Algorithmen und Daten-Strukturen

Karl Stroetmann

26. Juli 2006

Inhaltsverzeichnis

Kapitel 1

Einführung

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 einzelnen 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 `C`-Programm gibt, dass für eine gegebene `C`-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 *O-Schreibweise* wird verwendet, um bei der Berechnung des Rechenzeit-Verbrauchs von unwichtigen Details abstrahieren zu können.

3. Abstrakte Daten-Typen und elementare Daten-Strukturen

   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.

4. Sortier-Algorithmen

   Sortier-Algorithmen sind die in der Praxis am häufigsten verwendeten Algorithmen. Wir behandeln folgende Algorithmen:

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

   (b) Sortieren durch Minimums-Bildung (engl. *min sort*)

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

   (d) *Quick-Sort*

5. 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 wichtigsten Daten-Strukturen, mit denen sich Abbildungen realisieren lassen.

6. Graphen

   Graphen spielen insbesondere in der künstlichen Intelligenz (KI) eine wichtige Rolle. Nachdem wir im ersten Semester einen naiven Algorithmus zur Suche in Graphen kennengelernt haben, betrachten wir nun den Algorithmus von Dijkstra zur Berechnung eines kürzesten Weges zwischen zwei Knoten.

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

1.2 Algorithmen und Programme

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

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

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

Nachdem wir uns den Unterschied zwischen einem Algorithmus und einem Programm klar gemacht 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 SETL2 an, denn diese Programmier-Sprache stellt die Daten-Strukturen Mengen und Funktionen, die in der Mathematik allgegenwärtig sind, zur Verfügung. Natürlich ist es auch wichtig zu wissen, wie Algorithmen in einer klassischen Programmier-Sprache wie beispielsweise `C` oder *Java* implementiert werden können. Daher werden es Ihre Aufgabe sein, einige der vorgestellten Algorithmen in *Java* zu implementieren.

1.3 Eigenschaften von Algorithmen und Programmen

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

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

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

1. Für einen Algorithmus, dessen konzeptionelle Komplexität hoch ist, ist die Korrektheit nicht mehr einsehbar und damit auch nicht gewährleistbar.
2. Selbst wenn der Algorithmus an sich korrekt ist, so kann doch die Korrektheit der Implementierung nicht mehr sichergestellt werden.

1.4 Literatur

Das vorliegende Skript hat weder die Ausführlichkeit noch die Qualität eines Lehrbuches. Es wäre kaum ökonomisch, wenn jeder Dozent, der eine Vorlesung über dieses Thema hält, ein solches Lehrbuch schreibt. Glücklicherweise ist dies auch gar nicht notwendig, denn es gibt eine Reihe sehr guter Bücher zu den in der Vorlesung behandelten Themen.

1. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *Data Structures and Algorithms*, Addison-Wesley, 1987.

   Dieses Buch ist die Neuauflage des 1974 erschienen Titels.
3. *Frank M. Carrano*: *Data Abstraction and Problem Solving with* `C++`, Benjamin/Cummings Publishing Company, 1995.

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

   Eine Neuauflage des obigen Werkes, bei der die Algorithmen jetzt in *Java* statt `C++` beschrieben werden.

5. *Thomas H. Cormen* and *Charles E. Leiserson*: *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.

6. *Robert Sedgewick*: *Algorithms in* `C`, Addison-Wesley, 1992.

   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.

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

   Eine Neuauflage des obigen Werkes, bei der die Algorithmen in *Java* statt in `C` beschrieben werden.

Von den meisten der oben aufgeführten Bücher existieren Übersetzungen ins Deutsche. Ich rate dringend von dem Gebrauch dieser Übersetzungen ab, da die Qualität weit hinter den Originalen zurück bleibt.

Kapitel 2

Grenzen der Berechenbarkeit

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 ein String der Form

$$\texttt{int } n\texttt{(char* x) } \{ \cdots \}$$

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

Beispiele:

1. s_1 = "`int simple(char* x) { return 0; }`"
   s_1 ist eine (sehr einfache) Test-Funktion mit dem Namen `simple`.
2. s_2 = "`int loop(char* x) { while (1) ++x; }`"
   s_2 ist eine Test-Funktion mit dem Namen `loop`.
3. s_3 = "`int loop(char* x);`"
   s_3 ist keine Test-Funktion, denn es ist lediglich eine Deklaration einer `C`-Funktion und keine Definition.
4. s_4 = "`int hugo(char* x) begin i := 1; end;`"
   s_4 ist keine Test-Funktion, denn es läßt sich mit einem `C`-Compiler nicht fehlerfrei parsen.
5. s_5 = "`int hugo(int x) { return i*i; }`"
   s_5 ist auch keine Test-Funktion, denn der Typ des Arguments ist `int` und nicht `char*`.

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

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

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

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

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

Terminiert der Aufruf $n(a_1, \cdots, a_k)$ nicht, so schreiben wir

$$n(a_1, \cdots, a_k) \uparrow$$ □

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 `C` ist die Frage, ob es eine `C`-Funktion

`int stops(char*` t, `char*` a)

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

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

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

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

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

   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 `C`-Funktion `stops` mit den obigen Eigenschaften existiert, dann sagen wir, dass das Halte-Problem für `C` 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 dann 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 "`turing`" ist:

$$\textit{Turing} \in TF \wedge \texttt{name}(\textit{Turing}) = \texttt{turing}.$$

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

`stops(`*Turing*`,` *Turing*`)`.

Da *Turing* eine Test-Funktion ist, können nur zwei Fälle auftreten:

$$\texttt{stops}(\textit{Turing}, \textit{Turing}) \rightsquigarrow 0 \quad \vee \quad \texttt{stops}(\textit{Turing}, \textit{Turing}) \rightsquigarrow 1.$$

Diese beiden Fälle analysieren wir nun im Detail:

```
1   Turing := "int turing(char* x) {
2                   int result;
3                   result = stops(x, x);
4                   if (result == 1) {
5                       while (1) {
6                           ++result;
7                       }
8                   }
9                   return result;
10              }"
```

Abbildung 2.1: Die Definition des Strings *Turing*.

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

   Nach der Spezifikation von `stops` bedeutet dies

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

   Schauen wir nun, was wirklich beim Aufruf `turing(Turing)` passiert: In Zeile 3 erhält die Variable `result` den Wert 0 zugewiesen. In Zeile 4 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 7 den Wert 0 zurück. Insbesondere terminiert der Aufruf also, im Widerspruch zu dem, was die Funktion `stops` behauptet hat.

   Damit ist der erste Fall ausgeschlossen.

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

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

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

   Schauen wir nun, was wirklich beim Aufruf `turing(Turing)` passiert: In Zeile 3 erhält die Variable `result` den Wert 1 zugewiesen. In Zeile 4 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.

3. `stops`(*Turing*, *Turing*) $\rightsquigarrow 2$.

   Dann müßte aber der Programmtext *Turing* einen Syntax-Fehler enthalten, was nicht der Fall ist.

   Damit ist auch der dritte und letzte Fall ausgeschlossen.

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

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

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

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

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

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

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

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

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

Definieren Sie eine Menge `Cantor` wie folgt:

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

Versuchen Sie nun, einen Widerspruch herzuleiten.

2.2 Andere nicht berechenbare Funktionen

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

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

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

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

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

Wir kommen jetzt zum *Äquivalenz-Problem.* Die Funktion

```
int equal(char* p1, char* p2, char* a)
```

möge folgender Spezifikation genügen:

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

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

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

Machen wir uns also an die Arbeit und zeigen, dass es keine Funktion geben kann, die das Äquivalenz-Problem löst. Wie oben skizziert führen wir den Beweis indirekt und nehmen an, dass

```
1   int stops(char* p, char* a) {
2       int e = equal("int loop(char* x) { while (1) {++x;} }", p, a);
3       if (e == 2) {
4           return 2;
5       } else {
6           return 1 - e;
7       }
8   }
```

Abbildung 2.2: Eine Implementierung der Funktion `stops`.

es doch eine Implementierung der Funktion `equal` gibt, die das Äquivalenz-Problem löst. Wir betrachten die in Abbildung 2.2 angegeben Implementierung der Funktion `stops`.

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 hat die folgende Form:

```
int loop(char* x) { while (1) { ++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.

Kapitel 3

Komplexität von Algorithmen

In diesem Kapitel führen wir Rekurrenz-Gleichungen[1] ein und zeigen, wie diese in einfachen Fällen gelöst werden können. Außderdem stellen wir die O-Notation vor. Diese beiden Begriffe benötigen wir, um die Laufzeit von Algorithmen analysieren zu können. Die Algorithmen selber stehen in diesem Kapitel noch im Hintergrund.

3.1 Die Fibonacci-Zahlen

Wir wollen *Rekurrenz-Gleichungen* an Hand eines eher spielerischen Beispiels einführen. Dazu betrachten wir eine Kaninchen-Farm, für die wir einen Geschäftsplan erstellen wollen. Wir beschäftigen uns hier nur mit der Frage, wie sich eine Kaninchen-Population entwickelt. Wir gehen dabei von folgenden vereinfachenden Annahmen aus:

1. Jedes Kaninchen-Paar bringt jeden Monat ein neues Kaninchen-Paar zur Welt.
2. Kaninchen haben nach zwei Monaten Junge.
3. Kaninchen leben ewig.

Wir nehmen nun an, wir hätten ein neugeborenes Kaninchen-Paar und stellen uns die Frage, wie viele Kaninchen-Paare wir nach n Monaten haben. Bezeichnen wir die Zahl der Kaninchen-Paare nach n Monaten mit $k(n)$, so gilt:

1. $k(0) = 1$

   Wir starten mit einem neugeborenem Kaninchen-Paar.

2. $k(1) = 1$

   Kaninchen bekommen das erste Mal nach zwei Monaten Junge, also hat sich die Zahl der Kaninchen-Paare nach einem Monat noch nicht verändert.

3. $k(2) = 1 + 1$

   Nach zwei Monaten bekommt unser Kaninchen-Paar zum ersten Mal Junge.

4. Allgemein gilt nach $n + 2$ Monaten:

   $$k(n + 2) = k(n + 1) + k(n)$$

   Alle Kaninchen-Paare, die zum Zeitpunkt n schon da sind, bekommen zum Zeitpunkt $n + 2$ Junge. Dies erklärt den Term $k(n)$. Da wir zur Vereinfachung unserer Rechnung von genetisch manipulierten unsterblichen Kaninchen ausgehen, sind alle Kaninchen, die zum Zeitpunkt $n + 1$ vorhanden sind, auch noch zum Zeitpunkt $n + 2$ vorhanden. Dies erklärt den Term $k(n + 1)$.

[1] Rekurrenz-Gleichungen werden in der Literatur auch als *Rekursions-Gleichungen* bezeichnet.

Die Folge der Zahlen $\big(k(n)\big)_{n\in\mathbb{N}}$ heißt Folge der *Fibonacci-Zahlen*. Das *Java*-Programm in Abbildung 3.1 auf Seite 13 berechnet diese Zahlen.

```
1  import java.util.*;
2
3  public class Fibonacci
4  {
5      public static void main(String[] args) {
6          for (int i = 0; i < 100; ++i) {
7              int n = fibonacci(i);
8              System.out.printf("fibonacci(%d) = %d\n", i, n);
9          }
10     }
11
12     public static int fibonacci(int n) {
13         if (n == 0) return 1;
14         if (n == 1) return 1;
15         return fibonacci(n - 1) + fibonacci(n - 2);
16     }
17 }
```

Abbildung 3.1: Ein *Java*-Programm zur Berechnung der Fibonacci-Zahlen.

Wenn wir dieses Programm laufen lassen, stellen wir fest, dass die Laufzeiten mit wachsendem Parameter n sehr schnell anwachsen. Um dieses Phänomen zu analysieren, untersuchen wir exemplarisch, wie viele Additionen bei der Berechnung von `fibonacci`(n) für ein gegebenes $n \in \mathbb{N}$ benötigt werden. Bezeichnen wir diese Zahl mit a_n, so finden wir:

1. $a_0 = 0$.
2. $a_1 = 0$.
3. $a_n = a_{n-1} + a_{n-2} + 1$,

   denn in den rekursiven Aufrufen `fibonacci`$(n-1)$ und `fibonacci`$(n-2)$ haben wir a_{n-1} bzw. a_{n-2} Additionen und dazu kommt noch die Addition der Werte `fibonacci`$(n-1)$ und `fibonacci`$(n-2)$.

Wir setzen in der Gleichung $a_n = 1 + a_{n-1} + a_{n-2}$ für n den Wert $i+2$ ein und haben dann

$$a_{i+2} = a_{i+1} + a_i + 1 \tag{1}$$

Eine solche Gleichung nennen wir eine *lineare inhomogene Rekurrenz-Gleichung*. Die dieser Gleichung zugeordnete *homogene Rekurrenz-Gleichung* lautet

$$a_{i+2} = a_{i+1} + a_i \tag{2}$$

Wir lösen diese Gleichung mit folgendem Ansatz:

$$a_i = \lambda^i.$$

Einsetzen dieses Ansatzes in $(*)$ führt auf die Gleichung

$$\lambda^{i+2} = \lambda^{i+1} + \lambda^i$$

Wenn wir beide Seiten dieser Gleichung durch λ^i dividieren, erhalten wir die quadratische Gleichung

$$\lambda^2 = \lambda + 1,$$

die wir wie folgt umformen:

$$
\begin{array}{lcll}
\lambda^2 & = & \lambda + 1 & \mid -\lambda \\
\lambda^2 - 2 * \frac{1}{2}\lambda & = & 1 & \mid +\frac{1}{4} \\
\lambda^2 - 2 * \frac{1}{2}\lambda + \left(\frac{1}{2}\right)^2 & = & \frac{5}{4} & \\
\left(\lambda - \frac{1}{2}\right)^2 & = & \frac{5}{4} & \mid \sqrt{\ } \\
\lambda_{1/2} & = & \frac{1}{2}(1 \pm \sqrt{5}) &
\end{array}
$$

Wir bemerken, dass für die Lösungen λ_1 und λ_2 folgende Identitäten gelten:

$$\lambda_1 - \lambda_2 = \sqrt{5} \quad \text{und} \quad \lambda_1 + \lambda_2 = 1. \tag{1}$$

Aus der letzen Gleichung folgt dann sofort

$$1 - \lambda_1 = \lambda_2 \quad \text{und} \quad 1 - \lambda_2 = \lambda_1 \tag{2}$$

Um nun die ursprüngliche Rekurrenz-Gleichung (1) zu lösen, machen wir den Ansatz $a_i = c$. Das führt auf die Gleichung

$$c = c + c + 1,$$

die die Lösung $c = -1$ hat. Die allgemeine Lösung der Rekurrenz-Gleichung (1) lautet damit

$$a_i = \alpha * \lambda_1^i + \beta * \lambda_2^i - 1$$

mit $\lambda_1 = \frac{1}{2}(1 + \sqrt{5})$ und $\lambda_2 = \frac{1}{2}(1 - \sqrt{5})$. Die Koeffizienten α und β sind jetzt so zu bestimmen, dass die Anfangs-Bedingungen $a_0 = 0$ und $a_1 = 0$ erfüllt sind. Das führt auf folgendes lineares Gleichungs-System:

$$
\begin{array}{rcl}
0 & = & \alpha * \lambda_1^0 + \beta * \lambda_2^0 - 1 \\
0 & = & \alpha * \lambda_1^1 + \beta * \lambda_2^1 - 1
\end{array}
$$

Addieren wir bei beiden Gleichungen 1 und vereinfachen für $i = 1, 2$ die Potenzen λ_i^0 zu 1 und λ_i^1 zu λ_i, so erhalten wir:

$$
\begin{array}{rcl}
1 & = & \alpha + \beta \\
1 & = & \alpha * \lambda_1 + \beta * \lambda_2
\end{array}
$$

Die erste dieser beiden Gleichungen liefert die Beziehung $\alpha = 1 - \beta$. Setzen wir dies für α in der zweiten Gleichung ein, so erhalten wir

$$
\begin{array}{rcll}
 & 1 & = & (1 - \beta)\lambda_1 + \beta * \lambda_2 \\
\Leftrightarrow & 1 & = & \lambda_1 + \beta * \left(\lambda_2 - \lambda_1\right) \\
\Leftrightarrow & 1 - \lambda_1 & = & \beta * \left(\lambda_2 - \lambda_1\right) \\
\Leftrightarrow & \frac{1 - \lambda_1}{\lambda_2 - \lambda_1} & = & \beta
\end{array}
$$

Wegen $\alpha = 1 - \beta$ finden wir dann

$$\alpha = -\frac{1 - \lambda_2}{\lambda_2 - \lambda_1}.$$

Verwenden wir hier die Gleichungen (1) und (2), so finden wir

$$\alpha = \frac{\lambda_1}{\sqrt{5}} \quad \text{und} \quad \beta = -\frac{\lambda_2}{\sqrt{5}}.$$

Damit können wir die Folge $(a_i)_i$ explizit angeben:

$$a_i = \frac{1}{\sqrt{5}} * \left(\lambda_1^{i+1} - \lambda_2^{i+1}\right) - 1$$

Wegen $\lambda_1 \approx 1.61803$ und $\lambda_2 \approx -0.61803$ dominiert der erste Term der Summe und die Zahl der Additionen wächst exponentiell mit dem Faktor λ_1 an. Dies erklärt das starke Anwachsen der Rechenzeit.

Die Ursache der Ineffezienz ist leicht zu sehen: Berechnen wir beispielsweise den Wert `fibonacci(5)` mit dem Programm aus Abbildung 3.1, so müssen wir `fibonacci(4)` und `fibonacci(3)` berechnen. Die Berechnung von `fibonacci(4)` erfordert ihrerseits die Berechnung von `fibonacci(3)`

und `fibonacci(2)`. Dann berechnen wir `fibonacci(3)` aber zweimal! Wir können das Problem lösen, indem wir uns die berechneten Werte merken. Dazu können wir in *Java* ein Feld benutzen. Dies führt zu dem in Abbildung 3.2 auf Seite 15 angegebenen Programm. Da die Werte der Funktion `fibonacci`() exponentiell wachsen, reichen 32-Bit-Zahlen nicht aus, um diese Werte darzustellen. Wir verwenden daher die Klasse `BigInteger`, mit der sich ganze Zahlen beliebiger Größe darstellen lassen. Da Felder in *Java* genau wie in `C` mit 0 beginnend indiziert werden, hat ein Feld, dessen oberster Index n ist, insgesamt $n + 1$ Elemente. Wir legen daher in Zeile 19 ein Feld von $n + 1$ Elementen an.

```
1  import java.util.*;
2  import java.math.*;
3
4  public class FibonacciBig
5  {
6      public static void main(String[] args)
7      {
8          for (int i = 0; i < 100; ++i) {
9              BigInteger n = fibonacci(i);
10             System.out.println("fib(" + i + ") = " + n);
11         }
12     }
13
14     public static BigInteger fibonacci(int n)
15     {
16         if (n <= 2) {
17             return BigInteger.valueOf(1);
18         }
19         BigInteger[] mem = new BigInteger[n+1];
20         mem[0] = BigInteger.valueOf(1);  // fibonacci(0) = 1
21         mem[1] = BigInteger.valueOf(1);  // fibonacci(1) = 1
22         for (int i = 0; i < n - 1; ++i) {
23             mem[i + 2] = mem[i].add(mem[i + 1]);
24         }
25         return mem[n];
26     }
27 }
```

Abbildung 3.2: Berechnung der Fibonacci-Zahlen mit Speicherung der Zwischenwerte.

3.2 Lineare Rekurrenz-Gleichung

Wir waren bei der Analyse der Komplexität des ersten Programms zur Berechnung der Fibonacci-Zahlen auf die Gleichung

$$a_{i+2} = a_{i+1} + a_i + 1 \quad \text{für alle } n \in \mathbb{N}$$

gestoßen. Gleichungen dieser Form treten bei der Analyse der Komplexität rekursiver Programme häufig auf. Wir wollen uns daher in diesem Abschnitt näher mit solchen Gleichungen beschäftigen.

Definition 5 (Lineare homogene Rekurrenz-Gleichung) Die *lineare homogene Rekurrenz-Gleichung der Ordnung k mit konstanten Koeffizienten* hat die Form

$$a_{n+k} = c_{k-1} * a_{n+k-1} + c_{k-2} * a_{n+k-2} + \cdots + c_1 * a_{n+1} + c_0 * a_n \quad \text{für alle } n \in \mathbb{N} \quad (\star)$$

Zusätzlich werden *Anfangs-Bedingungen*

$$a_0 = d_0, \cdots, a_{k-1} = d_{k-1}$$

für die Folge $\big(a_n\big)_{n\in\mathbb{N}}$ vorgegeben. □

Durch eine lineare homogene Rekurrenz-Gleichung wird die Folge $(a_n)_{n\in\mathbb{N}}$ eindeutig bestimmt: Die Werte a_n für $n < k$ sind durch die Anfangs-Bedingungen gegeben und alle weiteren Werte können dann durch die Rekurrenz-Gleichung $(\star)$ bestimmt werden. Noch etwas zur Nomenklatur:

1. Die Rekurrenz-Gleichung $(\star)$ heißt *linear*, weil die Glieder der Folge $(a_n)_n$ nur linear in der Gleichung $(\star)$ auftreten. Ein Beispiel für eine Rekurrenz-Gleichung, die nicht linear ist, wäre

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

   Nicht-lineare Rekurrenz-Gleichungen sind nur in Spezialfällen geschlossen lösbar.

2. Die Rekurrenz-Gleichung $(\star)$ heißt *homogen*, weil auf der rechten Seite dieser Gleichung kein konstantes Glied mehr auftritt. Ein Beispiel für eine Gleichung, die nicht homogen ist (wir sprechen auch von *inhomogenen* Rekurrenz-Gleichungen), wäre

   $$a_{n+2} = a_{n+1} + a_n + 1 \quad \text{für alle } n \in \mathbb{N}.$$

   Mit inhomogenen Rekurrenz-Gleichungen werden wir uns später noch beschäftigen.

3. Die Rekurrenz-Gleichung $(\star)$ hat *konstante Koeffizienten*, weil die Werte c_i Konstanten sind, die nicht von dem Index i abhängen. Ein Beispiel für eine Rekurrenz-Gleichung, die keine konstanten Koeffizienten hat, ist

   $$a_{n+1} = n * a_n \quad \text{für alle } n \in \mathbb{N}.$$

   Solche Rekurrenz-Gleichungen können in vielen Fällen auf Rekurrenz-Gleichungen mit konstanten Koeffizienten zurück geführt werden. Wir werden das später noch im Detail besprechen.

Wie lösen wir eine lineare homogene Rekurrenz-Gleichung? Wir versuchen zunächst den Ansatz

$$a_n = \lambda^n \quad \text{für alle } n \in \mathbb{N}.$$

Einsetzen dieses Ansatzes in $(\star)$ führt auf die Gleichung

$$\lambda^{n+k} = c_{k-1} * \lambda^{n+k-1} + c_{k-2} * \lambda^{n+k-2} + \cdots + c_1 * \lambda^{n+1} + c_0 * \lambda^n$$

Dividieren wir diese Gleichung durch λ^n, so haben wir:

$$\lambda^k = c_{k-1} * \lambda^{k-1} + c_{k-2} * \lambda^{k-2} + \cdots + c_1 * \lambda^1 + c_0 * \lambda^0$$

Das Polynom

$$\chi(x) = x^k - c_{k-1} * x^{k-1} - c_{k-2} * x^{k-2} - \cdots - c_1 * x - c_0$$

heißt *charakteristisches Polynom* der Rekurrenz-Gleichung $(\star)$. Wir betrachten zunächst den Fall, dass das charakteristische Polynom k verschiedene Nullstellen hat. In diesem Fall sagen, dass die Rekurrenz-Gleichung $(\star)$ *nicht entartet* ist. Bezeichnen wir diese Nullstellen mit

$$\lambda_1, \lambda_2, \cdots, \lambda_k,$$

so gilt für alle $i = 1, \cdots, k$

$$\lambda_i^{n+k} = c_{k-1} * \lambda_i^{n+k-1} + c_{k-2} * \lambda_i^{n+k-2} + \cdots + c_1 * \lambda_i^{n+1} + c_0 * \lambda_i^n.$$

Damit ist jede Folge

$$\big(\lambda_i^n\big)_{n\in\mathbb{N}}$$

eine Lösung der Rekurrenz-Gleichung $(\star)$. Außerdem ist auch jede Linear-Kombination dieser Lösungen eine Lösung von $(\star)$: Definieren wir die Folge a_n durch

$$a_n = \alpha_1\lambda_1^n + \cdots + \alpha_k\lambda_k^n \quad \text{für alle } n \in \mathbb{N}$$

mit beliebigen Koeffizienten $\alpha_i \in \mathbb{R}$, so erfüllt auch die Folge $(a_n)_n$ die Gleichung $(\star)$. Die oben definierte Folge $(a_n)_n$ bezeichnen wir als die *allgemeine Lösung* der Rekurrenz-Gleichung $(\star)$:

Die Koeffizienten α_1 bis α_k müssen wir nun so wählen, dass die Anfangs-Bedingungen $a_0 = d_0$,

$\cdots$, $a_{k-1} = d_{k-1}$ erfüllt sind. Das liefert ein lineares Gleichungs-System für die Koeffizienten α_1, $\cdots$, α_k:

$$\begin{array}{lcl} d_0 & = & \lambda_1^0 * \alpha_1 + \cdots + \lambda_k^0 * \alpha_k \\ d_1 & = & \lambda_1^1 * \alpha_1 + \cdots + \lambda_k^1 * \alpha_k \\ \vdots & & \vdots \\ d_{k-1} & = & \lambda_1^{k-1} * \alpha_1 + \cdots + \lambda_k^{k-1} * \alpha_k \end{array}$$

Hier sind die Werte λ_i die Nullstellen des charakteristischen Polynoms. Die Matrix V, die diesem Gleichungs-System zugeordnet ist, lautet:

$$V = \begin{pmatrix} \lambda_1^0 & \cdots & \lambda_k^0 \\ \lambda_1^1 & \cdots & \lambda_k^1 \\ \vdots & & \vdots \\ \lambda_1^{k-1} & \cdots & \lambda_k^{k-1} \end{pmatrix}$$

Diese Matrix ist in der Mathematik als *Vandermond*'sche Matrix bekannt. Für die Determinate dieser Matrix gilt

$$\det(V) = \prod_{\substack{i,j \\ i>j}} (\lambda_i - \lambda_j)$$

Sind die Nullstellen λ_i für $i = 1, \cdots, k$ paarweise verschieden, so ist jeder der Faktoren $(\lambda_i - \lambda_j)$ von 0 verschieden und damit ist auch das Produkt von 0 verschieden. Daraus folgt, das das zugehörige lineare Gleichungs-System eindeutig lösbar ist. Mit der Lösung dieses Gleichungs-Systems haben wir dann die Lösung der Rekurrenz-Gleichung $(\star)$ gefunden.

Beispiel: Wie demonstrieren das Verfahren an einem Beispiel: Wie betrachten die Rekurrenz-Gleichung

$$a_{n+2} = a_{n+1} + a_n \quad \text{für alle } n \in \mathbb{N}$$

mit den Anfangs-Bedingungen $a_0 = 1$ und $a_1 = 1$. Die Lösung dieser Rekurrenz-Gleichung sind übrigens gerade die Fibonacci-Zahlen. Das *charakteristische Polynom* dieser Rekurrenz-Gleichung lautet:

$$\chi(x) = x^2 - x - 1.$$

Das führt auf die quadratische Gleichung

$$x^2 - x - 1 = 0$$

Wir haben eben schon gesehen, dass diese quadratische Gleichung die Lösung

$$x_{1/2} = \tfrac{1}{2}(1 \pm \sqrt{5})$$

hat. Wir definieren

$$\lambda_1 = \tfrac{1}{2}(1 + \sqrt{5}) \quad \text{und} \quad \lambda_2 = \tfrac{1}{2}(1 - \sqrt{5}).$$

Damit lautet die *allgemeine Lösung* der betrachteten Rekurrenz-Gleichung

$$a_n = \alpha_1 * \lambda_1^n + \alpha_2 * \lambda_2^n \quad \text{für alle } n \in \mathbb{N}.$$

Setzen wir hier die Anfangs-Bedingungen ein, so erhalten wir

$$\begin{array}{lcl} 1 & = & \lambda_1^0 * \alpha_1 + \lambda_2^0 * \alpha_2 \\ 1 & = & \lambda_1^1 * \alpha_1 + \lambda_2^1 * \alpha_2 \end{array}$$

Dies ist ein lineares Gleichungs-System in den Unbekannten α_1 und α_2. Vereinfachung führt auf

$$\begin{array}{lcl} 1 & = & \alpha_1 + \alpha_2 \\ 1 & = & \lambda_1 * \alpha_1 + \lambda_2 * \alpha_2 \end{array}$$

Die erste dieser beiden Gleichungen lösen wir nach α_2 auf und finden $\alpha_2 = 1 - \alpha_1$. Diesen Wert setzen wir der zweiten Gleichung ein. Das führt auf

$$\begin{array}{rrcl} & 1 & = & \lambda_1 * \alpha_1 + \lambda_2 * (1 - \alpha_1) \\ \Leftrightarrow & 1 & = & (\lambda_1 - \lambda_2) * \alpha_1 + \lambda_2 \\ \Leftrightarrow & 1 - \lambda_2 & = & (\lambda_1 - \lambda_2) * \alpha_1 \\ \Leftrightarrow & \frac{1 - \lambda_2}{\lambda_1 - \lambda_2} & = & \alpha_1 \end{array}$$

Setzen wir diesen Wert in der Gleichung $\alpha_2 = 1 - \alpha_1$ ein, so erhalten wir

$$\alpha_2 = 1 - \frac{1 - \lambda_2}{\lambda_1 - \lambda_2} = \frac{\lambda_1 - 1}{\lambda_1 - \lambda_2}.$$

Berücksichtigen wir die Gleichungen (1) und (2), so finden wir:

$$\alpha_1 = \frac{\lambda_1}{\sqrt{5}} \quad \text{und} \quad \alpha_2 = -\frac{\lambda_2}{\sqrt{5}}.$$

Die Lösung der Rekurrenz-Gleichung

$$a_{n+2} = a_{n+1} + a_n \quad \text{für alle } n \in \mathbb{N}$$

mit den Anfangs-Bedingungen $a_0 = 1$ und $a_1 = 1$ lautet also

$$a_n = \frac{1}{\sqrt{5}} * \left(\lambda_1^{n+1} - \lambda_2^{n+1}\right) \quad \text{für alle } n \in \mathbb{N}.$$

Damit haben wir eine geschlossene Formel zur Berechnung der Fibonacci-Zahlen gefunden. Diese Formel zeigt uns, dass die Fibonacci-Zahlen selbst exponentiell anwachsen. Wir werden diese Lösung bei der Analyse des *Quick-Sort*-Algorithmus benötigen.

Aufgabe: Lösen Sie die Rekurrenz-Gleichung $a_{n+2} = \frac{3}{2} * a_{n+1} - \frac{1}{2} a_n$ mit den Anfangs-Bedingungen $a_0 = 3$ und $a_1 = \frac{5}{2}$.

3.2.1 Entartete Rekurrenz-Gleichungen

Wir hatten oben zunächst den Fall betrachtet, dass das charakteristische Polynom der Rekurrenz-Gleichung $(\star)$ insgesamt k verschiedene Nullstellen hat. Dies muss keineswegs immer der Fall sein. Wir betrachten die Rekurrenz-Gleichung

$$a_{n+2} = 4 * a_{n+1} - 4 * a_n \quad \text{für alle } n \in \mathbb{N} \tag{+}$$

mit den Anfangs-Bedingungen $a_0 = 1$, $a_1 = 4$. Das charakteristische Polynom lautet

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

und hat offensichtlich nur eine Nullstelle bei $x = 2$. Eine Lösung der Rekurrenz-Gleichung (+) lautet daher

$$a_n = 2^n \quad \text{für alle } n \in \mathbb{N}.$$

Eine weitere Lösung ist

$$a_n = n * 2^n \quad \text{für alle } n \in \mathbb{N}.$$

Wir verifizieren dies durch Einsetzen:

$$\begin{array}{rlcll} & (n+2) * 2^{n+2} & = & 4 * (n+1) * 2^{n+1} - 4 * n * 2^n & \mid \div 2^n \\ \Leftrightarrow & (n+2) * 2^2 & = & 4 * (n+1) * 2^1 - 4 * n & \mid \div 4 \\ \Leftrightarrow & n + 2 & = & (n+1) * 2 - n & \\ \Leftrightarrow & n + 2 & = & 2 * n + 2 - n & \\ \Leftrightarrow & n + 2 & = & n + 2 & \end{array}$$

Die allgemeine Lösung der Rekurrenz-Gleichung finden wir durch Linear-Kombination der beiden Lösungen:

$$a_n = \alpha * 2^n + \beta * n * 2^n \quad \text{für alle } n \in \mathbb{N}.$$

Setzen wir hier die Anfangs-Bedingungen $a_0 = 1$ und $a_2 = 4$ ein, so erhalten wir:

$$\left\{ \begin{array}{rcl} 1 & = & \alpha * 2^0 + \beta * 0 * 2^0 \\ 4 & = & \alpha * 2^1 + \beta * 1 * 2^1 \end{array} \right\} \quad \Leftrightarrow \quad \left\{ \begin{array}{rcl} 1 & = & \alpha \\ 4 & = & \alpha * 2 + \beta * 2 \end{array} \right\}$$

Die Lösung lautet offenbar $\alpha = 1$ und $\beta = 1$. Damit lautet die Lösung der Rekurrenz-Gleichung (+) mit den Anfangs-Bedingungen $a_0 = 1$ und $a_2 = 4$

$$a_n = 2^n + n * 2^n = (n+1) * 2^n \quad \text{für alle } n \in \mathbb{N}.$$

Im allgemeinen nennen wir die Rekurrenz-Gleichung

$$a_{n+k} = c_{k-1} * a_{n+k-1} + c_{k-2} * a_{n+k-2} + \cdots + c_1 * a_{n+1} + c_0 * a_n$$

entartet, wenn das charakteristische Polynom

$$\chi(x) = x^k - c_{k-1} * x^{k-1} - c_{k-2} * x^{k-2} - \cdots - c_1 * x - c_0$$

weniger als k verschiedene Nullstellen hat. Dann läßt sich folgendes zeigen: Hat das charakteristische Polynom $\chi(x)$ eine r-fache Nullstelle λ, gilt also

$$\chi(x) = (x - \lambda)^r * \phi(x)$$

mit einem geeigneten Polynom $\phi(x)$, so sind die Folgen

1. $(\lambda^n)_{n\in\mathbb{N}}$
2. $(n * \lambda^n)_{n\in\mathbb{N}}$
3. $(n^2 * \lambda^n)_{n\in\mathbb{N}}$
4. $\vdots$
5. $(n^{r-1} * \lambda^n)_{n\in\mathbb{N}}$

Lösungen der Rekurrenz-Gleichung (+). Durch eine geeignete Linear-Kombination dieser Lösungen zusammen mit den Lösungen, die sich aus den Nullstellen des Polynoms ϕ ergeben, läßt sich dann immer eine Lösung finden, die auch den Anfangs-Bedingungen genügt.

Aufgabe: Lösen Sie die Rekurrenz-Gleichung

$$a_{n+3} = a_{n+2} + a_{n+1} - a_n$$

für die Anfangs-Bedingungen $a_0 = 0$, $a_1 = 3$, $a_2 = 2$.

3.2.2 Inhomogene Rekurrenz-Gleichungen

Definition 6 (Lineare inhomogene Rekurrenz-Gleichung) Die *lineare inhomogene Rekurrenz-Gleichung der Ordnung k mit konstanten Koeffizienten und konstanter Inhomogenität* hat die Form

$$a_{n+k} = c_{k-1} * a_{n+k-1} + c_{k-2} * a_{n+k-2} + \cdots + c_1 * a_{n+1} + c_0 * a_n + c_{-1} \qquad (\#)$$

mit $a_0 = d_0, \cdots, a_{k-1} = d_{k-1}$.

Dabei gilt für die Koeffizienten

$$c_i \in \mathbb{R} \quad \text{für alle } i = -1, 0, \cdots, k-1.$$

Für die *Anfangs-Bedingungen* $d_0, \cdots, d_{k-1}$ gilt ebenfalls

$$d_i \in \mathbb{R} \quad \text{für alle } i = 0, \cdots, k-1.$$

Die Konstante c_{-1} bezeichnen wir als die *Inhomogenität*. □

Wie läßt sich die inhomogene Rekurrenz-Gleichung (#) lösen? Wir zeigen zunächst, wie sich eine *spezielle Lösung* der Rekurrenz-Gleichung (#) finden läßt. Dazu betrachten wir das charakteristische Polynom

$$\chi(x) = x^k - c_{k-1} * a_{n+k-1} - c_{k-2} * a_{n+k-2} - \cdots - c_1 * a_{n+1} - c_0 * a_n$$

und definieren die *Spur* $\mathtt{sp}(\chi)$ wie folgt:

$$\mathtt{sp}(\chi) := \chi(1) = 1 - c_{k-1} - c_{k-2} - \cdots - c_1 - c_0.$$

Es können zwei Fälle auftreten, $\mathtt{sp}(\chi) \neq 0$ und $\mathtt{sp}(\chi) = 0$. Wir betrachten die beiden Fälle getrennt.

1. $\mathtt{sp}(\chi) \neq 0$.

   Dann erhalten wir eine spezielle Lösung von (#) durch den Ansatz

   $$a_n = \delta \quad \text{für alle } n \in \mathbb{N}.$$

   Den Wert von δ bestimmen wir durch Einsetzen, es muß für alle $n \in \mathbb{N}$ gelten:

   $$\delta = c_{k-1} * \delta + c_{k-2} * \delta + \cdots + c_1 * \delta + c_0 * \delta + c_{-1}.$$

   Daraus ergibt sich

   $$\delta * (1 - c_{k-1} - c_{k-2} - \cdots - c_1 - c_0) = c_{-1}.$$

   Das ist aber nichts anderes als

   $$\delta * \mathtt{sp}(\chi) = c_{-1}$$

   und damit lautet eine spezielle Lösung von (#)

   $$a_n = \delta = \frac{c_{-1}}{\mathtt{sp}(\chi)}.$$

   Jetzt sehen wir auch, warum die Voraussetzung $\mathtt{sp}(\chi) \neq 0$ wichtig ist, denn anderfalls wäre der Quotient $\frac{c_{-1}}{\mathtt{sp}(\chi)}$ undefiniert.

2. $\mathtt{sp}(\chi) = 0$.

   In diesem Fall versuchen wir, eine spezielle Lösung von (#) durch den Ansatz

   $$a_n = \varepsilon * n$$

   zu finden. Den Wert ε erhalten wir durch Einsetzen, es muß für alle $n \in \mathbb{N}$ gelten:

   $$\varepsilon * (n + k) = c_{k-1} * \varepsilon * (n + k - 1) + \cdots + c_1 * \varepsilon * (n + 1) + c_0 * \varepsilon * n + c_{-1}$$

   Dies formen wir wie folgt um:

   $$\begin{aligned} \varepsilon * n + \varepsilon * k &= \varepsilon * n * \big(c_{k-1} + c_{k-2} + \cdots + c_1 + c_0\big) \\ &+ \varepsilon * \big((k-1) * c_{k-1} + (k-2) * c_{k-2} + \cdots + 1 * c_1\big) \\ &+ c_{-1} \end{aligned}$$

   Aus $\mathtt{sp}(\chi) = 0$ folgt $1 = c_{k-1} + c_{k-2} + \cdots + c_1 + c_0$ und damit gilt

   $$\varepsilon * n * 1 = \varepsilon * n * \big(c_{k-1} + c_{k-2} + \cdots + c_1 + c_0\big).$$

   Daher vereinfacht sich die obige Gleichung zu

   $$\begin{aligned} & \varepsilon * k = \varepsilon * \big((k-1) * c_{k-1} + (k-2) * c_{k-2} + \cdots + 1 * c_1\big) + c_{-1} \\ \Leftrightarrow \quad & \varepsilon * \big(k - c_{k-1} * (k-1) - c_{k-2}(k-2) - \cdots - c_1\big) = c_{-1} \\ \Leftrightarrow \quad & \varepsilon = \frac{c_{-1}}{\big(k - c_{k-1} * (k-1) - c_{k-2}(k-2) - \cdots - c_1\big)} \end{aligned}$$

   Wenn wir genau hin schauen sehen wir, dass der Wert im Nenner nicht anderes ist als der Wert der Ableitung des charakteristischen Polynoms an der Stelle 1, denn es gilt:

   $$\chi' = \frac{d\chi}{dx} = k * x^{k-1} - c_{k-1} * (k-1) * x^{k-2} - c_{k-2}(k-2) * x^{k-3} - \cdots - c_2 * x - c_1$$

   Setzen wir hier für x den Wert 1 ein, so finden wir

   $$\chi'(1) = k - c_{k-1} * (k-1) - c_{k-2}(k-2) - \cdots - c_1.$$

   Insgesamt haben wir damit also die folgende spezielle Lösung $(a_n)_{n\in\mathbb{N}}$ der Gleichung (#) gefunden:

   $$a_n = \frac{c_{-1}}{\chi'(1)} * n.$$

   Wir haben oben zur Vereinfachung angenommen, dass dieser Wert von 0 verschieden ist, dass also das charakteristische Polynom $\chi(x)$ an der Stelle $x = 1$ keine mehrfache Nullstelle hat, denn nur dann ist ε durch die obige Gleichung wohldefiniert und wir haben eine spezielle Lösung der Rekurrenz-Gleichung (#) gefunden. Andernfalls können wir die Reihe nach die Ansätze $a_n = \varepsilon * n^2$, $a_n = \varepsilon * n^3$, $\cdots$ versuchen, denn es kann folgendes gezeigt werden: Hat

das charakteristische Polynom $\chi(x)$ am Punkt $x = 1$ eine Nullstelle vom Rang r, so führt der Ansatz $a_n = \varepsilon * n^r$ zu einer speziellen Lösung von (#).

Diese spezielle Lösung genügt i. a. noch nicht den Anfangs-Bedingungen. Eine Lösung, die auch den Anfangs-Bedingungen genügt, erhalten wir, wenn wir zu der speziellen Lösung die allgemeine Lösung der zugehörigen homogenen linearen Rekurrenz-Gleichung

$$a_{n+k} = c_{k-1} * a_{n+k-1} + c_{k-2} * a_{n+k-2} + \cdots + c_1 * a_{n+1} + c_0 * a_n$$

addieren und die Koeffizienten der allgemeinen Lösung so wählen, dass die Anfangs-Bedingungen erfüllt sind. Wir betrachten ein Beispiel: Die zu lösende Rekurrenz-Gleichung lautet

$$a_{n+2} = 3 * a_{n+1} - 2 * a_n - 1 \quad \text{für alle } n \in \mathbb{N}.$$

Die Anfangs-Bedingungen sind $a_0 = 1$ und $a_1 = 3$. Wir berechnen zunächst eine spezielle Lösung. Das charakteristische Polynom ist

$$\chi(x) = x^2 - 3 * x + 2 = (x - 1) * (x - 2).$$

Es gilt $\mathtt{sp}(\chi) = \chi(1) = 0$. Wir versuchen für die spezielle Lösung den Ansatz

$$a_n = \varepsilon * n.$$

Einsetzen in die Rekurrenz-Gleichung liefert

$$\varepsilon * (n + 2) = 3 * \varepsilon * (n + 1) - 2 * \varepsilon * n - 1 \quad \text{für alle } n \in \mathbb{N}.$$

Das ist äquivalent zu

$$\varepsilon * (2 - 3) = -1$$

und daraus folgt sofort $\varepsilon = 1$. Damit lautet eine spezielle Lösung

$$a_n = n \quad \text{für alle } n \in \mathbb{N}.$$

Da die Nullstellen des charakteristischen Polynoms $\chi(x)$ bei 1 und 2 liegen, finden wir für die allgemeine Lösung

$$a_n = \alpha * 1^n + \beta * 2^n + n \quad \text{für alle } n \in \mathbb{N}.$$

Setzen wir hier für n die Werte 0 und 1 und für a_n die beiden Anfangs-Bedingungen ein, so erhalten wir das Gleichungs-System

$$\left\{ \begin{array}{rcl} 1 & = & \alpha * 1^0 + \beta * 2^0 + 0 \\ 3 & = & \alpha * 1^1 + \beta * 2^1 + 1 \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} 1 & = & \alpha + \beta \\ 3 & = & \alpha + 2 * \beta + 1 \end{array} \right\}$$

Sie können leicht nachrechnen, dass dieses Gleichungs-System die Lösung $\alpha = 0$ und $\beta = 1$ hat. Damit lautet die Lösung der Rekurrenz-Gleichung

$$a_n = 2^n + n \quad \text{für alle } n \in \mathbb{N}.$$

Aufgabe: Lösen Sie die inhomogene Rekurrenz-Gleichung

$$a_{n+2} = 2 * a_n - a_{n+1} + 3$$

für die Anfangs-Bedingungen $a_0 = 2$ und $a_1 = 1$.

3.2.3 Lineare inhomogene Rekurrenz-Gleichungen mit veränderlichen Inhomogenitäten

Gelegentlich tauchen in der Praxis Rekurrenz-Gleichungen auf, in denen die Inhomogenität keine Konstante ist, sondern von n abhängt. In solchen Fällen führt die Technik des *diskreten Differenzieren* oft zum Erfolg. Wir stellen die Technik an einem Beispiel vor und betrachten die Rekurrenz-Gleichung

$$a_{n+1} = 2 * a_n + n \quad \text{für alle } n \in \mathbb{N} \qquad (\mathtt{I})$$

und der Anfangs-Bedingungen $a_0 = 0$. Das Verfahren zur Lösung solcher Rekurrenz-Gleichung besteht aus vier Schritten:

1. Substitutions-Schritt: Im *Substitutions-Schritt* setzen wir in der ursprünglichen Rekurrenz-Gleichung (I) für n den Wert $n + 1$ ein und erhalten

$$a_{n+2} = 2 * a_{n+1} + n + 1 \quad \text{für alle } n \in \mathbb{N} \qquad (\texttt{II})$$

2. Subtraktions-Schritt: Im *Subtraktions-Schritt* ziehen wir von der im Substitutions-Schritt erhaltenen Rekurrenz-Gleichung (`II`) die ursprüngliche gegebene Rekurrenz-Gleichung (`I`) ab. In unserem Fall erhalten wir

   $$a_{n+2} - a_{n+1} = 2 * a_{n+1} + n + 1 - (2 * a_n + n) \quad \text{für alle } n \in \mathbb{N}.$$

   Vereinfachung dieser Gleichung liefert

   $$a_{n+2} = 3 * a_{n+1} - 2 * a_n + 1 \quad \text{für alle } n \in \mathbb{N}. \qquad (\texttt{III})$$

   Die beiden Schritte 1. und 2. bezeichnen wir zusammen als *diskretes Differenzieren* der Rekurrenz-Gleichung.

3. Berechnung zusätzlicher Anfangs-Bedingungen: Die Rekurrenz-Gleichung (`III`) ist eine inhomogene Rekurrenz-Gleichung der Ordnung 2 mit nun aber konstanter Inhomogenität. Wir haben bereits gesehen, wie eine solche Rekurrenz-Gleichung zu lösen ist, wir benötigen aber eine zusätzliche Anfangs-Bedingung für $n = 1$. Diese erhalten wir, indem wir in der ursprünglichen Rekurrenz-Gleichung (`I`) für n den Wert 0 einsetzen:

   $$a_1 = 2 * a_0 + 0 = 0.$$

4. Lösen der inhomogenen Rekurrenz-Gleichung mit konstanter Inhomogenität: Das charakteristische Polynom der Rekurrenz-Gleichung (`III`) lautet:

   $$\chi(x) = x^2 - 3 * x + 2 = (x - 2) * (x - 1).$$

   Offenbar gilt $\texttt{sp}(\chi) = 0$. Um eine spezielle Lösung der Rekurrenz-Gleichung (`III`) zu erhalten, machen wir daher den Ansatz

   $$a_n = \varepsilon * n$$

   und erhalten

   $$\varepsilon * (n + 2) = 3 * \varepsilon * (n + 1) - 2 * \varepsilon * n + 1$$

   Diese Gleichung liefert die Lösung

   $$\varepsilon = -1.$$

   Damit lautet die allgemeine Lösung der Rekurrenz-Gleichung (`III`):

   $$a_n = \alpha_1 * 2^n + \alpha_2 * 1^n - n$$

   Die Koeffizienten α_1 und α_2 finden wir nun durch Einsetzen der Anfangs-Bedingungen:

   $$\begin{array}{rcl} 0 & = & \alpha_1 + \alpha_2 \\ 0 & = & 2 * \alpha_1 + \alpha_2 - 1 \end{array}$$

   Aus der ersten Gleichung folgt $\alpha_2 = -\alpha_1$. Damit vereinfacht sich die zweite Gleichung zu

   $$0 = 2 * \alpha_1 - \alpha_1 - 1$$

   und damit lautet die Lösung $\alpha_1 = 1$ und $\alpha_2 = -1$. Die Lösung der ursprünglichen Rekurrenz-Gleichung (`I`) mit der Anfangs-Bedingung $a_0 = 0$ ist also

   $$a_n = 2^n - 1 - n.$$

Das oben gezeigte Verfahren funktioniert, wenn die Inhomogenität der Rekurrenz-Gleichung linear ist, also die Form $\delta * n$. Ist die Inhomogenität quadratisch, so können wir die Gleichung durch diskretes Differenzieren auf eine Rekurrenz-Gleichung reduzieren, deren Inhomogenität linear ist. Diese kann dann aber mit dem eben gezeigten Verfahren gelöst werden. Allgemein gilt: Hat die Inhomogenität der Rekurrenz-Gleichung die Form

$$\delta * n^r \quad r \in \mathbb{N} \text{ und } r > 0,$$

so kann die Rekurrenz-Gleichung durch r-maliges diskretes Differenzieren auf eine inhomogene Rekurrenz-Gleichung mit konstanter Inhomogenität reduziert werden.

Aufgabe: Lösen Sie die Rekurrenz-Gleichung

$$a_{n+1} = a_n + 2 * n \quad \text{für alle } n \in \mathbb{N}$$

mit der Anfangs-Bedingung $a_0 = 0$.

Die oben vorgestellte Technik des diskreten Differenzierens führt in leicht variierter Form oft auch dann noch zu einer Lösung, wenn die Inhomogenität nicht die Form eines Polynoms hat. Wir betrachten als Beispiel die Rekurrenz-Gleichung

$$a_{n+1} = a_n + 2^n \tag{I}$$

mit der Anfangs-Bedingungen $a_0 = 0$. Setzen wir in (I) für n den Wert $n + 1$ ein, erhalten wir

$$a_{n+2} = a_{n+1} + 2^{n+1} \tag{II}$$

Würden wir von Gleichung (II) die Gleichung (I) subtrahieren, so würde der Term 2^n erhalten bleiben. Um diesen Term zu eliminieren müssen wir statt dessen von Gleichung (II) 2 mal die Gleichung (I) subtrahieren:

$$a_{n+2} - 2 * a_{n+1} = a_{n+1} + 2^{n+1} - 2 * \left(a_n - 2^n\right)$$

Dies vereinfacht sich zu der homogenen Rekurrenz-Gleichung

$$a_{n+2} = 3 * a_{n+1} - 2 * a_n. \tag{III}$$

Das charakteristische Polynom lautet

$$\chi(x) = x^2 - 3 * x + 2 = (x - 1) * (x - 2).$$

Damit lautet die allgemeine Lösung der homogenen Rekurrenz-Gleichung

$$a_n = \alpha + \beta * 2^n.$$

Da wir hier mit α und β zwei Unbekannte haben, brauchen wir eine zusätzliche Anfangs-Bedingung. Diese erhalten wir, indem wir in der Gleichung (I) für n den Wert 0 einsetzen:

$$a_1 = a_0 + 2^0 = 0 + 1 = 1.$$

Damit erhalten wir das Gleichungs-System

$$\begin{array}{rcl} 0 & = & \alpha + \beta \\ 1 & = & \alpha + 2 * \beta \end{array}$$

Dieses Gleichungs-System hat die Lösung $\alpha = -1$ und $\beta = 1$. Damit lautet die Lösung der Rekurrenz-Gleichung (I) mit der Anfangs-Bedingung $a_0 = 0$

$$a_n = 2^n - 1.$$

3.2.4 Weitere Rekurrenz-Gleichungen

Wir zeigen eine weitere Gruppe von Rekurrenz-Gleichungen, die geschlossen lösbar sind. Wir demonstrieren das Verfahren an Hand der Rekurrenz-Gleichung

$$a_n = a_{n/2} + n \quad \text{für alle } n \in \{2^k : k \in \mathbb{N}\}$$

mit der Anfangs-Bedingung $a_1 = 0$. Um diese Rekurrenz-Gleichung zu lösen, machen wir den Ansatz

$$b_k = a_{2^k} \quad \text{für alle } k \in \mathbb{N}.$$

Setzen wir dies in die ursprüngliche Rekurrenz-Gleichung ein, so erhalten wir

$$b_{k+1} = a_{2^{k+1}} = a_{2^{k+1}/2} + 2^{k+1} = a_{2^k} + 2^{k+1} = b_k + 2^{k+1}.$$

Also muss die Folge $(b_k)_k$ der Rekurrenz-Gleichung

$$b_{k+1} = b_k + 2^{k+1} \quad \text{für alle } k \in \mathbb{N} \tag{I}$$

mit der Anfangs-Bedingung $b_0 = a_{2^0} = a_1 = 0$ genügen. Das ist eine lineare inhomogene Rekurrenz-Gleichung mit der Inhomogenität 2^{k+1}. Wir setzen in (I) für k den Wert $k + 1$ ein und erhalten

$$b_{k+2} = b_{k+1} + 2^{k+2} \quad \text{für alle } k \in \mathbb{N}. \tag{II}$$

Wir multiplizieren nun die Rekurrenz-Gleichung (I) mit 2 und ziehen das Ergebnis von (II) ab:

$$b_{k+2} - 2 * b_{k+1} = b_{k+1} + 2^{k+2} - 2 * b_k - 2 * 2^{k+1} \quad \text{für alle } k \in \mathbb{N}.$$

Nach Vereinfachung erhalten wir

$$b_{k+2} = 3 * b_{k+1} - 2 * b_k \quad \text{für alle } k \in \mathbb{N}. \tag{III}$$

Die Anfangs-Bedingung für $k = 1$ berechnen wir aus (I)

$$b_1 = b_0 + 2^1 = 0 + 2 = 2.$$

Damit haben wir das ursprüngliche Problem auf eine homogene lineare Rekurrenz-Gleichung mit konstanten Koeffizienten zurück geführt. Das charakteristische Polynom dieser Rekurrenz-Gleichung ist

$$\chi(x) = x^2 - 3 * x + 2 = (x - 2) * (x - 1).$$

Damit lautet die allgemeine Lösung der Rekurrenz-Gleichung (III)

$$b_k = \alpha_1 * 2^k + \alpha_2 * 1^k \quad \text{für alle } k \in \mathbb{N}.$$

Wir setzen die Anfangs-Bedingungen ein und erhalten so für die Koeffizienten α_1 und α_2 das lineare Gleichungs-System

$$\begin{array}{rcl} 0 & = & \alpha_1 + \alpha_2 \\ 2 & = & 2 * \alpha_1 + \alpha_2 \end{array}$$

Ziehen wir die erste Gleichung von der zweiten ab, so sehen wir $\alpha_1 = 2$. Dann folgt aus der ersten Gleichung $\alpha_2 = -2$. Damit haben wir

$$b_k = 2^{k+1} - 2 \quad \text{für alle } k \in \mathbb{N}.$$

Setzen wir hier $b_k = a_{2^k}$ ein, so finden wir

$$a_{2^k} = 2^{k+1} - 2 \quad \text{für alle } k \in \mathbb{N}.$$

Mit $n = 2^k$ wird das

$$a_n = 2 * n - 2 \quad \text{für alle } n \in \{2^k : k \in \mathbb{N}\}.$$

Aufgabe: Lösen Sie die Rekurrenz-Gleichung

$$a_n = a_{n/2} + 1 \quad \text{für alle } n \in \{2^k : k \in \mathbb{N}\}$$

mit der Anfangs-Bedingungen $a_1 = 1$.

Die Lösung allgemeiner Rekurrenz-Gleichungen kann beliebig schwierig sein und es gibt viele Fälle in denen eine gegebene Rekurrenz-Gleichungen überhaupt keine Lösung hat, die sich durch elementare Funktionen als geschlossene Formel ausdrücken läßt. Wir haben im letzen Abschnitt eine Reihe von Methoden kennengelernt, mit denen wir versuchen können, eine Rekurrenz-Gleichung zu lösen. Diese Methoden reichen aber für die Praxis nicht aus. Daher werden wir im Laufe der Vorlesung noch weitere Methoden zur Lösung von Rekurrenz-Gleichung präsentieren. Wir warten damit aber bis wir die Beispiele kennenlernen, für deren Behandlung die neuen Methoden gebraucht werden.

3.3 Die $\mathcal{O}$-Notation

Wollen wir die Komplexität eines Algorithmus abschätzen, so wäre ein mögliches Vorgehen wie folgt: Wir kodieren den Algorithmus in einer Programmiersprache und berechnen, wieviele Additionen, Multiplikationen, Zuweisungungen, und andere elementare Operationen bei einer gegebenen Eingabe von dem Programm ausgeführt werden. Anschließend schlagen wir im Prozessor-Handbuch nach, wieviel Zeit die einzelnen Operationen in Anspruch nehmen und errechnen daraus die Gesamtlaufzeit des Programms.[2] 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.

  Läuft ein Programm auf einem Rechner, dessen Prozessor die doppelte Takt-Frequenz hat wie der Prozessor eines anderen Rechners, so könnten wir in erster Näherung erwarten, dass das Programm dort auch doppelt so schnell läuft.

- 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

  $$T(n) = 3 * n^3 + 2 * n^2 + 7$$

  gefunden. Dann nimmt der *proportionale Anteil* des Terms $2 * n^2 + 7$ 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:

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

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

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

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

Dann kann die Menge aller Funktionen von $\mathbb{N}$ nach $\mathbb{R}_+$ als $\mathbb{R}_+^{\mathbb{N}}$ geschrieben werden:

$$\mathbb{R}_+^{\mathbb{N}} = \big\{f \subseteq \mathbb{N} \times \mathbb{R}_+ \mid f \text{ funktionale Relation auf } \mathbb{N}\big\}$$

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

Definition 7 ($\mathcal{O}(f)$) Es sei eine Funktion

$$f : \mathbb{N} \to \mathbb{R}_+$$

gegeben. Dann definieren wir die Menge der Funktionen, die asymptotisch das gleiche Wachstumsverhalten haben wie die Funktion f wie folgt:

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

Was sagt die obige Definition aus? Zunächst kommt es auf kleine Werte des Arguments n nicht an, denn die obige Formel sagt ja, dass $g(n) \leq c * f(n)$ 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 $c * f(n)$ sein und die Konstante c können wir beliebig wählen. Um den Begriff zu verdeutlichen, geben wir einige Beispiele.

Beispiel: Es gilt

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

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

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

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

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

voraussetzen und müssen zeigen, dass daraus

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

folgt. Erheben wir beide Seiten der Ungleichung (1) in die dritte Potenz, so sehen wir, dass

$$1 \leq n^3$$

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

$$7 \leq 7 * n^3. \tag{3}$$

Multiplizieren wir die Ungleichung (1) mit $2 * n^2$, so erhalten wir

$$2 * n^2 \leq 2 * n^3. \tag{4}$$

Schließlich gilt trivialerweise

$$3 * n^3 \leq 3 * n^3. \tag{5}$$

Die Addition der Ungleichungen (3), (4) und (5) liefert nun

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

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 $n \geq k$

$$n \leq c * 2^n$$

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

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

durch vollständige Induktion über n.

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

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

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

   Einerseits gilt nach Induktions-Voraussetzung

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

   andererseits haben wir

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

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

$$n + 1 \leq 2^n + 2^n = 2^{n+1}.$$ □

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

Aufgabe: Zeigen Sie

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

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

Satz 8 (Reflexivität) Für alle Funktionen $f: \mathbb{N} \to \mathbb{R}_+$ gilt

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

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

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

Satz 9 (Abgeschlossenheit unter Multiplikation mit Konstanten)
Es seien $f, g: \mathbb{N} \to \mathbb{R}_+$, und $d \in \mathbb{N}$. Dann gilt

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

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

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

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

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

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

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

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

Satz 10 (Abgeschlossenheit unter Addition) Es seien $f, g, h: \mathbb{N} \to \mathbb{R}_+$. Dann gilt

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

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

$$\forall n \in \mathbb{N}: n \geq k_1 \to f(n) \leq c_1 * h(n) \quad \text{und}$$
$$\forall n \in \mathbb{N}: n \geq k_2 \to g(n) \leq c_2 * h(n)$$

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

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

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

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

Satz 11 (Transitivität) Es seien $f, g, h: \mathbb{N} \to \mathbb{R}_+$. Dann gilt

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

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

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

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

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

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

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

Die zweite dieser Ungleichungen multiplizieren wir mit c_1 und erhalten

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

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

Satz 12 (Grenzwert-Satz) Es seien $f, g: \mathbb{N} \to \mathbb{R}_+$. Außerdem existiere der Grenzwert

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

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

Beweis: Es sei

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

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

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

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

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

Daraus folgt wegen

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

die Ungleichung

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

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

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

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

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

Beweis: Es gilt

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

Die Behauptung folgt nun aus dem Grenzwert-Satz. □

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

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

Beweis: Wir zeigen, dass

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = 0 \qquad (\star)$$

ist, denn dann folgt die Behauptung aus dem Grenzwert-Satz. Um $(\star)$ einzusehen, definieren wir reellwertigen Funktionen $f : \mathbb{R}_+ \to \mathbb{R}_+$ und $g : \mathbb{R}_+ \to \mathbb{R}_+$ mit

$$f(x) := x^k \quad \text{und} \quad g(x) := \lambda^x.$$

Nach dem Satz von L'Hospital können wir den Grenzwert wie folgt berechnen

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

Die Ableitungen können wir berechnen, es gilt:

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

Berechnen wir die zweite Ableitung so sehen wir

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

Für die k-te Ableitung gilt analog

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

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

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

□

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

Beweis: Wir benutzen Satz 12 und zeigen mit der Regel von L'Hospital, dass

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

ist. Es gilt

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

Also haben wir

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

□

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

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

Beweis: Zunächst haben wir

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

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

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

Wir logarithmieren beide Seiten dieser Ungleichung und finden

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

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

Aufgaben:

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

3.4 Ein Beispiel

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 3.3 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

$$m^n = \underbrace{m * \cdots * m}_{n}$$

durchzuführen.

```
1  static BigInteger power(BigInteger base, int n)
2  {
3      if (n == 0)
4          return BigInteger.valueOf(1);
5      BigInteger result = base;
6      for (int i = 2; i <= n; ++i) {
7          result = result.multiply(base);
8      }
9      return result;
10 }
```

Abbildung 3.3: 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

$$m^4 = (m * m) * (m * m).$$

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

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

Berechnen wir den Term $(m*m)*(m*m)$ 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 3.4 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:

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

Da es keineswegs offensichtlich ist, dass das Programm in 3.4 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.

```
1  static BigInteger power(BigInteger base, int n)
2  {
3      if (n == 0)
4          return BigInteger.valueOf(1);
5      BigInteger p = power(base, n / 2);
6      if (n % 2 == 0) {
7          return p.multiply(p);
8      } else {
9          return p.multiply(p).multiply(base);
10     }
11 }
```

Abbildung 3.4: Berechnung von m^n für $m, n \in \mathbb{N}$.

2. *Induktions-Schritt*

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

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

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

1. **Induktions-Anfang**.

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

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

2. **Induktions-Schritt**.

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

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

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

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

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

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

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

Damit ist der Beweis der Korrektheit abgeschlossen. □

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

$$n/2 = 2^{l-1} - 1$$

und das heißt, dass bei jedem rekursiven Aufruf der Exponent ungerade ist. 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:

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

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

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

wobei wir bei der Division $(2^l - 1)/2$ das Ergebnis abgerundet haben. Die Anfangs-Bedingungen $b_0 = a_{2^0-1} = a_0 = 0$ lautet. Um eine Rekurrenz-Gleichung in der gewohnten Form zu erhalten, substituieren wir $l \mapsto l + 1$ und erhalten dann für die Folge $(b_l)_{l\in\mathbb{N}}$ die Rekurrenz-Gleichung

$$b_{l+1} = b_l + 2 \qquad \text{für alle } l \in \mathbb{N} \quad \text{mit } b_0 = 0.$$

Dies ist eine lineare Rekurrenz-Gleichung erster Ordnung mit konstanter Inhomogenität. Das charakteristische Polynom lautet

$$\chi(x) = x - 1.$$

Da $\mathtt{sp}(\chi) = \chi(1) = 0$ ist, lautet die spezielle Lösung der Rekurrenz-Gleichung

$$b_l = \varepsilon * l \qquad \text{mit } \varepsilon = \frac{2}{\chi'(1)}.$$

Wegen $\chi'(x) = 1$ folgt $\varepsilon = 2$. Damit lautet die spezielle Lösung

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

und die allgemeine Lösung ist dann

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

Wir bestimmen den Parameter α indem wir die Anfangs-Bedingungen $b_0 = 0$ berücksichtigen:

$$0 = \alpha + 2 * 0.$$

Daraus folgt sofort $\alpha = 0$. Also lautet die Lösung

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

Für die Folge a_n haben wir dann:

$$a_{2^l-1} = 2 * l.$$

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

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

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

$$n = 2^l$$

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

Zunächst gilt $a_{2^0} = a_1 = 2$, denn wenn $n = 1$ ist, scheitert der Test in Zeile 6 und Zeile 9 liefert 2 Multiplikationen. Zeile 5 liefert in diesem Fall keine Multiplikation, weil beim Aufruf `power`$(m, 0)$ sofort das Ergebnis in Zeile 3 zurück gegeben wird.

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

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

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

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

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

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

Dies ist eine lineare Rekurrenz-Gleichung erster Ordnung mit konstanter Inhomogenität. Das charakteristische Polynom dieser Rekurrenz-Gleichung lautet

$$\chi(x) = x - 1$$

und es gilt $\mathtt{sp}(\chi) = \chi(1) = 0$. Wegen $\chi'(x) = 1$ lautet die spezielle Lösung der Rekurrenz-Gleichung

$$b_l = \frac{1}{\chi'(1)} * l = \frac{1}{1} * l = 1 * l = l \qquad \text{für alle } l \in \mathbb{N}.$$

Damit lautet die allgemeine Lösung

$$b_l = \alpha + l.$$

Wir bestimmen den Parameter α, indem wir die Anfangs-Bedingungen $b_0 = 2$ berücksichtigen:

$$2 = \alpha + 0.$$

Daraus folgt sofort $\alpha = 2$. Damit lautet nun die Lösung

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

Dann finden wir für a_n:

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

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

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

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

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

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

Aufgabe: Schreiben Sie eine Prozedur `prod` zur Multiplikation zweier Zahlen. Für zwei natürliche Zahlen m und n soll der Aufruf `prod`(m, n) das Produkt $m*n$ 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.

Kapitel 4

Abstrakte Daten-Typen und elementare Daten-Stukturen

Ähnlich wie der Begriff des Algorithmus von bestimmten Details eines Programms abstrahiert, abstrahiert der Begriff des *abstrakten Daten-Typs* von bestimmten Details konkreter Daten-Stukturen. Durch die Verwendung dieses Begriffs wird es möglich, Algorithmen von den zugrunde liegenden Daten-Stukturen zu trennen. Wir geben im nächsten Abschnitt eine Definition von abstrakten Daten-Typen und illustrieren das Konzept im folgenden Abschnitt an Hand eines einfachen Beispiels. In einem weiteren Abschnitt zeigen wir dann, wie die Sprache SETL2 die Verwendung abstrakter Daten-Typen unterstützt.

4.1 Abstrakte Daten-Typen

Formal ist ein *abstrakter Daten-Typ* ein Tupel $\langle T, P, Fz, Ts, Ax \rangle$. 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.
4. Ts ist eine Menge von Typ-Spezifikation, die zu jedem Funktions-Zeichen f eine *Typ-Spezifikation* der Form
   $$f : T_1 \times \cdots \times T_n \to S.$$
   enthält. Dabei sind T_1, $\cdots$, T_n und S Namen von Daten-Typen. Hier gibt es drei Möglichkeiten:
   (a) Namen konkreter Daten-Typen, wie z. B. "`int`" oder "`String`".
   (b) Namen abstrakter Daten-Typen.
   (c) Typ-Parameter.

   Die Typ-Spezifikation $f : T_1 \times \cdots \times T_n \to S$ drückt aus, dass die Funktion f in der Form
   $$f(t_1, \cdots, t_n)$$
   aufgerufen wird und dass für $i = 1, \cdots, n$ das Argument t_i vom Typ T_i sein muß. Außerdem sagt die Typ-Spezifikation aus, dass das Ergebnis, das von der Funktion f berechnet wird, immer vom Typ S ist.

Zusätzlich fordern wir, dass entweder $T_1 = T$ ist, oder aber $S = T$ gilt. Es soll also entweder das erste Argument der Funktion f den Wert T haben, oder der Typ des von f berechneten Ergebnisses soll gleich T sein. Falls $T_1 \neq T$ ist (und damit zwangsläufig $S = T$ gilt), dann nennen wir die Funktion f auch einen *Konstruktor* des Daten-Typs T, andernfalls bezeichnen wir f als *Methode*.

5. Ax ist eine Menge von prädikaten-logischen Formeln, die das Verhalten des abstrakten Daten-Typs beschreiben. Diese Formeln bezeichnen wir auch als die *Axiome* des Daten-Typs.

Wir geben sofort ein einfaches Beispiel für einen abstrakten Daten-Typ: den *Keller* (engl. *stack*). Ein Keller kann man sich anschaulich als einen Stapel von Elementen eines bestimmten Typs vorstellen, die aufeinander gelegt werden, ähnlich wie die Teller in der Essensausgabe einer Kantine. Dort werden Teller immer oben auf den Stapel gelegt und in umgekehrter Reihenfolge wieder vom Stapel entfernt. Insbesondere ist es nicht möglich, einen Teller aus der Mitte des Stapels zu entfernen. Formal definieren wir den Daten-Typ des *Kellers* wie folgt:

1. Als Namen wählen wir *Stack*.
2. Die Menge der Typ-Parameter ist $\{Element\}$.
3. Die Menge der Funktions-Zeichen ist
   $$\big\{Stack, push, pop, top, isEmpty\big\}.$$
4. Die Typ-Spezifikationen der Funktions-Zeichen sind wie folgt:
   (a) $Stack : Stack$
   Links von dem Doppelpunkt steht hier die Funktion mit dem Namen *Stack*, rechts steht der Name des ADT. In den gängigen Programmier-Sprachen (*Java*, `C++`, etc.) werden bestimmte Funktionen mit dem selben Namen bezeichnet wie der zugehörige ADT. Solche Funktionenen heißen Konstruktoren. Der Rückgabe-Wert eines Konstruktors hat immer den Typ des ADT.
   Der Konstruktor *Stack* kommt ohne Eingabe-Argumente aus. Ein solcher Konstruktor wird auch als der *Default-Konstruktor* bezeichnet.
   Der Aufruf $Stack()$ erzeugt einen neuen, leeren Stack.
   (b) $push : Stack \times Element \to Stack$
   Der Aufruf $push(S, x)$ legt das Element x oben auf den Stack S. Wir werden im folgenden eine Objekt-orientierte Schreibweise verwenden und den Aufruf $push(S, x)$ als $S.push(x)$ schreiben.
   (c) $pop : Stack \to Stack$
   Der Aufruf $S.pop()$ entfernt das oberste Element von dem Stack S.
   (d) $top : Stack \to Element$
   Der Aufruf $S.top()$ liefert das auf dem Stack S zuoberst liegende Element.
   (e) $isEmpty : Stack \to \mathbb{B}$
   Der Aufruf $S.isEmpty()$ testet, ob der Stack S leer ist.

Die Anschauung, die dem Begriff des Stacks zu Grunde liegt, wird durch die folgenden Axiome erfaßt:

1. $Stack().top() = \Omega$
   Hier bezeichnet Ω den undefinierten Wert. Das Axiom drückt aus, das ein leerer Stack kein oberstes Element hat.
2. $S.push(x).top() = x$
   Legen wir auf den Stack S mit $S.push(x)$ ein Element x, so ist x das oberste Element, was auf dem neu erhaltenen Stack liegt.

3. $Stack().pop() = \Omega$

   Der Versuch, von einem leeren Stack das oberste Element zu entfernen, liefert einen undefinierten Stack.

4. $S.push(x).pop() = S$

   Wenn wir auf den Stack S ein Element legen, und anschließend von dem resultierenden Stack das oberste Element wieder herunter nehmen, dann erhalten wir den ursprünglichen Stack S, mit dem wir gestartet sind.

5. $Stack().isEmpty() =$ `true`

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

6. $S.push(x).isEmpty() =$ `false`

   Legen wir ein Element x auf einen Stack S, so kann der Stack S danach nicht leer sein.

Beim Betrachten der Axiome läßt sich eine gewisse Systematik erkennen. Bezeichnen wir die Funktionen `Stack` und `push` als Generatoren so geben die Axiome das Verhalten der restlichen Funktionen auf den von den Generatoren erzeugten Stacks an.

Stacks spielen in vielen Bereichen der Informatik eine wichtige Rolle. Es gibt sogar Stack-basierte Programmier-Sprachen: Dort müssen bei einem Funktions-Aufruf alle Argumente zunächst auf einen Stack gelegt werden. Die aufrufende Funktion nimmt dann ihre Argumente vom Stack und legt das berechnete Ergebnis wieder auf den Stack. Die Sprache *PostScript* funktioniert nach diesem Prinzip. Die Sprache *Java* wird in einen *Byte-Code* übersetzt, der von der *Java Virtual Machine* (kurz JVM) interpretiert wird. Die JVM ist ebenfalls stack-basiert.

Wir werden später noch sehen, wie arithmetische Ausdrücke mit Hilfe eines Stacks ausgewertet werden können. Vorher zeigen wir, wie sich der abstrakte Daten-Typ des Stacks in der Programmier-Sprache *Java* implementieren läßt.

4.2 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 4.1 auf Seite 37 gezeigt wird. Die Darstellung durch eine abstrakte Klasse ist insofern flexibler, als wir hier die Möglichkeit haben, zusätzliche 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 die Klasse `Stack<Element>` als *abstrakt*. Das Schlüsselwort `abstract` drückt dabei aus, das wir in dieser Klasse lediglich die Signaturen der Methoden angeben, die Implementierungen der Methoden werden in der abstrakten Klasse nicht angegeben.

   Der Name der Klasse ist `Stack<Element>`. Die Typ-Parameter des ADT sind hier in spitzen Klammern eingefaßt. Wenn es mehr als einen Typ-Parameter gibt, dann müssen diese durch Kommata getrennt werden.

   Durch "`implements Cloneable`" drücken wir aus, dass Objekte der Klasse `Stack<Element>` geklont, d.h. kopiert werden können.

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

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

   angegeben, in *Java* hat diese Typ-Spezifikation die Form

   ```
   void push(Element e);
   ```

   Hier fallen zwei Dinge auf:

```
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 abstract 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 4.1: darstellung des ADT-Stack in *Java*.

(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

$$s.\texttt{push}(e)$$

an Stelle von $\texttt{push}(s, e)$.

(b) Der Rückgabe-Typ von `push` ist als `void` deklariert und nicht als `Stack<Element>`. Der Grund ist, dass ein Aufruf der Form

$$s.\texttt{push}(e)$$

nicht einen neuen Stack berechnet, sondern den als implizites erstes Argument gegeben Stack s verändert.

3. Die Zeilen 4 – 6 enthalten die Typ-Spezifikationen der restlichen Methoden.

4. Die Zeilen 8 – 10 enthält die Definition einer Methode `clone()`. Diese Methode ermöglicht es, einen Klon (also eine Kopie) eines Objektes vom Typ `Stack` zu erzeugen. Die Implementierung diskutieren wir später.

5. In den Zeilen 12 – 25 definieren wir die Methode `toString()`, mit der wir ein Objekt vom Daten-Typ `Stack` in einen String umwandeln können. Um die Implementierung dieser Methode zu verstehen, betrachten wir zunächst die Wirkung dieser Methode an Hand des folgenden Beispiels:

   ```
   Stack<Element> stack = new Stack<Element>();
   stack.push(1);
   stack.push(2);
   stack.push(3);
   stack.toString();
   ```

   Dann hat der Ausdruck `stack.toString()` den Wert

   ```
   -------------
   | 1 | 2 | 3 |
   -------------
   ```

   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. Diesen Mechanismus werden wir einstweilen als Black-Box betrachten.

   (b) Dann berechnet die Hilfs-Methode `convert()` einen String der Form

   ```
   | 1 | 2 | 3 |.
   ```

   Hierzu wird mit der `if`-Abfrage in Zeile 28 eine Fallunterscheidung durchgeführt: Falls der Stack leer ist, so ist das Ergebnis einfach nur der String "`|`". Andernfalls fragen wir das oberste Element des Stacks mit dem Aufruf `top()` in 31 ab, entfernen es durch einen Aufruf von `pop()` vom Stack und rufen anschließend für den so verkleinerten Stack rekursiv die Methode `toString()` auf. Das oberste Element des ursprünglichen Stacks wird dann hinten an das Ergebnis des rekursiven Aufrufs gehängt.

   (c) Um zum Schluß noch die Linien darüber und darunter zu zeichnen, erzeugen wir in der `for`-Schleife in Zeile 21 – 23 eine Linie der erforderlichen Länge und verketten diese mit dem von `convert()` gelieferten String.

Beachten Sie, dass wir für Stacks die Methode `toString()` implementieren konnten ohne etwas darüber zu wissen, wie die Stacks überhaupt implementiert werden. Dies ist der wesentliche Vorteil des Konzeptes des ADT: Der Begriff des ADT abstrahiert von den Details der Implementierung und bietet damit eine Schnittstelle zu Stacks die einfacher zu bedienen ist, als wenn wir uns mit allen Details auseinander setzen müßten. Ein weiterer wesentlicher Vorteil ist die Austauschbarkeit.

Wir werden später verschiedene Implementierungen des ADT `Stack` entwickeln. Da die Methode `toString` auf der abstrakten Ebene entwickelt worden ist, ist sie von den Details einer konkreten Implementierung unabhängig und funktioniert für jede korrekte Implementierung des ADT `Stack`!

4.3 Implementierung eines Stacks mit Hilfe eines *Arrays*

Die naheliegenste Möglichkeit, einen Stack zu implementieren, besteht darin, einen Array (Feld) zu verwenden. Abbildung 4.2 auf Seite 39 zeigt eine Implementierung in *Java*, die auf auf einem Array basiert.

```
1  public class ArrayStack<Element> extends Stack<Element>
2  {
3      Element[] mArray;
4      int       mIndex;
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 4.2: Array-basierte Implementierung eines Stacks.

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 ein Feld, in dem die einzelnen Elemente, die auf den Stack geschoben werden, gespeichert sind.

(b) Die in Zeile 4 definierte Variable `mIndex` gibt den Index in dem Feld `mArray` an, in dem das nächste Element abgelegt werden kann.

3. In dem Konstruktor legen wir in Zeile 7 das Feld `mArray` zunächst 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 11, 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 auf das neue Feld. Der *Garbage-Collector* sorgt jetzt dafür, dass das alte Feld recycled wird.

   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.

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.

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 4.3 auf Seite 40 zeigt ein solches Programm. Wir legen nacheinander die Zahlen $0, 1, \cdots, 32$ auf den Stack und geben jedesmal den Stack aus. Anschließend nehmen wir diese Zahlen der Reihe nach vom Stack herunter.

```
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 4.3: Test der Stack-Implementierung.

4.4 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 4.4 auf Seite 42 zeigt die Implementierung.

Um eine verkette Liste darzustellen, brauchen wir eine Daten-Struktur die Paare darstellt. Dabei ist die erste Komponente eines solchen Paares ein Element, das abgespeichert werden soll, und die zweite Komponente ist ein Zeiger auf das nächste Paar. In der Klasse `ListStack<Element>` definieren wir daher zunächst eine *lokale* 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 einen Zeiger auf das folgende Paar. Der Zeiger wird in der Variablen `mNextPointer` abgespeichert.

2. Der Konstruktor dieser Klasse bekommt als Argumente ein abzuspeicherndes Element und einen Zeiger auf das nächste Paar. Mit diesen Argumenten werden dann die Variablen `mData` und `mNextPointer` initialisiert.

3. Weierhin enthält die Klasse noch die Methode `recursiveCopy()`, die später gebraucht wird um eine Liste zu klonen. Diese Methode wollen wir jetzt als Black-Box betrachten und nicht weiter erläutern.

4. Die Klasse `ListStack<Element>` selber enthält als einzige Member-Variable dem Zeiger `mPointer`. Wenn der Stack leer ist, dann hat dieser Pointer den Wert 0. Sonst zeigt der Pointer 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 einen Zeiger auf die Liste enthält, die den bisherigen Stack repräsentierte. Anschließend lassen wir `mPointer` auf dieses Paar zeigen.

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 4.3 wie folgt zu ändern:

```
Stack<Integer> stack = new ListStack<Integer>();
```

```
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 4.4: Implementierung eines Stacks mit Hilfe einer Liste

4.5 Auswertung arithmetischer Ausdrücke

Wir zeigen jetzt, wie Stacks zur Auswertung arithmetischer Ausdrücke benutzt werden können. Unter einem *arithmetischen Ausdruck* verstehen wir in diesem Zusammenhang einen String, der aus natürlichen Zahlen und den Operator-Symbolen "`+`", "`-`", "`*`", "`/`", "`%`" und "`^`" aufgebaut ist. Hierbei steht $x \,\%\, y$ für den Rest, der bei der Division von x durch y übrig bleibt und $x \hat{}\, y$ steht für die Potenz x^y. Alternativ kann die Potenz x^y auch als $x ** y$ geschrieben werden. Außerdem können arithmetische Ausdrücke noch die beiden Klammer-Symbole "`(`" und "`)`" enthalten. 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.

4.5.1 Ein einführendes Beispiel

Wir demonstrieren das Verfahren, mit dem wir arithmetische Ausdrücke auswerten, zunächst an Hand eines Beispiels. Wir betrachten den arithmetischen Ausdruck

`1 + 2 * 3 - 4`.

Wir verarbeiten einen solchen Ausdruck von links nach rechts, Token für Token. Ein *Token* ist dabei entweder eine Zahl, eines der Operator-Symbole oder ein Klammer-Symbol. Bei der Verarbeitung benutzen wir drei Stacks:

1. Der *Token-Stack* enthält die eingegebenen Token. Dieser Stack enthält also sowohl Zahlen als auch Operator-Symbole und Klammer-Symbole.

2. Der *Argument-Stack* enthält Zahlen.

3. Der *Operator-Stack* enthält Operator-Symbole und Klammer-Symbole der Form "`(`".

Die Auswertung von `1 + 2 * 3 - 4` verläuft wie folgt:

1. Zu Beginn des Algorithmus enthält der Token-Stack die eingegebenen Tokens und die anderen beiden Stacks sind leer:

   ```
   mTokens    = [ 4, "-", 3, "*", 2, "+", 1 ],
   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. Dann gilt:

   ```
   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.

4.5.2 Ein Algorithmus zur Auswertung arithmetischer Ausdrücke

Nach dem einführenden Beispiel entwickeln wir nun einen Algorithmus zur Auswertung arithmetischer Ausdrücke. Zunächst legen wir fest, welche Daten-Stukturen wir benutzen wollen.

1. `mTokens` ist ein Stack von Eingabe-Token. Wenn es sich bei den Token um Operatoren oder Klammer-Symbole handelt, dann haben diese Token den Typ `String`. Andernfalls stellen die Token Zahlen dar und haben den Typ `BigInteger`. Die gemeinsame Oberklasse der Klassen `String` und `BigInteger` ist `Object`. Daher deklarieren wir die Variable `mTokens` in der zu entwickelnden Klasse `Calculator` als:

    ```
    ArrayStack<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, stellen wir fest, dass wir Zahlen immer auf den Argument-Stack gelegt haben, 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 Operator bindet stärker als der Operator, der oben auf dem Operator-Stack liegt.
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 4.5, 4.6 und 4.7 auf den Seiten 46, 48 und 49 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 4.1 zeigt die Präzedenzen der von uns verwendeten Operatoren.

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

```
1  import java.util.*;
2  import java.math.*;
3
4  public class Calculator {
5      Stack<BigInteger> mArguments;
6      Stack<String>     mOperators;
7      Stack<Object>     mTokenStack;
8
9      static boolean evalBefore(String op1, String op2) {
10         if (op1.equals("(")) {
11             return false;
12         }
13         if (precedence(op1) > precedence(op2)) {
14             return true;
15         } else if (precedence(op1) == precedence(op2)) {
16             return op1.equals(op2) ? isLeftAssociative(op1) : true;
17         } else {
18             return false;
19         }
20     }
21     static int precedence(String operator) {
22         if (operator.equals("+") || operator.equals("-")) {
23             return 1;
24         } else if ( operator.equals("*") || operator.equals("/") ||
25                     operator.equals("%")) {
26             return 2;
27         } else if (operator.equals("**") || operator.equals("^")) {
28             return 3;
29         } else {
30             System.out.println("ERROR: *** unkown operator *** ");
31         }
32         System.exit(1);
33         return 0;
34     }
35     static boolean isLeftAssociative(String operator) {
36         if (operator.equals("+") || operator.equals("-") ||
37             operator.equals("*") || operator.equals("/") ||
38             operator.equals("%")) {
39             return true;
40         } else if (operator.equals("**") || operator.equals("^")) {
41             return false;
42         } else {
43             System.out.println("ERROR: *** unkown operator *** ");
44         }
45         System.exit(1);
46         return false;
47     }
```

Abbildung 4.5: Die Klasse `Calculator`

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

Tabelle 4.1: Präzedenzen der Operatoren.

kleiner als die Präzedenz des Operators `op2`, so wird der Operator `op2` auf den Operator-Stack gelegt. In dem Fall, dass die Präzedenzen von `op1` und `op2` gleich sind, gibt es zwei Fälle:

1. `op1` $\neq$ `op2`.

   Betrachten wir eine Beispiel: Der arithmetischer Ausdruck

   `2 + 3 - 4` wird implizit links geklammert: `(2 + 3) - 4`.

   Also wird in diesem Fall zunächst `op1` ausgewertet.

2. `op1` = `op2`.

   In diesem Fall spielt die *Assoziativität* des Operators eine Rolle. Betrachten wir dazu zwei Beispiele:

   `2 + 3 + 4` wird interpretiert wie `(2 + 3) + 4`,

   denn wir sagen, dass der Operator "`"+"`" *links-assoziativ* ist. Andererseits wird

   `2 ^ 3 ^ 4` interpretiert als `2 ^ (3 ^ 4)`,

   denn wir sagen, dass der Operator "`^`" *rechts-assoziativ* ist.

   Die Operatoren "`"+"`", "`"-"`", "`"*"`", "`/`" und "`%`" sind alle links-assoziativ. Hier wird als zunächst `op1` ausgewertet. Der Operator "`^`" ist rechts-assoziativ. Ist der oberste Operator auf dem Operator-Stack also "`^`" und wird dann nochmal der Operator "`^`" gelesen, so wird auch die neue Instanz dieses Operators auf den Stack gelegt.

Mit diesem Vorüberlegung können wir nun die Implementierung von `evalBefore`(`op1`, `op2`) in Abbildung 4.5 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) Andernfalls hat das Ergebnis von `evalBefore(op1,op2)` den Wert `false`.

   Diese beiden Fälle werden in Zeile 16 behandelt.

4. Ist die Präzedenz des Operators `op1` kleiner als die Präzedenz des Operators `op2`, so liefert `evalBefore`(`op1`, `op2`) in Zeile 18 den Wert `false`.

Die Implementierung der Methode `predence()` in den Zeilen 21 – 34 ergibt sich unmittelbar aus der Tabelle 4.1 auf Seite 47. 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 4.6 auf Seite 48 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 4.6: Die Klasse `Calculator`

Damit können wir die 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 initialiseren wir den Argument-Stack und den Operator-Stack in Zeile 75 und 76.

2. In der nächsten Phase verarbeiten wir die einzelnen Tokens des Token-Stacks und verteilen diese Tokens auf Argument-Stack und Operator-Stack wie folgt:

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

   Im folgenden können wir immer davon ausgehen, dass das gelesene Token ein Operator oder eine der beiden Klammern `"("` oder `")"` ist.

```
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 4.7: Die Klasse `Calculator`

(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 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.

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 vor dem zuletzt gelesenen Operator auszuwerten ist, 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()` aus. Wenn die Eingabe ein syntaktisch korrekter arithmetischer Ausdruck war, dann sollte nun 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 4.8 noch die Implementierung der Klasse `MyScanner`.

4.6 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 der Daten-Strukturen von der Implementierung eines Algorithmus unabhängig.

   Bei der Implementierung des Algorithmus zur Auswertung arithmetischer Ausdrücke mußten wir uns um die zugrunde liegenden Daten-Strukturen nicht weiter kümmern. Es reichte aus, zwei Dinge zu wissen:

   (a) Die Typ-Spezifikationen der verwendeten Funktionen.

   (b) Die Axiome, die das Verhalten dieser Funktionen beschreiben.

   Der abstrakte Daten-Typ ist damit eine *Schnittstelle* zwischen dem Algorithmus einerseits und der Daten-Struktur andererseits. Dadurch ist es möglich, Algorithmus und Daten-Struktur von unterschiedlichen Personen entwickeln zu lassen.

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

```
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 4.8: Die Klasse `MyScanner`

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.

4.7 Abstrakte Daten-Typen in Setl2

Das Konzept abstrakter Daten-Typen kann in jeder Programmier-Sprache verwendet werden. Einige Programmier-Sprachen bieten zusätzliche Unterstützung für dieses Konzept. Die Sprache SETL2 unterstützt ADTs durch die Definition von *Klassen*. Wir demonstrieren die Verwendung von Klassen an einem Beispiel.

4.7.1 Stacks in Setl2

Abbildung 4.9 auf Seite 52 zeigt, wie der ADT *Stack* sich in SETL2 mit Hilfe einer Klasse implementieren läßt. Alle Schlüsselwörter der Sprache SETL2 sind in der Abbildung unterstrichen worden.

```
1   class Stack;
2       procedure create();
3       procedure push(x);
4       procedure pop();
5       procedure top();
6       procedure isEmpty();
7   end Stack;
8
9   class body Stack;
10      var list;
11
12      procedure create();
13          list := [];
14      end create;
15
16      procedure push(x);
17          list := list + [x];
18      end push;
19
20      procedure pop();
21          list := list(1 .. #list-1);
22      end pop;
23
24      procedure top();
25          return list(#list);
26      end top;
27
28      procedure isEmpty();
29          return list = [];
30      end isEmpty;
31
32      procedure selfstr();
33          return str(list);
34      end selfstr;
35  end Stack;
```

Abbildung 4.9: Implementierung eines Stacks in SETL2

1. In SETL2 besteht ein ADT aus zwei Teilen, einer *Klassen-Deklaration* und einer *Klassen-Definition*.

2. Die Klassen-Deklaration hat die Form

   ```
   class name;
       procedure create(···);
       procedure-declaration;
           ⋮
       procedure-declaration;
   end name;
   ```

   Hierbei ist *name* der Name der Klasse. Innerhalb der Klasse stehen dann die Deklarationen der verschiedenen Prozeduren, die die Klasse nach außen zur Verfügung stellt. Dabei hat die

erste Prozedur-Deklaration eine besondere Form: Die dort deklarierte Funktion hat immer den Namen `create()`. Bei dieser Funktion handelt es sich um den *Konstruktor* der Klasse.

Abbildung 4.9 zeigt in den Zeilen 1 – 7 die Deklaration der Klasse `Stack`.

3. Die *Klassen-Definition* hat die Form

   `class body` *name*`;`
   var-declaration`;`
   procedure-definition`;`
   ⋮
   procedure-definition`;`
   `end` *name*`;`

   Die einzelnen Komponenten haben die folgende Bedeutung:

   (a) *var-declaration* listet die Variablen auf, die in einem Objekt des ADT vorhanden sind. Dadurch werden die internen Daten-Strukturen, durch die Klasse realisiert wird, festgelegt.

   In Abbildung 4.9 wird in Zeile 10 festgelegt, dass ein Stack intern durch den Wert der Variablen `list` dargestellt wird.

   (b) Danach folgen die Definitionen der in der Klassen-Deklaration aufgelisteten Prozeduren. Diese Prozedur-Definitionen unterscheiden sich nicht von den Definitionen gewöhnlicher Prozeduren.

   (c) Zusätzlich können weiter Prozeduren definiert werden, die in der Klassen-Deklaration nicht genannt werden. Solche Hilfs-Prozeduren sind dann außerhalb der Klasse nicht sichtbar und können nur innerhalb der Klasse verwendet werden, also nur innerhalb der Prozedur-Definitionen in der Klassen-Definition.

   (d) Die Prozedur `selfstr()` hat eine besondere Funktion: Sie wandelt ein Objekt des ADT in einen String um. Dadurch können Objekte des ADTs dann mit `print()` ausgedruckt werden.

   In Abbildung 4.9 wird `selfstr()` mit Hilfe der eingebauten Funktion `str()` realisiert: Diese Funktion kann alle `Setl2`-Datenstrukturen in einen String umwandeln.

Werden die obige Klassen-Deklaration und die Klassen-Definition übersetzt, so wird der ADT `Stack` mit seinen Funktionen in der SETL2-Bibliothek `setl2.lib` abgespeichert. Um diesen Daten-Typ auch nutzen zu können, muß ein Programm die Direktive

```
use Stack;
```

enthalten. Abbildung 4.10 zeigt ein solches Programm mit der Direktive in Zeile 2. Ein Stack kann nun mit Hilfe der Funktion `Stack()` erzeugt werden. Ein solcher Aufruf ruft intern den *Konstruktor* der Klasse `Stack` auf. Der Konstruktor war die Prozedur, die den Namen `create()` hat. Aufgabe dieser Prozedur ist es, die internen Daten-Strukturen zu initialisieren, die den ADT repräsentieren. Im Falle des Stacks wurde einfach die Variable `list` auf die leere Liste `[]` gesetzt.

Die Prozeduren, die der Daten-Typ Stack zur Verfügung stellt, werden nun in der Form

$S\,.\,function(\cdots)$

aufgerufen. Hierbei muss S ein Objekt sein, dass mit dem Konstruktor des Daten-Typs Stack erzeugt wurde. Zeile 8 und Zeile 13 zeigen, wie die Prozeduren `push()` und `pop()` aufgerufen werden.

4.7.2 Realisierung von komplexen Zahlen in Setl2

Wir geben ein weiteres Beispiel für die Implementierung eines abstrakten Daten-Typs in SETL2, an dem wir das Konzept des *Überladens von Operatoren* demonstrieren können: Die Sprache SETL2

```
1  program main;
2      use Stack;
3
4      S := Stack();
5      print("S = ", S);
6
7      for i in {1..12} loop
8          S.push(i);
9          print("S = ", S);
10     end loop;
11
12     for i in {1..12} loop
13         S.pop();
14         print("S = ", S);
15     end loop;
16 end main;
```

Abbildung 4.10: Benutzung eines Stacks in SETL2

selbst kennt keine komplexen Zahlen. Es ist aber einfach, komplexe Zahlen als ADT in SETL2 zu realisieren. Abbildung 4.11 auf Seite 55 zeigt eine Implementierung des ADT `Complex`.

1. Die Klassen-Deklaration deklariert nur den Konstruktor. Dieser bekommt als Argumente den Real- und Imaginär-Teil der komplexen Zahl.

2. Die Klassen-Definition legt in Zeile 6 zunächst fest, dass eine komplexe Zahl intern durch die beiden Variablen `real` und `imag` repräsentiert wird, die respektive den Real- und Imaginär-Teil der Zahl enthalten. Im Konstruktor werden diese beiden Variablen mit den Argumenten `x` und `y` initialisiert.

3. Zusätzlich enthält die Klassen-Definition die Definition der Rechen-Operationen. Diese Definitionen erfolgen mit einer speziellen Syntax:

   `procedure self` *op arg*`;`

   body

   `end;`

   Hier ist *op* der Name des zu definierenden Operators, also z.B. “`"+"`” oder “`"*"`” und *arg* steht für das zweite Argument dieses Operators. *body* steht für den Rumpf der Prozedur. Innerhalb des Rumpfes kann auf Real- und Imaginär-Teil des Arguments *arg* mit Hilfe der Notation *arg*`.real` und *arg*`.imag` zurück gegriffen werden.

Das Programm in Abbildung 4.12 zeigt, dass mit Hilfe der so eingeführten Klasse `Complex` in SETL2 so mit komplexen Zahlen gerechnet werden kann, als ob diese ein Teil der Sprache SETL2 wären. In *Java* ist soetwas nicht möglich, dort können Operatoren nicht überladen werden. Im Gegensatz dazu bietet die Sprache `C++` ebenfalls die Möglichkeit, Operatoren zu überladen.

```
1  class Complex;
2      procedure create(x, y);
3  end Complex;
4
5  class body Complex;
6      var real, imag;
7
8      procedure create(x,y);
9          real := x;
10         imag := y;
11     end create;
12
13     procedure self + z;
14         return Complex(real + z.real, imag + z.imag);
15     end;
16
17     procedure self - z;
18         return Complex(real - z.real, imag - z.imag);
19     end;
20
21     procedure self * z;
22         resultReal := real * z.real - imag * z.imag;
23         resultImag := real * z.imag + imag * z.real;
24         return Complex(resultReal, resultImag);
25     end;
26
27     procedure self / z;
28         denominator := z.real ** 2 + z.imag ** 2;
29         resultReal := (real * z.real + imag * z.imag) / denominator;
30         resultImag := (imag * z.real - real * z.imag) / denominator;
31         return Complex(resultReal, resultImag);
32     end;
33
34     procedure selfstr();
35         return str(real) + " + " + str(imag) + " * i";
36     end selfstr;
37 end Complex;
```

Abbildung 4.11: Komplexe Zahlen in SETL2

```
1  program main;
2      use Complex;
3
4      z1 := Complex(1,  1);
5      z2 := Complex(1, -1);
6      print(z1, " + ", z2, " = ", z1 + z2);
7      print(z1, " - ", z2, " = ", z1 - z2);
8      print("(", z1, ") * (", z2, ") = ", z1 * z2);
9      print("(", z1, ") / (", z2, ") = ", z1 / z2);
10 end main;
```

Abbildung 4.12: Komplexe Zahlen in SETL2

Kapitel 5

Sortier-Algorithmen

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

1. $\forall x \in M : x \leq x$.
2. $\forall x, y \in M : x \leq y \wedge y \leq x \rightarrow x = y$.
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 $\{1\}$ uhnd $\{2\}$ nicht vergleichbar, es gilt
   $$\{1\} \not\subseteq \{2\} \text{ und } \{2\} \not\subseteq \{1\}.$$
3. Ist P die Menge der Mitarbeiter einer Firma und definieren wir für zwei Mitarbeiter $a, b \in M$
   $$a < b \quad \text{g.d.w.} \quad a \text{ verdient weniger als } b,$$
   so ist $\langle P, \leq \rangle$ eine partielle Ordnung.

Bemerkung: In dem letzen 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 Elemente zu sortieren, so stellen wir fest, dass dies im ersten und dritten Fall möglich ist, im zweiten Fall aber keinen Sinn macht. Offensichtlich ist eine totale Ordnung hinreichend zum Sortieren aber, wie das dritte Beispiel zeigt, nicht unbedingt notwendig. Eine partielle Ordnung reicht hingegen zum Sortieren nicht aus. Wir führen daher einen weiteren Ordnungs-Begriff ein.

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

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

Ist $\langle M, \preceq \rangle$ eine Quasi-Ordnung, so können wir auf M eine Äquivalenz-Relation $\approx$ wie folgt definieren:

$$x \approx y \overset{def}{\longleftrightarrow} x \preceq y \wedge y \preceq x.$$

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

1. Gegeben ist eine Liste L von Elementen aus M.
2. Gesucht ist eine Liste S mit folgenden Eigenschaften:
   (a) S ist aufsteigend sortiert:
   $$\forall i \in \{1, \cdots, \#S - 1\} : S(i) \preceq S(i+1)$$
   Hier bezeichnen wir die Länge der Liste S mit $\#S$.
   (b) S und L enthalten die selben Elemente, es gilt also
   $$set(L) = set(S).$$
   Dabei ist für eine Liste L der Ausdruck $set(L)$ als die Menge der Elemente der Liste L definiert, es gilt
   $$set(L) := \big\{ L(i) : i \in \{1, \cdots, \#L\} \big\}.$$
   (c) Die Elemente treten in L und S mit der selben Häufigkeit auf:
   $$\forall x : count(x, L) = count(x, S).$$
   Dabei zählt die Funktion $count(x, L)$ wie oft das Element x in der Liste L auftritt:
   $$count(x, L) = \#\big\{ i \in \{1, \cdots, \#L\} : L(i) = x \big\}.$$
   Bemerkung: Die letzte Forderung impliziert offensichtlich die zweite Forderung, denn wenn alle Elemente in den Listen L und S mit der selben Häufigkeit auftreten, dann enthalten L und S natürlich auch die selben Elemente.

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 aber etwas aufwendiger ist.

5.1 Sortieren durch Einfügen

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

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

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

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

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

   (a) Falls $x \preceq y$ ist, können wir x vorne an die Liste S anfügen:
   $$x \preceq y \rightarrow \texttt{insert}\big(x, [y] + R\big) = [x, y] + R.$$
   (b) Sonst, wenn also nicht $x \preceq y$ gilt, fügen wir x rekursiv in die Liste R ein:
   $$\neg x \preceq y \rightarrow \texttt{insert}\big(x, [y] + R\big) = [y] + \texttt{insert}(x, R).$$

Dieser Algorithmus läßt sich leicht in SETL2 umsetzen. Abbildung 5.1 zeigt das resultierende Programm.

```
1   procedure insertionSort(L);
2       if L = [] then
3           return L;
4       end if;
5       x := L(1); R := L(2..);
6       return insert(x, insertionSort(R));
7   end insertionSort;
8
9   procedure insert(x, S);
10      if S = [] then
11          return [x];
12      end if;
13      y := S(1); R := S(2..);
14      if x <= y then
15          return [x] + S;
16      else
17          return [y] + insert(x, R);
18      end if;
19  end insert;
```

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

5.1.1 Nachweis der Korrektheit

Wir zeigen, dass der Algorithmus "*Sortieren durch Einfügen*" korrekt ist. Dazu benötigen wir zunächst einige Hilfs-Funktionen, die wir über bedingte Gleichungen definieren.

Definition 14 ($\texttt{le}(x, L)$) Für ein Element $x \in M$ und eine Liste L definieren wir die Funktion $\texttt{le}(x, L)$ so, dass $\texttt{le}(x, L)$ genau dann gilt, wenn für alle Elemente y aus L die Ungleichung $x \preceq y$ gilt, es soll also gelten:

$$\forall i \in \{1, \cdots \#L\}: x \leq L(i).$$

Um mit der Funktion `le` arbeiten zu können, geben wir jetzt eine formale Definition durch Induktion über L.

1. $\texttt{le}(x, []) = \texttt{true}$
2. $x \preceq y \rightarrow \texttt{le}(x, [y] + R) = \texttt{le}(x, R)$
3. $\neg x \preceq y \rightarrow \texttt{le}(x, [y] + R) = \texttt{false}$ □

Die letzten beiden Gleichungen können wir zu einer Gleichungen zusammen fassen:

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

Die Syntax dieser Gleichung bedarf einer Erläuterung: Wir fassen hier $\preceq$ als eine zweistellige Funktion mit der Signatur

$$\texttt{le} : M \times M \to \mathbb{B}$$

auf. Die Funktion nimmt als Eingaben also zwei Elemente der Menge M und liefert als Ergebnis einen Wahrheitswert aus der Menge $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$. Genauso fassen wir den aussagenlogischen Junktor $\wedge$ auf als eine zweistellige Funktion mit der Typ-Spezifikation

$$\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}.$$

Definition 15 (isSorted(L)) Für eine Liste L definieren wir die Funktion $\texttt{isSorted}(L)$ so, dass $\texttt{isSorted}(L)$ genau dann den Wert $\texttt{true}$ hat, wenn die Liste L sortiert ist. Die formale Definition erfolgt durch Induktion über L.

1. Die leere Liste ist sicher sortiert, also gilt

   $$\texttt{isSorted}([]) = \texttt{true}.$$

2. Die Liste $[x] + R$ ist sicher dann sortiert, wenn einerseits x kleiner-gleich den Elementen aus L ist und wenn andererseits die Liste R sortiert ist.

   $$\texttt{isSorted}([x] + R) = \big(\texttt{le}(x, R) \wedge \texttt{isSorted}(R)\big).$$ □

Unser Ziel ist es zu zeigen, dass für eine Liste L immer $\texttt{isSorted}\big(\texttt{sort}(L)\big)$ gilt. Dazu benötigen wir aber noch einige Hilfssätze.

Lemma 16 (Distributivität von le über insert)
Sind x und y Elemente aus M und ist L eine Liste, so gilt:

$$\texttt{le}\big(x, \texttt{insert}(y, L)\big) \leftrightarrow x \preceq y \wedge \texttt{le}(x, L).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion nach der Definition von $\texttt{insert}$.

1. Fall: $L = []$. Dann gilt

   $$\begin{array}{lll}
   & \texttt{le}\big(x, \texttt{insert}(y, [])\big) & \\
   \leftrightarrow & \texttt{le}\big(x, [y]\big) & \text{nach Definition von } \texttt{insert} \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, []\big) & \text{nach Definition von } \texttt{le} \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, L\big) & \text{wegen } L = [].
   \end{array}$$

2. Fall: $L = [z] + R$ und $y \preceq z$. Dann gilt

   $$\begin{array}{lll}
   & \texttt{le}\big(x, \texttt{insert}(y, [z] + R)\big) & \\
   \leftrightarrow & \texttt{le}\big(x, [y, z] + R\big) & \text{nach Definition von } \texttt{insert} \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, [z] + R\big) & \text{nach Definition von } \texttt{le} \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, L\big) & \text{wegen } L = [z] + R.
   \end{array}$$

3. Fall: $L = [z] + R$ und $\neg y \preceq z$. In diesem Fall wird der Wert von $\texttt{insert}(x, L)$ durch die folgende bedingte Gleichung berechnet:

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

   Daher lautet die Induktions-Voraussetzung für diesen Fall

   $$\texttt{le}\big(x, \texttt{insert}(y, R)\big) \leftrightarrow x \preceq y \wedge \texttt{le}(x, R).$$

   Damit haben wir

   $$\begin{array}{lll}
   & \texttt{le}\big(x, \texttt{insert}(y, [z] + R)\big) & \\
   \leftrightarrow & \texttt{le}\big(x, [z] + \texttt{insert}(y, R)\big) & \text{nach Definition von } \texttt{insert} \\
   \leftrightarrow & x \preceq z \wedge \texttt{le}\big(x, \texttt{insert}(y, R)\big) & \text{nach Definition von } \texttt{le} \\
   \leftrightarrow & x \preceq z \wedge x \preceq y \wedge \texttt{le}\big(x, R\big) & \text{nach I.V.} \\
   \leftrightarrow & x \preceq y \wedge x \preceq z \wedge \texttt{le}\big(x, R\big) & \text{wegen Kommutativität von } \wedge \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, [z] + R\big) & \text{nach Definition von } \texttt{le} \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, L\big) & \text{wegen } L = [z] + R.
   \end{array}$$ □

Lemma 17 (Transitivität von `le`) Es gilt:

$$x \preceq y \wedge \texttt{le}(y, L) \rightarrow \texttt{le}(x, L).$$

Beweis: Wir führen den Beweis durch Induktion nach L.

I.A.: $L = []$. Nach Definition von `le` gilt

$$\texttt{le}(x, []) \leftrightarrow \texttt{true}.$$

I.S.: $L = [z] + R$.

Wir formen zunächst die Voraussetzung $\texttt{le}(y, L)$ um:

$$\texttt{le}(y, L) \leftrightarrow \texttt{le}(y, [z] + R) \leftrightarrow y \preceq z \wedge \texttt{le}(y, R).$$

Aus der Voraussetzung $\texttt{le}(y, L)$ folgt also

$$(y \preceq z) \leftrightarrow \texttt{true}, \tag{1}$$

$$\texttt{le}(y, R) \leftrightarrow \texttt{true}. \tag{2}$$

Aus der Voraussetzung $x \preceq y$ und (1) folgt mit der Transitivität der Relation $\preceq$

$$x \preceq z \leftrightarrow \texttt{true}. \tag{3}$$

Aus der Voraussetzung $x \preceq y$ und (2) folgt mit der Induktions-Voraussetzung

$$\texttt{le}(x, R) \leftrightarrow \texttt{true}. \tag{4}$$

Jetzt können wir die Behauptung zeigen:

$$\begin{array}{lll}
 & \texttt{le}(x, L) & \\
\leftrightarrow & \texttt{le}(x, [z] + R) & \\
\leftrightarrow & x \preceq z \wedge \texttt{le}(x, R) & \text{nach Definition von } \texttt{le} \\
\leftrightarrow & \texttt{true} \wedge \texttt{le}(x, R) & \text{wegen (3)} \\
\leftrightarrow & \texttt{true} \wedge \texttt{true} & \text{wegen (4)} \\
\leftrightarrow & \texttt{true}. & \qquad \square
\end{array}$$

Korollar 18 Es sei $L = [y] + R$. Dann gilt

$$x \preceq y \wedge \texttt{isSorted}(L) \rightarrow \texttt{le}(x, L).$$

Beweis: Wir nehmen an, dass

$$x \preceq y \quad \text{und} \quad \mathit{isSorted}(L)$$

gilt und zeigen, dass daraus

$$\texttt{le}(x, L)$$

folgt. Wir setzen in die Annahme $\mathit{isSorted}(L)$ für L den Wert $[y] + R$ ein und erhalten

$$\begin{array}{lll}
 & \texttt{isSorted}([y] + R) & \\
\leftrightarrow & \texttt{le}(y, R) \wedge \texttt{isSorted}(R) & \text{nach Definition von } \texttt{isSorted} \\
\rightarrow & \texttt{le}(y, R) & \\
\rightarrow & \texttt{le}(x, R) & \text{wegen der Transitivität von } \texttt{le}. \qquad \square
\end{array}$$

Lemma 19 (Distributivität von `isSorted` über `insert`)
Es sei $x \in M$ und S eine Liste. Dann gilt

$$\texttt{isSorted}\big(\texttt{insert}(x, S)\big) \leftrightarrow \texttt{isSorted}(S).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion nach der Definition von `insert`.

1. Fall: $S = []$. Es gilt

$$\begin{array}{lll}
 & \texttt{isSorted}\big(\texttt{insert}(x, [])\big) & \\
\leftrightarrow & \texttt{isSorted}\big([x]\big) & \text{nach Definition von } \texttt{insert} \\
\leftrightarrow & \texttt{le}(x, []) \wedge \texttt{isSorted}\big([]\big) & \text{nach Definition von } \texttt{isSorted} \\
\leftrightarrow & \texttt{true} \wedge \texttt{isSorted}\big([]\big) & \text{nach Definition von } \texttt{le} \\
\leftrightarrow & \texttt{isSorted}\big([]\big) & \\
\leftrightarrow & \texttt{isSorted}\big(S\big) & \text{wegen } S = [].
\end{array}$$

2. Fall: $S = [y] + R$ und $x \preceq y$. Dann gilt

$$\begin{array}{lll} & \texttt{isSorted}\big(\texttt{insert}(x, [y] + R)\big) & \\ \leftrightarrow & \texttt{isSorted}\big([x, y] + R\big) & \text{nach Definition von } \texttt{insert} \\ \leftrightarrow & \texttt{isSorted}\big([x] + S\big) & \text{wegen } S = [y] + R \\ \leftrightarrow & \texttt{le}(x, S) \wedge \texttt{isSorted}\big(S\big) & \text{nach Definition von } \texttt{isSorted} \\ \leftrightarrow & \texttt{true} \wedge \texttt{isSorted}\big(S\big) & \text{nach dem Korollar zur Transitivität von } \texttt{le} \\ \leftrightarrow & \texttt{isSorted}\big(S\big) & \end{array}$$

3. Fall: $S = [y] + R$ und $\neg x \preceq y$. Da wir voraussetzen, dass die Relation $\preceq$ eine TQO ist, muß $y \preceq x$ gelten. Damit haben wir:

$$\begin{array}{lll} & \texttt{isSorted}\big(\texttt{insert}(x, [y] + R)\big) & \\ \leftrightarrow & \texttt{isSorted}\big([y] + \texttt{insert}(x, R)\big) & \text{nach Definition von } \texttt{insert} \\ \leftrightarrow & \texttt{le}\big(y, \texttt{insert}(x, R)\big) \wedge \texttt{isSorted}\big(\texttt{insert}(x, R)\big) & \text{nach Definition von } \texttt{isSorted} \\ \leftrightarrow & y \preceq x \wedge \texttt{le}\big(y, R\big) \wedge \texttt{isSorted}\big(\texttt{insert}(x, R)\big) & \text{wegen der Distributivität } \texttt{le} \text{ über } \texttt{insert} \\ \leftrightarrow & y \preceq x \wedge \texttt{le}\big(y, R\big) \wedge \texttt{isSorted}\big(R\big) & \text{nach IV} \\ \leftrightarrow & \texttt{true} \wedge \texttt{le}\big(y, R\big) \wedge \texttt{isSorted}\big(R\big) & \text{wegen } y \preceq x \\ \leftrightarrow & \texttt{le}\big(y, R\big) \wedge \texttt{isSorted}\big(R\big) & \\ \leftrightarrow & \texttt{isSorted}\big([y] + R\big) & \text{nach Definition von } \texttt{isSorted} \\ \leftrightarrow & \texttt{isSorted}\big(L\big) & \text{wegen } L = [y] + R. \end{array}$$ □

Satz 20 (Korrektheit von `sort`, Teil 1)
Es sei L eine Liste. Dann gilt

$$\texttt{isSorted}\big(\texttt{sort}(L)\big).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion nach der Definition von `sort`.

I.A.: $L = []$. Dann gilt

$$\begin{array}{lll} & \texttt{isSorted}\big(\texttt{sort}([])\big) & \\ \leftrightarrow & \texttt{isSorted}\big([]\big) & \text{nach Definition von } \texttt{sort} \\ \leftrightarrow & \texttt{true} & \text{nach Definition von } \texttt{isSorted}. \end{array}$$

I.S.: $L = [x] + R$. Es gilt

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

Daher gilt nach Induktions-Voraussetzung

$$\texttt{isSorted}\big(\texttt{sort}(R)\big).$$

Also gilt

$$\begin{array}{lll} & \texttt{isSorted}\big(\texttt{sort}([x] + R)\big) & \\ \leftrightarrow & \texttt{isSorted}\Big(\texttt{insert}\big(x, \texttt{sort}(R)\big)\Big) & \text{nach Definition } \texttt{sort} \\ \leftrightarrow & \texttt{isSorted}\big(\texttt{sort}(R)\big) & \text{wegen der Distributivität von } \texttt{isSorted} \text{ über } \texttt{insert} \\ \leftrightarrow & \texttt{true} & \text{nach I.V.} \end{array}$$ □

Definition 21 ($\texttt{eq}(x, y)$) Für zwei Elemente x und y aus einer Menge M definieren wir die Funktion $\texttt{eq} : M \times M \to \mathbb{N}$ wie folgt

$$\texttt{eq}(x, y) = \begin{cases} 1 & \text{falls} \quad x = y \\ 0 & \text{falls} \quad x \neq y. \end{cases}$$

Damit können wir die Funktion

$$\texttt{count}\colon M \times \mathit{List}(M) \to \mathbb{N}$$

die zählt, wie oft ein Element x in einer Liste L vorkommt, durch Gleichungen definieren:

1. Fall: $L = []$.
   $\texttt{count}(x, []) = 0$.
2. Fall $L = [y] + R$.
   $\texttt{count}(x, [y] + R) = \texttt{eq}(x, y) + \texttt{count}(x, R)$.

Lemma 22 (Distributivität von `count` über `insert`)
Es seien x und y Elemente der Menge M und S sei eine Liste von Elementen aus M. Dann gilt

$$\texttt{count}\big(x, \texttt{insert}(y, S)\big) = \texttt{eq}(x, y) + \texttt{count}(x, S).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion.

1. Fall: $S = []$. Es gilt

$$\begin{array}{lll} & \texttt{count}\big(x, \texttt{insert}(y, S)\big) & \\ = & \texttt{count}\big(x, \texttt{insert}(y, [])\big) & \\ = & \texttt{count}\big(x, [y]\big) & \\ = & \texttt{eq}(x, y) + \texttt{count}\big(x, []\big) & \\ = & \texttt{eq}(x, y) + \texttt{count}(x, S). & \end{array}$$

2. Fall: $S = [z] + R$ und $y \preceq z$. Dann gilt:

$$\begin{array}{lll} & \texttt{count}\big(x, \texttt{insert}(y, S)\big) & \\ = & \texttt{count}\big(x, \texttt{insert}(y, [z] + R)\big) & \\ = & \texttt{count}\big(x, [y, z] + R)\big) & \text{nach Definition von } \texttt{insert} \\ = & \texttt{eq}(x, y) + \texttt{count}\big(x, [z] + R)\big) & \text{nach Definition von } \texttt{count} \\ = & \texttt{eq}(x, y) + \texttt{count}(x, S). & \end{array}$$

3. Fall: $S = [z] + R$ und $\neg y \preceq z$. Dann gilt:

$$\begin{array}{lll} & \texttt{count}\big(x, \texttt{insert}(y, S)\big) & \\ = & \texttt{count}\big(x, \texttt{insert}(y, [z] + R)\big) & \\ = & \texttt{count}\big(x, [z] + \texttt{insert}(y, R)\big) & \text{nach Definition von } \texttt{insert} \\ = & \texttt{eq}(x, z) + \texttt{count}\big(x, \texttt{insert}(y, R)\big) & \text{nach Definition von } \texttt{count} \\ = & \texttt{eq}(x, z) + \texttt{eq}(x, y) + \texttt{count}(x, R) & \text{nach Induktions-Voraussetzung} \\ = & \texttt{eq}(x, y) + \texttt{eq}(x, z) + \texttt{count}(x, R) & \\ = & \texttt{eq}(x, y) + \texttt{count}(x, [z] + R) & \text{nach Definition von } \texttt{count} \\ = & \texttt{eq}(x, y) + \texttt{count}(x, S). & \end{array}$$

□

Satz 23 (Distributivität von `count` über `sort`)
Ist x ein Element und L eine Liste, so gilt

$$\texttt{count}\big(x, \texttt{sort}(L)\big) = \texttt{count}(x, L).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion nach der Definition von `sort`.

1. Fall: $L = []$. Es gilt

$$\begin{array}{lll} & \texttt{count}\big(x, \texttt{sort}(L)\big) & \\ = & \texttt{count}\big(x, \texttt{sort}([])\big) & \\ = & \texttt{count}\big(x, []\big) & \\ = & \texttt{count}\big(x, L\big). & \end{array}$$

2. Fall: $L = [y] + R$.

$$\begin{array}{lll} & \texttt{count}\big(x, \texttt{sort}(L)\big) & \\ = & \texttt{count}\big(x, \texttt{sort}([y] + R)\big) & \\ = & \texttt{count}\big(x, \texttt{insert}(y, \texttt{sort}(R))\big) & \\ = & \texttt{eq}(x, y) + \texttt{count}(x, \texttt{sort}(R)) & \text{wegen der Distributivität von } \texttt{count} \text{ über } \texttt{insert} \\ = & \texttt{eq}(x, y) + \texttt{count}(x, R) & \text{nach Induktions-Voraussetzung} \\ = & \texttt{count}(x, [y] + R) & \text{nach Definition von } \texttt{count} \\ = & \texttt{count}(x, L) & \text{wegen } L = [y] + R. \end{array}$$

□

Damit haben wir nun auch den zweiten Teil der Korrektheit von "*Sortieren durch Einfügen*" nachgewiesen.

5.1.2 Komplexität

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die bei einem Aufruf von "`Sortieren durch Einfügen`" in Zeile 14 von Abbildung 5.1 auf Seite 59 durchgeführt werden. Dazu berechnen wir zunächst die Anzahl der Aufrufe von "`<=`", die bei einem Aufruf von `insert`(x, L) im schlimmsten Fall bei einer Liste der Länge n durchgeführt werden. Wir bezeichnen diese Anzahl mit a_n. Dann haben wir

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

Dies ist eine lineare inhomogene Rekurrenz-Gleichungen erster Ordnung mit der Inhomogenität $c_{-1} = 1$. Das charakteristische Polynom lautet

$$\chi(x) = x - 1$$

und offenbar ist $\mathtt{sp}(\chi) = 0$. Also hat die spezielle Lösung die Form

$$a_n = \varepsilon * n \quad \text{mit } \varepsilon = \frac{c_{-1}}{\chi'(1)} = \frac{1}{1} = 1,$$

die spezielle Lösung ist also

$$a_n = n.$$

Damit lautet die allgemeine Lösung:

$$a_n = \alpha * 1^n + n.$$

Durch Einsetzen der Anfangs-Bedingungen erhalten wir die Gleichung

$$0 = \alpha * 1^n + 1 * 0, \quad \text{also } \alpha = 0.$$

Damit lautet die Lösung

$$a_n = n,$$

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 dieses Anzahl mit b_n. Offenbar gilt

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

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

Die Rekurrenz-Gleichungen (1) ist eine lineare inhomogene Rekurrenz-Gleichungen erster Ordnung mit einer nicht-konstanten Inhomogenität. Um diese Rekurrenz-Gleichungen zu lösen, verwenden wir die Methode des diskreten Differenzierens. Dazu führt wir in der Gleichung (1) die Substitution $n \mapsto n + 1$ durch:

$$b_{n+2} = b_{n+1} + n + 1, \tag{2}$$

Wir subtrahieren nun die Gleichung (1) von der Gleichung (2) ab und erhalten nach Vereinfachung

$$b_{n+2} = 2 * b_{n+1} - b_n + 1.$$

Dies ist eine lineare inhomogene Rekurrenz-Gleichungen zweiter Ordnung mit konstanter Inhomogenität. Für das charakteristische Polynom gilt $\chi(x) = (x-1)^2$. Offenbar hat dieses Polynom eine doppelte Null-Stelle an der Stelle $x = 1$. Um eine spezielle Lösung zu erhalten, machen wir also den Ansatz

$$b_n = \varepsilon * n^2.$$

Einsetzen in die Gleichung (2) ergibt:

$$\varepsilon * (n+2)^2 = 2 * \varepsilon * (n+1)^2 - \varepsilon * n^2 + 1.$$

Diese Gleichung vereinfacht sich zu

$$4 * \varepsilon = 2 * \varepsilon + 1$$

und daraus folgt sofort $\varepsilon = \frac{1}{2}$. Also ist

$$a_n = \frac{1}{2}n^2$$

eine spezielle Lösung der Rekurrenz-Gleichungen (2). Daher ergibt sich die allgemeine Lösung als

$$b_n = \alpha * 1^n + \beta * n * 1^n + \frac{1}{2} * n^2.$$

Wir finden α und β indem wir die Anfangs-Bedingungen $a_0 = 0$ und $a_1 = a_0 + 0 = 0$ hier einsetzen. Das liefert die beiden Gleichungen

$$0 = \alpha \quad \text{und} \quad 0 = \alpha + \beta + \frac{1}{2}, \quad \text{also} \quad \alpha = 0 \quad \text{und} \quad \beta = -\tfrac{1}{2}.$$

Damit lautet die Lösung unserer Rekurrenz-Gleichungen (1)

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

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

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

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

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

5.2 Sortieren durch Auswahl

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

1. Ist die zu sortierende Liste L leer, so wird als Ergebnis die leere Liste zurück gegeben:
   $$\texttt{sort}([]) = []$$
2. Andernfalls suchen wir in der Liste L das kleinste Element und entfernen dieses Element aus L. Wir sortieren rekursiv die resultierende Liste, die ja ein Element weniger enthält. Zum Schluß fügen wir das kleinste Element vorne an die sortierte Liste an:
   $$L \neq [] \rightarrow \texttt{sort}\big(L\big) = \big[\texttt{min}(L)\big] + \texttt{sort}\big(\texttt{delete}(\texttt{min}(L), L)\big).$$

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

1. Falls L leer ist, gilt
   $$\texttt{delete}(x, []) = [].$$
2. Falls x gleich dem ersten Element der Liste L ist, gibt die Funktion den Rest R zurück:
   $$\texttt{delete}(x, [x] + R) = R.$$
3. Andernfalls wird das Element x rekursiv aus R entfernt:
   $$x \neq y \rightarrow \texttt{delete}(x, [y] + R) = [y] + \texttt{delete}(x, R).$$

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

1. Das Minimum der leeren Liste ist größer als alles andere
   $$\texttt{min}([]) = \infty.$$
2. Um das Minimum der Liste $[x] + R$ zu berechnen, berechnen wir rekursiv das Minimum von R und benutzen die zweistellige Minimums-Funktion:
   $$\texttt{min}([x] + R) = \texttt{min}\big(x, \texttt{min}(R)\big).$$
   Dabei ist die zweistellige Minimums-Funktion wie folgt definiert:
   $$\texttt{min}(x, y) = \begin{cases} x & \text{falls } x \preceq y\,; \\ y & \text{sonst.} \end{cases}$$

Die Implementierung dieses Algorithmus in SETL2 sehen Sie in Abbildung 5.2 auf Seite 66.

```
1   procedure minSort(L);
2       if L = [] then
3           return L;
4       end if;
5       minValue := minList(L);
6       return [ minValue ] + minSort(delete(minValue, L));
7   end minSort;
8
9   procedure delete(x, L);
10      y := L(1); R := L(2..);
11      if x = y then
12          return R;
13      else
14          return [y] + delete(x, R);
15      end if;
16  end delete;
17
18  procedure minList(L);
19      if L = [] then
20          return;     -- return Omega
21      end if;
22      x := L(1); R := L(2..);
23      return minPair(x, minList(R));
24  end minList;
25
26  procedure minPair(x, y);
27      if x = om then
28          return y;
29      elseif y = om then
30          return x;
31      elseif x <= y then
32          return x;
33      else
34          return y;
35      end if;
36  end minPair;
```

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

5.2.1 Nachweis der Korrektheit

Um den Nachweis der Korrektheit erbringen zu können, brauchen wir eine Reihe von Hilfssätzen.

Lemma 24 (Verträglichkeit von `min` und `le`) Für $x \in M$ und eine Liste L gilt:

$$\texttt{le}(x, L) \leftrightarrow x \preceq \texttt{min}(L).$$

Beweis: Wir führen den Beweis durch Induktion über die Liste L.

1. Fall: $L = []$. Es gilt:

$$\begin{array}{ll} & \texttt{le}(x, L) \\ \leftrightarrow & \texttt{le}(x, []) \\ \leftrightarrow & \texttt{true} \\ \leftrightarrow & x \preceq \infty \\ \leftrightarrow & x \preceq \texttt{min}([]) \\ \leftrightarrow & x \preceq \texttt{min}(L). \end{array}$$

2. Fall: $L = [y] + R$. Es gilt:

$$\begin{array}{lll} & \texttt{le}(x, L) & \\ \leftrightarrow & \texttt{le}(x, [y] + R) & \\ \leftrightarrow & x \preceq y \wedge \texttt{le}(x, R) & \text{nach Definition von } \texttt{le} \\ \leftrightarrow & x \preceq y \wedge x \preceq \texttt{min}(R) & \text{nach Induktions-Voraussetzung} \\ \leftrightarrow & x \preceq \texttt{min}\big(y, \texttt{min}(R)\big) & \text{wegen } c \preceq \texttt{min}(a, b) \leftrightarrow c \preceq a \wedge c \preceq b \\ \leftrightarrow & x \preceq \texttt{min}\big([y] + R\big) & \text{nach Definition von } \texttt{min} \\ \leftrightarrow & x \preceq \texttt{min}\big(L\big). & \end{array}$$

□

Korollar 25 Für jede Liste L gilt: $\texttt{le}\big(\texttt{min}(L), L\big)$.

Beweis: Setzen wir in dem letzten Lemma für x den Wert $\texttt{min}(L)$ ein, so erhalten wir:

$$\texttt{le}\big(\texttt{min}(L), L\big) \leftrightarrow \texttt{min}(L) \preceq \texttt{min}(L) \leftrightarrow \texttt{true}.$$

□

Lemma 26 (Distributivität von `le` über `delete`)
Sind $x, y \in M$ und es gelte $x \preceq y$. Dann gilt für jede Liste L:

$$\texttt{le}\big(x, \texttt{delete}(y, L)\big) \leftrightarrow \texttt{le}(x, L).$$

Beweis durch Wertverlaufs-Induktion nach der Definition von $\texttt{delete}(x, L)$.

1. Fall: $L = []$. Es gilt

$$\texttt{le}\big(x, \texttt{delete}(y, L)\big) \leftrightarrow \texttt{le}\big(x, \texttt{delete}(y, [])\big) \leftrightarrow \texttt{le}(x, []) \leftrightarrow \texttt{le}(x, L).$$

2. Fall: $L = [y] + R$. Es gilt

$$\begin{array}{lll} & \texttt{le}\big(x, \texttt{delete}(y, L)\big) & \\ \leftrightarrow & \texttt{le}\big(x, \texttt{delete}(y, [y] + R)\big) & \\ \leftrightarrow & \texttt{le}\big(x, R\big) & \\ \leftrightarrow & \texttt{true} \wedge \texttt{le}\big(x, R\big) & \\ \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, R\big) & \text{wegen } x \preceq y \\ \leftrightarrow & \texttt{le}\big(x, [y] + R\big) & \text{nach Definition von } \texttt{le} \\ \leftrightarrow & \texttt{le}\big(x, L\big). & \end{array}$$

3. Fall: $L = [z] + R$ mit $y \neq z$. Es gilt:

$$\begin{array}{lll} & \texttt{le}\big(x, \texttt{delete}(y, L)\big) & \\ \leftrightarrow & \texttt{le}\big(x, \texttt{delete}(y, [z] + R)\big) & \\ \leftrightarrow & \texttt{le}\big(x, [z] + \texttt{delete}(y, R)\big) & \text{nach Definition von } \texttt{delete} \\ \leftrightarrow & x \preceq z \wedge \texttt{le}\big(x, \texttt{delete}(y, R)\big) & \text{nach Definition von } \texttt{le} \\ \leftrightarrow & x \preceq z \wedge \texttt{le}\big(x, R\big) & \text{nach Induktions-Voraussetzung} \\ \leftrightarrow & \texttt{le}\big(x, [z] + R\big) & \\ \leftrightarrow & \texttt{le}\big(x, L\big). & \end{array}$$

□

Lemma 27 (Distributivität von `le` über `sort`) Ist $x \in M$ und L eine Liste, so gilt

$$\texttt{le}\big(x, \texttt{sort}(L)\big) \leftrightarrow \texttt{le}(x, L).$$

Beweis durch Wertverlaufs-Induktion nach der Definition von $\texttt{sort}(L)$:

1. Fall: $L = []$. Dann gilt:

   $$\texttt{le}\big(x, \texttt{sort}(L)\big) \leftrightarrow \texttt{le}\big(x, \texttt{sort}([])\big) \leftrightarrow \texttt{le}\big(x, []\big) \leftrightarrow \texttt{le}\big(x, L\big).$$

2. Fall: $L \neq []$. Zur Abkürzung setzen wir $m = \texttt{min}(L)$.

$$\begin{array}{lll}
 & \texttt{le}\big(x, \texttt{sort}(L)\big) & \\
\leftrightarrow & \texttt{le}\Big(x, [m] + \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \\
\leftrightarrow & x \preceq m \wedge \texttt{le}\Big(x, \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \text{nach Definition von } \texttt{le} \\
\leftrightarrow & x \preceq m \wedge \texttt{le}\Big(x, \texttt{delete}(m, L)\Big) & \text{nach Induktions-Voraussetzung} \\
\leftrightarrow & x \preceq m \wedge \texttt{le}\big(x, L\big) & \text{denn unter der Voraussetzung } x \preceq m \text{ gilt nach} \\
 & & \text{Lemma 26: } \texttt{le}\Big(x, \texttt{delete}(m, L)\Big) \leftrightarrow \texttt{le}\big(x, L\big) \\
\leftrightarrow & x \preceq \texttt{min}(L) \wedge \texttt{le}\big(x, L\big) & \text{wegen } m = \texttt{min}(L) \\
\leftrightarrow & \texttt{le}\big(x, L\big) \wedge \texttt{le}\big(x, L\big) & \text{denn nach Lemma 24 gilt: } x \preceq \texttt{min}(L) \leftrightarrow \texttt{le}(x, L) \\
\leftrightarrow & \texttt{le}\big(x, L\big). & \qquad \square
\end{array}$$

Satz 28 (Korrektheit von *Sortieren durch Auswahl*, 1. Teil) Für beliebige Listen L gilt:

$$\texttt{isSorted}\big(\texttt{sort}(L)\big).$$

Beweis durch Wertverlaufs-Induktion nach der Definition der Funktion `sort`:

1. Fall: $L = []$. Es gilt:

   $$\texttt{isSorted}\big(\texttt{sort}(L)\big) \leftrightarrow \texttt{isSorted}\big(\texttt{sort}([])\big) \leftrightarrow \texttt{isSorted}\big([]\big) \leftrightarrow \texttt{true}.$$

2. Fall: $L \neq []$. Zur Abkürzung setzen wir $m = \texttt{min}(L)$. Es gilt:

$$\begin{array}{lll}
 & \texttt{isSorted}\big(\texttt{sort}(L)\big) & \\
\leftrightarrow & \texttt{isSorted}\Big([m] + \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \text{nach Definition von } \texttt{sort} \\
\leftrightarrow & \texttt{le}\Big(m, \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) \wedge & \\
 & \texttt{isSorted}\Big(\texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \text{nach Definition von } \texttt{isSorted} \\
\leftrightarrow & \texttt{le}\Big(m, \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) \wedge \texttt{true} & \text{nach Induktions-Voraussetzung} \\
\leftrightarrow & \texttt{le}\Big(m, \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \\
\leftrightarrow & \texttt{le}\big(m, \texttt{delete}(m, L)\big) & \text{nach Lemma 27} \\
\leftrightarrow & \texttt{le}\big(m, L\big) & \text{wegen } m \preceq m \text{ nach Lemma 26} \\
\leftrightarrow & \texttt{le}\big(\texttt{min}(L), L\big) & \text{wegen } m = \texttt{min}(L) \\
\leftrightarrow & \texttt{true}. & \text{nach Korollar 25} \qquad \square
\end{array}$$

Um das nächste Lemma elegant formulieren zu können, benötigen wir noch die Hilfsfunktion

$$\texttt{member} : M \times \texttt{List}(M) \to \{0, 1\},$$

die überprüft, ob ein gegebenes Element in einer Liste auftritt. Der Aufruf $\texttt{member}(x, L)$ liefert genau dann 1, wenn x in der Liste L auftritt. Rekursiv können wir die Funktion `member` wie folgt definieren:

1. $\texttt{member}(x, []) = 0$.
2. $\texttt{member}(x, [x] + R) = 1$.

3. Falls $x \neq y$ ist, setzen wir $\texttt{member}(x, [y] + R) = \texttt{member}(x, R)$.

Aufgabe: Beweisen Sie die beiden folgenden Lemmata.

Lemma 29 (Distributivität von `count` über `delete`)
Für beliebige Elemente x und y und für beliebige Listen L gilt:

$$\texttt{count}\big(x, \texttt{delete}(y, L)\big) = \texttt{count}(x, L) - \texttt{eq}(x, y) * \texttt{member}(y, L).$$

Wir führen den **Beweis** durch Wertverlaufs-Induktion nach der Definition der Funktion `delete`:

1. Fall: $L = []$

$$\begin{array}{ll} & \texttt{count}\big(x, \texttt{delete}(y, L)\big) \\ = & \texttt{count}\big(x, \texttt{delete}(y, [])\big) \\ = & \texttt{count}\big(x, []\big) \\ = & \texttt{count}\big(x, []\big) - \texttt{eq}(x, y) * 0 \\ = & \texttt{count}\big(x, []\big) - \texttt{eq}(x, y) * \texttt{member}(y, []) \\ = & \texttt{count}\big(x, L\big) - \texttt{eq}(x, y) * \texttt{member}(y, L). \end{array}$$

2. Fall: $L = [y] + R$

$$\begin{array}{ll} & \texttt{count}\big(x, \texttt{delete}(y, L)\big) \\ = & \texttt{count}\big(x, \texttt{delete}(y, [y] + R)\big) \\ = & \texttt{count}\big(x, R\big) \\ = & \texttt{eq}(x, y) + \texttt{count}\big(x, R\big) - \texttt{eq}(x, y) * 1 \\ = & \texttt{eq}(x, y) + \texttt{count}\big(x, R\big) - \texttt{eq}(x, y) * \texttt{member}(y, [y] + R) \\ = & \texttt{count}\big(x, [y] + R\big) - \texttt{eq}(x, y) * \texttt{member}(y, [y] + R) \\ = & \texttt{count}\big(x, L\big) - \texttt{eq}(x, y) * \texttt{member}(y, L). \end{array}$$

3. Fall: $L = [z] + R$ mit $y \neq z$

$$\begin{array}{lll} & \texttt{count}\big(x, \texttt{delete}(y, L)\big) & \\ = & \texttt{count}\big(x, \texttt{delete}(y, [z] + R)\big) & \\ = & \texttt{count}\big(x, [z] + \texttt{delete}(y, R)\big) & \\ = & \texttt{eq}(x, z) + \texttt{count}\big(x, \texttt{delete}(y, R)\big) & \\ = & \texttt{eq}(x, z) + \texttt{count}\big(x, R\big) - \texttt{eq}(x, y) * \texttt{member}(y, R) & \text{nach Induktions-Voraussetzung} \\ = & \texttt{eq}(x, z) + \texttt{count}\big(x, R\big) - \texttt{eq}(x, y) * \texttt{member}(y, [z] + R) & \text{nach Definition von } \texttt{member} \\ = & \texttt{count}\big(x, [z] + R\big) - \texttt{eq}(x, y) * \texttt{member}(y, [z] + R) & \text{nach Definition von } \texttt{count} \\ = & \texttt{count}\big(x, L\big) - \texttt{eq}(x, y) * \texttt{member}(y, L). & \end{array}$$

□

Lemma 30 Für jede nicht-leere Liste L gilt:

$$\texttt{member}\big(\texttt{min}(L), L\big) = 1.$$

Beweis durch Wertverlaufs-Induktion nach der Definition der Funktion `min`.

1. $L = [x]$. Es gilt

$$\texttt{member}\big(\texttt{min}(L), L\big) = \texttt{member}\big(\texttt{min}([x]), [x]\big) = \texttt{member}\big(x, [x]\big) = 1$$

2. Sei $L = [x] + R$. Wir führen eine Fall-Unterscheidung nach der Größe von x durch:

   (a) $x \preceq \texttt{min}(R)$.

$$\begin{array}{lll} & \texttt{member}\big(\texttt{min}(L), L\big) & \\ = & \texttt{member}\big(\texttt{min}([x] + R), [x] + R\big) & \\ = & \texttt{member}\big(\texttt{min}(x, \texttt{min}(R)), [x] + R\big) & \\ = & \texttt{member}\big(x, [x] + R\big) & \text{wegen } x \preceq \texttt{min}(R) \\ = & 1 & \text{nach Definition von } \texttt{member} \end{array}$$

(b) $\neg x \preceq \texttt{min}(R)$.

$$
\begin{array}{lll}
 & \texttt{member}\big(\texttt{min}(L), L\big) & \\
= & \texttt{member}\big(\texttt{min}([x]+R), [x]+R\big) & \\
= & \texttt{member}\big(\texttt{min}(x, \texttt{min}(R)), [x]+R\big) & \\
= & \texttt{member}\big(\texttt{min}(R), [x]+R\big) & \text{nach Definition von } \texttt{min} \\
= & \texttt{member}\big(\texttt{min}(R), R\big) & \text{nach Definition von } \texttt{member} \\
 & & \text{denn es muß } \texttt{min}(R) \neq x \text{ gelten} \\
= & 1. & \text{nach Induktions-Voraussetzung} \qquad \square
\end{array}
$$

Satz 31 (Distributivität von `count` über `sort`)
Es sei x ein Element und L sei eine Liste. Dann gilt

$$\texttt{count}\big(x, \texttt{sort}(L)\big) = \texttt{count}(x, L).$$

Beweis durch Wertverlaufs-Induktion nach der Definition von `sort`.

1. Fall: $L = []$. Nach Definition von `sort` gilt
$$\texttt{count}\big(x, \texttt{sort}(L)\big) = \texttt{count}\big(x, \texttt{sort}([])\big) = \texttt{count}\big(x, []\big) = \texttt{count}\big(x, L\big).$$
2. Fall: $L \neq []$. Es sei $m = \texttt{min}(L)$. Dann gilt:

$$
\begin{array}{lll}
 & \texttt{count}\big(x, \texttt{sort}(L)\big) & \\
= & \texttt{count}\Big(x, [m] + \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \text{nach Definition von } \texttt{sort} \\
= & \texttt{eq}(x, m) + \texttt{count}\Big(x, \texttt{sort}\big(\texttt{delete}(m, L)\big)\Big) & \text{nach Definition von } \texttt{count} \\
= & \texttt{eq}(x, m) + \texttt{count}\big(x, \texttt{delete}(m, L)\big) & \text{nach Induktions-Voraussetzung} \\
= & \texttt{eq}(x, m) + \texttt{count}\big(x, L\big) - \texttt{eq}(x, m) * \texttt{member}(m, L) & \text{Distributivität von } \texttt{count} \text{ über } \texttt{delete} \\
= & \texttt{eq}(x, m) + \texttt{count}\big(x, L\big) - \texttt{eq}(x, m) & \text{denn } \texttt{member}(m, L) = 1 \\
= & \texttt{count}\big(x, L\big). & \qquad \square
\end{array}
$$

5.2.2 Komplexität

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

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

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

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

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

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

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

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*".

5.2.3 Implementierung von *Sortieren durch Auswahl* in *Java*

Zum Abschluß unserer Diskussion des Algorithmus "*Sortieren durch Auswahl*" präsentieren wir noch eine Implementierung des Algorithmus in der Sprache *Java*. Abbildung 5.4 auf Seite 72 zeigt diese Implementierung. Die Klasse `MinSortAlgorithm` implementiert die Schnittstelle

`SortingAlgorithm`,

die in Abbildung 5.3 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 5.3: Das Interface "`SortingAlgorithm`"

In Zeile 3 der Implementierung der Klass `MinSortAlgorithm` (Abbildung 5.4) 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

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

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.

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 $i <$ `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

$$\big[\ \texttt{mArray}[i],\ \texttt{mArray}[i+1],\ \cdots,\ \texttt{mArray}[\texttt{mArray.length} - 1]\ \big]$$

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

1. Falls $i =$ `mArray.length` ist, dann ist der zu sortierende Teil des Feldes leer und folglich ist nichts zu tun.

2. Andernfalls berechnet der Aufruf `minIndex`(i) in Zeile 15 einen Index `minIndex` so, dass das Element `mArray[minIndex]` in dem zu sortierenden Teil der Liste minimal ist, es gilt also

   $$\forall j \in \big\{\, i, i+1, \cdots, \texttt{mArray.length} - 1 \,\big\}\colon \texttt{mArray}[\texttt{minIndex}] \leq \texttt{mArray}[i].$$

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

   $$\big[\ \texttt{mArray}[i],\ \texttt{mArray}[i+1],\ \cdots,\ \texttt{mArray}[\texttt{mArray.length} - 1]\ \big].$$

```
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 5.4: Der Algorithmus "*Sortieren durch Auswahl*"

Sortieren wir danach rekursiv die Liste

$$\big[\, \texttt{mArray}[i+1],\ \cdots,\ \texttt{mArray}[\texttt{mArray.length}-1] \,\big],$$

so ist anschließend auch die Liste

$$\big[\, \texttt{mArray}[i],\ \texttt{mArray}[i+1],\ \cdots,\ \texttt{mArray}[\texttt{mArray.length}-1] \,\big]$$

sortiert, denn $\texttt{mArray}[i]$ 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 $\texttt{mArray}[i]$ kleiner als das Element $\texttt{mArray}[\mathit{minIndex}]$ ist, dann wird `minIndex` auf i gesetzt. Dadurch ist gewährleistet dass `minIndex` am Ende der Schleife tatsächlich auf das kleinste Element zeigt.

5.3 Sortieren durch Mischen

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

1. Hat die zu sortierende Liste L weniger als zwei Elemente, so wird L zurück gegeben:
   $$\#L < 2 \rightarrow \texttt{sort}(L) = L.$$

2. Ansonsten wird die Liste in zwei etwa gleich große Listen zerlegt. Diese Listen werden rekursiv sortiert und anschließend so gemischt, dass das Ergebnis sortiert ist:
   $$\#L \geq 2 \rightarrow \texttt{sort}(L) = \texttt{merge}\Big(\texttt{sort}\big(\texttt{split}_1(L)\big), \texttt{sort}\big(\texttt{split}_2(L)\big)\Big)$$
   Hier bezeichnen $\texttt{split}_1$ und $\texttt{split}_2$ die Funktionen, die eine Liste in zwei Teil-Listen zerlegen und `merge` ist eine Funktion, die zwei sortierte Listen so mischt, dass das Ergebnis wieder sortiert ist.

Abbildung 5.5 zeigt die Umsetzung dieser sortierten Gleichungen in ein SETL2-Programm. Die beiden Funktionen $\texttt{split}_1$ und $\texttt{split}_2$ haben wir dabei zu einer Funktion zusammen gefaßt, die ein Paar von Ergebnissen produziert, es gilt:

$$\texttt{split}(L) = [\,\texttt{split}_1(L), \texttt{split}_2(L)\,].$$

```
1  procedure sort(L);
2      if #L < 2 then
3          return L;
4      end if;
5      [ L1, L2 ] := split(L);
6      S1 := sort(L1);
7      S2 := sort(L2);
8      return merge(S1, S2);
9  end sort;
```

Abbildung 5.5: Der Algorithmus "*Sortieren durch Mischen*"

Wir überlegen uns jetzt, wie wir die Funktion `split` implementieren können.

1. Falls L leer ist, produziert $\texttt{split}(L)$ zwei leere Listen:
   $$\texttt{split}(\texttt{[]}) = [\texttt{[]}, \texttt{[]}].$$

2. Falls L genau ein Element besitzt, stecken wir dieses in die erste Liste:
   $$\texttt{split}([x]) = [\texttt{[}x\texttt{]}, \texttt{[]}].$$

3. Sonst hat L die Form $[x, y] + R$. Dann spalten wir rekursiv R in zwei Listen auf. Vor die erste Liste fügen wir x an, vor die zweite Liste fügen wir y an:
   $$\texttt{split}(R) = [R_1, R_2] \rightarrow \texttt{split}\big([x, y] + R\big) = \big[[x] + R_1, [y] + R_2\big].$$

Abbildung 5.6 auf Seite 74 zeigt die Umsetzung dieser bedingten Gleichungen in SETL2.

Als letztes geben wir noch den Algorithmus an, um zwei sortierte Listen L_1 und L_2 so zu mischen, dass das Ergebnis wieder sortiert ist.

1. Falls die Liste L_1 leer ist, ist das Ergebnis L_2:
   $$\texttt{merge}([], L_2) = L_2.$$

2. Falls die Liste L_2 leer ist, ist das Ergebnis L_1:
   $$\texttt{merge}(L_1, []) = L_1.$$

```
1   procedure split(L);
2       if L = [] then
3           return [ [], [] ];
4       end if;
5       if #L = 1 then
6           return [ [ L(1) ], [] ];
7       end if;
8       x := L(1); y := L(2); R := L(3..);
9       [ R1, R2 ] := split(R);
10      return [ [x] + R1, [y] + R2 ];
11  end split;
```

Abbildung 5.6: Die Prozedur `split`.

3. Andernfalls hat L_1 die Form $[x] + R_1$ und L_2 hat die Gestalt $[y] + R_2$. Dann führen wir eine Fallunterscheidung nach der relativen Größe von x und y durch:

   (a) $x \preceq y$.

   In diesem Fall mischen wir R_1 und L_2 und setzen x an den Anfang dieser Liste:

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

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

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

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

Abbildung 5.7 auf Seite 74 zeigt die Umsetzung dieses Algorithmus in SETL2.

```
1   procedure merge(L1, L2);
2       if L1 = [] then
3           return L2;
4       end if;
5       if L2 = [] then
6           return L1;
7       end if;
8       x1 := L1(1);  R1 := L1(2..);
9       x2 := L2(1);  R2 := L2(2..);
10      if x1 <= x2 then
11          return [x1] + merge(R1, L2);
12      else
13          return [x2] + merge(L1, R2);
14      end if;
15  end merge;
```

Abbildung 5.7: Die Prozedur `merge`.

5.3.1 Korrektheit

Um die Korrektheit des Algorithmus "*Sortieren durch Mischen*" zu zeigen, benötigen wir zunächst einige Hilfs-Sätze, die das Verhalten der Funktionen `split` und `merge` beschreiben.

Lemma 32 (Distributivität von `le` über `merge`)
Für beliebige Listen L_1 und L_2 gilt:

$$\texttt{le}\big(x, \texttt{merge}(L_1, L_2)\big) \leftrightarrow \texttt{le}(x, L_1) \wedge \texttt{le}(x, L_2).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion entsprechend der Definition der Funktion `merge`.

1. Fall: $L_1 = []$. Wegen $\texttt{merge}([], L_2) = L_2$ gilt einerseits
   $$\texttt{le}\big(x, \texttt{merge}(L_1, L_2)\big) \leftrightarrow \texttt{le}\big(x, \texttt{merge}([], L_2)\big) \leftrightarrow \texttt{le}(x, L_2)$$
   und andererseits haben wir wegen $\texttt{le}(x, []) \leftrightarrow \texttt{true}$
   $$\texttt{le}(x, L_1) \wedge \texttt{le}(x, L_2) \leftrightarrow \texttt{le}(x, []) \wedge \texttt{le}(x, L_2) \leftrightarrow \texttt{true} \wedge \texttt{le}(x, L_2) \leftrightarrow \texttt{le}(x, L_2).$$

2. Fall: $L_2 = []$. Der Beweis ist analog zum 1. Fall.

3. Fall: $L_1 = [y] + R_1$, $L_2 = [z] + R_2$, $y \preceq z$. Es gilt:

   $$\begin{array}{lll}
   & \texttt{le}\big(x, \texttt{merge}(L_1, L_2)\big) & \\
   \leftrightarrow & \texttt{le}\big(x, \texttt{merge}([y] + R_1, [z] + R_2)\big) & \\
   \leftrightarrow & \texttt{le}\big(x, [y] + \texttt{merge}(R_1, [z] + R_2)\big) & \text{wegen } y \preceq z \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}\big(x, \texttt{merge}(R_1, [z] + R_2)\big) & \text{nach Definition von } \texttt{le} \\
   \leftrightarrow & x \preceq y \wedge \texttt{le}(x, R_1) \wedge \texttt{le}(x, [z] + R_2)\big) & \text{nach Induktions-Voraussetzung} \\
   \leftrightarrow & \texttt{le}(x, [y] + R_1) \wedge \texttt{le}(x, [z] + R_2)\big) & \text{nach Definition von } \texttt{le} \\
   \leftrightarrow & \texttt{le}(x, L_1) \wedge \texttt{le}(x, L_2)\big) &
   \end{array}$$

4. Fall: $L_1 = [y] + R_1$, $L_2 = [z] + R_2$, $\neg y \preceq z$.
   Der Beweis verläuft analog zum 3. Fall. □

Lemma 33 (Verträglichkeit von `isSorted` und `merge`)
Für beliebige Listen L_1 und L_2 gilt:

$$\texttt{isSorted}\big(\texttt{merge}(L_1, L_2)\big) \leftrightarrow \texttt{isSorted}(L_1) \wedge \texttt{isSorted}(L_2).$$

Beweis durch Wertverlaufs-Induktion nach der Definition der Funktion `merge`.

1. Fall: $L_1 = []$. Es gilt

   $$\begin{array}{lll}
   & \texttt{isSorted}\big(\texttt{merge}(L_1, L_2)\big) & \\
   \leftrightarrow & \texttt{isSorted}\big(\texttt{merge}([], L_2)\big) & \\
   \leftrightarrow & \texttt{isSorted}(L_2) & \text{nach Definition von } \texttt{merge} \\
   \leftrightarrow & \texttt{true} \wedge \texttt{isSorted}(L_2) & \\
   \leftrightarrow & \texttt{isSorted}([]) \wedge \texttt{isSorted}(L_2) & \text{nach Definition von } \texttt{isSorted} \\
   \leftrightarrow & \texttt{isSorted}(L_1) \wedge \texttt{isSorted}(L_2) &
   \end{array}$$

2. Fall: $L_2 = []$.
   Dieser Fall ist analog zum 1. Fall.

3. Fall: $L_1 = [x_1] + R_1$, $L_2 = [x_2] + R_2$, $x_1 \preceq x_2$.

$$
\begin{array}{ll}
 & \texttt{isSorted}\big(\texttt{merge}(L_1, L_2)\big) \\
\leftrightarrow & \texttt{isSorted}\big(\texttt{merge}([x_1] + R_1, [x_2] + R_2)\big) \\
\leftrightarrow & \texttt{isSorted}\big([x_1] + \texttt{merge}(R_1, [x_2] + R_2)\big) \\
 & \text{nach Definition von } \texttt{merge} \\
\leftrightarrow & \texttt{le}\big(x_1, \texttt{merge}(R_1, [x_2] + R_2)\big) \wedge \texttt{isSorted}\big(\texttt{merge}(R_1, L_2)\big) \\
 & \text{nach Definition von } \texttt{isSorted} \\
\leftrightarrow & \texttt{le}(x_1, R_1) \wedge \texttt{le}(x_1, [x_2] + R_2) \wedge \texttt{isSorted}\big(\texttt{merge}(R_1, L_2)\big) \\
 & \text{wegen der Distributivität von } \texttt{le} \text{ über } \texttt{merge} \\
\leftrightarrow & \texttt{le}(x_1, R_1) \wedge \texttt{le}(x_1, [x_2] + R_2) \wedge \texttt{isSorted}(R_1) \wedge \texttt{isSorted}(L_2) \\
 & \text{nach Induktions-Voraussetzung} \\
\leftrightarrow & \texttt{le}(x_1, R_1) \wedge \texttt{isSorted}(R_1) \wedge \texttt{le}(x_1, [x_2] + R_2) \wedge \texttt{isSorted}(L_2) \\
\leftrightarrow & \texttt{isSorted}(L_1) \wedge \texttt{le}(x_1, [x_2] + R_2) \wedge \texttt{isSorted}(L_2) \\
 & \text{nach Definition von } \texttt{isSorted} \\
\leftrightarrow & \texttt{isSorted}(L_1) \wedge x_1 \preceq x_2 \wedge \texttt{le}(x_1, R_2) \wedge \texttt{isSorted}(L_2) \\
 & \text{nach Definition von } \texttt{le} \\
\leftrightarrow & \texttt{isSorted}(L_1) \wedge \texttt{le}(x_1, R_2) \wedge \texttt{isSorted}(L_2) \\
 & \text{wegen } x_1 \preceq x_2
\end{array}
$$

Der Beweis ist abgeschlossen, wenn wir

$$\texttt{isSorted}(L_2) \leftrightarrow \texttt{le}(x_1, R_2) \wedge \texttt{isSorted}(L_2)$$

zeigen können. Dazu reicht es offenbar, wenn wir

$$\texttt{isSorted}([x_2] + R_2) \rightarrow \texttt{le}(x_1, R_2) \wedge \texttt{isSorted}([x_2] + R_2) \qquad (\star)$$

zeigen, denn die umgekehrte Richtung ist trivial. Um $(\star)$ zu zeigen, reicht es wenn wir

$$\texttt{isSorted}([x_2] + R_2) \rightarrow \texttt{le}(x_1, R_2)$$

nachweisen. Nach Definition von `isSorted` gilt

$$\texttt{isSorted}([x_2] + R_2) \leftrightarrow \texttt{le}(x_2, R_2) \wedge \texttt{isSorted}(R_2).$$

Aus $\texttt{isSorted}([x_2] + R_2)$ folgt also insbesondere

$$\texttt{le}(x_2, R_2).$$

Da $x_1 \preceq x_2$ ist, folgt mit dem Lemma über die Transitivität von `le` (Lemma 17):

$$\texttt{le}(x_1, R_2)$$

und das war zu zeigen.

4. Fall: $L_1 = [x_1] + R_1$, $L_2 = [x_2] + R_2$, $x_1 > x_2$.
   Der Beweis verläuft analog zum 3. Fall. □

Satz 34 (Verträglichkeit von `isSorted` und `sort`)
Für beliebige Listen L von Elementen aus M gilt

$$\texttt{isSorted}\big(\texttt{sort}(L)\big).$$

Beweis: Wir führen den Beweis durch Wertverlaufs-Induktion entsprechend der Definition der Funktion `sort`.

1. Fall: $\#L < 2$. Es gilt

   $$\texttt{isSorted}\big(\texttt{sort}(L)\big) \leftrightarrow \texttt{isSorted}(L) \leftrightarrow \texttt{true},$$

   denn alle Listen mit weniger als zwei Elementen sind sortiert.

2. Fall: $\#L \geq 2$. Es sei $L = [x_1, x_2] + R$. Zunächst definieren wir

   $$[R_1, R_2] = \texttt{split}(R).$$

Dann gilt

$$
\begin{array}{ll}
 & \texttt{isSorted}\big(\texttt{sort}(L)\big) \\
\leftrightarrow & \texttt{isSorted}\big(\texttt{sort}([x_1, x_2] + R)\big) \\
\leftrightarrow & \texttt{isSorted}\Big(\texttt{merge}\big(\texttt{sort}([x_1] + R_1), \texttt{sort}([x_2] + R_2)\big)\Big) \\
 & \text{nach Definition von } \texttt{sort} \\
\leftrightarrow & \texttt{isSorted}\big(\texttt{sort}([x_1] + R_1)\big) \wedge \texttt{isSorted}\big(\texttt{sort}([x_2] + R_2)\big) \\
 & \text{wegen der Distributivität von } \texttt{isSorted} \text{ über } \texttt{merge} \\
\leftrightarrow & \texttt{true} \wedge \texttt{true} \\
 & \text{nach Induktions-Voraussetzung} \\
\leftrightarrow & \texttt{true} \qquad \square
\end{array}
$$

Satz 35 (Verträglichkeit von `count` und `sort`)
Für beliebige $x \in M$ und beliebige Listen L von Elementen aus M gilt

$$\texttt{count}\big(x, \texttt{sort}(L)\big) = \texttt{count}(x, L).$$

Aufgabe: Formulieren und beweisen Sie die Lemmata, die Sie zum Beweis des letzten Satzes benötigen und beweisen Sie dann den Satz.

5.3.2 Komplexität

Wir wollen wieder berechnen, wieviele Vergleiche beim Sortieren einer Liste mit n Elementen durchgeführt werden. Dazu analysieren wir zunächst, wieviele Vergleiche zum Mischen zweier Listen L_1 und L_2 benötigt werden. Wir definieren eine Funktion

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

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

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

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

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

Dann müssen L_1 und L_2 leer sein und somit ist beim Aufruf von $\texttt{merge}(L_1, L_2)$ kein Vergleich erforderlich. Also gilt

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

I.S.: $\#L_1 + \#L_2 = n + 1$.

Falls entweder L_1 oder L_2 leer ist, so ist kein Vergleich erforderlich und wir haben

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

Wir nehmen also an, dass gilt:

$$L_1 = [x] + R_1 \quad \text{und} \quad L_2 = [y] + R_2.$$

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

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

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

Also haben wir:

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

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

Wir bezeichnen nun die Anzahl der Vergleiche, die beim Aufruf von `sort`(L) für eine Liste L der Länge n schlimmstenfalls durchgeführt werden müssen mit a_n. Zur Vereinfachung nehmen wir an, dass n die Form

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

hat und definieren $b_k = a_n = a_{2^k}$. Zunächst berechnen wir den Angfangs-Wert $b_0 = a_{2^0} = a_1 = 0$. Im rekursiven Fall wird zur Berechnung von `sort`(L) die Liste L zunächst durch `split` in zwei gleich große Listen geteilt, die dann rekursiv sortiert werden. Anschließend werden die sortierten Listen gemischt. Das liefert für das Sortieren einer Liste der Länge 2^{k+1} die Rekurrenz-Gleichung

$$b_{k+1} = 2 * b_k + 2^{k+1}, \tag{1}$$

denn das Mischen der beiden halb so großen Listen kostet schlimmstenfalls $2^k + 2^k = 2^{k+1}$ Vergleiche und das rekursive Sortieren der beiden Teil-Listen kostet insgesamt $2 * b_k$ Vergleiche.

Um diese Rekurrenz-Gleichung zu lösen, führen wir in (1) die Substitution $k \mapsto k + 1$ durch und erhalten

$$b_{k+2} = 2 * b_{k+1} + 2^{k+2}. \tag{2}$$

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

$$b_{k+2} - 2 * b_{k+1} = 2 * b_{k+1} - 4 * b_k. \tag{3}$$

Diese Gleichung vereinfachen wir zu

$$b_{k+2} = 4 * b_{k+1} - 4 * b_k. \tag{4}$$

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

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

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

$$b_k = \alpha * 2^k + \beta * k * 2^k. \tag{5}$$

Setzen wir hier den Wert für $k = 0$ ein, so erhalten wir

$$0 = \alpha.$$

Aus (1) erhalten wir den Wert $b_1 = 2 * b_0 + 2^1 = 2$. Setzen wir also in Gleichung (5) für k den Wert 1 ein, so finden wir

$$2 = 0 * 2^1 + \beta * 1 * 2^1,$$

also muß $\beta = 1$ gelten. Damit lautet die Lösung

$$b_k = k * 2^k.$$

Aus $n = 2^k$ folgt $k = \log_2(n)$ und daher gilt für a_n

$$a_n = n * \log_2(n).$$

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

$$n * \log_2(n) - n + 1$$

Vergleiche beim "*Sortieren durch Mischen*" einer nicht-leeren Liste der Länge n durchgeführt werden müssen.

5.3.3 Implementierung in *Java*

Abbildung 5.8 auf Seite 80 zeigt die Implementierung des Algorithmus "*Sortieren durch Mischen*" in *Java*. Wir diskutieren zunächst die Member-Variablen der Klasse `MergeSortAlgorithm`.

1. `mArray` ist das zu sortierende Feld.

2. `mAux` ist ein Hilfsfeld, das wir während der Durchführung des Algorithmus benötigen um Werte zwischenzuspeichern.

3. `mComparator` ist ein sogenanntes *Funktions-Objekt*. Dieses Objekt stellt die Methode

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

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

   Die Methode `compare` hat noch eine andere Funktion: Über diese Methode wird die graphische Animation des Algorithmus gesteuert, die wir allerdings in diesem Skript nicht weiter diskutieren werden.

Der Konstruktor in Zeile 7 erwartet als Argumente das zu sortierende Feld und das Funktions-Objekt zum Vergleichen von Zahlen. Er speichert diese Argumente in den Member-Variablen 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*`]`, $\cdots$, `array[`*end*`-1]`.

sortiert.

Die Implementierung in *Java* weicht insofern von der Implementierung in SETL2 ab, als wir keine Funktion `split` mehr benötigen, denn wir rechnen einfach die Mitte des Feldes `mArray` mit

```
middle = (start + end) / 2;
```

aus 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 23 — 43 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

`array[start]`, $\cdots$, `array[middle-1]`,

das zweite Teilfeld besteht aus den Elementen

`array[middle]`, $\cdots$, `array[end-1]`.

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 – 26 in das Hilfs-Feld `mAux` kopiert.

2. Anschließend definieren wir drei Indizes:

   (a) `idx1` zeigt auf das nächste zu untersuchende Element im ersten Teilfeld.

   (b) `idx2` zeigt auf das nächste zu untersuchende Element im zweiten Teilfeld.

   (c) `i` gibt die Position im Ergebnis-Feld an, in die das nächste Element geschrieben wird.

3. Dann vergleichen wir in Zeile 31 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 37 — 39 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 40 — 42, die verbleibenden Elemente des zweiten Teilfeldes in das Feld `mArray` kopiert

Die in Abbildung 5.8 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

```
1  public class MergeSortAlgorithm extends SortingAlgorithm
2  {
3      private Double[]           mArray;
4      private Double[]           mAux;
5      private Comparator<Double> mComparator;
6
7      MergeSortAlgorithm(Double[] array, Comparator<Double> comparator) {
8          mArray      = array;
9          mAux        = new Double[mArray.length];
10         mComparator = comparator;
11     }
12     public void sort() {
13         mergeSort(0, mArray.length);
14     }
15     private void mergeSort(int start, int end) {
16         if (end - start < 2)
17             return;
18         int middle = (start + end) / 2;
19         mergeSort( start,  middle );
20         mergeSort( middle, end    );
21         merge( start, middle, end );
22     }
23     private void merge(int start, int middle, int end) {
24         for (int i = start; i < end; ++i) {
25             mAux[i] = mArray[i];
26         }
27         int idx1 = start;
28         int idx2 = middle;
29         int i    = start;
30         while (idx1 < middle && idx2 < end) {
31             if (mComparator.compare(mAux[idx1], mAux[idx2]) <= 0) {
32                 mArray[i++]  = mAux[idx1++];
33             } else {
34                 mArray[i++]  = mAux[idx2++];
35             }
36         }
37         while (idx1 < middle) {
38             mArray[i++]  = mAux[idx1++];
39         }
40         while (idx2 < end) {
41             mArray[i++]  = mAux[idx2++];
42         }
43     }
44 }
```

Abbildung 5.8: Die Klasse `MergeSortAlgorithm`.

Zeil kostet, denn beim Aufruf einer Funktion müssen zum einen die lokalen Variablen der Funktion auf dem Stack gesichert werden und zum anderen müssen die Argumente, mit denen die Funktion aufgerufen wird, ebenfalls auf den Stack gelegt werden. Wir zeigen daher, wie sich der Merge-Sort-Algorithmus sequentiell implementieren läßt. Abbildung 5.9 auf Seite 81 zeigt eine solche Implementierung. Statt der rekursiven Aufrufe haben wir hier zwei ineinander geschachtelte

```
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), min(l * (k + 2), mArray.length));
9          }
10     }
11 }
```

Abbildung 5.9: Eine sequentielle Implementierung des Merge-Sort-Algorithmus

.

`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

$$\Big[x[k*l],\ x[k*l+1],\ x[k*l+2],\ \cdots,\ x[(k+1)*l-1]\Big]$$

und wenn n die Länge des Feldes x ist, dann läßt sich die Aussage, dass dieses Teilfeld sortiert ist, durch die Formel

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

beschreiben. Die Bedingung $k*l+j+1 < n$ ist notwendig um sicherzustellen, dass $x[k*l+j+1]$ 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 $2*l$ 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

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

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

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

5.4 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:
   $$\texttt{sort}([]) = [].$$
2. Sonst nehmen wir das erste Element der Liste L, nennen es x und verteilen die restlichen Elemente von L auf zwei Listen L_1 und L_2, dass L_1 alle Elemente enthält die kleiner-gleich x sind, während L_2 die restlichen Elemente enthält. Anschließend sortieren wir diese Listen rekursiv, hängen dann die sortierte Liste L_1 vor x und die sortierte Liste L_2 dahinter:
   $$\texttt{sort}([x] + R) = \texttt{sort}\big(\texttt{smaller}(x, R)\big) + [x] + \texttt{sort}\big(\texttt{bigger}(x, R)\big).$$
   Die Liste $\texttt{smaller}(x, R)$ enthält alle Elemente aus R, die kleiner-gleich x sind, die restlichen Elemente aus R sammeln wir in der Liste $\texttt{bigger}(x, R)$. Wir implementieren die Berechnung der beiden Listen über eine Funktion $\texttt{partition}(x, L)$ für die gilt
   $$\texttt{partition}(x, L) = \big[\texttt{smaller}(x, L), \texttt{bigger}(x, L)\big].$$

Abbildung 5.10 zeigt die Implementierung des "*Quick-Sort*-Algorithmus".

```
1  procedure sort(L);
2      if L = [] then
3          return L;
4      end if;
5      x := L(1); R := L(2..);
6      [ L1, L2 ] := partition(x, R);
7      return sort(L1) + [x] + sort(L2);
8  end sort;
```

Abbildung 5.10: Der *Quick-Sort*-Algorithmus.

Die Implementierung der Funktion `partition` verläuft nach folgendem Algorithmus:

1. Ist die Liste L leer, so liefert $\texttt{partition}(x, L)$ als Ergebnis ein Paar leerer Listen:
   $$\texttt{partition}(x, []) = \big[[], []\big].$$
2. Hat L die Form $[y] + R$, so wird R rekursiv aufgeteilt in R_1 und R_2, wobei R_1 alle Elemente kleiner-gleich x enthält und R_2 die restlichen Elemente. Anschließend wird y an den Anfang von R_1 oder R_2 gesetzt, je nach dem ob $y \preceq x$ gilt oder nicht:
   $$y \preceq x \wedge \texttt{partition}(x, R) = [R_1, R_2] \rightarrow \texttt{partition}\big(x, [y] + R\big) = \big[[y] + R_1, R_2\big],$$
   $$\neg y \preceq x \wedge \texttt{partition}(x, R) = [R_1, R_2] \rightarrow \texttt{partition}\big(x, [y] + R\big) = \big[R_1, [y] + R_2\big].$$

Abbildung 5.11 zeigt die Implementierung der Funktion `partition`.

5.4.1 Komplexität

Wir untersuchen wieder die Zahl der Vergleiche, die beim Aufruf von $\texttt{sort}(L)$ für eine Liste L mit n Elementen durchgeführt werden. Dazu betrachten wir zunächst die Anzahl der Vergleiche, die wir durchführen müssen, um $\texttt{partition}(x, L)$ für eine Liste L mit n Elementen zu berechnen. Da wir jedes der n Elemente der Liste L mit x vergleichen müssen, ist klar, dass wir insgesamt n Vergleiche durchführen müssen.

```
1   procedure partition(x, L);
2       if L = [] then
3           return [ [], [] ];
4       end if;
5       y := L(1); R := L(2..);
6       [ R1, R2 ] := partition(x, R);
7       if y <= x then
8           return [ [y] + R1, R2 ];
9       else
10          return [ R1, [y] + R2 ];
11      end if;
12  end partition;
```

Abbildung 5.11: Die Prozedur `partition`.

Komplexität im schlechtesten Fall

Wir berechnen als nächstes die Anzahl a_n von Vergleichen, die wir im schlimmsten Fall beim Aufruf von `sort`(L) für eine Liste L der Länge n durchführen müssen. Der schlimmste Fall tritt beispielsweise dann ein, wenn die Liste L_1 leer ist und L_2 folglich gleich R sein muss. Damit hat L_2 dann die Länge $n-1$ und für die Anzahl a_n der Vergleiche gilt

$$a_n = a_{n-1} + n - 1.$$

Der Term $n-1$ rührt von den $n-1$ Vergleichen, die beim Aufruf von `partition`(x, R) in Zeile 6 von Abbildung 5.10 durchgeführt werden und der Term a_{n-1} erfaßt die Anzahl der Vergleiche, die beim rekursiven Aufruf von `sort`(L_2) benötigt werden.

Die Anfangs-Bedingung für die Rekurrenz-Gleichung lautet $a_0 = 0$, denn beim Sortieren einer leeren Liste sind keine Vergleiche notwendig. Damit läßt sich die obige Rekurrenz-Gleichung mit dem *Teleskop-Verfahren* lösen:

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

Das ist die selbe Zahl von Vergleichen, die wir beim "*Sortieren durch Einfügen*" benötigen. 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 nicht bei sortierten Listen eintritt. Wir gehen später näher darauf ein.

Durschnittliche 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+1}, die wir beim Sortieren einer Liste L mit $n+1$ Elementen erwarten müssen. Im Allgemeinen gilt: Ist L eine Liste mit $n+1$ Elementen, so ist die Zahl der Elemente der Liste `L1`, die in Zeile 6 von Abbildung 5.10 berechnet wird, ein Element der Menge $\{0, 1, 2, \cdots, n\}$. Hat die Liste `L1` insgesamt i Elemente, so enthält die Liste `L2` die restlichen $n-i$ Elemente. Gilt $\#\texttt{L1} = i$, so werden zum rekursiven Sortieren von `L1` und `L2` durchschnittlich

$d_i + d_{n-i}$

Vergleiche durchgeführt. Bilden wir den Durchschnitt dieses Wertes für alle $i \in \{0, 1, \cdots, n\}$, so erhalten wir für d_{n+1} die Rekurrenz-Gleichung

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

Der Term n rührt von den n Vergleichen, die beim Aufruf von `partition`(x, R) durchgeführt werden. Um die Rekurrenz-Gleichung (1) zu vereinfachen, bemerken wir zunächst, dass für beliebige Funktionen $f : \mathbb{N} \to \mathbb{N}$ gilt:

$$\begin{aligned} \sum_{i=0}^{n} f(n-i) &= f(n) + f(n-1) + \cdots + f(1) + f(0) \\ &= f(0) + f(1) + \cdots + f(n-1) + f(n) = \sum_{i=0}^{n} f(i) \end{aligned}$$

Damit vereinfacht sich die Rekurrenz-Gleichung (1) zu

$$d_{n+1} = n + \frac{2}{n+1} \sum_{i=0}^{n} d_i. \tag{2}$$

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

$$d_{n+2} = n + 1 + \frac{2}{n+2} \sum_{i=0}^{n+1} d_i. \tag{3}$$

Wir multiplizieren Gleichung (3) mit $n+2$ und Gleichung (2) mit $n+1$ und erhalten

$$(n+2) * d_{n+2} = (n+2) * (n+1) + 2 * \sum_{i=0}^{n+1} d_i. \tag{3'}$$

$$(n+1) * d_{n+1} = (n+1) * n + 2 * \sum_{i=0}^{n} d_i. \tag{2'}$$

Wir bilden die Differenz der Gleichungen (3′) und (2′) und beachten, dass sich die Summationen bis auf den Term $2 * d_{n+1}$ gerade gegenseitig aufheben:

$$(n+2) * d_{n+2} - (n+1) * d_{n+1} = (n+2) * (n+1) - (n+1) * n + 2 * d_{n+1}$$

Diese Gleichung vereinfachen wir:

$$(n+2) * d_{n+2} = (n+3) * d_{n+1} + 2 * (n+1).$$

Diese Gleichung teilen wir durch $(n+2) * (n+3)$ und erhalten

$$\frac{1}{n+3} * d_{n+2} = \frac{1}{n+2} * d_{n+1} + \frac{2 * (n+1)}{(n+2) * (n+3)}. \tag{4}$$

Als nächstes bilden wir die Partialbruch-Zerlegung von dem Bruch $\dfrac{2 * (n+1)}{(n+2) * (n+3)}$ indem wir den Ansatz

$$\frac{2 * (n+1)}{(n+2) * (n+3)} = \frac{\alpha}{n+2} + \frac{\beta}{n+3}$$

machen. Dann erhalten wir $\alpha = -2$ und $\beta = 4$, so dass wir die Gleichung (4) als

$$\frac{1}{n+3} * d_{n+2} = \frac{1}{n+2} * d_{n+1} - \frac{2}{n+2} + \frac{4}{n+3}. \tag{5}$$

schreiben können. Wir definieren $a_n = \dfrac{d_n}{n+1}$ und erhalten dann aus (5)

$$a_{n+2} = a_{n+1} - \frac{2}{n+2} + \frac{4}{n+3}.$$

Die Substitution $n \mapsto n-2$ vereinfacht dies zu

$$a_n = a_{n-1} - \frac{2}{n} + \frac{4}{n+1}. \tag{6}$$

Die Gleichung (6) können wir mit dem Teleskop-Verfahren lösen. Es gilt

$$a_n = 4 * \sum_{i=1}^{n} \frac{1}{i+1} - 2 * \sum_{i=1}^{n} \frac{1}{i}. \tag{7}$$

Wir vereinfachen diese Summe:

$$\begin{aligned}
a_n &= 4 * \sum_{i=1}^{n} \frac{1}{i+1} - 2 * \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 * \sum_{i=2}^{n+1} \frac{1}{i} - 2 * \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 * \frac{1}{n+1} - 4 * \frac{1}{1} + 4 * \sum_{i=1}^{n} \frac{1}{i} - 2 * \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 * \frac{1}{n+1} - 4 * \frac{1}{1} + 2 * \sum_{i=1}^{n} \frac{1}{i} \\
&= -\frac{4*n}{n+1} + 2 * \sum_{i=1}^{n} \frac{1}{i}
\end{aligned}$$

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 der Grenzwert

$$\gamma := \lim_{n\to\infty} \left(\left(\sum_{i=1}^{n} \tfrac{1}{i} \right) - \ln(n) \right) \approx 0.5772156649\cdots$$

existiert. Die Konstante γ wird als die *Euler-Mascheroni-Konstante* bezeichnet. Genauer gilt

$$H_n = \ln(n) + \gamma + \mathcal{O}\left(\tfrac{1}{n}\right) = \ln(n) + \mathcal{O}(1).$$

Wir haben bei dieser Gleichung eine Schreibweise benutzt, die wir bisher noch nicht eingeführt haben. Sind f, g, h Funktionen aus $\mathbb{R}^{\mathbb{N}}$, so schreiben wir

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

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

$$d_n = -4 * n + 2 * (n+1) * H_n = -4 * n + 2 * (n+1) * \big(\ln(n) + \mathcal{O}(1)\big) = 2 * n * \ln(n) + \mathcal{O}(n).$$

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

$$n * \log_2(n) + \mathcal{O}(n)$$

ergeben. Wegen $\ln(n) = \ln(2) * \log_2(n)$ benötigen wir bei Quick-Sort im Durschnitt

$$2 * \ln(2) * n * \log_2(n)$$

Vergleiche, also etwa $2 * \ln(2) \approx 1.39$ mehr Vergleiche als beim "*Sortieren durch Mischen*".

5.4.2 Implementierung von *Quick-Sort* in *Java*

Zum Abschluß geben wir eine Implementierung des *Quick-Sort*-Algorithmus in *Java* an. Abbildung 5.12 zeigt die Implementierung der Methode `quickSort` in *Java*. Diese Methode wird mit zwei Argumenten aufgerufen:

1. `start` gibt den Index des ersten Elementes des Teilfeldes an, das zu sortieren ist.

2. `end` gibt den Index des letzten Elementes des Teilfeldes an, das zu sortieren ist.

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

$$\texttt{mArray[start]}, \texttt{mArray[start+1]}, \cdots, \texttt{mArray[end]}$$

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

$$\texttt{mArray[start]} \preceq \texttt{mArray[start+1]} \preceq \cdots \preceq \texttt{mArray[end]}.$$

Die Implementierung weicht hier kaum von dem SETL2-Programm ab. Der wesentliche Unterschied besteht darin, dass die Funktion `partition`, die in Zeile 5 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.

```
1  private void quickSort(int start, int end) {
2       if (end <= start) {
3           return;
4       }
5       int splitIdx = partition(start, end);
6       quickSort(start, splitIdx - 1);
7       quickSort(splitIdx + 1, end );
8   }
9
```

Abbildung 5.12: Implementierung des *Quick-Sort*-Algorithmus in *Java*.

Die Schwierigkeit bei der Implementierung von *Quick-Sort* liegt in der Codierung der Methode `partition`, die in Abbildung 5.13 auf Seite 83 gezeigt wird. Die Funktion `partition` erhält zwei Argumente:

1. `start` ist der Index des ersten Elementes in dem aufzuspaltenden Teilbereich.
2. `end` ist der Index des letzten Elementes in dem aufzuspaltenden Teilbereich.

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

$$\texttt{splitIdx} \in \{\texttt{start}, \texttt{start} + 1, \cdots, \texttt{end}\}.$$

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

1. Alle Elemente mit Index aus der Menge $\{\texttt{start}, \cdots, \texttt{splitIdx} - 1\}$ kommen in der Ordnung "$\preceq$" vor dem Element an der Stelle `splitIdx`:
   $$\forall i \in \{\texttt{start}, \cdots, \texttt{splitIdx} - 1\}\colon \texttt{array}[i] \preceq \texttt{array}[\texttt{splitIdx}].$$
2. Alle Elemente mit Index aus der Menge $\{\texttt{splitIdx} + 1, \cdots, \texttt{end}\}$ kommen in der Ordnung "$\preceq$" hinter dem Element an der Stelle `splitIdx`:
   $$\forall i \in \{\texttt{splitIdx} + 1, \cdots, \texttt{end}\}\colon \texttt{array}[\texttt{splitIdx}] \prec \texttt{array}[i].$$

Der Algorithmus, um diese Bedingungen zu erreichen, wählt zunächst das Element `array[start]` als sogenanntes *Pivot-Element* aus. Anschließend läuft der Index `leftIdx` von `start` + 1 von links nach rechts bis ein Element gefunden wird, das größer als das Pivot-Element ist. Analog läuft der Index `rightIdx` von `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.

```
1  int partition(int start, int end) {
2      Double x  = mArray[start];
3      int left  = start + 1;
4      int right = end;
5      while (true) {
6          while (left <= end && mComparator.compare(mArray[left], x) <= 0) {
7              ++left;
8          }
9          while (mComparator.compare(mArray[right], x) > 0) {
10             --right;
11         }
12         if (left >= right) {
13             break;
14         }
15         swap(left, right);
16     }
17     swap(start, right);
18     return right;
19 }
```

Abbildung 5.13: Die Prozedur `partition` in *Java*.

Die Methode `swap`(i, j) vertauscht die Feld-Elemente mit den Indizes i und j. Da die Implementierung die selbe ist wie bei dem Algorithmus "*Sortieren durch Mischen*" verzichten wir darauf, die Implementierung noch einmal anzugeben.

5.4.3 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 5 bis Zeile 16 erstreckt, gelten. Beim Betreten der Schleife in Zeile 6 gilt:

1. $\forall i \in \{\texttt{start} + 1, \cdots, \texttt{left} - 1\}$: $\texttt{mArray}[i] \preceq x$
2. $\forall j \in \{\texttt{right} + 1, \cdots, \texttt{end}\}$: $x \prec \texttt{mArray}[j]$
3. $\texttt{start} + 1 \leq \texttt{left}$
4. $\texttt{right} \leq \texttt{end}$
5. $\texttt{left} \leq \texttt{right} + 1$

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

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

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

$$\{\texttt{start} + 1, \cdots, \texttt{left} - 1\} = \{\texttt{start} + 1, \cdots, \texttt{start}\} = \emptyset$$

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

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

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

$$\{\texttt{right}+1,\cdots,\texttt{end}\} = \{\texttt{end}+1,\cdots,\texttt{end}\} = \emptyset,$$

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

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

denn die Methode `partition`(`start`, `end`) wird nur aufgerufen, falls $\texttt{start} < \texttt{end}$ ist.

Als nächstes zeigen wir, dass die Invarianten bei einem Schleifen-Durchlauf erhalten bleiben.

1. Die erste Invariante gilt, weil `left` nur dann inkrementiert wird, wenn vorher

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

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

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

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

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

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

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

3. Die Gültigkeit der dritten Invariante folgt aus der Tatsache, dass `left` in der ganzen Schleife höchstens inkrementiert wird. Wenn also zu Beginn $\texttt{start}+1 \leq \texttt{left}$ gilt, so wird dies immer gelten.

4. Analog ist die vierten Invariante gültig, weil zu Beginn $\texttt{right} \leq \texttt{end}$ gilt und `right` immer nur dekrementiert wird.

5. Um die fünfte Invariante zu zeigen, führen wir eine Fallunterscheidung durch:

   (a) Fall: Nach dem Ende der Schleife in Zeile 6 — 8 gilt

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

   Diese Schleife bricht also ab, weil die Bedingung $\texttt{left} \leq \texttt{end}$ verletzt ist. Da aber vor Beginn der Schleife

   $$\texttt{left} \leq \texttt{right}+1 \leq \texttt{end}+1$$

   gilt und die Variable `left` in jedem Schleifen-Durchlauf höchstens um 1 erhöht wird und das auch nur, wenn $\texttt{left} \leq \texttt{end}$ gilt, muß

   $$\texttt{left} = \texttt{end}+1$$

   gelten. Aus der Gültigkeit der ersten Invariante und der Gleichung $\texttt{left} = \texttt{end}+1$ folgt, dass

   $$\forall i \in \{\texttt{start}+1,\cdots,\texttt{end}\}\colon \texttt{mArray}[i] \preceq x$$

   gilt. Andererseits sagt die zweite Invariante

   $$\forall j \in \{\texttt{right}+1,\cdots,\texttt{end}\}\colon\; x \prec \texttt{mArray}[j].$$

   Das kann nur dann richtig sein, wenn $\texttt{right} = \texttt{end}$ ist. Damit haben wir also

   $$\texttt{left} \leq \texttt{right}+1 \;\leftrightarrow\; \texttt{end}+1 \leq \texttt{end}+1 \;\leftrightarrow\; \texttt{true}.$$

   (b) Fall: Nach dem Ende der Schleife in Zeile 6 — 8 gilt

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

   Die Schleife bricht also ab, weil die Bedingung $\texttt{mArray}[\texttt{left}] \preceq x$ verletzt ist. In diesem Fall folgt die Gültigkeit der dritten Invariante aus den ersten beiden Invarianten, denn die ersten beiden Invarianten garantieren, dass die folgenden Index-Mengen disjunkt sind:

   $$\{\texttt{start}+1,\cdots,\texttt{left}-1\} \cap \{\texttt{right}+1,\cdots,\texttt{end}\} = \emptyset.$$

Daraus folgt aber

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

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 der Bedingung

$$\texttt{left} \leq \texttt{end}$$

enthält, kann `left` nicht beliebig oft inkrementiert werden und die Schleife muß irgendwann abbrechen.

Die zweite innere `while`-Schleife terminiert, weil einerseits `right` in jedem Schleifen-Durchlauf dekrementiert wird und andererseits aus der dritten und der fünften Invariante folgt:

$$\texttt{right} \geq \texttt{left} - 1 \geq \texttt{start}.$$

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

$$M = \{\texttt{left}, \cdots, \texttt{right}\}$$

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

1. Fall: Nach dem Ende der Schleife in Zeile 6 — 8 gilt

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

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

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

   gelten muß. Daraus folgt aber sofort

   $$\texttt{left} \geq \texttt{right}$$

   und folglich wird die Schleife durch den Befehl `break` in Zeile 13 abgebrochen.

2. Fall: Nach dem Ende der Schleife in Zeile 6 — 8 gilt

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

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

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

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

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

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

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

   Wenn nun also die äußere Schleife erneut durchlaufen wird, dann wird die zweite innere Schleife mindestens einmal durchlaufen, so dass also `right` dekrementiert wird und folglich die Menge $M = \{\texttt{left}, \cdots, \texttt{right}\}$ um ein Element verkleinert wird. Das geht aber nur endlich oft, denn spätestens wenn die Menge nur noch ein Element hat, gilt $\texttt{left} = \texttt{right}$ und die Schleife wird durch den Befehl `break` in Zeile 13 abgebrochen.

Jetzt haben wir alles Material zusammen, um die Korrektheit unserer Implementierung zu zeigen. Da die Schleife abbricht, gilt $\texttt{left} \geq \texttt{right}$. Wegen der dritten Invariante gilt $\texttt{left} \leq \texttt{right} + 1$. Also gibt es nur noch zwei Möglichkeiten: $\texttt{left} = \texttt{right}$ oder $\texttt{left} = \texttt{right} + 1$. Der Fall $\texttt{left} = \texttt{right}$ kann ausgeschlossen werden, denn definieren wir $y := \texttt{mArray}[\texttt{left}] = \texttt{mArray}[\texttt{right}]$, so gilt entweder $y \preceq x$ oder $y \succ x$. Im ersten Fall wäre dann die erste innere Schleife weiter gelaufen und hätte `left` noch einmal inkrementiert, im zweiten Fall hätte die zweite innere Schleife `right` dekrementiert. Also gilt $\texttt{left} = \texttt{right} + 1$. Wegen den ersten beiden Invarianten wissen wir also

$$\forall i \in \{\texttt{start} + 1, \cdots, \texttt{right}\}\colon \texttt{mArray}[i] \preceq x$$

$$\forall j \in \{\texttt{right} + 1, \cdots, \texttt{end}\}\colon x \prec \texttt{mArray}[j]$$

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.

5.4.4 Mögliche Verbesserungen

In der Praxis gibt es noch eine Reihe Tricks, um die Implementierung von *Quick-Sort* effizienter zu machen:

1. Anstatt immer das erste Element als Pivot-Element zu wählen, werden drei Elemente aus der zu sortierenden Liste ausgewählt, z. B. das erste, das letzte und ein Element aus der Mitte des Feldes. Als Pivot-Element wird dann das Element gewählt, was der Größe nach zwischen den anderen Elementen liegt.

   Der Vorteil dieser Strategie liegt darin, dass der schlechteste Fall, bei dem die Laufzeit von *Quick-Sort* quadratisch ist, wesentlich unwahrscheinlicher wird. Insbesondere kann der schlechteste Fall nicht mehr bei Listen auftreten, die bereits sortiert sind.

2. Falls weniger als 10 Elemente zu sortieren sind, wird auf "*Sortieren durch Einfügen*" zurück gegriffen.

Aufgabe: Implementieren Sie diese Verbesserungen.

5.5 Bewertung der Algorithmen

Für die Praxis empfehle ich den Einsatz von "*Sortieren durch Mischen*", denn dieser Algorithmus hat auch im schlechtesten Fall eine Komplexität der Form $\mathcal{O}\big(n * \ln(n)\big)$, der Algorithmus ist folglich robust. Es ist zwar in der Praxis so, dass *Quick-Sort* für viele Anwendungen fast doppelt so schnell ist wie "*Sortieren durch Mischen*", trotzdem gibt es zwei Gründe, die gegen den Einsatz von *Quick-Sort* sprechen:

1. Die Implementierung von *Quick-Sort* in einer Sprache wie Java ist erheblich komplizierter als die Implementierung von "*Sortieren durch Mischen*". Bei *Quick-Sort* ist es insbesondere leicht, subtile Fehler zu machen, die durch Tests nur schwer zu entdecken sind. Es gibt eine Reihe Lehrbücher die diesen Punkt dadurch belegen, dass sie fehlerhafte Versionen von *Quick-Sort* angeben.

2. Im schlechtesten Fall ist die Komplexität von *Quick-Sort* eben doch quadratisch und dann dauert das Sortieren wesentlich länger als beim "*Sortieren durch Mischen*".

Es gibt allerdings eine Situation, in der die Verwendung von *Quick-Sort* dem "*Sortieren durch Mischen*" vorzuziehen ist. Eine Implementierung von *Quick-Sort* in Java kommt mit vergleichsweise wenig Speicherplatz aus, denn außer dem zu sortierenden Feld wird eigentlich nur noch etwas Platz auf dem Stack benötigt. Hingegen ist bei der Implementierung von "*Sortieren durch Mischen*" ein Hilfs-Feld erforderlich, das die selbe Größe hat wie das zu sortierende Feld. Es wird also in etwa doppelt soviel Speicherplatz benötigt wie bei *Quick-Sort*.

Kapitel 6

Der abstrakte Daten-Typ *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. Wir verallgemeinern das Beispiel mit der folgenden Definition eines abstrakten Daten-Typs.

Definition 36 (Abbildung)
Wir definieren den abstrakten Daten-Typ der *Abbildung* wie folgt:

1. Als Namen wählen wir *Map*.
2. Die Menge der Typ-Parameter ist $\{Key, Value\}$.
3. Die Menge der Funktions-Zeichen ist
   $$\{create, find, insert, delete\}.$$
4. Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:
   (a) $create : Map$
   Der Aufruf $create()$ erzeugt eine leere Abbildung, also eine Abbildung, die keinem Schlüssel einen Wert zuweist.
   (b) $find : Map \times Key \to Value \cup \{\Omega\}$
   Der Aufruf $find(M, k)$ überprüft, ob in der Abbildung M zu dem Schlüssel k ein Wert abgespeichert ist. Wenn ja, wird dieser Wert zurück gegeben, sonst wird der Wert Ω zurück gegeben.
   (c) $insert : Map \times Key \times Value \to Map$
   Der Aufruf $insert(M, k, v)$ fügt in der Abbildung M für den Schlüssel k den Wert v ein. Falls zu dem Schlüssel k bereits ein Eintrag in der Abbildung M existiert, so wird dieser überschrieben. Andernfalls wird ein entsprechender Eintrag neu angelegt. Als Ergebnis wird die geänderte Abbildung zurück gegeben.

(d) $delete : Map \times Key \to Map$
Der Aufruf $delete(M, k)$ entfernt den Eintrag zu dem Schlüssel k in der Abbildung M. Falls kein solcher Eintrag existiert, bleibt die Abbildung M unverändert. Als Ergebnis wird die eventuell geänderte Abbildung zurück gegeben.

5. Das genaue Verhalten der Funktionen wird durch die nachfolgenden Axiome spezifiziert.

(a) $find\big(create(), k\big) = \Omega$,
denn der Aufruf $create()$ erzeugt eine leere Abbildung.

(b) $find\big(insert(M, k, v), k\big) = v$,
denn wenn wir zu dem Schlüssel k einen Wert v einfügen, so finden wir anschließend eben diesen Wert v, wenn wir nach k suchen.

(c) $k_1 \neq k_2 \to find\big(insert(M, k_1, v), k_2\big) = find(M, k_2)$,
denn wenn wir für einen Schlüssel eine Wert einfügen, so ändert das nichts an dem Wert, der für einen anderen Schlüssel abgespeichert ist.

(d) $find\big(delete(M, k), k\big) = \Omega$,
denn wenn wir einen Schlüssel löschen, so finden wir anschließend auch keinen Wert mehr unter diesem Schlüssel.

(e) $k_1 \neq k_2 \to find\big(delete(M, k_1), k_2\big) = find(M, k_2)$,
denn wenn wir einen Schlüssel löschen, so ändert das nichts an dem Wert, der unter einem anderen Schlüsseln abgespeichert ist. □

Es ist in SETL2 sehr einfach, den ADT *Abbildung* zu implementieren. Dazu müssen wir uns nur klar machen, dass Abbildungen nichts anderes sind als Funktionen und die können wir in der Mengenlehre durch binäre Relationen darstellen. Ist R eine binäre Relation die genau ein Paar der Form $[k, v]$ enthält, so liefert der Ausdruck

$$R(k)$$

als Ergebnis den Wert v. Umgekehrt wird durch den Aufruf

$$R(k) := v$$

das Paar $[k, v]$ in die Relation R eingefügt. Um den Eintrag unter einem Schlüssel k zu löschen, reicht es aus, dem Schlüssel k den undefinierten Wert Ω zuzuweisen:

$$R(k) := \Omega.$$

Dieser Wert wird auch bei der Auswertung des Ausdrucks $R(k)$ zurück gegeben, wenn die binäre Relation kein Paar der Form $[k, v]$ enthält.

Abbildung 6.1 zeigt eine Implementierung des ADT Abbildung, die auf diesen Überlegungen aufbaut.

6.1 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 37 (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:

$$nil : \mathcal{B} \qquad \text{und} \qquad node : Key \times Value \times \mathcal{B} \times \mathcal{B} \to \mathcal{B}.$$

```
1   class Map;
2       procedure create();
3       procedure find(k);
4       procedure insert(k, v);
5       procedure delete(k);
6   end Map;
7
8   class body Map;
9
10      var relation;
11
12      procedure create();
13          relation := {};
14      end create;
15
16      procedure find(k);
17          return relation(k);
18      end find;
19
20      procedure insert(k, v);
21          relation(k) := v;
22      end insert;
23
24      procedure delete(k);
25          relation(k) := om;
26      end delete;
27
28  end Map;
```

Abbildung 6.1: Eine triviale Implementierung des ADT *Map*

1. *nil* ist ein binärer Baum.

   Dieser Baum wird als der *leere Baum* bezeichnet.

2. $node(k, v, l, r)$ ist ein binärer Baum, falls gilt:

   (a) k ist ein Schlüssel aus der Menge *Key*.

   (b) v ist ein Wert aus der Menge *Value*.

   (c) l ist ein binärer Baum.
   l wird als der *linke Teilbaum* von $node(k, v, l, r)$ bezeichnet.

   (d) r ist ein binärer Baum.
   r wird als der *rechte Teilbaum* von $node(k, v, l, r)$ bezeichnet. □

Als nächstes definieren wir, was wir unter einem *geordneten binären Baum* verstehen.

Definition 38 (Geordneter binärer Baum)
Die Menge $\mathcal{B}_<$ der *geordneten binären Bäume* wird induktiv definiert.

1. $nil \in \mathcal{B}_<$

2. $node(k, v, l, r) \in \mathcal{B}_<$ falls folgendes gilt:

   (a) Alle Schlüssel, die in dem linken Teilbaum l auftreten, sind kleiner als k.

(b) Alle Schlüssel, die in dem rechten Teilbaum r auftreten, sind größer als k.

(c) $l \in \mathcal{B}_<$.

(d) $r \in \mathcal{B}_<$.

Die ersten beiden Bedingungen bezeichnen wir als die *Ordnungs-Bedingung.* □

Geordnete binäre Bäume lassen sich grafisch wir folgt darstellen:

1. Der leere Baum *nil* wird durch einen dicken schwarzen Punkt dargestellt.

2. Ein Baum der Form $node(k, v, l, r)$ wird dargestellt, indem zunächst ein Oval gezeichnet wird, in dem oben der Schlüssel k und darunter, getrennt durch einen waagerechten Strich, der dem Schlüssel zugeordnete Wert v eingetragen wird. Dieses Oval bezeichnen wir auch als einen *Knoten* des binären Baums. Anschließend wird links unten von diesem Knoten rekursiv der Baum l gezeichnet und rechts unten wird rekursiv der Baum r gezeichnet. Zum Abschluß zeichnen wir von dem mit k und v markierten Knoten jeweils einen Pfeil zu dem linken und dem rechten Teilbaum.

Abbildung 6.2 zeigt ein Beispiel für einen geordneten binären Baum. Der oberste Knoten, in der Abbildung ist das der mit dem Paar $\langle 8, 22\rangle$ markierte Knoten, wird als die *Wurzel* des Baums bezeichnet. Ein *Pfad der Länge* k in dem Baum ist eine Liste $[n_0, n_1, \cdots, n_k]$ von $k+1$ Knoten, die durch Pfeile verbunden sind. Identifizieren wir Knoten mit ihren Markierungen, so ist

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

ein Pfad der Länge 3.

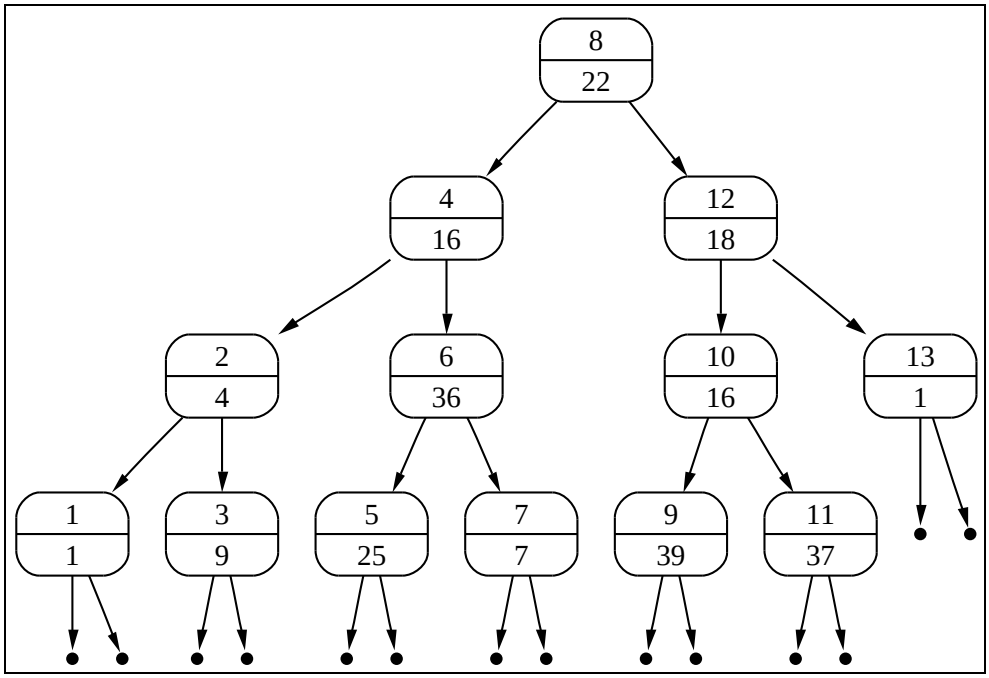

Abbildung 6.2: Ein geordneter binärer Baum

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. Wir beginnen mit der Definition von Gleichungen für die Methode `find`(). Wir verwenden dabei eine Objekt-orientierte Schreibweise, ist $b \in \mathcal{B}_<$ so schreiben wir

$$b.find(k) \quad \text{statt} \quad find(b, k).$$

1. $\mathit{nil}.\mathit{find}(k) = \Omega$,

   denn der leere Baum repräsentiert die leere Abbildung.

2. $\mathit{node}(k, v, l, r).\mathit{find}(k) = v$,

   denn der Knoten $\mathit{node}(k, v, l, r)$ speichert die Zuordnung $k \mapsto v$.

3. $k_1 < k_2 \rightarrow \mathit{node}(k_2, v, l, r).\mathit{find}(k_1) = l.\mathit{find}(k_1)$,

   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. $k_1 > k_2 \rightarrow \mathit{node}(k_2, v, l, r).\mathit{find}(k_1) = r.\mathit{find}(k_1)$,

   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. $\mathit{nil}.\mathit{insert}(k, v) = \mathit{node}(k, v, \mathit{nil}, \mathit{nil})$,

   denn wenn der Baum vorher leer ist, so kann die einzufügende Information direkt an der Wurzel abgespeichert werden.

2. $\mathit{node}(k, v_2, l, r).\mathit{insert}(k, v_1) = \mathit{node}(k, v_1, l, r)$,

   denn wenn wir den Schlüssel k an der Wurzel finden, überschreiben wir einfach den zugeordneten Wert.

3. $k_1 < k_2 \rightarrow \mathit{node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{node}\big(k_2, v_2, l.\mathit{insert}(k_1, v_1), r\big)$,

   denn wenn der Schlüssel k_1, unter dem wir Informationen einfügen wollen, kleiner als der Schlüssel k_2 an der Wurzel ist, so müssen wir die einzufügende Information im linken Teilbaum einfügen.

4. $k_1 > k_2 \rightarrow \mathit{node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{node}\big(k_2, v_2, l, r.\mathit{insert}(k_1, v_1)\big)$,

   denn wenn der Schlüssel k_1, unter dem wir Informationen einfügen wollen, größer als der Schlüssel k_2 an der Wurzel ist, so müssen wir die einzufügende Information im rechten Teilbaum einfügen.

Als letztes definieren wir die Methode *delete*. Diese Definition ist schwieriger als die Implementierung der andern beiden Methoden. Falls wir in einen Baum der Form $t = \mathit{node}(k, v, l, r)$ den Eintrag mit dem Schlüssel k löschen wollen, so kommt es auf die beiden Teilbäume l und r an. Ist l der leere Teilbaum, so liefert $t.\mathit{delete}(k)$ 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 $t = \mathit{node}(k, v, l, r')$ die Werte k und v mit dem eben gefundenen kleinsten Schlüssel k_{min} und dem k_{min} zugeordneten Wert v_{min}. Der dadurch entstehende binäre Baum $t = \mathit{node}(k_{min}, v_{min}, l, r')$ ist auch wieder geordnet, denn einerseits ist der Schlüssel k_{min} größer als der Schlüssel k und damit sicher auch größer als alle Schlüssel im linken Teilbaum l und andererseits ist k_{min} kleiner als alle Schlüssel im Teilbaum r' den k_{min} ist ja der kleinste Schlüssel aus r.

Zur Veranschaulichung betrachten wir ein Beispiel: Wenn wir in dem Baum aus Abbildung 6.2 den Knoten mit der Markierung $\langle 4, 16\rangle$ löschen wollen, so suchen wir zunächst in dem Teilbaum, dessen Wurzel mit $\langle 6, 36\rangle$ markiert ist, den Knoten, der mit dem kleinsten Schlüssel markiert ist. Dies ist der Knoten mit der Markierung $\langle 5, 25\rangle$. Wir löschen diesen Knoten und überschreiben die Markierung $\langle 4, 16\rangle$ mit der Markierung $\langle 5, 25\rangle$. Abbildung 6.3 auf Seite 96 zeigt das Ergebnis.

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

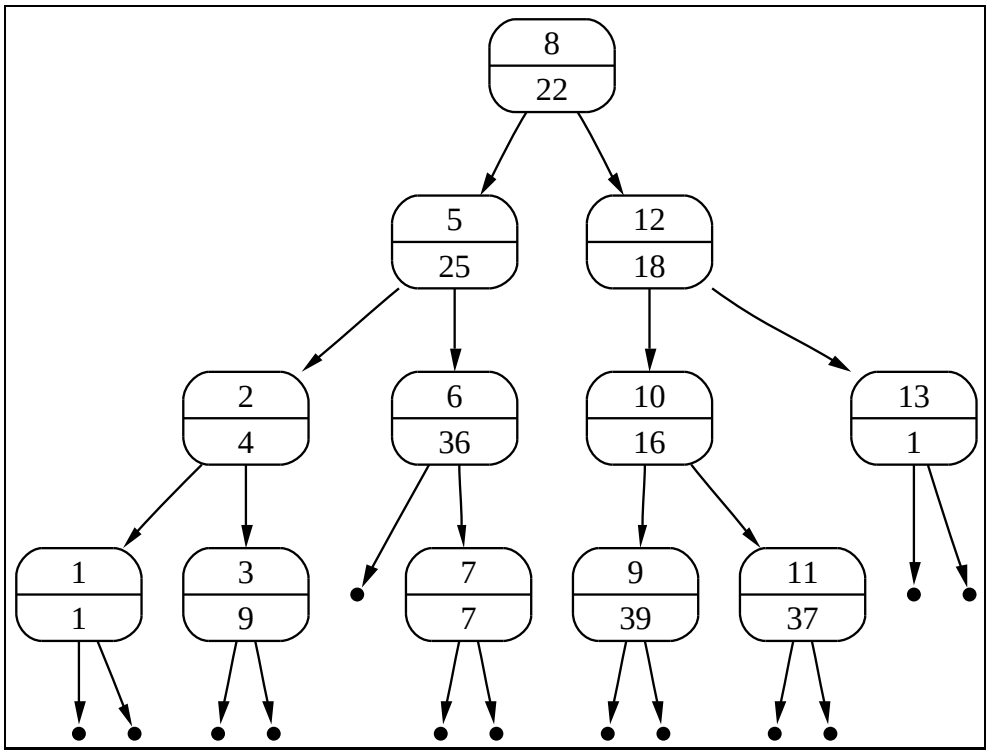

Abbildung 6.3: Der geordnete binärer Baum aus Abbildung 6.2 nach dem Entfernen des Knotens mit der Markierung $\langle 4, 16\rangle$.

1. $\mathit{node}(k, v, \mathit{nil}, r).\mathit{delMin}() = [r, k, v]$,

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

   denn wenn der linke Teilbaum l in dem binären Baum $t = \mathit{node}(k, v, l, r)$ nicht leer ist, so muss der kleinste Schlüssel von t in l liegen. Wir entfernen daher rekursiv den kleinsten Schlüssel aus l und erhalten dabei den Baum l'. In dem Baum ursprünglich gegebenen Baum t ersetzen wir l durch l' und erhalten $t = \mathit{node}(k, v, l', r)$.

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

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

   Falls der zu entfernende Schlüssel mit dem Schlüssel an der Wurzel des Baums übereinstimmt, entfernen wir mit dem Aufruf $r.\mathit{delMin}()$ den kleinsten Schlüssel aus dem rechten Teilbaum r und produzieren dabei den Baum r'. Gleichzeitig berechnen wir dabei für den rechten Teilbaum den kleinsten Schlüssel k_{min} und den diesem Schlüssel zugeordneten Wert v_{min}. Diese Werte setzen wir nun an die Wurzel des neuen Baums.

5. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node\big(k_2, v_2, l.delete(k_1), r\big)$,

   Falls der zu entfernende Schlüssel kleiner ist als der Schlüssel an der Wurzel, so kann sich der Schlüssel nur im linken Teilbaum befinden. Daher wird der Schlüssel k_1 rekursiv in dem linken Teilbaum l entfernt.

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

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

6.1.1 Implementierung geordneter binärer Bäume in *Java*

```
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 6.4: Das Interface *MyMap*.

```
1  public class BinaryTree<Key extends Comparable<Key>, Value>
2      1implements MyMap<Key, Value>
3  {
4      Node<Key, Value> mRoot;
5
6      public BinaryTree() {
7          mRoot = new EmptyNode<Key, Value>();
8      }
9
10     public Value find(Key key) {
11         return mRoot.find(key);
12     }
13
14     public void insert(Key key, Value value) {
15         mRoot = mRoot.insert(key, value);
16     }
17
18     public void delete(Key key) {
19         mRoot = mRoot.delete(key);
20     }
21
22     // Transform the tree into a sorted list.
23     public  LinkedList<Key> toList() {
24         return mRoot.toList();
25     }
26 }
```

Abbildung 6.5: Die Klasse *BinaryTree*.

Abbildung 6.4 auf Seite 97 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 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 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 6.6: Die abstrakte Klasse *Node*.

Abbildung 6.5 auf Seite 97 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 6.6 auf Seite 98 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 Klasee *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 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 6.7 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 Abbildungen 6.8 und 6.9 auf den Seiten 100 und 101 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.

```
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
10     public Node<Key, Value> insert(Key key, Value value) {
11         return new BinaryNode<Key, Value>(key, value);
12     }
13
14     public Node<Key, Value> delete(Key key) {
15         return this;
16     }
17
18     public boolean isEmpty() {
19         return true;
20     }
21
22     public LinkedList<Key> toList() {
23         return new LinkedList<Key>();
24     }
25
26     Triple<Node<Key, Value>, Key, Value> delMin() {
27         throw new UnsupportedOperationException();
28     }
29 }
```

Abbildung 6.7: Die Klasse *EmptyNode*.

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

$$node(o.\texttt{mKey}, o.\texttt{mValue}, o.\texttt{mLeft}, o.\texttt{mRight}).$$

Die Implementierung der Methoden *find()*, *insert()* und *delete()* setzt die bedingten Gleichungen, mit denen wir diese Methoden spezifiziert haben, eins-zu-eins um.
Abbildung 6.10 auf Seite 102 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.

6.1.2 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 6.11 zeigt den geordneten binären Baum der dann entsteht, wenn die Paare aus Schlüssel und Werten aus der Abbildung 6.2 in aufsteigender Reihenfolge eingegeben werden. Wird hier nach dem größten Schlüssel gesucht, so muß der komplette Baum durchlaufen werden. Enthält der Baum n Knoten, 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

```
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     }
15
16     public Value find(Key key) {
17         int cmp = key.compareTo(mKey);
18         if (cmp < 0) {                // key < mKey
19             return mLeft.find(key);
20         } else if (cmp > 0) {         // key > mKey
21             return mRight.find(key);
22         } else {                      // key == mKey
23             return mValue;
24         }
25     }
26
27     public Node<Key, Value> insert(Key key, Value value) {
28         int cmp = key.compareTo(mKey);
29         if (cmp < 0) {                          // key < mKey
30             mLeft = mLeft.insert(key, value);
31         } else if (cmp > 0) {                   // key > mKey
32             mRight = mRight.insert(key, value);
33         } else {                                // key == mKey
34             mValue = value;
35         }
36         return this;
37     }
```

Abbildung 6.8: Die Klasse *BinaryNode*, Teil `I`.

Aufruf von `find()` für einen Baum mit n Knoten durchschnittlich $\mathcal{O}\big(n * \ln(n)\big)$ Vergleiche erforderlich sind. Die Lage ist hier ganz ähnlich wie beim *Quick-Sort*-Algorithmus. Wir werden diese Behauptung nun beweisen.

Wir bezeichnen mit d_n die Anzahl von Vergleichen, die beim Aufruf $b.\mathit{find}(k)$ für einen geordneten binären Baum b durchgeführt werden müssen, falls b insgesamt n Knoten enthält. Wir wollen außerdem 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

$$d_0 = 0$$

ist, denn um zu sehen, dass der Schlüssel k nicht im leeren Baum auftritt, ist kein Vergleich erforderlich. Wir betrachten nun einen binären Baum b, der $n+1$ Knoten enthält. b hat die Form

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

```
38      public Node<Key, Value> delete(Key key) {
39          int cmp = key.compareTo(mKey);
40          if (cmp == 0) {
41              if (mLeft.isEmpty()) {
42                  return mRight;
43              }
44              if (mRight.isEmpty()) {
45                  return mLeft;
46              }
47              Triple<Node<Key, Value>, Key, Value> triple = mRight.delMin();
48              mRight = triple.getFirst();
49              mKey   = triple.getSecond();
50              mValue = triple.getThird();
51          }
52          if (cmp < 0) {
53              mLeft = mLeft.delete(key);
54          }
55          if (cmp > 0) {
56              mRight = mRight.delete(key);
57          }
58          return this;
59      }
60
61      public boolean isEmpty() {
62          return false;
63      }
64
65      public LinkedList<Key> toList() {
66          LinkedList<Key> result = mLeft.toList();
67          result.addLast(mKey);
68          result.addAll(mRight.toList());
69          return result;
70      }
71
72      Triple<Node<Key, Value>, Key, Value> delMin() {
73          if (mLeft.isEmpty()) {
74              return new
75                  Triple<Node<Key, Value>, Key, Value>(mRight, mKey, mValue);
76          } else {
77              Triple<Node<Key, Value>, Key, Value> t = mLeft.delMin();
78              mLeft = t.getFirst();
79              Key   key   = t.getSecond();
80              Value value = t.getThird();
81              return new
82                  Triple<Node<Key, Value>, Key, Value>(this, key, value);
83          }
84      }
85  }
```

Abbildung 6.9: Die Klasse *BinaryNode*, Teil II.

```
86  class Triple<First, Second, Third>
87  {
88      First  mFirst;
89      Second mSecond;
90      Third  mThird;
91
92      Triple(First first, Second second, Third third) {
93          mFirst  = first;
94          mSecond = second;
95          mThird  = third;
96      }
97
98      First  getFirst()  { return mFirst;  }
99      Second getSecond() { return mSecond; }
100     Third  getThird()  { return mThird;  }
101 }
```

Abbildung 6.10: Die Klasse *Triple*.

Da b aus $n+1$ Knoten besteht, gibt es insgesamt $n+1$ Möglichkeiten, wie die verbleibenden n Knoten auf die beiden Teilbäume l und r verteilt sein können, denn l kann i Knoten enthalten, wobei

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

gilt. Entsprechend enthält r dann $n-i$ Knoten. Betrachten wir nun einen dieser Fälle genauer: Falls l aus i Knoten besteht und die restlichen $n-i$ Knoten in r liegen, so gibt es für den Schlüssel k, nach dem wir in dem Aufruf $b.\mathit{find}(k)$ suchen, wiederum $n+1$ Möglichkeiten:

1. k kann mit dem Schlüssel an der Wurzel des Baums übereinstimmen. In diesem Fall führen wir nur einen Vergleich durch.

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 l auftritt, den Wert
   $$\frac{i}{n+1}.$$
   Also werden im Durchschnitt in diesem Fall
   $$\frac{i}{n+1} * d_i$$
   Vergleiche durchgeführt. Dazu kommt noch der Vergleich mit dem Schlüssel an der Wurzel.

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 r auftritt, den Wert
   $$\frac{n-i}{n+1}.$$
   Also werden im Durchschnitt in diesem Fall
   $$\frac{n-i}{n+1} * d_i$$
   Vergleiche durchgeführt. Dazu kommt noch der Vergleich mit dem Schlüssel an der Wurzel.

Damit können wir nun die Rekurrenz-Gleichung aufstellen:

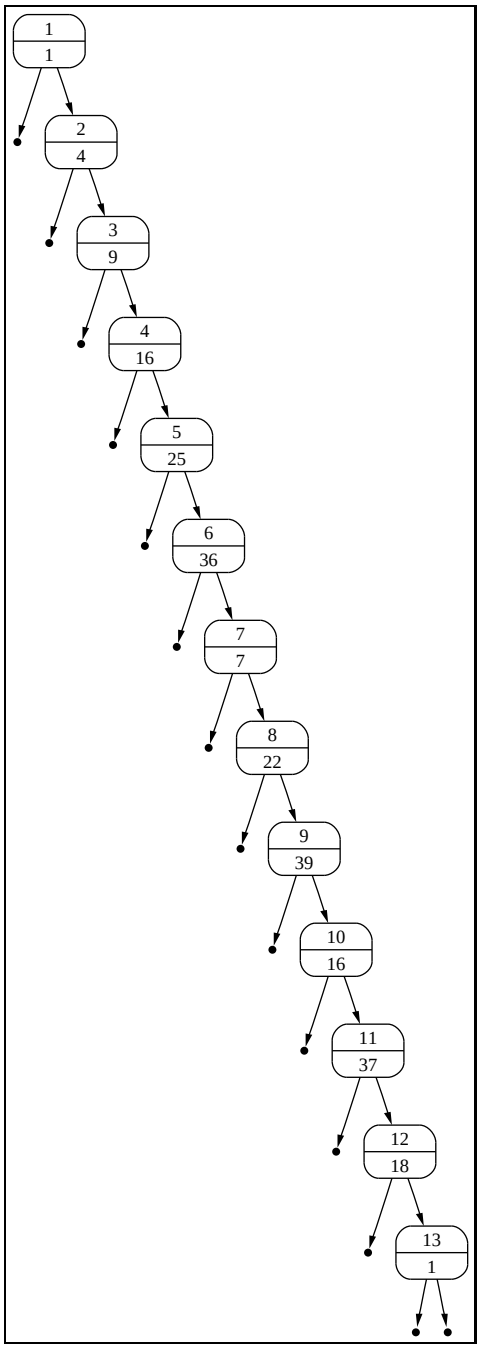

Abbildung 6.11: Ein entarteter geordneter binärer Baum.

$$
\begin{aligned}
d_{n+1} &= 1 + \frac{1}{n+1}\left(\sum_{i=0}^{n}\left(\frac{i}{n+1} * d_i + \frac{n-i}{n+1} * d_{n-i}\right)\right) \\
&= 1 + \frac{1}{(n+1)^2}\left(\sum_{i=0}^{n}\left(i * d_i + (n-i) * d_{n-i}\right)\right) \\
&= 1 + \frac{2}{(n+1)^2}\sum_{i=0}^{n} i * d_i
\end{aligned}
$$

Bei der letzten Umformung haben wir die Gleichung

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

benutzt. Wir lösen jetzt die Rekurrenz-Gleichung

$$d_{n+1} = 1 + \frac{2}{(n+1)^2} \sum_{i=0}^{n} i * d_i \tag{1}$$

mit der Anfangs-Bedingungen $d_0 = 0$. Zur Lösung von (1) führen wir die Substitution $n \mapsto n+1$ durch und erhalten

$$d_{n+2} = 1 + \frac{2}{(n+2)^2} \sum_{i=0}^{n+1} i * d_i \tag{2}$$

Wir multiplizieren nun (1) mit $(n+1)^2$ und (2) mit $(n+2)^2$ und finden

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

$$(n+2)^2 * d_{n+2} = (n+2)^2 + 2 * \sum_{i=0}^{n+1} i * d_i \tag{2'}$$

Subtrahieren wir (1′) von (2′) so erhalten wir

$$(n+2)^2 * d_{n+2} - (n+1)^2 * d_{n+1} = (n+2)^2 - (n+1)^2 + 2 * (n+1) * d_{n+1}.$$

Dies vereinfachen wir zu

$$\begin{aligned} (n+2)^2 * d_{n+2} &= \big((n+1)^2 + 2 * (n+1)\big) * d_{n+1} + (n+2)^2 - (n+1)^2 \\ &= (n+1) * (n+3) * d_{n+1} + (n+2)^2 - (n+1)^2 \end{aligned}$$

Zur Vereinfachung substituieren wir hier $n \mapsto n-1$ und erhalten

$$(n+1)^2 * d_{n+1} = n * (n+2) * d_n + 2 * n + 1.$$

Bei dieser Gleichung teilen wir auf beiden Seiten durch $(n+2) * (n+1)$ und bekommen

$$\frac{n+1}{n+2} * d_{n+1} = \frac{n}{n+1} * d_n + \frac{2*n+1}{(n+2)*(n+1)}.$$

Nun definieren wir

$$c_n = \frac{n}{n+1} d_n.$$

Dann gilt $c_0 = \frac{0}{1} d_0 = 0$ und wir haben die Rekurrenz-Gleichung

$$c_{n+1} = c_n + \frac{2*n+1}{(n+2)*(n+1)}.$$

Durch Partialbruch-Zerlegung finden wir

$$\frac{2*n+1}{(n+2)*(n+1)} = \frac{3}{n+2} - \frac{1}{n+1}.$$

Also haben wir

$$c_{n+1} = c_n + \frac{3}{n+2} - \frac{1}{n+1}.$$

Diese Rekurrenz-Gleichung können wir mit dem Teleskop-Verfahren lösen. Wegen $c_0 = 0$ gilt:

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

Wir substituieren $n \mapsto n-1$ und vereinfachen dies zu

$$\begin{aligned} c_n &= \sum_{i=2}^{n+1} \frac{3}{i} - \sum_{i=1}^{n} \frac{1}{i} \\ &= \sum_{i=1}^{n} \frac{3}{i} - \frac{3}{1} + \frac{3}{n+1} - \sum_{i=1}^{n} \frac{1}{i} \\ &= \sum_{i=1}^{n} \frac{2}{i} - 3 + \frac{3}{n+1} \\ &= 2 * H_n - 3 * \frac{n}{n+1} \end{aligned}$$

Wegen $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln(n) + \mathcal{O}(1)$ gilt

$$c_n = 2 * \ln(n) + \mathcal{O}(1).$$

Berücksichtigen wir $d_n = \frac{n+1}{n} c_n$, so finden wir für große n ebenfalls

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

Das ist unser zentrales Ergebnis: Im Durchschnitt erfordert das Suchen in einem zufällig erzeugten geordneten binären Baum für große Werte von n etwa $2 * \ln(n)$ Vergleiche. Ähnliche Ergebnisse können wir für das Einfügen oder Löschen erhalten.

6.2 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 Knoten abhängt. Eine solche Variante sind die *AVL-Bäume*, die wir jetzt vorstellen werden. Dazu definieren wir zunächst die *Höhe* eines binären Baums:

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

Definition 39 (AVL-Baum)
Wir definieren die Menge $\mathcal{A}$ der *AVL-Bäume* induktiv:

1. $\mathit{nil} \in \mathcal{A}$.
2. $\mathit{node}(k, v, l, r) \in \mathcal{A}$ g.d.w.
   (a) $\mathit{node}(k, v, l, r) \in \mathcal{B}_<$,
   (b) $l, r \in \mathcal{A}$ und
   (c) $|l.\mathit{height}() - r.\mathit{height}()| \leq 1$.
   Diese Bedingungen bezeichnen wir auch als die *Balancierungs-Bedingung*.

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

Um AVL-Bäume zu implementieren, können wir auf unserer Implementierung der geordneten binären Bäume aufsetzen. Neben den Methoden, die wir schon aus der Klasse *Map* kennen, brauchen wir noch die Methode

$$\mathit{restore} : \mathcal{B}_< \to \mathcal{A},$$

mit der wir die Bedingung über den Höhenunterschied von Teilbäumen wiederherstellen können, wenn diese beim Einfügen oder Löschen eines Elements verletzt wird. Der Aufruf $b.restore()$ 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. 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. $$\begin{array}{ll} & l_1.height() = r_1.height() + 2 \\ \wedge & l_1 = node(k_2, v_2, l_2, r_2) \\ \wedge & l_2.height() \geq r_2.height() \\ \rightarrow & node(k_1, v_1, l_1, r_1).restore() = node\big(k_2, v_2, l_2, node(k_1, v_1, r_2, r_1)\big) \end{array}$$

   Um diese Gleichung zu verstehen, betrachten wir Abbildung 6.12 auf Seite 106. Der linke Teil der Abbildung beschreibt die Situation vor dem Ausbalancieren, es wird also der Baum

   $$node(k_1, v_1, node(k_2, v_2, l_2, r_2), r_1)$$

   dargestellt. Der rechte Teil der Abbildung zeigt das Ergebnis des Ausbalancierens, es wird also der Baum

   $$node\big(k_2, v_2, l_2, node(k_1, v_1, r_2, r_1)\big)$$

   dargestellt. Wir haben hier die Höhen der einzelnen Teilbäume jeweils in die zweiten Zeilen der entsprechenden Markierungen geschrieben. Hier ist h die Höhe des Teilbaums l_2. Der Teilbaum r_1 hat die Höhe $h - 1$. Der Teilbaum r_2 hat die Höhe h' und es gilt $h' \leq h$. Da r_2 ein AVL-Baum ist, gilt also entweder $h' = h$ oder $h' = h - 1$.

   Die gezeigte Situation kann entstehen, wenn im linken Teilbaum l_2 ein Element eingefügt wird oder wenn im rechten Teilbaum r_1 eine Element gelöscht wird.

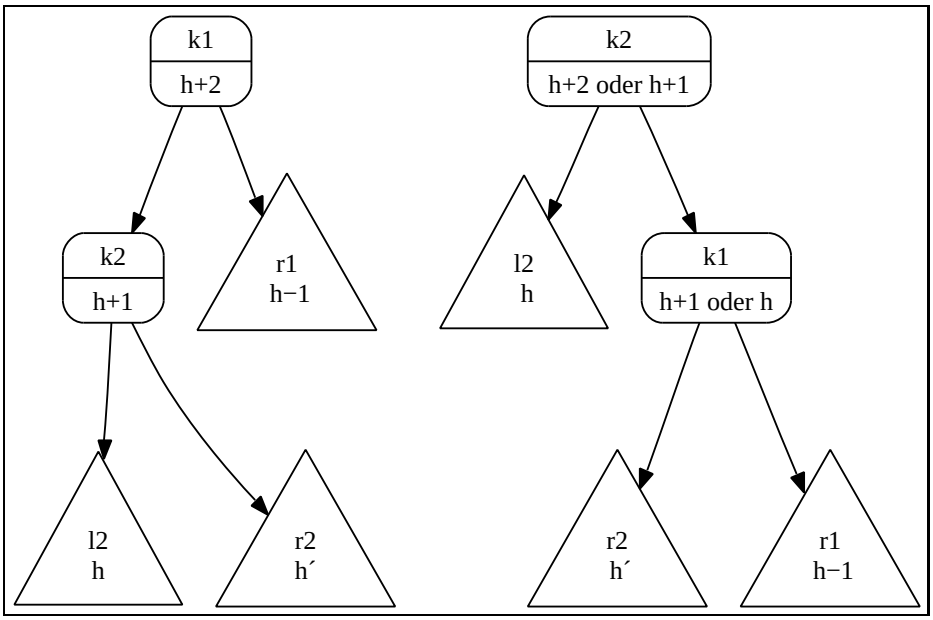

Abbildung 6.12: Ein unbalancierter Baum und der rebalancierte Baum

Wir müssen uns davon überzeugen, dass der im rechten Teil von Abbildung 6.12 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:

$$l_2 < k_2 < r_2 < k_1 < r_1. \tag{$\star$}$$

Dabei schreiben wir für einen Schlüssel k und einen Baum b

$$k < b$$

um auzudrücken, dass k kleiner ist als alle Schlüssel, die in dem Baum b vorkommen. Analog schreiben wir $b < k$ wenn alle Schlüssel, die in dem Baum b vorkommen, kleiner sind als der Schlüssel k. Die Ungleichung ($\star$) beschreibt die Anordnung der Schlüssel sowohl für den im linken Teil der Abbildung gezeigten Baum als auch für den Baum im rechten Teil der Abbildung und damit sind beide Bäume geordnet.

4. $$\begin{array}{ll} & l_1.\mathit{height}() = r_1.\mathit{height}() + 2 \\ \wedge & l_1 = \mathit{node}(k_2, v_2, l_2, r_2) \\ \wedge & l_2.\mathit{height}() < r_2.\mathit{height}() \\ \wedge & r_2 = \mathit{node}(k_3, v_3, l_3, r_3) \\ \rightarrow & \mathit{node}(k_1, v_1, l_1, r_1).\mathit{restore}() = \mathit{node}\big(k_3, v_3, \mathit{node}(k_2, v_2, l_2, l_3), \mathit{node}(k_1, v_1, r_3, r_1)\big) \end{array}$$

   Die linke Seite der Gleichung wird durch die Abbildung 6.13 auf Seite 108 illustriert. Dieser Baum kann in der Form

   $$\mathit{node}\big(k_1, v_1, \mathit{node}(k_2, v_2, l_2, \mathit{node}\big(k_3, v_3, l_3, r_3)\big), r_1\big)$$

   geschrieben werden. Die Teilbäume l_3 und r_3 haben hier entweder die Höhe h oder $h-1$, wobei mindestens einer der beiden Teilbäume die Höhe h haben muß.

   Die Situation der rechten Seite der obigen Gleichung zeigt Abbildung 6.14 auf Seite 108. Der auf dieser Abbildung gezeigte Baum hat die Form

   $$\mathit{node}\big(k_3, v_3, \mathit{node}(k_2, v_2, l_2, l_3), \mathit{node}(k_1, v_1, r_3, r_1)\big).$$

   Die Ungleichung, die die Anordnung der Schlüssel sowohl im linken als auch rechten Baum wieder gibt, lautet

   $$l_2 < k_2 < l_3 < k_3 < r_3 < k_1 < r_1.$$

   Es gibt noch zwei weitere Fälle die auftreten, wenn der rechte Teilbaum um mehr als Eins größer ist als der linke Teilbaum. Diese beiden Fälle sind aber zu den beiden vorherigen Fällen völlig analog, so dass wir die Gleichungen hier ohne weitere Diskussion angeben.

5. $$\begin{array}{ll} & r_1.\mathit{height}() = l_1.\mathit{height}() + 2 \\ \wedge & r_1 = \mathit{node}(k_2, v_2, l_2, r_2) \\ \wedge & r_2.\mathit{height}() \geq l_2.\mathit{height}() \\ \rightarrow & \mathit{node}(k_1, v_1, l_1, r_1).\mathit{restore}() = \mathit{node}\big(k_2, v_2, \mathit{node}(k_1, v_1, l_1, l_2), r_2\big) \end{array}$$

6. $$\begin{array}{ll} & r_1.\mathit{height}() = l_1.\mathit{height}() + 2 \\ \wedge & r_1 = \mathit{node}(k_2, v_2, l_2, r_2) \\ \wedge & r_2.\mathit{height}() < l_2.\mathit{height}() \\ \wedge & l_2 = \mathit{node}(k_3, v_3, l_3, r_3) \\ \rightarrow & \mathit{restore}\big(\mathit{node}(k_1, v_1, l_1, r_1)\big) = \mathit{node}\big(k_3, v_3, \mathit{node}(k_1, v_1, l_1, l_3), \mathit{node}(k_2, v_2, r_3, r_2)\big) \end{array}$$

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. $\mathit{nil}.\mathit{insert}(k, v) = \mathit{node}(k, v, \mathit{nil}, \mathit{nil})$.
2. $\mathit{node}(k, v_2, l, r).\mathit{insert}(k, v_1) = \mathit{node}(k, v_1, l, r)$.

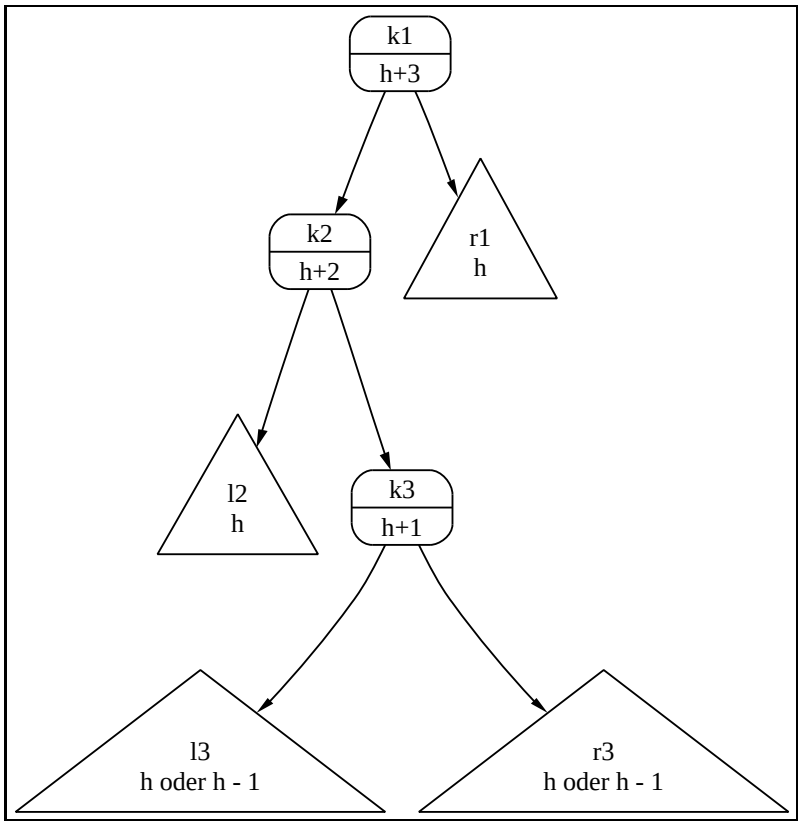

Abbildung 6.13: Ein unbalancierter Baum: 2. Fall

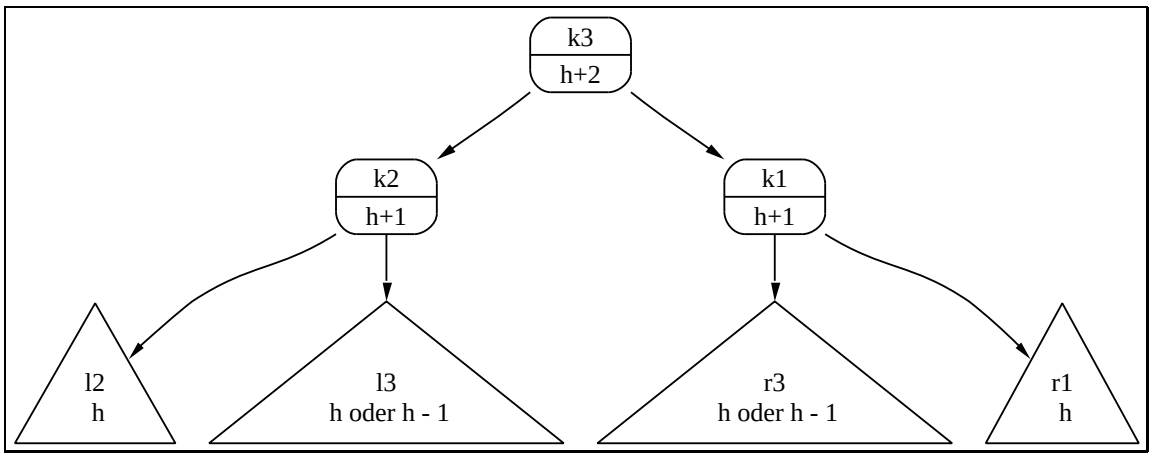

Abbildung 6.14: Der rebalancierte Baum im 2. Fall
1

3. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).insert(k_1, v_1) = node\big(k_2, v_2, l.insert(k_1, v_1), r\big).restore()$.
4. $k_1 > k_2 \rightarrow node(k_2, v_2, l, r).insert(k_1, v_1) = node\big(k_2, v_2, l, r.insert(k_1, v_1)\big).restore()$.

Analog ändern sich die Gleichungen für *delMin*() wie folgt:

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

$$node(k, v, l, r).delMin() = [node(k, v, l', r).restore(), k_{min}, v_{min}].$$

Damit können wir die Gleichungen zur Spezifikation der Methode `delete`() angeben.

1. $nil.delete(k) = nil$.
2. $node(k, v, nil, r).delete(k) = r$.
3. $node(k, v, l, nil).delete(k) = l$.
4. $l \neq nil \wedge r \neq nil \wedge r.delMin() = [r', k_{min}, v_{min}] \rightarrow$
   $node(k, v, l, r).delete(k) = node(k_{min}, v_{min}, l, r').restore()$.
5. $k_1 < k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node\big(k_2, v_2, l.delete(k_1), r\big).restore()$.
6. $k_1 > k_2 \rightarrow node(k_2, v_2, l, r).delete(k_1) = node\big(k_2, v_2, l, r.delete(k_1)\big)$.

6.2.1 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 6.15: Die Klasse *AVLTree*.

Abbildung 6.15 auf Seite 109 zeigt die Implementierung der Klasse *AVLTree*. Gegenüber der Implementierung der Klasse *BinaryTree* aus Abbildung 6.5 auf Seite 97 wurde hier nur der Name der Klasse geändert.

Abbildung 6.16 auf Seite 110 zeigt die neue Implementierung der Klasse *Node*. Gegenüber der Implementierung in Abbildung 6.6 auf Seite 98 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.

Abbildung 6.17 auf Seite 110 zeigt die Implementierung der Klasse *EmptyNode*. Gegenüber der in Abbildung 6.7 auf Seite 99 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.

```
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 6.16: Die abstrakte Klasse *Node*.

```
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() { return; }
23 }
```

Abbildung 6.17: Die Klasse *EmptyNode*.

Die Abbildungen 6.18, 6.19 und 6.20 auf Seite 111 und den folgenden Seiten zeigen die Implementierung der Klasse *BinaryNode*. Gegenüber der entsprechenden Implementierung in den Abbildungen 6.8 und 6.9 auf den Seiten 100 und 101 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.

3. Am Ende der Methode *insert*() wird in Zeile 43 die Methode *restore*() aufgerufen um die

```
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 6.18: Die Klasse *BinaryNode*, Teil I.

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,

```
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 6.19: Die Klasse *BinaryNode*, Teil `II`.

die Methode am Ende der `if`-Blöcken 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.

4. Genauso wird in Zeile 66 vor der Beendigung der Methode *delete*() die Methode *restore*() aufgerufen.

5. In Zeile 80 wird *restore*() aufgerufen um sicherzustellen, dass der von *delMin*() erzeugte binärer Baum die Balancierungs-Bedingung nicht verletzt.

6. In Zeile 84 implementieren wir die Methode *restore*().

```
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         restoreHeight();
118     }
119     void restoreHeight() {
120         mHeight = 1 + Math.max(mLeft.mHeight, mRight.mHeight);
121     }
122 }
123
```

Abbildung 6.20: Die Klasse *BinaryNode*, Teil `III`.

(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 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 6.12 gezeigt ist, während die Zeilen 104 – 112 den Fall behandeln, der in den Abbildungen 6.13 und 6.14 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.

(d) In Zeile 118 wird am Ende der Methode *restore*() noch dafür gesorgt, dass die Member-Variable `mHeight` an dem Knoten aktualsisiert wird.

6.2.2 Analyse der Komplexität

Wir analysieren jetzt die Komplexität von AVL-Bäumen im schlechtesten Fall. Der schlechteste Fall tritt dann ein, wenn bei einer vorgegebenen Zahl von Schlüsseln die Höhe maximal wird. Das ist aber das selbe wie wenn in einem Baum gegebener Höhe die Zahl der Schlüssel mininmal wird. Wir definieren daher c_h als die minimale Zahl von Schlüsseln, die ein AVL-Baum der Höhe h haben kann. Dann gilt:

1. $c_0 = 0$,

   denn es gibt genau einen AVL-Baum der Höhe 0 und dieser enthält keinen Schlüssel.

2. $c_1 = 1$,

   denn es gibt genau einen AVL-Baum der Höhe 1 und dieser enthält einen Schlüssel.

3. $c_{h+2} = c_{h+1} + c_h + 1$,

   denn um einen AVL-Baum der Höhe $h + 2$ mit einer minimalen Anzahl an Schlüsseln zu konstruieren, erzeugen wir zunächst einen AVL-Baum l der Höhe $h + 1$ mit einer minimalen Anzahl an Schlüsseln, desweiteren erzeugen wir einen AVL-Baum r der Höhe h mit minimaler Anzahl an Schlüsseln und bilden dann den Baum $node(k, v, l, r)$, der die Höhe $h + 2$ hat und dessen Anzahl an Schlüsseln minimal ist. Dieser Baum hat aber genau einen Schlüssel mehr als die Bäume l und r zusammen.

Wir haben jetzt also die Rekurrenz-Gleichung $c_{h+2} = c_{h+1} + c_h + 1$ mit den Anfangs-Bedingungen $c_0 = 0$ und $c_1 = 1$ zu lösen. Im ersten Kapitel hatten wir die Rekurrenz-Gleichung

$$a_{n+2} = a_{n+1} + a_n + 1, \quad \text{mit } a_0 = 0,\ a_1 = 0$$

gelöst. Wegen $a_2 = a_1 + a_0 + 1 = 0 + 0 + 1 = 1$ gibt es zwischen c_h und a_n den Zusammenhang

$$c_h = a_{h+1}.$$

Wir hatten damals die Lösung $a_n = \dfrac{1}{\sqrt{5}} * \left(\lambda_1^{n+1} - \lambda_2^{n+1}\right) - 1$

gefunden. Setzen wir hier für n den Wert $h + 1$ ein, so erhalten wir

$$c_h = \frac{1}{\sqrt{5}} \left(\lambda_1^{h+2} - \lambda_2^{h+2}\right) - 1 \quad \text{mit } \lambda_1 = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62 \text{ und } \lambda_2 = \frac{1}{2}(1 - \sqrt{5}) \approx -0.62.$$

Da $|\lambda_2| < 1$ ist, spielt der Wert λ_2^h für große Werte von h praktisch keine Rolle und die minimale Zahl n der Schlüssel in einem Baum der Höhe h wird durch

$$n \approx \frac{1}{\sqrt{5}} \lambda_1^{h+2}$$

gegeben. Lösen wir diese Gleichung nach h auf, so finden wir für große n

$$h = \frac{\ln(n) + \ln(\sqrt{5})}{\ln(\lambda_1)} - 2 = \frac{\ln(n)}{\ln(\lambda_1)} + \mathcal{O}(1) = \frac{\ln(2)}{\ln(\lambda_1)} * \log_2(n) + \mathcal{O}(1) \approx 1.44 \log_2(n) + \mathcal{O}(1),$$

denn es gilt allgemein für beliebige positive Zahlen a, b und c die Gleichung

$$\frac{\ln(a)}{\ln(b)} = \log_b(a).$$

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 6.21 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 6.11

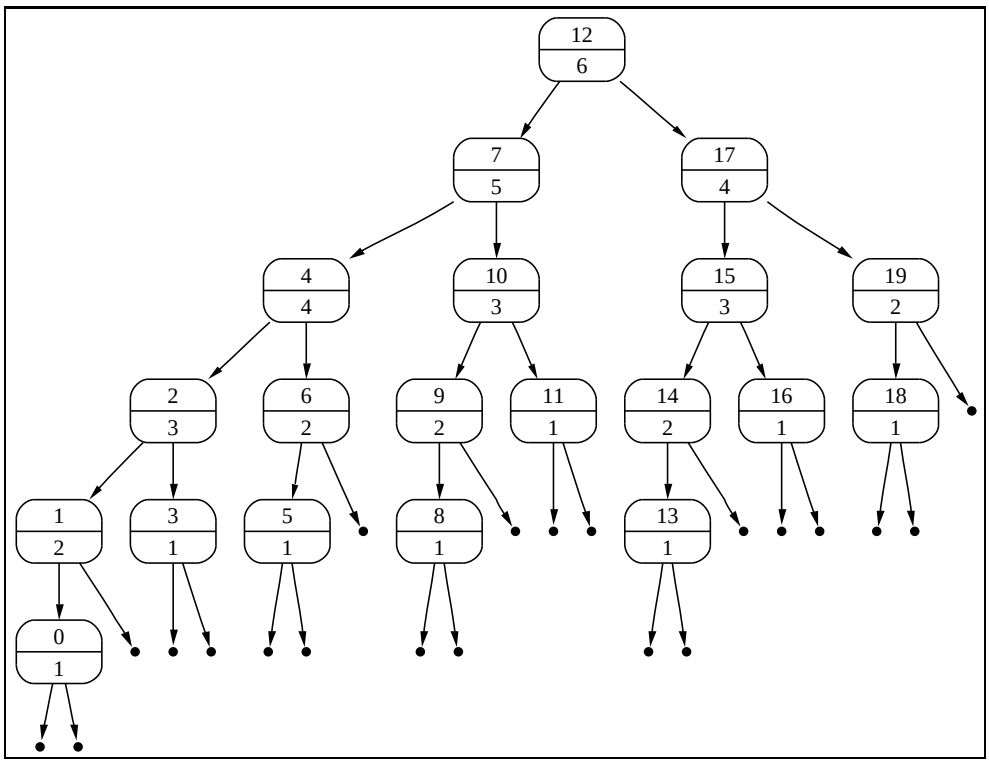

Abbildung 6.21: Ein AVL-Baum mit dem ungünstigsten Verhältnis von Höhe zur Anzahl an Knoten

Aufgabe:

1. Spezifizieren Sie mit Hilfe bedingter Gleichungen eine Methode
   $$\mathit{maxKey} : \mathcal{B}_< \rightarrow \mathit{Key} \cup \{\Omega\}$$
   die für einen gegebenen nicht-leeren binären b Baum den größten Schlüssel berechnet, der in b enthalten ist.

2. Spezifizieren Sie mit Hilfe bedingter Gleichungen eine Funktion
   $$\mathit{skewedTree} : \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{A}$$
   so dass der Aufruf $\mathit{skewedTree}(h, k)$ einen AVL-Baum b mit folgenden Eigenschaften berechnet:

   (a) Der Baum b hat die Höhe h.

   (b) Die in b enthaltenen Schlüssel sind natürliche Zahlen.

   (c) Falls b nicht der leere Baum ist, hat der kleinste Schlüssel in b den Wert k.

   (d) Die Anzahl der in b enthaltenen Schlüssel ist minimal.

6.3 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.

Die Grundidee bei der Datenstruktur *Trie* ist eine 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 c und r wie folgt aus dem String w gewinnen:
   $$c = w.\mathit{charAt}(0); \quad \text{und} \quad r = w.\mathit{substring}(1);$$

3. ε bezeichnet das leere Wort. In *Java* können wir schreiben
   $$\texttt{epsilon} = \text{""};$$

4. *Value* ist eine Menge von *Werten*.

Die Menge $\mathbb{T}$ der Tries definieren wir nun induktiv mit Hilfe des Konstruktors
$$\mathit{node} : \mathit{Value} \times \mathit{List}(\Sigma) \times \mathit{List}(\mathbb{T}) \rightarrow \mathbb{T}.$$
Die induktive Definition besteht nur aus einer einzigen Klausel. Falls

1. $v \in \mathit{Value} \cup \{\Omega\}$

2. $L = [c_1, \cdots, c_n] \in \mathit{List}(\Sigma)$ eine Liste von Buchstaben der Länge n ist,

3. $T = [t_1, \cdots, t_n] \in \mathbb{T}$ eine Liste von Tries der selben Länge n ist,

dann gilt

$$\mathit{node}(v, L, T) \in \mathbb{T}.$$

Als erstes fragen Sie sich vermutlich, wo bei dieser induktiven Definition der Induktions-Anfang ist. Der Induktions-Anfang ist der Fall, in dem die Listen L und T leer sind.

Als nächstes überlegen wir uns, welche Funktion von dem Trie

$$\mathit{node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]) \in \mathbb{T}$$

dargestellt wird. Wir beantworten diese Frage, indem wir rekursive Gleichungen für die Methode

$$\mathit{find} : \mathbb{T} \times \Sigma^* \to \mathit{Value} \cup \{\Omega\}$$

angeben. Wir werden den Ausdruck $\mathit{node}(v, L, T).\mathit{find}(s)$ durch Induktion über den String s definieren:

1. $\mathit{node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{find}(\varepsilon) = v.$

   Der dem leeren String zugeordnete Wert wird also unmittelbar an der Wurzel des Tries abgespeichert.

2. $$\mathit{node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{find}(cr) = \begin{cases} t_1.\mathit{find}(r) & \text{falls} \quad c = c_1; \\ \vdots & \\ t_i.\mathit{find}(r) & \text{falls} \quad c = c_i; \\ \vdots & \\ t_n.\mathit{find}(r) & \text{falls} \quad c = c_n; \\ \Omega & \text{falls} \quad c \notin \{c_1, \cdots, c_n\}. \end{cases}$$

   Der Trie $\mathit{node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$ enthält genau dann einen Wert zu dem Schlüssel cr, wenn einerseits c in der Buchstaben-Liste an der Stelle i auftritt und wenn andererseits der Trie t_i einen Wert zu dem Schlüssel r enthält.

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

$$\mathit{node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$$

darzustellen, zeichnen wir einen Kreis, den wir durch einen horizontalen Strich in der Mitte aufteilen. Falls v von Ω verschieden ist, schreiben wir den Wert v in die untere Hälfte des Kreises. (Bei den in Abbildung 6.22 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 $t_1, \cdots, t_n$ darstellen. Außerdem markieren wir die diese Knoten darstellenden Kreise in den oberen Hälfte mit den Buchstaben $c_1, \cdots, c_n$.

Zur Verdeutlichung geben wir ein Beispiel in Abbildung 6.22 auf Seite 118. Die Funktion, die hier dargestellt wird, läßt sich wie folgt als binäre Relation schreiben:

$$\big\{\langle \text{“Stahl”}, 1\rangle, \langle \text{“Stolz”}, 2\rangle, \langle \text{“Stoeger”}, 3\rangle, \langle \text{“Salz”}, 4\rangle, \langle \text{“Schulz”}, 5\rangle,$$
$$\langle \text{“Schulze”}, 6\rangle, \langle \text{“Schnaut”}, 7\rangle, \langle \text{“Schnupp”}, 8\rangle, \langle \text{“Schroer”}, 9\rangle\big\}.$$

Der Wurzel-Knoten ist hier leer, denn dieser Knoten hat keinen Vater-Knoten, von dem er eine Markierung erben könnte. Diesem Knoten entspricht der Term

$$\mathit{node}(\Omega, [\text{‘S’}], [t]).$$

Dabei bezeichnet t den Trie, dessen Wurzel mit dem Buchstaben ‘S’ markiert ist. Diesen Trie können wir seinerseits durch den Term

$$\mathit{node}(\Omega, [\text{‘t’}, \text{‘a’}, \text{‘c’}], [t_1, t_2, t_3])$$

darstellen. Daher hat dieser Knote drei Söhne, die mit den Buchstaben ‘t’, ‘a’ und ‘c’ markiert sind.

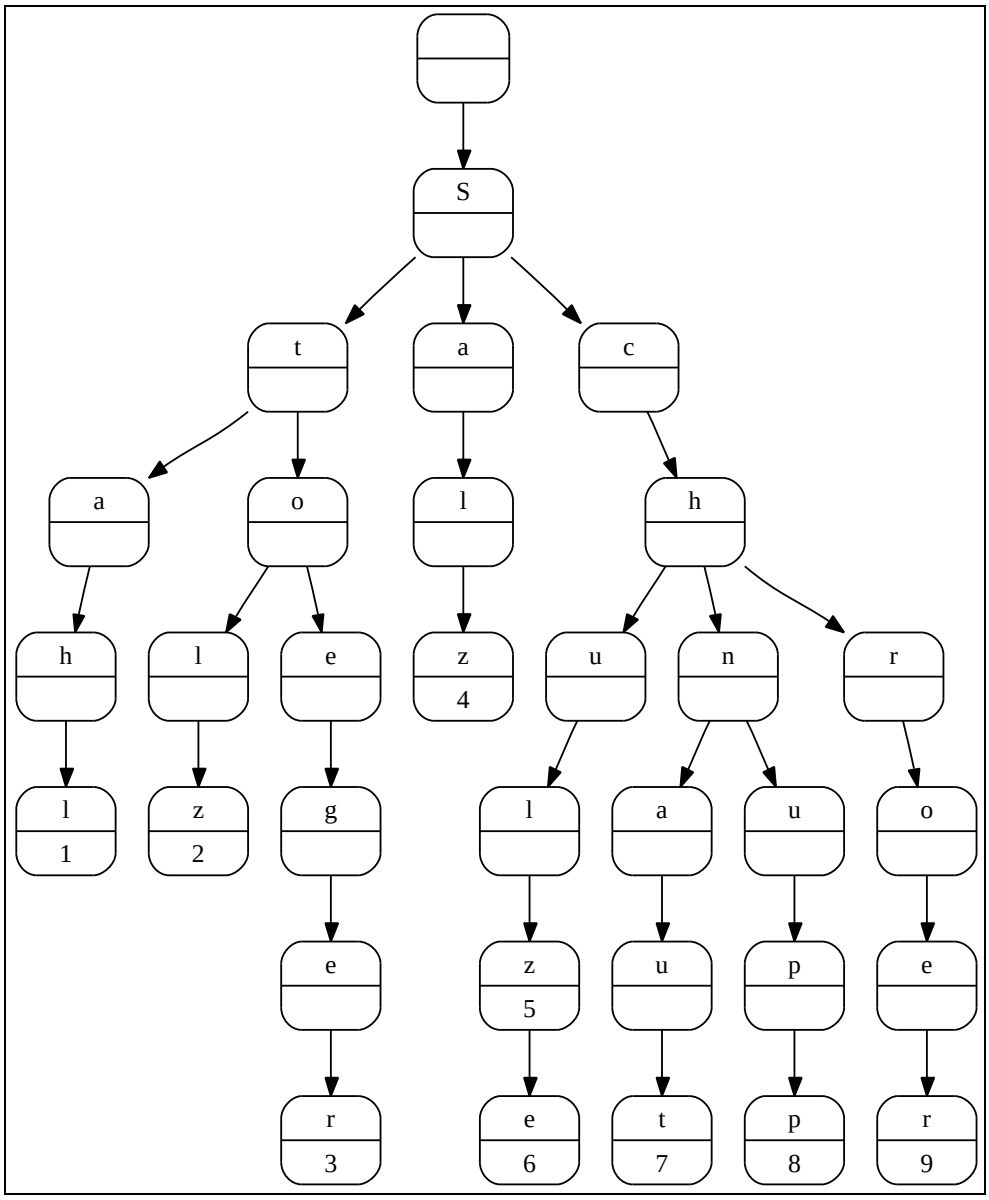

Abbildung 6.22: Ein Beispiel Trie

6.3.1 Einfügen in Tries

Wir stellen nun bedingte Gleichungen auf, mit denen wir das Einfügen eines Schlüssels mit einem zugehörigen Wert beschreiben können. Wir definieren den Wert von

$$node(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).insert(s, v)$$

für ein Wort $w \in \Sigma^*$ und einen Wert $v \in V$ durch Induktion nach der Länge des Wortes w.

1. $node(v_1, L, T).insert(\varepsilon, v_2) = node(v_2, L, T)$,

   Einfügen eines Wertes mit dem leeren String als Schlüssel überschreibt also einfach den an dem Knoten gespeicherten Wert.

2. $node\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big).insert(c_i r, v_2) =$
   $node\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i.insert(r, v_2), \cdots, t_n]\big).$

Wenn in dem Trie $node\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big)$ ein Wert v_2 zu dem Schlüssel cr eingefügt werden soll, und falls der Buchstabe c in der Liste $[c_1, \cdots, c_n]$ an der Stelle i vorkommt, wenn also gilt $c = c_i$, dann muß der Wert v_2 rekursiv in dem Trie t_i unter dem Schlüssel r eingefügt werden.

3. $c \notin \{c_1, \cdots, c_n\} \;\rightarrow\; node\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big).insert(cr, v_2) =$
   $node\big(v_1, [c_1, \cdots, c_n, c], [t_1, \cdots, t_n, node(\Omega, [], []).insert(r, v_2)]\big).$

   Wenn in dem Trie $node\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big)$ ein Wert v_2 zu dem Schlüssel cr eingefügt werden soll, und falls der Buchstabe c in der Liste $[c_1, \cdots, c_n]$ nicht vorkommt, dann wird zunächst ein Trie erzeugt, der die leere Abbildung repräsentiert. Dieser Trie hat die Form

   $$node(\Omega, [], []).$$

   Anschließend wird in diesem Trie der Wert v_2 rekursiv unter dem Schlüssel r eingefügt. Zum Schluß hängen wir den Buchstaben c an die Liste $c_1, \cdots, c_n]$ and und fügen den Trie

   $$node(\Omega, [], []).insert(r, v_2)$$

   am Ende der Liste $[t_1, \cdots, t_n]$ ein.

6.3.2 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

$$isEmpty : \mathbb{T} \rightarrow \mathbb{B},$$

so dass $t.isEmpty()$ genau dann `true` liefert, wenn der Trie t die leere Funktion darstellt. Wir definieren also:

1. $node(\Omega, [], []).isEmpty() = \texttt{true}$
2. $v \neq \Omega \rightarrow node(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).isEmpty() = \texttt{false}$
3. $n \neq 0 \wedge \big(\exists \in \{1, \cdots, n\} : t_i.isEmpty() = \texttt{false}\big) \rightarrow$
   $node(\Omega, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).isEmpty() = \texttt{false}$
4. $n \neq 0 \wedge \big(\forall \in \{1, \cdots, n\} : t_i.isEmpty() = \texttt{true}\big) \rightarrow$
   $node(\Omega, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).isEmpty() = \texttt{true}$

Nun können wir den Wert von

$$t.delete(w)$$

für einen Trie $t \in \mathbb{B}$ und ein Wort $w \in \Sigma^*$ durch Induktion nach der Länge des Wortes w definieren.

1. $node(v, L, T).delete(\varepsilon) = node(\Omega, L, T),$

   denn der Wert, der unter dem leeren String ε in einem Trie gespeichert wird, befindet sich unmittelbar an der Wurzel des Tires und kann dort sofort gelöscht werden.

2. $$\begin{array}{ll} c = c_i & \wedge \\ t_i.delete(r).isEmpty() & \rightarrow \\ node(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]).delete(cr) & = \\ \quad node(v, [c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n], [t_1, \cdots, c_{i-1}, c_{i+1}, \cdots, t_n]). & \end{array}$$

   Wenn der zu löschende String mit dem Buchstaben c_i anfängt, und wenn das Löschen des Schlüssels r in dem i-ten Trie t_i einen leeren Trie ergibt, dann streichen wir den i-ten Buchstaben und den dazu korrespondierenden i-ten Trie t_i.

3. $c = c_i \quad \wedge$
   $t_i' = t_i.delete(r) \quad \wedge$
   $\neg\, t_i'.isEmpty() \quad \wedge$
   $node(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]).delete(cr) \quad =$
   $node(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i', \cdots, t_n]).$

   Wenn der zu löschende String mit dem Buchstaben c_i anfängt, und wenn der Baum t_i, der durch das Löschen des Schlüssels r in dem i-ten Trie t_i entsteht nicht leer ist, dann ersetzen wir t_i durch t_i'.

4. $c \notin \{c_1, \cdots, c_n\} \quad \rightarrow$
   $node(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).delete(cr) \quad =$
   $node(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).$

   Wenn der zu löschende String mit dem Buchstaben c anfängt und wenn der Buchstabe c gar kein Element der Buchstaben-Liste $[c_1, \cdots, c_n]$ ist, unter der in diesem Trie Informationen abgespeichert sind, dann verändert das Löschen den Trie nicht.

6.3.3 Implementierung in *Java*

Wir zeigen nun, wie sich die Tries in *Java* implementieren lassen. Die Abbildungen 6.23 und 6.24 auf den Seiten 121 und 122 zeigen die Implementierung, die wir jetzt diskutieren.

1. Den Trie $node(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$ 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 $[c_1, \cdots, c_n]$.

   (c) `mNodeList` entspricht der Trie-Liste $[t_1, \cdots, t_n]$.

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 schauen wieder, ob wir dabei den Buchstaben c finden. Wenn wir den Buchstaben 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, indem

```
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 6.23: Die Klasse *TrieNode-I*, Teil `I`.

wir 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*.

(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 schauen wieder, ob wir dabei den Buchstaben c finden. Wenn wir den Buchstaben 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              }
62          }
63      }
64      public Boolean isEmpty() {
65          return mValue == null && mNodeList.size() == 0;
66      }
67  }
```

Abbildung 6.24: Die Klasse *TrieNode*, Teil `II`.

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) $l, r \in BT$.

Die Semantik legen wir fest, indem wir eine Methode

$$find : BT \times \mathbb{N} \to Value \cup \{\Omega\}$$

definieren. Für einen binären Trie b und eine natürliche Zahl n gibt $b.find(n)$ den Wert zurück, der unter dem Schlüssel n in dem binären Trie b gespeichert ist. Fall in dem binären Trie b unter

dem Schlüssel n kein Wert gespeichert ist, wird Ω zurück gegeben. Formal definieren wir den Wert von $b.find(n)$ durch Induktion nach dem Aufbau von b definiert. Im Induktions-Schritt ist eine Neben-Induktion nach n erforderlich.

1. $nil.find(n) = \Omega$.
2. $bin(v, l, r).find(0) = v$.
3. $2 * n > 0 \rightarrow bin(v, l, r).find(2*n) = l.find(n)$.
4. $bin(v, l, r).find(2*n+1) = r.find(n)$.

Aufgabe:

1. Stellen Sie Gleichungen auf, die das Einfügen und das Löschen in einem binären Trie beschreiben.
2. Für Leute mit sportlichem Ehrgeiz: Implementieren Sie binäre Tries in *Java*.

Bemerkung: Binäre Tries werden auch als *digitale Suchbäume* bezeichnet.

6.4 Hash-Tabellen

Eine Abbildung

$$f : Key \rightarrow Value$$

kann dann sehr einfach implementiert werden, wenn

$$Key = \{0, 1, 2, \cdots, n\},$$

denn dann reicht es aus, ein Feld der Größe $n + 1$ zu verwenden. Abbildung 6.25 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 6.25: Die Klasse *ArrayMap*.

Falls nun der Definitions-Bereich D der darzustellenden Abbildung nicht die Form einer Menge der Gestalt $\{1, \cdots, n\}$ hat, könnten wir versuchen, D zunächst auf eine Menge der Form $\{1, \cdots, n\}$ abzubilden. Wir erläutern diese Idee an einem einfachen Beispiel. Wir betrachten eine naive Methode um ein Telefon-Buch abzuspeichern:

1. Wir machen zunächst die Annahme, dass alle Namen aus genau 8 Buchstaben bestehen. Dazu werden kürzere Namen mit Blanks aufgefüllt und Namen die länger als 8 Buchstaben sind, werden nach dem 8-ten Buchstaben abgeschnitten.

2. Als nächstes übersetzen wir Namen in einen Index. Dazu fassen wir die einzelnen Buchstaben als Ziffern auf, die die Werte von 0 bis 26 annehmen können. Dem Blank ordnen wir dabei den Wert 0 zu. Nehmen wir an, dass die Funktion *ord* jedem Buchstaben aus der Menge $\Sigma = \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\}$ einen Wert aus der Menge $\{0, \cdots, 26\}$ zuordnet

   $$ord : \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\} \to \{0, \cdots, 26\},$$

   so läßt sich der Wert eines Strings $w = c_0c_1 \cdots c_7$ durch eine Funktion

   $$code : \Sigma^* \to \mathbb{N}$$

   berechnen, die wie folgt definiert ist:

   $$code(c_0c_1 \cdots c_7) = \sum_{i=0}^{7} ord(c_i) * 27^i.$$

   Die Menge *code* bildet die Menge aller Wörter mit 8 Buchstaben bijektiv auf die Menge der Zahlen $\{0, \cdots, 27^8 - 1\}$ 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 eines Maschinen-Wortes hat und ein Maschinen-Wort aus 4 Byte besteht, so bräuchten wir ca. 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:

   $$code(c_0c_1 \cdots c_n) = \left(\sum_{i=0}^{n} ord(c_i) * 27^i\right) \ \% \ size.$$

   Um eine Überlauf zu vermeiden, können wir für $k = n, n-1, \cdots, 1, 0$ die Teilsummen s_k wie folgt induktiv definieren:

   (a) $s_n = ord(c_n)$

   (b) $s_k = (ord(c_k) + s_{k+1} * 27) \ \% \ size$

   Dann gilt $$s_0 = \left(\sum_{i=0}^{n} ord(c_i) * 27^i\right) \ \% \ size.$$

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 6.26 auf Seite 125 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*.

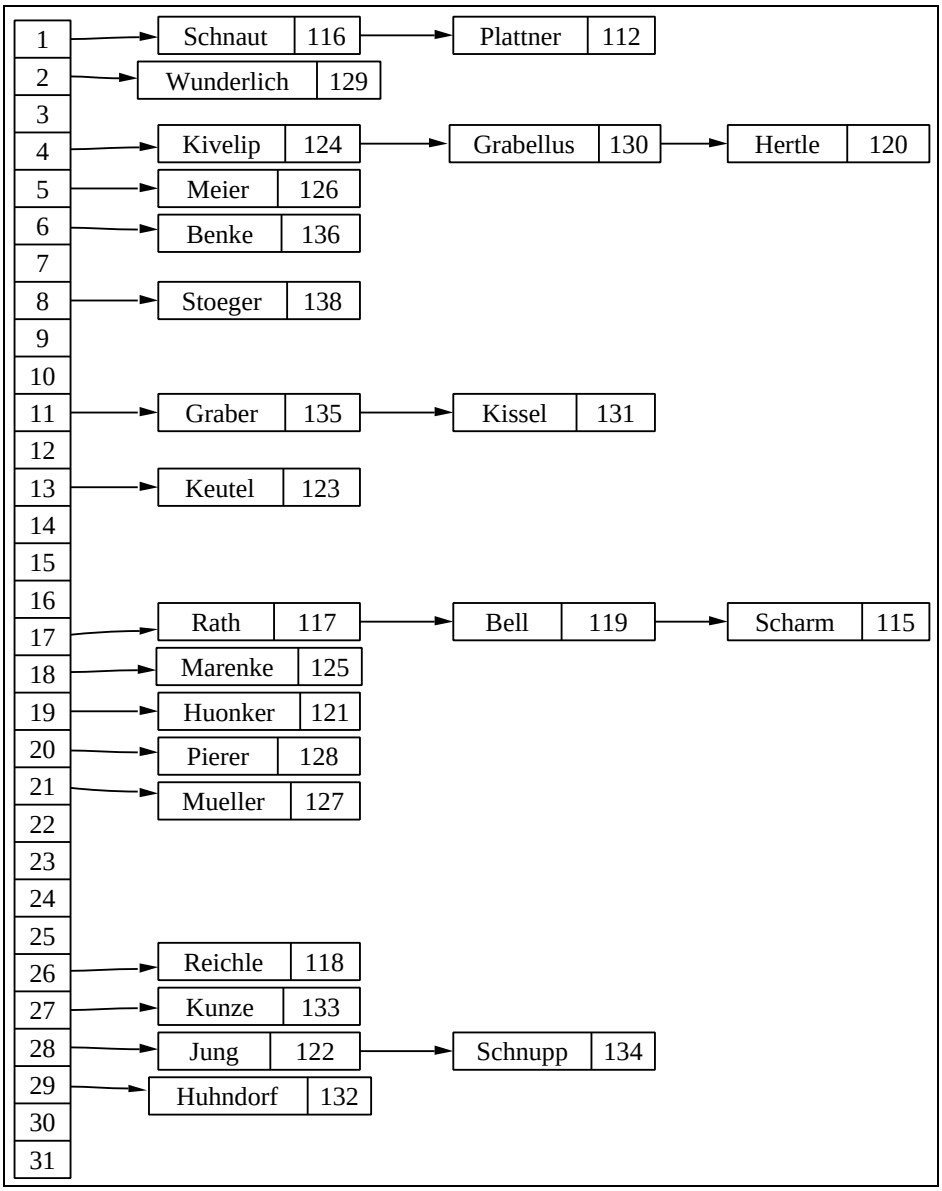

Abbildung 6.26: Eine Hash-Tabelle

1. Als erstes überlegen wir uns, welche Daten-Strukturen wir brauchen, um ein 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

```
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 6.27: Die Klasse *MyHashMap*, Teil `I`.

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.

Aus Gründen, die wir später noch diskutieren, sollte die Größe der Tabelle eine Primzahl sein. 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 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 initialsiert `mPrimeIndex` mit dem gegebenen Argument. Wird

```
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 6.28: Die Klasse *MyHashMap*, Teil `II`.

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 $\{\, 0, \cdots, \texttt{mArray.length} - 1 \,\}$ 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.

   Andernfalls laufen wir mit einer Schleife durch die Liste durch und vergleichen die einzelnen

```
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 6.29: Die Klasse *MyHashMap*, Teil `III`.

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.

4. 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. 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. Anschließende 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 eine 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 durchschnittlich Länge einer Liste ist dann

$$\alpha = \frac{\texttt{count}}{\texttt{size}}.$$

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`.

Kapitel 7

Prioritäts-Warteschlangen

Prioritäts-Warteschlangen spielen in vielen Bereichen der Informatik eine wichtige Rolle. Wir werden Prioritäts-Warteschlangen später in dem Kapitel über Graphen-Theorie einsetzen. Daneben werden Prioritäts-Warteschlangen unter anderem in Simulations-Systemen und beim Scheduling von Prozessen in Betriebs-Systemen eingesetzt.

Um den Begriff der *Prioritäts-Warteschlange* zu verstehen, betrachten wir zunächst den Begriff der *Warteschlange*. Dort werden Daten hinten eingefügt und vorne werden Daten entnommen. Das führt dazu, dass Daten in der selben Reihenfolge entnommen werden, wie sie eingefügt werden. Anschaulich ist das so wie bei der Warteschlange vor einer Kino-Kasse, wo die Leute in der Reihenfolge bedient werden, in der sie sich anstellen. Bei einer Prioritäts-Warteschlange haben die Daten zusätzlich Prioritäten. Es wird immer das Datum entnommen, was die höchste Priorität hat. Anschaulich ist das so wie im Wartezimmer eines Zahnarztes. Wenn Sie schon eine Stunde gewartet haben und dann ein Privat-Patient aufkreuzt, dann müssen Sie halt noch eine Stunde warten, weil der Privat-Patient eine höhere Priorität hat.

7.1 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 40 (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
   $$\{Key, Value\}.$$
   Dabei muß auf der Menge *Key* eine totale Quasi-Ordnung $<$ existieren, so dass wir die Prioritäten verschiedener Elemente an Hand der Schlüssel vergleichen können.
3. Die Menge der Funktions-Zeichen ist
   $$\{PrioQueue, insert, remove, top, change\}.$$
4. Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:
   (a) $PrioQueue : PrioQueue$
   Der Aufruf "*new PrioQueue*()" erzeugt eine leere Prioritäts-Warteschlange.

(b) $top : PrioQueue \to (Key \times Value) \cup \{\Omega\}$
Der Aufruf $Q.top()$ liefert ein Paar $\langle k, v\rangle$. Dabei ist v das Element aus Q, das die höchste Priorität hat. k ist die Priorität des Elements v.

(c) $insert : PrioQueue \times Key \times Value \to PrioQueue$
Der Aufruf $Q.insert(k, v)$ fügt das Element v mit der Priorität k in die Prioritäts-Warteschlange Q ein.

(d) $remove : PrioQueue \to PrioQueue$
Der Aufruf $Q.remove()$ entfernt aus der Prioritäts-Warteschlange Q das Element, das die höchste Priorität hat.

(e) $change : PrioQueue \times Key \times Value \to PrioQueue$
Der Aufruf $Q.change(k, v)$ 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 dass Element v in der Prioritäts-Warteschlange Q auftritt und dass andererseits die neue Priorität mindestens so hoch ist wie die Priorität, die v vorher hatte.

5. Bevor wir das Verhalten der einzelnen Methoden axiomatisch definieren, müssen wir noch festlegen, was wir unter den *Prioritäten* verstehen wollen, die den einzelnen Elementen aus *Value* zugeordnet sind. Wir nehmen an, dass die Prioritäten Elemente einer Menge *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) $new\ PrioQueue() = \{\}$,
   der Konstruktor erzeugt also eine leere Prioritäts-Warteschlange, die als leere Menge dargestellt wird.

   (b) $Q.insert(k, v) = Q \cup \{\langle k, v\rangle\}$,
   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 $Q.top()$ undefiniert:
   $$Q = \{\} \ \to \ Q.top() = \Omega.$$

   (d) Wenn Q nicht leer ist, wenn es also ein Paar $\langle k_1, v_1\rangle$ in Q gibt, dann liefert $Q.top()$ ein Paar $\langle k_2, v\rangle$ aus der Menge Q, so dass $k_2 \leq k_1$ ist.
   $$\langle k_1, v_1\rangle \in Q \ \wedge \ Q.top() = \langle k_2, v_2\rangle \ \to \ k_2 \leq k_1 \ \wedge \ \langle k_2, v_2\rangle \in Q.$$

   (e) Falls Q leer ist, dann ändert $remove()$ nichts daran:
   $$Q = \{\} \to Q.remove() = Q.$$

   (f) Sonst entfernt $Q.remove()$ ein Paar, dessen Schlüssel minimal ist:
   $$Q \neq \{\} \to Q.remove() = Q \backslash \big\{Q.top()\big\}.$$

   (g) Die Methode $change()$ ändert die Priorität eines Paares:
   $$Q.change(k_1, v_1) = \big\{\langle k_2, v_2\rangle \in Q : v_2 \neq v_1\big\} \cup \big\{\langle k_1, v_1\rangle\big\}.$$

Wir können den abstrakten Daten-Typ *PrioQueue* dadurch implementieren, dass wir eine Prioritäts-Warteschlange durch eine Liste realisieren, in der die Elemente aufsteigend geordnet sind. Die einzelnen Operationen werden dann wie folgt implementiert:

1. *new PrioQueue*() erzeugt eine leere Liste.
2. $Q.\mathit{insert}(d)$ kann durch die Prozedur `insert` implementiert werden, die wir beim "*Sortieren durch Einfügen*" entwickelt haben.
3. $Q.\mathit{top}()$ gibt das erste Element der Liste zurück.
4. $Q.\mathit{remove}()$ entfernt das erste Element der Liste.
5. $Q.\mathit{change}(k, v)$ 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 $\mathcal{O}\big(\log(n)\big)$ hat. Dazu müssen wir eine neue Daten-Struktur einführen: *Heaps*.

7.2 Die Daten-Struktur *Heap*

Wir definieren die Menge *Heap* induktiv als Teilmenge der Menge $\mathcal{B}$ der binären Bäume. Dazu definieren wir zunächst für einen Schlüssel $k_1 \in \mathit{Key}$ 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. $k_1 \leq \mathit{nil}$,

   denn in dem leeren Baum treten überhaupt keine Schlüssel auf.

2. $k_1 \leq \mathit{node}(k_2, v, l, r) \stackrel{\text{def}}{\longleftrightarrow} k_1 \leq k_2 \wedge k_1 \leq l \wedge k_2 \leq r$,

   denn k_1 ist genau dann kleiner-gleich als alle Schlüssel, die in dem Baum $\mathit{node}(k_2, v, l, r)$ auftreten, wenn $k_1 \leq k_2$ gilt und wenn zusätzlich k_1 kleiner-gleich alle Schlüssel ist, die in l oder r auftreten.

Als nächstes definieren wir eine Funktion

$$\mathit{count} : \mathcal{B} \to \mathbb{N},$$

die für einen binären Baum die Anzahl der Knoten berechnet. Die Definition erfolgt durch Induktion:

1. $\mathit{nil}.\mathit{count}() = 0$.
2. $\mathit{node}(k, v, l, r).\mathit{count}() = 1 + l.\mathit{count}() + r.\mathit{count}()$.

Mit diesen Vorbereitungen können wir nun die Menge *Heap* induktiv definieren:

1. $\mathit{nil} \in \mathit{Heap}$.
2. $\mathit{node}(k, v, l, r) \in \mathit{Heap}$ 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*.

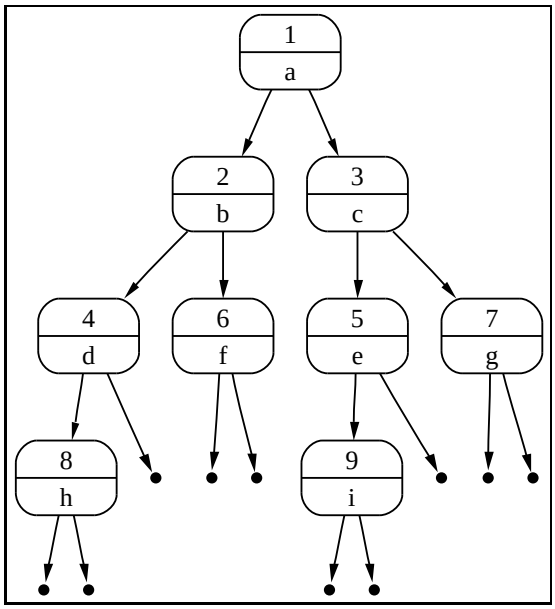

Abbildung 7.1: Ein Heap

(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 bezeichen wir als die *Balancierungs-Bedingung*. Sie ist ganz ähnlich zu der Balancierungs-Bedingung bei AVL-Bäumen, nur dass es dort die Höhe der Bäume ist, die verglichen wird, während wir hier die Zahl der im Baum gespeicherten Elemente vergleichen.

(c) $l \in \mathit{Heap} \wedge r \in \mathit{Heap}$.

Aus der *Heap-Bedingung* folgt, dass ein nicht-leerer Heap die Eigenschaft hat, dass das Element, das an der Wurzel steht, immer die höchste Priorität hat. Abbildung 7.1 auf Seite 133 zeigt einen einfachen Heap. In den Knoten zeigen wir oben die Prioritäten, die in diesem Fall natürliche Zahlen sind und darunter stehen die Werte, bei denen es sich jetzt um Buchstaben handeln.

Da Heaps binäre Bäume sind, können wir Sie ganz ähnlich wie geordnete binäre Bäume implementieren. Wir stellen zunächst Gleichungen auf, die die Implementierung der verschiedenen Methoden beschreiben. Wir beginnen mit der Methode *top*. Es gilt:

1. $\mathit{nil}.\mathit{top}() = \Omega$.

2. $\mathit{node}(k, v, l, r).\mathit{top}() = \langle k, v \rangle$,
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. $\mathit{nil}.\mathit{insert}(k, v) = \mathit{node}(k, v, \mathit{nil}, \mathit{nil})$.

2. $k_{\mathrm{top}} \leq k \;\wedge\; l.\mathit{count}() \leq r.\mathit{count}() \;\rightarrow$
$$\mathit{node}(k_{\mathrm{top}}, v_{\mathrm{top}}, l, r).\mathit{insert}(k, v) = \mathit{node}\big(k_{\mathrm{top}}, v_{\mathrm{top}}, l.\mathit{insert}(k, v), r\big).$$
Falls das einzufügende Paar eine geringere oder die selbe Priorität hat wie das Paar, welches sich an der Wurzel befindet, und falls zusätzlich die Zahl der Paare im linken Teilbaum kleiner-gleich der Zahl der Paare im rechten Teilbaum ist, dann fügen wir das Paar im linken Teilbaum ein.

3. $k_{\text{top}} \leq k \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
$$\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) = \mathit{node}\big(k_{\text{top}}, v_{\text{top}}, l, r.\mathit{insert}(k, v)\big).$$
Falls das einzufügende Paar eine geringere oder die selbe Priorität hat als das Paar an der Wurzel und falls zusätzlich die Zahl der Paare im linken Teilbaum größer als die Zahl der Paare im rechten Teilbaum ist, dann fügen wir das Paar im rechten Teilbaum ein.

4. $k_{\text{top}} > k \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
$$\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) = \mathit{node}\big(k, v, l.\mathit{insert}(k_{\text{top}}, v_{\text{top}}), r\big).$$
Falls das einzufügende Paar eine höhere Priorität hat als das Paar an der Wurzel, dann müssen wir das neu einzufügende Paar an der Wurzel positionieren. Das Paar, das dort vorher steht, fügen wir in den linken Teilbaum ein, falls die Zahl der Paare im linken Teilbaum kleiner-gleich der Zahl der Paare im rechten Teilbaum ist.

5. $k_{\text{top}} > k \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
$$\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) = \mathit{node}\big(k, v, l, r.\mathit{insert}(k_{\text{top}}, v_{\text{top}})\big).$$
Falls wir das einzufügende 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 das Paar, das vorher an der Wurzel stand, im rechten Teilbaum einfügen.

Als nächstes beschreiben wir die Implementierung der Methode *remove*.

1. $\mathit{nil}.\mathit{remove}() = \mathit{nil}$,
denn aus dem leeren Heap ist nichts mehr zu entfernen.

2. $\mathit{node}(k, v, \mathit{nil}, r).\mathit{remove}() = r$,

3. $\mathit{node}(k, v, l, \mathit{nil}).\mathit{remove}() = l$,
denn wir entfernen immer das Paar mit der höchsten Priorität und das ist an der Wurzel. Wenn einer der beiden Teilbäume leer ist, können wir einfach den anderen zurück geben.

   Jetzt betrachten wir die Fälle, wo keiner der beiden Teilbäume leer ist. Dann muss entweder das Paar an der Wurzel des linken Teilbaums oder das Paar an der Wurzel des rechten Teilbaums an die Wurzel aufrücken. Welches dieser beiden Paare wir nehmen hängt davon ab, welches die höhere Priorität hat.

4. $k_1 \leq k_2 \ \wedge \ l = \mathit{node}(k_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{node}(k_2, v_2, l_2, r_2) \ \rightarrow$
$$\mathit{node}(k, v, l, r).\mathit{remove}() = \mathit{node}(k_1, v_1, l.\mathit{remove}(), r),$$
denn wenn das Paar an der Wurzel des linken Teilbaums eine höhere Priorität hat als das Paar an der Wurzel des rechten Teilbaums, dann rückt dieses Paar an die Wurzel auf und muss folglich aus dem linken Teilbaum gelöscht werden.

5. $k_1 > k_2 \ \wedge \ l = \mathit{node}(k_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{node}(k_2, v_2, l_2, r_2) \rightarrow$
$$\mathit{node}(k, v, l, r).\mathit{remove}() = \mathit{node}(k_2, v_2, l, r.\mathit{remove}()),$$
denn wenn das Paar an der Wurzel des rechten Teilbaums eine höhere Priorität hat als das Paar an der Wurzel des linken Teilbaums, dann rückt dieses Paar an die Wurzel auf und muss folglich aus dem rechten Teilbaum gelöscht werden.

An dieser Stelle wird der aufmerksame Leser 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.

7.3 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 7.2: 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 7.2 auf Seite 135 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 7.3: Die Klasse *EmptyHeapNode*.

Abbildung 7.3 auf Seite 135 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 7.4: Die Klasse *BinaryHeapNode*, Teil I.

Die Klasse *BinaryHeapNode*, deren Implementierung in den Abbildungen 7.4 und 7.5 auf den Seiten 136 und 137 gezeigt wird, repräsentiert einen Knoten der Form

$$node(\texttt{mKey}, \texttt{mValue}, \texttt{mLeft}, \texttt{mRight}).$$

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 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 7.5: Die Klasse *BinaryHeapNode*, Teil II.

Aufgabe: Geben Sie eine Implementierung der Methode *remove*() an, welche die Balancierungs-Bedingung nicht verletzt. Definieren Sie dazu eine Methode

$$repair : BinaryHeapNode \rightarrow BinaryHeapNode,$$

die die Balancierungs-Bedingung an einem Knoten wieder herstellt, wenn Sie vorher dort durch eine *remove*-Operation verletzt worden ist. Sie können bei der Implementierung der Methode *repair*() also davon ausgehen, dass für den Knoten $node(k, v, l, r)$, auf den *repair*() angewendet wird, die Bedungung $\mid l.count() - r.count \mid \leq 2$ gilt.

Gehen Sie bei der Lösung der Aufgabe methodisch vor und entwickeln Sie zunächst rekursive Gleichungen, die das Verhalten der Methoden *remove*() und *repair*() beschreiben. Setzen Sie dann diese Gleichungen in *Java* um.

Lösung: Die Methode *repair* kann durch die folgenden Gleichungen spezifiziert werden:

1. $|l.count() - r.count| \leq 1 \;\rightarrow\; node(k,v,l,r).repair() = node(k,v,l,r)$,

   denn wenn die Balancierungs-Bedingung nicht verletzt ist, ist nichts zu tun.

2. $l.count() = r.count + 2 \;\wedge\; l = node(k',v',l',r') \;\rightarrow$
   $node(k,v,l,r).repair() = node\big(k, v, l.remove(), r.insert(k',v')\big)$,

   denn wenn der linke Teilbaum l ein Paar zuviel enthält, dann entfernen wir ein Paar aus dem linken Teilbaum und fügen es im rechten Teilbaum r ein.

3. $r.count() = l.count + 2 \;\wedge\; r = node(k',v',l',r') \;\rightarrow$
   $node(k,v,l,r).repair() = node\big(k, v, l.insert(k',v'), r.remove()\big)$,

   denn wenn der rechte Teilbaum r ein Paar zuviel enthält, dann entfernen wir ein Paar aus dem rechten Teilbaum und fügen es im linken Teilbaum l ein.

Bei der Implementierung der Methode *remove*() müssen wir die letzten beiden Gleichungen wie folgt anpassen:

4'. $k_1 \leq k_2 \;\wedge\; l = node(k_1,v_1,l_1,r_1) \;\wedge\; r = node(k_2,v_2,l_2,r_2) \;\rightarrow$
$node(k,v,l,r).remove() = node(k_1,v_1,l.remove(),r).repair()$.

5.' $k_1 > k_2 \;\wedge\; l = node(k_1,v_1,l_1,r_1) \;\wedge\; r = node(k_2,v_2,l_2,r_2) \rightarrow$
$node(k,v,l,r).remove() = node(k_2,v_2,l,r.remove()).repair()$.

Hier wird also lediglich am Ende *repair*() aufgerufen um die Balancierungs-Bedingung sicher zu stellen. Abbildung 7.6 zeigt die Implementierung dieser Gleichungen in *Java*. Bei der Implementierung der Methode *remove*() hat sich fast nichts geändert, wir rufen nur in Zeile 38 die Methode *repair*() auf, um die Balancierungs-Bedingung wiederherzustellen. Die Implementierung der Methode *repair*() orientiert sich 1-zu-1 an den bedingten Gleichungen, die wir oben angegeben haben.

7.3.1 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 vergößert 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-Tabelle[1] die Zuordnung der Knoten zu den Werten speichern. Diese HashTabelle realisiert also eine Funktion

$$nodeMap : Value \rightarrow Heap,$$

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 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

```
67      public HeapNode<Key, Value> remove() {
68          --mCount;
69          if (mLeft.isEmpty()) {
70              return mRight;
71          }
72          if (mRight.isEmpty()) {
73              return mLeft;
74          }
75          BinaryHeapNode<Key, Value> left  = (BinaryHeapNode<Key, Value>) mLeft;
76          BinaryHeapNode<Key, Value> right = (BinaryHeapNode<Key, Value>) mRight;
77          Key leftKey  = left .mKey;
78          Key rightKey = right.mKey;
79          if (leftKey.compareTo(rightKey) < 0) {
80              mKey   = left.mKey;
81              mValue = left.mValue;
82              mLeft  = mLeft.remove();
83          } else {
84              mKey   = right.mKey;
85              mValue = right.mValue;
86              mRight = mRight.remove();
87          }
88          repair();
89          return this;
90      }
91      private void repair() {
92          if (Math.abs(mLeft.mCount - mRight.mCount) <= 1) {
93              return;
94          }
95          if (mLeft.mCount == mRight.mCount + 2) {
96              BinaryHeapNode<Key, Value> left =
97                  (BinaryHeapNode<Key, Value>) mLeft;
98              Key   key   = left.mKey;
99              Value value = left.mValue;
100             mLeft  = mLeft.remove();
101             mRight = mRight.insert(key, value);
102         } else if (mRight.mCount == mLeft.mCount + 2) {
103             BinaryHeapNode<Key, Value> right =
104                 (BinaryHeapNode<Key, Value>) mRight;
105             Key   key   = right.mKey;
106             Value value = right.mValue;
107             mLeft  = mLeft.insert(key, value);
108             mRight = mRight.remove();
109         }
110     }
```

Abbildung 7.6: Die Methoden *remove* und *repair*.

p gespeichert ist, eine höhere Priorität hat als der Vater-Knoten dieses Knotens. In diesem Fall müssen wir die beiden Knoten vertauschen.

[1] Wir 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.

```
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     public abstract String toString();
15 }
```

Abbildung 7.7: Die abstrakte Klasse *HeapNode*.

Die Abbildungen 7.7, 7.8, 7.9, 7.11, 7.11 und 7.12 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 7.7 auf Seite 140 gezeigt wird. Gegenüber der Implementierung in Abbildung 7.2 auf Seite 7.2 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 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-Variable `mNodeMap` gespeichert.

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 vorher dort stand, wird entweder in den linken Teilbaum `mLeft` oder in den rechten Teilbaum `mRight` eingefügt.

```
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
14     public Pair<Key, Value> top() {
15         return null;
16     }
17
18     public BinaryHeapNode<Key, Value> insert(Key key, Value value) {
19         BinaryHeapNode<Key, Value> binaryNode =
20             new BinaryHeapNode<Key, Value>(key, value, mParent, mNodeMap);
21         mNodeMap.put(value, binaryNode);
22         return binaryNode;
23     }
24
25     public HeapNode<Key, Value> remove() {
26         return this;
27     }
28
29     public void change(Key key, Value value) {}
30
31     public boolean isEmpty() {
32         return true;
33     }
34 }
```

Abbildung 7.8: Die Klasse *EmptyHeapNode*.

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 den Zeiger zum Vaterknoten umsetzen, denn sonst würde dieser immer noch auf einen Knoten zeigen, den wir löschen.

```
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 7.9: 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 7.10: 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;
106           }
107           Key   parentKey   = mParent.mKey;
108           Value parentValue = mParent.mValue;
109           if (parentKey.compareTo(mKey) <= 0) {
110               return;
111           }
112           mNodeMap.remove(mValue);
113           mNodeMap.remove(mParent.mValue);
114           mNodeMap.put(mValue, mParent);
115           mNodeMap.put(parentValue, this);
116           mParent.mKey   = mKey;
117           mParent.mValue = mValue;
118           mKey           = parentKey;
119           mValue         = parentValue;
120           mParent.upHeap();
121       }
122       public boolean isEmpty() {
123           return false;
124       }
125   }
```

Abbildung 7.11: Die Methoden *change* und *upheap*.

4. 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.

5. 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.

Wir betrachten als letztes die Klasse *HeapTree*, die in Abbildungen 7.12 auf Seite 145 gezeigt wird. Diese Klasse repräsentiert einen vollständigen *Heap*.

1. Der Wurzel-Knoten des Heaps wird in der Member-Variablen `mRoot` gepeichert.

```
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     public String toString() {
25         return mRoot.toString();
26     }
27 }
```

Abbildung 7.12: Die Klasse *HeapTree*.

2. 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.

3. 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 beispsielsweise 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.

Kapitel 8

Graphentheorie

Wir wollen zum Abschluß der Vorlesung wenigstens ein graphen-theoretisches Problem vorstellen: Das Problem der Berechnung kürzester Wege.

8.1 Die Berechnung kürzester Wege

Um das Problem der Berechnung kürzester Wege formulieren zu können, führen wir zunächst den Begriff des *gewichteten Graphen* ein.

Definition 41 (Gewichteter Graph) Ein *gewichteter Graph* ist ein Tripel $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ so dass gilt:

1. $\mathbb{V}$ ist eine Menge von *Knoten*.
2. $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ ist eine Menge von *Kanten*.
3. $\|\cdot\| : \mathbb{E} \to \mathbb{N}\backslash\{0\}$ ist eine Funktion, die jeder Kante eine positive *Länge* zuordnet. □

Ein *Pfad* P ist eine Liste der Form

$$P = [x_1, x_2, x_3, \cdots, x_n]$$

so dass für alle $i = 1, \cdots, n-1$ gilt:

$$\langle x_i, x_{i+1} \rangle \in \mathbb{E}.$$

Die Menge aller Pfade bezeichnen wir mit $\mathbb{P}$. Die Länge eines Pfads definieren wir als die Summe der Länge aller Kanten:

$$\big\|[x_1, x_2, \cdots, x_n]\big\| \;:=\; \sum_{i=1}^{n-1} \big\|\langle x_i, x_{i+1}\rangle\big\|.$$

Ist $p = [x_1, x_2, \cdots, x_n]$ ein Pfad so sagen wir, dass p den Knoten x_1 mit dem Knoten x_n verbindet. Die Menge alle Pfade, die den Knoten v mit dem Knoten w verbinden, bezeichnen wir als

$$\mathbb{P}(v, w) \;:=\; \big\{p \in \mathbb{P} \colon p = [x_1, x_2, \cdots, x_n] \wedge x_1 = v \wedge x_n = w\big\}.$$

Damit können wir nun das Problem der Berechnung kürzester Wege formulieren.

Definition 42 (Kürzeste-Wege-Problem) Gegeben sei ein gewichteter Graph $G = \langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ und ein Knoten $\texttt{source} \in \mathbb{V}$. Dann besteht das *kürzeste-Wege-Problem* darin, die folgende Funktion zu berechnen:

$$\texttt{sp} : \mathbb{V} \to \mathbb{N}$$
$$\texttt{sp}(v) \;:=\; \texttt{min}\big\{\|p\| \;:\; p \in \mathbb{P}(\texttt{source}, v)\big\}.$$ □

8.1.1 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

$$\texttt{dist} : \mathbb{V} \rightarrow \mathbb{N} \cup \{\Omega\}$$

definieren, die für einen Punkt $u \in \mathbb{V}$ eine obere Abschätzung des Abstandes zum Knoten *source* angibt, es soll also immer gelten:

$$dist(u) \neq \Omega \;\rightarrow\; \texttt{sp}(u) \leq dist(u).$$

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 $dist(u) = \Omega$. Anfangs ist die Funktion *dist*() also nur für den Knoten *source* definiert, es gilt

$$dist(source) = 0.$$

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 $dist(u) + \|\langle u, v\rangle\|$ hat. Falls bisher also $dist(v)$ undefiniert war, weil wir noch keinen Weg gefunden hatten, der *source* mit v verbindet, können wir

$$dist(v) := dist(u) + \|\langle u, v\rangle\|$$

setzen. Diese Zuweisung ist ebenfalls gültig wenn $dist(v)$ bereits definiert ist aber einen Wert hat, der größer als $dist(u) + \|\langle u, v\rangle\|$ ist. Wir fassen diese Überlegungen in den beiden *ASM*-Regeln[1] zusammen, die in Abbildung 8.1 dargestellt sind.

```
1   Rule Init
2       if  dist(source) = Ω
3       then
4           dist(source) := 0;
5       endif
6
7   Rule Run
8       if choose ⟨u, v⟩ ∈ 𝔼 satisfying
9           dist(u) ≠ Ω and (dist(v) = Ω or dist(u) + ‖⟨u, v⟩‖ < dist(v))
10      then
11          dist(v) := dist(u) + ‖⟨u, v⟩‖;
12      endif
```

Abbildung 8.1: ASM-Regeln zur Lösung des kürzeste-Wege-Problems.

1. Die erste Regel besagt, dass wir $dist(source)$ auf den Wert 0 setzen, wenn die Funktion *dist* an der Stelle *source* noch undefiniert ist. Diese Regel kann nur einmal ausgeführt werden, denn anschließend ist ja $dist(source)$ nicht mehr undefiniert.

2. Die zweite Regel benutzt das Konstrukt `choose`. Dieses Konstrukt hat allgemein die Form

   $$\texttt{choose}\ \langle x_1, \cdots, x_n\rangle \in M : F(x_1, \cdots, x_n)$$

   Hierbei sind $x_1, \cdots, x_n$ verschiedene Variablen, M ist eine Menge von n-Tupeln und $F(x_1, \cdots, x_n)$ 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 $\langle t_1, \cdots, t_n\rangle$ gibt, so dass die Formel $F(t_1, \cdots, t_n)$ wahr wird. In diesem Fall werden gleichzeitig die Variablen $x_1, \cdots, x_n$ mit den entsprechenden Elementen belegt.

[1] Der Begriff *ASM* steht für *Abstract State Machine*. Dieser Begriff wurde von Yuri Gurevich eingeführt und von Egon Börger zur Spezifikation und Verifikation von Algorithmen propagiert und weiterentwickelt.

Bei der zweiten Regel suchen wir über das `choose`-Konstrukt eine Kante $\langle u, v\rangle$, für die gilt:

(a) $dist(u)$ ist definiert, es gibt also einen Pfad von dem Knoten *source* zu dem Knoten u.

(b) $dist(v)$ ist undefiniert oder größer als $dist(u) + \|\langle u, v\rangle\|$.

Dann können wir die Abschätzung für den Abstand $dist(v)$ von dem Knoten *source* zu dem Knoten v zu dem Wert $dist(u) + \|\langle u, v\rangle\|$ 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 *dist*() an einem Punkt. Da der Werte-Bereich dieser Funktion aus natürlichen Zahlen besteht, geht das nur endlich oft.

8.1.2 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 Kanten ausgewählt werden, nicht weiter spezifiert. Wir verfeinern den Algorithmus indem wir über die Auswahl der Kanten Buch führen. 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 `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   dist(source) = Ω
3       then dist(source) := 0;
4            mode := scan;
5       endif
6
7   Rule Scan
8       if   mode = scan and choose u ∈ fringe
9       then
10          E      := edges(u);
11          fringe := fringe \ {u};
12          mode   := relabel;
13      endif
14
15  Rule Relabel
16      if        mode = relabel
17           and  choose ⟨u, v⟩ ∈ E satisfying
18                    dist(v) = Ω or dist(u) + ‖⟨u, v⟩‖ < dist(v);
19      then
20          dist(v) := dist(u) + ‖⟨u, v⟩‖;
21          fringe  := fringe ∪ {v};
22      else
23          mode    := scan;
24      endif
```

Abbildung 8.2: Verbesserte ASM-Regel zur Lösung des kürzeste-Wege-Problems.

$$edges : \mathbb{V} \rightarrow 2^{\mathbb{E}}$$

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

$$edges(u) = \{\langle u, v\rangle \colon \langle u, v\rangle \in \mathbb{E}\}.$$

Der Algorithmus läuft nun in drei Phasen ab.

1. In der *Initialisierungs*-Phase setzen wir $dist(source) := 0$.

2. In der *Scanning*-Phase wählen wir einen Knoten u aus der Menge *fringe* aus, entfernen ihn aus dieser Menge und setzen

   $$\mathcal{E} := edges(u).$$

   Ansschließend 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

   $$dist(v) = \Omega \quad \text{oder} \quad dist(u) + \|\langle u, v\rangle\| < dist(v)$$

   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 $dist(u)$ verkleinern können, wechseln wir wieder in die *Scanning*-Phase zurück.

Der Algorithmus bricht ab, wenn die Menge *fringe* leer wird. Abbildung 8.2 auf Seite 148 zeigt die Spezifikation dieses Algorithmus durch eine ASM.

8.1.3 Der 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 Dijkstra vorgeschlagenen Algorithmus 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 8.3 auf Seite 150 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, ich habe die Variable *Visited* nur eingeführt um die Korrektheit des Algorithmus nachweisen zu können, denn es gilt die folgende Invariante:

$$\forall u \in \texttt{Visited} : dist(u) = sp(u)$$

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 $dist(u) = sp(u)$ 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 $sp(source) = 0$ 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

```
1   Rule Init
2       if   dist(source) = Ω
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          ⟨d, u⟩ := fringe.top();
15          fringe.remove();
16          Visited := Visited ∪ { u };
17          E      := edges(u);
18          mode   := relabel;
19      else
20          mode   := scan;
21      endif
22
23  Rule Relabel
24      if        mode = relabel
25           and  choose ⟨u, v⟩ ∈ E satisfying
26                    dist(v) = Ω or dist(u) + ‖⟨u, v⟩‖ < dist(v);
27      then
28          dist(v) := dist(u) + ‖⟨u, v⟩‖;
29          fringe  := fringe.insert(⟨dist(v), v⟩);
30      else
31          mode    := scan;
32      endif
```

Abbildung 8.3: ASM-Regeln für den Algorithmus von Dijkstra.

Visited enthalten ist, denn sonst wird u ja nicht wirklich in *Visited* eingefügt. Wir führen den Beweis nun indirekt und nehmen an, dass

$$dist(u) > sp(u)$$

gilt. Dann gibt es einen kürzesten Pfad

$$p = [x_0 = source, x_1, \cdots, x_n = u]$$

von *source* nach u, der insgesamt die Länge $sp(u)$ hat. Es sei $i \in \{0, \cdots, n-1\}$ der Index für den

$$x_i \in Visited \quad \text{aber} \quad x_{i+1} \notin \texttt{Visited},$$

gilt, x_i ist also der erste Knoten aus dem Pfad p, für den x_{i+1} nicht mehr in der Menge *Visited* liegt. Nachdem x_i in die Menge Visited eingefügt wurde, wurde für alle Knoten, die mit x_i über eine Kante verbunden sind, die Funktion *dist*() neu ausgerechnet. Insbesondere wurde auch $dist(x_{i+1})$ neu berechnet und der Knoten x_{i+1} wurde spätestens zu diesem Zeitpunkt in die Menge *Fringe* eingefügt. Außerdem wissen wir, dass $dist(x_{i+1}) = sp(x_{i+1})$ gilt, denn nach Induktions-Voraussetzung gilt $dist(x_i) = sp(x_i)$ und die Kante $\langle x_i, x_{i+1} \rangle$ ist Teil eines kürzesten Pfades von x_i nach x_{i+1}.

Da wir nun angenommen haben, dass $x_{i+1} \notin \mathit{Visited}$ ist, muss x_{i+1} immer noch in der Prioritäts-Warteschlange *Fringe* liegen. Dass heißt dann aber, dass $\mathit{dist}(x_{i+1}) \geq \mathit{dist}(u)$ gilt, denn sonst wäre x_{i+1} vor u aus der Prioritäts-Warteschlange entfernt worden. Wegen $\mathit{sp}(x_{i+1}) = \mathit{dist}(x_{i+1})$ haben wir dann aber den Widerspruch

$$\mathit{sp}(u) \geq \mathit{sp}(x_{i+1}) = \mathit{dist}(x_{i+1}) \geq \mathit{dist}(u) > \mathit{sp}(u). \qquad \square$$

Abbildung 8.4 auf Seite 152 zeigt eine Implementierung des von Dijkstra vorgeschlagenen Algorithmus in *Java*, die wir jetzt im Detail diskutieren:

1. Die Member-Variable `mEdges` repräsentiert die Funktion
   $$\mathit{edges} : \mathbb{V} \to 2^{\mathbb{E}}.$$

2. Die Member-Variable `mLength` repräsentiert die Funktion, die die Länge der einzelnen Kanten berechnet, es gilt
   $$\texttt{mLength}\big(\langle u, v\rangle\big) = \|\langle u, v\rangle\|.$$

3. In Zeile 10 und 11 initialsieren wir die Funktion *dist* und implementieren die Zuweisung
   $$\mathit{dist}(\mathit{source}) \;:=\; 0.$$

4. Die `while`-Schleife in Zeile 13 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.

5. Die Relabeling-Phase wird durch die `for`-Schleife in Zeile 18 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 die Zuweisung
   $$\mathit{dist}(v) \;:=\; \mathit{dist}(u) + \|\langle u, v\rangle\|.$$
   Gleichzeitig fügen wir den Knoten v in die Menge *fringe* ein.

   (b) Andernfalls ist $\mathit{dist}(v)$ 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 Prioritäts des Knotens v in der Warteschlange erhöhen.

Es kann gezeigt werden, dass die Komplexität von Dijkstra's Algorithmus durch den Ausdruck

$$\mathcal{O}\big((\#\mathbb{E} + \#\mathbb{V}) * \ln(\#\mathbb{V})\big)$$

abgeschätzt werden kann. Hierbei bezeichnet $\#\mathbb{E}$ die Zahl der Kanten des Graphen und $\#\mathbb{V}$ die Zahl der Knoten. Ist die Zahl der Kanten, die von den Knoten ausgehen können, 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 der Bestimmung des kürzesten Weges durch den Ausdruck $\mathcal{O}\big((\#\mathbb{V}) * \ln(\#\mathbb{V})\big)$ gegeben.

Aufgabe: Begründen Sie die obige Aussage.

```
1  public class Dijkstra
2  {
3      Map<Node, Set<Edge>> mEdges;
4      Map<Edge, Integer>   mLenght;
5
6      ⋮
7
8      public Map<Node, Integer> shortestPath(Node source)
9      {
10         Map<Node, Integer> dist = new TreeMap<Node, Integer>();
11         dist.put(source, 0);
12         HeapTree<Integer, Node> fringe = new HeapTree<Integer, Node>();
13         while (!fringe.isEmpty()) {
14             Pair<Integer, Node> p     = fringe.top();
15             Integer             distU = p.getFirst();
16             Node                u     = p.getSecond();
17             fringe.remove();
18             for (Edge edge: mEdges.get(u)) {
19                 Node v = edge.getSecond();
20                 if (dist.get(v) == null) {
21                     Integer d = distU + mLenght.get(edge);
22                     dist.put(v, d);
23                     fringe.insert(d, v);
24                 } else {
25                     Integer oldDist = dist.get(v);
26                     Integer newDist = dist.get(u) + mLenght.get(edge);
27                     if (newDist < oldDist) {
28                         dist.put(v, newDist);
29                         fringe.change(newDist, v);
30                     }
31                 }
32             }
33         }
34         return dist;
35     }
36 }
```

Abbildung 8.4: Dijkstra's Algorithmus zur Lösung des kürzeste-Wege-Problems.

Literaturverzeichnis

[1] *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

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

[3] *Egon Börger* and *Robert Stärk*: *Abstract State Machines*, Springer, 2003.

[4] *Frank M. Carrano*: *Data Abstraction and Problem Solving with* `C++`, Benjamin/Cummings Publishing Company, 1995.

[5] *Frank M. Carrano* and *Janet J. Prichard*: *Data Abstraction and Problem Solving with Java*, Prentice Hall, 2003.

[6] *Thomas H. Cormen* and *Charles E. Leiserson*: *Introduction to Algorithms*, MIT Press, 2001.

[7] *Yuri Gurevich*: *Evolving Algebras. Bull. EATCS*, **43**:264–284, 1991.

[8] *Robert Sedgewick*: *Algorithmen in* `C`, Addison-Wesley, 1992.

[9] *Robert Sedgewick*: *Algorithmen in Java*, Pearson, 2002.