# Informatik II: Algorithmen und Daten-Strukturen

— Sommersemester 2012 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

17. Juli 2012

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

## 1.1 Motivation

Im ersten Semester haben wir gesehen, wie sich Probleme durch die Benutzung von Mengen und Relationen formulieren und lösen lassen. Eine Frage blieb dabei allerdings unbeantwortet: Mit welchen Datenstrukturen lassen sich Mengen und Relationen am besten darstellen und mit welchen Algorithmen lassen sich die Operationen, mit denen wir in der Mengenlehre gearbeitet haben, am effizientesten realisieren? Die Vorlesung *Algorithmen und Datenstrukturen* beantwortet diese Frage sowohl für die Datenstrukturen Mengen und Relationen als auch für einige andere Datenstrukturen, die in der Informatik eine wichtige Rolle spielen.

### 1.1.1 Überblick

Die Vorlesung *Algorithmen und Datenstrukturen* beschäftigt sich mit dem Design und der Analyse von Algorithmen und den diesen Algorithmen zugrunde liegenden Daten-Strukturen. Im Detail werden wir die folgenden Themen behandeln:

1. Unlösbarkeit des Halte-Problems

   Zu Beginn der Vorlesung zeigen wir die Grenzen der Berechenbarkeit auf und beweisen, dass es praktisch relevante Funktionen gibt, die sich nicht durch Programme berechnen lassen. Konkret werden wir zeigen, dass es kein SetlX-Programm gibt, dass für eine gegebene SetlX-Funktion $f$ und ein gegebenes Argument $s$ entscheidet, ob der Aufruf $f(s)$ terminiert.

2. Komplexität von Algorithmen

   Um die Komplexität von Algorithmen behandeln zu können, führen wir zwei Hilfsmittel aus der Mathematik ein.

   (a) *Rekurrenz-Gleichungen* sind die diskrete Varianten der Differential-Gleichungen. Diese Gleichungen treten bei der Analyse des Rechenzeit-Verbrauchs rekursiver Funktionen auf.

   (b) Die *$\mathcal{O}$-Notation* wird verwendet, um das Wachstumsverhalten von Funktionen kompakt beschreiben zu können. Sie bieten die Möglichkeit, bei der Beschreibung des des Rechenzeit-Verbrauchs eines Algorithmus von unwichtigen Details abstrahieren zu können.

3. Abstrakte Daten-Typen

   Beim Programmieren treten bestimmte Daten-Strukturen in ähnlicher Form immer wieder auf. Diesen Daten-Strukturen liegen sogenannte *abstrakte Daten-Typen* zugrunde. Als konkretes Beispiel stellen wir in diesem Kapitel den abstrakten Daten-Typ *Stack* vor.

Dieser Teil der Vorlesung überlappt sich mit der Vorlesung zur Sprache *Java*, denn abstrakte Datentypen sind eine der Grundlagen der Objekt-orientierten Programmierung.

4. Sortier-Algorithmen

   Sortier-Algorithmen sind die in der Praxis mit am häufigsten verwendeten Algorithmen. Da Sortier-Algorithmen zu den einfacheren Algorithmen gehören, bieten Sie sich als Einstieg in die Theorie der Algorithmen an. Wir behandeln im einzelnen die folgenden Sortier-Algorithmen:

   (a) Sortieren durch Einfügen (engl. *insertion sort*),

   (b) Sortieren durch Auswahl (engl. *min sort*),

   (c) Sortieren durch Mischen (engl. *merge sort*),

   (d) Den *Quick-Sort*-Algorithmus von C. A. R. Hoare.

5. Hoare-Kalkül

   Die wichtigste Eigenschaft eines Algorithmus' ist seine Korrektheit. Der *Hoare-Kalkül* ist ein Verfahren, mit dessen Hilfe die Frage der Korrektheit eines Algorithmus' auf die Frage der Gültigkeit logischer Formeln reduziert werden kann. An dieser Stelle werden wir eine Brücke zu der im ersten Semester vorgestellten Logik schlagen.

6. Abbildungen

   Abbildungen (in der Mathematik auch als Funktionen bezeichnet) spielen nicht nur in der Mathematik sondern auch in der Informatik eine wichtige Rolle. Wir behandeln die verschiedene Daten-Strukturen, mit denen sich Abbildungen realisieren lassen. Im einzelnen besprechen wir binäre Bäume, AVL-Bäume und Hash-Tabellen.

7. Prioritäts-Warteschlangen

   Die Daten-Struktur der Prioritäts-Warteschlangen spielt einerseits bei der Simulation von Systemen und bei Betriebssystemen eine wichtige Rolle, andererseits benötigen wir diese Datenstruktur bei der Diskussion graphentheoretischer Algorithmen.

8. Graphen

   Graphen spielen in vielen Bereichen der Informatik eine wichtige Rolle. Beispielsweise basieren die Navigationssysteme, die heute in fast allen Autos zu finden sind, auf dem Algorithmus von Dijkstra zur Bestimmung des kürzesten Weges. Wir werden diesen Algorithmus in der Vorlesung herleiten.

9. Monte-Carlo-Simulation

   Viele interessante Fragen aus der Wahrscheinlichkeits-Theorie lassen sich aufgrund ihrer Komplexität nicht analytisch lösen. Als Alternative bietet sich an, durch Simulation eine approximative Lösung zu gewinnen. Als konkretes Beispiel werden wir zeigen, wie komplexe Wahrscheinlichkeiten beim Poker-Spiel durch Monte-Carlo-Simulationen bestimmt werden können.

Ziel der Vorlesung ist nicht primär, dass Sie möglichst viele Algorithmen und Daten-Strukturen kennen lernen. Vermutlich wird es eher so sein, dass Sie viele der Algorithmen und Daten-Strukturen, die Sie in dieser Vorlesung kennen lernen werden, später nie gebrauchen können. Worum geht es dann in der Vorlesung? Das wesentliche Anliegen ist es, Sie mit den *Denkweisen* vertraut zu machen, die bei der Konstruktion und Analyse von Algorithmen verwendet werden. Sie sollen in die Lage versetzt werden, algorithmische Lösungen für komplexe Probleme selbstständig zu entwickeln und zu analysieren. Dabei handelt es sich um einen kreativen Prozeß, der sich nicht in einfachen Kochrezepten einfangen läßt. Wir werden in der Vorlesung versuchen, den Prozess an Hand verschiedener Beispiele zu demonstrieren.

## 1.2 Algorithmen und Programme

Gegenstand der Vorlesung ist die Analyse von Algorithmen, nicht die Erstellung von Programmen. Es ist wichtig, dass die beiden Begriffe "*Algorithmus*" und "*Programm*" nicht verwechselt werden. Ein *Algorithmus* ist seiner Natur nach zunächst einmal ein abstraktes Konzept, das ein Vorgehen beschreibt um ein gegebenes Problem zu lösen. Im Gegensatz dazu ist ein *Programm* eine konkrete Implementierungen eines Algorithmus. Bei einer solchen Implementierung muss letztlich jedes Detail festgelegt werden, sonst könnte das Programm nicht vom Rechner ausgeführt werden. Bei einem Algorithmus ist das nicht notwendig: Oft wollen wir nur einen Teil eines Vorgehens beschreiben, der Rest interessiert uns nicht, weil beispielsweise ohnehin klar ist, was zu tun ist. Ein Algorithmus läßt also eventuell noch Fragen offen.

In Lehrbüchern werden Algorithmen oft mit Hilfe von *Pseudo-Code* dargestellt. Syntaktische hat Pseudo-Code eine ähnliche Form wie ein Programm. Im Gegensatz zu Programmen kann Pseudo-Code aber auch natürlich-sprachlichen Text beinhalten. Sie sollten sich aber klar machen, dass *Pseudo-Code* genau so wenig ein Algorithmus ist, wie ein Programm ein Algorithmus ist, denn auch der *Pseudo-Code* ist ein konkretes Stück Text, wohingen der Algorithmus eine abstrakte Idee ist. Allerdings bietet der Pseudo-Code dem Informatiker die Möglichkeit, einen Algorithmus auf der Ebene zu beschreiben, die zur Beschreibung am zweckmäßigisten ist, denn man ist nicht durch die Zufälligkeiten der Syntax einer Programmier-Sprache eingeschränkt.

Konzeptuell ist der Unterschied zwischen einem Algorithmus und einem Programm vergleichbar mit dem Unterschied zwischen einer philosophischen Idee und einem Text, der die Idee beschreibt: Die Idee selbst lebt in den Köpfen der Menschen, die diese Idee verstanden haben. Diese Menschen können dann versuchen, die Idee konkret zu fassen und aufzuschreiben. Dies kann in verschiedenen Sprachen und mit verschiedenen Worten passieren, es bleibt die selbe Idee. Genauso kann ein Algorithmus in verschiedenen Programmier-Sprachen kodiert werden, es bleibt der selbe Algorithmus.

Nachdem wir uns den Unterschied zwischen einem Algorithmus und einem Programm diskutiert haben, überlegen wir uns, wie wir Algorithmen beschreiben können. Zunächst einmal können wir versuchen, Algorithmen durch natürliche Sprache zu beschreiben. Natürliche Sprache hat den Vorteil, dass Sie sehr ausdrucksstark ist: Was wir nicht mit natürlicher Sprache ausdrücken können, können wir überhaupt nicht ausdrücken. Der Nachteil der natürlichen Sprache besteht darin, dass die Bedeutung nicht immer eindeutig ist. Hier hat eine Programmier-Sprache den Vorteil, dass die Semantik wohldefiniert ist. Allerdings ist es oft sehr mühselig, einen Algorithmus vollständig auszukodieren, denn es müssen dann Details geklärt werden, die für das Prinzip vollkommen unwichtig sind. Es gibt noch eine dritte Möglichkeit, Algorithmen zu beschreiben und das ist die Sprache der Mathematik. Die wesentlichen Elemente dieser Sprache sind die Prädikaten-Logik und die Mengen-Lehre. In diesem Skript werden wir die Algorithmen in dieser Sprache beschreiben. Um diese Algorithmen dann auch ausprobieren zu können, müssen wir sie in eine Programmier-Sprache übersetzen. Hier bietet sich SetlX an, denn diese Programmier-Sprache stellt die Daten-Strukturen Mengen und Funktionen, die in der Mathematik allgegenwärtig sind, zur Verfügung. Sie werden sehen, dass es in SetlX möglich ist, die Algorithmen auf einem sehr hohen Abstraktions-Niveau darzustellen. Eine Implementierung der Algorithmen in `C++` oder *Java* ist erheblich aufwendiger.

## 1.3 Eigenschaften von Algorithmen und Programmen

Bevor wir uns an die Konstruktion von Algorithmen machen, sollten wir uns überlegen, durch welche Eigenschaften Algorithmen charakterisiert werden und welche dieser Eigenschaften erstrebenswert sind.

1. Algorithmen sollen *korrekt* sein.
2. Algorithmen sollen *effizient* sein.
3. Algorithmen sollen möglichst **einfach** sein.

Die erste dieser Forderung ist so offensichtlich, dass sie oft vergessen wird: Das schnellste Programm nutzt nichts, wenn es falsche Ergebnisse liefert. Nicht ganz so klar ist die letzte Forderung. Diese Forderung hat einen ökonomischen Hintergrund: Genauso wie die Rechenzeit eines Programms Geld kostet, so kostet auch die Zeit, die Programmierer brauchen um ein Programm zu erstellen und zu warten, Geld. Aber es gibt noch zwei weitere Gründe für die dritte Forderung:

1. Für einen Algorithmus, dessen konzeptionelle Komplexität hoch ist, ist die Korrektheit nicht mehr einsehbar und damit auch nicht gewährleistbar.

2. Selbst wenn der Algorithmus an sich korrekt ist, so kann doch die Korrektheit der Implementierung nicht mehr sichergestellt werden.

## 1.4 Literatur

Ergänzend zu diesem Skript möchte ich die folgende Literatur empfehlen.

1. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *Data Structures and Algorithms*, Addison-Wesley, 1987.

   Dieses Buch gehört zu den Standardwerken über Algorithmen. Die Algorithmen werden dort auf einem hohen Niveau erklärt.

2. *Frank M. Carrano* and *Janet J. Prichard*: *Data Abstraction and Problem Solving with Java*, Addison-Wesley, 2003.

   In diesem Buch sind die Darstellungen der Algorithmen sehr breit und verständlich. Viele Algorithmen sind graphisch illustriert. Leider geht das Buch oft nicht genug in die Tiefe, so wird zum Beispiel die Komplexität von Algorithmen kaum untersucht.

3. *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, and *Clifford Stein*: *Introduction to Algorithms*, MIT Press, 2001.

   Aufgrund seiner Ausführlichkeit eignet sich dieses Buch sehr gut zum Nachschlagen von Algorithmen. Die Darstellungen der Algorithmen sind eher etwas knapper gehalten, dafür wird aber auch die Komplexität analysiert.

4. *Robert Sedgewick*: *Algorithms in Java*, Pearson, 2002.

   Dieses Buch liegt in der Mitte zwischen den Büchern von Carrano und Cormen: Es ist theoretisch nicht so anspruchsvoll wie das von Cormen, enthält aber wesentlich mehr Algorithmen als das Buch von Carrano. Zusätzlich wird die Komplexität der Algorithmen diskutiert.

5. *Heinz-Peter Gumm und Manfred Sommer*, *Einführung in die Informatik*, Oldenbourg Verlag, 2006.

   Dieses Buch ist eine sehr gute Einführung in die Informatik, die auch ein umfangreiches Kapitel über Algorithmen und Datenstrukturen enthält. Die Darstellung der Algorithmen ist sehr gelungen.

# Kapitel 2

# Grenzen der Berechenbarkeit

In jeder Disziplin der Wissenschaft wird die Frage gestellt, welche Grenzen die verwendeten Methoden haben. Wir wollen daher in diesem Kapitel beispielhaft ein Problem untersuchen, bei dem die Informatik an ihre Grenzen stößt. Es handelt sich um das Halte-Problem.

## 2.1 Das Halte-Problem

Das Halte-Problem ist die Frage, ob eine gegebene Funktion für eine bestimmte Eingabe terminiert. Wir werden zeigen, dass dieses Problem nicht durch ein Programm gelöst werden kann. Dazu führen wir folgende Definition ein.

**Definition 1 (Test-Funktion)** *Ein String $t$ ist eine* Test-Funktion *mit Namen $n$ wenn $t$ die Form*

$$n \texttt{ := procedure(x) \{ } \cdots \texttt{ \}}$$

*hat, und sich als Definition einer* SetlX*-Funktion parsen läßt. Die Menge der Test-Funktionen bezeichnen wir mit $TF$. Ist $t \in TF$ und hat den Namen $n$, so schreiben wir* $\texttt{name}(t) = n$. □

**Beispiele**:

1. $s_1 =$ "`simple := procedure(x) { return 0; }`"
   $s_1$ ist eine (sehr einfache) Test-Funktion mit dem Namen `simple`.
2. $s_2 =$ "`loop := procedure(x) { while (true) { x := x + 1; } }`"
   $s_2$ ist eine Test-Funktion mit dem Namen `loop`.
3. $s_3 =$ "`hugo := procedure(x) { return ++x; } `"
   $s_3$ ist keine Test-Funktion, denn da SetlX den Präfix-Operator "`++`" nicht unterstützt, läßt sich der String $s_3$ nicht fehlerfrei parsen.

Um das Halte-Problem übersichtlicher formulieren zu können, führen wir noch drei zusätzliche Notationen ein.

**Notation 2 ($\rightsquigarrow$, $\downarrow$, $\uparrow$)** *Ist $n$ der Name einer C-Funktion und sind $a_1$, $\cdots$, $a_k$ Argumente, die vom Typ her der Deklaration von $n$ entsprechen, so schreiben wir*

$$n(a_1, \cdots, a_k) \rightsquigarrow r$$

*wenn der Aufruf $n(a_1, \cdots, a_k)$ das Ergebnis $r$ liefert. Sind wir an dem Ergebnis selbst nicht interessiert, sondern wollen nur angeben, daß ein Ergebnis existiert, so schreiben wir*

$$n(a_1, \cdots, a_k) \downarrow$$

*und sagen, dass der Aufruf* $n(a_1, \cdots, a_k)$ terminiert. *Terminiert der Aufruf* $n(a_1, \cdots, a_k)$ *nicht, so schreiben wir*

$$n(a_1, \cdots, a_k) \uparrow$$

*und sagen, dass der Aufruf* $n(a_1, \cdots, a_k)$ divergiert. □

**Beispiele**: Legen wir die Funktions-Definitionen zugrunde, die wir im Anschluß an die Definition des Begriffs der Test-Funktion gegeben haben, so gilt:

1. `simple("emil")` $\rightsquigarrow 0$
2. `simple("emil")` $\downarrow$
3. `loop("hugo")` $\uparrow$

Das *Halte-Problem* für SetlX-Funktionen ist die Frage, ob es eine `SetlX`-Funktion

`stops := procedure`$(t,\ a)\ \{\cdots\}$

gibt, die als Eingabe eine Testfunktion $t$ und einen String $a$ erhält und die folgende Eigenschaft hat:

1. $t \notin TF \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 2$.

   Der Aufruf `stops(`$t$`, `$a$`)` liefert genau dann den Wert 2 zurück, wenn $t$ keine Test-Funktion ist.

2. $t \in TF \wedge$ `name`$(t) = n \wedge n(a) \downarrow \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 1$.

   Der Aufruf `stops(`$t$`, `$a$`)` liefert genau dann den Wert 1 zurück, wenn $t$ eine Test-Funktion mit Namen $n$ ist und der Aufruf $n(a)$ terminiert.

3. $t \in TF \wedge$ `name`$(t) = n \wedge n(a) \uparrow \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 0$.

   Der Aufruf `stops(`$t$`, `$a$`)` liefert genau dann den Wert 0 zurück, wenn $t$ eine Test-Funktion mit Namen $n$ ist und der Aufruf $n(a)$ nicht terminiert.

Falls eine SetlX-Funktion `stops` mit den obigen Eigenschaften existiert, dann sagen wir, dass das Halte-Problem für SetlX entscheidbar ist.

**Theorem 3 (Turing, 1936)** *Das Halte-Problem ist unentscheidbar.*

**Beweis**: Zunächst eine Vorbemerkung. Um die Unentscheidbarkeit des Halte-Problems nachzuweisen müssen wir zeigen, dass etwas, nämlich eine Funktion mit gewissen Eigenschaften nicht existiert. Wie kann so ein Beweis überhaupt funktionieren? Wie können wir überhaupt zeigen, dass irgendetwas nicht existiert? Die einzige Möglichkeit zu zeigen, dass etwas nicht existiert ist indirekt: Wir nehmen also an, dass eine Funktion `stops` existiert, die das Halte-Problem löst. Aus dieser Annahme werden wir einen Widerspruch ableiten. Dieser Widerspruch zeigt uns dann, dass eine Funktion `stops` mit den gewünschten Eigenschaften nicht existieren kann. Um zu einem Widerspruch zu kommen, definieren wir den String *turing* wie in Abbildung 2.1 gezeigt.

Mit dieser Definition ist klar, dass *turing* eine Test-Funktion mit dem Namen "`alan`" ist:

$$\mathit{turing} \in TF \ \wedge\ \texttt{name}(\mathit{turing}) = \texttt{alan}.$$

Damit sind wir in der Lage, den String *Turing* als Eingabe der Funktion `stops` zu verwenden. Wir betrachten nun den folgenden Aufruf:

`stops(`*turing*`, `*turing*`);`

Offenbar ist *turing* eine Test-Funktion. Daher können nur zwei Fälle auftreten:

$$\texttt{stops}(\mathit{turing}, \mathit{turing}) \rightsquigarrow 0 \quad \vee \quad \texttt{stops}(\mathit{turing}, \mathit{turing}) \rightsquigarrow 1.$$

Diese beiden Fälle analysieren wir nun im Detail:

```
1  turing := "alan := procedure(x) {
2                  result := stops(x, x);
3                  if (result == 1) {
4                      while (true) {
5                          print("... looping ...");
6                      }
7                  }
8                  return result;
9              };"
```

Abbildung 2.1: Die Definition des Strings *turing*.

1. `stops`(*turing*, *turing*) $\rightsquigarrow 0$.

   Nach der Spezifikation von `stops` bedeutet dies

   `alan`(*turing*) $\uparrow$

   Schauen wir nun, was wirklich beim Aufruf `alan(turing)` passiert: In Zeile 2 erhält die Variable `result` den Wert 0 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Dieser Test schlägt fehl. Daher wird der Block der `if`-Anweisung nicht ausgeführt und die Funktion liefert als nächstes in Zeile 8 den Wert 0 zurück. Insbesondere terminiert der Aufruf also, im Widerspruch zu dem, was die Funktion `stops` behauptet hat.

   Damit ist der erste Fall ausgeschlossen.

2. `stops`(*turing*, *turing*) $\rightsquigarrow 1$.

   Aus der Spezifikation der Funktion `stops` folgt, dass der Aufruf `alan`(*turing*) terminiert:

   `alan`(*turing*) $\downarrow$

   Schauen wir nun, was wirklich beim Aufruf `alan(turing)` passiert: In Zeile 2 erhält die Variable `result` den Wert 1 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Diesmal gelingt der Test. Daher wird der Block der `if`-Anweisung ausgeführt. Dieser Block besteht aber nur aus einer Endlos-Schleife, aus der wir nie wieder zurück kommen. Das steht im Widerspruch zu dem, was die Funktion `stops` behauptet hat.

   Damit ist der zweite Fall ausgeschlossen.

Insgesamt haben wir also in jedem Fall einen Widerspruch erhalten. Also ist die Annahme, dass die SetlX-Funktion `stops` das Halte-Problem löst, falsch. Insgesamt haben wir gezeigt, dass es keine SetlX-Funktion geben kann, die das Halte-Problem löst. □

**Bemerkung**: Der Nachweis, dass das Halte-Problem unlösbar ist, wurde 1936 von Alan Turing (1912 – 1954) [Tur36] erbracht. Turing hat das Problem damals natürlich nicht für die Sprache SetlX gelöst, sondern für die heute nach ihm benannten *Turing-Maschinen*. Eine Turing-Maschine ist abstrakt gesehen nichts anderes als eine Beschreibung eines Algorithmus. Turing hat also gezeigt, dass es keinen Algorithmus gibt, der entscheiden kann, ob ein gegebener anderer Algorithmus terminiert.

**Bemerkung**: An dieser Stelle können wir uns fragen, ob es vielleicht eine andere Programmier-Sprache gibt, in der wir das Halte-Problem dann vielleicht doch lösen könnten. Wenn es in dieses Programmier-Sprache Unterprogramme gibt, und wenn wir dort Programm-Texte als Argumente von Funktionen übergeben können, dann ist leicht zu sehen, dass der obige Beweis der Unlösbarkeit des Halte-Problems sich durch geeignete syntaktische Modifikationen auch auf die andere Programmier-Sprache übertragen läßt.

**Aufgabe 1**: Wir nennen eine Menge $X$ *abzählbar*, wenn es eine Funktion

$$f : \mathbb{N} \to X$$

gibt, so dass es für alle $x \in X$ ein $n \in \mathbb{N}$ gibt, so dass $x$ das Bild von $n$ unter $f$ ist:

$$\forall x \in X : \exists n \in \mathbb{N} : x = f(n).$$

Zeigen Sie, dass die Potenz-Menge $2^{\mathbb{N}}$ der natürlichen Zahlen $\mathbb{N}$ nicht abzählbar ist.

**Hinweis**: Gehen Sie ähnlich vor wie beim Beweis der Unlösbarkeit des Halte-Problems. Nehmen Sie an, es gäbe eine Funktion $f$, die die Teilmengen von $\mathbb{N}$ aufzählt:

$$\forall x \in 2^{\mathbb{N}} : \exists n \in \mathbb{N} : x = f(n).$$

Definieren Sie eine Menge `Cantor` wie folgt:

$$\texttt{Cantor} := \big\{n \in \mathbb{N} \mid n \notin f(n)\big\}.$$

Versuchen Sie nun, einen Widerspruch herzuleiten.

## 2.2 Unlösbarkeit des Äquivalenz-Problems

Es gibt noch eine ganze Reihe anderer Funktionen, die nicht berechenbar sind. In der Regel werden wir den Nachweis, dass eine bestimmt Funktion nicht berechenbar ist, dadurch führen, dass wir zunächst annehmen, dass die gesuchte Funktion doch implementierbar ist. Unter dieser Annahme konstruieren wir dann eine Funktion, die das Halte-Problem löst, was im Widerspruch zu dem am Anfang dieses Abschnitts bewiesen Sachverhalts steht. Dieser Widerspruch zwingt uns zu der Folgerung, dass die gesuchte Funktion nicht implementierbar ist. Wir werden dieses Verfahren an einem Beispiel demonstrieren. Vorweg benötigen wir aber noch eine Definition.

**Definition 4 ($\simeq$)** *Es seien $n_1$ und $n_2$ Namen zweier* SetlX*-Funktionen und $a_1, \cdots, a_k$ seien Argumente, mit denen wir diese Funktionen füttern können. Wir definieren*

$$n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$$

*g.d.w. einer der beiden folgen Fälle auftritt:*

1. $n_1(a_1, \cdots, a_k) \uparrow \quad \wedge \quad n_2(a_1, \cdots, a_k) \uparrow$
2. $\exists r : \Big(n_1(a_1, \cdots, a_k) \rightsquigarrow r \quad \wedge \quad n_2(a_1, \cdots, a_k) \rightsquigarrow r\Big)$

   *In diesem Fall sagen wir, dass die beiden Funktions-Aufrufe $n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$* partiell äquivalent *sind.* □

Wir kommen jetzt zum *Äquivalenz-Problem*. Die Funktion *equal*, die die Form

```
equal := procedure(p1, p2, a) { ... }
```

hat, möge folgender Spezifikation genügen:

1. $p_1 \notin TF \ \vee \ p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \rightsquigarrow 2.$
2. Falls
   (a) $p_1 \in TF \ \wedge \ \texttt{name}(p_1) = n_1,$
   (b) $p_2 \in TF \ \wedge \ \texttt{name}(p_2) = n_2$ und
   (c) $n_1(a) \simeq n_2(a)$

   gilt, dann muß gelten:
   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 1.$$
3. Ansonsten gilt
   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 0.$$

Wir sagen, dass eine Funktion, die der eben angegebenen Spezifikation genügt, das *Äquivalenz-Problem* löst.

**Theorem 5 (Rice, 1953)** Das Äquivalenz-Problem ist unlösbar.

**Beweis**: Wir führen den Beweis indirekt und nehmen an, dass es doch eine Implementierung der Funktion `equal` gibt, die das Äquivalenz-Problem löst. Wir betrachten die in Abbildung 2.2 angegeben Implementierung der Funktion `stops`.

```
1   stops := procedure(p, a) {
2       f := "loop := procedure(x) {           \n"
3          + "    while (true) { x := x + x; } \n"
4          + "    return 0;                    \n"
5          + "};                               \n";
6       e := equal(f, p, a);
7       if (e == 2) {
8           return 2;
9       } else {
10          return 1 - e;
11      }
12  }
```

Abbildung 2.2: Eine Implementierung der Funktion `stops`.

Zu beachten ist, dass in Zeile 2 die Funktion `equal` mit einem String aufgerufen wird, der eine Test-Funktion ist, und zwar mit dem Namen `loop`. Diese Test-Funktion hat die folgende Form:

```
loop := procedure(x) { while (1) { x := x + x; } };
```

Es ist offensichtlich, dass die Funktion `loop` für kein Ergebnis terminiert. Ist also das Argument $p$ eine Test-Funktion mit Namen $n$, so liefert die Funktion `equal` immer dann den Wert 1, wenn $n(a)$ nicht terminiert, andernfalls muß sie den Wert 0 zurück geben. Damit liefert die Funktion `stops` aber für eine Test-Funktion $p$ mit Namen $n$ und ein Argument $a$ genau dann 1, wenn der Aufruf $n(a)$ terminiert und würde folglich das Halte-Problem lösen. Das kann nicht sein, also kann es keine Funktion `equal` geben, die das Äquivalenz-Problem löst. □

Die Unlösbarkeit des Äquivalenz-Problems und vieler weiterer praktisch interessanter Problem folgen aus einem 1953 von Henry G. Rice [Ric53] bewiesenen Satz.

# Kapitel 3

# Die $\mathcal{O}$-Notation

In diesem Kapitel stellen wir die $O$-Notation vor. Diese beiden Begriffe benötigen wir, um die Laufzeit von Algorithmen analysieren zu können. Die Algorithmen selber stehen in diesem Kapitel noch im Hintergrund.

## 3.1 Motivation

Wollen wir die Komplexität eines Algorithmus abschätzen, so wäre ein mögliches Vorgehen wie folgt: Wir kodieren den Algorithmus in einer Programmiersprache und berechnen, wieviele Additionen, Multiplikationen, Zuweisungungen, und andere elementare Operationen bei einer gegebenen Eingabe von dem Programm ausgeführt werden. Anschließend schlagen wir im Prozessor-Handbuch nach, wieviel Zeit die einzelnen Operationen in Anspruch nehmen und errechnen daraus die Gesamtlaufzeit des Programms.[1] Dieses Vorgehen ist aber in zweifacher Hinsicht problematisch:

1. Das Verfahren ist sehr kompliziert.
2. Würden wir den selben Algorithmus anschließend in einer anderen Programmier-Sprache kodieren, oder aber das Programm auf einem anderen Rechner laufen lassen, so wäre unsere Rechnung wertlos und wir müßten sie wiederholen.

Der letzte Punkt zeigt, dass das Verfahren dem Begriff des Algorithmus, der ja eine Abstraktion des Programm-Begriffs ist, nicht gerecht wird. Ähnlich wie der Begriff des Algorithmus von bestimmten Details einer Implementierung abstrahiert brauchen wir zur Erfassung der rechenzeitlichen Komplexität eines Algorithmus einen Begriff, der von bestimmten Details der Funktion, die die Rechenzeit für ein gegebenes Programm berechnet, abstrahiert. Wir haben drei Forderungen an den zu findenden Begriff.

- Der Begriff soll von konstanten Faktoren abstrahieren.
- Der Begriff soll von *unwesentlichen Termen* abstrahieren.

  Nehmen wir an, wir hätten ein Programm, dass zwei $n \times n$ Matrizen multipliziert und wir hätten für die Rechenzeit $T(n)$ dieses Programms in Abhängigkeit von $n$ die Funktion

  $$T(n) = 3 \cdot n^3 + 2 \cdot n^2 + 7$$

  gefunden. Dann nimmt der *proportionale Anteil* des Terms $2 \cdot n^2 + 7$ an der gesamten Rechenzeit mit wachsendem $n$ immer mehr ab. Zur Verdeutlichung haben wir in einer Tabelle die Werte des proportionalen Anteils für $n = 1,\ 10,\ 100,\ 1000,\ 10\,000$ aufgelistet:

[1] Da die heute verfügbaren Prozessoren fast alle mit *Pipelining* arbeiten, werden oft mehrere Befehle gleichzeitig abgearbeitet. Da gleichzeitig auch das Verhalten des Caches eine wichtige Rolle spielt, ist die genaue Berechnung der Rechenzeit faktisch unmöglich.

| $n$ | $\frac{2 \cdot n^2 + 7}{3 \cdot n^3 + 2 \cdot n^2 + 7}$ |
|---|---|
| 1 | 0.75000000000000 |
| 10 | 0.06454630495800 |
| 100 | 0.00662481908150 |
| 1000 | 0.00066622484855 |
| 10 000 | 6.6662224852e-05 |

- Der Begriff soll das *Wachstum* der Rechenzeit abhängig von *Wachstum* der Eingaben erfassen. Welchen genauen Wert die Rechenzeit für kleine Werte der Eingaben hat, spielt nur eine untergeordnete Rolle, denn für kleine Werte der Eingaben wird auch die Rechenzeit nur klein sein.

Wir bezeichnen die Menge der positiven reellen Zahlen mit $\mathbb{R}_+$

$$\mathbb{R}_+ := \{x \in \mathbb{R} \mid x > 0\}.$$

Wir bezeichnen die Menge aller Funktionen von $\mathbb{N}$ nach $\mathbb{R}_+$ mit $\mathbb{R}_+^{\mathbb{N}}$, es gilt also:

$$\mathbb{R}_+^{\mathbb{N}} = \big\{f \mid f \text{ ist Funktion der Form } f : \mathbb{N} \to \mathbb{R}_+\big\}.$$

**Definition 6 ($\mathcal{O}(f)$)** *Es sei eine Funktion $f \in \mathbb{R}_+^{\mathbb{N}}$ gegeben. Dann definieren wir die Menge der Funktionen, die asymptotisch das gleiche Wachstumsverhalten haben wie die Funktion $f$, wie folgt:*

$$\mathcal{O}(f) \;:=\; \big\{g \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N}\colon \big(\exists c \in \mathbb{R}\colon \forall n \in \mathbb{N}\colon n \geq k \to g(n) \leq c \cdot f(n)\big)\big\}. \qquad \square$$

Was sagt die obige Definition aus? Zunächst kommt es auf kleine Werte des Arguments $n$ nicht an, denn die obige Formel sagt ja, dass $g(n) \leq c \cdot f(n)$ nur für die $n$ gelten muss, für die $n \geq k$ ist. Außerdem kommt es auf Proportionalitäts-Konstanten nicht an, denn $g(n)$ muss ja nur kleinergleich $c \cdot f(n)$ sein und die Konstante $c$ können wir beliebig wählen. Um den Begriff zu verdeutlichen, geben wir einige Beispiele.

**Beispiel**: Es gilt

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \in \mathcal{O}(n^3).$$

**Beweis**: Wir müssen eine Konstante $c$ und eine Konstante $k$ angeben, so dass für alle $n \in \mathbb{N}$ mit $n \geq k$ die Ungleichung

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq c \cdot n^3$$

gilt. Wir setzen $k := 1$ und $c := 12$. Dann können wir die Ungleichung

$$1 \leq n \tag{3.1}$$

voraussetzen und müssen zeigen, dass daraus

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3 \tag{3.2}$$

folgt. Erheben wir beide Seiten der Ungleichung (3.1) in die dritte Potenz, so sehen wir, dass

$$1 \leq n^3 \tag{3.3}$$

gilt. Diese Ungleichung multiplizieren wir auf beiden Seiten mit 7 und erhalten:

$$7 \leq 7 \cdot n^3 \tag{3.4}$$

Multiplizieren wir die Ungleichung (3.1) mit $2 \cdot n^2$, so erhalten wir

$$2 \cdot n^2 \leq 2 \cdot n^3 \tag{3.5}$$

Schließlich gilt trivialerweise

$$3 \cdot n^3 \leq 3 \cdot n^3 \tag{3.6}$$

Die Addition der Ungleichungen (3.4), (3.5) und (3.6) liefert nun

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3$$

und das war zu zeigen. □

**Beispiel**: Es gilt $n \in \mathcal{O}(2^n)$.

**Beweis**: Wir müssen eine Konstante $c$ und eine Konstante $k$ angeben, so dass für alle $n \geq k$

$$n \leq c \cdot 2^n$$

gilt. Wir setzen $k := 0$ und $c := 1$. Wir zeigen

$$n \leq 2^n \quad \text{für alle } n \in \mathbb{N}$$

durch vollständige Induktion über $n$.

1. **I.A.**: $n = 0$

   Es gilt $0 \leq 1 = 2^0$.

2. **I.S.**: $n \mapsto n + 1$

   Einerseits gilt nach Induktions-Voraussetzung

   $$n \leq 2^n, \tag{1}$$

   andererseits haben wir

   $$1 \leq 2^n. \tag{2}$$

   Addieren wir (1) und (2), so erhalten wir

   $$n + 1 \leq 2^n + 2^n = 2^{n+1}.$$ □

   **Bemerkung**: Die Ungleichung $1 \leq 2^n$ hätten wir eigentlich ebenfalls durch Induktion nachweisen müssen.

**Aufgabe 2**: Zeigen Sie

$$n^2 \in \mathcal{O}(2^n).$$

Wir zeigen nun einige Eigenschaften der $\mathcal{O}$-Notation.

**Satz 7 (Reflexivität)** *Für alle Funktionen $f: \mathbb{N} \to \mathbb{R}_+$ gilt*

$$f \in \mathcal{O}(f).$$

**Beweis**: Wählen wir $k := 0$ und $c := 1$, so folgt die Behauptung sofort aus der Ungleichung

$$\forall n \in \mathbb{N}: f(n) \leq f(n).$$ □

**Satz 8 (Abgeschlossenheit unter Multiplikation mit Konstanten)**
*Es seien $f, g: \mathbb{N} \to \mathbb{R}_+$ und $d \in \mathbb{R}_+$. Dann gilt*

$$g \in \mathcal{O}(f) \Rightarrow d \cdot g \in \mathcal{O}(f).$$

**Beweis**: Aus $g \in \mathcal{O}(f)$ folgt, dass es Konstanten $c' \in \mathbb{R}_+$, $k' \in \mathbb{N}$ gibt, so dass

$$\forall n \in \mathbb{N}: \big(n \geq k' \to g(n) \leq c' \cdot f(n)\big)$$

gilt. Multiplizieren wir die Ungleichung mit $d$, so haben wir

$$\forall n \in \mathbb{N}: \big(n \geq k' \to d \cdot g(n) \leq d \cdot c' \cdot f(n)\big)$$

Setzen wir nun $k := k'$ und $c := d \cdot c'$, so folgt

$$\forall n \in \mathbb{N}: \big(n \geq k \to d \cdot g(n) \leq c \cdot f(n)\big)$$

und daraus folgt $d \cdot g \in \mathcal{O}(f)$. □.

**Satz 9 (Abgeschlossenheit unter Addition)** *Es seien $f, g, h: \mathbb{N} \to \mathbb{R}_+$. Dann gilt*

$$f \in \mathcal{O}(h) \wedge g \in \mathcal{O}(h) \;\to\; f + g \in \mathcal{O}(h).$$

**Beweis**: Aus den Voraussetzungen $f \in \mathcal{O}(h)$ und $g \in \mathcal{O}(h)$ folgt, dass es Konstanten $k_1, k_2 \in \mathbb{N}$ und $c_1, c_2 \in \mathbb{R}$ gibt, so dass

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot h(n)\big) \quad \text{und}$$
$$\forall n \in \mathbb{N}\colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

gilt. Wir setzen $k := \max(k_1, k_2)$ und $c := c_1 + c_2$. Für $n \geq k$ gilt dann

$$f(n) \leq c_1 \cdot h(n) \text{ und } g(n) \leq c_2 \cdot h(n).$$

Addieren wir diese beiden Gleichungen, dann haben wir für alle $n \geq k$

$$f(n) + g(n) \leq (c_1 + c_2) \cdot h(n) = c \cdot h(n).$$ □

**Satz 10 (Transitivität)** *Es seien $f, g, h\colon \mathbb{N} \to \mathbb{R}_+$. Dann gilt*

$$f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \;\rightarrow\; f \in \mathcal{O}(h).$$

**Beweis**: Aus $f \in \mathcal{O}(g)$ folgt, dass es $k_1 \in \mathbb{N}$ und $c_1 \in \mathbb{R}$ gibt, so dass

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot g(n)\big)$$

gilt und aus $g \in \mathcal{O}(h)$ folgt, dass es $k_2 \in \mathbb{N}$ und $c_2 \in \mathbb{R}$ gibt, so dass

$$\forall n \in \mathbb{N}\colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

gilt. Wir definieren $k := \max(k_1, k_2)$ und $c := c_1 \cdot c_2$. Dann haben wir für alle $n \geq k$:

$$f(n) \leq c_1 \cdot g(n) \text{ und } g(n) \leq c_2 \cdot h(n).$$

Die zweite dieser Ungleichungen multiplizieren wir mit $c_1$ und erhalten

$$f(n) \leq c_1 \cdot g(n) \text{ und } c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n).$$

Daraus folgt aber sofort $f(n) \leq c \cdot h(n)$. □

**Satz 11 (Grenzwert-Satz)** *Es seien $f, g\colon \mathbb{N} \to \mathbb{R}_+$. Außerdem existiere der Grenzwert*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

*Dann gilt $f \in \mathcal{O}(g)$.*

**Beweis**: Es sei

$$\lambda := \lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

Nach Definition des Grenzwertes gibt es dann eine Zahl $k \in \mathbb{N}$, so dass für alle $n \in \mathbb{N}$ mit $n \geq k$ die Ungleichung

$$\left| \frac{f(n)}{g(n)} - \lambda \right| \leq 1$$

gilt. Multiplizieren wir diese Ungleichung mit $g(n)$, so erhalten wir

$$|f(n) - \lambda \cdot g(n)| \leq g(n).$$

Daraus folgt wegen

$$f(n) \leq \big|f(n) - \lambda \cdot g(n)\big| + \lambda \cdot g(n)$$

die Ungleichung

$$f(n) \leq g(n) + \lambda \cdot g(n) = (1 + \lambda) \cdot g(n).$$

Definieren wir $c := 1 + \lambda$, so folgt für alle $n \geq k$ die Ungleichung $f(n) \leq c \cdot g(n)$. □

Wir zeigen die Nützlichkeit der obigen Sätze an Hand einiger Beispiele.

**Beispiel**: Es sei $k \in \mathbb{N}$. Dann gilt

$$n^k \in \mathcal{O}(n^{k+1}).$$

**Beweis**: Es gilt

$$\lim_{n\to\infty} \frac{n^k}{n^{k+1}} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Die Behauptung folgt nun aus dem Grenzwert-Satz. □

**Beispiel**: Es sei $k \in \mathbb{N}$ und $\lambda \in \mathbb{R}$ mit $\lambda > 1$. Dann gilt

$$n^k \in \mathcal{O}(\lambda^n).$$

**Beweis**: Wir zeigen, dass

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = 0 \tag{3.7}$$

ist, denn dann folgt die Behauptung aus dem Grenzwert-Satz. Nach dem Satz von L'Hospital können wir den Grenzwert wie folgt berechnen

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = \lim_{x\to\infty} \frac{x^k}{\lambda^x} = \lim_{x\to\infty} \frac{\frac{d\,x^k}{dx}}{\frac{d\,\lambda^x}{dx}}$$

Die Ableitungen können wir berechnen, es gilt:

$$\frac{d\,x^k}{dx} = k \cdot x^{k-1} \quad \text{und} \quad \frac{d\,\lambda^x}{dx} = \ln(\lambda) \cdot \lambda^x.$$

Berechnen wir die zweite Ableitung so sehen wir

$$\frac{d^2\,x^k}{dx^2} = k \cdot (k-1) \cdot x^{k-2} \quad \text{und} \quad \frac{d^2\,\lambda^x}{dx^2} = \ln(\lambda)^2 \cdot \lambda^x.$$

Für die $k$-te Ableitung gilt analog

$$\frac{d^k\,x^k}{dx^k} = k \cdot (k-1) \cdot \cdots \cdot 1 \cdot x^0 = k! \quad \text{und} \quad \frac{d^k\,\lambda^x}{dx^k} = \ln(\lambda)^k \cdot \lambda^x.$$

Wenden wir also den Satz von L'Hospital zur Berechnung des Grenzwertes $k$ mal an, so sehen wir

$$\lim_{x\to\infty} \frac{x^k}{\lambda^x} = \lim_{x\to\infty} \frac{\frac{d\,x^k}{dx}}{\frac{d\,\lambda^x}{dx}} = \lim_{x\to\infty} \frac{\frac{d^2\,x^k}{dx^2}}{\frac{d^2\,\lambda^x}{dx^2}} = \cdots = \lim_{x\to\infty} \frac{\frac{d^k\,x^k}{dx^k}}{\frac{d^k\,\lambda^x}{dx^k}} = \lim_{x\to\infty} \frac{k!}{\ln(\lambda)^k \lambda^x} = 0.$$

□

**Beispiel**: Es gilt $\ln(n) \in \mathcal{O}(n)$.

**Beweis**: Wir benutzen Satz 11 und zeigen mit der Regel von L'Hospital, dass

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

ist. Es gilt

$$\frac{d\,\ln(x)}{dx} = \frac{1}{x} \quad \text{und} \quad \frac{d\,x}{dx} = 1.$$

Also haben wir

$$\lim_{n\to\infty} \frac{\ln(n)}{n} = \lim_{x\to\infty} \frac{\frac{1}{x}}{1} = \lim_{x\to\infty} \frac{1}{x} = 0.$$

□

**Aufgabe 3**: Zeigen Sie $\sqrt{n} \in \mathcal{O}(n)$.

**Beispiel**: Es gilt $2^n \in \mathcal{O}(3^n)$, aber $3^n \notin \mathcal{O}(2^n)$.

**Beweis**: Zunächst haben wir

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$$

Den Beweis, dass $3^n \notin \mathcal{O}(2^n)$ ist, führen wir indirekt und nehmen an, dass $3^n \in \mathcal{O}(2^n)$ ist. Dann muss es Konstanten $c$ und $k$ geben, so dass für alle $n \geq k$ gilt

$$3^n \leq c \cdot 2^n.$$

Wir logarithmieren beide Seiten dieser Ungleichung und finden

$$
\begin{array}{lll}
 & \ln(3^n) & \leq \ln(c \cdot 2^n) \\
\leftrightarrow & n \cdot \ln(3) & \leq \ln(c) + n \cdot \ln(2) \\
\leftrightarrow & n \cdot \big(\ln(3) - \ln(2)\big) & \leq \ln(c) \\
\leftrightarrow & n & \leq \dfrac{\ln(c)}{\ln(3) - \ln(2)}
\end{array}
$$

Die letzte Ungleichung müßte nun für beliebig große natürliche Zahlen $n$ gelten und liefert damit den gesuchten Widerspruch zu unserer Annahme.

**Aufgabe 4**:

1. Es sei $b \geq 1$. Zeigen Sie $\log_b(n) \in \mathcal{O}(\ln(n))$.
2. $3 \cdot n^2 + 5 \cdot n + \sqrt{n} \in \mathcal{O}(n^2)$
3. $7 \cdot n + \big(\log_2(n)\big)^2 \in \mathcal{O}(n)$
4. $\sqrt{n} + \log_2(n) \in \mathcal{O}\left(\sqrt{n}\right)$
5. $n^n \in \mathcal{O}\big(2^{2^n}\big)$.

   Hinweis: Die letzte Teilaufgabe ist schwer!

## 3.2 Fallstudie: Effiziente Berechnung der Potenz

Wir verdeutlichen die bisher eingeführten Begriffe an einem Beispiel. Wir betrachten ein Programm zur Berechnung der Potenz $m^n$ für natürliche Zahlen $m$ und $n$. Abbildung 3.1 zeigt ein naives Programm zur Berechnung von $m^n$. Die diesem Programm zu Grunde liegende Idee ist es, die Berechnung von $m^n$ nach der Formel

$$m^n = \underbrace{m \cdot \ldots \cdot m}_{n}$$

durchzuführen.

```
1  power := procedure(m, n) {
2      r := 1;
3      for (i in {1 .. n}) {
4          r := r * m;
5      }
6      return r;
7  };
```

Abbildung 3.1: Naive Berechnung von $m^n$ für $m, n \in \mathbb{N}$.

Das Programm ist offenbar korrekt. Zur Berechnung von $m^n$ werden für positive Exponenten $n$ insgesamt $n - 1$ Multiplikationen durchgeführt. Wir können $m^n$ aber wesentlich effizienter berechnen. Die Grundidee erläutern wir an der Berechnung von $m^4$. Es gilt

$$m^4 = (m \cdot m) \cdot (m \cdot m).$$

Wenn wir den Ausdruck $m \cdot m$ nur einmal berechnen, dann kommen wir bei der Berechnung von $m^4$ nach der obigen Formel mit zwei Multiplikationen aus, während bei einem naiven Vorgehen 3 Multiplikationen durchgeführt würden! Für die Berechnung von $m^8$ können wir folgende Formel verwenden:

$$m^8 = \big((m \cdot m) \cdot (m \cdot m)\big) \cdot \big((m \cdot m) \cdot (m \cdot m)\big).$$

Berechnen wir den Term $(m \cdot m) \cdot (m \cdot m)$ nur einmal, so werden jetzt 3 Multiplikationen benötigt um $m^8$ auszurechnen. Ein naives Vorgehen würde 7 Multiplikationen benötigen. Wir versuchen die oben an Beispielen erläuterte Idee in ein Programm umzusetzen. Abbildung 3.2 zeigt das Ergebnis. Es berechnet die Potenz $m^n$ nicht durch eine naive $(n-1)$-malige Multiplikation sondern es verwendet das Paradigma

*Teile und Herrsche.* (engl. *divide and conquer*)

Die Grundidee um den Term $m^n$ für $n \geq 1$ effizient zu berechnen, läßt sich durch folgende Formel beschreiben:

$$m^n = \begin{cases} m^{n/2} \cdot m^{n/2} & \text{falls } n \text{ gerade ist;} \\ m^{n/2} \cdot m^{n/2} \cdot m & \text{falls } n \text{ ungerade ist.} \end{cases}$$

```
1  power := procedure(m, n) {
2      if (n == 0) {
3          return 1;
4      }
5      p := power(m, floor(n / 2));
6      if (n % 2 == 0) {
7          return p * p;
8      } else {
9          return p * p * m;
10     }
11 };
```

Abbildung 3.2: Berechnung von $m^n$ für $m, n \in \mathbb{N}$.

Da es keineswegs offensichtlich ist, dass das Programm in 3.2 tatsächlich die Potenz $m^n$ berechnet, wollen wir dies nachweisen. Wir benutzen dazu die Methode der *Wertverlaufs-Induktion* (engl. *computational induction*). Die Wertverlaufs-Induktion ist eine Induktion über die Anzahl der rekursiven Aufrufe. Diese Methode bietet sich immer dann an, wenn die Korrektheit einer rekursiven Prozedur nachzuweisen ist. Das Verfahren besteht aus zwei Schritten:

1. *Induktions-Anfang.*

   Beim Induktions-Anfang weisen wir nach, dass die Prozedur in allen den Fällen korrekt arbeitet, in denen sie sich nicht selbst aufruft.

2. *Induktions-Schritt*

   Im Induktions-Schritt beweisen wir, dass die Prozedur auch in den Fällen korrekt arbeitet, in denen sie sich rekursiv aufruft. Beim Beweis dieser Tatsache dürfen wir voraussetzen, dass die Prozedur bei jedem rekursiven Aufruf den korrekten Wert produziert. Diese Voraussetzung wird auch als *Induktions-Voraussetzung* bezeichnet.

Wir demonstrieren die Methode, indem wir durch Wertverlaufs-Induktion beweisen, dass gilt:

$$\texttt{power}(m, n) \rightsquigarrow m^n.$$

1. **Induktions-Anfang**.

   Die Methode ruft sich dann nicht rekursiv auf, wenn $n = 0$ gilt. In diesem Fall haben wir

   $$\texttt{power}(m, 0) \rightsquigarrow 1 = m^0.$$

2. **Induktions-Schritt**.

   Der rekursive Aufruf der Prozedur `power` hat die Form $\texttt{power}(m, n/2)$. Also gilt nach Induktions-Voraussetzung

   $$\texttt{power}(m, n/2) \rightsquigarrow m^{n/2}.$$

Danach können in der weiteren Rechnung zwei Fälle auftreten. Wir führen daher eine Fallunterscheidung entsprechend der `if`-Abfrage in Zeile 6 durch:

(a) $n \,\%\, 2 = 0$, $n$ ist also gerade.
Dann gibt es ein $k \in \mathbb{N}$ mit $n = 2 \cdot k$ und also ist $n/2 = k$. In diesem Fall gilt

$$\begin{array}{lcl} \texttt{power}(m,n) & \rightsquigarrow & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \\ & \overset{I.V.}{\rightsquigarrow} & m^k \cdot m^k \\ & = & m^{2\cdot k} \\ & = & m^n. \end{array}$$

(b) $n \,\%\, 2 = 1$, $n$ ist also ungerade.
Dann gibt es ein $k \in \mathbb{N}$ mit $n = 2 \cdot k + 1$ und wieder ist $n/2 = k$. In diesem Fall gilt

$$\begin{array}{lcl} \texttt{power}(m,n) & \rightsquigarrow & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \cdot m \\ & \overset{I.V.}{\rightsquigarrow} & m^k \cdot m^k \cdot m \\ & = & m^{2\cdot k+1} \\ & = & m^n. \end{array}$$

Damit ist der Beweis der Korrektheit abgeschlossen. □

Als nächstes wollen wir die Komplexität des obigen Programms untersuchen. Dazu berechnen wir zunächst die Anzahl der Multiplikationen, die beim Aufruf $\texttt{power}(m,n)$ durchgeführt werden. Je nach dem, ob der Test in Zeile 6 negativ ausgeht oder nicht, gibt es mehr oder weniger Multiplikationen. Wir untersuchen zunächst den schlechtesten Fall (engl. *worst case*). Der schlechteste Fall tritt dann ein, wenn es ein $l \in \mathbb{N}$ gibt, so dass

$$n = 2^l - 1$$

ist, denn dann gilt

$$n/2 = 2^{l-1} - 1 \quad \text{und} \quad n \,\%\, 2 = 1,$$

was wir sofort durch die Probe

$$2 \cdot (n/2) + n \,\%\, 2 = 2 \cdot (2^{l-1} - 1) + 1 = 2^l - 1 = n$$

verifizieren. Folglich ist, wenn $n$ die Form $2^l - 1$ hat, bei jedem rekursiven Aufruf der Exponent $n$ ungerade. Wir nehmen also $n = 2^l - 1$ an und berechnen die Zahl $a_n$ der Multiplikationen, die beim Aufruf von $\texttt{power}(m,n)$ durchgeführt werden.

Zunächst gilt $a_0 = 0$, denn wenn $n = 0$ ist, wird keine Multiplikation durchgeführt. Ansonsten haben wir in Zeile 9 zwei Multiplikationen, die zu den Multiplikationen, die beim rekursiven Aufruf in Zeile 5 anfallen, hinzu addiert werden müssen. Damit erhalten wir die folgende Rekurrenz-Gleichung:

$$a_n = a_{n/2} + 2 \qquad \text{für alle } n \in \left\{2^l - 1 \mid l \in \mathbb{N}\right\} \quad \text{mit } a_0 = 0.$$

Wir definieren $b_l := a_{2^l-1}$ und erhalten dann für die Folge $(b_l)_l$ die Rekurrenz-Gleichung

$$b_l = a_{2^l-1} = a_{(2^l-1)/2} + 2 = a_{2^{l-1}-1} + 2 = b_{l-1} + 2 \qquad \text{für alle } l \in \mathbb{N}.$$

Die Anfangs-Bedingung lautet $b_0 = a_{2^0-1} = a_0 = 0$. Offenbar lautet die Lösung der Rekurrenz-Gleichung

$$b_l = 2 \cdot l \qquad \text{für alle } l \in \mathbb{N}.$$

Diese Behauptung können Sie durch eine triviale Induktion verifizieren. Für die Folge $a_n$ haben wir dann:

$$a_{2^l-1} = 2 \cdot l.$$

Formen wir die Gleichung $n = 2^l - 1$ nach $l$ um, so erhalten wir $l = \log_2(n+1)$. Setzen wir diesen Wert ein, so sehen wir

$$a_n = 2 \cdot \log_2(n+1) \in \mathcal{O}\big(\log_2(n)\big).$$

Wir betrachten jetzt den günstigsten Fall, der bei der Berechnung von `power`$(m, n)$ auftreten kann. Der günstigste Fall tritt dann ein, wenn der Test in Zeile 6 immer gelingt weil $n$ jedesmal eine gerade Zahl ist. In diesem Fall muss es ein $l \in \mathbb{N}$ geben, so dass $n$ die Form

$$n = 2^l$$

hat. Wir nehmen also $n = 2^l$ an und berechnen die Zahl $a_n$ der Multiplikationen, die dann beim Aufruf von `power`$(m, n)$ durchgeführt werden.

Zunächst gilt $a_{2^0} = a_1 = 2$, denn wenn $n = 1$ ist, scheitert der Test in Zeile 6 und Zeile 9 liefert 2 Multiplikationen. Zeile 5 liefert in diesem Fall keine Multiplikation, weil beim Aufruf `power`$(m, 0)$ sofort das Ergebnis in Zeile 4 zurück gegeben wird.

Ist $n = 2^l > 1$, so haben wir in Zeile 7 eine Multiplikation, die zu den Multiplikationen, die beim rekursiven Aufruf in Zeile 5 anfallen, hinzu addiert werden muß. Damit erhalten wir die folgende Rekurrenz-Gleichung:

$$a_n = a_{n/2} + 1 \qquad \text{für alle } n \in \big\{2^l \mid l \in \mathbb{N}\big\} \quad \text{mit } a_1 = 2.$$

Wir definieren $b_l := a_{2^l}$ und erhalten dann für die Folge $(b_l)_l$ die Rekurrenz-Gleichung

$$b_l = a_{2^l} = a_{(2^l)/2} + 1 = a_{2^{l-1}} + 1 = b_{l-1} + 1 \qquad \text{für alle } l \in \mathbb{N},$$

mit der Anfangs-Bedingungen $b_0 = a_{2^0} = a_1 = 2$. Also lösen wir die Rekurrenz-Gleichung

$$b_{l+1} = b_l + 1 \qquad \text{für alle } l \in \mathbb{N} \quad \text{mit } b_0 = 2.$$

Offenbar lautet die Lösung

$$b_l = 2 + l \qquad \text{für alle } l \in \mathbb{N}.$$

Setzen wir hier $b_l = a_{2^l}$, so erhalten wir:

$$a_{2^l} = 2 + l.$$

Formen wir die Gleichung $n = 2^l$ nach $l$ um, so erhalten wir $l = \log_2(n)$. Setzen wir diesen Wert ein, so sehen wir

$$a_n = 2 + \log_2(n) \in \mathcal{O}\big(\log_2(n)\big).$$

Da wir sowohl im besten als auch im schlechtesten Fall das selbe Ergebnis bekommen haben, können wir schließen, dass für die Zahl $a_n$ der Multiplikationen allgemein gilt:

$$a_n \in \mathcal{O}\big(\log_2(n)\big).$$

**Bemerkung**: Wenn wir nicht die Zahl der Multiplikationen sondern die Rechenzeit ermitteln wollen, die der obige Algorithmus benötigt, so wird die Rechnung wesentlich aufwendiger. Der Grund ist, dass wir dann berücksichtigen müssen, dass die Rechenzeit bei der Berechnung der Produkte in den Zeilen 7 und 9 von der Größe der Faktoren abhängig ist.

**Aufgabe 5**: Schreiben Sie eine Prozedur `prod` zur Multiplikation zweier Zahlen. Für zwei natürliche Zahlen $m$ und $n$ soll der Aufruf `prod`$(m, n)$ das Produkt $m \cdot n$ mit Hilfe von Additionen berechnen. Benutzen Sie bei der Implementierung das Paradigma "Teile und Herrsche" und beweisen Sie die Korrektheit des Algorithmus mit Hilfe einer Wertverlaufs-Induktion. Schätzen Sie die Anzahl der Additionen, die beim Aufruf von `prod`$(m, n)$ im schlechtesten Fall durchgeführt werden, mit Hilfe der $\mathcal{O}$-Notation ab.

## 3.3 Der Hauptsatz der Laufzeit-Funktionen

Im letzten Abschnitt haben wir zur Analyse der Rechenzeit der Funktion *power*() zunächst eine Rekurrenz-Gleichung aufgestellt, diese gelöst und anschließend das Ergebnis mit Hilfe der $\mathcal{O}$-Notation abgeschätzt. Wenn wir nur an einer Abschätzung interessiert sind, dann ist es in vielen Fällen nicht notwendig, die zu Grunde liegende Rekurrenz-Gleichung exakt zu lösen, denn der *Hauptsatz der Laufzeit-Funktionen* (Englisch: *Master Theorem*) [CLRS01] bietet eine Methode zur Gewinnung von Abschätzungen, bei der es nicht notwendig ist, die Rekurrenz-Gleichung zu lösen. Wir präsentieren eine etwas vereinfachte Form dieses Hauptsatzes.

**Theorem 12 (Hauptsatz der Laufzeit-Funktionen)** Es seien

1. $\alpha, \beta \in \mathbb{N}$ mit $\alpha \geq 1$ und $\beta > 1$,
2. $f : \mathbb{N} \to \mathbb{R}_+$,
3. die Funktion $g : \mathbb{N} \to \mathbb{R}_+$ genüge der Rekurrenz-Gleichung

   $$g(n) = \alpha \cdot g\left(n/\beta\right) + f(n),$$

   wobei der Ausdruck $n/\beta$ die ganzzahlige Division von $n$ durch $\beta$ bezeichnet.

Dann können wir in den gleich genauer beschriebenen Situationen asymptotische Abschätzungen für die Funktion $g(n)$ angeben:

1. Falls es eine Konstante $\varepsilon > 0$ gibt, so dass

   $$f(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)-\varepsilon}\big)$$

   gilt, dann haben wir

   $$g(n) \in \mathcal{O}\left(n^{\log_\beta(\alpha)}\right).$$

2. Falls sowohl $f(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)}\big)$ als auch $n^{\log_\beta(\alpha)} \in \mathcal{O}\big(f(n)\big)$ gilt, dann folgt

   $$g(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^{\log_\beta(\alpha)}\big).$$

3. Falls es eine Konstante $\gamma < 1$ und eine Konstante $k \in \mathbb{N}$ gibt, so dass für $n \geq k$

   $$\alpha \cdot f\left(n/\beta\right) \leq \gamma \cdot f(n)$$

   gilt, dann folgt

   $$g(n) \in \mathcal{O}\big(f(n)\big).$$

□

**Erläuterung**: Ein vollständiger Beweis dieses Theorems geht über den Rahmen einer einführenden Vorlesung hinaus. Wir wollen aber erklären, wie die drei Fälle zustande kommen.

1. Wir betrachten zunächst den ersten Fall. In diesem Fall kommt der asymptotisch wesentliche Anteil des Wachstums der Funktion $g$ von der Rekursion. Um diese Behauptung einzusehen, betrachten wir die homogene Rekurrenz-Gleichung

   $$g(n) = \alpha \cdot g\left(n/\beta\right).$$

   Wir beschränken uns auf solche Werte von $n$, die sich als Potenzen von $\beta$ schreiben lassen, also Werte der Form

   $$n = \beta^k \quad \text{mit } k \in \mathbb{N}.$$

   Definieren wir für $k \in \mathbb{N}$ die Folge $\big(b_k\big)_{k\in\mathbb{N}}$ durch

   $$b_k := g\big(\beta^k\big),$$

so erhalten wir für die Folgenglieder $b_k$ die Rekurrenz-Gleichung

$$b_k = g\big(\beta^k\big) = \alpha \cdot g\left(\beta^k/\beta\right) = \alpha \cdot g\big(\beta^{k-1}\big) = \alpha \cdot b_{k-1}.$$

Wir sehen unmittelbar, dass diese Rekurrenz-Gleichung die Lösung

$$b_k = \alpha^k \cdot b_0 \tag{3.8}$$

hat. Aus $n = \beta^k$ folgt sofort

$$k = \log_\beta(n).$$

Berücksichtigen wir, dass $b_k = g(n)$ ist, so liefert Gleichung (3.8) also

$$g(n) = \alpha^{\log_\beta(n)} \cdot b_0. \tag{3.9}$$

Wir zeigen, dass

$$\alpha^{\log_\beta(n)} = n^{\log_\beta(\alpha)} \tag{3.10}$$

gilt. Dazu betrachten wir die folgende Kette von Äquivalenz-Umformungen:

$$\begin{array}{lllll} & \alpha^{\log_\beta(n)} & = & n^{\log_\beta(\alpha)} & \mid \log_\beta(\cdot) \\ \Leftrightarrow & \log_\beta\left(\alpha^{\log_\beta(n)}\right) & = & \log_\beta\left(n^{\log_\beta(\alpha)}\right) & \\ \Leftrightarrow & \log_\beta(n) \cdot log_\beta(\alpha) & = & \log_\beta(\alpha) \cdot \log_\beta(n) & \text{wegen } \log_b(x^y) = y \cdot \log_b(x) \end{array}$$

Da die letzte Gleichung offenbar richtig ist, und wir zwischendurch nur Äquivalenz-Umformungen durchgeführt haben, ist auch die erste Gleichung richtig und wir haben Gleichung (3.10) gezeigt. Insgesamt haben wir damit

$$g(n) = n^{\log_\beta(\alpha)} \cdot b_0$$

gezeigt. Also gilt: Vernachlässigen wir die Inhomogenität $f$, so erhalten wir die folgende asymptotische Abschätzung:

$$g(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)}\big).$$

2. Im zweiten Fall liefert die Inhomogenität $f$ einen Beitrag, der genau so groß ist wie die Lösung der homogenen Rekurrenz-Gleichung. Dies führt dazu, dass die Lösung asymptotisch um einen Faktor $\log_\beta(n)$ größer wird. Um das zu verstehen, betrachten wir exemplarisch die Rekurrenz-Gleichung

   $$g(n) = \alpha \cdot g\left(n/\beta\right) + n^{\log_\beta(\alpha)}$$

   mit der Anfangs-Bedingung $g(1) = 0$. Wir betrachten wieder nur Werte $n \in \{\beta^k \mid k \in \mathbb{N}\}$ und setzen daher

   $$n = \beta^k.$$

   Wie eben definieren wir

   $$b_k := g(n) = g\big(\beta^k\big).$$

Das liefert

$$b_k = \alpha \cdot g\left(\beta^k/\beta\right) + \left(\beta^k\right)^{\log_\beta(\alpha)} = \alpha \cdot g(\beta^{k-1}) + \left(\beta^{\log_\beta(\alpha)}\right)^k = \alpha \cdot b_{k-1} + \alpha^k.$$

Nun gilt $b_0 = g(1) = 0$. Um die Rekurrenz-Gleichung $b_k = \alpha \cdot b_{k-1} + \alpha^k$ zu lösen, berechnen wir zunächst die Werte für $k = 1, 2, 3$:

$$\begin{aligned} b_1 &= \alpha \cdot b_0 + \alpha^1 \\ &= \alpha \cdot 0 + \alpha \\ &= 1 \cdot \alpha^1 \end{aligned}$$

$$\begin{aligned} b_2 &= \alpha \cdot b_1 + \alpha^1 \\ &= \alpha \cdot 1 \cdot \alpha^1 + \alpha^2 \\ &= 2 \cdot \alpha^2 \end{aligned}$$

$$\begin{aligned} b_3 &= \alpha \cdot b_2 + \alpha^2 \\ &= \alpha \cdot 2 \cdot \alpha^2 + \alpha^3 \\ &= 3 \cdot \alpha^3 \end{aligned}$$

Wir vermuten hier, dass die Lösung dieser Rekurrenz-Gleichung durch die Formel

$$b_k = k \cdot \alpha^k$$

gegeben wird. Den Nachweis dieser Vermutung führen wir durch eine triviale Induktion:

I.A.: $k = 0$
Einerseits gilt $b_0 = 0$, andererseits gilt $0 \cdot \alpha^0 = 0$.

I.S.: $k \mapsto k + 1$

$$\begin{aligned} b_{k+1} &= \alpha \cdot b_k + \alpha^{k+1} \\ &\overset{IV}{=} \alpha \cdot k \cdot \alpha^k + \alpha^{k+1} \\ &= k \cdot \alpha^{k+1} + \alpha^{k+1} \\ &= (k+1) \cdot \alpha^{k+1}. \end{aligned}$$

Da aus $n = \beta^k$ sofort $k = \log_\beta(n)$ folgt, ergibt sich für die Funktion $g(n)$

$$g(n) = b_k = k \cdot \alpha^k = \log_\beta(n) \cdot \alpha^{\log_\beta(n)} = \log_\beta(n) \cdot n^{\log_\beta(\alpha)}$$

und das ist genau die Form, durch die im zweiten Fall des Hauptsatzes die Funktion $g(n)$ abgeschätzt wird.

3. Im letzten Fall des Hauptsatzes überwiegt schließlich der Beitrag der Inhomogenität, so dass die Lösung nun asymptotisch durch die Inhomogenität dominiert wird. Wir machen wieder den Ansatz

$$n = \beta^k \quad \text{und} \quad b_k = g\big(\beta^k\big).$$

Wir überlegen uns, wie die Ungleichung

$$\alpha \cdot f\left(n/\beta\right) \leq \gamma \cdot f(n)$$

für $n = \beta^k$ aussieht und erhalten

$$\alpha \cdot f\left(\beta^{k-1}\right) \leq \gamma \cdot f\big(\beta^k\big) \tag{3.11}$$

Setzen wir hier für $k$ den Wert $k-1$ ein, so erhalten wir

$$\alpha \cdot f\left(\beta^{k-2}\right) \leq \gamma \cdot f\left(\beta^{k-1}\right) \tag{3.12}$$

Wir multiplizieren nun die Ungleichung (3.12) mit $\alpha$ und Ungleichung (3.11) mit $\gamma$ und erhalten die Ungleichungen

$$\alpha^2 \cdot f\left(\beta^{k-2}\right) \leq \alpha \cdot \gamma \cdot f\left(\beta^{k-1}\right) \quad \text{und} \quad \alpha \cdot \gamma \cdot f\left(\beta^{k-1}\right) \leq \gamma^2 \cdot f\left(\beta^{k}\right)$$

Setzen wir diese Ungleichungen zusammen, so erhalten wir die neue Ungleichung

$$\alpha^2 \cdot f\left(\beta^{k-2}\right) \leq \gamma^2 \cdot f\left(\beta^{k}\right)$$

Iterieren wir diesen Prozess, so sehen wir, dass

$$\alpha^i \cdot f\left(\beta^{k-i}\right) \leq \gamma^i \cdot f(\beta^k) \quad \text{für alle } i \in \{1, \cdots k\} \text{ gilt.} \tag{3.13}$$

Wir berechnen nun $g(\beta^k)$ durch Iteration der Rekurrenz-Gleichung:

$$\begin{aligned}
g(\beta^k) &= \alpha \cdot g(\beta^{k-1}) + f(\beta^k) \\
&= \alpha \cdot \left(\alpha \cdot g(\beta^{k-2}) + f(\beta^{k-1})\right) + f(\beta^k) \\
&= \alpha^2 \cdot g(\beta^{k-2}) + \alpha \cdot f(\beta^{k-1}) + f(\beta^k) \\
&= \alpha^3 \cdot g(\beta^{k-3}) + \alpha^2 \cdot f(\beta^{k-2}) + \alpha \cdot f(\beta^{k-1}) + f(\beta^k) \\
&\;\;\vdots \\
&= \alpha^k \cdot g(\beta^0) + \alpha^{k-1} \cdot f(\beta^1) + \cdots + \alpha^1 \cdot f(\beta^{k-1}) + \alpha^0 \cdot f(\beta^k) \\
&= \alpha^k \cdot g(\beta^0) + \sum_{i=1}^{k} \alpha^{k-i} \cdot f(\beta^i)
\end{aligned}$$

Da bei der $\mathcal{O}$-Notation die Werte von $f$ für kleine Argumente keine Rolle spielen, können wir ohne Beschränkung der Allgemeinheit annehmen, dass $g(\beta^0) \leq f(\beta^0)$ ist. Damit erhalten wir dann die Abschätzung

$$\begin{aligned}
g(\beta^k) &\leq \alpha^k \cdot f(\beta^0) + \sum_{i=1}^{k} \alpha^{k-i} \cdot f(\beta^i) \\
&= \sum_{i=0}^{k} \alpha^{k-i} \cdot f(\beta^i) \\
&= \sum_{j=0}^{k} \alpha^{j} \cdot f(\beta^{k-j})
\end{aligned}$$

wobei wir im letzten Schritt den Index $i$ durch $k-j$ ersetzt haben. Berücksichtigen wir nun die Ungleichung (3.13), so erhalten wir die Ungleichungen

$$\begin{aligned}
g(\beta^k) &\leq \sum_{j=0}^{k} \gamma^j \cdot f(\beta^k) \\
&= f(\beta^k) \cdot \sum_{j=0}^{k} \gamma^j \\
&\leq f(\beta^k) \cdot \sum_{j=0}^{\infty} \gamma^j \\
&= f(\beta^k) \cdot \frac{1}{1-\gamma},
\end{aligned}$$

wobei wir im letzten Schritt die Formel für die geometrische Reihe

$$\sum_{j=0}^{\infty} q^j = \frac{1}{1-q}$$

benutzt haben. Ersetzen wir nun $\beta^k$ wieder durch $n$, so sehen wir, dass

$$g(n) \leq \frac{1}{1-\gamma} \cdot f(n)$$

gilt und daraus folgt sofort

$$g(n) \in \mathcal{O}\big(f(n)\big).$$ □

**Beispiel**: Wir untersuchen das asymptotische Wachstum der Folge, die durch die Rekurrenz-Gleichung

$$a_n = 9 \cdot a_{n/3} + n$$

definiert ist. Wir haben hier

$$g(n) = 9 \cdot g(n/3) + n, \quad \text{also} \quad \alpha = 9, \quad \beta = 3, \quad f(n) = n.$$

Damit gilt

$$\log_\beta(\alpha) = \log_3(9) = 2.$$

Wir setzen $\varepsilon := 1 > 0$. Dann gilt

$$f(n) = n \in \mathcal{O}(n) = \mathcal{O}\left(n^{2-1}\right) = \mathcal{O}\left(n^{2-\varepsilon}\right).$$

Damit liegt der erste Fall des Hauptsatzes vor und wir können schließen, dass

$$g(n) \in \mathcal{O}(n^2)$$

gilt. □

**Beispiel**: Wir betrachten die Rekurrenz-Gleichung

$$a_n = a_{n/2} + 2$$

und analysieren das asymptotische Wachstum der Funktion $n \mapsto a_n$ mit Hilfe des Hauptsatzes der Laufzeit-Funktionen. Wir setzen $g(n) := a_n$ und haben also für die Funktion $g$ die Rekurrenz-Gleichung

$$g(n) = 1 \cdot g\left(n/2\right) + 2$$

Wir definieren $\alpha := 1$, $\beta := 2$ und $f(n) = 2$. Wegen

$$\log_\beta(\alpha) = \log_2(1) = 0 \quad \text{und} \quad 2 \in \mathcal{O}(1) = \mathcal{O}(n^0) \quad \text{sowie} \quad n^0 \in \mathcal{O}(2)$$

sind die Voraussetzungen des zweiten Falls erfüllt und wir erhalten

$$a_n \in \mathcal{O}\big(\log_2(n)\big).$$ □

**Beispiel**: Diesmal betrachten wir die Rekurrenz-Gleichung

$$a_n = 3 \cdot a_{n/4} + n \cdot \log_2(n).$$

Es gilt $\alpha = 3$, $\beta = 4$ und $f(n) = n \cdot \log_2(n)$. Damit gilt

$$log_\beta(\alpha) = \log_4(3) < 1.$$

Damit ist klar, dass die Funktion $f(n) = n \cdot log_2(n)$ schneller wächst als die Funktion $n^{\log_4(3)}$. Damit kann höchstens der dritte Fall des Hauptsatzes vorliegen. Wir suchen also ein $\gamma < 1$, so dass die Ungleichung

$$\alpha \cdot f(n/\beta) \leq \gamma \cdot f(n)$$

gilt. Setzen wir hier die Funktion $f(n) = n \cdot \log_2(n)$ und die Werte für $\alpha$ und $\beta$ ein, so erhalten wir die Ungleichung

$$3 \cdot n/4 \cdot \log_2(n/4) \leq \gamma \cdot n \cdot \log_2(n),$$

die für durch 4 teilbares $n$ offenbar äquivalent ist zu

$$\frac{3}{4} \cdot \log_2(n/4) \leq \gamma \cdot \log_2(n).$$

Setzen wir $\gamma := \frac{3}{4}$ und kürzen, so geht diese Ungleichung über in die offensichtlich wahre Ungleichung

$$\log_2(n/4) \leq \log_2(n).$$

Damit liegt also der dritte Fall des Hauptsatzes vor und wir können schließen, dass

$$a_n \in \mathcal{O}\left(n \cdot \log_2(n)\right)$$

gilt. □

**Aufgabe 6**: Benutzen Sie den Hauptsatz der Laufzeit-Funktionen um das asymptotische Wachstum der Folgen $\left(a_n\right)_{n\in\mathbb{N}}$, $\left(b_n\right)_{n\in\mathbb{N}}$ und $\left(c_n\right)_{n\in\mathbb{N}}$ abzuschätzen, falls diese Folgen den nachstehenden Rekurrenz-Gleichungen genügen:

1. $a_n = 4 \cdot a_{n/2} + 2 \cdot n + 3$.
2. $b_n = 4 \cdot b_{n/2} + n^2$.
3. $c_n = 3 \cdot c_{n/2} + n^3$.

**Bemerkung**: Es ist wichtig zu sehen, dass die drei Fälle des Theorems nicht vollständig sind: Es gibt Situationen, in denen der Hauptsatz nicht anwendbar ist. Beispielsweise läßt sich der Hauptsatz nicht für die Funktion $g$, die durch die Rekurrenz-Gleichung

$$g(n) = 2 \cdot g(n/2) + n \cdot \log_2(n) \quad \text{mit der Anfangs-Bedingung } g(1) = 0$$

definiert ist, anwenden, denn die Inhomogenität wächst schneller als im zweiten Fall, aber nicht so schnell, dass der dritte Fall vorliegen würde. Dies können wir wie folgt sehen. Es gilt

$$\alpha = 2, \quad \beta = 2 \quad \text{und damit} \quad \log_\beta(\alpha) = 1.$$

Damit der zweite Fall vorliegt, müßte

$$n \cdot \log_2(n) \in \mathcal{O}(n^1)$$

gelten, was sicher falsch ist. Da die Inhomogenität $n \cdot \log_2(n)$ offenbar schneller wächst als der Term $n^1$, kann jetzt höchstens noch der dritte Fall vorliegen. Um diese Vermutung zu überprüfen, nehmen wir an, dass ein $\gamma < 1$ existiert, so dass die Inhomogenität

$$f(n) := n \cdot \log_2(n)$$

die Ungleichung

$$\alpha \cdot f(n/\beta) \leq \gamma \cdot f(n)$$

erfüllt. Einsetzen von $f$ sowie von $\alpha$ und $\beta$ führt auf die Ungleichung

$$2 \cdot n/2 \cdot \log_2(n/2) \leq \gamma \cdot n \cdot \log_2(n).$$

Dividieren wir diese Ungleichung durch $n$ und vereinfachen, so erhalten wir

$$\log_2(n) - \log_2(2) \leq \gamma \cdot \log_2(n).$$

Wegen $\log_2(2) = 1$ addieren wir auf beiden Seiten 1 und subtrahieren $\gamma \cdot \log_2(n)$. Dann erhalten wir

$$\log_2(n) \cdot (1 - \gamma) \leq 1,$$

woraus schließlich

$$\log_2(n) \leq \frac{1}{1-\gamma}$$

folgt. Daraus folgt durch Anwenden der Funktion $x \mapsto 2^x$ die Ungleichung

$$n \leq 2^{\frac{1}{1-\gamma}},$$

die aber sicher nicht für beliebige $n$ gelten kann. Damit haben wir einen Widerspruch zu der Annahme, dass der dritte Fall des Hauptsatzes vorliegt. □

**Aufgabe 7**: Lösen Sie die Rekurrenz-Gleichung

$$g(n) = 2 \cdot g(n/2) + n \cdot \log_2(n) \quad \text{mit der Anfangs-Bedingung } g(1) = 0$$

für den Fall, dass $n$ eine Zweier-Potenz ist. □

# Kapitel 4

# Der Hoare-Kalkül

In diesem Kapitel stellen wir den *Hoare-Kalkül* vor, mit dessen Hilfe sich die Korrektheit nichtrekursiver Programme zeigen läßt. Dieses Hilfsmittel ist der *Hoare-Kalkül*, der 1969 von C. A. R. Hoare [Hoa69] vorgestellt wurde.

## 4.1 Vor- und Nachbedingungen

Grundlage des Hoare-Kalkül sind sogenannte *Vor-* und *Nach-Bedingungen*. Ist `P` ein Programm-Fragment und sind $F$ und $G$ logische Formeln, so sagen wir, dass $F$ eine Vor-Bedingung und $G$ eine Nach-Bedingung für das Programm-Fragment `P` ist, falls folgendes gilt: Wird das Programm-Fragment `P` in einer Situation ausgeführt, in der vor der Ausführung von `P` die Formel $F$ gilt, so gilt nach der Ausführung von $P$ die Formel $G$. Dies schreiben wir als

$$\{F\} \quad \texttt{P} \quad \{G\}$$

und sagen, dass $P$ *die Spezifikation "wenn vorher $F$, dann nachher $G$"* erfüllt. Die Spezifikation

$$\{F\} \quad \texttt{P} \quad \{G\}$$

wird in der Literatur als *Hoare-Tripel* bezeichnet, denn diese Notation wurde von Sir Charles Antony Richard Hoare (geb. 1934) [Hoa69] eingeführt.

**Beispiele**:

1. Das Programm-Fragment, das nur aus der Zuweisung "`x := 1;`" besteht, erfüllt trivialerweise die Spezifikation

   $$\{\texttt{true}\} \quad \texttt{x := 1;} \quad \{x = 1\}.$$

   Hier ist die Vorbedingung die triviale Bedingung `true`, denn die Nachbedingung $x = 1$ ist in jedem Fall richtig.

2. Das Programm-Fragment, das aus der Zuweisung "`x = x + 1;`" besteht, erfüllt die Spezifikation

   $$\{x = 1\} \quad \texttt{x := x + 1;} \quad \{x = 2\}.$$

   Hier ist die $x = 1$ die Vorbedingung und die Nachbedingung ist $x = 2$.

3. Wir betrachten wieder as Programm-Fragment "`x = x + 1;`" und wählen als Vorbedingung diesmal die Formel $prim(x)$, die ausdrückt, dass $x$ eine Primzahl ist. Das entsprechende Hoare-Tripel ist diesmal

   $$\{prim(x)\} \quad \texttt{x := x + 1;} \quad \{prim(x-1)\}.$$

Auf den ersten Blick sieht das seltsam aus. Viele Studenten denken zunächst, dass dieses Hoare-Tripel die Form

$$\{prim(x)\} \quad \texttt{x := x + 1;} \quad \{prim(x+1)\}$$

haben müßte. Das diese letzte Zeile falsch ist, können wir sehen, wenn wir für $x$ den Wert 2 einsetzen, dann dann ist die Vorbedingung $prim(2)$ erfüllt. Nach der Ausführung der Zuweisung hat $x$ den Wert 3 und

$$x - 1 = 3 - 1 = 2$$

ist immer noch eine Primzahl, aber offenbar ist

$$x + 1 = 3 + 1 = 4 = 2 \cdot 2$$

keine Primzahl!

Wir überlegen uns nun, wie sich die verschiedenen Teile eines Programms mit Hilfe von Hoare-Tripeln spezifizieren lassen. Die elementarsten Bestandteile eines Programms sind die Zuweisungen. Wir fangen daher mit der Analyse der Zuweisungen an.

### 4.1.1 Spezifikation von Zuweisungen

Wir wollen das letzte Beispiel verallgemeinern und untersuchen, wie sich die Vor- und Nachbedingungen bei einer Zuweisung der Form

```
x := h(x);
```

zueinander verhalten. Konkret stellen wir uns die Frage, wie wir die Nachbedingung $G$ aus einer bereits bekannten Vorbedingung $F$ berechnen können. Zur Vereinfachung nehmen wir an, dass die Funktion $h$ umkehrbar ist, dass es also eine Umkehr-Funktion $h^{-1}$ gibt, so dass für alle $x$ gilt

$$h^{-1}\big(h(x)\big) = x \quad \texttt{und} \quad h\big(h^{-1}(x)\big) = x.$$

Um die Sache konkret zu machen, betrachten wir ein Beispiel. Die Zuweisung

```
x := x + 1;
```

können wir in der Form

```
x := h(x);
```

schreiben. Die Funktion $h$ hat dann die Form

$$h(x) = x + 1$$

und die Umkehr-Funktion ist offenbar

$$h^{-1}(x) = x - 1,$$

denn um die Addition von 1 rückgängig zu machen, müssen wir 1 subtrahieren. Mit der Umkehr-Funktion läßt sich nun die Nachbedingung der Zuweisung "`x := h(x);`" aus der Vorbedingung berechnen, denn es gilt

$$\big\{F\big\} \quad \texttt{x := h(x);} \quad \big\{F\sigma\big\} \quad \text{mit} \quad \sigma = \big[x \mapsto h^{-1}(x)\big].$$

Hier bezeichnet $F\sigma$ die Anwendung der Substitution $\sigma$ auf die Formel $F$: $F\sigma$ ist dadurch definiert, dass alle Auftreten der Variable $x$ durch den Term $h^{-1}(x)$ ersetzt werden. Um zu verstehen warum die Nachbedingung sich gerade so berechnet, greifen wir das Beispiel `x := x + 1` wieder auf und wählen als Vorbedingung $F$ die Formel $x = 7$. Wegen $h^{-1}(x) = x - 1$ hat die Substitution $\sigma$ die Form $\sigma = [x \mapsto x - 1]$ und damit hat $F\sigma$ die Form

$$(x = 7)[x \mapsto x - 1] \;\equiv\; (x - 1 = 7).$$

(An dieser Stelle habe ich das Zeichen "≡" benutzt, um die syntaktische Gleichheit von Formeln ausdrücken zu können, die ihrerseits das Gleichheitszeichen "=" enthalten.) Damit lautet die Spezifikation also

$$\big\{x = 7\big\} \quad \texttt{x := x + 1;} \quad \big\{x - 1 = 7\big\}.$$

Wegen der Äquivalenz $x - 1 = 7 \leftrightarrow x = 8$ ist das logisch das Selbe wie

$$\big\{x = 7\big\} \quad \texttt{x := x + 1;} \quad \big\{x = 8\big\}$$

und diese Spezifikation ist offenbar korrekt, denn wenn $x$ vor der Zuweisung "`x := x + 1;`" den Wert 7 hat, dann hat $x$ hinterher den Wert 8.

Wir überlegen uns nun, warum das Hoare-Tripel

$$\big\{F\big\} \quad \texttt{x := h(x);} \quad \big\{F\sigma\big\} \quad \text{mit} \quad \sigma = \big[x \mapsto h^{-1}(x)\big]$$

korrekt ist: Bevor die Zuweisung "x := h(x);" durchgeführt wird, hat die Variable $x$ einen festen Wert, den wir mit $x_0$ bezeichnen. Für diesen Wert $x_0$ gilt die Formel $F$, es gilt also vor der Zuweisung $F[x \mapsto x_0]$. In der Formel $F[x \mapsto x_0]$ tritt die Variable $x$ nicht mehr auf, denn wir haben diese Variable ja durch den Wert $x_0$ ersetzt. Damit bleibt die Formel

$$F[x \mapsto x_0]$$

auch nach der Zuweisung "`x = h(x);`" gültig. Nach der Zuweisung hat die Variable $x$ aber den Wert $x = h(x_0)$. Diese Gleichung lösen wir nach $x_0$ auf und finden

$$x_0 = h^{-1}(x).$$

Damit gilt also nach der Zuweisung die Formel

$$F[x \mapsto x_0] \equiv F[x \mapsto h^{-1}(x)].$$

Betrachten wir zum Abschluss dieser Diskussion ein weiteres Beispiel. Es sei *prim* ein einstelliges Prädikat, so dass $\mathit{prim}(x)$ genau dann wahr ist, wenn $x$ eine Primzahlen ist. Dann gilt

$$\big\{\texttt{prim}(x)\big\} \quad \texttt{x := x + 1;} \quad \big\{\texttt{prim}(x - 1)\big\}.$$

Anschaulich ist das klar: Wenn die Zahl $x$ eine Primzahl ist und $x$ um den Wert Eins inkrementiert wird, dann ist hinterher die Zahl $x - 1$ eine Primzahl.

**Andere Formen von Zuweisungen** Nicht immer haben Zuweisungen die Form "`x = h(x);`" mit einer invertierbaren Funktion $h$. Oft wird einer Variable $x$ eine Konstante $c$ zugewiesen. Falls die Variable $x$ in der Formel $F$ nicht vorkommt, gilt dann offenbar

$$\big\{F\big\} \quad \texttt{x := c;} \quad \big\{F \wedge x = c\big\}.$$

Die Formel $F$ kann hier dafür benutzt werden, um den Wert anderer Variablen einzugrenzen.

**Allgemeine Zuweisungs-Regel** In der Literatur findet sich die folgende Regel für Zuweisungen

$$\big\{F[x \mapsto t]\big\} \quad \texttt{x := t;} \quad \big\{F\big\}.$$

Hierbei ist $t$ ein beliebiger Term, der die Variable $x$ enthalten kann. Diese Regel ist wie folgt zu lesen:

> "*Gilt die Formel $F(t)$ und weisen wir der Variablen $x$ den Wert $t$ zu, so gilt danach die Formel $F(x)$.*"

Die Korrektheit dieser Regel ist offensichtlich, aber in der vorliegenden Form ist die Regel nicht sehr nützlich, denn um Sie anwenden zu können, muss die Vorbedingung erst auf die Form $F(t)$, also $F[x \mapsto t]$ gebracht werden.

### 4.1.2 Die Abschwächungs-Regel

Erfüllt ein Programm-Fragment $P$ die Spezifikation

$$\{F\} \quad \texttt{P} \quad \{G\}$$

und folgt aus der Formel $G$ die Formel $H$, gilt also

$$G \to H,$$

so erfüllt das Programm-Fragment $P$ erst recht die Spezifikation

$$\{F\} \quad \texttt{P} \quad \{H\},$$

denn wenn nach Ausführung von $P$ die Formel $G$ gilt, dann gilt die Formel $H$ erst recht. Wir fassen diesen Tatbestand in einer *Verifikations-Regel* zusammen:

$$\frac{\{F\} \quad \texttt{P} \quad \{G\}, \qquad G \to H}{\{F\} \quad \texttt{P} \quad \{H\}}.$$

Über dem Bruchstrich stehen hier die beiden *Prämissen* und unter dem Bruchstrich steht die *Konklusion*. Die Konklusion und die erste Prämisse sind Hoare-Tripel, die zweite Prämisse ist eine ganz normale logische Formel. Falls die Prämissen wahr sind, so gilt auch die Konklusion.

### 4.1.3 Zusammengesetze Anweisungen

Haben die Programm-Fragmente `P` und `Q` die Spezifikationen

$$\{F_1\} \quad \texttt{P} \quad \{G_1\} \quad \text{und} \quad \{F_2\} \quad \texttt{Q} \quad \{G_2\}$$

und folgt die Vorbedingung $F_2$ aus der Nach-Bedingung $G_1$, dann gilt für die Zusammensetzung `P;Q` von `P` und `Q` die Spezifikation

$$\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}$$

denn wenn anfangs $F_1$ gilt und zunächst $P$ ausgeführt wird, dann gilt danach $G_1$. Aus $G_1$ folgt $F_2$ und wenn das Programm-Fragment $Q$ in der Situation $F_2$ ausgeführt wird, dann wissen wir, dass danach $G_2$ gilt. Wir fassen diese Überlegungen in einer Verifikations-Regel zusammen:

$$\frac{\{F_1\} \quad \texttt{P} \quad \{G_1\}, \qquad G_1 \to F_2, \qquad \{F_2\} \quad \texttt{Q} \quad \{G_2\}}{\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}}$$

Gelegentlich ist es bei dieser Verifikations-Regel so, dass die Formeln $G_1$ und $F_2$ identisch sind. In diesem Fall ist die Implikation $G_1 \to F_2$ trivial und kann weggelassen werden. Dann nimmt die obige Verifikations-Regel die folgende vereinfachte Form an:

$$\frac{\{F_1\} \quad \texttt{P} \quad \{G_1\}, \qquad \{G_1\} \quad \texttt{Q} \quad \{G_2\}}{\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}}$$

```
1  x := x - y;
2  y := y + x;
3  x := y - x;
```

Abbildung 4.1: Eine trickreiche Implementierung zur Vertauschung zweier Werte.

**Beispiel:** Wir betrachten das in Abbildung 4.1 gezeigte Programm-Fragment. Wir analysieren nun die Wirkung dieses Programm-Fragments. Dazu starten wir unsere Analyse mit der Vorbedingung

$$\texttt{x} = a \wedge \texttt{y} = b.$$

Hier sind $a$ und $b$ zwei Variablen, in denen wir uns die Startwerte von `x` und `y` merken. Die erste Zuweisung liefert das Hoare-Tripel

$$\big\{\texttt{x} = a \wedge \texttt{y} = b\big\} \quad \texttt{x := x - y;} \quad \big\{(\texttt{x} = a \wedge \texttt{y} = b)\sigma\big\} \quad \text{mit} \quad \sigma = [x \mapsto x + y],$$

denn $x \mapsto x + y$ ist die Umkehr-Funktion von $x \mapsto x - y$. Führen wir die Substitution aus, so erhalten wir

$$\big\{\texttt{x} = a \wedge \texttt{y} = b\big\} \quad \texttt{x := x - y;} \quad \big\{\texttt{x + y} = a \wedge \texttt{y} = b\big\}. \tag{4.1}$$

Die zweite Zuweisung liefert das Hoare-Tripel

$$\big\{\texttt{x + y} = a \wedge \texttt{y} = b\big\} \quad \texttt{y := y + x;} \quad \big\{(\texttt{x + y} = a \wedge \texttt{y} = b)\sigma\big\} \quad \text{mit} \quad \sigma = [y \mapsto y - x],$$

denn $y \mapsto y - x$ ist die Umkehr-Funktion der Funktion $y \mapsto y + x$. Ausführung der Substitution liefert diesmal

$$\big\{\texttt{x + y} = a \wedge \texttt{y} = b\big\} \quad \texttt{y := y + x;} \quad \big\{\texttt{x + y - x} = a \wedge \texttt{y - x} = b\big\}.$$

Vereinfachen wir noch die Nachbedingung, so haben wir

$$\big\{\texttt{x + y} = a \wedge \texttt{y} = b\big\} \quad \texttt{y := y + x;} \quad \big\{\texttt{y} = a \wedge \texttt{y - x} = b\big\} \tag{4.2}$$

gezeigt. Jetzt betrachten wir die letzte Zuweisung. Wir erhalten

$$\big\{\texttt{y} = a \wedge \texttt{y - x} = b\big\} \quad \texttt{x := y - x;} \quad \big\{(\texttt{y} = a \wedge \texttt{y - x} = b)\sigma\big\} \quad \text{mit} \quad \sigma = [x \mapsto y - x],$$

denn $x \mapsto y - x$ ist die Umkehr-Funktion der Funktion $x \mapsto y - x$. Führen wir die Substitution aus, so ergibt sich

$$\big\{\texttt{y} = a \wedge \texttt{y - x} = b\big\} \quad \texttt{x := y - x;} \quad \big\{\texttt{y} = a \wedge \texttt{y - (y - x)} = b\big\}$$

Vereinfachung der Nachbedingung liefert schließlich

$$\big\{\texttt{y} = a \wedge \texttt{y - x} = b\big\} \quad \texttt{x := y - x;} \quad \big\{\texttt{y} = a \wedge \texttt{x} = b\big\}. \tag{4.3}$$

Fassen wir die Hoare-Tripel (4.1), (4.2) und (4.3) zusammen, so erhalten wir

$$\big\{\texttt{x} = a \wedge \texttt{y} = b\big\} \quad \texttt{x := x + y; y := y + x; x := y - x;} \quad \big\{\texttt{y} = a \wedge \texttt{x} = b\big\}. \tag{4.4}$$

Das Hoare-Tripel (4.4) zeigt, dass das Programm-Fragment aus Abbildung 4.1 die Werte der Variablen $x$ und $y$ vertauscht: Wenn vor der Ausführung dieses Programm-Fragments $x$ den Wert $a$ und $y$ den Wert $b$ hat, dann ist es nachher gerade umgekehrt: $y$ hat den Wert $a$ und $x$ hat den Wert $b$. Der in dem in Abbildung 4.1 gezeigte Trick wurde früher benutzt, um Werte zu vertauschen, denn bei der oben angegebenen Implementierung wird keine Hilfsvariable benötigt, so dass wir bei der Umsetzung dieses Programms in Assembler mit weniger Registern auskommen.

### 4.1.4 Alternativ-Anweisungen

Um die Wirkung einer Alternativ-Anweisung der Form

`if (`$B$`) P else Q`

zu berechnen, nehmen wir an, dass vor der Ausführung dieser Anweisung die Vorbedingung $F$ gilt. Dann müssen wir die Wirkung der Programm-Fragmente $P$ und $Q$ analysieren. Wenn $P$ ausgeführt wird, können wir neben $F$ noch die Bedingung $B$ annehmen, während wir für die Ausführung von $Q$ die zusätzliche Bedingung $\neg B$ annehmen können. Das liefert die folgende Verifikations-Regel:

$$\frac{\big\{F \wedge B\big\} \quad \texttt{P} \quad \big\{G\big\}, \qquad \big\{F \wedge \neg B\big\} \quad \texttt{Q} \quad \big\{G\big\}}{\big\{F\big\} \quad \texttt{if (}B\texttt{) P else Q} \quad \big\{G\big\}} \tag{4.5}$$

In der vorliegenden Form ist die Regel oft nicht unmittelbar anwendbar, denn die Analyse der Programm-Fragmente `P` und `Q` liefert zunächst Hoare-Tripel der Form

$$\big\{F \wedge B\big\} \quad \texttt{P} \quad \big\{G_1\big\} \qquad \text{und} \qquad \big\{F \wedge \neg B\big\} \quad \texttt{Q} \quad \big\{G_2\big\}, \tag{4.6}$$

wobei $G_1$ und $G_2$ zunächst verschieden sind. Um die obige Regel dann trotzdem anwenden zu können suchen wir eine Formel $G$, die sowohl aus $G_1$ als auch aus $G_2$ folgt, für die also

$$G_1 \rightarrow G \qquad \text{und} \qquad G_2 \rightarrow G$$

gilt. Haben wir eine solche Formel $G$ gefunden, dann können wir mit Hilfe der Abschwächungs-Regel (4.6) auf die Gültigkeit von

$$\big\{F \wedge B\big\} \quad \texttt{P} \quad \big\{G\big\} \qquad \text{und} \qquad \big\{F \wedge \neg B\big\} \quad \texttt{Q} \quad \big\{G\big\},$$

schließen und damit haben wir genau die Prämissen, um die Verifikations-Regel (4.5) anwenden zu können.

**Beispiel**: Wir analysieren das folgende Programm-Fragment:

```
if (x < y) { z := x; } else { z := y; }
```

Wir starten mit der Vorbedingung

$$F = \big(\texttt{x} = a \wedge \texttt{y} = b\big)$$

und wollen zeigen, dass nach Ausführung der obigen Alternativ-Anweisung die Nachbedingung

$$G = \big(z = \mathit{min}(a, b)\big)$$

gültig ist. Für die erste Zuweisung erhalten wir das Hoare-Tripel

$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y}\big\} \quad \texttt{z := x;} \quad \big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y} \wedge \texttt{z} = \texttt{x}\big\}.$$

Analog erhalten wir für die zweite Zuweisung

$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y}\big\} \quad \texttt{z := y;} \quad \big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y} \wedge \texttt{z} = \texttt{y}\big\}.$$

Nun gilt einerseits

$$\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y} \wedge \texttt{z} = \texttt{x} \rightarrow \texttt{z} = \min(a, b)$$

und andererseits

$$\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y} \wedge \texttt{z} = \texttt{y} \rightarrow \texttt{z} = \min(a, b).$$

Durch Anwendung der Abschwächungs-Regel sehen wir also, dass

$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y}\big\} \quad \texttt{z := x;} \quad \big\{\texttt{z} = \min(a, b)\big\} \qquad \text{und}$$
$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y}\big\} \quad \texttt{z := y;} \quad \big\{\texttt{z} = \min(a, b)\big\}$$

gilt. Durch Anwendung der Verifikations-Regel für die Alternativ-Anweisung folgt dann, dass

$$\big\{\texttt{x} = a \wedge \texttt{y} = b\big\} \quad \texttt{if (x < y) \{ z := x; \} else \{ z := y; \}} \quad \big\{\texttt{z} = \min(a, b)\big\}$$

gilt und damit ist nachgewiesen, dass das entsprechende Programm-Fragment das Minimum der Zahlen $a$ und $b$ berechnet.

### 4.1.5 Schleifen

Als letztes analysieren wir die Wirkung einer Schleife der Form

`while (`$B$`) { P }`

Der entscheidende Punkt ist hier, dass die Nachbedingung für den $n$-ten Schleifen-Durchlauf gleichzeitig die Vorbedingung für den $(n+1)$-ten Schleifen-Durchlauf sein muss. Das führt zu der Forderung, dass Vor- und Nachbedingung für den Schleifen-Rumpf im wesentlichen identisch sein müssen. Diese Bedingung trägt daher den Namen *Schleifen-Invariante*. Im Detail hat die Verifikations-Regel die folgende Form:

$$\frac{\{I \wedge B\} \quad \texttt{P} \quad \{I\}}{\{I\} \quad \texttt{while (}B\texttt{) \{ P \}} \quad \{I \wedge \neg B\}}$$

Die Prämisse dieser Regel besagt, dass bei der Ausführung von $P$ die Invariante $I$ wahr bleiben muss, wobei wir als zusätzliche Vorbedingung noch die Formel $B$ annehmen dürfen, denn `P` wird nur ausgeführt, wenn $B$ wahr ist. Die Konklusion besagt, dass wenn vor der Ausführung der Schleife die Invariante $I$ gültig ist, dann gilt sie hinterher immer noch. Anschaulich ist das klar, denn jede einzelne Ausführung des Schleifen-Rumpfs `P` hat die Invariante $I$ ja erhalten. Zusätzlich wissen wir noch, dass nach Beendigung der Schleife die Bedingung $\neg B$ gilt, denn wenn $B$ gelten würde, würde die Schleife weiterlaufen.

## 4.2 Der Euklid'sche Algorithmus

Wir zeigen nun, wie die im letzten Abschnitt präsentierten Regeln verwendet werden können, um die Korrektheit eines nicht-trivialen Programms zu zeigen. Unser Ziel ist es, die Korrektheit der in Abbildung 4.2 auf Seite 34 gezeigten `C`-Funktion nachzuweisen. Diese Funktion implementiert den Euklid'schen Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier positiver natürlicher Zahlen. Bevor wir für diesen Algorithmus den Nachweis der Korrektheit erbringen, erinnern wir daran, dass wir in der Mathematik-Vorlesung gezeigt haben dass

$$\texttt{ggt}(x+y, y) = \texttt{ggt}(x, y) \quad \text{für alle } x, y \in \mathbb{N}$$

gilt.

### 4.2.1 Nachweis der Korrektheit des Euklid'schen Algorithmus

Wir haben nun alles Material zusammen, um die Korrektheit des Euklid'schen Algorithmus, der in Abbildung 4.2 gezeigt wird, nachweisen zu können.

```
1  ggt := procedure(x, y) {
2      while (x != y) {
3          if (x < y) {
4              y := y - x;
5          } else {
6              x := x - y;
7          }
8      }
9      return x;
10 };
```

Abbildung 4.2: Der Euklid'sche Algorithmus zur Berechnung des größten gemeinsamen Teilers.

Wir beginnen den Nachweis der Korrektheit damit, dass wir die Invariante formulieren, die von der `while`-Schleife erhalten wird. Wir definieren

$$I := \big(x > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b)\big).$$

Hierbei haben wir die Startwerte von $x$ und $y$ mit $a$ und $b$ bezeichnet. Um die Invariante $I$ zu Beginn sicherzustellen, ist zu fordern, dass die Funktion `ggt` nur mit positiven ganzen Zahlen aufgerufen wird. Bezeichnen wir diese Zahlen mit $a$ und $b$, so gilt die Invariante offenbar zu Beginn, denn aus $x = a$ und $y = b$ folgt sofort $\texttt{ggt}(x,y) = \texttt{ggt}(a,b)$.

Um nachzuweisen, dass die Invariante $I$ in der Schleife erhalten bleibt, stellen wir für die beiden Alternativen der `if`-Abfrage jeweils ein Hoare-Tripel auf. Für die erste Alternative wissen wir, dass

$$\big\{I \wedge x \neq y \wedge x < y\big\} \quad \texttt{y := y - x;} \quad \big\{(I \wedge x \neq y \wedge x < y)\sigma\big\} \quad \text{mit} \quad \sigma = [y \mapsto y + x]$$

gilt. Die Bedingung $x \neq y$ ist dabei die Bedingung der `while`-Schleife und die Bedingung $x < y$ ist die Bedingung der `if`-Abfrage. Wir formen den Ausdruck $(I \wedge x \neq y \wedge x < y)\sigma$ um:

$$\begin{array}{ll}
 & \big(I \wedge x \neq y \wedge x < y\big)\sigma \\
\leftrightarrow & \big(I \wedge x < y\big)\sigma \qquad\qquad \text{denn aus } x < y \text{ folgt } x \neq y \\
\leftrightarrow & \big(x > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b) \wedge x < y\big)[y \mapsto y + x] \\
\leftrightarrow & x > 0 \wedge y + x > 0 \wedge \texttt{ggt}(x, y + x) = \texttt{ggt}(a,b) \wedge x < y + x \\
\leftrightarrow & x > 0 \wedge y + x > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b) \wedge 0 < y
\end{array}$$

Dabei haben wir bei der letzten Umformung die im letzten Abschnitt bewiesene Gleichung

$$\texttt{ggt}(x, y + x) = \texttt{ggt}(x,y)$$

benutzt und die Ungleichung $x < y + x$ zu $0 < y$ vereinfacht. Aus der letzten Formel folgt offenbar

$$x > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b).$$

Das ist aber genau unsere Invariante $I$. Damit haben wir gezeigt, dass

$$\big\{I \wedge x \neq y \wedge x < y\big\} \quad \texttt{y := y - x;} \quad \big\{I\big\} \tag{4.7}$$

gilt. Nun betrachten wir die zweite Alternative der `if`-Abfrage. Offenbar gilt

$$\big\{I \wedge x \neq y \wedge x \geq y\big\} \quad \texttt{x := x - y;} \quad \big\{(I \wedge x \neq y \wedge x \geq y)\sigma\big\} \quad \text{mit} \quad \sigma = [x \mapsto x + y].$$

Wir formen den Ausdruck $(I \wedge x \neq y \wedge x \geq y)\sigma$ um:

$$\begin{array}{ll}
 & \big(I \wedge x \neq y \wedge x \geq y\big)\sigma \\
\leftrightarrow & \big(I \wedge x > y\big)\sigma \\
\leftrightarrow & \big(x > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b) \wedge x > y\big)[x \mapsto x + y] \\
\leftrightarrow & x + y > 0 \wedge y > 0 \wedge \texttt{ggt}(x + y, y) = \texttt{ggt}(a,b) \wedge x + y > y \\
\leftrightarrow & x + y > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b) \wedge x > 0
\end{array}$$

Aus der letzten Formel folgt nun

$$x > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b).$$

Das ist aber wieder unsere Invariante $I$. Also haben wir insgesamt

$$\big\{I \wedge x \neq y \wedge x \geq y\big\} \quad \texttt{x := x - y;} \quad \big\{I\big\} \tag{4.8}$$

gezeigt. Aus den beiden Hoare-Tripeln (4.7) und (4.8) folgt nun mit der Regel für Alternativen die Gültigkeit von

$$\big\{I \wedge x \neq y\big\} \quad \texttt{if (x < y) \{ x := x - y; \} else \{ y := y - x;\}} \quad \big\{I\big\}.$$

Mit der Verifikations-Regel für Schleifen folgt daraus

$\{I\}$

```
while (x != y ) {
    if (x < y) { x := x - y; } else { y := y - x;}
}
```

$\{I \wedge x = y\}$.

Schreiben wir die Formel $I \wedge x = y$ aus, so sehen wir, dass die Nachbedingung der `while`-Schleife durch die Formel

$$x > 0 \wedge y > 0 \wedge \texttt{ggt}(x, y) = \texttt{ggt}(a, b) \wedge x = y$$

gegeben ist. Daraus erschließen wir die Korrektheit des Euklid'schen Algorithmus wie folgt:

$$\begin{array}{lll} & x > 0 \wedge y > 0 \wedge \texttt{ggt}(x, y) = \texttt{ggt}(a, b) \wedge x = y & \\ \Rightarrow & \texttt{ggt}(x, y) = \texttt{ggt}(a, b) \wedge x = y & \\ \Rightarrow & \texttt{ggt}(x, x) = \texttt{ggt}(a, b) & \\ \Rightarrow & x = \texttt{ggt}(a, b) & \text{denn} \quad \texttt{ggt}(x, x) = x. \end{array}$$

Damit haben wir folgendes gezeigt: Wenn die `while`-Schleife terminiert, dann enthält die Variable $x$ den größten gemeinsamen Teiler der Werte $a$ und $b$, mit denen die Variablen $x$ und $y$ initialisiert wurden. Um den Beweis der Korrektheit abzuschließen, müssen wir noch nachweisen, dass die `while`-Schleife tatsächlich in jedem Fall terminiert. Zu diesem Zweck definieren wir die Variable $s$ als

$$s := x + y.$$

Die Variablen $x$ und $y$ sind natürliche Zahlen. Damit ist dann auch $s$ eine natürliche Zahl. Bei jedem Schleifendurchlauf wird die Zahl $s$ verkleinert, denn entweder wird $x$ von $s$ abgezogen oder es wird $y$ von $s$ abgezogen und die Invariante $I$ zeigt uns, dass sowohl $x$ als auch $y$ positiv sind. Würde die Schleife unendlich lange laufen, so müßte $s$ irgendwann negative Werte annehmen. Da wir dies ausschließen können, muss die Schleife abbrechen. Damit ist die Korrektheit des Euklid'schen Algorithmus gezeigt.

**Aufgabe**: Zeigen Sie, dass der Aufruf `power`$(x, y)$ der in Abbildung 4.3 gezeigten Funktion `power`$(x, y)$ für gegebene natürliche Zahlen $x$ und $y$ die Potenz $x^y$ berechnet.

```
1   power := procedure(x, y) {
2       r := 1;
3       while (y > 0) {
4           if (y % 2 == 1) {
5               r := r * x;
6           }
7           x := x * x;
8           y := y / 2;
9       }
10      return r;
11  };
```

Abbildung 4.3: Ein SetlX-Programm zur Berechnung der Potenz.

**Hinweise**:

1. Bezeichnen wir die Start-Werte von $x$ und $y$ mit $a$ und $b$, so ist eine mögliche Invariante für die `while`-Schleife durch die Formel

   $$I := \left(r \cdot x^y = a^b\right)$$

   gegeben.

2. Die Verifikations-Regel für die *einarmige Alternative* lautet

   $$\frac{\{F \wedge B\}\ \texttt{P}\ \{G\}, \qquad F \wedge \neg B \rightarrow G}{\{F\}\ \texttt{if (}B\texttt{) \{ P \}}\ \{G\}}$$

   Diese Verifikations-Regel kann wie folgt interpretiert werden:

   (a) Falls $F$ und die Bedingung $B$ gilt, dann muss die Ausführung von dem Programm-Fragment $P$ bewirken, dass nachher $G$ gilt.

   (b) Falls $F$ und die Bedingung $\neg B$ gilt, dann muss daraus die Bedingung $G$ folgen.

   (c) Unter diesen Bedingungen folgt dann aus der Vorbedingung $F$ nach Ausführung von dem Programm-Fragment "`if (`$B$`) { P }`" die Nachbedingung $G$.

**Lösung**: Wir untersuchen Zeile für Zeile die Wirkung des Programms.

1. Wir bezeichnen den Anfangswert von $x$ mit $a$ und den Anfangswert von $y$ mit $b$. Damit lautet die Vorbedingung der Anweisung "`r = 1;`"

   $$x = a \wedge y = b$$

   Damit finden wir für die erste Anweisung das Hoare-Tripel

   $$\{x = a \wedge y = b\} \quad \texttt{r := 1;} \quad \{x = a \wedge y = b \wedge r = 1\}.$$

2. Als nächstes müssen wir zeigen, dass aus der Nachbedingung der ersten Anweisung die oben angegebene Invariante $I$ folgt. Offenbar gilt

   $$x = a \wedge y = b \wedge r = 1 \rightarrow r \cdot x^y = a^b.$$

3. Wir untersuchen nun die `if`-Abfrage. Als Vorbedingung der `if`-Abfrage nehmen wir die Invariante $I$ zusammen mit der Bedingung $y > 0$. Für die Zuweisung "`r = r * x;`" erhalten wir dann die Bedingung $I \wedge y > 0 \wedge y \,\%\, 2 = 1$. Das liefert das Hoare-Tripel

   $$\{r \cdot x^y = a^b \wedge y > 0 \wedge y \,\%\, 2 = 1\}$$

   ```
   r := r * x;
   ```

   $$\{\big(r \cdot x^y = a^b \wedge y > 0 \wedge y \,\%\, 2 = 1\big)[r \mapsto r/x]\}$$

   Die Nachbedingung vereinfacht sich zu

   $$r/x \cdot x^y = a^b \wedge y > 0 \wedge y \,\%\, 2 = 1$$

   und das ist das Selbe wie

   $$r \cdot x^{y-1} = a^b \wedge y > 0 \wedge y \,\%\, 2 = 1.$$

   Um dies weiter vereinfachen zu können, schreiben wir

   $$y = 2 \cdot (y/2) + y \,\%\, 2.$$

   Setzen wir diesen Wert von $y$ in der letzten Gleichung ein, so erhalten wir

   $$r \cdot x^{2\cdot(y/2)} = a^b \wedge y > 0 \wedge y \,\%\, 2 = 1,$$

   denn aus $y \,\%\, 2 = 1$ folgt $y - 1 = 2 \cdot (y/2)$. Damit haben wir die Gültigkeit von

   $$\{r \cdot x^y = a^b \wedge y > 0 \wedge y \,\%\, 2 = 1\} \quad \texttt{r := r * x;} \quad \{r \cdot x^{2\cdot(y/2)} = a^b\}$$

   bewiesen. Wir versuchen nun, unter der Bedingung $y \,\%\, 2 = 0$ die Invariante $I$ so umzuformen, dass sie mit der Nachbedingung dieses Hoare-Tripels übereinstimmt. Es gilt

   $$r \cdot x^y = a^b \wedge y > 0 \wedge y \,\%\, 2 = 0 \rightarrow r \cdot x^{2\cdot(y/2)} = a^b,$$

   denn aus $y \,\%\, 2 = 0$ folgt $y = 2 \cdot (y/2)$. Damit haben wir insgesamt für die `if`-Abfrage das folgende Hoare-Tripel gefunden:

   $$\{r \cdot x^y = a^b \wedge y > 0\}$$

   ```
   if (y % 2) == 1 { r := r * x;}
   ```

   $$\{r \cdot x^{2\cdot(y/2)} = a^b\}$$

4. Als nächstes untersuchen wir die Zuweisung "`x := x * x;`". Als Vorbedingung nehmen wir natürlich die Nachbedingung der `if`-Abfrage. Wir erhalten das Hoare-Tripel

   $$\{r \cdot x^{2\cdot(y/2)} = a^b\} \quad \texttt{x := x * x;} \quad \{\big(r \cdot x^{2\cdot(y/2)} = a^b\big)\big[x \mapsto x^{\frac{1}{2}}\big]\}$$

   Führen wir die Substitution aus, so vereinfacht sich die Nachbedingung zu

   $$r \cdot (x^{\frac{1}{2}})^{2\cdot(y/2)} = a^b$$

   und dies kann weiter vereinfacht werden zu

$$r \cdot x^{y/2} = a^b$$

Damit haben wir also insgesamt das Hoare-Tripel

$$\{r \cdot x^{2\cdot(y/2)} = a^b\} \quad \texttt{x := x * x;} \quad \{r \cdot x^{y/2} = a^b\}$$

gefunden.

5. Die letzte Zuweisung "`y := y / 2;`" liefert nun das Hoare-Tripel

   $$\{r \cdot x^{y/2} = a^b\} \quad \texttt{y := y / 2;} \quad \{r \cdot x^{y} = a^b\}.$$

   Hier haben wir die allgemeine Zuweisungs-Regel

   $$\{F[y \mapsto t]\} \quad \texttt{y := t;} \quad \{F\}$$

   benutzt, wobei wir für $F$ die Formel $r \cdot x^y = a^b$ und für $t$ den Term $y/2$ verwendet haben.

   **Bemerkung**: Es ist hier nicht möglich, die Zuweisungs-Regel

   $$\{F\} \quad \texttt{x := h(x);} \quad \{F\sigma\} \quad \text{mit} \quad \sigma = \big[x \mapsto h^{-1}(x)\big]$$

   zu verwenden, denn die Funktion $y \mapsto y/2$ ist nicht umkehrbar, da sowohl $2/2 = 1$ als auch $3/2 = 1$ gilt.

6. Da die Nachbedingung des letzten Hoare-Tripels genau die Schleifen-Invariante $I$ ist, haben wir insgesamt die Korrektheit des folgenden Hoare-Tripels gezeigt:

   $\{r \cdot x^y = a^b\}$

   ```
   while (y > 0) {
      if (y % 2 == 1) { r = r * x; }
      x = x * x;
      y = y / 2;
   }
   ```

   $\{r \cdot x^y = a^b \wedge \neg y > 0\}$

   Da $y$ eine natürlich Zahl sein muss, denn $y$ wird ja nur durch die Ganzzahl-Divison

   $$y = y/2$$

   verändert, folgt aus $\neg y > 0$, dass $y = 0$ gilt. Damit lautet die Nachbedingung der `while`-Schleife also

   $$r \cdot x^y = a^b \wedge y = 0,$$

   was sich wegen $x^0 = 1$ zu

   $$r = a^b,$$

   vereinfacht. Insgesamt haben wir also gezeigt, dass die Variable $r$ am Ende der `while`-Schleife den Wert $a^b$ hat.

Es bleibt noch zu zeigen, dass die `while`-Schleife immer terminiert. Einerseits wird die Variable `y` in jedem Schritt ganzzahlig durch zwei geteilt, andererseits läuft die Schleife nur solange, wie die Variable `y` positiv ist. Daher muss die Schleife abbrechen. □

### 4.2.2 Maschinelle Programm-Verifikation

Die manuelle Verifikation nicht-trivialer Programm mit Hilfe des Hoare-Kalküls ist sehr aufwendig. Von Hand können nur Programme verifiziert werden, die in der selben Größenordnung liegen, wie der oben behandelte Euklid'sche Algorithmus. Es ist aber möglich, den Prozess der Programm-Verifikation zumindest partiell zu automatisieren. Sogenannte VCGs (*verification condition generators*) reduzieren die Verifikation eines Programms auf den Nachweis bestimmter logischer Formeln, die als *Verifikations-Bedingungen* bezeichnet werden. Die Verifikations-Bedingungen können dann mit

der Unterstützung automatischer Beweise nachgewiesen werden. Auch dieses Vorgehen ist nur für Programme mittlerer Komplexität praktikabel. Im Internet finden Sie unter der Adresse

`http://www.mathematik.uni-marburg.de/~gumm/NPPV/JavaProgramVerifierII.zip`

das System JPV (Java program verifier), mit dessen Hilfe Programme verifiziert werden können. Das dort zur Verfügung gestellten Systeme ist nicht darauf ausgelegt, umfangreiche Programme zu verifizieren, es reicht aber aus, um einen Eindruck in die Technik zu vermitteln. Wir demonstrieren das System am Beispiel der Verifikation des Programms zur Berechnung der Potenz. Die Benutzer-Eingabe hat in diesem Fall die in Abbildung 4.4 gezeigte Form.

1. Zeile 1 enthält, eingefasst in den Zeichen "`##`", die Vorbedingung des Programms. Hier wird ausgesagt, dass die Variablen $x$ und $y$ zu Beginn zwei Zahlen $a$ und $b$. Beim Beweis werden wir hinterher davon ausgehen, dass es sich bei $a$ und $b$ um natürliche Zahlen handelt.

2. Hinter der `while`-Schleife formulieren wir in Zeile 3 die Invariante der Schleife:

   $$r \cdot x^y = a^b.$$

   Da der Potenz-Operator in JPV nicht zur Verfügung steht, haben wir für $x^y$ den Ausdruck $\texttt{pow}(x, y)$ verwendet.

3. Die letzte Zeile enthält die Nachbedingung des Programm-Fragments. Hier wird ausgedrückt, dass am Ende der Rechnung $r = a^b$ gilt.

```
1   ## x == a & y == b & b > 0 ##
2     r = 1;
3     while(y > 0) ## r * pow(x,y) == pow(a,b) ##
4     {
5        if (y % 2 == 1) {
6           r = r * x;
7        }
8        x = x * x;
9        y = y / 2;
10    }
11  ## r == pow(a,b) ##
```

Abbildung 4.4: Verifikation der Berechnung der Potenz mit JPV.

Wir können nun JPV benutzen, um nachzuweisen, dass das oben gezeigte Programm-Fragment die annotierte Spezifikation erfüllt. Starten wir das System mit Hilfe des Befehls

```
java -jar JPV.jar
```

so werden die folgenden Verifikations-Bedingungen erzeugt:

1. $x = a \wedge y = b \rightarrow 1 \cdot x^y = a^b$.
2. $r \cdot x^y = a^b \rightarrow r \cdot x^y = a^b$.
3. $y > 0 \wedge r \cdot x^y = a^b \wedge y \,\%\, 2 = 1 \rightarrow r \cdot x \cdot (x \cdot x)^{y/2} = a^b$.
4. $y > 0 \wedge r \cdot x^y = a^b \wedge y \,\%\, 2 \neq 1 \rightarrow r \cdot (x \cdot x)^{y/2} = a^b$.
5. $r \cdot (x \cdot x)^{y/2} = a^b \rightarrow r \cdot (x \cdot x)^{y/2} = a^b$.
6. $r \cdot x^{y/2} = a^b \rightarrow r \cdot x^{y/2} = a^b$.
7. $r \cdot x^y = a^b \wedge y \leq 0 \rightarrow r = a^b$.

Der in JPV integrierte automatische Beweiser ist in der Lage, die 1., die 2., die 5. und die 6. dieser Verifikations-Bedingungen unmittelbar nachzuweisen. Der Nachweis der Korrektheit der restlichen Bedingungen ist dann vom Benutzer mit Papier und Bleistift zu erbringen. Da wir die entsprechenden Beweise im wesentlichen schon beim Nachweis der Korrektheit des in Abbildung 4.3 gezeigten Programms geführt haben, wiederholen wir sie hier nicht noch einmal.

## 4.3 Symbolische Programm-Ausführung

Wir haben im letzten Abschnitt gesehen, dass der Hoare-Kalkül sehr schwerfällig ist. Es gibt noch eine weitere Methode, um die Korrektheit eines sequentiellen Programms nachzuweisen. Dies ist die Methode der *symbolischen Programm-Ausführung*. Wir demonstrieren diese Methode an Hand des in Abbildung 4.3 gezeigten Programms zur iterativen Berechnung der Potenz.

```
1   power := procedure(x0, y0) {
2       r0 := 1;
3       while (yn > 0) {
4           if (yn % 2 == 1) {
5               rn+1 := rn * xn;
6           }
7           xn+1 := xn * xn;
8           yn+1 := yn / 2;
9       }
10      return rN;
11  };
```

Abbildung 4.5: Das indizierte C-Programm zur Berechnung der Potenz.

Der wesentliche Unterschied zwischen einer mathematischen Formel und einen Programm ist der, dass alle Auftreten einer Variablen in einer mathematischen Formel den selben Wert bezeichnen. In einem Programm ist dies anders, denn die Werte einer Variablen ändern sich dynamisch. Um diesem Verhalten Rechnung zu tragen, müssen wir die verschiedenen Auftreten einer Formel unterscheiden können. Dies geht am einfachsten, wenn wir die Variablen so indizieren, dass der Index der Variablen sich jedesmal ändert, wenn die Variable einen neuen Wert zugewiesen bekommt. Dabei müssen wir allerdings berücksichtigen, dass ein und das selbe textuelle Auftreten einer Variablen immer noch verschiedene Werte annehmen kann und zwar dann, wenn das Auftreten in einer Schleife liegt. In diesem Fall müssen wir die Variable so indizieren, dass noch ein Zähler für die Anzahl der Schleifen-Durchläufe in dem Index eingebaut ist. Um die Diskussion nicht zu abstrakt werden zu lassen, betrachten wir das Beispiel in Abbildung 4.5. Hier hat die Variable `r` auf der rechten Seite der Zuweisung in Zeile 6 den Wert $\mathtt{r}_n$, auf der linken Seite hat sie dann anschließend den Wert $\mathtt{r}_{n+1}$ und am Ende der Schleife in Zeile 11 hat die Variable `r` den Wert $\mathtt{r}_N$, wobei $N$ die Anzahl der Schleifen-Durchläufe angibt.

Wir beweisen nun die Korrektheit des abgebildeten Programms. Wir definieren

$$a := x_0, \quad b := y_0$$

und zeigen, dass für die `while`-Schleife die Invariante

$$r_n \cdot x_n^{y_n} = a^b \tag{4.9}$$

gilt. Diese Behauptung beweisen wir durch eine Induktion nach der Anzahl der Schleifen-Durchläufe.

I.A. $n = 0$: Wegen $r_0 = 1$ und $x_0 = a$ sowie $y_0 = b$ gilt für $n = 0$

$$r_n \cdot x_n^{y_n} = r_0 \cdot x_0^{y_0} = 1 \cdot a^b = a^b.$$

I.S. $n \mapsto n+1$: Wir führen eine Fallunterscheidung nach dem Wert von $y$ `%` 2 durch:

(a) $y_n$ `%` $2 = 1$. Dann gilt $y_n = 2 \cdot (y_n/2) + 1$. Damit finden wir

$$\begin{aligned}
& r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\
= \quad & (r_n \cdot x_n) \cdot (x_n \cdot x_n)^{y_n/2} \\
= \quad & r_n \cdot x_n \cdot x_n^{y_n/2} \cdot x_n^{y_n/2} \\
= \quad & r_n \cdot x_n^{1+y_n/2+y_n/2} \\
= \quad & r_n \cdot x_n^{2\cdot(y_n/2)+1} \\
= \quad & r_n \cdot x_n^{y_n} \\
\overset{IV}{=} \quad & a^b
\end{aligned}$$

(b) $y_n$ `%` $2 = 0$. Dann gilt $y_n = 2 \cdot (y_n/2)$. Damit finden wir

$$\begin{aligned}
& r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\
= \quad & r_n \cdot (x_n \cdot x_n)^{y_n/2} \\
= \quad & r_n \cdot x_n^{y_n/2} \cdot x_n^{y_n/2} \\
= \quad & r_n \cdot x_n^{y_n/2+y_n/2} \\
= \quad & r_n \cdot x_n^{2\cdot(y_n/2)} \\
= \quad & r_n \cdot x_n^{y_n} \\
\overset{IV}{=} \quad & a^b
\end{aligned}$$

Damit ist die Gleichung (4.9) bewiesen. Wenn die `while`-Schleife abbricht, dann muss $y_N = 0$ gelten. Gleichung (4.9) liefert für $n = N$:

$$r_N \cdot x_N^{y_N} = x_0^{y_0} \leftrightarrow r_N \cdot x_N^0 = a^b \leftrightarrow r_N \cdot 1 = a^b \leftrightarrow r_N = a^b$$

Damit haben wir insgesamt $r_N = a^b$ bewiesen und da wir schon wissen, dass die `while`-Schleife terminiert, ist damit gezeigt, dass der Funktions-Aufruf `power`$(a, b)$ tatsächlich den Wert $a^b$ berechnet.

**Aufgabe**: Weisen Sie mit der Methode der symbolischen Programm-Ausführung die Korrektheit der in Abbildung 4.6 gezeigten effizienteren Version des Euklid'schen Algorithmus nach. Sie dürfen dabei benutzen, dass für positive natürliche Zahlen $a$ und $b$ die Beziehung

$$\texttt{ggt}(a, b) = \texttt{ggt}(a \,\texttt{\%}\, b, b)$$

erfüllt ist.

```
1  ggt := procedure(a, b) {
2      while (b != 0) {
3          r := a % b;
4          a := b;
5          b := r;
6      }
7      return a;
8  };
```

Abbildung 4.6: Der Euklid'sche Algorithmus zur Berechnung des größten gemeinsamen Teilers.

# Kapitel 5

# Sortier-Algorithmen

Im Folgenden gehen wir davon aus, dass wir eine Liste $L$ gegeben haben, deren Elemente aus einer Menge $M$ entstammen. Die Elemente von M können wir *vergleichen*, das heißt, dass es auf der Menge $M$ eine Relation $\leq$ gibt, die *reflexiv*, *anti-symmetrisch* und *transitiv* ist, es gilt also

1. $\forall x \in M: x \leq x$.
2. $\forall x, y \in M: \big(x \leq y \wedge y \leq x \rightarrow x = y\big)$.
3. $\forall x, y, z \in M: \big(x \leq y \wedge y \leq z \rightarrow x \leq z\big)$.

Ein Paar $\langle M, \leq \rangle$ bestehend aus einer Menge und einer binären Relation $\leq \subseteq M \times M$ mit diesen Eigenschaften bezeichnen wir als eine *partielle Ordnung*. Gilt zusätzlich

$$\forall x, y \in M: \big(x \leq y \vee y \leq x\big),$$

so bezeichnen wir $\langle M, \leq \rangle$ als eine *totale Ordnung*.

**Beispiele**:

1. $\langle \mathbb{N}, \leq \rangle$ ist eine totale Ordnung.
2. $\langle 2^{\mathbb{N}}, \subseteq \rangle$ ist eine partielle Ordnung aber keine totale Ordnung, denn beispielsweise sind die Mengen $\{1\}$ und $\{2\}$ nicht vergleichbar, es gilt

   $$\{1\} \not\subseteq \{2\} \text{ und } \{2\} \not\subseteq \{1\}.$$

3. Ist $P$ die Menge der Mitarbeiter einer Firma und definieren wir für zwei Mitarbeiter $a, b \in P$

   $$a < b \quad \text{g.d.w.} \quad a \text{ verdient weniger als } b,$$

   so ist $\langle P, \leq \rangle$ eine partielle Ordnung.

**Bemerkung**: In dem letzten Beispiel haben wir anstelle der Relation $\leq$ die Relation $<$ definiert. Ist eine Relation $<$ gegeben, so ist die dazugehörige Relation $\leq$ wie folgt definiert:

$$x \leq y \stackrel{\text{def}}{\iff} x < y \vee x = y.$$

Betrachten wir die obigen Beispiele und überlegen uns, in welchen Fällen es möglich ist, eine Liste von Elemente zu sortieren, so stellen wir fest, dass dies im ersten und dritten Fall möglich ist, im zweiten Fall aber keinen Sinn macht. Offensichtlich ist eine totale Ordnung hinreichend zum Sortieren aber, wie das dritte Beispiel zeigt, nicht unbedingt notwendig. Eine partielle Ordnung reicht hingegen zum Sortieren nicht aus. Wir führen daher einen weiteren Ordnungs-Begriff ein.

**Definition 13 (Quasi-Ordnung)**
*Ein Paar $\langle M, \preceq \rangle$ ist eine* Quasi-Ordnung*, falls $\preceq$ eine binäre Relation auf $M$ ist, für die gilt:*

1. $\forall x \in M: x \preceq x$. *(Reflexivität)*
2. $\forall x, y, z \in M: \big(x \preceq y \wedge y \preceq z \rightarrow x \preceq z\big)$. *(Transitivität)*

*Gilt zusätzlich*

$$\forall x, y \in M: \big(x \preceq y \vee y \preceq x\big),$$

*so bezeichnen wir* $\langle M, \preceq \rangle$ *als eine* totale Quasi-Ordnung*, was wir als* TQO *abkürzen.*

Bei dem Begriff der Quasi-Ordnung wird im Unterschied zu dem Begriff der partiellen Ordnung auf die Eigenschaft der Anti-Symmetrie verzichtet. Trotzdem sind die Begriffe fast gleichwertig, denn wenn $\langle M, \preceq \rangle$ eine Quasi-Ordnung ist, so kann auf $M$ eine Äquivalenz-Relation $\approx$ durch

$$x \approx y \stackrel{\text{def}}{\longleftrightarrow} x \preceq y \wedge y \preceq x$$

definiert werden. Setzen wir die Ordnung $\preceq$ auf die von der Relation $\approx$ erzeugten Äquivalenz-Klassen fort, so kann gezeigt werden, dass diese Fortsetzung eine partielle Ordnung ist.

Es sei nun $\langle M, \preceq \rangle$ eine TQO. Dann ist das *Sortier-Problem* wie folgt definiert:

1. Gegeben ist eine Liste $L$ von Elementen aus M.
2. Gesucht ist eine Liste $S$ mit folgenden Eigenschaften:
   (a) $S$ ist aufsteigend sortiert:
   $$\forall i \in \{1, \cdots, \#S - 1\}: S(i) \preceq S(i+1)$$
   Hier bezeichnen wir die Länge der Liste $S$ mit $\#S$.
   (b) Die Elemente treten in $L$ und $S$ mit der selben Häufigkeit auf:
   $$\forall x \in M: \mathit{count}(x, L) = \mathit{count}(x, S).$$
   Dabei zählt die Funktion $\mathit{count}(x, L)$ wie oft das Element $x$ in der Liste $L$ auftritt:
   $$\mathit{count}(x, L) := \#\big\{i \in \{1, \cdots, \#L\} \mid L(i) = x\big\}.$$

In diesem Kapitel präsentieren wir verschiedene Algorithmen, die das Sortier-Problem lösen, die also zum Sortieren von Listen benutzt werden können. Wir stellen zunächst zwei Algorithmen vor, die sehr einfach zu implementieren sind, deren Effizienz aber zu wünschen übrig läßt. Im Anschluß daran präsentieren wir zwei effizientere Algorithmen, deren Implementierung allerdings aufwendiger ist.

## 5.1 Sortieren durch Einfügen

Wir stellen zunächst einen sehr einfachen Algorithmus vor, der als "*Sortieren durch Einfügen*" (engl. *insertion sort*) bezeichnet wird. Wir beschreiben den Algorithmus durch *Gleichungen*. Der Algorithmus arbeitet nach dem folgenden Schema:

1. Ist die zu sortierende Liste $L$ leer, so wird als Ergebnis die leere Liste zurück gegeben:
   $$\texttt{sort}([]) = []$$
2. Andernfalls muß die Liste $L$ die Form $[x] + R$ haben. Dann sortieren wir den Rest $R$ und fügen das Element $x$ in diese Liste so ein, dass die Liste sortiert bleibt.
   $$\texttt{sort}\big([x] + R\big) = \texttt{insert}\big(x, \texttt{sort}(R)\big)$$

Das Einfügen eines Elements $x$ in eine sortierte Liste $S$ erfolgt nach dem folgenden Schema:

1. Falls die Liste $S$ leer ist, ist das Ergebnis $[x]$:
   $$\texttt{insert}(x, []) = [x].$$

2. Sonst hat $S$ die Form $[y] + R$. Wir vergleichen $x$ mit $y$.

   (a) Falls $x \preceq y$ ist, können wir $x$ vorne an die Liste $S$ anfügen:

   $$x \preceq y \rightarrow \texttt{insert}\big(x, [y] + R\big) = [x, y] + R.$$

   (b) Andernfalls fügen wir $x$ rekursiv in die Liste $R$ ein:

   $$\neg x \preceq y \rightarrow \texttt{insert}\big(x, [y] + R\big) = [y] + \texttt{insert}(x, R).$$

```
1   sort := procedure(l) {
2       match (l) {
3           case []   : return [];
4           case [x|r]: return insert(x, sort(r));
5       }
6   };
7
8   insert := procedure(x, l) {
9       match (l) {
10          case []   : return [ x ];
11          case [y|r]: if (x <= y) {
12                          return [x, y] + r;
13                      } else {
14                          return [y] + insert(x, r);
15                      }
16      }
17  };
18
19  l := [ rnd({1 .. 200}) : n in [1 .. 20] ];
20  print("l = $l$");
21  s := sort(l);
22  print("s = $s$");
```

Abbildung 5.1: Der Algorithmus "*Sortieren durch Einfügen*"

Der Algorithmus "*Sortieren durch Einfügen*" läßt sich leicht in SetlX umsetzen. Abbildung 5.1 zeigt das resultierende Programm.

1. Bei der Definition der Funktion `sort`$(l)$ benutzen wir das `match`-Konstrukt der Sprache `SetlX`. Das `match`-Konstrukt ist eine Weiterentwicklung des `switch`-Konstruktes. Zeile 3 greift, wenn die zu sortierende Liste leer ist. In Zeile 4 testen wir, ob die Liste $l$ die Form

   $$l = [x] + r$$

   hat. Hier bezeichnet $x$ das erste Element der Liste und $r$ umfasst alle restlichen Elemente der Liste $l$. In SetlX dient die Syntax `[x|r]` dazu, eine List der Form $[x] + r$ zu erkennen.

2. Bei der Definition der Funktion `insert`$(x, l)$ verwenden wir ebenfalls ein `match`-Konstrukt. Die drei bedingten Gleichungen, welche die Funktion `insert`() spezifizieren, lassen sich dadurch 1-zu-1 umsetzen.

3. In Zeile 19 definieren wir eine Liste der Länge 20, deren Elemente mit Hilfe der Funktion `rnd`() zufällig aus der Menge

   $$\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 200\}$$

   ausgewählt werden. Diese Liste wird dann ausgegeben und sortiert und die sortierte Liste wird zur Kontrolle ebenfalls ausgegeben.

### 5.1.1 Komplexität

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die bei einem Aufruf von "`Sortieren durch Einfügen`" in Zeile 26 von Abbildung 5.1 auf Seite 46 durchgeführt werden. Dazu berechnen wir zunächst die Anzahl der Aufrufe von "`<=`", die bei einem Aufruf von `insert`$(x, l)$ im schlimmsten Fall bei einer Liste der Länge $n$ durchgeführt werden. Wir bezeichnen diese Anzahl mit $a_n$. Der schlimmste Fall liegt dann vor, wenn $x$ größer als jedes Element aus $x$ ist, denn dann ist der Test "`x <= y`" in Zeile 12 immer falsch und wir haben solange einen rekursiven Aufruf der Funktion `insert`, solange die Restliste $r$ nicht leer ist. Insgesamt haben wir dann

$$a_0 = 0 \quad \text{und} \quad a_{n+1} = a_n + 1.$$

Durch eine einfache Induktion läßt sich sofort sehen, dass diese Rekurrenz-Gleichung die Lösung

$$a_n = n$$

hat. Im schlimmsten Falle führt der Aufruf von `insert`$(x, l)$ bei einer Liste $l$ mit $n$ Elementen also $n$ Vergleichs-Operationen durch, denn wir müssen dann $x$ mit jedem der $n$ Elemente aus $l$ vergleichen.

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die im schlimmsten Fall beim Aufruf von `sort`$(l)$ für eine Liste $l$ der Länge $n$ durchgeführt werden. Wir bezeichnen dieses Anzahl mit $b_n$. Offenbar gilt

$$b_0 = 0 \quad \text{und} \quad b_{n+1} = b_n + n, \tag{1}$$

denn für eine Liste $l = [x] + r$ der Länge $n + 1$ wird zunächst für die Liste $r$ rekursiv die Funktion `sort`$(r)$ aufgerufen. Das liefert den Summanden $b_n$. Anschließend wird mit `insert`$(x, $`sort(r)`$)$ das erste Element in diese Liste eingefügt. Wir hatten oben gefunden, dass dazu schlimmstenfalls $n$ Vergleichs-Operationen notwendig sind, was den Summanden $n$ erklärt.

Ersetzen wir in der Rekurrenz-Gleichung (1) $n$ durch $n - 1$, so erhalten wir

$$b_n = b_{n-1} + (n - 1).$$

Diese Rekurrenz-Gleichung können wir lösen, wenn wir die rechte Seite mit dem *Teleskop-Verfahren* expandieren:

$$\begin{aligned}
b_n &= b_{n-1} + (n-1) \\
&= b_{n-2} + (n-2) + (n-1) \\
&\vdots \\
&= b_{n-k} + (n-k) + \cdots + (n-1) \\
&\vdots \\
&= b_0 + 0 + 1 + \cdots + (n-1) \\
&= b_0 + \sum_{i=0}^{n-1} i \\
&= \tfrac{1}{2} \cdot n \cdot (n-1),
\end{aligned}$$

denn $b_0 = 0$ und für die Summe der Zahlen von 0 bis $n - 1$ läßt sich die Gleichung

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

durch eine einfache Induktion nachweisen. Wir halten fest, dass für die Anzahl der Vergleiche im schlimmsten Fall folgendes gilt:

$$b_n = \tfrac{1}{2} \cdot n^2 - \tfrac{1}{2} \cdot n = \tfrac{1}{2} \cdot n^2 + \mathcal{O}(n)$$

Im schlimmsten Fall werden also $\mathcal{O}(n^2)$ Vergleiche durchgeführt, der Algorithmus "*Sortieren durch Einfügen*" erfordert einen quadratischen Aufwand. Sie können sich überlegen, dass der schlimmste Fall genau dann eintritt, wenn die zu sortierende Liste $l$ absteigend sortiert ist, so dass die größten

Elemente gerade am Anfang der Liste stehen.

Der günstigste Fall für den Algorithmus "*Sortieren durch Einfügen*" liegt dann vor, wenn die zu sortierende Liste bereits aufsteigend sortiert ist. Dann wird beim Aufruf von $\texttt{insert}(x, \texttt{sort}(r))$ nur ein einziger Vergleich durchgeführt. Die Rekurrenz-Gleichungen für die Anzahl der Vergleiche in $\texttt{sort}(L)$ lautet dann

$$b_0 = 0 \quad \text{und} \quad b_{n+1} = b_n + 1. \tag{1}$$

Die Lösung dieser Rekurrenz-Gleichung haben wir oben berechnet, sie lautet $b_n = n$. Im günstigsten Falle ist der Algorithmus "*Sortieren durch Einfügen*" also linear.

## 5.2 Sortieren durch Auswahl

Wir stellen als nächstes den Algorithmus "*Sortieren durch Auswahl*" (engl. *selection sort*) vor. Der Algorithmus kann wie folgt beschrieben werden:

1. Ist die zu sortierende Liste $l$ leer, so wird als Ergebnis die leere Liste zurück gegeben:

   $$\texttt{sort}([]) = []$$

2. Andernfalls suchen wir in der Liste $l$ das kleinste Element und entfernen dieses Element aus $l$. Wir sortieren rekursiv die resultierende Liste, die ja ein Element weniger enthält. Zum Schluß fügen wir das kleinste Element vorne an die sortierte Liste an:

   $$l \neq [] \rightarrow \texttt{sort}\big(l\big) = \big[\texttt{min}(l)\big] + \texttt{sort}\big(\texttt{delete}(\texttt{min}(l), l)\big).$$

Der Algorithmus um ein Auftreten eines Elements $x$ aus einer Liste $l$ zu entfernen, kann ebenfalls leicht rekursiv formuliert werden. Wir unterscheiden drei Fälle:

1. Falls $l$ leer ist, gilt

   $$\texttt{delete}(x, []) = [].$$

2. Falls $x$ gleich dem ersten Element der Liste $l$ ist, gibt die Funktion den Rest $r$ zurück:

   $$\texttt{delete}(x, [x] + r) = r.$$

3. Andernfalls wird das Element $x$ rekursiv aus $r$ entfernt:

   $$x \neq y \rightarrow \texttt{delete}(x, [y] + r) = [y] + \texttt{delete}(x, r).$$

Schließlich geben wir noch rekursive Gleichungen an um das Minimum einer Liste zu berechnen:

1. Das Minimum der leeren Liste ist größer als irgendein Element. Wir schreiben daher

   $$\texttt{min}([]) = \infty.$$

2. Um das Minimum der Liste $[x] + r$ zu berechnen, berechnen wir rekursiv das Minimum von $r$ und benutzen die zweistellige Minimums-Funktion:

   $$\texttt{min}([x] + r) = \texttt{min}\big(x, \texttt{min}(r)\big).$$

   Dabei ist die zweistellige Minimums-Funktion wie folgt definiert:

   $$\texttt{min}(x, y) = \begin{cases} x & \text{falls } x \preceq y\,; \\ y & \text{sonst.} \end{cases}$$

Die Implementierung dieses Algorithmus in SetlX sehen Sie in Abbildung 5.2 auf Seite 49. Es war nicht notwendig, die Funktion *min*() zu implementieren, denn diese Funktion ist bereits vordefiniert. Die Funktion $\texttt{delete}(x, l)$ ist *defensiv* programmiert: Normalerweise sollte diese Funktion nur dann aufgerufen werden, wenn das zu entfernende Element $x$ auch tatsächlich in der Liste $l$ auftritt. Daher liegt in dem Fall, dass am Ende der Kette von rekursiven Aufrufen der Funktion $\texttt{delete}(x, l)$ die Liste $l$ leer ist, ein Fehler vor, der ausgegeben wird.

```
sort := procedure(l) {
    if (l == []) {
        return [];
    }
    x := min(l);
    return [x] + sort(delete(x,l));
};

delete := procedure(x, l) {
    match (l) {
        case []    : assert(false, "element $x$ not in list $l$");
        case [y|r] : if (x == y) {
                         return r;
                     }
                     return [y] + delete(x,r);
    }
};
```

Abbildung 5.2: Der Algorithmus "*Sortieren durch Auswahl*"

### 5.2.1 Komplexität

Um die Komplexität von "*Sortieren durch Auswahl*" analysieren zu können, müssen wir zunächst die Anzahl der Vergleiche, die bei der Berechnung von $\texttt{min}(l)$ durchgeführt werden, bestimmen. Es gilt

$$\texttt{min}([x_1, x_2, x_3, \cdots, x_n]) = \texttt{min}(x_1, \texttt{min}(x_2, \texttt{min}(x_3, \cdots \texttt{min}(x_{n-1}, x_n) \cdots))).$$

Also wird bei der Berechnung von $\texttt{min}(l)$ für eine Liste $l$ der Länge $n$ der Operator `min` insgesamt $(n-1)$-mal aufgerufen. Jeder Aufruf von `min` bedingt dann einen Aufruf des Vergleichs-Operators "$\preceq$". Bezeichnen wir die Anzahl der Vergleiche bein Aufruf von $\texttt{sort}(l)$ für eine Liste der Länge $l$ mit $b_n$, so finden wir also

$$b_0 = 0 \quad \text{und} \quad b_{n+1} = b_n + n,$$

denn um eine Liste mit $n+1$ Elementen zu sortieren, muss zunächst das Minimum dieser Liste berechnet werden. Dazu sind $n$ Vergleiche notwendig. Dann wird das Minimum aus der Liste entfernt und die Rest-Liste, die ja nur noch $n$ Elemente enthält, wird rekursiv sortiert. Das liefert den Beitrag $b_n$ in der obigen Summe.

Bei der Berechnung der Komplexität von "*Sortieren durch Einfügen*" hatten wir die selbe Rekurrenz-Gleichung erhalten. Die Lösung dieser Rekurrenz-Gleichung lautet also

$$b_n = \tfrac{1}{2} \cdot n^2 - \tfrac{1}{2} \cdot n = \tfrac{1}{2} \cdot n^2 + \mathcal{O}(n).$$

Das sieht so aus, als ob die Anzahl der Vergleiche beim "*Sortieren durch Einfügen*" genau so wäre wie beim "*Sortieren durch Auswahl*". Das stimmt aber nicht. Bei "*Sortieren durch Einfügen*" ist die Anzahl der durchgeführten Vergleiche im schlimmsten Fall $\frac{1}{2} \cdot n \cdot (n-1)$, beim "*Sortieren durch Auswahl*" ist Anzahl der Vergleiche immer $\frac{1}{2} \cdot n \cdot (n-1)$. Der Grund ist, dass zur Berechnung des Minimums einer Liste mit $n$ Elementen immer $n-1$ Vergleiche erforderlich sind. Um aber ein Element in eine Liste mit $n$ Elementen einzufügen, sind im Durchschnitt nur etwa $\frac{1}{2}n$ Vergleiche erforderlich, denn im Schnitt sind etwa die Hälfte der Elemente kleiner als das einzufügende Element und daher müssen beim Einfügen in eine sortierte Liste der Länge $n$ im Durchschnitt nur die ersten $\frac{n}{2}$ Elemente betrachtet werden. Daher ist die durchschnittliche Anzahl von Vergleichen beim "*Sortieren durch Einfügen*" $\frac{1}{4}n^2 + \mathcal{O}(n)$, also halb so groß wie beim "*Sortieren durch Auswahl*".

## 5.3 Sortieren durch Mischen

Wir stellen nun einen Algorithmus zum Sortieren vor, der für große Listen erheblich effizienter ist als die beiden bisher betrachteten Algorithmen. Der Algorithmus wird als "*Sortieren durch Mischen*" (engl. *merge sort*) bezeichnet und verläuft nach dem folgenden Schema:

1. Hat die zu sortierende Liste $l$ weniger als zwei Elemente, so ist $l$ bereits sortiert und es wird $l$ zurück gegeben:

   $$\#l < 2 \rightarrow \texttt{sort}(l) = l.$$

2. Ansonsten wird die Liste in zwei etwa gleich große Listen zerlegt. Diese Listen werden rekursiv sortiert und anschließend so gemischt, dass das Ergebnis sortiert ist:

   $$\#l \geq 2 \rightarrow \texttt{sort}(l) = \texttt{merge}\Big(\texttt{sort}\big(\texttt{split}_1(l)\big), \texttt{sort}\big(\texttt{split}_2(l)\big)\Big)$$

   Hier bezeichnen $\texttt{split}_1$ und $\texttt{split}_2$ die Funktionen, die eine Liste in zwei Teil-Listen zerlegen und `merge` ist eine Funktion, die zwei sortierte Listen so mischt, dass das Ergebnis wieder sortiert ist.

Abbildung 5.3 zeigt die Umsetzung dieser sortierten Gleichungen in ein SetlX-Programm, das eine verkettete Liste sortiert. Die beiden Funktionen $\texttt{split}_1$ und $\texttt{split}_2$ haben wir dabei zu einer Funktion `split` zusammen gefaßt, die zwei Listen zurück liefert. Ein Aufruf der Form

$$\texttt{split}(l)$$

liefert als Ergebnis ein Paar von Listen

$$[l_1, l_2],$$

wobei die Elemente der Liste $l$ gleichmäßig auf die beiden Listen $l_1$ und $l_2$ verteilt werden. Damit können wir die Details des SetlX-Programms in Abbildung 5.3 verstehen:

1. Wenn die zu sortierende Liste aus weniger als zwei Elementen besteht, dann ist diese Liste bereits sortiert und wir können diese Liste unverändert zurück geben.

2. Der Aufruf von `split` verteilt die Elemente der zu sortierenden Liste auf die beiden Listen `l1` und `l2`.

3. Diese Listen werden rekursiv sortiert und anschließend durch den Aufruf von `merge` zu einer sortierten Liste zusammen gefaßt.

Als nächstes überlegen wir uns, wie wir die Funktion `split` durch Gleichungen spezifizieren können.

1. Falls $l$ leer ist, produziert $\texttt{split}(l)$ zwei leere Listen:

   $$\texttt{split}(\texttt{[]}) = [\texttt{[]}, \texttt{[]}].$$

2. Falls $l$ genau ein Element besitzt, stecken wir dieses in die erste Liste:

   $$\texttt{split}([x]) = [\texttt{[}x\texttt{]}, \texttt{[]}].$$

3. Sonst hat $l$ die Form $[x, y] + r$. Dann spalten wir rekursiv $r$ in zwei Listen auf. Vor die erste Liste fügen wir $x$ an, vor die zweite Liste fügen wir $y$ an:

   $$\texttt{split}(r) = [r_1, r_2] \rightarrow \texttt{split}\big([x, y] + r\big) = \big[[x] + r_1, [y] + r_2\big].$$

Als letztes spezifizieren wir, wie zwei sortierte Listen $l_1$ und $l_2$ so gemischt werden können, dass das Ergebnis anschließend wieder sortiert ist.

1. Falls die Liste $l_1$ leer ist, ist das Ergebnis $l_2$:

   $$\texttt{merge}([], l_2) = l_2.$$

```
1   sort := procedure(l) {
2       if (#l < 2) {
3           return l;
4       }
5       [ l1, l2 ] := split(l);
6       return merge(sort(l1), sort(l2));
7   };
8
9   split := procedure(l) {
10      match (l) {
11          case []      : return [ [], [] ];
12          case [x]     : return [ [x], [] ];
13          case [x,y|r] : [r1,r2] := split(r);
14                         return [ [x] + r1, [y] + r2 ];
15      }
16  };
17
18  merge := procedure(l1, l2) {
19      if (l1 == []) {
20          return l2;
21      }
22      if (l2 == []) {
23          return l1;
24      }
25      x := l1[1];
26      y := l2[1];
27      if (x < y) {
28          return [x] + merge(l1[2..], l2);
29      } else {
30          return [y] + merge(l1, l2[2..]);
31      }
32  };
```

Abbildung 5.3: Implementierung des Algorithmus "*Sortieren durch Mischen*".

2. Falls die Liste $l_2$ leer ist, ist das Ergebnis $l_1$:

   $$\texttt{merge}(l_1, []) = l_1.$$

3. Andernfalls hat $l_1$ die Form $[x] + r_1$ und $l_2$ hat die Gestalt $[y] + r_2$. Dann führen wir eine Fallunterscheidung nach der relativen Größe von $x$ und $y$ durch:

   (a) $x \preceq y$.
   In diesem Fall mischen wir $r_1$ und $l_2$ und setzen $x$ an den Anfang dieser Liste:

   $$x \preceq y \rightarrow \texttt{merge}\big([x] + r_1, [y] + r_2\big) = [x] + \texttt{merge}\big(r_1, [y] + r_2\big).$$

   (b) $\neg x \preceq y$.
   In diesem Fall mischen wir $l_1$ und $r_2$ und setzen $y$ an den Anfang dieser Liste:

   $$\neg x \preceq y \rightarrow \texttt{merge}\big([x] + r_1, [y] + r_2\big) = [y] + \texttt{merge}\big([x] + r_1, r_2\big).$$

1. Falls eine der beiden Listen leer ist, so geben wir als Ergebnis die andere Liste zurück.

2. Wenn der Kontrollfluß in Zeile 7 ankommt, dann wissen wir, dass beide Listen nicht leer sind. Wir holen uns jeweils das erste Element der beiden Listen und speichern diese in

den Variablen $x$ und $y$ ab. Da wir diese Elemente aber mit Hilfe der Methode `getFirst` bekommen, bleiben diese Elemente zunächst Bestandteil der beiden Listen.

3. Anschließend prüfen wir, welche der beiden Variablen die kleinere ist.

   (a) Falls $x$ kleiner als $y$ ist, so entfernen wir $x$ aus der ersten Liste und mischen rekursiv die verkürzte erste Liste mit der zweiten Liste. Anschließen fügen wir $x$ an den Anfang der Ergebnis-Liste ein.

   (b) Andernfalls entfernen wir $y$ aus der zweiten Liste und mischen rekursiv die erste Liste mit der verkürzten zweiten Liste und fügen dann $y$ am Anfang der beim rekursiven Aufruf erhaltenen Liste ein.

### 5.3.1 Komplexität

Wir wollen wieder berechnen, wieviele Vergleiche beim Sortieren einer Liste mit $n$ Elementen durchgeführt werden. Dazu analysieren wir zunächst, wieviele Vergleiche zum Mischen zweier Listen $l_1$ und $l_2$ benötigt werden. Wir definieren eine Funktion

$$\texttt{cmpCount} : \mathit{List}(M) \times \mathit{List}(M) \to \mathbb{N}$$

so dass für zwei Listen $l_1$ und $l_2$ der Term $\texttt{cmpCount}(l_1, l_2)$ die Anzahl Vergleiche angibt, die bei Berechnung von $\texttt{merge}(l_1, l_2)$ erforderlich sind. Wir behaupten, dass für beliebige Listen $l_1$ und $l_2$

$$\texttt{cmpCount}(l_1, l_2) \leq \#l_1 + \#l_2$$

gilt. Für eine Liste $l$ bezeichnet dabei $\#l$ die Anzahl der Elemente der Liste. Wir führen den Beweis durch Induktion nach der Summe $\#l_1 + \#l_2$.

I.A.: $\#l_1 + \#l_2 = 0$.

Dann müssen $l_1$ und $l_2$ leer sein und somit ist beim Aufruf von $\texttt{merge}(l_1, l_2)$ kein Vergleich erforderlich. Also gilt

$$\texttt{cmpCount}(l_1, l_2) = 0 \leq 0 = \#l_1 + \#l_2.$$

I.S.: $\#l_1 + \#l_2 = n + 1$.

Falls entweder $l_1$ oder $l_2$ leer ist, so ist kein Vergleich erforderlich und wir haben

$$\texttt{cmpCount}(l_1, l_2) = 0 \leq \#l_1 + \#l_2.$$

Wir nehmen also an, dass gilt:

$$l_1 = [x] + r_1 \quad \text{und} \quad l_2 = [y] + r_2.$$

Wir führen eine Fallunterscheidung bezüglich der relativen Größe von $x$ und $y$ durch.

(a) $x \preceq y$. Dann gilt

$$\texttt{merge}\big([x] + r_1, [y] + r_2\big) = [x] + \texttt{merge}\big(r_1, [y] + r_2\big).$$

Also haben wir:

$$\texttt{cmpCount}(l_1, l_2) = 1 + \texttt{cmpCount}(r_1, l_2) \overset{IV}{\leq} 1 + \#r_1 + \#l_2 = \#l_1 + \#l_2.$$

(b) $\neg x \preceq y$. Dieser Fall ist völlig analog zum 1. Fall. □

**Aufgabe 8**: Überlegen Sie, wie die Listen $l_1$ und $l_2$ aussehen müssen, damit der Wert von

$$\texttt{cmpCount}(l_1, l_2)$$

maximal wird und geben Sie an, welchen Wert $\texttt{cmpCount}(l_1, l_2)$ in diesem Fal annimmt.

Wir wollen nun die Komplexität des Merge-Sort-Algorithmus im schlechtesten Fall berechnen und bezeichnen dazu die Anzahl der Vergleiche, die beim Aufruf von $\texttt{sort}(l)$ für eine Liste $l$ der Länge

$n$ schlimmstenfalls durchgeführt werden müssen mit $a_n$. Zur Vereinfachung nehmen wir an, dass $n$ die Form

$$n = 2^k \qquad \text{für ein } k \in \mathbb{N}$$

hat und definieren $b_k = a_n = a_{2^k}$. Zunächst berechnen wir den Anfangs-Wert, es gilt

$$b_0 = a_{2^0} = a_1 = 0,$$

denn bei einer Liste der Länge 1 findet noch kein Vergleich statt. Im rekursiven Fall wird zur Berechnung von `sort`$(l)$ die Liste $l$ zunächst durch `split` in zwei gleich große Listen geteilt, die dann rekursiv sortiert werden. Anschließend werden die sortierten Listen gemischt. Das liefert für das Sortieren einer Liste der Länge $2^{k+1}$ die Rekurrenz-Gleichung

$$b_{k+1} = 2 \cdot b_k + 2^{k+1}, \tag{1}$$

denn das Mischen der beiden halb so großen Listen kostet schlimmstenfalls $2^k + 2^k = 2^{k+1}$ Vergleiche und das rekursive Sortieren der beiden Teil-Listen kostet insgesamt $2 \cdot b_k$ Vergleiche.

Um diese Rekurrenz-Gleichung zu lösen, führen wir in (1) die Substitution $k \mapsto k + 1$ durch und erhalten

$$b_{k+2} = 2 \cdot b_{k+1} + 2^{k+2}. \tag{2}$$

Wir multiplizieren Gleichung (1) mit dem Faktor 2 und subtrahieren die erhaltene Gleichung von Gleichung (2). Dann erhalten wir

$$b_{k+2} - 2 \cdot b_{k+1} = 2 \cdot b_{k+1} - 4 \cdot b_k. \tag{3}$$

Diese Gleichung vereinfachen wir zu

$$b_{k+2} = 4 \cdot b_{k+1} - 4 \cdot b_k. \tag{4}$$

Diese Gleichung ist eine homogene lineare Rekurrenz-Gleichung 2. Ordnung. Das charakteristische Polynom der zugehörigen homogenen Rekurrenz-Gleichung lautet

$$\chi(x) = x^2 - 4 * x + 4 = (x - 2)^2.$$

Weil das charakteristische Polynom an der Stelle $x = 2$ eine doppelte Null-Stelle hat, lautet die allgemeine Lösung

$$b_k = \alpha \cdot 2^k + \beta \cdot k \cdot 2^k. \tag{5}$$

Setzen wir hier den Wert für $k = 0$ ein, so erhalten wir

$$0 = \alpha.$$

Aus (1) erhalten wir den Wert $b_1 = 2 \cdot b_0 + 2^1 = 2$. Setzen wir also in Gleichung (5) für $k$ den Wert 1 ein, so finden wir

$$2 = 0 \cdot 2^1 + \beta \cdot 1 \cdot 2^1,$$

also muß $\beta = 1$ gelten. Damit lautet die Lösung

$$b_k = k \cdot 2^k.$$

Aus $n = 2^k$ folgt $k = \log_2(n)$ und daher gilt für $a_n$

$$a_n = n \cdot \log_2(n).$$

Wir sehen also, dass beim "*Sortieren durch Mischen*" für große Listen wesentlich weniger Vergleiche durchgeführt werden müssen, als dies bei den beiden anderen Algorithmen der Fall ist.

Zur Vereinfachung haben wir bei der obigen Rechnung nur eine obere Abschätzung der Anzahl der Vergleiche durchgeführt. Eine exakte Rechnung zeigt, dass im schlimmsten Fall

$$n \cdot \log_2(n) - n + 1$$

Vergleiche beim "*Sortieren durch Mischen*" einer nicht-leeren Liste der Länge $n$ durchgeführt

werden müssen.

### 5.3.2 Eine feldbasierte Implementierung

Unsere bisherigen Implementierungen der Sortier-Algorithmen waren bezüglich des Speicherverbrauchs sehr ineffizient, denn wir haben dort nach Belieben neue Listen erzeugt. In der Realität ist es so, dass die zu sortierende Liste als Feld gegeben ist. Dann ist es das Ziel, möglichst mit diesem Feld auszukommen und beim Sortieren keinen weiteren Speicher zu allokieren. Bei dem Algorithmen "*Sortieren durch Mischen*" ist diese Ziel nur sehr schwer erreichbar, aber wenn wir uns die Benutzung eines Hilfsfeldes gestatten, das die gleiche Länge wie das zu sortierende Feld hat, dann läßt sich der Algorithmus so implementieren, dass bis auf das Hilfsfeld nur noch etwas Speicher auf dem Stack benötigt wird. Abbildung 5.4 auf Seite 54 zeigt eine feldbasierte Implementierung des Algorithmus "*Sortieren durch Mischen*".

```
1  sort := procedure(rw l) {
2      a := l;
3      mergeSort(l, 1, #l + 1, a);
4  };
5
6  mergeSort := procedure(rw l, start, end, rw a) {
7      if (end - start < 2) {
8          return;   // nothing to do if there is at most one element
9      }
10     middle := (start + end) \ 2;
11     mergeSort(l, start,  middle, a);
12     mergeSort(l, middle, end   , a);
13     merge(l, start, middle, end, a);
14 };
15
16 merge := procedure(rw l, start, middle, end, rw a) {
17     for (i in [start .. end-1]) {
18         a[i] := l[i];
19     }
20     idx1 := start;
21     idx2 := middle;
22     i    := start;
23     while (idx1 < middle && idx2 < end) {
24         if (a[idx1] <= a[idx2]) {
25             l[i] := a[idx1]; i += 1; idx1 += 1;
26         } else {
27             l[i] := a[idx2]; i += 1; idx2 += 1;
28         }
29     }
30     while (idx1 < middle) {
31         l[i] := a[idx1]; i += 1; idx1 += 1;
32     }
33     while (idx2 < end) {
34         l[i] := a[idx2]; i += 1; idx2 += 1;
35     }
36 };
```

Abbildung 5.4: Eine feldbasierte Implementierung des Algorithmus "*Sortieren durch Mischen*".

In der Methode `sort`($l$) haben wir mit dem Schlüsselwort "`rw`" den Parameter $l$ als einen *read-write Parameter* spezifiziert. Dies bewirkt, dass Änderungen des Parameters $l$ auch außerhalb der Methode `sort`() sichtbar sind. Daher gibt der Aufruf `sort`($l$) auch kein Ergebnis zurück. Statt dessen wird die als Argument übergebene Liste $l$ sortiert.

Der Algorithmus "*Sortieren durch Mischen*" wird in der Methode `mergeSort` implementiert. Diese Methode erhält die Argumente `start` und `end`, die den Bereich der Liste eingrenzen, der zu sortieren ist: Der Aufruf `mergeSort`(l, *start*, *end*, a) sortiert nur den Bereich

$$l\,[start, \cdots, end - 1].$$

Der Parameter $a$ bezeichnet dabei eine Liste, die als Hilfsfeld benutzt wird. Diese Liste muss die selbe Länge haben wie die Liste $l$. Die feldbasierte Implementierung weicht von der listenbasierten Implementierung ab, da wir keine Funktion `split` mehr benötigen, um die Liste aufzuspalten. Statt dessen berechnen wir die Mitte des zu sortierenden Teils der Liste $l$ mit der Formel

```
middle = (start + end) \ 2;
```

und spalten das Feld dann an dem Index `middle` in zwei etwa gleich große Teile, die wir in den Zeilen 11 und 12 rekursiv sortieren. Beachten Sie, dass wir hier bei der Division den Operator "\" benutzen, der eine ganzzahlige Division durchführt.

Nachdem wir die Liste in zwei Teile aufgeteilt haben, sortieren wir die beiden Teile rekursiv und rufen dann in Zeile 13 die Methode `merge` aus, welche die beiden sortierten Felder zu einem sortierten Feld zusammenfaßt. Diese Methode ist in den Zeilen 16 — 36 implementiert. Die Methode erhält 5 Argumente:

1. Der erste Parameter $l$ spezifiziert die zu sortierende Liste.
2. Die Parameter `start`, `middle` und `end` spezifizieren die beiden Teilfelder, die zu mischen sind. Das erste Teilfeld besteht aus den Elementen

   $$l\,[start, \cdots, middle - 1],$$

   das zweite Teilfeld besteht aus den Elementen

   $$l\,[middle, \cdots, end - 1].$$
3. Der letzte Parameter $a$ bezeichnet ein Hilfsfeld, das die selbe Größe hat wie die Liste $l$.

Der Aufruf setzt voraus, dass die beiden Teilfelder bereits sortiert sind. Das Mischen funktioniert dann wie folgt.

1. Zunächst werden die Daten aus den beiden zu sortierenden Teilfelder in Zeile 14 in das Hilfs-Feld $a$ kopiert.
2. Anschließend definieren wir drei Indizes:
   (a) `idx1` zeigt auf das nächste Element im ersten Teilfeld von $a$.
   (b) `idx2` zeigt auf das nächste Element im zweiten Teilfeld von $a$.
   (c) `i` gibt die Position im Ergebnis-Feld $l$ an, an die das nächste Element geschrieben wird.
3. Solange weder das erste noch das zweite Teilfeld des Feldes $a$ vollständig abgearbeitet ist, vergleichen wir in Zeile 30 die Elemente aus den beiden Teilfeldern und schreiben das kleinere von beiden an die Stelle, auf die der Index $i$ zeigt.
4. Falls in der `while`-Schleife eines der beiden Felder vor dem anderen erschöpft ist, müssen wir anschließend die restlichen Elemente des verbleibenden Teilfeldes in das Ergebnis-Feld kopieren. Die Schleife in Zeile 30 — 32 wird aktiv, wenn das zweite Teilfeld zuerst erschöpft wird. Dann werden die verbleibenden Elemente des ersten Teilfeldes in das Feld $l$ kopiert. Ist umgekehrt das erste Teilfeld zuerst erschöpft, dann werden in Zeile 33 — 35 die verbleibenden Elemente des zweiten Teilfeldes in das Feld $l$ kopiert.

### 5.3.3 Eine iterative Implementierung von *Sortieren durch Mischen*

```
1  sort := procedure(rw l) {
2      a := l;
3      mergeSort(l, a);
4  };
5
6  mergeSort := procedure(rw l, rw a) {
7      n := 1;
8      while (n < #l) {
9          k := 0;
10         while (n * (k + 1) + 1 <= #l) {
11             merge(l, n*k + 1, n*(k+1) + 1, min([n*(k+2), #l]) + 1, a);
12             k += 2;
13         }
14         n *= 2;
15     }
16 };
```

Abbildung 5.5: Eine nicht-rekursive Implementierung des Merge-Sort-Algorithmus

Die in Abbildung 5.4 gezeigte Implementierung des Merge-Sort-Algorithmus ist rekursiv. Die Effizienz einer rekursiven Implementierung ist in der Regel schlechter als die Effizienz einer sequentiellen Implementierung. Der Grund ist, dass der Aufruf einer rekursiven Funktion relativ viel Zeit kostet, denn beim Aufruf einer rekursiven Funktion müssen die lokalen Variablen der Funktion zusammen mit den Argumenten auf den Stack gelegt werden. Wir zeigen daher, wie sich der Merge-Sort-Algorithmus iterativ implementieren läßt. Abbildung 5.5 auf Seite 56 zeigt eine solche Implementierung. Statt der rekursiven Aufrufe haben wir hier zwei ineinander geschachtelte `while`-Schleifen. Die Idee ist, dass wir zunächst die ursprüngliche Liste $l$ in Listen der Länge 1 aufspalten, die offenbar bereits sortiert sind, denn jede Liste der Länge 1 ist sortiert. Anschließend mischen wir immer zwei benachbarte Listen der Länge 1 mit Hilfe der Funktion `merge`() und erhalten dann sortierte Listen der Länge 2. Diesen Schritt wiederholen wir nun für die Listen der Länge 2: Wir mischen jeweils zwei benachbarte Listen der Länge 2, die ja nun sortiert sind, zu einer sortierten Liste der Länge 4. Aus diesen Listen erzeugen wir in analoger Weise sortierte Listen der Länge 8, 16, $\cdots$. Das Verfahren bricht ab, wenn die gesamte Liste sortiert ist.

Die genaue Arbeitsweise der Implementierung wird deutlich, wenn wir die Invarianten der beiden `while`-Schleifen formulieren. Die Invariante der äußeren Schleife besagt, dass alle Teil-Felder der Liste $l$, welche eine Länge von $n$ haben und deren erstes Element einen Index der Form $n \cdot k + 1$ hat, bereits sortiert sind. Diese Teilfelder haben also die Form

$$l[n \cdot k + 1, \cdots, n \cdot (k + 1)].$$

Aufgabe der Schleife ist es dann, mit Hilfe der Funktion `merge`() jeweils zwei aufeinander folgende Teilfelder dieser Form zu einem sortierten Teilfeld der doppelten Länge zusammenzufassen.

In dem Ausdruck $l[n \cdot k + 1, \cdots, n \cdot (k + 1)]$ ist $k$ eine natürliche Zahl, mit der die einzelnen Teilfelder durchnummeriert werden. Ein solches Teilfeld hat also die Länge $n$, das erste Element ist

$$l[n \cdot k + 1]$$

und das letzte Element eines solchen Teilfeldes ist

$$l[n \cdot (k + 1)].$$

Für das erste Teilfeld hat der Index $k$ natürlich den Wert 0, so dass dieses Teilfeld die Form

$$l[1, \cdots, n]$$

hat, es besteht als aus den ersten $n$ Elementen der Liste $l$. Das zweite Teilfeld ist dann

$$l[n+1, \cdots, 2 \cdot n]$$

und so geht es bis zum Ende der Liste $l$ weiter. Möglicherweise hat das letzte Teilfeld eine Länge, die kleiner als $n$ ist. Dieser Fall tritt dann auf, wenn ein Feld der Länge $n$ nicht mehr in die Liste $l$ hinein paßt. Daher nehmen wir für das dritte Argument der Methode *merge* das Minimum der beiden Zahlen $n \cdot (k+2)$ und $\#l$.

**Aufgabe 9**: Geben Sie die Invariaten der beiden `while`-Schleifen explizit als prädikatenlogische Formeln an!

## 5.4 Der *Quick-Sort*-Algorithmus

Der von Tony Hoare entdeckte "*Quick-Sort*-Algorithmus" [Hoa61] funktioniert nach folgendem Schema:

1. Ist die zu sortierende Liste $L$ leer, so wird $L$ zurück gegeben:

   $$\texttt{sort}([]) = [].$$

2. Sonst hat $L$ die Form $L = [x] + R$. Dann verteilen wir die Elemente von $R$ so auf zwei Listen $S$ und $B$, dass $S$ alle Elemente von $R$ enthält, die kleiner-gleich $x$ sind, während $B$ die restlichen Elemente von $R$ enthält. Wir implementieren die Berechnung der beiden Listen über eine Funktion `partition`, die das Paar von Listen $S$ und $B$ erzeugt:

   $$\mathit{partition}(x, R) = \langle S, B \rangle.$$

   Hierbei gilt dann

   (a) Die Listen $S$ und $B$ enthalten zusammen genau die Elemente der Liste $R$

   $$\forall y \in R : \mathit{count}(y, S) + \mathit{count}(y, B) = \mathit{count}(y, R)$$

   (b) Alle Elemente aus $S$ sind kleiner-gleich $x$, die Elemente aus $B$ sind größer als $x$:

   $$\forall y \in S : y \preceq x \quad \text{und} \quad \forall y \in B : x \prec y.$$

   Formal können wir die Funktion *partition*() durch die folgenden Gleichungen beschreiben:

   (a) $\mathit{partition}(x, []) = \big\langle [], [] \big\rangle$,

   (b) $y \preceq x \ \wedge \mathit{partition}(x, R) = \big\langle S, B \big\rangle \rightarrow \mathit{partition}(x, [y] + R) = \big\langle [y] + S,\ B \big\rangle$

   (c) $\neg\,(y \preceq x) \wedge \mathit{partition}(x, R) = \big\langle S, B \big\rangle \rightarrow \mathit{partition}(x, [y] + R) = \big\langle S,\ [y] + B \big\rangle$,

   Anschließend sortieren wir die Listen $S$ und $B$ rekursiv. An die sortierte Liste $S$ hängen wir dann das Element $x$ und darauf folgt die sortierte Liste $B$. Insgesamt wird dieser Algorithmus durch die folgende Gleichung beschrieben:

   $$\mathit{partition}(x, R) = \langle S, B \rangle \rightarrow \mathit{sort}([x] + R) = \mathit{sort}(S) + [x] + \mathit{sort}(B).$$

Abbildung 5.6 zeigt die Umsetzung dieser Überlegung in einem SetlX-Programm. Die Funktion *sort*($l$) ist für eine Liste $l$ durch eine Fall-Unterscheidung implementiert.

1. Falls die zu sortierende Liste $l$ leer ist, wird als Ergebnis die leere Liste zurück gegeben.

2. Andernfalls können wir die Liste in der Form $[x|r]$ aufspalten. Hier ist $x$ das erste Element der Liste $l$, während die Liste $r$ alle restlichen Elemente der Liste $l$ enthält. Mit Hilfe der Funktion *partition*() wird nun die Restliste $r$ in zwei Listen $s$ und $b$ aufgespalten. Die Liste $s$ enthält alle die Elemente aus der Liste $r$, die kleiner oder gleich $x$ sind, während $b$ die Liste alle der Elemente der Liste $r$ ist, die größer als $x$ sind. Anschließend werden die beiden Teillisten $s$ und $b$ rekursiv sortiert und in der richtigen Reihenfolge zum Ergebnis zusammengesetzt.

```
1  sort := procedure(l) {
2      match (l) {
3          case []   : return [];
4          case [x|r]: [s,b] := partition(x, r);
5                      return sort(s) + [x] + sort(b);
6      }
7  };
8
9  partition := procedure(p, l) {
10     match (l) {
11         case []   : return [ [], [] ];
12         case [x|r]: [ r1, r2 ] := partition(p, r);
13                     if (x <= p) {
14                         return [ [x] + r1, r2 ];
15                     }
16                     return [ r1, [x] + r2 ];
17     }
18 };
```

Abbildung 5.6: Der *Quick-Sort*-Algorithmus.

Die Funktion *partition*$(p, l)$ bekommt als erstes Argument eine Zahl $p$ und als zweites Argument eine Liste von Zahlen $l$. Die Funktion berechnet dann ein Paar $[s, b]$, das aus zwei Listen $s$ und $b$ besteht. Die Zahlen in der Liste $s$ sind kleiner oder gleich dem *Pivot-Element* $p$, während die Elemente in der Liste $b$ alle größer als $p$ sind. Auch diese Funktion ist durch eine Fallunterscheidung definiert:

1. Falls die Liste $l$, deren Elemente auf zwei Listen zu verteilen sind, leer ist, dann werden als Ergebnis zwei leere Listen zurück gegeben.

2. Anderfalls hat $l$ die Form $[x|r]$. Dann wird zunächst die Liste $r$ rekursiv in zwei Teillisten $r_1$ und $r_2$ aufgeteilt. Jetzt muss nur noch durch einen Vergleich von $x$ mit dem Pivot-Element $p$ entschieden werden, in welche der beiden Listen $r_1$ oder $r_2$ das Element $x$ einzuordnen ist.

### 5.4.1 Komplexität

Als nächstes analysieren wir die Komplexität von *Quick-Sort*. Dazu untersuchen wir wieder die Zahl der Vergleiche, die beim Aufruf von *sort*$(L)$ für eine Liste $L$ mit $n$ Elementen durchgeführt werden. Wir betrachten zunächst die Anzahl der Vergleiche, die wir bei einem Aufruf der Form

$$\mathit{partition}(x, L, S, B)$$

für eine Liste $L$ mit $n$ Elementen durchführen müssen. Da wir jedes der $n$ Elemente der Liste $L$ mit $x$ vergleichen müssen, ist klar, dass dafür insgesamt $n$ Vergleiche erforderlich sind.

#### Komplexität im schlechtesten Fall

Wir berechnen als nächstes die Anzahl $a_n$ von Vergleichen, die wir im schlimmsten Fall beim Aufruf von *sort*$(L)$ für eine Liste $L$ der Länge $n$ durchführen müssen. Der schlimmste Fall tritt beispielsweise dann ein, wenn die Liste `small` leer ist und die Liste `big` folglich die Länge $n - 1$ hat. Für die Anzahl $a_n$ der Vergleiche gilt in diesem Fall

$$a_n = a_{n-1} + n - 1.$$

Der Term $n - 1$ rührt von den $n - 1$ Vergleichen, die beim Aufruf von *partition*$(x, R)$ in Zeile 6

von Abbildung 5.6 durchgeführt werden und der Term $a_{n-1}$ erfaßt die Anzahl der Vergleiche, die beim rekursiven Aufruf von $sort(L_2)$ benötigt werden.

Die Anfangs-Bedingung für die Rekurrenz-Gleichung lautet $a_0 = 0$, denn beim Sortieren einer leeren Liste sind keine Vergleiche notwendig. Damit läßt sich die obige Rekurrenz-Gleichung mit dem *Teleskop-Verfahren* lösen:

$$\begin{aligned} a_n &= a_{n-1} + (n-1) \\ &= a_{n-2} + (n-2) + (n-1) \\ &= a_{n-3} + (n-3) + (n-2) + (n-1) \\ &= \vdots \\ &= a_0 + 0 + 1 + 2 + \cdots + (n-2) + (n-1) \\ &= 0 + 0 + 1 + 2 + \cdots + (n-2) + (n-1) \\ &= \sum_{i=0}^{n-1} i = \tfrac{1}{2}n \cdot (n-1) = \tfrac{1}{2}n^2 - \tfrac{1}{2}n. \end{aligned}$$

Damit ist $a_n$ in diesem Fall genauso groß wie im schlimmsten Fall des Algorithmus' *Sortieren durch Einfügen*. Es ist leicht zu sehen, dass der schlechteste Fall dann eintritt, wenn die zu sortierende Liste $L$ bereits sortiert ist. Es existieren Verbesserungen des *Quick-Sort*-Algorithmus, für die der schlechteste Fall sehr unwahrscheinlich ist und insbesondere nicht bei sortierten Listen eintritt. Wir gehen später näher darauf ein.

### Durchschnittliche Komplexität

Der Algorithmus *Quick-Sort* würde seinen Namen zu Unrecht tragen, wenn er im *Durchschnitt* ein Komplexität der Form $\mathcal{O}(n^2)$ hätte. Wir analysieren nun die *durchschnittliche* Anzahl von Vergleichen $d_n$, die wir beim Sortieren einer Liste $L$ mit $n$ Elementen erwarten müssen. Im Allgemeinen gilt: Ist $L$ eine Liste mit $n+1$ Elementen, so ist die Zahl der Elemente der Liste `small`, die in Zeile 19 von Abbildung 5.6 berechnet wird, ein Element der Menge $\{0, 1, 2, \cdots, n\}$. Hat die Liste `small` insgesamt $i$ Elemente, so enthält die Liste `big` die restlichen $n - i$ Elemente. Gilt $\#\texttt{small} = i$, so werden zum rekursiven Sortieren von `small` und `big` durchschnittlich

$$d_i + d_{n-i}$$

Vergleiche durchgeführt. Bilden wir den Durchschnitt dieses Wertes für alle $i \in \{0, 1, \cdots, n\}$, so erhalten wir für $d_{n+1}$ die Rekurrenz-Gleichung

$$d_{n+1} = n + \frac{1}{n+1} \sum_{i=0}^{n} (d_i + d_{n-i}) \tag{5.1}$$

Der Term $n$ stellt die Vergleiche in Rechnung, die beim Aufruf von

```
partition(pivot, list, small, big)
```

durchgeführt werden. Um die Rekurrenz-Gleichung (1) zu vereinfachen, bemerken wir zunächst, dass für beliebige Funktionen $f : \mathbb{N} \to \mathbb{N}$ folgendes gilt:

$$\begin{aligned} \sum_{i=0}^{n} f(n-i) &= f(n) + f(n-1) + \cdots + f(1) + f(0) && (5.2) \\ &= f(0) + f(1) + \cdots + f(n-1) + f(n) && (5.3) \\ &= \sum_{i=0}^{n} f(i) && (5.4) \end{aligned}$$

Damit vereinfacht sich die Rekurrenz-Gleichung (5.1) zu

$$d_{n+1} = n + \frac{2}{n+1} \cdot \sum_{i=0}^{n} d_i. \tag{5.5}$$

Um diese Rekurrenz-Gleichung lösen zu können, substituieren wir $n \mapsto n+1$ und erhalten

$$d_{n+2} = n + 1 + \frac{2}{n+2} \cdot \sum_{i=0}^{n+1} d_i. \tag{5.6}$$

Wir multiplizieren nun Gleichung (5.6) mit $n+2$ und Gleichung (5.5) mit $n+1$ und erhalten

$$(n+2) \cdot d_{n+2} = (n+2) \cdot (n+1) + 2 \cdot \sum_{i=0}^{n+1} d_i \qquad \text{und} \tag{5.7}$$

$$(n+1) \cdot d_{n+1} = (n+1) \cdot n + 2 \cdot \sum_{i=0}^{n} d_i. \tag{5.8}$$

Wir bilden die Differenz der Gleichungen (5.7) und (5.8) und beachten, dass sich die Summationen bis auf den Term $2 \cdot d_{n+1}$ gerade gegenseitig aufheben. Dann erhalten wir

$$(n+2) \cdot d_{n+2} - (n+1) \cdot d_{n+1} = (n+2) \cdot (n+1) - (n+1) \cdot n + 2 \cdot d_{n+1} \tag{5.9}$$

Diese Gleichung vereinfachen wir zu

$$(n+2) \cdot d_{n+2} = (n+3) \cdot d_{n+1} + 2 \cdot (n+1). \tag{5.10}$$

Einer genialen Eingebung folgend teilen wir diese Gleichung durch $(n+2) \cdot (n+3)$ und erhalten

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} + \frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}. \tag{5.11}$$

Als nächstes bilden wir die Partialbruch-Zerlegung von dem Bruch

$$\frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}.$$

Dazu machen wir den Ansatz

$$\frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)} = \frac{\alpha}{n+2} + \frac{\beta}{n+3}.$$

Wir multiplizieren diese Gleichung mit dem Hauptnenner und erhalten

$$2 \cdot n + 2 = \alpha \cdot (n+3) + \beta \cdot (n+2),$$

was sich zu

$$2 \cdot n + 2 = (\alpha + \beta) \cdot n + 3 \cdot \alpha + 2 \cdot \beta$$

vereinfacht. Ein Koeffizientenvergleich liefert dann das lineare Gleichungs-System:

$$\begin{aligned} 2 &= \alpha + \beta \\ 2 &= 3 \cdot \alpha + 2 \cdot \beta \end{aligned}$$

Ziehen wir die erste Gleichung zweimal von der zweiten Gleichung ab, so erhalten wir $\alpha = -2$ und Einsetzen in die erste Gleichung liefert $\beta = 4$. Damit können wir die Gleichung (5.11) als

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} - \frac{2}{n+2} + \frac{4}{n+3} \tag{5.12}$$

schreiben. Wir definieren $a_n = \dfrac{d_n}{n+1}$ und erhalten dann aus der letzten Gleichung

$$a_{n+2} = a_{n+1} - \frac{2}{n+2} + \frac{4}{n+3}$$

Die Substitution $n \mapsto n-2$ vereinfacht diese Gleichung zu

$$a_n = a_{n-1} - \frac{2}{n} + \frac{4}{n+1} \tag{5.13}$$

Diese Gleichung können wir mit dem Teleskop-Verfahren lösen. Wegen $a_0 = \frac{d_0}{1} = 0$ gilt

$$a_n = 4 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i}. \tag{5.14}$$

Wir vereinfachen diese Summe:

$$\begin{aligned}
a_n &= 4 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 \cdot \sum_{i=2}^{n+1} \frac{1}{i} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 \cdot \frac{1}{n+1} - 4 \cdot \frac{1}{1} + 4 \cdot \sum_{i=1}^{n} \frac{1}{i} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 \cdot \frac{1}{n+1} - 4 \cdot \frac{1}{1} + 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= -\frac{4 \cdot n}{n+1} + 2 \cdot \sum_{i=1}^{n} \frac{1}{i}
\end{aligned}$$

Um unsere Rechnung abzuschließen, berechnen wir eine Näherung für die Summe

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$

Der Wert $H_n$ wird in der Mathematik als die $n$-te *harmonische Zahl* bezeichnet. Dieser Wert hängt mit dem Wert $\ln(n)$ zusammen, Leonhard Euler (1707 – 1783) hat gezeigt, dass für große $n$ die Approximation

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln(n)$$

benutzt werden kann. Genauer hat er folgendes gezeigt:

$$H_n = \ln(n) + \mathcal{O}(1).$$

Wir haben bei dieser Gleichung eine Schreibweise benutzt, die wir bisher noch nicht eingeführt haben. Sind $f$, $g$, $h$ Funktionen aus $\mathbb{R}^{\mathbb{N}}$, so schreiben wir

$$f(n) = g(n) + \mathcal{O}\big(h(n)\big) \quad \text{g.d.w.} \quad f(n) - g(n) \in \mathcal{O}\big(h(n)\big).$$

Also haben wir insgesamt für $a_n$ nun die Näherung

$$a_n = -\frac{4 \cdot n}{n+1} + 2 \cdot \ln(n) + \mathcal{O}(1)$$

gefunden. Wegen $d_n = (n+1) \cdot a_n$ gilt jetzt:

$$\begin{aligned}
d_n &= -4 \cdot n + 2 \cdot (n+1) \cdot H_n \\
&= -4 \cdot n + 2 \cdot (n+1) \cdot \big(\ln(n) + \mathcal{O}(1)\big) \\
&= 2 \cdot n \cdot \ln(n) + \mathcal{O}(n).
\end{aligned}$$

Wir vergleichen dieses Ergebnis mit dem Ergebnis, das wir bei der Analyse von "*Sortieren durch Mischen*" erhalten haben. Dort hatte sich die Anzahl der Vergleiche, die zum Sortieren eine Liste mit $n$ Elementen durchgeführt werden mußte, als

$$n \cdot \log_2(n) + \mathcal{O}(n)$$

ergeben. Wegen $\ln(n) = \ln(2) \cdot \log_2(n)$ benötigen wir bei Quick-Sort im Durchschnitt

$$2 \cdot \ln(2) \cdot n \cdot \log_2(n)$$

Vergleiche, also etwa $2 \cdot \ln(2) \approx 1.39$ mehr Vergleiche als beim "*Sortieren durch Mischen*".

### 5.4.2 Eine feldbasierte Implementierung von *Quick-Sort*

Zum Abschluß geben wir eine feldbasierte Implementierung des *Quick-Sort*-Algorithmus an. Abbildung 5.7 zeigt diese Implementierung.

```
1  sort := procedure(rw l) {
2      quickSort(1, #l, l);
3  };
4  quickSort := procedure(s, e, rw l) {
5      if (e <= s) {
6          return; // at most one element, nothing to do
7      }
8      m := partition(s, e, l);  // split index
9      quickSort(s, m - 1, l);
10     quickSort(m + 1, e, l);
11 };
12 partition := procedure(start, end, rw l) {
13     pivot := l[start];
14     left  := start + 1;
15     right := end;
16     while (true) {
17         while (left <= end && l[left] <= pivot) {
18             left += 1;
19         }
20         while (l[right] > pivot) {
21             right -= 1;
22         }
23         if (left >= right) {
24             break;
25         }
26         swap(left, right, l);
27     }
28     swap(start, right, l);
29     return right;
30 };
```

Abbildung 5.7: Implementierung des *Quick-Sort*-Algorithmus in SetlX.

1. Im Gegensatz zu der feldbasierten Implementierung des *Merge-Sort-Algorithmus* benötigen wir diesmal kein zusätzliches Hilfsfeld. Dies ist ein wesentlicher Vorteil des *Quick-Sort-Algorithmus*.

2. Die Funktion `sort` wird auf die Implementierung der Funktion `quickSort` zurück geführt. Diese Funktion bekommt zunächst die beiden Parameter `s` und `e`.

   (a) `s` gibt den Index des ersten Elementes des Teilfeldes an, das zu sortieren ist.

   (b) `e` gibt den Index des letzten Elementes des Teilfeldes an, das zu sortieren ist.

   Außerdem bekommt diese Funktion als letzten Parameter das zu sortierende Feld. Der Aufruf `quickSort`$(s, e)$ sortiert die Elemente

   $$l\texttt{[}s\texttt{]},\ l\texttt{[}s+1\texttt{]},\ \cdots,\ l\texttt{[}e\texttt{]}$$

   des Feldes $l$, d. h. nach dem Aufruf gilt:

   $$l\texttt{[}s\texttt{]} \preceq l\texttt{[}s+1\texttt{]} \preceq \cdots \preceq l\texttt{[}e\texttt{]}.$$

   Die Implementierung der Funktion `quickSort` entspricht weitgehend der listenbasierten Implementierung. Der wesentliche Unterschied besteht darin, dass die Funktion `partition`, die in Zeile 8 aufgerufen wird, die Elemente des Feldes $l$ so umverteilt, dass hinterher alle Elemente, die kleiner oder gleich dem *Pivot-Element* sind, links vor dem Index $m$ stehen, während die restlichen Elemente rechts von $m$ stehen. Das Pivot-Element selbst steht hinterher an der durch $m$ definierten Stelle.

3. Die Schwierigkeit bei der Implementierung von *Quick-Sort* liegt in der Codierung der Funktion `partition`, die in Zeile 17 beginnt. Die Funktion `partition` erhält zwei Argumente:

   (a) `start` ist der Index des ersten Elementes in dem aufzuspaltenden Teilbereich.

   (b) `end` ist der Index des letzten Elementes in dem aufzuspaltenden Teilbereich.

   Die Funktion `partition` liefert als Resultat einen Index $m$ aus der Menge

   $$\texttt{splitIdx} \in \{\texttt{start},\ \texttt{start}+1,\ \cdots,\ \texttt{end}\}.$$

   Außerdem wird der Teil des Feldes zwischen `start` und `end` so umsortiert, dass nach dem Aufruf der Funktion gilt:

   (a) Alle Elemente mit Index aus der Menge $\{\texttt{start}, \cdots, m-1\}$ kommen in der Ordnung "$\preceq$" vor dem Element an der Stelle `splitIdx`:

   $$\forall i \in \{\texttt{start}, \cdots, m-1\}:\ l[i] \preceq l[m].$$

   (b) Alle Elemente mit Index aus der Menge $\{m+1, \cdots, \texttt{end}\}$ kommen in der Ordnung "$\preceq$" hinter dem Element an der Stelle $m$:

   $$\forall i \in \{m+1, \cdots, \texttt{end}\}:\ l[m] \prec l[i].$$

   Der Algorithmus, um diese Bedingungen zu erreichen, wählt zunächst das Element

   $$l\texttt{[start]}$$

   als sogenanntes *Pivot-Element* aus. Anschließend läuft der Index `left` ausgehend von dem Index `start`+1 von links nach rechts bis ein Element gefunden wird, das größer als das Pivot-Element ist. Analog läuft der Index `right` ausgehend von dem Index `end` von rechts nach links, bis ein Element gefunden wird, dass kleiner oder gleich dem Pivot-Element ist. Falls nun `left` kleiner als `right` ist, werden die entsprechenden Elemente des Feldes ausgetauscht. In dem Moment, wo `left` größer oder gleich `right` wird, wird dieser Prozeß abgebrochen. Jetzt wird noch das Pivot-Element in die Mitte gestellt, anschließend wird `right` zurück gegeben.

4. Der Aufruf `swap`$(i, j, l)$ vertauscht die Elemente des Feldes $l$, die die Indizes $i$ und $j$ haben.

Der einfachste Weg um zu verstehen, wie die Funktion `partition` funktioniert, besteht darin, diese Funktion an Hand eines Beispiels auszuprobieren. Wir betrachten dazu einen Ausschnitt aus einem Feld, der die Form

$$\cdots, 7, 2, 9, 1, 8, 5, 11, \cdots$$

hat. Wir nehmen an, dass der Index `start` die Position der Zahl 7 angibt und das der Index `end` auf die 11 zeigt.

1. Dann zeigt der Index `left` zunächst auf die Zahl 2 und der Index `right` zeigt auf die die Zahl 11.
2. Die erste `while`-Schleife vergleicht zunächst die Zahl 2 mit der Zahl 7. Da $2 \leq 7$ ist, wird der Index `left` inkrementiert, so dass er jetzt auf die Zahl 9 zeigt.
3. Anschließend vergleicht die erste `while`-Schleife die Zahlen 9 und 7. Da $\neg(9 \leq 7)$ ist, wird die erste `while`-Schleife abgebrochen.
4. Nun startet die zweite `while`-Schleife und vergleicht die Zahlen 7 und 11. Da $7 < 11$ ist, wird der Index `right` dekrementiert und zeigt nun auf die Zahl 5.
5. Da $\neg(7 < 5)$ ist, wird auch die zweite `while`-Schleife abgebrochen.
6. Anschließend wir geprüft, ob der Index `right` bereits über den Index `left` hinüber gelaufen ist und somit `left` $\geq$ `right` gilt. Dies ist aber nicht der Fall, denn `left` zeigt auf die 9, während `right` auf die 5 zeigt, die rechts von der 9 liegt. Daher wird die äußere `while`-Schleife noch nicht abgebrochen.
7. Jetzt werden die Elemente, auf die die Indizes `left` und `right` zeigen, vertauscht. In diesem Fall werden also die Zahlen 9 und 5 vertauscht. Damit hat der Ausschnitt aus dem Feld die Form

   $$\cdots, 7, 2, 5, 1, 8, 9, 11, \cdots$$

8. Jetzt geht es in die zweite Runde der äußeren `while`-Schleife. Zunächst vergleichen wir in der inneren `while`-Schleife die Elemente 5 und 7. Da $5 \leq 7$ ist, wird der Index `left` inkrementiert.
9. Dann vergleichen wir die Zahlen 1 und 7. Da $1 \leq 7$ ist, wird der Index `left` ein weiteres Mal inkrementiert und zeigt nun auf die 8.
10. Der Vergleich $8 \leq 7$ fällt negativ aus, daher wird die erste `while`-Schleife jetzt abgebrochen.
11. Die zweite `while`-Schleife vergleicht nun 7 und 9. Da $7 < 9$ ist, wird der Index `right` dekrementiert und zeigt jetzt auf die 8.
12. Anschließend werden die Zahlen 7 und 8 verglichen. Da auch $7 < 8$ gilt, wird der Index `right` ein weiteres Mal dekrementiert und zeigt nun auf die 1.
13. Jetzt werden die Zahlen 7 und 1 verglichen. Wegen $\neg(7 < 1)$ bricht nun die zweite `while`-Schleife ab.
14. Nun wird geprüft, ob der Index `right` über den Index `left` hinüber gelaufen ist und somit `left` $\geq$ `right` gilt. Diesmal ist der Test positiv, denn `left` zeigt auf die 8, während `right` auf die 1 zeigt, die links von der 8 steht. Also wird die äußere Schleife durch den `break`-Befehl in Zeile 28 abgebrochen.
15. Zum Abschluß wird das Pivot-Element, das durch den Index `start` identifiziert wird, mit dem Element vertauscht, auf das der Index `right` zeigt, wir vertauschen also die Elemente 7 und 1. Damit hat das Feld die Form

    $$\cdots, 1, 2, 5, 7, 8, 9, 11, \cdots$$

    Als Ergebnis wird nun der Index `right`, der jetzt auf das Pivot-Element zeigt, zurück gegeben.

### 5.4.3 Korrektheit

Die Implementierung der Funktion `partition` ist trickreich. Daher untersuchen wir die Korrektheit der Funktion jetzt im Detail. Zunächst formulieren wir Invarianten, die für die äußere `while`-Schleife, die sich von Zeile 23 bis Zeile 35 erstreckt, gelten. Zur Abkürzun bezeichnen wir das Pivot-Element mit $x$, wir setzen also

$$x := \texttt{pivot} = l[\texttt{start}].$$

Dann gelten die folgenden Invarianten:

(I1) $\forall i \in \{\texttt{start}+1, \cdots, \texttt{left}-1\}$: $l[i] \preceq x$

(I2) $\forall j \in \{\texttt{right}+1, \cdots, \texttt{end}\}$: $x \prec l[j]$

(I3) $\texttt{start} + 1 \leq \texttt{left}$

(I4) $\texttt{right} \leq \texttt{end}$

(I5) $\texttt{left} \leq \texttt{right} + 1$

Wir weisen die Gültigkeit dieser Invarianten durch Induktion nach. Dazu ist zunächst zu zeigen, dass diese Invarianten dann erfüllt sind, bevor die Schleife zum ersten Mal durchlaufen wird. Zu Beginn gilt

$$\texttt{left} = \texttt{start} + 1.$$

Daraus folgt sofort, dass die dritte Invariante anfangs erfüllt ist. Außerdem gilt dann

$$\{\texttt{start}+1, \cdots, \texttt{left}-1\} = \{\texttt{start}+1, \cdots, \texttt{start}\} = \{\}$$

und damit ist auch klar, dass die erste Invariante gilt, denn für $\texttt{left} = \texttt{start} + 1$ ist die erste Invariante eine leere Aussage. Weiter gilt zu Beginn

$$\texttt{right} = \texttt{end},$$

woraus unmittelbar die Gültigkeit der vierten Invariante folgt. Außerdem gilt dann

$$\{\texttt{right}+1, \cdots, \texttt{end}\} = \{\texttt{end}+1, \cdots, \texttt{end}\} = \{\},$$

so dass auch die zweite Invariante trivialerweise erfüllt ist. Für die fünfte Invariante gilt anfangs

$$\texttt{left} \leq \texttt{right} + 1 \;\Leftrightarrow\; \texttt{start} + 1 \leq \texttt{end} + 1 \;\Leftrightarrow\; \texttt{start} \leq \texttt{end} \;\Leftrightarrow\; \texttt{true},$$

denn die Funktion `partition`(`start`, `end`) wird nur aufgerufen, falls $\texttt{start} < \texttt{end}$ ist.

Als nächstes zeigen wir, dass die Invarianten bei einem Schleifen-Durchlauf erhalten bleiben.

1. Die erste Invariante gilt, weil `left` nur dann inkrementiert wird, wenn vorher

   $$l[\texttt{left}] \preceq x$$

   gilt. Wenn die Menge $\{\texttt{start}+1, \cdots, \texttt{left}-1\}$ also um $i = \texttt{left}$ vergrößert wird, so ist sichergestellt, dass für dieses $i$ gilt:

   $$l[i] \preceq x.$$

2. Die zweite Invariante gilt, weil `right` nur dann dekrementiert wird, wenn vorher

   $$x \prec l[\texttt{right}]$$

   gilt. Wenn die Menge $\{\texttt{right}+1, \cdots, \texttt{end}\}$ also um $i = \texttt{right}$ vergrößert wird, so ist sichergestellt, dass für dieses $i$ gilt

   $$x \prec l[i].$$

3. Die Gültigkeit der dritten Invariante folgt aus der Tatsache, dass `left` in der ganzen Schleife höchstens inkrementiert wird. Wenn also zu Beginn $\texttt{start} + 1 \leq \texttt{left}$ gilt, so wird dies immer gelten.

4. Analog ist die vierten Invariante gültig, weil zu Beginn $\texttt{right} \leq \texttt{end}$ gilt und `right` immer nur dekrementiert wird.

5. Aus den ersten beiden Invarianten (I1) und (I2) folgt:

   $$\{\texttt{start} + 1, \cdots, \texttt{left} - 1\} \cap \{\texttt{right} + 1, \cdots, \texttt{end}\} = \{\},$$

   denn ein Element des Feldes kann nicht gleichzeitig kleiner-gleich $x$ und größer $x$ sein. Wenn $\texttt{right} + 1 \leq \texttt{end}$ ist, dann ist die zweite Menge nicht-leer und es folgt

   $$\texttt{left} - 1 < \texttt{right} + 1 \quad \text{und das impliziert} \quad \texttt{left} \leq \texttt{right} + 1.$$

   Andernfalls gilt $\texttt{right} = \texttt{end}$. Dann haben wir

   $$\texttt{left} \leq \texttt{right} + 1 \;\Leftrightarrow\; \texttt{left} \leq \texttt{end} + 1 \;\Leftrightarrow\; \texttt{true},$$

   denn wenn $\texttt{left} > \texttt{end}$ ist, wird `left` in der ersten Schleife nicht mehr erhöht. `left` wird nur dann und auch nur um 1 inkrementiert, solange $\texttt{left} \leq \texttt{end}$ gilt. Also kann `left` maximal den Wert $\texttt{end} + 1$ annehmen.

Um den Beweis der Korrektheit abzuschließen, muß noch gezeigt werden, dass alle `while`-Schleifen terminieren. Für die erste innere `while`-Schleife folgt das daraus, dass bei jedem Schleifen-Durchlauf die Variable `left` inkrementiert wird. Da die Schleife andererseits die Bedingung

$$\texttt{left} \leq \texttt{end}$$

enthält, kann `left` nicht beliebig oft inkrementiert werden und die Schleife muß irgendwann abbrechen.

Die zweite innere `while`-Schleife terminiert, weil einerseits `right` in jedem Schleifen-Durchlauf dekrementiert wird und andererseits aus der dritten und der fünften Invariante folgt:

$$\texttt{right} + 1 \geq \texttt{left} \geq \texttt{start} + 1, \quad \text{also} \quad \texttt{right} \geq \texttt{start}.$$

Die äußere `while`-Schleife terminiert, weil die Menge

$$M := \{\texttt{left}, \cdots, \texttt{right}\}$$

ständig verkleinert wird. Um das zu sehen, führen wir eine Fall-Unterscheidung durch:

1. Fall: Nach dem Ende der Schleife in Zeile 17 – 19 gilt

   $$\texttt{left} > \texttt{end}.$$

   Diese Schleife bricht also ab, weil die Bedingung $\texttt{left} \leq \texttt{end}$ verletzt ist. Wir haben oben schon gesehen, dass dann

   $$\texttt{left} = \texttt{end} + 1 \quad \text{und} \quad \texttt{right} = \texttt{end}$$

   gelten muß. Daraus folgt aber sofort

   $$\texttt{left} > \texttt{right}$$

   und folglich wird die äußere Schleife dann durch den Befehl `break` in Zeile 24 abgebrochen.

2. Fall: Nach dem Ende der Schleife in Zeile 17 – 19 gilt

   $$\texttt{left} \leq \texttt{end}.$$

   Die Schleife bricht also ab, weil die Bedingung $l[\texttt{left}] \preceq x$ verletzt ist, es gilt also

   $$x \prec l[\texttt{left}].$$

   Analog gilt nach dem Abbruch der zweiten inneren `while`-Schleife

$$l[\texttt{right}] \preceq x.$$

Wenn die äußere Schleife nun nicht abbricht weil $\texttt{left} < \texttt{right}$ ist, dann werden die Elemente $l[\texttt{left}]$ und $l[\texttt{right}]$ vertauscht. Nach dieser Vertauschung gilt offenbar

$$x \prec l[\texttt{right}] \quad \text{und} \quad l[\texttt{left}] \preceq x.$$

Wenn nun also die äußere Schleife erneut durchlaufen wird, dann wird die zweite innere Schleife mindestens einmal durchlaufen, so dass also `right` dekrementiert wird und folglich die Menge $M = \{\texttt{left}, \cdots, \texttt{right}\}$ um ein Element verkleinert wird. Das geht aber nur endlich oft, denn spätestens wenn die Menge leer ist, gilt $\texttt{left} = \texttt{right} + 1$ und die Schleife wird durch den Befehl `break` in Zeile 24 abgebrochen.

Jetzt haben wir alles Material zusammen, um die Korrektheit unserer Implementierung zu zeigen. Wenn die Schleife abbricht, gilt $\texttt{left} > \texttt{right}$. Wegen der fünften Invariante gilt $\texttt{left} \leq \texttt{right}+1$. Also gibt es nur noch die Möglichkeit

$$\texttt{left} = \texttt{right} + 1.$$

Wegen den ersten beiden Invarianten wissen wir also

$$\forall i \in \{\texttt{start} + 1, \cdots, \texttt{right}\}\colon\ l[i] \preceq x$$

$$\forall j \in \{\texttt{right} + 1, \cdots, \texttt{end}\}\colon\ x \prec l[j].$$

Durch das `swap` in Zeile 28 wird nun $x$ mit dem Element an der Position `right` vertauscht. Dann sind anschließend alle Elemente links von $x$ kleiner-gleich $x$ und alle Elemente rechts von $x$ sind größer. Damit ist die Korrektheit von `partition`() nachgewiesen.

### 5.4.4 Mögliche Verbesserungen

In der Praxis gibt es noch eine Reihe Tricks, um die Implementierung von *Quick-Sort* effizienter zu machen:

1. Anstatt immer das erste Element als Pivot-Element zu wählen, werden drei Elemente aus der zu sortierenden Liste ausgewählt, z. B. das erste, das letzte und ein Element aus der Mitte des Feldes. Als Pivot-Element wird dann das Element gewählt, was der Größe nach zwischen den anderen Elementen liegt.

   Der Vorteil dieser Strategie liegt darin, dass der schlechteste Fall, bei dem die Laufzeit von *Quick-Sort* quadratisch ist, wesentlich unwahrscheinlicher wird. Insbesondere kann der schlechteste Fall nicht mehr bei Listen auftreten, die bereits sortiert sind.

2. Falls weniger als 10 Elemente zu sortieren sind, wird auf "*Sortieren durch Einfügen*" zurück gegriffen.

Der Artikel von Bentley and M. Douglas McIlroy "*Engineering a Sort Function*" [BM93] beschreibt diese und weitere Verbesserungen des Quick-Sort Algorithmus.

**Aufgabe 10**: Implementieren Sie die oben aufgeführten Verbesserungen.

## 5.5 Eine untere Schranke für die Anzahl der Vergleiche

Wir wollen in diesem Abschnitt zeigen, dass jeder Sortier-Algorithmus, der in der Lage ist, eine beliebige Liste von Elementen zu sortieren, mindesten die Komplexität $\mathcal{O}\big(n \cdot \ln(n)\big)$ haben muss. Dabei setzen wir voraus, dass die einzelnen Elemente nur mit Hilfe des Operators $<$ verglichen werden können und wir setzen weiter voraus, dass wir eine Liste von $n$ verschiedenen Elementen haben, die wir sortieren wollen. Wir betrachten zunächst eine Liste von zwei Elementen: $[a_1, a_2]$. Um diese Liste zu sortieren, reicht ein Vergleich aus, denn es gibt nur zwei Möglichkeiten, wie diese beiden Elemente sortiert sein können:

1. Falls $a_1 < a_2$ ist, dann ist $[a_1, a_2]$ aufsteigend sortiert.
2. Falls $a_2 < a_1$ ist, dann ist $[a_2, a_1]$ aufsteigend sortiert.

Falls wir eine Liste von drei Elementen $[a_1, a_2, a_3]$ haben, so gibt es bereits 6 Möglichkeiten, diese anzuordnen:

$$[a_1, a_2, a_3], \quad [a_1, a_3, a_2], \quad [a_2, a_1, a_3], \quad [a_2, a_3, a_1], \quad [a_3, a_1, a_2], \quad [a_3, a_2, a_1].$$

Hier benötigen wir bereits drei Vergleiche zum Sortieren, denn mit zwei Vergleichen können wir maximal aus vier verschiedenen Möglichkeiten auswählen.

Im allgemeinen gibt es $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \prod_{i=1}^{n} i$ verschiedene Möglichkeiten, die Elemente einer $n$-elementigen Liste $[a_1, a_2, \cdots, a_n]$ anzuordnen. Dies läßt sich am einfachsten durch Induktion nachweisen:

1. Offenbar gibt es genau eine Möglichkeit, eine Liste von einem Element anzuordnen.
2. Um eine Liste von $n+1$ Elementen anzuordnen haben wir $n+1$ Möglichkeiten, das erste Element der Liste auszusuchen. In jedem dieser Fälle haben wir dann nach Induktions-Voraussetzung $n!$ Möglichkeiten, die restlichen Elemente anzuordnen, so dass wir insgesamt auf $(n+1) \cdot n! = (n+1)!$ verschiedene Anordnungsmöglichkeiten kommen.

Wir überlegen jetzt umgekehrt, aus wievielen Möglichkeiten wir mit $k$ verschiedenen Tests auswählen können.

1. Offenbar können wir mit einem Test aus zwei Möglichkeiten auswählen.
2. Mit zwei Tests können wir aus vier Möglichkeiten wählen.
3. Im Allgemeinen können wir mit $k$ Tests aus $2^k$ Möglichkeiten auswählen.

Die letzte Aussage ist einfach zu begründen: Stellen wir die Ergebnisse der Tests durch 0 und 1 dar, so entsprechen $k$ verschiedene Tests einem binären String der Länge $k$. Die binären Strings der Länge $k$ kodieren aber genau die Zahlen von 0 bis $2^k - 1$ im Zweiersystem. Nun gilt

$$card\big(\{0, 1, 2, \cdots, 2^k - 1\}\big) = 2^k.$$

Daher gibt es genau $2^k$ solcher Strings.

Wenn wir eine beliebig angeordnete Liste der Länge $n$ haben, dann gibt es ingesamt $n!$ Möglichkeiten, diese anzuordnen. Um nun herauszufinden, welche spezielle Anordnung unter den insgesamt möglichen $n!$ Anordnungen vorliegt, müssen wir $k$ Vergleiche durchführen, wobei für $k$ die Abschätzung

$$2^k \geq n!$$

gelten muss. Daraus folgt sofort

$$k \geq \log_2(n!).$$

An dieser Stelle benötigen wir eine Näherungsformel zur Berechnung von $\log_2(n!)$. Die einfachste solche Formel ist

$$\log_2(n!) = n \cdot \log_2(n) + \mathcal{O}(n).$$

Damit erhalten wir dann die Abschätzung

$$k \geq n \cdot \log_2(n) + \mathcal{O}(n)$$

und da der Merge-Sort-Algorithmus tatsächlich mit dieser Anzahl Vergleichen auskommt, ist dieser Algorithmus bezüglich der Anzahl der Vergleichs-Operationen optimal.

# Kapitel 6

# Abstrakte Daten-Typen und elementare Daten-Stukturen

Ähnlich wie der Begriff des Algorithmus von bestimmten Details eines Programms abstrahiert, abstrahiert der Begriff des *abstrakten Daten-Typs* von bestimmten Details konkreter Daten-Stukturen. Durch die Verwendung dieses Begriffs wird es möglich, Algorithmen von den zugrunde liegenden Daten-Stukturen zu trennen. Wir geben im nächsten Abschnitt eine Definition von abstrakten Daten-Typen und illustrieren das Konzept im folgenden Abschnitt an Hand von dem abstrakten Daten-Typ *Stack*. Im zweiten Abschnitt zeigen wir, wie die Sprache *Java* die Verwendung abstrakter Daten-Typen unterstützt. In den folgenden Abschnitten betrachten wir verschiedene Implementierungen des abstrakten Daten-Typs *Stack*. Anschließend zeigen wir, wie sich arithmetische Ausdrücke mit Hilfe eines Stacks auswerten lassen. Im vorletzten Abschnitt diskutieren wir den Nutzen, den die Verwendung abstrakter Daten-Typen hat.

## 6.1 Abstrakte Daten-Typen

Formal definieren wir einen *abstrakter Daten-Typ* als ein Tupel der Form $\langle T, P, Fz, Ts, Ax\rangle$. Die einzelnen Komponenten dieses Tupels haben dabei die folgende Bedeutung.

1. $T$ ist der *Name* des abstrakten Daten-Typs.
2. $P$ ist die Menge der verwendeten *Typ-Parameter*. Ein Typ-Parameter ist dabei einfach ein String. Diesen String interpretieren wir als Typ-Variable, d.h. wir können später für diesen String den Namen eines Daten-Typen einsetzen.
3. $Fz$ ist eine Menge von *Funktions-Zeichen*. Diese Funktions-Zeichen sind die Namen der Operationen, die der abstrakte Daten-Typ zur Vefrügung stellt,
4. $Ts$ ist eine Menge von *Typ-Spezifikation*, die zu jedem Funktions-Zeichen $f \in Fz$ eine *Typ-Spezifikation* der Form
   $$f : T_1 \times \cdots \times T_n \to S.$$
   enthält. Dabei sind $T_1$, $\cdots$, $T_n$ und $S$ Namen von Daten-Typen. Hier gibt es drei Möglichkeiten:
   (a) Die Namen konkreter Daten-Typen, wie z. B. “`int`” oder “`String`”.
   (b) Die Namen abstrakter Daten-Typen.
   (c) Ein Typ-Parameter aus der Menge $P$.

   Die Typ-Spezifikation $f : T_1 \times \cdots \times T_n \to S$ drückt aus, dass die Funktion $f$ in der Form
   $$f(t_1, \cdots, t_n)$$

aufgerufen wird und dass für $i = 1, \cdots, n$ das Argument $t_i$ vom Typ $T_i$ sein muß. Außerdem sagt die Typ-Spezifikation aus, dass das Ergebnis, das von der Funktion $f$ berechnet wird, immer vom Typ $S$ ist.

Zusätzlich fordern wir, dass entweder $T_1 = T$ ist, oder aber $S = T$ gilt. Es soll also entweder das erste Argument der Funktion $f$ den Wert $T$ haben, oder der Typ des von $f$ berechneten Ergebnisses soll gleich $T$ sein. Falls $T_1 \neq T$ ist (und damit zwangsläufig $S = T$ gilt), dann nennen wir die Funktion $f$ auch einen *Konstruktor* des Daten-Typs $T$, andernfalls bezeichnen wir $f$ als *Methode*.

5. $Ax$ ist eine Menge von prädikaten-logischen Formeln, die das Verhalten des abstrakten Daten-Typs beschreiben. Diese Formeln bezeichnen wir auch als die *Axiome* des Daten-Typs.

Wir geben sofort ein einfaches Beispiel für einen abstrakten Daten-Typ: den *Keller* (engl. *stack*). Einen Keller kann man sich anschaulich als einen Stapel von Elementen eines bestimmten Typs vorstellen, die aufeinander gelegt werden, ähnlich wie die Teller in der Essensausgabe einer Kantine. Dort werden Teller immer oben auf den Stapel gelegt und in umgekehrter Reihenfolge wieder vom Stapel entfernt. Insbesondere ist es nicht möglich, einen Teller aus der Mitte des Stapels zu entfernen. Formal definieren wir den Daten-Typ des *Kellers* wie folgt:

1. Als Namen wählen wir *Stack*.
2. Die Menge der Typ-Parameter ist $\{Element\}$.
3. Die Menge der Funktions-Zeichen ist
   $$\big\{Stack, push, pop, top, isEmpty\big\}.$$
4. Die Typ-Spezifikationen der Funktions-Zeichen sind wie folgt:
   (a) $Stack : Stack$
   Links von dem Doppelpunkt steht hier die Funktion mit dem Namen *Stack*, rechts steht der Name des ADT. In den gängigen Programmier-Sprachen (*Java*, `C++`, etc.) werden bestimmte Funktionen mit dem selben Namen bezeichnet wie der zugehörige ADT. Solche Funktionenen heißen Konstruktoren. Der Rückgabe-Wert eines Konstruktors hat immer den Typ des ADT.
   Der Konstruktor *Stack* kommt ohne Eingabe-Argumente aus. Ein solcher Konstruktor wird auch als der *Default-Konstruktor* bezeichnet.
   Der Aufruf $Stack()$ erzeugt einen neuen, leeren Stack.
   (b) $push : Stack \times Element \to Stack$
   Der Aufruf $push(S, x)$ legt das Element $x$ oben auf den Stack $S$. Wir werden im folgenden eine Objekt-orientierte Schreibweise verwenden und den Aufruf $push(S, x)$ als $S.push(x)$ schreiben.
   (c) $pop : Stack \to Stack$
   Der Aufruf $S.pop()$ entfernt das oberste Element von dem Stack $S$.
   (d) $top : Stack \to Element$
   Der Aufruf $S.top()$ liefert das auf dem Stack $S$ zuoberst liegende Element.
   (e) $isEmpty : Stack \to \mathbb{B}$
   Der Aufruf $S.isEmpty()$ testet, ob der Stack $S$ leer ist.

Die Anschauung, die dem Begriff des Stacks zu Grunde liegt, wird durch die folgenden Axiome erfaßt:

1. $Stack().top() = \Omega$
   Hier bezeichnet $\Omega$ den undefinierten Wert. Der Aufruf $Stack()$ liefert zunächst einen leeren Stack. Das Axiom drückt also aus, das ein leerer Stack kein oberstes Element hat.

2. $S.push(x).top() = x$

   Legen wir auf den Stack $S$ mit $S.push(x)$ ein Element $x$, so ist $x$ das oberste Element, was auf dem neu erhaltenen Stack liegt.

3. $Stack().pop() = \Omega$

   Der Versuch, von einem leeren Stack das oberste Element zu entfernen, liefert ein undefiniertes Ergebnis.

4. $S.push(x).pop() = S$

   Wenn wir auf den Stack $S$ ein Element legen, und anschließend von dem resultierenden Stack das oberste Element wieder herunter nehmen, dann erhalten wir den ursprünglichen Stack $S$, mit dem wir gestartet sind.

5. $Stack().isEmpty() =$ `true`

   Erzeugen wir mit $Stack()$ einen neuen Stack, so ist dieser zunächst leer.

6. $S.push(x).isEmpty() =$ `false`

   Legen wir ein Element $x$ auf einen Stack $S$, so kann der Stack $S$ danach nicht leer sein.

Beim Betrachten der Axiome läßt sich eine gewisse Systematik erkennen. Bezeichnen wir die Funktionen `Stack` und `push` als Generatoren so geben die Axiome das Verhalten der restlichen Funktionen auf den von den Generatoren erzeugten Stacks an.

Stacks spielen in vielen Bereichen der Informatik eine wichtige Rolle. Es gibt sogar Stack-basierte Programmier-Sprachen: Dort müssen bei einem Funktions-Aufruf alle Argumente zunächst auf einen Stack gelegt werden. Die aufrufende Funktion nimmt dann ihre Argumente vom Stack und legt das berechnete Ergebnis wieder auf den Stack. Die Sprache *PostScript* funktioniert nach diesem Prinzip. Die Sprache *Java* wird in einen *Byte-Code* übersetzt, der von der *Java Virtual Machine* (kurz JVM) interpretiert wird. Die JVM ist ebenfalls stack-basiert.

Wir werden später noch sehen, wie arithmetische Ausdrücke mit Hilfe eines Stacks ausgewertet werden können. Vorher zeigen wir, wie sich der abstrakte Daten-Typ des Stacks in der Programmier-Sprache *Java* implementieren läßt.

## 6.2 Darstellung abstrakter Daten-Typen in *Java*

In *Java* können abstrakte Daten-Typen entweder durch ein *Interface* oder durch eine *abstrakte Klasse* repräsentiert werden. Für den Stack wählen wir die Darstellung durch eine abstrakte Klasse, die in Abbildung 6.1 auf Seite 72 gezeigt wird. Die Darstellung durch eine abstrakte Klasse ist insofern flexibler, als wir hier die Möglichkeit haben, Methoden, die sich bereits auf der Abstraktions-Ebene des ADT realisieren lassen, zu implementieren.

Wir diskutieren die Darstellung des ADT Stack in Java nun Zeile für Zeile.

1. In der ersten Zeile deklarieren wir die Klasse `Stack<Element>` als *abstrakt*. Das Schlüsselwort `abstract` drückt dabei aus, das wir in dieser Klasse lediglich die Signaturen der Methoden angeben, die Implementierungen der Methoden werden in der abstrakten Klasse nicht angegeben.

   Der Name der Klasse ist `Stack<Element>`. Die Typ-Parameter des ADT sind hier in spitzen Klammern eingefaßt. Wenn es mehr als einen Typ-Parameter gibt, dann müssen diese durch Kommata getrennt werden.

   Durch "`implements Cloneable`" drücken wir aus, dass Objekte der Klasse `Stack<Element>` *geklont* werden können, d.h. wir fordern, dass die Klasse `Stack<Element>` eine Methode *clone*() zur Verfügung stellt. Die Idee ist, dass für ein *Java*-Objekt $o$ der Aufruf

   $$o.clone()$$

   eine Kopie des Objektes erzeugt.

```
1  public abstract class Stack<Element> implements Cloneable
2  {
3      public abstract void    push(Element e);
4      public abstract void    pop();
5      public abstract Element top();
6      public abstract boolean isEmpty();
7
8      public Stack<Element> clone() throws CloneNotSupportedException {
9          return (Stack<Element>) super.clone();
10     }
11
12     public final String toString() {
13         Stack<Element> copy;
14         try {
15             copy = clone();
16         } catch (CloneNotSupportedException e) {
17             return "*** ERROR ***";
18         }
19         String result = copy.convert();
20         String dashes = "\n";
21         for (int i = 0; i < result.length(); ++i) {
22             dashes = dashes + "-";
23         }
24         return dashes + "\n" + result + dashes + "\n";
25     }
26
27     private String convert() {
28         if (isEmpty()) {
29             return "|";
30         } else {
31             Element top = top();
32             pop();
33             return convert() + " " + top + " |";
34         }
35     }
36 }
```

Abbildung 6.1: darstellung des ADT-Stack in *Java*.

2. Zeile 3 enthält die Typ-Spezifikationen der Method `push`. Oben hatten wir diese Typ-Spezifikation als

   $$\texttt{push} : \texttt{Stack} \times \texttt{Element} \rightarrow \texttt{Stack}$$

   angegeben, in *Java* hat diese Typ-Spezifikation die Form

   ```
   void push(Element e);
   ```

   Hier fallen zwei Dinge auf:

   (a) In *Java* hat die Methode `push` ein Argument, während sie in der Definition des ADT zwei Argumente hat.

   In Java wird das erste Argument unterdrückt, denn dieses Argument ist bei jeder Methode vorhanden und hat den Wert Stack. Dieses Argument wird daher auch als *implizites Argument* bezeichnet. Diesem Umstand wird auch durch die Syntax eines Methoden-

Aufrufs Rechnung getragen. Wir schreiben

$s$.`push`$(e)$

an Stelle von `push`$(s, e)$.

(b) Der Rückgabe-Typ von `push` ist als `void` deklariert und nicht als `Stack<Element>`. Der Grund ist, dass ein Aufruf der Form

$s$.`push`$(e)$

nicht einen neuen Stack berechnet, sondern den Stack $s$, der als als implizites erstes Argument der Methode verwendet wird, verändert.

3. Die Zeilen 4 – 6 enthalten die Typ-Spezifikationen der restlichen Methoden.

4. Die Zeilen 8 – 10 enthält die Definition einer Methode `clone()`. Diese Methode ermöglicht es, einen Klon (also eine Kopie) eines Objektes vom Typ `Stack` zu erzeugen. Die Implementierung diskutieren wir später.

5. In den Zeilen 12 – 25 definieren wir die Methode `toString()`, mit der wir ein Objekt vom Daten-Typ `Stack` in einen String umwandeln können. Um die Implementierung dieser Methode zu verstehen, betrachten wir zunächst die Wirkung dieser Methode an Hand des folgenden Beispiels:

```
Stack<Element> stack = new Stack<Element>();
stack.push(1);
stack.push(2);
stack.push(3);
stack.toString();
```

Dann hat der Ausdruck `stack.toString()` den folgenden Wert:

```
-------------
| 1 | 2 | 3 |
-------------
```

Bei dieser Darstellung ist das oberste Element des Stacks also das Element, was am weitesten rechts liegt.

Die Implementierung der Methode `toString` verläuft in drei Schritten.

(a) Zunächst erzeugen wir mit Hilfe der Methode `clone()` eine Kopie des Stacks. Das ist deswegen notwendig, weil wir mit der Methode `top()` ja immer nur das erste Element des Stacks anschauen können. Um das zweite Element zu bekommen, müssen wir vorher das erste Element vom Stack herunter nehmen. Das geht mit der Operation `pop()`. Die Methode `toString()` soll aber den Stack selbst nicht verändern. Also kopieren wir vorher den Stack und ändern dann die Kopie.

Beim Ausführen der Methode `clone()` könnte es Probleme geben, es könnte eine *Exception* (Ausnahme) ausgelöst werden. Die Ausnahme fangen wir durch den `try-catch`-Block in den Zeilen 14 – 18 ab und geben in diesem Fall als Ergebnis eine Fehlermeldung zurück.

(b) Die Hilfs-Methode `convert()` berechnet einen String der Form

`| 1 | 2 | 3 |`.

Hierzu wird mit der `if`-Abfrage in Zeile 28 eine Fallunterscheidung durchgeführt: Falls der Stack leer ist, so ist das Ergebnis einfach nur der String "`|`". Andernfalls fragen wir das oberste Element des Stacks mit dem Aufruf `top()` in 31 ab, entfernen es durch einen Aufruf von `pop()` vom Stack und rufen anschließend für den so verkleinerten Stack rekursiv die Methode `toString()` auf. Das oberste Element des ursprünglichen Stacks wird dann hinten an das Ergebnis des rekursiven Aufrufs gehängt.

(c) Um zum Schluß noch die Linien darüber und darunter zu zeichnen, erzeugen wir in der `for`-Schleife in Zeile 21 – 23 eine Linie der erforderlichen Länge und verketten diese mit dem von `convert()` gelieferten String.

Beachten Sie, dass wir für Stacks die Methode `toString()` implementieren konnten ohne etwas darüber zu wissen, wie die Stacks überhaupt implementiert werden. Dies ist der wesentliche Vorteil des Konzeptes des ADT: Der Begriff des ADT abstrahiert von den Details der Implementierung und bietet damit eine Schnittstelle zu Stacks, die einfacher zu bedienen ist, als wenn wir uns mit allen Details auseinander setzen müßten. Ein weiterer wesentlicher Vorteil ist die Austauschbarkeit konkreter Implementierungen des Stacks. Wir werden später verschiedene konkrete Implementierungen des ADT `Stack` entwickeln. Da die Methode `toString` auf der abstrakten Ebene entwickelt worden ist, ist sie von den Details einer solchen konkreten Implementierung unabhängig!

## 6.3 Implementierung eines Stacks mit Hilfe eines *Arrays*

Eine Möglichkeit, einen Stack zu implementieren, besteht darin, die in einem Stack abgelegten Elemente in einem Feld abzuspeichern. Zusätzlich wird dann noch eine Zeiger benötigt, der auf das oberste Element des Stacks zeigt. Abbildung 6.2 auf Seite 75 zeigt eine solche Implementierung.

1. Durch "`extends Stack<Element>`" deklarieren wir, dass die Klasse `ArrayStack<Element>` den abstrakten Daten-Typ `Stack<Element>` implementiert.

2. Die Daten-Struktur wird durch zwei Member-Variablen realisiert:

   (a) Die in Zeile 3 definierte Variable `mArray` bezeichnet das Feld, in dem die einzelnen Elemente, die auf den Stack geschoben werden, abgespeichert werden.

   (b) Die in Zeile 4 definierte Variable `mIndex` gibt den Index in dem Feld `mArray` an, an dem das nächste Element abgelegt werden kann.

   Ich habe mir angewöhnt, jede Member-Variable mit dem Buchstaben `m` anfangen zu lassen. Durch diese Konvention lassen sich Member-Variablen später einfach von den lokalen Variablen einer Methode unterscheiden.

3. In dem Konstruktor legen wir in Zeile 7 das Feld `mArray` mit einer Größe von 1 an und initialisieren die Variable `mIndex` mit 0, denn 0 ist der erste freie Index in diesem Feld.

4. Bei der Implementierung der Methode `push`($e$) überprüfen wir zunächst in Zeile 12, ob in dem Feld noch Platz vorhanden ist um ein weiteres Element abzuspeichern. Falls dies nicht der Fall ist, legen wir in Zeile 13 ein neues Feld an, das doppelt so groß ist wie das alte Feld. Anschließend kopieren wir in der `for`-Schleife in den Zeilen 14 – 16 die Elemente aus dem alten Feld in das neue Feld und setzen dann die Variable `mArray` auf das neue Feld. Der *Garbage-Collector* der *Java Virtual Machine* sorgt jetzt dafür, dass der Speicher, der für das alte Feld allokiert worden war, wieder verwendet werden kann.

   Anschließend speichern wir das Element $e$ an der durch `mIndex` angegebenen Stelle ab und erhöhen die Variable `mIndex`, so dass diese jetzt wieder auf den nächsten freien Index in dem Array zeigt.

5. Die Funktion `pop()` können wir dadurch implementieren, dass wir die Variable `mIndex` dekrementieren. Vorher stellen wir durch den Aufruf von `assert` in Zeile 23 sicher, dass der Stack nicht leer ist. Damit der `assert` zur Laufzeit auch tatsächlich ausgeführt wird, müssen wir das Programm hinter mit der Option "`-ea`" starten, wir werden das Programm also in der Form

   ```
   java -ea StackTest
   ```

   ausführen. Die Option "`-ea`" steht für *enable assertions*.

```
1  public class ArrayStack<Element> extends Stack<Element>
2  {
3      Element[] mArray;  // abgespeicherte Elemente
4      int       mIndex;  // Index der nächsten freien Stele in mArray
5
6      public ArrayStack() {
7          mArray = (Element[]) new Object[1];
8          mIndex = 0;
9      }
10     public void push(Element e) {
11         int size = mArray.length;
12         if (mIndex == size) {
13             Element[] newArray = (Element[]) new Object[2 * size];
14             for (int i = 0; i < size; ++i) {
15                 newArray[i] = mArray[i];
16             }
17             mArray = newArray;
18         }
19         mArray[mIndex] = e;
20         ++mIndex;
21     }
22     public void pop() {
23         assert mIndex > 0 : "Stack underflow!";
24         --mIndex;
25     }
26     public Element top() {
27         assert mIndex > 0 : "Stack is empty!";
28         return (Element) mArray[mIndex - 1];
29     }
30     public boolean isEmpty() {
31         return mIndex == 0;
32     }
33 }
```

Abbildung 6.2: Array-basierte Implementierung eines Stacks.

6. Da der Stack-Pointer immer auf das nächste noch freie Feld zeigt, liefert der Ausdruck

   `mArray[mIndex-1]`

   in Zeile 28 das Element, das als letztes im Stack abgespeichert wurde. Dieses Element wird von der Methode `top`() zurück gegeben.

7. Die Prozedur `isEmpty()` überprüft in Zeile 31, ob der Index `mIndex` den Wert 0 hat, denn dann ist der Stack leer.

Damit ist unsere Implementierung des Daten-Typs Stack vollständig. Es bleibt ein Programm zu erstellen, mit dem wir diese Implementierung testen können. Abbildung 6.3 auf Seite 76 zeigt ein sehr einfaches Programm, in dem Stacks benutzt werden. Wir legen nacheinander die Zahlen $0, 1, \cdots, 32$ auf den Stack und geben jedesmal den Stack aus. Anschließend nehmen wir diese Zahlen der Reihe nach vom Stack herunter. Wir werden am Ende dieses Kapitels noch eine anspruchsvollere Anwendung von Stacks präsentieren, wenn wir Stacks zur Auswertung arithmetischer Ausdrücke verwenden.

```
1  import java.util.*;
2
3  public class StackTest
4  {
5      public static void main(String[] args) {
6          Stack<Integer> stack = new ArrayStack<Integer>();
7          for (int i = 0; i < 33; ++i) {
8              stack.push(i);
9              System.out.println(stack);
10         }
11         for (int i = 0; i < 33; ++i) {
12             System.out.println(i + ":" + stack.top());
13             stack.pop();
14             System.out.println(stack);
15         }
16     }
17 }
```

Abbildung 6.3: Test der Stack-Implementierung.

## 6.4 Eine Listen-basierte Implementierung von Stacks

Als nächstes zeigen wir eine alternative Implementierung des abstrakten Daten-Typs *Stack*, die auf einer verketteten Liste basiert. Abbildung 6.4 auf Seite 77 zeigt die Implementierung.

Um eine verkette Liste darzustellen, brauchen wir eine Daten-Struktur die Paare darstellt. Dabei ist die erste Komponente eines solchen Paares ein Element, das abgespeichert werden soll, und die zweite Komponente ist eine Referenz auf das nächste Paar. In der Klasse `ListStack<Element>` definieren wir daher zunächst eine *innere* Klasse `DPP` (zu lesen als *data pointer pair*), die ein solches Paar darstellt.

1. Die Klasse enthält ein Element, abgespeichert in der Variablen `mData` und eine Referenz auf das folgende Paar. Die Referenz wird in der Variablen `mNextPointer` abgespeichert.

2. Der Konstruktor dieser Klasse bekommt als Argumente ein abzuspeicherndes Element und eine Referenz auf das nächste Paar. Mit diesen Argumenten werden dann die Variablen `mData` und `mNextPointer` initialisiert.

3. Weierhin enthält die Klasse noch die Methode `recursiveCopy()`, die später gebraucht wird um eine Liste zu klonen. Diese Methode erzeugt ein neues Paar. Die erste Komponente dieses Paars ist ist das Daten-Element, die zweite Komponente erhalten wir durch einen rekursiven Aufruf von `recursiveCopy`().

4. Die Klasse `ListStack<Element>` selber enthält als einzige Member-Variable die Referenz `mPointer`. Wenn der Stack leer ist, dann hat dieser Pointer den Wert `null`. Sonst verweist die Referenz auf ein Objekt vom Typ `DPP`. In diesem Objekt liegt dann das oberste Stack-Element.

5. Der Konstruktor erzeugt einen leeren Stack, indem die Variable `mPointer` mit dem Wert `null` initialisiert wird.

6. Um ein neues Element auf den Stack zu legen, erzeugen wir ein Paar, das als erste Komponente das neue Element und als zweite Komponente eine Referenz auf die Liste enthält, die den bisherigen Stack repräsentierte. Anschließend lassen wir `mPointer` auf dieses Paar zeigen.

```
1  public class ListStack<Element> extends Stack<Element>
2  {
3      class DPP {
4          Element mData;
5          DPP     mNextPointer;
6
7          DPP(Element data, DPP nextPointer) {
8              mData        = data;
9              mNextPointer = nextPointer;
10         }
11         DPP recursiveCopy(DPP pointer) {
12             if (pointer == null) {
13                 return pointer;
14             } else {
15                 Element data        = pointer.mData;
16                 DPP     nextPointer = recursiveCopy(pointer.mNextPointer);
17                 return new DPP(data, nextPointer);
18             }
19         }
20     }
21
22     DPP mPointer;
23
24     public ListStack() {
25         mPointer = null;
26     }
27     public void push(Element e) {
28         mPointer = new DPP(e, mPointer);
29     }
30     public void pop() {
31         assert mPointer != null : "Stack underflow!";
32         mPointer = mPointer.mNextPointer;
33     }
34     public Element top() {
35         assert mPointer != null : "Stack is empty!";
36         return mPointer.mData;
37     }
38     public boolean isEmpty() {
39         return mPointer == null;
40     }
41     public ListStack<Element> clone() throws CloneNotSupportedException {
42         ListStack<Element> result = new ListStack<Element>();
43         if (mPointer != null) {
44             result.mPointer = mPointer.recursiveCopy(mPointer);
45         }
46         return result;
47     }
48 }
```

Abbildung 6.4: Implementierung eines Stacks mit Hilfe einer Liste

7. Um die Funktion `pop()` zu implementieren, setzen wir `mPointer` auf die zweite Komponente des ersten Paares.

8. Die Funktion `top()` implementieren wir, indem wir die erste Komponente des Paares, auf das `mPointer` zeigt, zurück geben.

9. Der Stack ist genau dann leer, wenn `mPointer` den Wert `null` hat.

Um diese zweite Implementierung des ADT *Stack* zu testen, reicht es aus, die Zeile 6 in der Implementierung der Klasse `StackTest` in Abbildung 6.3 zu ändern. Ursprünglich steht dort:

```
Stack<Integer> stack = new ArrayStack<Integer>();
```

Wir ersetzen hier den Konstruktor-Aufruf `new ArrayStack<Integer>()` durch den Aufruf des Konstruktors der Klasse `ListStack` und erhalten dann:

```
Stack<Integer> stack = new ListStack<Integer>();
```

Vergleichen wir die beiden Implementierungen, so stellen wir fest, das die Listen-basierte Implementierung mehr Speicherplatz als die Feld-basierte Implementierung verwendet, weil wir jedes Element in einem Objekt der Klasse `DPP` verpacken müssen. Auf der anderen Seite sind aber die Laufzeiten bei allen Methoden der Listen-basierten Implementierung konstant, wohingegen die Operation `push`$(x)$ bei der Feld-basierten Implementierung unter Umständen einen Aufwand erfordert, der proportional zur Anzahl der im Stack gespeicherten Elemente ist.

## 6.5 Auswertung arithmetischer Ausdrücke

Wir zeigen jetzt, wie Stacks zur Auswertung arithmetischer Ausdrücke benutzt werden können. Unter einem *arithmetischen Ausdruck* verstehen wir in diesem Zusammenhang einen String, der aus natürlichen Zahlen und den Operator-Symbolen "`+`", "`-`", "`*`", "`/`", "`%`" und "`^`" aufgebaut ist. Hierbei steht $x \,\%\, y$ für den Rest, der bei der Division von $x$ durch $y$ übrig bleibt und $x \,\hat{}\, y$ steht für die Potenz $x^y$. Alternativ kann die Potenz $x^y$ auch als $x ** y$ geschrieben werden. Außerdem können arithmetische Ausdrücke noch die beiden Klammer-Symbole "`(`" und "`)`" enthalten. Formal wird die Menge der arithmetische Ausdrücke *ArithExpr* induktiv definiert:

1. Jede Zahl $n \in \mathbb{N}$ ist ein arithmetischer Ausdruck:

   $$n \in \mathit{ArithExpr} \quad \text{f.a. } n \in \mathbb{N}.$$

2. Sind $s$ und $t$ arithmetische Ausdrücke, so sind auch $s + t$, $s - t$, $s * t$, $s/t$, $s\%t$ und $s\hat{}t$ arithmetische Ausdrücke:

   (a) $s \in \mathit{ArithExpr} \wedge t \in \mathit{ArithExpr} \rightarrow s + t \in \mathit{ArithExpr}$,

   (b) $s \in \mathit{ArithExpr} \wedge t \in \mathit{ArithExpr} \rightarrow s - t \in \mathit{ArithExpr}$,

   (c) $s \in \mathit{ArithExpr} \wedge t \in \mathit{ArithExpr} \rightarrow s * t \in \mathit{ArithExpr}$,

   (d) $s \in \mathit{ArithExpr} \wedge t \in \mathit{ArithExpr} \rightarrow s \,/\, t \in \mathit{ArithExpr}$,

   (e) $s \in \mathit{ArithExpr} \wedge t \in \mathit{ArithExpr} \rightarrow s \,\%\, t \in \mathit{ArithExpr}$,

   (f) $s \in \mathit{ArithExpr} \wedge t \in \mathit{ArithExpr} \rightarrow s \,\hat{}\, t \in \mathit{ArithExpr}$,

   (g) $s \in \mathit{ArithExpr} \rightarrow (\, s\,) \in \mathit{ArithExpr}$.

Haben wir nun einen String gegeben, der einen arithmetischen Ausdruck repräsentiert, als Beispiel betrachten wir den String

$$4 + 3 * 2 \,\hat{}\, 2 \,\hat{}\, 3,$$

so ist zunächst nicht klar, in welcher Reihenfolge die arithmetischen Operationen ausgeführt werden sollen. Um hier Klarheit zu schaffen müssen wir festlegen, wie stark die verschiedenen

Operator-Symbol binden. Wir vereinbaren, dass, wie in der Mathematik üblich, die Operatoren "`*`", "`/`" und "`%`" stärker binden als die Operatoren "`+`" und "`-`". Der Operator "`^`" bindet stärker als alle anderen Operatoren. Außerdem sind die Operatoren "`+`", "`-`", "`*`", "`/`", "`%`" alle *links-assoziativ*: Ein Ausdruck der Form

`1 - 2 - 3` wird wie der Ausdruck `(1 - 2) - 3`

gelesen. Der Operator "`^`" ist hingegen *rechts-assoziativ*: Ein arithmetischer Ausdruck der Form

`2 ^ 3 ^ 2` wird wie der Ausdruck `2 ^ (3 ^ 2)`

interpretiert. Unser Ziel ist es, ein Programm zu erstellen, dass einen String, der einen arithmetischen Ausdruck darstellt, auswertet. Dieses Programm wird ganz wesentlich mit dem abstrakten Daten-Typ *Stack* arbeiten.

### 6.5.1 Ein einführendes Beispiel

Wir demonstrieren das Verfahren, mit dem wir arithmetische Ausdrücke auswerten, zunächst an Hand eines Beispiels. Wir betrachten den arithmetischen Ausdruck

`1 + 2 * 3 - 4`.

Wir verarbeiten einen solchen Ausdruck von links nach rechts, Token für Token. Ein *Token* ist dabei entweder eine Zahl, eines der Operator-Symbole oder ein Klammer-Symbol. Bei der Verarbeitung benutzen wir drei Stacks:

1. Der *Token-Stack* enthält die eingegebenen Token. Dieser Stack enthält also sowohl Zahlen als auch Operator-Symbole und Klammer-Symbole.
2. Der *Argument-Stack* enthält Zahlen.
3. Der *Operator-Stack* enthält Operator-Symbole und Klammer-Symbole der Form "`(`".

Die Auswertung von `1 + 2 * 3 - 4` verläuft wie folgt:

1. Zu Beginn des Algorithmus enthält der Token-Stack die eingegebenen Tokens und die anderen beiden Stacks sind leer:

   ```
   mTokens    = [ 4, "-", 3, "*", 2, "+", 1 ],
   ```

   Beachten Sie, dass bei der horizontalen Darstellung des Stacks das Token, was als nächstes von der Methode *top*() zurück gegeben würde, am rechten Ende der Liste liegt.

   ```
   mArguments = [],
   mOperators = [].
   ```

2. Wir nehmen die Zahl `1` vom Token-Stack und legen sie auf den Argument-Stack. Die Werte der Stacks sind jetzt

   ```
   mTokens    = [ 4, "-", 3, "*", 2, "+" ],
   mArguments = [ 1 ],
   mOperators = [].
   ```

3. Wir nehmen den Operator `"+"` vom Token-Stack und legen ihn auf den Operator-Stack. Dann gilt:

   ```
   mTokens    = [ 4, "-", 3, "*", 2 ],
   mArguments = [ 1 ]
   mOperators = [ "+" ].
   ```

4. Wir nehmen die Zahl `2` vom Token-Stack und legen sie auf den Argument-Stack. Dann gilt:

   ```
   mTokens    = [ 4, "-", 3, "*" ],
   mArguments = [ 1, 2 ],
   mOperators = [ "+" ].
   ```

5. Wir nehmen den Operator `"*"` vom Token-Stack und vergleichen diesen Operator mit dem Operator `"+"`, der oben auf dem Operator-Stack liegt. Da der Operator `"*"` stärker bindet als der Operator `"+"` legen wir den Operator `"*"` ebenfalls auf den Operator-Stack, denn wir müssen diesen Operator auswerten, bevor wir den Operator `"+"` auswerten können. Dann gilt:

   ```
   mTokens    = [ 4, "-", 3 ],
   mArguments = [ 1, 2 ],
   mOperators = [ "+", "*"].
   ```

6. Wir nehmen die Zahl `3` vom Token-Stack und legen sie auf den Argument-Stack. Dann gilt:

   ```
   mTokens    = [ 4, "-" ],
   mArguments = [ 1, 2, 3 ],
   mOperators = [ "+", "*" ].
   ```

7. Wir nehmen den Operator `"-"` vom Token-Stack und vergleichen diesen Operator mit dem Operator `"*"`, der jetzt oben auf dem Stack liegt. Da der Operator `"*"` stärker bindet als der Operator `"-"`, werten wir jetzt den Operator `"*"` aus: Dazu nehmen wir die beiden Argumente 3 und 2 vom Argument-Stack, nehmen den Operator `"*"` vom Operator-Stack und berechnen, wie vom Operator `"*"` gefordert, das Produkt der beiden Argumente. Dieses Produkt legen wir dann wieder auf den Argument-Stack. Den Operator `"-"` legen wir wieder auf den Token-Stack zurück, denn wir haben die entsprechende Operation ja noch nicht ausgeführt. Dann haben unsere Stacks die folgende Gestalt:

   ```
   mTokens    = [ 4, "-" ],
   mArguments = [ 1, 6 ],
   mOperators = [ "+" ].
   ```

8. Wir nehmen den Operator `"-"` vom Token-Stack und vergleichen diesen Operator mit dem Operator `"+"` der nun zuoberst auf dem Operator-Stack liegt. Da beide Operatoren gleich stark binden und verschieden sind, werten wir jetzt den Operator `"+"` aus: Dazu nehmen wir die letzten beiden Argumente vom Argument-Stack, nehmen den Operator `"+"` vom Operator-Stack und berechnen die Summe der beiden Argumente. Diese Summe legen wir dann auf den Argument-Stack. Außerdem legen wir den Operator `"-"` wieder auf den Token-Stack zurück. Dann gilt:

   ```
   mTokens    = [ 4, "-" ],
   mArguments = [ 7 ],
   mOperators = [].
   ```

9. Wir nehmen den Operator `"-"` vom Token-Stack und legen ihn auf den Operator-Stack. Dann gilt:

   ```
   mTokens    = [ 4 ],
   mArguments = [ 7 ],
   mOperators = [ "-" ].
   ```

10. Wir nehmen die Zahl `4` vom Token-Stack und legen sie auf den Argument-Stack. Dann gilt:

    ```
    mTokens    = [],
    mArguments = [ 7, 4 ],
    mOperators = [ "-" ].
    ```

11. Der Input ist nun vollständig gelesen. Wir nehmen daher nun den Operator `"-"` vom Operator-Stack, der damit leer wird. Anschließend nehmen wir die beiden Argumente vom Argument-Stack, bilden die Differenz und legen diese auf den Argument-Stack. Damit gilt:

```
mTokens    = [],
mArguments = [ 3 ],
mOperators = [].
```

Das Ergebnis unserer Rechnung ist jetzt die noch auf dem Argument-Stack verbliebene Zahl 3.

### 6.5.2 Ein Algorithmus zur Auswertung arithmetischer Ausdrücke

Nach dem einführenden Beispiel entwickeln wir nun einen Algorithmus zur Auswertung arithmetischer Ausdrücke. Zunächst legen wir fest, welche Daten-Stukturen wir benutzen wollen.

1. `mTokens` ist ein Stack von Eingabe-Token. Wenn es sich bei den Token um Operatoren oder Klammer-Symbole handelt, dann haben diese Token den Typ `String`. Andernfalls stellen die Token Zahlen dar und haben den Typ `BigInteger`. Die gemeinsame Oberklasse der Klassen `String` und `BigInteger` ist `Object`. Daher deklarieren wir die Variable `mTokens` in der zu entwickelnden Klasse `Calculator` als:

   ```
   Stack<Object> mTokenStack;
   ```

2. `mArguments` ist ein Stack von ganzen Zahlen. Wir deklarieren diesen Stack als

   ```
   Stack<BigInteger> mArguments;
   ```

3. `mOperators` ist ein Stack, der die Operatoren und eventuell öffnende Klammern enthält. Da wir Operatoren durch Strings darstellen, deklarieren wir diesen Stack als

   ```
   Stack<String> mOperators;
   ```

Wenn wir das einführende Beispiel betrachten und verallgemeinern, dann stellen wir fest, dass wir Zahlen immer auf den Argument-Stack legen müssen, während bei Behandlung der Operatoren zwei Fälle auftreten können:

1. Der Operator wird auf den Operator-Stack gelegt, falls einer der folgenden Fälle vorliegt:
   (a) Der Operator-Stack ist leer.
   (b) Es liegt eine öffnende Klammer `"("` auf dem Operator-Stack.
   (c) Der neue Operator bindet stärker als der Operator, der bereits oben auf dem Operator-Stack liegt.
   (d) Der neue Operator ist identisch mit dem Operator, der bereits oben auf dem Operator-Stack liegt und dieser Operator-Stack ist außerdem rechts-assoziativ.

2. Andernfalls wird der Operator wieder auf den Token-Stack zurück gelegt. Dann wird der oberste Operator, der auf dem Operator-Stack liegt, vom Operator-Stack heruntergenommen, die Argumente dieses Operators werden vom Argument-Stack genommen, der Operator wird ausgewertet und das Ergebnis wird auf den Argument-Stack gelegt.

Die Abbildungen 6.5, 6.6 und 6.7 auf den Seiten 82, 84 und 85 zeigen eine Implementierung des Algorithmus zur Auswertung arithmetischer Ausdrücke in *Java*. Wir diskutieren zunächst die Implementierung der statischen Methode

```
static boolean evalBefore(String op1, String op2).
```

Diese Methode vergleicht die Operatoren `op1` und `op2` und entscheidet, ob der Operator `op1` vor dem Operator `op2` ausgewertet werden muß. Beim Aufruf dieser Methode ist der Operator `op1` der Operator, der oben auf dem Operator-Stack liegt und der Operator `op2` ist der Operator, der als letztes von dem Token-Stack genommen worden ist. Um entscheiden zu können, ob der Operator `op1` vor dem Operator `op2` auszuwerten ist, ordnen wir jedem Operator eine *Präzedenz* zu. Dies ist eine natürliche Zahl, die angibt, wie stark der Operator bindet. Tabelle 6.1 zeigt die Präzedenzen der von uns verwendeten Operatoren.

Ist die Präzedenz des Operators `op1` höher als die Präzedenz des Operators `op2`, so bindet `op1` stärker als `op2` und wird daher vor diesem ausgewertet. Ist die Präzedenz des Operators `op1`

```
1  import java.util.*;
2  import java.math.*;
3
4  public class Calculator {
5      Stack<BigInteger> mArguments;
6      Stack<String>     mOperators;
7      Stack<Object>     mTokenStack;
8
9      static boolean evalBefore(String op1, String op2) {
10         if (op1.equals("(")) {
11             return false;
12         }
13         if (precedence(op1) > precedence(op2)) {
14             return true;
15         } else if (precedence(op1) == precedence(op2)) {
16             return op1.equals(op2) ? isLeftAssociative(op1) : true;
17         } else {
18             return false;
19         }
20     }
21     static int precedence(String operator) {
22         if (operator.equals("+") || operator.equals("-")) {
23             return 1;
24         } else if ( operator.equals("*") || operator.equals("/") ||
25                     operator.equals("%")) {
26             return 2;
27         } else if (operator.equals("**") || operator.equals("^")) {
28             return 3;
29         } else {
30             System.out.println("ERROR: *** unkown operator *** ");
31         }
32         System.exit(1);
33         return 0;
34     }
35     static boolean isLeftAssociative(String operator) {
36         if (operator.equals("+") || operator.equals("-") ||
37             operator.equals("*") || operator.equals("/") ||
38             operator.equals("%")) {
39             return true;
40         } else if (operator.equals("**") || operator.equals("^")) {
41             return false;
42         } else {
43             System.out.println("ERROR: *** unkown operator *** ");
44         }
45         System.exit(1);
46         return false;
47     }
```

Abbildung 6.5: Die Klasse `Calculator`

| Operator | Präzedenz |
|---|---|
| `"+"`, `"-"` | 1 |
| `"*"`, `/`, `%` | 2 |
| `^`, `**` | 3 |

Tabelle 6.1: Präzedenzen der Operatoren.

kleiner als die Präzedenz des Operators `op2`, so wird der Operator `op2` auf den Operator-Stack gelegt. In dem Fall, dass die Präzedenzen von `op1` und `op2` gleich sind, gibt es zwei Fälle:

1. `op1` $\neq$ `op2`.

   Betrachten wir eine Beispiel: Der arithmetischer Ausdruck

   `2 + 3 - 4` wird implizit links geklammert: `(2 + 3) - 4`.

   Also wird in diesem Fall zunächst `op1` ausgewertet.

2. `op1` = `op2`.

   In diesem Fall spielt die *Assoziativität* des Operators eine Rolle. Betrachten wir dazu zwei Beispiele:

   `2 + 3 + 4` wird interpretiert wie `(2 + 3) + 4`,

   denn wir sagen, dass der Operator "`"+"`" *links-assoziativ* ist. Andererseits wird

   `2 ^ 3 ^ 4` interpretiert als `2 ^ (3 ^ 4)`,

   denn wir sagen, dass der Operator "`^`" *rechts-assoziativ* ist.

   Die Operatoren "`"+"`", "`"-"`", "`"*"`", "`/`" und "`%`" sind alle links-assoziativ. Hier wird als zunächst `op1` ausgewertet. Der Operator "`^`" ist rechts-assoziativ. Ist der oberste Operator auf dem Operator-Stack also "`^`" und wird dann nochmal der Operator "`^`" gelesen, so wird auch die neue Instanz dieses Operators auf den Stack gelegt.

Mit diesem Vorüberlegung können wir nun die Implementierung von `evalBefore`(`op1`, `op2`) in Abbildung 6.5 verstehen.

1. Falls `op1` der String `"("` ist, so legen wir `op2` auf jeden Fall auf den Stack, denn `"("` ist ja gar kein Operator, denn wir auswerten könnten. Daher geben wir in Zeile 11 den Wert `false` zurück.

2. Falls die Präzedenz des Operators `op1` höher ist als die Präzedenz des Operators `op2`, so liefert `evalBefore`(`op1`, `op2`) in Zeile 14 den Wert `true`.

3. Falls die Präzedenzen der Operatoren `op1` und `op2` identisch sind, so gibt es zwei Fälle:

   (a) Sind die beiden Operatoren gleich, so ist das Ergebnis von `evalBefore(op1,op2)` genau dann `true`, wenn der Operator links-assoziativ ist.

   (b) Falls die beiden Operatoren verschieden sind, hat das Ergebnis von `evalBefore(op1,op2)` den Wert `true`.

   Diese beiden Fälle werden in Zeile 16 behandelt.

4. Ist die Präzedenz des Operators `op1` kleiner als die Präzedenz des Operators `op2`, so liefert `evalBefore`(`op1`, `op2`) in Zeile 18 den Wert `false`.

Die Implementierung der Methode `predence()` in den Zeilen 21 – 34 ergibt sich unmittelbar aus der Tabelle 6.1 auf Seite 83. Die Implementierung der Methode `isLeftAssociative()` in den Zeilen 35 – 47 legt fest, dass die Operatoren `"+"`, `"-"`, `"*"`, `"/"` und `"%"` links-assoziativ sind, während die Operatoren `"**"` und `"^"` rechts-assoziativ sind.

Abbildung 6.6 auf Seite 84 zeigt die Implementierung der Methode `popAndEvaluate()`. Aufgabe dieser Methode ist es,

1. einen Operator vom Operator-Stack zu nehmen (Zeile 49 – 50),
2. dessen Argumente vom Argument-Stack zu holen, (Zeile 51 – 54),
3. den Operator auszuwerten (Zeile 55 – 69) und
4. das Ergebnis wieder auf dem Argument-Stack abzulegen (Zeile 70).

```
48      void popAndEvaluate() {
49          String operator = mOperators.top();
50          mOperators.pop();
51          BigInteger rhs = mArguments.top();
52          mArguments.pop();
53          BigInteger lhs = mArguments.top();
54          mArguments.pop();
55          BigInteger result = null;
56          if (operator.equals("+")) {
57              result = lhs.add(rhs);
58          } else if (operator.equals("-")) {
59              result = lhs.subtract(rhs);
60          } else if (operator.equals("*")) {
61              result = lhs.multiply(rhs);
62          } else if (operator.equals("/")) {
63              result = lhs.divide(rhs);
64          } else if (operator.equals("**") || operator.equals("^")) {
65              result = lhs.pow(rhs.intValue());
66          } else {
67              System.out.println("ERROR: *** Unknown Operator ***");
68              System.exit(1);
69          }
70          mArguments.push(result);
71      }
```

Abbildung 6.6: Die Klasse `Calculator`

Damit können wir die in Abbildung 6.7 gezeigte Implementierung des Konstruktors der Klasse `Calculator` diskutieren.

1. Zunächst erzeugen wir in Zeile 73 ein Objekt der Klasse `MyScanner`. Dieser Scanner liest einen String ein und zerlegt diesen in Token. Wir erhalten in Zeile 74 einen Stack zurück, der die Token in der Reihenfolge enthält, in der sie eingelesen worden sind. Geben wir beispielsweise den String

   “`1 + 2 * 3 - 4`”

   ein, so bekommt die Variable `mTokenStack` in Zeile 74 den Wert

   [ 4, ”-”, 3, ”*”, 2, ”+”, 1 ]

   zugewiesen. Außerdem initialisieren wir den Argument-Stack und den Operator-Stack in Zeile 75 und 76.

2. In der nächsten Phase verarbeiten wir die einzelnen Tokens des Token-Stacks und verteilen diese Tokens auf Argument-Stack und Operator-Stack wie folgt:

   (a) Ist das gelesene Token eine Zahl, so legen wir diese auf den Argument-Stack und lesen das nächste Token.

```
72      public Calculator() {
73          MyScanner scanner = new MyScanner(System.in);
74          mTokenStack = scanner.getTokenStack();
75          mArguments  = new ArrayStack<BigInteger>();
76          mOperators  = new ArrayStack<String>();
77          while (!mTokenStack.isEmpty()) {
78              if (mTokenStack.top() instanceof BigInteger) {
79                  BigInteger number = (BigInteger) mTokenStack.top();
80                  mTokenStack.pop();
81                  mArguments.push(number);
82                  continue;
83              }
84              String nextOp = (String) mTokenStack.top();
85              mTokenStack.pop();
86              if (mOperators.isEmpty() || nextOp.equals("(")) {
87                  mOperators.push(nextOp);
88                  continue;
89              }
90              String stackOp = mOperators.top();
91              if (stackOp.equals("(") && nextOp.equals(")") ) {
92                  mOperators.pop();
93              } else if (nextOp.equals(")")) {
94                  popAndEvaluate();
95                  mTokenStack.push(nextOp);
96              } else if (evalBefore(stackOp, nextOp)) {
97                  popAndEvaluate();
98                  mTokenStack.push(nextOp);
99              } else {
100                 mOperators.push(nextOp);
101             }
102         }
103         while (!mOperators.isEmpty()) {
104             popAndEvaluate();
105         }
106         BigInteger result = mArguments.top();
107         System.out.println("The result is: " + result);
108     }
109     public static void main(String[] args) {
110         Calculator calc = new Calculator();
111     }
112 }
```

Abbildung 6.7: Der Konstruktor der Klasse `Calculator`.

Im folgenden können wir immer davon ausgehen, dass das gelesene Token ein Operator oder eine der beiden Klammern `"("` oder `")"` ist.

(b) Falls der Operator-Stack leer ist oder wenn das gelesene Token eine öffnende Klammer `"("` ist, legen wir den Operator oder die Klammer auf den Operator-Stack.

(c) Falls das Token eine schließende Klammer `")"` ist und wenn zusätzlich der Operator auf dem Operator-Stack eine öffnende Klammer `"("` ist, so entfernen wir diese Klammer vom Operator-Stack.

(d) Falls jetzt das Token aus dem Token-Stacks eine schließende Klammer `")"` ist, so wissen wir, dass das Token auf dem Operator-Stack keine öffnende Klammer sein kann, sondern ein echter Operator ist. Diesen Operator evaluieren wir mit Hilfe der Methode `popAndEvaluate()`. Gleichzeitig schieben wir die schließende Klammer, die wir vom Token-Stack genommen haben, wieder auf den Token-Stack zurück, denn wir haben die dazu gehörige öffnende Klammer ja noch nicht gefunden.

Da wir danach wieder zum Beginn der Schleife zurück kehren, werden wir in diesem Fall solange Operatoren vom Operator-Stack nehmen und auswerten bis wir im Operator-Stack auf eine öffnende Klammer treffen.

Im folgenden können wir davon ausgehen, dass weder das oberste Zeichen auf dem Operator-Stack, noch das oberste Token auf dem Token-Stack eine Klammer ist.

(e) Falls der oberste Operator auf dem Operator-Stack eine höhere Präzedenz hat als der zuletzt gelesene Operator, evaluieren wir den obersten Operator auf dem Operator-Stack mit Hilfe der Methode `popAndEvaluate()`.

Gleichzeitig schieben wir den Operator, den wir vom Token-Stack genommen haben, wieder auf den Token-Stack zurück, denn wir haben diesen Operator ja noch nicht weiter behandelt.

(f) Andernfalls legen wir den zuletzt gelesenen Operator auf den Operator-Stack.

Diese Phase endet sobald der Token-Stack leer ist.

3. Zum Abschluß evaluieren wir alle noch auf dem Operator-Stack verbliebenen Operatoren mit Hilfe der Methode `popAndEvaluate()`. Wenn die Eingabe ein syntaktisch korrekter arithmetischer Ausdruck war, dann sollte am Ende der Rechnung noch genau eine Zahl auf dem Argument-Stack liegen. Diese Zahl ist dann unser Ergebnis, das wir ausgeben.

Aus Gründen der Vollständigkeit zeigen wir in Abbildung 6.8 noch die Implementierung der Klasse `MyScanner`. Wir benutzen die Klasse `Scanner` aus dem Paket `java.io`. Diese Klasse stellt unter anderem die Methoden *hasNext*() und *hasNextBigInteger*() mit denen wir überprüfen können, ob die Eingabe noch ungelesene Zeichen enthält und ob diese ungelesenen Zeichen als ganze Zahl interpretiert werden können. Die Methode *nextBigInteger*() gibt dann diese Zahl zurück. Ein Aufruf von *next*() liefert als Ergebnis den nächsten String, der durch Leerzeichen, Tabulatoren oder Zeilenumbrüche begrenzt wird. Um das Programm später laufen lassen zu können, müssen also alle arithmetischen Operatoren von Leerzeichen begrenzt werden. Außerdem ist beim Aufruf zu beachten, dass die Eingabe mit einem *End-Of-File*-Zeichen abgeschlossen werden muss. Unter Unix ist dies Ctrl-D, unter Windows wird hierfür Ctrl-Z verwendet.

## 6.6 Nutzen abstrakter Daten-Typen

Wir sind nun in der Lage den Nutzen, den die Verwendung abstrakter Daten-Typen hat, zu erkennen.

1. Abstrakte Daten-Typen machen die Implementierung eines Algorithmus von der Implementierung der Daten-Typen unabhängig.

   Bei der Implementierung des Algorithmus zur Auswertung arithmetischer Ausdrücke mußten wir uns um die zugrunde liegenden Daten-Strukturen nicht weiter kümmern. Es reichte aus, zwei Dinge zu wissen:

   (a) Die Typ-Spezifikationen der verwendeten Funktionen.

   (b) Die Axiome, die das Verhalten dieser Funktionen beschreiben.

   Der abstrakte Daten-Typ ist damit eine *Schnittstelle* zwischen dem Algorithmus einerseits und der Daten-Struktur andererseits. Dadurch ist es möglich, Algorithmus und Daten-Struktur von unterschiedlichen Personen entwickeln zu lassen.

```
113 import java.math.*;
114 import java.io.*;
115 import java.util.*;
116
117 public class MyScanner {
118     private ArrayStack<Object> mTokenStack;
119
120     public MyScanner(InputStream stream) {
121         ArrayList<Object> tokenList = new ArrayList<Object>();
122         System.out.println( "Enter arithmetic expression. " +
123                             "Separate Operators with white space:");
124         Scanner scanner = new Scanner(stream);
125         while (scanner.hasNext()) {
126             if (scanner.hasNextBigInteger()) {
127                 tokenList.add(scanner.nextBigInteger());
128             } else {
129                 tokenList.add(scanner.next());
130             }
131         }
132         mTokenStack = new ArrayStack<Object>();
133         for (int i = tokenList.size() - 1; i >= 0; --i) {
134             mTokenStack.push(tokenList.get(i));
135         }
136     }
137     public ArrayStack<Object> getTokenStack() {
138         return mTokenStack;
139     }
140 }
```

Abbildung 6.8: Die Klasse `MyScanner`

2. Abstrakte Daten-Typen sind *wiederverwendbar*.

   Die Definition des abstrakten Daten-Typs *Stack* ist sehr allgemein. Dadurch ist dieser Daten-Typ vielseitig einsetzbar: Wir werden später noch sehen, wie der ADT *Stack* bei der Traversierung gerichteter Graphen eingesetzt werden kann.

3. Abstrakte Daten-Typen sind *austauschbar*.

   Bei der Auswertung arithmetischer Ausdrücke können wir die Feld-basierte Implementierung des ADT *Stack* mit minimalen Aufwand durch eine Listen-basierte Implementierung ersetzen. Dazu ist lediglich an drei Stellen der Aufruf eines Konstruktors abzuändern. Dadurch wird bei der Programm-Entwicklung das folgende Vorgehen möglich: Wir entwerfen den benötigten Algorithmus auf der Basis abstrakter Daten-Typen. Für diese geben wir zunächst sehr einfache Implementierungen an, deren Effizienz eventuell noch zu wünschen übrig läßt. In einem späteren Schritt wird evaluiert wo der Schuh am meisten drückt. Die ADTs, die bei dieser Evaluierung als performance-kritisch erkannt werden, können anschließend mit dem Ziel der Effizienz-Steigerung reimplementiert werden.

# Kapitel 7

# Mengen und Abbildungen

Wir haben bereits im ersten Semester gesehen, wie wichtig Mengen und Abbildungen in der Informatik sind. In diesem Kapitel zeigen wir, wie sich Mengen und Abbildungen effizient implementieren lassen. Wir können uns auf Abbildungen beschränken, denn eine Menge $M$ läßt sich immer als eine Abbildung $f$ in die Menge $\{\texttt{true}, \texttt{false}\}$ darstellen, wenn wir

$$x \in M \;\Leftrightarrow\; f(x) = \texttt{true}$$

definieren. Der Rest dieses Kapitels ist wie folgt aufgebaut:

1. Zunächst definieren wir einen abstrakten Daten-Typ, der Abbildungen spezifiziert.

   Anschließend stellen wir verschiedene Implementierungen dieses Daten-Typs vor.

2. Wir beginnen mit geordneten binären Bäumen.

3. Da die Komplexität der Implementierung, die auf geordneten binären Bäumen basiert, im schlechtesten Fall linear mit der Anzahl der Enträge wächst, betrachten wir als nächstes *balancierte binäre Bäume*. Bei diesen wächst die Komplexität der Operationen auch im schlechtesten Fall nur logarithmisch mit der Zahl der Einträge.

4. Anschließend betrachten wir die Daten-Struktur der *Tries*, die dann verwendet werden kann, wenn die Schlüssel, nach denen gesucht werden soll, String sind.

5. *Hash-Tabellen* stellen eine weitere Möglichkeit zur Implementierung von Abbildungen dar und werden im vierten Abschnitt diskutiert.

6. Im fünften Abschnitt diskutieren wir, welche vordefinierten Datenstrukturen zur Implementierung von Abbildungen dem Entwickler in der Sprache *Java* zur Verfügung gestellt werden.

7. Im letzten Abschnitt zeigen wir als Anwendung, wie das *Wolf-Ziege-Kohl*-Problem, das wir bereits im ersten Semester diskutiert hatten, in *Java* gelöst werden kann.

## 7.1 Der abstrakte Daten-Typ der *Abbildung*

In vielen Anwendungen der Informatik spielen *Abbildungen* einer Menge von sogenannten *Schlüsseln* in eine Menge von sogenannten *Werten* eine wichtige Rolle. Als ein Beispiel betrachten wir ein elektronisches Telefon-Buch wie es beispielsweise von einem Handy zur Verfügung gestellt wird. Die wesentlichen Funktionen, die ein solches Telefon-Buch anbietet, sind:

1. Nachschlagen eines gegebenen Namens und Ausgabe der diesem Namen zugeordneten Telefon-Nummer.

2. Einfügen eines neuen Eintrags mit Namen und Telefon-Nummer.

3. Löschen eines vorhandenen Eintrags.

Im Falle des Telefon-Buchs sind die *Schlüssel* die Namen und die *Werte* sind die Telefon-Nummern.

**Definition 14 (Abbildung)**
*Wir definieren den abstrakten Daten-Typ der* Abbildung *wie folgt:*

1. *Als Namen wählen wir Map.*
2. *Die Menge der Typ-Parameter ist* $\{\mathit{Key}, \mathit{Value}\}$.
3. *Die Menge der Funktions-Zeichen ist*
   $$\{\mathit{map}, \mathit{find}, \mathit{insert}, \mathit{delete}\}.$$
4. *Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:*
   (a) $\mathit{map} : \mathit{Map}$
   *Der Aufruf* Map() *erzeugt eine leere Abbildung, also eine Abbildung, die keinem Schlüssel einen Wert zuweist.*
   (b) $\mathit{find} : \mathit{Map} \times \mathit{Key} \to \mathit{Value} \cup \{\Omega\}$
   *Der Aufruf* $\mathit{find}(m, k)$ *überprüft, ob in der Abbildung* $m$ *zu dem Schlüssel* $k$ *ein Wert abgespeichert ist. Wenn ja, wird dieser Wert zurück gegeben, sonst wird der Wert* $\Omega$ *zurück gegeben.*
   (c) $\mathit{insert} : \mathit{Map} \times \mathit{Key} \times \mathit{Value} \to \mathit{Map}$
   *Der Aufruf* $\mathit{insert}(m, k, v)$ *fügt in der Abbildung* $m$ *für den Schlüssel* $k$ *den Wert* $v$ *ein. Falls zu dem Schlüssel* $k$ *bereits ein Eintrag in der Abbildung* $m$ *existiert, so wird dieser überschrieben. Andernfalls wird ein entsprechender Eintrag neu angelegt. Als Ergebnis wird die geänderte Abbildung zurück gegeben.*
   (d) $\mathit{delete} : \mathit{Map} \times \mathit{Key} \to \mathit{Map}$
   *Der Aufruf* $\mathit{delete}(m, k)$ *entfernt den Eintrag zu dem Schlüssel* $k$ *in der Abbildung* $m$*. Falls kein solcher Eintrag existiert, bleibt die Abbildung* $m$ *unverändert. Als Ergebnis wird die eventuell geänderte Abbildung zurück gegeben.*
5. *Das genaue Verhalten der Funktionen wird durch die nachfolgenden Axiome spezifiziert.*
   (a) $\mathit{find}(\mathit{map}(), k) = \Omega$,
   *denn der Aufruf* map() *erzeugt eine leere Abbildung, in der sich zu keinem Schlüssel ein Wert findet.*
   (b) $\mathit{find}\big(\mathit{insert}(m, k, v), k\big) = v$,
   *denn wenn wir zu dem Schlüssel* $k$ *einen Wert* $v$ *einfügen, so finden wir anschließend eben diesen Wert* $v$*, wenn wir nach* $k$ *suchen.*
   (c) $k_1 \neq k_2 \to \mathit{find}\big(\mathit{insert}(m, k_1, v), k_2\big) = \mathit{find}(m, k_2)$,
   *denn wenn wir für einen Schlüssel* $k_1$ *eine Wert einfügen, so ändert das nichts an dem Wert, der für einen anderen Schlüssel* $k_2$ *abgespeichert ist.*
   (d) $\mathit{find}\big(\mathit{delete}(m, k), k\big) = \Omega$,
   *denn wenn wir einen Schlüssel* $k$ *löschen, so finden wir anschließend auch keinen Wert mehr, der unter dem Schlüssel* $k$ *abgespeichert ist.*
   (e) $k_1 \neq k_2 \to \mathit{find}\big(\mathit{delete}(m, k_1), k_2\big) = \mathit{find}(m, k_2)$,
   *denn wenn wir einen Schlüssel* $k_1$ *löschen, so ändert das nichts an dem Wert, der unter einem anderen Schlüssel* $k_2$ *abgespeichert ist.* □

Es ist in SetlX sehr einfach, den ADT *Abbildung* zu implementieren. Dazu müssen wir uns nur klar machen, dass Abbildungen nichts anderes sind als Funktionen und die können wir in der Mengenlehre durch binäre Relationen darstellen. Ist $r$ eine binäre Relation die genau ein Paar der Form $[k, v]$ enthält, bei dem die erste Komponente den Wert $k$ hat, dann liefert der Ausdruck

$$r[k]$$

als Ergebnis den Wert $v$. Umgekehrt wird durch den Aufruf

$$r[k] := v$$

das Paar $[k, v]$ in die Relation $r$ eingefügt. Um den Eintrag unter einem Schlüssel $k$ zu löschen, reicht es aus, dem Schlüssel $k$ den undefinierten Wert $\Omega$ zuzuweisen:

$$r[k] := \texttt{om}.$$

Dieser Wert wird auch bei der Auswertung des Ausdrucks $r[k]$ zurück gegeben, wenn die binäre Relation kein Paar der Form $[k, v]$ enthält. Abbildung 7.1 zeigt eine Implementierung des ADT Abbildung, die diese Überlegungen unmittelbar umsetzt.

```
1  map := procedure() {
2      return {};
3  };
4
5  find := procedure(m, k) {
6      return m[k];
7  };
8
9  insert := procedure(m, k, v) {
10     m[k] := v;
11     return m;
12 };
13
14 delete := procedure(m, k) {
15     m[k] := om;
16     return m;
17 };
```

Abbildung 7.1: Eine triviale Implementierung des ADT *Map* in SetlX.

## 7.2 Geordnete binäre Bäume

Falls auf der Menge *Key* der Schlüssel eine totale Ordnung $\leq$ existiert, so kann eine einfache und zumindest im statistischen Durchschnitt effiziente Implementierung des abstrakte Daten-Typs *Map* mit Hilfe *geordneter binärer Bäume* erfolgen. Um diesen Begriff definieren zu können, führen wir zunächst *binäre Bäume* ein.

**Definition 15 (Binärer Baum)**
*Gegeben sei eine Menge Key von Schlüsseln und eine Menge Value von Werten. Dann definieren wir die Menge der binären Bäume $\mathcal{B}$ induktiv mit Hilfe der beiden Funktions-Zeichen Nil und Node, deren Typ-Spezifikationen wie folgt gegeben sind:*

$$\mathit{Nil} : \mathcal{B} \qquad \text{und} \qquad \mathit{Node} : \mathit{Key} \times \mathit{Value} \times \mathcal{B} \times \mathcal{B} \to \mathcal{B}.$$

1. *Nil ist ein binärer Baum.*

   *Dieser Baum wird als der* leere Baum *bezeichnet.*

2. *Node*$(k, v, l, r)$ *ist ein binärer Baum, falls gilt:*

   (a) $k$ *ist ein Schlüssel aus der Menge Key.*

   (b) $v$ *ist ein Wert aus der Menge Value.*

   (c) $l$ *ist ein binärer Baum.*
   $l$ *wird als der* linke Teilbaum *von* *Node*$(k, v, l, r)$ *bezeichnet.*

   (d) $r$ *ist ein binärer Baum.*
   $r$ *wird als der* rechte Teilbaum *von* *Node*$(k, v, l, r)$ *bezeichnet.* □

Als nächstes definieren wir, was wir unter einem *geordneten binären Baum* verstehen.

**Definition 16 (Geordneter binärer Baum)**
*Die Menge* $\mathcal{B}_<$ *der* geordneten binären Bäume *wird induktiv definiert.*

1. *Nil* $\in \mathcal{B}_<$

2. *Node*$(k, v, l, r) \in \mathcal{B}_<$ *falls folgendes gilt:*

   (a) $k$ *ist ein Schlüssel aus der Menge Key.*

   (b) $v$ *ist ein Wert aus der Menge Value.*

   (c) $l$ *und* $r$ *sind geordnete binäre Bäume.*

   (d) *Alle Schlüssel, die in dem linken Teilbaum* $l$ *auftreten, sind kleiner als* $k$*.*

   (e) *Alle Schlüssel, die in dem rechten Teilbaum* $r$ *auftreten, sind größer als* $k$*.*

   *Die beiden letzten Bedingungen bezeichnen wir als die* Ordnungs-Bedingung. □

Geordnete binäre Bäume lassen sich grafisch wir folgt darstellen:

1. Der leere Baum *Nil* wird durch einen dicken schwarzen Punkt dargestellt.

2. Ein Baum der Form *Node*$(k, v, l, r)$ wird dargestellt, indem zunächst ein Oval gezeichnet wird, in dem oben der Schlüssel $k$ und darunter, getrennt durch einen waagerechten Strich, der dem Schlüssel zugeordnete Wert $v$ eingetragen wird. Dieses Oval bezeichnen wir auch als einen *Knoten* des binären Baums. Anschließend wird links unten von diesem Knoten rekursiv der Baum $l$ gezeichnet und rechts unten wird rekursiv der Baum $r$ gezeichnet. Zum Abschluß zeichnen wir von dem mit $k$ und $v$ markierten Knoten jeweils einen Pfeil zu dem linken und dem rechten Teilbaum.

Abbildung 7.2 zeigt ein Beispiel für einen geordneten binären Baum. Der oberste Knoten, in der Abbildung ist das der mit dem Schlüssel 8 und dem Wert 22 markierte Knoten, wird als die *Wurzel* des Baums bezeichnet. Ein *Pfad der Länge* $k$ in dem Baum ist eine Liste $[n_0, n_1, \cdots, n_k]$ von $k+1$ Knoten, die durch Pfeile verbunden sind. Identifizieren wir Knoten mit ihren Markierungen, so ist

$$\big[\langle 8, 22\rangle, \langle 12, 18\rangle, \langle 10, 16\rangle, \langle 9, 39\rangle\big]$$

ein Pfad der Länge 3.

Wir überlegen uns nun, wie wir mit Hilfe geordneter binärer Bäume den ADT *Map* implementieren können. Wir spezifizieren die einzelnen Methoden dieses Daten-Typs durch (bedingte) Gleichungen. Der Konstruktor *map*() liefert als Ergebnis den leeren Baum zurück:

$$\mathit{map}() = \mathit{Nil}.$$

Für die Methode *find*() erhalten wir die folgenden Gleichungen:

1. $\mathit{find}\big(\mathit{Nil}, k\big) = \Omega$,
   denn der leere Baum repräsentiert die leere Abbildung.

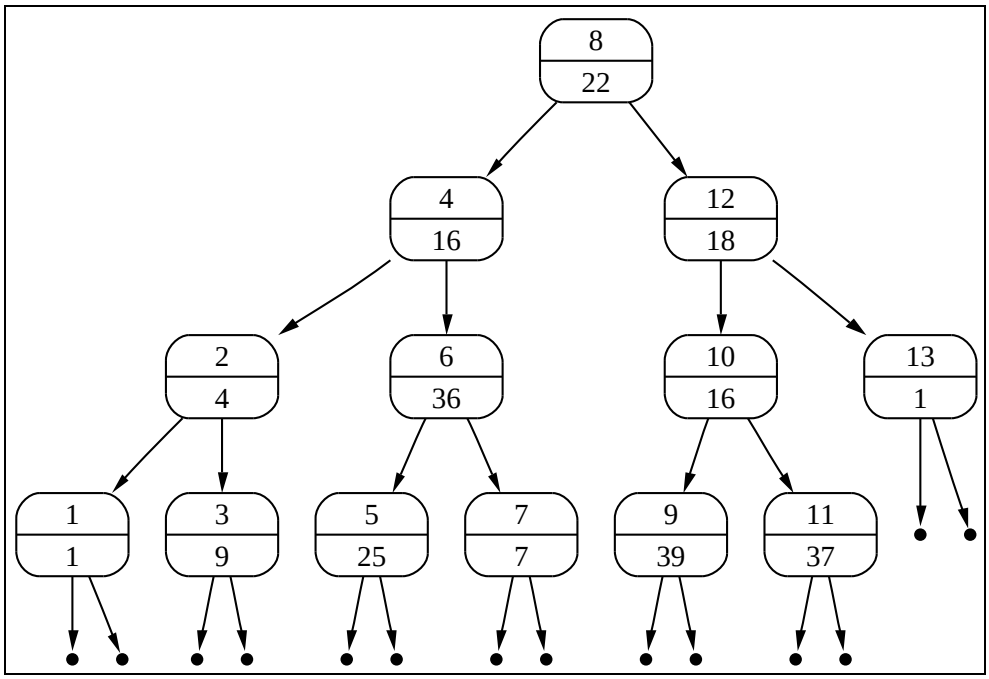


Abbildung 7.2: Ein geordneter binärer Baum

2. $\mathit{find}\big(\mathit{Node}(k, v, l, r), k\big) = v$,
   denn der Knoten $\mathit{Node}(k, v, l, r)$ speichert die Zuordnung $k \mapsto v$.

3. $k_1 < k_2 \rightarrow \mathit{find}\big(\mathit{Node}(k_2, v, l, r), k_1\big) = \mathit{find}(l, k_1)$,
   denn wenn $k_1$ kleiner als $k_2$ ist, dann kann aufgrund der Ordnungs-Bedingung eine Zuordnung für $k_1$ nur in dem linken Teilbaum $l$ gespeichert sein.

4. $k_1 > k_2 \rightarrow \mathit{find}\big(\mathit{Node}(k_2, v, l, r), k_1\big) = \mathit{find}(r, k_1)$,
   denn wenn $k_1$ größer als $k_2$ ist, dann kann aufgrund der Ordnungs-Bedingung eine Zuordnung für $k_1$ nur in dem rechten Teilbaum $r$ gespeichert sein.

Als nächstes definieren wir die Funktion *insert*. Die Definition erfolgt ebenfalls mit Hilfe rekursiver Gleichungen und ist ganz analog zur Definition der Funktion *find*.

1. $\mathit{insert}(\mathit{Nil}, k, v) = \mathit{Node}(k, v, \mathit{Nil}, \mathit{Nil})$,
   denn wenn der Baum vorher leer ist, so kann die einzufügende Information direkt an der Wurzel abgespeichert werden.

2. $\mathit{insert}\big(\mathit{Node}(k, v_2, l, r), k, v_1\big) = \mathit{Node}(k, v_1, l, r)$,
   denn wenn wir den Schlüssel $k$ an der Wurzel finden, überschreiben wir einfach den zugeordneten Wert.

3. $k_1 < k_2 \rightarrow \mathit{insert}\big(\mathit{Node}(k_2, v_2, l, r), k_1, v_1\big) = \mathit{Node}\big(k_2, v_2, \mathit{insert}(l, k_1, v_1), r\big)$,
   denn wenn der Schlüssel $k_1$, unter dem wir Informationen einfügen wollen, kleiner als der Schlüssel $k_2$ an der Wurzel ist, so müssen wir die einzufügende Information im linken Teilbaum einfügen.

4. $k_1 > k_2 \rightarrow \mathit{insert}\big(\mathit{Node}(k_2, v_2, l, r), k_1, v_1\big) = \mathit{Node}\big(k_2, v_2, l, \mathit{insert}(r, k_1, v_1)\big)$,
   denn wenn der Schlüssel $k_1$, unter dem wir Informationen einfügen wollen, größer als der Schlüssel $k_2$ an der Wurzel ist, so müssen wir die einzufügende Information im rechten Teilbaum einfügen.

Als letztes definieren wir die Methode *delete*. Diese Definition ist schwieriger als die Implementierung der andern beiden Methoden. Falls wir in einen Baum der Form $t = \mathit{Node}(k, v, l, r)$ den Eintrag mit dem Schlüssel $k$ löschen wollen, so kommt es auf die beiden Teilbäume $l$ und $r$ an. Ist $l$ der leere Teilbaum, so liefert $\mathit{delete}(t, k)$ als Ergebnis den Teilbaum $r$ zurück. Ist $r$ der leere Teilbaum, so ist das Ergebnis $l$. Problematisch ist die Situation, wenn weder $l$ noch $r$ leer sind. Die Lösung besteht dann darin, dass wir in dem rechten Teilbaum $r$ den Knoten mit dem kleinsten Schlüssel suchen und diesen Knoten aus dem Baum $r$ entfernen. Den dadurch entstehenden Baum nennen wir $r'$. Anschließend überschreiben wir in $t = \mathit{Node}(k, v, l, r')$ die Werte $k$ und $v$ mit dem eben gefundenen kleinsten Schlüssel $k_{min}$ und dem $k_{min}$ zugeordneten Wert $v_{min}$. Der dadurch entstehende binäre Baum $t = \mathit{Node}(k_{min}, v_{min}, l, r')$ ist auch wieder geordnet, denn einerseits ist der Schlüssel $k_{min}$ größer als der Schlüssel $k$ und damit sicher auch größer als alle Schlüssel im linken Teilbaum $l$ und andererseits ist $k_{min}$ kleiner als alle Schlüssel im Teilbaum $r'$ den $k_{min}$ ist ja der kleinste Schlüssel aus $r$.

Zur Veranschaulichung betrachten wir ein Beispiel: Wenn wir in dem Baum aus Abbildung 7.2 den Knoten mit der Markierung $\langle 4, 16\rangle$ löschen wollen, so suchen wir zunächst in dem Teilbaum, dessen Wurzel mit $\langle 6, 36\rangle$ markiert ist, den Knoten, der mit dem kleinsten Schlüssel markiert ist. Dies ist der Knoten mit der Markierung $\langle 5, 25\rangle$. Wir löschen diesen Knoten und überschreiben die Markierung $\langle 4, 16\rangle$ mit der Markierung $\langle 5, 25\rangle$. Abbildung 7.3 auf Seite 93 zeigt das Ergebnis.

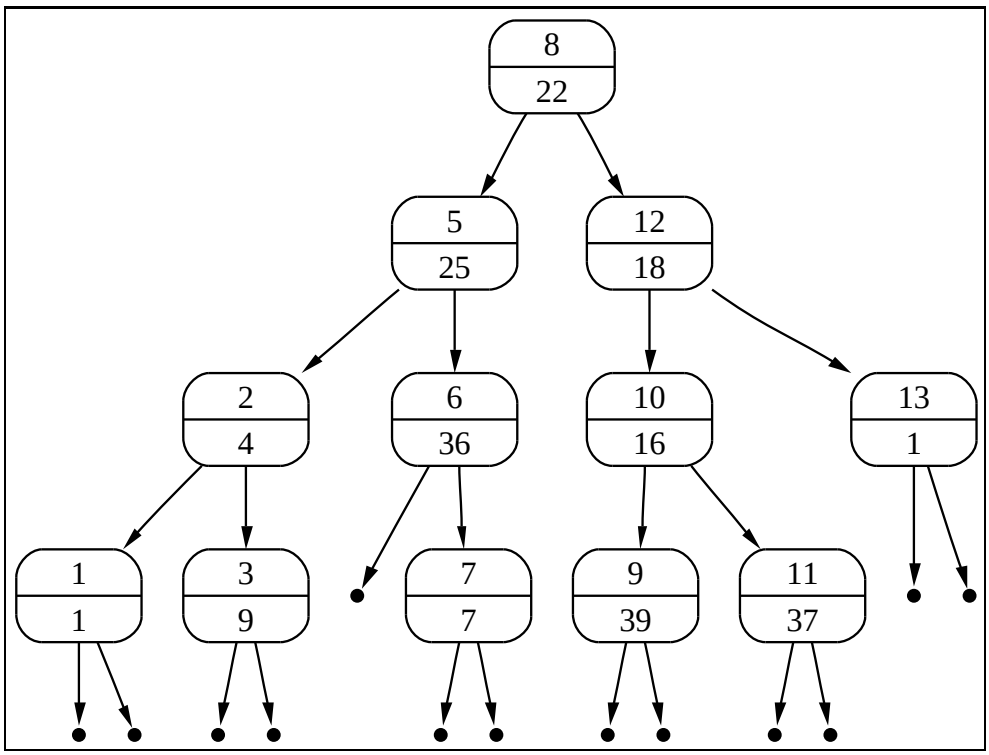


Abbildung 7.3: Der geordnete binärer Baum aus Abbildung 7.2 nach dem Entfernen des Knotens mit der Markierung $\langle 4, 16\rangle$.

Wir geben nun bedingte Gleichungen an, die die Methode *delMin* spezifizieren.

1. $\mathit{delMin}\big(\mathit{Node}(k, v, \mathit{Nil}, r)\big) = [r, k, v]$,

   denn wenn der linke Teilbaum leer ist, muß $k$ der kleinste Schlüssel in dem Baum sein. Wenn wir diesen Schlüssel nebst dem zugehörigen Wert aus dem Baum entfernen, bleibt der rechte Teilbaum übrig.

2. $l \neq \mathit{Nil} \wedge \mathit{delMin}(l) = [l', k_{min}, v_{min}] \rightarrow$
$$\mathit{delMin}\big(\mathit{Node}(k, v, l, r)\big) = [\mathit{Node}(k, v, l', r), k_{min}, v_{min}],$$
denn wenn der linke Teilbaum $l$ in dem binären Baum $t = \mathit{Node}(k, v, l, r)$ nicht leer ist, so muss der kleinste Schlüssel von $t$ in $l$ liegen. Wir entfernen daher rekursiv den kleinsten Schlüssel aus $l$ und erhalten dabei den Baum $l'$. In dem ursprünglich gegebenen Baum $t$ ersetzen wir $l$ durch $l'$ und erhalten $t = \mathit{Node}(k, v, l', r)$.

Damit können wir nun die Methode `delete`() spezifizieren.

1. $\mathit{delete}(\mathit{Nil}, k) = \mathit{Nil}$.
2. $\mathit{delete}\big(\mathit{Node}(k, v, \mathit{Nil}, r), k\big) = r$.
3. $\mathit{delete}(\mathit{Node}\big(k, v, l, \mathit{Nil}), k\big) = l$.
4. $l \neq \mathit{Nil} \wedge r \neq \mathit{Nil} \wedge \mathit{delMin}(r) = [r', k_{min}, v_{min}] \rightarrow$
$$\mathit{delete}\big(\mathit{Node}(k, v, l, r), k\big) = \mathit{Node}(k_{min}, v_{min}, l, r').$$
Falls der zu entfernende Schlüssel mit dem Schlüssel an der Wurzel des Baums übereinstimmt, entfernen wir mit dem Aufruf $r.\mathit{delMin}()$ den kleinsten Schlüssel aus dem rechten Teilbaum $r$ und produzieren dabei den Baum $r'$. Gleichzeitig berechnen wir dabei für den rechten Teilbaum den kleinsten Schlüssel $k_{min}$ und den diesem Schlüssel zugeordneten Wert $v_{min}$. Diese Werte setzen wir nun an die Wurzel des neuen Baums.
5. $k_1 < k_2 \rightarrow \mathit{delete}\big(\mathit{Node}(k_2, v_2, l, r), k_1\big) = \mathit{Node}\big(k_2, v_2, \mathit{delete}(l, k_1), r\big)$,
Falls der zu entfernende Schlüssel kleiner ist als der Schlüssel an der Wurzel, so kann sich der Schlüssel nur im linken Teilbaum befinden. Daher wird der Schlüssel $k_1$ rekursiv in dem linken Teilbaum $l$ entfernt.
6. $k_1 > k_2 \rightarrow \mathit{delete}(\mathit{Node}\big(k_2, v_2, l, r), k_1\big) = \mathit{Node}\big(k_2, v_2, l, \mathit{delete}(r, k_1)\big)$,
denn in diesem Fall kann sich der Eintrag mit dem Schlüssel $k_1$ nur in dem rechten Teilbaum $r$ befinden.

### 7.2.1 Implementierung geordneter binärer Bäume in SetlX

Die Abbildungen 7.4 und 7.5 zeigen die Implementierung geordneter binärer Bäume in SetlX. Die Idee ist, den leeren Baum *Nil* durch den Term `Nil`() darzustellen, während ein binärer Baum der Form $\mathit{Node}(k, v, l, r)$ durch den Term `Node`$(k, v, l, r)$ dargestellt wird. Die Sprache SetlX kann generell ein Term mit Funktions-Zeichen $F$ und Argumenten $t_1, \cdots, t_n$ als $F(t_1, \cdots, t_n)$ dargestellt werden. Dabei ist zu beachten, dass Funktions-Zeichen in SetlX groß geschrieben werden müssen, denn nur so können Terme syntaktisch von Funktions-Aufrufen unterschieden werden. Soll doch einmal ein Funktionszeichen verwendet werden, das mit einem kleinen Buchstaben beginnt, so muss dem Funktionszeichen das Zeichen "`@`" vorangestellt werden.

Die Abbildungen 7.4 enthält die Implementierung der Funktionen *map*(), $\mathit{find}(m, k_1)$ und $\mathit{insert}(m, k, v)$, die wir jetzt im Detail diskutieren.

1. Die Funktion gibt den Term `Nil`() zurück, der dem leeren Baum entspricht und eine leere Abbildung darstellt.
2. Die Funktion `find`$(m, k_1, \mathit{cmp})$ bekommt drei Argumente:
   (a) $m$ ist der binäre Baum und
   (b) $k_1$ ist der Schlüssel, der in diesem Baum gesucht werden soll.
   (c) *cmp* ist eine Funktion, mit deren Hilfe sich verschiedene Schlüssel vergleichen lassen. Diese wird in der Form $\mathit{cmp}(k_1, k_2)$ aufgerufen und entscheidet, ob der Schlüssel $k_1$ kleiner ist als der Schlüssel $k_2$.

```
1  map := procedure() {
2      return Nil();
3  };
4
5  find := procedure(m, k1, cmp) {
6      match(m) {
7          case Nil() : return;
8          case Node(k2, v, l, r) :
9               if (k1 == k2) {
10                  return v;
11              } else if (cmp(k1, k2)) {
12                  return find(l, k1, cmp);
13              } else {
14                  return find(r, k1, cmp);
15              }
16     }
17 };
18
19 insert := procedure(m, k1, v1, cmp) {
20     match(m) {
21         case Nil() : return Node(k1, v1, Nil(), Nil());
22         case Node(k2, v2, l, r) :
23              if (k1 == k2) {
24                  return Node(k1, v1, l, r);
25              } else if (cmp(k1, k2)) {
26                  return Node(k2, v2, insert(l, k1, v1, cmp), r);
27              } else {
28                  return Node(k2, v2, l, insert(r, k1, v1, cmp));
29              }
30     }
31 };
```

Abbildung 7.4: Implementierung geordneter Bäume in SetlX, 1. Teil.

Die Implementierung der Funktion `find`() erfolgt über einen `match`-Befehl:

(a) Falls der Baum $m$ leer ist, kann der Aufruf `find`$(m, k_1, cmp)$ offenbar nicht erfolgreich sein und es wird durch ein `return` der Wert $\Omega$ zurück gegeben.

(b) Andernfalls hat $m$ die Form `Node`$(k_2, v, l, r)$. Hier gibt es nun drei Möglichkeiten:

- Falls $k_1 = k_2$ ist, war die Suche erfolgreich und es wird der an der Wurzel des Baums gespeicherte Wert $v$ zurück gegeben.
- Falls $k_1 < k_2$ ist, muss die Suche im linken Teilbaum $l$ fortgesetzt werden.
- Andernfalls muss $k_1 > k_2$ gelten und die Suche ist im rechten Teilbaum $r$ fortzusetzen.

3. Die Funktion `insert`$(m, k_1, v_1, cmp)$ bekommt vier Argumente:

(a) $m$ ist der binäre Baum,

(b) $k_1$ ist der Schlüssel unter dem der Wert

(c) $v_1$ in dem Baum $m$ eingefügt werden soll.

(d) $cmp$ ist die Funktion zum Vergleichen zweier Schlüssel.

Die Implementierung der Funktion `insert`() erfolgt ebenfalls über eine Fallunterscheidung, die durch einen `match`-Befehl realisiert wird. `match`-Befehl:

(a) Falls der Baum $m$ leer ist, wird als Ergebnis der Term $\texttt{Node}(k_1, v_1, \texttt{Nil}(), \texttt{Nil}())$ zurück gegeben.

(b) Andernfalls hat $m$ die Form $\texttt{Node}(k_2, v_2, l, r)$. Hier gibt es nun wieder drei Möglichkeiten:

- Falls $k_1 = k_2$ ist, wird der Wert $v_2$ der momentan ander Wurzel des Baums steht, durch den einzufügenden Wert $v_1$ ersetzt.
- Falls $k_1 < k_2$ ist, wird die Einfügung rekursiv im linken Teilbaum $l$ durchgeführt.
- Andernfalls muss $k_1 > k_2$ gelten und der Wert wird im rechten Teilbaum eingefügt.

Die Implementierung der restlichen in Abbildung 7.5 gezeigten Funktionen *delMin*$(m)$ und *delete*$(m, k_1, cmp)$ verläuft analog und wird daher nicht im Detail diskutiert. Beachten Sie, dass die resultierende Implementierung die im vorigen Abschnitt hergeleiteten Gleichungen 1-zu-1 umsetzt.

```
32  delMin := procedure(m) {
33      match (m) {
34          case Node(k, v, Nil(), r):
35               return [ r, k, v ];
36          case Node(k, v, l, r):
37               [ ls, km, vm ] := delMin(l);
38               return [ Node(k, v, ls, r), km, vm ];
39      }
40  };
41
42  delete := procedure(m, k1, cmp) {
43      match (m) {
44          case Nil():
45               return Nil();
46          case Node(k2, v2, l, r):
47               if (k1 == k2) {
48                   if (l == Nil()) {
49                       return r;
50                   } else if (r == Nil()) {
51                       return l;
52                   }
53                   [ rs, km, vm ] := delMin(r);
54                   return Node(km, vm, l, rs);
55               } else if (cmp(k1, k2)) {
56                   return Node(k2, v2, delete(l, k1, cmp), r);
57               } else {
58                   return Node(k2, v2, l, delete(r, k1, cmp));
59               }
60      }
61  };
```

Abbildung 7.5: Implementierung geordneter Bäume in SetlX, 2. Teil.

### 7.2.2 Analyse der Komplexität

Wir untersuchen zunächst die Komplexität der Funktion *find* im schlechtesten Fall. Dieser Fall tritt dann ein, wenn der binäre Baum zu einer Liste entartet. Abbildung 7.6 zeigt den geordneten

binären Baum der dann entsteht, wenn die Paare aus Schlüssel und Werten aus der Abbildung 7.2 in aufsteigender Reihenfolge eingegeben werden. Wird hier nach dem größten Schlüssel gesucht, so muß der komplette Baum durchlaufen werden. Enthält der Baum $n$ Schlüssel, so sind also insgesamt $n$ Vergleiche erforderlich. In diesem Fall ist ein geordneter binärer Baum also nicht besser als eine Liste.

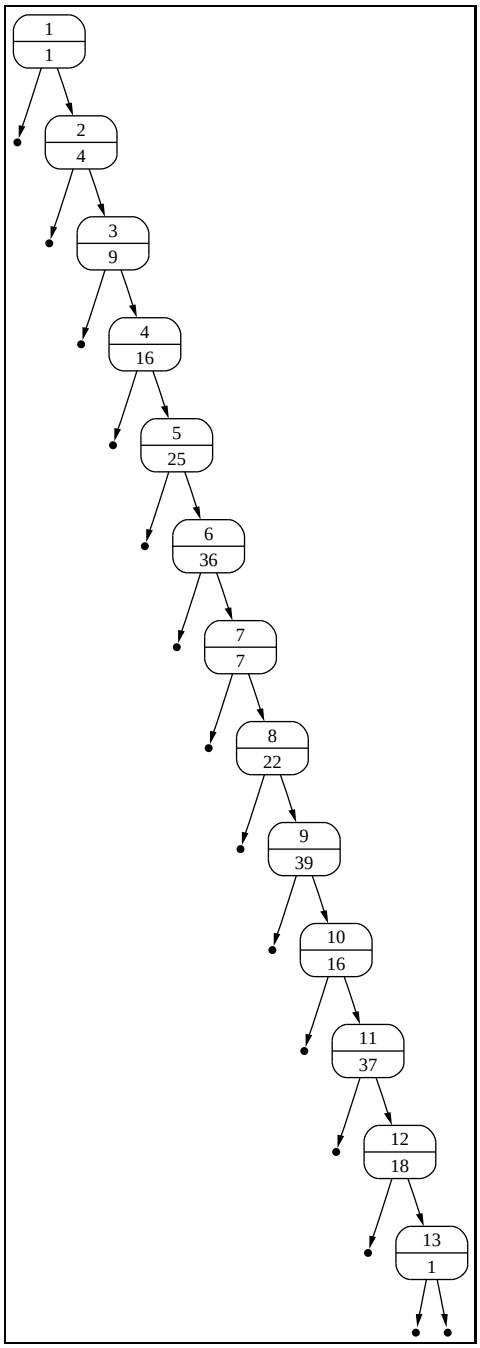


Abbildung 7.6: Ein entarteter geordneter binärer Baum.

Erfreulicherweise tritt der schlechteste Fall im statistischen Durchschnitt selten auf. Im Durchschnitt ist ein zufällig erzeugter binärer Baum recht gut balanciert, so dass beispielsweise für einen Aufruf von `find()` für einen Baum mit $n$ Schlüsseln durchschnittlich $\mathcal{O}\big(\ln(n)\big)$ Vergleiche

erforderlich sind. Wir werden diese Behauptung nun beweisen.

Wir bezeichnen die durchschnittliche Anzahl von Vergleichen, die beim Aufruf $b.\mathit{find}(k)$ für einen geordneten binären Baum $b$ durchgeführt werden müssen, falls $b$ insgesamt $n$ Schlüssel enthält, mit $d_n$. Wir wollen annehmen, dass der Schlüssel $k$ auch wirklich in $b$ zu finden ist. Unser Ziel ist es, für $d_n$ eine Rekurrenz-Gleichung aufzustellen. Zunächst ist klar, dass

$$d_1 = 1$$

ist, denn wenn der Baum $b$ nur einen Schlüssel enthält, wird genau einen Vergleich durchgeführt. Wir betrachten nun einen binären Baum $b$, der $n+1$ Schlüssel enthält. Dann hat $b$ die Form

$$b = \mathit{node}(k', v, l, r).$$

Ordnen wir die $n+1$ Schlüssel der Größe nach in der Form

$$k_0 < k_1 < \cdots < k_i < k_{i+1} < k_{i+2} < \cdots < k_{n-1} < k_n,$$

so gibt es $n+1$ verschiedene Positionen, an denen der Schlüssel $k'$ auftreten kann. Wenn $k' = k_i$ ist, so enthält der linke Teilbaum $i$ Schlüssel und der rechte Teilbaum enthält $n-i$ Schlüssel:

$$\underbrace{k_0 < k_1 < \cdots < k_{i-1}}_{\text{Schlüssel in } l} < \underbrace{k_i}_{\substack{\shortparallel \\ k'}} < \underbrace{k_{i+1} < \cdots < k_{n-1} < k_n}_{\text{Schlüssel in } r},$$

Da $b$ insgesamt $n+1$ Schlüssel enthält, gibt es $n+1$ Möglichkeiten, wie die verbleibenden $n$ Schlüssel auf die beiden Teilbäume $l$ und $r$ verteilt sein können, denn $l$ kann $i$ Schlüssel enthalten, wobei

$$i \in \{0, 1, \cdots, n\}$$

gilt. Entsprechend enthält $r$ dann $n-i$ Schlüssel. Bezeichnen wir die durchschnittliche Anzahl von Vergleichen in einem Baum mit $n+1$ Schlüsseln, dessen linker Teilbaum $i$ Elemente hat, mit

$$\mathit{anzVgl}(i,\, n{+}1),$$

so gilt

$$d_{n+1} = \sum_{i=0}^{n} \frac{1}{n+1} \cdot \mathit{anzVgl}(i,\, n{+}1),$$

denn wir haben ja angenommen, dass alle Werte von $i$ die gleiche Wahrscheinlichkeit, nämlich $\frac{1}{n+1}$, haben.

Berechnen wir nun den Wert von $\mathit{anzVgl}(i, n+1)$: Falls $l$ aus $i$ Schlüsseln besteht und die restlichen $n-i$ Schlüssel in $r$ liegen, so gibt es für den Schlüssel $k$, nach dem wir in dem Aufruf $b.\mathit{find}(k)$ suchen, 3 Möglichkeiten:

1. $k$ kann mit dem Schlüssel $k'$ an der Wurzel des Baums übereinstimmen. In diesem Fall führen wir nur einen Vergleich durch. Da es insgesamt $n+1$ Schlüssel in dem Baum gibt und nur in einem dieser Fälle der Schlüssel, den wir suchen, an der Wurzel steht, hat dieser Fall die Wahrscheinlichkeit
   $$\frac{1}{n+1}.$$
2. $k$ kann mit einem der $i$ Schlüssel im linken Teilbaum $l$ übereinstimmen. Da der linke Teilbaum $i$ Schlüssel enthält und es insgesamt $n+1$ Schlüssel gibt, hat die Wahrscheinlichkeit, dass $k$ in dem linken Teilbaum $l$ auftritt, den Wert
   $$\frac{i}{n+1}.$$
   In diesem Fall werden
   $$d_i + 1$$
   Vergleiche durchgeführt, denn außer den $d_i$ Vergleichen mit den Schlüsseln aus dem linken

Teilbaum muss der Schlüssel, der gesucht wird, ja noch mit dem Schlüssel an der Wurzel verglichen werden.

3. $k$ kann mit einem der $n-i$ Schlüssel im rechten Teilbaum $r$ übereinstimmen.

   Da der rechte Teilbaum $n-i$ Schlüssel enthält und es insgesamt $n+1$ Schlüssel gibt, hat die Wahrscheinlichkeit, dass $k$ in dem rechten Teilbaum $r$ auftritt, den Wert

   $$\frac{n-i}{n+1}.$$

   Analog zum zweiten Fall werden diesmal

   $$d_{n-i}+1$$

   Vergleiche durchgeführt.

Um nun $anzVgl(i, n{+}1)$ berechnen zu können, müssen wir in jedem der drei Fälle die Wahrscheinlichkeit mit der Anzahl der Vergleiche multiplizieren und die Werte, die sich für die drei Fälle ergeben, aufsummieren. Wir erhalten

$$\begin{aligned}
anzVgl(i,n{+}1) &= \frac{1}{n+1}\cdot 1 + \frac{i}{n+1}\cdot(d_i+1) + \frac{n-i}{n+1}\cdot(d_{n-i}+1) \\
&= \frac{1}{n+1}\cdot\big(1+i\cdot(d_i+1)+(n-i)\cdot(d_{n-i}+1)\big) \\
&= \frac{1}{n+1}\cdot\big(1+i+(n-i)+i\cdot d_i+(n-i)\cdot d_{n-i}\big) \\
&= \frac{1}{n+1}\cdot\big(n+1+i\cdot d_i+(n-i)\cdot d_{n-i}\big) \\
&= 1+\frac{1}{n+1}\cdot\big(i\cdot d_i+(n-i)\cdot d_{n-i}\big)
\end{aligned}$$

Damit können wir nun die Rekurrenz-Gleichung für $d_{n+1}$ aufstellen:

$$\begin{aligned}
d_{n+1} &= \sum_{i=0}^{n}\frac{1}{n+1}\cdot anzVgl(i,\, n{+}1) \\
&= \frac{1}{n+1}\cdot\sum_{i=0}^{n}\left(1+\frac{1}{n+1}\cdot\big(i\cdot d_i+(n-i)\cdot d_{n-i}\big)\right) \\
&= \frac{1}{n+1}\cdot\left(\underbrace{\sum_{i=0}^{n}1}_{\;=\;n+1}+\frac{1}{n+1}\cdot\sum_{i=0}^{n}\big(i\cdot d_i+(n-i)\cdot d_{n-i}\big)\right) \\
&= 1+\frac{1}{(n+1)^2}\cdot\left(\sum_{i=0}^{n}\big(i\cdot d_i+(n-i)\cdot d_{n-i}\big)\right) \\
&= 1+\frac{2}{(n+1)^2}\cdot\sum_{i=0}^{n}i\cdot d_i
\end{aligned}$$

Bei der letzten Umformung haben wir die Gleichung (5.4)

$$\sum_{i=0}^{n}f(n-i)=\sum_{i=0}^{n}f(i)$$

benutzt, die wir bei der Analyse der Komplexität von Quick-Sort gezeigt hatten. Wir lösen jetzt

die Rekurrenz-Gleichung

$$d_{n+1} = 1 + \frac{2}{(n+1)^2} \cdot \sum_{i=0}^{n} i \cdot d_i \tag{7.1}$$

mit der Anfangs-Bedingungen $d_1 = 1$. Zur Lösung von Gleichung (7.1) führen wir die Substitution $n \mapsto n+1$ durch und erhalten

$$d_{n+2} = 1 + \frac{2}{(n+2)^2} \cdot \sum_{i=0}^{n+1} i \cdot d_i \tag{7.2}$$

Wir multiplizieren nun Gleichung (7.1) mit $(n+1)^2$ und Gleichung (7.2) mit $(n+2)^2$ und finden die Gleichungen

$$(n+1)^2 \cdot d_{n+1} = (n+1)^2 + 2 \cdot \sum_{i=0}^{n} i \cdot d_i, \tag{7.3}$$

$$(n+2)^2 \cdot d_{n+2} = (n+2)^2 + 2 \cdot \sum_{i=0}^{n+1} i \cdot d_i \tag{7.4}$$

Subtrahieren wir Gleichung (7.3) von Gleichung (7.4), so erhalten wir

$$(n+2)^2 \cdot d_{n+2} - (n+1)^2 \cdot d_{n+1} = (n+2)^2 - (n+1)^2 + 2 \cdot (n+1) \cdot d_{n+1}.$$

Zur Vereinfachung substituieren wir hier $n \mapsto n-1$ und erhalten

$$(n+1)^2 \cdot d_{n+1} - n^2 \cdot d_n = (n+1)^2 - n^2 + 2 \cdot n \cdot d_n.$$

Dies vereinfachen wir zu

$$(n+1)^2 \cdot d_{n+1} = n \cdot (n+2) \cdot d_n + 2 \cdot n + 1.$$

Bei dieser Gleichung teilen wir auf beiden Seiten durch $(n+2) \cdot (n+1)$ und bekommen

$$\frac{n+1}{n+2} \cdot d_{n+1} = \frac{n}{n+1} \cdot d_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)}.$$

Nun definieren wir

$$c_n = \frac{n}{n+1} \cdot d_n.$$

Dann gilt $c_1 = \frac{1}{2} \cdot d_1 = \frac{1}{2}$ und wir haben die Rekurrenz-Gleichung

$$c_{n+1} = c_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)}.$$

Durch Partialbruch-Zerlegung finden wir

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

Also haben wir

$$c_{n+1} = c_n + \frac{3}{n+2} - \frac{1}{n+1}.$$

Wegen $c_1 = \frac{1}{2}$ ist die Gleichung auch für $n = 0$ richtig, wenn wir $c_0 = 0$ setzen, denn es gilt

$$\frac{1}{2} = 0 + \frac{3}{0+2} - \frac{1}{0+1}.$$

Die Rekurrenz-Gleichung für $c_n$ können wir mit dem Teleskop-Verfahren lösen:

$$c_{n+1} = c_0 + \sum_{i=0}^{n} \frac{3}{i+2} - \sum_{i=0}^{n} \frac{1}{i+1}$$

$$= \sum_{i=2}^{n+2} \frac{3}{i} - \sum_{i=1}^{n+1} \frac{1}{i}.$$

Wir substituieren $n \mapsto n-1$ und vereinfachen dies zu

$$c_n = \sum_{i=2}^{n+1} \frac{3}{i} - \sum_{i=1}^{n} \frac{1}{i}$$

Die harmonische Zahl $H_n$ ist als

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

definiert. Wir können $c_n$ auf $H_n$ zurückführen:

$$\begin{aligned} c_n &= 3 \cdot H_n - \frac{3}{1} + \frac{3}{n+1} - H_n \\ &= 2 \cdot H_n - 3 \cdot \frac{n}{n+1} \end{aligned}$$

Wegen $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln(n) + \mathcal{O}(1)$ gilt dann

$$c_n = 2 \cdot \ln(n) + \mathcal{O}(1).$$

Berücksichtigen wir $d_n = \frac{n+1}{n} \cdot c_n$, so finden wir für große $n$ ebenfalls

$$d_n = 2 \cdot \ln(n) + \mathcal{O}\big(1\big).$$

Das ist unser zentrales Ergebnis: Im Durchschnitt erfordert das Suchen in einem zufällig erzeugten geordneten binären Baum für große Werte von $n$ etwa

$$2 \cdot \ln(n) = 2 \cdot \ln(2) \cdot \log_2(n) \approx 1.386 \cdot \log_2(n)$$

Vergleiche. Damit werden etwa 39 % mehr Vergleiche ausgeführt als bei einem optimal balancierten binären Baum. Ähnliche Ergebnisse können wir für das Einfügen oder Löschen erhalten.

## 7.3 AVL-Bäume

Es gibt verschiedene Varianten von geordneten binären Bäumen, bei denen auch im schlechtesten Fall die Anzahl der Vergleiche nur logarithmisch von der Zahl der Schlüssel abhängt. Eine solche Variante sind die *AVL-Bäume* [AVL62], die nach ihren Erfindern G. M. Adel'son-Vel'skiĭ und E. M. Landis benannt sind. Diese Variante stellen wir jetzt vor. Dazu definieren wir zunächst die *Höhe* eines binären Baums:

1. $\mathit{height}(\mathit{Nil}) = 0$.
2. $\mathit{height}\big(\mathit{Node}(k, v, l, r)\big) = \max\big(\mathit{height}(l), \mathit{height}(r)\big) + 1$.

**Definition 17 (AVL-Baum)**
*Wir definieren die Menge $\mathcal{A}$ der* AVL-Bäume *induktiv:*

1. $\mathit{Nil} \in \mathcal{A}$.
2. $\mathit{Node}(k, v, l, r) \in \mathcal{A}$ *g.d.w.*
   (a) $\mathit{Node}(k, v, l, r) \in \mathcal{B}_<$,
   (b) $l, r \in \mathcal{A}$ *und*
   (c) $|\mathit{height}(l) - \mathit{height}(r)| \leq 1$.
   *Diese Bedingungen bezeichnen wir auch als die* Balancierungs-Bedingung.

*AVL-Bäume sind also geordnete binäre Bäume, für die sich an jedem Knoten $\mathit{Node}(k, v, l, r)$ die Höhen der Teilbäume $l$ und $r$ maximal um 1 unterscheiden.* □

Um AVL-Bäume zu implementieren, können wir auf unserer Implementierung der geordneten binären Bäume aufsetzen. Neben den Methoden, die wir schon aus der Klasse *Map* kennen, brauchen wir noch die Methode

$$\mathit{restore} : \mathcal{B}_< \to \mathcal{A},$$

mit der wir die Bedingung über den Höhenunterschied von Teilbäumen wiederherstellen können, wenn diese beim Einfügen oder Löschen eines Elements verletzt wird. Der Aufruf $\mathit{restore}(b)$ setzt voraus, dass $b$ ein geordneter binärer Baum ist, für den außer an der Wurzel überall die Balancierungs-Bedingung erfüllt ist. An der Wurzel kann die Höhe des linken Teilbaums um maximal 2 von der Höhe des rechten Teilbaums abweichen. Beim Aufruf der Methode $\mathit{restore}(b)$ liegt also einer der beiden folgenden Fälle vor:

1. $b = \mathit{Nil}$ oder
2. $b = \mathit{Node}(k, v, l, r) \wedge l \in \mathcal{A} \wedge r \in \mathcal{A} \wedge |\mathit{height}(l) - \mathit{height}(r)| \leq 2$.

Wir spezifizieren die Methode *restore*() durch bedingte Gleichungen.

1. $\mathit{restore}(\mathit{Nil}) = \mathit{Nil}$,
   denn der leere Baum ist ein AVL-Baum.
2. $|\mathit{height}(l) - \mathit{height}(r)| \leq 1 \rightarrow \mathit{restore}\big(\mathit{Node}(k, v, l, r)\big) = \mathit{Node}(k, v, l, r)$,
   denn wenn die Balancierungs-Bedingung bereits erfüllt ist, braucht nichts getan werden.
3. $$\begin{array}{ll} & \mathit{height}(l_1) = \mathit{height}(r_1) + 2 \\ \wedge & l_1 = \mathit{Node}(k_2, v_2, l_2, r_2) \\ \wedge & \mathit{height}(l_2) \geq \mathit{height}(r_2) \\ \rightarrow & \mathit{restore}\big(\mathit{Node}(k_1, v_1, l_1, r_1)\big) = \mathit{Node}\big(k_2, v_2, l_2, \mathit{Node}(k_1, v_1, r_2, r_1)\big) \end{array}$$
   Um diese Gleichung zu verstehen, betrachten wir Abbildung 7.7 auf Seite 103. Der linke Teil der Abbildung beschreibt die Situation vor dem Ausbalancieren, es wird also der Baum

$$Node(k_1, v_1, Node(k_2, v_2, l_2, r_2), r_1)$$

dargestellt. Der rechte Teil der Abbildung zeigt das Ergebnis des Ausbalancierens, es wird also der Baum

$$Node\big(k_2, v_2, l_2, Node(k_1, v_1, r_2, r_1)\big)$$

dargestellt. Wir haben hier die Höhen der einzelnen Teilbäume jeweils in die zweiten Zeilen der entsprechenden Markierungen geschrieben. Hier ist $h$ die Höhe des Teilbaums $l_2$. Der Teilbaum $r_1$ hat die Höhe $h-1$. Der Teilbaum $r_2$ hat die Höhe $h'$ und es gilt $h' \leq h$. Da $r_2$ ein AVL-Baum ist, gilt also entweder $h' = h$ oder $h' = h-1$.

Die gezeigte Situation kann entstehen, wenn im linken Teilbaum $l_2$ ein Element eingefügt wird oder wenn im rechten Teilbaum $r_1$ eine Element gelöscht wird.

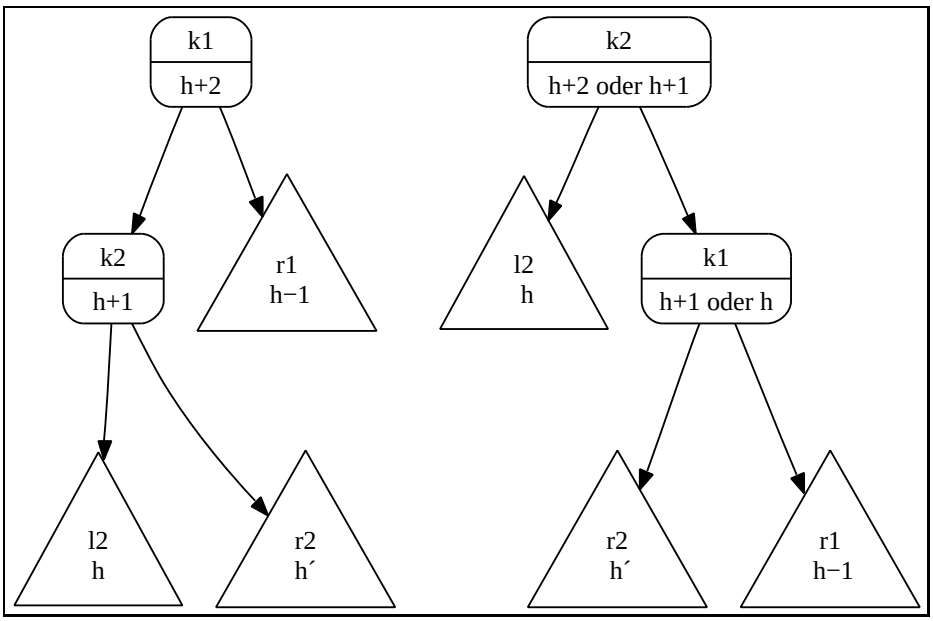


Abbildung 7.7: Ein unbalancierter Baum und der rebalancierte Baum

Wir müssen uns davon überzeugen, dass der im rechten Teil von Abbildung 7.7 gezeigte Baum auch tatsächlich ein AVL-Baum ist. Was die Balancierungs-Bedingung angeht, so rechnet man dies sofort nach. Die Tatsache, dass der mit $k_1$ markierte Knoten entweder die Höhe $h$ oder $h+1$ hat folgt daraus, dass $r_1$ die Höhe $h-1$ hat und dass $h' \in \{h, h-1\}$ gilt.

Um zu sehen, dass der Baum geordnet ist, können wir folgende Ungleichung hinschreiben:

$$l_2 < k_2 < r_2 < k_1 < r_1. \qquad (\star)$$

Dabei schreiben wir für einen Schlüssel $k$ und einen Baum $b$

$$k < b$$

um auzudrücken, dass $k$ kleiner ist als alle Schlüssel, die in dem Baum $b$ vorkommen. Analog schreiben wir $b < k$ wenn alle Schlüssel, die in dem Baum $b$ vorkommen, kleiner sind als der Schlüssel $k$. Die Ungleichung $(\star)$ beschreibt die Anordnung der Schlüssel sowohl für den im linken Teil der Abbildung gezeigten Baum als auch für den Baum im rechten Teil der Abbildung und damit sind beide Bäume geordnet.

4. $\quad height(l_1) = height(r_1) + 2$
$\wedge \quad l_1 = Node(k_2, v_2, l_2, r_2)$
$\wedge \quad height(l_2) < height(r_2)$
$\wedge \quad r_2 = Node(k_3, v_3, l_3, r_3)$
$\rightarrow \quad restore\big(Node(k_1, v_1, l_1, r_1)\big) = Node\big(k_3, v_3, Node(k_2, v_2, l_2, l_3), Node(k_1, v_1, r_3, r_1)\big)$

Die linke Seite der Gleichung wird durch die Abbildung 7.8 auf Seite 104 illustriert. Dieser Baum kann in der Form

$$Node\big(k_1, v_1, Node(k_2, v_2, l_2, Node\big(k_3, v_3, l_3, r_3)\big), r_1\big)$$

geschrieben werden. Die Teilbäume $l_3$ und $r_3$ haben hier entweder die Höhe $h$ oder $h-1$, wobei mindestens einer der beiden Teilbäume die Höhe $h$ haben muß.

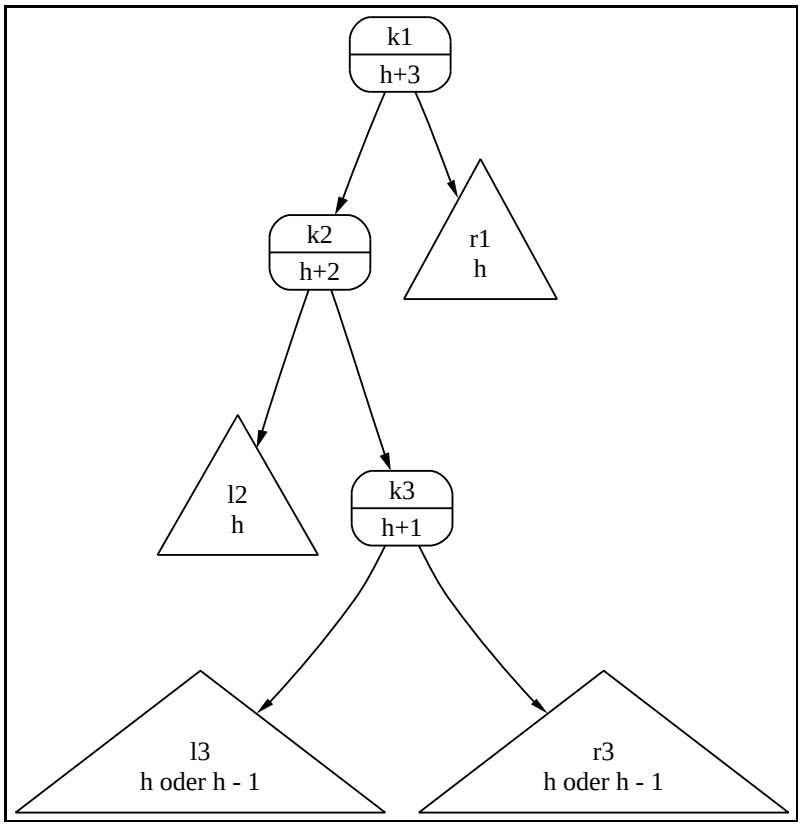


Abbildung 7.8: Ein unbalancierter Baum: 2. Fall

Die Situation der rechten Seite der obigen Gleichung zeigt Abbildung 7.9 auf Seite 105. Der auf dieser Abbildung gezeigte Baum hat die Form

$$Node\big(k_3, v_3, Node(k_2, v_2, l_2, l_3), Node(k_1, v_1, r_3, r_1)\big).$$

Die Ungleichung, die die Anordnung der Schlüssel sowohl im linken als auch rechten Baum wieder gibt, lautet

$$l_2 < k_2 < l_3 < k_3 < r_3 < k_1 < r_1.$$

Es gibt noch zwei weitere Fälle die auftreten, wenn der rechte Teilbaum um mehr als Eins größer ist als der linke Teilbaum. Diese beiden Fälle sind aber zu den beiden vorherigen Fällen völlig analog, so dass wir die Gleichungen hier ohne weitere Diskussion angeben.

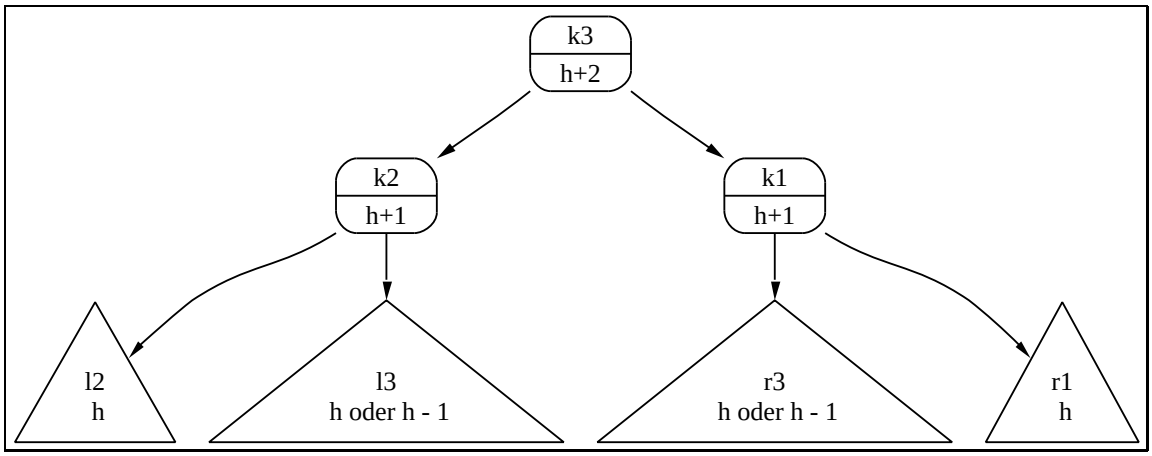


Abbildung 7.9: Der rebalancierte Baum im 2. Fall
1

5. $\quad height(r_1) = height(l_1) + 2$
   $\wedge \quad r_1 = Node(k_2, v_2, l_2, r_2)$
   $\wedge \quad height(r_2) \geq height(l_2)$
   $\rightarrow \quad restore\big(Node(k_1, v_1, l_1, r_1)\big) = Node\big(k_2, v_2, Node(k_1, v_1, l_1, l_2), r_2\big)$

6. $\quad height(r_1) = height(l_1) + 2$
   $\wedge \quad r_1 = Node(k_2, v_2, l_2, r_2)$
   $\wedge \quad height(r_2) < height(l_2)$
   $\wedge \quad l_2 = Node(k_3, v_3, l_3, r_3)$
   $\rightarrow \quad restore\big(Node(k_1, v_1, l_1, r_1)\big) = Node\big(k_3, v_3, Node(k_1, v_1, l_1, l_3), Node(k_2, v_2, r_3, r_2)\big)$

Damit können wir nun die Methode *insert*() durch bedingte rekursive Gleichungen beschreiben. Dabei müssen wir die ursprünglich für geordnete Bäume angegebene Gleichungen dann ändern, wenn die Balancierungs-Bedingung durch das Einfügen eines neuen Elements verletzt werden kann.

1. $insert(Nil, k, v) = Node(k, v, Nil, Nil)$.
2. $insert\big(Node(k, v_2, l, r), k, v_1\big) = Node(k, v_1, l, r)$.
3. $k_1 < k_2 \rightarrow insert\big(Node(k_2, v_2, l, r), k_1, v_1\big) = restore\big(Node\big(k_2, v_2, insert(l, k_1, v_1), r\big)\big)$.
4. $k_1 > k_2 \rightarrow insert\big(Node(k_2, v_2, l, r), k_1, v_1\big) = restore\big(Node\big(k_2, v_2, l, insert(r, k_1, v_1)\big)\big)$.

Analog ändern sich die Gleichungen für *delMin*() wie folgt:

1. $delMin\big(Node(k, v, Nil, r)\big) = \langle r, k, v\rangle$.
2. $l \neq Nil \wedge delMin(l) = \langle l', k_{min}, v_{min}\rangle \rightarrow$
   $delMin\big(Node(k, v, l, r)\big) = \langle restore(Node(k, v, l', r)), k_{min}, v_{min}\rangle$.

Damit können wir die Gleichungen zur Spezifikation der Methode `delete`() angeben.

1. $delete(Nil, k) = Nil$.
2. $delete\big(Node(k, v, Nil, r), k\big) = r$.
3. $delete\big(Node(k, v, l, Nil), k\big) = l$.
4. $l \neq Nil \wedge r \neq Nil \wedge delMin(r) = \langle r', k_{min}, v_{min}\rangle \rightarrow$
   $delete\big(Node(k, v, l, r), k\big) = restore\big(Node(k_{min}, v_{min}, l, r')\big)$.

5. $k_1 < k_2 \rightarrow \mathit{delete}\big(\mathit{Node}(k_2, v_2, l, r), k_1\big) = \mathit{restore}\big(\mathit{Node}\big(k_2, v_2, \mathit{delete}(l, k_1), r\big)\big)$.

6. $k_1 > k_2 \rightarrow \mathit{delete}\big(\mathit{Node}(k_2, v_2, l, r), k_1\big) = \mathit{restore}\big(\mathit{Node}\big(k_2, v_2, l, \mathit{delete}(r, k_1)\big)\big)$.

### 7.3.1 Implementierung von AVL-Bäumen in SetlX

Wollen wir AVL-Bäume in SetlX implementieren, so müssen wir uns überlegen, wie wir die Höhe der Bäume abspeichern wollen, denn es wäre ineffizient, wenn wir diese Höge jedesmal neu berechnen wollten. Die einzige Möglichkeit ist, diese Höhe in den Termen selbst zu speichern. Wir stellen daher einen Baum der Höhe $h$, an dessen Wurzel der Schlüssel $k$ mit dem Wert $v$ gespeichert ist, durch den Term

$$\texttt{Node}(k, v, l, r, h)$$

dar, wobei $l$ und $r$ den linken bzw. rechten Teilbaum bezeichnen. Damit erhalten wir die in Abbildung 7.10 gezeigte Implementierung der Funktion *insert*$(b, k_1, v_1, \mathit{cmp})$, deren Aufgabe es ist, in dem AVL-Baum $b$ den Wert $v_1$ unter dem Schlüssel $k_1$ einzufügen. Das letzte Argument *cmp* ist eine Funktion mit deren Hilfe Schlüssel verglichen werden können.

```
1   insert := procedure(b, k1, v1, cmp) {
2       match (b) {
3           case Nil():
4                return Node(k1, v1, Nil(), Nil(), 1);
5           case Node(k2, v2, l, r, h):
6                if (k1 == k2) {
7                    return Node(k1, v1, l, r, h);
8                } else if (cmp(k1, k2)) {
9                    return restore(mkNode(k2, v2, insert(l, k1, v1, cmp), r));
10               } else {
11                   return restore(mkNode(k2, v2, l, insert(r, k1, v1, cmp)));
12               }
13          default: abort("Error in insert($b$, $k1$, $v1$, $cmp$)");
14      }
15  };
```

Abbildung 7.10: Implementierung der Funktion *insert*$(b, k, v, \mathit{cmp})$.

Gegenüber der in Abbildung 7.4 gibt es die folgenden Änderungen:

1. In Zeile 4 erhält der Term *Node*() die Höhe 1 als zusätzliches Argument.
2. In Zeile 5 und 7 hat der Term *Node*() ebenfalls die Höhe $h$ als zusätzliches Argument.
3. Bei der Berechnung der neuen Knoten in Zeile 9 und 11 rufen wir nun die Hilfsfunktion *mkNode*() auf, deren Implementierung in Abbildung 7.11 gezeigt wird. Diese Hilfsfunktion hat die Aufgabe, die Höhe des Baums zu berechnen und diese Information in dem erzeugten Knoten zu speichern. Auf den von *mkNode*() erzeugten Knoten müssen wir dann noch die Funktion *restore*() anwenden, denn durch das Einfügen im linken oder rechten Teilbaum wird unter Umständen die Balancierungs-Bedingung verletzt.

Abbildung 7.12 zeigt die Implementierung der Funktion *delMin*(). In dem Matching-Befehl haben wir das letzte Argument von *Node*() durch das *Wildcard-Zeichen* "_" ersetzt, das beim Matchen dann verwendet wird, wenn der Wert des zugehörigen Arguments irrelevant ist. In Zeile 7 war es wieder erforderlich, die Funktion *restore*() aufzurufen, denn durch das Löschen des minimalen Elements in dem linken Teilbaum $l$ kann es passieren, dass dessen Höhe schrumpft wodurch eventuell die Balancierungs-Bedingung verletzt wird.

```
1   height := procedure(b) {
2       match (b) {
3           case Nil()              : return 0;
4           case Node(k, v, l, r, h): return h;
5           default: abort("Error in height($b$)");
6       }
7   };
8
9   mkNode := procedure(k, v, l, r) {
10      return Node(k, v, l, r, max([height(l), height(r)]) + 1);
11  };
```

Abbildung 7.11: Implementierung der Funktionen *height*$(b)$ und *mkNode*$(k, v, l, r)$.

```
1   delMin := procedure(b) {
2       match (b) {
3           case Node(k, v, Nil(), r, _):
4                return [ r, k, v ];
5           case Node(k, v, l, r, _):
6                [ ls, km, vm ] := delMin(l);
7                return [ restore(mkNode(k, v, ls, r)), km, vm ];
8           default: abort("Error in delMin($b$)");
9       }
10  };
```

Abbildung 7.12: Implementierung der Funktion *delMin*$(b)$.

Abbildung 7.13 zeigt die Implementierung der Funktion *delete*(). Jedesmal, wenn ein Knoten zurück gegeben wird, rufen wir die Funktion *restore*() auf um sicherzustellen, dass die Balancierungs-Bedingung erhalten bleibt.

Abbildung 7.14 zeigt schließlich die Implementierung der Funktion *restore*().

1. In Zeile behandeln wir den trivialen Fall, dass der Baum leer ist.

2. In Zeile 7 testen wir, ob die Balancierungs-Bedingung verletzt ist. Wenn dies nicht der Fall ist, reicht es aus, die Höhe neu zu berechnen. Dies erreichen wir durch den Aufruf von *mkNode*() in Zeile 8.

3. Falls die Balancierungs-Bedingung verletzt ist, prüfen wir in Zeile 10, ob die Höhe des linken Teilbaums größer ist als die Höhe des rechten Teilbaums. Wenn dies so ist, hat der Baum die Form

   $$\texttt{Node}(k_1, v_1, \texttt{Node}(k_2, v_2, l_2, r_2), r_1)$$

   Dann sind zwei Unterfälle zu betrachten:

   (a) Die Höhe von $l_2$ ist größer oder gleich der Höhe von $r_2$. In diesem Fall hat der Ergebnis-Baum die Form

   $$\texttt{Node}(k_2, v_2, l_2, \texttt{Node}(k_2, v_1, r_2, r_1)).$$

   Dieser Baum wird in Zeile 14 zurück gegeben.

   (b) Die Höhe von $l_2$ ist kleiner als die Höhe von $r_2$. Dann hat $r_2$ die Form

```
1  delete := procedure(b, k1, cmp) {
2      match (b) {
3          case Nil(): return Nil();
4          case Node(k2, v2, l, r, _):
5               if (k1 == k2) {
6                   if (l == Nil()) {
7                       return r;
8                   } else if (r == Nil()) {
9                       return l;
10                  }
11                  [ rs, km, vm ] := delMin(r);
12                  return restore(mkNode(km, vm, l, rs));
13              } else if (cmp(k1, k2)) {
14                  return restore(mkNode(k2, v2, delete(l, k1, cmp), r));
15              } else {
16                  assert(cmp(k2,k1), "$k2$ < $k1$");
17                  return restore(mkNode(k2, v2, l, delete(r, k1, cmp)));
18              }
19         default: abort("Error in delete($b$)");
20     }
21 };
```

Abbildung 7.13: Implementierung der Funktion $delete(b, k, cmp)$.

$$r_2 = \texttt{Node}(k_3, v_3, l_3, r_3).$$

In diesem Fall hat der Ergebnis-Baum die Form

$$\texttt{Node}(k_3, v_3, l_2, \texttt{Node}(k_2, v_2, l_2, l3), \texttt{Node}(k_1, v_1, r_3, r_1)).$$

Dieser Baum wird in Zeile 18 zurück gegeben.

4. Der Fall, dass die Balancierungs-Bedingung verletzt ist und die Höhe des rechten Teilbaums größer ist als die Höhe des linken Teilbaums, ist analog zum letzten Fall.

```
restore := procedure(b) {
  match (b) {
    case Nil(): return Nil();
    case Node(k1, v1, l1, r1, _):
         hl1 := height(l1);
         hr1 := height(r1);
         if (abs(hl1 - hr1) <= 1) {
             return mkNode(k1, v1, l1, r1);
         }
         if (hl1 == hr1 + 2) {
             match (l1) {
                 case Node(k2, v2, l2, r2, _):
                      if (height(l2) >= height(r2)) {
                          return mkNode(k2, v2, l2, mkNode(k1, v1, r2, r1));
                      }
                      match (r2) {
                          case Node(k3, v3, l3, r3, _):
                               return mkNode(k3, v3, mkNode(k2, v2, l2, l3),
                                                     mkNode(k1, v1, r3, r1));
                          default: abort("Error in restore($r2$)");
                      }
                 default: abort("Error in restore($l1$)");
             }
         }
         if (hr1 == hl1 + 2) {
             match (r1) {
                 case Node(k2, v2, l2, r2, _):
                      if (height(r2) >= height(l2)) {
                          return mkNode(k2, v2, mkNode(k1, v1, l1, l2), r2);
                      }
                      match (l2) {
                          case Node(k3, v3, l3, r3, _):
                               return mkNode(k3, v3, mkNode(k1, v1, l1, l3),
                                                     mkNode(k2, v2, r3, r2));
                          default: abort("Error in restore($l2$)");
                      }
                 default: abort("Error in restore($r1$)");
             }
         }
    default: abort("Error in restore($b$)");
  }
};
```

Abbildung 7.14: Implementierung der Funktion *restore*().

### 7.3.2 Analyse der Komplexität

Wir analysieren jetzt die Komplexität von AVL-Bäumen im schlechtesten Fall. Der schlechteste Fall tritt dann ein, wenn bei einer vorgegebenen Zahl von Schlüsseln die Höhe maximal wird. Das ist aber das selbe wie wenn in einem Baum gegebener Höhe die Zahl der Schlüssel minimal wird. Wir definieren daher $b_h(k)$ als einen AVL-Baum der Höhe $h$, der unter allen AVL-Bäumen der Höhe $h$ die minimale Anzahl von Schlüsseln hat. Außerdem sollen alle Schlüssel, die in $b_h(k)$ auftreten, größer als der Schlüssel $k$ sein. Sowohl die Schlüssel als auch die Werte sind in diesem Zusammenhang eigentlich unwichtig, wir müssen nur darauf achten, dass die Ordnungs-Bedingung für binäre Bäume erfüllt ist. Wir werden für die Schlüssel natürliche Zahlen nehmen, für die Werte nehmen wir immer die Zahl 0. Bevor wir mit der Definition von $b_h(k)$ beginnen können, benötigen wir noch eine Hilfs-Funktion $maxKey()$ mit der Signatur

$$maxKey : \mathcal{B}_< \to Key$$

Für einen gegebenen geordneten nicht-leeren binären Baum $b$ berechnet $b.maxKey()$ den größten Schlüssel, der in $b$ auftritt. Die Definition von $b.maxKey()$ ist induktiv:

1. $Node(k, v, l, Nil).maxKey() = k$,
2. $r \neq Nil \to Node(k, v, l, r).maxKey() = r.maxKey()$.

Damit können wir nun die Bäume $b_h(k)$ durch Induktion nach der Höhe $h$ definieren.

1. $b_0(k) = Nil$,

   denn es gibt genau einen AVL-Baum der Höhe 0 und dieser enthält keinen Schlüssel.

2. $b_1(k) = Node(k + 1, 0, Nil, Nil)$,

   denn es gibt genau einen AVL-Baum der Höhe 1.

3. $b_{h+1}(k).maxKey() = l \to b_{h+2}(k) = Node\big(l + 1,\, 0,\, b_{h+1}(k),\, b_h(l + 1)\big)$,

   denn um einen AVL-Baum der Höhe $h + 2$ mit einer minimalen Anzahl an Schlüsseln zu konstruieren, erzeugen wir zunächst den AVL-Baum $b_{h+1}(k)$ der Höhe $h+1$. Dann bestimmen wir den maximalen Schlüssel $l$ in diesem Baum, der Schlüssel $l+1$ kommt nun an die Wurzel des zu erzeugenden Baums der Höhe $h + 2$ und schließlich erzeugen wir noch den Baum $b_h(l + 1)$ der Höhe $h$, den wir als rechten Teilbaum in den neu zu erzeugenden Baum der Höhe $h + 2$ einfügen.

Für einen beliebigen binären Baum $b$ bezeichne $\# b$ die Anzahl der Schlüssel, die in $b$ auftreten. Dann definieren wir

$$c_h := \# b_h(k)$$

als die Anzahl der Schlüssel des Baums $b_h(k)$. Wir werden sofort sehen, dass $\# b_h(k)$ nicht von $k$ abhängt. Für $c_h$ finden wir in Analogie zu der induktiven Definition von $b_h(k)$ die folgenden Gleichungen.

1. $c_0 = \# b_0(k) = \# Nil = 0$,
2. $c_1 = \# b_1(k) = \# Node(k + 1, 0, Nil, Nil) = 1$,
3. $$\begin{array}{rcl} c_{h+2} & = & \# b_{h+2}(k) \\ & = & \# Node\big(l + 1,\, 0,\, b_{h+1}(k),\, b_h(l + 1)\big) \\ & = & \# b_{h+1}(k) + \# b_h(l + 1) + 1 \\ & = & c_{h+1} + c_h + 1. \end{array}$$

Also haben wir zur Bestimmung von $c_h$ die Rekurrenz-Gleichung

$$c_{h+2} = c_{h+1} + c_h + 1 \quad \text{mit den Anfangs-Bedingungen } c_0 = 0 \text{ und } c_1 = 1$$

zu lösen. Das ist eine Rekurrenz-Gleichung, die wir, allerdings mit leicht veränderten Anfangs-Bedingungen, bereits im dritten Kapitel gelöst haben. Sie können leicht nachrechnen, dass die Lösung dieser Rekurrenz-Gleichung wie folgt lautet:

$$c_h = \frac{1}{\sqrt{5}} \left(\lambda_1^{h+2} - \lambda_2^{h+2}\right) - 1 \quad \text{mit}$$

$$\lambda_1 = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62 \quad \text{und} \quad \lambda_2 = \frac{1}{2}(1 - \sqrt{5}) \approx -0.62.$$

Da $|\lambda_2| < 1$ ist, spielt der Wert $\lambda_2^{h+2}$ für große Werte von $h$ praktisch keine Rolle und die minimale Zahl $n$ der Schlüssel in einem Baum der Höhe $h$ ist durch

$$n \approx \frac{1}{\sqrt{5}} \lambda_1^{h+2} - 1$$

gegeben. Um diese Gleichung nach $h$ aufzulösen, bilden wir auf beiden Seiten den Logarithmus zur Basis 2. Dann erhalten wir

$$\log_2(n+1) = (h+2) \cdot \log_2(\lambda_1) - \tfrac{1}{2} \cdot \log_2(5)$$

Daraus folgt nach Addition von $\frac{1}{2} \cdot \log_2(5)$

$$\log_2(n+1) + \tfrac{1}{2} \cdot \log_2(5) = (h+2) \cdot \log_2(\lambda_1)$$

Jetzt teilen wir durch $\log_2(\lambda_1)$. Dann erhalten wir

$$\frac{\log_2(n+1) + \frac{1}{2} \cdot \log_2(5)}{\log_2(\lambda_1)} = h + 2$$

Lösen wir diese Gleichung nach $h$ auf, so haben wir für große $n$ das Ergebnis

$$h = \frac{\log_2(n+1) + \frac{1}{2} \cdot \log_2(5)}{\log_2(\lambda_1)} - 2 = \frac{1}{\log_2(\lambda_1)} \cdot \log_2(n) + \mathcal{O}(1) \approx 1,44 \cdot \log_2(n) + \mathcal{O}(1)$$

gewonnen. Die Größe $h$ gibt aber die Zahl der Vergleiche an, die wir im ungünstigsten Fall bei einem Aufruf von *find* in einem AVL-Baum mit $n$ Schlüsseln durchführen müssen. Wir sehen also, dass bei einem AVL-Baum auch im schlechtesten Fall die Komplexität logarithmisch bleibt. Abbildung 7.15 zeigt einen AVL-Baum der Höhe 6, für den das Verhältnis von Höhe zur Anzahl der Knoten maximal wird. Wie man sieht ist auch dieser Baum noch sehr weit weg von dem zur Liste entarteten Baum aus der Abbildung 7.6.

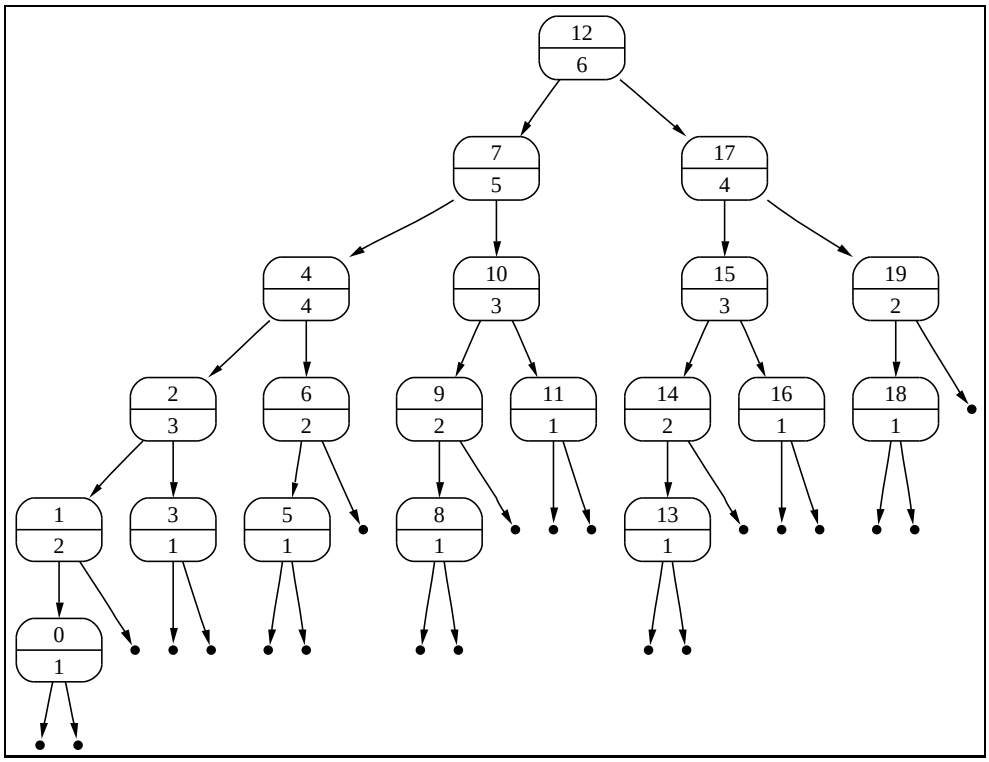


Abbildung 7.15: Ein AVL-Baum mit dem ungünstigsten Verhältnis von Höhe zur Anzahl an Knoten

## 7.4 Tries

In der Praxis kommt es häufig vor, dass die Schlüssel des ADT *Map* Strings sind. In dem einführenden Beispiel des elektronischen Telefon-Buchs ist dies der Fall. Es gibt eine Form von Such-Bäumen, die auf diese Situation besonders angepaßt ist. Diese Such-Bäume haben den Namen *Tries*. Dieses Wort ist von dem Englischen Wort *retrieval* abgeleitet. Damit man *Tries* und *Trees* unterscheiden kann, wird *Trie* so ausgesprochen, dass es sich mit dem Englischen Wort *pie* reimt. Diese Datenstruktur wurde 1959 von René de la Briandais [dlB59] vorgeschlagen.

Die Grundidee bei der Datenstruktur *Trie* ist ein Baum, an dem jeder Knoten nicht nur zwei Nachfolger hat, wie das bei binären Bäumen der Fall ist, sondern statt dessen potentiell für jeden Buchstaben des Alphabets einen Ast besitzt. Um Tries definieren zu können, nehmen wir zunächst an, dass folgendes gegeben ist:

1. $\Sigma$ ist eine endliche Menge, deren Elemente wir als *Buchstaben* bezeichnen. $\Sigma$ selbst heißt das *Alphabet*.

2. $\Sigma^*$ bezeichnet die Menge der *Wörter* (engl. *strings*), die wir aus den Buchstaben des Alphabets bilden können. Mathematisch können wir Wörter als Listen von Buchstaben auffassen. Ist $w \in \Sigma^*$ so schreiben wir $w = cr$, falls $c$ der erste Buchstabe von $w$ ist und $r$ das Wort ist, das durch Löschen des ersten Buchstabens aus $w$ entsteht.

3. $\varepsilon$ bezeichnet das leere Wort.

4. *Value* ist eine Menge von *Werten*.

Die Menge $\mathbb{T}$ der Tries definieren wir nun induktiv mit Hilfe des Konstruktors

$$\texttt{Node} : \mathit{Value} \times \mathit{List}(\Sigma) \times \mathit{List}(\mathbb{T}) \to \mathbb{T}.$$

Die induktive Definition besteht nur aus einer einzigen Klausel: Falls

1. $v \in \mathit{Value} \cup \{\Omega\}$

2. $C = [c_1, \cdots, c_n] \in \mathit{List}(\Sigma)$ eine Liste von Buchstaben der Länge $n$ ist,

3. $T = [t_1, \cdots, t_n] \in \mathit{List}(\mathbb{T})$ eine Liste von Tries der selben Länge $n$ ist,

dann gilt

$$\texttt{Node}(v, C, T) \in \mathbb{T}.$$

Als erstes fragen Sie sich vermutlich, wo bei dieser induktiven Definition der Induktions-Anfang ist. Der Induktions-Anfang ist der Fall $n = 0$, denn dann sind die Listen $L$ und $T$ leer. Als nächstes überlegen wir uns, welche Funktion von dem Trie

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]) \in \mathbb{T}$$

dargestellt wird. Wir beantworten diese Frage, indem wir rekursive Gleichungen für die Methode

$$\mathit{find} : \mathbb{T} \times \Sigma^* \to \mathit{Value} \cup \{\Omega\}$$

angeben. Wir werden den Ausdruck $\mathit{find}\big(\texttt{Node}(v, L, T), s\big)$ durch Induktion über den String $s$ definieren:

1. $\mathit{find}\big(\texttt{Node}(v, C, T), \varepsilon\big) = v$.

   Der dem leeren String zugeordnete Wert wird also unmittelbar an der Wurzel des Tries abgespeichert.

2. $\mathit{find}\big(\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]), cr\big) = \begin{cases} \mathit{find}(t_1, r) & \text{falls} \quad c = c_1; \\ \vdots & \\ \mathit{find}(t_i, r) & \text{falls} \quad c = c_i; \\ \vdots & \\ \mathit{find}(t_n, r) & \text{falls} \quad c = c_n; \\ \Omega & \text{falls} \quad c \notin \{c_1, \cdots, c_n\}. \end{cases}$

Der Trie $\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$ enthält also genau dann einen Wert zu dem Schlüssel $cr$, wenn es in der Liste $[c_1, \cdots, c_n]$ eine Position $i$ gibt, so dass der Buchstabe $c$ mit dem Buchstaben $c_i$ übereinstimmt wenn außerdem der Trie $t_i$ einen Wert zu dem Schlüssel $r$ enthält.

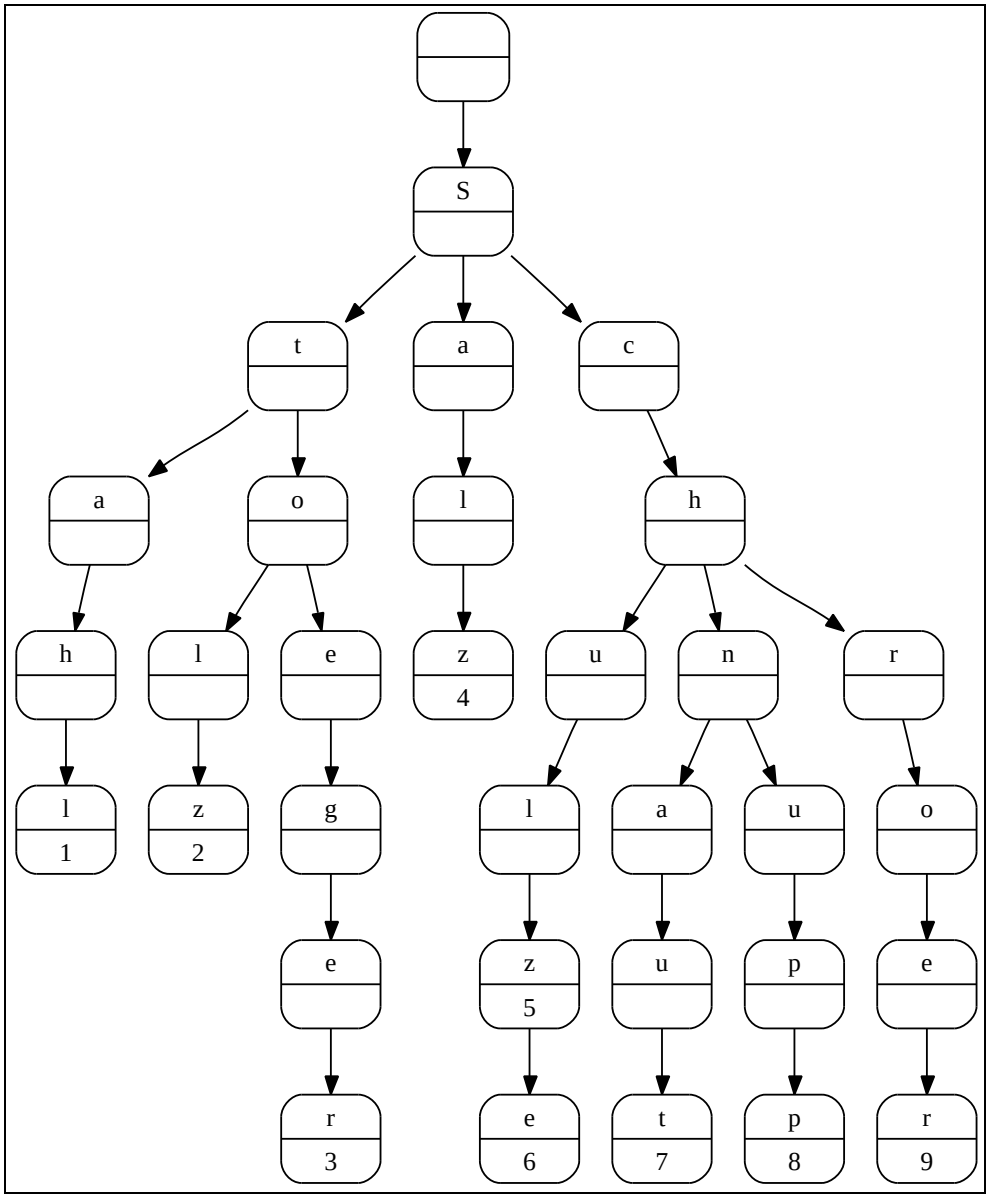


Abbildung 7.16: Ein Beispiel Trie

Zum besseren Verständnis wollen wir Tries graphisch als Bäume darstellen. Nun ist es nicht

sinnvoll, die Knoten dieser Bäume mit langen Listen zu beschriften. Wir behelfen uns mit einem Trick. Um einen Knoten der Form

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$$

darzustellen, zeichnen wir einen Kreis, den wir durch einen horizontalen Strich in der Mitte aufteilen. Falls $v$ von $\Omega$ verschieden ist, schreiben wir den Wert $v$ in die untere Hälfte des Kreises. Das, was wir über dem Strich schreiben, hängt von dem Vater des jeweiligen Knotens ab. Wie genau es vom Vater abhängt, sehen wir gleich. Der Knoten selber hat $n$ Kinder. Diese $n$ Kinder sind die Wurzeln der Bäume, die die Tries $t_1$, $\cdots$, $t_n$ darstellen. Außerdem markieren wir die diese Knoten darstellenden Kreise in den oberen Hälfte mit den Buchstaben $c_1$, $\cdots$, $c_n$.

Zur Verdeutlichung geben wir ein Beispiel in Abbildung 7.16 auf Seite 114. Die Funktion, die hier dargestellt wird, läßt sich wie folgt als binäre Relation schreiben:

$$\big\{\langle \text{“Stahl”}, 1\rangle, \langle \text{“Stolz”}, 2\rangle, \langle \text{“Stoeger”}, 3\rangle, \langle \text{“Salz”}, 4\rangle, \langle \text{“Schulz”}, 5\rangle,$$
$$\langle \text{“Schulze”}, 6\rangle, \langle \text{“Schnaut”}, 7\rangle, \langle \text{“Schnupp”}, 8\rangle, \langle \text{“Schroer”}, 9\rangle\big\}.$$

Der Wurzel-Knoten ist hier leer, denn dieser Knoten hat keinen Vater-Knoten, von dem er eine Markierung erben könnte. Diesem Knoten entspricht der Term

$$\texttt{Node}(\Omega, [\text{‘S’}], [t]).$$

Dabei bezeichnet $t$ den Trie, dessen Wurzel mit dem Buchstaben ‘S’ markiert ist. Diesen Trie können wir seinerseits durch den Term

$$\texttt{Node}(\Omega, [\text{‘t’}, \text{‘a’}, \text{‘c’}], [t_1, t_2, t_3])$$

darstellen. Daher hat dieser Knoten drei Söhne, die mit den Buchstaben ‘t’, ‘a’ und ‘c’ markiert sind.

### 7.4.1 Einfügen in Tries

Wir stellen nun bedingte Gleichungen auf, mit denen wir das Einfügen eines Schlüssels mit einem zugehörigen Wert beschreiben können. Bezeichnen wir die Methode für das Einfügen mit *insert*(), so hat diese Methode die Signatur

$$\mathit{insert} : \mathbb{T} \times \Sigma^* \times V \to \mathbb{T}.$$

Wir definieren den Wert von

$$\mathit{insert}\big(\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]), s, v\big)$$

für ein Wort $w \in \Sigma^*$ und einen Wert $v \in V$ durch Induktion nach der Länge des Wortes $w$.

1. $\mathit{insert}\big(\texttt{Node}(v_1, L, T), \varepsilon, v_2\big) = \texttt{Node}(v_2, L, T)$,

   Einfügen eines Wertes mit dem leeren String als Schlüssel überschreibt einfach den an dem Wurzel-Knoten gespeicherten Wert.

2. $\mathit{insert}\big(\texttt{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big), c_i r, v_2\big) =$

   $$\texttt{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, \mathit{insert}(t_i, r, v_2), \cdots, t_n]\big).$$

   Wenn in dem Trie $\texttt{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big)$ ein Wert $v_2$ zu dem Schlüssel $cr$ eingefügt werden soll, und falls der Buchstabe $c$ in der Liste $[c_1, \cdots, c_n]$ an der Stelle $i$ vorkommt, wenn also gilt $c = c_i$, dann muß der Wert $v_2$ rekursiv in dem Trie $t_i$ unter dem Schlüssel $r$ eingefügt werden.

3. $c \notin \{c_1, \cdots, c_n\} \;\rightarrow\; \mathit{insert}\big(\texttt{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big), cr, v_2\big) =$

   $$\texttt{Node}\big(v_1, [c_1, \cdots, c_n, c], [t_1, \cdots, t_n, \mathit{insert}(\texttt{Node}(\Omega, [], []), r, v_2)]\big).$$

   Wenn in dem Trie $\texttt{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big)$ ein Wert $v_2$ zu dem Schlüssel $cr$ eingefügt werden soll, und falls der Buchstabe $c$ in der Liste $[c_1, \cdots, c_n]$ nicht vorkommt, dann wird

zunächst ein Trie erzeugt, der die leere Abbildung repräsentiert. Dieser Trie hat die Form

$$\texttt{Node}(\Omega, [], []).$$

Anschließend wird in diesem Trie der Wert $v_2$ rekursiv unter dem Schlüssel $r$ eingefügt. Zum Schluß hängen wir den Buchstaben $c$ an die Liste $[c_1, \cdots, c_n]$ an und fügen den Trie

$$\mathit{insert}(\texttt{Node}(\Omega, [], []), r, v_2)$$

am Ende der Liste $[t_1, \cdots, t_n]$ ein.

### 7.4.2 Löschen in Tries

Als letztes stellen wir die bedingten Gleichungen auf, die das Löschen von Schlüsseln und den damit verknüpften Werten in einem Trie beschreiben. Um diese Gleichungen einfacher schreiben zu können, definieren wir zunächst eine Hilfs-Funktion

$$\mathit{isEmpty} : \mathbb{T} \to \mathbb{B},$$

so dass $t.\mathit{isEmpty}()$ genau dann `true` liefert, wenn der Trie $t$ die leere Funktion darstellt. Wir definieren also:

1. $\mathit{isEmpty}\big(\texttt{Node}(\Omega, [], [])\big) = \texttt{true}$
2. $v \neq \Omega \rightarrow \mathit{isEmpty}\big(\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])\big) = \texttt{false}$
3. $\mathit{isEmpty}\big(\texttt{Node}(\Omega, L, T)\big) = \mathit{isEmptyList}(T)$

In der letzten Gleichung haben wir eine weitere Hilfs-Funktion benutzt, die wir noch definieren müssen. Die Funktion

$$\mathit{isEmptyList} : \mathit{List}(\mathbb{T}) \to \mathbb{B}$$

prüft für eine gegebene Liste von Tries, ob alle in der Liste vorhandenen Tries leer sind. Die Definition dieser Funktion erfolgt durch Induktion über die Länge der Liste.

1. $\mathit{isEmptyList}\big([]\big) = \texttt{true}$,
2. $\mathit{isEmptyList}\big([t] + R\big) = \big(\mathit{isEmpty}(t) \wedge \mathit{isEmptyList}(R)\big)$,

   denn alle Tries in der Liste $[t] + R$ sind leer, wenn einerseits $t$ ein leerer Trie ist und wenn andererseits auch alle Tries in $R$ leer sind.

Nun können wir die Methode

$$\mathit{delete} : \mathbb{T} \times \Sigma^* \to \mathbb{T}$$

spezifizieren: Wir definieren den Wert von

$$\mathit{delete}(t, w)$$

für einen Trie $t \in \mathbb{B}$ und ein Wort $w \in \Sigma^*$ durch Induktion nach der Länge des Wortes $w$.

1. $\mathit{delete}\big(\texttt{Node}(v, L, T), \varepsilon\big) = \texttt{Node}(\Omega, L, T)$,

   denn der Wert, der unter dem leeren String $\varepsilon$ in einem Trie gespeichert wird, befindet sich unmittelbar an der Wurzel des Tires und kann dort sofort gelöscht werden.

2. $\mathit{isEmpty}\big(\mathit{delete}(t_i, r)\big) \rightarrow$
   $\texttt{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]).\mathit{delete}(c_i r) =$
   $\texttt{Node}(v, [c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n], [t_1, \cdots, t_{i-1}, t_{i+1}, \cdots, t_n]).$

   Wenn der zu löschende String mit dem Buchstaben $c_i$ anfängt, und wenn das Löschen des Schlüssels $r$ in dem $i$-ten Trie $t_i$ einen leeren Trie ergibt, dann streichen wir den $i$-ten Buchstaben und den dazu korrespondierenden $i$-ten Trie $t_i$.

3. $\neg \mathit{isEmpty}(\mathit{delete}(t_i, r)) \quad \wedge$
   $\mathit{delete}\big(\texttt{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]), c_i r\big) \quad =$
   $\texttt{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, \mathit{delete}(t_i, r), \cdots, t_n]).$

   Wenn der zu löschende String mit dem Buchstaben $c_i$ anfängt, und wenn der Baum $t_i$, der durch das Löschen des Schlüssels $r$ in dem $i$-ten Trie $t_i$ entsteht nicht leer ist, dann löschen wir rekursiv in dem Baum $t_i$ den Schlüssel $r$.

4. $c \notin C \rightarrow \mathit{delete}\big(\texttt{Node}(v, C, T), cr\big) = \texttt{Node}(v, C, T).$

   Wenn der zu löschende String mit dem Buchstaben $c$ anfängt und wenn der Buchstabe $c$ gar kein Element der Buchstaben-Liste $C$ des Tries ist, dann verändert das Löschen den Trie nicht.

### 7.4.3 Implementierung in SetlX

Zunächst müssen wir uns überlegen wie wir den Trie

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$$

in SetlX darstellen. Es ist naheliegend diesen Trie durch den Term

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$$

darzustellen.

```
1  find := procedure(t, w) {
2      match(t) {
3          case Node(v, cl, tl) :
4               match (w) {
5                     case []:    return v;
6                     case [c|r]: for (i in [1 .. #cl]) {
7                                     if (cl[i] == c) {
8                                         return find(tl[i], r);
9                                     }
10                                }
11              }
12         default: abort("Error in find($t$, $w$)");
13     }
14 };
```

Abbildung 7.17: Die Implementierung der Funktion $\mathit{find}(t, w)$.

Wir zeigen nun, wie sich die Tries in SetlX implementieren lassen. Die Abbildung 7.17 zeigt die Realisierung der Funktion `find`$(t, w)$. Hierbei ist $t$ der Trie, in dem nach dem Wort $w$ gesucht werden soll. Zur Vereinfachung der Implementierung gehen wir davon aus, dass es sich bei $w$ um eine Liste von Buchstaben handelt. Das ist keine Einschränkung, denn in SetlX läßt sich ein String $s$ durch den Aufruf

```
strSplit(s,"");
```

in eine Liste von Buchstaben zerlegen. Wir diskutieren jetzt die Details der Implementierung der Funktion $\mathit{find}(t, w)$.

1. In Zeile 3 extrahieren wir den an der Wurzel des Tries $t$ gespeicherten Wert $v$, die Buchstaben-Liste *cl* und die Liste der Teilbäume *tl* über einen `match`-Befehl.

2. Durch den `match`-Befehl in Zeile 4 führen wir eine Fall-Unterscheidung durch:

(a) In Zeile 5 behandeln wir den Fall, dass es sich bei dem String $w$ um den leeren String $\varepsilon$ handelt, der durch die leere Liste dargestellt wird. In diesem Fall geben wir den an der Wurzel des Tries gespeicherten Wert $v$ zurück.

(b) In Zeile 6 zerlegen wir den String $w$ in den ersten Buchstaben $c$ und den Rest $r$. Anschließend suchen wir den Buchstaben $c$ in der Buchstaben-Liste *cl*. Falls wir ihn dort an der Position $i$ finden, suchen wir rekursiv in dem $i$-ten Teilbaum nach dem String $r$.

In dem Fall, dass der Buchstabe $c$ nicht in der Buchstaben-Liste *cl* gefunden wird, wird kein expliziter `return`-Befehl ausgeführt, so dass die Funktion in diesem Fall als Ergebnis $\Omega$ zurück liefert.

```
1  insert := procedure(t, w, v) {
2      match(t) {
3          case Node(v0, cl, tl) :
4               match (w) {
5                     case []:
6                          return Node(v, cl, tl);
7                     case [c|r]:
8                          for (i in [1 .. #cl]) {
9                              if (cl[i] == c) {
10                                 tiNew := insert(tl[i], r, v);
11                                 tlNew := tl[1..i-1] + [tiNew] + tl[i+1..#cl];
12                                 return Node(v0, cl, tlNew);
13                             }
14                         }
15                         n := insert(Node(om, [], []), r, v);
16                         return Node(v0, cl + [c], tl + [n]);
17              }
18          default: abort("Error in insert($t$, $w$)");
19      }
20 };
```

Abbildung 7.18: Die Implementierung der Funktion *insert*$(t, w, v)$.

Die Abbildung 7.18 zeigt die Realisierung der Funktion `insert`$(t, w, v)$. Hierbei ist $t$ der Trie, in dem der Wert $v$ unter dem Schlüssel $w$ eingefügt werden soll. Der Schlüssel $w$ ist eigentlich ein String, der aber in dieser Implementierung durch eine Liste von Buchstaben dargestellt wird.

1. In Zeile 3 extrahieren wir den an der Wurzel des Tries $t$ gespeicherten Wert $v0$, die Buchstaben-Liste *cl* und die Liste der Teilbäume *tl* über einen `match`-Befehl.

2. Durch den `match`-Befehl in Zeile 4 führen wir eine Fall-Unterscheidung durch:

   (a) In Zeile 5 behandeln wir den Fall, dass es sich bei dem String $w$ um den leeren String $\varepsilon$ handelt, der durch die leere Liste dargestellt wird. In diesem Fall überschreiben wir den an der Wurzel des Baums gespeicherten Wert $v0$ mit dem neuen Wert $v$.

   (b) In Zeile 7 zerlegen wir den String $w$ in den ersten Buchstaben $c$ und den Rest $r$. Anschließend suchen wir den Buchstaben $c$ in der Buchstaben-Liste *cl*. Falls wir ihn dort an der Position $i$ finden, fügen wir den Wert $v$ in Zeile 10 rekursiv in dem $i$-ten Teilbaum $tl[i]$ ein.

   In dem Fall, dass der Buchstabe $c$ nicht in der Buchstaben-Liste *cl* gefunden wird, erzeugen wir einen neuen leeren Baum und fügen den Wert $v$ in Zeile 15 in diesem

Baum ein. Anschließend fügen wir den neuen Baum am Ende der Liste *tl* der Teilbäume an. Entsprechend wird dann auch der Buchstabe *c* am Ende der Buchstaben-Liste *cl* angefügt.

```
1   delete := procedure(t, w) {
2       match(t) {
3           case Node(v, cl, tl) :
4                match (w) {
5                      case []:
6                           return Node(om, cl, tl);
7                      case [c|r]:
8                           for (i in [1 .. #cl]) {
9                               if (cl[i] == c) {
10                                  tiNew := delete(tl[i], r);
11                                  if (isEmpty(tiNew)) {
12                                      clNew := cl[1..i-1] + cl[i+1..#cl];
13                                      tlNew := tl[1..i-1] + tl[i+1..#cl];
14                                      return Node(v, clNew, tlNew);
15                                  }
16                                  tlNew := tl[1..i-1] + [tiNew] + tl[i+1..#cl];
17                                  return Node(v, cl, tlNew);
18                              }
19                          }
20                          return t;  // key not found, no change
21               }
22          default: abort("Error in delete($t$, $w$)");
23      }
24  };
25
26  isEmpty := procedure(t) {
27      match(t) {
28          case Node(v, cl, tl) :
29               if (v != om) {
30                   return false;
31               }
32               return forall (t in tl| isEmpty(t));
33          default: abort("Error in find($t$, $w$)");
34      }
35  };
```

Abbildung 7.19: Die Implementierung der Funktion $delte(t, w)$.

Die Abbildung 7.19 zeigt die Realisierung der Funktion `delete`$(t, w)$. Hierbei ist $t$ der Trie, in dem der Schlüssel $w$ gelöscht werden soll. Der Schlüssel $w$ wird wieder durch eine Liste von Buchstaben dargestellt .

1. In Zeile 3 extrahieren wir den an der Wurzel des Tries $t$ gespeicherten Wert $v$, die Buchstaben-Liste *cl* und die Liste der Teilbäume *tl* über einen `match`-Befehl.
2. Durch den `match`-Befehl in Zeile 4 führen wir eine Fall-Unterscheidung durch:
   (a) In Zeile 5 behandeln wir den Fall, dass es sich bei dem String $w$ um den leeren String $\varepsilon$ handelt, der durch die leere Liste dargestellt wird. In diesem Fall löschen wir den an der Wurzel des Baums gespeicherten Wert *v0* indem wir diesen Wert durch $\Omega$ ersetzen.

(b) In Zeile 7 zerlegen wir den String $w$ in den ersten Buchstaben $c$ und den Rest $r$. Anschließend suchen wir den Buchstaben $c$ in der Buchstaben-Liste *cl*. Falls wir ihn dort an der Position $i$ finden, löschen wir den Schlüssel $r$ in Zeile 10 rekursiv in dem $i$-ten Teilbaum $tl[i]$.

Nun gibt es zwei Fälle: Falls der Teilbaum $tl[i]$ nach dem Löschen leer ist, entfernen wir diesen Teilbaum aus der Liste *tl* und müssen dann natürlich auch den dazugehörigen Buchstaben aus der Liste *cl* entfernen.

Andernfalls ersetzen wir einfach den alten $i$-ten Teilbaum durch den Teilbaum *tiNew*, den wir beim rekursiven Aufruf von *delete*() erhalten haben.

In dem Fall, dass der Buchstabe $c$ nicht in der Buchstaben-Liste *cl* gefunden wird, geben wir den Trie $t$ unverändert zurück.

3. Zeile 26 zeigt noch die Definition der Funktion *isEmpty*$(t)$, die überprüft, ob der Trie $t$ leer ist, d.h. keine Werte enthält. Dazu darf zunächst an der Wurzel des Tries $t$ kein Wert gespeichert sein, was in Zeile 29 geprüft wird. Außerdem müssen alle Tries in der Liste *tl* ebenfalls leer sein, was wir in Zeile 32 testen.

**Binäre Tries**: Wir nehmen im folgenden an, dass unser Alphabet nur aus den beiden Ziffern 0 und 1 besteht, es gilt also $\Sigma = \{0, 1\}$. Dann können wir natürliche Zahlen als Worte aus $\Sigma^*$ auffassen. Wir wollen die Menge der *binären Tries* mit BT bezeichnen und wie folgt induktiv definieren:

1. $\mathit{Nil} \in \mathrm{BT}$.
2. $\mathit{Bin}(v, l, r) \in \mathrm{BT}$ falls
   (a) $v \in \mathit{Value} \cup \{\Omega\}$.
   (b) $l, r \in \mathrm{BT}$.

Die Semantik legen wir fest, indem wir eine Funktion

$$\mathit{find} : \mathrm{BT} \times \mathbb{N} \to \mathit{Value} \cup \{\Omega\}$$

definieren. Für einen binären Trie $b$ und eine natürliche Zahl $n$ gibt $\mathit{find}(b, n)$ den Wert zurück, der unter dem Schlüssel $n$ in dem binären Trie $b$ gespeichert ist. Falls in dem binären Trie $b$ unter dem Schlüssel $n$ kein Wert gespeichert ist, wird $\Omega$ zurück gegeben. Formal definieren wir den Wert von $\mathit{find}(b, n)$ durch Induktion nach dem Aufbau von $b$. Im Induktions-Schritt ist eine Neben-Induktion nach $n$ erforderlich.

1. $\mathit{find}(\mathit{Nil}, n) = \Omega$,
   denn in dem leeren binären Trie finden wir keine Werte.
2. $\mathit{find}\big(\mathit{Bin}(v, l, r), 0\big) = v$,
   denn der Schlüssel 0 entspricht dem leeren String $\varepsilon$.
3. $n \neq 0 \to \mathit{find}\big(\mathit{Bin}(v, l, r), 2{\cdot}n\big) = \mathit{find}(l, n)$,
   denn wenn wir Zahlen im Binärsystem darstellen, so hat bei geraden Zahlen das letzte Bit den Wert 0 und die 0 soll dem linken Teilbaum entsprechen.
4. $\mathit{find}\big(\mathit{Bin}(v, l, r), 2{\cdot}n{+}1\big) = \mathit{find}(r, n)$,
   denn wenn wir Zahlen im Binärsystem darstellen, so hat bei ungeraden Zahlen das letzte Bit den Wert 1 und die 1 soll dem rechten Teilbaum entsprechen.

**Aufgabe**:

1. Stellen Sie Gleichungen auf, die das Einfügen und das Löschen in einem binären Trie beschreiben. Achten Sie beim Löschen darauf, dass binäre Tries der Form $\mathit{Bin}(\Omega, \mathit{Nil}, \mathit{Nil})$ zu *Nil* vereinfacht werden.

   **Hinweis**: Um die Gleichungen zur Spezifikation der Funktion *delete*() nicht zu komplex werden zu lassen ist es sinnvoll, eine Hilfsfunktion zur Vereinfachung von binären Tries zu definieren.

2. Implementieren Sie binäre Tries in SetlX.

**Bemerkung**: Binäre Tries werden auch als *digitale Suchbäume* bezeichnet.

## 7.5 Hash-Tabellen

Eine Abbildung

$$f : \mathit{Key} \rightarrow \mathit{Value}$$

kann dann sehr einfach implementiert werden, wenn

$$\mathit{Key} = \{0, 1, 2, \cdots, n\},$$

denn dann reicht es aus, ein Feld der Größe $n+1$ zu verwenden. Falls nun der Definitions-Bereich $D$ der darzustellenden Abbildung nicht die Form einer Menge der Gestalt $\{1, \cdots, n\}$ hat, könnten wir versuchen, $D$ zunächst auf eine Menge der Form $\{1, \cdots, n\}$ abzubilden. Wir erläutern diese Idee an einem einfachen Beispiel. Wir betrachten eine naive Methode um ein Telefon-Buch abzuspeichern:

1. Wir machen zunächst die Annahme, dass alle Namen aus genau 8 Buchstaben bestehen. Dazu werden kürzere Namen mit Blanks aufgefüllt und Namen die länger als 8 Buchstaben sind, werden nach dem 8-ten Buchstaben abgeschnitten.

2. Als nächstes übersetzen wir Namen in einen Index. Dazu fassen wir die einzelnen Buchstaben als Ziffern auf, die die Werte von 0 bis 26 annehmen können. Dem Blank ordnen wir dabei den Wert 0 zu. Nehmen wir an, dass die Funktion *ord* jedem Buchstaben aus der Menge $\Sigma = \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\}$ einen Wert aus der Menge $\{0, \cdots, 26\}$ zuordnet

   $$\mathit{ord} : \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\} \rightarrow \{0, \cdots, 26\},$$

   so läßt sich der Wert eines Strings $w = c_0 c_1 \cdots c_7$ durch eine Funktion

   $$\mathit{code} : \Sigma^* \rightarrow \mathbb{N}$$

   berechnen, die wie folgt definiert ist:

   $$\mathit{code}(c_0 c_1 \cdots c_7) = \sum_{i=0}^{7} \mathit{ord}(c_i) \cdot 27^i.$$

   Die Menge *code* bildet die Menge aller Wörter mit 8 Buchstaben bijektiv auf die Menge der Zahlen $\{0, \cdots, (27^8 - 1)/26\}$ ab.

Leider hat diese naive Implementierung mehrere Probleme:

1. Das Feld, das wir anlegen müssen, hat eine Größe von

   $$27^8 = 282\,429\,536\,481$$

   Einträgen. Selbst wenn jeder Eintrag nur die Größe zweier Maschinen-Worte hat und ein Maschinen-Wort aus 4 Byte besteht, so bräuchten wir etwas mehr als ein Terabyte um eine solche Tabelle anzulegen.

2. Falls zwei Namen sich erst nach dem 8-ten Buchstaben unterscheiden, können wir zwischen diesen Namen nicht mehr unterscheiden.

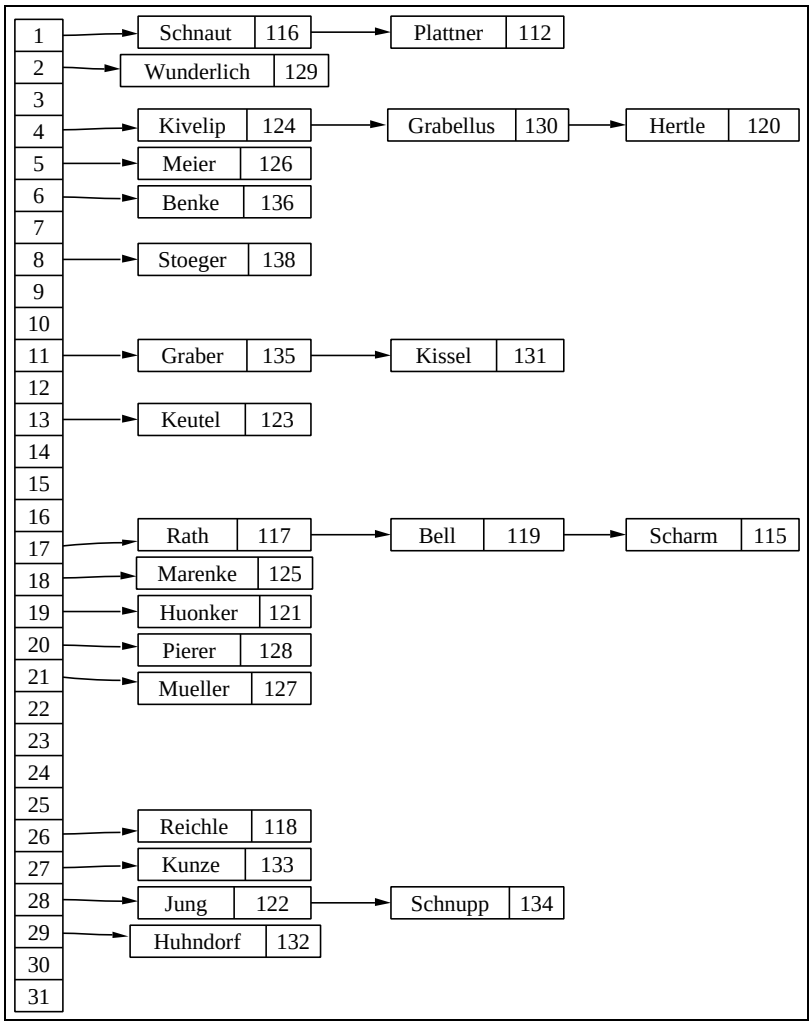


Abbildung 7.20: Eine Hash-Tabelle

Wir können diese Probleme wir folgt lösen:

1. Wir ändern die Funktion `code` so ab, dass das Ergebnis immer kleiner-gleich einer vorgegebene Zahl `size` ist. Die Zahl `size` gibt dabei die Größe eines Feldes an und ist so klein, dass wir ein solches Feld bequem anlegen können.

   Eine einfache Möglichkeit, die Funktion *code* entsprechend abzuändern, besteht in folgender Implementierung:

   $$code(c_0c_1\cdots c_n) = \left(\sum_{i=0}^{n} ord(c_i)\cdot 27^i\right) \ \% \ \texttt{size}.$$

   Um eine Überlauf zu vermeiden, können wir für $k = n, n-1, \cdots, 1, 0$ die Teilsummen $s_k$ wie folgt induktiv definieren:

   (a) $s_n = ord(c_n)$

   (b) $s_k = (ord(c_k) + s_{k+1}\cdot 27) \ \%\ \texttt{size}$

   Dann gilt $$s_0 = \left(\sum_{i=0}^{n} ord(c_i)\cdot 27^i\right) \ \% \ \texttt{size}.$$

2. In dem Feld der Größe `size` speichern wir nun nicht mehr die Werte, sondern statt dessen Listen von Paaren aus Schlüsseln und Werten. Dies ist notwendig, denn wir können nicht verhindern, dass die Funktion `code`() für zwei verschiedene Schlüssel den selben Index liefert.

Abbildung 7.20 auf Seite 122 zeigt, wie ein Feld, in dem Listen von Paaren abgebildet sind, aussehen kann. Ein solches Feld bezeichnen wir als Hash-Tabelle. Wir diskutieren nun die Implementierung dieser Idee in SetlX.

```
1  var gOrd;
2  gOrd := { [ char(i), i ] : i in [ 0 .. 127 ] };
3
4  hashCode := procedure(s, n) {
5      return hashCodeAux(s, n) + 1;
6  };
7
8  hashCodeAux := procedure(s, n) {
9      if (s == "") {
10         return 0;
11     }
12     return (gOrd[s[1]] + 128 * hashCode(s[2..], n)) % n;
13 };
```

Abbildung 7.21: Die Funktion `hashCode`.

Als erstes implementieren wir die Funktion, die jedem String einen Hash-Code zuordnet. Abbildung 7.21 zeigt die Implementierung.

1. Es gibt in `SetlX` keine Funktion, die jedem Ascii-Zeichen den dazugehörigen Ascii-Code zuordnet, aber erfreulicherweise läßt sich diese Funktion mühelos wir in Zeile 2 gezeigt als funktionale Relation definieren, denn es gibt die Funktion $\texttt{char}(i)$, die zu einer Zahl $i \in \{0, \cdots, 127\}$ das Zeichen mit dem Ascii-Code $i$ berechnet.

2. Die Funktion $\texttt{hashCode}(s, n)$ berechnet für den String $s$ einen Hash-Code. Dabei gilt

   $$\texttt{hashCode}(s, n) \in \{1, \cdots, n\}.$$

   Dadurch kann dieser Hash-Code zum Indizieren eines Feldes in SetlX benutzt werden. Die Berechnung greift dabei zurück auf die Prozedur $\texttt{hashCodeAux}(s, n)$, für die

   $$\texttt{hashCode}(s, n) \in \{0, \cdots, n-1\}$$

   gilt.

3. Die Funktion $\texttt{hashCodeAux}(s, n)$ ist durch Rekursion über den String $s$ definiert. Für einen String $s$ der Länge $m$ gilt

   $$\texttt{hashCodeAux}(s, n) := \left(\sum_{i=1}^{m} \texttt{ord}(s[i]) \cdot 128^{i-1}\right) \,\%\, n.$$

   Hierbei liefert der Ausdruck $\texttt{ord}(c)$ für ein Ascii-Zeichen $c$ den dazugehörigen Ascii-Code.

Als nächstes überlegen wir uns, welche Daten-Strukturen wir brauchen, um eine Hash-Tabelle zu repräsentieren.

1. Wir benötigen ein Feld, indem wir die einzelnen Listen ablegen.

2. Wenn die einzelnen Listen zu groß werden, wird die Suche ineffizient. Daher ist es notwendig, die Größe dieser Listen zu kontrollieren: Wenn die Listen zu groß werden, muss das Feld

vergrößert werden. Um diesen Prozeß zu steuern, müssen wir zunächst nachhalten, wieviele Elemente schon in der Hash-Tabelle abgespeichert sind.

Wir stellen daher eine Hash-Tabelle durch einen Term der Form

`HashMap`(*array*, *numEntries*)

dar. Hier ist *array* das Feld, in dem die einzelnen Listen gespeichert werden und *numEntries* gibt die Zahl der Schlüssel an, für die es in der Hash-Tabelle einen Eintrag gibt. Um sicherzustellen, dass die Performanz unserer Hash-Tabelle auch im schlechtesten Fall nicht schlechter als logarithmisch ist, werden wir in dem Feld *array* an Stelle von Listen von *Key-Value*-Paaren allerdings Mengen von *Key-Value*-Paaren speichern, denn das Suchen in einer Liste erfordert einen linearen Aufwand, während die Suche in einer Menge nur einen logarithmischen Aufwand erfordert, denn eine Menge wird in SetlX intern durch einen Rot-Schwarz-Baum dargestellt. Obwohl die Einträge in dem Feld *array* eigentlich Mengen sind, die funktionale Relationen repräsentieren, werden wir diese Einträge im Folgenden zur Vereinfachung als Listen bezeichnen.

```
14  createHashMap := procedure(n) {
15      array := [ {} : i in [1 .. n] ];
16      return HashMap(array, 0);
17  };
```

Abbildung 7.22: Erzeugung einer Hash-Tabelle.

Die in Abbildung 7.22 definierte Funktion *createHashMap*($n$) legt ein Feld der Größe $n$ an, indem zunächst alle Listen leer sind und erzeugt dann denn dazu passenden Term.

Abbildung 7.23 zeigt die Implementierung der Funktion `insert`(*map*, *key*, *value*), die den Schlüssel *key* zusammen mit dem Wert *value* in der Hash-Tabelle *map* einfügt und die neue Tabelle zurück gibt.

1. Zunächst definieren wir in Zeile 18 den sogenannten *Auslastungs-Faktor* der Hash-Tabelle. Die Idee ist, die Hash-Tabelle dynamisch zu vergrößern, wenn die durchschnittliche Länge der Listen der Tabelle den *Auslastungs-Faktor* übersteigt.

2. Daher überprüfen wir in Zeile 24, ob die Anzahl der Einträge größer als die Größe des Feldes multipliziert mit dem *Auslastungs-Faktor* ist, denn in diesem Fall vergrößern wir die Tabelle durch einen Aufruf von `rehash`(*map*). Anschließend fügen wir den Schlüssel in die passendend vergrößerte Tabelle ein.

3. Falls die Tabelle nicht vergrößert werden muss, berechnen wird den Hash-Code zu dem Schlüssel *key* und fügen den neuen Wert dann in Zeile 31 in die Liste ein. Dabei können zwei Fälle auftreten:

   (a) Zu dem Schlüssel war bereits vorher ein Wert gespeichert. In diesem Fall ändert sich die Zahl der Einträge der Tabelle nicht.

   (b) Andernfalls erhöht sich die Zahl der in der Tabelle gespeicherten Einträge um 1.

   Um herausfinden zu können, welcher dieser beiden Fälle tatsächlich vorliegt, überprüfen wir die Größe der Liste vor und nach dem Einfügen. Ändert sich diese Größe, so wurde ein neuer Schlüssel eingefügt, andernfalls haben wir nur einen Wert überschrieben.

Abbildung 7.24 zeigt die Implementierung der Funktion `rehash`(*map*), welche die Aufgabe hat, die Größe des in der Hash-Tabelle *map* gespeicherten Feldes zu verdoppeln. Theoretische Untersuchungen, die über den Rahmen der Vorlesung hinausgehen, zeigen, dass die Größe der Tabelle eine Primzahl sein sollte. Daher definieren wir in Zeile 43 eine Liste von Primzahlen, wobei die $i+1$-te Primzahlen in dieser Liste in etwa doppelt so groß ist wie die $i$-te Primzahl.

```
18  var gAlpha;  // load factor
19  gAlpha := 2;
20
21  insert := procedure(map, key, value) {
22      match (map) {
23          case HashMap(array, numEntries):
24              if (numEntries > #array * gAlpha) {
25                  map := rehash(map);
26                  return insert(map, key, value);
27              }
28              index := hashCode(key, #array);
29              aList := array[index];
30              oldSz := #aList;
31              aList[key] := value;
32              newSz := #aList;
33              array[index] := aList;
34              if (oldSz == newSz) {
35                  return HashMap(array, numEntries);
36              } else {
37                  return HashMap(array, numEntries + 1);
38              }
39          default: abort("Error in insert($map$, $key$, $value$)");
40      }
41  };
```

Abbildung 7.23: Die Implementierung der Funktion `insert`(*map*, *key*, *value*).

1. In Zeile 52 bestimmen wir die kleinste Primzahl, die bei dem gegebenen *Auslastungs-Faktor* noch zulässig ist.

2. Anschließend konstruieren wir in Zeile 53 eine neue Hash-Tabelle mit der vorher berechneten Größe, die natürlich noch leer ist.

3. In der anschließenden `for`-Schleife werden alle Daten aus der gegebenen Hash-Tabelle *map* in die neue Hash-Tabelle *bigMap* kopiert.

Als letztes besprechen wir die Implementierung der Funktion `delete`(*map*, *key*), die in Abbildung 7.25 gezeigt ist. Im wesentlichen ist die Implementierung analog zur Implementierung der Funktion `insert`(*map*, *key*, *value*), denn anstelle von *value* fügen wir einfach den Wert $\Omega$ ein und löschen damit den eigentlichen Wert, aber ein wesentlicher Unterschied besteht darin, dass kein Aufruf der Funktion `rehash`(`map`) notwendig ist, denn die Anzahl der Einträge kann ja durch einen Aufruf von `delete`() nicht wachsen. Wir könnten Speicherplatz einsparen, wenn wir die Tabelle auch wieder schrumpfen würden, wenn der Auslastungsfaktor zu weit unterschritten wird, aber in der Praxis bringt eine solche Optimierung wenig, weil die Tabellen oft nach einem Löschen wieder wachsen.

Falls wir in unserer Implementierung tatsächlich mit Listen und nicht mit Mengen arbeiten würden, dann könnte die Komplexität der Methoden *find*, *insert* und *delete* im ungünstigsten Fall linear mit der Anzahl der Einträge in der Hash-Tabelle wachsen. Dieser Fall tritt dann auf, wenn die Funktion `hashTable`$(k, n)$ für alle Schlüssel $k$ den selben Wert berechnet. Dieser Fall ist allerdings sehr unwahrscheinlich. Der Normalfall ist der, dass alle Listen etwa gleich lang sind. Die durchschnittliche Länge einer Liste ist dann

$$\alpha = \frac{\texttt{count}}{\texttt{size}}.$$

```
42  var gPrimes;
43  gPrimes := [ 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381,
44               32749, 65521, 131071, 262139, 524287, 1048573, 2097143, 4194301,
45               8388593, 16777213, 33554393, 67108859, 134217689, 268435399,
46               536870909, 1073741789, 2147483647
47             ];
48
49  rehash := procedure(map) {
50      match (map) {
51          case HashMap(array, numEntries):
52               primeIdx := min({ p in gPrimes | p * gAlpha > numEntries });
53               bigMap   := createHashMap(primeIdx);
54               for (aList in array) {
55                   for ([k, v] in aList) {
56                       bigMap := insert(bigMap, k, v);
57                   }
58               }
59               return bigMap;
60          default: abort("Error in rehash($map$)");
61      }
62  };
```

Abbildung 7.24: Implementierung der Funktion `rehash`(*map*).

```
63   delete := procedure(map, key) {
64       match (map) {
65           case HashMap(array, numEntries):
66                index := hashCode(key, #array);
67                aList := array[index];
68                oldSz := #aList;
69                aList[key] := om;
70                newSz := #aList;
71                array[index] := aList;
72                if (oldSz == newSz) {
73                    return HashMap(array, numEntries);
74                } else {
75                    return HashMap(array, numEntries - 1);
76                }
77           default: abort("Error in insert($map$, $key$, $value$)");
78       }
79   };
```

Abbildung 7.25: Die Funktion `delete`(*map*, *key*).

Hierbei ist `count` die Gesamtzahl der Einträge in der Tabelle und `size` gibt die Größe der Tabelle an. Das Verhältnis $\alpha$ dieser beiden Zahlen ist genau der *Auslastungs-Faktor* der Hash-Tabelle. In der Praxis zeigt sich, dass $\alpha$ kleiner als 4 sein sollte. In *Java* gibt es die Klasse *HashMap*, die Abbildungen als Hash-Tabellen implementiert. Dort hat der per Default eingestellte maximale Auslastungs-Faktor sogar nur den Wert `0.75`.

## 7.6 Mengen und Abbildungen in Java

Mengen und Abbildungen gehören zu den wichtigsten Werkzeuge im Werkzeugkasten eines Informatikers. In Java gibt es zwei abstrakte Daten-Typen, um die Begriffe *Mengen* und *Abbildungen* zu beschreiben. In der *Java*-Terminologie werden abstrakte Daten-Typen als Schnittstelle (engl. `interface`) bezeichnet. Wir diskutieren nun diese von *Java* zur Verfügung gestellten Schnittstellen.

### 7.6.1 Das Interface `Collection<E>`

Die Schnittstelle `Collection<E>` beschreibt eine beliebige *Zusammenfassung* von Elementen. Dieser Begriff ist eine Verallgemeinerung des Mengen-Begriffs, denn in einer *Zusammenfassung* können Elemente auch mehrfach enthalten sein. Die Schnittstelle `Collection<E>` hat den Typ-Parameter `E`, der für den Typ der Elemente steht, die in der Zusammenfassung enthalten sind. Diese Schnittstelle spezifiziert die folgenden Methoden:

1. `void clear()`

   Der Aufruf $c$.`clear`() löscht alle Elemente aus der Zusammenfassung $c$. Diese ist danach leer. Falls $c$ eine Menge ist, können wir die Semantik formal wie folgt spezifizieren:

   $$c.\texttt{clear}() : \quad c' = \{\}.$$

   Hier bezeichnet $c'$ den Wert, den die Zusammenfassung $c$ hat, nachdem der Aufruf $c$.`clear`() beendet ist. Außerdem haben wir zur Spezifikation der Semantik eine neue Schreibweise benutzt: Wenn $o$ ein Objekt ist, $m$ eine Methode ist und $a_1$, $\cdots$, $a_n$ Argumente sind, so dass die Methode $m$ für das Objekt $o$ mit diesen Argumenten aufgerufen werden kann, dann schreiben wir

   $$o.m(a_1, \cdots, a_n) : \quad F(a_1, \cdots, a_n)$$

   wenn nach dem Aufruf $o.m(a_1, \cdots, a_n)$ die Formel $F(a_1, \cdots, a_n)$ gilt.

2. `boolean add(E` $e$`)`

   Für eine Zusammenfassung $c$ fügt der Aufruf $c$.`add`($e$) das Element $e$ der Zusammenfassung $c$ hinzu. Falls sich die Zusammenfassung $c$ bei dieser Operation ändert, gibt die Methode `true` zurück. Wenn $c$ eine Menge ist, so können wir die Semantik durch die folgenden Gleichungen beschreiben:

   (a) $c.\texttt{add}(e) : \quad c' = c \cup \{e\}$.

   (b) $c.\texttt{add}(e) = (e \notin c)$,

   denn wenn $e$ kein Element der Menge $c$ ist, dann wird $e$ in die Menge $c$ eingefügt und der Aufruf $c$.`add`($e$) gibt folglich als Ergebnis `true` zurück.

3. `boolean addAll(Collection<E>` $d$`)`

   Bei dem Aufruf $c$.`addAll`($d$) werden alle Elemente der Zusammenfassung $d$ in die Zusammenfassung $c$ eingefügt. Falls sich die Zusammenfassung $c$ dabei ändert, liefert die Methode als Ergebnis `true` zurück. Falls es sich bei $c$ und $d$ um Mengen handelt, können wir also schreiben

   (a) $c.\texttt{addAll}(d) : \quad c' = c \cup d$,

   (b) $c.\texttt{addAll}(d) = (c' \neq c)$.

4. `boolean contains(Element` $e$`)`

   Der Aufruf $c$.`contains`($e$) liefert genau dann `true`, wenn $e$ ein Element der Zusammenfassung $c$ ist. Ist $c$ eine Menge, so gilt also

   $$c.\texttt{contains}(e) = (e \in c).$$

5. `boolean containsAll(Collection<E>` $d$`)`

   Der Aufruf $c$.`containsAll`($d$) liefert genau dann `true`, wenn alle Elemente der Zusammenfassung $d$ in der Zusammenfassung $c$ enthalten sind. Falls $c$ und $d$ Mengen sind, gilt also

   $$c.\texttt{containsAll}(d) = (d \subseteq c).$$

6. `boolean isEmpty()`

   Der Aufruf $c$.`isEmpty`() liefert genau dann `true`, wenn die Zusammenfassung $c$ keine Elemente enthält. Falls $c$ eine Menge ist, gilt also

   $$c.\texttt{isEmpty}() = \big(c = \{\}\big).$$

7. `boolean remove(Object` $e$`)`

   Der Aufruf $c$.`remove`($e$) entfernt das Element $e$ aus der Zusammenfassung $c$, sofern $e$ in der Zusammenfassung auftritt. Sollte die Zusammenfassung $c$ das Element $e$ mehrfach enthalten, so wird nur ein Auftreten von $e$ entfernt. Falls $e$ ein Element von $c$ ist, liefert die Methode als Ergebnis `true`, sonst `false`. Falls $c$ eine Menge ist, so läßt sich die Semantik durch folgende Gleichungen spezifizieren.

   (a) $c.\texttt{remove}(e): \quad c' = c \setminus \{e\}$,

   (b) $c.\texttt{remove}(e) = (e \in c)$.

8. `boolean removeAll(Collection<?>` $d$`)`

   Der Aufruf $c$.`removeAll`($d$) entfernt alle Elemente der Zusammenfassung $d$ aus der Zusammenfassung $c$. Sollte die Zusammenfassung $c$ ein Element $e$ mehrfach enthalten, dass in der Zusammenfassung $d$ nur einmal auftritt, so werden alle Auftreten von $e$ aus der Zusammenfassung $c$ entfernt. Die Methode gibt als Ergebnis `true` zurück, wenn bei dem Aufruf wenigstens ein Element aus der Zusammenfassung $c$ entfernt wurde. Falls $c$ und $d$ Mengen sind, kann die Semantik wie folgt beschrieben werden.

   (a) $c.\texttt{removeAll}(d): \quad c' = c \setminus d$,

   (b) $c.\texttt{removeAll}(d) = \big(c' \neq c\big)$.

9. `boolean retainAll(Collection<?>` $d$`)`

   Der Aufruf $c$.`retainAll`($d$) bildet den Schnitt der Zusammenfassungen $c$ und $d$. Es werden alle Elemente aus $c$, die nicht in $d$ auftreten, aus $c$ entfernt. Der Aufruf gibt als Ergebnis `true` zurück, wenn Elemente aus $c$ entfernt wurden. Falls $c$ und $d$ Mengen sind, läßt sich die Semantik wie folgt spezifizieren:

   (a) $c.\texttt{retainAll}(d): \quad c' = c \cap d$,

   (b) $c.\texttt{retainAll}(d) = (c' \neq c)$.

10. `int size()`

    Der Aufruf $c$.`size`() liefert die Anzahl der Elemente der Zusammenfassung $c$. Tritt ein Element mehrfach in $c$ auf, so wird es auch mehrfach gezählt.

11. `Object[] toArray()`

    Der Aufruf $c$.`toArray` wandelt die Zusammenfassung in ein Feld um, das alle Elemente der Zusammenfassung enthält. Beim Aufruf dieser Methode geht das Wissen über den Typ der Elemente für den *Java*-Typ-Checker verloren.

12. `T[] toArray(T[]` $a$`)`

    Falls $c$ eine Zusammenfassung vom Typ `T` ist und wenn außerdem $a$ ein Feld von Elementen des selben Typs `T` ist, dann liefert der Aufruf $c$.`toArray`($a$) ein Feld, das alle Elemente der Zusammenfassung $c$ enthält und das außerdem ein Feld vom Typ `T` ist. Die Verwendung dieser

Methode erhält im Gegensatz zu der vorhin diskutierten Methode die Typ-Information. Die Methode kann allerdings nur verwendet werden, wenn ein geeignetes Hilfsfeld $a$ zur Verfügung steht. Das Problem ist hier, dass es in *Java* nicht möglich ist, *generische* Felder zu erzeugen: Wenn `T` ein Typ-Parameter ist, so liefert die Anweisung

```
T[] a = new T[10];
```

einen Compiler-Fehler. Es ist ebenfalls nicht möglich ein Feld vom Typ `Object[]`, das von der Methode `toArray()` ohne Parameter angelegt wurde, in ein anderes Feld zu casten. In Abbildung 7.26 wird zunächst eine Liste von Elementen des Typs `Integer` angelegt. anschließend wird diese Liste mit der Methode `toArray` in das Feld `a` umgewandelt. Dieses Feld enthält zwar jetzt nur Elemente des Typs `Integer`, trotzdem kann es nicht zu einem Feld des Typs `Integer[]` gecastet werden, die Anwendung des Cast-Operators in Zeile 10 liefert eine `ClassCastException`. Das ist auch richtig so, denn wir können in ein Feld vom Typ `Object[]` beliebige Objekte schreiben, während wir in ein Feld vom Typ `Integer[]` nur Objekte vom Typ `Integer` schreiben dürfen.

```
1  import java.util.*;
2
3  public class TestCast
4  {
5      public static void main(String[] args) {
6          List<Integer> l = new LinkedList<Integer>();
7          for (Integer i = 0; i < 10; ++i) {
8              l.add(i);
9          }
10         Object [] a = l.toArray();
11         Integer[] b = (Integer[]) a;
12     }
13 }
```

Abbildung 7.26: Casten eines Feldes

Um die Liste `l` in ein Feld vom Typ `Integer[]` zu transformieren, müssen wir daher anders vorgehen. Abbildung 7.27 zeigt, wie die Transformation mit Hilfe der zweiten Variante der Methode `toArray` gelingt.

```
1  import java.util.*;
2
3  public class TestCast2
4  {
5      public static void main(String[] args) {
6          List<Integer> l = new LinkedList<Integer>();
7          for (Integer i = 0; i < 10; ++i) {
8              l.add(i);
9          }
10         Integer[] a = new Integer[0];
11         Integer[] b = l.toArray(a);
12     }
13 }
```

Abbildung 7.27: Verwendung des Hilfsfeldes bei der Methode `toArray`().

13. `Iterator<E> iterator()`

    Für eine Zusammenfassung $c$ liefert der Aufruf $c$.`iterator`() einen *Iterator*, mit dessen Hilfe es möglich ist, die Elemente der Zusammenfassung $c$ aufzuzählen. Dadurch wird es möglich, die *erweiterte* `for`*-Schleife* zu benutzen. Ist $c$ eine Zusammenfassung vom Typ `Collection<E>`, so können wir beispielsweise alle Elemente von $c$ mit der folgenden `for`-Schleife ausdrucken:

```
for (E e: c) {
   System.out.println(e);
}
```

Von der Schnittstelle `Collection<E>` gibt es eine Reihe von Spezialisierungen, von denen für uns die folgenden drei wichtig sind:

1. Die Schnittstelle `Set<E>` beschreibt Mengen. Mengen sind Zusammenfassungen, die jedes Element höchstens einmal enthalten. Falls es möglich ist die Elemente der Menge miteinander zu vergleichen, so läßt sich eine Menge in *Java* durch die Klasse `TreeSet<E>` darstellen. Diese Klasse hat neben den Methoden der Schnittstelle `Collection<E>` unter anderem noch die folgenden Methoden.

   (a) `E first()`
   Der Aufruf $s$.`first`() liefert das kleinste Element der Menge $s$.

   (b) `E last()`
   Der Aufruf $s$.`last`() liefert das größte Element der Menge $s$.

   Die Klasse `TreeSet<E>` stellt die folgenden Konstruktoren zur Verfügung.

   (a) `TreeSet()`
   Dieser Konstruktor erzeugt die leere Menge.

   (b) `TreeSet(Collection<E>` $c$`)`
   Dieser Konstruktor erzeugt eine Menge, die alle Elemente der Zusammenfassung $c$ enthält.

   Die Klasse `TreeSet` wird durch *Rot-Schwarz-Bäume* implementiert. Genau wie AVL-Bäume, sind auch Rot-Schwarz-Bäume binäre Bäume, die näherungsweise balanciert sind. Bei den Rot-Schwarz-Bäume ist die Idee, dass die Knoten eines Baumes entweder rot oder schwarz markiert sind. Die roten Knoten werden durch einen Term der Form

   $$\texttt{Red}(k, v, l, r)$$

   dargestellt, die schwarzen Knoten haben die Form

   $$\texttt{Black}(k, v, l, r).$$

   Dabei ist jedesmal $k$ der Schlüssel, der in dem Knoten gespeichert ist, $v$ der mit dem Schlüssel $k$ assoziierte Wert, $l$ ist der linke Teilbaum und $r$ ist der rechte Teilbaum. Der leere Baum wird weiterhin durch den Term `Nil` dargestellt. Bei der Berechnung der Höhe eines Rot-Schwarz-Baums werden die roten Knoten nicht gezählt. Formal ist die Funktion $\mathit{height}(b)$ für einen Rot-Schwarz-Baum durch Induktion definiert:

   (a) $\mathit{height}(\texttt{Nil}) := 0$,

   (b) $\mathit{height}\big(\texttt{Black}(k, v, l, r)\big) := \max\big(\mathit{height}(l), \mathit{height}(r)\big) + 1$,

   (c) $\mathit{height}\big(\texttt{Red}(k, v, l, r)\big) := \max\big(\mathit{height}(l), \mathit{height}(r)\big)$.
   Im Gegensatz zur Definition von $\mathit{height}\big(\texttt{Black}(k, v, l, r)\big)$ wird also bei der Berechnung der Höhe von $\texttt{Red}(k, v, l, r)$ dem Maximum der Höhen der Teilbäume $l$ und $r$ keine Eins dazu addiert, so dass ein Baum der Form $\texttt{Red}(k, v, l, r)$ nicht höher ist, als der höchste seiner Teilbäume.

Zusätzlich gelten für Rot-Schwarz-Bäume die folgenden Bedingungen:

(a) Der Knoten an der Wurzel ist schwarz.

(b) Die Kinder eines roten Knotens sind immer schwarz.

(c) Die Kinder eines schwarzen Knotens können sowohl rot als auch schwarz sein.

(d) Linker und rechter Teil-Baum eines Rot-Schwarz-Baums haben die selbe Höhe.

Asymptotisch haben die Operationen *find*(), *insert*() und *delete*() für Rot-Schwarz-Bäume und AVL-Bäume die gleiche Komplexität. In der Praxis sind Rot-Schwarz-Bäume etwas schneller.

Falls die Elemente einer Menge nicht in natürlicher Weise geordnet werden können, dann kann an Stelle der Klasse `TreeSet<E>` die Klasse `HashSet<E>` verwendet werden. In der Praxis sind Hash-Tabellen meist schneller als Rot-Schwarz-Bäume, aber wenn die Schlüssel ungüngstig verteilt sind, dann ist die Komplexität der Methode *find*() linear in der Anzahl der Einträge der Hash-Tabelle. Die Klasse `HashSet<E>` hat die folgenden Konstruktoren:

(a) `HashSet()`
Dieser Konstruktor erzeugt eine leere Hash-Tabelle. Per Default ist hier der Load-Faktor auf 0.75 gesetzt, was zur Folge hat, dass mindestens ein Viertel der Einträge des Feldes, das der Hash-Tabelle zu Grunde liegt, leer sind. Das Feld selbst hat zunächst eine Größe von 16.

(b) `HashSet(Collection<E>` $c$`)`
Dieser Konstruktor erzeugt eine Hash-Tabelle, die alle Elemente aus der Zusammenfassung $c$ enthält. Der Load-Faktor ist 0.75.

(c) `HashSet(int` $n$`)`
Dieser Konstruktor erzeugt eine leere Hash-Tabelle, für die das zu Grunde liegende Feld die Größe $n$ hat. Der Load-Faktor ist 0.75.

(d) `HashSet(int` $n$`, float` $\alpha$`)`
Dieser Konstruktor erzeugt eine leere Hash-Tabelle, für die das zu Grunde liegende Feld die Größe $n$ hat. Der Load-Faktor ist $\alpha$.

2. Die Schnittstelle `List<E>` beschreibt Listen, deren Elemente den Typ `E` haben.

Gegenüber einer allgemeinen Zusammenfassung, in der die Elemente in keiner Weise geordnet sind, haben alle Elemente einer Liste einen Index. Dieser Index ist eine natürliche Zahl. Über diesen Index kann auf die einzelnen Elemente zugegriffen werden. Die beiden wesentlichen Methoden sind hier Methoden `get`() und `set`(). Diese Methoden haben die folgenden Signaturen:

(a) `E get(int` $i$`)`
Der Aufruf $l$.`get`($i$) liefert das $i$-te Element der Liste $l$, wobei die Zählung bei $i = 0$ beginnt.

(b) `void set(int` $i$`, E` $e$`)`
Der Aufruf $l$.`set`($i, e$) ersetzt das $i$-te Element der Liste $l$ durch $e$.

Die Methoden `add` und `addAll`, welche die Schnittstelle `List` von der Schnittstelle `Collection` erbt, fügt die neuen Elemente am Ende der Liste an. Um auch Elemente an beliebiger Position in einer Liste einfügen zu können, gibt es die folgenden Varianten.

(a) `void add(int` $i$`, E` $e$`)`
Der Aufruf $l$.`add`($i, e$) fügt das Element $e$ an der Position in die Liste $l$ ein, die durch den Index $i$ gegeben ist. Die Elemente, die vorher schon in der Liste $l$ enthalten waren und die zusätzlich einen Index größer oder gleich $i$ hatten, vergrößern ihren Index um 1.

Beispielsweise hat das Element, das vorher den Index $i$ hatte, hinterher den Index $i+1$. Also wird bei dem Aufruf $l$.`add`$(0, e)$ das Element $e$ am Anfang der Liste $l$ eingefügt, wobei alle bereits vorher in der Liste vorhandenen Elemente um einen Platz nach hinten geschoben werden.

(b) `void addAll(int` $i$`, Collection<E>` $c$`)`
Analog werden hier die Elemente der Zusammenfassung $c$ in der Liste $l$ an der Position eingefügt, die durch den Index $i$ gegeben ist.

Die beiden wichtigsten Klassen, welche die Schnittstelle `List` implementieren, sind die Klassen `LinkedList` und `ArrayList`.

(a) `LinkedList`
Diese Klasse ist durch verkettete Listen implementiert. Die bedingt, dass die Operationen $l$.`get`$(i)$ und $l$.`set`$(i, e)$ eine Komplexität haben, die linear mit $i$ anwächst. Auf der anderen Seite ist der Aufwand bei einem Aufruf der Form $l$.`add`$(e)$ konstant.

(b) `ArrayList`
Diese Klasse wird durch ein Feld implementiert. Das hat den Vorteil, dass die Operationen $l$.`get`$(i)$ und $l$.`set`$(i, e)$ nur einen konstanten Aufwand erfordern. Dafür müssen bei dem Aufruf

$$l.\texttt{add}(0, e)$$

alle Elemente der Liste $l$ um eine Position nach rechts geschoben werden, so dass der Aufruf proportional zu der Anzahl der Elemente ist, die schon in der Liste $l$ abgespeichert sind.

3. Die Schnittstelle `Queue` beschreibt *Warteschlangen*. Eine Warteschlange ist eine Liste, bei Elemente nur am Ende eingefügt werden können und bei der nur die Elemente am Anfang entfernt werden können. Die Schnittstelle `Queue<E>` beschreibt die folgenden Methoden.

(a) `E element()`
Der Aufruf $q$.`element`() liefert das erste Element der Warteschlange $q$ als Ergebnis. Die Warteschlange $q$ wird dabei nicht verändert. Falls die Warteschlange leer ist, wird die Ausnahme `NoSuchElementException` geworfen.

(b) `boolean offer(E` $e$`)`
Der Aufruf $q$.`offer`$(e)$ fügt das Element $e$ an das Ende der Warteschlange $q$ an. Falls die Warteschlange voll ist und daher das Element nicht eingefügt werden konnte, liefert der Aufruf das Ergebnis `false`, ansonsten ist das Ergebnis `true`.

(c) `E peek()`
Der Aufruf $q$.`peek`() liefert das erste Element der Warteschlange $q$ als Ergebnis. Die Warteschlange $q$ wird dabei nicht verändert. Falls die Warteschlange leer ist, wird `null` zurück gegeben.

(d) `E poll()`
Der Aufruf $q$.`poll`() liefert das erste Element der Warteschlange $q$ als Ergebnis. Das Element wird dabei aus der Warteschlange $q$ entfernt. Falls die Warteschlange leer ist, wird `null` zurück gegeben.

(e) `E remove()`
Der Aufruf $q$.`remove`() liefert das erste Element der Warteschlange $q$ als Ergebnis. Das Element wird dabei aus der Warteschlange $q$ entfernt. Falls die Warteschlange leer ist, wird die Ausnahme `NoSuchElementException` geworfen.

Warteschlangen sind nützlich, wenn Daten in einer bestimmten Reihenfolge verarbeitet werden sollen. Die Schnittstelle `Queue` wird von der bereits diskutierten Klasse `LinkedList` implementiert.

### 7.6.2 Anwendungen von Mengen

Im ersten Semester hatten wir die Menge der Primzahlen, die kleiner als eine gegebene Zahl $n$ sind, mit dem folgenden Einzeiler berechnet:

```
primes := {2..n} - { p * q : p in {2..n}, q in {2..n} };
```

Wir wollen nun den selben Algorithmus in *Java* implementieren. Abbildung 7.28 auf Seite 133 zeigt das resultierende Programm, das wir jetzt diskutieren.

```
1  import java.util.*;
2
3  public class Primes
4  {
5      static Set<Integer> range(int low, int high) {
6          Set<Integer> result = new TreeSet<Integer>();
7          for (int i = low; i <= high; ++i) {
8              result.add(i);
9          }
10         return result;
11     }
12     static Set<Integer> products(Set<Integer> s1, Set<Integer> s2) {
13         Set<Integer> result = new TreeSet<Integer>();
14         for (Integer p : s1) {
15             for (Integer q : s2) {
16                 result.add(p * q);
17             }
18         }
19         return result;
20     }
21     static Set<Integer> primes(int n) {
22         Set<Integer> primes   = range(2, n);
23         Set<Integer> numbers  = range(2, n);
24         Set<Integer> products = products(numbers, numbers);
25         primes.removeAll(products);
26         return primes;
27     }
28     public static void main(String[] args) {
29         assert args.length == 1;
30         int n = Integer.parseInt(args[0]);
31         Set<Integer> primes = primes(n);
32         for (Integer p: primes) {
33             System.out.println(p);
34         }
35     }
36 }
```

Abbildung 7.28: Berechnung der Primzahlen mit Hilfe von Mengen

1. Die Methode `range`$(l, h)$ liefert für zwei Zahlen $l$ und $h$ die Menge aller ganzen Zahlen, die zwischen $l$ und $h$ inklusive liegen:

   $$\{n \in \mathbb{Z} \mid l \leq n \wedge n \leq h\}$$

   In SETL schreibt sich diese Menge als {`l..h`}.

Um diese Menge zu erzeugen, legen wir in Zeile 6 eine leere Menge an. Abschließend lassen wir in einer Schleife die Variable $i$ von $l$ bis $h$ laufen und fügen jedesmal $i$ zu der Menge hinzu.

2. Die Methode `products`$(s_1, s_2)$ berechnet für zwei Mengen $s_1$ und $s_2$ die Menge aller Produkte von Zahlen aus $s_1$ und $s_2$:

$$\{p \cdot q \mid p \in s_1 \wedge q \in s_2\}.$$

3. Die Methode `primes`$(n)$ berechnet die Menge aller Primzahlen bis zur Größe $n$. Dazu wird in Zeile 25 von der Menge $\{2..n\}$ die Menge aller Produkte $p \cdot q$ von Zahlen aus der Menge $\{2..n\}$ abgezogen. Die dann in der Menge `primes` verbleibenden Zahlen lassen sich nicht als nicht-triviales Produkt darstellen und sind folglich Primzahlen.

### 7.6.3 Die Schnittstelle `Map<K,V>`

Die Schnittstelle `Map<K,V>` beschreibt Abbildungen, deren Schlüssel den Typ `K` und deren Werte den Typ `V` haben. Mathematisch betrachtet repräsentiert ein Objekt vom Typ `Map<K,V>` eine Funktion, deren Definitions-Bereich eine Teilmenge von `K` ist und deren Werte-Bereich eine Teilmenge von `V` ist. Wir stellen die wichtigsten Methoden der Schnittstelle `Map<K,V>` vor.

1. `V get(Object` $k$`)`

   Für eine Abbildung $m$ und einen Schlüssel $k$ liefert der Aufruf $m$.`get`$(k)$ den Wert, den die Abbildung $m$ dem Schlüssel $k$ zuordnet. Falls die Abbildung $m$ dem Schlüssel $k$ keinen Wert zuordnet, dann wird als Ergebnis `null` zurück gegeben.

   In unserer Implementierung des Daten-Typs *Abbildung* hatten wir diese Funktion mit `find`() bezeichnet.

2. `boolean containsKey(K` $k$`)`

   Der Aufruf $m$.`containsKey`$(k)$ überprüft, ob die Abbildung $m$ dem Schlüssel $k$ einen Wert zuordnet. Da eine Abbildung einem Schlüssel auch den Wert `null` explizit zuordnen kann, kann diese Information nicht durch einen Aufruf von `get` gewonnen werden.

3. `V put(K` $k$`, V` $v$`)`

   Der Aufruf $m$.`put`$(k, v)$ ordnet dem Schlüssel $k$ den Wert $v$ zu. Außerdem wird der Wert zurück gegeben, der vorher unter dem Schlüssel $k$ in $m$ gespeichert war. Falls die Abbildung $m$ dem Schlüssel $k$ vorher keinen Wert zugeordnet hat, dann wird als Ergebnis `null` zurück gegeben.

   In unserer Implementierung des Daten-Typs *Abbildung* hatten wir diese Funktion mit `insert`() bezeichnet.

4. `V remove(K` $k$`)`

   Der Aufruf $m$.`remove`$(k)$ entfernt den Eintrag zu dem Schlüssel $k$ aus der Abbildung $m$. Gleichzeitig wird der Wert zurück gegeben, der vorher unter diesem Schlüssel gespeichert war.

   In unserer Implementierung des Daten-Typs *Abbildung* hatten wir diese Funktion mit `delete`() bezeichnet.

5. `void clear()`

   Der Aufruf $m$.`clear`() entfernt alle Einträge aus der Abbildung $m$.

6. `boolean containsValue(V` $v$`)`

   Der Aufruf $m$.`containsValue`$(v)$ überprüft, ob die Abbildung $m$ einem Schlüssel den Wert $v$ zuordnet. Die Komplexität dieses Aufrufs ist linear in der Zahl der Schlüssel, die in der Abbildung gespeichert sind.

7. `boolean isEmpty()`

   Der Aufruf $m$.`isEmpty`() überprüft, ob die Abbildung $m$ leer ist, also keinem Schlüssel einen Wert zuordnet.

8. `Set<K> keySet()`

   Der Aufruf $m$.`keySet`() liefert eine *Ansicht* (engl. *view*) der Menge aller Schlüssel, die in der Abbildung $m$ gespeichert sind. Mit dieser Ansicht können wir so arbeiten, als wäre es eine normale Menge. Wenn wir allerdings aus dieser Menge Schlüssel entfernen, so werden diese Schlüssel auch aus der Abbildung $m$ entfernt. Es ist nicht möglich, in diese Menge neue Schlüssel einzufügen. Jeder Versuch ein Element einzufügen liefert eine Ausnahme vom Typ

   `UnsupportedOperationException`.

9. `Collection<V> values()`

   Der Aufruf $m$.`values`() liefert eine Ansicht der Zusammenfassung aller Werte, die in der Abbildung $m$ gespeichert sind. Wenn wir aus dieser Ansicht Werte entfernen, dann verschwinden auch die entsprechenden Einträge in der Abbildung $m$. Es ist nicht möglich, Werte zu dieser Ansicht hinzuzufügen.

10. `void putAll(Map<K,V>` $t$`)`

    Nach dem Aufruf $m$.`putAll`($t$) enthält die Abbildung $m$ alle Zuordnungen, die in der Abbildung $t$ gespeichert sind. Enthält die Abbildung $t$ eine Zuordnung zu einem Schlüssel $k$ und enthält auch $m$ eine Zuordnung zu dem Schlüssel $k$, so wird die in $m$ bestehende Zuordnung durch die neue Zuordnung überschrieben.

11. `int size()`

    Der Aufruf $m$.`size`() liefert die Anzahl der Schlüssel, für die in der Zuordnung $m$ Werte gespeichert sind.

Die beiden wichtigsten Klassen, welche die Schnittstelle `Map<K,V>` implementieren, sind die Klasse `TreeMap<K,V>` und die Klasse `HashMap<K,V>`.

### Die Klasse `TreeMap<K,V>`

Die Klasse `TreeMap<K,V>` ist mit Hilfe von Rot-Schwarz-Bäumen implementiert. Um diese Klasse verwenden zu können, müssen die Schlüssel vergleichbar sein. Standardmäßig wird zum Vergleich zweier Schlüssel $k_1$ und $k_2$ die Methode

$k_1$.`compareTo`($k_2$)

aufgerufen, aber es gibt auch eine andere Möglichkeit. Dazu muss ein sogenanntes *Komparator-Objekt* erzeugt werden. Dies ist ein Objekt einer Klasse, die die Schnittstelle `Comparator<O>` implementiert. Diese Schnittstelle schreibt die Existenz einer Methode

`compare(O` $o_1$`, O` $o_2$`)`

vor. Ist $c$ ein Komparator, so vergleicht der Aufruf $c$.`compare`($o_1, o_2$) die beiden Objekte $o_1$ und $o_2$. Falls $o_1 < o_2$ ist, gibt der Aufruf eine negative Zahl zurück. Ist $o_1 > o_2$ wird entsprechend eine positive Zahl zurück gegeben. Sind $o_1$ und $o_2$ gleich, dann wird 0 zurück gegeben.

Die Klasse `TreeMap<K,V>` stellt die folgenden Konstruktoren zur Verfügung.

1. `TreeMap()`

   Dieser Konstruktor erzeugt eine leere Abbildung, bei der die Schlüssel mit Hilfe der Methode `compareTo`() verglichen werden.

2. `TreeMap(Comparator<K>` $c$`)`

   Dieser Konstruktor erzeugt eine leere Abbildung, bei der die Schlüssel mit Hilfe des Komparators $c$ verglichen werden.

3. `TreeMap(Map<K,V>` $m$`)`

   Dieser Konstruktor erzeugt eine neue Abbildung, die die selben Zuordnungen enthält wie die Abbildung $m$. Die Schlüssel werden dabei mit Hilfe der Methode `compareTo`() verglichen.

**Die Klasse `HashMap<K,V>`**

Falls die Schlüssel der Menge `K` nicht geordnet sind, kann die Klasse `HashMap<K,V>` verwendet werden, um eine Abbildung zu darzustellen. Diese Klasse wird durch eine Hash-Tabelle implementiert. Diese Klasse verfügt über die folgenden Konstruktoren.

1. `HashMap()`

   Dieser Konstruktor erzeugt eine leere Hash-Tabelle. Per Default ist hier der Load-Faktor auf 0.75 gesetzt, was zur Folge hat, dass mindestens ein Viertel der Einträge des Feldes, das der Hash-Tabelle zu Grunde liegt, leer sind. Das Feld selbst hat zunächst eine Größe von 16.

2. `HashMap(int` $n$`)`

   Dieser Konstruktor erzeugt eine leere Hash-Tabelle, für die das zu Grunde liegende Feld die Größe $n$ hat. Der Load-Faktor ist 0.75.

3. `HashMap(int` $n$`, float` $\alpha$`)`

   Dieser Konstruktor erzeugt eine leere Hash-Tabelle, für die das zu Grunde liegende Feld die Größe $n$ hat. Der Load-Faktor ist $\alpha$.

4. `HashMap(Map<K,V>` $m$`)`

   Dieser Konstruktor erzeugt eine Hash-Tabelle, die alle Zuordnungen aus der Abbildung $m$ enthält. Der Load-Faktor ist 0.75.

### 7.6.4 Anwendungen

Der Datentyp der Abbildungen wird überall dort benutzt, wo Tabellen abgespeichert werden müssen. Alle modernen Skriptsprachen stellen dem Benuter den abstrakten Datentyp *Abbildung* in der einen oder anderen Form zur Verfügung: In *Perl* [WS92] wird dieser Datentyp durch ein *assoziatives Feld* (im Orginal: *associative array*) implementiert, in *Lua* [Ier06, IdFF96] wird der entsprechende Datentyp als Tabelle (im Orginal: *table*) bezeichnet. In einem späteren Kapitel, wenn wir den Algorithmus von Dijkstra zur Bestimmung des kürzesten Weges in einem Graphen diskutieren, werden wir eine direkte Anwendungen des Datentyps Abbildung sehen.

## 7.7 Das Wolf-Ziege-Kohl-Problem

Zum Abschluss dieses Kapitels wollen wir zeigen, wie mit Hilfe von Mengen auch komplexere Problem gelöst werden können. Wir wählen dazu das *Wolf-Ziege-Kohl-Problem*, das wir bereits im ersten Semester bearbeitet haben:

> *Ein Bauer will mit einem Wolf, einer Ziege und einem Kohl über einen Fluß übersetzen, um diese als Waren auf dem Markt zu verkaufen. Das Boot ist aber so klein, dass er nicht zwei Waren gleichzeitig mitnehmen kann. Wenn er den Wolf mit der Ziege allein läßt, dann frißt der Wolf die Ziege und wenn er die Ziege mit dem Kohl allein läßt, dann frißt die Ziege den Kohl.*

```
1  findPath := procedure(x, y, r) {
2      p := { [x] };
3      while (true) {
4          oldP  := p;
5          p     := p + pathProduct(p, r);
6          found := { l in p | l[#l] == y };
7          if (found != {}) { return arb(found); }
8          if (p == oldP)   { return;            }
9      }
10 };
11 pathProduct := procedure(p, q) {
12     return { add(x,y) : x in p, y in q | x[#x] == y[1] && !cyclic(add(x,y)) };
13 };
14 cyclic := procedure(p) {
15     return #{ x : x in p } < #p;
16 };
17 add := procedure(p, q) {
18     return p + q[2..];
19 };
20 problem := procedure(s) {
21     return "goat" in s && "cabbage" in s || "wolf" in s && "goat" in s;
22 };
23 all := { "farmer", "wolf", "goat", "cabbage" };
24 p   := { [ s1, s2 ] : s1 in pow(all), s2 in pow(all)
25                     | s1 + s2 == all && s1 * s2 == {}
26        };
27 r1  := { [ [ s1, s2 ], [ s1 - b, s2 + b ] ]: [s1, s2 ] in p, b in pow(s1)
28          | "farmer" in b && #b <= 2 && !problem(s1 - b)
29        };
30 r2  := { [ [ s1, s2 ], [ s1 + b, s2 - b ] ]: [s1, s2] in p, b in pow(s2)
31          | "farmer" in b && #b <= 2 && !problem(s2 - b)
32        };
33 r   := r1 + r2;
34
35 start := [ all, {} ];
36 goal  := [ {}, all ];
37 path  := findPath(start, goal, r);
38 print(path);
```

Abbildung 7.29: Lösung des Wolf-Ziege-Kohl-Problems in SetlX.

Wir hatten damals das in Abbildung 7.29 auf Seite 137 gezeigte SETL2-Programm zur Lösung dieses Problems entwickelt. Wir wollen nun versuchen, diese Lösung in *Java* zu reimplementieren. Als erstes müssen wir überlegen, wie wir die Mengen, mit denen dort gearbeitet wird, darstellen wollen. In *Java* stehen hierfür die Klassen `TreeSet` und `HashSet` zur Auswahl. Wir entscheiden uns für die Klasse `TreeSet`, denn bei der Klasse `HashSet` gibt es Probleme, wenn wir Mengen von Mengen definieren wollen. Auch bei der Klasse `TreeSet` gibt es an dieser Stelle Probleme, allerdings sind diese Probleme einfacher zu lösen.

Versuchen wir das Programm aus Abbildung 7.29 in *Java* umzusetzen, so müssen wir benötigen wir bei der Umsetzung Mengen, deren Elemente selbst wieder Mengen sind. Die Elemente eines `TreeSet`s müssen aber vergleichbar sein, für eine beliebige Klasse `E` kann nur dann eine Klasse `TreeSet<E>` gebildet werden, wenn

```
E implements Comparable<E>
```

gilt. Leider wird die Schnittstelle `Comparable` von der Klasse `TreeSet` selber nicht implementiert. Damit erleiden wir Schiffbruch, wenn wir versuchen, eine Klasse der Form

```
TreeSet<TreeSet<E>>
```

zu erzeugen und damit zu arbeiten. Abbildung 7.30 zeigt ein Programm, bei dem wir eine Menge von Mengen von Zahlen anlegen wollen. Der Versuch scheitert in dem Moment, wo wir die zweite Menge in die Menge von Mengen einfügen wollen, denn dann merkt die virtuelle Maschine, dass Objekte der Klasse `TreeSet` das Interface `Comparable` nicht implementieren. Wir erhalten die Fehlermeldung

```
Exception in thread "main" java.lang.ClassCastException:
     java.util.TreeSet cannot be cast to java.lang.Comparable
```

Wir behelfen uns dadurch, dass wir eine neue Klasse `ComparableSet<E>` definieren, die das Interface `Comparable` implementiert.

```
1  import java.util.*;
2
3  public class SetOfSet {
4      public static void main(String[] args) {
5          TreeSet<TreeSet<Integer>> all = new TreeSet<TreeSet<Integer>>();
6          TreeSet<Integer> a = new TreeSet<Integer>();
7          a.add(1);
8          a.add(2);
9          a.add(3);
10         TreeSet<Integer> b = new TreeSet<Integer>();
11         b.add(1);
12         b.add(2);
13         b.add(3);
14         all.add(a);
15         all.add(b);
16         System.out.println(all);
17     }
18 }
```

Abbildung 7.30: Mengen von Mengen: Vorsicht Falle!

### 7.7.1 Die Klasse `ComparableSet`

```
1  import java.util.*;
2
3  public class ComparableSet<T extends Comparable<? super T>>
4      implements Comparable<ComparableSet<T>>,
5                 Iterable<T>
6  {
7      protected TreeSet<T> mSet;
8
9      public TreeSet<T> getSet()           { return mSet;             }
10     public ComparableSet()               { mSet = new TreeSet<T>(); }
11     public ComparableSet(TreeSet<T> set) { mSet = set;              }
12     public ComparableSet<T> deepCopy() {
13         return new ComparableSet<T>(new java.util.TreeSet<T>(mSet));
14     }
15     public boolean isEmpty()      { return mSet.isEmpty();    }
16     public boolean add(T element) { return mSet.add(element); }
17     public Iterator<T> iterator() { return mSet.iterator();   }
18     public int size()             { return mSet.size();       }
19
20     public T any()                { return mSet.first();      }
21     public String toString()      { return mSet.toString();   }
22
23     public boolean equals(Object x) {
24         if (x instanceof ComparableSet) {
25             ComparableSet cmpSet = (ComparableSet) x;
26             TreeSet       set    = cmpSet.mSet;
27             return mSet.equals(set);
28         }
29         return false;
30     }
31     public boolean member(T element) {
32         return mSet.contains(element);
33     }
34     public boolean isSubset(ComparableSet<T> set) {
35         return set.getSet().containsAll(mSet);
36     }
```

Abbildung 7.31: Die Klasse `ComparableSet<T>`, 1. Teil.

Die Abbildungen 7.31, 7.32, 7.33, 7.34 und 7.35 zeigen die Implementierung der Klasse `ComparableSet`. Wir diskutieren die Implementierung jetzt im Detail.

1. In Zeile 3 fordern wir für den Typ-Parameter `T` der Klasse `ComparableSet<T>`, dass

   ```
   T extends Comparable<? super T>
   ```

   gilt. Hier steht das Fragezeichen "`?`" für eine Oberklasse `O` von `T`. Die Forderung ist also, dass es eine Klasse `O` gibt, die eine Oberklasse der Elemente von `T` ist, und für die außerdem

   ```
   T extends Comparable<O>
   ```

   gilt. Das Interface `Comparable<O>` hat die Form

```
37      public int compareTo(ComparableSet<T> comparableSet) {
38          TreeSet<T>  set        = comparableSet.getSet();
39          Iterator<T> iterFirst  = mSet.iterator();
40          Iterator<T> iterSecond =  set.iterator();
41          while (iterFirst.hasNext() && iterSecond.hasNext()) {
42              T   first  = iterFirst .next();
43              T   second = iterSecond.next();
44              int cmp    = first.compareTo(second);
45              if (cmp == 0) {
46                  continue;
47              }
48              return cmp;
49          }
50          if (iterFirst.hasNext())  { return  1; }
51          if (iterSecond.hasNext()) { return -1; }
52          return 0;
53      }
54      public ComparableSet<T> union(ComparableSet<T> comparableSet) {
55          TreeSet<T> union = new TreeSet<T>(mSet);
56          union.addAll(comparableSet.getSet());
57          return new ComparableSet<T>(union);
58      }
59      public ComparableSet<T> intersection(ComparableSet<T> comparableSet) {
60          TreeSet<T> intersection = new TreeSet<T>(mSet);
61          intersection.retainAll(comparableSet.getSet());
62          return new ComparableSet<T>(intersection);
63      }
64      public ComparableSet<T> difference(ComparableSet<T> comparableSet) {
65          TreeSet<T> difference = new TreeSet<T>(mSet);
66          difference.removeAll(comparableSet.getSet());
67          return new ComparableSet<T>(difference);
68      }
69      public <S extends Comparable<? super S>> ComparableSet<Pair<T,S>>
70          product(ComparableSet<S> comparableSet)
71      {
72          TreeSet<Pair<T,S>> product = new TreeSet<Pair<T,S>>();
73          for (T x: mSet) {
74              for (S y: comparableSet.getSet()) {
75                  product.add(new Pair<T,S>(x, y));
76              }
77          }
78          return new ComparableSet<Pair<T,S>>(product);
79      }
80      public ComparableSet<ComparableSet<T>> powerSet() {
81          return new ComparableSet<ComparableSet<T>>(powerSet(mSet));
82      }
```

Abbildung 7.32: Die Klasse `ComparableSet<T>`, 2. Teil.

```
interface Comparable<O> { int compareTo(O o); }
```

```
83      private static <S extends Comparable<? super S>> TreeSet<ComparableSet<S>>
84          powerSet(TreeSet<S> set)
85      {
86          if (set.isEmpty()) {
87              TreeSet<ComparableSet<S>> power = new TreeSet<ComparableSet<S>>();
88              ComparableSet<S>          empty = new ComparableSet<S>();
89              power.add(empty);
90              return power;
91          }
92          S          last = set.last();
93          TreeSet<S> rest = (TreeSet<S>) set.headSet(last);
94          TreeSet<ComparableSet<S>> powerRest = powerSet(rest);
95          TreeSet<ComparableSet<S>> powerSet  = cloneSet(powerRest);
96          addElement(powerRest, last);
97          powerSet.addAll(powerRest);
98          return powerSet;
99      }
100     private static <S extends Comparable<? super S>> void
101         addElement(TreeSet<ComparableSet<S>> setOfSets, S element)
102     {
103         for (ComparableSet<S> set: setOfSets) {
104             set.add(element);
105         }
106     }
107     private static <S extends Comparable<? super S>> TreeSet<ComparableSet<S>>
108         cloneSet(TreeSet<ComparableSet<S>> set)
109     {
110         TreeSet<ComparableSet<S>> result = new TreeSet<ComparableSet<S>>();
111         for (ComparableSet<S> s: set) {
112             result.add(s.deepCopy());
113         }
114         return result;
115     }
116     public static <T extends Comparable<? super T>> ComparableSet<T>
117         singleton(T element)
118     {
119         TreeSet<T> set = new TreeSet<T>();
120         set.add(element);
121         return new ComparableSet<T>(set);
122     }
123     public static <T extends Comparable<? super T>> ComparableSet<T>
124         doubleton(T first, T second)
125     {
126         TreeSet<T> set = new TreeSet<T>();
127         set.add(first);
128         set.add(second);
129         return new ComparableSet<T>(set);
130     }
```

Abbildung 7.33: Die Klasse `ComparableSet<T>`, 3. Teil.

```
131     public static ComparableSet<Integer> range(int low, int high) {
132         ComparableSet<Integer> result = new ComparableSet<Integer>();
133         for (int i = low; i <= high; ++i) {
134             result.add(i);
135         }
136         return result;
137     }
138     public static <U extends Comparable<? super U>,
139                    V extends Comparable<? super V>,
140                    W extends Comparable<? super W>> ComparableSet<Pair<U,W>>
141         compose(ComparableSet<Pair<U,V>> R1, ComparableSet<Pair<V,W>> R2)
142     {
143         ComparableSet<Pair<U,W>> result = new ComparableSet<Pair<U,W>>();
144         for (Pair<U,V> xy: R1) {
145             for (Pair<V,W> yz: R2) {
146                 if (xy.getSecond().equals(yz.getFirst())) {
147                     result.add(new Pair<U,W>(xy.getFirst(), yz.getSecond()));
148                 }
149            }
150         }
151         return result;
152     }
153     public ComparableSet<T> select(Selector<T> selector) {
154         TreeSet<T> result = new TreeSet<T>();
155         for (T element: mSet) {
156             if (selector.select(element)) { result.add(element); }
157         }
158         return new ComparableSet<T>(result);
159     }
160     public <S extends Comparable<? super S>> ComparableSet<S>
161         transform(Transformer<S, T> transformer)
162     {
163         TreeSet<S> result = new TreeSet<S>();
164         for (T element: mSet) {
165             result.add(transformer.transform(element));
166         }
167         return new ComparableSet<S>(result);
168     }
```

Abbildung 7.34: Die Klasse `ComparableSet<T>`, 4. Teil.

Also spezifiziert der String "`T extends Comparable<? super T>`", dass Elemente der Klasse `T` mit Elementen jeder Oberklasse von `T` vergleichbar sein müssen. Im ersten Moment denken Sie eventuell, dass es reichen würde, wenn wir

```
ComparableSet<T extends ComparableSet<T>>
```

schreiben würden. Das würde aber dann nicht mehr funktionieren, wenn wir zunächst eine Klasse `A` hätten, die als

```
class A implements Comparable<A> { ... }
```

```
1       public static <T extends Comparable<? super T>,
2                      X extends Comparable<? super X>,
3                      Y extends Comparable<? super Y>> ComparableSet<T>
4       combineSets(ComparableSet<X> S1,
5                   ComparableSet<Y> S2,
6                   Combinator<T,X,Y> combinator)
7       {
8           TreeSet<T> result = new TreeSet<T>();
9           for (X x: S1) {
10              for (Y y: S2) {
11                  result.add(combinator.combine(x, y));
12              }
13          }
14          return new ComparableSet<T>(result);
15      }
16  }
```

Abbildung 7.35: Die Klasse `ComparableSet<T>`, 5. Teil.

definiert ist und von dieser Klasse später eine Klasse `B` ableiten, welche die Methode *compareTo*() von der Klasse `A` erbt. Das Problem ist, dass für `B` in dem Fall nur

`B implements Comparable<A>`

gilt und eben nicht

`B implements Comparable<B>`.

Damit könnten wir keine Klasse `ComparableSet<B>` mehr bilden und müssten statt dessen auf die ungenauere Klasse `ComparableSet<A>` ausweichen, wobei wir Typinformationen verlieren würden.

2. Die Klasse `ComparableSet` ist letztlich nur eine Verpackung eines Objektes der Klasse `TreeSet`, das in Zeile 7 durch die Member-Variable `mSet` definiert wird. An dieser Stelle stellt sich die Frage, warum wir die Klasse `ComparableSet` nicht von der Klasse `TreeSet` ableiten. Der Grund für dieses Vorgehen ist, dass die wesentlichen Methoden, die in der Klasse `TreeSet` implementiert sind, die Mengen, auf denen sie arbeiten, verändern. Wenn wir beispielsweise für zwei Mengen $a$ und $b$ die Methode

   $a.addAll(b)$

   aufrufen um die Vereinigung $a \cup b$ zu berechnen, so hat die Menge $a$ nach diesem Aufruf ihren alten Wert verloren. Solche Seiteneffekte sind für die Art und Weise, in der wir Mengen benutzen wollen, sehr unerwüscht. Die Methoden, die dem Benutzer der Klasse `ComparableSet` zur Verfügung gestellt werden, sollen frei von Seiteneffekten sein. Beispielsweise werden wir eine Methode *union*() implementieren, die so beschaffen ist, dass für zwei Mengen $a$ und $b$

   $a.union(b) = a \cup b$

   gilt und dass außerdem die Variablen $a$ und $b$ bei dieser Operation ihre alten Werte behalten. Diese Methode *union*() soll die in der Klasse `TreeSet` implementierte Klasse *addAll*() ersetzen. Würden wir die Klasse `ComparableSet` von der Klasse `TreeSet` ableiten, so hätte der Beutzer immer noch die Möglichkeit, beispielsweise die Methode *addAll*() zu benutzen. Dies soll verhindert werden, denn nur so können wir für die Klasse `ComparableSet` die folgende Garantie geben: Eine Variable vom Typ `ComparableSet` ändert Ihren Wert nur, wenn ihr explizit ein neuer Wert zugewiesen wird.

3. Die Methode *deepCopy*() erzeugt eine Kopie einer gegebenen Menge. Diese Methode ist einem Konstruktor-Aufruf der Form

   `new ComparableSet<T>`$(s)$

   dann vorzuziehen, wenn sich der `TreeSet` $s$ ändern kann, denn dann würde sich auch der erzeugte `ComparableSet<T>` ändern.

4. Eine Reihe von Methoden werden in den Zeilen 15 – 21 dadurch implementiert, dass die entsprechenden Methoden der Klasse `TreeSet` aufgerufen werden.

5. Die Methode *equals*() ermöglicht uns, ein Objekt vom Typ `ComparableSet` mit einem beliebigen anderem Objekt zu vergleichen.

6. Der Aufruf $c.member(e)$ überprüft, ob $e$ ein Element der Menge $c$ ist.

7. Der Aufruf $c.isSubset(s)$ überprüft, ob $c$ eine Teilmenge der Menge $s$ ist.

8. Der Aufruf $c.compareTo(s)$ vergleicht die Menge $c$ mit der Menge $s$. Der Vergleich ist ein *lexikografischer* Vergleich. Da sowohl $c$ als auch $s$ geordnete Mengen sind, lassen sich die Elemente von $c$ und $s$ der Größe nach auflisten. Wir vergleichen nun die Elemente von $c$ und $s$ paarweise, wobei wir mit dem kleinsten Element beginnen. Das erste Element, bei dem sich die Mengen $c$ und $s$ unterscheiden, entscheidet dann über den Vergleich. Werden beispielsweise die Mengen

   $$c = \{2, 7\} \quad \text{und} \quad s = \{2, 3, 7, 14\}$$

   auf diese Weise verglichen, so vergleichen wir zunächst das kleinste Element beider Mengen. Das ist die 2. Da dieses Element für beide Mengen gleich ist, gehen wir zum zweiten Element. In der Menge $c$ finden wir hier die 7, in der Menge $s$ steht hier die 3. Da $3 < 7$ ist, ist folgern wir $s < c$.

   Falls die Ordnung auf den Elementen eine totale Ordnung ist, so läßt sich zeigen, dass auch die lexikografische Ordnung, die auf Mengen von Mengen definiert ist, eine totale Ordnung ist.

9. Anschließend definieren wir die Methoden *union*(), *intersection*() und *difference*() so, dass die Methoden die Vereinigung, den Schnitt und die Mengendifferenz berechnen. Der wesentliche Unterschied zu den analogen Methoden der Klasse `TreeSet` besteht hier darin, dass beispielsweise bei dem Aufruf $c.union(s)$ die Menge $c$ nicht verändert wird.

10. Der Aufruf $c.product(s)$ bildet das kartesische Produkt der Mengen $c$ und $s$, es gilt also

    $$c.product(s) = \big\{ \langle x, y \rangle \mid x \in c \wedge y \in s \big\}.$$

    Damit die Implementierung dieser Methode funktioniert, muss die Klasse `Pair` so definiert sein, dass die Klasse `Pair<S,T>` das Interface `Comparable` implementiert. Abbildung 7.39 auf Seite 146 zeigt die Implementierung dieser Klasse.

11. Der Aufruf *c.powerSet*() berechnet die Potenz-Menge von $c$. Die Implementierung geschieht unter Zuhilfenahme der statischen Methode $powerSet(s)$, die für einen gegeben `TreeSet` $s$ die Potenz-Menge berechnet. Die Implementierung der Methode *powerSet*() basiert auf den folgenden Gleichungen:

    (a) $powerSet\big(\{\}\big) = \big\{\{\}\big\}$,

    (b) $powerSet\big(A \cup \{x\}\big) = powerSet(A) \cup \big\{\{x\} \cup s : s \in powerSet(A)\big\}$.

    Die Methode *headSet*(), die bei der Realisierung benutzt wird, kann wir folgt spezifiziert werden: Der Aufruf $s.headSet(l)$ liefert alle alle Elemente aus der Menge $s$, die kleiner als $l$ sind:

    $$s.headSet(l) = \{x \in s \mid x < l\}.$$

Der Aufruf $s.last()$ liefert das größte Element der Menge $s$. Damit können wir eine Menge $s$ in der Form

$$s = \{s.last()\} \cup s.headSet(s.last())$$

in eine einelementige Menge und die restlichen Elemente disjunkt zerlegen.

12. Für eine Menge von Mengen $S$ fügt der Aufruf $addElement(S, e)$ das Element $e$ in jede Menge aus $S$ ein:

$$addElement(S, e) = \big\{m \cup \{e\} \mid m \in S\big\}.$$

13. Für eine Menge von Mengen $S$ liefert der Aufruf $c.cloneSet(S)$ eine Kopie der Menge $S$, die nicht die Mengen von $S$ enthält sondern Kopien dieser Mengen.

14. Der Aufruf $singleton(x)$ liefert die Menge $\{x\}$.

15. Der Aufruf $doubleton(x, y)$ liefert die Menge $\{x, y\}$.

16. Der Aufruf $range(a, b)$ liefert die Menge

$$\{n \in \mathbb{Z} \mid a \leq n \wedge n \leq b\}.$$

17. Die Methode $compose(R_1, R_2)$ berechnet das relationale Produkt $R_1 \circ R_2$ der Relationen $R_1$ und $R_2$. Dieses Produkt ist wie folgt definiert:

$$R_1 \circ R_2 = \big\{\langle x, z\rangle \mid \exists y : \langle x, y\rangle \in R_1 \wedge \langle y, z\rangle \in R_2\big\}.$$

18. Die Methode *select*() gestattet die Mengenbildung durch Auswahl. Der Aufruf $c.select(s)$ berechnet für eine Menge $c$ und einen *Selektor* $s$ die Menge

$$\{x \in c \mid s.select(x)\}.$$

Ein Selektor ist dabei einfach ein Objekt $s$, dass eine Methode $s.select(x)$ zur Verfügung stellt. Diese Methode gibt als Ergebnis entweder `true` oder `false` zurück. Abbildung 7.36 zeigt das Interface `Selector`.

```
1  public interface Selector<T> {
2      public boolean select(T element);
3  }
```

Abbildung 7.36: Das Interface `Selector`.

19. Die Methode *transform*() berechnet eine Bild-Menge. Der Aufruf $c.transform(t)$ berechnet für einen *Transformer* $t$ die Menge

$$\big\{t.transform(x) \mid x \in c\big\}.$$

Ein Transformer ist hier ein Objekt, dass eine Methode *transform*() zur Verfügung stellt, mit der Elemente einer Menge `T` in Elemente einer Menge `S` umgewandelt werden können. Abbildung 7.37 zeigt das Interface `Transformer`.

20. Die Methode $combineSets(s_1, s_2, k)$ verknüpft zwei Mengen mit einem *Kombinator* $k$. Es gilt

$$combineSets(s_1, s_2, k) = \big\{k.combine(x, y) \mid x \in s_1 \wedge y \in s_2\big\}$$

Ein *Kombinator* ist ein Objekt, dass eine Methode *combine*() zur Verfügung stellt, mit dem zwei Elemente verknüpft werden können. Abbildung 7.38 zeigt das Interface `Combinator`.

```
1  public interface Transformer<S extends Comparable<? super S>,
2                               T extends Comparable<? super T>>
3  {
4      public S transform(T x);
5  }
```

Abbildung 7.37: Das Interface `Transformer`.

```
1  public interface Combinator<T, X, Y>
2  {
3      public T combine(X x, Y y);
4  }
```

Abbildung 7.38: Das Interface `Combinator`.

```
1  public class Pair<S extends Comparable<? super S>,
2                    T extends Comparable<? super T>>
3      implements Comparable<Pair<S,T>>
4  {
5      S mFirst;
6      T mSecond;
7
8      public Pair(S first, T second) {
9          mFirst  = first;
10         mSecond = second;
11     }
12     public int compareTo(Pair<S, T> pair) {
13         int cmpFirst = mFirst.compareTo(pair.getFirst());
14         if (cmpFirst < 0 || cmpFirst > 0) {
15             return cmpFirst;
16         }
17         return mSecond.compareTo(pair.getSecond());
18     }
19     public String toString() {
20         return "<" + mFirst + ", " + mSecond + ">";
21     }
22     public S getFirst()  { return mFirst;  }
23     public T getSecond() { return mSecond; }
24
25     public void setFirst(S first) {
26         mFirst = first;
27     }
28     public void setSecond(T second) {
29         mSecond = second;
30     }
31 }
```

Abbildung 7.39: Die Klasse `Pair`.

```
1  public class ComparableList<T extends Comparable<? super T>>
2      extends LinkedList<T>
3      implements Comparable<ComparableList<T>>
4  {
5      public int compareTo(ComparableList<T> comparableList) {
6          Iterator<T> iterFirst  = iterator();
7          Iterator<T> iterSecond = comparableList.iterator();
8          while (iterFirst.hasNext() && iterSecond.hasNext()) {
9              T   first  = iterFirst .next();
10             T   second = iterSecond.next();
11             int cmp    = first.compareTo(second);
12             if (cmp == 0) {
13                 continue;
14             }
15             return cmp;
16         }
17         if (iterFirst.hasNext()) {
18             return 1;
19         }
20         if (iterSecond.hasNext()) {
21             return -1;
22         }
23         return 0;
24     }
25 }
```

Abbildung 7.40: Die Klasse `ComparableList`.

### 7.7.2 Die Klasse `ComparableList`

Nachdem wir nun mit Mengen so arbeiten können, wie wir dass in dr Sprache SetlX gewohnt sind, benötigen wir als nächstes eine Klasse `ComparableList`, die Listen darstellt und außerdem das Interface `Comparable` implementiert. Die Implementierung dieser Klasse ist in Abbildung 7.40 auf Seite 147 gezeigt. Bei der Lösung des Wolf-Ziege-Kohl-Problems haben wir Listen nur benötigt, um die verschiedenen Pfade in einem Graphen darzustellen. Wir haben keine Operationen benutzt die Listen manipulieren. Daher reicht es für dieses Beispiel aus, wenn wir die Klasse `ComparableList` von `LinkedList` ableiten. In diesem Fall muss nur die Methode *compareTo*() implementiert werden. Dies geschieht ähnlich wie bei Mengen über einen lexikografischen Vergleich der beiden Listen.

### 7.7.3 Lösung des Wolf-Ziege-Kohl-Problems in *Java*

Nach dem wir uns im letzten Abschnitt einen Rahmen geschaffen haben, in dem konzeptionell die selben Funktionen wie in SetlX zur Verfügung stehen, können wir nun daran gehen, das im ersten Semester entwickelte SetlX-Programm in *Java* zu übersetzen. Abbildung 7.41 zeigt das Ergebnis dieser Übersetzung.

1. In den ersten beiden Zeilen definieren wir zwei Abkürzungen. Zum einen möchten wir

   `Point` als Abkürzung für `ComparableSet<String>`

   benutzen, zum anderen steht

   `PointPair` als Abkürzung für `Pair<Point, Point>`.

   Durch die Verwendung dieser Abkürzungen ersparen wir es uns, später mit unlesbaren Klas-

```
#define Point     ComparableSet<String>
#define PointPair Pair<Point, Point>

public class WolfZiegeKohl
{
    public static void main(String args[]) {
        Point all = new Point();
        all.add("Bauer");
        all.add("Wolf");
        all.add("Ziege");
        all.add("Kohl");
        ComparableSet<Point> p = all.powerSet();
        ComparableSet<PointPair> r = new ComparableSet<PointPair>();
        for (Point s: p) {
            for (Point b : s.powerSet()) {
                Point sb = s.difference(b);
                if (b.member("Bauer") && b.size() <= 2 && !problem(sb))
                {
                    PointPair ssb = new PointPair(s, sb);
                    r.add(ssb);
                }
            }
        }
        for (Point s: p) {
            Point as = all.difference(s);
            for (Point b : as.powerSet()) {
                if (b.member("Bauer") && b.size() <= 2 &&
                    !problem(as.difference(b))           )
                {
                    Point sb = s.union(b);
                    PointPair ssb = new PointPair(s, sb);
                    r.add(ssb);
                }
            }
        }
        Point goal = new Point();
        Relation<Point> relation = new Relation(r);
        ComparableList<Point> path = relation.findPath(all, goal);
        for (Point left : path) {
            Point right = all.difference(left);
            System.out.println(left + ", " + right);
        }
    }
    static boolean problem(Point s) {
        return (s.member("Ziege") && s.member("Kohl")) ||
               (s.member("Wolf") && s.member("Ziege"));
    }
}
```

Abbildung 7.41: Lösung des Wolf-Ziege-Kohl-Problems.

senbezeichnungen der Form

```
ComparableSet<Pair<ComparableSet<String>, ComparableSet<String>>>
```

arbeiten zu müssen. Leider gibt es in der Sprache *Java* (im Gegensatz zu der Sprache `C#`) keine Möglichkeit, Abkürzungen zu definieren, denn ein zu einem `typedef` analoges Konstrukt, wie Sie es beispielsweise in der Sprache `C` finden, gibt es in *Java* nicht. Wir behelfen uns mit einem Trick und verwenden den `C`-Präprozessor, denn dieser kann Makros expandieren. Daher haben wir in den ersten beiden Zeilen die entsprechenden Abkürzungen mit Hilfe der Präprozessor-Direktive "`#define`" definiert. Um diese Abkürzungen expandieren zu können, speichern wir das in Abbildung 7.41 gezeigte Programm in einer Datei mit dem Namen `WolfZiegeKohl.jpre` und rufen dann den `C`-Präprozessor mit dem Befehl

```
cpp -P WolfZiegeKohl.jpre WolfZiegeKohl.java
```

auf. Dieser Befehl expandiert die Makro-Definitionen und schreibt das Ergebnis in die Datei `WolfZiegeKohl.java`. Die Option "`-P`" ist hier notwendig um die Zeilenmarkierungen, die andernfalls vom Präprozessor erzeugt würden, zu unterdrücken.

2. Die Klasse `WolfZiegeKohl` enthält nur die Methode *main*(). Diese Methode löst das Problem und gibt die Lösung (allerdings sehr spartanisch) aus. Zunächst bilden wir dort die Menge `all`, die die Strings "`Bauer`", "`Wolf`", "`Ziege`", "`Kohl`" enthält.

3. Die Menge `p` ist die Potenz-Menge von `all`.

4. Die Menge `r` beschreibt die Zustandsübergangsrelation. Diese Relation wird in den beiden `for`-Schleifen in den Zeilen 14 – 23 und 24 – 35 berechnet. Die erste `for`-Schleife berechnet die Übergänge, bei denen das Boot von links nach rechts übersetzt. Mathematisch können diese Übergänge wie folgt zu einer Relation zusammen gefaßt werden:

   $$\big\{\langle s, s-b\rangle \colon s \in p, b \in 2^s \mid \text{"}\texttt{bauer}\text{"} \in b \wedge \#b \leq 2 \wedge \neg \mathit{problem}(s-b)\big\}$$

   Die Variable $s$ ist hier die Menge der Objekte am linken Ufer und ist daher ein Element der Menge $p$, denn $p$ ist ja die Potenz-Menge von `all`. Die Variable $b$ bezeichnet die Menge der Objekte, die im Boot vom linken Ufer zum rechten Ufer übersetzen. Diese Menge ist eine Teilmenge von $s$ und damit ein Element der Potenz-Menge von $s$. Die Menge `sb` besteht aus den Objekten, die nach der Überfahrt am linken Ufer verbleiben. Das ist gerade die Mengendifferenz $s \backslash b$.

   Damit eine Überfahrt legal ist, müssen folgende Bedingungen erfüllt sein:

   (a) Der Bauer muss im Boot sitzen:

   ```
   b.member("Bauer").
   ```

   (b) Im Boot dürfen sich maximal zwei Objekte befinden:

   ```
   b.size() <= 2
   ```

   (c) Es darf nach der Überfahrt am linken Ufer kein Problem geben:

   ```
   !problem(sb)
   ```

   Diese Bedingungen werden durch die `if`-Abfrage in Zeile 17 sichergestellt. Wenn die Bedingungen erfüllt sind, wird das Paar $\langle \texttt{s}, \texttt{sb}\rangle$ der Relation $r$ hinzugefügt.

5. Die Übergänge, bei denen das Boot von rechts nach links fährt, werden analog berechnet. Mathematisch hat diese Relation die Form

   $$\big\{\langle s, s+b\rangle \colon s \in p, b \in 2^{\texttt{all}\backslash s} \mid \text{"}\texttt{Bauer}\text{"} \in b \wedge \#B \leq 2 \wedge \neg \texttt{problem}((\texttt{all}\backslash s)\backslash b)\big\}.$$

   Da $s$ wieder die Menge der Objekte am linken Ufer ist, finden wir die Menge der Objekte am rechten Ufer, indem wir die Menge $\texttt{all}\backslash s$ bilden. Diese Menge wird im Programm mit

`as` bezeichnet. Das Boot, also die Menge der Objekte, die von rechts nach links übersetzen, ist daher nun eine Teilmenge von `as`. Der Rest der Rechnung ist nun analog zum ersten Fall.

6. Anschließend wird in Zeil 36 der Zielzustand definiert: Am Ende sollen alle Objekte am rechten Ufer sein. Links ist dann niemand mehr, folglich ist die entsprechende Menge leer. Da der Aufruf

   ```
   new Point()
   ```

   vom Präprozessor zu

   ```
   ComparableSet<String>()
   ```

   expandiert wird, wird in Zeile 36 als Zielzustand tatsächlich die leere Menge berechnet.

7. Die Zuweisung "`relation = new Relation(r)`" wandelt nun die Menge von Paaren von Zuständen in ein Objekt der Klasse `Relation` um, die es uns über den Aufruf von *findPath*() ermöglicht, einen Weg vom Start zum Ziel zu berechnen.

8. Die Methode *problem*($s$) überprüft für eine gegebene Menge von Objekten, ob es zu einem Problem kommt, weil entweder die Ziege den Kohl oder der Wolf die Ziege frißt.

Als letztes diskutieren wir die Implementierung der Klasse `Relation`. Diese Klasse verwaltet eine Menge von Paaren, die einen Graphen repräsentiert. Diese Klasse stellt drei Methoden zur Verfügung.

1. Die Methode $r$.`findPath`$(x, y)$ berechnet einen Pfad, der von dem Punkt $x$ zu dem Punkt $y$ führt.

2. Die Methode $R.pathProduct(P)$ berechnet für eine Relation $R$ und eine Menge von Pfaden $P$ das sogenannte Pfad-Produkt $P \bullet R$, das für eine Relation $R$ und eine Menge von Pfaden $P$ wie folgt definiert ist:

   $$P \bullet R = \left\{ l + [y] \mid l \in P \wedge \langle x, y \rangle \in R \wedge last(l) = x \right\}.$$

   Für einen Pfad $p$ bezeichnet dabei $last(p)$ den letzten Punkt des Pfades. Anschaulich gesehen werden bei der Berechnung von $P \bullet R$ die Pfade aus $P$ um die Relation $R$ verlängert: Wenn einerseits eine Liste $l$ aus $P$ mit einem Punkt $x$ endet und wenn andererseits die Relation $R$ ein Paar der Form $\langle x, y \rangle$ enthält, dann kann das Element $y$ an die Liste $l$ angehängt werden.

3. Die Methode *cyclic*() überprüft für eine gegebene Liste $l$, ob diese Liste ein Element mehrfach enthält und damit einen zyklischen Pfad darstellt. Um dies zu prüfen wird die Liste in eine Menge umgewandelt. Wenn die Menge genauso viele Elemente enthält wie die Liste, dann kann die Liste kein Element doppelt enthalten haben und ist damit nicht zyklisch.

```
1  public class Relation<E extends Comparable<? super E>>
2  {
3      ComparableSet<Pair<E, E>> mR;
4
5      public Relation(ComparableSet<Pair<E, E>> r) { mR = r; }
6
7      public ComparableList<E> findPath(E start, E goal) {
8          ComparableList<E> first = new ComparableList<E>();
9          first.add(start);
10         ComparableSet<ComparableList<E>> p    = ComparableSet.singleton(first);
11         ComparableSet<ComparableList<E>> oldP = null;
12         while (true) {
13             oldP = p;
14             p    = p.union(pathProduct(p));
15             for (ComparableList<E> l : p) {
16                 if (l.getLast().compareTo(goal) == 0) {
17                     return l;
18                 }
19             }
20             if (p.compareTo(oldP) == 0) {
21                 return null;
22             }
23         }
24     }
25     private ComparableSet<ComparableList<E>>
26         pathProduct(ComparableSet<ComparableList<E>> P)
27     {
28         ComparableSet<ComparableList<E>> result =
29             new ComparableSet<ComparableList<E>>();
30         for (ComparableList<E> p : P) {
31             for (Pair<E, E> q : mR) {
32                 if (p.getLast().compareTo(q.getFirst()) == 0) {
33                     ComparableList<E> pq = new ComparableList<E>(p);
34                     E second = q.getSecond();
35                     pq.add(second);
36                     if (!cyclic(pq)) {
37                         result.add(pq);
38                     }
39                 }
40             }
41         }
42         return result;
43     }
```

Abbildung 7.42: Die Klasse `Relation`.

```
44      private static <T extends Comparable<? super T>>
45              boolean cyclic(ComparableList<T> l)
46      {
47          ComparableSet<T> all = new ComparableSet<T>();
48          for (T x : l) {
49              all.add(x);
50          }
51          return all.size() < l.size();
52      }
53  }
```

Abbildung 7.43: Die Methode *cyclic*().

# Kapitel 8

# Prioritäts-Warteschlangen

Um den Begriff der *Prioritäts-Warteschlange* zu verstehen, betrachten wir zunächst den Begriff der *Warteschlange*. Dort werden Daten hinten eingefügt und vorne werden Daten entnommen. Das führt dazu, dass Daten in der selben Reihenfolge entnommen werden, wie sie eingefügt werden. Anschaulich ist das so wie bei der Warteschlange vor einer Kino-Kasse, wo die Leute in der Reihenfolge bedient werden, in der sie sich anstellen. Bei einer Prioritäts-Warteschlange haben die Daten zusätzlich Prioritäten. Es wird immer das Datum entnommen, was die höchste Priorität hat. Anschaulich ist das so wie im Wartezimmer eines Zahnarztes. Wenn Sie schon eine Stunde gewartet haben und dann ein Privat-Patient aufkreuzt, dann müssen Sie halt noch eine Stunde warten, weil der Privat-Patient eine höhere Priorität hat.

Prioritäts-Warteschlangen spielen in vielen Bereichen der Informatik eine wichtige Rolle. Wir werden Prioritäts-Warteschlangen später sowol in dem Kapitel über Daten-Kompression als auch bei der Implementierung des Algorithmus zur Bestimmung kürzester Wege in einem Graphen einsetzen. Daneben werden Prioritäts-Warteschlangen unter anderem in Simulations-Systemen und beim Scheduling von Prozessen in Betriebs-Systemen eingesetzt.

## 8.1 Definition des ADT *PrioQueue*

Wir versuchen den Begriff der Prioritäts-Warteschlange jetzt formal durch Definition eines abstrakten Daten-Typs zu fassen. Wir geben hier eine eingeschränkte Definition von Prioritäts-Warteschlangen, die nur die Funktionen enthält, die wir später für den Algorithmus von Dijkstra benötigen.

**Definition 18 (Prioritäts-Warteschlange)**
*Wir definieren den abstrakten Daten-Typ der* Prioritäts-Warteschlange *wie folgt:*

1. *Als Namen wählen wir PrioQueue.*
2. *Die Menge der Typ-Parameter ist*
   $$\{Priority, Value\}.$$
   *Dabei muß auf der Menge Priority eine totale Quasi-Ordnung $<$ existieren, so dass wir die Prioritäten verschiedener Elemente vergleichen können.*
3. *Die Menge der Funktions-Zeichen ist*
   $$\{prioQueue, insert, remove, top, change\}.$$
4. *Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:*
   (a) $prioQueue : PrioQueue$
   *Der Aufruf "prioQueue()" erzeugt eine leere Prioritäts-Warteschlange.*

(b) *top* : *PrioQueue* $\to$ (*Priority* $\times$ *Value*) $\cup \{\Omega\}$
*Der Aufruf* $Q$.*top*() *liefert ein Paar* $\langle p, v\rangle$. *Dabei ist* $v$ *ein Element aus* $Q$, *das eine maximale Priorität hat.* $p$ *ist die Priorität des Elements* $v$.

(c) *insert* : *PrioQueue* $\times$ *Priority* $\times$ *Value* $\to$ *PrioQueue*
*Der Aufruf* $Q$.*insert*$(p, v)$ *fügt das Element* $v$ *mit der Priorität* $p$ *in die Prioritäts-Warteschlange* $Q$ *ein.*

(d) *remove* : *PrioQueue* $\to$ *PrioQueue*
*Der Aufruf* $Q$.*remove*() *entfernt aus der Prioritäts-Warteschlange* $Q$ *ein Element, das eine maximale Priorität hat.*

(e) *change* : *PrioQueue* $\times$ *Priority* $\times$ *Value* $\to$ *PrioQueue*
*Der Aufruf* $Q$.*change*$(p, v)$ *ändert die Priorität des Elements* $v$ *in der Prioritäts-Warteschlange* $Q$ *so ab, dass* $p$ *die neue Priorität dieses Elements ist. Wir setzen dabei voraus, dass einerseits dass Element* $v$ *in der Prioritäts-Warteschlange* $Q$ *auftritt und dass andererseits die neue Priorität mindestens so hoch ist wie die Priorität, die* $v$ *vorher hatte.*

5. *Bevor wir das Verhalten der einzelnen Methoden axiomatisch definieren, müssen wir noch festlegen, was wir unter den* Prioritäten *verstehen wollen, die den einzelnen Elementen aus Value zugeordnet sind. Wir nehmen an, dass die Prioritäten Elemente einer Menge Priority sind und dass auf der Menge Priority eine totale Quasi-Ordnung* $\leq$ *existiert. Falls dann* $p_1 < p_2$ *ist, sagen wir, dass* $p_1$ *eine höhere Priorität als* $p_2$ *hat. Dies erscheint im ersten Moment vielleicht paradox. Es wird aber später verständlich, wenn wir den Algorithmus zur Berechnung kürzester Wege von Dijkstra diskutieren. Dort sind die Prioritäten Entfernungen im Graphen und die Priorität eines Knotens ist um so höher, je näher der Knoten zu einem als* Startknoten *ausgezeichneten Knoten ist.*

   *Wir spezifizieren das Verhalten der Methoden nun dadurch, dass wir eine einfache* Referenz-Implementierung *des ADT PrioQueue angeben und dann fordern, dass sich eine Implementierung des ADT PrioQueue genauso verhält wie unsere Referenz-Implementierung. Bei unserer Referenz-Implementierung stellen wir eine Prioritäts-Warteschlange durch eine Menge von Paaren von Prioritäten und Elementen dar. Für solche Mengen definieren wir unserer Methoden wie folgt.*

   (a) *prioQueue*() $= \{\}$,
   *der Konstruktor erzeugt also eine leere Prioritäts-Warteschlange, die als leere Menge dargestellt wird.*

   (b) $Q$.*insert*$(p, v) = Q \cup \{\langle p, v\rangle\}$,
   *Um ein Element* $v$ *mit einer Priorität* $p$ *in die Prioritäts-Warteschlange* $Q$ *einzufügen, reicht es aus, das Paar* $\langle p, v\rangle$ *zu der Menge* $Q$ *hinzuzufügen.*

   (c) *Wenn* $Q$ *leer ist, dann ist* $Q$.*top*() *undefiniert:*
   $$Q = \{\} \;\to\; Q.\mathit{top}() = \Omega.$$

   (d) *Wenn* $Q$ *nicht leer ist, wenn es also ein Paar* $\langle p_1, v_1\rangle$ *in* $Q$ *gibt, dann liefert* $Q$.*top*() *ein Paar* $\langle p_2, v\rangle$ *aus der Menge* $Q$, *so dass die Priorität* $p_2$ *minimal wird. Dann gilt also für alle* $\langle p_1, v_1\rangle \in Q$, *dass* $p_2 \leq p_1$ *ist. Formal können wir schreiben:*
   $$\langle p_1, v_1\rangle \in Q \;\wedge\; Q.\mathit{top}() = \langle p_2, v_2\rangle \;\to\; p_2 \leq p_1 \;\wedge\; \langle p_2, v_2\rangle \in Q.$$

   (e) *Falls* $Q$ *leer ist, dann ändert* remove() *nichts daran:*
   $$Q = \{\} \to Q.\mathit{remove}() = Q.$$

   (f) *Sonst entfernt* $Q$.*remove*() *ein Paar mit der höchsten Priorität:*
   $$Q \neq \{\} \to Q.\mathit{remove}() = Q \backslash \big\{Q.\mathit{top}()\big\}.$$

   (g) *Die Methode change() ändert die Priorität des Paares, dessen Element als Argument übergeben wird:*
   $$Q.\mathit{change}(p_1, v_1) = \big\{\langle p_2, v_2\rangle \in Q \mid v_2 \neq v_1\big\} \cup \big\{\langle p_1, v_1\rangle\big\}.$$

Wir können den abstrakten Daten-Typ *PrioQueue* dadurch implementieren, dass wir eine Prioritäts-Warteschlange durch eine Liste realisieren, in der die Elemente aufsteigend geordnet sind. Die einzelnen Operationen werden dann wie folgt implementiert:

1. *prioQueue*() erzeugt eine leere Liste.
2. $Q.insert(d)$ kann durch die Prozedur `insert` implementiert werden, die wir beim "*Sortieren durch Einfügen*" entwickelt haben.
3. $Q.top()$ gibt das erste Element der Liste zurück.
4. $Q.remove()$ entfernt das erste Element der Liste.
5. $Q.change(p, v)$ geht alle Einträge der Liste durch. Falls dabei ein Eintrag mit dem Element $v$ gefunden wird, so wird das zugehörige Paar aus der Liste gelöscht. Anschließend wird das Paar $\langle p, v\rangle$ neu in die Liste eingefügt.

Bei dieser Implementierung ist die Komplexität der Operationen *insert*() und *change*() linear in der Anzahl $n$ der Elemente der Prioritäts-Warteschlange. Alle anderen Operationen sind konstant. Wir werden jetzt eine andere Implementierung vorstellen, bei der die Komplexität von *insert*() und *change*() den Wert $\mathcal{O}\big(\log(n)\big)$ hat. Dazu müssen wir eine neue Daten-Struktur einführen: *Heaps*.

## 8.2 Die Daten-Struktur *Heap*

Wir definieren die Menge *Heap*[1] induktiv als Teilmenge der Menge $\mathcal{B}$ der binären Bäume. Dazu definieren wir zunächst für eine Priorität $p_1 \in \mathit{Priority}$ und einen binären Baum $b \in \mathcal{B}$ die Relation $p_1 \leq b$ durch Induktion über $b$. Die Intention ist dabei, dass $p_1 \leq b$ genau dann gilt, wenn für jede Priorität $p_2$, die in $b$ auftritt, $p_1 \leq p_2$ gilt. Die formale Definition ist wie folgt:

1. $p_1 \leq \mathit{Nil}$,

   denn in dem leeren Baum treten überhaupt keine Prioritäten auf.

2. $p_1 \leq \mathit{Node}(p_2, v, l, r) \stackrel{\text{def}}{\longleftrightarrow} p_1 \leq p_2 \ \wedge\ p_1 \leq l \ \wedge\ p_1 \leq r$,

   denn $p_1$ ist genau dann kleiner-gleich als alle Prioritäten, die in dem Baum $\mathit{Node}(p_2, v, l, r)$ auftreten, wenn $p_1 \leq p_2$ gilt und wenn zusätzlich $p_1$ kleiner-gleich als alle Prioritäten ist, die in $l$ oder $r$ auftreten.

Als nächstes definieren wir eine Funktion

$$\mathit{count} : \mathcal{B} \to \mathbb{N},$$

die für einen binären Baum die Anzahl der Knoten berechnet. Die Definition erfolgt durch Induktion:

1. $\mathit{Nil}.\mathit{count}() = 0$.
2. $\mathit{Node}(p, v, l, r).\mathit{count}() = 1 + l.\mathit{count}() + r.\mathit{count}()$.

Mit diesen Vorbereitungen können wir nun die Menge *Heap* induktiv definieren:

1. $\mathit{Nil} \in \mathit{Heap}$.
2. $\mathit{Node}(p, v, l, r) \in \mathit{Heap}$ g.d.w. folgendes gilt:

   (a) $p \leq l \ \wedge\ p \leq r$,

   Die Priorität an der Wurzel ist also kleiner-gleich als alle anderen Prioritäten. Diese Bedingung bezeichnen wir auch als die *Heap-Bedingung*.

[1] Der Begriff *Heap* wird in der Informatik für zwei unterschiedliche Dinge verwendet: Zum einen wird die in diesem Abschnitt beschriebene Daten-Struktur als *Heap* bezeichnet, zum anderen wird der Teil des Speichers, in dem dynamisch erzeugte Objekte abgelegt werden, als *Heap* bezeichnet.

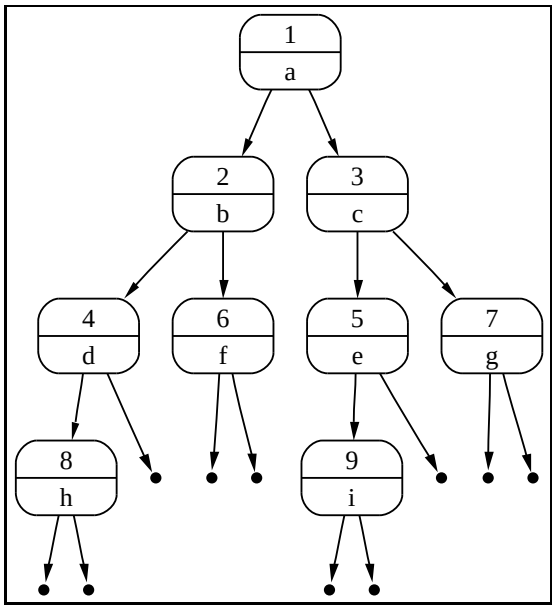


Abbildung 8.1: Ein Heap

(b) $| \, l.count() - r.count() \, | \leq 1$,
Die Zahl der Elemente im linken Teilbaum ist also höchstens 1 größer oder kleiner als die Zahl der Elemente im rechten Teilbaum. Diese Bedingung bezeichen wir als die *Balancierungs-Bedingung*. Sie ist ganz ähnlich zu der Balancierungs-Bedingung bei AVL-Bäumen, nur dass es dort die Höhe der Bäume ist, die verglichen wird, während wir hier die Zahl der im Baum gespeicherten Elemente vergleichen.

(c) $l \in \mathit{Heap} \ \wedge \ r \in \mathit{Heap}$.

Aus der *Heap-Bedingung* folgt, dass ein nicht-leerer Heap die Eigenschaft hat, dass das Element, welches an der Wurzel steht, immer die höchste Priorität hat. Abbildung 8.1 auf Seite 156 zeigt einen einfachen Heap. In den Knoten steht im oberen Teil die Prioritäten (in der Abbildung sind das natürliche Zahlen) und darunter stehen die Elemente (in der Abbildung sind dies Buchstaben).

Da Heaps binäre Bäume sind, können wir Sie ganz ähnlich wie geordnete binäre Bäume implementieren. Wir stellen zunächst Gleichungen auf, die die Implementierung der verschiedenen Methoden beschreiben. Wir beginnen mit der Methode *top*. Es gilt:

1. $\mathit{Nil}.\mathit{top}() = \Omega$.

2. $\mathit{Node}(p, v, l, r).\mathit{top}() = \langle p, v \rangle$,
denn aufgrund der Heap-Bedingung wird das Element mit der höchsten Priorität an der Wurzel gespeichert.

Die Methoden *insert* müssen wir nun so implementieren, dass sowohl die Balancierungs-Bedingung als auch die Heap-Bedingung erhalten bleiben.

1. $\mathit{Nil}.\mathit{insert}(p, v) = \mathit{Node}(p, v, \mathit{Nil}, \mathit{Nil})$.

2. $p_{\mathrm{top}} \leq p \ \wedge \ l.count() \leq r.count() \ \rightarrow$
$$\mathit{Node}(p_{\mathrm{top}}, v_{\mathrm{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p_{\mathrm{top}}, v_{\mathrm{top}}, l.\mathit{insert}(p, v), r\big).$$
Falls das einzufügende Paar eine geringere oder die selbe Priorität hat wie das Paar, welches sich an der Wurzel befindet, und falls zusätzlich die Zahl der Paare im linken Teilbaum kleiner-gleich der Zahl der Paare im rechten Teilbaum ist, dann fügen wir das Paar im linken Teilbaum ein.

3. $p_{\text{top}} \leq p \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
$$\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p_{\text{top}}, v_{\text{top}}, l, r.\mathit{insert}(p, v)\big).$$
Falls das einzufügende Paar eine geringere oder die selbe Priorität hat als das Paar an der Wurzel und falls zusätzlich die Zahl der Paare im linken Teilbaum größer als die Zahl der Paare im rechten Teilbaum ist, dann fügen wir das Paar im rechten Teilbaum ein.

4. $p_{\text{top}} > p \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
$$\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p, v, l.\mathit{insert}(p_{\text{top}}, v_{\text{top}}), r\big).$$
Falls das einzufügende Paar eine höhere Priorität hat als das Paar an der Wurzel, dann müssen wir das neu einzufügende Paar an der Wurzel positionieren. Das Paar, das dort vorher steht, fügen wir in den linken Teilbaum ein, falls die Zahl der Paare im linken Teilbaum kleiner-gleich der Zahl der Paare im rechten Teilbaum ist.

5. $p_{\text{top}} > p \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
$$\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p, v, l, r.\mathit{insert}(p_{\text{top}}, v_{\text{top}})\big).$$
Falls wir das einzufügende Paar an der Wurzel positionieren müssen und die Zahl der Paare im linken Teilbaum größer als die Zahl der Paare im rechten Teilbaum ist, dann müssen wir das Paar, das vorher an der Wurzel stand, im rechten Teilbaum einfügen.

Als nächstes beschreiben wir die Implementierung der Methode *remove*.

1. $\mathit{Nil}.\mathit{remove}() = \mathit{Nil}$,
denn aus dem leeren Heap ist nichts mehr zu entfernen.

2. $\mathit{Node}(p, v, \mathit{Nil}, r).\mathit{remove}() = r$,

3. $\mathit{Node}(p, v, l, \mathit{Nil}).\mathit{remove}() = l$,
denn wir entfernen immer das Paar mit der höchsten Priorität und das ist an der Wurzel. Wenn einer der beiden Teilbäume leer ist, können wir einfach den anderen zurück geben.

   Jetzt betrachten wir die Fälle, wo keiner der beiden Teilbäume leer ist. Dann muss entweder das Paar an der Wurzel des linken Teilbaums oder das Paar an der Wurzel des rechten Teilbaums an die Wurzel aufrücken. Welches dieser beiden Paare wir nehmen, hängt davon ab, welches der Paare die höhere Priorität hat.

4. $p_1 \leq p_2 \ \wedge \ l = \mathit{Node}(p_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{Node}(p_2, v_2, l_2, r_2) \ \rightarrow$
$$\mathit{Node}(p, v, l, r).\mathit{remove}() = \mathit{Node}(p_1, v_1, l.\mathit{remove}(), r),$$
denn wenn das Paar an der Wurzel des linken Teilbaums eine höhere Priorität hat als das Paar an der Wurzel des rechten Teilbaums, dann rückt dieses Paar an die Wurzel auf und muss folglich aus dem linken Teilbaum gelöscht werden.

5. $p_1 > p_2 \ \wedge \ l = \mathit{Node}(p_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{Node}(p_2, v_2, l_2, r_2) \rightarrow$
$$\mathit{Node}(p, v, l, r).\mathit{remove}() = \mathit{Node}(p_2, v_2, l, r.\mathit{remove}()),$$
denn wenn das Paar an der Wurzel des rechten Teilbaums eine höhere Priorität hat als das Paar an der Wurzel des linken Teilbaums, dann rückt dieses Paar an die Wurzel auf und muss folglich aus dem rechten Teilbaum gelöscht werden.

An dieser Stelle wird der aufmerksame Leser bemerken, dass die obige Implementierung der Methode *remove*() die Balancierungs-Bedingung verletzt. Es ist nicht schwierig, die Implementierung so abzuändern, dass die Balancierungs-Bedingung erhalten bleibt. Es zeigt sich jedoch, dass die Balancierungs-Bedingung nur beim Aufbau eines Heaps mittels *insert*() wichtig ist, denn dort garantiert sie, dass die Höhe des Baums in logarithmischer Weise von der Zahl seiner Knoten abhängt. Beim Löschen wird die Höhe des Baums sowieso nur kleiner, also brauchen wir uns da keine Sorgen machen.

### 8.2.1 Implementierung der Methode *change*

Als letztes beschreiben wir, wie die Methode *change*() effizient implementiert werden kann. Wir setzen voraus, dass bei einem Aufruf der Form

$$h.change(q, p, v)$$

die Priorität des Elements $v$ vergößert wird. Ist $p = Node(p', v, l, r)$ der Knoten, in dem das Element $v$ gespeichert ist, dann gilt also $p < p'$. Um die Priorität von $v$ abzuändern, müssen wir zunächst den Knoten $p$ finden. Eine Möglichkeit um diesen Knoten zu finden besteht darin, dass wir einfach alle Knoten des Heaps durchsuchen und das dort gespeicherte Element mit $v$ vergleichen. Wenn der Heap aus $n$ Knoten besteht, dann brauchen wir dazu insgesamt $n$ Vergleiche. Damit würde die Implementierung der Methode *change*() eine Komplexität $\mathcal{O}(n)$ haben. Es geht aber schneller. Die Idee ist, dass wir in einer Abbildung die Zuordnung der Knoten zu den Elementen speichern. Implementieren wir diese Abbildung beispielsweise durch einen Rot-Schwarz-Baum, so hat die Suche die Komplexität $\mathcal{O}\big(\ln(n)\big)$. Damit eine eindeutige Zuordnung von Elementen zu Knoten überhaupt möglich ist, gehen wir davon aus, dass jedes Element höchstens einmal in einem Heap auftritt. Die Abbildung realisiert dann die Funktion

$$nodeMap : Value \to Node,$$

so, dass die Invariante

$$nodeMap(v_1) = Node(k, v_2, l, r) \to v_1 = v_2$$

gilt. Mit andern Worten: Der Aufruf $nodeMap(v)$ gibt den Knoten zurück, in dem das Element $v$ gespeichert ist.

Wenn wir nun den Knoten $n = Node(p', v, l, r)$ gefunden haben, in dem das Element $v$ gespeichert ist, dann reicht es nicht aus, wenn wir in dem Knoten $n$ einfach die Priorität $p'$ durch $p$ ersetzen, denn es könnte sein, dass dann die Heap-Bedingung verletzt wird und der Schlüssel, der in dem Knoten $n$ gespeichert ist, eine höhere Priorität hat als der Vater-Knoten dieses Knotens. In diesem Fall müssen wir das Paar, dass in diesem Knoten gespeichert ist, mit dem Paar, das in dem Vater-Knoten gespeichert ist, vertauschen. Anschließend könnte es sein, dass für den Vater-Knoten und dessen Vater-Knoten die Heap-Bedingung verletzt ist, so dass wir nun rekursiv den Vater-Knoten weiter untersuchen müssen. Das Verfahren läßt sich nicht ohne weiteres durch rekursive Gleichungen beschreiben, denn wenn wir einen binären Knoten in der Form

$$Node(p, v, l, r)$$

darstellen, haben wir keine Informationen über den Vaterknoten. Wir führen daher zunächst ein Paar Hilfsfunktionen ein.

1. $parent : Node \to Node \cup \{\Omega\}$

   Für jeden Knoten $n$ gibt der Aufruf $parent(n)$ den Vaterknoten zurück. Falls zu dem Knoten $n$ kein Vaterknoten existiert, wird statt dessen $\Omega$ zurück geliefert.

2. $nodeMap : Value \to Node \cup \{\Omega\}$

   Für ein Element $v$ liefert der Aufruf $nodeMap(v)$ den Knoten, in dem das Element $v$ gespeichert ist.

3. $priority : Node \to Priority$

   Für einen Knoten $n$ liefert der Aufruf *priority*() den in dem Knoten gespeicherten Schlüssel zurück.

4. $value : Node \to Value$

   Für einen Knoten $n$ liefert der Aufruf $n.value()$ das in dem Knoten gespeicherte Element zurück.

Damit können wir nun eine Methode *upheap*() entwickeln, so dass der Aufruf $n$.*upheap*() die Heap-Bedingung an dem Knoten $n$ wiederherstellt, falls diese dadurch verletzt wurde, dass die an dem Knoten gespeicherte Priorität erhöht wurde. Abbildungen 8.2 zeigt den Pseudo-Code, den wir jetzt im Detail diskutieren.

```
1   upheap := procedure(n) {
2       p1 := priority(n);
3       v1 := value(n);
4       p  := parent(n);
5       if (p = Ω) {
6           return;
7       }
8       p2 := priority(p);
9       v2 := value(p);
10      if (p1 < p2) {
11          priority(n) := k2;
12          value(n)    := v2;
13          priority(p) := k1;
14          value(p)    := v1;
15          nodeMap(v2) := n;
16          nodeMap(v1) := p;
17          upheap(p);
18      }
19  };
```

Abbildung 8.2: Pseudo-Code zur Implementierung der Methode *upheap*()

1. Zunächst bezeichnen wir die Priorität, die an dem Knoten $n$ gespeichert ist, mit $p_1$ und das zugehörige Element mit $v_1$.

2. Der Vaterknoten von $n$ wird mit $p$ bezeichnet und die Priorität, die dort gespeichert ist, wird mit $p_2$, das zugehörige Element mit $v_2$ bezeichnet.

   Falls der Knoten $n$ bereits der Wurzel-Knoten ist, so exitiert kein Vaterknoten und damit kann die Heap-Bedingung auch nicht verletzt sein, so dass die Methode *upheap*() beendet werden kann.

3. Hat nun das an dem Knoten $n$ gespeicherte Element eine höhere Priorität als das an dem Vaterknoten $p$ gespeicherte Element, so werden die Elemente und die Prioritäten, die an den Knoten $n$ und $p$ gespeichert sind, vertauscht.

   Zusätzlich achten wir darauf, dass die in der Tabelle *nodeMap* hinterlegte Zuordung von Elementen zu Knoten korrekt bleibt.

4. Schließlich müssen wir die Methode *upheap* rekursiv für den Vaterknoten aufrufen.

## 8.3 Implementierung in SetlX

Bei der Implementierung der Methode `change`$(p, v)$ benötigen wir eine Möglichkeit, auf den Vater-Knoten eines gegebenen Knotens zugreifen zu können. Das schließt eine Implementierung aus, bei der die Knoten durch Terme dargestellt werden, denn Terme können keine zyklischen Strukturen repräsentieren. Wir repräsentieren einen Heap $h$ statt dessen durch ein Feld von Tripeln der Form

$$[p, v, c].$$

Ein solches Tripel repräsentiert einen Knoten

$$Node(p, v, l, r)$$

an dem das Element $v$ mit der Priorität $p$ gespeichert ist. Die Zahl $c$ gibt an, wieviele Knoten sich in dem Teilbaum befinden, an dessen Wurzel sich das Element $v$ findet, es gilt also $Node(p, v, l, r).\texttt{count}() = c$. Die Baum-Struktur stellen wir in dem Feld von Tripeln wie folgt dar:

1. Falls der Knoten $\texttt{Node}(p, v, l, r)$ in dem Heap $h$ an der Position $i$ gespeichert ist, falls also

   $$h[i] = [p, v, c]$$

   gilt, dann wird die Wurzel des linken Teilbaums $l$ an der Position

   $$2 \cdot i$$

   gespeichert, während die Wurzel des rechten Teilbaums an der Position

   $$2 \cdot i + 1$$

   gespeichert wird.

2. Daraus folgt, dass der Vater-Knoten des Knotens, der an der Position $i$ gespeichert ist, an der Position $i\backslash 2$ gespeichert wird, wobei der Operator "\" hier die ganzzahlige Division bezeichnet. Diese Darstellung funktioniert, denn es gilt sowohl

   $$(2 \cdot i)\backslash 2 = i \quad \text{als auch} \quad (2 \cdot i + 1)\backslash 2 = i.$$

Abbildung 8.3 auf Seite 161 zeigt die Implementierung der Funktionen `prioQueue`(), `top`() und `insert`(), die wir jetzt im Detail diskutieren.

1. Zunächst deklarieren wir in Zeile 1 die Variable `gValueMap`, welche die Zuordung von Elementen zu den Knoten beschreibt. Technisch ist der Wert von `gValueMap` eine funktionale binäre Relation. Die ersten Komponenten der dort gespeicherten Paare sind die Elemente, die zweite Komponente gibt dann den Index an, an dem dieses Element in dem Feld gespeichert ist.

   Am Anfang ist diese Relation natürlich noch leer, weil auch der Heap leer ist. Daher wird `gValueMap` in Zeile 2 mit der leeren Menge initialisiert.

2. Die Funktion `prioQueue`() erzeugt einen leeren Heap und gibt daher in Zeile 5 eine leere Liste zurück, die als leeres Feld interpretiert wird.

3. Die Funktion $\texttt{top}(h)$ gibt den Knoten zurück, der an der Wurzel des Heaps gespeichert ist. Da die Wurzel an der Position 1 gespeichert ist, wird also `h`[`1`] zurück gegeben.

4. Der Aufruf $\texttt{insert}(h, p, v)$ fügt das Element $v$ mit der Priorität $p$ in dem Heap $h$ ein. Da der Heap $h$ dabei verändert wird, ist der Parameter $h$ mit dem Schlüsselwort "`rw`" als *Read-Write-Parameter* deklariert worden.

   Die Implementierung der Funktion $\texttt{insert}(h, p, v)$ benutzt die Hilfsfunktion

   $$\texttt{insertPosition}(h, 1, p, v),$$

   die das Element $v$ an der Wurzel des Heaps $h$ einfügt. Da die Wurzel an der Position 1 gespeichert ist, hat der zweite Parameter des Aufrufs von `insertPosition`() den Wert 1.

5. Die Funktion $\texttt{insertPosition}(h, i, p, v)$ versucht das Element $v$ in dem Teilbaum, dessen Wurzel an der Position $i$ steht, einzufügen.

   (a) Falls dieser Teilbaum leer ist, gilt $\texttt{h}[\texttt{i}] = \Omega$. In diesem Fall wird in Zeile 15 durch die Zuweisung ein neuer Knoten erzeugt, in dem das Element $v$ mit der Priorität $p$ gespeichert ist. Zusätzlich müssen wir sicherstellen, dass die Zuordung des Elements $v$ zu diesem Knoten in der Tabelle `gValueMap` abgespeichert wird.

```
1  var gValueMap;  // assign values to indices
2  gValueMap := {};
3
4  prioQueue := procedure() {
5      return [];
6  };
7  top := procedure(h) {
8      return h[1];
9  };
10 insert := procedure(rw h, p, v) {
11     insertPosition(h, 1, p, v);
12 };
13 insertPosition := procedure(rw h, i, p, v) {
14     if (h[i] == om) {
15         h[i] := [p, v, 1];
16         gValueMap[v] := i;
17         return;
18     }
19     [pt, vt, ct] := h[i];
20     cl := count(h, left (i));
21     cr := count(h, right(i));
22     if (pt <= p) {
23         if (cl <= cr) {
24             insertPosition(h, left(i), p, v);
25         } else {
26             insertPosition(h, right(i), p, v);
27         }
28         h[i] := [pt, vt, ct + 1];
29     } else {
30         if (cl <= cr) {
31             insertPosition(h, left(i), pt, vt);
32         } else {
33             insertPosition(h, right(i), pt, vt);
34         }
35         h[i] := [p, v, ct + 1];
36         gValueMap[v] := i;
37     }
38 };
```

Abbildung 8.3: Implementierung von `insert`().

(b) Andernfalls ist der Teilbaum an der Position $i$ in dem Heap nicht leer und wir müssen zunächst einmal schauen, was genau an der Wurzel dieses Teilbaums gespeichert ist. Dies erledigen wir in Zeile 19. Außerdem müssen wir wissen, wieviele Knoten in dem linken und dem rechten Teilbaum gespeichert sind. Diese Anzahlen werden in den Zeilen 20 und 21 berechnet.

(c) Falls die Priorität `pt` des Elements an der Wurzel höher ist als die Priorität des Elements, das wir einfügen wollen, dann müssen wir das Elements $v$ entweder im linken oder rechten Teilbaum einfügen, je nachdem, wo mehr Platz ist. Anschließend dürfen wir nicht vergessen, für die Wurzel des an der Position $i$ gespeicherten Teilbaum die Anzahl der Knoten zu inkrementieren, denn der Baum enthält ja jetzt einen zusätzlichen

Knoten. Dies wird durch die Zuweisung in Zeile 28 sichergestellt.

(d) Falls die Priorität `pt` des Elements `vt` an der Wurzel geringer ist als die Priorität `p` des neu einzufügenden Elements `v`, dann wird das Element `v` in Zeile 35 an der Wurzel des Teilbaums eingefügt. Gleichzeitig wird das Element `vt`, das sich ursprünglich an der Position $i$ befunden hat, im linken oder rechten Teilbaum eingefügt. Außerdem müssen wir in diesem Fall noch die Zuordnung des Elements `v` zu der Position $i$ in der Tabelle `gValueMap` eintragen.

```
39  remove := procedure(rw h) {
40      removePosition(h, 1);
41  };
42  removePosition := procedure(rw h, i) {
43      t := h[i];
44      if (t == om) {
45          return;
46      }
47      [pt, vt, ct] := t;
48      pl := priority(h, left(i) );
49      pr := priority(h, right(i));
50      gValueMap[vt] := om;
51      if (pl != om && (pr == om || pl < pr)) {
52          [pl, vl, _] := h[left(i)];
53          h[i] := [pl, vl, ct - 1];
54          gValueMap[vl] := i;
55          removePosition(h, left(i));
56      }
57      if (pr != om && (pl == om || pr <= pl)) {
58          [pr, vr, _] := h[right(i)];
59          h[i] := [pr, vr, ct - 1];
60          gValueMap[vr] := i;
61          removePosition(h, right(i));
62      }
63      if (pl == om && pr == om) {
64          h[i] := om;
65      }
66  };
```

Abbildung 8.4: Implementierung von `remove`().

Abbildung 8.4 auf Seite 162 zeigt die Implementierung der Funktion `remove`($h$), die das Element mit der höchsten Priorität aus dem Heap $h$ entfernt. Auch diese Funktion wird über eine Hilfsfunktion realsiert. Die Funktion `removePosition`($h, i$) entfernt das Element, der an der Wurzel des Teilbaums abgelegt ist, dessen Wurzel-Knoten an der Position $i$ in dem Teilbaum gespeichert ist.

1. Falls dieser Teilbaum leer ist, kann kein Element entfernt werden. Dies wird in Zeile 44 getestet.

2. Andernfalls schauen wir in Zeile 47 welches Element an der Wurzel des Teilbaums gespeichert ist. An Stelle dieses Elements muss nun entweder das Element `vl`, der an der Wurzel des linken Teilbaums steht oder das Element `vr`, der an der Wurzel des rechten Teilbaums steht, gespeichert werden. Die Prioritäten dieser Elemente sind `pl` bzw. `pr`. Um die Heap-Bedingung

zu erhalten, müssen wir das Element mit der höheren Priorität wählen. Anschließend müssen wir das Element, das nach oben aufgerückt ist, aus seiner alten Position entfernen, was durch die rekursiven Aufrufe der Funktion `removePosition`() in den Zeilen 55 und 61 sichergestellt wird.

Diese Überlegungen werden dadurch kompliziert, dass sowohl der linke als auch der rechte Teilbaum eventuell leer sein kann.

3. Falls beide Teilbäume leer sind, können wir die Position $i$ in Zeile 64 aus dem Feld $h$ löschen.

```
67  change := procedure(rw h, v, p) {
68      i := gValueMap[v];
69      [p0, v0, c] := h[i];
70      assert(p <= p0, "new priority must be higher");
71      assert(v == v0, "gValueMap corrupted");
72      h[i] := [p, v, c];
73      upHeap(h, i);
74  };
75  upHeap := procedure(rw h, i) {
76      if (i == 1) {
77          return;
78      }
79      up := parent(i);
80      [p1, v1, _] := h[i];
81      [p2, v2, _] := h[up];
82      if (p1 < p2) {
83          exchange(h, i, up);
84          upHeap(h, up);
85      }
86  };
87  exchange := procedure(rw h, i, j) {
88      [pi, vi, ci] := h[i];
89      [pj, vj, cj] := h[j];
90      h[i] := [pj, vj, ci];
91      h[j] := [pi, vi, cj];
92      gValueMap[vi] := j;
93      gValueMap[vj] := i;
94  };
```

Abbildung 8.5: Implementierung der Funktion `change`().

Als nächstes diskutieren wir die Implementierung der Funktion `change`$(h, v, p)$, welche die Aufgabe hat, die Priorität des Elements $v$ so zu erhöhen, dass die neue Priorität den Wert $p$ hat.

1. Zunächst schlagen wir in Zeile 68 die Position $i$ nach, an der das Objekt $v$ gespeichert ist.

2. Dann holen wir uns in Zeile 69 den an der Position $i$ gespeicherten Knoten und überprüfen, ob die neue Priorität tatsächlich höher ist als die alte Priorität.

3. In Zeile 72 überschreiben wir die alte Priorität mit der neuen Priorität.

4. Eventuell ist nun die Heap-Bedingung verletzt, denn es könnte nun sein, dass der Vater-Knoten eine geringere Priorität hat als der Knoten, an dem wir gerade die Priorität verringert haben. Daher rufen wir nun die Prozedur `upHeap`$(h, i)$ auf, deren Aufgabe es ist,

gegebenenfalls den an der Position $i$ gespeicherten Knoten mit seinem Vater-Knoten zu vertauschen.

5. Die Prozedur `upHeap`$(h, i)$ prüft als erstes, ob der Knoten an der Position $i$ überhaupt einen Vater-Knoten hat, denn nur dann ist etwas zu tun.

6. In diesem Fall wird in Zeile 79 der Index des Vater-Knotens berechnet. Anschließend wird in Zeile 82 die Priorität des Knotens mit dem Index $i$ mit der Priorität des Vater-Knotens verglichen. Falls der Knoten an der Position $i$ eine höhere Priorität hat, wird er durch den Aufruf der Prozedur `exchange`() mit seinem Vater-Knoten vertauscht.

7. Anschließend muss für den Vater-Knoten rekursiv die Prozedur `upHeap`() aufgerufen werden, denn es könnte ja sein, dass jetzt dessen Vater-Knoten eine geringere Priorität hat als der Knoten, der gerade die Position `up` übernommen hat.

```
95   count := procedure(h, i) {
96       t := h[i];
97       if (t != om) {
98           [_, _, c] := t;
99       } else {
100          c := 0;
101      }
102      return c;
103  };
104  priority := procedure(h, i) {
105      t := h[i];
106      if (t != om) {
107          [p, _, _] := t;
108      }
109      return p;
110  };
111  parent := i |-> i \ 2;
112  left   := i |-> 2 * i;
113  right  := i |-> 2 * i + 1;
```

Abbildung 8.6: Implementierung der Hilfsfunktionen.

Abbildung 8.6 zeigt die Implementierung verschiedener Hilffunktionen.

1. Die Funktion `count`$(h, i)$ berechnet die Anzahl der Elemente, die in dem Teilbaum gespeichert sind, dessen Wurzel an der Position $i$ in dem Heap $h$ gespeichert ist. Dabei nutzen wir aus, dass diese Anzahl bereits in den Tripeln, aus denen das Feld $h$ aufgebaut ist, gespeichert ist.

   Der Aufruf `count`$(h, i)$ funktioniert auch dann, wenn der an der Position $i$ gespeicherte Teilbaum leer ist.

2. In analoger Weise berechnet die Funktion `priority`$(h, i)$ die Priorität des Elements, das an der Position $i$ in dem Heap $h$ gespeichert ist.

3. Die Funktion `parent`$(i)$ berechnet für einen Index $i$ den Index des Vater-Knotens.

4. Die Funktion `left`$(i)$ berechnet für einen Index $i$ den Index des linken Teilbaums.

5. Die Funktion `parent`$(i)$ berechnet für einen Index $i$ den Index des rechten Teilbaums.

# Kapitel 9

# Daten-Kompression

In diesem Kapitel untersuchen wir die Frage, wie wir einen gegebenen String $s$ möglichst platzsparend abspeichern können. Wir gehen davon aus, dass der String $s$ aus Buchstaben besteht, die Elemente einer Menge $\Sigma$ sind. Die Menge $\Sigma$ bezeichnen wir als unser *Alphabet*. Wenn das Alphabet aus $n$ verschiedenen Zeichen besteht und wir alle Buchstaben mit der selben Länge von $b$ Bits kodieren wollen, dann muss für diese Zahl von Bits offenbar

$$n \leq 2^b$$

gelten, woraus

$$b = \mathit{ceil}\big(\log_2(n)\big)$$

folgt. Hier bezeichnet $\mathit{ceil}(x)$ die *Ceiling-Funktion*. Diese Funktion rundet eine gegebene reelle Zahl immer auf, es gilt also

$$\mathit{ceil}(x) = \min\{k \in \mathbb{N} \mid x \leq k\}.$$

Besteht der String $s$ aus $m$ Buchstaben, so werden zur Kodierung des Strings insgesamt $m \cdot b$ Bits gebraucht. Lassen wir die Forderung, dass alle Buchstaben mit der selben Anzahl von Bits kodiert werden, fallen, dann ist es unter Umständen möglich, den String $s$ mit weniger Bits zu kodieren. Die zentrale Idee ist dabei, dass Buchstaben, die sehr häufig auftreten, mit möglichst wenig Bits kodiert werden, während Buchstaben, die sehr selten auftreten, mit einer größeren Anzahl Bits kodiert werden. Zur Verdeutlichung betrachten wir folgendes Beispiel: Unser Alphabet $\Sigma$ bestehe nur aus vier Buchstaben,

$$\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}.$$

In dem zu speichernden String $s$ trete der Buchstabe `a` insgesamt 990 mal auf, der Buchstabe `b` trete 8 mal auf und die Buchstaben `c` und `d` treten jeweil 1 mal auf. Dann besteht der String $s$ aus insgesamt 1 000 Buchstaben. Wenn wir jeden Buchstaben mit $2 = \log_2(4)$ Bits kodieren, dann werden also insgesamt 2 000 Bits benötigt um den String $s$ abzuspeichern. Wir können den String aber auch mit weniger Bits abspeichern, wenn wir die einzelnen Buchstaben mit Bitfolgen unterschiedlicher Länge kodieren. In unserem konkreten Beispiel wollen wir versuchen den Buchstaben `a`, der mit Abstand am häufigsten vorkommt, mit einem einzigen Bit zu kodieren. Bei den Buchstaben `c` und `d`, die nur sehr selten auftreten, ist es kein Problem auch mehr Bits zu verwenden. Tabelle 9.1 zeigt eine Kodierung, die von dieser Idee ausgeht.

| Buchstabe | a | b | c | d |
|---|---|---|---|---|
| Kodierung | 0 | 10 | 110 | 111 |

Tabelle 9.1: Kodierung der Buchstaben mit variabler Länge.

Um zu verstehen, wie diese Kodierung funktioniert, stellen wir sie in Abbildung 9.1 als Baum dar. Die inneren Knoten dieses Baums enthalten keine Attribute und werden als leere Kreis dargestellt. Die Blätter des Baums sind mit den Buchstaben markiert. Die Kodierung eines Buchstabens ergibt sich über die Beschriftung der Kanten, die von dem Wurzel-Knoten zu dem Buchstaben führen. Beispielsweise führt von der Wurzel eine Kante direkt zu dem Blatt, das mit dem Buchstaben "`a`" markiert ist. Diese Kante ist mit dem Label "`0`" beschriftet. Also wird der Buchstabe "`a`" durch den String "`0`" kodiert. Um ein weiteres Beispiel zu geben, betrachten wir den Buchstaben "`c`". Der Pfad, der von der Wurzel zu dem Blatt führt, das mit "`c`" markiert ist, enthält drei Kanten. Die ersten beiden Kanten sind jeweils mit "`1`" markiert, die letzte Kante ist mit "`0`" markiert. Also wird der Buchstabe "`c`" durch den String "`110`" kodiert. Kodieren wir nun unseren ursprünglichen String $s$, der aus 990 `a`'s, 8 `b`'s, einem `c` und einem `d` besteht, so benötigen wir insgesamt

$$990 \cdot 1 + 8 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 1\,012$$

Bits. Gegenüber der ursprünglichen Kodierung, die 2 000 Bits verwendet, haben wir $49,4\%$ gespart!

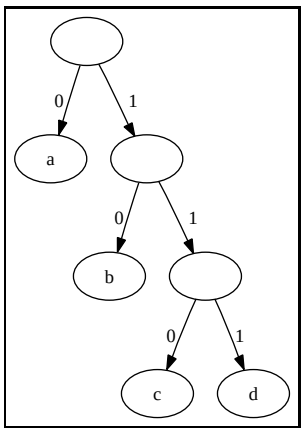


Abbildung 9.1: Baum-Darstellung der Kodierung.

Um zu sehen, wie mit Hilfe des Kodierungs-Baums ein String dekodiert werden kann, betrachten wir als Beispiel den String "`100111`". Wir beginnen mit der "`1`", die uns sagt, vom Wurzel-Knoten dem rechten Pfeil zu folgen. Die anschließende "`0`" spezifiziert dann den linken Pfeil. Jetzt sind wir bei dem mit "`b`" markierten Blatt angekommen und haben damit den ersten Buchstaben gefunden. Wir gehen wieder zur Wurzel des Baums zurück. Die folgende "`0`" führt uns zu dem Blatt, das mit "`a`" markiert ist, also haben wir den zweiten Buchstaben gefunden. Wir gehen wieder zur Wurzel zurück. Die Ziffern "`111`" führen uns nun zu dem Buchstaben "`d`". Damit haben wir insgesamt

"`100111`" $\simeq$ "`bad`".

## 9.1 Der Algorithmus von Huffman

Angenommen, wir haben einen String $s$, der aus Buchstaben eines Alphabets $\Sigma$ aufgebaut ist. Wie finden wir dann eine Kodierung für die einzelnen Buchstaben, die mit möglichst wenig Bits auskommt? Der Algorithmus von Huffman gibt eine Antwort auf diese Frage. Um diesen Algorithmus präsentieren zu können, definieren wir die Menge $\mathcal{K}$ der *Kodierungs-Bäume* induktiv.

1. $\mathit{Leaf}(c, f) \in \mathcal{K}$ falls $c \in \Sigma$ und $f \in \mathbb{N}$.

   Ausdrücke der Form $\mathit{Leaf}(c, f)$ sind die Blätter eines Kodierungs-Baums. Dabei ist $c$ ein Buchstabe aus unserem Alphabet $\Sigma$ und $f$ gibt die Häufigkeit an, mit der dieser Buchstabe in dem zu kodierenden String auftritt.

Gegenüber Abbildung 9.1 kommen hier bei den Blättern noch die Häufigkeiten hinzu. Diese benötigen wir, denn wir wollen ja später Buchstaben, die sehr häufig auftreten, mit möglichst wenig Bits kodieren.

2. $\mathit{Node}(l, r) \in \mathcal{K}$ falls $l \in \mathcal{K}$ und $r \in \mathcal{K}$.

   Ausdrücke der Form $\mathit{Node}(l, r)$ sind die inneren Knoten eines Kodierungs-Baums.

Als nächstes definieren wir eine Funktion

$$\mathit{count} : \mathcal{K} \to \mathbb{N},$$

welche die Gesamt-Häufigkeiten aller in dem Baum auftretenden Buchstaben aufsummiert.

1. Die Definition der Funktion *count* ist für Blätter trivial:

   $$\mathit{Leaf}(c, f).\mathit{count}() = f.$$

2. Die Gesamt-Häufigkeit des Knotens $\mathit{Node}(l, r)$ ergibt sich als Summe der Gesamt-Häufigkeiten von $l$ und $r$. Also gilt

   $$\mathit{Node}(l, r).\mathit{count}() = l.\mathit{count}() + r.\mathit{count}().$$

Weiter definieren wir auf Kodierungs-Bäumen die Funktion

$$\mathit{cost} : \mathcal{K} \to \mathbb{N}.$$

Die Funktion *cost* gibt an, wie viele Bits benötigt werden, um mit dem gegebenen Kodierungs-Baum eine String zu kodieren, wenn die Häufigkeiten, mit denen ein Buchstabe verwendet wird, mit den Häufigkeiten übereinstimmen, die an den Blättern des Baums notiert sind. Die Definition dieser Funktion ist induktiv:

1. $\mathit{Leaf}(c, f).\mathit{cost}() = 0$,

   denn solange nur ein einziger Buchstabe vorhanden ist, ist noch nichts zu kodieren.

2. $\mathit{Node}(l, r).\mathit{cost}() = l.\mathit{cost}() + r.\mathit{cost}() + l.\mathit{count}() + r.\mathit{count}()$.

   Wenn wir zwei Kodierungs-Bäume $l$ und $r$ zu einem neuen Kodierungs-Baum zusammenfügen, verlängern sich die Kodierungen für alle Buchstaben, die in $l$ oder $r$ auftreten, um ein Bit. Die Summe

   $$l.\mathit{count}() + r.\mathit{count}()$$

   gibt die Gesamt-Häufigkeiten aller Buchstaben an, die in dem linken und rechten Teilbaum auftreten. Da sich die Kodierung aller dieser Buchstaben durch die Bildung des Knotens $\mathit{Node}(l, r)$ gegenüber der Kodierung in $l$ und $r$ jeweils um 1 verlängert, müssen wir zu den Kosten der Teilbäume $l$ und $r$ den Term $l.\mathit{count}() + r.\mathit{count}()$ hinzuaddieren.

Wir erweitern die Funktion $\mathit{cost}()$ auf Mengen von Knoten, indem wir die Kosten einer Menge $M$ als die Summe der Kosten der Knoten von $M$ definieren:

$$\mathit{cost}(M) = \sum_{n \in M} n.\mathit{cost}().$$

Ausgangs-Punkt des von David A. Huffman (1925 – 1999) [Huf52] angegebenen Algorithmus ist eine Menge von Paaren der Form $\langle c, f \rangle$. Dabei ist $c$ ein Buchstabe und $f$ gibt die Häufigkeit an, mit der dieser Buchstabe auftritt. Im ersten Schritt werden diese Paare in die Blätter eines Kodierungs-Baums überführt. Besteht der zu kodierende String aus $n$ verschiedenen Buchstaben, so haben wir dann eine Menge von Kodierungs-Bäumen der Form

$$M = \big\{\mathit{Leaf}(c_1, f_1), \cdots, \mathit{Leaf}(c_k, f_k)\big\} \tag{9.1}$$

Es werden nun solange Knoten $a$ und $b$ aus $M$ zu einem neuen Knoten $\mathit{Node}(a, b)$ zusammen gefasst, bis die Menge $M$ nur noch einen Knoten enthält. Offenbar gibt es im Allgemeinen sehr viele Möglichkeiten, die Knoten aus der Menge zu neuen Knoten zusammen zu fassen. Das Ziel ist es die Knoten so zusammen zu fassen, dass die Kosten der Menge $M$ am Ende minimal sind. Um zu verstehen, welche Knoten wir am geschicktesten zusammenfassen können, betrachten wir, wie sich die Kosten der Menge durch das Zusammenfassen zweier Knoten ändert. Dazu betrachten wir zwei Mengen von Knoten $M_1$ und $M_2$, so dass

$$M_1 = N \cup \{a, b\} \quad \texttt{und} \quad M_2 = N \cup \{\mathit{Node}(a, b)\}$$

gilt, die Menge $M_1$ geht also aus der Menge $M_2$ dadurch hervor, dass wir die Knoten $a$ und $b$ zu einem neuen Knoten zusammen fassen und durch diesen ersetzen. Untersuchen wir, wie sich die Kosten der Menge dabei verändern, wir untersuchen also die folgende Differenz:

$$\begin{array}{ll} & \mathit{cost}\big(N \cup \{\mathit{Node}(a, b)\}\big) - \mathit{cost}\big(N \cup \{a, b\}\big) \\ = & \mathit{cost}\big(\{\mathit{Node}(a, b)\}\big) - \mathit{cost}\big(\{a, b\}\big) \\ = & \mathit{Node}(a, b).\mathit{cost}() - a.\mathit{cost}() - b.\mathit{cost}() \\ = & a.\mathit{cost}() + b.\mathit{cost}() + a.\mathit{count}() + b.\mathit{count}() - a.\mathit{cost}() - b.\mathit{cost}() \\ = & a.\mathit{count}() + b.\mathit{count}() \end{array}$$

Fassen wir die Knoten $a$ und $b$ aus der Menge $M$ zu einem neuen Knoten zusammen, so vergößern sich die Kosten der Menge um die Summe

$$a.\mathit{count}() + b.\mathit{count}().$$

Wenn wir die Kosten der Menge $M$ insgesamt möglichst klein halten wollen, dann ist es daher naheliegend, dass wir in der Menge $M$ die beiden Knoten $a$ und $b$ suchen, für die die Funktion $\mathit{count}()$ den kleinsten Wert liefert. Diese Knoten werden wir aus der Menge $M$ entfernen und durch den neuen Knoten $\mathit{Node}(a, b)$ ersetzen. Dieser Prozess wird solange iteriert, bis die Menge $M$ nur noch aus einem Knoten besteht. Dieser Knoten ist dann die Wurzel des gesuchten Kodierungs-Baums.

```
1  codingTree := procedure(m) {
2      while (#m > 1) {
3          a := first(m);
4          m -= { a };
5          b := first(m);
6          m -= { b };
7          m += { [ count(a) + count(b), Node(a, b) ] };
8      }
9      return arb(m);
10 };
11 count := p |-> p[1];
```

Abbildung 9.2: Der Algorithmus von Huffman in SetlX.

Die in Abbildung 9.2 gezeigte Funktion `codingTree`$(m)$ implementiert diesen Algorithmus.

1. Die Funktion *codingTree* wird mit einer Menge $m$ von Knoten aufgerufen, welche die Form

   $$m = \big\{\langle f_1, \mathit{Leaf}(c_1)\rangle, \cdots, \langle f_k, \mathit{Leaf}(c_k)\rangle\big\}$$

   hat. Hier bezeichnen die Variablen $c_i$ die verschiedenen Buchstaben, während die Zahl $f_i$ die Häufigkeit angibt, mit der der Buchstabe $c_i$ auftritt.

Wir haben hier die Information über die Häufigkeit an erster Stelle eines Paares gespeichert. Da SetlX diese Menge intern durch einen geordneten binären Baum abspeichert, ermöglicht uns diese Form der Darstellung einfach auf den Knoten mit der kleinsten Häufigkeit zuzugreifen, denn die Paare werden so verglichen, dass immer zunächst die erste Komponente zweier Paare zum Vergleich herangezogen werden. Nur wenn sich in der ersten Komponente kein Unterschied ergibt, wird auch die zweite Komponente verglichen. Daher finden wir das Paar mit der kleinsten ersten Komponente immer am Anfang der Menge $m$.

Durch diesen Trick haben wir uns de facto die Implementierung einer Prioritäts-Warteschlange gespart:

(a) Die in SetlX vordefinierte Funktion `first`($m$) liefert das erste Element der Menge $m$ und entspricht damit der Funktion *top*($m$) des abstrakten Daten-Typs *PrioQueue*.

(b) Anstelle von *insert*($m, p, v$) können wir einfach

$m$ `+= { [`$p$`, `$v$`] };`

schreiben um das Element $v$ mit der Priorität $p$ in die Prioritäts-Warteschlange $m$ einzufügen.

(c) Die Funktion *remove*($m$) realisieren wir durch den Aufruf

$m$ `-= { first(`$m$`) };`

denn *remove*($m$) soll ja das Element mit der höchsten Priorität aus $m$ entfernen.

Das Elegante an diesem Vorgehen ist, dass damit sämtliche Operationen des abstrakten Daten-Typs *PrioQueue* eine logarithmische Komplexität haben. Das ist zwar im Falle der Operation *top*($m$) nicht optimal, aber für die Praxis völlig ausreichend, denn in der Praxis kommt auf jeden Aufruf der Form *top*($m$) auch ein Aufruf der Form *remove*($m$) und der hat sowohl bei einer optimalen Implementierung als auch bei unserer Implementierung eine logarithmische Komplexität, die dann auch im Falle der optimalen Implementierung die gesamte Komplexität dominiert.

2. Die `while`-Schleife in Zeile 2 veringert die Anzahl der Knoten in der Menge $m$ in jedem Schritt um Eins.

   (a) Dazu werden mit Hilfe der Funktion *first*() die beiden Knoten $a$ und $b$ berechnet, für die der Wert von *count*() minimal ist. Die Funktion `count`($p$) ist in Zeile 11 definiert und liefert einfach die erste Komponente des Paares $p$, denn dort speichern wir die Häufigkeit der Buchstaben ab.

   (b) Die beiden Knoten $a$ und $b$ mit der geringsten Häufigkeit werden in Zeile 4 und 6 aus der Menge $m$ entfernt.

   (c) Anschließend wird aus den beiden Knoten $a$ und $b$ ein neuer Knoten *Node*($a, b$) gebildet. Dieser neue Knoten wird zusammen mit der Gesamthäufigkeit der Knoten $a$ und $b$ in Zeile 7 der Menge $m$ hinzugefügt.

3. Die `while`-Schleife wird beendet, wenn die Menge $m$ nur noch ein Element enthält. Dieses wird mit der Funktion `arb` extrahiert und als Ergebnis zurück gegeben.

Die Laufzeit des Huffman-Algorithmus hängt stark von der Effizienz der Funktion *first*() ab. Eine naive Implementierung würde die Knoten aus der Menge $m$ in einer geordneten Liste vorhalten. Die Knoten $n$ wären in dieser Liste nach der Größe *n.cost*() aufsteigend sortiert. Dann ist die Funktion *first*() zwar sehr effizient, aber das Einfügen des neuen Knotens, dass wir oben über den Befehl

```
m += { [ count(a) + count(b), Node(a, b) ] };
```

realisieren, würde einen Aufwand erfordern, der linear in der Anzahl der Elemente der Menge $m$

ist. Dadurch, dass wir mit einer Menge $m$ arbeiten, die in SetlX intern durch einen Rot-Schwarz-Baum dargestellt ist, erreichen wir, dass alle in der `while`-Schleife durchgeführten Operationen nur logarithmisch von der Anzahl der Buchstaben abhängen. Damit hat der Huffman-Algorithmus insgesamt die Komplexität $\mathcal{O}\big(n \cdot \ln(n)\big)$.

| Buchstabe | a | b | c | d | e |
|---|---|---|---|---|---|
| Häufigkeit | 1 | 2 | 3 | 4 | 5 |

Tabelle 9.2: Buchstaben mit Häufigkeiten.

Wir illustrieren den Huffman-Algorithmus, indem wir ihn auf die Buchstaben, die in Tabelle 9.2 zusammen mit ihren Häufigkeiten angegeben sind, anwenden.

1. Zu Beginn hat die Menge $m$ die Form

   $$m = \big\{\langle 1, \mathit{Leaf}(\mathtt{a})\rangle,\ \langle 2, \mathit{Leaf}(\mathtt{b})\rangle,\ \langle 3, \mathit{Leaf}(\mathtt{c})\rangle,\ \langle 4, \mathit{Leaf}(\mathtt{d})\rangle,\ \langle 5, \mathit{Leaf}(\mathtt{e})\rangle\big\}.$$

2. Die Häufigkeit ist hier für die Blätter mit den Buchstaben `a` und `b` minimal. Also entfernen wir diese Blätter aus der Menge und fügen statt dessen den Knoten

   $$\mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b}))$$

   in die Menge $m$ ein. Die Häufigkeit dieses Knotens ergibt sich als Summe der Häufigkeiten der Buchstaben `a` und `b`. Daher fügen wir insgesamt das Paar

   $$\langle 3, \mathit{Node}\big(\mathit{Leaf}(\mathtt{a})\rangle, \mathit{Leaf}(\mathtt{b})\big)\rangle$$

   in die Menge $m$ ein. Dann hat $m$ die Form

   $$\big\{\langle 3, \mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b}))\rangle,\ \langle 3, \mathit{Leaf}(\mathtt{c})\rangle,\ \langle 4, \mathit{Leaf}(\mathtt{d})\rangle,\ \langle 5, \mathit{Leaf}(\mathtt{e})\rangle\big\}.$$

3. Die beiden Paare mit den kleinsten Werten der Häufigkeiten in $m$ sind nun

   $$\langle 3, \mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b}))\rangle \quad \text{und} \quad \langle 3, \mathit{Leaf}(\mathtt{c})\rangle.$$

   Wir entfernen diese beiden Knoten und bilden aus diesen beiden Knoten den neuen Knoten

   $$\langle 6, \mathit{Node}(\mathit{Node}((\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})),\ \mathit{Leaf}(\mathtt{c}))\rangle,$$

   den wir der Menge $m$ hinzufügen. Dann hat $m$ die Form

   $$\Big\{\langle 4, \mathit{Leaf}(\mathtt{d})\rangle,\ \langle 5, \mathit{Leaf}(\mathtt{e})\rangle,\ \langle 6, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})),\ \mathit{Leaf}(\mathtt{c}))\Big\}.$$

4. Jetzt sind

   $$\langle 4, \mathit{Leaf}(\mathtt{d})\rangle \quad \text{und} \quad \langle 5, \mathit{Leaf}(\mathtt{e})\rangle$$

   die beiden Knoten mit dem kleinsten Werten der Häufigkeit. Wir entfernen diese Knoten und bilden den neuen Knoten

   $$\langle 9, \mathit{Node}(\mathit{Leaf}(\mathtt{d}), \mathit{Leaf}(\mathtt{e}))\rangle.$$

   Diesen fügen wir der Menge $m$ hinzu und erhalten

   $$\Big\{\langle 6, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})),\ \mathit{Leaf}(\mathtt{c}, 3))\rangle,\ \langle 9, \mathit{Node}(\mathit{Leaf}(\mathtt{d}, 4), \mathit{Leaf}(\mathtt{e}, 5))\rangle\Big\}.$$

5. Jetzt enthält die Menge $m$ nur noch zwei Knoten. Wir entfernen diese beiden Knoten und bilden daraus den neuen Knoten

   $$\mathit{Node}\Big(\mathit{Node}\Big(\mathit{Node}\big(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})\big),\ \mathit{Leaf}(\mathtt{c})\Big),\ \mathit{Node}\big(\mathit{Leaf}(\mathtt{d}), \mathit{Leaf}(\mathtt{e})\big)\Big)$$

   Dieser Knoten ist jetzt der einzige Knoten in $m$ und damit unser Ergebnis. Stellen wir diesen Knoten als Baum dar, so erhalten wir das in Abbildung 9.3 gezeigte Ergebnis. Wir haben hier jeden Knoten $n$ mit dem Funktionswert $n.\mathit{count}()$ beschriftet.

Die Kodierung, die sich daraus ergibt, wird in Tabelle 9.3 gezeigt.

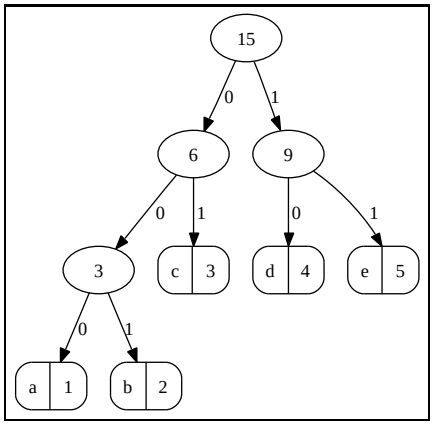


Abbildung 9.3: Baum-Darstellung der Kodierung.

| Buchstabe | a | b | c | d | e |
|---|---|---|---|---|---|
| Kodierung | 000 | 001 | 01 | 10 | 11 |

Tabelle 9.3: Kodierung der Buchstaben mit variabler Länge.

**Aufgabe 11**:

1. Berechnen Sie den Huffman-Code für einen Text, der nur die Buchstaben "a" bis "g" enthält und für den die Häufigkeiten, mit denen diese Buchstaben auftreten, durch die folgende Tabelle gegeben sind.

| Buchstabe | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| Häufigkeit | 1 | 1 | 2 | 3 | 5 | 8 | 13 |

Tabelle 9.4: Buchstaben mit Häufigkeiten.

2. Wie groß ist die Einsparung, wenn man die Buchstaben mit einem Huffman-Code kodiert gegenüber einer Kodierung mit drei Bits?
3. Versuchen Sie das Gesetz zu erkennen, nach dem die Häufigkeiten in der obigen Tabelle gebildet wurden und versuchen Sie, den Huffman-Code für den allgemeinen Fall, in dem $n$ Buchstaben gegeben sind, anzugeben.
4. Wie groß ist die Einsparung im allgemeinen Fall?

## 9.2 Optimalität des Huffman'schen Kodierungsbaums

In diesem Abschnitt zeigen wir, dass der durch den Algorithmus von Huffman berechnete Kodierungs-Baum der Kodierungs-Baum ist, für den die Funktion *cost*() minimal ist. Dazu geben zunächst eine andere Formel zur Berechnung von $n.cost()$ an: Wir definieren die *Tiefe* $n.depth(c)$ des Buchstabens $c$ in dem Kodierungs-Baum $n$ als den Abstand, den das Blatt $l$, das mit dem Buchstaben $c$ markiert ist, von der Wurzel des Kodierungs-Baums hat. Der Kodierungs-Baum wird dabei als Graph aufgefaßt. Dann gibt es genau einen Pfad von der Wurzel zu dem Blatt $l$ und $n.depth(c)$ wird als die Anzahl der Kanten dieses Pfades definiert. Jede der Kanten dieses Pfades trägt hinterher ein Bit zur Kodierung des Buchstabens bei, mit dem das Blatt $l$ markiert

ist. Um die gesamten Kosten zu berechnen müssen wir daher die Tiefe jedes Buchstabens mit seiner Häufigkeit multiplizieren. Bezeichnen wir die Häufigkeit des Buchstabens $c$ mit $\mathit{freq}(c)$, so erhalten wir insgesamt

$$n.\mathit{cost}() = \sum_{c \in \Sigma} \mathit{freq}(c) \cdot n.\mathit{depth}(c) \tag{9.2}$$

Die Summe läuft dabei über alle Buchstaben des Alphabets $\Sigma$, wobei wir voraussetzen, dass alle Buchstaben aus $\Sigma$ auch tatsächlich in dem Kodierungs-Baum auftreten. Buchstaben, die gar nicht auftreten, werden also vorher aus dem Alphabet entfernt.

**Definition 19 (Optimaler Kodierungs-Baum)** Ein Kodierungs-Baum $n$ ist *optimal*, wenn bei gegebener Häufigkeit der Buchstaben der Wert $n.\mathit{cost}()$ minimal ist, für alle Kodierungs-Bäume $k$, bei denen die Buchstaben mit der selben Häufigkeit auftreten wie in $n$, gilt also

$$n.\mathit{cost}() \leq k.\mathit{cost}().$$

**Lemma 20** Es seien $x$ und $y$ die beiden Buchstaben aus dem Alphabet $\Sigma$ mit der geringsten Häufigkeit. Dann gibt es einen optimalen Kodierungs-Baum $n$, bei dem sich die Kodierung der Buchstaben $x$ und $y$ nur im letzten Bit unterscheidet.

**Beweis**: Es sei $n_1$ ein optimaler Kodierungs-Baum. Wir zeigen, wie $n_1$ zu einem Kodierungs-Baum $n_2$ umgebaut werden kann, der einerseits optimal ist und bei dem sich andererseits die Kodierung der Buchstaben $x$ und $y$ nur im letzten Bit unterscheidet. Wir suchen in dem Kodierungs-Baum $n_1$ einen Knoten $k$ der Form $\mathit{Node}(l, r)$, der unter allen *inneren* Knoten eine maximale Tiefe hat. Dabei bezeichnen wir alle Knoten, die keine Blätter sind, als innere Knoten. An dem Knoten $k$ hängen zwei Buchstaben $a$ und $b$. Wir machen nun o.B.d.A. die folgenden Annahmen über die Häufigkeiten der Buchstaben:

1. $\mathit{freq}(x) \leq \mathit{freq}(y)$,
2. $\mathit{freq}(a) \leq \mathit{freq}(b)$.

Da wir angenommen haben, dass $x$ und $y$ die Buchstaben mit der geringsten Häufigkeiten sind, folgt daraus

$$\mathit{freq}(x) \leq \mathit{freq}(a) \quad \text{und} \quad \mathit{freq}(y) \leq \mathit{freq}(b). \tag{9.3}$$

Wir erhalten nun den Kodierungs-Baum $n_2$ aus dem Kodierungs-Baum $n_1$, indem wir in dem Baum $n_1$ die Positionen der Buchstaben $x$ und $a$ und die Positionen der Buchstaben $y$ und $b$ vertauschen. Daher gilt

$$\begin{aligned}
n_2.\mathit{depth}(a) &= n_1.\mathit{depth}(x), && (9.4)\\
n_2.\mathit{depth}(b) &= n_1.\mathit{depth}(y), && (9.5)\\
n_2.\mathit{depth}(x) &= n_1.\mathit{depth}(a), && (9.6)\\
n_2.\mathit{depth}(y) &= n_1.\mathit{depth}(b). && (9.7)
\end{aligned}$$

denn $a$ und $x$ und $b$ und $y$ vertauschen die Plätze. Für alle Buchstaben $c \in \Sigma \backslash \{a, b, x, y\}$ gilt natürlich

$$n_2.\mathit{depth}(c) = n_1.\mathit{depth}(c). \tag{9.8}$$

Weiterhin wissen wir aufgrund der Auswahl des Knotens $k$, dass

$$\begin{aligned}
&n_1.\mathit{depth}(a) = n_1.\mathit{depth}(b) \geq n_1.\mathit{depth}(x) \quad \text{und} && (9.9)\\
&n_1.\mathit{depth}(a) = n_1.\mathit{depth}(b) \geq n_1.\mathit{depth}(y) && (9.10)
\end{aligned}$$

gilt. Wir zeigen nun, dass $n_2.cost() \leq n_1.cost()$ gilt. Dazu geben wir zunächst $n_2.cost()$ an.

$$\begin{aligned} n_2.cost() &= \sum_{c \in \Sigma} freq(c) \cdot n_2.depth(c) \\ &= \sum_{c \in \Sigma \setminus \{a,b,x,y\}} freq(c) \cdot n_2.depth(c) \\ &\quad + freq(a) \cdot n_2.depth(a) + freq(b) \cdot n_2.depth(b) \\ &\quad + freq(x) \cdot n_2.depth(x) + freq(y) \cdot n_2.depth(y) \end{aligned}$$

Unter Berücksichtigung der Gleichungen (9.4) bis (9.8) können wir dies auch schreiben als

$$\begin{aligned} n_2.cost() &= \sum_{c \in \Sigma \setminus \{a,b,x,y\}} freq(c) \cdot n_1.depth(c) \\ &\quad + freq(a) \cdot n_1.depth(x) + freq(b) \cdot n_1.depth(y) \\ &\quad + freq(x) \cdot n_1.depth(a) + freq(y) \cdot n_1.depth(b) \end{aligned}$$

Analog berechnen wir $n_1.cost()$:

$$\begin{aligned} n_1.cost() &= \sum_{c \in \Sigma} freq(c) \cdot n_1.depth(c) \\ &= \sum_{c \in \Sigma \setminus \{a,b,x,y\}} freq(c) \cdot n_1.depth(c) \\ &\quad + freq(a) \cdot n_1.depth(a) + freq(b) \cdot n_1.depth(b) \\ &\quad + freq(x) \cdot n_1.depth(x) + freq(y) \cdot n_1.depth(y) \end{aligned}$$

Damit sehen wir, dass $n_2.cost() \leq n_1.cost()$ genau dann gilt, wenn die Ungleichung

$$\begin{aligned} &freq(a) \cdot n_1.depth(x) + freq(b) \cdot n_1.depth(y) + freq(x) \cdot n_1.depth(a) + freq(y) \cdot n_1.depth(b) \\ \leq\ &freq(a) \cdot n_1.depth(a) + freq(b) \cdot n_1.depth(b) + freq(x) \cdot n_1.depth(x) + freq(y) \cdot n_1.depth(y) \end{aligned}$$

erfüllt ist. Da in dieser Ungleichung nur noch der Knoten $n_1$ vorkommt, vereinfachen wir die Schreibweise und vereinbaren, dass wir einen Ausdruck der Form $n_1.depth(u)$ zu $depth(u)$ abkürzen. Die letzte Ungleichung ist dann äquivalent zu der Ungleichung

$$\begin{aligned} 0 \ \leq\ & freq(a) \cdot \big(depth(a) - depth(x)\big) + freq(b) \cdot \big(depth(b) - depth(y)\big) \\ & - freq(x) \cdot \big(depth(a) - depth(x)\big) - freq(y) \cdot \big(depth(b) - depth(y)\big) \end{aligned}$$

Diese Ungleichung vereinfachen wir zu

$$0 \ \leq\ \underbrace{\big(freq(a) - freq(x)\big)}_{\geq 0} \cdot \underbrace{\big(depth(a) - depth(x)\big)}_{\geq 0} + \underbrace{\big(freq(b) - freq(y)\big)}_{\geq 0} \cdot \underbrace{\big(depth(b) - depth(y)\big)}_{\geq 0}$$

Hier gilt $freq(a) - freq(x) \geq 0$ wegen Ungleichung 9.3, die Ungleichung $depth(a) - depth(x) \geq 0$ folgt aus Ungleichung 9.9, die Ungleichung $freq(b) - freq(y) \geq 0$ folgt aus Ungleichung 9.3 und die Ungleichung $depth(b) - depth(y) \geq 0$ folgt aus Ungleichung 9.10. Damit haben wir

$$n_2.cost() \leq n_1.cost()$$

gezeigt. Da $n_1$ optimal ist, muss auch $n_2$ optimal sein. Nach Wahl des Knotens $k$ unterscheiden sich die Kodierungen von $x$ und $y$ nur in dem letzten Bit. Damit ist $n_2$ der gesuchte Kodierungs-Baum. □

**Satz 21** Der Kodierungs-Baum, der von dem Huffman-Algorithmus erzeugt wird, ist optimal.

**Beweis**: Wir beweisen den Satz durch Induktion über die Anzahl $n$ der Buchstaben in dem Alphabet $\Sigma$.

I.A.: $n = 2$. Es sei $\Sigma = \{a, b\}$. In diesem Fall führt der Huffman-Algorithmus nur einen Schritt durch und liefert den Kodierungs-Baum

$$k = \mathit{Node}\big(\mathit{Leaf}(a, \mathit{freq}(a)), \mathit{Leaf}(b, \mathit{freq}(b))\big).$$

Bei der Kodierung eines Alphabets, das aus zwei Buchstaben besteht, haben wir keine Wahl, was die Länge der Kodes angeht: Wir brauchen für jeden Buchstaben genau ein Bit und daher ist das vom Huffman-Algorithmus in diesem Fall gelieferte Ergebnis offenbar optimal.

I.S.: $n \mapsto n + 1$

Wir gehen jetzt davon aus, dass das Alphabet $\Sigma$ aus $n + 1$ Buchstaben besteht. Es seien $x$ und $y$ die beiden Buchstaben, deren Häufigkeit minimal ist. Es sei $z$ ein neuer Buchstabe, der nicht in dem Alphabet $\Sigma$ auftritt. Wir definieren ein neues Alphabet $\Sigma'$ als

$$\Sigma' = \big(\Sigma \backslash \{x, y\}\big) \cup \{z\}.$$

Die Häufigkeit des neuen Buchstabens $z$ definieren wir als

$$\mathit{freq}(z) := \mathit{freq}(x) + \mathit{freq}(y).$$

Dann enthält das Alphabet $\Sigma'$ insgesamt $n$ Buchstaben. Wenden wir den Huffman-Algorithmus auf dieses Alphabet an, so erhalten wir nach Induktions-Voraussetzung für $\Sigma'$ einen optimalen Kodierungs-Baum $k_1$. Die Anwendung des Huffman-Algorithmus auf das Alphabet $\Sigma$ ersetzt in diesem Kodierungs-Baum das Blatt, das mit dem Buchstaben $z$ markiert ist, durch den Knoten

$$\mathit{Node}\big(\mathit{Leaf}(x, \mathit{freq}(x)), \mathit{Leaf}(y, \mathit{freq}(y))\big).$$

Bezeichnen wir den so entstanden Kodierungs-Baum mit $k_2$, so müssen wir zeigen, dass $k_2$ optimal ist. Wir führen den Beweis indirekt und nehmen an, dass $k_2$ nicht optimal ist. Dann gibt es einen Kodierungs-Baum $k_3$ für das Alphabet $\Sigma$, so dass

$$k_3.\mathit{cost}() < k_2.\mathit{cost}()$$

ist. Nach Lemma 20 können wir o.B.d.A. voraussetzen, dass sich die Kodierung der Buchstaben $x$ und $y$ in dem Kodierungs-Baum $k_3$ nur in dem letzten Bit unterscheidet. Also gibt es in dem Kodierungs-Baum $k_3$ einen Knoten der Form

$$\mathit{Node}\big(\mathit{Leaf}(x, \mathit{freq}(x)), \mathit{Leaf}(y, \mathit{freq}(y))\big).$$

Wir transformieren den Kodierungs-Baum $k_3$ in einen Kodierungs-Baum $k_4$ für das Alphabet $\Sigma'$ indem wir diesen Knoten durch das Blatt

$$\mathit{Leaf}\big(z, \mathit{freq}(x) + \mathit{freq}(y)\big)$$

ersetzen. Damit gilt

$$\begin{aligned}
k_4.\mathit{cost}() &= \sum_{c \in \Sigma'} \mathit{freq}(c) \cdot k_4.\mathit{depth}(c) \\
&= \sum_{c \in \Sigma \backslash \{x,y\} \cup \{z\}} \mathit{freq}(c) \cdot k_4.\mathit{depth}(c) \\
&= \sum_{c \in \Sigma \backslash \{x,y\}} \mathit{freq}(c) \cdot k_4.\mathit{depth}(c) + \mathit{freq}(z) \cdot k_4.\mathit{depth}(z) \\
&= \sum_{c \in \Sigma \backslash \{x,y\}} \mathit{freq}(c) \cdot k_3.\mathit{depth}(c) + \mathit{freq}(z) \cdot \big(k_3.\mathit{depth}(x) - 1\big)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{c \in \Sigma \setminus \{x,y\}} \mathit{freq}(c) \cdot k_3.\mathit{depth}(c) \\
&\quad + \big(\mathit{freq}(x) + \mathit{freq}(y)\big) \cdot \big(k_3.\mathit{depth}(x) - 1\big) \\
&= \sum_{c \in \Sigma} \mathit{freq}(c) \cdot k_3.\mathit{depth}(c) - \big(\mathit{freq}(x) + \mathit{freq}(y)\big) \\
&= k_3.\mathit{cost}() - \big(\mathit{freq}(x) + \mathit{freq}(y)\big)
\end{aligned}
$$

Wir halten dieses Ergebnis in einer Gleichung fest:

$$
k_4.\mathit{cost}() = k_3.\mathit{cost}() - \big(\mathit{freq}(x) + \mathit{freq}(y)\big). \tag{9.11}
$$

Da die Kodierungs-Bäume $k_1$ und $k_2$ in der selben Relation stehen wie die Kodierungs-Bäume $k_4$ und $k_3$, gilt analog

$$
k_1.\mathit{cost}() = k_2.\mathit{cost}() - \big(\mathit{freq}(x) + \mathit{freq}(y)\big). \tag{9.12}
$$

Damit können wir zeigen, dass die Kosten des Kodierungs-Baums $k_4$ geringer sind als die Kosten des Kodierungs-Baums $k_1$:

$$
\begin{aligned}
k_4.\mathit{cost}() &= k_3.\mathit{cost}() - \big(\mathit{freq}(x) + \mathit{freq}(y)\big) \\
&< k_2.\mathit{cost}() - \big(\mathit{freq}(x) + \mathit{freq}(y)\big) \\
&= k_1.\mathit{cost}().
\end{aligned}
$$

Dieses Ergebnis steht aber im Widerspruch dazu, dass der Kodierungs-Baum $k_1$ optimal ist. Folglich ist die Annahme $k_3.\mathit{cost}() < k_2.\mathit{cost}()$ falsch und der Kodierungs-Baum $k_2$ ist bereits optimal. □

# Kapitel 10

# Graphentheorie

Wir wollen zum Abschluß der Vorlesung wenigstens ein graphentheoretisches Problem vorstellen: Das Problem der Berechnung kürzester Wege.

## 10.1 Die Berechnung kürzester Wege

Um das Problem der Berechnung kürzester Wege formulieren zu können, führen wir zunächst den Begriff des *gewichteten Graphen* ein.

**Definition 22 (Gewichteter Graph)** *Ein* gewichteter Graph *ist ein Tripel* $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ *so dass gilt:*

1. $\mathbb{V}$ *ist eine Menge von* Knoten.
2. $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ *ist eine Menge von* Kanten.
3. $\|\cdot\| : \mathbb{E} \to \mathbb{N}\backslash\{0\}$ *ist eine Funktion, die jeder Kante eine positive* Länge *zuordnet.* □

Ein *Pfad* $P$ ist eine Liste der Form

$$P = [x_1, x_2, x_3, \cdots, x_n]$$

so dass für alle $i = 1, \cdots, n-1$ gilt:

$$\langle x_i, x_{i+1} \rangle \in \mathbb{E}.$$

Die Menge aller Pfade bezeichnen wir mit $\mathbb{P}$. Die Länge eines Pfads definieren wir als die Summe der Länge aller Kanten:

$$\big\|[x_1, x_2, \cdots, x_n]\big\| \;:=\; \sum_{i=1}^{n-1} \big\|\langle x_i, x_{i+1}\rangle\big\|.$$

Ist $p = [x_1, x_2, \cdots, x_n]$ ein Pfad, so sagen wir, dass $p$ den Knoten $x_1$ mit dem Knoten $x_n$ verbindet. Die Menge alle Pfade, die den Knoten $v$ mit dem Knoten $w$ verbinden, bezeichnen wir als

$$\mathbb{P}(v, w) \;:=\; \big\{[x_1, x_2, \cdots, x_n] \in \mathbb{P} \mid x_1 = v \wedge x_n = w\big\}.$$

Damit können wir nun das Problem der Berechnung kürzester Wege formulieren.

**Definition 23 (Kürzeste-Wege-Problem)** *Gegeben sei ein gewichteter Graph* $G = \langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ *und ein Knoten* $\texttt{source} \in \mathbb{V}$. *Dann besteht das* kürzeste-Wege-Problem *darin, die folgende Funktion zu berechnen:*

$$\texttt{sp} : \mathbb{V} \to \mathbb{N}$$
$$\texttt{sp}(v) \;:=\; \texttt{min}\big\{\|p\| \mid p \in \mathbb{P}(\texttt{source}, v)\big\}.$$ □

### 10.1.1 Der Algorithmus von Moore

Wir betrachten zunächst den Algorithmus von Moore [Moo59] zur Berechnung des kürzeste-Wege-Problems. Abbildung 10.1 zeigt eine Implementierung dieses Algorithmus' in SetlX.

```
1  shortestPath := procedure(source, edges) {
2      dist   := { [source, 0] };
3      fringe := { source };
4      while (fringe != {}) {
5          u := arb(fringe);
6          fringe -= { u };
7          n := {[v,l]:[v,l] in edges[u] | dist[v] == om || dist[u]+l < dist[v]};
8          for ([v,l] in n) {
9              dist[v] := dist[u] + l;
10             fringe  += { v };
11         }
12     }
13     return dist;
14 };
```

Abbildung 10.1: Algorithmus von Moore zur Lösung des kürzeste-Wege-Problems.

1. Die Funktion `shortestPath`(*source*, *edges*) wird mit zwei Parametern aufgerufen:

   (a) *source* ist der Start-Knoten, von dem aus wir die Entfernungen zu den anderen Knoten berechnen.

   (b) *edges* ist eine binäre Relation, die für jeden Knoten $x$ eine Menge von Paaren der Form

   $$\{[y_1, l_1], \cdots, [y_n, l_n]\}$$

   speichert. Die Idee dabei ist, dass für jeden in dieser Menge gespeicherten Knoten $y_i$ eine Kante $\langle x, y_i \rangle$ der Länge $l_i$ existiert.

2. Die Variable *dist* speichert die Abstands-Funktions als binäre Relation, also als Menge von Paaren der Form $[x, \mathtt{sp}(x)]$. Hierbei ist $x \in \mathbb{V}$ und $\mathtt{sp}(x)$ ist der Abstand, den der Knoten $x$ von dem Start-Knoten *source* hat.

   Der Knoten *source* hat von dem Knoten *source* offenbar den Abstand 0 und zu Beginn unserer Rechnung ist das auch alles, was wir wissen. Daher wird die Relation *dist* in Zeile 2 mit dem Paar `[source, 0]` initialsiert.

3. Die Variable *fringe* enthält alle die Knoten, von denen ausgehend wir als nächstes die Abstände benachbarter Knoten berechnen sollten. Am Anfang wissen wir nur von *source* den Abstand und daher ist dies der einzige Knoten, mit dem wir die Menge *fringe* in Zeile 3 initialisieren.

4. Solange es nun Knoten gibt, von denen ausgehend wir neue Wege berechnen können, wählen wir in Zeile 5 einen beliebigen Knoten aus der Menge *fringe* aus und entfernen ihn aus dieser Menge.

5. Anschließend berechnen wir die Menge aller Knoten $v$, für die wir jetzt einen neuen Abstand gefunden haben:

   (a) Das sind einerseits die Knoten $v$, für welche die Funktion *dist*$(v)$ bisher noch undefiniert war, weil wir diese Knoten in unserer bisherigen Reechnung noch gar nicht gesehen haben.

(b) Andererseits sind dies aber auch Knoten, für die wir schon einen Abstand haben, der aber größer ist als der Abstand des Weges, den wir erhalten, wenn wir die Knoten von $u$ aus besuchen.

Für alle diese Knoten berechnen wir den Abstand und fügen diese Knoten dann in die Menge *fringe* ein.

6. Der Algorithmus terminiert, wenn die Menge *fringe* leer ist, denn dann haben wir alle Knoten abgeklappert.

### 10.1.2 Der Algorithmus von Dijkstra

```
1  shortestPath := procedure(source, edges) {
2      dist    := { [source, 0] };
3      fringe  := { [0, source] };
4      visited := {};
5      while (fringe != {}) {
6          [d, u]  := first(fringe);
7          fringe  -= { [d, u] };
8          n := {[v,l] : [v,l] in edges[u] | dist[v] == om || d + l < dist[v]};
9          for ([v,l] in n) {
10             dvOld := dist[v];
11             dvNew := d + l;
12             dist[v] := dvNew;
13             if (dvOld != om) {
14                 fringe -= { [dvOld, v] };
15             }
16             fringe  += { [dvNew, v] };
17         }
18         visited += { u };
19     }
20     return dist;
21 };
```

Abbildung 10.2: Der Algorithmus von Dijkstra zur Lösung des kürzeste-Wege-Problems.

Im Algorithmus von Moore ist die Frage, in welcher Weise die Knoten aus der Menge *fringe* ausgewählt werden, nicht weiter spezifiziert. Die Idee bei dem von Edsger W. Dijkstra (1930 – 2002) im Jahre 1959 veröffentlichten Algorithmus [Dij59] besteht darin, immer den Knoten auszuwählen, der den geringsten Abstand zu dem Knoten *source* hat. Dazu wird die Menge *fringe* nun als eine Prioritäts-Warteschlange implementiert. Als Prioritäten wählen wir die Entfernungen zu dem Knoten `source`. Abbildung 10.2 auf Seite 178 zeigt die Implementierung des Algorithmus von Dijkstra zur Berechnung der kürzesten Wege in der Sprache SetlX.

In dem Problem in Abbildung taucht noch eine Variable mit dem Namen *visited* auf. Diese Variable bezeichnet die Menge der Knoten, die der Algorithmus schon *besucht* hat. Genauer sind das die Knoten $u$, die aus der Prioritäts-Warteschlange `fringe` entfernt wurden und für die dann anschließend in der `for`-Schleife, die in Zeile 9 beginnt, alle zu $u$ benachbarten Knoten untersucht wurden. Die Menge *visited* hat für die eigentliche Implementierung des Algorithmus keine Bedeutung, denn die Variable *visited* wird nur in Zeile 4 und Zeile 18 geschrieben, aber sie wird an keiner Stelle gelesen. Ich habe die Variable *visited* nur deshalb eingeführt, damit ich eine Invariante formulieren kann, die für den Beweis der Korrektheit des Algorithmus zentral ist. Diese Invariante lautet

$$\forall u \in \texttt{visited} : \mathit{dist}[u] = \mathit{sp}(u).$$

Für alle Knoten aus *visited* liefert die Funktion *dist*() also bereits den kürzesten Abstand zum Knoten *source*.

**Beweis**: Wir zeigen durch Induktion, dass jedesmal wenn wir einen Knoten $u$ in die Menge *visited* einfügen, die Gleichung

$$\mathit{dist}(u) = \mathit{sp}(u)$$

gilt. In dem Programm gibt es genau zwei Stellen, an denen die Menge *visited* verändert wird.

I.A.: Zu Beginn wird die Menge *visited* mit der leeren Menge initialisiert und daher ist die Behauptung für alle Elemente, die zu Beginn in der Menge *visited* sind, trivialerweise richtig.

I.S.: In Zeile 18 fügen wir den Knoten $u$ in die Menge *visited* ein. Wir betrachten nun die Situation unmittelbar vor dem Einfügen von $u$. Falls $u$ bereits ein Elemente der Menge *visited* sein sollte, gilt die Behauptung nach Induktions-Voraussetzung. Wir brauchen also nur den Fall betrachten, dass $u$ vor dem Einfügen noch kein Element der Menge *visited* ist.

Wir führen den weiteren Beweis nun indirekt und nehmen an, dass

$$\mathit{dist}(u) > \mathit{sp}(u)$$

gilt. Dann gibt es einen kürzesten Pfad

$$p = [x_0 = \mathit{source}, x_1, \cdots, x_n = u]$$

von *source* nach $u$, der insgesamt die Länge $\mathit{sp}(u)$ hat. Es sei $i \in \{0, \cdots, n-1\}$ der Index für den

$$x_0 \in \mathit{visited}, \cdots, x_i \in \mathit{visited} \quad \text{aber} \quad x_{i+1} \notin \texttt{Visited},$$

gilt, $x_i$ ist also der erste Knoten aus dem Pfad $p$, für den $x_{i+1}$ nicht mehr in der Menge *visited* liegt. Nachdem $x_i$ in die Menge Visited eingefügt wurde, wurde für alle Knoten, die mit $x_i$ über eine Kante verbunden sind, die Funktion *dist* neu ausgerechnet. Insbesondere wurde auch $\mathit{dist}[x_{i+1}]$ neu berechnet und der Knoten $x_{i+1}$ wurde spätestens zu diesem Zeitpunkt in die Menge *fringe* eingefügt. Außerdem wissen wir, dass $\mathit{dist}[x_{i+1}] = \mathit{sp}(x_{i+1})$ gilt, denn nach Induktions-Voraussetzung gilt $\mathit{dist}[x_i] = \mathit{sp}(x_i)$ und die Kante $\langle x_i, x_{i+1} \rangle$ ist Teil eines kürzesten Pfades von $x_i$ nach $x_{i+1}$.

Da wir nun angenommen haben, dass $x_{i+1} \notin \mathit{visited}$ ist, muss $x_{i+1}$ immer noch in der Prioritäts-Warteschlange *fringe* liegen. Also muss $\mathit{dist}[x_{i+1}] \geq \mathit{dist}[u]$ gelten, denn sonst wäre $x_{i+1}$ vor $u$ aus der Prioritäts-Warteschlange entfernt worden. Wegen $\mathit{sp}(x_{i+1}) = \mathit{dist}[x_{i+1}]$ haben wir dann aber den Widerspruch

$$\mathit{sp}(u) \geq \mathit{sp}(x_{i+1}) = \mathit{dist}[x_{i+1}] \geq \mathit{dist}[u] > \mathit{sp}(u). \qquad \square$$

### 10.1.3 Komplexität

Wenn ein Knoten $u$ aus der Warteschlange *fringe* entfernt wird, ist er anschließend ein Element der Menge *visited* und aus der oben gezeigten Invariante folgt, dass dann

$$\mathit{sp}(u) = \mathit{dist}[u]$$

gilt. Daraus folgt aber notwendigerweise, dass der Knoten $u$ nie wieder in die Prioritäts-Warteschlange *fringe* eingefügt werden kann, denn ein Knoten $v$ wird nur dann in *fringe* neu eingefügt, wenn die Funktion $\mathit{dist}[v]$ noch undefiniert ist. Das Einfügen eines Knoten in eine Prioritäts-Warteschlange mit $n$ Elementen kostet eine Rechenzeit, die durch $\mathcal{O}\big(\log_2(n)\big)$ abgeschätzt werden kann. Da die Warteschlange sicher nie mehr als $\#\mathbb{V}$ Knoten enthalten kann und da jeder Knoten maximal einmal eingefügt werden kann, liefert das einen Term der Form

$$\mathcal{O}\big(\#V \cdot \log_2(\#V)\big)$$

für das Einfügen der Knoten. Neben dem Einfügen eines Knotens durch den Befehl

```
fringe += { [dvNew, v] };
```

müssen wir auch die Komplexität des Aufrufs

```
fringe -= { [dvOld, v] };
```

analysieren. Die Anzahl dieser Aufrufe ist durch die Anzahl der Kanten begrenzt, die zu dem Knoten $v$ hinfügen. Da das Entfernen eines Elements aus einer Menge mit $n$ Elementen eine Rechenzeit der Größe $\mathcal{O}\big(\log_2(n)\big)$ erfordert, haben wir die Abschätzung

$$\mathcal{O}\big(\#\mathbb{E} \cdot \log_2(\#\mathbb{V})\big)$$

für diese Rechenzeit. Dabei bezeichnet $\#\mathbb{E}$ die Anzahl der Kanten. Damit erhalten wir für die Komplexität von Dijkstra's Algorithmus insgesamt den Ausdruck

$$\mathcal{O}\big((\#\mathbb{E} + \#\mathbb{V}) * \ln(\#\mathbb{V})\big).$$

Ist die Zahl der Kanten, die von den Knoten ausgehen können, durch eine feste Zahl begrenzt (z.B. wenn von jedem Knoten nur maximal 4 Kanten ausgehen), so kann die Gesamt-Zahl der Kanten durch ein festes Vielfaches der Knoten-Zahl abgeschätzt werden. Dann ist die Komplexität für Dijkstra's Algorithmus zur Bestimmung der kürzesten Wege durch den Ausdruck

$$\mathcal{O}\big(\#\mathbb{V} * \log_2(\#\mathbb{V})\big)$$

gegeben.

# Literaturverzeichnis


[AHU87] Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[AVL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[BM93] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CP03] Frank M. Carrano and Janet J. Prichard. *Data Abstraction and Problem Solving with Java*. Prentice Hall, 2003.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[dlB59] René de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of the Western Joint Computer Conference*, pages 195–298, 1959.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4:321, 1961.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, 2nd edition, 2006.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.

[Sed02] Robert Sedgewick. *Algorithms in Java*. Pearson, 2002.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[WS92] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Assoc., 1992.