

# **Informatik IV**

Vorlesung Sommersemester 2005

Professor Dr. A. Schmitt

Universität Karlsruhe (TH)

20. April 2005

# Inhaltsverzeichnis

<b>1</b>	<b>Effiziente Algorithmen</b>	<b>2</b>
1.1	Das RAM-Algorithmenmodell . . . . .	2
1.2	Asymptotisches Wachstum von Zeit und Speicher . . . . .	2
1.3	Spezifikation und Analyse von Algorithmen . . . . .	6
	1.3.1 Das Gegeben-Gesucht-Schema . . . . .	6
	1.3.2 Algorithmenanalyse . . . . .	7
1.4	Zwei allgemeine Sätze über untere Zeitschranken . . . . .	8
1.5	Anwendung von Satz A und Satz B . . . . .	16
1.6	Sortieren mit Zellrastermethode . . . . .	19
1.7	Zweidimensionale Zellraster . . . . .	24
	1.7.1 Das Bentley-Ottmann-Problem . . . . .	24
	1.7.2 Das Post-Office-Problem . . . . .	26
1.8	Die Ergebnisse von Ben-Or . . . . .	34
1.9	Ausgeglichene Bäume . . . . .	39
	1.9.1 AVL-Bäume . . . . .	39
	1.9.2 Hash-Bäume . . . . .	42
	1.9.3 B-Bäume . . . . .	47

# Kapitel 1

## Effiziente Algorithmen

### 1.1 Das RAM-Algorithmenmodell

Über unsere *random access machine* (RAM) machen wir folgende Annahmen:

1. Alle arithmetischen Operationen  $+$ ,  $-$ ,  $/$ ,  $\cdot$ ,  $<$ ,  $\leq$  erfordern eine Zeiteinheit, also Rechenzeit von  $O(1)$ , unabhängig davon, ob Gleit- oder Festpunktzahlen daran beteiligt sind.
2. Zeiger- und Indexoperationen, wie  $B := A[i, j]$ , benötigen eine Rechenzeit von  $O(1)$ , ebenso die Speicherreservierung und die Initialisierung für ein Maschinenwort. Dies gilt jedoch nicht für die z.B. in *Pascal* verfügbare Zuweisung  $A := B$  mit eventuell sehr großen Feldern (Arrays). Diese müssen bei der Rechenzeitbestimmung in Einzelwortzuweisungen aufgelöst werden.

Beim RAM-Modell spielt die Wortlänge keine Rolle. Bei theoretischen Überlegungen nehmen wir an, dass mit reellen Zahlen der Mathematik, also beliebig genau gerechnet wird.

### 1.2 Asymptotisches Wachstum von Zeit und Speicher

**Definition:**

Die Funktion  $T_A(P)$  beschreibt die Anzahl der Zeiteinheiten,<sup>1</sup>

---

<sup>1</sup> $\approx$  Elementaroperationen

die der Algorithmus  $A$  benötigt, um die Eingabedaten<sup>2</sup>  $P$  zu verarbeiten.

Der Index  $A$  kann in der Regel entfallen, falls klar ist, um welchen Algorithmus es sich im gegebenen Zusammenhang handelt. In den nächsten Abschnitten werden wir daher nur noch von  $T(P)$  sprechen.

$T_{max}(|P| = n)$  := Maximale Anzahl der Zeiteinheiten, die bei Problemen der Komplexität  $n$  nötig ist.

Es gilt demnach folgender Zusammenhang:

$$T_{max}(n) := \max_{P, |P|=n} T(P)$$

Interessant – vor allem aus praxisbezogener Sicht – ist die durchschnittliche Zeit, die der Algorithmus zur Verarbeitung benötigt:

$$T_{mittel}(n) := \text{mittel}_{P, |P|=n} T(P)$$

Der Vollständigkeit halber sei das Zeitmaß für die minimale Ausführungszeit bei einem optimalen Parametersatz angegeben:

$$T_{min}(n) := \min_{P, |P|=n} T(P)$$

Dieses Maß ist bei Komplexitätsüberlegungen in der Regel bedeutungslos, denn fast jeder Algorithmus kann so modifiziert werden, dass  $T_{min}$  sehr günstige Werte annimmt. So kann jeder Sortieralgorithmus durch einen trivialen Zusatz, in dem abgefragt wird, ob die zu sortierende Folge bereits sortiert ist, so erweitert werden, dass  $T_{min}(n) = O(n)$  gilt.

### Definition:

Für ein bestimmtes algorithmisches Problem bezeichnen wir eine Wachstumsfunktion  $f(n)$  als **untere Schranke**, wenn für jeden Algorithmus zur Lösung des Problems gilt:

---

<sup>2</sup>die Parameterliste

$$T_{max}(n) = \Omega(f(n))$$

Eine solche untere Schranke muss nicht erreichbar sein. So ist  $\Omega(1)$  eine untere Schranke für alle Algorithmen.

**Definition:**

Ein Algorithmus heißt **optimal** (bzgl. des  $T_{max}$ -Zeitverhaltens), wenn gilt:

$$T_{max}(n) = O(f(n))$$

wobei  $f(n)$  eine untere Schranke des algorithmischen Problems ist.

Da  $f(n)$  eine untere Schranke ist, gilt auch  $T_{max}(n) = \Omega(f(n))$  und damit

$$T_{max}(n) = \Theta(f(n))$$

Das Beweisverfahren für die Optimalität eines Algorithmus verläuft meist so, dass das durch den Algorithmus gelöste Problem zur Lösung eines Standardproblems benutzt wird, von dem man eine untere Zeitschranke kennt. („Problemreduktion“)

Geschieht die Transformation auf das Standardproblem in einer Weise, dass die Gesamtkomplexität von Algorithmus und Transformation die untere Schranke des Standardproblems nicht überschreitet, so kann folgendermaßen argumentiert werden:

Ein Algorithmus, der das konkrete Problem mit geringerer Zeitkomplexität lösen könnte, würde auch den notwendigen Zeitaufwand zur Lösung des Standardproblems senken. Dies ist aber aufgrund der bereits gefundenen unteren Zeitschranke nicht möglich, also kann es solch einen Algorithmus nicht geben.

Für einen Algorithmus gilt stets die Aussage:

$$T_{min}(n) \leq T_{mittel}(n) \leq T_{max}(n)$$

Auf gleiche Weise wird der Speicherbedarf eines Algorithmus beschrieben, indem man den Zeitbedarf  $T$  durch den Speicherbedarf<sup>3</sup>  $S$  ersetzt. Es gelten dann die oben genannten Zusammenhänge analog.

Wie bereits festgestellt wurde, kann man das Zeit- und Speicher- verhalten von Algorithmen in erster Näherung durch Funktionen wie  $T_{min}(n)$  oder  $S_{max}(n)$  charakterisieren. Um das Wachstum derartiger Funktionen miteinander vergleichen zu können, werden asymptotische Wachstumsklassen eingeführt. Die **Definitionen** lauten:

$$O(f(n)) := \{g(n) \mid \exists n_o \text{ und } c > 0 : \forall n \geq n_o \text{ gilt } g(n) \leq c \cdot f(n)\}, \quad \text{d.h. } g(n) \text{ wächst höchstens so stark wie } f(n)$$

$$\Omega(f(n)) := \{g(n) \mid \exists n_o \text{ und } c > 0 : \forall n \geq n_o \text{ gilt } g(n) \geq c \cdot f(n)\}, \quad \text{d.h. } g(n) \text{ wächst mindestens so stark wie } f(n)$$

$$\Theta(f(n)) := \{g(n) \mid \exists c_1, c_2, n_0 : \forall n \geq n_0 \text{ gilt } c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}, \quad \text{d.h. } g(n) \text{ wächst genau so stark wie } f(n), \text{ also } g(n) = O(f(n)) \text{ und } g(n) = \Omega(f(n))$$

In der Menge  $O(f(n))$  sind alle Funktionen enthalten, die asymptotisch höchstens so stark wie  $f(n)$  wachsen. Diese Funktionen bilden eine eigene Komplexitätsklasse. Ein Algorithmus mit einem maximalen Zeitverhalten  $T_{max}(n) = kn^2$  gehört demnach in die Komplexitätsklasse  $O(n^2)$ . Nebenbei bemerkt, gehört er natürlich auch in die Klasse  $O(n^{34})$ , da auch hier die notwendigen Konstanten gefunden werden können.

Es hat sich eingebürgert, für die Einordnung des Algorithmus in die Komplexitätsklasse  $O(n^2)$  die Schreibweise  $T_{max}(n) = O(n^2)$  zu verwenden, obwohl es eigentlich  $T_{max}(n) \in O(n^2)$  heißen müsste.

---

<sup>3</sup>engl.: *storage*

## 1.3 Spezifikation und Analyse von Algorithmen

### 1.3.1 Das Gegeben-Gesucht-Schema

Selbst bei sehr „kleinen“ Problemen ist es wichtig, genau zu sagen, worum es im Einzelnen geht. Wir werden, wenn möglich, das „Gegeben-Gesucht-Schema“ fordern.

Beispiel: *nearest neighbour problems*

#### Gegeben:

Eine feste Menge  $M$  von  $n$  Punkten im  $\mathbb{R}^2$ .

#### Gesucht:

1. Für viele Punkte  $q_i$  jeweils der nächste Nachbar (bzgl. Euklidischem Abstandsmaß) in  $M$ . (Die Anfragepunkte  $q$  sind nacheinander abzuarbeiten und sind bei der Vorverarbeitung von  $M$  noch unbekannt.)  
Bekannt als *post office problem*
2. Für einen Punkt  $q$  der nächste Nachbar in  $M$ . (Nur genau ein Anfragepunkt.)
3. Zu jedem Punkt  $p \in M$  der jeweils nächste Nachbar in  $M$ .
4. Zu jedem Punkt  $p \in M$  die jeweils  $k$  nächsten Nachbarn in  $M$ , mit einer ebenfalls fest vorgegebenen Konstanten  $k$ .
5. Zu einer gegebenen Menge  $Q \subset \mathbb{R}^2$  von Anfragepunkten  $q_1, q_2, \dots, q_k$  (simultan) der jeweils nächste Nachbar in  $M$ .

Eine etwas abgewandelte Aufgabestellung könnte lauten:

#### Gegeben:

Eine Menge  $M$  von Punkten im  $\mathbb{R}^2$ .

#### Gesucht:

Eine effiziente Datenstruktur, die es erlaubt, folgende Operationen in beliebiger Reihenfolge und Anzahl nacheinander auszuführen:

1. Erweitere  $M$  um einen neuen Punkt  $p \in \mathbb{R}^2$ .

2. Entferne aus  $M$  den Punkt  $p \in M$ .
3. Bestimme zu  $q \in \mathbb{R}^2$  die  $k$  nächsten Nachbarn in  $M$ .

Diese Formulierung stellt eine Dynamisierung von Fall 4 der ersten Problemstellung dar. Offensichtlich können fast alle graphisch-geometrischen Probleme durch Dynamisierung „erschwert“ werden.

Alle im obigen Gegeben-Gesucht-Schema spezifizierten Probleme können in die Klasse der *nearest-neighbour*-Probleme eingeordnet werden. Es sollte jedoch klar sein, dass sich in jedem einzelnen Fall sehr verschiedene Algorithmen ergeben können, auch wenn die Aufgabenstellung – oberflächlich betrachtet – sehr ähnlich klingt.

Beim Gegeben-Gesucht-Spezifikationsschema ist außerdem von großer Bedeutung, ob eine *Vorverarbeitung* zugelassen wird oder nicht. In diesem Schritt werden die gegebenen Daten in eine geeignete Datenstruktur umgewandelt, die sich dadurch auszeichnet, dass sie hilft, die Suchanfragen möglichst effizient zu beantworten. Um eine möglichst klare Aufgabenstellung zu erzielen, ist es also hilfreich, die Möglichkeit der *Vorverarbeitung* im Gegeben-Gesucht-Schema ausdrücklich aufzuführen:

**Gegeben:**

Feste Menge  $M$  von  $n$  Punkten im  $\mathbb{R}^2$ .

**Gesucht:**

Vorverarbeitung von  $M$ , sodass zu jedem Anfragepunkt  $q$  sehr schnell entschieden werden kann, welcher Punkt von  $M$  zu  $q$  den geringsten Abstand hat.

### 1.3.2 Algorithmenanalyse

Eine in unserem Sinne einigermaßen erschöpfende Algorithmenanalyse umfasst mindestens die folgenden Punkte:<sup>4</sup>

1. Suche geeignete Repräsentation von Daten-Objekten und passende interne Datenstrukturen.

---

<sup>4</sup>die Liste ist erweiterbar, vgl. Software-Engineering

2. Suche Algorithmus mit geringstem Programmieraufwand.
3. Suche Algorithmus mit asymptotisch bestem Zeitverhalten.
4. Definiere typische Problemkomplexitäten in der Praxis und daraus abgeleitete Abschätzungen für konkrete Rechenzeiten.
5. Bestimme Algorithmus mit bester Eignung für die Implementierung in professioneller Software.
6. Bestimme bzw. verbessere die numerische Stabilität.
7. Bestimme Algorithmus mit bester Parallelisierbarkeit.
8. Kombiniere Algorithmen einer bestimmten Problemklasse, um insgesamt einen besseren Algorithmus zu erhalten.

#### 1.4 Zwei allgemeine Sätze über untere Zeitschranken

Wir wollen mit unserer *random access machine* reelle Funktionen der Art  $f: W \rightarrow \mathbb{R}$  in einem Definitionsbereich  $W \subset \mathbb{R}^n$  berechnen. Da zur Berechnung von  $f(x_1, x_2, \dots, x_n) = f(X)$  nur die Operationen  $+$ ,  $-$ ,  $/$ ,  $\cdot$ ,  $>$  verwendet werden sollen, können die möglichen Programmabläufe als schleifenfreie (!) Entscheidungsbäume – siehe Abb. 1.1 – dargestellt werden:

- Eingabe:  $x_1, x_2, \dots, x_n$   
(vorbelegte Variablen)
- Konstanten:  $c_1, c_2, c_3, \dots$
- Variablen:  $A, v_1, v_2, v_3, \dots$   
( $A$  = spezielle Variable, nimmt das Funktionsergebnis auf)

Der individuelle Ablauf des Algorithmus (Baum-Abstieg) hängt nur von  $X$  und den sich jeweils ergebenden Entscheidungen bei „ $>$ “ ab.

Es gilt nun:

*Jeder im Baum vorkommende Variablenwert<sup>5</sup> ist eine rationale Funktion in  $X$ .*

---

<sup>5</sup>an einer bestimmten Stelle

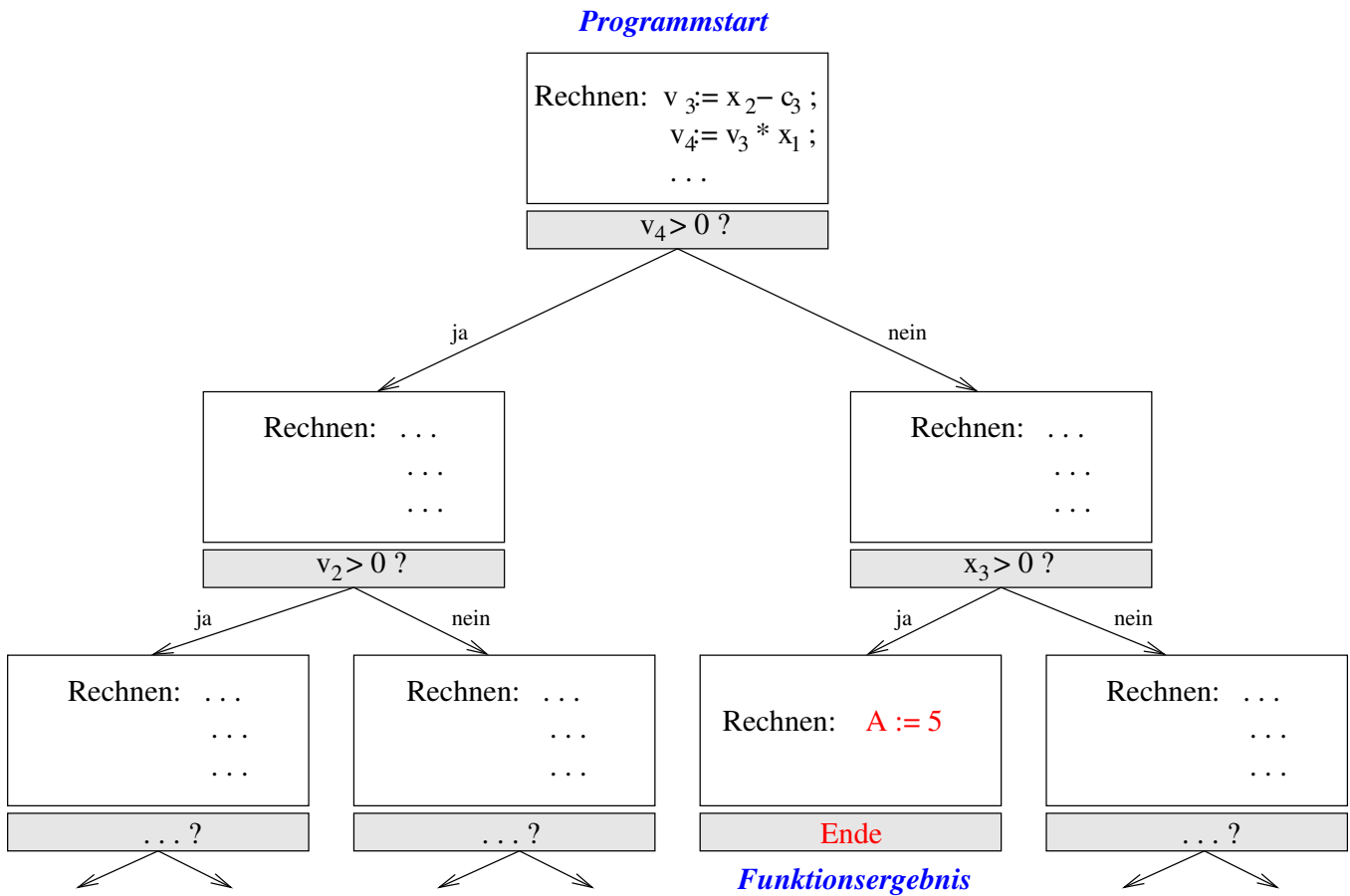


Abbildung 1.1: Entscheidungsbaum

**Beweis:**

Man verfolge, was im Einzelnen gerechnet wird, bis man an der bestimmten Stelle im Baum ankommt (siehe Abb. 1.2).

Also:

- Rationale Funktion + Rationale Funktion  $\rightarrow$  Rationale Funktion
- Rationale Funktion - Rationale Funktion  $\rightarrow$  Rationale Funktion
- Rationale Funktion  $\cdot$  Rationale Funktion  $\rightarrow$  Rationale Funktion
- Rationale Funktion / Rationale Funktion  $\rightarrow$  Rationale Funktion

Bis auf eine letzte können übrigens alle Divisionen eliminiert

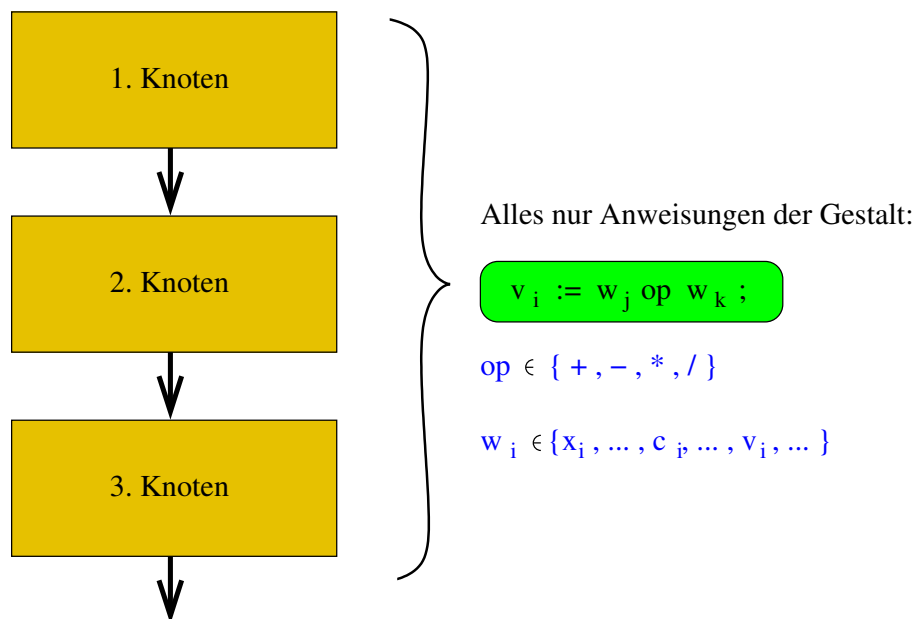


Abbildung 1.2: Rationale Funktionen

werden. Es bleiben dann Ausdrücke etwa der Form

$$\frac{8 \cdot x_1 \cdot x_3^3 + 5 \cdot x_2^2 \cdot x_7 + \dots}{x_3 \cdot x_4^2 - 3 \cdot x_5 \cdot x_8 + \dots}$$

übrig. Analoges gilt für die Subtraktion in Zähler und Nenner.

Ergebnis:

Der Algorithmus kann grundsätzlich in der folgenden Form dargestellt werden:

$$f(X) = A(X) := \begin{cases} A_1(X) & \text{falls } (B_{11}(X) \geq 0) \wedge \dots \wedge (B_{1k_1}(X) \geq 0) \\ A_2(X) & \text{falls } (B_{21}(X) \geq 0) \wedge \dots \wedge (B_{2k_2}(X) \geq 0) \\ \vdots & \\ A_m(X) & \text{falls } (B_{m1}(X) \geq 0) \wedge \dots \wedge (B_{mk_m}(X) \geq 0) \end{cases} \quad (\star)$$

Dabei sind die  $A_i$  sowie die  $B_{ij}$  rationale Funktionen in  $X$ , der Vergleich  $\geq$  bedeutet hierbei fallweise  $>$  oder  $\geq$ , denn bei einem Knoten (siehe Abb. 1.3) wird für den Ja-Weg der Term  $B(X) > 0$ ,

für den Nein-Weg der Term  $-B(X) \geq 0$  in das logische Produkt übernommen.

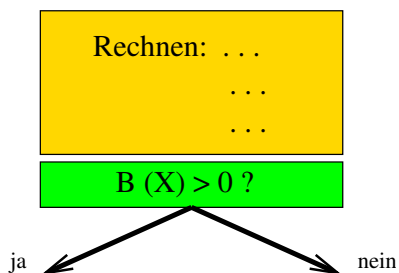


Abbildung 1.3: Ja-/Nein-Entscheidung

### Rationale Mengen, dicke und dünne

Eine Menge

$$M := \{X \in \mathbb{R}^n \mid B_1(X) \geq 0 \wedge B_2(X) \geq 0 \wedge \dots \wedge B_r(X) \geq 0\}$$

heißt **rational**, wenn alle  $B_i$  rationale Funktionen in  $X$  sind.

Also: Die Definitionsbereiche der  $A_i(X)$  in (\*) sind rationale Mengen.

Eine Menge  $U(X_0, \varepsilon) := \{X \mid |X_0 - X| < \varepsilon\}$  wird  **$\varepsilon$ -Kugel**<sup>6</sup> genannt, wenn  $\varepsilon > 0$ .

Eine Menge  $M \in \mathbb{R}^n$  heißt **dünn** oder **Dünn-Menge**, wenn es keine  $\varepsilon$ -Kugel gibt, die ganz in  $M$  enthalten ist. **Dicke Mengen** enthalten also  $\varepsilon$ -Kugeln.

#### Lemma 1:

Die Vereinigung  $M_1 \cup M_2$  zweier rationaler Dünn-Mengen ist in einer rationalen Dünn-Menge enthalten.

**Beweis:**

Sei die rationale Menge  $M$  definiert durch  $\bigwedge^i (B_i(X) \geq 0)$ .  $M$  kann keine Dünn-Menge sein, wenn es ein  $n$ -Tupel  $X_0 \in M$  gibt, so

<sup>6</sup> „ $\varepsilon$ -Umgebung“

dass  $B_i(X_0) > 0$  ist für alle  $i$ . Denn sonst könnte ein  $\varepsilon$  gefunden werden, mit  $B_i(U(X_0, \varepsilon)) > 0$ , denn rationale Funktionen sind an den Punkten, an denen sie definiert sind, stetig.

Also können wir schließen

$$M \subset \{X \mid X \in \mathbb{R}^n \wedge B_i(X) = 0 \text{ für mindestens ein } i\}$$

$$\text{Also: } X \in M \Rightarrow B_1(X) \cdot B_2(X) \cdot \dots \cdot B_r(X) = 0$$

Also gilt für die Vereinigung zweier rationaler Dünn-Mengen:

$$X \in M_1 \cup M_2 \Rightarrow B_{11}(X) \cdot \dots \cdot B_{1r_1}(X) \cdot B_{21}(X) \cdot \dots \cdot B_{2r_2}(X) = B(X) = 0$$

Das lange Produkt  $B(X)$  ist eine rationale Funktion.

Nun gilt aber:

Mengen der Gestalt  $N := \{X \mid B(X) = 0\}$  sind dünn genau dann, wenn  $B$  nicht die Nullfunktion<sup>7</sup> ist. Wenn es die Nullfunktion ist, so gilt  $N = \mathbb{R}^n$ ,  $N$  ist also dick. Wenn das Produkt  $B(X)$  die Nullfunktion ist,  $N$  also für unser Lemma viel zu groß ist, so können wir die eventuell bestehenden Nullfaktoren entfernen, die  $M_1$  und  $M_2$  definieren, weil ein  $B_{ij}(X) \geq 0$  stets wahr ist und die betreffende Menge  $M$  also im Produkt  $\bigwedge_i$  nicht eingeschränkt wird.<sup>8</sup>

Ergebnis:

Falls in  $B(X)$  eine Nullfunktion auftaucht, kann sie eliminiert werden und die Menge  $N$  ist dann eine rationale Dünn-Menge. Daher ist die Vereinigung zweier rationaler Dünn-Mengen in einer rationalen Dünn-Menge enthalten.

Anmerkungen:

Man kann Mengen  $M_1, M_2 \subset \mathbb{R}^n$  konstruieren, die dünn (aber nicht rational) sind und für die  $M_1 \cup M_2 = \mathbb{R}^n$  gilt.<sup>9</sup> Auch die Vereinigung endlich vieler  $M_1, M_2, \dots, M_r$ , die rational und dünn sind, ist in einer rationalen Dünn-Menge enthalten. Im allgemeinen

<sup>7</sup>Beispiel für die Nullfunktion:  $X_1 - X_1 = B(X)$

<sup>8</sup>wäre statt „ $\geq$ “ die Relation „ $>$ “ anzuwenden, so wäre  $M$  leer

<sup>9</sup>Können Sie solche Mengen konstruieren?

ist  $M_1 \cup M_2$  keine rationale Menge.<sup>10</sup>

Beispiel:

Rationale Dünn-Mengen kann man leicht konstruieren. Zum Beispiel:

$$M_1 := \{(x, y) \in \mathbb{R}^2 \mid x = y\}$$
$$M_2 := \{(x, y) \in \mathbb{R}^2 \mid x = -y\}$$

$M_1 \cup M_2$  ist eine rationale Dünn-Menge (Abb. 1.4).

$$\{(x, y) \mid x^2 - y^2 \geq 0 \wedge y^2 - x^2 \geq 0\} \Rightarrow |x| = |y|$$

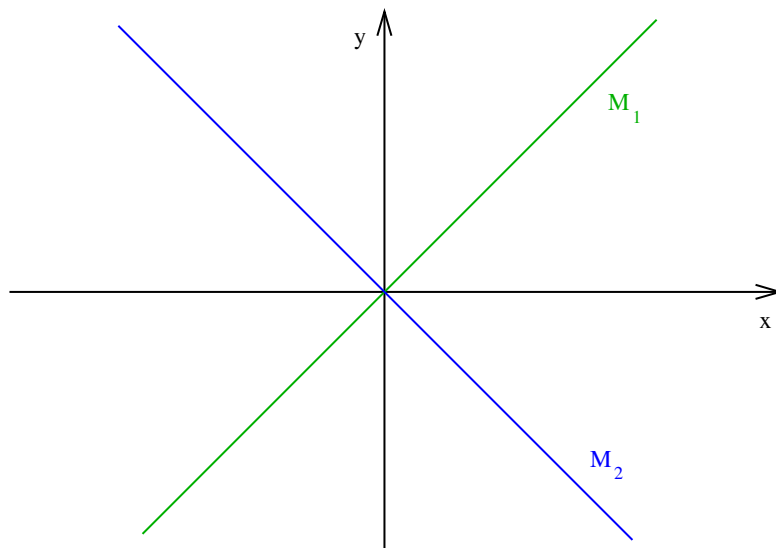


Abbildung 1.4: Rationale Dünn-Mengen

**Lemma 2:**

Sei  $U(X_0, \varepsilon)$  eine  $\varepsilon$ -Kugel und seien  $F_1(X), F_2(X)$  zwei rationale Funktionen mit der Eigenschaft  $F_1(X) = F_2(X)$  für alle  $x \in U(X_0, \varepsilon)$ .

Dann gilt:  $F_1(X) = F_2(X)$  für  $X \in \mathbb{R}^n$ .

<sup>10</sup>Dies wird vom Dozenten vermutet, weiß er jedoch nicht sicher.

**Beweis:**

Die Funktion  $R(X) := F_1(X) - F_2(X)$  ist in  $U(X_0, \varepsilon)$  die Nullfunktion. Also ist das Zählerpolynom  $z(X)$  von  $R(X) = z(X)/n(X)$  das Nullpolynom in  $U(X_0, \varepsilon)$ . Wenn aber ein Polynom in einer ganzen  $\varepsilon$ -Umgebung verschwindet, kann es nur das Nullpolynom sein.

**Satz A:** (*lower-bound*-Theorem für RAM-Algorithmen)

Sei  $f(X)$  mit  $X \in \mathbb{R}^n$  eine reellwertige Funktion mit Definitionsbereich  $W \subset \mathbb{R}^n$  und sei  $A(X)$  im Sinne von (\*) ein korrekter Algorithmus, der  $f$  im RAM-Modell<sup>11</sup> berechnet. Seien ferner bekannt:

- $q$  paarweise verschiedene Punkte  $X_1, X_2, \dots, X_q \in W$
- $\varepsilon > 0$
- $q$  paarweise verschiedene rationale Funktionen  $Q_1, Q_2, \dots, Q_q$  mit der Eigenschaft  $Q_i(X) = f(X)$  für alle  $X \in U(X_i, \varepsilon) \subset W$

Dann gilt, dass im schlechtesten Fall die Zahl  $R$  der Vergleiche „>“ zur Berechnung von  $A(X)$  der Ungleichung

$$R \geq \log_2 q$$

genügt.

**Beweis:**

Sei  $A(X)$  eine korrekte Implementierung zur Berechnung der Funktion  $f$ , also

$$f(X) = A(X) := \begin{cases} A_1(X) & \text{falls } X \in M_1 \\ A_2(X) & \text{falls } X \in M_2 \\ \vdots & \\ A_m(X) & \text{falls } X \in M_m \end{cases}$$

Wir behaupten nun, dass zu jeder der rationalen Funktionen  $Q_i$  (aus dem Satz) mindestens ein  $A_j$  existieren muss, mit

$$Q_i(X) = A_j(X)$$

Dazu prüfen wir für  $j = 1, 2, \dots, m$  die Mengen  $U(X_i, \varepsilon) \cap M_j$ , ob

<sup>11</sup>d.h. mit reellen Zahlen und den Operationen  $+, -, \cdot, /, >$

sie dick sind. Da  $U(X_i, \varepsilon)$  durch die Gleichung

$$\varepsilon^2 - (x_{i1} - x_1)^2 - (x_{i2} - x_2)^2 - \dots - (x_{in} - x_n)^2 > 0$$

definiert werden kann, sind die  $\varepsilon$ -Kugeln rational und dick.

Der Schnitt rationaler Mengen ist stets eine rationale Menge (trivial!). Da  $A$  eine korrekte Berechnung für  $f$  ist, gilt:

$$U(X_i, \varepsilon) = (U(X_i, \varepsilon) \cap M_1) \cup (U(X_i, \varepsilon) \cap M_2) \cup \dots \cup (U(X_i, \varepsilon) \cap M_m)$$

Da die linke Menge dick ist, muss auch mindestens eine der Mengen rechts dick sein, also bspw.  $(U(X_i, \varepsilon) \cap M_j)$ , denn die endliche Vereinigung dünner rationaler Mengen bleibt dünn. Wegen Lemma 2 muss dann

$$Q_i = A_j$$

gelten, denn sie sind in mindestens einer  $\varepsilon$ -Kugel identisch. Daraus folgt  $m \geq q$ .

Ein binärer Entscheidungsbaum mit mindestens  $q$  Blättern hat mindestens die Gesamthöhe

$$\log_2 q$$

und dies ist eine untere Schranke für die Zahl der Vergleichsoperationen „>“ im schlechtesten Fall.

Bevor wir auf Anwendungen zu sprechen kommen, noch ein zweiter Satz:

**Satz B:**

Die Voraussetzungen sind wie in Satz A.

Wenn auf dem Definitionsbereich  $W$  von  $f$  eine Wahrscheinlichkeitsverteilung gegeben ist in dem Sinne, dass das Wahrscheinlichkeitsmaß der Mengen  $S_i := \{X \mid f(X) = Q_i(X) \wedge f \text{ ist in Punkt } X \text{ lokal gleich } Q_i\}$  mit  $p_i \geq 0$  angegeben werden kann, so ist der Erwartungswert  $E$  für die Zahl der Vergleichsoperationen „>“ begrenzt durch

$$E \geq \sum_{i=1}^q p_i \log_2 \frac{1}{p_i}$$

wobei vorausgesetzt werden muss, dass dünne rationale Mengen stets das Wahrscheinlichkeitsmaß 0 haben.

**Beweis:**

Um den Erwartungswert  $E$  zu berechnen, nehmen wir zunächst an, dass es jeweils nur ein Blatt des Baumes gibt, das die Menge  $S_i$  überdeckt. Sei  $m_i$  die Pfadlänge des Blatts, von der Wurzel aus gerechnet. Dann gilt

$$E \geq \sum_{i=1}^q p_i \cdot m_i \geq \sum_{i=1}^q p_i \log_2 \left( \frac{1}{p_i} \right)$$

Der zweite Teil der Ungleichung ergibt sich aus einem Satz der Informationstheorie<sup>12</sup> über optimale Codes im ungestörten Fall, für deren mittlere Codewortlänge<sup>13</sup>  $\bar{l}$  bei gegebenen Codewort-Wahrscheinlichkeiten  $p_i$  gilt:

$$\bar{l} \geq \sum_{i=1}^q p_i \log_2 \frac{1}{p_i}$$

Wenn nun die einzelnen Mengen  $S_i$  über mehrere Blätter verteilt sind, werden die  $p_i$  aufgespalten, was den Wert von  $E$  weiter erhöht.

Also: Wir kennen die diskreten Eingaben  $X_1, X_2, \dots$ . In ihren Umgebungen gilt  $f(X) = Q_i(X)$ , die  $Q_i(X)$  sind paarweise verschiedene rationale Funktionen. Daraus folgt: mindestens  $O(\log_2 n)$  Vergleichsoperationen im schlechtesten Fall.

## 1.5 Anwendung von Satz A und Satz B

### 1. Beispiel (untere Schranken für Sortieren)

Berechne

$$f(x_1, x_2, \dots, x_n) := x_1^{j_1} + x_2^{j_2} + \dots + x_n^{j_n}$$

wobei die  $j_k$  der Sortierindex von  $x_k$  ist, nachdem  $\{x_1, x_2, \dots, x_n\}$  sortiert wurde<sup>14</sup>.

Sei z.B. für  $n = 5$

---

<sup>12</sup>vgl. späteres Kapitel der Vorlesung, dann wird einiges klarer

<sup>13</sup>Binär-Alphabet

<sup>14</sup>Der Sortierindex ist eine eindeutige Funktion  $(x_1, x_2, \dots, x_n) \rightarrow (j_1, j_2, \dots, j_n) \in \mathbb{N}^n$ , wenn gleiche  $x$ -Werte beim Sortieren ihre relative Position zueinander beibehalten, also nicht vertauscht werden.

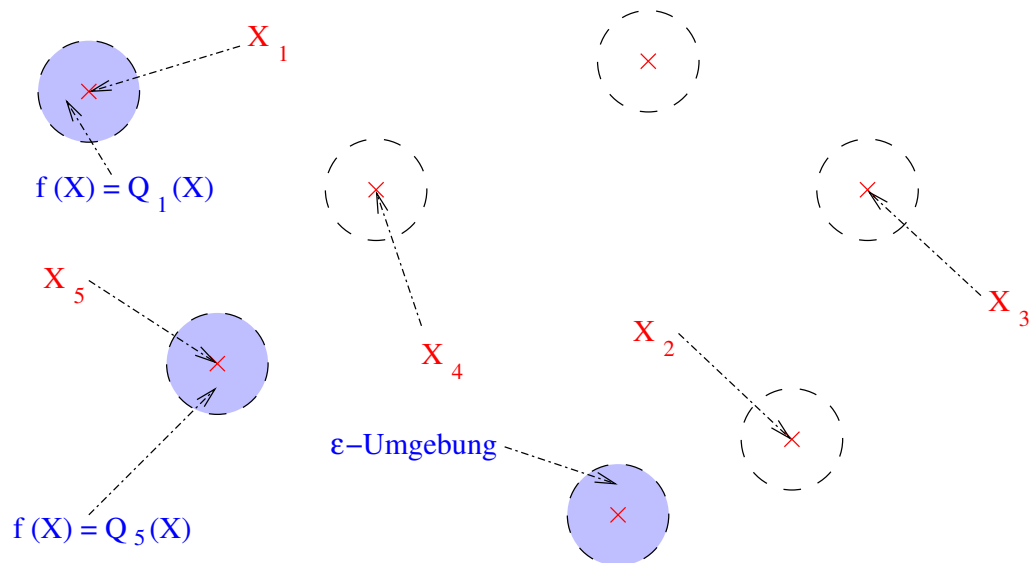


Abbildung 1.5: Veranschaulichung von Satz A und Satz B:

$$f(1, 3, 2, 6, 3) := 1^1 + 3^3 + 2^2 + 6^5 + 3^4 = 1^1 + 2^2 + 3^3 + 3^4 + 6^5$$

Anwendung von Satz A:

$$q := n!$$

$X_i := (i_1, i_2, \dots, i_n) =$  Permutationen der Zahlen  $1, 2, \dots, n$  (Indexfolge)

$$Q_i := x_1^{i_1} + x_2^{i_2} + \dots + x_n^{i_n}$$

$\varepsilon = 0, 4$  ist okay, weil  $|X_i - X_j| \geq 2$  für  $i \neq j$

Ergebnis:

Im schlechtesten Fall werden mindestens

$$\log_2(n!) = O(n \log n)$$

Vergleichsoperationen benötigt.<sup>15</sup>

Das gilt auch für das Sortieren:

1. Sortiere zuerst die  $x_1, \dots, x_n$ , d.h.: ermittle  $i_1, \dots, i_n$
2. In jedem Blattknoten sind die Zahlen  $i_1, i_2, \dots, i_n$  bekannt, also berechne  $x_1^{i_1} + x_2^{i_2} + \dots + x_m^{i_m}$ .

---

<sup>15</sup>vgl. Übungsblatt 1

### Reduktionsbeweis:

Der obige Algorithmus (Schritte 1. und 2.) kann nicht mit weniger Vergleichen als die Berechnung von  $f$  auskommen!

### Satz:

Das Sortieren auf unserer RAM hat eine untere Schranke von  $T_{max}(n) = \Omega(n \log n)$  Vergleichsoperationen. Auch massiver und kostenloser Einsatz von arithmetischen Operationen  $+$ ,  $-$ ,  $\cdot$ ,  $/$  kann diese untere Schranke nicht absenken.

*Jeder Sortieralgorithmus, der nur mit Vergleichsoperatoren und evtl. Arithmetik arbeitet, ist optimal, wenn  $T_{max}(n) = O(n \log n)$  gilt.*

Wenn alle Sortierpermutationen gleich wahrscheinlich sind, gilt auch  $T_{mittel} = \Omega(n \log n)$ .

### 2. Beispiel (Suchen durch Halbieren<sup>16</sup>)

Es seien  $n - 1$  Intervalle  $[a_i, a_{i+1})$  fest vorgegeben, also  $a_1 < a_2 < \dots < a_n$ . Wir wollen die Funktion

$$f(X) := i, \text{ falls } a_i \leq x < a_{i+1}$$

berechnen.

Anwendung von Satz A:

$$q := n - 1$$

$$X_i := (a_i + a_{i+1})/2 \text{ f\"ur } i = 1, \dots, q$$

$$\varepsilon := \min_i \{(a_{i+1} - a_i)/3\} \text{ ist okay, } Q_i(x) = i \text{ (konstante Funktionen)}$$

### Satz:

Das Halbierungsverfahren (binäre Suche, „Fächersortieren“) hat eine untere Schranke von  $T_{max}(n) = \Omega(\log n)$  Vergleichsoperationen und ist damit optimal.

### 3. Beispiel (Rundungsoperation)

Wir wollen die Funktion

$$f(x) = \text{floor}(x) := \text{größter ganzzahliger Wert } \leq x$$

im Wertebereich  $x \in [0, n) = W$  berechnen. Also:

$$q := n$$

$$X_i := i - \frac{1}{2}$$

---

<sup>16</sup> $O(\log n)$

$$Q_i(x) := i$$

$$\varepsilon := 0,3$$

**Satz:**

Die *floor*-Funktion im Intervall  $[0, n)$  hat eine untere Schranke von  $T_{max}(n) = \Omega(\log_2 n)$  Vergleichsoperationen.

**Konsequenzen:**

Rundungsoperationen haben eine wesentlich höhere „Rechenkraft“ als normale Vergleiche. Auch bei ihrer Berechnung nützen arithmetische Operationen nichts, denn man berechnet sie auf der RAM mit Halbierungsverfahren. Auf unserer heutigen<sup>17</sup> Hardware ist *floor* eine sehr schnelle Operation, was man ausnutzen kann.

**Nachtrag:**

Wie kann Satz A verallgemeinert werden für den Fall, dass Funktionen

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

berechnet werden sollen?<sup>18</sup>

## 1.6 Sortieren mit Zellrastermethode

Auch bekannt als Bucket-Sort<sup>19</sup> oder Dobosiewicz-Sort (DBS)

**Gegeben** :  $x_1, x_2, \dots, x_n \in \mathbb{R}$

**Gesucht** :  $x_{i_1} \leq x_{i_2} \leq \dots \leq x_{i_n}$  (sortierte Folge der  $x_i$ )

Verbesserte Variante von DBS:

1. Bestimme  $x_{min}, x_{max} \in \{x_1, x_2, \dots, x_n\}$ .
2. Initialisiere ein äquidistantes<sup>20</sup> Zellraster für das Intervall  $[x_{min}, x_{max}]$  mit  $n$  Zellen. Jede Zelle nimmt die  $x$ -Werte auf, die in ihr liegen.<sup>21</sup>

---

<sup>17</sup>2005 n.u.Z.

<sup>18</sup>hier keine Antwort ...

<sup>19</sup>„Eimersortieren“

<sup>20</sup>gleich weit voneinander entfernt, gleiche Abstände aufweisend

<sup>21</sup>mehrere Werte in einer Zelle möglich

3. Der Index  $i$  der Zelle, in die ein Wert  $x$  abzulegen ist, wird wie folgt berechnet:

$$i(x) := 1 + \text{floor} \left( n \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} \right)$$

4. Die sortierte Folge wird bestimmt, indem man nacheinander die Zellen  $1, 2, \dots$  abarbeitet und die jeweils dort abgelegten  $x$ -Werte mit einem Sortierverfahren in  $O(m \log m)$  intern sortiert.<sup>22</sup>

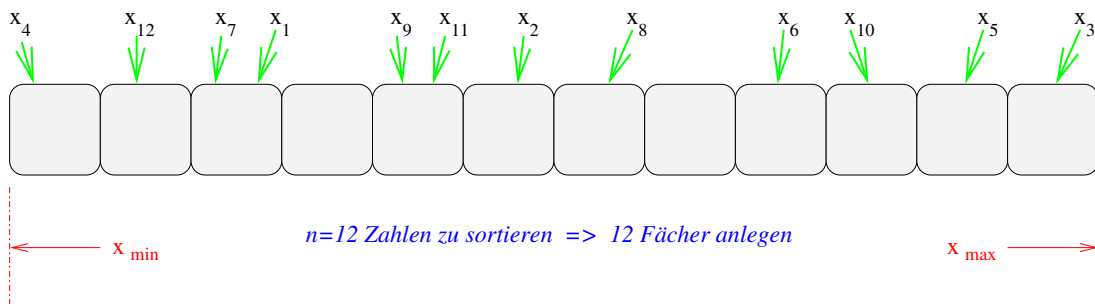


Abbildung 1.6: Dobosiewicz-Sort

**Satz:**

Das modifizierte Dobosiewicz-Sort hat die Zeitschranke

$$T_{\max}(n) = O(n \log n)$$

und ist damit optimal; und zwar auch dann, wenn die Rundungsoperation  $\text{floor}$  mit einem Aufwand der Ordnung  $O(\log n)$  eingerechnet wird.

**Beweis:** O-Kalkül

1. Schritt:  $O(n)$
2. Schritt:  $O(n)$
3. Schritt:  $n \cdot O(\log n)$  ( $n$  Aufrufe von  $\text{floor}$ )
4. Schritt:

<sup>22</sup>Im Original-DBS werden die Zellen der Reihe nach mit Bubble-Sort bearbeitet, bei uns mit  $O(m \log m)$ -Verfahren;  $m$  ist hier die Anzahl von Elementen in *einer* Zelle

$$\begin{aligned}
O(\sum_{k=1}^n i_k \cdot \log_2 i_k) &\leq O(n \log_2 n) && \text{wegen } \sum i_k = n \\
&\leq n \log_2 \underbrace{(\max i_k)}_{\leq n} \\
&\leq n \log_2 n
\end{aligned}$$

(auch die Zahlen mit  $i_k = 0$  oder 1 können höchstens  $O(n)$  beitragen ...)

Dobosiewicz<sup>23</sup> hat aber anders gerechnet: Er hat einen *floor*-Aufruf ebenfalls mit  $O(1)$  bewertet und hat angenommen, dass die  $x_i$  gleichverteilt sind, z.B. im Intervall  $[0, 1)$ . Und dann hat er noch die Zellen mit einem  $O(n^2)$ -Verfahren sortiert!

**Gegeben:**

$n$  gleich wahrscheinliche Zellen<sup>24</sup> für  $n$  Elemente

**Gesucht:**

Wahrscheinlichkeit, dass in einer Zelle  $i$  Elemente liegen (Abb. 1.7).

$$w(i) = \binom{n}{i} p^i (1-p)^{n-i}$$

Um eine Zelle mit  $i$  Elementen zu sortieren, setzen wir  $i(i-1)$  Vergleichsoperationen an. Zellen mit weniger als zwei Elementen interessieren uns nicht, denn sie können insgesamt nur  $O(n)$  zum Aufwand beitragen.

$$\text{Sortieraufwand} = \sum_{i=2}^n i(i-1) \binom{n}{i} p^i (1-p)^{n-i}$$

Es gilt

$$i(i-1) \binom{n}{i} = n(i-1) \binom{n-1}{i-1} = n(n-1) \binom{n-2}{i-2}, \quad (\text{für } i \geq 2)$$

Also ist der

$$\text{Sortieraufwand} = \sum_{i=2}^n n(n-1) \binom{n-2}{i-2} p^i (1-p)^{n-1} \quad (\text{substituiere } j := i-2)$$

<sup>23</sup>Włodzimierz Dobosiewicz: „Sorting by Distributive Partitioning“ Inform. Processing Letters (7) 1978

<sup>24</sup>Fächer, Eimer (*buckets*) etc.

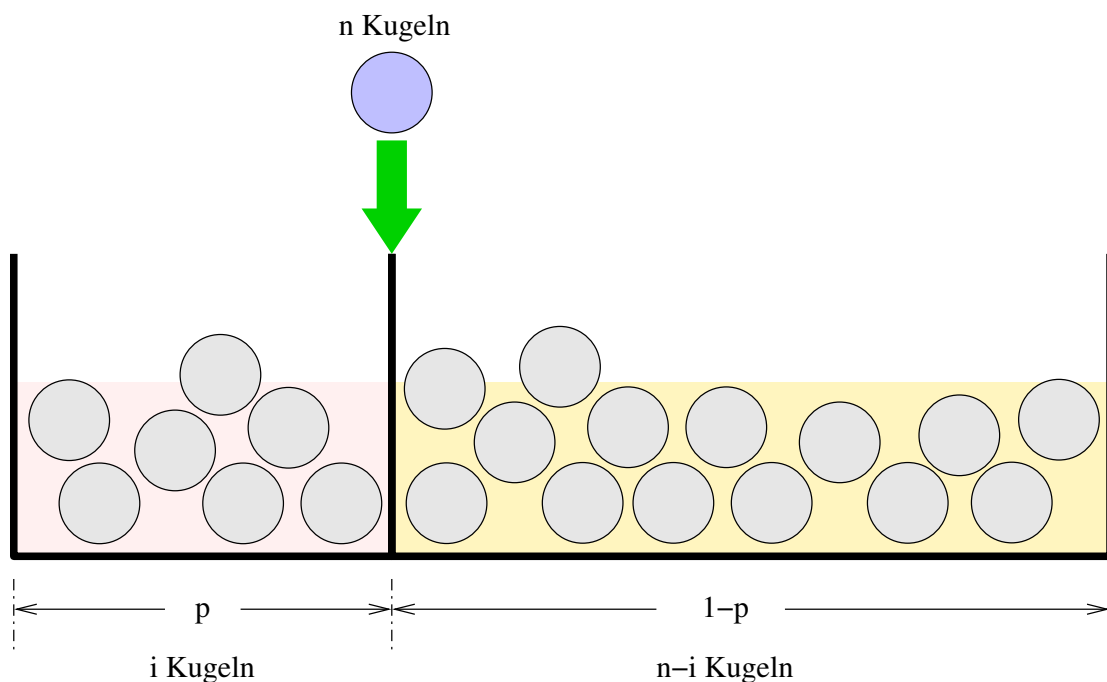


Abbildung 1.7: Bernoulli-Verteilung

$$= n(n-1) \underbrace{\sum_{j=0}^{n-2} \binom{n-2}{j} p^{j+2} (1-p)^{n-2-j}}_{=p^2(p+(1-p))^{n-2}=p^2}$$

Jetzt setzen wir  $p = 1/n$  (bei  $n$  gleich wahrscheinlichen Fächern) und erhalten den

$$\begin{aligned} \text{Sortieraufwand} &= n \cdot (n-1) \cdot p^2 \\ &= \frac{n \cdot (n-1)}{n \cdot n} \\ &= \frac{n-1}{n} \\ &\leq 1 \end{aligned}$$

**Satz:**

Selbst wenn die Zellen bei DBS mit einem  $O(n^2)$ -Verfahren sortiert werden, ergibt sich bei Gleichverteilung der zu sortierenden Elemente

$$T_{\text{mittel}}(n) = O(n)$$

Hier spart also die Rundung den Faktor  $\log n$ .

**Beweis:** siehe oben

Der Satz kann noch verschärft werden, wenn man Ungleichverteilungen betrachtet (Abb. 1.8).

Modell:

$$p_i := \frac{i^k}{N(k, n)} = \text{Wahrscheinlichkeitsmaß von Fach } i$$

wobei

$$N(k, n) := \underbrace{\sum_{i=1}^n i^k}_{\leq n \cdot n^k} = O(n^{k+1})$$

$$\begin{aligned} \text{Sortieraufwand}_k &= n(n-1) \sum_{i=1}^n \left( \frac{i^k}{N(k, n)} \right)^2 \\ &= n(n-1) \frac{1}{N(k, n)^2} \sum_{i=1}^n i^{2k} \\ &= O(n^2) \cdot \frac{1}{O(n^{2k+2})} \cdot O(n^{2k+1}) \\ &= O(n) \end{aligned}$$

**Satz:**

Dobosiewicz-Sort (DBS) arbeitet in

$$T_{\text{mittel}}(n) = O(n)$$

auch dann, wenn die zu sortierenden Elemente diskret polynomial verteilt sind.

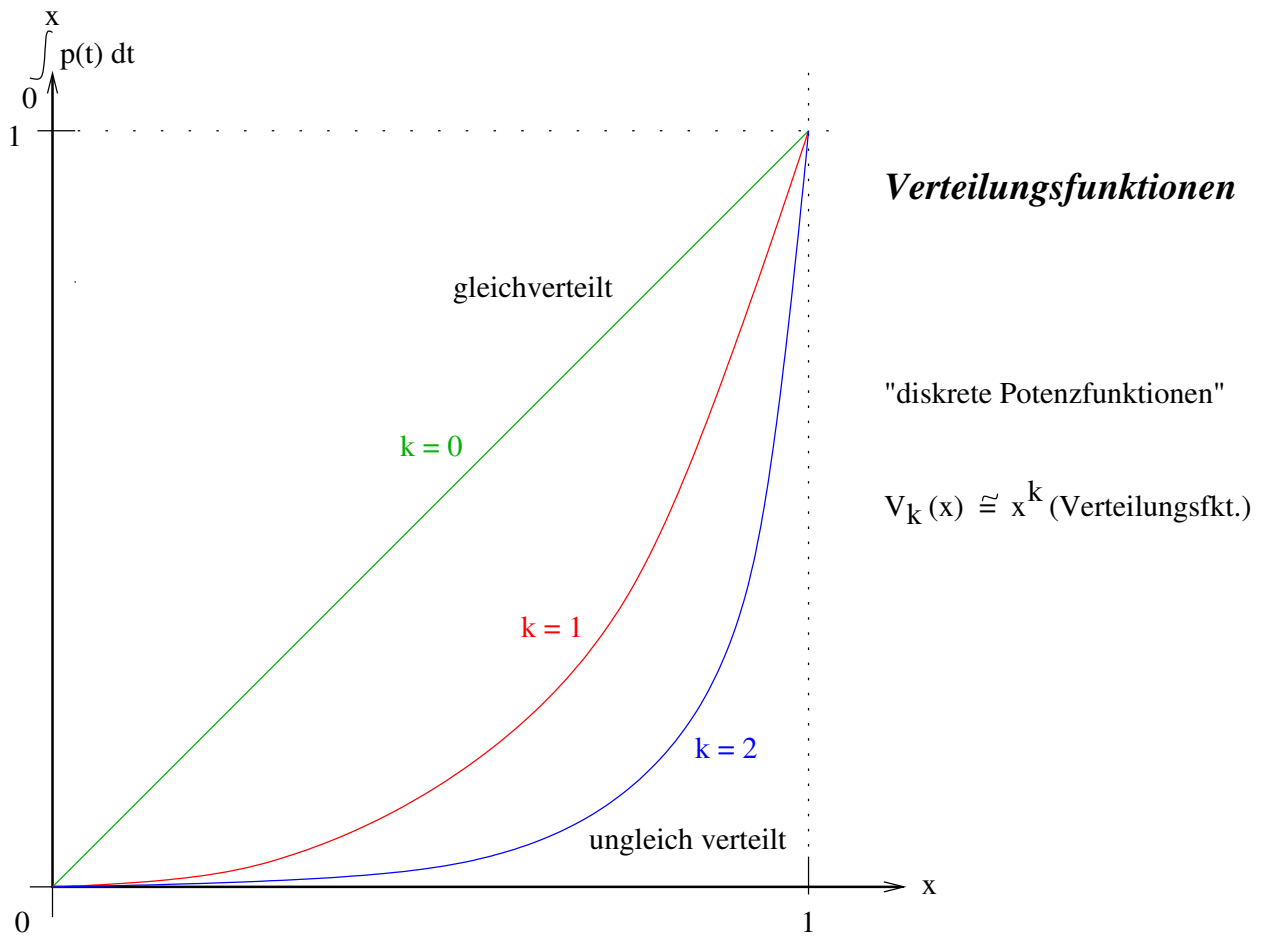


Abbildung 1.8: Ungleich-Verteilung

## 1.7 Zweidimensionale Zellraster

### 1.7.1 Das Bentley-Ottmann-Problem

Wir betrachten das Bentley-Ottmann-Problem.<sup>25</sup>

**Gegeben:**

$n$  Strecken  $S_i = (P_{i1}, P_{i2})$  mit  $P_{ij} \in \mathbb{R}^2$

**Gesucht:**

<sup>25</sup>J. Bentley, T. Ottmann: „Algorithms for Reporting and Counting Geometric Intersections“, IEEE Tr. Computers, 1979 (in Karlsruhe entstanden)

Sämtliche Paare  $(i, j)$ , sodass sich  $s_i$  und  $s_j$  schneiden oder berühren (Abb. 1.9).

$n = 16$  Strecken  $\Rightarrow$  16 Fächer einrichten:

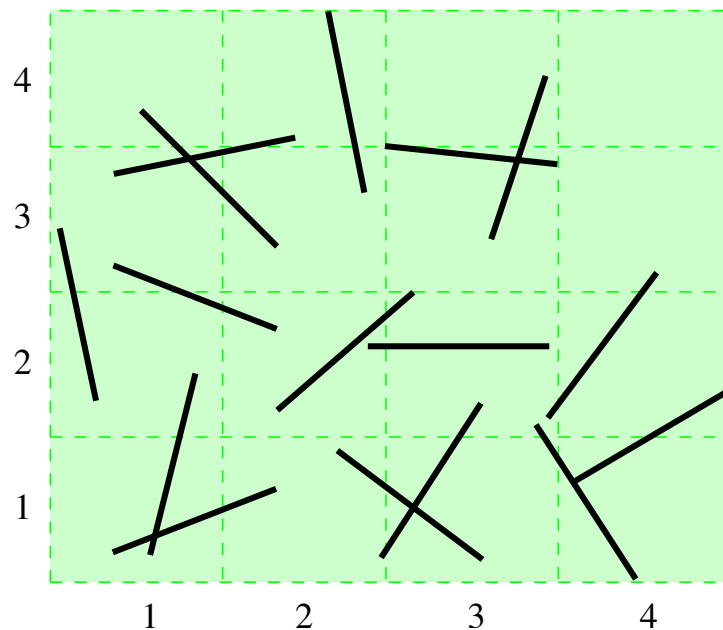


Abbildung 1.9: das Bentley-Ottmann-Problem

$$T_{max}(n, k) = O((n + k) \log n), \quad k = \text{Schnittpunktzahl}$$

Methode: „Plane Sweep“, „Scan Line“<sup>26</sup>

Algorithmus ist nicht optimal.

Zellrastermethode:

1.  $\lfloor \sqrt{n} \rfloor \cdot \lfloor \sqrt{n} \rfloor = O(n)$  Zellen bei  $n$  Objekten<sup>27</sup>
2. Für alle  $i$ : Trage  $S_i$  in alle die Zellen ein, die von  $S_i$  berührt werden (Datenstruktur)
3. Für alle Zellen: Prüfe die in der Zelle eingetragenen  $S_i$  paarweise auf Schnittpunkte bzw. Berührungspunkte, berichte diese.

---

<sup>26</sup>vgl. Info II  
<sup>27</sup>warum  $\sqrt{n}$ ?

Komplexitätsabschätzung:  
Wir benötigen eine Art Gleichverteilung.

Definition: Für eine gegebene Menge  $M := \{S_i \mid i = 1, \dots, n\}$  von Strecken

$MZK(M) :=$  maximale Zahl von Strecken, die bei Schritt 2. in eine Zelle eingetragen werden

**Satz:**

Für die Menge aller 2-D-Szenen  $M$  mit  $MZK(M) \leq k_0 = \text{const.}$  gilt: Der Zellrasteralgorithmus bestimmt sämtliche Schnittpunkte in  $T_{\max}(n) = O(n)$ . Für die Zahl  $k$  der gefundenen Schnittpunkte gilt:  $k(n) = O(n)$ .

**Beweis:** O-Kalkül

1. Schritt:  $O(n)$
2. Schritt:  $O(n \cdot k_0) \Rightarrow O(n)$
3. Schritt:  $O(n \cdot k_0^2) \Rightarrow O(n) \leftarrow$  Verbesserung möglich?

Außerdem kann es höchstens  $n \cdot k_0^2$  Schnittpunkte geben.<sup>28</sup>

Wie kann man „Entgleisungen“<sup>29</sup> des Zellraster-Algorithmus (wie in Abb. 1.10) abfedern?

### 1.7.2 Das Post-Office-Problem

Wir betrachten das Post-Office-Problem („POP“):

**Gegeben:**

$n$  Punkte  $M := \{P_i \in \mathbb{R}^2 \mid i = 1, \dots, n\}$

**Gesucht:**

Vorverarbeitung der Punkte  $M$ , sodass für eine Folge von Anfragepunkten  $Q_1, Q_2, \dots \in \mathbb{R}^2$  der jeweils nächste Nachbar in  $M$  bestimmt werden kann.<sup>30</sup>

<sup>28</sup> $k_0$  rastergünstig  $\Leftrightarrow MZK(M) \leq k_0$

<sup>29</sup>zu viel Rechenzeit, Rastermethode ungünstig

<sup>30</sup>Euklidischer Abstand in der Ebene

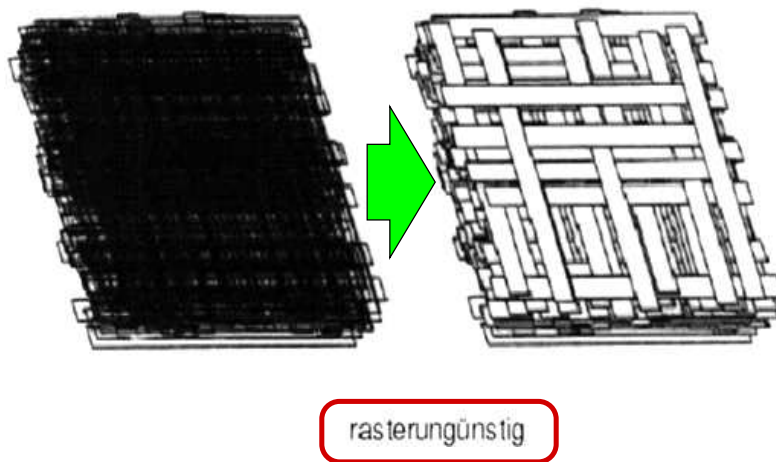
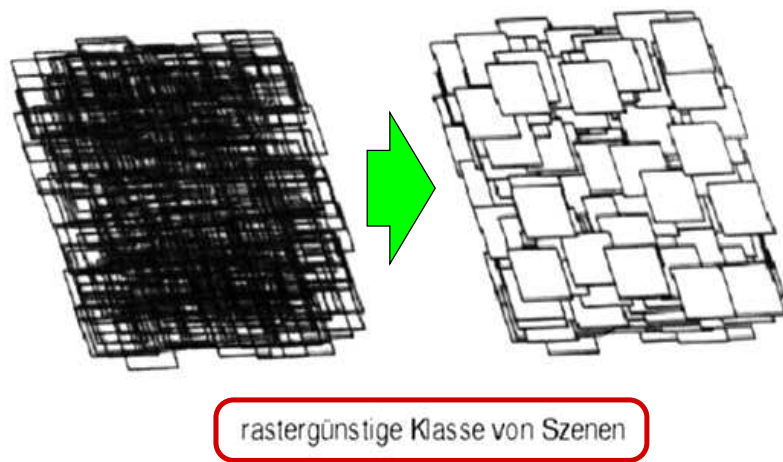


Abbildung 1.10: rastergünstige/-ungünstige Szenen

Bekannt ist:

Optimale Lösung erreicht man, indem man das Voronoi-Diagramm von  $M$  bestimmt<sup>31</sup> und ein optimales Verfahren für die Punktlokalisierung<sup>32</sup> anwendet.

**Satz:**

Die Zeitschranke  $T_{max}(n) = O(\log n)$  ist optimal.

---

<sup>31</sup> $T_{max}(n) = O(n \log n)$  möglich

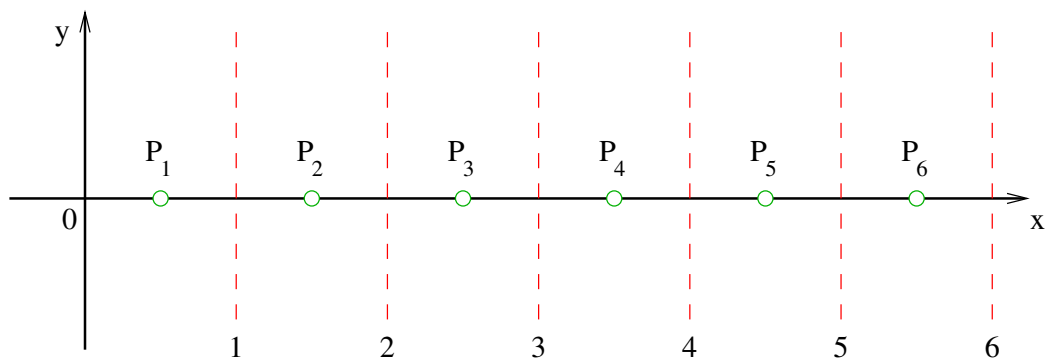
<sup>32</sup> $T_{max}(n) = O(\log n)$

**Beweis:**

„Missbrauche“ die Punktlokalisierung, um  $\text{floor}(x)$  für  $x \in [0, n)$  zu berechnen (klassischer Reduktionsbeweis).

Um  $\text{floor}(x)$  zu berechnen, wählen wir folgende Punkte für das Post-Office-Problem, vgl. Abb. 1.7.2:

$$P_i := (i - 0,5 \quad 0) \quad i = 1, \dots, n$$



Um nun  $\text{floor}(x)$  zu berechnen, lösen wir das POP für  $Q := (x \quad 0)$ . Wenn  $P_i$  der Antwortpunkt ist, so gilt  $\text{floor}(x) = i - 1$ . Kommen zwei Antworten  $P_i, P_{i+1}$ , so ist  $\text{floor}(x) = i$ . Der Zusatzaufwand ist  $O(1)$ .

**Ergebnis:**

Wenn POP-Anfragen schneller als  $T_{max}(n) = O(\log n)$  wären, so könnten wir auch  $\text{floor}$  schneller als  $O(\log n)$  berechnen. Widerspruch!

Voronoi-Diagramme (VD, Abb. 1.11) sind aus den Voronoi-Gebieten (VG) der einzelnen  $P_i$  zusammengesetzt.

Voronoi-Gebiet des Punktes  $P_i$ :

$$VG(P_i) := \{X \in \mathbb{R}^2 \mid \forall k : |X - P_i| \leq |X - P_k|\}$$

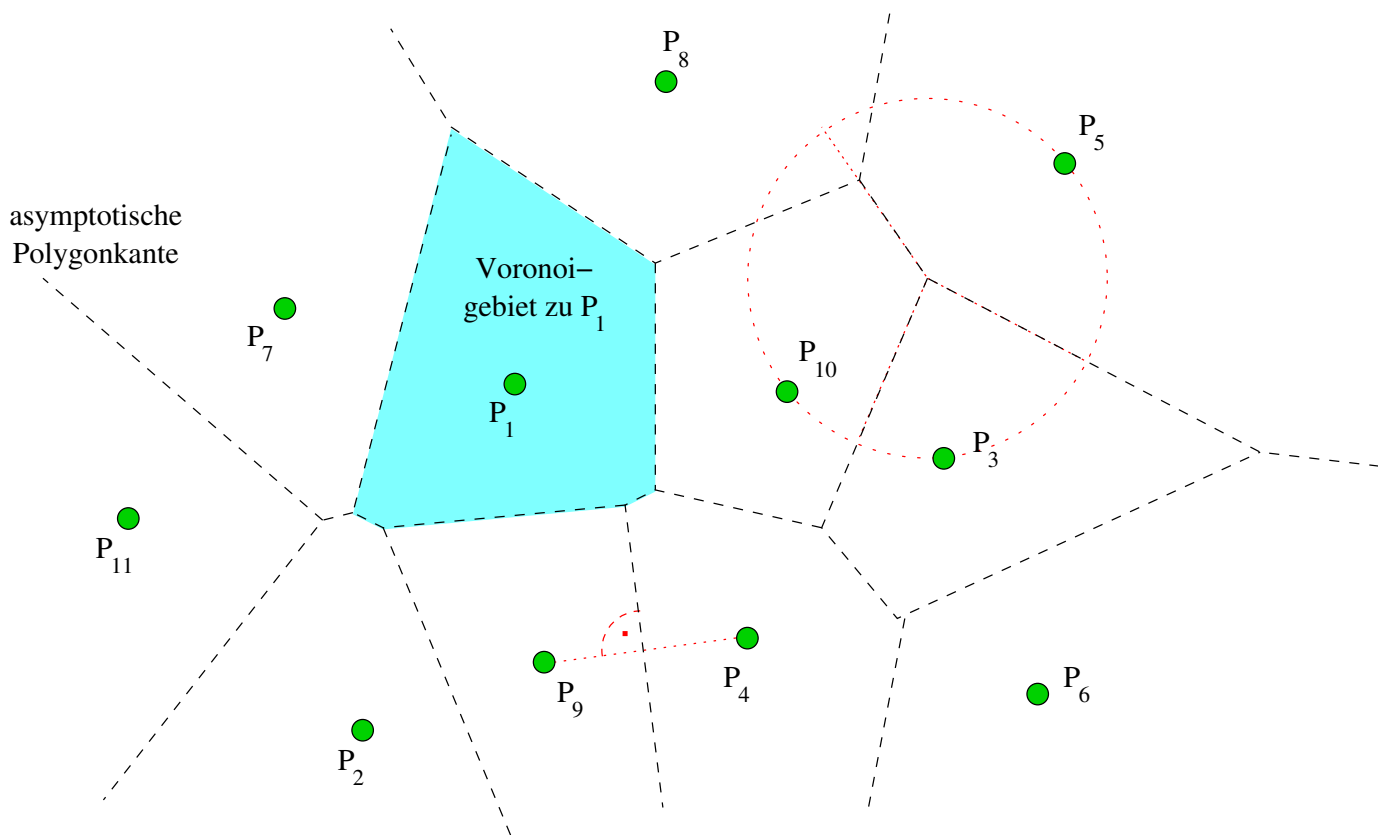


Abbildung 1.11: Voronoi-Diagramm

$VG(P_i) =$  Menge aller Punkte  $X \in \mathbb{R}^2$ , die näher (im Sinne von  $\leq$ ) bei  $P_i$  als bei irgendeinem  $P_k$  liegen.

Datenstruktur für Voronoi-Diagramme:

1.  $VG(P_i)$  ist ein konvexes Polygon, das durch  $P_i$  und die Polygonkantenfolge  $K_{i1}, K_{i2}, \dots, K_{im_i}$  im Uhrzeigersinn definiert ist.
2. Das Voronoi-Diagramm besteht aus allen  $VG(P_i)$ .
3. Jede Kante  $K_{ij}$  trägt einen Verweis auf die gleiche Kante des entsprechenden Nachbargebietes.

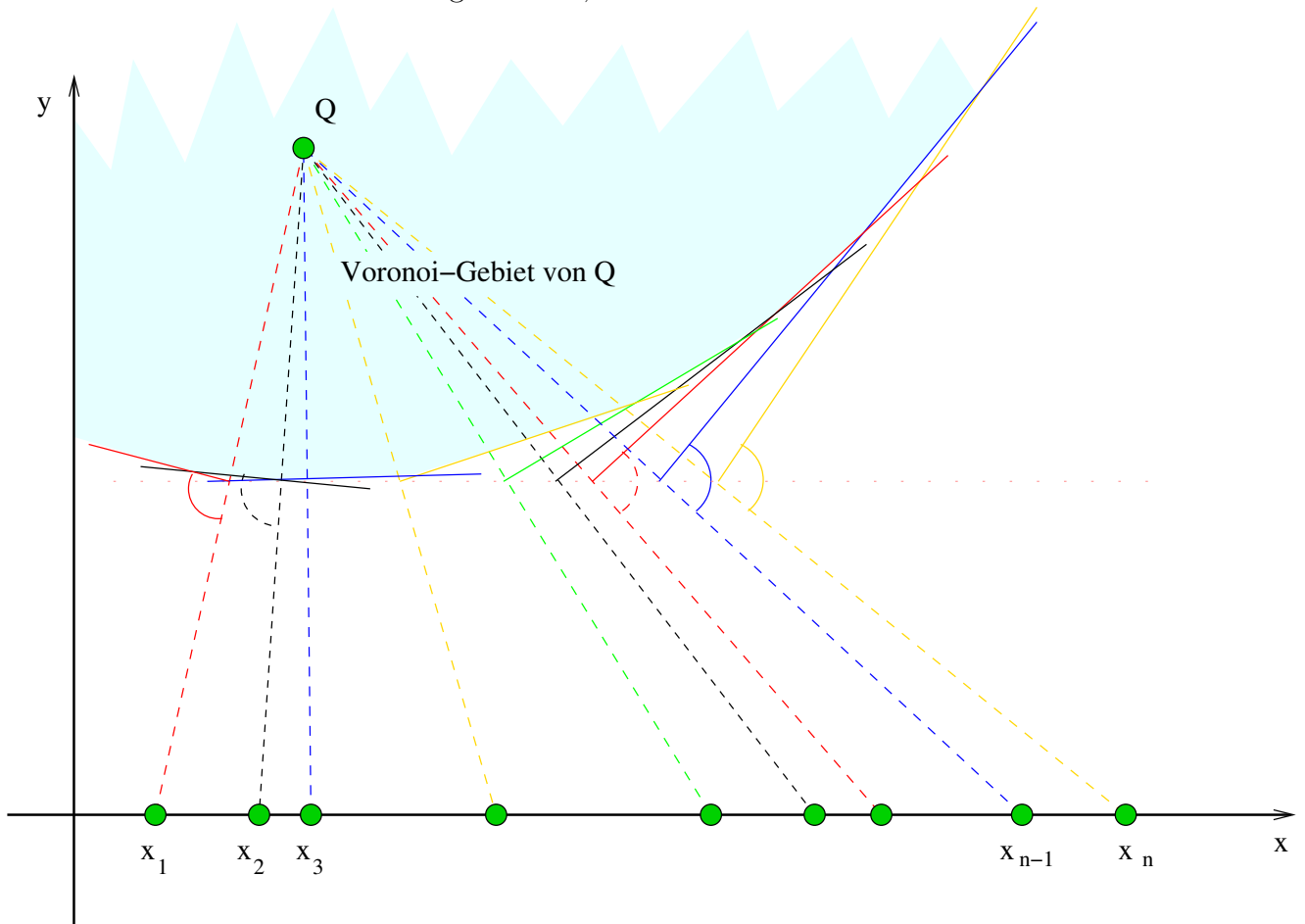
Man kann so Voronoidiagramme effizient durchlaufen und absuchen.

**Satz:**

Ein Algorithmus, der einzelne Voronoi-Gebiete berechnen kann, ist optimal, wenn  $T_{max} = O(n \log n)$ . Hierbei ist  $n$  die Anzahl der Punkte.

**Beweis:**

Missbrauche den Algorithmus, um Zahlen zu sortieren:



Wir bauen das Voronoi-Gebiet von  $Q$  auf, indem wir die  $x_i$  für  $i = 1, 2, \dots$  jeweils neu hinzufügen.

**Behauptung:**

Jeder rechts hinzukommende Wert  $x_n$  definiert eine neue Asym-

ptote an  $VG(Q)$  und jeder frühere Punkt  $x_i$  ( $i < n$ ) ist an  $VG(Q)$  beteiligt.

**Beweis** durch Induktion:

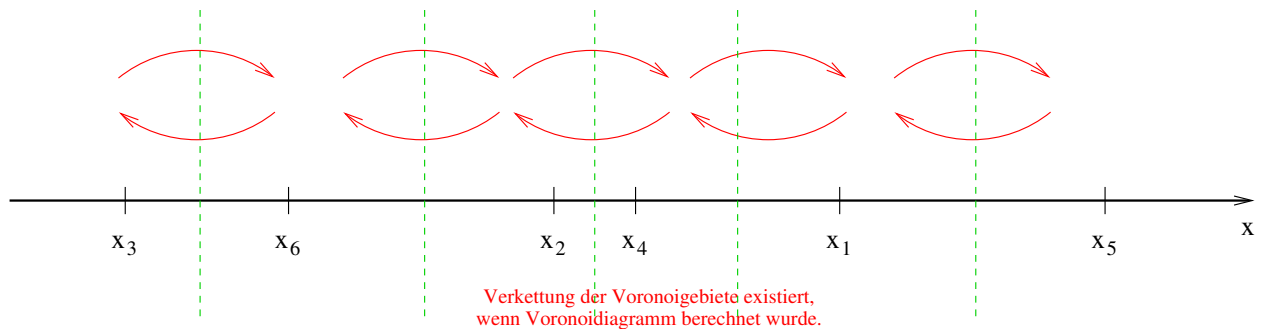
Induktionsanfang,  $n = 2$ : trivial.

Induktionsschritt: Sei die Behauptung bis  $n - 1$  bewiesen (Ind.voraus.); Fall  $n$ :

Neue Asymptote schneidet alte Asymptote, ein Stück von der alten Asymptote bleibt stehen.

Also:  $VG(Q)$  besteht aus einem Kantenzug, in dem jedes  $x_i$  in sortierter Reihenfolge durch eine Kante repräsentiert ist<sup>33</sup>.

Der Satz gilt natürlich auch für die Bestimmung des ganzen Voronoidiagramms. Dafür gibt es noch ein extrem elegantes **Be-  
weisverfahren**:



Problem:

Kann die Bestimmung des nächsten Nachbarn beim POP mit Zellraster-Verfahren beschleunigt werden?

Annahmen:

Punkte  $P_i$  in Einheitsquadrat gleichverteilt, ebenso die Anfragepunkte  $Q$ .

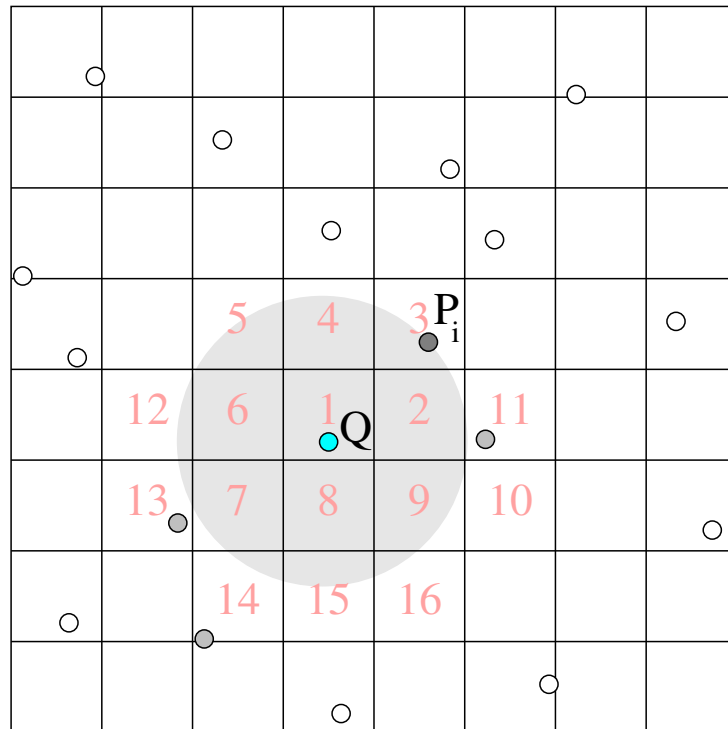
Vorgehen (Algorithmus):

1. Vorverarbeitung:  $\sqrt{n} \cdot \sqrt{n}$ -Zellraster, trage die  $P_i$  in die Zellen ein.<sup>34</sup>

<sup>33</sup>Grenzkurve ist eine Parabel mit Scheitelpunkt  $(x_0 \quad y_0/2)$

<sup>34</sup>floor-Aufrufe

2. Gehe mit Anfragepunkt  $Q$  in die Zelle, in der  $Q$  liegt.<sup>35</sup>
3. Suche mit Spiralmethode, von  $Q$  ausgehend, einen ersten Punkt (vgl. Abb.). Dieser ist Kandidat für nächsten Nachbarn  $P_i$ .
4. Suche Kreisscheibe mit Mittelpunkt  $Q$  und Radius  $|P_i - Q|$  nach eventuell näher liegenden Punkten ab.



**Satz:**

Der Zellersteralgorithmus findet den nächsten Nachbarn von beliebigen Anfragepunkten in

$$T_{\text{mittel}}(n) = O(1)$$

**Beweis:**

In J. L. Bentley, B. W. Weide, A. C. Yao: „Optimal Expected-Time Algorithms for Closest Point Problems.“ ACM Tr. Mathematical Software (1980), 563-580.

---

<sup>35</sup> floor-Aufrufe

Eng verwandte Aufgabe: (vgl. Schritt 3.)

Wie viele Zellen muss man im Mittel durchsuchen, um einen ersten Punkt  $P$  zu finden?

**Gegeben:**

$n$  Zellen,  $n$  Punkte  $P_i$  zufällig und gleichverteilt in den Zellen abgelegt.

**Gesucht:**

$w_i$  = Wahrscheinlichkeit, dass Zelle  $i$  mindestens einen Punkt enthält und die Zellen  $1, 2, \dots, i - 1$  keinen Punkt

*Back to the Roots der Wahrscheinlichkeitstheorie:*

Alle möglichen Füll-Vorgänge der  $n$  Fächer mit  $n$  Kugeln als Indexfolge aufschreiben:

$\underbrace{\quad}_{i_1}, \quad \underbrace{\quad}_{i_2}, \quad \dots, \quad \underbrace{\quad}_{i_n} \in \{1, \dots, n\}$   
 1. gefülltes Fach    2. gefülltes Fach    letztes gefülltes Fach

(Man beachte, dass theoretisch alle Kugeln z.B. in Fach Nr. 7 landen können mit einer Wahrscheinlichkeit  $> 0!$  Indexfolge dann  $\underbrace{7, 7, \dots, 7}_{n\text{-mal}}$ )

Es gibt insgesamt  $n^n$  verschiedene Indexfolgen = Anzahl der möglichen Fälle.

Wenn die Fächer  $1, 2, \dots, i - 1$  keine Kugel enthalten sollen, so fallen die Indizes  $1, 2, \dots, i - 1$  ganz aus, übrig bleiben

	$(n - i + 1)^n$	Indexfolgen ohne Vorkommen von $1, 2, \dots, i - 1$
außerdem	$(n - i)^n$	Indexfolgen ohne Vorkommen von $1, 2, \dots, i - 1, i$
d.h.:	$(n - i + 1)^n - (n - i)^n$	Indexfolgen ohne $1, 2, \dots, i - 1$ , aber mit mindestens einem $i$

Also ist die gesuchte Wahrscheinlichkeit

$$w_i = \frac{(n - i + 1)^n - (n - i)^n}{n^n} = \frac{\# \text{ günstige Fälle}}{\# \text{ mögliche Fälle}}$$

Möglicherweise auch mit geschickter Anwendung von Bernoulli oder

ähnlichen Formeln zu rechnen, aber dabei werden Abhängigkeiten zwischen den Füllungen berücksichtigt.

Wie viele Zellen muss man durchsuchen, um einen Punkt zu finden? Erwartungswert  $E$ :

$$\begin{aligned} E &= \sum_{i=1}^n i \cdot w_i \\ &= \frac{1}{n^n} \sum_{i=1}^n i \cdot (n-i+1)^n - i \cdot (n-i)^n \\ &= 1 \cdot n^n - 1 \cdot (n-1)^n + 2 \cdot (n-1)^n - 2 \cdot (n-2)^n + 3 \cdot \dots + n \cdot 1^n - 0 \\ &= n^n + (n-1)^n + (n-2)^n + \dots + 1^n \end{aligned}$$

Also ist

$$E = \frac{1}{n^n} \sum_{i=1}^n i^n$$

der Erwartungswert für die Zahl der zu durchsuchenden Fächer.

Ob  $\lim_{n \rightarrow \infty} E$  durch Konstante beschränkt oder von  $n$  abhängig ist, wird in den Übungen geklärt.

## 1.8 Die Ergebnisse von Ben-Or

M. Ben-Or: „Lower Bounds For Algebraic Computation Trees“<sup>36</sup> on Proc. 15th ACM Annual Symp. on Theory of Computation, 80-86, 1983.

Gleiche RAM wie unsere, versucht aber<sup>37</sup>, auch die Operationen  $+$ ,  $-$ ,  $\cdot$ ,  $/$  mitzuzählen.

---

<sup>36</sup>Berühmtes Papier

<sup>37</sup>unter Einsatz tiefliegender Theoreme über algebraische Mannigfaltigkeiten ( $\approx$  rationale Funktionen)

## E1: Elementeindeutigkeit

Gegeben:

$$x_1, x_2, \dots, x_n \in \mathbb{R}$$

Gesucht:

Antwort auf die Frage, ob mindestens zwei der  $x_i$  gleich sind.

Wenn nur die Vergleiche gezählt werden, so gilt für den Test auf Elementeindeutigkeit<sup>38</sup> (EE)

$$EE(x_1, \dots, x_n) = \left( \prod_{i \neq j} (x_i - x_j) \stackrel{!}{=} 0 \right) \in \text{BOOLEAN}$$

also mit Satz A untere Schranke  $T_{max}(n) = \Omega(1)$ .

**Satz:** (Ben-Or)

Jeder RAM-Algorithmus für das Problem der Elementeindeutigkeit hat die Laufzeit  $T_{max}(n) \geq \Omega(n \log n)$  wobei arithmetische Operationen mit  $O(1)$  mitgerechnet werden, Vergleiche sowieso.

Anwendungen:

Das EE-Problem kann in vielen Fällen dazu benutzt werden, untere Schranken für andere Probleme zu beweisen.

### a) Bentley-Ottmann-Problem

Das BOP hat eine untere Schranke<sup>39</sup> von  $T_{max}(n) = \Omega(k + n \log n)$

**Beweis:**

„Missbrauche“ einen Algorithmus zur Lösung des BOP zur Lösung des EE-Problems:

**Gegeben**  $(x_1, x_2, \dots, x_n) \in \mathbb{R}$ . **Gesucht** Gibt es  $i \neq j$  mit  $x_i = x_j$ ?

Konstruiere<sup>40</sup> die Strecken-Menge<sup>41</sup>  $M$ :

$$S_1 = ((x_1, 0), (x_1, 0)), S_2 = ((x_2, 0)(x_2, 0)), \dots, S_n = ((x_n, 0)(x_n, 0))$$

(degenerierte Strecken)

<sup>38</sup>auch: Elementverschiedenheit (*element distinctness*)

<sup>39</sup>Hinweis: Multivariable Zeitschranken sind in vielen Bereichen notwendig, um genauere Aussagen über asymptotisches Zeitverhalten zu machen, Nimmt man z.B.  $k =$  Zahl der gefundenen Schnittpunkte (beim BOP) nicht mit, so ist  $T_{max}(n) = \Omega(n^2)$ .

<sup>40</sup>in linearer Zeit  $O(n)$

<sup>41</sup>als Datensatz

Dies ist ein korrekter Eingabe-Datensatz für jeden BOP-Löser. Wende also einen beliebigen Algorithmus zur Lösung des BOP an.

Ergebnis:

Falls mindestens ein Schnitt- oder Berührungspunkt  $(i, j)$  gefunden wird, gilt  $x_i = x_j$ . Also: korrekte Entscheidung über das EE-Problem.

Final-Argumentation für diesen Reduktionsbeweis<sup>42</sup>:

Würde es einen Algorithmus für das BOP geben, dessen untere Schranke für das Auffinden des ersten Schnittpunktes besser als  $\Omega(n \log n)$  wäre, so würde obige Konstruktion auch das EE-Problem besser als in  $T_{max}(n) = \Omega(n \log n)$  lösen. Widerspruch.

Finesse:

BOP-Algorithmus wird beim Auffinden des ersten Schnittpunktes angehalten, also  $k = 1$ . Eine korrekte untere Schranke für das BOP sollte auch den Reporting-Aufwand<sup>43</sup> enthalten, also  $\Omega(k)$ . Daher

$$T_{max}(n, k) = \Omega(k + n \log n)$$

Es gibt (komplizierte) Algorithmen, die diese untere Schranke erreichen. Man kennt also einen optimalen Algorithmus zur Lösung des BOP<sup>44</sup>.

Warum konnten wir für gutartige (=rastergünstige) Szenen das BOP in  $T_{max}(n, k) = O(n)$  lösen?

## b) Rechteck-Schnittproblem

*rectangle intersection problem* (RIP)

**Gegeben:**

$n$  achsenparallele Rechtecke  $R_i$  in der Ebene  $\mathbb{R}^2$ .

**Gesucht:**

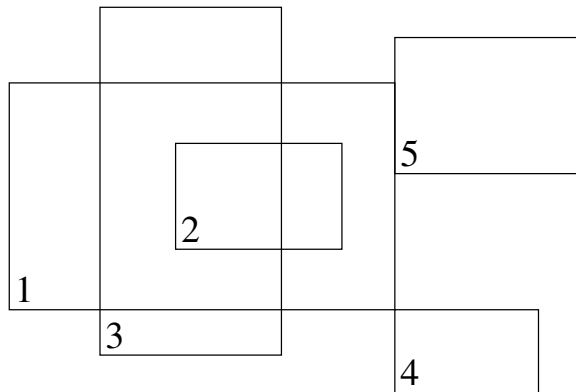
Alle Paare  $(i, j)$  mit  $i \neq j$ , sodass  $R_i \cap R_j \neq \emptyset$ , bezogen auf die Punktmengen.

---

<sup>42</sup>unbedingt erforderlich z.B. in Klausur, da Beweis sonst noch nicht fertig

<sup>43</sup>Komplexität der Ausgabedaten

<sup>44</sup>Zur Erinnerung: die erste nichttriviale Lösung des BOP von Bentley + Ottmann hatte  $T_{max}(n, k) = O((n + k) \log n)$ .



Behauptung:

Das RIP hat eine untere Schranke von  $T_{max}(n, k) = \Omega(k + n \log n)$

**Beweis:**

Analog zum BOP; löse das EE-Problem mit einem beliebigen Algorithmus für das RIP. Also:  $(x_1, \dots, x_n) \rightarrow$  degenerierte Rechtecke

$$\left( \underbrace{(x_1, 0)}_{\text{Punkt links unten}}, \underbrace{(x_1, 0)}_{\text{Punkt rechts oben}}, \dots, (x_n, 0), (x_n, 0) \right)$$

Punkt links unten Punkt rechts oben

Optimale Algorithmen für das RIP sind bekannt, nicht ganz so kompliziert wie für das BOP.

## E2: Mengenthaltensein

**Gegeben:**

Zwei endliche Mengen  $M_1, M_2 \subset \mathbb{R}$

**Gesucht:**

Antwort auf die Fragen

- $M_1 = M_2?$
- $M_1 \subset M_2?$
- $M_1 \cap M_2 = \emptyset?$

Ben-Or: Alle drei Probleme haben eine untere Schranke von  $T_{max}(n) = \Omega(n \log n)$ .

- Hilft kostenlose Verwendung arithmetischer Operationen?

- Wie konstruiert man optimale Algorithmen?
- Können lineare Zellraster eingesetzt werden?

### E3: Das Maß-Problem

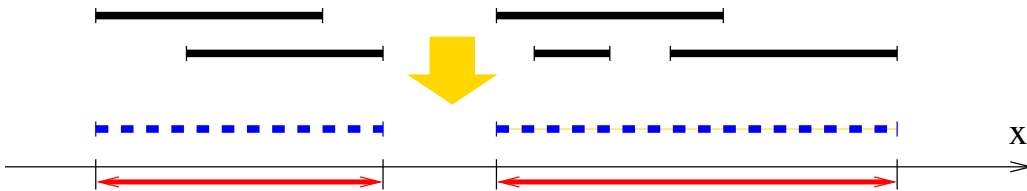
Gegeben:

Eine Menge  $M := \{[a_i, b_i] \mid a_i, b_i \in \mathbb{R}, a_i \leq b_i, i = 1, \dots, n\}$  von Intervallen auf  $\mathbb{R}$ .

Gesucht:

Das Maß von  $M$  bzw. von  $I := \bigcup_i [a_i, b_i]$ :

$$|I| := \int_{x \in I} 1 \cdot dx$$



Das Maß-Problem hat eine untere Schranke von

$$T_{max}(n) = \Omega(n \log n)$$

Ein optimaler Algorithmus ist leicht zu konstruieren: Intervalle sortieren, vereinigen, messen.

Folgerung:

Bereits die Intervallvereinigung  $\bigcup_i [a_i, b_i]$ , die eine disjunkte Intervallmenge liefert, hat ebenfalls die untere Schranke von  $\Omega(n \log n)$ .

Beweis:

?

### E4: Konvexe Hülle

Gegeben:

$n$  Punkte  $P_i \in \mathbb{R}^2$

Gesucht:

Hat die konvexe Hülle der  $P_i$  genau  $n$  Ecken? (Problem ist schärfer als die einfach zu beweisende untere Schranke von  $\Omega(n \log n)$  für das Erstellen der konvexen Hülle)

Das Problem zur Berechnung der Eckenzahl der konvexen Hülle hat die untere Schranke  $T_{max}(n) = \Omega(n \log n)$

## E5: Interpolationspolynome berechnen

Gegeben:

$n$  Stützstellen  $(x_i, y_i) \in \mathbb{R}^2$

Gesucht:

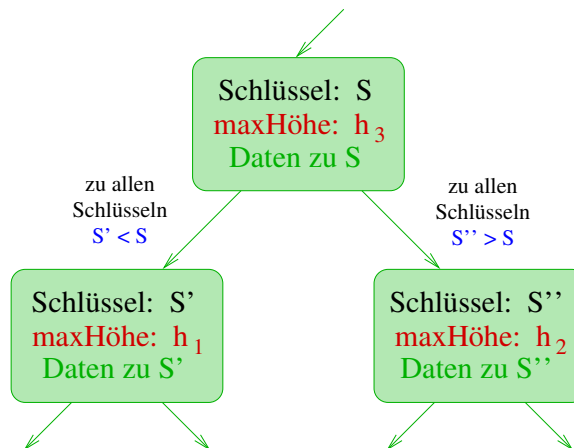
Polynom  $P(x)$  mit  $P(x_i) = y_i$  für alle  $i = 1 \dots n$

Untere Schranke:<sup>45</sup>  $T_{max}(n) = \Omega(n \log n)$

## 1.9 Ausgeglichene Bäume

### 1.9.1 AVL-Bäume

„Ausgeglichenheit“ bei  $n$  Elementen im Baum heißt Tiefe =  $O(\log n)$ .  
1962 von Adelson-Velskii und Landis in Russland erfunden.



<sup>45</sup>Hat etwas mit FFT = *fast Fourier transform* zu tun

AVL-Bedingung:

$$|h_1 - h_2| \leq 1 \quad h_3 = 1 + \max\{h_1, h_2\}$$

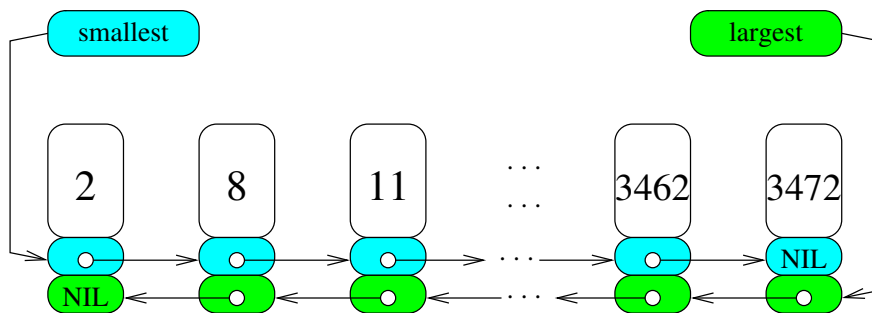
**Unser Ziel:** AVL-Bäume so erweitern, daß möglichst viele Anfrage-Operationen effizient erledigt werden können.

Welche Operationen sind mit gewöhnlichen AVL-Bäumen effizient ausführbar?

Operation		Zeitbedarf
<code>create(B)</code>	leeren Baum initialisieren	$O(1)$
<code>empty(B) : boolean</code>	Abfrage, ob Baum leer	$O(1)$
<code>insert(B,S,Daten)</code>	einfügen	$O(\log n)$
<code>delete(B, S)</code>	entfernen	$O(\log n)$
<code>member(B,S) : boolean</code>	Abfrage	$O(\log n)$
<code>rechtsnachbar(B,S)</code>	rechter Nachbar	$O(\log n)$
<code>linksnachbar(B,S)</code>	linker Nachbar	$O(\log n)$
<code>swap(B,S,S')</code>	vertauschen von zwei Nachbarknoten	$O(1)$
<code>reportup(B,S,k)</code>	Liest, mit <b>S</b> startend, die nächsten <b>k</b> Datensätze in aufsteigender Ordnung	$O(k + \log n) ???$
<code>reportdown(B,S,k)</code>	Liest, mit <b>S</b> startend, die nächsten <b>k</b> Datensätze in absteigender Ordnung	$O(k + \log n) ???$
<code>minimum(B)</code>	Minimum aller Elemente im Baum	$O(1) ???$
<code>maximum(B)</code>	Maximum aller Elemente im Baum	$O(1) ???$
<code>reportall(B)</code>	Alle Elemente in aufsteigender Ordnung ausgeben	$O(n) ???$
<code>AVLbaum(Var, B Seq)</code>	Aufbau eines AVL-Baumes aus vorsortierten Elementen	$O(n)$

Können die Knoten eines AVL-Baumes linear verknüpft werden, um sequentielle Durchläufe wesentlich zu beschleunigen?

Ja: Doppel-Verzeigerung der Größe nach:



Erfordert 2 Zeigervariablen zusätzlich pro Knoten.

Was könnte bei der Implementierung Schwierigkeiten machen?

1. *balance*-Prozedur?<sup>46</sup>
2. *delete*-Prozedur
  - (a) Zuerst das zu entfernende Element aus der linearen Verkettung herausnehmen.
  - (b) Rest wie üblich.
3. *insert*-Prozedur?
  - (a) Zeiger auf linken und rechten Nachbarn des einzufügenden Elementes bestimmen.
  - (b) Einfügen, wie üblich.
  - (c) Knotenverkettung (insges. 4 Zeigervariablen) herstellen

Konsequenzen:

- **delete:**  
Aufruf von `member`, dann einen Schritt links oder rechts, findet den eventuell erforderlichen Tauschkandidaten
- **rechtsnachbar, linksnachbar:**  
trivial mit `member`
- **report:** in allen Varianten trivial
- **swap:** einfach
- **minimum, maximum:** trivial

---

<sup>46</sup>Nein. (Warum?)

- **reportall: ohne Baumdurchlaufen**

Fazit:

AVL-Bäume mit linearer Verkettung erlauben sowohl schlüsselorientiertes als auch sequentielles Arbeiten in Datenmengen. Die einzelnen Operationen erfordern höchstens einen Aufwand von

$$T_{max}(n, k) = O(k + \log n)$$

( $k$  = Reporting-Aufwand)

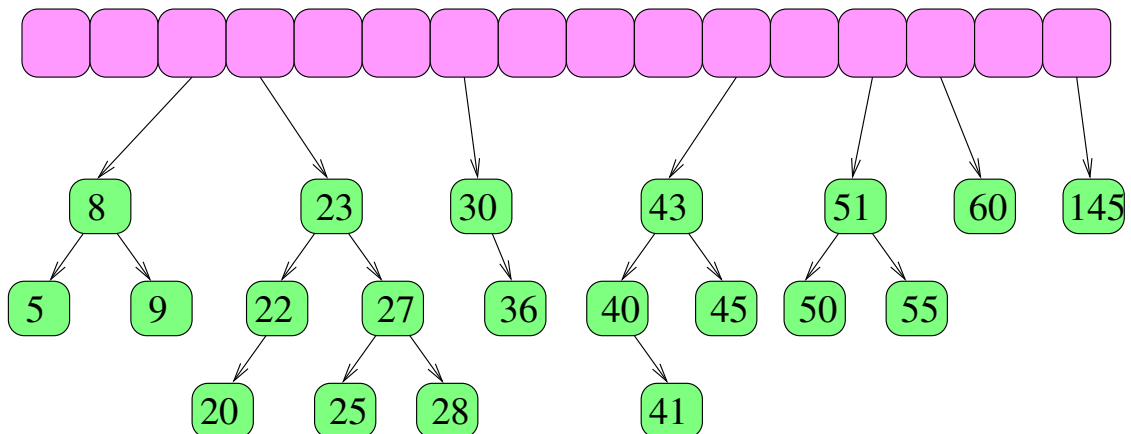
Also universell einsetzbare Datenstruktur, z.B. in einer Bankkontenverwaltung:

- Stammdaten (alle Konto-Attribute, Name, etc.) mit Schlüssel **Kontonummer**
- Für jedes suchrelevante Attribut (Name, Vorname, Stadt, Strasse, Telefon, etc.) eine AVL-Datei mit (Mehrfach-) Schlüssel für jedes Attribut; Daten: **Kontonummer**

Pferdefuß: Daten waren schon immer so umfangreich, dass höchstens ein winziger Bruchteil in den Arbeitsspeicher passt.

### 1.9.2 Hash-Bäume

Können wir AVL-Bäume und Zellraster so kombinieren, dass wir sowohl schnell sind als auch bezüglich des *worst case* mit  $O(\log n)$  abgesichert sind?



Lösung des Problems:

- Einstieg in die Datenstruktur über Zellraster<sup>47</sup>
- In den Zellen jeweils ein ausgeglichener Baum, wir unterstellen AVL wie besprochen

Problem:

Index-Berechnung für Strings. Neu einzutragende Elemente fallen aus dem Indexbereich des Zellrasters heraus, Daten wandern weg, breiten sich aus, etc.

Wie können wir Zeichenketten (Strings) indizieren und einigermaßen brauchbar in ein Intervall von  $\mathbb{R}$  ordnungserhaltend abbilden?

Eine Möglichkeit: GE-Hash (GE=„Gleitpunkt-Einheitsintervall“)  
Definiere die Abbildung GE-Hash :  $A^* \rightarrow [0, 1] \subset \mathbb{R}$ ; Alphabet  $A$  enthält nur ca. 30 Zeichen, Repräsentanten für die üblichen ASCII-Zeichen.

$$\underbrace{\text{Mäander}}_{\in \text{ASCII}^*} \mapsto \underbrace{\text{MAEANDER}}_{\in A^*}$$

Die Zeichen in  $A$  werden quasi als Ziffern behandelt:

„ $\square$ “  $\rightarrow 0$   
„A“  $\rightarrow 1$   
...  $\rightarrow$  ...  
„Z“  $\rightarrow 29$

$$\text{GEHash}(„\text{Mäander}“) := 0, \text{MAEANDER} \in \mathbb{R}$$

Wird nun als Gleitpunktzahl mit den Ziffern  $0, 1, \dots, 29$  zur Basis 30 aufgefasst. Wertebereich im Intervall  $[0, 1] \in \mathbb{R}$ .

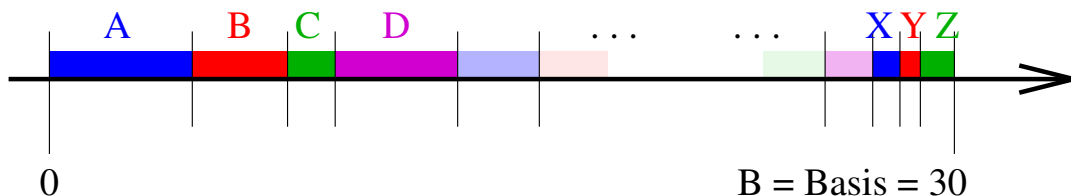
Ist keine übliche Hash-Funktion, weil sie ordnungserhaltend ist.

Bei 56 Bit Mantisse (im Double-Format) und 5-Bit-Ziffern werden 11 Zeichen aufgelöst.

---

<sup>47</sup>also Adressierung eines Feldes

Ungleichverteilung ist erheblich; siehe Duden. Da wir heute extrem schnelle Gleitpunkt-Arithmetik haben, könnte man eine „Gleichverteilungsabbildung“ konstruieren:



Das  $i$ -te Alphabetezeichen so aufgespreizt oder geschrumpft, dass im Intervall  $[0, 1]$  Gleichverteilung aller Wörter der deutschen Sprache entsteht.<sup>48</sup>

Fazit:

GE-Hash kann so konstruiert werden, dass bei bekanntem „Wortschatz“ im Einheitsintervall gute Gleichverteilung erzielt wird.

Aufbau eines Hash-Baumes für die Schlüsselemente  $x_1, x_2, \dots, x_n \in \mathbb{R}$ :

1. Bestimme  $x_{min}, x_{max}$ ;  
#Elemente := 0
2. Initialisiere Zellen:  
array[1..n] of AVLPointer = nil
3. Definiere Index-Abbildung  $IHash(x) \rightarrow \{1, 2, \dots, n\}$  so, dass  $IHash(x_{min}) = 1$  und  $IHash(x_{max}) = n$
4. Für jedes  $i$ :  $insert(x_i, Zelle[IHash(x_i)])$   
#Elemente := #Elemente+1
5. IDcount := 0 (Zähler für insert-delete-Operationen)

Zeitbedarf:

$$T_{min}(n) = O(n) \quad \text{und} \quad T_{max}(n) = O(n \log n)$$

Auf jeden Fall nicht langsamer, als alles in einen AVL-Baum einzufügen.

<sup>48</sup>Problem: Buchstabenverteilung im Wort variiert  $\Rightarrow$  bedingte Wahrscheinlichkeiten besser

**Satz:**

Der Aufbau (Initialisierung) eines Hash-Baumes mit  $n$  unsortierten Elementen hat im schlechtesten Fall die Zeitschranke  $T_{max}(n) = O(n \log n)$ . Bei Gleichverteilung<sup>49</sup> gilt  $T_{mittel}(n) = O(n)$ , wobei hier *floor*-Funktionsaufrufe mit  $O(1)$  bewertet werden.

Durchführung einzelner Operationen:

<b>insert(x)</b>	$x$ in den AVL-Baum von Zelle $IHash(x)$ eintragen (mit vollem Schlüssel, ungehasht) Falls $IHash(x) < 1$ , so in Zelle 1 Falls $IHash(x) > n$ , so in Zelle $n$ Hier Annahme: Elemente liegen nicht notwendigerweise im Intervall $[x_{min}, x_{max}]$ IDcount++
<b>delete(x)</b>	Entferne $x$ aus dem AVL-Baum in Zelle $[IHash(x)]$ , dabei voller Schlüssel genutzt!
<b>member</b>	analog
<b>get</b>	analog
<b>reportall</b>	analog
etc.	analog

**Satz:**

Die für linear verkettete AVL-Bäume besprochenen Operationen sind analog realisierbar. Im Mittel kann der Zeitbedarf von  $O(\log n)$  durch  $O(1)$  ersetzt werden!

Winziges Randproblem:

z.B. **reportup(S,k)**,  $k$  Elemente gewünscht.

Leere Zellen müssen übersprungen werden. Im Mittel kein Problem, aber im *worst case*  $O(n)$  leere Zellen möglich.

Würde es helfen, die lineare Verkettung der AVL-Bäume über Teilbäume hinweg vorzunehmen?

Kann sich der Hash-Baum an wesentlich veränderte Elementzahlen (z.B.  $2n$ , oder  $n/2$ ) anpassen?

Ja. Wenn Zähler **IDcount**  $> n$ , so generiere völlig neuen zweiten Hash-Baum mit den aktuell gespeicherten Elementen. Der

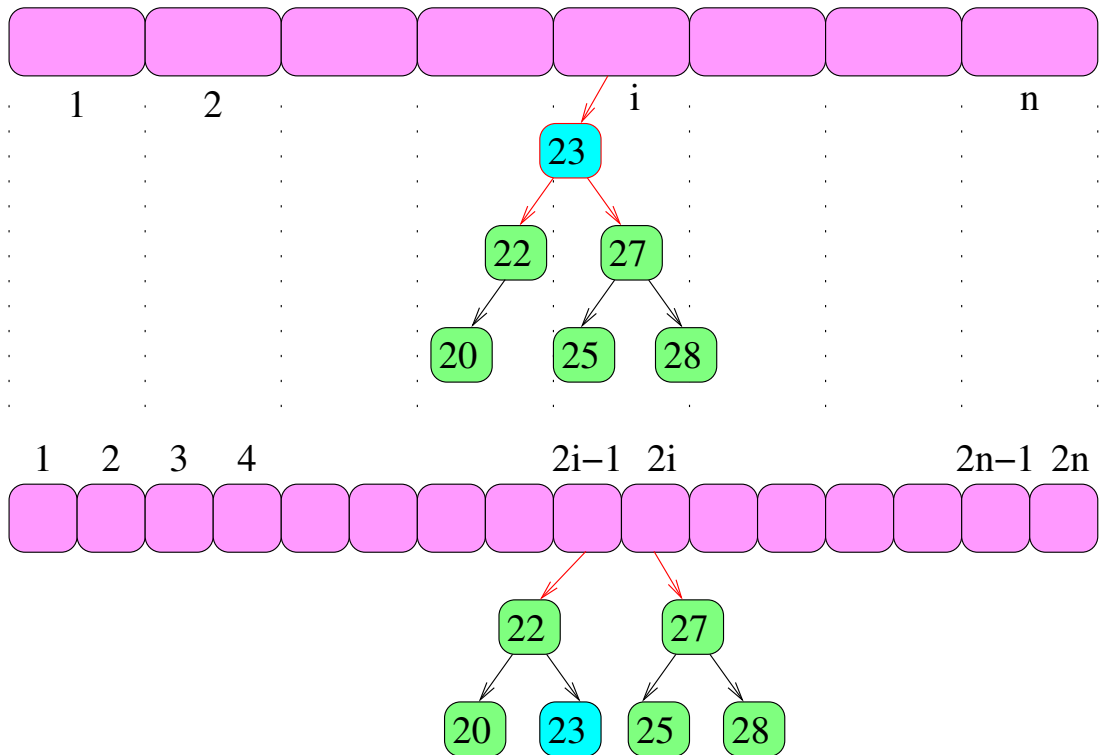
---

<sup>49</sup>oder Ungleichverteilung mit einer Potenzverteilung  $x^k$

Zeitaufwand von  $T_{max}(n) = O(n)$  wird auf die zurückliegenden  $n$  Insert-Operationen verteilt, amortisiert bleibt also  $T_{mittel}(n) = O(1)$  für die einzelnen Operationen bestehen.

Reorganisationsvorgang ist mit der Speicherbereinigung (*garbage collection*) z.B. von Java vergleichbar.

Für Realzeit-Anwendungen kann man das nicht dulden – die Anwendung wäre so lange tot. Man kann kontinuierlich umfüllen:



Alle Elemente  $y < x =$  Grenzschlüssel sind im neuen Baum. **insert**, **delete** etc. müssen das jetzt berücksichtigen. Nach jeder  $O(\log n)$ -Operation werden 2 Elemente vom alten in den neuen Baum umgeschaufelt.

**Satz:**

Der Reorganisationsaufwand kann so zwischen die regulären Operationen verteilt werden – gestückelt –, dass im schlechtesten Fall Zeitintervalle  $T_{max}(n) = O(\log n)$  als Totzeiten auftreten.

Bei geeigneten gleichverteilten Schlüsselmenge und großen  $n$  bringen Hash-Bäume noch einmal deutlich bessere Leistungen als AVL-Bäume. Auch hier gelingt es, die Rechenkraft der *floor*-Operation ins Spiel zu bringen und im Mittel statt  $O(\log n)$  auf  $O(1)$  Operationen zu reduzieren.

### 1.9.3 B-Bäume

Legendäre, viel zitierte Veröffentlichung:

Bayer, R.,<sup>50</sup>, McCreight, C.: „Organization and maintenance of large ordered indexes“, Acta Informatica 1, 3 (1972), 173-189.



Es konnte nicht mehr eindeutig geklärt werden, warum die Bäume B-Bäume heißen: „Balanced“, „Broad“, „Bushy“, „Boeing“ ... Heute Bayer-Trees.

---

<sup>50</sup>Rudolf Bayer, Informatik-Professor an der TU München

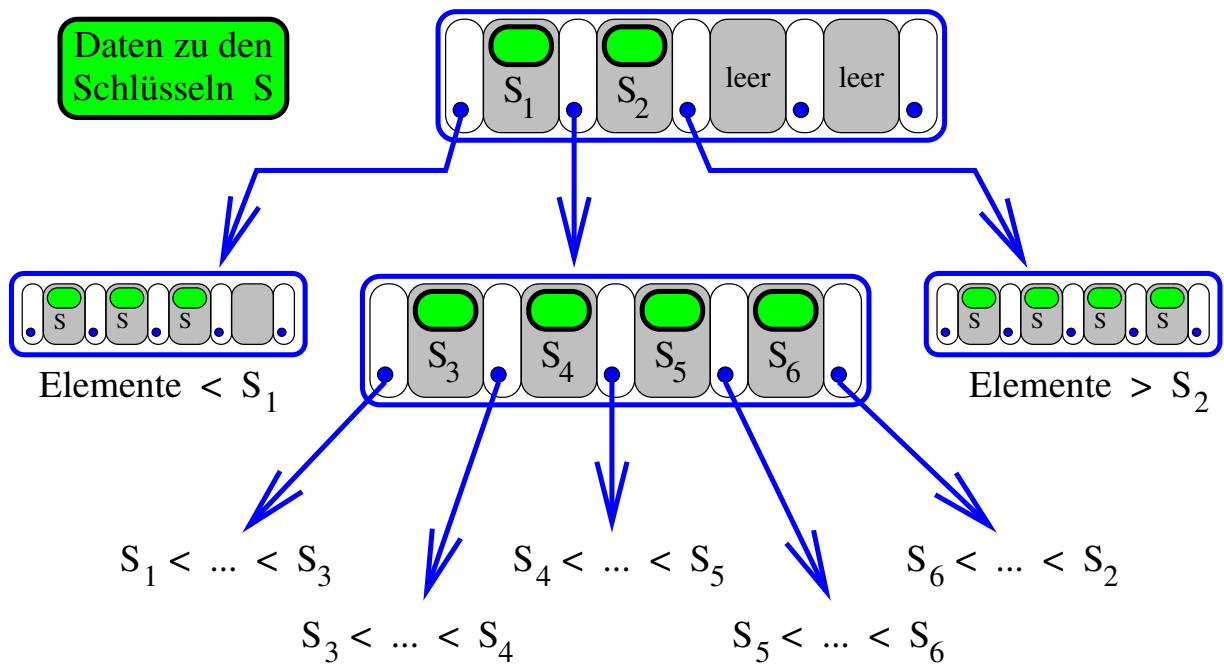
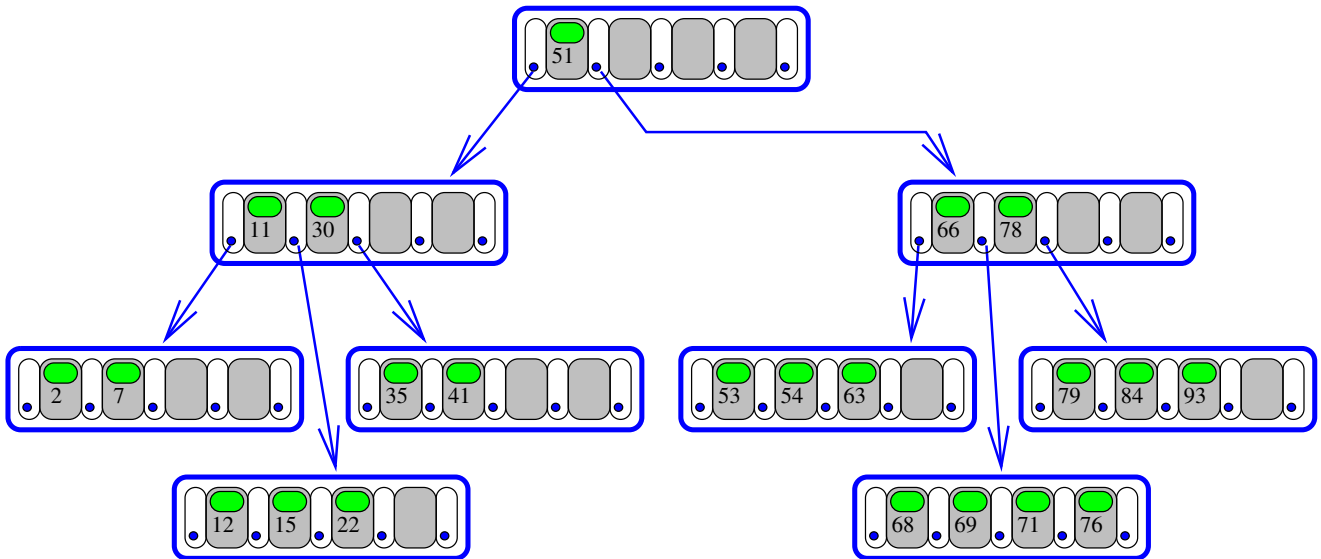


Abbildung 1.12: Prinzip eines 2-4-Baums

Eigenschaften:

- 2-4-Baum-Knoten haben mindestens 2 und höchstens 4 Schlüssel. Mindestens 3 und höchstens 5 Verweise auf Sohn-Knoten.
- Bäume sind vollständig ausgeglichen, d.h. alle Blatt-Knoten haben gleiche Tiefe.
- Normalfall: Daten sind im Knoten, also bei den Schlüsseln gespeichert.
- Im allgemeinen: Knoten können 20, 40 oder gar 100 Schlüssel tragen, bei  $2n$  maximalem Inhalt müssen mindestens  $n$  Schlüssel vorhanden sein, also mittlere Speicherausnutzung ca. 75%.
- Anpassung an Hardware: Knoten  $\approx$  Speicherseite bei virtueller Speicherorganisation. Page-Grösse: 1k, 4k, 16k Byte, ...

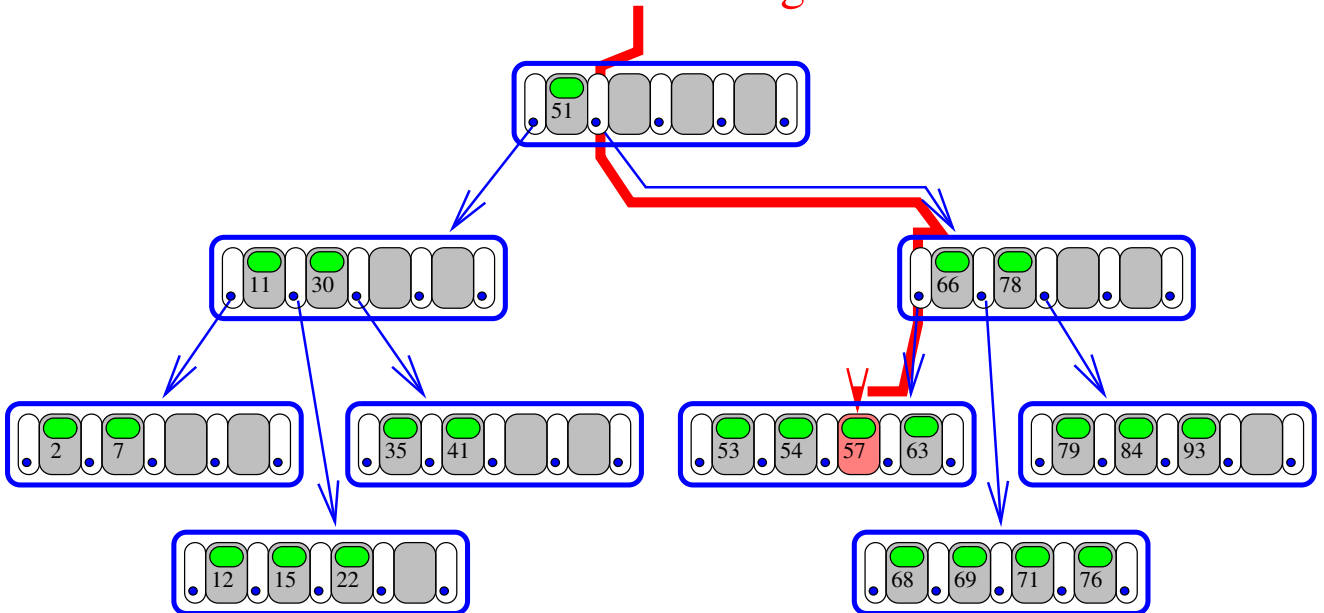
- Die alles überragende Methode für sehr große schlüsselzugreifende Datenbestände



Entfernen von 41:

1. Lösche 41, Unterlauf: Borge ein Element vom Nachbarknoten:
2. Transferiere 30
3. Transferiere 22
4. Ergebnis:

## Schlüssel 57 einfügen



Entfernen von 51:

1. Suche nächstgrößeres oder nächstkleineres Element (53 bzw. 41)
2. Da kein Knotenunterlauf, ist 53 sofort verfügbar.  
Transferiere 53 auf Platz von 51.

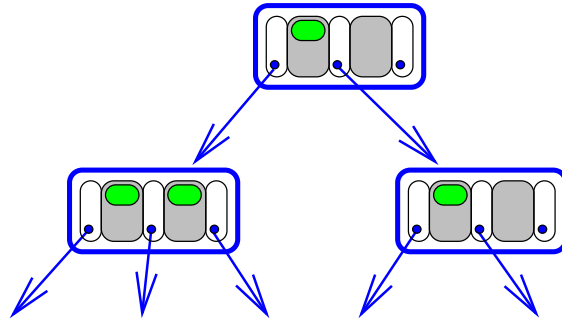
**Definition** der **B-Bäume** zur Ordnung  $m$  ( $m \geq 2$ )

1. Mit Ausnahme der Wurzel hat jeder Knoten mindestens  $\text{floor}\left(\frac{m}{2}\right) = \left\lfloor \frac{m}{2} \right\rfloor$  Schlüssel (und Datensätze).
2. Alle Wege von der Wurzel zu den Blättern sind gleich lang, d.h. der Baum ist perfekt ausgeglichen.
3. Wenn ein Knoten  $k$  Schlüssel hat, so hat er  $k + 1$  Verweise auf seine Söhne.
4. Blatt-Knoten haben leere Verweise auf Söhne.
5. Hat ein Knoten die Schlüssel  $S_1 < S_2 < \dots < S_k$  und die Verweise  $Z_0, Z_1, \dots, Z_k$ , so muss gelten:  
Alle über  $Z_i$  ( $i = 1, 2, \dots, k - 1$ ) erreichbaren Knoten haben

Schlüssel  $S$  mit  $S_i < S < S_{i+1}$ , die über  $Z_0$  erreichbaren Knoten haben Schlüssel  $S$  mit  $S < S_1$  und die über  $Z_k$  erreichbaren Knoten haben Schlüssel  $S > S_k$ .

Spezialfälle:

Ein B-Baum der Ordnung  $m = 2$  heißt **binärer B-Baum**,<sup>51</sup> hat also Verzweigungsgrade 2 und 3.



2-4-Bäume sind uns bekannt, werden manchmal auch 2-3-4-Bäume genannt.

### B-Bäume für große Dateien

Große B-Bäume für Speicherung auf Festplatten-Laufwerken: Die elementar (von der Hardware her) adressierbare kleinste Speichereinheit – 1 „Block“ – ist perfekt für 1 Knoten, entspricht auch der Page-Grösse beim Paging.

#### Beispielrechnung:

Page-Größe: 1k Byte  $\approx$  Speicher für 1 Knoten

Maximale Schlüssellänge: 12 Byte

Daten pro Schlüssel: 30 Byte

Erster Zeiger: 4 Byte extra

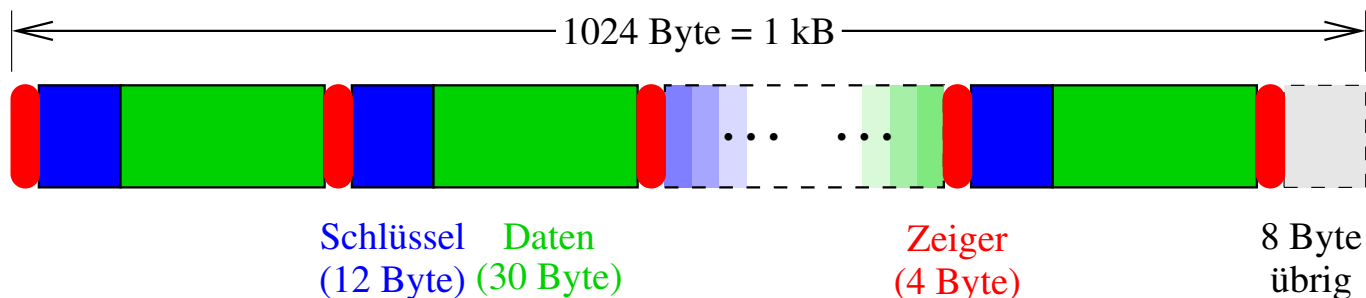
$$1024 = (m + 1) \cdot \underbrace{4}_{\text{Schlüssel}} + m \cdot \underbrace{(12 + 30)}_{\text{Record}}$$

$$= 46 \cdot m + 4$$

$$46 \cdot m = 1020 \quad \Rightarrow \quad m = \left\lfloor \frac{1020}{46} \right\rfloor = 22$$

<sup>51</sup>wegen der Verzweigungsgrade auch: 2-3-Baum

In 1024-Byte-Speicherblöcke können wir 22 Schlüssel unterbringen:



(Statt 46 Byte pro Record müssen wir  $\approx 1/3$  Aufschlag für 75% Speichernutzung einkalkulieren, also  $\approx 64$  Byte, großzügig nach oben abgeschätzt)

Annahme: Die Datei muss 1 Giga-Records aufnehmen, hat also ein Volumen von ca.

$$10^9 \cdot 64\text{Byte} = 64\text{GigaByte}$$

Wie tief ist der B-Baum im Beispiel?

Ein Knoten hat im Mittel (bei 75% Auslastung) 16 Schlüssel  $\approx$  Records

$$R_0 := 16$$

$$R_1 = 16 + 16^2$$

$$R_h = \sum_{j=1}^{h+1} 16^j > 16^{h+1}$$

$$10^9 = 16^{h+1}$$

$$10^9 \approx 2^{30} \quad (\text{denn } 10^3 \approx 2^{10})$$

$$2^{30} = 2^{4 \cdot 7.5} = 16^{7.5} \approx 16^{h+1}$$

$$\Rightarrow 7,5 \approx h + 1 \text{ und } h \approx 6,5$$

Der Baum hat also eine Höhe von 6 oder 7.

Wie viele Plattenzugriffe sind bei ca. 1 MByte verfügbarem Zentralspeicher für die Top-Knoten erforderlich, um auf einen

Record zuzugreifen?

$$\begin{aligned} 10^6 &\approx 2^{20} \stackrel{!}{=} 16^{h+1} \\ 2^4 \cdot 5 &= 16^5 \stackrel{!}{=} 16^{h+1} \Rightarrow h = 4 \end{aligned}$$

Die ersten 4 Ebenen des B-Baumes können also locker um Zentral-  
speicher bereitgestellt werden, d.h.:

Ein Record (Schlüssel + Datensatz) der 64 GigaByte umfas-  
senden Datei kann mit 2 bis 3 Plattenspeicherzugriffen gelesen  
werden, also wenige Millisekunden.

Mit Binärbäumen wie z.B. AVL hat man große Schwierigkei-  
ten, die Knoten so in Speicherseiten abzulegen, dass mit ähnlich  
geringer Zahl von Seitenzugriffen zu rechnen ist.

Noch ein Problem: Was ist ein Zeiger (Pointer)?

## Die Operationen member, insert und delete

**member** :

Suche Daten und Schlüssel  $S$

Verfolge die Zeiger beginnend beim Wurzelknoten rekursiv wie folgt:

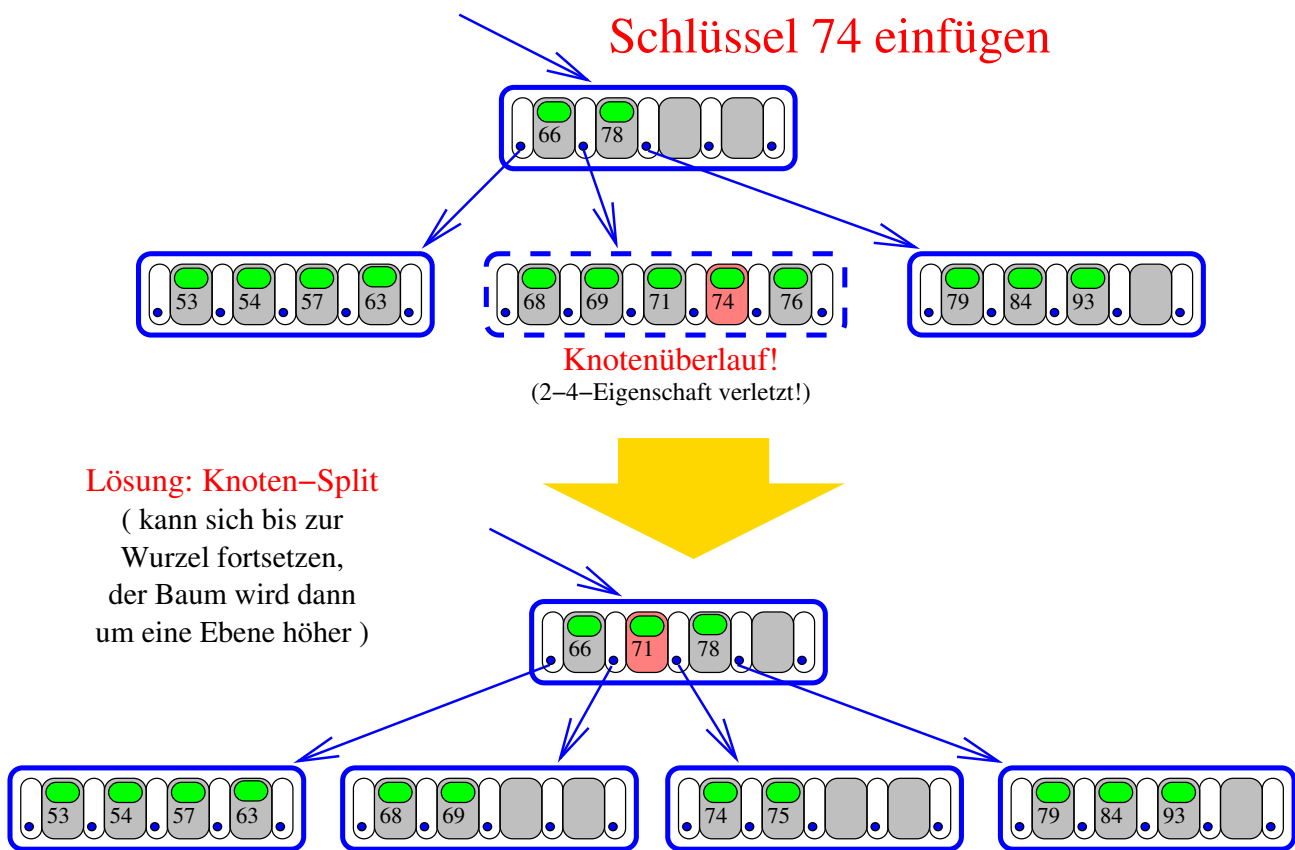
Prüfe Knoten  $Z_0 \ S_1 \ Z_1 \ S_2 \ Z_2 \ S_3 \ Z_3 \ S_4 \ Z_4$  (hier: 2-4-Baum).

- Wenn  $S = S_i$  für eines der  $S_i$  im Knoten, so Schlüssel und  
Daten gefunden, fertig.
- Wenn  $S < S_1$ , verfolge  $Z_0$  (rekursiv),  
sonst: wenn  $S < S_2$ , so verfolge  $Z_1$  (rekursiv),  
sonst: Wenn  $S < S_3$ , so verfolge  $Z_2$  (rekursiv),  
sonst: Wenn  $S < S_4$ , so verfolge  $Z_3$  (rekursiv),  
sonst verfolge  $Z_5$  (rekursiv).
- Wenn für einen der zu verfolgenden Zeiger  $Z = \text{nil}$  gilt, so  
Element mit Schlüssel  $S$  nicht im Baum; **member:=FALSE**

(Anmerkung: Da die Knoten zwischen  $m/2$  und  $m$  Schlüssel ent-  
halten, muß die aktuelle Schlüsselzahl eines Knotens im Knoten  
gespeichert sein)

**insert (S,D)** :

1. Suche  $S$  im Baum. Wenn gefunden: Tausche Daten  $D$  aus und fertig.
2. Suche erfolglos: Endet in Blattknoten.  
Wenn Zahl der Schlüssel im Knoten  $< m$ , füge Record  $(S,D)$  in den Knoten ein; fertig.
3. Blattknoten hat bereits  $m$  Schlüssel:  
Knotenüberlauf: aus  $m + 1$  Schlüsseln werden 2 Knoten mit je  $m/2$  Schlüssel gesplittet und der mittlere Schlüssel wird im Vaterknoten eingefügt. (Knotensplitt kann bis zur Wurzel aufsteigen und die Tiefe des Baumes um Eins erhöhen)



- Bei Überlauf eines Knotens wandern die jeweils mittleren Schlüssel des übergelaufenen Knotens nach oben.
- Balance bleibt erhalten.

- Überlauf an Wurzel: Neue Wurzel hat 1 Schlüssel, Baumhöhe erhöht sich um 1

**delete(S) :**

1. Suche  $S$  im Baum
2. Gefunden in Blattknoten<sup>52</sup>: Lösche Record ( $S,D$ ).  
Eventuell Unterlauf, zu wenig Schlüssel im Knoten
3. Gefunden in innerem Knoten: Platz muss besetzt bleiben<sup>53</sup>.  
Record-Tausch mit größtem Schlüssel  $S' < S$  oder kleinstem Schlüssel  $S' > S$ .  
Danach ist  $S$  in einem Blattknoten. Weiter bei Schritt 2.

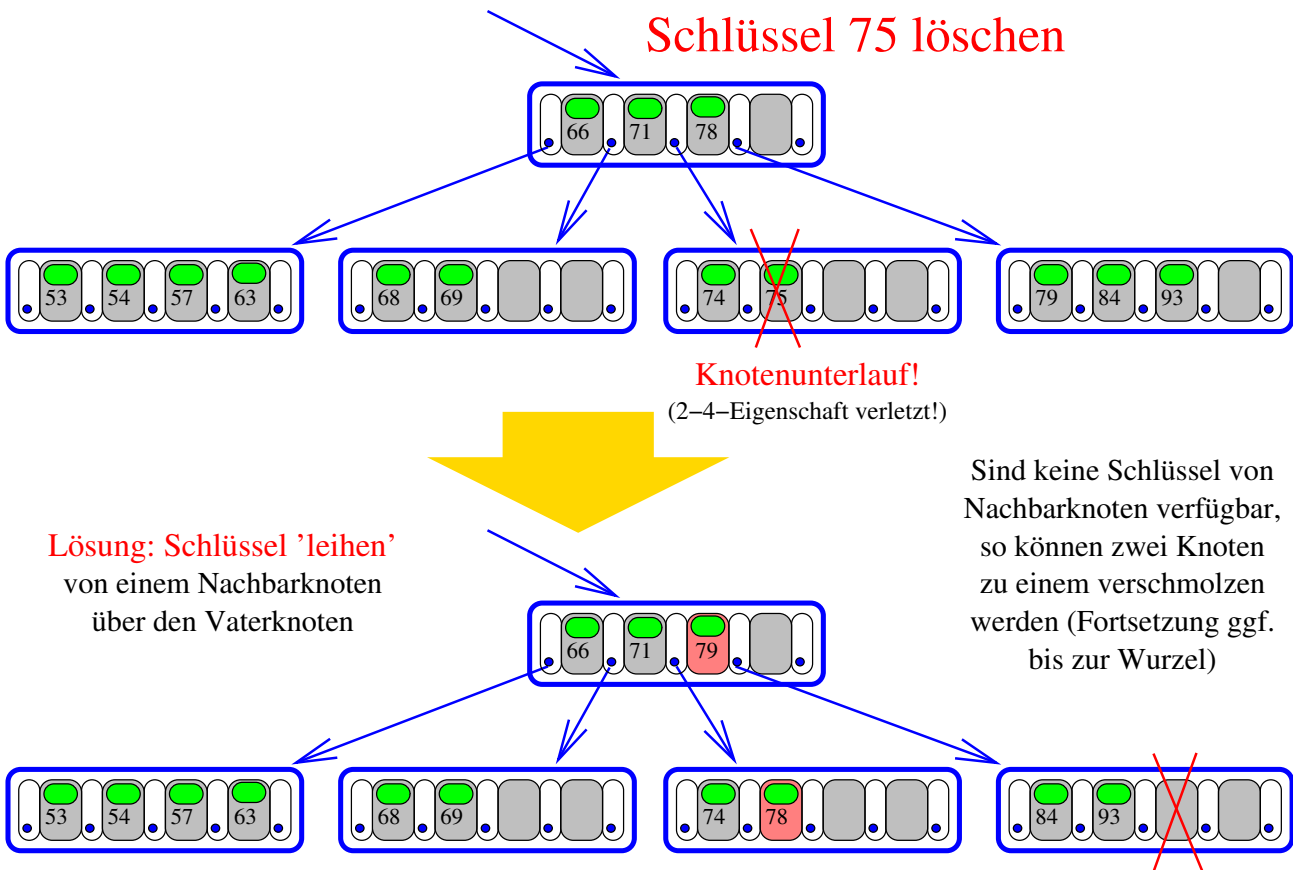
Restproblem: Unterlauf (2 Fälle)

1. Man kann sich vom Nachbarknoten einen oder mehrere Schlüssel leihen
2. Wenn nicht (Nachbarknoten am Minimum): Verschmelzen von Knoten

---

<sup>52</sup>hat hohe Wahrscheinlichkeit

<sup>53</sup>wie bei Binärbäumen



**Konsequenzen:**

Vaterknoten hat 1 Schlüssel weniger. Falls Unterlauf, Vorgehen wie in Schritten 1. und 2.

- Balance bleibt erhalten
- Unterlauf bei Wurzelknoten mit  $< 1$  Schlüssel:  
Wurzelknoten verschwindet, der verschmolzene ist neuer Wurzelknoten, Höhe um 1 verringert.

**$T_{max}$ -Effizienz von B-Bäumen**

Offensichtlich gilt  $T_{max}(n) = O(\log n)$  für die wichtigen Einzeloperationen `member`, `insert`, `delete`. Auch bezüglich der anderen bei AVL-Bäumen besprochenen Operationen ergeben sich die gleichen Verhältnisse wie bei AVL-Bäumen. Aber aufgepasst: Wie viele Ver-

gleichsoperationen sind z.B. bei `member` erforderlich?

$$n \approx \left(\frac{3}{4}m\right)^{h+1}$$

$$\log n \approx (h+1) \cdot \log\left(\frac{3}{4}m\right)$$

$$\log_2 n \approx (h+1) \cdot \log_2\left(\frac{3}{4}m\right)$$

Hierbei ist  $\log_2\left(\frac{3}{4}m\right)$  die mit dem Halbierungsverfahren erforderliche Zahl von Schlüsselvergleichen, um in einem Knoten mit  $\frac{3}{4}m$  Schlüsseln, das Element oder den Fortsetzungszeiger zu lokalisieren!

Ergebnis:

Auch bei B-Bäumen mit  $n$  Elementen sind im Mittel  $\log_2 n$  Vergleichsoperationen erforderlich, um einen bestimmten Schlüssel zu lokalisieren.

## Varianten von B-Bäumen

### B\*-Bäume:

Die Elemente eines überlaufenden Knotens werden auf Nachbarknoten verteilt. Erst wenn auch der Nachbarknoten „voll“ ist, werden aus 2 Knoten 3 gemacht, die dann mindestens zu 66% gefüllt sind<sup>54</sup>.

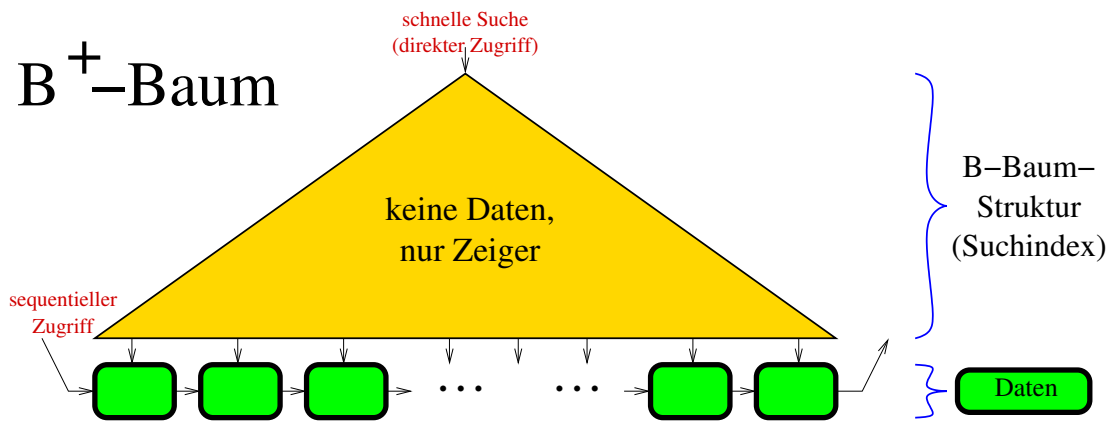
### B<sup>+</sup>-Bäume:

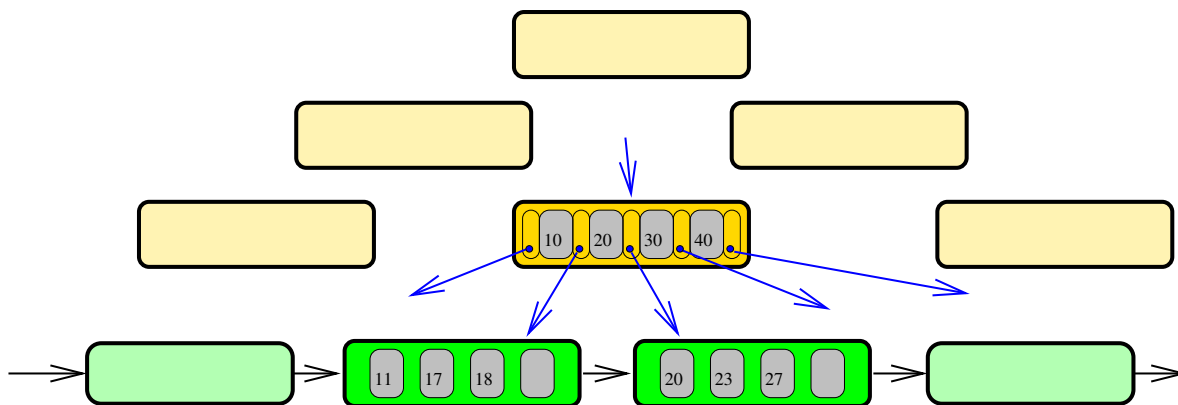
Die Records sind linear verkettet und tragen Schlüssel und Daten. Die höheren Knoten enthalten nur Indizes, die nicht notwendigerweise im Datenbereich vorkommen.

---

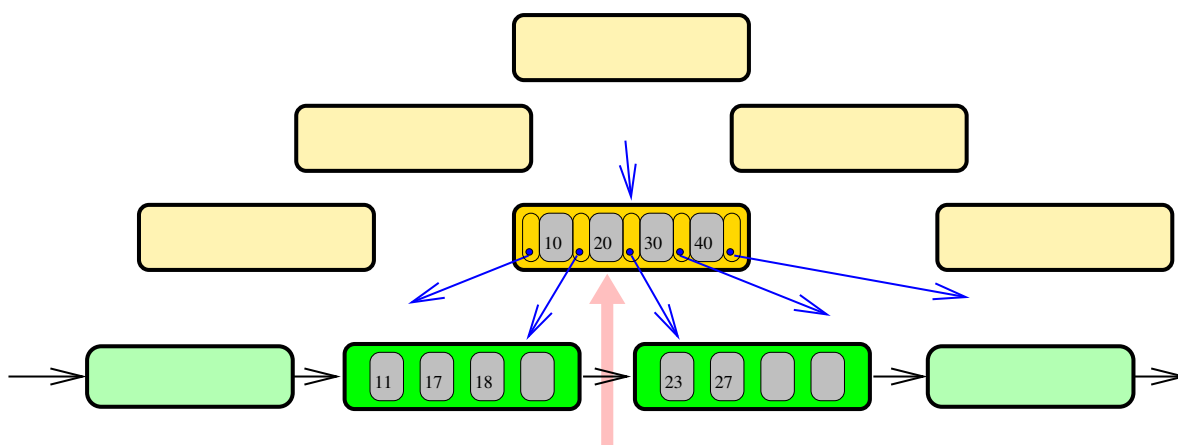
<sup>54</sup>normaler B-Baum: 50%

# B<sup>+</sup>-Baum





Löschen der Daten mit dem Schlüssel 20:



in der Verzeigerungsstruktur des überliegenden B-Baums bleibt die 20 als Separatorwert erhalten, obwohl der Schlüssel im Datenteil gelöscht wurde

Literatur: D. Comer: „The Ubiquitous B-Tree“, Computing Surveys (AVM), Vol. 11, No. 2, June 1979, 121-137.