür muss man $n/4$ mal "Verschmelzen" aufrufen), danach das Gleiche für sortierte Folgen der Länge 4, 8, 16 usw., bis zwei Folgen der Länge $n/2$ zu einer sortierten Folge der Länge n verschmolzen wurden. Sofern n eine Zweierpotenz war, funktioniert dieses Verfahren bereits; im allgemeinen Fall muss man beim Verschmelzen unterschiedliche Längen von Folgen berücksichtigen (vgl. Beispiel). Dieses Mischen heißt in der Literatur "straight mergesort".

Sortieren durch Mischen (Fortsetzung)

Nun die "*Top-Down-Denkweise*": Um ein Feld zu sortieren, sortiert man zuerst die linke Hälfte, dann die rechte Hälfte und verschmilzt die beiden sortierten Folgen. Das Ergebnis ist eine rekursive Prozedur.

Da dieses Vorgehen leichter zu verstehen und aufzuschreiben ist, wird die rekursive Version im Folgenden realisiert.

Der Ansatz ist einfach. Die Hauptschwierigkeit besteht hier in der präzisen Angabe der jeweiligen Teilfeld-Grenzen.

Wir verzichten auf volle Allgemeinheit und wollen "nur" das Feld $A(L..R)$ sortieren. Sortierte Teilfelder $A(x..y)$ und $A(u..v)$ mischen wir in ein Hilfsfeld $B(L..R)$ und schreiben danach das Ergebnis wieder nach A zurück.

Beachten Sie, dass die Abfrage " $A(i) \leq A(j)$ " die Stabilität des Mischens sichert (im Gegensatz zu " $A(i) < A(j)$ ").

10.5.3: Programm zum Sortieren durch Mischen ("Mergesort")

```
procedure Mergesort (L, R: in Integer) is  
  Mitte: Integer; i, j, k: Integer;           -- Die Felder A und B sind global.  
begin                                         -- Es wird A(L..R) sortiert.  
  if R > L then Mitte := (L+R)/2;           -- Sortiere rekursiv zwei Hälften der Folge  
    Mergesort(L, Mitte); Mergesort(Mitte+1, R);  
    i := L; j := Mitte+1; k := L-1;         -- Nun mischen von A nach B, wie in 10.5.1  
    while i ≤ Mitte and j ≤ R loop k := k+1;  
      if A(i) ≤ A(j) then B(k) := A(i); i := i+1;  
        else B(k) := A(j); j := j+1; end if;  
    end loop;                                -- nun muss der Rest einer Folge noch angefügt werden  
    if i ≤ Mitte then  
      for m in i..Mitte loop k:=k+1; B(k):=A(m); end loop;  
    else for m in j..R loop k:=k+1; B(k):=A(m); end loop; end if;  
    for m in L..R loop A(m) := B(m); end loop; -- zurückkopieren nach A  
  end if;  
end Mergesort;
```

Wir haben das Verfahren mit Feldern realisiert.

Es ist aber klar, dass man es auch leicht mit linearen Listen implementieren kann, wobei nur Zeiger umgesetzt werden müssen. Hierbei kann man alle Umspeicherungen vermeiden. Vor allem der Teil im Programm, der mit

-- nun muss der Rest einer Folge noch angefügt werden

beginnt, kann durch eine einzige Zeigersetzung erledigt werden.

Durchdenken Sie die Vor- und Nachteile eines solchen Verfahrens und entwerfen Sie ein Programm, für das die zu sortierenden Daten als einfach verkettete lineare Liste vorliegen.

10.5.4 Aufwandsabschätzung für das Mischen:

Wenn $V(n)$ die maximale Zahl der Vergleiche zum Sortieren von n Elementen ist, dann gilt:

$$V(1) = 0 \quad \text{und für alle } n > 1: \quad V(n) = 2 \cdot V(n/2) + n - 1.$$

Die maximale Rekursionstiefe ist beim Mischen $\log(n)$, da in jedem Rekursionsschritt halbiert wird. Also muss die maximale Zahl der Vergleiche durch $n \cdot \log(n) - n/2 - n/4 - n/8 - \dots - 1 = n \cdot \log(n) - n + 1$ beschränkt sein (das sieht man unmittelbar am Beispiel auf Folie 426). Dies ist auch die Lösung der obigen Gleichung für Zweierpotenzen n :

$$V(n) = n \cdot \log(n) - n + 1 \quad (\text{Beweis durch Einsetzen}).$$

Mergesort ist also ein garantiertes $n \cdot \log(n)$ -Verfahren.

Die Zahl seiner Vergleiche kommt sehr dicht an die untere theoretische Grenze heran, siehe Satz 10.1.11.

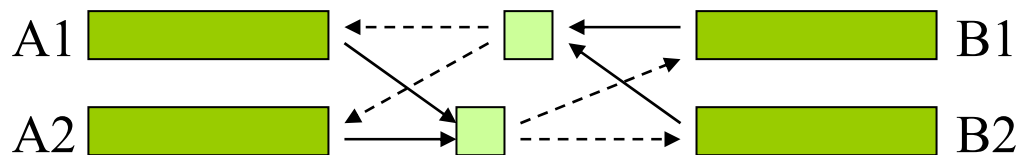
Bei Mergesort sind *viele Umspeicherungen* erst beim Verschmelzen von A nach B und dann zurück nach A erforderlich. Wie hoch ist die Zahl der Speichervorgänge? In jeder Rekursionstiefe werden alle Daten genau einmal nach B und wieder zurück transportiert. Folglich werden insgesamt stets $4 \cdot n \cdot \log(n)$ Speicherzugriffe durchgeführt. (Dies setzt allerdings voraus, dass das Programm leicht modifiziert wird, dass also "if $A(i) \leq A(j)$ then" nicht zu zusätzlichen Speicherzugriffen führt.)

Dies scheint viel zu sein. Man mache sich aber klar, dass bei Quicksort je Rekursionstiefe bis zu $n/2$ Vertauschungen stattfinden, die jeweils 4 Speicherzugriffe erfordern, weshalb Quicksort auch im günstigsten Falle, dass die Rekursionstiefe durch $\log(n)$ beschränkt bleibt, bis zu $2 \cdot n \cdot \log(n)$ Umspeicherungen ausführen kann.

10.5.5 Varianten des Sortierens durch Mischen

Das obige Vorgehen bezeichnet man als "*Zwei-Phasen-Mischen*", da die Daten von Feld A nach B verschmolzen (erste Phase) und anschließend zurück nach A (zweite Phase) kopiert werden. Man spricht auch von "*2-Wege-Mischen*", da ein Weg nach B und einer zurück nach A führt.

Natürlich kann man die Rolle der Ziel-Felder in jedem Durchgang ändern, d.h.: Man verschmilzt zwei sortierte Folgen von A1 und A2 abwechselnd auf die Felder B1 und B2 und anschließend zwei sortierte Folgen von B1 und B2 zurück nach A1 bzw. A2 usw. So spart man die Hälfte der Umspeicheroperationen. Man muss sich hierbei die jeweiligen Grenzen merken und bis zu $2n$ zusätzliche Speicherplätze bereitstellen.



Ein-Phasen-Mischen, wobei 4 Wege möglich sind.

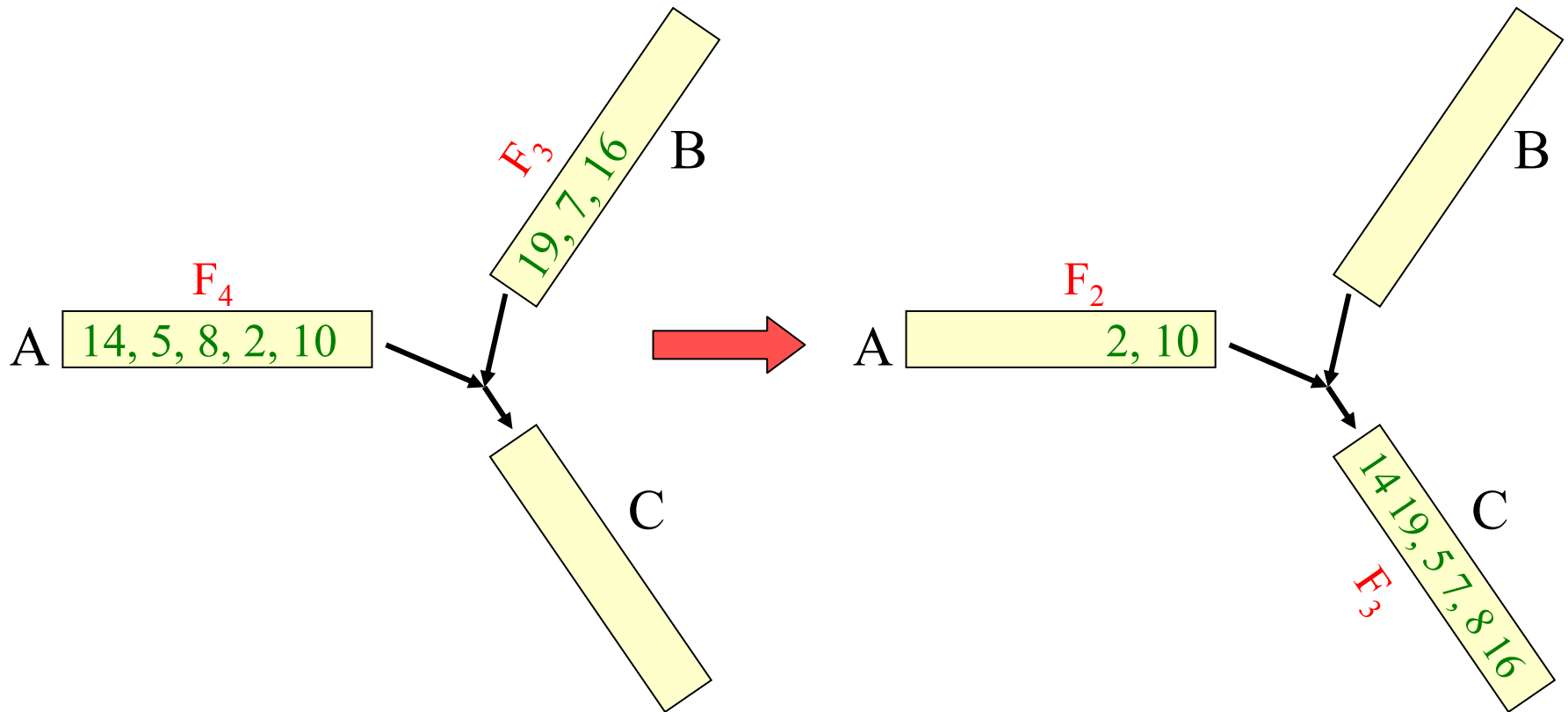
Die durchgezogenen Pfeile bezeichnen das Zusammenführen zweier sortierter Folgen. Das sortierte Ergebnis des Verschmelzens wird abwechselnd über einen der gestrichelten Pfeile weitergeleitet.

3-Wege-Mischen:

Man kann auch mit drei Feldern A, B und C arbeiten. Zuerst wird die zu sortierende Folge auf zwei Bänder A und B verteilt und zwar F_k sortierte Teilfolgen auf Band A und F_{k-1} sortierte Teilfolgen auf Band B (F_k ist die k -te Fibonaccizahl, siehe 8.4.7; falls die Anzahlen nicht "aufgehen", so fülle man mit "dummy"-Folgen auf die nächste Fibonaccizahl auf). Nun werden sortierte Teilfolgen von A und B nach C gemischt, bis das Feld B keine sortierte Folge mehr besitzt. Dann besitzen C genau F_{k-1} und A genau $F_k - F_{k-1} = F_{k-2}$ sortierte Teilfolgen. Nun werden die sortierten Teilfolgen von A und C nach B gemischt, bis das Feld A keine sortierte Teilfolge mehr besitzt (auf den Feldern B und C befinden sich nun F_{k-2} bzw. F_{k-3} sortierte Teilfolgen). Nun wird A das Ziel-Feld, d.h., es werden sortierte Teilfolgen von B und C nach A gemischt usw., bis am Ende auf einem der Bänder die sortierte Gesamtfolge steht. (Man wählt hier Fibonacci-Zahlen, weil man dann nicht ständig die Folgenlängen abfragen muss.)

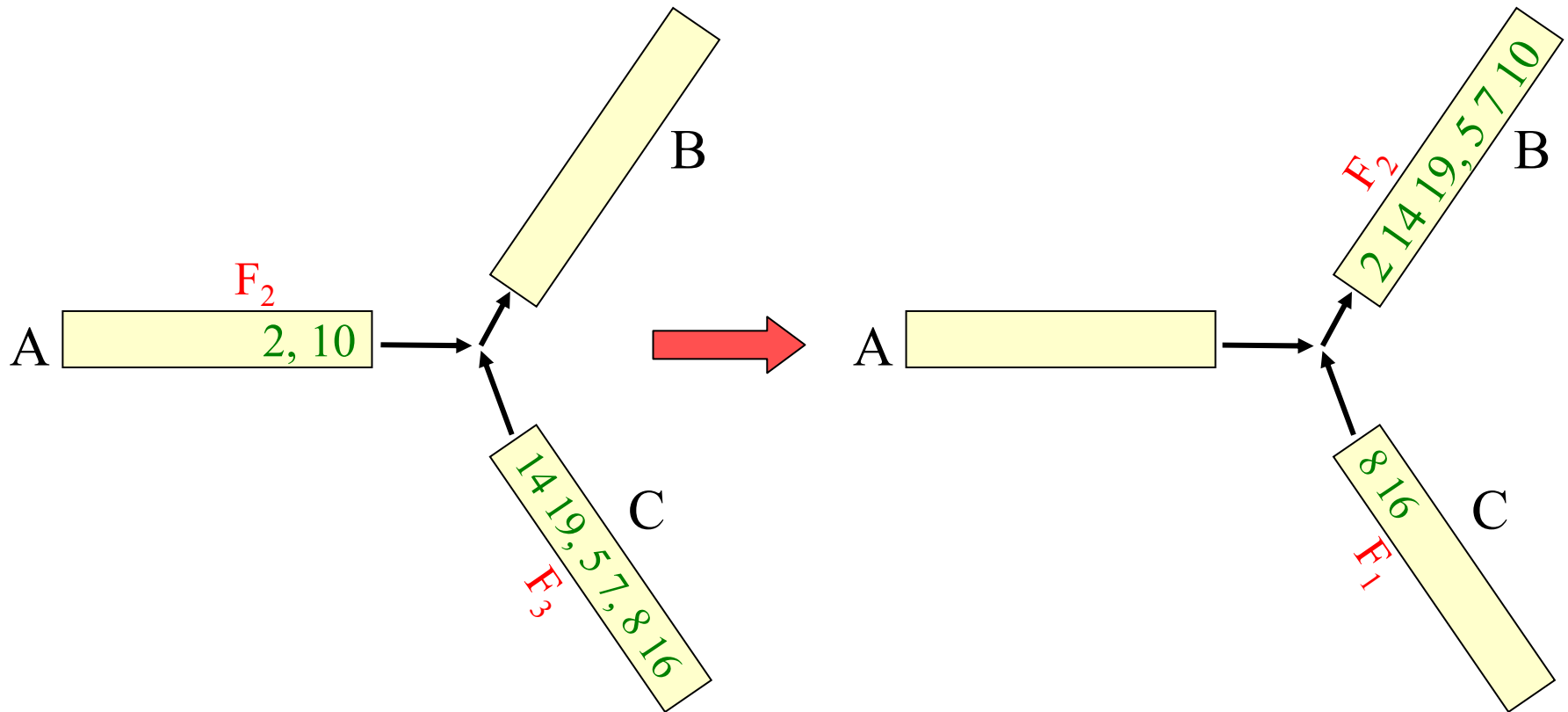
Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16

Dies sind $F_6 = 8$ Elemente. Verteile also $F_5 = 5$ und $F_4 = 3$ Elemente auf zwei Bänder A und B und mische diese auf Band C:



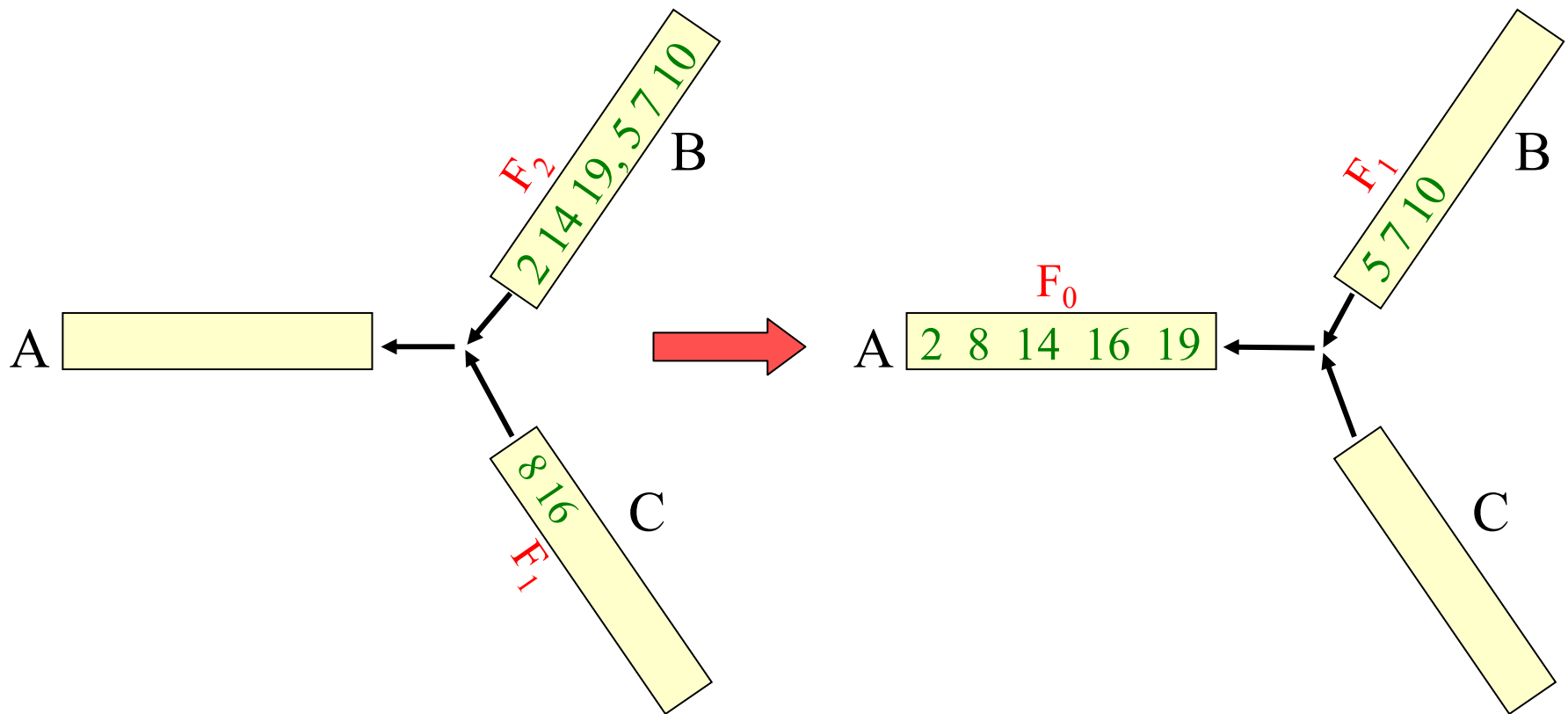
Verschmelze nun $F_4 = 3$ sortierte Teilfolgen von A mit $F_4 = 3$ sortierten Teilfolgen von B. Das Band B wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



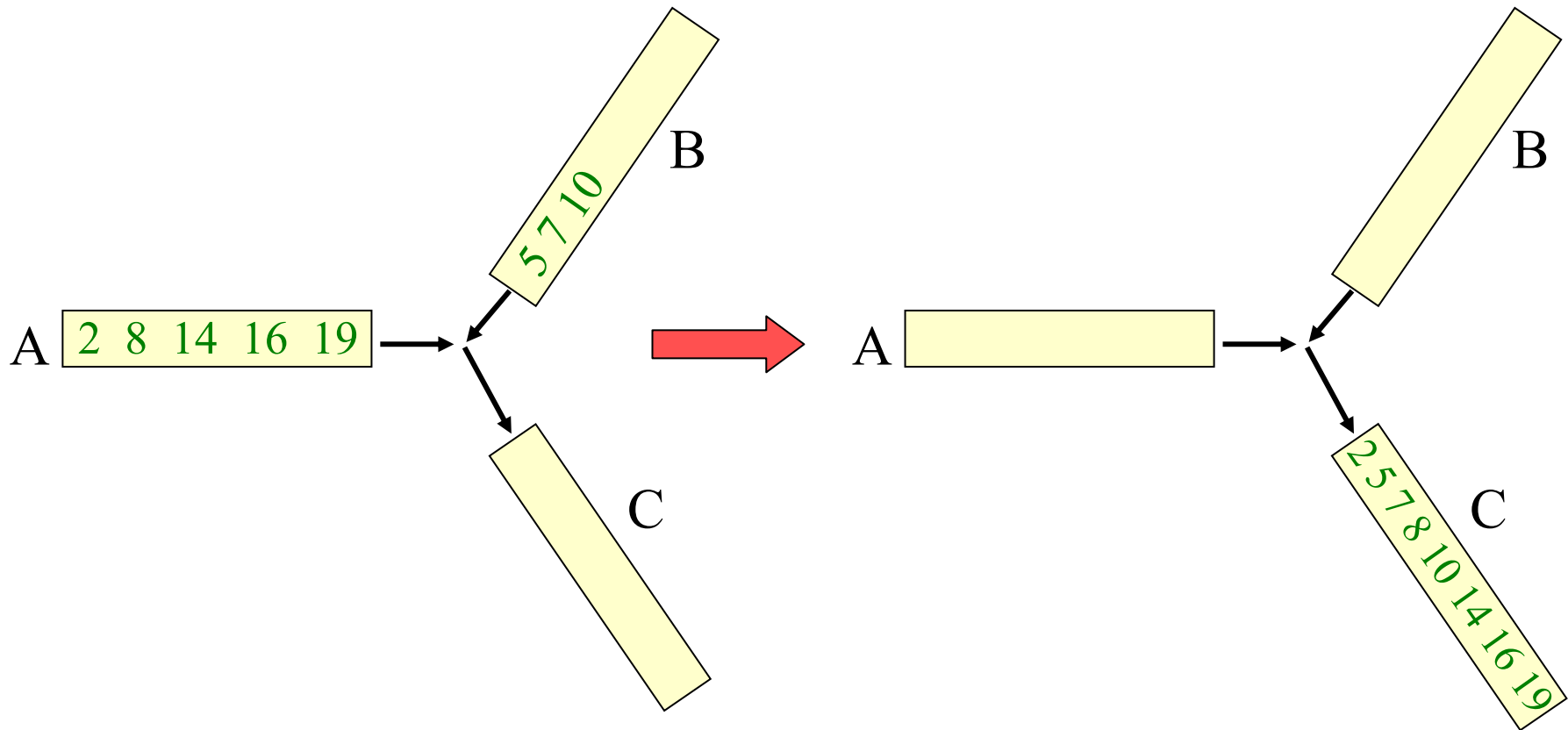
Verschmelze $F_3 = 2$ sortierte Teilfolgen von A mit $F_3 = 2$ sortierten Teilfolgen von C. Das Band A wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



Verschmelze $F_2 = 1$ sortierte Teilfolge von B mit $F_1 = 1$ sortierten Teilfolge von C. Das Band C wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



Verschmelze $F_1 = 1$ sortierte Teilfolge von A mit $F_0 = 1$ sortierten Teilfolge von B auf Band C. Fertig.

10.5.6 Natürliches Mischen:

Eine Teilfolge $a_i a_{i+1} \dots a_j$ der Folge $a_1 a_2 \dots a_n$ heißt ein Lauf, wenn dies eine maximal lange nicht fallende Teilfolge ist, d.h., wenn $a_{i-1} > a_i \leq a_{i+1} \leq \dots \leq a_j > a_{j+1}$ gilt (wenn $i=1$ oder $j=n$ ist, so entfallen die entsprechenden Ungleichungen am Rande).

Statt Teilfolgen der Länge 1, dann der Länge 2, 4, ... usw. zu mischen, kann man in jedem Durchgang die vorhandenen Läufe mischen, wodurch sich die Rekursionstiefe und damit die Zahl der Umspeicherungen verringert, die Zahl der Abfragen aber wegen des Tests, wo ein Lauf endet, insgesamt nicht geringer wird. Ein solches Ausnutzen der zufällig vorhandenen Teilsortierungen bezeichnet man als *natürliches Mischen*. Dadurch wird das Mischen mit seiner garantierten $n \cdot \log(n)$ -Laufzeit auch zu einem ordnungsverträglichen Sortierverfahren.

Aufgabe: Programmieren Sie das "natürliche 3-Wege-Mischen".

10.5.7 Parallelisieren

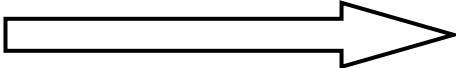
Hat man einen Baustein, der das Verschmelzen zweier Folgen vornimmt, so kann man in jeder Rekursionstiefe alle Operationen parallel ausführen (siehe die Kaskade zu Beginn von 10.5; in der obersten Zeile kann man alle $n/2$ Vergleiche parallel zueinander durchführen).

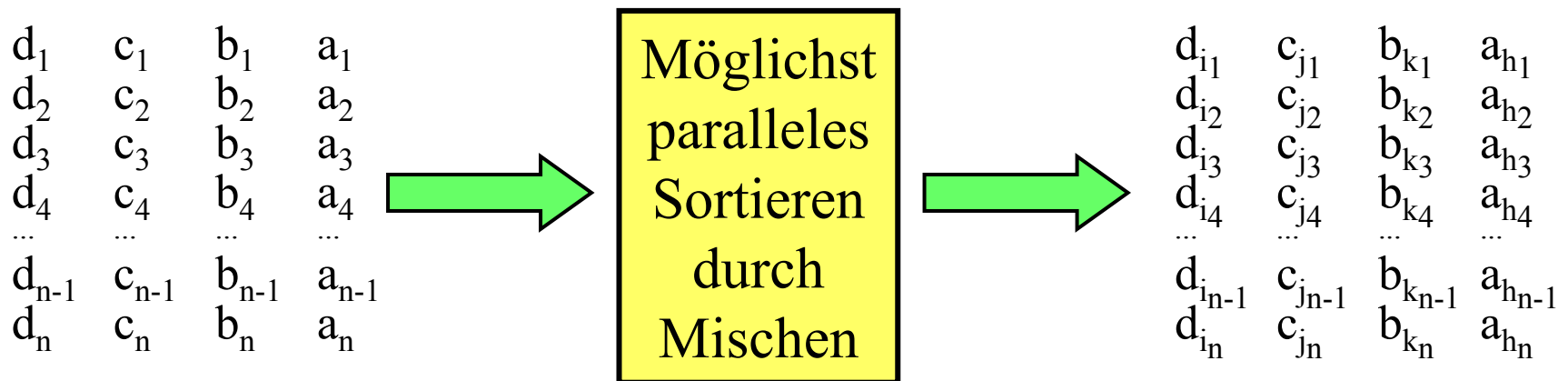
Man benötigt dann $n/2$ solcher Bausteine zum Mischen von Teilfolgen der Länge 1, man braucht $n/4$ dieser $n/2$ Bausteine für das Verschmelzen aller Teilfolgen der Länge 2 usw.

Durch geschicktes Zusammenschalten kann man also die Kaskade mit $n/2$ solcher Bausteine realisieren.

Wie lange dauert dann das Sortieren? Die erste Schicht benötigt genau 2 "Takte", die nächste 4, die nächste 8 usw. bis zur letzten Schicht mit n Takten. Folglich kann man wegen $2+4+8+\dots+n/4+n/2+n=2n-2$ die n Daten mit diesem Parallelverfahren in $2n-2$ Schritten sortieren. Der "Preis" hierfür sind die $n/2$ Bausteine und ihr Verschaltungsnetz.

Bringt dieses Verfahren trotzdem Vorteile, z.B. im Falle, dass man viele Zahlenfolgen a, b, c, d, ... sortieren möchte?

Folgen nacheinander  *sortierte Folgen nacheinander*



Faktisch bringt unser Mischverfahren leider kaum etwas, weil ein "Pipelining" zwar denkbar ist, aber wegen der sequenziellen Abarbeitung des letzten Schritts mit mindestens $n/2$ Schritten zu entsprechend langen Wartezeiten führt. Ein schnelles Verfahren, das das ständige Einschleusen der nächsten Folge in den Verarbeitungsprozess erlaubt, stellen wir in 10.7 vor.

10.6 Streuen und Sammeln

Manchmal liegen die zu sortierenden Werte $a_1 a_2 \dots a_n$ in einem festen Intervall $[UNT, OB]$: $UNT \leq a_i \leq OB$.

Der Einfachheit halber nehmen wir an, die a_i seien natürliche Zahlen und es seien $UNT = 0$ und $OB = m-1$.

Bucket sort: Verteile ("streue") die n Elemente $A(1..n)$ auf m nacheinander angeordnete Fächer $bucket(0..m-1)$ ("Eimer" genannt, daher "bucket sort"), hole sie anschließend in der Reihenfolge der Fächer wieder heraus und lege sie im Feld A ab.

Aufwand: $O(n+m)$ sowohl für die Zeit als auch für den Platz.

Einsatz: Vor allem, wenn m in der Größenordnung von n liegt.

Programmstück (fügen Sie die Listenbearbeitung selbst hinzu):

```

declare A: array (1..n) of Integer; k: 0..n;    -- n ist global
bucket: array (0..m-1) of "liste von Integer"; ...
begin
    for j in 0..m-1 loop bucket(j) := "leer"; end loop;
    for i in 1..n loop                                -- streuen
        "hänge A(i) an bucket(A(i)) an";
    end loop;
    k := 0;
    for j in 0..m-1 loop                                -- sammeln
        while "bucket(j) nicht leer" loop
            k := k+1; A(k) := "erstes Element von bucket(j)";
            "Entferne aus bucket(j) das erste Element"; end loop;
        end loop;
    end;

```

10.7 Paralleles Sortieren

Das Sortieren von n Elementen kostet mindestens $n \cdot \log(n)$ Vergleiche, wenn nur ein Prozessor vorhanden ist. Ein guter Rechner kann heute etwa 10 Millionen Vergleiche pro Sekunde durchführen, sofern man 32-stellige Zahlen als Schlüssel verwendet. Setzt man für die Umspeicherungen usw. den Faktor 10 an, so lassen sich mit einem Programm 1 Million Vergleiche einschl. der übrigen Operationen pro Sekunde durchführen. Will man maximal eine Minute auf das Ergebnis warten, so lassen sich also $60 \cdot 10^6$ Vergleiche ausführen. Berechne n so, dass $n \cdot \log(n) = 60 \cdot 10^6$ gilt, d.h., es lassen sich knapp 3.000.000 Schlüssel in einer Minute sortieren.

Solche Größenordnungen sind selten, so dass im Prinzip das Sortieren heute kein Problem mehr darstellen sollte. Allerdings entstehen Probleme, wenn eine Sortierung sehr schnell erfolgen muss, weil sicherheitskritische Systeme hiervon abhängen oder andere Gründe vorliegen.

Will man also schneller sortieren, so kann man entweder auf noch schnellere Rechner warten oder man kann eine Beschleunigung des Sortierens durch paralleles Vorgehen erhoffen. Wir stellen hierzu zwei „einfache“ Verfahren vor. (Weitere Verfahren finden Sie z.B. in dem Buch von Ingo Wegener, "Effiziente Algorithmen für grundlegende Funktionen", Teubner-Verlag.)

Verfahren 1:

Lineare Kette mit n Prozessoren.

Verfahren 2:

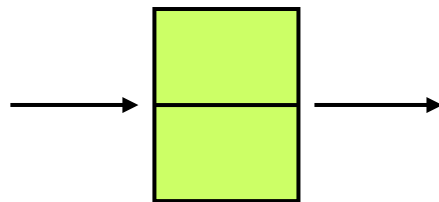
Divide and Conquer Verfahren mit $(1/4) \cdot n \cdot \log^2(n)$ Prozessoren.

10.7.1 Verfahren 1: Lineare Kette

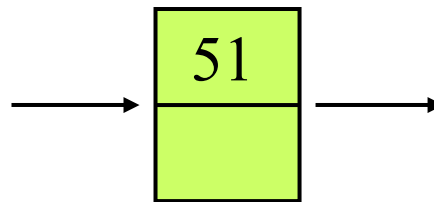
Hierzu betrachten wir einen Prozessor, der

zwei Speicherplätze für Zahlen,
einen Eingang (links) und
einen Ausgang (rechts)

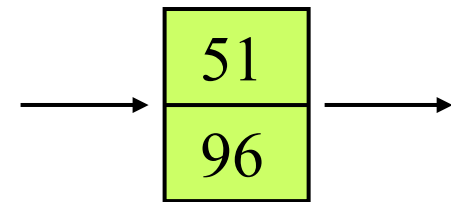
besitzt:



Grundbaustein,
leer



Grundbaustein
mit einer Zahl



Grundbaustein
mit zwei Zahlen

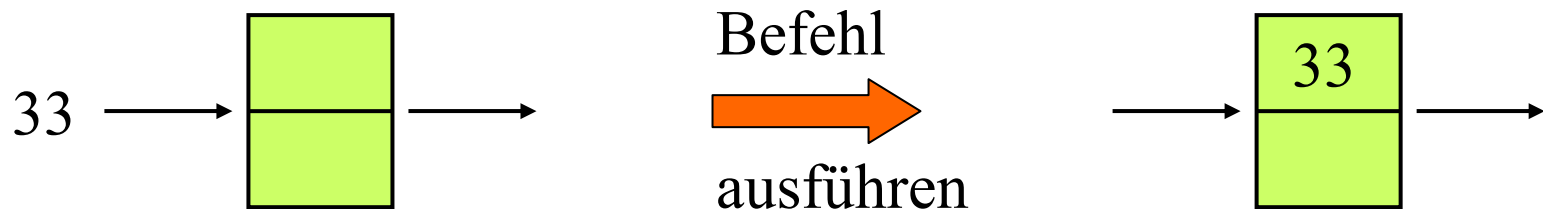
(Die Zahl darf auch unten stehen.)

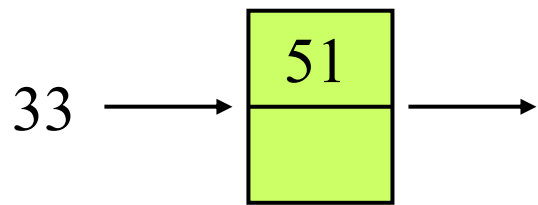
Der Grundbaustein kann nur folgenden **Befehl** ausführen:


1. Falls er zwei Zahlen besitzt, gibt er die größere der beiden nach rechts aus.
2. Falls eine Zahl von links kommt, legt er sie in einen seiner freien Speicherplätze.

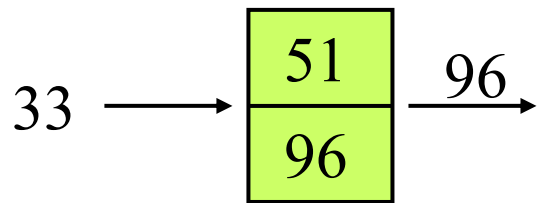
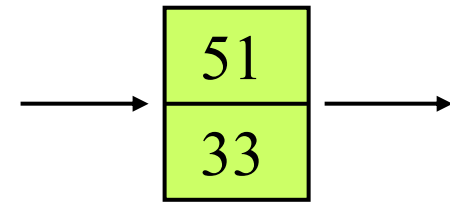
Die Befehlsteile 1. und 2. werden gleichzeitig ausgeführt!


Wenn die Bedingung in 1. und/oder in 2. zutrifft, dann muss der Befehl auch ausgeführt werden.

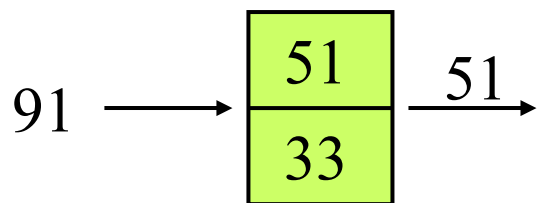
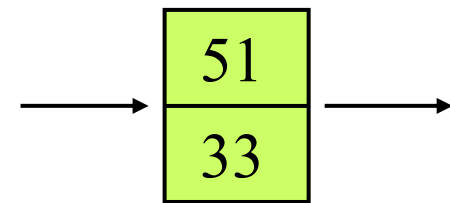





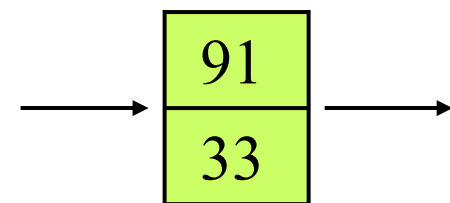
Befehl

ausführen



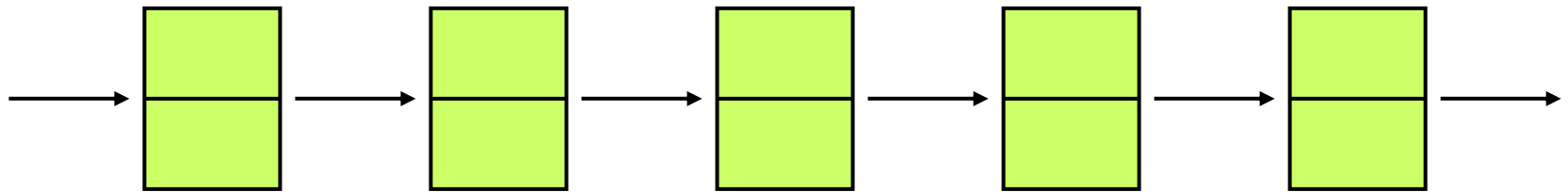
Befehl

ausführen



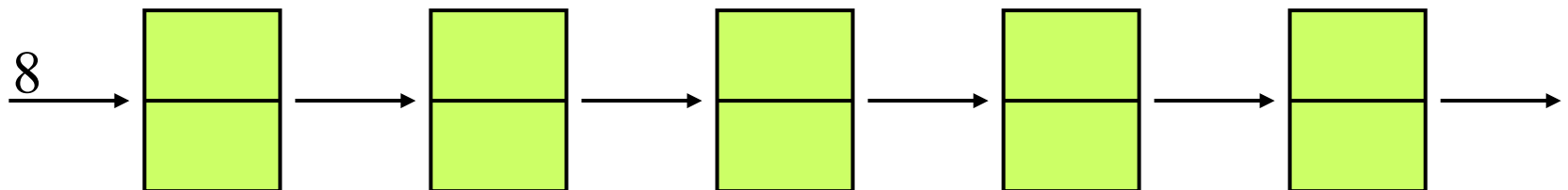
Befehl

ausführen

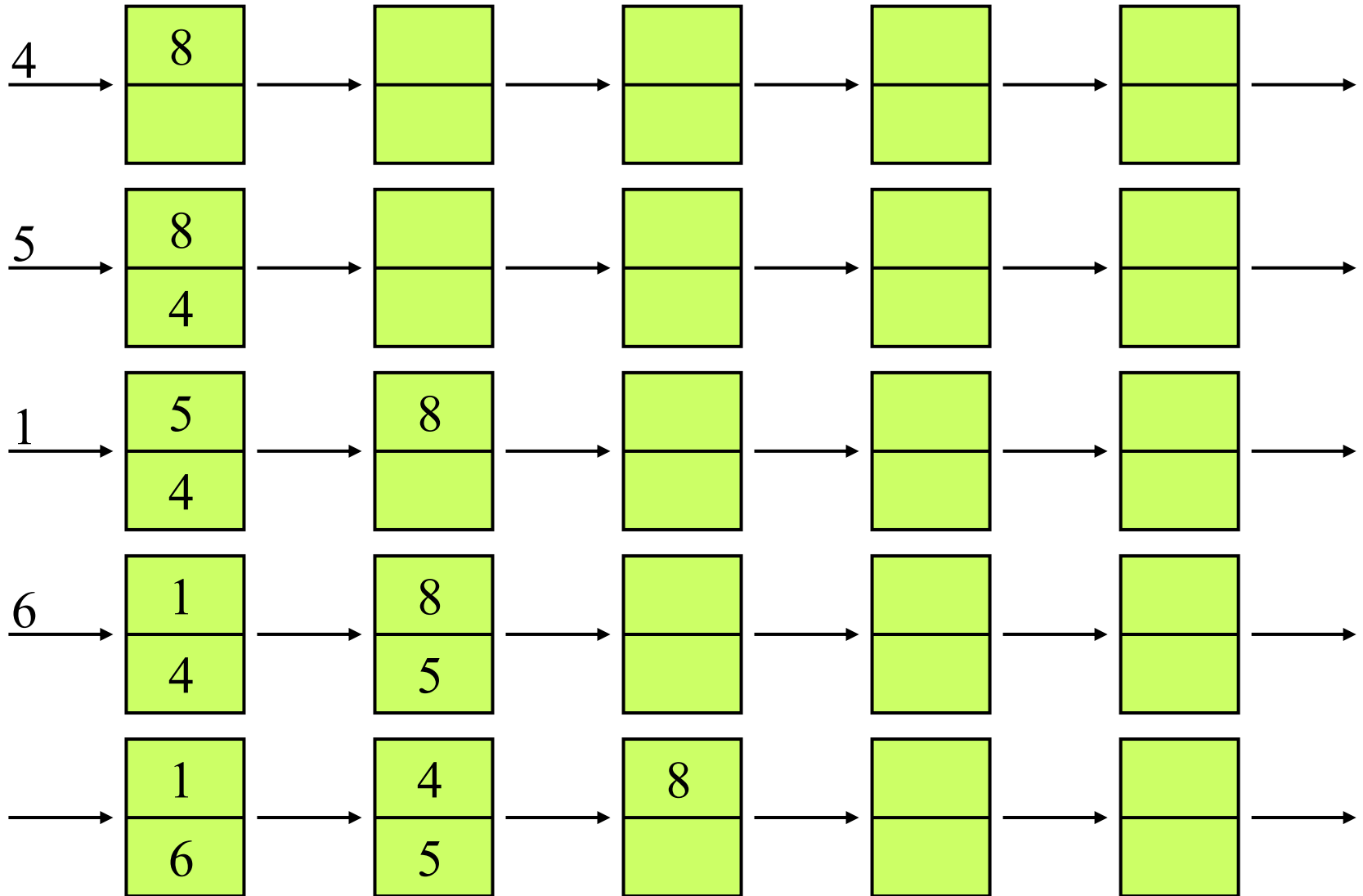


Nun koppeln wir $n=5$ solche Grundbausteine zu einer Kette aneinander:

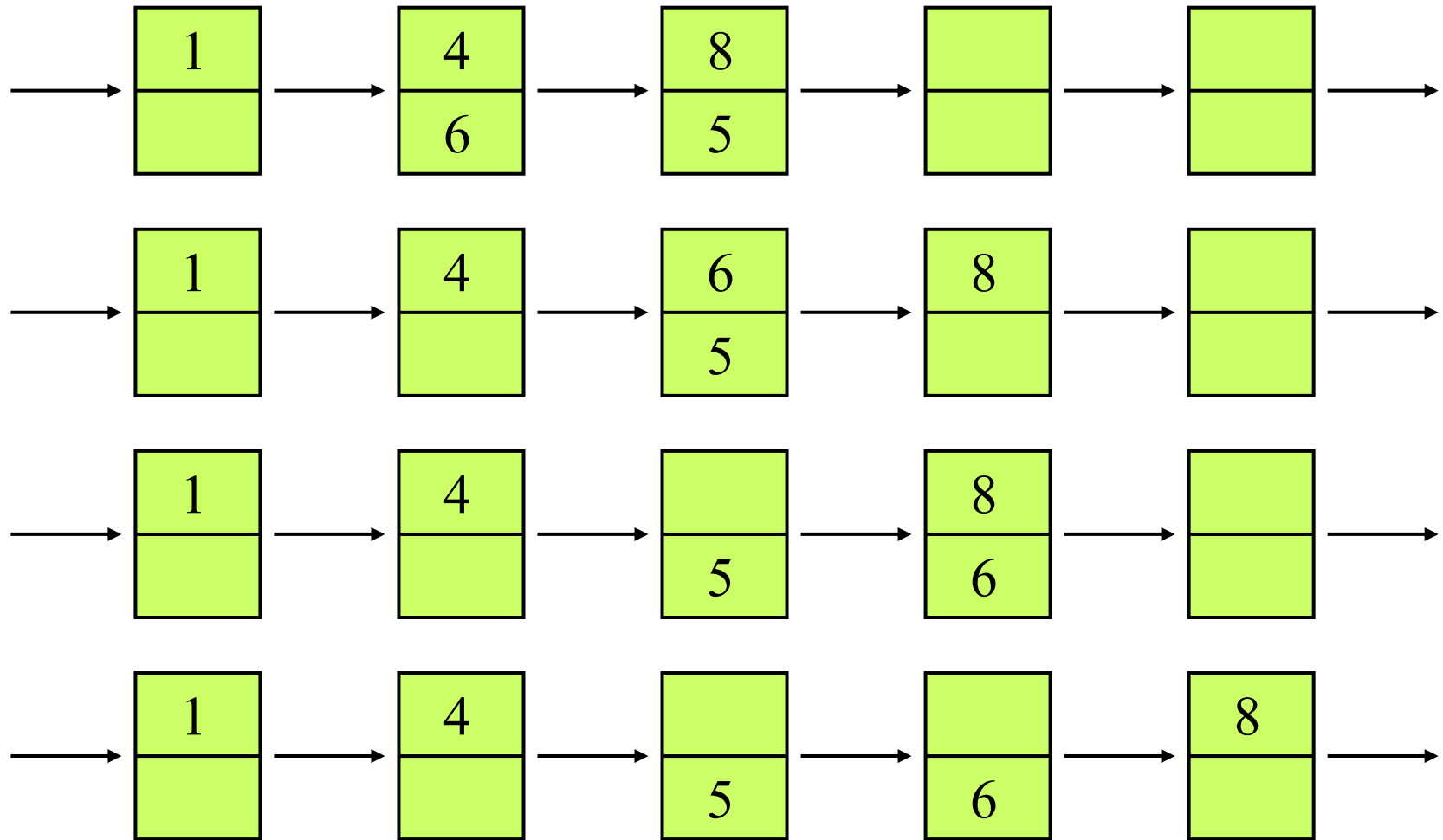


Wir verlangen, dass alle n Bausteine synchron arbeiten. Nun geben wir in $n=5$ Takten die Zahlen 8, 4, 5, 1, 6 von links ein:





Die Eingabe ist beendet. Die Kette arbeitet aber noch weiter, bis in jedem Baustein genau eine Zahl steht.



Nach $2n-1$ Takten endet das Verfahren und jeder Baustein gibt im $2n$ -ten Takt seine Zahl aus. Diese Folge ist sortiert.

10.7.2 Wie viele Schritte benötigt dieses Verfahren?

Offensichtlich sind n Werte nach **$2n-1$** Schritten sortiert.

Hierfür benötigt man n Prozessoren.

Für große Werte von n ist dies eine deutliche Verbesserung gegenüber den bisherigen $O(n \cdot \log(n))$ -Verfahren.

Hinweis: In 10.5.7 haben wir ein Verfahren skizziert, wie man n Werte schon mit $n/2$ Prozessoren parallel in $2n-2$ Schritten sortieren kann. Jene Prozessoren hatten aber zwei Eingänge und waren komplexer verschaltet.

Gibt es bessere Verfahren? Möglichst solche, die viel schneller als in $O(n)$ Schritten arbeiten - dafür dürfen sie dann auch mehr als $O(n)$ Prozessoren besitzen ...

10.7.3 Verfahren 2: Divide and Conquer Ansatz

Der Divide-and-Conquer-Ansatz lautet:

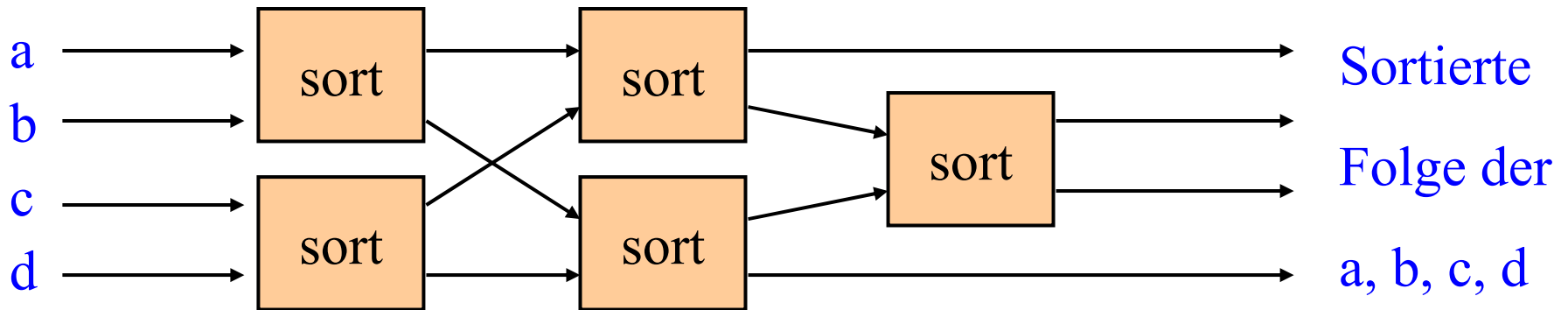
Zerlege das Problem in z.B. zwei Teilprobleme, löse diese und setze aus den Teillösungen die Gesamtlösung zusammen.

Wir gehen nun von einer allgemeinen Struktur aus, bei der die zu sortierenden Elemente nicht nacheinander, sondern parallel zueinander eingegeben werden. Dies bedeutet, dass wir mit mindestens $O(n)$ Grundbausteinen rechnen müssen, um n Elemente zu sortieren.

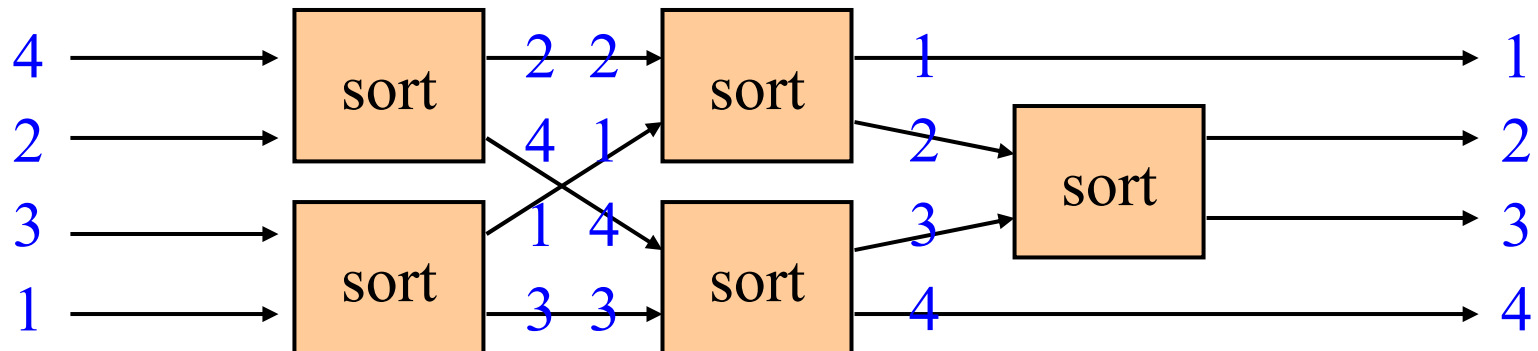
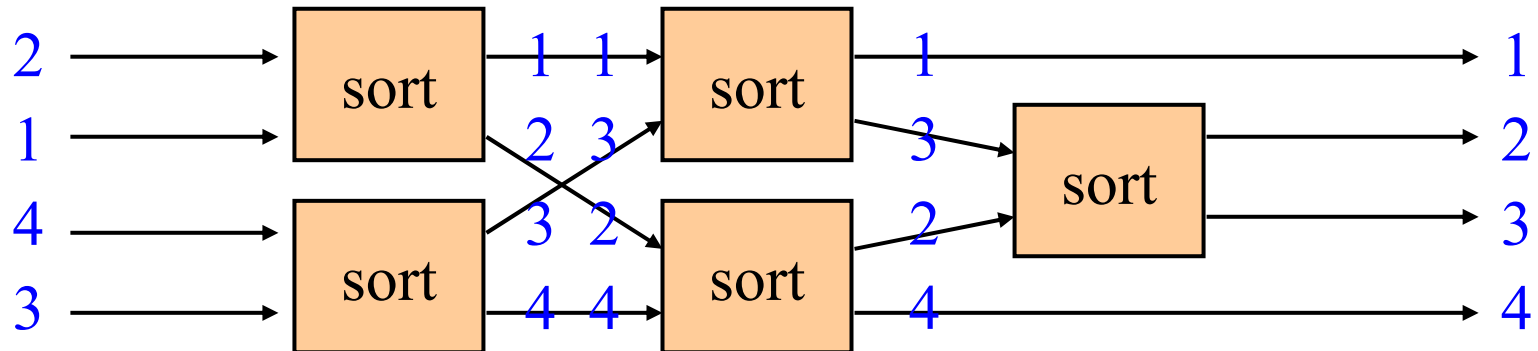
10.7.4 Als Grundbaustein verwenden wir einen elementaren **Sortierbaustein** **sort** für zwei Werte:



Hiermit kann man beispielsweise $n = 4$ Werte wie folgt sortieren:

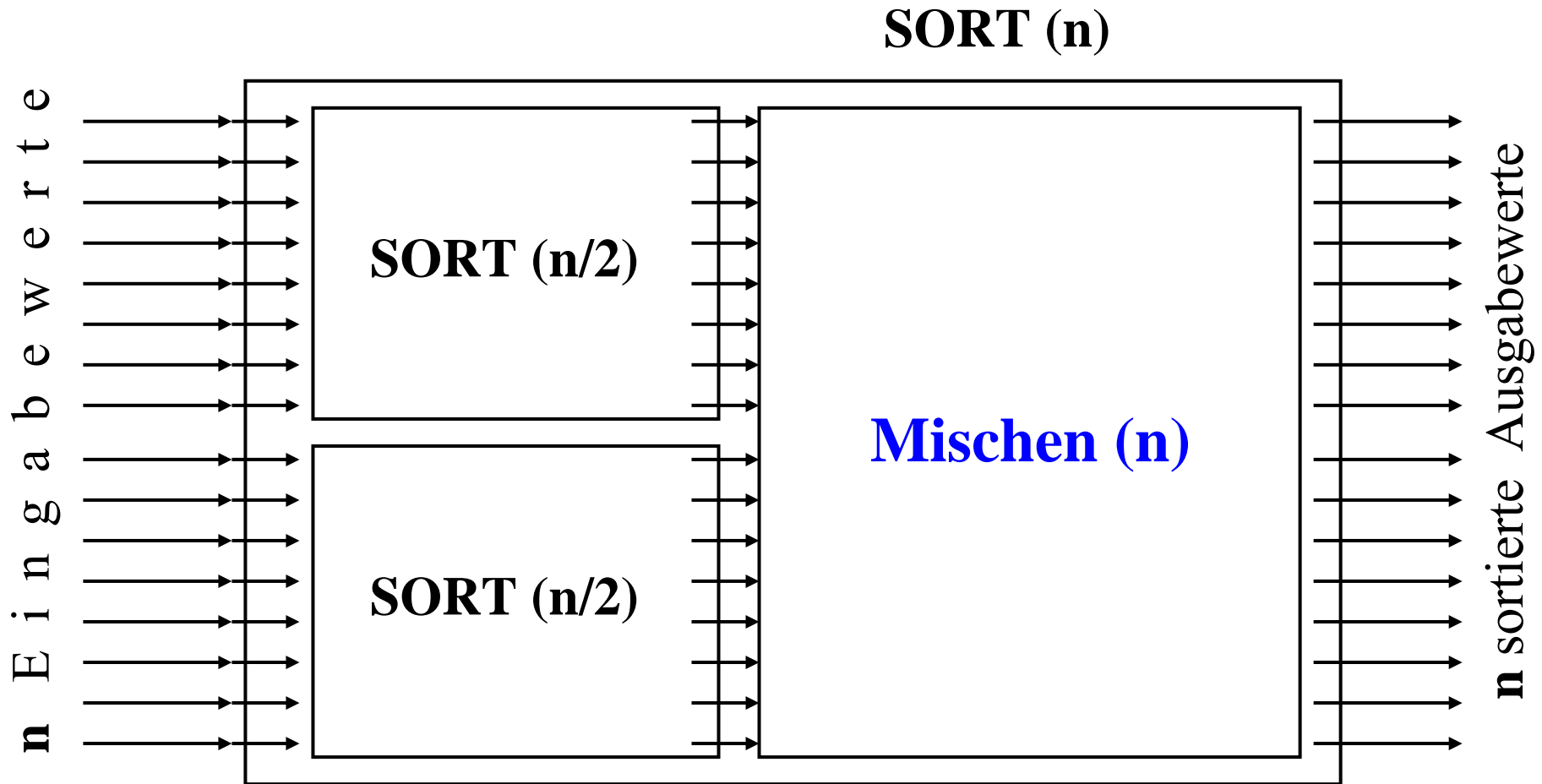


Beispiele:



4 Elemente werden in 3 Schritten mit 5 Bausteinen sortiert.

10.7.5 Divide and Conquer Ansatz für n Eingabewerte:

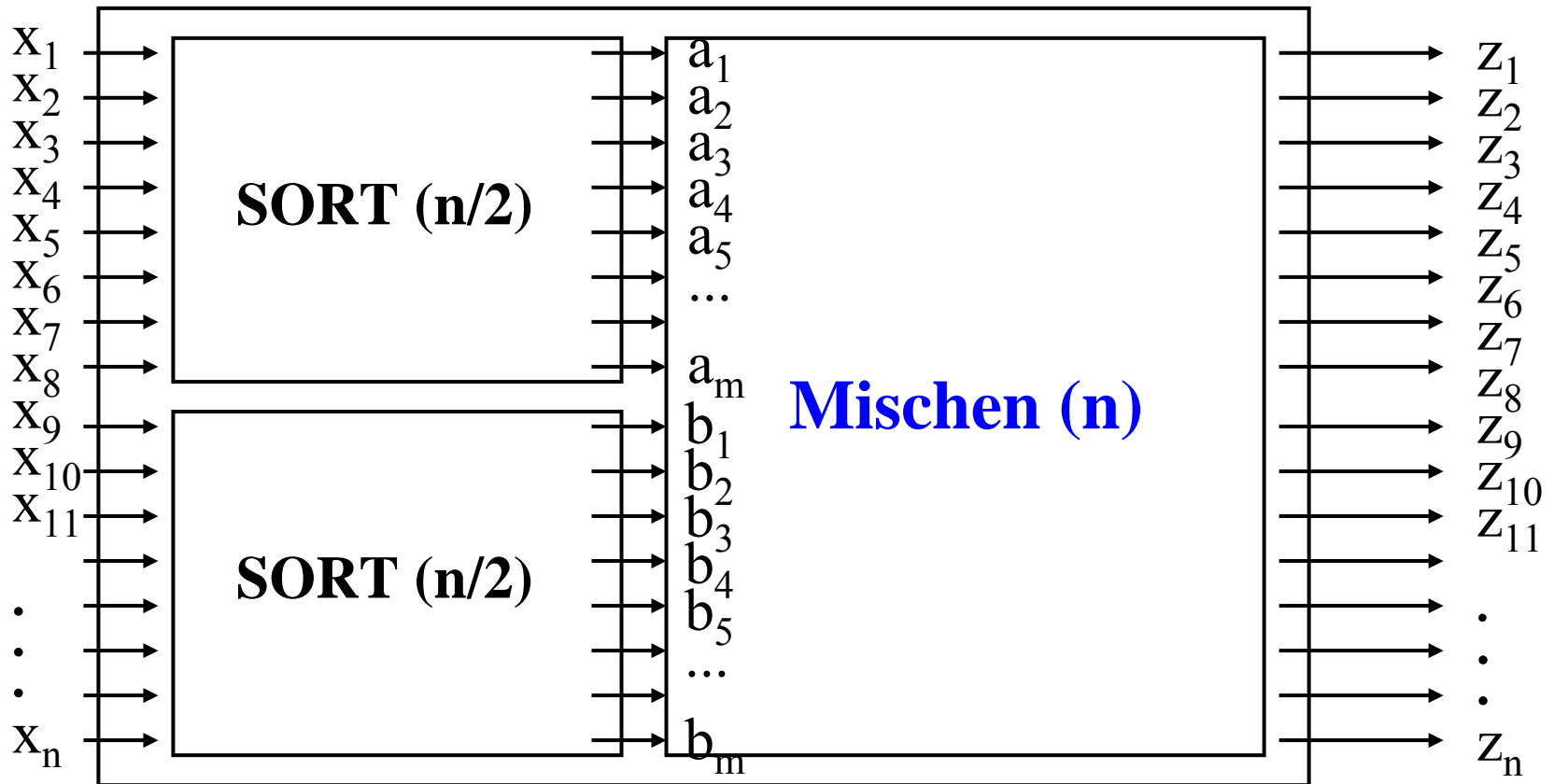


Genauer: Gegeben sind n Eingabewerte x_1, x_2, \dots, x_n .

Wir erhalten 2 sortierte Folgen mit je m Elementen:

a_1, a_2, \dots, a_m und b_1, b_2, \dots, b_m ($m = n/2$)

und n sortierte Ausgabewerte z_1, z_2, \dots, z_n . **Sort (n)**



Die Rekursion sorgt dafür, dass die Folgen a_1, a_2, \dots, a_m und b_1, b_2, \dots, b_m sortiert sind ($m=n/2$).

Wenn es einen "guten" Baustein **Mischen(n)** gibt, dann lässt sich hieraus auch ein "gutes" Sortierwerk für $n = 2^s$

Eingabewerte, für jede natürliche Zahl s , konstruieren.

In der Tat gibt es eine Technik, um zwei sortierte Folgen *parallel* in relativ wenigen Schritten zu einer gemeinsamen sortierten Folge zu mischen.

Diese Technik nennt sich "odd-even-merge" (dtsch.: gerade-ungerad-Mischen) und wird ebenfalls rekursiv definiert.

Definition 10.7.6: **odd-even-merge** "Mischen(2m)"

Gegeben: zwei sortierte Folgen (für $m = 2^k$ für eine natürliche Zahl $k \geq 0$) a_1, a_2, \dots, a_m und b_1, b_2, \dots, b_m .

Ergebnis: die zugehörige sortierte Folge z_1, z_2, \dots, z_{2m} .

Vorgehen zur Durchführung von "Mischen (2m)":

$m = 1$: Vergleiche mit dem Baustein "**sort**"; fertig.

$m > 1$:

- (1) Mische rekursiv die ungeraden Glieder der a- und der b-Folge sowie die geraden Glieder der a- und der b-Folge, d.h., mische die sortierten Folgen $a_1, a_3, a_5, \dots, a_{m-1}$ und $b_1, b_3, b_5, \dots, b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3, \dots, u_m$ und mische die sortierten Folgen $a_2, a_4, a_6, \dots, a_m$ und $b_2, b_4, b_6, \dots, b_m$ zur sortierten Folge $v_1, v_2, v_3, \dots, v_m$.
- (2) $z_1 = u_1, z_{2m} = v_m$ und führe parallel $m-1$ Vergleiche durch:
 $z_{2i} = \text{Min}(u_{i+1}, v_i), z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für $i=1, 2, \dots, m-1$.

Beispiel 10.7.7 für $m = 1, 2, 4$.

Wir konstruieren zunächst das Mischen für einige Werte von m .

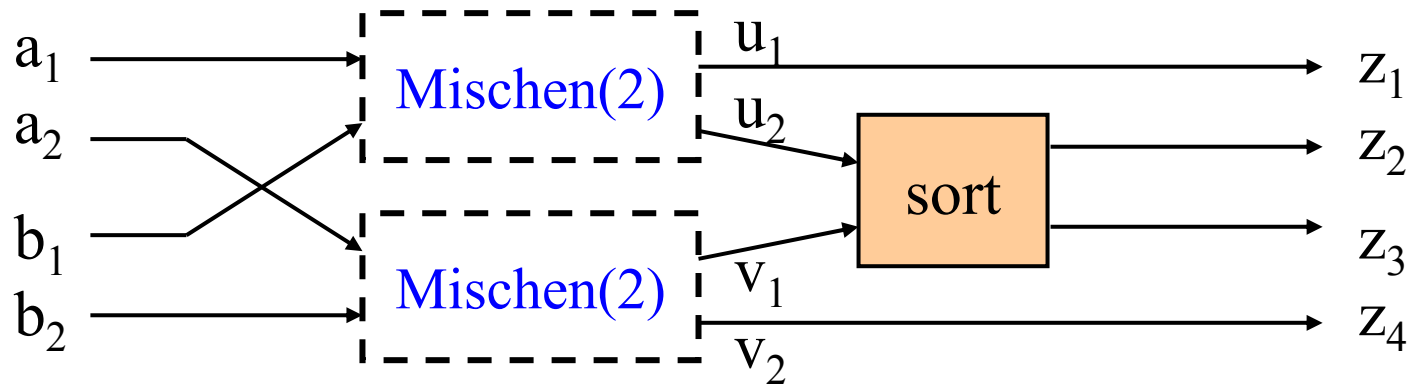
Der Fall $m=1$ benötigt genau einen Sortierbaustein:

$m = 1$:

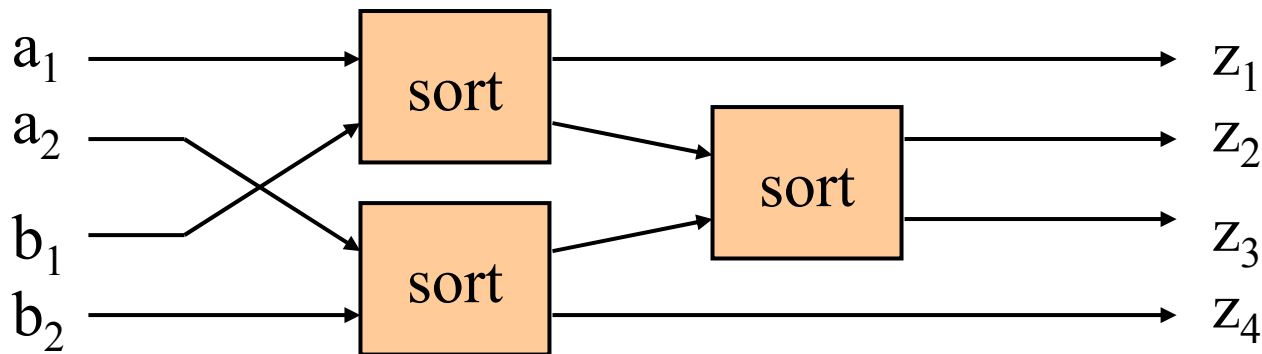


Dies ist der Baustein **Mischen(2)**.

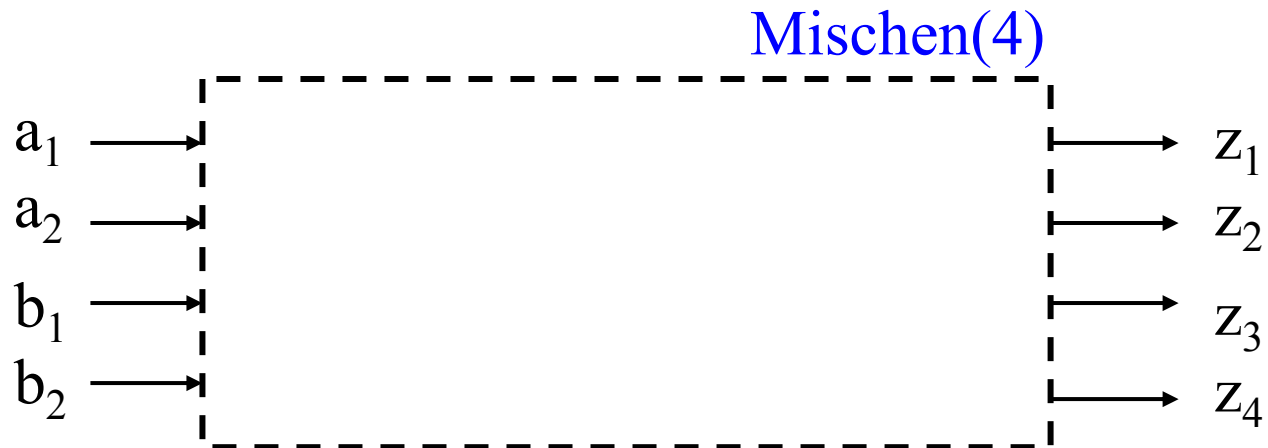
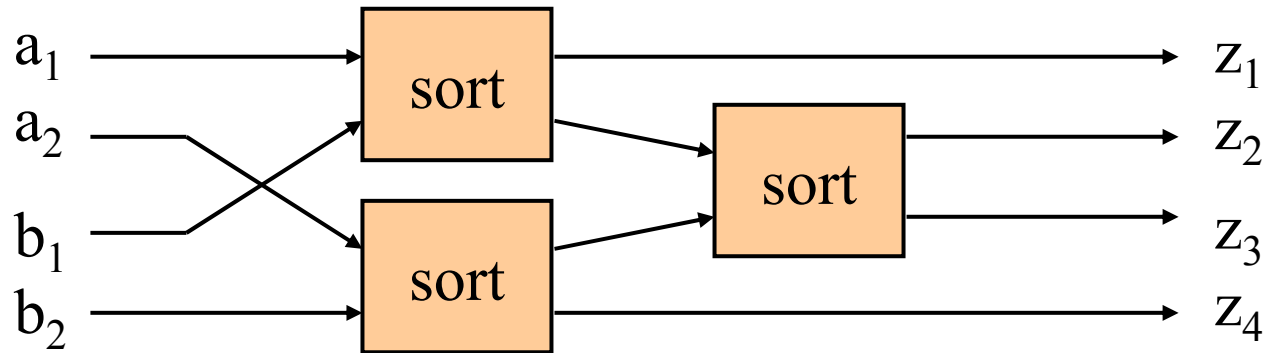
m = 2: (beachte, dass a_1, a_2 bzw. b_1, b_2 sortiert sind.)



Einsetzen von **Mischen(2)** ergibt den Baustein **Mischen(4)**:

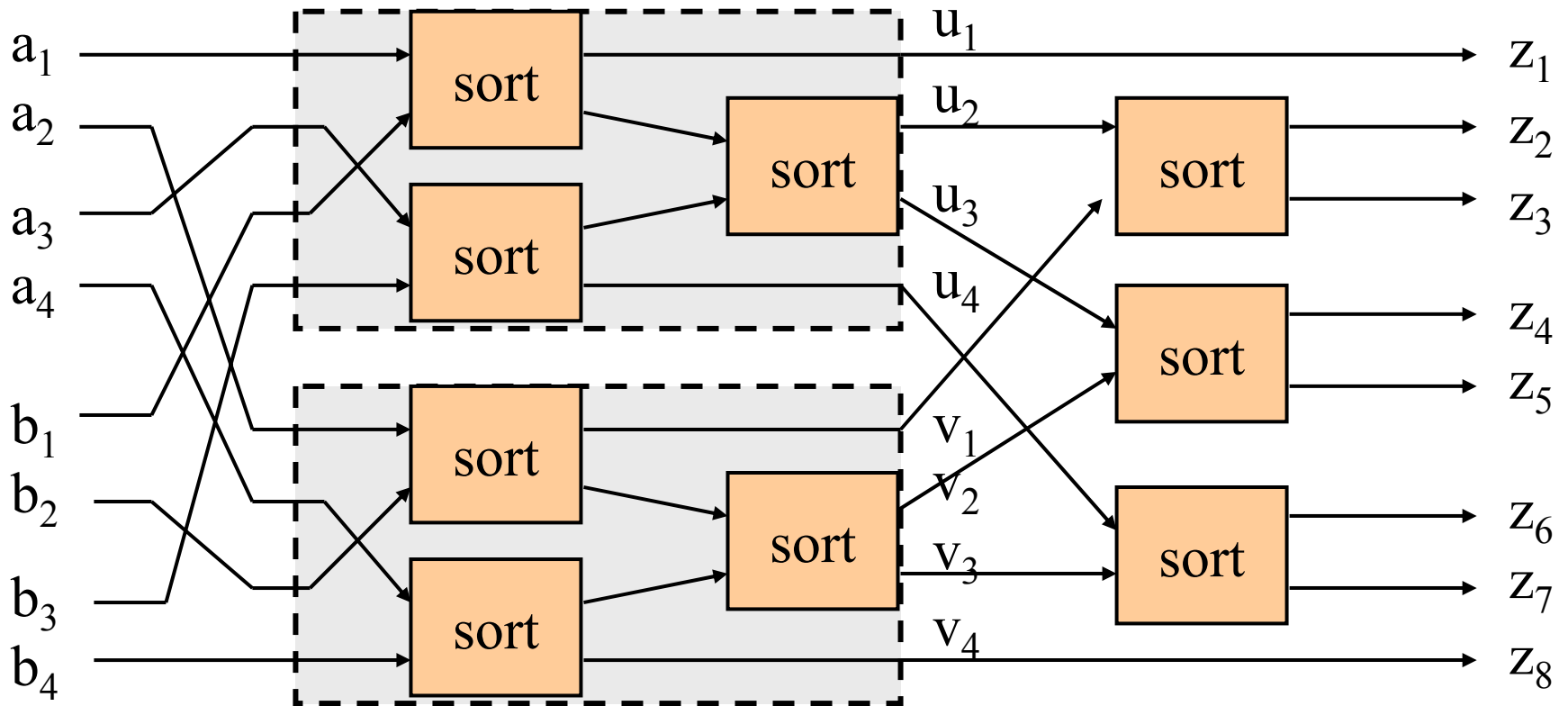


Wir kürzen diesen Baustein durch Mischen(4) ab:



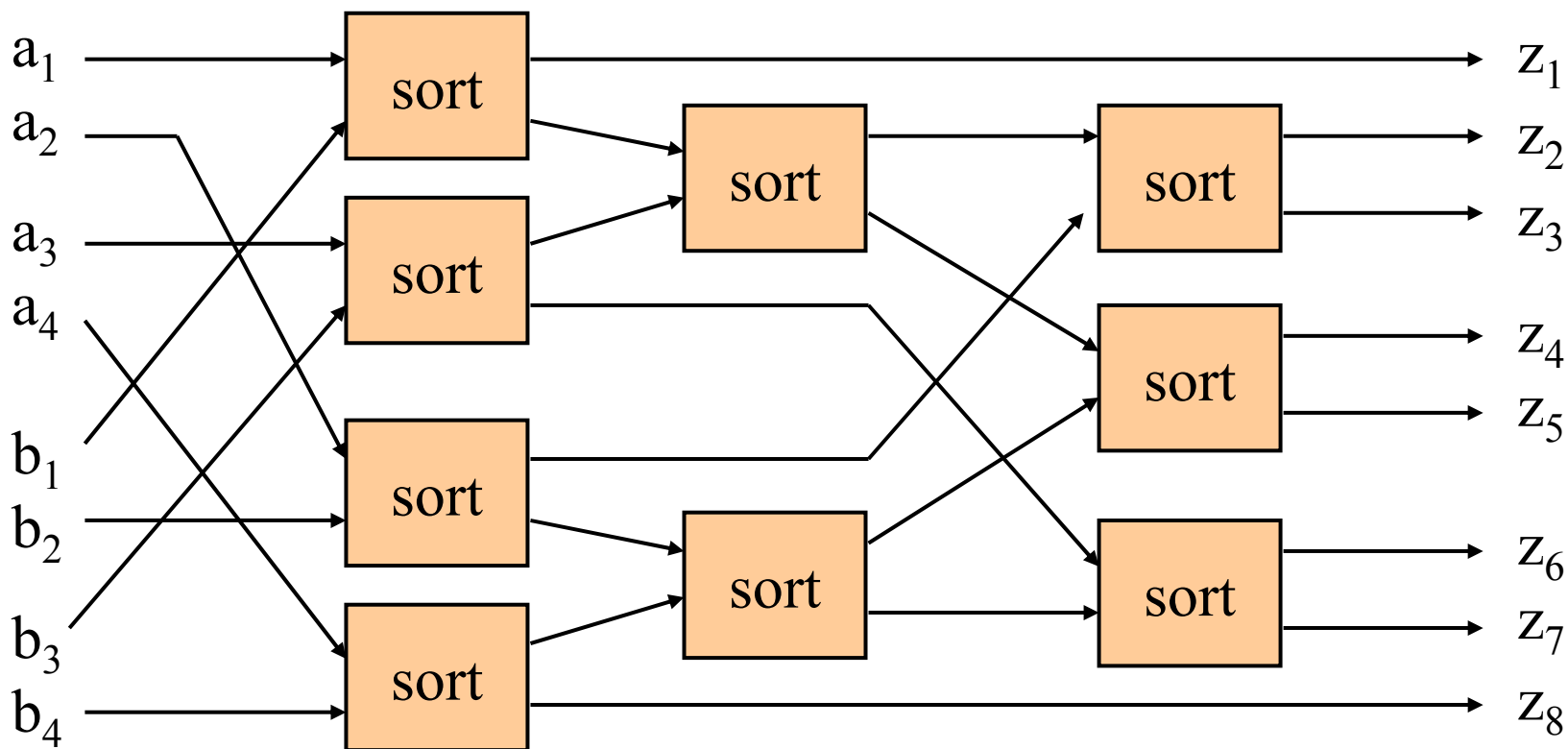
m = 4:

Mischen(4)



Mischen(4)

m = 4: Bereinigung der Verbindungen ergibt:



Beachte: a_1, a_2, a_3, a_4 bzw. b_1, b_2, b_3, b_4 sind sortierte Folgen.

m = 4: Dies kürzen wir erneut ab durch

Mischen(8)

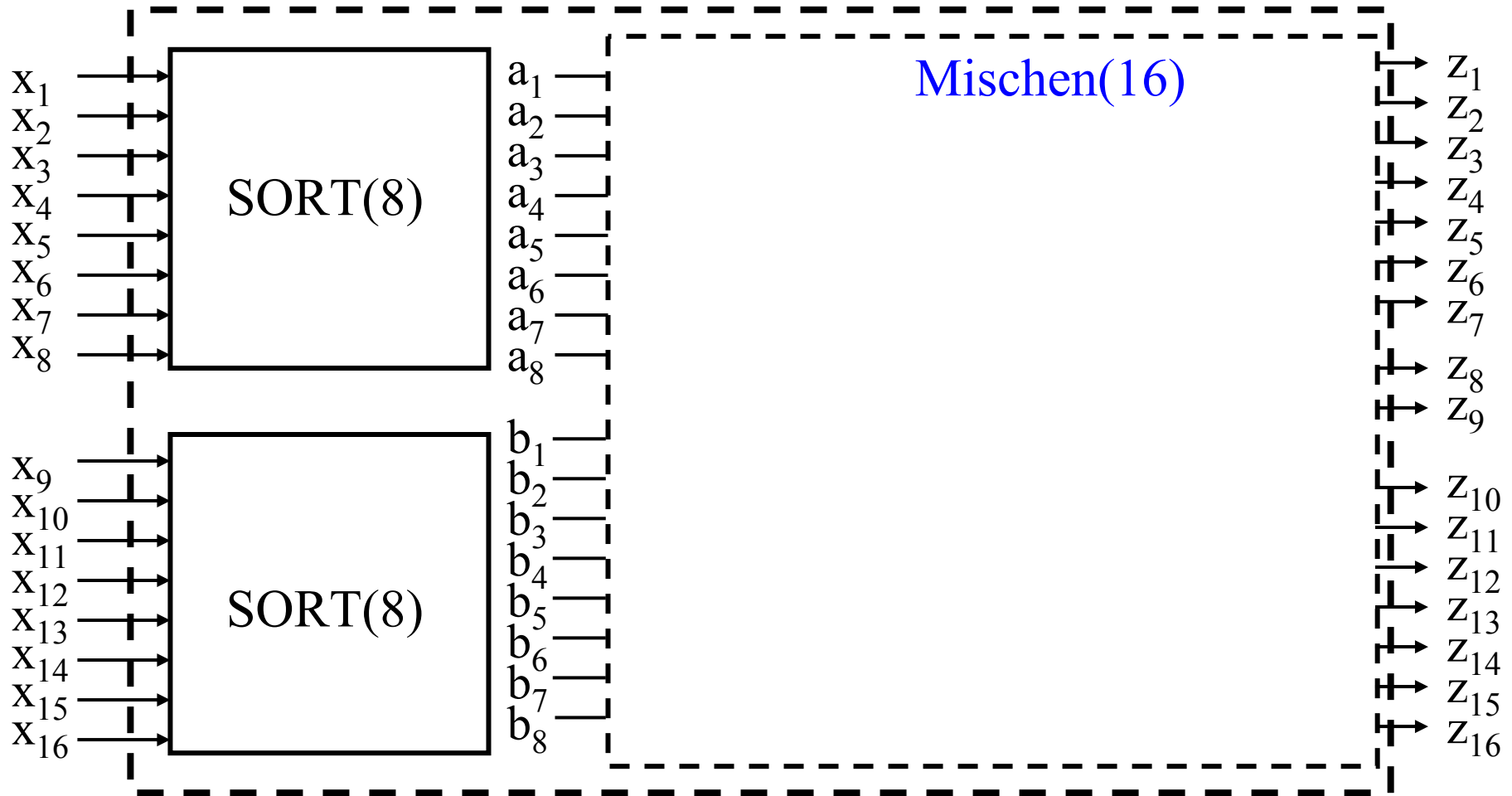


10.7.8 Betrachte nun den Gesamtalgorithmus SORT für $n=16$:

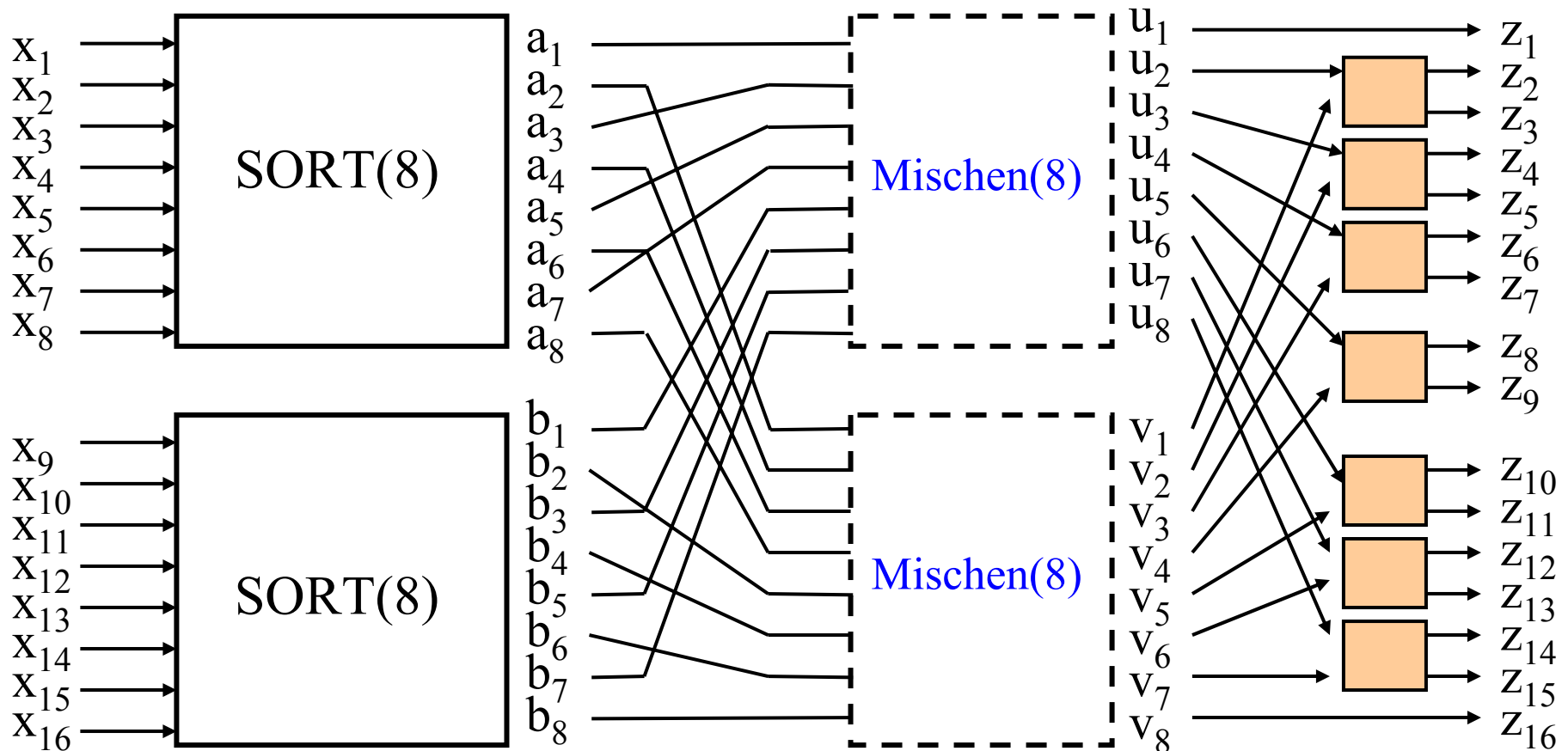
SORT (16)



Rekursives Ersetzen für SORT(16) ergibt:

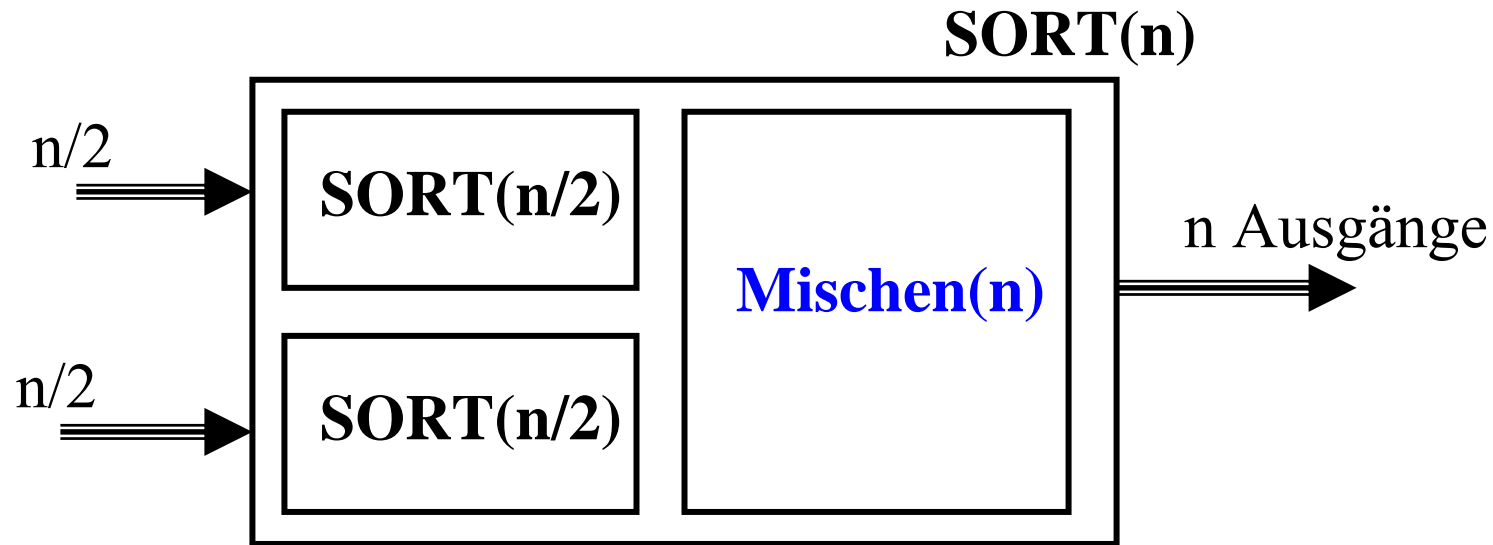


Einsetzen von Mischen(16) ergibt die Struktur:



Setzt man alle kleineren schon konstruierten Bausteine ein, so erhält man den ausführbaren Algorithmus (selbst durchführen, vgl. dann letzte Folie in Kap. 10).

10.7.9 Die rekursive Struktur des parallelen Sortierens:

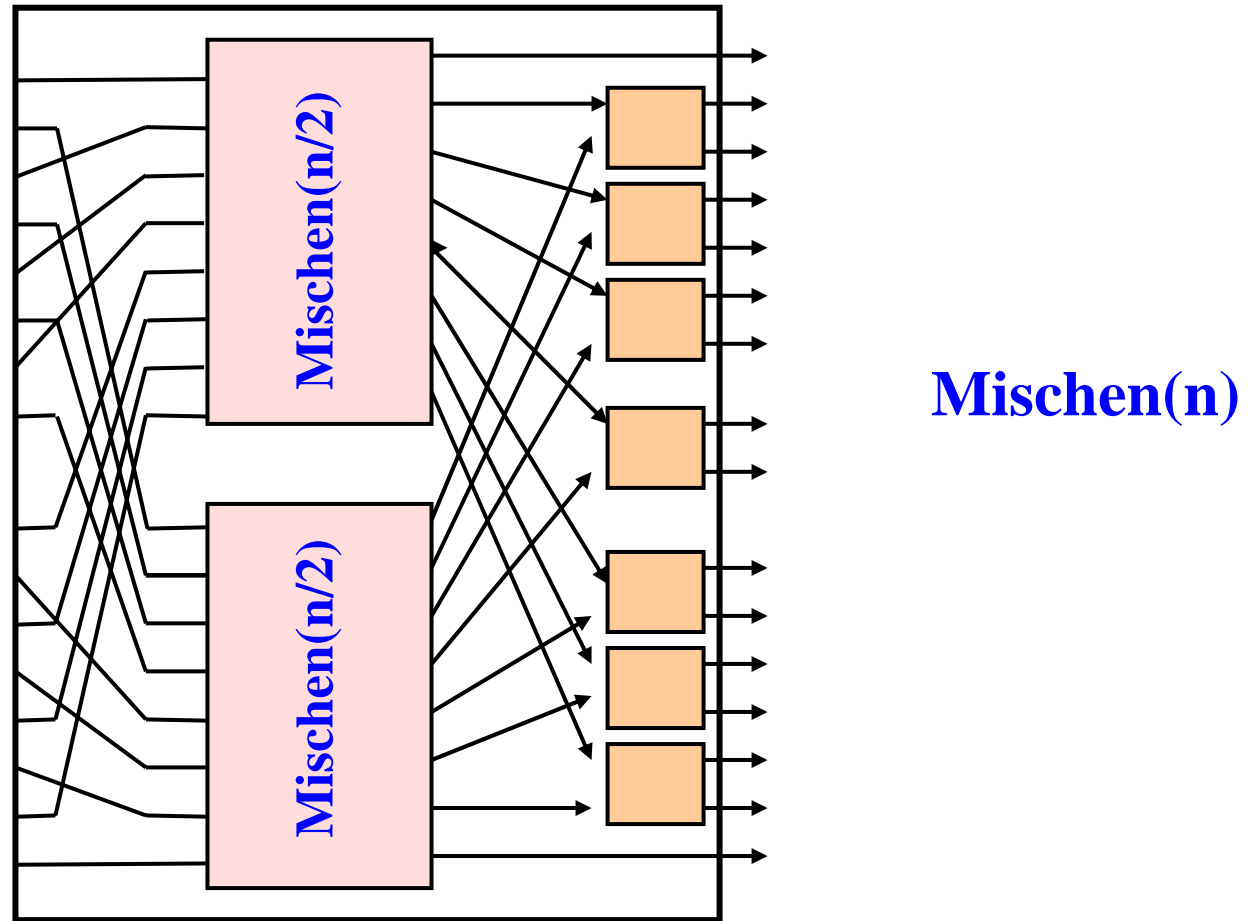


Anzahl $G(n)$ der Bausteine **sort**: $G(2) = 1$ und für $n=2^k$, $k>1$:

$$G(n) = 2 \cdot G(n/2) + M(n),$$

wobei $M(n)$ die Anzahl **sort** in **Mischen(n)** ist.

10.7.10 Mischen(n) ist ebenfalls rekursiv definiert:



Anzahl **M(n)** der Bausteine **sort** im Algorithmus **Mischen(n)**:
 $M(2) = 1$ und für $n > 2$: $M(n) = 2 \cdot M(n/2) + n/2 - 1$.

Nachdem der Aufbau des parallelen Sortieralgorithmus klar ist, müssen wir beweisen, dass die Technik "odd-even-merge" die beiden geordneten Folgen a_1, a_2, \dots, a_m und b_1, b_2, \dots, b_m tatsächlich zu der geordneten Folge $z_1, z_2, \dots, z_{2m-1}, z_{2m}$ zusammenmischt.

Satz 10.7.11:

odd-even-merge sortiert die geordneten Folgen a_1, a_2, \dots, a_m und b_1, b_2, \dots, b_m zur geordneten Folge $z_1, z_2, \dots, z_{2m-1}, z_{2m}$.

Beweis:

Wenn $m=1$ ist, dann werden zwei Zahlen durch den Sortierbaustein **sort** geordnet und der Satz ist richtig.

Sei daher $m > 1$. Gegeben sind zwei sortierte Folgen a_1, a_2, \dots, a_m und b_1, b_2, \dots, b_m . Nach Definition von odd-even-merge werden die sortierten Teilfolgen mit jeweils $m/2$ Elementen $a_1, a_3, a_5, \dots, a_{m-1}$ und $b_1, b_3, b_5, \dots, b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3, \dots, u_m$ bzw. $a_2, a_4, a_6, \dots, a_m$ und $b_2, b_4, b_6, \dots, b_m$ zur sortierten Folge $v_1, v_2, v_3, \dots, v_m$ rekursiv gemischt.

Die Ergebnisfolge ist definiert durch $z_1 = u_1, z_{2m} = v_m$ und $z_{2i} = \text{Min}(u_{i+1}, v_i), z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für $i=1, 2, \dots, m-1$. Wir zeigen, dass die z -Folge hierdurch korrekt sortiert ist.

u_1 muss das Minimum der beiden Folgen sein, da a_1 und b_1 die Minima der a - bzw. b -Folge und beide Elemente in der u -Folge sind. Analog gilt, dass v_m das Maximum der Ergebnisfolge sein muss. Also sind z_1 und z_{2m} richtig bestimmt worden.

Um zu zeigen, dass $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ richtig festgelegt wurden, genügt es zu zeigen, dass sich das Element u_{i+1} in der sortierten Ergebnisfolge an der Position $2i$ oder $2i+1$ befinden muss (analog muss man dies für das Element v_i nachweisen). Wir beweisen dies in acht Schritten (a) bis (h).

Wir betrachten u_{i+1} für ein i zwischen 1 und $m-1$. O.B.d.A. nehmen wir an, dass dieses Element aus der a -Folge stammt und das j -te Element der Teilfolge mit den ungeraden Indizes ist, d.h., es gibt ein j mit $1 \leq j \leq m/2$ mit $u_{i+1} = a_{2j-1}$.

(a) Wie viele der Elemente der a -Folge sind kleiner als u_{i+1} ?

Genau $2j-2$ Elemente;

denn weil $u_{i+1} = a_{2j-1}$ ist, müssen die Elemente $a_1, a_2, \dots, a_{2j-2}$ der geordneten a -Folge kleiner als u_{i+1} sein.

(b) Wie viele der Elemente der b-Folge sind kleiner als u_{i+1} ?

Mindestens $2i-2j+1$ Elemente. Beweis hierfür:

Wegen $u_{i+1} = a_{2j-1}$ und wegen $a_1 \leq a_3 \leq a_5 \leq \dots \leq a_{2j-3}$ müssen sich unter den ersten i Elementen u_1, u_2, \dots, u_i der u-Folge genau diese $(j-1)$ Elemente aus der a-Folge befinden. Folglich müssen unter diesen ersten i Elementen der u-Folge genau $i-(j-1) = i-j+1$ Elemente der b-Folge sein. Da in der u-Folge aber nur die Elemente mit ungeradem Index sind, müssen dies genau die Elemente $b_1, b_3, b_5, \dots, b_{2(i-j+1)-1}$ sein. Da die b-Folge sortiert ist, müssen daher mindestens die Elemente $b_1, b_2, b_3, \dots, b_{2(i-j+1)-1}$ kleiner als u_{i+1} sein. Ihre Anzahl ist $2i-2j+1$.

(c) Folglich stehen in der sortierten Ergebnisfolge mindestens

$2j-2 + 2i-2j+1 = 2i-1$ Elemente vor u_{i+1} , d.h., in der Ergebnisfolge kann u_{i+1} frühestens das Element z_{2i} sein.

(d) Wie viele der Elemente der a-Folge sind größer als u_{i+1} ?

Genau $m-2j+1$ Elemente,

denn wegen $u_{i+1} = a_{2j-1}$ müssen $a_{2j}, a_{2j+1}, \dots, a_m$ größer als u_{i+1} sein.

(e) Wie viele der Elemente der b-Folge sind größer als u_{i+1} ?

Mindestens $m-2i+2j-2$ Elemente. Beweis hierfür:

Wegen (b) muss $u_{i+1} \leq b_{2(i-j+1)+1}$ sein; denn sonst wäre u_{i+1} nicht das $(i+1)$ -te, sondern ein späteres Element der u-Folge. Also müssen mindestens die Elemente $b_{2(i-j+1)+1}, b_{2(i-j+1)+2}, \dots, b_m$ größer als u_{i+1} sein. Ihre Anzahl ist $m - (2i-2j+3) + 1 = m-2i+2j-2$.

(f) Folglich stehen in der sortierten Ergebnisfolge mindestens

$m-2j+1 + m-2i+2j-2 = 2m-2i-1$ Elemente hinter u_{i+1} , d.h., in der sortierten Ergebnisfolge muss u_{i+1} spätestens das Element $z_{2m-(2m-2i-1)} = z_{2i+1}$ sein.

(g) Nun führe man genau die gleiche Untersuchung für v_i durch. Auch hier erhält man, dass v_i frühestens das Element z_{2i} und spätestens das Element z_{2i+1} in der Ergebnisfolge sein kann. *(Führen Sie diesen Beweis selbst analog zu (a) bis (f).)*

(h) Aus (f) und (g) folgt nun, dass die Elemente u_{i+1} und v_i sich an den Positionen $2i$ und $2i+1$ der z -Folge befinden müssen. Daher muss $z_{2i} = \min(u_{i+1}, v_i)$, $z_{2i+1} = \max(u_{i+1}, v_i)$ für die sortierte Ergebnisfolge z_1, z_2, \dots, z_{2m} gelten.

Damit ist der Satz bewiesen.

Skizze zum Beweis: O.B.d.A. sei $u_{i+1} = a_{2j-1}$.

Es sind kleiner als u_{i+1}

Es sind größer als u_{i+1}

$$\begin{array}{ccc}
 a_1 \leq a_2 \leq \dots \leq a_{2j-2} & & a_{2j} \leq a_{2j+1} \leq \dots \leq a_m \\
 & \searrow \quad \swarrow & \\
 & \leq u_{i+1} \leq & \\
 & \swarrow \quad \searrow & \\
 b_1 \leq b_2 \leq \dots \leq b_{2(i-j+1)-1} & & b_{2(i-j+1)+1} \leq b_{2(i-j+1)+2} \leq \dots \leq b_m
 \end{array}$$

$$z_1, z_2, \dots, z_{2i-2}, z_{2i-1} \qquad z_{2i} \qquad z_{2i+1} \qquad z_{2i+2}, z_{2i+3}, \dots, z_{2m}$$

[Wenn $u_{i+1} = a_{2j-1}$ ist, so ist $v_i = b_{2(i-j+1)}$. Beide müssen an den Positionen $2i$ oder $2i+1$ in der sortierten z -Ergebnisfolge stehen. Es ist nur noch zu prüfen, ob a_{2j-1} kleiner oder größer als $b_{2(i-j+1)}$ ist. Folglich gilt $z_{2i} = \min(u_{i+1}, v_i)$, $z_{2i+1} = \max(u_{i+1}, v_i)$.]

10.7.12 Zur Komplexität (Größe, Tiefe, Breite):

Wie groß, wie breit und wie tief ist dieser parallele Sortieralgorithmus?

"Größe" soll ein Maß für die "Herstellungskosten" sein, wenn man den Algorithmus hardwaremäßig realisiert:

$G(n)$ = Anzahl der Bausteine **sort** in **SORT(n)**.

"Tiefe" soll die Laufzeit des Sortierens angeben:

$T(n)$ = Länge des längsten gerichteten Weges durch Bausteine **sort** in **SORT(n)**.

"Breite" soll die Anzahl der Mikroprozessoren angeben, die für eine Softwarelösung benötigt werden:

$B(n)$ = Minimale Zahl der Bausteine **sort**, die parallel zueinander liegen müssen, ohne die Tiefe zu verändern.

Wir gehen nun von 10.7.9 und 10.7.10 aus:

Anzahl $G(n)$ der Bausteine **sort**: $G(2) = 1$ und für $n=2^k$, $k>1$:

$$G(n) = 2 \cdot G(n/2) + M(n),$$

wobei $M(n)$ die Anzahl **sort** in $Mischen(n)$ ist:

$$M(2) = 1 \text{ und für } n=2^k, k>1: M(n) = 2 \cdot M(n/2) + n/2 - 1.$$

Für die Tiefe gilt: $T(n) = T(n/2) + \text{"Tiefe von Mischen(n)"}$.

Die Breite beträgt $B(n) = n/2$. Denn von jedem Eingang x_i muss ein längster Weg ausgehen und die Anzahl der Bausteine, die unmittelbar hinter den Eingängen liegen, beträgt $n/2$. Im weiteren Verlauf der Rekursion kommt man mit dieser Zahl auch aus, siehe die Formeln für $SORT(n)$ und für $Mischen(n)$. Dies lässt sich auch formal beweisen, worauf wir hier verzichten.

Einige Größen der Mischen-Bausteine: $M(2) = 1$, $M(4) = 3$,
 $M(8) = 9$, $M(16) = 25$, $M(32) = 65$. Aus $M(n) = 2 \cdot M(n/2) + n/2 - 1$
gewinnt man durch Einsetzen rasch die Gleichungen:

$$\begin{aligned} M(n) &= 2 \cdot M(n/2) + n/2 - 1 \\ &= 2 \cdot (2 \cdot M(n/4) + n/4 - 1) + n/2 - 1 \\ &= 4 \cdot M(n/4) + 2 \cdot n/2 - 1 - 2 \\ &= 4 \cdot (2 \cdot M(n/8) + n/8 - 1) + 2 \cdot n/2 - 1 - 2 \\ &= 8 \cdot M(n/8) + 3 \cdot n/2 - 1 - 2 - 4 \\ &= \dots \\ &= 2^{\log(n)-1} \cdot M(2) + (\log(n)-1) \cdot n/2 - (2^{\log(n)-1} - 1) \\ &= n/2 + n/2 \cdot \log(n) - n/2 - n/2 + 1 \\ &= n/2 \cdot (\log(n) - 1) + 1 \end{aligned}$$

Es gilt also $M(n) = n/2 \cdot (\log(n) - 1) + 1$.

(Vgl. die ähnliche Gleichung in 10.5.4 für $V(n)$.)

Hiermit können wir nun die "Größe" $G(n)$ ausrechnen:

$$\begin{aligned} G(n) &= 2 \cdot G(n/2) + M(n) \\ &= 2 \cdot G(n/2) + n/2 \cdot (\log(n) - 1) + 1 \\ &= 2 \cdot (2 \cdot G(n/4) + n/4 \cdot (\log(n/2) - 1) + 1) + n/2 \cdot (\log(n) - 1) + 1 \\ &= 4 \cdot G(n/4) + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) + 1 + 2 \\ &= 4 \cdot (2 \cdot G(n/8) + n/8 \cdot (\log(n/4) - 1) + 1) \\ &\quad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) + 1 + 2 \\ &= 8 \cdot G(n/8) + n/2 \cdot (\log(n) - 3) \\ &\quad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) + 1 + 2 + 4 \\ &= \dots \\ &= 2^{\log(n)-1} \cdot G(2) + n/2 \cdot (1+2+\dots+(\log(n) - 1)) + 1+2+\dots+2^{\log(n)-2} \\ &= 2^{\log(n)-1} + n/4 \cdot \log(n) \cdot (\log(n)-1) + 2^{\log(n)-1} - 1 \\ &= (n/4) \cdot \log(n) \cdot (\log(n)-1) + n - 1. \end{aligned}$$

Ergebnis: **$G(n) = (n/4) \cdot \log(n) \cdot (\log(n)-1) + n - 1 \in O(n \cdot \log^2(n))$** .

Entscheidend für den praktischen Einsatz ist die Tiefe $T(n)$.

Hierzu betrachten wir 10.7.10:

Die **Tiefe von Mischen(n)** ist die **Tiefe von Mischen($n/2$) + 1**,
woraus sofort die Tiefe $\log(n)$ für den Teil Mischen(n) folgt.

Nach Folie 118 gilt dann:

$$\begin{aligned} T(n) &= T(n/2) + \log(n) \\ &= T(n/4) + \log(n/2) + \log(n) = T(n/4) + \log(n)-1 + \log(n) \\ &= T(n/8) + \log(n/4) + \log(n/2) + \log(n) \\ &= \dots \\ &= T(2) + \log(n/2^{\log(n)-2}) + \log(n/2^{\log(n)-3}) + \dots + \log(n)-1 + \log(n) \\ &= 1 + 2 + 3 + \dots + \log(n) \\ &= (1/2) \cdot \log(n) \cdot (\log(n) + 1). \end{aligned}$$

Ergebnis: **$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1) \in O(\log^2(n))$** .

10.7.13 Resultat:

$$G(n) = (n/4) \cdot \log(n) \cdot (\log(n) - 1) + n - 1$$

$$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1)$$

Faustformel:

$$G(n) \approx n/2 \cdot T(n).$$

n Eingänge	G(n) Größe	T(n) Tiefe
2	1	1
4	5	3
8	19	6
16	63	10
32	191	15
64	543	21
128	1471	28
256	3839	36
512	9727	45
1024	24063	55

n Eingänge	G(n) Größe	T(n) Tiefe
16	63	10
128	1471	28
1024	24063	55
16384	1490957	105

Mit 24.063 Bausteinen **sort** kann man also in 55 Schritten 1024 Zahlen sortieren.

Mit etwa 100 Millionen Bausteinen **sort** kann man in nur 210 Schritten rund 1 Million Zahlen sortieren.

Solche Algorithmen sind technisch durchaus realisierbar. Sie können in Zukunft die Leistungsfähigkeit beim Sortieren deutlich steigern.

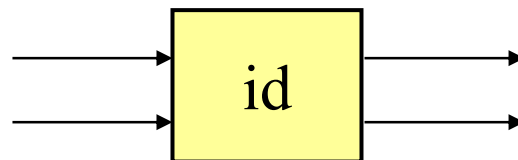
10.7.14 Durchdenken Sie eine mögliche Realisierung weiter, z.B.:

- Gibt es Probleme bei der Hardware-Realisierung? Lassen sich die vielen Leitungen problemlos verschalten / auf Platten drucken? ... Jede Leitung in unserer Skizze besitzt eine "Breite", z.B. 64 Bits zuzüglich Kontrollbits.
- Wie kann man den Sortierbaustein **sort** realisieren? Nehmen Sie an, dass die zu sortierenden Schlüssel k -stellige 0-1-Folgen sind, wobei man wegen der vielfältig möglichen Schlüssel von $k=64$ ausgehen sollte. Was ist dann die minimale Tiefe (=längster Weg über Gatter von einem Eingang zu einem Ausgang) von **sort**?
- Wie kann man Millionen von Daten *gleichzeitig* an alle Eingänge legen? Welche Strukturen müssen die Speicher hierfür besitzen?

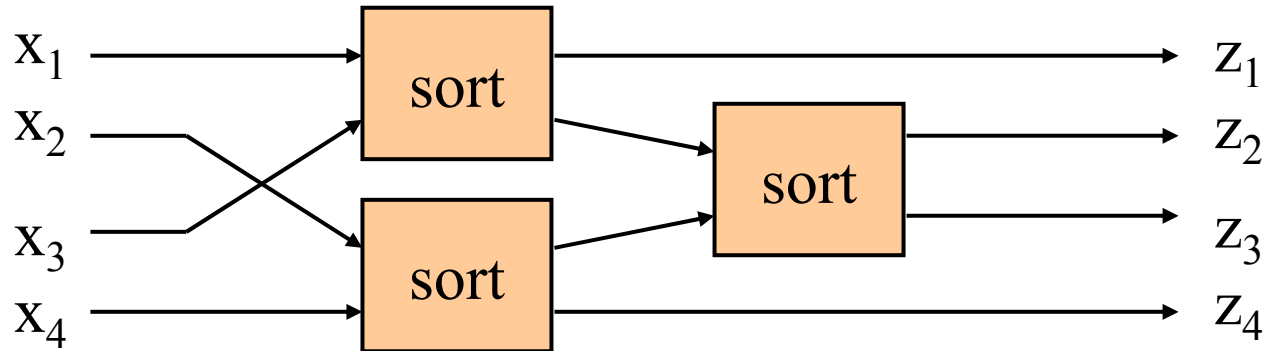
10.7.15 Wie sehen softwaremäßige Realisierungen aus?

Bei n Eingabewerten kann man $n/2$ Prozeduren definieren, die jeweils "den nächsten Gesamtschritt" (= alle parallel durchführbaren Sortierschritte) ausführen. Hierfür greifen sie immer wieder auf das Feld der zu sortierenden Daten x : array(1.. n) of \langle Elementtyp \rangle zu, aus dem in jedem Schritt gelesen und in das in jedem Schritt geschrieben wird.

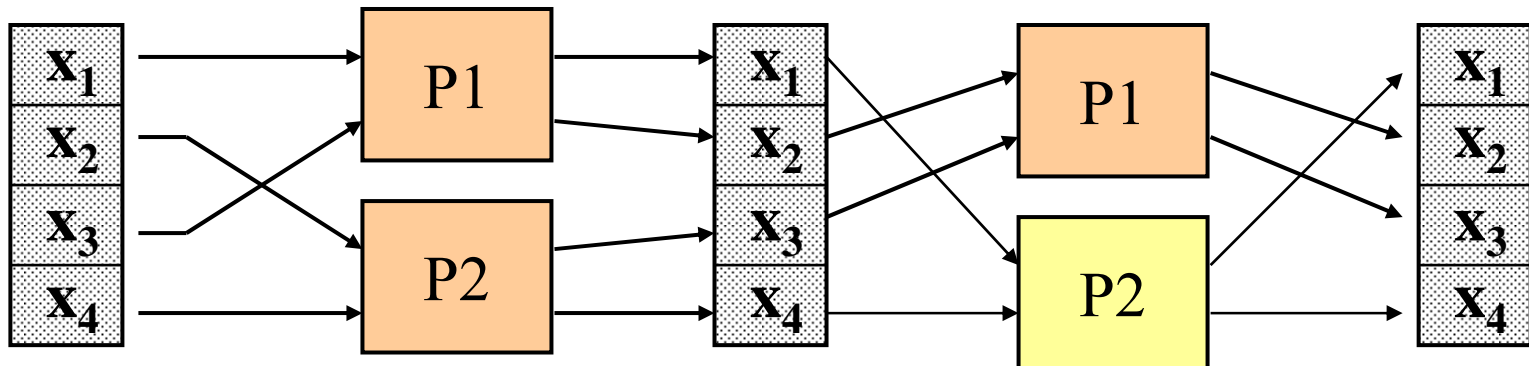
Im Algorithmus gibt es Teile, in denen ein Wert nur weitergereicht wird. Hierfür führen wir den zusätzlichen Baustein "id" (Identität) ein, der zwei Werte einliest und unverändert wieder ausgibt:

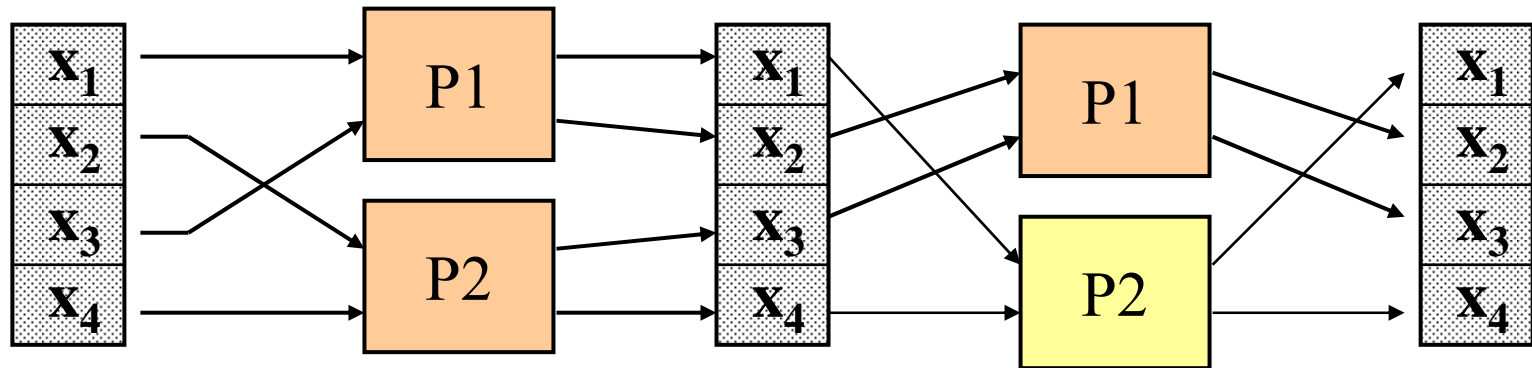


Der Mischbaustein Mischen (4) (siehe 10.7.4)



wird dann realisiert durch zwei Prozeduren P1 und P2:





procedure P1 is

-- x: array(1..n) of <Elementtyp> ist global

h1, h2: <Elementtyp>;

begin (h1, h2) := **sort** (x(1), x(3)); x(1) := h1; x(2) := h2; **\$**

(h1, h2) := **sort** (x(2), x(3)); x(2) := h1; x(3) := h2;

end;

procedure P2 is

-- x: array(1..n) of <Elementtyp> ist global

h1, h2: <Elementtyp>;

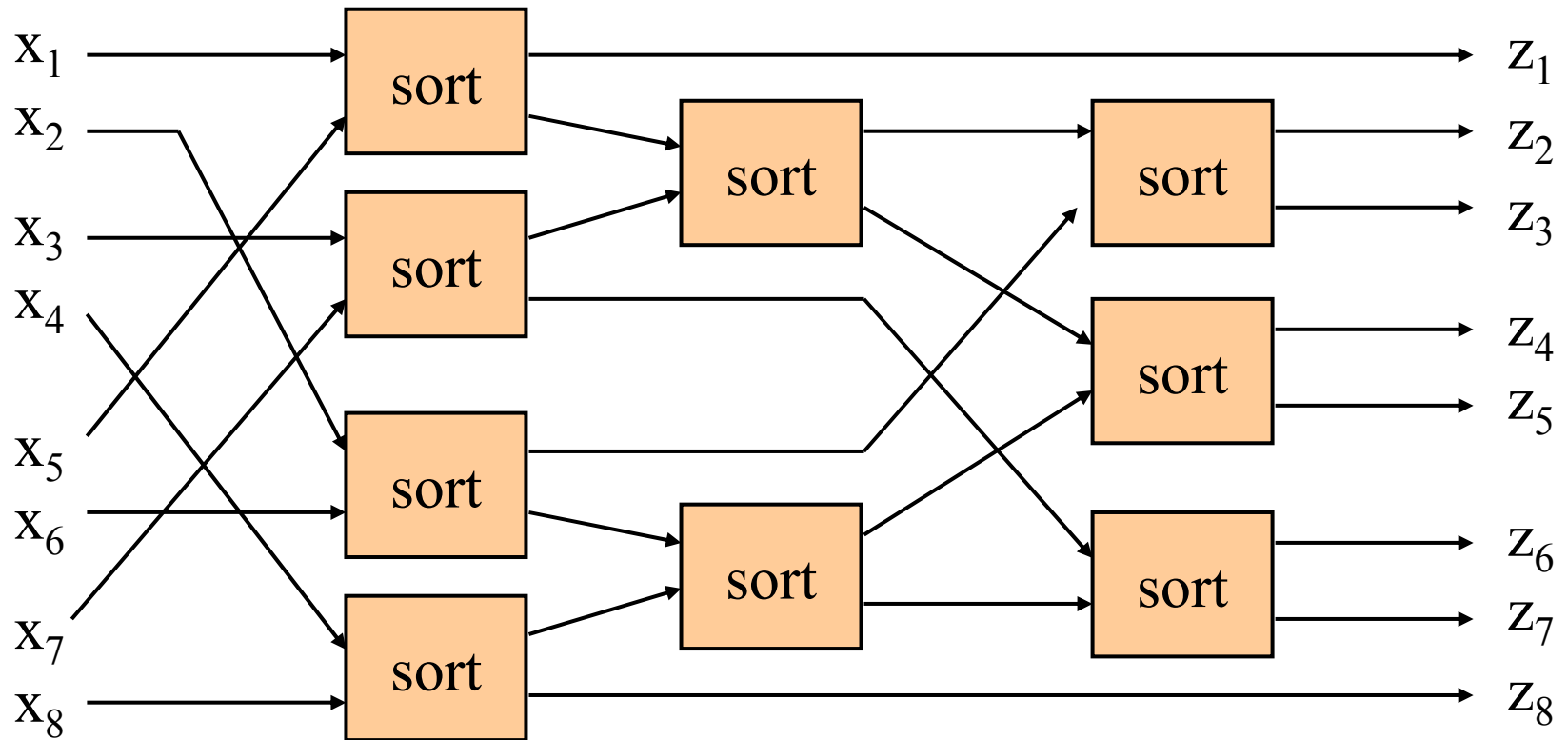
begin (h1, h2) := **sort** (x(2), x(4)); x(3) := h1; x(4) := h2; **\$**

(h1, h2) := **id** (x(1), x(4)); x(1) := h1; x(4) := h2;

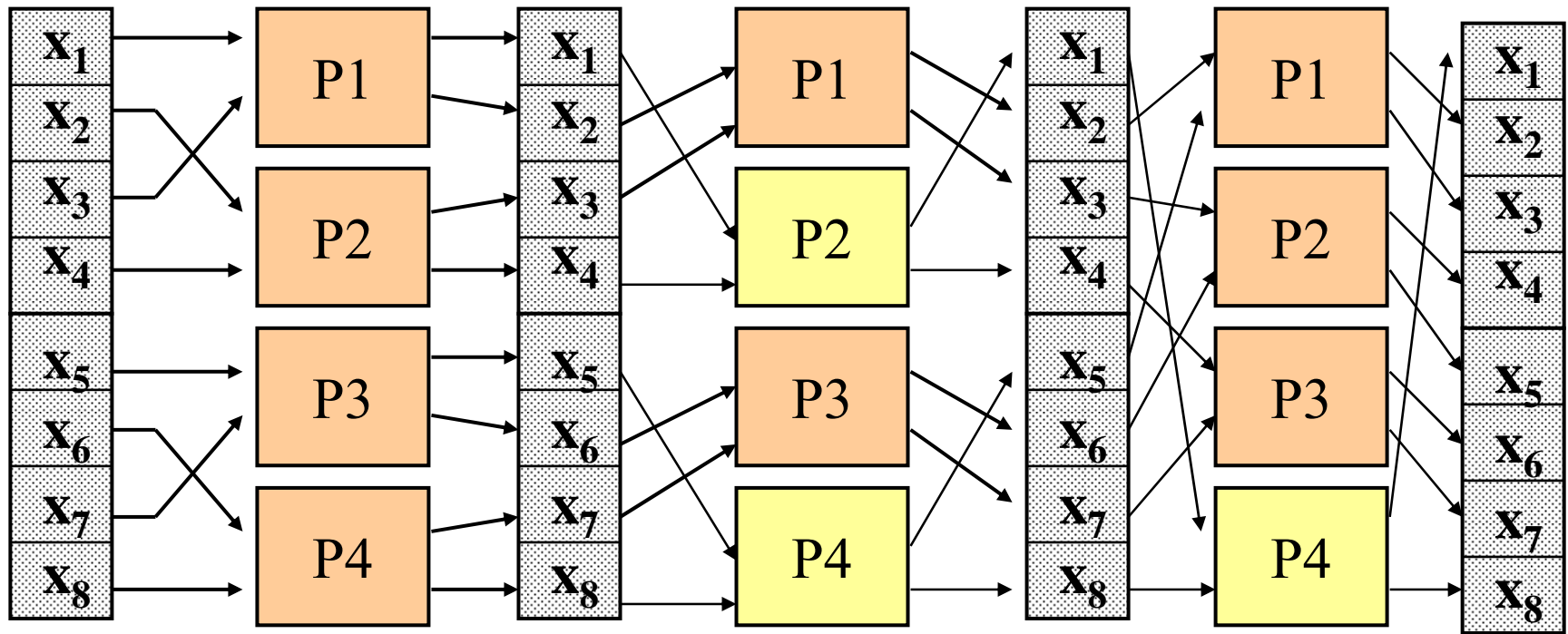
end;

\$ steht für "Synchronisation"

Der Mischbaustein Mischen(8)



wird dann realisiert durch vier Prozeduren, die ihre Arbeitsweise synchronisieren müssen:



procedure Mischen8 is

procedure P1 is begin ... end;

procedure P2 is begin ... end;

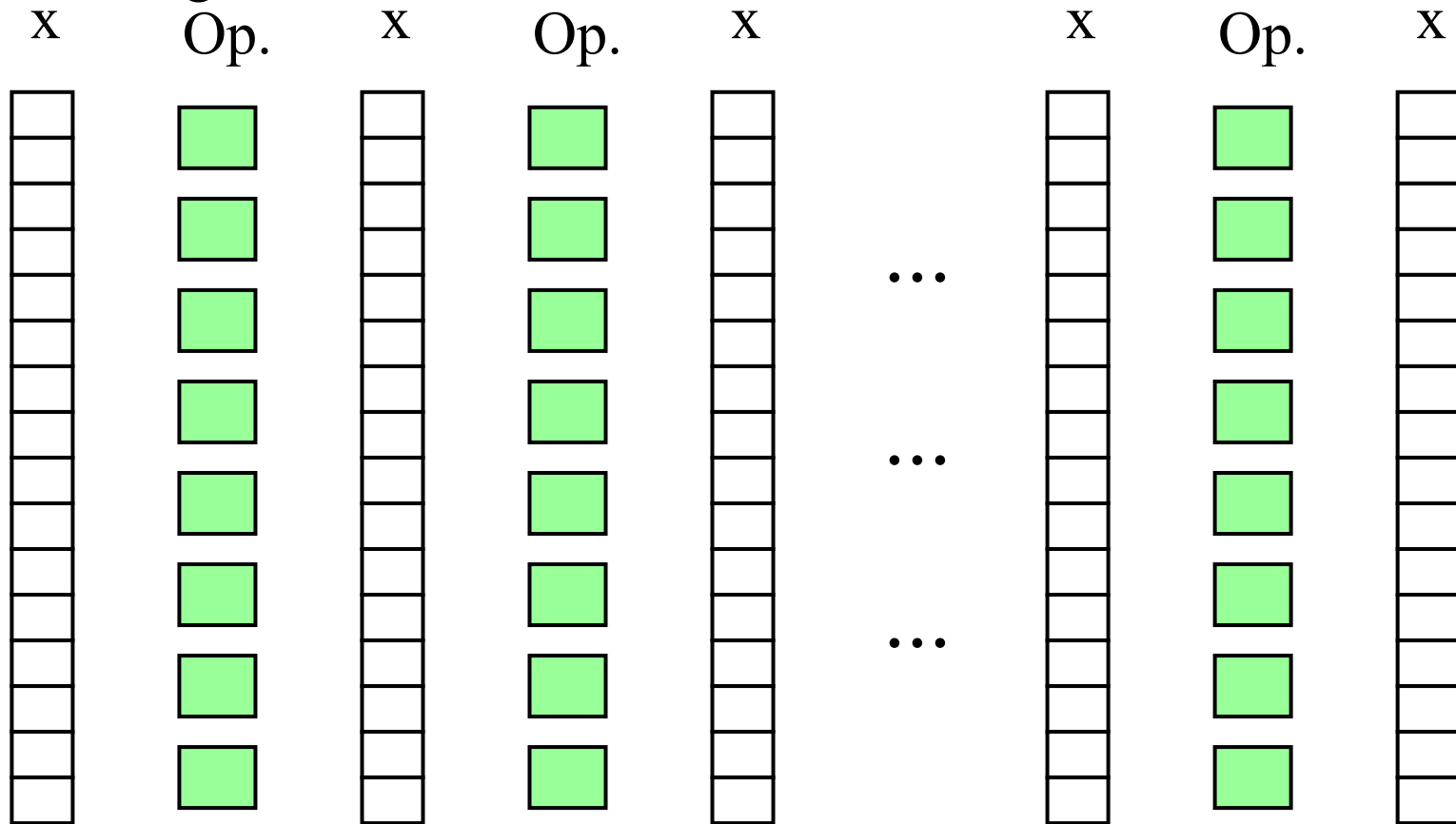
procedure P3 is begin ... end;

procedure P4 is begin ... end;

begin P1; P2; P3; P4; end;

*Die Synchronisation
ist innerhalb der
Prozeduren P1 bis P4
sicherzustellen.*

10.7.16: Man erhält auf diese Weise eine "normierte Darstellung":



x = Datenschicht, Op. = Operationsschicht. Jeder Operationsbaustein **sort** oder **id** hat zwei einlaufende und zwei ausgehende Verbindungen.

Jede Operationsschicht lässt sich durch $n/2$ Prozeduren softwaremäßig beschreiben; diese Prozeduren können wir ebenfalls für die nächste Operationsschicht verwenden usw., so dass wir aus Software-Sicht folgendes Ergebnis erhalten (die Operationen **id** kann man natürlich im Programm weglassen, nicht aber die Synchronisierung):

Mit $n/2$ Prozeduren lassen sich n Elemente in

$$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1)$$

Schritten sortieren, wobei jede der $n/2$ Prozeduren aus einem sequentiellen Programmstück mit bis zu $T(n)$ einzelnen Sortieroperationen **sort** besteht und $T(n)-1$ Synchronisationsanweisungen besitzt.

Diese Prozeduren können automatisch durch ein Programm erzeugt werden; sie hängen nur vom Parameter n ab.

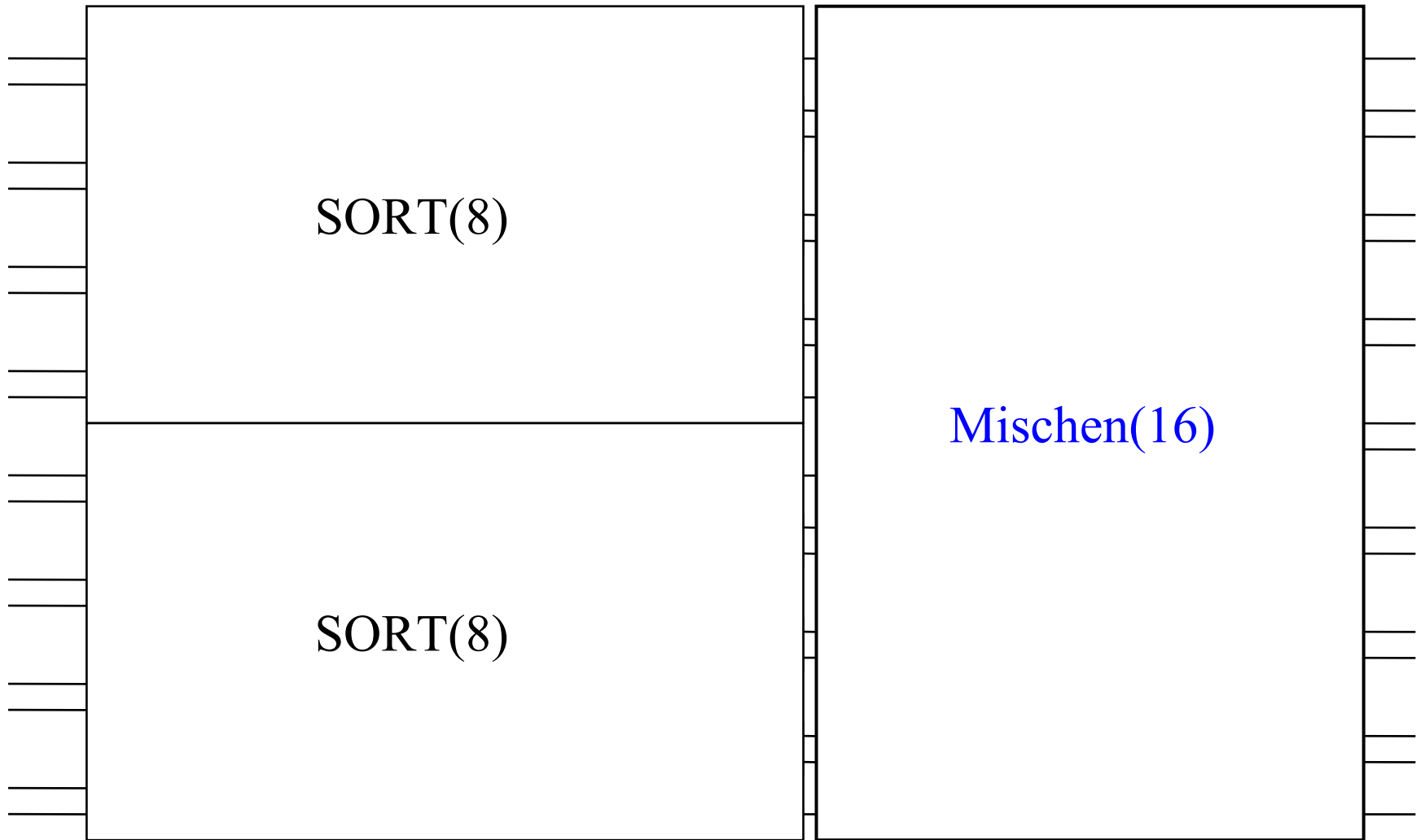
Wie müssen Programmiersprachen beschaffen sein, damit der synchrone Ablauf und die Verzögerungen gesichert werden?

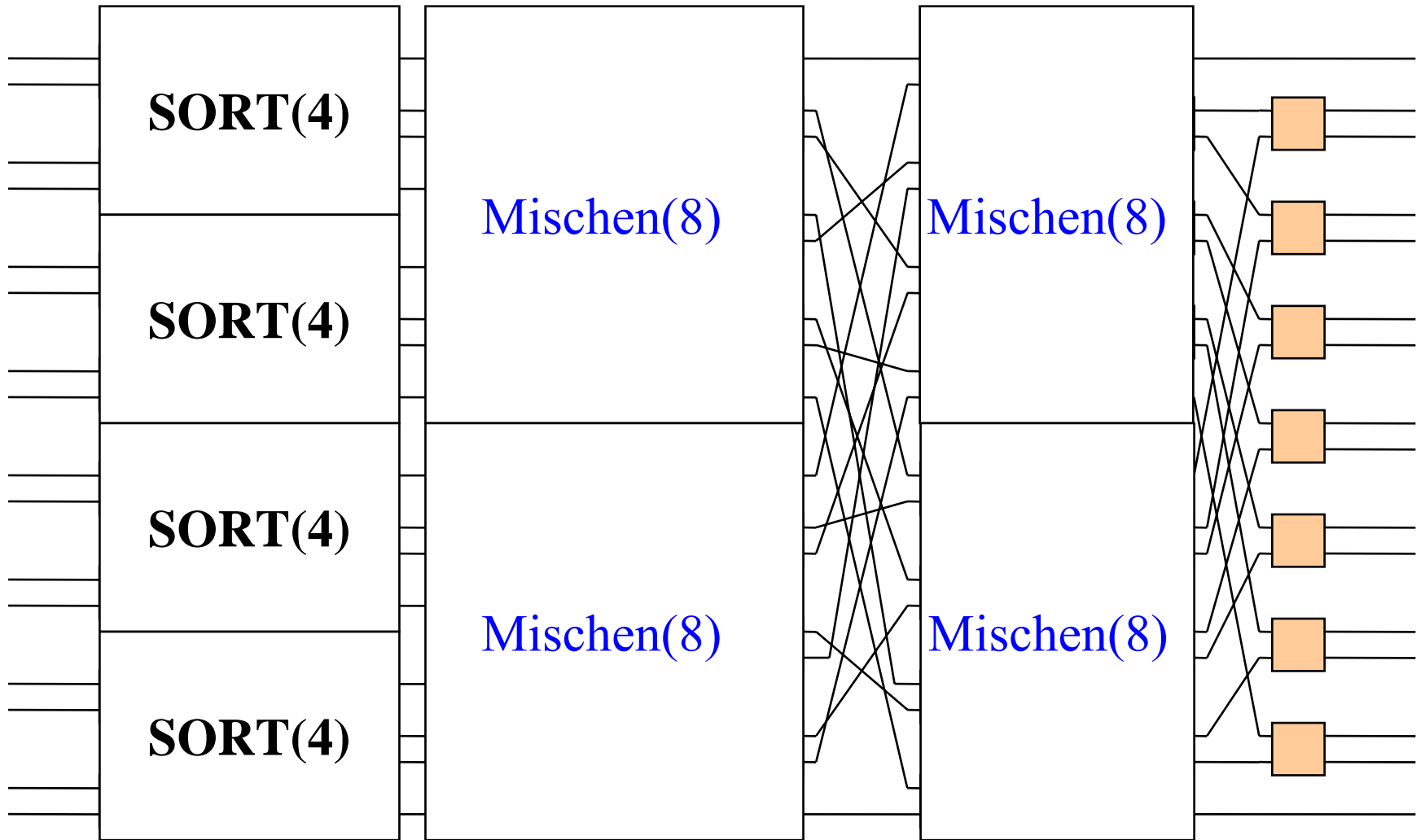
Diese Sprachen müssen in der Regel mehrere Anforderungen erfüllen:

- Teile eines Programms müssen unabhängig voneinander ablaufen können (Nebenläufigkeit, "tasks").
- Es muss Synchronisationsmechanismen geben.
- Man muss Zeitvorgaben (innerhalb von ... Mikrosekunden führe durch) machen können und es muss Verzögerungselemente (delay) geben.
- Es muss der parallele Zugriff auf Daten möglich sein.
- Es muss ...

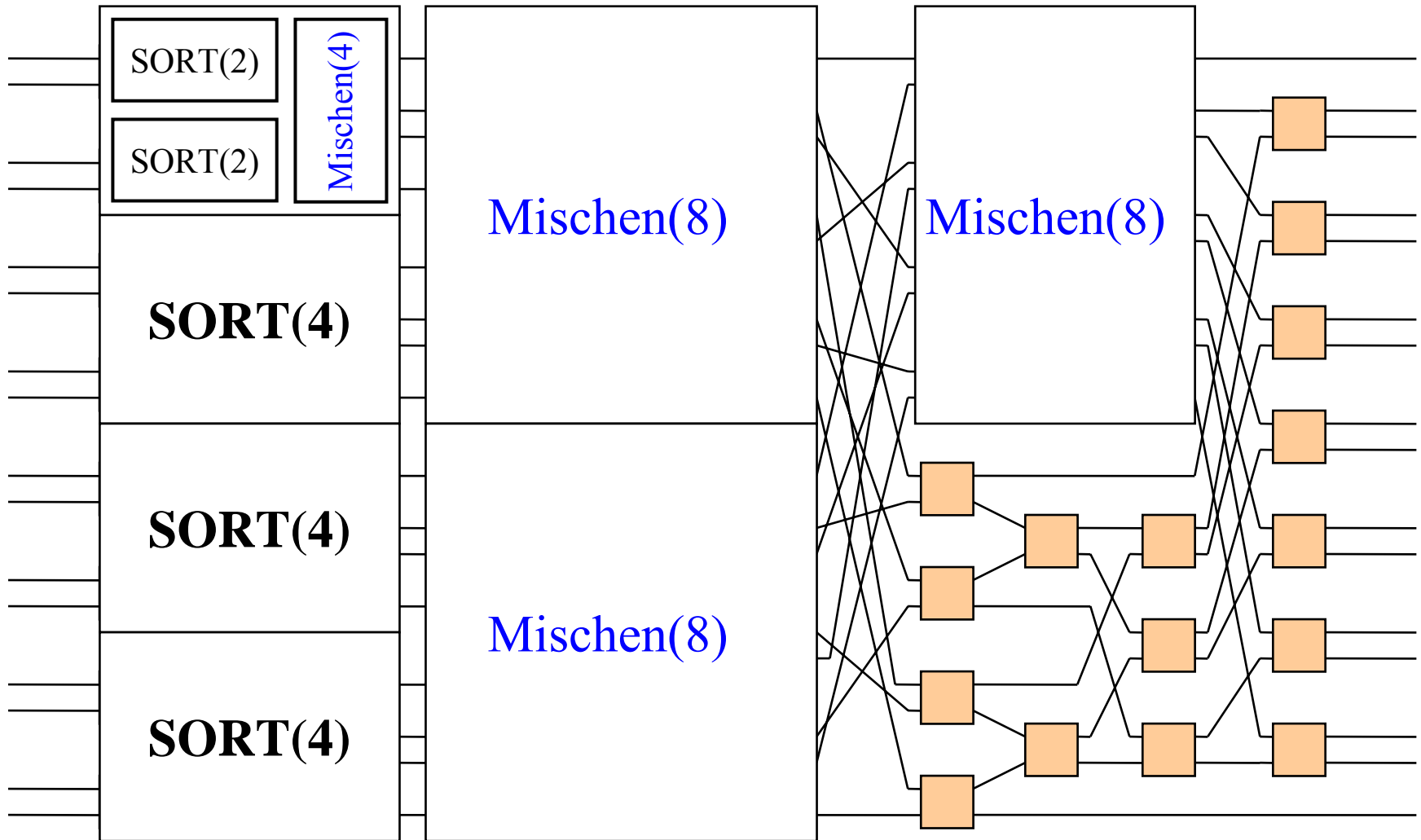
Siehe weiteres Studium, Praktika, Projekte

10.7.17 Zum Abschluss das vollständige Beispiel SORT(16)

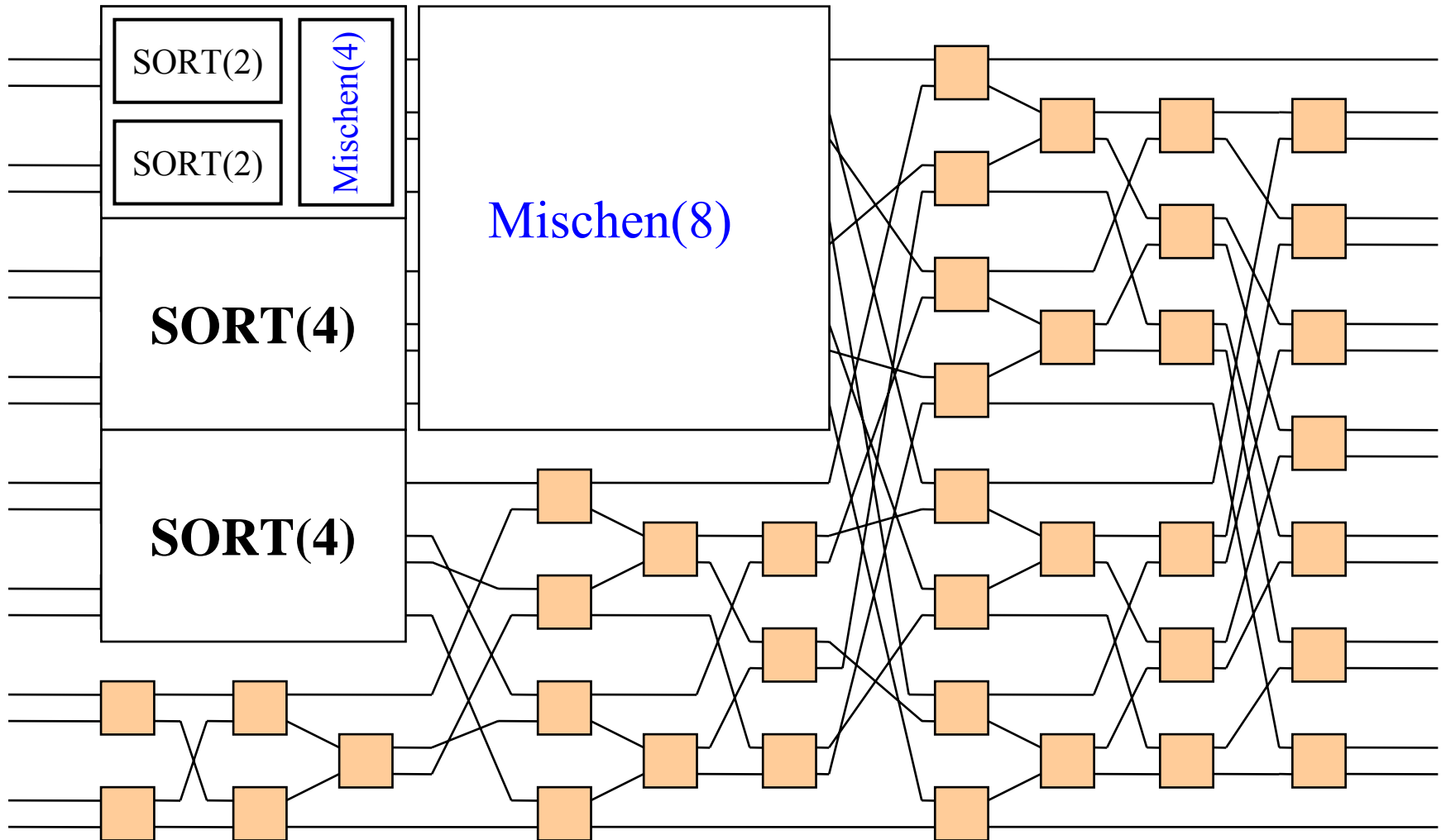




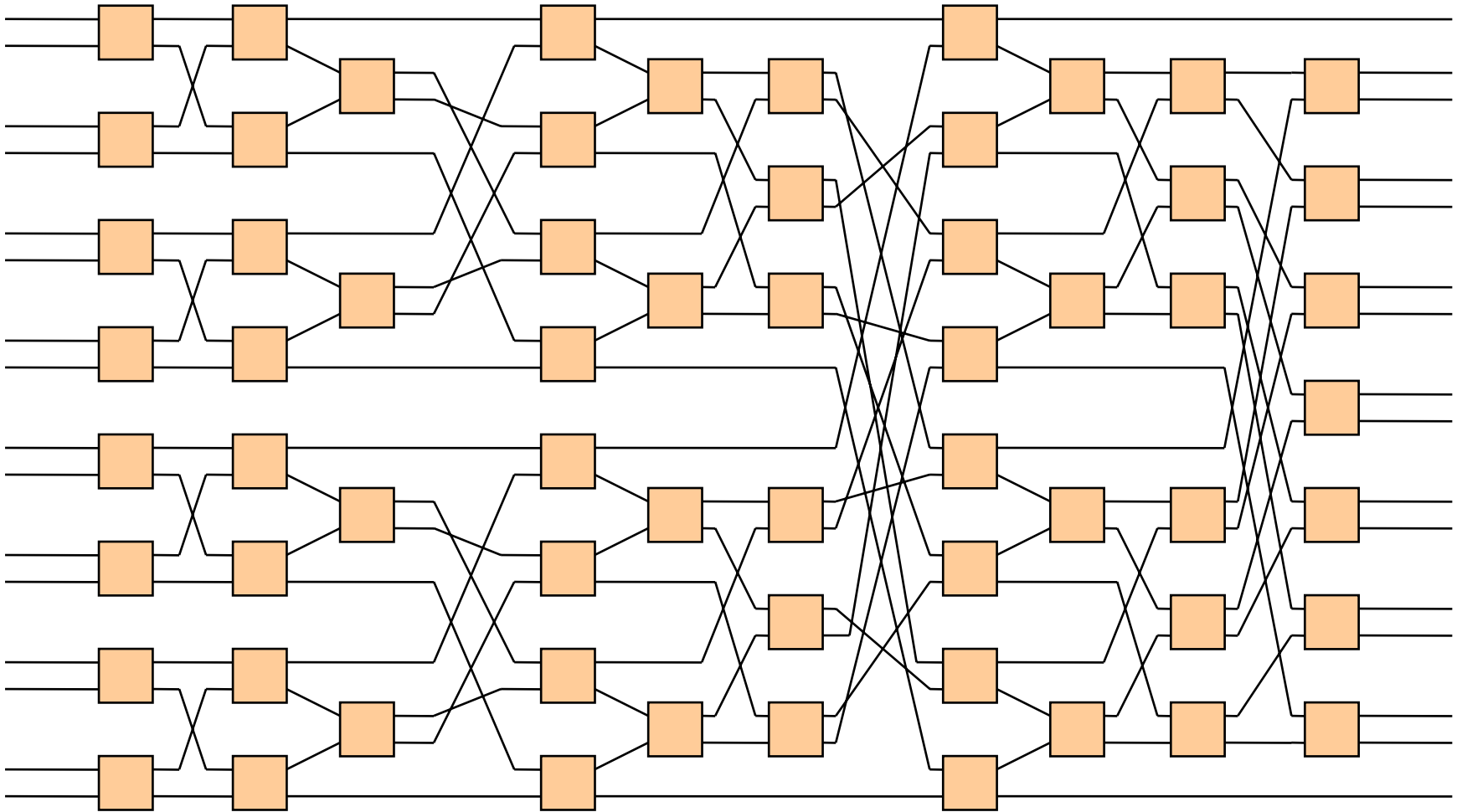
SORT(16)



SORT(16)



SORT(16)



63 Bausteine **sort**, größte Tiefe: 10, Breite: 8.

Einführung in die Informatik II

Universität Stuttgart, Studienjahr 2007

Gliederung der Grundvorlesung

~~8. Suchen~~

~~9. Hashing~~

~~10. Sortieren~~

11. Graphalgorithmen

12. Speicherverwaltung

11. Graphalgorithmen

11.1 Topologisches Sortieren

11.2 Kürzeste Wege

11.3 Minimaler Spannbaum

11.4 Maximales Matching

Graphen

Viele zu verarbeitende Informationen besitzen eine Graph-Struktur: Sie bestehen aus Teilen (dies entspricht den Knoten), die untereinander in Beziehung stehen (dies entspricht den Kanten), wobei die Beziehungen gewichtet sind oder Kommunikation ermöglichen und die Informationen Attribute tragen oder wiederum aus Graphen bestehen (markierte Graphen, hierarchische Graphen).

Alle erforderlichen Definitionen über Graphen finden Sie in 3.7 und 3.8., sowie 8.2 und 8.8. Die Adjazenzdarstellung steht in 3.8.6; Graphdurchläufe (zunächst GD genannt, nun als Tiefen- und als Breitendurchlauf DFS bzw. BFS bezeichnet) finden Sie in 3.8.7 und 8.8.9. In 6.5.4 wird die transitive Hülle eines gerichteten Graphen in $O(n^3)$ Schritten berechnet.

Wir gehen davon aus, dass Sie die grundlegenden Begriffe bereits früher eingeübt haben und kennen.

Ziele des 11. Kapitels:

Gewisse Verfahren auf Graphen werden in den unterschiedlichsten Anwendungen gebraucht. Den Graph- und den Baumdurchlauf haben wir bereits kennen gelernt und ihn auf die Ermittlung der Zusammenhangskomponenten angewendet (DFS und BFS, inorder usw., Baumsortieren, siehe z.B. 3.7.7 und 8.8.9).

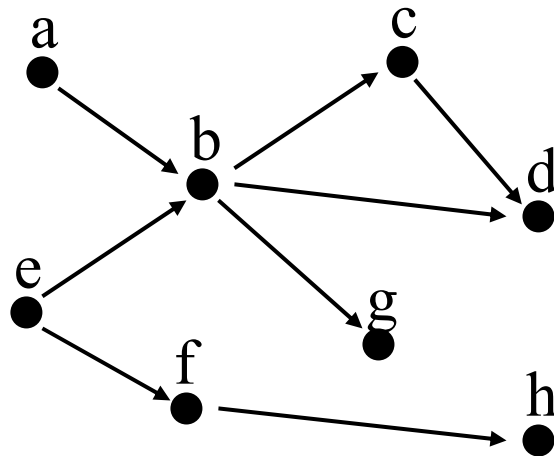
In diesem Kapitel lernen Sie weitere Verfahren, und zwar:

- ordne eine Halbordnung linear,
- finde den kürzesten Weg in einem kantenmarkierten Graphen,
- finde das minimale Gerüst in einem Graphen,
- kombiniere die Kanten eines Graphens optimal
(= löse ein spezielles Zuordnungsproblem).

Sie sollen hierfür Lösungsalgorithmen beschreiben und deren Komplexität herleiten können. Zugleich sollen Sie die Korrektheit dieser Verfahren begründen können.

11.1 Topologisches Sortieren

Topologisches Sortieren ist das Einbetten einer Halbordnung in eine totale (= lineare) Ordnung. Die Halbordnung ist durch einen azyklischen Graphen und seine transitive Hülle gegeben.



Angabe zweier Funktionen
ord, vgl. Def. 8.8.2:

a	b	c	d	e	f	g	h
1	4	5	7	2	3	6	8

a	b	c	d	e	f	g	h
4	5	7	8	1	2	6	3

Es gibt noch weitere Anordnungen ord für dieses Beispiel.

Hilfssatz 11.1.1:

Es sei G ein DAG, dann gibt es mindestens einen Knoten mit Eingangsgrad 0 und mindestens einen Knoten mit Ausgangsgrad 0.

Diese Aussage ist "klar": Folge von irgendeinem Knoten den auslaufenden Kanten. Da man in einem azyklischen Graphen nicht wieder auf einen bereits besuchten Knoten stoßen darf, muss jede Kantenfolge nach spätestens $n-1$ Schritten in einem Knoten mit Ausgangsgrad 0 enden. Analog beim Rückwärts-Verfolgen von Kanten.

(DAG = gerichteter azyklischer Graph, siehe 3.8)

Satz 11.1.2: Ein gerichteter Graph besitzt genau dann eine topologische Sortierung, wenn er azyklisch ist.

" \Rightarrow ": Ein Graph möge eine topologische Sortierung $\text{ord}: V \rightarrow \mathbb{N}$ mit $\forall u, v \in V$ mit $u \neq v$ gilt: $(u, v) \in E^* \Rightarrow \text{ord}(u) < \text{ord}(v)$ besitzen. Betrachte dann einen geschlossenen Weg in G : $(u_0, u_1, \dots, u_{k-1}, u_0)$ mit $k \geq 0$. Wegen $(u_0, u_0) \in E^*$ muss dann $\text{ord}(u_0) < \text{ord}(u_0)$ gelten, Widerspruch.

" \Leftarrow ": Setze $i = 1$. Ein gerichteter azyklischer Graph besitzt mindestens einen Knoten u mit dem Eingangsgrad 0. Setze $\text{ord}(u) = i$, erhöhe i um 1, entferne den Knoten u und die zu ihm inzidenten Kanten (es entsteht ein gerichteter azyklischer Graph G') und fahre rekursiv mit G' fort. So erhält jeder Knoten u aus G eine Nummer $\text{ord}(u)$. Wenn nun $(u, v) \in E^*$ in G gilt, so kann der Eingangsgrad von v bei dieser Konstruktion erst dann 0 werden, wenn der Knoten u bereits entfernt ist. Dies heißt aber: $\text{ord}(u) < \text{ord}(v)$.

11.1.3: Der Beweis liefert zugleich einen
Algorithmus zur Berechnung einer topologischen Sortierung:

```
i := 0;  
for j in 1..n loop  
    wähle einen Knoten u mit Eingangsgrad 0;  
    i:=i+1; setze ord(u) := i;  
    entferne u aus dem Graphen;  
end loop;
```

Man beachte, dass sich der Eingangsgrad der Knoten bei jedem Entfernen eines Knotens um 1 verringern kann.

Da es mehrere Knoten mit dem Eingangsgrad 0 geben kann und da jede Auswahl eines solchen Knotens zu einer topologischen Sortierung führt, sind topologische Sortierungen in der Regel nicht eindeutig.

Probleme bei diesem Algorithmus:

- Wie findet man am Anfang rasch einen Knoten mit dem Eingangsgrad 0? (siehe Programmstück unten)
- Wie aktualisiert man die Eingangsgrade beim Entfernen eines Knotens?

(Zur Adjazenzdarstellung von Graphen siehe 3.8.6.)

```
p := Anfang;           -- Die Eingangsgrade werden in zahl1 gespeichert
while p /= null loop p.zahl1 := 0; p := p.NKn; end loop;
p := Anfang;           -- Gehe alle Kanten durch und erhöhe den
while p /= null loop edge := p.EIK; -- Eingangsgrad des Zielknotens
  while edge /= null loop
    edge.EKn.zahl1 := edge.EKn.zahl1 + 1;
    edge := edge.NKa;
  end loop;
  p := p.NKn;
end loop;
```

Dieses Programmstück ermittelt alle Eingangsgrade in $O(n+m)$ Schritten.

Um schnell einen Knoten mit Eingangsgrad 0 finden zu können, speichern wir alle diese Knoten in einer Liste (oder einem Feld) "*GradNull*":

```
p := Anfang; "Setze GradNull auf die leere Liste";  
while p  $\neq$  null loop  
    if p.zahl1 = 0 then "füge p an GradNull an" end if;  
    p := p.NKn; end loop;
```

Nun müssen wir noch die Eingangsgrade aktualisieren, sobald ein Knoten u aus dem Graphen entfernt wird.

Hierzu erniedrigen wir die Eingangsgrade aller Knoten, die über die Kantenliste von u erreichbar sind, um 1. Falls hierbei ein Eingangsgrad auf 0 absinkt, wird der entsprechende Knoten in *GradNull* aufgenommen. (u sei der Verweis auf den als nächstes zu entfernenden Knoten mit Eingangsgrad 0.) Dies liefert:

```
edge := u.EIK;  
while edge /= null loop  
    edge.EKn.zahl1 := edge.EKn.zahl1 - 1;  
    if edge.EKn.zahl1 = 0  
        then "füge edge.EKn an GradNull an" end if;  
    edge := edge.NKa;  
end loop;  
"entferne den Knoten u aus dem Graphen G"
```

Die restlichen Details überlassen wir den Leser(inne)n.
(Könnte z.B. zahl1 irrtümlich kleiner als 0 werden, so dass die Abfrage "if edge.EKn.zahl1 = 0 ..." übergangen wird?)
Zur Datenstruktur: Entweder arbeitet man auf einer Kopie von G und trägt zusätzlich zahl1 im Original ein oder man verwendet einen Booleschen Wert, um zu simulieren, dass man den jeweiligen Knoten gelöscht hat.

11.1.4: Komplexität dieses Algorithmus:

Zeitkomplexität des topologischen Sortierens: $O(n+m)$.

Platzkomplexität $O(n)$ (bei Verwendung eines Booleschen Wertes).

Begründung zur Zeitkomplexität:

Jeder Knoten erhält seinen Eingangsgrad: $O(n+m)$.

Aufbau der Liste *GradNull*: $O(n)$.

n mal: Einen Knoten mit Eingangsgrad 0 finden (= nimm stets den ersten Knoten in der Liste *GradNull*) und später entfernen, insgesamt: $O(n)$.

m mal insgesamt: Eingangsgrade erniedrigen und Knoten mit Eingangsgrad 0 an *GradNull* anhängen: $O(n+m)$.

Alle übrigen Operationen ($i := i+1$; setze $\text{ord}(u) := i$; usw.) erfordern insgesamt höchstens $O(n)$ Schritte.

11.1.5: Wir betrachten folgenden Algorithmus **TopSort**, der in $O(n+m)$ Schritten arbeitet und im Wesentlichen gleich der Tiefensuche ist (n =Zahl der Knoten, m =Zahl der Kanten):

```
procedure TS (u: NextKnoten) is                                -- nummer ist global
begin  u.Besucht := true;
        for alle Nachfolgerknoten v von u loop
            if not v.Besucht then TS (v); end if; end loop;
        u.zahl2 := nummer; nummer := nummer - 1;
end;

...
for alle Knoten p loop  p.Besucht := false; end loop;
nummer := n;
for alle Knoten p loop
    if not p.Besucht then TS (p); end if; end loop;
```

Hinweis: Diese zahl2-Werte heißen in der Literatur auch "finish"-Werte.

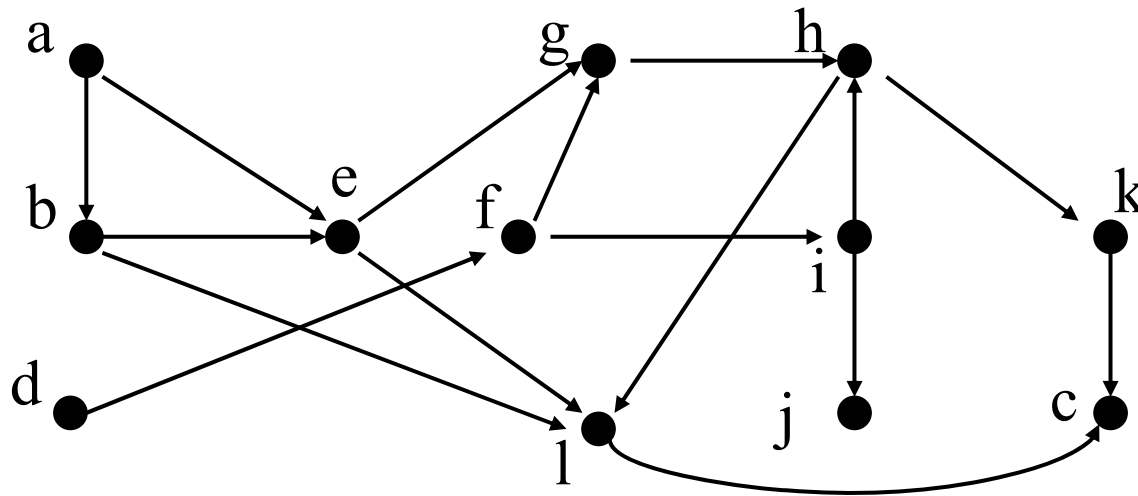
Dieser Algorithmus macht Folgendes:

Er beginnt in einem beliebigen Knoten und führt einen Tiefendurchlauf durch. Immer wenn er hierbei alle von einem Knoten ausgehenden Kanten abgearbeitet hat, ordnet er diesem Knoten die Zahl der Variablen "nummer" zu. nummer wird dann um 1 erniedrigt. Somit erhält jeder Knoten eine kleinere Nummer als alle seine Nachfolger. Da nummer mit n initialisiert wird, bekommt ein Knoten ohne Nachfolger den größten Wert.

Machen Sie sich klar, dass diese Nummern als ord-Werte verwendet werden können, da eine Ordnungsfunktion ord in einem DAG genau dann zu einer topologischen Sortierung gehört, wenn für jeden Knoten x gilt, dass $ord(x) < ord(y)$ für alle Nachfolgerknoten y , d.h. für alle $(x,y) \in E$, ist.

Also berechnet der Algorithmus eine topologische Sortierung.

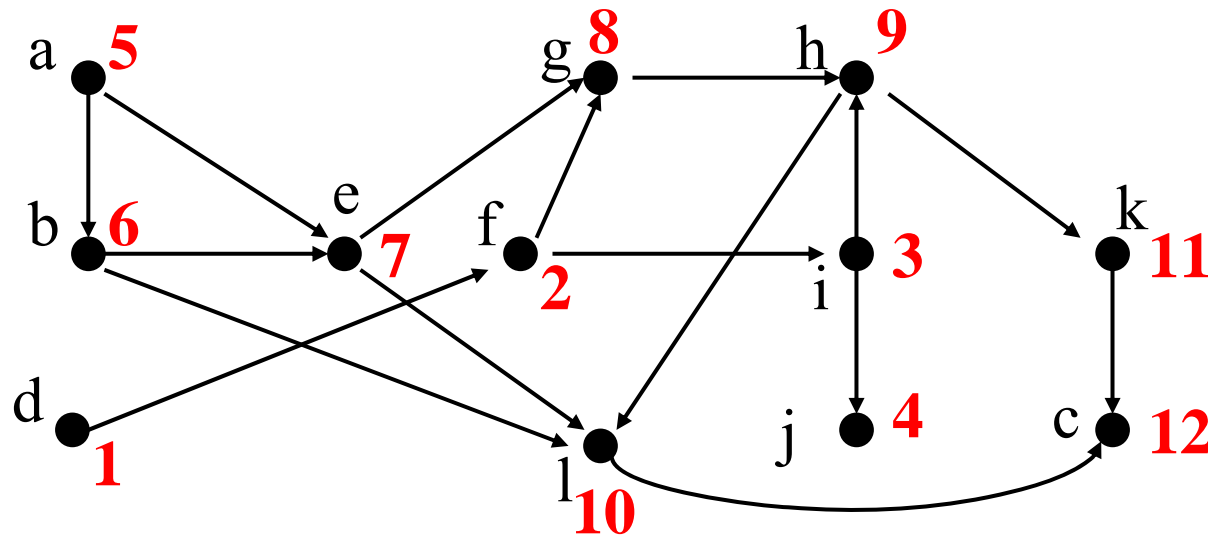
Beispiel: Azyklischer Graph mit $n=12$ Knoten und $m=16$ Kanten



Die Reihenfolge der Knoten in der Adjazenzliste sei $k, e, h, a, f, j, i, d, c, b, g, l$.

Dann liefert der obige Algorithmus TopSort folgende Zahlen für die Knotenkomponente *zahl2*, sofern man beim Nachfolger zuerst der Kante folgt, die bzgl. der Zahlen auf einer Uhr die kleinste ist.

Beispiel: Azyklischer Graph mit $n=12$ Knoten und $m=16$ Kanten



Knotenreihenfolge:
k, e, h, a,
f, j, i, d,
c, b, g, l

Frage: Was liefert der Algorithmus für Werte für beliebige gerichtete Graphen?

Antwort: Man kann die Kanten dann in Kanten, die vom Algorithmus zu einem neuen Knoten durchlaufen werden (sog. Baum-Kanten) und andere Kanten unterteilen, wobei sich letztere wiederum in Rückwärts-, Vorwärts- und Quer-Kanten einteilen lassen. Siehe Algorithmik-Vorlesungen.

11.2 Kürzeste Wege

Gegeben sei ein gerichteter Graph $G = (V, E, \delta)$ mit $\delta: E \rightarrow \mathbb{R}^{\geq 0}$ (= Menge der nichtnegativen reellen Zahlen) und ein Knoten $u \in V$.
Gesucht werden alle kürzesten Entfernungen $\delta(u, v)$ für alle Knoten $v \in V$.
Wir wollen also SSSP lösen (8.8.5).

Hierfür verwendet man in der Regel den [Dijkstra-Algorithmus](#), der sich wie eine Breitensuche (gewichtet bzgl. der Entfernungsfunktion δ) über den Graphen vorarbeitet. Zeitkomplexität bei Implementierung mit einem Heap: $O((n+m) \cdot \log(n))$.

Hinweis: Sucht man die kürzesten Abstände zwischen *allen* Knoten (APSP), so kann man den Dijkstra-Algorithmus für jeden Knoten einmal durchführen oder den [Floyd-Algorithmus](#) mit Zeitkomplexität $\Theta(n^3)$ verwenden. Dieser Algorithmus ähnelt stark dem Warshall-Algorithmus in 6.5.4 und ist in Lehrbüchern gut beschrieben.

(Edgar W. Dijkstra und R. W. Floyd: holländischer bzw. amerikanischer Wissenschaftler, "Pioniere" der Informatik. Ihre Algorithmen wurden 1959 und 1962 veröffentlicht. <Aussprache dieser Namen: "Deik-stra" und "Fleut">)

11.2.1 Aufgabenstellung (SSSP)

SSSP = single source shortest paths
= alle kürzesten Wege, die von einem gegebenen Knoten zu jedem anderen Knoten führen.

Gegeben ist ein gerichteter Kanten markierter Graph $G = (V, E, \delta)$ mit $\delta: E \rightarrow \mathbb{R}^{\geq 0}$ und ein "Startknoten" $u \in V$ (u ist "single source", also die einzige Quelle, von der ausgehend alle kürzesten Wege berechnet werden sollen; $\mathbb{R}^{\geq 0}$ ist die Menge der nichtnegativen reellen Zahlen).

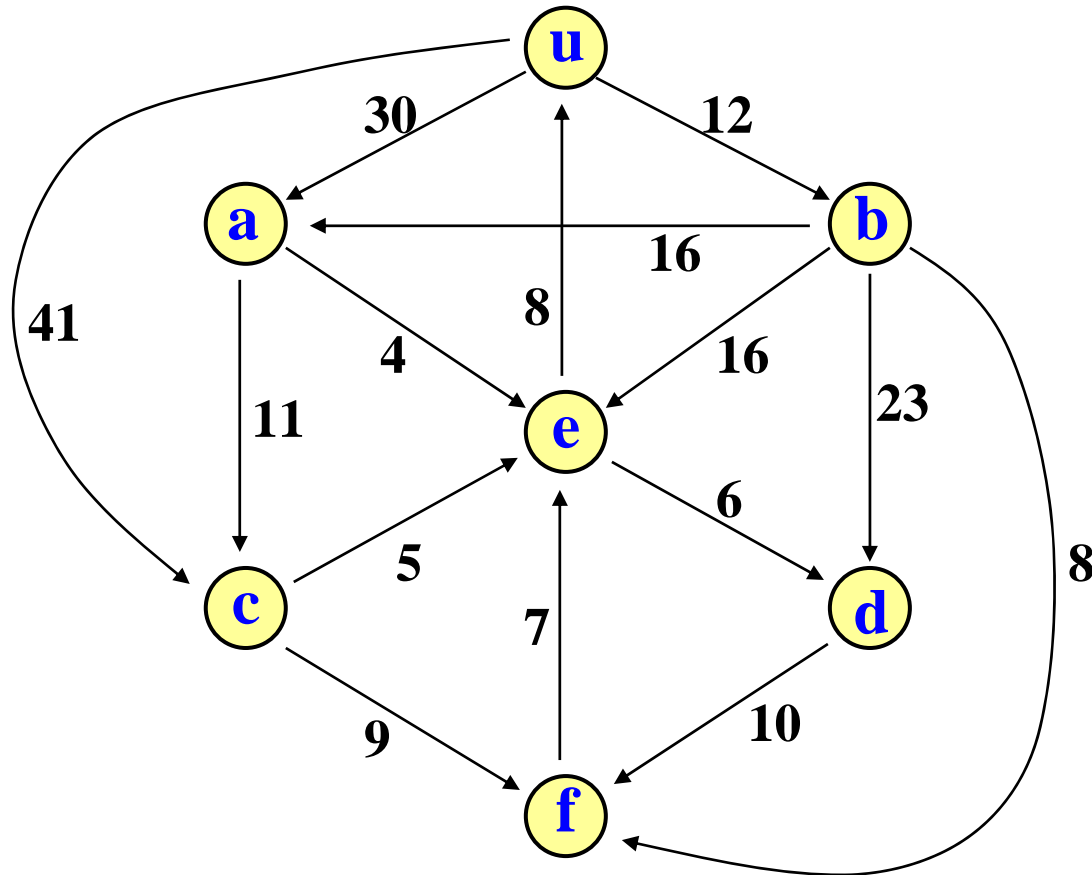
Gesucht werden für jeden Knoten $v \in V$ ein kürzester Weg von u nach v einschließlich seiner Länge $\text{dist}(u,v)$
= Summe aller $\delta((u,v))$ für alle Kanten (u,v) , aus denen dieser Weg besteht.

Zuerst zu klären: Wie sollen diese Wege dargestellt werden? Würde man alle Wege tatsächlich hinschreiben, so benötigt man $O(n^2)$ Speicherplatz, da jeder doppelpunktfreie Weg von u zu einem Knoten v bis zu n Knoten enthalten kann.

Vorschlag: Man speichert zu jedem Knoten v nur den Knoten, der auf einem kürzesten Weg von u nach v direkter Vorgänger von v ist. Dann kann man (rückwärts gehend) schrittweise einen kürzesten Weg von hinten nach vorne zu jedem Knoten aufbauen.

11.2.2 Beispiel

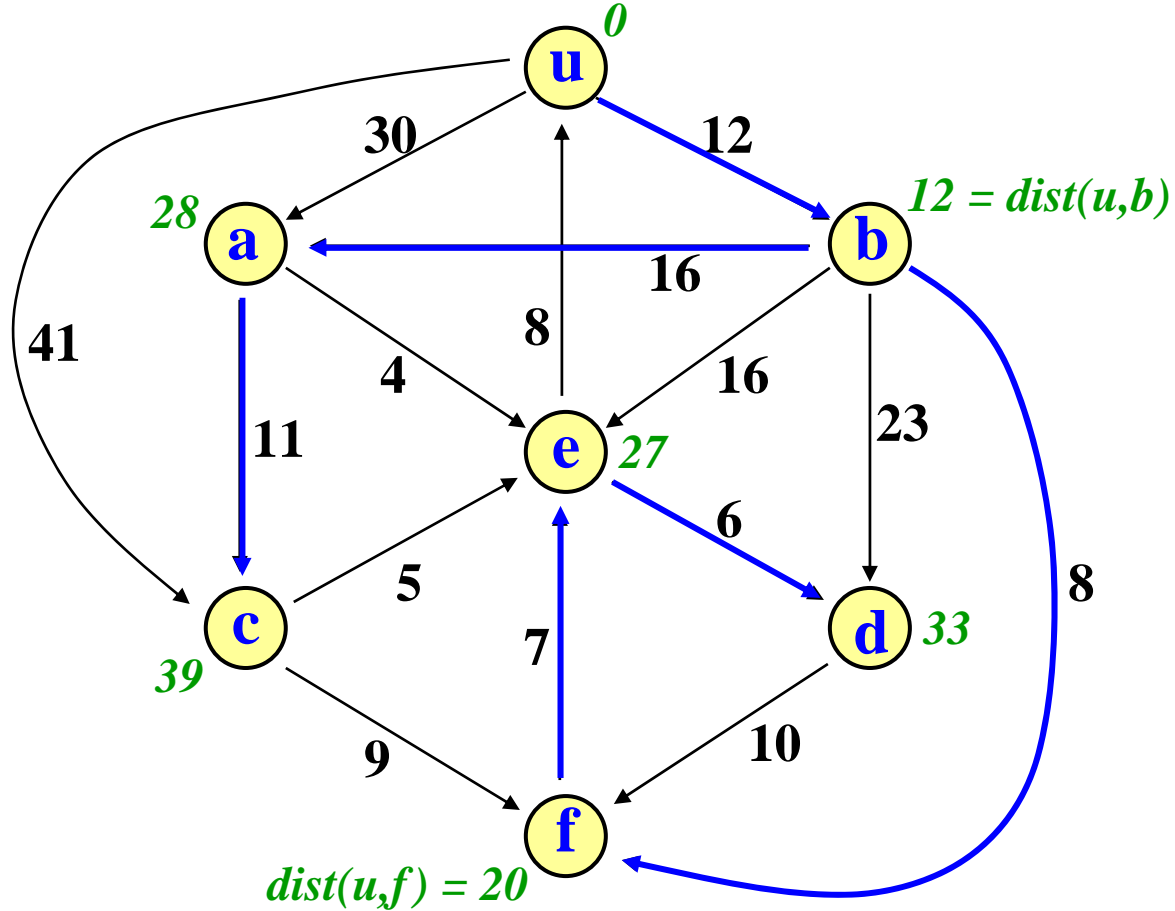
$n=7, m=15$



Dieser Graph ist stark zusammenhängend. Startknoten sei u .
Wir konstruieren nun einen Teilgraphen mit kürzesten Wegen.

Beispiel:

$n=7, m=15$

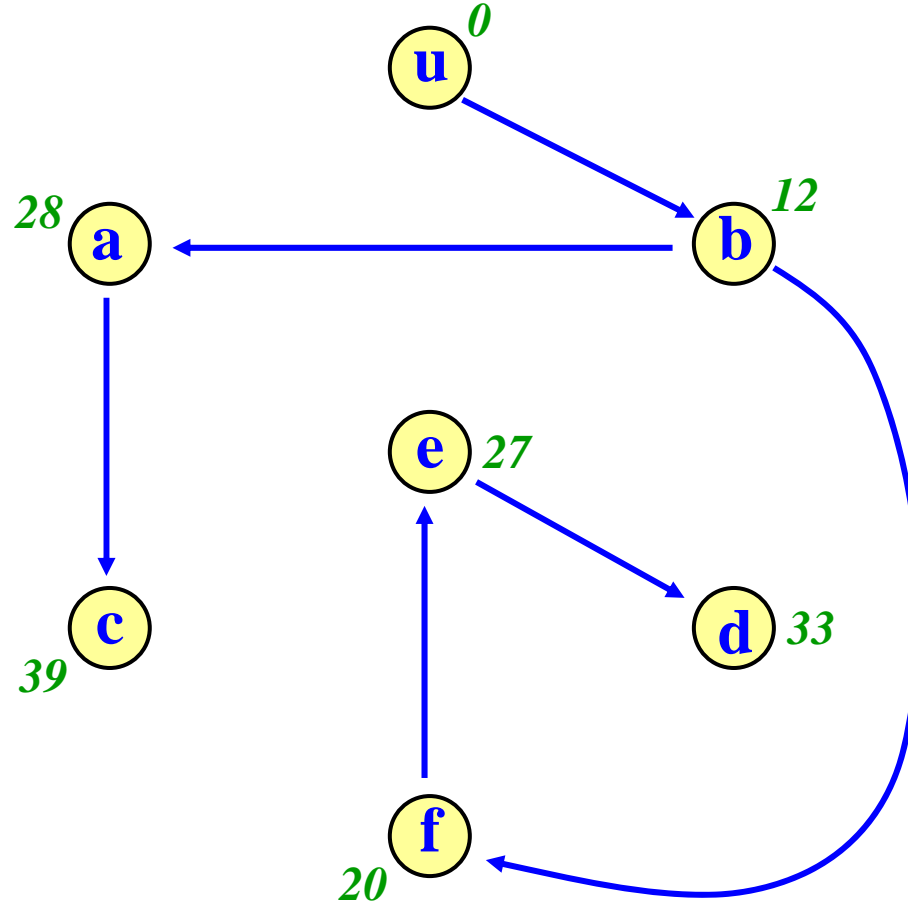


Der Teilgraph mit den kürzesten Wegen bildet einen Baum!

Beispiel:

$n=7, m=15$

Dies ist ein
"Kürzeste-
Wege-
Baum"
für den
Knoten u

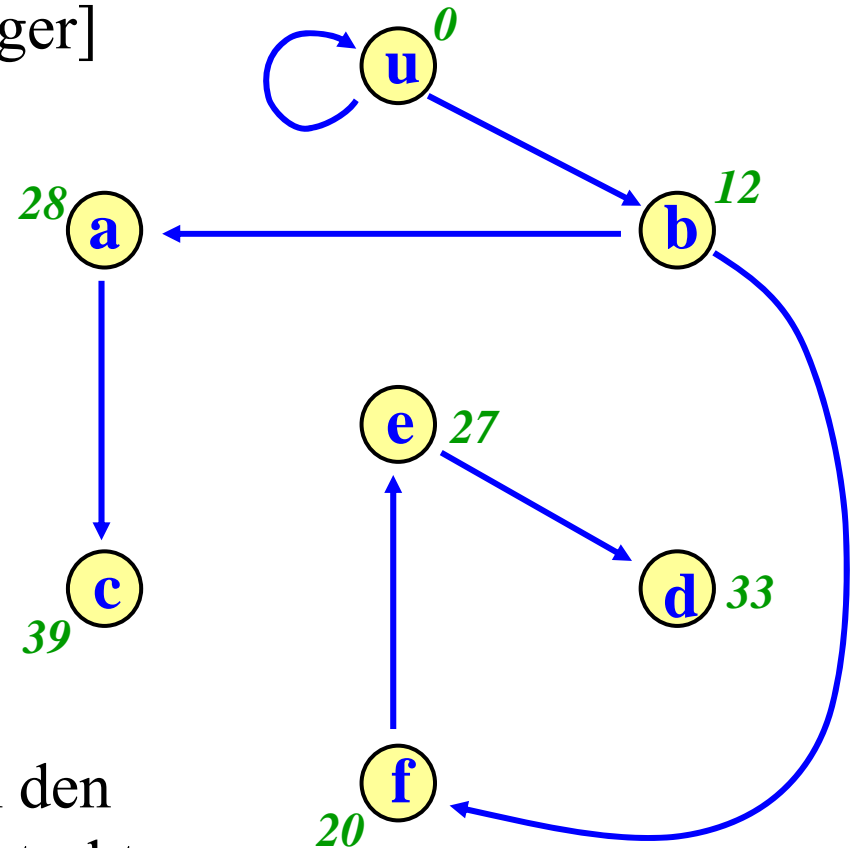


Also stellen wir kürzeste Wege bzgl. u wie folgt dar:

Schema für die Darstellung kürzester Wege:
(Knoten, Vorgängerknoten, Distanz von u)
[für u nehmen wir u als Vorgänger]

Hieraus
kann man
kürzeste
Wege rück-
wärts rekon-
struieren.

(u, u, 0),
(a, b, 28),
(b, u, 12),
(c, a, 39),
(d, e, 33),
(e, f, 27),
(f, b, 20).



Diese Werte schreibt man oft in den Graphen hinein, d.h., die Datenstruktur "Knoten" muss dann um entsprechende Komponenten erweitert werden.

11.2.3 Hinweis: Im Allgemeinen gibt es mehrere kürzeste Wege zwischen Knoten.

Wir interessieren uns hier immer nur für genau *einen* kürzesten Weg, der beliebig ausgewählt werden darf. Unter dieser Annahme **bilden die kürzesten Wege einen Baum**.

Grund: Wenn in diesem Teilgraphen zwei verschiedene Wege von einem Knoten x zu einem Knoten y führen würden, dann lägen mindestens zwei kürzeste Wege vor, von denen wir einen (durch Entfernen mindestens einer Kante) streichen. Entfernt man auf diese Weise alle doppelten Wege, so entsteht schließlich der genannte "**Kürzeste-Wege-Baum**" von u zu allen erreichbaren Knoten. (Dieser ist nicht eindeutig, aber für jeden Startknoten u sind selbstverständlich die Entfernungen zu allen Knoten in jedem zu u gehörenden Kürzeste-Wege-Baum gleich.)

An obigem Beispiel sieht man die Vorgehensweise, um einen Kürzeste-Wege-Baum zu einem Knoten u zu konstruieren:

Annahme, man hat bereits zu einer Menge von Knoten B je einen kürzesten Weg mit Distanz $d(v) = \text{dist}(u, v)$ für jedes $v \in B$ berechnet, dann wähle man als nächstes einen Knoten x ($x \notin B$), der die geringste Distanz von u besitzt, für den also gilt:

$$d(v) + \delta((v, x)) \leq d(v) + \delta((v, y)) \quad \text{für alle Knoten } v \in B, y \notin B.$$

Setze $d(x) = \text{"dieses Minimum } d(v) + \delta((v, x))\text{"}$, füge x zu B hinzu und fahre rekursiv fort, bis alle Knoten betrachtet wurden. Initialisierung: $B = \{u\}$ mit $d(u) = 0$.

Dieses Vorgehen wird als **Dijkstra-Algorithmus** bezeichnet. Problem hierbei: Finde einen solchen Knoten x möglichst schnell. Man speichert hierzu die als nächstes zu betrachtenden Knoten in einer "Prioritätswarteschlange". Dann gibt es drei Grundoperationen, die wir nun vorstellen.

11.2.4 Struktur des Dijkstra-Algorithmus

Grundlegende Mengen und Funktionen (blau gefärbt):

Gerichteter Graph $G = (V, E, \delta)$ mit $\delta: E \rightarrow \mathbb{R}^{\geq 0}$ und Knoten u .

B = Menge der bereits abschließend "bearbeiteten" Knoten.

R = Menge der "Randknoten", die von B aus erreichbar sind.

U = Menge der übrigen, bisher noch "unsichtbaren" Knoten.

Es gilt stets: $V = B \cup R \cup U$ (und B, R, U sind paarweise disjunkt).

Initialisierung: $B = \{u\}$, $R = \{v \mid (u,v) \in E\}$, $U = V - R - \{u\}$.

$D: V \rightarrow \mathbb{R}^{\geq 0}$ mit $D(v) = \text{vorläufiger Wert}$ für die kürzeste Entfernung von u zum Knoten v (stets ist $D(v) \geq \text{dist}(u,v)$).

Aber: Für die Knoten v in B wird gelten: $D(v) = \text{dist}(u,v)$.

Man stellt D als array (V) of real dar mit der Initialisierung

$D(v) = \text{maximal darstellbare reelle Zahl}$, außer $D(u) = 0.0$.

Vorg: array (V) of V (mit null) gibt zu jedem Knoten den Vorgänger auf einem kürzesten Weg an. Initialisierung mit null.

In jedem Schritt wird ein Knoten x aus R mit seinem D-Wert in die Menge B aufgenommen, die Menge R wird um alle Knoten v aus U , die von x aus erreichbar sind, erweitert und die von x aus erreichbaren Knoten y in R werden darauf überprüft, ob über x ein kürzerer Weg von u nach y existiert.

Hierfür verwenden wir folgende drei Grundoperationen, wobei der D-Wert als "Schlüssel" (key) verwendet wird:

DELETEMIN = Finde und liefere x mit minimalem D-Wert, entferne x aus R , füge x zu B hinzu.

INSERT(y) = Berechne für y den D-Wert und füge y in R ein.

DECREASEKEY(y) = Wenn ein kürzerer Weg von u über den Knoten x nach $y \in R$ existiert, so berechne das kleinere $D(y)$ und positioniere y neu in R .

Dies liefert dann folgendes Verfahren (man sieht rasch, dass man die Menge U nicht explizit braucht):

```
B := {u}; R := { v | (u,v) ∈ E };  
while R ≠ ∅ do  
    X := DELETMIN;  
    for all Y ∈ S(X) and Y ∉ B do  
        if Y ∉ R then INSERT(Y)  
        else DECREASEKEY(Y) fi  
    od  
od;
```

Hierbei ist $S(x) = \{y \mid (x,y) \in E\}$ die Menge der Nachfolgerknoten ("successors") eines Knotens x im gegebenen Graphen. Um dies entsprechend der Bedeutungen der Grundoperationen in ein Ada-Programm umzusetzen, muss die Datenstruktur für R festgelegt werden. Alles Übrige ergibt sich hieraus.

11.2.5 Prioritätswarteschlange

Eine "abstrakte" Datenstruktur, in der diverse Elemente auf ihre Abarbeitung warten (wobei sich deren Priorität nach einem Schlüssel aus einer geordneten Menge richtet), in die jederzeit Elemente eingefügt werden dürfen und aus der jederzeit das Element mit dem kleinsten Schlüssel entfernt werden kann, bezeichnet man als **Prioritätswarteschlange**.

Eine Prioritätswarteschlange besitzt die drei Grundoperationen **DELETEMIN**, **INSERT** und **DECREASEKEY**.

Anregung: Formulieren Sie die Prioritätswarteschlange als abstrakten Datentyp (sofern Kapitel 5 bekannt ist).

Prioritätswarteschlangen realisiert man oft als Heap (10.4.3).

Eine Menge von n Schlüsseln werde als Heap gespeichert.

Dann realisiert man die Operation $X := \text{DELETEMIN}$, indem man das oberste Element des Heaps der Variablen X zuweist, das letzte Element des Heaps an die oberste Stelle setzt und es dann absinken lässt. Zeitkomplexität: $\leq 2 \cdot \log(n)$ Vergleiche.

Die Operation $\text{INSERT}(Y)$ wird realisiert, indem der Wert von Y als letztes Element an den Heap angehängt wird und dieses Element dann im Heap aufsteigt, d.h., man vergleicht diesen neuen Schlüssel $D(Y)$ mit seinem Elternknoten; falls der Elternknoten ein größeres Element enthält, werden die Inhalte vertauscht und man fährt genauso an dem neuen Knoten fort. Zeitkomplexität: $\leq \log(n) + 1$ Vergleiche.

Bei $\text{DECREASEKEY}(Y)$ lässt man das Element Y , das ja schon im Heap steht, aufsteigen, sofern sich $D(Y)$ durch einen Weg über den Knoten X verringert hat. Zeitkomplexität: $\leq \log(n) + 1$ Vergleiche.

Mit dieser Heap-Realisierung lässt sich nun die Komplexität des Dijkstra-Algorithmus abschätzen. Weil in jedem äußeren Schleifendurchlauf genau ein Element aus $V - \{u\}$ bearbeitet wird, gibt es maximal $n-1$ Durchgänge der while-Schleife.

$B := \{u\}; R := \{v \mid (u,v) \in E\};$	-- n Schritte
<u>while</u> $R \neq \emptyset$ <u>do</u>	-- maximal $n-1$ Schleifen
$X := \text{DELETEMIN};$	-- $\leq 2 \cdot \log(n)$ Vergleiche
<u>for all</u> $Y \in S(X)$ <u>and</u> $Y \notin B$ <u>do</u>	-- maximal $n-1$ Schleifen
<u>if</u> $Y \notin R$ <u>then</u> $\text{INSERT}(Y)$	-- $\leq \log(n)$ Vergleiche
<u>else</u> $\text{DECREASEKEY}(Y)$ <u>fi</u>	-- $\leq \log(n)$ Vergleiche
<u>od</u>	
<u>od</u> ;	

Es gibt also höchstens

$(n-1) \cdot (2 \cdot \log(n) + (n-1) \cdot \log(n)) = O(n^2 \cdot \log(n))$ Vergleiche.

Eine genauere Betrachtung ergibt eine günstigere Schranke:

Die Abfrage " $Y \in S(X)$ and $Y \notin B$ " wird für jede Kante genau ein Mal durchgeführt, insgesamt also m Mal ($m = |E|$). Das Einfügen und das Aufsteigen eines Elements erfordert jedes Mal höchstes $\log(n)$ Vertauschungen. Somit erfordert die innere Schleife *insgesamt* höchstens $O(m \cdot \log(n))$ Schritte.

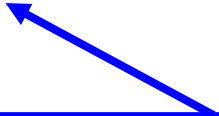
Hinzu kommen dann nur noch die maximal $O((n-1) \cdot 2 \cdot \log(n))$ Operationen, die durch **DELETEMIN** verursacht werden.

Satz 11.2.6: Der gesamte Zeitaufwand beträgt daher

$$O((n-1) \cdot 2 \cdot \log(n)) + m \cdot \log(n) = \mathbf{O((n+m) \cdot \log(n))}.$$

verursacht durch
DELETEMIN

verursacht durch **INSERT**
und **DECREASEKEY**



Laufzeit des Dijkstra-Algorithmus, wenn man ihn mit Heaps implementiert

Aufgabe: Man könnte nun folgendermaßen argumentieren:

Die Abfrage " $Y \in S(X)$ and $Y \notin B$ " wird für jede Kante genau ein Mal durchgeführt, insgesamt also m Mal ($m = |E|$). Das Einfügen und das Aufsteigen eines Elements erfordern insgesamt höchstes $n \cdot \log(n)$ Vertauschungen (spätestens dann ist das jeweilige Element an der Spitze des Heaps angekommen); alle $n-1$ Elemente zusammen benötigen daher maximal $(n-1) \cdot \log(n)$ Vertauschungen. Somit erfordert die innere Schleife insgesamt höchstens $O(m + n \cdot \log(n))$ Schritte.

Hinzu kommen dann nur noch die maximal $O((n-1) \cdot 2 \cdot \log(n))$ Operationen, die durch **DELETEMIN** verursacht werden.

Somit würden wir insgesamt folgenden Zeitaufwand erhalten:
 $O((n-1) \cdot 2 \cdot \log(n) + m + n \cdot \log(n)) = O(m + n \cdot \log(n))$.

Was ist an dieser Argumentation falsch??

11.2.7 Realisierung des Dijkstra-Algorithmus in Ada

Wir geben eine "strukturierte Lösung" für die Realisierung an, also eine Lösung, die den Algorithmus aus 11.2.4 beibehält und die Prioritätswarteschlange als Modul (package) formuliert. **Wer sich hier einige Stunden lang durchbeißt (und eventuelle Fehler aufdeckt), hat den Algorithmus und die Implementierung wirklich verstanden.**

Eine Prioritätswarteschlange wird über einer Schlüsselmenge definiert, die eine Ordnung besitzt. In Ada bietet sich hierfür ein generisches Paket an:

Die Typen für Knoten und Schlüssel sind generisch, zusammen mit den Operationen "Addition" und "Kleiner-Relation" auf dem Schlüsseltyp.

Das Paket muss einen privaten Typ für die Prioritätswarteschlange (pqk), die drei Grundoperationen und einige Hilfsfunktionen (Initialisieren, Test auf Leerheit) besitzen. Die Spezifikation kann man daher unmittelbar hinschreiben:

```

generic  type Node is private; type Key is private;
        Z: constant Natural;
        with function "+"(I, J: Key) return Key;
        with function "<"(I, J: Key) return Boolean;

```

```

package Prio_Queue is

```

```

    type Item is record

```

```

        Kn: Node; Vor: Node; Sch: Key;

```

```

    end record;

```

```

    type pqk is private;

```

```

    procedure Empty;

```

```

    function Iempty return Boolean;

```

```

    function DeleteMin return Item;

```

```

    procedure Insert (K: Item);

```

```

    procedure DecreaseKey (K: Item);

```

```

    private

```

```

        type MaxHeap is 0..Z;

```

```

        type pqk is array (MaxHeap) of Item;

```

```

end package;

```

Aktueller Knoten *Kn*,
 Vorgängerknoten *Vor* auf
 einem kürzesten Weg,
Sch = Distanz zum Startknoten
 (zum Speichern des D-Werts)
 (*Sch* negativ => Fehlerfall)

-- pqk \cong priority queue w.r.t. key

Die Implementierung erfolgt in Ada als package body.

Die vorläufigen Werte für die kürzeste Entfernung werden in der Item-Komponente *Sch* für jeden Knoten gespeichert.

Die Programmierung muss dafür sorgen, dass dieser Wert beim Löschen in R und Aufnehmen in B erhalten bleibt.

Dies geschieht außerhalb des Pakets Prio_Queue. Um die kürzesten Wege später rekonstruieren zu können, benötigen wir die Komponente *Vor*.

Wir stellen nun die zwölfseitige Implementierung des Pakets vor. Das Feld KK vom Typ pqr bildet den Heap der Items. In der Reihenfolge des Entfernens von Knoten aus KK wird das Feld D der Ergebnisse aufgebaut. Um auf die Knoten im Heap direkt zugreifen zu können, führen wir das Feld HeapIndex ein mit

$\text{HeapIndex}(K) = \text{"der Index } i \text{ mit } \text{KK}(i).\text{Kn} = K\text{"}$.


```

package body Prio_Queue is
type MaxHeap is 0..Z;
type pqk is array (MaxHeap) of Item;
KK: pqk;                                -- KK wird hier als Heap aufgefasst
Anzahl: MaxHeap;
HeapIndex: array (Node) of MaxHeap;
procedure Empty is
  begin Anzahl := 0; end;
function Isempty return Boolean is
  begin return (Anzahl=0); end;

```

- Für das Folgende beachte man:
- Es besteht eine Beziehung zwischen einem Knoten und seinem Index im
- Heap. Hierfür verwenden wir HeapIndex: array (Node) of MaxHeap.
- Im Feld HeapIndex wird für jeden Knoten K notiert, ob er sich im Heap
- befindet; genau dann gilt: $(\text{HeapIndex}(K) > 0)$, und in diesem Falle ist
- $\text{HeapIndex}(K)$ sein Index im Heap KK.
- Delta: array (Node,Node) of Key ist die Kantenmarkierung δ im Graphen,
- die eigentlich als Komponente W in den Kanten gespeichert ist.

- Wir übernehmen die Prozedur sink für den Heap KK aus 10.4.4,
- müssen allerdings die Verknüpfung zwischen einem Knoten und seinem
- Index im Heap stets aktualisieren, d.h., HeapIndex muss beim
- Aufsteigen oder Absteigen eines Knotens im Heap entsprechend
- umgesetzt werden.

```
procedure sink (links, rechts: MaxHeap) is  
i, j: Natural; weiter: Boolean:=true; v: Item;  
begin v := KK(links); i := links; j := i+i;  
  while (j <= rechts) and weiter loop  
    if j = rechts then  
      if v.Sch < KK(j).Sch then  
        KK(i):=KK(j);  
        HeapIndex(KK(j).Kn) := i;  
        i:=j;  
      end if;  
    weiter:=false;
```

```

elsif KK(j).Sch < KK(j+1).Sch then
    if v.Sch < KK(j+1).Sch then
        KK(i) := KK(j+1);
        HeapIndex(KK(j+1).Kn) := i;
        i := j+1;
    else weiter:=false; end if;
else if v.Sch < KK(j).Sch then
    KK(i) := KK(j);
    HeapIndex(KK(j).Kn) := i;
    i := j;
    else weiter:=false; end if;
end if;
j := i+i;
end loop;
KK(i):=v;
HeapIndex(v.Kn) := i;
end sink;

```

function DeleteMin return Item is

Wurzel: Item;

begin

if not Iseempty then Wurzel := KK(1); Anzahl := Anzahl - 1;
else Wurzel := KK(0); end if; -- dieser else-Fall tritt nie ein

if not Iseempty then KK(1) := KK(Anzahl+1);
 sink(1, Anzahl); end if;

HeapIndex(Wurzel.Kn) := 0;

return Wurzel;

end DeleteMin;

-- Man beachte, dass wie üblich die zu entfernenden Elemente nicht gelöscht
-- werden, sondern dass nur der Index "Anzahl", der auf das letzte Element
-- im Heap zeigt, um 1 verringert wird. Die Manipulationen der Menge B
-- ziehen wir heraus; sie werden im Dijkstra-Algorithmus direkt ausgeführt.
-- In KK(0) speichern wir ein "null record", welches immer dann zurück
-- gegeben wird, wenn die Prioritätswarteschlange R leer ist; dieser Fall tritt
-- aber wegen while R $\neq \emptyset$ do (= while not Iseempty do) nicht ein.

```

procedure Aufsteigen(Los: MaxHeap) is           -- eine Hilfsprozedur
i, j: Natural; v: Item;
begin i := Los; j := i/2;           -- j zeigt auf den Elternknoten von i im Heap
  if j = 0 then HeapIndex(KK(1).Kn) := 1;
  else while j > 0 and then KK(i).Sch < KK(j).Sch do
    v := KK(i); KK(i) := KK(j); KK(j) := v;
    HeapIndex(KK(i).Kn) := j; HeapIndex(KK(j).Kn) := i;
    i := j; j := i/2; end loop;
  end if;
end Aufsteigen;

procedure Insert (K: Item) is
begin                               -- K am Ende einfügen und aufsteigen lassen
  Anzahl := Anzahl+1; KK(Anzahl) := K;
  HeapIndex(K.Kn) := Anzahl;
  Aufsteigen (Anzahl);
end Insert;

```

```
procedure DecreaseKey (K: Item) is  
  IndexK: MaxHeap;  
begin IndexK := HeapIndex(K.Kn);  
    if K.Sch < KK(IndexK).Sch  
      then KK(IndexK) := K;  
      Aufsteigen (IndexK); end if;  
end DecreaseKey;
```

```
begin Empty;                                -- Initialisierung des Pakets  
    for q in Node loop HeapIndex(q) := 0; end loop;  
end package Prio_Queue;
```

Mit diesem Paket lässt sich nun der Dijkstra-Algorithmus als Block (mit einem Unterblock und Prozedur) ausformulieren.

declare

-- erforderliche Vorab-Deklarationen

```
generic ... "generischer Teil wie oben angegeben";  
package Prio_Queue ... "Spezifikation wie oben angegeben";  
package body Prio_Queue ... "wie oben angegeben";  
type Knoten; type NextKnoten is access Knoten;  
type Kante; type NextKante is access Kante;  
type Knoten is record  
    Id: Knotenname;  
    Besucht: Boolean; zahl1, zahl2: Integer;  
    Inhalt: <weitere Komponenten>;  
    NKn: NextKnoten;    -- nächster Knoten (in Knotenliste)  
    EIK: NextKante;     -- erste inzidente Kante  
end record;  
type Kante is record  
    W: <Typ des Gewichts der Kanten>;  
    EKn: NextKnoten;    -- Endknoten dieser Kante  
    NKa: NextKante;     -- nächste Kante (bzgl. Knoten)  
end record;
```

```

N, M: Natural; Anfang: NextKnoten;
Delta: array (Knoten, Knoten) of Float := (others => " $\infty$ ");
procedure Graph_Aufbau(Anker: in out NextKnoten) is
  "Prozedur, die einen Graphen einliest und/oder als Adjazenzliste
  aufbaut, auf die danach Anker zeigt; zugleich werden N und M als
  Zahl der Knoten und der Kanten bestimmt."
end Graph_Aufbau;

function G_korrekt(Anker: NextKnoten) return Boolean is
  ok: Boolean := true; p: Nextknoten; e: Nextkante;
  begin p := Anker;
    while p /= null loop e := p.EIK;
      while e /= null loop
        Delta(p.all, e.EKn.all) := e.W; e:=e.NKa;
        if e.W < 0.0 then ok := false; end if;
      end loop; end loop;
  return ok;
end G_korrekt;

```

G-korrekt überträgt die Kantenmarkierung nach Delta und prüft, ob alle Werte nichtnegativ sind. Man sollte noch prüfen, dass keine Schlingen und Mehrfachkanten vorliegen (selbst einfügen).

begin

-- Beginn des Anweisungsteils für den Graphen

Graph_Aufbau (Anfang); -- jetzt sind N und M bekannt

if not G_korrekt then ... <"Fehlerfall behandeln">;

else -- Deltatabelle ist nun gesetzt

declare -- Beginn des Blockes für Dijkstra-Alg.

package G_Prio_Queue is -- Prio-Warteschlange für Knoten

new Prio_Queue (Knoten, Float, N, "+", "<");

with Ada.Integer_Text_IO; with Ada.Text_IO;

with G_Prio_Queue; use G_Prio_Queue;

Startknoten: Knoten;

Zustand: array (Knoten) of (B, R, U) := (others => U);

 -- gibt an, ob ein Knoten in B, R oder U liegt

D: pqk; -- zum Speichern der Ergebnisse

 -- wegen "use" ist pqk hier sichtbar

Zähler: Natural; -- zählt die Knoten in B

```

procedure Dijkstra_with_Heap (Start: Knoten) is
X: Item; Y: Knoten; Edge: NextKante; H: Float;
begin                                     -- entsprechend Verfahren in 11.2.4
    Edge := Start.EIK;                     -- Initialisiere die Menge R (= KK).
    while Edge /= null loop
        -- Alle Nachfolger des Startknotens Start kommen nach R
        Y := Edge.EKn.all;
        Zustand(Y) := R;
        Insert((X.Kn, Y, Delta(Start, Y)));
        Edge := Edge.NKa;
    end loop;
    Zustand(Start) := B;
    D(1) := (Start, Start, 0.0);
    Zähler := 1;

```

```

while not Iseempty loop                                     -- solange R nicht leer ist
    X := DeleteMin; Zustand(X.Kn) := B;
    Zähler := Zähler+1; D(Zähler) := X;
    Edge := X.Kn.EIK;
    while Edge /= null loop
        Y := Edge.EKn.all;
        if Zustand(Y) /= B then
            H := X.Sch + Delta(X.Kn, Y.all);
            if Zustand(Y) = U then
                Insert((Y, X, H)); Zustand(Y) := R;
            else DecreaseKey((Y, X, H));
            end if;
            Edge := Edge.NKa;
        end loop;
    end loop;
end Dijkstra_with_Heap;

```

begin

Startknoten := <"wähle einen Knoten aus">;

Dijkstra_with_Heap (Startknoten);

for i in 1..Zähler loop

Put(D(i).Kn.Id); Put(D(i).Vor.Id); Put(D(i).Sch);

end loop;

...

end; -- Ende des inneren Blocks für den Dijkstra-Alg.

end if;

...

end; -- Ende des äußeren Blocks der Vorab-Deklarationen

11.2.8 Hinweise

1. Obiger Algorithmus wurde nicht getestet und kann daher (und wird vermutlich auch) noch einige Flüchtigkeitsfehler enthalten.
2. Da im Verfahren stets der Knoten X mit der geringsten Entfernung ausgewählt wird, gehört der Dijkstra-Algorithmus zu den Greedy-Verfahren, siehe 6.7.
3. Eine andere Implementierung (mit den sog. Fibonacci-Heaps) bringt die bessere Zeitkomplexität $O(m + n \cdot \log(n))$; allerdings werden die Konstanten hierbei größer, sodass sich diese Datenstruktur für die Praxis oft nicht lohnt.

4. Dringende Empfehlung für alle: "Opfern" Sie in der vorlesungsfreien Zeit drei Tage nur zu dem Zweck, den Dijkstra-Algorithmus in Ada zum Laufen gebracht zu haben. Anders werden Sie die Schwierigkeiten, die bereits mit dem "Programmieren im Kleinen" verbunden sind, kaum verinnerlichen. Auch erhalten Sie ein Gespür dafür, wie mühsam es ist, für die Praxis taugliche Bibliotheksprogramme (oder objektorientierte Klassen) zu erstellen.

5. Für die Berechnung der kürzesten Wege zwischen allen Knotenpaaren verwendet man den sehr leicht zu implementierenden Floyd-Algorithmus. Dieser ist dem "Warshall-Algorithmus" zur Berechnung der transitiven Hülle (6.5.4) sehr ähnlich, siehe Übungen oder Literatur.

11.3 Minimale Spannbäume

Gegeben sei ein ungerichteter Graph $G=(V, E, \delta)$ mit $\delta: E \rightarrow \mathbb{R}$ (= Menge der reellen Zahlen).

Gesucht wird ein minimaler Spannbaum, also ein Baum $B=(V, E_B, \delta)$ mit gleicher Knotenmenge, der Teilgraph von G ist und dessen Gewicht $\delta(B)$ minimal ist bzgl. aller Spannbäume von G (siehe Def. 8.8.6).

Zur Lösung dieses Problems verwendet man den Prim-Algorithmus mit der gleichen Zeitkomplexität des Dijkstra-Algorithmus $O((n+m) \cdot \log(n))$ oder den Kruskal-Algorithmus mit der Zeitkomplexität $O(m \cdot \log(m))$.

J.B.Kruskal und R.C.Prim veröffentlichten ihre Algorithmen 1956 bzw. 1957.

11.3.1 Der Algorithmus von Prim

Der Prim-Algorithmus arbeitet genau wie der Dijkstra-Algorithmus, jedoch mit einem modifizierten Schlüssel Sch , der die Reihenfolge in der Prioritätswarteschlange festlegt. Während *bei Dijkstra* für alle $y \in R$ zu jedem Zeitpunkt gilt

$$Sch(y) = \text{Min} \{ Sch(v) + \delta((v,y)) \mid \text{für alle Knoten } v \in B \},$$

verwendet man *beim Prim-Algorithmus* den kleinsten Abstand von y zu irgendeinem Knoten aus B , d. h., für alle $y \in R$ lautet jetzt der Schlüssel Sch zu jedem Zeitpunkt

$$Sch(y) = \text{Min} \{ \delta((v,y)) \mid \text{für alle Knoten } v \in B \}.$$

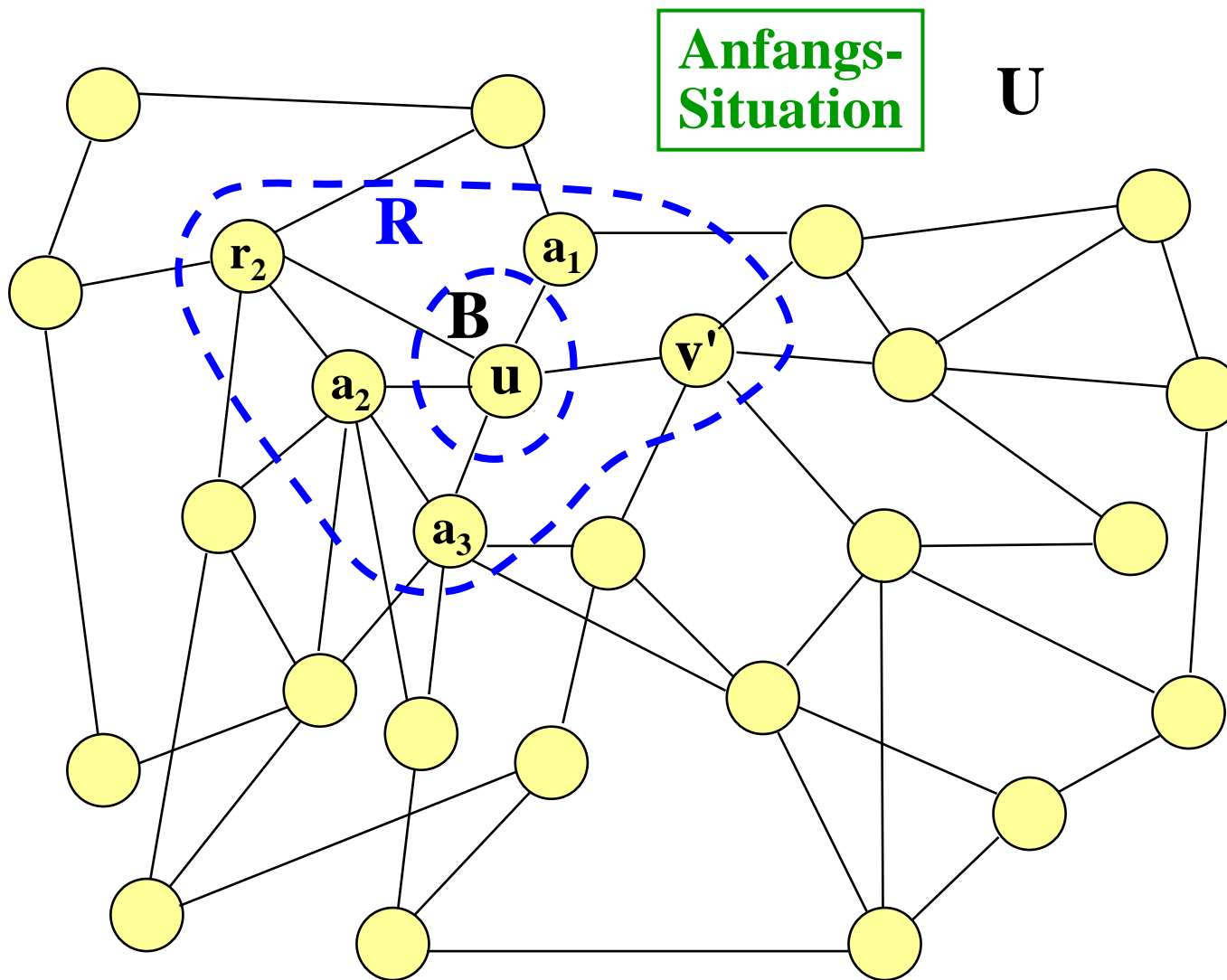
Mit dieser geringen Änderung geht das Programm 11.2.7 in den Prim-Algorithmus über. Auch hier entsteht ein Baum, da in die Menge B jeder Knoten mit genau einer Kante (zu seinem jeweiligen Vorgänger in B) aufgenommen wird.

Genauer: Nur in der Prozedur Dijkstra_with_Heap ist die Zeile
"H := X.Sch + Delta(X.Kn, Y.all);"
zu ersetzen durch
H := Delta(X.Kn, Y.all);

Die Initialisierung der Schlüsselwerte im Aufruf "Insert" zu Beginn der Prozedur Dijkstra_with_Heap erfolgt mit den Delta-Werten und ist daher bereits korrekt.

Da der Graph beim Prim-Algorithmus ungerichtet, beim Dijkstra-Algorithmus dagegen gerichtet ist, muss jede ungerichtete Kante durch zwei gerichtete Kanten (mit gleichem δ -Wert) dargestellt sein, wenn wir das Programm ohne weitere Änderungen übernehmen wollen!

Wir demonstrieren den Prim-Algorithmus an einem Beispiel.



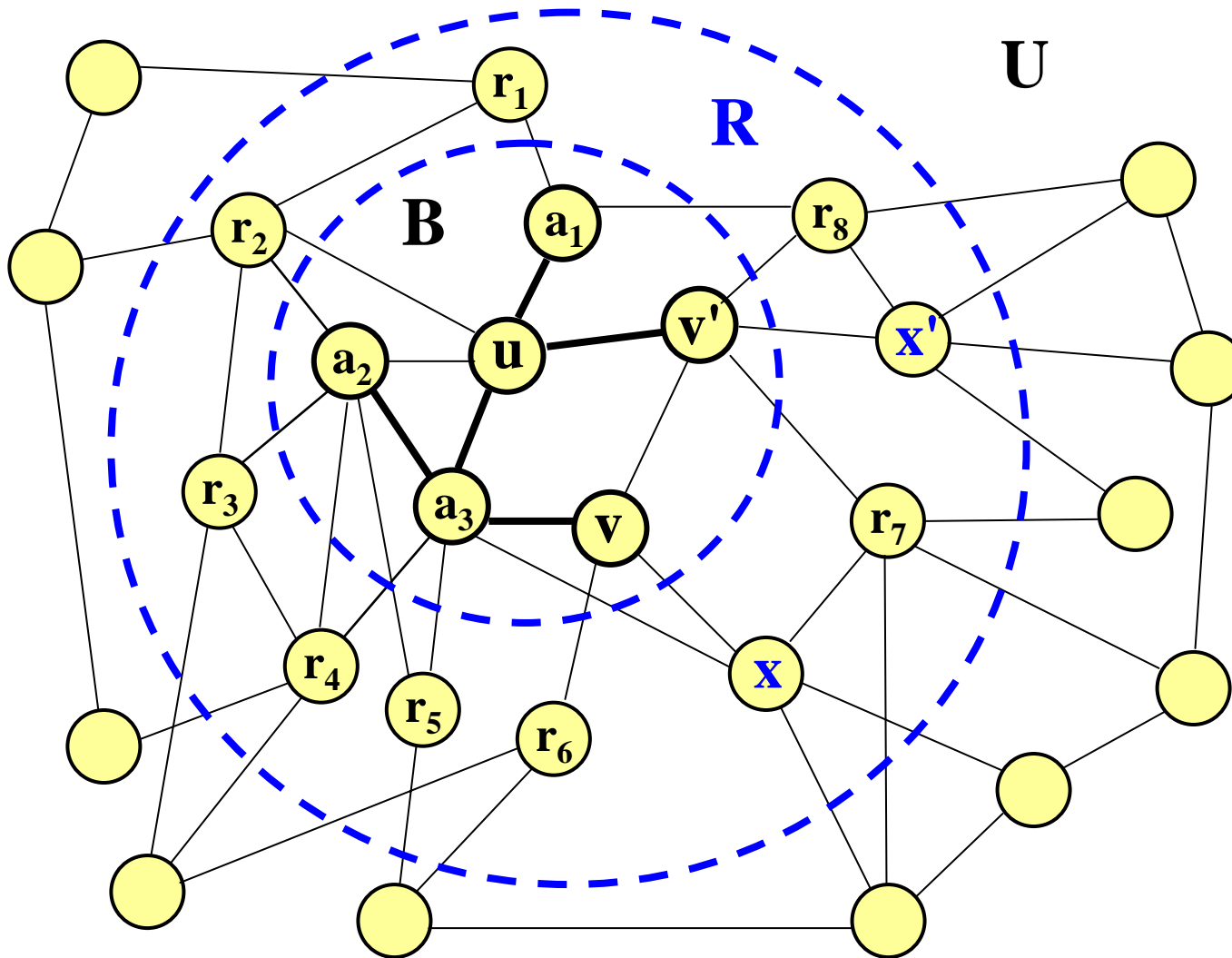
Jede ungerichtete Kante liegt im Programm als zwei gerichtete Kanten vor.

Die Werte der Kantenmarkierung δ sind hier nicht eingetragen.

Beachte: Die Menge der Kanten, die zwischen B und R verlaufen, trennt die Mengen B und V-B, d.h., jeder Weg von einem Knoten aus B zu einem Knoten aus V-B führt über mindestens eine solche Kante.

Ein Beispiel-Graph $G = (V, E, \delta)$ mit Startknoten u .

Anfangs werden $B = \{u\}$ und $R = \{r_2, a_1, a_2, a_3, v'\}$ = Menge der Nachbarknoten von u gesetzt. U = Menge der restlichen Knoten.

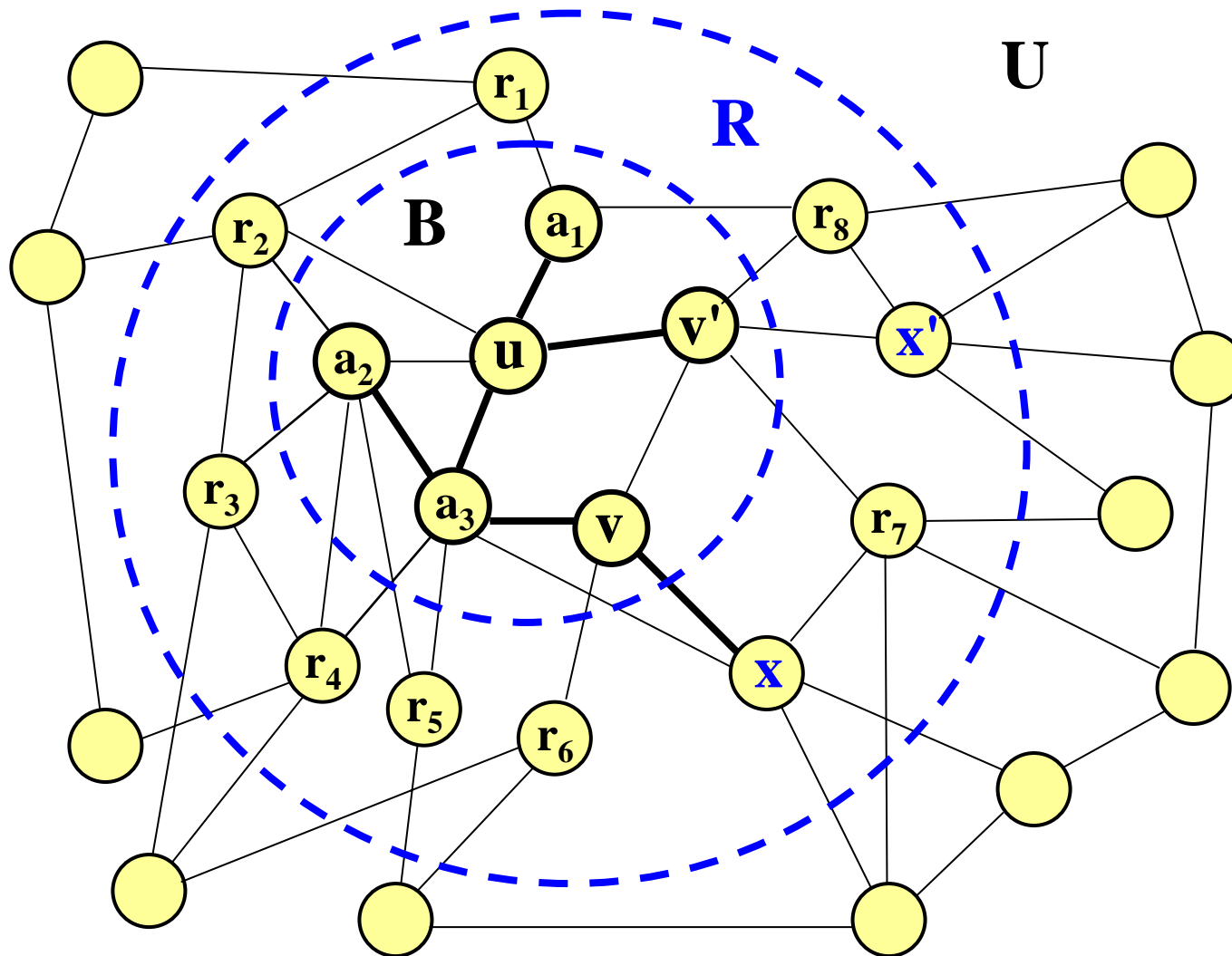


Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun $\{v, x\}$.

Betrachte eine aktuelle Situation beim Prim-Algorithmus:

$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,

U = Menge der restlichen Knoten (diese haben hier keine Namen).

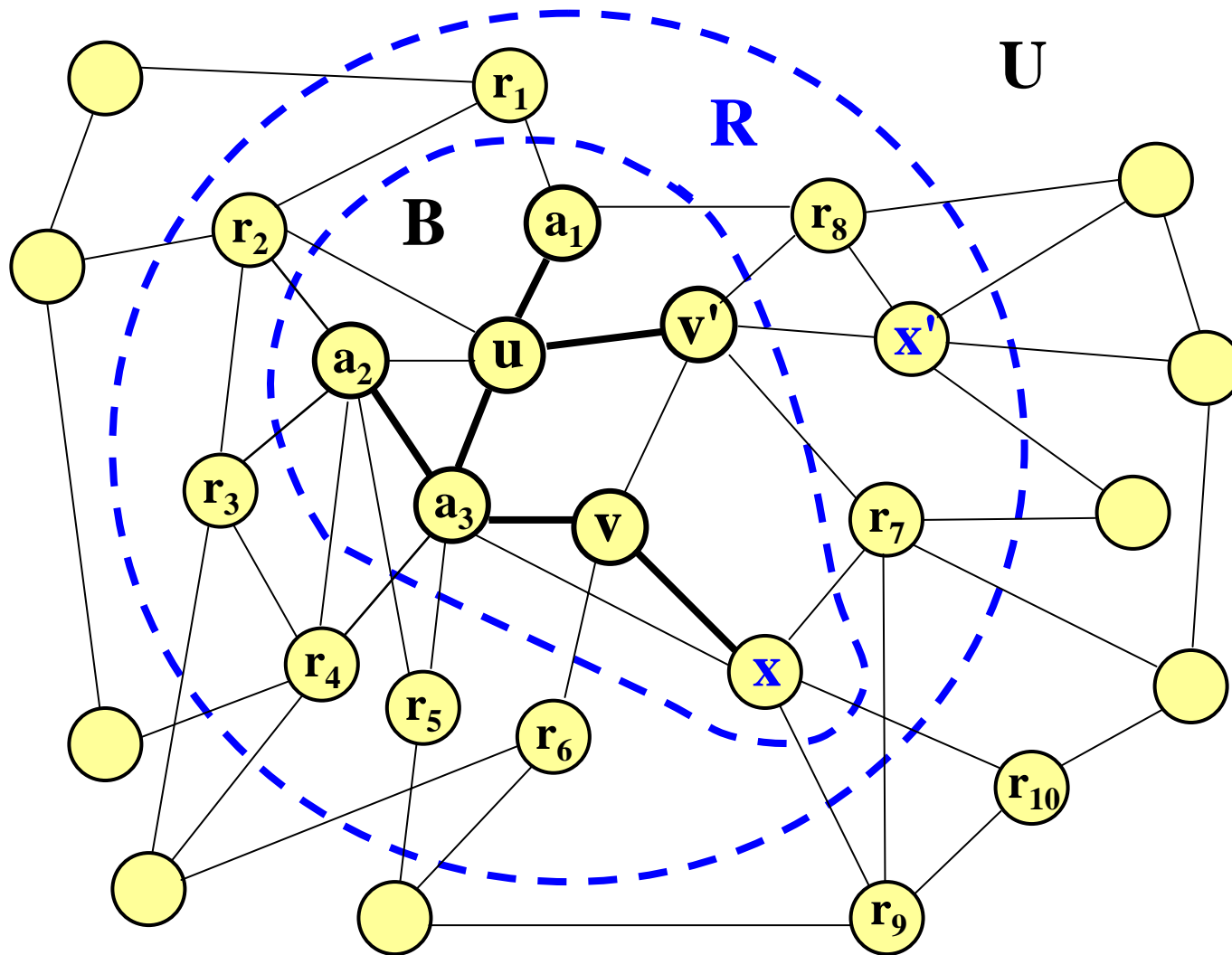


Der bereits erstellte Teil eines minimal spannenden Baums ist mit **fetten Kanten** und **fett umrandeten Knoten (B)** dargestellt. **Die kleinste Kante sei nun $\{v, x\}$.** Dann wird x zusammen mit dem Verweis auf v und $\delta(\{v, x\})$ in B aufgenommen und die mit x adjazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüsselwert Sch wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation beim Prim-Algorithmus:

$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,

U = Menge der restlichen Knoten (diese haben hier keine Namen).



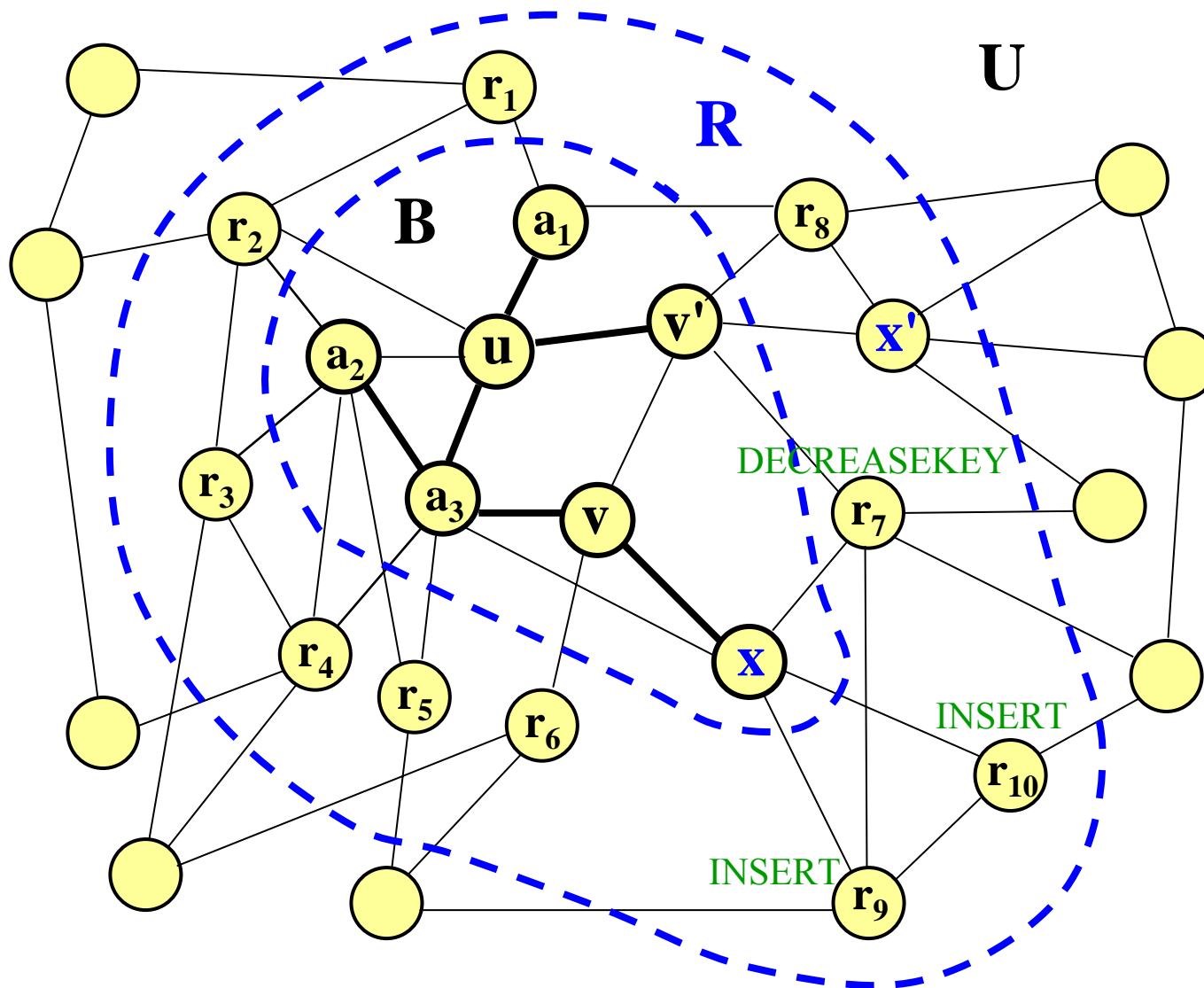
Der bereits erstellte Teil eines minimal spannenden Baums ist mit **fetten Kanten** und **fett umrandeten Knoten (B)** dargestellt. Die **kleinste Kante** sei nun $\{v, x\}$.

Dann wird x zusammen mit dem Verweis auf v und $\delta(\{v, x\})$ in B aufgenommen und die mit x adjazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüsselwert Sch wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation beim Prim-Algorithmus:

$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,

U = Menge der restlichen Knoten (diese haben hier keine Namen).

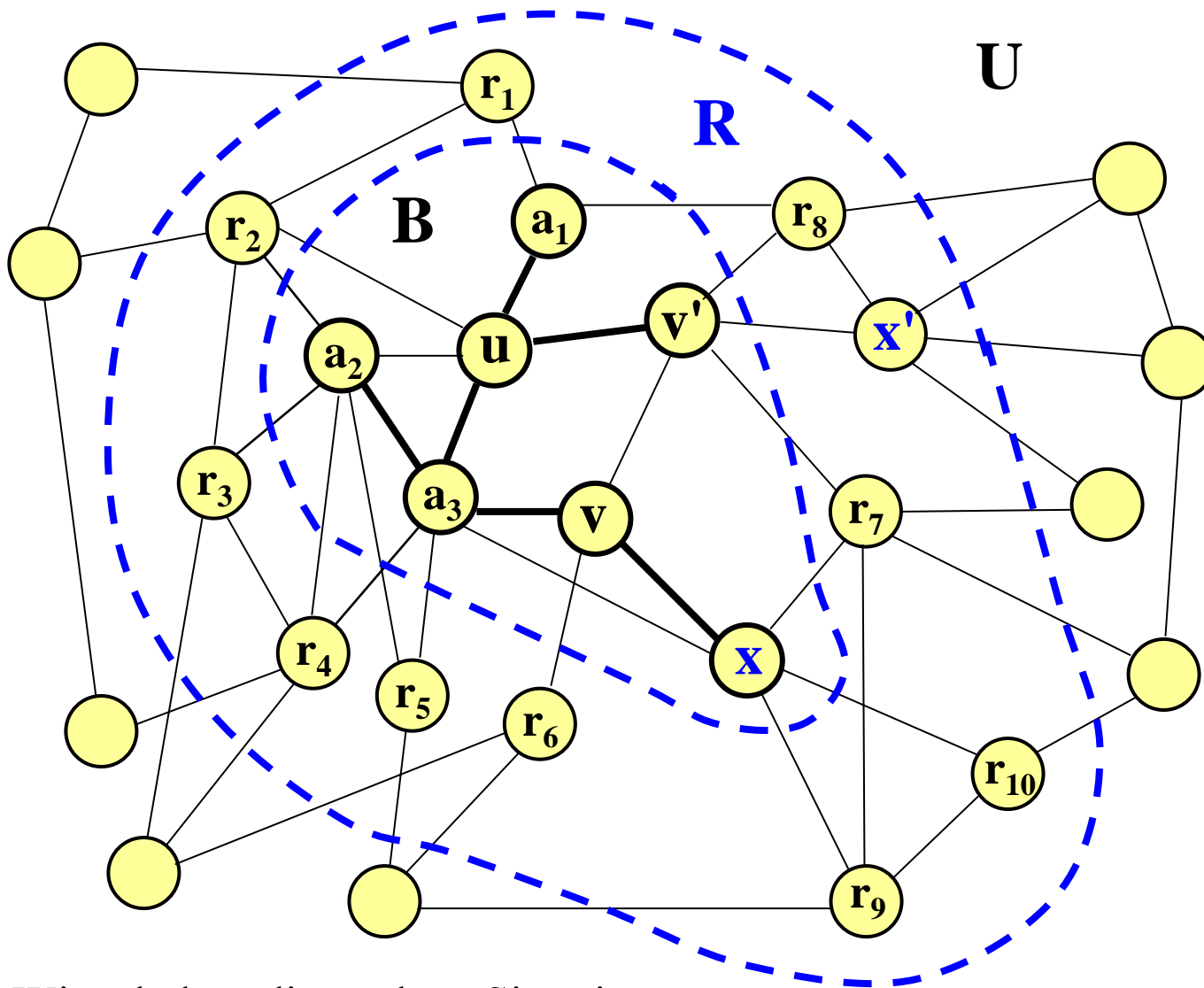


Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun $\{v, x\}$. Dann wird x zusammen mit dem Verweis auf v und $\delta(\{v, x\})$ in B aufgenommen und die mit x adjazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüsselwert Sch wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation:

$B = \{u, a_1, a_2, a_3, v, v'\}$, $R = \{x, x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$,

U = Menge der restlichen Knoten (diese haben hier keine Namen).



Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante sei nun ...

Wir erhalten die nächste Situation:

$B = \{u, a_1, a_2, a_3, v, v', x\}$, $R = \{x', r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$,

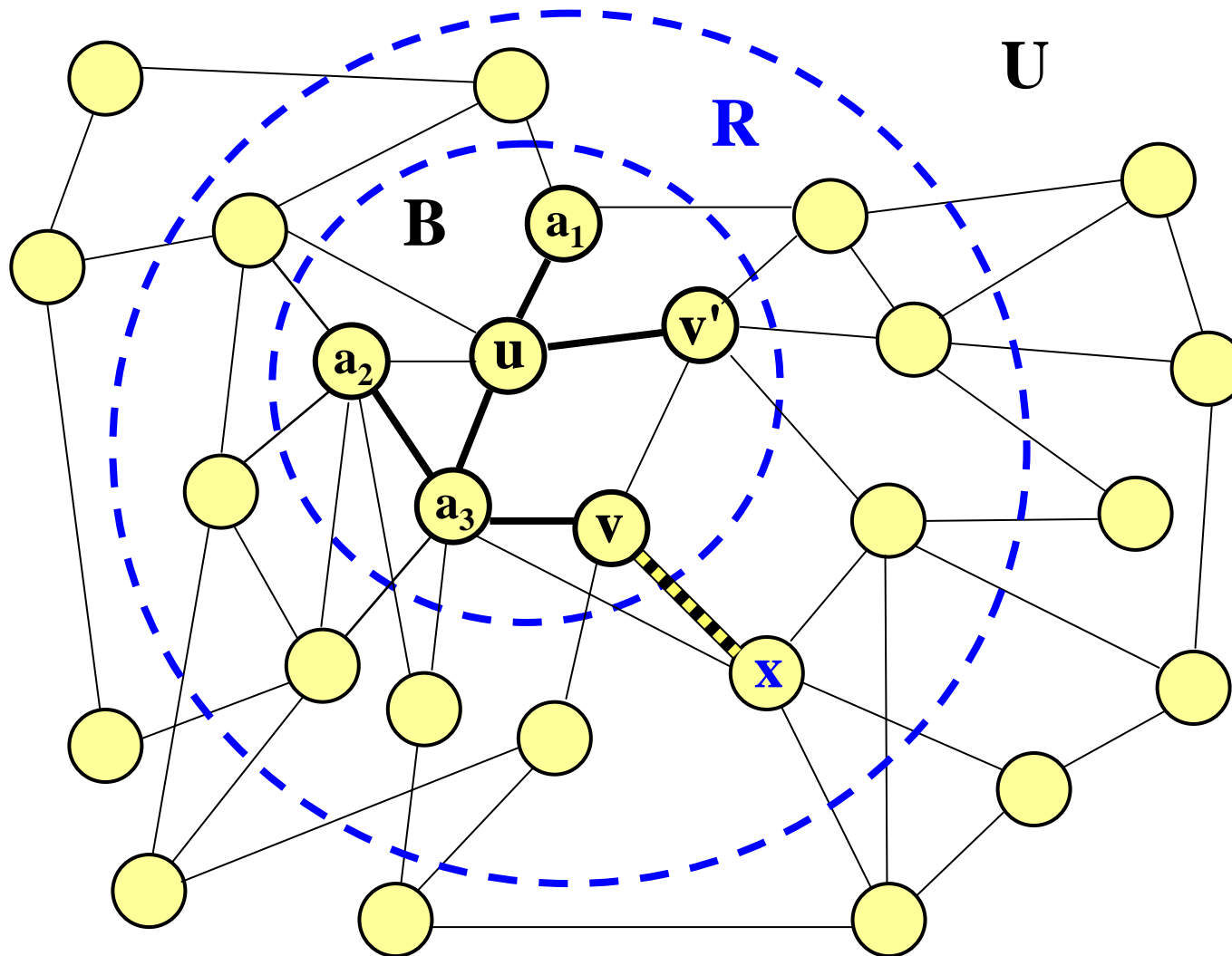
U = Menge der restlichen Knoten (diese haben hier keine Namen).

11.3.2 Wie kann man beweisen, dass dieser Algorithmus tatsächlich einen minimalen Spannbaum konstruiert?

Mit Hilfe der Eigenschaft, dass die Kanten zwischen B und R die Mengen B und V-B trennen, geht dies sehr einfach (siehe Folie mit der Anfangssituation oben).

Korrektheitsbeweis: Zunächst sei der Graph zusammenhängend, denn sonst besitzt er keinen Spannbaum.

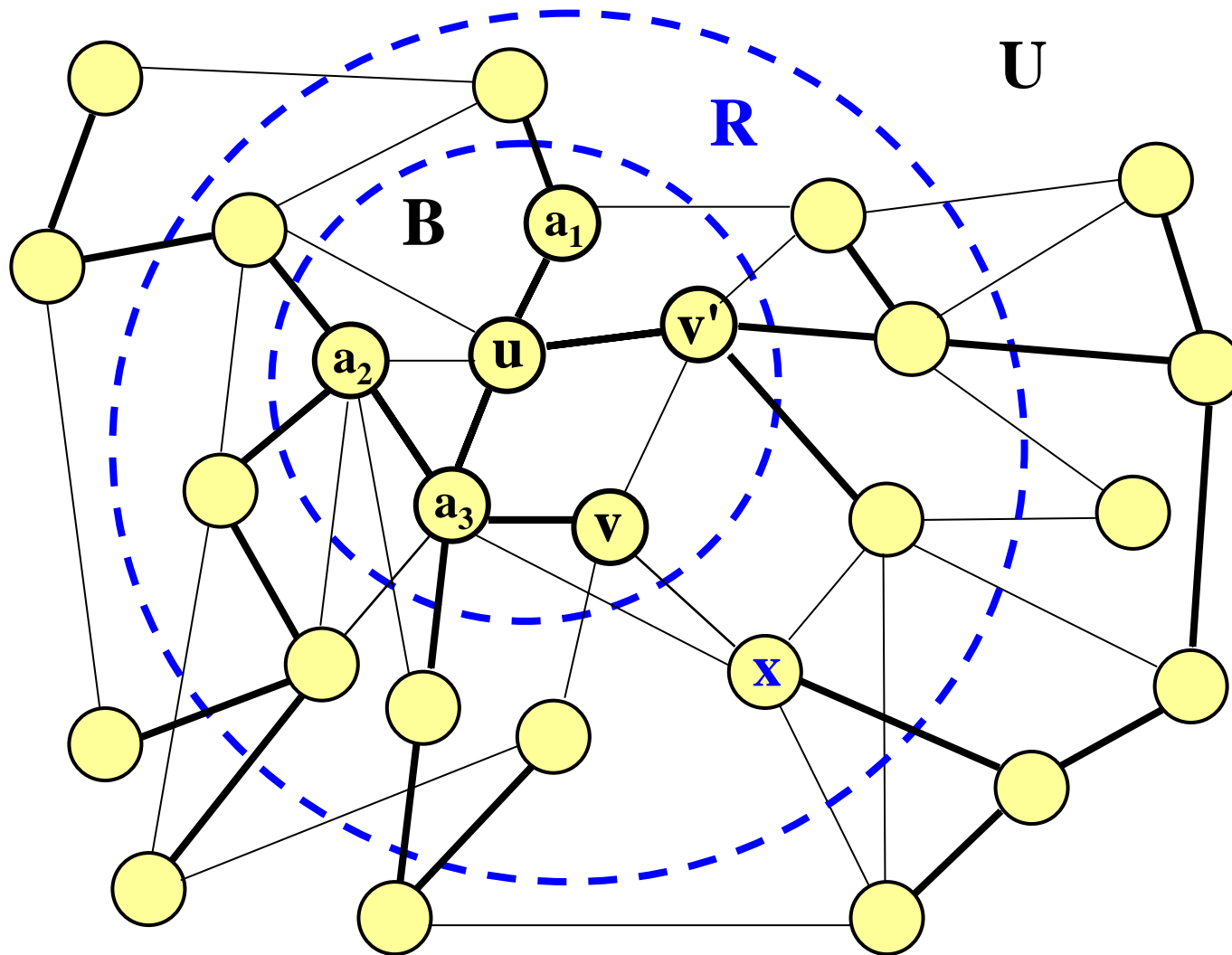
Annahme, wir hätten für den minimalen Spannbaum eine kleinste Kante nicht verwenden dürfen. D.h., in keinem minimalen Spannbaum kommt die Kante $\{v, x\}$ vor, obwohl sie in einer Situation des Prim-Algorithmus kleiner gleich allen anderen Kanten, die von B nach R führten, war und folglich vom Prim-Algorithmus ausgewählt wurde. Wir betrachten eine Situation aus dem obigen Beispiel zur Illustration:



Der bereits erstellte Teil eines minimal spannenden Baums ist mit fetten Kanten und fett umrandeten Knoten (B) dargestellt. Die kleinste Kante von B nach R sei nun $\{v, x\}$.

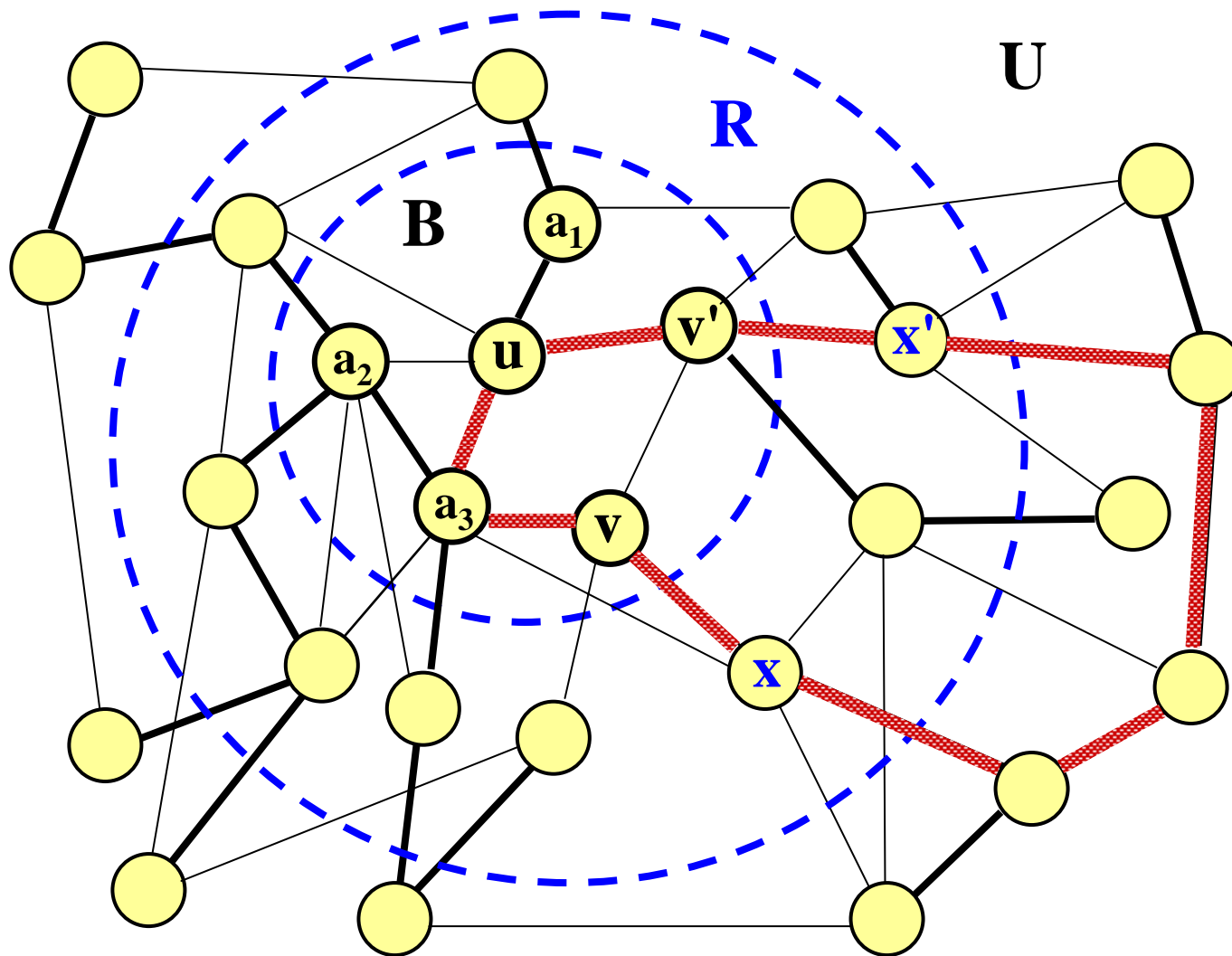
Wir nehmen nun an:
Diese Kante $\{v, x\}$ ist in keinem minimalen Spannbaum enthalten.

In dieser aktuellen Situation des Prim-Algorithmus sei der Schlüssel Sch von x minimal und die Kante $\{v, x\}$ habe den kleinsten δ -Wert aller Kanten zwischen B und R.



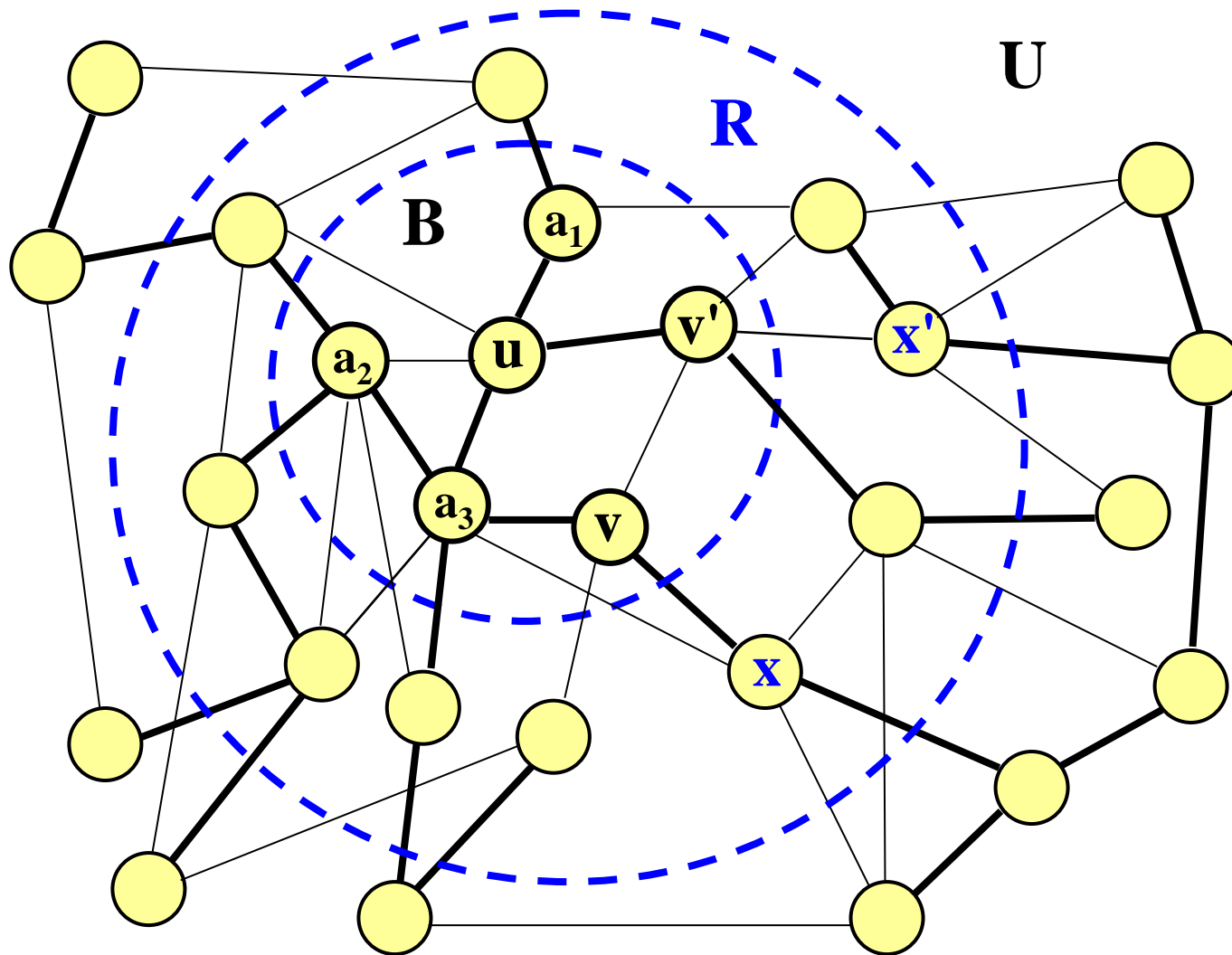
Ein minimaler
Spannbaum wird
mit fetten Kanten
dargestellt.

Sei dies ein minimaler Spannbaum, in dem die Kante $\{v, x\}$ nicht enthalten ist. Fügt man nun die Kante $\{v, x\}$ zu diesem Spannbaum hinzu, so muss ein Zyklus entstehen, auf dem v und x liegen.



Wir haben $\{v, x\}$ hinzugefügt. Der entstandene Zyklus ist mit dickeren Kanten hervorgehoben.

Da v in B und x in $V-B$ lag, muss es genau eine Kante $\{v', x'\}$ auf diesem Zyklus geben, die zum Zeitpunkt, als $\{v, x\}$ ausgewählt wurde, ebenfalls vorhanden war, d.h., es gilt $\delta(\{v, x\}) \leq \delta(\{v', x'\})$.



Wir lassen nun $\{v', x'\}$ weg. Der neue Spannbaum ist mit fetten Kanten dargestellt.

Ersetzt man also $\{v', x'\}$ im minimalen Spannbaum durch $\{v, x\}$, so erhält man wieder einen Spannbaum, dessen Gewicht um den Wert $\delta(\{v', x'\}) - \delta(\{v, x\}) \geq 0$ kleiner ist!

Dieser neue Spannbaum, in dem $\{v, x\}$ vorkommt, besitzt also ein Gewicht, das kleiner oder gleich dem des zuerst betrachteten minimalen Spannbaums ist.

Folglich gibt es doch einen minimalen Spannbaum, der die Kante $\{v, x\}$ enthält, im **Widerspruch zur Annahme**. Daher gibt es also stets einen minimalen Spannbaum, in dem die vom Prim-Algorithmus ausgewählten Kanten vorkommen. Der Prim-Algorithmus liefert also stets einen minimalen Spannbaum. ■

Natürlich kann es mehrere minimale Spannbäume zu einem Graphen geben, wenn es Kanten mit gleichem δ -Wert im Graphen gibt. Dann treten eventuell in der Prioritätswarteschlange mehrere Knoten mit gleichem minimalem Schlüssel *Sch* auf. Egal welchen solchen Knoten man dann wählt, man erhält am Ende stets einen minimalen Spannbaum.

11.3.3 Der Algorithmus von Kruskal

Der Kruskal-Algorithmus sortiert die Kanten bzgl. δ , beginnt dann mit der kleinsten Kante und nimmt jeweils die nächste Kante hinzu, sofern diese Kante keinen Zyklus mit den bereits ausgewählten Kanten bildet. Auf diese Weise entsteht ein minimaler Spannbaum.

Die Zeitkomplexität beträgt $O(m \cdot \log(m))$ mit $m = |E|$.

Dies gilt aber nur bei geeigneter Implementierung, insbesondere muss es eine Liste oder ein Feld der Kanten geben (hier empfiehlt sich eine [Inzidenzlistendarstellung des Graphen](#), in der die Knoten und die Kanten in eigenen Listen hintereinander stehen und Verweise zu den Endknoten von der Kanten- in die Knotenliste existieren).

"Einziges" Problem hierbei:

Wie stellt man fest, ob eine Kante mit den bereits ausgewählten Kanten einen Zyklus bildet?

Lösung: Man bildet ein System von Mengen aus Knoten und fasst jeweils in einer Menge genau die Knoten zusammen, die untereinander durch bereits ausgewählte Kanten verbunden sind.

Für jede Kante prüft man dann, ob ihre Endknoten in der gleichen Menge liegen. Falls ja, so verwirft man die Kante, denn dann bewirkt diese Kante einen Zyklus; falls nein, so nimmt man die Kante zu den Baumkanten hinzu und vereinigt die beiden Mengen, in denen die Endknoten lagen, zu einer neuen Menge.

Kruskal-Algorithmus für einen minimalen Spannbaum:

Gegeben sei ein ungerichteter Graph $G = (V, E, \delta)$.

1. Sortiere alle Kanten nach ihrem δ -Wert.
2. Bilde für jeden Knoten $y \in V$ die Menge $\{y\}$;
BK := \emptyset ; -- dies wird die Menge der Baumkanten
3. while (die Kantenliste ist nicht leer) and ($|BK| < n-1$) do
 Entferne die kleinste Kante e von der Kantenliste;
 if die Endknoten x und y der Kante e liegen in
 verschiedenen Mengen M und N then
 bilde $M := M \cup N$; vergiss N ; $BK := BK \cup \{e\}$ fi
 od;

Die Kanten von BK bilden einen minimalen Spannbaum.

11.3.4 *Korrektheit des Kruskal-Algorithmus:*

Der Beweis, dass dieses Vorgehen einen minimalen Spannbaum liefert, wird ähnlich wie der Beweis zum Prim-Algorithmus geführt. Annahme, eine Kante e , die vom Kruskal-Algorithmus ausgewählt wurde, würde in keinem minimalen Spannbaum sein, dann fügt man sie zu einem minimalen Spannbaum hinzu, wodurch ein Zyklus entsteht, auf dem mindestens eine Kante mit nicht-kleinerem δ -Wert liegen muss; gegen diese tauscht man die Kante e aus und erhält einen minimalen Spannbaum, in dem die Kante e doch vorkommt.

(Hinweis: Gäbe es auf dem Zyklus außer der Kante e nur Kanten mit kleinerem δ -Wert, so könnte die Kante e nicht vom Kruskal-Algorithmus ausgewählt worden sein, da alle anderen Kanten vorher hätten ausgewählt worden sein müssen und e dann einen Zyklus bewirkt hätte.) ■

Der Algorithmus stellt nebenbei auch fest, ob der gegebene Graph zusammenhängend war: Dies trifft genau dann zu, wenn ein Spannbaum konstruiert wird, wenn also am Ende genau $n-1$ Kanten ausgewählt wurden.

Das Vereinigen der beiden Mengen wurde hier so realisiert, dass eine der Mengen (hier: M) neu auf $M \cup N$ gesetzt wird, während die andere Menge entfernt wird. Dies kann man natürlich in dieser Weise nicht implementieren, da das Löschen viel zu viel Aufwand erfordern würde. Vielmehr fasst man die beiden Mengen zusammen und vergisst deren alte Bedeutung.

11.3.5 Der Kruskal-Algorithmus braucht zwei Operationen:

FIND (**x**) = Stelle die Menge fest, in der das Element **x** liegt.

UNION (**M**, **N**) = Vereinige die beiden Mengen **M** und **N**,
nenne das Ergebnis wieder **M** und vergiss **N**.

Hiermit lautet der Kruskal-Algorithmus:

```
Sortiere alle Kanten nach ihrem  $\delta$ -Wert in eine Liste L;  
for all  $y \in V$  do  $M_y := \{y\}$  od; BK := leere Liste;  
while not isempty(L) and ( $|BK| < n-1$ ) do  
     $e := \text{DELETEMIN}(L)$ ; x und y seien die Endknoten von e;  
     $M := \text{FIND}(x)$ ;  $N := \text{FIND}(y)$ ;  
    if  $M \neq N$  then  $\text{UNION}(M, N)$ ; füge e zu BK hinzu fi  
od;
```

Das sog. **UNION-FIND-Problem** fragt nach einer effizienten Implementierung der Operationen UNION und FIND unter der Nebenbedingung, dass je $O(n)$ dieser Operationen auf einer Gesamtmenge von n Elementen durchgeführt werden müssen.

Als gute Implementierung gilt die Darstellung jeder Menge als Baum, dessen Kanten von jedem Knoten zu seinem Vorgänger gerichtet sind und in dessen Wurzel zusätzlich die Anzahl der Elemente dieses Baums vermerkt ist (s.u.). Dann lässt sich **FIND(x)** als Bestimmung der Wurzel des Baums, in dem sich x befindet, und

UNION(M,N) als Anhängen des kleineren Baums an die Wurzel des größeren Baums realisieren.

$\text{FIND}(x) \neq \text{FIND}(y)$ läuft dann auf die einfache Abfrage, ob beide Wurzeln der gleiche Knoten sind, hinaus.

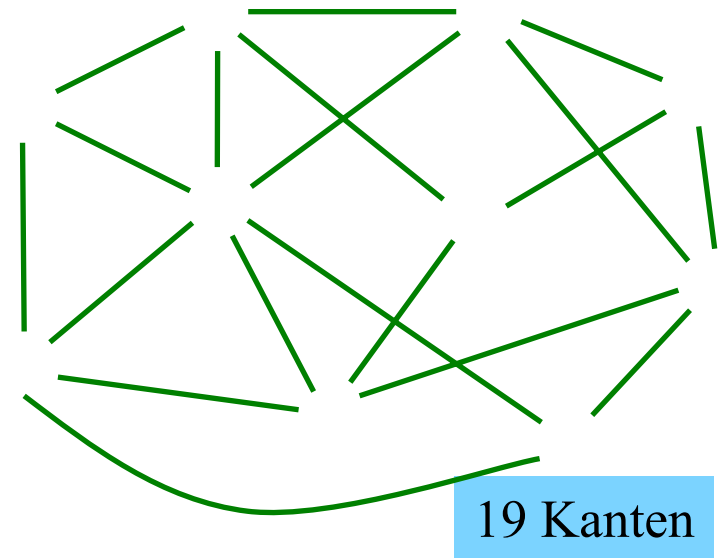
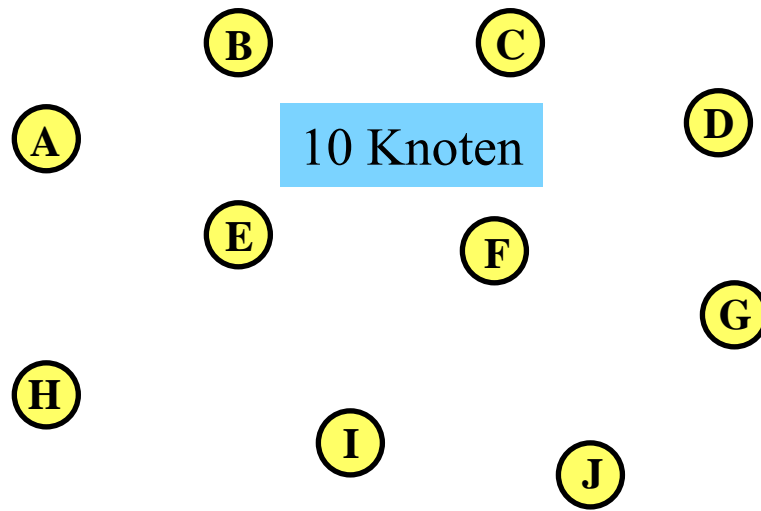
Mit dieser Darstellung benötigt jedes FIND (x) höchstens $\log(n)$ Vergleiche (warum? Beachten Sie, dass stets der kleinere an den größeren Baum gehängt wird), und UNION kann in konstant vielen Schritten ausgeführt werden. Der eigentliche Kruskal-Algorithmus benötigt also höchstens $O(m \cdot \log(n))$ viele Schritte, weil die while-Schleife bis zu m Mal durchlaufen wird und jedes FIND höchstens $\log(n)$ Schritte erfordert.

Nach Kapitel 10 kostet das Sortieren der m Kanten $O(m \cdot \log(m))$ Schritte. In der Praxis ist n in der Regel kleiner als m , daher wird die Komplexität des Kruskal-Algorithmus durch das Sortieren der m Kanten dominiert. Insgesamt ergibt sich in dem normalen Fall $n \leq m$ somit eine Zeitkomplexität von $O(m \cdot \log(m) + m \cdot \log(n)) = O(m \cdot \log(m))$.

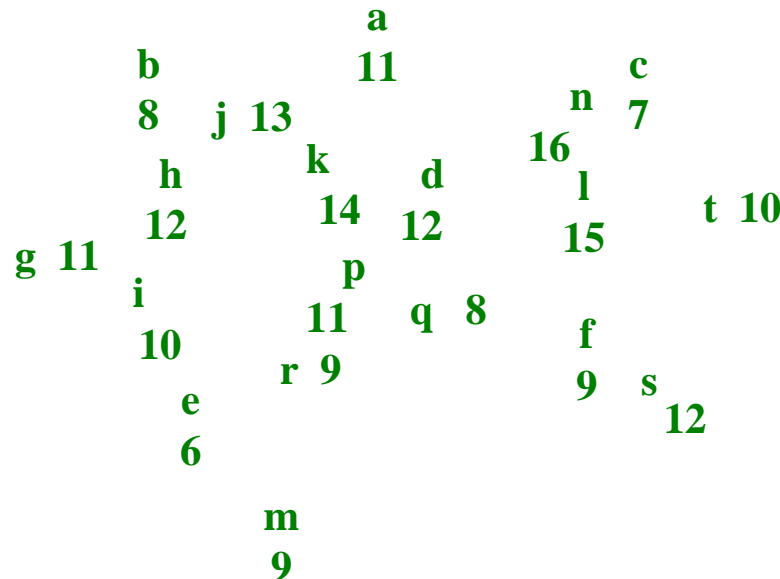
Hinweis:

"Verflacht" man die Bäume bei jeder FIND-Operation (sog. "Pfadkompression"), dann braucht man für alle UNION- und FIND-Operationen nur $O(m \cdot \log^*(n))$ Schritte. (Siehe Vorlesungen zur Algorithmik im weiteren Studium oder siehe Literatur; \log^* wurde in 6.1.4 behandelt.)

Die Zeitkomplexität wird hierdurch allerdings nicht verringert, da das Sortieren der Kanten unverändert $O(m \cdot \log(m))$ Schritte erfordert.

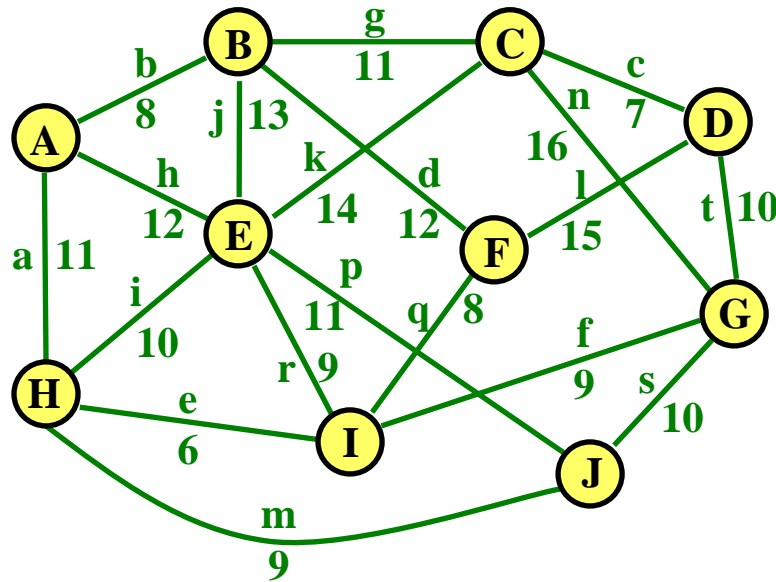


11.3.6 Erläuterung der Datenstrukturen am Beispiel.



Benennung der
19 Kanten und
Gewichte $\delta : E \rightarrow \mathbb{R}$

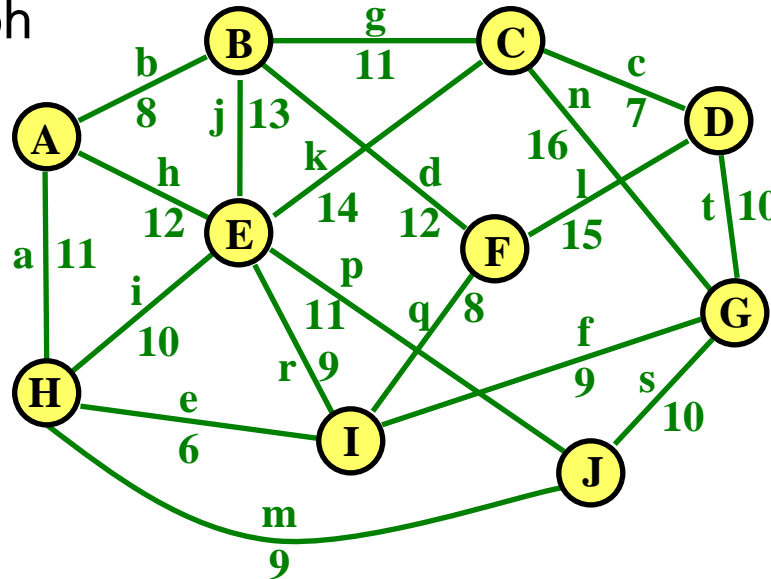
Der Graph $G = (V, E, \delta: E \rightarrow \mathbb{R})$:



Erste Aufgabe: Bilde die sortierte Liste der Kanten. Ergebnis:

$(e,6), (c,7), (b,8), (q,8), (f,9), (m,9), (r,9), (i,10), (s,10), (t,10),$
 $(a,11), (g,11), (p,11), (d,12), (h,12), (j,13), (k,14), (l,15), (n,16).$

Graph

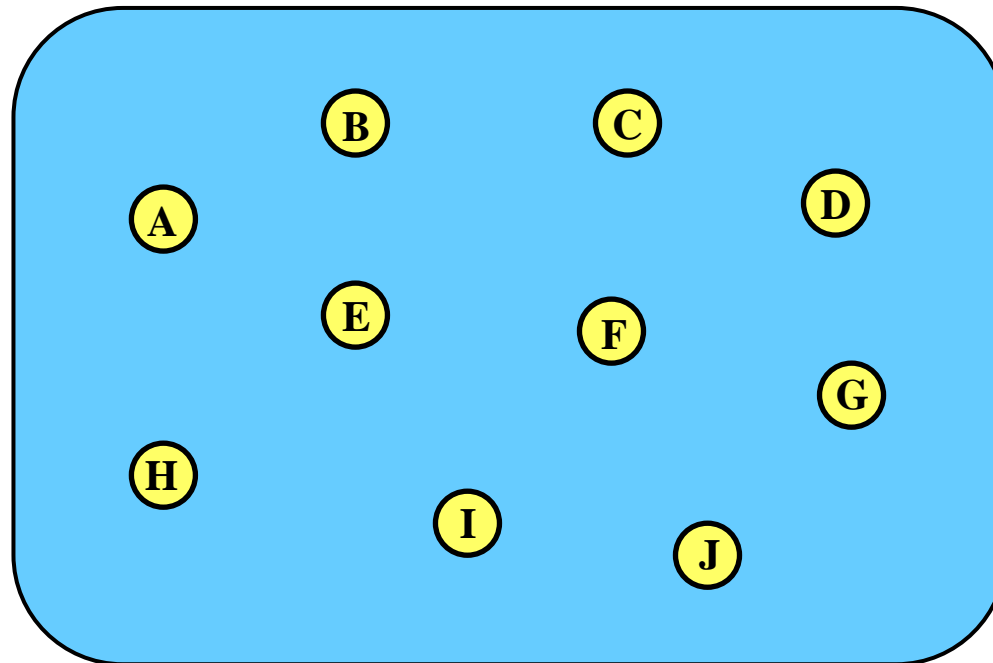


Sortierte
Kantenliste

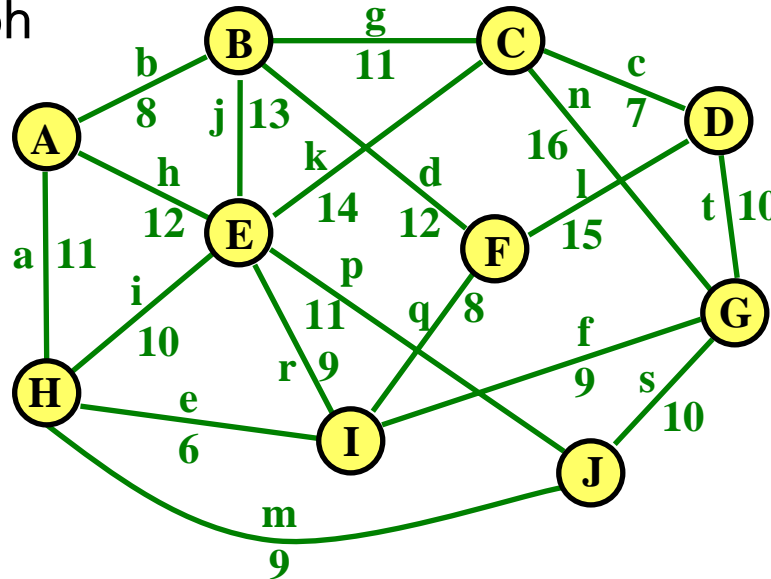
(e,6),
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

"Arbeitsfläche":
Ein Wald isolierter Knoten

Bilde nun die
einelementigen
Mengen M_y für
jeden Knoten y :



Graph



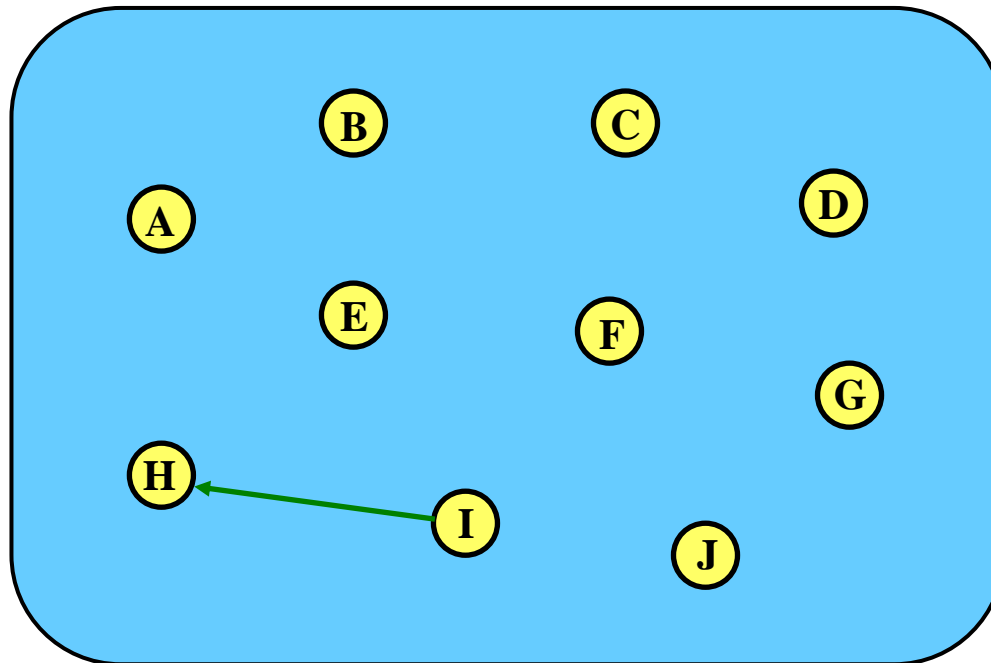
Sortierte
Kantenliste



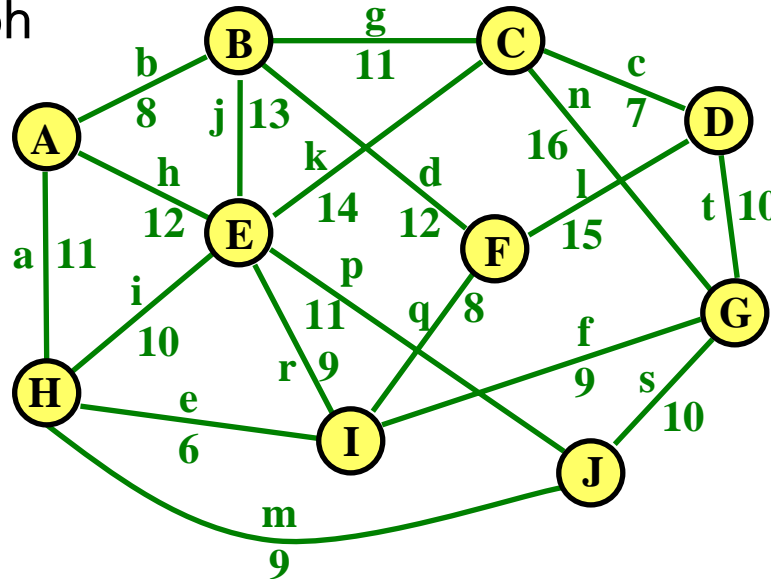
(e,6),
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Im Wald werden zwei
Knoten zu einem Baum
zusammengefasst.

Betrachte die
erste Kante der
sortierten Liste,
also (e,6):
(Die Ausrichtung der
Kanten ist bei den
Mengen zunächst
willkürlich, später
wird stets der kleine-
re an den größeren
Baum gehängt.)



Graph

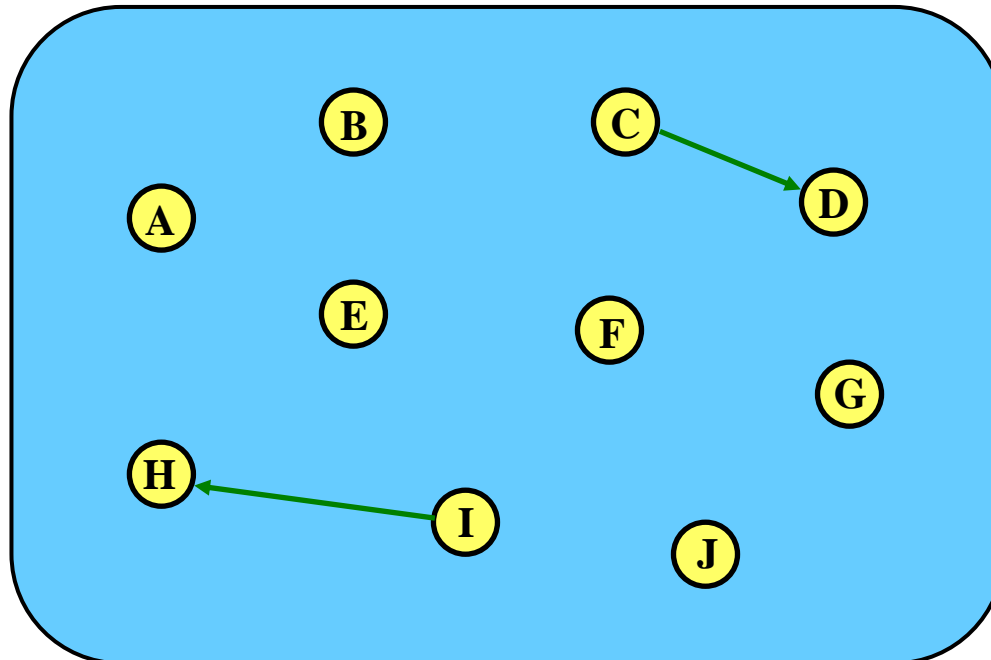


Sortierte
Kantenliste

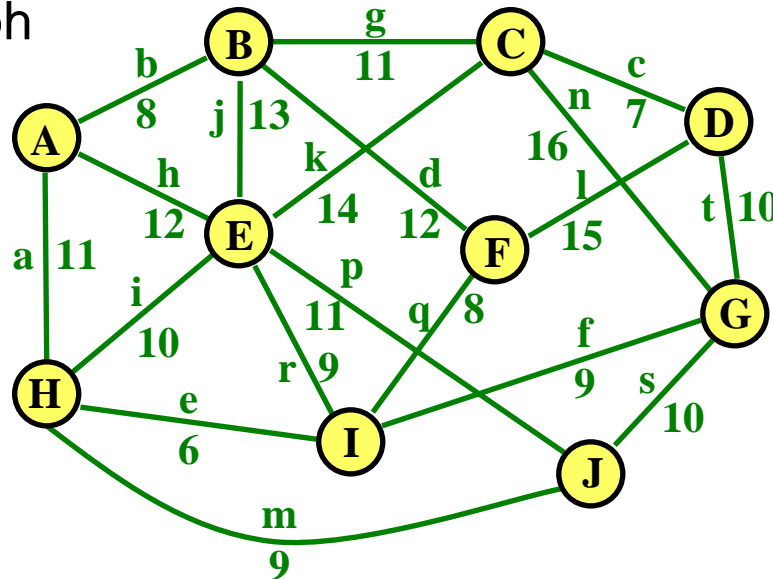


(e,6), +
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun
die zweite Kante
der sortierten
Liste, also (c,7):
(Die Ausrichtung der
Kanten ist bei den
Mengen zunächst
willkürlich, später
wird stets der kleiner-
e an den größeren
Baum gehängt.)



Graph

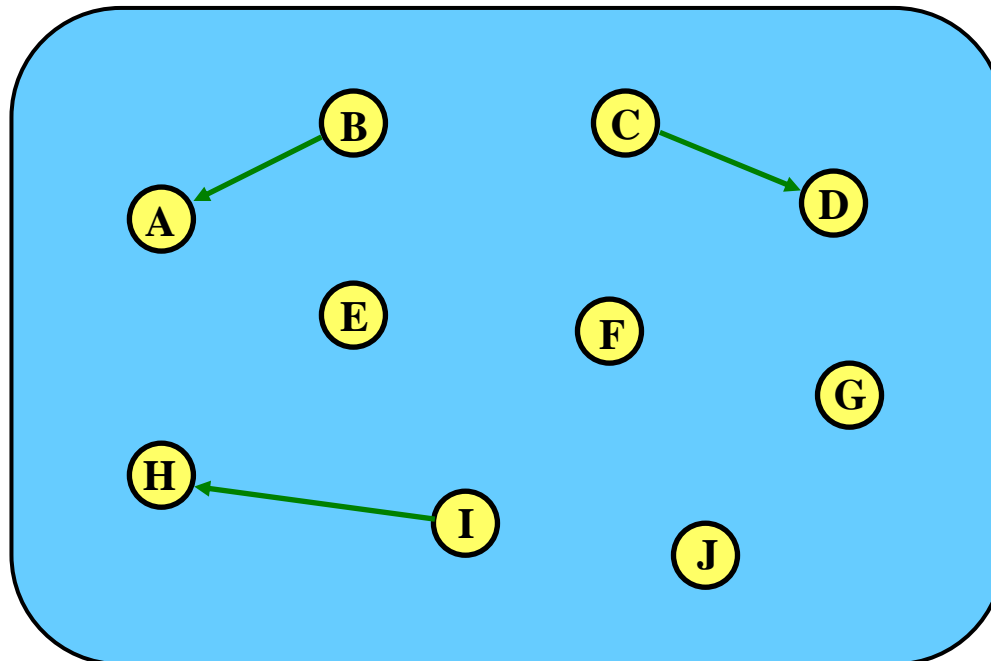


Sortierte
Kantenliste

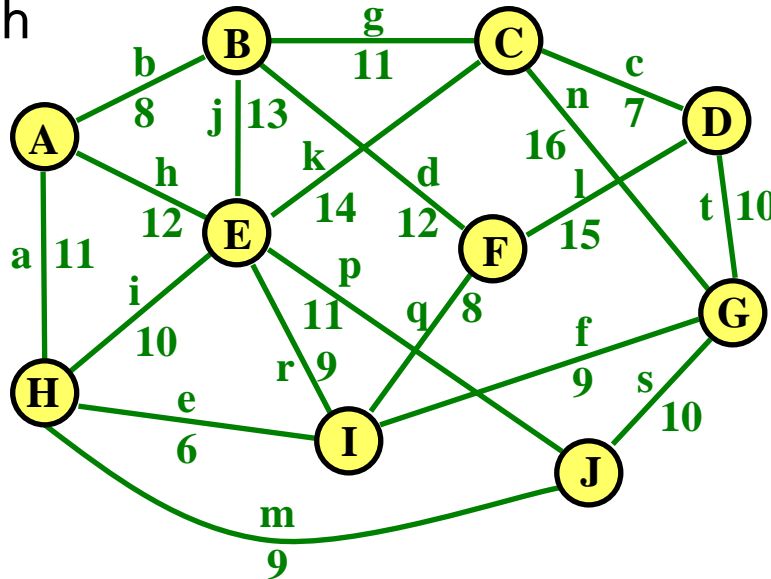


(e,6), +
(c,7), +
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun
die dritte Kante
der sortierten
Liste, also **(b,8)**:



Graph

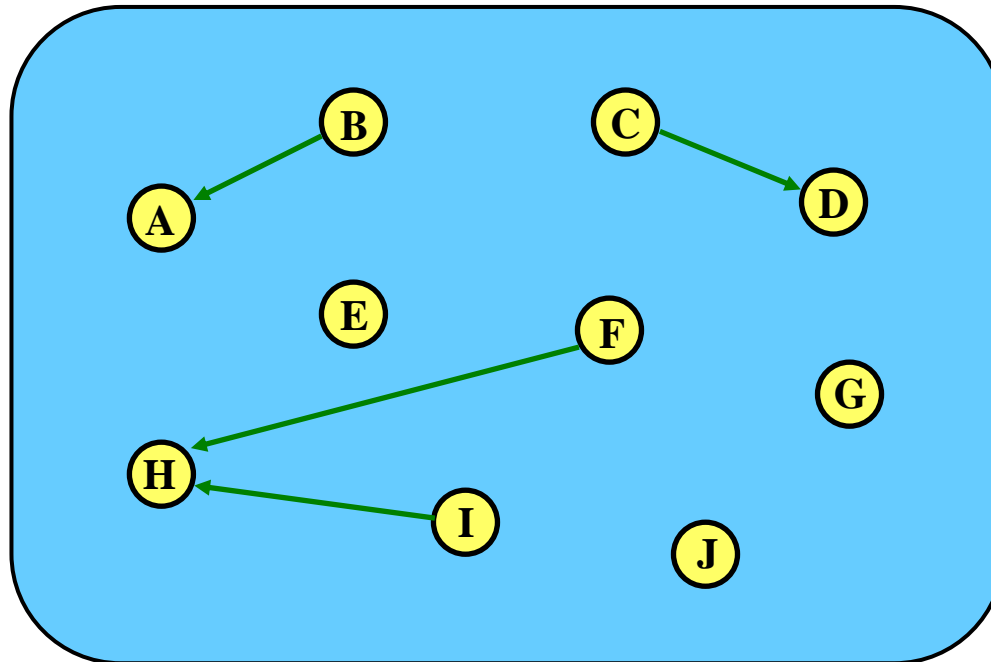


Sortierte
Kantenliste

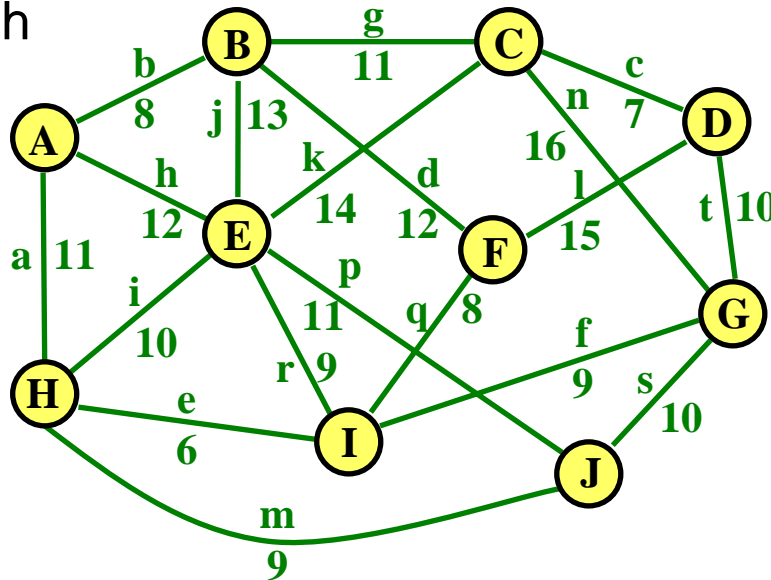
(e,6), +
(c,7), +
(b,8), +
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



Betrachte nun
die vierte Kante
(q,8). Hänge
den kleineren
Baum "F"
an
den größeren,
der zu I gehört.



Graph

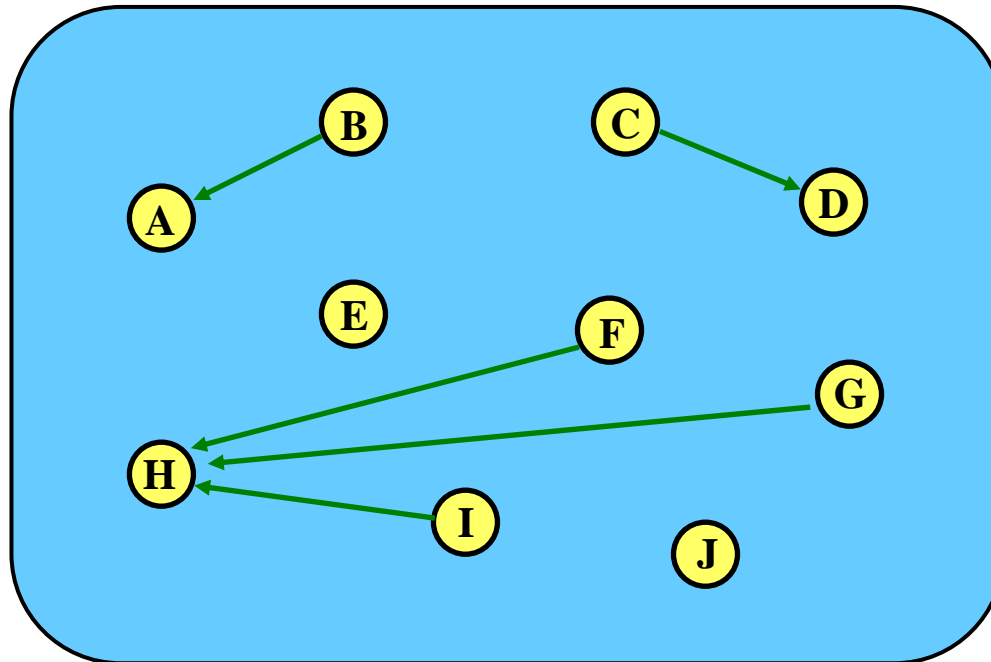


Sortierte
Kantenliste

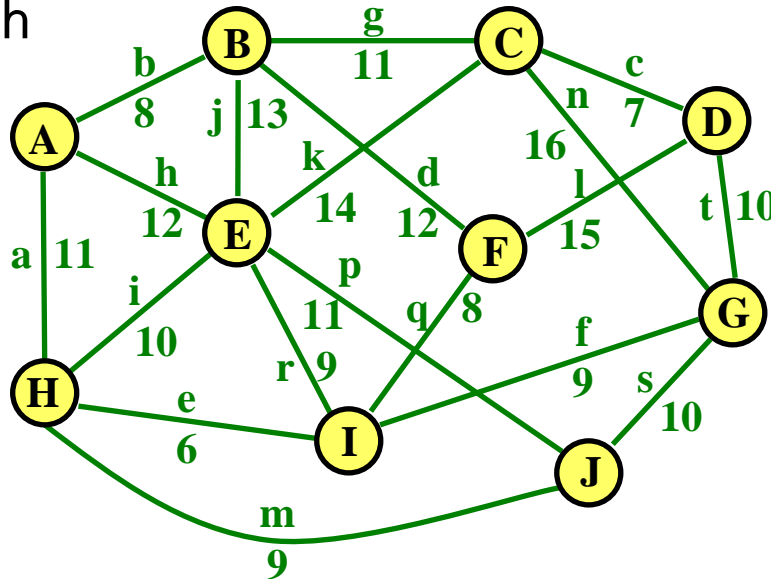
(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



Betrachte nun
die fünfte Kante
(f,9). Hänge
den kleineren
Baum "G"
an den größeren,
der zu I gehört.



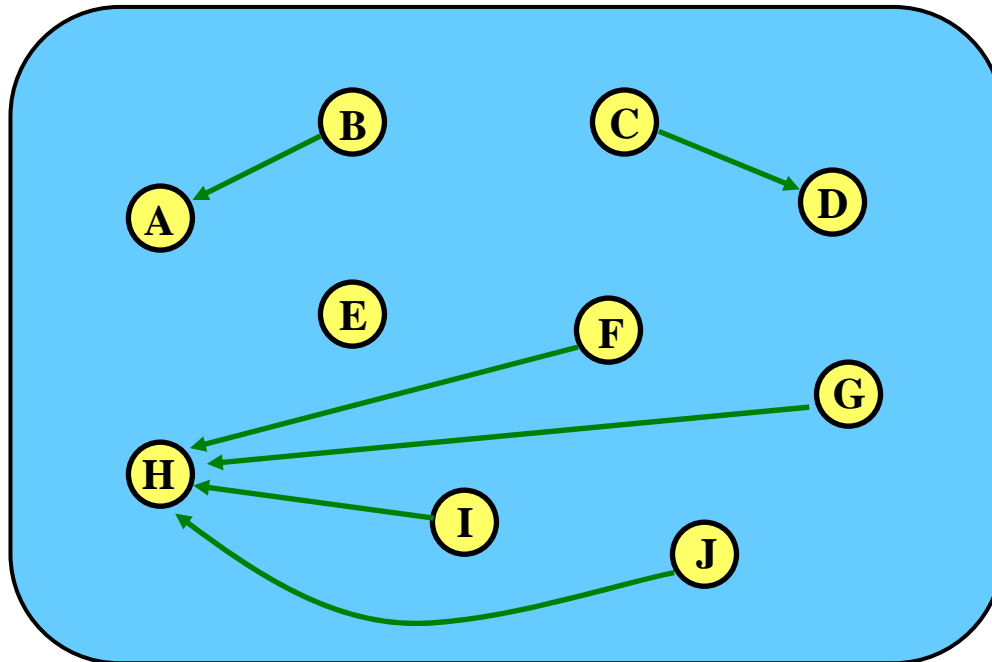
Graph



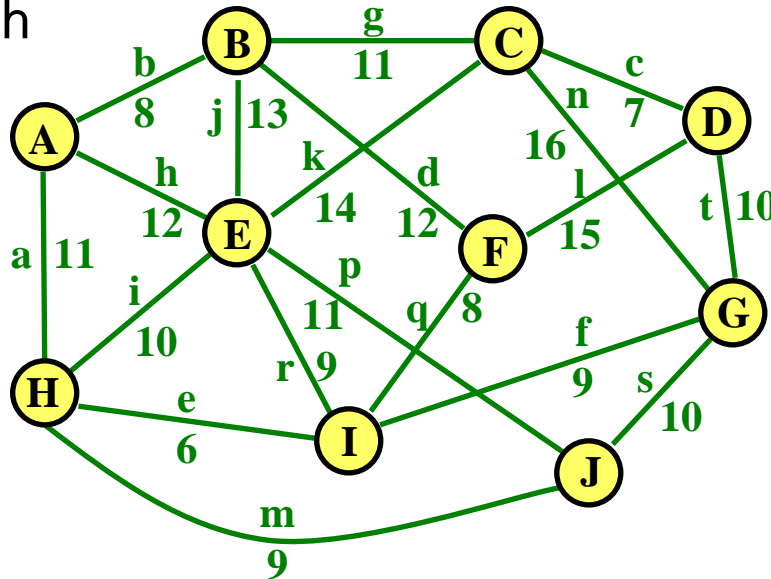
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die sechste Kante **(m,9)**.
Hänge den kleineren Baum "J" an den größeren, der zu H gehört.



Graph

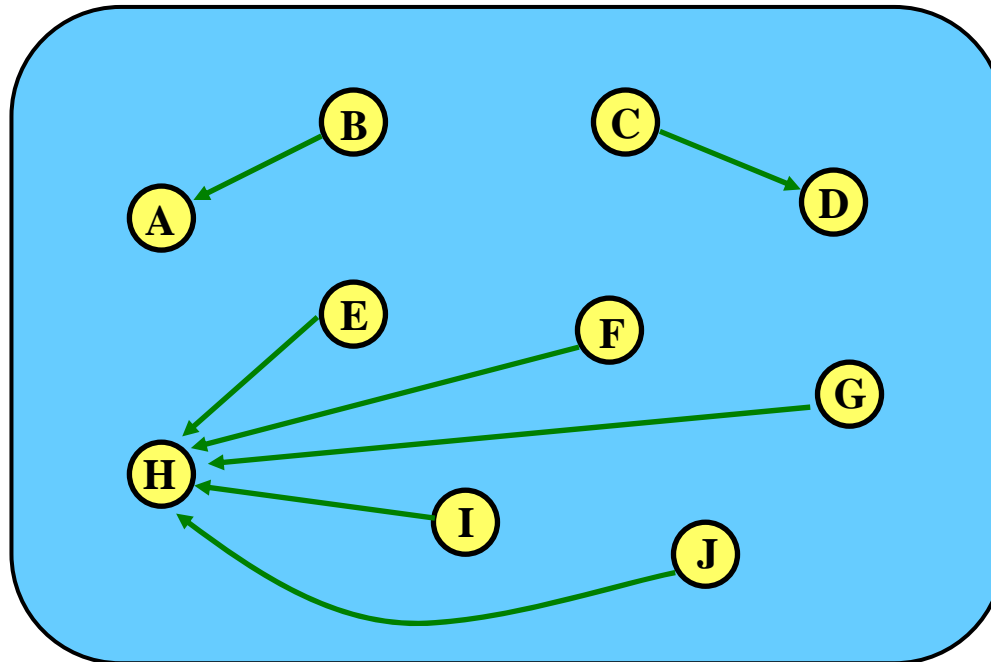


Sortierte
Kantenliste

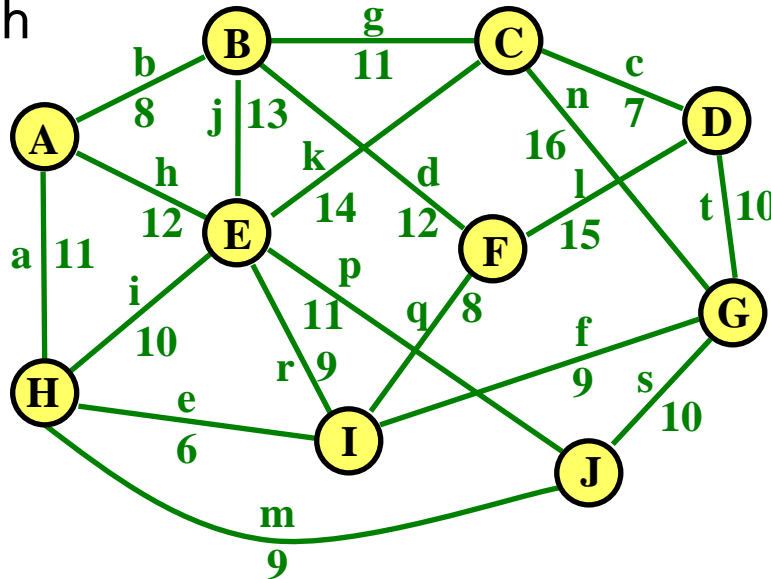
(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



Betrachte nun
die siebente
Kante **(r,9)**.
Hänge den
kleineren Baum
"E" an den
größeren, der
zu I gehört.



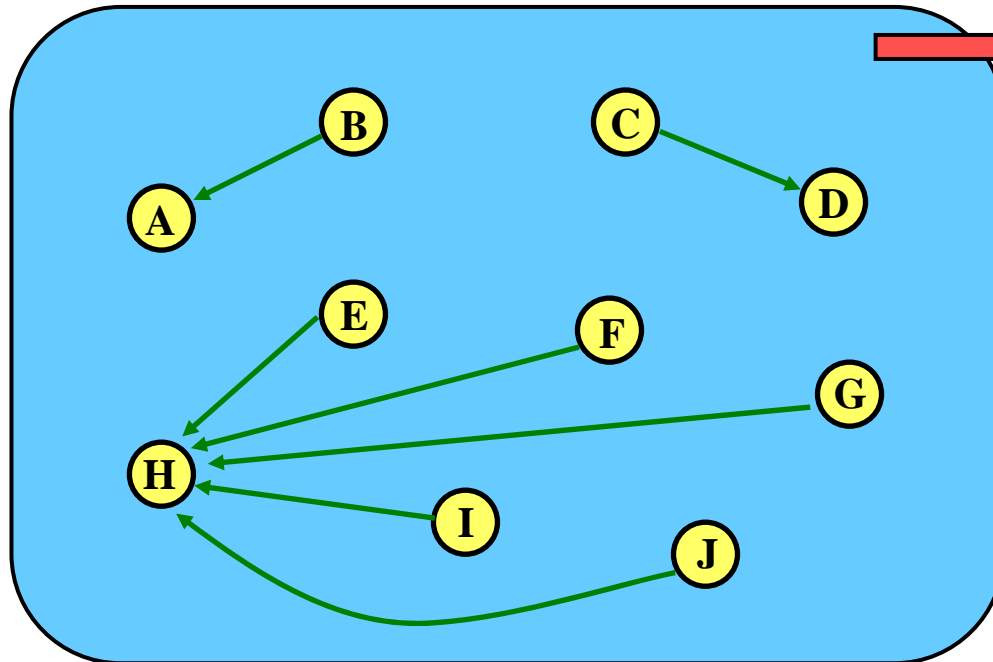
Graph



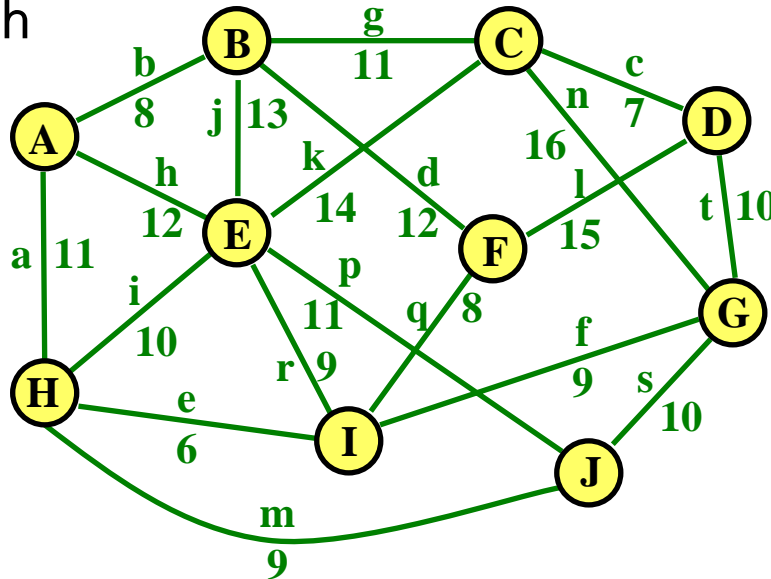
Sortierte
Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun
die achte und
neunte Kante.
Jedes Mal trifft
man auf "H",
d. h., diese
Kanten führen
zu Zyklen.



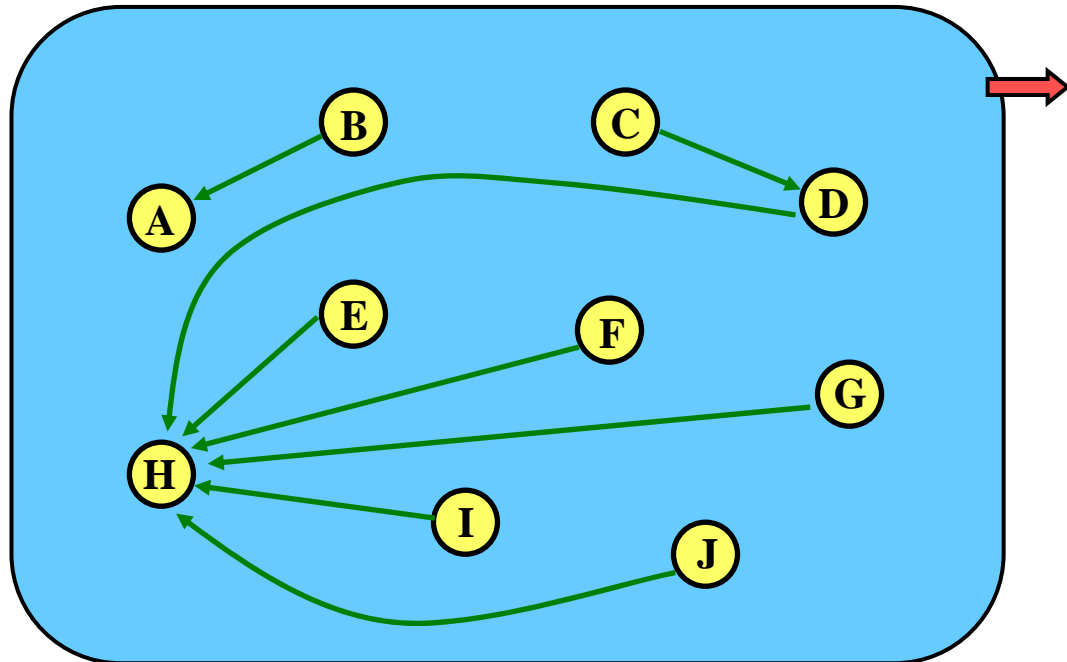
Graph



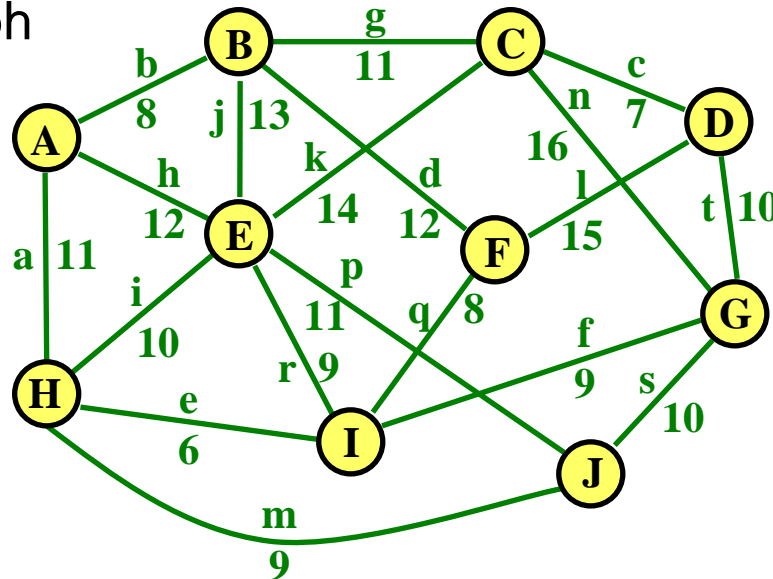
Sortierte
Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11), +
(g,11), +
(p,11), +
(d,12), +
(h,12), +
(j,13), +
(k,14), +
(l,15), +
(n,16), +

Betrachte nun
die zehnte
Kante **(t,10)**.
Nun wird der
Baum "D" an
den größeren
Baum "H"
gehängt.



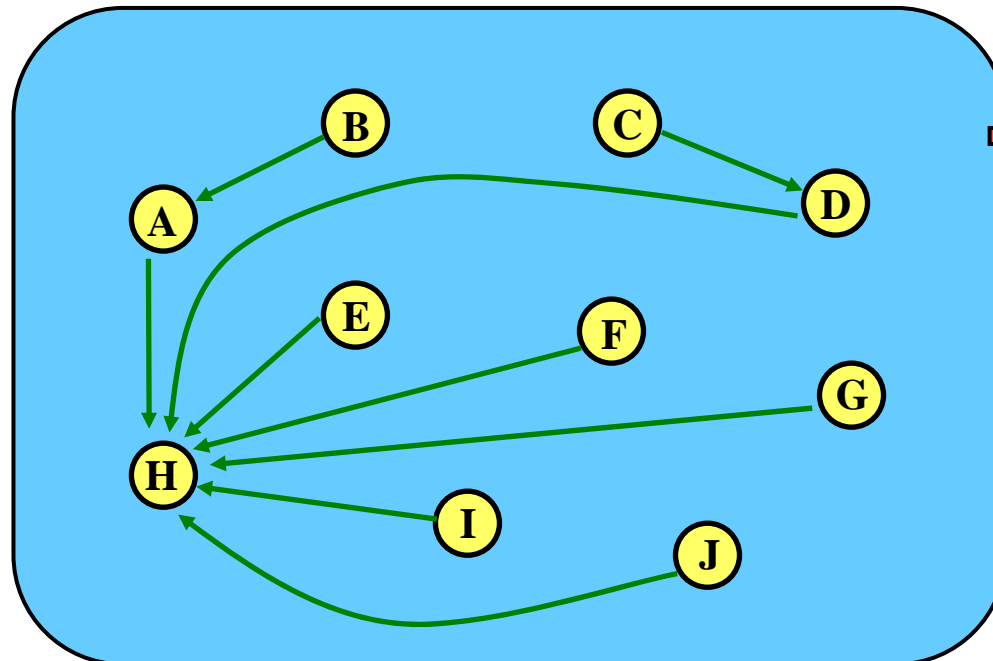
Graph



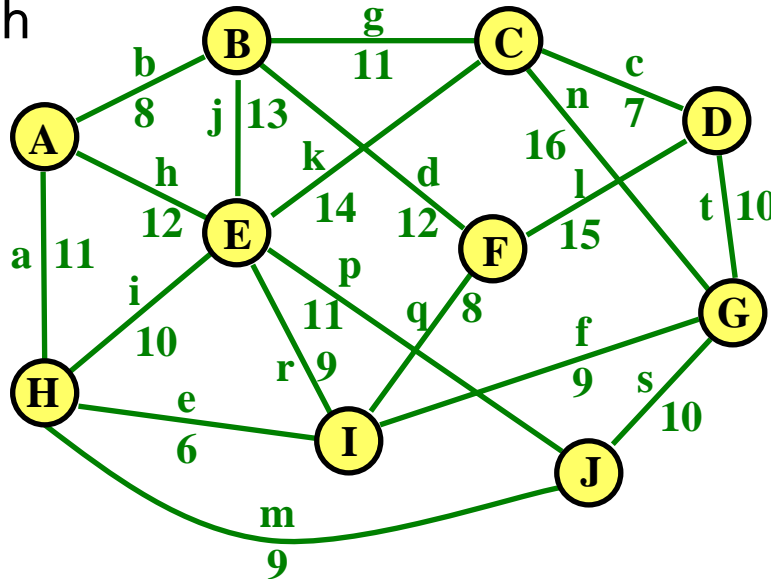
Sortierte
Kantenliste

(e,6), +
 (c,7), +
 (b,8), +
 (q,8), +
 (f,9), +
 (m,9), +
 (r,9), +
 (i,10), -
 (s,10), -
 (t,10), +
 (a,11), +
 (g,11), +
 (p,11), +
 (d,12), +
 (h,12), +
 (j,13), +
 (k,14), +
 (l,15), +
 (n,16), +

Betrachte jetzt
 die elfte Kante
 (a,11). Nun wird
 der Baum "A"
 an den Baum
 "H" gehängt -
 und wir sind
 fertig, da $n-1$
 Kanten gefun-
 den wurden.



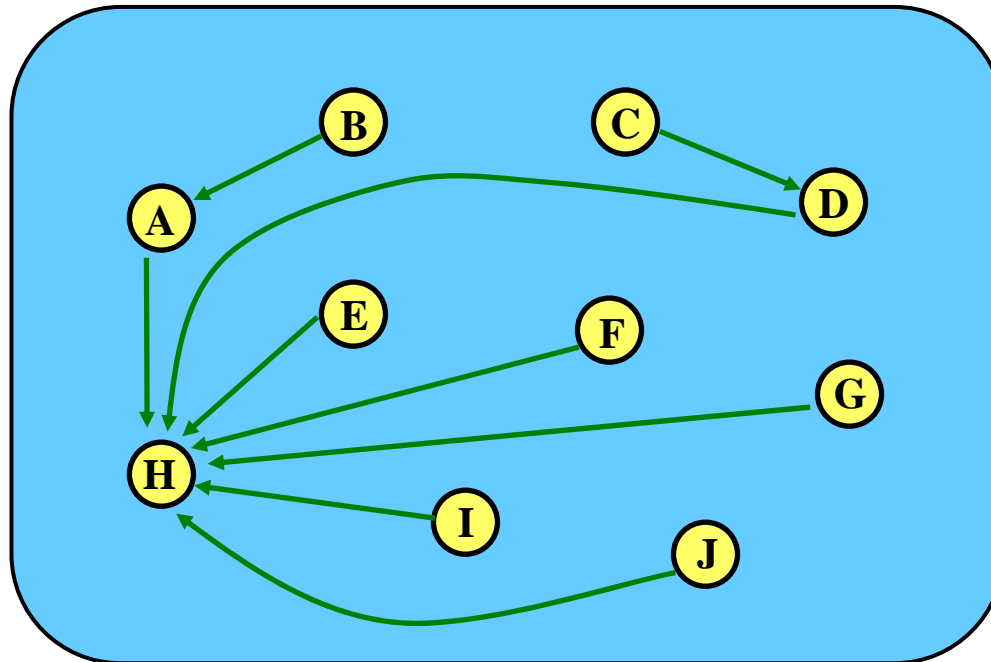
Graph



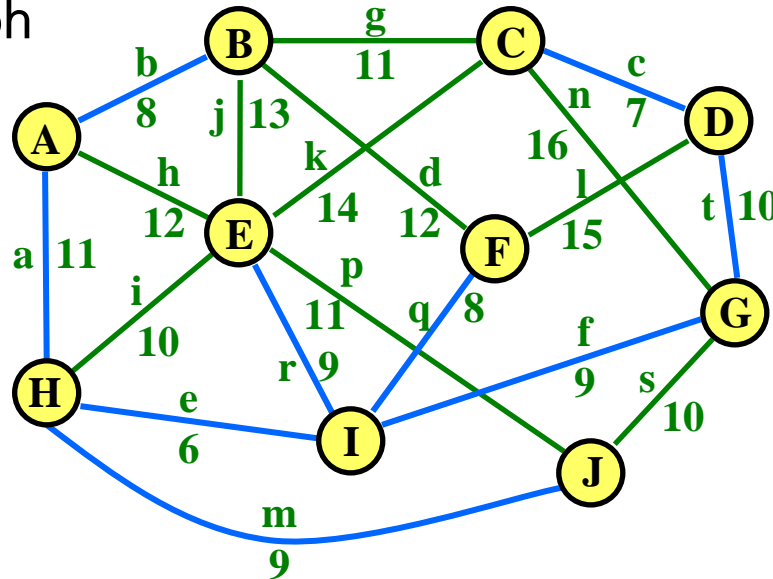
Sortierte Kantenliste

(e,6), +
 (c,7), +
 (b,8), +
 (q,8), +
 (f,9), +
 (m,9), +
 (r,9), +
 (i,10), -
 (s,10), -
 (t,10), +
 (a,11), +
 (g,11), +
 (p,11), +
 (d,12), +
 (h,12), +
 (j,13), +
 (k,14), +
 (l,15), +
 (n,16), +

Ergebnis: Die mit "+" markierten 9 Kanten **e, c, b, q, f, m, r, t** und **a** bilden einen minimalen Spannbaum.



Graph



Sortierte
Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11), +
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Ergebnis: Die mit "+" markierten 9 Kanten **e, c, b, q, f, m, r, t** und **a** bilden einen minimalen Spannbaum.

Zugleich wird bestätigt, dass der Graph zusammenhängend ist.

Ende des Beispiels.

Zur Datenstruktur: Verwandele einen Wald in einem Baum.

Zunächst bildet jeder Knoten des Graphen einen eigenen einelementigen Baum ohne Kanten (= isolierter Knoten). Man nehme die nächste Kante e der sortierten Kantenliste; ihre Endpunkte seien x und y . Von x ausgehend durchläuft man dessen Baum und endet in einem Wurzel-Knoten $Z1$; von y ausgehend landet man in einem Wurzel-Knoten $Z2$. Sind $Z1$ und $Z2$ verschieden, so hängt man den kleineren der beiden Bäume mit den Wurzeln $Z1$ und $Z2$ an den größeren (man verzeigert hier zur Wurzel hin!) und e wird zu den Kanten, die später einen minimalen Spannbaum bilden, hinzugenommen. Anderenfalls sind x und y bereits durch Kanten des Spannbaums verbunden und die aktuelle Kante e wird daher verworfen.

Programmieren Sie dies!

11.4 Maximales Matching

Vorbemerkung: Das Problem, zu speziellen Graphen ein maximales Matching zu finden, ist unter dem Namen Heiratsproblem bekannt. Seine Lösung ist unter den Problemen aufgeführt, die als "Algorithmus der Woche" im Informatikjahr 2006 ins Netz gestellt wurden, siehe:

<http://www.informatikjahr.de/>

Das Heiratsproblem lautet: Gegeben sind zwei disjunkte Mengen D und H ('Damen' und 'Herren') und eine Relation $E \subseteq \{ \{x,y\} \mid x \in D \text{ und } y \in H \}$. Gesucht ist eine möglichst große Teilmenge F von E , in der jedes Element von D und von H höchstens einmal vorkommt. Die Relation E wird als "wechselseitig sympathisch" und die Menge F als Menge von Hochzeiten aufgefasst.

Definition 11.4.1:

$G=(V, E)$ sei ein ungerichteter Graph.

a) Eine Teilmenge der Kanten $F \subseteq E$ heißt Matching, wenn je zwei verschiedene Kanten von F disjunkt sind, d. h., für alle $\{x, y\}, \{x', y'\} \in F$ gilt:

aus $\{x, y\} \neq \{x', y'\}$ folgt $\{x, y\} \cap \{x', y'\} = \emptyset$.

b) Ein Matching $F \subseteq E$ heißt maximal, wenn es kein Matching $F' \subseteq E$ mit $|F| < |F'|$ gibt.

Man könnte meinen, das Problem, ein maximales Matching zu finden, sei einfach: Ausgehend von der leeren Menge nehme man schrittweise immer wieder eine Kante hinzu, die zwei noch nicht ausgewählte Knoten miteinander verbindet. Dies führt in der Regel jedoch nicht zu einem maximalen Matching (selbst ein Beispiel konstruieren!).

Definition 11.4.2: Ein ungerichteter Graph $G=(V, E)$ heißt **bipartit**, wenn seine Knotenmenge V so in zwei disjunkte Teilmengen D und H zerlegt werden kann, dass Kanten nur von einer zur anderen Teilmenge führen, d.h.,
 $V=D \cup H$, $D \cap H = \emptyset$ und $E \subseteq \{ \{x,y\} \mid x \in D \text{ und } y \in H \}$.
Man schreibt dann auch $G=(D \cup H, E)$ oder $G=(D, H; E)$.

Das Problem, ein maximales Matching in einem bipartiten Graphen zu finden, bezeichnet man als *Heiratsproblem*. Es wurde 1935 von dem englischen Mathematiker Philip Hall charakterisiert.

Bemerkung: Diese Definitionen lassen sich leicht auf gerichtete Graphen übertragen, indem die Begriffe für die zugehörigen ungerichteten Versionen (siehe 3.8.5 c) gelten.

11.4.3 Maximales Matching in bipartiten Graphen

siehe zugehöriger "Algorithmus der Woche" unter

www.informatikjahr.de

(im August 2006 als Algorithmus 22 ins Netz gestellt)

Dieser Algorithmus wurde in der Vorlesung präsentiert und erläutert. Zeitkomplexität: $O(n \cdot m)$. Lässt sich um den Faktor \sqrt{n} verbessern.

11.4.4 Maximales Matching in beliebigen Graphen

-- entfällt hier leider --

Abschließende Hinweise zu weiteren Graphalgorithmen

Beispiele solcher Algorithmen:

Zusammenhangskomponenten

Färbbarkeit

größte vollständige Teilgraphen (Cliquesproblem)

minimale Rundreise (TSP = travelling salesman problem)

zweit-, dritt-, ..., k-t kürzeste Wege

Baumisomorphie (gerichtet / ungerichtet)

Graphisomorphie

Teilgraph-Isomorphie

Maximaler Fluss in einem Graphen-Netzwerk

(Stichwort: MaxFlow = MinCut).

Eine Informatik-Anwendung: Schicke maximal viele

Datenpakete durch ein Netzwerk. Wie hoch ist der Fluss?

Kann man eine feste Empfangszeit garantieren? Wie

hängt dies von der Zahl der Empfänger ab?

Einführung in die Informatik II

Universität Stuttgart, Studienjahr 2007

Gliederung der Grundvorlesung

~~8. Suchen~~

~~9. Hashing~~

~~10. Sortieren~~

~~11. Graphalgorithmen~~

12. Speicherverwaltung

12 Verwaltung von Datenstrukturen

12.1 Keller und Halde

12.2 Kellerverwaltung

12.3 Freispeicher- und Haldenverwaltung

12.4 Historische Hinweise

12.1 Keller und Halde

12.1.1 Überblick über die wichtigsten Speicher: Wie in 3.4.3 vorgestellt benötigen Programme mindestens drei Typen von Speichern:

1. Zur Übersetzungszeit bekannter ("statischer") Speicher.

Am Ende der Übersetzung des Programms liegen fest:

1.1: Der Platz, den das übersetzte Programm braucht.

1.2: Der Platz für alle Konstanten und für alle Variablen, die zu einem Datentyp gehören, dessen Speicherplatzbedarf von vornherein feststeht, z.B.: elementarer Datentyp, Aufzählungstyp, Unterbereiche hiervon, records, die nur aus solchen Komponenten bestehen, Felder hierüber von konstanter Länge, access-Datentypen (nur der Zeiger, nicht die hiermit aufgebaute Liste).

2. Zur Laufzeit beim Abarbeiten der Deklarationen weiterhin erforderlicher dynamischer Speicher (Keller-Speicher).

Viele Variablen, die in klammerartig ineinander geschachtelten Blockstrukturen oder in aufgerufenen Unterprogrammen oder anderen Einheiten stehen oder die parametrisiert sind (dynamische Felder, Records mit Diskriminanten), erhalten ihren Speicherplatz erst zur Laufzeit zugewiesen. Der Platz für solche Objekte wird beim Erreichen der zugehörigen Deklarationen hinten an den zum Programm gehörenden (lokalen) Speicher angehängt und er wird am Ende ihrer Lebensdauer wieder frei gegeben. Wegen der Klammerstruktur, in die die Deklarationen eingebunden sind, ist dieser dynamische Speicher ein Kellerspeicher (Pushdown, Keller), siehe 4.1.4.

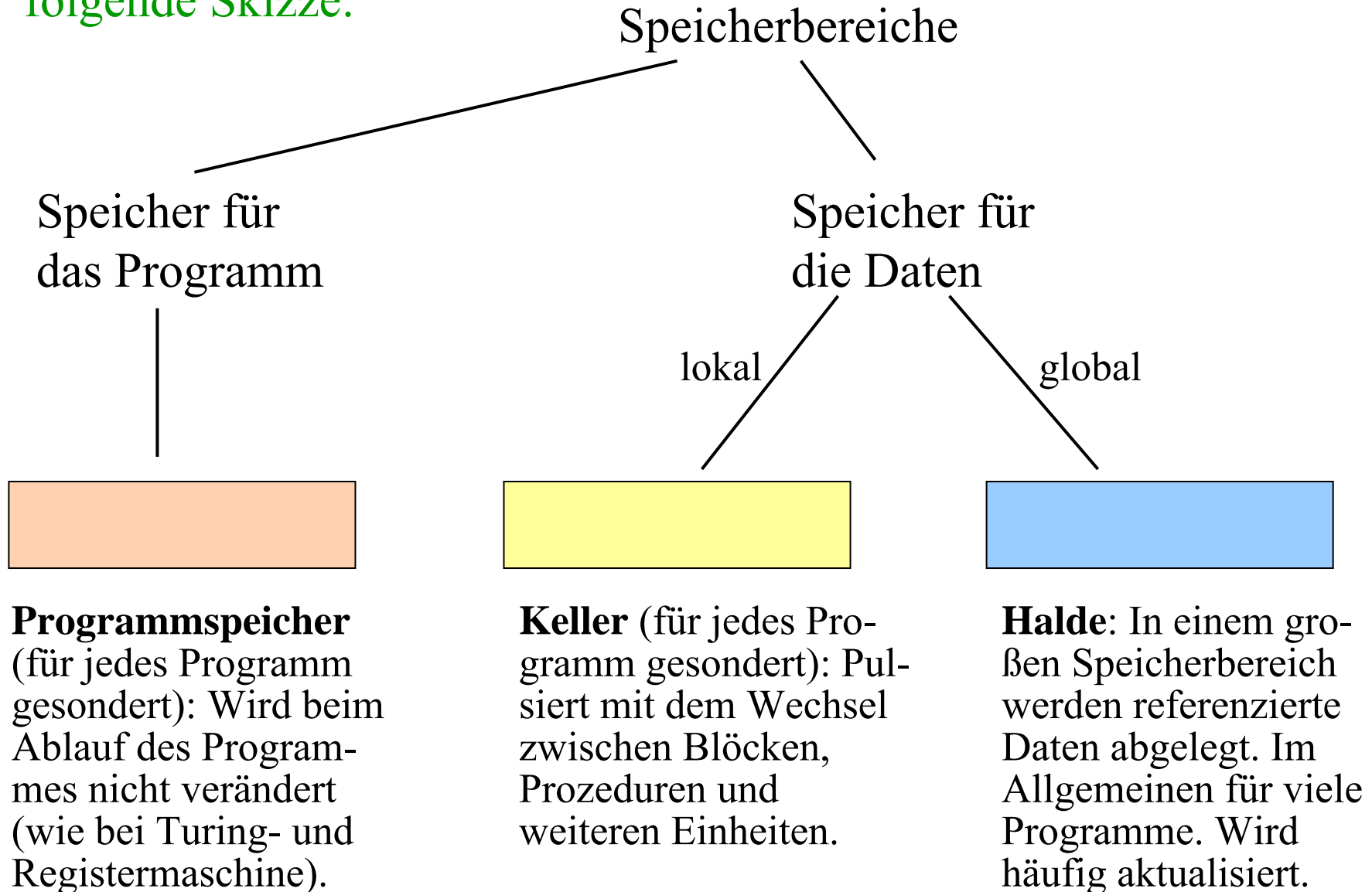
Keller-Speicher in der Praxis:

In der Regel legt man auch die Daten, die zum statischen Speicher zählen, in diesem Kellerspeicher (ganz am Anfang) ab, wodurch man die Deklarationen zur Laufzeit einheitlich abarbeiten kann.

3. Zur Laufzeit durch Anweisungen erzeugte Daten, deren Lebensdauer sich nicht an den Programmeinheiten orientiert.

Die Speicherplätze für solche dynamisch erzeugten Daten werden mittels new angefordert und zugeordnet ("allokiert"). Dieser Speicher pulsiert nicht kellerartig, daher liegen diese Speicherplätze in einem allgemeinen Speicherbereich, der Halde. Die Halde kann von vielen Programmen gleichzeitig genutzt werden. Die Speicherplätze in der Halde werden meist explizit freigegeben, sobald die Daten nicht mehr gebraucht werden, oder sie werden mit einer Speicherbereinigung entfernt, sobald die Halde überzulaufen droht.

So erhält man
folgende Skizze:



12.1.2 Erinnerung an Pointer/Zeiger/Listen:

Listen sind Folgen von Elementen des gleichen Datentyps. Sie werden durch *Zeiger (pointer, access-Datentypen)* realisiert. Die Verkettung kann einfach oder doppelt, die Anordnung sequentiell oder ringförmig sein. Das erste und/oder das letzte Element sind von außen über einen Zeiger erreichbar (auch *Anker* der Liste genannt). Der Zugriff erfolgt *sequenziell*; man durchläuft also die Liste von vorne nach hinten bzw. von hinten nach vorne, um nach einem Element zu suchen oder um ein Element einzufügen. Die mit einer Liste üblicherweise verbundenen Operationen finden sich unter 3.5.

Das Schlüsselwort für Zeiger lautet in Ada access.
Typische Definition einer Liste in Ada:

```
type Element is ....;    -- Definiere den Datentyp der Listenelemente
type List_Element;      -- Vorwärtsverweis
type List_Element_Zeiger is access List_Element;
type List_Element is record
                        Inhalt: Element;
                        Next: List_Element_Zeiger;
end record;
```

Dies ist eine Liste, in der die Elemente "im Original" stehen.
In der Praxis ist dies oft lästig, da ein Element in vielen Listen auftreten kann und dann auch in allen Listen gespeichert und simultan geändert werden muss. Also:

```

type Element is ....;      -- Definiere den Datentyp der Listenelemente
type Element_Zeiger is access Element;
type Zelle;                  -- Vorwärtsverweis
type Zelle_Zeiger is access Zelle;
type Zelle is record

```

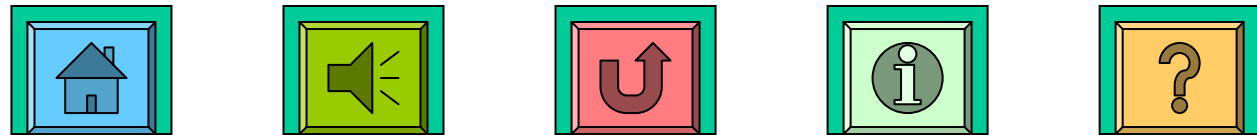
Inhalt: Element_Zeiger;

Next: Zelle_Zeiger;

end record;

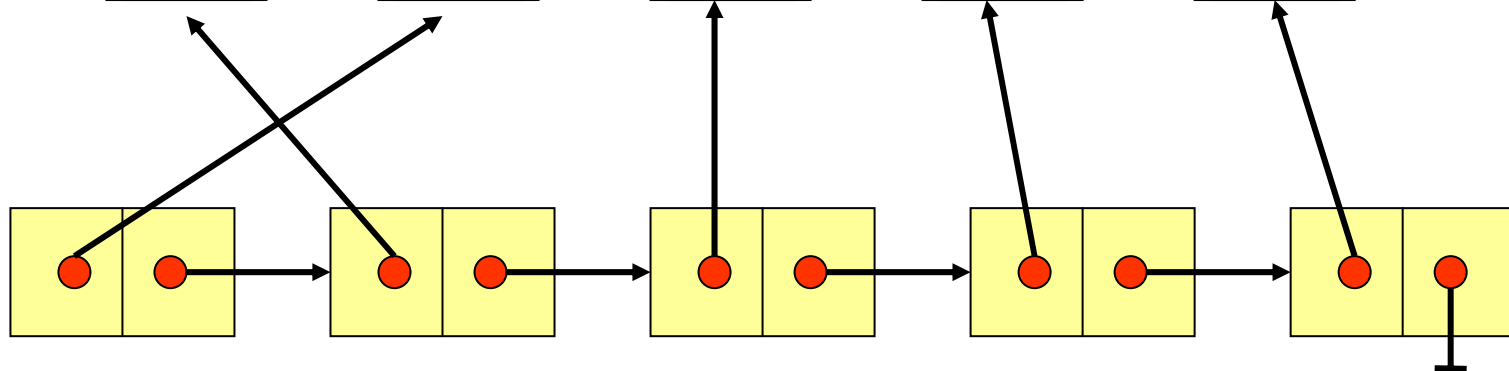
Stellen Sie sich
diese Elemente
bitte riesig vor!

Elemente:

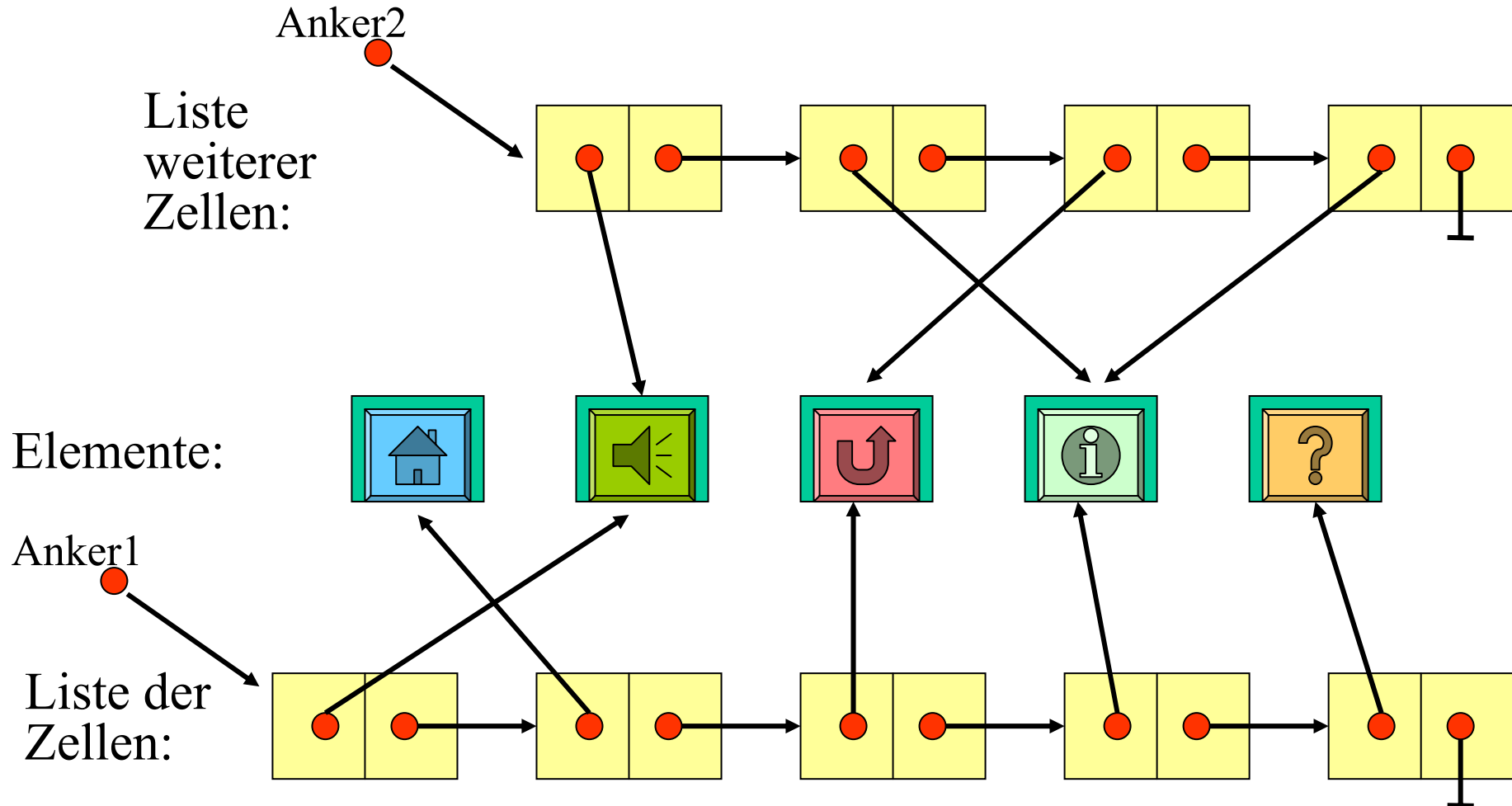


Anker

Liste der
Zellen:



Will man die Elemente in mehreren Listen gleichzeitig verwenden, so kann dies ohne Kopien der Elemente geschehen:



Vorteile dieser Zellen-Darstellung:

- Elemente können in verschiedenen Listen sein.
- Elemente werden in allen Listen gleichzeitig geändert (da nur das Original geändert werden muss).
- Das Einfügen in andere Listen ist einfach.
- Es lassen sich weitere Zugriffsstrukturen leicht aufbauen.

Nachteile dieser Zellen-Darstellung:

- Der Zugriff auf Elemente dauert evtl. etwas länger.
- Man braucht evtl. mehr Speicherplatz (als z.B. mit arrays).
- Es muss die Halde als Speicher verwaltet werden (meist keine direkte Kontrolle über die dortigen Abläufe).

Hinweis: Listen werden in der Halde abgelegt. Nur die Anker stehen im statischen Bereich oder lokalen Keller des Programms, sofern sie deklarierte Variablen sind.

12.2 Kellerverwaltung

Hiermit ist nicht die (recht einfache) Verwaltung eines einzelnen Kellers (3.5.4) gemeint, sondern die Verwaltung vieler Keller in einem beschränkten linearen gemeinsamen Speicherbereich.

In einem Computer werden gleichzeitig viele Programme ("jobs") verwaltet. Jedes besitzt einen lokalen Keller (die dynamischen Daten werden in der Halde abgelegt, siehe 12.3).

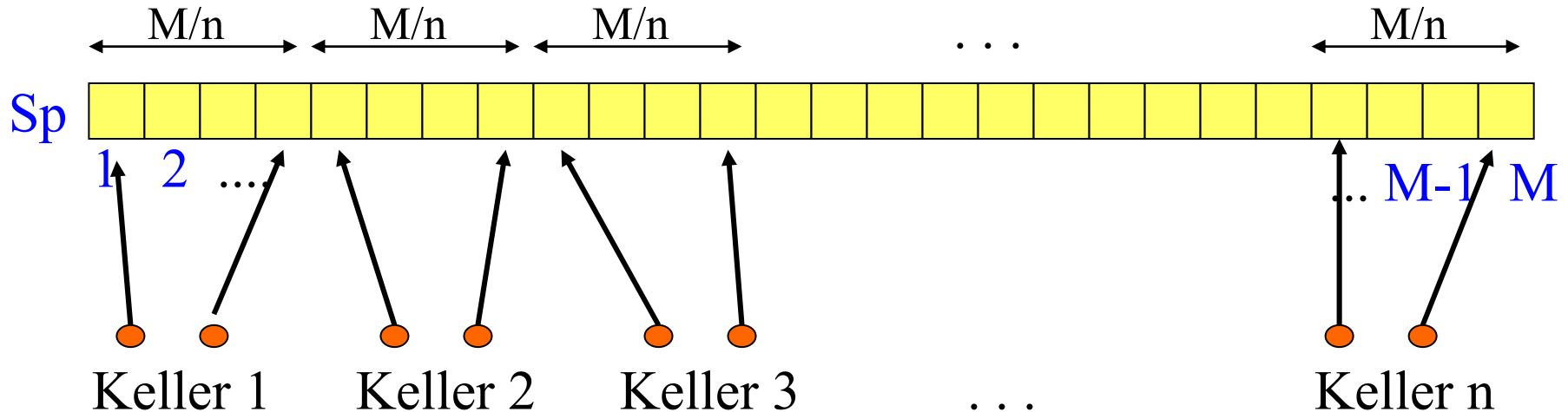
Alle Keller liegen in einem vorgegebenen Speicherbereich $Sp(1..M)$. Manche schwanken stark, andere nur wenig. In der Regel wird der Gesamt-Speicherplatz nicht überschritten, aber wenn man jedem Programm einen festen Bereich zuweisen würde, so wird es oft einen Speicherüberlauf geben. Daher muss man den Gesamt-Speicherplatz geeignet auf die jeweils laufenden Programme verteilen. Allgemeine Formulierung des Problems:

12.2.1: Implementierung von mehreren Kellern

Aufgabe: Wir wollen eine **Multikellerverwaltung** in einem eindimensionalen Feld durchführen, d.h.:

Es sollen n Keller verwaltet werden. Insgesamt steht hierfür ein linearer Speicher $\text{Sp}(1..M)$ zur Verfügung.

Spontane Idee: Jeder Keller erhält gleich viele Speicherplätze, nämlich M/n :



Diese Verweise werden wir durch Indizes realisieren.

Einfachster Fall: **n = 1**. Es liegt ein einzelner Keller vor. Die Ada-Formulierung hierfür ist ein generisches Paket, z.B.:

generic

M: Positive := 5_000_000;

-- Initialisierung willkürlich

type Element is private;

package Keller is

procedure newkeller;

-- Leeren des Kellers

function isempty return Boolean;

-- Ist der Keller leer?

function isfull return Boolean;

-- Ist der Keller voll?

function top return Element;

-- Oberstes Kellerelement

procedure push (x: in Element);

-- Füge Element x oben an

procedure pop;

-- Lösche oberstes Element

function length return Natural;

-- Aktuelle Kellerlänge

unterlauf, ueberlauf: exception;

-- Ausnahmebehandlung

end Keller;

Hieran schließt sich der Modulrumpf an:

```
package body Keller is  
  type Speicher is array (1..M) of Element;  
  Sp: Speicher;  
  index: Integer range 0..M := 0;  
  procedure newkeller is begin index := 0; end;  
  function isfull return Boolean is  
    begin return index >= M; end;  
  procedure push (x: in Element) is  
    begin if isfull then raise ueberlauf;  
      else index := index + 1; Sp(index) := x; end if; end;  
  ...  
  < selbst schreiben: die Prozedur pop, die Funktionen isempty,  
    length, top und die Ausnahmen unterlauf und ueberlauf >  
end Keller;
```

Eine Instanz kann nun lauten:

```
package Ganzzahlkeller is  
    new Keller (M => 800_000, Element => Integer);
```

Nächster Fall: **n = 2**. Wenn man zwei Keller auf einem linearen Speicher Sp der Größe 1 .. M unterbringen möchte, so wird man den ersten Keller von 1 an aufwärts und den zweiten Keller mit M beginnend abwärts implementieren. Auf diese Weise wird der Speicher Sp optimal genutzt.

Aufgabe: Realisieren Sie diesen Fall selbst!

12.2.2 Allgemeiner Fall: $n \geq 3$. Vorhandener Speicher $Sp(1..M)$. Hier gibt es mindestens zwei Varianten:

- *Variante 1:* Jeder Keller hat seine eigene maximale Größe, die in `Max: array (1..n) of Natural` abgelegt ist (einfachster Fall: $Max(i) = \lfloor M/n \rfloor$ für alle i) und für die gilt

$$\sum_{i=1}^n Max(i) \leq M.$$

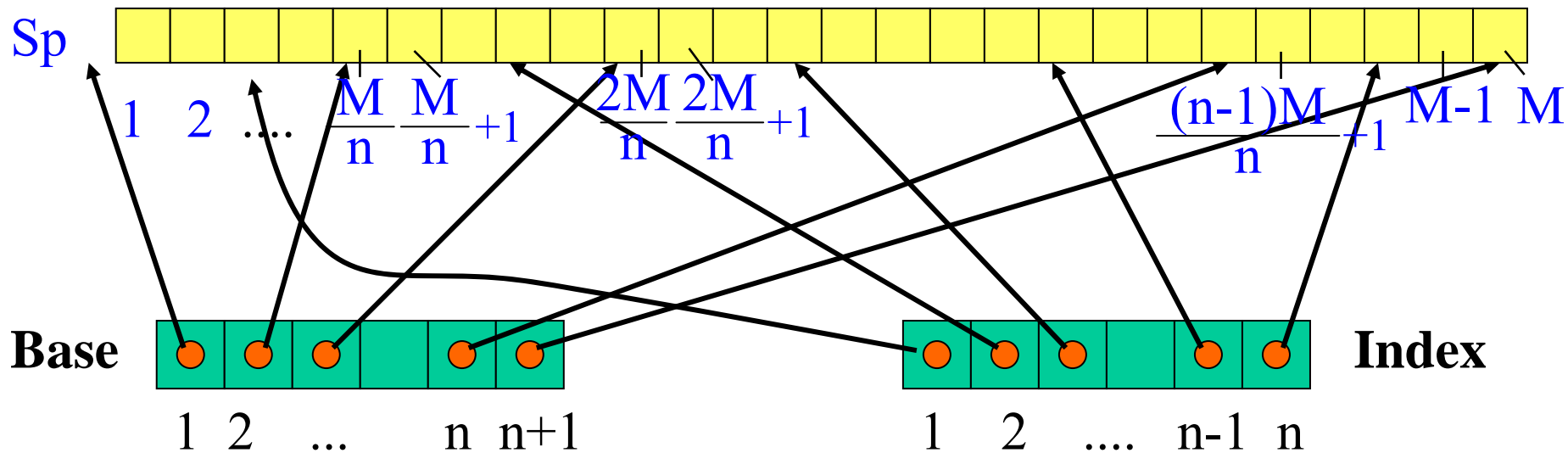
Dieser Fall wird wie "n=1" behandelt, indem überall die Nummer des Kellers hinzugefügt wird und jeder Keller unabhängig von den anderen ist. Es gibt dann zwei Felder für die Adresse vor dem Beginn des i-ten Kellers ("Base") und für seine aktuelle oberste Position ("Index") mit $0 \leq Index(i) - Base(i) \leq Max(i)$ für $i = 1, 2, \dots, n$.

Wir formulieren dies nun genau aus.

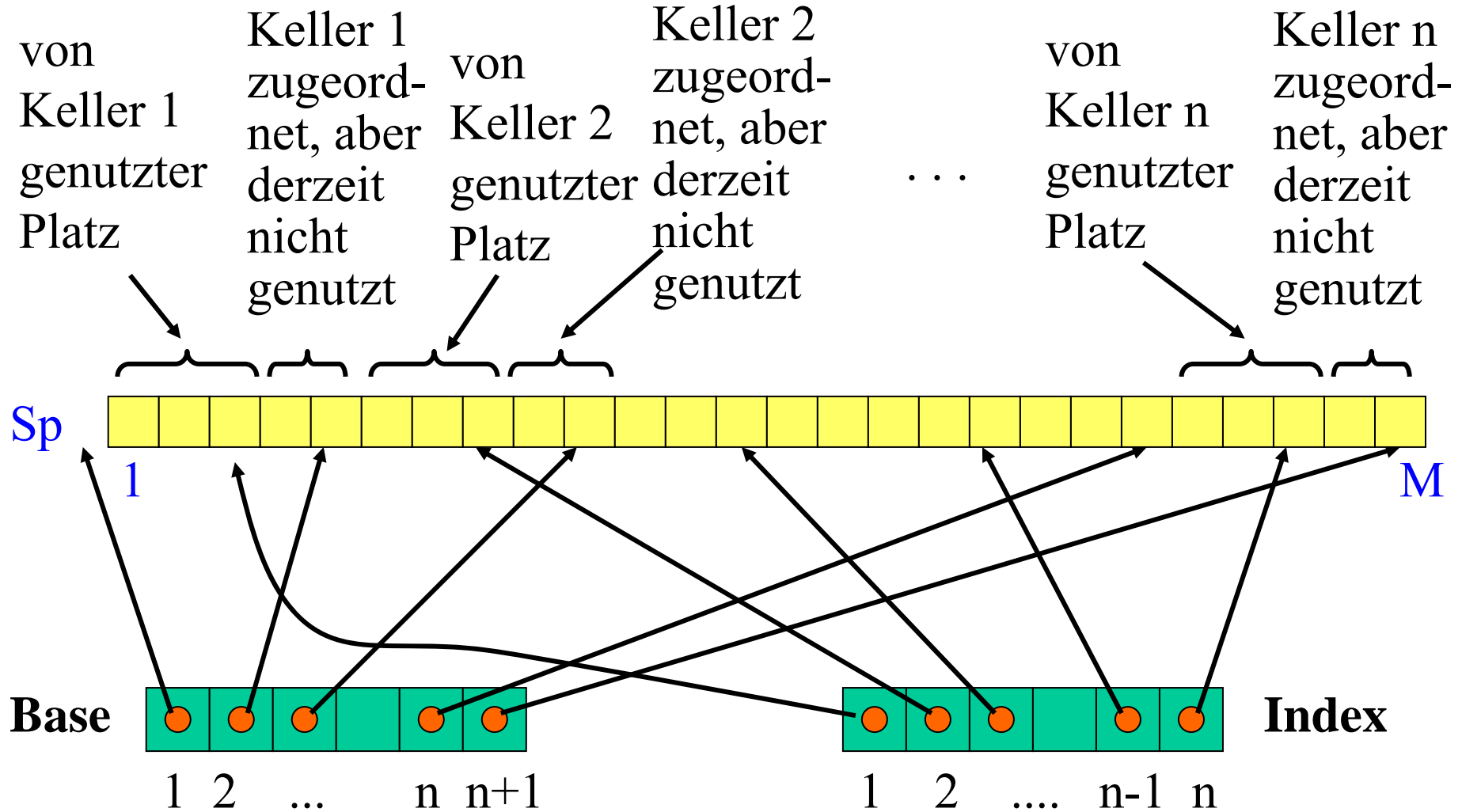
Implementierung: Jeder Keller erhält den gleichen Platz der Größe M/n . Wir verwenden hierfür zwei Zeiger bzw. Indizes:

Base(i) zeigt auf den Speicherplatz, der unmittelbar vor dem Bereich für den i-ten Keller liegt;

Index(i) zeigt auf den Speicherplatz, auf dem sich das oberste Element des i-ten Kellers befindet.



Nochmals zur Illustration: Aufteilung des Speichers Sp



12.2.3 Ada Deklarationen hierzu: (MKV = Multikellerverwaltung)

generic

M: Positive := 5_000_000;	-- willkürlicher Default-Wert
n: Positive;	-- Anzahl der Keller, $n \geq 2$
<u>type</u> Element <u>is private</u> ;	-- Datentyp der Kellerelemente

package MKV1 is

<u>type</u> ZK <u>is</u> Positive <u>range</u> (1..n+1);	-- für Zugriff auf Keller
<u>procedure</u> newkeller (i: <u>in</u> ZK);	-- Leeren des Kellers
<u>function</u> isempty (i: ZK) <u>return</u> Boolean;	-- Ist der Keller leer?
<u>function</u> isfull (i: ZK) <u>return</u> Boolean;	-- Ist der Keller voll?
<u>function</u> top (i: ZK) <u>return</u> Element;	-- Oberstes Kellerelement
<u>procedure</u> push (i: <u>in</u> ZK; x: <u>in</u> Element);	-- Füge x oben an
<u>procedure</u> pop (i: <u>in</u> ZK);	-- Lösche oberstes Element
<u>function</u> length (i: <u>in</u> ZK) <u>return</u> Natural;	-- Aktuelle Kellerlänge
unterlauf (i: ZK), ueberlauf (i: ZK): <u>exception</u> ;	

end MKV1;

Pakettrumpf hierzu: (MKV = Multikellerverwaltung)

```
package body MKV1 is  
type Adressen is Natural range 0..M; -- Speicher“adressen“  
Sp: array (Adressen) of Element;      -- Speicher  
Base: array (ZK) of Adressen;          -- Beginn jedes Kellers  
Index: array (ZK) of Adressen;         -- Aktueller Stand jedes Kellers  
procedure newkeller (i: in ZK) is  
    begin Index(i) := Base(i); end newkeller;  
function isempty (i: ZK) return Boolean is  
    begin return Base(i) = Index(i); end isempty;  
function isfull (i: ZK) return Boolean is  
    begin return Base(i+1) >= Index(i); end isfull;  
function top (i: ZK) return Element is  
    begin return Sp(Index(i)); end top;
```


Pakettrumpf MKV1 (Fortsetzung)

```
procedure push (i: in ZK; x: in Element) is  
  begin  if isfull(i) then raise ueberlauf (i);  
        else Index(i) := Index(i) + 1;  
          Sp(Index(i)) := x; end if;  
  end push;  
procedure pop (i: in ZK) is  
  begin  if isempty(i) then raise unterlauf(i);  
        else Index(i) := Index(i) - 1; end if; end pop;  
function length (i: ZK) return Natural is  
  begin return Index(i) - Base(i); end length;  
exception  when ... => .....  
end MKV1;
```

Eine konkrete Instanz für zehn Keller könnte dann sein (hier wird der voreingestellte Wert von M genommen):

```
package Zahlenkeller is new MKV1(n => 10; Element => Integer);  
use Zahlenkeller; ...  
for i in 1..n loop Base(i) := (i-1)*(M/n); Index(i):=Base(i); end loop;  
Base(n+1) := M; ...
```

Nachteilig ist, dass die Multikellerverwaltung zusammenbricht, falls irgendein Keller überläuft. In der Regel stehen ja noch weitere Speicherplätze in Sp zur Verfügung.

Es gibt diverse nahe liegende Veränderungen. Diese ersetzen alle „raise ueberlauf(i)“ durch den Prozeduraufruf „umordnen(i)“, um weiteren Speicherplatz bereitzustellen (siehe unten in Variante 2):

```
procedure umordnen (i: in ZK); ...
```

12.2.4 Variante 2: Die Größe jedes einzelnen Kellers ist nicht vorab beschränkt und alle Keller zusammen sollen den Speicherplatz der Größe M möglichst gut nutzen. Hierfür muss es wiederum zwei Felder Base, Index: array (1..n) of Natural geben, für die zu jedem Zeitpunkt gilt

$$\sum_{i=1}^n \text{Index}(i) - \text{Base}(i) \leq M.$$

Wenn einer der Keller überläuft (d.h. push-Operation bei $\text{Index}(i) = \text{Base}(i+1)$) und wenn zugleich andere Keller den ihnen zugewiesenen Bereich noch nicht voll ausnutzen, *so muss der Speicherplatz neu auf die Keller verteilt werden.* Hier sind mehrere Untervarianten möglich.

Möglichkeit 1:

Schaue nach, ob der rechte oder linke Nachbar des Kellers i noch genügend freien Platz hat und tritt dann die Hälfte dieser Plätze an den Keller i ab.

Möglichkeit 2:

Suche einen Keller j mit maximal viel freiem Platz, das heißt, $\text{Index}(j) - \text{Base}(j)$ ist maximal, und tritt die Hälfte dieser Plätze an den Keller i ab. Konkret muss dann der Speicherbereich zwischen den Kellern i und j um q Speicherplätze verschoben werden, wobei q die Hälfte der freien Plätze von Keller j ist.

Möglichkeit 3:

Berechne den Speicherplatz, den jeder Keller bekommen soll, neu, indem jedem Keller eine Mindestzahl an Plätzen und weitere Plätze **entsprechend seines bisherigen Wachstums** zugewiesen werden, und ordne den Speicher dann komplett um.

Möglichkeit 1: (nur die benachbarten Keller betrachten)

```
procedure umordnen (i: in ZK) is           -- n muss mindestens 3 sein
    k: ZK; q: Adressen;
begin k := i;  -- Teste auch, ob beim Keller k mindestens 2 Plätze frei sind
    if (i=1) and (Index(2) + 1 < Base(3)) then k := 2;
    elsif (i=n) and (Index(n-1) + 1 < Base(n)) then k := n-1;
    elsif Base(i) - Index(i-1) + 1 < Base(i+2) - Index(i+1)
        then k := i+1;
        elsif Index(i-1) + 1 < Base(i) then k := i-1; end if;
        -- Keller k dient nun als Platz-Lieferant
    if k=i then raise ueberlauf;
    elsif k<i then
        q := (Base(k+1) - Index(k))/2;      -- Hälfte des freien Platzes
        Base(i) := Base(i) - q; Index(i) := Index(i) - q;
        for j in Base(i)..Index(i) loop Sp(j) := Sp(j+q); end loop;
    else <das Gleiche, aber nach oben verschieben; selbst einfügen> end if;
end umordnen;
```

Möglichkeit 2: (Keller, der maximal viel Platz abgeben kann, suchen)

procedure umordnen (i: in ZK) is

 k: ZK; q: Adressen;

begin k := 1; -- Suche Keller k mit maximal freiem Platz

for j in 2..n loop

if (Base(j+1) - Index(j)) > (Base(k+1) - Index(k))

then k := j; end if;

end loop;

 q := (Base(k+1) - Index(k))/2; -- Hälfte des freien Platzes

if q <= 0 then raise ueberlauf;

elsif k < i then

for j in k+1..i loop

 Base(j) := Base(j) - q; Index(j) := Index(j) - q; end loop;

for j in Base(k+1)..Index(i) loop Sp(j) := Sp(j+q); end loop;

else < das Gleiche, nur nach oben verschieben; selbst einfügen > end if;

end umordnen;

Nachteil der Möglichkeit 1:

Ein Abbruch kann geschehen, obwohl noch irgendwelche anderen Keller ihren Platz kaum benötigen. Denn man prüft ja nur die benachbarten Keller ab. Auch kann "umordnen" relativ rasch wieder aufgerufen werden.

Nachteil von Möglichkeit 2:

Eventuell wird die Prozedur "umordnen" nach q Schritten erneut aufgerufen, vor allem, wenn ein Keller schnell wächst. Komplettes Neuverteilen der Speicherbereiche vermeidet dies.

Vorteil:

Die Prozedur "umordnen" wird sehr schnell abgearbeitet.

12.2.5 Möglichkeit 3: (Garwick-Algorithmus)

- Berechne den insgesamt freien Platz aller Keller ("sum").
- Berechne den gesamten Zuwachs seit dem letzten Umordnen.
- Verteile 10% des freien Platzes gleichmäßig an alle Keller.
- Verteile 90% des freien Platzes proportional zum Zuwachs.

Um den Zuwachs zu berechnen, muss man sich in einem array **AltIndex** merken, welches die Indexpositionen *unmittelbar nach* dem letzten Umordnen waren. Um die Umordnung durchzuführen, muss man die neuen Basispositionen in einem array **NewBase** notieren. Der Zuwachs ergibt sich dann aus der Summe der Werte $(\text{Index}(j) - \text{AltIndex}(j))$, wobei man nur die positiven Werte addieren darf. $\text{NewBase}(j)$ ergibt sich aus den Newbase-Werten der darunter liegenden Keller erhöht um den festen Anteil u , der jedem Keller zusteht, und dem Zuwachs-Anteil.

Die Formeln wollen wir zunächst genau angeben.

Zu berechnende Größen (u und w gerundet, weil ganzzahlig):

$\text{sum} :=$ gesamter freier Speicherplatz aller n Keller

$\text{zuwachs} :=$

Summiere die nichtnegativen Werte von $\text{Index}(j) - \text{Altindex}(j)$

Hier werden nur die Keller berücksichtigt, die seit dem letzten Umordnen gewachsen sind.

$u := \lfloor (\text{sum}/10) / n \rfloor$

u ist 10% der Zahl freier Speicherplätze anteilig für jeden Keller; mindestens u Speicherplätze werden jedem Keller als freie Plätze zugewiesen.

$u*n$ sind die freien Speicherplätze, die vorab an alle Keller verteilt werden.

$\text{sum} - u*n$ ist der restliche Speicherplatz, der nach Zuwachs zuzuordnen ist.

$v := (\text{sum} - u*n) / \text{zuwachs}$, $w := \lfloor v * \delta \rfloor$

Wenn ein Keller um δ Plätze gewachsen ist, so erhält er diese w zusätzlichen Speicherplätze, aber nur im Falle $\delta > 0$. (v ist der Faktor je Zuwachseinheit.)

Damit sind die Formeln in folgender Prozedur "umordnen" erklärt (diese Prozedur braucht keine Parameter mehr):

Garwick-Algorithmus: Füge zum "package body MKV1" hinzu:
NewBase: array (ZK) of Adressen; -- Neuer Beginn der Kellers

AltIndex: array (ZK) of Adressen; -- Alter Stand jedes Kellers

procedure umordnen is -- bei Möglichkeit 3 brauchen wir keinen Parameter i

 k: ZK; sum, Zuwachs, u, h: Integer; v: Float; -- n ist global

 Delta: array (ZK) of Adressen; -- für den Zuwachs jedes Kellers

begin sum := 0; -- Addiere freien Platz in "sum" auf

for j in 1..n loop sum:=sum + Base(j+1) - Index(j); end loop;

if sum <= n then raise ueberlauf;

 -- Nicht genug Platz frei; bei sum > n vermeidet man eine unendliche

 -- Schleife durch ständig erneutes Aufrufen des Garwickalgorithmus

else Zuwachs := 0; -- ermittle Zuwächse seit letztem "umordnen"

for j in ZK loop h := Index(j) - AltIndex(j);

if h > 0 then Delta(j) := h; Zuwachs := Zuwachs + h;

else Delta(j) := 0; end if;

end loop;

if zuwachs >= 1 then

u := INTEGER(0.1*FLOAT(sum)/FLOAT(n));

-- 10% Anteil, der jedem Keller zusteht)

v := (FLOAT(sum) - FLOAT(u*n))/FLOAT(zuwachs);

else u := INTEGER(FLOAT(sum)/FLOAT(n)); v:=0; end if;

-- Dieser else-Fall darf eigentlich nicht eintreten, da ja ein Keller

-- überläuft; man könnte also auch eine exception erwecken.

NewBase(1) := 0; NewBase(n+1) := M;

for j in 2..n loop

NewBase(j) := NewBase(j-1) + Index(j-1) - Base(j-1)

+ u + INTEGER(Delta(j-1)*v - 0.5); end loop;

speicherumordnen;

for j in ZK loop AltIndex(j) := Index(j); end loop;

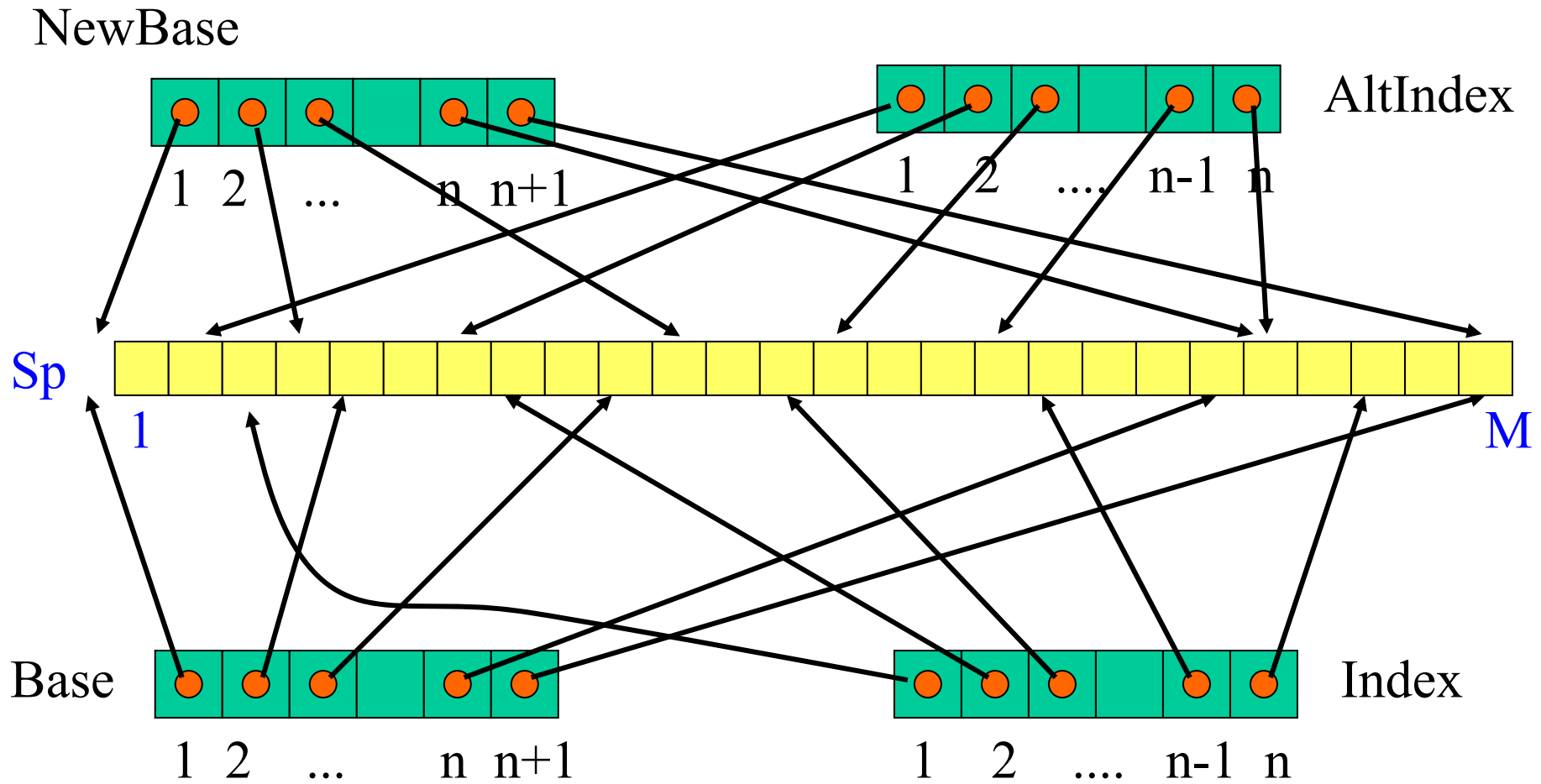
end if;

end umordnen;

-- Hier kann am Ende wegen Rundungsfehlern freier Speicherplatz übrig

-- bleiben (INTEGER(...-0.5)). Wie viel? Wie kann man dies berücksichtigen?

Garwick-Algorithmus: Verwaltung des Speichers Sp



Unterprozedur zu "umordnen":

procedure speicherumordnen is

 m, j, k: ZK;

begin j := 2;

while (j <= n) loop

 k := j;

if NewBase(k) < Base(k) then verschieben(k);

else while NewBase(k+1) > Base(k+1) loop

 k := k + 1; end loop;

 -- Diese Schleife endet spätestens für k = n

for m in reverse j..k loop verschieben(m); end loop;

end if;

 j := k + 1;

end loop;

end speicherumordnen;

Unterprozedur zu "speicherumordnen":

procedure verschieben (i: in ZK) is

d: Integer;

begin d := NewBase(i) - Base(i);

-- d gibt an, um wie viele Stellen Keller i verschoben werden muss

if (d \neq 0) then

if d > 0 then

for a in reverse Base(i)+1 .. Index(i) loop

Sp(a+d) := Sp(a); end loop;

else

for a in Base(i)+1 .. Index(i) loop

Sp(a+d) := Sp(a); end loop;

-- beachte hier d < 0

end if;

Index(i) := Index(i) + d;

Base(i) := NewBase(i);

end if;

end verschieben;

Hinweis: Wo kommen die Konstanten 10 und 90 im Garwickalgorithmus her? Warum nicht 30 und 70?

Dies sind Erfahrungswerte: Man hat die obige Umordnung der Kellerspeicherbereiche für eine feste Menge von Programmen immer wieder für verschiedene Prozentzahlen durchgeführt und hierbei festgestellt, dass der Wert 10% für die Festzuweisung (und somit 90% für den Zuwachs) den besten Durchsatz, d.h., im Mittel die geringste Zahl an Umordnungen ergab.

12.2.6 Verwaltung durch ein Betriebssystem

- Der gesamte Kellerspeicher wird vom Betriebssystem verwaltet.
- Jedes Programm, das neuen Speicherplatz benötigt oder alten zurückgibt oder auf seine Daten zu greifen möchte, schickt eine Anfrage an das Betriebssystem mit "Nummer des Programms i " und "Relative Speicherplatzadresse a ".
- Die zugehörige absolute Speicherplatzadresse im Rechner ist dann $\text{Base}(i) + a$. Ist dieser Wert kleiner oder gleich $\text{Base}(i+1)$, stellt das Betriebssystem den Wert zur Verfügung bzw. speichert ihn ab bzw. legt ihn neu an oder gibt ihn frei; anderenfalls wird die Prozedur umordnen durchgeführt und erst danach entweder die Anfrage bedient oder, falls neuer Speicherplatz nicht mehr zugewiesen werden kann, die weitere Bearbeitung des Programms i unterbrochen und der Konfliktfall irgendwie gelöst (notfalls mit dem Abbruch mindestens eines Programms).

12.3 Freispeicher- und Haldenverwaltung

Daten oder Objekte lassen sich durch Zeiger miteinander verflechten. Früher sprach man dann von "Geflechten", die im Speicher aufzubauen sind. Es handelt sich hierbei um *Graphen* (vgl. 3.8).

Um solche Geflechte oder Vernetzungen aufzubauen, muss in jedem Datenobjekt mindestens ein Zeiger existieren.

Existiert genau ein Zeiger, so kann man nur Listen aufbauen. Ab zwei Zeigern lassen sich stark vernetzte Strukturen realisieren. Beispiel: Binäre Bäume. Siehe hierzu Abschnitte 3.7 und 8.2.

Erläuternder Hinweis:

In der Implementierung werden "Zeiger" stets durch die "Adresse eines Speicherplatzes", ab der die referenzierte Struktur beginnt, dargestellt.

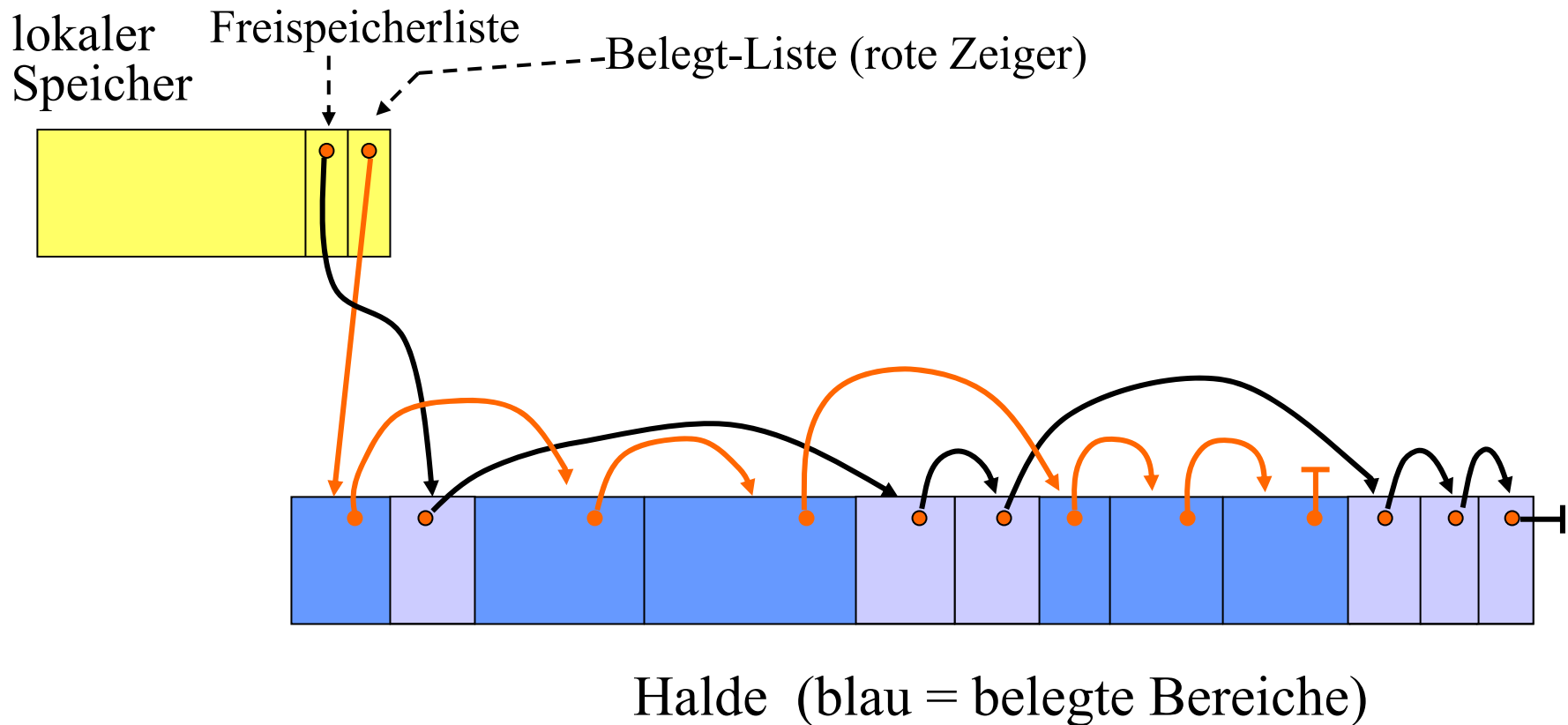
Grund: Man beachte, dass heutige Rechner in der Regel einen ein-dimensionalen Speicher besitzen, auf dessen Speicherplätze über einen Index von 0 bis $2^s - 1$ (für eine natürliche Zahl s) zugegriffen wird. Einen Zeiger implementiert man daher als die Adresse derjenigen Speicherzelle, ab der das Objekt, auf das verwiesen wird, steht.

12.3.1 Halde (*engl.: Heap*): Dies ist ein Speicherbereich, dessen Größe hinreichend groß ist, um die mittels new erzeugten Datenobjekte (Zugriff über Zeiger!) abzulegen. Wenn diese Datenobjekte im Laufe der Rechnungen nicht mehr gebraucht werden, sollten sie explizit wieder frei gegeben werden (in Ada mit Hilfe des pragmas "Controlled" und der Prozedur FREE). Die Verwaltung erfolgt oft über eine Freispeicherliste.

Werden die nicht mehr benötigten Speicherplätze nicht wieder frei gegeben, so liegen nach einiger Zeit in der Halde viele unnötige Datenobjekte herum (= Daten, auf die nicht mehr von irgendeinem Programm über seinen lokalen Speicher zugegriffen werden kann). So kann die Halde rasch voll werden. Um dann noch weiterarbeiten zu können, müssen die nicht mehr benötigten Datenobjekte erkannt, ihre Speicherplätze frei gegeben und die Halde in geeigneter Weise umorganisiert werden (Speicherbereinigung).

12.3.2 Freispeicherliste

Um die dynamischen Daten der Halde zu verwalten, tragen wir die freien Speicherplätze in eine "Freispeicherliste" ein, aber nicht jede Speicherzelle einzeln, sondern immer ganze "Datenblöcke".



Benutzen mehrere Programme die Halde, so werden die "Freispeicherliste" und eventuell auch eine "Belegt-Liste" vom Betriebssystem verwaltet. Folgende Aufgaben sind unter anderen Fragestellungen zu lösen:

1. Ein Programm fordert einen Speicherplatzbereich der Größe "G" an. Weise dem Programm einen geeigneten Bereich in der Halde zu und modifiziere die Freispeicherliste.
2. Ein Programm gibt einen Speicherplatzbereich wieder frei. Füge diesen Bereich "geschickt" in die Freispeicherliste ein.
3. Verschmelze aneinander grenzende freie Datenblöcke der Halde zu größeren Einheiten.

4. Falls keine Zuweisung erfolgen kann, ordne die Halde so um, dass alle freien Bereiche nebeneinander liegen (das ist nicht trivial). Füge hierbei alle Datenblöcke, die nicht mehr benutzt werden, in die Freispeicherliste ein.
5. Falls auch dies nicht erfolgreich ist, führe einen Austausch der Speicherinhalte mit dem Hintergrundspeicher durch (Stichwort: Seitenaustauschstrategien, Paging; siehe Vorlesungen über Betriebssysteme).

Um diese Aufgaben durchzuführen, muss die Freispeicherliste oft durchlaufen werden, wobei wir die Datenblöcke, die zur Freispeicherliste gehören, markieren, um sie später "erkennen" zu können. Ein Datenblock muss also neben dem Inhalt, den das jeweilige Programm hineinschreibt, mindestens seine Größe, ein Markierungsfeld und den Verweis auf den nächsten freien Datenblock enthalten.

Wir legen daher folgenden Datentyp "**DBlock**" für Datenblöcke fest. Es sei eine natürliche Zahl "maxgröße" (= die maximale Größe an Speicherplätzen je Datenblock) vorgegeben (vgl. auch 1.12.3 "Diskriminanten für Größenangaben"):

```
type DBlock;  
type DBlockzeiger is access DBlock;  
subtype AnzahlZellen is Positive range 1..maxgröße;  
type DBlock (größe: AnzahlZellen := maxgröße) is record  
    inhalt: array (1..größe) of Speicherzelle;  
    -- Komponenten, die genau "größe" viele Speicherzellen belegen  
    mark: Boolean;  
    next: DBlockzeiger;  
end record;
```

Jeder DBlock belegt also $\text{größe} + x$ viele Speicherzellen in der Halde, wobei x die Zahl der Speicherzellen für "größe", "mark" und "next" bezeichnet.

Skizze:

Verankerung

Datenblöcke in der Halde

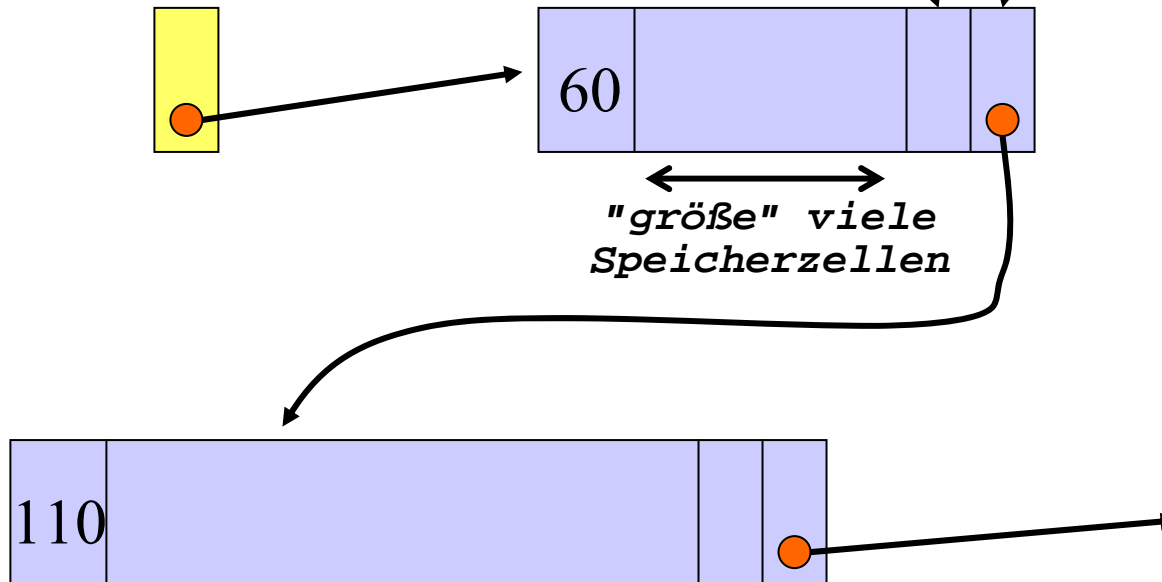
Freispeicherliste

größe

mark

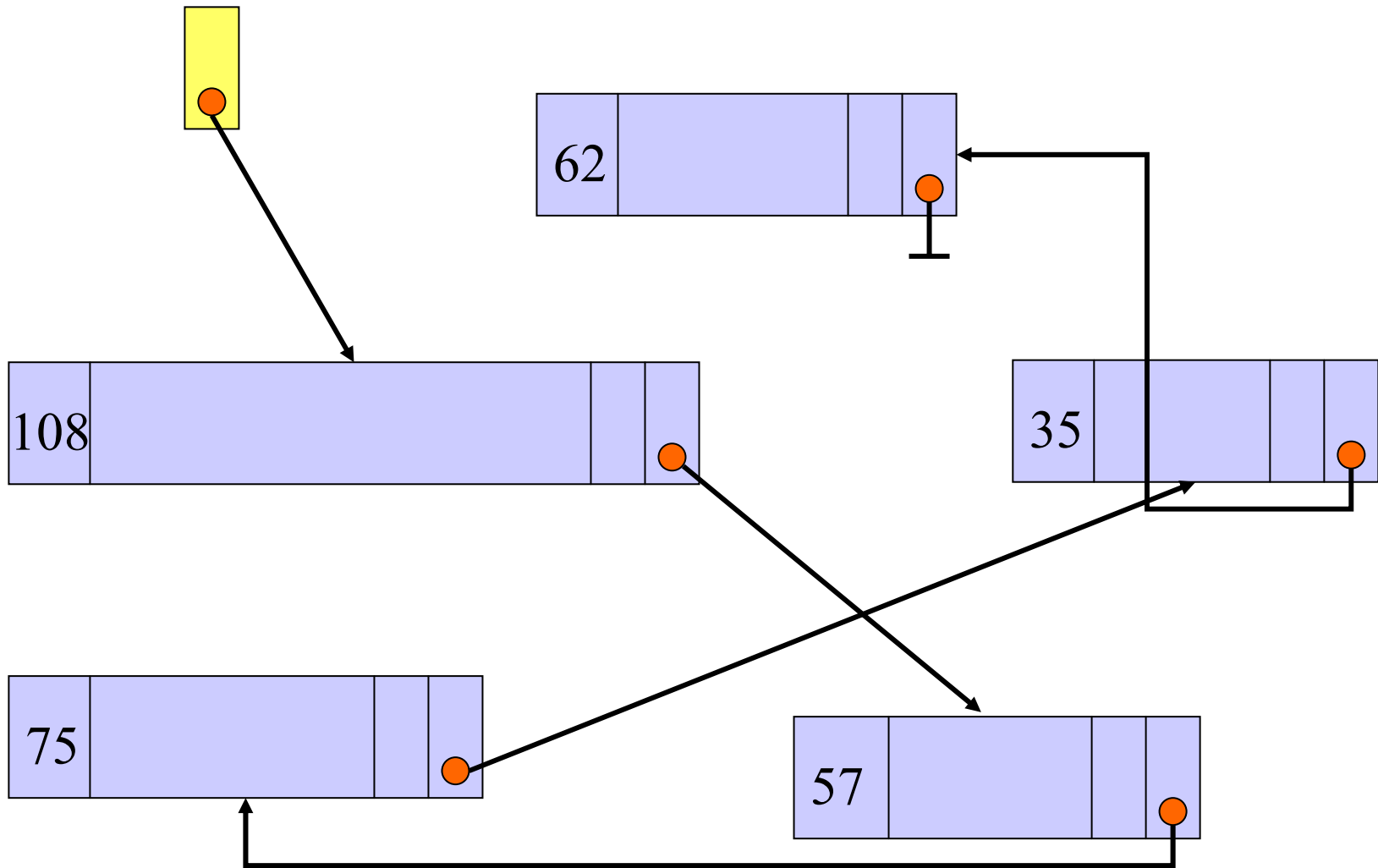
next

ein Datenblock



Freispeicherliste

Die freien Datenblöcke sind irgendwo in der Halde verteilt. Dazwischen liegen irgendwo die von Programmen belegten Datenblöcke.



12.3.3 Bearbeitungsstrategien: Ein Programm fordert einen Datenblock mit m Speicherplätzen an.

Algorithmus 1: First Fit

Gehe die Freispeicherliste durch, bis ein Datenblock D mit $\text{größe} \geq m$ gefunden ist.

Mache hieraus zwei Datenblöcke: Einen mit $m+x$ und einen mit $\text{größe}-m-x$ Speicherplätzen (x = Speicherplatz für größe , mark und next , siehe Datentyp DBlock in 12.3.2).

Füge diese beiden Datenblöcke in die Freispeicherliste anstelle des DBlocks D ein.

Klinke den ersten dieser beiden Datenblöcke aus der Freispeicherliste aus und ordne ihn dem Programm zu.

Hinweise: Falls der zweite Block "zu klein" ist, vermeide die Aufspaltung in zwei Datenblöcke und weise ganz D dem Programm zu. Falls kein geeigneter Block D existiert, rufe die Speicherbereinigung auf, siehe unten.

Algorithmus 2: Best Fit

Gehe die gesamte Freispeicherliste durch und ermittle den kleinsten Datenblock D mit $\text{größe} \geq m$.

Fahre anschließend fort wie bei "First Fit".

Welche Strategie ist besser?

Bei beiden Methoden entstehen im Lauf der Zeit viele kleine Datenblöcke, die verstreut in der Halde liegen. Diese sog. "**Fragmentierung**" des Speichers erfordert häufige Aufrufe der Speicherbereinigung. In der Praxis erweist sich die Best-Fit-Strategie gegenüber der "First-Fit-Strategie" nach einiger Zeit als schlechter (!), da hierbei besonders kleine Datenblöcke entstehen; außerdem muss bei Best-Fit stets die gesamte Freispeicherliste durchlaufen werden.

Aus der Praxis weiß man: Solange der freie Speicher etwa ein Drittel der Halde ausmacht, ist die First-Fit-Strategie gut anwendbar. Wird aber der freie Platz geringer, so muss oft eine zeitaufwändige Speicherbereinigung durchgeführt werden, die zu **Wartezeiten bei den "Kunden"** führt.

Recht nachteilig ist die Zeit, die beim Durchlaufen der Liste verstreicht. Zwei Ideen zur Verbesserung:

- Halte die Freispeicherliste stets nach der Größe der Datenblöcke sortiert. Nachteil: Das Einfügen freigegebener Speicherbereiche ist dann aufwändiger, dafür erfolgt aber die Suche nach einem passenden DBlock im Mittel schneller.
- Lege einen binären Suchbaum über die Freispeicherliste. Die Suche erfolgt dann schneller. Nachteile: Weiterer Speicherplatz; diese Suchbäume müssen ebenfalls in der Halde untergebracht werden, da sie dynamische Datenstrukturen sind; sie müssen eventuell ständig geändert werden.

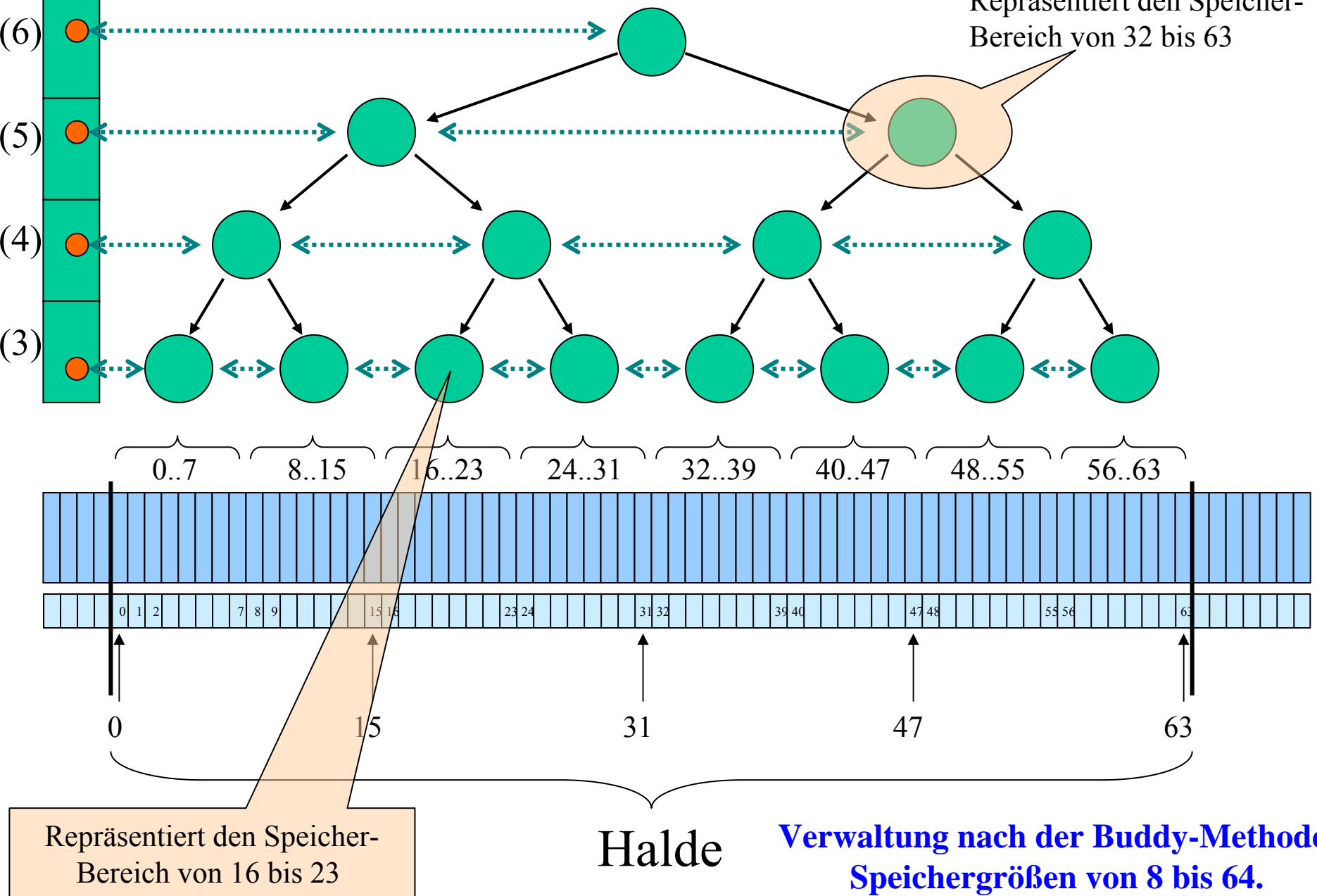
12.3.4 Buddy-Verfahren

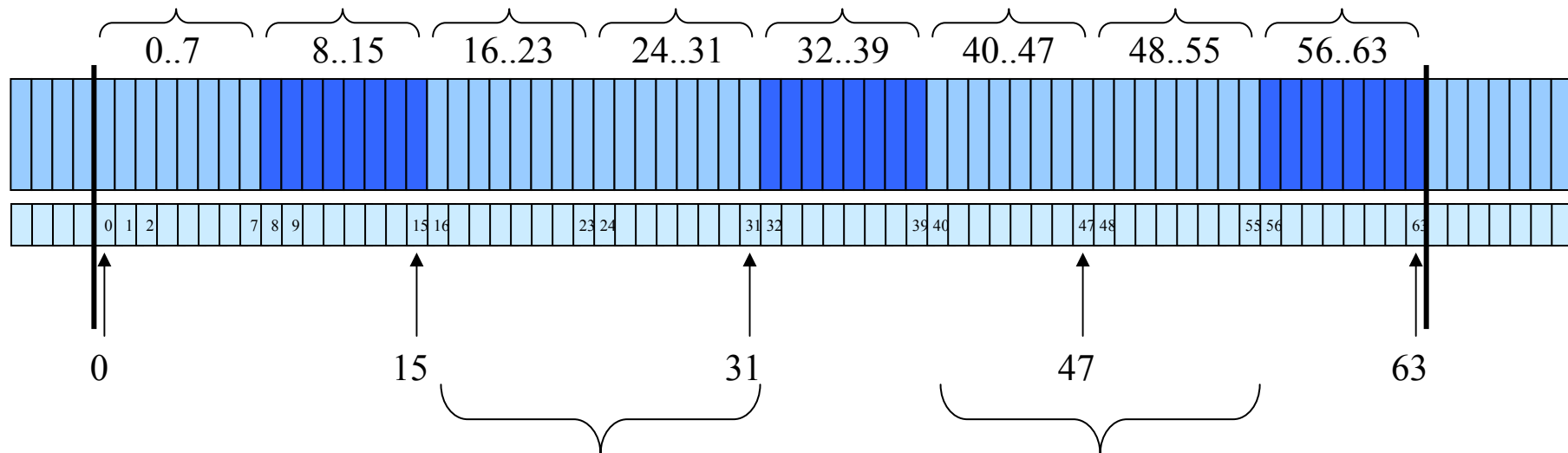
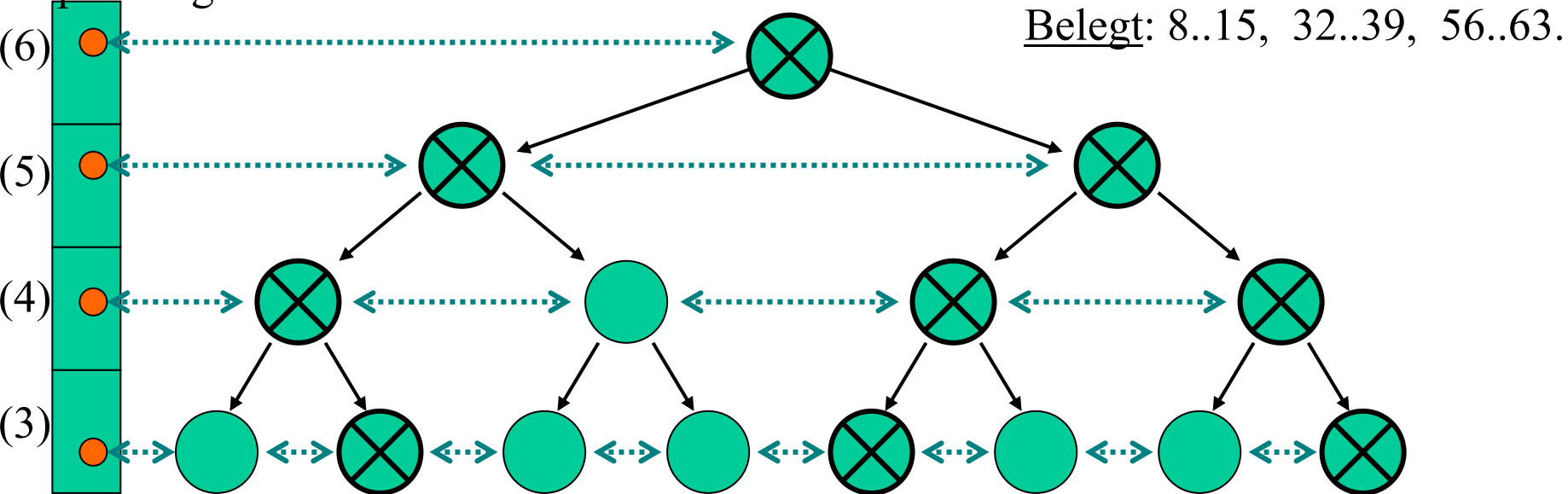
Wir stellen kurz einen alten Vorschlag zur Verwaltung des freien Speicherplatzes anstelle einer Freispeicherliste vor, der leicht implementiert werden kann, aber gewisse Nachteile besitzt, die sog. **Buddy-Methode**. (Buddy = (engl.) Kamerad, Kumpel.)

Idee: Das Verfahren legt einen gleichverzweigten binären Baum über den Speicher (also über die Halde). Das Aufspalten und das Verschmelzen sind aber nicht beliebig möglich.

Vorgehen: Die Halde wird in Datenblöcke unterteilt, deren Länge jeweils eine Zweierpotenz ist. Jeder Datenblock muss genau 2^k Speicherplätze belegen für eine natürliche Zahl k mit $min_k \leq k \leq max_k$. (Man schränkt in der Praxis k ein z.B. zwischen $min_k = 10$ und $max_k = 25$.) Jedem Datenblock wird genau einer seiner beiden Nachbarn als "Buddy" zugeordnet.

Speichergr: array (*mink..maxk*) of ... ;





Dieser Speicherplatz steht für 16 Speicherplätze zur Verfügung

Dieser Speicherplatz steht für 16 Speicherplätze **nicht** zur Verfügung

Wir nummerieren die Halde also von 0 bis $2^{maxk} - 1$ durch. Die kleinste Blockgröße sei 2^{mink} . Alle Speicherblöcke mit festem $mink \leq k \leq maxk$ stehen in einer Liste, erreichbar über den Zeiger des Feldelements $Speichergr(k)$.

Zu jedem Speicherblock, der an der Adresse x beginnt und die Größe 2^k besitzt, sei **buddy_k(x)** die Anfangsadresse seines Buddy (dieser liegt entweder links oder rechts von ihm und besitzt die gleiche Größe).

Es gilt:

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, & \text{falls } x = 0 \pmod{2^{k+1}} \\ x - 2^k, & \text{falls } x = 2^k \pmod{2^{k+1}} \end{cases}$$

Wir programmieren diese Form der Speicherverwaltung nicht aus, sondern geben nur die Vorgehensweisen an.

Beginn: Anfangs werden alle Speicherbereiche als frei markiert.

Speicheranforderung: Ein Programm fordert einen Datenblock der Größe m an. Berechne k so, dass $2^{k-1} < m \leq 2^k$ gilt.

Suche: Die Speicherverwaltung durchläuft dann die Liste, die über $\text{Speichergr}(k)$ erreichbar ist. Diese Liste hat 2^{maxk-k} Knoten.

Zuordnung und Aktualisierung: Wird hier ein freier Speicherbereich gefunden, so wird er dem Programm zugewiesen; zugleich werden dieser Bereich, **alle Knoten in seinem Unterbaum** und seine Vorgänger im Baum bis zur Wurzel als belegt markiert.

Ablehnung und "Wiedervorlage": Wird kein freier Speicherbereich gefunden, so lege die Speicheranforderung in einer Warteschlange des Systems ab, sende dem Programm einen "Wartehinweis" und prüfe später erneut.

So realisiert man es natürlich nicht, sondern...? Und warum?

ok, aber ...

Beispiel: Wird in der Situation der obigen Folie ein Bereich der Größe 14 angefordert, so ist $k = 4$ und es wird ausgehend von $\text{Speichergr}(4)$ die Liste der Speicherblöcke der Größe $2^4 = 16$ durchsucht. Bereits der zweite Block ist frei, so dass dem anfordernden Programm der Bereich 16..31 zugewiesen wird.

Speicherfreigabe: Ein Programm gibt einen Datenblock der Größe 2^k wieder zurück. Dieser Block wird in der Liste zu $\text{Speichergr}(k)$ als frei markiert. Ist sein Buddy frei, so wiederhole diesen Vorgang mit seinem Vorgängerknoten.

Beispiel: Wird in der Situation der obigen Folie der Bereich 8..15 der Größe 8 freigegeben, so kann dieser Block, aber auch sein Vorgänger und dessen Vorgänger frei gegeben werden, so dass anschließend die linken drei als belegt markierten Knoten im Baum wieder als frei markiert sind.

Vorteile der Buddy-Methode:

- Einfach zu handhaben und leicht zu programmieren.
- Das Verfahren bewährt sich in der Praxis hinreichend gut.

Nachteile:

- Benachbarte Bereiche, die nicht Buddys sind, können nicht verschmolzen werden, und es gibt nicht genutzte Speicherbereiche (Fragmentierung), da immer nur Blöcke von der Länge einer Zweierpotenz zugewiesen werden können.
- Es können Verklemmungen entstehen, wenn zwei Programme Speicher nachfordern, die nicht verfügbar sind. (Man kann dies durch Überwachung durch das Betriebssystem abfangen, oder man kann verbieten, dass ein Programm eine zweite Speicheranforderung stellt, was aber bei rekursiven oder nebenläufigen Programmen nicht sinnvoll ist.)

12.3.5 Speicherbereinigung (garbage collection)

Wir nehmen nun an, die Programme legen immer mehr dynamische Datenstrukturen (also: verzeigerte Strukturen) in der Halde an. Es ist absehbar, dass in Kürze kein Speicherplatz mehr zur Verfügung steht.

Nun muss geprüft werden, ob die Datenobjekte, die in der Halde stehen, wirklich alle benötigt werden oder ob man sie löschen und auf diese Weise neuen Speicherplatz bereitstellen kann. Dies ist die Aufgabe der "Speicherbereinigung", die in der Regel automatisch erfolgt.

Standardtechniken zur Lösung dieser Aufgabe sind:
Verweiszählermethode, Graphdurchlauf und Kopieren.

(Hinweis: In Ada 95 gibt es keine Speicherbereinigung, vielmehr muss alles explizit mit FREE freigegeben werden.)

12.3.6 Verweiszählermethode ("Reference Counting")

Wenn die Zeigerstrukturen keine Kreise bilden (also "azyklisch" sind), dann kann man in jeden Knoten einen "Verweiszähler" aufnehmen, der angibt, wie oft auf dieses Objekt verwiesen wird. Wird ein Knoten mit k Zeigern hinzugefügt, so müssen die Verweiszähler der k Knoten, auf die diese Zeiger zeigen, jeweils um 1 erhöht werden. Wird ein Knoten gelöscht, so muss man die Verweiszähler in den k Objekten, auf die die Zeiger zeigten, um jeweils 1 erniedrigen. Wird ein Verweiszähler hierbei 0, dann muss man auch in allen ihren nachfolgenden Knoten den Verweiszähler um 1 erniedrigen. Entsprechende Operationen kann der Compiler in den übersetzten Code einfügen, ohne dass der Programmierer hiervon etwas bemerkt. Benötigt man irgendwann Speicherplatz, so kann man zu einem gegebenen Zeitpunkt genau alle die Knoten löschen, deren Verweiszähler 0 ist. Bei zyklischen Strukturen funktioniert dieses einfache Verfahren aber nicht mehr. (Klar!?)

12.3.7 Graphdurchlauf

Hier verfolgt man alle Zeiger, die von den lokalen Speichern der Programme ausgehen, und markiert alle Datenobjekte, die auf diese Weise auf irgendeinem Weg erreichbar sind. Die nicht-markierten Datenobjekte kann man löschen.

Die Halde ist nichts anderes als ein großer Graph mit mehreren Einstiegspunkten (= den Zeigern der lokalen Speicher in den Programmen). Ausgehend von diesen Einstiegspunkten wird der Graph mit den bekannten Techniken (3.8.7 und 8.8.9) oder gewissen Varianten durchlaufen, wobei die erreichbaren Knoten mit "true" markiert werden.

Das Verfahren besteht aus zwei Teilen:

- Markiere alle erreichbaren Objekte,
- Entferne alle nicht-markierten Objekte.

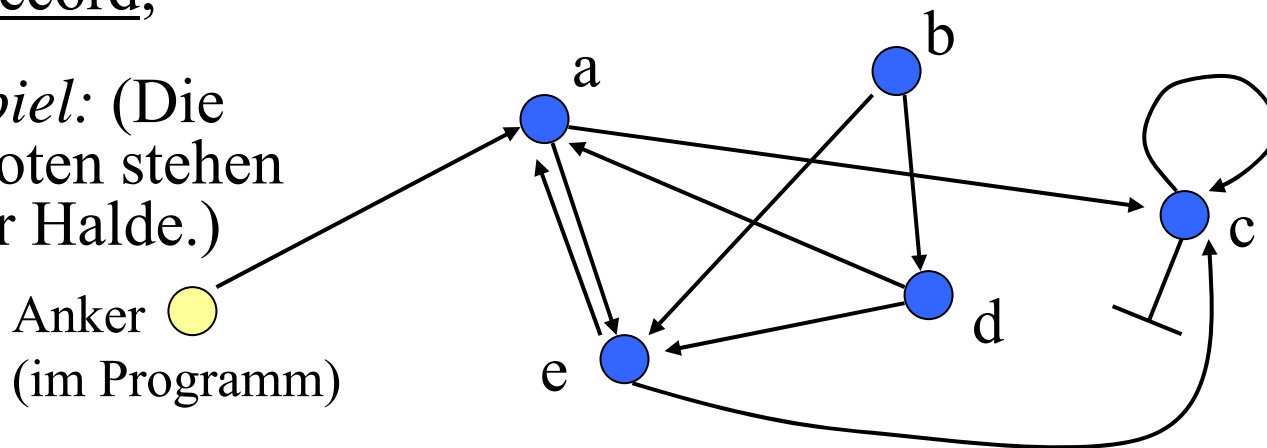
Wir behandeln nur den ersten Teil, da der zweite ein einfacher Halden-Durchlauf mit Umhängen in die Freispeicherliste ist.

Vereinfachung: Um das Vorgehen zu erläutern, genügt es, Datenobjekte mit zwei Zeigern zu betrachten, also Objekte des folgenden Typs "Knoten":

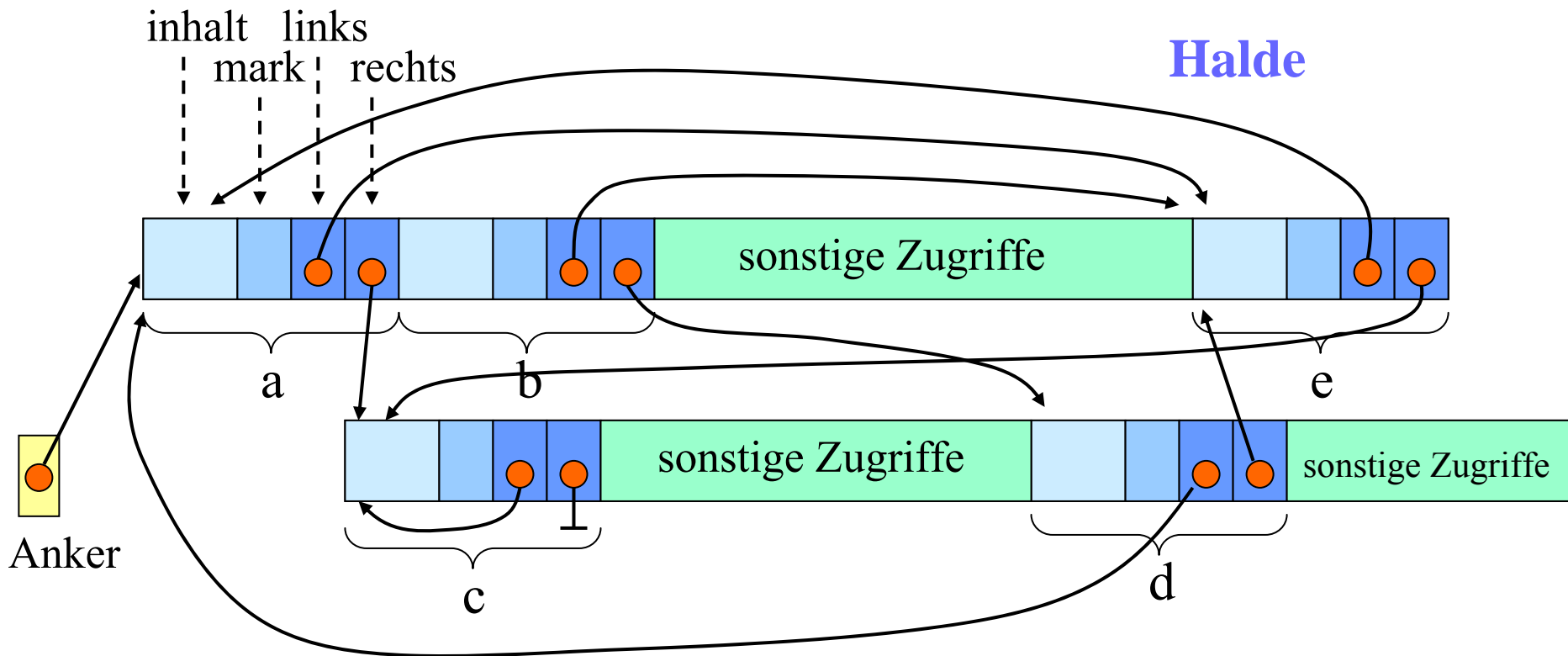
```
type Knoten;  
type Kante is access Knoten;  
type Knoten is record  
    inhalt: ...  
    mark: Boolean;  
    links, rechts: Kante;  
end record;
```

Beispiel: (Die 5 Knoten stehen in der Halde.)

Anker (im Programm)

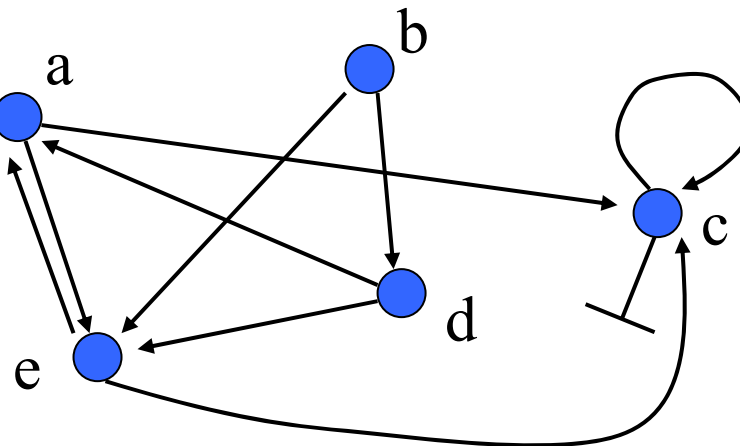


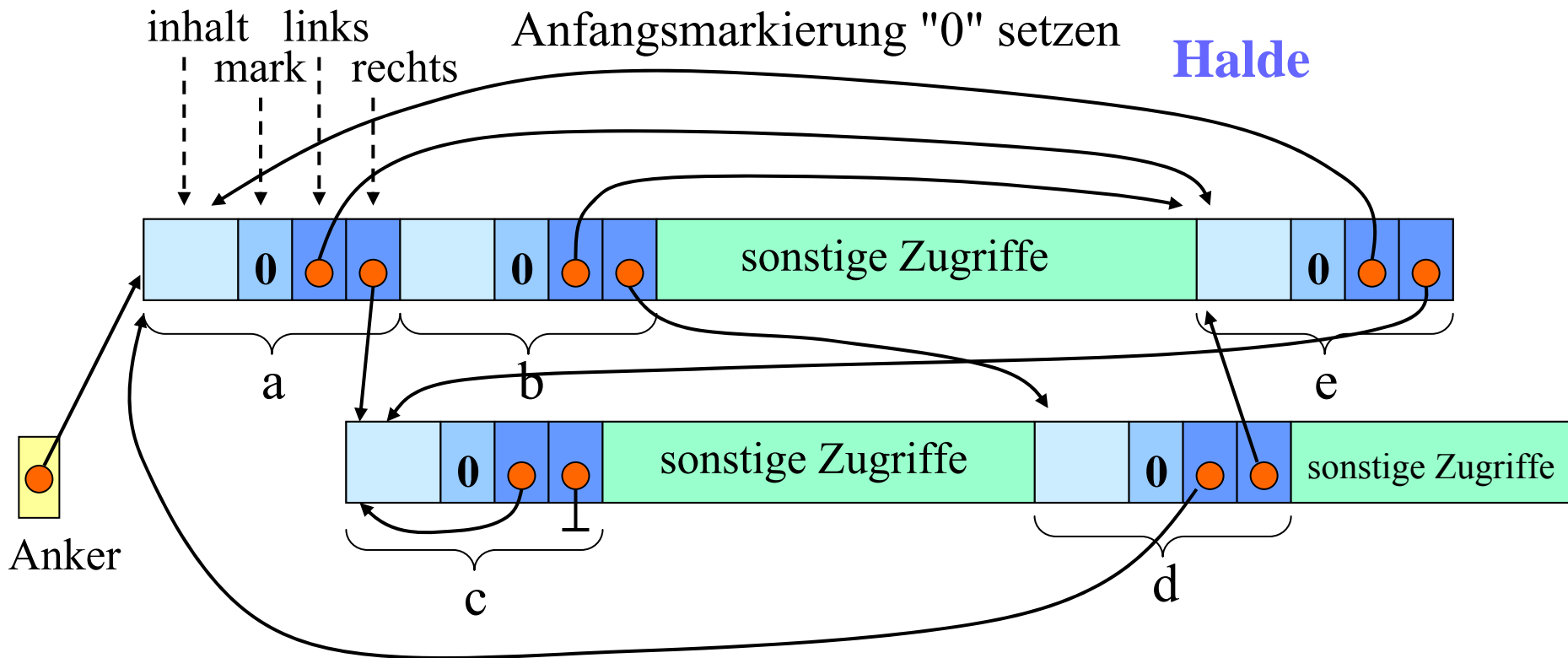
Die Knoten b und d sind vom Programm aus nicht erreichbar.



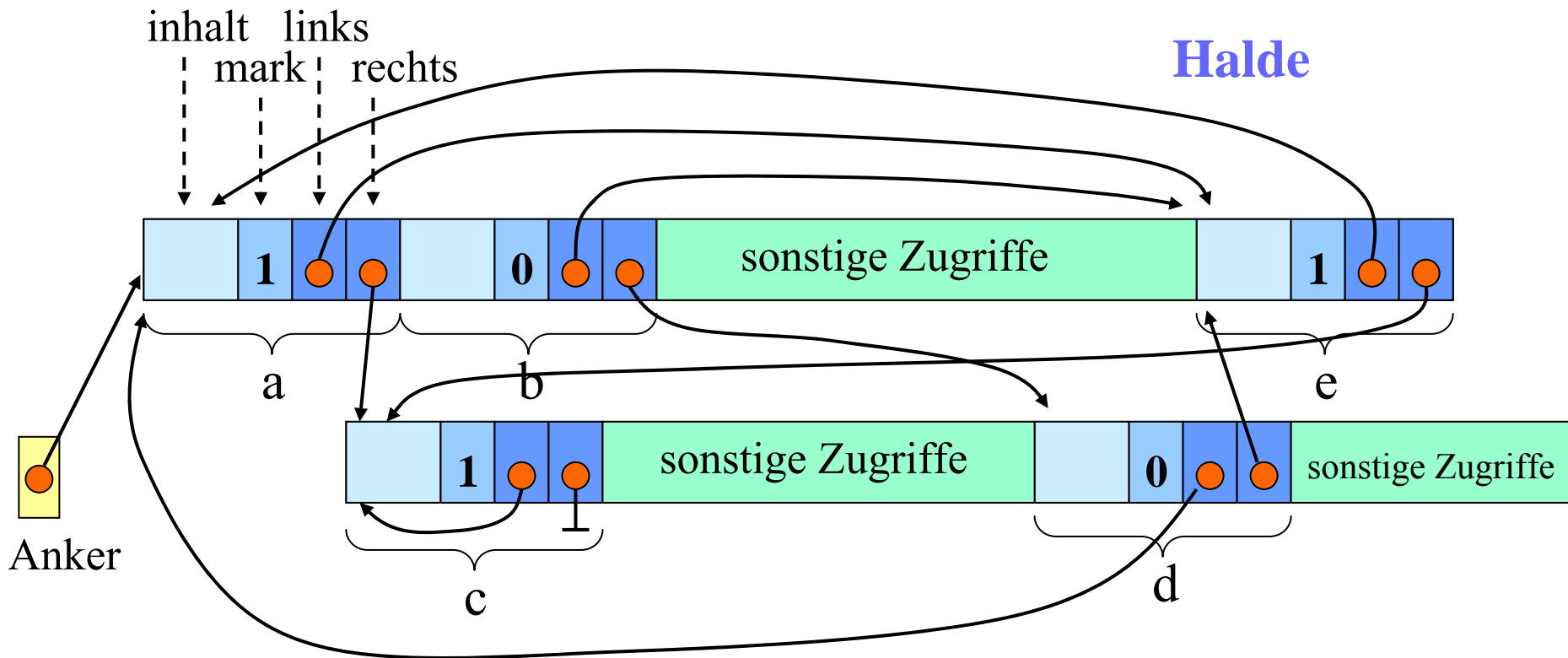
Beispiel: (Die 5 Knoten stehen in der Halde.)

Anker (im Programm)





Ziel muss es nun sein, die Knoten b und d als löschbare Knoten zu erkennen. Hierzu durchläuft man ausgehend von "Anker" alle Zeiger und markiert die erreichten Knoten mit true (oder einer "1"). Dies geschieht für alle "Anker", die aus den lokalen Speichern in die Halde verweisen. Danach löscht man alle mit false (oder "0") markierten Objekte und schiebt ggf. den Speicher zusammen.



Ergebnis des Durchlaufs: Ausgehend von "Anker" wurden alle Zeiger nachverfolgt und die hierbei erreichten Knoten mit einer "1" markiert; die nicht erreichbaren bleiben mit "0" markiert.

Anschließend kann man den Speicherbereich neu organisieren, sofern man möglichst große Freispeicherbereiche benötigt.

12.3.8 Wir kommen nun zum **Algorithmus zur Markierung der erreichbaren und der unerreichbaren Knoten** unter der Annahme, dass kein freier Speicherplatz für den Algorithmus zur Verfügung steht:

Schritt 1:

Markiere alle Knoten in der Halde mit "false" (bzw. mit 0).

Schritt 2:

Markiere alle Knoten in der Halde, die von einem der lokalen Speicher direkt erreicht werden können, mit "true" (bzw. mit 1).

Schritt 3 (eigentlicher Algorithmus): Die Halde möge von Adresse 0 bis Adresse M im Speicher nummeriert sein. Jeder Knoten möge genau r Speicherplätze (Adressen) belegen.
 u, v sind vom Typ Knoten, i und j sind Adressen in der Halde.

```

i := 0;                                -- i ist die Adresse des betrachteten Knotens
while i <= M loop
    j := i + r;                          -- j wird die Adresse des nächsten Knotens
    if der Knoten mit Adresse i ist mit "true" markiert
        and then der Knoten mit Adresse i besitzt mindestens einen
            Nachfolger (d.h.: (links /= null) or (rechts /= null))
    then if (links /= null) and then (der Knoten u, auf den links
        verweist, ist mit "false" markiert)
        then markiere den Knoten u mit "true";
            j := Minimum (j, Adresse von u); end if;
    if (rechts /= null) and then (der Knoten v, auf den rechts
        verweist, ist mit "false" markiert)
    then markiere den Knoten v mit "true";
        j := Minimum (j, Adresse von v); end if;
    end if;
    i := j;                             -- zum nächsten Knoten gehen
end loop;

```

Idee dieses Vorgehens: Durchlaufe die Knoten von vorne nach hinten in der Halde. Es interessieren nur die mit true markierten Knoten (nur sie sind bisher vom Programm aus erreichbar). Betrachte deren beide Nachfolgeknoten. Markiere sie mit true und setze das Verfahren an der minimalen Adresse der drei Knoten

- nächster Knoten in der Halde
 - linker Nachfolgeknoten
 - rechter Nachfolgeknoten
- fort.

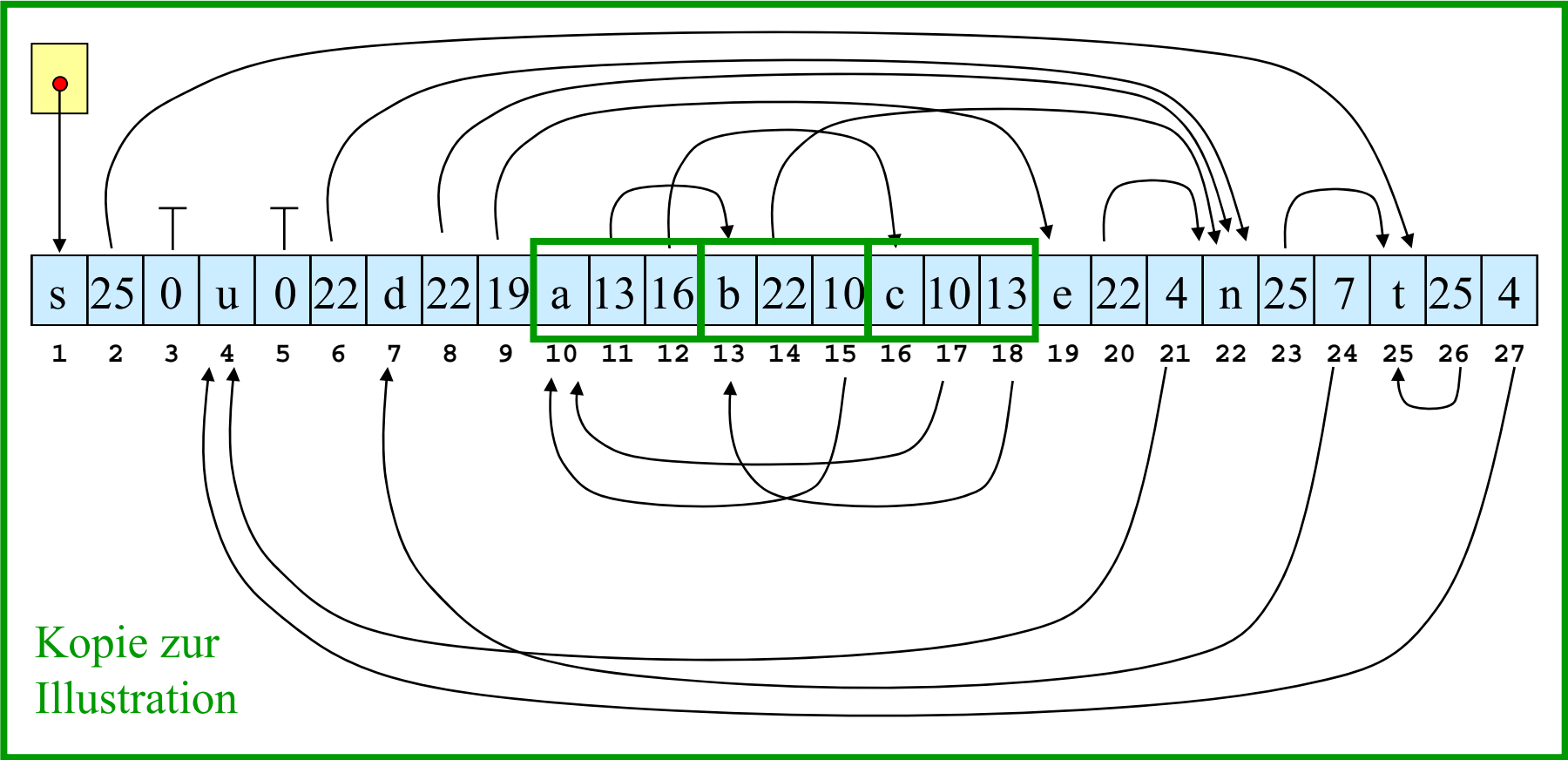
Auf diese Weise gelangt man schließlich an alle erreichbaren Knoten. Beachte: Dieser Algorithmus ist eine rekursionsfreie Variante des Graphdurchlaufs, der in 3.8.7 beschrieben wurde. Um ihn zu verstehen, müssen Sie ihn präzise nachvollziehen.

Beispiel 12.3.9:



nicht erreichbare Plätze

s	25	0	u	0	22	d	22	19	a	13	16	b	22	10	c	10	13	e	22	4	n	25	7	t	25	4
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27



Beispiel:



$r=3$

nicht erreichbare Plätze

s	25	0	u	0	22	d	22	19	a	13	16	b	22	10	c	10	13	e	22	4	n	25	7	t	25	4
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

Ablaufdiagramm für i , j und "Markierung auf true setzen":

i	j	Mark.	i	j	Mark.	i	j	Mark.
		1		4	4		10	19
1	4		4	7		10	13	
	4	25		7	22	13	16	
4	7		7	10		16	19	
7	10		10	13		19	22	
10	13		13	16			22	
13	16		16	19		22	25	
16	19		19	22		25	28	
19	22		22	25		28		
22	25			7	7	Ende		
25	28		7	10				

Dies ist bereits
der worst case:
Es gibt viele
Verweise vom
Ende der Hal-
de nach vorne
(vgl. Komple-
xitätsabschät-
zung unten).

Aufwand dieses Verfahrens? (Im worst case quadratisch in M.)

Im ungünstigsten Fall beim Durchlauf durch die Halde ist die if-Bedingung erst beim letzten Knoten erfüllt und dessen Verweis führt auf den ersten Knoten zurück, siehe obiges Beispiel.

Nach dem zweiten Durchlauf geschieht das Gleiche mit dem vorletzten Knoten usw.

Wenn n die Zahl der Knoten in der Halde ist, so würde man also $n + (n-2) + (n-4) + \dots + 3 + 1 \approx n \cdot (n+1)/4 = O(n^2)$

Schritte ausführen müssen. Wegen $n \approx M/r$ erhält man ein $O(M^2)$ -Verfahren. Die Speicherplatzkomplexität ist dagegen konstant (M = Anzahl der Speicherplätze in der Halde).

In der Tat erweist sich dieser Algorithmus in der Praxis auch im Mittel als ein quadratisch mit M wachsendes Verfahren.

Hinweis:

Es ist klar, wie man dieses Verfahren auf Knoten, die mehr als zwei Nachfolger haben können oder deren Größe im Datenobjekt selbst gespeichert ist, erweitern kann:

- for all Nachfolgeknoten (im äußersten then-Teil),
- ersetze $j := i+r$ durch $j := i + \text{größe_des_aktuellen_Knotens}$.

Das oben genannte Verfahren eignet sich besonders dann, wenn man (fast) keinen freien Speicherplatz mehr zur Verfügung hat. Gibt es dagegen noch Speicherplatz, den man für einen Keller S nutzen kann, dann empfiehlt sich folgender deutlich schnellere Algorithmus, der die weiter zu verfolgenden Zeiger im Keller S ablegt (machen Sie sich klar: dies ist der Algorithmus GD aus 3.8.7 beschränkt auf Ausgangsgrad 2):

12.3.10 Kellerverfahren

Schritt 1: Markiere alle Knoten in der Halde mit "false".

Schritt 2: Markiere alle Knoten in der Halde, die von einem lokalen Speicher direkt erreicht werden können, mit "true" und lege sie (in der Praxis: ihre Adressen) im Keller S ab.

Schritt 3: K sei vom Typ Kante (= "Zeiger auf Knoten").

while not isempty(S) loop

while not isempty(S) and (top(S) hat keinen Nachfolger)

loop pop(S); end loop;

if not isempty(S) then

 K := top(S); pop(S);

if (K.links /= null) and then (not K.links.mark) then

 K.links.mark := true; push(S, K.links); end if;

if (K.rechts /= null) and then (not K.rechts.mark) then

 K.rechts.mark := true; push(S, K.rechts); end if;

end if;

end loop;

Aufwand dieses Keller-Verfahrens? (Im worst case linear in M .)

Das Verfahren durchläuft jeden Zeiger, der in einem Knoten auftritt, höchstens einmal. Da es höchstens doppelt so viele Zeiger wie Knoten gibt, handelt es sich bei Schritt 3 also um ein $O(n')$ -Verfahren (n' = Zahl der erreichbaren Knoten in der Halde).

Allerdings bezahlt man diese Schnelligkeit mit dem benötigten Speicherplatz für den Keller S . Dieser kann bis zu $n/2$ Knoten groß werden. Im Mittel wird man aber deutlich weniger Platz brauchen.

Die uniforme Zeitkomplexität dieses Keller-Verfahrens wird also vor allem durch Schritt 1 bestimmt, welcher M/r Zeiteinheiten benötigt. Insgesamt ergibt sich damit ein $O(M)$ -Verfahren sowohl bzgl. der Zeit als auch bzgl. des Platzes.

Man kann nun die beiden Algorithmen kombinieren:

Variante 1: Solange noch genügend Platz für den Keller S vorhanden ist, arbeite nach dem Kellerverfahren. Sobald der Keller überläuft, suche man die kleinste Adresse im Keller und schalte mit ihr beginnend auf das andere Verfahren um.

Variante 2: Man verwende statt des Kellers eine sortierte Liste, worin die Zeiger geordnet nach ihrer Adresse eingetragen werden und verwende stets den Zeiger mit der kleinsten Adresse als nächsten zu untersuchenden Knoten. Da die Liste bei jedem Eintrag durchlaufen werden muss, wird dieses Vorgehen zu einem $O(n^2)$ -Verfahren und damit letztlich zu einem $O(M^2)$ -Verfahren.

Überlegen Sie sich weitere Kombinationen, Varianten oder Verbesserungen. Ist es z.B. sinnvoll, zuerst den Nachfolgeknoten mit der größeren Adresse in den Keller S zu legen?

Man erkennt nun auch die Abhängigkeit der Speicherbereinigung von der jeweiligen Programmiersprache: Man muss wissen, wie die Datenobjekte / Blöcke / Knoten usw. aufgebaut sind, um Zeiger auch als Zeiger erkennen zu können. Andererseits kann natürlich das Betriebssystem ein universelles Datenformat vorgeben, in dem zum Beispiel die Informationen über Zeiger und die Markierungen an vorgegebenen Stellen notiert werden müssen.

Siehe auch 12.4.

Zum Durchlaufalgorithmus in Linearzeit ohne einen Keller vgl. den Durchlauf durch binäre Bäume (Stichwort: Schorr-Waite-Algorithmus, siehe Literatur) bzw. Hinweise in den Übungen.

Nachdem wir nun die erreichbaren Knoten bzw. Datenblöcke mit "true" markiert haben, kann man alle anderen in die Freispeicherliste (oder in die Buddy-Verwaltung) eintragen und normal weitermachen.

Oft möchte man zusätzlich den Speicher "zusammenschieben" ("**kompaktifizieren**") und dabei die im Laufe der Rechnungen entstandenen kleinen Fragmente (= nicht nutzbaren freien Speicherbereiche) beseitigen. Dieser Kompaktifizierungs-Algorithmus ist bei beliebiger Verzeigerung aufwändig. Man kann Verschiebe-Zeiger mitführen, die die Lage nach der Kompaktifizierung angeben. Dies ist insbesondere bei der Kopiermethode (12.3.11) vorteilhaft.

Diese und weitere Fragen zur Verwaltung von Programmen und Daten lernen Sie in Vorlesungen über Betriebssysteme oder auch in speziellen Praktika kennen.

12.3.11 Kopiertechniken

Man verwendet aktuell immer nur den halben Speicher. Läuft dieser über, so kopiert man (ausgehend von den Zeigern in den lokalen Speichern der Programme) die erreichbaren Objekte in die andere Hälfte des Speichers, wobei man die relative Anordnung unverändert lässt. Auf diese Weise werden die nicht-markierten Objekte zu Lücken im neuen Speicher und können in die Freispeicherliste eingetragen werden.

Mit etwas erhöhtem Aufwand können die Objekte beim Kopieren von vorne nach hinten nacheinander abgelegt werden, wodurch zugleich der Speicherplatz optimal genutzt wird. In mehreren Durchläufen können hierbei die Verweise entsprechend der neuen Anordnung umgesetzt werden.

(Überlegen Sie, wie so etwas geschehen könnte! Z.B. mit zusätzlichen Verweiszeigern, die den neuen Speicherplatz im alten Objekt speichern.)

Um sich in dieses Problem hineinzudenken, sollten Sie die Frage lösen, wie man einen Graphen kopiert.

Dies sieht einfach aus, hat aber seine Tücken, wenn man keine Zusatzzeiger mitführen oder (wegen fehlendem Speicherplatz) keine Rekursion verwenden darf.

12.4 Historische Hinweise

Mit der Entwicklung der ersten Computer in den 1940er Jahren entstanden auch sogleich eindimensionale Felder, da diese genau die Speicherstruktur wiedergaben. Allgemeine Felder finden sich bereits in den Programmiersprachen der 1950er Jahre (Fortran 58, Algol 60, APL). Verbunde und Folgen von Buchstaben treten in Cobol auf (ab 1961). Verschiedene Konzepte der Datenstrukturen wurden in Algol 68 (Standard: 1975) zusammengeführt. Dessen "gut verständlicher" Anteil wurde von Nikolaus Wirth in die Sprache PASCAL (1972) eingebracht, die bis heute als "didaktisches Vorbild" für Programmiersprachen gilt. (Deren Sprachelemente gehören zum Kern des "Programmieren im Kleinen" und sind auch in Ada enthalten.)

Die Datenstruktur "Keller" entstand ca. 1954 mit ersten Arbeiten über die korrekte Auswertung von arithmetischen Ausdrücken. Sie wurde zugleich ab 1960 verwendet, um den Aufruf von (auch rekursiven) Prozeduren und generell alle klammerartigen Strukturen in Programmen und Programmiersprachen korrekt zu implementieren.

Listen bilden die Grundlage der Programmiersprache LISP (McCarthy, ab 1959). Statt der expliziten Zeiger wurden Operationen wie "head" und "tail" für den Zugriff auf das erste Element einer Liste bzw. auf die "Restliste ohne das erste Element" benutzt. In SIMULA und PL/I (beide ab 1965) konnten Zeiger verwendet werden. Die Unterscheidung zwischen einem statischen und einem dynamischen Speicher geschah bereits in den ersten Programmiersprachen; die Halde wurde in SIMULA (Koroutinenkonzept) und PL/I erforderlich.

Dass sehr allgemeine Datenstrukturen korrekt übersetzbar sind, demonstrierten die Compiler von SIMULA 67 und etwas später von PL/1. Probleme bereiteten aber die ganz allgemeinen Datenstrukturen von Algol 68, bei denen kartesische Produkte, Vereinigungen, Referenzen, Potenzmengen, Funktionenbildung usw. beliebig miteinander verknüpft werden können: Die Laufzeitsysteme wurden derart kompliziert, dass jeder Algol-68-Compiler gewisse Einschränkungen machen musste.

Für die automatische Speicherbereinigung des Betriebssystems ist es unverzichtbar, *einen Zeiger auch als Zeiger zu erkennen!* Hierzu legt der Compiler (unsichtbar für den Benutzer) für jeden Zeiger einen "Deskriptor" an, aus dem die Struktur des referenzierten Objektes (sein Typ einschl. der Weiterverweise und das Markierungsbit) zu ersehen ist.

Mitte der 1960er Jahre entstanden die ersten Betriebssysteme, die mehrere Programme gleichzeitig verwalten konnten. Ab dieser Zeit entwickelte man diverse Verfahren für die Speicher-
verwaltung (wie Multikeller, Freispeicher, Bereinigung usw.). Die in diesem Kapitel 12 behandelten Vorgehensweisen sind also bereits als "klassische Verfahren" einzustufen.

Für *Echtzeitsysteme* und für *Verteilte Systeme* wurden in den 80er und 90er Jahren neue Verfahren entwickelt. Echtzeitsysteme z. B. können nicht einfach unterbrochen werden, so dass die Standard-
verfahren zu "inkrementellen Techniken" erweitert wurden: Während des Programmablaufs läuft ein Prozess mit, der nicht erreichbare Objekte aufspürt und in die Freispeicherliste einträgt. Die beiden Prozesse dürfen allerdings nicht gleichzeitig auf gewisse Teile zugreifen ("wechselseitiger Ausschluss"), was der Compiler automatisch in das übersetzte Programm einfügt.

Anhang: Ein paar Literaturhinweise

Literaturangaben im Grundstudium sind immer subjektiv verfärbt. Es gibt weitere gute Lehrbücher, auf die ich aber oft nur deshalb nicht direkt zurückgegriffen habe, weil sie zufällig nicht in meinem Arbeitszimmer, sondern irgendwo anders standen oder weil ich als Herausgeber einer Lehrbuchreihe ein anderes Buch soeben besonders sorgfältig durchgearbeitet hatte. Die folgende Auflistung besitzt daher keinen Anspruch auf Vollständigkeit und enthält keine Wertungen.

- Manuskripte: V.Claus (WS 03/04, SS 2005), K.Lagally (WS 01/02 und 04/05), E.Plödereder (SS 01).
- Appelrath, Hans-Jürgen und Ludewig, Jochen, "Skriptum Informatik - eine konventionelle Einführung", Verlag der Fachvereine Zürich und B.G. Teubner Stuttgart, 4. Auflage 1999
- Balzert, Helmut, "Lehrbuch Grundlagen der Informatik", Elsevier, 2. Auflage, 2004
- Broy, Manfred, „Informatik. Eine grundlegende Einführung“. Band 1: Programmierung und Rechnerstrukturen, Springer, 1998. Band 2: Systemstrukturen und Theoret. Informatik, Springer-Verlag, 1998
- Cormen, Leiserson, Rivest, "Introduction to Algorithms", MIT Press, 1996 (zweite Auflage 2001), auch auf Deutsch erhältlich
- Goos, Gerhard, "Vorlesungen über Informatik", Band 1 und 2, dritte Auflage, Springer-Verlag, Berlin 2000 und 2001 (vierte Auflage, zusammen mit Wolf Zimmermann, 2005/2006)
- Gumm, H.-P., Sommer, M., „Einführung in die Informatik“, 7. Auflage, Oldenbourg-Verlag, München 2006
- Güting, R.H., "Datenstrukturen und Algorithmen", B.G.Teubner, 3. Auflage, Wiesbaden 2004
- Hotz, Günter, "Einführung in die Informatik", Teubner-Verlag, Stuttgart, 1992
- Klaeren, H., Sperber, M., "Vom Problem zum Programm", 3. Auflage, B.G. Teubner Stuttgart und Wiesbaden, 2001
- Loeckx, J., Mehlhorn, K., Wilhelm, R., "Grundlagen der Programmiersprachen", Teubner, Stuttgart 1986
- Ottmann, T., Widmayer, P., "Algorithmen und Datenstrukturen", Spektrum Verlag, Heidelberg, 4. Auflage 2002
- Saake, G., Sattler, K.-U., "Algorithmen und Datenstrukturen", d.punkt-Verlag, Heidelberg, 2004
- Schöning, Uwe, "Algorithmik", Spektrum Akademischer Verlag, Heidelberg 2001
- Sedgewick, Robert, "Algorithms in C", 3rd Edition, Addison-Wesley, 1998, sowie "Algorithms in Java", Addison-Wesley, 1998

Bücher zur Sprache Ada 95 (mittlerweile gibt es auch entsprechende Bücher zu Ada 2005):

Ein recht kompakt geschriebenes Buch stammt von Manfred Nagl; hier sind zugleich Programmierprinzipien und tiefer gehende Probleme angesprochen: Nagl., M., „Softwaretechnik mit Ada 95. Entwicklung großer Systeme.“, Vieweg-Verlag, Wiesbaden 1999

Andere Bücher (alle englisch-sprachig):

Barnes, J.G.P., „Programming in Ada 95“, 2. Auflage, Addison-Wesley 1998

Barnes, John, „Ada 95 Rationale. The Language - The Standard Libraries“, Springer-Verlag, Lecture Notes in Computer Science (Vol. 1247), 1997

English, J., "Ada 95: The Craft of Object-Oriented Programming",
<http://www.it.bton.ac.uk/staff/je/adacraft/>

Feldman, M.B. und Koffman, E.B., „Ada 95 - Problem Solving and Program Design“, 2. Auflage, Addison-Wesley 1997 (3. Auflage Textbook Binding, 1999)

Riehle, R., „Ada Distilled – ...“, <http://www.adapower.com/> (- Books & Tutorials)

Weitere Literatur:

Als Einstieg:

Appelrath, Boles, Claus, Wegener, "Starthilfe Informatik", Teubner-Verlag, Stuttgart-Leipzig, 2001

Für diverse Definitionen und Erläuterungen:

"Duden Informatik", dritte Auflage, Bibliografisches Institut, Mannheim, 2001; vierte Auflage 2006

Für Mathematik und Ideen:

Kiyek, K. und Schwarz, F., "Mathematik für Informatiker 1 und 2", Teubner-Verlag,
3. bzw. 2. Auflage, 2000 und 1999

Meinel, Christoph und Mundhenk, Martin, "Mathematische Grundlagen der Informatik", Teubner-Verlag,
Wiesbaden, 2. Auflage, 2002

Schöning, Uwe, "Ideen der Informatik", Oldenbourg-Verlag, München, 2002