Skript zur Vorlesung „Datenstrukturen“

Prof. Dr. Georg Schnitger

SS 2015

Hinweise auf Fehler und Anregungen zum Skript bitte an

besser@thi.informatik.uni-frankfurt.de oder
seiwert@thi.informatik.uni-frankfurt.de

Mit einem Stern gekennzeichnete Abschnitte werden in der Vorlesung nur kurz angesprochen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Ein *abstrakter Datentyp* besteht aus einer Sammlung von Operationen auf einer Objektmenge und entspricht in etwa einer Klasse in einer objekt-orientierten Programmiersprache wie C++ oder Java. Eine *Datenstruktur* ist eine Implementierung eines abstrakten Datentyps und kann als Implementierung der entsprechenden Klasse aufgefasst werden.

Warum interessieren wir uns für abstrakte Datentypen und ihre Implementierungen? In vielen Algorithmen treten dieselben Operationen für verschiedenste Datenmengen auf, und wir möchten versuchen, diese Operationen in möglichst allgemeiner Formulierung ein und für alle Mal *möglichst effizient* zu implementieren. Zum Beispiel möchte man oft Schlüssel einfügen und, nach ihrem Alter geordnet, entfernen: Wenn wir stets den jüngsten Schlüssel entfernen möchten, dann ist die Datenstruktur „Stack“ zu empfehlen, ist hingegen stets der älteste Schlüssel zu entfernen, dann bietet sich die Datenstruktur „Schlange“ an. In anderen Situationen, zum Beispiel in der Modellierung von Prioritätswarteschlangen, sind möglicherweise Schlüssel mit zugeordneten Prioritäten einzufügen und der Schlüssel mit jeweils höchster Priorität ist zu entfernen. Für diesen Anwendungsfall werden wir mit „Heaps“ eine im Vergleich zu Stacks und Schlangen etwas kompliziertere Datenstruktur entwickeln.

In den obigen Anwendungsfällen ist der konkrete Datentyp nicht von Interesse, sondern allein die zu implementierenden Operationen, also in unseren Fällen das Einfügen und Entfernen von Schlüsseln nach verschiedenen Kriterien. Das Vernachlässigen der konkreten Datentypen und damit die Betonung des abstrakten Datentyps bedeutet natürlich eine wesentliche vergrößerte Anwendbarkeit und ist der Grund der Betonung abstrakter Datentypen.

Wann können wir sagen, dass eine Datenstruktur eine gute Implementierung eines abstrakten Datentyps ist? Wir haben bereits erwähnt, dass eine Datenstruktur eine möglichst effiziente Implementierung eines vorgegebenen abstrakten Datentyps sein sollte und es liegt nahe, Effizienz durch den *Speicherplatzverbrauch*, also die Größe der Datenstruktur, und durch die *Zugriffszeiten*, also die Schnelligkeit der Implementierungen der einzelnen Operationen zu messen. Methoden zur Speicherplatz- und Laufzeitmessung führen wir deshalb in Kapitel 3 ein.

Wir beginnen in Kapitel 4 mit den bereits aus der Veranstaltung „Grundlagen der Programmierung“ bekannten elementaren Datenstrukturen Liste, Stack, Schlange und Prioritätswarteschlange. Desweiteren stellen wir verschiedene Darstellungen für Bäume und Graphen vor und bestimmen ihre Vor- und Nachteile. Alle Implementierungen werden in C++ beschrieben.

Das Wörterbuch ist ein abstrakter Datentyp mit den Operationen „Einfügen, Entfernen und Suche“. Das Wörterbuch ist der vielleicht wichtigste abstrakte Datentyp der Informatik und wird neben der Verwaltung von Daten auch zum Beispiel in der Implementierung von Peer-to-Peer Systemen benutzt: Benutzer treten einem Peer-to-Peer Sytem bei, melden sich ab oder

suchen nach Daten. Wir werden in Kapitel 5 verschiedenste Wörterbuch-Datenstrukturen, nämlich *binäre Suchbäume, AVL-Bäume, Splay-Bäume,* (a, b)*-Bäume* und *Hashing-Verfahren*, mit ihren jeweils individuellen Stärken und Schwächen kennenlernen.

Zusammengefasst, die angestrebten Lernziele sind:

- die Kenntnis grundlegender abtrakter Datentypen und ihrer Datenstrukturen,
- die Fähigkeit, gute Datenstrukturen für neue abstrakte Datentypen zu entwickeln und ihre Effizienz beurteilen zu können. Insbesondere spielt der Zusammenhang zwischen einem Algorithmus und seiner effizienten Implementierung durch Datenstrukturen eine wichtige Rolle.

Die folgenden Texte vertiefen und ergänzen das Skript:

- T.H. Cormen, C.E. Leiserson, R.L. Rivest und C. Stein, „Introduction to Algorithms“, second edition, MIT Press, 2009. (Kapitel 3-5, 10-12 und 18)
- M. T. Goodrich, R. Tamassia und D. M. Mount, „Data Structures and Algorithms in C++“, John Wiley, 2011.
- M. Dietzfelbinger, K. Mehlhorn und P. Sanders, „Algorithmen und Datenstrukturen, die Grundwerkzeuge“, Springer Vieweg 2014. (Kapitel 1-4,6,8,9)
- R. Sedgewick, „Algorithmen in C++“, Pearson Studium, 2002. (Kapitel 1-7)
- J. Kleinberg und E. Tardos, „Algorithm Design“, Addison-Wesley, 2005. (Kapitel 1-3)
- N. Schweikardt, Diskrete Modellierung, eine Einführung in grundlegende Begriffe und Methoden der Theoretischen Informatik, 2013. (Kapitel 2,5)

Welches Vorwissen ist für das Verständnis des Skriptes notwendig, bzw. von Vorteil?

1. Mengen, Relationen und Funktionen sind wesentliche Bestandteile der „Sprache“ des Skripts und werden in Abschnitt 2.1 behandelt.
2. Eine Hauptaufgabe des Studiums der Informatik ist die Vermittlung der Fähigkeit, funktionierende komplexe Systeme zu entwickeln. Wie aber stellt man sicher, dass das System *immer* funktioniert? Man beweist, dass das System korrekt ist. Was ist ein Beweis, bzw. was ist ein schlüssiges Argument? In Abschnitt 2.3 werden verschiedene Beweismethoden zusammengestellt, wobei die vollständige Induktion eine besonders wichtige Rolle einnimmt, da man mit ihrer Hilfe rekursive Programme als korrekt nachweisen kann.
3. In der Schatztruhe 2.4 sind wichtige Identitäten gesammelt, die für die Laufzeitanalyse häufig benötigt werden. Hier finden sich Notationen, Summenformeln, Rechenregeln für den Logarithmus und für Binomialkoeffizienten, Anzahlbestimmung von Permutationen, bzw. Teilmengen und verschiedes mehr.
4. Grundbegriffe für Bäume und Graphen werden in Abschnitt 2.2 zusammengestellt.
5. Einige elementare Begriffe aus der Stochastik wie etwa Elementarereignisse, Ereignisse, Zufallsvariable, Rechnen mit Wahrscheinlichkeiten, Erwartungswert, Binomialverteilung und geometrische Verteilung finden sich im Abschnitt 2.5

Ein entsprechendes Vorwissen aus der Veranstaltung „Diskrete Modellierung“ ist bis auf die Schatztruhe und die Grundbegriffe aus der Stochastik völlig ausreichend.

Kapitel 2

Mathematische Grundlagen

2.1 Mengen, Relationen und Funktionen

(Dieser Abschnitt ist eine Kurzfassung der Abschnitte 2.1,2.2 und 2.3 des Skripts „Diskrete Modellierung“ von Nicole Schweikardt.)

Wir schreiben

$$m \in M$$

um auszusagen, dass M eine Menge ist und dass m ein Element in der Menge M ist. Wir schreiben

$$m \notin M$$

um auszusagen, dass m kein Element in der Menge M ist.

Was genau ist eine Menge, bzw. wie sind Mengen zu beschreiben? Wir beschreiben, bzw. definieren Mengen

- *extensional*, durch Aufzählen der Elemente, z.B.

  $$M_1 := \{0, 1, 2, 3, 4, 5\} = \{0, 1, 2, \ldots, 5\}$$

  oder

- *intensional*, durch Angabe von charakteristischen Eigenschaften der Elemente der Menge, z.B.

  $$\begin{aligned} M_2 &:= \{x : x \in M_1 \text{und } x \text{ ist gerade } \} \\ &= \{x \in M_1 : \ x \text{ ist gerade } \} \\ &= \{x : \ x \text{ ist eine natürliche Zahl und } x \text{ ist gerade und } 0 \leq x \leq 5 \ \}. \end{aligned}$$

Extensional lässt sich die Menge M_2 folgendermaßen beschreiben:

$$M_2 \ = \ \{0, 2, 4\}.$$

Oft schreibt man statt „:“ auch „ | “ und statt „und“ einfach ein „Komma“, also

$$M_2 = \{x \ : \ x \in M_1, \, x \text{ gerade } \}.$$

Vorsicht:

(a) $\{\, x \,:\, 0 \leq x \leq 5 \,\}$ definiert nicht eindeutig eine Menge, weil nicht festgelegt ist, ob x beispielsweise eine ganze Zahl oder eine reelle Zahl ist.

(b) Wenn wir Mengen „ohne Sinn und Verstand“ bilden, geraten wir in Widersprüche. Ein Beispiel ist die Russelsche Antinomie: Angenommen

$$R = \{M \,:\, M \text{ ist eine Menge und } M \notin M\}$$

ist eine Menge. Wir unterscheiden zwei Fälle.

Fall 1: $R \in R$. Dann kann R kein Element von R sein, da R nach Definition nur aus Mengen besteht, die sich *nicht* selbst enthalten. $\lightning$

Fall 2: $R \notin R$. Dann, nach Definition von R, ist R ein Element von R. $\lightning$

In beiden Fällen geraten wir in einen Widerspruch: R kann keine Menge sein!

Fazit: Um solche Probleme zu vermeiden, sollte man bei intensionalen Mengendefinitionen immer angeben, aus welcher anderen Menge die ausgewählten Elemente kommen sollen, also:

$$\{\, x \in M \,:\, x \text{ hat Eigenschaft(en) } E \,\},$$

wobei M eine Menge und E eine Eigenschaft oder eine Liste von Eigenschaften ist, die jedes einzelne Element aus M haben kann oder nicht. In der Veranstaltung „Diskrete Modellierung“ beantworten wir die Frage nch der „Definition“ von Mengen durch Angabe der Axiome der Zermelo-Fraenkel Mengenlehre (kurz: `ZF`).

Übrigens garantieren die Axiome von `ZF`, dass keine Menge sich selbst entält. Macht das Sinn? Wenn sich die Menge M selbst enthält, wenn also $M = \{M, \ldots\}$ gilt, dann können wir diese Gleichheit einsetzen und erhalten $M = \{\{M, \ldots\}, \ldots\}$. Auch diese, und alle folgenden Gleichheiten können wir einsetzen und erhalten eine *nie endende* Rekursion.

Wichtige grundsätzliche Eigenschaften von Mengen:

- Alle Elemente einer Menge sind verschieden. D.h. ein Wert ist entweder Element der Menge oder eben nicht — aber der Wert kann nicht „mehrfach“ in der Menge vorkommen.
- Die Elemente einer Menge haben keine feste Reihenfolge.
- Dieselbe Menge kann auf verschiedene Weisen beschrieben werden, z.B.

$$\{1, 2, 3\} = \{1, 2, 2, 3\} = \{2, 1, 3\} = \{i \,:\, i \text{ ist eine ganze Zahl, } 0 < i \leq 3 \,\}.$$

- Mengen können aus „atomaren“ oder aus zusammengesetzten Elementen gebildet werden. Die Menge kann auch „verschiedenartige“ Elemente enthalten:

  Die Menge

$$M := \{\, 1,\ (\text{Pik}, 8),\ \{\text{rot}, \text{blau}\}, 5, 1 \,\}$$

  besteht aus 4 Elementen: dem atomaren Wert 1, dem Tupel $(\text{Pik}, 8)$, der Menge $\{\text{rot}, \text{blau}\}$ und dem atomaren Wert 5.

Die leere Menge verdient ihr eigenes Symbol.

Definition 2.1 **(Leere Menge)**

Die leere Menge ist die Menge, die keine Elemente enthält. Wir bezeichnen sie mit $\emptyset$.

Notation: Für eine endliche Menge M bezeichnen wir die Anzahl der Elemente von M mit $|M|$.

2.1.1 Mengenalgebra

In diesem Abschnitt werden einige grundlegende Operationen auf Mengen betrachtet. Zuerst, wann sollte man zwei Mengen *gleich* nennen?

Definition 2.2 **(Gleichheit von Mengen)**

Zwei Mengen M und N sind gleich (kurz: $M = N$), falls sie dieselben Elemente enthalten, d.h. falls gilt:

(a) für alle $x \in M$ gilt $x \in N$, und

(b) für alle $x \in N$ gilt $x \in M$.

Beachte: $\emptyset \neq \{\emptyset\}$, denn $\emptyset$ ist die Menge, die keine Elemente enthält, während $\{\emptyset\}$ eine Menge ist, die ein Element, nämlich $\emptyset$ enthält.

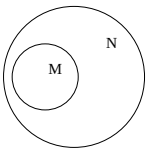
Wie zeigt man, dass zwei Mengen M, N gleich sind? Sehr häufig ist das folgende Rezept erfolgreich: Zeige, dass M in N und dass N in M enthalten ist.

Definition 2.3 **(Teilmengen, Obermengen)**

Seien M, N Mengen.

(a) M ist eine **Teilmenge** von N (kurz: $M \subseteq N$), wenn jedes Element von M auch ein Element von N ist.

Skizze:



(b) M ist eine **echte Teilmenge** von N (kurz: $M \subset N$), wenn $M \subseteq N$ und $M \neq N$.

(c) M ist eine **Obermenge** von N (kurz: $M \supseteq N$), wenn $N \subseteq M$.

(d) M ist eine **echte Obermenge** von N (kurz: $M \supset N$), wenn $M \supseteq N$ und $M \neq N$.

Wir führen als nächstes grundlegende Operationen auf Mengen ein.

Definition 2.4 **(Mengenoperationen)**

Seien M und N Mengen.

(a) Der **Durchschnitt** $M \cap N$ von M und N ist die Menge

$$M \cap N := \{x : x \in M \text{ und } x \in N \}.$$

(b) Die **Vereinigung** $M \cup N$ von M und N ist die Menge

$$M \cup N := \{x : x \in M \text{ oder } x \in N \}.$$

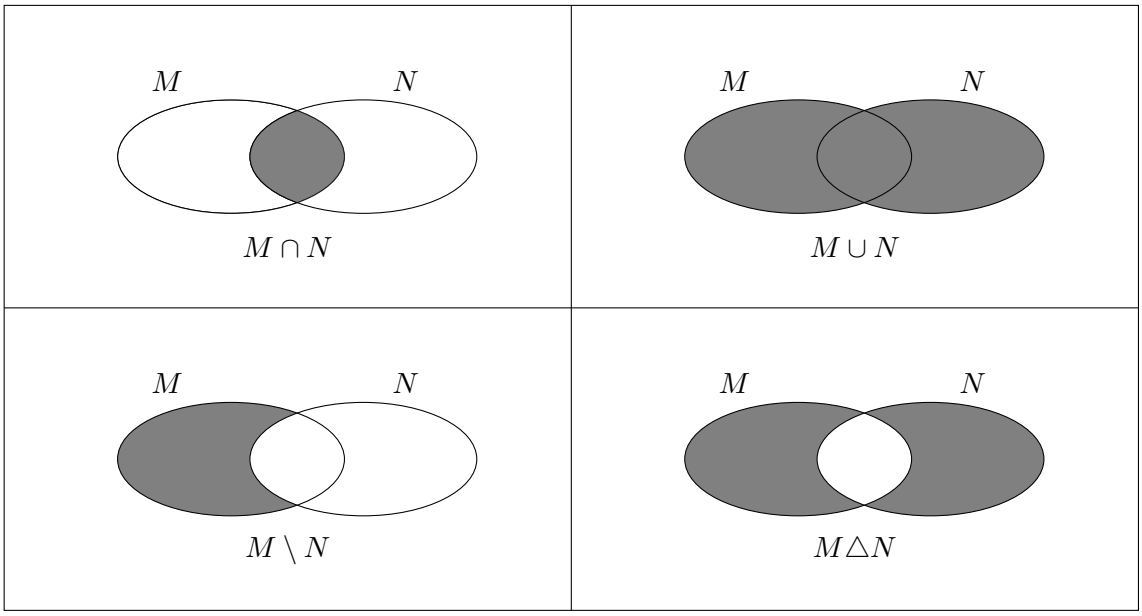
(c) Die **Differenz** $M \setminus N$ von M und N ist die Menge

$$M \setminus N := M - N := \{x : x \in M \text{ und } x \notin N \}.$$

(d) Die **symmetrische Differenz** $M \triangle N$ von M und N ist die Menge

$$M \triangle N := (M \setminus N) \cup (N \setminus M).$$

Veranschaulichung durch Venn-Diagramme:



Um die Komplementmenge zu definieren, legen wir zuerst ein Universum U fest und fordern, dass U eine Menge ist. Für eine Teilmenge $M \subseteq U$ des Universums definieren wir das Komplement von M durch

$$\overline{M} := U \setminus M.$$

Satz 2.1 *Die De Morganschen Regeln.*

Für ein Universum U und Mengen $M, N \subseteq U$ gilt

$$\overline{M \cap N} = \overline{M} \cup \overline{N} \qquad \textit{und} \qquad \overline{M \cup N} = \overline{M} \cap \overline{N}.$$

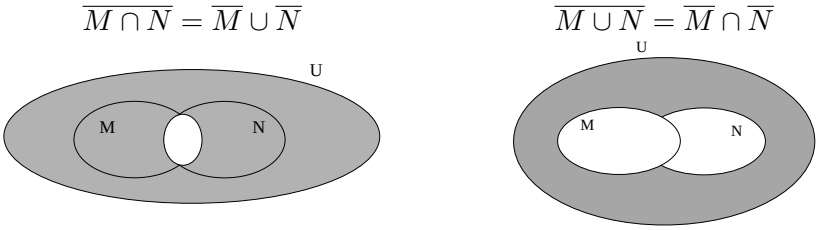
Beweis: Wir beschränken uns auf die erste Gleichung. Wir zeigen zuerst $\overline{M \cap N} \subseteq \overline{M} \cup \overline{N}$ und dann $\overline{M \cap N} \supseteq \overline{M} \cup \overline{N}$.

(a) Sei $x \in \overline{M \cap N}$. Wir müssen zeigen, dass $x \in \overline{M} \cup \overline{N}$ gilt.

Nach Definition der Komplementmenge ist x ein Element des Universums U, gehört aber nicht zum Durchschnitt $M \cap N$. Ohne Beschränkung der Allgemeinheit gehört x nicht zur Menge M. Also folgt $x \in U \setminus M = \overline{M} \subseteq \overline{M} \cup \overline{N}$.

(b) Sei $x \in \overline{M} \cup \overline{N}$. Wir müssen zeigen, dass $x \in \overline{M \cap N}$ gilt.

Ohne Beschränkung der Allgemeinheit können wir annehmen, dass $x \in \overline{M}$ gilt. D.h. es ist $x \in U$, aber x ist kein Element von M. Aber dann gehört x erst recht nicht zum Durchschnitt $M \cap N$ und $x \in U \setminus (M \cap N) = \overline{M \cap N}$ folgt. □



Definition 2.5 Die **Potenzmenge** (engl.: power set) einer Menge M (kurz: $\mathcal{P}(M)$) ist die Menge aller Teilmengen von M. D.h.:

$$\mathcal{P}(M) := \{\, X \,:\, X \subseteq M \,\}.$$

Beispiel:

- $\mathcal{P}(\{a, b\}) = \{\, \emptyset, \{a\}, \{b\}, \{a, b\} \,\}$.
- $\mathcal{P}(\{1, 2, 3\}) = \{\, \emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \,\}$.
- $\mathcal{P}(\emptyset) = \{\, \emptyset\}$. Insbesondere gilt: $\mathcal{P}(\emptyset) \neq \emptyset$.

2.1.2 Paare, Tupel und kartesische Produkte

Definition 2.6 **(Paare und Tupel)**

(a) Für beliebige Objekte a und b bezeichnet (a, b) das **geordnete Paar** mit Komponenten a und b.

(b) Für eine natürliche Zahl k und beliebige Objekte $a_1, \ldots, a_k$ bezeichnet $(a_1, \ldots, a_k)$ das **k-Tupel** mit Komponenten $a_1, \ldots, a_k$. Wir bezeichnen Tupel auch manchmal als Vektoren.

(a) Die Gleichheit zweier Tupel ist wie folgt definiert: Für alle natürlichen Zahlen k, ℓ und $a_1, \ldots, a_k$, $b_1, \ldots, b_\ell$ gilt:

$$(a_1, \ldots, a_k) = (b_1, \ldots, b_\ell) :\Leftrightarrow k = \ell \text{ und } a_1 = b_1 \text{ und } a_2 = b_2 \text{ und } \cdots \text{ und } a_k = b_k.$$

Bemerkung 2.1

(a) Für $k = 0$ gibt es genau ein k-Tupel, nämlich das leere Tupel $()$ das keine Komponente(n) hat.

(a) Man beachte den Unterschied zwischen Tupeln und Mengen: z.B.

- $(1, 2) \neq (2, 1)$, *aber* $\{1, 2\} = \{2, 1\}$.
- $(1, 1, 2) \neq (1, 2)$, *aber* $\{1, 1, 2\} = \{1, 2\}$.

Definition 2.7

Sei k eine positive natürliche Zahl und seien $M_1, \ldots, M_k$ Mengen. Das **kartesische Produkt** von $M_1, \ldots, M_k$ ist die Menge

$$M_1 \times \cdots \times M_k := \{ (m_1, \ldots, m_k) : m_1 \in M_1, \ldots, m_k \in M_k \}.$$

Das kartesische Produkt von $M_1, \ldots, M_k$ besteht also aus allen k-Tupeln, deren $i$te Komponente ein Element der Menge M_i ist.

Beispiel: Sei $M = \{a, b\}$ und $N = \{1, 2, 3\}$. Dann gilt:

- $M \times N = \{ (a,1), (a,2), (a,3), (b,1), (b,2), (b,3) \}$.
- $M \times \{1\} = \{ (a,1), (b,1) \}$.
- $M \times \emptyset = \emptyset$.
- $M^2 = \{ (a,a), (a,b), (b,a), (b,b) \}$.
- $M^1 = \{ (a), (b) \}$.
- $M^0 = \{()\}$.
- $\emptyset^2 = \emptyset$.
- $\emptyset^1 = \emptyset$.
- $\emptyset^0 = \{()\}$.
- Wir können ein Skat-Kartenspiel durch folgende Wertebereiche modellieren

$$\begin{aligned} \text{KartenArten} &= \{\text{Kreuz}, \text{Pik}, \text{Herz}, \text{Karo}\}, \\ \text{KartenSymbole} &= \{7, 8, 9, 10, \text{Bube}, \text{Dame}, \text{König}, \text{Ass}\}, \\ \text{Karten} &= \text{KartenArten} \times \text{KartenSymbole}. \end{aligned}$$

- Uhrzeiten kann man repräsentieren durch Elemente der Menge

$$\text{Uhrzeiten} := \text{Stunden} \times \text{Minuten} \times \text{Sekunden},$$

wobei

$$\begin{aligned} \text{Stunden} &:= \{0, 1, 2, ..., 23\}, \\ \text{Minuten} &:= \{0, 1, 2, ..., 59\}, \\ \text{Sekunden} &:= \{0, 1, 2, ..., 59\}. \end{aligned}$$

Das Tupel $(9, 45, 0)$ repräsentiert dann die Uhrzeit „9 Uhr, 45 Minuten und 0 Sekunden“.

2.1.3 Relationen

$\mathbb{N}$ bezeichnet die Menge der natürlichen Zahlen. In $\mathbb{N}_{>0}$ wird die Null herausgenommen.

Relationen sind Teilmengen von kartesischen Produkten. Präzise:

Definition 2.8 **(Relationen)**

(a) Sei $k \in \mathbb{N}_{>0}$ und seien $M_1, \ldots, M_k$ Mengen.

Eine **Relation auf $\mathbf{M_1}, \ldots, \mathbf{M_k}$** ist eine Teilmenge von $M_1 \times \cdots \times M_k$. Die **Stelligkeit** einer solchen Relation ist k. Gilt $M_1 = \cdots = M_k = M$ so bezeichnen wir die entsprechende Relation mit M^k.

(a) Sei M eine Menge und sei $k \in \mathbb{N}$. Eine **k-stellige Relation über M** ist eine Teilmenge von M^k.

Beispiel: Um Datumsangaben im Format (Tag, Monat, Jahr) anzugeben, nutzen wir die Wertebereiche

$$\begin{aligned} \text{TagWerte} &:= \{1, 2, \ldots, 31\} \\ \text{MonatsWerte} &:= \{1, 2, \ldots, 12\} \\ \text{JahresWerte} &:= \mathbb{Z} \end{aligned}$$

Die Menge „Gültig“ aller gültigen Daten ist dann eine Teilmenge von

$$\text{TagWerte} \times \text{MonatsWerte} \times \text{JahresWerte},$$

d.h. eine Relation auf TagWerte, MonatsWerte, JahresWerte zu der beispielsweise das Tupel $(23, 6, 1912)$ gehört,[1] nicht aber das Tupel $(30, 2, 1912)$.

Notation:

- Ist R eine Relation von M nach N (für zwei Mengen M, N), so schreiben wir oft

  $$mRn \quad \text{statt} \quad (m, n) \in R.$$

  Beispiel:

  - $m \leq n$, für natürliche Zahlen m, n
  - $m \neq n$

- Ist R eine Relation auf $M_1, \ldots, M_k$, so schreiben wir manchmal

  $$R(m_1, \ldots, m_k) \quad \text{statt} \quad (m_1, \ldots, m_k) \in R.$$

  Das soll verdeutlichen, dass R eine „Eigenschaft“ ist, die ein Tupel aus $M_1 \times \cdots \times M_k$ haben kann — oder eben nicht haben kann.

  Im Datums-Beispiel gilt: Gültig$(23, 6, 1912)$, aber es gilt nicht: Gültig$(30, 2, 1912)$.

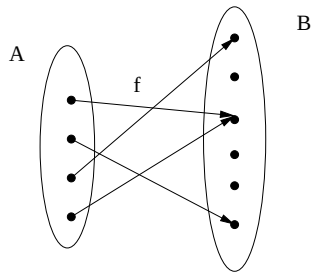
[1] Der 23. Juni 1912 ist der Geburtstag von Alan M. Turing, einem der einflussreichsten Pioniere der Informatik.

2.1.4 Funktionen

Definition 2.9 Seien A, B Mengen.

Eine **Funktion** (oder **Abbildung**) von A nach B ist eine 2-stellige Relation f auf A und B (d.h. $f \subseteq A \times B$) mit der Eigenschaft, dass für jedes $a \in A$ genau ein $b \in B$ mit $(a, b) \in f$ existiert.

Anschaulich:



Notation:

(a) Wir schreiben $f : A \to B$, um auszudrücken, dass f eine Funktion von A nach B ist.

(b) Ist $f\colon A \to B$ und ist $a \in A$, so bezeichnet $f(a)$ das (eindeutig bestimmte) $b \in B$ mit $(a, b) \in f$. Insbesondere schreiben wir meistens $f(a) = b$ an Stelle von $(a, b) \in f$.

(c) Für $f\colon A \to B$ und $A' \subseteq A$ sei

$$f(A') \ := \{\, f(a) \,:\, a \in A' \}.$$

(d) Die Menge aller Funktionen von A nach B bezeichnen wir mit $\mathrm{Abb}(A, B)$, manchmal wird auch die Notation $A \to B$, bzw. B^A benutzt.

Später, in Satz 2.11, werden wir sehen, dass

$$|\mathrm{Abb}(A, B)| \ = \ |B|^{|A|}.$$

Definition 2.10 **(Definitionsbereich, Bildbereich, Bild)**

Sei $f\colon A \to B$ eine Funktion.

(a) Der **Definitionsbereich** Def(f) von f ist die Menge Def(f) := A.

(b) Der **Bildbereich** von f ist die Menge B. Das **Bild** Bild(f) von f (genauer: das Bild von A unter f) ist die Menge

$$\mathrm{Bild}(f) \ := f(A) = \{\, f(a) \,:\, a \in A \,\} \subseteq B.$$

Definition 2.11 **(Eigenschaften von Funktionen)**

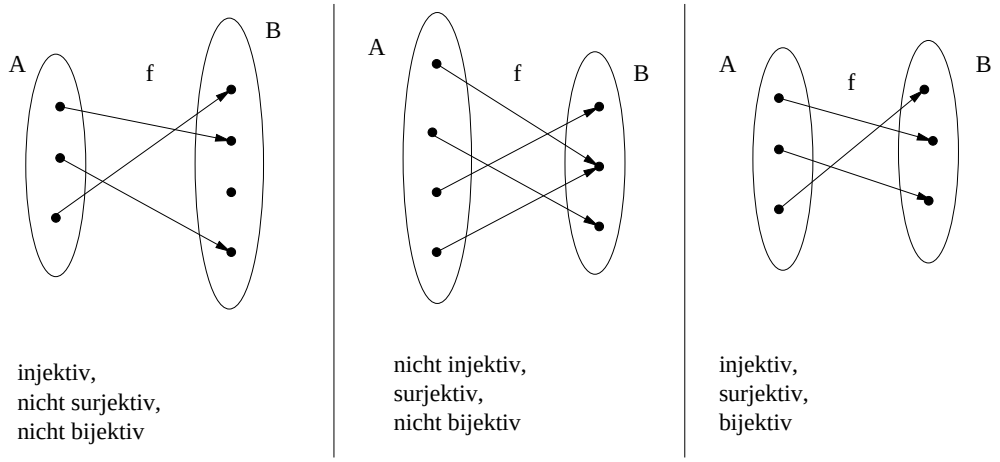
Sei $f\colon A \to B$ eine Funktion.

(a) f heißt **injektiv**, falls es für jedes $b \in B$ höchstens ein $a \in A$ mit $f(a) = b$ gibt.

(b) f heißt **surjektiv**, falls es für jedes $b \in B$ mindestens ein $a \in A$ mit $f(a) = b$ gibt.

(c) f heißt **bijektiv**, falls es für jedes $b \in B$ genau ein $a \in A$ mit $f(a) = b$ gibt.

Anschaulich:



Für jede Funktion $f\colon A \to B$ gilt:

$$f \text{ ist bijektiv } \Leftrightarrow \ f \text{ ist injektiv und surjektiv.}$$

2.2 Bäume und Graphen

Ein **ungerichteter Graph** ist ein Paar $G = (V, E)$, das aus einer Knotenmenge V und einer Kantenmenge E besteht, wobei alle Kanten Teilmengen von V der Größe zwei, also *Paarmengen* sind: Für jede Kante $e \in E$ gibt es Knoten $u, v \in V$ mit $u \neq v$ und

$$e = \{u, v\}.$$

Wir sagen, dass u und v **Endknoten** von e sind, bzw. dass e die Knoten u und v verbindet. Die Knoten $u, v \in V$ heißen **benachbart** (bzw. **adjazent**), wenn $\{u, v\}$ eine Kante von G ist. Die Anzahl der Nachbarn eines Knotens u ist der **Grad** von u. Der Grad eines Graphen ist der maximale Grad eines Knotens.

In einem **gerichteten Graphen** sind Kanten *geordnete Paare*, d.h. es gilt stets $E \subseteq V \times V$. Die Kante

$$e = (u, v)$$

besitzt den **Ausgangsknoten** u und den **Endknoten** v; der Knoten v heißt auch ein **direkter Nachfolger** von u und u heißt **direkter Vorgänger** von v. Der Ausgangsgrad Aus-Grad$_G(u)$ von Knoten u in G (Englisch: outdegree) ist die Anzahl der direkten Nachfolger von u, also

$$\text{Aus-Grad}_G(u) = \big|\{\, v \in V \,:\, (u, v) \in E \,\}\big|.$$

Der Eingangsgrad Ein-Grad$_G(v)$ von Knoten v in G (Englisch: indegree) ist die Anzahl der direkten Vorgänger von u, also

$$\text{Ein-Grad}_G(v) = \big|\{\, u \in V \,:\, (u, v) \in E \,\}\big|.$$

Während Kanten für ungerichte Graphen stets zwei verschiedene Knoten verbinden, sind „Schleifen“ (u, u) für gerichtete Graphen erlaubt.

Wir sagen (für ungerichtete wie auch gerichtete Graphen), dass die Knoten u und v mit der Kante e **inzident** sind, wenn u und v Endknoten von e sind, bzw. Anfangs- und Endknoten von e sind. Schließlich heißt $G_1 = (V, E_1)$ ein **Teilgraph** von $G_2 = (V, E_2)$, wenn $E_1 \subseteq E_2$, wenn also jede Kante von G_1 auch eine Kante von G_2 ist.

Ein **Weg** der Länge k in G ist ein Tupel

$$(v_0, v_1 \ldots, v_k)$$

von Knoten, so dass $\{v_{i-1}, v_i\} \in E$, bzw. $(v_{i-1}, v_i) \in E$. Ein Weg ohne wiederholte Knoten ist ein **einfacher Weg**. Die *Distanz* zwischen zwei Knoten ist die minimale Länge eines Weges zwischen den beiden Knoten.

Ein *Kreis* der Länge k ist ein Weg $(v_0, \ldots, v_k)$ der Länge k mit $v_0 = v_k$ und der Eigenschaft, dass $(v_0, \ldots, v_{k-1})$ ein Weg ist. Ein Kreis $(v_0, \ldots, v_{k-1}, v_0)$ heißt **einfach**, wenn $(v_0, \ldots, v_{k-1})$ ein einfacher Weg ist.

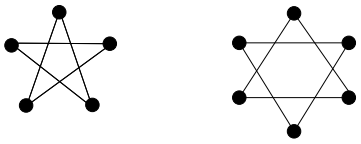


Abbildung 2.1: Der erste Graph ist ein Kreis der Länge 5. Der zweite Graph besteht aus zwei Kreisen der Länge 3.

Ein **Hamilton-Weg** ist ein einfacher Weg, der jeden Knoten genau einmal durchläuft, ein **Hamilton-Kreis** ist ein einfacher Kreis, der jeden Knoten genau einmal durchläuft, Ein **Euler-Weg** ist ein Weg, der jede Kante genau einmal durchläuft, ein **Euler-Kreis** ist ein Kreis, der jede Kante genau einmal durchläuft.

Nenne einen Graphen G **azyklisch**, wenn G keinen einfachen Kreis besitzt. Ein gerichteter Graph heißt **stark zusammenhängend**, wenn es für je zwei Knoten u und v einen Weg von u nach v gibt.

Eine Knotenmenge in einem ungerichteten Graphen ist **zusammenhängend**, wenn je zwei Knoten der Menge durch einen Weg verbunden sind. Eine **Zusammenhangskomponente** ist eine größte zusammenhängende Knotenmenge. Sei $G = (V, E)$ ein ungerichteter Graph. Ein Knoten $u \in V$ besitzt offenbar genau eine Zusammenhangskomponente, nämlich die Knotenmenge

$$\mathrm{ZH}(u) = \{v \in V \,:\, \text{es gibt einen Weg von } u \text{ nach } v\,\}.$$

Beachte, dass damit die Knotenmenge V eine *disjunkte* Vereinigung der Zusammenhangskomponenten von G ist. Warum? Angenommen, es ist $w \in \mathrm{ZH}(u) \cap \mathrm{ZH}(v)$. Damit wissen wir, dass es Wege $u \xrightarrow{*} w$ von u nach w und $v \xrightarrow{*} w$ von v nach w gibt. Ein beliebiger Knoten $v' \in \mathrm{ZH}(v)$ kann somit durch einen Weg $w \xrightarrow{*} v'$ erreicht werden, wenn wir zuerst von w („rückwärts“) nach v laufen und von dort aus nach v'. Also erhalten wir auch einen Weg $u \xrightarrow{*} w \xrightarrow{*} v'$ von u nach v' und $\mathrm{ZH}(v) \subseteq \mathrm{ZH}(u)$ folgt. Analog zeigt man $\mathrm{ZH}(u) \subseteq \mathrm{ZH}(v)$ und die Gleichheit $\mathrm{ZH}(u) = \mathrm{ZH}(v)$ folgt, wann immer $\mathrm{ZH}(u)$ und $\mathrm{ZH}(v)$ gemeinsame Knoten besitzen.

Ein **Wald** ist ein ungerichteter Graph ohne einfache Kreise. Ein **Baum** ist ein zusammenhängender Wald. (Mit anderen Worten, ein Wald ist eine knoten-disjunkte Vereinigung von Bäumen.) Die Knoten mit Grad höchstens Eins heißen **Blätter**. Ein Wald W heißt ein **Spannwald** für einen ungerichteten Graphen G, wenn W ein Teilgraph von G ist und wenn G und

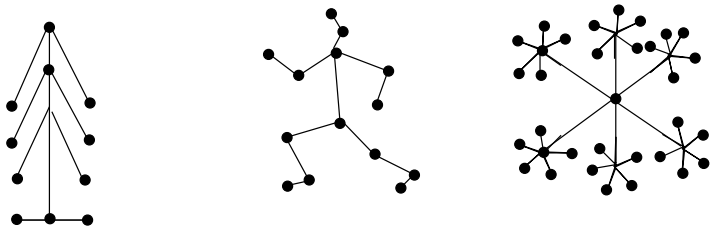


Abbildung 2.2: Ein Wald, der aus drei Bäumen besteht.

W dieselbe Knotenmenge besitzen; ist W sogar ein Baum, so heißt W ein **Spannbaum** von G.

Wir erhalten einen **gewurzelten Baum** $T = (V, E)$ aus einem Baum $T' = (V, E')$, indem wir einen Knoten $r \in V$ als **Wurzel** auszeichnen und alle Kanten von T' „von der Wurzel weg“ richten.

Genauer, ein gerichteter Graph $T = (V, E)$ ist ein gewurzelter Baum, falls T die folgenden Eigenschaften besitzt:

1. Es gibt genau einen Knoten $r \in V$ mit Ein-Grad$_T(r) = 0$: Die Wurzel hat keine eingehenden Kanten.
2. Für jeden Knoten $v \in V$ gibt es einen in r beginnenden und in v endenden Weg.
3. Für jeden von der Wurzel verschiedenen Knoten $v \in V$ ist Ein-Grad$_T(v) = 1$: Bis auf die Wurzel hat jeder Knoten genau einen Elternknoten.

Sei also $T = (V, E)$ ein gewurzelter Baum. Für jede Kante $e = (u, v) \in E$ heißt v ein **Kind** von u und u der **Elternknoten** von v. Ist neben der Kante (u, v) auch (u, w) eine Kante von B, dann nennen wir w einen **Geschwisterknoten** von v. Gibt es einen Weg von Knoten u nach Knoten v, dann heißt u **Vorfahre** von v und v **Nachfahre** von u. Knoten vom Grad 1 heißen **Blätter**. Ein Knoten $v \in V$ heißt ein **innerer Knoten** von T, wenn v kein Blatt ist und von der Wurzel verschieden ist.

Die **Tiefe** des Knotens v ist die Länge des Weges von der Wurzel nach v, die **Höhe** von v ist die größte Länge eines in v beginnenden Weges. Die Tiefe eines Baums ist die größte Tiefe eines Blatts, die Höhe eines Baums ist die Höhe der Wurzel.

Ein gewurzelter Baum $T = (V, E)$ heißt **binär**, wenn Aus-Grad$_T(v) \leq 2$ für jeden Knoten $v \in V$ gilt. Ein **vollständiger binärer** Baum der Tiefe t ist ein binärer Baum, in dem alle Blätter die Tiefe t besitzen und alle inneren Knoten wie auch die Wurzel den Aus-Grad genau 2 besitzen.

Ein **vollständiger Graph** oder **eine Clique** ist ein ungerichteter Graph, in dem je zwei Knoten benachbart sind. Eine **unabhängige Menge** ist eine Knotenmenge, in der je zwei Knoten *nicht* durch eine Kante verbunden sind. Ein Graph ist **bipartit**, wenn seine Knotenmenge in zwei unabhängige Mengen zerlegt werden kann.

Eine **konfliktfreie Färbung** eines ungerichteten Graphen $G = (V, E)$ ist eine Farbzuweisung an die Knoten, so dass benachbarte Knoten verschiedene Farben erhalten. Eine konfliktfreie Färbung zerlegt somit die Knotenmenge in unabhängige Mengen. Die minimale Farbenzahl, die für die Färbung eines Graphen G benötigt wird, heißt die **chromatische Zahl** $\chi(G)$ von G.

Aufgabe 1
(a) Zeige, dass die Tiefe eines binären Baums mit n Blättern mindestens $\lceil \log_2 n \rceil$ beträgt.

(b) Sei B ein binärer Baum mit N Blättern und sei T_i die Tiefe des $i$ten Blatts. Zeige die sogenannte Kraft'sche Ungleichung

$$\sum_{i=1}^{n} 2^{-T_i} \leq 1.$$

Hinweis: Assoziere einen binären String mit jedem Blatt.

2.3 Was ist ein korrektes Argument?

(Teile dieses Abschnitts orientieren sich an den Abschnitten 2.5 und 2.6 des Skripts „Diskrete Modellierung“ von Nicole Schweikardt.)

Die theoretische Informatik arbeitet mit mathematischen Methoden, um unumstößliche Aussagen machen zu können. In dieser Veranstaltung möchten wir zum Beispiel zweifelsfreie Aussagen über das Verhalten von Algorithmen und Datenstrukturen zur Lösung eines algorithmischen Problems angeben. Typische Aussagen sind von der folgenden Form:

> Der Algorithmus löst das Problem für alle Eingaben, und für alle Eingaben der Länge n garantieren die eingesetzten Datenstrukturen eine Laufzeit von höchstens $t(n)$ Schritten.

Kompliziertere Analysen unterteilen wir in Zwischenbehauptungen, die wir „Lemmas“ nennen und die abschließende Behauptung, die in einem „Satz“ festgehalten wird. Die Zwischenbehauptungen wie auch die abschließende Behauptung müssen natürlich in jeder Situation richtig sein und durch jeweils einen „Beweis“ untermauert sein. Der Beweis einer Behauptung darf verwenden:

- etwaige Voraussetzungen der Behauptung,
- Definitionen und bereits bekannte Tatsachen und Sätze,
- im Beweis selbst oder anderswo bereits als wahr bewiesene Aussagen,
- und darf diese Fakten durch logische Schlussregeln verknüpfen.

Das hört sich alles sehr kompliziert an, ist aber höchst einfach: Der Beweis muss aus einer Folge von einfach nachvollziehbaren Schritten bestehen, die selbst der „Metzger um die Ecke“ nachvollziehen kann und überzeugend findet.

Ziel dieses Abschnitts ist ein kurzer Überblick über die folgenden grundlegenden Beweistechniken, insbesondere:

- direkter Beweis
- Beweis durch Kontraposition
- Beweis durch Widerspruch (indirekter Beweis)
- und vollständige Induktion.

2.3.1 Direkte Beweise

Bei einem direkten Beweis wird die Behauptung eines Satzes „direkt“, d.h. ohne „Umwege“, bewiesen. Ein erstes Beispiel ist der Nachweis, dass das arithmetische Mittel stets mindestens so groß wie das geometrische Mittel ist. Im Verlauf der Vorlesung werden wir viele weitere Beispiele kennenlernen.

Satz 2.2 *Für alle reellen Zahlen $a, b \geq 0$ gilt*

$$\frac{a+b}{2} \geq \sqrt{a \cdot b}.$$

Zuerst versuchen wir eine Beweisidee zu erhalten.

1. Die Wurzel stört und wir quadrieren:
   - Statt $\frac{a+b}{2} \geq \sqrt{a \cdot b}$ zeige die Ungleichung $(\frac{a+b}{2})^2 \geq a \cdot b$.
2. Wir multiplizieren aus und erhalten die Ungleichung $\frac{a^2+2a\cdot b+b^2}{4} \geq a \cdot b$.
3. Wir multiplizieren mit 4 und erhalten $a^2 + 2a \cdot b + b^2 \geq 4a \cdot b$.
4. Wenn wir $4a \cdot b$ nach links „bringen“, ist $a^2 - 2a \cdot b + b^2 \geq 0$ zu zeigen.
   - $a^2 - 2a \cdot b + b^2 = (a - b)^2$ gilt.
   - Jedes Quadrat ist nicht-negativ und die Ungleichung stimmt!?!

Das ist leider kein Beweis, weil wir aus der Ungleichung $\frac{a+b}{2} \geq \sqrt{a \cdot b}$ eine wahre Aussage folgern. Hoffentlich haben wir nur mit äquivalenten Umformungen gearbeitet.

Beweis:

1. $a^2 - 2a \cdot b + b^2 = (a - b)^2$ gilt und $a^2 - 2a \cdot b + b^2 \geq 0$ folgt.
2. Wir addieren $4a \cdot b$ auf beide Seiten: $a^2 + 2a \cdot b + b^2 \geq 4a \cdot b$ gilt ebenfalls.
3. Die linke Seite der Ungleichung stimmt mit $(a + b)^2$ überein: Es gilt also
$$(a + b)^2 \geq 4a \cdot b.$$
4. Wir dividieren beide Seiten durch 4 und ziehen die Wurzel:
$$\frac{a+b}{2} \geq \sqrt{ab}$$
folgt und das war zu zeigen. □

2.3.2 Beweis durch Kontraposition

Der *Beweis durch Kontraposition* beruht auf der Äquivalenz der beiden Implikationen

wenn Aussage A wahr ist, dann ist auch Aussage B wahr

und

wenn Aussage B falsch ist, dann ist auch Aussage A falsch.

Hier ist ein erstes Beispiel für einen Beweis durch Kontraposition.

Satz 2.3
Für jedes $n \in \mathbb{N}$ gilt: Falls n^2 eine ungerade Zahl ist, so ist auch n eine ungerade Zahl.

Beweis: Durch Kontraposition. Sei $n \in \mathbb{N}$ beliebig. Falls n ungerade ist, so ist nichts weiter zu beweisen.

Wir zeigen: Falls n *keine* ungerade Zahl ist, so ist auch n^2 *keine* ungerade Zahl.

Beachte: Nach Definition ist eine natürliche Zahl m genau dann *gerade*, wenn es ein $k \in \mathbb{N}$ gibt, s.d. $m = 2 \cdot k$. Daher gilt:

$$\begin{aligned} n \text{ ist gerade} &\Rightarrow \text{es existiert } k \in \mathbb{N} \text{ s.d. } n = 2 \cdot k \quad (\text{denn } n \text{ ist gerade}) \\ &\Rightarrow \text{es existiert } k \in \mathbb{N} \text{ s.d. } n^2 = 4 \cdot k^2 = 2 \cdot (2 \cdot k^2) \\ &\Rightarrow n^2 \text{ ist gerade} \quad (\text{nach Definition gerader Zahlen}). \end{aligned}$$

Somit ist n^2 gerade, d.h. n^2 ist keine ungerade Zahl. □

2.3.3 Beweis durch Widerspruch

Man beweist einen Satz der Form

„Falls die Voraussetzungen A erfüllt sind, so gilt Aussage B“

durch Widerspruch dadurch, dass man

- annimmt, dass die Voraussetzungen A erfüllt sind, aber die Aussage B *nicht* gilt und
- daraus einen Widerspruch herleitet.

Nicht nur Implikationen lassen sich durch Widerspruch beweisen. Möchten wir zum Beispiel die Aussage B durch Widerspruch beweisen, dann stellen wir uns vor, dass die Implikation „Wenn $1 = 1$, dann gilt Aussage B“ zu zeigen ist: Nimm an, dass die Aussage B *nicht* gilt und leite daraus einen Widerspruch her.

Als ein erstes Beispiel für einen Beweis durch Widerspruch betrachten wir folgenden Satz:

Satz 2.4 *$\sqrt{2}$ ist irrational, also keine rationale Zahl.*

Beweis: Durch Widerspruch. Wir nehmen an, dass $\sqrt{2}$ eine rationale Zahl ist und müssen einen Widerspruch herleiten.

Wenn $\sqrt{2}$ eine rationale Zahl ist, dann gibt es *teilerfremde* Zahlen $p, q \in \mathbb{N}$ mit

$$\sqrt{2} = \frac{p}{q}.$$

Wir quadrieren und erhalten die Gleichung

$$p^2 = 2 \cdot q^2.$$

Also ist p^2 eine gerade Zahl. Wir haben aber in Satz 2.3 gezeigt, dass dann auch p gerade ist. Wenn aber $p = 2r$, dann folgt $p^2 = 4r^2 = 2q^2$ und damit $2r^2 = q^2$. Dann ist aber q^2 gerade. Wir wenden wieder Satz 2.3 an und erhalten, dass auch q gerade ist: Die Zahlen p und q haben den gemeinsamen Teiler 2 im Widerspruch zur Teilerfremdheit ↯ □

Der griechische Mathematiker Euklid (300 v. Chr.) hat gezeigt, dass es unendliche viele Primzahlen[2] gibt. Dieses Resultat ist Grundlage für viele Verfahren der Public-Key Kryptographie. Mehr über die Public-Key Kryptographie erfahren Sie in der Veranstaltung „Mathematik 2“.

Satz 2.5 (Satz von Euklid) *Es gibt unendlich viele Primzahlen.*

Beweis: Wir benutzen, dass sich jede natürliche Zahl als Produkt von Primzahlen schreiben lässt.

Wir nehmen an, dass es nur endlich viele Primzahlen gibt und dass dies die Primzahlen $p_1, \ldots, p_n$ sind. Definiere die Zahl

$$N = p_1 \cdots p_n + 1.$$

Dann ist $N - 1$ durch alle Primzahlen teilbar und N kann durch keine Primzahl teilbar sein, denn Zahlen mit einem gemeinsamen Teiler größer als Eins haben einen Abstand größer als Eins. ↯ □

2.3.4 Vollständige Induktion

(Teile dieses Abschnitts orientieren sich am Skript „Diskrete Modellierung“ von Nicole Schweikardt.)

Wir möchten die Aussage $A(n)$ für alle natürlichen Zahlen n zeigen. Dazu ist es ausreichend, wenn wir

1. im Induktionsanfang die Aussage $A(0)$ zeigen und
2. im Induktionsschritt die Implikation

$$(A(0) \wedge A(1) \wedge \cdots \wedge A(n)) \to A(n+1)$$

für jedes $n \in \mathbb{N}$ nachweisen.

[2] Eine Primzahl ist eine natürliche Zahl größer als Eins, die nur durch sich selbst und durch die Eins teilbar ist.

Typischerweise, wie auch in Beispiel 2.1, nimmt man „nur“ die Aussage $A(n)$ in der *Induktionsannahme* an und weist die Aussage $A(n+1)$ nach. In Beispiel 2.2 müssen wir aber zum Beispiel die stärkere Induktionsannahme $A(n-1) \wedge A(n)$ benutzen, um $A(n+1)$ zeigen zu können. Um aber diese stärkere Induktionsannahme abrufen zu können, müssen wir im Induktionsanfang mehr tun und die Aussagen $A(1)$ und $A(2)$ zeigen.

Manchmal gilt eine Aussage $A(n)$ nur für alle Zahlen $n \in \mathbb{N}$ ab einer Zahl n_0. In einem solchen Fall zeigt man die Aussage $A(n_0)$ im INDUKTIONSANFANG und behält die Vorgehensweise des INDUKTIONSSCHRITTS bei.

Beispiel 2.1 (**Es ist richtig dunkel**)

Wir befinden uns in einem stockdunklem Gang, der nach einer Seite unbeschränkt lang ist. Den Gang können wir nur über die andere Seite verlassen. Was tun, wir kennen noch nicht einmal die Länge **N** des Weges bis zum Ende des Ganges? Wie können wir mit möglichst wenigen Schritten den Gang verlassen?

> Wie wär's mit: Ein Schritt nach „vorn“, zwei zurück, drei nach vorn, vier zurück, ...

Und wieviele Schritte benötigen wir dann, um den Gang zu verlassen? Wir nehmen an, dass der erste Schritt in die falsche Richtung führt. Nach $2k$ Wiederholungen sind wir dann insgesamt $(-1+2)+(-3+4)+\cdots+(-(2k-1)+2k) = k$ Schritte in die richtige Richtung gegangen. Wir müssen N Schritte gehen, um den Gang zu verlassen und müssen deshalb insgesamt

$$(1+2)+(3+4)+\cdots+(2N-1+2N) = \sum_{i=1}^{2N} i$$

Schritte zurücklegen. Wir werden mächtig erschöpft sein, denn das sind quadratisch viele Schritte:

Satz 2.6 $\sum_{i=1}^{n} i = 1+2+\ldots+n = \frac{n\cdot(n+1)}{2}$.

Beweis: Wir geben zwei Argumente und beginnen mit der vollständigen Induktion nach n:

(a) INDUKTIONSANFANG für $n=0$: Es ist $\sum_{i=1}^{0} i = 0$ und $\frac{0\cdot(0+1)}{2} = 0$. ✓

(b) INDUKTIONSSCHRITT von n auf $n+1$:

- Wir können die *Induktionsannahme* $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$ voraussetzen.
- $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) \overset{\text{Ind.ann.}}{=} \frac{n\cdot(n+1)}{2} + (n+1) = \frac{n\cdot(n+1)+2\cdot(n+1)}{2} = \frac{(n+2)\cdot(n+1)}{2} = \frac{(n+1)\cdot(n+2)}{2}$. ✓

Unser zweites Argument ist ein direkter Beweis: Wir betrachten ein Gitter mit n Zeilen und n Spalten. Das Gitter hat n^2 Gitterpunkte. Die Summe $\sum_{i=1}^{n} i$ stimmt überein mit der Anzahl der Gitterpunkte unterhalb der Hauptdiagonale und auf der Hauptdiagonale. Die Hauptdiagonale besitzt n Gitterpunkte und unterhalb der Hauptdiagonale befindet sich die Hälfte der verbleibenden $n^2 - n$ Gitterpunkte. Also folgt

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

und auch dieses Argument ist abgeschlossen. □

Sind quadratisch viele Schritte wirklich notwendig? Alles auf eine Karte zu setzen, also nur in eine Richtung zu marschieren, ist Unfug. Aber können wir etwas mutiger sein, als immer nur einen weiteren Schritt zu wagen?

> Zum Beispiel, Ein Schritt nach vorn, zwei zurück, vier nach vorn, acht zurück, ...,

Und wieviele Schritte benötigen wir diesmal, um den Gang zu verlassen? Wir nehmen auch diesmal an, dass der erste Schritt in die falsche Richtung führt. Nach $2k$ Wiederholungen sind wir

$$\begin{aligned} & (-1+2)+(-4+8)+\cdots+(-2^{2k-2}+2^{2k-1}) \\ = \; & -(1+4+\cdots+2^{2k-2})+2\cdot(1+4+\cdots+2^{2k-2}) \\ = \; & (1+4+\cdots+2^{2k-2}) = \sum_{i=0}^{k-1} 4^i \end{aligned}$$

Schritte in die richtige Richtung gegangen und haben insgesamt

$$\begin{aligned} (1+2)+(4+8)+\cdots+(2^{2k-2}+2^{2k-1}) &= (1+4+\cdots+2^{2k-2})+2\cdot(1+4+\cdots+2^{2k-2}) \\ &= 3\cdot(1+4+\cdots+2^{2k-2}) = 3\cdot\sum_{i=0}^{k-1} 4^i \end{aligned}$$

Schritte zurückgelegt.

Um den Gang nach $2k+2$ Wiederholungen zu verlassen, muss $\sum_{i=0}^{k-1} 4^i < N$ gelten, da wir sonst den Gang schon nach $2k$ Wiederholungen verlassen hätten. In der letzten Wiederholung gehen wir 2^{2k} Schritte in die falsche Richtung, kehren mit derselben Schrittzahl wieder um und gehen die letzten maximal N Schritte. Insgesamt sind wir

$$3\cdot\sum_{i=0}^{k-1} 4^i + 2\cdot 2^{2k} + N < 3\cdot\sum_{i=0}^{k-1} 4^i + 2\cdot 4^k + N \leq (3+2\cdot 4)\cdot\sum_{i=0}^{k-1} 4^i + N < 12\cdot N$$

Schritte gelaufen. Können wir die Analyse verbessern?

Satz 2.7 (Die geometrische Reihe) *Für alle $n \in \mathbb{N}$ gilt*

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1},$$

falls $a \neq 1$ eine reelle Zahl ist.

Beweis: Wir geben wieder zwei Argumente und beginnen mit der vollständigen Induktion nach n:

(a) Induktionsanfang für $n=0$: $\sum_{i=1}^{0} a^i = 1$ und $\frac{a^{0+1}-1}{a-1} = 1$. ✓

(b) Induktionsschritt von n auf $n+1$:

- Wir können die *Induktionsannahme* $\sum_{i=1}^{n} a^i = \frac{a^{n+1}-1}{a-1}$ voraussetzen. Dann ist

$$- \sum_{i=1}^{n+1} a^i = \sum_{i=1}^{n} a^i + a^{n+1} \overset{\text{Ind.ann}}{=} \frac{a^{n+1}-1}{a-1} + a^{n+1} = \frac{a^{n+1}-1+a^{n+2}-a^{n+1}}{a-1} = \frac{a^{n+2}-1}{a-1} \checkmark$$

Auch hier ist unser zweites Argument ein direkter Beweis:

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

Jetzt dividiere durch $a-1$ und die Behauptung ist gezeigt. ✓ □

Wir können unsere Analyse tatsächlich verbessern. Dazu beachte zuerst, dass $\sum_{i=0}^{k-1} 4^i = \frac{4^k-1}{4-1} < N$ und damit $4^k - 1 < 3 \cdot N$ gilt. Deshalb ist $3 \cdot \sum_{i=0}^{k-1} 4^i + 2 \cdot 4^k = 3 \cdot \frac{4^k-1}{4-1} + 2 \cdot 4^k = 3 \cdot (4^k - 1) + 2 \leq 9 \cdot N + 2$. Die Gesamtzahl $3 \cdot \sum_{i=0}^{k-1} 4^i + 2 \cdot 4^k + N$ der Schritte ist also durch $10 \cdot N + 2$ beschränkt.

Vorher quadratisch viele Schritte, jetzt linear viele! □ Ende Beispiel 2.1

Beispiel 2.2 **(Die Fibonacci-Folge)**

Ein Bauer züchtet Kaninchen. Jedes weibliche Kaninchen bringt im Alter von zwei Monaten ein weibliches Kaninchen zur Welt und danach jeden Monat ein weiteres. Wie viele weibliche Kaninchen hat der Bauer am Ende des n-ten Monats, wenn er mit einem neu geborenen weiblichen Kaninchen startet? Diese Anzahl bezeichnen wir mit $\text{fib}(n)$.

Wie schnell wächst $\text{fib}(n)$? Können wir sogar einen expliziten Ausdruck für $\text{fib}(n)$ bestimmen? Um diese Fragen beantworten zu können, geben wir zuerst eine rekursive Definition, benutzen also die Grundidee der vollständigen Induktion, um die Folge $\text{fib}(n)$ besser zu verstehen.

- Rekursionsanfang für $n = 1$ und $n = 2$: Es ist $\text{fib}(1) := 1$ und $\text{fib}(2) := 1$.
- Rekursionsschritt von $n-1$ und n nach $n+1$: Es ist $\text{fib}(n+1) := \text{fib}(n) + \text{fib}(n-1)$ für alle $n \in \mathbb{N}$ mit $n \geq 2$.

  Warum? Genau die $\text{fib}(n-1)$ Kaninchen, die sich im Monat $n-1$ im Besitz des Bauern befinden, haben jeweils einen Nachkommen im Monat $n+1$. Des Weiteren besitzt der Bauer $\text{fib}(n)$ Kaninchen im Monat n und diese Kaninchen bleiben auch im Monat $n+1$ in seinem Besitz.

Somit gilt:

n	1	2	3	4	5	6	7	8	9	10	11	12
$\text{fib}(n)$	1	1	2	3	5	8	13	21	34	55	89	144

Die Folge $\text{fib}(n)$ wird auch **Fibonacci-Folge** genannt; sie ist benannt nach dem italienischen Mathematiker Leonardo Fibonacci (13. Jh.). Die Zahl $\text{fib}(n)$ heißt auch **n-te Fibonacci-Zahl**. Wir zeigen, dass

$$2^{n/2} \leq \text{fib}(n) \leq 2^n$$

für alle $n \geq 6$ gilt.

Satz 2.8

(a) Für alle natürlichen Zahlen $n \geq 1$ gilt $\text{fib}(n) \leq 2^n$.

(b) Für alle natürlichen Zahlen $n \geq 6$ gilt $2^{n/2} \leq fib(n)$.

Beweis: Wir zeigen Teil (a) durch Induktion nach n.

INDUKTIONSANFANG: Betrachte $n = 1$ und $n = 2$.

Behauptung: $\text{fib}(1) \leq 2^1$ und $\text{fib}(2) \leq 2^2$.

Beweis: Es gilt: $\text{fib}(1) \overset{\text{Def.}}{=} 1 \leq 2 = 2^1$ und $\text{fib}(2) \overset{\text{Def.}}{=} 1 \leq 4 = 2^2$. ✓

INDUKTIONSSCHRITT: $n \to n + 1$

Sei $n \in \mathbb{N}$ mit $n \geq 2$ beliebig.

Induktionsannahme: Für alle natürlichen Zahlen i mit $1 \leq i \leq n$ gilt $\text{fib}(i) \leq 2^i$.

Behauptung: $\text{fib}(n + 1) \leq 2^{n+1}$.

Beweis: $\text{fib}(n+1) \overset{\text{Def.}}{=} \text{fib}(n) + \text{fib}(n-1) \overset{\text{Ind.ann.}}{\leq} 2^n + 2^{n-1} \leq 2 \cdot 2^n = 2^{n+1}$. □

Analog zeigt man auch die Aussage in Teil (b). Allerdings muss man diesmal im Induktionsanfang $2^{n/2} \leq \text{fib}(n)$ für $n = 6$ und $n = 7$ zeigen. □

Aufgabe 2
Zeige: Für alle natürlichen Zahlen $n \geq 6$ gilt $2^{n/2} \leq \text{fib}(n)$.

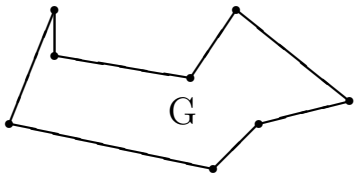
Bemerkung 2.1 Es gibt auch einen expliziten Ausdruck für die n-te Fibonacci-Zahl. Für alle $n \in \mathbb{N}$ mit $n \geq 1$ gilt nämlich:

$$\text{fib}(n) = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right). \tag{2.1}$$

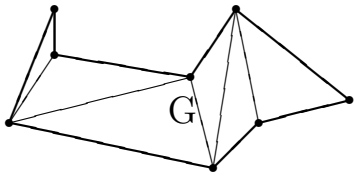
Aufgabe 3
Zeige Darstellung (2.1) durch vollständige Induktion nach n.

Hinweis: $\frac{1+\sqrt{5}}{2}$ und $\frac{1-\sqrt{5}}{2}$ erfüllen die quadratische Gleichung $x^2 = x + 1$.

Aufgabe 4
Ein einfaches Polygon ist ein von einem geschlossenen, sich nicht schneidenden, Streckenzug begrenztes, ebenes geometrisches Objekt. Hier ein Beispiel für ein einfaches Polygon G mit $n = 8$ Ecken.



Wir möchten ein einfaches Polygon triangulieren, das heißt in disjunkte Dreiecke zerlegen. Dazu können nicht überschneidende Diagonalen in das einfache Polygon eingefügt werden, wobei eine Diagonale eine im Polygon verlaufende Strecke ist, die zwei Eckpunkte verbindet. Hier ist ein Beispiel einer möglichen Triangulation des obigen einfachen Polygons:



(a) **Beweise**, dass sich jedes einfache Polygon G mit n Ecken durch $n-3$ Diagonalen triangulieren lässt und, dass dabei $n-2$ Dreiecke entstehen.

(b) **Beschreibe** für allgemeine Eckenzahl n ein einfaches Polygon, bei dem diese Zerlegung eindeutig ist, d.h. jede Triangulierung besitzt dieselbe Menge von Diagonalen. Aus deiner Beschreibung soll erkennbar sein, wie man für gegebenes n ein solches einfaches Polygon konstruiert. Argumentiere, warum die Zerlegung deiner einfachen Polygone eindeutig ist.

Allerdings müssen induktive Beweise sorgfältig geführt werden. Wo steckt der Fehler in den beiden folgenden Argumenten?

Aufgabe 5
(a) Wir zeigen, dass je zwei natürliche Zahlen a und b gleich sind. Dazu setzen wir $k = \max\{a, b\}$ und führen eine Induktion nach k. Im Basisschritt haben wir $k = 0$ und deshalb ist $a = 0 = b$ und das war zu zeigen. Im Induktionsschritt ist $\max\{a, b\} = k + 1$, und wir können die Induktionsbehauptung auf $a - 1$ und $b - 1$ anwenden, denn $\max\{a - 1, b - 1\} = k$. Also ist $a - 1 = b - 1$ und die Behauptung $a = b$ folgt.

(b) Wir zeigen, dass alle Pferde die gleiche Farbe besitzen und führen einen Beweis durch Induktion über die Zahl k aller Pferde. Im Basisschritt ist $k = 1$ und die Behauptung ist offensichtlich richtig. Für den Induktionsschritt nehmen wir an, dass es $k+1$ Pferde $p_1, \ldots, p_{k+1}$ gibt. Dann haben aber nach Induktionsvoraussetzung sowohl $p_1, p_2, \ldots, p_k$ die Farbe von p_2 wie auch $p_2, \ldots, p_{k+1}$ ebenfalls die Farbe von p_2: Wir haben die Behauptung gezeigt.

2.3.4.1 Analyse rekursiver Programme

Wir untersuchen ein rekursives Programm $R(\vec{p})$ (mit den Parametern $\vec{p}$) und legen eine Eingabe x fest. Während der Abarbeitung von $R(\vec{p})$ werden rekursive Aufrufe getätigt, die ihrerseits weitere rekursive Aufrufe starten und so weiter. Der Rekursionsbaum B modelliert die Struktur all dieser rekursiven Aufrufe. B wird nach den folgenden Regeln gebaut.

1. Beschrifte die Wurzel von B mit den Parametern $\vec{p}$ des Erstaufrufs.
   - Wenn innerhalb von $R(\vec{p})$ keine rekursiven Aufrufe getätigt werden, dann wird die Wurzel zu einem Blatt.
   - Ansonsten erhält die Wurzel für jeden rekursiven Aufruf innerhalb $R(\vec{p})$ ein neues Kind, das mit den Parametern des rekursiven Aufrufs beschriftet wird.
2. Wenn ein Knoten v von B mit den Parametern $\vec{q}$ beschriftet ist, gehen wir mit v genauso wie mit der Wurzel vor.
   - Wenn innerhalb von $R(\vec{q})$ keine rekursiven Aufrufe getätigt werden, wird v zu einem Blatt.
   - Ansonsten erhält v für jeden rekursiven Aufruf innerhalb von $R(\vec{q})$ ein neues Kind, das mit den Parametern des rekursiven Aufrufs beschriftet wird.

Die Anzahl der Knoten von B stimmt überein mit der Anzahl aller rekursiven Aufrufe für Eingabe x.

Beispiel 2.3 **(Berechnung der $n$ten Fibonacci-Zahl)**

Wir möchten die $n$te Fibonacci-Zahl $\text{fib}(n)$ für eine natürliche Zahl $n \geq 1$ berechnen und tun dies mit zwei verschiedenen Algorithmen. Welcher Algorithmus ist schneller?

`Algo1`(n):

(a) Falls $n = 1$, dann gib `Algo1`$(1) := 1$ als Ergebnis zurück.

(b) Falls $n = 2$, dann gib `Algo1`$(2) := 1$ als Ergebnis zurück.

(c) Falls $n \geq 3$, dann gib `Algo1`$(n) :=$ `Algo1`$(n-1) +$ `Algo1`$(n-2)$ als Ergebnis zurück.

Wenn wir jede Addition, jeden Vergleich, und jedes Zurückgeben eines Ergebnisses als einen Schritt zählen, dann benötigt dieser rekursive Algorithmus bei Eingabe einer Zahl n genau $g_1(n)$ Schritte, wobei

$$\begin{aligned} g_1(1) &= 2 \quad \text{und} \quad g_1(2) = 3 \quad \text{und} \\ g_1(n) &= 3 + g_1(n-1) + g_1(n-2) + 2 \\ &= 5 + g_1(n-1) + g_2(n-2) \quad \text{für alle } n \in \mathbb{N} \text{ mit } n \geq 3. \end{aligned}$$

Ein anderer Algorithmus, der für eine natürliche Zahl $n \geq 1$ den Wert $\text{fib}(n)$ berechnet, ist:

`Algo2`(n):

(a) Falls $n = 1$ oder $n = 2$, dann gib 1 als Ergebnis zurück.

(b) Seien $a_0 := 0$, $a_1 := 1$ und $a_2 := 1$.

(c) Wiederhole für alle i von 3 bis n:

(d) Ersetze a_0 durch a_1 und a_1 durch a_2.

(e) Ersetze a_2 durch $a_0 + a_1$.

(f) Gib den Wert a_2 als Ergebnis zurück.

Dieser Algorithmus benötigt bei Eingabe $n \geq 1$ genau $g_2(n) := 6+5\cdot(n-2)$ Schritte. (Ähnlich wie oben zählen wir jeden Vergleich, jedes Zurückgeben eines Werts und jedes Setzen eines Werts als einen Schritt. Für jeden Schleifendurchlauf berechnen wir zusätzlich zwei Schritte, um den Wert von i um eins zu erhöhen und zu testen, ob das Ergebnis kleiner oder gleich n ist).

1. Frage: Welchen Algorithmus nehmen wir?

Offensichtlich gilt $g_1(n) \geq \text{fib}(n)$. Wir wissen aber schon aus Beispiel **??**, dass $\text{fib}(n) \geq 2^{n/2}$ für $n \geq 6$ gilt. Der elegante rekursive `Algo1` besitzt eine exponentielle Laufzeit und ist absolut gräßlich, während sein unscheinbarer Kollege `Algo2` mit „linearer Laufzeit“ glänzt.

2. Frage: Wie sehen die Rekursionsbäume für `Algo1` und `Algo2` aus?

Der Rekursionsbaum für `Algo1`(n) ist ein markierter binärer Baum der wie folgt rekursiv definiert ist. Die Rekursionsbäume für $n = 1$ und $n = 2$ bestehen nur aus einem einzigen Knoten der jeweils mit „1“, bzw. „2“ markiert ist. Kennen wir die Rekursionsbäume B_{n-2} für $n-2$ und B_{n-1} für $n-1$, dann erhalten wir den Rekursionsbaum B_n für n, indem wir

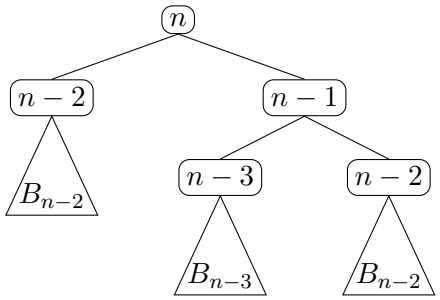


Abbildung 2.3: Die rekursive Definition des Baums B_n

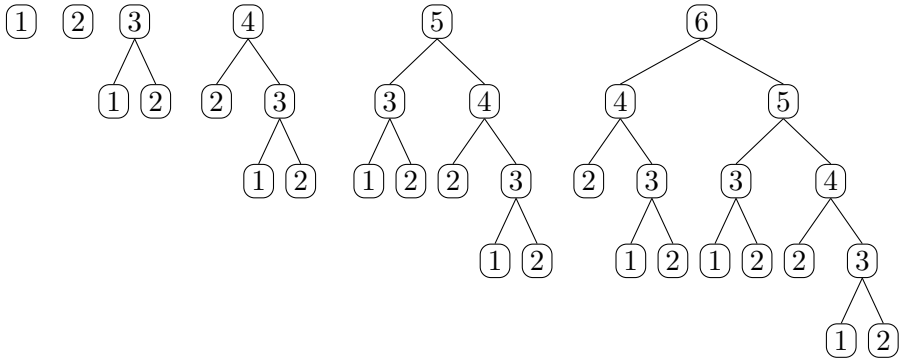


Abbildung 2.4: Die „Fibonacci-Bäume“ B_1, B_2, B_3, B_4, B_5 und B_6.

die Wurzel r von B_n mit n markieren, die Wurzel von B_{n-2} zum linken Kind und die Wurzel von B_{n-1} zum rechten Kind von r machen.

Der Baum B_n hat „beinahe“ die Graph-Struktur eines vollständigen Binärbaums, aber der linke Baum B_{n-2} ist nicht so tief wie der rechte Baum B_{n-1}. Die Wurzel „berechnet“ fib(n), das linke Kind fib$(n-2)$ und das rechte Kind fib$(n-1)$. Und jetzt wird auch offensichtlich wie idiotisch sich `Algo2` verhält, denn um fib$(n-1)$ im rechten Kind der Wurzel zu berechnen, wird fib$(n-3)$ aber auch fib$(n-2)$ neu berechnet, obwohl fib$(n-2)$ im linken Kind der Wurzel bereits berechnet wurde. Als ein weiteres Beispiel: Die Fibonacci-Zahl fib(2) tritt 5-mal in B_6 auf!

Diesen Fehler macht `Algo2` nicht, denn jede Fibonacci-Zahl wird genau einmal berechnet. `Algo2` ist ein iteratives Programm und besitzt deshalb natürlich keinen Rekursionsbaum, seine Vorgehensweise wird aber am besten durch einen Weg $n \to n-1 \to n-2 \to \cdots \to 2$ wiedergegeben. Dieser Weg ist nur ein „klitzekleiner“ Teilbaum von B_n, die restlichen Knoten von B_n führen alle unnötige Neuberechnungen bereits bekannter Fibonacci-Zahlen durch.

Beispiel 2.4 **(Binärsuche)**

Ein Array $A = (A[1], \ldots, A[n])$ von n Zahlen und eine Zahl x ist gegeben. Wir möchten wissen, ob und wenn ja wo die Zahl x in A vorkommt.

Wenn A nicht sortiert ist, dann bleibt uns nichts anderes übrig als uns alle Zellen von A auf der Suche nach x anzuschauen. Wir führen eine **lineare Suche** durch, die im schlimmsten Fall alle n Zellen des Arrays inspizieren muss. Wenn das Array aber aufsteigend sortiert ist, dann können wir **Binärsuche** anwenden.

```
void Binärsuche( int unten, int oben){
```

```
if (oben < unten)
  std::cout « x « " wurde nicht gefunden."
   « std::endl;
int mitte = (unten+oben)/2;
if (A[mitte] == x)
  std::cout « x « " wurde in Position " « mitte « " gefunden." « std::endl;
  else {
    if (x < A[mitte])
      Binärsuche(unten, mitte-1);
    else
      Binärsuche(mitte+1,oben);}}
```

1. Frage: Ist `Binärsuche` korrekt?

Es sei $n = \mathit{oben} - \mathit{unten} + 1$. Wir geben einen Beweis mit vollständiger Induktion nach n.

(a) Induktionsanfang für $n = 0$: Es ist $n = \mathit{oben} - \mathit{unten} + 1 = 0$ und $\mathit{oben} < \mathit{unten}$ folgt. In diesem Fall wird `Binärsuche`$(\mathit{unten}, \mathit{oben})$ richtigerweise mit der Antwort „x wurde nicht gefunden“ abbrechen. ✓

(b) Induktionsschritt $n \to n+1$: Es ist $\mathit{oben} - \mathit{unten} + 1 = n + 1$. Wenn $A(\mathit{mitte})$ mit x übereinstimmt, dann wird richtigerweise mit Erfolgsmeldung abgebrochen.

Ansonsten sucht `Binärsuche` richtigerweise in der linken Hälfte, wenn $x < A[\mathit{mitte}]$, und sonst in der rechten Hälfte. Die rekursive Suche in der linken bzw. rechten Hälfte verläuft aber nach Induktionsannahme korrekt. ✓

Wir fordern $n = 2^k - 1$ für eine Zahl $k \in \mathbb{N}$. Sei $T(n)$ die maximale Anzahl von Zellen, die Binärsuche für ein sortiertes Array von n Zahlen inspiziert.

2. Frage: Wie groß ist $T(n)$?

Hier ist eine rekursive Definition von $T(n)$:

(a) Rekursionsanfang: $T(0) = 0$.

(b) Rekursionsschritt: $T(n) = T(\frac{n-1}{2}) + 1$.

Wir haben $n = 2^k - 1$ gefordert. Beachte, dass $\frac{n-1}{2} = \frac{2^k-2}{2} = 2^{k-1} - 1$ gilt. Nach jedem rekursiven Aufruf wird der Exponent k also um 1 erniedrigt und $T(2^k - 1) = k$ „sollte“ folgen. Wir zeigen $T(2^k - 1) = k$ mit vollständiger Induktion nach k.

(a) Induktionsanfang: $T(2^0 - 1) = T(0) = 0$. ✓

(b) Induktionsschritt: $T(2^{k+1} - 1) = T(2^k - 1) + 1 \overset{\text{Induktionsannahme}}{=} k + 1$. ✓

Und was bedeutet das jetzt? Binärsuche muss höchstens k Zahlen inspizieren gegenüber bis zu $2^k - 1$ Zahlen für die lineare Suche: Die lineare Suche ist exponentiell langsamer als Binärsuche.

3. Frage: Wie sieht der Rekursionsbaum B für ein Array A und einen Schlüssel y aus, der nicht in A vorkommt?

Wir nehmen wieder an, dass das Array A genau $n = 2^k - 1$ Schlüssel für eine Zahl $k \in \mathbb{N}$ besitzt. Beachte, dass ein Knoten von B entweder ein Blatt ist oder `genau einen` rekursiven Aufruf verursacht. Der Rekursionsbaum B besitzt also den Aus-Grad Eins.

Wir wissen bereits, dass Binärsuche im schlimmsten Fall k Zellen inspiziert. Da der Schlüssel y nicht in A vorkommt, tritt der schlimmste Fall ein und es werden genau k Zellen inspiziert. In jedem rekursiven Aufruf der Binärsuche wird nur eine Zelle des Arrays inspiziert, nämlich die mittlere Zelle des Array-Abschnitts in dem gegenwärtig gesucht wird.

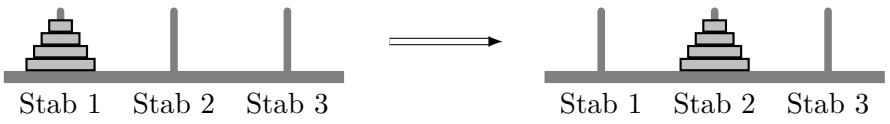
Der Rekursionsbaum B ist also ein markierter Weg der Länge k.

Binärsuche ist schnell, weil stets höchstens ein rekursiver Aufruf getätigt wird und weil die Länge des Array-Abschnitts in dem gesucht wird, mindestens halbiert wird.

Beispiel 2.5 **(Türme von Hanoi)**

Wir haben drei Stäbe mit den Nummern 1, 2 und 3. Ursprünglich liegen N Ringe auf Stab 1, wobei die Ringe in absteigender Größe auf dem Stab aufgereiht sind: Der größte Ring von Stab 1 ist also der unterste Ring. Die Stäbe 2 und 3 sind zu Anfang leer.

In einem Zug können wir einen zuoberst liegenden Ring von einem Stab auf einen anderen bewegen. Der Zug ist nur dann erlaubt, wenn der Ring auf einen größeren Ring gelegt wird oder wenn der Stab leer ist. Alle Ringe sollen am Ende auf Stab 2 liegen.



Das folgende in Pseudocode geschriebene rekursive Programm soll dieses Ziel für $N \geq 1$ erreichen.

```
void Hanoi( int N, int stab1, int stab2, int stab3)
// Annahmen: Auf allen Stäben sind die Ringe der Größe nach geordnet.
// Jeder der oberen N Ringe auf Stab „stab1“ passt auf Stab „stab2“ und „stab3“.
// Die Folge (stab1,stab2,stab3) ist eine Permutation der Zahlen 1,2 und 3.
{if (N==1)
   bewege den obersten Ring von Stab „stab1“ nach Stab „stab2“;
else {
   Hanoi(N-1,stab1,stab3,stab2);
   bewege den obersten Ring von Stab „stab1“ nach Stab „stab2“;
   Hanoi(N-1,stab3,stab2,stab1); }}
```

1. Frage: Ist `Hanoi`(N-1,stab1,stab2, stab3) korrekt?

Wir zeigen mit vollständiger Induktion nach N: Für jede Permutation (stab1,stab2,stab3) der drei Stäbe wird „Hanoi (N, stab1, stab2, stab3)“ die obersten N Ringe von stab1 auf stab2 bewegen ohne andere Ringe anzufassen. Vorausgesetzt wird, dass jeder der obersten N Ringe von stab1 auf die beiden anderen Stäbe passt.

Beachte, dass wir eine stärkere Aussage behaupten als auf den ersten Blick notwendig zu sein scheint: Wir behaupten nämlich die Richtigkeit für alle(!) Permutationen der drei Stäbe. Diese verschärfte Behauptung ist auch notwendig, um Aussagen über die beiden rekursiven Aufrufe machen zu können. Die Formulierung einer verschärften Aussage ist charakteristisch für viele Induktionsbeweise.

Induktionsanfang für $N = 1$. Richtigerweise wird der eine zuoberst liegende Ring von „stab1“ nach „stab2“ bewegt.

Induktionsschritt von N auf $N + 1$: Wir können in der *Induktionsannahme* voraussetzen, dass „Hanoi(N, stab1, stab2,stab3)“ – für jede Permutation (stab1,stab2,stab3) der drei Stäbe – die N obersten Ringe von „stab1“ nach „stab2“ bewegt ohne andere Ringe anzufassen.

Im ersten rekursiven Aufruf wird „Hanoi(N, stab1, stab3,stab2)“, nach Induktionsannahme, die obersten N Ringe von „stab1“ nach „stab3“ bewegen. Der jetzt zuoberst liegende Ring von „stab1“ wird auf „stab2“ gelegt: Nach Annahme passt dieser Ring auf „stab2“.

Da dieser Ring der größte der ursprünglichen $N + 1$ obersten Ringe von „stab1“ ist, passen alle jetzt auf „stab3“ hinzu gepackten Ringe auf „stab1“ und „stab2“. Der zweite und letzte rekursive Aufruf „Hanoi(N, stab3, stab2,stab1)“ wird deshalb nach Induktionsannahme alle durch den ersten rekursiven Aufruf auf „stab3“ bewegten Ringe erfolgreich auf „stab2“ bewegen: Damit liegen die ursprünglich obersten $N + 1$ Ringe von „stab1“ jetzt auf „stab2“.

2. Frage: Wieviele Ringbewegungen führt `Hanoi`(N-1,stab1,stab2, stab3) aus?

Sei $\mathbf{T}(\mathbf{N})$ die Anzahl der Ringbewegungen nach Aufruf des Programms Hanoi(N,stab1,stab2,stab3). (Beachte, dass diese Anzahl nicht von der Permutation (stab1,stab2,stab3) abhängt.) Wir geben eine rekursive Definition von $\mathbf{T}(\mathbf{N})$ an.

(a) Rekursionsanfang: Es ist $T(1) = 1$ und

(b) Rekursionsschritt: Es ist $T(N) = 2 \cdot T(N - 1) + 1$.

Und wie sieht ein expliziter Ausdruck für $T(N)$ aus? Es ist $T(1) = 1, T(2) = 3, T(3) = 7, T(4) = 15$ und das sieht ganz so aus als ob $T(N) = 2^N - 1$ gilt. Wir verifizieren unsere Vermutung mit vollständiger Induktion nach N.

(a) Induktionsanfang für $N = 1$: Unser Programm bewegt einen Ring und $1 = 2^1 - 1$. ✓

(b) Induktionsschritt von N auf $N + 1$: Wir können in der *Induktionsannahme* voraussetzen, dass $T(N) = 2^N - 1$ gilt. Dann folgt

$$T(N + 1) = 2 \cdot T(N) + 1 \stackrel{\text{Ind.ann}}{=} 2 \cdot (2^n - 1) + 1 = 2^{N+1} - 1$$

und das war zu zeigen. ✓

3. Frage: Wie sieht der Rekursionsbaum B für `Hanoi`(N,1,2,3) aus?

Es werden zwei rekursive Aufrufe mit dem Parameter $N-1$ getätigt. Der Rekursionsbaum B hat also die Graph-Struktur eines Binärbaums.

Mit vollständiger Induktion über N zeigt man, das B die Graph-Struktur eines vollständigen Binärbaums der Höhe $N-1$ besitzt: Trifft diese Aussage für die beiden rekursiven Aufrufe zu, dann gilt sie auch für den „Master-Aufruf“. Die Anzahl der Ringbewegungen ist so groß, weil für $N > 1$ stets zwei rekursive Aufrufe getätigt werden, wobei der Parameter N nur um Eins reduziert wird:

B ist ein vollständiger Binärbaum der Höhe $N-1$ und besitzt $2^{N+1}-1$ Knoten.

2.4 Schatztruhe

Wir stellen zuerst Schreibweisen (oder Notationen) für wichtige Begriffe zusammen. Geschlossene Ausdrücke für Summen, Anzahlbestimmungen häufig auftretender Mengen und Rechenregeln für den Logarithmus folgen.

2.4.1 Schreibweisen

0. Notation aus der Logik: $\neg\phi$ ist die **Negation** der Formel ϕ. Für Formeln ϕ und ψ ist $\phi \wedge \psi, \phi \vee \psi$ und $\phi \oplus \psi$ die **Konjunktion**, **Disjunktion**, bzw. das **exklusive Oder** (**XOR**) von ϕ und ψ. Die Formeln $\phi \to \psi$ und $\phi \leftrightarrow \psi$ bezeichnen die **Implikation** und die **Äquivalenz**.

   Man schreibt $\phi \equiv \psi$, falls die Formel ψ genau dann wahr ist, wenn ϕ wahr ist. Beachte $(\phi \to \psi) \equiv (\neg\phi \vee \psi)$ und $(\phi \leftrightarrow \psi) \equiv ((\phi \to \psi) \wedge (\psi \to \phi))$.

1. Wir erinnern an die Mengennotationen aus Abschnitt 2.1: Die Beschreibung $x \in M$ bedeutet, dass x ein **Element** der **Menge** M ist, $x \notin M$ hingegen bedeutet, dass x kein Element von M ist. Die Schreibweise

   $$M' := \{x \in M \,:\, x \text{ hat Eigenschaft } E\,\}$$

   drückt aus, dass wir die Menge M' als die Menge aller Elemente $x \in M$ definieren, die die Eigenschaft E besitzen. Alternativ schreiben wir auch

   $$M' := \{x \,:\, x \in M \text{ und } x \text{ hat Eigenschaft } E\,\}.$$

   - Für Mengen M_1, M_2 bedeutet $M_1 \subseteq M_2$, bzw. $M_2 \supseteq M_1$ dass M_1 eine Teilmenge von M_2 ist. $\mathcal{P}(M) := \{\, M' \,:\, M' \subseteq M\,\}$ ist die **Potenzmenge** von M und $\emptyset$ ist die leere Menge. Beachte, dass die leere Menge Teilmenge jeder Menge ist.
   - Wir erinnern an die Mengenoperationen

   $$\begin{aligned} \textbf{Durchschnitt } M_1 \cap M_2 &:= \{x \,:\, x \in M_1 \text{ und } x \in M_2\,\}, \\ \textbf{Vereinigung } M_1 \cup M_2 &:= \{x \,:\, x \in M_1 \text{ oder } x \in M_2\,\}, \\ \textbf{Mengendifferenz } M_1 \setminus M_2 &:= M_1 - M_2 := \{x \,:\, x \in M_1 \text{ und } x \notin M_2\,\}. \end{aligned}$$

2. **Natürliche, ganze, rationale** und **reelle Zahlen**:

   $\mathbb{N}$:= Menge der natürlichen Zahlen := $\{0, 1, 2, 3, ...\}$

   $\mathbb{N}_{>0}$:= Menge der positiven natürlichen Zahlen := $\{1, 2, 3, ...\}$

   $\mathbb{Z}$:= Menge der ganzen Zahlen := $\{0, 1, -1, 2, -2, 3, -3, ...\}$

   $\mathbb{Q}$:= Menge der rationalen Zahlen := $\{\frac{a}{b} : a, b \in \mathbb{Z}\, b \neq 0\}$

   $\mathbb{R}$:= Menge der reellen Zahlen

   Für eine reelle Zahl $x \in \mathbb{R}$ ist $\lfloor x \rfloor$ die größte ganze Zahl kleiner oder gleich x und $\lceil x \rceil$ die kleinste ganze Zahl größer oder gleich x.

3. Das **kartesische Produkt** der Mengen $M_1, \ldots, M_k$ ist die Menge

$$M_1 \times \cdots \times M_k = \{\, (m_1, \ldots, m_k) : m_1 \in M_1, \ldots, m_k \in M_k \,\}.$$

4. Graphen werden durch Paare $G = (V, E)$ bezeichnet, wobei V die Menge der Knoten und E die Menge der Kanten ist. Wenn G **ungerichtet** ist, dann ist $E \subseteq \{\, \{u, v\} : u, v \in V \text{ und } u \neq v\}$. Wenn G ein **gerichteter Graph** ist, dann ist stets $E \subseteq V \times V$.

5. Um eine **Funktion** (oder **Abbildung**) f zu beschreiben, benutzen wir den Formalismus $f : A \to B$ um anzugeben, dass die Funktion genau auf allen Elementen $x \in A$ des **Definitionsbereichs** A definiert ist. Der Wert von f für das **Argument** $x \in A$ wird durch $f(x)$ bezeichnet; beachte, dass $f(x)$ ein Element des **Bildbereichs** B ist, es gilt also $f(x) \in B$.

   - f ist **injektiv**, falls es für jedes Element $b \in B$ des Bildbereichs *höchstens* ein Element a des Definitionsbereichs mit $f(a) = b$ gibt.
   - f ist **surjektiv**, falls es für jedes Element $b \in B$ des Bildbereichs *mindestens* ein Element a des Definitionsbereichs mit $f(a) = b$ gibt.
   - f ist **bijektiv**, falls es für jedes Element $b \in B$ des Bildbereichs *genau* ein Element a des Definitionsbereichs mit $f(a) = b$ gibt.

   Die Funktionen $f : A \to B$ und $g : B \to C$ lassen sich **hintereinander ausführen**: Zuerst wende f auf das Argument x an, dann wende g auf das Zwischenergebnis $f(x)$ an, um das Endergebnis $g(f(x))$ zu erhalten. Man bezeichnet die Hintereinanderausführung auch als **Komposition** und benutzt die Notation $g \circ f$.

6. Summen- und Produktsymbole haben sich für Summen und Produkte mit vielen Operanden eingebürgert.

   (a) Ist $k \in \mathbb{N}_{>0}$ und sind $z_1, \ldots, z_k$ (natürliche, ganze, rationale, reelle oder komplexe) Zahlen, so schreiben wir

$$\sum_{i=1}^{k} z_i \qquad \text{bzw.} \qquad \sum_{i \in \{1,\ldots,k\}} z_i$$

   um die Summe der Zahlen $z_1, \ldots, z_k$ zu bezeichnen (d.h. die Zahl $z_1 + \cdots + z_k$). Wir schreiben

$$\prod_{i=1}^{k} z_i \qquad \text{bzw.} \qquad \prod_{i \in \{1,\ldots,k\}} z_i$$

   um das Produkt der Zahlen $z_1, \ldots, z_k$ zu bezeichnen (d.h. die Zahl $z_1 \cdot \cdots \cdot z_k$).

(b) Sind $M_1, \ldots, M_k$ Mengen, so schreiben wir

$$\bigcup_{i=1}^{k} M_i \qquad \text{bzw.} \qquad \bigcup_{i \in \{1,\ldots,k\}} M_i$$

um die Vereinigung der Mengen $M_1, \ldots, M_k$ zu bezeichnen (d.h. die Menge $M_1 \cup \cdots \cup M_k$).
Wir schreiben

$$\bigcap_{i=1}^{k} M_i \qquad \text{bzw.} \qquad \bigcap_{i \in \{1,\ldots,k\}} M_i$$

um den Durchschnitt der Mengen $M_1, \ldots, M_k$ zu bezeichnen (d.h. die Menge $M_1 \cap \cdots \cap M_k$).

(c) Ist K eine Menge, deren Elemente Teilmengen einer Menge U sind (d.h.: $K \subseteq \mathcal{P}(U)$), so ist

$$\bigcup_{M \in K} M \quad := \quad \{x \in U \ : \ \text{es existiert eine Menge } \ M \in K, \text{ so dass } x \in M \ \}$$

die Vereinigung aller Mengen $M \in K$ (d.h. die Menge aller Elemente x, die in mindestens einer Menge $M \in K$ liegen). Analog ist

$$\bigcap_{M \in K} M \quad := \quad \{x \in U \ : \ \text{für alle } \ M \in K \text{ gilt } x \in M\}$$

der Durchschnitt aller Mengen $M \in K$ (d.h. die Menge aller Elemente x, die in jeder Menge $M \in K$ liegen).

2.4.2 Geschlossene Ausdrücke für Summen

Lemma 2.9 (Summenformeln) *Für alle natürlichen Zahlen $n \geq 0$ gilt:*

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

(b) $\displaystyle\sum_{i=0}^{n} i^2 = \frac{1}{6} \cdot n \cdot (n+1) \cdot (2n+1).$

(c) $\displaystyle\sum_{i=0}^{n} a^i = \begin{cases} \frac{a^{n+1}-1}{a-1} & \textit{falls } \ a \neq 1 \\ n+1 & \textit{sonst} \end{cases}$

(d) $\displaystyle\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$ *für eine reelle Zahl a mit $-1 < a < 1$.*

Beweis: Teil (a) wird in Satz 2.6 und Teil (c) in Satz 2.7 gezeigt. Teil (d) folgt aus Teil (c), da

$$\lim_{n \to \infty} \frac{a^{n+1} - 1}{a - 1} = \frac{1}{1-a}.$$

Teil (c) zeigt man mit vollständiger Induktion nach n. □

2.4.3 Größe von endlichen Mengen

Zur Erinnerung: Für eine endliche Menge M bezeichnen wir die Anzahl der Elemente von M mit $|M|$. Seien A und B endliche Mengen. Dann gilt:

$$|A| = |B| \;\Leftrightarrow\; \text{es gibt eine bijektive Funktion von } A \text{ nach } B.$$

Der Binomialkoeffizient $\binom{n}{k}$ ist für natürliche Zahlen n, k mit $0 \leq k \leq n$ definiert und zwar setzen wir

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!},$$

wobei $0! := 1$ und $r! := 1 \cdot 2 \cdots r$ für alle natürlichen Zahlen $r \geq 1$.

Binomialkoeffizienten treten im binomischen Lehrsatz beim Ausmultiplizieren der Potenz $(a+b)^n$ auf. Desweiteren werden wir gleich sehen, dass $\binom{n}{k}$ mit der Anzahl von k-elementigen Teilmengen einer n-elementigen Menge übereinstimmt.

Lemma 2.10 (Binomialkoeffizienten) *Für alle natürlichen Zahlen $n, k \in \mathbb{N}$ mit $k \leq n$ gilt:*

(a) $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, *falls* $k \geq 1$

(b) $\binom{n}{k} = \binom{n}{n-k}$ *und*

(c) $\binom{n}{k} = \binom{n-1}{k-1} \cdot \frac{n}{k}$ *falls* $k \geq 1$.

(d) Binomischer Lehrsatz: Es gilt

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}.$$

Beweis: Die Aussagen (a), (b) und (c) lassen sich mit einem direkten Beweis verifizieren. Der binomische Lehrsatz kann mit vollständiger Induktion nach n gezeigt werden. □

Aufgabe 6
Beweise den binomischen Lehrsatz. Nimm dazu die Teile (a), (b) und (c) aus Lemma 2.10 an.

Satz 2.11 *Die Kardinalität wichtiger Mengen*

(a) Sei M eine Menge von n Elementen.

1. *M hat genau $\binom{n}{k}$ Teilmengen der Größe k*
2. *und es ist $|\mathcal{P}(M)| = 2^n$, M hat also genau 2^n Teilmengen.*

(b) Sei $k \in \mathbb{N}_{>0}$ und seien $M_1, \ldots, M_k$ endliche Mengen. Dann gilt:

$$|\, M_1 \times \cdots \times M_k \,| \;=\; \prod_{i=1}^{k} |M_i|.$$

(c) A und B seien endliche Mengen. Dann gibt es genau $|B|^{|A|}$ Funktionen $f : A \to B$:

$$|Abb(A, B)| = |B|^{|A|}.$$

Beweis: **(a1)** Ohne Beschränkung der Allgemeinheit nehmen wir $S = \{1, \ldots, n\}$ an und führen einen Beweis durch vollständige Induktion nach n.

Für den Induktionsanfang $n = 1$ ist nur zu beobachten, dass es genau eine Menge mit keinen Elementen gibt und es ist $\binom{1}{0} = 1$. Desweiteren gibt es genau eine Menge mit einem Element, nämlich die Menge selbst und es ist $\binom{1}{1} = 1$.

Für den Induktionsschritt unterscheiden wir, ob eine k-elementige Teilmenge $T \subseteq S$ das Element n enthält oder nicht. Wenn ja, dann ist $T \setminus \{n\}$ eine $(k-1)$-elementige Teilmenge von $\{1, \ldots, n-1\}$ und nach Induktionsvoraussetzung gibt es genau $\binom{n-1}{k-1}$ $(k-1)$-elementige Teilmengen von $\{1, \ldots, n-1\}$. Wenn nein, dann verbleiben genau $\binom{n-1}{k}$ Teilmengen, denn wir müssen die k-elementigen Teilmengen von $\{1, \ldots, n-1\}$ zählen. Die Behauptung folgt jetzt aus Teil (b), denn es ist $\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$.

(a2) Wir wenden den binomischen Lehrsatz für $a = b = 1$ an und erhalten $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. Aber $\binom{n}{k}$ stimmt überein mit der Anzahl der Teilmengen von M der Mächtigkeit k und $\sum_{k=0}^{n} \binom{n}{k}$ ist die Anzahl der Teilmengen von M.

(b) Wende vollständige Induktion nach k an.

(c) Wir konstruieren eine bijektive Funktion

$$b : \mathrm{Abb}(A, B) \to B^{|A|},$$

von der Menge aller Abbildungen $f : A \to B$ in das $|A|$-fache Potenz der Menge B: Definiere nämlich $b(f)$ als die Funktionstabelle von f, die in „Position“ $a \in A$ den Funktionswert $f(a)$ vermerkt. Funktionstabellen sind $|A|$-Tupel mit Komponenten in B, da für jedes Argument $a \in A$ der Funktionswert $f(a) \in B$ vermerkt wird.

Warum ist b bijektiv? Weil Funktionstabellen nur eine andere Darstellung der Funktionen sind. Die Behauptung folgt jetzt aus Teil (b). □

Beispiel 2.6 **(Die Fakultät)** n Teilnehmer nehmen an einem Rennen teil. Wieviele verschiedene Reihenfolgen gibt es für den Zieleinlauf? Wir beschreiben die Anzahl

$$\mathrm{fak}(n)$$

der verschiedenen Reihenfolgen mit einer rekursiven Definition.

- Rekursionsanfang für $n = 1$: Es ist $\mathrm{fak}(1) := 1$.
- Wir überlegen uns zuerst, dass es bei $n+1$ Teilnehmern genau $n+1$ verschiedene Gewinner des Rennens geben kann. Wenn wir aber den Gewinner kennen, dann gibt es genau $\mathrm{fak}(n)$ verschiedene Reihenfolgen für den Zieleinlauf der verbleibenden n Teilnehmer.

  Rekursionsschritt von n auf $n + 1$: Es ist $\mathrm{fak}(n + 1) := (n + 1) \cdot \mathrm{fak}(n)$.

Wir behaupten, dass

$$\text{fak}(n) \quad = \quad n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 = \prod_{i=1}^{n} i$$

gilt und beweisen diese Behauptung mit vollständiger Induktion. Der INDUKTIONSANFANG für $n = 1$ ist klar, denn nach Definition ist $\text{fak}(1) = 1$.

Der INDUKTIONSSCHRITT von n auf $n+1$ ist auch klar, denn es ist $\text{fak}(n) = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$ nach Induktionsannahme und die Behauptung $\text{fak}(n+1) = (n+1) \cdot n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$ folgt aus der Definition $\text{fak}(n+1) := (n+1) \cdot \text{fak}(n)$.

Notation: Die Funktion $\text{fak} : \mathbb{N}_{>0} \to \mathbb{N}$ wird **Fakultät** genannt. Meistens schreibt man $n!$ statt $\text{fak}(n)$ und spricht $n!$ als „n Fakultät“ aus. □Ende Beispiel 2.6

Definition 2.12 M sei eine endliche Mengen. Eine bijektive Funktion $f : M \to M$ wird auch eine **Permutation** der Menge M genannt.

Wieviele Permutationen besitzt eine endliche Menge M?

Satz 2.12 *M, M_1, M_2 seien Mengen mit n Elementen.*

(a) Es gibt genau $n!$ bijektive Funktionen $f : M_1 \to M_2$.

(b) Die Menge M besitzt genau $n!$ Permutationen.

Wenn M_1 die Menge der n Teilnehmer eines Rennens ist, dann entspricht eine bijektive Funktion $f : M_1 \to \{1, \ldots, n\}$ einer möglichen Reihenfolge im Zieleinlauf. Teil (a) verallgemeinert also unser Ergebnis über die Anzahl verschiedener Zieleinläufe. Beachte, dass auch Teil (b) eine Konsequenz von Teil (a) ist, wenn wir nämlich $M_1 := M$ und $M_2 := M$ setzen.

Beweis: Wir können unsere Argumentation für die Anzahl der verschiedenen Reihenfolgen beim Zieleinlauf übernehmen, denn die Anzahl $b(n)$ der bijektiven Funktionen zwischen zwei n-elementigen Mengen besitzt die rekursive Definition

$$b(1) := 1, \; b(n+1) := (n+1) \cdot b(n)$$

und diese rekursive Definition stimmt mit der rekursiven Definition der Fakultät überein. □

Aufgabe 7
Ein Handlungsreisender muss jede von n Städten genau einmal besuchen. Gesucht ist eine Rundreise minimaler Länge. Wieviele verschiedene Rundreisen gibt es?

2.4.4 Der Logarithmus

Definition 2.13 Seien $a > 1$ und $x > 0$ reelle Zahlen. $\log_a(x)$ ist der Logarithmus von x zur Basis a und stimmt mit ℓ genau dann überein, wenn $a^\ell = x$ gilt.

Warum ist der Logarithmus eine zentale Funktion für die Informatik?

Aufgabe 8
Wieviele Bitpositionen besitzt die Binärdarstellung einer natürlichen Zahl n?

Bevor wir Aussagen über den Logarithmus machen können, müssen wir uns daran erinnern, wie die Umkehrung des Logarithmus, nämlich die Exponentiation funktioniert: Für alle reellen Zahlen $a \geq 0$ und x, y gilt

$$a^x \cdot a^y = a^{x+y} \tag{2.2}$$
$$(a^x)^y = a^{x \cdot y} \tag{2.3}$$
$$a^x = a^y \Rightarrow x = y. \tag{2.4}$$

Lemma 2.13 (Logarithmen) *$a, b > 1$ und $x, y > 0$ seien reelle Zahlen. Dann gilt*

(a) $\log_a(x \cdot y) = \log_a(x) + \log_a(y)$.

(b) $\log_a(x^y) = y \cdot \log_a(x)$.

(c) $a^{\log_a x} = x$.

(d) $\log_a(x) = (\log_a(b)) \cdot (\log_b(x))$.

(e) $b^{\log_a(x)} = x^{\log_a(b)}$.

Beweis: (a) Angenommen $\log_a(x) = \ell_x$ und $\log_a y = \ell_y$. Aus Definition 2.13 folgt $a^{\ell_x} = x$ und $a^{\ell_y} = y$. Wir multiplizieren und erhalten mit (2.2)

$$x \cdot y = a^{\ell_x} \cdot a^{\ell_y} = a^{\ell_x + \ell_y}.$$

Damit gilt $x \cdot y = a^{\ell_x + \ell_y}$ und deshalb ist $\log_a(x \cdot y) = \ell_x + \ell_y = \log_a(x) + \log_a(y)$.

(b) Angenommen $\log_a x = \ell$. Dann ist $a^\ell = x$. Wir wenden (2.3) an und erhalten

$$x^y = (a^\ell)^y = a^{y \cdot \ell}.$$

Aber dann erhalten wir $\log_a(x^y) = y \cdot \ell = y \cdot \log_a(x)$.

(c) Angenommen $\log_a x = \ell$. Dann ist $a^\ell = x$ und deshalb ist $a^{\log_a(x)} = a^\ell = x$.

(d) Wir wenden Teil (c) an und erhalten $a^{\log_a(x)} = x$ sowie $a^{(\log_a(b))} = b$ und $b^{\log_b(x)} = x$. Aber dann ist

$$a^{(\log_a(b)) \cdot (\log_b(x))} = \left(a^{\log_a(b)}\right)^{\log_b(x)} = b^{\log_b(x)} = x = a^{\log_a(x)}.$$

Also folgt $a^{\log_a(x)} = a^{(\log_a(b)) \cdot (\log_b(x))}$ und deshalb $\log_a(x) = (\log_a(b)) \cdot (\log_b(x))$ aus (2.4).

(e) Wir wenden Teil (d) an und erhalten

$$b^{\log_a(x)} = b^{(\log_a(b)) \cdot (\log_b(x))} = \left(b^{\log_b(x)}\right)^{\log_a(b)} = x^{\log_a(b)}$$

und das war zu zeigen. □

Eine erste Anwendung: Sei k eine natürliche Zahl. Wir möchten das Wachstum der Funktionen n^k und 2^n vergleichen. Dazu wenden wir zuerst Teil (c) von Lemma 2.13 an

und schreiben n^k als Zweierpotenz: Es ist $n^k = 2^{\log_2(n^k)}$. Mit Teil (b) folgt $\log_2(n^k) = \log_2(n) + \log_2(n^{k-1}) = \cdots = k \log_2(n)$. Also ist $n^k = 2^{k \log_2(n)}$ und als Konsequenz

$$\lim_{n\to\infty} \frac{n^k}{2^n} = \lim_{n\to\infty} 2^{k\log_2(n)-n}.$$

Im nächsten Abschnitt sehen wir, dass der Logarithmus von n sehr viel schwächer als n wächst: Der Exponent $k \log_2 n - n$ strebt gegen $-\infty$ und damit strebt sogar der Logarithmus des Quotienten $n^k/2^n$ gegen Null. Die Zweierpotenz 2^n wächst also sehr, sehr, sehr viel schneller als jedes „Polynom“ n^k.

Eine zweite Anwendung: $a > 1$ und $b > 1$ seien zwei reelle Zahlen. Führt der Wechsel der Basis zu einem stärkeren oder schwächerem Wachstum des Logarithmus, bzw. gibt es eine Beziehung zwischen $\log_a(n)$ und $\log_b(n)$? Ein Wechsel der Basis hat keine Konsequenz für das Wachstum, denn Teil (d) besagt, dass

$$\log_a(n) = (\log_a(b)) \cdot \log_b(n) \tag{2.5}$$

und die beiden Logarithmen, nämlich $\log_a(n)$ und $\log_b(n)$, unterschieden sich nur um den konstanten Faktor $\log_a(b)$.

2.5 Einige Grundlagen aus der Stochastik

Wir werden in dieser Vorlesung meistens die pessimistische Sicht des Worst-Case-Szenarios analysieren. Das heißt, wir werden uns bei der Analyse von Lösungsstrategien fragen, was schlimmstenfalls geschehen könnte. Manchmal verzerrt dieser Blick aber die wahren Gegebenheiten zu stark, und eine Betrachtung dessen, was man vernünftigerweise *erwarten* sollte, ist geboten. Insbesondere verliert die Worst-Case-Analyse drastisch an Wert, wenn der Worst-Case extrem *unwahrscheinlich* ist.

Eine kurze Wiederholung elementarer Begriffe der Stochastik ist daher angebracht. Wir konzentrieren uns auf endliche **Wahrscheinlichkeitsräume**. Also ist eine endliche Menge Ω von **Elementarereignissen** gegeben sowie eine **Wahrscheinlichkeitsverteilung** p, die jedem Elementarereignis $e \in \Omega$ die Wahrscheinlichkeit $p(e)$ zuweist. Desweiteren muss $\sum_{e\in\Omega} p(e) = 1$ und $p(e) \geq 0$ für alle $e \in \Omega$ gelten. Ein **Ereignis** E ist eine Teilmenge von Ω und $\text{prob}[E] = \sum_{e\in E} p(e)$ ist die Wahrscheinlichkeit von E.

Beispiel 2.1 Wir wählen die Menge der 37 Fächer eines Roulettspiels als unsere Menge von Elementarereignissen. Die Ereignisse *gerade, ungerade, rot* oder *schwarz* können dann als Vereinigung von Elementarereignissen beschrieben werden. Die Wahrscheinlichkeit eines solchen Ereignisses ergibt sich aus der Summe der jeweiligen Elementarereignisse.

Lemma 2.14 (Rechnen mit Wahrscheinlichkeiten) *Seien A und B Ereignisse über dem endlichen Wahrscheinlichkeitsraum $\Omega = \{e_1, \ldots, e_n\}$ und sei $p = (p_1, \ldots, p_n)$ eine Wahrscheinlichkeitsverteilung.*

(a) $prob[A \cap B] = \sum_{e\in A\cap B} p(e)$. *Insbesondere ist* $0 \leq prob[A \cap B] \leq \min\{prob[A], prob[B]\}$.

(b) $prob[A \cup B] = \sum_{e\in A\cup B} p(e)$. *Insbesondere ist*

$$\begin{aligned} max\{prob[A], prob[B]\} &\leq prob[A \cup B] \\ &= prob[A] + prob[B] - prob[A \cap B] \leq prob[A] + prob[B]. \end{aligned}$$

(c) $prob[\neg A] = \sum_{e \notin A} p(e) = 1 - prob[A]$.

Erwartungswerte spielen bei der Analyse von erwarteten Laufzeiten eine zentrale Rolle. Die allgemeine Definition ist wie folgt.

Definition 2.14 **(a)** Die Menge $A \subseteq \mathbb{R}$ sei gegeben. Eine Zufallsvariable $X : \Omega \to A$ wird durch eine Wahrscheinlichkeitsverteilung $(q(a) : a \in A)$ spezifiziert. Wir sagen, dass $q(a)$ die Wahrscheinlichkeit des Ereignisses $X = a$ ist.

(b) Sei X eine Zufallsvariable und sei $q(a)$ die Wahrscheinlichkeit für das Ereignis $X = a$. Dann wird der Erwartungswert von X durch

$$E[X] = \sum_{a \in A} a \cdot q(a)$$

definiert.

Die möglichen Ausgänge a der Zufallsvariable werden also mit ihrer Wahrscheinlichkeit gewichtet und addiert. Eine zentrale Eigenschaft des Erwartungswerts ist seine Additivität.

Lemma 2.15 *Für alle Zufallsvariablen X und Y ist $E[X + Y] = E[X] + E[Y]$.*

Der Informationsgehalt eines Erwartungswertes hängt dabei vom konkreten Experiment ab. Dass man bei einem handelsüblichen Würfel eine 3.5 erwartet ist eine wenig nützliche Information. Dass der erwartete Gewinn bei einem Einsatz von 10 Euro auf rot am Roulettetisch $\frac{18}{37} \cdot 20Euro + \frac{19}{37} \cdot 0Euro = 9.73Euro$ ist, hat dagegen die nützliche Botschaft: *Finger weg.*

Aufgabe 9
Wir betrachten die Menge $\Omega = \{1, 2, \ldots, n\}$. In einem ersten Experiment bestimmen wir eine Menge $A \subseteq \Omega$, indem wir jedes Element aus Ω mit Wahrscheinlichkeit p_A in A aufnehmen. Wir wiederholen das Experiment und bilden eine Menge B, wobei wir jedes Element aus Ω mit Wahrscheinlichkeit p_B in B aufnehmen.

(a) **Bestimme** $E(|A \cap B|)$, den Erwartungswert der Mächtigkeit der Schnittmenge.

(b) **Bestimme** $E(|A \cup B|)$, den Erwartungswert der Mächtigkeit der Vereinigungsmenge.

Aufgabe 10
Wir spielen ein Spiel gegen einen Gegner. Der Gegner denkt sich zwei Zahlen aus und schreibt sie für uns nicht sichtbar auf je einen Zettel. Wir wählen *zufällig* einen Zettel und lesen die darauf stehende Zahl. Sodann haben wir die Möglichkeit, diese Zahl zu behalten oder sie gegen die uns unbekannt gebliebene Zahl zu tauschen. Sei x die Zahl, die wir am Ende haben, und y die andere. Dann ist unser (möglicherweise negativer) Gewinn $x - y$.

- Wir betrachten Strategien S_t der Form *„Gib Zahlen $< t$ zurück und behalte diejenigen $\geq t$“*. Analysiere den Erwartungswert $E_{x,y}(Gewinn(S_t))$ des Gewinns dieser Strategie in Abhängigkeit von t, x und y.
- Gib eine randomisierte Strategie an, die für beliebige $x \neq y$ einen positiven erwarteten Gewinn für uns aufweist.

Häufig untersuchen wir wiederholte Zufallsexperimente. Wenn die Wiederholungen unabhängig voneinander sind, hilft die Binomialverteilung weiter.

Lemma 2.16 (Binomialverteilung) *Sei A ein Ereignis, welches mit Wahrscheinlichkeit p auftritt. Wir führen n Wiederholungen des betreffenden Experimentes durch und zählen, wie häufig ein Erfolg eintritt, das heißt wie häufig A eingetreten ist. Die Zufallsvariable X möge genau dann den Wert k annehmen, wenn k Erfolge vorliegen.*

(a) Die Wahrscheinlichkeit für k Erfolge ist gegeben durch

$$prob[X = k] = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}.$$

(b) Die Wahrscheinlichkeit, dass die Anzahl der Erfolge im Intervall $[a, b]$ liegt ist also

$$prob[X \in [a, b]\ ] = \sum_{k=a}^{b} \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}.$$

(c) $E[X] = n \cdot p$ ist die erwartete Anzahl der Erfolge.

Damit haben wir den Erwartungswert einer binomialverteilten Zufallsvariable berechnet. Zufallsvariablen mit prinzipiell unendlich vielen Ausgängen werden uns zum Beispiel immer dann begegnen, wenn wir uns fragen, wie lange man auf das Eintreten eines bestimmten Ereignisses warten muss. Warten wir zum Beispiel am Roulettetisch auf die erste 0 des Abends, so gibt es keine Anzahl von Runden, so dass die erste 0 innerhalb dieser Rundenzahl gefallen sein *muss*. Natürlich wird aber anhaltendes Ausbleiben immer unwahrscheinlicher und deswegen ergibt sich trotzdem ein endlicher Erwartungswert. Wir haben damit die *geometrische Verteilung* beschrieben.

Lemma 2.17 (geometrische Verteilung) *Sei A ein Ereignis mit Wahrscheinlichkeit p. Die Zufallsvariable X beschreibe die Anzahl der Wiederholungen des Experimentes bis zum ersten Eintreten von A.*

(a) Die Wahrscheinlichkeit, dass X den Wert k annimmt ist

$$prob[X = k] = (1-p)^{k-1} \cdot p.$$

(b) Der Erwartungswert ist $E(X) = \frac{1}{p}$.

Aufgabe 11

Wir nehmen in einem Casino an einem Spiel mit Gewinnwahrscheinlichkeit $p = 1/2$ teil. Wir können einen beliebigen Betrag einsetzen. Geht das Spiel zu unseren Gunsten aus, erhalten wir den Einsatz zurück und zusätzlich denselben Betrag aus der Bank. Endet das Spiel ungünstig, verfällt unser Einsatz. Wir betrachten die folgende Strategie:

```
i:=0
REPEAT
    setze 2^i$
    i:=i+1
UNTIL(ich gewinne zum ersten mal)
```

Bestimme den erwarteten Gewinn dieser Strategie und die erwartete notwendige Liquidität (also den Geldbetrag, den man zur Verfügung haben muss, um diese Strategie ausführen zu können).

Kapitel 3

Laufzeitmessung

Um sinnvoll über die Laufzeit eines Programms sprechen zu können, müssen wir zuerst die Länge der Eingabe messen. Die Laufzeitfunktion eines Programms weist dann jeder Eingabelänge n die größtmögliche Schrittzahl des Programms für eine Eingabe der Länge n zu. Wie beurteilt man wie gut das Programm *skaliert*, d.h. wie stark die Laufzeit zunimmt, wenn die Eingabelänge vergrößert wird? Diese Frage beantworten wir mit Hilfe der asymptotischen Notation.

3.1 Eingabelänge und Worst-Case Laufzeit

Wie sollte man die Laufzeit eines Programms P messen? Selbst wenn die Laufzeit von P für jede mögliche Eingabe bekannt ist, wird uns das nicht unbedingt helfen, die vermutliche Laufzeit für „unsere“ Eingaben vorauszusagen, da es ja gar nicht klar ist, welche Eingaben wir denn zu erwarten haben. Wir sollten aber zumindest ungefähr wissen wie „groß“ unsere Eingabe sein werden. Genau aus diesem Grund

1. misst man zuerst die **Eingabelänge** durch eine natürliche Zahl n und
2. bestimmt dann die **größte Laufzeit** einer Eingabe der Länge n.

Wir nehmen also eine pessimistische Sichtweise ein, weil wir für jede Eingabelänge nur die schlechteste Laufzeit festhalten. Wenn also die Funktion

$$\text{Länge} : \text{Menge aller möglichen Eingaben} \to \mathbb{N}$$

einer Eingabe x die Eingabelänge „Länge(x)“ zuweist, dann ist

$$\text{Zeit}_P(n) = \max\{\ \text{Anzahl der Schritte von Programm } P \text{ für Eingabe } x\ :\ \text{Länge}(x) = n\ \}$$

die „worst-case Laufzeit“ von P für Eingaben der Länge n.

Beispiel 3.1 **(Das Sortierproblem)**

Im Sortierproblem erwarten wir ein Array A von „Schlüsseln“, wobei je zwei Schlüssel miteinander vergleichbar sind. Unsere Aufgabe besteht darin, das Array in eine aufsteigende Folge „umzusortieren“. Es liegt nahe, die Eingabelänge durch die Anzahl der zu sortierenden Schlüssel zu definieren, wir setzen also

$$\text{Länge}(A) :=\ \text{Anzahl der Schlüssel von } A.$$

Beispiel 3.2 **(Graph-theoretische Probleme)**

Wir erhalten einen ungerichteten Graphen $G = (V, E)$ und sollen festzustellen, ob G zusammenhängend ist. Hier, aber auch für andere Graph-theoretische Fragestellungen, ist es sinnvoll, die Länge der Eingabe durch die Summe der Anzahl der Knoten und der Anzahl der Kanten zu definieren. Wir setzen also für $G = (V, E)$,

$$\text{Länge}(G) = |V| + |E|.$$

3.2 Die asymptotischen Notation

Wenn wir wissen möchten, wie das Programm P „**skaliert**“, dann interessieren wir uns für das **asymptotische Wachstum** der Laufzeit-Funktion

$$\text{Zeit}_P : \mathbb{N} \to \mathbb{N},$$

wir möchten nämlich wissen, um wieviel die Laufzeit ansteigt, wenn die Eingabelänge vergrößert wird. Wenn wir beispielsweise die Laufzeit $N = \text{Zeit}_P(n)$ kennen und die Eingabelänge n verdoppeln, dann möchten wir $M = \text{Zeit}_P(2n)$ bestimmen:

1. Wenn $\text{Zeit}_P(n) = n$, dann ist $M = 2 \cdot N$,
2. wenn $\text{Zeit}_P(n) = n^2$, dann ist $M = 4 \cdot N$,
3. wenn $\text{Zeit}_P(n) = n^k$, dann ist $M = 2^k \cdot N$,
4. wenn $\text{Zeit}_P(n) = 2^n$, dann ist $M = 2^{2n} = (2^n)^2 = N^2$, Oooops $\lightning$.

Wir drücken das asymptotische Wachstum mit Hilfe der asymptotischen Notation aus. Wir definieren die asymptotische Notationen für Funktionen $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, haben aber stets Laufzeitfunktionen h im „Hinterkopf“, die einer Eingabelänge $n \in \mathbb{N}$ eine nicht-negative Laufzeit $h(n) \in \mathbb{N}$ zuweisen.

Definition 3.1 Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ Funktionen.

(a) **(Groß-Oh Notation)** $f = O(g) :\Leftrightarrow$ Es gibt eine positive Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ gilt

$$f(n) \leq c \cdot g(n).$$

(b) **(Groß-Omega Notation)** $f = \Omega(g) :\Leftrightarrow g = O(f)$.

(c) **(Theta-Notation)** $f = \Theta(g) :\Leftrightarrow f = O(g)$ und $g = O(f)$.

(d) **(Klein-Oh Notation)** $f = o(g) :\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

(e) **(Klein-Omega Notation)** $f = \omega(g) :\Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Was besagen die einzelnen Notationen? $f = O(g)$ drückt aus, dass f asymptotisch nicht stärker als g wächst. Ein Programm mit Laufzeit f ist also, unabhängig von der Eingabelänge, um höchstens einen konstanten Faktor langsamer als ein Programm mit Laufzeit g. (Beachte, dass $f = O(g)$ sogar dann gelten kann, wenn $f(n)$ stets größer als $g(n)$ ist.)

Gilt hingegen $f = o(g)$, dann ist ein Programm mit Laufzeit f für hinreichend große Eingabelänge wesentlich schneller als ein Programm mit Laufzeit g, denn der Schnelligkeitsunterschied kann durch keine Konstante beschränkt werden. Während $f = \Omega(g)$ (und damit äquivalent $g = O(f)$) besagt, dass f zumindest so stark wie g wächst, impliziert $f = \Theta(g)$, dass f und g gleichstark wachsen.

Beispiel 3.3 Wie drückt man aus, dass die Funktion f unbeschränkt wächst, dass also $\lim_{n\to\infty} f(n) = \infty$ gilt? Diese Aussage ist äquivalent zu

$$1 = o(f).$$

Beispiel 3.4 (**Logarithmen: Wachstum in Abhängigkeit von der Basis**)

Das asymptotische Wachstum des Logarithmus hängt nicht von der Wahl der Basis ab. Wir behaupten nämlich, dass für alle reelle Zahlen $a > 1$ und $b > 1$ gilt

$$\log_a n = \Theta(\log_b n).$$

Wir wissen aus (2.5), dass $\log_a(n) = (\log_b(a)) \cdot \log_b(n)$ gilt. Insbesondere folgt $\log_a(n) = O(\log_b(n))$ (mit $c = \log_a b$ und $n_0 = 1$). Ganz analog ist $\log_b(n) \leq (\log_a(n)/\log_a(b))$ und $\log_b(n) = O(\log_a(n))$ folgt (mit $c = 1/\log_a b$ und $n_0 = 1$).

Aus $\log_a(n) = O(\log_b(n))$ und $\log_b(n) = O(\log_a(n))$ folgt die Behauptung $\log_a(n) = \Theta(\log_b(n))$. □

Die asymptotischen Notationen erwecken zunächst den Anschein, als würden einfach Informationen *weggeworfen*. Dieser Eindruck ist auch sicher nicht falsch. Man mache sich jedoch klar, dass wir, wenn wir eine Aussage wie etwa

$$4n^3 + \frac{2n}{3} - \frac{1}{n^2} = O(n^3)$$

machen, nur untergeordnete Summanden und konstante Faktoren weglassen. Wir werfen also sehr gezielt nachrangige Terme weg, die für hinreichend hohe Werte von n nicht ins Gewicht fallen, und wir verzichten auf die führende Konstante (hier 4), da die konkreten Konstanten bei realen Implementierungen ohnehin compiler- und rechnerabhängig sind:

> Wir reduzieren einen Ausdruck auf das asymptotisch Wesentliche.

Die asymptotischen Notationen erlauben es uns, Funktionen in verschiedene *Wachstumsklassen* einzuteilen. Was nützt uns die Information, dass die Laufzeit f eines Programms **kubisch** ist, dass also $f(n) = \Theta(n^3)$ gilt? Wenn wir ein Programm mit kubischer Laufzeit starten, haben wir natürlich keine Vorstellung von der Laufzeit, die sich in Minuten oder Sekunden ausdrücken ließe. Wir können jedoch zum Beispiel verlässlich vorhersagen, dass sich die Laufzeit des Programms, wenn man es auf eine doppelt so große Eingabe ansetzt, in etwa verachtfachen wird. Diese Prognose wird um so zutreffender sein, je größer die Eingabe ist, da sich dann die Vernachlässigung der niederen Terme zunehmend rechtfertigt.

Natrlich entscheiden konstante Faktoren in der Praxis, denn wer möchte schon mit mit einem Programm arbeiten, das um den Faktor 100 langsamer ist als ein zweites Programm. Neben der asymptotischen Analyse muss deshalb auch eine experimentelle Analyse treten, um das tatsächliche Laufzeitverhalten auf „realen Daten“ zu untersuchen. Dieser Ansatz, auch „**Algorithm Engineering**“ genannt, also die Kombination der asymptotischen und experimentellen Analyse und ihre Wechselwirkung mit dem Entwurfsprozess, sollte das Standard-Verfahren für den Entwurf und die Evaluierung von Algorithmen und Datenstrukturen sein.

Beispiel 3.1 Es ist $n^3 = O\left(\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n\right)$. Warum?

$$n^3 \leq 6\left(\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n\right)$$

falls $n \geq 1$. Die Definition der „Groß-Oh“-Relation ist also für $c = 6$ und $n_0 = 0$ erfüllt.

Es gilt aber auch $\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n = O(n^3)$. Warum? Wir erhalten

$$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n \leq \frac{1}{6}n^3 + \frac{1}{2}n^3 + \frac{1}{3}n^3 = n^3$$

und die Definition der Groß-Oh-Relation ist für $c = 1$ und $n_0 = 0$ erfüllt. Wir erhalten somit

$$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n = \Theta(n^3)$$

Aufgabe 12

(a) Zeige $f_1 = O(g_1) \wedge f_2 = O(g_2) \rightarrow f_1 + f_2 = O(g_1 + g_2) \wedge f_1 \cdot f_2 = O(g_1 \cdot g_2)$

(b) Bleibt die Aussage in (a) richtig, wenn stets O durch o oder Θ ersetzt wird?

(c) Ist stets $f(2n) = \Theta(f(n))$?

Aufgabe 13
Lokalisiere so genau wie möglich den Fehler im folgenden „Induktionsbeweis“.

Behauptung:

$$\sum_{i=1}^{n}(2i+1) = O(n)$$

Beweis durch vollständige Induktion nach n. **Verankerung** für $n = 1$: In diesem Fall ist

$$\sum_{i=1}^{1}(2i+1) = 3 = O(1). \surd$$

Induktionsschritt von n nach $n + 1$: Wir beginnen mit der Induktionsannahme

$$\sum_{i=1}^{n}(2i+1) = O(n).$$

Wir addieren auf beiden Seiten $2(n + 1) + 1$ und erhalten

$$\sum_{i=1}^{n}(2i+1) + 2(n+1) + 1 = O(n) + 2(n+1) + 1.$$

Nach einer Vereinfachung folgt

$$\sum_{i=1}^{n+1}(2i+1) = O(n) + 2n + 3.$$

Aber $(2n + 3) = O(n)$ und somit folgt

$$\sum_{i=1}^{n}(2i+1) = O(n) + O(n) = O(n). \surd$$

3.2.1 Grenzwerte

Anstatt den umständlichen Weg der Konstruktion der Konstanten c und n_0 zu gehen, können wir oft die folgenden Charakterisierungen des asymptotischen Wachstums durch Grenzwerte anwenden.

Lemma 3.1 *Der Grenzwert der Folge* $\frac{f(n)}{g(n)}$ *möge existieren und es sei* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.

(a) Wenn $c = 0$, *dann ist* $f = o(g)$.

(b) Wenn $0 < c < \infty$, *dann ist* $f = \Theta(g)$.

(c) Wenn $c = \infty$, *dann ist* $f = \omega(g)$.

(d) Wenn $0 \leq c < \infty$, *dann ist* $f = O(g)$.

(e) Wenn $0 < c \leq \infty$, *dann ist* $f = \Omega(g)$.

Dieses Lemma erlaubt es also, eine asymptotische Relation durch Berechnung des entsprechenden Grenzwertes zu verifizieren. Diese Methode versagt nur dann, wenn der Grenzwert nicht existiert.

Beispiel 3.2 Wir definieren $f(n) = \begin{cases} 1 & n \text{ gerade} \\ 0 & \text{sonst.} \end{cases}$ Es sei $g(n) = 1$ für alle $n \in \mathbb{N}$. Dann existiert der Grenzwert von $\frac{f(n)}{g(n)}$ nicht, da die Werte 0 und 1 unendlich oft auftauchen. Es ist aber

$$f = O(g)$$

(mit $c = 1$ und $n_0 = 0$). Man verifiziere, dass mit Ausnahme der Beziehung $g = \Omega(f)$ alle weiteren asymptotischen Relationen zwischen f und g nicht gelten. (Warum ist zum Beispiel die Beziehung $g = \Theta(f)$ falsch?)

Beispiel 3.3 In Beispiel 3.1 haben wir mühsam $\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n = \Theta(n^3)$ nachgewiesen. Alternativ hätte es auch genügt, zu bemerken, dass

$$0 < \lim_{n\to\infty} \frac{\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n - 1}{n^3} = \frac{1}{6} < \infty$$

gilt.

Wie rechnet man mit Grenzwerten? Wir sagen, dass die Funktion $F : D \to \mathbb{R}$ **stetig** im Punkt $a \in D$ ist, wenn für alle Funktionen $f : \mathbb{N} \to D$ gilt

$$F(\lim_{n\to\infty} f(n)) = \lim_{n\to\infty} F(f(n)). \tag{3.1}$$

Es gibt viele wichtige Funktionen, die in allen Punkten ihres Definitionsbereichs steig sind. Dazu gehören

- Polynome (mit $D = \mathbb{R}$),
- rationale Funktionen, also Quotienten $\frac{p(x)}{q(x)}$ von Polynomen, wenn man die Nullstellen von q ausnimmt,

- Logarithmen $\log_a x$ für $a > 1$ (mit $D =]0, \infty[$) und Exponentialfunktionen a^x (mit $D = \mathbb{R}$),
- trigonometrische Funktionen und viele mehr.

Der Clou hinter Aussage (3.1) ist, dass der Grenzwert $\lim_{n\to\infty} F(f(n))$ für eine stetige Funktion mit $F(a)$ übereinstimmt, wenn die Folge f nur aus Elementen des Definitionsbereichs D von F besteht und wenn f gegen ein Element $a \in D$ konvergiert.

Beispiel 3.5 Die Folge $f(n) = 4 - \frac{1}{n}$ konvergiert gegen den Wert 4. Die Folge $g(n) = \sqrt{f(n)}$ konvergiert gegen den Wert 2, denn die Funktion $\sqrt{x}$ (mit Definitionsbereich $D = \mathbb{R}_{\geq 0}$) ist stetig in allen Punkten von D.

Wie verhalten sich Grenzwerte unter Addition, Subtraktion, Multiplikation und Quotientenbildung?

Lemma 3.2 *$f, g : \mathbb{N} \to \mathbf{R}_{\geq 0}$ seien Funktionen. Dann gilt*

(a) $\lim_{n\to\infty} (f(n) \circ g(n)) = \lim_{n\to\infty} f(n) \circ \lim_{n\to\infty} g(n)$*, wobei vorausgesetzt wird, dass die beiden Grenzwerte auf der rechten Seite existieren und endlich sind. Die Operation $\circ$ bezeichnet stellvertretend eine der Operationen $+, -$ oder $*$.*

(b) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\lim_{n\to\infty} f(n)}{\lim_{n\to\infty} g(n)}$ *wobei vorausgesetzt wird, dass beide Grenzwerte auf der rechten Seite existieren, endlich sind und dass* $\lim_{n\to\infty} g(n) \neq 0$ *gilt.*

Aufgabe 14
Bestimme für jedes der folgenden Paare, ob $f(n) = O(g(n))$, $f(n) = o(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$ oder $f(n) = \Theta(g(n))$ gilt.

(1) $f(n) = \log_2 n$, $g(n) = \log_{10} n$.

(2) $f(n) = \dfrac{n}{\log_{100} n}$, $g(n) = \sqrt{n}$.

(3) $f(n) = \displaystyle\sum_{i=1}^{\lfloor \log_2 n \rfloor} 3^i$, $g(n) = n^2$.

Das Integral-Kriterium ermöglicht es, endliche Summen durch Integrale abzuschätzen.

Lemma 3.3 *$f : \mathbf{R} \to \mathbf{R}$ sei gegeben.*

(a) Wenn f monoton wachsend ist, dann gilt

$$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)dx.$$

(b) Wenn f monoton fallend ist, dann gilt

$$\int_1^{n+1} f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_0^n f(x)dx.$$

Beweis: Wir verifizieren nur Teil (a). Teil (b) folgt mit analogem Argument. Da f monoton wachsend ist, gilt $\int_i^{i+1} f(x)dx \leq f(i+1)$, denn die Fläche unter der Kurve $f(x)$, zwischen i und $i+1$, ist nach oben beschränkt durch die Fläche des Rechtecks zwischen i und $i+1$ mit der Höhe $f(i+1)$; die Fläche dieses Rechtecks ist $f(i+1) \cdot 1 = f(i+1)$. Es ist also

$$\int_0^n f(x)dx = \sum_{i=0}^{n-1} \int_i^{i+1} f(x)dx \leq \sum_{i=0}^{n-1} f(i+1) = \sum_{i=1}^{n} f(i).$$

Analog erhalten wir

$$f(i) \leq \int_i^{i+1} f(x)dx,$$

denn f ist monoton wachsend und damit folgt

$$\sum_{i=1}^{n} f(i) \leq \sum_{i=1}^{n} \int_i^{i+1} f(x)dx = \int_1^{n+1} f(x)dx.$$

□

Wir erhalten als Konsequenz:

Lemma 3.4

(a) Die harmonische Reihe wächst logarithmisch: $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln(n))$.

(b) Sei k eine nicht-negative reelle Zahl. Dann gilt $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$.

Aufgabe 15
Beweise Lemma 3.4.

3.2.2 Eine Wachstums-Hierarchie

Welche Funktionen wachsen unbeschränkt? Für monoton wachsende Funktionen f, also Funktionen $f : \mathbb{N} \to \mathbb{R}$ mit $f(n) \leq f(n+1)$ für alle $n \in \mathbb{N}$ können wir eine sehr allgemeine Aussage geben.

Satz 3.5 ***(Unbeschränkt wachsende Funktionen)***
Die Funktionen $f : \mathbb{N} \to \mathbb{R}$ und $g : \mathbb{R} \to \mathbb{R}$ seien gegeben.

(a) f sei monoton wachsend. Dann gibt es eine Konstante $K \in \mathbb{N}$ mit $f(n) \leq K$ oder aber $\lim_{n\to\infty} f(n) = \infty$ gilt. Insbesondere ist $f = O(1)$ oder $1 = o(f)$.

(b) Für jedes $a > 1$ ist $1 = o(\log_a(n))$.

(c) Wenn $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, dann gilt $\lim_{n\to\infty} g(f(n)) = \infty$.

(d) Wenn $g = o(n)$ und $1 = o(f)$, dann ist $g \circ f = o(f)$. Wenn man eine „sublineare“ Funktion g auf eine unbeschränkt wachsende Funktion f anwendet, dann nimmt das Wachstum von $g \circ f$ ab.

Beweis (a): Angenommen, es gibt keine Konstante K mit $f(n) \leq K$ für alle $n \in \mathbb{N}$. Da f monoton wachsend ist, gibt es zu jeder Zahl $K \in \mathbb{N}$ eine Zahl N_K, so dass $f(n) > K$ für alle $n \geq N_K$. Aber damit folgt $\lim_{n\to\infty} f(n) = \infty$.

(b) Der Logarithmus $\log_a x$ ist monoton wachsend. Denn wenn $0 < x < y$ und $a^{\ell_x} = x$ und $a^{\ell_y} = y$, dann folgt $\ell_x < \ell_y$. Desweiteren ist $\log_a(a^m) = m$ und der Logarithmus kann durch keine Konstante beschränkt werden. Die Behauptung folgt aus Teil (a).

(c,d) sind als Übungsaufgaben gestellt. □

Aufgabe 16
Zeige Satz 3.5 (c) und (d).

Definition 3.2 $a > 1$ sei eine reelle Zahl.

(a) Der k-fach iterierte Logarithmus ist rekursiv wie folgt definiert.

- Es ist $\log_a^{(0)}(n) := n$ und
- $\log_a^{(k+1)}(n) := \log_a(\log_a^{(k)}(n))$.

(b) Die „Log-Stern“ Funktion ist definiert durch

$$\log_a^*(n) := \text{ die kleinste Zahl } k \text{ mit } \log_a^{(k)}(n) \leq 1.$$

Beispiel 3.6 (**Der k-fach iterierte Logarithmus**)

Die Zahl $A_k(m)$ sei der „Exponenten-Turm“ $a^{a^{a^{\cdots^m}}}$ von $k-1$ a's gefolgt von dem Exponenten m. Da $\log_a^{(k)} A_k(m) = m$, ist $\log_a^{(k)}(n)$ unbeschränkt wachsend.

Beachte, dass die Hintereinanderausführung monoton wachsender Funktionen monoton wachsend ist und deshalb ist $\log_a^{(k)}(n)$ für jedes k eine monoton wachsende Funktion. Wir erhalten

$$1 = o(\log_a^{(k)}(n))$$

aus Satz 3.5 (c).

Beispiel 3.7 (**Die Log-Stern Funktion**)

Die Log-Stern Funktion ist eine furchterregend langsam wachsende Funktion. Dazu definiere die Folge N_k durch $N_1 = 2$, $N_{k+1} = 2^{N_k}$ und $\log_2^*(N_k) = k$ folgt mit vollständiger Induktion, denn jede Anwendung des Logarithmus erniedrigt die „Höhe“ k des Turms der Zweierpotenzen um Eins. Die Log-Stern Funktion ist offensichtlich eine monoton wachsende Funktion, denn je größer die Argumente sind umso mehr „Log-Anwendungen“ sind notwendig, um unter Eins zu fallen. Auch kann $\log_2^a stn$ nicht durch eine Konstante beschränkt werden, da $\log_2^*(N_k) = k$ gilt. Also folgt

$$1 = o(\log_2^*(n)).$$

Betrachte zum Beispiel $N_4 := 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$. Es ist $\log_2^*(N_4) = 4$. Die Zahl $N_5 := 2^{N_4}$ ist schwindelerregend groß, wenn man bedenkt, dass die Anzahl M der Atome des Universums auf ca. $M \approx 10^{78} < 2^{312}$ geschätzt wird. Aber der Log-Stern Wert von N_5 ist nur 5: „Für natürliche Zahlen n im Hausgebrauch“ ist $\log_2^*(n) \leq 5$.

Wer wächst schneller, der natürliche Logarithmus $\ln n$ oder eine beliebig große Wurzel der lineare Funktion g mit $g(n) = n$? Wir bestimmen zuerst den Grenzwert $\lim_{n\to\infty} \frac{\ln n}{n}$ mit Hilfe der Regel von de l'Hospital.

Satz 3.6 (Satz von de l'Hospital)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)},$$

falls der letzte Grenzwert existiert und falls $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) \in \{0, \infty\}$.

Beispiel 3.8 **(Wachstum des Logarithmus)**

Es ist $\lim_{n\to\infty} \ln n = \lim_{n\to\infty} n = \infty$ und $\lim_{n\to\infty} \frac{(\ln(n))'}{n'} = \lim_{n\to\infty} \frac{1/n}{1} = 0$. Als Konsequenz folgt deshalb $\lim_{n\to\infty} \frac{\ln n}{n} = 0$: Der natürliche Logarithmus $\ln(n)$ wächst sehr viel langsamer als n. Aus Gleichung (2.5) folgt

$$\log_a(n) = o(n),$$

also, dass der Logarithmus $\log_a(n)$ für jedes $a > 1$ sehr viel langsamer als n wächst.

Sei $b > 0$ eine positive reelle Zahl. Der Logarithmus wächst sogar selbst dann schwächer, wenn wir beliebig kleine Potenzen n^b der linearen Funktion $f(n) = n$ bilden, denn $\lim_{n\to\infty} \ln n = \lim_{n\to\infty} n^b = \infty$ und $\lim_{n\to\infty} \frac{(\ln(n))'}{(n^b)'} = \lim_{n\to\infty} \frac{1/n}{b\cdot n^{b-1}} = \lim_{n\to\infty} \frac{1}{b\cdot n^b} = 0$. Also ist

$$\log_a(n) = o(n^b). \tag{3.2}$$

Folgerung 3.7 *Für alle natürlichen Zahlen k und alle reellen Zahlen $a > 1$ und $b > 0$ gilt*

(a) $1 = o(\log_a^*(n))$ *und* $\log_a^*(n) = o(\log_a^{(k)}(n))$.

(b) $\log_a^{(k+1)}(n) = o(\log_a^{(k)}(n))$.

(c) $\log_a(n) = o(n^b)$.

Beweis (a): Die asymptotische Beziehung $1 = o(\log_a^*(n))$ haben wir in Beispiel 3.7 festgestellt, die Beziehung $\log_a^*(n) = o(\log_a^{(k)}(n))$ ist als Übungsaufgabe gestellt.

(b) Wir wissen aus Beispiel 3.6, dass $1 = o(\log_a^{(k)}(n))$ für jede natürliche Zahl k gilt und haben in Beispiel 3.8 beobachtet, dass $\log_a(n) = o(n)$ gilt. Wir erhalten also die Behauptung $\log_a^{(k+1)}(n) = o(\log_a^{(k)}(n))$ aus Satz 3.5 (d).

(c) Die Behauptung $\log_a(n) = o(n^b)$ haben wir in Beispiel 3.8 gezeigt. □

Folgerung 3.7 beschreibt eine Wachstum-Hierarchie sehr langsam wachsender Funktionen. Jetzt betrachten wir schneller wachsende Funktionen.

Satz 3.8 *Es sei $a, k > 1$ und $0 < b \leq 1$ für reelle Zahlen a, b, k. Wir definieren*

$$f_1(n) = (\log_a n)^k, \quad f_2(n) = n^b, \quad f_3(n) = n \log_a n, \quad f_4(n) = n^k, \quad f_5(n) = a^n, \quad f_6(n) = n!$$

Dann gilt $f_i = o(f_{i+1})$ für $1 \leq i \leq 5$.

Beweis: Um $f_1 = o(f_2)$ zu zeigen, ziehen wir die $k$te Wurzel in f_1:

$$\lim_{n\to\infty} \frac{f_1(n)}{f_2(n)} = \lim_{n\to\infty} \frac{(\log_a n)^k}{n^b} = \lim_{n\to\infty} \left(\frac{\log_a n}{n^{b/k}}\right)^k \overset{(3.1)}{=} \left(\lim_{n\to\infty} \frac{\log_a n}{n^{b/k}}\right)^k \overset{(3.2)}{=} 0.$$

Für einen Vergleich von f_2 und f_3 beachte, dass

$$\lim_{n\to\infty} \frac{f_2(n)}{f_3(n)} = \lim_{n\to\infty} \frac{n^b}{n \log_a n} \leq \lim_{n\to\infty} \frac{1}{\log_a n} = 0,$$

denn $1 = o(\log_a n)$. Und somit folgt $f_2 = o(f_3)$.

f_3 wächst langsamer als f_4, denn es ist $k > 1$ und

$$\lim_{n\to\infty} \frac{f_3(n)}{f_4(n)} = \lim_{n\to\infty} \frac{n \log_a n}{n^k} = \lim_{n\to\infty} \frac{\log_a n}{n^{k-1}} =^{(3.2)} 0.$$

Als nächstes betrachten wir f_4 und f_5.

$$\lim_{n\to\infty} \frac{f_4(n)}{f_5(n)} = \lim_{n\to\infty} \frac{n^k}{b^n} = \lim_{n\to\infty} \frac{(b^{\log_b n})^k}{b^n} = \lim_{n\to\infty} b^{(k \log_b n - n)}$$

Der Exponent $k \log_b n - n$ strebt aber gegen $-\infty$, denn n wächst schneller als $\log_b n$ wie wir in (3.2) gesehen haben. Also folgt

$$\lim_{n\to\infty} \frac{f_4(n)}{f_5(n)} = 0$$

aus $b > 1$. Nun zum letzten Vergleich:

$$\begin{aligned} 0 &\leq \lim_{n\to\infty} \frac{f_5(n)}{f_6(n)} = \lim_{n\to\infty} \frac{b^n}{n!} \\ &\leq \lim_{n\to\infty} \frac{b^n}{(\frac{n}{2})^{n/2}} \qquad \text{denn } n! \geq n \cdot (n-1) \cdots (n/2+1) \geq (n/2)^{n/2} \\ &= \lim_{n\to\infty} \left(\frac{b^2}{n/2} \right)^{n/2} = 0, \end{aligned}$$

denn $\lim_{n\to\infty} \frac{b^2}{n/2} = 0$. Mit anderen Worten $\lim_{n\to\infty} \frac{f_5(n)}{f_6(n)} = 0$. □

Bemerkung 3.1 **(Wachstumsklassen)**

Wir fassen zusammen: Die konstante Funktion wächst am langsamsten, gefolgt von der Log-Stern Funktion, die ihrerseits langsamer wächst als Hintereinanderanwendungen des Logarithmus.

Selbst für sehr, sehr kleine Potenzen $0 < b < 1$ wächst der Logarithmus viel, viel langsamer als n^b. Dann kommen die recht häufig auftauchenden Laufzeitfunktionen $f(n) = n$ und $g(n) = n \cdot \log_2 n$. Beachte, dass jedes Programm, dass sich alle n Eingabedaten anschauen muss, mindestens $\Omega(n)$ Schritte investieren muss. Die Laufzeitfunktion $n \cdot \log_2 n$ taucht für schnelle Divide & Conquer Algorithmen auf.

Mit den Funktionen n^k für $k > 1$ beginnt es teuer zu werden. Schon quadratische Laufzeiten tun weh und erlauben keine routinemäßigen Anwendungen des zugrunde liegenden Programms für große Eingabelängen. Die Laufzeiten $h(n) = a^n$ sind unbezahlbar und für $h'(n) = n!$ mehr als unverschämt: Die Funktionen a^n und $n!$ sind Stars in jedem Gruselkabinett und selbst kubische Laufzeiten lassen einen kalten Schauer über den Rücken laufen.

Hier ist die Hierarchie (für $0 < b < 1, a, k > 1$ und die natürliche Zahl $\ell \geq 1$) auf einen Blick:

$$1 << \log_a^*(n) << \log_a^{(\ell+1)}(n) << \log_a^{(\ell)}(n) << n^b << n << n \cdot \log_a n << n^k << a^n << n!$$

Im Entwurf von Datenstrukturen sind wir zum Beispiel an Suchanfragen interessiert wie etwa im einfachsten Fall an der Frage

„Ist das Datum x gespeichert?“

Da diese Suchanfragen sehr häufig auftreten, sind wir nur bereit, unabhängig von der Anzahl gespeicherter Daten, eine konstante Reaktionszeit $r(n)$ zu tolerieren oder allenfalls eine logarithmische Reaktionszeit. Es sollte also $r(n) = O(1)$, bzw. $r(n) = O(\log_2 n)$ gelten.

Aufgabe 17
Ordne die folgenden Funktionen gemäß ihrem asymptotischen „Wachstum“ beginnend mit der „langsamsten“ Funktion.

$f_1(n) = n!$	$f_2(n) = 2^{\log_4 n}$	$f_3(n) = 4^{\log_2 n}$	$f_4(n) = n^{1/\log_2 n}$
$f_5(n) = 2^{-n}$	$f_6(n) = \log_2(n!)$	$f_7(n) = n \log_2 n$	$f_8(n) = (1.1)^n$
$f_9(n) = n^3$	$f_{10}(n) = n^{\log_2 n}$	$f_{11}(n) = n^{3/2}$	$f_{12}(n) = n^n$

Aufgabe 18
Ordne die folgenden Funktionen nach ihrem asymptotischen Wachstum, beginnend mit der langsamsten Funktion. Beweise deine Aussagen. Die Aussagen sollen jeweils so scharf wie möglich sein. Gilt also $f(n) = o(g(n))$ oder $f(n) = \Theta(g(n))$, so soll nicht nur $f(n) = O(g(n))$ gezeigt werden. Es ist also möglich, dass Funktionen gleiches asymptotisches Wachstum zeigen.

$$f_1(n) = 4^n$$

$$f_2(n) = 3^{(3^n)}$$

$$f_3(n) = 2n + 3$$

$$f_4(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 5 \cdot f_4(\frac{n}{5}) + 7n - 1 & \text{sonst} \end{cases}$$

$$f_5(n) = 4^{\log_2 n}$$

$$f_6(n) = (3^3)^n$$

$$f_7(n) = \sum_{k=0}^{n} (\tfrac{1}{2})^k$$

$$f_8(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 9 \cdot f_8(\frac{n}{3}) + n \cdot \log_2(n) & \text{sonst} \end{cases}$$

Aufgabe 19

	$\log n$	$s(n)$	$\sqrt{n}$	5	2^n	$1/n$	n	e^n	n^2
$\log n$									
$s(n)$									
$\sqrt{n}$									
5									
2^n									
$1/n$									
n									
e^n									
n^2									

In der Tabelle stehe $s(n)$ abkürzend für die Funktion $s(n) = n \cdot (1 + (-1)^n) + 1$

Trage in die obige Tabelle die Wachstumsverhältnisse ein und verwende dabei die Symbole O, o, ω, Ω, Θ. Versuche, so genau wie möglich zu sein. Beispiel: $f(n) := n^3; g(n) := n^4$ ergibt $f = o(g)$ und nicht nur $f = O(g)$.

3.3 Ein Beispiel zur Laufzeitbestimmung

Wenn wir die Laufzeit eines Programms messen wollen, müssen wir die folgenden Fragen beantworten:

1. Für welchen Rechner berechnen wir die Laufzeit?
2. Mit welcher Programmiersprache und mit welchem Compiler arbeiten wir?

Es ist sicherlich unbefriedigend, eine Laufzeitanalyse für einen bestimmten Rechner, eine bestimmte Programmiersprache und einen bestimmten Compiler durchzuführen, da die Analyse für jede neue Konfiguration zu wiederholen wäre! Wir sind deshalb bereits mit einer Schätzung der tatsächlichen Laufzeit zufrieden, die

bis auf einen konstanten Faktor

für jede Konfiguration exakt ist. Gelingt uns dies, so lässt sich weiterhin das asymptotische Wachstum der Laufzeit verläßlich bestimmen und das Programm kann als entweder „möglicherweise praxistauglich“ oder aber „für die Tonne“ klassifiziert werden.

Und noch ein weiterer, sehr wichtiger Punkt, der für eine „bis auf konstante Faktoren exakte“ Laufzeitbestimmung spricht: Ein Programm implementiert das algorithmische „Rezept“ oder die algorithmische Idee hinter dem Programm. Dieses Rezept wird **Algorithmus** genannt und muss so detailliert sein, dass das Programm „nur noch“ die Übersetzung des Rezepts in eine Programmiersprache ist. Wenn wir schon vom Algorithmus eine bis auf konstante Faktoren exakte Laufzeit des Programms ablesen könnten, würden wir uns die teure Entwicklungsarbeit für das Programm als Implmentierung des Algorithmus sparen.

Wie man in der Laufzeitbestimmung vorgeht, besprechen wir am Beispiel des **Teilfolgenproblems**. Die *Eingabe* besteht aus n ganzen Zahlen $a_1, \ldots, a_n$. Es sei, für $1 \leq i \leq j \leq n$,

$$f(i,j) = a_i + a_{i+1} + \cdots + a_j.$$

Das *Ziel* ist die Berechnung von

$$\max\{f(i,j) \,:\, 1 \leq i \leq j \leq n\},$$

also den maximalen f-Wert für Teilfolgen $(i, \ldots, j)$. Die Eingabelänge der Eingabe $(a_1, \ldots, a_n)$ definieren als n.

Wir möchten Algorithmen betrachten, deren Laufzeit durch Additionen und Vergleichsoperationen auf den Daten dominiert wird. Für einen vorgelegten Algorithmus A betrachten wir deshalb die Effizienzmaße

$$\text{Additionen}_A(n) = \max_{a_1,\ldots a_n \in \mathbb{Z}} \{ \text{ Anzahl der Additionen von } A \text{ für Eingabe } (a_1, \ldots, a_n)\}$$

sowie

$$\text{Vergleiche}_A(n) = \max_{a_1,\ldots,a_n \in \mathbb{Z}} \{ \text{ Anzahl der Vergleiche von } A \text{ für Eingabe } (a_1, \ldots, a_n)\}.$$

$\text{Zeit}_A(n) = \text{Additionen}_A(n) + \text{Vergleiche}_A(n)$ ist somit die **worst-case Rechenzeit** von Algorithmus A, wenn nur Additionen und Vergleiche, und diese gleichberechtigt gezählt werden.

Ein erster Algorithmus $\mathbf{A_1}$:

(1) Max $= -\infty$;
(2) for $(i = 1\ ;\ i <= n;\ i{+}{+})$
for $(j = i;\ j <= n;\ j{+}{+})$
{Berechne $f(i,j)$ mit $j - i$ Additionen;
Max $= \max\{f(i,j), \text{Max}\};$ }

A_1 berechnet nacheinander die Werte $f(i,j)$ für alle Paare (i,j) mit $i \leq j$ unabhängig voneinander. Wir bestimmen zuerst $\text{Additionen}_{A_1}(n)$. Wir benötigen $j-i$ Additionen zur Berechnung von $f(i,j)$ und erhalten deshalb

$$\textit{Additionen}_{A_1}(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i).$$

Wir wenden Formeln aus Lemma 2.9 an:

$$\begin{aligned}
\textit{Additionen}_{A_1}(n) &= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) \\
&= \sum_{i=1}^{n}\sum_{j=0}^{n-i} j \qquad \text{mit Indexverschiebung} \\
&\overset{\text{Lemma 2.9}(a)}{=} \sum_{i=1}^{n}\frac{(n-i)(n-i+1)}{2} \\
&= \frac{1}{2}\sum_{i=1}^{n}\left(n^2+n-2in-i+i^2\right) \\
&= \frac{1}{2}\left(\sum_{i=1}^{n}(n^2+n) - \sum_{i=1}^{n}(2in+i) + \sum_{i=0}^{n} i^2\right) \\
&\overset{\text{Lemma 2.9}(b)}{=} \frac{1}{2}\left(n\cdot(n^2+n) - (2n+1)\cdot\sum_{i=1}^{n} i + \frac{n\cdot(n+1)\cdot(2n+1)}{6}\right) \\
&= \frac{1}{2}\left(n^3+n^2 - \frac{(2n+1)\cdot n\cdot(n+1)}{2} + \frac{n\cdot(n+1)\cdot(2n+1)}{6}\right) \\
&= \frac{1}{2}\left(n^3+n^2 - \frac{n\cdot(n+1)\cdot(2n+1)}{3}\right) \\
&= \frac{3n^3+3n^2-2n^3-3n^2-n}{6} \\
&= \frac{n^3-n}{6}.
\end{aligned}$$

Algorithmus A_1 führt $\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}(n-i+1)$ Vergleiche durch. Also folgt

$$\text{Vergleiche}_{A_1}(n) = n + (n-1) + \cdots + 2 + 1 = \frac{n\cdot(n+1)}{2}$$

Vergleiche. Mit anderen Worten

$$\text{Zeit}_{A_1}(n) = \frac{1}{6}n^3 - \frac{1}{6}n + \frac{1}{2}n^2 + \frac{1}{2}n = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n$$

Wir möchten das wesentliche Ergebnis festhalten, nämlich, dass die Laufzeit kubisch ist. Genau dies gelingt mit Hilfe der asymptotischen Notation, denn

$$\text{Zeit}_{A_1}(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n = \Theta(n^3).$$

Wir modifizieren jetzt A_1, indem wir die $f(i,j)$-Werte sorgfältiger berechnen.

Algorithmus $\mathbf{A_2}$:

(1) Max $= -\infty$;
(2) for $(i = 1\ ;\ i <= n\ ;\ i{+}{+})$

$\{\ f(i, i-1) = 0;$
for $(j = i\ ;\ j <= n;\ j{+}{+})$
$\{\ f(i,j) = f(i,j-1) + a_j;$
$\text{Max} = \max\{f(i,j), \text{Max}\};\ \}$

Algorithmus A_2 berechnet also $f(i, j+1)$ mit nur einer Addition aus $f(i,j)$. Wieviel Additionen führt A_2 durch? Genau so viele Additionen wie Vergleiche und es ist deshalb

$$\text{Additionen}_{A_2}(n) = \text{Vergleiche}_{A_2}(n) = \frac{n \cdot (n+1)}{2}.$$

Wir erhalten deshalb

$$\text{Zeit}_{A_2}(n) = n \cdot (n+1).$$

In asymptotischer Terminologie: Die Laufzeit von A_2 ist tatsächlich quadratisch, da

$$n \cdot (n+1) = \Theta(n^2)$$

gilt. Zusätzlich ist A_2 schneller als A_1, denn $n^2 = o(n^3)$ und damit folgt

$$\text{Zeit}_{A_2} = o(\text{Zeit}_{A_1}).$$

Aber wir können mit Divide und Conquer fast-lineare Laufzeit erhalten! **Algorithmus $\mathbf{A_3}$** baut auf der folgenden Idee auf: Angenommen wir kennen den maximalen $f(i,j)-$Wert für $1 \le i \le j \le \lceil \frac{n}{2} \rceil$ sowie den maximalen $f(i,j)$-Wert für $\lceil \frac{n}{2} \rceil + 1 \le i \le j \le n$. Welche Möglichkeiten gibt es für den maximalen $f(i,j)$-Wert für $1 \le i \le j \le n$?

Möglichkeit 1: $\max\ \{f(i,j)\ :\ 1 \le i \le j \le n\} = \max\{f(i,j)\ :\ 1 \le i \le j \le \lceil \frac{n}{2} \rceil\}$

Möglichkeit 2: $\max\ \{f(i,j)\ :\ 1 \le i \le j \le n\} = \max\{f(i,j)\ :\ \lceil \frac{n}{2} \rceil + 1 \le i \le j \le n\}$

Möglichkeit 3: $\max\ \{f(i,j)\ :\ 1 \le i \le j \le n\} = \max\{f(i,j)\ :\ 1 \le i \le \lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil + 1 \le j \le n\}$.

Was ist im Fall der dritten Möglichkeit zu tun? Wir können i und j unabhängig voneinander bestimmen, nämlich wir bestimmen i, so dass

$$f(i, \lceil \frac{n}{2} \rceil) = \max\{f(k, \lceil \frac{n}{2} \rceil)\ :\ 1 \le k \le \lceil \frac{n}{2} \rceil\}$$

und bestimmen j, so dass

$$f(\lceil \frac{n}{2} \rceil + 1, j) = \max\{f(\lceil \frac{n}{2} \rceil + 1, k)\ :\ \lceil \frac{n}{2} \rceil + 1 \le k \le n\}.$$

Wir können somit i und j (für die dritte Möglichkeit) mit $\lceil \frac{n}{2} \rceil - 1 + (n - (\lceil \frac{n}{2} \rceil + 1)) = n-2$ Additionen und derselben Anzahl von Vergleichen berechnen. Eine zusätzliche Addition erlaubt uns, den optimalen Wert $f(i, \lceil \frac{n}{2} \rceil) + f(\lceil \frac{n}{2} \rceil + 1, j)$ zu bestimmen.

Mit zwei zusätzlichen Vergleichen bestimmen wir den optimalen $f(i,j)$-Wert unter den drei Möglichkeiten.

Fazit: Wir nehmen an, dass n eine Potenz von zwei ist. Dann benötigt Algorithmus A_3 mit Eingabe $(a_1, \ldots, a_n)$ zwei Aufrufe für Folgen von je $n/2$ Zahlen (nämlich einen Aufruf für Eingabe $(a_1, \ldots, a_{n/2})$ und einen Aufruf für $(a_{n/2+1}, \ldots, a_n)$). $\frac{n}{2} - 1 + \frac{n}{2} - 1 + 1 = n - 1$ Additionen und $\frac{n}{2} - 1 + \frac{n}{2} - 1 + 2 = n$ Vergleiche sind ausreichend, wobei wir die Additionen und Vergleiche in den beiden rekursiven Aufrufen **nicht** mitzählen. Damit werden wir also auf eine Rekursionsgleichung für Zeit_{A_3} geführt:

- $\text{Zeit}_{A_3}(1) = 1$,
- $\text{Zeit}_{A_3}(n) = 2Zeit_{A_3}\left(\frac{n}{2}\right) + 2n - 1$.

Rekursionen dieses Typs lassen sich leicht mit dem Mastertheorem asymptotisch lösen.

3.4 Das Mastertheorem

Satz 3.9 *Das Mastertheorem.*

Die Rekursion

- $T(1) = c$,
- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

ist zu lösen, wobei vorausgesetzt sei, dass n eine Potenz der natürlichen Zahl $b > 1$ ist. Dann gilt für $a \geq 1$, $c > 0$ und $f : \mathbb{N} \to \mathbb{R}$:

- *Wenn $f(n) = O\left(n^{(\log_b a)-\varepsilon}\right)$ für eine positive Konstante $\varepsilon > 0$, dann ist $T(n) = \Theta(n^{\log_b a})$.*
- *Wenn $f(n) = \Theta(n^{\log_b a})$, dann ist $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.*
- *Wenn $f(n) = \Omega\left(n^{(\log_b a)+\varepsilon}\right)$ für eine positive Konstante $\varepsilon > 0$ und $a \cdot f\left(\frac{n}{b}\right) \leq \alpha f(n)$ für eine Konstante $\alpha < 1$, dann ist $T(n) = \Theta(f(n))$.*

Bestimmen wir zuerst die asymptotische Lösung für die Rekursionsgleichung von Zeit_{A_3}. Es ist $c = 1, a = 2, b = 2$ und $f(n) = 2n - 1$. Wir erhalten

$$\log_b a = \log_2 2 = 1 \quad \text{und} \quad f(n) = \Theta(n^{\log_b a}).$$

Damit ist der zweite Fall von Satz 3.9 anzuwenden und es folgt, dass

$$\text{Zeit}_{A_3} = \Theta(n \log_2 n).$$

Algorithmus A_3 ist schneller als Algorithmus A_2, denn

$$\lim_{n\to\infty} \frac{n \log_2 n}{n^2} = \lim_{n\to\infty} \frac{\log_2 n}{n} \overset{(3.2)}{=} 0.$$

Beweis von Satz 3.9. Wir entwickeln die Rekursionsgleichung und erhalten

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= a\left(aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) \\ &= a^2T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \\ &= a^3T\left(\frac{n}{b^3}\right) + a^2f\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n). \end{aligned}$$

Dies legt die Vermutung

$$(*) \qquad T(n) = a^k T\left(\frac{n}{b^k}\right) + a^{k-1} f\left(\frac{n}{b^{k-1}}\right) + \cdots + af\left(\frac{n}{b}\right) + f(n) = a^k T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)$$

nahe. Diese Vermutung ist richtig, wie man leicht durch einen Beweis mit vollständiger Induktion sieht:

Verankerung : $k = 0$

$$T(n) = 1 \cdot T\left(\frac{n}{1}\right) + 0 = a^0 T\left(\frac{n}{b^0}\right) + \sum_{i=0}^{-1} a^i f\left(\frac{n}{b^i}\right)$$

und das war zu zeigen. Im Induktionsschritt können wir annehmen, dass $T(n) = a^k \cdot T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)$ bereits gezeigt ist. Wir führen einen weiteren Expansionsschritt aus und erhalten

$$\begin{aligned} T(n) &= a^k \cdot \left[aT\left(\frac{\frac{n}{b^k}}{b}\right) + f\left(\frac{n}{b^k}\right)\right] + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right) \\ &= a^{k+1} \cdot T\left(\frac{n}{b^{k+1}}\right) + \sum_{i=0}^{k} a^i f\left(\frac{n}{b^i}\right). \end{aligned}$$

Damit ist der Induktionsschritt vollzogen und $(*)$ ist gezeigt. □

Um die Rekursion asymptotisch zu lösen, wählen wir k in $(*)$ so, dass

$$\frac{n}{b^k} = 1.$$

Mit anderen Worten, wir müssen $k = \log_b n$ setzen und erhalten mit $(*)$:

$$\begin{aligned} T(n) &= a^{\log_b n} T(1) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \\ &= a^{\log_b n} \cdot c + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right), \end{aligned}$$

denn $T(1) = c$. Der erste Summand in der Darstellung von $T(n)$ ist aber $n^{\log_b a} \cdot c$ in Verkleidung (vgl. Lemma 2.13), da

$$\begin{aligned} a^{\log_b n} &= a^{(\log_b a \cdot \log_a n)}. \\ &= \left(a^{\log_a n}\right)^{\log_b a} \\ &= n^{\log_b a} \end{aligned}$$

und somit folgt

$$(**) \qquad T(n) = n^{\log_b a} \cdot c + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right).$$

Fall 1: $f(n) = O(n^{(\log_b a)-\varepsilon})$. Es ist also $f(n) \leq d \cdot n^{\log_b a-\varepsilon}$ für eine positive Konstante d. Es folgt

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) &\leq d \cdot \sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon} \\
&= d \sum_{j=0}^{\log_b n-1} \frac{a^j}{a^j} \cdot n^{(\log_b a)-\varepsilon} \cdot b^{\varepsilon j}, \quad \text{denn } b^{j \log_b a} = a^j \\
&= d \cdot n^{(\log_b a)-\varepsilon} \sum_{j=0}^{\log_b n-1} b^{\varepsilon j} \\
&= d \cdot n^{(\log_b a)-\varepsilon} \cdot \frac{b^{\varepsilon \cdot \log_b n}-1}{b^\varepsilon - 1} \quad \text{(mit Lemma 2.9)} \\
&= d \cdot n^{(\log_b a)-\varepsilon} \cdot \frac{n^\varepsilon - 1}{(b^\varepsilon - 1)} \\
&\leq d \cdot n^{(\log_b a)-\varepsilon} \cdot \frac{n^\varepsilon}{(b^\varepsilon - 1)} = \frac{d}{b^\varepsilon - 1} \cdot n^{\log_b a}.
\end{aligned}$$

Unser Ergebnis ist somit

$$\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) = O\left(n^{\log_b a}\right)$$

und mit $(**)$ folgt

$$T(n) = n^{\log_b a} \cdot c + O\left(n^{\log_b a}\right).$$

Als Konsequenz: $T(n)$ ist die Summe von 2 Werten, wobei der erste $c \cdot n^{\log_b a}$ ist und der zweite durch $O\left(n^{\log_b a}\right)$ beschränkt ist. Es folgt $T(n) = \Theta\left(n^{\log_b a}\right)$.

Fall 2: $f(n) = \Theta\left(n^{\log_b a}\right)$. Jetzt erhalten wir

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \cdot f\left(\frac{n}{b^j}\right) &= \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b a}\right) \\
&= \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \cdot \frac{n^{\log_b a}}{a^j}\right) \\
&= \Theta\left(\log_b n \cdot n^{\log_b a}\right).
\end{aligned}$$

In der Darstellung

$$T(n) = n^{\log_b a} \cdot c + \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right)$$

dominiert somit der zweite Summand und es folgt

$$T(n) = \Theta\left(\log_b n \cdot n^{\log_b a}\right).$$

Fall 3: $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ und es gibt $\alpha < 1$ mit $af\left(\frac{n}{b}\right) \leq \alpha f(n)$.

Wir betrachten wiederum die Summe der f-Terme in der Darstellung von T, wobei wir benutzen, dass

$$a^k f\left(\frac{n}{b^k}\right) \leq \alpha a^{k-1} f\left(\frac{n}{b^{k-1}}\right) \leq \alpha^2 a^{k-2} f\left(\frac{n}{b^{k-2}}\right) \leq \cdots \leq \alpha^k f(n).$$

Es folgt

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \cdot f\left(\frac{n}{b^j}\right) &\leq \sum_{j=0}^{\log_b n-1} \alpha^j \cdot f(n) \\ &\leq f(n) \cdot \sum_{j=0}^{\infty} \alpha^j \\ &= f(n) \cdot \frac{1}{1-\alpha} \quad \text{(mit Lemma 2.9).}\end{aligned}$$

Es ist somit

$$f(n) \leq T(n) \leq c \cdot n^{\log_b a} + f(n) \cdot \frac{1}{1-\alpha}.$$

Aber $f(n)$ wächst stärker als $n^{\log_b a}$ nach Fallannahme und wir erhalten

$$f(n) \leq T(n) = O(f(n))$$

und deshalb $T(n) = \Theta(f(n))$. Damit ist der Beweis von Satz 3.9 erbracht. ■

Aufgabe 20
Löse die folgenden Rekursionsgleichungen einmal mit Satz 1.12 und zum anderen genau (d.h. ohne O-Notation):

(a) $T(1) = 0$, $T(n) = 2T(\frac{n}{2}) + \frac{n}{2}$

(b) $T(2) = 1$, $T(n) = T(\frac{n}{2}) + n - 1$

(c) $T(1) = 1$, $T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$, wobei $n = 2^k$ gilt

---

Aufgabe 21
Die Funktion

```
int euclid (int a, int b) {
    if(b==0)
      return a;
    else
    {
      a = a % b;
      return euclid (b,a);
    }
}
```

implementiert Euklids Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen a und b. Für $n = a + b$ sei $T(n)$ die worst-case Laufzeit von euclid(a, b) auf zwei natürlichen Zahlen a, b mit $0 \leq b \leq a$.

Stelle eine Rekursionsgleichung für $T(n)$ auf und bestimme $T(n)$ in O-Notation. Die obere Schranke soll natürlich so gut wie möglich sein. (HINWEIS: Zeige, daß $b + (a\%b) \leq \frac{2}{3} \cdot (a + b)$, falls $0 \leq b \leq a$.)

Das Mastertheorem erlaubt die Laufzeit-Bestimmung vieler rekursiver Algorithmen, über seine Grenzen gibt das folgende Beispiel Aufschluß.

Beispiel 3.4 Wir betrachten die Rekursion

- $T(1) = 1$
- $T(n) = T(n-1) + n$.

Die „Eingabelänge“ n wird also nur additiv um eins verkleinert. Unser Satz 3.9 verlangt jedoch, dass die Problemgröße um einen echten Bruchteil $b > 1$ verkleinert wird. Wir erreichen aber wiederum eine Lösung durch sukzessives Einsetzen der Rekursionsgleichung. Wie im Beweis von Satz 3.9 führen wir die folgen Schritte durch

a) Wiederholtes Einsetzen, um eine Vermutung über die allgemeine Form der entwickelten Rekursion zu erhalten.

b) Beweis der Vermutung.

c) Eliminierung der T-Terme und

d) asymptotische Bestimmung der Lösung.

Für a) beachte

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

und die Vermutung

$$T(n) = T(n-k)+(n-(k-1))+\cdots+n$$

liegt nahe. Die Vermutung lässt sich jetzt einfach induktiv beweisen und damit ist Schritt b) durchgeführt. Für Schritt c) wählen wir $k = n-1$ und erhalten

$$T(n) = T(n-(n-1))+(n-(n-2))+\cdots+n = T(1)+2+3+\cdots n = \sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}.$$

Also erhalten wir

$$T(n) = \Theta(n^2)$$

in Schritt d).

Aufgabe 22
Löse die folgenden Rekursionsgleichungen. (Um die Berechnung zu vereinfachen, ignorieren wir den Einfluß von Rundungsoperationen. Wir stellen uns die Funktionen T also als Funktionen reeller Zahlen vor und wenden den Basisfall an, sobald das Argument kleiner-gleich der genannten Schranke ist.)

(a) $T(n) = \sqrt{n}\cdot T(\sqrt{n})+n$, $T(2) = 1$,
(b) $T(n) = 2\sqrt{n}\cdot T(\sqrt{n})+n$, $T(2) = 1$,
(c) $T(n) = 2T(n/2)+n\log_2 n$, $T(1) = 1$.

Hinweis: Entwickle, wie im Beweis vom Mastertheorem, die Rekursionsgleichung, bilde eine Vermutung, beweise die Vermutung und führe auf dem Basisfall zurück. (Achtung, das Mastertheorem ist hier nicht mehr anwendbar.)

Aufgabe 23
Wir haben die allgemeine Rekursionsgleichung

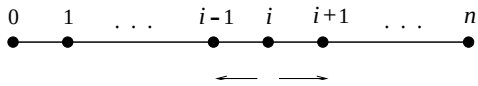
$$T(1) = c,\ T(n) = aT(n/b)+f(n)$$

(für Konstanten a, b, c mit $a \geq 1, b > 1, c > 0$) gelöst, falls n eine Potenz von b ist. Wir betrachten jetzt konkrete Wahlen von $f(n)$ für $b = 2$. Für die jeweilige Wahl von $f(n)$ bestimme $T(n)$ in Θ-Notation, *abhängig* von a. Für jeden Fall soll $T(n)$ asymptotisch exakt bestimmt sein.

(a) $f(n) = n$.
(b) $f(n) = 1$.

(c) $f(n) = n \log n$. (Achtung, die allgemeine Formel der Vorlesung ist hier nicht immer anwendbar.)

Aufgabe 24
In einem Teich befinden sich $n+1$ Steine $0, 1, \ldots, n$ in einer Reihe. Ein Frosch sitzt anfänglich auf Stein n. Befindet er sich zu irgendeinem Zeitpunkt auf einem Stein, der einen linken und einen rechten Nachbarstein hat, so springt er jeweils mit Wahrscheinlichkeit $1/2$ auf einen der beiden. Erreicht er eines der Enden der Reihe, so springt er im nächsten Sprung mit Sicherheit auf den einzigen Nachbarn. Nach wievielen Sprüngen wird der Frosch erwartungsgemäß zum ersten Mal Stein 0 erreichen?



Sei $E_n(i)$ die erwartete Zahl der Sprünge von Stein i aus bis zum erstmaligen Erreichen von Stein 0. Wir können die Erwartungswerte kanonisch in Abhängigkeit voneinander formulieren.

$$\begin{aligned} E_n(0) &= 0, \\ E_n(i) &= 1 + \frac{1}{2} \cdot E_n(i-1) + \frac{1}{2} \cdot E_n(i+1) \text{ für } 1 \leq i \leq n-1, \\ E_n(n) &= 1 + E_n(n-1). \end{aligned}$$

Berechne $E_n(n)$.

Wir kommen jetzt mit **Algorithmus $\mathbf{A_4}$** zu unserer vierten und letzten Lösung des Teilfolgenproblems. A_4 basiert auf der folgenden Idee: Wir nehmen an, wir kennen

$$Max_k^* = \max\{f(i,k) \,:\, 1 \leq i \leq k\},$$

also den optimalen Wert einer Teilfolge, *die mit dem Element a_k endet*. Wie berechnen wir dann Max_{k+1}^*?

$$Max_{k+1}^* = \max\{Max_k^* + a_{k+1}, a_{k+1}\}.$$

Es sei außerdem

$$Max_k = \max\{f(i,j) \,:\, 1 \leq i \leq j \leq k\}.$$

Max_k beschreibt also die optimale Teilfolge für die ersten k Folgenelemente. Wie berechnen wir Max_{k+1}?

$$Max_{k+1} = \max\{Max_k, Max_{k+1}^*\}.$$

Warum ist das richtig? Für die optimale Teilfolge zu den ersten $k+1$ Folgenelementen trifft genau eine der beiden folgenden Möglichkeiten zu. **Erstens**: Die optimale Teilfolge enthält das Element a_{k+1} **nicht**, dann ist sie auch optimale Lösung für die ersten k Folgenelemente, also ist in diesem Fall $Max_{k+1} = Max_k$. **Zweitens**: Die Teilfolge enthält das Element a_{k+1}, endet also mit diesem Element. Dann ist aber zwangsläufig $Max_{k+1} = Max_{k+1}^*$.

In **Algorithmus $\mathbf{A_4}$** führen wir die folgenden Schritte durch:

(1) Wir initialisieren $Max_1 = Max_1^* = a_1$.

(2) Für $k = 1, \ldots, n-1$ setze

$$Max_{k+1}^* = \max\{Max_k^* + a_{k+1}, a_{k+1}\} \text{ und}$$
$$Max_{k+1} = \max\{Max_k, Max_{k+1}^*\}.$$

(3) Max_n wird ausgegeben.

Algorithmus A_4 führt in jeder Schleife zwei Vergleiche und eine Addition durch. Insgesamt werden also $2(n-1)$ Vergleiche und $n-1$ Additionen benutzt. Es ist also

$$\text{Zeit}_{A_4}(n) = 3(n-1).$$

A_4 hat lineare Laufzeit, da $\text{Zeit}_{A_4} = \Theta(n)$. Weiterhin ist A_4 schneller als A_3, denn

$$\lim_{n\to\infty} \frac{n}{n \log_2 n} = \lim_{n\to\infty} \frac{1}{\log_2 n} = 0$$

weil der binäre Logarithmus $\log_2 n$ nach Satz 3.5 (b) unbeschränkt wächst.

Aufgabe 25
Wir erhalten ein sortiertes Array A, dessen Felder von 0 bis $n-1$ indiziert sind. Die Einträge sind ganze, nicht notwendigerweise positive Zahlen. Ein Beispiel:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$A[i]$	-9	-6	-4	-2	-1	0	2	3	5	8	9	12	14	16	19	21	22

Wir möchten wissen, ob es ein Element i mit $i = A[i]$ gibt. Falls ja, soll das Element auch ausgegeben werden. Im obigen Beispiel existiert ein solches Element nicht. **Entwerfe** einen Algorithmus, der in möglichst schneller asymptotischer Zeit diese Frage beantwortet. Laufzeit $O(\log(n))$ ist dabei möglich.

Eine Beschreibung des Algorithmus in Pseudo-Code genügt. Allerdings soll die Beschreibung so strukturiert, dass jeder anhand dieser Beschreibung den Algorithmus in seiner Lieblingsprogrammiersprache implementieren kann.

Aufgabe 26
n sei eine gerade Zahl. Das Maximum **und** das Minimum von n Zahlen $x_1, \ldots x_n$ kann offenbar mit $n-1+n-2 = 2n-3$ Vergleichen bestimmt werden. **Skizziere** einen Algorithmus, der Maximum und Minimum mit wesentlich weniger Vergleichen bestimmt.

Hinweis: $\frac{3}{2}n - 2$ Vergleiche sind ausreichend.

Aufgabe 27
Wir wollen die Anzahl der Multiplikationen und Additionen von Zahlen, bei der Polynommultiplikation untersuchen. Dazu betrachten wir 2 Polynome p und q vom Grad $n-1$ mit

$$p(x) = \sum_{k=0}^{n-1} p_k x^k \quad \text{und} \quad q(x) = \sum_{k=0}^{n-1} q_k x^k$$

Wir nehmen an, n sei eine Zweierpotenz.

(a) Geht man nach Schulmethode vor, so multipliziert man alle Paare von p und q Koeffizienten und fasst Ergebnisse mit gleicher Indexsumme zusammen. Wenn $r(x) = \sum_{k=0}^{2n-2} r_k x^k$ das Ergebnispolynom ist, berechnet man:

$$\begin{aligned}
r_0 &= p_0 \cdot q_0 \\
r_1 &= p_0 \cdot q_1 + p_1 \cdot q_0 \\
r_2 &= p_0 \cdot q_2 + p_1 \cdot q_1 + p_2 \cdot q_0 \\
&\vdots \\
r_{n-1} &= p_0 \cdot q_{n-1} + \ldots + p_{n-1} \cdot q_0 \\
&\vdots \\
r_{2n-2} &= p_{n-1} \cdot q_{n-1}
\end{aligned}$$

Wieviele Additionen und Multiplikationen fallen bei dieser Methode an?

(b) Wir wollen das Verfahren verbessern. Oft kann ein Vorteil gewonnen werden, wenn man ein Problem in kleinere Instanzen desselben Problems zerlegt (Divide & Conquer). Wir verfolgen diesen Ansatz auch hier und zerlegen p wie folgt.

$$\begin{aligned} p(x) &= \overline{p}(x) \cdot x^{n/2} + \underline{p}(x) \\ \overline{p}(x) &= \sum_{k=0}^{n/2-1} p_{k+n/2} \cdot x^k \\ \underline{p}(x) &= \sum_{k=0}^{n/2-1} p_k \cdot x^k \end{aligned}$$

Mit q verfahren wir genauso. So erhalten wir 4 Polynome $\overline{p}(x), \underline{p}(x), \overline{q}(x), \underline{q}(x)$ vom Grade $n/2-1$. Wir multiplizieren nun

$$p(x) \cdot q(x) = \overline{p}(x) \cdot \overline{q}(x) \cdot x^n + \overline{p}(x) \cdot \underline{q}(x) \cdot x^{n/2} + \underline{p}(x) \cdot \overline{q}(x) \cdot x^{n/2} + \underline{p}(x) \cdot \underline{q}(x)$$

Die Multiplikationen von Polynomen mit x-Potenzen sind leicht, da nur die Exponenten entsprechend erhöht werden müssen. Wir zählen sie daher nicht mit. **Gib** je eine Rekursionsgleichung für die Zahl der Multiplikationen in diesem Schema und für die Zahl der Additionen in diesem Schema **an** und **löse** sie.

(c) In einer Variante des Ansatzes berechnen wir nun rekursiv die Ergebnisse der folgenden drei Polynommultiplikationen für Polynome vom Grad $n/2-1$:

$$\overline{p}(x) \cdot \overline{q}(x),\ \underline{p}(x) \cdot \underline{q}(x) \ \text{ und } \ (\overline{p}(x) + \underline{p}(x)) \cdot (\overline{q}(x) + \underline{q}(x))$$

Beschreibe, wie sich aus den ermittelten Ergebnissen das Gesamtergebnis zusammensetzen lässt. **Gib** je eine Rekursionsgleichung für die Zahl der Multiplikationen in diesem Schema und die Zahl der Additionen in diesem Schema **an** und **löse** sie.

Hinweis: Subtraktionen gelten im Kontext dieser Aufgabe als Additionen.

Aufgabe 28

Du bekommst einen Job in einer Firma, in der bereits n Leute beschäftigt sind. Ein guter Freund, der **nicht** in besagter Firma arbeitet, verrät Dir, dass es in dieser Firma unehrliche Menschen gibt, aber mehr als die Hälfte der Mitarbeiter ehrlich ist. Ehrliche Menschen sagen stets die Wahrheit, die anderen lügen gelegentlich, aber auch nicht immer. Es darf davon ausgegangen werden, dass jeder in der Firma von jedem anderen weiß, ob es sich um einen ehrlichen Menschen handelt.

Natürlich möchtest Du so schnell wie möglich von jedem Deiner Kollegen wissen, ob es sich um einen ehrlichen Menschen handelt. Zu diesem Zweck kannst Du den Mitarbeitern Fragen stellen. Über einen konkreten Mitarbeiter kann man sich ein zutreffendes Bild verschaffen, indem man jeden seiner Kollegen über ihn befragt. Iteriert man das Verfahren, so muss man allerdings im schlimmsten Fall $\Omega(n^2)$ Fragen stellen. **Zeige**, dass $O(n)$ Fragen genügen. Dabei muss eine Frage stets an einen konkreten Mitarbeiter gestellt werden.

Hinweis: Versuche zuerst eine ehrliche Person ausfindig zu machen. Bilde dazu zunächst Paare (X_i, Y_i) und frage in jedem Paar i die Person X_i, ob Y_i ein ehrlicher Mensch ist, und Y_i, ob X_i ein ehrlicher Mensch ist. Analysiere die möglichen Antworten.

3.5 Laufzeit-Analyse von C++ Programmen

Wir werden in der Vorlesung zumeist die *uniforme* Laufzeit von Programmen analysieren: Das heißt, wir zählen die *Anzahl ausgeführter Anweisungen*. Um die Analyse zu vereinfachen, ohne aber die ermittelte Laufzeit asymptotisch zu verfälschen, werden wir nur elementare Anweisungen wie Zuweisungen und Auswahl-Anweisungen (if-else und switch) zählen. Insbesondere zählen wir weder iterative Anweisungen (for, while und do-while), noch Sprunganweisungen (break, continue, goto und return) oder Funktionsaufrufe; beachte aber, dass die in einer Schleife oder in einer Funktion ausgeführten elementaren Anweisungen sehr wohl zu zählen sind.

Uns interessiert nun sicher nicht, wieviele Anweisungen in einem Algorithmus auftreten (das entspräche etwa der Länge des Source-Codes, der in unserem Zusammenhang eine irrelevante

Größe ist), sondern vielmehr, wieviele Anweisungen während der Berechnung zur Ausführung kommen. Solange das Programm einfach Schritt für Schritt von oben nach unten abgearbeitet wird, haben wir keine Schwierigkeiten. Problematisch sind Schleifen und Verzweigungen im Programmablauf.

Für die Analyse von Auswahl-Anweisungen genügt es häufig, den Aufwand für die Auswertung der Bedingung und den Gesamtaufwand für den längsten der beiden alternativen Anweisungsblöcke zu bestimmen. Dies ist eine pessimistische Betrachtung, mit der man die Laufzeit einer Verzweigung nach oben abschätzen kann. Sicher wird es im konkreten Einzelfall möglich sein, durch zusätzliches Wissen um die Natur der Bedingung oder der Alternativen, zu besseren Schranken zu kommen.

Für die Analyse von Schleifen genügt in einfachen Situationen eine Bestimmung der maximalen Anzahl der ausführbaren Anweisungen des Anweisungsblocks der Schleife. Diese Maximalzahl ist mit der Anzahl der Schleifendurchläufe zu multiplizieren. Häufig ist also die Bestimmung der Schleifendurchläufe das wesentliche Problem in der Aufwandsbestimmung von Schleifen.

Beispiel 3.5 **(Matrizenmultiplikation)**

In dem folgenden Programmsegment werden zwei $n \times n$ Matrizen A und B miteinander multipliziert. Die Produktmatrix ist C:

```
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
      {
        C[i][j] = 0;
        for (k=0; k < n ; k++)
          C[i][j] += A[i][k]*B[k][j];
      }
```

Das Programm beginnt mit einer Schleife, die von 0 bis $n-1$ läuft und deren Rumpf eine Schleife von 0 bis $n-1$ ist. Der Rumpf dieser mittleren Schleife besteht aus einer Zuweisung (der C-Initialisierung) und einer dritten Schleife, die von 0 bis $n-1$ läuft und deren Rumpf nur eine arithmetische Operation beinhaltet.

Jede der Schleifen wird also n-mal durchlaufen. Wenn $T_i(n)$, $T_m(n)$ und $T_{\ddot{a}}(n)$ die Laufzeiten der inneren, mittleren und äußeren Schleife sind, so erhalten wir $T_{\ddot{a}}(n) = n \cdot T_m(n)$, $T_m(n) = n \cdot (1 + T_i(n))$ da die Initialisierung C[i][j] = 0 zu zählen ist und $T_i(n) = n$. Durch Einsetzen *von innen nach außen* erhalten wir $T_i(n) = n$, $T_m(n) = n \cdot (1 + n)$ und damit

$$\begin{aligned} T_{\ddot{a}}(n) &= n^2 \cdot (1 + n) \\ &= \theta(n^3). \end{aligned}$$

Wir haben Zuweisungen und arithmetische Operationen mit konstantem Aufwand veranschlagt. Auch Vergleiche werden wir in der Regel mit Aufwand $O(1)$ ansetzen. Im Abschnitt 3.6 werden wir auf die versteckten Gefahren dieser Sichtweise zurückkommen.

Beispiel 3.6 `while (n >= 1) { n = n/2; Eine Menge von maximal c Anweisungen (wobei c konstant in n ist). }`

$T(n)$ bezeichne die Laufzeit der while-Schleife für Parameter n. Die Berechnung von $T(n)$ gelingt mit Hilfe der Rekursionsungleichung

$$T(1) \leq 1 + c \quad \text{und} \quad T(n) \leq T(n/2) + 1 + c$$

und Satz 3.9 liefert die Lösung $T(n) = O(\log n)$.

Aufgabe 29
Bestimme in O-Notation die Laufzeit (bzw. den Wert der Variablen m) am Ende der folgenden Prozeduren in Abhängigkeit von der natürlichen Zahl n. Die Analyse sollte asymptotisch exakt sein. Die Funktion `floor()` berechnet die untere Gaußklammer und die Funktion `sqrt(n)` berechnet den Wert $\lfloor\sqrt{n}\rfloor$. .

(a)

```
int m=0;
while (n>1)
      {m++;n=floor(0.9n);}
```

(b)

```
int m=0;
while (n>1)
      {m++;n=sqrt(n);}
```

(c)

```
int m=0;
for(i=0;i<n;i++)
   for(j=i;j<=n;j++)
      for(k=1;k<=j;k++)
         m++;
```

Aufgabe 30
Bestimme exakt den Wert, den m nach Aufruf der beiden Programme in Abhängigkeit von $n > 0$ annimmt.

(a)
```
int procA(int n)
{ int m=1;
  int i=1;
  while(i < n)
    { m=m+i;
      i=i*2;
    }
  return m ;
}
```

(b)
```
int procB(int n)
{ int i,j,k;
  int m=0;
  for(i=0;i<=n;i++)
     for(j=i;j<=2*i;j++)
        for(k=1;k<=j;k++)
           m++;
  return m ;
}
```

In Beispiel 3.6 ist es uns also gelungen eine obere Schranke für die Zahl der Iterationen zu erhalten. Gänzlich anders verhält es sich in dem nächsten Beispiel.

Beispiel 3.7 `while (n > 1) n = ( n & 1 ) ?  3*n + 1 :  n/2;`

So simpel die Anweisung auch ist, es ist bis heute eine offene Frage, ob dieses Programm überhaupt auf jede Eingabe n hin anhält. Erst recht ist vollkommen ungeklärt ob, und wenn ja wie, sich die Zahl der Iterationen nach oben abschätzen lässt. Man beachte die fast willkürlich wirkenden Iterationszahlen in der partiellen Funktionstabelle.

n	Folge	It.Anz.
1	1	0
2	2, 1	1
3	3, 10, 5, 16, 8, 4, 2, 1	7
4	4, 2, 1	2
5	5, 16, 8, 4, 2, 1	5
6	6, 3, 10, 5, 16, 8, 4, 2, 1	8
7	7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1	16
8	8, 4, 2, 1	3
9	9, 28, 14, 7 (... ↑ 7 ...)	18
10	10, 5, 16, 8, 4, 2, 1	6
11	11, 34, 17, 52, 26, 13, 40, 20, 10 (...↑ 10...)	15
12	12, 6, 3, (... ↑ 3 ...)	9
13	13, 40, 20, 10, 5, 16, 8, 4, 2, 1	9
14	14, 7, (... ↑ 7 ...)	17
15	15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10 (... ↑ 10 ...)	17
16	16, 8, 4, 2, 1	4
17	17, 52, 26, 13 (...↑ 13 ...)	12
18	18, 9 (...↑ 9 ...)	19
19	19, 58, 29, 88, 44, 22, 11 (...↑ 11...)	21
20	20, 10 (...↑ 10 ...)	7
21	21, 64, 32, 16, 8, 4, 2, 1	7
22	22, 11 (...↑ 11 ...)	16
23	23, 70, 35, 106, 53, 160, 80, 40, 20 (...↑ 20...)	15
24	24, 12 (...↑ 12 ...)	10
25	25, 76, 38, 19 (...↑ 19...)	24
26	26, 13 (...↑ 13 ...)	10
27	27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23 (...↑ 23 ...)	112
28	28, 14 (...↑ 14 ...)	18
29	29, 88, 44, 22 (...↑ 22 ...)	19
30	30, 15 (...↑ 15 ...)	18
31	(...↑ 27 ab 6. Iteration...)	106
32	32, 16, 8, 4, 2, 1	5
33	33, 100, 50, 25 (...↑ 25...)	27
34	34, 17 (...↑ 17...)	13
35	35, 106, 53, 160, 80, 40, 20 (...↑ 20...)	13
36	36, 18, (...↑ 18...)	20
37	37, 112, 56, 28, (...↑ 28...)	21
38	38, 19, (...↑ 19...)	22
39	39, 118, 59, 178, 69, 208, 104, 52, 26 (...↑ 26...)	18
40	40, 20 (...↑ 20...)	8

3.6 Registermaschinen*

Grundsätzlich „denken“ wir unsere Algorithmen stets auf dem formalen Modell einer **Registermaschine** implementiert. Um die Wahl einer speziellen Programmiersprache und damit die Abhängigkeit von einem speziellen Compiler zu umgehen, wählen wir die Assembler-Programmiersprache. Damit können wir also *rechner-* und *compiler-unabhängige* Aussagen über die Effizienz der implementierten Algorithmen machen.

Die **Architektur** einer Registermaschine besteht aus

- einem linear geordneten Eingabeband, das von links nach rechts gelesen wird. Die Zellen des Bandes können ganze Zahlen speichern.

- der CPU. In der CPU ist das Programm gespeichert, wobei ein Befehlszähler den gegenwärtig auszuführenden Befehl festhält. Weiterhin besitzt die CPU einen Akkumulator,

der im Stande ist, die arithmetischen Operationen Addition, Subtraktion, Multiplikation und Division auszuführen. Schließlich erlaubt der Akkumulator auch, Vergleiche auszuführen.

- einem Speicher, der aus Registern besteht. Wir nehmen an, dass wir eine unbeschränkte Anzahl von Registern zur Verfügung haben und dass jedes Register eine ganze Zahl speichern kann.

- einem linear geordneten Ausgabeband, das von links nach rechts beschrieben wird. Eine Zelle des Bandes kann wiederum eine ganze Zahl speichern.

Die Register versehen wir mit den Adressen 1,2,.... Den Inhalt (contents) des i-ten Registers bezeichnen wir mit $c(i)$. Wir gehen als nächstes die Befehlsliste unserer Assembler-Programmiersprache durch:

Ein/Ausgabe.

- Read: Der Akkumulator erhält die ganze Zahl, die gegenwärtig vom Lesekopf gelesen wird. Der Lesekopf wird dann zur rechten Nachbarzelle bewegt. Sollte hingegeben die Eingabe schon vollständig gelesen sein, erhält der Akkumulator den Wert *eof*, end of file.

- Write: Der Inhalt des Akkumulators wird auf das Ausgabeband geschrieben, und der Ausgabekopf wird zur rechten Nachbarzelle bewegt.

Arithmetik.

- Add i: Akkumulator: = Akkumulator $+$ $c(i)$.

- Sub i: Akkumulator: = Akkumulator $-$ $c(i)$.

- Mult i: Akkumulator: = Akkumulator $\cdot$ $c(i)$.

- Div i Akkumulator: $= \begin{cases} \text{Fehlerhalt} & \text{wenn } c(i) = 0 \\ \lfloor \frac{\text{Akkumulator}}{c(i)} \rfloor & \text{sonst.} \end{cases}$

- CAdd i, CSub i, CMult i, CDiv i sind die entsprechenden Konstantenbefehle. So addiert CAdd i die Konstante i zum gegenwärtigen Inhalt des Akkumulators.

Sprünge.

- Goto j: Setze den Befehlszähler auf j.

- if (Akkumulator $<, \leq, =, \neq, \geq, > i$) goto j: Entsprechend dem Ausgang des Vergleichs wird der Befehlszähler auf j gesetzt oder um 1 erhöht. i ist dabei eine numerische Konstante oder es ist $i = eof$.

- End: Programm hält.

Direkter Speicherzugriff.

- Load i: Akkumulator: = $c(i)$.

- Store i: $c(i)$: = Akkumulator.

Indirekter Speicherzugriff.

- ILoad i: Akkumulator: $= c(|c(i)|)$.
- IStore i: $c(|c(i)|) :=$ Akkumulator.

(Zu Beginn der Rechnung werden alle Registerinhalte auf 0 gesetzt.)

Nachdem das formale Rechnermodell und seine Programmiersprache eingeführt ist, können wir Laufzeit und Speicherplatzkomplexität von Programmen bestimmen. Bei der Bestimmung der Laufzeit wie auch der Speicherplatzkomplexität müssen wir aber vorsichtig vorgehen: Wenn wir nur die Anzahl der ausgeführten Befehle festhalten, nehmen wir an, dass komplexe Befehle (wie die Division langer Zahlen) genauso schnell ausgeführt werden können wie simple Befehle (Goto j).

Nehmen wir die Addition als Beispiel. Bekanntermaßen liegen Zahlen im Rechner intern in ihrer Binärdarstellung vor. Die Länge der Binärdarstellung entspricht dem abgerundeten 2er-Logarithmus der Zahl plus 1. Um zwei Zahlen zu addieren, arbeitet der Rechner von den niederwertigsten Bits zu den hochwertigsten. Er addiert binär, verrechnet das Resultat ggf. noch mit einem Übertrag, trägt das Ergebnis in den dafür angelegten Speicherplatz ein und testet, ob zur nächsten Binäraddition ein Übertrag mitzuschleppen ist: Eine Menge Arbeit für Konstantzeit.

Üblicherweise werden wir bei diesen Operationen trotzdem großzügig Konstantzeit annehmen. Es empfiehlt sich aber, diese Problematik im Hinterkopf präsent zu halten, wenn man mit sehr großen Zahlen arbeitet. In diesen Fällen sollte man dann unbedingt einen modifizierten Begriff der Eingabelänge benutzen. Man wird dann eine Zahl n nicht als ein einzelnes Eingabedatum betrachten, sondern als einen Eingabestring der Länge $\lfloor \log_2 n \rfloor + 1$.

Wenn wir diesen Ansatz konsequent durchführen, werden wir damit auf die **logarithmische** Zeitkomplexität geführt, die im Gegensatz zur bisher von uns betrachteten **uniformen** Zeitkomplexität nicht einfach die Anzahl der ausgeführten Anweisungen bestimmt, sondern die Kosten einer Operation von der Länge der Operanden abhängig macht.

Definition 3.3 P sei ein Programm für eine Registermaschine und $x = (x_1, \ldots, x_n)$ sei eine Eingabe

(a) $\text{UZeit}_P(x) :=$ Anzahl der Befehle, die P für Eingabe x ausführt.
$\text{UZeit}_P(x)$ ist die uniforme Zeitkomplexität von P für Eingabe x.

(b) Sei b ein arithmetischer Befehl mit den Operanden x, dem Wert des Akkumulators, und y, dem Wert des Registers. Wir weisen dem Befehl die logarithmischen Kosten

$$\lfloor \log_2(|x|) \rfloor + 1 + \lfloor \log_2(|y|) \rfloor + 1$$

zu, also die Summe der Längen der Binärdarstellungen der beiden Operanden.
Ein Lese- oder Schreibbefehl mit Operand x erhält die logarithmischen Kosten

$$\lfloor \log_2(|x|) + 1 \rfloor.$$

Ein Sprungbefehl wird weiterhin als ein Befehl gezählt und erhält also die Kosten 1.
Der indirekte Load-Befehl, mit $x = c(i)$ und $y = c(|c(i)|)$), erhält die logarithmischen Kosten

$$\lfloor \log_2(|x|) \rfloor + 1 + \lfloor \log_2(|y|) \rfloor + 1.$$

Der indirekte Store-Befehl erhält dieselbe Kostenzuweisung, wobei aber jetzt $x = c(i)$ und y der Wert des Akkumulators ist.

Wir definieren $\text{LZeit}_P(x)$ als die gemäß den logarithmischen Kosten des jeweiligen Befehls gewichtete Summe der Befehle, die P für Eingabe x ausführt. $\text{LZeit}_P(x)$ ist die logarithmische Zeitkomplexität von P für Eingabe x.

Die Speicherplatzkomplexität behandeln wir analog:

Definition 3.4 P sei ein Programm für eine Registermaschine und $x = (x_1, \ldots, x_n)$ sei eine Eingabe.

(a) $\text{UPlatz}_P(x)$ ist die größte Adresse eines Registers, das von P für Eingabe x benutzt wird. $\text{UPlatz}_P(x)$ heißt die uniforme Speicherplatzkomplexität von P für Eingabe x.

(b) Für $1 \leq i \leq UPlatz_P(x)$ sei y_i der Absolutbetrag der vom Absolutbetrag aus gesehen größten im Register i gespeicherten Zahl. y_0 sei die vom Absolutbetrag aus gesehen größte im Akkumulator gespeicherte Zahl. Dann ist

$$\text{LPlatz}_P(x) = \sum_{i=0}^{s} (\lfloor \log_2(y_i) \rfloor + 1)$$

die logarithmische Speicherplatzkomplexität von P für Eingabe x.

Es macht keinen Sinn, jeder Eingabe x eine Laufzeit (oder einen Speicherplatzbedarf) zuzuweisen. Stattdessen versuchen wir, eine Laufzeitfunktion in kompakter Form zu berechnen, wobei wir Eingaben gemäß ihrer **Eingabelänge** zusammenfassen.

Wenn wir zum Beispiel n ganze Zahlen $x_1, \ldots, x_n$ sortieren möchten, dann bietet sich

$$n = \text{ Anzahl der zu sortierenden Zahlen}$$

als Eingabelänge an. Eine Aussage der Form

„Bubblesort benötigt $O(n^2)$ Schritte zum Sortieren von n Zahlen“

ist sicherlich klarer als eine Auflistung der Laufzeiten von Bubblesort für spezifische Eingaben. Für das Sortierproblem ist die Anzahl der zu sortierenden Zahlen aber nur als Eingabelänge für die uniforme Zeitkomplexität angemessen. Für die logarithmische Zeitkomplexität macht diese Form der Eingabelänge keinen Sinn: Da es beliebig lange Zahlen gibt, kann einer festen Eingabelänge keine obere Laufzeitschranke zugewiesen werden. Im Fall der logarithmischen Zeitkomplexität bietet sich

$$\sum_{i=1}^{n} (\lfloor \log_2(|x_i|) \rfloor + 1)$$

als Eingabelänge an.

Definition 3.5 Für eine Menge X (z. B. $X = \mathbb{Z}, X = \{0, 1\}$) ist

$$X^i = \{(a_1, \ldots, a_i) : a_1, \ldots, a_i \in X\}$$

die Menge aller Worte der Länge i über X und

$$X^* = \bigcup_{i=0}^{\infty} X^i$$

ist die Menge aller Worte über X. X^0 besteht nur aus dem leeren Wort, das man auch häufig durch ε abkürzt.

Wir kommen im nächsten und letzten Schritt zur Definition der Laufzeit- und Speicherplatzkomplexität *relativ* zu einem vorgegebenen Begriff der Eingabelänge. Wir benutzen dieselbe Notation wie in Definition 3.3: Zum Beispiel wird UZeit_P in Definition 3.3 auf *Eingaben* definiert; hier definieren wir UZeit_P auf *Eingabelängen*.

Definition 3.6 A sei eine Teilmenge von $\mathbb{Z}^*$, der Menge der Eingaben. Die Funktion

$$\text{länge} : A \to \mathbb{N}$$

weise jeder möglichen Eingabe eine Eingabelänge zu. Schließlich sei P ein Programm für eine Registermaschine.

(a) $\text{UZeit}_P(n)$ ist die **uniforme worst-case Zeitkomplexität** von P, nämlich

$$\text{UZeit}_P(n) := sup\{\text{UZeit}_P(x) \ : \ x \in A \text{ und länge}(x) = n\}.$$

(b) $\text{LZeit}_P(n)$ ist die **logarithmische worst-case Zeitkomplexität** von P, nämlich

$$\text{LZeit}_P(n) := sup\{\text{LZeit}_P(x) \ : \ x \in A \text{ und länge}(x) = n\}.$$

(c) $\text{UPlatz}_P(n)$ ist die **uniforme worst-case Speicherplatzkomplexität** von P, nämlich

$$\text{UPlatz}_P(n) := sup\{\text{UPlatz}_P(x) \ : \ x \in A \text{ und länge}(x) = n\}.$$

(d) $\text{LPlatz}_P(n)$ ist die **logarithmische worst-case Speicherplatzkomplexität** von P, wobei

$$\text{LPlatz}_P(n) := sup\{\text{LPlatz}_P(x) \ : \ x \in A \text{ und länge}(x) = n\}.$$

Aufgabe 31
In dieser Aufgabe werden wir sehen, dass sehr schnell große Zahlen generiert werden können: Die logarithmische Laufzeit sagt also die tatsächliche Laufzeit im allgemeinen exakter voraus.

(a) Wir betrachten zuerst Programme, in denen als Operationen nur die Addition und Subtraktion von zwei Operanden erlaubt sind. Als Konstanten dürfen nur die natürlichen Zahlen $0, 1, \ldots, k$ verwandt werden.

Bestimme die größte Zahl, die mit maximal t Operationen berechnet werden kann und **begründe** deine Antwort.

(b) Wir lassen jetzt als dritte Operation die Multiplikation zweier Operanden zu. **Bestimme** wiederum die größte Zahl, die mit maximal t Operationen berechnet werden kann und **begründe** deine Antwort.

Beispiel 3.8 : Wir betrachten wieder das Teilfolgenproblem. Dann ist

$$\text{UZeit}_{A_1}(n) = O(n^3), \text{UZeit}_{A_2}(n) = O(n^2)$$

$$\text{UZeit}_{A_3}(n) = O(n \log\ n), \text{UZeit}_{A_4}(n) = O(n).$$

Streng genommen müßten wir die vier Algorithmen zuerst auf einer Registermaschine implementieren und dann analysieren. Doch wird eine geschickte Implementierung die oben behaupteten Laufzeiten auch einhalten können: Additionen und Vergleiche dominieren die Laufzeit für jeden der vier Algorithmen. In Registermaschinen können wir zwar nur mit Konstanten

vergleichen, aber ein Vergleich beliebiger Zahlen kann durch eine Subtraktion und einen darauffolgenden Vergleich mit 0 simuliert werden. Ein Vergleich kann somit durch höchstens 3 Assembleranweisungen (Load, Subtraktion und Vergleich auf 0) nachvollzogen werden, eine Addition benötigt höchstens 2 Assembleranweisungen (Load und Addition).

Alle vier Algorithmen lassen sich also mit konstantem Zeitverlust auch auf einer Registermaschine implementieren. Wie sieht es mit der uniformen Speicherplatzkomplexität aus?

$$\text{UPlatz}_{A_1}(n) = \text{UPlatz}_{A_2}(n) = O(n^2) \text{ sowie}$$

$$\text{UPlatz}_{A_3}(n) = O(\log_2 n), \text{UPlatz}_{A_4}(n) = O(1).$$

Algorithmus A_3 benötigt logarithmischen Speicherplatz für die Ausführung der Rekursion: Wie wir im nächsten Kapitel sehen werden, können wir mit Hilfe eines Stacks logarithmischer Größe die Rekursion simulieren. Um konstante Speicherplatzkomplexität für Algorithmus A_4 zu erreichen, beachte, dass wir tatsächlich nur zwei Variablen Max und Max* - an Stelle der $2n$ Variablen Max_k und Max_k^* - benötigen.

Betrachten wir nun die logarithmische Zeitkomplexität. Wir wechseln zu einem neuen Begriff der Eingabelänge, um die Längen der Binärdarstellungen der Eingaben festzuhalten. Statt der Eingabelänge n für $a_1, \ldots, a_n$ betrachten wir jetzt

$$N_1 = \sum_{i=1}^{n} (\lfloor \log(|a_i|) \rfloor + 1)$$

als Eingabelänge. Für die Analyse halten wir n, die Anzahl der Schlüssel a_i, fest. Beachte, dass jede Operation logarithmische Kosten von höchstens $O(\log n + N_1)$ hat, denn alle Algorithmen addieren bis zu höchstens n Operanden, und die logarithmische Größe des größten Operanden ist deshalb durch

$$\lfloor \log(|a_1| + \cdots + |a_n|) \rfloor + 1 \leq N_1$$

beschränkt. (Eine indirekte Load/Store Operation bei n benutzten Registern kann die Kosten $\log n$ erreichen, selbst wenn die Operanden klein sind: Dies geschieht zum Beispiel dann, wenn auf ein Register mit der Addresse n zugegriffen wird.) Wir erhalten deshalb

$$\text{LZeit}_{A_1}(N_1) = O(n^3(\log n + N_1)), \qquad \text{LZeit}_{A_2}(N_1) = O(n^2(\log n + N_1)),$$

$$\text{LZeit}_{A_3}(N_1) = O((n \log\ n)(\log n + N_1)), \qquad \text{LZeit}_{A_4}(N_1) = O(n \cdot (\log n + N_1)).$$

Wenn die $|a_i|$ „ungefähr“ gleich groß sind, lassen sich die obigen Abschätzungen verbessern. Dazu betrachten wir die Eingabelänge

$$N_2 = \max\{\lfloor \log(|a_i|) + 1 \rfloor | 1 \leq i \leq n\}$$

(Wenn die $|a_i|$ ungefähr gleich groß sind, folgt $N_2 \approx N_1/n$.) Wir erhalten mit der neuen Eingabelänge N_2:

$$\begin{aligned}
\text{LZeit}_{A_1}(N_2) &= O(n^3(\log n + N_2)), \\
\text{LZeit}_{A_2}(N_2) &= O(n^2(\log n + N_2)), \\
\text{LZeit}_{A_3}(N_3) &= O(n \log\ n(\log\ n + N_2)), \\
\text{LZeit}_{A_4}(N_2) &= O(n(\log\ n + N_2)).
\end{aligned}$$

Warum? Es sei $|a_i| = \max\{|a_1|, \ldots, |a_n|\}$. Dann werden die Algorithmen nur Operanden der logarithmischen Größe höchstens

$$\begin{aligned}
\lfloor \log(|a_1| + \cdots + |a_n|) \rfloor + 1 &\leq \lfloor \log(n \cdot |a_i|) \rfloor + 1 \\
&\leq \lfloor \log\ n \rfloor + N_2
\end{aligned}$$

erzeugen.

3.7 Zusammenfassung

In diesem Kapitel haben wir uns vorwiegend mit dem Aspekt der Analyse von Algorithmen beschäftigt. Um eine Rechner-unabhängige Analyse anbieten zu können, beziehen sich unsere Laufzeit-Berechnungen (bzw. Speicherplatzberechnungen) auf das formale Modell der Registermaschine. Im folgenden geben wir aber nicht Implementierungen auf Registermaschinen explizit an, sondern verweisen darauf, dass die von uns benutzten Makrobefehle sich mit konstantem Zeitverlust auf Registermaschinen implementieren lassen. Ein konstanter Zeitverlust ist aber für eine asymptotische Analyse nicht von Belang. (Jedoch ist ein konstanter Zeitverlust für die Praxis von Belang. Man überzeuge sich stets, dass es sich um „kleine“ Zeitverluste handelt.)

Wir haben die uniforme Laufzeit/Speicherplatz-Komplexität betrachtet. Im allgemeinen erhalten wir keine verlässlichen Prognosen für die tatsächliche Laufzeit, nämlich dann nicht, wenn man „schmutzige Programmiertricks“ mit großen Zahlen betreibt. Das logarithmische Kostenmaß ist hingegen verlässlicher, aber schwieriger auszuwerten. Ansonsten, wenn die Registerinhalte moderat groß sind, ist das uniforme Kostenmaß verlässlich und hat den Vorteil der leichten Anwendbarkeit. Wir werden deshalb von jetzt an das uniforme Kostenmaß verwenden.

Wir haben die asymptotischen Relationen O, o, Ω, ω und Θ betrachtet. Wenn zum Beispiel $T(n)$ die Laufzeit eines Algorithmus ist, dann bedeutet die Aussage

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

dass die Laufzeit des Algorithmus, bis auf eine Konstante, durch $f(n)$ beschränkt ist. Sind $T_1(n)$ und $T_2(n)$ die Laufzeiten zweier Algorithmen, so bedeutet die Aussage

$$T_1(n) = o(T_2(n)),$$

dass der erste Algorithmus für genügend große Eingabelängen schneller als der zweite ist.

Wir haben gesehen, dass es in vielen Fällen ausreicht, Grenzwerte zu bestimmen, um eine asymptotische Relation zu etablieren. Ein weiteres wichtiges Hilfsmittel für die asymptotische Analyse ist das Integralkriterium.

Schließlich haben wir Hilfsmittel zur Analyse rekursiver Programme erarbeitet. Insbesondere haben wir mit Hilfe des Mastertheorems eine asymptotische Lösung der Rekursionsgleichung

$$T(1) = c$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

angeben können.

Wie relevant ist eine asymptotische Analyse, denn letztendlich interessiert ja nicht das Laufzeitverhalten für beliebig große Daten? Mit anderen Worten, wann, bzw. für welche Eingabelängen sagt eine asymptotische Aussage die tatsächliche Laufzeit akkurat voraus?

In dem folgenden Beispiel nehmen wir an, dass ein einfacher Befehl in 10^{-9} Sekunden ausgeführt werden kann. (Damit können höchstens $60 \cdot 10^9 \leq 10^{11}$ Befehle in der Minute, $3600 \cdot 10^9 \leq 10^{13}$ Befehle in der Stunde, $86400 \cdot 10^9 \leq 10^{14}$ Befehle am Tag und höchstens $10^8 \cdot 10^9 \leq 10^{17}$ Befehle im Jahr ausgeführt werden). Für Eingabelänge n betrachten wir die Laufzeitfunktionen $n^2, n^3, n^{10}, 2^n$ und $n!$

n	n^2	n^3	n^{10}	2^n	$n!$
16	256	4.096	$\geq 10^{12}$	65536	$\geq 10^{13}$
32	1.024	32.768	$\geq 10^{15}$	$\geq 4 \cdot 10^9$	$\underbrace{\geq 10^{31}}$
64	4.096	262.144	$\underbrace{\geq 10^{18}}$	$\underbrace{\geq 6 \cdot 10^{19}}$	
					10^{14} Jahre
128	16.384	2.097.152	mehr als	mehr als	Rechenzeit
256	65.536	16.777.216	10 Jahre	600 Jahre	
512	262.144	134.217.728			
1024	1.048.576	$\geq 10^9$			

Für Eingabelänge $n = 1024$ benötigen n^2 Befehle ungefähr 10^{-3} Sekunden Rechenzeit, n^3 Befehle aber bereits eine Sekunde. Für n^{10} kommt das definitive Aus für Eingabelänge 128: mehr als 100.000 Jahre Rechenzeit werden benötigt. 2^n kommt für Eingabelänge 128 auf „beeindruckende“ 10^{22} Jahre. Der Champion der „Rechenzeit-Fresser“ ist $n!$: Schon für Eingabelänge 32 sind Algorithmen mit Laufzeit $n!$ indiskutabel.

Kapitel 4

Elementare Datenstrukturen

C++ stellt bereits viele Datentypen zur Verfügung: Neben Grundtypen wie char, int, double und float, können abgeleitete Typen wie Zeiger, Referenzen und Arrays definiert werden. Weiterhin erlauben Strukturen die Bildung von benutzer-definierten Datengruppen.

Betrachten wir als nächstes **abstrakte Datentypen**. Ein abstrakter Datentyp ist eine (weitgehend unspezifizierte) Menge von **Objekten** sowie **Operationen**, die auf den Objekten definiert sind. Beispiele einfacher abstrakter Datentypen sind

- Stacks (mit den Operationen `push` (Einfügen eines Objektes) und `pop` (Entfernen des „jüngsten“ Objektes)
- und Queues mit den Operationen `enqueue` (Einfügen eines Objektes) und `dequeue` (Enfernen des „ältesten“ Objektes).

Wir sind vor allem an *effizienten Implementierungen* wichtiger abstrakter Datentypen, also an effizienten **Datenstrukturen** interessiert. Für die Kodierung von Datenstrukturen wird sich das Klassenkonzept von C++ als sehr hilfreich herausstellen.

C++ stellt das Template-Konzept zur Verfügung, um Klassen auf verschiedene Datentypen anwenden zu können. Wir erinnern an diese Möglichkeit, werden Sie aber im folgenden nicht benutzen.

4.1 Lineare Listen

Im folgenden betrachten wir eine Zeiger-Implementierung von einfach-verketteten Listen (also Listen mit Vorwärtszeigern). Zweifach-verkettete Listen (Listen mit Vorwärts- und Rückwärtszeigern) können analog behandelt werden.

Zuerst müssen wir uns darüber klar werden, welche Funktionen zur Verfügung gestellt werden sollten. Die Schnittstelle der unten deklarierten Klasse *liste* stellt die Funktionen insert und remove (für das Einfügen und Entfernen von Elementen) zur Verfügung, erlaubt Bewegungen in der Liste (durch die Funktionen movetofront und next) und Tests über den Status der Liste (ist die Liste leer? bzw. ist das Ende der Liste erreicht?). Weiterhin können die Werte der Listenelemente gelesen bzw. überschrieben werden und eine Suche nach einem Element kann durchgeführt werden.

```
//Deklarationsdatei liste.h fuer einfach-verkettete Listen.

enum boolean {False, True};

class liste{

private:
  typedef struct Element {
    int data;
    Element *next;};
  Element *head, *current;

public:
  liste( ) // Konstruktor
     {head = new Element; current = head; head->next = 0;}

  void insert( int data );
  // Ein neues Element mit Wert data wird nach dem gegenwaertigen
  // Element eingefuegt.  Der Wert von current ist unveraendert.

  void remove( );
  // Das dem gegenwaertigen Element folgende Element wird entfernt.
  // Der Wert von current ist unveraendert.

  void movetofront( );
  // current erhaelt den Wert head.

  void next( );
  // Das naechste Element wird aufgesucht: current wird um eine
  // Position nach rechts bewegt.

  boolean end( );
  // Ist das Ende der Liste erreicht?

  boolean empty( );
  // Ist die Liste leer?

  int read( );
  // Gib das Feld data des naechsten Elements aus.

  void write(int wert);
  // Ueberschreibe das Feld data des naechsten Elements mit der Zahl wert.

  void search(int wert);
  // Suche, rechts von der gegenwaertigen Zelle, nach der ersten Zelle z
  // mit Datenfeld wert.  Der Zeiger current wird auf den Vorgaenger
  // von z zeigen.
};
```

Man beachte, dass diese Implementierung stets eine leere Kopfzelle besitzt: liste() erzeugt diese Zelle, nachfolgende Einfügungen können nur nach der Kopfzelle durchgeführt werden und eine Entfernung der Kopfzelle ist unmöglich wenn man der Funktionsbeschreibung folgt. Diese Implementierung führt zu einer vereinfachten Definition der Member-Funktionen.

Für die Implementierung der einzelnen Funktionen sei auf die Vorlesung „Grundlagen der Informatik 1“ verwiesen. Man überzeuge sich, dass alle Funktionen, bis auf die Funktion search(), in konstanter Zeit ausgeführt werden können!

Diese Klasse ist ausreichend, um viele Anwendungen einfach-verketterter Listen effizient zu unterstützen. Allerdings kann es, je nach Anwendung, Sinn machen, die Daten *sortiert* zu halten, zum Beispiel um eine erfolglose Suche frühzeitig abbrechen zu können. Dazu wird eine zweite Version der insert-Operation benötigt und die Funktion search ist so zu modifizieren, dass eine erfolglose Suche schnell abgebrochen werden kann. Zwei weitere nützliche Operationen, die nicht in der Klasse liste vertreten sind, betreffen die *Konkatenation* von zwei Listen (hänge Liste_2 an das Ende von Liste_1) und das Entfernen aller Elemente (rechts oder links vom gegenwärtigen Element).

Aufgabe 32
Die Klasse „liste“ sei durch die obige Deklarationsdatei `liste.h` definiert.

(a) Wir möchten auch einen Zeiger `*tail` zur Verfügung stellen, um auf das Listenende zugreifen zu können sowie eine Funktion `movetotail()`.
Was ist zu tun, damit eine Zelle am Listenende eingefügt werden kann, bzw. damit das Listenende entfernt werden kann?

(b) Eine gegebene einfach verkettete Liste ist zu invertieren. Beschreibe, wie eine solche Invertierung nur mit den Operationen der Klasse `liste` gelingt.

Der große Vorteil von Listen ist, dass sie sich dynamisch den Speichererfordernissen anpassen. Diese Fähigkeit macht sie zu einer attraktiven Datenstruktur zum Beispiel für Graphen, wie wir später sehen werden. Jetzt betrachten wir eine weitere Verwendung von Listen, nämlich zur **Addition dünnbesetzter Matrizen**.

A, B und C seien Matrizen mit n Zeilen und m Spalten. Die Addition

$$C = A + B$$

wird durch den folgenden Algorithmus durchgeführt:

```
for (i=0 ; i < n; i++)
  for (j=0 ; j < m; j++)
    C[i,j] = A[i,j] + B[i,j];
```

Die uniforme Laufzeit ist somit $O(n \cdot m)$. Wir nutzen also nicht aus, dass A und B beide dünn besetzt sein könnten, also beide wenige von Null verschiedene Einträge besitzen könnten.

Angenommen, A hat a und B hat b von Null verschiedene Einträge. Unser Ziel ist eine Addition von A und B in Zeit $O(a+b)$. Wir verlangen allerdings, dass A und B in geeigneten Datenstrukturen vorliegen: Für A wie auch B sei eine Liste (L_A bzw. L_B) vorhanden, in der die von Null verschiedenen Einträge in „Zeilenordnung“ verkettet sind. Dabei ist neben dem Wert eines Eintrags auch die Zeile und die Spalte des Eintrags zu speichern.

Beispiel 4.1 $A \equiv \begin{bmatrix} 0 & 9 & 0 \\ 7 & 0 & 5 \\ 0 & 0 & 6 \end{bmatrix}$

Das erste Feld bezeichnet also jeweils die Zeile, das zweite Feld die Spalte und das dritte Feld den Wert des Eintrags. Die von Null verschiedenen Einträge der ersten Zeile erscheinen zuerst, gefolgt von der zweiten Zeile usw. Innerhalb einer Zeile erscheinen die Einträge gemäß aufsteigender Spaltennummer.

Es ist jetzt einfach, eine Liste L_c zu erstellen, die die von Null verschiedenen Einträge von $C = A + B$ in Zeilenordnung verkettet:

(1) Beginne jeweils am Anfang der Listen L_A und L_B.

(2) Solange beide Listen nicht-leer sind, wiederhole

   (a) das gegenwärtige Listenelement von L_A (bzw. L_B) habe die Koordinaten (i_A, j_A) (bzw. (i_B, j_B)).

   (b) Wenn $i_A < i_B$ (bzw. $i_A > i_B$), dann füge das gegenwärtige Listenelement von L_A (bzw. L_B) in die Liste L_C ein und gehe zum nächsten Listenelement von L_A (bzw. L_B).

   (c) Wenn $i_A = i_B$ und $j_A < j_B$ (bzw. $j_A > j_B$), dann füge das gegenwärtige Listenelement von L_A (bzw. L_B) in die Liste L_C ein und gehe zum nächsten Listenelement von L_A (bzw. L_B).

   (d) Wenn $i_A = i_B$ und $j_A = j_B$, dann addiere die beiden Einträge und füge die Summe in die Liste L_C ein. Die Zeiger in beiden Listen werden nach rechts bewegt.

(3) Wenn die Liste L_A (bzw. L_B) leer ist, kann der Rest der Liste L_B (bzw. L_A) an die Liste L_C angehängt werden.

Da jeder von Null verschiedene Eintrag von A oder B in konstanter Zeit behandelt wird und die Nulleinträge überhaupt nicht mehr betrachtet werden, ist die Laufzeit tatsächlich durch $O(a + b)$ beschränkt.

Wir bemerken noch, dass $a = O(n \cdot m)$ sowie $b = O(n \cdot m)$ und damit $a + b = O(n \cdot m)$ gilt. Das heißt auch bei Matrizen, die nicht dünn besetzt sind, ist dieser Algorithmus *asymptotisch* der Schulmethode gleichwertig. Man sieht aber auch, dass diese Methode bei fast voll besetzten Matrizen der Standardtechnik in den Konstanten weit unterlegen ist.

Auch sei darauf hingewiesen, dass eine Überführung einer Matrix von der Listendarstellung in die gewohnte, tabellarische Darstellung in Zeit $O(a + b)$ möglich ist, nicht jedoch die Gegenrichtung, die $O(n \cdot m)$ benötigt. Es ist also nicht ratsam, permanent zwischen den verschiedenen Darstellungsformen hin und her zu transformieren, in der Hoffnung stets das Letzte an Effizienz herauszukitzeln.

Aufgabe 33

In dieser Aufgabe interessieren wir uns für dünnbesetzte Matrizen ($A : [1 \dots n] \times [1 \dots n]$). Solche Matrizen in einem Array der Größe n^2 abzuspeichern wäre Speicherplatzverschwendung. Die obige Darstellung dünnbesetzter Matrizen ist ungeeignet, wenn wir Matrizen sowohl addieren als auch multiplizieren wollen.
Entwirf eine speicherplatzeffiziente Datenstruktur, die sowohl die Addition als auch die Multiplikation zweier Matrizen unterstützt und **beschreibe** wie man in dieser Darstellung zwei Matrizen multipliziert.

(Der Algorithmus für die Multiplikation soll natürlich möglichst effizient und auf keinen Fall mehr als die für übliche Matrizenmultiplikation benötigte Laufzeit $O(n^3)$ benötigen.)

Gib den Speicherplatzbedarf der Darstellung und die Laufzeit des Algorithmus an.

Aufgabe 34
$A[0]$ und $A[1]$ seien zwei aufsteigend sortierte Listen von insgesamt n ganzen Zahlen. **Beschreibe** einen Algorithmus, der die beiden Listen *mischt*: Der Algorithmus muss eine aufsteigend sortierte Liste B berechnen, die aus genau den Zahlen in $A[0]$ und den Zahlen in $A[1]$ besteht. **Analysiere** die Laufzeit deines Algorithmus; Laufzeit $O(n)$ ist erreichbar.

Diese Betrachtungen deuten an, dass die Frage nach der *besten Datenstruktur* ebenso sinnvoll ist wie die Frage nach dem besten Auto. (Wer würde mit einem Kleinwagen einen Umzug bewältigen wollen, und wer würde mit einem LKW in der Innenstadt einen Parkplatz suchen wollen?) Die Frage nach der *besten* Datenstruktur muss also stets die Gegenfrage: „Für welche Zwecke?“ nach sich ziehen.

4.2 Stacks, Queues und Deques

Stacks und Queues sind besonders einfache Datenstrukturen. In einem **Stack** werden nur die Operationen

- `pop` : Entferne den jüngsten Schlüssel.
- `push` : Füge einen Schlüssel ein.

unterstützt. Eine **Queue** modelliert das Konzept einer Warteschlange und unterstützt die Operationen

- `dequeue`: Entferne den ältesten Schlüssel.
- `enqueue` : Füge einen Schlüssel ein.

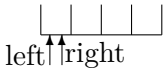
Wir geben als Nächstes eine Implementierung von Stacks und Queues an, die auf Arrays basiert. Dazu betrachten wir zuerst **Deques**, eine Verallgemeinerung von Stacks und Queues. Ein Deque unterstützt die Operationen

- `insertleft` : Füge einen Schlüssel am linken Ende ein.
- `insertright` : Füge einen Schlüssel am rechten Ende ein.
- `removeleft` : Entferne den Schlüssel am linken Ende.
- `removeright` : Entferne den Schlüssel am rechten Ende.

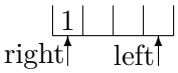
Wie werden die Deque-Operationen implementiert? Man denke sich das 1-dimensionale Array an den beiden Enden zusammengeschweißt. Der Inhalt des Deques wird jetzt entsprechend den Operationen „über den entstandenen Ring“ geschoben. Wir benutzen zwei Positionsvariable *left* und *right*, wobei left jeweils **vor** dem linken Ende und right jeweils **auf** dem rechten Ende steht. (Wir fordern, dass die Zelle mit Index left stets leer ist. Dann bedeutet (left == right), dass das Deque leer ist, vorausgesetzt, wir benutzen die Initialisierung left = right.)

Beispiel 4.2 ($n = 3$)

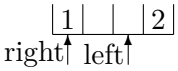
(1) Anfangs:



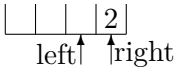
(2) insertleft(1):



(3) insertleft(2):



(4) removeright():



Da Deques eine Verallgemeinerung von Stacks und Queues sind, bietet es sich an, Klassen stack (für Stacks) und queue (für Queues) als abgeleitet von einer Basisklasse deque aufzufassen:

```
// Deklarationsdatei deque.h fuer Deques, Stacks und Queues
enum boolean {False, True};
class deque {
private:
  int *feld; // Das Array fuer die Deque Datenstruktur.
  int size;  // maximale Anzahl der Elemente im Deque.
  int left;  // Index vor(!) dem linken Ende.
  int right; // Index des rechten Endes.

public:
  deque (int n)
  { feld = new int[n + 1];
    size = n;
    left = right = 0;
  } // Konstruktor.

  void insertleft(int item);
  // Einfuegen am linken Ende.

  int removeleft( );
  // Entfernen am linken Ende.

  void insertright(int item);
  // Einfuegen am rechten Ende.

  int removeright();
  // Entfernen am rechten Ende.

  boolean empty( );
  // Stellt fest, ob das Deque leer ist.

  boolean full( );
  // Stellt fest, ob das Deque voll ist.
};

class queue : private deque {
```

```
public:
   queue (int n) : deque (n){}
   void enqueue(int item)
      { deque::insertleft (item); }

   int dequeue( )
      { return deque::removeright( ); }

   boolean empty( )
      { return deque::empty( );}

   boolean full( )
      { return deque::full( );}
};


class stack : private deque {
public:
   stack (int n) : deque (n) {}

   void push(int item)
      { deque::insertright(item);}

   int pop( )
      {return deque::removeright( ); }

   boolean empty( )
      { return deque::empty( );}

   boolean full( )
      { return deque::full( );}
};
```

Offensichtlich werden die vier Deque-Operationen in konstanter Zeit unterstützt. Gleiches gilt damit natürlich auch für die Stack- und Queue-Operationen.

Aufgabe 35
In dieser Aufgabe wollen wir geeignete Datenstrukturen für Mengen entwerfen.

(a) Gegeben sei eine feste Grundmenge $G = \{1, \ldots, n\}$. Unser Ziel ist der Entwurf einer Datenstruktur zur Repräsentation einer Teilmenge $M \subseteq G$. Diese Menge M wird, für $i \in G$, durch die Operationen *add(i)* (füge das Element i zu M hinzu) und *remove(i)* (entferne i aus M) beginnend mit der leeren Menge aufgebaut. Die zu erzeugende Datenstruktur soll zusätzlich die Operation *lookup(i)* (gehört i zur Menge M?) unterstützen. Die Anforderungen an die Datenstruktur sind:

1. Die Datenstruktur darf Speicherplatz $O(n)$ einnehmen.
2. Die Operationen *add(i), remove(i)* und *lookup(i)* laufen in Zeit $O(1)$.

(b) Diesmal sind zwei Mengen M_1 und M_2 (die gemäß obigen Operationen erzeugt werden) zu repräsentieren. Zusätzlich soll die Operation $M_1 = M_1 \cap M_2$ unterstützt werden. Die neuen Datenstrukturen sollen folgendes gewährleisten:

- Der Speicherplatzbedarf ist durch $O(|M_1| + |M_2|)$ beschränkt.
- Die Operationen *add(i,M_j)* (füge Element i zu Menge M_j hinzu), *remove(i,M_j)* (entferne Element i aus Menge M_j) und *lookup(i,M_j)* (gehört Element i zu Menge M_j?) sollen in Zeit $O(|M_j|)$ laufen.

- Die Operation $M_1 = M_1 \cap M_2$ soll in Zeit $O(|M_1| + |M_2|)$ unterstützt werden. **Beschreibe** die Implementierung der Durchschnittsoperation.

Begründe jeweils warum die Anforderungen von den gewählten Datenstrukturen erfüllt werden.

Aufgabe 36
Entwerfe eine Datenstruktur, die Zahlen speichert und die die Operationen `pop`, `push(x)` und `minfind` in **konstanter Zeit** unterstützt: Die Dauer einer jeden Operation darf also nicht von der Anzahl der schon in der Datenstruktur enthaltenen Zahlen abhängen. (Die Funktion `minfind` benennt den Wert der kleinsten gespeicherten Zahl, verändert aber die Menge der gespeicherten Zahlen nicht.)

Beachte, dass die Datenstruktur Elemente auch mehrfach enthalten kann.

Wir schließen mit einer Implementierung von Listen durch Arrays.

Beispiel 4.3 : Angenommen, wir möchten eine Liste implementieren, deren Zellen aus einem integer-Datenfeld und einem Vorwärtszeiger bestehen. Die Benutzung zweier integer-Arrays *Daten* und *Zeiger* bietet sich an: Wenn ein beliebiges Listenelement den „Index i erhält“, dann ist Daten$[i]$ der Wert des Listenelements und Zeiger$[i]$ stimmt mit dem Index des rechten Nachbarn überein. Es fehlt aber noch eine wesentliche Komponente, nämlich die Speicherverwaltung: Um keinen unnötigen Platz zu verbrauchen, benutzen wir eine zusätzliche Datenstruktur *Frei*.

Wir nehmen an, dass wir Listen von höchstens n Elementen simulieren. Demgemäß werden *Daten* und *Zeiger* als integer-Arrays mit n Zellen definiert. Zu Anfang der Simulation weisen wir $Frei$ die Indexmenge $0, \ldots, n-1$ zu.

Angenommen, ein Element mit Wert w ist in die Liste nach einem Element mit Index i einzufügen. Wir prüfen, ob Frei nicht-leer ist. Wenn ja, entnehmen wir einen freien Index j und setzen

$$\text{Daten}[j] = w; \quad \text{Zeiger}[j] = \text{Zeiger}[i]; \quad \text{Zeiger}[i] = j;$$

Angenommen, ein Element ist von der Liste zu entfernen. Wir nehmen an, dass wir den Index i des Vorgängers kennen. Wir fügen

$$j = \text{Zeiger}[i];$$

in die Datenstruktur Frei ein und setzen

$$\text{Zeiger}[i] = \text{Zeiger}[j];$$

Welche Datenstruktur ist für $Frei$ zu wählen? Eine Datenstruktur, die es erlaubt, Elemente einzufügen und zu entfernen: ein Stack, eine Queue oder ein Deque tut's!

Aufgabe 37
Für jedes der folgenden Probleme ist ein Algorithmus **und** eine geeignete Datenstruktur zu entwerfen.

(a) Zuerst betrachten wir das Problem der Addition von Polynomen. Die Eingabe besteht aus zwei Polynomen p und q. p (bzw. q) wird durch seine von 0 verschiedenen Koeffizienten repräsentiert; d.h. wenn

$$p(x) = p_{i_1}x^{i_1} + p_{i_2}x^{i_2} + \cdots + p_{i_n}x^{i_n} \text{ (bzw. } q(x) = q_{j_1}x^{j_1} + q_{j_2}x^{j_2} + \cdots + q_{j_m}x^{j_m})$$

mit $i_1 < i_2 < \cdots < i_n$ (bzw. $j_1 < j_2 < \cdots < j_m$), dann ist p (bzw. q) durch die Folge

$$(p_{i_1}, i_1), \ldots, (p_{i_n}, i_n) \text{ (bzw. } (q_{j_1}, j_1), \ldots, (p_{j_m}, j_m)) \text{ repräsentiert.}$$

Beschreibe ein Programm, das das Polynom $p+q$ so schnell wie möglich berechnet. Die Ausgabe sollte aus der Folge der von 0 verschiedenen Koeffizienten von $p+q$ bestehen. **Beschreibe** die benutzten Datenstrukturen sorgfältig.

(b) Wir beschreiben einen Sortieralgorithmus, der kleine Zahlen schnell sortiert. Der Einfachheit halber nehmen wir an, dass n natürliche Zahlen $x_1, \ldots, x_n$ mit $0 \leq x_i < n^3$ gegeben sind. Die Eingaben x_i liegen in n-ärer Darstellung vor, d.h. jede Eingabe besitzt 3 Stellen $x_i = z_{i,2}\ z_{i,1}\ z_{i,0}$ mit den Ziffern $z_{i,0}, z_{i,1}, z_{i,2} \in \{0, \ldots n-1\}$.

Der Sortieralgorithmus besteht in diesem Fall aus drei Phasen. In der ersten Phase werden die Eingaben *gemäß ihrer niedrigst-wertigen Stelle* sortiert. Dieses Sortierproblem ist einfach: Die Eingabe x_i (mit niedrigst-wertiger Stelle $z_{i,0} = j$) wird in die Zelle j einer geeigneten Datenstruktur (mit Zellen $0, ..., n-1$) eingetragen. Nachdem alle Eingaben eingetragen wurden, erhält man die (nach der niedrigst-wertigen Stelle) sortierte Eingabefolge, wenn man die Datenstruktur von „links nach rechts“ ausgibt.

In der zweiten Phase wird das obige Verfahren wiederholt, aber diesmal für die „mittel-wertige“ Stelle. Das Gleiche passiert in Phase 3, aber jetzt mit der höchst-wertigen Stelle.

Beschreibe die Details dieses Algorithmus und **entwirf** die dazugehörenden Datenstrukturen. Die Wahl der Datenstrukturen muss erreichen, dass die dritte Phase die Sortier-Ergebnisse der beiden ersten Phasen ausnutzen kann: Dies ist notwendig bei zwei Eingaben mit identischen höchst-wertigen Stellen.

Die Implementation sollte zu einer schnellst-möglichen Laufzeit führen. Laufzeit $O(n)$ ist erreichbar.

(c) Im letzten Problem muss entschieden werden, ob ein vorgelegter Klammerausdruck –mit den Klammern ( ) [ ] – legal ist. **Entwirf** einen schnellen Algorithmus und **beschreibe** die dazugehörende Datenstruktur.

4.3 Bäume

Gewurzelte Bäume haben wir in Abschnitt 2.2 eingeführt.

Zur Erinnerung: Ein gewurzelter Baum T besteht aus einer endlichen **Knotenmenge** V und einer **Kantenmenge** $E \subseteq V \times V$. Die gerichtete Kante (i, j) führt **von** i **nach** j. Wir fordern, dass T genau einen Knoten r besitzt, in den keine Kante hineinführt. r wird als die **Wurzel** von T bezeichnet. Eine weitere Forderung ist, dass in jeden anderen Knoten genau eine Kante hineinführt und, dass jeder Knoten von der Wurzel aus erreichbar ist.

Falls für Knoten v die Kante (v, w) vorhanden ist, nennen wir v den **Elternknoten** von w und sagen, dass w ein **Kind** von v ist. Die anderen Kinder von v sind die **Geschwister von** w. Gibt es einen Weg von Knoten u nach Knoten v, dann ist u **Vorfahre** von v und v **Nachfahre** von u. **Tiefe** (v) ist die Länge des (eindeutigen) Weges von r nach v, **Höhe** (v) die Länge des längsten Weges von v zu einem Blatt.

T heißt **geordnet**, falls die Kinder eines jeden Knotens geordnet sind. Zum Beispiel im Fall binärer Bäume können wir dann über das „linke“, bzw. „rechte“ Kind eines Elternknotens reden.

Gewurzelte Bäume sind ein wesentliches Hilfsmittel zur hierarchischen Strukturierung von Daten. Wir benutzen sie für konzeptionelle Darstellungen (z. B. für die von einem Benutzer angelegten Verzeichnisse, als Nachfahrenbaum oder zur Veranschaulichung rekursiver Programme) oder als Datenstrukturen zur Unterstützung von Algorithmen. Die Datenstruktur *gewurzelter Baum* wird zum Beispiel in der Auswertung von arithmetischen Ausdrücken oder in der Syntaxerkennung mit Hilfe von Syntaxbäumen benutzt. Suchprobleme ergeben ein weiteres Einsatzgebiet von gewurzelten Bäumen: Dies ist nicht überraschend, da (binäre) Bäume in natürlicher Weise die Binärsuche unterstützen.

Ab jetzt sprechen wir nur noch von Bäumen und meinen damit stets gewurzelte Bäume.

Aufgabe 38
Schreibe eine nicht-rekursive Funktion in Pseudocode, die die Anzahl der Blätter von T berechnet und **bestimme** die Laufzeit (in O-Notation) für binäre Bäume mit n Knoten.

Wiederum sollte die Funktion so effizient wie möglich sein: Laufzeit $O(n)$ ist auch diesmal erreichbar. Die Funktionen *push* und *pop* einer Klasse *stack* dürfen benutzt werden.

Zuerst müssen wir entscheiden, welche Operationen ein Baum (als Datenstruktur) unterstützen sollte. Für einen Baum T und Knoten v haben sich die folgenden Operationen als wichtig herausgestellt:

(1) `Wurzel` : Bestimme die Wurzel von T.

(2) `Eltern`(v) : Bestimme den Elternknoten des Knotens v in T. Wenn $v = r$, dann ist der Null-Zeiger auszugeben.

(3) `Kinder`(v) : Bestimme die Kinder von v. Wenn v ein Blatt ist, dann ist der Null-Zeiger als Antwort zu geben.

(4) Für binäre (also 2-äre) geordnete Bäume sind die beiden folgenden Operationen von Interesse:

(4a) `LKind`(v) : Bestimme das linke Kind von v.

(4b) `RKind`(v) : Bestimme das rechte Kind von v. Sollte das entsprechende Kind nicht existieren, ist der Null-Zeiger als Antwort zu geben.

(5) `Tiefe`(v) : Bestimme die Tiefe von v.

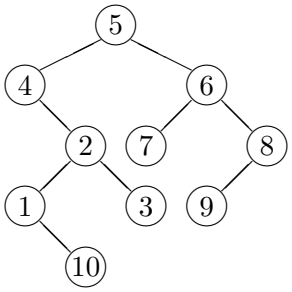
(6) `Höhe`(v) : Bestimme die Höhe von v.

(7) `Baum`$(v, T_1, \ldots, T_m)$: Erzeuge einen geordneten Baum mit Wurzel v und Teilbäumen $T_1, \ldots, T_m$. (Diese Operation erlaubt es uns, Bäume zu „bauen“.)

(8) `Suche`(x) : Bestimme alle Knoten, die den Wert x besitzen.
(Diese Operation ist natürlich nur dann sinnvoll, wenn wir annehmen, dass ein jeder Knoten, neben den Baum-Zeigern, auch noch einen Wert speichert.)

4.3.1 Baum-Implementierungen

Wir vergleichen jetzt verschiedene Baum-Implementierungen in Hinsicht auf ihre Speicher-Effizienz und in Hinsicht auf eine effiziente Unterstützung der obigen Operationen. Wir werden als durchgehendes Beispiel den folgenden Baum wählen:



Das Eltern-Array.
Wir nehmen an, dass jeder Knoten eine Zahl aus der Menge $\{1, \ldots, n\}$ als Namen besitzt und, dass es zu jedem $i \in \{1, \ldots, n\}$ genau einen Knoten mit Namen i gibt.

Ein integer-Array *Baum* speichert für jeden Knoten v den Namen des Elternknotens von v: Es ist also Baum[v]= Name des Elternknotens von v. Wenn $v = r$, dann setzen wir Baum $[v] = r$.

Knotennr.	1	2	3	4	5	6	7	8	9	10
Elternknoten	2	4	2	5	0	5	6	6	8	1

Diese Implementierung ist maßgeschneidert für Operation (2): nur konstante Zeit wird benötigt. Operation (5) gelingt in Zeit $O(\text{Tiefe }(v))$ (warum?), ein respektables Ergebnis. Um Operation (1) zu realisieren, können wir entweder das Array nach dem Null-Eintrag in der Elternzeile durchsuchen, oder man beginnt irgendwo und hangelt sich von Eltern- zu Elternknoten nach oben. Ersteres benötigt $O(n)$ viel Zeit, wenn n die Anzahl der Knoten ist, die zweite Variante führt in $O(Tiefe(T))$ zum Erfolg. Da die Tiefe des Baumes durch die Anzahl der Knoten nach oben beschränkt ist, ist diese Alternative vorzuziehen. Es ist natürlich auch möglich die Implementierung derart zu erweitern, dass in einer separaten Variable der Name der Wurzel abgelegt wird.

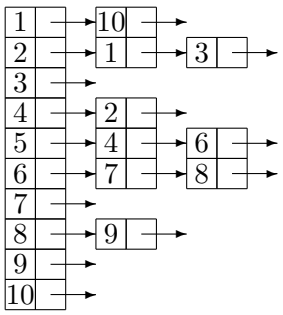
Die Operationen (3), (4) und (6) erzwingen ein Absteigen von v (in Richtung Blätter) und unsere Datenstruktur versagt: Mindestens lineare Laufzeit (linear in der Anzahl der Knoten) ist erforderlich.

Operation (7) ist Zeit-aufwändig, wenn die Bäume in anderen Arrays gespeichert sind. Operation (8) benötigt, wie auch für die übrigen Baum-Implementierungen, ein Durchsuchen des ganzen Baumes.

Diese Implementierung ist sicherlich nicht erste Wahl, wenn viele der obigen Operationen anzuwenden sind. Sie ist die erste Wahl für die Operationenmenge {Elternknoten, Tiefe}. Der Grund ist die Speicherplatz-Effizienz: Ein Baum mit n Knoten kann durch ein Array mit n Zellen gespeichert werden.

Die Adjazenzlisten-Implementierung.

Ein „Kopf-Array“ Head mit den Zellen $1, \ldots, n$ wird für Bäume mit n Knoten zur Verfügung gestellt. Head$[i]$ ist ein Zeiger auf die Liste der Kinder von i.



Die Operationen „Wurzel, Kinder, LKind, RKind“ können in konstanter Zeit durchgeführt werden, wenn wir zusätzlich eine Variable zur Verfügung stellen, die den Namen der Wurzel speichert. Für Operation 7 haben wir dieselben Schwierigkeiten wie in der Array-Implementierung. „Höhe (v, T)“ wird auch angemessen unterstützt mit der Laufzeit

$$O(\text{Anzahl der Knoten im Teilbaum mit Wurzel } v).$$

(Warum?) Die Datenstruktur versagt aber für die Operationen „Elternknoten“ und „Tiefe“, denn der ganze Baum muss durchlaufen werden.

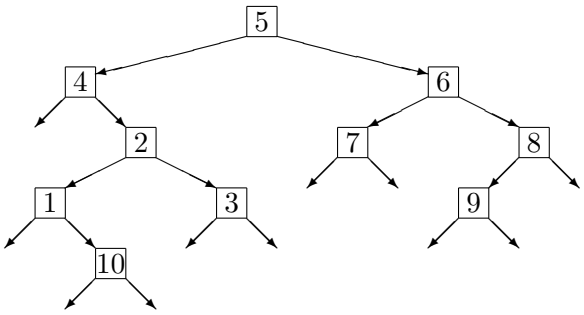
Für einen Baum mit n Knoten benötigen wir $2n-1$ Zeiger (einen Zeiger für jede der n Zellen von Head und einen Zeiger für jede der $n-1$ Kanten) und $2n-1$ Zellen (n Zellen für das

Array Head und $n-1$ Zellen für die Kanten). Der Speicherbedarf, verglichen mit der Array-Implementierung, ist also relativ stark angewachsen.
Durch die Bereitstellung eines Eltern-Zeigers können wir die Eltern-Operation in konstanter Zeit und die Operation Tiefe (v) in linearer Zeit $O(\text{Tiefe}(v))$ unterstützen. Doch zahlen wir dann mit einem zusätzlichen Speicheraufwand von n Zeigern.

Die Binärbaum-Implementierung.
Wir deklarieren einen Binärbaum wie folgt: Ein Knoten wird durch die Struktur

```
typedef struct Knoten{
  int wert;
  Knoten *links, *rechts;};
```

dargestellt. Wenn der Zeiger z auf die Struktur des Knotens v zeigt, dann ist $z \to wert$ der *Wert* von v und $z \to links$ (bzw. $z \to rechts$) zeigt auf die Struktur des linken (bzw. rechten) Kindes von v. Der Zeiger *wurzel* zeigt stets auf die Struktur der Wurzel des Baums.

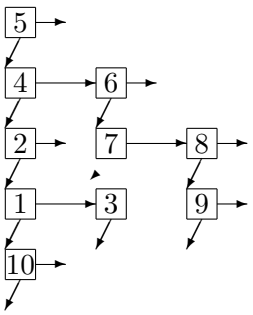


Im Laufzeitverhalten schneidet diese Implementierung ähnlich ab wie die Adjazenzlisten-Implementierung für binäre Bäume. Aber diese Implementierung hat eine größere Speichereffizienz: $2n$ Zeiger (zwei Zeiger pro Knoten) und nur n Zellen werden benötigt.

Diese Implementierung ist erste Wahl für allgemeine binäre Bäume. Durch zusätzliche Verwendung eines Eltern-Feldes kann auch die Schwäche bezüglich der Operationen „Elternknoten“ und „Tiefe“ ausgeglichen werden.

Aufgabe 39
Wir betrachten die Binärbaum-Implementierung. **Schreibe** eine Funktion in Pseudocode, die die Tiefe eines binären Baumes T berechnet (also die Länge des längsten Weges von der Wurzel zu einem Blatt) und **bestimme** die Laufzeit (in O-Notation) für binäre Bäume mit n Knoten.

Die Kind-Geschwister Implementierung.
Diese Implementierung funktioniert für allgemeine Bäume. Wir stellen jetzt für jeden Knoten v eine Struktur zur Verfügung, die aus 2 Zeigern besteht: einem Zeiger auf das erste (linkeste) Kind und einem Zeiger auf den nächsten Geschwister-Knoten.



Im Laufzeitverhalten unterscheidet sich diese Implementierung kaum von seinem Kontrahenten, der Adjazenzlisten-Implementierung. Der Unterschied zeigt sich im Speicherverhalten: Es werden nur $2n$ Zeiger und n Zellen benötigt!

Diese Implementierung ist erste Wahl für allgemeine Bäume. (Warum haben wir den Verlierer Adjazenzliste betrachtet? Dieser Verlierer wird sich als großer Gewinner bei der Darstellung von *Graphen* herausstellen.)

Aufgabe 40
Sei v ein Knoten in einem Baum. Dann nennen wir $Tiefe(v)$ die Länge des Weges von der Wurzel zu v und $Höhe(v)$ die Länge des längsten Weges von v zu einem Blatt im Teilbaum mit Wurzel v. Die Länge eines Weges messen wir stets mit der Zahl der Kanten entlang des Weges. Als Durchmesser eines Baumes T bezeichnen wir die Länge des längsten *ungerichteten* Weges zwischen zwei Knoten v und w aus T.

Ein Baum T sei in Kind-Geschwister Darstellung gegeben.

(a) **Beschreibe** einen Algorithmus, der jeden Knoten aus v zusammen mit seiner Höhe ausgibt.

(b) **Beschreibe** einen Algorithmus, der den Wert des Durchmessers des Baumes ausgibt.

Wichtig:

- Beide Algorithmen sollen in Zeit $O(|T|)$ laufen, wobei $|T|$ die Zahl der Knoten im Baum T beschreibt.
- Zunächst ist die prinzipielle Vorgehensweise des Algorithmus darzustellen. Erst danach soll der Algorithmus strukturiert im Pseudocode notiert werden. Anschließend sind jeweils Korrektheit und das Einhalten der Laufzeitschranke zu belegen.

4.3.2 Suche in Bäumen

In der obigen Diskussion der verschiedenen Implementierungen haben wir die Operation Suche (x) im wesentlichen ausgeklammert. Keiner der obigen Implementierungen wird eine „superschnelle“ Unterstützung des Suchproblems gelingen, doch eine zufriedenstellende Unterstützung wird möglich sein. Aber wie durchsucht man Bäume?

Definition 4.1 Sei T ein geordneter Baum mit Wurzel r und Teilbäumen $T_1, \ldots, T_m$.

(a) Wenn T in **Postorder** durchlaufen wird, dann werden (rekursiv) die Teilbäume $T_1, \ldots, T_m$ nacheinander durchlaufen und danach wird die Wurzel r besucht.

(b) Wenn T in **Präorder** durchlaufen wird, dann wird zuerst r besucht und dann werden die Teilbäume $T_1, \ldots, T_m$ (rekursiv) durchlaufen.

(c) Wenn T in **Inorder** durchlaufen wird, wird zuerst T_1 (rekursiv) durchlaufen, sodann wird die Wurzel r besucht und letztlich werden die Teilbäume $T_2, \ldots, T_m$ (rekursiv) durchlaufen.

Aufgabe 41
Gegeben seien ein Präorder- und ein Inorder-Durchlauf durch einen binären Baum, dessen n Knoten mit paarweise verschiedenen Zahlen beschriftet sind. Diese Folgen seien in je einem Array abgespeichert.

(a) Zeige, daß der Baum durch die oben beschriebenen Folgen eindeutig definiert ist. Gib dazu einen Algorithmus an, der den Baum rekonstruiert, und analysiere die Laufzeit.

(b) Die Namen der Knoten seien aus dem Bereich $\{1, \ldots, n\}$ gewählt. **Gib** einen Linearzeitalgorithmus für Teil (a) an.

Postorder, Präorder und Inorder sind auch als Nebenreihenfolge, Hauptreihenfolge und symmetrische Reihenfolge bekannt. Die elementarste Implementierung geschieht mittels Rekursion: Die obige rekursive Definition lässt sich direkt in ein rekursives Programm umsetzen:

Algorithmus 4.1 **(Eine rekursive Implementierung von Präorder)**

```
void praeorder (Knoten *p)
   {Knoten *q;
    if (p != null)
      {cout << p->wert;
       for (q=p->LKind; q != null; q=q->RGeschwister)
         praeorder(q); } }
```

Um die Rekursion zu verstehen, betrachten wir einen beliebigen Knoten v. Welcher Knoten wird direkt nach v besucht? In der Postorder-Reihenfolge ist dies das linkeste Blatt im rechten Nachbarbaum. Wenn v keinen rechten Geschwisterknoten besitzt, wird der Elternknoten von v als nächster besucht. In der Präorder-Reihenfolge wird, falls vorhanden, das linkeste Kind von v besucht, während in der Inorder-Reihenfolge der linkeste Knoten im zweiten Teilbaum (falls vorhanden) „als nächster dran“ ist.

Wenn wir zusätzliche Zeiger gemäß dieser Reihenfolge einsetzen, können wir somit eine Rekursion (deren Implementierung zusätzlichen Verwaltungsaufwand bedeutet) vermeiden: Die jeweilige Durchlaufreihenfolge erhalten wir durch Verfolgung der neuen Zeiger. Allerdings bezahlen wir mit zusätzlichem Speicheraufwand. Die jeweiligen Datenstrukturen werden respektive als **Postorder-**, **Präorder-** oder **Inorder-gefädelte** Bäume bezeichnet.

Aufgabe 42
In der Binärbaum-Implementierung besitzen Knoten mit höchstens einem Kind mindestens einen Zeiger auf $NULL$. Wir wollen diese Zeiger benutzen, um *inorder*-gefädelte binäre Bäume zu bilden. Um „Baum“-Zeiger von „Fädelungs“-Zeigern unterscheiden zu können, wird jeder Zeiger durch einen Booleschen Parameter gekennzeichent: 0 kennzeichent einen Fädelungs-Zeiger und 1 einen Baum-Zeiger.
z zeige auf die Struktur des Knotens v. Dann zeigt $z \to links$ ($z \to rechts$) auf die Struktur des linken (rechten) Kindes, falls dieses Kind vorhanden ist. Falls ein linkes (rechtes) Kind nicht existiert, zeigt $z \to links$ ($z \to rechts$) auf die Struktur des direkten Vorgängers (Nachfolgers) in der Inorder-Numerierung (dies sind Fädelungs-Zeiger). Für den ersten (letzten) Knoten in Inorder-Numerierung gibt es keinen Inorder-Vorgänger (Inorder-Nachfolger): Diese Zeiger bleiben Zeiger auf $NULL$.

(a) Beschreibe einen Algorithmus zur Konstruktion einer Inorder-Fädelung für einen binären Baum in Binärbaum-Implementierung (Laufzeitanalyse nicht vergessen).

(b) Beschreibe einen nicht-rekursiven Algorithmus, der einen Inorder-Durchlauf (Ausgabe der Knoten in Inorder-Reihenfolge) für einen *inorder*-gefädelten Baum in Zeit $O(n)$ mit möglichst wenig zusätzlichem Speicherplatz durchführt.

Wir können eine Rekursion aber natürlich auch durch eine Verwendung eines Stacks vermeiden. Als ein Beispiel betrachten wir die Präorder-Reihenfolge für Bäume, die eine Kind-Geschwister Implementierung besitzen. Wir nehmen an, dass der Teilbaum mit Wurzel v in der Präorder-Reihenfolge zu durchlaufen ist.

Algorithmus 4.2 **(Eine nicht-rekursive Implementierung von Präorder)**

(1) Wir fügen v (genauer, einen Zeiger auf die Struktur von v) in einen anfänglich leeren Stack ein.

(2) Solange der Stack nicht-leer ist, wiederhole:

   (a) Entferne das erste Stack-Element mit Hilfe der Pop-Operation. Dies sei das Element w. w wird besucht.

   (b) Die Kinder von w werden in umgekehrter Reihenfolge in den Stack eingefügt; natürlich füge nur tatsächlich existierende Kinder ein.
   /∗ Durch die Umkehrung der Reihenfolge werden die Bäume später in ihrer natürlichen Reihenfolge abgearbeitet. ∗ /

Angenommen, ein Baum mit n Knoten liegt in Kind-Geschwister-Implementierung vor. Was ist die Laufzeit der obigen nicht-rekursiven Präorder-Prozedur? Die Laufzeit ist linear (= $O(n)$), denn jeder Knoten wird genau einmal in den Stack eingefügt. Insgesamt werden also höchstens $O(n)$ Stackoperationen durchgeführt. Die lineare Laufzeit folgt, da Stackoperationen die Laufzeit dominieren.

Ebenfalls lineare Laufzeiten erreichen wir auch für die rekursive Version der Präorder-Reihenfolge: Die Prozedur wird für jeden Knoten w genau einmal aufgerufen und das Abarbeiten der nicht-rekursiven Befehle gelingt in Zeit $O(\text{Kinder}(w))$, wobei $\text{Kinder}(w)$ die Anzahl der Kinder von w bezeichnet. Die insgesamt benötigte Zeit ist somit durch

$$O\left(\sum_{v \in V} (1 + Kinder\,(v))\right) = O(|V| + |E|) = O(2|V| - 1) = O(n)$$

beschränkt. Wir werden analoge Ergebnisse auch für die Postorder- und Inorder-Reihenfolge erhalten. Zusammengefaßt:

Satz 4.1 *Für einen Baum mit n Knoten, der in*

Kind-Geschwister-Darstellung, in Adjazenzlisten- oder Binärbaum-Darstellung

vorliegt, kann die

Präorder-Reihenfolge, die Postorder-Reihenfolge und die Inorder-Reihenfolge

in Zeit $O(n)$ berechnet werden.

Aufgabe 43

In dieser Aufgabe betrachten wir *einfache* arithmetische Ausdrücke auf den zweistelligen Operationen + und ∗. Ein Beispiel für einen solchen Ausdruck ist:

$$(3 + 4 * 6) + ((5 * 2 + 1) * 7).$$

Zunächst ordnen wir den Ausdruck so um, dass jeder Operator **vor** seinen Operanden auftritt: ein Ausdruck A in *Präfix Notation* ist entweder eine Zahl oder hat die Form $+BC$ oder $*BC$ für Ausdrücke B, C in Präfix Notation.

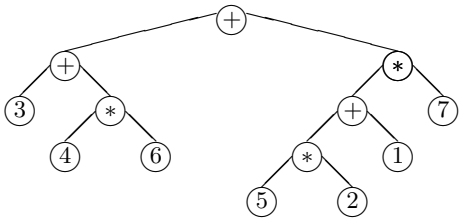
Dem obigen Beispiel entspricht dann der Ausdruck:

$$+ + 3 * 4\,6 * + * 5\,2\,1\,7.$$

Die Präfix-Notation benötigt also keine Klammern.

Arithmetische Ausdrücke können auch als Syntaxbäume dargestellt werden. Der Syntaxbaum für einen arithmetischen Ausdruck wird durch die folgende rekursive Regel definiert: Der erste Operator wird an die Wurzel

gesetzt. Der Baum für den Ausdruck, der dem ersten Operanden entspricht, wird als linker Teilbaum eingehängt, und der Baum, der dem zweiten Operanden entspricht, wird als rechter Teilbaum eingehängt. Der Syntaxbaum für das obige Beispiel hat also die folgende Form:



Gegeben sei ein einfacher arithmetischer Ausdruck in Präfix-Notation. **Beschreibe** einen Algorithmus, der einen Syntaxbaum für diesen Ausdruck aufbaut und bestimme seine Laufzeit.

4.4 Graphen

Wir haben Graphen in Abschnitt 2.2 eingeführt. Zur Erinnerung: Ein **ungerichteter Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge $E \subseteq \{\{i, j\} : i, j \in V, i \neq j\}$. Die Kanten eines ungerichteten Graphen werden also als Paarmengen repräsentiert.

Ein **gerichteter Graph** $G = (V, E)$ besteht ebenfalls aus einer endlichen Knotenmenge V und einer Kantenmenge $E \subseteq \{(i, j) : i, j \in V, i \neq j\}$. Die Kanten eines gerichteten Graphen werden also als geordnete Paare dargestellt.

Hier sind einige erste wichtige Anwendungen.

(a) Der Graph des World-Wide-Web ist ein *gerichteter* Graph. Dazu verwende die Webseiten als Knoten und setzen eine Kante von Webseite w_1 nach Webseite w_2 ein, wenn die Webseite w_1 einen Hyperlink auf die Webseite w_2 besitzt.

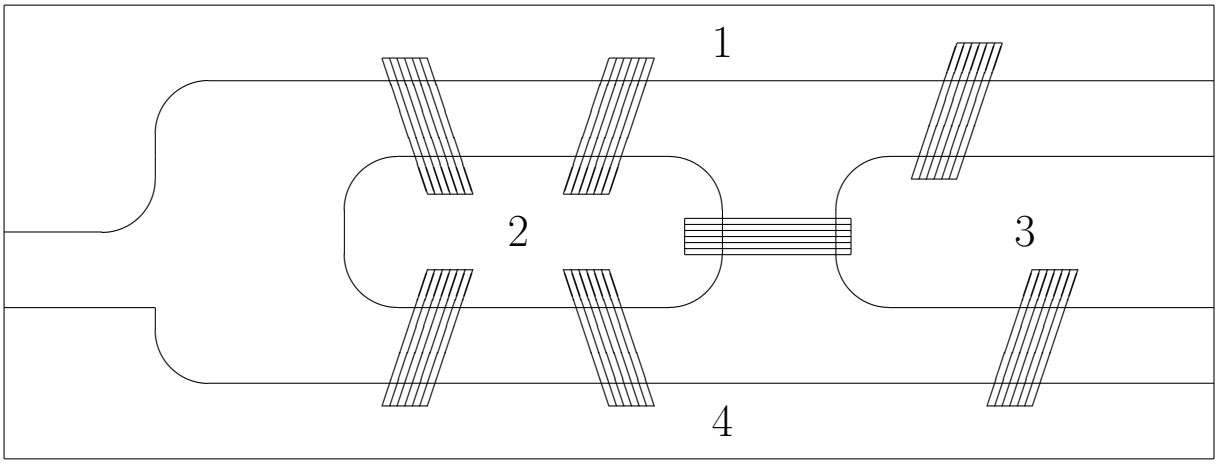
(b) **Kürzeste-Wege-Probleme**:

- Das Streckennetz der deutschen Bahn entspricht einem *ungerichteten* Graphen. Die Knoten des „Bahn-Graphen“ entsprechen den Bahnhöfen und Kanten verbinden ohne Zwischenstop erreichbare Bahnhöfe. Bei der Erstellung von Reiseplänen handelt es sich somit um das graph-theoretische Problem der Konstruktion kürzester (gewichteter) Wege zwischen einem Start- und einem Zielbahnhof. Verschiedenste Metriken werden benutzt wie etwa
  - die Gesamtfahrzeit,
  - die Kosten der Fahrkarte oder
  - die Häufigkeit des Umsteigens.
- Auch ein Routenplaner muss kürzeste Wege zwischen einem Anfangs- und einem Zielpunkt bestimmen. Dazu wird eine detaillierte Straßenkarte in einen *gerichteten* Graphen umgewandelt:
  - Kreuzungspunkte, bzw. Abfahrten entsprechen den Knoten,
  - Straßenabschnitte entsprechen Kanten, wobei jede Kante mit der Zeit zum Durchfahren markiert wird.

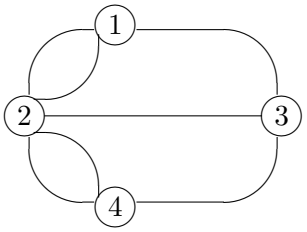
(c) Ein relativ einfaches Problem ist das **Labyrinth-Problem**: Ein Labyrinth ist als *ungerichteter* Graph beschrieben und die Knoten s und t sind gegeben. Es ist zu entscheiden, ob es einen Weg von s nach t gibt, und wenn ja, ist ein solcher Weg zu bestimmen.

(d) Ein ähnlich, doch weitaus schwieriger zu lösendes Problem ist das des Handlungsreisenden, der alle seine Kunden mit einer Rundreise kleinstmöglicher Länge besuchen möchte. Hier entsprechen die Knoten den Kunden und die • Kanten den Direktverbindungen zwischen zwei Kunden. Diese Fragestellung ist auch als das **Traveling Salesman Problem** bekannt.

Euler hat 1736 das *Königsberger Brückenproblem* gelöst. Dabei geht es darum, einen Rundweg durch Königsberg zu konstruieren, so dass jede Brücke über die Pregel genau einmal überquert wird.

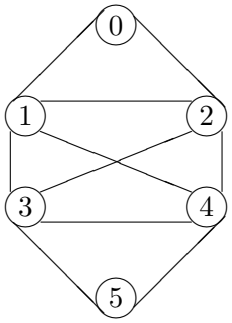


Die entsprechende graph-theoretische Abstraktion führt für jede der sieben Brücken eine Kante ein. Diese Kanten (oder Brücken) verbinden vier Knoten, die genau den vier Stadtteilen entsprechen.



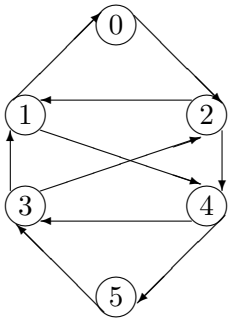
(Beachte, dass ungerichtete Graphen keine Schleifen besitzen. Weder gerichtete noch ungerichtete Graphen besitzen Mehrfachkanten.) Der „Graph“ des Königsberger Brückenproblems ist also in unserem Sinne kein Graph.) Das Königsberger Brückenproblem lautet jetzt: Finde einen Weg, der jede Kante genau einmal durchläuft und an seinem Anfangspunkt endet, einen sogenannten **Euler-Kreis**. Der obige Graph hat keinen Euler-Kreis, denn die Existenz eines Euler-Kreises bedingt, dass jeder Knoten eine gerade Anzahl von inzidenten Kanten besitzt: Das Königsberger Brückenproblem ist nicht lösbar!

Beispiel 4.4



$$G = (V, E) \text{ mit } V = \{0, 1, 2, 3, 4, 5\} \text{ und}$$
$$E = \{\{1, 0\}, \{1, 2\}, \{1, 4\}, \{1, 3\}, \{2, 0\}, \{2, 4\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$$

Dieser Graph besitzt einen Euler-Kreis (welchen?). Der gerichtete Graph $G = (V, E)$



mit $V = \{0, 1, 2, 3, 4, 5\}$ und $E = \{(0, 2), (1, 0), (1, 4), (2, 1), (2, 4), (3, 2), (3, 1), (4, 3), (4, 5), (5, 3)\}$ besitzt ebenso einen gerichteten Euler-Kreis (welchen?).

Als ein vorläufig letztes Beispiel sei das **Zuordnungs**- oder **Matching-Problem** angesprochen. Im Zuordnungsproblem haben wir eine Menge von Personen mit unterschiedlichen Qualifikationsgraden zur Ausführung einer bestimmten Menge von Tätigkeiten. Wir erfinden einen bipartiten Graphen, der Knoten sowohl für die Personen wie auch für die Tätigkeiten besitzt; die Kante zwischen einer Person p und einer Tätigkeit t erhält den Qualifikationsgrad von p für t als Gewicht. Wir haben eine optimale Zuweisung von Tätigkeit zu Person zu finden, so dass der Gesamtqualifkationsgrad so hoch wie möglich ist: Die Zuweisung besteht aus einem **Matching**, also einer Menge von Endpunkt disjunkten Kanten, der Gesamtqualifikationsgrad ist die Summe der Kantengewichte. In der graphtheoretischen Formulierung ist das Zuordnungsproblem als das bipartite Matching Problem bekannt.

Weitere Anwendungen finden Graphen im Entwurf von Schaltkreisen und Netzwerken: die Architektur eines Schaltkreises oder Netzwerks ist ein Graph!

Wir nehmen im folgenden stets an, dass $V = \{0, ...n-1\}$. Dies bedeutet keine Einschränkung der Allgemeinheit, da wir Knoten ja umbenennen können.

4.4.1 Topologisches Sortieren

Wir lernen jetzt an dem Beispiel des **topologischen Sortierens**, wie wir aus den bisherigen elementaren Datenstrukturen eine recht mächtige Datenstruktur bauen können.

Gegeben sind n Aufgaben $a_0, \ldots, a_{n-1}$ und Prioritäten zwischen den einzelnen Aufgaben. Die Prioritäten sind in der Form von p Paaren $P_1, \ldots, P_p \in \{(i, j) : 0 \leq i, j < n, i \neq j\}$ gegeben.

Die Priorität (k, l) impliziert, dass Aufgabe a_k auszuführen ist, bevor Aufgabe a_l ausgeführt werden kann. Das **Ziel** ist die Erstellung einer Reihenfolge, in der alle Aufgaben ausgeführt werden können, respektive festzustellen, dass eine solche Reihenfolge nicht existiert.

Natürlich ist hier ein typisches graph-theoretisches Problem versteckt. Dazu wählen wir $V = \{0, \ldots, n-1\}$ als Knotenmenge und verabreden, dass Knoten i der Aufgabe a_i entspricht. Wir setzen genau dann eine Kante von i nach j ein, wenn das Paar (i, j) als eine der p Priorität vorkommt.

Wann können wir eine Aufgabe a_j ausführen? Aufgabe a_j muss die Eigenschaft haben, dass keine andere Aufgabe a_i *vor* a_j auszuführen ist: Es gibt also *keine* Kante (i, j) von einem Knoten i in den Knoten j: Knoten j ist somit eine Quelle, d.h. es gilt In-Grad$(j) = 0$. Wenn j gefunden ist, können wir unsere Vorgehensweise wiederholen: Wir entfernen j mit allen inzidenten Kanten und suchen eine ausführbare Aufgabe unter den verbleibenden Aufgaben.

Welche Datenstrukturen sollten wir für unseren Algorithmus wählen? Zuerst verfolgen wir einen naiven Ansatz: Wir verketten alle p Kanten in einer Liste „Priorität“ und benutzen ein integer Array „Reihenfolge“ sowie zwei boolesche Arrays „Erster“ und „Fertig“ mit jeweils n Zellen.

Algorithmus 4.3 **(Eine naive Implementierung)**

Setze Zaehler= 0 und Fertig$[i]$ = falsch für $1 \leq i \leq n$.

Wiederhole n-mal:

(0) Setze jede Zelle Erster$[i]$ auf wahr, wenn die entsprechende Zelle Fertig$[i]$ auf falsch gesetzt ist. Sonst setze Erster$[i]$ auf falsch.

(1) Durchlaufe die Prioritätsliste. Wenn Kante (i, j) angetroffen wird, setze

$$\text{Erster}[j] = \text{falsch}.$$

(2) Nachdem die Prioritätsliste abgearbeitet ist, durchlaufe das Array Erster. Wenn Erster $[j]$ = falsch, gehe zur Nachbarzelle. Wenn Erster$[j]$ = wahr, dann führe die folgenden Schritte durch:

   (a) Reihenfolge[Zähler++] $= j$.
   (Aufgabe j wird ausgeführt.)

   (b) Durchlaufe die Prioritätsliste und entferne jede Kante (j, k), die in j startet.
   (Aufgabe a_j behindert nicht mehr eine Ausführung von Aufgabe a_k.)

   (c) Setze Fertig$[j]$ auf wahr.

Laufzeitanalyse:
In Schritt (0) und (1) der Hauptschleife wird jedesmal das Array Erster initialisiert (n Schritte) und die Prioritätsliste durchlaufen ($O(p)$ Schritte).
In jeder Iteration der Hauptschleife wird mindestens eine ausführbare Aufgabe gefunden und somit wird die Hauptschleife höchstens n-mal durchlaufen. Wir haben also, *abgesehen* von der Behandlung der ausführbaren Aufgaben, $O(n \cdot (p + n))$ Schritte ausgeführt. Für jede ausführbare Aufgabe entstehen die wesentlichen Kosten bei dem Durchlaufen der Prioritätsliste. Da die Prioritätsliste für alle n Aufgaben durchlaufen wird, „bezahlen“ wir mit einer Laufzeit von $O(n \cdot p)$. Insgesamt wird diese Implementierung also in Zeit

$$O(n(p + n))$$

laufen. Wie erreichen wir eine Verbesserung?

Zuerst fällt auf, dass das vollständige Durchlaufen der Prioritätsliste beim Auffinden einer ausführbaren Aufgabe umgangen werden kann, wenn wir für jede Aufgabe a_j die Aufgaben aufführen, die erst nach a_j aufgeführt werden dürfen. Dazu ersetzen wir die Prioritätsliste durch eine **Adjazenzliste**: Diese Adjazenzliste ist als ein Array *Priorität* von n Zellen implementiert. Priorität $[i]$ ist dabei ein Zeiger auf eine Liste L_i, in der alle Aufgaben der Form

$$\{j : (i,j) \text{ ist eine Kante } \}$$

verkettet sind. Also erfasst L_i alle Aufgaben a_j, die nur *nach* Aufgabe i ausgeführt werden dürfen.

Mit Hilfe der Adjazenzliste haben wir also die Prioritätsliste unterteilt. Eine zweite wesentliche Beobachtung ist jetzt, dass Aufgabe a_k nur dann eine *neue* ausführbare Aufgabe werden kann, wenn für eine gerade ausgeführte Aufgabe a_j eine Kante (j,k) entfernt wird **und** wenn Aufgabe a_j zu diesem Zeitpunkt die einzige vor a_k auszuführende Aufgabe ist, d.h. wenn In-Grad$(k) = 1$ gilt.

Aber wie können wir feststellen, dass a_j die einzige Aufgabe war, die eine Ausführung von a_k verhinderte? Wir ändern unsere Datenstruktur und ersetzen das boolesche Array Erster durch ein integer Array „In-Grad“ und führen zuerst eine einmalige Vorbereitungsphase durch:

Algorithmus 4.4 **(Vorbereitungsphase)**

(1) Für $i = 1, \ldots, n$ setze
In-Grad$[i] = 0$;

(2) Durchlaufe die Adjazenzliste Priorität. Wenn die Kante (i,j) angetroffen wird, setze In-Grad$[j]++$.

/* Nach diesen Schritten haben wir also für jede Aufgabe a_j die Anzahl der Aufgaben a_i berechnet, die vor a_j ausgeführt werden müssen. Wir können jetzt die ausführbaren Aufgaben bestimmen: Dies sind alle Aufgaben a_i mit In-Grad$[i] = 0$. Wir legen alle diese Aufgaben in einer Queue mit Namen *Schlange* ab. (Schlange ist zu Anfang leer). */

(3) Für $i = 1, \ldots, n$
wenn In-Grad$[i] == 0$, dann stopfe i in *Schlange*.

Damit ist die Vorbereitungsphase beendet. Beachte, dass wir mit der Laufzeit $O(n)$ für die Schritte (1) und (3), sowie mit der Laufzeit $O(n+p)$ für Schritt (2) bezahlen. (Warum erscheint der Beitrag n für die Laufzeit von Schritt (2)? Es kann passieren, dass viele Listen L_i leer sind, aber dies muss nachgeprüft werden!)

Nach der Vorbereitungsphase sind alle sofort ausführbaren Aufgaben in *Schlange* gesammelt. Wir entnehmen einen Knoten i aus *Schlange* und dürfen Aufgabe a_i ausführen. Demgemäß tragen wir i in das Reihenfolge-Array ein. Jeder Nachfolger j von i verliert durch die Ausführung von Aufgabe a_i einen Verstoß! Können wir diese Verstoßänderung leicht vermerken? Natürlich, mit Hilfe der Liste Priorität $[i]$!

Algorithmus 4.5 **(Topologische Sortierung: Eine effiziente Implementierung)**

(1) Initialisiere die Adjazenliste Priorität durch Einlesen der Prioritäten. (Zeit $= O(n+p)$).

(2) Führe die Vorbereitungsphase durch. /∗ ($Zeit = O(n + p)$). ∗/

(3) Setze $Zähler = 0$;
Wiederhole solange, bis *Schlange* leer ist:

(a) Entferne einen Knoten i aus *Schlange*.

(b) Setze Reihenfolge [Zähler++] $= i$.

(c) Durchlaufe die Liste Priorität $[i]$ und reduziere den In-Grad für jeden Nachfolger j von i um 1. Wenn jetzt In-Grad$[j] = 0$ gilt, dann stopfe j in *Schlange*.
/∗ Aufgabe a_j ist jetzt ausführbar. ∗/

Betrachten wir jetzt die Laufzeit von Schritt (3): n Aufgaben werden (irgendwann) genau einmal in *Schlange* eingefügt, aus *Schlange* entfernt und in das Array Reihenfolge eingetragen. Die Laufzeit $O(n)$ genügt somit für alle Operationen in Schritt (3), die *Schlange* betreffen.

Wenn Aufgabe a_i ausgeführt wird, dann muss die Liste Priorität$[i]$ durchlaufen werden. Wir nehmen an, dass Priorität$[i]$ p_i Elemente besitzt. Demgemäß ist die Laufzeit für Aufgabe i höchstens $O(p_i)$ und beträgt

$$O(p_1 + \cdots + p_n) = O(p)$$

für alle n Aufgaben. Insgesamt lässt sich also Schritt (3) in Zeit $O(n + p)$ durchführen und wir haben erhalten:

Satz 4.2 *Das Problem des topologischen Sortierens von n Aufgaben und p Prioritäten kann in Zeit*

$$O(n + p)$$

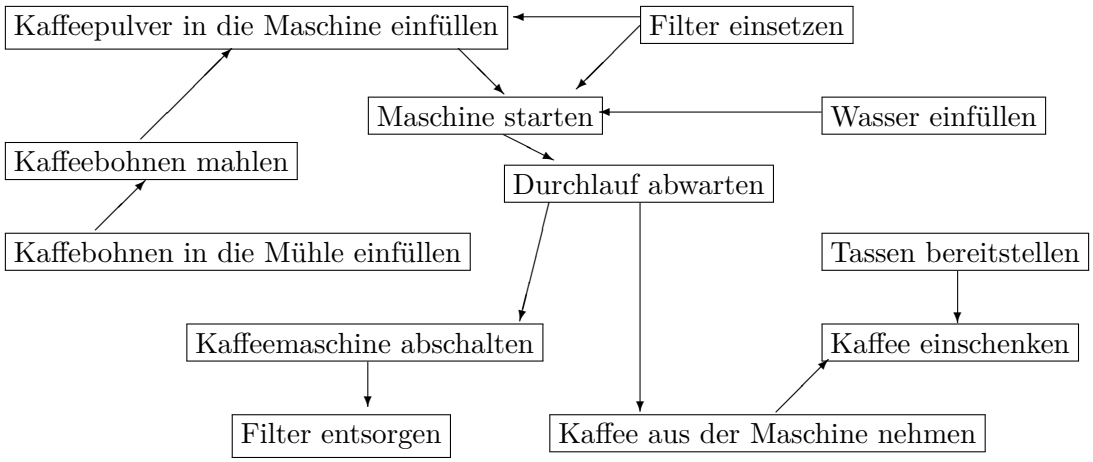
gelöst werden.

Wir haben somit, verglichen mit der Laufzeit der naiven Implementierung, den Faktor n gespart! (Nebenbei, war es notwendig, eine Queue zur Verfügung zu stellen? Hätte ein Stack gereicht?)

Wir wollen unseren Algorithmus nun am Beispiel eines Problems verdeutlichen, dem sich Informatiker üblicherweise eher mit ausführlichen, empirischen Testreihen nähern, dem des Kaffeekochens.

Definition 4.2 Ein Kaffee ist ein ausschließlich aus gemahlenen Kaffeebohnen und Wasser gewonnenes Aufgußgetränk. Insbesondere sind ihm weder Zucker noch Milch beizumischen.

Der Vorgang der Herstellung eines Kaffees im Sinne dieser Definition wird durch das folgende Diagramm beschrieben.



Die Listen Priorität[i] enthalten alle Prioritäten.

Filter entsorgen	→	
Kaffee einschenken	→	
Kaffeemaschine abschalten	→	(Kaffeemaschine abschalten, Filter entsorgen)→
Kaffee aus der Maschine nehmen	→	(Kaffee aus der Maschine nehmen, Kaffee einschenken)→
Tassen bereitstellen	→	(Tassen bereitstellen, Kaffee einschenken) →
Durchlauf abwarten	→	(Durchlauf abwarten, Kaffeemaschine abschalten) →(Durchlauf abwarten, Kaffee aus der Maschine nehmen)→
Maschine starten	→	(Maschine starten, Durchlauf abwarten)→
Kaffeebohnen mahlen	→	(Kaffeebohnen mahlen, Kaffeepulver in die Maschiene einfüllen)→
Kaffebohnen in die Mühle einfüllen	→	(Kaffebohnen in die Mühle einfüllen, Kaffeebohnen mahlen) →
Kaffeepulver in die Maschine einfüllen	→	(Kaffeepulver in die Maschine einfüllen, Maschine starten) →
Filter einsetzen	→	(Filter einsetzen, Kaffeepulver in die Maschine einfüllen) →(Filter einsetzen, Maschine starten)→
Wasser einfüllen	→	(Wasser einfüllen, Maschine starten)→

Nach der Initialisierung erhalten wir das Array In-Grad

Filter entsorgen	1
Kaffee einschenken	2
Kaffeemaschine abschalten	1
Kaffee aus der Maschine nehmen	1
Tassen bereitstellen	0
Durchlauf abwarten	1
Maschine starten	3
Kaffeebohnen mahlen	1
Kaffebohnen in die Mühle einfüllen	0
Kaffeepulver in die Maschine einfüllen	2
Filter einsetzen	0
Wasser einfüllen	0

In unserer Queue *Schlange* befinden sich nach der Initialisierung alle Arbeitsschritte mit In-Grad= 0:

Tassen bereitstellen ⟶ *Kaffeebohnen in die Mühle einfüllen* ⟶ *Filter einsetzen* ⟶ *Wasser einfüllen*

Als Nächstes würden wir *Tassen bereitstellen* aus der Schlange entfernen und zur ersten Aktion erklären. Anhand der betreffenden Priorität-Liste sehen wir, dass die Aktion *Kaffee einschenken* einen Verstoß verliert. Der betreffende Eintrag im Array In-Grad sinkt also von 2 auf 1.

Danach würden wir *Kaffeebohnen in die Mühle einfüllen* aus *Schlange* entfernen und zur zweiten Aktion machen. Jetzt wird *Kaffeebohnen mahlen* den letzten Verstoß verlieren und sein In-Grad sinkt auf 0. *Kaffeebohnen mahlen* ist jetzt ausführbar und ist in *Schlange* aufzunehmen.

Durch die Behandlung von *Filter einsetzen* und *Wasser einfüllen* werden nur In-Gradwerte modifiziert, aber es werden keine neuen Aufgaben *frei*. In der Schlange befindet sich dann nur noch die Aufgabe *Kaffeebohnen mahlen*. Die In-Grade der noch nicht abgearbeiteten Aktionen sind jetzt:

Filter entsorgen	1
Kaffee einschenken	1
Kaffeemaschine abschalten	1
Kaffee aus der Maschine nehmen	1
Durchlauf abwarten	1
Maschine starten	1
Kaffeebohnen mahlen	0
Kaffeepulver in die Maschine einfüllen	1

Kaffeebohnen mahlen wird ausgeführt. Als einzige Kosequenz wird *Kaffepulver in Maschine einfüllen* ausführbar. Als dann einzige ausführbare Aktion, wird das auch gleich erledigt, und *Maschine starten* wird frei. *Durchlauf abwarten* folgt analog. Nachdem dieser Punkt erledigt ist, werden zwei Aktionen ausführbar, *Kaffeemaschine abschalten* und *Kaffee aus der Maschine nehmen*. Führen wir diese Aufgaben nacheinander aus, so ergibt sich das folgende array In-Grad für die verbleibenden Aufgaben:

Filter entsorgen	1
Kaffee einschenken	1
Kaffeemaschine abschalten	0
Kaffee aus der Maschine nehmen	0

Wird die Maschine also abgeschaltet, so rutscht *Filter entsorgen* in die Schlange der ausführbaren Aktionen. Als Nächstes wird der Kaffe aus der Maschine genommen und *Kaffee einschenken* wird endlich möglich. Der Filter landet im Abfall, der Kaffee in den Tassen, und wir haben uns den Kaffee redlich verdient.

Der praktische Testlauf der so ermittelten Reihenfolge sei dem geneigten Leser zur Übung überlassen.

4.4.2 Graph-Implementierungen

Welche Datenstruktur sollten wir für die Implementierung von Graphen wählen? Um diese Frage zu beantworten, müssen wir uns erst darüber klar werden, welche Operationen eine Graph-Implementierung effizient unterstützen soll. Offensichtlich sind die Operationen

- Ist e eine Kante?
- Bestimme die Nachbarn, bzw. Vorgänger und Nachfolger eines Knotens.

relevant. In den Anwendungen, zum Beispiel in unserer Behandlung des Problems der topologischen Sortierung stellt sich heraus, dass die zweite Operation von besonderer Wichtigkeit ist.

Wir betrachten **Adjazenzmatrizen** als eine erste Datenstruktur: Für einen Graphen $G = (V, E)$ (mit $V = \{0, ..., n-1\}$) ist die Adjazenzmatrix A_G von G durch

$$A_G[u,v] = \begin{cases} 1 & \text{wenn } \{u,v\} \in E \text{ (bzw. wenn } (u,v) \in E), \\ 0 & \text{sonst} \end{cases}$$

definiert. Positiv ist die schnelle Beantwortung eines Kantentests in konstanter Zeit. Negativ ist der benötigte Speicherplatzbedarf $\Theta(n^2)$ selbst für Graphen mit relativ wenigen Kanten.

Desweiteren erfordert die Bestimmung der Nachbarn, bzw. Vorgänger oder Nachfolger Zeit $\Theta(n)$, eine Laufzeit, die zum Beispiel in unserer Lösung des topologischen Sortierens nicht akzeptabel ist.

Jetzt zur zweiten Datenstruktur, der **Adjazenzlisten-Repräsentation**, die wir bereits in Abschnitt 4.4.1 kennengelernt haben. Diesmal wird G durch ein Array A von Listen dargestellt. Die Liste $A[v]$ führt alle Nachbarn von v auf, bzw. alle Nachfolger von v für gerichtete Graphen.

Der Speicherplatzbedarf beträgt $\Theta(n+|E|)$ und ist damit *proportional* zur Größe des Graphen. Ein Kantentest wird, im Vergleich zu Adjazenzmatrizen, wesentlich langsamer ausgeführt: Um $(i,j) \in E$? zu beantworten, ist Zeit $\Theta(\text{Aus-Grad } (i))$ notwendig. Die Bestimmung der Nachbarn (bzw. der Nachfolger von v) gelingt in schnellstmöglicher Zeit, nämlich in Zeit $O(\text{Grad } (v))$ (bzw. $O(\text{Aus-Grad } (v))$).

Für dünne Graphen (also Graphen mit wenigen Kanten) ist die Methode der Adjazenzlisten der Methode der Adjazenzmatrix überlegen. Adjazenzlisten sind speicher-effizienter und liefern eine schnellere Implementierung der wichtigen Nachbar- oder Nachfolger-Operation.

4.4.3 Tiefensuche

Wie findet man aus einem Labyrinth, das als ungerichteter Graph dargestellt ist? Zum Einstieg hier ein Auszug aus *Umbert Eco's „Der Name der Rose"*. William von Baskerville und sein Schüler Adson van Melk sind heimlich in die als Labyrinth gebaute Bibliothek eines hochmittelalterlichen Klosters irgendwo im heutigen Norditalien eingedrungen. Fasziniert von den geistigen Schätzen, die sie beherbergt, haben sie sich icht die Mühe gemacht, sich ihren Weg zu merken. Erst zu spät erkennen sie, dass die Räume unregelmäßig und scheinbar wirr miteinander verbunden sind. Man sitzt fest.

> „Um den Ausgang aus einem Labyrinth zu finden,", dozierte William, „gibt es nur ein Mittel. An jedem Kreuzungspunkt wird der Durchgang, durch den man gekommen ist, mit drei Zeichen markiert. Erkennt man an den bereits vorhandenen Zeichen auf einem der Durchgänge, dass man an der betreffenden Kreuzung schon einmal gewesen ist, bringt man an dem Durchgang, durch den man gekommen ist, nur ein Zeichen an. Sind alle Durchgänge schon mit Zeichen versehen, so muss man umkehren und zurückgehen. Sind aber einer oder zwei Durchgänge der Kreuzung noch nicht mit Zeichen versehen, so wählt man einen davon und bringt zwei Zeichen an. Durchschreitet man einen Durchgang, der nur ein Zeichen trägt, so markiert man ihn mit zwei weiteren, so dass er nun drei Zeichen trägt. Alle Teile des Labyrinthes müßten durchlaufen worden sein, wenn man, sobald man an eine Kreuzung gelangt, niemals den Durchgang mit drei Zeichen nimmt, sofern noch einer der anderen Durchgänge frei von Zeichen ist."
>
> „Woher wißt Ihr das? Seid Ihr ein Experte in Labyrinthen?"
> „Nein, ich rezitiere nur einen alten Text, den ich einmal gelesen habe."
> „Und nach dieser Regel gelangt man hinaus?"
> „Nicht dass ich wüßte. Aber wir probieren es trotzdem.[...]"

Eco versetzt den Leser mit dieser Geschichte in die letzte Novemberwoche 1327. Wenn die Figur des William von Baskerville also von einem *alten Text* redet, so kann dieser durchaus aus dem ersten Jahrtausend unserer Zeitrechnung stammen. Graphprobleme wurden schon vor vielen Jahrhunderten (vielleicht Jahrtausenden) diskutiert und sind keineswegs eine Erfindung der Informatik.

Geht das denn nicht viel einfacher? Prinzessin Ariadne, Tochter des Königs Minos, hat Theseus den „Ariadne-Faden" geschenkt. Theseus hat den Ariadne-Faden während der Suche im Labyrinth abgerollt. Nachdem er den Minotauros aufgespürt und getötet hat, braucht er nur den Faden zurückverfolgen, um das Labyrinth wieder verlassen zu können. Aber wie durchsucht man das Labyrinth systematisch mit Hilfe eines Fadens?

Und der nächste Versuch: Präorder (siehe Algorithmus 4.1) hat systematisch alle Knoten eines Baums besucht. Funktioniert Präorder auch für Graphen? Wir haben Pech, für ungerichtete Graphen wie auch für gerichtete Graphen wird Präorder nicht terminieren. Was ist das Problem? Präorder erkennt nicht, dass es Knoten bereits besucht hat!

Und wenn wir, wie schon im Präorder-Verfahren, die Idee des Ariadne-Fadens benutzen, aber zusätzlich alle besuchten Knoten anpinseln? Genau das macht Tiefensuche: Es führt Präorder durch, markiert aber sofort alle erstmals besuchten Knoten.

Aufgabe 44
An welcher Stelle benutzt Präorder die Idee des Ariadne-Fadens?

Wie setzen wir die Idee um? Ein boolesches Array, in dem wir für jeden Knoten vermerken, ob er bereits besucht wurde. Desweiteren nehmen wir an, dass der (gerichtete oder ungerichtete) Graph durch seine Adjazenzliste repräsentiert ist, um schnellstmöglich auf die Nachbarn, bzw. Nachfolger eines Knotens zugreifen zu können. Die Adjazenzliste bestehe aus einem Array $A_$Liste von Zeigern auf die Struktur Knoten mit

```
struct Knoten {
  int name;
  Knoten * next;}
```

Algorithmus 4.6 **(Tiefensuche: Die globale Struktur)**

```
void Tiefensuche()
{
  for (int k = 0; k < n; k++) besucht[k] = 0;
  for (k = 0; k < n; k++)
    if (! besucht[k]) tsuche(k);
}
```

Die globale Organisation garantiert, dass jeder Knoten besucht wird: Es ist nämlich möglich, dass innerhalb des Aufrufes tsuche(0) nicht alle Knoten besucht werden.

`tsuche(v)` wird nur dann aufgerufen, wenn v als nicht besucht markiert ist. Zuallererst wird v deshalb als „besucht“ markiert. Danach wird die Adjazenzliste von v abgearbeitet und alle noch nicht besuchten Nachbarn (oder Nachfolger) von v werden besucht. Die sofortige Markierung von v verhindert somit einen mehrmaligen Besuch von v.

Algorithmus 4.7 **(Tiefensuche für den Knoten** v**)**

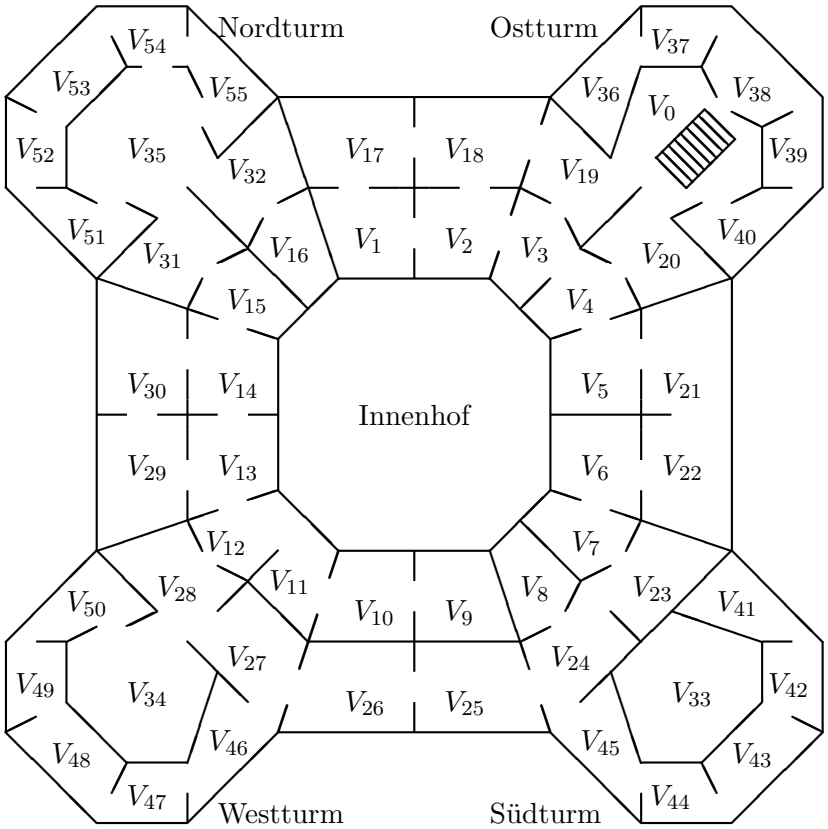
```
void tsuche(int v)
{
   Knoten *p ; besucht[v] = 1;
   for (p = A_Liste [v]; p !=0; p = p->next)
     if (!besucht [p->name]) tsuche(p->name);
}
```

Aufgabe 45
An welcher Stelle benutzt Algorithmus 4.7 die Idee des Ariadne-Fadens?

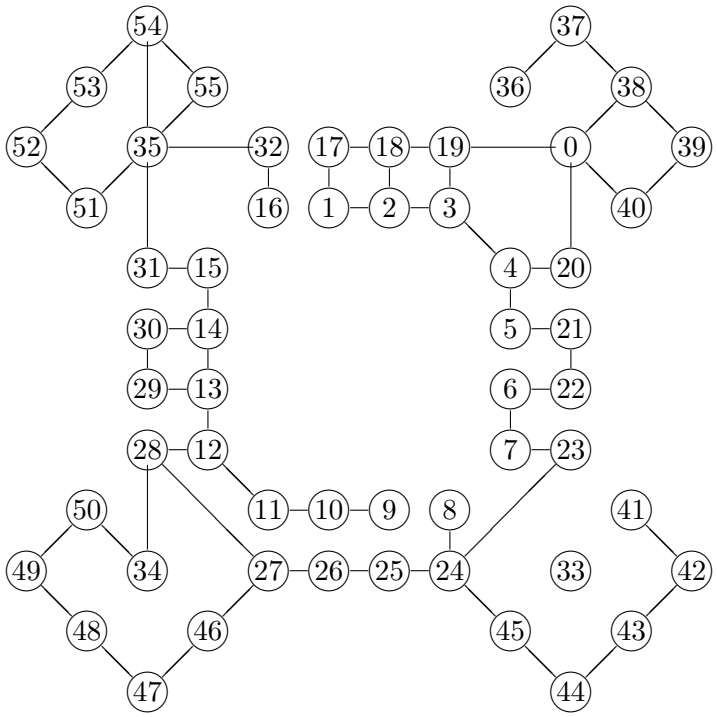
Aufgabe 46
Bestimme den Baum der Tiefensuche für den folgenden, durch seine Adjazenzliste gegebenen ungerichteten Graphen $G = (V, E)$. Starte die Berechnung im Knoten 1.

1	$\longmapsto$	2	3				9	$\longmapsto$	6	8				
2	$\longmapsto$	4	1	5			10	$\longmapsto$	3	11	14	6		
3	$\longmapsto$	6	1	10	4		11	$\longmapsto$	12	10	13	14		
4	$\longmapsto$	5	2	3			12	$\longmapsto$	11	13				
5	$\longmapsto$	4	2				13	$\longmapsto$	12	11				
6	$\longmapsto$	7	3	8	10	9	14	$\longmapsto$	11	10	15			
7	$\longmapsto$	8	6				15	$\longmapsto$	14	16				
8	$\longmapsto$	9	7	6			16	$\longmapsto$	15					

Wir spielen diese Prozedur an dem Beispiel eines Labyrinth-Problems durch: Hier also ist die geheimnisumwitterte Bibliothek aus *„Der Name der Rose“*. Natürlich ist die Benennung der Räume im Original eine andere. Wir haben hier eine fortlaufende Numerierung angelegt. Der Eingang zu diesem Labyrinth befindet sich im Raum V_0, in den man über eine Treppe aus dem unteren Stockwerk gelangen kann. Ziel ist es nun, alle Räume der Bibliothek zu erkunden, die wir von V_0 aus erreichen können.

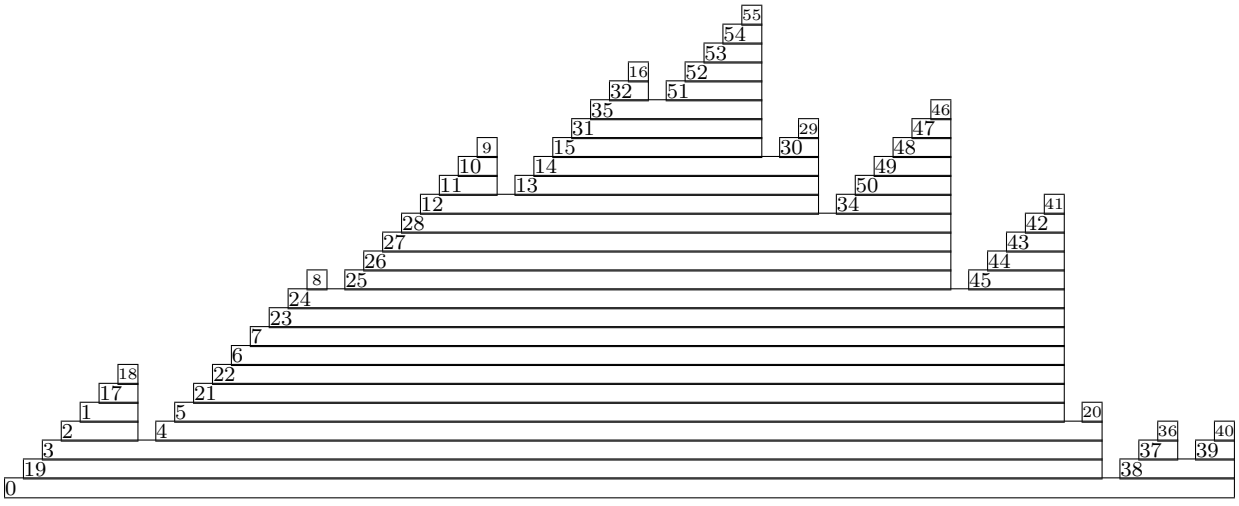


Die Interpretation dieses Labyrinths als Graph ist an sich offensichtlich, dennoch hier die gewohnte Darstellung:



Wir nehmen an, dass die Knoten in jeder Liste von $A_$Liste aufsteigend angeordnet sind. Das heißt, wann immer der Algorithmus mehrere mögliche, nicht markierte Räume zur Auswahl hat, entscheidet er sich für den Raum mit der kleinsten Laufnummer. Um festzuhalten, welcher Knoten welchen Aufruf veranlasst, führen wir für den Aufruf tsuche(0) einen Baum T ein. T hat 0 als Wurzel. Verfolgen wir also den Aufruf tsuche(0): 0 wird (als besucht) markiert und Knoten 19 wird aufgerufen (da 19 nicht markiert ist). Knoten 19 wird markiert und ruft nun seinerseits den *kleinsten* nicht markierten Nachbarknoten auf. Das ist Nummer 3. In V_3 hat der Algorithmus die Wahl zwischen Raum 2 und Raum 4. Er entscheidet sich für Raum 2 und geht von dort weiter zu Nummer 1. Wir erreichen so V_{17} und V_{18}. Hier kommt nun zum ersten mal zum Tragen, dass unser Array „besucht“ global definiert ist. Der Aufruf von V_{18} bemerkt, dass sowohl V_2 als auch V_{19} bereits besucht waren. Es wird festgestellt, dass kein neuer Knoten erreicht werden kann, und dieser Aufruf terminiert. Auch die Aufrufe aus den Knoten 17, 1 und 2 brechen ab, da sie keine neuen Wege aufspüren können. Der Aufruf von V_3 aus übernimmt schließlich wieder das Kommando. Er hat mit V_4 noch einen nicht markierten Nachbarknoten, den er auch prompt aufruft...

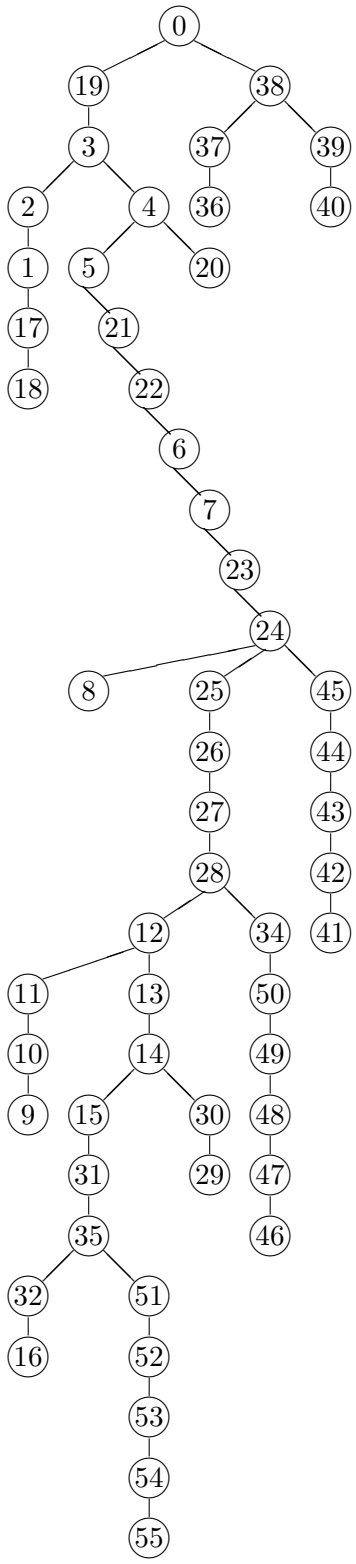
Die weiteren Aufrufe in zeitlicher Reihenfolge zeigt die folgende Grafik.



Wie ist diese Grafik zu lesen? Die horizontale Achse stellt die Zeit dar. Die vertikale kann als Rekusionstiefe gelesen werden. Jeder Aufruf wird durch einen Balken vertreten in dem die Nummer des aufrufenden Knotens (Raumes) eingetragen ist. Die Länge des Balkens entspricht somit der Dauer des Aufrufs. Die auf einem Balken liegenden kürzeren Balken sind die rekursiv eingebetteten Aufrufe. Unmittelbar unter einem Balken findet man den Balken des Aufrufs, der den betrachteten ins Leben gerufen hat.

Was können wir dieser Grafik entnehmen? Wir erkennen die maximale Rekursionstiefe (25, wenn man den V_0-Aufruf mitzählt). Wenn wir die Zahlen von links nach rechts lesen, erhalten wir die Reihenfolge, in der die Tiefensuche die Räume ansteuert. Die diagonalen „Anstiege“ (etwa von der vier bis zur acht) sind die Phasen, in denen wir *in Neuland vorstoßen*. Die senkrechten Abwärtssprünge entsprechen den Suchabschnitten, in denen wir ein entsprechend langes Stück zurücklaufen müssen, um in einen Raum zurück zu gelangen, von dem aus es noch nicht verfolgte Alternativen gibt. Außerdem können wir dieser Grafik bequem Aufruf- und Terminierungszeitpunkt einer bestimmten Teilsuche entnehmen, die sich später als wichtige Größen entpuppen werden.

Wir sehen auch, dass Raum V_{33} nicht erreicht wurde. Nachdem man sich den Plan oder den Graphen angesehen hat, ist das nicht weiter verwunderlich. Er liegt nicht in der selben Zusammenhangskomponente wie der Startknoten V_0. Es führt keine Tür in den Raum mit der Nummer 33. (Eigentlich ist im Roman schon eine Tür, aber es ist eine Geheimtür und um die zu öffnen, muss man zunächst auf den *ersten und siebten der Vier...* . Doch wir wollen hier auch nicht die ganze Geschichte verraten.)



Betrachten wir den oben erwähnten Aufrufbaum T als alternative, weniger aufwändige Darstellung des Ablaufs.

Wir können hier die maximale Rekursionstiefe als Tiefe des Baumes wiederfinden. Die Reihenfolge, in der man die Räume besucht, entspricht dem Präorder-Durchlauf des Baumes. Wie Rekursionen ineinander eingebettet sind, lässt sich auch ablesen. Jeder Aufruf enthält genau jene Aufrufe intern, die im Aufrufbaum direkte Nachfahren von ihm sind, die also im entsprechenden Teilbaum liegen.

Wir beobachten weiter, dass alle Kanten des Baumes auch Kanten des Graphen waren. Andererseits sind aber längst nicht alle Kanten des Graphen auch im Baum enthalten. (Wir wissen, dass ein Baum mit n Knoten genau $n-1$ Kanten besitzt, dass aber andererseits ein allgemeiner ungerichteter Graph mit n Knoten bis zu $\frac{1}{2}n\cdot(n-1)$ Kanten haben kann. Also überrascht uns das nicht.) Die Kanten des Graphen, die im Aufrufbaum der Tiefensuche enthalten sind, nennen wir **Baumkanten**.

Andere Kanten, wie zum Beispiel die Kante {18,2}, verbinden einen Knoten mit einem seiner Vorfahren. Wir erinnern uns: Im Raum 18 haben wir den Durchgang zu Raum 2 zwar gesehen, aber an der Markierung erkannt, dass das für uns ein Schritt zurück wäre, an einen Punkt, an dem wir schon waren. Solche Kanten nennen wir fortan **Rückwärtskanten**.

Haben wir damit alle Kanten des ursprünglichen Graphen erfaßt? Oder können Kanten des Graphen zwei Knoten des Baumes verbinden, die nicht Vor- und Nachfahre voneinander sind? Nein. (Warum nicht?)

Man beachte dass in unserem Beispiel mehrere Zusammenhangskomponenten vorliegen. Es hat nicht ausgereicht, die Tiefensuche einmal zu starten, da V_{33} nicht erreichbar war. An sich hätte ein zweiter Aufruf, beginnend beim ersten noch nicht gefundenen Knoten, erfolgen müssen. Wir hätten also noch einen zweiten trivialen Einknotenbaum erhalten. Im Allgemeinen werden wir deswegen von nun an von dem **Wald der Tiefensuche** sprechen.

Wir haben Tiefensuche als Möglichkeit, systematisch sämtliche Knoten eines Graphen zu besuchen, kennengelernt. Wir haben am Beispiel dieses Labyrinths gezeigt, wie uns Tiefensuche hilft, aus dem Labyrinth herauszufinden. Nun ist es nicht gerade ein alltägliches Problem der Informatik, dass man in einer mittelalterlichen Bibliothek ausgesetzt wird und den Ausgang finden muss. Die Anwendung auf weitere Labyrinth-Problemen, wie etwa das Durchsuchen des world-wide-web, ist aber vom selben Kaliber:

Beispiel 4.5 Im Sekundentakt wird das world-wide-web von einer stetig steigender Zahl von Informationssuchenden durchforstet. Im selben Tempo wächst das Netz selbst. Längst ist es völlig hoffnungslos, auf gut Glück durch das Web zu streifen und dabei bestimmte Informationen finden zu wollen. Hier leisten die Suchmaschinen gute Arbeit. *Spinnen* werden durchs Web geschickt, die sich von Link zu Link hangeln, versuchen innerhalb der aktuellen Sei-

te Stichworte ausfindig zu machen und diese dann möglichst raffiniert in einer Datenbank abrufbar zu halten.

Nun soll eine Spinne das Web natürlich möglichst systematisch erforschen. Was ist also die Grundidee in einem Spinnenprogramm? – Ein Graph-Traversal und damit insbesondere Tiefensuche oder die noch zu besprechende Breitensuche bzw. andere Varianten.

Wir kommen als Nächstes zur zentralen Eigenschaft von Tiefensuche für ungerichtete Graphen:

Satz 4.3 *Sei $G = (V, E)$ ein ungerichteter Graph und es gelte $\{u, v\} \in E$. W_G sei der Wald der Tiefensuche für G.*

(a) *tsuche(u) werde vor tsuche(v) aufgerufen. Dann ist v ein Nachfahre von u in W_G.*

(b) *G besitzt nur Baum- und Rückwärtskanten.*

Beweis : **(a)** Da tsuche(u) vor tsuche(v) aufgerufen wird, ist Knoten v zum Zeitpunkt der Markierung von Knoten u *unmarkiert*. Aber tsuche(u) kann nur dann terminieren, wenn auch v markiert wurde. Mit anderen Worten, v wird irgenwann während der Ausführung von tsuche(v) markiert und wird deshalb zu einem Nachfahren von u. **(b)** Wenn $\{u, v\}$ eine Kante ist, dann ist u Vorfahre oder Nachfahre von v. □

Satz 4.4 *Sei $G = (V, E)$ ein ungerichteter Graph, der als Adjazenzliste vorliege.*

(a) *Tiefensuche besucht jeden Knoten von V genau einmal.*

(b) *Die Laufzeit von Tiefensuche ist durch*

$$O(|V| + |E|)$$

beschränkt.
(c) *Sei W_G der Wald der Tiefensuche für G und v sei ein Knoten von G. Dann enthält der Baum von v genau die Knoten der Zusammenhangskomponente von v, also alle die Knoten, die durch einen Weg mit Anfangsknoten v erreichbar sind.*

Damit entsprechen die Bäume von W_G eindeutig den Zusammenhangskomponenten von G.

Beweis : **(a)** Das Programm Tiefensuche() erzwingt, dass jeder Knoten mindestens einmal besucht wird; also wird mindestens ein Aufruf tsuche(v) für einen jeden Knoten v durchgeführt. Nach einem Aufruf von tsuche(v) wird aber v sofort markiert, und nachfolgende Aufrufe von tsuche(v) sind deshalb unmöglich.

(b) Sei $v \in V$ beliebig. Dann werden höchstens

$$O(1 + \text{grad}(v))$$

nicht-rekursive Operationen in tsuche(v) ausgeführt. Nach **(a)** wissen wir, dass tsuche(v) für jeden Knoten V genau einmal ausgeführt wird. Die Gesamtzahl der damit ausgeführten Operationen ist somit

$$\begin{aligned} & O \quad (\sum_{v \in V} (1 + \text{grad } (v)) \\ = \; & O \quad (\sum_{v \in V} 1 + \sum_{v \in V} \text{grad } (v)) = O(|V| + |E|). \end{aligned}$$

Zusätzlich zu den während eines Aufrufs von tsuche ausgeführten Operationen ist noch der Aufwand für die for-Schleife im Program Tiefensuche() zu berücksichtigen. Dies führt aber nur zu $O(|V|)$ zusätzlichen Schritten.

(c) Sei $v \in V$ beliebig. T sei ein Baum im Wald W_G und T besitze v als Knoten. v erreicht jeden Knoten in T, denn T ist zusammenhängend. Also ist die Zusammenhangskomponente von v eine Obermenge der Knotenmenge von T. Wenn es aber einen Weg

$$v = v_0 \quad \rightarrow \quad v_1 \quad \rightarrow \cdots \rightarrow \quad v_m = u$$

gibt, dann gehören $v_0, v_1, \ldots, v_m$ alle zum selben Baum (nach Satz 4.3 (b)). Also ist die Zusammenhangskomponente von v eine Untermenge der Knotenmenge von T.

Bemerkung 4.1 Tiefensuche kann damit jedes Labyrinth-Problem, das sich als ungerichteter Graph interpretieren lässt, lösen. Denn, wenn es möglich ist, vom Eingang den Ausgang zu erreichen, dann befinden sich Eingang und Ausgang in derselben Zusammenhangskomponente. Der Aufrufbaum des Eingangs wird uns stets einen Weg aus dem Labyrinth zeigen.

Satz 4.5 *Sei $G = (V, E)$ ein ungerichteter Graph. Dann kann in Zeit $O(|V| + |E|)$ überprüft werden, ob*

(a) *G zusammenhängend ist,*

(b) *G ein Baum ist.*

Beweis : **(a)** Dies ist eine Konsequenz von Satz 4.4. Denn G ist genau dann zusammenhängend, wenn tsuche(0) alle Knoten besucht.

(b) G ist genau dann ein Baum, wenn es keine Rückwärtskanten gibt und wenn G zusammenhängend ist. Wir wissen schon aus Teil (a), wie Zusammenhang schnell nachgeprüft wird. Wann hat G eine Rückwärtskante? Genau dann, wenn es eine Kante $e = \{v, w\} \in E$ gibt, die **keine** Baumkante ist. Also ist G genau dann ein Baum, wenn G zusammenhängend ist und genau $n - 1$ Kanten, nämlich nur die Baumkanten besitzt. □

Aufgabe 47
Gegeben sei ein zusammenhängender, ungerichteter Graph $G = (V, E)$ durch seine Adjazenzliste. Jeder Knoten $v \in V$ besitzt eine gerade Anzahl von Nachbarn.
Ein Euler-Kreis ist ein Kreis, in dem jede Kante des Graphen genau einmal vorkommt. (Knoten dürfen also mehrmals durchlaufen werden.)

Beschreibe einen *möglichst effizienten* Algorithmus (**und** die zugehörigen Datenstrukturen), um einen Euler-Kreis zu bestimmen. **Bestimme** die Laufzeit deines Algorithmus. (Laufzeit $O(|V| + |E|)$ ist möglich.)

Vorsicht: Wenn du einen Euler-Kreis sukzessive, Kante nach Kante, aufbaust, kann es passieren, dass der bisher konstruierte Weg zum Anfangsknoten zurückkehrt. Was tun, wenn alle mit dem Anfangsknoten inzidenten Kanten bereits durchlaufen sind, aber es noch Kanten gibt, die nicht durchlaufen wurden?

Als Nächstes besprechen wir Tiefensuche für gerichtete Graphen. Die Situation ist ein wenig komplizierter.

Satz 4.6 *Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste vorliegt.*

(a) *Tiefensuche besucht jeden Knoten von V genau einmal.*

(b) *Die Laufzeit von Tiefensuche() ist durch*

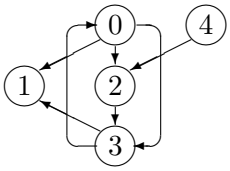
$$O(|V| + |E|)$$

beschränkt.

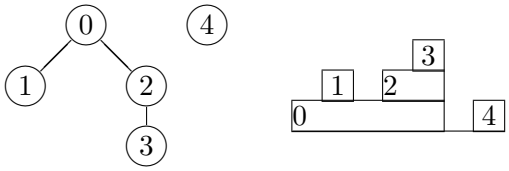
(c) *Für jeden Knoten v in V gilt: tsuche(v) wird genau die Knoten besuchen, die auf einem unmarkierten Weg mit Anfangsknoten v liegen. (Ein Weg heißt unmarkiert, wenn alle Knoten vor Beginn von tsuche(v) unmarkiert sind).*

Beweis : analog zum Beweis von Satz 4.4.

Beispiel 4.6



Wir nehmen wieder an, dass in allen Listen von $A_$Liste die Knoten in aufsteigender Reihenfolge erscheinen. Dann produziert Tiefensuche den folgenden Wald W_G:

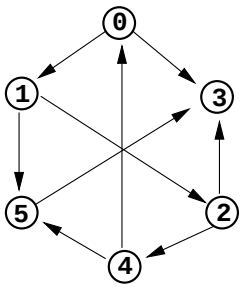


Für ungerichtete Graphen erhielten wir nur zwei Typen von Kanten. Diesmal erhalten wir vier Typen. Nämlich

- die **Baumkanten** $(0,1),(0,2),(2,3)$. Diese Kanten sind genau die Kanten des Graphen, die auch im Wald der Tiefensuche auftauchen.
- die **Vorwärtskante** $(0,3)$, die im Aufrufbaum von einem Knoten zu einem (nicht unmittelbaren) Nachfolger "zielt",
- die **Rückwärtskante** $(3,0)$, die im Aufrufbaum von einem Knoten zu einem (nicht unmittelbaren) Vorgänger "zielt", und
- die (rechts nach links) **Querkanten** $(3,1)$ und $(4,2)$, die Knoten im Aufrufbaum verbinden, die nicht in einer Nachfolger-Vorgänger Beziehung stehen.

Beachte auch, dass Knoten 2 nicht im Baum von 4 auftaucht, obwohl 2 ein Nachfolger von 4 ist. Der Grund: 2 war schon besucht, wenn tsuche(4) aufgerufen wurde.

Aufgabe 48
Bestimme den Baum der Tiefensuche für den folgenden, durch seine Adjazenzliste gegebenen, gerichteten Graphen $G = (V, E)$. Starte die Berechnung im Knoten 0. **Klassifiziere** alle Kanten als Baum-, Rückwärts-, Vorwärts- oder Querkanten.




```
0 -> 1,3
1 -> 2,5
2 -> 3,4
3 -> nil
4 -> 0,5
5 -> 3
```

Bemerkung 4.2 Wir verwenden hier die Konvention, die diversen Bäume des Waldes in zeitlicher Folge von links nach rechts einzutragen. Links nach rechts Querkanten können nicht auftauchen. Denn sei $e = (v, w)$ eine beliebige Kante. Nach Satz 4.6(c) wissen wir, dass w nach Terminierung von tsuche(v) markiert sein wird. Wenn w vor dem Aufruf von tsuche(v) markiert wurde, dann ist e entweder eine Rückwärtskante oder eine (rechts nach links) Querkante. Wenn w während des Aufrufs markiert wird, dann ist e eine Vorwärtskante oder eine Baumkante.

Eine automatische Erkennung der verschiedenen Kantentypen ist in einigen Anwendungen der Tiefensuche wichtig. Dies ist zum Beispiel der Fall, wenn wir feststellen wollen, ob ein gerichteter Graph azyklisch ist (also keine Kreise besitzt):

$$G \text{ azyklisch} \Leftrightarrow G \text{ besitzt keine Rückwärtskanten.}$$

Denn eine Rückwärtskante schließt im Wald W_G einen Kreis und azyklische Graphen besitzen somit keine Rückwärtskanten. Wenn andererseits G keine Rückwärtskanten besitzt, können seine Baum-, Vorwärts- und (rechts nach links) Querkanten keinen Kreis schließen!

Wie aber erkennt man die einzelnen Kantentypen? Sei $e = (u, v)$ eine Kante von G.

(1) e ist eine Baumkante $\Leftrightarrow$ tsuche(v) wird unmittelbar aus tsuche(u) aufgerufen.

Zur automatischen Charakterisierung der restlichen Kantentypen müssen wir noch zusätzliche Information in tsuche(u) zur Verfügung stellen. Dazu definieren wir die globalen integer-Arrays „Anfang“ und „Ende“ und benutzen die globalen integer Variablen „Anfang_nr“ und „Ende_nr“. Die beiden Arrays wie auch die beiden Variablen werden zu Anfang auf 0 gesetzt.

Algorithmus 4.8 **(Tiefensuche: Anfang- und Ende-Zähler)**

```
void tsuche(int v) {
  Knoten *p;
  Anfang[v] = ++Anfang_nr;
  for (p = A_Liste[v]; p != 0; p = p->next)
    if (!Anfang[p->name]) tsuche(p->name);
  Ende[v] = ++Ende_nr;
}
```

Wir benutzen also „Anfang [$p \to$ name] == 0“ als Ersatz für „besucht[$p \to$ name] == 0“. Die entsprechende Änderung ist auch in Tiefensuche() durchzuführen, insbesondere entfällt das Array „besucht“.

Algorithmus 4.9 **(Tiefensuche: Kantenklassifizierung)**

(2) $e = (u, v)$ ist eine Vorwärtskante $\Leftrightarrow$

- Anfang $[u] <$ Anfang $[v]$ und
- $e = (u, v)$ ist keine Baumkante.

(3) $e = (u, v)$ ist eine Rückwärtskante $\Leftrightarrow$

- Anfang $[u] >$ Anfang $[v]$ und
- Ende $[u] <$ Ende $[v]$.

(4) $e = (u, v)$ ist eine Querkante $\Leftrightarrow$

- Anfang $[u] >$ Anfang $[v]$ und
- Ende $[u] >$ Ende $[v]$.

Am Beispiel des obigen Graphen erhalten wir:

Knoten	Anfang	Ende
0	1	4
1	2	1
2	3	3
3	4	2
4	5	5

Kante	Anfang	Ende	Klassifizierung
(0,1)	$<$	$>$	Baumkante
(0,2)	$<$	$>$	Baumkante
(0,3)	$<$	$>$	Vorwärtskante
(2,3)	$<$	$>$	Baumkante
(3,0)	$>$	$<$	Rüchwärtskante
(3,1)	$>$	$>$	Querkante
(4,2)	$>$	$>$	Querkante

Aufgabe 49
Bestimme die verschiedenen Kantentypen der Tiefensuche für den folgenden, durch seine Adjazenzliste gegebenen gerichteten Graphen $G = (V, E)$. Starte die Berechnung der Tiefensuche im Knoten a.

$$\begin{array}{lcl} a & \longmapsto & c \\ b & \longmapsto & h \quad i \\ c & \longmapsto & d \quad e \quad g \\ d & \longmapsto & a \quad g \\ e & \longmapsto & g \end{array} \qquad \begin{array}{lcl} f & \longmapsto & c \quad e \quad h \\ g & \longmapsto & d \\ h & \longmapsto & a \quad i \\ i & \longmapsto & f \end{array}$$

Satz 4.7 *Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste repräsentiert ist. Dann lassen sich die beiden folgenden Probleme in Zeit $O(|V| + |E|)$ lösen:*

(a) *Ist G azyklisch?*

(b) *Ist G stark zusammenhängend?*

Beweis : **(a)** Wir wissen schon, dass G genau dann azyklisch ist, wenn G keine Rückwärtskanten besitzt. Eine Erkennung einer Rückwärtskante gelingt aber mit den Anfang- und Ende-Arrays in konstanter Zeit.

(b) Offensichtlich ist G genau dann stark zusammenhängend, wenn alle Knoten von Knoten 1 durch einen Weg in G erreichbar sind und wenn jeder Knoten auch Knoten 1 erreicht. Die erste Bedingung können wir leicht mit tsuche(1) überprüfen; die zweite Bedingung kann aber auch mit tsuche(1) überprüft werden, wenn wir in G die Richtungen aller Kanten umkehren! Man mache sich klar, dass diese Umkehrung in Zeit $O(|V| + |E|)$ möglich ist. □

Aufgabe 50
Gegeben sei ein gerichteter Graph $G = (V, E)$ in Adjazenzlisten-Darstellung. **Erweitere** die Tiefensuche so, dass ein beliebiger Kreis ausgegeben wird, falls ein solcher vorhanden ist.

Aufgabe 51
Ein Graph-Algorithmus heißt polynomiell, wenn für Graphen mit n Knoten und e Kanten worst-case Laufzeit $O((n+e)^k)$ für eine Konstante $k \in \mathbb{N}$ benötigt wird.

Gibt es einen polynomiellen Algorithmus, der *alle* Kreise ausgibt? **Begründe** deine Antwort.

4.4.4 Breitensuche

Tiefensuche durchsucht den Graphen gemäß dem Motto „depth first“: Für jeden Knoten wird mit dem ersten Nachfolger, dann dessen erstem Nachfolger usw. weitergesucht. Wir entwickeln eine zweite Methode, nämlich **Breitensuche**, um systematisch alle Knoten zu besuchen. Diese Methode hat den Vorteil, dass einfache Distanzprobleme effizient lösbar sind.

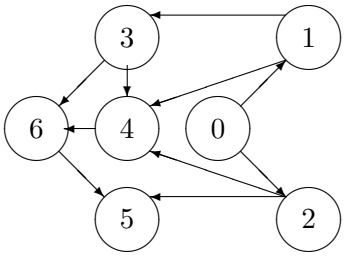
Breitensuche folgt dem „breadth first“-Motto: Für jeden Knoten werden zuerst alle Nachfolger besucht, gefolgt von der Generation der Enkel und so weiter. Die Knoten werden also in der Reihenfolge ihres Abstands von v, dem Startknoten der Breitensuche, erreicht.

Algorithmus 4.10 **(Breitensuche: eine Implementierung mit der Klasse “queue”.)**

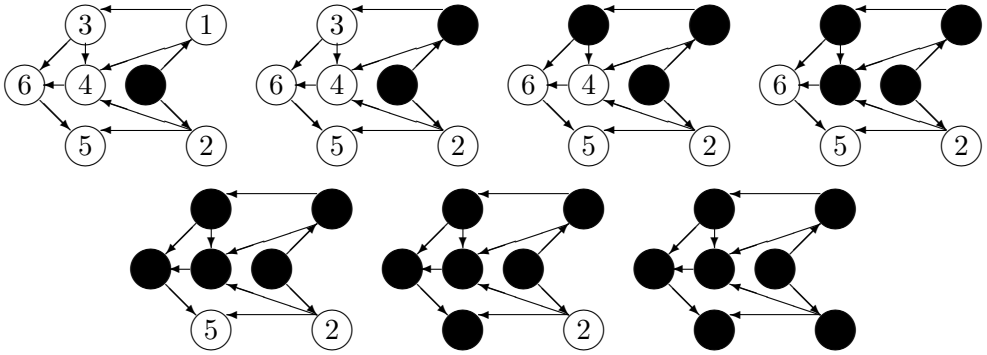
```
void Breitensuche(int v) {
  Knoten *p; int w;
  queue q; q.enqueue(v);
  for (int k =0; k < n ; k++)
    besucht[k] = 0;
  besucht[v] = 1;
  /* Die Schlange q wird benutzt. v wird in die Schlange eingefuegt
  und markiert. */

  while (!q.empty ( ))
  {
    w = q. dequeue ( );
    for (p = A_List[w]; p != 0; p = p->next)
      if (!besucht[p->name])
      {
        q.enque(p->name);
        besucht[p->name] = 1;
      }
  }
}
```

Wir untersuchen die Unterschiede zwischen Tiefen- und Breitensuche.

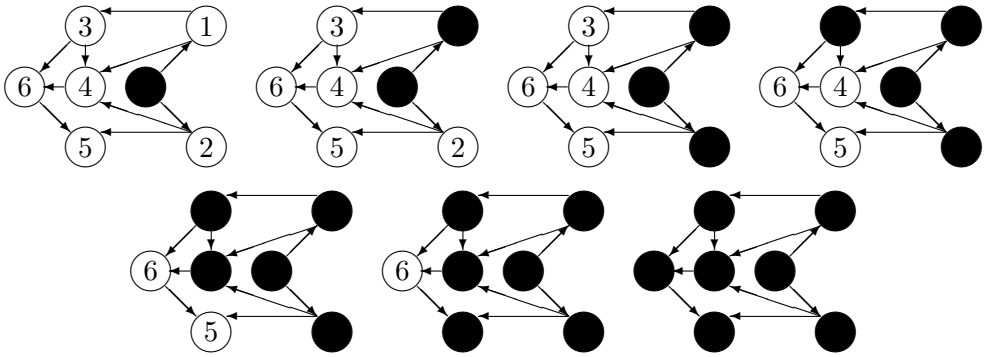
Beispiel 4.7



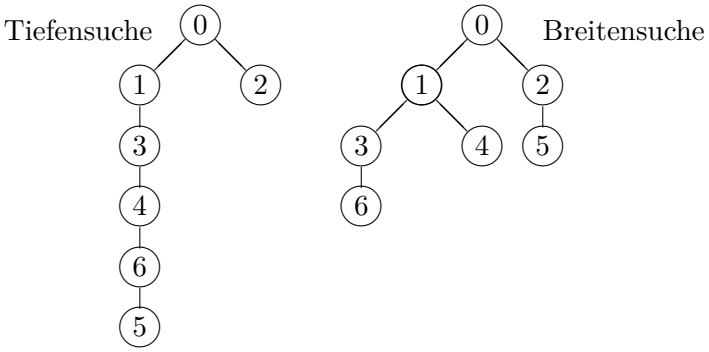
Wir wählen 0 als Startknoten und gehen wieder von einer aufsteigenden Reihenfolge der Knoten innerhalb der Adjazenzlisten aus. Wir betrachten zuerst Tiefensuche.



Als Nächstes ist Breitensuche dran:



Warum das eine Tiefen- und das andere Breitensuche heißt, wird klar, wenn man die Aufrufbäume vergleicht.



Aufgabe 52
Welche Kantentypen (Baumkanten, Vorwärtskanten, Rückwärtskanten oder Querkanten) treten bei Breitensuche für ungerichtete Graphen auf? **Welche** Kantentypen treten für gerichtete Graphen auf?
Gib Beispielgraphen an, in denen die von dir behaupteten Kantentypen auftreten.

Wie effizient ist Breitensuche und was wird erreicht? Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Für Knoten w setze

$$V_w = \{u \in V : \text{ Es gibt einen Weg von } w \text{ nach } u\}$$

und

$$E_w = \{e \in E : \text{ beide Endpunkte von } e \text{ gehören zu } V_w\}.$$

(V_w, E_w) ist also der von V_w induzierte Graph.

Satz 4.8 *Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph, der als Adjazenzliste repräsentiert ist. Sei $w \in V$.*

(a) Breitensuche(w) besucht jeden Knoten in V_w genau einmal und sonst keinen anderen Knoten.

(b) Breitensuche(w) läuft in Zeit höchstens

$$O(|V_w| + |E_w|).$$

Beweis : Übungsaufgabe.

Breitensuche ist also ebenso effizient wie Tiefensuche. Für ungerichtete Graphen wird Breitensuche(w) (ebenso wie tsuche(w)) die Zusammenhangskomponente von w besuchen. Auch für gerichtete Graphen zeigen Breitensuche und Tiefensuche ähnliches Verhalten: Breitensuche(w) und tsuche(w) besuchen alle von w aus erreichbaren Knoten.

Aber der von Breitensuche(w) erzeugte „Baum“ T_w hat eine wichtige zusätzliche Eigenschaft. Zuerst besprechen wir die Definition von T_w. T_w ist anfänglich leer. Wenn in Breitensuche(w) ein Knoten u aus der Schlange entfernt wird und ein Nachbar (bzw. Nachfolger) u' in die Schlange eingefügt wird, dann fügen wir die Kante $\{u, u'\}$ (bzw. (u, u')) in T_w ein.

Definition 4.3 Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph und $T = (V', E')$ sei ein Baum mit $V' \subseteq V$ und $E' \subseteq E$. T besitze w als Wurzel. Wir sagen, dass T ein **Baum von kürzesten Wegen** für G ist, falls

- $V' = V_w$
- für jeden Knoten $u \in V'$ ist

$$\text{Tiefe } (u) = \text{Länge eines kürzesten Weges von } w \text{ nach } u.$$

(Die Wege im Baum T sind also kürzeste Wege.)

Satz 4.9 *Der von Breitensuche(w) definierte Baum T_w ist ein Baum von kürzesten Wegen für G.*

Beweis : Übungsaufgabe.

Als Konsequenz von Satz 4.8 und Satz 4.9 erhalten wir

Satz 4.10 *Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph, der als Adjazenzliste repräsentiert sei. Sei $w \in V$. Dann können wir in Zeit*

$$O(|V| + |E|)$$

kürzeste Wege von w zu allen anderen Knoten bestimmen.

Beweis : Das Korollar folgt direkt aus Satz 4.8 und Satz 4.9, wenn wir wissen, wie der Baum T_w effizient aufgebaut werden kann. Der Aufbau gelingt mühelos, wenn wir eine geeignete Baum-Implementierung wählen. Aber welche?

Aufgabe 53
Ein ungerichteter Graph ist zwei-färbbar, wenn seine Knoten mit zwei Farben gefärbt werden können, so dass keine Kante gleichfarbige Knoten verbindet.

Gegeben sei ein ungerichteter Graph $G = (V, E)$ durch seine Adjazenzliste. **Beschreibe** einen *möglichst effizienten* Algorithmus, der entscheidet, ob G zwei-färbbar ist. **Bestimme** die Laufzeit deines Algorithmus.

Aufgabe 54
Gegeben sei ein **azyklischer** gerichteter „Spielgraph“ G. Ein Knoten sei als Startknoten ausgewiesen. Einen Knoten v mit Aus-Grad$(v) = 0$ nennen wir eine Senke. Die Senken weisen, in Abhängigkeit davon, wer am Zug ist, den Sieger aus, d.h. sie tragen eine der folgenden vier Beschriftungen. „A gewinnt“, „B gewinnt“, „Wer auf den Knoten zieht, gewinnt“, „Wer auf den Knoten zieht, verliert“.

Zwei Spieler A und B spielen auf G ein Spiel, indem die Spieler abwechselnd, mit A beginnend, vom Startknoten ausgehend eine Kante auswählen und so einen Weg bilden. Wird ein Senke erreicht, so wird der Sieger ermittelt.

(a) **Entwerfe** einen Algorithmus, der zu gegebenem Spielgraph $G = (V, E)$ entscheidet, welcher der beiden Spieler eine Gewinnstrategie hat, also stets gewinnt, wenn er optimal spielt. Der Algorithmus soll in Zeit $O(|V| + |E|)$ laufen.

(b) **Modelliere** folgendes Spiel durch einen Spielgraph.
Auf einem Tisch befinden sich h Streichhölzer. Der Spieler, der an der Reihe ist, nimmt wahlweise $k_1, k_2, \ldots, k_{r-1}$ oder k_r Hölzer vom Tisch. Dabei gilt stets $k_i > 0$. Der Spieler, der das letzte Streichholz nehmen muss, verliert. Demonstriere Deinen Algorithmus für $h = 9, r = 3$ und $k_1 = 1, k_2 = 2, k_3 = 3$. Bestimme die Laufzeit aber für allgemeine h, r und $k_1, \ldots, k_r$.

(c) Für das folgende Spiel wird vorgeschlagen, den Spielgraphen zu generieren und ihn dem Algorithmus aus Teil a) zu übergeben. Was ist von diesem Vorgehen zur Bestimmung der Strategie zu halten?
Die beiden Spieler nennen abwechselnd die Hauptstadt eines Staates. Bedingung ist, dass die genannte Stadt immer mit dem Buchstaben anfangen muss, mit dem die vorhergehende endet. Also etwa : Berlin - Nairobi - Islamabad - Dublin - Nikosia - Ankara - Athen - Nassau - Ulan Bator - Rom - Montevideo - Oslo - Ottawa - Addis Abeba - Algier - Reykjavik - Kiew - Wien - .... Außerden darf keine Stadt zweimal genannt werden. Wer als erstes nicht mehr weiter weiß verliert.

Aufgabe 55
Um das Parkplatzproblem zu mindern, beschließt die Verwaltung eines Frankfurter Stadtteils die bisher in beiden Fahrtrichtungen nutzbaren Straßen komplett in Einbahnstraßen umzuwandeln. Danach soll es natürlich noch möglich sein, von jedem Ort zu jedem anderen zu gelangen.

Wir interpretieren das Straßensystem als zunächst ungerichteten Graphen. Jeder Kante soll nun eine Richtung zugewiesen werden. Der resultierende Graph muß stark zusammenhängend sein.

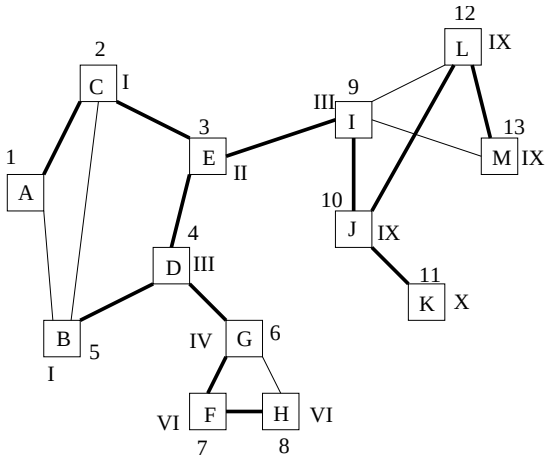
Entwerfe einen möglichst effizienten Algorithmus, der zu einem gegebenem Netz von in beiden Richtungen befahrbaren Straßen ein stark zusammenhängendes Einbahnstraßennetz findet, wann immer ein solches Netz existiert.

Hinweis: Ein stark zusammenhängendes Einbahnstraßennetz existiert genau dann, wenn der Ausgangsgraph zusammenhängend ist und keine Brücke existiert. Eine Brücke ist eine Kante, durch deren Löschung der Graph in zwei Zusammenhangskomponenten zerfällt.

Bei der Konstruktion könnten die Eigenschaften des Walds der Tiefensuche und insbesondere die Eigenschaften von Baumkanten und Rückwärtskanten, hilfreich sein.

Aufgabe 56
Eine Kante in einem zusammenhängenden, ungerichteten Graphen heißt *Brücke*, wenn das Entfernen dieser Kante den Graphen in zwei Teile zerfallen läßt.

(a) Wir möchten zunächst den Algorithmus der Tiefensuche so erweitern, dass jedem Knoten, mit Ausnahme des Startknotens, v zusätzlich zu der eigenen Anfangszeit der frühste Zeitpunkt $T(v)$ zugewiesen wird, zu dem Tiefensuche einen Nachbarn von v besucht.
Beispiel:



Der Verlauf der Tiefensuche ist durch die dicken Kanten gegeben. In arabischen Ziffern findet sich die Anfangszeit eines jeden Knotens, in römischen Ziffern die zu bestimmende Zeit des ersten Besuchs eines Nachbarn.

(b) **Wie** lassen sich *alle* Brücken in Zeit $O(|V| + |E|)$ berechnen? Hinweis: Die Berechnung von $T^*(v) = \min\{T(w) \mid w \text{ ist Nachfahre von } v\}$ könnte hilfreich sein.

4.5 Prioritätswarteschlagen

Wir werden jetzt die Datenstruktur Queue wesentlich verallgemeinern: Statt stets das älteste Element zu entfernen, möchten wir im Stande sein, das Element höchster Priorität zu entfernen. Mit anderen Worten, wir nehmen an, dass jedem Element eine Priorität zugewiesen ist und dass jeweils das Element höchster Priorität zu entfernen ist.

Der abstrakte Datentyp „Prioritätswarteschlange“ umfasst dann die Operationen

- `insert(Priorität)` und
- `delete_max()`.

Wir werden zusätzlich auch die Operationen

- `change_priority(wo,neue Priorität)` und
- `remove(wo)`

unterstützen. Allerdings benötigen wir für die beiden letzten Operationen Hilfe: Es muss uns gesagt werden, wo das betroffene Element ist, die bloße Angabe des Namens reicht uns hier nicht aus.

Natürlich ist die Simulation von Warteschlangen die wesentliche Anwendung von Prioritätswarteschlangen. Viele Algorithmen wenden Warteschlangen mehr oder minder explizit an wie wir im nächsten Abschnitt sehen werden.

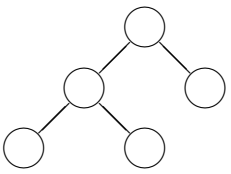
Wir implementieren Prioritätswarteschlangen durch die Datenstruktur „Heap“. Ein Heap ist ein Binärbaum mit Heap-Struktur, der Prioritäten gemäß einer Heap-Ordnung abspeichert.

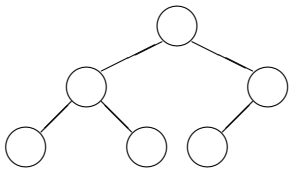
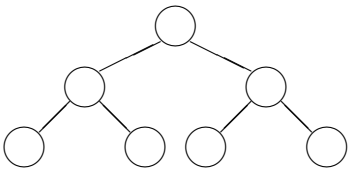
Definition 4.4 Ein geordneter binärer Baum T (der Tiefe t) hat **Heap-Struktur**, falls

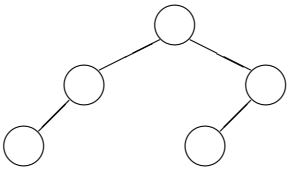
(a) jeder Knoten der Tiefe höchstens $t-2$ genau 2 Kinder hat,

(b) wenn ein Knoten v der Tiefe $t-1$ weniger als 2 Kinder hat, dann haben alle Knoten der Tiefe $t-1$, die rechts von v liegen, kein Kind.

(c) wenn ein Knoten v der Tiefe $t-1$ genau ein Kind hat, dann ist dieses Kind ein linkes Kind. (Beachte dass, Eigenschaft (b) erzwingt, dass nur ein Knoten genau ein Kind hat.)

Der Baum  hat Heap-Struktur ebenso wie

 und  .

Der Baum  hingegen hat keine Heap-Struktur.

Wir werden binäre Bäume mit Heapstruktur benutzen, um unsere Daten zu verwalten. Dazu speichert jeder Knoten seine Priorität. Die Zuordnung der Prioritäten muss allerdings sehr sorgfältig geschehen:

Definition 4.5 Sei T ein geordneter binärer Baum mit Heap-Struktur, wobei jedem Knoten v die Priorität $p(v)$ zugewiesen sei. Wir sagen, dass T **Heap-Ordnung** besitzt, falls für jeden Knoten v und für jedes Kind w von v

$$p(v) \geq p(w)$$

gilt.

Offensichtlich bedeutet die Heap-Ordnung, dass die Wurzel die maximal Priorität speichert. Wir kommen also sehr einfach an die maximale Priorität „ran“, eine Vorbedingung für eine effiziente Implementierung von delete_max.

Bevor wir die einzelnen Operationen implementieren, suchen wir eine geeignete Darstellung für Bäume mit Heap-Struktur. Wir wählen eine Version des Eltern-Arrays, die für Bäume mit Heap-Struktur unschlagbar sein wird was Effizienz und Einfachheit der Kodierung anbelangt.

Definition 4.6 T sei ein geordneter binärer Baum, wobei jedem Knoten v die Priorität $p(v)$ zugewiesen sei. T besitze Heap-Struktur und Heap-Ordnung. Wir sagen, dass das Array H ein **Heap** für T ist, falls

- $H[1] = p(r)$, wobei r die Wurzel von T sei.
- Wenn $H[i]$ die Priorität des Knotens v speichert, und v_L (bzw. v_R) das linke (bzw. rechte) Kind von v ist, dann gilt

$$H[2 * i] = p(v_L) \quad \text{und} \quad H[2 * i + 1] = p(v_R).$$

Aufgabe 57
Gegeben sei das folgende Array:

int H[21] = {0, 38, 35, 28, 19, 29, 23, 25, 11, 17, 27, 14, 20, 2, 23, 1, 9, 3, 18, 15, 24};

($H[0]$ erhält den Dummy-Wert 0.) Besitzt der durch H definierte Binärbaum Heap-Ordnung?

Aufgabe 58
Sei j die Arrayposition eines Heapelements. An welcher Arrayposition steht der Vorfahre in Abstand d von j? Warum?

(Der Vorfahre im Abstand 1 (bzw. Abstand 2) ist der Elternknoten (bzw. Großelternknoten).)

Aufgabe 59

(a) Wir nehmen stets an, dass ein Baum B mit Heap-Struktur und Heap-Ordnung gegeben ist und dass keine zwei Prioritäten übereinstimmen. **Beweise** oder **widerlege** die folgenden Aussagen durch ein Gegenbeispiel:

(i) Die kleinste Priorität wird stets von einem Blatt gespeichert.

(ii) Die zweitkleinste Priorität wird stets von einem Blatt gespeichert.

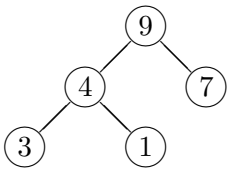
(iii) Die drittgrößte Priorität wird stets von einem Kind der Wurzel gespeichert.

(b) Wir nehmen zusätzlich an, dass B mindestens $n = 2^{t+1} - 2$ Knoten besitzt. Wird dann die t-größte Priorität *nie* von einem Blatt gespeichert?

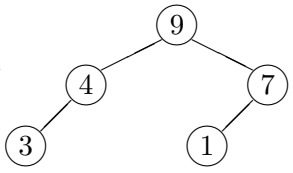
(c) Ein Heap-Array H mit n ganzzahligen Prioritäten ist vorgegeben, ebenso wie eine ganze Zahl x. **Beschreibe** einen Algorithmus in Pseudocode, der alle Prioritäten von H ausgibt, die mindestens so groß wie x sind. Wenn es K solche Prioritäten gibt, dann sollte die Laufzeit $O(K)$ erreicht werden.

Aufgabe 60
Gegeben sei ein Heap H der Größe 32, wobei alle Prioritäten des Heaps verschieden sind. Welche Positionen können von der drittgrößten Priorität in H eingenommen werden? Welche Positionen können von der drittkleinsten Priorität in H nicht eingenommen werden?

Man beachte, dass Heaps maßgeschneidert für Bäume mit Heap-Struktur sind: Nur Bäume mit Heap-Struktur besitzen „Heaps ohne Löcher“.

Beispiel 4.8 

besitzt den Heap (9,4,7,3,1).

Der Baum 

verletzt die Heapstruktur und sein „Heap“

(9,4,7,3, ,1) weist prompt ein Loch auf.

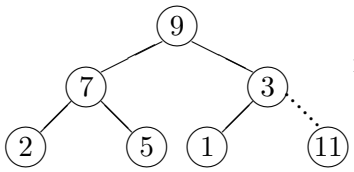
Wir können jetzt darangehen, die einzelnen Operationen zu implementieren. Wir beginnen mit der Operation

void insert (int p);

Wohin mit der Priorität p? Wenn unser gegenwärtiger Heap n Prioritäten speichert, dann legen wir p auf der ersten freien Position ab, setzen also

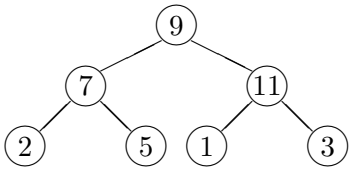
$$H[++n] = p;$$

Beispiel 4.9

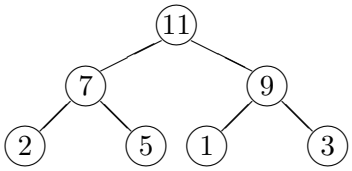


nach insert (11).

Heap-Struktur ist weiterhin gegeben, aber die Heap-Ordnung kann verletzt sein. Was tun? Vertausche Pioritäten, wenn notwendig.



Der Vertauschungsschritt repariert die Heap-Ordnung am Elternknoten des vorigen Blattes, aber jetzt kann die Heap-Ordnung am Großelternknoten verletzt sein! Anscheinend haben wir keinen Fortschritt erzielt? Doch, „das Problem rutscht nach oben“. Wenn wir die Wurzel erreicht haben, kann es keine Probleme mehr geben.



und fertig.

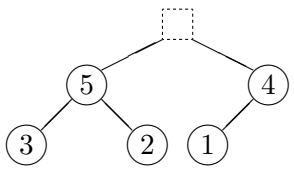
Wir nennen dieses Vorgehen *repair_up*.

Kümmern wir uns als Nächstes um die Prozedur

int delete_max ( );

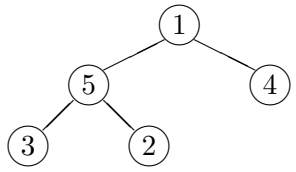
delete_max entfernt das Maximum und stellt sodann Heap-Struktur und Heap-Ordnung wieder her.

Beispiel 4.10

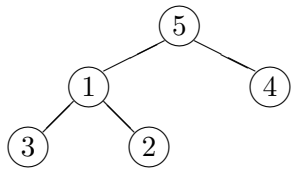


nach Entfernen des Maximums.

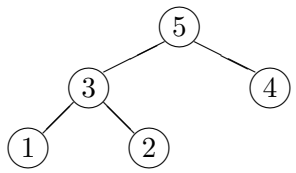
Zuerst schließen wir das entstandene Loch ein und für alle mal: Das letzte Heap-Element wird auf die Wurzel gesetzt.



Leider kann damit die Heap-Ordnung an der Wurzel verletzt sein. Naja, dann repariere man halt: Es bleibt nichts anderes übrig, als die Priorität der Wurzel mit der Priorität des „größten“ Kindes zu vertauschen.



Aber damit können wir uns eine weitere Verletzung der Heap-Ordnung einhandeln: Kein Problem, wir wenden das alte Rezept an: Vertausche die Priorität mit der Priorität des größten Kindes (wenn diese größer ist.)



Diesmal machen wir Fortschritte, weil „das Problem nach unten rutscht“. Wir nennen dieses Vorgehen *repair_down.*

Wir können eine Zwischenbilanz ziehen: insert und delete_max laufen in Zeit $O(\text{Tiefe}(T))$. Betrachten wir als Nächstes die Operation

void change_priority (int *wo*, int *p*);

wo spezifiziert den Index des Heaps H, dessen Priorität auf p zu setzen ist. Also setze:

$$H[wo] = p;$$

Pech gehabt, wir verletzen möglicherweise die Heap-Ordnung! Aber die Reparatur ist einfach:

- wenn p größer als die ursprüngliche Priorität ist, dann starte repair_up in *wo*.
- wenn p kleiner als die ursprüngliche Priorität ist, dann starte repair_down in *wo*.

Nun zur letzten Operation

void remove(int *wo*);

Was tun? Wir ersetzen H[*wo*] durch H[*n*]: Also

H[*wo*] = H[*n*- -];

Die möglicherweise verletzte Heap-Ordnung reparieren wir wie für change_priority.

Ergebnis 1: Alle vier Operationen laufen in Zeit $O(\text{Tiefe}(T))$.
Wie groß ist aber die Tiefe eines Baumes T mit n Knoten, falls T Heap-Struktur besitzt?

Wenn T die Tiefe t besitzt, dann hat T mindestens

$$1 + 2^1 + 2^2 + \cdots + 2^{t-1} + 1 = 2^t$$

Knoten, aber nicht mehr Knoten als

$$1 + 2^1 + 2^2 + \cdots + 2^{t-1} + 2^t = 2^{t+1} - 1.$$

Mit anderen Worten,

$$2^{Tiefe(T)} \leq n < 2^{Tiefe(T)+1}.$$

Ergebnis 2: Tiefe $(T) = \lfloor \log_2 n \rfloor$. Alle vier Operationen werden somit in logarithmischer Zeit unterstützt!

Was kann man so alles mit Heaps anstellen? Wir können zum Beispiel ein Array $(A[1], \ldots, A[n])$ sehr schnell sortieren:

Algorithmus 4.11 **(Heapsort)**

```
for (i=1; i <= n ; i++)
  insert(A[i]);
//Die Elemente von A werden nacheinander in den Heap H eingefuegt.
int N = n;
for (n=N; n >= 1 ; n--)
  H[n+1] = delete_max( );
//Der Heap H ist jetzt aufsteigend sortiert.
n = N;
```

Da die Operationen insert und delete_max jeweils in logarithmischer Zeit arbeiten, läuft Heapsort in Zeit $O(n \log n)$ und ist damit wesentlich schneller als eine elementare Sortierprozedure wie Bubble Sort.

Wir können Heapsort sogar noch beschleunigen. Die Idee ist, zuerst den Heap zu laden ohne Reparaturen durchzuführen, das heißt, wir „erklären“ das Eingabearray H zu einem „Fast-Heap“.

Dann führen wir eine groß-angelegte Reparatur von den Blättern an aufwärts durch. Jedes Blatt ist schon ein Heap und keine Reparatur ist notwendig. Wenn t die Tiefe des Heaps ist, kümmern wir uns als Nächstes um Knoten der Tiefe $t-1$. Wenn v ein solcher Knoten ist, dann betrachte den Teilbaum T_v mit Wurzel v. T_v ist nur dann kein Heap, wenn die Heap-Ordnung im Knoten v verletzt ist: Repariere mit repair_down, gestartet in v. Beachte, dass repair_down nur höchstens einen Vertauschungsschritt durchführt.
Wenn wir sichergestellt haben, dass T_v ein Heap für jeden Knoten der Tiefe $t-1$ ist, dann kümmern wir uns um Knoten der Tiefe $t-2$. Wenn w ein solcher Knoten ist und T_w kein Heap ist, dann ist die Heap-Ordnung nur in w verletzt. Eine Reparatur gelingt mit repair_down und höchstens 2 Vertauschungsschritten. Diese Prozedur wird für Knoten der Tiefen $t-3, t-4, \ldots, 0$ wiederholt.

Wir nennen diese Prozedur *build-heap*. Wieviel Zeit benötigt build-heap, wenn ein Heap von n Prioritäten zu bauen ist? Wir zählen nur die worst-case Anzahl der Vertauschungsschritte:

Es gibt 2^{t-i} Knoten der Tiefe $t-i$ (für $i \geq 1$). Für jeden dieser Knoten sind höchstens i Vertauschungsschritte durchzuführen. Also benötigen wir höchstens

$$\sum_{i=1}^{t} i2^{t-i} = 2^{t+1} - t - 2$$

Vertauschungsschritte. Warum gilt die obige Identität? Wir verifizieren induktiv

$$\begin{aligned}
\sum_{i=1}^{t+1} i2^{t+1-i} &= 2\sum_{i=1}^{t} i2^{t-i} + t + 1 \\
&= 2 \cdot (2^{t+1} - t - 2) + t + 1 \\
&= 2^{t+2} - (t+1) - 2.
\end{aligned}$$

□

Mit vollständiger Induktion, lassen sich solche Identitäten immer recht schnell und bequem nachweisen. Wie aber kommt man auf einen geschlossenen Ausdruck wie den obigen, wenn man ihn nicht in irgendeiner Formelsammlung findet? Wir zeigen auch dies hier kurz auf. Nehmen wir also an, wir haben nur unsere Summe und keine Ahnung, was herauskommt:

$$\begin{aligned}
\sum_{i=1}^{t} i2^{t-i} &= \sum_{i=0}^{t-1} (t-i)2^i \qquad \text{durch Rückwärtssummation} \\
&= \sum_{i=0}^{t-1} t2^i - i2^i \\
&= (\sum_{i=0}^{t-1} t2^i) - (\sum_{i=0}^{t-1} i2^i) \\
&= (t \cdot \sum_{i=0}^{t-1} 2^i) - (\sum_{i=0}^{t-1} i2^i) \\
&= (t \cdot (2^t - 1)) - (\sum_{i=0}^{t-1} i2^i)
\end{aligned}$$

Ein Teilerfolg, denn der linke Summand ist jetzt bekannt. Die rechte Summe nehmen wir uns jetzt separat vor und wenden einen netten, kleinen Kunstgriff an. Wir ersetzen die 2 durch

eine Variable a. Das erlaubt uns, den Ausdruck als eine Funktion in a zu bearbeiten.

$$\begin{aligned}
\sum_{i=0}^{t-1} ia^i &= \sum_{i=1}^{t-1} ia^i && \text{der } (i=0)\text{-Summand ist 0} \\
&= a \cdot \sum_{i=1}^{t-1} ia^{i-1} \\
&= a \cdot \sum_{i=1}^{t-1} \frac{\partial}{\partial a} a^i && \text{man leite zur Kontrolle nach } a \text{ ab} \\
&= a \cdot \frac{\partial}{\partial a} (\sum_{i=1}^{t-1} a^i) && \text{die Summe der Ableitungen ist die Ableitung der Summe} \\
&= a \cdot \frac{\partial}{\partial a} (\frac{a^t - 1}{a-1} - 1) && \text{siehe Lemma 2.9} \\
&= a \cdot \frac{ta^{t-1}(a-1) - (a^t - 1)}{(a-1)^2} && \text{Quotientenregel} \\
&= 2 \cdot \frac{t2^{t-1} - 2^t + 1}{1} && \text{Wir haben } a = 2 \text{ gesetzt} \\
&= t2^t - 2^{t+1} + 2.
\end{aligned}$$

Wir fassen unsere beiden Summen wieder zusammen und erhalten:

$$\begin{aligned}
\sum_{i=1}^{t} i2^{t-i} &= (t \cdot (2^t - 1)) - (t2^t - 2^{t+1} + 2) \\
&= (t2^t - t) - t2^t + 2^{t+1} - 2 \\
&= 2^{t+1} - t - 2.
\end{aligned}$$

□

Beachte, dass $2^t \leq n$ und damit werden höchstens

$$2n - \log n - 2 \leq 2n$$

Vertauschungsschritte durchgeführt.

Satz 4.11 *(a) Ein Heap mit n Prioritäten unterstützt jede der Operationen insert, delete_max, change_priority und remove in Zeit $O(\log_2 n)$.*

(b) Build-heap baut einen Heap mit n Prioritäten in Zeit $O(n)$.

(c) Heapsort sortiert n Zahlen in Zeit $O(n \log_2 n)$.

Wir schließen mit einer $C{+}{+}$ Implementierung von Heaps. Zuerst definieren wir eine Klasse Heap.

```
class heap{
  private:
    int *H; // H ist der Heap.
    int n;  // n bezeichnet die Groesse des Heaps.
    void repair_up (int wo);
```

```
    void repair_down (int wo);
  public:
    heap (int max) // Konstruktor.
     {H = new int[max]; n = 0;}

    int read (int i) {return H[i];}
    void write (int i){ H[++n] = j;}

    void insert (int priority);
    int  delete_max( );
    void change_priority (int wo, int p);
    void remove(int wo);
    void buildheap();
    void heapsort();
};
```

Die Funktion *write* wird benutzt, um den Heap für die Funktion *buildheap* zu initialisieren. *heapsort* wird *buildheap* aufrufen und dann den Heap durch wiederholtes Aufrufen der Funktion *delete_max* sortieren. Der sortierte Heap kann dann durch die Funktion *read* ausgelesen werden. Wir schließen mit einer Beschreibung der zentralen Prozeduren *repair_up* und *repair_down*.

Algorithmus 4.12 **(Heaps: Die Aufwärts-Reparatur)**

```
void heap::repair_up (int wo)
  {
    int p = H[wo];
    while ((wo > 1) && (H[wo/2] < p))
      {H[wo] = H[wo/2];
          wo = wo/2;}
    H[wo] = p;
  }
```

Wir haben hier benutzt, dass der Elternknoten von wo den Index $wo/2$ hat. Weiterhin reparieren wir solange, bis entweder die Wurzel erreicht ist ($wo = 1$) oder die Heap-Ordnung wiederhergestellt ist ($H[wo/2] >= p$).

Algorithmus 4.13 **(Heaps: Die Abwärts-Reparatur)**

```
void heap::repair_down (int wo)
  {
    int kind; int p = H[wo];
    while (wo <= n/2)
     {
       kind = 2 * wo;
       if ((kind < n) && (H[kind] < H[kind + 1])) kind ++;
       if (p >= H [kind]) break;
       H[wo] = H[kind]; wo = kind;
     }
    H[wo] = p;
  }
```

In der while Schleife wird das größte Kind ermittelt. Wenn nötig wird eine Vertauschungsoperation durchgeführt; ansonsten wird die while-Schleife verlassen. Beachte, dass wo genau dann ein Blatt ist, wenn $2 * wo > n$, das heißt, wenn $wo > n/2$.

Aufgabe 61
In einen anfänglich leeren Heap werden die Prioritäten $p_1, \ldots, p_i, \ldots$ durch die Funktion *insert* eingefügt. Wir nennen p_i die i-te Priorität.

Beschreibe eine erweiterte Version des abstrakten Datentyp heap. In dieser Erweiterung ist die zusätzliche öffentliche Funktion

```
int adresse(int i);
```

zur Verfügung zu stellen. adresse(i) soll für die i-te Prioritität die Position dieser Priorität im Heap berechnen; wenn die i-te Priorität nicht mehr vorhanden ist, ist 0 auszugeben.

Um addresse(i) schnell berechnen zu können (Laufzeit $O(1)$ ist möglich), müssen Adressenveränderungen (bei den beteiligten Funktionen der Klasse *heap*) schnell vermerkt werden. **Beschreibe** welche Funktionen der Klasse *heap* wie geändert werden müssen.

Aufgabe 62
Bisher haben wir nur binäre Heaps betrachtet: also die Array-Implementierung binärer Bäume mit Heap-Struktur und Heap-Ordnung. In dieser Aufgabe betrachten wir d-äre Heaps (für $d \geq 2$).

Verallgemeinere die Begriffe *Heap-Struktur* und *Heap-Ordung* von binären Bäumen auf d-äre Bäume. Wie sollte die Array-Implementierung von d-ären Bäumen mit Heap-Struktur und Heap-Ordnung aussehen; insbesondere wie kann man die Position des Elternknotens und des i-ten Kindes einfach berechnen?

Aufgabe 63
Gib eine möglichst exakte obere Schranke für die Tiefe eines d-ären Heaps mit n Knoten **an**.

Aufgabe 64
Beschreibe effiziente Algorithmen für die Operationen *insert* und *delete_max* für d-äre Heaps und **bestimme** die Laufzeit für d-äre Heaps mit n Knoten.

4.6 Datenstrukturen und Algorithmen

Wir beschreiben Dijkstra's Algorithmus zur Berechnung kürzester Wege und die Algorithmen von Prim und Kruskal zur Berechnung minimaler Spannbäume. Wir beschränken uns jeweils auf die Implementierung der Algorithmen, Korrektheitsbeweise werden in der Vorlesung „Algorithmentheorie“ vorgestellt.

4.6.1 Dijkstra's Single-Source-Shortest Path Algorithmus

Wir nehmen an, dass ein gerichteter Graph $G = (V, E)$ zusammen mit einer Längen-Zuweisung länge: $E \to \mathbb{R}_{\geq 0}$ an die Kanten des Graphen gegeben ist. Für einen ausgezeichneten Startknoten $s \in V$ sind kürzeste Wege von s zu allen Knoten von G zu bestimmen, wobei die Länge eines Weges die Summe seiner Kantengewichte ist.

Mit Hilfe der Breitensuche können wir kürzeste-Wege Probleme lösen, falls länge$(e) = 1$ für jede Kante $e \in E$ gilt; für allgemeine (nicht-negative) Längen brauchen wir aber ein stärkeres Geschütz.

Algorithmus 4.14 **(Dijkstras Algorithmus)**

(1) Für den Graphen $G = (V, E)$ setze $S = \{s\}$ und

$$\text{distanz}[v] = \begin{cases} \text{länge } (s, v) & \text{wenn} \quad (s, v) \in E \\ \infty & \text{sonst.} \end{cases}$$

/* distanz[v] ist die Länge des bisher festgestellten kürzesten Weges von s nach v. Insbesondere werden zu Anfang nur Kanten, also Wege der Länge 1 betrachtet. */

(2) Solange $S \neq V$ wiederhole

(a) wähle einen Knoten $w \in V - S$ mit kleinstem Distanz-Wert.
/* Dijkstra's Algorithmus setzt darauf, dass distanz[w] mit der Länge eines tatsächlich kürzesten Weges von s nach w übereinstimmt. Glücklichreweise ist dies stets der Fall, wenn alle Kantenlängen nicht-negativ sind. */

(b) Füge w in S ein.

(c) Aktualisiere die Distanz-Werte der Nachfolger von w. Insbesondere setze für jeden Nachfolger $u \in V \setminus S$ von w:

```
            c = distanz[w] + laenge(w,u);
distanz[u] = (distanz[u] > c) ? c : distanz[u];
```

In der Vorlesung „Algorithmentheorie“ wird gezeigt, dass Dijkstra's Algorithmus korrekt ist und das kürzeste-Wege Problem effizient löst. Wir kümmern uns hier nur um seine Implementierung.

Offensichtlich sollten wir G als Adjazenzliste implementieren, da wir dann sofortigen Zugriff auf die Nachfolger u von w in Schritt (2c) haben. Kritisch ist die Implementierung der Menge $V \setminus S$: Wir wählen eine modifizierte Prioritätswarteschlange, wobei wir die Funktion `delete_max()` durch die Funktion `delete_min()` ersetzen. In Schritt (1) wird die Prioritätswarteschlange durch die `insert()` Operation gefüllt. Schritt (2a) kann dann durch einen Aufruf von `delete_min()` implementiert werden. Im Schritt (2c) müssen wir möglicherweise die Priorität eines Nachfolgers von w reduzieren; dies erreichen wir durch die Operation `change_priority(wo,c)`. Allerdings, woher kennen wir die Position wo?

Aufgabe 65
Welche Datenstruktur sollte für die Menge S gewählt werden? Die Datenstruktur sollte für jeden Knoten v die Bestimmung eines kürzesten Weg von s nach v in Zeit proportional zur Kantenzahl des kürzesten Weges erlauben.

4.6.2 Prim's Algorithmus

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Ein Baum $T = (V', E')$ heißt ein **Spannbaum** für G, falls $V' = V$ und $E' \subseteq E$. Ein Spannbaum für G ist also ein Teilgraph von G, der gleichzeitig ein Baum ist. Beachte, dass zwei je Knoten von G in einem Spannbaum verbunden bleiben, denn Bäume sind zusammenhängend. Spannbäume garantieren also weiterhin alle Verbindungen, aber tun dies mit einer kleinstmöglichen Kantenzahl.

Wir nehmen jetzt zusätzlich an, dass die Kanten in E durch eine Längenfunktionen länge : $E \to \mathbb{R}$ gewichtet sind. Wir interpretieren die Länge einer Kante als die Kosten der Kante und definieren

$$\text{Länge}(T) = \sum_{e \in E'} \text{länge}(e)$$

als die Kosten von T. Unser Ziel ist die Berechnung eines **minimalen Spannbaums** T, also eines Spannbaums minimaler Länge unter allen Spannbäumen von G.

Als ein Anwendungsbeispiel betrachte das Problem des Aufstellens eines Kommunikationsnetzwerks zwischen einer Menge von Sites. Jede Site ist im Stande, Nachrichten, die es von anderen Sites erhält, potentiell an alle Nachbarn weiterzugeben. Für die Verbindungen zwischen den Sites müssen Kommunikationsleitungen gekauft werden, deren Preis von der Distanz der Sites abhängt. Es genüge aber, soviele Leitungen zu kaufen, dass je zwei Zentralen indirekt (über andere Sites) miteinander kommunizieren können. Wir repräsentieren jede Site durch einen eigenen Knoten und verbinden je zwei Sites mit einer Kante, die wir mit der Distanz zwischen den Sites markieren. Wenn G der resultierende Graph ist, dann entspricht ein billigstes Kommunikationsnetzwerk einem minimalen Spannbaum für G!

Der Algorithmus von Prim lässt einen minimalen Spannbaum, beginnend mit dem Knoten 1, wachsen. Wenn gegenwärtig alle Knoten in der Teilmenge $S \subseteq V$ überdeckt werden, dann sucht Prim's Algorithmus nach einer **kürzesten kreuzenden Kante** e: Ein Endpunkt von e muss zur Menge S und der verbleibende Endpunkt muss zur Menge $V \setminus S$ gehören. Desweiteren muss e minimale Länge unter allen kreuzenden Kanten besitzen. Prim's Algorithmus fügt e in seinen Baum ein und wiederholt diese Schritte solange, bis alle Knoten in V überdeckt sind. Wir werden in der Vorlesung „Algorithmentheorie“ nachweisen, dass dieses Verfahren korrekt ist, hier implementieren wir den Algorithmus.

Algorithmus 4.15 **(Algorithmus von Prim)**

(1) Setze $S = \{1\}$. T ist ein Baum mit Knotenmenge S, der zu Anfang nur aus dem Knoten 1 besteht.

(2) Solange $S \neq V$, wiederhole:

   (a) Bestimme eine kürzeste kreuzende Kante $e = \{u, v\}$.

   (b) Füge e zu T hinzu.

   (c) Wenn $u \in S$, dann füge v zu S hinzu. Ansonsten füge u zu S hinzu.

Die zentrale Operation ist die Suche nach einer kürzesten Kante, die einen Knoten in S mit einem Knoten in $V \setminus S$ verbindet. Sollten wir alle Kanten mit einer Prioritätswarteschlange verwalten, wobei wir wieder die Funktion `delete_max()` durch die Funktion `delete_min()` ersetzen? Nein, denn wenn wir die Menge S um einen Knoten u vergrößern, dann müssen auch alle Kanten $\{u, w\}$ mit $w \in S$ entfernt werden, da alle diese Kanten nicht mehr kreuzen. Um diesen zu großen Aufwand zu umgehen, verwalten wir die Knoten in $V \setminus S$ mit einer Prioritätswarteschlange: Wir initialisieren die Prioritätswarteschlange, indem wir jeden Knoten $w \neq 1$ mit Priorität länge($\{1, w\}$) einfügen. Wenn ein Knoten u in Schritt (2) zu S hinzugefügt wird, wenn also $u =$ `deletemin`() gilt, dann ändern sich höchstens die Prioritäten der Nachbarn von u und wir wenden `change_priority(wo, neue Priorität)` an. Beachte, dass wir auch diesmal den Graph G durch eine Adjazenz-Liste implementieren sollten, damit wir alle Nachbarn schnell bestimmen können.

Aufgabe 66

Im *Independent Set* Problem ist ein ungerichteter Graph mit Knotenmenge $V = \{1, \ldots, n\}$ durch seine Adjazenzliste A gegeben. ($A[i]$ ist also ein Zeiger auf die Liste der Nachbarn von Knoten i, also der Knoten, die mit i durch eine Kante verbunden sind.)

Gesucht ist eine unabhängige Menge größter Mächtigkeit. (Eine unabhängige Menge W ist eine Teilmenge der Knotenmenge V, so dass keine zwei Knoten in W durch eine Kante verbunden sind. Das Independent Set Problem modelliert also die Bestimmung größter „kollisionsfreier“ Knotenmengen.)

Für dieses Problem sind effiziente Lösungen nicht bekannt und man arbeitet deshalb auch mit Heuristiken, die hoffentlich gute, aber im Allgemeinen suboptimale Antworten liefern. Eine solche Strategie ist die Heuristik des *minimalen Grads*.

```
W = leer; while  (V nicht leer)
  {Bestimme einen Knoten w in V mit minimalem Grad;
   /* Der Grad eines Knotens w in V ist die Anzahl seiner Nachbarn in V */
   Entferne w und alle Nachbarn von w aus V;
   Fuege w zu W hinzu;};
```

(a) Zeige durch ein Gegenbeispiel, dass die Heuristik des *minimalen Grads* im Allgemeinen keine größtmögliche Knotenmenge findet.

(b) Implementiere die Heuristik mit einer Laufzeit von $O((n+m)\cdot \log n))$ für Graphen mit n Knoten und m Kanten. Beschreibe die benutzten Datenstrukturen im Detail und benutze ansonsten nur Pseudocode.

4.6.3 Die Union-Find Datenstruktur und Kruskal's Algorithmus

Während Prim's Algorithmus einen minimalen Spannbaum Kante für Kante durch das Einfügen minimaler kreuzender Kanten wachsen lässt, beginnt Kruskal's Algorithmus mit einem Wald[1] von Einzelknoten. Danach versucht Kruskal's Algorithmus, die Kanten gemäß aufsteigender Länge in den Wald einzufügen: Die Einfügung gelingt, wenn die jeweilige Kante zwei verschiedene Bäume des Walds verbindet, die Einfügung wird verworfen, wenn die Kante einen Kreis in einem der Bäume schließt.

Algorithmus 4.16 **(Kruskals Algorithmus)**

(1) Sortiere die Kanten gemäß aufsteigendem Längenwert. Sei $W = (V, F)$ der leere Wald, also $F = \emptyset$.

(2) Solange W kein Spannbaum ist, wiederhole

  (a) Nimm die gegenwärtige kürzeste Kante e und entferne sie aus der sortierten Folge.

  (b) Verwerfe e, wenn e einen Kreis in W schließt.

  (c) Ansonsten akzeptiere e und setze $F = F \cup \{e\}$.

Wir beschränken uns auf die Implementierung von Kruskal's Algorithmus und verweisen für den Korrektheitsbeweis auf die Vorlesung „Algorithmentheorie".

Wir können die Kanten mit Heapsort sortieren. Danach aber müssen wir in jeder Iteration für eine Kante $e = \{u, v\}$ entscheiden, ob u und v verschiedenen Bäumen des Waldes angehören. Hier liegt es nahe, die Wurzeln der Bäume von u und v zu bestimmen und auf Gleichheit zu überprüfen. Wir führen deshalb die Operation `find`(x) ein, um die Wurzel des Baums von x auszugeben.

Wir wählen die denkbar einfachste Repräsentation des Waldes W, nämlich die Eltern-Repräsentation. Da wir mit einem leeren Wald beginnen, setzen wir zu Anfang

$$\text{Eltern}[i] = i \quad \text{für} \quad i = 1, \ldots, |V|.$$

(Wir fassen i immer dann als eine Wurzel auf, wenn $\text{Eltern}[i] = i$ gilt.) Einen find-Schritt führen wir aus, indem wir den Baum mit Hilfe des Eltern-Arrays hochklettern bis die Wurzel gefunden ist. Damit benötigt ein find-Schritt Zeit höchstens proportional zur Tiefe des jeweiligen Baumes. Wie garantieren wir, dass die Bäume nicht zu tief werden?

Wenn wir die Bäume durch das Hinzufügen der von Kruskal's Algorithmus gefundenen Kanten vereinigen, dann haben wir keine Kontrolle über das Tiefenwachstum. Stattdessen vereinigen

[1] Ein Wald ist eine knotendisjunkte Vereinigung von Bäumen.

wir zwei Bäume mit den Wurzeln i und j „eigenmächtig“ mit der Operation `union`(i, j); natürlich halten wir die Menge der gefundenen Kanten und damit den berechneten minimalen Spannbaum in einem weiteren Eltern-Array fest.

Wie sollte `union`(i, j) verfahren? Hänge die Wurzel des kleineren Baumes unter die Wurzel des größeren Baumes! Betrachten wir einen beliebigen Knoten v. Seine Tiefe vergrößert sich nur dann um 1, wenn v dem kleineren Baum angehört. Das aber heißt,

> die Tiefe von v vergrößert sich nur dann um 1, wenn sein Baum sich in der Größe mindestens verdoppelt.

Dann ist aber die Tiefe aller Bäume durch $\log_2(|V|)$ beschränkt. Also läuft ein union-Schritt in konstanter Zeit, während ein find-Schritt höchstens logarithmische Zeit benötigt.

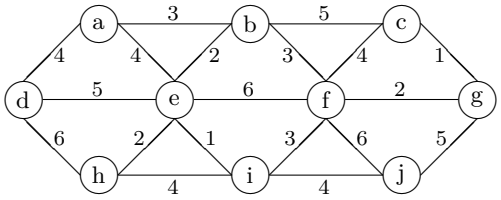
Satz 4.12 *Sei $G = (V, E)$ ein ungerichteter Graph. Dann berechnet Kruskal's Algorithmus einen minimalen Spannbaum in Zeit*

$$O(|E| \cdot \log_2 |V|).$$

Der Algorithmus von Kruskal benutzt den abstrakten Datentyp `Union-Find` mit den Operationen „Union“ und „Find“. Unsere Datenstruktur ist schon recht schnell, kann aber noch weiter zu einer fast linearen Laufzeit beschleunigt werden: Wenn wir das Sortieren nicht in der Laufzeit berücksichtigen, dann ist die Laufzeit asymptotisch sogar noch kleiner als $|E| \cdot \log_2^*(n)$. (Die Log-Stern Funktion wird in Definition 3.2 eingeführt.)

Aufgabe 67
Bestimme die Laufzeit von Kruskal's Algorithmus, wenn wir die Laufzeit für das Sortieren der Kanten nach aufsteigender Länge *nicht* miteinbeziehen.

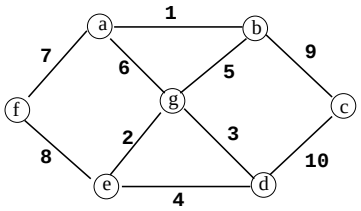
Aufgabe 68
Gegeben sei der folgende gewichtete, ungerichtete Graph G:



(a) Benutze den Algorithmus von Kruskal, um einen minimalen Spannbaum für G zu konstuieren. **Gib** die Reihenfolge der vom Algorithmus gefundenen Kanten **an**.

(b) Ist der minimale Spannbaum für obigen Graphen G eindeutig? **Begründe** deine Antwort!

Aufgabe 69
Gegeben sei der folgende gewichtete, ungerichtete Graph G:



Gib die *Reihenfolgen* der von Kruskal's und von Prim's Algorithmen gefundenen Kanten **an**. Wähle Knoten a als Startknoten für Prims Algorithmus.

4.7 Zusammenfassung

Wir haben uns zuerst mit den elementaren Datenstrukturen Liste, Deque, Stack, Queue und Bäumen beschäftigt.

Listen bieten die Möglichkeit, sich dynamisch an die Größe der Datenmenge anzupassen. Die Matrizenaddition diente als ein erstes Beispiel. Ein zweites Anwendungsbeispiel haben wir in Form der Adjazenzliste im Problem des topologischen Sortierens angetroffen.

Listen, Deques, Stacks und Queues „beherrschen“ einfache Formen des Einfügens und Entfernens von Daten: Einfügen/Entfernen an vorher spezifizierten Stellen (am linken oder rechten Ende sowie durch Zeiger explizit spezifiziert). Diese einfachen Formen werden in konstanter Zeit unterstützt. Für komplizierte Probleme, wie das Einfügen von Daten unter Beibehaltung einer sortierten Reihenfolge, werden wir später kompliziertere Datenstrukturen entwickeln müssen.

Bäume bieten sich als Datenstruktur an, wenn Daten hierarchisch zu ordnen sind. Wir haben verschiedene Implementierungen von Bäumen betrachtet und verglichen in Hinsicht auf ihre Speichereffizienz und in Hinsicht auf eine effiziente Implementierung wichtiger Operationen. Zu den wichtigeren dieser Operationen zählen

> die Eltern-Operation, die Kinder-Operation, die Wurzel-Operation, sowie die Operation des „Baumbauens“ aus Teilbäumen.

Die Binärbaum-Implementierung sowie die Kind-Geschwister-Implementierung stellten sich als Gewinner heraus (notfalls zusätzlich mit einem Eltern-Array ausgestattet).

Wir haben weiterhin die Präorder-, Postorder- bzw. die Inorder-Reihenfolge als Suchmethoden für Bäume beschrieben und rekursive wie auch nicht-rekursive Implementierungen besprochen. Eine erste Möglichkeit einer nicht-rekursiven Implementierung besteht im Verfolgen zusätzlich eingesetzter Zeiger, die eine „Fädelung“ bilden. Eine zweite, speicher-effizientere Methode ist die Simulation der Rekursion mit Hilfe eines Stacks.

Mit Prioritätswarteschlangen haben wir dann eine erste mächtige Datenstruktur kennengelernt: Die Operationen insert, delete_max, remove und change_priority werden unterstützt, wobei remove und change_priority Zeiger auf das betroffene Element benötigen. Mit der Heap-Implementierung laufen alle vier Operationen in logarithmischer Zeit. Da wir mit den Operationen insert und delete_max sortieren können, erhalten wir die Sortiermethode Heapsort: n Zahlen werden in Zeit $O(n \log n)$ sortiert.

Die bisher betrachteten Datenstrukturen liefern eine effiziente Implementierung wichtiger Operationen wie wir zuerst in der Implementierung eines Algorithmus für das topologische Sortieren gesehen haben. Desweiteren können wir die für Graph-Algorithmen wichtigen Nachbar- bzw. Nachfolger-Operationen mit Adjazenzlisten effizient unterstützen, und wir haben deshalb Adjazenzlisten sowohl für die Implementierung von Tiefen- und Breitensuche wie auch für die Implementierung der Algorithmen von Dijkstra, Prim und Kruskal benutzt. Heaps waren die wesentlichen zusätzlichen Bausteine in der Implementierung der Algorithmen von Dijkstra und Prim, während wir eine `union-find` Datenstruktur mit Hilfe der Eltern-Darstellung von Bäumen für Kruskal's Algorithmus angewandt haben.

Im nächsten Kapitel möchten wir eine uneingeschränkte delete-Operation, sowie eine Suchoperation effizient implementieren. Beachte, dass Heaps für das Suchproblem

„Ist Element p vorhanden?“

ungeeignet sind: möglicherweise muss der ganze Heap untersucht werden!

Kapitel 5

Das Wörterbuchproblem

Der abstrakte Datentyp **Wörterbuch** für eine Menge S besteht aus den folgenden Operationen:

- `insert`(x) : $S = S \cup \{x\}$.
- `remove`(x) : $S = S - \{x\}$.
- `lookup`(x) : finde heraus, ob $x \in S$ und greife gegebenfalls auf den Datensatz von x zu.

Beispiel 5.1 In Anwendungen des Wörterbuchproblems sind die Elemente häufig Strukturen der Form (Schlüssel, Info). Betrachten wir zum Beispiel eine Firmendatenbank.

Der Name eines Angestellten stellt den Schlüssel dar. Der Info-Teil kann aus Adresse, Telefonnummer, Gehalt etc. bestehen. Für die `lookup`-Operation wird der Schlüssel eingegeben und als Antwort wird der zugehörige Info-Teil erwartet. Für die `insert-Operation` wird Schlüssel und Info-Teil eingegeben und entweder neu eingefügt, oder, falls der Schlüssel schon vertreten ist, wird der alte mit dem neuen Info-Teil überschrieben. Bei der `remove`-Operation wird nur der Schlüssel eingegeben, der dann mitsamt seinem Info-Teil entfernt wird. (Wir werden deshalb im Folgenden der Konvention folgen, dass ein Schlüssel nur einmal vertreten sein darf.)

Wenn eine vollständige Ordnung auf den Daten definiert ist, dann können wir auch die folgenden Operationen betrachten:

- `select`(k) : bestimme den k-kleinsten Schlüssel.
- `rang`(x) : bestimme den Rang von x, wobei rang$(x) = k$ genau dann, wenn x der k-kleinste Schlüssel ist.
- `interval`(a,b) : bestimme, in aufsteigender Reihenfolge, die Schlüssel $y \in S$ mit $a \leq y \leq b$.

Während das Wörterbuch nur aus den ersten drei Operationen besteht, besteht der abstrakte Datentyp **geordnetes Wörterbuch** aus allen sechs Operationen. Zuerst beschäftigen wir uns mit Datenstrukturen für das geordnete Wörterbuchproblem. Dann beschreiben wir Hashing-Verfahren für das (einfachere) Wörterbuchproblem.

Effiziente Datenstrukturen für **statische Wörterbücher**[1] lassen sich leicht konstruieren. Z.B. können wir die Menge S durch das sortierte Array seiner Elemente darstellen. Diese Datenstruktur ist effizient: Das sortierte Array (von n Elementen) kann in Zeit $O(n \log_2 n)$ erstellt werden und ein lookup kann in logarithmischer Zeit durch binäre Suche implementiert werden.

Heaps wie auch Listen versagen als effiziente Datenstruktur für Wörterbücher, da die `lookup`-Operation ein Durchsuchen aller gespeicherten Schlüssel erfordert. Sortierte Arrays sind ideal für `lookup`, aber viel zu klobig für eine effiziente Unterstützung der `insert`- und der `remove`-Operation.

Beispiel 5.2 Suchmaschinen im Internet

Sogenannte *Spinnen* (oder Crawler) hangeln sich, den Links folgend, von Seite zu Seite. Erreicht ein solches Spinnenprogramm eine neue Seite, so wird es versuchen den dortigen Inhalt nach Stichworten und tragenden Begriffen zu durchsuchen. Wie diese Registrierung genau funktioniert ist von Suchmaschine zu Suchmaschine unterschiedlich.

Die Suchmaschine hat nun also sowohl die Menge der Stichworte als auch die zu einem Stichwort gespeicherten Seitenadressen zu verwalten. Insbesondere muss es möglich gemacht werden

- zu einem gegebenen Stichwort schnell die gespeicherten Seiten verfügbar zu haben, falls das Stichwort im Katalog ist,
- ein neues Stichwort aufzunehmen,
- die Adresse einer gefundenen Seite bei den betreffenden Stichworten abzulegen,
- eine Adresse zu löschen, falls die Seite von den Spinnen nicht mehr gefunden wird oder sich inhaltliche Änderungen vollzogen haben,
- ein Stichwort zu löschen, wenn die letzte Adresse zu diesem Stichwort entfernt wurde.

Wir greifen das Thema der Suchmaschinen im Abschnitt 5.6 wieder auf.

5.1 Binäre Suchbäume

Definition 5.1 T sei ein geordneter binärer Baum, so dass jedem Knoten v von T ein Paar daten(v) = (Schlüssel (v), Info (v)) zugewiesen sei. T heißt ein **binärer Suchbaum**, wenn T die folgenden Eigenschaften hat:

(a) Für jeden Schlüsselwert x gibt es höchstens einen Knoten v mit Schlüssel $(v) = x$.

(b) Für jeden Knoten v, jeden Knoten v_{links} im linken Teilbaum von v und jeden Knoten v_{rechts} im rechten Teilbaum von v gilt

$$\text{Schlüssel}(v_{links}) < \text{Schlüssel}(v) < \text{Schlüssel}(v_{rechts}).$$

[1] Der abstrakte Datentyp „statisches Wörterbuch“ besteht nur aus der Operation `lookup`(x).

Aufgabe 70
In dieser Aufgabe betrachten wir binäre Suchbäume, die genau die Zahlen $1 \ldots 7$ enthalten.

(a) Wieviele verschiedene solche Suchbäume gibt es, die zu linearen Listen degeneriert sind, d.h. Tiefe 6 haben? **Begründe** deine Antwort.

(b) Wieviele verschiedene solche Suchbäume der Tiefe 2 gibt es? **Beweise** deine Aussage.

(c) Wie viele verschiedene Einfügungsreihenfolgen für die 7 Elemente gibt es, die zu einem Baum wie in Aufgabenteil (b) führen? **Begründe** deine Antwort.

Vergleichen wir zuerst die Schlüsselanordnung in binären Suchbäumen mit der Schlüsselanordnung in Heaps. Während ein Heap den maximalen Schlüssel in der Wurzel speichert, wird der maximale Schlüssel im binären Suchbaum vom rechtesten Knoten gespeichert. Während das Suchen in Heaps schwierig ist, gelingt die Suche in binären Suchbäumen „sofort“ mit Hilfe der binären Suche: Es ist nicht verwunderlich, dass die Binärsuche ausschlaggebend für die Definition binärer Suchbäume war.

Die Implementierung der Operation *lookup*(x) ist jetzt einfach: Wenn wir am Knoten v angelangt sind (wobei anfangs v die Wurzel ist), vergleichen wir x mit Schlüssel (v):

- **Fall 1**: $x =$ Schlüssel (v) und wir haben den Schlüssel gefunden.
- **Fall 2**: $x <$ Schlüssel (v). Uns bleibt keine andere Wahl als im linken Teilbaum weiterzusuchen.
- **Fall 3**: $x >$ Schlüssel (v). Diesmal muss im rechten Teilbaum weitergesucht werden.

Offensichtlich ist die worst-case Laufzeit durch $O(\text{Tiefe}(T))$ beschränkt, wenn T der gegenwärtige binäre Suchbaum ist. Wenden wir uns jetzt der Operation *insert*(x, y) zu, wobei x der Schlüssel und y der Info-Teil sei. Zuerst verhalten wir uns so, als ob wir den Schlüssel x suchen. Sollten wir ihn tatsächlich finden, überschreiben wir den alten Info-Teil mit y. Ansonsten ist der Schlüssel dort einzufügen, wo die Suche scheitert.

Auch diesmal ist die worst-case Laufzeit durch $O(\text{Tiefe}(T))$ beschränkt. Da wir einfügen können, müßten wir auch sortieren können: Dies ist tatsächlich der Fall, denn die Inorder-Reihenfolge wird die Schlüssel aufsteigend ausgeben!

Damit drängt sich die folgende C++ Klasse auf:

```
//Deklarationsdatei bsbaum.h fuer binaere Suchbaeume.

typedef struct Knoten
  { schluesseltyp schluessel; infotyp info;
    //schluesseltyp und infotyp sind vorspezifizierte Typen.
    Knoten *links, *rechts;
    Knoten (schluesseltyp s, infotyp i, Knoten *l, Knoten *r)
    {schluessel = s; info = i; links = l; rechts = r;}
    //Konstruktor.
  };
```

```
class bsbaum
{
private:
  Knoten *Kopf;

public:
  bsbaum ( ) {Kopf = new Knoten (0,0,0,0);}
  //Konstruktor.  Kopf->rechts wird stets auf die Wurzel zeigen.

  ~bsbaum ( );
  //Destruktor.

  Knoten *lookup (schluesseltyp x);

  void insert (schluesseltyp x, infotyp info);

  void remove (schluesseltyp x);

  void inorder ( );
};
```

Die Operationen `lookup` und `insert` lassen sich leicht wie folgt kodieren.

Algorithmus 5.1 **(Binäre Suchbäume: Lookup)**

```
Knoten *bsbaum::lookup (schluesseltyp x)
{
  Knoten *Zeiger = Kopf->rechts;
  while ((Zeiger != 0) && (x != Zeiger->schluessel))
    Zeiger = (x < Zeiger->schluessel) ? Zeiger->links : Zeiger->rechts;
  return Zeiger;
};
```

Algorithmus 5.2 **(Binäre Suchbäume: Insert)**

```
void bsbaum::insert (schluesseltyp x, infotyp info)
{
  Knoten *Eltern, *Zeiger;
  Eltern = Kopf; Zeiger = Kopf->rechts;
  while ((Zeiger != 0) && (x != Zeiger->schluessel))
  { Eltern = Zeiger;
    Zeiger = (x < Zeiger->schluessel) ? Zeiger->links : Zeiger->rechts;
  }
  if (Zeiger == 0)
  {
    Zeiger = new Knoten (x, info, 0, 0);
    if (x < Eltern->schluessel) Eltern->links = Zeiger;
    else Eltern->rechts = Zeiger;
  }
  else Zeiger->info = info;
}
```

Beschäftigen wir uns als Nächstes mit der Operation `remove` (x). Natürlich müssen wir den Schlüssel x zuerst suchen. Angenommen, die Suche endet erfolgreich im Knoten v.

Fall 1: v ist ein Blatt. Mit dem Abhängen von v sind wir fertig.
Fall 2: v hat genau ein Kind w. Jetzt können wir v entfernen, indem wir den Zeiger auf v umsetzen, so dass er jetzt auf w zeigt.
Fall 3: v hat 2 Kinder.
Diesmal wird es ein wenig schwieriger. Wir ersetzen v durch den kleinsten Schlüssel im rechten Teilbaum. Der Knoten u speichere diesen kleinsten Schlüssel. Dann ist

- u der linkeste Knoten im rechten Teilbaum (und damit leicht zu finden) und
- u hat kein linkes Kind (und damit kann u leicht entfernt werden).

Auch diesmal ist die worst-case Laufzeit offensichtlich durch $O(\text{Tiefe}(T))$ beschränkt. Leider können aber binäre Suchbäume großer Tiefe leicht erzeugt werden. Die Folge `insert`(1, info), `insert`(2, info), ... `insert`(n, info) erzeugt einen Baum der (maximalen) Tiefe $n-1$. Dies ist sehr enttäuschend, da wir lineare Laufzeit auch zum Beispiel mit (sogar unsortierten) Arrays oder Listen erreichen können. Um die worst-case Laufzeit zu senken, werden wir neue Baumstrukturen entwerfen müssen.

Aber, und das sind die guten Nachrichten, die erwartete Laufzeit ist logarithmisch für die Operationen `insert` und `lookup`. Wir beschränken uns hier auf die Bestimmung der *erwarteten Laufzeit* $L(n)$ einer erfolgreichen `lookup`-Operation. Dazu nehmen wir an,

- dass die Schlüssel $1, 2, \ldots, n$ gemäß einer zufälligen Eingabepermutation π eingefügt werden, wobei jede Permutation die Wahrscheinlichkeit $1/n!$ besitzt
- und dass die Zeit von `lookup`(x) für ein zufälliges Element $x \in \{1, \ldots, n\}$ zu ermitteln ist.

Wir möchten also die erwartete Lookup-Zeit, gemittelt über alle Argumente x und alle Eingabepermutationen, bestimmen.

Sei T_π der resultierende binäre Suchbaum, wenn wir die Schlüssel gemäß π einfügen und sei $\text{Tiefe}_\pi(v)$ die Tiefe des Knotens v, wenn wir die Schlüssel in der Reihenfolge π einfügen. Die erwartete Lookup-Zeit können wir dann mit

$$L(n) = \frac{1}{n!} \cdot \sum_{\pi} \frac{1}{n} \cdot \sum_{v \in T_\pi} (\text{Tiefe}_\pi(v) + 1)$$

angeben: $\frac{1}{n} \cdot \sum_{v \in T_\pi} (\text{Tiefe}_\pi(v) + 1)$ ist die erwartete Lookup-Zeit bei gegebener Permutation π; die äußere Summe mittelt die erwartete Lookup-Zeit über alle möglichen Eingabepermutationen.

Wir konzentrieren uns auf die Bestimmung der inneren Summe, verändern die Perspektive und verkomplizieren unseren Ausdruck kurzzeitig durch das Einführen von $a_\pi(w, v) \in \{0, 1\},\ w, v \in T$, wobei

$$a_\pi(w, v) := \begin{cases} 1 & \text{falls } w \text{ Vorfahre von } v \text{ in } T_\pi \text{ ist,} \\ 0 & \text{sonst} \end{cases}$$

gilt. Wir erhalten

$$\sum_{v \in T_\pi} (\text{Tiefe}_\pi(v) + 1) = n + \sum_{v \in T_\pi} \text{Tiefe}_\pi(v) = n + \sum_{v \in T_\pi} \sum_{w \in T_\pi} a_\pi(w, v)$$

und deshalb ist

$$\begin{aligned} L(n) &= \frac{1}{n!} \cdot \sum_{\pi} \frac{1}{n} \cdot \sum_{v \in T_\pi} (\text{Tiefe}_\pi(v) + 1) \\ &= \frac{1}{n!} \cdot \sum_{\pi} \left(1 + \frac{1}{n} \cdot \sum_{v \in T_\pi} \sum_{w \in T_\pi} a_\pi(w, v) \right) \\ &= 1 + \frac{1}{n!} \cdot \sum_{\pi} \frac{1}{n} \cdot \sum_{v \in T_\pi} \sum_{w \in T_\pi} a_\pi(w, v). \end{aligned} \tag{5.1}$$

Wir müssen also (5.1) analysieren und führen deshalb die Zufallsvariablen $a(i, j)$ mit $1 \leq i < j \leq n$ ein. Wir definieren

$$a(i,j) = \begin{cases} 1 & \text{entweder ist der Knoten mit Schlüssel } i \text{ Vorfahre des Knotens mit Schlüssel } j \text{ oder der Knoten mit Schlüssel } i \text{ ist Nachfahre des Knotens mit Schlüssel } j, \\ 0 & \text{sonst.} \end{cases}$$

Jetzt zahlt sich die Einführung der Parameter $a_\pi(i, j)$ aus, denn wir können $L(n)$ mit Hilfe des Erwartungswerts E umschreiben:

$$L(n) = 1 + E\,[\frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=i+1}^{n} a(i,j)].$$

Bisher haben wir nur formale Manipulationen durchgeführt, aber keine wirkliche Arbeit geleistet. Auch der nächste Schritt, nämlich das Ausnutzen der Additivität des Erwartungswert, ist vorgezeichnet und es ist

$$L(n) = 1 + \frac{1}{n} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \cdot E[a(i,j)]. \tag{5.2}$$

Wir kommen aber jetzt zum zentralen Schritt:

Lemma 5.1 $E\,[a(i,j)] = \frac{2}{j-i+1}$.

Beweis: Sei $p_{i,j}$ die Wahrscheinlichkeit, dass das Ereignis $a(i, j) = 1$ eintritt. Dann ist

$$\begin{aligned} E[a(i,j)] &= 1 \cdot p_{i,j} + 0 \cdot (1 - p_{i,j}) \\ &= p_{i,j}. \end{aligned}$$

Wir haben also die $p_{i,j}$ zu bestimmen. Verfolgen wir also, wie der binäre Suchbaum T_π aufgebaut wird. Angenommen π erzwingt, dass zuerst Schlüssel k eingefügt wird. Wenn $i < k < j$, dann landen i und j in verschiedenen Teilbäumen der Wurzel und es ist $a(i, j) = 0$. Ist hingegen $k = i$ oder $k = j$, dann wird $a(i, j) = 1$ erzwungen. Schließlich, wenn $k < i$ oder $k > j$, dann bleibt der Wert von $a(i, j)$ offen.

Diese Situation bleibt unverändert, wenn wir den Baum T_π weiter wachsen lassen: Angenommen wir haben gerade einen Schlüssel l eingefügt und sowohl die Binärsuche nach i wie auch die Binärsuche nach j enden in dem Knoten, der l speichert. Wiederum lässt l mit $l < i$ oder $l > j$ den Wert von $a(i, j)$ offen, während l mit $i < l < j$ (bzw. $l = i$ oder $l = j$) den Wert $a(i, j) = 0$ (bzw. $a(i, j) = 1$) erzwingt.

Die Frage, welches Element aus dem Intervall $[i, j]$ als Erstes eingefügt wird, ist also entscheidend. Da jedes Element gleichwahrscheinlich ist, genügt es festzuhalten, dass das Intervall

$j - i + 1$ Elemente enthält und es nur bei der Wahl von i oder j zum Vergleich zwischen i und j kommt. Also ist

$$p_{i,j} = \frac{2}{j - i + 1}$$

und das war zu zeigen. □

Für benachbarte Elemente $j = i+1$ ist $p_{i,j}$ also beispielsweise 1. Richtig! Benachbarte Elemente müssen verglichen werden, da sie sich allen dritten Elementen gegenüber gleich verhalten und nur durch den direkten Vergleich getrennt werden können.

Wir können jetzt die Darstellung (5.2) von L_n auswerten:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n} E[a(i,j)] &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} p_{i,j} \\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2 \cdot \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= 2 \cdot \sum_{i=1}^{n}\sum_{j=2}^{n-i+1} \frac{1}{j} \\
&= 2 \cdot \sum_{i=1}^{n} O(\log(n-i+1)) && \text{siehe Lemma 3.4} \\
&= O\left(\sum_{i=1}^{n} \log(n)\right) \\
&= O(n \cdot \log(n))
\end{aligned}$$

und wir erhalten $L_n = O(\log_2 n)$ mit (5.2).

Satz 5.2 *Sei T ein binärer Suchbaum mit n Knoten.*

(a) Die worst-case Laufzeit von `insert`, `remove` *oder* `lookup` *ist* $\Theta(\text{Tiefe}(T))$.

(b) Es ist bestenfalls $\text{Tiefe}(T) = \lfloor \log_2 n \rfloor$ *und schlechtestenfalls* $\text{Tiefe}(T) = n - 1$.

(c) Die erwartete Laufzeit einer erfolgreichen lookup-Operation in binären Suchbäumen mit n Schlüsseln ist $O(\log_2 n)$.

Obwohl die erwartete Laufzeit zufriedenstellend ist, ist das Verhalten von binären Suchbäumen auf fast sortierten Daten katastrophal. Neue Datenstrukturen müssen her.

Aufgabe 71
Wir betrachten binäre Suchbäume T, die Mengen $M = M(T) \subseteq \mathbb{Z}$ darstellen. **Beschreibe** Algorithmen für die folgenden Probleme:

(a) Eingabe: Zwei binäre Suchbäume T_1 und T_2 sowie die Tiefen $Tiefe(T_1)$ und $Tiefe(T_2)$ der beiden Bäume. Es gelte $x < y$ für alle $x \in M(T_1)$ und alle $y \in M(T_2)$.
Ausgabe: Ein binärer Suchbaum T für die Menge $M(T_1) \cup M(T_2)$. Die Tiefe von T soll die größere der Tiefen von T_1 und T_2 möglichst wenig überschreiten.

(b) Eingabe: Ein binärer Suchbaum T und ein Element $x \in \mathbb{Z}$.
Ausgabe: Ein binärer Suchbaum für die Menge $split(T, x) := \{y \in M(T) : y \leq x\}$.

Beide Aufgaben sollen natürlich möglichst effizient gelöst werden. **Gib** die Laufzeiten der Algorithmen jeweils als Funktion in der Tiefe der Bäume an.

Aufgabe 72
Entwerfe einen Algorithmus für die Operation `split(T,x)`. T ist ein binärer Suchbaum (gegeben durch einen Zeiger auf die Wurzel) und x eine natürliche Zahl. Als Ergebnis liefert `split` zwei binäre Suchbäume T_1 und T_2 (also zwei Zeiger auf die Wurzeln), wobei T_1 alle Schlüssel aus T enthält, die kleiner x sind, und T_2 die restlichen Schlüssel. Der Algorithmus soll in worst-case Laufzeit $O(|\text{Tiefe}(T)|)$ laufen. Es darf angenommen werden, dass der Baum nur verschiedene Schlüssel speichert.

Aufgabe 73
Beweise oder **widerlege**, dass sich ein binärer Suchbaum eindeutig aus der Postorder Reihenfolge seiner Schlüssel rekonstruieren lässt.

5.2 AVL-Bäume

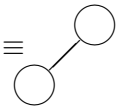
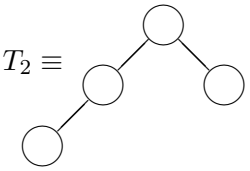
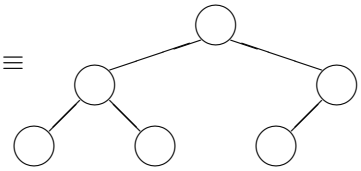
AVL-Bäume sind nach ihren Erfindern Adelson- Velskii und Landis benannt.

Definition 5.2 Ein binärer Suchbaum heißt **AVL-Baum**, wenn für jeden Knoten v mit linkem Teilbaum $T_L(v)$ und rechtem Teilbaum $T_R(v)$

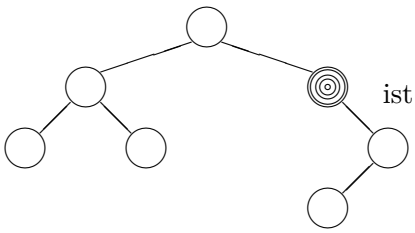
$$|\,\text{Tiefe}(T_L(v)) - \text{Tiefe}(T_R(v))\,| \leq 1$$

gilt. $b(v) := \text{Tiefe}(T_L(v)) - \text{Tiefe}(T_R(v))$ heißt auch der **Balance-Grad** von v. (Beachte, dass für AVL-Bäumen stets $b(v) \in \{-1, 0, 1\}$ gilt.)

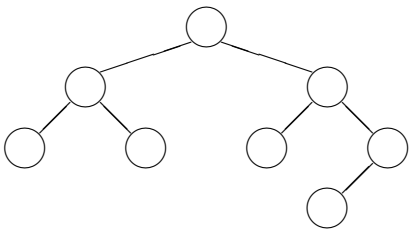
Beispiel 5.3

$T_0 \equiv$  $T_1 \equiv$  $T_2 \equiv$  $T_3 \equiv$ 

Sämtliche Bäume sind AVL-Bäume.



ist kein AVL-Baum, während

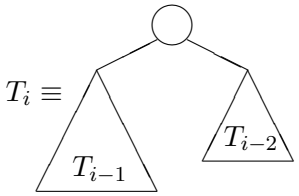


ein AVL-Baum ist.

Wie tief kann ein AVL-Baum mit n Knoten werden? Anders gefragt, wieviele Knoten muss ein AVL-Baum der Tiefe t mindestens besitzen? Sei $\min(t)$ diese minimale Anzahl. Dann gilt offensichtlich die Rekursion

$$\min(0) = 1, \quad \min(1) = 2 \text{ und } \min(t) = \min(t-1) + \min(t-2) + 1$$

und die rekursiv definierten Bäume



minimieren die Knotenanzahl. Wir behaupten, dass

$$\min(t) \geq 2^{t/2}$$

Dies ist offensichtlich richtig für $t = 0$ und $t = 1$ und mit der Induktionsannahme folgt

$$\begin{aligned} \min(t+1) &= \min(t) + \min(t-1) + 1 \\ &\geq 2^{t/2} + 2^{(t-1)/2} + 1 \\ &\geq 2 \cdot 2^{(t-1)/2} = 2^{(t+1)/2}. \end{aligned}$$

(Eine exaktere Auswertung von $\min(t)$ wird durchgeführt, indem die Ähnlichkeit zur Fibonacci-Folge $\mathrm{fib}(n)$, mit $\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$ ausgenutzt wird.) Unser Ergebnis ist somit:

Satz 5.3 *Ein AVL-Baum mit n Knoten hat Tiefe höchstens $2\log_2 n$.*

Beweis: Wir haben gerade gezeigt, dass ein AVL-Baum der Tiefe $t = 2\log_2 n + 1$ mindestens

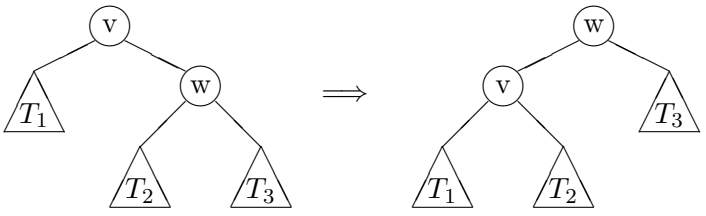
$$2^{t/2} = 2^{\log_2 n + \frac{1}{2}} = \sqrt{2} \cdot n > n$$

Knoten besitzt. □

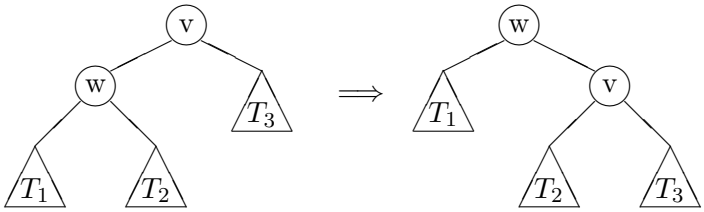
AVL-Bäume werden `lookup` somit problemlos in logarithmischer Zeit durchführen. Es wird aber kritisch, wenn die Operation `insert` und `remove` ausgeführt werden: Ein bloßes Anhängen/Abhängen von Knoten kann die AVL-Eigenschaft zerstören.

Zuerst drücken wir uns um die `remove`-Operation herum: wir führen nur eine `lazy remove` Operation durch. `lazy remove` markiert einen gelöschten Knoten als entfernt ohne ihn tatsächlich zu entfernen. Wenn allerdings mehr als 50 % aller Knoten als entfernt markiert sind, beginnt ein Großreinemachen: Der gegenwärtige Baum wird durchlaufen und ein neuer AVL-Baum wird aus den nicht markierten Knoten des alten Baumes aufgebaut. Die Laufzeit für den Neuaufbaus ist groß, aber gegen die vielen remove-Operationen *amortisiert*. Mit anderen Worten, die Laufzeit für den Neuaufbau ist asymptotisch beschränkt durch die Gesamtlaufzeit aller lazy remove-Operationen, die den Neuaufbau auslösten.

Beachte, dass `lazy remove` harmlos ist, da im wesentlichen nur ein lookup durchzuführen ist. Kommen wir also zur Operation `insert`. Das Konzept einer Linksrotation/Rechtsrotation ist wesentlich:



Eine **Linksrotation in** v wird durch das obige Diagramm dargestellt. Beachte, dass eine Linksrotation die Eigenschaft eines binären Suchbaums bewahrt. Analog arbeitet eine **Rechtsrotation in** v**:**

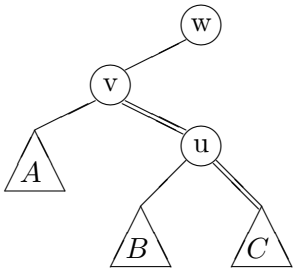


Ein `insert` in einem AVL-Baum wird anfänglich genau wie ein `insert` in einem binären Suchbaum behandelt: Der Schlüssel wird gesucht und am Ende einer erfolglosen Suche eingefügt. Wie schon erwähnt, haben wir durch das Einfügen möglicherweise die AVL-Eigenschaft verletzt. Wo? Nur Knoten auf dem Pfad von der Wurzel zum frisch eingefügten Blatt können betroffen sein!

Wir werden deshalb nach dem Einfügen den Weg (möglicherweise ganz) zurücklaufen um nachzuprüfen, ob die Balance-Eigenschaft verletzt ist. Dabei wird uns der Balance-Grad der Knoten helfen, den wir in Definition 5.2 eingeführt haben. Wir nehmen an, dass die AVL-Datenstruktur den Balance-Grad $b(v)$ für jeden Knoten v in der Struktur von v speichert. Insbesondere müssen wir also auch, wenn notwendig, die Balance-Grade neu berechnen.

Angenommen, wir sind bis zum Knoten u zurückgelaufen, haben dort den Balance-Grad neu berechnet und einen möglichen Verstoß gegen die AVL-Eigenschaft behoben. Wenn wir die Reparatur fortsetzen müssen, müssen wir uns als Nächstes um den Elternknoten v von u kümmern. w bezeichne den Großelternknoten von u.

Fall 1: Der Zick-Zick Fall.



Ein neues Blatt wurde im Teilbaum mit Wurzel u eingefügt und Tiefe$(C) \geq$ Tiefe(B).

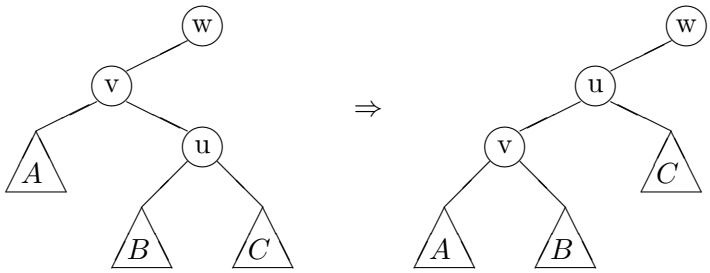
Wir erreichen diesen Fall nur dann, wenn die Tiefe des Teilbaums von u um 1 angewachsen ist: ansonsten hätte die Reparaturphase (also das Zurücklaufen des Weges) in u schon beendet werden können. Sei d die neue (um 1 größere) Tiefe des Teilbaums von u.

- Der Fall Tiefe$(A) \geq d + 1$ kann nicht auftreten, da sonst $b(v) \geq 2$ vor Einfügen des neuen Blatt gilt.

- Im Fall Tiefe$(A) = d$ brauchen wir nur den (neuen) Balance-Grad zu setzen, nämlich $b(v) = 0$. Die Reparatur kann in diesem Fall abgebrochen werden, da der Teilbaum mit Wurzel v seine Tiefe nicht verändert hat.

- Wenn Tiefe$(A) = d - 1$, setzen wir $b(v) = -1$. Diesmal müssen wir allerdings die Reparatur in w fortsetzen: Die Tiefe des Teilbaums mit Wurzel v ist um 1 angestiegen.

- Wenn Tiefe$(A) = d-2$, dann ist die AVL-Eigenschaft verletzt, da der neue Balance-Grad -2 beträgt. Die Reparatur gelingt mit einer Linksrotation:

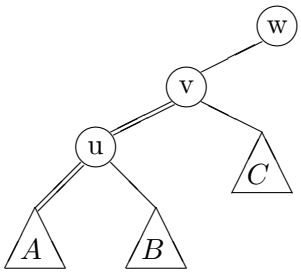


Es ist Tiefe$(C) - 1 \leq$ Tiefe$(B) \leq$ Tiefe(C). Warum? Es ist Tiefe$(B) \leq$ Tiefe (C) nach Fallannahme. Weiterhin gilt die AVL-Eigenschaft in u und deshalb ist Tiefe$(B) \geq$ Tiefe$(C) - 1$.

Die AVL-Eigenschaft gilt somit nach der Rotation für u und v. Schließlich setze $b(u)$ und $b(v)$ entsprechend und fahre fort, wenn der (neue) Teilbaum von u tiefer ist als der alte Teilbaum von v (vor dem Einfügen des Blatts). (Tatsächlich kann die Reparaturphase erfolgreich abgebrochen werden. Warum?)

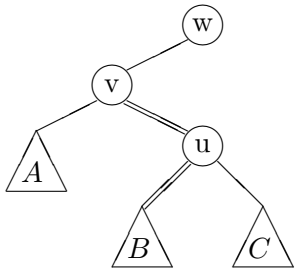
- Der Fall Tiefe$(A) \leq d-3$ kann nicht auftreten, da sonst $b(v) \leq -2$ vor Einfügen des neuen Blatts.

Fall 2: Der Zack-Zack Fall.



Ein neues Blatt wurde im Teilbaum mit Wurzel u eingefügt und Tiefe$(A) \geq$ Tiefe(B). Der Zack-Zack Fall wird wie der Zick-Zick Fall behandelt.

Fall 3: Der Zick-Zack Fall.

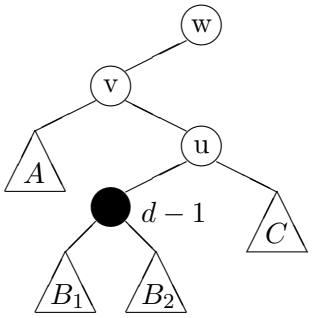


Ein neues Blatt wurde im Teilbaum mit Wurzel u eingefügt und Tiefe$(B) >$ Tiefe(C).

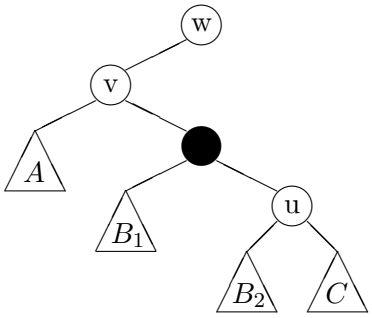
Auch diesen Fall erreichen wir nur, wenn die Tiefe des Teilbaums von u um 1 angestiegen ist. Sei d die neue Tiefe des Teilbaums von u. Bis auf den Fall Tiefe$(A) = d-2$ können alle weiteren Fälle wie in Fall 1 behandelt werden. Falls Tiefe$(A) = d-2$ gilt, erhalten wir

$$\text{Tiefe}(A) = \text{Tiefe}(C) = d-2 \quad \text{und} \quad \text{Tiefe}(B) = d-1.$$

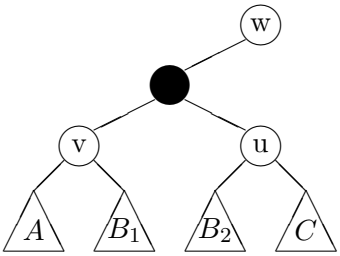
Beachte, dass diesmal eine Linksrotation in v keine Reparatur darstellt: Der rechte Teilbaum (nach der Rotation) ist nicht tief genug. Wir betrachten zuerst den Teilbaum B genauer:



Wir führen eine Rechtsrotation in u durch



sodann eine Linksrotation in v



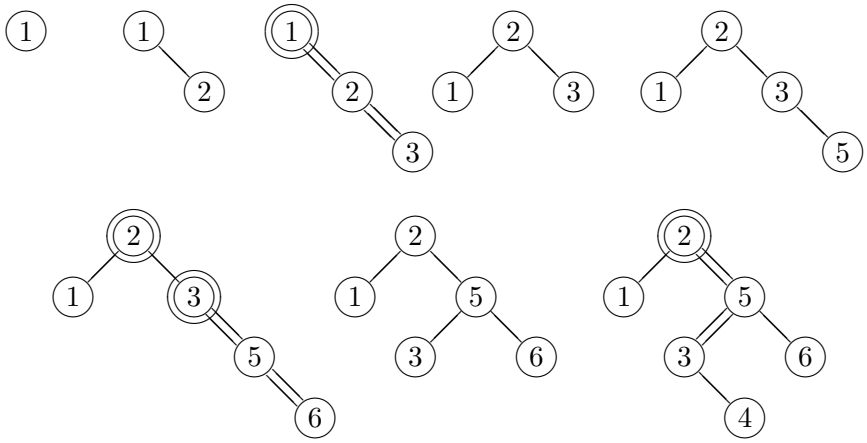
und die AVL-Eigenschaft ist überall repariert. Beachte, dass die Tiefe nach der Doppelrotation um 1 gesunken ist und damit mit der Tiefe vor dem Einfügen des neuen Blatts übereinstimmt. Nach Setzen der neuen Balance-Grade kann die Reparatur damit abgebrochen werden.

Fall 4 Zack-Zick: Behandle analog zu Fall 3.

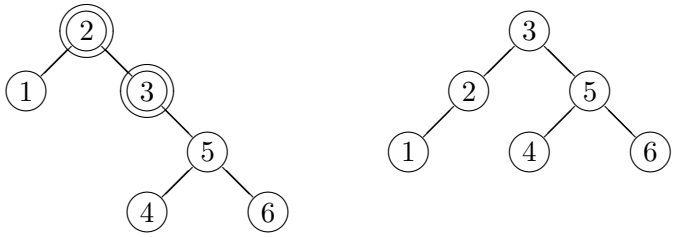
Satz 5.4 *Die Operationen* `lookup` *und* `insert` *haben die worst-case Laufzeit* $O(\log_2 n)$ *für AVL-Bäume mit* n *Knoten.*

Bemerkung 5.1 Wie im Falle binärer Suchbäume liegt es nahe, eine C++ Klasse für AVL-Bäume zu bilden. Diese Klasse wird der Klasse bsbaum der binären Suchbäumen im Aufbau ähneln. Allerdings ist jedem Knoten ein Feld für den Balance-Grad sowie ein Feld zum Eintragen seines Status (entfernt oder nicht entfernt) beizufügen. Weiterhin sollten die Operationen Linksrotation und Rechtsrotation als zusätzliche Operationen zur Verfügung gestellt werden.

Beispiel 5.4 Wir führen nacheinander die Operationen insert (1), insert (2), insert (3), insert (5), insert(6) und insert (4) durch. Wenn ein Balance-Verstoß vorliegt, wird der betroffene Knoten doppelt umrandet.



Wir haben den Zick-Zack Fall erreicht. Eine Rechtsrotation in 5 wird durchgeführt, gefolgt von einer Linksrotation in 2:

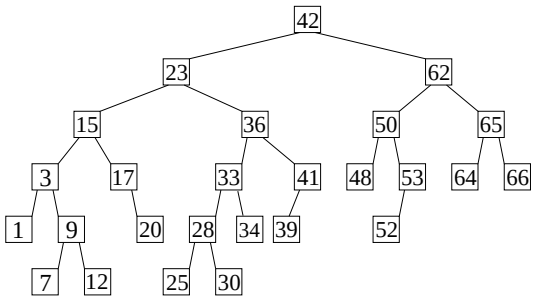


Aufgabe 74

Füge in einen leeren AVL Baum nacheinander die Elemente $9, 16, 5, 13, 18, 12, 6$ und 19 ein. **Zeichne** den Baum nach jeder Einfügung und nach jeder Rotation.

Aufgabe 75

(a) Gegeben sei der folgende AVL-Baum.



Füge in Abhängigkeit des Anfangsbuchstaben deines Vornamens einen Schlüssel in diesen Baum **ein**.

Anfangsbuchstabe des Vornamens	A,Ä	E,Q,Ö	I,J,W	O,F	U,Ü,P	S,X,Y
einzufügender Schlüssel	5	8	11	13	18	22

Anfangsbuchstabe des Vornamens	N,B	R,C	T,G	L,K,V	H,D	M,Z
einzufügender Schlüssel	24	27	29	32	38	40

Füge danach bei 54 beginnend, aufsteigend soviele weitere Schlüssel in den Baum **ein**, bis es zu einer Rotation an dem Knoten mit Wert 62 kommt.

Die Darstellung soll dabei möglichst ausführlich sein.

(b) **Gib** eine Familie von $AVL - Bäumen$ **an**, bei der die maximale Differenz der Tiefen zweier Blätter $\Omega(Tiefe(T))$ ist.

Aufgabe 76
Wir betrachten neben lookup(x), insert(x) und remove(x) auch die Operationen select(k) und rang(x). Die Operation select(k) bestimmt den k-kleinsten Schlüssel und rang(x) $= k$ gilt genau dann, wenn x der k-kleinste Schlüssel ist. **Beschreibe** eine Modifikation der AVL-Bäume, die bei n gegenwärtig gespeicherten Schlüsseln *alle* fünf Operationen in worst-case Laufzeit $O(\log n)$ unterstützt.

Aufgabe 77
Wir beschreiben den abstrakten Datentyp *Intervall*. Die Operationen des abstrakten Datentyps sind:

- Einfügen eines Intervalls (falls dieses noch nicht vorhanden ist),
- Punktabfrage für $x \in \mathbb{R}$: ein Intervall, das x enthält, ist auszugeben (wenn ein solches existiert).

Beschreibe eine Datenstruktur, die bei n gegenwärtig gespeicherten (verschiedenen) Intervallen das Einfügen eines Intervalls in worst-case Laufzeit $O(\log n)$ unterstützt. Ebenso muss die Punktabfrage in worst-case Zeit $O(\log n)$ implementiert werden.
Hinweis: Benutze (z.B.) AVL-Bäume und ordne die Intervalle gemäß ihrem linken Endpunkt an. Es muss jedoch Zusatzinformation zur Verfügung gestellt werden. Welche?

Aufgabe 78
Warum kann die Reparatur der Tiefenbalancierung für AVL-Bäume nach einer `insert`(x) Operation abgebrochen werden, wann immer eine einfache Rotation für Zick-Zick (oder Zack-Zack) durchgeführt wird?

5.3 Splay-Bäume

Splay-Bäume werden sich auch als eine effiziente Datenstruktur für das Wörterbuchproblem herausstellen. Zuerst besprechen wir ihre Vorteile gegenüber AVL-Bäumen.

- Keine zusätzliche Information (wie Balance-Grad und die „entfernt“-Option) ist notwendig. Splay-Bäume sind damit speicher-effizienter als AVL-Bäume.
- Der Programmieraufwand ist ungleich geringer. So können wir zum Beispiel eine vollwertige `remove`-Operation mit wenigen zusätzlichen Operationen ausführen.
- Splay-Bäume „passen sich den Daten an“: Schlüssel, die relativ oft abgefragt werden, werden an die Spitze des Baumes bewegt. (Splay-Bäume werden deshalb den *selbstorganisierenden* Datenstrukturen zugerechnet.)

Jetzt zu den Nachteilen: Splay-Bäume besitzen eine miserable worst-case Laufzeit *pro* Operation, nämlich $\Theta(n)$, wenn n Schlüssel zu speichern sind. Die Ausführung einer *einzelnen* Operation kann somit sehr lange dauern. **Aber**, wenn wir n Operationen in einem anfänglich leeren Splay-Baum durchführen, dann werden diese Operationen in der worst-case Laufzeit $O(n \log_2 n)$ berechnet, obwohl es durchaus einzelne Operationen mit worst-case Laufzeit $\Theta(n)$ geben kann. Man sagt deshalb auch, dass Splay-Bäume die *amortisierte Laufzeit* $O(\log_2 n)$ besitzen: eine teure Operation wird amortisiert gegen die vorher ausgeführten billigen Operationen.

Kommen wir zurück zum Aspekt der Selbstorganisation. Angenommen, ein Splay-Baum ist aufgebaut und wird als ein statisches Wörterbuch benutzt. Im allgemeinen wird man nicht erwarten können, dass alle Schlüssel mit gleicher Wahrscheinlichkeit abgefragt bzw. akualisiert werden, vielmehr werden einige Schlüssel häufiger als andere abgefragt. (Ein Beispiel ist die Kundendatei einer Bank: Die Anzahl der finanziellen Transaktionen wird sich von Kunde zu Kunde stark unterscheiden.) Splay-Bäume werden sich dann fast optimal auf die unbekannte Wahrscheinlichkeit einstellen: Sind zum Beispiel m von n Schlüssel hochwahrscheinlich, wird

die amortisierte Laufzeit einer lookup- oder Update-Operation für jeden dieser m Schlüssel durch $O(\log_2(m))$ beschränkt sein.

Die Struktur eines Splay-Baumes ist die eines binären Suchbaumes. Splay-Bäume arbeiten „ohne Netz und doppelten Boden“, also ist keine zusätzliche Information pro Knoten erforderlich. Wie sollen sich aber dann Splay-Bäume den Daten anpassen?

Die Grundidee ist einfach: Wenn nach einem Schlüssel x gefragt wird, dann wird der Schlüssel mit Rotationen zur Wurzel gebracht. Nachfolgende Operationen für andere Schlüssel werden Schlüssel x langsam tiefer und tiefer nach unten drücken. Wenn aber nach einiger Zeit Schlüssel x wieder abgefragt wird, wird er sofort wieder zur Wurzel gebracht. Wenn also zwischen sukzessiven Anfragen nach x nicht zuviel Zeit vergeht, wird jede Abfrage schnell sein: da x nicht tief im Baum sitzt, ist die Binärsuche schnell erfolgreich.

Zusammengefaßt, Splay-Bäume sind dann hervorragend geeignet, wenn bestimmte Schlüssel sehr viel wahrscheinlicher als die restlichen Schlüssel sind.

Splay-Bäume arbeiten nur mit einer Operation, der `splay`-Operation. Für Schlüssel x verläuft `splay`(x) wie folgt:

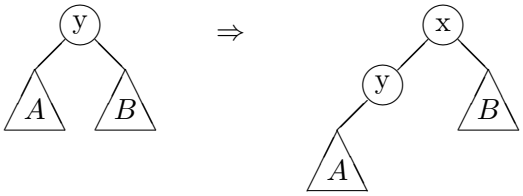
(1) Zuerst wird eine Binärsuche nach x gestartet, die mit dem Schlüssel y terminieren möge. Bei einer erfolgreichen Suche ist $x = y$. Ansonsten ist y der kleinste Schlüssel, der größer als x ist, oder der größte Schlüssel, der kleiner als x ist.

(2) Der Schlüssel y wird zur Wurzel gemacht. Wir werden diesen Schritt später im Detail beschreiben.

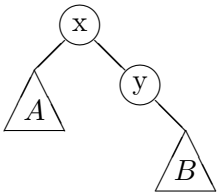
Alle Operationen werden auf die `splay`-Operation zurückgeführt, weshalb der Programmieraufwand dementsprechend gering ist.

- `lookup`(x): Führe `splay`(x) durch und überprüfe die Wurzel.

- `insert`(x, info) : Führe `splay`(x) durch. Wenn die Wurzel den Schlüssel x speichert, überschreibe den Info-Teil. Ansonsten speichert die Wurzel den Schlüssel y. Wenn $y < x$, dann füge x wie folgt ein:



Der Fall $y > x$ ist analog zu behandeln:

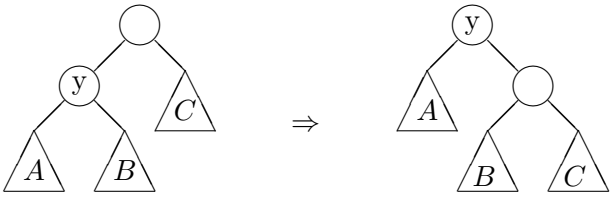


- `remove`(x): Führe `splay`(x) durch. Wenn die Wurzel den Schlüssel x nicht speichert, gib eine Fehlermeldung aus. Ansonsten:

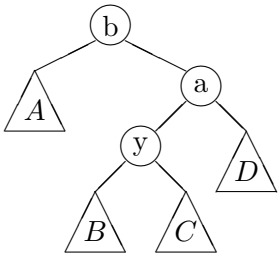
Wende `splay`(x) auf den Baum A an:

Für die Implementierung von `splay` müssen wir besprechen, wie der gefundene Schlüssel y zur Wurzel gebracht wird.

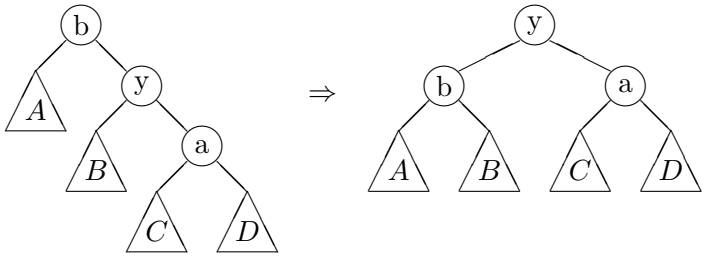
Fall 1: y ist ein Kind der Wurzel.
Wende die entsprechende Rotation an. Zum Beispiel:



Fall 2: y hat Tiefe mindestens 2 und der Zick-Zack Fall liegt vor:



Wir verhalten uns genauso wie in AVL-Bäumen: Einer Rechtsrotation in a folgt eine Linksrotation in b. Also:



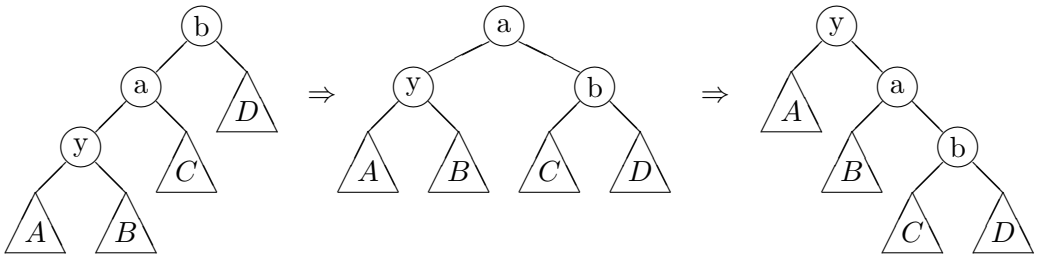
Was passiert?

- y ist näher zur Wurzel gerückt.
- Alle Knoten in B und C sind ebenso der Wurzel nähergerückt.
- Die Knoten in D behalten ihren Abstand, die Knoten in A rücken von der Wurzel weg.

y rückt der Wurzel um zwei Kanten näher, während alle Knoten in B oder C der Wurzel um eine Kante näherrücken. Mit anderen Worten, nach wiederholte Anwendungen von Fall 2 *rücken die Knoten des Suchpfades der Wurzel fast um die Hälfte näher.* Wir nennen dies die *Suchpfad-Eigenschaft.*

Fall 3: behandelt den zu Fall 2 symmetrischen Zack-Zick Fall, der wie Fall 2 behandelt wird.

Fall 4: y hat Tiefe mindestens 2 und der Zack-Zack Fall liegt vor: Wir führen eine Rechtsrotation in b durch, gefolgt von einer Rechtsrotation in a:

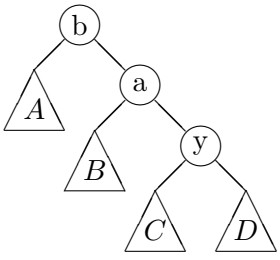


Was ist passiert?

- y ist näher zur Wurzel gerückt.
- Alle Knoten in A und B sind ebenso der Wurzel näher gekommen.
- Alle Knoten in C oder D rücken von der Wurzel weg.

Wiederholte Anwendungen von Fall 4 führen also wiederum ein Art von Balancierung durch: y rückt der Wurzel um zwei Kanten näher, während alle Knoten in A oder B der Wurzel um mindestens eine Kante näherrücken. *Die Knoten des Suchpfades rücken der Wurzel auch jetzt fast um die Hälfte näher.*

Fall 5: y hat Tiefe mindestens 2 und der Zick-Zick Fall liegt vor:

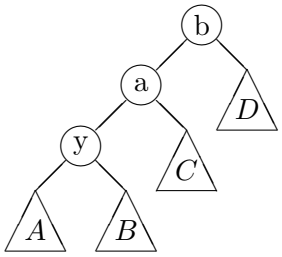


Verfahre wie in Fall 4.

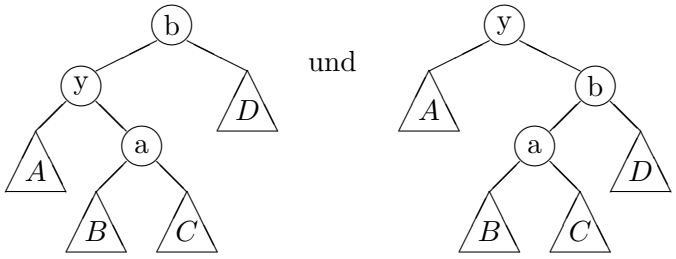
Bemerkung 5.2 **(a)** Insgesamt erhalten wir, dass in jedem betrachteten Fall die Knoten des Suchpfades der Wurzel fast um die Hälfte näherrücken.

(b) Fall 2 und Fall 3 führen Rotationen wie im Fall von AVL-Bäumen durch, aber in Fall 4 und Fall 5 gehen wir anders vor. Die `splay`-Operation beginnt mit einer Rotation am Großelternknoten, gefolgt von einer Rotation am Elternknoten, während AVL-Bäume zuerst eine Rotation am Elternknoten durchführen. Warum die unterschiedliche Behandlung?

Die Ausgangssituation für den Zack-Zack Fall war:



Angenommen, wir führen stets eine Rotation am Elternknoten durch:



Mit y ist auch Teilbaum A nach oben gerückt. Der Teilbaum B hingegen stagniert: Seine Distanz zur Wurzel bleibt unverändert. Diese Ausführung von Fall 4 würde also nicht die Suchpfad-Eigenschaft garantieren.

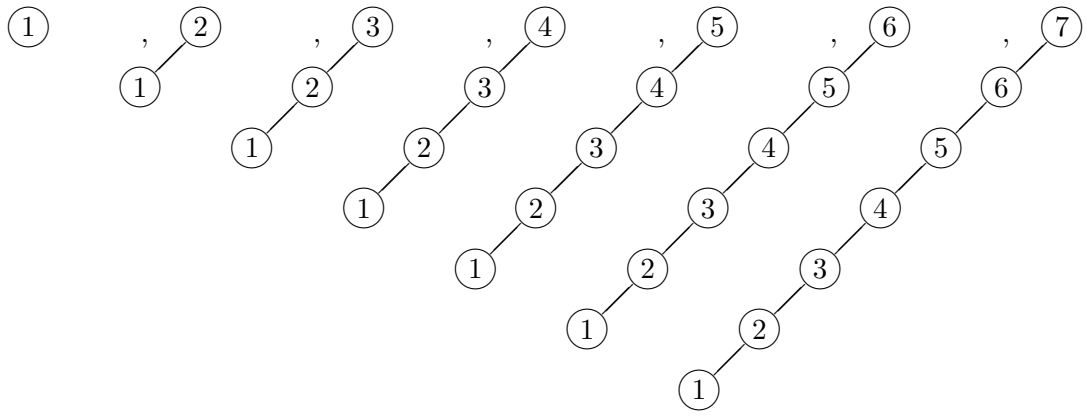
Die Verletzung der Suchpfad-Eigenschaft wäre verheerend wie wir auch im nächsten Beispiel sehen werden: Wenn der Suchpfad nicht nachhaltig verkürzt wird, dann werden wir ein schlechtes worst-case Verhalten selbst für eine *Folge* von lookup-Operationen erhalten. Diese Folge würde nämlich nach weiteren Knoten des immer noch langen Suchpfades suchen. Die `splay`-Operation vermeidet dies, in dem ein langer Suchpfad durch Rotationen „nach oben gefaltet wird“, und folgt damit der Devise:

> Wenn durch das Durchlaufen eines langen Pfades viel investiert wurde, dann muss die nachfolgende Reparatur durch die sorgfältige Wahl der Rotationen den Pfad substantiell kürzen.

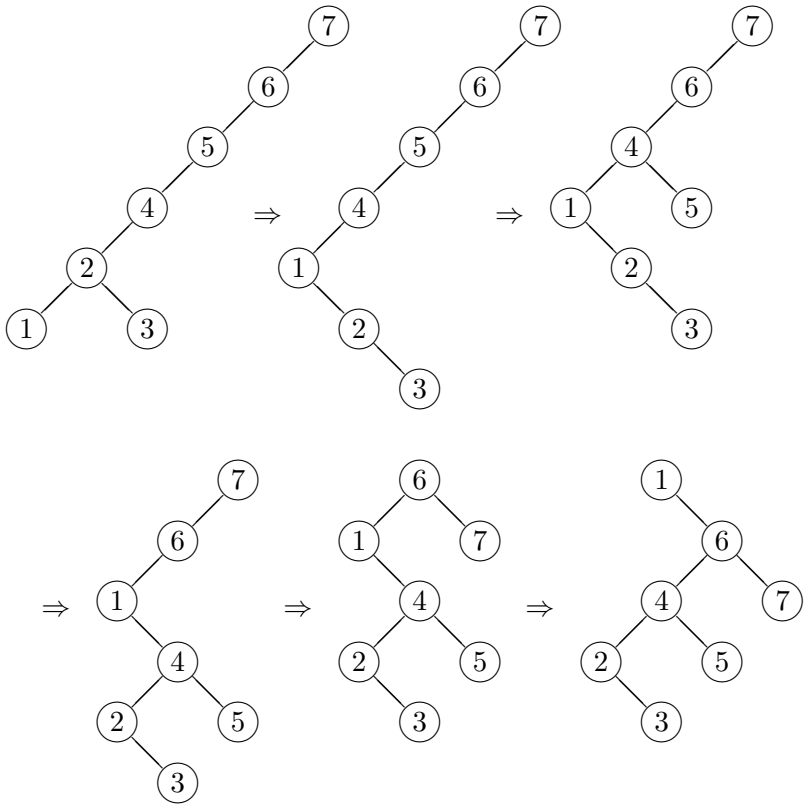
Satz 5.5 *Wenn n splay-Operationen (in einem anfänglich leeren Splay-Baum) durchgeführt werden, beträgt die worst-case Laufzeit O $(n \log_2 n)$.*

Für den Beweis verweisen wir auf Ottman, Widmayer: Algorithmen und Datenstrukturen, pp. 351 - 359.

Beispiel 5.5 Wir führen zuerst die Operationen insert (1), insert (2), insert(3), insert (4), insert (5), insert (6) und insert (7) aus.

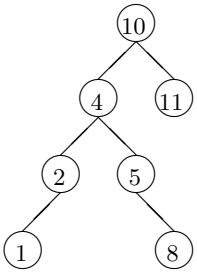


Alle Operationen sind sehr schnell. Dies ist auch nötig, denn die Operation lookup (1) muss bezahlt werden:



Die Tiefe ist von 6 auf 4 gesunken. (In der letzten splay-Operation haben wir wiederholt Fall 4 angewandt. Was passiert, wenn wir die in der obigen Bemerkung erwähnte Modifikation von Fall 4 anwenden?)

Aufgabe 79
Gegeben sei der folgende Splay-Baum:



Füge nacheinander die Schlüssel $3, 9, 1, 6$ (durch die `insert`-Operation für Splay-Bäume) ein. (Falls ein Schlüssel bereits vorhanden ist, wird dieser nur an die Wurzel gebracht.) **Zeichne** den Baum nach jeder Einfügung. Entferne dann nacheinander die Schlüssel $10, 5$ (durch die `remove`-Operation für Splay-Bäume). **Zeichne** auch hier den Baum nach jedem Entfernen.

Aufgabe 80
Beschreibe einen Algorithmus, der die Operation `interval(a,b)` in einem Splay-Baum B durchführt. Die Operation `interval(a,b)` liefert alle Schlüssel y von B, mit $a \leq y \leq b$, in aufsteigender Reihenfolge. Der Algorithmus soll in Laufzeit $O(\text{Tiefe}(B) + Y)$ laufen, wobei Y die Anzahl der von *interval(a,b)* ausgegebenen Schlüssel ist.

Aufgabe 81
Wenn wir die Knoten mit den Schlüsseln $1, 2, \ldots, 1000$ nacheinander in einen Splay-Baum einfügen, dann erhalten wir einen Splay-Baum von linken Kindern.

Berechne die Tiefe des obigen Splay-Baums nach *lookup*(1). **Begründe** deine Antwort.

Berechne die Tiefe des obigen Splay-Baums nach `lookup`(1), `lookup`(2),..., `lookup`(1000). **Begründe** deine Antwort.

Aufgabe 82
Sei T ein beliebiger Splay-Baum, der die Schlüssel $x_1 < x_2 < x_3 < \cdots < x_n$ enthält. **Wie** sieht der Baum T aus, nachdem die Operationen `splay`(x_1), `splay`$(x_2), \ldots,$ `splay`(x_n) durchgeführt sind? Begründe Deine Antwort.

5.4 (a, b)-Bäume

Definition 5.3 Seien a und b natürliche Zahlen mit $a \geq 2$ und $b \geq 2a - 1$. Ein Baum T hat die (a, b)**-Eigenschaft** genau dann, wenn

(a) alle Blätter von T die gleiche Tiefe haben,

(b) alle Knoten höchstens b Kinder besitzen und

(c) die Wurzel mindestens zwei Kinder hat, während alle inneren Knoten mindestens a Kinder haben.

(a, b)-Bäume besitzen die (a, b)-Eigenschaft und stellen weitere Information zur Verfügung, um die Schlüsselsuche zu erleichtern:

Definition 5.4 Der Baum T ist ein (a, b)**-Baum** für die Schlüsselmenge S genau dann, wenn

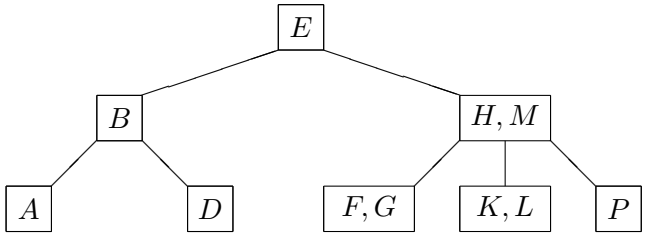
(a) T hat die (a, b)-Eigenschaft.

(b) Jeder Schlüssel in S wird in genau einem Knoten von T gespeichert. Ein Knoten speichert die ihm zugewiesenen Schlüssel in aufsteigender Reihenfolge.

(c) Jeder Knoten mit k Kindern speichert genau $k-1$ Schlüssel. Ein Blatt speichert höchstens $b-1$ Schlüssel und mindestens $a-1$ Schlüssel.

(d) Falls v kein Blatt ist und falls v die Schlüssel $x_1, ..., x_c$ (mit $x_1 < x_2 < \cdots < x_c$ und $c \leq b-1$) speichert, dann speichert der linkeste (bzw. rechteste) Teilbaum nur Schlüssel aus dem Intervall $(-\infty, x_1)$ (bzw. (x_c, ∞)). Der i.te Teilbaum (für $2 \leq i \leq c$) speichert nur Schlüssel aus dem Intervall (x_{i-1}, x_i).

Beispiel 5.6 (1) Für $a = 2$ und $b = 3$ erhalten wir **2-3 Bäume:**



(2) Für $a = 2$ und $b = 4$ erhalten wir **2-4 Bäume**, die in der Literatur als 2-3-4 Bäume bezeichnet werden. Beachte, dass 2-3 Bäume insbesondere auch 2-3-4 Bäume sind.

(3) (a, b)-Bäume mit $b = 2a-1$ heißen **B-Bäume**. (2-3 Bäume sind also ein Spezialfall von B-Bäumen.)

Die Programmierung von (a, b)-Bäumen ist recht kompliziert, wie wir bald sehen werden. (a, b)-Bäume liefern aber eine Datenstruktur mit guten worst-case Eigenschaften, die vor allem für B-Bäume von großem praktischen Interesse ist. B-Bäume werden nämlich dann eingesetzt, wenn ein Wörterbuch auf einem Hintergrundspeicher abgelegt ist. In diesem Fall müssen wir versuchen, die Anzahl der sehr langsamen Speicherzugriffe zu minimieren und können aber gleichzeitig ausnutzen, dass viele Daten in einem Speicherzugriff gelesen (bzw. geschrieben) werden können. Mit anderen Worten, man wähle a so, dass $2a-1$ Daten in einen Schritt eingelesen werden können.

Beachte, dass nur kleine Werte von a und b von Interesse sind, wenn wir einen (a, b)-Baum im Hauptspeicher halten: Die Vorteile einer geringen Tiefe (bei großen Werten von a und b) werden sofort durch die Mehrkosten einer linearen Schlüsselsuche in jedem Knoten aufgefressen. Deshalb werden auch vornehmlich nur 2 - 3 Bäume oder 2 - 3 - 4 Bäume verwandt, wenn vollständig im Hauptspeicher gearbeitet werden kann.

Wenn wir eine Suche mit Schlüssel x durchführen und gegenwärtig den Knoten v inspizieren, erlaubt Eigenschaft (d) von (a, b)-Bäumen, das für die Suche notwendige Kind von v zu bestimmen und im nächsten Schritt einzulesen. Damit benötigt eine Suche also höchstens (Tiefe (T)+1) viele Speicherzugriffe, wenn T der zu bearbeitende B-Baum ist. Wie drastisch können wir die Tiefe von T reduzieren (im Vergleich zu binären Bäumen)?

Satz 5.6 *T sei ein (a, b)-Baum mit n Knoten. Dann gilt*

$$\log_b(n) - 1 < \text{Tiefe}(T) < \log_a(\frac{n-1}{2}) + 1.$$

Die letzte Ungleichung gilt, falls $\text{Tiefe}(T) \geq 2$.

Beweis: Die Tiefe ist am geringsten, wenn jeder Knoten genau b Kinder hat. In Tiefe t können wir damit höchstens

$$1 + b + ... + b^t = \frac{b^{t+1} - 1}{b - 1} < b^{t+1}$$

Knoten erreichen. Für die Tiefe t eines (a, b)-Baums mit n Knoten gilt also

$$b^{t+1} > \frac{b^{t+1} - 1}{b - 1} \geq n \quad \text{und damit} \quad t > \log_b n - 1.$$

Die Tiefe ist am höchsten, wenn die Wurzel zwei Kinder und jeder innere Knoten a Kinder besitzt. Wir erhalten also in Tiefe t mindestens

$$1 + 2(1 + ... + a^{t-1}) = 1 + 2 \cdot \frac{a^t - 1}{a - 1}$$

Knoten. Für die Tiefe t eines (a, b)-Baums mit n Knoten gilt deshalb

$$1 + 2 \cdot \frac{a^t - 1}{a - 1} \leq n \quad \text{oder} \quad \frac{a^t - 1}{a - 1} \leq \frac{n - 1}{2}.$$

Da $a^{t-1} < \frac{a^t-1}{a-1}$ für $t \geq 2$, gilt deshalb

$$t < \log_a \left(\frac{n-1}{2} \right) + 1.$$

□

Wir beginnen jetzt mit der Implementierung der Operationen `lookup`, `insert` und `remove`. Für die `lookup`-Operation benutzen wir, wie schon oben erwähnt, Eigenschaft (d), um einen Suchpfad zu bestimmen. `lookup` benötigt also nur Tiefe $(T) + 1$ Speicherzugriffe. Die Tiefenersparnis von (a, b)-Bäumen im Vergleich zu binären Bäumen ist deshalb substantiell. Zum Beispiel erhalten wir mit Satz 5.6 für $a = 10^6$ und ein Terabyte ($n = 10^{12}$), dass

$$\text{Tiefe}(T) < \log_a \frac{10^{12} - 1}{2} + 1 \leq 2.$$

Damit genügen also drei Speicherzugriffe. Für ein Petabyte ($n = 10^{15}$) und sogar für ein Exabyte ($n = 10^{18}$) reichen vier Zugriffe.

Wir stellen die *bottom-up Version* der `insert`- und `remove`-Operation vor. Für die Operation `insert` $(x, info)$ bestimmen wir zuerst den dazugehörenden Suchpfad. Wenn x nicht im Baum vorhanden ist, wird der Suchpfad in einem Blatt v enden. (Wenn x vorhanden ist, brauchen wir nur den alten Infoteil zu überschreiben.) Sodann fügen wir x und info in v ein.

- **Fall 1:** v hat jetzt höchstens $b - 1$ Schlüssel.

  Wir sind fertig, da die (a, b)-Eigenschaft erfüllt ist.

- **Fall 2:** v hat jetzt b Schlüssel $x_1 < ... < x_b$.

  Die (a, b)-Eigenschaft ist verletzt. Wir ersetzen v durch zwei Knoten v_{links} und v_{rechts}, wobei v_{links} die Schlüssel

  $$x_1, ...x_{\lceil b/2 \rceil - 1} \quad (\lceil b/2 \rceil - 1 \text{ Schlüssel })$$

  und v_{rechts} die Schlüssel

  $$x_{\lceil b/2 \rceil + 1}, ..., x_b \quad (\lfloor b/2 \rfloor \text{ Schlüssel})$$

  besitzt. (a, b)-Bäume erfüllen die Ungleichung $2a - 1 \leq b$. Deshalb gilt $a - 1 \leq \lfloor \frac{b+1}{2} \rfloor - 1 \leq \lceil \frac{b}{2} \rceil - 1 \leq \lfloor \frac{b}{2} \rfloor$ und v_{links} und v_{rechts} besitzen die notwendige Mindestanzahl von Schlüsseln. Schlüssel $x_{\lceil b/2 \rceil}$ fügen wir in entsprechender Stelle unter den Schlüsseln des Elternknotens w von v ein: Dieser Schlüssel erlaubt eine Unterscheidung von v_{links} und v_{rechts}. Wir sind fertig, falls w jetzt höchstens b Kinder besitzt. Möglicherweise hat w aber jetzt $b + 1$ Kinder. Was tun? Wiederhole Fall 2 für w!

  Dieses rekursive Zerlegungs-Verfahren wird spätestens an der Wurzel enden. Wenn wir auch die Wurzel zerlegen müssen, müssen wir eine neue Wurzel schaffen, die die erlaubte Zahl von zwei Kindern besitzt. In diesem Fall erhöht sich also die Tiefe des Baums.

In der *top-down Version* der `insert`-Operation zerlegen wir bereits alle Knoten mit b Kindern auf dem Weg von der Wurzel zum Blatt. Damit sparen wir das Zurücklaufen, müssen aber eine möglicherweise unnötige Tiefenerhöhung in Kauf nehmen. Wie auch immer wir die `insert`-Operation ausführen, wir kommen mit höchstens 2 (Tiefe $(T) + 1$) Speicherzugriffen aus.

Als Letztes ist die Operation `remove` (x) zu besprechen. Zuerst müssen wir nach x suchen. Angenommen wir finden x im Knoten v. Wenn v kein Blatt ist, suchen wir zuerst den kleinsten Schlüssel y mit $x \leq y$: Schlüssel y wird sich im linkesten Blatt b des entsprechenden Teilbaums von v befinden. Wir ersetzen in v den Schlüssel x durch y: Damit ist x entfernt, aber das Blatt b hat einen Schlüssel verloren. Wenn v ein Blatt ist, entfernen wir x (und auch v verliert einen Schlüssel).

Sei $w \in \{b, v\}$ das jeweilige Blatt mit Schlüsseln $x_1 < \cdots < x_k$

- **Fall 1:** w hat jetzt mindestens $a - 1$ Schlüssel.
  Wir sind fertig, da die (a, b)-Eigenschaft erfüllt ist.

- **Fall 2:** w hat $a - 2$ Schlüssel.
  Die $(a, b)-$Eigenschaft ist damit in w verletzt. Zuerst begeben wir uns auf „Schlüsselklau“.

  Fall 2.1: Der linke oder rechte Geschwisterknoten hat mindestens a Schlüssel.
  Sei dies z. B. der linke Geschwisterknoten w' mit Schlüsseln

  $$y_1 < \cdots < y_a < \cdots < y_{k'}.$$

  Sei z der Schlüssel des Elternknotens u, der w' und w „trennt“, also $y_{k'} < z < x_1$. w klaut den Schlüssel z und hat damit die erforderliche Mindestanzahl von $a-1$ Schlüsseln. Schlüssel z wird großzügigerweise durch Schlüssel $y_{k'}$ ersetzt und die $(a, b)-$Eigenschaft ist wiederhergestellt: Fertig!

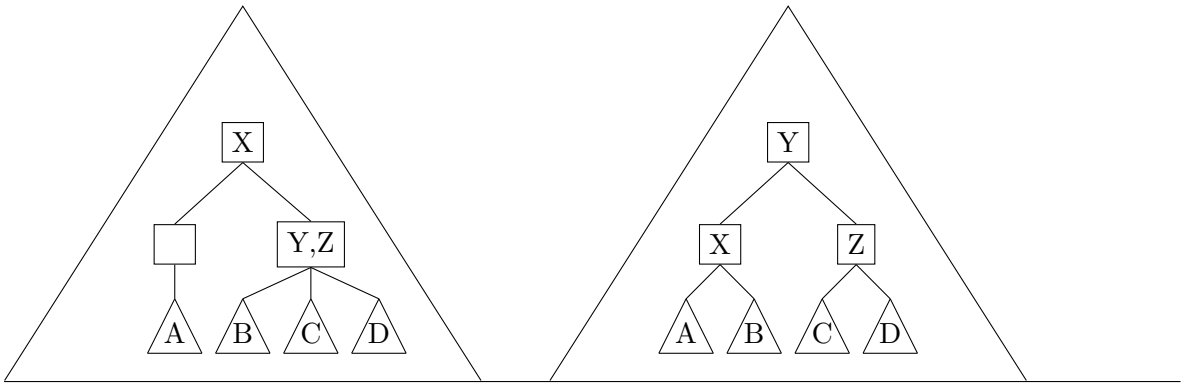
  Fall 2.2: Beide Geschwisterknoten besitzen nur $a - 1$ Schlüssel.
  Wir betrachten wieder den linken Geschwisterknoten w' mit den $a - 1$ Schlüsseln

  $$y_1 < \cdots < y_{a-1}.$$

Wir möchten w und w' zu einem neuen Knoten w_1 verschmelzen. Da dann der trennende Schlüssel z des Elternknotens u überflüssig ist, speichert w_1 die Schlüssel

$$y_1 < \cdots < y_{a-1} < z < x_1 < \cdots < x_{a-2}.$$

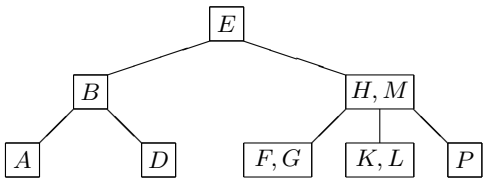
w_1 hat damit $2a - 2 \leq b - 1$ Schlüssel und die Höchstanzahl wird nicht überschritten. Wir sind fertig, wenn u jetzt mindestens $a - 1$ Schlüssel hat. Wenn nicht, müssen wir das Vorgehen von Fall 2 rekursiv wiederholen!



Aufgabe 83
Wir betrachten die remove(x) Operation für (a, b)-Bäume. Bisher haben wir die „Schlüsselklau-Regel“ (Fall 2.1) wie auch die „Verschmelzungsregel“ (Fall 2.2) nur für Blätter beschrieben.

(a) **Wie** sind die beiden Regeln für innere Knoten zu verallgemeinern?

(b) Betrachte den folgenden (2,3)-Baum. **Stelle** den Ablauf der Operation remove(D) **dar**.



Damit benötigt die `remove`-Operation höchstens 4 (Tiefe$(T) + 1$) Speicherzugriffe. Tiefe(T) Zugriffe sind auf dem Weg nach unten erforderlich. Weiterhin sind für jeden Knoten auf dem Weg nach oben 3 Zugriffe (Schreiben des Knotens und Lesen/Schreiben eines seiner Geschwisterknoten) erforderlich. Wir erhalten:

Satz 5.7 *Sei T ein (a, b)-Baum. Dann genügen Tiefe$(T)+1$ Speicherzugriffe für eine* `lookup`*-Operation, 2(Tiefe$(T) + 1$) Speicherzugriffe für eine* `insert`*-Operation und 4 (Tiefe$(T) + 1$) Speicherzugriffe für die Operation* `remove`*.*

Beachte, dass Tiefe $(T) < \log_a(\frac{n-1}{2}) + 1$ gilt, falls der Baum T aus n Knoten besteht.

Aufgabe 84
Zeichne den $(2, 3)$-Baum B_1, der durch folgende Operationen aus dem leeren Baum entsteht:

`insert`(1)	`insert`(4)	`insert`(6)	`insert`(9)	`insert`(5)	`insert`(11)
`insert`(2)	`insert`(3)	`insert`(12)	`insert`(10)	`insert`(7)	`insert`(8)

Zeichne den Baum B_2, der aus B_1 durch die Operation `remove`(4) entsteht.

Aufgabe 85
Im Binpacking Problem sind n Objekte $x_1, \ldots, x_n$ mit den Gewichten $g_1, \ldots, g_n$ ($g_i \in [0, 1]$) gegeben. Die Objekte sind in möglichst wenige Behälter mit jeweils Kapazität 1 zu verteilen.

(a) Die *Best-Fit* Strategie ist eine Heuristik zur Berechnung einer möglichst guten, aber im Allgemein nicht optimalen Lösung. Best Fit fügt ein Objekt stets in den Behälter ein, der das Objekt „am knappsten“ noch aufnehmen kann.

Genauer: Best-Fit arbeitet die Objekte, beginnend mit dem ersten Objekt x_1, nacheinander ab und verwaltet eine anfänglich leere Menge B von angebrochenen Behältern. Bei der Bearbeitung des i-ten Objektes x_i prüft Best Fit, ob angebrochene Behälter existieren, die das Objekt x_i noch aufnehmen können.

- Falls solche Behälter *nicht existieren*, dann wird das Objekt in einen neuen Behälter eingefügt, der dann in die Menge B der angebrochenen Behälter aufgenommen wird.
- Falls angebrochene Behälter *existieren*, die das Objekt x_i aufnehmen können, wählt Best Fit unter ihnen denjenigen aus, der bereits das *größte* Gewicht beinhaltet, d.h. denjenigen Behälter mit der geringsten Restkapazität.

Beschreibe eine Implementierung von Best-Fit und analysiere die Laufzeit für n Objekte. Konzentriere Dich bei der Beschreibung insbesondere auf eine geeignete Datenstruktur für die Menge B. Die Laufzeit $O(n \log_2 n)$ ist erreichbar.

(b) Die *First-Fit* Strategie fügt ein Objekt stets in den ersten Behälter ein, der das Objekt aufnehmen kann. Falls kein Behälter das Objekt aufnehmen kann, wird ein neuer Behälter geöffnet. Beschreibe eine Implementierung von First-Fit, so dass Laufzeit $O(n \log_2 n)$ nicht überschritten wird.

5.5 Hashing

Das Wörterbuchproblem wird einfacher, wenn die Menge U der möglicherweise einzufügenden Daten in den Hauptspeicher passt. In diesem Fall löst die **Bitvektor-Datenstruktur** das Wörterbuchproblem:

> Ein boolesches Array wird angelegt, das für jedes Element $u \in U$ eine Zelle $f(u)$ besitzt. In der Zelle $f(u)$ wird vermerkt, ob u präsent ist.

Bis auf die Berechnung der Funktion $f(u)$ gelingt damit die Ausführung einer jeden lookup-, insert- oder remove-Operation in konstanter Zeit!

Selbst bei einem kleinen Universum U ist aber die Berechnung einer geeigneten Funktion f möglicherweise ein komplexes Problem. Zudem ist in praktischen Anwendungen im Allgemeinen das Universum der möglichen Schlüssel groß wie das folgende Beispiel zeigt: Betrachten wir den Fall von Kundendateien (sogar von Firmen mit kleinem Kundenstamm). Bei Angabe des Nachnamens des Kundens als Schlüssel und bei der Annahme, dass nur Nachnamen der Länge höchstens 10 auftreten, besteht die Menge der möglichen Schlüssel aus $26^{10} \geq 2^{10} \cdot 10^{10} \geq 10^3 \cdot 10^{10} = 10^{13}$ Elementen, also aus 10 Billionen Elementen!

Mit **Hashing** versuchen wir dennoch, die Eigenschaften der Bitvektor-Datenstruktur so weit wie möglich zu bewahren. Sei U die Menge aller möglichen Schlüssel und sei m die Größe einer intern abspeicherbaren Tabelle. Dann nennen wir eine Funktion

$$h : U \to \{0, 1, ..., m-1\}$$

eine Hashfunktion. Wir können damit z. B. die Operation `insert` (x, info) sehr einfach implementieren:

- Berechne $h(x) = i$ und
- trage $(x, info)$ in Zelle i der Tabelle ein.

Allerdings haben wir keine Gewähr, dass Zelle i auch tatsächlich frei ist: Es gibt durchschnittlich $|U|/m$ Schlüssel, die auf Zelle i abgebildet werden können! Sollte Zelle i besetzt sein, sprechen wir von einer *Kollision*. Eine Hashing-Datenstruktur muss deshalb angeben, wie Kollisionen zu behandeln sind. In **Hashing mit Verkettung** wird für jede Zelle i eine anfänglich leere Liste angelegt. Einzufügende Schlüssel mit Hashwert i werden dann nach linearer Suche in die Liste von i eingefügt. (Da die Liste sowieso linear zu durchlaufen ist, bedeutet es keinen Mehraufwand, die Listen sortiert zu halten. Dies führt bei einer erfolglosen `lookup` Operation zu einer durchschnittlichen Halbierung der Suchzeit.)

In **Hashing mit offener Adressierung** wird unmittelbar in die Tabelle eingetragen. Was tun, wenn $h(x) = i$ und Zelle i besetzt ist? Wir „hashen“ mit einer weiteren Hashfunktion. Aber auch die dann erhaltene Zelle kann besetzt sein und deshalb wird eine Folge

$$h_0, ..., h_{m-1} : U \to \{0, ..., m-1\}$$

zur Verfügung gestellt: Wenn wir $i-$mal auf eine besetzte Zelle gestoßen sind, wenden wir beim $(i+1)$-ten Versuch die Funktion h_i an und versuchen unser Glück mit Zelle $h_i(x)$.

Hashing mit offener Adressierung ist also speicher-effizienter als Hashing mit Verkettung. Allerdings müssen wir vorher eine gute obere Schranke für die maximale Anzahl einzufügender Elemente kennen. (Ein Ausweg ist allerdings der Aufbau sukzessiv größerer Tabellen. Wenn die Tabellengröße um den Faktor 2 wächst, ist die Reorganisationszeit gegen die Anzahl der zwischenzeitlich ausgeführten Operationen amortisiert.) Ein weiterer Nachteil von Hashing mit offener Adressierung ist das Laufzeitverhalten bei fast gefüllter Tabelle. Dies werden wir im folgenden genauer untersuchen.

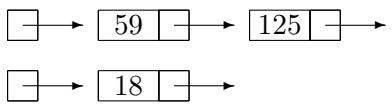
Die Wahl des geeigneten Hashing-Verfahrens hängt damit von den Umständen ab: Wenn Speicherplatz fehlt, ist Hashing mit offener Adressierung die einzige Wahl. Ansonsten ist das generell etwas schnellere Hashing mit Verkettung vorzuziehen.

5.5.1 Hashing mit Verkettung

Wir besprechen zuerst die Wahl der Hashfunktion h. Eine beliebte und gute Wahl ist $h(x) = x \bmod m$. $h(x)$ kann schnell berechnet werden und wird bei „guter“ Wahl von m zufriedenstellende Ergebnisse liefern. Was ist eine schlechte Wahl von m?

Angenommen unsere Schlüssel sind Character Strings und wir wählen m als eine Zweierpotenz. Wir denken uns dann den Schlüssel durch seine interne Bitdarstellung x repräsentiert. Als Konsequenz werden alle Character-Strings mit gleicher Endung durch h auf dieselbe Zelle abgebildet. Da bestimmte Endungen wahrscheinlicher als andere sein werden, haben wir somit eine große Menge von Kollisionen provoziert. Primzahlen (mit großem Abstand von der nächstliegenden Zweierpotenz) haben sich hingegen als eine gute Wahl erwiesen.

Beispiel 5.7 Sei $h(x) = (x \bmod 11)$
Die Operationen `insert` (59), `insert` (18) und `insert` (125) führen auf die Tabelle



Die Operation lookup (26) benötigt nur einen Suchschritt: Schlüssel 59 wird gefunden und es wird geschlossen, dass 26 nicht präsent ist (denn jede Liste ist aufsteigend sortiert).

Angenommen, es befinden sich n Schlüssel in der Tabelle. Wir sagen dann, dass $\lambda = \frac{n}{m}$ der **Auslastungsfaktor** der Tabelle ist. Betrachten wir die Laufzeit einer `insert`(x), `remove`(x) oder `lookup`(x) Operation für eine mit n Schlüsseln gefüllte Hash-Tabelle. *Bestenfalls* ist die Liste für $h(x) = i$ leer und wir erreichen damit eine konstante Laufzeit. *Schlimmstenfalls* sind alle n Schlüssel auf die Liste von i verteilt und die worst-case Laufzeit $\Theta(n)$ folgt. Was ist die *erwartete Laufzeit* einer einzelnen Operation?

Wir betrachten das allgemeine Szenario. Eine Hash-Tabelle mit m Zellen ist bereits irgendwie (z.B. bösartig) mit n Schlüsseln gefüllt worden. Über das angefragte Element und die herangezogene Hashfunktion machen wir die folgenden Annahmen.

(1) jedes Element $x \in U$ hat die Wahrscheinlichkeit $\frac{1}{|U|}$ als Operand in einer Operation aufzutreten.

(2) Die Hashfunktion h hat für jedes i $(0 \leq i < m)$ die Eigenschaft, dass

$$|\{x \in U \,:\, h(x) = i\}| \;\in\; \left\{ \lfloor \frac{|U|}{m} \rfloor, \lceil \frac{|U|}{m} \rceil \right\}$$

(Das heißt, die Hashfunktion „streut“ die Schlüssel regelmäßig. Beachte, dass die Hashfunktion $h(x) = (x \bmod m)$ diese Eigenschaft besitzt.)

Die Wahrscheinlichkeit p_i, dass ein zufällig gezogener Schlüssel auf die Zelle i gehasht wird, ist auf Grund der Streubedingung höchstens

$$p_i \leq \frac{\lceil \frac{|U|}{m} \rceil}{|U|} \leq \frac{\frac{|U|}{m} + 1}{|U|} \leq \frac{1}{m} + \frac{1}{|U|}.$$

Sei n_i die Zahl der Schlüssel im Universum U, die in Zelle i gehast werden. Jetzt können wir die erwartete Listenlänge $\sum_{i=0}^{m-1} p_i \cdot n_i$ nach oben abschätzen:

$$\begin{aligned} \sum_{i=0}^{m-1} p_i \cdot n_i \leq \sum_{i=0}^{m-1} \left(\frac{1}{m} + \frac{1}{|U|} \right) \cdot n_i &= \left(\frac{1}{m} + \frac{1}{|U|} \right) \cdot \sum_{i=0}^{m-1} n_i \\ &= \left(\frac{1}{m} + \frac{1}{|U|} \right) n \\ &= \lambda + \frac{n}{|U|}. \end{aligned}$$

Man beachte das die Hash-Tabelle bereits mit n Schlüsseln belegt ist. Damit gilt $n \leq |U|$ und insbesondere ist $\frac{n}{|U|} \leq 1$. In allen Fällen von praktischer Relevanz wird n sehr viel kleiner als $|U|$ sein und der zweite Bruch kann vernachlässigt werden. Ist $|U|$ ein Vielfaches von m, so ist $\lceil \frac{|U|}{m} \rceil = \frac{|U|}{m}$ und der zweite Summand taucht gar nicht erst auf.

Lemma 5.8 *Die erwartete Länge einer Liste für Hashing mit Verkettung ist*

(a) λ*, falls* $|U|$ *ein Vielfaches von* m *ist und*

(b) höchstens $\lambda + \frac{n}{|U|} \leq \lambda + 1$ *im allgemeinen Fall.*

Die erwartete Listenlänge ist die dominierende Größe für die Laufzeitbestimmung von `lookup`, `insert` und `remove`. Hier ist jeweils der Wert der Hashfunktion zu bestimmen (Konstantzeit), die Liste auf der Suche nach dem Schlüssel abzuschreiten (Laufzeit höchstens Listenlänge) und schließlich das Einfügen, Löschen oder Lesen lokal vorzunehmen (wieder Konstantzeit). Also dominiert die Listenlänge und wir erhalten:

Satz 5.9 *Die erwarteten Laufzeiten der Operationen* `lookup`, `insert` *und* `remove` *sind jeweils durch* $\lambda + O(1)$ *beschränkt.*

5.5.2 Hashing mit offener Adressierung

Statt mit einer einzelnen Hashfunktion müssen wir jetzt mit einer Folge von Hashfunktionen arbeiten. In der Methode des **linearen Austestens** wird die Folge

$$h_i(x) = (x + i) \mod m$$

benutzt. Für festes x wird also die jeweils nächste Zelle untersucht.

Im linearen Austesten besteht die Gefahr, dass Daten „zusammenklumpen“: Die Daten besetzen ein Intervall der Tabelle. Wenn ein späteres Element y in ein Intervall hasht, wird y zwangsläufig an das Ende des Intervalls eingefügt. Das Intervall wächst somit und beeinträchtigt die Laufzeit nachfolgender Operationen. So kann gezeigt werden, dass im Durchschnitt $\frac{1}{2} \cdot (1 + \frac{1}{(1-\lambda)^2})$ Zellen getestet werden. Allerdings ist lineares Austesten „cache-freundlich“ und bleibt deshalb ein interessantes Hashing-Verfahren.

Aufgabe 86

In dieser Aufgabe betrachten wir Hashing mit offener Adressierung und linearem Austesten. Gegeben ist eine Hashtabelle mit m Zellen, wobei m eine durch vier teilbare Zahl sei. In diese Tabelle werden $\frac{m}{2}$ zufällig gezogene Schlüssel aus einem Universum U mittels einer Hashfunktion h eingefügt. Es darf angenommen werden, dass ein zufällig gezogener Schlüssel des Universums stets mit der Wahrscheinlichkeit $\frac{1}{m}$ auf eine konkrete Zelle i gehasht wird.

Wir beobachten die beiden folgenden Verteilungen:

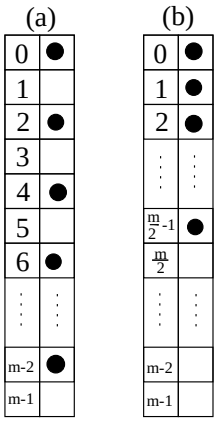


Abbildung 5.1: Bei (a) ist jede zweite Zelle besetzt. Im Fall (b) ist die erste Hälfte der Tafel besetzt.

(a) **Bestimme** die Wahrscheinlichkeit für die Verteilung (a).

(b) **Zeige**, dass die Wahrscheinlichkeit für (b) mindestens um einen multiplikativen Faktor der Form

$$c^m \text{ mit } c > 1$$

größer ist.

Hinweis: Man kann zum Beispiel so vorgehen, dass man die obere Hälfte der Tabelle in Zweiergruppen einteilt und die Anzahl der Eingabefolgen bestimmt, bei denen es nicht zu einem Weiterschieben der Schlüssel von einem Zweierblock zum nächsten kommt.

(c) **Bestimme** in beiden Szenarien die erwartete Anzahl an Versuchen beim Einfügen eines weiteren Elementes.

Dieses Klumpungsphänomen wird in der Methode des **doppelten Hashing** vermieden: Wir benutzen zwei Hashfunktionen f und g und verwenden die Folge

$$h_i(x) = (f(x) + i \cdot g(x)) \mod m.$$

Zum Beispiel liefern die simplen Hashfunktionen

$$f(x) = x \mod m$$

und

$$g(x) = m^* - (x \mod m^*)$$

gute experimentelle Ergebnisse. Man sollte m als Primzahl wählen und $m^* < m$ fordern. (Beachte, dass $g(x)$ stets von Null verschieden ist. Weiterhin garantiert die Primzahleigenschaft, dass

$$\{h_i(x) \,:\, 0 \leq i < m\} = \{0, ..., m-1\}$$

gilt. Es werden also alle Zellen aufgezählt.)

Die Operationen `lookup` und `insert` lassen sich für jede Folge von Hashfunktionen leicht implementieren, die Operation `remove` bereitet aber Kopfzerbrechen. Wird nämlich nach Einfügen des Schlüssels x (in Zelle $h_1(x)$) der Schlüssel in Zelle $h_0(x)$ entfernt, dann hat die Operation lookup (x) ein Problem: Ist x nicht präsent, weil Zelle $h_0(x)$ leer ist oder muss weitergesucht werden? Natürlich können wir dieses Problem leicht durch Einfügen einer „entfernt“ Markierung lösen, die zu einer Weiterführung der Suche veranlasst. Die erwartete Laufzeit einer erfolglosen Suche wird aber anwachsen. Deshalb sollte lineares Austesten oder doppletes Hashing vermieden werden, wenn erwartet wird, dass viele Daten entfernt werden müssen.

Um eine Idee zu erhalten, wie sich der Auslastungsfaktor in Abhängigkeit von der Tabellengröße m und der Schlüsselzahl n auf die Laufzeit auswirkt, betrachten wir die erwartete Laufzeit einer erfolglosen `lookup`-Operation. Für die Analyse machen wir die beiden folgenden Annahmen:

- Jeder Schlüssel $x \in U$ tritt mit Wahrscheinlichkeit $\frac{1}{:U:}$ als Operand einer Operation auf.
- Für jedes $x \in U$ ist die Folge

  $$(h_0(x), h_1(x), ..., h_{m-1}(x)) = \pi_x$$

  eine Permutation von $\{0, 1, ..., m-1\}$ und jede Permutation tritt für $\frac{:U:}{m!}$ Schlüssel in U auf.

Angenommen, wir haben n verschiedene Einfügungen durchgeführt. Jede der $\binom{m}{n}$ möglichen Tabellenbelegungen ist gleichwahrscheinlich: Die n eingefügten Schlüssel sind zufällig zu wählen (gemäß der ersten Annahme) und damit auch die von ihnen besetzten Zellen (gemäß der zweiten Annahme). Die erfolglose Operation `lookup`(x) sei auszuführen

Mit Wahrscheinlichkeit $\frac{m-n}{m} = 1 - \frac{n}{m} = 1 - \lambda$ stossen wir bereits im ersten Versuch auf eine freie Zelle. Die erwartete Wartezeit bis zum ersten Auffinden der ersten freien Zelle ist dann aber $\frac{1}{1-\lambda}$ mit Lemma 2.17.

Satz 5.10 *Eine anfänglich leere Tabelle mit m Zellen werde durch n Einfügungen gefüllt. Die erwartete Laufzeit einer erfolglosen* `lookup`*-Operation ist dann durch*

$$\frac{1}{1-\lambda}$$

nach oben beschränkt. Dabei haben wir angenommen, dass die beiden obigen Annahmen gelten und dass $\lambda = \frac{n}{m}$.

Die beiden obigen Annahmen sind sehr stark und können von realistischen Hash-Funktionen nicht erreicht werden. Um so ernster sollte man die Aussage des Satzes nehmen: Eine zu 90 % (bzw. 99 %) gefüllte Tabelle erfordert 10 (bzw. 100) Schritte. **Fazit:** Man sollte stets garantieren, dass der Auslastungsfaktor nicht zu gross wird; zum Beispiel könnte man nach Überschreiten des Auslastungsfaktors $\frac{1}{2}$ in einem Schritt des Großreinemachens die Schlüssel in eine doppelt so große Tabelle umlagern. Dieses Fazit gilt nur für Hashing mit offener Adressierung. Hashing mit Verkettung verkraftet eine Überladung der Tabelle mit Leichtigkeit, denn die erwartete Laufzeit ist $\lambda + O(1)$.

Aufgabe 87

Wir betrachten erneut das Hashen zufällig gezogener Schlüssel in eine Hashtabelle der Größe m. Auch hier unterstellen wir, dass ein zufällig gezogener Schlüssel mit einer Wahrscheinlichkeit von $\frac{1}{m}$ auf eine konkrete Zelle i gehasht wird. Es werden k zufällig gezogene Schlüssel eingefügt. Sei $p_{k,m}$ die Wahrscheinlichkeit, dass keine Kollisionen auftreten, wenn wir k zufällig gezogene Schlüssel in die Hashtabelle der Größe m einfügen. Zeige:

$$\left(1 - \frac{k}{m}\right)^k \leq p_{k,m} \leq e^{-(k-1)\cdot k/(2m)}.$$

(Also werden wir bereits mit Kollisionen rechnen müssen, wenn $k \approx \sqrt{m}$.)

Hinweis: Es gilt stets $(1 - x) \leq e^{-x}$.

5.5.3 Cuckoo Hashing

Diesmal arbeiten wir mit zwei Hashtabellen T_1 und T_2 sowie zwei Hashfunktionen h_1 und h_2. Zu jedem Zeitpunkt wird ein gespeicherter Schlüssel x entweder in Tabelle T_1 und Zelle $h_1(x)$ oder in Tabelle T_2 und Zelle $h_2(x)$ abgelegt. Damit ist die Durchführung einer lookup- oder remove-Operation sehr einfach und schnell, die insert(x) Operation ist hingegen aufwändiger:

Algorithmus 5.3 **(Cuckoo Hashing: Insert)**

Wiederhole bis zu M mal:

- Speichere x in Tabelle T_1 und Zelle $h_1(x)$ ab.
- Wenn die Zelle $h_1(x)$ in Tabelle T_1 durch einen Schlüssel y belegt war, dann füge y in Tabelle T_2 und Zelle $h_2(y)$ ein.

- Wenn die Zelle $h_2(y)$ in Tabelle T_2 durch einen Schlüssel z belegt war, dann setze $x = z$ und beginne eine neue Iteration.

Warum der Name Cuckoo Hashing? Eine bestimmte Kuckuck Art wirft ein oder mehrere Eier aus einem fremden Nest und legt ein eigenes Ei hinzu.

Cuckoo Hashing hasht direkt in die Tabellen und ist damit dem Schema der offenen Adressierung zuzurechnen; ganz im Gegensatz zum linearen Austesten oder zum doppelten Hashing ist die remove-Operation aber mit Leichtigkeit implementierbar. Der wesentliche Vorteil des Cuckoo Hashing gegenüber allen anderen bisher betrachteten Hashing Verfahren ist die Schnelligkeit mit der eine lookup- oder remove-Operation durchgeführt wird: Höchstens zwei Zellen werden inspiziert.

Da wir in zwei Tabellen einfügen, ist der Auslastungsfaktor λ durch $\frac{1}{2}$ beschränkt. Es kann gezeigt werden, dass die erwartete Anzahl getesteter Zellen höchstens $O(1 + \frac{1}{0.5-\lambda})$ beträgt und damit ist auch die insert-Operation recht schnell. Wir müssen allerdings noch klären, was zu tun ist, wenn die insert Prozedur nach M Iterationen nicht erfolgreich ist: In einem solchen Fall wird der zuletzt herausgeworfene Schlüssel in einer „Liste von Fehlversuchen“ abgelegt.

Welche Hashfunktionen sollten benutzt werden? Wähle drei Parameter $0 < a_1, a_2, a_3 < m \cdot 2^w$ zufällig und setze $h(x) = f_{a_1}(x) \oplus f_{a_2}(x) \oplus f_{a_3}(x)$, wobei

$$f_a(x) = (a \cdot x \bmod m \cdot 2^w) \operatorname{div} 2^w.$$

Um eine schnelle Auswertung zu gewährleisten, sollte auch m als eine Zweierpotenz gewählt werden.

5.5.4 Universelles Hashing

Offensichtlich können wir für jede Hashfunktion, ob für Hashing mit Verkettung oder für Hashing mit offener Addressierung, eine worst-case Laufzeit von $\Theta(n)$ erzwingen. Wir müssen also Glück haben, dass unsere Operationen kein worst-case Verhalten zeigen.

Aber was passiert, wenn wir mit einer Klasse H von Hashfunktionen arbeiten, anstatt mit einer einzelnen Hashfunktion? Zu Beginn der Berechnung wählen wir **zufällig** eine Hashfunktion $h \in H$ und führen Hashing mit Verkettung mit dieser Hashfunktion durch. Die Idee dieses Vorgehens ist, dass eine einzelne Hashfunktion durch eine bestimmte Operationenfolge zum Scheitern verurteilt ist, dass aber die meisten Hashfunktion mit Bravour bestehen. Die erste Frage: Was ist eine geeignete Klasse H?

Definition 5.5 Eine Menge $H \subseteq \{h \mid h : U \to \{0, ..., m-1\}\}$ ist **c-universell**, falls für alle $x, y \in U$ mit $x \neq y$ gilt, dass

$$|\{h \in H \ : \ h(x) = h(y)\}| \leq c \cdot \frac{|H|}{m}$$

Wenn H c-universell ist, dann gilt

$$\frac{|\{h \in H \ : \ h(x) = h(y)\}|}{|H|} \leq \frac{c}{m}$$

und es gibt somit keine zwei Schlüssel, die mit Wahrscheinlichkeit größer als $\frac{c}{m}$ auf die gleiche Zelle abgebildet werden. Gibt es c-universelle Klassen von Hashfunktionen für kleine Werte von c? Offensichtlich, man nehme zum Beispiel alle Hashfunktionen und wir erhalten $c =$

1. Diese Idee ist alles andere als genial: Wie wollen wir eine zufällig gewählte, **beliebige** Hashfunktion auswerten? Der folgende Satz liefert eine kleine Klasse von leicht auswertbaren Hashfunktionen.

Satz 5.11 *Es sei* $U = \{0, 1, 2, ..., p-1\}$ *für eine Primzahl* p*. Dann ist*

$$H = \{h_{a,b} \ : \ 0 \leq a, b < p, \ \ h_{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$$

c-universell mit $c = (\lceil \frac{p}{m} \rceil \ / \ \frac{p}{m})^2$.

Die Hashfunktionen in H folgen dem Motto „erst durchschütteln ($x \mapsto y = (ax + b) \bmod p$) und dann hashen ($y \mapsto y \bmod m$)“.

Beweis: Seien $x, y \in U$ beliebig gewählt. Es gelte $h_{a,b}(x) = h_{a,b}(y)$.
Dies ist genau dann der Fall, wenn es $q \in \{0, ..., m-1\}$ und $r, s \in \{0, ..., \lceil \frac{p}{m} \rceil - 1\}$ gibt mit

$$(*) \qquad \begin{array}{rcrcrcrcr} ax & + & b & = & q & + & r \cdot m & \bmod & p \\ ay & + & b & = & q & + & s \cdot m & \bmod & p. \end{array}$$

Für feste Werte von r, s und q ist dieses Gleichungssystem (mit den Unbekannten a und b) eindeutig lösbar. (Da p eine Primzahl ist, müssen wir ein Gleichungssystem über dem Körper Z_p lösen. Die Matrix des Gleichungssystem ist

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$$

und damit regulär, denn $x \neq y$). Die Abbildung, die jedem Vektor (a, b) (mit $h_{a,b}(x) = h_{a,b}(y)$), den Vektor (q, r, s) mit Eigenschaft $(*)$ zuweist, ist somit bijektiv. Wir beachten, dass es $m \cdot (\lceil \frac{p}{m} \rceil)^2$ viele Vektoren (q, r, s) gibt und erhalten deshalb

$$\begin{aligned} |\{h \in H \ : \ h(x) = h(y)\}| &\leq m \cdot \left(\lceil \frac{p}{m} \rceil \right)^2 \\ &= \left(\lceil \frac{p}{m} \rceil / \frac{p}{m} \right)^2 \frac{p^2}{m} = c \cdot \frac{|H|}{m} \end{aligned}$$

denn $c = (\lceil \frac{p}{m} \rceil \ / \ \frac{p}{m})^2$ und $|H| = p^2$. □

Die Annahme „$U = \{0, ..., p-1\}$ für eine Primzahl p“ sieht auf den ersten Blick realitätsfremd aus, denn wir wollen ja beliebige Objekte abspeichern. Jedoch ist für uns nur die interne Binärdarstellung interessant und deshalb können wir tatsächlich stets annehmen, dass $U = \{0, ..., p-1\}$. Es bleibt die Forderung, dass p prim ist. Diese Forderung ist in den seltensten Fällen erfüllt, aber wer hindert uns daran U so zu vergrößern, dass die Mächtigkeit von der Form „Primzahl“ ist? Niemand!

Sei H eine beliebige c-universelle Klasse von Hashfunktionen und sei eine *beliebige* Folge von $n-1$ `insert`-, `remove`- und `lookup`-Operationen vorgegeben. Wir betrachten die n-te Operation mit Operand x und möchten die erwartete Laufzeit dieser Operation bestimmen.

Beachte, dass der Erwartungswert über alle Hashfunktionen in H zu bestimmen ist. Die Operationenfolge ist fest vorgegeben und könnte möglicherweise von einem cleveren Gegner gewählt worden sein, um unser probabilistisches Hashingverfahren in die Knie zu zwingen. Der Gegner kennt sehr wohl die Klasse H, aber nicht die zufällig gewählte Funktion $h \in H$.

Sei S die Menge der vor Ausführung der n-ten Operation präsenten Elemente aus U. Die erwartete Laufzeit der n-ten Operation bezeichnen wir mit E_n. Die erwartete Anzahl von Elementen, mit denen der Operand x der n-ten Operation kollidiert, sei mit K_n bezeichnet. Dann gilt

$$\begin{aligned} E_n \;=\; 1 + K_n \;&=\; 1 + \frac{1}{|H|} \sum_{h \in H} |\{y \in S \,:\, h(x) = h(y)\}| \\ &=\; 1 + \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S, h(x)=h(y)} 1 \\ &=\; 1 + \frac{1}{|H|} \sum_{y \in S} \sum_{h \in H, h(x)=h(y)} 1 \\ &\leq\; 1 + \frac{1}{|H|} \sum_{y \in S} c \cdot \frac{|H|}{m}, \end{aligned}$$

denn die Klasse H ist c-universell! Und damit folgt

$$\begin{aligned} E_n \;&\leq\; 1 + c \cdot \frac{|S|}{m} \\ &\leq\; 1 + c \cdot \frac{n-1}{m}. \end{aligned}$$

Jetzt können wir auch sofort die erwartete Laufzeit E der ersten n Operationen bestimmen:

$$\begin{aligned} E \;&=\; E_1 + ... + E_n \\ &\leq\; \sum_{i=1}^{n} \left(1 + c \cdot \frac{i-1}{m}\right) \\ &=\; n + \frac{c}{m} \cdot \sum_{i=1}^{n} (i-1) \\ &=\; n + \frac{c}{m} \cdot \frac{n \cdot (n-1)}{2} \\ &\leq\; n \cdot (1 + \frac{c}{2} \cdot \frac{n}{m}) \end{aligned}$$

und die erwartete Laufzeit ist linear, wenn $n = O\ (m)$.

Satz 5.12 *Sei H eine c-universelle Klasse von Hashfunktionen. Eine Folge von n Operationen sei beliebig, zum Beispiel in worst-case Manier, gewählt. Dann ist die erwartete Laufzeit aller n Operationen durch*

$$n \left(1 + \frac{c}{2} \cdot \frac{n}{m}\right)$$

nach oben beschränkt.

Aufgabe 88
Sei $S \subseteq U$ eine Schlüsselmenge mit $|S| = n$. H sei eine Klasse c-universeller Hashfunktionen für Tabellengröße m. $B_i(h) = h^{-1}(i) \cap S$ ist die Menge der von h auf i abgebildeten Schlüssel.

(a) Zeige, daß bei zufälliger Wahl der Hashfunktion h aus H gilt:

$$E[\sum_i (|B_i(h)|^2 - |B_i(h)|)] \leq \frac{cn(n-1)}{m}.$$

(b) Die Ungleichung von Markov besagt: $Prob(x \geq aE[x]) \leq 1/a$ für alle Zufallsvariablen x und alle $a > 0$. **Zeige:**

$$Prob(\sum_i |B_i(h)|^2 \geq n + a\frac{cn(n-1)}{m}) \leq 1/a.$$

(c) Bestimme m, so daß eine zufällige Hashfunktion mit Wahrscheinlichkeit $1/2$ injektiv auf S ist.

Aufgabe 89
In dieser Aufgabe wollen wir ein statisches Wörterbuch mittels Hashing realisieren. Ein statisches Wörterbuch unterstützt die Operation `lookup` auf einer beliebigen, aber festen Schlüsselmenge $S \subseteq U$ aus einem Universum U. Dabei sollen `lookup`'s in worst-case Zeit $O(1)$ unterstützt werden.

Sei $|S| = n$. H_m sei eine Klasse c-universeller Hashfunktionen mit Tabellengröße m. Wir verwenden das folgende zweistufige Schema, um die Schlüssel aus S zu hashen. Wir bilden S mit einem zufälligen $h \in H_n$ ab. Sei $B_i(h) = h^{-1}(i) \cap S$ die Menge der von h auf i abgebildeten Schlüssel. In der zweiten Stufe bildet für alle i jeweils ein zufälliges $h_i \in H_{m_i}$ die Schlüssel aus $B_i(h)$ ab. Wie groß muß m_i sein, damit h_i mit Wahrscheinlichkeit $1/2$ injektiv auf $B_i(h)$ ist?

Beschreibe einen randomisierten Algorithmus, der die Menge S in einem Array der Größe $O(n)$ speichert, so daß `lookup` Operationen in worst-case Zeit $O(1)$ bearbeitet werden können.

Hinweis: Wende die vorherige Aufgabe an.

Aufgabe 90
Wir wollen Hashing verwenden, um *Pattern Matching* auszuführen. In einem Text T soll eine bestimmte Zeichenkette p gesucht werden, wobei der Text in einem Array der Länge n gespeichert sei und $|p| = m$ gelte. Es soll das erste Vorkommen von p in T bestimmt werden, kommt p nicht vor, wird $n + 1$ ausgegeben.

Eine einfache Methode ist folgende. Suche $p(0)$ im Text. Wenn $p(0)$ an Position i gefunden ist, so vergleiche die folgenden Zeichen im Text mit dem Rest des Patterns. Ist das Pattern gefunden, so ist man fertig; ansonsten sucht man ab $i + 1$ weiter nach $p(0)$.

(a) Bestimme die worst case Laufzeit dieser Methode.

Wir gehen anders vor und kodieren jedes Zeichen als eine Zahl. Unser Alphabet habe 32 Zeichen, welche durch $0, \ldots, 31$ kodiert seien. Einer Zeichenkette z der Länge m wird der Wert

$$W(z) = z[0] \cdot 32^{m-1} + z[1] \cdot 32^{m-2} + \cdots + z[m-1] \cdot 32 + z[m],$$

zugewiesen und gehasht. Das Pattern Matching kann nun so vor sich gehen, daß wir in T die erste Zeichenkette $t_i = T[i] \ldots T[i+m-1]$ suchen, für die $h(W(t_i)) = h(W(p))$ gilt. Nun müssen wir noch prüfen, ob $t_i = p$ gilt, da die Hashfunktion nicht notwendigerweise injektiv ist.

(b) Betrachte die Klasse H_t der Hashfunktionen $h_q(x) = x \bmod q$ für Primzahlen $q \leq t$. **Gib** an, wie groß t gewählt sein muß, damit eine zufällig gewählte Hashfunktion $h_q \in H_t$ mit Wahrscheinlichkeit $1/2$ injektiv auf $\{W(t_1), \ldots, W(t_{n-m+1}), W(p)\}$ ist. Beachte: es gibt mindestens $\sqrt{t}$ Primzahlen $q \leq t$. **Zeige dann:** Auf Schlüsseln der Größe 2^s ist H_t $s\sqrt{t}$-universell.

(c) Beschreibe eine möglichst effiziente Implementierung des Pattern Matching Algorithmus' und **bestimme** seine Laufzeit unter der Annahme, daß h_q injektiv ist und daß $h_q(p)$ mit $h_q(t_i)$ in konstanter Zeit verglichen werden kann.

Bemerkung: $W(t_i)$ wird nicht direkt mit $W(p)$ verglichen, da ein Test auf Gleichheit bei Integer Zahlen der Größe 32^m im allgemeinen nicht effizient (d.h. in konstanter Zeit) möglich ist. Es sollen daher in der Implementierung immer nur möglichst kleine Zahlen bearbeitet werden.

5.5.5 Bloom-Filter

Auch Bloom-Filter implementieren ein Wörterbuch. Diesmal gehen wir aber davon aus, dass die `lookup`(x)-Operation extrem eingeschränkt ist: Statt die mit dem Schlüssel x verbundene Information auszugeben, beschränken wir uns jetzt ausschließlich auf die Entscheidung, ob x vorhanden ist. Deshalb werden Bloom-Filter vor Allem in Situation mit außergewöhnlich hohem Datenaufkommen eingesetzt, um Speicherplatz zu sparen.

Ein Bloom-Filter repräsentiert eine Menge X durch ein boolesches Array B der Länge m und benutzt dazu k „Hashfunktionen“ $h_1, \ldots, h_k$:

> Anfänglich ist $X = \emptyset$ und alle Zellen von B sind auf Null gesetzt. Ein Element $x \in U$ wird in die Menge X eingefügt, indem das Array B nacheinander an den Stellen $h_1(x), \ldots, h_k(x)$ auf Eins gesetzt wird.
>
> Um nachzuprüfen, ob x ein Element von X ist, wird B an den Stellen $h_1(x), \ldots, h_k(x)$ überprüft. Wenn B an allen Stellen den Wert 1 besitzt, dann wird die Vermutung „$x \in X$“ ausgegeben und ansonsten wird die definitive Ausgabe „$x \notin X$“ getroffen.

Offensichtlich erhalten wir konventionelles Hashing aus Bloom-Filtern für $k = 1$. Wir beachten, dass eine negative Antwort auf eine $x \in X$? Anfrage stets richtig ist. Eine positive Antwort kann allerdings falsch sein und Bloom-Filter produzieren damit „falsche Positive“.

Wie groß ist die Wahrscheinlichkeit $q_{n,m,k}$ einer falschen positiven Antwort, wenn eine Menge X der Größe **n** durch ein boolesches Array der Größe **m** mit Hilfe von **k** zufällig gewählten Hashfunktionen repräsentiert wird? Bevor wir diese Frage beantworten, bestimmen wir die Wahrscheinlichkeit $p_{n,m,k}$, dass B an einer fixierten Position i eine Null speichert. Es ist

$$p_{n,m,k} = (\frac{m-1}{m})^{kn} = (1 - \frac{1}{m})^{kn},$$

denn Position i darf in keinem der $k \cdot n$ Versuche getroffen werden. Desweiteren kann gezeigt werden, dass $p_{n,m,k} \approx e^{-(kn/m)}$ gilt, solange m^2 wesentlich größer als kn ist.

Die Wahrscheinlichkeit $q_{n,m,k}$ einer falschen positiven Antwort ist beschränkt durch die Wahrscheinlichkeit, in k Versuchen jeweils eine Eins „zu ziehen“. Wenn wir die Anzahl der Einsen kennen, dann ist das k-malige Ziehen von Einsen äquivalent zu k unabhängigen Experimenten, in denen jeweils geprüft wird, ob eine Eins gezogen wurde. Die Anzahl der Einsen ist aber nicht vorher bekannt und Unabhängigkeit liegt nicht vor wie die folgende Aufgabe zeigt.

Aufgabe 91
Wir vergleichen die tatsächliche Wahrscheinlichkeit $q_{n,m,k}$ einer falschen positiven Antwort mit der Abschätzung

$$Q := \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k.$$

Gib ein Beispiel mit $|U| = 2$, $n = 1$ und $k = m = 2$ **an**, für das $Q < q_{n,m,k}$ gilt. Die tatsächliche Wahrscheinlichkeit einer falschen positiven Antwort kann also größer als die Abschätzung Q sein!

Es stellt sich aber heraus, dass Q eine sehr scharfe Approximation von $q_{n,m,k}$ ist. Wir verwenden diese Beziehung ohne weitere Argumentation und erhalten:

$$q_{n,m,k} \approx \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{-kn/m})^k = e^{k \cdot \ln(1 - e^{-kn/m})}.$$

Wieviele Hash-Funktionen sollten wir bei gegebener Anzahl n der gespeicherten Elemente und bei gegebener Größe m des booleschen Arrays benutzen, um die Wahrscheinlichkeit einer falschen positiven Antwort zu minimieren?

> Einerseits „sollte“ k möglichst groß sein, da dann die Wahrscheinlichkeit wächst eine Null zu erwischen, aber andererseits „sollte“ k möglichst klein sein, da dann eine Einfüge-Operation nur wenige neue Einsen einträgt.

Das optimale k wird die Wahrscheinlichkeit $g(k) = q_{n,m,k}$ minimieren und damit auch die Funktion $\ln g(k)$ minimieren. Wir beachten $\ln g(k) = -\frac{m}{n} \cdot \ln(p) \cdot \ln(1-p)$ für $p = e^{-kn/m}$ und erhalten ein globales Minimum für $p = \frac{1}{2}$. Die Setzung $e^{-kn/m} = \frac{1}{2}$ führt dann auf

$$k = \ln(2) \cdot m/n$$

mit der Fehlerwahrscheinlichkeit $q_{n,m,k} \approx (1-p)^k = 2^{-k}$.

Lemma 5.13 *Wenn wir $k = \ln(2) \cdot m/n$ Hashfunktionen verwenden, um eine Menge der Größe n durch ein boolesches Array der Größe m zu repräsentieren, dann ist die Wahrscheinlichkeit einer falschen positiven Antwort höchstens 2^{-k}.*

Die Laufzeit einer `insert`*- oder* `lookup`*-Operation ist durch $O(k)$ und die Speicherplatzkomplexität ist durch $O(n)$ beschränkt.*

Führen wir einen Vergleich mit konventionellem Hashing ($k = 1$) durch: Bei einer Hashtabelle der Größe m ist $\frac{n}{m}$ die Wahrscheinlichkeit einer falschen positiven Antwort und eine Fehlerwahrscheinlichkeit von höchstens ε führt auf die Ungleichung $\frac{n}{m} \leq \varepsilon$ und damit auf die Forderung $m \geq \frac{1}{\varepsilon} \cdot n$. Bloom-Filter erreichen eine Fehlerwahrscheinlichkeit von höchstens ε für $\ln(2) \cdot m/n = k = \log_2(1/\varepsilon)$ und damit für $m = \log_2(1/\varepsilon) \cdot \frac{n}{\ln(2)}$.

Insbesondere erreicht Hashing die Fehlerwahrscheinlichkeit $\varepsilon = \frac{1}{n}$ mit n^2 Hashzellen, während Bloom-Filter nur $m = n \cdot \log_2(n)/\ln(2)$ Zellen benötigen und dabei $k = \log_2 n$ Hashfunktionen einsetzen. Allgemein genügen $k = c \cdot \log_2 n = \log_2(n^c)$ Hashfunktionen mit Hashtabellen der Größe $\log_2(n^c) \cdot \frac{n}{\ln(2)}$ für eine Fehlerwahrscheinlichkeit von höchstens $\frac{1}{n^c}$ und $m = O(n \log_2 n)$ ist für praktische Anwendungen ausreichend. In Anwendungen wird man deshalb

$$\Omega(n) = m = O(n \cdot \log_2 n) \text{ und entsprechend } \Omega(1) = k = O(\log_2 n)$$

wählen. Natürlich sollte der Speicherplatz des Bloom-Filters nicht größer sein als der für die Spezifizierung der Menge hinreichende Speicherplatz $O(n \cdot \log_2 |U|)$.

Wir haben bisher die `insert`- und `lookup`-Operationen für Bloom-Filter implementiert, aber die Implementierung einer `remove`-Operation ist schwieriger: Wenn wir das Array B an den Positionen $h_1(x), \ldots, h_k(x)$ auf Null setzen und wenn $\{h_1(x), \ldots, h_k(x)\} \cap \{h_1(y), \ldots, h_k(y)\} \neq \emptyset$, dann wird auch y entfernt! Hier hilft ein *zählender Bloom-Filter*: Statt einem booleschen Array wird ein Array von Zählern benutzt und Elemente werden durch das Hochsetzen (bzw. Heruntersetzen) der entsprechenden Zähler eingefügt (bzw. entfernt).

Aufgabe 92

Eine n-elementige Menge $S \subseteq U$ sei durch einen zählenden Bloom-Filter Z mit m Zählern und k Hashfunktionen $h_1, \ldots, h_k : U \to \{1, \ldots, m\}$ repräsentiert. Wir nehmen an, dass das Heruntersetzen der entsprechenden Zähler bei der Entfernung eines Elements x nur dann geschieht, wenn $Z[h_i(x)] \neq 0$ für **alle** $i = 1, \ldots, k$ gilt. Sei p_+ (bzw. p_-) die Wahrscheinlichkeit einer falschen *positiven* (bzw. *negativen*) Antwort.

(a) Verkleinern oder vergrößern sich die Wahrscheinlichkeiten p_+ und p_-, wenn wir ein Element $x \in S$ entfernen?

(b) Angenommen, N Elemente $x \in U$ werden aus dem Filter entfernt, wobei wir jetzt auch zulassen, falsche Positive zu entfernen. Sei p'_+ die Wahrscheinlichkeit einer falschen positiven Antwort und p'_- die Wahrscheinlichkeit einer falschen negativen Antwort in diesem neuen Modell. Zeige, dass dann $p'_+ \leq p_+$, $p'_- \leq Np_+$ gilt.

Bemerkung 5.3 *Angenommen wir arbeiten nur mit Schlüsseln aus einem Universum U. Tatsächlich können wir Bloom-Filter bereits mit der zufälligen Wahl von zwei Hashfunktionen $g_1, g_2 : U \to \{0, \ldots, p-1\}$ (für eine Primzahl p) implementieren, wenn wir eine geringfügig anwachsende Fehlerwahrscheinlichkeit tolerieren. Dazu setzen wir $m = k \cdot p$ und definieren die k Hashfunktionen $h_1, \ldots, h_k$ durch*

$$h_i(x) = (i-1) \cdot p + (g_1(x) + (i-1) \cdot g_2(x) \bmod p) .$$

Wir beachten, dass $h_i(x) = h_i(y)$ für alle i genau dann eintritt, wenn $g_1(x) = h_1(x) = h_1(y) = g_1(y)$ und $g_2(x) = g_2(y)$, und wir haben die Schwäche dieses Ansatzes herausgefunden: Wenn wir eine Menge X von n Elementen aufgebaut haben, dann kann die Fehlerwahrscheinlichkeit

$$\text{prob}[\exists x \in X : h_1(y) = h_1(x) \wedge \cdots \wedge h_k(y) = h_k(x)] \leq \frac{n}{p^2}$$

für $y \notin X$ nicht mit wachsendem k abgesenkt werden! Aber in praktischen Anwendungen wird man $p \approx n$ wählen und wir erhalten die relativ kleine Fehlerwahrscheinlichkeit von ungefähr $1/n$. y kann aber auch falsch positiv sein, wenn es zu jedem i ein $x_i \in X$ mit $h_i(y) = h_i(x_i)$ gibt und *die x_i sind unterschiedlich. In diesem Fall kann aber gezeigt werden, dass die Fehlerwahrscheinlichkeit wiederum durch $(1 - e^{-kn/m})^k$ abgeschätzt werden kann!*

Der Vorteil des neuen Schemas ist die schnellere Berechnung der k Hashfunktionen.

Aufgabe 93
Wir zerlegen die m Positionen von B in k Gruppen von jeweils m/k Positionen und würfeln dann Hashfunktionen $h_1, \ldots, h_k$ zufällig aus, wobei h_i für die $i$te Gruppe von Positionen zuständig ist.

Bestimme die Wahrscheinlichkeit p, dass B an einer zufällig ausgewählten Position eine Null speichert. Sinkt oder steigt die Wahrscheinlichkeit einer falschen positiven Antwort im Vergleich zum konventionellen Ansatz?

Wir erwähnen drei Anwendungen der Bloom-Filter:

5.5.5.1 Verteiltes Caching:

In Web Cache Sharing speichern mehrere Proxies Webseiten in ihren Caches und kooperieren, um zwischengespeicherte Webseiten abzurufen: Bei Nachfrage nach einer bestimmten Seite ermittelt der zuständige Proxy, ob ein Cache eines anderen Proxies die nachgefragte Seite enthält und bittet diesen Proxy um Lieferung. In diesem Szenario liegt die Benutzung von Bloom-Filtern nahe. Jeder Proxy speichert in einem Bloom-Filter ab, welche Seiten zwischengespeichert wurden und sendet in regelmäßigen Abständen seinen aktualisierten Bloom-Filter an seine Kollegen. Natürlich ist es möglich, dass ein Proxy aufgrund des Phänomens falscher Positiver vergebens kontaktiert wird und die Laufzeit erhöht sich in diesem Fall. Allerdings entstehen falsche Positive wie auch falsche Negative bereits durch die Auslagerung von Seiten während eines Aktualisierungszyklus und die Fehler des Bloom-Filters erscheinen durch die Verringerung des Netzwerk-Verkehrs mehr als amortisiert.

5.5.5.2 Aggressive Flüsse im Internet Routing

Die Ermittlung aggressiver Flüsse, also die Ermittlung von Start-Ziel Verbindungen, die Datenstaus durch den Transfer von besonders vielen Paketen verschärfen oder sogar verursachen, ist ein wichtiges Problem in der Internet Verkehrskontrolle.

Um aggressiven Flüssen auf die Schliche zu kommen, repräsentiert ein Router die Menge aller Start-Ziel Paare, deren Pakete er befördert, durch einen zählenden Bloom-Filter. Insbesondere werden für eine Start-Ziel Verbindung x die Zähler aller Positionen $h_1(x), \ldots, h_k(x)$ inkrementiert. Überschreitet der minimale der k Werte einen vorgegebenen Schwellenwert T, dann wird x als „möglicherweise aggressiv“ gebrandmarkt. Beachte, dass alle aggressiven Flüsse erkannt werden; allerdings ist es nicht ausgeschlossen, dass auch unschuldige Flüsse als aggressiv eingeschätzt werden.

Aufgabe 94
Wir nehmen an, dass eine Menge F von n Flüssen in einen Bloom-Filter mit m Zählern $Z[i]$, $i = 1, \ldots, m$ eingefügt wird. Angenommen, v_x Pakete eines Start-Ziel Paares $x \in F$ durchlaufen unseren Router.

(a) Was ist der Zusammenhang zwischen prob[$\min\{Z[h_1(x)], \ldots, Z[h_k(x)]\} > v_x$] und der Wahrscheinlichkeit p_+ einer falschen positiven Antwort?

(b) Tatsächlich können wir unseren Ansatz noch verbessern. Wie sollte eine Modifikation aussehen, die Überschätzungen der Vielfachheiten v_x für $x \in F$ erschwert? Insbesondere, wenn v'_x die Abschätzung unseres Verfahrens ist, und wenn v_{x1} die Abschätzung deines neuen Verfahrens ist, dann sollte stets $v_x \leq v_{x1} \leq v'_x$ gelten und die Ungleichung $v_{x1} < v'_x$ sollte in einigen Fällen auftreten können.

5.5.5.3 IP-Traceback:

Hier nehmen wir an, dass ein Router durch eine Paketflut angegriffen wird. Unser Ziel ist die Bestimmung aller Wege, über die das Opfer angegriffen wird. In unserer Lösung protokolliert ein Router jedes transportierte Paket in einem Bloom-Filter. Ein angreifendes Paket kann jetzt vom Opfer an alle Nachbarn weitergereicht werden. Der Nachbar, der das Paket befördert hat, wird dies erkennen, wobei aber falsche Positive dazu führen, dass Nachbarn fälschlicherweise annehmen, am Transport beteiligt gewesen zu sein. Wird dieser Rekonstruktionsprozess iteriert, dann sollten diese falschen Verzweigungen aber, bei entsprechend kleiner Wahrscheinlichkeit für falsche Positive, hochwahrscheinlich aussterben.

5.5.6 Verteiltes Hashing in Peer-to-Peer Netzwerken

Die Musiktauschbörse *Napster* erlaubt den Austausch von MP3-Musikdateien über das Internet und verfolgt dazu eine Mischung des Client-Server und des Peer-to-Peer Ansatzes: Die Napster-Software durchsucht den Kunden-Rechner nach MP3-Dateien und meldet die Ergebnisse an einen zentralen Server, der auch alleinigen Zugriff auf die Angebote und Suchanfragen der anderen Teilnehmer hat (Client-Server Ansatz). Der Server meldet dann als Ergebnis auf eine Anfrage die IP-Adressen der Rechner, die die gesuchte Musikdatei anbieten. Anbieter und Käufer können sich daraufhin direkt miteinander verbinden (Peer-to-Peer Ansatz).

Das *Gnutella-Netzwerk* ist ein vollständig dezentrales Netzwerk ohne zentrale Server. Startet ein Benutzer des Netzwerkes eine Suchanfrage, so wird diese zunächst nur an benachbarte Systeme weitergeleitet. Diese leiten dann ihrerseits die Anfrage an ihre benachbarten Systeme weiter, bis die angeforderte Datei gefunden wurde. Anschließend kann eine direkte Verbindung zwischen suchendem und anbietendem Benutzer für die Datenübertragung hergestellt werden. Gnutella überflutet also das Netz mit der Suchanfrage und wird deshalb nur unzureichend mit einer wachsenden Zahl von Peers skalieren.

Reine Peer-to-Peer Systeme sind ausfallsicherer und verteilen die hohe Belastung des Servers auf die Peers. Wir beschreiben das reine Peer-to-Peer Sytem *CHORD*, das sowohl den Zugriff auf Information wie auch die Registrierung neuer Peers oder das Abmelden alter Peers dezentral und effizient erlaubt.

5.5.6.1 Chord

Unser Ziel ist die Definition eines Protokolls, dass den effizienten Zugriff auf Information erlaubt, wenn Informationen auf verschiedenste Rechner verteilt ist und wenn Rechner das Netzwerk verlassen bzw. dem Netzwerk beitreten dürfen. Beachte, dass Information durch den Weggang, bzw. den Zugewinn von Rechnern bewegt werden muss und gute Protokolle sollte diese dynamischen Veränderungen ohne großen zusätzlichen Kommunikationsaufwand bewältigen.

Die zentrale algorithmische Idee besteht in dem Konzept konsistenter Hashfunktionen, die sowohl auf die Menge $\mathcal{R}$ aller möglichen IP-Adressen wie auch auf die Menge $\mathcal{D}$ aller möglichen Dateien-ID's angewandt werden dürfen. Im Unterschied zu den bisherigen Anwendungen liefern unsere Hashfunktionen jetzt reelle Zahlen in dem Intervall $[0, 1]$.

Definition 5.6 (a) Die Klasse konsistenter Hashfunktionen besteht aus allen Funktionen der Form $h : \mathcal{R} \cup \mathcal{D} \to [0, 1]$.

(b) Sei R eine Teilmenge von $\mathcal{R}$. Wenn $h(r_1) < h(r_2)$ für zwei Rechner $r_1, r_2 \in R$ gilt und wenn es keinen Rechner $r_3 \in R$ mit $h(r_1) < h(r_3) < h(r_2)$ gibt, dann heisst r_2 der Nachfolger von r_1 und r_1 der Vorgänger von r_2 bezüglich h.

Wir rechnen modulo 1, d.h. der Rechner mit größtem Hashwert ist Vorgänger des Rechners Rechners mit kleinstem Hashwert. Demgemäß sollte man sich das Intervall $[0, 1]$ als Kreis vorstellen.

(c) Sei R eine Teilmenge von $\mathcal{R}$ und D eine Teilmenge von $\mathcal{D}$. Dann sagen wir, dass der Rechner $r_2 \in R$ für die Dateienmenge $D' \subseteq D$ zuständig ist, falls $r_2 \in R$ der Nachfolger von r_1 ist und $D' = \{d \in D : h(r_1) < h(d) < h(r_2)$ gilt.

Angenommen die Dateienmenge D ist von den Rechnern in R abzuspeichern. Dann wird in Chord ein Rechner genau die Dateien speichern, für die er zuständig ist. Wenn jetzt ein Rechner $r \in R$ auf die Datei $d \in D$ zugreifen möchte, dann berechnet r den Hashwert $h(d)$. Chord stellt zusätzliche Information, die **Routing-Tabelle**, bereit, so dass r den für d zuständigen Rechner in wenigen Schritten lokalisieren kann:

(1) r erhält die IP-Adresse seines Vorgängers und seines Nachfolgers.

(2) Die Rechner sind durch die Hashfunktion auf dem Kreis angeordnet. Für jedes k erhält r desweiteren die IP-Adresse $R_r(k)$ des ersten Rechners mit einem Abstand von mindestens 2^{-k}, d.h. es ist $h(R_r(k)) \geq h(r) + 2^{-k}$ und für jeden Rechner $t \in R$ zwischen r und $R_r(k)$ ist $h(t) < h(r) + 2^{-k}$.

Auf der Suche nach Datei d bestimmt r den Wert k, so dass $h(R_r(k)) \leq h(d) \leq h(R_r(k-1))$ und sendet seine Anfrage nach d an den Rechner $s = R_r(k)$ mit der Bitte bei der Suche zu helfen; s ist also der Rechner der Routingtabelle von r mit größtem Hashwert beschränkt durch $h(d)$. Rechner s bestimmt seinerseits den Wert l mit $h(R_s(l)) \leq h(d) \leq h(R_s(l-1))$ und kontaktiert $R_s(l)$. Dieses Verfahren wird solange wiederholt bis der Vorgänger v des für

die Datei d zuständigen Rechners erreicht ist. Rechner v kontaktiert seinen Nachfolger, der d dann direkt an den ursprünglich anfragenden Rechner r verschickt.

Beachte, dass r und alle auf der Suche nach d beteiligten Rechner jeweils Rechner r' mit $h(r') \leq h(v)$ kontaktieren. Wir behaupten, dass die Hashdistanz zu $h(v)$ in jedem Suchschritt mindestens halbiert wird.

Lemma 5.14 *Wenn r' auf der Suche nach Datei d den Rechner r_1 kontaktiert, dann ist*

$$h(v) - h(r_1) \leq \frac{h(v) - h(r')}{2}.$$

Beweis: Es gelte $h(r') + 2^{-k} \leq h(v) < h(r') + 2^{-(k-1)}$. Dann wird r' den Rechner r_1 aus seiner Routingtabelle mit $h(r') + 2^{-k} \leq h(r_1) \leq h(v)$ kontaktieren. Also folgt

$$h(v) - h(r') = h(v) - h(r_1) + h(r_1) - h(r') \geq h(v) - h(r_1) + 2^{-k} \geq 2 \cdot (h(v) - h(r_1))$$

und das war zu zeigen. □

Wir erwähnen ohne Beweis, dass die Rechner bei zufälliger Wahl der Hashfunktion gleichmäßig mit Dateien belastet werden und dass die Rechner gleichmäßig auf dem Kreis verteilt sind.

Lemma 5.15 *Sei $|R| = n$ und $|D| = m$. Für hinreichend großes α gelten die folgenden Aussagen nach zufälliger Wahl einer konsistenten Hashfunktion mit „großer“ Wahrscheinlichkeit:*

(a) Jeder Rechner ist für höchstens $\alpha \cdot \frac{m}{n} \cdot \log_2(n+m)$ Dateien zuständig.

(b) Kein Interval der Länge $\frac{\alpha}{n^2}$ enthält zwei Rechner und kein Interval der Länge $\frac{\alpha}{m^2}$ enthält zwei Dateien.

(c) Die Suche nach der Datei s ist nach höchstens $\alpha \cdot \log_2 n$ Schritten erfolgreich.

Betrachten wir als Nächstes das Einfügen von Rechnern. Wenn Rechner r dem System beitreten möchte, dann bittet r einen beliebigen Rechner des Systems die Suche nach $h(r)$ durchzuführen. Nach erfolgreicher Suche hat r seinen Nachfolger r' gefunden und kann r' benachrichtigen. r' übermittelt die IP-Adresse des bisherigen Vorgängers und betrachtet r als seinen neuen Vorgänger. Danach baut r seine Routing-Tabelle mit Hilfe seines Nachfolgers r' auf: Wenn r auf die Anfrage nach dem fiktiven Hashwert $h(r) + 2^{-k}$ die IP-Adresse r_k erhält, dann ist $R_r(k) = r_k$.

Warum besteht kein Grund zur Panik, wenn ein Hinzufügen von Rechnern nicht sofort vollständig in den Routing-Tabellen vermerkt wird? Solange der Rechner $R_r(k)$ nicht entfernt wurde, hilft $R_r(k)$ weiterhin in der Halbierung der Distanz und das gilt selbst dann, wenn nähere Knoten im Abstand mindestens 2^{-k} zwischenzeitlich eingefügt wurden. Das erwartete Verhalten nach dem Einfügen von Rechnern ist beweisbar „gutmütig“:

Lemma 5.16 *Zu einem gegebenen Zeitpunkt bestehe das Netz aus n Rechnern und alle Zeiger-Informationen (also Vorgänger, Nachfolger und Distanz 2^{-k} Rechner) seien korrekt. Wenn dann n weitere Rechner hinzugefügt werden und selbst wenn nur Vorgänger und Nachfolger richtig aktualisiert werden, dann gelingt eine Suche hochwahrscheinlich in Zeit $O(\log_2 n)$.*

Aufgabe 95
Zeige Lemma 5.16. Benutze, dass hochwahrscheinlich höchstens $O(\log_2 n)$ neue Rechner zwischen zwei alten Rechnern liegen.

Wenn sich ein Rechner r abmeldet, dann ist jetzt der Nachfolger von r für die Schlüssel von r zuständig. Desweiteren muss sich r bei seinem Nachfolger und seinem Vorgänger abmelden. Auch diesmal versucht Chord nicht, die Routing-Tabellen vollständig zu aktualisieren und stattdessen überprüft jeder Rechner seine Routing-Tabelle in periodischen Abständen. Sollte sich während der Suche nach einer Datei herausstellen, dass Rechner $R_r(k)$ verschwunden ist, dann leidet die Suche unmittelbar: r muss sich auf die Suche machen und kontaktiert $R_r(k-1)$ mit dem imaginären Schlüsselwert $h(r) + 2^{-k}$; nach erhaltener Antwort kann die Dateisuche fortgesetzt werden.

Aufgabe 96
(a) Zeige, dass die folgenden pathologischen Fälle nach Einfügen und Entfernen von Rechnern auftreten können:

- Die Nachfolger-Zeiger definieren eine Liste, die mehrfach über das Intervall $[0, 1]$ „streicht“.
- Die Nachfolger-Zeiger definieren mehrere disjunkte Zyklen.

(b) Wie lassen sich diese Pathologien erkennen?

Ein zentrales Problem für Chord ist die Behandlung bösartiger Rechner, ein Problem das zum jetzigen Zeitpunkt noch ungelöst ist. Ein ähnliches Problem ist der Ausfall von Rechnern ohne vorige Abmeldung; dieses Problem ist aber zumindest approximativ durch Datenreplikation über eine fixe Anzahl von Nachfolgern in den Griff zu bekommen. Weitere Distributed Hashing Ansätze werden in den Systemen CAN, Pastry und Tapestry verfolgt.

5.6 Datenstrukturen für das Information Retrieval

Die Aufgabe des Information Retrievals besteht in der Konstruktion von Informationssystemen, die die inhaltliche Suche nach Dokumenten unterstützen. Die entwickelten Informationssysteme müssen die „relevantesten“ Dokumente bei meist nur „vagen“ Anfragen ermitteln. Beispiele solcher Informationssysteme sind:

- Suchmaschinen für das Internet.
- Informationssysteme aller Art.
- Nachrichtenarchive überregionaler Zeitungen wie etwa das Archiv der Frankfurter Allgemeinen Zeitung (`http://www.faz.net/`).
- Literatur- und Fachinformationsdatenbanken wie etwa das Fachinformationszentrum Karlsruhe (`http://www.fiz-karlsruhe.de`).

Wir gehen im Folgenden von dem einfachsten Fall aus, nämlich dass ein Stichwort w als Anfrage vorliegt. Das Informationssystem muss dann die relevantesten Dokumente bestimmen, in denen die Anfrage w vorkommt.

Wir konzentrieren uns in diesem Abschnitt auf die Datenstrukturen in Informationssystemen und nehmen an, dass bereits eine Relevanzgewichtung vorgegeben ist. Im nächsten Abschnitt werden wir eine erfolgreiche Relevanzgewichtung im Detail kennenlernen

Ein Informationssystem hat im Wesentlichen zwei Aufgaben zu lösen. Zuerst müssen Dokumente *indiziert*, also nach den im Dokument auftreten Stichworten durchsucht werden und sodann sind die anfallenden Anfragen zu beantworten. Für die Bestimmung der Stichwörter ist ein Index bereitzustellen. Der Index sollte nur "inhalt-tragende Wörter" enthalten, wobei Kürzel wie ARD, BVB, DM durchaus auftreten dürfen; Artikel hingegen wie auch Stopworte (z.B. „und, oder, aber") treten im Allgemeinen nicht auf. Die zu entwickelnde Datenstruktur muss dann die folgenden Operationen effizient unterstützen:

- Das Entfernen und Einfügen von Dokumenten.
- Das Einfügen neuer Stichwörter wie auch das Suchen nach Stichwörtern.
- Eine nach Relevanz sortierte Auflistung der Dokumente, die ein vorgegebenes Stichwort enthalten.

Typischerweise müssen Informationssysteme gigantische Datenmengen verwalten. Beispielsweise speichert die Suchmaschine Google mehrere Milliarden Webseiten und beantwortet Millionen von Anfragen täglich. Während die Speicherkapazität moderner Platten wie auch die Prozessorgeschwindigkeit stetig zunimmt, so hat sich andererseits die Zugriffszeit für Platten mit 10 Millisekunden nur unwesentlich verändert. Deshalb setzt man Datenstrukturen ein, die die Anzahl der Plattenzugriffe minimieren.

Wir beschreiben eine Datenstruktur, die Speicherplatz verschwenderisch nutzt, um die Beantwortungszeit der Anfragen möglichst klein zu halten; hierzu sind unter Anderem, wie oben erwähnt, die Anzahl der Plattenzugriffe zu minimieren. Die Datenstruktur setzt sich aus den folgenden Komponenten zusammen:

(1) Die Sekundärstruktur besteht aus der *invertierten Liste*. Die invertierte Liste unterstützt die effiziente Beantwortung der Anfragen und besteht ihrerseits aus dem *Stichwortindex* und den *Fundstellendateien*.

Für jedes Stichwort des Indexes wird eine eigene Fundstellendatei angelegt, die sämtliche Dokumente aufführt, die das Stichwort (oder ähnliche Stichworte) enthalten. Eine Fundstellendatei ist im Allgemeinen eine nach Relevanz sortierte Liste von Zeigern auf die jeweiligen Dokumente des Dokumentenindexes

Der Stichwortindex wird in zwei Varianten implementiert. In der ersten Variante sind die Stichworte alphabetisch zu ordnen und ein B-Baum dient zur Implementierung. In der zweiten Variante wird Hashing eingesetzt, wobei vorausgesetzt wird, dass der Index im Hauptspeicher gespeichert werden kann. Diese zweite Variante wird zum Beispiel von Google benutzt.

(2) Die Primärstruktur besteht aus dem *Dokumentenindex* und der *Dokumentendatei*. Der Dokumentenindex enthält sämtliche gespeicherten Dokumente, die jeweils über eindeutige Dokumentennummern repräsentiert sind. Jedem Dokument ist eine Dokumentendatei zugeordnet, die auf sein Auftreten in den Fundstellendateien seiner Stichworte zeigt.

Der Dokumentenindex wird als B-Baum implementiert.

Für eine Anfrage w werden die relevantesten Dokumente der Fundstellendatei von w angezeigt. Wird ein Dokument mit Dokumentennummer d gelöscht, so wird das Dokument d zuerst mit Hilfe des Dokumentenindexes lokalisiert. Sodann wird d aus dem Dokumentenindex entfernt und sämtliche Verweise auf d werden mit Hilfe seiner Dokumentendatei aus den jeweiligen Fundstellendateien gelöscht.

Wird Dokument d hinzugefügt, so werden, falls präsent, neue Worte aus d dem Stichwortindex hinzugefügt. (Für die Aufnahme eines neuen Wortes sind sorgfältige Kriterien zu erarbeiten!) Sodann wird d in den Dokumentenindex eingefügt und seine Dokumentendatei wird erstellt. Gleichzeitig werden die Fundstellendateien der invertierten Liste um einen Verweis auf d erweitert, falls es sich um Fundstellen für Stichworte von d handelt. Um ein schnelles Einfügen in Fundstellendateien zu gewährleisten, bietet sich auch für Fundstellendateien eine Organisation über B-Bäume an. Für die Erweiterung einer Fundstellendatei zum Stichwort w ist die Relevanz von d zu ermitteln.

Beispiel 5.1 **(Die Suchmaschine Google)**

Wir haben bereits erwähnt, dass sich das World-Wide-Web in natürlicherweise Weise als ein gerichteter Graph auffassen lässt. Wir benutzen die oben eingeführten Begriffe und zeigen die entsprechenden Google-Begriffe zeigen in Klammern.

Google speichert nicht nur die Stichwörter einer Webseite ab, sondern hält jede besuchte Webseite vollständig in einem *Repository* bereit. Die Grundstruktur eines Dokumentenindexes und der Dokumentendateien wird beibehalten.

Google benutzt ein Stichwortindex (Lexikon) von mehreren Millionen Stichwörtern und verwaltet den Index über Hashing im Hauptspeicher. Die Fundstellendatei (Hit List) eines vorgegebenen Stichworts w hält für jede Webseite d, die w als Stichwort enthält, Zusatzinformation bereit wie die Häufigkeit des Vorkommens von w in d, Fontgröße und Schrifttyp, Auftreten in Überschriften sowie das Auftreten von w in Beschriftungen von Zeigern **auf** die Webseite d (Anchor Text). Sollte zum Beispiel w in vielen Beschriftungen von Zeigern auf d vorkommen, so dient dies als eine externe Bekräftigung der Relevanz von d für Stichwort w, da auch andere Webseiten d als relevant für w erachten. Natürlich ist hier Missbrauch möglich, da referenzierende Webseiten vom Autor von d erstellt werden können.

Google führt den *Page-Rank* p_d als ein Maß für die globale Relevanz der Webseite d ein, um diesen Missbrauch zu bekämpfen. Wie sollte p_d definiert werden? Die Webseite d sollte einen hohen Page-Rank p_d besitzen, wenn genügend viele andere Webseiten c von hohem Page-Rank Verweise auf d besitzen. Sei g_c die Anzahl der Webseiten, auf die Webseite c verweist. Wenn die Webseite c auf die Seite d verweist, dann könnte man intuitiv sagen, dass die Seite d den Bruchteil $\frac{p_c}{g_c}$ von der Webseite c erbt. Dementsprechend bietet sich die Definition

$$p_d = \sum_{c \text{ zeigt auf } d} \frac{p_c}{g_c} \tag{5.3}$$

an. Wie können wir den Page-Rank p_d in (5.3) bestimmen? Wir definieren die Wahrscheinlichkeit $p_{c,d}$ durch

$$p_{c,d} = \begin{cases} \frac{1}{g_e} & c \text{ zeigt auf } d, \\ 0 & \text{sonst} \end{cases}$$

und lassen einen „random Surfer“ zufällig durch das Web springen, wobei der random Surfer mit Wahrscheinlichkeit $p_{c,d}$ von der Webseite c auf die Webseite d springen darf. Unser Ziel ist die Definition von p_d als die relative Häufigkeit mit der der random Surfer auf die Seite d trifft.

Sollten wir mit diesem Ansatz zufrieden sein? Sicherlich nicht, denn wenn unser random Surfer auf eine Webseite ohne ausgehende Links trifft, ist er gefangen! Wir müssen unseren Ansatz also reparieren. Wir erlauben dem random Surfer mit Wahrscheinlichkeit $1 - d$ auf eine der N Webseiten zufällig zu springen, während ein Link der gegenwärtig benutzten Webseite mit

Wahrscheinlichkeit d benutzt werden muss. Jetzt bietet sich die Definition

$$p_t = (1-d) + d \cdot \sum_{s \text{ zeigt auf } t} \frac{p_s}{g_e} \tag{5.4}$$

an, die auch tatsächlich von Google benutzt wird. Wie gelingt die (approximative) Berechnung von (5.4)? Mit dem mehrstündigen Einsatz von mehreren Tausenden PC's.

Wir können jetzt zusammenfassen. Google berechnet Page-Ranks in einer Vorabrechnung und aktualisiert Page-Ranks bei laufendem Betrieb. Für eine Anfrage w werden die Webseiten t der Fundstellendatei von w bestimmt. Die Relevanz von t für w berechnet sich unter anderem aus der Häufigkeit des Vorkommens von w in d, der Fontgrösse und des Schrifttyps der Vorkommen sowie dem Auftreten in Überschriften. Weiterhin fließt der Page-Rank von t ein wie auch der Page-Rank von Seiten, die auf t zeigen und für die t Teil der Zeigerbeschriftung ist. Google berechnet die Relevanz von t für w also im Wesentlichen aus drei Komponenten:

- der Betonung von w innerhalb der Webseite d,
- dem „lokalen" Page-Rank, also dem Gewicht der Webseiten, für die w Teil der Beschriftung des Zeigers auf t ist
- und dem globalen Page-Rank p_t.

Mehr über Google's Page-Rank, und insbesondere über den Zusammenhang zu Markoff-Ketten, erfahren Sie in der Veranstaltung „Diskrete Modellierung".

5.7 Zusammenfassung

Im Wörterbuchproblem müssen wir die Operationen `insert`, `remove` und `lookup` effizient unterstützen. Hashing ist den Baum-Datenstrukturen klar überlegen, andererseits sind die Baum-Datenstrukturen klare Gewinner, wenn auch ordnungsorientierte Operationen (wie: finde Minimum, finde Maximum, sortiere, drucke alle Schlüssel in einem vorgegebenen Intervall, bestimme den Rang eines Schlüssels) zu unterstützen sind.

Die fundamentalste Baum-Datenstruktur ist der binäre Suchbaum. Die erwartete Laufzeit ist logarithmisch, die worst-case Laufzeit hingegen linear in der Anzahl der gespeicherten Schlüssel. Da der binäre Suchbaum z. B. bei fast sortierten Schlüsseln, einem Fall von praktischer Relevanz, versagt, müssen auch andere Baum-Datenstrukturen betrachtet werden.

AVL-Bäume besitzen eine logarithmische worst-case Laufzeit. Dies wird erreicht, da ein AVL-Baum für n Schlüssel die Tiefe $2\log_2\ n$ nicht überschreitet: AVL-Bäume sind tiefenbalanciert. Die Tiefenbalance wird durch Rotationen während des Zurücklaufens des Suchpfades erreicht.

Splay-Bäume sind selbstorganisierende Datenstrukturen: Sie passen sich den Daten automatisch an. So werden häufig abgefragte Schlüssel stets an der Baumspitze zu finden sein. Die worst-case Laufzeit von Splay-Bäumen ist miserabel (nämlich linear), aber ihre amortisierte Laufzeit ist logarithmisch. Mit anderen Worten, n Operationen benötigen $O\ (n\ \log_2\ n)$ Schritte, obwohl wenige der n Operationen mehr als logarithmische Zeit benötigen können. Splay-Bäume sind zu empfehlen, wenn es relativ wenige hochwahrscheinliche Schlüssel gibt.

(a, b)-Bäume verallgemeinern 2-3 und 2-3-4 Bäume (für Wörterbücher, die im Hauptspeicher angelegt werden) sowie B-Bäume (für Wörterbücher, die auf einem Hintergrund-Speicher

abgelegt sind.) Höchstens $O(\log_a n)$ Knoten müssen für die Ausführung einer Wörterbuch-Operation besucht werden. Diese Eigenschaft insbesondere macht B-Bäume zu der bevorzugten Baum-Datenstruktur, wenn die Daten auf einem Hintergrund-Speicher abgelegt sind.

Wir unterscheiden die Hashing-Verfahren Hashing mit Verkettung und Hashing mit offener Adressierung. Die Laufzeit einer Wörterbuch Operation ist linear im Auslastungsfaktor λ für Hashing mit Verkettung, während Hashing mit offener Addressierung selbst unter idealen Bedingung $\frac{1}{1-\lambda}$ Operationen benötigt. Hashing mit offener Adressierung sollte deshalb nur bei kleinem Auslastungsfaktor (z. B. $\lambda \leq 1/2$) benutzt werden, bzw Cuckoo Hashing nur für $\lambda \leq \frac{1}{3}$. Cuckoo Hashing besitzt die bemerkenswerte Eigenschaft, dass höchstens zwei Zellen für eine lookup- oder remove-Operation getestet werden müssen.

Die erwartete Laufzeit des universellen Hashing ist für **jede** Folge von n Operationen durch $O(n \cdot (1 + \frac{c}{2}\frac{n}{m}))$ nach oben beschränkt. Das universelle Hashing, eine Variante des Hashing mit Verkettung, kann also durch keine Operationsfolge „in die Knie gezwungen werden“.

Bloom-Filter sind eine weitere Implementierung von Wörterbüchern, die besonders bei einem großen Datenaufkommen zum Einsatz kommen. Die beträchtliche Speicherplatzersparnis wird erkauft durch möglicherweise „falsche positive“ Antworten; desweiteren kann nur beantwortet werden, ob ein Schlüssel vorhanden ist oder nicht, weitere Informationen können nicht gegeben werden.

Schließlich haben wir verteiltes Hashing in Peer-to-Peer Systemen mit konsistenten Hash-Funktionen besprochen. Eine konsistene Hash-Funktion kann sowhl auf die IP-Addresse eines Rechner wie auch auf Dateien-ID's angewandt werden. Gleichmäßige Rechnerbelastung und schnelle Suchzeit wurde durch die zufällige Wahl der konsisten Hash-Funktion sichergestellt.

Kapitel 6

Klausuren

Prof. Dr. G. Schnitger
S.C. Ionescu

Frankfurt, den 29.07.2006

Modulabschlussprüfung
Datenstrukturen

SS 2006

⇓ BITTE GENAU LESEN ⇓

Die Klausur besteht aus 3 Aufgaben. Die Klausur ist mit Sicherheit bestanden, wenn (zusammen mit der Bonifikation aus den Übungspunkten) mindestens **50%** der Höchstpunktzahl erreicht wird.

Bitte schreiben Sie oben auf **jeder** Seite in Blockschrift Namen und Matrikelnummer. Überprüfen Sie, ob die Klausur aus insgesamt **14** durchnummerierten Seiten besteht.

Schreiben Sie **nicht** mit Bleistift. Zugelassene Hilfmittel: 1 Blatt DIN A4 mit Notizen.

Bitte beachten Sie, dass gemäß der Bachelor-Ordnung das Mitbringen nicht zugelassener Hilfsmittel eine Täuschung darstellt und zum Nichtbestehen der Klausur führt. Bitte deshalb Handys vor Beginn der Klausur abgeben.

Bitte benutzen Sie Rückseiten und die beigefügten Zusatzblätter. Weitere Blätter sind erhältlich.

Werden zu einer Aufgabe 2 Lösungen angegeben, so gilt die Aufgabe als nicht gelöst. Entscheiden Sie sich also immer für **eine** Lösung. Eine **umgangssprachliche**, aber **strukturierte** Beschreibung von Datenstrukturen oder Algorithmen ist völlig ausreichend. Begründungen sind nur dann notwendig, wenn die Aufgabenformulierung dies verlangt.

Die Klausur dauert 100 Minuten.

Der Termin für die Einsichtnahme in die Klausur ist am 03.08.2006 von 14:00 Uhr bis 16:00 Uhr im SR 307.

Name:	Matrikelnummer:

AUFGABE 1 **Short Answer und Laufzeit-Analyse, 42 Punkte**

(a, 6 Punkte) Bewerten Sie die folgende Aussagen. (Richtige Antworten erhalten 2 Punkte, falsche Antworten werden mit -2 Punkten bewertet, keine Antwort ergibt 0 Punkte. Die Gesamtpunktzahl beträgt aber mindestens 0 Punkte.)

(i) $\log_2(8^n) = 3n$ ◯ richtig ◯ falsch

(ii) Wenn $f(n) = O(g(n))$, dann gilt $2^{f(n)} = O(2^{g(n)})$ ◯ richtig ◯ falsch

(iii) Für alle festen $a, b > 1$ ist $\log_a n = \Theta(\log_b n)$. ◯ richtig ◯ falsch

(b, 10 Punkte) **Ordnen** Sie die Funktionen

$f_1(n) = 2^{\log_4 n}$ $f_2(n) = 4^{\log_2 n}$ $f_3(n) = n^{1/3}$ $f_4(n) = n^3$ $f_5(n) = n \cdot \log_2 n$

nach ihrem asymptotischen Wachstum, beginnend mit der „langsamsten“ Funktion.

(c, 6 Punkte) **Bestimmen** Sie die Laufzeit der folgenden While-Schleife in Abhängigkeit von n.

```
while (n >= 1)
   { n = n/3;
     Eine Menge von bis zu 5 Anweisungen.
     /* Die Anweisungen veraendern den Wert von n nicht.   */}
```

Die Laufzeit der While-Schleife in Abhängigkeit von n ist: $\Theta($ _____ $)$

(d, 8 Punkte) Ein rekursiver Algorithmus A arbeitet auf einem Array

$$X = (X[0], \ldots, X[n-1])$$

mit n Schlüsseln.

(1) *Zuerst* wird A nacheinander auf den Teilarrays $X_1 = (X[0], \ldots, X[n/2-1])$, $X_2 = (X[n/4], \ldots, X[3n/4-1])$ und $X_3 = (X[n/2], \ldots, X[n-1])$ rekursiv aufgerufen.

(2) *Danach* arbeitet A für $f(n)$ Schritte und terminiert.

Geben Sie zuerst eine Rekursionsgleichung für die Laufzeit $T(n)$ von Algorithmus A an, wobei wir annehmen, dass n eine Potenz von 2 ist. Es gelte $T(1) = 1$ und $T(2) = 2$.

$T(n) =$ _____

Bestimmen Sie eine Lösung der Rekursionsgleichung in Θ-Notation für $\mathbf{f(n) = n}$.

$T(n) = \Theta($_____$)$

Bestimmen Sie eine Lösung der Rekursionsgleichung in Θ-Notation für $\mathbf{f(n) = n^2}$.

$T(n) = \Theta($_____$)$

(e, 12 Punkte) Geben Sie kurze Antworten:

(i) Geben Sie die minimale Tiefe $T_{min}(n)$ und maximale Tiefe $T_{max}(n)$ eines binären Suchbaums mit n Schlüsseln an.

Antwort:

$T_{min}(n) = \Theta($_____$)$

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

(ii) Wir führen Tiefensuche in einem ungerichteten Graphen $G = (V, E)$ aus und erhalten den Wald W_G der Tiefensuche. Wir möchten feststellen, ob G zusammenhängend ist. Geben Sie mit Hilfe von W_G ein schnell zu überprüfendes Kriterium an.

Antwort: G is genau dann zusammenhängend, wenn

(iii) Für welches a ist $\log_a n = 3$?

Antwort: $a =$ _____

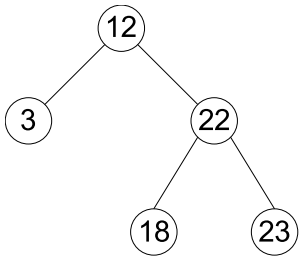
(iv) Wir betrachten die Postorder-Reihenfolge für einen Baum B. Angenommen, wir besuchen gerade einen Knoten v. Welcher Knoten wird als nächster Knoten besucht?

Antwort:

Name:	Matrikelnummer:

AUFGABE 2 **How to, 24 Punkte**

(a, 12 Punkte) Wir betrachten den folgenden AVL Baum B.

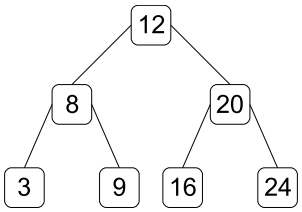


(i) Fügen Sie die Zahl 30 in den Baum B ein. Zeigen Sie alle erforderlichen Rotationen.

(ii) Fügen Sie die Zahl 20 in den **ursprünglichen** Baum B ein. Zeigen Sie alle erforderlichen Rotationen.

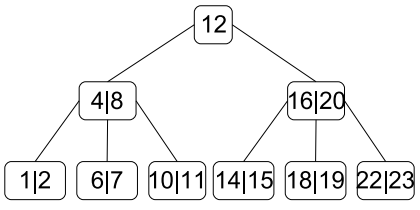
(b, 12 Punkte) (2,3)-Bäume

(i) Entfernen Sie den Schlüssel 16 aus dem folgendem (2,3)-Baum:



Verdeutlichen Sie die notwendigen Schritte.

(ii) Fügen Sie den Schlüssel 21 in den folgenden (2,3)-Baum ein:



Verdeutlichen Sie die notwendigen Schritte.

Name:	Matrikelnummer:

AUFGABE 3 **Anwendung von Datenstrukturen, 34 Punkte**

(a, 14 Punkte) Im Problem der Aufgabenverteilung sind n Aufgaben $1, 2, \ldots, n$ gegeben, wobei jede Aufgabe durch jeweils einen von k identischen Prozessoren $p_1, \ldots, p_k$ zu bearbeiten ist. Die Reihenfolge der Abarbeitung ist beliebig, da keine Abhängigkeiten zwischen den Aufgaben bestehen. Die Ausführungslänge von Aufgabe i sei t_i. Das **Ziel** der Aufgabenverteilung ist eine Prozessorzuweisung, so dass die Abarbeitung *aller* Aufgaben frühestmöglich beendet ist.

Wir suchen also eine Zerlegung von $\{1, \ldots, n\}$ in disjunkte Teilmengen $I_1, \ldots, I_k$ (mit $\bigcup_{j=1}^{k} I_j = \{1, \ldots, n\}$), wobei I_j die Menge der von Prozessor j zu bearbeitenden Aufgaben ist. Prozessor j benötigt Zeit $\mathbf{T_j} = \sum_{\mathbf{i \in I_j}} \mathbf{t_i}$ und die maximale Bearbeitungszeit $\max_{\mathbf{1 \le j \le k}} \mathbf{T_j}$ ist zu minimieren.

Eine optimale Prozessorzuweisung kann mit effizienten Algorithmen vermutlich nicht bestimmt werden. Deshalb betrachten wir die Heuristik $\mathcal{H}$, die

- die Aufgaben in der ursprünglichen Reihenfolge $1, 2, \ldots, n$ betrachtet und
- Aufgabe i dem Prozessor mit der bisher geringsten Bearbeitungszeit zuweist.

Welche Datenstrukturen sollten benutzt werden, damit die Heuristik $\mathcal{H}$ eine Aufgabenverteilung möglichst schnell bestimmt? Bestimmen Sie die Laufzeit ihrer Implementierung in Abhängigkeit von n und k und begründen Sie ihre Antwort.

(b, 20 Punkte) Im Problem der topologischen Sortierung ist ein gerichteter Graph $G = (\{1, \ldots, n\}, E)$ gegeben. Eine Reihenfolge $(v_1, \ldots, v_n)$ der Knoten heißt *legal*, wenn es keine Kante von einem („späten“) Knoten v_j zu einem („frühen“) Knoten v_i mit $i < j$ gibt. Eine legale Reihenfolge existiert immer, wenn der Graph G azyklisch ist, also keine Kreise besitzt.

Wir bestimmen in dieser Aufgabe eine topologische Sortierung mit Hilfe der Tiefensuche. Der Wald W_G der Tiefensuche sei bekannt, ebenso wie eine Klassifizierung aller Kanten in Baum-, Rückwärts-, Vorwärts- und Querkanten.

(i) Wann ist G azyklisch? Formulieren Sie ein einfaches Kriterium mit Hilfe der verschiedenen Kantentypen und begründen Sie ihre Antwort.

(ii) Wir nehmen an, dass G azyklisch ist. Begründen Sie warum, die Wurzel w des rechtesten Baums von W_G als erster Knoten in einer legalen Reihenfolge auftreten darf. (Der rechteste Baum von W_G ist der letzte Baum, der von Tiefensuche erzeugt wird.)

(iii) Wir nehmen wieder an, dass G azyklisch ist. Zeigen Sie, dass die Knotenreihenfolge gemäß absteigendem Ende-Wert legal ist.

Für einen Knoten u ist Ende$[u]$ der Zeitpunkt der Terminierung von tsuche(u).

Prof. Dr. G. Schnitger
S.C. Ionescu

Frankfurt, den 28.08.2006

Modulabschlussprüfung
Datenstrukturen

SS 2006

Name: _____ Vorname: _____

Matrikelnummer: _____

Geburtsdatum: _____

⇓ BITTE GENAU LESEN ⇓

Die Klausur besteht aus 3 Aufgaben. Die Klausur ist mit Sicherheit bestanden, wenn (zusammen mit der Bonifikation aus den Übungspunkten) mindestens **50%** der Höchstpunktzahl erreicht wird.

Bitte schreiben Sie oben auf **jeder** Seite in Blockschrift Namen und Matrikelnummer. Überprüfen Sie, ob die Klausur aus insgesamt **14** durchnummerierten Seiten besteht.

Schreiben Sie **nicht** mit Bleistift. Zugelassene Hilfmittel: 1 Blatt DIN A4 mit Notizen.

Bitte beachten Sie, dass gemäß der Bachelor-Ordnung das Mitbringen nicht zugelassener Hilfsmittel eine Täuschung darstellt und zum Nichtbestehen der Klausur führt. Bitte deshalb Handys vor Beginn der Klausur abgeben.

Bitte benutzen Sie Rückseiten und die beigefügten Zusatzblätter. Weitere Blätter sind erhältlich.

Werden zu einer Aufgabe 2 Lösungen angegeben, so gilt die Aufgabe als nicht gelöst. Entscheiden Sie sich also immer für **eine** Lösung. Eine **umgangssprachliche**, aber **strukturierte** Beschreibung von Datenstrukturen oder Algorithmen ist völlig ausreichend. Begründungen sind nur dann notwendig, wenn die Aufgabenformulierung dies verlangt.

Die Klausur dauert 100 Minuten.

Der Termin für die Einsichtnahme in die Klausur ist am 31.08.2006 von 14:00 Uhr bis 16:00 Uhr im Raum 313.

1a	1b	1c	1d	1e	2a	2b	2c	3a	3b	Σ
6	10	6	8	12	8	8	8	16	18	100

Note

Bonifikation	Σ

Name:	Matrikelnummer:

AUFGABE 1 **Short Answer und Laufzeit-Analyse, 42 Punkte**

(a, 6 Punkte) Bewerten Sie die folgende Aussagen. (Richtige Antworten erhalten 2 Punkte, falsche Antworten werden mit -2 Punkten bewertet, keine Antwort ergibt 0 Punkte. Die Gesamtpunktzahl beträgt aber mindestens 0 Punkte.)

(i) Wenn $f(n) = O(g(n))$, dann gilt $\log_2 f(n) = O(\log_2 g(n))$ ◯ richtig ◯ falsch

(ii) $9^{\log_3 n} = n^2$ ◯ richtig ◯ falsch

(iii) $\log_2(f(n) \cdot g(n))) = O(\max\{\log_2 f(n), \log_2 g(n)\})$, falls stets $f(n), g(n) \geq 1$ ◯ richtig ◯ falsch

(b, 10 Punkte) **Ordnen** Sie die Funktionen

$f_1(n) = 8^{\log_2 n}$ $f_2(n) = 2^{\log_8 n}$ $f_3(n) = n^{1/2}$ $f_4(n) = n^2$ $f_5(n) = n^3 \cdot \log_2 n$

(c, 6 Punkte) Bestimmen Sie die Laufzeit der folgenden While-Schleife in Abhängigkeit von n.

```
while (n >= 1)
   { n = n/5;
     Eine Menge von bis zu n Operatonen werden ausgef"uhrt, wobei diese den Wert
     von n nicht ver"andern. }
```

Die Laufzeit der While-Schleife in Abhängigkeit von n ist: $O\Big($_____$\Big)$

Name:	Matrikelnummer:

(d, 8 Punkte) Ein rekursiver Algorithmus A arbeitet auf einem Array $X = (X[0], \ldots, X[n-1])$ mit n Schlüsseln.

(1) *Zuerst* arbeitet A für $f(n)$ Schritte.

(2) *Danach* wird A nacheinander auf den Teilarrays $X_1 = (X[n/3], \ldots, X[2n/3 - 1])$ und $X_2 = (X[2n/3], \ldots, X[n-1])$ rekursiv aufgerufen.

Geben Sie zuerst eine Rekursionsgleichung für die Laufzeit $T(n)$ von Algorithmus A an, wobei wir annehmen, dass n eine Potenz von 3 ist. Es gelte $T(1) = 1$.

$T(n) =$ _____

Bestimmen Sie eine Lösung der Rekursionsgleichung in Θ-Notation für $f(n) = n$.

$T(n) = \Theta($_____$)$

Bestimmen Sie eine Lösung der Rekursionsgleichung in Θ-Notation für $f(n) = 1$.

$T(n) = \Theta($_____$)$

Name:	Matrikelnummer:

(e, 12 Punkte) Geben Sie kurze Antworten:

(i) Geben Sie eine kurze Beschreibung von Heapsort und beweisen Sie, dass n Schlüssel in Zeit $O(n \log n)$ sortiert werden.

Antwort:

(ii) Warum benutzt man die Adjazenz-Listen Darstellung und nicht die Adjazenzmatrix Darstellung für die Implementierung der Tiefensuche?

Antwort:

(iii) Für welches a ist $log_a 2^n = 4$?

Antwort:

(iv) Wir betrachten die Präorder-Reihenfolge für einen Baum B. Angenommen, wir besuchen gerade einen Knoten v. Welcher Knoten wird als nächster Knoten besucht?

Antwort:

Name:	Matrikelnummer:

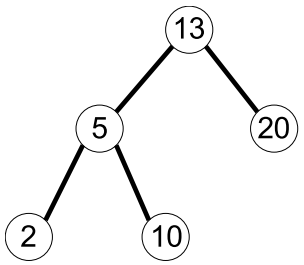
AUFGABE 2 **How to, 24 Punkte**

(a, 8 Punkte) Die Zahlen $[3, 10, 13, 17]$ sind in dieser Reihenfolge in eine Hashtabelle der Größe 7 einzufügen. Gegeben sind die Hashfunktionen $f(x) = x \bmod 7$ und $g(x) = 5 - (x \bmod 5)$. Die Zahlen sollen mit mit doppeltem Hashing, $h_i(x) = (f(x) + i \cdot g(x)) \bmod 7$ eingefügt werden:

Antwort:

0	1	2	3	4	5	6

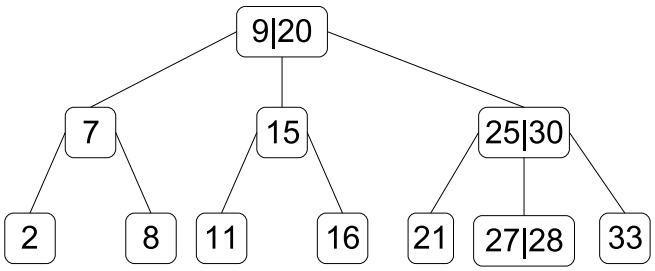
(b, 8 Punkte) Wir betrachten den folgenden AVL Baum B.



Fügen Sie die Zahl 11 in den Baum B ein. Zeigen Sie alle erforderlichen Rotationen.

Name:	Matrikelnummer:

(c, 8 Punkte) Fügen Sie den Schlüssel 26 in den folgenden (2,3)-Baum ein:



Verdeutlichen Sie die notwendigen Schritte.

Name:	Matrikelnummer:

AUFGABE 3 **Anwendung von Datenstrukturen, 34 Punkte**

(a, 16 Punkte) Die Union-Find Datenstruktur verwaltet einen Wald und unterstützt die Operationen

- $find(u)$: Bestimmt die Wurzel des Baums, der Knoten u enthält.
- $union(r, s)$: Die Wurzel r wird zum Kind der Wurzel s.

Beschreibe eine Datenstruktur mit den folgenden Eigenschaften:

- $find(u)$ soll in Zeit $O(T)$ laufen, wobei T die Tiefe des Baumes ist, der u enthält.
- $union(r, s)$ soll in Zeit $O(1)$ unterstützt werden.

Name:	Matrikelnummer:

(b, 18 Punkte) Die Mengendatenstruktur verwaltet Mengen $A_1, \cdots, A_m$ mit $A_i \subseteq \{1, \cdots, n\}$. Die folgenden Operationen sind schnellstmöglich zu unterstützen:

- $findmax()$: Bestimmt die größte Menge
- $remove(i, j)$: Ersetzt A_i durch die Mengendifferenz $A_i \backslash A_j = \{x \in A_i | x \notin A_j\}$

Beschreiben Sie eine Implementierung und analysieren Sie die Laufzeit von $findmax()$ in Abhängigkeit von m und n. Bestimmen Sie die Laufzeit von $remove(i, j)$ in Abhängigkeit von $|A_i|$ und $|A_j|$.