

Informatik **II**: Algorithmen und Daten-Strukturen

— Sommersemester 2014 —

DHBW Stuttgart

Prof. Dr. Karl Stroetmann

28. November 2015

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einführung | 4 |
| 1.1 | Motivation | 4 |
| 1.1.1 | Überblick | 4 |
| 1.2 | Algorithmen und Programme | 6 |
| 1.3 | Eigenschaften von Algorithmen und Programmen | 6 |
| 1.4 | Literatur | 7 |
| 2 | Grenzen der Berechenbarkeit | 8 |
| 2.1 | Das Halte-Problem | 8 |
| 2.2 | Unlösbarkeit des Äquivalenz-Problems | 11 |
| 3 | Die \mathcal{O}-Notation | 13 |
| 3.1 | Motivation | 13 |
| 3.2 | Fallstudie: Effiziente Berechnung der Potenz | 18 |
| 3.3 | Der Hauptsatz der Laufzeit-Funktionen | 22 |
| 4 | Der Hoare-Kalkül | 29 |
| 4.1 | Vor- und Nachbedingungen | 29 |
| 4.1.1 | Spezifikation von Zuweisungen | 30 |
| 4.1.2 | Die Abschwächungs-Regel | 32 |
| 4.1.3 | Zusammengesetzte Anweisungen | 32 |
| 4.1.4 | Alternativ-Anweisungen | 33 |
| 4.1.5 | Schleifen | 35 |
| 4.2 | Der Euklid'sche Algorithmus | 35 |
| 4.2.1 | Nachweis der Korrektheit des Euklid'schen Algorithmus | 35 |
| 4.2.2 | Maschinelle Programm-Verifikation | 40 |
| 4.3 | Symbolische Programm-Ausführung | 42 |
| 5 | Abstrakte Daten-Typen und elementare Daten-Stukturen | 45 |
| 5.1 | Abstrakte Daten-Typen | 45 |
| 5.2 | Darstellung abstrakter Daten-Typen in <i>Java</i> | 47 |
| 5.3 | Implementierung eines Stacks mit Hilfe eines <i>Arrays</i> | 50 |
| 5.4 | Eine Listen-basierte Implementierung von Stacks | 52 |
| 5.5 | Auswertung arithmetischer Ausdrücke | 54 |
| 5.5.1 | Ein einführendes Beispiel | 55 |
| 5.5.2 | Ein Algorithmus zur Auswertung arithmetischer Ausdrücke | 57 |
| 5.6 | Nutzen abstrakter Daten-Typen | 62 |
| 6 | Sortier-Algorithmen | 64 |
| 6.1 | Sortieren durch Einfügen | 65 |
| 6.1.1 | Komplexität | 66 |
| 6.2 | Sortieren durch Auswahl | 68 |

| | | |
|----------|---|------------|
| 6.2.1 | Komplexität | 70 |
| 6.2.2 | Eine feldbasierte Implementierung | 70 |
| 6.3 | Sortieren durch Mischen | 72 |
| 6.3.1 | Komplexität | 75 |
| 6.3.2 | Eine feldbasierte Implementierung | 78 |
| 6.3.3 | Eine nicht-rekursive Implementierung von <i>Sortieren durch Mischen</i> | 80 |
| 6.4 | Der <i>Quick-Sort</i> -Algorithmus | 82 |
| 6.4.1 | Komplexität | 83 |
| 6.4.2 | Eine feldbasierte Implementierung von <i>Quick-Sort</i> | 87 |
| 6.4.3 | Korrektheit | 90 |
| 6.4.4 | Mögliche Verbesserungen | 92 |
| 6.5 | Timsort | 94 |
| 6.5.1 | Bewertung | 107 |
| 7 | Mengen und Abbildungen | 108 |
| 7.1 | Der abstrakte Daten-Typ der <i>Abbildung</i> | 108 |
| 7.2 | Geordnete binäre Bäume | 110 |
| 7.2.1 | Implementierung geordneter binärer Bäume in <i>Java</i> | 113 |
| 7.2.2 | Analyse der Komplexität | 118 |
| 7.3 | AVL-Bäume | 123 |
| 7.3.1 | Implementierung von AVL-Bäumen in <i>Java</i> | 127 |
| 7.3.2 | Analyse der Komplexität | 131 |
| 7.4 | Tries | 136 |
| 7.4.1 | Einfügen in Tries | 138 |
| 7.4.2 | Löschen in Tries | 139 |
| 7.4.3 | Implementierung in <i>Java</i> | 140 |
| 7.5 | Hash-Tabellen | 144 |
| 7.6 | Mengen und Abbildungen in <i>Java</i> | 151 |
| 7.6.1 | Das Interface <code>Collection<E></code> | 151 |
| 7.6.2 | Anwendungen von Mengen | 156 |
| 7.6.3 | Die Schnittstelle <code>Map<K,V></code> | 157 |
| 7.6.4 | Anwendungen | 160 |
| 7.7 | Das Wolf-Ziege-Kohl-Problem | 161 |
| 7.7.1 | Die Klasse <code>ComparableSet</code> | 161 |
| 7.7.2 | Die Klasse <code>ComparableList</code> | 170 |
| 7.7.3 | Lösung des Wolf-Ziege-Kohl-Problems in <i>Java</i> | 173 |
| 8 | Prioritäts-Warteschlangen | 178 |
| 8.1 | Definition des ADT <i>PrioQueue</i> | 178 |
| 8.2 | Die Daten-Struktur <i>Heap</i> | 180 |
| 8.3 | Implementierung in <i>Java</i> | 183 |
| 8.3.1 | Implementierung der Methode <i>change</i> | 185 |
| 8.3.2 | Prioritäts-Warteschlangen in <i>Java</i> | 193 |
| 9 | Daten-Kompression | 197 |
| 9.1 | Der Algorithmus von Huffman | 198 |
| 9.1.1 | Implementierung in <i>Java</i> | 202 |
| 9.2 | Optimalität des Huffman'schen Kodierungsbaums | 206 |

| | |
|---|------------|
| 10 Graphentheorie | 211 |
| 10.1 Die Berechnung kürzester Wege | 211 |
| 10.1.1 Ein naiver Algorithmus zur Lösung des kürzeste-Wege-Problems | 212 |
| 10.1.2 Der Algorithmus von Moore | 213 |
| 10.1.3 Der Algorithmus von Dijkstra | 215 |
| 10.1.4 Implementierung in <i>Java</i> | 216 |
| 10.1.5 Komplexität | 219 |
| 11 Die Monte-Carlo-Methode | 220 |
| 11.1 Berechnung der Kreiszahl π | 220 |
| 11.2 Theoretischer Hintergrund | 222 |
| 11.3 Erzeugung zufälliger Permutationen | 224 |

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, das 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 verschiedenen 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 anhand 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 Implementierung 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. Syntaktisch 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, wohingegen 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 zweckmäßigsten 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 dieselbe Idee. Genauso kann ein Algorithmus in verschiedenen Programmier-Sprachen kodiert werden, es bleibt derselbe 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 gewährleistet.
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*

$$n := \text{procedure}(x) \{ \dots \}$$

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 $\text{name}(t) = n$. \square

Beispiele:

1. $s_1 = \text{"simple := procedure}(x) \{ \text{return } 0; \} \text{"}$

s_1 ist eine (sehr einfache) Test-Funktion mit dem Namen `simple`.

2. $s_2 = \text{"loop := procedure}(x) \{ \text{while (true)} \{ x := x + 1; \} \} \text{"}$

s_2 ist eine Test-Funktion mit dem Namen `loop`.

3. $s_3 = \text{"hugo := procedure}(x) \{ \text{return ++x; } \} \text{"}$

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, \dots, a_k Argumente, die vom Typ her der Deklaration von n entsprechen, so schreiben wir*

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

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

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

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

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

und sagen, dass der Aufruf $n(a_1, \dots, a_k)$ 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

$$\text{stops} := \text{procedure}(t, a) \{ \dots \}$$

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

1. $t \notin TF \Leftrightarrow \text{stops}(t, a) \rightsquigarrow 2$.

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

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

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. $t \in TF \wedge \text{name}(t) = n \wedge n(a) \uparrow \Leftrightarrow \text{stops}(t, a) \rightsquigarrow 0$.

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:

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

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

$$\text{stops}(turing, turing);$$

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

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

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. $\text{stops}(\text{turing}, \text{turing}) \rightsquigarrow 0$.

Nach der Spezifikation von **stops** bedeutet dies

$$\text{alan}(\text{turing}) \uparrow$$

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. $\text{stops}(\text{turing}, \text{turing}) \rightsquigarrow 1$.

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

$$\text{alan}(\text{turing}) \downarrow$$

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. \square

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 Programmiersprache gibt, in der wir das Halte-Problem dann vielleicht doch lösen könnten. Wenn es in dieser Programmiersprache 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 Programmiersprache übertragen lässt.

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

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

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:

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

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 bestimmte 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 bewiesenen Sachverhalt 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, \dots, a_k seien Argumente, mit denen wir diese Funktionen füttern können. Wir definieren*

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

g.d.w. einer der beiden folgenden Fälle auftritt:

1. $n_1(a_1, \dots, a_k) \uparrow \quad \wedge \quad n_2(a_1, \dots, a_k) \uparrow$
2. $\exists r : \left(n_1(a_1, \dots, a_k) \rightsquigarrow r \quad \wedge \quad n_2(a_1, \dots, a_k) \rightsquigarrow r \right)$

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

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

$$\text{equal} := \text{procedure}(\text{p1}, \text{p2}, \text{a}) \{ \dots \}$$

hat, möge folgender Spezifikation genügen:

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

gilt, dann muss gelten:

$$\text{equal}(p_1, p_2, a) \rightsquigarrow 1.$$

3. Ansonsten gilt

$$\text{equal}(p_1, p_2, a) \rightsquigarrow 0.$$

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 angegebenen 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. \square

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 \mathcal{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, Zuweisungen, 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:

¹Da 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 | 0.750000000000000 |
| 10 | 0.06454630495800 |
| 100 | 0.00662481908150 |
| 1000 | 0.00066622484855 |
| 10 000 | 6.6662224852 e-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:

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

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:*

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

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. \square

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

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

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 $f: \mathbb{N} \rightarrow \mathbb{R}_+$ gilt

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

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

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

Satz 8 (Abgeschlossenheit unter Multiplikation mit Konstanten)

Es seien $f, g: \mathbb{N} \rightarrow \mathbb{R}_+$ 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

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

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

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

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

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

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

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

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

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

$$\forall n \in \mathbb{N}: (n \geq k_1 \rightarrow f(n) \leq c_1 \cdot h(n)) \quad \text{und}$$

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

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 $n \geq k$

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

Satz 10 (Transitivität) *Es seien $f, g, h: \mathbb{N} \rightarrow \mathbb{R}_+$. 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

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

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

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

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 $f(n) \leq c \cdot h(n)$. \square

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

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

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

Beweis: Es sei

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

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

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

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

Wir zeigen die Nützlichkeit der obigen Sätze anhand einiger Beispiele.

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

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

Beweis: Es gilt

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

Die Behauptung folgt nun aus dem Grenzwert-Satz. \square

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

$$\lim_{n \rightarrow \infty} \frac{n^k}{\lambda^n} = 0 \quad (3.7)$$

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

$$\lim_{n \rightarrow \infty} \frac{n^k}{\lambda^n} = \lim_{x \rightarrow \infty} \frac{x^k}{\lambda^x} = \lim_{x \rightarrow \infty} \frac{\frac{d x^k}{d x}}{\frac{d \lambda^x}{d x}}$$

Die Ableitungen können wir berechnen, es gilt:

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

Berechnen wir die zweite Ableitung so sehen wir

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

Für die k -te Ableitung gilt analog

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

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

$$\lim_{x \rightarrow \infty} \frac{x^k}{\lambda^x} = \lim_{x \rightarrow \infty} \frac{\frac{d x^k}{d x}}{\frac{d \lambda^x}{d x}} = \lim_{x \rightarrow \infty} \frac{\frac{d^2 x^k}{d x^2}}{\frac{d^2 \lambda^x}{d x^2}} = \dots = \lim_{x \rightarrow \infty} \frac{\frac{d^k x^k}{d x^k}}{\frac{d^k \lambda^x}{d x^k}} = \lim_{x \rightarrow \infty} \frac{k!}{\ln(\lambda)^k \lambda^x} = 0.$$

\square

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

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

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

ist. Es gilt

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

Also haben wir

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

\square

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

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

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

$$\begin{aligned}
& \ln(3^n) & \leq & \ln(c \cdot 2^n) \\
\leftrightarrow & n \cdot \ln(3) & \leq & \ln(c) + n \cdot \ln(2) \\
\leftrightarrow & n \cdot (\ln(3) - \ln(2)) & \leq & \ln(c) \\
\leftrightarrow & n & \leq & \frac{\ln(c)}{\ln(3) - \ln(2)}
\end{aligned}$$

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. $7 \cdot n + (\log_2(n))^2 \in \mathcal{O}(n)$
4. $\sqrt{n} + \log_2(n) \in \mathcal{O}(\sqrt{n})$
5. $n^n \in \mathcal{O}(2^{2^n})$.

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

$$m^n = \underbrace{m \cdot \dots \cdot m}_n$$

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:

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

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:

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

1. **Induktions-Anfang.**

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

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

2. **Induktions-Schritt.**

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

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

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

$$\begin{aligned} \text{power}(m, n) &\rightsquigarrow \text{power}(m, k) \cdot \text{power}(m, k) \\ &\stackrel{I.V.}{\rightsquigarrow} m^k \cdot m^k \\ &= m^{2 \cdot k} \\ &= m^n. \end{aligned}$$

(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

$$\begin{aligned} \text{power}(m, n) &\rightsquigarrow \text{power}(m, k) \cdot \text{power}(m, k) \cdot m \\ &\stackrel{I.V.}{\rightsquigarrow} m^k \cdot m^k \cdot m \\ &= m^{2 \cdot k + 1} \\ &= m^n. \end{aligned}$$

Damit ist der Beweis der Korrektheit abgeschlossen. \square

Als nächstes wollen wir die Komplexität des obigen Programms untersuchen. Dazu berechnen wir zunächst die Anzahl der Multiplikationen, die beim Aufruf `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 `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:

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

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

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

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

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

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

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

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:

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

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

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

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

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

Offenbar lautet die Lösung

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

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

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

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

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

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. $f : \mathbb{N} \rightarrow \mathbb{R}_+$,
3. die Funktion $g : \mathbb{N} \rightarrow \mathbb{R}_+$ genüge der Rekurrenz-Gleichung

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

wobei der Ausdruck n/β die ganzzahlige Division von n durch β 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

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

gilt, dann haben wir

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

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

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

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

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

gilt, dann folgt

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

□

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

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

Wir beschränken uns auf solche Werte von n , die sich als Potenzen von β 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 $(b_k)_{k \in \mathbb{N}}$ durch

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

so erhalten wir für die Folgenglieder b_k die Rekurrenz-Gleichung

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

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:

$$\begin{aligned} \alpha^{\log_\beta(n)} &= n^{\log_\beta(\alpha)} & | \log_\beta(\cdot) \\ \Leftrightarrow \log_\beta(\alpha^{\log_\beta(n)}) &= \log_\beta(n^{\log_\beta(\alpha)}) \\ \Leftrightarrow \log_\beta(n) \cdot \log_\beta(\alpha) &= \log_\beta(n) \cdot \log_\beta(\alpha) \quad \text{wegen } \log_b(x^y) = y \cdot \log_b(x) \end{aligned}$$

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:

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

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

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

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

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

Das liefert

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

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 $k = 1, 2, 3$:

$$\begin{aligned} b_1 &= \alpha \cdot b_0 + \alpha^1 \\ &= \alpha \cdot 0 + \alpha \\ &= 1 \cdot \alpha^1 \\ b_2 &= \alpha \cdot b_1 + \alpha^1 \\ &= \alpha \cdot 1 \cdot \alpha^1 + \alpha^2 \\ &= 2 \cdot \alpha^2 \\ b_3 &= \alpha \cdot b_2 + \alpha^2 \\ &= \alpha \cdot 2 \cdot \alpha^2 + \alpha^3 \\ &= 3 \cdot \alpha^3 \end{aligned}$$

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.: $k \mapsto k + 1$

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

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

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

Wir überlegen uns, wie die Ungleichung

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

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

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

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

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

Wir multiplizieren nun die Ungleichung (3.12) mit α und Ungleichung (3.11) mit γ und erhalten die Ungleichungen

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

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

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

Iterieren wir diesen Prozess, so sehen wir, dass

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

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

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

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 β^k wieder durch n , so sehen wir, dass

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

gilt und daraus folgt sofort

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

□

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

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

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

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

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

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

□

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 α und β 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

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

gilt. □

Aufgabe 6: Benutzen Sie den Hauptsatz der Laufzeit-Funktionen um das asymptotische Wachstum der Folgen $(a_n)_{n \in \mathbb{N}}$, $(b_n)_{n \in \mathbb{N}}$ und $(c_n)_{n \in \mathbb{N}}$ 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 α und β 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. \square

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. \square

Kapitel 4

Der Hoare-Kalkül

In diesem Kapitel stellen wir den *Hoare-Kalkül* vor, mit dessen Hilfe sich die Korrektheit nicht-rekursiver 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

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

und sagen, dass P die Spezifikation „wenn vorher F , dann nachher G “ erfüllt. Die Spezifikation

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

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

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

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

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

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

3. Wir betrachten wieder als Programm-Fragment „ $x = x + 1$;“ und wählen als Vorbedingung diesmal die Formel $\text{prim}(x)$, die ausdrückt, dass x eine Primzahl ist. Das entsprechende Hoare-Tripel ist diesmal

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

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

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

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

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

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

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

Hier bezeichnet $F\sigma$ die Anwendung der Substitution σ 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 σ 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 “ \equiv ” benutzt, um die syntaktische Gleichheit von Formeln ausdrücken zu können, die ihrerseits das Gleichheitszeichen “ $=$ ” enthalten.) Damit lautet die Spezifikation also

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

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

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

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

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

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 *prim*(x) genau dann wahr ist, wenn x eine Primzahlen ist. Dann gilt

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

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

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

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

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

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

$$\{F\} \text{ P } \{G\}$$

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

$$G \rightarrow H,$$

so erfüllt das Programm-Fragment P erst recht die Spezifikation

$$\{F\} \text{ P } \{H\},$$

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:

$$\frac{\{F\} \text{ P } \{G\}, \quad G \rightarrow H}{\{F\} \text{ P } \{H\}}.$$

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 Zusammengesetzte Anweisungen

Haben die Programm-Fragmente P und Q die Spezifikationen

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

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

$$\{F_1\} \text{ P;Q } \{G_2\}$$

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:

$$\frac{\{F_1\} \text{ P } \{G_1\}, \quad G_1 \rightarrow F_2, \quad \{F_2\} \text{ Q } \{G_2\}}{\{F_1\} \text{ P;Q } \{G_2\}}$$

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

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

```

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

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

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

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

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

$$\{x = a \wedge y = b\} \quad x := x - y; \quad \{x + y = a \wedge y = b\}. \quad (4.1)$$

Die zweite Zuweisung liefert das Hoare-Tripel

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

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

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

Vereinfachen wir noch die Nachbedingung, so haben wir

$$\{x + y = a \wedge y = b\} \quad y := y + x; \quad \{y = a \wedge y - x = b\} \quad (4.2)$$

gezeigt. Jetzt betrachten wir die letzte Zuweisung. Wir erhalten

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

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

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

Vereinfachung der Nachbedingung liefert schließlich

$$\{y = a \wedge y - x = b\} \quad x := y - x; \quad \{y = a \wedge x = b\}. \quad (4.3)$$

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

$$\{x = a \wedge y = b\} \quad x := x + y; \quad y := y + x; \quad x := y - x; \quad \{y = a \wedge x = b\}. \quad (4.4)$$

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:

$$\frac{\{F \wedge B\} \quad P \quad \{G\}, \quad \{F \wedge \neg B\} \quad Q \quad \{G\}}{\{F\} \quad \text{if } (B) \text{ P else Q } \{G\}} \quad (4.5)$$

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

$$\{F \wedge B\} \ P \ \{G_1\} \quad \text{und} \quad \{F \wedge \neg B\} \ Q \ \{G_2\}, \quad (4.6)$$

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

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

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

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

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

$$F = (x = a \wedge y = b)$$

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

$$G = (z = \min(a, b))$$

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

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

Analog erhalten wir für die zweite Zuweisung

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

Nun gilt einerseits

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

und andererseits

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

Durch Anwendung der Abschwächungs-Regel sehen wir also, dass

$$\begin{aligned} &\{x = a \wedge y = b \wedge x < y\} \ z := x; \ \{z = \min(a, b)\} \quad \text{und} \\ &\{x = a \wedge y = b \wedge x \geq y\} \ z := y; \ \{z = \min(a, b)\} \end{aligned}$$

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

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

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:

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

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 35 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

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

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

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

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 $\text{ggt}(x, y) = \text{ggt}(a, b)$.

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

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

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:

$$\begin{aligned} & (I \wedge x \neq y \wedge x < y)\sigma \\ \Leftrightarrow & (I \wedge x < y)\sigma \quad \text{denn aus } x < y \text{ folgt } x \neq y \\ \Leftrightarrow & (x > 0 \wedge y > 0 \wedge \text{ggt}(x, y) = \text{ggt}(a, b) \wedge x < y)[y \mapsto y + x] \\ \Leftrightarrow & x > 0 \wedge y + x > 0 \wedge \text{ggt}(x, y + x) = \text{ggt}(a, b) \wedge x < y + x \\ \Leftrightarrow & x > 0 \wedge y + x > 0 \wedge \text{ggt}(x, y) = \text{ggt}(a, b) \wedge 0 < y \end{aligned}$$

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

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

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

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

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

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

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

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

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

$$\begin{aligned} & (I \wedge x \neq y \wedge x \geq y)\sigma \\ \Leftrightarrow & (I \wedge x > y)\sigma \\ \Leftrightarrow & (x > 0 \wedge y > 0 \wedge \text{ggt}(x, y) = \text{ggt}(a, b) \wedge x > y)[x \mapsto x + y] \\ \Leftrightarrow & x + y > 0 \wedge y > 0 \wedge \text{ggt}(x + y, y) = \text{ggt}(a, b) \wedge x + y > y \\ \Leftrightarrow & x + y > 0 \wedge y > 0 \wedge \text{ggt}(x, y) = \text{ggt}(a, b) \wedge x > 0 \end{aligned}$$

Aus der letzten Formel folgt nun

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

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

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

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

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

Mit der Verifikations-Regel für Schleifen folgt daraus

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

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

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

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

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

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: C-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

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

gegeben.

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

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

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 " $\mathbf{r} = 1;$ "

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

Damit finden wir für die erste Anweisung das Hoare-Tripel

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

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 " $\mathbf{r} = \mathbf{r} * \mathbf{x};$ " erhalten wir dann die Bedingung $I \wedge y > 0 \wedge y \% 2 = 1$. Das liefert das Hoare-Tripel

$$\begin{aligned} &\{r \cdot x^y = a^b \wedge y > 0 \wedge y \% 2 = 1\} \\ &\quad \mathbf{r} := \mathbf{r} * \mathbf{x}; \\ &\{ (r \cdot x^y = a^b \wedge y > 0 \wedge y \% 2 = 1) [r \mapsto r/x] \} \end{aligned}$$

Die Nachbedingung vereinfacht sich zu

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

und das ist dasselbe 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

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

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:

$$\begin{aligned} &\{r \cdot x^y = a^b \wedge y > 0\} \\ &\quad \mathbf{if} \ (y \% 2) == 1 \ \{ \ \mathbf{r} := \mathbf{r} * \mathbf{x}; \} \\ &\quad \{r \cdot x^{2 \cdot (y/2)} = a^b\} \end{aligned}$$

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

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

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

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

gefunden.

5. Die letzte Zuweisung “ $y := y / 2;$ ” liefert nun das Hoare-Tripel

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

Hier haben wir die allgemeine Zuweisungs-Regel

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

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

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

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:

$$\begin{aligned} &\{r \cdot x^y = a^b\} \\ &\quad \text{while } (y > 0) \{ \\ &\quad \quad \text{if } (y \% 2 == 1) \{ r = r * x; \} \\ &\quad \quad x = x * x; \\ &\quad \quad y = y / 2; \\ &\quad \} \\ &\{r \cdot x^y = a^b \wedge \neg y > 0\} \end{aligned}$$

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

$$y = y/2$$

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. \square

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 derselben 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 `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 anhand 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 einem 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 dasselbe 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 r_n , auf der linken Seite hat sie dann anschließend den Wert r_{n+1} und am Ende der Schleife in Zeile 11 hat die Variable r den Wert 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

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

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

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

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

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

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

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 anhand 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.

5.1 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 Verfügung stellt,
4. Ts ist eine Menge von *Typ-Spezifikation*, die zu jedem Funktions-Zeichen $f \in Fz$ eine *Typ-Spezifikation* der Form

$$f : T_1 \times \cdots \times T_n \rightarrow S.$$

enthält. Dabei sind T_1, \dots, 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 $f : T_1 \times \cdots \times T_n \rightarrow S$ drückt aus, dass die Funktion f in der Form

$$f(t_1, \dots, t_n)$$

aufgerufen wird und dass für $i = 1, \dots, n$ 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 $\{Stack, push, pop, top, isEmpty\}$.
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 Funktionen 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) $push : Stack \times Element \rightarrow Stack$

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) $pop : Stack \rightarrow Stack$

Der Aufruf $S.pop()$ entfernt das oberste Element von dem Stack S .

(d) $top : Stack \rightarrow Element$

Der Aufruf $S.top()$ liefert das auf dem Stack S zuoberst liegende Element.

(e) $isEmpty : Stack \rightarrow \mathbb{B}$

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 Ω den undefinierten Wert. Der Aufruf *Stack()* liefert zunächst einen leeren Stack. Das Axiom drückt also aus, dass 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.

$$5. Stack().isEmpty() = \text{true}$$

Erzeugen wir mit $Stack()$ einen neuen Stack, so ist dieser zunächst leer.

$$6. S.push(x).isEmpty() = \text{false}$$

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.

5.2 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 5.1 auf Seite 48 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, dass 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, d.h. kopiert werden können.

2. Zeile 3 enthält die Typ-Spezifikationen der Method **push**. Oben hatten wir diese Typ-Spezifikation als

```

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 5.1: darstellung des ADT-Stack in *Java*.

$\text{push} : \text{Stack} \times \text{Element} \rightarrow \text{Stack}$

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 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 anhand 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 verketteten 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!

5.3 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 5.2 auf Seite 51 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 5.2: 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 5.3 auf Seite 52 zeigt ein sehr einfaches Programm, in dem Stacks benutzt werden. Wir legen nacheinander die Zahlen $0, 1, \dots, 32$ 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 5.3: Test der Stack-Implementierung.

5.4 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 5.4 auf Seite 53 zeigt die Implementierung.

Um eine verkettete 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. Weiterhin 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 Paares 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 5.4: 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 5.3 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, dass 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.

5.5 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 $x \wedge y$ 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 arithmetischen Ausdrücke *ArithExpr* induktiv definiert:

1. Jede Zahl $n \in \mathbb{N}$ ist ein arithmetischer Ausdruck:

$$n \in \text{ArithExpr} \quad \text{f.a. } n \in \mathbb{N}.$$

2. Sind s und t arithmetische Ausdrücke, so sind auch $s + t$, $s - t$, $s * t$, s / t , $s \% t$ und $s \wedge t$ arithmetische Ausdrücke:

- (a) $s \in \text{ArithExpr} \wedge t \in \text{ArithExpr} \rightarrow s + t \in \text{ArithExpr}$,
- (b) $s \in \text{ArithExpr} \wedge t \in \text{ArithExpr} \rightarrow s - t \in \text{ArithExpr}$,
- (c) $s \in \text{ArithExpr} \wedge t \in \text{ArithExpr} \rightarrow s * t \in \text{ArithExpr}$,
- (d) $s \in \text{ArithExpr} \wedge t \in \text{ArithExpr} \rightarrow s / t \in \text{ArithExpr}$,
- (e) $s \in \text{ArithExpr} \wedge t \in \text{ArithExpr} \rightarrow s \% t \in \text{ArithExpr}$,
- (f) $s \in \text{ArithExpr} \wedge t \in \text{ArithExpr} \rightarrow s \wedge t \in \text{ArithExpr}$,
- (g) $s \in \text{ArithExpr} \rightarrow (s) \in \text{ArithExpr}$.

Haben wir nun einen String gegeben, der einen arithmetischen Ausdruck repräsentiert, als Beispiel betrachten wir den String

$$4 + 3 * 2 \wedge 2 \wedge 3,$$

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.

5.5.1 Ein einführendes Beispiel

Wir demonstrieren das Verfahren, mit dem wir arithmetische Ausdrücke auswerten, zunächst anhand 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.

5.5.2 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-Strukturen 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 5.5, 5.6 und 5.7 auf den Seiten 58, 60 und 61 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 5.1 zeigt die Präzedenzen der von uns verwendeten Operatoren.

Ist die Präzedenz des Operators op1 höher als die Präzedenz des Operators op2, so bindet

```
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 5.5: Die Klasse Calculator

| Operator | Präzedenz |
|--------------|-----------|
| "+", "-" | 1 |
| "*", "/", "% | 2 |
| "^", "** | 3 |

Tabelle 5.1: Präzedenzen der Operatoren.

`op1` stärker als `op2` und wird daher vor diesem ausgewertet. Ist die Präzedenz des Operators `op1` 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

$2 \wedge 3 \wedge 4$ interpretiert als $2 \wedge (3 \wedge 4)$,

denn wir sagen, dass der Operator " \wedge " *rechts-assoziativ* ist.

Die Operatoren "+", "-", "*", "/" und "%" sind alle links-assoziativ. Hier wird als zunächst `op1` ausgewertet. Der Operator " \wedge " ist rechts-assoziativ. Ist der oberste Operator auf dem Operator-Stack also " \wedge " und wird dann nochmal der Operator " \wedge " 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 5.5 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 `precedence()` in den Zeilen 21 – 34 ergibt sich unmittelbar aus der Tabelle 5.1 auf Seite 59. 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 5.6 auf Seite 60 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 5.6: Die Klasse `Calculator`

Damit können wir die in Abbildung 5.7 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:

```

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 5.7: Der Konstruktor der Klasse `Calculator`.

- (a) Ist das gelesene Token eine Zahl, so legen wir diese auf den Argument-Stack und lesen das nächste Token.
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 5.8 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.

5.6 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.

```

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 5.8: Die Klasse MyScanner

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.

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 6

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

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

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

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 $\{1\} \not\subseteq \{2\}$ 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:

$$x \leq y \leftrightarrow x < y \vee x = y.$$

Betrachten wir die obigen Beispiele und überlegen uns, in welchen Fällen es möglich ist, eine Liste von Elementen 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. $\forall x, y, z \in M: (x \preceq y \wedge y \preceq z \rightarrow x \preceq z)$. (Transitivität)

Gilt zusätzlich

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

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:

$$\forall i \in \{1, \dots, \#S - 1\}: S(i) \preceq S(i + 1)$$

Hier bezeichnen wir die Länge der Liste S mit $\#S$.

- (b) Die Elemente treten in L und S mit derselben Häufigkeit auf:

$$\forall x \in M: \text{count}(x, L) = \text{count}(x, S).$$

Dabei zählt die Funktion $\text{count}(x, L)$ wie oft das Element x in der Liste L auftritt:

$$\text{count}(x, L) := \#\{i \in \{1, \dots, \#L\} \mid L(i) = x\}.$$

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.

6.1 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:

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

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.

$$\text{sort}([x] + R) = \text{insert}(x, \text{sort}(R))$$

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 $[x]$:

$$\text{insert}(x, []) = [x].$$

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:

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

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

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

Dieser Algorithmus lässt sich leicht in *Java* umsetzen. Abbildung 6.1 zeigt das resultierende Programm.

1. In Zeile 4 deklarieren wir die Member-Variable `mList` als Objekt vom Typ `LinkedList<Double>`. Die Klasse `LinkedList` repräsentiert verkettete Listen und diese Datenstruktur ist zur Umsetzung der rekursiven Gleichungen am besten geeignet.
2. In Zeile 6 definieren wir den Konstruktor der Klasse `InsertionSort`. Dieser Konstruktor hat die Aufgabe, die Member-Variable `mList` zu initialisieren.
3. In Zeile 12 bis 33 implementieren wir die Methoden `sort()` und `insert()`. Dabei benutzen wir die folgenden Methoden der Klasse `LinkedList`:
 - (a) `L.isEmpty()` testet, ob die Liste L leer ist.
 - (b) `L.removeFirst()` entfernt das erste Element aus der Liste L und liefert als Ergebnis das entfernte Element zurück.
 - (c) `L.addFirst(x)` fügt das Element x als erstes Element in die Liste L ein.
 - (d) `L.getFirst()` liefert als Ergebnis das erste Element der Liste L . Die Liste L wird dabei nicht verändert.
4. Zeile 34 definieren wir der Vollständigkeit halber noch eine Methode `main()`, mit der wir die Klasse `InsertionSort` testen können.

6.1.1 Komplexität

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die bei einem Aufruf von “Sortieren durch Einfügen” in Zeile 26 von Abbildung 6.1 auf Seite 67 durchgeführt werden. Dazu berechnen wir zunächst die Anzahl der Aufrufe von “ \leq ”, 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 . Dann haben wir

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

Durch eine einfache Induktion lässt sich sofort nachweisen, 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 Element aus L vergleichen.

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die im schlimmsten Fall beim Aufruf von `sort(L)` für eine Liste der Länge L durchgeführt werden. Wir bezeichnen diese 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

```

1  import java.util.*;
2
3  public class InsertionSort {
4      LinkedList<Double> mList;
5
6      InsertionSort(Double[] a) {
7          mList = new LinkedList<Double>();
8          for (Double x : a) {
9              mList.add(x);
10         }
11     }
12     public void sort() {
13         if (mList.isEmpty()) {
14             return;
15         }
16         Double x = mList.removeFirst();
17         sort();
18         insert(x);
19     }
20     private void insert(Double x) {
21         if (mList.isEmpty()) {
22             mList.addFirst(x);
23             return;
24         }
25         Double y = mList.getFirst();
26         if (x <= y) {
27             mList.addFirst(x);
28         } else {
29             mList.removeFirst(); // remove y
30             insert(x);
31             mList.addFirst(y);
32         }
33     }
34     public static void main(String[] args) {
35         Double[] a = { 3.0, 7.0, 5.0, 2.0, 4.0, 11.0, 1.0 };
36         InsertionSort IS = new InsertionSort(a);
37         System.out.println(IS.mList);
38         IS.sort();
39         System.out.println(IS.mList);
40     }
41 }

```

Abbildung 6.1: Der Algorithmus “Sortieren durch Einfügen”

`sort(R)` aufgerufen. Das liefert den Summanden b_n . Anschließend wird mit `insert(x , Sorted)` 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:

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

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

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

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

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

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 `insert(x, Sorted)` nur ein einziger Vergleich durchgeführt. Die Rekurrenz-Gleichungen für die Anzahl der Vergleiche in `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.

6.2 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:

$$\text{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:

$$L \neq [] \rightarrow \text{sort}(L) = [\min(L)] + \text{sort}(\text{delete}(\min(L), L)).$$

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

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

2. Falls x gleich dem ersten Element der Liste L ist, gibt die Funktion den Rest R zurück:

$$\text{delete}(x, [x] + R) = R.$$

3. Andernfalls wird das Element x rekursiv aus R entfernt:

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

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

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

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

$$\text{min}([x] + R) = \text{min}(x, \text{min}(R)).$$

Dabei ist die zweistellige Minimums-Funktion wie folgt definiert:

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

Die Implementierung dieses Algorithmus in *Java* sehen Sie in Abbildung 6.2 auf Seite 69. Es war nicht notwendig, die Funktion *delete()* zu implementieren, denn die Methode *remove()* der Klasse *LinkedList<T>* leistet das Gleiche wie die oben beschriebene Funktion *delete()*.

```
1  import java.util.*;
2
3  public class SelectionSort {
4      LinkedList<Double> mList;
5
6      SelectionSort(Double[] a) {
7          mList = new LinkedList<Double>();
8          for (Double x : a) {
9              mList.add(x);
10         }
11     }
12     public void sort() {
13         if (mList.isEmpty()) {
14             return;
15         }
16         Double x = min();
17         mList.remove(x);
18         sort();
19         mList.addFirst(x);
20     }
21     private Double min() {
22         Double min = mList.getFirst();
23         for (Double x : mList) {
24             min = Math.min(min, x);
25         }
26         return min;
27     }
28 }
```

Abbildung 6.2: Der Algorithmus “*Sortieren durch Auswahl*”

6.2.1 Komplexität

Um die Komplexität von “*Sortieren durch Auswahl*” analysieren zu können, müssen wir zunächst die Anzahl der Vergleiche, die bei der Berechnung von $\min(L)$ durchgeführt werden, bestimmen. Es gilt

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

Also wird bei der Berechnung von $\min(L)$ 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 “ \leq ”. Bezeichnen wir die Anzahl der Vergleiche beim Aufruf von $\text{sort}(L)$ 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

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

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 $\frac{1}{2}n * (n-1)$, beim “*Sortieren durch Auswahl*” ist Anzahl der Vergleiche immer $\frac{1}{2}n * (n-1)$. 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*”.

6.2.2 Eine feldbasierte Implementierung

In der Anwendung besteht die Aufgabe häufig darin, ein Feld von Daten zu sortieren. Bisher haben wir nur Listen sortiert. Daher präsentieren wir zum Abschluss unserer Diskussion des Algorithmus “*Sortieren durch Auswahl*” noch eine Implementierung dieses Algorithmus’ in der Sprache *Java*, in der die zu sortierenden Daten in einem Feld an Stelle einer Liste vorliegen. Abbildung 6.4 auf Seite 71 zeigt diese Implementierung. Die Klasse `MinSortAlgorithm` implementiert die Schnittstelle

`SortingAlgorithm`,

die in Abbildung 6.3 gezeigt wird. Diese Schnittstelle schreibt die Implementierung einer einzigen Methode vor, der Methode

```
void sort();
```

Die Einhaltung dieser Schnittstelle ist notwendig, um das Programm später in eine Umgebung zur Algorithmen-Visualisierung einbinden zu können.

```
1 public interface SortingAlgorithm {
2     public void sort();
3 }
```

Abbildung 6.3: Das Interface “`SortingAlgorithm`”

In Zeile 3 der Implementierung der Klasse `MinSortAlgorithm` (Abbildung 6.4) definieren wir das zu sortierende Feld und in Zeile 4 definieren wir ein Objekt vom Typ `Comparator`. Dieses Objekt hat eine Methode

```
int compare(Double x, Double y);
```

mit deren Hilfe wir zwei Zahlen vergleichen können. Es gilt

$$\text{compare}(x, y) = \begin{cases} -1 & \text{falls } x < y; \\ 0 & \text{falls } x = y; \\ 1 & \text{falls } x > y. \end{cases}$$

Sowohl das zu sortierende Feld als auch der Komparator werden dem Konstruktor in Zeile 6 als Argument mitgegeben. Der Konstruktor initialisiert mit diesen Argumenten die entsprechenden Member-Variablen.

```

1  public class MinSortAlgorithm implements SortingAlgorithm
2  {
3      private Double[]      mArray;
4      private Comparator<Double> mComparator;
5
6      MinSortAlgorithm(Double[] array, Comparator<Double> comparator) {
7          mArray      = array;
8          mComparator = comparator;
9      }
10     public void sort() { sort(0); }
11
12     private void sort(int i) {
13         if (i == mArray.length)
14             return;
15         int minIndex = minIndex(i);
16         swap(i, minIndex);
17         sort(i + 1);
18     }
19     private int minIndex(int first) {
20         int minIndex = first;
21         for (int i = first + 1; i < mArray.length; ++i) {
22             if (mComparator.compare(mArray[minIndex], mArray[i]) > 0) {
23                 minIndex = i;
24             }
25         }
26         return minIndex;
27     }
28     private void swap(int i, int j) {
29         if (i == j) return;
30         Double temp = mArray[i];
31         mArray[i]   = mArray[j];
32         mArray[j]   = temp;
33     }
34 }

```

Abbildung 6.4: Der Algorithmus “Sortieren durch Auswahl”

Die Implementierung der Methode `sort()` in Zeile 10 ist trivial, denn die ganze Arbeit wird in der Hilfs-Methode `sort(int)` abgewickelt. Für eine natürliche Zahl i mit $i < \text{mArray.length}$

sortiert der Aufruf `sort(i)` den Teil des Feldes, dessen Indizes größer oder gleich dem Argument i sind, nach dem Aufruf `sort(i)` ist also die Liste

`[mArray[i], mArray[i + 1], ..., mArray[mArray.length - 1]]`

sortiert. Die Implementierung der Methode `sort(int)` ist rekursiv:

1. Falls $i = \text{mArray.length}$ ist, dann ist der zu sortierende Teil des Feldes leer und folglich ist nichts zu tun.
2. Andernfalls berechnet der Aufruf `minIndex(first)` in Zeile 15 einen Index `minIndex` so, dass das Element `mArray[minIndex]` in dem zu sortierenden Teil der Liste minimal ist, es gilt also

$$\forall j \in \{ \text{first}, \text{first}+1, \dots, \text{mArray.length}-1 \}: \text{mArray}[\text{minIndex}] \leq \text{mArray}[j].$$

Anschließend wird das Element, das an der Stelle `minIndex` steht mit dem Element an der Stelle `first` vertauscht. Damit steht an der Stelle `first` jetzt ein kleinstes Element der Liste

`[mArray[first], mArray[first + 1], ..., mArray[mArray.length - 1]]`.

Sortieren wir danach rekursiv die Liste

`[mArray[first + 1], ..., mArray[mArray.length - 1]]`,

so ist anschließend auch die Liste

`[mArray[first], mArray[first + 1], ..., mArray[mArray.length - 1]]`

sortiert, denn `mArray[first]` ist ja ein minimales Element.

Die Implementierung der Methode `minIndex(first)` berechnet den Index eines kleinsten Elements iterativ. Zunächst wird `minIndex` mit dem Index `first` initialisiert. Anschließend wird eine Schleife durchlaufen. Falls ein Index i gefunden wird, so dass das Element `mArray[i]` kleiner als das Element `mArray[minIndex]` ist, dann wird `minIndex` auf i gesetzt. Dadurch ist gewährleistet dass `minIndex` am Ende der Schleife tatsächlich auf das kleinste Element zeigt.

6.3 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 wird L zurück gegeben:

$$\#L < 2 \rightarrow \text{sort}(L) = L.$$

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:

$$\#L \geq 2 \rightarrow \text{sort}(L) = \text{merge}\left(\text{sort}(\text{split}_1(L)), \text{sort}(\text{split}_2(L))\right)$$

Hier bezeichnen `split1` und `split2` 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 6.5 zeigt die Umsetzung dieser sortierten Gleichungen in ein *Java*-Programm, das eine verkettete Liste sortiert. Die beiden Funktionen `split1` und `split2` haben wir dabei zu einer Funktion `split` zusammengefasst, die zwei Listen zurück liefert. Ein Aufruf der Form

`split(L, L1, L2)`

verteilt die Elemente der Liste L auf die beiden Listen L_1 und L_2 . Damit das funktioniert, müssen

die Variablen L_1 und L_2 zu Beginn des Aufrufs auf leere Listen verweisen. Damit können wir die Details des *Java*-Programms in Abbildung 6.5 verstehen:

```

1  public LinkedList<Double> sort(LinkedList<Double> list) {
2      if (list.size() < 2) {
3          return list;
4      }
5      LinkedList<Double> first  = new LinkedList<Double>();
6      LinkedList<Double> second = new LinkedList<Double>();
7      split(list, first, second);
8      LinkedList<Double> firstSorted  = sort(first);
9      LinkedList<Double> secondSorted = sort(second);
10     return merge(firstSorted, secondSorted);
11 }

```

Abbildung 6.5: Die Methode `sort`.

1. Wenn die zu sortierende Liste aus weniger als zwei Elementen besteht, dann ist diese Liste bereits sortiert und wir können diese Liste in Zeile 3 unverändert zurück geben.
2. In den Zeilen 5 und 6 legen wir zwei leere Listen `first` und `second` an.
3. Der Aufruf von `split` verteilt die Elemente der zu sortierenden Liste auf die beiden Listen `first` und `second`.
4. In den Zeilen 8 und 9 werden diese Listen durch einen rekursiven Aufruf der Methode `sort` sortiert.
5. Die sortierten Teillisten werden dann in Zeile 10 durch den Aufruf der Methode `merge` zu einer sortierten Liste zusammengefasst.

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

1. Falls L leer ist, produziert `split(L)` zwei leere Listen:

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

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

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

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:

$$\text{split}(R) = [R_1, R_2] \rightarrow \text{split}([x, y] + R) = [[x] + R_1, [y] + R_2].$$

Abbildung 6.6 auf Seite 74 zeigt die Umsetzung dieser bedingten Gleichungen in *Java*.

1. Falls die Liste, deren Elemente auf zwei Listen verteilt werden sollen, leer ist, so ist nichts weiter zu tun, denn wir setzen voraus, dass die Listen `first` und `second` beim Aufruf der Methode `split` leer sind.
2. Andernfalls entfernen wir zunächst das erste Element aus der Liste `list` und speichern es in der Variablen x . Falls die Liste danach leer ist, fügen wir das Element x in die Liste `first` ein und beenden den Methoden-Aufruf.

```

1  public void split(LinkedList<Double> list,
2                      LinkedList<Double> first, LinkedList<Double> second)
3  {
4      if (list.size() == 0) {
5          return;
6      }
7      Double x = list.removeFirst();
8      if (list.size() == 0) {
9          first.addFirst(x);
10         return;
11     }
12     Double y = list.removeFirst();
13     split(list, first, second);
14     first.addFirst(x);
15     second.addFirst(y);
16 }

```

Abbildung 6.6: Die Methode `split`.

3. Wenn der Kontrollfluss in Zeile 12 ankommt, dann muss die Liste `list` beim Aufruf wenigstens zwei Elemente gehabt haben. Wir entfernen nun das zweite dieser beiden Elemente und speichern es in der Variablen `y`.
4. Anschließend teilen wir das, was jetzt noch von der Liste `list` übrig ist, durch den rekursiven Aufruf von `split` auf die Listen `first` und `second` auf.
5. Zum Abschluss fügen wir das Element `x` in die Liste `first` ein und das Element `y` schieben wir in die Liste `second`.

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 :

$$\text{merge}([], L_2) = L_2.$$

2. Falls die Liste L_2 leer ist, ist das Ergebnis L_1 :

$$\text{merge}(L_1, []) = L_1.$$

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:

$$x \preceq y \rightarrow \text{merge}([x] + R_1, [y] + R_2) = [x] + \text{merge}(R_1, [y] + R_2).$$

- (b) $\neg x \preceq y$.

In diesem Fall mischen wir L_1 und R_2 und setzen y an den Anfang dieser Liste:

$$\neg x \preceq y \rightarrow \text{merge}([x] + R_1, [y] + R_2) = [y] + \text{merge}([x] + R_1, R_2).$$

Abbildung 6.7 auf Seite 75 zeigt die Umsetzung dieses Algorithmus in *Java*.

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

```

1  public LinkedList<Double> merge(LinkedList<Double> first,
2                                LinkedList<Double> second)
3  {
4      if (first.size() == 0) { return second; }
5      if (second.size() == 0) { return first; }
6      LinkedList<Double> result;
7      Double x = first.getFirst();
8      Double y = second.getFirst();
9      if (x < y) {
10         first.removeFirst();
11         result = merge(first, second);
12         result.addFirst(x);
13     } else {
14         second.removeFirst();
15         result = merge(first, second);
16         result.addFirst(y);
17     }
18     return result;
19 }

```

Abbildung 6.7: Die Methode `merge`.

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. Anschließend 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.

Zum Abschluss zeigen wir in Abbildung 6.8, wie die eben diskutierten Methoden zusammen spielen.

6.3.1 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

$$\text{cmpCount} : \text{List}(M) \times \text{List}(M) \rightarrow \mathbb{N}$$

so dass für zwei Listen L_1 und L_2 der Term $\text{cmpCount}(L_1, L_2)$ die Anzahl Vergleiche angibt, die bei Berechnung von $\text{merge}(L_1, L_2)$ erforderlich sind. Wir behaupten, dass für beliebige Listen L_1 und L_2

$$\text{cmpCount}(L_1, L_2) \leq \#L_1 + \#L_2$$

gilt. Für eine Liste L bezeichnet dabei $\#L$ die Anzahl der Elemente der Liste. Wir führen den Beweis durch Induktion nach der Summe $\#L_1 + \#L_2$.

```

1  import java.util.*;
2
3  public class MergeSort
4  {
5      public LinkedList<Double> sort(LinkedList<Double> list) {
6          ...
7      }
8      public void split(LinkedList<Double> list,
9                      LinkedList<Double> first, LinkedList<Double> second)
10     {
11         ...
12     }
13     public LinkedList<Double> merge(LinkedList<Double> first,
14                                     LinkedList<Double> second)
15     {
16         ...
17     }
18     public static void main(String[] args) {
19         Double[] a = { 3.0, 7.0, 5.0, 2.0, 4.0, 2.0, 11.0, 1.0 };
20         LinkedList<Double> list = new LinkedList<Double>();
21         for (Double x : a) {
22             list.addFirst(x);
23         }
24         MergeSort MS = new MergeSort();
25         System.out.println(list);
26         list = MS.sort(list);
27         System.out.println(list);
28     }
29 }

```

Abbildung 6.8: Die Klasse MergeSort

I.A.: $\#L_1 + \#L_2 = 0$.

Dann müssen L_1 und L_2 leer sein und somit ist beim Aufruf von `merge(L_1, L_2)` kein Vergleich erforderlich. Also gilt

$$\text{cmpCount}(L_1, L_2) = 0 \leq 0 = \#L_1 + \#L_2.$$

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

$$\text{cmpCount}(L_1, L_2) = 0 \leq \#L_1 + \#L_2.$$

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

$$\text{merge}([x] + R_1, [y] + R_2) = [x] + \text{merge}(R_1, [y] + R_2).$$

Also haben wir:

$$\text{cmpCount}(L_1, L_2) = 1 + \text{cmpCount}(R_1, L_2) \stackrel{IV}{\leq} 1 + \#R_1 + \#L_2 = \#L_1 + \#L_2.$$

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

Aufgabe 8: Überlegen Sie, wie die Listen L_1 und L_2 aussehen müssen, damit

$$\text{cmpCount}(L_1, L_2) = \#L_1 + \#L_2$$

gilt.

Wir wollen nun die Komplexität des Merge-Sort-Algorithmus im schlechtesten Fall berechnen und bezeichnen dazu die Anzahl der Vergleiche, die beim Aufruf von $\text{sort}(L)$ für eine Liste L der Länge n schlimmstenfalls durchgeführt werden müssen mit a_n . Zur Vereinfachung nehmen wir an, dass n die Form

$$n = 2^k \quad \text{für ein } k \in \mathbb{N}$$

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 $\text{sort}(L)$ 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

$$b_{k+1} = 2 \cdot b_k + 2^{k+1}, \quad (1)$$

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

$$b_{k+2} = 2 \cdot b_{k+1} + 2^{k+2}. \quad (2)$$

Wir multiplizieren Gleichung (1) mit dem Faktor 2 und subtrahieren die erhaltene Gleichung von Gleichung (2). Dann erhalten wir

$$b_{k+2} - 2 \cdot b_{k+1} = 2 \cdot b_{k+1} - 4 \cdot b_k. \quad (3)$$

Diese Gleichung vereinfachen wir zu

$$b_{k+2} = 4 \cdot b_{k+1} - 4 \cdot b_k. \quad (4)$$

Diese Gleichung ist eine homogene lineare Rekurrenz-Gleichung 2. Ordnung. Das charakteristische Polynom der zugehörigen homogenen Rekurrenz-Gleichung lautet

$$\chi(x) = x^2 - 4 \cdot x + 4 = (x - 2)^2.$$

Weil das charakteristische Polynom an der Stelle $x = 2$ eine doppelte Null-Stelle hat, lautet die allgemeine Lösung

$$b_k = \alpha \cdot 2^k + \beta \cdot k \cdot 2^k. \quad (5)$$

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.

6.3.2 Eine feldbasierte Implementierung

Abbildung 6.9 auf Seite 79 zeigt eine feldbasierte Implementierung des Algorithmus “*Sortieren durch Mischen*”. Die Klasse `MergeSortAlgorithm` ist von der in der Übung diskutierten Klasse `SortingAlgorithm` abgeleitet. Wir diskutieren zunächst die Member-Variablen.

1. Das zu sortierende Feld wird in der Member-Variablen `mArray` abgelegt. Diese Member-Variable ist in der Oberklasse `SortingAlgorithm` definiert und wird von dieser Klasse geerbt.
2. `mAux` ist ein Hilfsfeld, das wir während der Durchführung des Algorithmus benötigen um Werte zwischenzuspeichern.
3. Der Komparator, den wir zum Vergleich zweier Elemente benötigen, wird in der Variablen `mComparator` abgespeichert. Diese Variable ist ebenfalls in der Oberklasse `SortingAlgorithm` definiert.

Der Komparator stellt die Methode

```
mComparator.compare(Element x, Element y)
```

zur Verfügung, mit deren Hilfe wir zwei Elements x und y vergleichen können.

Der Konstruktor in Zeile 7 erwartet als Argumente das zu sortierende Feld und den Komparator zum Vergleichen zweier Elemente. Er speichert diese Argumente durch den Aufruf des Konstruktors der Oberklasse in den Member-Variablen der Oberklasse ab und legt gleichzeitig das Hilfsfeld `mAux` an.

Der Algorithmus “*Sortieren durch Mischen*” `sort()` wird in der Methode `mergeSort` implementiert. Diese Methode erhält die Argumente `start` und `end`, die den Bereich des Feldes eingrenzen, der zu sortieren ist: Der Aufruf `mergeSort(start, end)` sortiert nur der Bereich

```
mArray[start], ..., mArray[end-1].
```

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 Feldes `mArray` 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 19 und 20 rekursiv sortieren.

Anschließend rufen wir in Zeile 21 die Methode `merge` aus, die die beiden sortierten Felder zu einem sortierten Feld zusammenfasst. Diese Methode ist in den Zeilen 22 — 42 implementiert. Die Methode erhält 3 Argumente: Die Parameter `start`, `middle` und `end` spezifizieren die beiden Teilfelder, die zu mischen sind. Das erste Teilfeld besteht aus den Elementen

```
mArray[start], ..., mArray[middle-1],
```

das zweite Teilfeld besteht aus den Elementen

```
mArray[middle], ..., mArray[end-1].
```

```

1  import java.util.*;
2
3  public class MergeSortAlgorithm extends SortingAlgorithm
4  {
5      private Double[] mAux;
6
7      MergeSortAlgorithm(Double[] array, Comparator<Double> comparator) {
8          super(array, comparator);
9          mAux = (Double[]) new Object[mArray.length];
10     }
11     public void sort() {
12         mergeSort(0, mArray.length);
13     }
14     private void mergeSort(int start, int end) {
15         if (end - start < 2)
16             return;
17         int middle = (start + end) / 2;
18         mergeSort( start, middle );
19         mergeSort( middle, end );
20         merge( start, middle, end );
21     }
22     private void merge(int start, int middle, int end) {
23         for (int i = start; i < end; ++i) {
24             mAux[i] = mArray[i];
25         }
26         int idx1 = start;
27         int idx2 = middle;
28         int i = start;
29         while (idx1 < middle && idx2 < end) {
30             if (mComparator.compare(mAux[idx1], mAux[idx2]) <= 0) {
31                 mArray[i++] = mAux[idx1++];
32             } else {
33                 mArray[i++] = mAux[idx2++];
34             }
35         }
36         while (idx1 < middle) {
37             mArray[i++] = mAux[idx1++];
38         }
39         while (idx2 < end) {
40             mArray[i++] = mAux[idx2++];
41         }
42     }
43 }

```

Abbildung 6.9: Die Klasse MergeSortAlgorithm.

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 24 in das Hilfs-Feld `mAux` kopiert.
2. Anschließend definieren wir drei Indizes:

- (a) `idx1` zeigt auf das nächste zu untersuchende Element im ersten Teilfeld des Arrays `mAux`.
 - (b) `idx2` zeigt auf das nächste zu untersuchende Element im zweiten Teilfeld des Arrays `mAux`.
 - (c) `i` gibt die Position im Ergebnis-Feld `mArray` an, in die das nächste Element geschrieben wird.
3. Solange weder das erste noch das zweite Teilfeld des Arrays `mAux` 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 bei diesem Vergleich 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 36 — 38 wird aktiv, wenn das zweite Teilfeld zuerst erschöpft wird. Dann werden dort die verbleibenden Elemente des ersten Teilfeldes in das Feld `mArray` kopiert. Ist umgekehrt das erste Teilfeld zuerst erschöpft, dann werden in Zeile 39 — 41 die verbleibenden Elemente des zweiten Teilfeldes in das Feld `mArray` kopiert.

6.3.3 Eine nicht-rekursive Implementierung von *Sortieren durch Mischen*

```

1  public void sort() {
2      mergeSort();
3  }
4  private void mergeSort() {
5      for (int l = 1; l < mArray.length; l *= 2) {
6          int k;
7          for (k = 0; l * (k + 1) <= mArray.length; k += 2) {
8              merge(l * k, l * (k + 1), Math.min(l * (k + 2), mArray.length));
9          }
10     }
11 }

```

Abbildung 6.10: Eine sequentielle Implementierung des Merge-Sort-Algorithmus

Die in Abbildung 6.9 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 einerseits die lokalen Variablen der Funktion auf dem Stack gesichert werden und andererseits müssen die Argumente, mit denen die Funktion aufgerufen wird, auf den Stack gelegt werden. Wir zeigen daher, wie sich der Merge-Sort-Algorithmus sequentiell implementieren lässt. Abbildung 6.10 auf Seite 80 zeigt eine solche Implementierung. Statt der rekursiven Aufrufe haben wir hier zwei ineinander geschachtelte `for`-Schleifen. Die Arbeitsweise des Algorithmus wird deutlich, wenn wir die *Invarianten* dieser Schleifen formulieren. Eine solche *Invariante* ist eine logische Formel, die vor jedem Durchlauf der Schleife wahr ist. Dieser Begriff wird am besten durch ein Beispiel klar. Die Invariante der äußeren Schleife besagt, dass alle Teil-Felder der Länge l , die bei einem Index beginnen, der ein Vielfaches von l ist, sortiert sind. Bezeichnen wir das zu sortierende Feld `mArray` jetzt der Kürze halber mit x , so hat ein solches Teilfeld die Form

$$[x[k \cdot l], x[k \cdot l + 1], x[k \cdot l + 2], \dots, x[k \cdot l + (l - 1)]]$$

und wenn n die Länge des Feldes x ist, dann lässt sich die Aussage, dass alle diese Teilfelder sortiert sind, durch die Formel

$$\forall k \in \{0, \dots, n/l\} : \forall j \in \{0, \dots, l-2\} : k \cdot l + j + 1 < n \rightarrow x[k \cdot l + j] \preceq x[k \cdot l + j + 1]$$

beschreiben. Die Bedingung $k \cdot l + j + 1 < n$ ist notwendig um sicherzustellen, dass der Array-Zugriff $x[k \cdot l + j + 1]$ definiert ist. Wenn diese Bedingung am Anfang eines Schleifen-Durchlaufs erfüllt sein soll, dann ist es die Aufgabe des Schleifen-Rumpfs diese Bedingung für den nächsten Wert, den die Schleifen-Variable l annimmt, sicherzustellen. Der erste Wert der Schleifen-Variable l ist 1. Für diesen Wert ist die Schleifen-Invariante trivial denn dann sagt die Invariante nur aus, dass Teilfelder der Länge 1 sortiert sind. In der Schleife wird der Wert von l nach jedem Schleifen-Durchlauf verdoppelt. Es werden jeweils zwei Teilfelder der Länge l genommen und so gemischt, dass das resultierende Teilfeld, dass die Länge $2 \cdot l$ hat, danach sortiert ist. Die innere Schleife in den Zeilen 7 bis 9 mischt für gerade Zahlen k das k -te Teilfeld mit dem $k + 1$ -ten Teilfeld, es werden also die Teilfelder

$$x[k \cdot l], x[k \cdot l + 1], \dots, x[k \cdot l + (l - 1)] \quad \text{und} \\ x[(k + 1) \cdot l], x[(k + 1) \cdot l + 1], \dots, x[(k + 1) \cdot l + (l - 1)]$$

gemischt. Möglicherweise hat das letzte Teilfeld eine Länge, die kleiner als l ist. Daher nehmen wir für das dritte Argument der Methode *merge* das Minimum der beiden Zahlen $l \cdot (k + 2)$ und *mArray.length*.

Der Algorithmus lässt sich noch dadurch verbessern, dass die Ergebnisse abwechselnd in dem Feld **mArray** und **mAux** zurück gegeben werden, denn dann entfällt in der Methode *merge* das Kopieren des Feldes **mArray** in das Hilfs-Feld **mAux**.

6.4 Der *Quick-Sort*-Algorithmus

Der “*Quick-Sort*-Algorithmus” funktioniert nach folgendem Schema:

1. Ist die zu sortierende Liste L leer, so wird L zurück gegeben:

$$\text{sort}([]) = [].$$

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:

$$\text{partition}(x, R) = \langle S, B \rangle.$$

Hierbei gilt dann

- (a) Die Listen S und B enthalten zusammen genau die Elemente der Liste R

$$\forall y \in R : \text{count}(y, S) + \text{count}(y, B) = \text{count}(y, R)$$

- (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 $\text{partition}()$ durch die folgenden Gleichungen beschreiben:

- (a) $\text{partition}(x, []) = \langle [], [] \rangle,$
- (b) $y \preceq x \wedge \text{partition}(x, R) = \langle S, B \rangle \rightarrow \text{partition}(x, [y] + R) = \langle [y] + S, B \rangle$
- (c) $\neg(y \preceq x) \wedge \text{partition}(x, R) = \langle S, B \rangle \rightarrow \text{partition}(x, [y] + R) = \langle S, [y] + B \rangle,$

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:

$$\text{partition}(x, R) = \langle S, B \rangle \rightarrow \text{sort}([x] + R) = \text{sort}(S) + [x] + \text{sort}(B).$$

Abbildung 6.11 zeigt die Umsetzung dieser Überlegung in einem *Java*-Programm.

1. Der *Quick-Sort*-Algorithmus benötigt keine globalen Variablen, daher enthält die Klasse **QuickSort** keine Member-Variablen. Folglich können die Methoden dieser Klasse als **static** deklariert werden.
2. Die Methode **partition** hat die Aufgabe, eine Liste **list** anhand eines gegebenen *Pivot-Elements* **pivot** so in zwei Listen aufzuteilen, dass nach einem Aufruf der Form

$$\text{partition}(\text{pivot}, \text{list}, \text{small}, \text{big})$$

die Liste **small** alle die Elemente aus **list** enthält, die kleiner-gleich **pivot** sind, während **big** die Elemente enthält, die größer als **pivot** sind. Voraussetzung ist dabei, dass die Listen **small** und **big** vor dem Aufruf leer sind.

Falls die Listen **small** und **big** bei dem Aufruf nicht leer sind, so ist die Idee, dass die Elemente aus **list** entsprechend ihrer Größe den beiden Listen **small** und **big** hinzugefügt werden.

- (a) Falls **list** leer ist, so gibt es keine Elemente, die verteilt werden müssen.
- (b) Andernfalls wird das erste Element von **list** aus **list** entfernt und, je nach dem wie groß es im Vergleich zu dem Pivot-Element **pivot** ist, in die Liste **small** oder **big** eingefügt.

```

1  public class QuickSort
2  {
3      private static void partition(Double pivot,
4                                  LinkedList<Double> list,
5                                  LinkedList<Double> small,
6                                  LinkedList<Double> big)
7      {
8          if (list.isEmpty()) { return; }
9          Double x = list.removeFirst();
10         if (x <= pivot) { small.addFirst(x); }
11         else           { big.addFirst(x); }
12         partition(pivot, list, small, big);
13     }
14     public static LinkedList<Double> sort(LinkedList<Double> list) {
15         if (list.isEmpty()) {
16             return list;
17         }
18         Double pivot = list.removeFirst();
19         LinkedList<Double> small = new LinkedList<Double>();
20         LinkedList<Double> big = new LinkedList<Double>();
21         partition(pivot, list, small, big);
22         LinkedList<Double> smallSorted = sort(small);
23         LinkedList<Double> bigSorted = sort(big);
24         smallSorted.add(pivot);
25         smallSorted.addAll(bigSorted);
26         return smallSorted;
27     }
28 }

```

Abbildung 6.11: Der *Quick-Sort*-Algorithmus.

- (c) Schließlich werden die restlichen Elemente von `list` durch einen rekursiven Aufruf der Funktion `partition` auf die Listen `small` und `big` verteilt.
3. Die Methode `sort` sortiert die als Argument übergebene Liste.
- (a) Falls diese Liste leer ist, kann sie unverändert zurück gegeben werden.
 - (b) Andernfalls wird das erste Element der Liste `list` aus dieser Liste entfernt und in der Variablen `pivot` abgespeichert.
 - (c) Dann werden zwei leere Listen `small` und `big` angelegt.
 - (d) Der anschließende Aufruf der Methode `partition` verteilt die restlichen Elemente der übergebenen Liste auf diese beiden Listen.
 - (e) Danach werden diese Listen durch rekursive Aufrufe der Methode `sort` sortiert.
 - (f) Zum Schluss wird an die sortierte Liste der Elemente, die kleiner als das Pivot-Element sind, erst das Pivot-Element und dann die sortierte Liste der Elemente, die größer als das Pivot-Element sind, angehängt.
 - (g) Die so erhaltene Liste wird als Ergebnis zurück gegeben.

6.4.1 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

$$\text{partition}(x, L, S, B)$$

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 $\text{sort}(L)$ 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 $\text{partition}(x, R)$ in Zeile 6 von Abbildung 6.11 durchgeführt werden und der Term a_{n-1} erfasst die Anzahl der Vergleiche, die beim rekursiven Aufruf von $\text{sort}(L_2)$ 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:

$$\begin{aligned} a_n &= a_{n-1} + (n - 1) \\ &= a_{n-2} + (n - 2) + (n - 1) \\ &= a_{n-3} + (n - 3) + (n - 2) + (n - 1) \\ &= \vdots \\ &= a_0 + 0 + 1 + 2 + \cdots + (n - 2) + (n - 1) \\ &= 0 + 0 + 1 + 2 + \cdots + (n - 2) + (n - 1) \\ &= \sum_{i=0}^{n-1} i = \frac{1}{2}n \cdot (n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n. \end{aligned}$$

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 6.11 berechnet wird, ein Element der Menge $\{0, 1, 2, \dots, n\}$. Hat die Liste **small** insgesamt i Elemente, so enthält die Liste **big** die restlichen $n - i$ Elemente. Gilt $\#\text{small} = i$, 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 $i \in \{0, 1, \dots, n\}$, so erhalten wir für d_{n+1} die Rekurrenz-Gleichung

$$d_{n+1} = n + \frac{1}{n+1} \sum_{i=0}^n (d_i + d_{n-i}) \quad (6.1)$$

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 $f : \mathbb{N} \rightarrow \mathbb{N}$ folgendes gilt:

$$\sum_{i=0}^n f(n-i) = f(n) + f(n-1) + \cdots + f(1) + f(0) \quad (6.2)$$

$$= f(0) + f(1) + \cdots + f(n-1) + f(n) \quad (6.3)$$

$$= \sum_{i=0}^n f(i) \quad (6.4)$$

Damit vereinfacht sich die Rekurrenz-Gleichung (6.1) zu

$$d_{n+1} = n + \frac{2}{n+1} \cdot \sum_{i=0}^n d_i. \quad (6.5)$$

Um diese Rekurrenz-Gleichung lösen zu können, substituieren wir $n \mapsto n+1$ und erhalten

$$d_{n+2} = n+1 + \frac{2}{n+2} \cdot \sum_{i=0}^{n+1} d_i. \quad (6.6)$$

Wir multiplizieren nun Gleichung (6.6) mit $n+2$ und Gleichung (6.5) mit $n+1$ und erhalten

$$(n+2) \cdot d_{n+2} = (n+2) \cdot (n+1) + 2 \cdot \sum_{i=0}^{n+1} d_i \quad \text{und} \quad (6.7)$$

$$(n+1) \cdot d_{n+1} = (n+1) \cdot n + 2 \cdot \sum_{i=0}^n d_i. \quad (6.8)$$

Wir bilden die Differenz der Gleichungen (6.7) und (6.8) und beachten, dass sich die Summationen bis auf den Term $2 \cdot d_{n+1}$ gerade gegenseitig aufheben. Dann erhalten wir

$$(n+2) \cdot d_{n+2} - (n+1) \cdot d_{n+1} = (n+2) \cdot (n+1) - (n+1) \cdot n + 2 \cdot d_{n+1} \quad (6.9)$$

Diese Gleichung vereinfachen wir zu

$$(n+2) \cdot d_{n+2} = (n+3) \cdot d_{n+1} + 2 \cdot (n+1). \quad (6.10)$$

Einer genialen Eingebung folgend teilen wir diese Gleichung durch $(n+2) \cdot (n+3)$ und erhalten

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} + \frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}. \quad (6.11)$$

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 (6.11) als

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} - \frac{2}{n+2} + \frac{4}{n+3} \quad (6.12)$$

schreiben. Wir definieren $a_n = \frac{d_n}{n+1}$ 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

$$a_n = a_{n-1} - \frac{2}{n} + \frac{4}{n+1} \quad (6.13)$$

Diese Gleichung können wir mit dem Teleskop-Verfahren lösen. Wegen $a_0 = \frac{d_0}{1} = 0$ gilt

$$a_n = 4 \cdot \sum_{i=1}^n \frac{1}{i+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i}. \quad (6.14)$$

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 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

$$f(n) = g(n) + \mathcal{O}(h(n)) \quad \text{g.d.w.} \quad f(n) - g(n) \in \mathcal{O}(h(n)).$$

Wegen $d_n = (n+1) \cdot a_n$ gilt jetzt:

$$\begin{aligned} d_n &= -4 \cdot n + 2 \cdot (n+1) \cdot H_n \\ &= -4 \cdot n + 2 \cdot (n+1) \cdot (\ln(n) + \mathcal{O}(1)) \\ &= 2 \cdot n \cdot \ln(n) + \mathcal{O}(n). \end{aligned}$$

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*”.

6.4.2 Eine feldbasierte Implementierung von *Quick-Sort*

Zum Abschluss geben wir eine feldbasierte Implementierung des *Quick-Sort*-Algorithmus an. Abbildung 6.12 zeigt diese Implementierung.

1. Im Gegensatz zu der feldbasierten Implementierung des *Merge-Sort-Algorithmus* benötigen wir diesmal kein zusätzliches Hilfsfeld. Die Klasse `QuickSortAlgorithm` hat daher nur die Member-Variable `mArray`. Diese Member-Variablen werden durch den Konstruktor initialisiert.
2. Die Methode `sort` wird auf die Implementierung der Methode `quickSort` zurück geführt. Diese Methode bekommt die beiden Parameter `start` und `end`.
 - (a) `start` gibt den Index des ersten Elementes des Teilfeldes an, das zu sortieren ist.
 - (b) `end` gibt den Index des letzten Elementes des Teilfeldes an, das zu sortieren ist.

Der Aufruf `quickSort(start, end)` sortiert die Elemente

$$\text{mArray}[\text{start}], \text{mArray}[\text{start}+1], \dots, \text{mArray}[\text{end}]$$

des Feldes `mArray`, d. h. nach dem Aufruf gilt:

$$\text{mArray}[\text{start}] \preceq \text{mArray}[\text{start}+1] \preceq \dots \preceq \text{mArray}[\text{end}].$$

Die Implementierung der Methode `quickSort` entspricht weitgehend der listenbasierten Implementierung. Der wesentliche Unterschied besteht darin, dass die Funktion `partition`, die in Zeile 13 aufgerufen wird, die Elemente des Feldes `array` so umverteilt, dass hinterher alle Elemente, die kleiner oder gleich dem *Pivot-Element* sind, links vor dem Index `splitIdx` stehen, während die restlichen Elemente rechts von dem Index stehen. Das *Pivot-Element* selbst steht hinterher an der durch `splitIdx` bezeichneten Stelle.

3. Die Schwierigkeit bei der Implementierung von *Quick-Sort* liegt in der Codierung der Methode `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.

```

1  public class QuickSortArray
2  {
3      private Double[] mArray; // the array to be sorted
4
5      QuickSortArray(Double[] array) {
6          mArray = array;
7      }
8      public void sort() {
9          quickSort(0, mArray.length - 1);
10     }
11     private void quickSort(int start, int end) {
12         if (end <= start) {
13             return;
14         }
15         int splitIdx = partition(start, end);
16         quickSort(start, splitIdx - 1);
17         quickSort(splitIdx + 1, end );
18     }
19     int partition(int start, int end) {
20         Double x      = mArray[start];
21         int leftIdx   = start + 1;
22         int rightIdx  = end;
23         while (true) {
24             while (leftIdx <= end && mArray[leftIdx] <= x) {
25                 ++leftIdx;
26             }
27             while (mArray[rightIdx] > x) {
28                 --rightIdx;
29             }
30             if (leftIdx >= rightIdx) {
31                 assert leftIdx == rightIdx + 1 : "left == right";
32                 break;
33             }
34             swap(leftIdx, rightIdx);
35         }
36         swap(start, rightIdx);
37         return rightIdx;
38     }
39     protected void swap(int i, int j) {
40         if (i == j) return;
41         Double temp = mArray[i];
42         mArray[i]   = mArray[j];
43         mArray[j]   = temp;
44     }
45 }

```

Abbildung 6.12: Implementierung des *Quick-Sort*-Algorithmus in *Java*.

Die Funktion `partition` liefert als Resultat einen Index `splitIdx` aus der Menge

$$\text{splitIdx} \in \{\text{start}, \text{start} + 1, \dots, \text{end}\}.$$

Außerdem wird der Teil des Feldes zwischen `start` und `end` so umsortiert, dass nach dem

Aufruf der Methode gilt:

- (a) Alle Elemente mit Index aus der Menge $\{\text{start}, \dots, \text{splitIdx} - 1\}$ kommen in der Ordnung “ \preceq ” vor dem Element an der Stelle `splitIdx`:

$$\forall i \in \{\text{start}, \dots, \text{splitIdx} - 1\}: \text{mArray}[i] \preceq \text{mArray}[\text{splitIdx}].$$

- (b) Alle Elemente mit Index aus der Menge $\{\text{splitIdx} + 1, \dots, \text{end}\}$ kommen in der Ordnung “ \preceq ” hinter dem Element an der Stelle `splitIdx`:

$$\forall i \in \{\text{splitIdx} + 1, \dots, \text{end}\}: \text{mArray}[\text{splitIdx}] \prec \text{mArray}[i].$$

Der Algorithmus, um diese Bedingungen zu erreichen, wählt zunächst das Element

`mArray[start]`

als sogenanntes *Pivot-Element* aus. Anschließend läuft der Index `leftIdx` 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 `rightIdx` ausgehend von dem Index `end` von rechts nach links, bis ein Element gefunden wird, dass kleiner oder gleich dem Pivot-Element ist. Falls nun `leftIdx` kleiner als `rightIdx` ist, werden die entsprechenden Elemente des Feldes ausgetauscht. In dem Moment, wo `leftIdx` größer oder gleich `rightIdx` wird, wird dieser Prozess abgebrochen. Jetzt wird noch das Pivot-Element in die Mitte gestellt, anschließend wird `rightIdx` zurück gegeben.

4. Der Aufruf `swap(i, j)` vertauscht die Elemente des Arrays `mArray`, die die Indizes i und j haben.

Der einfachste Weg um zu verstehen, wie die Methode `partition` funktioniert, besteht darin, diese Methode anhand eines Beispiels auszuprobieren. Wir betrachten dazu einen Ausschnitt aus einem Feld, der die Form

$\dots, 7, 2, 9, 1, 8, 5, 11, \dots$

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 wird 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

$\dots, 7, 2, 5, 1, 8, 9, 11, \dots$

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 $\mathbf{left} \geq \mathbf{right}$ 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

$\dots, 1, 2, 5, 7, 8, 9, 11, \dots$

Als Ergebnis wird nun der Index **right**, der jetzt auf das Pivot-Element zeigt, zurück gegeben.

6.4.3 Korrektheit

Die Implementierung der Methode **partition** ist trickreich. Daher untersuchen wir die Korrektheit der Methode jetzt im Detail. Zunächst formulieren wir Invarianten, die für die äußere **while**-Schleife, die sich von Zeile 23 bis Zeile 35 erstreckt, gelten. Wir bezeichnen das Pivot-Element mit x . Dann gelten die folgenden Invarianten:

$$(I1) \quad \forall i \in \{\mathbf{start} + 1, \dots, \mathbf{left} - 1\}: \mathbf{mArray}[i] \preceq x$$

$$(I2) \quad \forall j \in \{\mathbf{right} + 1, \dots, \mathbf{end}\}: x \prec \mathbf{mArray}[j]$$

$$(I3) \quad \mathbf{start} + 1 \leq \mathbf{left}$$

$$(I4) \quad \mathbf{right} \leq \mathbf{end}$$

$$(I5) \quad \mathbf{left} \leq \mathbf{right} + 1$$

Wir weisen die Gültigkeit dieser Invarianten nach. Dazu ist zunächst zu zeigen, dass diese Invarianten dann erfüllt sind, wenn die Schleife zum ersten Mal durchlaufen wird. Zu Beginn gilt

$$\mathbf{left} = \mathbf{start} + 1.$$

Daraus folgt sofort, dass die dritte Invariante anfangs erfüllt ist. Außerdem gilt dann

$$\{\mathbf{start} + 1, \dots, \mathbf{left} - 1\} = \{\mathbf{start} + 1, \dots, \mathbf{start}\} = \{\}$$

und damit ist auch klar, dass die erste Invariante gilt, denn für $\mathbf{left} = \mathbf{start} + 1$ ist die erste Invariante eine leere Aussage. Weiter gilt zu Beginn

$$\mathbf{right} = \mathbf{end},$$

woraus unmittelbar die Gültigkeit der vierten Invariante folgt. Außerdem gilt dann

$$\{\mathbf{right} + 1, \dots, \mathbf{end}\} = \{\mathbf{end} + 1, \dots, \mathbf{end}\} = \{\},$$

so dass auch die zweite Invariante trivialerweise erfüllt ist. Für die fünfte Invariante gilt anfangs

$$\mathbf{left} \leq \mathbf{right} + 1 \leftrightarrow \mathbf{start} + 1 \leq \mathbf{end} + 1 \leftrightarrow \mathbf{start} \leq \mathbf{end} \leftrightarrow \mathbf{true},$$

denn die Methode `partition(start, end)` wird nur aufgerufen, falls `start < 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

$$\mathbf{mArray}[\mathbf{left}] \preceq x$$

gilt. Wenn die Menge $\{\mathbf{start} + 1, \dots, \mathbf{left} - 1\}$ also um $i = \mathbf{left}$ vergrößert wird, so ist sichergestellt, dass für dieses i gilt:

$$\mathbf{mArray}[i] \preceq x.$$

2. Die zweite Invariante gilt, weil `right` nur dann dekrementiert wird, wenn vorher

$$x \prec \mathbf{mArray}[\mathbf{right}]$$

gilt. Wenn die Menge $\{\mathbf{right} + 1, \dots, \mathbf{end}\}$ also um $i = \mathbf{right}$ vergrößert wird, so ist sichergestellt, dass für dieses i gilt

$$x \prec \mathbf{mArray}[i].$$

3. Die Gültigkeit der dritten Invariante folgt aus der Tatsache, dass `left` in der ganzen Schleife höchstens inkrementiert wird. Wenn also zu Beginn $\mathbf{start} + 1 \leq \mathbf{left}$ gilt, so wird dies immer gelten.
4. Analog ist die vierten Invariante gültig, weil zu Beginn $\mathbf{right} \leq \mathbf{end}$ gilt und `right` immer nur dekrementiert wird.
5. Aus den ersten beiden Invarianten (I1) und (I2) folgt:

$$\{\mathbf{start} + 1, \dots, \mathbf{left} - 1\} \cap \{\mathbf{right} + 1, \dots, \mathbf{end}\} = \{\},$$

denn ein Element des Arrays kann nicht gleichzeitig kleiner-gleich x und größer x sein. Wenn $\mathbf{right} + 1 \leq \mathbf{end}$ ist, dann ist die zweite Menge nicht-leer und es folgt

$$\mathbf{left} - 1 < \mathbf{right} + 1 \quad \text{und das impliziert} \quad \mathbf{left} \leq \mathbf{right} + 1.$$

Andernfalls gilt $\mathbf{right} = \mathbf{end}$. Dann haben wir

$$\mathbf{left} \leq \mathbf{right} + 1 \leftrightarrow \mathbf{left} \leq \mathbf{end} + 1 \leftrightarrow \mathbf{true},$$

denn wenn $\mathbf{left} > \mathbf{end}$ ist, wird `left` in der ersten Schleife nicht mehr erhöht. `left` wird nur dann und auch nur um 1 inkrementiert, solange $\mathbf{left} \leq \mathbf{end}$ gilt. Also kann `left` maximal den Wert $\mathbf{end} + 1$ 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

$$\mathbf{left} \leq \mathbf{end}$$

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:

$$\mathbf{right} + 1 \geq \mathbf{left} \geq \mathbf{start} + 1.$$

Die äußere **while**-Schleife terminiert, weil die Menge

$$M = \{\text{left}, \dots, \text{right}\}$$

ständig verkleinert wird. Um das zu sehen, führen wir eine Fall-Unterscheidung durch:

1. Fall: Nach dem Ende der Schleife in Zeile 24 – 26 gilt

$$\text{left} > \text{end}.$$

Diese Schleife bricht also ab, weil die Bedingung $\text{left} \leq \text{end}$ verletzt ist. Wir haben oben schon gesehen, dass dann

$$\text{left} = \text{end} + 1 \quad \text{und} \quad \text{right} = \text{end}$$

gelten muss. Daraus folgt aber sofort

$$\text{left} > \text{right}$$

und folglich wird die äußere Schleife dann durch den Befehl **break** in Zeile 32 abgebrochen.

2. Fall: Nach dem Ende der Schleife in Zeile 24 – 26 gilt

$$\text{left} \leq \text{end}.$$

Die Schleife bricht also ab, weil die Bedingung $\text{mArray}[\text{left}] \preceq x$ verletzt ist, es gilt also

$$x \prec \text{mArray}[\text{left}].$$

Analog gilt nach dem Abbruch der zweiten inneren **while**-Schleife

$$\text{mArray}[\text{right}] \preceq x.$$

Wenn die äußere Schleife nun nicht abbricht weil $\text{left} < \text{right}$ ist, dann werden die Elemente $\text{mArray}[\text{left}]$ und $\text{mArray}[\text{right}]$ vertauscht. Nach dieser Vertauschung gilt offenbar

$$x \prec \text{mArray}[\text{right}] \quad \text{und} \quad \text{mArray}[\text{left}] \preceq x.$$

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 $M = \{\text{left}, \dots, \text{right}\}$ um ein Element verkleinert wird. Das geht aber nur endlich oft, denn spätestens wenn die Menge leer ist, gilt $\text{left} = \text{right} + 1$ und die Schleife wird durch den Befehl **break** in Zeile 32 abgebrochen.

Jetzt haben wir alles Material zusammen, um die Korrektheit unserer Implementierung zu zeigen. Wenn die Schleife abbricht, gilt $\text{left} > \text{right}$. Wegen der fünften Invariante gilt $\text{left} \leq \text{right} + 1$. Also gibt es nur noch die Möglichkeit

$$\text{left} = \text{right} + 1.$$

Wegen den ersten beiden Invarianten wissen wir also

$$\forall i \in \{\text{start} + 1, \dots, \text{right}\}: \text{mArray}[i] \preceq x$$

$$\forall j \in \{\text{right} + 1, \dots, \text{end}\}: x \prec \text{mArray}[j]$$

Durch das **swap** in Zeile 17 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.

6.4.4 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.

6.5 Timsort

Der Algorithmus “*Sortieren durch Mischen*” ist in der Praxis der Algorithmus, der am effizientesten arbeitet. Dies schließe ich daraus, dass dieser Algorithmus beispielsweise sowohl in der Sprache *Python* als auch in *Java* in den Bibliotheken zum Sortieren eingesetzt wird. Bei einer praktischen Implementierung von Merge-Sort gibt es eine Reihe von Tricks, die verwendet werden können, um die Effizienz zu steigern. Tim Peters hat eine Reihe solcher Tricks zusammengestellt:

<http://mail.python.org/pipermail/python-dev/2002-July/026837.html>

Der so verbesserte Algorithmus “*Sortieren durch Mischen*” wird als “*Timsort*” bezeichnet. In der neuesten Version der Sprache *Java*, die voraussichtlich im Sommer des Jahres 2011 erscheinen wird, ist die Methode `sort()` in der Klasse `java.util.Arrays` mit Hilfe von *Timsort* implementiert:

<http://hg.openjdk.java.net/jdk7/jdk7/file/jdk7-b76/src/share/classes/java/util/TimSort.java>

Ausgangspunkt der Entwicklung von *Timsort* war die Tatsache, dass die zu sortierenden Daten in der Praxis häufig die folgenden Eigenschaften haben:

1. Oft sind Teilfelder bereits vorsortiert, allerdings nicht immer aufsteigend sondern genau so häufig auch absteigend.
2. Die Daten innerhalb eines Feldes sind oft *klumpig*: Damit ist gemeint, dass das Feld in Teilfelder aufgeteilt werden kann, in denen die Daten entweder alle relativ groß oder klein sind.

Aus diesem Grunde verwendet *Timsort* die folgenden Tricks um ein Feld zu sortieren.

1. Erkennen vorsortierter Felder.

In einem ersten Schritt unterteilen wir das zu sortierende Feld in Teilfelder, die wahlweise aufsteigend oder absteigend sortiert sind. Anschließend wird ein absteigend sortiertes Teilfeld umgedreht, so dass es danach aufsteigend sortiert ist.

2. Verlängern zu kleiner Felder.

“*Sortieren durch Mischen*” hat nur dann eine Komplexität von $\mathcal{O}(n \cdot \ln(n))$, wenn wir sicherstellen können, dass die zu mischenden Teilfelder annähernd die gleiche Größe haben. Daher wird ein vorsortiertes Teilfeld, dessen Länge kürzer als eine gewisse Mindestlänge ist, künstlich auf eine vorgegebene Mindestlänge verlängert. Als Mindestlänge wird ein Zahl zwischen 32 und 63 gewählt.

Zum Verlängern der Teilfelder auf die Mindestlänge wird das Verfahren “*Sortieren durch Einfügen*” benutzt, denn dieses Verfahren hat für Felder, die bereits teilweise vorsortiert sind, nur eine lineare Komplexität. Das Verfahren wird noch dadurch verbessert, dass beim Einfügen eine binäre Suche verwendet wird. Diese verbesserte Variante von “*Sortieren durch Einfügen*” bezeichnen wir als “*binäres Sortieren durch Einfügen*”.

3. Verwaltung eines Stacks mit den zu sortierenden Teilfeldern.

Wie bereits oben erwähnt wurde, kann die Komplexität von $\mathcal{O}(n \cdot \ln(n))$ nur dann sichergestellt werden, wenn die zu mischenden Teilfelder im wesentlichen dieselbe Länge haben. Dies wird dadurch erreicht, dass die vorsortierten Teilfelder auf einem Stack verwaltet werden. Dabei wird darauf geachtet, dass die zu mischenden Teilfelder im wesentlichen dieselbe Größe haben.

4. Verbesserungen des Algorithmus zum Mischen.

Werden zwei Teilfelder gemischt bei denen alle Elemente des ersten Teilfeldes kleiner als alle Elemente des zweiten Teilfeldes sind, so würde der konventionelle Algorithmus zum Mischen alle Elemente des ersten Teilfeldes mit dem ersten Element des zweiten Teilfeldes

vergleichen und hätte daher eine lineare Komplexität. *Timsort* erkennt, wenn zwei Teilfelder stark unterschiedlich sind und verwendet in diesem Fall *exponentielle Suche*. Dadurch hat das Mischen zweier Teilfelder in vielen in der Praxis wichtigen Spezialfällen nur eine Komplexität, die logarithmisch von der Größe der Teilfelder abhängt.

Wir diskutieren nun eine vereinfachte Version des Algorithmus *Timsort*. Ausgangspunkt bei dieser vereinfachten Version war die Implementierung von *Timsort* in *Java 7*. Die Original-Version in der JDK ist etwa doppelt so lang, so dass eine Diskussion der Original-Version für die Vorlesung zu aufwendig wäre. Abbildung 6.13 zeigt die Struktur der Klasse `SimplifiedTimSort.java`:

```

1  public class SimplifiedTimSort {
2      private static final int MIN_MERGE = 32;
3      private static final int MIN_GALLOP = 7;
4
5      private Double[] mArray;      // the array to be sorted
6      private Double[] mAux;        // an auxilliary array
7
8      private int mStackSize = 0;    // number of pending runs on stack
9      private int[] mRunBase;
10     private int[] mRunLen;
11
12     public SimplifiedTimSort(Double[] array) {
13         mArray = array;
14         mAux = new Double[array.length];
15         mRunBase = new int[40];
16         mRunLen = new int[40];
17     }
18
19     public void sort() { ... }
20
21     private void binarySort(int low, int high, int start) { ... }
22     private int countRunAndMakeAscending(int low) { ... }
23     private void reverseRange(int low, int high) { ... }
24     private void pushRun(int runBase, int runLen) { ... }
25     private void mergeCollapse() { ... }
26     private void mergeForceCollapse() { ... }
27     private void mergeAt(int i) { ... }
28     private int gallop(Double x, int base, int len) { ... }
29     private void merge(int base1, int len1, int base2, int len2) { ... }
30 }

```

Abbildung 6.13: Struktur der Klasse `SimplifiedTimSort`.

1. Die Konstante `MIN_MERGE` gibt die Länge an, die Teilfelder mindestens haben müssen, bevor Sie mit anderen Teilfeldern gemischt werden. In der tatsächlichen Implementierung wird hier eine Zahl zwischen 32 und 63 gewählt, die aber noch von der Länge des zu sortierenden Feldes abhängt. In der optimalen Implementierung ist das Ziel, diese Zahl so zu wählen, dass möglichst alle zu mischenden Teilfelder die gleiche Länge haben.
2. Die Konstante `MIN_GALLOP` legt fest, wann beim Mischen zweier Teilfelder eine *exponentielle Suche* verwendet wird. Diesen Begriff werden wir später noch näher erläutern.
3. `mArray` bezeichnet das zu sortierende Feld.

4. `mAux` ist das Hilfsfeld, was wir zum Mischen benötigen.
5. Die Klasse verwaltet intern einen Stack, auf dem zu mischende Teilfelder abgelegt werden. Dieser Stack wird durch drei Variablen implementiert:
 - (a) `mStackSize` gibt die Anzahl der Teilfelder an, die auf dem Stack liegen und auf eine Sortierung warten.
 - (b) `mRunBase[i]` ist der Index des ersten Elements des i -ten Teilfeldes.
 - (c) `mRunLen[i]` gibt die Anzahl der Elemente des i -ten Teilfeldes an.
6. Der Konstruktor initialisiert die Member-Variablen der Klasse. Wir werden später sehen, dass der Stack, der die zu sortierenden Teilfelder enthält, nie mehr als 40 Elemente enthalten kann, falls das zu sortierende Feld mit einem *Java int* indiziert werden kann.

Wir diskutieren nun die verschiedenen Methoden der Klasse `SimplifiedTimSort`. Wir beginnen mit der in Abbildung 6.14 gezeigten Methode `sort()`, deren Aufgabe es ist, das Feld `mArray` zu sortieren.

```

1  public void sort() {
2      int low = 0;
3      int nRemaining = mArray.length;
4      if (nRemaining < 2) {
5          return; // Arrays of size 0 and 1 are always sorted
6      }
7      if (nRemaining < MIN_MERGE) {
8          int initRunLen = countRunAndMakeAscending(low);
9          binarySort(low, mArray.length, low + initRunLen);
10         return;
11     }
12     do {
13         int runLen = countRunAndMakeAscending(low);
14         if (runLen < MIN_MERGE) {
15             int force = nRemaining <= MIN_MERGE ? nRemaining : MIN_MERGE;
16             binarySort(low, low + force, low + runLen);
17             runLen = force;
18         }
19         pushRun(low, runLen);
20         mergeCollapse(); // establish stack invariants
21         low += runLen; // Advance to find next run
22         nRemaining -= runLen;
23     } while (nRemaining != 0);
24     mergeForceCollapse();
25 }

```

Abbildung 6.14: Die Methode `sort()`

1. Die Variable `low` ist der Index des ersten noch unsortierten Elements in dem Feld `mArray`. Diese Variable wird daher zunächst mit 0 initialisiert.
2. `nRemaining` ist die Anzahl der noch zu sortierenden Elemente.
3. Felder mit weniger als zwei Elementen sind bereits sortiert.

4. Kleine Felder, konkret solche Felder die weniger als `MIN_MERGE` Elemente haben, werden mit Hilfe einer Variante des Algorithmus “*Sortieren durch Einfügen*” sortiert. Dazu sucht die Methode `countRunAndMakeAscending()` zunächst das größte Teilfeld, das beginnend an dem Index `low` entweder aufsteigend oder absteigend sortiert ist. Falls das Teilfeld absteigend sortiert ist, werden die Elemente innerhalb des Teilfeldes umgedreht, so dass das Teilfeld anschließend auf jeden Fall aufsteigend sortiert ist. Die Methode `countRunAndMakeAscending()` gibt als Ergebnis die Länge des aufsteigend sortierten Teilfelds zurück. Wenn das Programm in Zeile 9 angekommen ist, dann wissen wir, dass das Teilfeld

$$[\text{mArray}[\text{low} + i] \mid i \in [0, \dots, \text{initRunLen} - 1]]$$

sortiert ist. Die Elemente beginnend mit dem Index `low + initRunLen` müssen noch in dieses Feld einsortiert werden. Dies wird von der Methode `binarySort()` in Zeile 9 geleistet.

5. Große Felder werden zunächst in Teilfelder, die bereits sortiert sind, aufgespalten. Dazu wird in Zeile 13 zunächst das längste sortierte Teilfeld berechnet, dass an dem Index `low` beginnt und das bereits sortiert ist. Dann wird in Zeile 14 geprüft, ob dieses Teilfeld die Mindestlänge `MIN_MERGE` besitzt. Falls nicht und wenn außerdem noch mehr als `MIN_MERGE` Elemente vorhanden sind, dann wird dieses Teilfeld durch den Aufruf der Methode `binarySort()` in Zeile 16 zu einem sortierten Teilfeld der Länge `force` verlängert. Diese Länge ist `MIN_MERGE`, falls mehr als `MIN_MERGE` Elemente übrig sind, sonst ist diese Länge einfach die Anzahl aller noch unsortierten Elemente. Zum Sortieren wird wieder der Algorithmus “*Sortieren durch Einfügen*” verwendet.
6. Das sortierte Teilfeld wird in Zeile 19 von der Methode `pushRun()` auf den Stack der bereits sortierten Teilfelder gelegt. Liegen schon mehrere Teilfelder auf dem Stack und sind die Größen dieser Teilfelder nicht zu stark unterschiedlich, so mischt die Methode `mergeCollapse()` einige der auf dem Stack liegenden Teilfelder. Dies werden wir später noch im Detail analysieren, wenn wir die Methode `mergeCollapse()` besprechen.
7. Anschließend wird in Zeile 21 der Index `low` um die bereits sortierten Elemente erhöht, und die Zahl der noch zu sortierenden Elemente wird entsprechend erniedrigt.
8. Zum Abschluss der Methode werden alle noch auf dem Stack verbliebenen Teilfelder so gemischt, dass das resultierende Feld insgesamt aufsteigend geordnet ist.

Abbildung 6.15 zeigt die Methode `binarySort()`. Ein Aufruf der Form

$$\text{binarySort}(\text{low}, \text{high}, \text{start})$$

hat die Aufgabe, das Teilfeld

$$\text{mArray}[\text{low}, \dots, \text{high} - 1]$$

zu sortieren. Dabei darf vorausgesetzt werden, dass das Teilfeld

$$\text{mArray}[\text{low}, \dots, \text{start} - 1]$$

bereits sortiert ist. Es müssen also lediglich die Elemente `mArray[start], ..., mArray[high - 1]`, in das bereits sortierte Teilfeld eingefügt werden. Dazu läuft die `for`-Schleife in Zeile 4 über alle Indizes i aus dem Intervall $[\text{start}, \text{high} - 1]$ und fügt die Elemente `mArray[i]` so in das schon sortierte Teilfeld ein, dass die Sortierung erhalten bleibt. Die Invariante der `for`-Schleife ist also, dass das Teilfeld

$$\text{mArray}[\text{low}, \dots, i - 1]$$

bereits sortiert ist und die Aufgabe des nächsten Schleifendurchlaufs ist es, für das Element `mArray[i]` eine Position $k \in \{\text{low}, \dots, i\}$ zu suchen, an der es eingefügt werden kann. Für diesen Index k soll gelten:

1. $\forall j \in \{\text{low}, \dots, k - 1\} : \text{mArray}[j] \leq \text{mArray}[i]$ und

```

1  private void binarySort(int low, int high, int start)
2  {
3      assert low < start && start <= high;
4      for (int i = start; i < high; ++i) {
5          Double next = mArray[i];
6          int left = low;
7          int right = i;
8          assert left <= right;
9          while (left < right) {
10             int middle = left + (right - left) / 2;
11             if (next < mArray[middle]) {
12                 right = middle;
13             } else {
14                 left = middle + 1;
15             }
16         }
17         assert left == right;
18         System.arraycopy(mArray, left, mArray, left + 1, i - left);
19         mArray[left] = next;
20     }
21     assert isSorted(low, high): "binarySort: not sorted";
22 }

```

Abbildung 6.15: Die Methode *binarySort()*

2. $\forall j \in \{k, \dots, i-1\} : \mathbf{mArray}[j] > \mathbf{mArray}[i]$.

Um den Index k zu bestimmen, verwendet die Methode *binarySort()* das Verfahren der Intervall-Halbierung. Dazu wird die linke Grenze **left** des Intervalls mit **low** initialisiert, die rechte Grenze **right** wird mit i initialisiert, denn das sind die beiden extremen Positionen, die der Index k annehmen kann:

1. Falls alle Elemente der Menge $\{\mathbf{mArray}[j] \mid j \in \{\mathbf{low}, \dots, i-1\}\}$ größer als $\mathbf{mArray}[i]$ sind, so wird das Element an der Position \mathbf{low} eingefügt und die Elemente des Feldes **mArray** werden nach rechts verschoben.
2. Falls alle Elemente der Menge $\{\mathbf{mArray}[j] \mid j \in \{\mathbf{low}, \dots, i-1\}\}$ kleiner-gleich $\mathbf{mArray}[i]$ sind, so wird das Element an der Position i eingefügt und bleibt folglich da, wo es schon ist.

Die **while**-Schleife in Zeile 9 hat die folgenden beiden Invarianten:

1. $\forall j \in \{\mathbf{low}, \mathbf{left} - 1\} : \mathbf{mArray}[j] \leq \mathbf{mArray}[i]$ und
2. $\forall j \in \{\mathbf{right}, i - 1\} : \mathbf{mArray}[i] < \mathbf{mArray}[j]$.

Zu Beginn sind diese beiden Invarianten sicher erfüllt, denn da $\mathbf{left} = \mathbf{low}$ ist, ist die Menge $\{\mathbf{low}, \mathbf{left} - 1\}$ leer und aus $\mathbf{right} = i$ folgt $\{\mathbf{right}, i - 1\} = \{\}$, so dass beide Aussagen trivial sind. Wir müssen nun zeigen, dass diese Invarianten bei jedem Schleifendurchlauf erhalten bleiben.

1. In Zeile 10 berechnen wir die Mitte *middle* des Intervalls $[\mathbf{left}, \mathbf{right}]$, wobei wir den Fall, dass $\mathbf{right} = \mathbf{left} + 1$ ist, später noch genauer analysieren müssen.

Der Ausdruck zur Berechnung der Mitte des Intervalls ist komplizierter, als Sie es auf den ersten Blick erwarten würden. Das Problem ist, dass es bei dem einfacheren Ausdruck

$$(\mathbf{left} + \mathbf{middle}) / 2$$

zu einem Überlauf kommen kann.

2. Falls $\text{mArray}[i] < \text{mArray}[middle]$ ist, so sind alle Elemente rechts von dem Index $middle$ sicher größer als das einzusortierende Element $\text{mArray}[i]$ und damit gilt für den Index k , den wir suchen, die Ungleichung

$$k \leq middle.$$

Daher können wir in diesem Fall die rechte Seite $right$ des Intervalls zu $middle$ verkleinern.

3. Falls $\text{mArray}[middle] \leq \text{mArray}[i]$ ist, so sind alle Elemente links von dem Index $middle$ sicher kleiner-gleich dem einzusortierenden Element $\text{mArray}[i]$. Da auch $\text{mArray}[middle] \leq \text{mArray}[i]$ gilt

$$k > middle.$$

Daher können wir in diesem Fall die linke Seite $left$ des Intervalls zu $middle + 1$ vergrößern, wobei die Invariante erhalten bleibt.

Falls nun die **while**-Schleife abbricht, muss danach $left = right$ gelten und damit ist $left$ (oder genauso gut $right$) der gesuchte Index k . Wir verschieben dann die Elemente des Teilfeldes

$$[\text{mArray}[left], \dots, i - 1]$$

um eine Position nach rechts. Dieses Teilfeld hat $(i - 1) - left + 1 = i - left$ Elemente. Anschließend kopieren wir das Element $\text{mArray}[i]$ an die nun freie Position $k = left$.

Es bleibt noch zu zeigen, dass die **while**-Schleife in Zeile 9 tatsächlich abbricht. Das Problem ist, dass das Intervall $[left, right]$ nur solange tatsächlich kleiner wird, solange $left$ und $right$ sich um mehr als 1 unterscheiden, denn nur dann ist $middle$ zwischen $left$ und $right$. Falls nun

$$right = left + 1$$

ist, liefert die Berechnung von $middle$ auf Grund der Ganzzahl-Division den Wert $left$:

$$middle = (left + left + 1)/2 = (2 \cdot left + 1)/2 = left.$$

Abhängig von dem Test in Zeile 11 gibt es nun zwei Fälle:

1. $\text{mArray}[i] < \text{mArray}[left]$

In diesem Fall wird $right := middle = left$ gesetzt, so dass neue Intervall jetzt die Form $[left, left]$ hat, so dass die Schleife abbricht, weil die linke Grenze mit der rechten Grenze übereinstimmt.

2. $\text{mArray}[i] \geq \text{mArray}[left]$

Jetzt haben wir

$$left := left + 1 = right$$

gesetzt, so dass das neue Intervall $[left, right]$ ebenfalls die Länge 0 hat, so dass die Schleife auch in diesem Fall abbricht.

In der Praxis zeigt sich, dass ein zu sortierendes Feld oft Teilfelder enthält, die bereits sortiert sind. Es ist sinnvoll, solche Felder vorab zu identifizieren. Die in Abbildung 6.16 gezeigte Methode `countRunAndMakeAscending(low)` hat die Aufgabe, innerhalb des Feldes `mArray` startend an der Position `low` das längste Teilfeld zu suchen, das bereits aufsteigend oder absteigend sortiert ist. Falls dieses Teilfeld absteigend sortiert ist, so wird es umgedreht. Die Methode `countRunAndMakeAscending` arbeitet wie folgt:

1. Der Index `high` ist eine echte obere Schranke für den oberen Index des bereits sortierten Teilfelds. Im günstigsten Fall geht dies bis zum Ende des Feldes, daher wird `high` mit `mArray.length` initialisiert.
2. Der Index `runHigh` zeigt auf den letzten Index des sortierten Feldes. Wir initialisieren `runHigh` mit `low + 1`, denn ein Feld der Länge 2 ist immer sortiert: Falls

```

1  private int countRunAndMakeAscending(int low)
2  {
3      int high    = mArray.length;
4      int runHigh = low + 1;
5      if (runHigh == high) {
6          return 1;
7      }
8      if (mArray[runHigh] < mArray[low]) {
9          ++runHigh;
10         while (runHigh < high && mArray[runHigh] < mArray[runHigh - 1]) {
11             ++runHigh;
12         }
13         reverseRange(low, runHigh); // reverse it
14     } else {
15         ++runHigh;
16         while (runHigh < high && mArray[runHigh - 1] <= mArray[runHigh] ) {
17             ++runHigh;
18         }
19     }
20     assert isSorted(low, runHigh): "run not sorted ";
21     return runHigh - low; // return length of actual run
22 }

```

Abbildung 6.16: Die Methode *countRunAndMakeAscending()*

$$mArray[low] \leq mArray[low + 1]$$

gilt, ist das Teilfeld

$$[mArray[low], mArray[low + 1]]$$

aufsteigend sortiert und wenn statt dessen

$$mArray[low] > mArray[low + 1]$$

gilt, dann ist dieses Teilfeld absteigend sortiert.

- Die beiden Fälle, dass das Teilfeld aufsteigend oder absteigend sortiert ist, werden nun getrennt betrachtet. Falls der Test

$$mArray[runHigh] < mArray[low]$$

in Zeile 8 erfolgreich ist, ist die Teilfolge, die wir suchen, absteigend sortiert. Die Invariante der *while*-Schleife in Zeile 10 lautet:

$$[mArray[low], \dots, mArray[runHigh - 1]] \text{ ist absteigend sortiert.}$$

Daher wird die Variable *runHigh* so lange inkrementiert, solange der nächste Wert kleiner als der vorhergehende Wert ist. Abschließend dreht die Methode *reverseRange* die Elemente des Teilfeldes

$$[mArray[low], \dots, mArray[runHigh - 1]]$$

so um, dass anschließend dieses Teilfeld aufsteigend sortiert ist.

- Falls das Teilfeld aufsteigend sortiert ist, wird das Teilfeld in analoger Weise solange nach oben erweitert, solange die neu hinzugefügten Elemente größer als die bereits vorhandenen sind.

5. Wenn am Schluss die Differenz $runHigh - low$ zurück gegeben wird, ist dies genau die Zahl der Elemente des Teilfeldes.

Die in Abbildung 6.17 gezeigte Methode `reverseRange(low, high)` hat die Aufgabe, das Teilfeld

$[mArray[low], \dots, mArray[high - 1]]$

umzudrehen. Dazu verwaltet diese Methode zwei Indizes l und h : l startet am linken Rand des Feldes und h am rechten Rand. In den Zeilen 5 – 7 werden die Werte, auf die l und h zeigen, vertauscht. Anschließend wird der linke Index inkrementiert und der rechte wird dekrementiert. Dies geschieht solange, bis sich die Indizes kreuzen. In diesem Fall bricht die Schleife ab.

```

1  private void reverseRange(int low, int high) {
2      int l = low;
3      int h = high - 1;
4      while (l < h) {
5          Double t = mArray[l];
6          mArray[l] = mArray[h];
7          mArray[h] = t;
8          ++l; --h;
9      }
10     assert isSorted(low, high - 1): "not sorted after reverse";
11 }

```

Abbildung 6.17: Die Methode `reverseRange()`.

Die in Abbildung 6.18 gezeigte Methode `pushRun()` hat die Aufgabe, ein Teilfeld, von dem wir bereits wissen, dass es aufsteigend sortiert ist, abzuspeichern. Hierzu reicht es aus, den Start-Index des Teilfeldes sowie die Länge des Feldes abzuspeichern. Hierzu werden die globalen Felder `mRunBase` und `mRunLen` als Stacks verwendet. Die globale Variable `mStackSize` gibt dabei an, wieviele Teilfelder bereits gespeichert sind.

```

1  private void pushRun(int runBase, int runLen) {
2      mRunBase[mStackSize] = runBase;
3      mRunLen [mStackSize] = runLen;
4      mStackSize++;
5  }

```

Abbildung 6.18: Die Methode `pushRun()`.

Abbildung 6.19 zeigt die Methode `mergeCollapse()`. Diese Methode hat die Aufgabe dafür zu sorgen, dass der Stack, auf dem die bereits sortierten Teilfelder abgespeichert sind, nicht zu groß wird. Dies wird durch zwei Invarianten sichergestellt:

1. Einerseits fordern wir, dass die Längen der Teilfelder, die auf dem Stack liegen, absteigend sind, es soll also gelten

$$\forall i \in \{0, \dots, mStackSize - 1\} : mRunLen[i - 1] > mRunLen[i].$$

2. Zusätzlich fordern wir, dass die Längen der Teilfelder, wenn wir den Stack von oben nach unten durchgehen, mindestens so schnell wachsen wie die Fibonacci-Zahlen:

$$\forall i \in \{0, \dots, mStackSize - 1\} : mRunLen[i - 2] > mRunLen[i - 1] + mRunLen[i].$$

Die zweite Bedingung stellt sicher, dass wir mit einem Stack der Größe 40 auskommen, denn

für die Summe der Fibonacci-Zahlen

$$S_n := \sum_{i=0}^n F_i$$

kann gezeigt werden, dass

$$\sum_{i=0}^n F_i = F_{n+2} - 1$$

gilt und F_{41} hat den Wert 267 914 296. Da jedes der Teilfelder mindestens eine Länge von 32 hat, reicht der Stack für Felder bis zur Größe $32 \cdot 267\,914\,291 = 8\,573\,257\,312$ auf jeden Fall aus. Ein Feld, das mit ganzen Zahlen indiziert wird, hat maximal eine Größe von $2^{31} = 2\,147\,483\,648$, so dass ein Stack der Größe 40 sicher ausreicht.

```

1  private void mergeCollapse() {
2      while (mStackSize > 1) {
3          int n = mStackSize - 2;
4          if (n > 0 && mRunLen[n-1] <= mRunLen[n] + mRunLen[n+1]) {
5              if (mRunLen[n - 1] < mRunLen[n + 1]) {
6                  --n;
7              }
8              mergeAt(n);
9          } else if (mRunLen[n] <= mRunLen[n + 1]) {
10             mergeAt(n);
11          } else {
12              break; // invariant is established
13          }
14      }
15  }
```

Abbildung 6.19: Die Methode *mergeCollapse()*.

Die Implementierung von *mergeCollapse()* stellt diese Invarianten sicher. Voraussetzung ist, dass die Invarianten bereits für alle Teilfelder mit eventueller Ausnahme des zuletzt auf den Stack gelegten Teilfeldes erfüllt sind.

1. Falls die Fibonacci-Invariante

$$\text{mRunLen}[i - 2] > \text{mRunLen}[i - 1] + \text{mRunLen}[i]$$

an der Spitze des Stacks verletzt ist, so werden entweder die beiden Teilfelder

$$i - 1 \text{ und } i$$

zu einem neuen Teilfeld gemischt, oder die beiden Teilfelder

$$i \text{ und } i + 1.$$

werden gemischt. Falls das Teilfeld $i + 1$ länger ist als das Teilfeld $i - 1$, so werden die beiden kürzeren Teilfelder $i - 1$ und i gemischt, andernfalls werden i und $i + 1$ gemischt.

2. Falls die Fibonacci-Invariante erfüllt ist, aber das neu auf dem Stack liegende Feld größer ist als das Feld darunter, so werden die beiden oben auf dem Stack liegenden Felder gemischt.
3. Da durch das Mischen der Stack verkürzt wird, sind die Invarianten eventuell wieder an der Spitze des Stacks verletzt. Daher müssen wir das Mischen solange weiterführen, bis die Invarianten erfüllt sind.

```

1  private void mergeForceCollapse() {
2      while (mStackSize > 1) {
3          mergeAt(mStackSize - 2);
4      }
5  }

```

Abbildung 6.20: Die Methode *mergeForceCollapse()*

Nachdem alle bereits sortierten Teilfelder auf den Stack gelegt worden sind, müssen wir diese solange mischen, bis nur noch ein einziges Teilfeld auf dem Stack liegen bleibt. Dieses Teilfeld ist dann das aufsteigend sortierte Feld. Die in Abbildung 6.20 gezeigte Methode *mergeForceCollapse()* leistet dies: Solange noch mindestens zwei Teilfelder auf dem Stack liegen, werden diese gemischt.

```

1  private void mergeAt(int i) {
2      int base1 = mRunBase[i];          // start of first run
3      int len1  = mRunLen[i];
4      int base2 = mRunBase[i + 1];    // start of second run
5      int len2  = mRunLen[i + 1];
6      mRunLen[i] = len1 + len2;
7      if (i == mStackSize - 3) {
8          mRunBase[i + 1] = mRunBase[i + 2];    // slide over last run
9          mRunLen [i + 1] = mRunLen [i + 2];
10     }
11     --mStackSize;
12     merge(base1, len1, base2, len2);
13     assert isSorted(base1, base2 + len2);
14 }

```

Abbildung 6.21: The methode *mergeAt()*.

Abbildung 6.21 zeigt die Methode *mergeAt(i)*, welche die Aufgabe hat, die beiden Teilfelder, die auf dem Stack an den Positionen i und $i + 1$ liegen, zu mischen. Dabei wird vorausgesetzt, dass

$$i \in \{mStackSize - 2, mStackSize - 3\}$$

gilt, es werden also entweder die beiden vorletzten odr die beiden letzten Teilfelder gemischt.

Das erste Teilfeld beginnt an der Position $mRunBase[i]$ und besteht aus $mRunLen[i]$ Elementen, das zweite Teilfeld beginnt entsprechend an der Position $mRunBase[i + 1]$ und besteht aus $mRunLen[i + 1]$ Elementen. Die beiden Teilfelder folgen unmittelbar aufeinander, es gilt also

$$mRunBase[i] + mRunLen[i] = mRunBase[i + 1].$$

Diese beiden Teilfelder werden durch den Aufruf der Methode *merge()* in Zeile 12 gemischt. Das dabei neue entstehende Teilfeld ersetzt die beiden ursprünglichen Teilfelder. Falls über den beiden Teilfeldern noch ein weiteres Teilfeld liegt, wird dieses nun in den Zeilen 8 und 9 im Stack an die Position $i + 1$ geschoben. Da die Methode nur aufgerufen wird, wenn entweder die letzten oder die vorletzten beiden Teilfelder auf dem Stack gemischt werden, reicht dies aus um den Stack zu verwalten.

Die in Abbildung 6.22 gezeigte Methode *gallop()* implementiert das Verfahren der *exponentiellen Suche* um die Position zu bestimmen, an der das erste Argument x in dem bereits sortierten Teilfeld, das an der Position b beginnt, eingeordnet werden muss. Genauer hat der Aufruf

$$gallop(x, b, l)$$

```

1  private int gallop(Double x, int b, int l) {
2      if (x < mAux[b]) {
3          return 0;
4      }
5      int lastK = 0;
6      int k      = 1;
7      while (k < l && mAux[b + k] <= x) {
8          lastK = k;
9          k      = 2 * k + 1;
10         if (k < 0) {
11             k = 1;
12         }
13     }
14     if (k > l) {
15         k = l;
16     }
17     while (lastK < k) {
18         int m = lastK + (k - lastK) / 2;
19         if (mAux[b + m] <= x) {
20             lastK = m + 1;
21         } else {
22             k = m;
23         }
24     }
25     return k;
26 }

```

Abbildung 6.22: Die Methode *gallop()*.

die Aufgabe, in dem Feld **mAux** innerhalb des Intervalls $[b, \dots, b + (l - 1)]$ die Position k zu finden, für die folgendes gilt:

1. Entweder haben wir

$$\forall i \in \{b, \dots, b + l - 1\} : \mathbf{mAux}[i] \leq x.$$

Dann gilt $k = \text{gallop}(x, b, l) = l$, denn in diesem Fall soll x hinter allen Elementen des Teilfeldes eingefügt werden.

2. Andernfalls bestimmen wir $k = \text{gallop}(x, b, l)$ so, dass folgendes gilt:

- (a) $\forall i \in \{b, \dots, b + k - 1\} : \mathbf{mAux}[i] \leq x,$
- (b) $\forall i \in \{b + k, \dots, b + (l - 1)\} : x < \mathbf{mAux}[i].$

Bei der Implementierung können wir voraussetzen, dass das Teilfeld

$$[\mathbf{mAux}[b], \dots, \mathbf{mAux}[b + (l - 1)]]$$

aufsteigend sortiert ist. Wir diskutieren nun die Details der in Abbildung 6.22 gezeigten Implementierung.

1. Falls das einzufügende Element x kleiner ist als das erste Element des Teilfeldes, soll x am Anfang des Teilfeldes einsortiert werden und wir geben in Zeile 3 den Index 0 zurück.
2. Ansonsten speichern wir in *lastK* den letzten Wert von k , den wir schon (erfolglos) ausprobiert haben und initialisieren k mit 1, den wir wissen ja schon, dass k größer als 0 sein muss, denn das erste Element des Teilfeldes ist ja kleiner-gleich x .

3. Solange k noch nicht über den rechten Rand des Teilfeldes herauszeigt und solange außerdem das Element an der Stelle $\mathbf{mAux}[b + k]$ kleiner-gleich x ist, müssen wir k vergrößern. Wir inkrementieren k aber nicht bloß um 1, was zu einer linearen Suche führen würde, sondern vergrößern k in Zeile 9 nach der Formel

$$k := 2 * k + 1.$$

Dabei merken wir uns jedesmal den alten Wert von k in der Variablen $lastK$.

4. Bei sehr großen Feldern kann es bei der Berechnung von $2 * k + 1$ zu einem Überlauf kommen. Einen Überlauf können wir daran erkennen, dass $2 * k + 1$ negativ wird. In diesem Fall setzen wir k auf den maximal zulässigen Wert l .
5. Wenn die **while**-Schleife in Zeile 7 abbricht, kann es passieren, dass k über die rechte Intervall-Grenze hinauszeigt. In diesem Fall setzen wir in Zeile 15 den Index k auf den maximal zulässigen Wert l .
6. Wenn das Programm in Zeile 17 ankommt, dann wissen wir, dass der gesuchte Wert von k sich innerhalb des Intervalls $[lastK + 1, \dots, k]$ befinden muss. Die genaue Position von k wird in der **while**-Schleife in Zeile 17 nun durch Intervall-Halbierung bestimmt:

- (a) Zunächst bestimmen wir in Zeile 18 die Mitte m des Intervalls.
- (b) Falls der Wert $\mathbf{mAux}[m] \leq x$ ist, muss x rechts von m liegen und wir können die linke Intervall-Grenze auf $m + 1$ erhöhen.
- (c) Andernfalls muss x in dem Intervall $[lastK, m]$ liegen und wir setzen die rechte Intervall-Grenze auf m .

Die Invariante der **while**-Schleife ist

$$\mathbf{mAux}[b + lastK] \leq x < \mathbf{mAux}[b + k].$$

Die Schleife verkleinert die Grenzen des Intervalls solange, bis $k = lastK + 1$ gilt. Wird dann m berechnet, so gilt

$$m = lastK,$$

so dass der Test $\mathbf{mAux}[b + m] \leq x$ erfolgreich ist und $lastK$ auf $lastK + 1 = k$ gesetzt wird. Im nächsten Schritt hat das Intervall dann die Länge 0 und die Schleife bricht ab. Tatsächlich sind dann alle Elemente des Teilfeldes links von k kleiner-gleich x , der Wert an der Position k ist aber echt größer als x , so dass k nun der gesuchte Index ist.

Abbildung 6.23 zeigt die Implementierung der Methode `merge()`. Der Aufruf `merge(b_1, l_1, b_2, l_2)` hat die Aufgabe, die beiden aufsteigend sortierten Teilfelder

$$[\mathbf{mArray}[b_1], \dots, \mathbf{mArray}[b_1 + (l_1 - 1)]] \quad \text{und} \quad [\mathbf{mArray}[b_2], \dots, \mathbf{mArray}[b_2 + (l_2 - 1)]]$$

so zu mischen, dass das resultierende Teilfeld wiederum aufsteigend sortiert ist. Dabei ist zusätzlich vorausgesetzt, dass das zweite Teilfeld dort beginnt, wo das erste Teilfeld endet, es gilt also

$$b_1 + l_1 = b_2.$$

Zunächst werden beide Teilfelder in das Hilfsfeld \mathbf{mAux} kopiert. An dieser Stelle ist die gezeigte Implementierung noch verbesserungsfähig, in dem Original von Tim Peters wird nur das kleinere Teilfeld in das Hilfsfeld kopiert. Das führt aber zu einer unübersichtlicheren Implementierung.

Der Index c_1 iteriert nun über das erste Teilfeld, während c_2 über das zweite Teilfeld läuft. Der Index d gibt an, wohin das nächste Element geschrieben werden soll. Die innere **do-while**-Schleife, die in Zeile 11 beginnt, mischt die beiden Teilfelder auf konventionelle Weise. Gleichzeitig zählen wir mit, wie oft das nächste Element, das in das Ergebnis-Feld eingefügt \mathbf{mAux} wird, hintereinander aus dem ersten bzw. dem zweiten Teilfeld kommt. Falls wir feststellen, dass ein zusammenhängender Block, der `MIN_GALLOP` oder mehr Elemente enthält aus dem ersten oder zweiten Teilfeld kopiert wird, dann wird die erste **do-while**-Schleife verlassen und das Programm geht in Zeile 26

```

1  private void merge(int b1, int l1, int b2, int l2) {
2      System.arraycopy(mArray, b1, mAux, b1, l1);
3      System.arraycopy(mArray, b2, mAux, b2, l2);
4      int c1 = b1;    // indexes into first  run
5      int c2 = b2;    // indexes into second run
6      int d  = b1;    // destination, index where to write next element
7  outer:
8      while (true) {
9          int n1 = 0; // Number of times in a row that first  run won
10         int n2 = 0; // Number of times in a row that second run won
11         do {
12             if (mAux[c2] < mAux[c1]) {
13                 mArray[d] = mAux[c2];
14                 ++d; ++c2; --l2;
15                 if (l2 == 0) {
16                     break outer;
17                 }
18                 ++n2; n1 = 0;
19             } else { // mArray[c1] <= mAux[c2]
20                 mArray[d] = mAux[c1];
21                 ++d; ++c1; --l1;
22                 if (l1 == 0) { break outer; }
23                 n1++; n2 = 0;
24             }
25         } while (n1 + n2 < MIN_GALLOP);
26         do {
27             n1 = gallop(mAux[c2], c1, l1);
28             if (n1 != 0) {
29                 System.arraycopy(mAux, c1, mArray, d, n1);
30                 d += n1; c1 += n1; l1 -= n1;
31                 if (l1 == 0) { break outer; }
32             }
33             n2 = gallop(mAux[c1], c2, l2);
34             if (n2 != 0) {
35                 System.arraycopy(mAux, c2, mArray, d, n2);
36                 d += n2; c2 += n2; l2 -= n2;
37                 if (l2 == 0) { break outer; }
38             }
39         } while (n1 + n2 >= MIN_GALLOP);
40     } // end of "outer" loop
41     if (l1 == 0) {
42         System.arraycopy(mArray, c2, mArray, d, l2);
43     } else { // l2 == 0
44         System.arraycopy(mAux, c1, mArray, d, l1);
45     }
46 }

```

Abbildung 6.23: The method *merge()*

in den *Gallop*, sprich exponentielle Suche, über. Der Hintergrund ist hier folgender: In der Praxis sind zu sortierende Felder oft *klumpig*, d.h. das Feld enthält Teilfelder der Art, dass beispielsweise alle Elemente des ersten Teilfeldes größer sind als die Elemente des zweiten Teilfeldes. Werden zwei

Teilfelder dieser Art auf konventionelle Art gemischt, so werden der Reihe nach alle Elemente des zweiten Teilfeldes mit dem ersten Element des ersten Teilfeldes verglichen. Dies ist zu aufwendig, denn wenn wir beispielsweise feststellen, dass das letzte Element des zweiten Teilfeldes kleiner ist als das erste Element des ersten Teilfeldes, dann ist kein weiterer Vergleich mehr notwendig, da wir das zweite Teilfeld vor das erste Teilfeld hängen können. Im Allgemeinen ist die Situation, wenn wir in Zeile 26 ankommen wie folgt: Wir wollen die Teilfelder

$$[\mathbf{mAux}[c_1], \dots, \mathbf{mAux}[c_1 + (l_1 - 1)]] \quad \text{und} \quad [\mathbf{mAux}[c_2], \dots, \mathbf{mAux}[c_2 + (l_2 - 1)]]$$

mischen. Dazu bestimmen wir zunächst eine Position n_1 innerhalb des ersten Teilfeldes, so dass alle Elemente links von $c_1 + n_1$ kleiner-gleich dem ersten Element $\mathbf{mAux}[c_2]$ des zweiten Teilfeldes sind,

$$\forall j \in \{c_1, \dots, c_1 + (n_1 - 1)\} : \mathbf{mAux}[j] \leq \mathbf{mAux}[c_2],$$

während das Element an der Position $c_1 + n_1$ größer als $\mathbf{mAux}[c_2]$ ist:

$$\mathbf{mAux}[c_1 + n_1] > \mathbf{mAux}[c_2].$$

Diese Position bestimmen wir durch exponentielle Suche. Dann wissen wir, dass alle die Elemente mit Indizes aus der Menge $\{c_1, \dots, c_1 + (n_1 - 1)\}$ sich bereits an der richtigen Position befinden. Analog bestimmen wir anschließend eine Position $c_2 + n_2$, so dass alle Elemente aus dem zweiten Teilfeld, die links von dieser Position stehen, kleiner-gleich dem ersten Element $\mathbf{mAux}[c_1]$ des verbliebenen ersten Teilfeldes sind. Diese Elemente können dann in einem Block in das Feld **mArray** kopiert werden. Werden bei diesem Verfahren die als Block kopierten Bereiche zu klein, so wechselt der Algorithmus in die konventionelle Methode zum Mischen zurück. Am Ende der äußeren **while**-Schleife sind eventuell noch Elemente in einem der beiden Teilfelder vorhanden, während das andere Teilfeld leer ist. Diese werden dann in Zeile 42 bzw. 44 in das Feld **mArray** kopiert.

6.5.1 Bewertung

Bei Feldern, die bereits weitgehend vorsortiert sind, hat *TimSort*, ähnlich wie “*Sortieren durch Einfügen*”, eine lineare Komplexität. Da in vielen in der Praxis auftretenden Sortierproblemen die zu sortierenden Daten zumindest teilweise vorsortiert sind, ist *TimSort* den anderen Sortier-Algorithmen überlegen.

Historisches Der Quick-Sort-Algorithmus wurde von Charles Antony Richard Hoare [Hoa61] entwickelt, der Merge-Sort-Algorithmus geht auf John von Neumann zurück.

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 $\{\text{true}, \text{false}\}$ darstellen, wenn wir

$$x \in M \Leftrightarrow f(x) = \text{true}$$

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 Einträge 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 $\{\text{Key}, \text{Value}\}$.
3. Die Menge der Funktions-Zeichen ist $\{\text{Map}, \text{find}, \text{insert}, \text{delete}\}$.
4. Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:
 - (a) $\text{Map} : \text{Map}$
Der Aufruf $\text{Map}()$ erzeugt eine leere Abbildung, also eine Abbildung, die keinem Schlüssel einen Wert zuweist.
 - (b) $\text{find} : \text{Map} \times \text{Key} \rightarrow \text{Value} \cup \{\Omega\}$
Der Aufruf $M.\text{find}(k)$ 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 Ω zurück gegeben.
 - (c) $\text{insert} : \text{Map} \times \text{Key} \times \text{Value} \rightarrow \text{Map}$
Der Aufruf $M.\text{insert}(k, v)$ 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) $\text{delete} : \text{Map} \times \text{Key} \rightarrow \text{Map}$
Der Aufruf $M.\text{delete}(k)$ 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) $\text{Map}().\text{find}(k) = \Omega$,
denn der Aufruf $\text{Map}()$ erzeugt eine leere Abbildung.
 - (b) $M.\text{insert}(k, v).\text{find}(k) = v$,
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) $k_1 \neq k_2 \rightarrow M.\text{insert}(k_1, v).\text{find}(k_2) = M.\text{find}(k_2)$,
denn wenn wir für einen Schlüssel eine Wert einfügen, so ändert das nichts an dem Wert, der für einen anderen Schlüssel abgespeichert ist.
 - (d) $M.\text{delete}(k).\text{find}(k) = \Omega$,
denn wenn wir einen Schlüssel löschen, so finden wir anschließend auch keinen Wert mehr unter diesem Schlüssel.
 - (e) $k_1 \neq k_2 \rightarrow M.\text{delete}(k_1).\text{find}(k_2) = M.\text{find}(k_2)$,
denn wenn wir einen Schlüssel löschen, so ändert das nichts an dem Wert, der unter einem anderen Schlüssel abgespeichert ist. □

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:

$$\text{nil} : \mathcal{B} \quad \text{und} \quad \text{node} : \text{Key} \times \text{Value} \times \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}.$$

1. *nil* ist ein binärer Baum.

Dieser Baum wird als der leere Baum bezeichnet.

2. $\text{node}(k, v, l, r)$ 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 $\text{node}(k, v, l, r)$ bezeichnet.

(d) *r* ist ein binärer Baum.

r wird als der rechte Teilbaum von $\text{node}(k, v, l, r)$ 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. $\text{nil} \in \mathcal{B}_{<}$

2. $\text{node}(k, v, l, r) \in \mathcal{B}_{<}$ 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 wie folgt darstellen:

1. Der leere Baum *nil* wird durch einen dicken schwarzen Punkt dargestellt.
2. Ein Baum der Form $\text{node}(k, v, l, r)$ 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 7.1 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 $[n_0, n_1, \dots, n_k]$ von $k+1$ Knoten, die durch Pfeile verbunden sind. Identifizieren wir Knoten mit ihren Markierungen, so ist

$$[\langle 8, 22 \rangle, \langle 12, 18 \rangle, \langle 10, 16 \rangle, \langle 9, 39 \rangle]$$

ein Pfad der Länge 3.

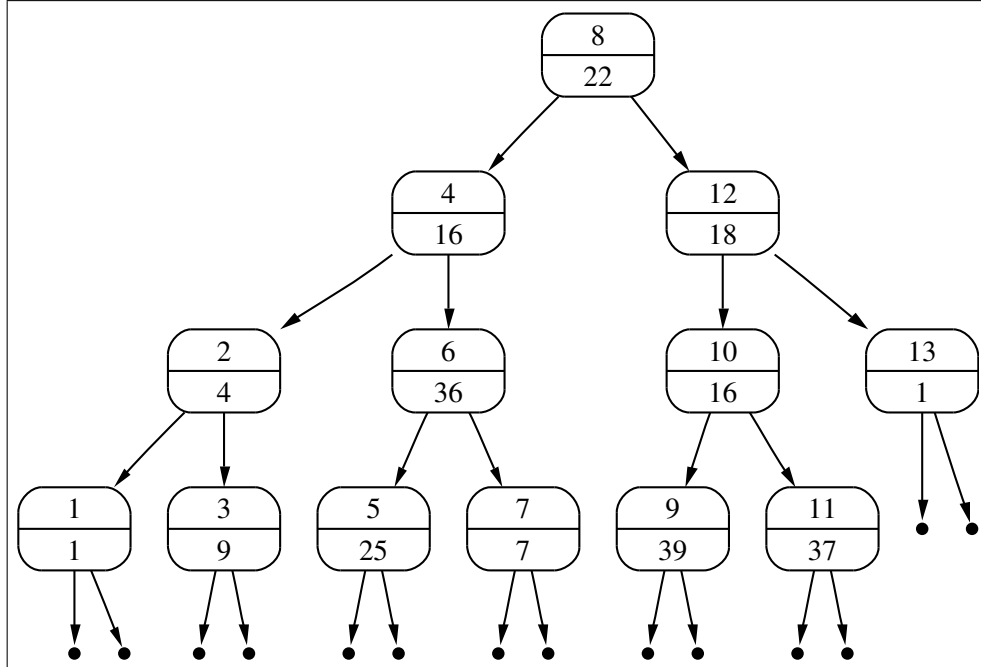


Abbildung 7.1: Ein geordneter binärer Baum

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:

$$\text{Map}() = \text{nil}.$$

Für die Methode *find()* erhalten wir die folgenden Gleichungen:

1. $\text{nil.find}(k) = \Omega$,
denn der leere Baum repräsentiert die leere Abbildung.
2. $\text{node}(k, v, l, r).\text{find}(k) = v$,
denn der Knoten $\text{node}(k, v, l, r)$ speichert die Zuordnung $k \mapsto v$.
3. $k_1 < k_2 \rightarrow \text{node}(k_2, v, l, r).\text{find}(k_1) = l.\text{find}(k_1)$,
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. $k_1 > k_2 \rightarrow \text{node}(k_2, v, l, r).\text{find}(k_1) = r.\text{find}(k_1)$,
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. $nil.insert(k, v) = node(k, v, nil, nil)$,
denn wenn der Baum vorher leer ist, so kann die einzufügende Information direkt an der Wurzel abgespeichert werden.
2. $node(k, v_2, l, r).insert(k, v_1) = node(k, v_1, l, r)$,
denn wenn wir den Schlüssel k an der Wurzel finden, überschreiben wir einfach den zugeordneten Wert.
3. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).insert(k_1, v_1) = node(k_2, v_2, l.insert(k_1, v_1), r)$,
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. $k_1 > k_2 \rightarrow node(k_2, v_2, l, r).insert(k_1, v_1) = node(k_2, v_2, l, r.insert(k_1, v_1))$,
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 $t = node(k, v, l, r)$ 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 $t.delete(k)$ 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 $t = node(k, v, l, r')$ 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 $t = node(k_{min}, v_{min}, l, r')$ 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 7.1 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 7.2 auf Seite 113 zeigt das Ergebnis.

Wir geben nun bedingte Gleichungen an, die die Methode *delMin* spezifizieren.

1. $node(k, v, nil, r).delMin() = [r, k, v]$,
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. $l \neq nil \wedge l.delMin() = [l', k_{min}, v_{min}] \rightarrow$
 $node(k, v, l, r).delMin() = [node(k, v, l', r), k_{min}, v_{min}]$,
denn wenn der linke Teilbaum l in dem binären Baum $t = node(k, v, l, r)$ 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 $t = node(k, v, l', r)$.

Damit können wir nun die Methode *delete*() spezifizieren.

1. $nil.delete(k) = nil$.
2. $node(k, v, nil, r).delete(k) = r$.

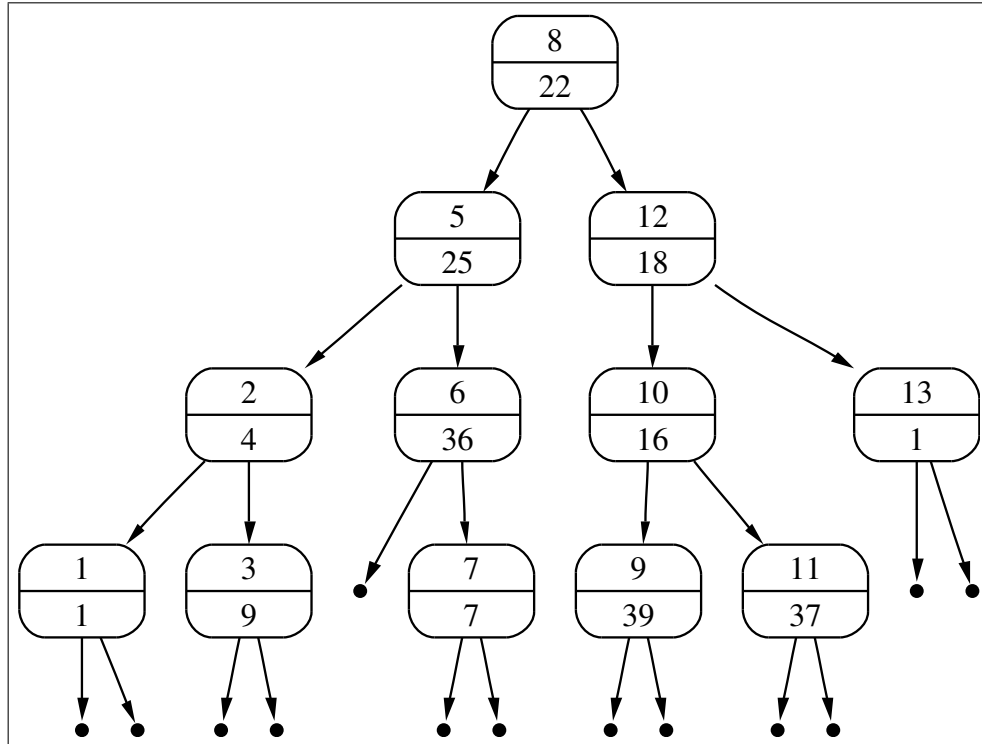


Abbildung 7.2: Der geordnete binärer Baum aus Abbildung 7.1 nach dem Entfernen des Knotens mit der Markierung $\langle 4, 16 \rangle$.

3. $node(k, v, l, nil).delete(k) = l$.
4. $l \neq nil \wedge r \neq nil \wedge r.delMin() = [r', k_{min}, v_{min}] \rightarrow$
 $node(k, v, l, r).delete(k) = node(k_{min}, v_{min}, l, r')$.

Falls der zu entfernende Schlüssel mit dem Schlüssel an der Wurzel des Baums übereinstimmt, entfernen wir mit dem Aufruf $r.delMin()$ 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. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node(k_2, v_2, l.delete(k_1), r)$,

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. $k_1 > k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node(k_2, v_2, l, r.delete(k_1))$,

denn in diesem Fall kann sich der Eintrag mit dem Schlüssel k_1 nur im rechten Teilbaum befinden.

7.2.1 Implementierung geordneter binärer Bäume in Java

Abbildung 7.3 auf Seite 114 zeigt, wie sich der abstrakte Daten-Typ *Map* in *Java* durch ein Interface beschreiben lässt. Das Interface hat den Namen *MyMap*, denn da der ADT *Map* von fundamentaler Bedeutung ist, gibt es in *Java* bereits ein Interface mit dem Namen *Map*. Vergleichen wir die Signaturen der Methoden in dem Interface *MyMap* mit den entsprechenden Signaturen in dem ADT *Map*, so stellen wir fest, dass die Methoden `insert()` und `delete()` in dem Interface

```

1  public interface MyMap<Key extends Comparable<Key>, Value>
2  {
3      public Value find(Key key);
4      public void  insert(Key key, Value value);
5      public void  delete(Key key);
6  }

```

Abbildung 7.3: Das Interface *MyMap*.

den Rückgabewert `void` an Stelle von *MyMap* haben. Anstatt also eine geänderte Map zurück zu geben ändern diese Methoden die Map, mit der sie aufgerufen werden. Dies ist einerseits aus Effizienz-Gründen wichtig und macht andererseits die Verwendung dieser Methoden einfacher.

```

1  public class BinaryTree<Key extends Comparable<Key>, Value>
2      implements MyMap<Key, Value>
3  {
4      Node<Key, Value> mRoot;
5
6      public BinaryTree() {
7          mRoot = new EmptyNode<Key, Value>();
8      }
9      public Value find(Key key) {
10         return mRoot.find(key);
11     }
12     public void insert(Key key, Value value) {
13         mRoot = mRoot.insert(key, value);
14     }
15     public void delete(Key key) {
16         mRoot = mRoot.delete(key);
17     }
18     // Transform the tree into a sorted list.
19     public LinkedList<Key> toList() {
20         return mRoot.toList();
21     }
22 }

```

Abbildung 7.4: Die Klasse *BinaryTree*.

Abbildung 7.4 auf Seite 114 zeigt die Implementierung der Klasse *BinTree* in *Java*. Die Klasse enthält ein Objekt vom Typ *Node* mit dem Namen `mRoot`. Dieser Knoten repräsentiert die Wurzel des binären Baums. Die Methoden der Klasse *BinTree* werden dadurch implementiert, dass die analogen Methoden der abstrakten Klasse *Node* aufgerufen werden. Abbildung 7.5 auf Seite 115 zeigt die Definition der Klasse *Node*. Bei der Betrachtung der Signaturen der Methoden stellen wir fest, dass die Methoden *insert()* und *delete()* nun einen Rückgabewert haben. Dies ist nötig, weil in *Java* ein Objekt seinen Typ nicht ändern kann. Von der Klasse *Node* werden zwei konkrete Klassen abgeleitet: Die Klasse *EmptyNode* repräsentiert einen leeren binären Baum, entspricht also dem *nil* und die Klasse *BinaryNode* dient dazu, einen Knoten der Form $node(k, v, l, r)$ darzustellen. Nun ist es so, dass beim Einfügen aus einem leeren Baum ein nicht-leerer Baum werden kann und umgekehrt kann beim Löschen aus einem nicht-leeren Baum ein leerer Baum werden. Da aber die Methode *insert()* ein Objekt vom Typ *EmptyNode* nicht in ein Objekt *BinaryNode* umwandeln kann, muss die Methoden *insert()* statt dessen den binären Baum, der durch das Einfügen eines

```

1  public abstract class Node<Key extends Comparable<Key>, Value>
2  {
3      public abstract Value find(Key key);
4      public abstract Node<Key, Value> insert(Key key, Value value);
5      public abstract Node<Key, Value> delete(Key key);
6      public abstract boolean isEmpty();
7      public abstract LinkedList<Key> toList();
8
9      abstract Triple<Node<Key, Value>, Key, Value> delMin();
10 }

```

Abbildung 7.5: Die abstrakte Klasse *Node*.

```

1  public class EmptyNode<Key extends Comparable<Key>, Value>
2      extends Node<Key, Value>
3  {
4      public EmptyNode() {}
5
6      public Value find(Key key) {
7          return null;
8      }
9      public Node<Key, Value> insert(Key key, Value value) {
10         return new BinaryNode<Key, Value>(key, value);
11     }
12     public Node<Key, Value> delete(Key key) {
13         return this;
14     }
15     public boolean isEmpty() {
16         return true;
17     }
18     public LinkedList<Key> toList() {
19         return new LinkedList<Key>();
20     }
21     Triple<Node<Key, Value>, Key, Value> delMin() {
22         throw new UnsupportedOperationException();
23     }
24 }

```

Abbildung 7.6: Die Klasse *EmptyNode*.

Schlüssels entsteht, als Ergebnis zurück geben. Analoges gilt für die Methode *delete()*.

Gegenüber den Methoden aus der Klasse *BinaryTree* hat die Klasse *Node* die zusätzliche Methode *isEmpty()* die überprüft, ob der Knoten den leeren Baum repräsentiert. Außerdem gibt es noch die Methode *delMin()*, die den Knoten mit dem kleinsten Schlüssel aus einem Baum löscht. Diese Methode ist nicht als *public* deklariert, da sie nur zur Implementierung der Methode *delete()* benutzt wird.

Abbildung 7.6 zeigt die Definition der Klasse *EmptyNode*. Bemerkenswert ist hier die Implementierung der Methode *delMin()*: Da der leere Baum keine Schlüssel enthält, macht die Methode *delMin()* keinen Sinn. Daher produziert der Aufruf dieser Methode eine Ausnahme.

Algorithmisch interessant ist die Implementierung der Klasse *BinaryNode*, die in den Abbildun-

```

1  public class BinaryNode<Key extends Comparable<Key>, Value>
2      extends Node<Key, Value>
3  {
4      private Key          mKey;          /**< The key stored at the root. */
5      private Value        mValue;        /**< The value attached to this key. */
6      private Node<Key, Value> mLeft;      /**< The left subtree. */
7      private Node<Key, Value> mRight;     /**< The right subtree. */
8
9      public BinaryNode(Key key, Value value) {
10         mKey   = key;
11         mValue = value;
12         mLeft  = new EmptyNode<Key, Value>();
13         mRight = new EmptyNode<Key, Value>();
14     }
15     public Value find(Key key) {
16         int cmp = key.compareTo(mKey);
17         if (cmp < 0) {                // key < mKey
18             return mLeft.find(key);
19         } else if (cmp > 0) {           // key > mKey
20             return mRight.find(key);
21         } else {                       // key == mKey
22             return mValue;
23         }
24     }
25     public Node<Key, Value> insert(Key key, Value value) {
26         int cmp = key.compareTo(mKey);
27         if (cmp < 0) {                 // key < mKey
28             mLeft = mLeft.insert(key, value);
29         } else if (cmp > 0) {           // key > mKey
30             mRight = mRight.insert(key, value);
31         } else {                       // key == mKey
32             mValue = value;
33         }
34         return this;
35     }

```

Abbildung 7.7: Die Klasse *BinaryNode*, Teil I.

gen 7.7 und 7.8 auf den Seiten 116 und 117 gezeigt wird. Die Klasse enthält vier Member-Variablen:

1. `mKey` ist der Schlüssel, der an diesem Knoten abgespeichert wird.
2. `mValue` ist der dem Schlüssel `mKey` zugeordnete Wert.
3. `mLeft` ist der linke Teilbaum.
4. `mRight` ist der rechte Teilbaum.

Ein Objekt *o* der Klasse *BinaryNode* entspricht also dem Term

node(o.mKey, o.mValue, o.mLeft, o.mRight).

Die Implementierung der Methoden *find()*, *insert()* und *delete()* setzt die bedingten Gleichungen, mit denen wir diese Methoden spezifiziert haben, eins-zu-eins um.

```

36     public Node<Key, Value> delete(Key key) {
37         int cmp = key.compareTo(mKey);
38         if (cmp == 0) {
39             if (mLeft.isEmpty()) {
40                 return mRight;
41             }
42             if (mRight.isEmpty()) {
43                 return mLeft;
44             }
45             Triple triple = mRight.delMin();
46             mRight = triple.getFirst();
47             mKey    = triple.getSecond();
48             mValue  = triple.getThird();
49         }
50         if (cmp < 0) {
51             mLeft = mLeft.delete(key);
52         }
53         if (cmp > 0) {
54             mRight = mRight.delete(key);
55         }
56         return this;
57     }
58     public boolean isEmpty() {
59         return false;
60     }
61     Triple delMin() {
62         if (mLeft.isEmpty()) {
63             return new Triple<Node<Key, Value>, Key, Value>(mRight, mKey, mValue);
64         } else {
65             Triple<Node<Key, Value>, Key, Value> t = mLeft.delMin();
66             mLeft = t.getFirst();
67             Key   key   = t.getSecond();
68             Value value = t.getThird();
69             return new Triple<Node<Key, Value>, Key, Value>(this, key, value);
70         }
71     }
72 }

```

Abbildung 7.8: Die Klasse *BinaryNode*, Teil II.

Abbildung 7.9 auf Seite 118 zeigt schließlich die Implementierung der Klasse *Triple*. Diese Klasse dient dazu ein Tripel darzustellen und hat folglich drei Member-Variablen *mFirst*, *mSecond*, *mThird*, die jeweils die erste, zweite und dritte Komponente des Tripels abspeichern.

```

73 public class Triple<First, Second, Third>
74 {
75     First mFirst;
76     Second mSecond;
77     Third mThird;
78
79     Triple(First first, Second second, Third value) {
80         mFirst = first;
81         mSecond = second;
82         mThird = value;
83     }
84
85     First getFirst() { return mFirst; }
86     Second getSecond() { return mSecond; }
87     Third getThird() { return mThird; }
88 }

```

Abbildung 7.9: Die Klasse *Triple*.

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 7.10 zeigt den geordneten binären Baum der dann entsteht, wenn die Paare aus Schlüssel und Werten aus der Abbildung 7.1 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.

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 $\mathcal{O}(\ln(n))$ Vergleiche erforderlich sind. Wir werden diese Behauptung nun beweisen.

Wir bezeichnen die durchschnittliche Anzahl von Vergleichen, die beim Aufruf $b.find(k)$ 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

$$b = \text{node}(k', v, l, r).$$

Ordnen wir die $n + 1$ Schlüssel der Größe nach in der Form

$$k_0 < k_1 < \dots < k_i < k_{i+1} < k_{i+2} < \dots < k_{n-1} < k_n,$$

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:

$$\underbrace{k_0 < k_1 < \dots < k_{i-1}}_{\text{Schlüssel in } l} < \underbrace{k_i}_{\substack{\parallel \\ k'}} < \underbrace{k_{i+1} < \dots < k_{n-1} < k_n}_{\text{Schlüssel in } r},$$

Da b insgesamt $n + 1$ Schlüssel enthält, gibt es $n + 1$ Möglichkeiten, wie die verbleibenden n

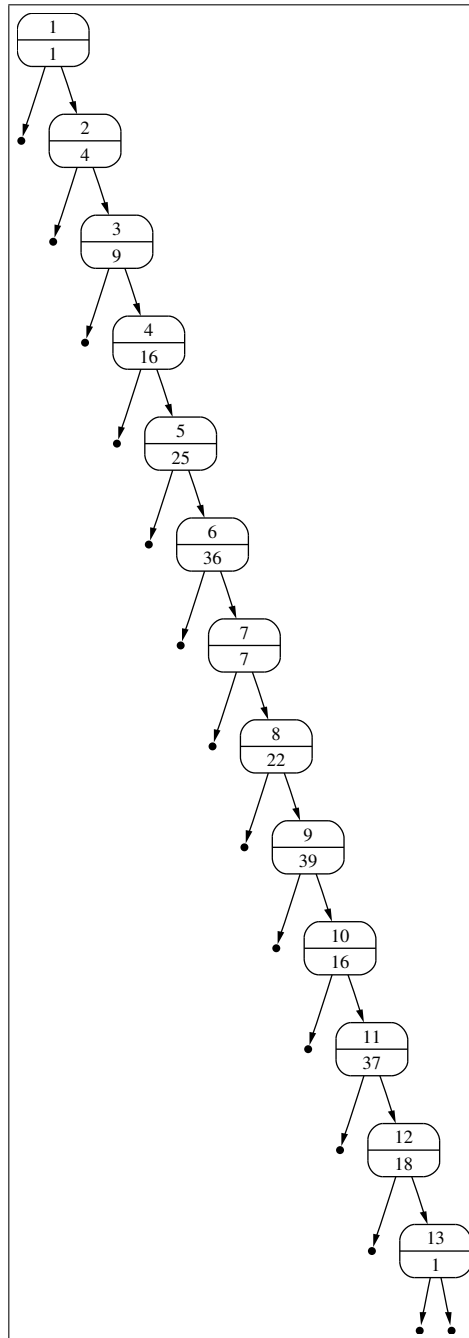


Abbildung 7.10: Ein entarteter geordneter binärer Baum.

Schlüssel auf die beiden Teilbäume l und r verteilt sein können, denn l kann i Schlüssel enthalten, wobei

$$i \in \{0, 1, \dots, n\}$$

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

$$\text{anzVgl}(i, n+1),$$

so gilt

$$d_{n+1} = \sum_{i=0}^n \frac{1}{n+1} \cdot \text{anzVgl}(i, n+1),$$

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 $\text{anzVgl}(i, n+1)$: 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 $b.\text{find}(k)$ 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 $\text{anzVgl}(i, n+1)$ 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

$$\begin{aligned} \text{anzVgl}(i, n+1) &= \frac{1}{n+1} \cdot 1 + \frac{i}{n+1} \cdot (d_i + 1) + \frac{n-i}{n+1} \cdot (d_{n-i} + 1) \\ &= \frac{1}{n+1} \cdot (1 + i \cdot (d_i + 1) + (n-i) \cdot (d_{n-i} + 1)) \\ &= \frac{1}{n+1} \cdot (1 + i + (n-i) + i \cdot d_i + (n-i) \cdot d_{n-i}) \\ &= \frac{1}{n+1} \cdot (n+1 + i \cdot d_i + (n-i) \cdot d_{n-i}) \\ &= 1 + \frac{1}{n+1} \cdot (i \cdot d_i + (n-i) \cdot d_{n-i}) \end{aligned}$$

Damit können wir nun die Rekurrenz-Gleichung für d_{n+1} aufstellen:

$$\begin{aligned}
d_{n+1} &= \sum_{i=0}^n \frac{1}{n+1} \cdot \text{anzVgl}(i, n+1) \\
&= \frac{1}{n+1} \cdot \sum_{i=0}^n \left(1 + \frac{1}{n+1} \cdot (i \cdot d_i + (n-i) \cdot d_{n-i}) \right) \\
&= \frac{1}{n+1} \cdot \left(\underbrace{\sum_{i=0}^n 1}_{n+1} + \frac{1}{n+1} \cdot \sum_{i=0}^n (i \cdot d_i + (n-i) \cdot d_{n-i}) \right) \\
&= 1 + \frac{1}{(n+1)^2} \cdot \left(\sum_{i=0}^n (i \cdot d_i + (n-i) \cdot d_{n-i}) \right) \\
&= 1 + \frac{2}{(n+1)^2} \cdot \sum_{i=0}^n i \cdot d_i
\end{aligned}$$

Bei der letzten Umformung haben wir die Gleichung (6.4)

$$\sum_{i=0}^n f(n-i) = \sum_{i=0}^n f(i)$$

benutzt, die wir bei der Analyse der Komplexität von Quick-Sort gezeigt hatten. Wir lösen jetzt die Rekurrenz-Gleichung

$$d_{n+1} = 1 + \frac{2}{(n+1)^2} \cdot \sum_{i=0}^n i \cdot d_i \quad (7.1)$$

mit der Anfangs-Bedingungen $d_1 = 1$. Zur Lösung von Gleichung (7.1) führen wir die Substitution $n \mapsto n+1$ durch und erhalten

$$d_{n+2} = 1 + \frac{2}{(n+2)^2} \cdot \sum_{i=0}^{n+1} i \cdot d_i \quad (7.2)$$

Wir multiplizieren nun Gleichung (7.1) mit $(n+1)^2$ und Gleichung (7.2) mit $(n+2)^2$ und finden die Gleichungen

$$(n+1)^2 \cdot d_{n+1} = (n+1)^2 + 2 \cdot \sum_{i=0}^n i \cdot d_i, \quad (7.3)$$

$$(n+2)^2 \cdot d_{n+2} = (n+2)^2 + 2 \cdot \sum_{i=0}^{n+1} i \cdot d_i \quad (7.4)$$

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:

$$\begin{aligned} c_{n+1} &= c_0 + \sum_{i=0}^n \frac{3}{i+2} - \sum_{i=0}^n \frac{1}{i+1} \\ &= \sum_{i=2}^{n+2} \frac{3}{i} - \sum_{i=1}^{n+1} \frac{1}{i}. \end{aligned}$$

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

$$d_n = 2 \cdot \ln(n) + \mathcal{O}(1).$$

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. $nil.height() = 0$.
2. $node(k, v, l, r).height() = 1 + \max(l.height(), r.height())$.

Definition 17 (AVL-Baum)

Wir definieren die Menge \mathcal{A} der AVL-Bäume induktiv:

1. $nil \in \mathcal{A}$.
2. $node(k, v, l, r) \in \mathcal{A}$ g.d.w.
 - (a) $node(k, v, l, r) \in \mathcal{B}_<$,
 - (b) $l, r \in \mathcal{A}$ und
 - (c) $|l.height() - r.height()| \leq 1$.

Diese Bedingungen bezeichnen wir auch als die *Balancierungs-Bedingung*.

AVL-Bäume sind also geordnete binäre Bäume, für die sich an jedem Knoten $node(k, v, l, r)$ die Höhen der Teilbäume l und r maximal um 1 unterscheiden. \square

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

$$restore : \mathcal{B}_< \rightarrow \mathcal{A},$$

mit der wir die Bedingung über den Höhenunterschied von Teilbäumen wiederherstellen können, wenn diese beim Einfügen oder Löschen eines Elements verletzt wird. Der Aufruf $b.restore()$ 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 $b.restore()$ liegt also einer der beiden folgenden Fälle vor:

1. $b = nil$ oder
2. $b = node(k, v, l, r) \wedge l \in \mathcal{A} \wedge r \in \mathcal{A} \wedge |l.height() - r.height()| \leq 2$.

Wir spezifizieren die Methode $restore()$ durch bedingte Gleichungen.

1. $nil.restore() = nil$,
denn der leere Baum ist ein AVL-Baum.
2. $|l.height() - r.height()| \leq 1 \rightarrow node(k, v, l, r).restore() = node(k, v, l, r)$,
denn wenn die Balancierungs-Bedingung bereits erfüllt ist, braucht nichts getan werden.
3. $l_1.height() = r_1.height() + 2$
 $\wedge l_1 = node(k_2, v_2, l_2, r_2)$
 $\wedge l_2.height() \geq r_2.height()$
 $\rightarrow node(k_1, v_1, l_1, r_1).restore() = node(k_2, v_2, l_2, node(k_1, v_1, r_2, r_1))$

Um diese Gleichung zu verstehen, betrachten wir Abbildung 7.11 auf Seite 124. Der linke Teil der Abbildung beschreibt die Situation vor dem Ausbalancieren, es wird also der Baum

$$\text{node}(k_1, v_1, \text{node}(k_2, v_2, l_2, r_2), r_1)$$

dargestellt. Der rechte Teil der Abbildung zeigt das Ergebnis des Ausbalancierens, es wird also der Baum

$$\text{node}(k_2, v_2, l_2, \text{node}(k_1, v_1, r_2, r_1))$$

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 ein Element gelöscht wird.

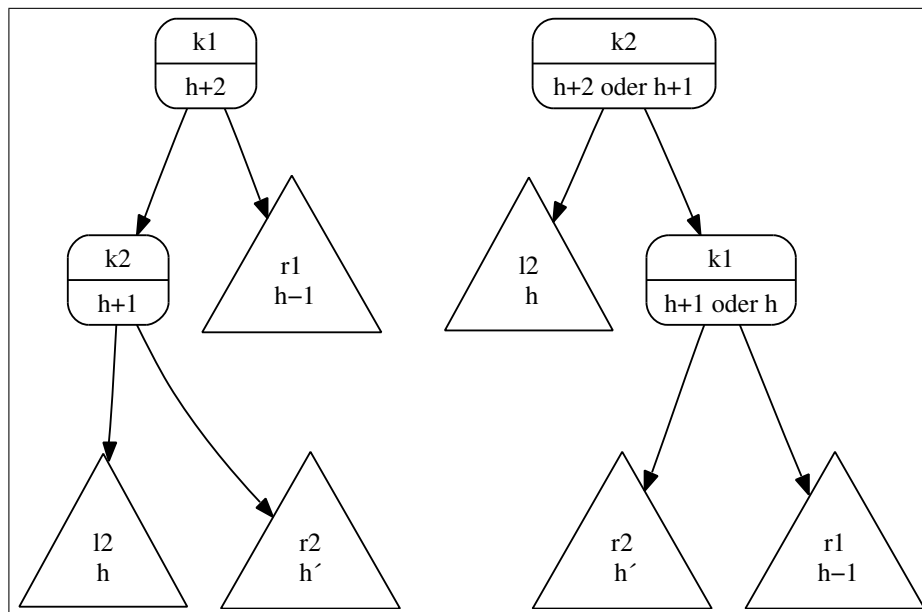


Abbildung 7.11: Ein unbalancierter Baum und der rebalancierte Baum

Wir müssen uns davon überzeugen, dass der im rechten Teil von Abbildung 7.11 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:

$$l_2 < k_2 < r_2 < k_1 < r_1. \quad (\star)$$

Dabei schreiben wir für einen Schlüssel k und einen Baum b

$$k < b$$

um auszudrücken, 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. $l_1.height() = r_1.height() + 2$
 $\wedge l_1 = node(k_2, v_2, l_2, r_2)$
 $\wedge l_2.height() < r_2.height()$
 $\wedge r_2 = node(k_3, v_3, l_3, r_3)$
 $\rightarrow node(k_1, v_1, l_1, r_1).restore() = node(k_3, v_3, node(k_2, v_2, l_2, l_3), node(k_1, v_1, r_3, r_1))$

Die linke Seite der Gleichung wird durch die Abbildung 7.12 auf Seite 126 illustriert. Dieser Baum kann in der Form

$$node(k_1, v_1, node(k_2, v_2, l_2, node(k_3, v_3, l_3, r_3)), r_1)$$

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.

Die Situation der rechten Seite der obigen Gleichung zeigt Abbildung 7.13 auf Seite 127. Der auf dieser Abbildung gezeigte Baum hat die Form

$$node(k_3, v_3, node(k_2, v_2, l_2, l_3), node(k_1, v_1, r_3, r_1)).$$

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.

5. $r_1.height() = l_1.height() + 2$
 $\wedge r_1 = node(k_2, v_2, l_2, r_2)$
 $\wedge r_2.height() \geq l_2.height()$
 $\rightarrow node(k_1, v_1, l_1, r_1).restore() = node(k_2, v_2, node(k_1, v_1, l_1, l_2), r_2)$
6. $r_1.height() = l_1.height() + 2$
 $\wedge r_1 = node(k_2, v_2, l_2, r_2)$
 $\wedge r_2.height() < l_2.height()$
 $\wedge l_2 = node(k_3, v_3, l_3, r_3)$
 $\rightarrow node(k_1, v_1, l_1, r_1).restore() = node(k_3, v_3, node(k_1, v_1, l_1, l_3), node(k_2, v_2, r_3, r_2))$

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. $nil.insert(k, v) = node(k, v, nil, nil).$
2. $node(k, v_2, l, r).insert(k, v_1) = node(k, v_1, l, r).$
3. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).insert(k_1, v_1) = node(k_2, v_2, l.insert(k_1, v_1), r).restore().$
4. $k_1 > k_2 \rightarrow node(k_2, v_2, l, r).insert(k_1, v_1) = node(k_2, v_2, l, r.insert(k_1, v_1)).restore().$

Analog ändern sich die Gleichungen für `delMin()` wie folgt:

1. $node(k, v, nil, r).delMin() = \langle r, k, v \rangle.$
2. $l \neq nil \wedge l.delMin() = \langle l', k_{min}, v_{min} \rangle \rightarrow$
 $node(k, v, l, r).delMin() = \langle node(k, v, l', r).restore(), k_{min}, v_{min} \rangle.$

Damit können wir die Gleichungen zur Spezifikation der Methode `delete()` angeben.

1. $nil.delete(k) = nil.$

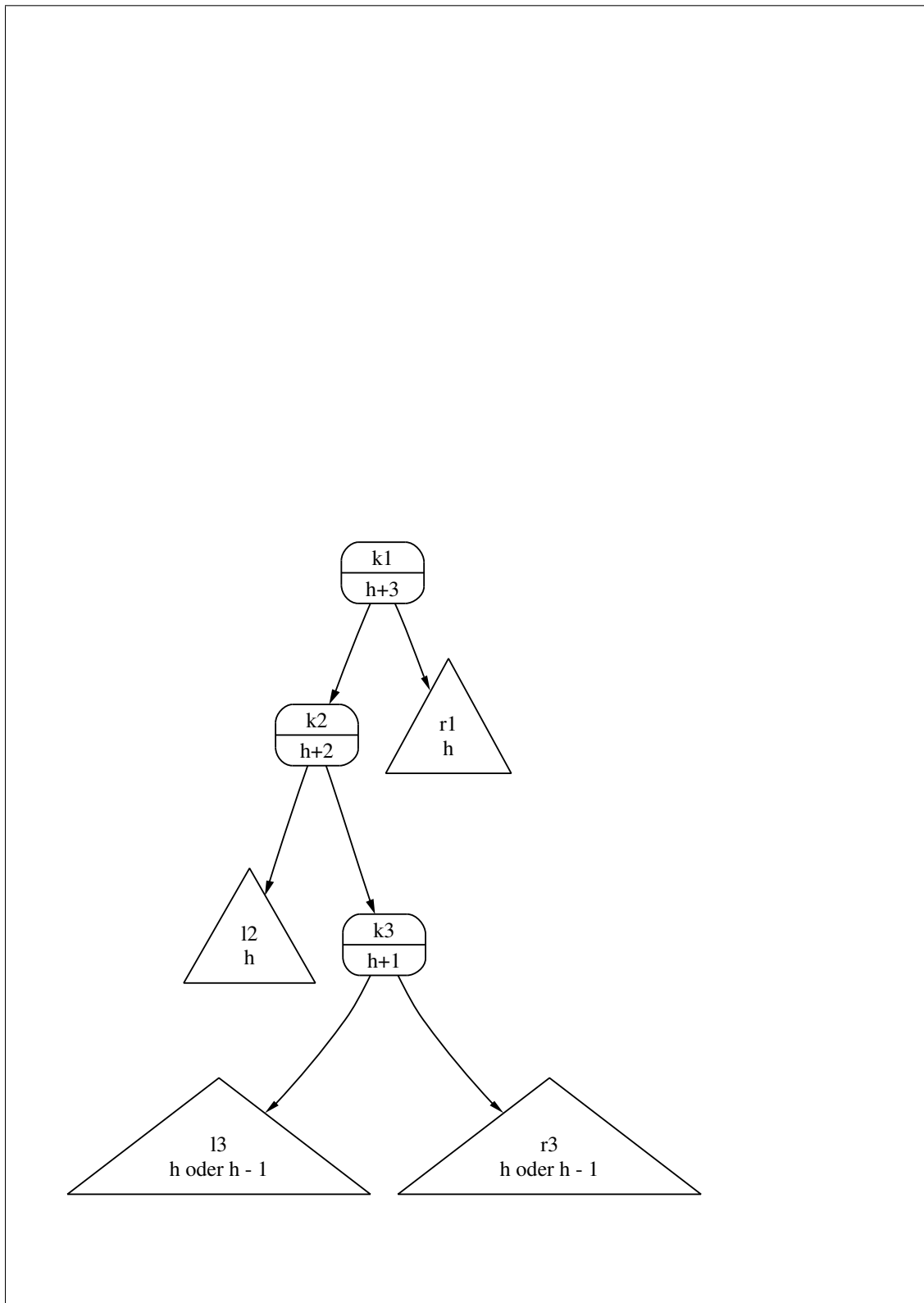


Abbildung 7.12: Ein unbalancierter Baum: 2. Fall

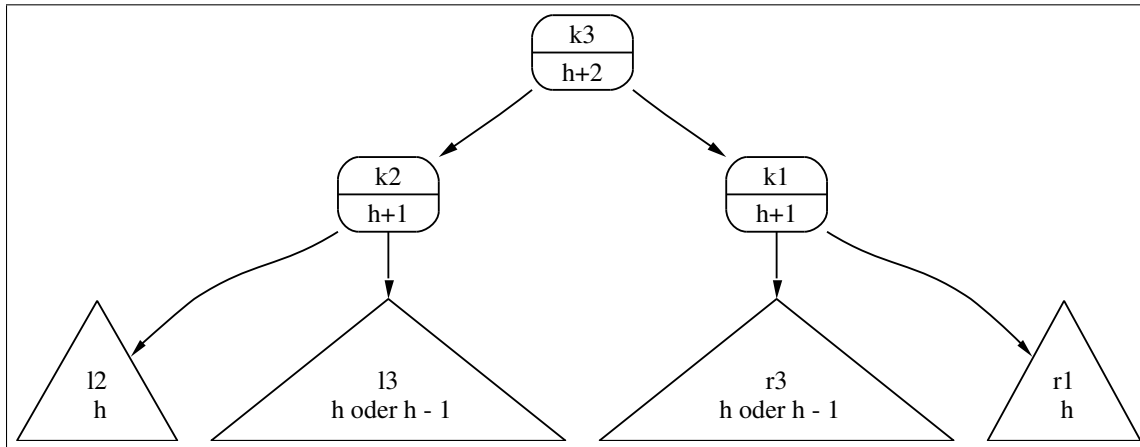


Abbildung 7.13: Der rebalancierte Baum im 2. Fall

1

2. $node(k, v, nil, r).delete(k) = r.$
3. $node(k, v, l, nil).delete(k) = l.$
4. $l \neq nil \wedge r \neq nil \wedge r.delMin() = \langle r', k_{min}, v_{min} \rangle \rightarrow$
 $node(k, v, l, r).delete(k) = node(k_{min}, v_{min}, l, r').restore().$
5. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node(k_2, v_2, l.delete(k_1), r).restore().$
6. $k_1 > k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node(k_2, v_2, l, r.delete(k_1)).restore().$

7.3.1 Implementierung von AVL-Bäumen in Java

```

1  public class AVLTree<Key extends Comparable<Key>, Value>
2      implements MyMap<Key, Value>
3  {
4      Node<Key, Value> mRoot;
5
6      public AVLTree() {
7          mRoot = new EmptyNode<Key, Value>();
8      }
9      public Value find(Key key) {
10         return mRoot.find(key);
11     }
12     public void insert(Key key, Value value) {
13         mRoot = mRoot.insert(key, value);
14     }
15     public void delete(Key key) {
16         mRoot = mRoot.delete(key);
17     }
18 }

```

Abbildung 7.14: Die Klasse *AVLTree*.

Abbildung 7.14 auf Seite 127 zeigt die Implementierung der Klasse *AVLTree*. Gegenüber der Implementierung der Klasse *BinaryTree* aus Abbildung 7.4 auf Seite 114 wurde hier nur der Name der Klasse geändert.

```

1  public abstract class Node<Key extends Comparable<Key>, Value>
2  {
3      protected int mHeight; // the height of the tree
4
5      public abstract Value find(Key key);
6      public abstract Node<Key, Value> insert(Key key, Value value);
7      public abstract Node<Key, Value> delete(Key key);
8      public abstract boolean      isEmpty();
9
10     abstract Triple<Node<Key, Value>, Key, Value> delMin();
11     abstract void                                restore();
12 }

```

Abbildung 7.15: Die abstrakte Klasse *Node*.

Abbildung 7.15 auf Seite 128 zeigt die neue Implementierung der abstrakten Klasse *Node*. Gegenüber der Implementierung in Abbildung 7.5 auf Seite 115 ist hier einerseits in Zeile 3 die Member-Variable *mHeight* neu hinzu gekommen und andererseits gibt es jetzt in Zeile 11 die Methode *restore()*, mit deren Hilfe sich die Balancierungs-Bedingung wiederherstellen lässt, wenn diese durch eine Einfüge- oder Lösch-Operation verletzt worden ist.

```

1  public class EmptyNode<Key extends Comparable<Key>, Value>
2      extends Node<Key, Value>
3  {
4      public EmptyNode() {
5          mHeight = 0;
6      }
7      public Value find(Key key) {
8          return null;
9      }
10     public Node<Key, Value> insert(Key key, Value value) {
11         return new BinaryNode<Key, Value>(key, value);
12     }
13     public Node<Key, Value> delete(Key key) {
14         return this;
15     }
16     public boolean isEmpty() {
17         return true;
18     }
19     Triple<Node<Key, Value>, Key, Value> delMin() {
20         throw new UnsupportedOperationException();
21     }
22     void restore() {}
23 }

```

Abbildung 7.16: Die Klasse *EmptyNode*.

Abbildung 7.16 auf Seite 128 zeigt die Implementierung der Klasse *EmptyNode*. Gegenüber

der in Abbildung 7.6 auf Seite 115 gezeigten Implementierung gibt es zwei kleine Änderungen:

1. In Zeile 5 setzt der Konstruktor die Höhe `mHeight` auf 0.
2. In Zeile 22 ist die Methode `restore()` implementiert. Diese Implementierung ist für einen leeren Knoten trivial.

Die Abbildungen 7.17, 7.18 und 7.19 auf Seite 130 und den folgenden Seiten zeigen die Implementierung der Klasse `BinaryNode`. Gegenüber der entsprechenden Implementierung in den Abbildungen 7.7 und 7.8 auf den Seiten 116 und 117 gibt es die folgenden Änderungen:

1. In dem ersten Konstruktor wird in Zeile 14 die Member-Variable `mHeight` auf 1 gesetzt.
2. In Zeile 16 – 23 haben wir einen neuen Konstruktor, der neben einem Schlüssel und einem Wert als zusätzliche Argumente noch den linken und den rechten Teilbaum des neu zu erstellenden Knoten erhält. Dieser Konstruktor wird später für die Implementierung der Methode `restore()` benötigt.
3. Die Implementierung von `find()` hat sich gegenüber der alten Implementierung nicht verändert, den jeder AVL-Baum ist ja auch ein geordneter binärer Baum.
4. Am Ende der Methode `insert()` wird in Zeile 43 die Methode `restore()` aufgerufen um die Balancierungs-Bedingung sicherzustellen.

Eigentlich müsste die Methode `restore()` nur dann aufgerufen werden, wenn entweder im linken oder im rechten Teilbaum ein neuer Schlüssel eingefügt wird. Es würde also reichen, die Methode am Ende der `if`-Blöcke in Zeile 37 und 39 aufzurufen. Dann hätten wir aber zwei Aufrufe von `restore()`. Der Code wird übersichtlicher, wenn `restore()` am Ende der Methode aufgerufen wird.

5. Genauso wird in Zeile 66 vor der Beendigung der Methode `delete()` die Methode `restore()` aufgerufen.

Entscheidend ist hier zu bemerken, dass sich die Implementierung der beiden Methoden `insert()` und `delete()` gegenüber der Implementierung, die wir für geordnete binäre Bäume verwendet haben, nur an einer einzigen Stelle geändert hat: Wir müssen nur vor dem `return`-Befehl `restore()` aufrufen.

6. Auch die Implementierung von `delMin()` unterscheidet sich von der alten Implementierung nur durch den Aufruf von `restore()` in Zeile 80.

7. In Zeile 84 implementieren wir die Methode `restore()`.

(a) Falls die Balancierungs-Bedingung bereits erfüllt ist, so muss die Methode `restore()` nur dafür sorgen, dass die Member-Variable `mHeight` an dem Knoten korrekt gesetzt ist. Dazu wird die Methode `restoreHeight()` aufgerufen. Diese Methode ist in Zeile 120 – 122 implementiert und berechnet die Höhe neu.

(b) Falls die Höhe des linken Teilbaums nun größer als die Höhe des rechten Teilbaums ist und außerdem die Balancierungs-Bedingung verletzt ist, dann gibt es eine weitere Fall-Unterscheidung, die wir bei der Herleitung der bedingten Gleichungen zur Spezifikation der Methode `restore()` bereits diskutiert hatten. Diese beiden Fälle werden in Zeile 89 – 113 behandelt.

In den Zeilen 98 – 102 behandeln wir den Fall, der in Abbildung 7.11 gezeigt ist, während die Zeilen 104 – 112 den Fall behandeln, der in den Abbildungen 7.12 und 7.13 dargestellt wird.

(c) Der Code, der den Fall betrachtet, in dem einerseits die Höhe des rechten Teilbaums größer als die Höhe des linken Teilbaums ist und andererseits die Balancierungs-Bedingung verletzt ist, ist völlig analog zu dem vorigen Fall und wird deshalb in der Abbildung nicht explizit wiedergegeben.

```

1  public class BinaryNode<Key extends Comparable<Key>, Value>
2      extends Node<Key, Value>
3  {
4      private Key          mKey;
5      private Value        mValue;
6      private Node<Key, Value> mLeft;
7      private Node<Key, Value> mRight;
8
9      public BinaryNode(Key key, Value value) {
10         mKey    = key;
11         mValue  = value;
12         mLeft   = new EmptyNode<Key, Value>();
13         mRight  = new EmptyNode<Key, Value>();
14         mHeight = 1;
15     }
16     public BinaryNode(Key key, Value value, Node<Key, Value> left,
17                     Node<Key, Value> right) {
18         mKey    = key;
19         mValue  = value;
20         mLeft   = left;
21         mRight  = right;
22         mHeight = 1 + Math.max(mLeft.mHeight, mRight.mHeight);
23     }
24     public Value find(Key key) {
25         int cmp = key.compareTo(mKey);
26         if (cmp < 0) {                // key < mKey
27             return mLeft.find(key);
28         } else if (cmp > 0) {          // key > mKey
29             return mRight.find(key);
30         } else {                      // key == mKey
31             return mValue;
32         }
33     }
34     public Node<Key, Value> insert(Key key, Value value) {
35         int cmp = key.compareTo(mKey);
36         if (cmp < 0) {                // key < mKey
37             mLeft = mLeft.insert(key, value);
38         } else if (cmp > 0) {          // key > mKey
39             mRight = mRight.insert(key, value);
40         } else {                      // key == mKey
41             mValue = value;
42         }
43         restore();
44         return this;
45     }

```

Abbildung 7.17: Die Klasse *BinaryNode*, Teil I.

- (d) In Zeile 118 wird am Ende der Methode *restore()* noch dafür gesorgt, dass die Member-Variable *mHeight* an dem Knoten aktualisiert wird.

```

46     public Node<Key, Value> delete(Key key) {
47         int cmp = key.compareTo(mKey);
48         if (cmp == 0) {
49             if (mLeft.isEmpty()) {
50                 return mRight;
51             }
52             if (mRight.isEmpty()) {
53                 return mLeft;
54             }
55             Triple<Node<Key, Value>, Key, Value> triple = mRight.delMin();
56             mRight = triple.getFirst();
57             mKey    = triple.getSecond();
58             mValue  = triple.getThird();
59         }
60         if (cmp < 0) {
61             mLeft = mLeft.delete(key);
62         }
63         if (cmp > 0) {
64             mRight = mRight.delete(key);
65         }
66         restore();
67         return this;
68     }
69     public boolean isEmpty() {
70         return false;
71     }
72     Triple<Node<Key, Value>, Key, Value> delMin() {
73         if (mLeft.isEmpty()) {
74             return new Triple(mRight, mKey, mValue);
75         } else {
76             Triple<Node<Key, Value>, Key, Value> t = mLeft.delMin();
77             mLeft = t.getFirst();
78             Key   key   = t.getSecond();
79             Value value = t.getThird();
80             restore();
81             return new Triple(this, key, value);
82         }
83     }

```

Abbildung 7.18: Die Klasse *BinaryNode*, Teil II.

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 dasselbe 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

```

84     void restore() {
85         if (Math.abs(mLeft.mHeight - mRight.mHeight) <= 1) {
86             restoreHeight();
87             return;
88         }
89         if (mLeft.mHeight > mRight.mHeight) {
90             Key k1 = mKey;
91             Value v1 = mValue;
92             BinaryNode<Key, Value> l1 = (BinaryNode<Key, Value>) mLeft;
93             Node<Key, Value> r1 = mRight;
94             Key k2 = l1.mKey;
95             Value v2 = l1.mValue;
96             Node<Key, Value> l2 = l1.mLeft;
97             Node<Key, Value> r2 = l1.mRight;
98             if (l2.mHeight >= r2.mHeight) {
99                 mKey = k2;
100                mValue = v2;
101                mLeft = l2;
102                mRight = new BinaryNode<Key, Value>(k1, v1, r2, r1);
103            } else {
104                BinaryNode<Key, Value> rb2 = (BinaryNode<Key, Value>) r2;
105                Key k3 = rb2.mKey;
106                Value v3 = rb2.mValue;
107                Node<Key, Value> l3 = rb2.mLeft;
108                Node<Key, Value> r3 = rb2.mRight;
109                mKey = k3;
110                mValue = v3;
111                mLeft = new BinaryNode<Key, Value>(k2, v2, l2, l3);
112                mRight = new BinaryNode<Key, Value>(k1, v1, r3, r1);
113            }
114        }
115        if (mRight.mHeight > mLeft.mHeight) {
116            :
117        }
118        restoreHeight();
119    }
120    void restoreHeight() {
121        mHeight = 1 + Math.max(mLeft.mHeight, mRight.mHeight);
122    }
123 }

```

Abbildung 7.19: Die Klasse *BinaryNode*, Teil III.

wir noch eine Hilfs-Funktion *maxKey()* mit der Signatur

$$\text{maxKey} : \mathcal{B}_{<} \rightarrow \text{Key}$$

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. $\text{node}(k, v, l, \text{nil}).\text{maxKey}() = k,$
2. $r \neq \text{nil} \rightarrow \text{node}(k, v, l, r).\text{maxKey}() = r.\text{maxKey}().$

Damit können wir nun die Bäume $b_h(k)$ durch Induktion nach der Höhe h definieren.

1. $b_0(k) = \text{nil}$,

denn es gibt genau einen AVL-Baum der Höhe 0 und dieser enthält keinen Schlüssel.

2. $b_1(k) = \text{node}(k + 1, 0, \text{nil}, \text{nil})$,

denn es gibt genau einen AVL-Baum der Höhe 1.

3. $b_{h+1}(k).\text{maxKey}() = l \rightarrow b_{h+2}(k) = \text{node}(l + 1, 0, b_{h+1}(k), b_h(l + 1))$,

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. $c_0 = \#b_0(k) = \#\text{nil} = 0$,

2. $c_1 = \#b_1(k) = \#\text{node}(k + 1, 0, \text{nil}, \text{nil}) = 1$,

3.
$$\begin{aligned} c_{h+2} &= \#b_{h+2}(k) \\ &= \#\text{node}(l + 1, 0, b_{h+1}(k), b_h(l + 1)) \\ &= \#b_{h+1}(k) + \#b_h(l + 1) + 1 \\ &= c_{h+1} + c_h + 1. \end{aligned}$$

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:

$$\begin{aligned} c_h &= \frac{1}{\sqrt{5}} (\lambda_1^{h+2} - \lambda_2^{h+2}) - 1 \quad \text{mit} \\ \lambda_1 &= \frac{1}{2}(1 + \sqrt{5}) \approx 1.62 \quad \text{und} \quad \lambda_2 = \frac{1}{2}(1 - \sqrt{5}) \approx -0.62. \end{aligned}$$

Da $|\lambda_2| < 1$ ist, spielt der Wert λ_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

$$\log_2(n + 1) = (h + 2) \cdot \log_2(\lambda_1) - \frac{1}{2} \cdot \log_2(5)$$

Daraus folgt nach Addition von $\frac{1}{2} \cdot \log_2(5)$

$$\log_2(n + 1) + \frac{1}{2} \cdot \log_2(5) = (h + 2) \cdot \log_2(\lambda_1)$$

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 7.20 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 7.10.

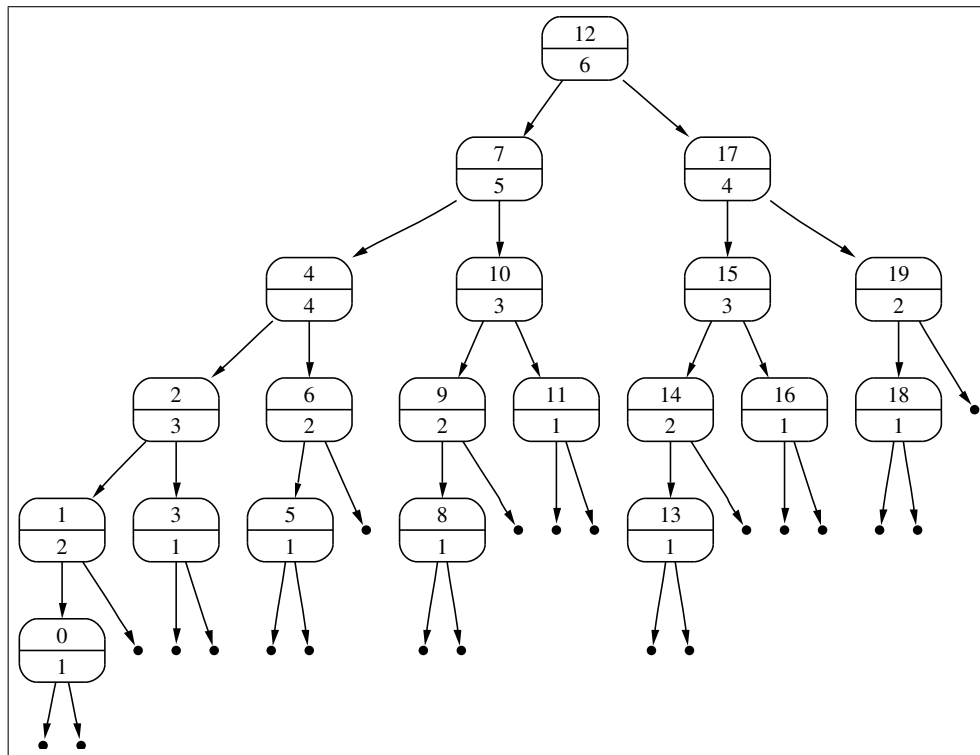


Abbildung 7.20: Ein AVL-Baum mit dem ungünstigsten Verhältnis von Höhe zur Anzahl an Knoten

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. Σ ist eine endliche Menge, deren Elemente wir als *Buchstaben* bezeichnen. Σ selbst heißt das *Alphabet*.
2. Σ^* 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.

In *Java* können wir später c und r wie folgt aus dem String w gewinnen:

$$c = w.\text{charAt}(0); \quad \text{und} \quad r = w.\text{substring}(1);$$

3. ε bezeichnet das leere Wort. In *Java* können wir schreiben

$$\text{epsilon} = "";$$

4. *Value* ist eine Menge von *Werten*.

Die Menge \mathbb{T} der Tries definieren wir nun induktiv mit Hilfe des Konstruktors

$$\text{node} : \text{Value} \times \text{List}(\Sigma) \times \text{List}(\mathbb{T}) \rightarrow \mathbb{T}.$$

Die induktive Definition besteht nur aus einer einzigen Klausel. Falls

1. $v \in \text{Value} \cup \{\Omega\}$
2. $C = [c_1, \dots, c_n] \in \text{List}(\Sigma)$ eine Liste von Buchstaben der Länge n ist,
3. $T = [t_1, \dots, t_n] \in \text{List}(\mathbb{T})$ eine Liste von Tries derselben Länge n ist,

dann gilt

$$\text{node}(v, C, T) \in \mathbb{T}.$$

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

$$\text{node}(v, [c_1, \dots, c_n], [t_1, \dots, t_n]) \in \mathbb{T}$$

dargestellt wird. Wir beantworten diese Frage, indem wir rekursive Gleichungen für die Methode

$$\text{find} : \mathbb{T} \times \Sigma^* \rightarrow \text{Value} \cup \{\Omega\}$$

angeben. Wir werden den Ausdruck $\text{node}(v, L, T).\text{find}(s)$ durch Induktion über den String s definieren:

1. $\text{node}(v, C, T).\text{find}(\varepsilon) = v.$

Der dem leeren String zugeordnete Wert wird also unmittelbar an der Wurzel des Tries abgespeichert.

$$2. \text{ node}(v, [c_1, \dots, c_n], [t_1, \dots, t_n]).\text{find}(cr) = \begin{cases} t_1.\text{find}(r) & \text{falls } c = c_1; \\ \vdots & \\ t_i.\text{find}(r) & \text{falls } c = c_i; \\ \vdots & \\ t_n.\text{find}(r) & \text{falls } c = c_n; \\ \Omega & \text{falls } c \notin \{c_1, \dots, c_n\}. \end{cases}$$

Der Trie $node(v, [c_1, \dots, c_n], [t_1, \dots, t_n])$ enthält also genau dann einen Wert zu dem Schlüssel cr , wenn einerseits der Buchstabe c in der Buchstaben-Liste an der Stelle i auftritt und wenn andererseits der Trie t_i einen Wert zu dem Schlüssel r enthält.

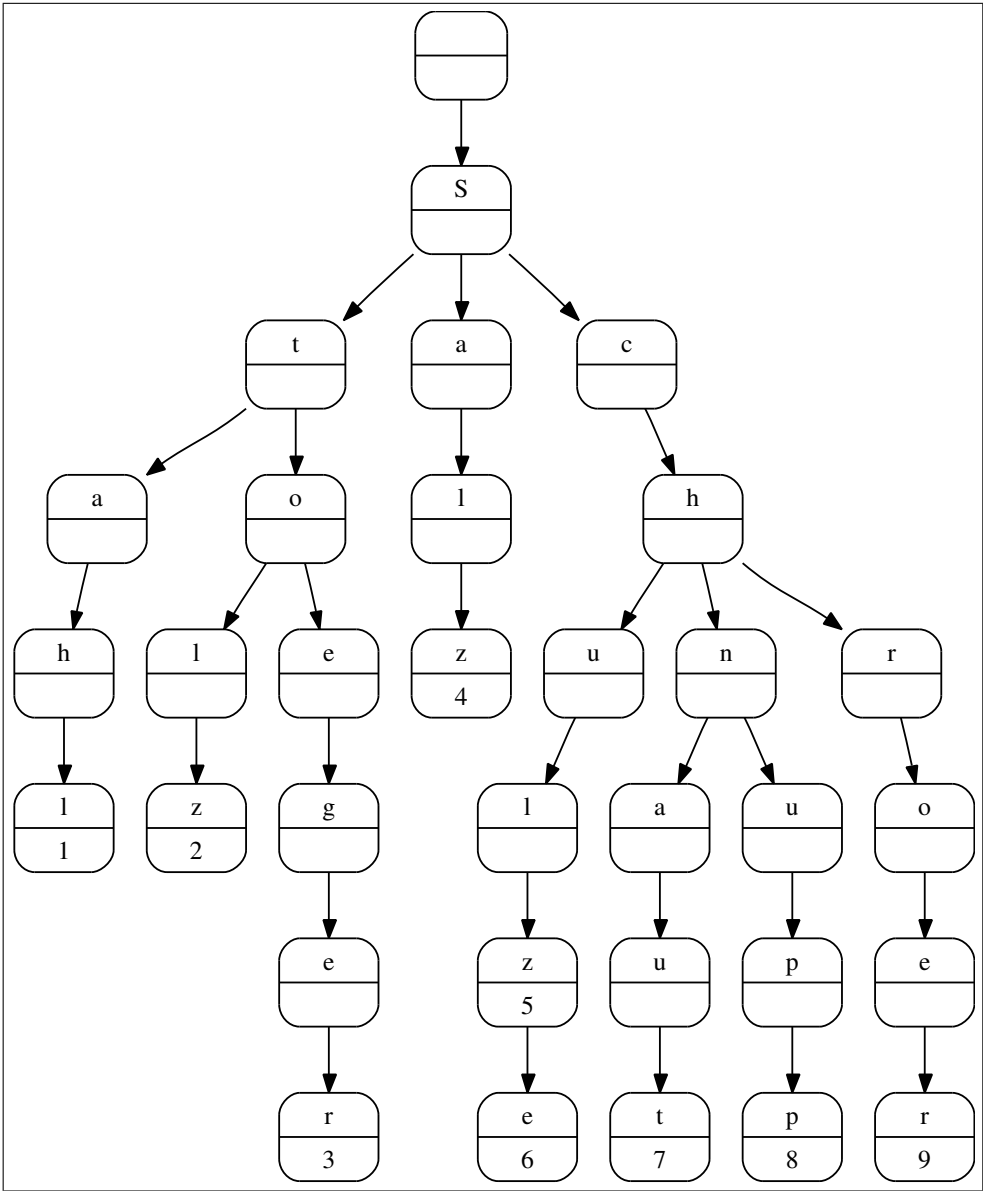


Abbildung 7.21: Ein Beispiel Trie

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

$$\text{node}(v, [c_1, \dots, c_n], [t_1, \dots, t_n])$$

darzustellen, zeichnen wir einen Kreis, den wir durch einen horizontalen Strich in der Mitte aufteilen. Falls v von Ω verschieden ist, schreiben wir den Wert v in die untere Hälfte des Kreises. (Bei den in Abbildung 7.21 gezeigten Kreisen handelt es sich um Mutantenkreise.) 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, \dots, t_n darstellen. Außerdem markieren wir die diese Knoten darstellenden Kreise in den oberen Hälfte mit den Buchstaben c_1, \dots, c_n .

Zur Verdeutlichung geben wir ein Beispiel in Abbildung 7.21 auf Seite 137. Die Funktion, die hier dargestellt wird, lässt sich wie folgt als binäre Relation schreiben:

$$\begin{aligned} &\{ \langle \text{“Stahl”}, 1 \rangle, \langle \text{“Stolz”}, 2 \rangle, \langle \text{“Stoeger”}, 3 \rangle, \langle \text{“Salz”}, 4 \rangle, \langle \text{“Schulz”}, 5 \rangle, \\ &\quad \langle \text{“Schulze”}, 6 \rangle, \langle \text{“Schnaut”}, 7 \rangle, \langle \text{“Schnupp”}, 8 \rangle, \langle \text{“Schroer”}, 9 \rangle \}. \end{aligned}$$

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

$$\text{node}(\Omega, [\text{‘S’}], [t]).$$

Dabei bezeichnet t den Trie, dessen Wurzel mit dem Buchstaben ‘S’ markiert ist. Diesen Trie können wir seinerseits durch den Term

$$\text{node}(\Omega, [\text{‘t’}, \text{‘a’}, \text{‘c’}], [t_1, t_2, t_3])$$

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 $\text{insert}()$, so hat diese Methode die Signatur

$$\text{insert} : \mathbb{T} \times \Sigma^* \times V \rightarrow \mathbb{T}.$$

Wir definieren den Wert von

$$\text{node}(v, [c_1, \dots, c_n], [t_1, \dots, t_n]).\text{insert}(s, v)$$

für ein Wort $w \in \Sigma^*$ und einen Wert $v \in V$ durch Induktion nach der Länge des Wortes w .

1. $\text{node}(v_1, L, T).\text{insert}(\varepsilon, v_2) = \text{node}(v_2, L, T)$,
Einfügen eines Wertes mit dem leeren String als Schlüssel überschreibt also einfach den an dem Wurzel-Knoten gespeicherten Wert.

2. $\text{node}(v_1, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i, \dots, t_n]).\text{insert}(c_i r, v_2) =$
 $\text{node}(v_1, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i.\text{insert}(r, v_2), \dots, t_n]).$

Wenn in dem Trie $\text{node}(v_1, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i, \dots, t_n])$ ein Wert v_2 zu dem Schlüssel $c_i r$ eingefügt werden soll, und falls der Buchstabe c in der Liste $[c_1, \dots, c_n]$ 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. $c \notin \{c_1, \dots, c_n\} \rightarrow \text{node}(v_1, [c_1, \dots, c_n], [t_1, \dots, t_n]).\text{insert}(c r, v_2) =$
 $\text{node}(v_1, [c_1, \dots, c_n, c], [t_1, \dots, t_n, \text{node}(\Omega, [], []).\text{insert}(r, v_2)]).$

Wenn in dem Trie $\text{node}(v_1, [c_1, \dots, c_n], [t_1, \dots, t_n])$ ein Wert v_2 zu dem Schlüssel $c r$ eingefügt werden soll, und falls der Buchstabe c in der Liste $[c_1, \dots, c_n]$ nicht vorkommt, dann wird

zunächst ein Trie erzeugt, der die leere Abbildung repräsentiert. Dieser Trie hat die Form

$$\text{node}(\Omega, [], []).$$

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 $[c_1, \dots, c_n]$ an und fügen den Trie

$$\text{node}(\Omega, [], []).\text{insert}(r, v_2)$$

am Ende der Liste $[t_1, \dots, t_n]$ 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

$$\text{isEmpty} : \mathbb{T} \rightarrow \mathbb{B},$$

so dass $t.\text{isEmpty}()$ genau dann **true** liefert, wenn der Trie t die leere Funktion darstellt. Wir definieren also:

1. $\text{node}(\Omega, [], []).\text{isEmpty}() = \text{true}$
2. $v \neq \Omega \rightarrow \text{node}(v, [c_1, \dots, c_n], [t_1, \dots, t_n]).\text{isEmpty}() = \text{false}$
3. $\text{node}(\Omega, L, T).\text{isEmpty}() = \text{isEmptyList}(T)$

In der letzten Gleichung haben wir eine weitere Hilfs-Funktion benutzt, die wir noch definieren müssen. Die Funktion

$$\text{isEmptyList} : \text{List}(\mathbb{T}) \rightarrow \mathbb{B}$$

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. $\text{isEmptyList}([]) = \text{true},$
2. $\text{isEmptyList}([t] + R) = (t.\text{isEmpty}() \wedge \text{isEmptyList}(R)),$
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

$$\text{delete} : \mathbb{T} \times \Sigma^* \rightarrow \mathbb{T}$$

spezifizieren: Wir definieren den Wert von

$$t.\text{delete}(w)$$

für einen Trie $t \in \mathbb{B}$ und ein Wort $w \in \Sigma^*$ durch Induktion nach der Länge des Wortes w .

1. $\text{node}(v, L, T).\text{delete}(\varepsilon) = \text{node}(\Omega, L, T),$
denn der Wert, der unter dem leeren String ε in einem Trie gespeichert wird, befindet sich unmittelbar an der Wurzel des Tries und kann dort sofort gelöscht werden.
2.
$$\begin{aligned} t_i.\text{delete}(r).\text{isEmpty}() &\rightarrow \\ \text{node}(v, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i, \dots, t_n]).\text{delete}(c_i r) &= \\ \text{node}(v, [c_1, \dots, c_{i-1}, c_{i+1}, \dots, c_n], [t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n]). \end{aligned}$$

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 .

$$\begin{aligned}
3. \quad & \neg t_i.delete(r).isEmpty() && \wedge \\
& node(v, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i, \dots, t_n]).delete(c_i r) && = \\
& node(v, [c_1, \dots, c_i, \dots, c_n], [t_1, \dots, t_i.delete(r), \dots, t_n]).
\end{aligned}$$

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 .

$$4. \quad c \notin C \rightarrow node(v, C, T).delete(cr) = node(v, C, T).$$

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 Java

Wir zeigen nun, wie sich die Tries in *Java* implementieren lassen. Die Abbildungen 7.22 und 7.23 auf den Seiten 141 und 142 zeigen die Implementierung, die wir jetzt diskutieren.

1. Den Trie $node(v, [c_1, \dots, c_n], [t_1, \dots, t_n])$ stellen wir durch ein Objekt der Klasse *TrieNode* dar. Diese Klasse hat drei Member-Variablen:
 - (a) **mValue** entspricht dem Wert v , der an der Wurzel des Tries gespeichert ist.
 - (b) **mCharList** entspricht der Buchstaben-Liste $[c_1, \dots, c_n]$.
 - (c) **mNodeList** entspricht der Trie-Liste $[t_1, \dots, t_n]$.
2. Der Konstruktor in Zeile 9 erzeugt den Trie $node(\Omega, [], [])$, der die leere Funktion repräsentiert.
3. Die Implementierung der Methode *find* orientiert sich genau an den Gleichungen, mit denen wir diese Methode spezifiziert haben.
 - (a) Falls der Schlüssel, nach dem wir suchen, der leere String ist, geben wir den Wert **mValue** zurück.
 - (b) Sonst hat der Schlüssel die Form $key = cr$. Wir setzen $firstChar = c$ und $rest = r$. Wir gehen nun die Buchstaben-Liste **mCharList** durch und schauen, ob wir dabei den Buchstaben c finden. Wenn wir den Buchstaben c an der i -ten Stelle finden, dann suchen wir anschließend in dem i -ten Trie in der Liste **mNodeList** nach dem Schlüssel r . Falls der Buchstabe c nicht gefunden wird, geben wir **null** zurück um den speziellen Wert Ω zu repräsentieren.
4. Die Implementierung der Methode *insert* ist analog zu der Implementierung der Methode *find*.
 - (a) Falls der Schlüssel, unter dem wir den Wert *value* einfügen wollen, der leere String ist, können wir die Member-Variable **mValue** überschreiben.
 - (b) Sonst hat der Schlüssel die Form $key = cr$. Wir setzen wieder $firstChar = c$ und $rest = r$. Wir gehen nun die Buchstaben-Liste **mCharList** durch und suchen den Buchstaben c . Wenn wir c an der i -ten Stelle finden, dann fügen wir anschließend in dem i -ten Trie in der Liste **mNodeList** den Wert **value** unter dem Schlüssel r ein. Falls der Buchstabe c nicht in der Liste **mCharList** auftritt, dann fügen wir c am Ende der Buchstaben-Liste ein. Gleichzeitig erzeugen wir einen zunächst leeren Trie, in dem wir dann den Wert **value** unter dem Schlüssel r einfügen. Diesen Trie fügen wir an das Ende der Liste **mNodeList** ein.
5. Als letztes diskutieren wir die Implementierung der Methode *delete*.

```

1  import java.util.*;
2
3  public class TrieNode<Value> implements MyMap<String, Value>
4  {
5      Value                mValue;
6      ArrayList<Character> mCharList;
7      ArrayList<TrieNode<Value>> mNodeList;
8
9      TrieNode() {
10         mValue = null;
11         mCharList = new ArrayList<Character>(0);
12         mNodeList = new ArrayList<TrieNode<Value>>(0);
13     }
14     public Value find(String key) {
15         if (key.length() == 0) {
16             return mValue;
17         } else {
18             Character firstChar = key.charAt(0);
19             String rest = key.substring(1);
20             for (int i = 0; i < mCharList.size(); ++i) {
21                 if (firstChar.equals(mCharList.get(i))) {
22                     return mNodeList.get(i).find(rest);
23                 }
24             }
25             return null;
26         }
27     }
28     public void insert(String key, Value value) {
29         if (key.length() == 0) {
30             mValue = value;
31         } else {
32             Character firstChar = key.charAt(0);
33             String rest = key.substring(1);
34             for (int i = 0; i < mCharList.size(); ++i) {
35                 if (firstChar.equals(mCharList.get(i))) {
36                     mNodeList.get(i).insert(rest, value);
37                     return;
38                 }
39             }
40             mCharList.add(firstChar);
41             TrieNode<Value> node = new TrieNode<Value>();
42             node.insert(rest, value);
43             mNodeList.add(node);
44         }
45     }

```

Abbildung 7.22: Die Klasse *TrieNode-I*, Teil I.

- (a) Falls der Schlüssel, den wir löschen wollen, der leere String ist, so setzen wir einfach die Member-Variable `mValue` auf `null`.
- (b) Sonst hat der Schlüssel die Form `key = cr`. Wir setzen wieder `firstChar = c` und `rest = r`.

Wir gehen nun die Buchstaben-Liste `mCharList` durch und suchen den Buchstaben c finden. Wenn wir c an der i -ten Stelle finden, dann löschen wir in dem i -ten Trie in der Liste `mNodeList` den Schlüssel r . Falls dieser Trie jetzt leer ist, so löschen wir einerseits diesen Trie aus `mNodeList` und andererseits löschen wir den Buchstaben c_i aus der Liste `mCharList`.

Falls der Buchstabe c nicht in der Liste `mCharList` auftritt, so ist nichts weiter zu tun, denn in diesem Fall sind in dem Trie keinerlei Informationen zu dem Schlüssel `key` gespeichert.

```

46     public void delete(String key) {
47         if (key.length() == 0) {
48             mValue = null;
49             return;
50         }
51         Character firstChar = key.charAt(0);
52         String rest = key.substring(1);
53         for (int i = 0; i < mCharList.size(); ++i) {
54             if (firstChar.equals(mCharList.get(i))) {
55                 TrieNode<Value> node = mNodeList.get(i);
56                 node.delete(rest);
57                 if (node.isEmpty()) {
58                     mCharList.remove(i);
59                     mNodeList.remove(i);
60                 }
61                 return;
62             }
63         }
64     }
65     public Boolean isEmpty() {
66         return mValue == null && mNodeList.size() == 0;
67     }
68 }

```

Abbildung 7.23: Die Klasse *TrieNode*, Teil II.

Bemerkung: Falls das Alphabet Σ viele Buchstaben enthält, können die Listen $[c_1, \dots, c_n]$ und $[t_1, \dots, t_n]$, die in einem Trie der Form

$$node(v, [c_1, \dots, c_n], [t_1, \dots, t_n])$$

abgespeichert sind, lang werden. Dann ist es eventuell effizienter, die Buchstaben-Liste $[c_1, \dots, c_n]$ zu sortieren. Dann könnte die Methode `find()` effizienter implementiert werden. Allerdings wird das Einfügen und Löschen in diesem Fall komplizierter.

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 Σ^* auffassen. Wir wollen die Menge der *binären Tries* mit BT bezeichnen und wie folgt induktiv definieren:

1. $nil \in BT$.
2. $bin(v, l, r) \in BT$ falls
 - (a) $v \in Value \cup \{\Omega\}$.

(b) $l, r \in \text{BT}$.

Die Semantik legen wir fest, indem wir eine Methode

$$\text{find} : \text{BT} \times \mathbb{N} \rightarrow \text{Value} \cup \{\Omega\}$$

definieren. Für einen binären Trie b und eine natürliche Zahl n gibt $b.\text{find}(n)$ 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 Ω zurück gegeben. Formal definieren wir den Wert von $b.\text{find}(n)$ durch Induktion nach dem Aufbau von b . Im Induktions-Schritt ist eine Neben-Induktion nach n erforderlich.

1. $\text{nil}.\text{find}(n) = \Omega$,
denn in dem leeren binären Trie finden wir keine Werte.
2. $\text{bin}(v, l, r).\text{find}(0) = v$,
denn der Schlüssel 0 entspricht dem leeren String ε .
3. $n \neq 0 \rightarrow \text{bin}(v, l, r).\text{find}(2 \cdot n) = l.\text{find}(n)$,
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. $\text{bin}(v, l, r).\text{find}(2 \cdot n + 1) = r.\text{find}(n)$,
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 $\text{bin}(\Omega, \text{nil}, \text{nil})$ zu nil vereinfacht werden.

Hinweis: Um die Gleichungen zur Spezifikation der Funktion $\text{delete}()$ nicht zu komplex werden zu lassen ist es sinnvoll, eine Hilfsfunktion zur Vereinfachung von binären Tries zu definieren.

2. Implementieren Sie binäre Tries in *Java*.

Bemerkung: Binäre Tries werden auch als *digitale Suchbäume* bezeichnet.

7.5 Hash-Tabellen

Eine Abbildung

$$f : \text{Key} \rightarrow \text{Value}$$

kann dann sehr einfach implementiert werden, wenn

$$\text{Key} = \{0, 1, 2, \dots, n\},$$

denn dann reicht es aus, ein Feld der Größe $n + 1$ zu verwenden. Abbildung 7.24 zeigt, dass sich der ADT *Map* in diesem Fall trivial implementieren lässt.

```
1  public class ArrayMap<Value> implements MyMap<Integer, Value>
2  {
3      Value[] mArray;
4
5      public ArrayMap(int n) {
6          mArray = (Value[]) new Object[n+1];
7      }
8      public Value find(Integer key) {
9          return mArray[key];
10     }
11     public void insert(Integer key, Value value) {
12         mArray[key] = value;
13     }
14     public void delete(Integer key) {
15         mArray[key] = null;
16     }
17 }
```

Abbildung 7.24: Die Klasse *ArrayMap*.

Falls nun der Definitions-Bereich D der darzustellenden Abbildung nicht die Form einer Menge der Gestalt $\{1, \dots, n\}$ hat, könnten wir versuchen, D zunächst auf eine Menge der Form $\{1, \dots, n\}$ 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 $\Sigma = \{ ' ', 'a', 'b', 'c', \dots, 'x', 'y', 'z' \}$ einen Wert aus der Menge $\{0, \dots, 26\}$ zuordnet

$$\text{ord} : \{ ' ', 'a', 'b', 'c', \dots, 'x', 'y', 'z' \} \rightarrow \{0, \dots, 26\},$$

so lässt sich der Wert eines Strings $w = c_0c_1 \dots c_7$ durch eine Funktion

$$\text{code} : \Sigma^* \rightarrow \mathbb{N}$$

berechnen, die wie folgt definiert ist:

$$\text{code}(c_0c_1 \dots c_7) = \sum_{i=0}^7 \text{ord}(c_i) \cdot 27^i.$$

Die Menge *code* bildet die Menge aller Wörter mit 8 Buchstaben bijektiv auf die Menge der

Zahlen $\{0, \dots, (27^8 - 1)/26\}$ 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.

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:

$$\text{code}(c_0 c_1 \dots c_n) = \left(\sum_{i=0}^n \text{ord}(c_i) \cdot 27^i \right) \% \text{size}.$$

Um eine Überlauf zu vermeiden, können wir für $k = n, n-1, \dots, 1, 0$ die Teilsummen s_k wie folgt induktiv definieren:

- (a) $s_n = \text{ord}(c_n)$
- (b) $s_k = (\text{ord}(c_k) + s_{k+1} \cdot 27) \% \text{size}$

Dann gilt
$$s_0 = \left(\sum_{i=0}^n \text{ord}(c_i) \cdot 27^i \right) \% \text{size}.$$

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 7.25 auf Seite 146 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 *Java*.

1. Als erstes überlegen wir uns, welche Daten-Strukturen wir brauchen, um eine Hash-Tabelle zu repräsentieren.

- (a) Wir benötigen ein Feld, indem wir die einzelnen Listen ablegen. Dieses Feld wird in Zeile 10 als die Member-Variable `mArray` abgespeichert.

Es mag Sie verwundern, dass dieses Feld den Typ `Object[]` hat. Eigentlich sollte dieses Feld in der Form

```
List<Pair<Key, Value>>[] mArray;
```

deklariert werden. Die Erzeugung generischer Felder ist in *Java* aber sehr trickreich. Um nicht zu sehr vom eigentlichen Thema abzukommen haben wir daher eine Implementierung gewählt, die nicht Typ-sicher ist.

- (b) 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

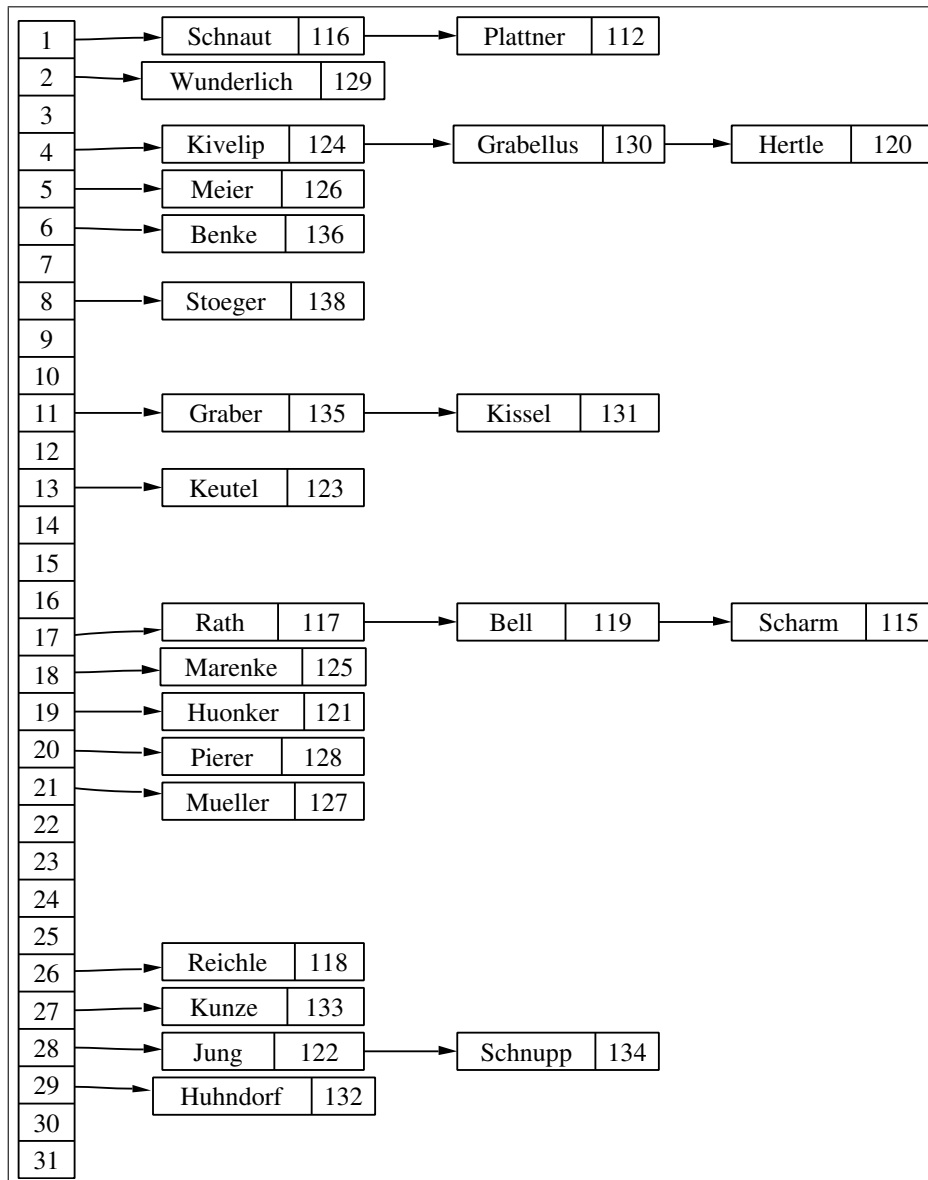


Abbildung 7.25: Eine Hash-Tabelle

das Feld vergrößert werden. Um diesen Prozess zu steuern, müssen wir zunächst nachhalten, wieviele Elemente schon in der Hash-Tabelle abgespeichert sind. Dies geschieht in der Member-Variable `mNumberEntries`, die in Zeile 12 definiert wird.

Theoretische Untersuchungen, die über den Rahmen der Vorlesung hinausgehen, zeigen, dass die Größe der Tabelle eine Primzahl sein sollte. Daher verfügt der Konstruktor über eine Liste von Primzahlen, die in der statistischen Member-Variablen `sPrimes`, die in Zeile 4 definiert ist, abgelegt sind. Die $i + 1$ -te Primzahlen in dieser Liste ist in etwa doppelt so groß wie die i -te Primzahl. Die Member-Variable `mPrimeIndex`, die in Zeile 11 definiert wird, kodiert nun die Größe des Feldes `mArray`. Es gilt immer

`mArray.length == sPrimes[mPrimeIndex].`

Die durchschnittliche Länge der einzelnen Listen ergibt sich als der Quotient aus der Zahl `mNumberEntries` und der Länge des Feldes `mArray`. Wird nun dieser Wert größer

```

1  public class MyHashMap<Key, Value> implements MyMap<Key, Value>
2  {
3      static final double sAlpha = 2;
4      static final int[] sPrimes = { 3, 7, 13, 31, 61, 127, 251,
5          509, 1021, 2039, 4093, 8191, 16381, 32749, 65521, 131071,
6          262139, 524287, 1048573, 2097143, 4194301, 8388593, 16777213,
7          33554393, 67108859, 134217689, 268435399, 536870909, 1073741789,
8          2147483647
9      };
10     Object[] mArray;
11     int      mPrimeIndex;
12     int      mNumberEntries;
13
14     public MyHashMap(int primeIndex) {
15         mPrimeIndex = primeIndex;
16         int size     = sPrimes[mPrimeIndex];
17         mArray       = new Object[size];
18     }
19     public Value find(Key key) {
20         int index = Math.abs(key.hashCode() % mArray.length);
21         LinkedList<Pair<Key, Value>> list =
22             (LinkedList<Pair<Key, Value>>) mArray[index];
23         if (list == null) {
24             return null;
25         }
26         for (int i = 0; i < list.size(); ++i) {
27             Pair<Key, Value> pair = list.get(i);
28             if (key.equals(pair.getFirst())) {
29                 return pair.getSecond();
30             }
31         }
32         return null;
33     }

```

Abbildung 7.26: Die Klasse *MyHashMap*, Teil I.

als der *Auslastungs-Faktor* (engl. *load factor*) **sAlpha**, der in Zeile 3 definiert ist, dann verdoppeln wir die Größe des Feldes.

2. Der Konstruktor in Zeile 14 initialisiert **mPrimeIndex** mit dem gegebenen Argument. Wird der Konstruktor zum Beispiel mit dem Argument 0 aufgerufen, dann wird ein Feld der Länge 3 angelegt, denn es gilt **sPrimes[0] = 3**.
3. Bei der Implementierung der Methode **find** wandeln wir den gegebenen Schlüssel **key** zunächst mit der Methode **hashCode()** in eine Zahl um. Die Methode **hashCode()** ist in Java für jedes Objekt definiert und erzeugt eine mehr oder weniger zufällige Zahl, die aber in eindeutiger Weise von dem Objekt abhängt. Diese Zahl kann auch negativ sein. Wir modifizieren diese Zahl in Zeile 20 durch Bilden von Modulo und Absolutbetrag so, dass das Ergebnis in der Menge $\{0, \dots, \mathbf{mArray.length} - 1\}$ liegt. Anschließend holen wir die Liste, in der Werte zu dem gegebenen Schlüssel abgespeichert sein müssen. Falls in dem Feld an der Stelle, die durch den berechneten Index angegeben wird, noch gar keine Liste gespeichert ist, hat die Hash-Tabelle zu dem gegebenen Schlüssel noch keinen Eintrag und wir geben in Zeile 24 **null** zurück.

```

34     public void insert(Key key, Value value) {
35         if (mNumberEntries / (double) mArray.length > sAlpha) {
36             rehash();
37         }
38         int index = Math.abs(key.hashCode() % mArray.length);
39         LinkedList<Pair<Key, Value>> list =
40             (LinkedList<Pair<Key, Value>>) mArray[index];
41         if (list == null) {
42             list = new LinkedList<Pair<Key, Value>>();
43             mArray[index] = list;
44         }
45         for (int i = 0; i < list.size(); ++i) {
46             Pair<Key, Value> pair = list.get(i);
47             if (key.equals(pair.getFirst())) {
48                 pair.setSecond(value);
49                 return;
50             }
51         }
52         list.add(new Pair<Key, Value>(key, value));
53         ++mNumberEntries;
54     }
55     private void rehash() {
56         ++mPrimeIndex;
57         MyHashMap<Key, Value> bigMap = new MyHashMap<Key, Value>(mPrimeIndex);
58         for (Object list: mArray) {
59             if (list == null) {
60                 continue;
61             }
62             for (Object object: (LinkedList<Pair<Key, Value>>) list) {
63                 Pair<Key, Value> pair = (Pair<Key, Value>) object;
64                 bigMap.insert(pair.getFirst(), pair.getSecond());
65             }
66         }
67         mArray = bigMap.mArray;
68     }

```

Abbildung 7.27: Die Klasse *MyHashMap*, Teil II.

Andernfalls laufen wir mit einer Schleife durch die Liste durch und vergleichen die einzelnen Schlüssel mit dem gegebenen Schlüssel `key`. Falls wir den Schlüssel finden, geben wir in Zeile 29 den mit diesem Schlüssel assoziierten Wert zurück.

Falls wir die Schleife bis zum Ende durchlaufen und den Schlüssel nicht gefunden haben, dann hat die Hash-Tabelle zu dem gegebenen Schlüssel `key` keinen Eintrag und wir geben in Zeile 32 wieder `null` zurück.

- Bei der Implementierung der Methode *insert* berechnen wir zunächst den aktuellen Auslastungs-Faktor, also die durchschnittliche Länge der Listen. Falls diese Länge größer als der vorgegebene maximale Auslastungs-Faktor `sAlpha` ist, führen wir ein sogenanntes *Rehashing* durch, das wir weiter unten im Detail diskutieren.

Anschließend ist die Implementierung analog zur Implementierung der Methode *find()*. Wir berechnen also zunächst die Liste, in der Schlüssel `key` und der Wert `value` einzufügen sind.

```

69     public void delete(Key key) {
70         int index = Math.abs(key.hashCode() % mArray.length);
71         LinkedList<Pair<Key, Value>> list =
72             (LinkedList<Pair<Key, Value>>) mArray[index];
73         if (list == null) {
74             return;
75         }
76         for (int i = 0; i < list.size(); ++i) {
77             Pair<Key, Value> pair = list.get(i);
78             if (key.equals(pair.getFirst())) {
79                 list.remove(i);
80                 --mNumberEntries;
81                 return;
82             }
83         }
84     }
85 }

```

Abbildung 7.28: Die Klasse *MyHashMap*, Teil III.

Falls unter dem Index noch keine Liste existiert, erzeugen wir in den Zeilen 42 und 43 eine neue leere Liste und tragen diese Liste in das Feld ein.

Anschließend durchlaufen wir die Liste und suchen den Schlüssel **key**. Wenn wir den Schlüssel finden, dann wird einfach der zugehörige Wert überschrieben. Wenn wir den Schlüssel **key** in der Liste nicht finden, dann fügen wir den Schlüssel zusammen mit dem zugeordneten Wert **value** in Zeile 52 an das Ende der Liste an. Gleichzeitig müssen wir in diesem Fall die Zahl der Einträge **mNumberEntries** inkrementieren.

5. Als nächstes besprechen wir das *Rehashing*, das in Zeile 55 – 69 implementiert ist. Wir inkrementieren zunächst **mPrimeIndex** und bilden dann eine neue Hash-Tabelle, die in etwa doppelt so groß ist, wie die alte Hash-Tabelle. Anschließend kopieren wir die Werte aus der alten Hash-Tabelle in die neue Tabelle. Dazu durchlaufen wir das Feld **mArray** in der **for**-Schleife, die sich von Zeile 58 – 66 erstreckt. Anschließend fügen wir die Elemente aus dem Feld **mArray** in der inneren Schleife, die sich von 62 – 65 erstreckt, in die neue Hash-Tabelle ein. Wir können die einzelnen Werte nicht einfach kopieren, denn der Index, der angibt, in welcher Liste ein Schlüssel eingetragen ist, hängt ja nicht nur von dem Hash-Code des Schlüssels sondern auch von der Größe des Feldes ab. Zum Schluss kopieren wir in Zeile 67 das Feld der neu angelegten Hash-Tabelle in die ursprüngliche Hash-Tabelle.
6. Als letztes diskutieren wir das Löschen in einer Hash-Tabelle. Genau wie beim Suchen und Einfügen berechnen wir zunächst den Index der Liste, in der sich der Schlüssel befinden muss, falls die Hash-Tabelle überhaupt einen Eintrag zu dem Schlüssel enthält. Anschließend vergleichen wir in der **for**-Schleife in Zeile 75 – 82 alle Schlüssel dieser Liste mit dem Schlüssel **key**. Falls wir den Schlüssel in der Liste finden, löschen wir das Paar, dass diesen Schlüssel enthält, aus der Liste. Zusätzlich erniedrigen wir in diesem Fall die Zahl der Einträge **mNumberEntries**.

Im ungünstigsten Fall kann die Komplexität der Methoden *find*, *insert* und *delete* linear mit der Anzahl der Einträge in der Hash-Tabelle wachsen. Dieser Fall tritt dann auf, wenn die Funktion **hash(*k*)** 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

$$\alpha = \frac{\text{count}}{\text{size}}.$$

Hierbei ist `count` die Gesamtzahl der Einträge in der Tabelle und `size` gibt die Größe der Tabelle an. Das Verhältnis α dieser beiden Zahlen bezeichnen wir als den *Auslastungs-Faktor* der Hash-Tabelle. In der Praxis zeigt sich, dass α 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 Werkzeugen 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. **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) $c.add(e) \rightarrow c' = c \cup \{e\}$.

Hier bezeichnet c' den Wert, den die Zusammenfassung c hat, nachdem der Aufruf $c.add(e)$ erfolgt ist.

(b) $c.add(e) = (e \notin c)$,

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.

2. **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) $c.addAll(d) \rightarrow c' = c \cup d$,

(b) $c.addAll(d) = (c' \neq c)$.

3. **void clear()**

Der Aufruf $c.clear()$ löscht alle Elemente aus der Zusammenfassung c . Diese ist danach leer. Falls c eine Menge ist, gilt also

$$c.clear() \rightarrow c' = \{\}.$$

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

$$c.contains(e) = (e \in c).$$

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

$$c.containsAll(d) = (d \subseteq c).$$

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

$$c.isEmpty() = (c = \{\}).$$

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) $c.remove(e) \rightarrow c' = c \setminus \{e\},$

(b) $c.remove(e) = (e \in c).$

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) $c.removeAll(d) \rightarrow c' = c \setminus d,$

(b) $c.removeAll(d) = (c' \neq c).$

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) $c.retainAll(d) \rightarrow c' = c \cap d,$

(b) $c.retainAll(d) = (c' \neq c).$

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 7.29 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 7.29: Casten eines Feldes

Um die Liste `l` in ein Feld vom Typ `Integer[]` zu transformieren, müssen wir daher anders vorgehen. Abbildung 7.30 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 7.30: 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-Bäumen ist die Idee, dass die Knoten eines Baumes entweder rot oder schwarz markiert sind. Zusätzlich gelten 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) Bei der Berechnung der Höhe eines Knotens werden die roten Knoten nicht gezählt.
- (e) 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 ungünstig 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 α)`

Dieser Konstruktor erzeugt eine leere Hash-Tabelle, für die das zu Grunde liegende Feld die Größe n hat. Der Load-Faktor ist α .

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. Dies bedingt, dass die Operationen `l.get(i)` und `l.set(i, e)` eine Komplexität haben, die linear mit i anwächst. Dafür erfordert eine Aufruf der Form `l.add(e)` allerdings nur einen konstanten Aufwand.

(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.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.

- 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 7.31 auf Seite 157 zeigt das resultierende Programm, das wir jetzt diskutieren.

- 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 **{1..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.

- 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:

$$\{p \cdot q \mid p \in s_1 \wedge q \in s_2\}.$$

- 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* · *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.

```

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 7.31: Berechnung der Primzahlen mit Hilfe von Mengen

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 `k1` und `k2` die Methode

`k1.compareTo(k2)`

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 o1, O o2)`

vor. Ist `c` ein Komparator, so vergleicht der Aufruf `c.compare(o1, o2)` die beiden Objekte `o1` und `o2`. Falls `o1 < o2` ist, gibt der Aufruf eine negative Zahl zurück. Ist `o1 > o2` wird entsprechend eine positive Zahl zurück gegeben. Sind `o1` und `o2` 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 dieselben 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 α)`

Dieser Konstruktor erzeugt eine leere Hash-Tabelle, für die das zu Grunde liegende Feld die Größe `n` hat. Der Load-Faktor ist `α`.

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 Benutzer den abstrakten Datentyp *Abbildung* in der einen oder anderen Form zur Verfügung: In *Perl* [WS92] wird dieser Datentyp durch ein *assoziatives Feld* (im Original: *associative array*) implementiert, in *Lua* [Ier06, IdFF96] wird der entsprechende Datentyp als *Tabelle* (im Original: *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.

Wir hatten damals das in Abbildung 7.32 auf Seite 162 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. Die Klasse `TreeSet` ist durch Rot-Schwarz-Bäumen implementiert, während die Klasse `HashSet` mit Hilfe von Hash-Tabellen implementiert ist. Letztere haben im schlechtesten Fall eine höhere Komplexität. Daher entscheiden wir uns für die Klasse `TreeSet`. Versuchen wir das Programm aus Abbildung 7.32 in *Java* umzusetzen, so stoßen wir allerdings sofort auf ein Problem: Bei der Umsetzung benötigen wir 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` aber 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 7.33 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.

7.7.1 Die Klasse `ComparableSet`

Die Abbildungen 7.34, 7.35, 7.36, 7.37 und 7.38 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

```
interface Comparable<O> {
    int compareTo(O o);
```

```

1  program main;
2      All := { "Bauer", "Wolf", "Ziege", "Kohl" };
3      P := pow All;
4      R1 := { [ S, S - B ] : S in P, B in pow S |
5          "Bauer" in B and #B <= 2 and not problem(S - B)
6          };
7      R2 := { [ S, S + B ] : S in P, B in pow (All - S) |
8          "Bauer" in B and #B <= 2 and not problem(All - S - B) };
9      R := R1 + R2;
10     start := All;
11     goal := {};
12     path := reachable(start, goal, R);
13     print(path);
14
15     procedure problem(S);
16         return ("Ziege" in S and "Kohl" in S) or
17             ("Wolf" in S and "Ziege" in S);
18     end problem;
19
20     procedure reachable(x, y, R);
21         P := { [x] };
22         loop
23             Old_P := P;
24             P := P + path_product(P, R);
25             Found := { p in P | p(#p) = y };
26             if Found /= {} then
27                 return arb Found;
28             end if;
29             if P = Old_P then
30                 return;
31             end if;
32         end loop;
33     end reachable;
34
35     procedure path_product(P, Q);
36         return { add(p,q) : p in P, q in Q
37             | p(#p) = q(1) and not cyclic(add(p,q)) };
38     end path_product;
39
40     procedure cyclic(p);
41         return #{ x : x in p } < #p;
42     end cyclic;
43
44     procedure add(p, q);
45         return p + q(2..);
46     end add;
47 end main;

```

Abbildung 7.32: Lösung des Wolf-Ziege-Kohl-Problems in SETL2.

}

```

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 7.33: Mengen von Mengen: Vorsicht Falle!

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> { ... }
```

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` mehr bilden und müssten statt dessen auf die ungenauere Klasse `ComparableSet<A>` ausweichen, wobei wir Typinformationen verlieren würden.

- 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

```

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 7.34: Die Klasse `ComparableSet<T>`, 1. Teil.

alten Wert verloren. Solche Seiteneffekte sind für die Art und Weise, in der wir Mengen benutzen wollen, sehr unerwünscht. 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 Methode `addAll()` ersetzen. Würden wir die Klasse `ComparableSet` von der Klasse `TreeSet` ableiten, so hätte der Benutzer 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.

```

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 7.35: Die Klasse `ComparableSet<T>`, 2. Teil.

3. Die Methode `deepCopy()` erzeugt eine Kopie einer gegebenen Menge. Diese Methode ist einem Konstruktor-Aufruf der Form

```

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 7.36: 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 7.37: Die Klasse `ComparableSet<T>`, 4. Teil.

`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.

```

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 7.38: Die Klasse `ComparableSet<T>`, 5. Teil.

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 folglich $s < c$.

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

$$c.product(s) = \{\langle x, y \rangle \mid x \in c \wedge y \in s\}.$$

Damit die Implementierung dieser Methode funktioniert, muss die Klasse `Pair` so definiert sein, dass die Klasse `Pair<S,T>` das Interface `Comparable` implementiert. Abbildung 7.42 auf Seite 171 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 gegebenen `TreeSet s` die Potenz-Menge berechnet. Die Implementierung der Methode `powerSet()` basiert auf den folgenden Gleichungen:

- (a) $\text{powerSet}(\{\}) = \{\{\}\}$,
- (b) $\text{powerSet}(A \cup \{x\}) = \text{powerSet}(A) \cup \{\{x\} \cup s : s \in \text{powerSet}(A)\}$.

Die Methode `headSet()`, die bei der Realisierung benutzt wird, kann wie folgt spezifiziert werden: Der Aufruf `s.headSet(l)` liefert alle Elemente aus der Menge `s`, die kleiner als `l` sind:

$$s.\text{headSet}(l) = \{x \in s \mid x < l\}.$$

Der Aufruf `s.last()` liefert das größte Element der Menge `s`. Damit können wir eine Menge `s` in der Form

$$s = \{s.\text{last}()\} \cup s.\text{headSet}(s.\text{last}())$$

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:

$$\text{addElement}(S, e) = \{m \cup \{e\} \mid m \in S\}.$$

- 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(R1, R2)` berechnet das relationale Produkt $R_1 \circ R_2$ der Relationen `R1` und `R2`. Dieses Produkt ist wie folgt definiert:

$$R_1 \circ R_2 = \{\langle x, z \rangle \mid \exists y : \langle x, y \rangle \in R_1 \wedge \langle y, z \rangle \in R_2\}.$$

- 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

$$\{x \in c \mid s.\text{select}(x)\}.$$

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 7.39 zeigt das Interface `Selector`.

```

1  public interface Selector<T> {
2      public boolean select(T element);
3  }
```

Abbildung 7.39: Das Interface `Selector`.

- 19. Die Methode `transform()` berechnet eine Bild-Menge. Der Aufruf `c.transform(t)` berechnet für einen `Transformer t` die Menge

$$\{t.\text{transform}(x) \mid x \in c\}.$$

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 7.40 zeigt das Interface `Transformer`.

```

1  public interface Transformer<S extends Comparable<? super S>,
2                                T extends Comparable<? super T>>
3  {
4      public S transform(T x);
5  }

```

Abbildung 7.40: Das Interface `Transformer`.

20. Die Methode $\text{combineSets}(s_1, s_2, k)$ verknüpft zwei Mengen mit einem *Kombinator* k . Es gilt

$$\text{combineSets}(s_1, s_2, k) = \{k.\text{combine}(x, y) \mid x \in s_1 \wedge y \in s_2\}$$

Ein *Kombinator* ist ein Objekt, dass eine Methode $\text{combine}()$ zur Verfügung stellt, mit dem zwei Elemente verknüpft werden können. Abbildung 7.41 zeigt das Interface `Combinator`.

```

1  public interface Combinator<T, X, Y>
2  {
3      public T combine(X x, Y y);
4  }

```

Abbildung 7.41: Das Interface `Combinator`.

7.7.2 Die Klasse `ComparableList`

Nachdem wir nun mit Mengen so arbeiten können, wie wir dass in der Sprache *Set1* 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 7.43 auf Seite 172 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 $\text{compareTo}()$ implementiert werden. Dies geschieht ähnlich wie bei Mengen über einen lexikografischen Vergleich der beiden Listen.

```
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 7.42: 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 7.43: Die Klasse ComparableList.

7.7.3 Lösung des Wolf-Ziege-Kohl-Problems in Java

```
1  #define Point      ComparableSet<String>
2  #define PointPair  Pair<Point, Point>
3
4  public class WolfZiegeKohl
5  {
6      public static void main(String args[]) {
7          Point all = new Point();
8          all.add("Bauer");
9          all.add("Wolf");
10         all.add("Ziege");
11         all.add("Kohl");
12         ComparableSet<Point> p = all.powerSet();
13         ComparableSet<PointPair> r = new ComparableSet<PointPair>();
14         for (Point s: p) {
15             for (Point b : s.powerSet()) {
16                 Point sb = s.difference(b);
17                 if (b.member("Bauer") && b.size() <= 2 && !problem(sb))
18                 {
19                     PointPair ssb = new PointPair(s, sb);
20                     r.add(ssb);
21                 }
22             }
23         }
24         for (Point s: p) {
25             Point as = all.difference(s);
26             for (Point b : as.powerSet()) {
27                 if (b.member("Bauer") && b.size() <= 2 &&
28                     !problem(as.difference(b))
29                 )
30                 {
31                     Point sb = s.union(b);
32                     PointPair ssb = new PointPair(s, sb);
33                     r.add(ssb);
34                 }
35             }
36             Point goal = new Point();
37             Relation<Point> relation = new Relation(r);
38             ComparableList<Point> path = relation.findPath(all, goal);
39             for (Point left : path) {
40                 Point right = all.difference(left);
41                 System.out.println(left + ", " + right);
42             }
43         }
44         static boolean problem(Point s) {
45             return (s.member("Ziege") && s.member("Kohl")) ||
46                 (s.member("Wolf") && s.member("Ziege"));
47         }
48     }
```

Abbildung 7.44: Lösung des Wolf-Ziege-Kohl-Problems.

Nach dem wir uns im letzten Abschnitt einen Rahmen geschaffen haben, in dem konzeptionell dieselben Funktionen wie in *Set1* zur Verfügung stehen, können wir nun daran gehen, das im ersten Semester entwickelte *Set1*-Programm in *Java* zu übersetzen. Abbildung 7.44 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 Klassenbezeichnungen 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 7.44 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 zusammengefasst werden:

$$\{(s, s - b) : s \in p, b \in 2^s \mid \text{"bauer"} \in b \wedge \#b \leq 2 \wedge \neg \text{problem}(s - b)\}$$

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 \ 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 $\langle s, sb \rangle$ der Relation r hinzugefügt.

- Die Übergänge, bei denen das Boot von rechts nach links fährt, werden analog berechnet. Mathematisch hat diese Relation die Form

$$\{ \langle s, s + b \rangle : s \in p, b \in 2^{\text{all} - s} \mid \text{"Bauer"} \in b \wedge \#B \leq 2 \wedge \neg \text{problem}((\text{all} \setminus s) \setminus b) \}.$$

Da s wieder die Menge der Objekte am linken Ufer ist, finden wir die Menge der Objekte am rechten Ufer, indem wir die Menge $\text{all} \setminus s$ 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.

- 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.

- 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.
- 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.

- Die Methode `r.findPath(x, y)` berechnet einen Pfad, der von dem Punkt x zu dem Punkt y führt.
- 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:

$$P \bullet R = \{ l + [y] \mid l \in P \wedge \langle x, y \rangle \in R \wedge \text{last}(l) = x \}.$$

Für einen Pfad p bezeichnet dabei $\text{last}(p)$ 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.

- 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 7.45: 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 7.46: 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 derselben 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 sowohl 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

$\{\text{Key}, \text{Value}\}$.

Dabei muss auf der Menge Key eine totale Quasi-Ordnung $<$ existieren, so dass wir die Prioritäten verschiedener Elemente anhand der Schlüssel vergleichen können.

3. Die Menge der Funktions-Zeichen ist

$\{\text{PrioQueue}, \text{insert}, \text{remove}, \text{top}, \text{change}\}$.

4. Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:

(a) *PrioQueue* : *PrioQueue*

Der Aufruf "*PrioQueue*()" erzeugt eine leere Prioritäts-Warteschlange.

- (b) $\text{top} : \text{PrioQueue} \rightarrow (\text{Key} \times \text{Value}) \cup \{\Omega\}$
 Der Aufruf $Q.\text{top}()$ liefert ein Paar $\langle k, v \rangle$. Dabei ist v ein Element aus Q , das eine maximale Priorität hat. k ist die Priorität des Elements v .
- (c) $\text{insert} : \text{PrioQueue} \times \text{Key} \times \text{Value} \rightarrow \text{PrioQueue}$
 Der Aufruf $Q.\text{insert}(k, v)$ fügt das Element v mit der Priorität k in die Prioritäts-Warteschlange Q ein.
- (d) $\text{remove} : \text{PrioQueue} \rightarrow \text{PrioQueue}$
 Der Aufruf $Q.\text{remove}()$ entfernt aus der Prioritäts-Warteschlange Q ein Element, das eine maximale Priorität hat.
- (e) $\text{change} : \text{PrioQueue} \times \text{Key} \times \text{Value} \rightarrow \text{PrioQueue}$
 Der Aufruf $Q.\text{change}(k, v)$ ändert die Priorität des Elements v in der Prioritäts-Warteschlange Q so ab, dass k die neue Priorität dieses Elements ist. Wir setzen dabei voraus, dass einerseits das 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 Key sind und dass auf der Menge Key eine totale Quasi-Ordnung \leq existiert. Falls dann $k_1 < k_2$ ist, sagen wir, dass k_1 eine höhere Priorität als k_2 hat. Dass die Prioritäten höher werden wenn die Schlüssel kleiner werden 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 Werten dar. Für solche Mengen definieren wir unserer Methoden wie folgt.
- (a) $\text{PrioQueue}() = \{\}$,
 der Konstruktor erzeugt also eine leere Prioritäts-Warteschlange, die als leere Menge dargestellt wird.
- (b) $Q.\text{insert}(k, v) = Q \cup \{\langle k, v \rangle\}$,
 Um einen Wert v mit einer Priorität k in die Prioritäts-Warteschlange Q einzufügen, reicht es aus, das Paar $\langle k, v \rangle$ zu der Menge Q hinzuzufügen.
- (c) Wenn Q leer ist, dann ist $Q.\text{top}()$ undefiniert:
 $Q = \{\} \rightarrow Q.\text{top}() = \Omega$.
- (d) Wenn Q nicht leer ist, wenn es also ein Paar $\langle k_1, v_1 \rangle$ in Q gibt, dann liefert $Q.\text{top}()$ ein Paar $\langle k_2, v \rangle$ aus der Menge Q , so dass der Schlüssel k_2 minimal wird. Dann gilt also für alle $\langle k_1, v_1 \rangle \in Q$, dass $k_2 \leq k_1$ ist. Formal können wir schreiben:
 $\langle k_1, v_1 \rangle \in Q \wedge Q.\text{top}() = \langle k_2, v_2 \rangle \rightarrow k_2 \leq k_1 \wedge \langle k_2, v_2 \rangle \in Q$.
- (e) Falls Q leer ist, dann ändert $\text{remove}()$ nichts daran:
 $Q = \{\} \rightarrow Q.\text{remove}() = Q$.
- (f) Sonst entfernt $Q.\text{remove}()$ ein Paar mit der höchsten Priorität:
 $Q \neq \{\} \rightarrow Q.\text{remove}() = Q \setminus \{Q.\text{top}()\}$.
- (g) Die Methode $\text{change}()$ ändert die Priorität des Paares, dessen Wert als Argument übergeben wird:
 $Q.\text{change}(k_1, v_1) = \{\langle k_2, v_2 \rangle \in Q \mid v_2 \neq v_1\} \cup \{\langle k_1, v_1 \rangle\}$.

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(k, v)* geht alle Einträge der Liste durch. Falls dabei ein Eintrag mit dem Wert v gefunden wird, so wird das zugehörige Paar aus der Liste gelöscht. Anschließend wird das Paar $\langle k, v \rangle$ 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 $\mathcal{O}(\log(n))$ 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 einen Schlüssel $k_1 \in \text{Key}$ und einen binären Baum $b \in \mathcal{B}$ die Relation $k_1 \leq b$ durch Induktion über b . Die Intention ist dabei, dass $k_1 \leq b$ genau dann gilt, wenn für jeden Schlüssel k_2 , der in b auftritt, $k_1 \leq k_2$ gilt. Die formale Definition ist wie folgt:

1. $k_1 \leq \text{nil}$,
denn in dem leeren Baum treten überhaupt keine Schlüssel auf.
2. $k_1 \leq \text{node}(k_2, v, l, r) \stackrel{\text{def}}{\iff} k_1 \leq k_2 \wedge k_1 \leq l \wedge k_1 \leq r$,
denn k_1 ist genau dann kleiner-gleich als alle Schlüssel, die in dem Baum $\text{node}(k_2, v, l, r)$ auftreten, wenn $k_1 \leq k_2$ gilt und wenn zusätzlich k_1 kleiner-gleich als alle Schlüssel ist, die in l oder r auftreten.

Als nächstes definieren wir eine Funktion

$$\text{count} : \mathcal{B} \rightarrow \mathbb{N},$$

die für einen binären Baum die Anzahl der Knoten berechnet. Die Definition erfolgt durch Induktion:

1. $\text{nil.count}() = 0$.
2. $\text{node}(k, v, l, r).\text{count}() = 1 + l.\text{count}() + r.\text{count}()$.

Mit diesen Vorbereitungen können wir nun die Menge *Heap* induktiv definieren:

1. $\text{nil} \in \text{Heap}$.
2. $\text{node}(k, v, l, r) \in \text{Heap}$ g.d.w. folgendes gilt:

- (a) $k \leq l \wedge k \leq r$,

Der Schlüssel an der Wurzel ist also kleiner-gleich als alle anderen Schlüssel. Diese Bedingung bezeichnen wir auch als die *Heap-Bedingung*.

¹Der 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.

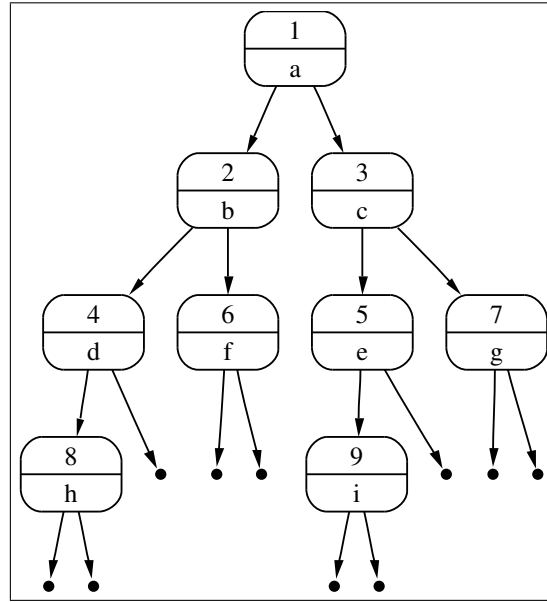


Abbildung 8.1: Ein Heap

(b) $|l.count() - r.count()| \leq 1$,

Die Zahl der Schlüssel im linken Teilbaum ist also höchstens 1 größer oder kleiner als die Zahl der Schlüssel im rechten Teilbaum. Diese Bedingung bezeichnen 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) $l \in \text{Heap} \wedge r \in \text{Heap}$.

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 181 zeigt einen einfachen Heap. In den Knoten steht im oberen Teil die Prioritäten (in der Abbildung sind das natürliche Zahlen) und darunter stehen die Werte (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. $nil.top() = \Omega$.

2. $node(k, v, l, r).top() = \langle k, v \rangle$,

denn aufgrund der Heap-Bedingung wird der Wert 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. $nil.insert(k, v) = node(k, v, nil, nil)$.

2. $k_{top} \leq k \wedge l.count() \leq r.count() \rightarrow$

$$node(k_{top}, v_{top}, l, r).insert(k, v) = node(k_{top}, v_{top}, l.insert(k, v), r).$$

Falls das einzufügende Paar eine geringere oder dieselbe 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. $k_{\text{top}} \leq k \wedge l.\text{count}() > r.\text{count}() \rightarrow$

$$\text{node}(k_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(k, v) = \text{node}(k_{\text{top}}, v_{\text{top}}, l, r.\text{insert}(k, v)).$$

Falls das einzufügende Paar eine geringere oder dieselbe 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. $k_{\text{top}} > k \wedge l.\text{count}() \leq r.\text{count}() \rightarrow$

$$\text{node}(k_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(k, v) = \text{node}(k, v, l.\text{insert}(k_{\text{top}}, v_{\text{top}}), r).$$

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. $k_{\text{top}} > k \wedge l.\text{count}() > r.\text{count}() \rightarrow$

$$\text{node}(k_{\text{top}}, v_{\text{top}}, l, r).\text{insert}(k, v) = \text{node}(k, v, l, r.\text{insert}(k_{\text{top}}, v_{\text{top}})).$$

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. $\text{nil.remove}() = \text{nil}$,

denn aus dem leeren Heap ist nichts mehr zu entfernen.

2. $\text{node}(k, v, \text{nil}, r).\text{remove}() = r$,

3. $\text{node}(k, v, l, \text{nil}).\text{remove}() = l$,

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. $k_1 \leq k_2 \wedge l = \text{node}(k_1, v_1, l_1, r_1) \wedge r = \text{node}(k_2, v_2, l_2, r_2) \rightarrow$

$$\text{node}(k, v, l, r).\text{remove}() = \text{node}(k_1, v_1, l.\text{remove}(), r),$$

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. $k_1 > k_2 \wedge l = \text{node}(k_1, v_1, l_1, r_1) \wedge r = \text{node}(k_2, v_2, l_2, r_2) \rightarrow$

$$\text{node}(k, v, l, r).\text{remove}() = \text{node}(k_2, v_2, l, r.\text{remove}()),$$

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 vermutlich 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.

8.3 Implementierung in Java

```
1 public abstract class HeapNode<Key extends Comparable<Key>, Value>
2 {
3     protected int mCount;
4
5     public abstract Pair<Key, Value> top();
6     public abstract BinaryHeapNode<Key, Value> insert(Key key, Value value);
7     public abstract HeapNode<Key, Value> remove();
8     public abstract boolean isEmpty();
9 }
```

Abbildung 8.2: Die abstrakte Klasse *HeapNode*.

Zunächst implementieren wir eine abstrakte Klasse *HeapNode*. Elemente dieser Klasse sollen Heaps repräsentieren und zwar sowohl leere Heaps als auch nicht-leere Heaps. Abbildung 8.2 auf Seite 183 zeigt die Implementierung. Wir haben eine Member-Variable mit dem Namen `mCount` in Zeile 3 definiert. Diese Variable gibt die Zahl der in dem Heap abgespeicherten Werte an.

Wir werden uns mit der Implementierung der Methode *change()* erst später beschäftigen. Daher fehlt diese Methode in der Klasse *HeapNode*. Statt dessen haben wir eine zusätzliche Methode *isEmpty()*, die wir später bei der Implementierung der Methoden *insert()* und *remove()* benutzen werden.

```
1 public class EmptyHeapNode<Key extends Comparable<Key>, Value>
2     extends HeapNode<Key, Value>
3 {
4     public EmptyHeapNode() {
5         mCount = 0;
6     }
7     public Pair<Key, Value> top() {
8         return null;
9     }
10    public BinaryHeapNode<Key, Value> insert(Key key, Value value) {
11        return new BinaryHeapNode<Key, Value>(key, value);
12    }
13    public HeapNode<Key, Value> remove() {
14        return this;
15    }
16    public boolean isEmpty() {
17        return true;
18    }
19 }
```

Abbildung 8.3: Die Klasse *EmptyHeapNode*.

Abbildung 8.3 auf Seite 183 zeigt die Implementierung der Klasse *EmptyHeapNode*. Elemente dieser Klasse repräsentieren den leeren Heap *nil*. Im Konstruktor setzen wir in Zeile 5 die Member-Variable `mCount` auf 0, denn der leere Heap enthält keine Werte. Die Methode *top()* gibt `null` zurück, denn es gibt ja keinen sinnvollen Wert, den wir hier zurück geben können. Die Methode *insert()* erzeugt einen neuen Knoten vom Typ *BinaryHeapNode*, der dann zurück gegeben wird.

```

1  public class BinaryHeapNode<Key extends Comparable<Key>, Value>
2      extends HeapNode<Key, Value>
3  {
4      private Key          mKey;      // The priority associated with the value.
5      private Value        mValue;    // The value.
6      private HeapNode<Key, Value> mLeft; // The root of the left subtree.
7      private HeapNode<Key, Value> mRight; // The root of the right subtree.
8
9      public BinaryHeapNode(Key key, Value value) {
10         mKey    = key;
11         mValue  = value;
12         mLeft   = new EmptyHeapNode<Key, Value>();
13         mRight  = new EmptyHeapNode<Key, Value>();
14         mCount  = 1;
15     }
16     public Pair<Key, Value> top() {
17         return new Pair<Key, Value>(mKey, mValue);
18     }
19     public BinaryHeapNode<Key, Value> insert(Key key, Value value)
20     {
21         ++mCount;
22         int cmp = key.compareTo(mKey);
23         if (cmp < 0) {
24             if (mLeft.mCount > mRight.mCount) {
25                 mRight = mRight.insert(mKey, mValue);
26             } else {
27                 mLeft = mLeft.insert(mKey, mValue);
28             }
29             mKey    = key;
30             mValue  = value;
31         } else {
32             if (mLeft.mCount > mRight.mCount) {
33                 mRight = mRight.insert(key, value);
34             } else {
35                 mLeft = mLeft.insert(key, value);
36             }
37         }
38         return this;
39     }
40     public boolean isEmpty() {
41         return false;
42     }

```

Abbildung 8.4: Die Klasse *BinaryHeapNode*, Teil I.

Die Klasse *BinaryHeapNode*, deren Implementierung in den Abbildungen 8.4 und 8.5 auf den Seiten 184 und 185 gezeigt wird, repräsentiert einen Knoten der Form

node(mKey, mValue, mLeft, mRight).

Die Implementierung setzt die rekursiven Gleichungen, die wir im vorhergehenden Unterabschnitt gezeigt haben, 1-zu-1 um. Diskutiert werden muss höchstens noch die Implementierung der Methode *remove*. Wenn der Kontrollfluss in Zeile 51 ankommt, dann ist klar, dass weder der linke

Teilbaum `mLeft` noch der rechte Teilbaum `mRight` leer ist. Daher sind diese Teilbäume Objekte der Klasse *BinaryHeapNode* und wir können `mLeft` und `mRight` auf den Typ *BinaryHeapNode* casten. Dies ist notwendig, weil wir auf die Schlüssel, die an der Wurzel dieser Bäume abgespeichert sind, zurückgreifen müssen. Das geht aber nur, wenn diese den Typ *BinaryHeapNode* haben, denn nur in diesem Typ sind die Member-Variablen `mKey` und `mValue` definiert.

```

43     public HeapNode<Key, Value> remove() {
44         --mCount;
45         if (mLeft.isEmpty()) {
46             return mRight;
47         }
48         if (mRight.isEmpty()) {
49             return mLeft;
50         }
51         BinaryHeapNode<Key, Value> left  = (BinaryHeapNode<Key, Value>) mLeft;
52         BinaryHeapNode<Key, Value> right = (BinaryHeapNode<Key, Value>) mRight;
53         Key leftKey  = left .mKey;
54         Key rightKey = right.mKey;
55         if (leftKey.compareTo(rightKey) < 0) {
56             mKey  = left.mKey;
57             mValue = left.mValue;
58             mLeft  = mLeft.remove();
59         } else {
60             mKey  = right.mKey;
61             mValue = right.mValue;
62             mRight = mRight.remove();
63         }
64         return this;
65     }
66 }

```

Abbildung 8.5: Die Klasse *BinaryHeapNode*, Teil II.

8.3.1 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

$h.change(k, v)$

die Priorität des Elements v vergrößert wird. Ist $p = node(k', v, l, r)$ der Knoten, in dem der Wert v gespeichert ist, dann gilt also $k < k'$. 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 den dort gespeicherten Wert 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 Hash-Tabelle² die Zuordnung der Knoten zu den Werten speichern. Damit eine eindeutige Zuordnung von Werten zu Knoten überhaupt möglich ist, gehen wir davon aus, dass jeder Wert höchstens einmal in einem Heap auftritt. Die HashTabelle realisiert dann die Funktion

$nodeMap : Value \rightarrow Node,$

²Wir können an dieser Stelle auch eine AVL-Baum nehmen um die Zuordnung der Knoten zu den Werten zu speichern. Damit dies möglich ist, muss allerdings auf der Menge der Werte eine totale Ordnung existieren.

für welche die Invariante

$$nodeMap(v_1) = node(k, v_2, l, r) \rightarrow v_1 = v_2$$

gilt. Mit andern Worten: Der Aufruf $nodeMap(v)$ gibt den Knoten zurück, in dem der Wert v gespeichert ist.

Wenn wir nun den Knoten $p = node(k', v, l, r)$ gefunden haben, in dem der Wert v gespeichert ist, dann reicht es nicht aus, wenn wir in dem Knoten p einfach die Priorität k' durch k ersetzen, denn es könnte sein, dass dann die Heap-Bedingung verletzt wird und der Schlüssel, der in dem Knoten p gespeichert ist, eine höhere Priorität hat als der Vater-Knoten dieses Knotens. In diesem Fall müssen wir das Paar, das 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

$$node(k, v, l, r)$$

darstellen, haben wir keine Informationen über den Vaterknoten. Wir führen daher zunächst ein Paar Hilfsfunktionen ein.

1. $parent : Node \rightarrow Node \cup \{\Omega\}$

Für jeden Knoten n gibt der Aufruf $n.parent(n)$ den Vaterknoten zurück. Falls zu dem Knoten n kein Vaterknoten existiert, wird statt dessen Ω zurück geliefert.

2. $nodeMap : Value \rightarrow Node \cup \{\Omega\}$

Für einen Wert v liefert der Aufruf $nodeMap(v)$ den Knoten, in dem der Wert v gespeichert ist.

3. $key : Node \rightarrow Key$

Für einen Knoten n liefert der Aufruf $n.key()$ den in dem Knoten gespeicherten Schlüssel zurück.

4. $value : Node \rightarrow Value$

Für einen Knoten n liefert der Aufruf $n.value()$ den in dem Knoten gespeicherten Wert 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 8.6 zeigt den Pseudo-Code, den wir jetzt im Detail diskutieren.

1. Zunächst bezeichnen wir die Priorität, die an dem Knoten n gespeichert ist, mit k_1 und den zugehörigen Wert mit v_1 .
2. Der Vaterknoten von n wird mit p bezeichnet und die Priorität, die dort gespeichert ist, wird mit k_2 , der zugehörige Wert mit v_2 bezeichnet.
Falls der Knoten n bereits der Wurzel-Knoten ist, so existiert kein Vaterknoten und damit kann die Heap-Bedingung auch nicht verletzt sein, so dass die Methode $upheap()$ beendet werden kann.
3. Hat nun der an dem Knoten n gespeicherte Wert eine höhere Priorität als der an dem Vaterknoten p gespeicherte Wert, so werden die Werte (inklusive Prioritäten), die an den Knoten n und p gespeichert sind, vertauscht.

Zusätzlich achten wir darauf, dass die in der Tabelle $nodeMap$ hinterlegte Zuordnung von Werten zu Knoten korrekt bleibt.

```

1  n.upheap() {
2      k1 := n.key();
3      v1 := n.value();
4      p := n.parent();
5      if (p = null) {
6          return;
7      }
8      k2 := p.key();
9      v2 := p.value();
10     if (k1 < k2) {
11         n.key() := k2;
12         n.value() := v2;
13         p.key() := k1;
14         p.value() := v1;
15         nodeMap(v2) := n;
16         nodeMap(v1) := p;
17         p.upheap();
18     }
19 }

```

Abbildung 8.6: Pseudo-Code zur Implementierung der Methode *upheap()*

```

1  import java.util.*;
2
3  public abstract class HeapNode<Key extends Comparable<Key>, Value>
4  {
5      protected int                mCount; // the number of nodes
6      protected BinaryHeapNode<Key, Value> mParent; // parent of this node
7      protected Map<Value, BinaryHeapNode<Key, Value>> mNodeMap;
8
9      public abstract Pair<Key, Value> top();
10     public abstract BinaryHeapNode<Key, Value> insert(Key key, Value value);
11     public abstract HeapNode<Key, Value> remove();
12     public abstract void change(Key k, Value v);
13     public abstract boolean isEmpty();
14 }

```

Abbildung 8.7: Die abstrakte Klasse *HeapNode*.

4. Schließlich müssen wir die Methode *upheap* rekursiv für den Vaterknoten aufrufen.

Die Abbildungen 8.7, 8.8, 8.9, 8.10, 8.10, 8.11 und 8.12 auf den folgenden Seiten zeigen eine vollständige Implementierung des abstrakten Daten-Typs *PrioQueue*. Wir diskutieren nun die Veränderungen gegenüber der bisher gezeigten Implementierung. Wir beginnen mit der abstrakten Klasse *HeapNode*, die in Abbildung 8.7 auf Seite 187 gezeigt wird. Gegenüber der Implementierung in Abbildung 8.2 auf Seite 8.2 gibt es die folgenden Änderungen.

1. Die Klasse enthält eine zusätzliche Member-Variable *mParent*, die in Zeile 6 definiert wird. Hierbei handelt es sich um eine Referenz auf den Vater-Knoten. Diese Referenz ist notwendig für die Implementierung der Methode *change()*, denn dort müssen wir nach einer Änderung der Priorität eines Schlüssel überprüfen, ob die Priorität des Vater-Knotens immer noch

größer ist als die Priorität des Knotens dessen Priorität wir geändert haben.

2. Außerdem enthält jeder Knoten jetzt eine Referenz auf die Abbildung *nodeMap*, in der die Zuordnung der Knoten zu den Werten gespeichert wird. Diese Referenz wird in der in Zeile 7 definierten Member-Variablen *mNodeMap* gespeichert.

```
1  import java.util.*;
2
3  public class EmptyHeapNode<Key extends Comparable<Key>, Value>
4      extends HeapNode<Key, Value>
5  {
6      public EmptyHeapNode(BinaryHeapNode<Key, Value> parent,
7                          Map<Value, BinaryHeapNode<Key, Value>> nodeMap)
8      {
9          mParent = parent;
10         mNodeMap = nodeMap;
11         mCount = 0;
12     }
13     public Pair<Key, Value> top() {
14         return null;
15     }
16     public BinaryHeapNode<Key, Value> insert(Key key, Value value) {
17         BinaryHeapNode<Key, Value> binaryNode =
18             new BinaryHeapNode<Key, Value>(key, value, mParent, mNodeMap);
19         mNodeMap.put(value, binaryNode);
20         return binaryNode;
21     }
22     public HeapNode<Key, Value> remove() { return this; }
23     public void change(Key key, Value value) {}
24     public boolean isEmpty() { return true; }
25 }
```

Abbildung 8.8: Die Klasse *EmptyHeapNode*.

Als nächstes diskutieren wir die Änderungen in der Klasse *EmptyHeapNode*.

1. Da die Klasse nun zwei zusätzliche Member-Variablen von der Klasse *HeapNode* erbt, hat der Konstruktor, der in Zeile 6 – 12 implementiert ist, zwei zusätzliche Argumente, die zur Initialisierung der beiden Member-Variablen *mParent* und *mNodeMap* genutzt werden.
2. Die Implementierung der Methode *insert()* ist nun aufwendiger, denn wir müssen den erzeugten Knoten in die HashTabelle *mNodeMap* eintragen.
3. Die Implementierung der Methode *change()*, die ebenfalls neu hinzu gekommen ist, ist für einen leeren Knoten trivial.

In der Klasse *BinaryHeapNode* gibt es die meisten Änderungen.

1. Zunächst bekommt der Konstruktor zwei zusätzliche Argumente um die Member-Variablen *mParent* und *mNodeMap* zu initialisieren.
2. Bei der Methode *insert()* behandeln wir in den Zeilen 31 – 39 den Fall, dass der neu einzufügende Wert eine höhere Priorität hat als der Wert, der momentan an der Wurzel steht. Daher wird der neu einzufügende Wert jetzt an der Wurzel gespeichert und der Wert, der

```

1  import java.util.*;
2
3  public class BinaryHeapNode<Key extends Comparable<Key>, Value>
4      extends HeapNode<Key, Value>
5  {
6      private Key          mKey;
7      private Value        mValue;
8      private HeapNode<Key, Value> mLeft;
9      private HeapNode<Key, Value> mRight;
10
11     public BinaryHeapNode(Key          key,
12                           Value        value,
13                           BinaryHeapNode<Key, Value> parent,
14                           Map<Value, BinaryHeapNode<Key, Value>> nodeMap)
15     {
16         mKey      = key;
17         mValue     = value;
18         mParent    = parent;
19         mNodeMap   = nodeMap;
20         mLeft      = new EmptyHeapNode<Key, Value>(this, nodeMap);
21         mRight     = new EmptyHeapNode<Key, Value>(this, nodeMap);
22         mCount     = 1;
23     }
24     public Pair<Key, Value> top() {
25         return new Pair<Key, Value>(mKey, mValue);
26     }
27     public BinaryHeapNode<Key, Value> insert(Key key, Value value) {
28         ++mCount;
29         int cmp = key.compareTo(mKey);
30         if (cmp < 0) {
31             mNodeMap.remove(mValue);
32             if (mLeft.mCount > mRight.mCount) {
33                 mRight = mRight.insert(mKey, mValue);
34             } else {
35                 mLeft  = mLeft .insert(mKey, mValue);
36             }
37             mKey      = key;
38             mValue     = value;
39             mNodeMap.put(value, this);
40         } else {
41             if (mLeft.mCount > mRight.mCount) {
42                 mRight = mRight.insert(key, value);
43             } else {
44                 mLeft  = mLeft .insert(key, value);
45             }
46         }
47         return this;
48     }

```

Abbildung 8.9: Die Klasse *BinaryHeapNode*, 1. Teil.

```

49     public HeapNode<Key, Value> remove() {
50         mNodeMap.remove(mValue);
51         if (mLeft.isEmpty()) {
52             mRight.mParent = mParent;
53             return mRight;
54         }
55         if (mRight.isEmpty()) {
56             mLeft.mParent = mParent;
57             return mLeft;
58         }
59         --mCount;
60         BinaryHeapNode<Key, Value> left  = (BinaryHeapNode<Key, Value>) mLeft;
61         BinaryHeapNode<Key, Value> right = (BinaryHeapNode<Key, Value>) mRight;
62         Key leftKey  = left .mKey;
63         Key rightKey = right.mKey;
64         if (leftKey.compareTo(rightKey) < 0) {
65             mKey  = left.mKey;
66             mValue = left.mValue;
67             mLeft  = mLeft.remove();
68         } else {
69             mKey  = right.mKey;
70             mValue = right.mValue;
71             mRight = mRight.remove();
72         }
73         mNodeMap.put(mValue, this);
74         repair();
75         return this;
76     }
77     private void repair() {
78         if (Math.abs(mLeft.mCount - mRight.mCount) <= 1) {
79             return;
80         }
81         if (mLeft.mCount == mRight.mCount + 2) {
82             BinaryHeapNode<Key, Value> left  = (BinaryHeapNode<Key, Value>) mLeft;
83             Key key  = left.mKey;
84             Value value = left.mValue;
85             mLeft  = mLeft.remove();
86             mRight = mRight.insert(key, value);
87             return;
88         } else if (mRight.mCount == mLeft.mCount + 2) {
89             BinaryHeapNode<Key, Value> right = (BinaryHeapNode<Key, Value>) mRight;
90             Key key  = right.mKey;
91             Value value = right.mValue;
92             mRight = mRight.remove();
93             mLeft  = mLeft .insert(key, value);
94             return;
95         }
96     }

```

Abbildung 8.10: Die Methoden *remove* und *repair*.

```

97     public void change(Key key, Value value) {
98         BinaryHeapNode<Key, Value> node = mNodeMap.get(value);
99         node.mKey = key;
100        node.upHeap();
101    }
102    private void upHeap()
103    {
104        if (mParent == null) {
105            return; // heap condition trivially satisfied
106        }
107        Key    parentKey    = mParent.mKey;
108        Value  parentValue = mParent.mValue;
109        if (parentKey.compareTo(mKey) <= 0) {
110            return; // heap condition already satisfied
111        }
112        mNodeMap.put(mValue, mParent);
113        mNodeMap.put(parentValue, this);
114        mParent.mKey    = mKey;
115        mParent.mValue = mValue;
116        mKey            = parentKey;
117        mValue          = parentValue;
118        mParent.upHeap();
119    }
120    public boolean isEmpty() {
121        return false;
122    }
123 }

```

Abbildung 8.11: Die Methoden *change* und *upheap*.

vorher dort stand, wird entweder in den linken Teilbaum `mLeft` oder in den rechten Teilbaum `mRight` eingefügt.

Wir müssen hier darauf achten, dass die HashTabelle `mNodeMap` konsistent bleibt. Daher entfernen wir in Zeile 31 die Zuordnung von `mValue` zu dem Knoten `this` aus der Tabelle und fügen in Zeile 39 statt dessen die Zuordnung von dem neu eingefügten Wert `value` zu dem Knoten `this` ein.

3. Entsprechende Änderungen gibt es auch in der Methode `remove`. Wir löschen zunächst in Zeile 51 die unter dem Schlüssel `mValue` abgespeicherte Zuordnung, denn diesen Wert wollen wir ja entfernen. Da wir anschließend entweder den Wert aus dem linken oder dem rechten Teilbaum nach oben ziehen, müssen wir für den Wert, den wir nach oben gezogen haben, eine Zuordnung in der HashTabelle `mNodeMap` eintragen. Dies geschieht in Zeile 73.

In der Methode `remove` gibt es in den Zeile 52 und 56 noch eine wichtige Änderung: Da wir dort den Knoten im rechten bzw. linken Teilbaum nach oben schieben, müssen wir dessen Zeiger zum Vaterknoten umsetzen, denn sonst würde dieser immer noch auf einen Knoten zeigen, den wir löschen.

4. Zusätzlich sehen Sie in den Zeilen 77 bis 98 noch die Methode `repair()`, mit der die Balancierungs-Bedingung an einem Knoten dann wiederhergestellt werden kann, falls sie durch einen vorhergehenden Aufruf von `remove()` verletzt worden ist. Die Idee, die dieser Methode zu Grunde liegt, ist simpel: Falls ein Teilbaum einen Knoten mehr als erlaubt enthält, so wird in diesem Teilbaum ein Knoten entfernt und dann in den andern Teilbaum eingefügt.

- Bei der Implementierung der Methode *change()* suchen wir zunächst den Knoten, an dem der zu ändernde Wert gespeichert ist. Anschließend ändern wir die in diesem Knoten abgespeicherte Priorität. Dabei kann die Heap-Bedingung verletzt werden: Es könnte sein, dass der Knoten jetzt eine höhere Priorität hat als der Vater dieses Knotens. Um die Heap-Bedingung wiederherzustellen rufen wir daher die Methode *upHeap()* auf.
- Die Methode *upHeap()* prüft zunächst, ob überhaupt ein Vater-Knoten vorhanden ist, denn nur dann kann die Heap-Bedingung verletzt sein. Falls ein Vater-Knoten vorhanden ist, wird als nächstes geprüft, ob die Heap-Bedingung verletzt ist. Wenn dies so ist, dann vertauscht die Methode die Priorität und den Wert, der im Vater-Knoten abgespeichert ist mit der Priorität und dem Wert, der in diesem Knoten abgespeichert ist. Gleichzeitig wird darauf geachtet, dass die HashTabelle *mNodeMap* konsistent bleibt.

Wenn der Wert und die Priorität des aktuellen Knotens an den Vater-Knoten geschoben worden sind, dann kann es passieren, dass nun dort die Heap-Bedingung verletzt ist. Daher wird jetzt für den Vater-Knoten rekursiv die Methode *upHeap()* aufgerufen.

```
1  import java.util.*;
2
3  public class HeapTree<Key extends Comparable<Key>, Value>
4  {
5      HeapNode<Key, Value> mRoot; // this is the node at the root of the tree
6
7      public HeapTree() {
8          Map<Value, BinaryHeapNode<Key, Value>> nodeMap =
9              new HashMap<Value, BinaryHeapNode<Key, Value>>();
10         mRoot = new EmptyHeapNode<Key, Value>(null, nodeMap);
11     }
12     public Pair<Key, Value> top() {
13         return mRoot.top();
14     }
15     public void insert(Key key, Value value) {
16         mRoot = mRoot.insert(key, value);
17     }
18     public void change(Key key, Value value) {
19         mRoot.change(key, value);
20     }
21     public void remove() {
22         mRoot = mRoot.remove();
23     }
24 }
```

Abbildung 8.12: Die Klasse *HeapTree*.

Wir betrachten als letztes die Klasse *HeapTree*, die in Abbildungen 8.12 auf Seite 192 gezeigt wird. Diese Klasse repräsentiert einen vollständigen *Heap*.

- Der Wurzel-Knoten des Heaps wird in der Member-Variablen *mRoot* gespeichert.
- Außerdem verwaltet diese Klasse die HashTabelle *nodeMap*, die die Zuordnung zwischen den Werten und den Knoten herstellt. Diese Tabelle wird in dem Konstruktor in Zeile 8 zunächst als leere Tabelle angelegt. Anschließend wird ein leerer Knoten erzeugt, der eine Referenz auf die gerade angelegte Tabelle erhält. Dieser leere Knoten ist der Wurzel-Knoten des Heaps.

- Die Signaturen der Methoden *insert()*, *change()* und *remove()* sind gegenüber den entsprechenden Methoden in der Klasse *HeapNode* und in den davon abgeleiteten Klassen *EmptyHeapNode* und *BinaryHeapNode* dahingehend verändert, dass diese Methoden nun nichts mehr zurück geben. Statt dessen ändern diese Methoden jetzt den zugrunde liegenden Heap *mRoot*. Diese Änderung der Signaturen ist der Grund für die Existenz der Klasse *HeapTree*: In der Klasse *HeapTree* haben die Methoden *insert()*, *change()* und *remove()* die Signaturen, die wir brauchen und die Methoden ändern den zugrunde liegenden Heap. In der Klasse *HeapNode* sind diese Signaturen so nicht möglich, denn beispielsweise ist es unmöglich, beim Einfügen aus einem *EmptyHeapNode* einen *BinaryHeapNode* zu machen, da sich in Java der Typ eines Objektes zur Laufzeit nicht ändern kann.

8.3.2 Prioritäts-Warteschlangen in Java

In Java werden Prioritäts-Warteschlangen durch die Klasse `PriorityQueue<E>` implementiert. Als Datenstruktur wird dabei ein Feld verwendet um den zu Grunde liegenden binären Baum darzustellen. Bezeichnen wir das Feld mit `mArray`, so gilt:

- Die Wurzel des binären Baums wird in dem Element `mArray[0]` gespeichert.
- Wird der Knoten $node(k, v, l, r)$ als `mArray[i]` abgespeichert, so wird der Knoten an der Wurzel des linken Teilbaums l an der Position $2 \cdot i + 1$ abgespeichert und der Knoten an der Wurzel des rechten Teilbaums r wird an der Position $2 \cdot (i + 1)$ abgespeichert.
- Wird der Knoten n an der Position i abgespeichert, so findet sich der Vater-Knoten von n an der Position $(i - 1)/2$.

Damit eine solche Feld-basierte Darstellung möglich ist, ist es erforderlich, die Balancierungs-Bedingung für Heaps etwas abzuändern. Wir skizzieren kurz die Idee, die der abgeänderten Definition zu Grunde liegt.

- Ein binärer Baum heißt *vollständig*, wenn alle Blätter des Baums den selben Abstand zur Wurzel haben. Bezeichnen wir die Menge der vollständigen binären Bäume mit $\bar{\mathcal{B}}$, so können wir die folgende induktive Definition geben:

- $nil \in \bar{\mathcal{B}}$,
- $node(k, v, l, r) \in \bar{\mathcal{B}} \stackrel{\text{def}}{\iff} l \in \bar{\mathcal{B}} \wedge r \in \bar{\mathcal{B}} \wedge l.height() = r.height()$

Abbildung 8.13 zeigt einen vollständigen Baum der Tiefe 4. Teilen wir die Knoten nach ihrem Abstand h von der Wurzel in verschiedene Schichten ein, so sehen wir, dass es auf der Schicht, die von der Wurzel den Abstand h hat, insgesamt 2^h Knoten gibt.

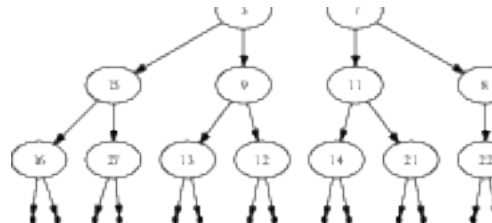


Abbildung 8.13: Ein vollständiger Baum der Tiefe 4.

Stellen wir den in Abbildung 8.13 gezeigten Baum durch ein Feld dar, so erhalten wir das folgende Feld:

[2, 3, 7, 15, 9, 11, 8, 16, 27, 13, 12, 14, 21, 22, 17]

2. Bei einem vollständigen binären Baum sind auf der untersten Ebene alle möglichen Knoten vorhanden. Sie können leicht durch Induktion nachrechnen, dass ein solcher Baum immer $2^h - 1$ Knoten enthält, wobei h die Höhe des Baums angibt. Daraus folgt, dass es nicht möglich ist, einen vollständigen binären Baum zu bilden, der beispielsweise aus 13 Knoten besteht, denn die Zahl 13 ist nicht in der Form $2^h - 1$ darstellbar. Der Begriff des *nahezu vollständigen* binären Baums ist eine Verallgemeinerung des Begriffs des vollständigen binären Baums, der eine beliebige Anzahl von Knoten zulässt.

Ein binärer Baum heißt *nahezu vollständig*, wenn er aus einem binären Baum dadurch entsteht, dass auf der untersten Ebene von rechts nach links Knoten entfernt werden, ohne dass dabei Knoten ausgelassen werden. Abbildung 8.14 auf Seite 194 zeigt einen nahezu vollständigen Baum, der aus dem Baum aus Abbildung 8.13 dadurch entsteht, dass die letzten drei Blätter weggelassen wurden.

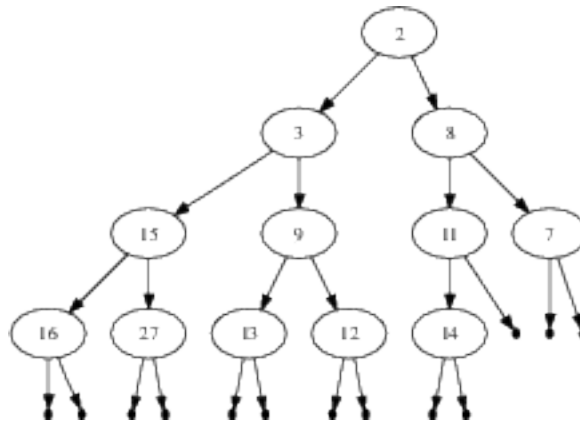


Abbildung 8.14: Ein nahezu vollständiger Baum der Tiefe 4.

Stellen wir diesen Baum durch ein Feld dar, so erhalten wir das folgende Feld:

[2, 3, 8, 15, 9, 11, 7, 16, 27, 13, 12, 14]

Dieses Feld entsteht aus dem vorigen Feld dadurch, dass wir die letzten drei Einträge gelöscht haben. Schließlich zeigt Abbildung 8.15 auf Seite 194 noch einen binären Baum, der nicht nahezu vollständig ist, denn er enthält auf der untersten Ebene eine Lücke, weil die Knoten nicht von rechts nach links entfernt wurden.

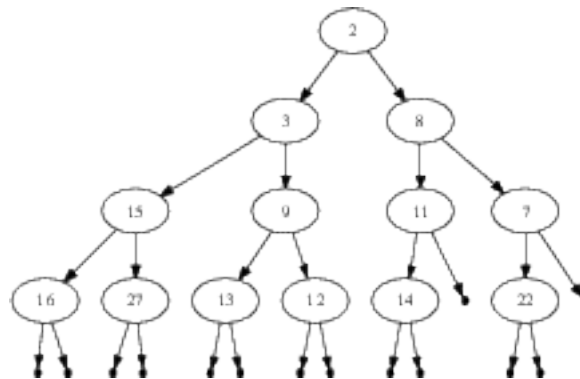


Abbildung 8.15: Ein Baum, der nicht nahezu vollständig ist.

Bezeichnen wir die Menge der nahezu vollständigen binären Bäume mit \mathcal{B}^* , so können wir diese Menge formal durch eine Induktion definieren:

- (a) $n \in \overline{\mathcal{B}} \rightarrow n \in \mathcal{B}^*$,
jeder vollständige binäre Baum ist natürlich erst recht nahezu vollständig.
- (b) $l \in \overline{\mathcal{B}} \wedge l.height() = h \wedge r \in \overline{\mathcal{B}} \wedge r.height() = h - 1 \rightarrow node(k, v, l, r) \in \mathcal{B}^*$,
falls l ein vollständiger binärer Baum der Höhe h ist und r ein vollständiger binärer Baum der Höhe $h - 1$, dann ist $node(k, v, l, r)$ ein nahezu vollständiger binärer Baum.
- (c) $l \in \overline{\mathcal{B}} \wedge l.height() = h \wedge r \in \mathcal{B}^* \wedge r.height() = h \rightarrow node(k, v, l, r) \in \mathcal{B}^*$,
falls l ein vollständiger binärer Baum der Höhe h ist und r ein nahezu vollständiger binärer Baum, der ebenfalls die Höhe h hat, dann ist $node(k, v, l, r)$ ein nahezu vollständiger binärer Baum.
- (d) $l \in \mathcal{B}^* \wedge l.height() = h \wedge r \in \overline{\mathcal{B}} \wedge r.height() = h - 1 \rightarrow node(k, v, l, r) \in \mathcal{B}^*$,
falls l ein nahezu vollständiger binärer Baum der Höhe h ist und r ein vollständiger binärer Baum der Höhe $h - 1$ hat, dann ist $node(k, v, l, r)$ ein nahezu vollständiger binärer Baum.

Nahezu vollständige binäre Bäume lassen sich durch ein Feld darstellen, weil dann für den linken und den rechten Teilbaum keine expliziten Referenzen abgespeichert werden müssen. Das spart Speicherplatz, die Implementierung ist aber etwas komplizierter als die von uns entwickelte Variante.

Wir kommen nun zurück zur Diskussion der Klasse `PriorityQueue`. Diese Klasse enthält die folgenden Konstruktoren.

1. `PriorityQueue()`

Dieser Konstruktor erzeugt einen neuen Prioritäts-Warteschlange. Das zu Grunde liegende Feld hat dabei zunächst die Größe 11. Wenn später der Platz in diesem Feld nicht mehr ausreicht, wird es dynamisch vergrößert.

2. `PriorityQueue(Collection<E> c)`

Dieser Konstruktor erzeugt eine Prioritäts-Warteschlange, die alle Elemente der Zusammenfassung c enthält.

3. `PriorityQueue(int initialCapacity)`

Dieser Konstruktor erzeugt eine neue Prioritäts-Warteschlange. Das zu Grunde liegende Feld hat dabei die Größe `initialCapacity`.

4. `PriorityQueue(int initialCapacity, Comparator<E> comparator)`

Dieser Konstruktor erzeugt eine neue Prioritäts-Warteschlange. Das zu Grunde liegende Feld hat dabei die Größe `initialCapacity`. Die Elemente dieser Warteschlange werden nicht durch einen Aufruf der Methode

`x.compareTo(y)`

verglichen, sondern statt dessen wird der als Argument übergebene Comparator zum Vergleich benutzt:

`comparator.compare(x, y)`.

Die Methoden, die zur Verfügung gestellt werden, kennen wir schon aus unserer Diskussion der Schnittstellen `Collection<E>` und `Queue<E>`.

1. `boolean offer(E e)`

Der Aufruf `q.offer(e)` fügt das Element e in die Prioritäts-Warteschlange q an und entspricht unserer Methode `insert`.

2. **E peek()**

Der Aufruf `q.peek()` liefert das Element mit der höchsten Priorität in der Prioritäts-Warteschlange `q` als Ergebnis. `q` wird dabei nicht verändert. Falls `q` leer ist, wird **null** zurück gegeben.

3. **E poll()**

Der Aufruf `q.poll()` liefert das Element mit der höchsten Priorität in der Prioritäts-Warteschlange `q` als Ergebnis. Das Element wird dabei aus `q` entfernt. Falls `q` leer ist, wird **null** zurück gegeben.

4. **boolean remove(E e)**

Der Aufruf `q.remove(e)` entfernt das Element `e` aus der Prioritäts-Warteschlange `q`. Falls `q` das Element `e` nicht enthält, bleibt `q` unverändert. In diesem Fall wird als Ergebnis **false** zurück gegeben, sonst wird **true** zurück gegeben.

Die Klasse **PriorityQueue** enthält keine Methode, um die Priorität eines Elements zu ändern. Das geht auch gar nicht, denn bei dieser Klasse wird nicht zwischen dem Wert und dem Schlüssel unterschieden, beide sind Teil eines Elements. Um also einen Priorität zu ändern, müssen wir das betreffende Element zunächst aus der Prioritäts-Warteschlange entfernen, die Priorität ändern und anschließend das Element mit der geänderten Priorität wieder einfügen.

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 Σ sind. Die Menge Σ bezeichnen wir als unser *Alphabet*. Wenn das Alphabet aus n verschiedenen Zeichen besteht und wir alle Buchstaben mit derselben Länge von b Bits kodieren wollen, dann muss für diese Zahl von Bits offenbar

$$n \leq 2^b$$

gelten, woraus

$$b = \text{ceil}(\log_2(n))$$

folgt. Hier bezeichnet $\text{ceil}(x)$ die *Ceiling-Funktion*. Diese Funktion rundet eine gegebene reelle Zahl immer auf, es gilt also

$$\text{ceil}(x) = \min\{k \in \mathbb{N} \mid x \leq k\}.$$

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 derselben 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 Σ bestehe nur aus vier Buchstaben,

$$\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}.$$

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 jeweils 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 Kreise 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

$$990 \cdot 1 + 8 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 1012$$

Bits. Gegenüber der ursprünglichen Kodierung, die 2000 Bits verwendet, haben wir 49,4% gespart!

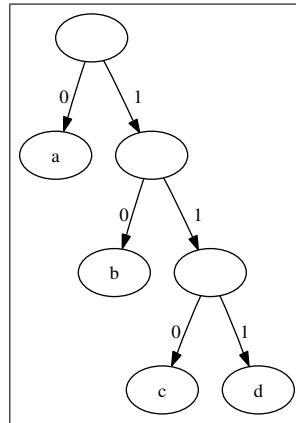


Abbildung 9.1: Baum-Darstellung der Kodierung.

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

$$\text{“100111”} \simeq \text{“bad”}.$$

9.1 Der Algorithmus von Huffman

Angenommen, wir haben einen String s , der aus Buchstaben eines Alphabets Σ 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. $\text{leaf}(c, f) \in \mathcal{K}$ falls $c \in \Sigma$ und $f \in \mathbb{N}$.

Ausdrücke der Form $\text{leaf}(c, f)$ sind die Blätter eines Kodierungs-Baums. Dabei ist c ein Buchstabe aus unserem Alphabet Σ 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. $node(l, r) \in \mathcal{K}$ falls $l \in \mathcal{K}$ und $r \in \mathcal{K}$.

Ausdrücke der Form $node(l, r)$ sind die inneren Knoten eines Kodierungs-Baums.

Als nächstes definieren wir eine Funktion

$$count : \mathcal{K} \rightarrow \mathbb{N},$$

welche die Gesamt-Häufigkeiten aller in dem Baum auftretenden Buchstaben aufsummiert.

1. Die Definition der Funktion $count$ ist für Blätter trivial:

$$leaf(c, f).count() = f.$$

2. Die Gesamt-Häufigkeit des Knotens $node(l, r)$ ergibt sich als Summe der Gesamt-Häufigkeiten von l und r . Also gilt

$$node(l, r).count() = l.count() + r.count().$$

Weiter definieren wir auf Kodierungs-Bäumen die Funktion

$$cost : \mathcal{K} \rightarrow \mathbb{N}.$$

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. $leaf(c, f).cost() = 0$,

denn solange nur ein einziger Buchstabe vorhanden ist, ist noch nichts zu kodieren.

2. $node(l, r).cost() = l.cost() + r.cost() + l.count() + r.count()$.

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

$$l.count() + r.count()$$

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 $node(l, r)$ gegenüber der Kodierung in l und r jeweils um 1 verlängert, müssen wir zu den Kosten der Teilbäume l und r den Term $l.count() + r.count()$ hinzuaddieren.

Wir erweitern die Funktion $cost()$ auf Mengen von Knoten, indem wir die Kosten einer Menge M als die Summe der Kosten der Knoten von M definieren:

$$cost(M) = \sum_{n \in M} n.cost().$$

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

$$M = \{leaf(c_1, f_1), \dots, leaf(c_k, f_k)\} \quad (9.1)$$

Es werden nun solange Knoten a und b aus M zu einem neuen Knoten $node(a, b)$ zusammengefasst, 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

$$M_1 = N \cup \{a, b\} \quad \text{und} \quad M_2 = N \cup \{node(a, b)\}$$

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:

$$\begin{aligned} & cost(N \cup \{node(a, b)\}) - cost(N \cup \{a, b\}) \\ &= cost(\{node(a, b)\}) - cost(\{a, b\}) \\ &= node(a, b).cost() - a.cost() - b.cost() \\ &= a.cost() + b.cost() + a.count() + b.count() - a.cost() - b.cost() \\ &= a.count() + b.count() \end{aligned}$$

Fassen wir die Knoten a und b aus der Menge M zu einem neuen Knoten zusammen, so vergrößern sich die Kosten der Menge um die Summe

$$a.count() + b.count().$$

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 $count()$ den kleinsten Wert liefert. Diese Knoten werden wir aus der Menge M entfernen und durch den neuen Knoten $node(a, b)$ 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. Der in Abbildung 9.2 gezeigte Pseudo-Code beschreibt diesen Algorithmus.

```

1  procedure codingTree(M) {
2      while (#M > 1) {
3          a := minCount(M);
4          M := M - { a };
5          b := minCount(M);
6          M := M - { b };
7          M := M + { node(a, b) };
8      }
9      return arb M;
10 }

```

Abbildung 9.2: Der Algorithmus von Huffman.

1. Die Funktion *codingTree* wird mit einer Menge M von Knoten aufgerufen, welche die in Gleichung (9.1) angegebene Form hat.
2. Die **while**-Schleife verringert die Anzahl der Knoten in der Menge M in jedem Schritt um Eins.
 - (a) Dazu werden mit Hilfe der Funktion *minCount()* die beiden Knoten a und b berechnet, für die der Wert von *count()* minimal ist. Beide Knoten werden aus der Menge M entfernt.

Die Funktion $\text{minCount}(M)$ berechnet den Knoten der Menge M , für den die Funktion $\text{count}()$ den kleinsten Wert annimmt, es gilt also

$$\text{minCount}(M) = m \rightarrow \forall n \in M : m.\text{count}() \leq n.\text{count}()$$

- (b) Anschließend wird aus den beiden Knoten a und b ein neuer Knoten $\text{node}(a, b)$ gebildet. Dieser Knoten wird 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 $\text{minCount}()$. 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 $n.\text{cost}()$ aufsteigend sortiert. Dann ist die Funktion $\text{minCount}()$ zwar sehr effizient, aber die Operation

$$M := M + \{ \text{node}(a, b) \};$$

würde einen Aufwand erfordern, der linear in der Anzahl der Elemente der Menge M ist. Es ist effizienter, die Menge M durch eine Prioritäts-Warteschlange Q darzustellen, denn dann kann die Funktion $\text{minCount}()$ durch $Q.\text{top}()$ realisiert werden, während das Einfügen von $\text{Node}(a, b)$ durch den Aufruf von $\text{insert}()$ realisiert wird.

| | | | | | |
|------------|---|---|---|---|---|
| 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

$$M = \{ \text{leaf}(a, 1), \text{leaf}(b, 2), \text{leaf}(c, 3), \text{leaf}(d, 4), \text{leaf}(e, 5) \}.$$

2. Die Funktion $\text{count}()$ 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

$$\text{node}(\text{leaf}(a, 1), \text{leaf}(b, 2))$$

in die Menge M ein. Es gilt

$$\text{node}(\text{leaf}(a, 1), \text{leaf}(b, 2)).\text{count}() = \text{leaf}(a, 1).\text{count}() + \text{leaf}(b, 2).\text{count}() = 1 + 2 = 3.$$

Um die Funktion $\text{count}()$ nicht jedes Mal neu berechnen zu müssen, annotieren wir den Wert dieser Funktion mit einem Doppelpunkt an einem Knoten n in der Form

$$n : n.\text{count}().$$

Dann hat M die Form

$$\{ \text{node}(\text{leaf}(a, 1), \text{leaf}(b, 2)) : 3, \text{leaf}(c, 3), \text{leaf}(d, 4), \text{leaf}(e, 5) \}.$$

3. Die beiden Knoten mit den kleinsten Werten von count sind nun

$$\text{node}(\text{leaf}(a, 1), \text{leaf}(b, 2)) : 3 \quad \text{und} \quad \text{leaf}(c, 3).$$

Wir entfernen diese beiden Knoten und bilden aus den beiden Knoten den neuen Knoten

$$\text{node}(\text{node}(\text{leaf}(a, 1), \text{leaf}(b, 2)), \text{leaf}(c, 3)) : 6,$$

den wir der Menge M hinzufügen. Dann hat M die Form

$$\{ \text{leaf}(d, 4), \text{leaf}(e, 5), \text{node}(\text{node}(\text{leaf}(a, 1), \text{leaf}(b, 2)), \text{leaf}(c, 3)) : 6 \}.$$

4. Jetzt sind

$leaf(d, 4)$ und $leaf(e, 5)$

die beiden Knoten mit dem kleinsten Werten von *count*. Wir entfernen diese Knoten und bilden den neuen Knoten

$node(leaf(d, 4), leaf(e, 5)) : 9$.

Diesen fügen wir der Menge *M* hinzu und erhalten

$\left\{ node(leaf(d, 4), leaf(e, 5)) : 9, node\left(node(leaf(a, 1), leaf(b, 2)), leaf(c, 3)\right) : 6 \right\}$.

5. Jetzt enthält die Menge *M* nur noch zwei Knoten. Wir entfernen diese Knoten und bilden daraus den neuen Knoten

$node\left(node\left(node(leaf(a, 1), leaf(b, 2)), leaf(c, 3)\right), node(leaf(d, 4), leaf(e, 5))\right) : 15$

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.count()* beschriftet.

Die Kodierung, die sich daraus ergibt, wird in Tabelle 9.3 gezeigt.

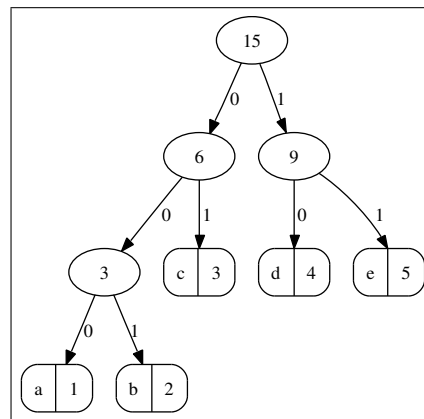


Abbildung 9.3: Baum-Darstellung der Kodierung.

| Buchstabe | a | b | c | d | e |
|-----------|-----|-----|----|----|----|
| Kodierung | 000 | 001 | 01 | 10 | 11 |

Tabelle 9.3: Kodierung der Buchstaben mit variabler Länge.

9.1.1 Implementierung in Java

Wir zeigen nun, wie die Berechnung des Huffman-Codes in *Java* implementiert werden kann. Als erstes präsentieren wir Klassen, um Kodierungs-Bäume darstellen zu können. Abbildung 9.4 zeigt die Implementierung der abstrakten Klasse **Node**, mit der wir Elemente der Menge der Kodierungs-Bäume *K* darstellen. Da wir später Knoten anhand der für diesen Knoten gespeicherten Häufigkeiten vergleichen müssen, implementiert diese Klasse die Schnittstelle **Comparable**. Dazu muss die Klasse **Node** die Methode *compareTo()* bereitstellen. Diese Methode vergleicht verschiedene Knoten über die Werte der Funktion *count()*.

Die Klasse **LeafNode** repräsentiert Knoten der Form $leaf(c, f)$. Der Buchstabe *c* wird in der Member-Variablen **mCharacter** abgespeichert und die Häufigkeit *f*, mit der dieser Buchstabe in dem zu kodierenden String *s* auftritt, wird in der Member-Variablen **mFrequency** abgelegt.

```

1  public abstract class Node implements Comparable<Node> {
2      public abstract Integer cost();
3      public abstract Integer count();
4
5      public int compareTo(Node rhs) {
6          return count().compareTo(rhs.count());
7      }
8
9  }

```

Abbildung 9.4: Die abstrakte Klasse `Node`.

```

1  public class LeafNode extends Node {
2      private char mCharacter;
3      private int mFrequency;
4
5      public LeafNode(char character, int frequency) {
6          mCharacter = character;
7          mFrequency = frequency;
8      }
9      public Integer cost() {
10         return 0;
11     }
12     public Integer count() {
13         return mFrequency;
14     }
15     public Character getCharacter() {
16         return mCharacter;
17     }
18 }

```

Abbildung 9.5: Die Klasse `Leaf`.

1. Der Konstruktor initialisiert die beiden Member-Variablen `mCharacter` und `mFrequency`.
2. Die Funktion `cost()` liefert als Ergebnis 0, denn für Blätter hatten wir

$$\text{leaf}(c, f).\text{cost}() = 0$$

definiert.

3. Die Funktion `count()` liefert die Häufigkeit `mFrequency`, denn für Blätter hatten wir

$$\text{leaf}(c, f).\text{count}() = f$$

definiert.

4. Die Funktion `getCharacter()` gibt als Ergebnis den in der Member-Variablen `mCharacter` gespeicherten Buchstaben zurück.

Die Klasse `BinaryNode` repräsentiert einen binären Knoten der Form $\text{node}(l, r)$. Die Klasse hat vier Member-Variablen:

1. `mLeft` speichert den linken Teilbaum l des Knotens $\text{node}(l, r)$.

```

1  public class BinaryNode extends Node {
2      private Node mLeft;
3      private Node mRight;
4      private int  mCount;
5      private int  mCost;
6
7      public BinaryNode(Node left, Node right) {
8          mLeft  = left;
9          mRight = right;
10         mCount = mLeft.count() + mRight.count();
11         mCost  = mLeft.cost()  + mRight.cost() + mCount;
12     }
13     public Integer cost() {
14         return mCost;
15     }
16     public Integer count() {
17         return mCount;
18     }
19 }

```

Abbildung 9.6: Die Klasse `BinaryNode`.

2. `mRight` speichert den rechten Teilbaum r des Knotens $node(l, r)$.
3. `mCount` speichert den Wert der Funktion $node(l, r).count()$. Wir speichern diesen Wert in einer Member-Variablen, damit wir ihn nur einmal berechnen müssen.
4. `mCost` speichert den Wert der Funktion $node(l, r).cost()$.

Der Konstruktor der Klasse `BinaryNode` bekommt als Argumente den linken Teilbaum l und den rechten Teilbaum r des zu konstruierenden Knotens $node(l, r)$. Außerdem berechnet der Konstruktor die Werte $node(l, r).count()$ und $node(l, r).cost()$ und speichert diese Werte in den Member-Variablen `mCount` und `mCost`.

Die Abbildung 9.7 auf Seite 205 zeigt die Implementierung der Klasse `Huffman`, die für einen gegebenen String den Huffman-Code berechnet. Die Klasse `Huffman` enthält zwei Member-Variablen:

1. `mFrequencyTable` ist eine Abbildung, die für jeden Buchstaben c , der in der gegebenen Datei auftritt, angibt, wie oft dieser Buchstabe in der Datei auftritt.
2. `mCoding` ist ein Knoten der Form $node(l, r)$. Dieser Knoten ist das Endergebnis der Berechnung und enthält den Kodierungs-Baum für den gegebenen Text.

Wir besprechen jetzt die Details der Implementierung des Konstruktors und der Methoden in der Klasse `Huffman`.

1. Der Konstruktor der Klasse `Huffman` bekommt als Argument den Namen der Datei, die den Text enthält, für den der Huffman-Code bestimmt werden soll. Anschließend liest die Methode `determineFrequencies()` die angegebene Datei und berechnet die Häufigkeiten, mit denen die einzelnen Buchstaben auftreten. Diese Häufigkeiten dann werden in der Tabelle `mFrequencyTable` abgespeichert. Daraus berechnet die Methode `createHuffmanCode()` den Huffman-Code mit Hilfe dieser Tabelle.
2. Die Methode `determineFrequencies()` geht davon aus, dass als Zeichen nur Buchstaben aus dem ASCII-Zeichensatz verwendet werden.

```

1  import java.util.*;
2  import java.io.*;
3
4  public class Huffman {
5      Map<Character, Integer> mFrequencyTable;
6      Node                    mCoding;        // the coding tree
7
8      public Huffman(String fileName) {
9          determineFrequencies(fileName);
10         mCoding = createHuffmanCode();
11     }
12     public void determineFrequencies(String fileName) {
13         mFrequencyTable = new TreeMap<Character, Integer>();
14         for (char c = 1; c < 128; ++c) {
15             mFrequencyTable.put(c, 0);
16         }
17         try {
18             FileReader fr = new FileReader(fileName);
19             while (true) {
20                 char c = (char) fr.read();
21                 if (c == 65535) { break; }
22                 int count = mFrequencyTable.get(c);
23                 ++count;
24                 mFrequencyTable.put(c, count);
25             }
26         } catch (IOException e) {
27             e.printStackTrace();
28         }
29     }
30     public Node createHuffmanCode() {
31         PriorityQueue<Node> queue = new PriorityQueue<Node>();
32         for (Character c: mFrequencyTable.keySet()) {
33             Integer frequency = mFrequencyTable.get(c);
34             if (frequency >= 1) {
35                 LeafNode leaf = new LeafNode(c, frequency);
36                 queue.offer(leaf);
37             }
38         }
39         while (queue.size() > 1) {
40             Node left  = queue.remove();
41             Node right = queue.remove();
42             Node node  = new BinaryNode(left, right);
43             queue.offer(node);
44         }
45         return queue.peek();
46     }
47 }

```

Abbildung 9.7: Die Klasse Huffman.

- (a) Zunächst wird daher eine Tabelle für die 127 Zeichen aus dem ASCII-Zeichensatz angelegt. Diese Tabelle soll jedem Buchstaben die Häufigkeit, mit der dieser Buchstabe in

der Datei auftritt, zuordnen. Die Einträge dieser Tabelle werden in der `for`-Schleife in Zeile 15 zunächst mit 0 initialisiert. Später wird jedesmal, wenn wir einen Buchstaben lesen, die dem Buchstaben zugeordnete Häufigkeit inkrementiert.

- (b) Anschließend wird die Datei in Zeile 18 zum Lesen geöffnet.
- (c) In der anschließenden `while`-Schleife wird die Datei zeichenweise gelesen. Falls dabei das Datei-Ende-Zeichen `EOF` gelesen wird, bricht diese Schleife durch den `break`-Befehl in Zeile 21 ab.

Wurde ein Zeichen c gelesen, so wird zunächst in der Tabelle `mFrequencyTable` nachgeschlagen, wie häufig dieses Zeichen schon aufgetreten ist. Diese Zahl wird um 1 inkrementiert und die inkrementierte Zahl wird wieder in die Tabelle zurückgeschrieben. Da die IO-Operationen in Zeile 18 und Zeile 20 Ausnahmen auslösen können, müssen diese Anweisungen in einem `try-catch`-Block eingerahmt werden.

3. Die Methode `createHuffmanCode()` implementiert den Algorithmus von Huffman.

- (a) Zunächst wird in Zeile 31 eine neue Prioritäts-Warteschlange angelegt.
- (b) In der `for`-Schleife in Zeile 32 iterieren wir über alle Zeichen. Falls ein Zeichen mindestens einmal in dem Text vorkommt, erzeugen wir in Zeile 35 einen Knoten `leaf(c, f)` und fügen diesen Knoten der Prioritäts-Warteschlange hinzu.
- (c) Die `while`-Schleife in Zeile 39 implementiert den Pseudo-Code aus Abbildung 9.2: Wir entfernen die beiden Knoten l und r mit den niedrigsten Prioritäten aus der Prioritäts-Warteschlange und bilden den neuen Knoten `node(l, r)`, den wir statt dessen in die Prioritäts-Warteschlange einfügen. Wenn die Prioritäts-Warteschlange nur noch genau einen Knoten enthält, sind wir am Ziel und geben diesen Knoten als Ergebnis zurück.

Aufgabe:

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 $\text{freq}(c)$, so erhalten wir insgesamt

$$n.\text{cost}() = \sum_{c \in \Sigma} \text{freq}(c) \cdot n.\text{depth}(c) \quad (9.2)$$

Die Summe läuft dabei über alle Buchstaben des Alphabets Σ , wobei wir voraussetzen, dass alle Buchstaben aus Σ 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 $n.\text{cost}()$ minimal ist, für alle Kodierungs-Bäume k , bei denen die Buchstaben mit derselben Häufigkeit auftreten wie in n , gilt also

$$n.\text{cost}() \leq k.\text{cost}().$$

Lemma 20 Es seien x und y die beiden Buchstaben aus dem Alphabet Σ 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 $\text{node}(l, r)$, 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. $\text{freq}(x) \leq \text{freq}(y)$,
2. $\text{freq}(a) \leq \text{freq}(b)$.

Da wir angenommen haben, dass x und y die Buchstaben mit der geringsten Häufigkeiten sind, folgt daraus

$$\text{freq}(x) \leq \text{freq}(a) \quad \text{und} \quad \text{freq}(y) \leq \text{freq}(b). \quad (9.3)$$

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

$$n_2.\text{depth}(a) = n_1.\text{depth}(x), \quad (9.4)$$

$$n_2.\text{depth}(b) = n_1.\text{depth}(y), \quad (9.5)$$

$$n_2.\text{depth}(x) = n_1.\text{depth}(a), \quad (9.6)$$

$$n_2.\text{depth}(y) = n_1.\text{depth}(b). \quad (9.7)$$

denn a und x und b und y vertauschen die Plätze. Für alle Buchstaben $c \in \Sigma \setminus \{a, b, x, y\}$ gilt natürlich

$$n_2.\text{depth}(c) = n_1.\text{depth}(c). \quad (9.8)$$

Weiterhin wissen wir aufgrund der Auswahl des Knotens k , dass

$$n_1.\text{depth}(a) = n_1.\text{depth}(b) \geq n_1.\text{depth}(x) \quad \text{und} \quad (9.9)$$

$$n_1.\text{depth}(a) = n_1.\text{depth}(b) \geq n_1.\text{depth}(y) \quad (9.10)$$

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

$$\begin{aligned}
0 &\leq freq(a) \cdot (depth(a) - depth(x)) + freq(b) \cdot (depth(b) - depth(y)) \\
&\quad - freq(x) \cdot (depth(a) - depth(x)) - freq(y) \cdot (depth(b) - depth(y))
\end{aligned}$$

Diese Ungleichung vereinfachen wir zu

$$0 \leq \underbrace{(freq(a) - freq(x))}_{\geq 0} \cdot \underbrace{(depth(a) - depth(x))}_{\geq 0} + \underbrace{(freq(b) - freq(y))}_{\geq 0} \cdot \underbrace{(depth(b) - depth(y))}_{\geq 0}$$

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. \square

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 Σ .

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

$$k = \text{node}(\text{leaf}(a, \text{freq}(a)), \text{leaf}(b, \text{freq}(b))).$$

Bei der Kodierung eines Alphabets, das aus zwei Buchstaben besteht, haben wir keine Wahl, was die Länge der Codes 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 Σ 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 Σ auftritt. Wir definieren ein neues Alphabet Σ' als

$$\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}.$$

Die Häufigkeit des neuen Buchstabens z definieren wir als

$$\text{freq}(z) := \text{freq}(x) + \text{freq}(y).$$

Dann enthält das Alphabet Σ' insgesamt n Buchstaben. Wenden wir den Huffman-Algorithmus auf dieses Alphabet an, so erhalten wir nach Induktions-Voraussetzung für Σ' einen optimalen Kodierungs-Baum k_1 . Die Anwendung des Huffman-Algorithmus auf das Alphabet Σ ersetzt in diesem Kodierungs-Baum das Blatt, das mit dem Buchstaben z markiert ist, durch den Knoten

$$\text{node}(\text{leaf}(x, \text{freq}(x)), \text{leaf}(y, \text{freq}(y))).$$

Bezeichnen wir den so entstandenen 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 Σ , so dass

$$k_3.\text{cost}() < k_2.\text{cost}()$$

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

$$\text{node}(\text{leaf}(x, \text{freq}(x)), \text{leaf}(y, \text{freq}(y))).$$

Wir transformieren den Kodierungs-Baum k_3 in einen Kodierungs-Baum k_4 für das Alphabet Σ' indem wir diesen Knoten durch das Blatt

$$\text{leaf}(z, \text{freq}(x) + \text{freq}(y))$$

ersetzen. Damit gilt

$$\begin{aligned} k_4.\text{cost}() &= \sum_{c \in \Sigma'} \text{freq}(c) \cdot k_4.\text{depth}(c) \\ &= \sum_{c \in \Sigma \setminus \{x, y\} \cup \{z\}} \text{freq}(c) \cdot k_4.\text{depth}(c) \\ &= \sum_{c \in \Sigma \setminus \{x, y\}} \text{freq}(c) \cdot k_4.\text{depth}(c) + \text{freq}(z) \cdot k_4.\text{depth}(z) \\ &= \sum_{c \in \Sigma \setminus \{x, y\}} \text{freq}(c) \cdot k_3.\text{depth}(c) + \text{freq}(z) \cdot (k_3.\text{depth}(x) - 1) \end{aligned}$$

$$\begin{aligned}
&= \sum_{c \in \Sigma \setminus \{x, y\}} \text{freq}(c) \cdot k_3.\text{depth}(c) \\
&\quad + (\text{freq}(x) + \text{freq}(y)) \cdot (k_3.\text{depth}(x) - 1) \\
&= \sum_{c \in \Sigma} \text{freq}(c) \cdot k_3.\text{depth}(c) - (\text{freq}(x) + \text{freq}(y)) \\
&= k_3.\text{cost}() - (\text{freq}(x) + \text{freq}(y))
\end{aligned}$$

Wir halten dieses Ergebnis in einer Gleichung fest:

$$k_4.\text{cost}() = k_3.\text{cost}() - (\text{freq}(x) + \text{freq}(y)). \quad (9.11)$$

Da die Kodierungs-Bäume k_1 und k_2 in derselben Relation stehen wie die Kodierungs-Bäume k_4 und k_3 , gilt analog

$$k_1.\text{cost}() = k_2.\text{cost}() - (\text{freq}(x) + \text{freq}(y)). \quad (9.12)$$

Damit können wir zeigen, dass die Kosten des Kodierungs-Baums k_4 geringer sind als die Kosten des Kodierungs-Baums k_1 :

$$\begin{aligned}
k_4.\text{cost}() &= k_3.\text{cost}() - (\text{freq}(x) + \text{freq}(y)) \\
&< k_2.\text{cost}() - (\text{freq}(x) + \text{freq}(y)) \\
&= k_1.\text{cost}().
\end{aligned}$$

Dieses Ergebnis steht aber im Widerspruch dazu, dass der Kodierungs-Baum k_1 optimal ist. Folglich ist die Annahme $k_3.\text{cost}() < k_2.\text{cost}()$ falsch und der Kodierungs-Baum k_2 ist bereits optimal. \square

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. $\|\cdot\| : \mathbb{E} \rightarrow \mathbb{N} \setminus \{0\}$ ist eine Funktion, die jeder Kante eine positive Länge zuordnet. □

Ein *Pfad* P ist eine Liste der Form

$$P = [x_1, x_2, x_3, \dots, x_n]$$

so dass für alle $i = 1, \dots, n-1$ gilt:

$$\langle x_i, x_{i+1} \rangle \in \mathbb{E}.$$

Die Menge aller Pfade bezeichnen wir mit \mathbb{P} . Die Länge eines Pfades definieren wir als die Summe der Länge aller Kanten:

$$\|[x_1, x_2, \dots, x_n]\| := \sum_{i=1}^{n-1} \|\langle x_i, x_{i+1} \rangle\|.$$

Ist $p = [x_1, x_2, \dots, x_n]$ ein Pfad, so sagen wir, dass p den Knoten x_1 mit dem Knoten x_n verbindet. Die Menge aller Pfade, die den Knoten v mit dem Knoten w verbinden, bezeichnen wir als

$$\mathbb{P}(v, w) := \{[x_1, x_2, \dots, x_n] \in \mathbb{P} \mid x_1 = v \wedge x_n = w\}.$$

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 $\text{source} \in \mathbb{V}$. Dann besteht das kürzeste-Wege-Problem darin, die folgende Funktion zu berechnen:

$$\begin{aligned} \text{sp} : \mathbb{V} &\rightarrow \mathbb{N} \\ \text{sp}(v) &:= \min\{\|p\| \mid p \in \mathbb{P}(\text{source}, v)\}. \end{aligned} \quad \square$$

10.1.1 Ein naiver Algorithmus zur Lösung des kürzeste-Wege-Problems

Als erstes betrachten wir einen ganz naiven Algorithmus zur Lösung des kürzeste-Wege-Problems. Die Idee ist, dass wir eine Funktion

$$\text{dist} : \mathbb{V} \rightarrow \mathbb{N} \cup \{\Omega\}$$

definieren, die für einen Punkt $u \in \mathbb{V}$ eine obere Abschätzung des Abstandes zum Knoten *source* angibt, es soll also immer gelten:

$$\text{dist}(u) \neq \Omega \rightarrow \text{sp}(u) \leq \text{dist}(u).$$

Die Funktion *dist* liefert zu einem Knoten x die kürzeste bisher bekannte Entfernung zum Knoten *source*. Solange noch kein Pfad von *source* zu dem Knoten u gefunden worden ist, gilt $\text{dist}(u) = \Omega$. Anfangs ist die Funktion *dist*() also nur für den Knoten *source* definiert, es gilt

$$\text{dist}(\text{source}) = 0.$$

Später, wenn wir für einen Knoten u einen Pfad gefunden haben, der den Knoten *source* mit dem Knoten u verbindet und wenn es zusätzlich eine Kante $\langle u, v \rangle$ gibt, die den Knoten u mit einem anderen Knoten v verbindet, dann wissen wir, dass auch der Knoten v von *source* erreichbar ist. Zusätzlich wissen wir, dass dieser Weg die Länge $\text{dist}(u) + \|\langle u, v \rangle\|$ hat. Falls bisher also $\text{dist}(v)$ undefiniert war, weil wir noch keinen Weg gefunden hatten, der *source* mit v verbindet, können wir

$$\text{dist}(v) := \text{dist}(u) + \|\langle u, v \rangle\|$$

setzen. Diese Zuweisung ist ebenfalls gültig wenn $\text{dist}(v)$ bereits definiert ist aber einen Wert hat, der größer als $\text{dist}(u) + \|\langle u, v \rangle\|$ ist. Wir fassen diese Überlegungen in den beiden ASM-Regeln zusammen, die in Abbildung 10.1 dargestellt sind. Die Abkürzung ASM steht für *abstract state machine*. Dieser Begriff wurde von Yuri Gurevich [Gur91] eingeführt und von Egon Börger [BS03] zur Spezifikation und Verifikation von Algorithmen propagiert und weiterentwickelt. ASMs sind eine Art Pseudo-Code. Die wesentlichen Eigenschaften von ASMs sind wie folgt:

1. ASMs bestehen aus Regeln. Dabei besteht jede Regel aus einem Namen, einer Bedingung und einer Menge von Zuweisungen.
2. Bei der Abarbeitung der Regeln wird willkürlich eine Regel ausgewählt, deren Bedingung wahr ist und die Zuweisungen dieser Regel werden ausgeführt.
3. Bei Zuweisungen können nicht nur Variablen geändert werden, sondern es können auch die Werte, die eine Funktion an einer Stelle annimmt, verändert werden können. Eine Zuweisung kann also die Form

$$f(x) := y$$

haben. Diese Zuweisung ändert die Funktion f so ab, dass die Funktion anschließend an der Stelle x den Wert y annimmt.

4. Wenn es keine Regel mehr gibt, deren Bedingung wahr ist, dann hält die ASM an.
5. Die Zuweisungen einer Regel werden alle gleichzeitig ausgeführt. In der Regel

```
Rule swap
  if x < y then
    x := y;
    y := x;
  endif
```

werden die beiden Zuweisungen also gleichzeitig ausgeführt, so dass im Endeffekt die Werte von x und y vertauscht werden.

Wie ASMs im Detail funktionieren, erklären wir bei der Diskussion des in Abbildung 10.1 gegebenen Beispiels.

```

1  Rule Init
2      if  $\text{dist}(\text{source}) = \Omega$ 
3      then
4           $\text{dist}(\text{source}) := 0;$ 
5      endif
6
7  Rule Run
8      if choose  $\langle u, v \rangle \in \mathbb{E}$  satisfying
9           $\text{dist}(u) \neq \Omega$  and  $(\text{dist}(v) = \Omega \text{ or } \text{dist}(u) + \|\langle u, v \rangle\| < \text{dist}(v))$ 
10     then
11          $\text{dist}(v) := \text{dist}(u) + \|\langle u, v \rangle\|;$ 
12     endif

```

Abbildung 10.1: ASM-Regeln zur Lösung des kürzeste-Wege-Problems.

1. Die erste Regel hat den Namen *Init*. In dieser Regel wird $\text{dist}(\text{source})$ auf den Wert 0 gesetzt, wenn die Funktion dist an der Stelle source noch undefiniert ist. Diese Regel kann nur einmal ausgeführt werden, denn nach Ausführung der Regel ist $\text{dist}(\text{source})$ von Ω verschieden.
2. Die zweite Regel benutzt das Konstrukt **choose**. Dieses Konstrukt hat allgemein die Form

$$\text{choose } \langle x_1, \dots, x_n \rangle \in M : F(x_1, \dots, x_n)$$

Hierbei sind x_1, \dots, x_n verschiedene Variablen, M ist eine Menge von n -Tupeln und $F(x_1, \dots, x_n)$ ist eine logische Formel, in der diese Variablen auftreten. Das **choose**-Konstrukt liefert genau dann als Ergebnis **true** zurück, wenn es in der Menge M ein Tupel $\langle t_1, \dots, t_n \rangle$ gibt, so dass die Formel $F(t_1, \dots, t_n)$ wahr wird. In diesem Fall werden gleichzeitig die Variablen x_1, \dots, x_n mit den entsprechenden Elementen belegt.

Bei der zweiten Regel suchen wir über das **choose**-Konstrukt eine Kante $\langle u, v \rangle$, für die gilt:

- (a) $\text{dist}(u)$ ist definiert, es gibt also einen Pfad von dem Knoten source zu dem Knoten u .
- (b) $\text{dist}(v)$ ist undefiniert oder größer als $\text{dist}(u) + \|\langle u, v \rangle\|$.

Dann können wir die Abschätzung für den Abstand $\text{dist}(v)$ von dem Knoten source zu dem Knoten v zu dem Wert $\text{dist}(u) + \|\langle u, v \rangle\|$ verbessern.

Der Algorithmus um das kürzeste-Wege-Problem zu lösen besteht nun darin, dass wir die ASM-Regeln solange ausführen, wie dies möglich ist. Der Algorithmus terminiert, denn die Regel *Init* kann nur einmal ausgeführt werden und die Regel *Run* vermindert bei jeder Ausführung den Wert der Funktion $\text{dist}()$ an einem Punkt. Da der Werte-Bereich dieser Funktion aus natürlichen Zahlen besteht, geht das nur endlich oft.

10.1.2 Der Algorithmus von Moore

Der oben gezeigte Algorithmus lässt sich zwar prinzipiell implementieren, er ist aber viel zu ineffizient um praktisch nützlich zu sein. Beim naiven Algorithmus ist die Frage, in welcher Reihenfolge Knoten ausgewählt werden, nicht weiter spezifiziert. Edward F. Moore [Moo59] hat den Algorithmus in naheliegender Weise verbessert, indem er über die Auswahl der Knoten Buch führte. Dazu benutzen wir die Variable *Fringe*, die die Menge aller Knoten enthält, von denen aus noch kürzere Pfade gefunden werden können. Am Anfang enthält diese Menge nur den Knoten **source**. Jedesmal, wenn für einem Knoten v die Funktion $\text{dist}(v)$ geändert wird, wird v der Menge *Fringe* hinzugefügt. Umgekehrt wird v aus der Menge *Fringe* entfernt, wenn alle Kanten, die von v ausgehen, betrachtet worden sind. Um leichter über diese Kanten iterieren zu können, nehmen wir an, dass eine Funktion

```

1  Rule Init
2      if     $\text{dist}(\text{source}) = \Omega$ 
3      then  $\text{dist}(\text{source}) := 0;$ 
4           $\text{mode} := \text{scan};$ 
5           $\text{Fringe} := \{ \text{source} \};$ 
6      endif
7
8  Rule Scan
9      if     $\text{mode} = \text{scan}$  and choose  $u \in \text{Fringe}$ 
10     then
11          $\mathcal{E} := \text{edges}(u);$ 
12          $\text{Fringe} := \text{Fringe} \setminus \{u\};$ 
13          $\text{mode} := \text{relabel};$ 
14     endif
15
16  Rule Relabel
17     if     $\text{mode} = \text{relabel}$ 
18     and    choose  $\langle u, v \rangle \in \mathcal{E}$  satisfying
19              $\text{dist}(v) = \Omega$  or  $\text{dist}(u) + \|\langle u, v \rangle\| < \text{dist}(v);$ 
20     then
21          $\text{dist}(v) := \text{dist}(u) + \|\langle u, v \rangle\|;$ 
22          $\text{Fringe} := \text{Fringe} \cup \{v\};$ 
23     else
24          $\text{mode} := \text{scan};$ 
25     endif

```

Abbildung 10.2: Algorithmus von Moore zur Lösung des kürzeste-Wege-Problems.

$$\text{edges} : \mathbb{V} \rightarrow 2^{\mathbb{E}}$$

gegeben ist, die für einen gegebenen Knoten u die Menge aller Kanten $\langle u, v \rangle$ berechnet, die von dem Knoten u ausgehen. Es gilt also

$$\text{edges}(u) = \{ \langle u, v \rangle \mid \langle u, v \rangle \in \mathbb{E} \}.$$

Der Algorithmus läuft nun in drei Phasen ab.

1. In der *Initialisierungs*-Phase setzen wir $\text{dist}(\text{source}) := 0$.
2. In der *Scanning*-Phase wählen wir einen Knoten u aus der Menge *Fringe* aus, entfernen ihn aus dieser Menge und setzen

$$\mathcal{E} := \text{edges}(u).$$

Anschließend wechseln wir in die *Relabeling*-Phase.

3. In der *Relabeling*-Phase wählen wir eine Kante $\langle u, v \rangle \in \mathcal{E}$ aus, für die

$$\text{dist}(v) = \Omega \quad \text{oder} \quad \text{dist}(u) + \|\langle u, v \rangle\| < \text{dist}(v)$$

gilt. Dann ändern wir die Abstands-Funktion dist für den Knoten v ab und fügen gleichzeitig den Knoten v der Menge *Fringe* hinzu.

Falls wir keinen Knoten finden können, für den wir die Funktion $\text{dist}(u)$ verkleinern können, wechseln wir wieder in die *Scanning*-Phase zurück.

Der Algorithmus bricht ab, wenn die Menge *Fringe* leer wird. Abbildung 10.2 auf Seite 214 zeigt die Spezifikation dieses Algorithmus durch eine ASM.

10.1.3 Der Algorithmus von Dijkstra

```
1  Rule Init
2      if    dist(source) =  $\Omega$ 
3      then
4          Fringe.insert(0, source);
5          dist(source) := 0;
6          Visited      := { source };
7          mode         := scan;
8      endif
9
10 Rule Scan
11     if      mode = scan
12     and not Fringe.isEmpty()
13     then
14          $\langle d, u \rangle := \text{Fringe.top}()$ ;
15         Fringe.remove();
16         Visited := Visited  $\cup$  { u };
17          $\mathcal{E}$       := edges(u);
18         mode := relabel;
19     endif
20
21 Rule Relabel
22     if      mode = relabel
23     and     choose  $\langle u, v \rangle \in \mathcal{E}$  satisfying
24             dist(v) =  $\Omega$  or dist(u) +  $\|\langle u, v \rangle\| < \text{dist}(v)$ ;
25     then
26         dist(v) := dist(u) +  $\|\langle u, v \rangle\|$ ;
27         if dist(v) =  $\Omega$  then
28             Fringe := Fringe.insert(dist(v), v);
29         else
30             Fringe := Fringe.change(dist(v), v);
31         endif
32     else
33         mode := scan;
34     endif
```

Abbildung 10.3: ASM-Regeln für den Algorithmus von Dijkstra.

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, in der Regel *Scan* immer den Knoten auszuwählen, der den geringsten Abstand zu dem Knoten *source* hat. Dazu wird die Menge *Fringe* durch eine Prioritäts-Warteschlange implementiert. Als Prioritäten wählen wir die Entfernungen zu dem Knoten *source*. Abbildung 10.3 auf Seite 215 zeigt die Spezifikation des Algorithmus von Dijkstra zur Berechnung der kürzesten Wege. Gegenüber dem Algorithmus von Moore hat sich vor allem die Regel *Scan* geändert, denn dort wählen wir jetzt immer den Knoten aus der Menge *Fringe*, der den kleinsten Abstand zum Knoten *source* hat.

In den ASM-Regeln 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, die aus der Prioritäts-Warteschlange *Fringe* entfernt wurden und für die dann anschließend in der Regel *Relabel* alle benachbarten Knoten untersucht wurden. Die Menge *Visited* hat

keine Bedeutung für die eigentliche Implementierung des Algorithmus. Die Variable wird eingeführt um eine Invariante formulieren zu können, die für den Beweis der Korrektheit des Algorithmus zentral ist. Die Invariante lautet

$$\forall u \in \text{Visited} : \text{dist}(u) = \text{sp}(u).$$

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 $\text{dist}(u) = \text{sp}(u)$ gilt. In den ASM-Regeln gibt es zwei Stellen, bei denen wir der Menge *Visited* neue Elemente hinzufügen.

I.A.: In Zeile 6 fügen wir den Start-Knoten *source* in die Menge *Visited* ein. Wegen $\text{sp}(\text{source}) = 0$ ist die Behauptung in diesem Fall offensichtlich.

I.S.: In Zeile 16 fügen wir den Knoten *u* in die Menge *Visited* ein. Wir betrachten nun die Situation unmittelbar vor dem Einfügen von *u*. Wir können annehmen, dass *u* noch nicht in der Menge *Visited* enthalten ist, denn sonst wird *u* ja nicht wirklich in *Visited* eingefügt. Wir führen den Beweis nun indirekt und nehmen an, dass

$$\text{dist}(u) > \text{sp}(u)$$

gilt. Dann gibt es einen kürzesten Pfad

$$p = [x_0 = \text{source}, x_1, \dots, x_n = u]$$

von *source* nach *u*, der insgesamt die Länge $\text{sp}(u)$ hat. Es sei $i \in \{0, \dots, n-1\}$ der Index für den

$$x_0 \in \text{Visited}, \dots, x_i \in \text{Visited} \quad \text{aber} \quad x_{i+1} \notin \text{Visited},$$

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 $\text{dist}(x_{i+1})$ neu berechnet und der Knoten x_{i+1} wurde spätestens zu diesem Zeitpunkt in die Menge *Fringe* eingefügt. Außerdem wissen wir, dass $\text{dist}(x_{i+1}) = \text{sp}(x_{i+1})$ gilt, denn nach Induktions-Voraussetzung gilt $\text{dist}(x_i) = \text{sp}(x_i)$ 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 $x_{i+1} \notin \text{Visited}$ ist, muss x_{i+1} immer noch in der Prioritäts-Warteschlange *Fringe* liegen. Also muss $\text{dist}(x_{i+1}) \geq \text{dist}(u)$ gelten, denn sonst wäre x_{i+1} vor *u* aus der Prioritäts-Warteschlange entfernt worden. Wegen $\text{sp}(x_{i+1}) = \text{dist}(x_{i+1})$ haben wir dann aber den Widerspruch

$$\text{sp}(u) \geq \text{sp}(x_{i+1}) = \text{dist}(x_{i+1}) \geq \text{dist}(u) > \text{sp}(u). \quad \square$$

10.1.4 Implementierung in Java

Zunächst müssen wir überlegen, wie wir einen Graphen repräsentieren wollen. Abbildung 10.4 zeigt die Klasse **Node**, mit der wir die Knoten des Graphen repräsentieren.

1. Die Klasse **Node** implementiert die Schnittstelle **Comparable**, damit wir später Knoten als Schlüssel einer **TreeMap** verwenden können. Dies ist bei der Funktion *dist()* erforderlich.
2. Ein Knoten hat einen eindeutigen Namen, der in der Member-Variablen **mName** abgespeichert wird. Dieser Name ist beim Einlesen eines Graphen nützlich.
3. Weiterhin verwaltet ein Knoten eine Liste von Kanten in der Member-Variablen **mEdges**. Diese Liste repräsentiert den Funktionswert *edges(this)*.
4. Die Methode *compareTo()* vergleicht Knoten anhand ihres Namens.

```

1  import java.util.*;
2
3  public class Node implements Comparable<Node>
4  {
5      private String      mName;
6      private List<Edge> mEdges;
7
8      public Node(String name) {
9          mName = name;
10         mEdges = new LinkedList<Edge>();
11     }
12     public String      toString() { return mName; }
13     public String      getName () { return mName; }
14     public List<Edge> getEdges() { return mEdges; }
15     public void        setEdges(List<Edge> edges) { mEdges = edges; }
16
17     public int compareTo(Node node) {
18         return mName.compareTo(node.mName);
19     }
20 }

```

Abbildung 10.4: Die Klasse Node.

```

1  class Edge {
2      private Node      mSource;
3      private Node      mTarget;
4      private Integer mLength;
5
6      public Edge(Node source, Node target, Integer length) {
7          mSource = source;
8          mTarget = target;
9          mLength = length;
10     }
11     public Node      getSource() { return mSource; }
12     public Node      getTarget() { return mTarget; }
13     public Integer getLength() { return mLength; }
14     public String    toString () {
15         return "<" + mSource + ", " + mTarget + ">: " + mLength;
16     }
17 }

```

Abbildung 10.5: Die Klasse Edge.

Die Klasse `Edge` repräsentiert eine Kante $\langle x, y \rangle$ in unserem Graphen.

1. Die Variable `mSource` entspricht dem Start-Knoten x der Kante $\langle x, y \rangle$.
2. Die Variable `mTarget` entspricht dem Ziel-Knoten y der Kante $\langle x, y \rangle$.
3. Die Variable `mLength` gibt die Länge der Kante $\langle x, y \rangle$ an.

Abbildung 10.6 auf Seite 218 zeigt eine Implementierung des von Dijkstra vorgeschlagenen Algorithmus in *Java*. Die Methode `shortestPath()` bekommt als Argument einen Knoten `source`. Sie

```

1  public class Dijkstra {
2      :
3      public Map<Node, Integer> shortestPath(Node source)
4      {
5          Map<Node, Integer> dist = new TreeMap<Node, Integer>();
6          dist.put(source, 0);
7          HeapTree<Integer, Node> fringe = new HeapTree<Integer, Node>();
8          fringe.insert(0, source);
9          while (!fringe.isEmpty()) {
10             Pair<Integer, Node> p      = fringe.top();
11             Integer             distU = p.getFirst();
12             Node                u      = p.getSecond();
13             fringe.remove();
14             for (Edge edge: u.getEdges()) {
15                 Node v = edge.getTarget();
16                 if (dist.get(v) == null) {
17                     Integer d = distU + edge.getLength();
18                     dist.put(v, d);
19                     fringe.insert(d, v);
20                 } else {
21                     Integer oldDist = dist.get(v);
22                     Integer newDist = dist.get(u) + edge.getLength();
23                     if (newDist < oldDist) {
24                         dist.put(v, newDist);
25                         fringe.change(newDist, v);
26                     }
27                 }
28             }
29         }
30         return dist;
31     }
32 }

```

Abbildung 10.6: Dijkstra's Algorithmus zur Lösung des kürzeste-Wege-Problems.

berechnet den Abstand aller anderen Knoten zu diesem Knoten.

1. In Zeile 5 und 6 initialisieren wir die Funktion *dist* und implementieren die Zuweisung $dist(source) := 0$.
2. In Zeile 7 und 8 wird die Menge **fringe** initialisiert. Diese Menge repräsentieren wir durch eine Prioritäts-Warteschlange, wobei wir nicht die von *Java* zur Verfügung gestellte Klasse benutzen sondern die Klasse, die wir im Kapitel 8 entwickelt haben. Dies ist erforderlich, weil die von *Java* zur Verfügung gestellte Klasse *PriorityQueue* keine Methode *change()* anbietet, mit der die Priorität eines Elementes geändert werden kann.
Am Anfang enthält die Prioritäts-Warteschlange *fringe* nur den Knoten *source*.
3. Die **while**-Schleife in Zeile 9 – 29 implementiert die Scanning-Phase. Solange die Prioritäts-Warteschlange *fringe* nicht leer ist, holen wir den Knoten *u* mit dem kürzesten Abstand zum Knoten *source* aus der Warteschlange heraus.
4. Die Relabeling-Phase wird durch die **for**-Schleife in Zeile 18 – 27 implementiert. Hierbei

iterieren wir über alle Kanten $\langle u, v \rangle$, die von dem Knoten u ausgehen. Dann gibt es zwei Fälle:

- (a) Falls die Funktion $dist$ für den Knoten v noch undefiniert ist, dann realisieren wir in Zeile 17 die Zuweisung

$$dist(v) := dist(u) + \|\langle u, v \rangle\|.$$

Gleichzeitig fügen wir den Knoten v in die Menge $Fringe$ ein.

- (b) Andernfalls ist $dist(v)$ schon definiert. Dann kommt es darauf an, ob der neu entdeckte Weg von $source$ nach v über u kürzer ist als die Länge des bisher gefundenen Pfades. Falls dies so ist, ändern wir die Funktion $dist$ entsprechend ab. Gleichzeitig müssen wir die Priorität des Knotens v in der Warteschlange erhöhen.

10.1.5 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

$$sp(u) = dist(u)$$

gilt. Daraus folgt aber notwendigerweise, dass der Knoten u nie wieder in die Menge $Fringe$ eingefügt werden kann, denn ein Knoten v wird nur dann in $Fringe$ neu eingefügt, wenn die Funktion $dist(v)$ noch undefiniert ist. Das Einfügen eines Knoten in eine Prioritäts-Warteschlange mit n Elementen kostet eine Rechenzeit, die durch $\mathcal{O}(\log_2(n))$ abgeschätzt werden kann. Da die Warteschlange sicher nie mehr als $\#V$ Knoten enthalten kann und da jeder Knoten maximal einmal eingefügt werden kann, liefert das einen Term der Form

$$\mathcal{O}(\#V \cdot \log_2(\#V))$$

für das Einfügen der Knoten. Neben dem Aufruf von $fringe.insert(d, v)$ müssen wir auch die Komplexität des Aufrufs $fringe.change(newDist, v)$ analysieren. Die Anzahl dieser Aufrufe ist durch die Anzahl der Kanten begrenzt, die zu dem Knoten v hinführen. Da ein Aufruf von $q.change()$ für eine Prioritäts-Warteschlange q mit n Elementen Rechenzeit in der Höhe von $\mathcal{O}(\log_2(n))$ erfordert, haben wir also insgesamt für den Aufruf von $change()$ die Abschätzung

$$\mathcal{O}(\#E \cdot \log_2(\#V))$$

Dabei bezeichnet $\#E$ die Anzahl der Kanten. Damit erhalten wir für die Komplexität von Dijkstra's Algorithmus den Ausdruck

$$\mathcal{O}((\#E + \#V) * \ln(\#V)).$$

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

$$\mathcal{O}(\#V * \log_2(\#V))$$

gegeben.

Kapitel 11

Die Monte-Carlo-Methode

Bestimmte Probleme sind so komplex, dass es mit vertretbarem Aufwand nicht möglich ist, eine exakte Lösung zu berechnen. Oft lässt sich jedoch mit Hilfe einer Simulation das Problem zumindest näherungsweise lösen.

1. Das Problem der Berechnung der Volumina von Körpern, die eine große Zahl von Begrenzungsflächen haben, lässt sich auf die Berechnung mehrdimensionaler Integrale zurückführen. In der Regel können diese Integrationen aber nicht analytisch ausgeführt werden. Mit der Monte-Carlo-Methode lässt sich hier zumindest ein Näherungswert bestimmen.
2. Die Gesetzmäßigkeiten des Verhaltens komplexer Systeme, die zufälligen Einflüssen einer Umgebung ausgesetzt sind, können oft nur durch Simulationen bestimmt werden. Wird beispielsweise ein neues U-Bahn-System geplant, so wird Kapazität eines projektierten Systems durch Simulationen ermittelt.
3. Bei Glücksspielen ist die exakte Berechnung bestimmter Wahrscheinlichkeiten oft nicht möglich. Mit Hilfe von Simulationen lassen sich aber gute Näherungswerte bestimmen.

Die obige Liste könnte leicht fortgesetzt werden. In diesem Kapitel werden wir zwei Beispiele betrachten.

1. Als einführendes Beispiel zeigen wir, wie sich mit Hilfe der Monte-Carlo-Methode Flächeninhalte bestimmen lassen. Konkret berechnen wir den Flächeninhalt eines Kreises und bestimmen auf diese Weise die Zahl π .
2. Als zweites Beispiel zeigen wir, wie sich Karten zufällig mischen lassen. Damit kann beispielsweise die Wahrscheinlichkeit dafür berechnet werden, dass im Texas Hold'em Poker eine gegebene Hand gegen eine zufällige Hand gewinnt.

11.1 Berechnung der Kreiszahl π

Eine sehr einfache Methode zur Berechnung einer Approximation der Zahl π funktioniert wie folgt. Wir betrachten in der reellen Ebene den Einheits-Kreis E , der als die Menge

$$E = \{\langle x, y \rangle \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1\}$$

definiert ist. Hier gibt der Ausdruck $x^2 + y^2$ nach dem Satz des Pythagoras gerade den Abstand an, den der Punkt $\langle x, y \rangle$ vom Koordinatenursprung $\langle 0, 0 \rangle$ hat. Der Einheits-Kreis hat offenbar den Radius $r = 1$ damit gilt für die Fläche dieses Kreises

$$\text{Fläche}(E) = \pi \cdot r^2 = \pi.$$

Wenn es uns gelingt, diese Fläche zu berechnen, dann haben wir also π bestimmt. Eine experimentelle Methode zur Bestimmung dieser Fläche besteht darin, dass wir in das Einheits-Quadrat Q , dass durch

$$Q = \{(x, y) \in \mathbb{R} \mid -1 \leq x \leq 1 \wedge -1 \leq y \leq 1\}$$

definiert ist, zufällig eine große Zahl n von Sandkörnern werfen. Wir notieren uns dabei die Zahl k der Sandkörner, die in den Einheits-Kreis fallen. Die Wahrscheinlichkeit p dafür, dass ein Sandkorn in den Einheits-Kreis fällt, wird nun proportional zur Fläche des Einheits-Kreises sein:

$$p = \frac{\text{Fläche}(E)}{\text{Fläche}(Q)}.$$

Das Einheits-Quadrat die Seitenlänge 2 hat, gilt für die Fläche des Einheits-Quadrats Q

$$\text{Fläche}(Q) = 2^2 = 4.$$

Auf der anderen Seite wird bei einer hohen Anzahl von Sandkörnern das Verhältnis $\frac{k}{n}$ gegen diese Wahrscheinlichkeit p streben, so dass wir insgesamt

$$\frac{k}{n} \approx \frac{\pi}{4}$$

haben, woraus sich für π die Näherungsformel

$$\pi \approx 4 \cdot \frac{k}{n}$$

ergibt. Während die alten Ägypter bei dieser historischen Methode zur Berechnung von π noch Tonnen von Sand¹ benötigten, können wir dieses Experiment heute einfacher mit Hilfe eines Computers durchführen.

Abbildung 11.1 zeigt die Implementierung der Klasse `CalculatePi`, die das Verfahren implementiert.

1. Die Klasse hat eine Member-Variable `mRandom`. Hierbei handelt es sich um einen Zufallszahlen-Generator, der von der Klasse `java.util.Random` zur Verfügung gestellt wird.
2. Die Methode `calcPi(n)` führt n Versuche zur Berechnung der Zahl π durch. Dazu werden mit Hilfe der Methode `nextDouble()` zunächst Zufallszahlen erzeugt, die in dem Intervall $[0, 1]$ liegen. Mit Hilfe der Transformation

$$t \mapsto 2 \cdot t - 1$$

wird das Intervall $[0, 1]$ in das Intervall $[-1, 1]$ transformiert, so dass die in den Zeilen 12 und 13 berechneten Koordinaten `x` und `y` dann einen zufällig in das Einheits-Quadrat Q geworfenes Sandkorn beschreiben. Wir berechnen in Zeile 14 den Abstand dieses Sandkorns vom Koordinatenursprung und überprüfen in Zeile 15, ob das Sandkorn innerhalb des Kreises liegt.

Lassen wir das Programm laufen, so erhalten wir die in Tabelle 11.1 gezeigten Ergebnisse. Wir sehen, dass wir zur Berechnung auf eine Genauigkeit von zwei Stellen hinter dem Komma etwa 100 000 Versuche brauchen, was angesichts der Rechenleistung heutiger Computer kein Problem darstellt. Die Berechnung weiterer Stellen gestaltet sich jedoch sehr aufwendig: Die Berechnung der dritten Stelle hinter dem Komma erfordert 100 000 000 Versuche. Grob geschätzt können wir sagen, dass sich der Aufwand bei der Berechnung einer weiteren Stelle ver Hundertfacht! Wir halten folgende Beobachtung fest:

Die Monte-Carlo-Methode ist gut geeignet, um einfache Abschätzungen zu berechnen, wird aber sehr aufwendig, wenn eine hohe Genauigkeit gefordert ist.

¹So ist die Sahara entstanden.

```

1  import java.util.*;
2
3  public class CalculatePi {
4      private Random mRandom;
5
6      public CalculatePi() {
7          mRandom = new Random();
8      }
9      public double calcPi(long n) {
10         long k = 0; // number of points inside the circle
11         for (long i = 0; i < n; ++i) {
12             double x = 2 * mRandom.nextDouble() - 1;
13             double y = 2 * mRandom.nextDouble() - 1;
14             double r = x * x + y * y;
15             if (r <= 1) { ++k; }
16         }
17         return 4.0 * k / n;
18     }
19     public static void main(String[] args) {
20         CalculatePi c = new CalculatePi();
21         for (long n = 10; n < 1000000000000L; n = n * 10) {
22             double pi = c.calcPi(n);
23             System.out.printf("n = %14d: %g, Fehler: %f\n",
24                               n, pi, pi - Math.PI);
25         }
26     }
27 }

```

Abbildung 11.1: Experimentelle Bestimmung von π mit der Monte-Carlo-Methode.

11.2 Theoretischer Hintergrund

Wir diskutieren nun den theoretischen Hintergrund der Monte-Carlo-Methode. Da im zweiten Semester noch keine detaillierteren Kenntnisse aus der Wahrscheinlichkeitsrechnung vorhanden sind, beschränken wir uns darauf, die wesentlichen Ergebnisse anzugeben. Eine Begründung dieser Ergebnisse erfolgt dann in der Mathematik-Vorlesung im vierten Semester.

Bei der Monte-Carlo-Methode wird ein Zufalls-Experiment, im gerade diskutierten Beispiel war es das Werfen eines Sandkorns, sehr oft wiederholt. Für den Ausgang dieses Zufalls-Experiments gibt es dabei zwei Möglichkeiten: Es ist entweder erfolgreich (im obigen Beispiel landet das Sandkorn im Kreis) oder nicht erfolgreich. Ein solches Experiment bezeichnen wir als *Bernoulli-Experiment*. Hat die Wahrscheinlichkeit, dass das Experiment erfolgreich ist, den Wert p und wird das Experiment n mal ausgeführt, so ist die Wahrscheinlichkeit, dass genau k dieser Versuche erfolgreich sind, durch die Formel

$$P(k) = \frac{n!}{k! \cdot (n-k)!} \cdot p^k \cdot (1-p)^{n-k}$$

gegeben, die auch als *Binomial-Verteilung* bekannt ist. Für große Werte von n ist die obige Formel sehr unhandlich, kann aber gut durch die *Gauß-Verteilung* approximiert werden, es gilt

$$\frac{n!}{k! \cdot (n-k)!} \cdot p^k \cdot (1-p)^{n-k} \approx \frac{1}{\sqrt{2 \cdot \pi \cdot n \cdot p \cdot (1-p)}} \cdot \exp\left(-\frac{(k - n \cdot p)^2}{2 \cdot n \cdot p \cdot (1-p)}\right)$$

Wird das Experiment n mal durchgeführt, so erwarten wir im Durchschnitt natürlich, dass $n \cdot p$

| n | Näherung für π | Fehler der Näherung |
|---------------|--------------------|---------------------|
| 10 | 2.40000 | -0.741593 |
| 100 | 3.28000 | +0.138407 |
| 1 000 | 3.21600 | +0.074407 |
| 10 000 | 3.13080 | -0.010793 |
| 100 000 | 3.13832 | -0.003273 |
| 1 000 000 | 3.13933 | -0.002261 |
| 10 000 000 | 3.14095 | -0.000645 |
| 100 000 000 | 3.14155 | -0.000042 |
| 1 000 000 000 | 3.14160 | +0.000011 |

Tabelle 11.1: Ergebnisse bei der Bestimmung von π mit der Monte-Carlo-Methode

der Versuche erfolgreich sein werden. Darauf basiert unsere Schätzung für den Wert von p , denn wir approximieren p durch die Formel

$$p \approx \frac{k}{n},$$

wobei k die Anzahl der erfolgreichen Experimente bezeichnet. Nun werden in der Regel nicht genau $n \cdot p$ Versuche erfolgreich sein sondern zufallsbedingt werden ein Paar mehr oder ein Paar weniger Versuche erfolgreich sein. Das führt dazu, dass unsere Schätzung von p eine Ungenauigkeit aufweist, deren ungefähre Größe wir irgendwie abschätzen müssen um unsere Ergebnisse beurteilen zu können.

Um eine Idee davon zu bekommen, wie sehr die Anzahl der erfolgreichen Versuche von dem Wert $\frac{k}{n}$ abweicht, führen wir den Begriff der *Streuung* σ ein, die für eine Gauß-verteilte Zufallsgröße durch die Formel

$$\sigma = \sqrt{n \cdot p \cdot (1 - p)}$$

gegeben ist. Die Streuung gibt ein Maß dafür, wie stark der gemessene Wert von k von dem im Mittel erwarteten Wert $p \cdot n$ abweicht. Es kann gezeigt werden, dass die Wahrscheinlichkeit, dass k außerhalb des Intervalls

$$[p \cdot n - 3 \cdot \sigma, p \cdot n + 3 \cdot \sigma]$$

liegt, also um mehr als das dreifache von dem erwarteten Wert abweicht, kleiner als 0.27% ist. Für die Genauigkeit unserer Schätzung $p \approx \frac{k}{n}$ heißt das, dass dieser Schätzwert mit hoher Wahrscheinlichkeit (99.73%) in dem Intervall

$$\left[\frac{p \cdot n - 3 \cdot \sigma}{n}, \frac{p \cdot n + 3 \cdot \sigma}{n} \right] = \left[p - 3 \cdot \frac{\sigma}{n}, p + 3 \cdot \frac{\sigma}{n} \right]$$

liegt. Die Genauigkeit $\varepsilon(n)$ ist durch die halbe Länge dieses Intervalls gegeben und hat daher den Wert

$$\varepsilon(n) = 3 \cdot \frac{\sigma}{n} = 3 \cdot \sqrt{\frac{p \cdot (1 - p)}{n}}.$$

Wir erkennen hier, dass zur Erhöhung der Genauigkeit den Faktor 10 die Zahl der Versuche um den Faktor 100 vergrößert werden muss.

Wenden wir die obige Formel auf die im letzten Abschnitt durchgeführte Berechnung der Zahl π an, so erhalten wir wegen $p = \frac{\pi}{4}$ die in Abbildung 11.2 gezeigten Ergebnisse.

Aufgabe 9: Wieviel Tonnen Sand benötigte Pharao Ramses II, als er π mit der Monte-Carlo-Methode auf 6 Stellen hinter dem Komma berechnet hat?

| Anzahl Versuche n | Genauigkeit $\varepsilon(n)$ |
|---------------------|------------------------------|
| 10 | 0.389478 |
| 100 | 0.123164 |
| 1 000 | 0.0389478 |
| 10 000 | 0.0123164 |
| 100 000 | 0.00389478 |
| 1 000 000 | 0.00123164 |
| 10 000 000 | 0.000389478 |
| 100 000 000 | 0.000123164 |
| 1 000 000 000 | 3.89478e-05 |
| 10 000 000 000 | 1.23164e-05 |
| 100 000 000 000 | 3.89478e-06 |

Tabelle 11.2: Genauigkeit der Bestimmung von π bei einer Sicherheit von 99,73%.

Hinweise:

1. Ein Sandkorn wiegt im Durchschnitt etwa $\frac{1}{1000}$ Gramm.
2. Um eine Genauigkeit von 6 Stellen hinter dem Komma zu haben, sollte der Fehler durch 10^{-7} abgeschätzt werden. Das Ergebnis soll mit einer Wahrscheinlichkeit von 99,7% korrekt sein.

Lösung: Nach dem Hinweis soll

$$\varepsilon(n) = 10^{-7}$$

gelten. Setzen wir hier die Formel für $\varepsilon(n)$ ein, so erhalten wir

$$\begin{aligned} 3 \cdot \sqrt{\frac{p \cdot (1-p)}{n}} &= 10^{-7} \\ \Leftrightarrow 9 \cdot \frac{p \cdot (1-p)}{n} &= 10^{-14} \\ \Leftrightarrow 9 \cdot p \cdot (1-p) \cdot 10^{14} &= n \end{aligned}$$

Um an dieser Stelle weitermachen zu können, benötigen wir den Wert der Wahrscheinlichkeit p . Der korrekte Wert von p ist für unser Experiment durch $\frac{\pi}{4}$ gegeben. Da wir π ja erst berechnen wollen, nehmen wir als Abschätzung von π den Wert 3, so dass p den Wert $\frac{3}{4}$ hat. Da eine Tonne insgesamt 10^9 Sandkörner enthält, bekommen wir für das Gewicht g das Ergebnis

$$\begin{aligned} g &\approx 9 \cdot \frac{3}{4} \cdot \frac{1}{4} \cdot 10^5 \text{ Tonnen} \\ &\approx 168\,750 \text{ Tonnen} \end{aligned}$$

Die Cheops-Pyramide ist 35 mal so schwer. □

11.3 Erzeugung zufälliger Permutationen

In diesem Abschnitt lernen wir ein Verfahren kennen, mit dem es möglich ist, eine gegebene Liste zufällig zu permutieren. Anschaulich kann ein solches Verfahren mit dem Mischen von Karten verglichen werden. Das Verfahren wird auch tatsächlich genau dazu eingesetzt: Bei der Berechnung von Gewinn-Wahrscheinlichkeiten bei Kartenspielen wie Poker wird das Mischen der Karten durch den gleich vorgestellten Algorithmus erledigt.

Um eine n -elementige Liste $L = [x_1, x_2, \dots, x_n]$ zufällig zu permutieren, unterscheiden wir zwei Fälle:

1. Die Liste L hat die Länge 1 und besteht folglich nur aus einem Element, $L = [x]$. In diesem Fall gibt die Funktion $permute(L)$ die Liste unverändert zurück:

$$\#L = 1 \rightarrow permute(L) = L$$

2. Die Liste L hat eine Länge, die größer als 1 ist. In diesem Fall wählen wir zufällig ein Element aus, das hinterher in der zu erzeugenden Permutation an der letzten Stelle stehen soll. Wir entfernen dieses Element aus der Liste und permutieren anschließend die verbleibende Liste. An die dabei erhaltene Permutation hängen wir noch das anfangs ausgewählte Element an. Haben wir eine Funktion

$$random : \mathbb{N} \rightarrow \mathbb{N},$$

so dass der Aufruf $random(n)$ zufällig eine Zahl aus der Menge $\{1, \dots, n\}$ liefert, so können wir diese Überlegung wie folgt formalisieren:

$$\#L = n \wedge n > 1 \wedge random(n) = k \rightarrow permute(L) = permute(delete(L, k)) + [L(k)].$$

Der Funktionsaufruf $delete(L, k)$ köscht dabei das k -te Element aus der Liste L , wir könnten also schreiben

$$delete(L, k) = L(1 .. k - 1) + L(k + 1 .. \#L).$$

```

1  public static void permute(int[] array) {
2      Random random = new Random(0);
3      int n = array.length;
4      for (int i = n; i > 1; --i) {
5          int k = random.nextInt(i);
6          swap(array, i - 1, k);
7      }
8  }
9  private static void swap(int[] array, int i, int j) {
10     int t = array[i];
11     array[i] = array[j];
12     array[j] = t;
13 }

```

Abbildung 11.2: Berechnung zufälliger Permutationen eines Feldes

Abbildung 11.2 zeigt die Umsetzung dieser Idee in *Java*. Die dort gezeigte Methode *permute* erzeugt eine zufällige Permutation des Feldes *array*, das dieser Methode als Argument übergeben wird.

1. Zunächst wird in Zeile 2 ein Zufallszahlen-Generator erzeugt.
2. Die Schleife in Zeile 4 läuft nun rückwärts über das Feld, denn wir wollen ja zunächst das letzte Element des Feldes bestimmen. Beim i -ten Durchlauf gehen wir davon aus, dass die Positionen $array[i, \dots, n - 1]$ bereits bestimmt sind und dass nur noch die Positionen $array[0, \dots, i - 1]$ permutiert werden müssen.
3. Der Aufruf `random.nextInt(i)` liefert eine Zufallszahl k aus der Menge $\{0, \dots, i - 1\}$. Diese Zahl gibt an, welches Element an die $(i - 1)$ -te Position der erzeugten Permutation gesetzt wird. Dieses Element wird dann durch den Aufruf von `swap` an die Position $i - 1$ gesetzt und das Element, das vorher an der Position $i - 1$ stand, wird an der Position k gespeichert.

Es kann gezeigt werden, dass der oben vorgestellte Algorithmus tatsächlich alle Permutationen einer gegebenen Liste mit derselben Wahrscheinlichkeit erzeugt. Einen Beweis dieser Behauptung finden Sie beispielsweise in [CLRS01].

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.
- [BS03] Egon Börger and Robert Stärk. *Abstract State Machines*. Springer-Verlag, 2003.
- [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.
- [Gur91] Yuri Gurevich. Evolving algebras. *Bull. EATCS*, 43:264–284, 1991.
- [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.