# Skript zur Vorlesung

# Datenstrukturen und Algorithmen

Wintersemester 1995/96

Prof. Seidel

# Vorwort

Dies sind sogenannte *"Scribe Notes"* über die Vorlesung **Datenstrukturen und Algorithmen**, gehalten an der Universität des Saarlandes im Wintersemester 1995/96.
Jeder ernsthafte Teilnehmer dieser Veranstaltung mußte eine Vorlesung mitschreiben und dann auch fein säuberlich "geTeXed" zu Papier bringen, wenn nötig auch mit Zeichnungen. Diese ausgearbeiteten Mitschriften wurden darauf von *Nicola Geismann* und dann noch von mir selbst in zwei weiteren Iterationen überarbeitet. Bei vielen der Mitschriften waren kaum Verbesserungen notwendig, andere spiegelten wohl zu sehr die Schwächen des Vortrags wider und brauchten stärkere Überarbeitungen.
Das Resultat hat natürlich nicht die Qualität eines Buches, ist aber wohl brauchbarer als ein von Skriptum, das von ein oder zwei Personen mit zu wenig Zeit erstellt wurde. Es fehlen Literaturverweise, und es sei hier gesagt, daß sich Teile der Vorlesung stark an die genannten Bücher von Mehlhorn und von Cormen, Leiserson, Rivest anlehnen. Als Experiment halte ich diese Art, eine Mitschrift zu erstellen, für gelungen.

Raimund Seidel

### Die Vorlesungen und ihre "Scribes"

| | | |
|---|---|---|
| Mo | 16.10.1995 | Jochen Könemann |
| Do | 19.10.1995 | Holger Bielenstein, Jochen Wagner, Mark Wehrmann |
| Mo | 23.10.1995 | Hans-Georg Dörr, Ulrich Hoffmann, Frank Oster |
| Do | 26.10.1995 | Tobias Miller, Christian Zenner |
| Mo | 30.10.1995 | Jörg Deßloch, Matthias Hüther |
| Do | 2.11.1995 | Marc Langenbach, Michael Schmidt |
| Mo | 6.11.1995 | Ulrich Alt, David Christensen, Stefan Janocha |
| Do | 9.11.1995 | ausgefallen wegen Geburt von *Marvin Emil S.* |
| Mo | 13.11.1995 | Nikolaus Gerteis, Michael Rumpf |
| Do | 16.11.1995 | Michael Eisemann, Marc Fontaine |
| Mo | 20.11.1995 | Ludger Bünger, Holger Sabo |
| Do | 23.11.1995 | Mirko Jerrentrup, Volker Morbach, Torsten Welller |
| Mo | 27.11.1995 | Marc Hohlweck, Frank Jost, Heiko Klein |
| Do | 30.11.1995 | Oliver Plaehn, Michael Rovatsos, Bianca Schröder |
| Mo | 4.12.1995 | Ralf Mindermann, Bare Said, Sven Schaumann |
| Do | 7.12.1995 | Ulrich Klos, Henrik Stormer |
| Mo | 11.12.1995 | Jürgen Zimmer |
| Do | 14.12.1995 | Thorsten Mohr, Andreas Wendling |
| Mo | 18.12.1995 | Jörg Brück, Igor Pouchkarev, Heiko Schütz |
| Do | 21.12.1995 | Thorsten Heit |
| Mo | 8.1.1996 | Michael Brandau, Jürgen Mohra |
| Do | 11.1.1996 | Sebastian Braun, Wolfgang Holz, Jens Simon |
| Mo | 15.1.1996 | Michaela Altherr, Ulrich Laux |
| Do | 18.1.1996 | Christian Fink, Tanja Maurer, Pascal Schüler |
| Mo | 22.1.1996 | Thomas Martin, Peter Noss |
| Do | 25.1.1996 | Dirk Holz, Marco Nissen, Stefan Reichert |
| Mo | 29.1.1996 | Stefan Dietrich, Daniel Wagner |
| Do | 1.2.1996 | Ralf Mindermann, Sven Wohlgemuth |
| Mo | 5.2.1996 | Daniel Kästner, Daniel Kreutzer |
| Do | 8.2.1996 | René Beier, Henning Biermann, Henrik Theiling |
| Mo | 12.2.1996 | Tim Schwartz, Daniel Wannemacher, Robert Wirth |
| Do | 15.2.1996 | Klausur |

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Vorlesung vom 16.10.1995
Jochen Könemann

## 1.1 Literaturhinweise und Scheinbedingungen

Bücher: Es gibt eine ganze Reihe von Büchern, die sich mit Algorithmen und Datenstrukturen beschäftigen. Hier sind einige englische und deutsche aufgezählt. Natürlich ist die Liste unvollständig. Allgemein werden in der englischsprachigen Literatur Datenstrukturen weniger behandelt.

1. Cormen,Leiserson,Rivest: **Introduction to Algorithms.** MIT Press and McGraw-Hill (1990).
   Der neue Algorithmenklassiker in der englischsprachigen Literatur; äußerst studentenfreundlich geschrieben.

2. Aho,Hopcroft,Ullman: **The Design and Analysis of Computer Algorithms.** Addison-Wesley (1974).
   Der alte Klassiker, der aber immer noch sehr viel Interessantes enthält.

3. Mehlhorn: **Data Structures and Algorithms.** Springer (1984), auch auf Deutsch erschienen als **Datenstrukturen und Algorithmen.** Teubner (1988).
   Enthält sehr viel, ist allerdings etwas knapp gefaßt geschrieben.

4. Rawlins: **Compared to what?** Freeman (1992).
   Ein etwas unkonventionell geschriebenes Buch.

5. Manber: **Introduction to Algorithms: A creative Approach.** Addison-Wesley (1989).
   Eines der vielen durchschnittlich bis guten englischsprachigen Algorithmenbücher.

6. Ottmann und Widmayer: **Algorithmen und Datenstrukturen.** Wissenschaftsverlag (1990). Ein relativ umfangreiches, relativ studentenfreundlich geschriebenes Buch.

Voraussetzungen für die Scheinvergabe:

1. 45 % Übungen (die schlechtesten 3 Übungen zählen nicht)

2. 45 % Klausur

3. 10 % Scribe Notes

## 1.2 Allgemeines Maschinenmodell

### 1.2.1 Maschinenmodell:

Bevor begonnen werden kann, die Effizienz von Algorithmen zu analysieren, ist es notwendig, das Maschinenmodell einzuführen, auf dem diese Algorithmen laufen sollen. Wir nehmen an, daß unsere Algorithmen auf der

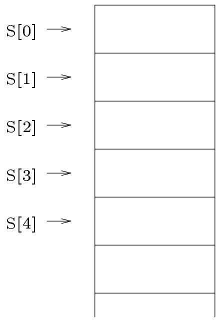



Abbildung 1.1: Speicher $S$ einer RAM

*RAM* (Random Access Machine) ausgeführt werden. Unsere RAM ist etwas ungewöhnlich, da sie je nach Bedarf mehr als eine Art von Speicher haben kann. Sie besitzt immer einen Speicher $S$ von linear angeordneten Zellen. Die Zellen von $S$ enthalten ganze Zahlen. Diese Zahlen dürfen allerdings nicht beliebig groß werden. Vielmehr nehmen wir an, daß die Zahlen polynomiell groß bleiben, das heißt, daß es für jeden Algorithmus eine Konstante $k$ gibt, sodaß bei Eingabegröße $n$ (weiter unten definiert) immer für alle $i \in \mathbb{N}$ gilt $|S[i]| < n^k$. Dies bedeutet, daß $(k+1)\log_2 n$ Bits für die Realisierung einer Speicherzelle ausreichen. [1]
Folgende Operationen sind auf $S$ erlaubt und benötigen jeweils eine Zeiteinheit:

- $S[I] \leftarrow \mathit{Konstante}$ (Zuweisung)
- $S[I] \leftarrow S[J]$ (Zuweisung)
- $S[I] \leftarrow S[J]\ \mathit{op}\ S[K]$ mit $\mathit{op} \in \{+, -, *, /, \bmod\}$
- $S[I]\ \mathit{vgl}\ S[J]$ mit $\mathit{vgl} \in \{<, >, \leq, \geq, =, \neq\}$ (Vergleiche)

Dabei stehen $I$, $J$ und $K$ entweder für eine Konstante $i \in \mathbb{N}$ oder einen Ausdruck $S[j]$ mit $j$ einer Konstanten in $\mathbb{N}$. Das bedeutet also, daß *indirekte Adressierung* möglich ist: $S[j] \leftarrow S[S[i]]$.
Neben dem normalen Speicher $S$ kann unsere RAM auch noch einen anderen linearen Speicher $E$ (oder auch mehrere) haben, dessen Zellen Elemente eines je nach Anwendung zu bestimmenden einheitlichen Typs sind. Der Einfachheit halber sagen wir einmal, die Elemente seien alle Elefanten, und wir nennen den Speicher $E$ den *Elefantenspeicher.*
Folgende Operationen sind auf diesem Elefantenspeicher erlaubt und benötigen jeweils eine Zeiteinheit:

- $E[I] \leftarrow \mathit{Konstante}$ (Zuweisung)
- $E[I] \leftarrow E[J]$ (Zuweisung)
- $E[I] \leftarrow E[J]\ \mathit{op}\ S[K]$ mit *op* aus einer zu definierenden Verknüpfungsoperationsmenge
- $E[I]\ \mathit{vgl}\ E[J]$ mit *vgl* aus einer zu definierenden Vergleichsoperationsmenge (die typischerweise mindestens $=$ und $\neq$ enthält)

Die Ausdrücke $I$, $J$, und $K$ seien dabei wie vorher definiert.
Um ein konkretes Beispiel zu haben, stelle man sich vor, der Elefantenspeicher enthalte reelle Zahlen, die mit unbegrenzter Genauigkeit gespeichert sind. Die Verknüpfungsoperationen seien die üblichen $+$, $-$, $*$, und $/$, aber vielleicht auch die Exponentialfunktion ^. Die Vergleichsoperationen seien die üblichen $<, \leq, =$ und $\neq$. Die große Annahme hier ist, daß diese Operationen in konstanter Zeit durchgeführt werden können, trotz der unbegrenzten Genauigkeit. Diese Annahme ist natürlich nicht realistisch, vereinfacht aber die Analyse vieler Algorithmen.
Die Eingabe für einen Algorithmus steht in den Bandzellen $S[0], \ldots, S[n-1]$ und gegebenenfalls dazu noch in $E[0], \ldots, E[n-1]$, wobei $n$ die *Eingabegröße* ist.
Beachte:

[1] Diese Beschränkung scheint eigenartig zu sein. Macht man sie aber nicht, und nimmt man, so wie wir es tun wollen, konstante Zeit pro Operation an, dann kann man sehr schwere Problem sehr leicht lösen. Siehe zum Beispiel A. Shamir's Arbeit "Factoring numbers in $O(n \log n)$ arithmetic step" in *Information Processing Letters 8*, (1979) pp. 28–31.

- Die Zahlen in $S$ sind polynomiell beschränkt.
- Jede Operation kann in konstanter Zeit ausgeführt werden.
- Operationen sind entweder auf $S$ oder $E$ definiert. Es ist also insbesondere keine Zuweisung eines Elements aus dem E-Speicher an eine Speicherzelle von $S$ zulässig.

### 1.2.2 Programme

Wir beschreiben unsere Algorithmen natürlich nicht in der "Assemblersprache" der RAM, sondern in einer höheren imperativen Programmiersprache. Wir wollen uns dabei viele Freiheiten lassen und uns nicht auf eine bestimmte Sprache wie PASCAL oder C++ festlegen. Die Bedeutung der verwendeten Sprachkonstrukte und, wie sie in RAM Programme übersetzt werden müßten, wird hoffentlich klar sein. Man beachte, daß wir Blockbildung in Programmen fast ausschließlich durch Einrücken des entsprechenden Textes darstellen werden und nicht durch **begin** und **end** Paare oder ähnliches.

## 1.3 Laufzeitanalyse an einem Beispiel

Wir versuchen nun einen einfachen Algorithmus für ein einfaches Problem zu analysieren. Das soll heißen, daß wir sein Laufzeitverhalten verstehen wollen. Wir behandeln dabei das folgende

Sortierproblem

| | |
|---|---|
| Eingabe: | Folge $\langle a_1, \ldots, a_n \rangle$ mit $n \in \mathbb{N}$. |
| Ausgabe: | Umgeordnete Folge $\langle a'_1, \ldots, a'_n \rangle$, so daß $a'_1 \le a'_2 \le \ldots \le a'_n$ gilt. |
| | Wichtig ist also, daß auf $\{a_1, \ldots, a_n\}$ eine lineare Ordnung $\le$ definiert sein muß. |
| Instanz des Problems: | |
| Eingabe: | $\langle 31, 41, 59, 26, 41, 58 \rangle$ |
| Ausgabe: | $\langle 26, 31, 41, 41, 58, 59 \rangle$ |

Im Folgenden wird nun der Insertion Sort Algorithmus (manchmal auch "Bubble Sort" genannt) an obiger Instanz vorgestellt. Der Algorithmus versucht, immer länger werdende Präfixe der Eingabefolge zu sortieren. Um aus dem sortierten Präfix der Länge $j-1$ den sortierten Präfix der Länge $j$ zu erhalten muß nur das Element $a_j$ an die richtige Stelle geschoben werden. Dies geschieht dadurch, daß die Zahl $a_j$ in der Folge nach vorn *geschoben* wird, bis ihr Vorgänger kleiner oder gleich ihrem Wert ist. Die Skizze 1.2 soll die Arbeitsweise des Algorithmus veranschaulichen.

Der Insertion Sort Algorithmus
Die zu sortierende Folge liegt in einem Feld $A[1..n]$.

```
(1)   for j := 2 to length(A) do
(2)       key ← A[j];
          (* füge A[j] in sortierte Folge A[1]...A[j−1] ein *)
(3)       i ← j − 1;
(4)       while (i > 0 and A[i] > key) do
(5)           A[i+1] ← A[i]
(6)           i ← i − 1;
(7)       A[i+1] ← key;
```

Zur Laufzeitanalyse dieses Algorithmus nehmen wir an, daß die einmalige Exekution der Zeile $(k)$ des Programmes konstante Zeit $c_k$ kostet. Wenn Zeile $(k)$ dann $e_k$ mal ausgeführt wird, dann ist die Gesamtlaufzeit des Algorithmus $\sum_k c_k e_k$. In der folgenden Tabelle werden nun die einzelnen Zeilen mit den (jeweils konstanten) Kosten $c_k$ und Aufrufanzahlen $e_k$ bei einer Eingabe der Länge $n$ aufgeführt:

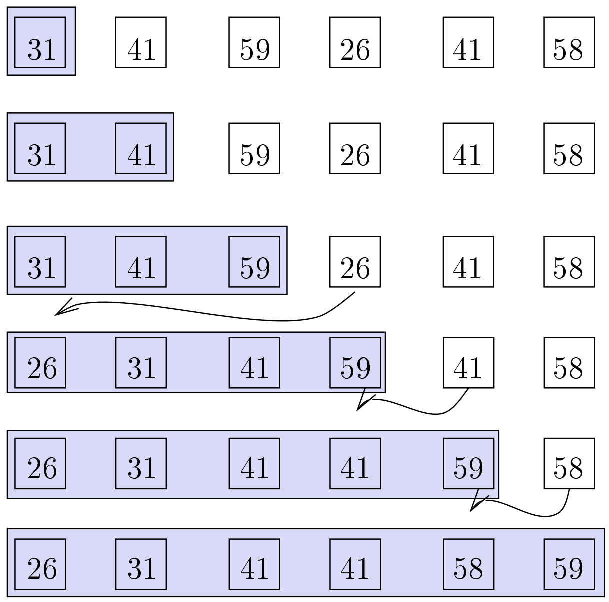



Abbildung 1.2: Arbeitsweise von Insertion-Sort

| Zeilennr. | Kosten | Anzahl Aufrufe |
|---|---|---|
| 1 | $c_1$ | $n$ |
| 2 | $c_2$ | $n-1$ |
| 3 | $c_3$ | $n-1$ |
| 4 | $c_4$ | $\sum_{j=2}^n t_j$ |
| 5 | $c_5$ | $\sum_{j=2}^n (t_j-1)$ |
| 6 | $c_6$ | |
| 7 | $c_7$ | $n-1$ |

Hierbei beschreibt $t_j$, wie oft die **while**-Bedingung in Zeile 4 für $A[j]$ getestet wird. $t_j - 1$ kann dann intuitiv als Anzahl der Verschiebungen von $A[j]$ interpretiert werden. Aus der obigen Tabelle folgt dann leicht die Gesamtlaufzeit:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + (c_5 + c_6) \cdot \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Mit $c_4 \cdot \sum_{j=2}^n t_j = c_4 n - c_4 + c_4 \cdot \sum_{j=2}^n (t_j - 1)$ folgt:

$$T(n) = \underbrace{(c_1 + c_2 + c_3 + c_4 + c_7)}_{A} \cdot n - \underbrace{(c_2 + c_3 + c_4 + c_7)}_{B} + \underbrace{(c_4 + c_5 + c_6)}_{D} \cdot \sum_{j=2}^{n} (t_j - 1)$$

Die Terme $t_j$ hängen natürlich von der jeweiligen Eingabe ab und können nicht genau bestimmt werden. Deshalb betrachten wir die zwei extremen Fälle bester und schlechtester Fall, sowie auch den "durchschnittlichen" Fall:

1. <u>bester Fall:</u> $t_j = 1$ für alle $2 \leq j \leq n$, d.h. hier ist die Folge schon aufsteigend sortiert.
   $T(n) = An - B$

2. <u>schlechtester Fall:</u> $t_j = j$ für alle $2 \leq j \leq n$, d.h. hier ist die Zahlenfolge absteigend sortiert.
   Es gilt: $\sum_{j=2}^n (j-1) = 1 + 2 + \ldots (n-1) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$ und damit
   $T(n) = \frac{D}{2}n^2 + (A - \frac{D}{2})n - B$

3. <u>durchschnittlicher Fall:</u> Angenommen, alle Permutationen von $\langle a_1, \ldots, a_n \rangle$ sind als Eingabe gleich wahrscheinlich.

Dann kann man sich überlegen, daß für den Erwartungswert von $t_j$ gilt $E[t_j] = 1 + (j-1)/2$.
Dann gilt für den Erwartungswert $\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j-1)/2 = \frac{1}{4}n(n-1)$ und damit
$T(n) = \frac{D}{4}n^2 + (A - \frac{D}{4})n - B$

Das bedeutet also, daß INSERTION-SORT für das Sortieren von $n$ Zahlen im besten Fall $An - B$ Zeiteinheiten braucht, im schlechtesten Fall $\frac{D}{2}n^2 + (A - \frac{D}{2})n - B$ Zeiteinheiten, und im "durchschnittlichen Fall" $\frac{D}{4}n^2 + (A - \frac{D}{4})n - B$ Zeiteinheiten.
Diese Information ist unbefriedigend: Einerseits wird durch die Konstanten $A$, $B$ und $D$ sehr viel Detail geboten (und es muß auch viel Aufwand betrieben werden, um diese Konstanten zu bestimmen), andererseits kennt man die Werte dieser Konstanten ja nicht. Daher ist es wünschenswert, die durch die Konstanten gegebene Detailinformation wegzuabstrahieren. Eine extreme, aber immer noch nützliche Art der Abstraktion wäre, sich auf den Grad des Polynoms, das $T(n)$ darstellt, zu beschränken. Demnach wäre die Laufzeit von INSERTION-SORT im besten Fall *linear* in der Größe der Eingabe, und im schlechtesten und im durchschnittlichen Fall *quadratisch.*
Laufzeiten $T(n)$ müssen aber nicht unbedingt immer Polynome in $n$ sein. Um auch kompliziertere Laufzeitfunktionen hinreichend abstrakt behandeln zu können, führen wir die asymptotische Notation ein.

## 1.4 Asymptotische Notation

Die *asymptotische Notation* erlaubt Klassifizierung und Vergleich von Funktionen nach ihrer Zuwachsrate. Hier sind einmal die Definitionen:

1. $O(g(n)) = \{f(n) | \exists c_2 > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c_2 g(n)\}$
2. $\Omega(g(n)) = \{f(n) | \exists c_1 > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c_1 g(n) \geq 0\}$
3. $\Theta(g(n)) = \{f(n) | \exists c_2 \geq c_1 > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} = O(g(n)) \cap \Omega(g(n))$
4. $o(g(n)) = \{f(n) | \forall c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) < cg(n)\}$
5. $\omega(g(n)) = \{f(n) | \forall c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) > cg(n) \geq 0\}$

Der Zweck dieser Notation ist es, Zuwachsraten von Funktionen zu vergleichen: wie schnell wachsen Funktionen, wenn $n$ groß wird. Intuitiv lassen sich obige Definitionen wie folgt verdeutlichen:

- $f \in \Theta(g) \longrightarrow f = g$
- $f \in O(g) \longrightarrow f \leq g$
- $f \in o(g) \longrightarrow f < g$
- $f \in \Omega(g) \longrightarrow f \geq g$
- $f \in \omega(g) \longrightarrow f > g$

Sehr oft verwendet man auch eine etwas schlampigere Notation, in der man zum Beispiel schreibt $f(n) \leq O(g(n))$ oder gar $f(n) = O(g(n))$ statt $f(n) \in O(g(n))$. Das "="-Zeichen ist hier mit besonderer Vorsicht zu genießen, denn $f(n) = O(g(n))$ bedeutet natürlich noch lange nicht, daß $g(n) = O(f(n))$. Asymptotische Notation wird auch of innerhalb von arithmetischen Ausdrücken verwendet. Die genauen Definitionen dafür und eine Diskussion der Probleme, die dabei auftreten können, findet man auf den Seiten 28–30 des Buches von *Cormen, Leiserson* und *Rivest.*

Vorlesung vom 19.10.1995
Jochen Wagner, Holger Bielenstein, Mark Wehrmann

### 1.4.1 Beispiel für die Verwendung der asymptotischen Notation

Sei $f(n) = n^2 + 1000n$.

**Behauptung 1** $f(n) \in O(n^2)$

**Beweis** Gesucht sind $c > 0$ und $n_0 \in \mathbb{N}$, sodaß für alle $n \geq n_0$ gilt: $f(n) \leq cn^2$.

Aus $f(n) = n^2 + 1000n \leq n^2 + 1000n^2 = 1001n^2$ folgt für c = 1001, $n_0 = 1$ die Behauptung. $\square$

**Behauptung 2** $f(n) \in \Omega(n^2)$

**Beweis** Gesucht sind $c > 0$ und $n_0 \in \mathbb{N}$, sodaß für alle $n \geq n_0$ gilt: $f(n) \geq cn^2$.

Aus $f(n) = n^2 + 1000n \geq 1 \cdot n^2$ folgt für $c = 1$, $n_0 = 1$ die Behauptung. $\square$

Aus den beiden obigen Behauptungen folgt: $f(n) = \Theta(n^2)$.

**Behauptung 3** $f(n) \in o(n^2 \log n)$

**Beweis** Zu zeigen ist, daß für alle $c > 0$ ein $n_0 \in \mathbb{N}$ existiert, sodaß für alle $n \geq n_0$ gilt: $f(n) = n^2 + 1000n < cn^2 \log n$.

Setze $n_0 := 2^{1001/c}$. Für $n > n_0$ gilt somit $1001$¡$\log n$ und daher $1001n^2 < cn^2 \log n$. Nun folgt die Behauptung aus $n^2 + 1000n \leq 1001n^2 < cn^2 \log n$. $\square$

### 1.4.2 Eigenschaften von Funktionen in asymptotischer Notation

Von den angegebenen Definitionen für $\Theta$, $O$, $o$, $\Omega$ und $\omega$ lassen sich leicht die folgenden Eigenschaften herleiten (Beweis wird als Übung gelassen).

**Transitivität:**

$$\begin{array}{lcll} f(n) \in O(g(n)) & \text{und} & g(n) \in O(h(n)) & \Rightarrow & f(n) \in O(h(n)) \\ f(n) \in o(g(n)) & \text{und} & g(n) \in o(h(n)) & \Rightarrow & f(n) \in o(h(n)) \\ f(n) \in \Omega(g(n)) & \text{und} & g(n) \in \Omega(h(n)) & \Rightarrow & f(n) \in \Omega(h(n)) \\ f(n) \in \omega(g(n)) & \text{und} & g(n) \in \omega(h(n)) & \Rightarrow & f(n) \in \omega(h(n)) \\ f(n) \in \Theta(g(n)) & \text{und} & g(n) \in \Theta(h(n)) & \Rightarrow & f(n) \in \Theta(h(n)) \end{array}$$

**Reflexivität:**

$$f(n) \in O(f(n)), \quad f(n) \in \Theta(f(n)), \quad f(n) \in \Omega(f(n))$$

**Symmetrie:**

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$$

**Schiefsymmetrie:**

$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

Daraus kann man sehen, daß $\Theta$ eine Äquivalenzrelation auf *schließlich positiven Funtionen* (siehe Übungsblatt 1) definiert, und daß $O$, $o$, $\Omega$ und $\omega$ Ordnungsrelationen auf solchen Funktionen definieren, so wie es $\leq$, $<$, $\geq$ und $>$ auf den reellen Zahlen tun. Allerdings ergibt sich in unserem Fall im Gegensatz zum Fall der reellen Zahlen *keine totale Ordnung.* Dies sieht man an dem folgenden Beispiel.
Sei $f(n) = n$ und $g(n) = n^{1+\cos \pi n}$. Diese Funktionen können in der asymptotischen Notation nicht verglichen werden, denn der Exponent von $g$ bewegt sich periodisch zwischen 2 (für gerade $n$) und 0 (für ungerade $n$).
Folgende Liste ermöglicht einen einfachen Vergleich häufig benutzter Funktionen.
Seien im folgenden $0 < \alpha < \beta, \quad 0 < a < b, \quad 1 < A < B$:

$$\begin{aligned}
\log\log n\\
(\log n)^{\alpha}\\
(\log n)^{\beta}\\
n^{a}\\
n^{a}(\log n)^{\alpha}\\
n^{a}(\log n)^{\beta}\\
n^{b}\\
A^{n}\\
A^{n}n^{a}\\
A^{n}n^{b}\\
B^{n}
\end{aligned}$$

Dabei gilt für zwei (nicht zwangsläufig direkt) untereinander stehende Funktionen $f(n)$ und $g(n)$ die Beziehung $f(n) = o(g(n))$. Zum Beispiel gilt: $n^a = o(n^b)$.

### 1.4.3 Rechenregeln für asymptotisch notierte Funktionen

Für asymptotisch notierte Funktionen gelten folgende Rechenregeln:

1. $\sum_i O(f(i)) = O(\sum_i f(i))$
2. $O(f_1(n)) + O(f_2(n)) + \ldots + O(f_i(n)) = O(\max\{f_1(n), \ldots, f_i(n)\})$

## 1.5 Laufzeitvergleich von Algorithmen

Die asymptotische Notation unterdrückt Konstanten und vergleicht Funktionen für den Fall, daß ihre Argumente sehr groß werden. Asymptotisch gesehen ist also ein Algorithmus, dessen Laufzeit $f(n) = 10^{100}n$ ist, besser als ein Algorithmus für das gleiche Problem, dessen Laufzeit $g(n) = 10n^2$ ist, denn $f(n) \in \Theta(n)$ und $g(n) \in \Theta(n^2)$ und sicher $f(n) \in o(g(n))$. Andererseits gilt aber, daß für alle $n < 10^{99}$ gilt $g(n) < f(n)$, das heißt, daß für alle vernünftigen Eingabegrößen, die man jemals betrachten wird, der zweite, asymptotisch schlechtere Algorithmus eigentlich der schnellere ist.
Nichtsdestotrotz werden wir fast immer Algorithmen nur auf ihre asymptotische Laufzeit hin analysieren und vergleichen. Konstanten so groß wie $10^{100}$ kommen bei tatsächlichen Algorithmen kaum vor (und wenn sie vorkommen sollten, ist wahrscheinlich der Algorithmus so kompliziert, daß man ihn gar nicht implementieren möchte). Wenn die Konstanten aber relativ klein bleiben, dann zeigt das aymptotische Laufzeitverhalten sehr bald das tatsächliche Laufzeitverhalten an, wie man aus den folgenden Tabellen ableiten kann.
Die erste Tabelle zeigt folgendes an: Gegeben sei dazu ein Computer, der $10^6$ Operationen/sec. ausführen kann. Dann gibt jeder Tabelleneintrag für die Laufzeit des Algorithmus in Abhängigkeit von der Problemgröße (Zeile) und einer vorgegeben Rechenzeit (Spalte) die maximale Problemgröße an, die in dieser Rechenzeit bewältigt werden kann.

| Laufzeit | 1 sec | 1 min | 1 Std. | 1 Tag | 1 Jahr |
|---|---|---|---|---|---|
| n | $10^6$ | $6 \cdot 10^7$ | $3,6 \cdot 10^8$ | $8,64 \cdot 10^{10}$ | $3,15 \cdot 10^{13}$ |
| $n \log n$ | 62746 | $\approx 2,8 \cdot 10^6$ | $1,33 \cdot 10^8$ | $2,74 \cdot 10^9$ | $7.97 \cdot 10^{11}$ |
| $n^2$ | 1000 | 7745 | 60000 | 293939 | $5,6 \cdot 10^6$ |
| $n^3$ | 100 | 391 | 1533 | 4420 | 31594 |
| $2^n$ | 20 | 25 | 31 | 36 | 44 |

Beispiel: Ein Sortieralogrithmus mit Laufzeit $n^2$ kann in einer Minute $n = 7745$ Elemente sortieren, während ein entsprechender Alogrithmus mit einer $n \log n$-Laufzeit in der gleichen Zeit ungefähr $2,8 \cdot 10^6$ Elemente sortieren kann. Selbst wenn der zweite Algorithmus Laufzeit, sagen wir, $60 \cdot n \log n$ hätte, könnte er noch immer $10^6$ Elemente sortieren.

Die Art und Weise, in der die Laufzeit eines Alorithmus in Abhängigkeit von der Problemgröße ansteigt, kann sich sehr viel stärker auf die gesamte Laufzeit des Algorithmus auswirken, als es die Leistungsfähigkeit des verwendeten Computers tut. Die folgende Tabelle demonstriert dies, indem sie für mehrere Laufzeitklassen angibt, wie sich für ein fixes Zeitintervall der Länge $T$ die in dieser Zeit verarbeitbare Problemgröße ändert, wenn ein 10 mal schnellerer Computer verwendet wird. Dabei sei $S$ jeweils die ursprüngliche Problemgröße.

| Laufzeitklasse | Ursprüngliche Problemgröße in Zeit $T$ | Problemgröße mit neuem Computer in Zeit $T$ |
|---|---|---|
| $n$ | $S$ | $10S$ |
| $n \log n$ | $S$ | $\approx 10S$ |
| $n^2$ | $S$ | $\approx 3,16S$ |
| $n^3$ | $S$ | $\approx 2,15S$ |
| $2^n$ | $S$ | $S + 3,3$ |

Man beachte vor allem den minimalen Zuwachs im Falle der exponentiellen Funktion $2^n$.

### 1.5.1 Sortieren nach dem Prinzip "Teile und Herrsche" ("Divide and Conquer")

Der in Abschnitt 1.3 analysierte Sortieralgorithmus folgt einem inkrementellen Ansatz: In ein sortiertes Feld A[1..j-1] wird das Element A[j] so eingefügt, daß man ein sortiertes Feld A[1..j] erhält. Seine Laufzeit im schlechtesten Fall war $O(n^2)$.
In diesem Abschnitt wird ein alternativer Sortieralgorithmus ("Merge Sort") angegeben, der eine Laufzeit im schlechtesten Fall von nur $O(n \log n)$ erreicht. Wie viele andere schnelle Algorithmen funktioniert Mergesort nach dem Prinzip "Teile und Herrsche". Algorithmen dieses Typs unterteilen ein gegebenes Problem in mehrere gleichartige Teilprobleme mit jeweils geringerer Größe als das Ausgangsproblem. Diese Teilprobleme werden rekursiv bearbeitet, und anschließend werden die gefundenen Teillösungen zur Gesamtlösung des Ausgangsproblems kombiniert.
Im Falle von Mergesort sieht dies folgendermaßen aus:

- Gegeben sei eine Folge von $n$ zu sortierenden Zahlen.
- Die $n$-elementige Folge wird in zwei Teilfolgen mit je $n/2$ Elementen unterteilt.
- Die beiden Teilfolgen werden rekursiv mittels Merge-Sort sortiert.
- Die so erhaltenen sortierten Teilfolgen werden zu einer sortierten Gesamtfolge der Länge $n$ verschmolzen.

Dieses Prinzip wird in Skizze 1.3 veranschaulicht.

<u>Der Merge-Sort Algorithmus</u>

**procedure** Merge-Sort(A,p,r) (* sortiert das Feld A[p..r] *)
(1) **if** $p < r$ **then**
(2) $\quad q = \lfloor (p+r)/2 \rfloor$
(3) $\quad$ Merge-Sort(A,p,q)
(4) $\quad$ Merge-Sort(A,q+1,r)
(5) $\quad$ Merge (A,p,q,r) (* verschmilzt A[p..q] und A[q+1..r] *)

Der Pseudocode der Verschmelzungsprozedur Merge wird hier nicht aufgeführt. Es ist jedoch leicht vorstellbar (und ist in Übungsblatt 1 zu zeigen), daß er für die Verschmelzung von $n$ Elementen die Zeit $\Theta(n)$ benötigt.

Für die einzelnen Zeilen des Algorithmus ergibt sich folgende Laufzeit:

| Zeilennummer | Laufzeit |
|---|---|
| 1 | $O(1)$ |
| 2 | $O(1)$ |
| 3 | $T(n/2)$ |
| 4 | $T(n/2)$ |
| 5 | $O(n)$ |

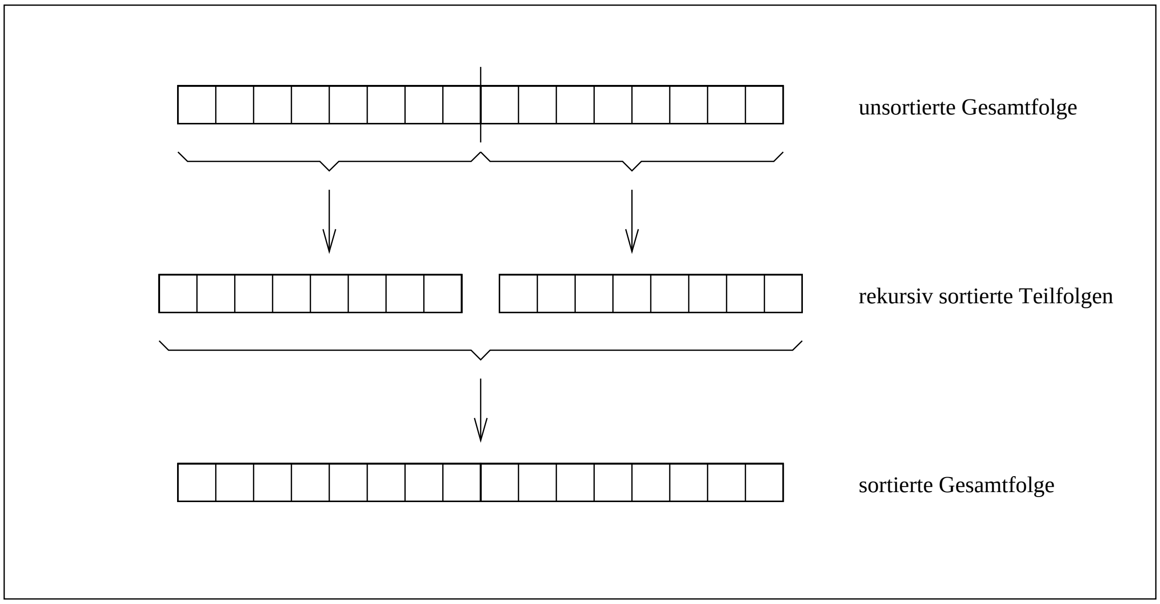



Abbildung 1.3: Arbeitsweise von MERGE-SORT

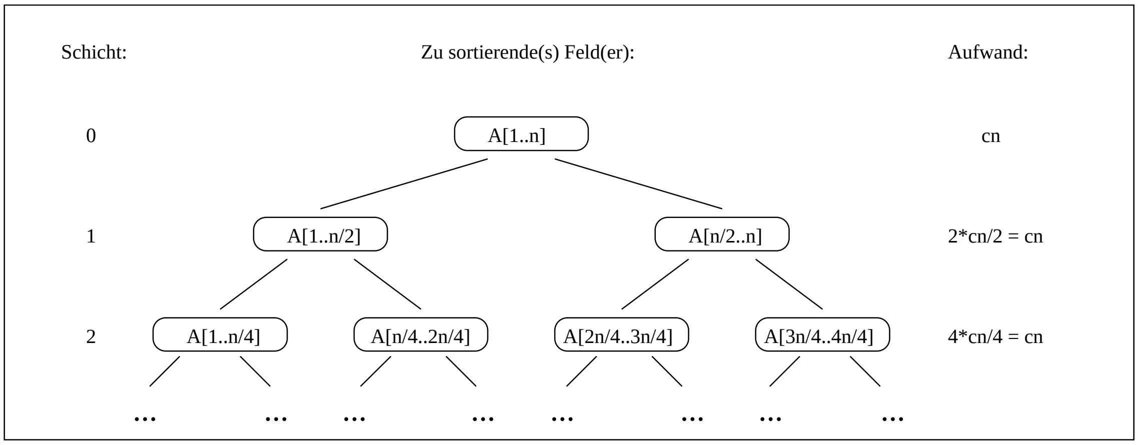



Abbildung 1.4: Laufzeit von MERGE-SORT

Dabei ist $T(n)$ die Laufzeit bei Problemgröße $n$.
Mit Hilfe der Abbildung 1.4 läßt sich die Laufzeit des MERGE-SORT-Algorithmus herleiten.

In jeder Schichte des Baums sind $2^i$ Felder der Größe $n/2^i$ zu mischen (dabei ist $0 \leq i < \log_2(n+1)$ die jeweilige Schichtnummer, mit der Zählung bei der Wurzel beginnend). Der Arbeitsaufwand innerhalb einer Schicht beträgt damit $2^i(n/2^i)c$, also $cn$ (wobei $c$ eine Konstante ist, die die Laufzeit für $n = 2$ angibt), und ist also linear von der gesamten ursprünglichen Feldgröße abhängig.
Der Baum hat außer der Wurzel höchstens $\log_2(n+1)$ Schichten. Für den gesamten Baum ergibt sich damit ein Aufwand von $cn \log_2 n$.

Formal: Ist $T(n)$ die Laufzeit bei Problemgröße $n$, so gilt:

$$T(n) \leq \begin{cases} c & \text{für} \quad n = 2 \\ cn + 2T(\frac{n}{2}) & \text{für} \quad n > 2. \end{cases}$$

Induktiv kann man $T(n) \leq cn \log_2 n$ zeigen.

## 1.6 Generalisierung der Teile- und Herrsche Formel

Allgemein treten rekursive Laufzeitabschätzungen in folgender Form auf:

$$T_f(n) = \begin{cases} D \cdot f(n) & n \leq N \\ f(n) + \sum_i T_f(n_i) & n > N\,. \end{cases}$$

Dabei sind $D \geq 1$ und $N > 0$ Konstante und $n_i$ irgendwelche positiven Zahlen.

**Lemma 1** *Unter den Annahmen, daß*

1. *es eine Konstante $b > 1$ gibt, sodaß für alle $n > N$ gilt $n_i \leq n/b$ für alle $i$, und*
2. *es eine Konstante $c > 0$ gibt, sodaß $\forall n > N$ gilt $\sum_i f(n_i) \leq cf(n)$,*

*folgt*

$$T_f(n) = O\Big(f(n) \sum_{j=0}^{\log_b n} c^j\Big)\,.$$

*Dies bedeutet, daß*

$$T_f(n) = \begin{cases} O(f(n)) & \textit{für} \quad c < 1 \\ O(f(n)\log n) & \textit{für} \quad c = 1 \\ O(f(n)\, n^{\log_b c}) & \textit{für} \quad c > 1\,. \end{cases}$$

**Beweis** Es reicht, wenn wir durch Induktion nach $n$ das folgende zeigen:

$$T_f(n) \leq D \cdot f(n) \cdot \sum_{j=0}^{\log_b n} c^j\,.$$

$0 < n \leq N:$ Nach Definition ist in diesem Fall $T_f(n) = D \cdot f(n)$. Die angebene Schranke ist dann

$$D \cdot f(n) \cdot \sum_{j=0}^{\overbrace{\log_b n}^{\geq 0}} c^j \geq D \cdot f(n)\,.$$

$n > N$ :

$$
\begin{aligned}
T_f(n) &= f(n) + \sum_i T(n_i) \\
&\le f(n) + \sum_i D \cdot f(n_i) \cdot \sum_{j=0}^{\log_b n_i} c^j && \text{nach Induktions-Voraussetzung} \\
&\le f(n) + D \cdot \sum_i f(n_i) \cdot \sum_{j=0}^{(\log_b n)-1} c^j && \text{wegen } n_i \le n/b \\
&\le D \cdot f(n) + D \cdot \sum_i f(n_i) \cdot \sum_{j=0}^{(\log_b n)-1} c^j && \text{weil } D \ge 1 \text{ nach Definition} \\
&\le D \cdot f(n) + D \cdot c f(n) \cdot \sum_{j=0}^{(\log_b n)-1} c^j && \text{nach Annahme 2.} \\
&= D \cdot f(n) \cdot \sum_{j=0}^{\log_b n} c^j
\end{aligned}
$$

Man kann nun $c$ bzgl. 1 diskriminiert und bekommt die folgenden drei Fälle für die Summe $\sum_{j=0}^{\log_b n} c^j$: Wenn $c < 1$ ist, dann ist diese Summe kleiner als $1/(1-c)$, da sie ohne obere Schranke für $j$ gegen diesen Wert konvergiert. Die Summe ist dann also $O(1)$. Für $c = 1$ ist jeder der $1 + \lceil \log_b n \rceil$ Summanden 1, also die gesamte Summe $O(\log n)$. Wenn $c > 1$ ist, dann gilt $\sum_{0 \le j \le m} c^j = (c^{m+1} - 1)/(c-1) = O(c^m)$. Das heißt, in diesem Fall ist die Summe $O(c^{\log_b n}) = O(b^{(\log_b c)\cdot(\log_b n)}) = O(n^{\log_b c})$.
Daraus folgt der Rest des Lemmas. $\square$

Vorlesung vom 23.10.1995
Ulrich Hoffmann, Hans-Georg Dörr, Frank Oster

Dieses Lemma sieht zwar schön aus, ist aber nicht unbedingt so brauchbar wie es sein sollte. Das zeigt sich an folgendem Beispiel:
Sei $f(n) = n^2$ und seien $n_1 = n_2 = n_3 = n_4 = \lceil n/2 \rceil$. Als Konstante $b$ im Lemma würde man dann wohl gerne $b = 2$ wählen, aber genau genommen ist das nicht zulässig, da für ungerade $n$ gilt $n_i = \lceil n/2 \rceil \not\le n/2$. Man müßte $b$ etwas kleiner wählen, aber das könnte die für $T_f(n)$ gefundene Schranke verschlechtern, da $\log_b c$ mit fallendem $b$ steigt.
Wie sieht es mit der Konstanten $c$ aus? $\sum_i f(n_i)$ wäre $4 \cdot \lceil n/2 \rceil^2$, was für ungerade $n$ zu $4 \cdot (n/2 + 1/2)^2 = (n+1)^2 \not\le 1 \cdot n^2$ käme. Das heißt, man müßte, genau genommen, eine Konstante $c > 1$ wählen, was automatisch die im Lemma angegebene Schranke für $T_f(n)$ von $O(f(n)\log n)$ auf $O(f(n)\, n^{\log_b c})$ verschlechtert.
Um diese Probleme zu umgehen, wollen wir die Annahmen des Lemmas ein bißchen abschwächen. Erstens werden wir verlangen, daß nur $n_i \le n/b + d$ gelten muß ($d \ge 0$ eine Konstante). Für unser Beispiel würde das reichen, da ja $n_i = \lceil n/2 \rceil \le n/2 + 1$ gilt. Man hat also $b = 2$ und $d = 1$.
Zweitens werden wir verlangen, daß nur $\sum_i f(n_i) \le cf(n) + g(n)$ notwendig ist, wobei der "Korrekturfaktor" $g(n)$ langsamer wachsen muß als $f(n)$ (und auch einer kleinen technischen Bedingung genügen muß). Für unser Beispiel könnte man $g(n) = 2n + 1$ verwenden, denn dann gälte $\sum_i f(n_i) = (n+1)^2 \le 1 \cdot f(n) + g(n)$.
Wie sich herausstellen wird, gelten die asymptotischen Behauptungen des Lemmas auch unter diesen schwächeren Annahmen.

### 1.6.1 Erste Verbesserung:

**Behauptung 4** *Die Aussagen des vorherigen Lemmas bleiben bestehen, wenn man Annahme 1 folgendermaßen ändert:*

1. *"Es gibt Konstanten $b > 1$ und $d \ge 0$, sodaß für alle $n > N$ und für all $i$ gilt $n_i \le \min\{n/b + d, n - 1\}$."*

**Beweis** Für feste vorgegeben $b$,$d$ und $n$ sei $N_0, N_1, \ldots$ eine Folge von natürlichen Zahlen mit den Eigenschaften $N_0 = n$ und $N_{k+1} \leq \min\{N_k/b + d, N_k - 1\}$. Weiterhin sei $L_{b,d}(n)$ das kleinstmögliche $k$, sodaß für so eine Folge $N_k \leq 1$ gilt.
Mit der neuen Annahme kann der induktive Beweis des vorherigen Lemmas beinahe wörtlich wiederholt werden, um zu zeigen, daß jetzt

$$T_f(n) \leq D \cdot f(n) \cdot \sum_{j=0}^{L_{b,d}(n)} c^j .$$

gilt. Es hat sich also nur die obere Schranke für den Summationsindex verändert. Wir brauchen jetzt nur mehr zu zeigen, daß es eine von $n$ unabhängige Konstante $A$ gibt, sodaß für all $n$ gilt $L_{b,d}(n) \leq A + \log_b n$. Denn dann haben wir

$$\sum_{j=0}^{L_{b,d}(n)} c^j \leq \sum_{j=0}^{A+\log_b n} c^j \leq \sum_{j=0}^{A} c^j \cdot \sum_{j=0}^{\log_b n} c^j = O\Big(\sum_{j=0}^{\log_b n} c^j\Big) .$$

Die folgende Rechnung zeigt, daß man $A = \left\lceil \frac{d+2}{1-1/b} \right\rceil$ verwenden kann. Die Folge $N_0, N_1, \ldots$ wird am längsten, wenn $N_{i+1} = N_i/b + d$ gesetzt wird, bis ein $k$ erreicht wird, sodaß $N_k - 1 \leq N_k/b + d$ gilt. Von da an muß $N_{i+1} = N_i - 1$ gesetzt werden. Das heißt also, die Länge unserer Folge ist höchstens $k + \lceil N_k \rceil$. Die Bedingung $N_k - 1 \leq N_k/b + d$ bedeutet, daß $N_k \leq \frac{d+1}{1-1/b}$ ist. Also hat unsere Folge höchstens Länge $k + \left\lceil \frac{d+1}{1-1/b} \right\rceil$. Wie groß ist nun $k$? Es gilt:

$$\begin{aligned}
N_0 &= n \\
N_1 = N_0/b + d &= n/b + d \\
N_2 = N_1/b + d &= (1/b)^2 n + (1/b)d + d \\
N_3 = N_2/b + d &= (1/b)^3 n + (1/b)^2 d + (1/b)d + d
\end{aligned}$$

und im allgemeinen

$$N_i = N_{i-1}/b + d = (1/b)^i n + d \sum_{0 \leq j < i} (1/b)^j = (1/b)^i n + d \frac{1-(1/b)^i}{1-1/b} \leq (1/b)^i n + \frac{d}{1-1/b} .$$

Wenn man $k = \lceil \log_b n \rceil$ setzt, ergibt sich

$$N_k \leq \underbrace{(1/b)^{\lceil \log_b n \rceil} n}_{\leq 1} + \frac{d}{1-1/b} \leq \frac{d+1}{1-1/b}$$

und das bedeutet, die Länge unserer Folge ist höchstens

$$\lceil \log_b n \rceil + \left\lceil \frac{d+1}{1-1/b} \right\rceil \leq \log_b n + \left\lceil \frac{d+2}{1-1/b} \right\rceil .$$

$\Box$

### 1.6.2 Zweite Verbesserung:

**Behauptung 5** *Die Aussagen des vorherigen Lemmas bleiben bestehen, wenn man Annahme 2 folgendermaßen ändert:*

2. *"Es gibt eine Konstanten $c > 0$, sodaß $\forall n > N$ gilt $\sum_i f(n_i) \leq cf(n) + g(n)$, wobei $g(n) = o(f(n))$ und es gibt eine Konstant $c' > c$ sodaß $\forall n > N$ gilt $\sum_i g(n_i) \geq c' g(n)$."*

**Beweis** Die Beweisidee ist sehr einfach: Wir definieren eine Funktion $h(n)$ mit $h(n) = O(f(n))$ und $f(n) = O(h(n))$, die der ursprünglichen Annahme 2 des Lemmas genügt. Dann haben wir

$$T_h(n) \leq O\left(h(n) \sum_{j=0}^{\log_b n} c^j\right) = O\left(f(n) \sum_{j=0}^{\log_b n} c^j\right) .$$

Da weiterhin aus $f(n) = O(h(n))$ folgt, daß $T_f(n) = O(T_h(n))$ (siehe die folgenden Übungen), erreichen wir das gewünschte

$$T_f(n) = O(T_h(n)) = O\Big(f(n) \sum_{j=0}^{\log_b n} c^j\Big).$$

Wir wählen $h(n) = 2f(n) - \frac{2}{c'-c}g(n)$. Klarerweise gilt dann $h(n) \leq 2f(n)$, also $h(n) = O(f(n))$, und, da $g(n) = o(f(n))$, gilt für genügend große $n$, daß $h(n) > f(n)$, also $f(n) = O(h(n))$. Es bleibt nur zu zeigen, daß $h$ der ursprünglichen Bedingung 2 genügt, daß $\sum_i h(n_i) \leq c \cdot h(n)$.

$$\begin{aligned}
\sum_i h(n_i) = \sum_i \Big(2f(n_i) - \frac{2}{c'-c}g(n_i)\Big) &= \sum_i 2f(n_i) - \sum_i \frac{2}{c'-c}g(n_i) \\
&\leq \Big(2cf(n) + 2g(n)\Big) - \frac{2c'}{c'-c}g(n) \\
&= 2cf(n) - \frac{2c}{c'-c}g(n) \\
&= c \cdot h(n)
\end{aligned}$$

□

**Übung:** Wir wollen uns das angebene Rekursionsschema etwas näher ansehen. Wenn man es sehr genau nimmt, müßte man es für eine Funktion $f$ und Zahlen $N$ und $D$ so schreiben und definieren:

$$T_{f;N,D}(n) = \begin{cases} D \cdot f(n) & n \leq N \\ f(n) + \sum_i T_{f;N,D}(n_i) & n > N\,, \end{cases}$$

nachdem für jedes $n \in \mathbb{N}$ eine endliche Folge $n_1, n_2, \ldots$ mit $n_i < n$ festgelegt wurde.

1. Zeigen Sie, daß es bei vorgegebenem positiven $f$ für jedes $N'$ ein $D'$ gibt, sodaß $T_{f;N,D}(n) = O(T_{f;N',D'}(n))$. (Dies zeigt, daß es Sinn macht, $N$ und $D$ nicht in der Notation von $T_f$ mitzuführen.)
2. Wenn $N$ und $D$ fixiert sind, dann ist $T_{f;N,D}$ ein Operator, der einer Funktion $f$ die Funktion $T_{f;N,D}(n)$ zuordnet. Zeigen Sie, daß dies ein linearer Operator ist.
3. Zeigen Sie, daß aus $f(n) = O(h(n))$ folgt, daß $T_{f;N,D}(n) = O(T_{h;N,D}(n))$.

Wir wollen nun unser verbessertes vorheriges Lemma vollständig als Satz aufschreiben:

**Satz 1** *Für Funktionen $f : \mathbb{N} \to \mathbb{R}^+$ und fixe Konstanten $N > 0$ und $D > 0$ sei $T_f$ definiert als*

$$T_f(n) = \begin{cases} D \cdot f(n) & : \quad n < N \\ f(n) + \sum_i T_f(n_i) & : \quad n \geq N\,, \end{cases}$$

*wobei für jedes $n \geq N$ eine fixe endliche Folge $n_1, n_2, \ldots$ vorgegeben ist.*
*Nehmen wir folgende Voraussetzungen an:*

1. *Es gibt Konstanten $b > 1$ und $d \geq 0$, sodaß für alle $n > N$ und für all $i$ gilt $n_i \leq \min\{n/b + d, n-1\}$.*
2. *Es gibt eine Konstanten $c > 0$, sodaß $\forall n > N$ gilt $\sum_i f(n_i) \leq cf(n) + g(n)$, wobei $g(n) = o(f(n))$ und es gibt eine Konstant $c' > c$ sodaß $\forall n > N$ gilt $\sum_i g(n_i) \geq c' g(n)$.*

*Dann gilt*

$$T_f(n) = O\Big(f(n) \sum_{j=0}^{\log_b n} c^j\Big)$$

*und insbesonders gilt*

$$T_f(n) = \begin{cases} O(f(n)) & \textit{für} \quad c < 1 \\ O(f(n)\log n) & \textit{für} \quad c = 1 \\ O(f(n)\, n^{\log_b c}) & \textit{für} \quad c > 1\,. \end{cases}$$

Die weitaus wichtigste Anwendung dieses Satzes ist der Fall, wenn $f$ ein Polynom ist, und die $n_i$'s alle ungefähr $n/b$. Da ergibt sich das folgende:

**Korollar 1 "Master Theorem for Recurrences"**[2]
*Seien $a \in \mathbb{Z}^+$ und $b > 1$ Konstanten, sei $k > 0$ und für $N, D > 0$ sei $T$ definiert als*

$$T(n) = \begin{cases} D \cdot n^k & : \quad n < N \\ a \cdot T(\lceil \frac{n}{b} \rceil) + n^k & : \quad n \geq N \end{cases}$$

*Mit $\lambda = \log_b a$ gilt dann*

$$T(n) = \begin{cases} O(n^k) & \textit{für} \quad \lambda < k \\ O(n^k \log n) & \textit{für} \quad \lambda = k \\ O(n^\lambda) & \textit{für} \quad \lambda > k\,. \end{cases}$$

**Beweis** Für $n > N$ gilt hier $n_i = \lceil n/b \rceil$, also $n_i \leq n/b + 1$ für $1 \leq i \leq a$, und damit ist Bedingung 1 von unserem Satz erfüllt.
Bzgl. Bedingung 2 habe wir

$$\sum_i f(n_i) = a \cdot \lceil n/b \rceil^k \leq a \cdot (n/b + 1)^k = \underbrace{(a/b^k)}_{=c} n^k + O(n^{k-1}) = cf(n) + O(n^{k-1})\,.$$

Das heißt, man kann $g(n) = En^{k-1}$ verwenden, wobei $E > 0$ irgendeine Konstante ist. Dann gilt aber

$$\sum_i g(n_i) = a \cdot E \lceil n/b \rceil^{k-1} \geq a \cdot E \cdot (n/b)^{k-1} = E \cdot \underbrace{(a/b^{k-1})}_{=c'} n^{k-1} = c' g(n)$$

mit $c' > c$. Also ist Bedingung 2 auch erfüllt.
Da $c = a/b^k$ haben wir $\lambda = \log_b a \overset{<}{\underset{>}{=}} k$ genau dann, wenn $c \overset{<}{\underset{>}{=}} 1$. Also gilt nach dem Satz

$$T(n) = \begin{cases} O(n^k) & \text{für} \quad \lambda < k \quad (c < 1) \\ O(n^k \log n) & \text{für} \quad \lambda = k \quad (c = 1) \\ O(n^k\, n^{\lambda - k}) = O(n^\lambda) & \text{für} \quad \lambda > k \quad (c > 1). \end{cases}$$

□

Vorlesung vom 26.10.1995
Christian Zenner, Tobias Miller

## 1.7 Beispiel: Arithmetik mit großen Zahlen

Sei $\beta$ eine beliebige Zahlenbasis (typischerweise 10). Eine sehr große Zahl wird dargestellt als:

| Vorzeichen | Array $A$ of $[0..n-1]$ |
|---|---|
| $+/-$ | 009312 |

bzw. $\sum_{i=0}^{n-1} A[i]\beta^i$.

Man soll sich also das Feld $A[]$ von rechts nach links geschrieben vorstellen. Wir wollen annehmen, daß Operationen auf Ziffern $O(1)$, also konstante Zeit, brauchen.

### 1.7.1 Addition

Gegeben seien zwei Zahlen $A$ und $B$ der Länge $n$, und wir wollen deren Summe berechnen. Um die Situation hier so einfach wie möglich zu halten, lassen wir Vorzeichen und führende Nullen unberücksichtigt, das heißt, wir betrachten nur die Addition von nicht-negativen Zahlen. Der gängige Algorithmus dafür lautet so:

```
carry ← 0
for i ← 0 to n − 1
    sum ← carry + A[i] + B[i]
    S[i] ← sum mod β
    carry ← sum/β
if carry ≠ 0 then
    S[n] ← carry
```

[2]siehe Cormen, Leiserson, Rivest : **Introduction to Algorithms** Seite 64

Die folgende Tabelle gibt die Laufzeiten der einzelnen Zeilen an:

<table>
<tr><th>Zeilennr.</th><th>Laufzeit</th><th>Anzahl Aufrufe</th></tr>
<tr><td>1</td><td rowspan="7">$O(1)$</td><td>1</td></tr>
<tr><td>2</td><td>$n+1$</td></tr>
<tr><td>3</td><td rowspan="3">$n$</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td rowspan="2">1</td></tr>
<tr><td>7</td></tr>
</table>

Das ergibt eine Gesamtlaufzeit von $O(n)$, d.h. bei der Addition reicht bereits ein einfacher Algorithmus, um lineare Laufzeit zu erreichen. Schwieriger wird es beim Multiplizieren.

### 1.7.2 Multiplikation:

Auch hier seien wieder zwei Zahlen $A$ und $B$ der Länge $n$ gegeben, und man möchte deren Produkt berechnen. Der übliche ("naive") Algorithmus lautet so:

(1) $prod \leftarrow 0$
(2) **for** $i \leftarrow 0$ **to** $n-1$
(3) $\quad S \leftarrow \text{PRODUKTMITZIFFER}(A, B[i])$ (* $S = A \cdot B[i]$ *)
(4) $\quad T \leftarrow \text{PRODUKTMITBASISPOTENZ}(S, i)$ (* $T = S \cdot \beta^i$ *)
(5) $\quad prod \leftarrow \text{SUMME}(prod, T)$

Die Hilfsfunktionen PRODUKTMITZIFFER, PRODUKTMITBASISPOTENZ und SUMME sind leicht zu implementieren (bei SUMME haben wir es ja schon gemacht) sodaß sie lineare Laufzeit haben. Man sollte sich allerdings überlegen, daß die Länge der Argumente wirklich nur $O(n)$ ist. Da das Produkt insgesamt höchstens Länge $2n$ hat, ist dies auch der Fall.
Die Laufzeit liefert wieder folgende Tabelle:

<table>
<tr><th>Zeilennr.</th><th>Laufzeit</th><th>Anzahl Aufrufe</th></tr>
<tr><td>1</td><td rowspan="2">$O(1)$</td><td>1</td></tr>
<tr><td>2</td><td>$n+1$</td></tr>
<tr><td>3</td><td rowspan="3">$O(n)$</td><td rowspan="3">$n$</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
</table>

Die Gesamtlaufzeit beläuft sich somit auf $O(n^2)$, was für große $n$ enttäuschend langsam ist.
Wir suchen nun einen <u>schnelleren Algorithmus</u> (nach dem Prinzip Teile und Herrsche):

| $A = a\beta^k + a'$ | | | $B = b\beta^k + b'$ | |
|---|---|---|---|---|
| $a$ | $a'$ | | $b$ | $b'$ |
| | $k = \lfloor \frac{n}{2} \rfloor$ | | | $k = \lfloor \frac{n}{2} \rfloor$ |

Nach dem Teile und Herrsche Prinzip werden die Ziffernarrays $A$ und $B$ in der Mitte halbiert, sodaß die entsprechenden Zahlen dargestellt sind als $A = a\beta^k + a'$ und $B = b\beta^k + b'$, wobei $k = \lfloor \frac{n}{2} \rfloor$. Das heißt $a'[0..k-1] = A[0..k-1]$ und $a[0..n-k-1] = A[k..n-1]$ und das Analoge gilt für $B$. Das Produkt zwischen $A$ und $B$ lautet jetzt

$$AB = (ab)\beta^{2k} + (ab' + a'b)\beta^k + a'b' .$$

Das heißt, wir haben das Produkt zweier Zahlen der Länge $n$ als 4 Produkte von Zahlen der Länge $n/2$ plus einigen Additionen und zwei Produkten mit Basispotenzen dargestellt.
Folgender Algorithmus berechnet das Produkt nach dieser Gleichung:

```
      procedure RecProd (A,B) (* A und B haben gleiche Länge n *)
(0)      if n ≤ 3 then benutze normalen Algorithmus und return
(1)      Erzeuge a, a' aus A, b, b' aus B    (* a, a', b, b' haben höchstens Länge ⌈n/2⌉ *)
(2)      p1 ← RecProd(a, b)
         p2 ← RecProd(a', b')
         p3 ← RecProd(a, b')
         p4 ← RecProd(a', b)
(3)      return (p1 * β^(2k) + (p3 + p4) * β^k + p2)
```

Genaugenommen sollte der Ausdruck in der letzten Zeile als

```
SUMME( PRODUKTMITBASISPOTENZ(p1, 2k),
       SUMME( PRODUKTMITBASISPOTENZ( SUMME(p3, p4), k ),
              p2 ) )
```

geschrieben werden, was wir zwecks bessere Lesbarkeit unterlassen haben und im folgenden auch unterlassen werden. Die Multiplikation $*$ bedeutet hier die Multiplikation mit der Basispotenz, was dem Anhängen von Nullen entspricht und linare Laufzeit benötigt.
Die Laufzeittabelle ergibt:

| Zeilennr. | Laufzeit |
|---|---|
| 0 | $O(1)$ |
| 1 | $O(n)$ |
| 2 | $4T(n/2)$ |
| 3 | $O(n)$ |

Daraus ergibt sich:

$$T(n) = O(n) + 4T(n/2) = n^1 + 4T(n/2).$$

In unsere Teile und Herrsche Formel eingesetzt ergibt sich $\lambda = \log_b a = \log_2 4 = 2 > 1$. Daraus folgt

$$T(n) = O(n^2).$$

Es hat sich also keine Verbesserung gegenüber unserem naiven Algorithmus ergeben! Der Grund liegt in den 4 Multiplikationen. Um uns zu verbessern benötigen wir somit einen Algorithmus mit nur 3 Multiplikationen. Die Möglichkeit, sich eine Multiplikation zu sparen, bietet sich bei der Summe $(ab' + a'b)$ an. Diese Summe benötigt zwei Multiplikationen. Durch die Umformung $(ab'+a'b)+ab+a'b' = (a+a')(b+b')$ läßt sich $(ab'+a'b)$ auch als $(a + a')(b + b') - ab - a'b'$ darstellen. Somit ist nur noch eine Multiplikation für diese Summe, also 3 Multiplikationen insgesamt nötig. Die Prozedur ändert sich ab der zweiten Zeile wie folgt:

```
(2)   p1 ← RecProd(a, b)
      p2 ← RecProd(a', b')
      p ← RecProd(a + a', b + b')
(3)   return (p1 * β^(2k) + (p − p1 − p2) * β^k + p2)
```

Daraus ergibt sich eine neue Laufzeit $T(n) = O(n) + 3T(\frac{n}{2})$. Wieder in die Teile und Herrsche Formel eingesetzt ergibt sich $\lambda = \log_b a = \log_2 3 \approx 1,59 > 1$, und damit:

$$T(n) = O(n^{1,59}).$$

Dies ist eine deutliche asymptotische Verbesserung, die sich ab gewissem, großem $n$ sicherlich bemerkbar macht. Dennoch ist auch diese Version der Multiplikation noch sehr schlecht. Der bestmögliche bekannte Algorithmus für die Multiplikation braucht $O(n \log n \log \log n)$ Zeit[3]. Einige Forscher vermuten sogar, es wäre in der Zeit $O(n)$ möglich, aber ein solcher Algorithmus wurde bisher nicht gefunden.
Das letzte Beispiel der Multiplikation zeigt sehr deutlich die quantitative Auswirkung des "Master Theorem for Recurrences", dessen Sinn intuitiv sicher sehr einleuchtend ist: Man versuche, so wenig Teilprobleme wie möglich zu erhalten, und man versuche, diese Teilprobleme so klein wie möglich zu halten.

[3]siehe A. Schönhage und V. Strassen "Schnelle Multiplikation großer Zahlen" in *Computing* 7 (1971) 281–292, oder auch Kapitel 7 des angegenben Buches von Aho, Hopcroft, Ullman.

# Kapitel 2

# Sortieren und Auswählen

In den Abschnitten 1.3 und 1.5.1 haben wir den INSERTION-SORT und den MERGE-SORT Algorithmus betrachtet. Im schlechtesten Fall benötigt

- INSERTION-SORT $O(n^2)$ Laufzeit und nur $O(1)$ zusätzlichen Speicher,
- MERGE-SORT $O(n \log n)$ Laufzeit und $O(n)$ zusätzlichen Speicher.

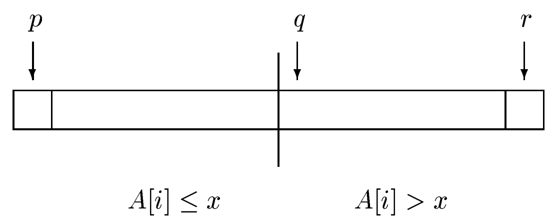
Gesucht wird ein Sortieralgorithmus mit Laufzeit $O(n \log n)$, der nur $O(1)$ zusätzlichen Speicher benötigt. Wir werden dazu wieder das Paradigma von “Teile und Herrsche” verwenden.

## 2.1 Sortieren mit SPLIT-SORT

Idee: SPLIT-SORT zerlegt das zu sortierende Array $A$ in zwei Teilarrays und sortiert diese rekursiv. Im Gegensatz zu MERGE-SORT wird das Array $A$ allerdings“intelligent” zerlegt. Das Verschmelzen der Teilergebnisse kann dann einfacher und ohne zusätzlichen Speicherbedarf ausgeführt werden.

Die folgende Definition wird sich als nützlich erweisen. Sei $S$ eine Folge von $n$ Zahlen. Für $x \in S$ bezeichnen wir als $\mathit{Rang}_S(x)$ ein $j$ mit $1 \leq j \leq n$, sodaß man $S \setminus \{x\}$ in $S'$ und $S''$ partitionieren kann, mit $|S'| = j - 1$, $|S''| = n - j$, und für all $y \in S'$ gilt $y \leq x$ und für alle $z \in S''$ gilt $x \leq z$. Anders gesagt: man könnte $S$ in $x_1 \leq x_2 \leq \cdots \leq x_n$ umordnen, sodaß $x = x_j$ gilt.
Man beachte, daß $\mathit{Rang}_S(x)$ nur eindeutig ist, wenn keine zwei Zahlen in $S$ gleich sind.[2mm]
**Wir werden von nun an annehmen, daß wir nur Folgen zu sortieren (oder später von ihnen zu selektieren) haben, in denen alle Zahlen verschieden sind. In manchem von dem, was kommt, wird diese Annahme vorerst wesentlich sein. Wir lassen es dem Leser als sehr wichtige Übung, sich zu überlegen, wie man diese Verschiedenheitsannahme los werden kann.**[2mm]
Um den neuen Sortieralgorithmus aufzuschreiben, nehmen wir nun an, daß wir die folgenden zwei Funktionen zur Verfügung haben, deren Implementierung (mit den angegebenen Zeitschranken) wir noch hinausschieben.

- Gegeben ist eine Funktion MEDIAN$(A, p, r)$, die ein $x \in A[p..r]$ erzeugt mit $\mathit{Rang}_{A[p..r]}(x) = \lfloor (r - p + 1)/2 \rfloor$. Für die Hälfte der $A[i]$, $p \leq i \leq r$, gilt also $A[i] \leq x$ und für die andere Hälfte $A[i] > x$. Die Funktion MEDIAN$(A, p, r)$ benötigt $O(r - p)$ Laufzeit und nur geringen zusätzlichen Speicher.
- Gegeben ist eine Funktion PARTITION$(A, p, r, x)$, die einen Wert $q$ erzeugt und das Array $A[p..r]$ umordnet, so daß für alle $i$ mit $p \leq i < q$ gilt $A[i] \leq x$ und für alle $i$ mit $q \leq i < r$ gilt $A[i] > x$. Die Funktion PARTITION$(A, p, r, x)$ benötigt $O(r - p)$ Laufzeit und $O(1)$ zusätzlichen Speicher.



SPLIT-SORT berechnet zunächst den Wert $x = \text{MEDIAN}(A, p, r)$. Anschließend teilt die Funktion $\text{PARTITION}(A, p, r, x)$ das Array $A[p..r]$ in die Arrays $A[p..q-1]$ und $A[q..r]$. Diese werden rekursiv sortiert. Im Gegensatz zu MERGE-SORT müssen die Teillösungen nicht extra zusammengefügt werden.

Der SPLIT-SORT Algorithmus:

```
    procedure SPLIT-SORT(A, p, r)
    (* Die zu sortierende Folge liegt im Array A[p..r]*)
(1)     if (r − p) < 3 then INSERTION-SORT(A, p, r)
(2)                         return
(3)     x ← MEDIAN(A, p, r)
(4)     q ← PARTITION(A, p, r, x)
(5)     SPLIT-SORT(A, p, q − 1)
(6)     SPLIT-SORT(A, q, r)
```

Nehmen wir für die Laufzeitanalyse an, daß das zu sortierende Feld die Länge $n$ hat. Die Funktion $T$ bezeichnet den Laufzeitbedarf der Prozedur SPLIT-SORT. Es gilt offensichtlich:

| Zeilennr. | Anzahl Aufrufe | Laufzeit |
|---|---|---|
| 1 | | $O(1)$ |
| 2 | | |
| 3 | 1 | $O(n)$ |
| 4 | | $O(n)$ |
| 5 | | $T(n/2)$ |
| 6 | | |

Entsprechend der Rekursionsformel des Teile und Herrsche Lemmas ergibt sich: $T(n) = O(n \cdot \log n)$

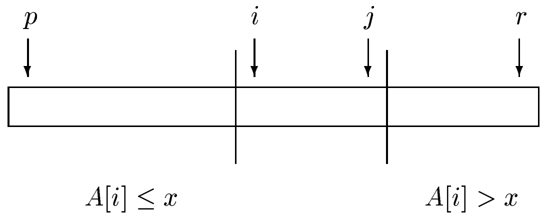
Um den Sortieralgorithmus SPLIT-SORT implementieren zu können müssen noch einige Hilfsfunktionen erstellt werden. Insbesondere müssen die Funktionen PARTITION und MEDIAN angegeben und auf Laufzeit und Speicherbedarf überprüft werden.

Die Funktion PARTITION:

```
function PARTITION(A, p, r, x)
(* PARTITION erzeugt ein q und ordnet das Array A[p..r] so um,
daß A[i] ≤ x für p ≤ i < q
und A[i] > x für q ≤ i ≤ r gilt. *)
    i ← p
    j ← r
    repeat
        while i ≤ r and A[i] ≤ x do i = i + 1
        while j ≥ p and A[j] > x do j = j − 1
        if i < j then SWAP(A[i],A[j])
            (* SWAP vertauscht den Inhalt der beiden Argumente *)
            else return j + 1
```

Die Variablen $i$ und $j$ bezeichnen Zellen in $A$, wobei $i$ Werte aus dem Intervall $[p, r+1]$ und $j$ Werte aus dem Intervall $[p-1, r]$ annehmen kann. Jede der While-Schleifen kann also höchstens $r-p+2$ mal ausgeführt werden. Für die Repeat-Schleife gilt folgende Invariante, deren Aufrechterhaltung man leicht überprüfen kann, und aus der bei Terminierung die Korrektheit dieses Programms folgt:

$$\text{für alle } k \text{ mit } p \leq k < i \text{ gilt } A[k] \leq x,$$
$$\text{für alle } k \text{ mit } j < k \leq r \text{ gilt } A[k] > x.$$




Die Repeat-Schleife und somit die 6. und 7. Zeile können offensichtlich höchstens $O(r-p)$ mal ausgeführt werden. Es ergibt sich also:

| Zeilennr. | Anzahl Aufrufe | Laufzeit |
|---|---|---|
| 1 | 1 | O(1) |
| 2 | | |
| 3 | $O(r-p)$ | $O(1)$ |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

Für die Laufzeit der Funktion PARTITION ergibt sich also im schlimmsten Fall: $T(r-p+1) = O(r-p)$. Offensichtlich benötigt die Funktion PARTITION nur $O(1)$ zusätzlichen Speicher.

Die Funktion MEDIAN

Die Funktion MEDIAN zu implementieren, ist um einiges schwieriger. Wir tun dies über den Aufruf SELECT($\lfloor (r-p+1)/2 \rfloor, A, p, r$). Der Prozedur SELECT widmen wir ein eigenes Teilkapitel.

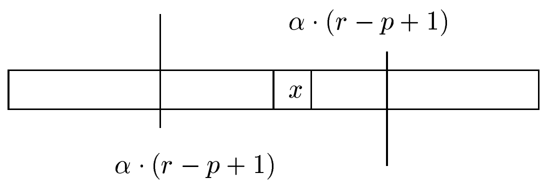
## 2.2 Selektion

Die Funktion SELECT$(k, A, p, r)$ soll aus dem Feld $A[p..r]$ das "$k$-kleinste" Element auswählen, also ein $x$ mit $Rang_{A[p..r]}(x) = k$. Das bedeutet: SELECT$(k, A, p, r)$ liefert ein $x \in A[p..r]$, so daß es $k-1$ Werte $A[j]$ gibt mit $p \leq j \leq r$ und $A[j] < x$ und es gibt $r-p-k+1$ Werte $A[j]$ mit $p \leq j \leq r$ und $A[j] > x$.

Für die Implementierung von SELECT nehmen wir die Existenz einer Funktion GOOD-SPLITTER an, die in linearer Zeit funktioniere soll und folgende Spezifikation hat:

- Die Funktion GOOD-SPLITTER$(A, p, r)$ liefert ein $x \in A[p..r]$, so daß höchstens $\alpha \cdot (r-p+1)$ der Elemente aus $A[p..r] \leq x$ und entsprechend höchstens $\alpha \cdot (r-p+1)$ der Elemente aus $A[p..r] > x$ sind. Hier ist $\alpha$ von vornherein festgelegt und es muß $1/2 < \alpha < 1$ gelten.

  Für ein sortiertes Array $A[p..r]$ ergibt sich folgende Graphik:




Die Funktion SELECT

**function** SELECT$(k, A, p, r)$
(1) **if** $p - r \leq 100$ **then**
SORT(A,p,r)
**return**$(A[p + k - 1])$
(* Wenn die Anzahl der Elemente relativ klein ist lohnt es sich nicht (für die Laufzeit), die Funktion auszuführen *)
(2) $x \leftarrow$ GOOD-SPLITTER$(A, p, r)$
(3) $q \leftarrow$ PARTITION$(A, p, r, x)$
(4) **if** $k \leq q - p$ **then return** SELECT$(k, A, p, q - 1)$
**else return** SELECT$(k - (q - p), A, q, r)$

Die Idee dieser Prozedur ist es, mit Hilfe eines guten Splitters sich das Feld $A[p..r]$ in ein Teilfeld $A[p..q-1]$ der "kleinen" Elemente und ein Teilfeld $A[q..r]$ der "großen" Elemente aufzuteilen, sodaß jedes dieser Teilfelder höchsten $\alpha(r - p + 1)$ Elemente enthält. Wenn es mehr mindestens $k$ "kleine" Elemente gibt, muß sich das Element vom Rang $k$ unter diesen befinden und hat natürlich auch unter den "kleinen" Elementen Rang $k$. Gibt es weniger als $k$ "kleine" Elemente, muß sich das gesuchte Element unter den "großen" befinden und hat natürlich unter ihnen Rang $k$ minus Anzahl der "kleinen" Elemente.
Das wichtige hier ist, daß der rekursive Aufruf auf einem Teilfeld stattfindet, dessen Größe höchstens $\alpha$ mal die Größe des ursprünglichen Feldes ist.
Die Laufzeittabelle für ein Feld der Länge $n$ ergibt:

| Zeilennr. | Laufzeit |
|---|---|
| 1 | $O(1)$ |
| 2 | $O(n)$ |
| 3 | |
| 4 | $T(\alpha n)$ |

Für die Gesamtlaufzeit ergibt sich die Rekursion

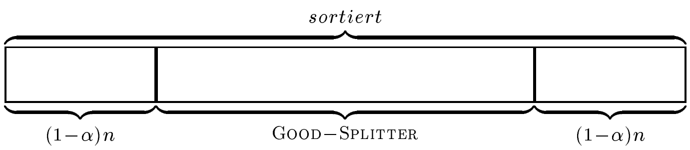
$$T(n) = O(n) + T(\alpha n),$$

woraus $T(n) = O(n)$ folgt.

Vorlesung vom 30.10.95
Matthias Hüther und Jörg Deßloch

Bei der Funktion SELECT hat man angenommen, daß man einen GOOD-SPLITTER in $O(n)$ Zeit finden kann. Nur dann läuft auch SELECT in linearer Zeit. In diesem Abschnitt geht es daher um die Frage: Wie findet man einen guten Splitter ?

### 2.2.1 1. Möglichkeit - RANDOMISIERT




Die Anzahl der GOOD-SPLITTER beträgt:

$$(1 - 2(1 - \alpha))n = \underbrace{(2\alpha - 1)}_{=:\beta>0} n.$$

Für $\alpha = \frac{3}{4}$ sind also die Hälfte aller Elemente Kandidaten für einen guten Splitter. Wählt man zufällig ein Element $x$ aus, so hat man mit Wahrscheinlichkeit $1/2$ einen guten Splitter. Allgemein beträgt die Wahrscheinlichkeit dafür $\beta$. Außerdem kann man in $O(n)$ Zeit überprüfen, ob das gewählte Element $x$ ein guter Splitter ist. Man muß nur alle Elemente $\leq x$ und alle Elemente $> x$ zählen. Hat man jeweils weniger als $\alpha n$ Elemente gezählt, so ist $x$ ein GOOD-SPLITTER. Falls nicht, so wählt man zufällig und unabhängig ein neues Element.

GOOD-SPLITTER$(A, p, r)$
**repeat**
Wähle zufälliges Element $x$ aus $A[p..r]$ und überprüfe, ob $x$ ein GOOD-SPLITTER ist. Falls ja, **return**$(x)$.

Wir nehmen an, daß wir zufällig ein Element auswählen können (gleichverteilt). In der Realität ist es natürlich kein „echter" Zufall, sondern ein sogenannter „kontrollierter" Zufall, da die zufällige Auswahl von uns selbst generiert und kontrolliert wird (Zufallsgenerator).
Die Lauzeit dieses Algorithmus kann natürlich nicht explizit angegeben werden, da sie von der Wahl des Elements $x$ abhängt. Im schlimmsten Fall läuft der Algorithmus unendlich lange. Wenn wir jetzt die Laufzeit betrachten, so betrachten wir die erwartete Laufzeit. Dazu definieren wir eine Zufallsvariable $X$, welche die Anzahl der Iterationen der repeat-Schleife angibt. Der Erwartungswert ist folgendermaßen definiert:

$$Ex[X] = \sum_{i \geq 1} i Pr(X = i).$$

In unserem Fall ist

$$Pr(X = i) = \underbrace{(1-\beta)^{i-1}}_{Fehlgriffe} \cdot \underbrace{\beta}_{Erfolg}$$

die Wahrscheinlichkeit, daß die repeat-Schleife genau $i$-mal durchlaufen wird (geometrische Verteilung). Für die erwartete Anzahl der Iterationen gilt nun:

$$\begin{aligned} Ex[X] &= \sum_{i \geq 1} i(1-\beta)^{i-1}\beta \\ &= \beta \sum_{i \geq 1} i(1-\beta)^{i-1} \\ &= \beta \sum_{i \geq 1} -\frac{d}{d\beta}(1-\beta)^i \\ &= -\beta \frac{d}{d\beta} \sum_{i \geq 1} (1-\beta)^i \\ &= -\beta \frac{d}{d\beta}((1-\beta) \underbrace{\sum_{i \geq 0} (1-\beta)^i)}_{=\frac{1}{1-(1-\beta)}=\frac{1}{\beta}} \\ &= -\beta \frac{d}{d\beta}(\frac{1}{\beta} - 1) \\ &= \frac{1}{\beta} = O(1) \end{aligned}$$

Die erwartete Laufzeit der Funktion GOOD-SPLITTER ist $\frac{1}{\beta}O(n) = O(n)$. Dies ist nicht die Laufzeit für den schlechtesten Fall, sondern die erwartete Laufzeit (über den Schleifendurchläufen)!!! Bezüglich dieses Zufalls beträgt die erwartete Laufzeit für Selektion $O(n)$. Solch ein Algorithmus, der den Zufall mit ins Spiel bringt, heißt „RANDOMISIERTER ALGORITHMUS".

### 2.2.2 2. Möglichkeit - DETERMINISTISCH (Blum, Floyd, Pratt, Rivest, Tarjan)

Wir wollen für $n$ Elemente *deterministisch*, also ohne die Verwendung von Zufall, einen guten Splitter finden. Dazu gehen wir folgendermaßen vor:

1. bilde $\lfloor \frac{n}{5} \rfloor$ 5-er Gruppen
2. für jede 5-er Gruppe finde ihren Median
3. finde $x$, den Median der $\lfloor \frac{n}{5} \rfloor$ Mediane
4. **return**$(x)$

| Zeilennr. | Kosten | Bemerkungen |
|---|---|---|
| 1 | $O(n)$ | |
| 2 | $\lceil \frac{n}{5} \rceil \cdot O(1) = O(n)$ | suche dritt-kleinste Elemente |
| 3 | $O(n)$ | mit Hilfe von SELECT |

Um den Wert von $\alpha$ zu ermitteln betrachten wir Bild 2.1.

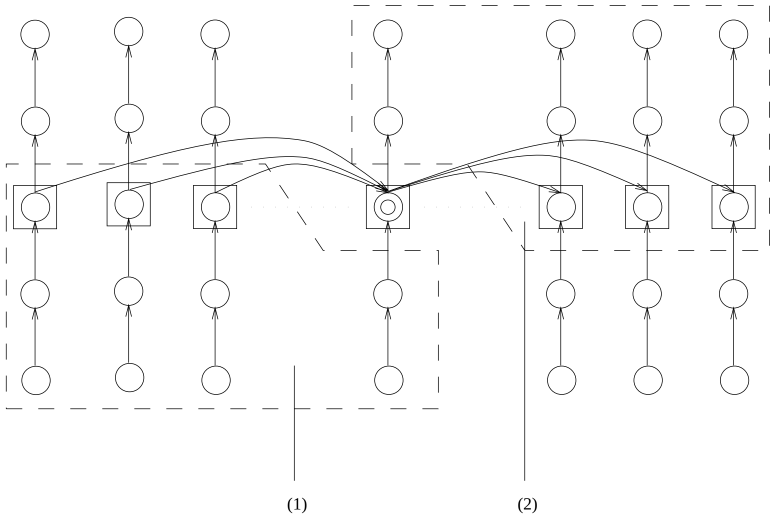



Abbildung 2.1: Aufteilung der Elemente in 5-er Gruppen

Die einzelnen 5-er Gruppen sind von links nach rechts und von unten nach oben aufsteigend sortiert (Pfeile). Die Mediane der einzelnen Gruppen sind jeweils eingerahmt. Das Element in der Mitte ist der „Median der Mediane“. Wir bezeichnen es mit $x$. Links von $x$ liegen die Mediane $\leq x$ und rechts von $x$ liegen die Mediane $> x$. Die Anzahl der Elemente aus (1) bzw. aus (2) beträgt jeweils $\frac{3}{10}n$ und die Anzahl der restlichen Elemente (über die man nichts aussagen kann) beträgt $\frac{4}{10}n$. Es sind also maximal $\frac{3}{10}n + \frac{4}{10}n = \frac{7}{10}n$ Elemente $\leq x$ bzw. $> x$.
Deshalb gilt: $x$ ist GOOD-SPLITTER mit $\alpha = \frac{7}{10}$.
Damit erhält man für die Laufzeit $T(n)$ von SELECT folgende Abschätzung:

$$T(n) = \begin{cases} O(1) & \text{für } n < 100 \\ \underbrace{A \cdot n}_{f(n)} + T(n/5) + T(n_2) & \text{fur } n \geq 100 \end{cases}$$

Dabei ist $T(n/5)$ die Zeit für den rekursiven Aufruf beim Finden des Goodsplitters und $T(n_2)$ ist die Zeit für den rekursiven Aufruf innerhalb von SELECT.
Es gilt dann mit $n_1 = n/5$ und $n_2 \leq 7/10$):

$$\sum_i f(n_i) = A\frac{n}{5} + A\frac{7}{10}n = \underbrace{\frac{9}{10}}_{c<1} f(n),$$

also

$$T(n) = O(f(n)) = O(n).$$

Wir wollen nun die 5-er Gruppen sortieren, ohne daß wir zusätzlichen Speicherplatz benötigen. Dazu verwenden wir folgenden Trick:
Die einzelnen 5-er Gruppen bestehen aus den Elementen: $A[i], A[\frac{1}{5}n + i], A[\frac{2}{5}n + i], A[\frac{3}{5}n + i], A[\frac{4}{5}n + i]$.

### 2.2.3 Einfach Randomisiert

Wir kommen nun zu einem einfacheren Algorithmus, der darauf verzichtet einen GOOD-SPLITTER zu suchen. Stattdessen wählt er zufällig ein Element aus, in der Hoffnung, daß es ein GOOD-SPLITTER ist, und verfährt weiter wie bei SELECT.
Wenn wir Pech haben, terminiert der Algorithmus nie! Dies ist z.B. der Fall, wenn immer das größte Element ausgewählt wird (s. Übung 3, Aufg.3).

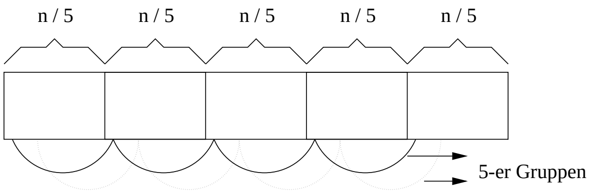



Abbildung 2.2: Speicheraufteilung

1. **RANDOMIZED-SELECT**$(k, A, p, r)$
2. **if** $r - p < 100$ **then** SORT$(A, p, r)$
3. **return**$(A[p + k - 1])$
4. $j$=zufällige Zahl aus $[p..r]$
5. $x = A[j]$
6. $q$=PARTITION$(A, p, r, x)$
7. **if** $k \leq q - p$ **then** SELECT$(k, A, p, q - 1)$
8. **else** SELECT$(k - (q - p), A, q, r)$

Für die erwartete Laufzeit $T(k, n)$, $n = r - p + 1$, von RANDOMIZED-SELECT, gilt:

$$T(n) = \begin{cases} O(1) & ,n < 100 \\ A \cdot n + \underbrace{\sum_{1 \leq i < k} \frac{1}{n} T(k - i, n - i)}_{Zeile 8} + \underbrace{\sum_{k \leq i \leq n} \frac{1}{n} T(k, i)}_{Zeile 7} & ,n \geq 100. \end{cases}$$

Dabei ist die Wahrscheinlichkeit, daß $x$ das $i$-kleinste Element $Pr(Rang(x) = i) = 1/n$ für alle $1 \leq i \leq n$.
<u>Beh.</u> $T(k, n) \leq En$, d.h. die erwartete Laufzeit ist linear.
<u>Bew.</u> durch vollständige Induktion nach $n$:

$$\begin{aligned}
T(k, n) &\leq An + \sum_{i=1}^{k-1} \frac{1}{n} E(n - i) + \sum_{i=k}^{n} \frac{1}{n} Ei \qquad \text{nach Induktionsvoraussetzung.} \\
&= An + \frac{1}{n} E\Big[ \underbrace{\sum_{i=1}^{k-1} (n - i)}_{\frac{n(n-1)}{2} - \frac{(n-k)(n-k+1)}{2}} + \underbrace{\sum_{i=k}^{n} i}_{\frac{n(n+1)}{2} - \frac{k(k-1)}{2}} \Big] \\
&\leq An + \frac{E}{n} \Big[ n^2 - \frac{1}{2} \big( \underbrace{k(k - 1) + (n - k)(n - k + 1)}_{\geq 2\left(\frac{n}{2}\right)^2, k = \frac{n}{2}} \big) \Big] \\
&\leq An + \frac{E}{n} [n^2 (1 - \frac{1}{4})] \\
&= (A + \frac{3}{4} E) n \overset{!}{\leq} En \qquad \text{wähle } E = 4A
\end{aligned}$$

Also: RANDOMIZED-SELECT hat ebenfalls erwartete lineare Laufzeit und ist einfacher (da ohne GOOD-SPLITTER) als SELECT.

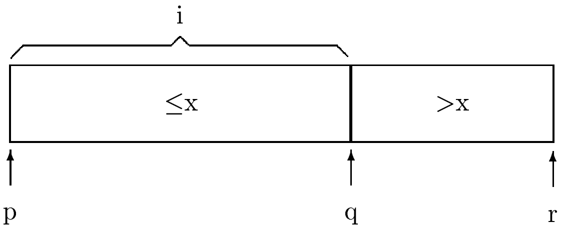
Vorlesung vom 02.11.1995
Marc Langenbach, Michael Schmidt

## 2.3 Quicksort

QUICKSORT ist ein weiterer (randomisierter) Sortieralgorithmus mit erwarteter Laufzeit von $O(n \log n)$. Er wurde 1960 von C.A.R. Hoare entwickelt. Ein Vorzug ist, daß er am Ort („in place“) sortiert, also kein Hilfsfeld benötigt.
Der eigentlicht Algorithmus läßt sich wie folgt darstellen:

(1) **function** QUICKSORT$(A, p, r)$
(2) **if** $r \leq p$ **then return**
(3) wähle zufällig (und gleichverteilt) ein $j \in [p \ldots r]$
und setze $x = A[i]$
(4) $q =$PARTITION$(A, p, r, x)$
(5) QUICKSORT$(A, p, q-1)$
(6) QUICKSORT$(A, q, r)$

Wenn der Algorithmus bis zum ersten rekursiven Aufruf (Zeile 5) ausgeführt wurde, ist das Feld folgendermaßen aufgebaut:




Zu beachten ist, daß mit zufälligem $j$ gilt: $i := q - p$ tritt mit einer Wahrscheinlichkeit von $\frac{1}{r-p+1}$ auf. Betrachten wir nun die Laufzeit, genauer gesagt die *erwartete* Laufzeit von Quicksort ($n = r - p + 1$):

$$T(n) = \begin{cases} O(1) & , n \leq 1 \\ dn + \sum\limits_{1 \leq i \leq n} \frac{1}{n}(T(i) + T(n-i)) & , n > 1 \end{cases}$$

Die Summe kommt dadurch zustande, da bei $i$ kleinen Elementen $n - i$ große auftreten und jede dieser Kombinationen die gleiche Wahrscheinlichkeit besitzt, nämlich $1/n$.
Wir formen nun obige Formel um, indem wir aus der Summe den Fall $i = n$ herausziehen und weiter beachten, daß jeder Summand der neuen Summe zweimal vertreten ist:

$$T(n) = dn + \underbrace{\frac{1}{n}T(0)}_{\text{fällt weg}} + \frac{1}{n}T(n) + \frac{2}{n} \sum_{1 \leq i < n} T(i).$$

Nun verwenden wir einen kleinen Trick: die gesamte Gleichung wird mit $n$ multipliziert und gleichzeitig der Summand $T(n)$ auf die linke Seite gebracht.

$$(1) \qquad (n-1)T(n) = dn^2 + 2 \sum_{1 \leq i < n} T(i).$$

Wenn diese Gleichung für $n$ gilt, dann gilt sie auch für $n - 1$:

$$(2) \qquad (n-2)T(n-1) = d(n-1)^2 + 2 \sum_{1 \leq i < n-1} T(i).$$

Man subtrahiert (2) von (1) und erhält:

$$\begin{aligned} (n-1)T(n) - (n-2)T(n-1) &= d(2n-1) + 2T(n-1) \\ (n-1)T(n) &= d(2n-1) + nT(n-1) \\ T(n) &= d \underbrace{\frac{2n-1}{n-1}}_{=2+\frac{1}{n-1} \leq 3} + \frac{n}{n-1}T(n-1) \\ T(n) &\leq 3d + \frac{n}{n-1}T(n-1) \end{aligned}$$

Durch sukzessives Auflösen der Rekursion erhält man

$$
\begin{aligned}
T(n) &\leq 3d + \frac{n}{n-1}T(n-1) \\
&\leq 3d + \frac{n}{n-1}(3d + \frac{n-1}{n-2}T(n-2)) \\
&= 3d(1+\frac{n}{n-1}) + \frac{n}{n-1}\cdot\frac{n-1}{n-2}\underbrace{T(n-2))}_{\leq(3d+\frac{n-2}{n-3}T(n-3))} \\
&\leq 3d(1+\frac{n}{n-1}+\frac{n}{n-2}) + \frac{n}{n-3}T(n-3) \\
&\vdots
\end{aligned}
$$

Im allgemeinen gilt somit

$$
\begin{aligned}
T(n) &\leq 3d(\underbrace{1}_{=\frac{n}{n}} + \frac{n}{n-1} + \frac{n}{n-2} + \cdots + \frac{n}{n-k}) + \frac{n}{n-k-1}T(n-k-1) \\
&\leq 3dn(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{n-k}) + \frac{n}{n-k-1}T(n-k-1)
\end{aligned}
$$

Für $k = n-2$ erhält man dann

$$
T(n) \leq 3dn \underbrace{(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2})}_{H_n - 1,\ n\text{-te harmonische Zahl-1} \approx \log n - 1} + \underbrace{\frac{n}{1}T(1)}_{O(n)}
$$

Somit ergibt sich für die erwartete Laufzeit

$$
T(n) \approx 3dn \ln n + O(n) = O(n \log n)
$$

## 2.4 Counting-Sort

Der Ansatz bei den bisher betrachteten Sortierverfahren war, durch Vergleiche die Elemente in die richtige Reihenfolge zu bringen. Wenn man *beschränkte* Mengen betrachtet, d.h. Mengen, deren Elemente in $[0 \ldots k-1]$, $k \in \mathbb{N}$ liegen, kann man sich das Adressierungspotential der verwendeten Maschine zunutze machen.
Der Counting-Sort Algorithmus sieht wie folgt aus:

(1) **function** Counting-Sort$(A, B, k)$
  (* A Eingabefeld, B Ausgabefeld, k wie oben *)
(2) **for** $i = 0$ **to** $k-1$ **do** $C[i] = 0$
  (* Zählerfeld C wird initialisiert *)
(3) **for** $j = 1$ **to** $length(A)$ **do** **incr**$C[A[j]]$
  (* $C[i]$ = Anzahl Vorkommen der Zahl $i$ in $A$ *)
(4) **for** $i = 1$ **to** $k-1$ **do** $C[i] = C[i] + C[i-1]$
  (* Kumulative Zähler erzeugen *)
  (* $C[i]$ = Anzahl der Vorkommen der Zahlen $\leq i$ in $A$ *)
(5) **for** $j = length(A)$ **downto** 1 **do**
(6)   $B[C[A[j]]] = A[j]$
(7)   **decr**$(C[A[j]]$

Der Algorithmus läßt sich leicht am folgenden Beispiel verstehen:

Sei A = $[1_a\ 3\ 2_a\ 1_b\ 2_b\ 2_c\ 1_c]$. In diesem Fall ist $k = 4$. Vor dem Schreiben in B enthält C = [0 3 6 7].
Es folgen die einzelnen Schleifendurchläufe der letzten Schleife des Algoritmus:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (0) | B=[ | | | | | | | | ] | C=[ | 0 | 3 | 6 | 7 | ] |
| (1) | B=[ | | | $1_c$ | | | | | ] | C=[ | 0 | 2 | 6 | 7 | ] |
| (2) | B=[ | | | $1_c$ | | | $2_c$ | | ] | C=[ | 0 | 2 | 5 | 7 | ] |
| (3) | B=[ | | | $1_c$ | | $2_b$ | $2_c$ | | ] | C=[ | 0 | 2 | 4 | 7 | ] |
| (4) | B=[ | | $1_b$ | $1_c$ | | $2_b$ | $2_c$ | | ] | C=[ | 0 | 1 | 4 | 7 | ] |
| (5) | B=[ | | $1_b$ | $1_c$ | $2_a$ | $2_b$ | $2_c$ | | ] | C=[ | 0 | 1 | 3 | 7 | ] |
| (6) | B=[ | | $1_b$ | $1_c$ | $2_a$ | $2_b$ | $2_c$ | 3 | ] | C=[ | 0 | 1 | 3 | 6 | ] |
| (7) | B=[ | $1_a$ | $1_b$ | $1_c$ | $2_a$ | $2_b$ | $2_c$ | 3 | ] | C=[ | 0 | 0 | 3 | 6 | ] |

Im Schritt von (0) nach (1) passiert folgendes:

$$j = 7 \Rightarrow A[7] = 1 \Rightarrow C[1] = 3 \Rightarrow B[3] = A[7] \text{ und } \mathbf{decr(C[1])}$$

In den übrigen Schritten wird ähnlich verfahren.
Die Laufzeit von Counting-Sort beträgt, da nur Felder durchlaufen werden müssen:

$$T(n) = O(n + k).$$

Wenn $k$ nicht sehr viel gröser als $n$ ist ($k = O(n)$), gilt: $T(n) = O(n)$.

Die letzte Schleife des Algorithmus, in der das Zielfeld aufgefüllt wird, legt die Frage nahe, warum von $length(A)$ bis 1 und nicht in umgekehrter Reihenfolge gezählt wird. Aber gerade durch diesen Sachverhalt bleibt die relative Ordnung des Quellfeldes erhalten, d.h. die Reihenfolge gleicher Elemente wird nicht vertauscht. Aufgrund dieser Tatsache wird Counting-Sort auch als *stabiler* Algorithmus bezeichnet.

**Definition 1** *(stabiler Algortihmus) Ein* ***stabiler*** *Sortieralgorithmus ist ein Algorithmus, bei dem die relative Ordnung von Elementen der Eingabe mit gleichen „Schlüssel" in der Ausgabe erhalten bleibt.*

Der Algorithmus ist sehr viel schneller als Quicksort, wenn $k$ nicht allzu groß wird ($k = O(n)$).

## 2.5 Radix-Sort

Es gibt Tricks, so daß auch bei großem $k$ in linearer Zeit sortiert werden kann. Bisher sortierten wir $n$ Zahlen aus $[0 \ldots k-1]$ in $O(n+k)$ Zeit (oder $O(n)$ Zeit, falls $k = O(n)$). Nun betrachten wir Elemente aus dem Intervall $[0 \ldots k^d - 1]$, unter der Voraussetzung, daß $d \in \mathbb{N}$ konstant und $k = O(n)$ ist.
Die Zahlen daraus kann man sich als $d$-stellige Zahl mit Ziffern aus $[0 \ldots k-1]$ vorstellen. Die Ziffern sind geordnet, so daß 0 die unwichtigste Ziffer und $k-1$ die wichtigste ist. Für $k = 10$ und $d = 4$ kann man Zahlen von 0 bis 9999 sortieren, ohne 10000 Zeiteinheiten oder Speicherzellen zu benutzen.
Im allgemeinen läßt sich eine Zahl $A \in [0 \ldots k^d - 1]$ wie folgt darstellen:

$$A = \sum_{i=0}^{d-1} a_i\, k^i \qquad a_i \in [0 \ldots k-1]$$

Der eigentliche Algorithmus:

**function** Radix-Sort$(A, d)$
    (* A Eingabefeld, d Anzahl Ziffern pro Zahl *)
    **for** $i = 0$ **to** $d - 1$ **do**
        sortiere $A$ bzgl. Ziffer $i$ mit Hilfe eines stabilen Sortieralgorithmus

Als Sortieralgorithmus kann man z.B. Counting-Sort verwenden.

**Beispiel:** Für $k = 10$, $d = 3$ und $n = 7$ wird so sortiert:

| | | | | | | |
|---|---|---|---|---|---|---|
| 329 | | 720 | | 720 | | 329 |
| 457 | | 355 | | 329 | | 355 |
| 657 | | 436 | | 436 | | 436 |
| 839 | $\xrightarrow{\text{Einer}}$ | 457 | $\xrightarrow{\text{Zehner}}$ | 839 | $\xrightarrow{\text{Hunderter}}$ | 457 |
| 436 | | 657 | | 355 | | 657 |
| 720 | | 329 | | 457 | | 720 |
| 355 | | 839 | | 657 | | 839 |

Die relative Ordnung muß bei diesen Schritten unbedingt erhalten bleiben. Das Sortierproblem wird hierbei nicht in $d$ Teilprobleme, die jeweils für sich sortiert und wieder zusammengefügt werden, aufgeteilt, wie man vielleicht rein naiv vorgehen würde.
Bei Verwendung eines linearen Sortieralgorithmus, der Zeit $O(n+k)$ benötigt, ergibt sich eine Laufzeit von

$$T(n) = O(d(n+k))$$

Falls $d$ klein und $k = O(n)$ ist, ergibt sich $T(n) = O(n)$.

Vorlesung vom 6.11.95
Ulrich Alt, David Christensen, Stefan Janocha

************* Dieser folgende kleingedruckte ************
************ Teil kann vorerst ignoriert werden *************

## 2.6 K-R-Sort

### 2.6.1 Idee

Mit RADIX-SORT verfügen wir über einen Algorithmus, der es ermöglicht, in linearer Zeit ($O(dn)$) $n$ Zahlen aus $[0,\ldots,k^d-1]$, $k \in O(n)$, zu sortieren. Man kann jedoch (zumindest aus theoretischem Interesse) einen Algorithmus finden, der in gleicher Laufzeit Werte aus einem viel größeren Wertebereich, nämlich $[0,\ldots,k^{2^d}-1]$ sortiert. Diesen Algorithmus nennen wir nach seinen Erfindern Kirkpatrick und Reisch K-R-SORT. Er basiert auf dem „Teile und Herrsche"-Prinzip.
Bei RADIX-SORT wurde eine Laufzeit von $O(n)$ benötigt, um das Problem von $d$ Ziffern auf $d-1$ Ziffern zu reduzieren, die im nächsten Durchlauf nur noch betrachtet werden. Wollen wir, wie in unserem Fall, also „exponentiell" viele Ziffern zulassen, sollten wir versuchen, in linearer Zeit von einem $D$-Ziffernproblem auf ein $D/2$-Ziffernproblem zu kommen.

### 2.6.2 Algorithmus

Dem Algorithmus wird eine $n$-elementige Liste $B$ von Zahlen aus dem Bereich $[0,\ldots,k^{2^d}-1]$ gegeben. Die Zahlen seien auf maximal $D = 2^d$ Stellen zur Basis $k$ codiert. Wir nehmen an, daß zu jedem Element noch zusätzlich ein Schlüssel der Länge $\frac{D}{2}$ gespeichert wird, nämlich das Element der Länge $\frac{D}{2}$, welches sich der rekursive Aufruf anschaut.

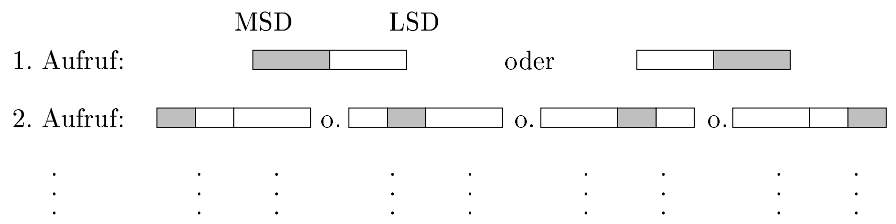



Abbildung 2.3: Mögliche Positionen des Schlüssels

Dieser Schlüssel wird in jeder Rekursionsstufe in seiner Länge halbiert, kann aber an verschiedenen Positionen innerhalb der zu sortierenden Zahl liegen, wie in Abbildung 2.3 veranschaulicht. Die Funktion KEYOF liefert zu einem Element den entsprechenden Schlüssel. Zusätzlich wird für jede Zahl noch ein Bit abgespeichert, das angibt, ob die spezielle Zahl „aus M kommt". Zwei weitere Funktionen, MSD bzw. LSD, geben die höherwertigen („more significant") bzw. niederwertigen („less significant") $\frac{D}{2}$ Stellen der übergebenen Zahl zurück.
Im Algorithmus werden die folgenden Datenelemente benutzt:

- $L$ ist ein Feld $[0..k^{D/2}-1]$, in dem die Elemente aus $B$ durch direktes Adressieren der höherwertigen $D/2$ Stellen mit den niederwertigen Stellen als Schlüssel in Listen abgespeichert werden.
- $M$ ist eine Liste von Zahlen, die beim ersten Durchlauf durch $B$ erstellt wird. Sie enthält für jeden vorkommenden $\frac{D}{2}$-stelligen Präfix einen Repräsentanten aus $B$.
- $MIN$ ist ein Feld $[0..k^{D/2}-1]$, in dem für jeden vorkommenden Präfix das Element gespeichert wird, das den kleinsten $\frac{D}{2}$-stelligen Suffix besitzt.
- $BB$ ist eine Hilfsdatenstruktur, in der Elemente für einen rekursiven Aufruf von K-R-SORT abgelegt werden können.

Der Algorithmus K-R-SORT benutzt die Datenstruktur einer *verketteten Liste*. Auf Listen stehen folgende Operationen zur Vefügung:

- INIT($M$) setzt $M$ auf die leere Liste.
- INSERT($M, x, key, mark$) fügt das Element $x$ am Ende in die Liste $M$ ein. $mark$ gibt an, ob das Markierungsbit für dieses Element 1 oder 0 sein soll. $key$ gibt die $\frac{d}{2}$ Bits an, die sich der Algorithmus beim nächsten rekursiven Aufruf ansehen soll.
- APPEND($M, L$) hängt die Elemente der Liste $L$ an die Liste $M$ an.

Desweiteren brauchen wir eine Prozedur DELETEMIN, die aus einer übergebenen Liste dem Schlüssel nach das kleinste Element heraussucht, dieses aus der Liste entfernt und dessen Wert zurückgibt. Diese Prozedur durchsucht linear die Liste, braucht also $O(k)$ Zeit, wenn die Liste $k$ Elemente enthält.
Der Algorithmus sieht wie folgt aus:

```
procedure K-R-SORT(B, D)
    if D = 1 then
        COUNTINGSORTUNDFERTIG(B);
    for each b ∈ B do
        x = KEYOF(b)
        x' = LSB(x, D)
        x'' = MSD(x,D)
        if L[x''] = leere Liste then
            INSERT(M, b, x'', True)
        INSERT(L[x''], b, x', False);
    INIT(BB)
    for each b ∈ M do
        MIN[KEYOF(b)] = DELETEMIN(L[KEYOF(b)])
        APPEND(BB, L[KEYOF(b)])
    APPEND(BB, M)

    K-R-SORT(BB, D/2)

    INIT(M)
    for each bb ∈ BB do
        if bb „kommt aus M“ then
            INSERT(M, bb, Unwichtig, Unwichtig)
        else
            INSERT(L[MSB(bb, D)], bb, Unwichtig, Unwichtig)
    INIT(B)
    for each bb ∈ M do
        INSERT(B, MIN[MSB(bb, D)])
        APPEND(B, L[MSB(bb, D])

    return B
```

### 2.6.3 Funktion

Wie und wieso funktioniert dieser Algorithmus ? Um das zu verstehen, kann man sich den Algorithmus in 3 Phasen aufgeteilt denken: das Vorsortieren vor dem rekursiven Aufruf von K-R-SORT, der rekursive Aufruf und das nachfolgende Sortieren. Das folgende Beispiel verdeutlicht das. Gibt man die im Bild aufgeführten Zahlen ein, so erhält man nach dem ersten Abschnitt des Algorithmus die Anordnung in Bild 2.5.

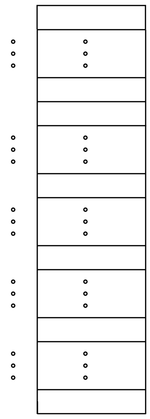


Abbildung 2.4: K-R-SORT mit $n = 8$ und $D = 4$

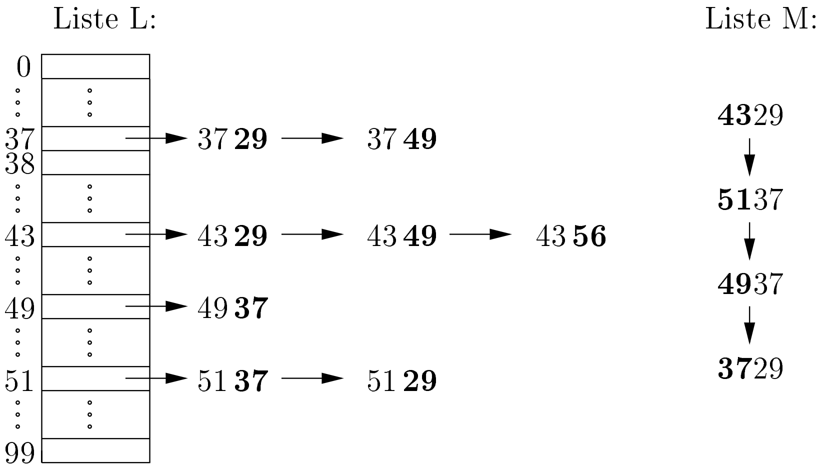



Abbildung 2.5: Inhalt der Listen $L$ und $M$ nach erstem Durchlauf

Im Feld $L$ wurden $l$ Listen angelegt, wobei $l$ die Anzahl verschiedener $\frac{D}{2}$- stelliger Präfixe von Zahlen ist. Das heißt, die Zahlen stimmen in jeder Liste in den ersten $\frac{D}{2}$ Stellen überein. Für jede dieser Listen enthält die Liste $M$ jeweils das erste Element. $M$ enthält also genau $l$ Elemente in der Reihenfolge des Auftretens in der Eingabe, die als besonders markiert sind. Eine Möglichkeit für einen „Teile und Herrsche"-Algorithmus wäre es, alle diese Listen nach den letzten, sowie $M$ nach den ersten $\frac{D}{2}$ Stellen zu sortieren. Dann müßten wir allerdings $n + l$ Zahlen sortieren, d.h. wir haben zwar die Stelligkeit des Problems verringert, dafür aber schlimmstenfalls $l = n$ Zahlen mehr zu sortieren, haben unser Problem also nicht unbedingt kleiner gemacht.
Wir müssen also aus den zu sortierenden Zahlen in $M$ und $L$ genau $l$ Elemente entfernen, die wir anschließend möglichst leicht wieder in das Feld einsortieren können. Hier nehmen wir die Minima jeder der $l$ Teillisten und merken und diese separat in einem Feld $MIN$. Die restlichen Elemente verketten wir nacheinander in der Liste $BB$. $BB$ enthält also die Elemente der Listen von $L$ ohne die jeweiligen kleinsten Zahlen, geordnet nach den ersten (wichtigsten) $\frac{D}{2}$ Stellen, danach die Elemente von $M$. In diesem Moment kommen unter Umständen manche Zahlen doppelt in der Liste vor, nämlich die ersten Zahlen von jeder Teilliste in $L$, sofern diese ersten Zahlen nicht auch die kleinsten in ihrer Liste waren, denn diese Zahlen sind sowohl in der Listen in $L$ als auch in $M$ enthalten. Wichtig ist jedoch, daß diese eventuell doppelten Zahlen nach verschiedenen Schlüsseln sortiert werden, die Zahlen aus $M$ nach den höherwertigen, die Zahlen aus $L$ nach den niederwertigen Stellen.
Man ruft also K-R-SORT mit $n$ Zahlen und der Breite $\frac{D}{2}$ für die Liste $BB$ auf. Dies wird in der Laufzeitanalyse wichtig sein. Zurück bekommt man, sofern K-R-SORT korrekt ist, eine sortierte Liste aller Elemente in $BB$, wobei jeweils der vorher gewählte Schlüssel betrachtet wurde.
Was ist mit den markierten Elementen, die vorher in $M$ waren, passiert ? Diese bilden nun eine nach den höherwertigen Stellen sortierte Teilfolge der Liste $BB$, d.h. wir können durch einfaches Durchlaufen von $BB$ eine nach den höherwertigen Stellen

sortierte Liste $M'$ erstellen, die die Elemente des alten $M$ (diese wurden vorher markiert) enthält. Um Platz zu sparen, speichert man diese Liste in der neu initialisierten Liste $M$. Die Elemente aus dem alten $L$ bilden nun auch eine aufsteigend sortierte Teilfolge in $BB_i$, allerdings nach den niederwertigen Bits. Fügt man diese Elemente $bb \in BB$ in die Listen L[MSB($bb, D$)] ein, so stehen in diesen Listen am Ende die gleichen Elemente, die vorher darin gestanden haben, jedoch sind die niederwertigen Stellen aufsteigend sortiert.
Der Rest ist einfach. Die höherwertigen Stellen der Zahlen in $M$ geben an, in welcher Reihenfolge die Listen in $L$ zu durchlaufen sind. Wir tragen also jetzt in die Ergebnisliste $B$ alle Elemente ein, die mit den höherwertigen Stellen des ersten Elements aus $M$ beginnen. Zuerst kommt hier das in $MIN$ gesicherte Minimum, danach alle Elemente aus $L$ in der dort vorliegenden (s.o.) Reihenfolge. Dies wiederholen wir nach und nach für alle Elemente in $M$. Ergebnis ist eine aufsteigend sortierte Liste $B$.

### 2.6.4 Laufzeit

Für den Fall, daß $D = 1$ ist, haben wir lineare Laufzeit in $n$, da wir COUNTINGSORT benutzen.
Die **for**-Schleifen werden jeweils $O(n)$ mal durchlaufen und benötigen jeweils konstante Zeit.
Alles in allem haben wir also lineare Laufzeit bis auf den Rekursionsschritt. Wir können demnach die Laufzeit abschätzen durch:

$$T(n, D) \leq \begin{cases} O(n) & D = 1 \\ O(n) + T(n, D/2) & D > 1 \end{cases}$$

Das $D$ wird hier immer halbiert währen $n$ konstant bleibt. Daraus sieht man leicht, daß für die Gesamlaufzeit gilt $T(n, D) = O(n \log D)$. Der Speicherbedarf dieses Algorithmus ist allerdings unpraktisch groß.

************* Ende des ignorierbaren Teils *************

## 2.7 Untere Schranken

In den vorangegangenen Abschnitten wurden verschiedene Sortieralgorithmen behandelt. Diese benötigen verschieden viel Zeit und verschieden viel Platz. Die allgemeine Tendenz war, daß eine Anhebung der Platzschranken für das Sortieren zu einem schnelleren Algorithmus führte.Man möchte jetzt gerne wissen, ab welcher Zeitschranke man „aufhören darf“ zu denken, d.h. ab welcher Schranke theoretisch keine Verbesserung mehr möglich ist. Man sieht leicht, daß man zum Sortieren von $n$ Zahlen mindestens Zeit $\Omega(n)$ braucht, weil man sonst die Eingabe nicht ganz anschauen könnte. Damit wir weitere Aussagen machen können, müssen wir bestimmte Voraussetzungen an unsere Algorithmen machen:

### 2.7.1 Voraussetzungen

- Die Algorithmen müssen deterministisch sein, sonst könnten sie nicht auf realen Maschinen implementiert werden.

- Die Algorithmen sind *vergleichsorientiert.* d.h. die einzige Operation, die mit zwei Schlüsseln oder Zahlen ausgeführt werden darf, ist ein Vergleich. Man darf die Schlüssel also nicht in arithmetischen Operationen oder zur Indizierung von Maschinenzellen benutzen

Mit diesen beiden Voraussetzungen können wir den Ablauf eines Sortieralgorithmus eindeutig auf einen sogenannten Entscheidungsbaum abbilden. Ein Entscheidungsbaum ist ein binärer Baum, dessen innere Knoten mit Paaren von Elementen, und dessen Blätter mit den $n!$ Permutationen von $n$ Elementen beschriftet sind. Jeder innere Knoten gibt also zwei Elemente $n_i$ und $n_j$ an, die verglichen werden, das linke Kind den Fall, daß $n_i \leq n_j$ ist, das rechte Kind den Fall, daß $n_i > n_j$ ist. Dies geht solange weiter, bis man an einem Blatt angekommen ist. Das Blatt gibt die Permutation an, die auf das Feld anzuwenden ist, um es zu sortieren.
Die Abbildung des Sortieralgorithmus auf einen Entscheidungsbaum funktioniert deshalb, weil der Algorithmus deterministisch ist, d.h. die Reihenfolge der Paare, die verglichen werden, ist bei jedem Lauf des Algorithmus gleich und von der Eingabe unabhängig. Zur Berechnung der richtigen Permutation dürfen nach Voraussetzung nur Vergleiche benutzt werden, also kann der komplette Rechenweg eindeutig in einen solchen Baum codiert werden.

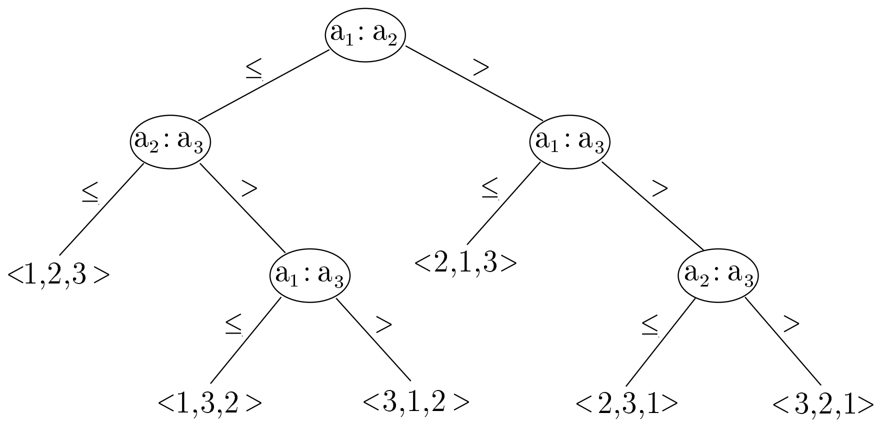



Abbildung 2.6: Entscheidungsbaum für $n = 3$

### 2.7.2 Beweisskizze

Die Länge des Pfades von der Wurzel bis zu einem Blatt gibt die Anzahl aller Vergleiche an, die der Algorithmus angestellt hat, um dieses Ergebnis zu berechnen. Nun hat jeder Entscheidungsbaum mindestens $n!$ Blätter, sonst könnte eine Permutation $\sigma$ nicht als Ergebnis auftreten, folglich könnte der Algorithmus die Zahlen $1, \ldots n$, die gemäß $\sigma^{-1}$ angeordnet sind, nicht richtig sortieren. Der Baum kann aber mehr als $n!$ Blätter haben. Dies bedeutet, daß er auf mehr als einem Berechnungspfad zum gleichen Ergebnis kommen kann, was nicht verboten ist. Also ist die minimale Höhe eines solchen Baumes eine untere Schranke für die Anzahl der Vergleichsoperationen und somit auch für die Gesamtoperationen des Algorithmus.
Man weiß, daß die Höhe eines binären Baumes mit $n!$ Blättern mindestens $\lceil \log_2 n \rceil$ beträgt. Also ist die minimale Höhe $\log_2 n! \in \Omega(n \log n)$ (Stirling-Formel) und damit ist die untere Schranke gefunden. Von dieser Eigenschaft überzeugt man sich dadurch, daß für jede neue „Etage" im Baum die Anzahl der Blätter höchstens verdoppelt wird. Mit anderen Worten, um $k$ Kinder zu erreichen, muß man mindestens $\lceil \log_2 k \rceil$ „Etagen" einziehen.
Wir formulieren den folgenden

**Satz:** Für jeden auf Vergleichen basierenden Algorithmus zum Sortieren von $n$ Elementen gibt es eine Eingabe, für die der Algorithmus mindestens $\lceil \log_2 n! \rceil \in \Omega(n \log n)$ Vergleiche verwendet.

Man nennt dies die „informationstheoretische" untere Schranke für das Sortierproblem.

# Kapitel 3

# Dynamische Mengen aus begrenzten Bereichen

Vorlesung vom 13.11.1995
Nikolaus Gerteis, Michael Rumpf

## 3.1 Operationen auf dynamischen Mengen

In der Praxis besteht oft die Notwendigkeit Mengen zu bearbeiten. So sollen zum Beispiel Elemente eingefügt oder gesucht werden. Solche Mengen bestehen aus sogenannten *Stücken* (engl. Items). Ein Stück setzt sich aus einem *Schlüssel* und beliebig vielen anderen Informationen zusammen (siehe Abbildung 3.1), wobei wir die Zuordnung zwischen Schlüsseln und Stücken als eindeutig annehmen.
Beispiele für Stücke sind u.a.: die Einträge in einem Wörterbuch (Schlüssel: deutsches Wort, Information: englische Übersetzung), Telefonbucheinträge (Schlüssel: Name des Teilnehmers, Information: Vorwahl und Telefonnummer), sowie die Einträge in der Symboltabelle eines Compilers (Schlüssel: Variablenname, Informationen: Speicherstelle, Typ).
Die dynamischen Mengen sollen durch folgende Operationen manipulierbar sein:

- $S$ = Makeset() - erzeugt eine leere Menge $S$
- Empty?($S$) : $\{True, False\}$ - testet, ob die angegebene Menge $S$ leer ist
- Insert($x$,$S$) - fügt Stück $x$ in Menge $S$ ein
- Delete($x$,$S$) - löscht $x$ aus $S$
- Search($k$,$S$) : $\{$*Stücke*$\}$ - liefert das Stück mit Schlüssel $k$, sofern es in $S$ ist
- SomeItem($S$) : $\{$*Stücke*$\}$ - liefert ein beliebiges Stück aus $S$
- ForAll $x$ in ($S$) do - erlaubt es, eine Operation auf alle Stücke in $S$ anzuwenden

Unter der Annahme, daß die Schlüssel eine geordnete Menge bilden, kann man zusätzlich die folgenden Funktionen einführen:

- Min($S$) : $\{$*Stück*$\}$ - liefert das Stück mit dem kleinsten Schlüssel aus $S$
- Max($S$) : $\{$*Stück*$\}$ - liefert das Stück mit dem größten Schlüssel aus $S$

| | | | | |
|---|---|---|---|---|
| Schlüssel | Info 1 | Info 2 | ... | Info n |

Abbildung 3.1: Item einer Datenstruktur

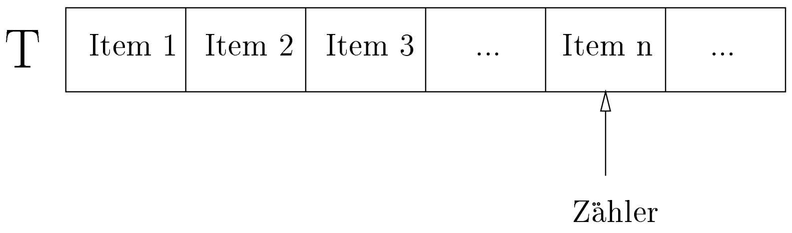



Abbildung 3.2: Feld $F$ mit $n$ Items

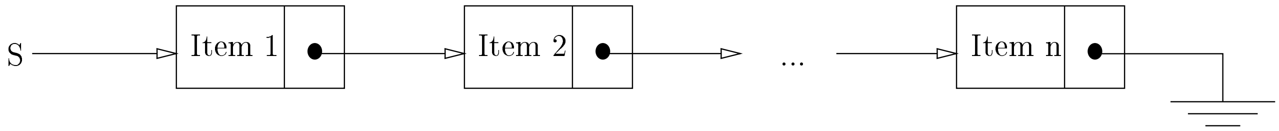



Abbildung 3.3: einfach verkettete Liste

- SUCCESSOR($S$,$k$) : {*Stück*} - liefert den kleinsten Schlüssel $m$ aus $S$, der größer als $k$ ist (Nachfolgerfunktion)
- PREDECESSOR($S$,$k$) : {*Stück*} - liefert den größten Schlüssel $m$ aus $S$, der kleiner als $k$ ist (Vorgängerfunktion)
- ENUMERATE IN ORDER($S$) : {*Stücke*} - gibt alle Stücke in $S$ aus, sortiert nach ihren Schlüsseln

In diesem Kapitel werden wir allerdings auf eine Schlüsselordnung verzichten und uns auf das reine Wörterbuchproblem konzentrieren, das nur die ersten sieben Operationen benötigt. Den Fall, daß die Schlüsselordnung wesentlich ist, werden wir im nächsten Kapitel behandeln.

## 3.2 Naive Methoden

Wir wollen nun ein paar einfache intuitive Methoden vergleichen, solche dynamischen Mengen zu verwalten.
Die einfachste ist sicher, die Stücke in den Anfangsteil eines genügend großen *statischen Feldes* $F[]$ abzulegen. Ein Zähler weist dabei immer auf das letzte benutzte Element von $F$ (siehe Abbildung 3.2). Ein einzufügendes Stück wird dann an der ersten freien Stelle von $F$ abgespeichert, und der Zähler wird um eins erhöht. Um ein Stück zu entfernen, wird an seine Stelle das letzte Stück im Feld umgespeichert, und der Zähler wird um eins erniedrigt.
Da eine statisches Feld eine fixe Größe hat, können in ihm nicht beliebig große Mengen abgespeichert werden. Dieses Problem wird typischerweise durch *verkettete Listen* umgangen, deren Länge je nach Anzahl der gespeicherten Stücke wächst, also theoretisch unbegrenzt ist. Die Liste kann auf zwei verschiedene Arten eine Speicherung der Elemente vornehmen, direkt oder indirekt. Bei der *direkten* Speicherung werden die kompletten Stücke eingefügt (siehe Abbildung 3.3), bei der *indirekten* Speicherung hingegen nur Zeiger auf die Stücke (siehe Abbildung 3.4), wobei es dann oft noch Zeiger von den Stücken zurück zu den entsprechenden Listenelementen gibt (in der Abbildung nicht gezeigt). Zusätzlich zur Unterscheidung zwischen direkten und indirekten Listen läßt sich eine Klassifizierung nach der Art der Verkettung durchführen. Hierbei unterscheidet man zwischen einfach und zweifach verketteter Liste. In der *einfach* verketteten Liste wird pro Element nur ein Zeiger auf das nachfolgende Element gespeichert. Bei der *zweifach* verketteten Liste werden pro Element zwei Zeiger hinzugefügt, wobei einer auf das Vorgängerelement und einer, wie bei der einfach verketteten Liste, auf das Nachfolgerelement zeigt (Abbildungen 3.5 und 3.6). Das Ende der Liste wird durch einen sogenannten Nullzeiger markiert. Zweifache Verkettung erlaubt das Entfernen eines durch einen Zeiger gegebenen Listenelementes in konstanter Zeit.
Schließlich gibt es noch die Methode der *direkten Adressierung.* Die setzt voraus, daß alle Schlüssel aus einem begrenzten *Universum* $U = \{0, \ldots, K-1\}$ stammen. Man kann dann eine Menge $S$ von Stücken in einer Tafel

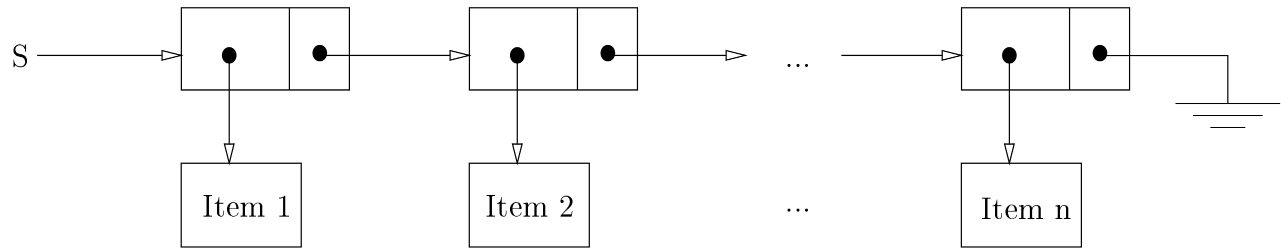



Abbildung 3.4: einfach verkettete, indirekte Liste

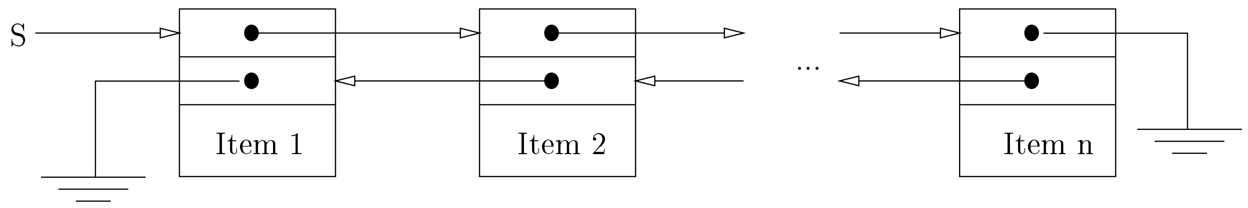



Abbildung 3.5: zweifach verkettete Liste

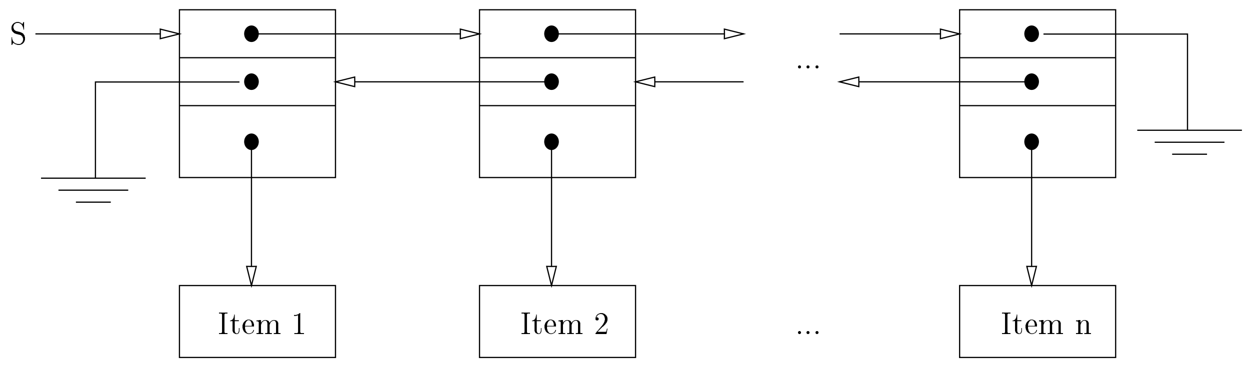



Abbildung 3.6: zweifach verkettete, indirekte Liste

$T[0..K-1]$ speichern, indem man ein Stück $x \in S$ mit Schlüssel $k$ in $T[k]$ ablegt. Es wird also über die Schlüssel direkt in die Tafel $T[]$ adressiert. Nicht besetzte Stellen von $T[]$ müssen natürlich als solche erkennbar gemacht sein.
Wir überlassen dem Leser die genaue Implementierung der sieben Wörterbuch Operationen für die vier angegebenen Realisierungen als leichte Übung. Sie/er möge sich von den Zeit- und Speicherschranken der folgenden Tabelle überzeugen (die Operation EMPTY? kann immer mit Hilfe eines zusätzlichen Zählers implementiert werden):

| Operation | statisches Feld | direkte Adressierung | einfach verkettete Liste | zweifach verkettete Liste |
|---|---|---|---|---|
| MAKESET | $O(1)$ | $O(K)$ | $O(1)$ | $O(1)$ |
| EMPTY? | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| INSERT | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| DELETE | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ |
| SEARCH | $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$ |
| SOMEITEM | $O(1)$ | $O(K)$ | $O(1)$ | $O(1)$ |
| FORALL | $O(n)$ | $O(K)$ | $O(n)$ | $O(n)$ |
| Speicherverbrauch | $\Omega(n)$ | $O(K)$ | $O(n)$ | $O(n)$ |

Offensichtlich ist die direkte Adressierung für solche Probleme gut geeignet, die häufiges Suchen erfordern. Ihre Nachteile liegen jedoch auf der Hand: hoher Zeitaufwand für SOMEITEM und FORALL (was allerdings mit Hilfe einer zusätzlichen doppelt verketten Liste umgangen werden könnte), hoher Speicherverbrauch, Notwendigkeit der Speicherinitialisierung[1] und begrenzte Anzahl verwendbarer Schlüssel. Hinzu kommt, daß höchstwahrscheinlich nur ein sehr geringer Teil des Feldes tatsächlich benutzt wird.
Man betrachte als Beispiel die Symboltabelle eines Compilers. Bei acht signifikanten Stellen und 42 in Variablennamen erlaubten Zeichen braucht man für direkte Adressierung ein Feld der Größe $42^8$. Ein Programm benutzt aber selten mehr als tausend dieser Namen. Daher wäre es wünschenswert, die direkte Adressierung mit einer der anderen Methoden zu kombinieren, so daß auch für große Universen der Speicherverbrauch gering ist.

## 3.3 Hashing

Eine Methode, die Probleme bei direkter Adressierung — jedenfalls teilweise — zu umgehen, stellt das *Hashing* dar. Sei $U$ ein Universum und $T[0..t-1]$ ein Feld, die sogenannte Hashtabelle oder auch Hashtafel. Weiterhin gebe es eine Hashfunktion $h : U \to \{0, \ldots, t-1\}$. Die Idee beim Hashing ist es, den Speicher nicht direkt, sondern über die Hashfunktion zu adressieren. Das Stück $x$ mit dem Schlüssel $k$ wird dabei in $T[h(k)]$ gespeichert.
Dabei tritt aber ein Problem auf. Zwei oder sogar mehrere Stücke können auf dieselbe Stelle hashen:

$$h(\mathit{Schlüssel}(x)) = h(\mathit{Schlüssel}(y)) .$$

Es gibt mehrere Möglichkeiten, so eine *Kollision* aufzulösen. Die einfachste ist wohl die der Verkettung: in die Hashtabelle werden nicht direkt Stücke eingetragen, sondern Zeiger auf verkettete Listen.
**Beispiel:** (siehe Abbildung 3.7)

$$S = \{27, 13, 12, 8, 10, 16\}, \quad t = 5, \quad h(k) = k \textbf{ mod } t .$$

Wenn man voraussetzt, daß $h$ in konstanter Zeit berechenbar ist, erhält man so ein Verfahren, daß die Vorteile von direkter Adressierung und verketteten Listen verbindet. Die Laufzeit von SEARCH hängt nun weitgehend von der Länge dieser Listen ab. Daher versuchen wir die Frage zu beantworten, wie lange eine solche Kette werden kann.
Der Einfachheit halber werden wir im folgenden nicht mehr zwischen Stücken und ihren Schlüsseln unterscheiden, sondern einfach nur über Schlüssel sprechen. Sei dazu $x \in U$ ein Schlüssel und $C_x$ die Anzahl der $y \in S$, die mit $x$ beim Hashen durch $h$ kollidieren. Wie groß wird dann $C_x$? Die Antwort liefert folgender

**Satz 2** *Sei $|U| = K$, $t$ die Größe der Hashtabelle und $h : U \to \{0..t-1\}$ eine Hashfunktion.*
*Dann gibt es für alle $n$, $0 \leq n \leq K/t$, einen Schlüssel $x$ und eine Teilmenge $S \subset U$ der Größe $n$, sodaß gilt $C_x = n$.*

[1] Wie sie zumindest theoretisch umgangen werden kann, siehe in Übung 4, Aufgabe 2.

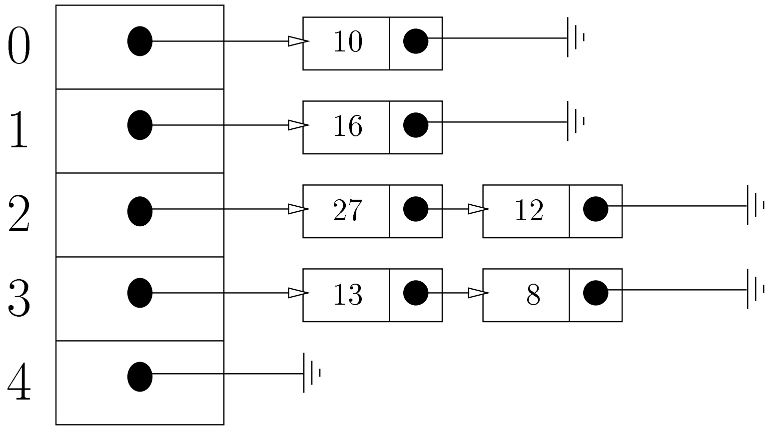



Abbildung 3.7: Beispiel für Hashtabelle

**Beweis** Nach dem Schubfachprinzip gibt es ein $i$, $0 \leq i < t$, sodaß $|\underbrace{h^{-1}(i)}_{=\{x \mid h(x)=i\}}| \geq \frac{|U|}{t}$ gilt. Wähle $S$ und $x$ aus $h^{-1}(i)$ passend aus. $\square$

Im schlechtesten Fall bringt das Hashing also keine Vorteile, weil alle Schlüssel auf denselben Feldeintrag hashen könnten. Daher brauchen wir eine Aussage darüber, welche Werte das $C_x$ im allgemeinen annimmt und nicht im schlechtesten Fall. Wir brauchen so etwas wie einen Erwartungswert von $C_x$. Um über so etwas wie einen Erwartungswert vernünftig sprechen zu können, sollten wir uns ein klein wenig mit Wahrscheinlichkeitstheorie beschäftigen.

## 3.4 Einschub: Wahrscheinlichkeitstheorie (Babyversion)

Sei $W$ eine endliche Menge. Für jedes $w \in W$ existiere ein $p_w \in [0,1]$ mit $\sum_{w \in W} p_w = 1$. Dann nennt man $W$ einen endlichen *Wahrscheinlichkeitsraum* und $p_w$ die *Wahrscheinlichkeitsverteilung* über $W$. Im einfachsten Fall, der *Gleichverteilung*, sind alle $p_w$ gleich, also $p_w = \frac{1}{|W|}$. Sei nun $E$ eine Teilmenge von $W$. Dann nennt man $E$ ein *Ereignis* und die Wahrscheinlichkeit von $E$ ist definiert als

$$\Pr(E) := \sum_{w \in E} p_w \,,$$

was im speziellen Fall der Gleichverteilung die klassische Definition von "Anzahl der günstigen durch Anzahl der möglichen Fälle" ergibt:

$$\Pr(E) = \frac{|E|}{|W|}.$$

Aus unserer Definition der Wahrscheinlichkeit folgen sofort die üblichen grundlegenden Eigenschaften, wie zum Beispiel

- $\Pr(\emptyset) = 0$
- $\Pr(W \setminus E) = 1 - \Pr(E)$
- $E_1 \subset E_2 \Longrightarrow \Pr(E_1) \leq \Pr(E_2)$
- $E_1 \cap E_2 = \emptyset \Longrightarrow \Pr(E_1 \cup E_2) = \Pr(E_1) + \Pr(E_2)$
- $\Pr(\bigcup_i E_i) \leq \sum_i \Pr(E_i)$

Eine *Zufallsvariable* ist eine Funktion

$$X : W \to \mathbb{R} \,.$$

Es wird also jedem *Element* des Wahrscheinlichkeitsraumes eine Zahl zugeordnet. Den *Erwartungswert* einer solche Zufallsvariable kann man im endlichen Fall definieren als

$$\mathrm{E}[X] = \sum_{w \in W} X(w) \cdot p_w .$$

Man beachte, daß bei Gleichverteilung der Erwartungswert nichts anderes ist als der Durchschnitt aller Zahlen $X(w)$.
Folgende und ähnliche Notation wird verwendet, um über Zufallsvariablen Ereignisse (also Teilmengen von $W$) zu definieren:

$$\begin{array}{lcl} (X = c) & : & \text{Ereignis } X^{-1}(c) = \{w \in W | X(w) = c\} \subset W \\ (X \leq c) & : & \text{Ereignis } \{w \in W | X(w) \leq c\} \end{array}$$

Mit dieser Notation ergibt sich eine alternative Definition für den Erwartungswert, die auch für unendliche Wahrscheinlichkeitsräume gilt und daher üblicherweise als primäre Definition verwendet wird:

$$\mathrm{E}[X] = \sum_{r \in \mathbb{R}} \Pr(X = r) \cdot r$$

Der Leser soll sich davon überzeugen, daß im endlichen Fall diese beiden Definitionen tatsächlich äquivalent sind.
Da Zufallsvariablen reellwertige Funktionen auf $W$ sind, macht es Sinn über Summen, Produkte und andere Operationen und Funktionen auf Zufallsvariablen zu sprechen: die entsprechenden Operationen werden einfach punktweise auf den Funktionswerten durchgeführt. Besonders wichtig für uns ist die Summenbildung von Zufallsvariablen, denn es gilt die Linearität des Erwartungswertes: Wenn $X = X_1 + X_2$ dann gilt $\mathrm{E}[X_1 + X_2] = \mathrm{E}[X_1] + \mathrm{E}[X_2]$. Für den Fall eines endlichen Wahrscheinlichkeitsraums ist der Beweis dieser Tatsache sehr einfach und wird dem Leser zur Übung überlassen.
Oft betrachten wir im folgenden Summen der Form $X = Y_1 + ... + Y_n$, bei denen die $Y_i$ sogenannte Indikatorvariablen (oder auch 0-1-Variablen) sind, d.h. $Y_i(w) \in \{0,1\}$ für alle $w \in W$. Dann gilt nämlich, daß

$$\mathrm{E}[Y] = \sum_{1 \leq i \leq n} \mathrm{E}[Y_i] = \sum_{1 \leq i \leq n} \Pr(Y_i = 1) .$$

Denn für eine 0-1-Variable $Y_i$ ergibt sich aus der zweiten Definition des Erwartungswertes

$$\mathrm{E}[Y_i] = \sum_{r \in \mathbb{R}} \underbrace{\Pr(Y_i = r)}_{=0 \ \forall r \neq 0,1} \cdot r = \Pr(Y_i = 0) \cdot 0 + \Pr(Y_i = 1) \cdot 1 = \Pr(Y_i = 1) .$$

Vorlesung vom 16.11.1995
Michael Eisemann, Marc Fontaine

Sei nun $X$ eine Zufallsvariable vom Wahrscheinlichkeitsraum in die natürlichen Zahlen, $X : W \to \mathbb{N}$. Dann läßt sich der Erwartungswert von $X$ auch durch

$$\mathrm{E}[X] = \sum_{n > 0} \Pr(X \geq n)$$

berechnen. Den Beweis dafür lassen wir als Übung.
Oft ist es von Interesse zu wissen, wie wahrscheinlich es ist, daß eine Zufallsvariable wesentlich von ihrem Erwartungswert abweicht. Die wohl einfachste Aussage darüber liefert das folgene Lemma:

**Lemma 2 (Markov Ungleichung)** *Sei $X$ eine Zufallsvariable, die nur nicht negative Werte annimmt. Dann gilt für jedes $c > 1$, daß*

$$\Pr(X \geq c\mathrm{E}[X]) \leq 1/c .$$

Man betrachtet alle Elemente $w$ eines Wahrscheinlichkeitsraums, deren Funktionswert $X(w)$ groß ist, und zwar mindestens $c\mathrm{E}[X]$ groß. Die Behauptung ist, daß es von diesen Elementen nicht viele geben kann. Als Intuition dazu kann man sich folgendes Beispiel überlegen: Wenn die Durchschnittsgröße der Studenten im Hörsaal 1.50m ist, dann können nicht mehr als die Hälfte der Studenten 3m groß sein.

**Beweis** : Wir führen den Beweis nur für den Sonderfall, daß $X$ nur ganzzahlige nicht negative Werte annimmt, und lassen den Beweis für den allgemeinen Fall als Übung.
Für $i \in \mathbb{N}$ sei $p_i = \Pr(X \geq i)$. Dann bilden die $p_i$'s eine fallende Folge. Für den Erwartungswert von $X$ gilt:

$$\mathrm{E}[X] = p_1 + p_2 + p_3 + \ldots + p_{\lceil c\mathrm{E}[X] \rceil} + \ldots$$

Angenommen, $p_{\lceil c\mathrm{E}[X] \rceil} > 1/c$. Dann gilt auch $p_j > 1/c$ für $1 \leq j \leq \lceil c\mathrm{E}[X] \rceil$. Also müßte gelten:

$$\mathrm{E}[X] \geq p_1 + p_2 + \ldots + p_{\lceil c\mathrm{E}[X] \rceil} > \frac{1}{c} \cdot \lceil c\mathrm{E}[X] \rceil \geq \mathrm{E}[X]\,,$$

ein offensichtlicher Widerspruch. Die Annahme war also falsch, das heißt, es muß $p_{\lceil c\mathrm{E}[X] \rceil} \leq 1/c$ gelten. $\square$

## 3.5 Hashing, noch einmal

Sei $U$ ein Schlüsseluniversum mit $|U| = K$, $T$ eine Hashtafel mit $|T| = t$ und $h : U \to \{0, \ldots, t-1\}$ eine Hashfunktion. Wir wollen versuchen, ein Stück mit Schlüssel $x \in U$ in $T[h(x)]$ zu speichern. Im folgenden werden wir nicht mehr zwischen Stücken und Schlüsseln unterscheiden.
Probleme treten bei Kollision $h(x) = h(y)$ für $x \neq y$ auf, d.h. wenn 2 Elemente unter derselben Speicheradresse in der Hashtabelle abgelegt werden. Zur Behebung dieses Problems gibt es folgende Lösungsmöglichkeiten, von denen wir nur die erste und dritte behandeln werden:

1. Verkettung
2. Offenes Adressieren (Es gibt mehrere Hashfunktionen. Kommt es bei einer Hashfunktion zur Kollision, probiere die Nächste.)
3. Perfektes Hashing (Angenommen man weiß, welche Menge abgespeichert werden soll. Dann wähle Hashfunktion so sorgfältig aus, daß es nicht zur Kollision kommt)

## 3.6 Hashing durch Verkettung

Die Elemente werden in der Hashtabelle in Listen abgelegt. Sucht man ein Element so schaut man mit Hilfe der Hashfunktion nach, in welche Liste man gehen soll. Die Hoffnung hierbei ist, daß die Listen nicht zu lang werden.
Methode zur Analyse: Sei $S$ die Menge der Schlüssel, die schon in der Liste abgespeichert sind, $S \subset U, |S| = n$. Für $x \in U$ sei $C_x$ die Anzahl der $y \in S$, die mit $x$ kollidieren, d.h.

$$C_x = |\{y \in S \mid h(x) = h(y)\}|.$$

Diese Zahl ist interessant, weil die wichtigsten Operationen für $x$ in folgender Zeit durchgeführt werden können.

- Insert $(x, S)$ kann in $O(1 + C_x)$ Zeit durchgeführt werden, unter der Voraussetzung, daß man nachsieht, ob der Schlüssel schon eingefügt wurde.
- Delete $(x, S)$ kann in $O(1 + C_x)$ Zeit durchgeführt werden.
- Search $(x, S)$ kann in $O(1 + C_x)$ Zeit durchgeführt werden.

Hierbei stellt die 1 in $O(1 + C_x)$ die grundlegende Annahme dar, daß die Hashfunktion in konstanter Zeit berechnet werden kann. Wie schon gezeigt, kann im schlechtesten Fall $C_x = n$ sein. Dann bietet Hashing keinen Vorteil. Uns interessiert daher die erwartete Laufzeit von Insert, Delete und Search, also insbesondere die *erwartete* Größe von $C_x$. Dafür muß $C_x$ zu einer Zufallsvariablen gemacht werden, also zu einer Funktion, die jedem Element des Wahrscheinlichkeitsraumes eine Zahl zuweist.
Was soll man nun als zugrundeliegenden Wahrscheinlichkeitsraum $W$ wählen? Aus der Definition von $C_x$ sieht man, daß $C_x$ von der Menge $S$ abhängt. Eine natürliche Wahl für den Wahrscheinlichkeitsraum wäre also

$$W = \binom{U}{n} = \{S \subset U \ : \ |S| = n\}\,.$$

Damit ist dann $C_x$ eine Zufallsvariable der Form

$$C_x : \binom{U}{n} \to \mathbb{N}.$$

Als Wahrscheinlichkeitsverteilung auf $W$ nehmen wir die Gleichverteilung an. Das heißt also, wir analysieren den Fall, daß jede Teilmenge $S \subset U$ mit gleicher Wahrscheinlichkeit in der Hashtafel abgespeichert wird. Ob dies ein realistischer Fall ist, bleibe dahingestellt.
Nach unserer Definition ist der Erwartungswert von $C_x$ gegeben durch

$$\mathrm{E}[C_x] = \frac{1}{\binom{K}{n}} \sum_{S \in \binom{U}{n}} C_x(S).$$

Diese Formel ist jedoch unangenehm zu analysieren. Deshalb verwenden wir folgenden Trick: wir zerlegen $C_x$ in einfache Zufallsvariablen, am einfachsten in Indikatorvariablen. Dazu definieren wir

$$C_{xy}(S) = \begin{cases} 1 & \text{falls } y \in S \text{ und } h(x) = h(y) \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt:

$$C_x(S) = \sum_{y \in U,\ y \neq x} C_{xy}(S).$$

Eine andere Art Indikatorvariablen zu definieren ist:

$$\Delta_y(S) = \begin{cases} 1 & \text{falls } y \in S \\ 0 & \text{sonst.} \end{cases}$$

Mit diesen läßt sich $C_x$ schreiben als:

$$C_x(S) = \sum_{\substack{y \in U \\ y \neq x \\ h(x)=h(y)}} \Delta_y(S).$$

Daraus folgt für den Erwartungswert:

$$\mathrm{E}[C_x] = \sum_{\substack{y \in U \\ y \neq x \\ h(x)=h(y)}} \mathrm{E}[\Delta_y].$$

Aber

$$\mathrm{E}[\Delta_y] = \Pr(\Delta_y = 1) = \Pr(y \in S) = \frac{|\{S \subset U \ : \ |S| = n,\ y \in S\}|}{|\{S \subset U \ : \ |S| = n\}|} = \frac{\binom{|U|-1}{n-1}}{\binom{|U|}{n}} = \frac{n}{|U|} = \frac{n}{K}.$$

Also gilt $E[\Delta_y] = n/K$. Angenommen, die Hashfunktion $h$ ist *fair*, d.h. für alle $0 \leq i < t$ gilt[2]

$$|\{x \in U | h(x) = i\}| \leq \left\lceil \frac{K}{t} \right\rceil.$$

Das bedeutet, daß die Schlüssel gleichmäßig auf die Hashtafel verteilt werden. Unter dieser Annahme hat die Summation höchstens $\lceil K/t \rceil$ Elemente. Also

$$\mathrm{E}[C_x] = \sum_{\substack{y \in U \\ y \neq x \\ h(x)=h(y)}} \mathrm{E}[\Delta_y] \leq \left\lceil \frac{K}{t} \right\rceil \cdot \frac{n}{K} \approx \frac{n}{t}.$$

Die Zahl $n/t$ wird in der Literatur oft als *Belegungsfaktor* bezeichnet. Damit gilt also der folgende Satz:

**Satz 3** *Sei $h$ eine faire Hashfunktion von einem Schlüsseluniversum $U$ in eine Hashtafel der Gröse $t$. Unter der Annahme, daß jedes $S \subset U$ mit $|S| = n$ mit der gleichen Wahrscheinlichkeit abgespeichert wird, können*

- INSERT $(x, S)$

[2] Ein typisches Beispiel für ein faires $h$ ist $h(x) = x \bmod t$.

- DELETE $(x, S)$
- SEARCH $(x, S)$

*in erwarteter Zeit $O(1 + n/t)$ durchgeführt werden.*

Wir haben also gezeigt, daß unter der Annahme, daß jedes $S$ gleichwahrscheinlich zum Abspeichern auftritt, Hashing im Durchschnitt gut funktioniert. Wir können allerdings überhaupt nicht verifizieren, ob diese Gleichverteilungsannahme gerechtfertigt ist. Man kann auch leicht Situationen nennen, in denen diese Annahme sicher nicht gilt: Zum Beispiel werden bestimmte Kombinationen von Buchstaben häufiger als Variablennamen verwendet und daher in der Symboltafel eines Compilers abgespeichert als andere; es gibt wahrscheinlich sehr wenig Programme, die den Variablennamen *x2* enthalten, aber nicht den Variablennamen *x1*.
Wenn man die Gleichverteilungsannahme auf $W = \binom{U}{n}$ ändern will, ist es nicht klar, worauf sie geändert werden soll. Und dann steht man erst wieder vor dem Rechtfertigungsproblem der neuen Annahme (und vor dem Problem, etwas Vernünftiges zu beweisen).
Oft hilft man sich, indem man implizite Annahmen macht, wie ungefähr die folgende: Man wählt sich eine Hashfunktion $h$ fix aus und nimmt dann an, daß die $S$ mit einer solchen Wahrscheinlichkeit auftreten, daß $\mathrm{E}[C_{xy}] \leq c \cdot 1/t \cdot n/K$ gilt, wobei $c$ irgendeine Konstante ist. Daraus folgt dann die gewünschte erwartete Größe der $C_x$:

$$\mathrm{E}[C_x] = \sum_{y \in U} \mathrm{E}[C_{xy}] \leq K \cdot c \cdot \frac{1}{t} \cdot \frac{n}{K} \leq c \cdot \frac{n}{t}\,.$$

Typischerweise wird als Wahl für die Hashfunktionen $h$

$$h(x) = x \bmod t,$$

verwendet, wobei $t$ möglichst weit weg von einer 2'er Potenz sein soll, oder

$$h(x) = \lfloor t(Ax \bmod 1) \rfloor\,,$$

wobei $A$ irrational sein soll, wie z.B. $A = \frac{\sqrt{5}-1}{2} \approx 0,6180$. Für weitere Information über diesen Ansatz verweisen wir den Leser auf Donald E. Knuth's **Sorting and Searching** Band 3 von **The Art of Computer Programming**, Addison-Wesley 1973.

## 3.7 Universelles Hashing

Um dem Dilemma zu entgehen, welche Wahrscheinlichkeitsverteilung man für die möglichen Eingabemengen annehmen soll, ändern wir unseren Ansatz. Aus der Definition von $C_x$ sieht man, daß $C_x$ nicht nur von der Menge $S$, sondern auch von der Hashfuntion $h$ abhängt. Das legt nahe, das $S$ zu fixieren, und $C_x$ als Funktion über einer Menge von Hashfunktionen zu betrachten. Unser Wahrscheinlichkeitsraum wird nun eine Menge $\mathcal{H}$ von Funktionen $h : U \to \{0, \ldots, t-1\}$, und wir nehmen darauf eine Gleichverteilung an.
Dies ist unter Umständen sinnvoller als der erste Ansatz: Eine bestimmte Menge $S$ soll gespeichert werden. Um das zu tun wählt sich der Algorithmus zufällig eine Hashfunktion $h \in \mathcal{H}$ aus und verwendet sie dann, um $S$ abzuspeichern. Manche der $h$'s werden sich gut dafür eignen, manche schlecht. Die Hoffnung ist, daß sich ein "durchschnittliches" $h \in \mathcal{H}$ gut dafür eignen wird. Man beachte, daß der Zufall ganz unter die Kontrolle des Algorithmus selbst geraten ist. Man muß nicht mehr irgendetwas über die Verteilung der möglichen Eingaben annehmen, man muß nur annehmen, daß der Algorithmus zufällig auswählen kann.
Ob ein "durchschnittliches" $h \in \mathcal{H}$ wirklich "gut geeignet" ist, hängt natürlich stark von der Wahl von $\mathcal{H}$ ab. Wir überlassen es dem Leser als Übung, zu zeigen, daß im Fall $\mathcal{H} = \{0, \ldots, t-1\}^U$, die Menge aller Funktionen von $U$ nach $\{0, 1, \ldots, t-1\}$, dies tatsächlich der Fall ist. Ein typisches $h \in \{0, \ldots, t-1\}^U$ kann allerdings nicht in konstanter Zeit berechnet werden. Man muß sich also eine andere Funktionsmenge einfallen lassen.

**Definition 2** $\mathcal{H} \subset \{0, \ldots, t-1\}^U$ *heißt eine Menge von* universellen Hashfunktionen, *wenn für alle $x, y \in U$ mit $x \neq y$ gilt:*

$$|\{h \in \mathcal{H} | h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{t}$$

Wir werden nun zeigen, daß ein "durchschnittliches" $h$ aus so einer universellen Menge tatsächlich gut geeignet ist. Danach werden wir zeigen, daß es solche Mengen von universellen Hashfunktionen tatsächlich gibt.
Sei nun $S \subset U$ fix. Für jedes $x, y \in U$ mit $x \neq y$ definieren wir eine 0-1-Zufallsvariable $\delta_{xy} : \mathcal{H} \to \mathbb{R}$ durch

$$\delta_{xy}(h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst.} \end{cases}$$

Damit gilt

$$C_x(h) = \sum_{y \in S} \delta_{xy}(h)\,.$$

$C_x$ ist also jetzt eine Zufallsvariable über dem Wahrscheinlichkeitsraum $\mathcal{H}$. Wegen der Linearität des Erwartungswertes gilt

$$\mathrm{E}[C_x] = \sum_{y \in S} \mathrm{E}[\delta_{xy}] \leq \sum_{y \in S} \frac{1}{t} = \frac{|S|}{t} = \frac{n}{t}\,,$$

da auf Grund der Universalität von $\mathcal{H}$ gilt

$$\mathrm{E}[\delta_{xy}] = \Pr(\delta_{xy} = 1) = \Pr(h(x) = h(y)) \leq \frac{|H|/t}{|H|} = \frac{1}{t}.$$

Das Ergebnis sieht also genauso aus wie vorher, aber die Annahmen sind ganz verschieden. Beim ersten Ansatz gab es eine Annahme über die Eingabe. Jetzt liegt die Annahme in der Hashfunktion, die der Algorithmus wählt. Es gilt also folgender Satz:

**Satz 4** *Sei $S \subset U$ mit $|S| = n$ und sei $\mathcal{H}$ eine Menge von universellen Hashfunktionen $U \to \{0, 1, \ldots, t-1\}$. Wenn zum Speichern von $S$ in einer Hashtafel der Größe $t$ eine zufällig aus $\mathcal{H}$ gewählte Hashfunktion $h$ verwenden wird, dann werden*

- INSERT $(x, S)$
- DELETE $(x, S)$
- SEARCH $(x, S)$

*in erwarteter Zeit $O(1 + n/t)$ durchgeführt.*

Es bleibt nun zu zeigen, daß es solche Mengen von universellen Hashfunktionen gibt. Hier ist eine besonders einfache Konstruktion. Wir nehmen an, daß $t$ eine Primzahl ist und daß jedes $x \in U$ in eindeutiger Weise als $(r+1)$-Tupel $\overline{x} = \langle x_0, x_1, \ldots, x_r \rangle$ geschrieben werden kann, mit $r > 0$ und $0 \leq x_i < t$ für alle $i$ (z.B. sei $t = 257$ und $x$ wird einfach byteweise geschrieben).
Für jedes $\overline{a} = \langle a_0, \ldots, a_r \rangle \in \{0, \ldots, t-1\}^{r+1}$ definieren wir eine Hashfunktion $h_{\overline{a}} : U \to \{0, \ldots, t-t\}$

$$h_{\overline{a}}(x) = (\sum_{0 \leq i \leq r} a_i x_i) \bmod t\,.$$

**Lemma 3** *Die Menge*

$$\mathcal{H} = \{h_{\overline{a}} \mid \overline{a} \in \{0, \ldots, t-1\}^{r+1}\}$$

*ist universell.*

**Beweis** Seien $x, y \in U$, $x \neq y$. O.B.d.A. nehmen wir an, daß sich $\langle x_0, \ldots, x_r \rangle$ und $\langle y_0, \ldots, y_r \rangle$ an der ersten Stelle unterscheiden, also $x_0 \neq y_0$. Wann gilt $h_{\overline{a}}(x) = h_{\overline{a}}(y)$, also

$$\sum_{0 \leq i \leq r} a_i x_i \equiv \sum_{0 \leq i \leq r} a_i y_i \pmod{t}\,?$$

Damit es zu einer Gleichheit kommt, muß es zu der folgenden Gleichheit kommen

$$a_0 \underbrace{(x_0 - y_0)}_{\neq 0} \equiv \sum_{1 \leq i \leq r} a_i (y_i - x_i) \pmod{t}\,.$$

Bei gegebenen $x$ und $y$ ergibt jede Wahl von $a_1$ bis $a_r$ irgendeine feste Zahl $C$ auf der rechten Seite. Da $t$ eine Primzahl ist, gibt es dann nur eine Möglichkeit $a_0$ zu wählen, sodaß die Gleichung wahr wird. Das heißt, die Anzahl der $\overline{a}$'s und daher die Anzahl der Funktionen in $\mathcal{H}$, die zur Gleichheit führen, ist

$$t^r = \frac{|\mathcal{H}|}{t} = \frac{t^{r+1}}{t},$$

wie verlangt. $\square$

Vorlesung vom 20.11.1995
Ludger Bünger, Holger Sabo

## 3.8 Perfektes Hashing

Beim universellen Hashing können natürlich Kollisionen auftreten und diese werden mit Hilfe von verketteten Listen behandelt. Besser wäre es, wenn sich Kollisionen vollkommen vermeiden ließen.

Für ein gegebens $S \subset U$ fix hätte man also gerne eine Hashfunktion $h : U \to \{0, \dots, t-1\}$, die $S$ ohne Kollision in einem Feld $T$ abspeichert, oder mathematisch ausgedrückt: $h|_S$ soll injektiv sein. Möchte man dann nach einem Stück mit Schlüssel $x$ suchen, muß man nur an der Stelle $T[h(x)]$ nachsehen, ob dort ein Stück gespeichert ist und ob dieses Stück den gewünschten Schlüssel hat. Durch perfektes Hashing wird also garantiert, daß sich der gesuchte Schlüssel an der gelieferten Adresse befindet, oder im Feld nicht vorhanden ist. Damit wird die Anzahl von Schlüsselvergleichen, die für eine Suche notwendig sind, auf eins reduziert und außerdem wird eine konstante Suchzeit *im schlechtesten Falle* garantiert.

Ein solches "perfektes" $h$ existiert natürlich immer (unter der Voraussetzung, daß $t \geq |S|$ ist — sonst wäre nicht genügend Platz vorhanden). Für unsere Zwecke muß $h$ aber auch in konstanter Zeit berechenbar sein. Im folgenden werden wir zeigen, daß sich so ein in konstanter Zeit berechenbares, perfektes $h$ zusammenbasteln läßt, sobald man universelle Mengen von Hashfunktionen zur Verfügung hat.

### 3.8.1 Einstufiges perfektes Hashing

Sei für die folgenden Betrachtungen $\mathcal{H}$ eine Menge von universellen Hashfunktionen $U \to \{0, \dots, t-1\}$ und $S$ fix, $n = |S|$. Wenn wir Glück haben, treten für ein $h \in \mathcal{H}$ keine Kollisionen auf, wenn es verwendet wird um $S$ abzuspeichern. Im allgemeinen wird das aber wohl nicht der Fall sein. Wir benötigen aber auf jeden Fall ein Maß für die Anzahl der Kollisionen.

Für $h \in \mathcal{H}$ sei $C_S(h)$ die Anzahl der Kollisionen zwischen Schlüsseln in $S$ unter Verwendung von $h$. $C_S$ ist also ein Funktion $\mathcal{H} \to \mathbb{N}$ und kann daher als Zufallsvariable über dem Wahrscheinlichkeitsraum $\mathcal{H}$ interpretiert werden. Mit Hilfe der oben eingeführten Indikatorfunktionen $\delta_{xy}$ kann man $C_S$ schreiben als

$$C_S(h) = \sum_{\{x,y\} \subset S} \delta_{xy}(h)$$

Der Erwartungswert von $C_S$ läßt sich nun auf Grund seiner Linearität leicht berechnen. Es gilt:

$$\mathrm{E}[C_S] = \sum_{\{x,y\} \subset S} \underbrace{\mathrm{E}[\delta_{xy}]}_{\leq 1/t} \quad \leq \quad \binom{n}{2}/t,$$

da $\mathrm{E}[\delta_{xy}] \leq 1/t$ wegen der Universalität von $\mathcal{H}$ und da es $\binom{n}{2}$ Teilmengen $\{x, y\} \subset S$ gibt.
Das heißt also, der Erwartungswert von $C_S$ kann beliebig klein gemacht werden, wenn nur die Tafelgröße $t$ groß genug gewählt wird. Zwei Fälle sind nun vorerst von Interesse:

- $t > \binom{n}{2}$: Dann gilt $\mathrm{E}[C_S] < 1$, woraus aber sofort die Existenz eines $h \in \mathcal{H}$ folgt mit $C_S(h) = 0$. Denn $\mathrm{E}[C_S] < 1$ bedeutet, daß der Durchschnitt der $C_S(h)$'s kleiner als 1 ist. Also muß es ein $h \in \mathcal{H}$ geben mit $C_S(h) < 1$ und da $C_S$ nur Werte aus $\mathbb{N}$ annimmt, bedeutet dies $C_S(h) = 0$.

- $t > 2\binom{n}{2}$: Dann gilt $\mathrm{E}[C_S] < 1/2$, woraus wiederum folgt, daß für mindestens die Hälfte der $h \in \mathcal{H}$ gilt $C_S(h) = 0$. Denn für die anderen $h$ gilt

$$\Pr(C_S \geq 1) \;=\; \Pr(C_S \geq \frac{1}{\mathrm{E}[C_S]} \cdot \mathrm{E}[C_S]) \overset{\text{Markovsche Ungl.}}{\leq} 1/\frac{1}{\mathrm{E}[C_S]} \;=\; \mathrm{E}[C_S] < 1/2\,,$$

  also ist der Anteil der $h$'s in $\mathcal{H}$ mit $C_S(h) \geq 1$ kleiner als die Hälfte.

Das bedeutet, daß, wenn die Hashtabelle genügend groß gewählt wird ($t > \binom{n}{2}$), man in $\mathcal{H}$ sicher eine Funktion $h$ findet, die die Elemente von $S$ ohne Kollision abbildet. Man braucht nur alle Funktionen in $\mathcal{H}$ durchzuprobieren und auf Kollisionsfreiheit zu überprüfen.
Wenn die Hashtafel noch größer gemacht wird ($t > 2\binom{n}{2}$), sind mindestens die Hälfte der Funktionen $h$ kollisionsfrei für $S$. Wählt man also eine Funktion $h$ zufällig, so ist diese mit Wahrscheinlichkeit mindestens $1/2$ kollisionsfrei, was in $O(n)$ Zeit verifiziert werden kann. Die erwartete Anzahl der Wahlwiederholungen, bis man auf ein kollisionsfreies $h$ stößt, ist daher höchstens zwei. Damit haben wir eine teilweise Lösung des Perfect-Hashing-Problems.

**Satz 5** *Sei $S \subset U$, $|S| = n$ und $\mathcal{H}$ eine universelle Menge von Hashfunktionen $U \to \{0, \ldots, t-1\}$.*

1. *Ist $t > \binom{n}{2}$, dann enthält $\mathcal{H}$ eine Funktion $h$ mit $h|_S$ injektiv, d.h. $S$ wird durch $h$ ohne Kollision abgespeichert. Wichtig: So ein $h$ kann in Zeit $O(|\mathcal{H}| \cdot n)$ gefunden werden (da maximal alle $h \in \mathcal{H}$ mit allen Elementen in $S$ getestet werden müssen).*

2. *Ist $t > 2\binom{n}{2}$, dann gilt für mindestens die Hälfte der $h \in \mathcal{H}$, daß $h|_S$ injektiv ist. Wichtig: So ein $h$ kann durch einen randomisierten Algorithmus in $O(n)$ (bzw. $O(n+t)$ mit Initialisierung) erwarteter Zeit gefunden werden.*

Das perfekte Hashing-Problem ist hiermit leider noch nicht zufriedenstellend gelöst, da $O(n^2)$ Speicher für die Hashtafel benötigt wird.

### 3.8.2 Zweistufiges perfektes Hashing

Um den Speicherbedarf auf $O(n)$ zu reduzieren betrachten wir zuerst einmal zwei weitere Konsequenzen von $\mathrm{E}[C_S] \leq \binom{n}{2}/t$.

- $t > \frac{n-1}{2}$: Dann gilt $\mathrm{E}[C_S] \leq n$ und daher gibt es ein $h \in \mathcal{H}$ mit $C_S(h) \leq n$.

- $t > n-1$: Dann gilt $\mathrm{E}[C_S] \leq n/2$ und durch die Markov Ungleichung folgt dann wieder, daß $C_S(h) \leq n$ für mindestens die Hälfte der $h$'s in $\mathcal{H}$, da

$$\Pr(C_S \geq n) \;=\; \Pr(C_S \geq \frac{n}{\mathrm{E}[C_S]} \cdot \mathrm{E}[C_S]) \;\leq\; 1/\frac{n}{\mathrm{E}[C_S]} = \mathrm{E}[C_S]/n \leq 1/2\,.$$

Zur Nutzung dieser Konsequenzen benötigen wir eine andere Interpretation für $C_S$. Definieren wir $S_i(h) = \{x \in S | h(x) = i\}$ für $0 \leq i < t$ und sei $B_i(h) = |S_i(h)|$, also die Anzahl der Elemente von $S$, die auf das gleiche Feld $i$ in der Hashtafel abgebildet werden. Für jedes $h$ gilt nun

$$C_S(h) = \sum_{0 \leq i < t} \binom{B_i(h)}{2}\,,$$

da man $C_S$ als Summe über alle Felder $i$ der Kollisionen aller Paare im Feld $i$ darstellen kann.

Aus dem vorher gesagten folgt nun, daß bei $t > n-1$ für die Hälfte der $h \in \mathcal{H}$ gilt:

$$\sum_{0 \leq i < t} \binom{B_i(h)}{2} \leq n\,.$$

Warum ist das interessant? Sagen wir, wir hätten so eine Funktion $h$. Betrachten wir Kübel $i$ der Hashtafel: Anstatt die $B_i(h)$ Stücke von $S_i(h)$ in einer verketteten Liste abzuspeichern, könnte man auch perfektes Hashing

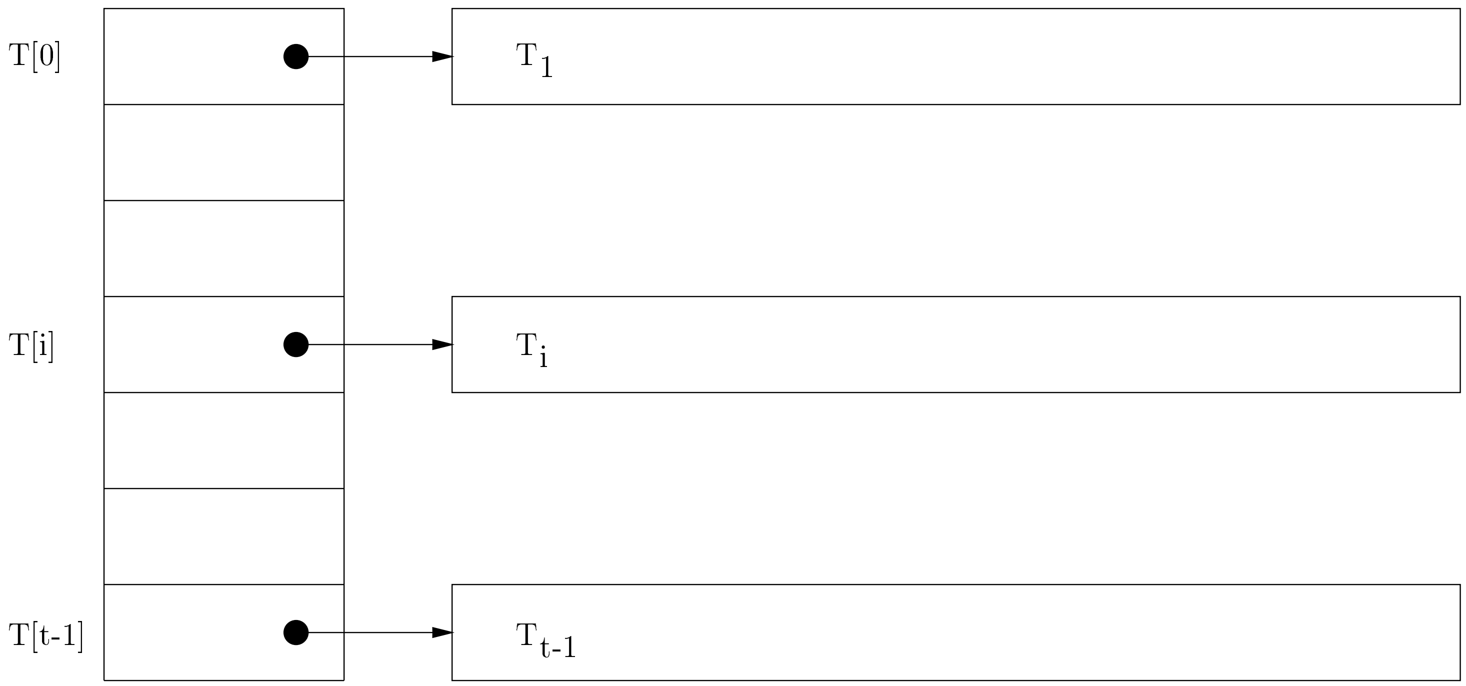



Abbildung 3.8: Schema des zweistufigen Hashings

nach dem vorherigen Satz 5 benutzen mit einer Hashtafelgröße von $t_i = 2\binom{B_i(h)}{2} + 1$. Für die Summe der Größen dieser sekundären Hashtafeln gilt dann

$$\sum_{0 \leq i < t} t_i = \sum_{0 \leq i < t} \left[ 2 \cdot \binom{B_i(h)}{2} + 1 \right] \leq 2n + t$$

Bei einer Wahl von $t = n$ brauchen diese sekundären Hashtafeln also insgesamt nur Platz $3n$.
Die wesentliche Idee des zweistufigen perfekten Hashing Verfahrens ist also, anstatt die Kollisionen durch verkettete Listen aufzulösen noch einmal Hashing zu verwenden.

Hier ist der Algorithmus für das **Zweistufige perfekte Hashverfahren:**
Gegeben ist $S \subset U$, $|S| = n$.

1. Wähle $t = n$. Sei $\mathcal{H}$ eine Menge von universellen Hashfunktionen $U \to \{0, \ldots, t-1\}$
2. Wähle $h \in \mathcal{H}$ zufällig und berechne $S_i(h)$, $B_i(h) = |S_i(h)|$. Wiederhole dies solange bis ein $h$ gefunden wurde mit $\sum_{0 \leq i < t} \left[ 2 \cdot \binom{B_i(h)}{2} + 1 \right] \leq 3n$.
3. Für jedes $i$ wähle $t_i = 2 \cdot \binom{B_i(h)}{2} + 1$, und sei $\mathcal{H}_i$ eine Menge von universellen Hashfunktionen $U \to \{0, \ldots, t_i - 1\}$
4. Für jedes $i$ wähle $h_i$ so lange zufällig aus $\mathcal{H}_i$ aus, bis $h_i|_{S_i}$ injektiv ist.
5. Baue primäre Hashtafel $T[0 \ldots t-1]$ und sekundäre Hashtafeln $T_i[0 \ldots t_i - 1]$ ($T[i]$ zeigt auf $T_i$), sowie eine Tabelle $H[0 \ldots t-1]$, die die $h_i$ speichert. Verwende $h_i$ um $S_i$ in $T_i$ perfekt zu speichern

Die einzelnen Schritte des Algorithmus besitzen folgende erwartete Komplexität:

| Schritt | Zeit |
|---|---|
| 1 | $O(n)$ |
| 2 | $O(n)$ |
| 3 | $\sum_i B_i + t_i = O(n)$ |
| 4 | insges.: $O(n)$ |
| 5 | |

Hieraus ergibt sich unmittelbar folgender Satz:

**Satz 6** *Mit Hilfe des zweistufigen Verfahrens kann eine perfekte Hashtafel der Größe $O(n)$ in erwarteter Zeit $O(n)$ erzeugt werden.*

Der Zugriff auf ein Element $x$ findet folgendermaßen statt:

- Anwendung von $h$ gibt sekundäre Tafel $T_i$ und sekundäre Hashfunktion $h_i$ zurück.
- Anwendung von $h_i$ in $T_i$ gibt entweder das Element $x$ zurück oder, daß $x$ nicht in unserer Hashtafel abgelegt ist.

Vorlesung vom 23.11.1995
Torsten Weller, Volker Morbach, Mirko Jerrentrup

## 3.9 Voll-Dynamisches Hashing

Wenn man a priori nicht weiß, wie groß die Menge $S$ sein soll, muß man die Größe der Hashtafel variierbar machen. Um bei dynamischer Hashtafel konstante Suchzeit zu erreichen, muß ihre Größe zu jedem Zeitpunkt durch die Größe der Menge linear beschränkt sein; d. h. es muß $|S|/t = O(1)$ gelten, denn die zum Suchen benötigte erwartete Zeit ist $O(1 + |S|/t)$.

Idee:
Ziel ist es nun, daß sich die Größe der Hashtabelle der Kardinalität von $S$ anpaßt; mit anderen Worten, wenn $|S|$ über die Größe der aktuellen Hashtabelle hinauswächst, muß sich diese vergrößern. Wenn $|S|$ unter eine noch näher zu bestimmende untere Grenze absinkt, muß sich die Hashtafel entsprechend verkleinern. Dieses Vorgehen nennt man *Rehashing*. Natürlich kostet das Vergrössern und Verkleinern der Hashtabelle Zeit, daher versucht man, möglichst selten zu rehashen, damit die dafür aufgewandte Zeit im Verhältnis zu der für die *search*- und *update*-Operationen notwendige Zeit vernachlässigbar ist.

Realisierung:
Sei $i$ so, daß $2^{i-2} \leq |S| < 2^i$ gilt.
Man verwendet eine Hashtabelle der Größe $t = 2^i$, damit sich $|S|$ immer zwischen $t$ und $t/4$ bewegt. Rehashing wird wie folgt realisiert:

- $|S|$ erreicht $2^i \Longrightarrow$ ersetze die Hashtabelle durch eine neue mit Größe $t = 2^{i+1}$ und erhöhe dann $i$ um 1.
- $|S|$ unterschreitet $2^{i-2} \Longrightarrow$ ersetze die Hashtabelle durch eine neue mit Größe $t = 2^{i-1}$ und erniedrige dann $i$ um 1.

Klarerweise braucht ein Rehash $O(t + |S|)$, also $O(|S|)$ erwartete Zeit, da immer $t = \Theta(|S|)$ gilt.

**Behauptung 6** *Sei $Z$ eine Folge von $m$ Such- bzw. Update-Operationen in einer ursprünglich leeren Hashtabelle. Die erwartete Zeit zur Abarbeitung von $Z$ ist $O(m)$.*

**Beweis** Wenn man die Kosten des Rehashings vernachlässigt, dann ist die Behauptung klar, denn mit $t$ immer ungefähr so groß wie $|S|$ bekommt man eine erwartete Zeit von $O(1 + |S|/t) = O(1)$ pro Such- und Update-Operation und damit $O(m)$ erwartete Zeit für diese höchstens $m$ Operationen in $Z$. Wir wollen nun zeigen, daß die erwartete Gesamtzeit für alle Rehashoperationen auch $O(m)$ nicht überschreitet.
Bezeichnen wir Insert-, Delete-, Search-Operationen in der Folge $Z$ jeweil mit $I$, $D$, und $S$, und fügen wir auch ein $R$ in die Folge ein, wenn jeweils eine Rehashoperation passiert. Betrachten wir nun eine Teilfolge von $Z$ zwischen zwei aufeinanderfolgenden Rehashes:

$$\underbrace{R}_{|S|=2^{i-1}} \; \underbrace{S \;\; S \;\; I \;\; I \;\; D \;\; I \;\; D \ldots}_{\#I's \geq 2^i - 2^{i-1} = 2^{i-1} = \Theta(t)} \; \underbrace{R}_{|S|=2^i}$$

Wenn das zweite Rehash die Hashtabelle auf $2^{i+1}$ vergrößert, dann muß es seit dem vorangegangenen Rehash mindestens $2^i - 2^{i-1}$, also mindestens $2^{i-1}$ Operationen gegeben haben..

$$\underbrace{R}_{|S|=2^{i-1}} \; \underbrace{S \;\; S \;\; I \;\; I \;\; D \;\; I \;\; D \ldots}_{\#D's \geq 2^{i-1} - 2^{i-2} = 2^{i-2} = \Theta(t)} \; \underbrace{R}_{|S|=2^{i-2}}$$

Wenn das zweite Rehash die Hashtabelle auf $2^{i-1}$ verkleinert, dann muß es seit dem vorangegangenen Rehash mindestens $2^{i-1} - 2^{i-2}$, also mindestens $2^{i-2}$ Operationen gegeben haben.
Das bedeutet, erzeugt ein Rehash eine Hashtabelle der Größe $t$, dann war die Anzahl der Search- und Update-Operationen davor $\Theta(t)$.
Für eine Folge von $m$ Search- und Update-Operationen und entprechend vielen Rehash-Operationen gilt dann: Der Zeitbedarf der Search- und Update-Operationen ist $O(m)$, der Zeitbedarf der Rehash-Operationen ist jeweils $O($*Länge der vorhergehenden Folge von Operationen*$)$.

$$R \underbrace{\ldots\ldots}_{l_1} \overbrace{R}^{O(l_1)} \underbrace{\ldots\ldots}_{l_2} \overbrace{R}^{O(l_2)} \underbrace{\ldots\ldots}_{l_3} \overbrace{R}^{O(l_3)} \underbrace{\ldots\ldots}_{l_4} \overbrace{R}^{O(l_4)} \ldots$$

Der Zeitbedarf aller Rehash-Operationen ist $O(\sum l_i) = O(m)$ wegen $\sum l_i \leq m$. □

### 3.9.1 Amortisierte Kosten

Sei $O_1, O_2, \ldots O_m$ eine Folge von Operationen, die insgesamt in der Zeit $T(m)$ abgearbeitet wird. Die *amortisierten Kosten* der Operation $O_i$ sind definiert als $\frac{T(m)}{m}$, d. h. die Gesamtkosten werden auf die Operationen gleichmäßig verteilt. Nach dieser Definition sind die *amortisierten Kosten* einer Rehash-Operation $O(1)$ im erwarteten Fall, und man kann mit dieser Terminologie folgende Aussage treffen:

**Satz 7** *Bei voll-dynamischem Hashing (mit Verketten) benötigt:*

- SEARCH — $O(1)$ *amortisierte erwartete Zeit*; $O(n)$ *Zeit im schlechtesten Fall*
- UPDATE — $O(1)$ *amortisierte erwartete Zeit*; $O(n)$ *Zeit im schlechtesten Fall*

*Der Speicherbedarf ist* $O(n)$.

Bemerkung: Man kann auch perfektes Hashing in ähnlicher Weise voll-dynamisch realisieren, siehe DIETZFELBINGER ET AL. *"Dynamic Perfect Hashing"* SIAM Journal on Computing 23 (1994) 738–761.

# Kapitel 4

# Mengen mit Schlüsseln aus Ordnungen

## 4.1 Binäres Suchen

Wenn eine Ordnung auf der Schlüsselmenge existiert, hat man die Möglichkeit, neben SEARCH, INSERT und DELETE-Operationen noch Funktionen wie MIN, MAX, PREDECESSOR und SUCCESSOR zu definieren. Außerdem kann die Ordnung verwendet werden, um bei einer Suche zu helfen. Das typische Beispiel hierzu ist ein Telefonbuch: Die Namen der Teilnehmer sind darin alphabetisch geordnet verzeichnet, und bei der Suche nach einem Namen verwendet man diese Ordnung. Typischerweise sucht man ein Telefonbuch nicht linear durch, sondern mit Sprüngen. Die Abstraktion davon ist die sogenannte *binäre Suche*.
Idee:
Die Menge $S$ wird in einem Feld $F[0..n-1]$ nach den Schlüsseln geordnet abgelegt. Binäre Suche versucht den Bereich, in der ein gesuchter Schlüssel liegen muß, immer mehr einzuschränken. Die folgende Prozedur sucht nach einem Eintrag mit Schlüssel $K$ im Feld $F$ im Bereich $F[a..b]$:

```
function BinSearch(K, F, a, b)
  if a = b then
    if F[a].key = K then return a
    else return FAILURE
  m = ⌈(a + b)/2⌉
  if K < F[m].key then
    return BinSearch(K, F, a, m − 1)
  else
    return BinSearch(K, F, m, b)
```

Auf ein Feld der Länge $n$ angewandt, beträgt die Laufzeit $T(n)$ von BinSearch $O(\log n)$, denn nach jedem Vergleich wird die Länge des Suchbereichs halbiert: $T(n) = 1 + T(\frac{n}{2})$.

Bei so einer einfachen geordneten Felddarstellung brauchen die Updateoperationen INSERT und DELETE $O(n)$ Zeit, da jedesmal ein guter Teil des Feldes verschoben werden muß. Darum ist es im folgenden unser Ziel, das Potential der binären Suche zu wahren, während schnelle Update Operationen auch möglich sein sollen. Dies führt uns zum Konzept der Suchbäume.

## 4.2 Suchbäume

Sei $T$ ein binärer Baum mit $n$ Knoten. Die Menge $S$ wird folgendermaßen in $T$ gespeichert: Jeder Knoten von $T$ bekommt genau ein Element von $S$; für jeden Knoten $x$ in $T$ gilt, daß alle Knotem im linken Unterbaum von $x$ kleinere Schlüssel haben als der von $x$ und alle Knoten im rechten Teilbaum größere Schlüssel haben als der von $x$ (siehe Beispiel 4.1). Man sagt, daß die Schlüssel in *symmetrischer Ordnung* in $T$ gespeichert sind.
Wir vereinbaren, daß jeder Knoten folgende Komponenten enthält:

- *key:* enthält den Schlüssel des Knotens

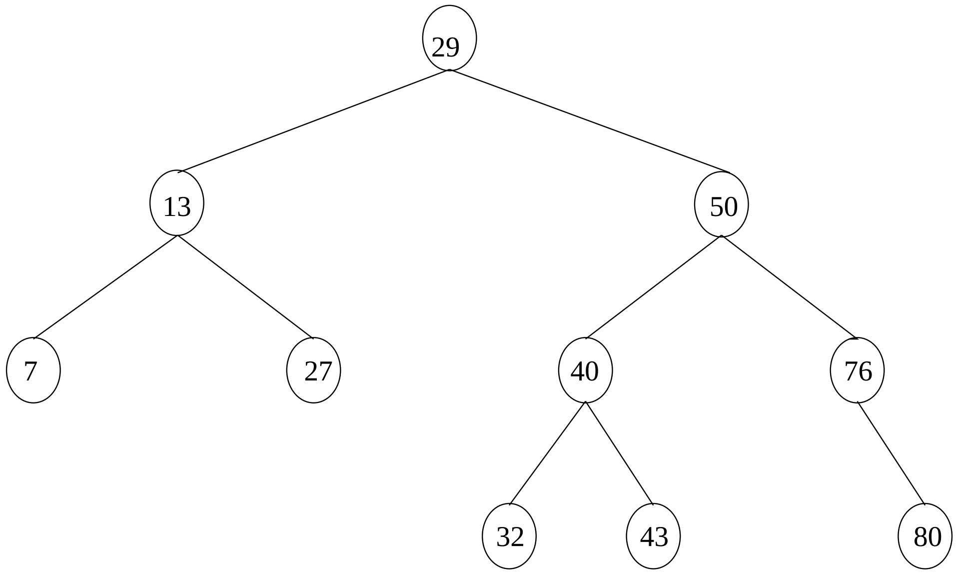



Abbildung 4.1: Die Menge $\{7, 13, 27, 29, 32, 40, 43, 50, 76, 80\}$ in einem binären Baum dargestellt.

- *lc:* Zeiger auf das linke Kind
- *rc:* Zeiger auf das rechte Kind
- *info:* irgendeine Information

Wie suchen wir nun nach einem bestimmten Schlüssel $k$ in einer solchen Datenstruktur? Wir versuchen einen ersten Algorithmus:

**function** SEARCH($k, T$)
  **if** $T =$ NULL **then return** NULL
  **if** $k < T \rightarrow key$ **then** SEARCH($k, T \rightarrow lc$)
  **else if** $k > T \rightarrow key$ **then** SEARCH($k, T \rightarrow rc$)
  **else return** $T$

Hierbei soll der Bezeichner NULL den Nullpointer repräsentieren. Bei dieser Version überprüfen wir allerdings bei jedem rekursiven Neuaufruf von SEARCH, ob als Argument der Nullpointer übergeben wurde. Es liegt daher in unserem Interesse, diesen Aufwand zu umgehen. Wir benutzen dafür das Konzept der *sentinels* (engl. Wächter). Dabei handelt es sich um einen "Baumknoten", auf den ein neu eingeführter Pointer TNULL zeigt (vgl. auch Abbildung 4.2).
Dabei soll jeder Nullpointer, der in unserem Kontext auftritt, fortan ein Pointer TNULL sein (siehe auch Abbildung 4.3), und damit *dereferenzierbar*. Durch Abspeichern des Suchschlüssels in diesem "Pseudoknoten" läßt sich obiger Algorithmus nun auch wie folgt formulieren (und zwar ohne den nervigen NULL-Test):

**function** SEARCH($k, T$)
  TNULL $\rightarrow key = k$
  $p =$ RECSEARCH($k, T$)
  **if** $p =$ TNULL **then return** FAILURE
  **else return** $p$

wobei

**function** RECSEARCH($k, T$)

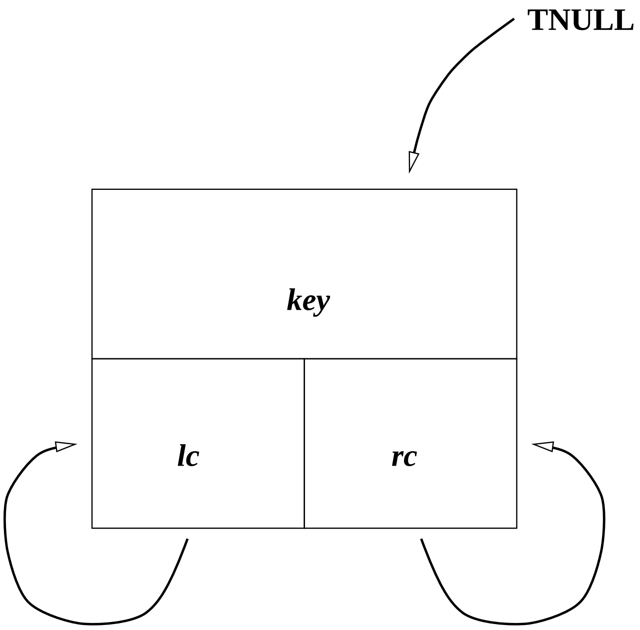



Abbildung 4.2: Der Pointer TNULL

**if** $k < T \to key$ **then** Search $(k, T \to lc)$
**else if** $x.key > T \to key$ **then** Search$(k, T \to rc)$
**else return** $T$

So eine Suche durchwandert also einen Pfad im Baum von der Wurzel bis zu dem gesuchten Knoten (bzw. bei erfolgloser Suche bis zum Vorgänger oder Nachfolger des gesuchten Schlüssels) und braucht dazu Zeit proportional zur Länge dieses Pfades. Diese Länge ist von oben beschränkt durch die Höhe des Baumes. Damit beträgt die Zeit für die Suche in $T$ also $O(\textit{Höhe des Baumes } T)$.
Um schnelle Suche erreichen zu können, ist es also von Vorteil, einen Suchbaum von geringer Höhe zu verwenden. Für jede abzuspeichernde Menge $S$ gibt es zwar einen binären Suchbaum mit einer Höhe von höchstens $\lceil \log_2(|S|+1) \rceil$, aber es ist nicht klar, wie so eine geringe Höhe beibehalten werden kann, wenn es Updateoperationen gibt.
Lassen wir dieses Höhenproblem vorerst außer acht und betrachten wir nur einmal, wie sich Updateoperationen überhaupt realisieren lassen. Das Einfügen ist sehr einfach:

**function** Insert$(x, T)$
**if** $T =$ TNULL **then** $T =$ newnode$(x)$
**else if** $x.key < T \to key$ **then** Insert$(x, T \to lc)$
**else if** $x.key > T \to key$ **then** Insert$(x, T \to rc)$
**else** (* Schlüssel schon vorhanden *)

Das Entfernen ist schwieriger: Es ist relativ einfach, wenn der zu entfernende Knoten $z$ nur ein Kind hat: der ursprünglich auf $z$ zeigende Pointer muß nur auf dieses Kind umgesetzt werden. Hat $z$ zwei Kinder, geht man folgendermaßen vor: Man sucht sich den Knoten $u$ im linken Teilbaum von $z$ mit maximalen Schlüssel (der bei $u$ gespeicherte Schlüssel muß der unmittelbare Vorgänger von dem bei $z$ sein); das bei $u$ gespeicherte Stück wird nach $z$ kopiert (womit das alte Stück bei $z$ praktisch entfernt ist) und dann wird $u$ aus dem linken Teilbaum von $z$ gelöscht (das geht nun einfach, da $u$ als "rechtester" Knoten in diesem Teilbaum höchstens ein Kind haben kann).

**function** Delete$(k, T)$
**if** $k < T \to key$ **then** Delete$(k, T \to lc)$
**else if** $k > t \to key$ **then** Delete$(k, T \to rc)$
**else if** $T =$ TNULL **then** return

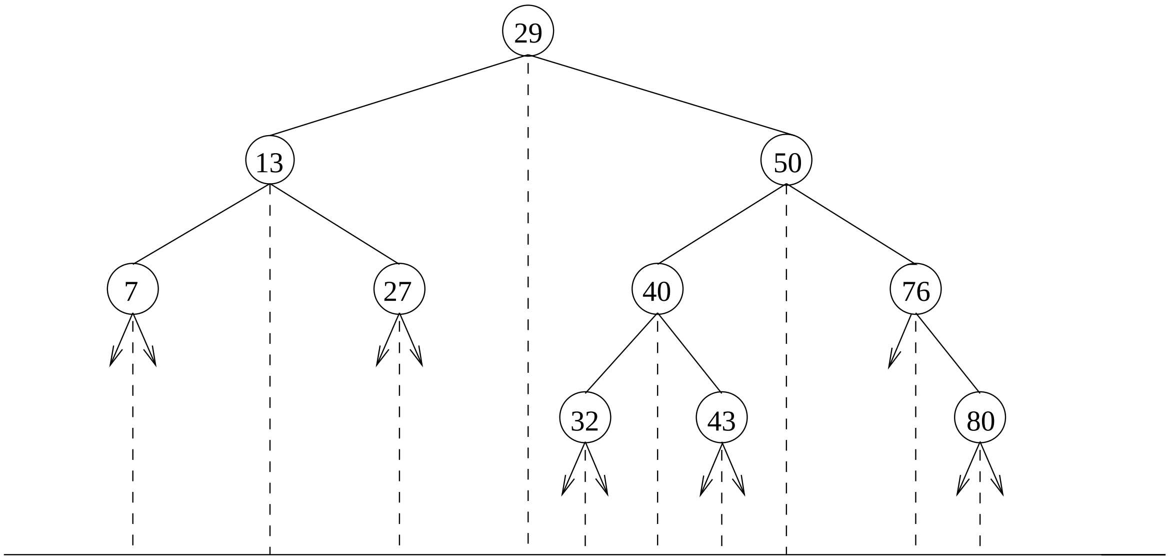



Abbildung 4.3:

**else if** $T \to lc =$ TNULL **then** $T = T \to rc$
**else if** $T \to rc =$ TNULL **then** $T = T \to lc$
**else** $u =$ TREE_MAX$(T \to lc)$
  Kopiere Information von $u$ nach $T$
  DELETE$(u \to key, T \to lc)$

Der Zeitbedarf sowohl für INSERT als auch für DELETE ist wieder $O(\textit{Höhe von } T)$, da in beiden Operationen im wesentlichen ein Pfad von der Wurzel des Baumes zu einem Knoten durchwandert wird. Im Falle von DELETE wird ein Teil dieses Weges zweimal durchwandert, siehe auch Abbildung 4.4.

Vorlesung vom 27.11.1995
Frank Jost

Da Einfüge- und Streich- und Suchoperationen Zeit proportional zur Höhe des Baumes brauchen, ist es von Interesse, diese Höhe klein zu halten. Dies ist mit den angegebenen Updateoperationen nicht unbedingt zu erreichen. Betrachten wir zum Beispiel die Eingabefolge $x_1, ..., x_n$ mit $x_i < x_{i+1}$. Fügt man diese Eingabefolge in einen anfangs leeren Baum ein, so erhält man einen Baum der Höhe $n-1$. Der Baum ist also zu einer verketteten Liste degeneriert.
Um zu großen Höhen bei Suchbäumen zu verhindern gibt es zahllose Strategien. Wir werden drei solche Strategien besprechen: explizite Höhenbalanzierung (AVL-Bäume), Randomisierung (Randomisierte Suchbäume) und amortisierte Umstrukturierung (Splay-Bäume).

## 4.3 AVL-Bäume

**Definition 3** *Ein binärer Suchbaum heißt* AVL-Baum *wenn für jeden Knoten im Baum gilt:*[1] *Die Höhen des linken und des rechten Teilbaums unterscheiden sich um höchstens* 1.

**Behauptung 7** *Ein AVL-Baum mit n Knoten hat Höhe* $\Theta(\log n)$.

**Beweis** Für jeden binärer Baum mit $n$ Knoten und Höhe $h$ gilt offensichtlich $n \leq 2^{h+1} - 1$ und daher $h \geq \log_2(n+1) - 1$. Insbesonders gilt das für AVL-Bäume. Also ist deren Höhe $\Omega(\log n)$.
Daß die Höhe von AVL-Bäumen auch nicht zu groß wird, wird durch die folgende Aussage bewiesen: □

[1] Die Bezeichnung ist nach den Namen ihrer beiden Erfinder: *Adel'son-Velskii* und *Landis*.

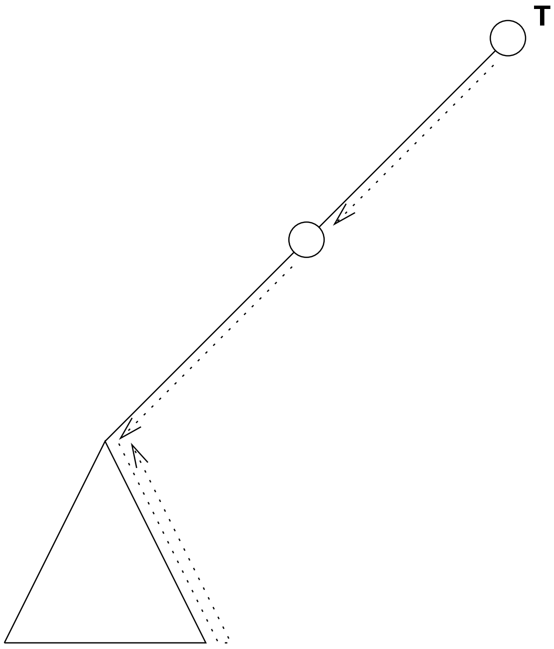



Abbildung 4.4: Weg des DELETE-Algorithmus im Baum

**Behauptung 8** *Ein AVL-Baum der Höhe $h$ hat mindestens $(\frac{3}{2})^h$ Knoten. Bei $n$ Knoten gilt also:*

$$n \geq \left(\frac{3}{2}\right)^h \quad \textit{und daher} \quad h \leq \frac{\log n}{\log 3/2}$$

**Beweis** Wir beweisen die Behauptung durch vollständige Induktion über $h$:

1. $h = 0$: Ein Baum der Höhe 0 hat genau einen Knoten; $1 = (3/2)^0$.
2. $h = 1$: Ein AVL-Baum der Höhe 1 hat mindestens 2 Knoten; $2 > (3/2)^1$.
3. $h > 1$: Ein AVL-Baum der Höhe $h$ besteht aus der Wurzel und einem Teilbaum der Höhe $h-1$ sowie einem Teilbaum, der mindestens die Höhe $h-2$ und höchstens die Höhe $h-1$ hat. Die Anzahl der Knoten des Baumes ist dann nach Induktionsvoraussetzung mindestens

$$1 + (3/2)^{h-1} + (3/2)^{h-2} \geq (3/2)^{h-2} \underbrace{[3/2+1]}_{=\frac{5}{2} \geq (3/2)^2} \geq (3/2)^h .$$

□

### 4.3.1 Rotation

Wir haben jetzt einen binären Suchbaum, dessen Höhe $\Theta(\log n)$ beträgt. Fügen wir nun Elemente hinzu oder streichen welche, wird der Baum verändert und die AVL-Eigenschaft kann verletzt werden. Um diese Eigenschaft wieder herzustellen, bedienen wir uns der *Rotation*. Dies ist eine lokale strukturelle Veränderung des Baumes. Wir unterscheiden zwei Arten von Rotationen: *Links*-Rotation und *Rechts*-Rotation (siehe Abbildung 4.5).
Die Operation ROTATE-LEFT($x$) transformiert die Anordnung der Knoten $x$ und $y$ auf der linken Seite in die Anordnung auf der rechten Seite. Dabei wird eine konstante Anzahl von Zeigern verändert. Der Knoten $y$ wird dadurch zur Wurzel des Teilbaumes, $x$ der linke Sohn von $y$ und der linke Sohn von $y$ wird der rechte Sohn von $x$. ROTATE-RIGHT($y$) macht diese Transformation rückgängig. Die Rotation kann an jedem beliebigen Knoten des Baumes durchgeführt werden.

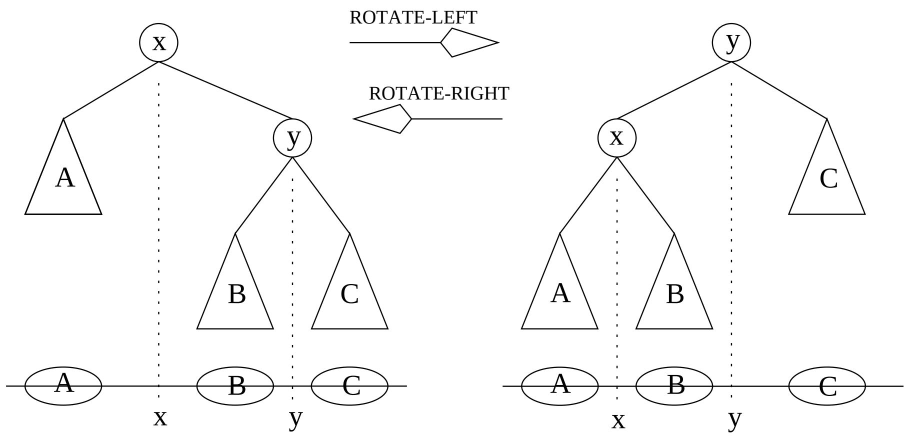



Abbildung 4.5: Rotation

Eine Rotationsoperation läßt die Schlüssel im Baum weiterhin in symmetrischer Ordnung abgespeichert; d.h. Schlüssel im Teilbaum $A$ stehen nach der Operation weiterhin links von Schlüssel $x$, der seinerseits links der Schlüssel im Teilbaum $B$ steht, usw.
Der Pseudocode für die Rotationsfunktionen lautet dann:

(* Knoten $t$ soll rotiert werden *)
Rotate-Left($t$)
$[t, t \to rc, t \to rc \to lc] := [t \to rc, t \to rc \to lc, t]$
(* parallele Zuweisung *)
Rotate-Righft($t$)
$[t, t \to lc, t \to lc \to rc] := [t \to lc, t \to lc \to rc, t]$

Man beachte, daß eine Linksrotation auf $x$ nur angewandt werden kann, wenn $x$ ein rechtes Kind hat. Mutatis mutandis gilt das gleiche für die Rechtsrotation.
Wie können wir nun die Rotation für unseren AVL-Baum ausnutzen? Dazu nennen wir einen Baum $t$ einen *beinahe-AVL-Baum*, wenn das AVL-Kriterium bei allen Knoten des Baumes erfüllt ist, außer bei der Wurzel, bei der allerdings der Höhenunterschied zwischen linkem und rechtem Teilbaum genau 2 ist. Im folgenden werden wir zeigen, wie sich mit Hilfe von höchstens 2 Rotationen sich so ein beinahe-AVL-Baum in einen echten AVL-Baum überführen läßt.
Wir betrachten hier nur die Fälle, in denen der linke Teilbaum der größere ist; der umgekehrte Fall läuft analog. In den Abbildungen 4.6 und 4.7 seien $x$ und $y$ Knoten, $A, B$ und $C$ AVL-Bäume. Der Teilbaum $C$ habe Höhe $h-1$. Der Baum mit Wurzel $y$ sei ein beinahe-AVL-Baum. Das heißt, der Teilbaum mit Wurzel $x$ muß Höhe $h+1$ haben, was wiederum bedeutet, daß $A$ oder $B$ Höhe $h$ haben muß. Wir unterscheiden nun die zwei Fälle, ob $A$ die Höhe $h$ hat oder nicht:

**Erster Fall:** $A$ hat Höhe $h$
Baum $B$ hat dann eine Höhe von $h$ oder $h-1$. Durch eine Rechts-Rotation im Knoten $y$ erreichen wir den Baum, der im rechten Teil von Abbildung 4.6 zu sehen ist. Dieser Baum erfüllt nun die AVL-Bedingung in jedem Knoten. Der neue Baum hat jetzt die Wurzel $x$ und seine beiden Teilbäume unterscheiden sich in der Höhe um höchsten 1. Man beachte, daß der ursprüngliche Baum Höhe $h+2$ hatte, während der Neue Höhe $h+1$ oder $h+2$ haben kann. Das heißt, durch diese Transformation bleibt die Baumhöhe gleich oder verringert sich um 1.

**Zweiter Fall:** $A$ hat Höhe $h-1$
Dies bedeutet, daß $B$ Höhe $h$ haben muß. Sei nun $z$ die Wurzel von $B$ und $B'$ und $B''$ der linke und rechte Teilbaum. Diese beiden Teilbäume haben Höhen $h-1$ oder $h-2$, wobei einer der beiden Höhe $h-1$ haben muß. Durch eine Links-Rotation im Knoten $x$ erhält man dann den rechten Baum in der Abbildung 4.7, aus dem sich durch Anwendung einer Rechts-Rotation im Knoten $y$ der Baum von Abbildung 4.8 ergibt,

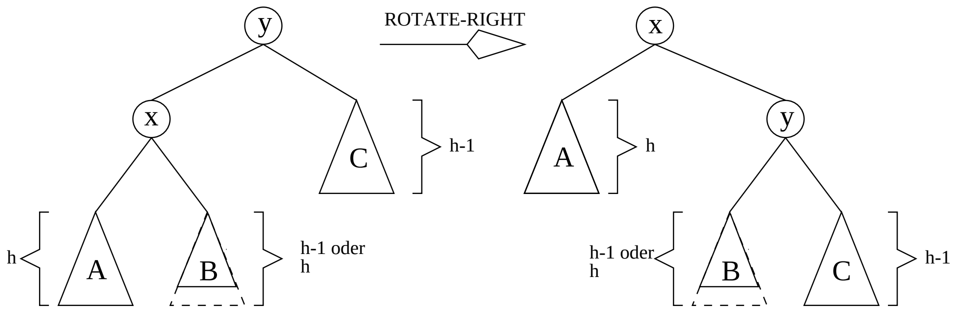



Abbildung 4.6: Rotation in AVL-Baum - erster Fall

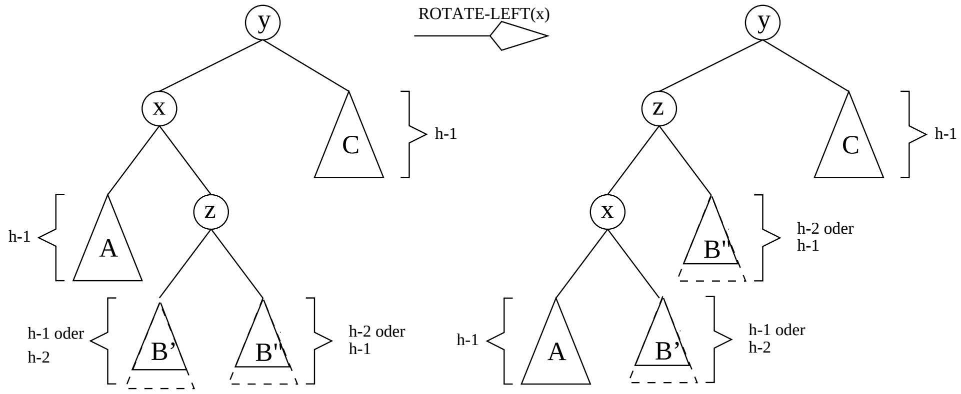



Abbildung 4.7: Rotation in AVL-Baum - zweiter Fall

welcher ein AVL-Baum ist. Man beachte, daß die Höhe des Ergebnisbaumes auf jeden Fall $h+1$ ist, d.h. diese *"Doppelrotation"* verringert die Höhe des ursprünglichen Baumes um 1.

Diese Transformation von einem beinahe-AVL-Baum zu einem echten AVL-Baum mittels höchstens 2 Rotationen wird durch die folgende BALANCE-Funktion implementiert. Es wir angenommen, daß jeder Knoten die Höhe seines Teilbaumes in einer Zelle *ht* gespeichert hat und daß die ROTATE-Funktionen die Höhe richtig umsetzen. Außerdem sei die Höhe eines Blattes Null, und es gelte $tnull \rightarrow ht = -1$.

BALANCE(tree $t$)
  **if** $t \rightarrow lc \rightarrow ht > t \rightarrow rc \rightarrow ht + 1$ **then**
    **if** $t \rightarrow lc \rightarrow lc \rightarrow ht < t \rightarrow lc \rightarrow rc \rightarrow ht$ **then** ROTATE-LEFT($t \rightarrow lc$)
    ROTATE-RIGHT($t$)
  **else if** $t \rightarrow rc \rightarrow ht > t \rightarrow lc \rightarrow ht + 1$ **then**
    **if** $t \rightarrow rc \rightarrow rc \rightarrow ht < t \rightarrow rc \rightarrow lc \rightarrow ht$ **then** ROTATE-RIGHT($t \rightarrow rc$)
    ROTATE-LEFT($t$)

Man beachte, daß eine Anwendung von BALANCE($t$) nur konstante Zeit braucht. Außerdem wird die Höhe von $t$ gleichgelassen oder um 1 veringert.
Abschließend müssen wir noch unsere Einfüge- und Streich-Funktionen für binäre Bäume derart verändern, daß die AVL-Eigenschaft erhalten bleibt. Dies geschieht einfach dadurch, daß an die schon spezifizierten Funktionen INSERT und DELETE einfach ein BALANCE-Operation angehängt wird. Außerdem müssen noch die Höhen richtig verwaltet werden. Daraus ergeben sich die folgenden Prgramme:

AVL-INSERT(Stueck $x$, tree $t$)
  **if** $t =$ tnull **then**
    $t =$ newnode($x$)

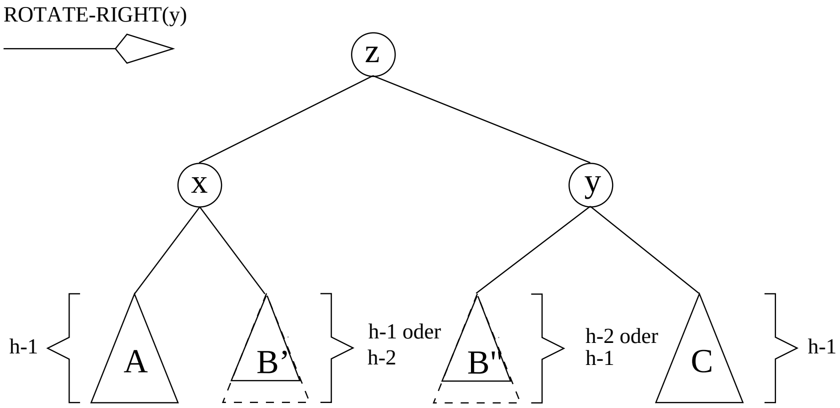



Abbildung 4.8: Der entstehende AVL-Baum - zweiter Fall

$t \to ht = 0$, **return**
**else if** $x$.key $< t \to$key **then** AVL-INSERT$(x, t \to lc)$
**else if** $x$.key $> t \to$key **then** AVL-INSERT$(x, t \to rc)$
**else return** (* Schluessel kommt schon vor *)
$t \to ht = 1 + \{\max(t \to lc \to ht, t \to rc \to ht)\}$
BALANCE$(t)$

AVL-DELETE(Schlüssel $k$, tree $t$)
**if** $k < t \to$key **then** AVL-DELETE$(k, t \to lc)$
**else if** $k >$key **then** AVL-DELETE$(k, t \to rc)$
**else if** $t =$ tnull **then return** (*$k$ nicht im Baum*)
**else if** $t \to lc =$ tnull **then** $t = t \to rc$
**else if** $t \to rc =$ tnull **then** $t = t \to lc$
**else** $u =$ TREE-MAX$(t \to lc)$
kopiere info von $u$ nach $t$
AVL-DELETE$(u \to key, t \to lc)$
$t \to ht = 1 + \max\{t \to lc \to ht, t \to rc \to ht\}$
BALANCE$(t)$

Warum erhalten diese Funktionen die AVL-Eigenschaft? Betrachten wir zuerst das Einfügen. Es gilt, daß AVL-INSERT$(x, t)$ die Höhe vom Baum $t$ gleichläßt oder um eins wachsen läßt. Das ist klar, wenn $t$ leer ist. Ist $t$ nicht leer, wird $x$ in einen der beiden Teilbäume eingefügt; induktiv bleibt die Höhe dieses Teilbaums entweder gleich, in welchem Fall die AVL-Eigenschaft und auch die Höhe von $t$ erhalten bleibt, oder die Höhe des Teilbaums wächst um 1. Wächst damit auch die Höhe von $t$ um 1, dann kann $t$ zu einem beinahe-AVL-Baum werden, und nur in diesem Fall führt das folgende BALANCE$(t)$ Rotationen durch, was die Höhe von $t$ weiterhin um 1 erhöht lassen kann oder wieder auf den ursprünglichen Wert zurückführt.
In ähnlicher Weise kann man sich überlegen, daß AVL-DELETE$(k, t)$ die Höhe von $t$ gleichläßt oder um 1 erniedrigt. Ein wesentlicher Unterschied zwischen der Einfüge- und der Streichoperation besteht jedoch darin, daß das Einfügen eines Stückes in einen AVL-Baum zu höchstens einer Umstrukturierung führen kann, während beim Streichen logarithmisch viele Umstrukturierungen auftreten können.
AVL-DELETE und AVL-INSERT brauchen wieder Zeit proportional zur Länge des durchlaufenen Pfades. Diese ist durch die Höhe des Baumes beschränkt, die aber, wie wir gezeigt haben, bei AVL-Bäumen logarithmisch ist. Damit können wir schließlich dieses Kapitel mit folgendem Satz beenden:

**Satz 8** *Mit AVL-Bäumen können* SEARCH*,* INSERT *und* DELETE *in* $O(\log n)$ *Zeit im schlechtesten Fall durchgeführt werden.*

## 4.4 Zufällig gebaute Suchbäume

Im letzten Kapitel haben wir Aufwand betrieben um sicherzustellen, daß Suchbäume nur logarithmische Höhe haben. Es stellt sich die Frage, ob so ein Aufwand wirklich nötig ist. Vielleicht haben Suchbäume, wenn sie irgendwie zufällig aufgebaut werden, von Haus aus mit hoher Wahrscheinlichkeit geringe Höhe. In diesem Kapitel werden wir zeigen, daß dies unter bestimmten Bedingungen tatsächlich der Fall ist. Speziell betrachten wir nun den Fall, daß in einen anfangs leeren binären Suchbaum eine Schlüsselmenge in zufälliger Reihenfolge eingefügt wird. Streichungen betrachten wir nicht.
Formal sei $S = \{s_1, ..., s_n\}$ eine Menge von $n$ Stücken mit $key(s_i) < key(s_{i+1})$ und $\pi$ eine Permutation auf $\{1, ..., n\}$. Für Permutation $\pi$ wird dann ein Baum $T(\pi)$ für $S$ wie folgt aufgebaut:

$T$ = empty
**for** $i = 1$ **to** $n$ **do** INSERT$(s_{\pi(i)}, T)$

Dabei ist INSERT die beschriebene Einfügeoperation für (unbalancierte) Binarbäume.
Für jede Permutation $\pi$ erhalten wir also einen Baum $T(\pi)$ mit einer gewissen Höhe $H = H(\pi)$. Wie groß ist $H$ "im Durchschnit"? Formal gesehen können wir $H$ als Zufallsvariable über dem Wahrscheinlichkeitsraum $\Sigma_n$ aller $n!$ Permutationen ansehen, und es macht Sinn über ihren Erwartungswert zu sprechen, wobei wir als Wahrscheinlichkeitsverteilung die Gleichverteilung über $\Sigma_n$ annehmen werden, also $\mathrm{E}[H] = \frac{1}{n!} \sum_{\pi \in \Sigma_n} H(\pi)$.
Allerdings gestaltet sich die Berechnung des Erwartungswertes von $H$ sehr schwierig. Daher ändern wir die Problemstellung etwas ab und fragen, wie lange wir im Mittel brauchen, um einen Stück $s_i$ im Baum $T(\pi)$ zu finden, d.h. wie lang ist im Erwartungswert der Pfad von der Wurzel zum gesuchten Stück $s_i$.
Für $i = 1, \ldots, n$ sei daher $D_i(\pi)$ die Länge des Weges von $s_i$ bis zur Wurzel im Baum $T(\pi)$, also die Anzahl der Ahnen vom Knoten für $s_i$ im Baum $T(\pi)$. Zur Berechnung des Erwartungswertes der Zufallsvariablen $D_i$ verwenden wir wieder Indikatorvariablen. Für $1 \le i, j \le n$ definieren wir

$$A_{ij}(\pi) = \begin{cases} 1 & \text{wenn Knoten für } s_j \text{ Ahne vom Knoten für } s_i \text{ in } T(\pi) \text{ ist} \\ 0 & \text{sonst.} \end{cases}$$

Damit gilt

$$D_i(\pi) = \sum_{1 \le j \le n} A_{ij}(\pi)$$

und

$$\mathrm{E}[D_i] = \sum_j \mathrm{E}[A_{ij}] = \sum_j \Pr(A_{ij} = 1) = \sum_j \Pr(s_j \text{ Ahne von } s_i)\,.$$

Zur Analyse der Wahrscheinlichkeit, ob $s_j$ ein Ahne von $s_i$ ist, stellen wir uns vor, daß wir zu jedem Knoten in $T(\pi)$ noch eine Zahl dazuschreiben, und zwar zum Knoten für $s_\ell$ den Wert $\pi(\ell)$. Es sollte klar sein, daß auf Grund, wie $T(\pi)$ gebaut wurde, die beim Vater stehende Zahl kleiner sein muß als die eigene (siehe Abbildung 4.9).
Um über all dies leichter reden zu können, führen wir den Begriff des *Treaps* ein:

**Definition 4** *Sei $M$ eine endliche Menge von Stücken, wobei jedes $m \in M$ einen Schlüssel $key(m)$ und eine Priorität $prio(m)$ besitzt.*
*Ein* Treap *(Tree + Heap) für $M$ ist ein binärer Baum $T$, in dem jeder Knoten ein Stück von $M$ speichert und der folgende Eigenschaften besitzt (siehe Abbildung 4.9).*

- *$T$ ist ein Suchbaum bzgl. der Schlüssel*
- *$T$ ist ein Heap bzgl. der Prioritäten, d.h. für jeden Knoten außer der Wurzel ist die Priorität des Vaters kleiner als die eigene.*

Es sollte klar sein, daß der Treap für $M$ eindeutig ist, falls alle auftretenden Schlüssel untereinander verschieden sind und das gleiche auch für die Prioritäten gilt: Das Stück $m$ mit der kleinsten Priorität wird zur Wurzel; den linken Teilbaum bildet induktiv der Treap für die Stücke in $M$ mit Schlüssel kleiner als $key(m)$ und den rechten Teilbaum bildet induktiv der Treap für die Stücke in $M$ mit Schlüssel größer als $key(m)$.
In unserer ursprünglichen Situation bekommt bei gegebener Permutation $\pi$ jedes Stück $s_i \in S$ die Priorität $\pi(i)$, und $T(\pi)$ ist dann der sich ergebende Treap für $S$.
Das folgende Lemma ist für uns von größter Wichtigkeit, da es Ahnen in Treaps charakterisiert.

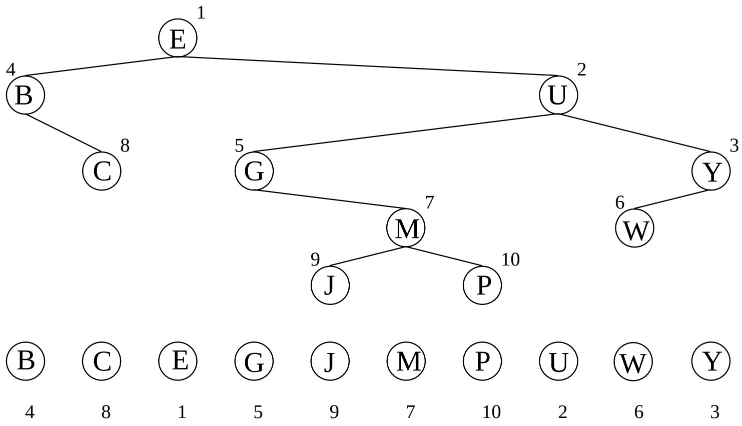



Abbildung 4.9: Ein TREAP

**Lemma 4 ("Ahnenlemma")** *Sei $S = \{s_1, ..., s_n\}$ eine Menge von Stücken. Jedes Stück $s_i$ besitze einen Schlüssel $key(s_i)$ und eine Priorität $prio(s_i)$. Es gelte $key(s_i) < key(s_{i+1})$. Dann ist im Treap $T$ für $S$ das Stück $s_j$ genau dann ein Ahne von $s_i$, wenn unter allen $s_\ell$ mit $\ell$ zwischen $i$ und $j$, $s_j$ die kleinste Priorität besitzt, d.h. es gilt*

$$s_j \text{ ist Ahne von } s_i \text{ in } T \;\Leftrightarrow \begin{cases} prio(s_j) = \min\left\{prio(s_\ell) \mid i \le \ell \le j\right\} & \text{für } i \le j \\ prio(s_j) = \min\left\{prio(s_\ell) \mid j \le \ell \le i\right\} & \text{für } i \ge j \end{cases}$$

Zum Beispiel möchte man in Abbildung 4.9 wissen, ob das Stück mit Schlüssel $E$ ein Ahne vom Stück mit Schlüssel $W$ ist. $E$ hat im Intervall $[E, W]$ die kleinste Priorität, also ist $E$ Ahne von $W$. Ist auch $B$ eine Ahne von $W$? $B$ hat im Intervall $[B, W]$ nicht die kleinste Priorität, denn die hat $E$. Also ist $B$ kein Ahne von $W$.
**Beweis** Sei $s_k$ das Stück in $S$ mit der kleinster Priorität; es bildet die Wurzel von $T$. Dann gilt für alle $i$, daß $prio(s_k) \le prio(s_i)$, und tatsächlich ist $s_k$ Ahne für alle $s_i$. Für $i < k < j$ ist $prio(s_k)$ die kleinste Priorität für $i \le \ell \le j$ und nicht $prio(s_i)$ oder $prio(s_j)$, und in der Tat ist in diesem Fall $s_i$ nicht Ahne von $s_j$ oder umgekert. Falls $i, j < k$ oder $i, j > k$ verwende das Argument rekursiv auf die Treaps, die den linken bzw. rechten Unterbaum der Wurzel bilden. □

Vorlesung vom 30.11.1995
Bianca Schröder, Michael Rovatsos, Oliver Plaehn

Um mit Hilfe des Ahnenlemmas zu Wahrscheinlichkeiten zu kommen, betrachten wir erstmal ein paar Eigenschaften der Permutationen $\pi \in \Sigma_n$:

**Behauptung 9** *Für alle $1 \le i \le n$ gilt:*

$$\left|\left\{\pi \in \Sigma_n \mid \pi(i) = \min\left\{\pi(l) \mid 1 \le l \le n\right\}\right\}\right| = (n-1)!$$

Weil $\pi$ in unserem Fall eine Permutation der Menge $\{1, 2, \ldots, n\}$ ist, ist $\pi(i) = \min\{\pi(l) \mid 1 \le l \le n\}$ nur eine umständliche Schreibweise für $\pi(i) = 1$. Dann sagt obige Behauptung aus, daß die Anzahl der Permutationen aus $\Sigma_n$, bei denen an der $i$-ten Stelle 1 (bzw. das kleinste Element) steht, gleich $(n-1)!$ ist. Intuitiv ist dies die Anzahl der Möglichkeiten, die verbleibenden $n-1$ Elemente anzuordnen, wenn eins (nämlich das kleinste) festgelegt ist.
Daraus folgt direkt, daß jedes Element der Permutation $\pi$ mit gleicher Wahrscheinlichkeit das kleinste ist, also gilt für $i = 1, 2, \ldots, n$

$$\Pr(\pi(i) = 1) = \frac{(n-1)!}{n!} = \frac{1}{n}\,.$$

Oft interessieren wir uns nicht für alle Elemente aus $\{1, 2, \ldots, n\}$, sondern nur für eine Teilmenge hiervon. Dann schreibt sich eine ähnliche Behauptung wie folgt:

**Behauptung 10** *Für $i \in I \subset \{1, \ldots, n\}$ gilt:*

$$\frac{\left|\left\{\pi \in \Sigma_n \mid \pi(i) = \min\left\{\pi(l) \mid l \in I\right\}\right\}\right|}{n!} = \frac{1}{|I|}.$$

**Beweis** Für $i \in I$ sei $\Pi_{i,I} = \{\pi \in \Sigma_n | \pi(i) = \min_{\ell \in I} \pi(\ell)\}$. Es genügt zu zeigen, daß für alle $i, j \in I$ gilt $|\Pi_{i,I}| = |\Pi_{j,I}|$. Dafür genügt es zu zeigen, daß es eine Bijektion zwischen diesen beiden Mengen gibt. Die Funktion $f_{ij} : \Sigma_n \to \Sigma_n$ eingeschränkt auf $\Pi_{i,I}$ liefert so eine Bijektion, wobei $f_{ij}(\pi) = \pi'$ mit $\pi'(i) = \pi(j)$, $\pi'(j) = \pi(i)$ und $\pi'(\ell) = \pi(\ell)$ sonst. ◻

Aus der letzten Behauptung folgt direkt das

**Korollar 2** *Sei wieder $S = \{s_1, s_2, \ldots, s_n\}$ mit $key(s_i) < key(s_{i+1})$ für $i = 1, 2, \ldots, n-1$, $\pi \in \Sigma_n$ zufällig, $prio(s_i) = \pi(i)$ für $i = 1, 2, \ldots, n$. Dann gilt folgende Behauptung:*

$$\Pr(s_j \textit{ Ahne von } s_i) = \begin{cases} \dfrac{1}{i-j+1} & \textit{für } i \geq j \\ \dfrac{1}{j-i+1} & \textit{für } j \geq i \end{cases}$$

<u>Anschaulich:</u>

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Stück $s_j$ : | $s_1$ | $\ldots$ | $s_{i-2}$ | $s_{i-1}$ | $s_i$ | $s_{i+1}$ | $s_{i+2}$ | $\ldots$ | $s_n$ |
| $\Pr(s_j$ Ahne von $s_i)$ : | $\frac{1}{i}$ | $\ldots$ | $\frac{1}{3}$ | $\frac{1}{2}$ | $1$ | $\frac{1}{2}$ | $\frac{1}{3}$ | $\ldots$ | $\frac{1}{n-i+1}$ |

Mithilfe dieses Korollars läßt sich nun endlich der Erwartungswert für $D_i$, die Tiefe vom Knoten für Stück $s_i$, berechnen. Es gilt

$$\begin{aligned}
\mathrm{E}[D_i] &= \sum_{1 \leq j \leq n} \mathrm{E}[A_{ij}] = \sum_{1 \leq j \leq n} \Pr(A_{ij} = 1) \\
&= \sum_{1 \leq j \leq n} \Pr(s_j \text{ Ahne von } s_i) \\
&= \underbrace{\frac{1}{i} + \ldots + \frac{1}{3} + \frac{1}{2} + 1}_{H_i} + \underbrace{\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1}}_{H_{n+1-i}-1} \\
&= H_i + H_{n+1-i} - 1 \\
&\leq 1 + 2\ln n \qquad (\text{wegen } \ln n < H_n \leq 1 + \ln n)
\end{aligned}$$

Wir fassen unser Ergebnis zusammen:

**Satz 9** *Wenn die Schlüssel $k_1, k_2, \ldots, k_n$ (mit $k_i < k_{i+1}$ für $i = 1, 2, \ldots, n-1$) in zufälliger Reihenfolge in einen anfangs leeren binären Baum eingefügt werden, dann ist die erwartete Tiefe vom Knoten für $k_i$:*

$$H_i + H_{n+1-i} - 1 \leq 1 + 2\ln n\,.$$

## 4.5 Randomisierte Suchbäume

Wir haben also im Prinzip bewiesen, daß man einen ausgewogenen balancierten Baum erwarten kann, wenn die Schlüssel in zufälliger Reihenfolge eingefügt werden. Wie kann man nun sicherstellen, daß tatsächlich eine "zufällige Reihenfolge" benutzt wurde? Man könnte sich einerseits darauf verlassen, daß die Eingabe eben "zufällig" ist. Viel sicherer und schöner wäre es allerdings, wenn man, wie beim universellen Hashen, den Zufall unter die Kontrolle des Programmes bringen könnte, also explizite Randomisierung verwendet. Das ist leicht, wenn die gesamte Schlüsselmenge gegeben ist und ein Suchbaum für sie aufgebaut werden soll: Durch Randomisierung bringt man zuerst die Menge in eine zufällige Ordnung; dann fügt man die Elemente in dieser Reihenfolge in einen Baum ein.

Was macht man aber, wenn anfangs die gesamte Menge noch nicht bekannt ist, oder wenn gar Streichungen aus dem Baum durchgeführt werden sollen. Die Idee ist nun, den Baum immer so aussehen zu lassen, *als ob er durch zufällige Einfügereihenfolge entstanden wäre.* Dies kann man erzielen, indem man jedem Stück einen "zufälligen Zeitpunkt" zuweist, zu dem es (angeblich) in den Baum eingefügt wurde, diesen Zeitpunkt als Priorität mit dem Stück speichert, und diese Stücke mit Schlüsseln und Prioritäten dann in einem Treap verwaltet. Damit kommen wir zur folgenden formalen

**Definition 5** *Ein* randomisierter Suchbaum *für eine Schlüsselmenge $K = \{k_1, k_2, \ldots, k_n\}$ ist ein Treap für die Stückmenge $\{(k_i, x_i) \mid 1 \le i \le n\}$, wobei die $x_i$ unabhängige, gleichverteilte, stetige Zufallsvariablen sind.*

Der Grund für die Adjektive "unabhängig, gleichverteilt, stetig" liegt in der folgenden Behauptung, die analog zu Behauptung 10 ist:

**Behauptung 11** *Seien $X_1, X_2, \ldots, X_n$ unabhängige, gleichverteilte, stetige Zufallsvariablen. Dann gilt für $i \in I \subset \{1, \ldots, n\}$:*

$$\Pr(X_i = \min\{X_\ell | \ell \in I\}) = \frac{1}{|I|}$$

Damit gilt dann sofort Korollar 2 des Ahnenlemmas und es ergibt sich sofort das folgende Analogon zu Satz 9:

**Satz 10** *Sei $K = \{k_1, k_2, \ldots, k_n\}$ eine Schlüsselmenge mit $k_i < k_{i+1}$ für $i = 1, 2, \ldots, n-1$. Dann ist die erwartete Tiefe vom Knoten für $k_i$ in einem randomisierten Suchbaum für $K$*

$$H_i + H_{n+1-i} - 1 \le 1 + 2 \ln n\,.$$

In einem randomisierten Suchbaum hat also jeder Knoten erwartete logarithmische Tiefe.
Zwei Fragen sind nun noch offen. Erstens, wie realisert man eigentlich "unabhängige, gleichverteilte, stetige" Zufallsvariablen? Man nimmt einfach Pseudozufallszahlen aus einem hinreichend großen Bereich (z.B. ganze Zahlen aus $[0..2^{31} - 1]$). Das heißt, um eine Priorität zu erzeugen, wird einfach der Zufallszahlengenerator verwendet. Erfahrungsgemäß verhalten sich solche Zufallszahlen so, daß Behauptung 11 hinreichend genau gilt.
Die zweite Frage ist komplizierter: Um randomisierte Suchbäume auch verwendbar zu machen, muß es möglich sein Treaps zu manipulieren, also Stücke in sie einfügen und aus ihnen entfernen können, wobei die Treapeigenschaft aufrecht erhalten bleiben soll. Dies ist aber nicht allzu schwer. Betrachten wir zuerst das Einfügen:
Die Idee ist, zunächst den Knoten mit einer gegebenen Priorität in den bereits aufgebauten Treap so wie bisher einzufügen, also an die durch die Ordnung auf den Schlüsseln gegebene Blattposition, und anschließend durch Rotationen den Knoten gemäß der Priorität an die richtige Stelle zu bringen, so daß die Heap-Eigenschaft wieder erfüllt ist.
Veranschaulicht wird dies in den Abbildungen 4.10 und 4.11: Abbildung 4.10 zeigt einen Treap, in dem die Knoten mit den Schlüsseln der Stücke beschriftet und mit deren Priorität versehen sind. Abbildung 4.11 veranschaulicht, wie in diesen Treap ein Schlüssel $H$ mit der Priorität 2.5 eingefügt wird: zunächst wird $H$ einfach lexikographisch korrekt eingefügt, ohne die Heap-Eigenschaft der Prioritäten zu berücksichtigen (Bild (a)). Nun werden Rotationen so oft ausgeführt, bis der Elternknoten von $H$ kleinere Priorität als $H$ hat (angedeutet in den Bildern (b), (c)). Ist der Algorithmus beendet, so erfüllt der Baum wieder die Heap-Eigenschaft.
Die Zeit, die für diese Einfügeoperation gebraucht wird, ist proportional zur Länge des Weges von der Wurzel zur erreichten Blattposition. Diese Länge ist um eins größer als die Tiefe des Vorgängers oder Nachfolgers (bzgl. Schlüsselrang) des einzufügenden Stückes, also in einem randomisierten Suchbaum logarithmisch im Erwartungswert.

Hier ist die genau Spezifikation der Einfügeoperation:

**function** TREAP-INSERT$\big((k,p) : \text{Stück}, T\big)$
- **if** $T = tnull$ **then** $T \leftarrow$ NEWNODE$(k,p)$
- **else if** $k < T \to key$ **then** TREAP-INSERT$\big((k,p), T \to lc\big)$
  - **if** $T \to lc \to prio < T \to prio$ **then** ROTATE-RIGHT$(T)$
- **else if** $k > T \to key$ **then** TREAP-INSERT$\big((k,p), T \to rc\big)$
  - **if** $T \to rc \to prio < T \to prio$ **then** ROTATE-LEFT$(T)$
- **else** (* $k$ schon im Baum $T$ enthalten *)

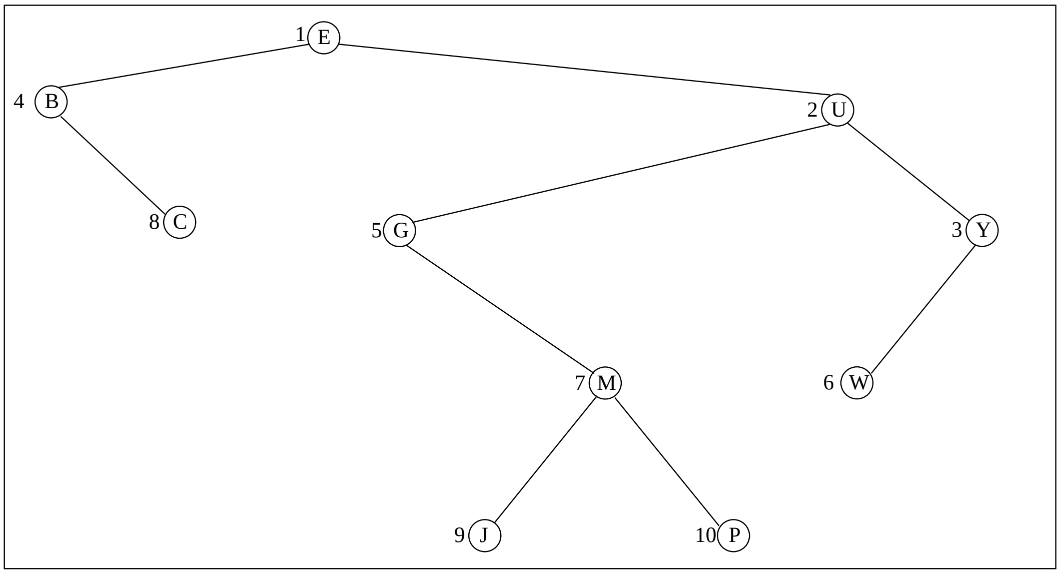



Abbildung 4.10: Treap bestehend aus 10 Stücken mit Buchstaben als Schlüsseln, versehen mit Prioritäten, die die Heap-Eigenschaft erfüllen

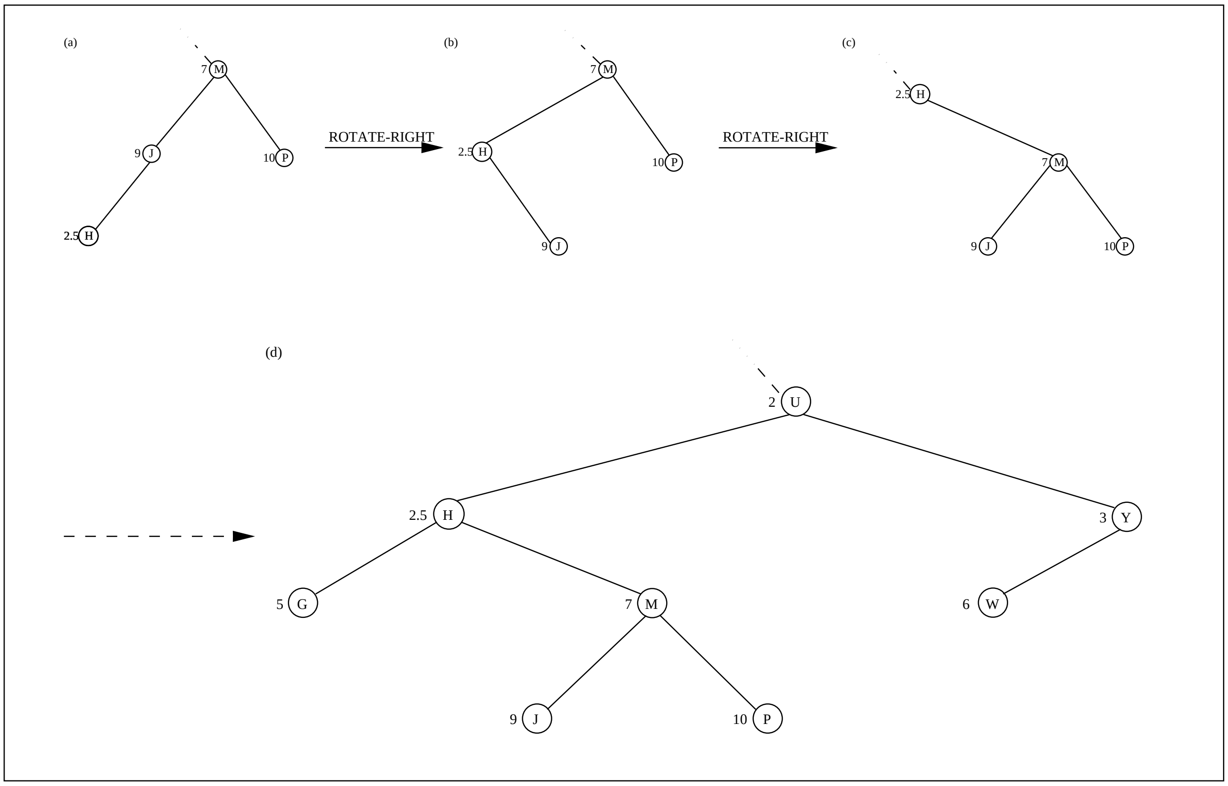



Abbildung 4.11: Veranschaulichung von Treap-Insert — (a) zeigt den Teilbaum, in den (H, 2.5) zunächst eingeordnet wird. (b), (c) zeigen die ersten zwei Rechtsrotationen und (d) das Endergebnis: (H, 2.5) ist auch seiner Priorität nach in den Treap eingefügt.

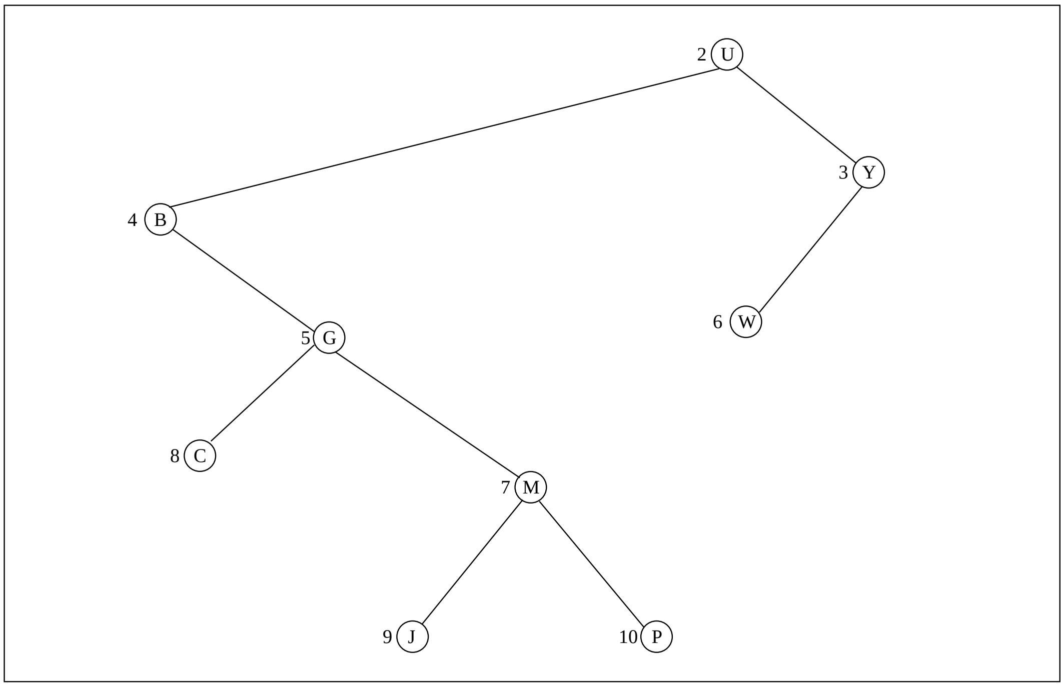



Abbildung 4.12: Das Ergebnis der Treap-Del-Funktion beim Löschen der Wurzel(E,1).

Nun zum Löschen eines Knotens aus einem Treap. Man mache sich zuerst das Vorgehen beim Löschen eines Knotens $K$ mit zwei Vorgängern aus einem binären Baum klar. In diesem Fall wird der Inhalt des größten Knotens des linken Teilbaums mit dem Inhalt von $K$ vertauscht, und anschließend wird der Knoten, in dem jetzt der Inhalt von $K$ steht, aus dem Baum gelöscht. Wird das genauso in einem Treap gemacht, kann dadurch natürlich die Heap-Ordnung des Treaps bezüglich der Prioritäten empfindlich gestört werden.
Das Problem läßt sich durch Einführung einer speziellen Funktion Treap-Del lösen. Diese Funktion sucht zunächst das zu entfernende Element mit Hilfe der Funktion Rec-Treap-Del, rotiert es dann mit Hilfe der Funktion Root-Del entlang der niedrigeren Prioritäten solange nach unten, bis es ein Blatt ist, und entfernt es schließlich.
Im wesentlichen beschreibt Treap-Del nichts anderes als die Umkehrung von Treap-Insert (man ersetze in Abbildung 4.11 die Rechtsrotationen durch Linksrotationen und stelle sich vor, daß Bild (d) die Ausgangssituation beschreibt, in der das zu löschende Element "hinunterrotiert" werden muß, bis es als Blatt gelöscht werden kann).
Da grundsätzlich durch Rotieren die Ordnung bzgl. der Schlüssel nicht beeinträchtigt wird und außerdem entlang der niedrigeren Prioritäten rotiert wird, so daß auch die Ordnung bzgl. der Prioritäten nicht gestört wird, verändert diese Art des Löschens die Treap-Eigenschaften des Baumes nicht.

**function** Treap-Del($k$ : Schlüssel, $T$)
  $tnull \rightarrow key = k$
  Rec-Treap-Del($k$, $T$)

**function** Rec-Treap-Del($k$ : *Schlüssel*, $T$)
  **if** $k < T \rightarrow key$ **then** Rec-Treap-Del($k$, $T \rightarrow lc$)
  **else if** $k > T \rightarrow key$ **then** Rec-Treap-Del($k$, $T \rightarrow rc$)
  **else** Root-Del($T$)

**function** Root-Del($T$)
  **if** Ist-Blatt-oder-Leer($T$) **then** $T = tnull$

**else** **if** $T \to lc \to prio < T \to rc \to prio$ **then**
  ROTATE-RIGHT($T$)
  ROOT-DEL($T \to rc$)
 **else** ROTATE-LEFT($T$)
  ROOT-DEL($T \to lc$)

Die Zeit, die für so eine Löschoperation gebraucht wird, ist wieder proportional zu Länge des Pfades von der Wurzel zur erreichten Blattposition, die wiederum um eins größer ist als die Tiefe des Vorgängers oder Nachfolgers (bzgl. Schlüsselrang) des zu löschenden Stückes, also in einem randomisierten Suchbaum logarithmisch im Erwartungswert.
Wir fassen zusammen:

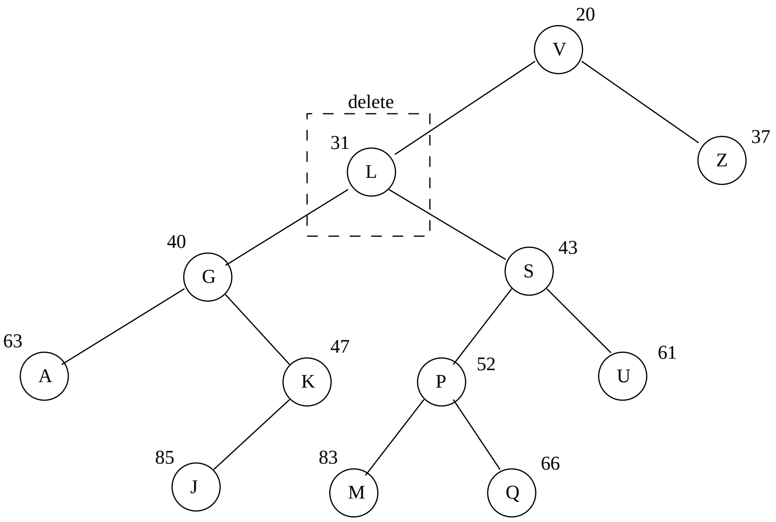
**Satz 11** *In einem randomisierten Suchbaum für $n$ Stücke können die Operationen SEARCH, INSERT und DELETE in erwarteter Zeit $O(\log n)$ durchgeführt werden.*

Vorlesung vom 04.12.95
Ralf Mindermann, Bare Said, Sven Schaumann

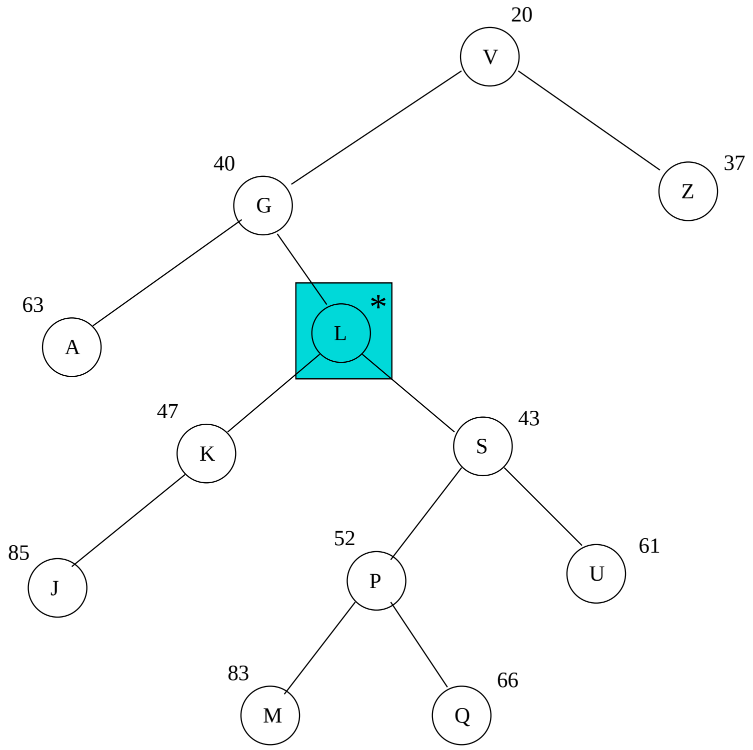
### 4.5.1 Rotationskosten in randomisierten Suchbäumen

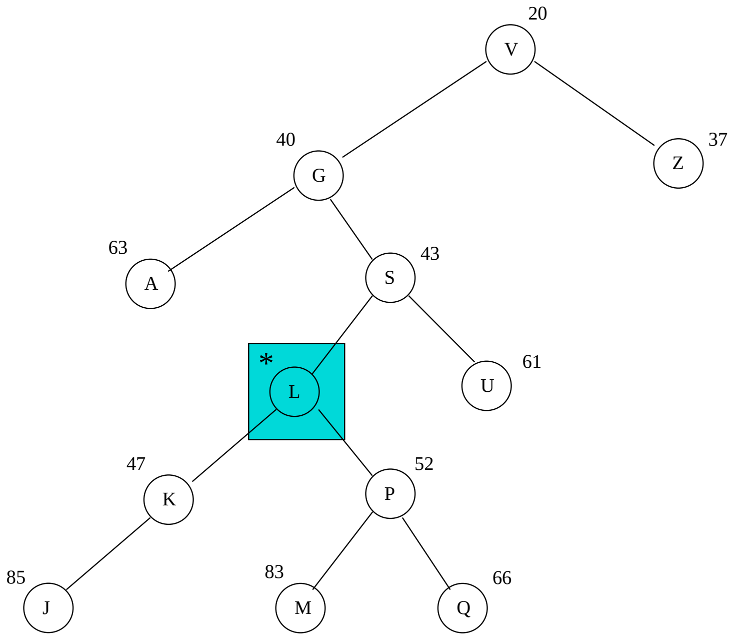
Aus der Tatsache, daß Updates in randomisierten Suchbäumen nur logarithmische erwartete Zeit benötigen, folgt sofort, daß die erwartete Anzahl von Rotationen während eines solchen Updates auch nur höchstens logarithmisch sein kann. Wir wollen nun zuerst zeigen, daß diese erwartete Rotationsanzahl sogar noch wesentlich kleiner ist, nämlich kleiner als 2. Eine Konsequenz davon ist zum Beispiel, daß das Löschen eines Knotens aus einem randomisierten Suchbaum in konstanter erwarteter Zeit erfolgen kann, wenn man nicht nach dem Knoten suchen muß, also wenn man z.B. mit einem Zeiger direkt auf diesen Knoten zugreifen kann[2].
Betrachten wir zuerst ein Beispiel. In den folgenden Abbildungen werden die Rotationen gezeigt, wenn der Knoten mit Schlüssel $L$ aus einem Treap gelöscht wird. Dazu muß er erst so lange nach unten rotiert werden, bis er ein Blatt ist. Dann kann er aus dem Baum gelöscht werden.
Einfügen geht genau umgekehrt: man sucht den Knoten, an dem man das einzufügende Element bzgl. der Schlüsselordnung als Blatt einfügen würde und rotiert es dann solange nach oben, bis auch die Heap-Ordnung wiederhergestellt ist. Die Abbildung in umgekehrter Reihenfolge betrachtet stellt also das Einfügen des Knotens mit Schlüssel $L$ dar.

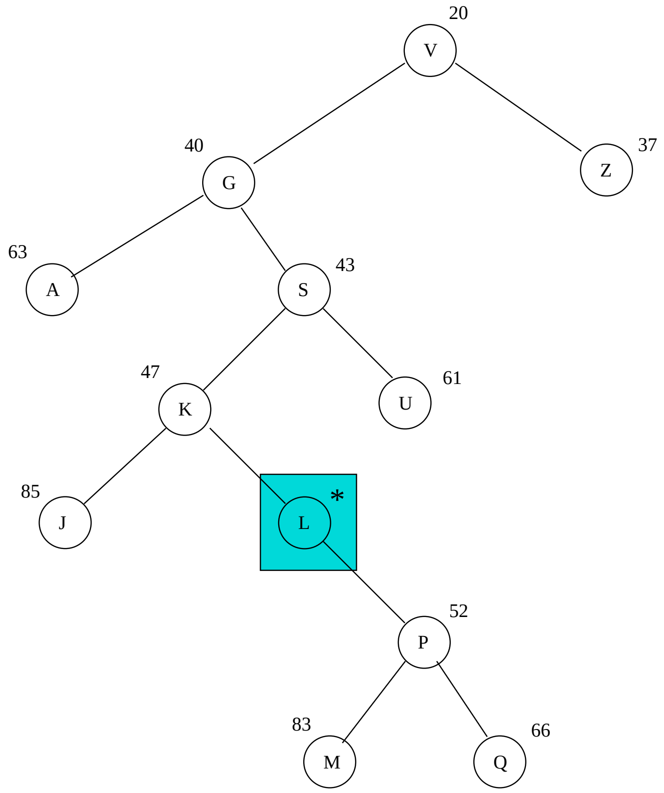
| Key: | A | G | J | K | L | M | P | Q | S | U | V | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Prio: | 63 | 40 | 85 | 47 | 31 | 83 | 52 | 66 | 43 | 61 | 20 | 37 |

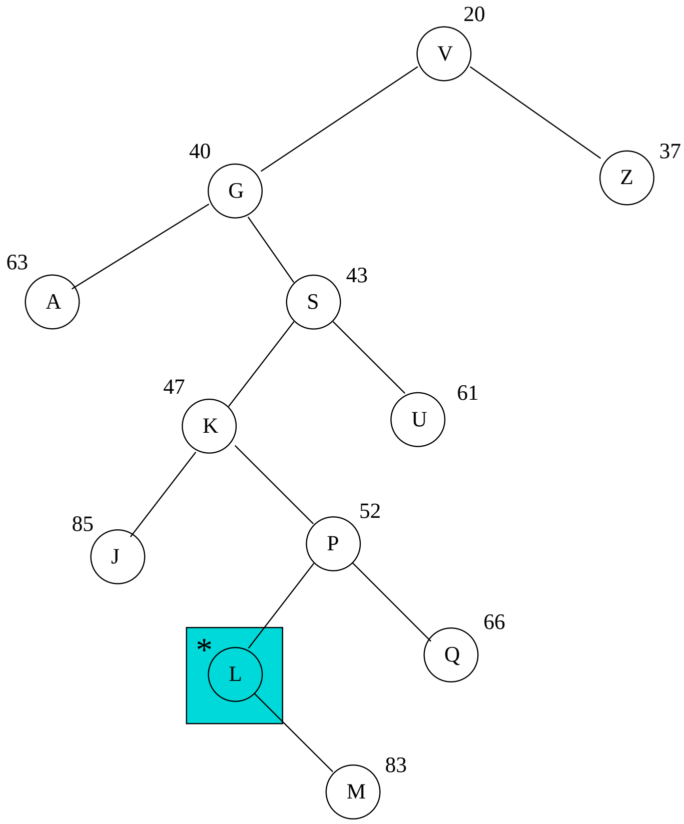


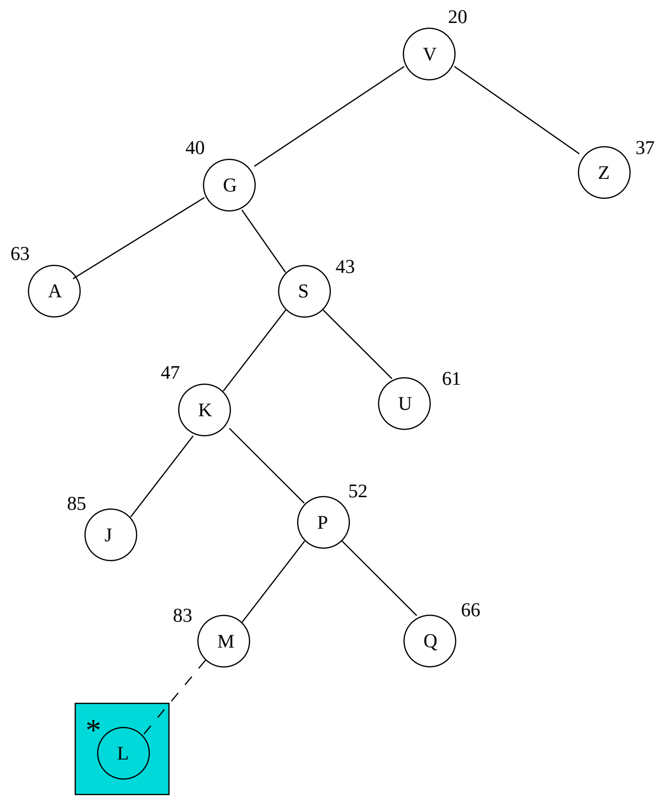

[2] Allerdings müssen dann Vaterzeiger zur Verfügung stehen.


20
V
40
G
37
Z
63
A
L
*
47
K
43
S
85
J
52
P
61
U
83
M
66
Q


20
V
40
G
37
Z
63
A
43
S
*
L
61
U
47
K
52
P
85
J
83
M
66
Q


20
V
40
G
37
Z
63
A
43
S
47
K
61
U
85
J
*
L
52
P
83
M
66
Q


20
V
40
G
37
Z
63
A
43
S
47
K
61
U
85
J
52
P
*
L
66
Q
83
M




Da Einfügen sich genau invers zum Löschen verhält, genügt es, wenn wir die erwartete Anzahl von Rotationen nur für die DELETE-Operation betrachten.

Ein Knoten $x$ soll nun aus einem Treap $T$ entfernt werden. Kann man nun schon an $T$ ablesen, wie viele Rotationen benötigt werden? Dies ist tatsächlich der Fall.
Bezeichnen wir den *linken Grat* eines Baumes $B$ als die Wurzel von $B$ zusammen mit dem linken Grat des linken Unterbaums der Wurzel. (Siehe hiezu auch Abbildung 4.13.) Der *rechte Grat* wird analog definiert.

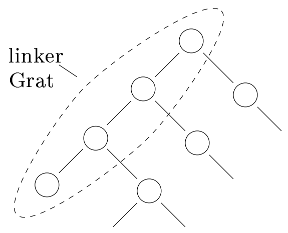



Abbildung 4.13: linker Grat eines Baumes

**Behauptung 12** *Die Anzahl der Rotationen, die notwendig sind, um $x$ aus dem Treap $T$ zu entfernen, ist gleich der Anzahl der Knoten am rechten Grat des linken Unterbaumes von $x$ zuzüglich der Anzahl der Knoten am linken Grat des rechten Unterbaumes von $x$.*

Diese Behauptung sieht man am leichtesten anhand der folgenden Abbildung 4.14: bei jeder Linksrotation wird die Länge des linken Grats des rechten Teilbaums um eins vermindert, die Länge des rechten Grats des linken Teilbaums bleibt erhalten. Bei einer Rechtsrotation geschieht das Umgekehrte. Man muß solange rotieren, bis das zu löschende Element ein Blatt geworden ist.
Um die erwartete Anzahl von Rotationen bei einer DELETE-Operation aus einem randomisierten Suchbaum zu bestimmen, benötigt man also die Erwartungswerte für die Anzahl der Knoten, die sich auf dem rechten,

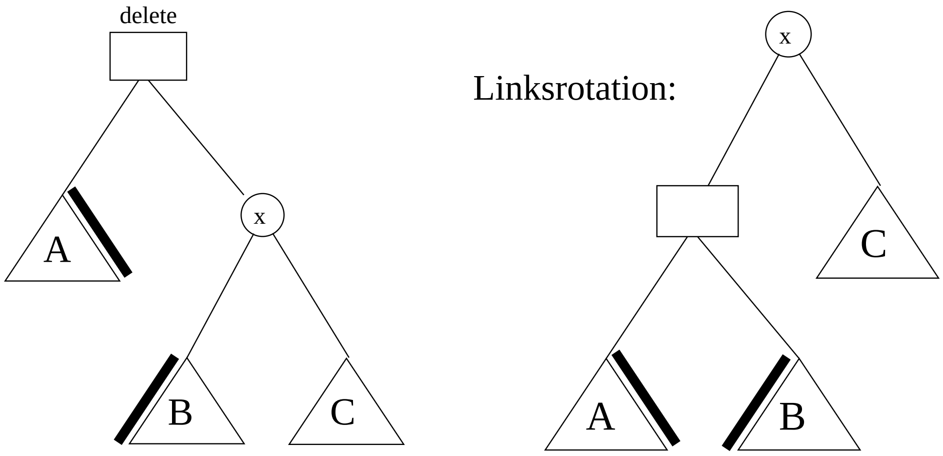



Abbildung 4.14: Linksrotation

bzw. linken Grat der Unterbäume befinden. Für $x_i$ mit Schlüsselrang $i$ (also, in dem der $i$-kleinste Schlüssel gespeichert wird) sei dazu

$L_i$ die Länge des rechten Grats des linken Unterbaumes von $x_i$

$R_i$ die Länge des linken Grats des rechten Unterbaumes von $x_i$.

Mit der Indikatorfunktion

$$M_{ij} = \begin{cases} 1 & \text{falls } x_j \text{ auf rechtem Grat des linken Unterbaumes von } x_i \\ 0 & \text{sonst} \end{cases}$$

für alle $1 \leq j < i$ gilt dann $L_i = \sum_{1 \leq j < i} M_{ij}$, und für den Erwartungswert gilt

$$\mathrm{E}[L_i] = \sum_{1 \leq j < i} \underbrace{\mathrm{E}[M_{ij}]}_{\Pr(M_{ij}=1)} .$$

Wir müssen also noch die Wahrscheinlichkeit dafür berechnen, daß $x_j$ auf dem rechten Grat des linken Unterbaumes von $x_i$ liegt. Das ist genau dann der Fall, wenn $x_i$ ein Ahne von $x_j$ und $x_j$ ein Ahne von $x_\ell$ für alle $j \leq \ell < i$ ist (siehe Abbildung 4.15). Anders gesagt, muß $x_i$ der "linkeste" Ahne von $x_j$ sein, der "rechts" von $x_j$ steht. Mit dem Ahnenlemma 2 kann man nun leicht sehen, daß dies in einem Treap genau dann eintritt, wenn in der Menge $\{x_\ell | j \leq \ell \leq i\}$ $x_i$ die kleinste und $x_j$ die zweitkleinste Priorität hat.
Sind die Prioritäten unabhängige, gleichverteilte, stetige Zufallsvariablen, dann ist die Wahrscheinlichkeit dafür

$$\frac{1}{(i-j+1)} \cdot \frac{1}{(i-j)} ,$$

da mit Wahrscheinlichkeit $1/(i-j+1)$ die Variable $x_i$ die kleinste der $i-j+1$ Zufallsvariablen ist und mit Wahrscheinlichkeit $1/(i-j)$ die Variable $x_j$ die kleinste der restlichen $i-j$ Zufallsvariablen ist.
Also gilt in einem randomisierten Suchbaum

$$\Pr(M_{ij} = 1) = \frac{1}{(i-j+1)} \cdot \frac{1}{(i-j)} = \frac{1}{i-j} - \frac{1}{i-j+1} ,$$

und daraus folgt

$$\begin{aligned} \mathrm{E}[L_i] &= \sum_{1 \leq j < i} \Pr(M_{ij} = 1) \\ &= \sum_{1 \leq j < i} \left( \frac{1}{i-j} - \frac{1}{i-j+1} \right) \qquad \text{(Teleskopsumme)} \\ &= 1 - \frac{1}{i} \\ &< 1 . \end{aligned}$$

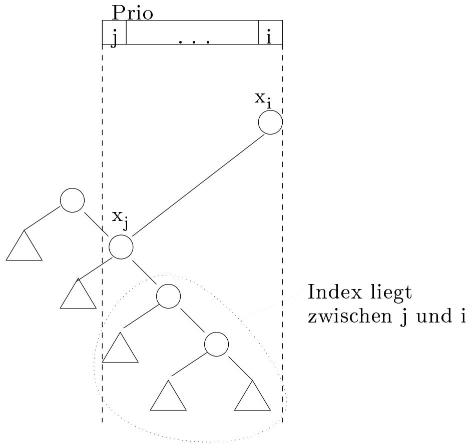



Abbildung 4.15: $x_j$ liegt auf dem rechten Grat des linken Unterbaums von $x_i$

Analog gilt für die erwartete Länge des linken Grats des rechten Unterbaums von $x_i$:

$$\mathrm{E}[R_i] = 1 - \frac{1}{n+1-i} < 1\,.$$

Die erwartete Anzahl der Rotationen pro Update ist also für jedes $i$ kleiner als zwei und damit konstant.

### 4.5.2 Teure Rotationen

In manchen Anwendungen von Bäumen, wie zum Beispiel bei Intervallbäumen, Bereichsbäumen oder Segmentbäumen, wird in jedem Knoten eine sekundäre Struktur abgespeichert, die bei einer etwaigen Rotation des Knotens neu berechnet werden muß. Dadurch kann es eintreten, daß die Zeitkosten einer Rotation nicht konstant sind. Nehmen wir also an, die Kosten einer Rotation eines Teilbaumes der Größe $s$ wären $f(s)$, wobei $f$ eine nicht-negative Funktion ist. Wie groß sind dann die erwarteten Kosten für ein Update in einem randomisierten Suchbaum?
Auf Grund der gegenseitigen Inversion von Einfügungen und Streichungen reicht es wieder, nur Streichungen zu analysieren. Seien dazu $R_f(x_k)$ die erwarteten Gesamtrotationskosten bei Streichung von $x_k$ unter der Annahme, daß die Rotation eines Baums der Größe $s$ die Kosten $f(s)$ verursacht. Wie groß ist $R_f(x_k)$?
Betrachte das Ereignis $E_{k:i,j}$ mit $1 \leq i \leq k \leq j \leq n$, daß während der DELETE-Operation von Knoten $x_k$ die Elemente $\{x_i, \ldots, x_j\}$ einen Teilbaum mit Wurzel $x_k$ bilden. Eine Rotation dieses Teilbaums würde $f(j-i+1)$ kosten. Wir definieren folgende Zufallvariable:

$$X_{k:i,j} = \begin{cases} f(j-i+1) & \text{falls } E_{k:i,j} \text{ auftritt,} \\ 0 & \text{sonst.} \end{cases}$$

Damit ist

$$\begin{aligned} R_f(x_k) &= \sum_{1\leq i\leq k} \sum_{k\leq j\leq n} E[X_{k:i,j}] \\ &= \sum_{1\leq i\leq k} \sum_{k\leq j\leq n} f(j-i+1)\cdot Pr(E_{k:i,j}) \end{aligned}$$

Es bleibt die Frage, wie groß $Pr(E_{k:i,j})$ für $1 \leq i \leq k \leq j \leq n$ ist. Wir werden dies wieder so angehen, daß wir zuerst charakterisieren, wie die Prioritäten aussehen müssen, damit so ein Ereignis eintritt, und daraus dann die Wahrscheinlichkeit berechnen. Betrachten wir zuerst den "normalen" Fall

**(a)** $1 < i \leq k \leq j < n$:
Wann gibt es überhaupt einen Teilbaum $\{x_i, x_{i+1}, \ldots, x_j\}$ mit Wurzel $x_k$? Aus dem Ahnenlemma 2 kann man leicht sehen, daß dies genau dann der Fall ist, wenn $x_{i-1}$ und $x_{j+1}$ die zwei kleinsten Prioritäten unter $x_{i-1}, x_i, \ldots, x_j, x_{j+1}$ haben und $x_k$ die drittkleinste Priorität hat (siehe Abbildung 4.16).

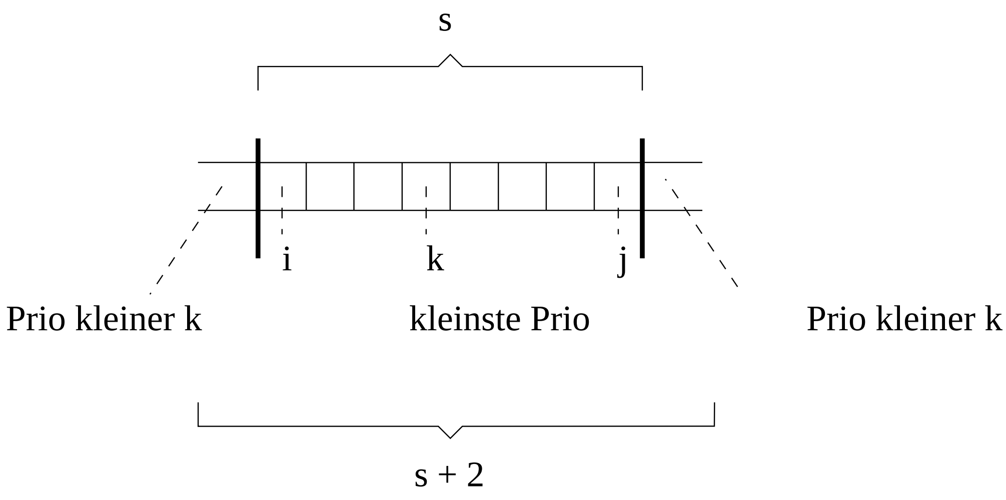



Abbildung 4.16: Teilbaum $\{x_i, x_{i+1}, \ldots, x_j\}$ mit Wurzel $x_k$

Wie kann es nun sein, daß so ein Teilbaum während der Streichung von $x_k$ auftritt. Man stelle sich dafür vor, daß bei der Streichung von $x_k$ seine Priorität immer größer wird und dadurch immer, wenn diese Priorität eine der Kinderprioritäten überschreitet, eine Rotation notwendig wird. Damit es also während des Wachsens der Priorität von $x_k$ zu der Situation kommt, daß $x_{i-1}$ und $x_{j+1}$ die zwei kleinsten Prioritäten unter $x_{i-1}, x_i, \ldots, x_j, x_{j+1}$ haben und $x_k$ die drittkleinste hat, mußten ursprünglich $x_{i-1}, x_{j+1}$ und $x_k$ die drei kleinsten Prioritäten unter $x_{i-1}, x_i, \ldots, x_j, x_{j+1}$ haben. Die Wahrscheinlichkeit dafür ist

$$\Pr(E_{k:i,j}) = \frac{6}{s \cdot (s+1) \cdot (s+2)} = O\left(\frac{1}{s^3}\right) ,$$

wenn $s = j - i + 1$, also $s$ die Größe des betrachteten Teilbaums ist.

In den restlichen drei Fällen erstreckt sich der zu betrachtende Teilbaum bis zu einem oder gar beiden der "Ränder" des Gesamtbaums. Die Idee der Analyse bleibt gleich, nur Details ändern sich.

**(b)** $1 = i \leq k \leq j < n$:
Ein Teilbaum $\{x_1, \ldots, x_j\}$ mit Wurzel $x_k$ tritt auf, wenn unter $x_1, \ldots, x_j, x_{j+1}$ der Knoten $x_{j+1}$ die kleinste Priorität hat und $x_k$ die zweitkleinste. Das Ereignis $E_{k:1,j}$ tritt dann ein, wenn $x_k$ und $x_{j+1}$ die zwei kleinsten Prioritäten bzgl. $x_1, \ldots, x_{j+1}$ haben, d.h.

$$\Pr(E_{k:1,j}) = \frac{2}{s \cdot (s+1)} = O\left(\frac{1}{s^2}\right) ,$$

wobei $s = j$ die Größe des betrachteten Teilbaums ist.

**(c)** $1 < i \leq k \leq j = n$:
Dieser Fall ist symmetrisch zu **(b)**, und wir bekommen

$$\Pr(E_{k:i,n}) = \frac{2}{s \cdot (s+1)} = O\left(\frac{1}{s^2}\right) ,$$

wobei $s = n - i + 1$ die Größe des betrachteten Teilbaums ist.

**(d)** $1 = i \leq k \leq j = n$:
Offensichtlich kann das Ereignis $E_{k:1,n}$ nur eintreten, wenn $x_k$ die kleinste aller Prioritäten hat, d.h.

$$\Pr(E_{k:1,n}) = \frac{1}{n} .$$

Nachdem wir all diese Wahrscheinlichkeiten berechnet haben, ist es im Prinzip möglich einen genauen Ausdruck für die erwarteten Gesamtrotationskosten zu geben. Wir begnügen uns hier mit einer Abschätzung. Wie oben hergeleitet gilt

$$\begin{aligned} R_f(x_k) &= \sum_{1 \leq i \leq k} \sum_{k \leq j \leq n} \Pr(E_{k:i,j}) \cdot f(j-i+1) \\ &= \sum_{1 \leq s \leq n} \sum_{\substack{1 \leq i \leq k \leq j \leq n \\ s=j-i+1}} \Pr(E_{k:i,j}) \cdot f(s) \,. \end{aligned}$$

Wir machen diese Umordnung der Summe, da in jedem der Fälle **(a)** bis **(d)** die Wahrscheinlichkeit des Ereignisses $E_{k:i,j}$ nur von $j-i+1$ abhängt.

Für fixes $k$ und $s$ kann es nun höchstens $s$ Ereignisse $E_{k:i,j}$ vom Typ **(a)** mit $j-i+1 = s$ geben (siehe Abbildung 4.17), jedes hat Wahrscheinlichkeit $O(1/s^3)$; es kann höchstens ein Ereignis vom Typ **(b)** und ein Ereignis vom Typ **(c)** geben mit $j-i+1 = s$, jeweils mit Wahrscheinlichkeit $O(1/s^2)$; schließlich gibt es genau ein Ereignis vom Typ **(d)**, welches mit Wahrscheinlichkeit $1/n$ auftritt. Insgesamt ergibt sich also

$$\begin{aligned} R_f(x_k) &= f(n) \cdot \frac{1}{n} + \sum_{1 \leq s < n} f(s) \left[ s \cdot O\left(\frac{1}{s^3}\right) + 1 \cdot O\left(\frac{1}{s^2}\right) + 1 \cdot O\left(\frac{1}{s^3}\right) \right] \\ &= \frac{f(n)}{n} + O\left( \sum_{1 \leq s < n} \frac{f(s)}{s^2} \right) . \end{aligned}$$

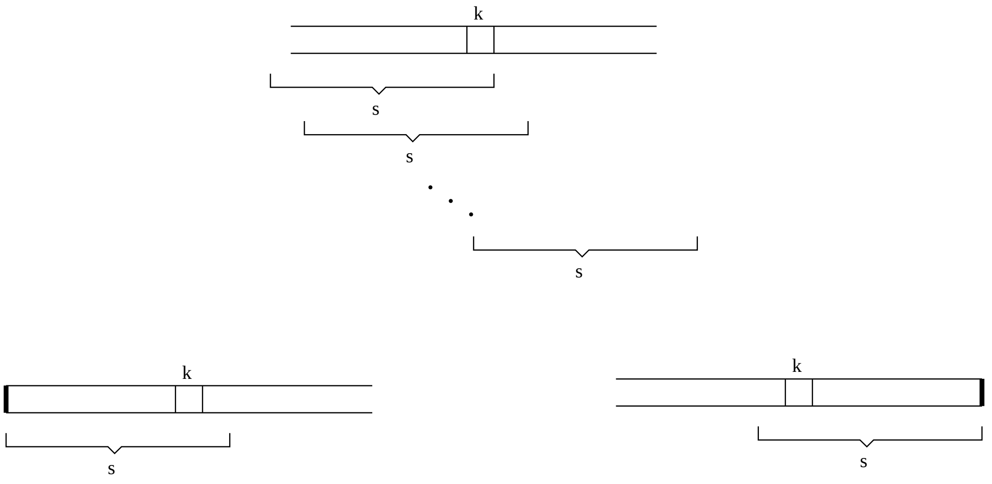



Abbildung 4.17: Möglichkeiten, Intervalle der Länge $s$ anzuordnen, so daß $k$ im Intervall liegt

Betrachten wir zwei Beispiele: Wenn $f(s) = 1$ für alle $s$, das ist der im letzten Abschnitt behandelte Fall von konstanten Rotationskosten, dann ergibt diese Formel

$$\frac{1}{n} + O\left( \sum_{1 \leq s < n} \frac{1}{s^2} \right) = O(1) \,.$$

Wenn $f(s) = s$, die Rotationskosten also gleich der Größe des rotierten Teilbaums sind, dann ergibt diese Formel

$$\frac{n}{n} + O\left( \sum_{1 \leq s < n} \frac{s}{s^2} \right) = O(\log n) \,.$$

Dies bedeutet, daß in randomisierten Suchbäumen Updates asymptotisch gesehen um nichts teurer werden, wenn die Kosten einer Rotation so hoch wie die Größe des rotierten Teilbaums werden.

Vorlesung vom 7.12.1995
Henrik Stormer, Ulrich Klos

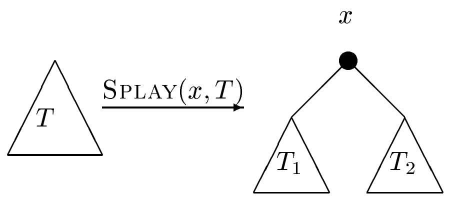
## 4.6 Splay-Bäume

Splay-Bäume stellen eine dritte Möglichkeit dar, binäre Suchbäume zu verwalten. Im Gegensatz zu den vorher gezeigten Methoden kann man aber bei Splay-Bäumen keine guten Aussagen über die Laufzeit von *einzelnen* Operation machen. Man kann nur gute *amortisierte* Aussagen mache, d.h. man kann gute Schranken angeben wie lange eine Folge von Operationen höchstens brauchen kann. Splay-Bäume haben eine ganze Menge von interessanten Eigenschaften, die wir hier nicht alle beleuchten können. Der ursprüngliche Artikel von D.D. Sleator und R.E. Tarjan ("Self-Adjusting Binary Search Trees" *Journal of the ACM* 32(3):652–686 (1985)) ist sehr zu empfehlen.
Splay-Bäume stellen folgende Funktionen bereit:

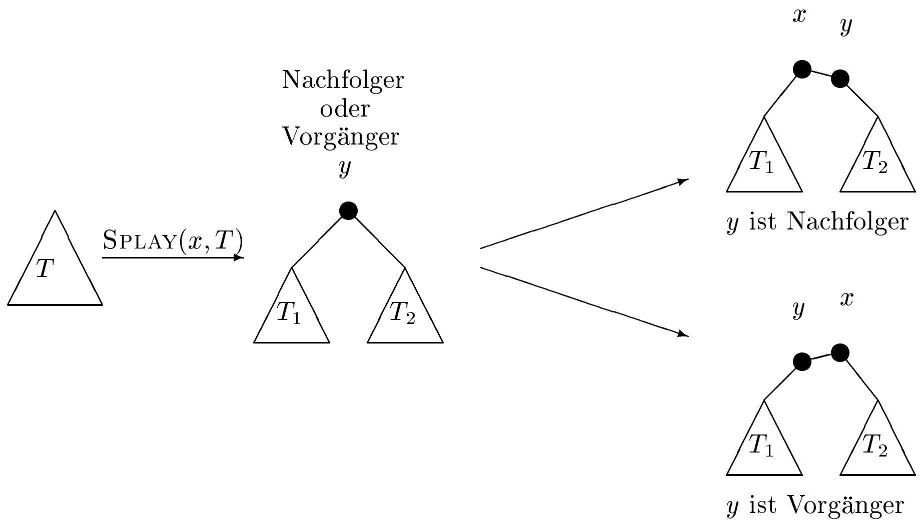
- ACCESS$(x, T)$
- INSERT$(x, T)$
- DELETE$(x, T)$
- SPLIT$(x, T)$ (Teilt den Baumes $T$ in zwei Teilbäume $T_1$ und $T_2$ auf, so daß $T_1$ alle Elemente aus $T$ enthält, die Schlüssel $\leq x$ haben, und $T_2$ alle Elemente enthält, die Schlüssel $> x$ haben.)
- JOIN $(T_1, T_2)$ (Inverse Operation zu SPLIT)
- SPLAY$(x, T)$ (Verändert den Baum so, daß $x$ zur Wurzel wird, oder, falls $x$ nicht in $T$ vorhanden ist, wird der direkte Vorgänger oder Nachfolger von $x$ in $T$ zur Wurzel.)

Mit Hilfe von SPLAY und *konstantem zusätzlichen Aufwand* kann man, wie die folgenden Skizzen zeigen, die anderen Operationen leicht implementieren:

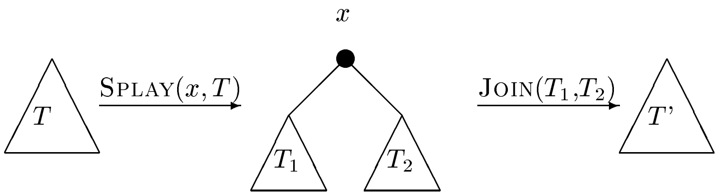
ACCESS$(x, T)$:



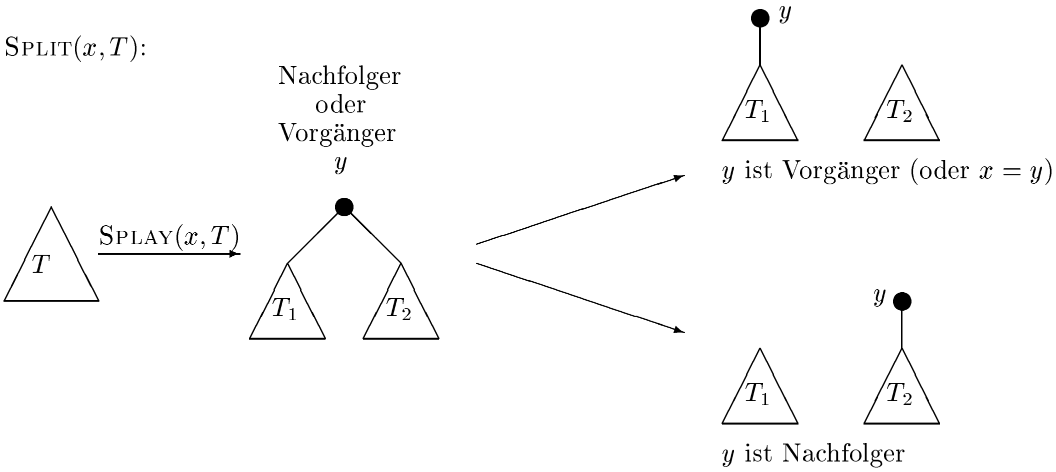

INSERT$(x, T)$:



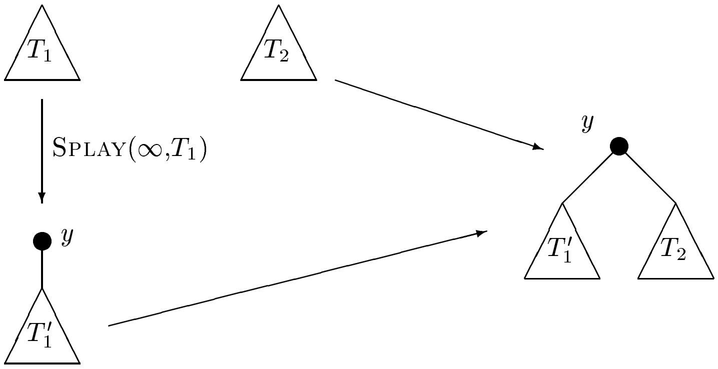
DELETE$(x, T)$:




SPLIT$(x, T)$:



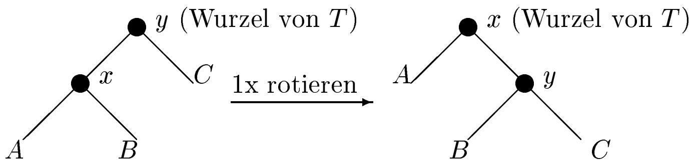

JOIN$(T_1, T_2)$:



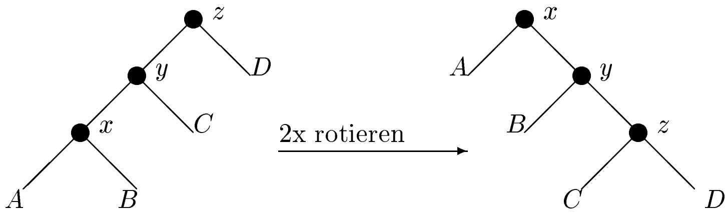

### 4.6.1 Implementierung von SPLAY

Beim Ausführen von SPLAY$(x, T)$ sucht man zunächst im Baum nach dem Element $x$. Wenn $x$ im Baum nicht gefunden wurde, befindet man sich bei dessen Vorgänger oder Nachfolger und wir bezeichnen dann diesen Knoten im folgenden mit $x$. Dann wird $x$ zur Wurzel hin bewegt. Das geschieht durch wiederholtes Anwenden, der sogenannten "Zick-Zick" und "Zick-Zack" Operationen, die $x$ jeweils um zwei Schritte näher zur Wurzel bringen. Nötigenfalls (wenn keine der beiden Operationen mehr anwendbar und $x$ Kind der Wurzel ist) folgt dann noch eine "Zick" Operation. Diese drei Operationen lassen sich folgendermaßen skizzieren. Bitte beachten Sie, daß es zu jeder dieser drei Operationen noch die symmetrische Variante gibt, in der $x$ als rechtes Kind von $y$ beginnt.

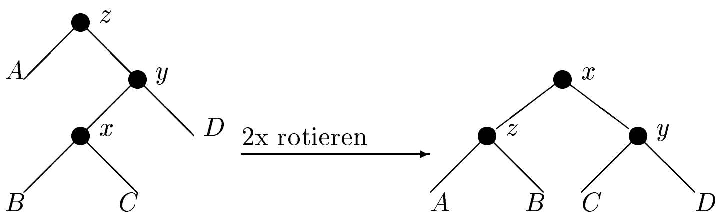
Fall 1: Zick



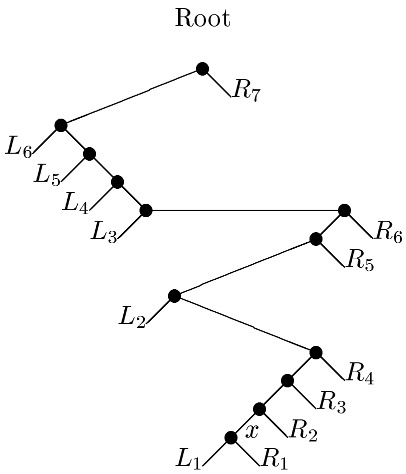

Fall 2: Zick-Zick



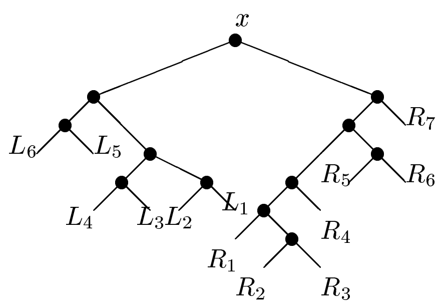

Fall 3: Zick-Zack




Als Beispiel betrachte man den folgenden Baum, von dem wir im wesentlichen nur nur den Pfad von der Wurzel zum Knoten $x$ aufgezeichnet haben zusammen mit den daran hängenden Teilbäumen.



Nach Anwendung der Operation SPLAY sieht dieser Baum (und der neue Pfad) folgendermaßen aus:




Man beachte, wie zum Beispiel aus dem Teilpfad im ursprünglichen Baum mit den herabhängenden Teilbäumen $L_3, L_4, L_5, L_6$ der Länge 4 zu einem viel besser balancierten Teilbaum geworden ist. Im allgemeinen kan man erkennen, daß sich nach der Splay-Operation die meisten Knoten auf dem ursprünglichen Pfad zwischen Wurzel und $x$ sich um ungefähr die Hälfte zur Wurzel genähert haben (und mit ihnen die daran hängenden Teilbäume)verkürzt haben. Man sollte sich klar machen, daß nirgendwo angenommen wurde, daß der Baum balanciert war, noch daß er nach der Splay-Operation balanciert ist.

### 4.6.2 Amortisierte Analyse von SPLAY

Im Folgenden wollen wir nun die Laufzeit von SPLAY untersuchen. Es wird dabei folgende Behauptung aufgestellt (die wir natürlich noch verfeinern müssen):

**Behauptung 13** *Eine Folge von Splay- und/oder Update-Operationen dauert insgesamt nicht zu lange.*

Hierbei ist das Wort *insgesamt* von besonderer Wichtigkeit, denn wir betrachten nur eine Folge von Splay Befehlen. Es ist nicht gesagt, daß es dabei keine “Ausreißer” geben kann. Insgesamt gesehen wird die Behauptung aber wahr. Um dieses zu beweisen, definieren wir uns nun zuerst die amortisierten Kosten einer Operation.

**Definition 6** *Sei $\Phi$ das Potential einer Datenstruktur, $\Phi$ : Bäume $\longrightarrow \mathbb{R}^+$. Die amortisierten Kosten $a$ einer Operation sind*

$$a = t + \Phi' - \Phi,$$

*wobei $t$ die tatsächlichen Kosten der Operation sind, $\Phi$ das Potential der Datenstruktur vor und $\Phi'$ das Potential der Datenstruktur nach der Operation ist.*

Als nächstes betrachten wir nun eine Folge von Operationen $Op_1, \ldots, Op_n$. Die zugehörigen amortisierten Kosten sind:

$$\begin{aligned} \sum_{i=1}^{m} a_i &= \sum_{i=1}^{m} (t_i + \Phi'_i - \Phi_i) \\ &= \left( \sum_{i=1}^{m} t_i \right) + \Phi'_m - \Phi_1, \end{aligned}$$

da $\Phi'_i = \Phi_{i+1}$. Also gilt:

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i + \Phi_1.$$

Das heißt, die tatsächlichen Gesamtkosten sind von oben durch die amortisierten Gesamtkosten und dem Anfangspotential beschränkt. Um dies auszunutzen brauchen wir eine gute Potentialfunktion $\Phi$. Dazu nehmen wir an, daß jeder Knoten $x$ ein positives Gewicht $w(x)$ besitzt (zum Beispiel $w(x) = 1$). Dann definieren wir:

$$\begin{aligned} W(z) &= \sum_{y \text{ in Teilbaum von } z} w(y) \\ r(z) &= \log_2 W(z) \\ \Phi(T) &= \sum_{z \text{ Knoten in } T} r(z) \end{aligned}$$

Wenn $w(z) = 1$ für alle $z$ ist, dann ist zum Beispiel $\Phi(T) \leq n \log_2 n$ mit $n$ geich Anzahl der Knoten in $T$, ganz gleich wie $T$ gestaltet ist.
Mit diesen Definitionen können wir nun folgendes Lemma aufstellen:

**Lemma 5** *Die amortisierten Kosten von* SPLAY$(x, T)$ *sind höchstens* $3(r(t) - r(x)) + 1 = O(1 + \log W(t)/w(x))$, *wobei $t$ die Wurzel von $T$ ist.*

Als Beispiel sei $w(z) = 1$ für alle Knoten $z$ in $T$. Dann folgt aus dem Lemma, daß die amortisierten Kosten von SPLAY$(x, T)$

$$3(\log_2 \underbrace{|T|}_{n} - \log_2 |T_x|) + 1 = O(\log_2 n)$$

sind. Das heißt, die tatsächlichen Kosten für $m$ SPLAYs sind dann $O(\underbrace{m \log_2 n}_{\sum a_i} + \underbrace{n \log_2 n}_{\Phi_1})$.

Da INSERT, DELETE, ACCESS, SPLIT und JOIN jeweils durch ein SPLAY und konstantem zusätzlichen Aufwand implementiert werden können, ergibt sich folgendes:

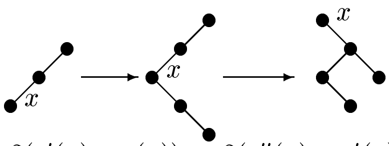
**Satz 12** *Ein beliebige Folge von $m$ Operationen auf einem Splay-Baum, der dabei maximale Größe $n$ erreicht, wird in Zeit $O((m + n) \log n)$ abgearbeitet.*

Nun zum Beweis des Lemmas:
Ein Schritt einer SPLAY-Operation ist die Durchführung eines Zick-, Zick-Zick- oder Zick-Zack-Schrittes. Sei $r(x)$ der Rang von $x$ vor und $r'(x)$ der Rang von $x$ nach dem Schritt.

**Behauptung 14** *Ein Zick-Schritt hat amortisierte Kosten* $\leq 3(r'(x) - r(x)) + 1$.

**Behauptung 15** *Ein Zick-Zick- oder Zick-Zack-Schritt hat amortisierte Kosten* $\leq 3(r'(x) - r(x))$.

Mit diesen zwei Behauptungen ist das Lemma bewiesen, denn bei mehrfacher Anwendung der Schritte ziehen sich die Ränge von $x$ in einer Teleskopsumme zusammen,




$3(r'(x) - r(x))$ $3(r''(x) - r'(x))$
gesamt: $3(r''(x) - r(x))$

und übrig bleibt:

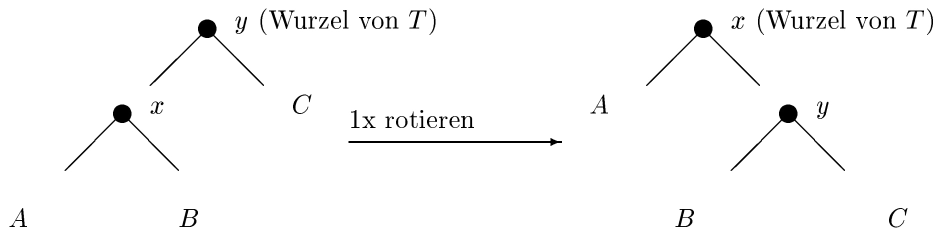
$$3 \cdot (\text{Endrang von } x - \text{ Anfangsrang von } x) + 1\,.$$

Da $x$ zum Schluß die Wurzel von $T$ ist, sind die Gesamtkosten:

$$3(r(t) - r(x)) + 1.$$

Es folgen jetzt noch die Beweise der beiden Behauptungen. In allen drei Fällen ergeben sich die amortisierten Kosten aus der Anzahl der Rotationen sowie der Rangveränderung der beteiligten Knoten. Man beachte, daß für ein Kind (oder einen Nachkommen) $x$ von $y$ immer gilt $r(x) \leq r(y)$.

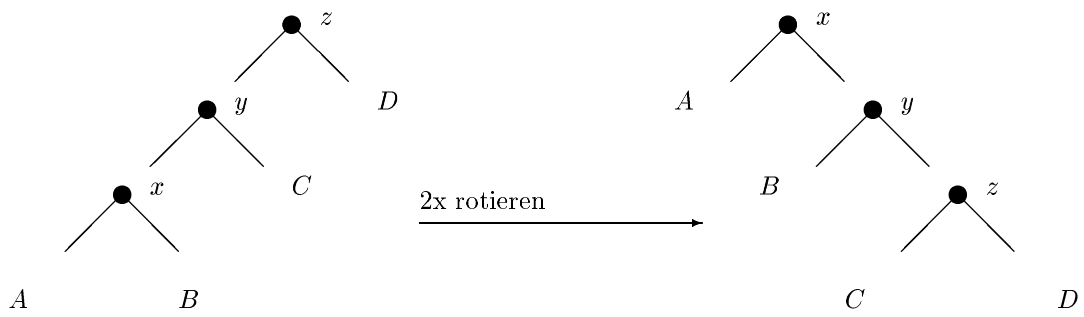
1. Der Zick-Fall:




Die amortisierten Kosten sind:

$$\underbrace{1}_{\substack{\text{Da nur eine}\\\text{Rotation nötig}}} +r'(x)+r'(y)-r(x)-r(y)$$
$$\overset{r'(x)=r(y)}{=} \quad 1+r'(y)-r(x)$$
$$\overset{r'(y)\le r'(x)}{\le} \quad 1+r'(x)-r(x)$$
$$\overset{r(x)\le r'(x)}{\le} \quad 1+3(r'(x)-r(x)).$$

2. Der Zick-Zick-Fall:




Die amortisierten Kosten sind:

$$\underbrace{2}_{\substack{\text{Da zwei}\\\text{Rotationen nötig}}} +r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)$$
$$\overset{r'(x)=r(z)}{=} \quad 2+r'(y)+r'(z)-r(x)-r(y)$$
$$\overset{\substack{r'(x)\ge r'(y)\\ r(y)\ge r(x)}}{\le} \quad 2+r'(x)+r'(z)-2r(x)$$
$$\overset{?}{\le} \quad 3(r'(x)-r(x))$$

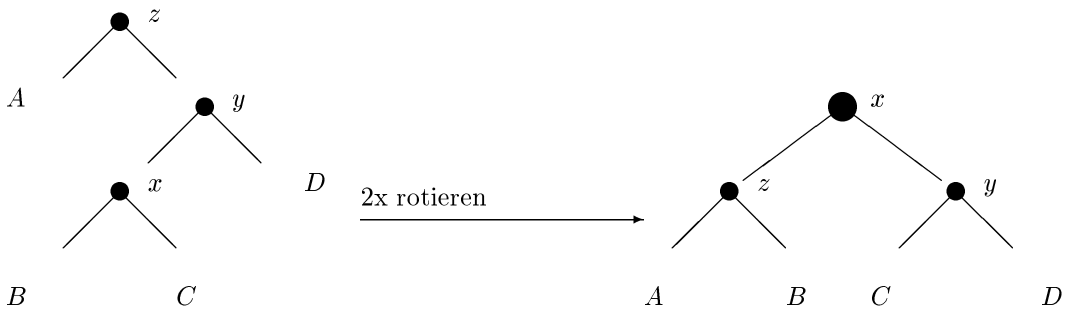
Durch Subtraktion der vorletzten von der letzten Zeile ergibt sich also folgende äquivalente Ungleichung:

$$-2+2r'(x)-r'(z)-r(x) \overset{?}{\ge} 0$$
$$\Leftrightarrow \quad r'(z)+r(x)-2r'(x) \overset{?}{\le} 2$$

$$
\begin{aligned}
& \text{Es gilt aber } r'(z) = \log_2 W'(z) \\
\Leftrightarrow \quad & \log_2 W'(z) + \log_2 W(x) - 2\log_2 W'(x) \overset{?}{\leq} -2 \\
\Leftrightarrow \quad & \log_2 \underbrace{\left(\frac{W'(z)}{W'(x)}\right)}_{\geq 0, \leq 1} + \log_2 \underbrace{\left(\frac{W(x)}{W'(x)}\right)}_{\geq 0, \leq 1} \overset{?}{\leq} -2
\end{aligned}
$$

Diese letzte Ungleichung ist wahr: Denn links steht ein Ausdruck der Form $\log a + \log b$ mit $0 \leq a, b \leq 1$ und $a + b \leq 1$ (da $W'(z) + W(x) \leq W'(x)$), und der Term $\log a + \log b$ nimmt sein Maximum für $a = b = 1/2$ an.

3. Der Zick-Zack-Fall:




Die amortisierten Kosten sind:

$$
\begin{aligned}
& \underbrace{2}_{\substack{\text{Da zwei} \\ \text{Rotationen nötig}}} && + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
& \overset{\substack{r'(x)=r(z) \\ r(x) \leq r(y)}}{\leq} && 2 + r'(y) + r'(z) - 2r(x) \\
& \overset{?}{\leq} && 2(r'(x) - r(x))
\end{aligned}
$$

Die Rechnung verläuft analog zum Fall vorher.

Vorlesung vom 11.12.1995
Jürgen Zimmer

### 4.6.3 Statische Optimalität der Splay-Methode

Nach der Einführung der Splay-Methode bleibt die Frage, ob sie auch wirklich einen Kostenvorteil gegenüber den herkömmlichen Methoden der „einfachen“ binären Suchbäume bringt. Man untersucht deshalb die Kosten der Splay-Methode für eine fixe Folge Such-Operationen und vergleicht sie mit den Kosten, die die gleiche Folge auf einem anderen fixen, nicht veränderbaren, statischen Suchbaum verursachen würde.
Dazu seien im folgenden:

- $S$ eine Menge von $n$ Schlüsseln.
- $C$ eine fixe Folge von Zugriffsoperationen auf Elemente von $S$.
- $B$ irgendein binärer Suchbaum für die Folge $C$. $B$ kann für die Folge $C$ optimal gewählt sein. Dann wären die Kosten für die Ausführung der Operationen von $C$ minimal.
- $t$ die Anzahl der Vergleiche, die benötigt werden, um $C$ auf $B$ abzuarbeiten.

Es gilt folgender *statischer Optimalitätssatz* für Splaybäume:

**Satz 13** *Ist $T$ ein beliebiger Suchbaum für die Menge $S$, und wird die Zugriffsfolge $C$ mit Hilfe der Splay-Methode und dem Startbaum $T$ abgearbeitet, dann sind die Kosten $K_{splay}$ begrenzt durch*

$$\boxed{K_{splay} = O(t + n^2)}$$

Anmerkung:
Dieser Satz besagt, daß die Kosten bei der Abarbeitung von $C$ mittels der Splay-Methode auf einem beliebigen (nicht unbedingt optimalen) Baum $T$ nicht sehr viel größer sind als bei der „konventionellen“ Abarbeitung auf irgendeinem, insbesondere auch **optimalen** binären Suchbaum.
Insbesondere sind die Kosten asymptotisch gleich, wenn $t = \Omega(n^2)$, d.h. wenn die Folge $C$ hinreichend lang ist. Wenn man mehr als $n^2$ Vergleich braucht, so fallen die Kosten, die durch die Splay-Operationen entstehen, kaum ins Gewicht.
Es ist ein schon lange offenes Problem, ob diese Aussage auch für den dynamischen Fall gilt, in dem sich der Baum $B$, während die Folge $C$ abgearbeitet wird, verändern darf.
**Beweis** Es seien:

$d$ die Tiefe (Höhe) von $B$.

$d(x)$ bezeichne für alle $x$ aus $S$ die Tiefe von $x$ in $B$ (d.h. $d(x)$ ist die Länge des Pfades von der Wurzel von $B$ zu $x$).

$w(x) := 3^{d-d(x)}$ sei die (geschickt gewählte) Gewichtsfunktion für die Elemente $x$ aus $S$.

Man interessiert sich zunächst für das Anfangspotential $\Phi(T)$ des Baumes $T$, denn mit ihm kann man — mit Hilfe der amortisierten Kosten — die tatsächlichen Kosten abschätzen.
Zur Erinnerung :

$$\Phi(T) = \sum_{x \in S} r(x) = \sum_{x \in S} \log_2 W(x)\,, \qquad \text{wobei}$$

$$W(x) = \sum_{y \in T_x} w(y) \qquad \text{und } T_x \text{ ist der Unterbaum mit Wurzel } x.$$

Das Gesamtgewicht $W(x)$ eines Knotens $x$ kann man nach oben durch das Gewicht der Wurzel von $T$ abschätzen, für das man folgende Behauptung zeigen kann.
**Behauptung:** Für das Gewicht der Wurzel von $T$ gilt :

$$\boxed{W(root(T)) = \sum_{x \in S} w(x) \;\leq\; 3^{d+1}}$$

**Beweis** Man sieht:

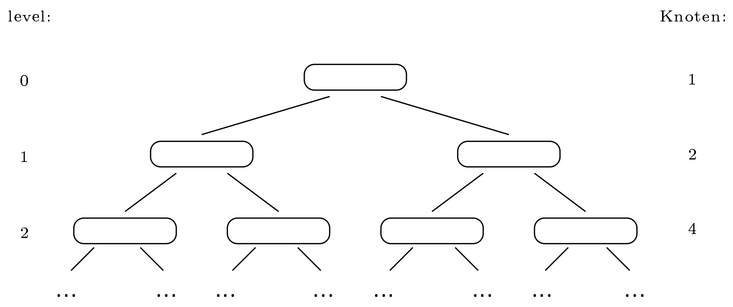



Abbildung 4.18: Ein vollständiger binärer Suchbaum

Es befinden sich höchstens $2^i$ Knoten auf der Ebene $i$ im Baum $B$ (siehe Abbildung 4.6.3) und jeder von ihnen hat nach Definition von $w$ das Gewicht $3^{d-i}$, also gilt für die Wurzel von $T$:

$$W(root(T)) = \sum_{x \in S} w(x) \;=\; \sum_{0 \leq i \leq d} 2^i 3^{d-i} \;\leq\; 3^d \underbrace{\sum_{0 \leq i \leq d} \left(\frac{2}{3}\right)^i}_{\leq 3} \;\leq\; 3^{d+1}$$

$\square$

Da das Gewicht $W(x)$ für einen Knoten $x \neq root(T)$ höchstens kleiner wird, gilt:

$$W(x) \leq 3^{d+1} \qquad \forall x \in S.$$

Also gilt (da die Tiefe des Baumes $T$ höchstens $n$ ist):

$$\underline{\underline{\Phi(T)}} = \sum_{x \in S} \log W(x) \leq \sum_{x \in S} O(d+1) = O(n \cdot (d+1)) = \underline{\underline{O(n^2).}}$$

Mit der Gewichtsfunktion $w$ hat der Baum $T$ also ein Anfangspotential $O(n^2)$.
Die Kosten für einen Zugriff auf ein $x \in S$ in unserem Suchbaum $B$ sind natürlich $O(d(x))$. Für die amortisierten Kosten $K_x$ eines Zugriffs auf $x$ im Baum $T$ mittels der Splay-Methode gilt:

$$K_x = O\left(1 + \log \frac{W(root(T))}{w(x)}\right).$$

Mit $w(x) = 3^{d-d(x))}$ und $W(root(T)) \leq 3^{d+1}$ folgt:

$$K_x = O\left(1 + \log\left(3^{1+d(x)}\right)\right) = O(2 + d(x)) = O(d(x)).$$

Daher sind die amortisierten Kosten $K_a$ für die Ausführung aller Operationen von $C$ mit Hilfe der Splay-Methode und Startbaum $T$ auch $O(t)$. Sie sind also asymptotisch gleich den Kosten ($t$ Vergleiche), die bei dem binären Suchbaum $B$ anfallen.
Bei der Definition der amortisierten Kosten wurde bereits gezeigt, daß für die tatsächlichen Kosten einer Folge von Operationen gilt:

$$\sum_i t_i \leq \sum_i^{a_i} + \Phi\,,$$

wobei $\Phi$ das Anfangspotential der Datenstruktur (in unserem Fall der Baum $T$) ist. Die $t_i$ sind die tatsächlichen und die $a_i$ die amortisierten Kosten der Operation $i$.
Folglich gilt für die tatsächlichen Kosten der Splay-Methode:

$$K_{splay} \leq K_a + \Phi(T) = O(t + n^2).$$

$\square$

## 4.7 Optimale Suchbäume

Aus dem vorherigen Abschnitt ergibt sich die nach Suchbäumen, die für eine vorgegebene Folge von Zugriffen optimal sind. Diese Frage wollen wir hier kurz behandeln.
Zu einer gegebenen Schlüsselmenge, für die man weiß, wie oft auf die einzelnen Elemente zugegriffen wird, kann man sich einen Suchbaum aufbauen, der für diese Menge, was die Kosten von Operationen auf diesem Baum anbelangt, optimal ist. Im Prinzip plaziert man einfach die Elemente, auf die am häufigsten zugegriffen wird, möglichst weit oben im Baum.
Im folgenden sei $S := \{x_1, \ldots, x_n\} \subseteq M$ eine Menge von Schlüsseln mit $x_1 < x_2 < \ldots < x_n$. An Stelle zu sagen, wie oft in einer Folge auf ein Element zugegriffen wird, werden wir Wahrscheinlichkeiten annehmen, die die relative Häufigkeit angeben, mit der auf die Elemente aus $S$ und auch auf die Elemente aus $M \setminus S$ zugegriffen wird:

$\beta_i$ sei die Wahrscheinlichkeit dafür, daß bei einer Suchoperation nach $x_i$ gesucht wird.

$\alpha_i$ sei die Wahrscheinlichkeit dafür, daß nach einem Schlüssel zwischen $x_i$ und $x_{i+1}$ gesucht wird, wobei $x_0 := -\infty$ und $x_{n+1} := \infty$ gesetzt wird.

Man kann also einen Suchbaum für die Menge $S$ auf folgende Art und Weise schematisch darstellen (siehe Abbildung 4.7:
Außerdem definiert man folgende Werte:

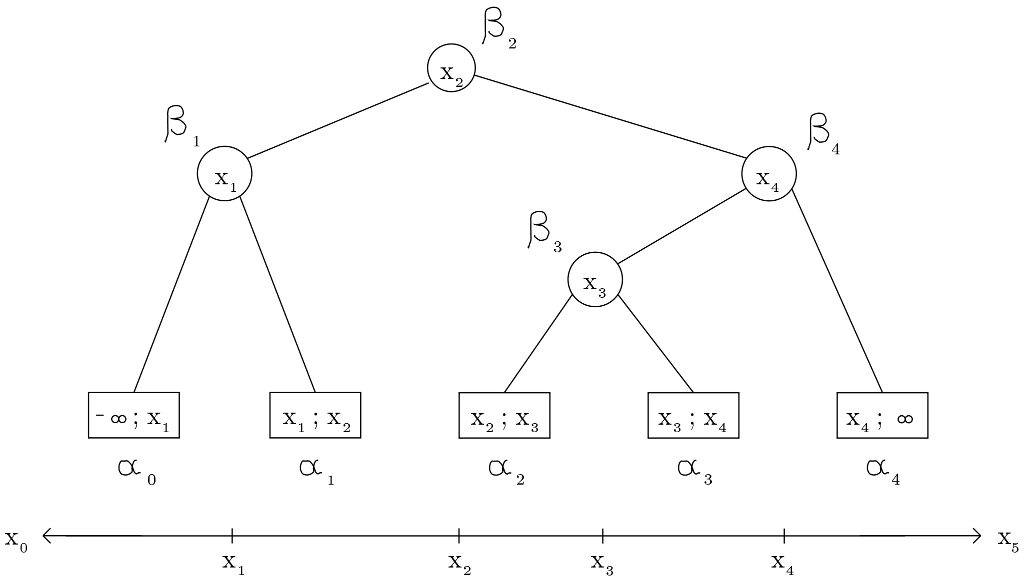



Abbildung 4.19: Schematische Darstellung eines Suchbaumes

- $b_i(T)$ soll die Tiefe von $x_i$ im Baum $T$ sein (also Kantenanzahl auf dem Weg von $x_i$ zur Wurzel)
- $a_i(T)$ die Tiefe vom Blatt $\boxed{x_{i-1}; x_i}$ in $T$.

Für die Suche nach einem Element $x_i$ sind also $\underbrace{1}_{root} + b_i(T)$ Vergleiche erforderlich. Für eine erfolglose Suche, die in dem Blatt $\boxed{x_{i-1}; x_i}$ endet, braucht man $a_i(T)$ Vergleiche.
Um nun für die gegebene Menge $S$ und die Zugriffs-Wahrscheinlichkeiten $\alpha_i$ und $\beta_i$ einen optimalen Suchbaum aufzubauen, definiert man zunächst den Begriff der gewichteten Pfadlänge, die der erwarteten Anzahl von Vergleichen entspricht, die man bei einer beliebigen Such-Operation durchführen muß.

**Definition 7** *Für einen binären Suchbaum $T$ für eine Menge $S$ von $n$ Schlüsseln mit den Zugriffswahrscheinlichkeiten $\alpha_i$ und $\beta_j$ heißt*

$$P(T) := \sum_{i=1}^{n+1} \alpha_i \, a_i(T) + \sum_{j=1}^{n} \beta_j (1 + b_j(T))$$

*die gewichtete Pfadlänge des Baumes $T$.*

Nun kann man auch den Begriff des optimalen Suchbaumes genauer festlegen ...

**Definition 8** *Ein binärer Suchbaum $T$ für die Menge $S$ heißt optimal, wenn seine gewichtete Pfadlänge minimal ist.*

Wie findet man nun bei gegebener Schlüsselmenge $S$ und den Zugriffswahrscheinlichkeiten $\alpha_i$ und $\beta_j$ einen optimalen Suchbaum ? Eine Lösung ist die, daß man sich in einem Teilbaum, der die Elemente $x_i, \ldots, x_j$ enthält, die Wurzel „geschickt" wählt, d.h. man nimmt als Wurzel genau den Knoten, für den die gewichteten Pfadlängen der Unterbäume minimal werden. Mit dieser Methode kann man sich einen optimalen Suchbaum für die gesamte Menge $S$ - also die „Knoten" $x_1 \ldots x_n$ - induktiv von den Blättern bis zur Wurzel aufbauen. Etwas formaler kann man das Verfahren folgendermaßen beschreiben, wobei im folgenden immer $\underline{i \leq j}$ sein soll. Man definiert:

1. $W_{i,j} := \alpha_{i-1} + \beta_i + \alpha_i + \ldots + \beta_j + \alpha_j$
   $W_{i,j}$ ist also die Wahrscheinlichkeit dafür, daß für den Suchschlüssel $x$ gilt:
$$x_{i-1} < x < x_{j+1}.$$

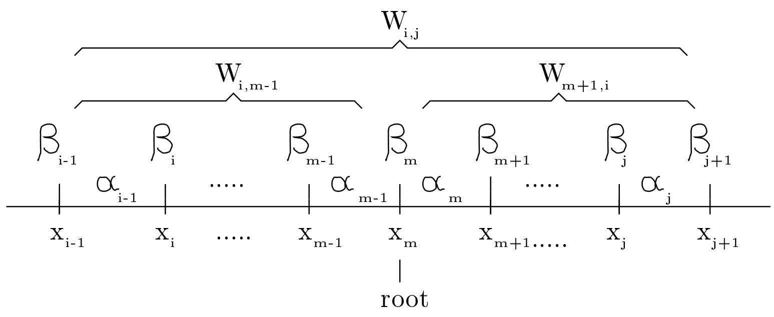



Abbildung 4.20: Die Wahrscheinlichkeiten $W_{i,m-1}$ , $W_{m+1,j}$ und $W_{i,j}$

2. $P_{i,j}$ := die optimale Pfadlänge des Baumes für die Schlüssel im offenen Intervall $(x_i, , x_j)$ bzgl. den Zugriffswahrscheinlichkeiten

$$\overline{\beta_l} := \frac{\beta_l}{W_{i,j}} \quad \text{und} \quad \overline{\alpha_l} := \frac{\alpha_l}{W_{i,j}}$$

   Die $\overline{\beta_l}$ und $\overline{\alpha_l}$ sind normiert, damit die Summe der Wahrscheinlichkeiten wieder 1 ergibt und $P_{i,j}$ wohldefiniert ist.

**Lemma 6** *Für die gewichtete Pfadlänge $P_{i,j}$ des Unterbaums für die Knoten $x_i, \ldots, x_j$ gilt*

*i)* $P_{i,i-1} = 0$

*ii)* $P_{i,j} = 1 + \min\limits_{i \leq m \leq j} \left\{ P_{i,m-1} \cdot \frac{W_{i,m-1}}{W_{i,j}} + P_{m+1,j} \cdot \frac{W_{m+1,j}}{W_{i,j}} \right\}$

**Anmerkung:**

ad *i)* Die optimale Pfadlänge für die leere Menge $S$ ist natürlich 0, da man ja gar keinen Baum aufbauen kann.

ad *ii)* Um die optimale Pfadlänge für einen Baum, der die Elemente $x_i, \ldots, x_j$ enthält, zu finden, sucht man zuerst das Minimum über alle $i \leq m \leq j$ der optimalen Pfadlängen seines linken und rechten Unterbaumes und addiert noch 1 für den Weg von der Wurzel zu einem der Unterbäume.

Dabei ist $\frac{W_{i,m-1}}{W_{i,j}}$ (bzw.$\frac{W_{m+1,j}}{W_{i,j}}$) die (bedingte) Wahrscheinlichkeit dafür, daß die Suche nach dem Schlüssel $x$ im linken (rechten) Unterbaum des Baumes für $x_i, \ldots, x_j$ weitergeht.

In der Abbildung 4.7 sieht man, wie das Intervall $[i; j]$ durch $m$ aufgeteilt wird und wie sich die Wahrscheinlichkeiten $W_{i,m-1}$ und $W_{m+1,j}$ aus den $\beta_l$ und den $\alpha_l$ ergeben.
Mit Hilfe der gewichteten Pfadlänge und dem obigen Lemma, kann man leicht eine Prozedur schreiben, die für gegebene Wahrscheinlichkeiten $W_{i,j}$ rekursiv die optimale Pfadlänge berechnet. Dabei wird vorausgesetzt, daß die $W_{i,j}$ in einem Feld gespeichert sind, bzw:

$$W_{i,j} = -A_i + A_j + \alpha_j, \quad \text{,wobei} \quad A_k := \sum_{l<k} (\alpha_l + \beta_l).$$

Die Prozedur Opl (<u>O</u>ptimale <u>P</u>fad<u>l</u>änge) in Pseudo-Code:

```
procedure OPL(i, j)
    if j < i then return 0
    min ← ∞
    for m ← i to  j do
        s ← 1+OPL(i, m − 1) · W_{i,m−1}/W_{i,j} + OPL(m + 1, j) · W_{m+1,j}/W_{i,j}
        if s < min then  min ← s
    return  min
```

Wie man unschwer erkennen kann, ist diese Prozedur nicht sehr effektiv, da sie $\Omega(2^n)$ rekursive Aufrufe macht und dabei die meisten Werte $\text{OPL}(i,j)$ mehrfach berechnet (exponentielle Laufzeit!). Für $\text{OPL}(i,j)$ gibt es aber eigentlich nur $n^2$ verschiedene Eingabepaare ($n$ ist die Größe von $S$), so daß man eine schnellere Variante von OPL schreiben kann, die sich die schon berechneten Werte in einem Feld $Erg[1..n, 1..n]$ merkt und darauf zurückgreift. Das Feld soll beim ersten Aufruf mit *undef* initialisiert sein. Die etwas modifizierte Prozedur sieht dann so aus (die Änderungen sind eingerahmt):

**procedure** $\text{OPL2}(i,j)$
  **if** $j < i$ **then return** $0$
  $\boxed{\textbf{if } Erg[i,j] \neq undef \textbf{ then return } Erg[i,j]}$
  $min \leftarrow \infty$
  **for** $m \leftarrow i$ **to** $j$ **do**
    $s \leftarrow 1 + \text{OPL2}(i, m-1) \cdot \frac{W_{i,m-1}}{W_{i,j}} + \text{OPL2}(m+1, j) \cdot \frac{W_{m+1,j}}{W_{i,j}}$
    **if** $s < min$ **then** $min \leftarrow s$
  $\boxed{Erg[i,j] \leftarrow min}$
  **return** $min$ ;

In welcher Größenordnung liegt nun die Laufzeit von OPL2 ? Zur Analyse unterscheidet man zwischen definierenden (Erst-)Aufrufen, bei denen ein Eintrag in das Feld $Erg[\,]$ berechnet wird, und nicht-definierenden Aufrufen, bei denen nur auf einen schon berechneten Wert zurückgegriffen wird. Natürlich gibt es nur $O(n^2)$ definierende Aufrufe, von denen jeder $O(n)$ Zeit braucht (wenn man von der Zeit für die darin vorkommenden rekursiven Aufrufen absieht), denn es wird ein bereits berechneter Wert ja kein zweites mal berechnet. Also braucht dieser Teil von OPL2 höchstens $O(n^3)$ Zeit.
Ein nicht-definierender Aufruf braucht konstant viel Zeit ($O(1)$) und es kann höchstens $O(n^3)$ viele nicht-definierende Aufrufe geben, da sie nur aus der for-Schleife ($O(n)$) eines definierenden Aufrufs ($O(n^2)$) stammen können. Damit brauchen die nicht-definierenden Aufrufe auch nur $O(n^3)$ Zeit, so daß man insgesamt eine Laufzeit $O(n^3)$ erhält, was natürlich einen riesigen Gewinn gegenüber $O(2^n)$ darstellt.

Vorlesung vom 14.12.1995
Thorsten Mohr, Andreas Wendling

Der OPL-Algorithmus aus der letzten Vorlesung berechnet nur die optimale Pfadlänge, er sagt aber nichts darüber aus, wie der Baum aussehen soll. Um den Baum zu konstruieren, stellen wir ein zweidimensionales Feld $R$ zur Verfügung und speichern in $R[i,j]$ die Wurzel des (neuen) optimalen Suchbaumes für das Intervall $(x_{i-1}, x_{j+1})$. Der Algorithmus merkt sich dies an folgender Stelle:

$$* \textbf{ if } s < min \textbf{ then } \{min = s;\; R[i,j] = m\} *$$

Das braucht offensichtlich nur konstante zusätzliche Zeit in dieser Zeile. Die asymptotische $O(n^3)$ Laufzeit des Algorithmus änderst sich dadurch nicht.
Aus dem Feld $R[]$ läßt sich dann der Baum leicht durch eine rekursive Funktion in linearer Zeit konstruieren. Wir überlassen dies dem Leser als Übung.

Die Konstruktion eines Algorithmus für OPL kann man auch Bottom-Up angehen. Dazu berechnen wir die $P[i,j]$ in Reihenfolge von wachsenden $(j-i)$-Unterschieden, lassen also $l = j - i$ in einer Schleife von 0 bis $n-1$ laufen und berechnen im Rumpf alle $P[i,j]$ mit $j-i$ gleich $l$ Abbildung 4.21. Beim Top-Down Verfahren aus der letzten Vorlesung wurde rekursiv das große Anfangsproblem (großer Abstand zwischen $i$ und $j$) zuerst in kleinere Teilprobleme zerlegt, aus deren rekursive Lösung man dann die Gesamtlösung berechnet. Das angenehme an diesem rekursiven Ansatz ist, daß man sich überhaupt nicht überlegen muß, in welcher Reihenfolge verschiedene Probleme eigentlich gelöst werden müssen. Rekursion und das Speichern von schon berechneten Ergebnissen versteckt diese Details.
Beim Bottom-Up Ansatz muß man sich dagegen die Reihenfolge, in der Teilprobleme gelöst werden müssen, genau überlegen. Hier berechnen wir zuerst die kleinsten Teilprobleme ($l = 0$) und lösen mit deren Hilfe die nächstgrößeren ($l = 1$) u.s.w.. Schließlich haben wir dann das große Problem (d.h $l = n$) gelöst. Funktional betrachtet ist der Algorithmus äquivalent zu dem aus der letzten Vorlesung (OPL). Er hat jedoch den Vorteil, daß

| i \ j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | | | | | | |
| 2 | | 0 | | | | | |
| 3 | | | 0 | | | | |
| 4 | | | | 0 | | | |
| 5 | | | | | 0 | | |
| 6 | | | | | | 0 | |

Abbildung 4.21: P[i,j]

seine Kontrollstruktur einfacher ist: drei **for**-Schleifen ineinander geschachtelt. Das bedeutet, daß wahrscheinlich so ein tatsächlich implementiertes Programm etwas schneller laufen wird als das rekursive Gegenstück. Außerdem ist die Laufzeitanalyse einfacher. Von den drei Schleifen sieht man sofort die $O(n^3)$ Laufzeit. Hier folgt das Bottom-Up Programm.

**procedure** OPL – BOTTOM – UP

```
for i = 1 to n do P[i, i − 1] = 0
for l = 0 to n − 1 do
   for i = 1 to n − l do
      j = i + l
      min = +∞
      for m = i to j do
            s = 1 + (w_{i,m−1} / w_{i,j}) · OPL(i, m − 1) + (w_{m+1,j} / w_{i,j}) · OPL(m + 1, j)
            if s < min then min = s
      P[i, j] = min
```

Durch Anwendung der sogenannten Vierecksungleichung kann dieses (und auch das rekursive) Programm noch verbessert werden, so daß $O(n^2)$ Laufzeit erzielt wird. Die Details dazu können Sie in Mehlhorn's Buch finden (Seiten 151–153).

## 4.8 Dynamisches Programmieren

Der eben beschriebene Ansatz, zuerst "kleine" Probleme zu lösen und aus den Lösungen der kleineren Probleme die Lösungen immer größerer Probleme aufzubauen, wird *Dynamisches Programmieren* genannt. Typischerweise muß eine Funktion über einen ganzzahligen endlichen Wertebereich berechnet werden — in unserem Fall war der Werebereich $\{1, \ldots, n\}^2$ — und die natürliche Definition dieser Funktion ist hoch rekursiv.

Der natürliche Ansatz zur Berechnung so einer Funktion besteht darin, die rekursive Definition einfach in ein rekursives Programm zu übersetzen. Dieser naive Ansatz kann zu sehr schlechten Laufzeiten führen, wenn für einzelne Argumentwerte die Funktion wiederholt berechnet wird. Diese wiederholte Berechnung kann aber leicht verhindert werden. Da es nur relative wenige mögliche Argumentwerte gibt, kann man sich für diese die schon berechneten Funktionswerte merken. Dieser Trick genügt normalerweise schon, um eine respektable Laufzeit zu erreichen.

Eine in der Praxis noch bessere Laufzeit kann man erzielen, wenn man auf die Rekursion verzichtet und sich genau überlegt, in welcher Reihenfolge die Funktion für verschiedene Argumente berechnet werden muß. Typischerweise ergibt dieser Bottom-Up Ansatz ein Programm, das nur aus ein paar geschachtelten **for**-Schleifen besteht.

# Kapitel 5

# Warteschlangen (Priority Queues)

## 5.1 Einführung

In diesem Kapitel befassen wir uns mit dem Begriff der **Warteschlange** (engl. Priority Queue). Eine Warteschlange dient dazu, eine Menge $S$ von Elemente zu verwalten. Dabei wird jedes Element mit einem Wert assoziiert, den wir einen Schlüssel nennen. Ein Element von $S$ bezeichnen wir, wie üblich als ein Stück $x$ (engl. Item) mit $x.key$ als Schlüsselkomponente. Die Schlüssel müssen hierbei einee Ordnung unterliegen.
Wir nehmen der Einfachheit halber die Schlüssel aus der Menge der rellen Zahlen. Man könnte aber auch jede beliebig andere Menge verwenden, auf der sich eine Ordnung definieren läßt.
Auf einer Warteschlange wünschen und definieren wir folgende Operationen:

- $H$ = MAKEQUEUE($S$) : erzeugt eine Warteschlange $H$ aus den Elemente von $S$
- INSERT($x$,$H$) : fügt das Stück $x$ in die Warteschlang $H$ ein
- MIN($H$) : liefert das Stück $x$ mit dem kleinsten Schlüssel in $H$
- EXTRACT-MIN($H$) : liefert das Stück $x$ mit dem kleinsten Schlüssel und löscht es gleichzeitig aus der Warteschlange $H$
- DECREASE-KEY($H$,$x$,$k$) : weist dem Stück $x$ in $H$ den neuen Schlüsselwert $k$ zu, von dem wir annehmen, daß er nicht größer als der bisherige Schlüsselwert ist
- DELETE($x$,$H$) : löscht das Stück $x$ aus $H$
- UNION($H_1$,$H_2$) : liefert eine neu kreierte Warteschlange, die alle Stücke von $H_1$ und $H_2$ beinhaltet. Die Warteschlangen $H_1$ und $H_2$ werden durch diese Operation "zerstört"

Anmerkung:
Die Operationen DECREASE-KEY und DELETE erwarten in ihrer Eingabe nicht etwa den Schlüssel des Stückes $x$, sondern das Stück $x$ selbst bzw. einen Pointer darauf. Von all diesen Operationen scheint uns UNION auf den ersten Blick am schwierigsten zu implementieren.
Ein Anwendungsbeispiel:
Eine Applikation für Warteschlangen ist ein Ereignis-gesteuerter-Simulator (engl. event-driven-simulator). Die Stücke in der Schlange bezeichnen Ereignisse, die zu simulieren sind, wobei jedes Ereignis mit dem Zeitpunkt seines Auftretens assoziiert wird, was den Schlüssel darstellt. Die Ereignisse müssen in Reihenfolge ihres Auftrittzeitpunktes simuliert werden. Die Simulation eines Ereignisses kann andere Ereignisse, die in der Zukunft auftreten, verursachen.
Die Operation EXTRACT-MIN liefert uns hierbei bei jedem Schritt das nächste zu simulierende Ereignis. Durch INSERT werden neu produzierte Ereignisse in die Warteschlange aufgenommen.
Eine Warteschlange mit ihren Operationen können wir auf verschiedenen Wegen implementieren. Hierzu betrachten wir folgende Datenstrukturen:

- **Binary Heaps**
- **Binominalschlangen**
- **Fibbonacci-Heaps**

## 5.2 Binary Heaps

Ein Binary Heap ist ein perfekter binärer Baum (engl. perfect binary tree) mit Heapeigenschaft, der implizit in einem Feld (engl. array) abgespeichert werden kann, indem der Inhalt der Knoten niveauweise von links nach rechts im Feld abgelegt wird. Umgekehrt kann man aber auch aus einem beliebigen Feld einen impliziten Binary Heap erzeugen.
Betrachten wir dazu die beiden folgenden Definitionen:

1. Sei T ein Baum mit Wurzel $root(T)$.
   T hat die **Heapeigenschaft**, wenn für jeden Knoten $v \neq root(T)$ gilt: $key(parent(v)) < key(v)$
   Informell: "Wenn man von einem Blatt zur Wurzel geht, sieht man immer kleinere Schlüssel"

2. Ein binärer Baum heißt **perfekter binärer Baum**, wenn
   (a) sich alle Blätter auf zwei benachbarten Niveaus befinden,
   (b) im höchsten Niveau alle Blätter möglichst weit links sind
   (c) und es höchstens einen Knoten mit genau einem Kind gibt.

Beispiel: Abbildung 5.1 zeigt uns die Darstellung eines Binary Heap als perfekten binären Baum. Die Zahl in jedem Knoten des Baumes ist der Schlüssel von diesem Knoten. Die Zahl neben einem Knoten ist der korrespondierende Index im Feld. Rechts ist das Niveau der Knoten in der entsprechenden Zeile angegeben.
Abbildung 5.2 zeigt uns denselben Baum implizit in einem Feld $A[1..n]$ abgespeichert, wobei $n = 12$ die Anzahl der Knoten ist.

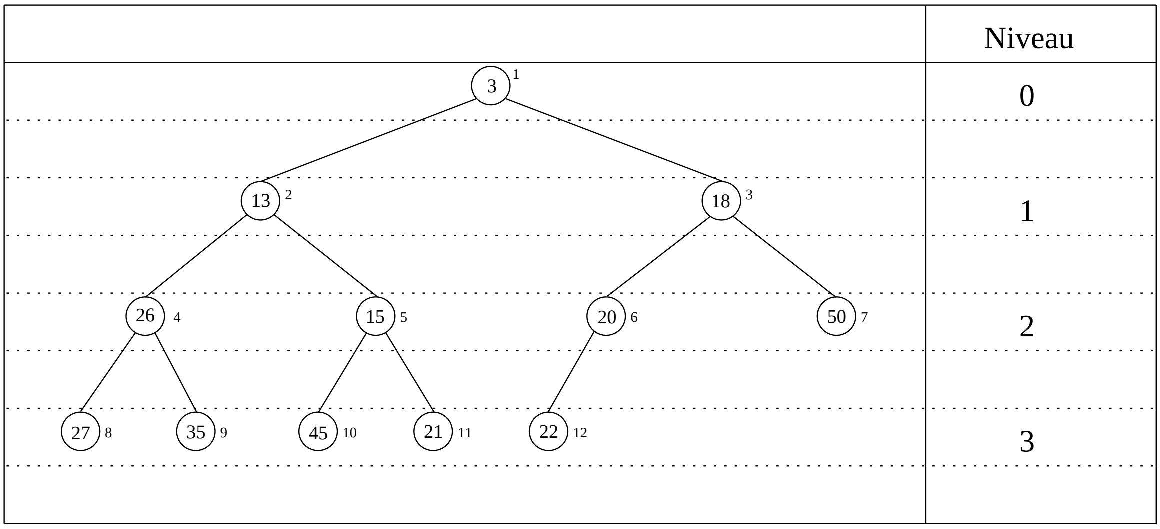



Abbildung 5.1: Ein perfekter binärer Baum mit Heapeigenschaft

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 13 | 18 | 26 | 15 | 20 | 50 | 27 | 35 | 45 | 21 | 22 | A[1..12] |

Abbildung 5.2: Implizite Darstellung des Baumes als Feld

Die Wurzel des Baumes ist $A[1]$. Sei der Index $i$ eines Knoten gegeben, dann können die Indizes der Eltern des Knoten $parent(i)$, seines linkes Kindes $lc(i)$ und seines rechten Kindes $rc(i)$ leicht berechnet werden:

- *parent*$(i)$: **return** $\lfloor i/2 \rfloor$
- $lc(i)$: **return** $2 \cdot i$
- $rc(i)$: **return** $2 \cdot i + 1$

Möchten wir wissen, ob es sich bei dem Index $i$ um ein Blatt handelt, so können wir dies mittels BLATT$(i)$ berechnen:

- BLATT$(i)$
  **if** $(i > n/2)$ **then return** *i ist Blatt* **else return** *i ist kein Blatt*

Diese einfachen Funktionen begegnen uns, wenn wir uns mit der Frage befassen, wie man jetzt konkret ein Feld $A[1..n]$ in einen impliziten Binary Heap "verwandelt".
Die Antwort auf diese Frage gibt uns folgender Algorithmus:

**procedure** HEAPIFY$(A, n)$
(0) **for** $j = \lfloor n/2 \rfloor$ **downto** 1 **do** SENKE$(j, n)$

**procedure** SENKE$(i, n)$
(1) **if** $(i > n/2)$ **then return** (∗ *bedeutet i istBlatt* ∗)
(2) **if** $(2 \cdot i = n)$ **then** { **if** $(A[n].key < A[i].key)$ **then** SWAP$(A[i], A[n])$ }
(∗ *vertausche die Feldinhalte* ∗)
(3) **else** { **if** $(A[2 \cdot i].key < A[2 \cdot i + 1].key)$ **then** $c = 2 \cdot i$
**else** $c = 2 \cdot i + 1$
(4) **if** $(A[c].key < A[i].key)$ **then** { SWAP$(A[c], A[i])$
(∗ *vertausche die Feldinhalte* ∗)
SENKE$(c, i)$
(∗ *rekursiver Aufruf der Prozedur* ∗)
}
}

<u>Erläuterung:</u>
Der HEAPIFY-Algorithmus arbeitet in einer typischen Bottom-Up Manier, d.h. der Baum wird von den Blättern zur Wurzel hin aufgebaut. Die SENKE-Prozedur dient dabei zur Aufrechterhaltung der Heapeigenschaft. In Zeile (0) verwenden wir implizit die *parent*-Funktion, in (1) die BLATT-Funktion, in (2) die *lc*-Funktion und in (3) sowohl die *lc*- als auch die *rc*-Funktion.
Analysieren wir die <u>Laufzeit</u> von HEAPIFY:
Die benötigte Zeit bzw. die Kosten für die SENKE-Prozedur entnehmen wir aus der folgenden Tabelle in Abbildung 5.3.
Daraus ergeben sich im schlechtesten Fall Gesamtkosten für HEAPIFY von:

$$\sum_{h=0}^{H-1} 2^h \cdot (H - h).$$

Diese Summe schreiben wir um, indem wir $i = H - h$ setzen, und schätzen sie ab wobei wir ausnutzen, daß für $n$, Anzahl der Knoten im Heap, gilt $2^H \leq n$:

$$\sum_{i=1}^{H} 2^{H-i} \cdot i \leq \sum_{i=1}^{H} \frac{n}{2^i} \cdot i \leq n \cdot \underbrace{\sum_{i=1}^{\infty} i \cdot (\frac{1}{2})^i}_{=2} = 2 \cdot n.$$

Es folgt also, daß HEAPIFY eine Laufzeit von $O(n)$ hat. Insbesondere benötigt die SENKE-Prozedur allein betrachtet eine worst-case Laufzeit von $O(\log\ n)$, und amortisiert eine Laufzeit von $O(1)$.

Wenden wir uns jetzt wieder unserem eigentlichen Ziel zu, nämlich der Implementierung der einzelnen Operationen einer Warteschlange unter Verwendung der Datenstruktur Binary Heap, und betrachten das Laufzeitverhalten der einzelnen Operationen.

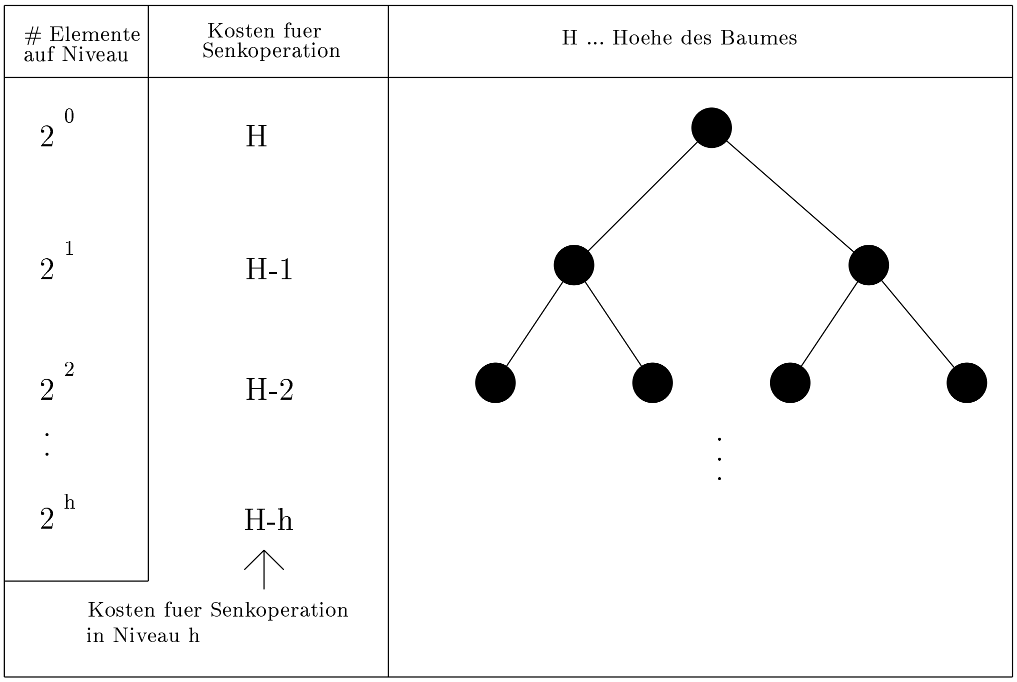



Abbildung 5.3: Kosten für $SENKE$-Operation

<u>Makequeue</u>:
Nehmen wir an, die Stücke aus der Menge $S$ mit $|S| = n$, für die eine Warteschlange erzeugt werden soll, sind in einem Feld $A[1..n]$ abgelegt. Dann verwenden wir für diese Operation den Heapify-Algorithmus, der, wie gesehen, eine Laufzeit von $O(n)$ hat.
<u>Insert</u>:
Um ein Stück $x$ in eine Warteschlange einzufügen, hängen wir dieses als Blatt in unseren Heap an die nächste freie Position und lassen es bis zur richtigen Position aufsteigen.

**procedure** Insert$(x, H)$
*Heapgröße von* $|H| = n$ *auf* $n + 1$ *erweitern*
$i = n + 1$
**while** $i > 1$ **and** $H[parent(i)].key > x.key$ **do**
  $H[i] = H[parent(i)]$
  $i = parent(i)$
$H[i] = x$

Da die Höhe des Baumes logarithmisch ist, benötigt diese Operation $O(\log\ n)$ Zeit. Die Anwendung der Insert-Prozedur wird in Abbildung 5.4 am Beispiel verdeutlicht.

<u>Min</u>:
Diese Operation ist sehr einfach zu realisieren, da das Stück $x$ mit dem kleinsten Schlüssel aufgrund der Heapeigenschaft an der Wurzel des Baumes zu finden ist.

**procedure** Min$(H)$
**if** *Heap ist leer* **then error**
**return** $H[1]$

Dies benötigt konstante Zeit ($O(1)$).
<u>Min-Extract</u>:
Merke dir das Stück $x$ an der Wurzel (dies ist das mit dem kleinsten Schlüssel), ersetze Wurzel durch das letzte im Heap gespeicherte Blatt und lasse den Schlüssel runtersinken. Liefere dann $x$ zurück.

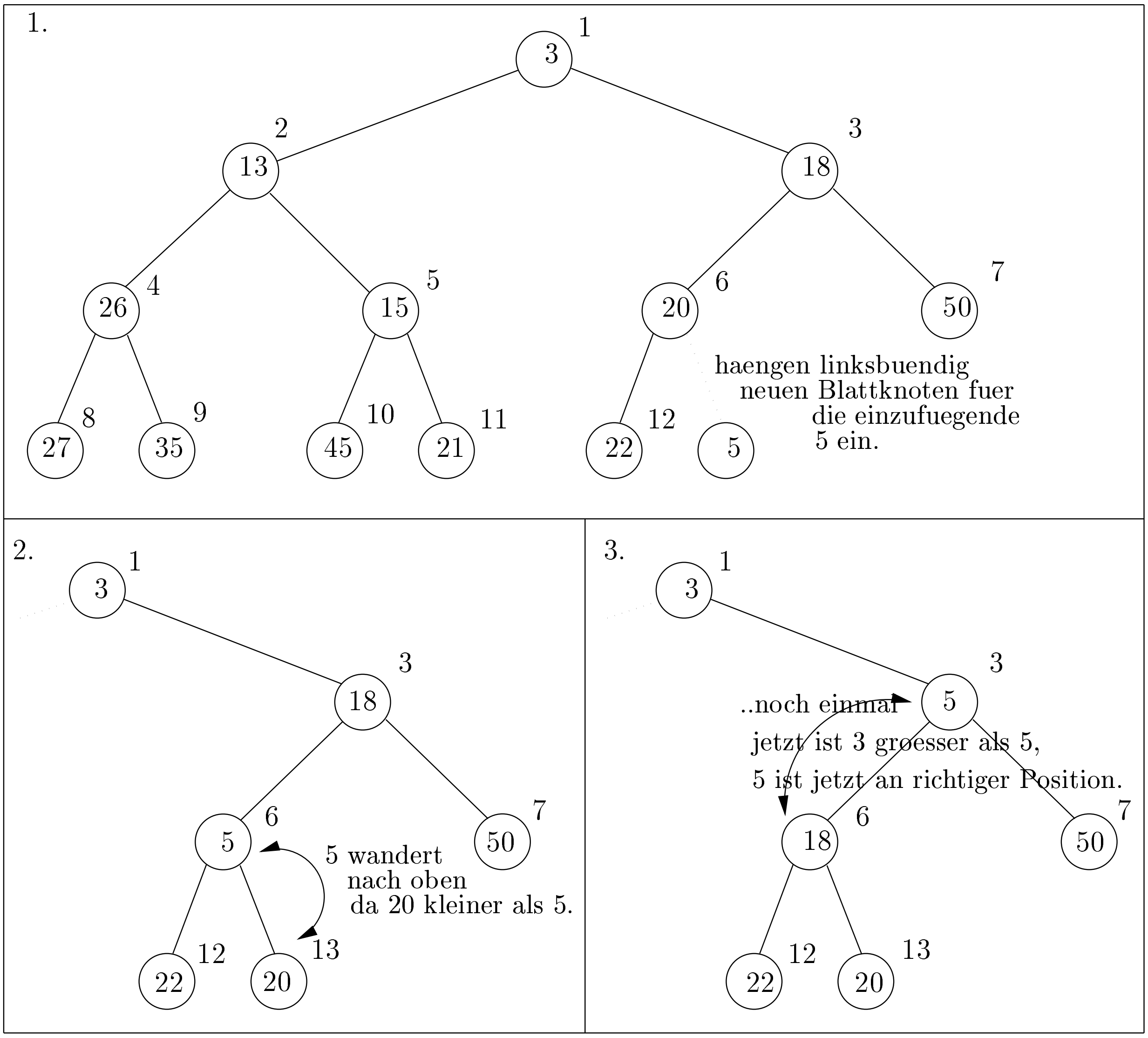



Abbildung 5.4: Einfügen eines Knoten mit Schlüssel (5) in den Heap

**procedure** MIN-EXTRACT($H$)
(1) **if** *Heap ist leer* **then error**
(2) $min = H[1]$
(3) $H[1] = H[n]$ *(* unter der Annahme* $|H| = n$ **)*
(4) *Heapgröße von* $|H| = n$ *auf* $n-1$ *verringern*
(5) SENKE$(1, n)$
(6) **return** $min$

Die Laufzeit dieser Prozedur wird durch den Aufruf von SENKE bestimmt, ist also $O(\log n)$.
<u>DECREASE-KEY</u>:
Wir vermindern den Schlüssel von Stück $x$ um $k$. Dabei nehmen wir an, daß $k$ nicht größer ist als der bisherige Schlüssel. Nun lassen wir das verminderte $x$ bis zur richtigen Position im Heap aufsteigen.

**procedure** DECREASE-KEY$(H, x, k)$
(1) *Sei* $H[i]$ *der Knoten im Heap der Stück* $x$ *enthält*
*(* diese Information erhalten wir, da wir das gesamte Stück* $x$ *der Prozedur übergeben *)*
(2) $x.key = x.key - k$

(3) $y = x$ (4) **while** $i > 1$ **and** $H[parent(i)] > x.key$ **do**
(5) $\quad H[i] = H[parent(i)]$
(6) $\quad i = parent(i)$
(7) $H[i] = y$

Die Laufzeit beträgt hierbei $O(\log n)$, da man im worst Case den Schlüssel von einer Blattposition bis zur Wurzel aufsteigen lassen muß.

<u>DELETE</u>:
Wollen wir ein Stück $x$ aus dem Heap löschen, so nehmen wir das Blatt an der Position $H[n]$ und weisen dieses Blatt dem Knoten, in dem sich der Schlüssel von $x$ befindet, zu. Um die Heapeigenschaft aufrecht zu halten, müssen wir diesen "neuen" Knoten jetzt aufsteigen lassen, falls der Schlüssel des Elternknoten größer ist als der eigene, bzw. absinken lassen, falls der Schüssel eines Kindes kleiner ist als der eigene. Den Pseudo-Code für diese Operation konkret zu schreiben, überlassen wir dem Leser als Übung. Dies dürfte aber keine Schwierigkeiten bereiten.
Die Laufzeit ist leicht ersichtlich wiederum logarithmisch ($O(\log\ n)$).
Die Anwendung von DELETE wird in Abbildung 5.5 am Beispiel verdeutlicht.

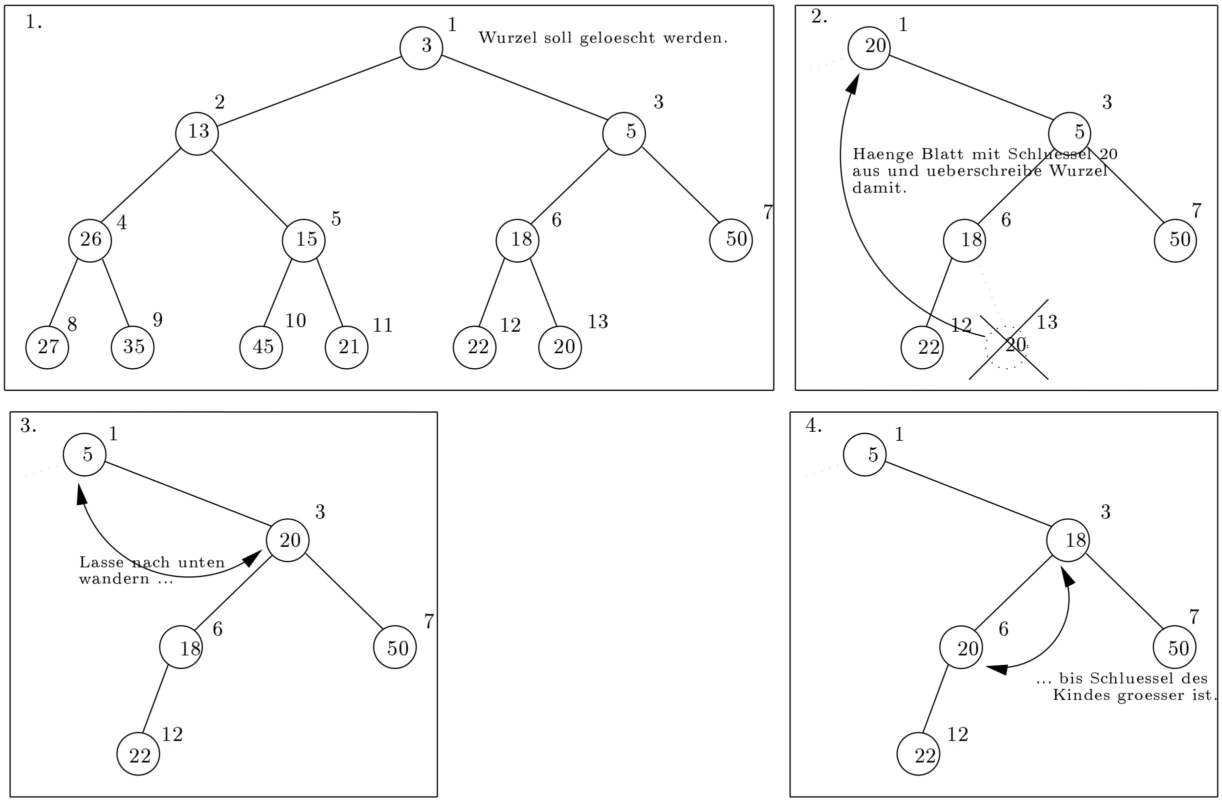



Abbildung 5.5: Löschen des Knoten mit Schlüssel (3) aus dem Heap

<u>UNION</u>:
Erzeuge ein Feld $H$ der Größe $|H_1 + H_2|$, lege $H_1$ und $H_2$ in diesem Feld ab und wende den HEAPIFY-Algorithmus an.

**procedure** UNION($H_1, H_2$)
(1) reserviere Speicherplatz der Größe $H[1..\text{LENGTH}(H_1)+\text{LENGTH}(H_2)]$
(2) kopiere $H_1$ und $H_2$ nach $H$
(3) HEAPIFY($H$,LENGTH($H_1$)+LENGTH($H_2$))

Die Laufzeit der Zeilen (1), (2), (3) ist jeweils linear, und somit kann UNION in $O(n)$ Zeit durchgeführt werden.
<u>Zusammenfassend</u> ergeben sich folgende Laufzeiten:

| Operationstyp | Laufzeit |
|---|---|
| MAKEQUEUE | $O(n)$ |
| INSERT | $\log n$ |
| MIN | $O(1)$ |
| EXTRACT-MIN | $\log n$ |
| DECREASE-MIN | $\log n$ |
| DELETE | $\log n$ |
| UNION | $O(n)$ |

**Heapsort**

Als eine weitere Anwendung von Binary Heap betrachte folgenden Sortieralgorithmus:

**procedure** HEAP-SORT$(A, n)$
(1) HEAPIFY$(A, n)$
(2) **for** $k = n$ **downto** 1 **do**
(3) (gib $A[1]$ aus) (4) SWAP$(A[k], A[1])$
(5) SENKE$(1, k-1)$

Nach Anwendung dieses Algorithmus ist dann das Feld $A[1..n]$ absteigen sortiert.
Dieser Sortieralgorithmus ist recht einfach zu implementieren und hat einige schöne Eigenschaften: Er benötigt außer dem Feld $A[]$ keinen wesentlichen zusätzlichen Speicher und er hat eine Laufzeit von $O(n \cdot \log n)$ im schlechtesten Fall.

## 5.3 Binominalschlangen

Eine *Binominalschlange* (engl. *Binominal Heap*) ist eine über die Wurzeln verkettete Liste von *Binominalbäumen*. Ein Binomialbaum $B_i$ vom Grad $i$ wird induktiv folgendermaßen definiert: $B_0$ ist ein nur einen Knoten umfassender Baum; für $i > 0$ wird ein Baum $B_i$ gebildet, indem an die Wurzel eines Baums $B_{i-1}$ ein weiterer Baum $B_{i-1}$ als zusätzlicher Unterbaum angehängt wird (siehe Abbildung).

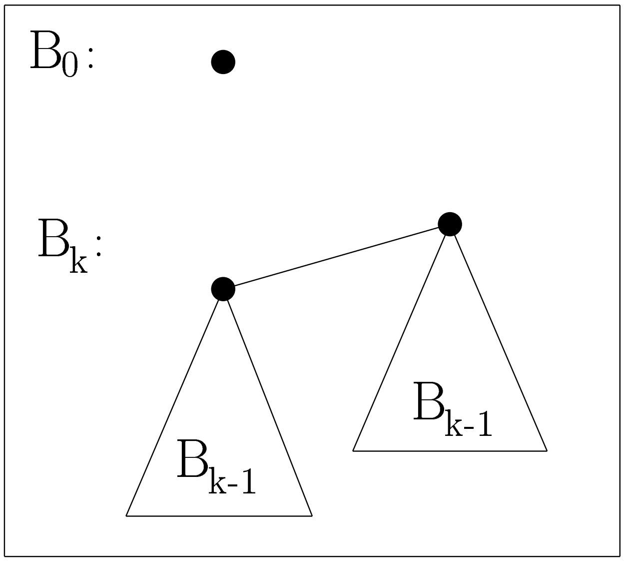



Abbildung 5.6: Induktive Definition eines Binominalbaumes

Eine andere Möglichkeit zur Betrachtung von Binominalbäumen zeigt Abbildung 5.7.

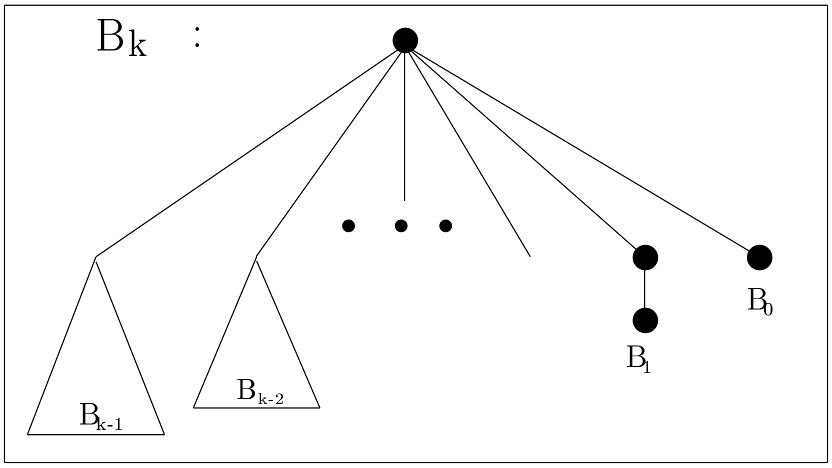



Abbildung 5.7: Binominalbaum $B_k$

Woher der Name “Binomiminalbaum” stammt, zeigt uns Abbildung 5.8. Die Anzahl der Knoten des Baumes auf einer Tiefenstufe ist mit den Binominalkoeffizienten identisch.

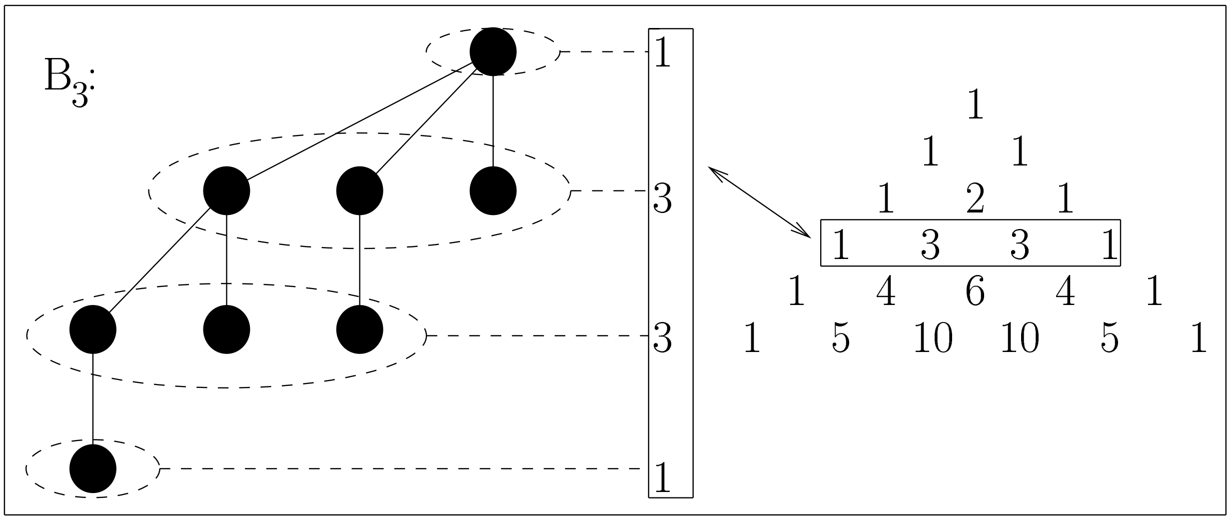



Abbildung 5.8: Als Beispiel ein Binominalbaum $B_3$

Induktiv kann man leicht die folgenden einfachen Eigenschaften zeigen.

**Lemma 7** *Sei $B_i$ ein Binominalbaum. Dann gelten folgende Eigenschaften:*

1. *$B_i$ hat $2^i$ Knoten.*
2. *Die Wurzel von $B_i$ hat $i$ Kinder (und alle anderen Knoten in $B_i$ haben $< i$ Kinder).*
3. *$B_i$ hat die Tiefe $i$.*

**Definition 9** *Eine* Binominalschlange *für eine Menge $S$ von Stücken ist eine Familie von Binominalbäumen (mit einem Knoten pro Element von $S$), die die folgenden Eigenschaften besitzt:*

1. *Jeder Baum besitzt die Heapeigenschaft.*

2. *Keine zwei Bäume haben den gleichen Grad.*

Für eine beliebige Menge $S$ mit $|S| = n$ kann man immer eine solche Famile von Binominalbäumen finden, denn wir können jede Zahl $n$ als Summe von Zweierpotenzen darstellen (Binärdarstellung). Insbesondere ist deshalb die Anzahl der Bäume in der Binominalschlange $O(\log |S|)$.

Beispiel: $|S| = 23 = \underbrace{2^4}_{B_4} + \underbrace{2^2}_{B_2} + \underbrace{2^1}_{B_1} + \underbrace{2^0}_{B_0}$

Vorlesung vom 18.12.1995
Igor Pouchkarev, Jörg Brück, Heiko Schütz

Betrachten wir nun die Implementierung der einzelnen Operationen einer Warteschlange und deren Laufzeitverhalten unter Verwendung von Binomialschlangen.

MAKEQUEUE
Zeit zur Initialisierung: $O(1)$

MIN
Finde die kleinste aller Wurzeln in der Binomial-Schlange. Da die Anzahl aller Bäume in der Binomialschlange $O(\log n)$ ist, beträgt die Laufzeit $O(\log n)$.

UNION
Beruht auf der Möglichkeit, zwei heap-geordnete Binomialbäume $B_k, B'_k$ der gleichen Größe in $O(1)$ Zeit zu einem Binomialbaum $B_{k+1}$ zu verknüpfen: der Baum mit der größeren Wurzel wird als linkestes Kind an die Wurzel des anderen Baumes angehängt. Zwei Binomialschlangen Größe werden vereinigt, indem man eine Art "Addition" der beiden Binomialschlangen durchführt (analog zur Addition von Binärzahlen): Beginnend mit $i = 0$ und $i$ hinaufzählend werden, falls es mehr als einen $B_i$ gibt, zwei solche $B_i$'s zu einem $B_{i+1}$ verknüpft, wie oben beschrieben.

**Beispiel:**

$$
\begin{array}{llllllllllll}
|S_1| & = & 23 & = & & 2^4 & & + & 2^2 & + \; 2^1 & + & 2^0 \\
 & & & & & B_4 & & & B_2 & \quad B_1 & & B_0 \\
 & & & & & & & & & & & \\
|S_2| & = & 19 & = & & 2^4 & & & & + \; 2^1 & + & 2^0 \\
 & & & & & B_4 & & & & \quad B_1 & & B_0 \\
 & & & & & & & & & & & \\
S_1 \vee S_2 & = & & & & B_4 & & & B_2 & \quad B_1 & & B_0 \\
 & & & & & B_4 & & & & \quad B_1 & & B_0 \\
\hline
 & & & & B_5 & & B_3 & & & \quad B_1 & &
\end{array}
$$

Dies braucht die Gesamtlaufzeit $O(\log n)$, da die Binomialschlange höchstens $\log n$ Bäume hat.

INSERT
Einfügen eines Schlüssels $x$ in eine Binomialschlange $H$ erfolgt durch Produzieren einer trivialen Binomialschlange $H_x$ für $x$ und die darauf folgende Vereinigung UNION$(H_x, H)$. Diese Operation erfordert ebenfalls in Zeit $O(\log n)$.

EXTRACTMIN
Entferne den Baum mit der kleinsten Wurzel aus der Binomial-Schlange $H$ und vereinige seine Teilbäume mit $H$. Da die Laufzeit von UNION $O(\log n)$ ist, ist die Gesamtlaufzeit auch hier $O(\log n)$.

DecreaseKey Der Zugriff auf das Element erfolgt direkt. Vermindere den Schlüssel des Elements um den entsprechenden Betrag. Jetzt läßt man das Element noch nach oben Richtung Wurzel wandern. Man vergleicht, ob das Element kleiner als der Vaterknoten ist. Trifft dies zu, so vertauscht man sie. Ist der Vaterknoten größer, so ist der Endzustand erreicht. Der Schlüssel wandert also höchstens bis zur Wurzel. Da der Baum logarithmische Tiefe besitzt, ist die Gesamtlaufzeit $O(\log n)$.

Delete
Vermindere den zu löschenden Knoten mit DecreaseKey auf $-\infty$ . Wende dann Extractmin an. Auch hier beträgt also die Laufzeit $O(\log n)$.
Um die Operationen in der angegebenen Zeit durchführen zu können benötigt man in jedem Knoten einen Pointer auf das linkeste Kind, auf den Bruder und auf den Vater. Desweiteren muß man in jedem Knoten den Schlüssel und den Grad des jeweiligen Unterbaums (um bei Vereinigung gleich große Bäume zu erkennen) speichern.

## 5.4 Fibonacci Heaps

In diesem Abschnitt betrachten wir eine dritte Möglichkeit, Warteschlangen zu implementieren. Die beiden vorherigen Datenstrukturen hatten eine starre Struktur (nur abhängig von ihrer Größe). Fibonacci-Queues sind dagegen typische 'lazy' Strukturen: Pro Zugriff wird möglichst wenig getan, zeitaufwendige Umstrukturierungen werden möglichst spät vorgenommen. Ihre Kosten werden amortisiert abgerechnet. Im Einzelfall kann ein Zugriff teuer sein.

Laufzeitvergleich:

| | binärer Heap | Binomial-Schlange | Fibonacci-Schlange (amortisierte Laufzeit) |
|---|---|---|---|
| Makequeue | $O(n)$ | $O(1)$ | $O(1)$ |
| Insert($x$,$H$) | $\log n$ | $\log n$ | $O(1)$ |
| Min($H$) | $O(1)$ | $\log n$ | $O(1)$ |
| Extract-Min($H$) | $\log n$ | $\log n$ | $\log n$ |
| DecreaseKey($H$,$x$,$k$) | $\log n$ | $\log n$ | $O(1)$ |
| Delete($x$,$H$) | $\log n$ | $\log n$ | $\log n$ |
| Union($H_1$,$H_2$) | $O(n)$ | $\log n$ | $O(1)$ |

Ein Fibonacci-Heap ist eine endliche Menge heapgeordneter Bäume. Die Wurzeln der Bäume werden in einer doppelt verketteten, zirkulären Liste gespeichert, ebenso die Kinder jedes Knotens. Doppelt verkettete Listen haben zwei Vorteile:

1. Man kann einen Knoten in $O(1)$ Zeit löschen.
2. Zwei solcher Listen kann man in $O(1)$ Zeit konkatenieren.

Die Anzahl Grad($v$) der Kinder eines Knotens $v$ wird in $v$ gespeichert. *In allen Operationen wird Grad($v$) korrekt aufrecht erhalten.* Außerdem brauchen wir bei jedem Knoten Platz für eine boolesche Markierung (die für DecreaseKey benötigt wird). Jeder Knoten hat einen Zeiger auf seinen Vater (NIL für die Wurzeln) und einen auf die Liste seiner Kinder. Zudem besitzt ein Fibonacci Heap einen Pointer auf das kleinste Element.
Abbildung 5.9 zeigt einen Fibonacci-Heap.
Wie werden die einzelnen Operationen einer Warteschlange unter Verwendung von Fibonacci-Heaps realisiert?

Makequeue
Ein leerer Fibonacci-Heap kann in konstanter Zeit erzeugt werden.

Union
Um eine Vereinigung von zwei Fibonacci-Heaps durchzuführen, hänge die beiden Wurzellisten zusammen. Da die Listen doppelt verkettet sind, kann man dies in konstanter Zeit erreichen. Anschließend muß eventuell noch der Pointer auf das minimale Element umgehängt werden, was ebenfalls in konstanter Zeit durchführbar ist, da

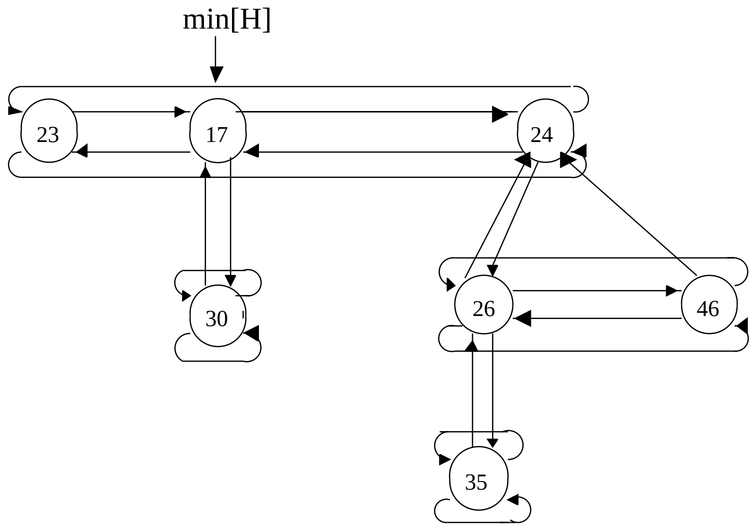



Abbildung 5.9: Beispiel eines Fibonacci Heaps

für jeden der beiden Fibonacci-Heaps das minimale Element bekannt ist. Vereinigung braucht also konstante Zeit.

<u>INSERT</u>
Einfügen eines Elementes $x$ in einen Fibonacci-Heap $H$ erfolgt (wie bei Binomialschalangen) durch UNION($H_x, H$). Dies braucht ebenfalls in konstante Zeit.

<u>MIN</u>
Da wir einen Pointer auf das minimales Element haben benötigen wir für diese Operation nur eine Laufzeit von $O(1)$.

<u>EXTRACTMIN</u>
Entferne den Baum mit der kleinsten Wurzel aus der Wurzelliste und hänge dafür seine Kinder in die Wurzelliste ein. Konsolidiere die Wurzelliste, d.h. solange es in der Wurzelliste noch zwei Bäume desselben Grades gibt, verschmelze sie. Dies erfolgt analog zur Vereinigung von Binomialschlangen. Es wird ein "Akkumulator" $A$, d.h. ein Array von Fibonacci-Heaps benutzt (anfangs sind alle Fibonacci-Heaps leer), in den jeder der Teilbäume entsprechend seinem Wurzelrang eingefügt wird. Dabei wird er mit einem eventuell dort schon vorhandenen Baum gleichen Grades zusammengelinkt (in $O(1)$ Zeit), der so entstandene Baum wird entsprechend eine Position weiter eingefügt. Am Ende werden die entstandenen Bäume wieder durch eine zirkuläre, doppelt verkette Liste zusammengehängt, und der Minimumzeiger wird auf die kleinste Wurzel gesetzt.

### 5.4.1 Amortisierte Laufzeitanalyse (über Bankkonten)

Zum Zweck der amortisierten Analyse nehmen wir an, daß die Durchführung von $O(1)$ Rechenoperation 1 DM kosten. Geld ist entweder irgendwo hinterlegt (z.B. bei Baumknoten) oder muß von der durchzuführenden Operation eingebracht werden. Dieses eingebrachte Geld kann auch zum Teil wieder irgendwo hinterlegt werden. Wenn man mit einer anfangs leeren Struktur ohne irgendein hinterlegtes Geld beginnt, dann sind die Gesamtkosten einer Folge von Opertionen von oben beschränkt durch das Geld das insgesamt von den Operationen eingebracht wurde.
Nehmen wir als Invariante an, daß bei jedem Element der Wurzelliste 1 DM hinterlegt ist. Man kann sich leicht davon überzeugen, daß diese Invariante von den vorher beschriebenen Operationen aufrecht erhalten werden kann, wenn sie jeweils selbst 1 DM mitbringen, um für den konstanten Aufwand der eigenen Durchführung zu bezahlen.

Betrachten wir nun die Operation EXTRACTMIN. Unsere Invariante könnte verletzt werden, wenn die Kinder des minimalen Elementes in die Wurzelliste eingehängt werden. Um sie aufrechtzuerhalten, muß die Operation für jedes dieser Kinder 1 DM mitbringen und sie auf dem entsprechend neuen Knoten der Wurzelliste hinterlegen. (Dies alles geschieht natürlich nur in unserer Vorstellung zwecks Analyse!!!) Wir werden später sehen, daß die Anzahl dieser Kinder höchstens $O(\log n)$ ist. Also muß die Operation $O(\log n)$ DM mitbringen.
Nun zum Konsolidieren der Wurzelliste. Das Verschmelzen von zwei Bäumen gleichen Ranges wird durchdie 1 DM des Baumes mit der größeren Wurzel bezahlt. Die 1 DM der Wurzel des anderen Baumes bleibt dort, um die Invariante aufrecht zu halten. Das Konsolidieren benutzt also die gesparten DM. Schließlich müssen noch die entstandenen Bäume in eine neue Wurzelliste vereinigt werden. Wir werden sehen, daß die Anzahl der Bäume in der neuen Wurzelliste wieder $O(\log n)$ ist. Ihrer Schaffung brauch also $O(\log n)$ Operationen, die durch weiterer $O(\log n)$ DM bezahlt werden, die von der EXTRACTMIN Operation mitgebracht werden müssen.
Modulo der Behauptungen, daß zu Beginn das minimale Element nur $O(\log n)$ Kinder hat und daß am Ende die Wurzelliste aus $O(\log n)$ Bäumen besteht, haben wir also gezeigt, däs EXTRACTMIN $O(\log n)$ DM mitbringen muß, um einerseits für die Rechenoperationen seiner Durchführung zu bezahlen und andererseits die genannte Invariante aufrechtzuerhalten. Das bedeutet, dass die amortisierte Laufzeit für die ganze Operation $O(\log n)$ beträgt.

DECREASEKEY
Der Knoten $x$, dessen Priorität herabgesetzt wird, muß schon bekannt sein (typischerweise durch einen Zeiger). Der Knoten wird von seinem Vater abgetrennt, der Vater wird markiert, der Knoten selbst mit verringertem Schlüssel in die Wurzelliste eingefügt. Durch Vergleich mit dem bisherigen Minimum wird in $O(1)$ Zeit der Zeiger auf das Minimum des Fibonacci-Heaps gefunden.
Das Problem ist, daß nach mehrfachen DECREASEKEY-Operationen dieses Verfahren den Baum zu einer Liste entarten lassen kann. Insbesonders würden dann die oben in der Analysie von EXTRACTMIN gemachten Behauptung nicht stimmen. Um das zu verhindern wird ein Vater, der markiert ist, also schon einmal ein Kind abgegeben hat, selbst ebenfalls abgetrennt. Dies wird gegebenenfalls bis zur Wurzel wiederholt ("cascading cut"). Beim Einfügen in die Wurzelliste werden die Marken gelöscht.

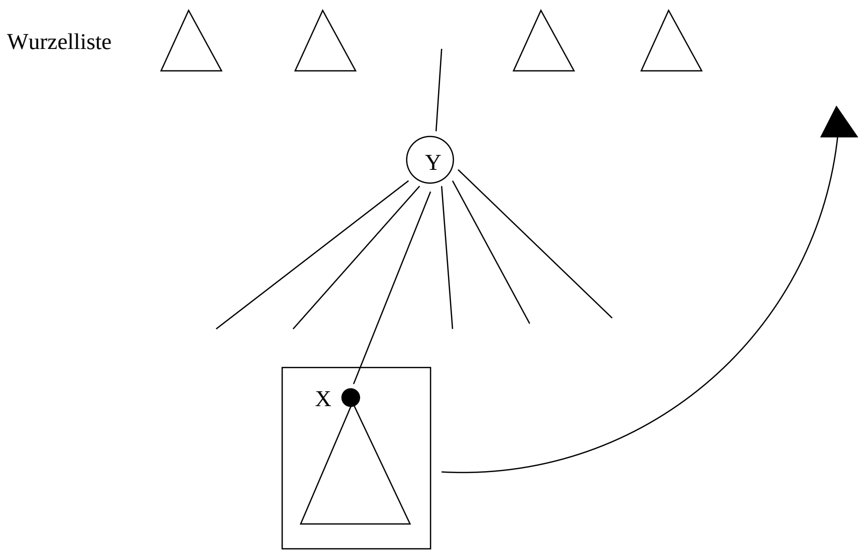



Abbildung 5.10: DecreaseKey(x,H)

DECREASEKEY($H, x, k$)
  **if** $k > \text{key}[x]$ **then**
    error "der aktuelle Schlüssel ist zu klein"
  $\text{key}[x] \leftarrow k$

```
    Cascading-cut(x)
    if key[x] < key[min[H]] then min[H] ← x

Cascading-cut(x)
   y ← Vater(x)
   if y = NIL then return
   Cut(x)
   if y markiert then Cascading-cut(y)
                 else if Vater(y) ≠ NIL then markiere(y)


Cut(x)
   y ← Vater(x)
   if y ≠ NIL then enferne x aus Kinderliste von y und reduziere Grad(y) um 1
                   füge x in Wurzelliste ein (lösche Markierung)
```

Für die amortisierte Laufzeitanalyse von DecreaseKey müssen wir unsere DM-Invariante etwas erweitern. Wir verlangen, daß bei jedem markierten Knoten 2 DM hinterlegt sind. Die DecreaseKey Operation bring $O(1)$ DM mit, mit denen die konstante Zeit bezahlt wird, die für das Abtrennen des Knotens vom Vater und das Einfügen in die Wurzelliste benötigt wird. Außerdem werden noch 2 DM beim ursprünglichen Vater dazu hinterlegt. Die Operationen für Cascading-cut werden durch die bei markierten Knoten hinterlegten DM bezahlt. Zu einem Cut in Cascading-Cut bei $z$ kommt es nur, wenn 4 DM beim Knoten $z$ hinterlegt sind (2 DM bei der ursprünglichen Markierung, 2 weitere beim Entfernen des zweiten Kindes, was jetzt den Cut auslöst). Von den 4 DM gibt man 1 DM zur Ausführung von Cut aus, 1 DM wird bei $z$ als neuem Element in der Wurzelliste hinterlegt (um die Invariante aufrechtzuerhalten) und 2 DM werden dem Vater von $z$ zusätzlich zugewiesen, womit der Cascading-Cut, falls nötig, weitergeführt werden kann. Insgesamt müssen also von DecreaseKey nur $O(1)$ DM mitgebracht werden, um für seine Durchführung und die Aufrechterhaltung der Invarianten zu bezahlen. Seine amortisierte Laufzeit ist also konstant.

Delete
Vermindere $x$ mit DecreaseKey auf $-\infty$ und führe Extractmin aus. Man erhält also eine amortisierte Laufzeit von $O(\log n)$.

## 5.4.2 Größenanalyse von Fibonacci Bäumen

In der vorangegangenen Analyse verwendeten wir zwei Behauptungen. Die erste war, daß ein Knoten $v$ in der Wurzelliste höchstens $O(\log n)$ Kinder hat, also Grad$(v) = O(\log n)$. (Dabei ist $n$ die Gesamtanzahl von Knoten im Fibonacci Heap.) Die zweite war, daß nach ihrer Konsolidierung in Extractmin die Wurzelliste höchstens $O(\log n)$ Bäume beinhaltet.
Beide Behauptungen ergeben sich aus dem folgenden Lemma:

**Lemma 8** *In einem Fibonacci Heap mit $n$ Knoten gilt für jeden Knoten $v$, daß Grad$(v) = O(\log n)$.*

Für die erste Behauptung ist die Folgerung klar; für die zweite beachte man, daß nach einer Konsolidierung die Wurzelliste keine zwei Wurzelknoten von gleichem Grad besitzt.
Für einen Knoten $v$ sei size$(v)$ die Anzahl seiner Nachkommen, also die Anzahl der Knoten im Teilbaum, dessen Wurzel $v$ ist. Das Lemma ist bewiesen, wenn wir zeigen können, daß size$(v) \geq c^{\mathrm{Grad}(v)}$ ist für irgendeine fixe Konstante $c > 1$.
Zuerst beweisen wir eine Aussage über die Grade der Kinder eines Knotens:

**Behauptung 16** *Sei $v$ ein Knoten von Grad $i$ und $w_1, w_2, \ldots, w_i$ die Kinder von $v$, geordnet in der Reihenfolge in der sie zu $v$ gelinkt wurden. Dann gilt: Grad$(w_j) \geq j - 2$ für alle $1 \leq j \leq i$.*

**Beweis** Zu dem Zeitpunkt, als $w_j$ zu $v$ gelinkt wurde, hatten beide den gleichen Grad, nämlich Grad$(w_j)$ = Grad$(v) \geq j - 1$. Später kann $w_j$ eventuell ein Kind verlieren (aber nicht mehrere). Der Grad von $w_j$ wird dadurch nur um höchstens eins verringert, also hat das $j$-te Kind mindestens $j - 2$ eigene Kinder. ☐

Im folgenden seien $f_0 = 0$, $f_1 = 1$, $f_{k+2} = f_{k+1} + f_k$ die Fibonacci-Zahlen. Induktiv kann man leicht zeigen, daß für alle $k$ die Ungleichung $f_{k+2} \geq c^k$ gilt, für $c = (1+\sqrt{5})/2 > 1$. Unser Lemma wird jetzt durch folgende Behauptung bewiesen:

**Behauptung 17** *Für jeden Knoten $v$ in einem Fibonacci-Heap mit Grad$(v) = k$ gilt* $\mathrm{size}(v) \geq f_{k+2}$.

**Beweis** Sei $S_k$ die minimale Größe eines Teilbaumes mit Wurzel vom Grad $k$. Es gilt:

$$\begin{aligned} S_0 &= 1 \\ S_1 &= 2 \\ S_k &= 1 + 1 + S_0 + S_1 + \cdots + S_{k-2} \qquad \text{nach obiger Behauptung} \end{aligned}$$

Analog gilt für $S_{k-1}$:

$$S_{k-1} = 1 + 1 + S_0 + S_1 + \cdots + S_{k-3}.$$

Subtraktion ergibt

$$S_k - S_{k-1} = S_{k-2} \qquad \text{oder} \qquad S_k = S_{k-1} + S_{k-2}\,.$$

Für die minimale Baumgrößen $S_k$ gilt also die Fibonacci Rekursion, und mit den Anfangsbedingungen sieht man, daß $S_k = f_{k+2}$.

Die Behauptung folgt, da $S_k$ das Minimum ist für $\mathrm{size}(v)$ mit $\mathrm{Grad}(v) = k$. □

# Kapitel 6

# Das Union-Find-Problem

Vorlesung vom 21.12.1995
Thorsten Heit

Sei $U = \{0, \ldots, n-1\}$ ein Universum, $n > 0$. Wir definieren eine Menge $P = \{P_0, \ldots, P_{k-1}\}$ von Partitionen $P_i$, die $U$ in disjunkte 'Blöcke unterteilen. Jedem $P_i$ ist ein Name zugeordnet, und zwar der Name eines Elements von $P_i$. Über dieser Menge $P$ können wir nun folgende Operationen durchführen:

- Find($x$): Es wird der Name des Blocks ermittelt, der $x \in U$ enthält.
- Union($A, B$): Zwei Blöcke $A$ und $B$ werden vereinigt. Dem entstandenen Block wird entweder der Name von $A$ oder der von $B$ zugeordnet

Eine Verwendungsmöglichkeit für diese Datenstruktur ist das Verwalten von Äquivalenzrelationen: Am Anfang bildet jedes Element einen eigenen Block. Durch entsprechende Union's werden alle äquivalenten Elemente in einem Block zusammengefaßt. Für zwei gegebene Elemente können wir mittels Find herausfinden, ob diese äquivalent sind. In diesem Fall liefert Find die gleichen Blocknamen zurück.

**Beispiel** Sei $U = \{0, \ldots, 8\}$ unser Universum, $P = \{P_0, \ldots, P_{k-1}\}$ mit $P_i = \{i\}$, d.h. jede Zahl aus $U$ bildet einen eigenen Block, dessen Name der zugehörigen Zahl entspricht. Wir führen nun auf $P$ eine Reihe von Union/Find-Operationen durch (die Namen der Blöcke sind fett gekennzeichnet):

| | Operation | Ergebnis | Blockstruktur |
|---|---|---|---|
| 0) | | | {**0**} {**1**} {**2**} {**3**} {**4**} {**5**} {**6**} {**7**} {**8**} |
| 1) | Union(2,3) | | {**0**} {**1**} {**2**,3} {**4**} {**5**} {**6**} {**7**} {**8**} |
| 2) | Union(7,8) | | {**0**} {**1**} {**2**,3} {**4**} {**5**} {**6**} {**7**,8} |
| 3) | Find(4) | 4 | |
| 4) | Union(4,7) | | {**0**} {**1**} {**2**,3} {4,**7**,8} {**5**} {**6**} |
| 5) | Find(4) | 7 | |
| 6) | Union(2,7) | | {**0**} {**1**} {2,3,4 ,**7**,8} {**5**} {**6**} |
| 7) | Find(4) | 7 | |

Zur Implementation des Union-Find-Problems geben wir drei Möglichkeiten an:

### Verkettete Listen

Die erste Möglichkeit ist, Blöcke durch einfach verkettete Listen zu verwaltet. Zusätzlich weiß jedes $x \in U$ den Namen seines Blocks, und jeder Block kennt seine Größe. Die Implementation der beiden Operationen sieht wie folgt aus:

a) Find($x$): Da jedes $x$ den Namen seines Blocks kennt, kann man direkt Name($x$) zurückliefern, benötigt also für diese Operation nur konstante Zeit $O(1)$.

b) UNION$(A, B)$: Hänge die kürzere der beiden Listen an die längere an und ändere die Blocknamen der Elemente der kürzeren Liste in die Blocknamen der Elemente der längeren ab. Eine Folge von $n-1$ UNION's benötigt eine Laufzeit von $O(n \log n)$: Bei Vereinigung der Blöcke $A$ und $B$ (ohne Einschränkung sei $A$ der kleinere von beiden) wird der resultierende Block mindestens doppelt so groß wie $A$. Daher kann jedes Element $x$ nur maximal $\log_2 n$-mal umbenannt werden (die Elemente des größeren Blocks bleiben "unangetastet").

Bei $m$ FIND- und $(n-1)$ UNION-Operationen führt dies zu einer Gesamtlaufzeit von $O(m + n \log n)$.

**Beispiel** Seien $U$, $P$ und die Folge von UNION's und FIND's wie im obigen Beispiel:

0) Startzustand:

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Blockname | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Größe | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Nachfolger | - | - | - | - | - | - | - | - | - |

1) UNION(2,3)

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Blockname | 0 | 1 | 2 | 2 | 4 | 5 | 6 | 7 | 8 |
| Größe | 1 | 1 | 2 | | 1 | 1 | 1 | 1 | 1 |
| Nachfolger | - | - | 3 | - | - | - | - | - | - |

2) UNION(7,8)

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Blockname | 0 | 1 | 2 | 2 | 4 | 5 | 6 | 7 | 7 |
| Größe | 1 | 1 | 2 | | 1 | 1 | 1 | 2 | |
| Nachfolger | - | - | 3 | - | - | - | - | 8 | - |

4) UNION(4,7)

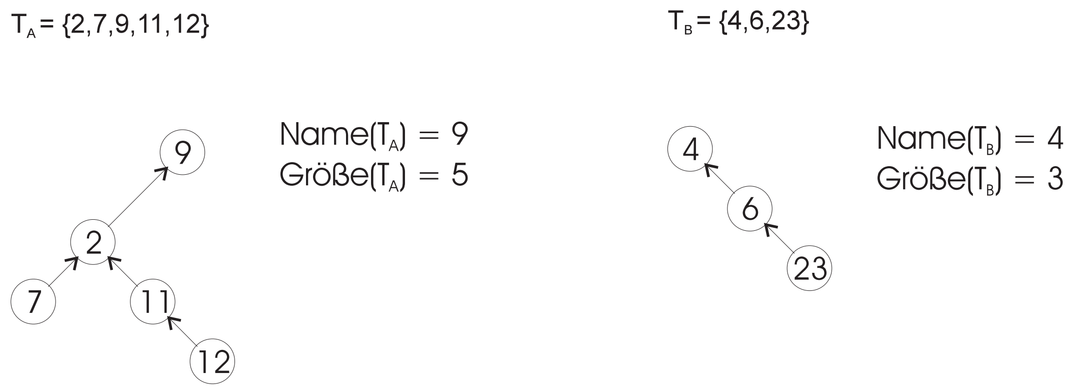
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Blockname | 0 | 1 | 2 | 2 | 7 | 5 | 6 | 7 | 7 |
| Größe | 1 | 1 | 2 | | | 1 | 1 | 3 | |
| Nachfolger | - | - | 3 | - | - | - | - | 8 | 4 |

6) UNION(2,7)

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Blockname | 0 | 1 | 7 | 7 | 7 | 5 | 6 | 7 | 7 |
| Größe | 1 | 1 | | | | 1 | 1 | 5 | |
| Nachfolger | - | - | 3 | - | 2 | - | - | 8 | 4 |

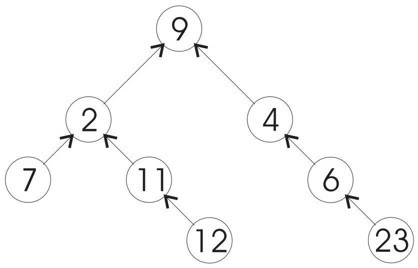
### Bäume mit Vereinigung nach Größe

Bei der zweiten Möglichkeit wird jeder Block in Form eines Baumes dargestellt, wobei der Name des Blocks dem Namen des Wurzelelements entspricht. Im Unterschied zur üblichen Implementierung existieren nur Zeiger von den Kinder- auf die Elternknoten; zusätzlich merkt sich jeder Block (Baum) seine Größe.




Die Implementation der beiden Operationen Union und Find geschieht folgendermaßen:

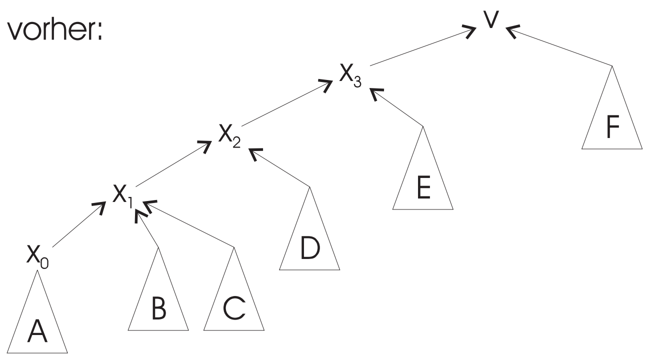
a) UNION$(A, B)$: Der kleinere Baum wird an die Wurzel des größeren angehängt (Vereinigung nach Größe) (Laufzeit: $O(1)$). Der dadurch entstandene Baum hat eine Tiefe, die um höchstens 1 größer ist als die Tiefe des tieferen der beiden Bäume $A$ und $B$. Für einen Baum mit $k$ Knoten ergibt sich somit eine Tiefe $\leq \lceil \log_2 k \rceil$ . Im obigen Beispiel ergibt die Vereinigung folgenden Baum:

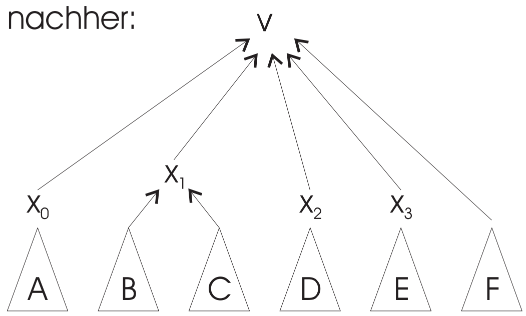



b) FIND$(x)$: Wandere im Baum vom Knoten $x$ an nach oben bis zur Wurzel und liefere deren Namen zurück. Die maximale Laufzeit beträgt (wenn $x$ ein Blatt ist) $O\left(\lceil \log_2 k \rceil\right)$ mit $k =$ Anzahl der Knoten im Baum.

Bei $m$ FIND- und $(n-1)$ UNION-Operationen führt dies zu einer Gesamtlaufzeit von $O\left(n + m \log n\right)$. Diese schlechte Laufzeit von FIND liegt daran, daß bereits erfolgte Berechnungen nicht wieder verwertet werden können. In obigem Beispiel wird z.B. bei FIND(11) und FIND(12) im Baum der gleiche Pfad zweimal durchlaufen. Um dieses zu verhindern, benutzt man die sogenannte *Pfadkompression*, was die dritte Möglichkeit ergibt:

**Pfadkompression**




Sei $\{x_0, \ldots, x_k, v\}$ die Menge aller Knoten, die bei einem Aufruf von FIND($x_0$) durchlaufen wird. Bei einer Pfadkompression werden alle $x_i$, $0 \le i \le k-1$, direkt an die Wurzel $v$ angehängt. Bei einer weiteren FIND-Operation, die den gleichen Pfad benutzen würde, muß man also nur noch einen der Knoten $x_0, \ldots, x_k$ betrachten.

**Satz 14** *Mit Vereinigung nach Größe und Pfadkompression braucht eine Folge von $n-1$ UNION's und $m \ge n$ FIND's die Zeit $O\left(m\dot\alpha\left(m,n\right)\right)$, wobei $\alpha$ die inverse Ackermann-Funktion bezeichnet.*

**Definition 10** *a) Die Ackermann-Funktion $A : \mathbb{N}^2 \to \mathbb{N}$ ist folgendermaßen definiert:*

$$\begin{aligned} A(i,0) &= 1 \qquad i \ge 1 \\ A(0,x) &= 2x \qquad x \ge 0 \\ A(i+1,x+1) &= A(i, A(i+1,x)) \qquad i,x \ge 0 \end{aligned}$$

*b)*

$$\alpha\left(m,n\right) := \min\left\{z \ge 1 \mid A\left(z, 4\left\lceil\frac{m}{n}\right\rceil\right) > \log n\right\}$$

*wird als inverse Ackermann-Funktion bezeichnet.*

Zur Verdeutlichung der Ackermann-Funktion:

| $i \diagdown x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | $\cdots$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 2 | 4 | 6 | 8 | 10 | 12 | |
| 1 | 1 | 2 | 4 | 8 | 16 | 32 | 64 | |
| 2 | 1 | 2 | 4 | 16 | 65536 | $2^{65536}$ | | |
| 3 | 1 | 2 | 4 | 65536 | $\left.2^{\cdot^{\cdot^{2}}}\right\}65536$ | | | |
| $\vdots$ | | | | | | | | |

Man sieht direkt, daß die Ackermann-Funktion schon bei kleinen Eingaben sehr schnell sehr groß wird (zur Verdeutlichung: $2^{65536} = 2.0035 \times 10^{19728}$). Umgekehrt gilt $\alpha(m,n) \le 3$ für alle praktischen Werte von $m$ und $n$.

<u>Beweisskizze</u>: (einen vollständigeren Beweis finden Sie in Mehlhorn's Buch, Seiten 284 ff.)

Wir betrachten eine fixe Folge von UNION's und FIND's. Zum Zweck der Analyse tun wir so, als würden zuerst *alle* UNION's durchführen. Dies ergibt einen Baum $T$. Erst danach werden alle FIND's durchgeführt, jeweils mit einer Pfadkompression wie in der ursprünglichen Folge. Das heißt, Pfadkompressionen enden nun nicht mehr unbedingt an der Wurzel, es sind also "partielle Kompressionen". Es gilt, daß eine beliebige Folge von $n-1$ UNIONS und $m$ FINDS durch $n-1$ UNIONS gefolgt von $m$ FINDS mit partieller Kompression mit den gleichen Kosten nachvollzogen werden kann.
Für jedes Element $x$ aus $T$ definieren wir einen Rang $rg(x)$ = anfängliche Höhe von $x$ in $T$. Falls $[x,y\rangle$ eine Kante in $T$ ist, so gilt $rg(x) < rg(y)$.
Wenn $F$ die Menge aller Kanten, die bei FIND-Operationen durchquert werden, bezeichnet, dann sind die Gesamtkosten aller FIND's $= |F|$.

Zu zeigen ist:

$$|F| = O\left(m \cdot \alpha(m,n)\right).$$

Definiere dazu

$$G_{ij} := \{x \mid A(i,j) \leq rg(x) < A(i,j+1)\}.$$

Wir unterteilen $F$ in Mengen $N_k$, $0 \leq k \leq \alpha(m,n)+1$, wobei wir die $N_k$ wie folgt definieren:

$$N_k := \{[x,y\rangle \in F \mid k = \min\{i \mid \exists j : x,y \in G_{ij}\}\}.$$

und

$$N_{\alpha+1} := F \setminus \bigcup_{j \leq k} N_j$$

Weiter definieren wir:

$$L_k := \{(x,y) \in N_k \mid \text{auf dem partiellen FIND-Pfad, der } (x,y) \text{ enthält, ist } (x,y) \text{ die letzte Kante in } N_k\}.$$

Dann gilt:

$$|F| = \sum_{k=0}^{\alpha+1} \underbrace{(|L_k| + |N_k - L_k|)}_{=|N_k|}$$

Wenn man die folgenden vier Behauptungen zeigen kann, so folgt $|F| = O\left(m \cdot \alpha(m,n)\right)$.

a) $|L_k| \leq m$ $\quad(0 \leq k \leq \alpha+1)$
b) $|N_0 - L_0| \leq n$
c) $|N_k - L_k| \leq n$ $\quad(1 \leq k \leq \alpha)$
d) $|N_{\alpha+1} - L_{\alpha+1}| \leq 8m$

# Kapitel 7

# Graphenalgorithmen

Vorlesung vom 08.01.1996
Michael Brandau, Jürgen Mohra

Nachdem wir bisher hauptsächlich Datenstrukturen behandelt haben, wollen wir uns nunmehr mit Algorithmen beschäftigen. Eine wichtige Klasse von Algorithmen sind die Graphen-Algorithmen, über die wir in diesem Kapitel sprechen wollen.
Viele Probleme der Informatik lassen sich auf Graphen abbilden und dadurch elegant lösen. Doch bevor wir uns mit Graphen-Algorithmen beschäftigen, müssen wir erst einmal klären, was ein Graph ist.

## 7.1 Grundlagen

Ein *gerichteter Graph* $G$ ist ein Paar $(V, E)$, wobei

- $V$ eine endliche Menge von Knoten (Vertices) und
- $E \subset V \times V$ eine endliche Menge von Kanten (Edges) ist.

Eine *Kante* $e$ von Knoten $v$ nach Knoten $w$ wird dargestellt als $e = [v, w\rangle$, wobei

- $v$ als Startknoten ($head(e)$ oder $source(e)$) und
- $w$ als Endknoten ($tail(e)$ oder $target(e)$)

bezeichnet wird.

Bei einer Kante $e = [v, w\rangle$ in einem Graphen bezeichnet man den Knoten $w$ als *Nachbarn* des Knotens $v$. Man sagt auch $v$ und $w$ sind *adjazent* oder $v$ ist adjazent zu $w$. Die Beziehung zwischen der Kante $e$ und ihren Knoten $v$ und $w$ bezeichnet man als *Inzidenz* und man sagt, $e$ ist inzident auf den Knoten $v$ und $w$ oder $e$ ist inzident von Knoten $v$ (verläßt Knoten $v$) und inzident nach Knoten $w$ (führt zu Knoten $w$).

### 7.1.1 Darstellung von Graphen

Die Beschreibung eines Graphen $G$ geschieht über die Mengen $V$ und $E$. Will man $G$ nun darstellen, so gibt es mehrere Möglichkeiten, die jeweils unterschiedlich günstig sind für spätere Betrachtungen:

1. Die einfachste Möglichkeit ist die, die Mengen $V$ und $E$ als Mengen wiederzugeben (wie wir gesehen haben, gibt es dafür mehrere Möglichkeiten).

2. Eine andere Möglichkeit ist, $V$ als Menge darzustellen (typischerweise als Feld realisiert) und für alle $v \in V$ in einem Feld $Out[v]$ die Menge der Kanten $[v, w\rangle$, die von $v$ *abgehen* bzw. die Menge der Knoten $w$ , zu denen eine Kante $[v, w\rangle$ führt, zu speichern:

$$\begin{aligned} Out[v] &= \{[v, w\rangle \mid [v, w\rangle \in E\} \quad \text{oder} \\ Out[v] &= \{w \mid [v, w\rangle \in E\} \end{aligned}$$

Analog zur Menge $Out[v]$ definiert man die Menge $In[v]$ als die Menge aller Kanten, die nach $v$ führen, bzw. als die Menge der Knoten $w$ von denen eine Kante nach $v$ führt:

$$\begin{array}{l} In[v] = \{[w,v\rangle \mid [w,v\rangle \in E\} \quad \text{oder} \\ In[v] = \{w \mid [w,v\rangle \in E\} \end{array}$$

Typischerweise realisiert man diese beiden Mengen entweder als verkettete Liste (dann erhält man Adjazenslisten) oder als Bitvektor (die Bitvektoren aller Knoten im Graph bilden eine Adjazenzmatrix).
Die Anzahl der von $v$ ausgehenden Kanten wird als *Ausgangsgrad (outdeg($v$))* des Knotens $v$ und die Anzahl der in $v$ hineinführenden Kanten wird als *Eingangsgrad (indeg($v$))* des Knotens $v$ bezeichnet:

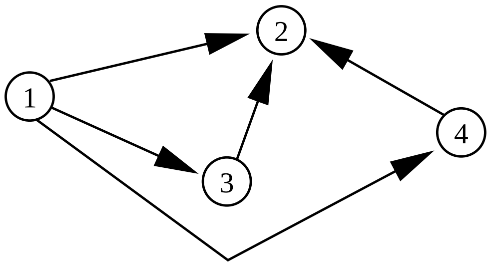
$$\begin{array}{l} outdeg(v) = |Out[v]| \\ indeg(v) = |In[v]| \end{array}$$

Dabei gilt:

$$\sum_{v \in V} indeg(v) = \sum_{v \in V} outdeg(v) = |E|,$$

d.h. die Summe der Eingangsgrade über alle Knoten des Graphen entspricht der Summe der Ausgangsgrade über alle Knoten und diese muß gleich der Anzahl der Kanten sein.

**Beispiel** Darstellungsmöglichkeiten eines gerichteten Graphen:




- Darstellung als Adjazensliste:
  Dem Knoten in der ersten Spalte (vor dem Doppelpunkt) folgt eine Liste der Knoten, zu denen eine Kante hinführt.

  1 : 2, 3, 4
  2 : -
  3 : 2
  4 : 2

- Darstellung als Adjazensmatrix:

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

  Eine 1 in Zeile $i$ und Spalte $j$ bedeutet, daß eine Kante von Knoten $i$ zum Knoten $j$ führt.

Neben Knoten und Kanten eines Graphen definiert man auch noch Pfade in einem Graphen:

Ein *Pfad der Länge k* von einem Knoten $u$ zu einem Knoten $u'$ in einem Graphen $G = (V, E)$ ist eine Folge $(v_0, v_1, ..., v_k)$ von Knoten, so daß $u = v_0$, $u' = v_k$ und $[v_i, v_{i+1}\rangle \in E$ für $i = 0, ..., k-1$.
Die Länge $k$ des Pfades entspricht der Anzahl der Kanten im Pfad.

Ein Pfad heißt *einfach*, wenn alle Knoten im Pfad verschieden sind: $v_i \neq v_j$ für $i \neq j$.

Ein Pfad bildet einen *Kreis (Zyklus)*, wenn $v_0 = v_k$.
Ein Graph, der einen Kreis enthält, heißt *zyklisch*, sonst heißt er *azyklisch*.

Ein Graph $G = (V, E)$ heißt *(gerichteter) Baum* genau dann, wenn folgendes gilt:

- es gibt genau einen Knoten $r$ mit $outdeg(r) = 0$,
- für alle Knoten $v \neq r$ gilt: $outdeg(v) = 1$,
- $G$ ist azyklisch.

## 7.2 Topologisches Sortieren

Gegeben sei ein gerichteter Graph $G = (V, E)$. Eine Abbildung

$$num : V \to \{1, ..., n\}, \qquad n = |V|$$

heißt *topologischeSortierung* von $G$, wenn für alle $[u, v\rangle \in E$ gilt:

$$num(u) < num(v)$$

**Lemma 9** *$G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn $G$ azyklisch ist.*

**Beweis**

$\Rightarrow$ : $G$ ist topologisch sortiert. Angenommen, $G$ wäre zyklisch. Dann gäbe es einen Pfad $(v_0, v_1, ..., v_k)$ mit $v_0 = v_k$. Da $G$ topologisch sortiert ist, gilt

$$num(v_0) < num(v_1) < ... < num(v_k)$$

und daher $num(v_0) < num(v_k)$. Widerspruch.
$\Longrightarrow G$ ist azyklisch

$\Leftarrow$ : $G$ ist azyklisch. Die Aussage folgt aus folgender Behauptung.

**Behauptung 18** *Wenn $G$ azyklisch ist, dann gibt es einen Knoten $v$ mit $indeg(v) = 0$.*

**Beweis** Angenommen, es existiert kein Knoten $v \in V$ mit $indeg(v) = 0$, d.h. für alle Knoten $v \in V$ gibt es einen Knoten $p(v) \in V$ mit $[p(v), v\rangle \in E$. Konstruiere eine Folge von Knoten $p^i$, $0 \leq i \leq n$:

$$p^i(v) = \begin{cases} v & , i = 0 \\ p(p^{i-1}(v)) & , i > 0 \end{cases}$$

Da $|V| = n$ ist, existiert nach dem Schubfachprinzip in der Folge $p^0(v), p^1(v), \ldots, p^n(v)$ ein $i \neq j$ mit $p^i(v) = p^j(v)$, d.h. in $G$ existiert ein Zykel $p^i(v) \to p^{i+1}(v) \to \ldots \to p^j(v)$. Widerspruch zu $G$ azyklisch.
□

□

Mit Behauptung 45 ergibt sich folgendes Verfahren zum topologischen Sortieren eines Graphen $G = (V, E)$:

1. Finde Knoten $v \in V$ mit $indeg(v) = 0$
2. numeriere ihn
3. entferne ihn aus der Menge der Knoten, die betrachtet werden müssen
4. wiederhole die drei Schritte solange noch Knoten unnummeriert sind.

**Algorithmus zum Topologischen Sortieren**

$G$ sei dargestellt durch die Mengen $V$ und $E$:

```
(1)   foreach v ∈ V do
(2)       indeg[v] = 0
(3)       OUT[v]=∅
(4)   foreach [v, w⟩ ∈ E do
(5)       indeg[w]++
(6)       Insert(w,OUT[v])
(7)   ZERO=∅
(8)   foreach v ∈ V do
(9)       if indeg[v]=0 then
(10)          Insert(v,ZERO)
(11)  count = 0
(12)  while ZERO ≠ ∅ do
(13)      wähle und entferne v ∈ ZERO
(14)      count++
(15)      num[v] = count
(16)        foreach w ∈ OUT[v] do
(17)          indeg[w]−−
(18)          if indeg[w] = 0 then
(19)              Insert(w,ZERO)
(20)   if count < |V| then Error (Zyklus)
```

Zunächst werden die Adjazenslisten für $G = (V, E)$ aufgebaut (1-6). Dann wird eine Menge *ZERO* aller Knoten mit Eingangsgrad 0 erzeugt (7-10). Der eigentliche Algorithmus zum topologischen Sortieren schließt sich an (11-20). Enthält der Graph $G$ einen Zyklus, so wird dieser dadurch entdeckt, daß die Anzahl *count* der nummerierten Knoten kleiner ist als die Anzahl der in $G$ enthaltenen Knoten.
Die Laufzeit des Algorithmus liefert folgende Tabelle:

| Zeilennr. | Laufzeit |
|---|---|
| 1 - 3 | $O(\|V\|)$ |
| 4 - 6 | $O(\|E\|)$ |
| 7 | $O(1)$ |
| 8 - 10 | $O(\|V\|)$ |
| 11 | $O(1)$ |
| 13 - 15 | $O(1)$ |
| 16 - 19 | $O(outdeg(v))$ |
| 20 | $O(1)$ |

Gesamtlaufzeit der WHILE-Schleife (Zeilen 12 - 19):

$$\sum_{v \in V} O(1 + outdeg(v)) = O(|V| + |E|).$$

Daraus ergibt sich als Gesamtlaufzeit des Algorithmus: $O(|V| + |E|)$.

Vorlesung vom 11.1.1996
Jens Simon, Sebastian Braun, Wolfgang Holz

## 7.3 Breadth First Search (BFS)

Gegeben ist ein Graph $G$ mit einem Startknoten $s$. Wir wollen für alle Knoten $v \in V$ die kürzesten Pfade von $s$ nach $v$ in $G$ sowie deren Längen (Anzahl der Kanten) bestimmen, sofern sie von $s$ aus erreichbar sind. Dies geschieht durch folgenden Algorithmus:

```
BFS(G, s)
(1)   for each u ∈ V\{s} do
(2)       color[u] = weiß
(3)       d[u] = ∞ /* distance */
(4)       p[u] = NIL /* parent */
(5)   color[s] = grau
(6)   d[s] = 0
(7)   p[s] = NIL
(8)   ENQUEUE(s, Q)
(9)   while Q ≠ empty do
(10)      u = head(Q)
(11)      for each v ∈ OUT(u) do
(12)          if color[v]=weiß then color[v]=grau
(13)             d[v] = d[u] + 1
(14)             p[v] = u
(15)             ENQUEUE(v, Q)
(15)      DEQUEUE(Q)
(16)      color[u] = schwarz
```

Dieser Algorithmus erzeugt für jeden Knoten $v$ zwei Werte, die folgendes darstellen sollen: die Distanz $d[v]$ soll die kleinste Anzahl von Kanten sein, die ein Pfad von $s$ nach $v$ haben kann; der Knoten $p[v]$ soll der vorletzte Knoten auf so einem kürzesten $s$ — $v$ Pfad sein. Ist ein Knoten $v$ von $s$ nicht erreichbar, dann gilt $d[v] = \infty$ und $p[v] = \text{NIL}$.
Der Algorithmus verwendet Farben für Knoten. Wenn ein Knoten weiß gefärbt ist, so bedeutet das, daß im Verlauf des Algorithmus noch nicht betrachtet wurde. Ist er schwarz gefärbt, so war er schon einmal in der Schlange, ist aber wieder draußen, und wenn er grau gefärbt ist, so ist er noch in der Schlange. Ein Knoten kann sich nur von weiß nach grau und von grau nach schwarz verfärben, d.h. ein Knoten kann nur einmal in die Schlange aufgenommen und auch nur einmal aus ihr entfernt werden.
Das führt zu folgender Laufzeitanalyse:
Sei $|V| = n$, $|E| = e$. Die Zeilen (1)-(8) benötigen eine Laufzeit von $O(n)$. Da jeder Knoten höchstens einmal in die Schlange $Q$ eingefügt werden kann, wird die **while**-Schleife in Zeile (9) höchstens $n$-mal durchlaufen. Die **for each**-Schleife in Zeile (11) läuft über alle ausgehenden Kanten von $u$, benötigt also $O(outdeg(u)$ Zeit. Damit ergibt sich folgende Gesamtlaufzeit:

$$O\left(\sum_{u\in V}(1 + outdeg(u))\right) = O(n + e).$$

Wieso ist das Programm korrekt?
Definiere für $x, y \in V$:

$$\delta(x,y) := \begin{cases} \text{Länge eines kürzesten Pfades von x nach y, falls Pfad existiert} \\ \infty\text{, falls es keinen Pfad von x nach y gibt} \end{cases}$$

**Satz 15** *Nach Beendigung von* BFS$(G, s)$ *gilt für alle Knoten* $v \in V$*:*

1. $d[v] = \delta(s, v)$
2. *falls $v \neq s$ ist, so ist ein kürzester Weg von $s$ nach $v$ gegeben durch einen kürzesten Weg von $s$ nach* $p[v]$ *und die Kante* $[p[v], v\rangle$.

Beachte: 1. folgt aus 2. induktiv mit Hilfe von Zeile (13) in BFS. Es genügt also, die erste Aussage zu beweisen. Dazu benötigen wir zwei Aussagen:

**Behauptung 19** *Für alle Knoten $v \in V$ gilt ab der Initialisierung während des gesamten Algorithmus:* $d[v] \geq \delta(s, v)$.

**Beweis** Induktion über die Entdeckungszeit der Knoten $v \in V$.

IA: Zu Beginn des Algorithmus werden alle Knoten $v \in V\backslash\{s\}$ mit $d[v] = \infty > \delta(v, s)$ initialisiert und auf weiß gesetzt. Knoten $s$ wird in den Zeilen (5)-(7) in die Queue $Q$ eingefügt, grau gefärbt und die Distanz $d[s]$ wird auf 0 gesetzt: $d[s] = 0 = \delta(s, s)$. Nach Beendigung der Zeile (8) gilt also die Behauptung.

IS: Sei $v$ ein weißer Knoten, der ausgehend von einem Knoten $u$ entdeckt wird. Nach Induktionsvoraussetzung gilt für den Knoten $u$: $d[u] \geq \delta(u,s)$. Aus Zeile (13) des Algorithmus folgt:

$$\begin{aligned} d[v] &= d[u]+1 \\ &\geq \delta(u,s)+1 \\ &\geq \delta(v,s) \end{aligned}$$

Knoten $v$ wird in die Queue $Q$ eingefügt und seine Farbe auf grau gesetzt. Sein $d[]$-Wert wird danach nicht mehr geändert, da die Zeilen (12)-(15) nur für weiße Knoten ausgeführt werden. Auch nach diesem Schritt gilt also die Invariante.

□

**Behauptung 20** *Sei $\langle v_1, v_2, \ldots, v_r\rangle$ der Inhalt von $Q$ zu irgendeinem Zeitpunkt des Algorithmus. Dann gilt:*

$$d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1]+1.$$

*(Bemerkung: $head(Q) = v_1$, $\quad enqueue(v,Q) \Rightarrow Q = \langle v_1, v_2, \ldots, v_r, v\rangle$)*

**Beweis** Induktion über der Anzahl der Queue-Operationen.

IA: Zu Beginn von Bfs gilt: $Q = \langle s\rangle$ mit $d[s] = 0 \leq d[s]+1$.

IS: Man muß nun die Zeitpunkte betrachten, an denen sich etwas an der Schlange ändert, also Enqueue und Dequeue. Bei Dequeue($Q$) passiert folgendes: $\langle v_1, v_2, \ldots, v_r\rangle$ wird zu $\langle v_2, \ldots, v_r\rangle$. Da nach Induktionsannahme $d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1]+1$ gilt, gilt auch

$$d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1]+1 \leq d[v_2]+1;$$

die Invariante bleibt also erhalten.
Betrachte nun Enqueue($v,Q$): Kurz bevor Enqueue in Zeile (15) ausgeführt und $v$ als letztes Element $v_{r+1}$ in $Q$ eingefügt wird, wird in Zeile (13) $d[v] = d[v_{r+1}] = d[u]+1$ gesetzt, wobei $u = head(Q) = v_1$ ist. Aus der Induktionsannahme folgt dann:

$$d[v_1] = d[u] \leq \ldots \leq d[v_r] \leq d[v_1]+1 = d[u]+1 = d[v] = d[v_{r=1}] = d[v_1]+1.$$

Auch beim Einfügen bleibt die Invariante erhalten.

□

Nun zum Beweis von Satz 15:
**Beweis**

1. $\delta(s,v) = \infty$
   d.h. $v$ ist von $s$ aus nicht erreichbar. Nach Behauptung 19 gilt immer $d[v] \geq \delta(s,v) = \infty$, also auch am Ende des Algorithmus. Daraus folgt $d[v] = \infty$.

2. $\delta(s,v) < \infty$
   Definiere
   $$V_k := \{v|\delta(s,v) = k\},$$
   also $V_0 = \{s\}$, $V_1 = OUT(s)$.

   Wir wollen nun durch Induktion über $k$ zeigen, daß für alle $k$ der Satz für alle $v$ in $V_0 \cup \ldots \cup V_k$, also für alle $v$ gilt:

   IA: $k = 0$
   Für alle $v \in V_0$ ist der Satz korrekt.

   IV: Der Satz ist für alle $v \in V_0 \cup V_1 \cup \cdots \cup V_{k-1}$ korrekt.

IS: $k-1 \to k$

Sei $v \in V_k$, also $\delta(s,v) = k$. Definiere $IN(v) := \{u \mid [u,v\rangle \in E\}$. Für alle $u \in IN(v)$ gilt: $\delta(s,u) \geq k-1$, denn gäbe es ein $u \in IN(v)$, so daß $\delta(s,u) < k-1$ wäre, so wäre $\delta(s,v) < k$ und damit $v \notin V_k$. Also gibt es auch ein $u \in IN(v)$, so daß $\delta(s,u) = k-1$ ist. Sei $w$ der erste Knoten aus $IN(v)$, der in $Q$ eingefügt wird.

Dann ist $w \in V_{k-1}$, da

- $w$ der erste Knoten aus $IN(v)$ ist, der eingefügt wird,
- nach Behauptung 20 die $d[]$-Werte nicht kleiner werden können,
- es mindestens ein $u \in IN(v)$ gibt, so daß $\delta(s,u) = k-1$, d.h. $u \in V_{k-1}$ ist,
- für alle Knoten $x \in V_{k-1}$ nach Induktionsannahme $d[x] = \delta(s,x)$ gilt.

Da $w$ als erstes Element aus $IN(v)$ in die Schlange $Q$ kam, wird später $d[v]$ zu $d[w] + 1 = k$ gesetzt.

□

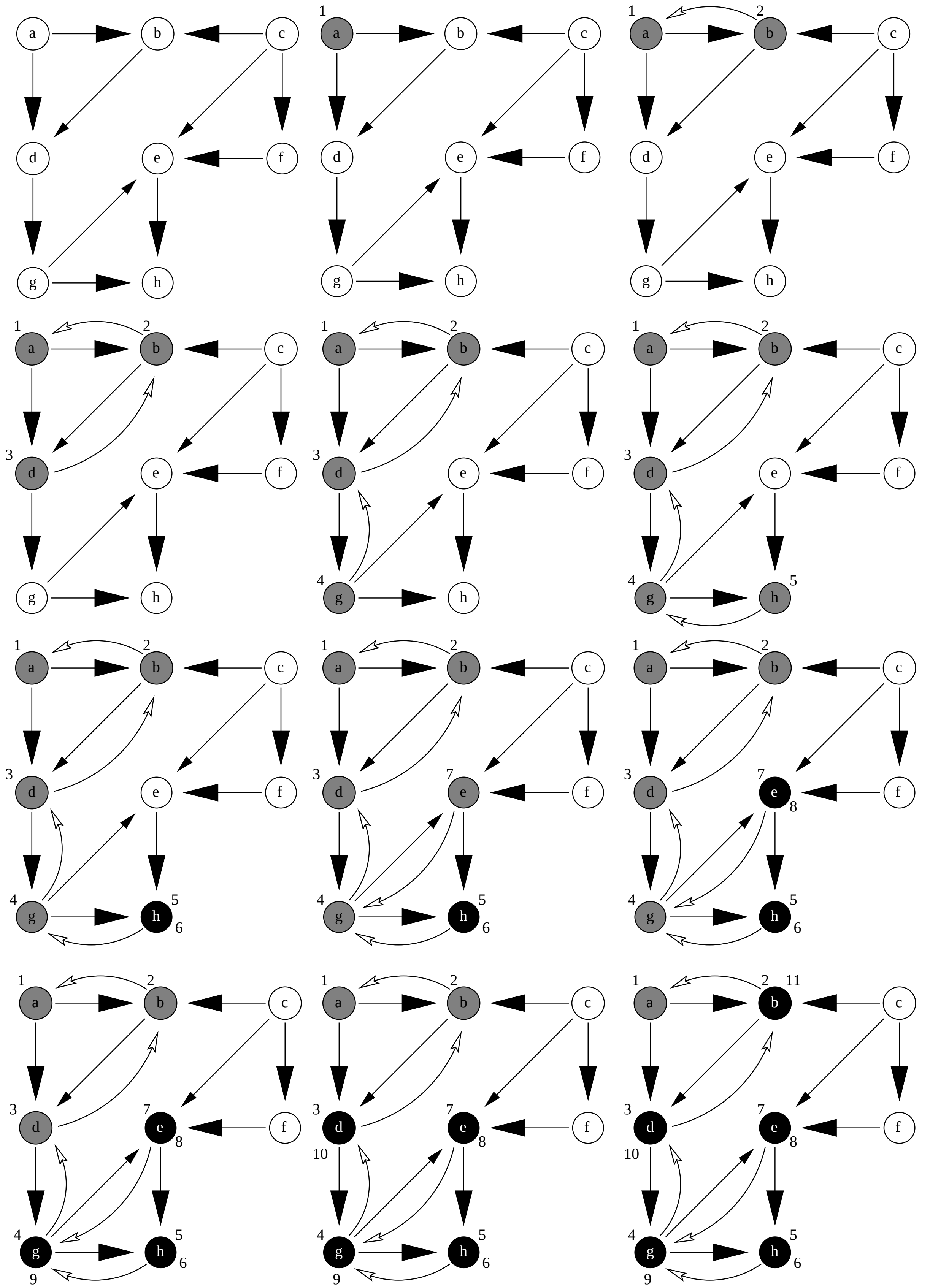
Wenn man die Kantenmenge $T = \{[p(v), v\rangle : v \in V,\ p(v) \neq \text{NIL}\}$ betrachtet, sieht man leicht, daß sie einen Baum aufspannt, der alle Knoten enhält, die von $s$ erreichbar sind. Außerdem stellen die von $s$ aus gehenden Wege im Baum *kürzeste Wege* dar (hinsichtlich Anzahl der Kanten).
Der Baum $T$ wird oft als *BFS-Baum* bezeichnet. Seine Knoten werden durch die Distanzwerte $d[]$ in Schichten nivelliert. Der Baum hat die schöne Eigenschaften, daß es im ursprüngliche Graphen keine Kante gibt, die ein Niveau überspringt; also für jede Kante $[u,v\rangle \in E$ gilt $d[v] \leq d[u] + 1$.
Ist der Graph *ungerichtet*, dann gilt daß jede Kante des Graphs zwei Knoten auf dem gleichen oder auf benachbarten Niveaus verbindet.

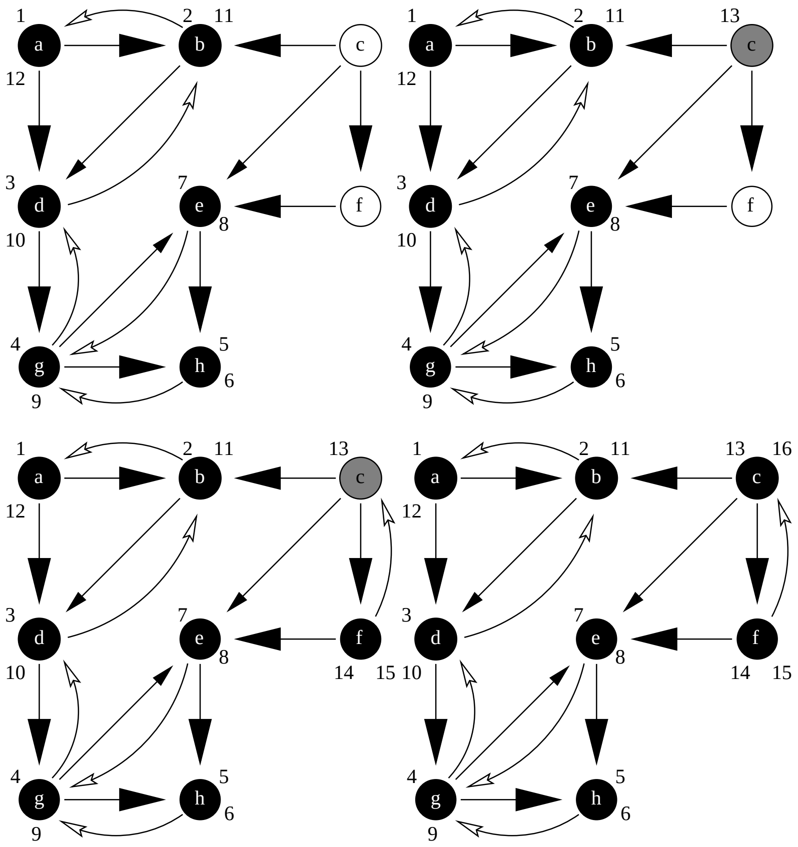
## 7.4 Depth First Search (DFS)

*Tiefensuche* (engl. Depth-First-Search, abgk. DFS) ist eine zweite Methode, um alle Knoten und Kanten eines Graphen organisiert zu durchforschen. Im Gegensatz zur Breitensuche, versucht man dabei möglichst schnell möglichst weit zu kommen. Das bedeutet, daß nach der Entdeckung eines Knotens man sofort bei einem seiner bis jetzt unentdeckten Nachbarn weitersuchen möchte.
Im folgenden geben wir ein Program DFS($G$) an, daß eine Tiefensuche in einem Graphen $G$ durchführt. Im Gegensatz zur Breitensuche BFS($G,s$), bei der nur die Knoten besucht werden, die vom Startknoten $s$ erreichbar sind, ist DFS so organisiert, daß alle Knoten des Graphs besucht werden. Dabei wird ein Wald von disjunkten Bäumen aufgebaut, die zusammen alle Knoten von $G$ aufspannen.
Der Algorithmus DFS verwendet so wie BFS wieder eine Knotenfärbung, wobei die Farben die gleiche Bedeutung haben wie vorher. Für jeden Knoten $v$ wird wieder eine Art Vorgängerknoten $\pi[v]$ berechnet (außer für die Wurzeln der Bäume). Schließlich werden noch für jeden Knoten $v$ zwei Zahlen $d[v]$ und $f[v]$ berechnet, die, intuitiv gesehen, angeben, wann ein Knoten $v$ entdeckt wurde und wann seine Erforschung abgeschlossen wurde. (Zu diesem Zweck verwendet der Algorithmus die globale Zeitvariable $t$.) Wie wir später sehen werden, sind die Zahlenpaare $(d[v], f[v])$ für alle möglichen Zwecke sehr nützlich.

```
DFS(G)
   for each vertex u ∈ V do color[u] = weiß
                            π[u] = NIL
   t = 0  /* globale Zeitvariable */
   for each vertex u ∈ V do
       if color[u] = weiß then DFS-VISIT(u)

DFS-VISIT(u)
   color[u] = grau
   t = t + 1
   d[u] = t /* discovery */
   for each v ∈ OUT(u) do
       if color[v] = weiß then π[v] = u
                               DFS-VISIT(v)
   color[u] = schwarz
   t = t + 1
   f[u] = t /* final */
```


a b c d e f g h
1 a b c d e f g h
1 2 a b c d e f g h
1 2 a b c 3 d e f g h
1 2 a b c 3 d e f 4 g h
1 2 a b c 3 d e f 4 g 5 h
1 2 a b c 3 d e f 4 g 5 h 6
1 2 a b c 3 d 7 e f 4 g 5 h 6
1 2 a b c 3 d 7 e 8 f 4 g 5 h 6
1 2 a b c 3 d 7 e 8 f 4 g 9 5 h 6
1 2 a b c 3 d 10 7 e 8 f 4 g 9 5 h 6
1 2 11 a b c 3 d 10 7 e 8 f 4 g 9 5 h 6




Die vorherigen Abbildungen illustrieren den Ablauf von DFS auf einem Graphen mit acht Knoten. Die weißen Pfeile geben die (inversen) Baumkanten $[v, \pi[v]\rangle$ an. Die bei den Knoten stehenden Zahlen sind die Werte $d[v]$ und $f[v]$, sobald sie bekannt sind.

Vorlesung vom 15.1.1996
Michaela Altherr, Ulrich Laux

Wir wollen nun einige wesentliche Eigenschaften der von DFS produzierten Werte $d[v]$, $f[v]$ und $\pi[v]$ beweisen. Den Wald, der durch die Kanten $[v, \pi[v]\rangle$ induziert wird, bezeichnen wir als *DFS-Wald.*

**Satz 16** *(Klammersatz)*
*Für einen Knoten $v \in V$ sei $I_v = [d[v], f[v]]$. Dann gilt für alle Knoten $u, v \in V$:*

$$I_u \subset I_v \quad \text{oder} \quad I_u \supset I_v \quad \text{oder} \quad I_u \cap I_v = \emptyset,$$

*d.h. die Intervalle $I_u$ und $I_v$ sind geschachtelt oder disjunkt.*

**Beweis** Ohne Einschränkung sei $d[u] < d[v]$.

**1.Fall:** $d[v] < f[u]$

In diesem Fall wird Knoten $v$ zu einem Zeitpunkt entdeckt, an dem $u$ noch grau gefärbt ist. Als nächstes werden die von $v$ zu bisher noch nicht gefärbten Knoten ausgehenden Kanten betrachtet. Danach bekommt $v$ seine Endmarke $f[v]$ und zum Schluß wird die Rekursion für $u$ beendet und $u$ bekommt seine Endmarke $f[u]$. Es gilt also $I_v \subset I_u$.

**2.Fall:** $f[u] < d[v]$

Da $d[u] < f[u]$ und $d[v] < f[v]$ gilt, folgt, daß $I_u$ und $I_v$ disjunkt sind.

$\square$

**Korollar 3** *$u$ ist genau dann ein Ahne von $v$ im DFS-Wald, wenn $I_v \subset I_u$ gilt.*

**Beweis** Die Aussage folgt unmittelbar aus dem Klammersatz. □

**Lemma 10** *(Lemma vom weißen Pfad) Im DFS-Wald ist $u$ genau dann ein Ahne von $v$, wenn es zur Entdeckungszeit von $u$ einen $u \to v$ Pfad gibt, der nur aus weißen Knoten besteht.*

**Beweis**

"$\Rightarrow$" Sei $v$ ein Nachfahre von $u$ und $w$ ein beliebiger Knoten auf dem Pfad zwischen $u$ und $v$ im DFS-Baum, so daß $w$ ein Nachfahre von $u$ ist. Nach dem vorigen Korollar gilt $d[u] < d[w]$ und somit ist $w$ weiß zur Entdeckungszeit $d[u]$ von $u$.

"$\Leftarrow$" Angenommen, Knoten $v$ ist zur Zeit $d[u]$ von $u$ aus entlang eines Pfades von weißen Knoten erreichbar, aber $v$ wird kein Nachfahre von $u$ im DFS-Baum. Ohne Einschränkung kann man annehmen, daß jeder andere Knoten auf dem Pfad ein Nachfahre von $u$ wird (sonst sei $v$ der nächste Knoten zu $u$, der nicht Nachfahre von $u$ wird). Sei $w$ der Vorgänger von $v$ auf dem Pfad ($w$ und $u$ können auch identisch sein). Dann ist $w$ ein Nachfahre von $u$ im DFS-Baum und nach dem obigen Korollar gilt $f[w] < f[u]$. Dann gilt, daß $v$ entdeckt werden muß nachdem $u$ entdeckt wurde, aber bevor $w$ fertig entdeckt ist. Somit gilt dann $d[u] < d[v] < f[w] \leq f[u]$. Der Klammersatz impliziert, daß das Intervall $[d[v], f[v]]$ vollständig in dem Intervall $[d[u], f[u]]$ enthalten sein muß. Daraus folgt, daß $v$ ein Nachfahre von $u$ ist.

□

Kantenklassifikation durch DFS:
Die Kanten eines Graphen werden durch eine Tiefensuche in disjunkte Mengen aufgeteilt:

**Baumkanten** sind Kanten im DFS-Wald von $G$. Eine Kante $[u, v\rangle$ ist eine Baumkante, wenn der Knoten $v$ zu dem Zeitpunkt, an dem man die Kante $[u, v\rangle$ betrachtet, entdeckt wird. Für diese Kanten gilt $f[u] > f[v]$.

**Vorwärtskanten** sind die Kanten $[u, v\rangle$, die keine Baumkanten sind, und für die gilt, daß $u$ ein Nachfahre von $v$ im DFS-Baum ist. Auch in diesem Fall gilt $f[u] > f[v]$.

**Rückwärtskanten** sind die Kanten $[u, v\rangle$, die einen Knoten $u$ mit einem seiner Vorfahren $v$ im DFS-Wald verbinden. Für diese Kanten gilt $f[u] < f[v]$.

**Querkanten** sind alle übrigen Kanten. Sie können zwischen zwei Knoten im selben Baum verlaufen, falls der eine nicht Vorgänger vom anderen ist und umgekehrt oder zwischen Knoten in verschiedenen Bäumen. Für solch eine Kante $[u, v\rangle$ gilt $f[u] > f[v]$.

**Satz 17** *Wenn DFS auf einen ungerichteten Graphen angewandt wird, dann gibt es keine Querkanten.*

## 7.5 Anwendungen von DFS

Obwohl Tiefensuche sehr einfach ist, hat sie viele Anwendungen. In ungerichteten Graphen kann man durch sie zum Beispiel zusammenhängende Komponenten, zweifach zusammenhängende Komponenten, oder auch sogenannte "Brücken" effizient berechnen, oder sogar bestimmen, ob ein Graph ohne Kantenüberkreuzungen in die Ebene einbettbar ist (siehe die Bücher von Mehlhorn und von Aho, Hopcroft, Ullman).
In gerichteten Graphen kann man Tiefensuche verwenden, um zu testen ob der Graph zykelfrei ist, und gegebenenfalls auch topologisch sortieren. Man nimmt dabei die aus DFS stammenden $f[u]$'s als "verkehrte" Sortierung (Sortierung in umgekehrter Reihenfolge). Dies ist möglich, da es wegen der Zykelfreiheit von $G$ keine Rückwärtskanten im DFS-Wald gibt.
Wir wenden uns jetzt einer besonders eleganten Anwendung von Tiefensuche zu.

### 7.5.1 Stark zusammenhängende Komponenten

Wir betrachten das Problem der Zerlegung eines gerichteten Graphen in seine stark zusammenhängenden Komponenten (SZ-Komponeneten).

**Definition 11** *Eine* stark zusammenhängende Komponente (SZ-Komponente) *eines gerichteten Graphen $G = (V, E)$ ist eine maximale Menge von Knoten $U \subset V$, so daß es für jedes Knotenpaar $(u, v) \in U$ sowohl einen Pfad von $u$ nach $v$ als auch einen von $v$ nach $u$ gibt.*

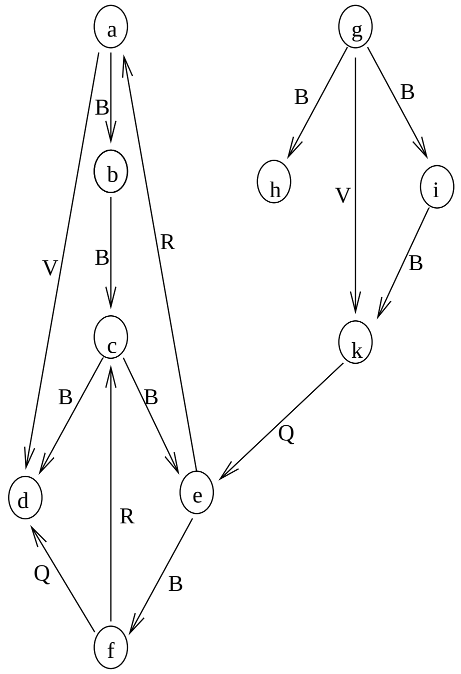



Abbildung 7.1: B=Baum-, V=Vorwärts-, R=Rückwärts-, Q=Querkante; die Knoten werden vom DFS-Algorithmus in lexikographischer Ordnung abgearbeitet

Anders ausgedrückt: Definiert man für einen gerichteten Graphen $G = (V, E)$ eine eine Äquivalenzrelation auf der Knotenmenge $V$ durch

$$u \sim_G v \iff u \rightsquigarrow_G v \text{ und } v \rightsquigarrow_G u$$

wobei $u \rightsquigarrow_G v$ bedeutet, daß es einen Pfad von $u$ nach $v$, dann sind die Äquivalenzklassen die stark zusammenhängenden Komponenten.
Die SZ-Komponenten im angegebenen Beispiel bestehen aus den Mengen $\{a, b, c, h\}$, $\{d, e, f\}$, $\{i\}$ und $\{g\}$.
Im folgenden bezeichne $G^t = (V, E^t)$ den zu $G = (V, E)$ *inversen Graphen*, der durch Umkehrung aller Kanten entsteht, also $E^t = \{[u, v\rangle \mid [v, u\rangle \in E\}$.
Der folgende einfache Algorithmus bestimmt die SZ-Komponenten eines Graphen $G = (V, E)$:

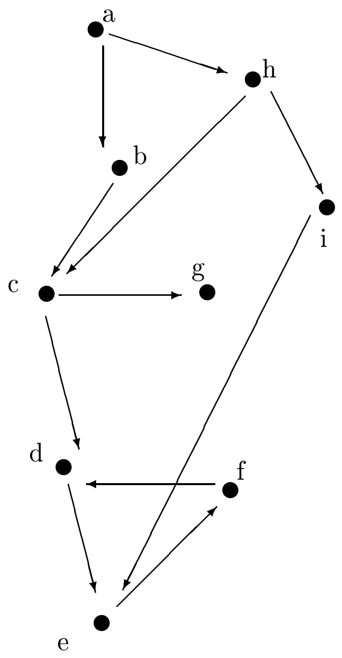
SZK($G$)
1. Verwende DFS($G$), um $f[u]$ für alle Knoten $u \in V$ zu bestimmen.
2. Berechne den zu $G$ inversen Graphen $G^t$.
3. Exekutiere DFS($G^t$), aber betrachte in der Hauptschleife die Knoten in fallender $f[u]$-Ordnung.
4. Die Bäume des DFS-Waldes aus Schritt 3 bilden die SZ-Komponenten von $G$.

Der Algorithmus besteht aus dem zweimaligen Durchführens einer Tiefensuche, sowie dem Berechnen von $G^t$. Man kann sich leicht davon überzeugen, daß jeder der drei Schritte in linearer Zeit $O(|V| + |E|)$ durchgeführt werden kann. Die Gesamtzeit für den Algorithmus ist also auch $O(|V| + |E|)$.
Es bleibt nun noch zu zeigen, daß der Algorithmus auch wirklich die SZ-Komponenten berechnet:

**Behauptung 21** *Wenn die Knoten $u$ und $v$ in derselben SZ-Komponente $C$ liegen, dann verläßt kein $u \to v$ Pfad die Komponenete $C$.*

**Beweis** Seien $u$ und $v$ zwei Knoten in derselben SZ-Komponente. Es existiert also ein Pfad von $u$ nach $v$ und umgekehrt. Sei $w$ ein beliebiger Knoten auf einem $u \to v$ Pfad. Damit existiert ein Pfad von $u$ nach $w$. Umgekehrt ist aber auch $u$ von $w$ aus erreichbar über den Pfad $w \rightsquigarrow_G v \rightsquigarrow_G u$. Also liegen $u$ und $w$ in derselben SZ-Komponenete. $\square$

**Behauptung 22** *Bei einer Tiefensuche liegen alle Knoten derselben SZ-Komponente in einem Baum des DFS-Waldes.*




**Beweis** Von den Knoten einer SZ-Komponente sei $u$ derjenige, der zuerst entdeckt wird. Zu diesem Zeitpunkt sind alle anderen Knoten der Komponente noch weiß. Es gibt Pfade von $u$ zu jedem anderen Knoten in der SZ-Komponente und da diese Pfade, wie gerade gezeigt, die SZ-Komponente nicht verlassen, sind auch die Knoten auf diesen Pfaden noch weiß. Aufgrund des weißen Pfad-Lemmas ist $u$ ein Ahne von allen anderen Knoten in der SZ-Komponenete und damit liegen all diese Knoten im selben DFS-Baum. $\square$

Im folgenden beziehen alle $d[]$ und $f[]$ Werte auf jene, die durch die Tiefensuche in $G$ in Schritt 1 berechnet wurden.

**Definition 12** *Der Vorvater $\varphi(u)$ von $u$ ist der Knoten $w$, so daß es einen Pfad in $G$ von $u$ nach $w$, $u \rightsquigarrow_G w$, gibt und $f[w]$ maximal ist.*
*Der Vorvater eines Knotens $u$ ist also der Knoten $w$, der von $u$ erreichbar ist und als letzter in der Tiefensuche vollkommen entdeckt wurde.*

**Behauptung 23** $f[u] \leq f[\varphi(u)]$.

**Beweis** Knoten $u$ ist immer trivialerweise von sich selbst aus erreichbar. $\square$

**Behauptung 24** *Wenn es einen Pfad von $u$ nach $v$ gibt, dann gilt $f[\varphi(v)] \leq f[\varphi(u)]$.*

**Beweis** Da $\{w : \text{es existiert ein Pfad von } v \text{ nach } w\ \} \subset \{w : \text{ es existiert ein Pfad von } u \text{ nach } w\ \}$ gilt und der Vorvater die maximale Endzeit aller erreichbaren Knoten hat, gilt die Behauptung. $\square$

Aus der gerade gezeigten Behauptung folgt, daß jede SZ-Komponente einen Knoten hat, der Vorvater von jedem anderen Knoten in der SZ-Komponente ist. Dieser Vorvater ist ein repräsentativer Knoten für die SZ-Komponente.

**Behauptung 25** $\varphi(\varphi(u)) = \varphi(u)$

**Beweis** : Nach Definition existiert ein Pfad vom Vorvater von $u$ zum Vorvater vom Vorvater von $u$: $\varphi(u) \rightsquigarrow_G \varphi(\varphi(u))$.
Aus Behauptung 24 folgt: $f[\varphi(\varphi(u))] \leq f[\varphi(u)]$.
Aus Behauptung 23 folgt: $f[\varphi(\varphi(u))] \geq f[\varphi(u)]$.
Daraus folgt Gleichheit. $\square$

**Behauptung 26** *$\varphi(u)$ ist ein Ahne von $u$ im DFS-Wald.*

**Beweis** Wenn $\varphi(u) = u$ ist, so gilt die Behauptung trivialerweise. Für $\varphi(u) \neq u$ betrachte die Färbung von $\varphi(u)$ zur Entdeckungszeit $d[u]$ von $u$.

1. Wenn $\varphi(u)$ schwarz gefärbt ist, so gilt $f[\varphi(u)] < f[u]$ und dies ist ein Widerspruch Behauptung 23. Dieser Fall tritt also nicht ein.
2. Wenn der Knoten $\varphi(u)$ grau gefärbt ist, dann ist er ein Ahne von $u$ und die Behauptung ist gezeigt.
3. Es bleibt noch zu zeigen, daß $\varphi(u)$ nicht weiß gefärbt sein kann. Hier unterscheiden wir zwei Fälle:
   (a) Es gibt nur weiße Knoten auf dem Pfad von $u$ nach $\varphi(u)$. Aufgrund des weißen Pfad-Lemma gilt dann, daß $u$ ein Ahne von $\varphi(u)$ ist, also $f[u] > f[\varphi(u)]$. Widerspruch.
   (b) Es gibt Knoten auf dem Pfad von $u$ nach $\varphi(u)$, die nicht weiß sind. Sei $w$ der letzte dieser nicht weißen Knoten. $w$ ist grau gefärbt, da es keine Kanten zwischen einem schwarz und einem weiß gefärbten Knoten gibt und der Nachfolger von $w$ weiß ist. Es existiert also ein Pfad mit weißen Knoten von $w$ nach $\varphi(u)$, und nach dem Lemma vom weißen Pfad ist damit $w$ ein Ahne von $\varphi(u)$. Also gilt $f[w] > f[\varphi(u)]$, im Widerspruch zur Wahl von $u$.

$\square$

**Behauptung 27** *$u$ und $\varphi(u)$ liegen in derselben SZ-Komponente.*

**Beweis** Nach Definition vom Vorvater gibt es einen Pfad von $u$ nach $\varphi(u)$ und aus Behauptung 26 folgt, daß es auch einen Pfad von $\varphi(u)$ nach $u$ gibt, nämlich den Pfad im DFS-Baum. Also liegen $u$ und $\varphi(u)$ in derselben SZ-Komponente. $\square$

**Behauptung 28** *Die Knoten $u$ und $v$ liegen genau dann in derselben SZ-Komponente, wenn $\varphi(u) = \varphi(v)$ gilt.*

**Beweis**

"$\Longrightarrow$" Seien $u$ und $v$ in derselben SZ-Komponente. Jeder von $u$ erreichbare Knoten ist also auch erreichbar von $v$ und umgekehrt. Nach der Definition vom Vorvater schließen wir, daß $\varphi(u) = \varphi(v)$ ist.

"$\Longleftarrow$" Sei $\varphi(u) = \varphi(v)$. Nach Behauptung 27 liegt $u$ in derselben SZ-Komponente wie $\varphi(u)$ und $v$ in derselben SZ-Komponente wie $\varphi(v)$. Deshalb liegen $u$ und $v$ in derselben SZ-Komponente.

$\square$

**Behauptung 29** *Es sei jedem Knoten $v$ eines Graphens $G = (V, E)$ eine eindeutige Zahl $f[v]$ zugeordnet. Nehmen wir an, $\mathrm{DFS}(G^t)$ wird durchgeführt, wobei in der Haupschleife die Knoten in Ordnung fallender $f[]$-Werte abgearbeitet werden.*
*Sei $B$ ein Baum im sich ergebenden DFS-Wald und sei $v$ seine Wurzel. Der Baum $B$ enthält jene Knoten $u$, für die $v$ der in $G$ erreichbare Knoten mit maximalen $f[]$-Wert ist.*

**Beweis** Wenn in $\mathrm{DFS}(G^t)$ ein neuer Baum mit Wurzel $v$ begonnen wird, dann enthält dieser Baum alle Knoten die in $G^t$ von $v$ erreichbar sind, aber von keinem Knoten mit $f[]$-Wert größer als $f[v]$ erreichbar sind. $\square$

**Satz 18** *Der Algorithmus $\mathrm{SZK}(G)$ berechnet korrekt die SZ-Komponenten von $G$ und tut dies in linearer Zeit.*

**Beweis** Korrektheit folgt aus der Definition des Vorvaters $\varphi(v)$ sowie Behauptungen 28 und 29. Die lineare Zeitschranke ist klar. $\square$

Vorlesung vom 18.1.1996
Tanja Maurer, Pascal Schüler, Christian Fink

## 7.6 Kürzeste Wege in Graphen

In diesem Abschnitt wollen wir uns mit der Frage beschäftigen, wie man in einem kantengewichteten Graphen kürzeste Pfade, d.h. Pfade mit minimalem Gewicht, zwischen Knoten finden kann. Was ist das Gewicht eines Pfades?

**Definition 13** *Sei $G = (V, E)$ ein gerichteter Graph und $w : E \to \mathbb{R}$ eine Gewichts-/Längenfunktion auf der Menge der Kanten. Wenn $p = \langle v_0, v_1, \ldots, v_k\rangle$ ein Pfad von $v_0$ nach $v_k$ ist, so ist das Gewicht des Pfades $p$ definiert als*

$$w(\langle v_0, v_1, ..., v_k\rangle) = \sum_{0<i\leq k} w(v_{i-1}, v_i)\,.$$

*Die Distanz $\delta : V \times V \to \mathbb{R}$ von Knoten $u$ nach Knoten $v$ wird definiert durch*

$$\delta(u,v) := \begin{cases} \min\{\ w(p)\ |p \textit{ ist } u \rightsquigarrow v \textit{ Pfad}\} & \textit{falls ein } u \rightsquigarrow v \textit{ Pfad existiert} \\ \infty & \textit{sonst.} \end{cases}$$

*Ein* kürzester Pfad *von einem Knoten $u$ zu einem Knoten $v$ ist ein Pfad $p$ mit $w(p) = \delta(u, v)$.*

**Beispiel 1:**

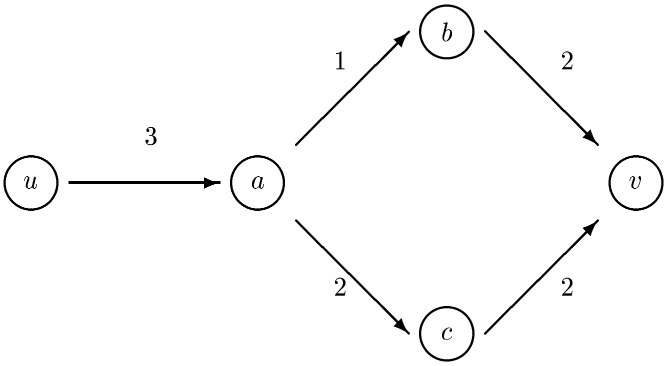



Abbildung 7.2: Die Distanz von $u$ nach $v$ ist $\delta(u, v) = 3 + 1 + 2 = 6$. Ein kürzester Pfad von $u$ nach $v$ ist $\langle u, a, b, v\rangle$.

Probleme treten bei Graphen mit Zyklen negativer Läenge auf:
**Beispiel 2:** Wählt man im Graphen vom Beispiel 2 den Weg $p = \langle u, a, b, c, v\rangle$ von $u$ nach $v$, so hat man eine

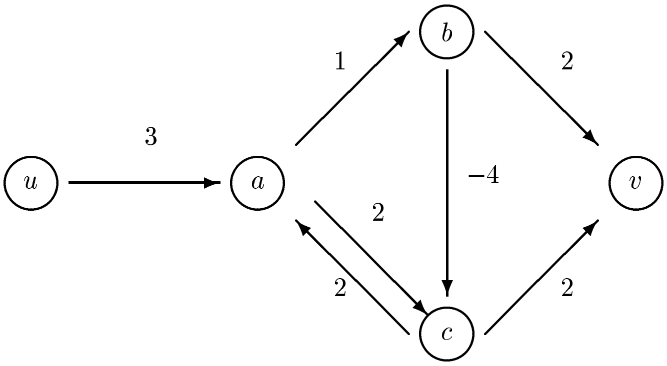



Abbildung 7.3: Graph mit Zyklus $a, b, c, a$ von negativer Länge $-1$.

Distanz von 2. Durch Wahl des Weges $\langle u(abc)^n v\rangle$ bzw. $\langle u(abc)^n abv\rangle$ könnte man die Distanz von $u$ nach $v$ aber

beliebig verkleinern, da diese beiden Wege den negativen Zyklus $\langle abca \rangle$ mit $w(\langle a,b,c,a \rangle) = 1 - 4 + 2 = -1$ enthalten.
Wir wollen uns mit diesem Phänomen nicht weiter auseinandersetzen und machen von nun an die Annahme, daß die Graphen keine negativen Zyklen haben, bzw. wenn sie einen haben, wollen wir diese Tatsache nur entdecken müssen, aber keine kürzesten Wege berechnen.

Man unterscheidet drei Typen von kürzesten Wege Problemen:

**Single-Source Shortest-Path Problem** Bestimme die Distanzen/kürzesten Wege von einem Knoten $s$ zu allen anderen Knoten des Graphen.

**Single-Pair Shortest-Path Problem** Bestimme die Distanz/kürzesten Weg vom Knoten $s$ zum Knoten $t$.

**All-Pairs Shortest-Path Problem** Bestimme die Distanzen/kürzesten Wege zwischen allen Knoten des Graphen.

Algorithmen für das erste und das letzte Problem lösen automatisch auch das Zweite. Interessanterweise sind keine Algorithmen bekannt, die explizit nur das Zweite, das Single-Pair Problem, lösen ohne nicht auch eigentlich das einen Single-Source Algorithmus zu lösen.
Zuerst behandeln wir ganz kurz ein paar wesentliche Eigenschaften von kürzesten Wegen und Distanzen.

**Behauptung 30** *Wenn* $p = < v_0, v_1, ..., v_k >$ *ein kürzester* $v_0 \rightsquigarrow v_k$ *Pfad ist, dann ist für alle* $0 \leq i \leq j \leq k$ $< v_i, v_{i+1}, ..., v_j >$ *ein kürzester* $v_i \rightsquigarrow v_j$ *Pfad.*

**Beweis** Das Gewicht des Pfades $p$ ist $w(p) = w(v_0, v_1) + w(v_1, v_2) + w(v_{k-1}, v_k)$. Angenommen, es gibt zwei Knoten $v_i$ und $v_j$ auf diesem Pfad, für die es einen kürzeren Pfad $p'_{ij}$ gibt als $p_{ij} = (\langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle)$ mit $w(p'_{ij}) < w(p_{ij})$. Ersetze dann im Pfad $p$ den Teilpfad $p_{ij}$ durch $p'_{i,j}$. Für diesen modifizierten Pfad $p'$ gilt:

$$\begin{aligned} w(p') &= w(\langle v_0, \ldots, v_{i-1}) + w(p'_{i,j}) + w(\langle v_{j+1}, \ldots, v_k) \\ &< w(\langle v_0, \ldots, v_{i-1}) + w(p_{i,j}) + w(\langle v_{j+1}, \ldots, v_k) \\ &= w(p). \end{aligned}$$

Dies ist aber ein Widerspruch zur Annahme, daß $p$ ein kürzester Pfad von $v_0$ nach $v_k$ ist. ☐

**Lemma 11** *Dreiecksungleichung*
*Für alle Kanten* $[u, v\rangle \in E$ *gilt:*

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

### 7.6.1 Das “Single-Source” Problem

Wir möchten also in einem gewichteten Graphen $G$ von einem Startknoten $s$ aus die Distanzen und kürzesten Wegen zu allen anderen Knoten $v$ des Graphen berechnen. Was die Distanzen betriff, so ist die Ausgabe eines Algorithmus für dieses Problem klar: Für jeden Knoten $v$ wird ein Wert $d[v]$ ausgegeben, der gleich $\delta(s, v)$ sein soll. Was die kürzesten Wege betriff, so ist die verlangte Ausgabe nicht so klar. Denn gäbe der Algorithmus alle $n - 1$ kürzesten Wege explizit aus ($n$ is Anzahl der Knoten in $G$), dann könnte die Gesamtanzahl der Kanten auf diesen Wegen, also die Ausgabegröße, $\Theta(n^2)$ sein. Dies hieße, daß der Algorithmus mindestens Zeit $\Theta(n^2)$ braucht, allein schon um die Ausgabe zu produzieren. Da wir aber, insbesonders wenn $G$ nur wenig Kanten hat, auf schneller Algorithmen hoffen, müssen dazu sehen, daß die Ausgabe anders organisiert ist. Dies macht man mit Hilfe eines sogenannten “Kürzesten Wege Baums”:

**Definition 14** *Für alle Knoten* $v \neq s$ *des Graphen sei* $\pi[v]$ *der vorletzte Knoten auf einem kürzesten* $s \rightsquigarrow v$ *Pfad;* $\pi[s] = NIL$*. Die Kanten* $[\pi[v], v]$ *bilden also einen Baum mit Wurzel* $s$*. Mit* $\Pi[v]$ *wird der kürzeste Weg von* $s$ *nach* $v$ *in diesem Baum bezeichnet.*

Das Ziel eines Algorithmus für das single-source kürzeste Wege Problem wird es also sein, für jeden Knoten $v$ die Distanz $d[v]$ und den Vorgängerknoten $\pi[v]$ zu berechnen. Mit Hilfe der Vorgängerknoten kann dann der explizite kürzester Pfad $\Pi[v]$ von $s$ nach $v$ in Zeit proportional zu seiner Kantenanzahl leicht berechnet werden. Die Algorithmen, die wir behandeln, beruhen alle auf der sukzessiven Verbesserung von $d[]$-Werten und das sich daraus ergebende Neusetzen der $\pi[]$-Werte. Sie nehmen an, daß diese Werte vernünftig initialisiert wurden, nämlich so:

```
INITIALIZE(G, s)
    for each vertex v do
        d[v] = ∞
        π[v] = NIL
    d[s] = 0
```

Das Verbessern von berechneten $d[]$-Werten geschieht durch sogenannte *Relaxation*. Die Relaxation einer Kante $[u,v\rangle$ besteht darin, daß man testet, ob man den bis zum jetzigen Zeitpunkt gefundenen kürzesten Pfad von $s$ nach $v$ verbessern kann, indem man zuerst von $s$ nach $u$ geht und dann die direkte Kante zwischen $u$ und $v$ benutzt.

```
RELAX(u, v)
    if  d[u] + w(u, v) < d[v] then d[v] = d[u] + w(u, v)
                                   π[v] = u
```

**Lemma 12** *Die folgende* Invariante *wird durch* RELAX$(u,v)$ *aufrecht erhalten:*

1. *Für alle Knoten $w$ gilt: $d[w] \geq \delta(s,w)$.*
2. *Für alle Knoten $w$ gilt: Aus $d[w] = \delta(u,w)$ folgt, daß $\Pi[w]$ ein kürzester $s \rightsquigarrow w$ Weg ist.*

**Beweis**

1. Die Invariante gilt vor Ausführung der RELAX$(u,v)$-Operation für alle Knoten. Nach der Ausführung hat sich für alle Knoten $w \neq v$ der Wert $d[w]$ nicht verändert. Für den Knoten $v$ hat sich entweder der Wert $d[v]$ ebenfalls nicht verändert, oder es gilt:
$$d[v] = \underbrace{d[u]}_{\geq \delta(s,u)} + w(u,v) \geq \delta(s,v)\,,$$
wegen der Dreiecksungleichung. Die Invariante bleibt also erhalten.

2. Die Invariante gilt wieder vor Ausführung der RELAX$(u,v)$-Operation für alle Knoten. Der einzige $\pi[]$-Wert, der sich verändern kann, ist $\pi[v]$, und der ändert sich nur, wenn das $d[v]$ verkleinert wird. Verkleinerung von $d[v]$ bedeutet nach Invariante 1, daß vor der Operation $d[v] > \delta(s,v)$ gegolten hat. Also war $\Pi[v]$ kein kürzester Weg, und $v$ konnte auch für keinen Knoten $w \neq v$ auf $\Pi[w]$ auftreten, wenn $\Pi[w]$ ein kürzester Pfad war. Das bedeutet, daß das Neusetzen von $\pi[v]$ Invariante 2 für alle $w \neq v$ unbehelligt läßt.

   Was $v$ selbst betrifft, ist es leicht zu sehen, daß, falls durch die Ausführung von RELAX$(u,v)$ der Wert $d[v]$ sich auf $\delta(s,v)$ verringert hat, schon vorher $d[u] = \delta(s,u)$ gegolten haben muß. Nach Invariante 2 galt da also schon, daß $\Pi[u]$ ein kürzester $s \rightsquigarrow u$ Weg war, und mit dem Setzen $\pi[v] = u$ wird jetzt $\Pi[v]$ ein kürzester $s \rightsquigarrow v$ Weg. Die Invariante 2 bleibt also auch für $v$ erhalten.

□

**Lemma 13** *(Hauptlemma)*
*Sei $s = \langle v_0, v_1, \ldots, v_k\rangle$ ein kürzester Weg von $s$ nach $v_k$. Nimm an, daß nach* INITIALIZE$(G,s)$ *eine Folge von* RELAX$(x_i, y_i)$, $i = 1, 2, \ldots$, *durchgeführt wird, in der* RELAX$(v_0, v_1)$, RELAX$(v_1, v_2)$, $\ldots$, RELAX$(v_{k-1}, v_k)$ *als Teilfolge in dieser Reihenfolge vorkommt. Dann gilt für $j = 0, 1, \ldots, k$: $d[v_j] = \delta(s, v_j)$ und $\Pi[v_j]$ ist ein kürzester $s \rightsquigarrow v_j$ Pfad.*

**Beweis** Induktion über $k$. □

Da es immer einen kürzesten $s \rightsquigarrow v$ Weg gibt, der aus höchstens $|V| - 1$ Kanten besteht (sonst gibt es einen negativen Zyklus), ergibt sich aus diesem Hauptlemma sofort ein Algorithmus, um die kürzesten Pfade von einem Startknoten $s$ zu allen anderen Knoten zu ermitteln. Dieser Algorithmus stammt von Bellmann und Ford.

BELLMANN-FORD-ALGORITHMUS
  INITIALIZE$(G, s)$
  **for** $i = 1$ **to** $|V| - 1$ **do**
    **for each** $[u, v\rangle \in E$ **do** RELAX$(u, v)$
  **for each** $[u, v\rangle \in E$ **do**
    **if** $d[v] > d[u] + w(u, v)$ **then**
      **return** Negativer Zyklus

Die letzte For-Schleife dient dabei als Test, ob der Graph einen negativen Zyklus enthält. Die Laufzeit dieses Algorithmus ist klarerweise $O(|V| \cdot |E|)$.
Interessanterweise ist für den allgemeinen Fall, in dem negative Kantengewichte erlaubt sind, kein anderer Algorithmus bekannt mit wesentlich besserer Laufzeit im schlechtesten Fall. Für den Fall, daß alle Kantengewichte $\geq 0$ sind, gibt es allerdings einen wesentlich schnelleren Algorithmus, der nach Dijkstra benannt ist. Er führt für jede Kante des Graphens nur eine RELAX Operation durch. Dies wird erreicht, indem RELAX$(u, v)$ erst durchgeführt wird, nachdem $d[u]$ schon auf $\delta(s, u)$ gesetzt wurde. Dies zu Erkennen ist wiederum möglich, indem man Knoten nach wachsenden $d[]$-Wert bearbeitet. Zu diesem Zweck wird eine Prioritätenschlane verwendet.

DIJKSTRA$(G, s)$
(1) INITIALIZE$(G, s)$
(2) $Q =$ EMPTY-PQ
(3) * Prioritätenschlange für Knoten, Schlüssel $d[v]$ *
(4) **for each** $v \in V$ **do** PQ-INSERT$(v, Q)$
(5) **while** $Q \neq \emptyset$ **do**
(6)   $u =$ EXTRACTMIN$(Q)$
(7)   **for each** $v \in OUT(u)$ **do** RELAX$(u, v)$

Um die Prioritätenschlange günstig verwenden zu können, ändern wir den RELAX-Algorithmus leicht ab:

RELAX$(u, v)$
  **if** $d[u] + w(u, v) < d[v]$ **then**
    $d[v] = d[u] + w(u, v)$
    DECREASE-KEY$(v, d[v])$
    $\pi[v] = u$

**Laufzeitanalyse:**

| | |
|---|---|
| (1) | $O(n)$ |
| (2) | $O(1)$ |
| (4) | $n$ INSERTS |
| (6) | $n$ EXTRACT-MINS |
| (7) | $\lvert E\rvert \cdot (O(1)+$ Zeit für DECREASE-KEY) |

Die Laufzeit hängt also wesentlich davon ab, wie die Prioritätenschlange $Q$ implementiert ist. Verwaltet man $Q$ nur naiv als Menge, braucht EXTRACT-MIN lineare Zeit, die anderen Operationen konstante Zeit. Damit ergibt sich für den Dijkstra-Algorithmus eine Laufzeit von $O(|V|^2)$.
Verwendet man für $Q$ einen Binary Heap oder einen Binomial Heap, dann braucht jede der Prioritätenschlangenoperationen logarithmische Zeit. Damit erreicht der Dijkstra-Algorithmus eine Laufzeit von $O((|V| + |E|) \log |V|)$.
Noch besser wir die Laufzeit bei Verwendung eines Fibonacci-Heaps für $Q$. Dieser erreicht ja logarithmische Laufzeit für EXTRACT-MIN und konstante Laufzeit für die beiden anderen Operationen (jeweils amortisiert). Dies ergibt eine Laufzeit von $O(|E| + |V| \log |V|)$ für Dijkstra's Algorithmus. Es sei hier bemerkt, daß genau diese Anwendung eine der Hauptmotivationen für die Erfindung von Fibonacci-Heaps war.
Es bleibt nur noch zu zeigen, daß der Algorithmus von Dijkstra auch wirklich das Single Source Shortest-Path Problem korrekt loest.

**Satz 19** *Wendet man den Algorithmus von Dijkstra auf einen gerichteten Graphen $G = (V, E)$, dessen Kanten nicht-negative Gewichte haben, und einen Startknoten $s \in V$ an, dann gilt nach Beendigung des Algorithmus für alle Knoten $v \in V$, daß $d[v] = \delta(s, v)$ und daß $\Pi[v]$ ein kürzester $s \rightsquigarrow v$ Weg ist.*

**Beweis** Nach Lemma 12 genügt es zu zeigen, daß $d[v] = \delta(s, v)$ für alle Knoten $v \in V$.
Wir zeigen, daß für jeden Knoten $v \in V$ ab dem Zeitpunkt, an dem $v$ aus der Queue $Q$ entfernt wir, $d[v] = \delta(s, v)$ gilt.
Angenommen, es gibt einen Knoten $v$, für den $d[v] \neq \delta(s, v)$ zu dem Zeitpunkt ist, wenn $v$ aus $Q$ entfernt wird. Sei $v$ der erste Knoten, für den das eintritt. Aus $d[s] = \delta(s, s)$ folgt $v \neq s$ und damit $V \backslash Q =: S \neq \emptyset$. Außerdem existiert ein Pfad von $s$ nach $v$, denn sonst würde $d[v] = \delta(s, v) = \infty$ gelten im Widerspruch zur Voraussetzung. Insbesondere existiert dann auch ein kürzester Pfad $p$. Sei $y$ der erste Knoten auf dem Pfad $p$, der nicht mehr in $S$ liegt und sei $x$ sein direkter Vorgänger, also $x \in S$. Da $v$ der erste Knoten in $S$ ist, für den nicht $d[v] = \delta(s, v)$ gilt, folgt $d[x] = \delta(s, x)$. Zu dem Zeitpunkt, an dem $x$ aus $Q$ entfernt wird, werden alle benachbarten Kanten relaxiert, also insbesondere $y$ und damit ist $d[y] = \delta(s, y)$ nach Behauptung 30. Da $y$ vor $v$ auf dem kürzesten Pfad von $s$ nach $v$ liegt, gilt $\delta(s, y) \leq \delta(s, u)$ und damit

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, v) \\ &\leq d[v]. \end{aligned}$$

Da aber sowohl $u$ als auch $v$ zu dem Zeitpunkt, als $v$ gewählt wurde, inn $Q$ lagen, ist auch $d[v] \leq d[u]$ und daraus folgt die Gleichheit:

$$d[y] = \delta(s, y) = \delta(s, v) = d[v].$$

Insbesondere folgt also $d[v] = \delta(s, v)$ im Widerspruch zur Voraussetzung.
Für jeden Knoten $v \in V$ gilt also zu dem Zeitpunkt, an dem er aus $Q$ entfernt wird, $d[v] = \delta(s, v)$. Da an dem $d$-Wert nachher nichts mehr geändert wird, gilt die Aussage also auch nach Beendigung des Algorithmus. $\Box$

Vorlesung vom 22.1.1996
Thomas Martin, Peter Noss

## 7.7 Das ALL-PAIRS kürzeste Wege Problem

In diesem Abschnitt möchten wir für jedes Knotenpaar $u, v$ in einem gerichteten, kantengewichteten Graphen $G = (V, E)$ die Distanz $\delta(u, v)$ sowie einen kürzesten $u \leadsto v$ Weg bestimmen. Man könnte dies auf einfache Art und Weise implementieren, indem man für jeden Knoten das Single-Source Problem löst. Die Laufzeit wäre dann ($n = |V|$ und $e = |E|$):

- mit dem Bellmann-Ford-Algorithmus: $O(n^2 \cdot e)$
- mit dem Algorithmus von Dijkstra (falls keine negativen Kantengewichte): $O(n^2 \cdot \log n + n \cdot e)$

Wir werden im folgenden versuchen, diese Laufzeit zu verbessern.

### 7.7.1 Floyd-Warshall-Algorithmus

Sei $V = \{1, ..., n\}$ die Knotenmenge des Graphen $G$. Wir nehmen an, daß beliebige Kantengewichte erlaubt sind, aber keine negativen Zyklen.
Betrachten wir vorerst nur das Berechnen der Distanzen. Sei nun $D^{(k)}$ eine $n \times n$-Matrix, wobei $d_{ij}^{(k)}$ die Länge eines kürzesten $i \leadsto j$ Pfades angibt, der nur interne Knoten kleiner gleich $k$ verwendet. Wenn man $D^{(n)}$ konstruiert hat, so hat man das Problem gelöst.
Beachte:

- $d_{ij}^{(0)} = w(i, j)$, da der Pfad nur interne Knoten kleiner gleich 0, also keine internen Knoten, verwenden darf.
- Für $d_{ij}^{(k)}$, $k > 0$, können zwei Fälle auftreten:
  1. Der kürzeste Pfad läuft über den Knoten $k$. Dann läuft der Pfad zuerst von $i$ zu $k$ und dann von $k$ zu $j$. Beide Teilpfade haben nur interne Knoten kleiner gleich $k - 1$.
  2. $k$ ist nicht im kürzesten Pfad. Es werden also nur interne Knoten kleiner gleich $k - 1$ verwendet.

Also gilt für $d_{ij}^{(k)}$:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}.$$

Man sieht also, daß man Matrix $D^{(k)}$ leicht aus Matrix $D^{(k-1)}$ berechnen kann. Insgesamt ergibt dies den folgenden Algorithmus, der Floyd und Warshall zugeschrieben wird.
Sei $W$ die Matrix, in der die Kantengewichte des Graphen gespeichert sind ($w_{ij} = \infty$, falls es keine Kante zwischen Knoten $i$ und Knoten $j$ gibt). Der Algorithmus sieht nun folgendermaßen aus:

**function** Floyd-Warshall($W$)
  $D^{(0)} = W$
  **for** $k = 1$ **to** $n$ **do**
    **for** $i = 1$ **to** $n$ **do**
      **for** $j = 1$ **to** $n$ **do**
        $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
  **return** $D^{(n)}$

Es ist offensichtlich, daß der Algorithmus die Laufzeit $O(n^3)$ hat. Damit ist er mindestens so schnell wie $n$ Anwendungen des Bellmann-Ford Algorithmus. Für den Fall der nicht-negativen Kantengewichte ist er, zumindest asymptotisch gesehen, nicht schneller als Dijkstras Algorithmus $n$-mal angewandt. Allerdings ist er offensichtlich um einiges leichter zu implementieren und unte Umständen in Praxis schneller.

Wie muß man den Algorithmus ändern, damit nicht nur Distanzen, sondern auch kürzeste Wege berechnet werden? Wie stellt man die Wege dar? Dies geschieht wieder implizit über Vorgänger. Sei dazu $\Pi$ die Vorgänger-$(n \times n)$-Matrix, d.h. $\pi_{ij}$ ist der Vorgänger von $j$ auf einem kürzesten $i \rightsquigarrow j$ Pfad in $G$. Die Einträge sind also Namen von Knoten und keine Gewichte.
Definiere auch hier $\Pi^{(k)}$ als die $(n \times n)$-Matrix, für die $\pi_{ij}^{(k)}$ der Vorgänger von $j$ auf einem kürzesten $i \rightsquigarrow j$ Pfad, der nur interne Knoten $\leq k$ verwendet, ist.
Mit dieser Definition gilt:

- $\pi_{ij}^{(0)} = \begin{cases} NIL & \text{wenn } w(i,j) = \infty \\ i & \text{sonst} \end{cases}$
- $k > 0: \qquad \pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{wenn } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{sonst.} \end{cases}$

Wieder läßt sich also $\Pi^{(k)}$ leicht aus $\Pi^{(k-1)}$ berechnen. Wir überlassen weitere Details dem Leser.

Wie wir gesehen haben, kann man Graphen durch Matrizen darstellen. Dann kommt man schnell zu Methoden aus der linearen Algebra: man kann beispielsweise bestimmte Eigenschaften von Graphen durch Eigenwerte von Matrizen darstellen. Darauf wolen wir im Moment aber nicht näher eingehen. Allerdings betraxhten wir nun folgenden Zusammenhang zwischen kürzesten Wegen und Matrizenmultiplikation:

### 7.7.2 ALL-PAIRS-kürzeste Wege und Matrizenmultiplikation

Wiederum sei $W^{(\lambda)}$ eine $(n \times n)$-Matrix, wobei $w_{ij}^{(\lambda)}$ die Länge des kürzesten $i \rightsquigarrow j$ Pfades mit $\leq \lambda$ Kanten ist. Das Ziel ist es, $W^{(n-1)}$ zu berechnen. Es gilt:

- $w_{ij}^{(1)} = w(i,j)$ für alle $1 \leq i,j \leq n$,
- Für gegebene $\lambda$, $\mu$ gilt für alle $1 \leq i,j \leq n$: $w_{ij}^{(\lambda+\mu)} = \min\limits_{1 \leq k \leq n}\{w_{ik}^{(\lambda)} + w_{kj}^{(\mu)}\}$. Um $W^{(\lambda+\mu)}$ zu berechnen, benötigt man $O(n^3)$ Zeit.

Um, ausgehend von $W^{(1)}$, $W^{(n-1)}$ zu berechnen, quadrieren wir wiederholt:

$$\begin{array}{lcccl} W^{(2)} & = & W^{(1)} & \otimes & W^{(1)} \\ W^{(4)} & = & W^{(2)} & \otimes & W^{(2)} \\ W^{(8)} & = & W^{(4)} & \otimes & W^{(4)} \\ & \vdots & & & \\ W^{(2^p)} & = & W^{(2^{p-1})} & \otimes & W^{(2^{p-1})}. \end{array}$$

Wir brauchen $2^p \geq n-1$, also $p = \lceil \log_2(n-1) \rceil$ Schritte. Zur Berechnung von $W^{(n-1)}$ benötigt man also $O(n^3 \log n)$ Zeit.
Die reellen Zahlen $\mathbb{R}$ mit der "Addition" min und der "Multiplikation" + bilden einen Semiring. (Ein Semiring ist ein Ring, bei dem die Addition keine Gruppe ist, weil das inverse Element nicht existiert). Die Operation $\otimes$ ist dann die Matrizenmultiplikation über diesem Semiring: $\min\limits_{1\leq k\leq n} \{w_{ik}^{(\lambda)} + w_{kj}^{(\mu)}\}$ statt $\sum\limits_{1\leq k\leq n} (w_{ik}^{(\lambda)} \otimes w_{kj}^{(\mu)})$.

Matrizenmultiplikation in einem Ring kann in $o(n^3)$ Zeit ausgeführt werden. Es ist unklar, ob $O(n^2)$ Zeit ausreicht. Der derzeitige "Rekord" für die Multiplikation von $n \times n$ - Matrizen ist $O(n^{2,376})$ (Coppersmith-Winograd). Der Geschwindigkeitsvorteil kommt dabei aber erst zum tragen, wenn $n$ sehr groß ist.

Hier nun ein Algorithmus, der in $o(n^3)$ Zeit zwei Matrizen multipliziert.
**Algorithmus von Strassen**

$A, B$ seien zwei $n \times n$ - Matrizen. Wir wollen das Produkt $C = A \cdot B$ berechnen. Wir nehmen an, daß $n$ eine Zweierpotenz ist: $n = 2^k$. Wir unterteilen $A$ und $B$ in je vier $(n/2 \times n/2)$-Matrizen.

$$\underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} e & g \\ f & h \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} r & s \\ t & u \end{pmatrix}}_{C}$$

Es gilt:

$$\begin{aligned} r &= a \cdot e + b \cdot f \\ s &= a \cdot g + b \cdot h \\ t &= c \cdot e + d \cdot f \\ u &= c \cdot g + d \cdot h \end{aligned}$$

Ein "einfacher" Algorithmus berechnet die acht Produkte rekursiv und addiert anschließend die Ergebnisse. Die Laufzeit ist dann:

$$T(n) = 8 \cdot T(\frac{n}{2}) + O(n^2) = O(n^{\log_2 8}) = O(n^3).$$

Will man das noch verbessern, so muß man einen Weg finden, weniger als acht Matrizenmultiplikationen durchzuführen. Der Agorithmus von Strassen kommt mit sieben Multiplikationen aus und benötigt daher nur $O(n^{\log_2} 7) = O(n^{2,81})$ Zeit.
Zuerst werden folgende Summen gebildet:

$$\begin{array}{lll@{\qquad}lll} A_1 &=& a & B_1 &=& g-h \\ A_2 &=& a+b & B_2 &=& h \\ A_3 &=& c+d & B_3 &=& e \\ A_4 &=& d & B_4 &=& f-e \\ A_5 &=& a+d & B_5 &=& e+h \\ A_6 &=& b-d & B_6 &=& f+h \\ A_7 &=& a-c & B_7 &=& e+g. \end{array}$$

Die einzelnen $A_i$'s und $B_i$'s werden multipliziert:

$$P_i = A_i \cdot B_i \qquad \text{für } i = 1, ..., 7.$$

Damit gilt:

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7. \end{aligned}$$

(Dies kann man durch einfaches Ausmultiplizieren nachrechnen.)

Damit haben wir einen Algorithmus, der nur sieben Multiplikationen benötigt. Durch geschickte Implementierung kann man noch vermeiden, daß man immer alle Matrizen gleichzeitig im Speicher halten muß, so daß sich auch der Speicherbedarf in Grenzen hält.

### 7.7.3 ALL-PAIRS-Distanz für ungewichtete, ungerichtete Graphen

Es spricht einiges dafür, daß Matrizenmultiplikation und All-Pairs Kürzeste-Wege sehr ähnliche Probleme sind, die auch eine ähnliche Laufzeit haben sollten. Für den allgemeinen Fall kann man das bis jetzt jedoch noch nicht beweisen. Wir zeigen jetzt, daß für einen sehr speziellen Fall ähnliche Laufzeiten tatsächlich erreicht werden können. Es handelt sich dabei um den Fall von *ungewichteten und ungerichteten Graphen.*
Wir beschränken uns zunächst auf die Distanzversion, also dem Bestimmen nur der*Längen* der kürzesten Pfade. Der nächste Abschnitt behandelt das Bestimmen der Pfade selbst.
Für ungerichtete, ungewichtete Graphen kann nan das *Single-Source Problem* durch *Breitensuche* lösen. Wenn man diese für jeden Knoten ausführt hat man das All-Pairs-Distanz Problem gelöst, und zwar mit einer Gesamtlaufzeit von $O(n \cdot (n+e))$. Dabei ist $n$ die Anzahl der Knoten im Graphen und $e$ die Anzahl der Kanten. Wir wollen nun einen Algorithmus zeigen, dessen Laufzeit $O(M(n) \log n)$ ist. Dabei ist $M(n)$ die Zeit, die zur Multiplikation zweier $n \times n$ Matrizen notwendig ist.
Sei $A$ die Adjazenz-Matrix des Graphen, also $a_{ij} = 1$ genau dann, wenn $i$ und $j$ benachbart sind. Sei $grad_A(j)$ der Grad des Knoten $j$ in $G$, also die Anzahl seiner Nachbarn. Beachte daß $grad_A(j) := \sum_{k=1}^{n} a_{kj}$. Der folgende Algorithmus APD berechnet aus der Adjazenzmatrix $A$ die Distanzmatrix $D$.

**function** APD( $A : n \times n$ Adjazenzmatrix ) : $n \times n$ Distanzmatrix
  $Z = A \cdot A$
  **for** $i = 1$ **to** $n$ **do**
    **for** $j = 1$ **to** $n$ **do**
      **if** $(i \neq j)$ **and** $((a_{ij} = 1)$ **or** $(z_{ij} > 0))$ **then** $b_{ij} = 1$
                                  **else** $b_{ij} = 0$
  **if** $(b_{ij} = 1)\ \forall i \neq j$ **then return** $(2 \cdot B - A)$
  $T = \mathrm{APD}(B)$
  $X = T \cdot A$
  **for** $i = 1$ **to** $n$ **do**
    **for** $j = 1$ **to** $n$ **do**
      **if** $x_{ij} \geq t_{ij} \cdot grad_A(j)$ **then** $d_{ij} = 2 \cdot t_{ij}$
                         **else** $d_{ij} = 2 \cdot t_{ij} - 1$
  **return** $D$

Wir zeigen zuerst, daß dieser rekursive Algorithmus tatschlich die korrekte Distanzmatrix berechnet.

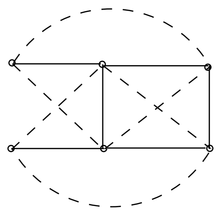
**Behauptung 31** $z_{ij} > 0 \Leftrightarrow$ *es gibt einen Pfad mit genau 2 Kanten zwischen i und j*

**Beweis**

$$\begin{aligned} z_{ij} > 0 \quad &\Leftrightarrow \quad \exists k: \quad a_{ik} = 1 \ \text{ und } \ a_{kj} = 1 \\ &\Leftrightarrow \quad \text{es gibt einen Pfad mit 2 Kanten zwischen } i \text{ und } j \end{aligned}$$

□

**Behauptung 32** *$B$ ist die Adjazenzmatrix eines Graphen $G'$, in dem für jeden Weg der Länge 2 in $G$ eine Abkürzung eingeführt wurde.*



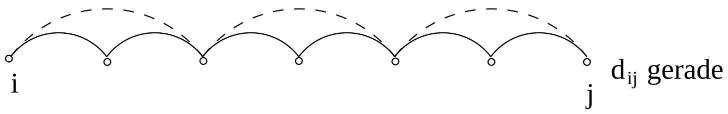
**Behauptung 33** *Wenn $G'$ ein vollständiger Graph ist (d.h. $b_{ij} = 1\ \forall i \neq j$), dann gilt:*

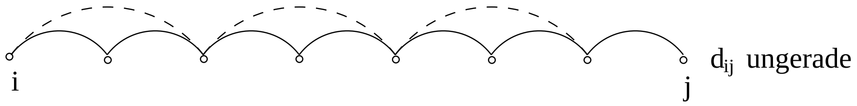
$$d_{ij} = \left\{ \begin{array}{ccc} 1 & \Leftrightarrow & a_{ij} = 1 \\ 2 & \Leftrightarrow & a_{ij} = 0 \end{array} \right\} \iff d_{ij} = 2 \cdot b_{ij} - a_{ij}.$$

**Behauptung 34** *Bezeichne $d_{ij}$ die Distanz zwischen den Knoten $i$ und $j$ in $G$ und $t_{ij}$ die Distanz zwischen ihnen im Graphen $G'$. Dann gilt:*

$$\begin{aligned} d_{ij} \text{ gerade} \quad &\Rightarrow \quad d_{ij} = 2 \cdot t_{ij}, \\ d_{ij} \text{ ungerade} \quad &\Rightarrow \quad d_{ij} = 2 \cdot t_{ij} - 1. \end{aligned}$$

**Beweis**



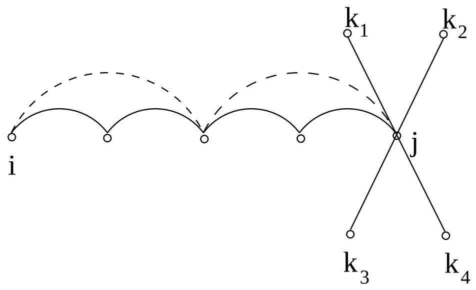

□

Die Distanzmatrix $T$ für $G'$ wird durch den rekursiven Aufruf korrekt berechnet. Die vorhergende Behauptung zeigte, daß man aus $T$ die gewünschte Distanzmatrix $D$ berechnen kann, wenn man nur für jedes $d_{ij}$ weiß, ob es gerade oder ungerade ist. Wie kann man dies nun feststellen, ohne die $d_{ij}$'s zu berechnen? Die folgende Behauptung gibt die wesentliche Charakterisierung:

**Behauptung 35**

1. *Aus $d_{ij}$ gerade folgt $t_{ik} \geq t_{ij}$ für alle Nachbarn $k$ von $j$ in $G$.*
2. *Wenn $d_{ij}$ ungerade ist, dann gilt $t_{ik} \leq t_{ij}$ für alle Nachbarn $k$ von $j$ und es gibt einen Nachbarn $k$ von $j$ in $G$ mit $t_{ik} < t_{ij}$.*

**Beweis**




□

Wenn man nun über alle Nachbarn $k$ von $j$ summiert, bekommt man folgende Aussage:

**Behauptung 36**

$$\begin{aligned} d_{ij} \text{ gerade} \quad &\Leftrightarrow \quad \sum_{k \text{ Nachbar von } j} t_{ik} \geq grad(j) \cdot t_{ij} \\ d_{ij} \text{ ungerade} \quad &\Leftrightarrow \quad \sum_{k \text{ Nachbar von } j} t_{ik} < grad(j) \cdot t_{ij} \end{aligned}$$

Aber beachte, daß

$$\sum_{k \text{ Nachbar von } j} t_{ik} = \sum_{1 \leq k \leq n} t_{ik} \cdot a_{kj} = x_{ij} .$$

Also kann man nach dem Berechnen der Matrix $X$ tatsächlich in der im Algorithmus angegebenen Weise aus den $t_{ij}$'s die gewünschten $d_{ij}$'s herleiten.

Nun zur Analyse der Laufzeit des Algorithmus. Wenn man vom rekursiven Aufruf absieht, braucht APD Zeit für zwei Matrizenmultiplikationen sowie $O(n^2)$ zusätzliche Operationen. Beim rekursiven Aufruf wird allerdings

das $n$ nicht kleiner. Woher weiß man also, daß der APD überhaupt terminiert. Das Wichtige ist, daß im Graphen $G'$ des rekursiven Aufrufs die kürzesten Wege nur mehr halb so lang sind wie in $G$.
Sei nun $\delta$ der *Durchmesser* von $G$, also die größte Distanz zwischen zwei Knoten in $G$. Wenn wir die Laufzeit von APD durch $T(n, \delta)$ beschreiben, dann gilt

$$T(n,\delta) = \begin{cases} M(n) + O(n^2) & \text{wenn } \delta \leq 2 \\ 2 \cdot M(n) + T(n, \lceil \delta/2 \rceil) + O(n^2) & \text{wenn } \delta > 2 \end{cases}$$

Dabei ist $M(n)$ die Zeit, die man zum Multiplizieren zweier $n \times n$ Matritzen benötigt. Da $M(n) = \Omega(n^2)$ kann man sehen, daß dann

$$T(n, \delta) = O(M(n) \log \delta) = O(M(n) \log n)$$

gelten muß. (Beachte $\delta \leq n - 1$.)

Vorlesung vom 25.01.1996
Dirk Holz, Marco Nissen, Stefan Reichert

## 7.8 Zeugen für (boolesche) Matrizen-Multiplikation

**Definition 15** *Es seien $A$ und $B$ zwei $n \times n$ Matrizen mit $0-1$-Werten und $C = A \cdot B$ das Produkt dieser beiden Matrizen. Es gilt, daß $c_{ij}$ genau dann $\neq 0$ ist, wenn ein $k$ existiert mit $a_{ik} = 1$ und $b_{kj} = 1$. So ein $k$ nennen wir* Zeugen *für $(i,j)$ (siehe Abbildung 7.4). Eine* Zeugenmatrix *für $A$ und $B$ ist analog definiert als eine $n \times n$-Matrix $Z$, für deren Eintäge $z_{ij}$ gilt:*

$$z_{ij} = \begin{cases} 0, & \textit{falls } c_{ij} = 0\ ; \\ \textit{irgendein Zeuge für}(i,j) & \textit{sonst}\ . \end{cases}$$

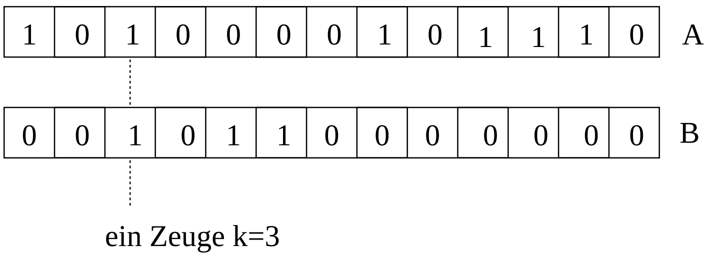



Abbildung 7.4: "$i$-te" Zeile von $A$ und "$j$-te" Spalte von $B$.

Betrachten wir dazu ein Beispiel:

$$\underbrace{\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} 1 & 0 & 2 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}}_{C}$$

Wir überprüfen nun für alle $(i,j)$ und für alle $k$, ob $k$ ein Zeuge für $(i,j)$ ist. Offensichtlich geht dies in $O(n^3)$ Zeit.
Eine Zeugenmatrix für $A$ und $B$ sieht z.B. so aus:

$$Z = \begin{pmatrix} 2 & 0 & 1 \\ 2 & 0 & 2 \\ 0 & 3 & 1 \end{pmatrix}$$

Warum mchte man überhaupt so eine Zeugenmatrix berechnen? Zum Beispiel kann man sie im ALL-PAIRS kürzesten Wege Problem für ungerichtete, ungewichtete Graphen verwenden. Wir lassen folgendes Problem dem Leser als Übung:
**Übung:** *Sei $G$ ein ungereichteter, ungewichteter Graph, $A$ seine Adjazenzmatrix und $D$ seine Distanzmatrix. Zeigen Sie, daß man mit Hilfe von drei Zeugenmatrixberechnungen und $O(n^2)$ zusätzlicher Zeit man aus $A$ und $D$ eine Vorgängermatrix $\Pi$ von $G$ bestimmen kann.*

Hinweis: *Zeigen Sie zuerst, daß man dies mit $n-1$ Zeugenmatrixberechnungen erzielen kann.*

Wir wollen nun einen Algorithmus zum Bestimmen eine Zeugenmatrix für Matrizen $A$ und $B$ entwickeln, der eine schnellere Laufzeit als kubisch hat. Es wird ein randomisierter Algorithmus sein, dessen *erwartete* Laufzeit gut ist.
Dazu ueberlegt wir uns zuerst einmal, was passiert, wenn man die $j$-te Spalte der booleschen Matrix $A$ mit $j$ multipliziert und dann diese veränderte Matrix $A$ mit $B$ multipliziert. Wenn es nun für ein Paar $(i,j)$ *genau einen* Zeugen gibt, dann kann er durch diese Matrizenmultiplikation gefunden werden.

**Beispiel:**

$$\begin{pmatrix} 1 & 2 & 0 \\ \mathbf{0} & \mathbf{2} & \mathbf{0} \\ 1 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{0} & 0 & 1 \\ \mathbf{1} & 0 & 1 \\ \mathbf{0} & 1 & 0 \end{pmatrix} = \begin{pmatrix} . & . & . \\ 2 & . & . \\ . & . & . \end{pmatrix}$$

Auf dieser Beobachtung basiert der folgende Algorithmus:

**function** ZEUGENMATRIX( $A, B : n \times n$ 0-1-Matrix) : $n \times n$ Matrix
(0) $W = -A \cdot B$
(1) **for** $l = 0$ **to** $\log_2 n$ **do**
(2) $d = 2^l$
(3) **repeat** $4 \cdot \log_2 n$ **times**
(4) wähle $d$ unabhängige Zufallszahlen $k_1, \ldots k_d$ gleichmäßig aus $\{1, \ldots n\}$
(5) $X$ sei die $n \times d$ Matrix mit Spalten $k_i \cdot A_{\star k_i}$ (* $A_{\star k_i}$ bedeutet die ,,$k_i$-te" Spalte *)
(6) $Y$ sei die $d \times n$ Matrix mit Zeilen $B_{k_i \star}$ (* $B_{k_i \star}$ bedeutet die ,,$k_i$-te" Zeile *)
(7) $C = X \cdot Y$
(8) **for each** $(i,j)$ **do**
(9) **if** $w_{ij} < 0$ und $c_{ij}$ ist Zeuge für $(i,j)$ **then** $w_{ij} := c_{ij}$
(10) **for each** $(i,j)$ **do**
(11) **if** $w_{ij} < 0$ **then**
(12) **for** $k = 1$ **to** $n$ **do**
(13) **if** $a_{ik} \cdot b_{kj} = 1$ **then** $w_{ij} = k$
(14) **return** $W$

Wir behandeln zuerst die Korrektheit diese Algorithmus.

**Behauptung 37** *Wenn $U = A \cdot B$ gilt, dann ist $u_{ij}$ gleich der Anzahl der Zeugen für das Paar $(i,j)$*

**Beweis** Offensichtlich für 0-1-Matrizen $A, B$. $\square$

Damit gilt für das in Schritt (1) berechnete $W$:

1. $w_{ij} < 0$: In diesem Fall gibt es einen Zeugen für $(i,j)$.
2. $w_{ij} = 0$: Es gibt keinen Zeugen für $(i,j)$.
3. $w_{ij} > 0$: In diesem Fall ist $w_{ij}$sicher ein Zeuge für $(i,j)$.

Nun kann man leicht sehen, daß der obige Algorithmus sicher korrekt ist. Denn nur ursprünglich negativen $w_{ij}$'s werden positive Werte zugewiesen, und dann wird jedesmal explizit getestet, ob es sich auch wirklich um einen Zeugen handelt. Zeilen (10) bis (13) stellen außerdem sicher, daß wirklich für alle Paare $(i,j)$, für die es gibt, ein Zeuge gefunden wurd.

Kommen wir zur Laufzeitanalyse des Algorithmus.

| | |
|---|---|
| (0) | $O(n^\omega)$ |
| (2) | $O(1)$ |
| (4) | $O(d)$ |
| (5),(6) | $O(n^2)$ |
| (7) | Zeit, eine $(n \times d) \cdot (d \times n)$ Matrizenmultiplikation durchzuführen |
| (8)-(9) | $O(n^2)$ |

Hier nehmen wir an, eine $n \times n$-Matrizenmultiplikation braucht $O(n^\omega)$ Zeit, wobei $\omega \geq 2$ ist.
Im Algorithmus verwenden wir eine $(n \times d) \cdot (d \times n)$ - Matrizenmultiplikation. Wir müssen deren Laufzeit bestimmen (siehe Abbildung 7.5). Dies erreichen wir, indem wir blockweise quadratisch multiplizieren. Wir teilen also eine $n \times d$ - Matrix in $n/d$ Blöcke mit je $d \times d$ - Größe.

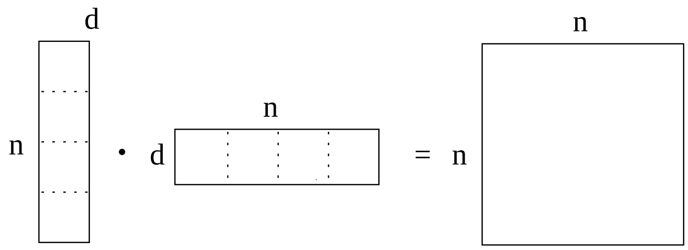



Abbildung 7.5: Matrizenmultiplikation von nicht-quadratischen Matrizen

Wir haben damit $(n/d)^2$ Multiplikationen von $d \times d$- Matrizen. Also folgt für die Laufzeit von Zeile (7) des Algorithmus: $(n/d)^2 \cdot O(d^\omega) = O(n^2 \cdot d^{\omega-2})$.
Da in einer endlichen geometrischen Summe der größte Summand die Summe dominiert, und in dem Algorithmus $d$ schrittweise $2^l$ zugewiesen wird, ergibt sich als Laufzeit der ersten **for**-Schleife:

1. Falls $\omega > 2$:

$$\begin{aligned} & O\left(\sum_{0 \leq l \leq \log n} \left[n^2 \cdot (2^l)^{\omega-2} \cdot \log n\right]\right) \\ = \; & O\left(n^2 \cdot \underbrace{(2^{\omega-2})^{\log n}}_{n^{\omega-2}} \cdot \log n\right) \\ = \; & O\left(n^\omega \cdot \log n\right) \end{aligned}$$

2. Falls $\omega = 2$:

$$O\left(\sum_{0 \leq l \leq \log n} \left[n^2 \cdot (2^l)^{\omega-2} \cdot \log n\right]\right) = O\left(n^\omega \cdot \log^2 n\right)$$

Leider gilt diese Laufzeit nur für den ersten Teil des Algorithmus. Wir müssen nun noch die erwartete Anzahl von Zeugen bestimmen, die im ersten Teil des Algorithmus nicht gefunden werden und im zweiten Teil noch explizit berechnet werden müssen. Wir hoffen, daß die Anzahl nicht zu groß ist, denn jede explizite Berechnung eines Zeugen kostet $O(n)$ Zeit. Im folgenden wollen wir Zeigen, daß die erwartete Laufzeit für Zeilen (10) bis (14) gleich $O(n^2)$ ist.
Sei $p_{ij}$ die Wahrscheinlichkeit, daß es einen Zeugen für $(i,j)$gibt, aber trotzdem in den Zeilen (3) bis (9) keiner gefunden wurde.
Dann gilt für die erwartete Laufzeit für das Paar $(i,j)$in der Schlußschleife:

$$p_{ij} \cdot O(n) + (1 - p_{ij}) \cdot O(1) = O(p_{ij} \cdot n + 1).$$

Unser Ziel ist es also, $p_{ij} \leq 1/n$ zu zeigen:

**Behauptung 38** *Nehmen wir an, es gebe $c$ Zeugen für ein Paar $(i,j)(1 \leq c \leq n)$. Sei $d$ so, daß $n/2 \leq c \cdot d \leq n$ ist.*
*Dann beträgt die Wahrscheinlichkeit, daß es unter $d$ zufällig, unabhängig und gleichmäßig aus $\{1,\ldots,n\}$ gewählten Zahlen $k_1,\ldots k_d$ genau einen Zeugen für $(i,j)$gibt, mindestens $1/(2e)$.*

**Beweis**

$$\begin{aligned} & Pr(\text{es gibt unter } k_1,\ldots k_d \text{ genau einen Zeugen für } (i,j)) \\ = & \underbrace{\binom{d}{1} \cdot \frac{c}{n}}_{\frac{d\cdot c}{n} \geq \frac{1}{2}} \cdot \Big(1 \underbrace{-\frac{c}{n}}_{-\frac{c}{n} \geq -\frac{1}{d}}\Big)^{d-1} \\ \geq & \frac{1}{2}\left(1-\frac{1}{d}\right)^{d-1} \\ \geq & \frac{1}{2 \cdot e} \end{aligned}$$

□

**Behauptung 39** *Seien $(k_1,\ldots k_d) \in \{1,\ldots n\}^d$, $X$, $Y$ und $C = X \cdot Y$ wie im Algorithmus angegeben. Falls es unter $(k_1,\ldots k_d)$ genau einen Zeugen $k_\lambda$ für $(i,j)$gibt, dann gilt $c_{ij} = k_\lambda$.*

**Beweis**

$$c_{ij} = \sum_{1 \leq \nu \leq d} \Big[k_\nu \cdot \underbrace{a_{ik_\nu} \cdot b_{k_\nu j}}_{\Leftrightarrow\ 1,\ \text{falls } \nu = \lambda}\Big].$$

□

Also folgt für die Wahrscheinlichkeit, daß in den $4 \cdot \log_2 n$ Iterationen mit relevantem $d$ kein Zeuge gefunden wird:

$$p_{ij} \leq \left(1 - \frac{1}{2e}\right)^{4 \cdot \log_2 n} < \left(\frac{1}{2}\right)^{\log_2 n} < \frac{1}{n}.$$

Damit haben wir gezeigt, daß die erwartete Laufzeit für $(i,j)$in der Schlußschleife

$$O(p_{ij} \cdot n + 1) = O\Big(\frac{1}{n} \cdot n + 1\Big) = O(1).$$

ist, und damit hat die gesamte Schleife erwartete Laufzeit $O(n^2)$.
Also ist die erwartete Gesamtlaufzeit des randomisierten Algorithmus zur Bestimmung der Zeugenmatrix

$$O\left(n^\omega \cdot \log n\right).$$

Vorlesung vom 29.1.1996
Daniel Wagner, Stefan Dietrich

## 7.9 Minimale aufspannende Bäume (MST)

Motivation:
Hat man ein Straßennetz, wie zum Beispiel die Landstraßen im Saarland, und es kommt zu einem alles glatt werden lassenden Eisregen, wie zum Beispiel in den letzten Tagen, dann könnte es ein Ziel des Streudienstes zu sein, möglichst schnell sicher zu stellen, daß jede Ortschaft von jeder anderen über gestreute Straßen erreichbar ist. Anders ausgedrückt, man möchte möglichst wenig Straßenkilometer streuen, aber trotzdem vollständige Erreichbarkeit erzielen.
Graphentheoretisch kann das folgendermaßen modelliert werden: Der Graph $G = (V,E)$ ist gegeben, dessen Knoten die Ortschaften sind, und dessen Kanten die Verbindungsstraßen zwischen den Ortschaften darstellen. Die Kanten sind durch die entsprechenden Straßenlängen gewichtet. Ziel ist es eine Teilmenge der Kanten zu finden, die minimale Gesamtlänge haben soll und die einen zusammenhängenden Graphen bilden soll, an dem jeder Knoten von $V$ anliegt. Es handelt sich hierbei genau um einen sogenannten *minimalen aufspannenden Baum*.

**Definition 16** *Sei $G = (V, E)$ ein ungerichteter, gewichteter Graph.*

- *Einen Teilbaum von $G$, an dem jeder Knoten von $G$ anliegt, nennt man einen aufspannenden Baum von $G$.*
- *Wenn $A \subset E$ eine Teilmenge der Kanten des Graphen $G$ ist, dann sei $w(A) = \sum_{e \in A} w(e)$ das Gewicht der Menge $A$.*
- *Ein minimal aufspannender Baum $T$ liegt vor, falls $w(T)$ minimal ist.*
- *Eine Menge $A \subset E$ heißt erweiterbar, wenn es einen minimalen aufspannenden Baum $T$ gibt mit $A \subset T$.*
- *Eine Kante $e \in E$ heißt sicher für $A$, wenn $A \cup \{e\}$ erweiterbar ist ($e \notin A$).*

Folgender Algorithmus liefert dann einen minimalen aufspannenden Baum:

"Greedy" Algorithmus (habgieriger Algorithmus)
    $A = \emptyset$
    **for** $k = 1$ **to** $n - 1$ **do**
        finde Kante $e$, die sicher für $A$ ist
        $A = A \cup \{e\}$

Die Frage ist nun, wie man eine sichere Kante findet. Dazu führt man zunächst folgende Begriffe ein:

**Definition 17** *Sei $G = (V, E)$ ein Graph.*

- *$(X, Y)$ heißt Schnitt von $G$, wenn $Y \cup X = V$ und $X \cap Y = \emptyset$.*
- *Eine Kante $e = [u, v]$ kreuzt den Schnitt $(X, Y)$, falls $u \in X$ und $v \in Y$.*
- *$B \subset E$ respektiert den Schnitt $(X, Y)$, wenn kein $e \in B$ den Schnitt kreuzt.*
- *Kanten, die den Schnitt kreuzen, heißen Schnittkanten.*

**Behauptung 40** *Sei $A \subset E$ erweiterbar und $(X, Y)$ ein Schnitt, der von $A$ respektiert wird. Dann gilt, daß jede leichteste Schnittkante sicher für $A$ ist.*

**Beweis** Sei $T$ ein minimaler aufspannender Baum, der das erweiterbare $A$ enthält, und sei $e = [x, y]$ eine leichteste Schnittkante zwischen $X$ und $Y$. Wenn $e \in T$, dann ist offensichtlich auch $A \cup \{e\}$ erweiterbar, also $e$ ist sicher für $A$. Nehmen wir also an, $e \notin T$.
In dem aufspanneden Baum $T$ muß es einen Pfad von $x$ nach $y$ geben. Auf diesem Pfad muß es eine Kante $e'$ geben, die den Schnitt $(X, Y)$ kreuzt. Ersetze $e'$ durch $e$ in $T$ und betrachte den sich so ergebenden aufspannenden Baum $T'$. Nach Annahme ist das Gewicht von $e'$ nicht kleiner als das der Kante $e$. Also ist das Gewicht von $T'$ nicht größer als das von $T$, und daher ist $T'$ auch ein minimaler aufspannender Baum, aber mit $A \cup \{e\} \subset T'$. Also ist $A \cup \{e\}$ erweiterbar. □

Mit Hilfe dieser Aussage ist es leicht sichere Kanten zu finden. Die Frage ist nur, welche Schnitte man betrachten soll. Welche Schnitte betrachtet werden, ist genau der Unterscheidungspunkt der verschiedenen Realisierungen des Greedy Algorithmus.
Man kann den Greedy Algorithmus auch als langsames Wachsen eines Waldes von Bäumen in einen Graphen $G$ betrachten. In jedem Schritt des Algorithmus kommt eine (sichere) Kante dazu. Diese Kante verbindet zwei Bäume des derzeitigen Waldes zu einem Baum. Anfangs gibt es $n$ triviale Bäume, die nur aus einzelnen Knoten bestehen; am Ende besteht der Wald aus nur einem Baum, dem gesuchten minimalen aufspannenden Baum von $G$.
Welche Schnitte werden nun betrachtet, um sichere Kanten zu finden. Im Greedy Algorithmus werden typischerweise nur Schnitte $(X, Y)$ betrachtet, in denen $X$ die Knotenmenge eines Baumes des derzeitigen Waldes bildet. Man muß also dann die leichteste Kante finden, die den Baum verläßt. Diese Methode läßt noch immer viel Freiheit, da es ja mehr als einen Baum im Wald zur Auswahl gibt. Es folgen nun drei Algorithmen, die diese Freiheit unterschiedlich nutzen.
In **Kruskals Algorithmus** erfolgt die Wahl des Schnittes implizit. Es werden nämlich die Kanten nach aufsteigendem Gewicht durchgegangen; hat die Kante beide Endpunkte im selben Baum des derzeitigen Waldes, wird

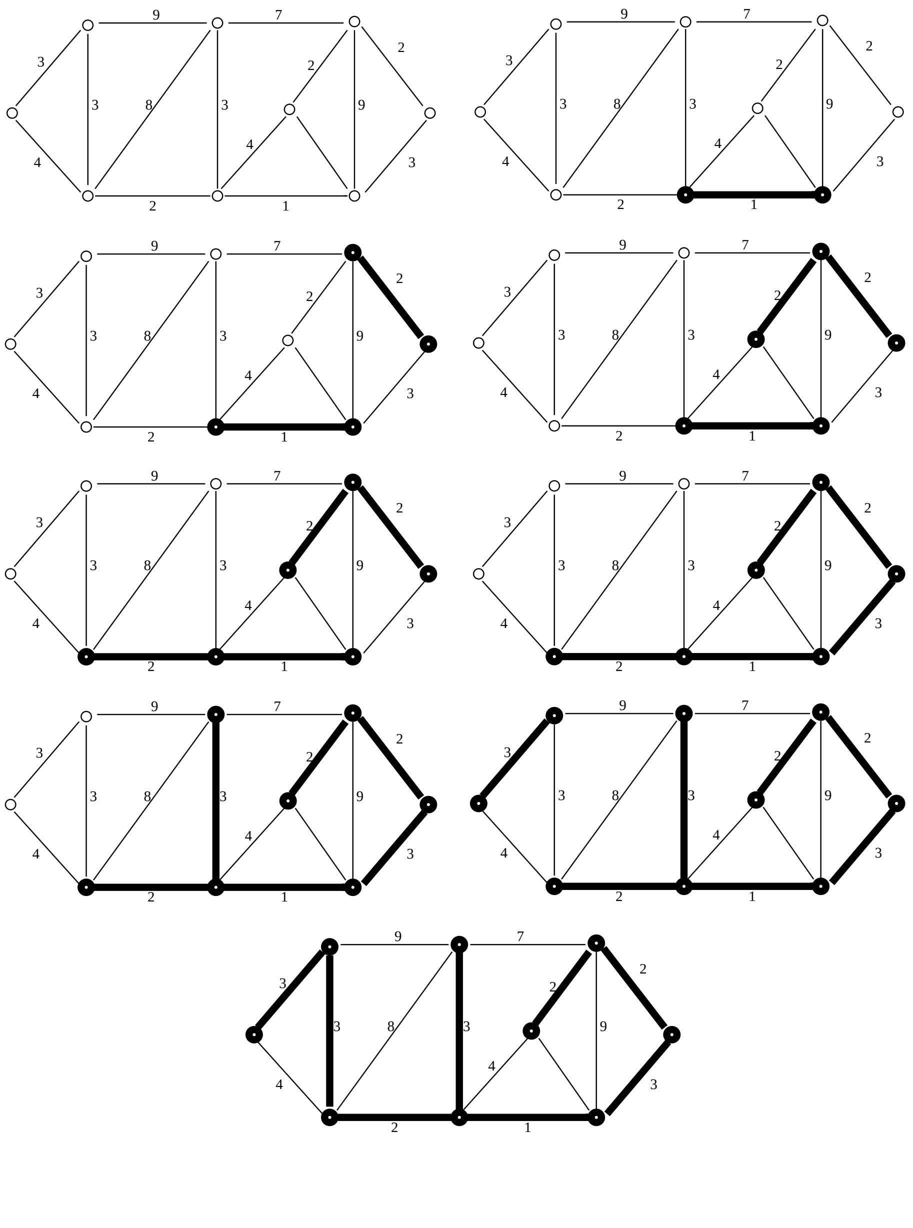
sie einfach ignoriert; hat sie ihre Endpunkte in verschiedenen Bäumen $T_1$ und $T_2$, dann wird sie in die erweiterbare Kantenmenge $A$ aufgenommen — die beiden Bäume werden also zu einem verbunden. Ein betrachteter Schnitt $(X, Y)$ hat also die Knoten von $T_1$ als $X$. Die dazugenommene Kante ist die leichteste dieses Schnittes, da ja Kanten nach aufsteigendem Gewicht betrachtet werden.
Es folgt eine genauere Beschreibung dieses Algorithmus. Sie verwendet als Datenstruktur die früher beschriebene *Union-Find* Struktur.

<u>Kruskals Algorithmus</u>

(1) $A = \emptyset$
(2) **for each** $v \in V$ **do** MAKESET($v$)
(3) sortiere Kanten nach aufsteigendem Gewicht
(4) **for each** $[u, v] \in E$ (in aufsteigender Folge) **do**
(5)     **if** FIND($u$) $\neq$ FIND($v$) **then**
(6)         $A = A \cup [u, v]$
(7)         UNION($u, v$)

Zur Laufzeit: Zeilen (1) bis (3) werden durch die $O(|E| \log n)$ Zeit fürs Sortieren in Zeile (3) dominiert. In Zeilen (4) bis (7) gibt es $2|E|$ FIND Operationen, $n - 1$ UNION Operationen, und $n - 1$ Vergrößerungen der Menge $A$. Mit der schnellsten Union-Find Datenstruktur (Pfadkompression, etc) kann dies in $O(|E| \cdot \alpha(|E|, n))$ Zeit bewerkstelligt werden. Die Gesamtlaufzeit ist also durchs Sortiern bestimmt und beträgt $O(|E| \log n)$.

**Beispiel für Kruskals Algorithmus:**



Als nächstes behandeln wir **Borouvka's Algorithmus**. Er verläuft in Phasen, und in einer Phase bestimmt jeder Baum des derzeitigen Waldes einen Schnitt. Für alle diese Schnitte werden die jeweils leichtesten Kanten gesucht und in die erweiterbare Menge $A$ eingefügt. Bei gleichgewichtigen Kanten muß man dabei Vorsicht walten lassen, damit es zu keiner Zyklenbildung kommt.
Mit Hilfe einer Union-Find Struktur wird wieder eine Partition der Knotenmenge $V$ verwaltet, deren Elemente die Knotenmengen der einzelnen Bäume des derzeitigen Waldes sind. Der Einfachheit halber bezeichnen wir diese Partition als *Wald*.

<u>Borouvka's Algorithmus</u>

(1) $A = \emptyset$
(2) **for each** vertex $v \in V$ **do** MAKESET($v$)
(3) **while** $|Wald| > 1$ **do**
(4)     **for each** Baum $B \in Wald$ **do**
(5)         $min[B] = \infty$
(6)         $minedge[B] = NIL$
(7)     **for each** edge $[u,v] \in E$ **do**
(8)         $x =$ FIND($u$); $y =$ FIND($v$);
(9)         **if** $x = y$ **then** delete $[u,v]$ from $E$
(10)         **else**
(11)             **if** $w(u,v) < min[x]$ **then** $min[x] = w(u,v)$
(12)                         $minedge[x] = [u,v]$
(13)             **if** $w(u,v) < min[y]$ **then** $min[y] = w(u,v)$
(14)                         $minedge[y] = [u,v]$
(15)     **for each** $x \in Wald$ **do**
(16)         $[u,v] = minedge[x]$
(17)         **if** FIND($u$) $\neq$ FIND($v$) **then**
(18)                         $A = A \cup [u,v]$
(19)                         $E = E \setminus [u,v]$
(20)                         UNION(FIND($u$), FIND($v$))

Zur Laufzeitanalyse: Mit jeder Phase — also Iteration der **while**-Schleife von Zeile (3) — verringert sich die Anzahl der Bäume im Wald um mindestens die Hälfte. Es gibt also insgesamt nur $\log n$ Phasen. Leider kann man keine allgemeine Aussage machen, wie sehr sich die Anzahl der Kanten zwischen den Phasen verringert. Wir nehmen also den (mehr als) schlechtesten Fall an, daß in jeder Phase alle Kanten vorhanden sind. Der Hauptaufwand in jeder Phase besteht darin, daß für jede noch bestende Kante zwei FIND's (Zeile (8)) sowie zusätzlich konstante Arbeit (Zeilen (9) bis (14)) durchgeführt werden müssen. Wenn man eine Union-Find Struktur verwendet, die FIND's in konstanter Zeit und UNION's in logarithmischer Zeit ermöglicht, dann bedeutet dies, daß Zeilen (1) bis (14) insgesamt $O(|E| \log n)$ benötigen. In den restlichen Zeilen werden im wesentlichen noch einmal $O(n)$ FIND's sowie $n-1$ UNION's durchgeführt, was zusätzlich Zeit $O(n \log n)$ erfordert.
Die Gesamtlaufzeit ist damit durch $O(|E| \log n)$ beschränkt. Dies ist allerdings eine sehr pessimistische Schranke, da sich oft zwischen den Phasen die Kantenmenge wesentlich verkleinern wird, was in unserer Analyse nicht berücksichtigt wird.

Die dritte Version des Greedy Algorithmus, die wir betrachten, ist **Prim's Algorithmus**. In ihm besteht der derzeitige Wald immer aus einem "großen" Baum $B$ und restlichen trivialen, aus einem Knoten bestenden Bäumen. Das bedeutet, daß immer die Knotenmenge von $B$ den Schnitt bestimmt, dessen leichteste Kante als nächstes dazugenommen wird. Man möchte jetzt also immer die leichteste Kante bestimmen können, die den derzeitigen Baum $B$ verläßt. Das legt die Verwendung einer Prioritätenschlange nahe. Diese Prioritätenschlange enthält die Knoten, die nicht im derzeitigen Baum $B$ sind. Der Schlüssel eines Knotens ist das Gewicht der leichtesten Kante, die ihn zu $B$ verbinden ($\infty$ falls keine solche Kante existiert). Der Algorithmus ist sehr ähnlich zu Dijkstra's Algorithmus für das Single-Source kürzeste Wege Problem. Er verwendet in gleicher Weise Vorgängerzeiger $\pi[\ ]$ um den konstruierten Baum darzustellen.

<u>Prim's Algorithmus</u>

(0) $A = \emptyset$
(1) $Q =$EMPTYQUEUE (Prioritätetenschlange)

Sei $r$ irgendein Knoten in $V$
**for each** $u \neq r$, $u \in V$ **do**
    key$[u] = \infty$
Insert$(u, Q)$
Key$[r] = 0$
$\pi[r]$ =NIL
Insert$(r, Q)$
**while** $Q \neq$ empty **do**
    $u$=Extract-Min$(Q)$
    Insert$[u, \pi(u)]$ Into $A$
    **for each** $v \in$ Adj$[u]$ **do**
        **if** $v \in Q$ **and** $w(u, v) < Key[v]$ **then**
            $\pi[v] = u$
            DecreaseKey$(v, w(u, v))$

Die Laufzeitanalyse verläuft vollkommen gleich wie bei Dijkstras Algorithmus, und je nach Leistungsfähigkeit der verwendeten Prioritätenschlange erzielt man eine Laufzeit von $O(n^2)$ naiv, $O(|E| \log n)$ mit Binary oder Binomial Heaps, und $O(|E| + n \log n)$ mit Fibonacci-Heaps.

Vorlesung vom 1.2.96
Ralf Mindermann, Sven Wohlgemuth

************* Dieser folgende kleingedruckte ************
************ Teil kann vorerst ignoriert werden *************

Man kann den Algorithmus von Barouvka so abändern, daß er pro Iteration nur lineare Zeit $O(|V| + |E|)$ benötigt, statt $O((n + |E|) \cdot \alpha(\ldots))$ (siehe Zettel12). Die Idee ist, statt einer Union-Operation zwei Knoten mittels Kantenkontraktion zu einem "Superknoten" zu vereinigen. Dann wiederholt man die Schritte des Algorithmus für diese Superknoten, da es zwischen ihnen wieder Kanten geben kann usw.

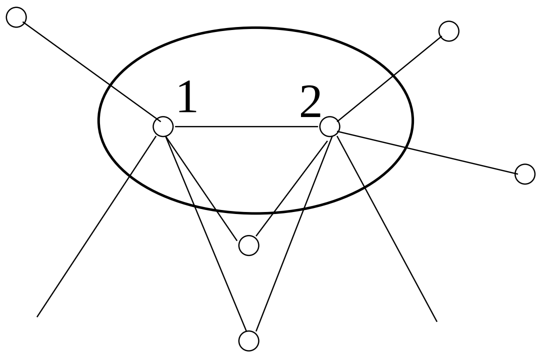



Abbildung 7.6: Zwei Knoten werden zu einem Superknoten vereinigt

Sei $n$ die Anzahl der Knoten, $e$ die Anzahl der Kanten vor einer Iteration. In einer Phase des Algorithmus verringert sich so die Anzahl der Knoten auf höchstens $n/2$, die der Kanten auf maximal $e - n/2$. Die kontrahierten Kanten verschwinden sozusagen in den Superknoten. Es folgt $T(n) = O(n) + T(\frac{n}{2}) = O(n)$, also lineare Laufzeit, wenn man die Kanten vernachlässigt.

Über die Kanten wollen wir nun eine ähnliche Aussage machen. Dazu brauchen wir ein Kriterium, wann man eine Kante eleminieren kann, wann also eine Kante sicherlich nicht im MSB des Graphen vorkommt.

**Behauptung 41** *Sei $G = (V, E)$ ein ungerichteter, gewichteter Graph und $C$ ein beliebiger Kreis von Kanten in diesem Graphen (siehe Abbildung 7.7). Falls $e$ die eindeutig schwerste Kante von $C$ ist, so ist $e$ sicher nicht im MSB von $G$.*

**Beweis** (Widerspruchsbeweis)
Angenommen, $e$ ist im MSB von $G$. Dann wurzeln an den zu $e$ inzidenten Knoten zwei disjunkte Teilbäume (siehe Abbildung 7.9).

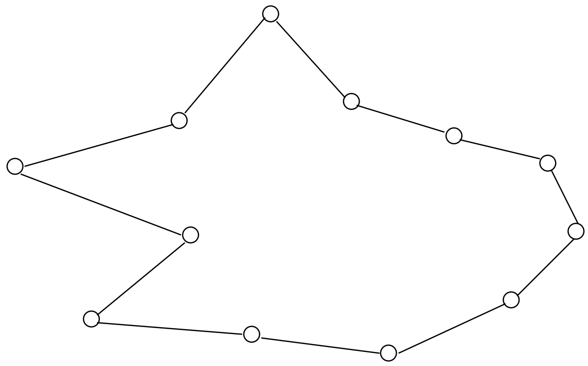


Abbildung 7.7: Kreis von Kanten in $G$

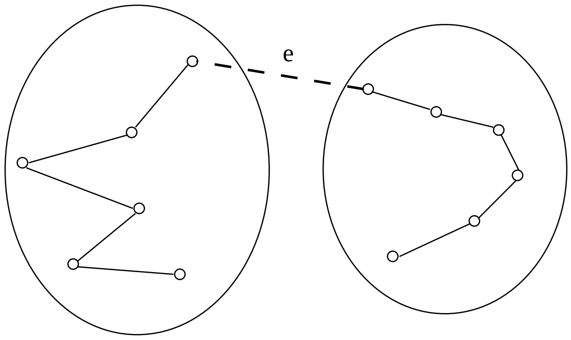



Abbildung 7.8: Kante $e$ verbindet zwei disjunkte Teilbäume

Andererseits ist $e \in C$ in einem Kreis. Wenn man nun $e$ aus dem MSB von $G$ entfernt, dann erhält man einen Schnitt. Es existiert aber eine Kante $e'$ auf dem Kreis, deren Gewicht $w(e')$ kleiner als das von $e$ ist und die den Schnitt kreuzt (siehe Abbildung 7.9).
Also: Nimmt man $e$ hinaus und setzt die Kante $e'$ dafür ein, so verkleinert sich das Gesamtgewicht des MSB. Widerspruch! $\square$

Man könnte also beliebige Kreise auswählen und daraus die jeweils schwerste Kante entfernen. Allerdings ist ein organisiertes Anwenden des Kriteriums kaum möglich, da zu viele Kreise in einem Graphen existieren. Eine zufällige Auswahl der Kreise bringt kaum etwas.

**Definition 18** *Sei $G = (V, E)$ ein Graph und $F$ ein Wald über diesem Graphen. Einzelne Bäume haben paarweise disjunkte Kanten. Eine Kante $e \in E$ ist nun entweder eine Baumkante, oder sie verläuft in einem Baum ohne dabei Baumkante zu sein, oder sie verbindet zwei solcher Bäume. Eine Kante $e$ heißt F-leicht, wenn entweder*

- *$e \in F$ gilt, oder*
- *$e$ zwei Bäume von $F$ verbindet, oder*
- *$e$ nicht die eindeutig schwerste Kante auf dem durch sie in $F$ induzierten Kreis ist.*

Aus Behauptung 41 folgt, daß für einen minimalen aufspannenden Baum eines Graphen $G$ nur $F$-leichte Kanten in Frage kommen. Wenn man nun für eine Teilmenge der Kanten von $G$ einen minimalen auspannenden Wald $F$ bildet und die $F$-schweren Kanten aus $G$ entfernt, so hat man nur für den MSB unrelevante Kanten entfernt. Es bleibt nur die Frage, wie sinnvoll diese Methode ist. Wie viele Kanten kann man auf diese Art in einem Schritt eliminieren?

**Behauptung 42** *Sei $G$ ein Graph mit $n$ Knoten und $e$ Kanten. Bilde einen Teilgraphen $H$, indem jede Kante mit Wahrscheinlichkeit $1/2$ unabhängig ausgewählt wird. Sei $F$ der minimale aufspannende Wald von $H$. Die erwartete Anzahl von $F$-leichten Kanten ist dann höchstens $2n$ (und damit vollkommen unabhängig von $e$).*

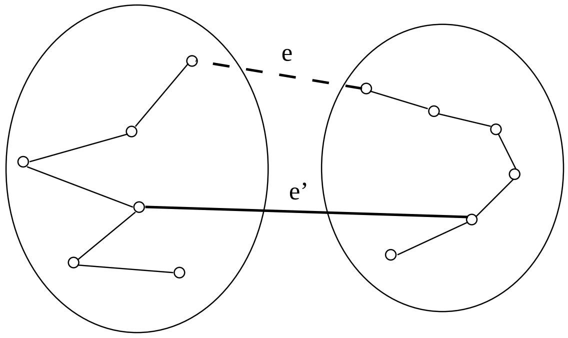



Abbildung 7.9: Kante $e'$ kreuzt die disjunkten Knotenmengen

**Beweis** Sortiere die Kanten von $G$ aufsteigend nach ihren Gewichten. Wende Kruskals Algorithmus an und wirf bei jeder Kante vorher eine Münze, ob sie betrachtet wird und damit zu $H$ gehört oder nicht. Die erwartete Anzahl der Kanten in $H$ ist somit $e/2$. Sei $F$ der minimal aufspannende Wald von $H$ ($H$ muß nicht mehr zusammenhängend sein). Es gilt $|F| < n$ und damit gibt es Kanten $e \in H$, die nicht in $F$ und damit $F$-schwer sind. Als $F$-leichte und damit für den minimalen aufspannenden Baum von $G$ relevante Kanten kommen also nur die Kanten aus $F$ und ein paar der Kanten, die vom Münzwurf ausgeschlossen wurden, in Frage. Zu zeigen ist jetzt noch, daß die Anzahl der $F$-leichten Kanten unter denen, die vom Münzwurf ausgeschlossen wurden, nicht gößer als $n$ ist. $\square$

Kommen wir nun zum randomisierten Algorithmus von Karger, Klein und Tarjan zur Berechnung des minimalen auspannenden Baumes eines ungerichteten, gewichteten Graphen $G = (V, E)$. Dieser Algorithmus läuft aufgrund der obigen Laufzeitbetrachtungen zum Algorithmus von Borouvka, unter Verwendung von Kantenkontraktion, linear in der Anzahl der Knoten. Wir werden zeigen, daß er auch etwartete lineare Laufzeit in der Anzahl der Kanten hat.

1) Lasse eine randomisierte Phase von Borouvka laufen, d.h. suche für jeden Knoten die leichteste Kante und kontrahiere sie. Dies führt zu einem Graphen $G' = (V', E')$.
2) Falls $\frac{|E'|}{|V'|} \leq 6$ ist, dann setze $E'' := E'$ und fahre bei Scritt 3) fort.
   Ansonsten wähle Teilgraphen $H$ von G', wobei jede Kante zufällig mit Wahrscheinlichkeit $1/2$ genommen wird. Wende den Algorithmus rekursiv auf $H$ an und berechne den minimalen aufspannenden Wald $F$ von $H$. Konstruiere $E'' := \{e \in E' \mid e \text{ ist } F\text{-leicht } \}$ (nach Algorithmen von Dixon, Rauch und Tarjan oder Viking, auf die hier nicht eingegangen wird).
3) Wende den Algorithmus rekursiv auf den Graphen $(V', E'')$ an und berechne so den MSB $F'$ von $G'$. Mache die Kantenkontaktionen aus Schritt 1) rückgängig und extrahiere den MSB von $G$.

Die einzelnen Schritte des Algorithmus brauchen jeweils, abgesehen von den rekursiven Aufrufen, $O(|E|)$ Zeit. Sei $c$ die entsprechende Konstante in der $O$-Notation. Man kann zeigen, daß die erwartete Laufzeit $T(e)$ insgesamt auch linear in der Anzahl der Kanten $|E|$ ist. Genauer gilt:

**Behauptung 43** *Die Laufzeit beträgt $T(e) \leq 7ce$ , sie ist also tatsächlich linearer in der Kantenanzahl.*

**Beweis** Es sei $|V| = n$, $|E| = e$, $|V'| = n'$ usw. Noch einmal zur Erinnerung:

$$\begin{aligned} |V'| &= n' \leq n/2 \\ |E'| &= e' \leq e - n/2 \leq e - n' \end{aligned}$$

Man muß zwei Fälle unterscheiden:

1. $e'/n' \leq 6$, bzw. $n' \geq e'/6$
   Aus
   $$e' \leq e - n'$$
   folgt dann
   $$e' \leq e - e'/6,$$
   und damit
   $$e' \leq 6/7 \cdot e.$$

Wegen $|E''| = e'' = e' \leq 6/7 \cdot e$ erhält man:

$$\begin{aligned} T(e) &\leq \underbrace{c \cdot e}_{\text{ohne rekursive Aufrufe}} + T(e'') \\ &\leq c \cdot e + \underbrace{T(6/7 \cdot e)}_{\text{IV: } \leq 7c \cdot \frac{6}{7} e} \\ &\leq ce + 6ce \\ &= 7ce. \end{aligned}$$

2. $e'/n' > 6$, bzw. $n' < e'/6 < e/6$
   Sei $\bar{e}$ die erwartete Anzahl von Kanten in $H$. Es gilt $\bar{e} < e/2$. Die erwartete Laufzeit des rekursiven Aufrufs in Schritt 2) ist dann $T(\bar{e})$ ($T$ ist linear). In Schritt 3) wird der Algorithmus dann noch einmal rekursiv auf $(V', E'')$ angewendet. Aus Behauptung 42 folgt:
   $$Ex[e''] \leq 2n' < e'/3 < e/3,$$
   und damit ergibt sich als erwartete Laufzeit:
   $$\begin{aligned} T(e) &\leq ce + T(\bar{e}) + T(e'') \\ &\leq ce + 7c(\bar{e} + e'') \\ &\leq ce + 7c(e/2 + e/3) \\ &= ce(1 + 7 \cdot \frac{5}{6}) \\ &\leq 7ce. \end{aligned}$$

$\sqcup\!\!\!\!\sqcap$

## *********** Ende des ignorierbaren Teils ***********

Vorlesung vom 05.02.1996
Daniel Kästner, Daniel Kreutzer

## 7.10 Maximale Flüsse in Graphen

**Definition 19** *Ein Netzwerk $G = (V, E)$ ist ein gerichteter Graph, in dem jede Kante $[u, v\rangle$ eine nichtnegative Kapazität $c(u, v) \geq 0$ hat. Ist $[u, v\rangle \notin E$, so nehmen wir an, daß $c(u, v) = 0$ ist, also*

$$\begin{array}{rrcl} & c : E & \rightarrow & \mathbb{R}^+ \\ \mathit{mit} & [u, v\rangle & \mapsto & c(u, v) = \begin{cases} c(e) & , \ \mathit{falls} \ [u, v\rangle \in E \\ 0 & , \ \mathit{sonst.} \end{cases} \end{array}$$

*Jedes Netzwerk enthält zwei ausgezeichnete Knoten, eine Quelle $s$ und eine Senke $t$, mit $s, t \in V$. Wir nehmen an, daß jeder Knoten auf einem Pfad von der Quelle zur Senke liegt, d.h. für jeden Knoten $v \in V$ gibt es einen Pfad $s \leadsto v \leadsto t$.*

**Definition 20** *Ein Fluß in einem Graphen $G$ ist eine reellwertige Funktion $f : V \times V \rightarrow \mathbb{R}$ mit folgenden drei Eigenschaften:*

| | | | |
|---|---|---|---|
| *(i)* | *Kapazitätsbedingung:* | $\forall u, v \in V$ : | $f(u, v) \leq c(u, v)$ |
| *(ii)* | *Schiefsymmetrie:* | $\forall u, v \in V$ : | $f(u, v) = -f(v, u)$ |
| *(iii)* | *Flußerhaltung:* | $\forall u \in V \setminus \{s, t\}$ : | $\sum_{v \in V} f(u, v) = 0$ |

*$f(u, v)$ heißt Nettofluß von $u$ nach $v$.*
*Der Wert eines Flusses $f$ ist definiert als*

$$|f| = \sum_{v \in V} f(s, v),$$

*stellt also den gesamten Nettofluß aus der Quelle heraus dar.*

Im Problem des maximalen Flusses ist ein Netzwerk $G$ mit Quelle $s$ und Senke $t$ gegeben und wir wollen einen maximalen Wert für den Fluß von $s$ nach $t$ bestimmen.
Man kann sich das Problem veranschaulichen, wenn man sich zum Beispiel vorstellt, die Quelle sei ein Erdölbohrloch, die Senke sei eine Raffinerie und dazwischen gebe es ein Pipelinenetz, in dem Knotenpunkte mit Röhren unterschiedlicher Dicke verbunden sind. Der Querschnitt eines Rohres ist dann die Kapazität der entsprechenden Kante. Der Fluß $f(u,v)$ entspricht dann dem Volumen Erdöl, das pro Zeiteinheit durch das $u$ und $v$ verbindende Rohr fließt (was bei gleichbleibenden Druck durch die Kapazität beschränkt ist). Flußerhaltung bei den Knoten bedeutet, daß dort kein Öl verloren geht. Und $|f|$ ist dann die Menge Erdöl die das Bohrloch pro Zeiteinheit verläst und, da es nirgends verlorengeht auch bei der Raffinerie ankommt. Das Problem ist dann natürlich, über welche Rohre man wieviel Öl schickt, damit möglichst viel pro Zeiteinheit vom Bohrloch zur Raffinerie kommt.
Neben dieser naiven Anwendung hat das Finden von maximalen Flüssen noch viele andere Anwendungen. Eine schöne Sammlung findet man in dem Buch **Network Flows** von R.K. Ahuja, T.L. Magnanti und J.B. Orlin, Prentice Hall (1993).
Abbildung 7.10 zeigt ein Beispielnetzwerk, bei dem die Kanten mit ihren Kapazitäten und dem Fluß (in eckigen Klammern) markiert sind. (Negative Nettoflüsse zwischen Knoten sind dabei nicht alle gezeigt.)

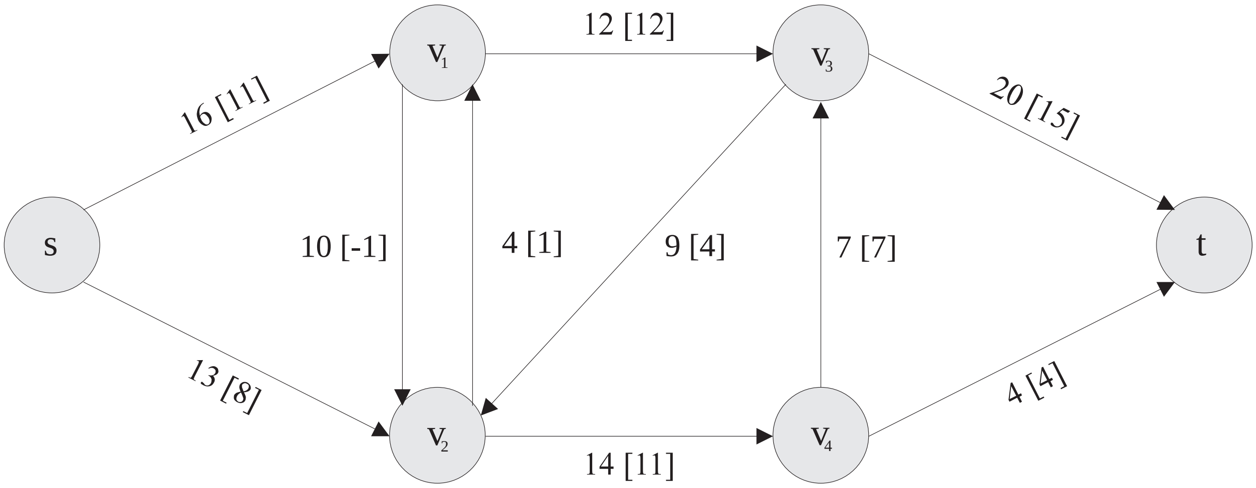



Abbildung 7.10: Kapazitäten und Flüsse in einem Netzwerk

**Folgerung**

1.
$$f(u,v) = -f(u,v) \Rightarrow f(u,u) = 0$$

2.
$$f(u,v) \neq 0 \Rightarrow [u,v\rangle \in E \text{ oder } [v,u\rangle \in E$$

3.
$$\begin{aligned}\sum_{v\in V} f(u,v) = (\sum_{v:f(u,v)>0} f(u,v)) + (\sum_{v:f(u,v)<0} f(u,v)) = 0 \quad &\text{(Flußerhaltung)}\\ \Leftrightarrow \sum_{v:f(u,v)>0} f(u,v) + \sum_{v:f(u,v)<0} -f(v,u) = 0 \quad &\text{(Symmetrie)}\\ \Leftrightarrow \sum_{v:f(u,v)>0} f(u,v) = \sum_{v:f(u,v)<0} f(u,v)\end{aligned}$$

Die Summe $\sum_{u\in V:f(u,v)>0} f(u,v)$ wird als positiver Nettofluß in den Knoten $v$ definiert.
Der einen Knoten verlassende negative Nettofluß wird symmetrisch definiert. Man kann sich dies so vorstellen, daß die Flußerhaltungseigenschaft darin besteht, daß der in einen von Quelle oder Senke verschiedenen Knoten eintretende Fluß genauso groß ist, wie der positive Fluß, der den Knoten verläßt.

**Definition 21** *(Implizite Summationsnotation)*
*Für Mengen von Knoten $X, Y \subset V$ sei*

$$\begin{aligned} f(X,Y) &:= \sum_{x \in X} \sum_{y \in Y} f(x,y), \\ f(x,Y) &:= \sum_{y \in Y} f(x,y). \end{aligned}$$

So können wir z.B. die Flußerhaltung jetzt schreiben als $f(u,V) = 0$ ist für alle $u \in V \setminus \{s,t\}$. Mengenklammern werden innerhalb dieser Notation üblicherweise weggelassen. Man schreibt z.B. $f(s, V - s)$ für $f(s, V \setminus \{s\})$.

**Lemma 14** *Es gelten folgende Regeln für Mengen von Knoten $X, Y \subset V$ :*

*1)* $f(X,X) = 0$

*2) Aus $X, Y \subset V$ folgt $f(X,Y) = -f(Y,X)$.*

*3) Aus $X \cap Y = \emptyset$ folgt:*
$f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$,
$f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

**Beweis** Dies kann durch einfache Umformungen mit Hilfe der Definition gezeigt werden:

zu (a) $f(X,X) = \sum_{x \in X} \sum_{y \in X} f(x,y) = 0$ gilt, da $f(x,x) = 0$ für alle $x$, und für jedes $x \neq y$, die Summe die beiden Terme $f(x,y)$ und $f(y,x)$ enthält, die einander wegen der Schiefsymmetrie streichen.

zu (b)

$$\begin{aligned} f(X,Y) &= \sum_{x \in X} \sum_{y \in Y} f(x,y) \\ &= \sum_{x \in X} \sum_{y \in Y} -f(y,x) \\ &= -\sum_{x \in X} \sum_{y \in Y} f(y,x) \\ &= -f(Y,X) \end{aligned}$$

zu (c)

$$\begin{aligned} f(X \cup Y, Z) &= \sum_{x \in X \cup Y} \sum_{z \in Z} f(x,z) \\ &= \sum_{x \in X} \sum_{z \in Z} f(x,z) + \sum_{y \in Y} \sum_{z \in Z} f(y,z) \\ &= f(X,Z) + f(Y,Z) \end{aligned}$$

$$\begin{aligned} f(Z, X \cup Y) &= \sum_{z \in Z} \sum_{x \in X \cup Y} f(z,x) \\ &= \sum_{z \in Z} \sum_{x \in X} f(z,x) + \sum_{z \in Z} \sum_{y \in Y} f(z,y) \\ &= f(Z,X) + f(Z,Y) \end{aligned}$$

□

**Behauptung 44** $|f| = f(V,t)$

**Beweis** Intuitiv ist dies klar, da bei keinem Knoten nach der Flußerhaltung Fluß verloren gehen kann. Etwas formaler:

$$
\begin{aligned}
|f| &= f(s,V) && \text{(nach Definition des Flusses)}\\
&= f(V,V) - f(V-s,V) && \text{(nach Lemma, Teil c)}\\
&= -f(V-s,V) && \text{(nach Lemma, Teil a)}\\
&= f(V,V-s) && \text{(nach Lemma, Teil b)}\\
&= f(V,t) + f(V,V-s-t) && \text{(nach Lemma, Teil c)}\\
&= f(V,t) && \text{(nach Flußerhaltung)}
\end{aligned}
$$

□

**Definition 22** *Sei ein Netzwerk $G = (V,E)$ mit Quelle $s$ und Senke $t$ gegeben und sei $f$ ein Fluß in $G$. Betrachte ein Knotenpaar $u,v \in V$.*
*Der Wert des zusätzlichen Nettoflusses, den wir von $u$ zu $v$ fließen lassen können, ohne die Kapazität $c(u,v)$ zu überschreiten, heißt Restkapazität von $(u,v)$ und ist gegeben durch*

$$c_f(u,v) = c(u,v) - f(u,v).$$

Es ist ohne weiteres möglich, daß $c_f(u,v) > c(u,v)$, da ein negativer Fluß auftreten kann.
Ausgehend von der Restkapazität können wir nun den Begriff des Restnetzwerkes definieren.

**Definition 23** *Sind ein Netzwerk $G = (V,E)$ und ein Fluß $f$ gegeben, dann ist das von $f$ induzierte Restnetzwerk $G_f$ definiert durch $G_f = (V,E_f)$ mit*

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}.$$

Jede Kante des Restnetzwerkes, Restkante genannt, kann einen positiven Netzfluß tragen. Weiterhin kann einer Kante $[u,v\rangle$ eine Restkante in $E_f$ sein, ohne auch in $E$ enthalten zu sein, also ist möglicherweise $E_f \not\subset E$. Da eine Kante $[u,v\rangle$ in einem Restnetzwerk nur vorkommen kann, wenn $[u,v\rangle$ oder $[v,u\rangle$ im Originalnetz erscheinen, erhalten wir $|E_f| \leq 2|E|$.

Folgende Abbildung zeigt das Restnetzwerk, wie es sich aus dem Netzwerk von Abbildung 7.10 ergibt; die Kantenmarkierungen stellen die Restkapazitäten dar. Kanten des Ausgangsnetzwerkes, die im Restnetzwerk wegfallen, sind gestrichelt gezeichnet.

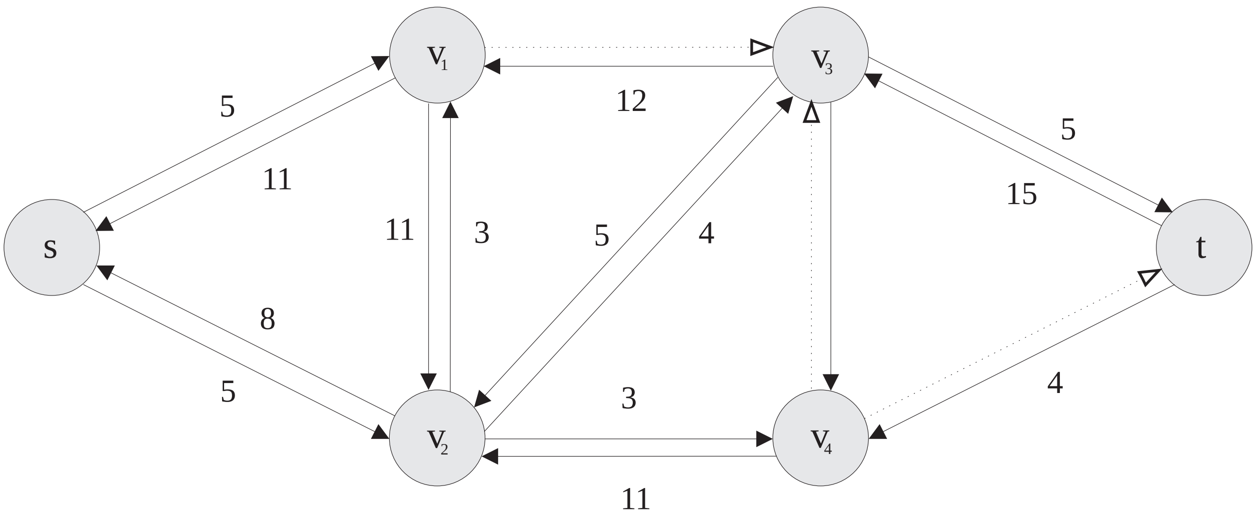



Abbildung 7.11: Restnetzwerk mit Restkapazitäten

**Lemma 15** *Sei $G = (V,E)$ ein Netzwerk mit Quelle $s$, Senke $t$ und sei $f$ ein Fluß in $G$. Sei $G_f$ das von $f$ induzierte Restnetzwerk von $G$ und sei $f'$ ein Fluß in $G_f$. Dann ist die Summe der Flüsse $f + f'$ ein Fluß in $G$ mit Wert $|f + f'| = |f| + |f'|$.*

**Beweis** Zu zeigen ist, daß Schiefsymmetrie, Kapazitätsbedingung und Flußerhaltung erfüllt sind. Für die Schiefsymmetrie erhalten wir für alle $u, v \in V$:

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &= -f(v,u)-f'(v,u)\\ &= -(f(v,u)+f'(v,u))\\ &= -(f+f')(v,u).\end{aligned}$$

Für die Kapazitätsbeschränkung gilt wegen $f'(u,v) \leq c_f(u,v)$ für alle $u, v \in V$:

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &\leq f(u,v)+(c(u,v)-f(u,v))\\ &= c(u,v).\end{aligned}$$

Für die Flußerhaltung erhalten wir für alle $u \in V \setminus \{s,t\}$:

$$\begin{aligned}\sum_{v\in V}(f+f')(u,v) &= \sum_{v\in V}(f(u,v)+f'(u,v))\\ &= \sum_{v\in V} f(u,v)+\sum_{v\in V} f'(u,v)\\ &= 0+0=0.\end{aligned}$$

Damit ergibt sich insgesamt:

$$\begin{aligned}|f+f'| &= \sum_{v\in V}(f+f')(s,v)\\ &= \sum_{v\in V}(f(s,v)+f'(s,v))\\ &= \sum_{v\in V} f(s,v)+\sum_{v\in V} f'(s,v)\\ &= |f|+|f'|.\end{aligned}$$

$\square$

**Definition 24** *Ein augmentierender Pfad $p$ in einem Netz $G = (V, E)$ mit einem Fluß $f$ ist ein einfacher Pfad $s \rightsquigarrow t$ im Restnetz $G_f$. Die Restkapazität von $p$ ist gegeben durch*

$$c_f(p) = \min\{c_f(u,v) : [u,v\rangle \textit{ auf } p\}.$$

*($c_f(p) > 0$, denn das Restnetzwerk enthält nur Kanten mit positiven Kapazitäten.)*

Im Netzwerk der Abbildungen 7.10 bzw 7.11 ergibt sich dann folgender augmentierender Pfad:

**Lemma 16** *Sei $G = (V, E)$ ein Netzwerk, $f$ ein Fluß in $G$ und $p$ ein augmentierender Pfad in $G$. Dann ist die Funktion $f_p : V \times V \to \mathbb{R}$ mit*

$$f_p(u,v) = \begin{cases} c_f(p) & , \textit{ falls } [u,v\rangle \textit{ auf } p \textit{ liegt}\\ -c_f(p) & , \textit{ falls } [v,u\rangle \textit{ auf } p \textit{ liegt}\\ 0 & , \textit{ sonst}\end{cases}$$

*ein Fluß im Restnetzwerk $G_f$ mit dem Wert $|f_p| = c_f(p) > 0$.*

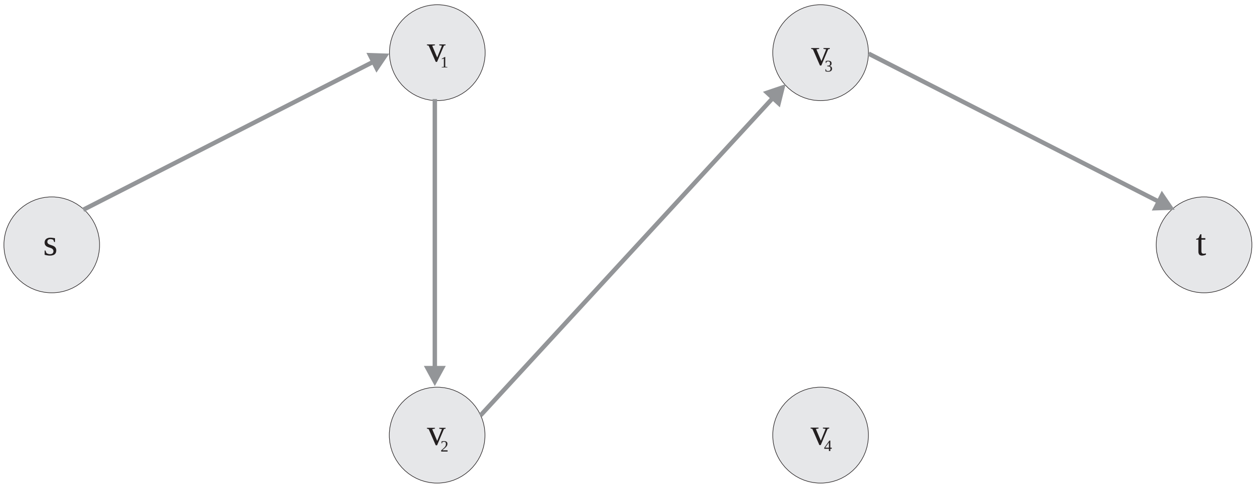



Abbildung 7.12: augmentierender Pfad

**Korollar 4** *Sei $G = (V, E)$ ein Netzwerk, $f$ ein Fluß in $G$, $p$ ein augmentierender Pfad in $G_f$ und $f_p$ wie oben definiert. Dann ist die Funktion $f' : V \times V \to \mathbb{R}$ mit $f' = f + f_p$ ein Fluß in $G$ mit Wert $|f'| = |f| + |f_p| > |f|$.*

Wir nennen das im Korrolar gezeigte Verändern von $f$ zu $f' = f + f_p$ das *Augmentieren von $f$ entlang Pfad $p$*. Auf diesem Augmentieren beruht der **Algorithmus von Ford-Fulkerson**:

(1) FORD-FULKERSON$(G, s, t)$
(2) $f = 0$
(2) **while** es existiert ein augmentierender Pfad $p$ **do**
(3) augmentiere $f$ entlang $p$
(4) **return** $f$

Dieser Algorithmus erhöht mit jeder Iteration den Fluß. Wie viele Iterationen muß er machen? Überraschenderweise ist es bei besonders geschickt gewählten, *reellen* Kapazitäten möglich, daß er unendlich viele Iterationen machen must, also gar nicht terminiert. Beschränkt man sich jedoch auf ganzzahlige Kapazitäten, so ist es klar, daß er immer terminiert, da die Restkapazitäten des Restnetzes, und damit auch die Restkapazität des augmentierenden Pfades, ganzahlig bleiben, und damit der Fluß immer um eine ganze Zahl erhöht wird (genauere Laufzeitanalyse folgt später).

Wir wollen nun versuchen eine obere Schranke für den maximalen Fluß herzuleiten. Wir brauchen so etwas, um zeigen zu können, daß bei Termination des Ford-Fulkerson Algorithmus tatsächlich ein maximaler Fluß erreicht wurde. Aus der Tatsache, daß es da keinen augmentierenden Pfad mehr gibt, folgt dies nicht sofort.

**Definition 25** *Ein <u>s-t-Schnitt</u> $(S, T)$ eines Netzwerkes $G = (V, E)$ ist eine Partition von $V$ in $S$ und $T$, d.h. $S, T \subset V$ und $S \cap T = \emptyset$, $S \cup T = V$ mit $s \in S$ und $t \in T$.*

**Lemma 17** *Sei $f$ ein Fluß in einem Netzwerk $G$ mit Quelle $s$ und Senke $t$. Ferner sei $(S, T)$ ein $s$-$t$-Schnitt von $G$. Dann gilt für den Fluß von $S$ nach $T$:*

$$f(S, T) = |f|$$

**Beweis** Den Spezialfall, daß die Menge $T$ nur das Element $t$ enthält, haben wir oben schon bewiesen: $|f| = f(S, t) = f(S, T)$ für $T = \{t\}$. Nun zum allgemeinen Fall:
Da $T \cup S = V$, kann man für $f(S, V)$ schreiben:

$$f(S, T) + f(S, S) = f(S, V)$$

Durch einige Umformungen, die im wesentlichen auf Anwendung der Definition des Flusses $f$ und den zugehörigen Rechenregeln basieren, erhält man:

$$\begin{aligned} f(S,T) &= f(S,V) - \underbrace{f(S,S)}_{=0} \\ &= f(S-s,V) + f(s,V) \\ &= \sum_{x \in S \setminus \{s\}} f(x,V) \\ &= f(s,V) \qquad \text{,da } f(x,V) = 0 \text{ für } x \in S \setminus \{s\} \\ &= |f|. \end{aligned}$$

$\square$

**Korollar 5** *Der Wert eines Flusses $f$ in einem Netzwerk $G$ ist durch die Kapazität aller Schnitte nach oben beschränkt:*

$$|f| \leq \min\{c(S,T) : (S,T) \textit{ ist } s\text{-}t\textit{-Schnitt } \}$$

**Beweis** Sei $f$ ein Fluß in einem Netzwerk $G$ und sei $(S,T)$ ein Schnitt. Nach dem letzten Lemma gilt:

$$|f| = f(S,T).$$

Dies kann man nun wie folgt umformen und nach oben abschätzen:

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T). \end{aligned}$$

$\square$

**Satz 20** *(Max-Flow-Min-Cut Theorem) Ist $f$ ein Fluß in einem Netzwerk $G = (V,E)$ mit Quelle $s$ und Senke $t$, dann sind die folgenden drei Aussagen äquivalent:*

1. *$f$ ist der maximale Fluß für $G$,*
2. *das Restnetzwerk $G_f$ enthält keinen augmentierenden Pfad,*
3. *$|f| = c(S,T)$ gilt für einen $s$-$t$-Schnitt $(S,T)$*

**Beweis** Wir führen den Beweis durch einen Ringschluß ( 1. ⇒ 2. ⇒ 3. ⇒ 1. ).

- 1. ⇒ 2. (Widerspruchsbeweis) Gäbe es einen augmentierenden Pfad, so könnte man den Wert des Flusses erhöhen. Dann wäre aber $f$ nicht der maximale Fluß.
- 2. ⇒ 3. Die Voraussetzung ist, daß $G$ keinen augmentierenden Pfad enthält, d.h. $G_f$ enthält keinen Pfad von $s$ nach $t$. Sei $S = \{x : x \text{ ist in } G_f \text{ von } s \text{ aus erreichbar } \}$. Dann gilt $s \in S$, aber $t \notin T$ (da es sonst einen augmentierenden Pfad gäbe). Setze $T = V \setminus S$.

  Die Partition $(S,T)$ ist ein Schnitt. Betrachte eine Kante $[u,v\rangle \in E$, die von einer Seite zur anderen verläuft, also $u \in S$ und $v \in T$. Dann ist die Kante nicht in $E_f$. Daraus ergibt sich

  $$c_f(u,v) = 0.$$

  Aus der Definition von $c_f$ folgt dann sofort:

  $$c(u,v) - f(u,v) = 0.$$

Also erhält man:

$$f(u,v) = c(u,v).$$

Mit $u \in S$ und $v \in T$ folgt die Behauptung:

$$f(S,T) = c(S,T).$$

- 3. $\Rightarrow$ 1. Es gilt $|f| \leq c(S,T)$ für alle Schnitte $(S,T)$. Die Bedingung $|f| = c(S,T)$ impliziert also, daß $f$ ein maximaler Fluß ist.

$\square$

Punkt 2 dieses Satzes impliziert also, daß der Ford-Fulkerson Algorithmus bei Termination tatsächlich einen maximalen Fluß berechnet hat.

Es stellt sich nun die Frage, wie man den Algorithmus von Ford-Fulkerson implementieren kann, so daß er auch für reelle Kapazitäten immer terminiert.

Wie schon gesagt, wenn alle Kapazitäten integral sind, d.h. ganze Zahlen, dann benötigt der Algorithmus von Ford-Fulkerson höchstens $|f^*|$ Iterationen, um den maximalen Fluß $f^*$ zu finden. Dies kann leider relativ leicht eintreten, wie das folgende Beispiel verdeutlicht:

**Beispiel**

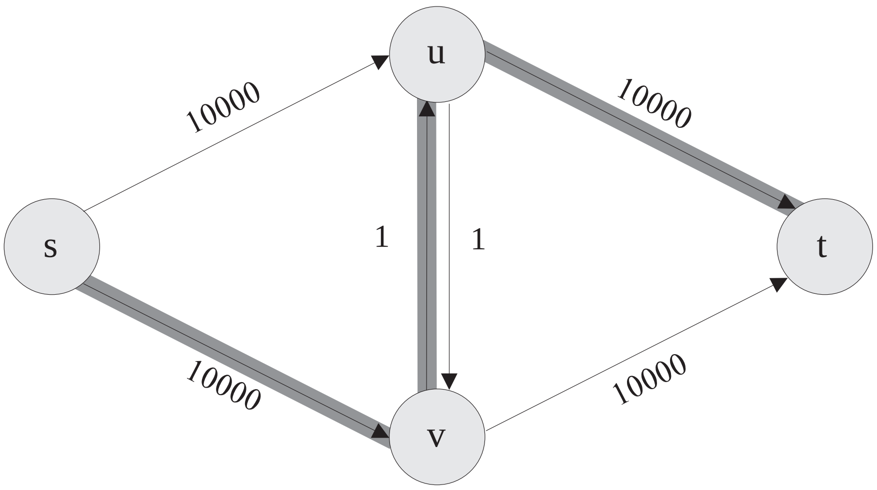



Abbildung 7.13: Ein Netzwerk mit augmentierendem Pfad der Kapazität 1

Es sei das Netzwerk in Abbildung 7.13 gegeben. Die minimale Kapazität entlang des grau unterlegten Pfades ist offensichtlich 1. Wir betrachten nun den in Abbildung 7.14 gezeigten Fluß.
Für diesen Fluß erhält man nach der ersten Iteration des Algorithmus das in Abbildung 7.15 gezeigte Restnetzwerk.
Man erhält letztendlich nach 10000 solchen doppelten Iterationen einen maximalen Fluß von 20000. Betrachtet man nun noch einmal das ursprüngliche Netzwerk in Abbildung 7.13 so wird man feststellen, daß man durch geschickte Wahl des augmentierenden Pfades (hier z.B der Pfad $s \to u \to v$) mit nur zwei Iterationen auch auskommen könnte.

Das Beispiel legt nahe, daß man vielleicht augmentierende Wege verwenden möchte, die möglichst wenig Kanten enthalten. Dies führt tatsächlich zu einer Verbesserung und ist Algorithmus von Edmonds und Karp bekannt.

Vorlesung vom 08.02.1996
René Beier, Henning Biermann, Henrik Theiling

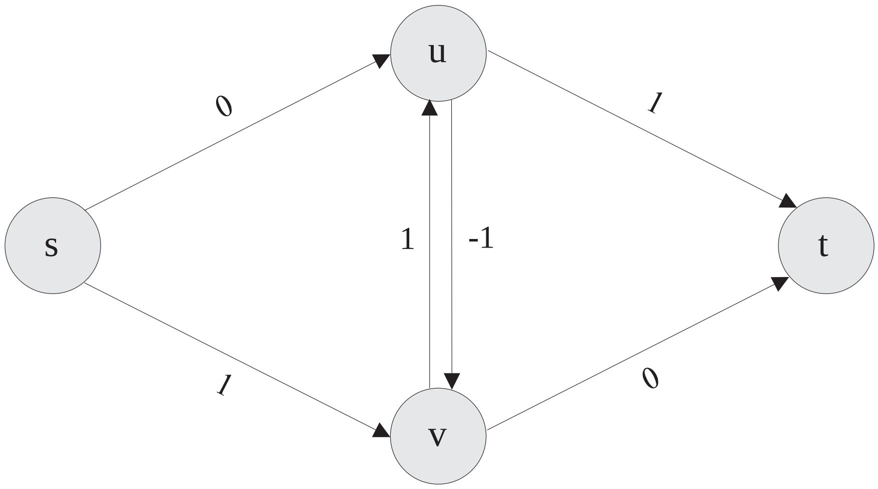



Abbildung 7.14: Ein möglicher Fluß

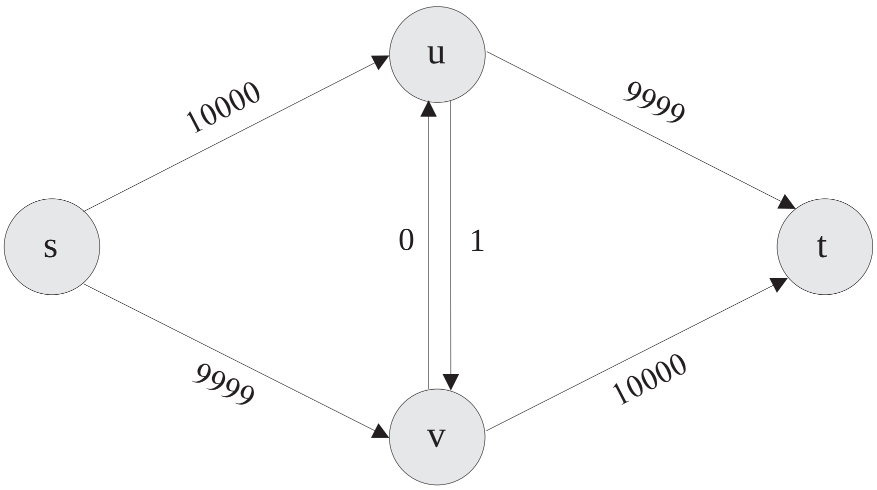



Abbildung 7.15: Restnetzwerk nach einer Iteration

### 7.10.1 Der Edmonds-Karp-Algorithmus

Diese Variation des Ford-Fulkerson Algorithmus skizziert sich folgendermaßen:

Edmonds-Karp$(G, s, t)$
    $f = 0$
    **wiederhole**
        bilde $G_f$
        **falls** ein augmentierender Pfad in $G_f$ existiert
            suche einen augmentierenden Pfad $p$ mit minimaler Kantenzahl
            erhöhe den Fluß entlang $p$
        **sonst**
            brich die Wiederholung ab

#### Laufzeitanalyse

Später wird bewiesen werden, daß der Algorithmus mit dieser Pfadwahlstrategie $O(|V| \cdot |E|)$ Iterationen benötigt, bis der maximale Fluß erreicht wird. Zur Laufzeitbestimmung überlegen wir uns zunächst die Kosten für eine Iteration. Das Restnetzwerk $G_f$ kann man in Zeit $O(|E|)$ bestimmen, indem alle Kanten von $G$ untersucht werden. Den Pfad $p$ findet man mit Breitensuche von $s$ aus in Zeit $O(|E|)$. Die Erhöhung des Flusses entlang $p$ kann ebenso in Zeit $O(|E|)$ erfolgen. Mit der obigen Behauptung ergibt sich so die Gesamtlaufzeit als $O(|V| \cdot |E|^2)$.

#### Abschätzung der Iterationsanzahl

**Definition 26** *Wir bezeichnen den Fluß nach der $i$-ten Iteration im* Edmonds-Karp-*Algorithmus mit $f_i$. Das zugehörige Restnetzwerk heiße $G_i$. Die Entfernung der Quelle $s$ zu einem Knoten $u$ in $G_i$, d. h. die minimale Anzahl der Kanten auf einem gerichteten Weg in $G_i$ von $s$ nach $u$, werde mit $\delta_i(u)$ bezeichnet.*

Der eigentliche Beweis wird über zwei Hilfsbehauptungen geführt.

**Behauptung 45** $\forall i < j : \forall u : \delta_i(u) \leq \delta_j(u)$. *Also kann sich ein $\delta_i(u)$ bei der Ausführung des Algorithmus nur erhöhen ($\infty$ ist auch möglich).*

**Beweis** Es genügt, die Behauptung für $j = i + 1$ zu zeigen. Daraus ergibt sich die allgemeine Aussage induktiv. Wir betrachten jetzt das Restnetzwerk $G_i$ in einer Höhendarstellung wie in Abbildung 7.16. Diese Anordnung

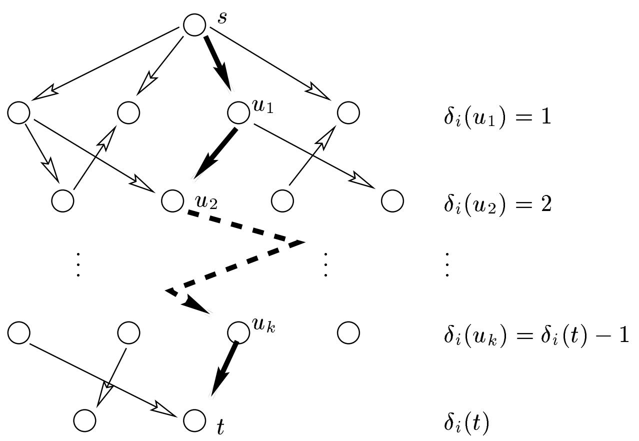



Abbildung 7.16: Restnetzwerk $G_i$ in Höhendarstellung mit augmentierendem Pfad $p$. Die Höhe eines Kontens $u$ gibt seine Entfernung von $s$, d. h. $\delta_i(u)$, an.

der Knoten nach ihrer Entfernung von $s$ kann man sich wie in einem Breitensuchbaum denken.
Was ändert sich jetzt im Restnetzwerk durch die Augmentierung entlang $p$? Für die Kanten $[v, u\rangle$ auf $p$ in $G_i$ werden die Gegenkanten $[u, v\rangle$ ins neue Restnetzwerk aufgenommen; wird $[v, u\rangle$ voll saturiert, so fällt diese

Kante im neuen Restnetzwerk weg. In unserer Höhendarstellung fallen also nur Abwärtskanten auf Pfad $p$ weg, dagegen weisen alle neu dazukommenden Kanten ein Niveau nach oben. Weiterhin werden keine Höhenniveaus übersprungen. Duch das Wegfallen von Abwärtskanten können kürzeste Wege unterbrochen werden (sie bestehen nur aus Abwärtskanten), und durch die neuen Aufwärtskanten ergeben sich keine echten Abkürzungen (man muß zuerst nach unten). Für einen Knoten $u$ verkleinert sich seine Entfernung von $s$ durch das Augmentieren nicht. Es gilt: $\delta_i(u) \leq \delta_{i+1}(u)$. □

**Behauptung 46** *Wird eine Kante $[u,v\rangle$ in Iteration $i$ saturiert und in Iteration $k$ mit $k > i$ erneut saturiert, dann gilt $\delta_k(v) \geq \delta_i(v) + 2$*

**Beweis** Mit der Saturierung in Iteration $i$ wird die Kante $[u,v\rangle$ aus dem Restnetzwerk entfernt. Da sie aber in Iteration $k$ erneut saturiert wird, muß sie durch eine dazwischenliegende Augmentierung wieder ins Restnetzwerk gelangt sein. Also liegt die Kante $[v,u\rangle$ in einer Iteration $l$ auf dem augmentierenden Pfad (und ihre Gegenkante $[u,v\rangle$ gelangt ins Restnetzwerk). Da der augmentierende Pfad ein kürzester Pfad von $s$ nach $t$ ist, gilt folgendes:

$$\delta_l(u) = \delta_l(v) + 1 \overset{\text{Beh. 45}}{\geq} \delta_i(v) + 1.$$

Ganz analog errechnet man für die in Iteration $k$ dazugekommende Kante $[u,v\rangle$

$$\delta_k(v) = \delta_k(u) + 1 \geq \delta_l(u) + 1.$$

Insgesamt erhalten wir

$$\delta_k(v) = \delta_k(u) + 1 \geq \delta_l(u) + 1 \geq \delta_i(v) + 2.$$

□

Mit diesen beiden Behauptungen schätzen wir nun die Iterationszahl ab. Behauptung 46 besagt, daß sich die Entfernung eines Knotens $v$ von $s$ mit jeder Saturierung von $[u,v\rangle$ um mindestens 2 erhöht. Allerdings ist $\delta_i(v)$ nach oben durch $|V|$-1 beschränkt. Deswegen wird eine Kante höchstens $|V|/2$-mal saturiert. Bei jeder Iteration wird mindestens eine Kante saturiert, daher gibt es höchstens $|E| \cdot |V|/2$ Iterationen, die jeweils Zeit $O(|E|)$ brauchen. Damit haben wir folgenden Satz bewiesen:

**Satz 21** *Der Algorithmus von* EDMONDS-KARP *berechnet den maximalen Fluß durch ein Netzwerk $G = (V,E)$ mit der Kapazitätsfunktion $c$ in $O(|V| \cdot |E|^2)$ Zeit.*

## 7.11 Preflow-Push-Algorithmen

Die bisherigen Algorithmen haben bei jedem Rechenschritt alle Flußeigenschaften erhalten. Gibt man diese Erhaltung auf, so kommt man zu einer neuen Klasse von effizienten Algorithmen. Der erste, den wir in diesem Abschnitt vorstellen, hat die Laufzeit $O(|V|^2 \cdot |E|)$ und ist damit i. A. schneller als die vorherigen Algorithmen.

### Intuition

Man kann sich diese Idee an einem Rohrnetzwerk mit Wasser klarmachen, dessen Knoten (Rohrkreuzungen) ein unbeschränktes Vorratsbecken besitzen. Jedem Knoten wird zusätzlich eine Höhe zugeordnet, die sich im Laufe des Algorithmus vergrößern kann. Wasser wird nur bergab durch die Rohre geschoben.
Die Quelle wird zu Beginn auf den Berggipfel gesetzt, und die angeschlossenen Rohre werden voll ausgelastet. Falls nun das Wasser aus noch gefüllten Vorratsbecken nicht weiter abfließen kann, da die angeschlossenen Knoten höher liegen, wird dieses Vorratsbecken soweit wie nötig den Berg hinaufgesetzt.

**Definition 27** *Wir definieren einen* **Quasifluß** *(engl. preflow) durch ein Netzwerk $G = (V,E)$ mit der Kapazitätsfunktion $c$ analog zu einem Netzwerkfluß. Eine Funktion $f : V \times V \to \mathbb{R}$ heißt Quasifluß, wenn sie die folgenden Eigenschaften besitzt.*

$$f(u,v) \leq c(u,v) \tag{7.1}$$
$$f(u,v) = -f(v,u) \tag{7.2}$$
$$f(V,u) \geq 0 \quad \forall u \neq s \tag{7.3}$$

*$f(V,u)$ heißt* **Überschuß** *von $u$ und werde mit $e(u)$ bezeichnet.*

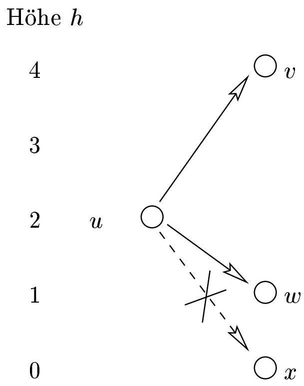



Abbildung 7.17: Kanten $[u,v\rangle$ und $[u,w\rangle \in E_f$ entsprechend der Höhenfunktion $h$ angeordnet. Die Höhe $h(x) = 0$ ist wegen Bedingung (7.6) für die Kante $[u,x\rangle$ nicht erlaubt.

Wie ein Fluß erfüllt ein Quasifluß die Kapazitätsbedingung (7.1) und die Schiefsymmetie (7.2). Dagegen ist Eigenschaft (7.3) eine Abschwächung der Flußerhaltung. Deswegen ist ein Quasifluß i. A. kein Fluß.

**Definition 28** *Eine Funktion $h : V \to \mathbb{N}$ heißt* **Höhenfunktion** *für $G$, falls*

$$\begin{aligned} h(s) &= |V| && (7.4) \\ h(t) &= 0 && (7.5) \\ h(u) &\leq h(v) + 1 \quad \forall [u,v\rangle \in E_f && (7.6) \end{aligned}$$

Bedingung (7.6) besagt, daß Kanten im Restnetzwerk genau ein Höhenniveau nach unten zeigen, gleichbleiben oder beliebig viele Niveaus nach oben weisen (vgl. Abbildung 7.17).
Nun werden zwei Operationen definiert, die auf ein Netzwerk mit Quasifluß und Höhenfunktion angewendet werden. Sie bilden den Kern des ersten Preflow-Push-Algorithmus.

**Operation** $\textsc{Push}(u,v)$

Die Operation $\textsc{Push}(u,v)$ dient dazu, den Überschuß an einem Knoten $u$ entlang einer Abwärtskante $[u,v\rangle$ im Restnetzwerk weiterzuschieben. Sie ist nur für $u \notin \{s,t\}$ andwendbar und wenn folgende Bedingungen erfüllt sind:

$$\begin{aligned} c_f(u,v) &> 0 && (7.7) \\ e(u) &> 0 && (7.8) \\ h(v) &= h(u) - 1. && (7.9) \end{aligned}$$

Zur Durchführung wird bestimmt, wieviel Fluß zusätzlich weitergeleitet werden kann, ohne die Quasiflußbedingungen zu verletzen: $d \leftarrow \min\{e(u), c_f(u,v)\}$. $\textsc{Push}$ läßt sich im Pseudocode folgendermaßen formulieren.

$\textsc{Push}(u,v)$
(1) $d \leftarrow \min\{e(u), c_f(u,v)\}$
(2) $f(u,v) \leftarrow f(u,v) + d$
(3) $f(v,u) \leftarrow -f(u,v)$
(4) $e(u) \leftarrow e(u) - d$
(5) $e(v) \leftarrow e(v) + d$

Implementiert man die Funktionen $f$ und $e$ als Felder, kann man sie in konstanter Zeit auswerten und abändern.

**Behauptung 47** *Angewendet auf einen Quasifluß $f$ und eine Höhenfunktion $h$ hinterläßt* $\textsc{Push}(u,v)$ *unter den obigen Voraussetzungen $f$ als Quasifluß und $h$ als Höhenfunktion.*

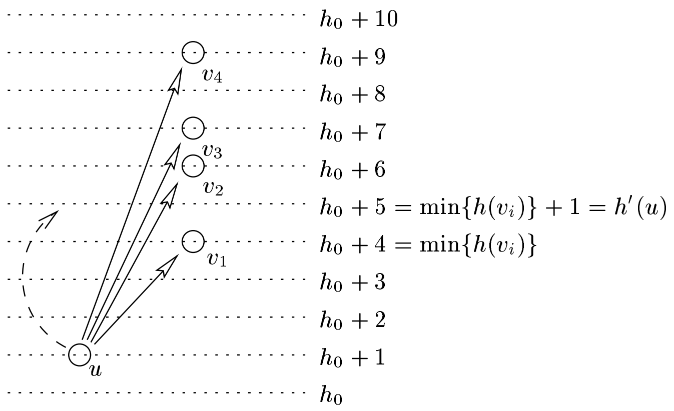



Abbildung 7.18: Schematische Darstellung einer LIFT-Operation. Der Knoten $u$ wird durch LIFT($u$) auf die Höhe $h'(u)$ angehoben.

**Beweis** Durch die Wahl des zusätzlichen Flusses $d$ als $\min\{e(u), c_f(u,v)\}$ werden Kapazitätsschranken- und Überschußbedingung eingehalten. Die Schiefsymmetrie wird ausdrücklich in Zeile (3) hergestellt. Damit ist $f$ weiterhin ein Quasifluß. Es werden bei der Operation keine Werte von $h$ verändert. Wird $[uv, b\rangle$ei PUSH($u, v$) saturiert, fällt sie aus dem Restnetz heraus. In das neue Restnetzwerk wird die Kante $[v, u\rangle$ aufgenommen, wenn sie nicht schon vorher vorhanden war. Ansonsten kommen keine neuen Kanten dazu. Da nun $[u, v\rangle$ nach Voraussetzung eine Abwärtskante ist, zeigt $[v, u\rangle$ nach oben, und es wird gegen keine Bedingung für $h$ verstoßen. □

### Operation LIFT($u$)

Damit PUSH angewendet werden kann, muß das Restnetzwerk Abwärtskanten enthalten. Wir geben nun die Operation LIFT an, mit der man sich solche Kanten erzeugt. LIFT($u$) für einen Knoten $u \in V \setminus \{s, t\}$ setzt voraus, daß

$$e(u) > 0 \tag{7.10}$$
$$h(u) \leq h(v) \quad \forall [u, v\rangle \in E_f. \tag{7.11}$$

Wir können nach dieser Definition nur Knoten $u$ mit Überschuß anheben, die im Restnetzwerk keine Abwärtskanten besitzen (vgl. Abbildung 7.18). Nach Definition der Höhenfunktion sind die Höhen von Quelle $s$ und Senke $t$ nicht veränderbar. Deswegen können sie nicht angehoben werden. Die neue Höhe des Knotens $u$ wird so bestimmt, daß im Restnetzwerk $G_f$ Abwärtskanten auftreten, die höchstens ein Niveau nach unten zeigen.

LIFT($u, v$)
    $h(u) \leftarrow 1 + \min\{h(v) \mid [u, v\rangle \in E_f\}$

**Behauptung 48** *Angewendet auf einen Quasifluß $f$ und eine Höhenfunktion $h$ hinterläßt* LIFT($u$) *unter den obigen Voraussetzungen $f$ als Quasifluß und $h$ als Höhenfunktion.*

**Beweis** $f$ bleibt durch LIFT($u$) unverändert, und $h$ wird an der Stelle $u$ so abgeändert, daß die Höhenbedingungen erfüllt bleibt:

$$\forall [u, v\rangle \in E_f : h(u) \leq 1 + h(v)$$

□

### Initialisierung

Um mit unseren Operationen arbeiten zu können, müssen wir uns einen Quasifluß erzeugen. Dazu setzen wir alle Knoten außer der Quelle $s$ auf die Höhe 0. Die Quelle erhält die Höhe $|V|$. Alle Kanten $[s, u\rangle$, die von $s$ ausgehen, werden entsprechend der maximalen Kapazität $c(s, u)$ ausgenutzt. Dieser Wert ist dann auch der Überschuß an den Knoten $u$. Diese Initialisierung erledigt man wie folgt.

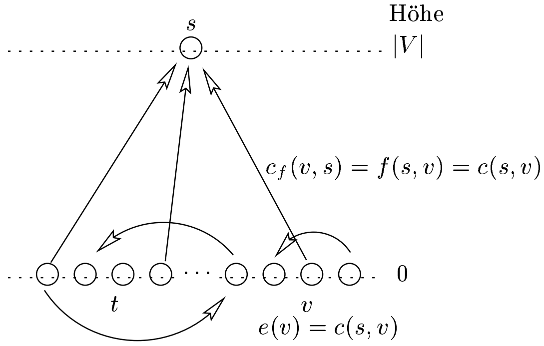



Abbildung 7.19: Restnetzwerk nach Initialisierung. Beispielhaft sind einige Kanten aus $E_f$ auf Höhe 0 angegeben.

INITIALIZE-PREFLOW
    **for each** $u \in V$ **do**
        $h(u) \leftarrow 0$
        $e(u) \leftarrow 0$
    **for each** $[u,v\rangle \in E$ **do**
        $f(u,v) \leftarrow f(v,u) \leftarrow 0$
    $h(s) \leftarrow |V|$
    **for each** $u \in \mathrm{OUT}(s)$ **do**
        $f(s,u) \leftarrow c(s,u)$
        $f(u,s) \leftarrow -f(s,u)$
        $e(u) \leftarrow c(s,u)$

Wenn wir $h$ so setzen, erhalten wir eine gültige Höhenfunktion: Alle Kanten, die von $s$ ausgehen, werden bei der Initialisierung saturiert. Kanten mit Restkapazität verlaufen daher nur zwischen Knoten der Höhe 0. Es treten also keine Abwärtskanten auf (vgl. Abbildung 7.19). Weiterhin werden die Quasiflußeigenschaften von $f$ ausdrücklich hergestellt (Kapazität, Antisymmetrie und abgeschwächte Flußerhaltung).
Nach dieser Vorarbeit können wir den eigentlichen Algorithmus in zwei Zeilen angeben.

Allgemeiner PREFLOW-PUSH-ALGORITHMUS
(1) INITIALIZE-PREFLOW
(2) **solange** es eine anwendbare PUSH- oder LIFT-Operation gibt,
        wähle eine aus und führe sie durch.

In Schritt (2) wird nicht festgelegt, welche Operation angewendet wird. Hier ergeben sich viele Möglichkeiten für Strategien. Bevor wir uns aber verschiedenen Strategien zuwenden beweisen zuerst die Korrektheit dieses Verfahrens. Das geschieht über zwei Hilfsbehauptungen.

**Behauptung 49** *Wenn bei der Ausführung des Algorithmus $e(u) > 0$ gilt, kann entweder* LIFT$(u)$ *oder* PUSH$(u,v)$ *für irgendein $v$ angewendet werden.*

**Beweis** Die Behauptung ergibt sich aus den Voraussetzungen für die beiden Operationen. PUSH setzt voraus, daß es mindestens eine Abwärtskante im Restnetzwerk gibt, LIFT dagegen, daß es keine solche Kante gibt. □

Aus Behauptung 49 folgt unmittelbar, daß bei Terminierung des Algorithmus gilt:

$$\forall u \in V, u \neq t : e(u) = 0$$

Also hat kein Knoten mehr Überschuß; der Quasifluß ist damit ein Fluß.

**Behauptung 50** *Für einen Quasifluß $f$ und eine Höhenfunktion $h$ gibt es keinen Pfad in $G_f$ von $s$ nach $t$.*

**Beweis** Der Beweis wird indirekt geführt. Wir nehmen an, daß es einen solchen Pfad gibt. Im Restnetzwerk zeigt jede Kante höchstens ein Niveau nach unten, da $h$ eine Höhenfunktion ist. Durch die Initialisierung haben $s$ die Höhe $|V|$ und $t$ die Höhe 0, was sich nicht mehr ändern kann. Also führt der Pfad von $s$ nach $t$ über $|V|$ Kanten. Es liegen damit $|V| + 1$ Knoten auf dem Pfad. Widerspruch, da es nur $|V|$ Knoten gibt. □

Die Behauptung 50 besagt insbesondere, daß es nach Terminierung im Restnetzwerk keinen Pfad von $s$ nach $t$ gibt. Mit dem Max-Flow-Min-Cut-Satz folgt dann, daß der gefundene Fluß maximal ist (es gibt keinen augmentierenden Pfad).
Jetzt ist noch zu zeigen, daß der Algorithmus tatsächlich terminiert.

Vorlesung vom 12. Februar 1996
Tim Schwartz, Daniel Wannemacher, Robert Wirth

Wir werden nicht nur Termination, sondern auch gleich eine Zeitschranke zeigen. Diese gilt für den allgemeinen Algorithmus. Danach beschreiben wir noch eine spezielle Version des Algorithmus, mit der man $O(|V|^3)$ Laufzeit erreichen kann.

**Satz 22** *Der allgemeine* PREFLOW-PUSH-ALGORITHMUS *benötigt* $O(|V|^2 \cdot |E|)$ *Zeit.*

Um diesen Satz zu beweisen, sind einige Zwischenüberlegungen notwendig. Wir werden zeigen, daß die drei möglichen Operationen des Algorithmus — LIFT, saturierender PUSH und nicht-saturierender PUSH — höchstens so oft ausgeführt werden müssen, daß die Gesamtlaufzeit diese obere Schranke hat.
Zuerst ein wichtiges Lemma, mit dem die Werte der Höhenfunktion beschränkt werden können.

**Lemma 18** *Wie üblich sei $G$ ein Netzwerk mit Quelle $s$ und sei $f$ ein Quasifluß in $G$.
Ist $u$ ein Knoten aus $G$ mit Überschuß, gilt also $e(u) > 0$, dann gibt es einen Pfad von $u$ nach $s$ im Restnetzwerk $G_f$.*

**Beweis** Der Überschuß im Knoten $u$ ist durch das Netzwerk von $s$ aus dort hingeflossen. Intuitiv ist klar, daß er durch das Restnetzwerk, das Kanten in umgekehrter Richtung enthält, wieder dorthin zurückfließen kann. Für diesen Rückfluß muß es einen Pfad von $u$ nach $s$ im Restnetzwerk geben.
Jetzt formaler: Sei $U$ die Menge der Knoten, die im Restnetzwerk $G_f$ von $u$ aus erreichbar sind, und nehmen wir an, $s$ sei nicht in $U$.
Setze $X = V \setminus U$. Nach Konstruktion führt keine Kante in $G_f$ von einem Knoten in $y \in U$ zu einem Knoten $x \in X$. Das bedeutet nach der Definition von $G_f$, daß für alle solche $y, x$ gelten muß,

$$c_f(y,x) = c(y,x) - f(y,x) = 0\,, \qquad \text{also} \qquad f(y,x) = c(y,x)\,.$$

Da Kapazitäten nicht-negativ sind, folgern wir $f(y,x) \geq 0$ und daher $f(x,y) \leq 0$. Dies gilt für alle $x \in X$ und $y \in U$, also haben wir $f(X,U) \leq 0$.
Betrachten wir jetzt $f(V,U) = \sum_{y \in U} f(V,y) = \sum_{y \in U} e(y)$. Da $f$ ein Quasifluß ist, gilt für die Überschüße bei Knoten $y \neq s$, daß $e(y) \geq 0$. Außerdem gilt $e(u) > 0$ nach Annahme, und deswegen $f(V,U) > 0$, da $u \in U$ und $s \notin U$.
Andererseits haben wir $f(V,U) = f(X,U) + \underbrace{f(U,U)}_{=0} = f(X,U) \leq 0$. Widerspruch. □

Aus diesem Lemma folgt ein für die weitere Argumentation wichtiges Korrolar.

**Korollar 6** *Sei $G = (V,E)$ ein Netzwerk mit Höhenfunktion $h$. Dann gilt*

$$\forall u \in V : h(u) < 2 \cdot |V|$$

**Beweis** Es ist leicht einzusehen, warum diese Beschränkung gelten muß. Wenn ein Knoten $u$ eine Höhe $h$ (durch eine LIFT-Operation) erreicht, hat er Überschuß und damit gibt es nach dem vorherigen Lemma einen Pfad von $u$ nach $s$. Da auf Grund der Eigenschaften von Höhenfunktionen jede Kante auf diesem Pfad höchstens um ein Niveau nach unten führen kann, und dieser Pfad (wenn er einfach gewählt ist) höchstens $|V| - 1$ Kanten enthalten kann, ist es unmöglich, daß $h[u] > h[s] + |V| - 1$. Aber laut Definition der Höhenfunktion hat die Quelle $s \in V$ die Höhe $h(s) = |V|$. Also ist $h[u] > 2|V| - 1$ unmöglich. □

**Behauptung 51** *Die Anzahl der* LIFT*-Operationen für alle Knoten beträgt maximal* $2|V|^2$ *und ihre Durchführung braucht höchstens* $O(|V| \cdot |E|)$ *Zeit.*

**Beweis** Die Behauptung wird nach folgenden Überlegungen klar:

1. Pro Knoten $u \in V$ gibt es $O(|V|)$ LIFT-Operationen, da ein Knoten höchstens eine Höhe von $2 \cdot |V|$ erreichen kann, zu Beginn aber alle Knoten außer der Quelle die Höhe 0 haben.
2. Jede LIFT-Operation braucht $deg(u)$ Zeit, weil sie gerade von allen vom Knoten erreichbaren Nachbarknoten den mit der minimalen Höhe findet.
3. Alle Knoten zusammen brauchen höchstens $O(|V| \cdot |E|)$ Zeit für alle LIFT-Operationen. Auch das ist offensichtlich, denn wenn jeder Knoten $u$ für eine LIFT-Operation $deg(u)$ Zeit braucht, brauchen alle Knoten zusammen für je eine LIFT-Operation $O(|E|)$ Zeit, weil jede Kante genau einmal zu einer Minimumsbestimmung benötigt wird. Da es pro Knoten höchstens $O(|V|)$ LIFT-Operationen gibt, folgt, daß die Gesamtzeit $O(|V| \cdot |E|)$ ist.

□

**Behauptung 52** *Die Anzahl der saturierenden* PUSH*-Operationen ist insgesamt*

$$O(|V| \cdot |E|)$$

**Beweis** Zunächst muß man sich klarmachen, warum die Anzahl der saturierenden PUSH-Operationen pro Kante höchstens $|V|$ ist:
Nach einem saturierenden PUSH wird eine Kante $[u, v\rangle$, über die der PUSH erfolgte, aus dem Netzwerk genommen und durch eine Kante in Gegenrichtung ersetzt. Damit wieder ein PUSH von $u$ nach $v$ erfolgen kann, muß zunächst über diese Gegenkante wieder ein Fluß von $v$ nach $u$ erfolgen, der diese Gegenkante entfernt und eine neue Kante $[u, v\rangle$ einfügt. Damit über die Gegenkante ein Fluß erfolgen kann, muß $v$ angehoben werden, ebenso anschließend wieder $u$ für den erneuten Fluß über die neue Kante $[u, v\rangle$. Insgesamt wird $u$ also für jeden erneuten saturierenden PUSH zwei Ebenen hochgehoben. Da jeder Knoten höchstens die Höhe $2 \cdot |V|$ erreichen kann, sind also maximal $O(|V|)$ PUSH-Operationen pro Kante möglich.
Insgesamt sind für alle $|E|$ Kanten des Netzwerkes dann $O(|V| \cdot |E|)$ saturierende PUSH-Operationen möglich.
□

**Behauptung 53** *Die Anzahl der nicht-saturierenden* PUSH*-Operationen beträgt insgesamt*

$$O(|V|^2 \cdot |E|)$$

**Beweis** Zum Beweis führen wir folgende Potentialfunktion ein:

$$\Phi = \sum_{\substack{v:e(v)>0 \\ v \neq s,t}} h(v)$$

- Anfangs gilt: $\Phi = 0$
- LIFT-Operationen: Pro Knoten $v$ kann $\Phi$ höchstens um $2 \cdot |V|$ erhöht werden. (siehe Korrolar 6 )

  Insgesamt kann damit $\Phi$ höchstens um $2 \cdot |V|^2$ anwachsen.
- saturierende PUSH-Operationen: Eine solche Operation PUSH$(u, v)$ kann $\Phi$ um höchstens $2 \cdot |V|$ erhöhen, weil $v$ Überschuß bekommt und damit mit seiner Höhe, die laut Korollar 6 durch $2 \cdot |V|$ beschränkt ist, zur Summe in $\Phi$ beiträgt.

  Insgesamt wächst dadurch das Potential um höchstens $O(|V|^2 \cdot |E|)$.
- nicht-saturierende PUSH-Operationen : Da der Knoten $u$ bei einem solchen PUSH$(u, v)$ seinen Überfluß verliert $(e(u) = 0)$ wird dadurch $h(v)$ von $\Phi$ weggenommen. Der Knoten $v$ bekommt Überschuß und erhöht dadurch möglicherweise $\Phi$ um $h(u)$. Aber es gilt $h(v) = h(u) - 1$, also wird das Potential $\Phi$ mindestens um eins vermindert.

- Das Anfangspotential ist 0. Für das Endpotential gilt $\Phi \geq 0$; der Gesamtanstieg von $\Phi$ ist $O(|V|^2 \cdot |E|)$. Dies bedeutet, daß auch die Gesamtreduzierung höchstens $O(|V|^2 \cdot |E|)$ ist. Da nur nicht-saturierende PUSH-Operationen Potentialreduzierung verursachen, and dann jedesmal mindestens um eins, folgt daraus, daß die Anzahl der nicht-saturierenden Push-Operationen $O(|V|^2 \cdot |E|)$ ist.

□

**Beweis** zu Satz 22: Da der Algorithmus aus der Anwendung dieser drei Operationen besteht und da die Durchführung eine LIFT-Operation nur konstante Zeit braucht folgt der Satz us den Behauptungen 51, 52 und 53. □

### 7.11.1 Eine spezielle Strategie für PREFLOW-PUSH:

Gesucht ist jetzt natürlich noch eine möglichst effiziente Anwendungsstrategie für PUSH und LIFT. Aus dem Beweis von Behauptung 53 kann man sehen, daß es günstig sein kann, PUSH-Operationen bei Knoten mit großer Höhe durchzuführen, da dies das dort genannte Potential $\Phi$ um das meiste verringert.
Wir werden nun so eine Strategie betrachten und analysieren. Zuerst einmal ein kleines "Unterprogramm":

ENTLEERE($u$) (Vorbedingung: $e(u) > 0$)
 **while** $e(u) > 0$ und $\exists[u, v\rangle \in E_f$ mit $h(v) = h(u) - 1$
  **do** PUSH($u, v$)

Die Funktion ENTLEERE($u$) versucht also den gesamten Überschuß von $u$ so weit wie möglich abzubauen. Beachten Sie, daß diese Funktion höchstens einen nicht-saturierenden PUSH auslösen kann, weil nach diesem kein Überfluß mehr vorhanden ist, d.h. $e(u) = 0$, und daher kein weiterer PUSH mehr erfolgt.
Wir können jetzt unsere Spezialisierung des Preflow-Push Algorithmus beschreiben:

OBEN-NACH-UNTEN-PREFLOW-PUSH-ALGORITHMUS
 INITIALIZE-PREFLOW
 **repeat**
  **for** $h = 2 \cdot |V| - 1$ **downto** 1 **do**
   ENTLEERE alle Knoten $v$ mit $e(v) > 0$ und $h(v) = h$
  **for each** $v$ mit $e(v) > 0$ **do** LIFT($v$)
 **until** kein LIFT mehr durchgeführt wurde

In der ersten **for**-Schleife wird Überschuß von "oben" nach unten geschoben. Da durch die PUSH-Operationen bei einem Höhenwert $h$ nur Überfluß bei Knoten mit Höhe $h - 1$ entstehen kann, sieht man, daß nach einer Beendigung der ersten **for**-Schleife bei keinem der Knoten mit Überschuß eine PUSH-Operation anwendbar ist. Wenn also dann auch keine LIFT-Operation anwendbar ist, bedeutet dies, daß es keinen Knoten mit Überschuß gibt, was bedeutet, daß unser Algorithmus korrekt mit einem maximalen Fluß terminiert hat.
Der OBEN-NACH-UNTEN-PREFLOW-PUSH-ALGORITHMUS ist eine Spezialisierung des allgemeinen Preflow-Push Algorithmus. Also wissen wir von den vorangegangenen Behauptungen, daß insgesamt nur $O(|V|^2)$ LIFT-Operationen und $O(|V| \cdot |E|)$ saturierende PUSH-Operationen durchgeführt werden, mit einem Gesamtaufwand von $O(|V| \cdot |E|)$.
Wir wollen nun zeigen, daß die Gesamtanzahl der nicht-saturierenden PUSH-Operationen $O(|V|^3)$ ist, was bedeutet, daß die Gesamtlaufzeit von OBEN-NACH-UNTEN-PREFLOW-PUSH-ALGORITHMUS durch $O(|V|^3)$ beschränkt ist.
Wie schon gesagt, gibt es pro ENTLEERE-Operation höchstens einen nicht-saturierenden PUSH. Da jeder Knoten in $V$ in einer Iteration der **repeat**-Schleife höchstens einmal einer ENTLEERE-Operation unterworfen wird, gibt es daher pro Iteration höchstens $|V|$ saturierende PUSH-Operationen. Die Anzahl der Iterationen der **repeat**-Schleife ist aber $O(|V|^2)$, da jede Iteration (außer der letzten) mit mindestens einer LIFT-Operationen endet, aber insgesamt nur $O(|V|^2)$ LIFT-Operationen durchgeführt werden können. Also ist die Gesamtanzahl der nicht-saturierenden PUSH-Operationen $O(|V|^3)$.
Zusammenfassend erreichen wir so den letzten Satz der Vorlesung:

**Satz 23** *Der* OBEN-NACH-UNTEN-PREFLOW-PUSH-ALGORITHMUS *berechnet den maximalen Fluß für ein Netzwerk in Zeit* $O(|V|^3)$.