Algorithms

— Lecture Notes for the Summer Term 2014 —

Baden-Württemberg Cooperative State University

Prof. Dr. Karl Stroetmann

May 27, 2014

These lecture notes, their LaTeXsources, and the programs discussed in these lecture notes are all available at

`https://github.com/karlstroetmann/Algorithms`.

The lecture notes are subject to continuous change. Provided the program `git` is installed on your computer, the repository containing the lecture notes can be cloned using the command

```
git clone https://github.com/karlstroetmann/Algorithms.git
```

Once you have cloned the repository, the command

```
git pull
```

can be used to load the current version of these lecture notes from `github`.

Contents

Chapter 1

Introduction

1.1 Motivation

The previous lecture in the winter term has shown us how interesting problems can be solved with the help of sets and relations. However, we did not discuss how sets and relations can be represented and how the operations on sets can be implemented in an efficient way. This course will answer these questions: We will see data structure that can be used to represent sets in an efficient way. Furthermore, we will discuss a number of other data structures and algorithms that should be in the toolbox of every computer scientist.

While the class in the last term has introduced the students to the theoretical foundations of computer science, this class is more application oriented. Indeed, it may be one of the most important classes for your future career: Stanford University regularly asks their former students to rank those classes that were the most useful for their professional career. Together with programming and databases, the class on algorithms consistently ranks highest. The practical importance of the topic of this class can also be seen by the availability of book titles like "Algorithms for Interviews" [AP10] or the Google job interview questions.

1.2 Overview

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

1. Undecidability of the halting problem.

   At the beginning of the lecture we discuss the limits of computability: We will show that there is no SetlX function `doesTerminate` such that for a given function f of one argument and a string s the expression

   $$\texttt{doesTerminate}(f, s)$$

   yields `true` if the evaluation of $f(s)$ terminates and yields `false` otherwise.

2. Complexity of algorithms.

   In general, in order to solve a given problem it is not enough to develop an algorithm that implements a function f that computes values $f(x)$ for some argument x. If the size of the argument x is big, then it is also important that the computation of $f(x)$ does not take too much time. Therefore, we want to have *efficient* algorithms. In order to be able to discuss the efficiency of algorithms we have to introduce two mathematical notions.

   (a) *Recurrence relations* are discrete analogues of differential equations. Recurrence relations occur naturally when analyzing the runtime of algorithms.
   (b) *Big O notation* offers a convenient way to discuss the growth rate of functions. This notation is useful to abstract from unimportant details when discussing the runtime of algorithms.

3. Abstract data types.

   Abstract data types are a means to describe the behavior of an algorithm in a concise way.

4. Sorting algorithms.

   Sorting algorithm are the algorithms that are most frequently used in practice. As these algorithms are, furthermore, quite easy to understand they serve best for an introduction to the design of algorithms. We discuss the following sorting algorithms:

   (a) insertion sort,
   (b) selection sort,
   (c) merge sort, and
   (d) quicksort.

5. Hoare logic.

   The most important property of an algorithm is its correctness. The *Hoare calculus* is a method to investigate the correctness of an algorithm.

6. Associative arrays.

   Associative arrays are a means to represent a function. We discuss various data structures that can be used to implement associative arrays efficiently.

7. Priority queues.

   Many graph theoretical algorithms use priority queues as a basic building block. Furthermore, priority queues have important applications in the theory of operating systems and in simulation.

8. Graph theory.

   There are many applications of graphs in computer science. The topic of graph theory is very rich and can easily fill a class of its own. Therefore, we can only cover a small subset of this topic. In particular, we will discuss Dijkstra's algorithm for computing the shortest path.

9. Monte Carlo Method

   Many important problems either do not have an exact solution at all or the computation of an exact solution would be prohibitively expensive. In these cases it is often possible to use simulation in order to get an approximate solution. As a concrete example we will show how certain probabilities in Texas hold 'em poker can be determined approximately with the help of the Monte Carlo method.

The primary goal of these lectures on algorithms is not to teach as many algorithms as possible. Instead, I want to teach how to think algorithmically: At the end of these lectures, you should be able to develop your own algorithms on yourself. This is a process that requires a lot of creativity on your side, as there are no cookbook recipes that can be followed. However, once you are acquainted with a fair number of algorithms, you should be able to discover algorithms on your own.

1.3 Algorithms and Programs

This is a lecture on algorithms, not on programming. It is important that you do not mix up programs and algorithms. An algorithm is an *abstract* concept to solve a given problem. In contrast, a program is a *concrete* implementation of an algorithm. In order to implement an algorithm by a program we have to cover every detail, be it trivial or not. On the other hand, to specify an algorithm it is often sufficient to describe the interesting aspects. It is quite possible for an algorithm to leave a number of questions open.

In the literature, algorithms are mostly presented as pseudo code. Syntactically, pseudo code looks similar to a program, but in contrast to a program, pseudo code can also contain parts that are only described in natural language. However, it is important to realize that a piece of pseudo code is not an algorithm but is only a *representation* of an algorithm. However, the advantage of pseudo code is that we are not confined by the randomness of the syntax of a programming language.

Conceptually, the difference between an algorithm and a program is similar to the difference between an *idea* and a *text* that describes the idea. If you have an idea, you can write it down to make it concrete. As you can write down the idea in English or French or any other language, the textual descriptions of the idea might be quite different. This is the same with an algorithm: We can code it in *Java* or COBOL or any other language. The programs will be very different but the algorithm will be the same.

Having discussed the difference between algorithms and programs, let us now decide how to present algorithms in this lecture.

1. We can describe algorithms using natural language. While natural language certainly is expressive enough, it also suffers from ambiguities. Furthermore, natural language descriptions of complex algorithms tend to be very hard to understand.

2. Instead, we can describe an algorithm by implementing it. There is certainly no ambiguity in a program, but on the other hand this approach would require us to implement every aspect of an algorithm and our descriptions of algorithms would therefore get longer than we want.

3. Finally, we can try to describe algorithms in the language of mathematics. This language is concise, unambiguous, and easy to understand, once you are accustomed to it. This is therefore our method of choice.

   However, after having presented an algorithm in the language of mathematics, it is often very simple to implement this algorithm in the programming language SETLX. The reason is that SETLX is based on set theory, which is the language of mathematics. We will see that SETLX enables us to present and implement algorithms on a very high abstraction level.

1.4 Desirable Properties of Algorithms

Before we start with our discussion of algorithms we should think about our goals when designing algorithms.

1. Algorithms have to be *correct*.

2. Algorithms should be as *efficient* as possible.

3. Algorithms should be *simple*.

The first goal in this list is so self-evident that it is often overlooked. The importance of the last goal might not be as obvious as the other goals. However, the reasons for the last goal are economical: If it takes too long to code an algorithm, the cost of the implementation might well be unaffordable. Furthermore, even if the budget is unlimited there is another reasons to strife for simple algorithms: If the conceptual complexity of an algorithm is too high, it may become impossible to check the correctness of the implementation. Therefore, the third goal is strongly related to the first goal.

1.5 Literature

These lecture notes are available online at

https://github.com/karlstroetmann/Algorithms/blob/master/English-Script/algorithms.pdf.

They are intended to be the main source for my lecture. Additionally, I want to mention those books that have inspired me most.

1. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *Data Structures and Algorithms*, Addison-Wesley, 1987, [AHU87].

   This book is quite dated now but it is one of the classics on algorithms. It discusses algorithms at an advanced level.

2. *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, and *Clifford Stein*: *Introduction to Algorithms*, third edition, MIT Press, 2009, [CLRS09]

   Due to the level of detail and the number of algorithms given, this book is well suited as a lookup library.

3. *Robert Sedgewick*: *Algorithms in Java*, fourth edition, Pearson, 2011, [SW11a].

   This book has a nice booksite containing a wealth of additional material. This book seems to be the best choice for the working practitioner. Furthermore, Professor Sedgewick teaches a course on algorithms on coursera that is based on this book.

4. *Einführung in die Informatik*, written by *Heinz-Peter Gumm* and `Manfred Sommer` [GS08].

   This German book is a very readable introduction to computer science and it has a comprehensive chapter on algorithms. Furthermore, this book is available electronically in our library.

5. Furthermore, there is a set of video lectures from Professor Roughgarden at coursera.

6. *Algorithms*, written by *Sanjoy Dasgupta, Christos H. Papadimitriou, and Unmesh V. Vazirani* [DPV08] is a short text book on Algorithms that is available online.

7. *Data Structures and Algorithms* written by *Kurt Mehlhorn and Peter Sanders* [MS08] is another good text book on algorithms that is available online.

1.6 A Final Remark

There is one final remark I would like to make at this point: Frequently, I get questions from students concerning the exams. While I will answer most of them, I should warn you that, 50% of the time, my answers will be lies. The other 50%, my answer will be either dirty lies or accidentally true.

1.7 A Request

Computer science is a very active field of research. Therefore, these lecture notes are constantly evolving and hence might contain typos or bugs. If you find a bug, please take the time and send me an email. My email address is

karl.stroetmann@dhbw-mannheim.de.

If you are familiar with `github`, you might even consider sending me a pull request.

Chapter 2

Limits of Computability

Every discipline of the sciences has its limits: Students of the medical sciences soon realize that it is difficult to raise the dead, while adepts of theology have to admit that there is no way they can walk on water. But believe it or not, even computer science has its limits! We will discuss these limits next. First, we show that we cannot decide whether a computer program will eventually terminate or whether it will run forever. Second, we prove that it is impossible to check whether two programs are equivalent.

2.1 The Halting Problem

In this subsection we prove that it is not possible for a computer program to decide whether another computer program does terminate. This problem is known as the *halting problem*. Before we give a formal proof that the halting problem is undecidable, let us discuss one example that shows why it is indeed difficult to decide whether a program does always terminate. Consider the program shown in Figure 2.1 on page 9. This program contains an obvious infinite loop in line 6. However, this loop is only entered if the expression

$$\texttt{legendre}(n)$$

in line 5 evaluates to `false`. Given a natural number `n`, the expression $\texttt{legendre}(n)$ tests whether there is a prime between $\texttt{n}^2$ and $(\texttt{n}+1)^2$. If, however, the intervall

$$[n^2, (n+1)^2] := \{k \in \mathbb{N} \mid n^2 \leq k \wedge k \leq (n+1)^2\}$$

does not contain a prime number, then $\texttt{legendre}(n)$ evaluates to `false` for this value of n. The function `legendre` is defined in line 2. Given a natural number n, it returns `true` if and only if the formula

$$\exists k \in \mathbb{N} : n^2 \leq k \wedge k \leq (n+1)^2 \wedge \mathit{isPrime}(k)$$

holds true. The French mathematican Adrien-Marie Legendre (1752 – 1833) conjectured that for any natural number $n \in \mathbb{N}$ there is prime number p such that

$$n^2 \leq p \wedge p \leq (n+1)^2$$

holds. Although there are a number of reasons in support of Legendre's conjecture, to this day nobody has been able to prove it. The question, whether the invocation of the function f will terminate for every user input is, therefore, unknown as it depends on the truth of Legendre's conjecture: If we had some procedure that could check whether the function `main` does terminate for every number n that is input by the user, then this procedure would be able to decide whether Legendre's theorem

is true. Therefore, it should come as no surprise that such a procedure does not exist.

```
1  main := procedure() {
2      legendre := n |-> exists (k in [n**2..(n+1)**2] | isPrime(k));
3
4      n := read("input a natural number: ");
5      if (!legendre(n)) {
6          while (true) {
7              print("looping");
8          }
9      }
10     return true;
11 };
```

Figure 2.1: A program running into trouble if Legendre's was wrong.

Let us proceed to prove formally that the halting problem is not solvable. To this end, we need the following definition.

Definition 1 (Test Function) A string t is a *test function with name* n iff t has the form

n `:= procedure(x) { ··· };`

and, furthermore, the string t can be parsed as a SetlX statement, that is the evaluation of the expression

```
parseStatements(t);
```

does not yield an error. The set of all test functions is denoted as TF. If $t \in TF$ and t has the name n, then this is written as

$\texttt{name}(t) = n.$ □

Examples:

1. s_1 = "`simple := procedure(x) { return 0; };`"
   s_1 is a test function with the name `simple`.
2. s_2 = "`loop := procedure(x) { while (true) { x := x + 1; } };`"
   s_2 is a test function with the name `loop`.
3. s_3 = "`hugo := procedure(x) { return ++x; };`"
   s_3 is not a test function. The reason is that SetlX does not support the operator "`++`". Therefore,

   ```
   parseStatements(s3)
   ```

   yields an error message complaining about the two "`+`" characters.

In order to be able to formalize the halting problem succinctly, we introduce three additional notations.

Notation 2 ($\rightsquigarrow$, $\downarrow$, $\uparrow$) If n is the name of a SetlX function that takes k arguments $a_1, \cdots, a_k$, then we write

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

iff the evaluation of the expression $n(a_1, \cdots, a_k)$ yields the result r. If we are not concerned with the result r but only want to state that the evaluation terminates, then we will write

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

and read this notation as "*evaluating* $n(a_1, \cdots, a_k)$ *terminates*". If the evaluation of the expression $n(a_1, \cdots, a_k)$ does not terminate, this is written as

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

This notation is read as "*evaluation of* $n(a_1, \cdots, a_k)$ *diverges*". □

Examples: Using the test functions defined earlier, we have:

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

The *halting problem* for SetlX functions is the question whether there is a `SetlX` function

```
stops := procedure(t, a) { ··· };
```

that takes as input a test function t and a string a and that satisfies the following specification:

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

   If the first argument of `stops` is not a test function, then `stops(`t`,` a`)` returns the number 2.

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

   If the first argument of `stops` is a test function and, furthermore, the evaluation of $n(a)$ terminates, then `stops(`t`,` a`)` returns the number 1.

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

   If the first argument of `stops` is a test function but the evaluation of $n(a)$ diverges, then `stops(`t`,` a`)` returns the number 0.

If there was a SetlX function `stops` that did satisfy the specification given above, then the halting problem for SetlX would be decidable.

Theorem 3 (Alan Turing, 1936) The halting problem is undecidable.

Proof: In order to prove the undecidabilty of the halting problem we have to show that there can be no function `stops` satisfying the specification given above. This calls for an indirect proof also known as *proof by contradiction*. We will therefore assume that a function `stops` solving the halting problem does exist and we will then show that this assumption leads to a contradiction. This contradiction will leave us with the conclusion that there can be no function `stops` that satisfies the specification given above and that, therefore, the halting problem is not solvable.

In order to proceed, let us assume that a SetlX function `stops` satisfying the specification given above exists and let us define the string *turing* as shown in Figure 2.2 below.

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

Figure 2.2: Definition of the string *turing*.

Given this definition it is easy to check that *turing* is, indeed, a test function with the name "`alan`', that is we have

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

Therefore, we can use the string *turing* as the first argument of the function `stops`. Let us think about the following expression:

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

Since we have already noted that *turing* is test function, according to the specification of the function `stops` there are only two cases left:

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

Let us consider these cases in turn.

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

   According to the specification of `stops` we should then have

   $$\texttt{alan}(\mathit{turing}) \uparrow.$$

   Let us check whether this is true. In order to do this, we have to check what happens when the expression

   ```
   alan(turing)
   ```

   is evaluated:

   (a) Since we have assumed for this case that the expression $\texttt{stops}(\mathit{turing}, \mathit{turing})$ yields 0, in line 2, the variable `result` is assigned the value 0.

   (b) Line 3 now tests whether `result` is 1. Of course, this test fails. Therefore, the block of the `if`-statement is not executed.

   (c) Finally, in line 8 the value of the variable `result` is returned.

   All in all we see that the call of the function `alan` does terminate with the argument *turing*. However, this is the opposite of what the function `stops` has claimed.

   Therefore, this case has lead us to a contradiction.

2. $\texttt{stops}(\mathit{turing}, \mathit{turing}) \rightsquigarrow 1$.

   According to the specification of `stops` we should then have

   $$\texttt{alan}(\mathit{turing}) \downarrow,$$

   i.e. the evaluation of $\texttt{alan}(\mathit{turing}) \downarrow$ should terminate.

Again, let us check in detail whether this is true.

(a) Since we have assumed for this case that the expression `stops`(*turing*, *turing*) yields 1, in line 2, the variable `result` is assigned the value 1.

(b) Line 3 now tests whether `result` is 1. Of course, this time the test succeeds. Therefore, the block of the `if`-statement is executed.

(c) However, this block contains an obvious infinite loop. Therefore, the evaluation of `alan`(*turing*) diverges. But this contradicts the specification of `stops`!

Therefore, the second case also leads to a contradiction.

As we have obtained contradictions in both cases, the assumption that there is a function `stops` that solves the halting problem is refuted. □

Remark: The proof of the fact that the halting problem is undecidable was given 1936 by Alan Turing (1912 – 1954) [Tur36]. Of course, Turing did not solve the problem for SetlX but rather for the so called *Turing machines*. A *Turing maschine* can be interpreted as a formal description of an algorithm. Therefore, Turing has shown, that there is no algorithm that is able to decide whether some given algorithm will always terminate.

Remark: Having read this far you might wonder whether there might be another programming language that is more powerful so that programming in this more powerful language it would be possible to solve the halting problem. However, if you check the proof given for SetlX you will easily see that this proof can be adapted to any other programming language that is as least as powerful as SetlX.

Of course, if a programming language is very restricted, then it might be possible to check the halting problem for this weak programming language. But for any programming language that supports at least `while`-loops, `if`-statements, and the definition of procedures the argument given above shows that the halting problem is not solvable.

Exercise 1: Show that if the halting problem would be solvable, then it would be possible to write a program that checks whether there are infinitely many *twin primes*. A *twin prime* is pair of natural numbers $\langle p, p+2\rangle$ such that both p and $p+2$ are prime numbers. The *twin prime conjecture* is one of the oldest unsolved mathematical problems.

Exercise 2: A set X is called *countable* iff there is a function

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

such that for all $x \in X$ there is a $n \in \mathbb{N}$ such that x is the image of n under f:

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

Prove that the set $2^{\mathbb{N}}$, which is the set of all subsets of $\mathbb{N}$ is not countable.

Hint: Your proof should be similar to the proof that the halting problem is undecidable. Proceed as follows: Assume that there is a function f enumerating the subsets of $\mathbb{N}$, that is assume that

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

holds. Next, and this is the crucial step, define a set `Cantor` as follows:

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

Now try to derive a contradiction.

2.2 Undecidability of the Equivalence Problem

Unfortunately, the halting problem is not the only undecidable problem in computer science. Another important problem that is undecidable is the question whether two given functions always compute the same result. To state this more formally, we need the following definition.

Definition 4 ($\simeq$) Assume n_1 and n_2 are the names of two SETLX functions that take arguments a_1, $\cdots$, a_k. Let us define

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

if and only if either of the following cases is true:

1. $n_1(a_1, \cdots, a_k) \uparrow \quad \wedge \quad n_2(a_1, \cdots, a_k) \uparrow$,
   that is both function calls diverge.
2. $\exists r : \Big(n_1(a_1, \cdots, a_k) \rightsquigarrow r \quad \wedge \quad n_2(a_1, \cdots, a_k) \rightsquigarrow r\Big)$
   that is both function calls terminate and compute the same result.

If $n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$ holds, then the expressions $n_1(a_1, \cdots, a_k)$ and $n_2(a_1, \cdots, a_k)$ are called *partially equivalent*. □

We are now ready to state the *equivalence problem*. A SETLX function `equal` solves the *equivalence problem* if it is defined as

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

and, furthermore, satisfies the following specification:

1. $p_1 \notin TF \;\vee\; p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \rightsquigarrow 2.$
2. If
   (a) $p_1 \in TF \;\wedge\; \texttt{name}(p_1) = n_1$,
   (b) $p_2 \in TF \;\wedge\; \texttt{name}(p_2) = n_2$ and
   (c) $n_1(a) \simeq n_2(a)$
   holds, then we must have:
   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 1.$$
3. Otherwise we must have
   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 0.$$

Theorem 5 (Henry Gordon Rice, 1953)
The equivalence problem is undecidable.

Proof: The proof is by contradiction. Therefore, assume that there is a function `equal` such that `equal` solves the equivalence problem. Assuming `equal` exists, we will then proceed to define a function `stops` that does solve the halting problem. Figure 2.3 shows how we construct the function `stops` that makes use of the function `equal`.

Notice that in line 2 the function `equal` is called with a string that is test function with name `loop`. This test function has the following form:

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

```
stops := procedure(p, a) {
    f := "loop := procedure(x) { while (true) {} };";
    e := equal(f, p, a);
    if (e == 2) {
        return 2;
    } else {
        return 1 - e;
    }
}
```

Figure 2.3: An implementation of the function `stops`.

Obviously, the function `loop` does never terminate. Therefore, if the argument p of `stops` is a test function with name n, the function `equal` will return 1 if $n(a)$ diverges, and will return 0 otherwise. But this implementation of `stops` would then solve the halting problem as for a given test function p with name n and argument a the function `stops` would return 1 if and only the evaluation of $n(a)$ terminates. As we have already proven that the halting problem is undecidable, there can be no function `equal` that solves the equivalence problem either. □

2.3 Concluding Remarks

Although, in general, we cannot decide whether a program terminates for a given input, this does not mean that we should not attempt to do so. After all, we only have proven that there is no procedure that can always check whether a given program will terminate. There might well exist a procedure for termination checking that works most of the time. Indeed, there are a number of systems that try to check whether a program will always terminate. For example, for *Prolog* programs, the article "*Automated Modular Termination Proofs for Real Prolog Programs*" [MGS96] describes a successful approach. The recent years have seen a lot of progress in this area. The article "Proving Program Termination" [CPR11] reviews these developments. However, as the recently developed systems rely on both *automatic theorem proving* and *Ramsey theory* they are quite out of the scope of this lecture.

2.4 Further Reading

The book "*Introduction to the Theory of Computation*" by Michael Sipser [Sip96] discusses the undecidability of the halting problem in section 4.2. It also covers many related undecidable problems. The exposition in the book of Sipser is based on Turing machines.

Another good book discussing undecidability is the book "*Introduction to Automata Theory, Languages, and Computation*" written by John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman [HMU06]. This book is the third edition of a classic text. In this book, the topic of undecidability is discussed in chapter 9.

Chapter 3

Big $\mathcal{O}$ Notation

This chapter introduces both the *big O notation* and the *tilde notation* advocated by Sedgewick [SW11a]. These two notions are needed to analyse the running time of algorithms. In order to have some algorithms to talk about, we will also discuss how to implement the computation of powers efficiently, i. e. we discuss how to evaluate the expression a^b for given $a, b \in \mathbb{N}_0$ in a way that is significantly faster than the naive approach.

3.1 Motivation

Often it is necessary to have a precise understanding of the complexitity of an algorithm. In order to obtain this understanding we could proceed as follows:

1. We implement the algorithm in a given programming language.
2. We count how many additions, multiplications, assignments, etc. are needed for an input of a given length.
3. We read the processor handbook to look up the amount of time that is needed for the different operations.
4. Using the information discovered in the previous two steps we can then predict the running time of our algorithm for given input.

As you might have already guessed by now, this approach is problematic for a number of reasons.

1. It is very complicated.
2. The execution time of the basic operations is highly dependend on the memory hierarchy of the computer system: For many modern computer architectures, adding two numbers that happen to be in a register is more than ten times faster than adding two numbers that reside in main memory.

   However, unless we peek into the machine code generated by our compiler, it is very difficult to predict whether a variable will be stored in memory or in a register. Even if a variable is stored in main memory, we still might get lucky if the variable is also stored in a cache.
3. If we would later code the algorithm in a different programming language or if we would port the program to a computer with a different processor we would have to redo most of the computation.

The final reason shows that the approach sketched above is not well suited to measure the complexity of an algorithm: After all, the notion of an algorithm is more abstract than the notion of a program and we really need a notion measuring the complexity of an algorithm that is more abstract than the notion of the running time of a program. This notion of complexity should satisfy the following specification:

- The notion of complexity should abstract from constant factors. After all, according to *Moore's law*, computers hitting the market 18 month from now will be about twice as powerful as todays computers.
- The notion should abstract from *insignificant terms*.

  Assume you have written a program that multiplies two $n \times n$ matrices. Assume, furthermore, that you have computed the running time $T(n)$ of this program as a function of the size n of the matrix as

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

  When compared with the total running time, the portion of running time that is due to the term $2 \cdot n^2 + 7$ will decrease with increasing value of n. To see this, consider the following table:

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

  This table clearly shows that, for large values of n, the term $2 \cdot n^2 + 7$ can be neglected.
- The notion of complexity should describe how the running time increases when the size of the input increases: For small inputs, the running time is not very important but the question is how the running time grows when input size is increased. Therefore the notion of complexity should capture the *growth* of the running time.

Let us denote the set of all positive real numbers as $\mathbb{R}_+$, i. e. let us define

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

Furthermore, the set of all functions defined on $\mathbb{N}_0$ yielding a positive real number is defined as:

$$\mathbb{R}_+^{\mathbb{N}_0} = \{f \mid f \text{ is a function of the form } f : \mathbb{N}_0 \to \mathbb{R}_+\}.$$

Definition 6 ($\mathcal{O}(g)$) Assume $g \in \mathbb{R}_+^{\mathbb{N}_0}$ is given. Let us define the set of all functions that *grow at most as fast* as the function g as follows:

$$\mathcal{O}(g) := \Big\{ f \in \mathbb{R}_+^{\mathbb{N}_0} \mid \exists k \in \mathbb{N}_0 \colon \big(\exists c \in \mathbb{R}_+ \colon \forall n \in \mathbb{N}_0 \colon n \geq k \to f(n) \leq c \cdot g(n)\big) \Big\}.$$

◇

The definition of $\mathcal{O}(g)$ contains three nested quantifiers and may be difficult to understand when first encountered. Therefore, let us analyse this definition carefully. We take some function g and try to understand what $f \in \mathcal{O}(g)$ means.

1. Informally, the idea is that, for increasing values of the argument n, the function f does not grow faster than the function g.

2. The fact that $f \in \mathcal{O}(g)$ holds does not impose any restriction on small values of n. After all, the condition

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

   if only required for those values of n that are bigger than k and the value k can be any suitable natural number.

   This property shows that the big $\mathcal{O}$ notation captures the growth rate of functions.

3. Furthermore, $f(n)$ can be bigger than $g(n)$ even for arbitrary values of n but it can only be bigger by a constant factor: There must be some fixed constant c such that

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

   holds for all values of n that are sufficiently big. This implies that if $f \in \mathcal{O}(g)$ holds then, for example, the function $2 \cdot f$ will also be in $\mathcal{O}(g)$.

   This last property shows that the big $\mathcal{O}$ notation abstracts from constant factors.

I have borrowed Figure 3.1 from the wikipedia article on asymptotic notation. It shows two functions $f(x)$ and $c \cdot g(x)$ such that $f \in \mathcal{O}(g)$. Note that the function $f(x)$, which is drawn in red, is less or equal than $c \cdot g(x)$ for all values of x such that $x \geq k$. In the figure, we have $k = 5$, since the condition $f(x) \leq g(x)$ is satisfied for $x \geq 5$. For values of x that are less than $k = 5$, sometimes $f(x)$ is bigger than $c \cdot g(x)$ but that does not matter. In Figure 3.1 the functions $f(x)$ and $g(x)$ are drawn as if they were functions defined for all positive real numbers. However, this is only done to support the visualization of these functions. In reality, the functions f and g are only defined for natural numbers.

Next, we discuss some examples in order to further clarify the notion $f \in \mathcal{O}(g)$.

Example: We claim that the following holds:

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

Proof: We have to provide a constant c and another constant k such that for all $n \in \mathbb{N}_0$ satisfying $n \geq k$ the inequality

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

holds. Let us define $k := 1$ and $c := 12$. Then we may assume that

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

holds and we have to show that this implies

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

If we take the third power of both sides of the inequality (3.1) then we see that

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

holds. Let us multiply both sides of this inequality with 7. We get:

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

Furthermore, let us multiply the inequality (3.1) with the term $2 \cdot n^2$. This yields

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

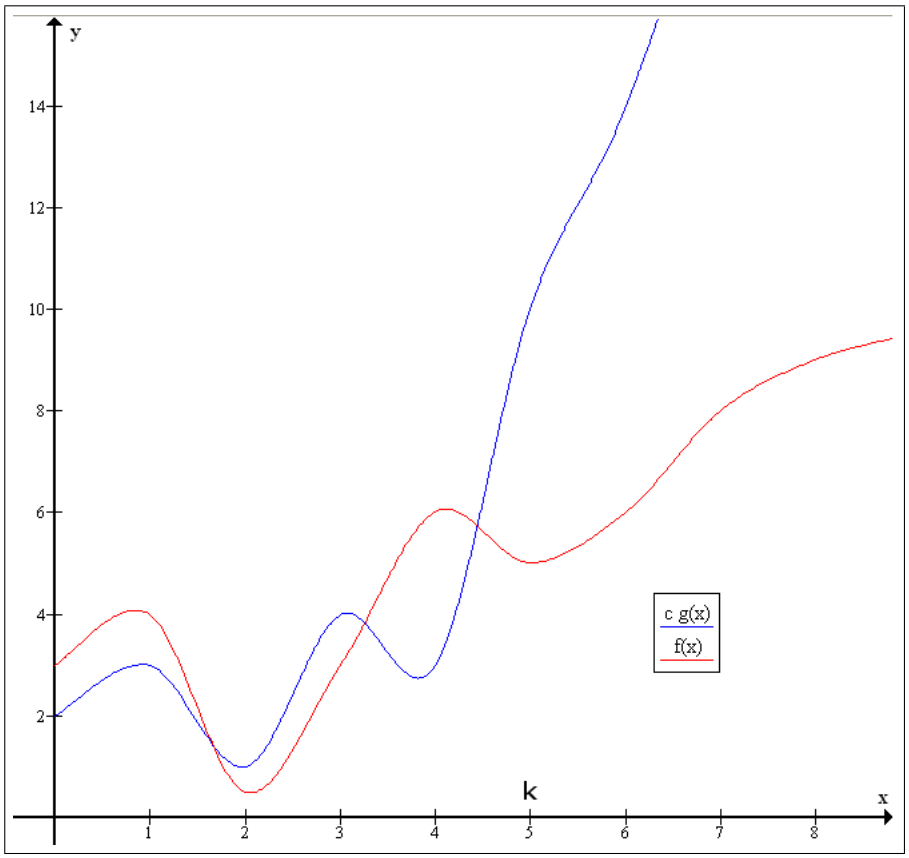

Figure 3.1: Example for $f \in \mathcal{O}(g)$.

Finally, we obviously have

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

Adding up the inequalities (3.4), (3.5), and (3.6) gives

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

and therefore the proof is complete. □

Example: We have $n \in \mathcal{O}(2^n)$.

Proof: We have to provide a constant c and a constant k such that

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

holds for all $n \geq k$. Let us define $k := 0$ and $c := 1$. We will then have to show that

$$n \leq 2^n \quad \text{holds for all } n \in \mathbb{N}_0.$$

We prove this claim by induction on n.

1. **Base case**: $n = 0$
   Obviously, $n = 0 \leq 1 = 2^0 = 2^n$ holds. Therefore, $n \leq 2^n$ holds for $n = 0$.
2. **I.S.**: $n \mapsto n + 1$
   By the induction hypothesis we have

$$n \leq 2^n. \tag{3.7}$$

Furthermore, we have

$$1 \leq 2^n. \tag{3.8}$$

Adding the inequalities (3.7) and (3.8) yields

$$n + 1 \leq 2^n + 2^n = 2^{n+1}.$$ □

Remark: To be complete, we should also have proven the inequality $1 \leq 2^n$ by induction. As this fact is so obvious and the proof is straightforward, it is left to the student.

Exercise 3: Prove that

$$n^2 \in \mathcal{O}(2^n).$$ ◇

It would be very tedious if we would have to use induction every time we need to prove that $f \in \mathcal{O}(g)$ holds for some functions f and g. Therefore, we show a number of properties of the big $\mathcal{O}$ notation next. These properties will later enable us to prove a claim of the form $f \in \mathcal{O}(g)$ much quicker than by induction.

Proposition 7 (Reflexivity) For all functions $f\colon \mathbb{N}_0 \to \mathbb{R}_+$ we have that

$$f \in \mathcal{O}(f) \quad \text{holds.}$$

Proof: Let us define $k := 1$ and $c := 1$. Then our claim follows immediately from the inequality

$$\forall n \in \mathbb{N}_0 \colon f(n) \leq f(n).$$ □

Proposition 8 (Multiplication with Constants) Assume that we have functions $f, g\colon \mathbb{N}_0 \to \mathbb{R}_+$ and a number $d \in \mathbb{R}_+$. Then we have that

$$g \in \mathcal{O}(f) \to d \cdot g \in \mathcal{O}(f) \quad \text{holds.}$$

Proof: The premiss $g \in \mathcal{O}(f)$ implies that there are constants $c' \in \mathbb{R}_+$ and $k' \in \mathbb{N}_0$ such that

$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k' \to g(n) \leq c' \cdot f(n)\big)$$

holds. If we multiply this inequality with d, we get

$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k' \to d \cdot g(n) \leq d \cdot c' \cdot f(n)\big)$$

Let us therefore define $k := k'$ and $c := d \cdot c'$. Then we have

$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k \to d \cdot g(n) \leq c \cdot f(n)\big)$$

and by definition this implies $d \cdot g \in \mathcal{O}(f)$. □

Remark: The previous proposition shows that the big $\mathcal{O}$ notation does indeed abstract from constant factors.

Proposition 9 (Addition) Assume that $f, g, h\colon \mathbb{N}_0 \to \mathbb{R}_+$. Then we have

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

Proof: The preconditions $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$ imply that there are constants $k_1, k_2 \in \mathbb{N}_0$ and $c_1, c_2 \in \mathbb{R}$ such that both

$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot h(n)\big) \quad \text{and}$$
$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

holds. Let us define $k := \max(k_1, k_2)$ and $c := c_1 + c_2$. For all $n \in \mathbb{N}$ such that $n \geq k$ it then follows that both

$$f(n) \leq c_1 \cdot h(n) \quad \text{and} \quad g(n) \leq c_2 \cdot h(n)$$

holds. Adding these inequalities we conclude that

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

holds for all $n \geq k$. □

Exercise 4: Assume that $f_1, f_2, h_1, h_2 \colon \mathbb{N}_0 \rightarrow \mathbb{R}_+$. Prove that

$$f_1 \in \mathcal{O}(h_1) \wedge f_2 \in \mathcal{O}(h_2) \;\rightarrow\; f_1 \cdot f_2 \in \mathcal{O}(h_1 \cdot h_2) \quad \text{holds.}$$ ◇

Exercise 5: Assume that $f_1, f_2, h_1, h_2 \colon \mathbb{N}_0 \rightarrow \mathbb{R}_+$. Prove or refute the claim that

$$f_1 \in \mathcal{O}(h_1) \wedge f_2 \in \mathcal{O}(h_2) \;\rightarrow\; f_1 / f_2 \in \mathcal{O}(h_1 / h_2) \quad \text{holds.}$$ ◇

Proposition 10 (Transitivity) Assume $f, g, h \colon \mathbb{N}_0 \rightarrow \mathbb{R}_+$. Then we have

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

Proof: The precondition $f \in \mathcal{O}(g)$ implies that there exists a $k_1 \in \mathbb{N}_0$ and a number $c_1 \in \mathbb{R}$ such that

$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot g(n)\big)$$

holds, while the precondition $g \in \mathcal{O}(h)$ implies the existence of $k_2 \in \mathbb{N}_0$ and $c_2 \in \mathbb{R}$ such that

$$\forall n \in \mathbb{N}_0 \colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

holds. Let us define $k := \max(k_1, k_2)$ and $c := c_1 \cdot c_2$. Then for all $n \in \mathbb{N}$ such that $n \geq k$ we have the following:

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

Let us multiply the second of these inequalities with c_1. Keeping the first inequality this yields

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

However, this immediately implies $f(n) \leq c \cdot h(n)$ and our claim has been proven. □

Proposition 11 (Limit Proposition) Assume that $f, g \colon \mathbb{N}_0 \rightarrow \mathbb{R}_+$. Furthermore, assume that the limit

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

exists. Then we have $f \in \mathcal{O}(g)$.

Proof: Define

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

According to the definition of the notion of a limit there exists a number $k \in \mathbb{N}_0$ such that for all $n \in \mathbb{N}_0$ satisfying $n \geq k$ the inequality

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

holds. Let us multiply this inequality with $g(n)$. This yields

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

The triangle inequality $|a + b| \leq |a| + |b|$ for real numbers tells us that

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

holds. Combining the previous two inequalities yields

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

If we now define

$$c := 1 + \lambda,$$

then we have shown that $f(n) \leq c \cdot g(n)$ holds for all $n \geq k$. □

The following examples show how to put the previous propositions to good use.

Example: Assume $k \in \mathbb{N}_0$. Then we have

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

Proof: We have

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

Therefore, the claim follows from the limit proposition. □

Example: Assume $k \in \mathbb{N}_0$ and $\lambda \in \mathbb{R}$ where $\lambda > 1$. Then we have

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

Proof: We will show that

$$\lim_{n \to \infty} \frac{n^k}{\lambda^n} = 0 \tag{3.9}$$

is true. Then the claim is an immediate consequence of the limit proposition. According to L'Hôpital's rule, the limit can be computed as follows:

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

The derivatives can be computed as follows:

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

We compute the second derivative and get

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

In the same manner, we compute the k-th order derivative and find

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

After k applications of L'Hôpital's rule we arrive at the following chain of equations:

$$\lim_{x \to \infty} \frac{x^k}{\lambda^x} = \lim_{x \to \infty} \frac{\frac{d\, x^k}{dx}}{\frac{d\, \lambda^x}{dx}} = \lim_{x \to \infty} \frac{\frac{d^2\, x^k}{dx^2}}{\frac{d^2\, \lambda^x}{dx^2}} = \cdots$$

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

Therefore the limit exists and the claim follows from the limit proposition. □

Example: We have $\ln(n) \in \mathcal{O}(n)$.

Proof: This claim is again a simple consequence of the limit proposition. We will use L'Hôpital's rule to show that we have

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

We know that

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

Therefore, we have

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

Exercise 6: Prove that $\sqrt{n} \in \mathcal{O}(n)$ holds. ◇

Exercise 7: Assume $\varepsilon \in \mathbb{R}$ and $\varepsilon > 0$. Prove that $n \cdot \ln(n) \in \mathcal{O}\big(n^{1+\varepsilon}\big)$ holds. ◇

Example: We have $2^n \in \mathcal{O}(3^n)$, but $3^n \notin \mathcal{O}(2^n)$. ◇

Proof: First, we have

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

and therefore we have $2^n \in \mathcal{O}\big(3^n\big)$. The proof of $3^n \notin \mathcal{O}(2^n)$ is a proof by contradiction. Assume that $3^n \in \mathcal{O}(2^n)$ holds. Then, there must be numbers c and k such that

$$3^n \leq c \cdot 2^n \quad \text{holds for } n \geq k.$$

Taking the logarithm of both sides of this inequality we find

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

The last inequality would have to hold for all natural numbers n that are bigger than k. Obviously, this is not possible as, no matter what value c takes, there are natural numbers n that are bigger than

$$\frac{\ln(c)}{\ln(3) - \ln(2)}.$$ □

Exercise 8:

(a) Assume that $b > 1$. Prove that $\log_b(n) \in \mathcal{O}(\ln(n))$.

Solution: By the definition of the natural logarithm we have for any positive number n we have that

$$n = \mathrm{e}^{\ln(n)}, \quad \text{where e denotes Euler's number.}$$

Therefore, we can rewrite the expression $\log_b(n)$ as follows:

$$\begin{aligned} \log_b(n) &= \log_b\left(\mathrm{e}^{\ln(n)}\right) \\ &= \ln(n) \cdot \log_b(\mathrm{e}) \\ &= \log_b(\mathrm{e}) \cdot \ln(n) \end{aligned}$$

This shows that the logarithm with respect to some base b and the natural logarithm only differ by a constant factor, namely $\log_b(\mathrm{e})$. Since the big $\mathcal{O}$ notation abstracts from constant factors, we conclude that

$$\log_b(n) \in \mathcal{O}(\ln(n))$$

holds. □

Remark: The previous exercise shows that, with respect to the big $\mathcal{O}$ notation, the base of a logarithm is not important. The reason is that if $b > 1$ and $c > 1$ are both used as bases for the logarithm, we have

$$\log_b(n) = \log_b(\mathrm{e}) \cdot \ln(n) \quad \text{and} \quad \log_c(n) = \log_c(\mathrm{e}) \cdot \ln(n).$$

Solving these equations for $\ln(n)$ yields

$$\frac{\log_b(n)}{\log_b(\mathrm{e})} = \ln(n) \quad \text{and} \quad \frac{\log_c(n)}{\log_c(\mathrm{e})} = \ln(n).$$

This show that

$$\frac{\log_b(n)}{\log_b(\mathrm{e})} = \frac{\log_c(n)}{\log_c(\mathrm{e})}$$

holds. This can be rewritten as

$$\log_b(n) = \frac{\log_b(\mathrm{e})}{\log_c(\mathrm{e})} \cdot \log_c(n),$$

showing that $\log_b(n)$ and $\log_c(n)$ differ only by a constant factor. ◇

(b) Prove $3 \cdot n^2 + 5 \cdot n + \sqrt{n} \in \mathcal{O}(n^2)$.

(c) Prove $7 \cdot n + \left(\log_2(n)\right)^2 \in \mathcal{O}(n)$.

(d) Prove $\sqrt{n} + \log_2(n) \in \mathcal{O}\left(\sqrt{n}\right)$.

(e) Assume that $f, g \in \mathbb{R}_+^{\mathbb{N}}$ and that, furthermore, $f \in \mathcal{O}(g)$. Proof or refute the claim that this implies

$$2^{f(n)} \in \mathcal{O}\left(2^{g(n)}\right).$$

(f) Assume that $f, g \in \mathbb{R}_+^{\mathbb{N}}$ and that, furthermore,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Proof or refute the claim that this implies

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

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

Hint: The last one is difficult! ◇

3.2 A Remark on Notation

Technically, for some function $g : \mathbb{N} \to \mathbb{R}_+$ the expression $\mathcal{O}(g)$ denotes a set. Therefore, for a given function $f : \mathbb{N} \to \mathbb{R}_+$ we can either have

$$f \in \mathcal{O}(g) \quad \text{or} \quad f \notin \mathcal{O}(g),$$

we can never have $f = \mathcal{O}(g)$. Nevertheless, in the literature it has become common to abuse the notation and write

$$f = \mathcal{O}(g) \quad \text{instead of} \quad f \in \mathcal{O}(g).$$

Where convenient, we will also use this notation. However, you have to be aware of the fact that this is quite dangerous. For example, if we have two different functions f_1 and f_2 such that both

$$f_1 \in \mathcal{O}(g) \quad \text{and} \quad f_2 \in \mathcal{O}(g)$$

holds, when we write this as

$$f_1 = \mathcal{O}(g) \quad \text{and} \quad f_2 = \mathcal{O}(g),$$

then we must not conclude that $f_1 = f_2$ as the functions f_1 and f_2 are merely members of the same set $\mathcal{O}(g)$ and are not necessarily equal. For example, $n \in \mathcal{O}(n)$ and $2 \cdot n \in \mathcal{O}(n)$, but $n \neq 2 \cdot n$.

Furthermore, for given functions f, g, and h we write

$$f = g + \mathcal{O}(h)$$

to express the fact that $(f - g) \in \mathcal{O}(h)$. For example, we have

$$n^2 + \tfrac{1}{2} \cdot n \cdot \log_2(n) + 3 \cdot n = n^2 + \mathcal{O}\big(n \cdot \log_2(n)\big).$$

This is true because

$$\tfrac{1}{2} \cdot n \cdot \log_2(n) + 3 \cdot n \in \mathcal{O}\big(n \cdot \log_2(n)\big).$$

The notation $f = g + \mathcal{O}(h)$ is useful because it is more precise than the pure big $\mathcal{O}$ notation. For example, assume we have two algorithms A and B for sorting a list of length n. Assume further that the number $\mathit{count}_A(n)$ of comparisons used by algorithm A to sort a list of length n is given as

$$\mathit{count}_A(n) = n \cdot \log_2(n) + 7 \cdot n,$$

while for algorithm B the corresponding number of comparisons is given as

$$\mathit{count}_B(n) = \tfrac{3}{2} \cdot n \cdot \log_2(n) + 4 \cdot n.$$

Then the big $\mathcal{O}$ notation is not able to distinguish between the complexity of algorithm A and algorithm B since we have

$$\mathit{count}_A(n) \in \mathcal{O}\big(n \cdot \log_2(n)\big) \quad \text{as well as} \quad \mathit{count}_B(n) \in \mathcal{O}\big(n \cdot \log_2(n)\big).$$

However, by writing

$$count_A(n) = n \cdot \log_2(n) + \mathcal{O}(n) \quad \text{and} \quad count_B(n) = \tfrac{3}{2} \cdot n \cdot \log_2(n) + \mathcal{O}(n)$$

we can abstract from lower order terms while still retaining the leading coefficient of the term determining the complexity.

3.3 Case Study: Efficient Computation of Powers

Let us study an example to clarify the notions introduced so far. Consider the program shown in Figure 3.2. Given an integer m and a natural number n, `power`(m, n) computes m^n. The basic idea is to compute the value of m^n according to the formula

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

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

Figure 3.2: Naive computation of m^n for $m, n \in \mathbb{N}_0$.

This program is obviously correct. The computation of m^n requires $n-1$ multiplication if the function `power` is implemented as shown in Figure 3.2. Fortunately, there is an algorithm for computing m^n that is much more efficient. Consider we have to evaluate m^4. We have

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

If the expression $m \cdot m$ is computed just once, the computation of m^4 needs only two multiplications while the naive approach would already need 3 multiplications. In order to compute m^8 we can proceed according to the following formula:

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

If the expression $(m \cdot m) \cdot (m \cdot m)$ is computed only once, then we need just 3 multiplications in order to compute m^8. On the other hand, the naive approach would take 7 multiplications to compute m^8. The general case is implemented in the program shown in Figure 3.3. In this program, the value of m^n is computed according to the *divide and conquer* paradigm. The basic idea that makes this program work is captured by the following formula:

$$m^n = \begin{cases} m^{n/2} \cdot m^{n/2} & \text{if } n \text{ is even;} \\ m^{n/2} \cdot m^{n/2} \cdot m & \text{if } n \text{ is odd.} \end{cases}$$

It is by no means obvious that the program shown in 3.3 does compute m^n. We prove this claim by *computational induction*. Computational induction is an induction on the number of recursive invocations. This method is the method of choice to prove the correctness of a recursive procedure. The method of computational induction consists of two steps:

1. The *base case*.

```
power := procedure(m, n) {
    if (n == 0) {
        return 1;
    }
    p := power(m, n \ 2);
    if (n % 2 == 0) {
        return p * p;
    } else {
        return p * p * m;
    }
};
```

Figure 3.3: Computation of m^n for $m, n \in \mathbb{N}_0$.

In the base case we have to show that the procedure is correct in all those cases were it does not invoke itself recursively.

2. The *induction step*.

   In the induction step we have to prove that the method works in all those cases were it does invoke itself recursively. In order to prove the correctness of these cases we may assume that the recursive invocations work correctly. This assumption is called the *induction hypotheses*.

Let us prove the claim

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

by computational induction.

1. **Base case**:

   The only case were `power` does not invoke itself recursively is the case $n = 0$. In this case, we have

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

2. **Induction step**:

   The recursive invocation of `power` has the form $\texttt{power}(m, n\backslash 2)$. By the induction hypotheses we know that

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

   holds. After the recursive invocation there are two different cases:

   (a) $n \,\%\, 2 = 0$, therefore n is even.
   Then there exists a number $k \in \mathbb{N}_0$ such that $n = 2 \cdot k$ and therefore $n/2 = k$. Then, we have the following:

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

   (b) $n \,\%\, 2 = 1$, therefore n is odd.

Then there exists a number $k \in \mathbb{N}_0$ such that $n = 2 \cdot k + 1$ and we have $n \backslash 2 = k$, where $n \backslash 2$ denotes integer division of n by 2. In this case we have:

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

As we have $\texttt{power}(m, n) = m^n$ in both cases, the proof is finished. □

Next, we want to investigate the computational complexity of this implementation of `power`. To this end, let us compute the number of multiplications that are done when $\texttt{power}(m, n)$ is called. If the number n is odd there will be more multiplications than in the case when n is even. Let us first investigate the *worst case*. The worst case happens if there is an $l \in \mathbb{N}_0$ such that

$$n = 2^l - 1$$

because then we have

$$n \backslash 2 = 2^{l-1} - 1 \quad \text{and} \quad n \,\%\, 2 = 1,$$

because in that case we have

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

Therefore, if $n = 2^l - 1$ the exponent n will be odd on every recursive call. Therefore, let us assume $n = 2^l - 1$ and let us compute the number a_n of multiplications that are done when $\texttt{power}(m, n)$ is evaluated.

First, we have $a_0 = 0$, because if we have $n = 2^0 - 1 = 0$, then the evaluation of $\texttt{power}(m, n)$ does not require a single multiplication. Otherwise, we have in line 9 two multiplications that have to be added to those multiplications that are performed in the recursive call in line 5. Therefore, we get the following *recurrence relation*:

$$a_n = a_{n \backslash 2} + 2 \qquad \text{for all } n \in \{2^l - 1 \mid l \in \mathbb{N}_0\} \quad \text{and } a_0 = 0.$$

In order to solve this recurrence relation, let us define $b_l := a_{2^l - 1}$. Then, the sequence $(b_l)_l$ satisfies the recurrence relation

$$b_l = a_{2^l - 1} = a_{(2^l - 1) \backslash 2} + 2 = a_{2^{l-1} - 1} + 2 = b_{l-1} + 2 \qquad \text{for all } l \in \mathbb{N}_0$$

and the initial term b_0 satisfies $b_0 = a_{2^0 - 1} = a_0 = 0$. It is quite obvious that the solution of this recurrence relation is given by

$$b_l = 2 \cdot l \qquad \text{for all } l \in \mathbb{N}_0.$$

Those who don't believe me should try to verify this claim by induction. Then, the sequence a_n satisfies

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

Let us solve the equation $n = 2^l - 1$ for l. This yields $l = \log_2(n + 1)$. Substituting this expression in the formula above gives

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

Next, we consider the best case. The computation of $\texttt{power}(m, n)$ needs the least number of multiplications if the test `n % 2 == 0` always evaluates as true. In this case, n must be a power of 2. Therefore, there must exist an $l \in \mathbb{N}_0$ such that we have

$$n = 2^l.$$

Therefore, let us now assume $n = 2^l$ and let us again compute the number a_n of multiplications that are needed to compute `power`(m, n).

First, we have $a_{2^0} = a_1 = 2$, because if $n = 1$, the test `n % 2 == 0` fails and in this case line 9 yields 2 multiplications. Furthermore, in this case line 5 does not add any multiplications since the call `power`$(m, 0)$ immediately returns its result.

Now, if $n = 2^l$ and $n > 1$ then line 7 yields one multiplication that has to be added to those multiplications that are done during the recursive invocation of `power` in line 5. Therefore, we have the following recurrence relation:

$$a_n = a_{n\backslash 2} + 1 \qquad \text{for all } n \in \{2^l \mid l \in \mathbb{N}_0\} \quad \text{and } a_1 = 2.$$

Let us define $b_l := a_{2^l}$. Then the sequence $(b_l)_l$ satisfies the recurrence relation

$$b_l = a_{2^l} = a_{(2^l)\backslash 2} + 1 = a_{2^{l-1}} + 1 = b_{l-1} + 1 \qquad \text{for all } l \in \mathbb{N}_0,$$

and the initial value is given as $b_0 = a_{2^0} = a_1 = 2$. Therefore, we have to solve the recurrence relation

$$b_{l+1} = b_l + 1 \qquad \text{for all } l \in \mathbb{N}_0 \quad \text{with } b_0 = 2.$$

Obviously, the solution is

$$b_l = 2 + l \qquad \text{for all } l \in \mathbb{N}_0.$$

If we substitute this into the definition of b_l in terms of a_l we have:

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

If we solve the equation $n = 2^l$ for l we get $l = \log_2(n)$. Substituting this values leads to

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

Since we have gotten the same result both in the worst case and in the best case we may conclude that in general the number a_n of multiplications satisfies

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

Remark: In reality, we are not interested in the number of multiplications but we are rather interested in the amount of computation time needed by the algorithm given above. However, this computation would be much more tedious because then we would have to take into account that the time needed to multiply to numbers depends on the size of these numbers.

Exercise 9: Implement a procedure `prod` that multiplies two numbers: For given natural numbers m and n, the expression `prod`(m, n) should compute the product $m \cdot n$. Of course, your implementation must not use the multiplication operator "`*`". However, you may use the operators "`\`" and "`%`" provided the second argument of these operators is the number 2. The reason it that division by 2 can be implemented by a simple shift, while $n \,\%\, 2$ is just the last bit of n.

In your implementation, you should use the divide and conquer paradigm. Furthermore, you should use computational induction to prove the correctness of your implementation. Finally, you should provide an estimate for the number of additions needed to compute `prod`(m, n). This estimate should make use of the big $\mathcal{O}$ notation. $\diamond$

3.4 The Master Theorem

In order to analyze the complexity of the procedure *power*(), we first computed a recurrence relation, then we solved this recurrence and, finally, we approximated the result using the big $\mathcal{O}$ notation. If we are only interested in this last approximation then, in many cases, it is not necessary to solve the recurrence relation. Instead, we can use the *master theorem*. We present a simplified version of this theorem.

Theorem 12 (Master Theorem) Assume that

1. $\alpha, \beta \in \mathbb{N}$ such that $\beta \geq 2$,
2. $\delta \in \mathbb{R}$ and $\delta \geq 0$,
3. the function $f : \mathbb{N}_0 \to \mathbb{R}_+$ satisfies the recurrence relation
$$f(n) = \alpha \cdot f\left(n \backslash \beta\right) + \mathcal{O}(n^\delta),$$
where $n \backslash \beta$ denotes *integer division*[1] of n by β.

Then we have the following:

1. $\alpha < \beta^\delta \to f(n) \in \mathcal{O}\big(n^\delta\big)$,
2. $\alpha = \beta^\delta \to f(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big)$,
3. $\alpha > \beta^\delta \to f(n) \in \mathcal{O}\big(n^{log_\beta(\alpha)}\big)$.

Proof: We will compute an upper bound for the expression $f(n)$ but in order to keep our exposition clear and simple we will only discuss the case where n is a power of β, that is n has the form

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

The general case is similar, but is technically much more involved. Observe that the equation $n = \beta^k$ implies $k = \log_\beta(n)$. We will need this equation later. We define $a_k := f(n) = f\big(\beta^k\big)$. Then the recurrence relation for the function f is transformed into a recurrence relation for the sequence a_k as follows:

$$\begin{aligned}
a_k &= f\big(\beta^k\big) \\
&= \alpha \cdot f\big(\beta^k \backslash \beta\big) + \mathcal{O}\Big(\big(\beta^k\big)^\delta\Big) \\
&= \alpha \cdot f\big(\beta^{k-1}\big) + \mathcal{O}\big(\beta^{k\cdot\delta}\big) \\
&= \alpha \cdot a_{k-1} + \mathcal{O}\big(\beta^{k\cdot\delta}\big) \\
&= \alpha \cdot a_{k-1} + \mathcal{O}\Big(\big(\beta^\delta\big)^k\Big)
\end{aligned}$$

In order to simplify this recurrence relation, let us define

$$\gamma := \beta^\delta.$$

Then, the recurrence relation for the sequence a_k can be written as

$$a_k = \alpha \cdot a_{k-1} + \mathcal{O}\big(\gamma^k\big).$$

Let us substitute $k-1$ for k in this equation. This yields

$$a_{k-1} = \alpha \cdot a_{k-2} + \mathcal{O}\big(\gamma^{k-1}\big).$$

[1] For given integers $a, b \in \mathbb{N}$, the *integer division* $a \backslash b$ is defined as the biggest number $q \in \mathbb{N}$ such that $q \cdot b \leq a$. It can be implemented via the formula $a \backslash b = \mathit{floor}(a/b)$, where $\mathit{floor}(x)$ rounds x down to the nearest integer.

Next, we plug the value of a_{k-1} into the equation for a_k. This yields

$$\begin{aligned} a_k &= \alpha \cdot a_{k-1} + \mathcal{O}(\gamma^k) \\ &= \alpha \cdot \Big(\alpha \cdot a_{k-2} + \mathcal{O}(\gamma^{k-1})\Big) + \mathcal{O}(\gamma^k) \\ &= \alpha^2 \cdot a_{k-2} + \alpha \cdot \mathcal{O}(\gamma^{k-1}) + \mathcal{O}(\gamma^k). \end{aligned}$$

We observe that

$$a_{k-2} = \alpha \cdot a_{k-3} + \mathcal{O}(\gamma^{k-2})$$

holds and substitute the right hand side of this equation into the previous equation. This yields

$$\begin{aligned} a_k &= \alpha^2 \cdot a_{k-2} + \alpha \cdot \mathcal{O}(\gamma^{k-1}) + \mathcal{O}(\gamma^k) \\ &= \alpha^2 \cdot \Big(\alpha \cdot a_{k-3} + \mathcal{O}(\gamma^{k-2})\Big) + \alpha \cdot \mathcal{O}(\gamma^{k-1}) + \mathcal{O}(\gamma^k) \\ &= \alpha^3 \cdot a_{k-3} + \alpha^2 \cdot \mathcal{O}(\gamma^{k-2}) + \alpha \cdot \mathcal{O}(\gamma^{k-1}) + \alpha^0 \cdot \mathcal{O}(\gamma^k). \end{aligned}$$

Proceeding in this way we arrive at the general formula

$$\begin{aligned} a_k &= \alpha^{i+1} \cdot a_{k-(i+1)} + \alpha^i \cdot \mathcal{O}(\gamma^{k-i}) + \alpha^{i-1} \cdot \mathcal{O}(\gamma^{k-(i-1)}) + \cdots + \alpha^0 \cdot \mathcal{O}(\gamma^k) \\ &= \alpha^{i+1} \cdot a_{k-(i+1)} + \sum_{j=0}^{i} \alpha^j \cdot \mathcal{O}(\gamma^{k-j}). \end{aligned}$$

If we take this formula and substitute $i+1 := k$, i. e. $i := k-1$, then we conclude

$$\begin{aligned} a_k &= \alpha^k \cdot a_0 + \sum_{j=0}^{k-1} \alpha^j \cdot \mathcal{O}(\gamma^{k-j}) \\ &= \alpha^k \cdot a_0 + \mathcal{O}\Big(\gamma^k \cdot \sum_{j=0}^{k-1} \Big(\frac{\alpha}{\gamma}\Big)^j\Big). \end{aligned}$$

At this point we have to remember the formula for the geometric series. This formula reads

$$\sum_{j=0}^{n} q^j = \frac{q^{n+1}-1}{q-1} \quad \text{provided } q \neq 1, \text{ while}$$

$$\sum_{j=0}^{n} q^j = n+1 \quad \text{if } q = 1.$$

For the geometric series given above, $q = \dfrac{\alpha}{\gamma}$. In order to proceed, we have to perform a case distinction:

1. Case: $\alpha < \gamma$, i. e. $\alpha < \beta^\delta$.

   In this case, the series $\sum_{j=0}^{k-1} \big(\frac{\alpha}{\gamma}\big)^j$ is bounded by the value

   $$\sum_{j=0}^{\infty} \Big(\frac{\alpha}{\gamma}\Big)^j = \frac{1}{1-\frac{\alpha}{\gamma}}.$$

   Since this value does not depend on k and the big $\mathcal{O}$ notation abstracts from constant factors, we are able to drop the sum. Therefore, we have

   $$a_k = \alpha^k \cdot a_0 + \mathcal{O}(\gamma^k).$$

   Furthermore, let us observe that, since $\alpha < \gamma$ we have that

$$\alpha^k \cdot a_0 \in \mathcal{O}\big(\gamma^k\big).$$

Therefore, the term $\alpha^k \cdot a_0$ is subsumed by $\mathcal{O}\big(\gamma^k\big)$ and we have shown that

$$a_k \in \mathcal{O}\big(\gamma^k\big).$$

The variable γ was defined as $\gamma = \beta^\delta$. Furthermore, by definition of k and a_k we have

$$k = \log_\beta(n) \quad \text{and} \quad f(n) = a_k.$$

Therefore we have

$$f(n) \in \mathcal{O}\Big(\big(\beta^\delta\big)^{\log_\beta(n)}\Big) = \mathcal{O}\Big(\big(\beta^{\log_\beta(n)}\big)^\delta\Big) = \mathcal{O}\big(n^\delta\big).$$

Thus we have shown the following:

$$\alpha < \beta^\delta \to f(n) \in \mathcal{O}\big(n^\delta\big).$$

2. Case: $\alpha = \gamma$, i. e. $\alpha = \beta^\delta$.

In this case, all terms in the series $\sum_{j=0}^{k-1} \big(\frac{\alpha}{\gamma}\big)^j$ have the value 1 and therefore we have

$$\sum_{j=0}^{k-1} \big(\tfrac{\alpha}{\gamma}\big)^j = \sum_{j=0}^{k-1} 1 = k.$$

Therefore, we have

$$a_k = \alpha^k \cdot a_0 + \mathcal{O}\big(k \cdot \gamma^k\big).$$

Furthermore, let us observe that, since $\alpha = \gamma$ we have that

$$\alpha^k \cdot a_0 \in \mathcal{O}\big(k \cdot \gamma^k\big).$$

Therefore, the term $\alpha^k \cdot a_0$ is subsumed by $\mathcal{O}\big(k \cdot \gamma^k\big)$ and we have shown that

$$a_k \in \mathcal{O}\big(k \cdot \gamma^k\big).$$

We have $\gamma = \beta^\delta$, $k = \log_\beta(n)$, and $f(n) = a_k$. Therefore,

$$f(n) \in \mathcal{O}\Big(\log_\beta(n) \cdot \big(\beta^\delta\big)^{\log_\beta(n)}\Big) = \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big).$$

Thus we have shown the following:

$$\alpha = \beta^\delta \to f(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big).$$

3. Case: $\alpha > \gamma$, i. e. $\alpha > \beta^\delta$.

In this case we have

$$\sum_{j=0}^{k-1} \big(\tfrac{\alpha}{\gamma}\big)^j = \frac{\big(\frac{\alpha}{\gamma}\big)^k - 1}{\frac{\alpha}{\gamma} - 1} \in \mathcal{O}\Big(\big(\tfrac{\alpha}{\gamma}\big)^k\Big).$$

Therefore, we have

$$a_k = \alpha^k \cdot a_0 + \mathcal{O}\Big(\gamma^k \cdot \big(\tfrac{\alpha}{\gamma}\big)^k\Big) = \alpha^k \cdot a_0 + \mathcal{O}\big(\alpha^k\big).$$

Since $\alpha^k \cdot a_0 \in \mathcal{O}\big(\alpha^k\big)$, we have shown that

$$a_k \in \mathcal{O}\big(\alpha^k\big).$$

Since $k = \log_\beta(n)$ and $f(n) = a_k$ we have

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

Next, we observe that

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

holds. This equation is easily proven by taking the logarithm with base β on both sides of the equation. Using this equation we conclude that

$$\alpha > \beta^\delta \to f(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)}\big)$$

holds. □

Example: Assume that f satisfies the recurrence relation

$$f(n) = 9 \cdot f(n\backslash 3) + n.$$

Define $\alpha := 9$, $\beta := 3$, and $\delta := 1$. Then we have

$$\alpha = 9 > 3^1 = \beta^\delta.$$

This is the last case of the master theorem and, since

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

we conclude that

$$f(n) \in \mathcal{O}(n^2) \quad \text{holds.}$$ □

Example: Assume that the function $f(n)$ satisfies the recurrence relation

$$f(n) = f(n\backslash 2) + 2.$$

We want to analyze the asymptotic growth of f with the help of the master theorem. Defining $\alpha := 1$, $\beta := 2$, $\delta = 0$ and noting that $2 \in \mathcal{O}(n^0)$ we see that the recurrence relation for f can be written as

$$f(n) = \alpha \cdot f(n\backslash \beta) + \mathcal{O}(n^\delta).$$

Furthermore, we have

$$\alpha = 1 = 2^0 = \beta^\delta.$$

Therefore, the second case of the master theorem tells us that

$$f(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big) = \mathcal{O}\big(\log_2(n) \cdot n^0\big) = \mathcal{O}\big(\log_2(n)\big).$$ □

Example: This time, f satisfies the recurrence relation

$$f(n) = 3 \cdot f(n\backslash 4) + n^2.$$

Define $\alpha := 3$, $\beta := 4$, and $\delta := 2$. Then we have

$$f(n) = \alpha \cdot f(n\backslash \beta) + \mathcal{O}(n^\delta).$$

Since this time we have

$$\alpha = 3 < 16 = \beta^\delta$$

the first case of the master theorem tells us that

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

Example: This next example is a slight variation of the previous example. Assume f satisfies the recurrence relation

$$f(n) = 3 \cdot f(n\backslash 4) + n \cdot \log_2(n).$$

Again, define $\alpha := 3$ and $\beta := 4$. This time we define $\delta := 1 + \varepsilon$ where ε is some small positive number that will be defined later. You can think of ε being $\frac{1}{2}$ or $\frac{1}{5}$ or even $\frac{1}{42}$. Since the logarithm of n grows slower than any positive power of n we have

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

We conclude that

$$n \cdot \log_2(n) \in \mathcal{O}\big(n \cdot n^\varepsilon\big) = \mathcal{O}\big(n^{1+\varepsilon}\big) = \mathcal{O}\big(n^\delta\big).$$

Therefore, we have

$$f(n) = \alpha \cdot f(n \backslash \beta) + \mathcal{O}(n^\delta).$$

Furthermore, we have

$$\alpha = 3 < 4 < 4^\delta = \beta^\delta.$$

Therefore, the first case of the master theorem tells us that

$$f(n) \in \mathcal{O}\big(n^{1+\varepsilon}\big) \quad \text{holds for all } \varepsilon > 0.$$

Hence, we have shown that

$$f(n) \in \mathcal{O}\big(n^{1+\frac{1}{2}}\big), \quad f(n) \in \mathcal{O}\big(n^{1+\frac{1}{5}}\big), \quad \text{and even} \quad f(n) \in \mathcal{O}\big(n^{1+\frac{1}{42}}\big)$$

holds. Using a stronger form of the master theorem it can be shown that

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

holds. This example shows that the master theorem, as given in these lecture notes, does not always produce the most precise estimate for the asymptotic growth of a function. □

Exercise 10: For each of the following recurrence relations, use the master theorem to give estimates of the growth of the function f.

1. $f(n) = 4 \cdot f(n \backslash 2) + 2 \cdot n + 3.$
2. $f(n) = 4 \cdot f(n \backslash 2) + n^2.$
3. $f(n) = 3 \cdot f(n \backslash 2) + n^3.$ ◇

Exercise 11: Consider the recurrence relation

$$f(n) = 2 \cdot f(n \backslash 2) + n \cdot \log_2(n).$$

How can you bound the growth of f using the master theorem?

Optional: Assume that n has the form $n = 2^k$ for some natural number k. Furthermore, you are told that $f(1) = 1$. Solve the recurrence relation in this case. ◇

3.5 Variants of Big $\mathcal{O}$ Notation

The big $\mathcal{O}$ notation is useful if we want to express that some function f does not grow faster than another function g. Therefore, when stating the running time of the worst case of some algorithm, big $\mathcal{O}$ notation is the right tool to use. However, sometimes we want to state a lower bound for the complexity of a problem. For example, it can be shown that every comparison based sort algorithm needs at least $n \cdot \log_2(n)$ comparisons to sort a list of length n. In order to be able to express lower bounds concisely, we introduce the big Ω notation next.

Definition 13 ($\Omega(g)$) Assume $g \in \mathbb{R}_+^{\mathbb{N}_0}$ is given. Let us define the set of all functions that grow at least as fast as the function g as follows:

$$\Omega(g) := \big\{ f \in \mathbb{R}_+^{\mathbb{N}_0} \mid \exists k \in \mathbb{N}_0 \colon \exists c \in \mathbb{R}_+ \colon \forall n \in \mathbb{N}_0 \colon n \geq k \to c \cdot g(n) \leq f(n) \big\}.$$ □

It is not difficult to show that

$$f \in \Omega(g) \quad \text{if and only if} \quad g \in \mathcal{O}(f).$$

Finally, we introduce big Θ notation. The idea is that $f \in \Theta(g)$ if f and g have the same asymptotic growth rate.

Definition 14 ($\Theta(g)$) Assume $g \in \mathbb{R}_{+0}^{\mathbb{N}}$ is given. The set of functions that have the same asymptotic growth rate as the function g is defined as

$$\Theta(g) := \mathcal{O}(g) \cap \Omega(g).$$ □

It can be shown that $f \in \Theta(g)$ if and only if the limit

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

exists and is greater than 0.

Sedgewick [SW11a] claims that the Θ notation is too imprecise and advocates the tilde notation instead. For two functions $f, g : \mathbb{N} \to \mathbb{R}_+$ he defines

$$f \sim g \quad \text{iff} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

To see why this is more precise, let us consider the case of two algorithms A and B for sorting a list of length n. Assume that the number $count_A(n)$ of comparisons used by algorithm A to sort a list of length n is given as

$$count_A(n) = n \cdot \log_2(n) + 7 \cdot n,$$

while for algorithm B the corresponding number of comparisons is given as

$$count_B(n) = \tfrac{3}{2} \cdot n \cdot \log_2(n) + 4 \cdot n.$$

Clearly, if n is big then algorithm A is better than algorithm B but as we have pointed out in a previous section, the big $\mathcal{O}$ notation is not able to distinguish between the complexity of algorithm A and algorithm B. However we have that

$$\tfrac{3}{2} \cdot count_A(n) \sim count_B(n)$$

and this clearly shows that for big values of n, algorithm A is faster than algorithm B by a factor of $\frac{3}{2}$.

3.6 Further Reading

Chapter 3 of the book "*Introduction to Algorithms*" by Cormen et. al. [CLRS09] contains a detailed description of several variants of the big O notation, while chapter 4 gives a more general version of the master theorem together with a detailed proof.

Chapter 4

Hoare Logic

In this chapter we introduce *Hoare logic*. This is a formal system that is used to prove the correctness of imperative computer programs. Hoare logic has been introduced 1969 by Sir Charles Antony Richard Hoare, who is the inventor of the quicksort algorithm.

4.1 Preconditions and Postconditions

Hoare logic is based on preconditions and postconditions. If `P` is a program fragment and if F and G are logical formulæ, then we call F a precondition and G a postcondition for the program fragment `P` if the following holds: If `P` is executed in a state s such that the formula F holds in s, then the execution of `P` will change the state s into a new state s' such that G holds in s', This is written as

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

We will read this notation as "*executing P changes F into G*". The formula

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

is called a *Hoare tripel*.

Examples:

1. The assignment "`x := 1;`" satisfies the specification

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

   Here, the precondition is the trivial condition "`true`", since the postcondition "$x = 1$" will always be satisfied after this assignment.

2. The assignment "`x = x + 1;`" satisfies the specification

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

   If the precondition is "$x = 1$", then it is obvious that the postcondition has to be "$x = 2$".

3. Let us consider the assignment "`x = x + 1;`" again. However, this time the precondition is given as "$prime(x)$", which is only true if x is a prime number. This time, the Hoare tripel is given as

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

   This might look strange at first. Many students think that this Hoare triple

should rather be written as

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

However, this can easily be refuted by taking x to have the value 2. Then, the precondition $prime(x)$ is satisfied since 2 is a prime number. After the assignment, x has the value 3 and

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

still is a prime number. However, we also have

$$x + 1 = 3 + 1 = 4$$

and as $4 = 2 \cdot 2$ we see that $x + 1$ is not a prime number!

Let us proceed to show how the different parts of a program can be specified using Hoare tripels. We start with the analysis of assignments.

4.1.1 Assignments

Let us generalize the previous example. Let us therefore assume that we have an assignment of the form

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

and we want to investigate how the postcondition G of this assignment is related to the precondition F. To simplify matters, let us assume that the function h is invertible, i. e. we assume that there is a function h^{-1} such that we have

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

for all x. Then, the function h^{-1} is the inverse of the function h. In order to understand the problem of computing the postcondition for the assignment statement given above, let us first consider an example. The assignment

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

can be written as

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

where the function h is given as

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

and the inverse function h^{-1} is

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

Now we are able to compute the postcondition of the assignment "`x := h(x);`" from the precondition. We have

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

Here, $F\sigma$ denotes the application of the substitution σ to the formula F. The expression $F\sigma$ is computed from the expression F by replacing every occurrence of the variable x by the term $h^{-1}(x)$. Therefore, the substitution σ undoes the effect of the assignment and restores the variables in F to the state before the assignment.

In order to understand why this is the correct way to compute the postcondition, we consider the assignment "`x := x + 1`" again and choose the formula $x = 7$ as precondition. Since $h^{-1}(x) = x - 1$, the substitution σ is given as $\sigma = [x \mapsto x - 1]$. Therefore, $F\sigma$ has the form

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

I have used the symbol "≡" here in order to express that these formulæ are syntactically identical. Therefore, we have

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

Since the formula $x - 1 = 7$ is equivalent to the formula $x = 8$ the Hoare tripel above can be rewritten as

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

and this is obviously correct: If the value of x is 7 before the assignment

"`x := x + 1;`"

is executed, then after the assignment is executed, x will have the value 8.

Let us try to understand why

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

is, indeed, correct: Before the assignment "x := h(x);" is executed, the variable x has some fixed value x_0. The precondition F is valid for x_0. Therefore, the formula $F[x \mapsto x_0]$ is valid before the assignment is executed. However, the variable x does not occur in the formula $F[x \mapsto x_0]$ because it has been replaced by the fixed value x_0. Therefore, the formula

$$F[x \mapsto x_0]$$

remains valid after the assignment "`x = h(x);`" is executed. After this assignment, the variable x is set to $h(x_0)$. Therefore, we have

$$x = h(x_0).$$

Let us solve this equation for x_0. We find

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

Therefore, after the assignment the formula

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

is valid and this is the formula that is written as $F\sigma$ above.

We conclude this discussion with another example. The unary predicate *prime* checks whether its argument is a prime number. Therefore, $\mathit{prime}(x)$ is true if x is a prime number. Then we have

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

The correctness of this Hoare triple should be obvious: If x is a prime and if x is then incremented by 1, then afterwards $x - 1$ is prime.

Different Forms of Assignments Not all assignments can be written in the form "`x := h(x);`" where the function h is invertible. Often, a constant c is assigned to some variable x. If x does not occur in the precondition F, then we have

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

The formula F can be used to restrict the values of other variables occurring in the program under consideration.

General Form of the Assignment Rule In the literature the rule for specifying an assignment is given as

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

Here, t is an arbitrary term that can contain the variable x. This rule can be read

as follows:

> "*If the formula $F(t)$ is valid in some state and t is assigned to x, then after this assignment we have $F(x)$.*"

This rule is obviously correct. However, it is not very useful because in order to apply this rule we first have to rewrite the precondition as $F(t)$. If t is some complex term, this is often very difficult to do.

4.1.2 The Weakening Rule

If a program fragment P satisfies the specification

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

and if, furthermore, the formula G implies the validity of the formula H, that is if

$$G \rightarrow H$$

holds, then the program fragment P satisfies

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

The reasoning is as follows: If after executing P we know that G is valid, then, since G implies H, the formula H has to be valid, too. Therefore, the following *verification rule*, which is known as the *weakening rule*, is valid:

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

The formulæ written over the fraction line are called the *premisses* and the formula under the fraction line is called the *conclusion*. The conclusion and the first premiss are Hoare tripels, the second premiss is a formula of first order logic. The interpretation of this rule is that the conclusion is true if the premisses are true.

4.1.3 Compound Statements

If the program fragments `P` and `Q` have the specifications

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

and if, furthermore, the postcondition G_1 implies the precondition F_2, then the composition `P;Q` of `P` and `Q` satisfies the specification

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

The reasoning is as follows: If, initially, F_1 is satisfied and we execute P then we have G_1 afterwards. Therefore we also have F_2 and if we now execute Q then afterwards we will have G_2. This chain of thoughts is combined in the following verification rule:

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

If the formulæ G_1 and F_2 are identical, then this rule can be simplified as follows:

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

Example: Let us analyse the program fragment shown in Figure 4.1. We start our analysis by using the precondition

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

Here, a and b are two variables that we use to store the initial values of `x` and `y`. The first assignment yields the Hoare triple

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

where $\sigma = [x \mapsto x + y]$. The form of σ follows from the fact that the function $x \mapsto x + y$ is the inverse of the function $x \mapsto x - y$. If we apply σ to the formula $x = a \wedge y = b$ we get

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

The second assignment yields the Hoare triple

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

where $\sigma = [y \mapsto y - x]$. The reason is that the function $y \mapsto y - x$ is the inverse of the function $y \mapsto y + x$. This time, we get

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

Simplifying the postcondition yields

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

Let us consider the last assignment. We have

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

where $\sigma = [x \mapsto y - x]$, since the function $x \mapsto y - x$ is the inverse of the function $x \mapsto y - x$. This yields

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

Simplifying the postcondition gives

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

Combining the Hoare tripels (4.1), (4.2) and (4.3) we get

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

The Hoare tripel (4.4) shows that the program fragment shown in Figure 4.1 swaps the values of the variables x and y: If the value of x is a and y has the value b before the program is executed, then afterwards y has the value a and x has the value b. The trick shown in Figure 4.1 can be used to swap variables without using an auxiliary variable. This is useful because when this code is compiled into machine language, the resulting code will only use two registers.

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

Figure 4.1: A tricky way to swap variables.

4.1.4 Conditional Statements

In order to compute the effect of a conditional of the form

`if (`B`) { `P` } else { Q }`

let us assume that before the conditional statement is executed, the precondition F is satisfied. We have to analyse the effect of the program fragments P and Q. The program fragment P is only executed when B is true. Therefore, the precondition for P is $F \wedge B$. On the other hand, the precondition for the program fragment Q is $F \wedge \neg B$, since Q is only executed if B is false. Hence, we have the following verification rule:

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

In this form, the rule is not always applicable. The reason is that the analysis of the program fragments `P` and `Q` yields Hoare tripel of the form

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

and in general G_1 and G_2 will be different from each other. In order to be able to apply the rule for conditionals we have to find a formula G that is a consequence of G_1 and also a consequence of G_2, i. e. we want to have

$$G_1 \rightarrow G \qquad \text{and} \qquad G_2 \rightarrow G.$$

If we find G, then the weakening rule can be applied to conclude the validity of

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

and this gives us the premisses that are needed for the rule (4.5).

Example: Let us analyze the following program fragment:

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

We start with the precondition

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

and want to show that the execution of the conditional establishes the postcondition

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

The first assignment "`z := x;`" gives the Hoare tripel

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

In the same way, the second assignment "`z := y`" yields

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

Since we have

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

and also

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

Using the weakening rule we conclude that

$$\begin{array}{lll} \big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y}\big\} & \texttt{z := x;} & \big\{\texttt{z} = \min(a, b)\big\} \qquad \text{and} \\ \big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y}\big\} & \texttt{z := y;} & \big\{\texttt{z} = \min(a, b)\big\} \end{array}$$

holds. Now we can apply the rule for the conditional and conclude that

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

holds. Thus we have shown that the program fragment above computes the minimum of the numbers a and b.

4.1.5 Loops

Finally, let us analyze the effect of a loop of the form

`while (`B`) { P }`

The important point here is that the postcondition of the n-th execution of the body of the loop P is the precondition of the $(n+1)$-th execution of P. Basically this means that the precondition and the postcondition of P have to be more or less the same. Hence, this condition is called the *loop invariant*. Therefore, the details of the verification rule for `while` loops are as follows:

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

The premiss of this rules expresses the fact that the invariant I remains valid on execution of P. However, since P is only executed as long as B is `true`, the precondition for P is actually the formula $I \wedge B$. The conclusion of the rule says that if the invariant I is true before the loop is executed, then I will be true after the loop has finished. This result is intuitive since every time P is executed I remains valid. Furthermore, the loop only terminates once B gets `false`. Therefore, the postcondition of the loop can be strengthened by adding $\neg B$.

4.2 The Euclidean Algorithm

In this section we show how the verification rules of the last section can be used to prove the correctness of a non-trivial program. We will show that the algorithm shown in Figure 4.5 on page 46 is correct. The procedure shown in this figure implements the *Euclidean algorithm* to compute the greatest common divisor of two natural numbers. Our proof is based on the following property of the function `gcd`:

$$\texttt{gcd}(x + y, y) = \texttt{gcd}(x, y) \quad \text{for all } x, y \in \mathbb{N}.$$

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

Figure 4.2: The Euclidean Algorithm to compute the greatest common divisor.

4.2.1 Correctness Proof of the Euclidean Algorithm

To start our correctness proof we formulate the invariant of the `while` loop. Let us define

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

In this formula we have defined the initial values of x and y as a and b. In order to

establish the invariant at the beginning we have to ensure that the function `gcd` is only called with positive natural numbers. If we denote these numbers as a and b, then the invariant I is valid initially. The reason is that $x = a$ and $y = b$ implies $\texttt{gcd}(x, y) = \texttt{gcd}(a, b)$.

In order to prove that the invariant I is maintained in the loop we formulate the Hoare triples for both alternatives of the conditional. For the first conditional we know that

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

holds, where σ is defined as $\sigma = [y \mapsto y + x]$. Here, the condition $x \neq y$ is the condition controlling the execution of the `while` loop and the condition $x < y$ is the condition of the `if` conditional. We rewrite the formula $(I \wedge x \neq y \wedge x < y)\sigma$:

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

In the last step we have used the formula

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

and we have simplified the inequality $x < y + x$ as $0 < y$. The last formula implies

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

However, this is precisely the invariant I. Therefore we have shown that

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

holds. Next, let us consider the second alternative of the `if` conditional. We have

$$\big\{I \wedge x \neq y \wedge x \geq y\big\} \quad \texttt{x := x - y;} \quad \big\{(I \wedge x \neq y \wedge x \geq y)\sigma\big\}$$

where $\sigma = [x \mapsto x + y]$. The expression $(I \wedge x \neq y \wedge x \geq y)\sigma$ is rewritten as follows:

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

The last formula implies that

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

holds. Again, this is our invariant I. Therefore we have shown that

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

holds. If we use the Hoare triples (4.7) and (4.8) as premisses for the rule for conditionals we have shown that

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

holds. Now the verification rule for `while` loops yields

$\{I\}$

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

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

Expanding the invariant I in the formula $I \wedge x = y$ shows that the postcondition of the `while` loop is given as

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

Now the correctness of the Euclidean algorithm can be established as follows:

$$\begin{array}{ll} & x > 0 \wedge y > 0 \wedge \texttt{gcd}(x, y) = \texttt{gcd}(a, b) \wedge x = y \\ \Rightarrow & \texttt{gcd}(x, y) = \texttt{gcd}(a, b) \wedge x = y \\ \Rightarrow & \texttt{gcd}(x, x) = \texttt{gcd}(a, b) \\ \Rightarrow & x = \texttt{gcd}(a, b) \qquad \text{because} \quad \texttt{gcd}(x, x) = x. \end{array}$$

All in all we have shown the following: If the `while` loop terminates, then the variable x will be set to the greatest common divisor of a and b, where a and b are the initial values of the variables x and y. In order to finish our correctness proof we have to show that the `while` loop does indeed terminate for all choices of a and b. To this end let us define the variable s as follows:

$$s := x + y.$$

The variables x and y are natural numbers. Therefore s is a natural number, too. Every iteration of the loop reduces the number s: either x is subtracted from s or y is subtracted from s and the invariant I shows that both x and y are positive. Therefore, if the `while` loop would run forever, at some point s would get negative. Since s can not be negative, the loop must terminate. Hence we have shown the correctness of the Euclidean algorithm.

Exercise 12: Show that the function `power`(x, y) that is defined in Figure 4.3 does compute x^y, i. e. show that `power`$(x, y) = x^y$ for all natural numbers x and y.

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

Figure 4.3: A program to compute x^y iteratively.

Hints:

1. If the initial values of x and y are called a and b, then an invariant for the `while` loop is given as
$$I := \big(r \cdot x^y = a^b\big).$$
2. The verification rule for the conditional without `else` is given as
$$\frac{\{F \wedge B\} \quad \texttt{P} \quad \{G\}, \qquad F \wedge \neg B \rightarrow G}{\{F\} \quad \texttt{if (}B\texttt{) \{ P \}} \quad \{G\}}$$
This rule is interpreted as follows:
   (a) If both the precondition F and the condition B is valid, then execution of the program fragment P has to establish the validity of the postcondition G.
   (b) If the precondition F is valid but we have $\neg B$, then this must imply the postcondition G.

Remark: Proving the correctness of a nontrivial program is very tedious. Therefore, various attempts have been made to automate the task. For example, *KeY Hoare* is a tool that can be used to verify the correctness of programs. It is based on Hoare calculus.

4.3 Symbolic Program Execution

The last section has shown that using Hoare logic to verify a program can be quite difficult. There is another method to prove the correctness of imperative programs. This method is called *symbolic program execution*. Let us demonstrate this method. Consider the program shown in Figure 4.4.

The main difference between a mathematical formula and a program is that in a formula all occurrences of a variable refer to the same value. This is different in a program because the variables change their values dynamically. In order to deal with this property of program variables we have to be able to distinguish the

```
1   power := procedure(x_0, y_0) {
2       r_0 := 1;
3       while (y_n > 0) {
4           if (y_n % 2 == 1) {
5               r_{n+1} := r_n * x_n;
6           }
7           x_{n+1} := x_n * x_n;
8           y_{n+1} := y_n \ 2;
9       }
10      return r_N;
11  };
```

Figure 4.4: An annotated programm to compute powers.

different occurrences of a variable. To this end, we index the program variables. When doing this we have to be aware of the fact that the same occurrence of a program variable can still denote different values if the variable occurs inside a loop. In this case we have to index the variables in a way that the index includes a counter that counts the number of loop iterations. For concreteness, consider the program shown in Figure 4.4. Here, in line 5 the variable `r` has the index n on the right side of the assignment, while it has the index r_{n+1} on the left side of the assignment in line 5. Here, n denotes the number of times the `while` loop has been iterated. After the loop in line 10 the variable is indexed as $\mathtt{r}_N$, where N denotes the total number of loop iterations. We show the correctness of the given program next. Let us define

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

We show, that the `while` loop satisfies the invariant

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

This claim is proven by induction on the number of loop iterations.

B.C. $n = 0$: Since we have $r_0 = 1$, $x_0 = a$, and $y_0 = b$ we have

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

I.S. $n \mapsto n + 1$: We need a case distinction with respect to y `%` 2:

(a) y_n `%` $2 = 1$. Then we have $y_n = 2 \cdot (y_n \backslash 2) + 1$ and $r_{n+1} = r_n \cdot x_n$. Hence

$$\begin{aligned} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = \;& (r_n \cdot x_n) \cdot (x_n \cdot x_n)^{y_n \backslash 2} \\ = \;& r_n \cdot x_n^{2 \cdot (y_n \backslash 2) + 1} \\ = \;& r_n \cdot x_n^{y_n} \\ \overset{i.h.}{=} \;& a^b \end{aligned}$$

(b) $y_n \,\%\, 2 = 0$. Then we have $y_n = 2 \cdot (y_n \backslash 2)$ and $r_{n+1} = r_n$. Therefore

$$\begin{aligned} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = \;& r_n \cdot (x_n \cdot x_n)^{y_n \backslash 2} \\ = \;& r_n \cdot x_n^{2 \cdot (y_n \backslash 2)} \\ = \;& r_n \cdot x_n^{y_n} \\ \overset{i.h.}{=} \;& a^b \end{aligned}$$

This shows the validity of the equation (4.9). If the `while` loop terminates, we must have $y_N = 0$. If $n = N$, then equation (4.9) yields:

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

This shows $r_N = a^b$ and since we already know that the `while` loop terminates, we have proven that $\texttt{power}(a, b) = a^b$.

Exercise 13: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 4.5. During the proof you should make use of the fact that for all positive natural numbers a and b the equation

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

is valid.

```
1  gcd := procedure(a, b) {
2      while (b != 0) {
3          [a, b] := [b, a % b];
4      }
5      return a;
6  };
```

Figure 4.5: An efficient version of the Euclidean algorithm.

Chapter 5

Sorting

In this chapter, we assume that we have been given a list l. The elements of l are members of some set S. If we want to *sort* the list l we have to be able to compare these elements to each other. Therefore, we assume that S is equipped with a binary relation $\leq$ which is *reflexive*, *anti-symmetric* and *transitive*, i. e. we have

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

A pair $\langle S, \leq \rangle$ where S is a set and $\leq \subseteq S \times S$ is a relation on S that is *reflexive*, *anti-symmetric* and *transitive* is called a *partially ordered set*. If, furthermore

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

holds, then the pair $\langle S, \leq \rangle$ is called a *totally ordered set*.

Examples:

1. $\langle \mathbb{N}, \leq \rangle$ is a totally ordered set.
2. $\langle 2^{\mathbb{N}}, \subseteq \rangle$ is a partially ordered set but it is not a totally ordered set. For example, the sets $\{1\}$ and $\{2\}$ are not comparable since we have

   $$\{1\} \not\subseteq \{2\} \quad \text{and} \quad \{2\} \not\subseteq \{1\}.$$

3. If P is the set of employees of some company and if we define for given employees $a, b \in P$

   $$a \preceq b \quad \text{iff} \quad a \text{ does not earn more than } b,$$

   then the $\langle P, \leq \rangle$ is not a partially ordered set. The reason is that the relation $\preceq$ is not anti-symmetric: If Mr. Smith earns as much as Mrs. Robinson, then we have both

   $$\text{Smith} \preceq \text{Robinson} \quad \text{and} \quad \text{Robinson} \preceq \text{Smith}$$

   but obviously Smith $\neq$ Robinson.

In the examples given above we see that it does not make sense to sort subsets of $\mathbb{N}$. However, we can sort natural numbers with respect to their size and we can also sort employees with respect to their income. This show that, in order to sort, we do not necessarily need a totally ordered set. In order to capture the requirements that are needed to be able to sort we introduce the notion of a *quasiorder*.

Definition 15 (Quasiorder)
A pair $\langle S, \preceq \rangle$ is a quasiorder *if $\preceq$ is a binary relation on S such that we have the following:*

1. $\forall x \in S: x \preceq x$. *(reflexivity)*
2. $\forall x, y, z \in S: \big(x \preceq y \wedge y \preceq z \rightarrow x \preceq z\big)$. *(transitivity)*

If, furthermore,

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

holds, then $\langle S, \preceq \rangle$ is called a total quasiorder*. This will be abbreviated as* TQO.

A quasiorder $\langle S, \preceq \rangle$ does not require the relation $\preceq$ to be anti-symmetric. Nevertheless, the notion of a quasiorder is very closely related to the notion of a partial order. The reason is as follows: If $\langle S, \preceq \rangle$ is a quasiorder, then we can define an equivalence relation $\approx$ on S by setting

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

If we extend the order $\preceq$ to the equivalence classes generated by the relation $\approx$, then it can be shown that this extension is a partial order.

Let us assume that $\langle M, \preceq \rangle$ is a TQO. Then the *sorting problem* is defined as follows:

1. A list l of elements of M is given.
2. We want to compute a list s such that we have the following:
   (a) s is sorted ascendingly:
   $$\forall i \in \{1, \cdots, \#s - 1\}: s[i] \preceq s[i+1]$$
   Here, the length of the list s is denoted as $\#s$ and $s[i]$ is the i-th element of s.
   (b) The elements of M occur in l and s with the same frequency:
   $$\forall x \in M: \mathit{count}(x, l) = \mathit{count}(x, s).$$
   Here, the function $\mathit{count}(x, l)$ returns the number of occurrences of x in l. Therefore, we have:
   $$\mathit{count}(x, l) := \#\big\{i \in \{1, \cdots, \#l\} \mid l[i] = x\big\}.$$

Next, we present various algorithms for solving the sorting problem. We start with two algorithms that are very easy to implement: *insertion sort* and *selection sort*. However, the efficiency of these algorithms is far from optimal. Next, we present *quick sort* and *merge sort*. Both of these algorithms are very efficient when implemented carefully. However, the implementation of these algorithms is much more involved.

5.1 Insertion Sort

Let us start our investigation of sorting algorithms with *insertion sort*. We will describe the algorithm via a set of equations.

1. If the list l that has to be sorted is empty, then the result is the empty list:
   $$\texttt{sort}([]) = [].$$

2. Otherwise, the list l must have the form $[x] + r$. Here, x is the first element of l and r is the rest of l, i. e. everything of l but the first element. In order to sort l we first sort the rest r and then we insert the element x into the resulting list in a way that the resulting list remains sorted:

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

Inserting x into an already sorted list s is done according to the following specification:

1. If s is empty, the result is the list $[x]$:

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

2. Otherwise, s must have the form $[y] + r$. In order to know where to insert x we have to compare x and y.

   (a) If $x \preceq y$, then we have to insert x at the front of the list s:

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

   (b) Otherwise, x has to be inserted recursively into the list r:

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

```
1  sort := procedure(l) {
2      match (l) {
3          case []   : return [];
4          case [x|r]: return insert(x, sort(r));
5      }
6  };
7  insert := procedure(x, l) {
8      match (l) {
9          case []                 : return [x];
10         case [y|r] |   x <= y   : return [x,y] + r;
11         case [y|r] | !(x <= y) : return [y] + insert(x, r);
12     }
13 };
```

Figure 5.1: Implementing *insertion sort* in SetlX.

Figure 5.1 shows how the *insertion-sort* algorithm can be implemented in SetlX.

1. The definition of the function `sort` makes use of the `match` statement that is available in `SetlX`. Essentially, the `match` statement is an upgraded `switch` statement. Therefore, line 3 is executed if the list l is empty.

   Line 4 tests whether l can be written as

   $$l = [x] + r.$$

   Here, x is the first element of l while r contains all but the first element of l. In SetlX we write `[x|r]` in order to match a list of the form $[x] + r$.

2. The definition of the function `insert` also uses a `match` statement. However, in the last two cases the match statement also has a logical condition attached via the operator "`|`": In line 10, this condition checks whether $x \leq y$, while line 11 checks for the complementary case.

5.1.1 Complexity of Insertion Sort

We will compute the number of comparisons that are done in the implementation of `insert`. Before doing so, let us note that the function `insert.stlx` can be rewritten as shown in Figure 5.2. In comparison to Figure 5.1, we have dropped the test "`!(x <= y)`" from line 5 since it is unnecessary: If control reaches line 5, it must have skipped line 4 before and for a non-empty list that can only happen if the test "`x <= y`" fails.

```
1  insert := procedure(x, l) {
2      match (l) {
3          case []             : return [x];
4          case [y|r] | x <= y : return [x, y] + r;
5          case [y|r]          : return [y] + insert(x, r);
6      }
7  };
```

Figure 5.2: More efficient implementation of *insert*.

Let us compute the number of evaluations of the comparison operator "`<=`" in line 4 in the worst case if we call `sort`(l) with a list of length n. In order to do that, we have to compute the number of evaluations of the operator "`<=`" when `insert`(x, l) is evaluated for a list l of length n. Let us denote this number as a_n. The worst case happens if x is bigger than every element of l because in that case the test "`x <= y`" in line 4 of Figure 5.2 will always evaluate to `false` and therefore `insert` will keep calling itself recursively. Then we have

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

A trivial induction shows that this recurrence relation has the solution

$$a_n = n.$$

In the worst case the evaluation of `insert`(x, l) will lead to n comparisons for a list l of length n. The reason is simple: If x is bigger than any element of l, then we have to compare x with every element of l in order to insert it into l.

Next, let us compute the number of comparisons that have to be done when calling `sort`(l) in the worst case for a list l of length n. Let us denote this number as b_n. The worst case happens if l is sorted in reverse order, i. e. if l is sorted descendingly. Then we have

$$b_1 = 0 \quad \text{and} \quad b_{n+1} = b_n + n, \tag{1}$$

because for a list of the form $l = [x] + r$ of length $n + 1$ we first have to sort the list r recursively. As r has length n this takes b_n comparisons. After that, the call `insert`$(x, \texttt{sort}(\texttt{r}))$ inserts the element x into `sort`(r). We have previously seen that this takes n comparisons if x is bigger than all elements of `sort`(l) and if the list l is sorted descendingly this will indeed be the case.

If we substitute n by $n - 1$ in equation (1) we find

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

This recurrence equation is solved by expanding the right hand side successively as follows:

$$\begin{aligned} b_n &= b_{n-1} + (n-1) \\ &= b_{n-2} + (n-2) + (n-1) \\ &\vdots \\ &= b_{n-k} + (n-k) + \cdots + (n-1) \\ &\vdots \\ &= b_1 + 1 + \cdots + (n-1) \\ &= b_1 + \sum_{i=1}^{n-1} i \\ &= \tfrac{1}{2} \cdot n \cdot (n-1), \end{aligned}$$

because $b_1 = 0$ and the sum of all natural numbers from 1 upto $n-1$ is given as

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

This can be shown by a straightforward induction. Therefore, in the worst case the number b_n of comparisons needed for sorting a list of length n satisfies

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

Therefore, in the worst case the number of comparisons is given as $\mathcal{O}(n^2)$ and hence *insertion sort* is quadratic.

Next, let us consider the best case. The best case happens if the list l is already sorted ascendingly. Then, the call of $\texttt{insert}(x, \texttt{sort}(r))$ only needs a single comparison. This time, the recurrence equation for the number b_l of comparisons when sorting l satisfies

$$b_1 = 0 \quad \text{and} \quad b_{n+1} = b_n + 1.$$

Obviously, the solution of this recurrence equation is $b_n = n - 1$. Therefore, in the best case *insertion sort* is linear. This is as good as it can get because when sorting a list l we must at least inspect all of the elements of l and therefore we will always have at least a linear amount of work to do.

5.2 Selection Sort

Next, we discuss *selection sort*. In order to sort a given list l this algorithms works as follows:

1. If l is empty, the result is the empty list:

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

2. Otherwise, we compute the smallest element of the list l and we remove this element from l. Next, the remaining list is sorted recursively. Finally, the smallest element is added to the front of the sorted list:

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

The algorithm to delete an element x from a list l is formulated recursively. There are three cases:

1. If l is empty, we have

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

2. If x is equal to the first element of l, then the function `delete` returns the rest of l:

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

3. Otherwise, the element x is removed recursively from the rest of the list:

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

Finally, we have to specify the computation of the minimum of a list l:

1. The minimum of the empty list is bigger than any element. Therefore we have

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

2. In order to compute the minimum of the list $[x] + r$ we compute the minimum of r and then use the binary function `min`:

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

Here, the binary function `min` is defined as follows:

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

Figure 5.3 on page 52 shows an implementation of selection sort in SetlX. There was no need to implement the function `min` as this function is already predefined in SetlX. The implementation of $\texttt{delete}(x, l)$ is *defensive*: Normally, $\texttt{delete}(x, l)$ should only be called if x is indeed an element of the list l. Therefore, there must be a mistake if we try to delete an element from the empty list. The predefined `assert` function will provide us with an error message in this case.

```
1  sort := procedure(l) {
2      if (l == []) {
3          return [];
4      }
5      x := min(l);
6      return [x] + sort(delete(x,l));
7  };
8  delete := procedure(x, l) {
9      match (l) {
10         case []    : assert(false, "$x$ not in list $l$");
11         case [x|r] : return r;
12         case [y|r] : return [y] + delete(x,r);
13     }
14 };
```

Figure 5.3: Implementing *selection sort* in SetlX.

5.2.1 Complexity of Selection Sort

In order to be able to analyze the complexity of *selection sort* we have to count the number of comparisons that are performed when $\texttt{min}(l)$ is computed. We have

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

Therefore, in order to compute $\texttt{min}(l)$ for a list l of length n the binary function `min` is called $(n-1)$ times. Each of these calls of `min` causes an evaluation of the comparison operator "$\preceq$". If the number of evaluations of the comparison operator used to sort a list l of length n is written as b_n, we have

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

The reasoning is as follows: In order to sort a list of $n+1$ elements using selection sort we first have to compute the minimum of this list. We need n comparisons for this. Next, the minimum is removed from the list and the remaining list, which only contains n elements, is sorted recursively. We need b_n evaluations of the comparisons operator for this recursive invocation of `sort`.

When investigating the complexity of *insertion sort* we had arrived at the same recurrence relation. We had found the solution of this recurrence relation to be

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

It seems that the number of comparisons done by *insertion sort* is the same as the number of comparisons needed for *selection sort*. However, let us not jump to conclusions. The algorithm *insertion sort* needs $\frac{1}{2} \cdot n \cdot (n-1)$ comparisons only in the worst case while *selection sort* always uses $\frac{1}{2} \cdot n \cdot (n-1)$ comparisons. In order to compute the minimum of a list of length n we always have to do $n-1$ comparisons. However, in order to insert an element into a list of n elements, we only expect to do about $\frac{1}{2} \cdot n$ comparisons on average. The reason is that we expect about half the elements to be less than the element to be inserted. Hence, we only have to compare the element to be inserted with half of the remaining elements. Therefore, the average number of comparisons used by insertion sort is only

$$\frac{1}{4} \cdot n^2 + \mathcal{O}(n)$$

and this is half as much as the number of comparisons used by *selection sort*. Therefore, on average we expect *selection sort* to need about twice as many comparisons as *insertion sort*.

5.3 Merge Sort

Next, we discuss *merge sort*. This algorithm is the first efficient sorting algorithm that we encounter: We will see that *merge sort* only needs $\mathcal{O}\big(n \cdot \log_2(n)\big)$ comparisons to sort a list of n elements. The *merge sort* algorithm was discovered by John von Neumann in 1945. John von Neumann was one of the most prominent mathematicians of the last century. Furthermore, he published a number of influential papers in physics. Finally, he is one of the fathers of the nuclear bomb.

In order to sort a list l the algorithm proceeds as follows:

1. If l has less than two elements, then l is already sorted. Therefore we have:

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

2. Otherwise, the list l is split into two lists that have approximately the same size. These lists are then sorted recursively. Then, the sorted lists are merged in a way that the resulting list is sorted:

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

   Here, $\texttt{split}_1$ and $\texttt{split}_2$ are functions that split up the list l into two parts, while the function `merge` takes two sorted lists and combines their element in a way that the resulting list is sorted.

Figure 5.4 shows how these equations can be implemented as a SetlX program. The two functions $\texttt{split}_1$ and $\texttt{split}_2$ have been combined into one function `split` that returns a pair of lists, i. e. the expression

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

returns a pair of lists of the form

$[l_1, l_2]$.

The idea is to distribute the elements of l to the lists l_1 and l_2 in a way that both lists have approximately the same size. Let us proceed to discuss the details of the program shown in Figure 5.4:

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

Figure 5.4: The *merge sort* algorithm implemented in SetlX.

1. If the list l has less than two elements, it is already sorted and, therefore, it can be returned as it is.
2. The call to `split` distributes the elements of l to the lists `l1` and `l2`.
3. These lists are sorted recursively and the resulting sorted lists are then merged.

Next, we specify the function `split` via equations.

1. If l is empty, `split`(l) returns two empty lists:

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

2. If l contains exactly one element, this element is put into the first of the two lists returned from `split`:

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

3. Otherwise, l must have the form $[x, y] + r$. Then we split r recursively into two lists r_1 and r_2. The element x is put in front of r_1, while y is put in front

of r_2:

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

Finally, we specify how two sorted lists l_1 and l_2 are merged in a way that the resulting list is also sorted.

1. If the list l_1 is empty, the result is l_2:

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

2. If the list l_2 is empty, the result is l_1:

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

3. Otherwise, l_1 must have the form $[x] + r_1$ and l_2 has the form $[y] + r_2$. Then there is a case distinction with respect to the comparison of x and y:

   (a) $x \preceq y$.
   In this case, we merge r_1 and l_2 and put x at the beginning of this list:

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

   (b) $\neg x \preceq y$.
   Now we merge l_1 and r_2 and put y at the beginning of this list:

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

5.3.1 Complexity of Merge Sort

Next, we compute the number of comparisons that are needed to sort a list of n elements via merge sort. To this end, we first analyze the number of comparisons that are done in a call of $\texttt{merge}(l_1, l_2)$. In order to do this we define the function

$$\texttt{cmpCount} : \mathit{List}(m) \times \mathit{List}(m) \rightarrow \mathbb{N}$$

such that, given two lists l_1 and l_2 of elements of some set m the expression $\texttt{cmpCount}(l_1, l_2)$ returns the number of comparisons needed to compute $\texttt{merge}(l_1, l_2)$. Our claim is that, for any lists l_1 and l_2 we have

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

The proof is done by induction on $\#l_1 + \#l_2$.

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

Then both l_1 and l_2 are empty and therefore the evaluation of $\texttt{merge}(l_1, l_2)$ does not need any comparisons. Therefore, we have

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

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

If either l_1 or l_2 is empty, then we do not need any comparisons in order to compute $\texttt{merge}(l_1, l_2)$ and, therefore, we have

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

Next, let us assume that

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

We have to do a case distinction with respect to the relative size of x and y.

(a) $x \preceq y$. Then we have

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

Therefore, we have

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

(b) $\neg x \preceq y$. This case is similar to the previous case. □

Exercise 14: What is the form of the lists l_1 and l_2 that maximizes the value of

$$\texttt{cmpCount}(l_1, l_2)?$$

What is the value of $\texttt{cmpCount}(l_1, l_2)$ in this case? ◇

Now we are ready to compute the complexity of *merge sort* in the worst case. Let us denote the number of comparisons needed to sort a list l of length n as $f(n)$. The algorithm *merge sort* splits the list l into two lists of length $l\backslash 2$, then sorts these lists recursively, and finally merges the sorted lists. Merging two lists of length $n\backslash 2$ can be done with at most n comparisons. Therefore, the function f satisfies the recurrence relation

$$f(n) = 2 \cdot f(n\backslash 2) + \mathcal{O}(n),$$

We can use the master theorem to get an upper bound for $f(n)$. In the master theorem, we have $\alpha = 2$, $\beta = 2$, and $\delta = 1$. Therefore, $\alpha = \beta^\delta$ and the master theorem shows that we have

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

This result already shows that, for large inputs, *merge sort* is considerably more efficient than both *insertion sort* and *selection sort*. However, if we want to compare *merge sort* with *quick sort*, the result $f(n) \in \mathcal{O}\big(n \cdot \log_2(n)\big)$ is not precise enough. In order to arrive at a bound for the number of comparisons that is more precise, we need to solve the recurrence equation given above. In order to simplify things, define $a_n := f(n)$ and assume that n is a power of 2, i.e. we assume that

$$n = 2^k \qquad \text{for some } k \in \mathbb{N}.$$

Let us define $b_k := a_n = a_{2^k}$. First, we compute the initial value b_0 as follows:

$$b_0 = a_{2^0} = a_1 = 0,$$

since we do not need any comparisons when sorting a list of length one. Since merging two lists of length 2^k needs at most $2^k + 2^k = 2^{k+1}$ comparisons, b_{k+1} can be upper bounded as follows:

$$b_{k+1} = 2 \cdot b_k + 2^{k+1}$$

In order to solve this recurrence equation, we divide the equation by 2^{k+1}. This yields

$$\frac{b_{k+1}}{2^{k+1}} = \frac{b_k}{2^k} + 1$$

Next, we define

$$c_k := \frac{b_k}{2^k}.$$

Then, we get the following equation for c_k:

$$c_{k+1} = c_k + 1.$$

Since $b_0 = 0$, we also habe $c_0 = 0$. Hence, the solution of the recurrence equation

for c_k is given as

$$c_k := k.$$

Substituting this value into the defining equation for c_k we conclude that

$$b_k = 2^k \cdot k.$$

Since $n = 2^k$ implies $k = \log_2(n)$ and $a_n = b_k$, we have found that

$$a_n = n \cdot \log_2(n).$$

5.3.2 Implementing Merge Sort for Arrays

All the implementations of the SetlX programs presented up to now are quite inefficient. The reason is that, in SetlX, lists are internally represented as arrays. Therefore, when we evaluate an expression of the form

```
[ x ] + r
```

the following happens:

1. A new array is allocated. This array will later hold the resulting list.
2. The element `x` is copied to the beginning of this array.
3. The elements of the list `r` are copied to the positions following `x`.

Therefore, evaluating `[x] + r` for a list `r` of length n requires $\mathcal{O}(n)$ data movements. Hence the SetlX programs given up to now are very inefficient. In order to arrive at an implementation that is more efficient we need to make use of the fact that lists are represented as arrays. Figure 5.5 on page 58 presents an implementation of *merge sort* that treats the list l that is to be sorted as an array.

We discuss the implementation shown in Figure 5.5 line by line.

1. In line 1 the keyword "`rw`" specifies that the parameter l is a *read-write* parameter. Therefore, changes to l remain visible after `sort`(l) has returned. This is also the reason that the procedure `sort` does not return a result. Instead, the evaluation of the expression `sort`(l) has the side effect of sorting the list l.
2. The purpose of the assignment "`a := l;`" in line 2 is to create an auxiliary array `a`. This auxiliary array is needed in the procedure `mergeSort` called in line 3.
3. The procedure `mergeSort` defined in line 5 is called with 4 arguments.
   (a) The first parameter `l` is the list that is to be sorted.
   (b) However, the task of `mergeSort` is not to sort all of `l` but only the part of `l` that is given as
   ```
   l[start .. end-1].
   ```
   Hence, the parameters `start` and `end` are indices specifying the subarray that needs to be sorted.
   (c) The final parameter `a` is used as an auxiliary array. This array is needed as temporary storage and it needs to have the same size as the list `l`.
4. Line 6 deals with the case that the sublist of `l` that needs to be sorted is of length less than two. In this case, there is nothing to do as any list of this length is already sorted.

```
1  sort := procedure(rw l) {
2      a := l;
3      mergeSort(l, 1, #l + 1, a);
4  };
5  mergeSort := procedure(rw l, start, end, rw a) {
6      if (end - start < 2) { return; }
7      middle := (start + end) \ 2;
8      mergeSort(l, start,  middle, a);
9      mergeSort(l, middle, end   , a);
10     merge(l, start, middle, end, a);
11 };
12 merge := procedure(rw l, start, middle, end, rw a) {
13     for (i in [start .. end-1]) { a[i] := l[i]; }
14     idx1 := start;
15     idx2 := middle;
16     i    := start;
17     while (idx1 < middle && idx2 < end) {
18         if (a[idx1] <= a[idx2]) {
19             l[i] := a[idx1]; i += 1; idx1 += 1;
20         } else {
21             l[i] := a[idx2]; i += 1; idx2 += 1;
22         }
23     }
24     while (idx1 < middle) { l[i] := a[idx1]; i += 1; idx1 += 1; }
25     while (idx2 < end   ) { l[i] := a[idx2]; i += 1; idx2 += 1; }
26 };
```

Figure 5.5: An array based implementation of *merge sort*.

5. One advantage of interpreting the list `l` as an array is that we do no longer need to implement a function `split` that splits the list `l` into two parts. Instead, in line 7 we compute the index pointing to the middle element of the list `l` using the formula

   ```
   middle = (start + end) \ 2;
   ```

   This way, the list `l` is split into the lists

   `l[start .. middle-1]` and `l[middle .. end-1]`.

   These two lists have approximately the same size which is half the size of the list `l`.

6. Next, the lists `l[start..middle-1]` and `l[middle..end-1]` are sorted recursively in line 8 and 9, respectively.

7. The call to `merge` in line 10 merges these lists.

8. The procedure `merge` defined in line 12 has 5 parameters:

   (a) The first parameter `l` is the list that contains the two sublists that have to be merged.

   (b) The parameters `start`, `middle`, and `end` specify the sublists that have to be merged. The first sublist is

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

while the second sublist is

`l[middle .. end-1]`.

(c) The final parameter a is used as an auxiliary array. It needs to be a list of the same size as the list `l`.

9. The function `merge` assumes that the sublists `l[start..middle-1]` and `l[middle..end-1]` are already sorted. The merging of these sublists works as follows:

   (a) First, line 13 copies the sublists into the auxiliary array `a`.

   (b) In order to merge the two sublists stored in `a` into the list `l` we define three indices:

   - `idx1` points to the next element of the first sublist stored in `a`.
   - `idx2` points to the next element of the second sublist stored in `a`.
   - `i` points to the position in the list `l` where we have to put the next element.

   (c) As long as neither the first nor the second sublist stored in `a` have been exhausted we compare in line 17 the elements from these sublists and then copy the smaller of these two elements into the list `l` at position `i`. In order to remove this element from the corresponding sublist in `a` we just need to increment the corresponding index pointing to the beginning of this sublist.

   (d) If one of the two sublists gets empty while the other sublist still has elements, then we have to copy the remaining elements of the non-empty sublist into the list `l`. The `while`-loop in line 24 covers the case that the second sublist is exhausted before the first sublist, while the `while`-loop in line 25 covers the case that the first sublist is exhausted before the second sublist.

5.3.3 An Iterative Implementation of Merge Sort

```
1  sort := procedure(rw l) {
2      a := l;
3      mergeSort(l, a);
4  };
5  mergeSort := procedure(rw l, rw a) {
6      n := 1;
7      while (n < #l) {
8          k := 0;
9          while (n * (k + 1) + 1 <= #l) {
10             merge(l, n*k+1, n*(k+1)+1, min([n*(k+2),#l])+1, a);
11             k += 2;
12         }
13         n *= 2;
14     }
15 };
```

Figure 5.6: A non-recursive implementation of *merge sort*.

The implementation of *merge sort* shown in Figure 5.5 on page 58 is recursive. Unfortunately, the efficiency of a recursive implementation of *merge sort* is suboptimal. The reason is that function calls are quite costly since the arguments of the function have to be placed on a stack. As a recursive implementation has lots of function calls, it is less efficient than an iterative implementation. Therefore, we present an iterative implementation of *merge sort* in Figure 5.6 on page 59.

Instad of recursive calls of the function `mergeSort`, this implementation has two nested `while`-loops. The idea is to first split the list `l` into sublists of length 1. Obviously, these sublists are already sorted. Next, we merge these lists in pairs into lists of length 2. After that, we take pairs of lists of length 2 and merge them into sorted lists of length 4. Proceeding in this way we generate sorted lists of length 8, 16, $\cdots$. This algorithm only stops when `l` itself is sorted.

The precise working of this implementation gets obvious if we formulate the invariants of the `while`-loops. The invariant of the outer loop states that all sublists of `l` that have the form

```
l[n*k+1 ..  n*(k+1)]
```

are already sorted. It is the task of the outer while loop to build pairs of sublists of this kind and to merge them into a sublist of length $2 \cdot n$.

In the expression $l[n \cdot k + 1, \cdots, n \cdot (k + 1)]$ the variable k denotes a natural number that is used to numerate the sublists. The index k of the first sublists is 0 and therefore this sublists has the form

```
l[1 ..  n],
```

while the second sublist is given as

```
l[n+1 ..  2*n].
```

It is possible that the last sublist has a length that is less than n. This happens if the length of `l` is not a multiple of n. Therefore, the third argument of the call to `merge` in line 10 is the minimum of $n \cdot (k + 2)$ and $\#l$.

5.3.4 Further Improvements of Merge Sort

The implementation given above can still be improved in a number of ways. Tim Peters has used a number of tricks to improve the practical performance of *merge sort*. The resulting algorithm is known as *Timsort*.

The starting point of the development of *Timsort* was the observation that the input arrays given to a sorting procedure often contain subarrays that are already sorted, either ascendingly or descendingly. For this reason, *Timsort* uses the following tricks:

1. First, Timsort looks for subarrays that are already sorted. If a subarray is sorted descendingly, this subarray is reversed.

2. Sorted subarrays that are too small (i. e. have less than 32 elements) are extended to sorted subarrays to have a length that is at least 32. In order to sort these subarrays, *insertion sort* is used. The reason is that *insertion sort* is very fast for arrays that are already partially sorted. The version of *insertion sort* that is used is called *binary insertion sort* since it uses *binary search* to insert the elements into the array.

3. The algorithm to merge two sorted lists can be improved by the following observation: If we want to merge the arrays

   $$[x] + r \quad \text{and} \quad l_1 + [y] + l_2$$

and if y is less than x, then all elements of the list l_1 are also less than x. Therefore, there is no need to compare these elements with x one by one.

Timsort uses some more tricks, but unfortunately we don't have the time to discuss all of them. *Timsort* is part of the *Java* library, the source code is available online at

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

Timsort is also used on the Android platform.

5.4 *Quick Sort*

In 1961, C.A.R. Hoare published the *quick sort* algorithm [Hoa61]. The basic idea is as follows:

1. If the list l that is to be sorted is empty, we return l:

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

2. Otherwise, we have $l = [x] + r$. In this case, we split r into two lists s and b. The list s (s stands for small) contains all the elements of r that are less or equal than x, while b (b stands for big) contains those elements of r that are bigger than x. The computation of s and b is done by a function called `partition`:

   $$\texttt{partition}(x, r) = \langle s, b \rangle.$$

   Formally, the function `partition` can be defined as follows:

   (a) $\mathit{partition}(x, []) = \langle [], [] \rangle$,

   (b) $y \preceq x \ \wedge \mathit{partition}(x, r) = \langle s, b \rangle \rightarrow \mathit{partition}(x, [y] + r) = \langle [y] + s,\ b \rangle$,

   (c) $\neg (y \preceq x) \wedge \mathit{partition}(x, r) = \langle s, b \rangle \rightarrow \mathit{partition}(x, [y] + r) = \langle s,\ [y] + b \rangle$.

   After partitioning the list l into s and b, the lists s and r are sorted recursively. Then, the result is computed by appending the lists $\texttt{sort}(s)$, $[x]$, and $\texttt{sort}(b)$:

   $$\mathit{partition}(x, r) = \langle s, b \rangle \rightarrow \mathit{sort}([x] + r) = \mathit{sort}(s) + [x] + \mathit{sort}(b).$$

Figure 5.7 on page 62 shows how these equations can be implemented in SetlX.

5.4.1 Complexity

Next, we investigate the computational complexity of *quick sort*. Our goal is to compute the number of comparisons that are needed when $\texttt{sort}(l)$ is computed for a list l of length n. In order to compute this number we first investigate how many comparisons are needed for evaluating

$$\texttt{partition}(p, l)$$

for a list l of n elements: Each of the n elements of l has to be compared with x. Therefore, we need n comparisons to compute $\texttt{partition}(p, l)$. The number of comparisons for evaluating $\texttt{sort}(l)$ depends on the result of $\texttt{partition}(p, l)$. There is a best case and a worst case. We investigate the worst case first.

```
1  sort := procedure(l) {
2      match (l) {
3          case []   : return [];
4          case [x|r]: [s,b] := partition(x, r);
5                      return sort(s) + [x] + sort(b);
6      }
7  };
8  partition := procedure(p, l) {
9      match (l) {
10         case []   : return [ [], [] ];
11         case [x|r]: [ r1, r2 ] := partition(p, r);
12                     if (x <= p) {
13                         return [ [x] + r1, r2 ];
14                     } else {
15                         return [ r1, [x] + r2 ];
16                     }
17     }
18 };
```

Figure 5.7: The *quick sort* algorithm.

Worst Case Complexity

Let us denote the number of comparisons needed to evaluate `sort`(l) for a list l of length n in the worst case as a_n. The worst case occurs if the call to `partition` returns a pair of the form

$$[[], b].$$

This happens if all elements of $l[2..]$ are bigger than $l[1]$. Then, we have

$$a_n = a_{n-1} + n - 1.$$

The term $n-1$ is due to the $n-1$ comparisons needed for execution `partition`(x, r) in line 4 of 5.7 and the term a_{n-1} is the number of comparisons needed for the recursive evaluation of $sort(b)$.

The initial condition is $a_1 = 0$, since we do not need any comparisons to sort a list containing only one element. Hence the recurrence relation can be solved as follows:

$$\begin{aligned}
a_n &= a_{n-1} + (n-1) \\
&= a_{n-2} + (n-2) + (n-1) \\
&= a_{n-3} + (n-3) + (n-2) + (n-1) \\
&= \vdots \\
&= a_1 + 1 + 2 + \cdots + (n-2) + (n-1) \\
&= 0 + 1 + 2 + \cdots + (n-2) + (n-1) \\
&= \sum_{i=0}^{n-1} i = \tfrac{1}{2} \cdot n \cdot (n-1) = \tfrac{1}{2} \cdot n^2 - \tfrac{1}{2} \cdot n \\
&\in \mathcal{O}(n^2)
\end{aligned}$$

This shows that in the worst case, the number of comparisons is as big as it is in the worst case of *insertion sort*. The worst case occurs if we try to sort a list l that is already sorted.

Average Complexity

By this time you probably wonder why the algorithm has been called *quick sort* since, in the worst case, it much slower as *merge sort*. The reason is that the average number d_n of comparisons needed to sort a list of n elements is $\mathcal{O}\big(n \cdot \log_2(n)\big)$. We prove this claim next. Let us first note the following: If l is a list of $n+1$ elements, then the number of elements of the list s that is computed in line 4 of Figure 5.7 is a member of the set $\{0, 1, 2, \cdots, n\}$. If the length of s is i and the length of l is $n+1$, then the length of b is $n-i$. Therefore, if $\#s = i$, then on average we need

$$d_i + d_{n-i}$$

comparisons to sort s and b recursively. If we take the average over all possible values of $i = \#s$ then, since $i \in \{0, 1, \cdots, n\}$, we get the following recurrence relation for d_{n+1}:

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

Here, the term n accounts for the number of comparisons needed to compute

`partition(x, r)`.

In order to simplify the recurrence relation (1) we note that

$$\begin{aligned} \sum_{i=0}^{n} f(n-i) &= f(n) + f(n-1) + \cdots + f(1) + f(0) \\ &= f(0) + f(1) + \cdots + f(n-1) + f(n) \\ &= \sum_{i=0}^{n} f(i) \end{aligned}$$

holds for any function $f : \mathbb{N} \to \mathbb{N}$. This observation can be used to simplify the recurrence relation (1) as follows:

$$d_{n+1} = n + \frac{2}{n+1} \cdot \sum_{i=0}^{n} d_i. \tag{2}$$

In order to solve this recurrence relation we substitute $n \mapsto n+1$ and arrive at

$$d_{n+2} = n + 1 + \frac{2}{n+2} \cdot \sum_{i=0}^{n+1} d_i. \tag{3}$$

Next, we multiply equation (3) with $n+2$ and equation (2) with $n+1$. This yields the equations

$$(n+2) \cdot d_{n+2} = (n+2) \cdot (n+1) + 2 \cdot \sum_{i=0}^{n+1} d_i, \tag{4}$$

$$(n+1) \cdot d_{n+1} = (n+1) \cdot n + 2 \cdot \sum_{i=0}^{n} d_i. \tag{5}$$

We take the difference of equation (4) and (5) and note that the summations cancel except for the term $2 \cdot d_{n+1}$. This leads to

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

This equation can be simplified as

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

In order to exhibit the true structure of this equation we divide by $(n+2) \cdot (n+3)$

and get

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} + \frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}. \tag{6}$$

We proceed by computing the *partial fraction decomposition* of the fraction

$$\frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}.$$

In order to so, we use the *ansatz*

$$\frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)} = \frac{\alpha}{n+2} + \frac{\beta}{n+3}.$$

Multiplying this equation with $(n+2) \cdot (n+3)$ yields

$$2 \cdot n + 2 = \alpha \cdot (n+3) + \beta \cdot (n+2).$$

This can be simplified as follows:

$$2 \cdot n + 2 = (\alpha + \beta) \cdot n + 3 \cdot \alpha + 2 \cdot \beta.$$

Comparing the coefficients produces two equations:

$$\begin{aligned} 2 &= \alpha + \beta \\ 2 &= 3 \cdot \alpha + 2 \cdot \beta \end{aligned}$$

If we subtract the second equation twice from the first equation we arrive at $\alpha = -2$. Substituting this into the first equation gives $\beta = 4$. Hence, the equation (6) can be written as

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} - \frac{2}{n+2} + \frac{4}{n+3}.$$

In order to simplify this equation, let us define

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

Then the last equation is simplified to

$$a_{n+2} = a_{n+1} - \frac{2}{n+2} + \frac{4}{n+3}.$$

Substituting $n \mapsto n-2$ simplifies this equation:

$$a_n = a_{n-1} - \frac{2}{n} + \frac{4}{n+1},$$

This equation can be rewritten as a sum. Since $a_0 = \frac{d_0}{1} = 0$ we have

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

Let us simplify this sum:

$$\begin{aligned} a_n &= 4 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\ &= 4 \cdot \sum_{i=2}^{n+1} \frac{1}{i} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\ &= 4 \cdot \frac{1}{n+1} - 4 \cdot \frac{1}{1} + 4 \cdot \sum_{i=1}^{n} \frac{1}{i} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\ &= 4 \cdot \frac{1}{n+1} - 4 \cdot \frac{1}{1} + 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\ &= -\frac{4 \cdot n}{n+1} + 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \end{aligned}$$

In order to finalize our computation we have to compute an approximation for the sum

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$

The number H_n is known in mathematics as the n-th *harmonic number*. Leonhard Euler (1707 – 1783) was able to prove that the harmonic numbers can be approximated as

$$H_n = \ln(n) + \gamma + \mathcal{O}\Big(\frac{1}{n}\Big).$$

Here, γ is the Euler-Mascheroni constant and has the value

$$\gamma = 0.5772156649\cdots.$$

Therefore, we have found the following approximation for a_n:

$$a_n = -\frac{4 \cdot n}{n+1} + 2 \cdot \ln(n) + \mathcal{O}(1).$$

Since we have $d_n = (n+1) \cdot a_n$ we can conclude that

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

holds. Let us compare this result with the number of comparisons needed for *merge sort*. We have seen previously that *merge sort* needs

$$n \cdot \log_2(n) + \mathcal{O}(n)$$

comparisons in order to sort a list of n elements. Since we have $\ln(n) = \ln(2) \cdot \log_2(n)$ we conclude that the average case of *quick sort* needs

$$2 \cdot \ln(2) \cdot n \cdot \log_2(n)$$

comparisons and hence *quick sort* needs $2 \cdot \ln(2) \approx 1.39$ times as many comparisons as *merge sort*.

5.4.2 Implementing *Quick Sort* for Arrays

Finally, we show how *quick sort* is implemented for arrays. Figure 5.8 on page 66 shows this implementation.

1. Contrary to the array based implementation of *merge sort*, we do not need an

```
1  sort := procedure(rw l) {
2      quickSort(1, #l, l);
3  };
4  quickSort := procedure(a, b, rw l) {
5      if (b <= a) {
6          return; // at most one element, nothing to do
7      }
8      m := partition(a, b, l);  // split index
9      quickSort(a, m - 1, l);
10     quickSort(m + 1, b, l);
11 };
12 partition := procedure(start, end, rw l) {
13     pivot := l[end];
14     left  := start - 1;
15     for (idx in [start .. end-1]) {
16         if (l[idx] <= pivot) {
17             left += 1;
18             swap(left, idx, l);
19         }
20     }
21     swap(left + 1, end, l);
22     return left + 1;
23 };
24 swap := procedure(x, y, rw l) {
25     [ l[x], l[y] ] := [ l[y], l[x] ];
26 };
```

Figure 5.8: An implementation of *quick sort* based on arrays.

auxiliary array. This is one of the main advantages of *quick sort* over *merge sort*.

2. The function `sort` is reduced to a call of `quickSort`. This function takes the parameters `a`, `b`, and `l`.
   (a) `a` specifies the index of the first element of the subarray that needs to be sorted.
   (b) `b` specifies the index of the last element of the subarray that needs to be sorted.
   (c) `l` is the array that needs to be sorted.

   Calling `quickSort(a, b, l)` sorts the subarray

   $$\texttt{l[a]}, \texttt{l[a+1]}, \cdots, \texttt{l[b]}$$

   of the array `l`, i. e. after that call we expect to have

   $$\texttt{l[a]} \preceq \texttt{l[a+1]} \preceq \cdots \preceq \texttt{l[b]}.$$

   The implementation of the function `quickSort` is quite similar to the list implementation. The main difference is that the function `partition` that is called in line 8 redistributes the elements of `l` such that afterwards all elements that are less or equal than the *pivot element* `l[m]` have an index that is lower than the index `m`, while the remaining elements will have an index that is bigger than `m`. The pivot element itself will have the index `m`.

3. The difficult part of the implementation of *quick sort* is the implementation of the function `partition` that is shown beginning in line 12. The `for` loop in line 15 satisfies the following invariants.

   (a) $\forall i \in \{\texttt{start}, \cdots, \texttt{left}\} : \texttt{l}[i] \leq \texttt{pivot}$.
   All elements in the subarray `l[start..left]` are less or equal than the pivot element.

   (b) $\forall i \in \{\texttt{left} + 1, \cdots, \texttt{idx} - 1\} : \texttt{pivot} < l[i]$.
   All elements in the subarray `l[left+1..idx-1]` are greater than the pivot element.

   (c) $\texttt{pivot} = \texttt{l}[\texttt{end}]$
   The pivot element itself is at the end of the array.

   Observe how the invariants (a) and (b) are maintained:

   (a) Initially, the invariants are true because the corresponding sets are empty. At the start of the `for`-loop we have
   $$\{\texttt{start}, \cdots, \texttt{left}\} = \{\texttt{start}, \cdots, \texttt{start} - 1\} = \{\}$$
   and
   $$\{\texttt{left} + 1, \cdots, \texttt{idx} - 1\} = \{\texttt{start}, \cdots, \texttt{start} - 1\} = \{\}.$$

   (b) If the element `l[idx]` is less than the pivot element, it need to become part of the subarray `l[start..left]`. In order to achieve this, it is placed at the position `l[left+1]`. The element that has been at that position is part of the subarray `l[left+1..idx-1]` and therefore, most of the times,[1] it is greater than the pivot element. Hence we move this element to the end of the subarray `l[left+1..idx-1]`.

   Once the `for` loop in line 15 terminates, the call to `swap` in line 21 moves the pivot element into its correct position.

5.4.3 Improvements for *Quick Sort*

There are a number of tricks that can be used to increase the efficiency of *quick sort*.

1. Instead of taking the first element as the pivot element, use three elements from the list l that is to be sorted. For example, take the first element, the last element, and an element from the middle of the list. Now compare these three elements and take that element as a pivot that is between the other two elements.

   The advantage of this strategy is that worst case performance is much more unlikely to occur. In particular, using this strategy the worst case won't occur for a list that is already sorted.

2. If a sublist contains fewer than 10 elements, use *insertion sort* to sort this sublist.

   The paper "*Engineering a Sort Function*" by Jon L. Bentley and M. Douglas McIlroy [BM93] describes the previous two improvements.

3. In order to be sure that the average case analysis of *quick sort* holds we can randomly *shuffle* the list l that is to be sorted. This approach is advocated

[1] It is not always greater than the pivot element because the subarray `l[left+1..idx-1]` might well be empty.

by Sedgewick [SW11b]. In SetlX this is quite easy as there is a predefined function `shuffle` that takes a list and shuffles it randomly. For example, the expression

```
shuffle([1..10]);
```

might return the result

```
[1, 9, 8, 5, 2, 10, 6, 3, 4, 7].
```

4. In 2009, Vladimir Yaroslavskiy introduced *dual pivot quick sort* [Yar09]. His paper can be downloaded at the following address:

   http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf

   The main idea of Yaroslavskiy is to use two pivot elements x and y. For example, we can define

   $$x := l[1] \quad \text{and} \quad y := l[\#l],$$

   i. e. we take x as the first element of l, while y is the last element of l. Then, the list l is split into three parts:

   (a) The first part contains those elements that are less than x.

   (b) The second part contains those elements that are bigger or equal than x but less or equal than y.

   (c) The third part contains those elements that are bigger than y.

   Figure 5.9 on page 69 shows a simple list based implementation of *dual pivot quick sort*.

   Various studies have shown that *dual pivot quick sort* is faster than any other sorting algorithm. For this reason, the version 1.7 of *Java* uses *dual pivot quick sort*:

   http://www.docjar.com/html/api/java/util/DualPivotQuicksort.java.html

Exercise 15: Implement a version of *dual pivot quick sort* that uses arrays instead of lists.

5.5 A Lower Bound for the Number of Comparisons Needed to Sort a List

In this section we will show that any sorting algorithm that sorts elements by comparing them must use at least

$$\mathcal{O}\big(n \cdot \ln(n)\big)$$

comparisons. The important caveat here is that the sorting algorithm is restricted to not make any assumptions on the elements of the list l that is to be sorted. The only operation that is allowed on these elements is the use of the comparison operator "<". Furthermore, to simplify matters let us assume that all elements of the list l are distinct.

Let us consider lists of two elements first, i. e. assume we have

$$l = [a_1, a_2].$$

In order to sort this list, one comparison is sufficient:

1. If $a_1 < a_2$ then $[a_1, a_2]$ is sorted ascendingly.

```
1  sort := procedure(l) {
2      match (l) {
3      case []     : return [];
4      case [x]    : return [x];
5      case [x,y|r]: [p1, p2] := [min({x,y}), max({x,y})];
6                    [l1,l2,l3] := partition(p1, p2, r);
7                    return sort(l1) + [p1] + sort(l2) + [p2] + sort(l3);
8      }
9  };
10 partition := procedure(p1, p2, l) {
11     match (l) {
12     case []   : return [ [], [], [] ];
13     case [x|r]: [ r1, r2, r3 ] := partition(p1, p2, r);
14                 if (x < p1) {
15                     return [ [x] + r1, r2, r3 ];
16                 } else if (x <= p2) {
17                     return [ r1, [x] + r2, r3 ];
18                 } else {
19                     return [ r1, r2, [x] + r3 ];
20                 }
21     }
22 };
```

Figure 5.9: A list based implementation of *dual pivot quick sort*.

2. If $a_2 < a_1$ then $[a_2, a_1]$ is sorted ascendingly.

If the list l that is to be sorted has the form

$$l = [a_1, a_2, a_3]$$

then there are 6 possibilities to arrange these elements:

$$[a_1, a_2, a_3], \quad [a_1, a_3, a_2], \quad [a_2, a_1, a_3], \quad [a_2, a_3, a_1], \quad [a_3, a_1, a_2], \quad [a_3, a_2, a_1].$$

Now we need at least three comparisons, since with two comparisons we could at most choose between four different possibilities. In general there are

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \prod_{i=1}^{n} i$$

different permutations of a list of n different elements. We prove this claim by induction.

1. $n = 1$:

   There is only 1 way to arrange one element in a list. As $1 = 1!$ the claim ist proven in this case.

2. $n \mapsto n + 1$:

   If we have $n+1$ different elements and want to arrange these elements in a list, then there are $n+1$ possibilities for the first element. In each of these cases the induction hypotheses tells us that there are $n!$ ways to arrange the remaining n elements in a list. Therefore, all in all there are $(n + 1) \cdot n! = (n + 1)!$ different arrangements of $n + 1$ elements in a list.

Next, we consider how many different cases can be distinguished if we have k different tests that only give yes or no answers. Tests of this kind are called *binary*

tests.

1. If we restrict ourselves to binary tests, then we can only distinguish between two cases.
2. If we have 2 tests, then we can distinguish between 2^2 different cases.
3. In general, k tests can choose from at most 2^k different cases.

The last claim can be argued as follows: If the results of the tests are represented as 0 and 1, then k binary tests correspond to a binary string of length k. However, binary strings of length k can be used to code the numbers from 0 up to $2^k - 1$. We have

$$card\big(\{0, 1, 2, \cdots, 2^k - 1\}\big) = 2^k.$$

Hence there are 2^k binary strings of length k.

If we have a list of n different elements then there are $n!$ different permutations of these elements. In order to figure out which of these $n!$ different permutations is given we have to perform k comparisons where we must have

$$2^k \geq n!.$$

This immediately implies

$$k \geq \log_2(n!).$$

In order to proceed we need an approximation for the expression $\log_2(n!)$. A simple approximation of this term is

$$\log_2(n!) = n \cdot \log_2(n) + \mathcal{O}(n).$$

Using this approximation we get

$$k \geq n \cdot \log_2(n) + \mathcal{O}(n).$$

As *merge sort* is able to sort a list of length n using only $n \cdot \log_2(n)$ comparisons we have shown that this algorithm is optimal with respect to the number of comparisons.

Chapter 6

Abstract Data Types

In the same way as the notion of an algorithm abstracts from the details of a concrete implementation of this algorithm, the notion of an *abstract data type* abstracts from concrete data structures. The notion enables us to separate algorithms from the data structures used in these algorithms. The next section gives a formal definition of "*abstract data types*". As an example, we introduce the abstract data type of *stacks*. The second section shows how abstract data types are supported in SetlX. Finally, we show how stacks can be used to evaluate arithmetic expressions.

6.1 A Formal Definition of Abstract Data Types

Formally, an *abstract data type* $\mathcal{D}$ is defined as a 5-tupel of the form

$$\mathcal{D} = \langle N, P, Fs, Ts, Ax \rangle.$$

where the meaning of the components is as follows:

1. N is the *name* of the abstract data type.
2. P is the set of *type parameters*. Here, a type parameter is just a string. This string is interpreted as a type variable. The idea is that we can later substitute a concrete data type for this string.
3. Fs is the set of *function symbols*. These function symbols denote the operations that are supported by this abstract data type.
4. Ts is a set of *type specifications*. For every function symbol $f \in Fs$ the set Ts contains a *type specifications* of the form

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

   Here, T_1, $\cdots$, T_n and S are names of data types. There are three cases for these data types:

   (a) We can have concrete data types like, e. g. "`int`" or "`String`".

   (b) Furthermore, these can be the names of abstract data types.

   (c) Finally, T_1, $\cdots$, T_n and S can be type parameters from the set P.

   The type specification $f : T_1 \times \cdots \times T_n \to S$ expresses the fact that the function f has to be called as

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

where for all $i \in \{1, \cdots, n\}$ the argument t_i has type T_i. Furthermore, the result of the function f is of type S.

Additionally, we must have either $T_1 = T$ or $S = T$. Therefore, either the first argument of f has to be of type T or the result of f has to be of type T. If we have $T_1 \neq T$ and, therefore, $S = T$, then f is called a *constructor* of the data type T. Otherwise, f is called a *method*.

5. Ax is a set of mathematical formulæ. These formulæ specify the behaviour of the abstract data type and are therefore called the *axioms* of $\mathcal{D}$.

The notion of an abstract data type is often abbreviated as ADT.

Next, we provide a simple example of an abstract data type, the *stack*. Informally, a stack can be viewed as a pile of objects that are put on top of each other, so that only the element on top of the pile is accessible. An ostensive example of a stack is a pile of plates that can be found in a canteen. Usually, the clean plates are placed on top of each other and only the plate on top is accessible. Formally, we define the data type *Stack* as follows:

1. The name of the data type is *Stack*.
2. The set of type parameters is $\{Element\}$.
3. The set of function symbols is
$$\{Stack, push, pop, top, isEmpty\}.$$
4. The type specifications of these function symbols are given as follows:
   (a) $stack : Stack$
   The function *stack* takes no arguments and produces an empty stack. Therefore, this function is a constructor. Intuitively, the function call $stack()$ creates an empty stack.
   (b) $push : Stack \times Element \to Stack$
   The function call $push(S, x)$ puts the element x on top of the stack S. In the following, we will use an object oriented notation and write $S.push(x)$, instead of $push(S, x)$.
   (c) $pop : Stack \to Stack$
   The function call $S.pop()$ removes the first element from the stack S.
   (d) $top : Stack \to Element$
   The function call $S.top()$ returns the element that is on top of the stack S.
   (e) $isEmpty : Stack \to \mathbb{B}$
   The function call $S.isEmpty()$ checks whether the stack S is empty.

The intuition that we have of a stack is captured by the following axioms.

1. $stack().top() = \Omega$
   Here, Ω denotes the undefined value[1]. The expression $stack()$ creates an empty stack. Therefore, the given axiom expresses the fact that there is no element on top of the empty stack.

[1] Some philosophers are concerned that it is not possible to define an undefined value. They argue that if an undefined value could be defined, it would be no longer undefined and hence it can not be defined. However, that is precisely the point of the undefined value: It is not defined.

2. $S.push(x).top() = x$

   If we have a stack S and then push an element x on top of S, then the element on top of the resulting stack is, obviously, x.

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

   Trying to remove an element from the empty stack yields an undefined result.

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

   If we have a stack S, then push an element x of top of S, and finally remove the element on top of the resulting stack, then we are back at our stack S.

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

   This axiom expresses the fact that the stack created by the function call $stack()$ is empty.

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

   If we push an element x on top of a stack S, then the resulting stack cannot be empty.

When contemplating the axioms given above we can recognize some structure. If we denote the functions `stack` and `push` as *generators*, then the axioms specify the behavior of the remaining functions on the stacks created by the generators.

The data type of a stack has many applications in computer science. To give just one example, the implementation of the *Java virtual machine* is based on a stack. Furthermore, we will later see how, using three stacks, arithmetic expressions can be evaluated.

6.2 Implementing Abstract Data Types in SetlX

In an object oriented programming language, abstract data types are conveniently implemented via a class. In a typed object oriented programming language like *Java*, the usual way to proceed is to create an interface describing the signatures of the abstract data type and then to implement the abstract data type as a class. Instead of an interface, we can also use an abstract class to describe the signature. In an untyped language like SetlX there is no way to neatly capture the signatures. Therefore, the implementation of an abstract data type in SetlX merely consists of a class. At this point we note that classes are discussed in depth in chapter 7 of the SetlX tutorial. If the reader hasn't encountered classes in SetlX, she is advised to consult this chapter before reading any further.

Figure 6.1 shows am implementation of the ADT *Stack* that is discussed next.

1. The definition of the Adt *Stack* starts with the keyword `class` in line 1. After the keyword `class`, the name of the class has to be given. In Figure 6.1 this name is `stack`.

   In SetlX, every class also defines a constructor wich has the same name as the class. Since in line 1 the name `class` is followed by "`()`", the constructor `stack` does not take any arguments. In order to use it, we can write

   ```
   s := stack();
   ```

   This assignment creates an empty stack and assigns this stack to the variable `s`.

```
1  class stack() {
2      mStackElements := [];
3
4    static {
5      push := procedure(e) {
6          this.mStackElements += [e];
7      };
8      pop := procedure() {
9          assert(#mStackElements > 0, "popping empty stack");
10         this.mStackElements := mStackElements[1 .. #mStackElements - 1];
11     };
12     top := procedure() {
13         assert(#mStackElements > 0, "top of empty stack");
14         return mStackElements[#mStackElements];
15     };
16     isEmpty := procedure() {
17         return mStackElements == [];
18     };
19     f_str := procedure() {
20          copy   := this;
21          result := convert(copy);
22          dashes := "\n";
23          for (i in {1 .. #result}) {
24               dashes += "-";
25          }
26          return dashes + "\n" + result + dashes + "\n";
27     };
28     convert := procedure(s) {
29         if (s.isEmpty()) {
30             return "|";
31         }
32         top := s.top();
33         s.pop();
34         return convert(s) + " " + top + " |";
35     };
36   }
37 }
38
39 createStack := procedure(l) {
40     result := stack();
41     n := #l;
42     for (i in [n, n-1 .. 1]) {
43         result.push(l[i]);
44     }
45     return result;
46 };
```

Figure 6.1: An array based implementation of the ADT *Stack* in SetlX.

2. Line 2 defines the first (and in this case only) member variable of the class `stack`. Therefore, every object o of class stack will have a *member variable* called `mStackElements`. We will use this list to store the elements of the stack.

To retrieve this member variable from the object o we can use the following expression:

o.mStackElements

The implementation of stacks shown in Figure 6.1 is based on storing the elements of the stack in a SetlX list. In SetlX, lists are internally implemented as arrays. However, this is not the only way to implement a stack: A stack can also be implemented as a linked list.

One word on naming convention. It is my convention to start all member variables of a class with the lower case letter "m" followed by a descriptive name.

3. The rest of the definition of the class `stack` is enclosed in one big static block that starts with the keyword `static` in line 4 and ends with the closing brace "}" in line 36. The static block is not part of the constructor. Therefore, the only thing that the constructor class `stack()` does, is to initialize the member variable `mStackElements`.

   The static block itself contains a number of procedure definitions. These procedures are called *methods*. As these methods are defined inside the static block, they are not considered to be defined in the constructor and, therefore, are not member variables but are, instead, class variables. Hence, they are available in the class `stack`. For example,

   ```
   stack.push
   ```

   refers to the method `push` defined in line 5 to 7. Of course, every object of class `stack` will also have access to these methods. For example, if s is an object of class `stack`, then we can invoke the method `push` by writing:

   s.push(x)

   The reason the methods in class `stack` are all inside the static block is the fact that these methods work the same way for all stacks. The only case where it is not a good idea to declare a method as static is when this method needs to be defined on a per-object-basis: If the code of the method differs for every object so that it is necessary to construct the method for every object differently, then the definition of the method has to be part of the constructor and not part of the static block.

4. Line 5 starts the definition of the method `push`. This method is called with one argument e, where e is the element that is to be pushed on the stack. In the array based implementation, this is achieved by appending e to the list `mStackElements`.

   If you are new to object-oriented programming, then at this point you might wonder why we do not need to specify the stack onto which the element e is pushed. However, we do have to specify the stack by prefixing it to the method invocation. That is, if s is a stack and we want to push e onto this stack, then we can do this by writing the following:

   s.push(e)

   There is another subtle point that needs to be discussed: When referring to the member variable `mStackElements` we had to prefix this variable with the string "`this.`". In the case that we just want to read a variable, it is not necessary to prefix the variable with "`this.`". However, once we want to change a member variable, the prefix "`this.`" is mandatory. If we just had written

   ```
   mStackElements += [e];
   ```

then we would have created a new local variable with the name `mStackElements` and we would have changed this local variable rather than the member variable `mStackElements`.

5. Line 8 starts the implementation of the method `pop`, which has the task to remove one element from the stack. Of course, it would not make sense to remove an element from the stack if the stack is empty. Therefore, the `assert` statement in line 9 checks whether the number of elements of the list `mStackElements` is bigger than 0. If this condition is satisfied, the last element of the list `mStackElements` is removed.

6. Line 12 starts the definition of the method `top`. First, it is checked that the stack is non-empty. Then, the element at the end of the list `mStackElements` is returned.

7. Line 16 defines the method `isEmpty`. This method checks whether the list `mStackElements` is empty.

8. Line 19 defines the method `f_str`. This method serves a similar purpose as the method `toString` in a *Java* program: If an object of class `stack` needs to be converted into a string, then the method `f_str` is invoked automatically to perform this conversion.

   In order to understand the implementation of `f_str` we look at an example and type

   ```
   s := stack(); s.push(1); s.push(2); s.push(3); print(s);
   ```

   at the prompt of the interpreter. This commands create an empty stack and push the numbers 1, 2, and 3 onto this stack. Finally, the resulting stack is printed. The string that is then printed is the result of calling `f_str` and has the following form:

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

   Hence, the topmost element of the stack is printed last.

   The implementation of the method `f_str` works as follows.

   (a) First, line 20 creates a copy of the stack. The reason for working with a copy is that all the methods that are available on a stack only permit us to look at the topmost element of the stack. In order to inspect the second element we first have to remove the first element. However, the method `f_str` is not supposed to change the stack. Therefore, we have to create a copy of the stack first.

   (b) Next, we use the auxiliary method `convert`. This method computes a string of the form

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

   The implementation of `convert` is done via a case distinction: If the given stack s is empty, the result of `convert` will be the string "`|`". Otherwise we get the top element of the stack using the method `top()` and remove it using `pop()`. Next, the remaining stack is converted to a string in line 34 and finally the element `top` is appended to this string.

   (c) The method `f_str` uses a `for` loop to create a line of dashes. The result of `convert` is then decorated with these dashes.

9. Note that there is no "`;`" at the end of the class definition in line 37. In contrast to a procedure definition, a class definition must not be terminated by the character "`;`".

You should note that we were able to implement the method `f_str` without knowing anything about the internal representation of the stack. In order to implement `f_str` we only used the methods `top`, `pop`, and `isEmpty`. This is one of the main advantages of an abstract data type: An abstract data type abstracts from the concrete data structures that implement it. If an abstract data type is done right, it can be used without knowing how the data that are administered by the abstract data type are actually represented.

6.3 Evaluation of Arithmetic Expressions

Next, in order to demonstrate the usefulness of stacks, we show how *arithmetic expressions* can be evaluated using stacks. An *arithmetic expression* is a string that is made up of numbers and the operator symbols "`+`", "`-`", "`*`", "`/`", "`%`", and "`**`". Here x `%` y denotes the remainder when doing integer division and x `**` y denotes the power x^y. Furthermore, arithmetic expressions can use the parentheses "`(`" and "`)`".

The set of arithmetic expressions can be defined inductively as follows:

1. Every number $n \in \mathbb{N}$ is an arithmetic expression.
2. If s and t are arithmetic expressions, then

   $$s + t, \quad s - t, \quad s * t, \quad s \,/\, t, \quad s \,\%\, t, \quad \text{and} \quad s \,**\, t$$

   are arithmetic expressions.
3. If s is an arithmetic expression, then (s) is an arithmetic expression.

If we have been given a string that is an arithmetic expression, then in order to evaluate this arithmetic expression we need to know the precedence and the associativity of the operators. In mathematics the operators "`*`", "`/`" and "`%`" have a higher precedence than the operators "`+`" and "`-`". Furthermore, the operator "`**`" has a precedence that is higher than the precedence of any other operators. The operators "`+`", "`-`", "`*`", "`/`", and "`%`" associate to the left: An expression of the form

`1 - 2 - 3` is interpreted as `(1 - 2) - 3`.

Finally, the operator "`**`" associates to the right: The arithmetic expression

`2 ** 3 ** 2` is interpreted as `2 ** (3 ** 2)`.

Our goal is to implement a program that evaluates an arithmetic expression.

6.3.1 A First Example

In order to explain the algorithm that is used to evaluate an arithmetic expression, we present a simple example first. Consider the arithmetic expression

`1 + 2 * 3 - 4`.

This string is processed from left to right, one *token*[2] at a time. In order to process the string we use three stacks.

[2] A *token* is either a number, an operator symbol, or a parenthesis.

1. The *token stack* contains all tokens of the arithmetic expression. The first token of the arithmetic expression is on top of this stack.
2. The *argument stack* contains numbers.
3. The *operator stack* contains operator symbols and the left parenthesis "`(`".

The evaluation of `1 + 2 * 3 - 4` proceeds as follows:

1. In the beginning, the token stack contains the arithmetic expression and the other two stacks are empty:

   ```
   mTokens    = [ 4, "-", 3, "*", 2, "+", 1 ],
   ```

   Note that the number that is at the beginning of the arithmetic expression is on top of the stack.

   ```
   mArguments = [],
   mOperators = [].
   ```

2. The number `1` is removed from the token stack and is put onto the argument stack instead. The three stacks are now as follows:

   ```
   mTokens    = [ 4, "-", 3, "*", 2, "+" ],
   mArguments = [ 1 ],
   mOperators = [].
   ```

3. Next, the operator `"+"` is removed from the token stack and is put onto the operator stack. Then we have:

   ```
   mTokens    = [ 4, "-", 3, "*", 2 ],
   mArguments = [ 1 ]
   mOperators = [ "+" ].
   ```

4. Now, we remove the number `2` from the token stack and put it onto the argument stack. We have:

   ```
   mTokens    = [ 4, "-", 3, "*" ],
   mArguments = [ 1, 2 ],
   mOperators = [ "+" ].
   ```

5. We remove the operator `"*"` from the token stack and compare the precedence of the operator with the precedence of the operator `"+"`, which is on top of the operator stack. Since the precedence of the operator `"*"` is greater than the precedence of the operator `"+"`, the operator `"*"` is put onto the operator stack. The reason is that we have to evaluate this operator before we can evaluate the operator `"+"`. Then we have:

   ```
   mTokens    = [ 4, "-", 3 ],
   mArguments = [ 1, 2 ],
   mOperators = [ "+", "*"].
   ```

6. We remove the number `3` from the token stack and put it onto the argument stack.

   ```
   mTokens    = [ 4, "-" ],
   mArguments = [ 1, 2, 3 ],
   ```

```
mOperators = [ "+", "*" ].
```

7. We remove the operator `"-"` from the token stack and compare this operator with the operator `"*"`, which is on top of the operator stack. As the precedence of the operator `"*"` is higher as the precedence of the operator `"-"`, we have to evaluate the operator `"*"`. In order to do so, we remove the arguments 3 and 2 from the argument stack, remove the operator `"*"` from the operator stack and compute the product of the two arguments. This product is then put back on the argument stack. The operator `"-"` is put back on the token stack since it has not been used. Hence, the stacks look as shown below:

   ```
   mTokens    = [ 4, "-" ],
   mArguments = [ 1, 6 ],
   mOperators = [ "+" ].
   ```

8. Again, we take the operator `"-"` from the token stack and compare it with the operator `"+"` that is now on top of the operator stack. Since both operators have the same precedence, the operator `"+"` is evaluated: We remove two arguments from the argument stack, remove the operator `"+"` from the operator stack and compute the sum of the arguments. The result is put back on the argument stack. Furthermore, the operator `"-"` is put back on the token stack. Then we have:

   ```
   mTokens    = [ 4, "-" ],
   mArguments = [ 7 ],
   mOperators = [].
   ```

9. Next, the operator `"-"` is removed from the token stack and is now put on the operator stack. We have:

   ```
   mTokens    = [ 4 ],
   mArguments = [ 7 ],
   mOperators = [ "-" ].
   ```

10. The number `4` is removed from the token stack and put onto the argument stack. We have:

    ```
    mTokens    = [],
    mArguments = [ 7, 4 ],
    mOperators = [ "-" ].
    ```

11. Now the input has been consumed completely. Hence, the operator `"-"` is removed from the operator stack and furthermore, the arguments of this operator are removed from the argument stack. Then, the operator `"-"` is evaluated and the result is put onto the argument stack. We have:

    ```
    mTokens    = [],
    mArguments = [ 3 ],
    mOperators = [].
    ```

    Therefore, the result of evaluating the arithmetic expression "`1+2*3-4`" is the number 3.

6.3.2 The Shunting-Yard-Algorithm

The algorithm introduced in the last example is known as the *shunting-yard-algorithm*. It was discovered by Edsger Dijkstra in 1961. We give a detailed presentation of this algorithm next. To begin with, we fix the data structures that are needed for this algorithm.

1. `mTokens` is a stack of input tokens. The operator symbols and parentheses are represented as strings, while the numbers are represented as rational numbers.
2. `mArguments` is a stack of rational numbers.
3. `mOperators` is the operator stack.

Considering the previous example we realize that the numbers are always put onto the argument stack, while there are two cases for operator symbols that are removed from the token stack:

1. We have to put the operator onto the operator stack in all of the following cases:
   (a) The operator stack is empty.
   (b) The operator on top of the operator stack is an opening parenthesis `"("`.
   (c) The operator has a higher precedence than the operator that is currently on top of the operator stack.
   (d) The operator is the same as operator that is on top of the operator stack and, furthermore, the operator associates to the right.
2. In all other cases, the operator that has been taken from the token stack is put back onto the token stack. In this case, the operator on top of the operator stack is removed from the operator stack and, furthermore, the arguments of this operator are removed from the argument stack. Next, this operator is evaluated and the resulting number is put onto the argument stack.

An implementation of this algorithm in SetlX is shown in the Figures 6.2 and 6.3 on the following pages. We start our discussion of the class `calculator` by inspecting the method `evalBefore` that is defined in line 35. This method takes two operators `stackOp` and `nextOp` and decides whether `stackOp` should be evaluated before `nextOp`. Of course, `stackOp` is intended to be the operator on top of the operator stack, while `nextOp` is an operator that is on top of the token stack. In order to decide whether the operator `stackOp` should be evaluated before the operator `nextOp`, we first have to know the *precedences* of these operators. Here, a *precedence* is a natural number that specifies how strong the operator binds to its arguments. Table 6.1 on page 80 lists the precedences of our operators. This table is coded as the binary relation `prec` in line 36.

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

Table 6.1: Precedences of the operators.

If the precedence of `stackOp` is bigger than the precedence of `nextOp`, then we have to evaluate `stackOp` before we evaluate `nextOp`. On the other hand, if the precedence of `stackOp` is smaller than the precedence of `nextOp`, then we have to

push `nextOp` onto the operator stack as we have to evaluate this operator before we

```
1  class calculator(s) {
2      mTokenStack := createStack(extractTokens(s));
3      mArguments  := stack();
4      mOperators  := stack();
5
6    static {
7      evaluate := procedure() {
8          while (!mTokenStack.isEmpty()) {
9              if (isInteger(mTokenStack.top())) {
10                 number := mTokenStack.top(); mTokenStack.pop();
11                 mArguments.push(number);
12                 continue;
13             }
14             nextOp := mTokenStack.top(); mTokenStack.pop();
15             if (mOperators.isEmpty() || nextOp == "(") {
16                 mOperators.push(nextOp);
17                 continue;
18             }
19             stackOp := mOperators.top();
20             if (stackOp == "(" && nextOp == ")") {
21                 mOperators.pop();
22             } else if (nextOp == ")") {
23                 popAndEvaluate();
24                 mTokenStack.push(nextOp);
25             } else if (evalBefore(stackOp, nextOp)) {
26                 popAndEvaluate();
27                 mTokenStack.push(nextOp);
28             } else {
29                 mOperators.push(nextOp);
30             }
31         }
32         while (!mOperators.isEmpty()) { popAndEvaluate(); }
33         return mArguments.top();
34     };
35     evalBefore := procedure(stackOp, nextOp) {
36         prec := {["+",1],["-",1],["*",2],["/",2],["%",2],["**",3]};
37         if (stackOp == "(") { return false;  }
38         if (prec[stackOp] > prec[nextOp]) {
39             return true;
40         } else if (prec[stackOp] == prec[nextOp]) {
41             if (stackOp == nextOp) {
42                 return stackOp in { "+", "-", "*", "/", "%" };
43             }
44             return true;
45         }
46         return false;
47     };
```

Figure 6.2: The class `calculator`, part 1.

```
48        popAndEvaluate := procedure() {
49            rhs := mArguments.top(); mArguments.pop();
50            lhs := mArguments.top(); mArguments.pop();
51            op  := mOperators.top(); mOperators.pop();
52            match (op) {
53                case "+" : result := lhs + rhs;
54                case "-" : result := lhs - rhs;
55                case "*" : result := lhs * rhs;
56                case "/" : result := lhs / rhs;
57                case "%" : result := lhs % rhs;
58                case "**": result := lhs ** rhs;
59                default: abort("ERROR: *** Unknown Operator *** $op$");
60            }
61            mArguments.push(result);
62        };
63      }
64    }
65
66    c := calculator("1+2*3**4-5/2");
67    c.evaluate();
```

Figure 6.3: The class `calculator`, part 2.

evaluate `stackOp`. If `stackOp` and `nextOp` have the same precedence, there are two cases:

1. `stackOp` $\neq$ `nextOp`.

   Let us consider an example: The arithmetic expression

   `2 + 3 - 4` is processed as `(2 + 3) - 4`.

   Therefore, in this case we have to evaluate `stackOp` first.

2. `op1` = `op2`.

   In this case we have to consider the *associativity* of the operator. Let us consider two examples:

   `2 + 3 + 4` is interpreted as `(2 + 3) + 4`.

   The reason is that the operator "`"+"`" *associates to the left.* On the other hand,

   `2 ** 3 ** 4` is interpreted as `2 ** (3 ** 4)`

   because the operator "`^`" *associates to the right.*

   The operators "`"+"`", "`"-"`", "`"*"`", "`/`" and "`%`" are all left associative. Hence, in this case `stackOp` is evaluated before `nextOp`. The operator "`**`" associates to the right. Therefore, if the operator on top of the operator stack is the operator "`**`" and then this operator is read again, then we have to push the operator "`**`" on the operator stack.

Now we can understand the implementation of `evalBefore`(`stackOp`, `nextOp`).

1. If `stackOp` is the opening parenthesis `"("`, we have to put `nextOp` onto the operator stack. The reason is that `"("` is no operator that can be evaluated. Hence, we return `false` in line 37.

2. If the precedence of `stackOp` is higher than the precedence of `nextOp`, we return `true` in line 39.
3. If the precedences of `stackOp` and `nextOp` are identical, there are two cases:
   (a) If both operator are equal, then the result of `evalBefore(stackOp,nextOp)` is `true` if and only if this operator associates to the left. The operators that associate to the left are listed in the set in line 42.
   (b) Otherwise, if `stackOp` is different from `nextOp`, then `evalBefore(stackOp,nextOp)` returns `true`.
4. If the precedence of `stackOp` is less than the precedence of `nextOp`, then `evalBefore(stackOp,nextOp)` returns `false`.

Figure 6.3 on page 82 shows the implementation of the method `popAndEvaluate`. This method works as follows:

1. It takes an operator from the operator stack (line 51),
2. it fetches the arguments of this operator from the argument stack (line 49 and line 50),
3. it evaluates the operator, and
4. finally puts the result back on top of the argument stack.

Finally, we are ready to discuss the implementation of the method `evaluate` in line 7 of Figure 6.2.

1. First, as long as the token stack is non-empty we take a token from the token stack.
2. If this token is a number, then we put it on the argument stack and continue to read the next token.

   In the following code of the `while` loop that starts at line 14, we can assume that the last token that has been read is either an operator symbol or one of the parentheses `"("` or `")"`.
3. If the operator stack is empty or if the token that has been read is an opening parenthesis `"("`, the operator or parenthesis is pushed onto the operator stack.
4. If the token that has been read as `nextOp` is a closing parenthesis `")"` and, furthermore, the operator on top of the operator stack is an opening parenthesis `"("`, then this parenthesis is removed from the operator stack.
5. If now in line 22 the token `nextOp` is a closing parenthesis `")"`, then we know that the token on the operator stack can't be an opening parenthesis but rather has to be an operator. This operator is then evaluated using the method `popAndEvaluate()`. Furthermore, the closing parenthesis `nextOp` is pushed back onto the token stack as we have not yet found the matching open parenthesis.

   After pushing the closing parenthesis back onto the token stack, we return to the beginning of the `while` loop in line 8. Hence, in this case we keep evaluating operators on the operator stack until we hit an opening parenthesis on the operator stack.

   In the part of the `while` loop following line 24 we may assume that `nextOp` is not a parenthesis, since the other case has been dealt with.

6. If the operator `stackOp` on top of the operator stack needs to be evaluated before the operator `nextOp`, we evaluate `stackOp` using the method `popAndEvaluate()`. Furthermore, the operator `nextOp` is put back on the token stack as it has not been consumed.

7. Otherwise, `nextOp` is put on the operator stack.

   The `while` loop ends when the token stack gets empty.

8. Finally, the operators remaining on the operator stack are evaluated using `popAndEvaluate`. If the input has been a syntactically correct arithmetic expression, then at the end of the computation there should be one number left on the argument stack. This number is the result of the evaluation and hence it is returned.

6.4 Benefits of Using Abstract Data Types

We finish this chapter with a short discussion of the benefits of abstract data types.

1. The use of abstract data types separates an algorithm from the data structures that are used to implement this algorithm.

   When we implemented the algorithm to evaluate arithmetic expressions we did not need to know how the data type *stack* that we used was implemented. It was sufficient for us to know

   (a) the signatures of its functions and

   (b) the axioms describing the behaviour of these functions.

   Therefore, an abstract data type can be seen as an interface that shields the user of the abstract data type from the peculiarities of an actual implementation of the data type. Hence it is possible that different groups of people develop the algorithm and the concrete implementation of the abstract data types used by algorithm.

   Today, many software systems have sizes that can only be described as gigantic. No single person is able to understand every single aspect of these systems. It is therefore important that these systems are structured in a way such that different groups of developers can work simultaneously on these systems without interfering with the work done by other groups.

2. Abstract data types are *reusable*.

   Our definition of stacks was very general. Therefore, stacks can be used in many different places: For example, we will see later how stacks can be used to traverse a directed graph.

   Modern industrial strength programming languages like `C++` or *Java* contain huge libraries containing the implementation of many abstract data types. This fact reduces the cost of software development substantially.

3. Abstract data types are *exchangeable*.

   In our program for evaluating arithmetic expressions it is trivial to substitute the given array based implementation of stacks with an implementation that is based on linked list. In general, this enables the following methodology for developing software:

   (a) First, an algorithm is implemented using abstract data types.

(b) The initial implementation of these abstract data may be quite crude and inefficient.

(c) Next, detailed performance tests spot those data types that are performance bottlenecks.

(d) Finally, the implementation of data types that have been identified as bottlenecks are optimized.

The reason this approach works is the 80-20 rule: 80 percent of the running time of most programs is spent in 20 percent of the code. It is therefore sufficient to optimize the implementation of those data structures that really are performance bottlenecks. If, instead, we would try to optimize everything we would only achieve the following:

(a) We would waste of time. There is no point optimizing some function to make it 10 times faster if the program spends less than a millisecond in this function anyway but the overall running time is several minutes.

(b) The resulting program would be considerably bigger and therefore more difficult to maintain.

Chapter 7

Sets and Maps

Wir haben bereits im ersten Semester gesehen, wie wichtig Mengen und funktionale Relationen, die wir im Folgenden als Abbildungen bezeichnen werden, in der Informatik sind. In diesem Kapitel zeigen wir, wie sich Mengen und Abbildungen effizient implementieren lassen. Wir können uns dabei auf Abbildungen beschränken, denn eine Menge M läßt sich immer als eine Abbildung f in die Menge $\{\texttt{true}, \texttt{false}\}$ darstellen, wenn wir

$$x \in M \overset{\text{def}}{\Longleftrightarrow} f(x) = \texttt{true}$$

definieren. Der Rest dieses Kapitels ist wie folgt aufgebaut:

1. Zunächst definieren wir den abstrakten Daten-Typ der *Abbildungen*.

   Anschließend stellen wir verschiedene Implementierungen dieses Daten-Typs vor.

2. Wir beginnen mit geordneten binären Bäumen. Die durchschnittliche Komplexität beim Einfügen eines Elements ist logarithmisch. Leider wächst die Komplexität im schlechtesten Fall linear mit der Anzahl der Enträge.

3. Wir betrachten als nächstes *balancierte binäre Bäume*. Bei diesen wächst die Komplexität der Operationen auch im schlechtesten Fall nur logarithmisch mit der Zahl der Einträge.

4. Anschließend betrachten wir die Daten-Struktur der *Tries*, die dann verwendet werden kann, wenn die Schlüssel, nach denen gesucht werden soll, Strings sind.

5. *Hash-Tabellen* stellen eine weitere Möglichkeit zur Implementierung von Abbildungen dar und werden im vierten Abschnitt diskutiert.

7.1 Der abstrakte Daten-Typ der *Abbildung*

In vielen Anwendungen der Informatik spielen *Abbildungen* einer Menge von sogenannten *Schlüsseln* in eine Menge von sogenannten *Werten* eine wichtige Rolle. Als ein Beispiel betrachten wir ein elektronisches Telefon-Buch wie es beispielsweise von einem Handy zur Verfügung gestellt wird. Die wesentlichen Funktionen, die ein solches Telefon-Buch anbietet, sind:

1. Nachschlagen eines gegebenen Namens und Ausgabe der diesem Namen zugeordneten Telefon-Nummer.

2. Einfügen eines neuen Eintrags mit Namen und Telefon-Nummer.

3. Löschen eines vorhandenen Eintrags.

Im Falle des Telefon-Buchs sind die *Schlüssel* die Namen und die *Werte* sind die Telefon-Nummern.

Definition 16 (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*
   $$\{map, find, insert, delete\}.$$
4. *Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:*
   (a) $map : Map$
   Der Aufruf $Map()$ *erzeugt eine leere Abbildung, also eine Abbildung, in der keine Werte gespeichert sind.*
   (b) $find : Map \times Key \to Value \cup \{\Omega\}$
   Der Aufruf $m.find(k)$ *überprüft, ob in der Abbildung* m *zu dem Schlüssel* k *ein Wert abgespeichert ist. Wenn ja, wird dieser Wert zurück gegeben, sonst wird der Wert* Ω *zurück gegeben.*
   (c) $insert : Map \times Key \times Value \to Map$
   Der Aufruf $m.insert(k, v)$ *fügt in der Abbildung* m *für den Schlüssel* k *den Wert* v *ein. Falls zu dem Schlüssel* k *bereits ein Eintrag in der Abbildung* m *existiert, so wird dieser überschrieben. Andernfalls wird ein entsprechender Eintrag neu angelegt. Als Ergebnis wird die geänderte Abbildung zurück gegeben.*
   (d) $delete : Map \times Key \to Map$
   Der Aufruf $m.delete(k)$ *entfernt den Eintrag zu dem Schlüssel* k *in der Abbildung* m. *Falls kein solcher Eintrag existiert, bleibt die Abbildung* m *unverändert. Als Ergebnis wird die eventuell geänderte Abbildung zurück gegeben.*
5. *Das genaue Verhalten der Funktionen wird durch die nachfolgenden Axiome spezifiziert.*
   (a) $map().find(k) = \Omega$,
   denn der Aufruf $map()$ *erzeugt eine leere Abbildung, in der sich zu keinem Schlüssel ein Wert findet.*
   (b) $m.insert(k, v).find(k) = v$,
   denn wenn wir zu dem Schlüssel k *einen Wert* v *einfügen, so finden wir anschließend eben diesen Wert* v, *wenn wir nach* k *suchen.*
   (c) $k_1 \neq k_2 \to m.insert(k_1, v).find(k_2) = m.find(k_2)$,
   denn wenn wir für einen Schlüssel k_1 *eine Wert einfügen, so ändert das nichts an dem Wert, der für einen anderen Schlüssel* k_2 *abgespeichert ist.*
   (d) $m.delete(k).find(k) = \Omega$,
   denn wenn wir einen Schlüssel k *löschen, so finden wir anschließend auch keinen Wert mehr, der unter dem Schlüssel* k *abgespeichert ist.*
   (e) $k_1 \neq k_2 \to m.delete(k_1).find(k_2) = m.find(k_2)$,
   denn wenn wir einen Schlüssel k_1 *löschen, so ändert das nichts an dem Wert, der unter einem anderen Schlüssel* k_2 *abgespeichert ist.* □

Es ist in SetlX sehr einfach, den ADT *Abbildung* zu implementieren. Dazu müssen wir uns nur klar machen, dass Abbildungen nichts anderes sind als Funktionen und die können wir in der Mengenlehre durch binäre Relationen darstellen. Ist r eine binäre Relation die genau ein Paar der Form $[k, v]$ enthält, bei dem die erste Komponente den Wert k hat, dann liefert der Ausdruck

$$r[k]$$

als Ergebnis den Wert v. Umgekehrt wird durch den Aufruf

$$r[k] := v$$

das Paar $[k, v]$ in die Relation r eingefügt. Um den Eintrag unter einem Schlüssel k zu löschen, reicht es aus, dem Schlüssel k den undefinierten Wert Ω zuzuweisen:

$$r[k] := \texttt{om}.$$

Dieser Wert wird auch bei der Auswertung des Ausdrucks $r[k]$ zurück gegeben, wenn die binäre Relation kein Paar der Form $[k, v]$ enthält. Abbildung 7.1 zeigt eine Implementierung des ADT Abbildung, die diese Überlegungen unmittelbar umsetzt.

```
1  class map() {
2      mRelation := {};
3    static {
4      find   := k |-> mRelation[k];
5      insert := procedure(k, v) { mRelation[k] := v;  };
6      delete := procedure(k)    { mRelation[k] := om; };
7    }
8  }
```

Figure 7.1: Eine triviale Implementierung des ADT *Map* in SetlX.

7.2 Geordnete binäre Bäume

Falls auf der Menge *Key* der Schlüssel eine totale Ordnung $\leq$ existiert, so kann eine einfache und zumindest im statistischen Durchschnitt effiziente Implementierung des abstrakte Daten-Typs *Map* mit Hilfe *geordneter binärer Bäume* erfolgen. Um diesen Begriff definieren zu können, führen wir zunächst *binäre Bäume* ein.

Definition 17 (Binäre Bäume)
Gegeben sei eine Menge Key von Schlüsseln und eine Menge Value von Werten. Dann definieren wir die Menge der binären Bäume $\mathcal{B}$ induktiv mit Hilfe der beiden Funktions-Zeichen Nil und Node, deren Typ-Spezifikationen wie folgt gegeben sind:

$$\mathit{Nil} : \mathcal{B} \qquad \text{und} \qquad \mathit{Node} : \mathit{Key} \times \mathit{Value} \times \mathcal{B} \times \mathcal{B} \to \mathcal{B}.$$

1. *Nil ist ein binärer Baum.*

   Dieser Baum wird als der leere Baum *bezeichnet.*

2. *Node(k, v, l, r) ist ein binärer Baum, falls gilt:*

   (a) *k ist ein Schlüssel aus der Menge Key.*

   (b) *v ist ein Wert aus der Menge Value.*

   (c) *l ist ein binärer Baum.*
   l wird als der linke Teilbaum *von Node(k, v, l, r) bezeichnet.*

(d) r *ist ein binärer Baum.*

r *wird als der* rechte Teilbaum *von* $Node(k, v, l, r)$ *bezeichnet.* □

Als nächstes definieren wir, was wir unter einem *geordneten binären Baum* verstehen.

Definition 18 (Geordnete binäre Bäume)
Die Menge $\mathcal{B}_<$ *der* geordneten binären Bäume *wird induktiv definiert.*

1. $Nil \in \mathcal{B}_<$
2. $Node(k, v, l, r) \in \mathcal{B}_<$ *falls folgendes gilt:*
   (a) k *ist ein Schlüssel aus der Menge Key.*
   (b) v *ist ein Wert aus der Menge Value.*
   (c) l *und* r *sind geordnete binäre Bäume.*
   (d) Alle Schlüssel, die in dem linken Teilbaum l *auftreten, sind kleiner als* k*.*
   (e) Alle Schlüssel, die in dem rechten Teilbaum r *auftreten, sind größer als* k*.*

   Die beiden letzten Bedingungen bezeichnen wir als die Ordnungs-Bedingung. □

Geordnete binäre Bäume lassen sich grafisch wir folgt darstellen:

1. Der leere Baum *Nil* wird durch einen dicken schwarzen Punkt dargestellt.
2. Ein Baum der Form $Node(k, v, l, r)$ wird dargestellt, indem zunächst ein Oval gezeichnet wird, in dem oben der Schlüssel k und darunter, getrennt durch einen waagerechten Strich, der dem Schlüssel zugeordnete Wert v eingetragen wird. Dieses Oval bezeichnen wir auch als einen *Knoten* des binären Baums. Anschließend wird links unten von diesem Knoten rekursiv der Baum l gezeichnet und rechts unten wird rekursiv der Baum r gezeichnet. Zum Abschluß zeichnen wir von dem mit k und v markierten Knoten jeweils einen Pfeil zu dem linken und dem rechten Teilbaum.

Abbildung 7.2 zeigt ein Beispiel für einen geordneten binären Baum. Der oberste Knoten, in der Abbildung ist das der mit dem Schlüssel 8 und dem Wert 22 markierte Knoten, wird als die *Wurzel* des Baums bezeichnet. Ein *Pfad der Länge* k in dem Baum ist eine Liste $[n_0, n_1, \cdots, n_k]$ von $k + 1$ Knoten, die durch Pfeile verbunden sind. Identifizieren wir Knoten mit ihren Markierungen, so ist

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

ein Pfad der Länge 3.

Wir überlegen uns nun, wie wir mit Hilfe geordneter binärer Bäume den ADT *Map* implementieren können. Wir spezifizieren die einzelnen Methoden dieses Daten-Typs durch bedingte Gleichungen. Der Konstruktor *map*() liefert als Ergebnis den leeren Baum zurück:

$$map() = Nil.$$

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

1. $Nil.find(k) = \Omega$,

   denn der leere Baum repräsentiert die leere Abbildung.

2. $Node(k, v, l, r), k).find(k) = v$,

   denn der Knoten $Node(k, v, l, r)$ speichert die Zuordnung $k \mapsto v$.

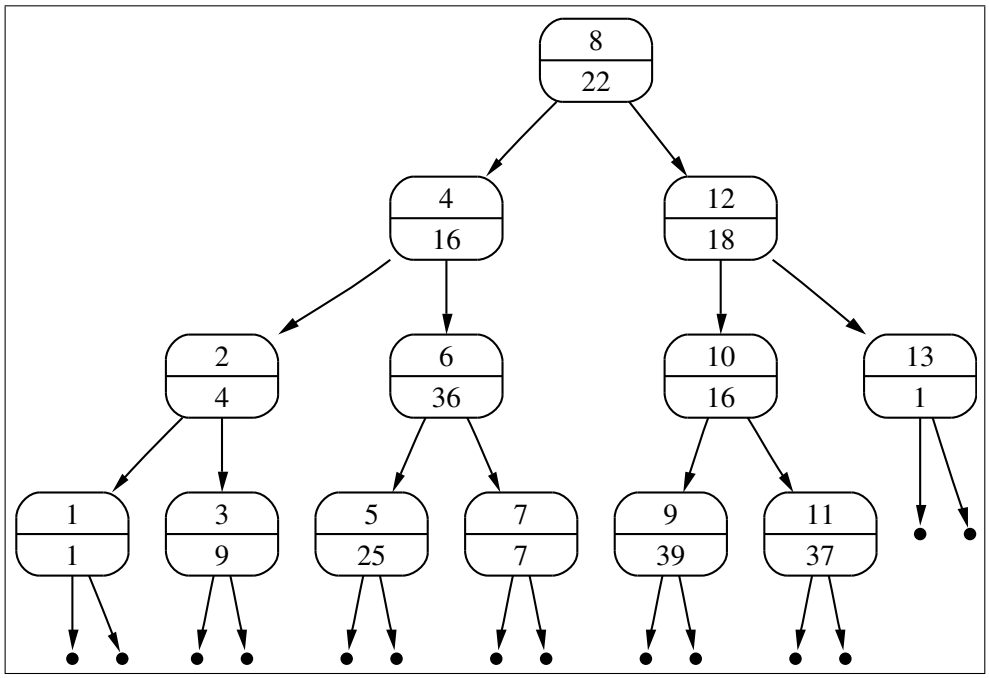

Figure 7.2: Ein geordneter binärer Baum

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. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{Node}(k_2, v_2, l.\mathit{insert}(k_1, v_1), r)$,

   denn wenn der Schlüssel k_1, unter dem wir Informationen einfügen wollen, kleiner als der Schlüssel k_2 an der Wurzel ist, so müssen wir die einzufügende Information im linken Teilbaum einfügen.

4. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{Node}(k_2, v_2, l, r.\mathit{insert}(k_1, v_1))$,

   denn wenn der Schlüssel k_1, unter dem wir Informationen einfügen wollen, größer als der Schlüssel k_2 an der Wurzel ist, so müssen wir die einzufügende Information im rechten Teilbaum einfügen.

Als letztes definieren wir die Methode *delete*. Diese Definition ist schwieriger als die Implementierung der andern beiden Methoden. Falls wir in einen Baum der Form $t = \mathit{Node}(k, v, l, r)$ 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 7.2 den Knoten mit der Markierung $\langle 4, 16\rangle$ löschen wollen, so suchen wir zunächst in dem Teilbaum, dessen Wurzel mit $\langle 6, 36\rangle$ markiert ist, den Knoten, der mit dem kleinsten Schlüssel markiert ist. Dies ist der Knoten mit der Markierung $\langle 5, 25\rangle$. Wir löschen diesen Knoten und überschreiben die Markierung $\langle 4, 16\rangle$ mit der Markierung $\langle 5, 25\rangle$. Abbildung 7.3 auf Seite 91 zeigt das Ergebnis.

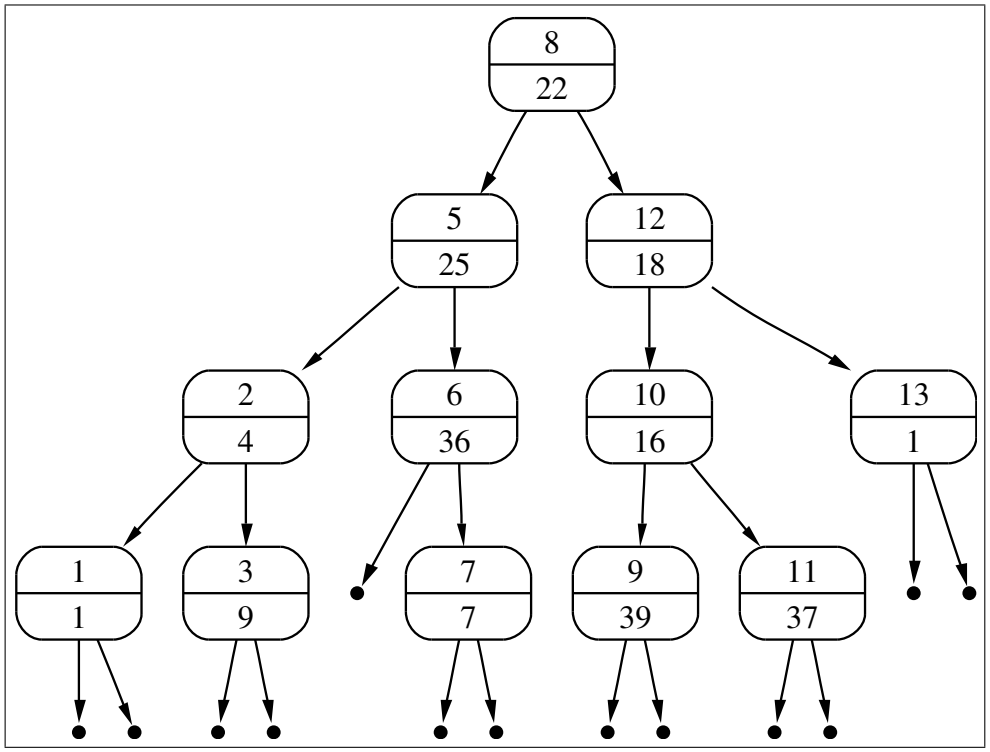

Figure 7.3: Der geordnete binärer Baum aus Abbildung 7.2 nach dem Entfernen des Knotens mit der Markierung $\langle 4, 16\rangle$.

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

1. $\mathit{Node}(k, v, \mathit{Nil}, r).\mathit{delMin}() = [r, k, v]$,

   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 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. $\mathit{Node}(k, v, \mathit{Nil}, r).\mathit{delete}\big(k\big) = r$.
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. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{delete}\big(k_1\big) = \mathit{Node}(k_2, v_2, l.\mathit{delete}(k_1), r)$,
   Falls der zu entfernende Schlüssel kleiner ist als der Schlüssel an der Wurzel, so kann sich der Schlüssel nur im linken Teilbaum befinden. Daher wird der Schlüssel k_1 rekursiv in dem linken Teilbaum l entfernt.
6. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{delete}(k_1) = \mathit{Node}(k_2, v_2, l, r.\mathit{delete}(k_1))$,
   denn in diesem Fall kann sich der Eintrag mit dem Schlüssel k_1 nur in dem rechten Teilbaum r befinden.

7.2.1 Implementing Ordered Binary Trees in SetlX

Figure 7.4 and Figure 7.5 show how ordered binary trees can be implemented in SetlX. Objects of class `map` encapsulate ordered binary trees. We discuss the implementation of this class next.

1. The constructor map is called with one argument. This argument, called `cmp` in line 1, is a function representing a total order "<". The idea is that the function `cmp` is called with two arguments and we have
   $$\texttt{cmp}(x, y) \quad \text{if and only if} \quad x < y.$$
   The function `cmp` is later stored in the member variable `mCmpFct` in line 6.
2. The class `map` represents a node in an ordered binary tree. In order to do so, it maintains four additional member variables.
   (a) `mKey` is the key stored at this node. For an empty node, `mKey` has the value `om`, which represents Ω.
   (b) `mValue` stores the value stored at this node. For an empty node, `mValue` is `om`.

(c) `mLeft` is the left subtree. An empty subtree is represented as `om`.

(d) `mRight` is the right subtree.

3. The function `isEmpty` checks whether `this` represents an empty tree. The assumption is that if `mKey` is `om`, then the member variables `mValue`, `mLeft`, and `mRight` will also be `om`.

4. The implementation of `find` works as follows:

   (a) If the node is empty, there is no value to find and the function returns `om`. Note that in SetlX a `return` statement which does not return a value automatically returns `om`.

   (b) If the key we are looking for is stored at the root of this tree, the value stored for this key is `mValue`.

   (c) Otherwise, we have to compare the key `k`, which is the key we are looking for, with the key `mKey`, which is the key stored in this node. If `k` is less than `mKey`, `k` can at most be stored in the left subtree `mLeft`, while if k is greater than `mKey`, `k` can only be stored in the right subtree.

```
1  class map(cmp) {
2      mKey    := om;
3      mValue  := om;
4      mLeft   := om;
5      mRight  := om;
6      mCmpFct := cmp;  // function to compare keys
7    static {
8      isEmpty := [] |-> mKey == om;
9      find := procedure(k) {
10         if      (isEmpty())        { return;                }
11         else if (mKey == k)        { return mValue;         }
12         else if (mCmpFct(k, mKey)) { return mLeft .find(k); }
13         else                       { return mRight.find(k); }
14     };
15     insert := procedure(k, v) {
16           if (isEmpty()) {
17             this.mKey   := k;
18             this.mValue := v;
19             this.mLeft  := map(mCmpFct);
20             this.mRight := map(mCmpFct);
21         } else if (mKey == k) {
22             mValue := v;
23         } else if (mCmpFct(k, mKey)) {
24             mLeft .insert(k, v);
25         } else {
26             mRight.insert(k, v);
27         }
28     };
```

Figure 7.4: Implementierung geordneter Bäume in SetlX, 1. Teil.

5. The implementation of `insert` is similar to the implementation of `find`.

(a) If the binary tree is empty, we just set the member variables `mKey` and `mValue` to the appropriate values.

(b) If the key `k` under which the value `v` is to be inserted is identical to the key `mKey` stored at this node, then we have found the node where we need to insert `v`. In this case, `mValue` is overwritten with `v`.

(c) Otherwise, `k` is compared with `mKey` and the search is continued in the appropriate subtree.

```
29      delMin := procedure() {
30          if (mLeft.isEmpty()) {
31              return [ mRight, mKey, mValue ];
32          } else {
33               [ ls, km, vm ] := mLeft.delMin();
34               this.mLeft := ls;
35               return [ this, km, vm ];
36          }
37      };
38      delete := procedure(k) {
39          if      (isEmpty())  { return; }
40          else if (k == mKey)  {
41              if      (mLeft .isEmpty()) { update(r); }
42              else if (mRight.isEmpty()) { update(l); }
43              else {
44                  [ rs, km, vm ] := mRight.delMin();
45                  this.mKey   := km;
46                  this.mValue := vm;
47                  this.mRight := rs;
48              }
49          } else if (mCmpFct(k, mKey)) {
50              if (!mLeft .isEmpty()) { mLeft.delete(k); }
51          } else {
52              if (!mRight.isEmpty()) { mRight.delete(k); }
53          }
54      };
55      update := procedure(t) {
56          this.mKey   := t.mKey;
57          this.mValue := t.mValue;
58          this.mLeft  := t.mLeft;
59          this.mRight := t.mRight;
60      };
61    }
62  }
```

Figure 7.5: Ordered binary trees in SetlX, part2.

6. The implementation of `delMin` is done in a similar way as the implementation of `insert`. It should be noted that the implementation follows directly from the equations derived previously.

7.2.2 Analyse der Komplexität

Wir untersuchen zunächst die Komplexität der Funktion *find* im schlechtesten Fall. Dieser Fall tritt dann ein, wenn der binäre Baum zu einer Liste entartet. Abbildung 7.6 zeigt den geordneten binären Baum der dann entsteht, wenn die Paare aus Schlüssel und Werten aus der Abbildung 7.2 in aufsteigender Reihenfolge eingegeben werden. Wird hier nach dem größten Schlüssel gesucht, so muß der komplette Baum durchlaufen werden. Enthält der Baum n Schlüssel, so sind also insgesamt n Vergleiche erforderlich. In diesem Fall ist ein geordneter binärer Baum also nicht besser als eine Liste.

Erfreulicherweise tritt der schlechteste Fall im statistischen Durchschnitt selten auf. Im Durchschnitt ist ein zufällig erzeugter binärer Baum recht gut balanciert, so dass beispielsweise für einen Aufruf von `find()` für einen Baum mit n Schlüsseln durchschnittlich $\mathcal{O}\big(\ln(n)\big)$ Vergleiche erforderlich sind. Wir werden diese Behauptung nun beweisen.

Wir bezeichnen die durchschnittliche Anzahl von Vergleichen, die beim Aufruf $b.find(k)$ für einen geordneten binären Baum b durchgeführt werden müssen, falls b insgesamt n Schlüssel enthält, mit d_n. Wir wollen annehmen, dass der Schlüssel k auch wirklich in b zu finden ist. Unser Ziel ist es, für d_n eine Rekurrenz-Gleichung aufzustellen. Zunächst ist klar, dass

$$d_1 = 1$$

ist, denn wenn der Baum b nur einen Schlüssel enthält, wird genau einen Vergleich durchgeführt. Wir betrachten nun einen binären Baum b, der $n+1$ Schlüssel enthält. Dann hat b die Form

$$b = node(k', v, l, r).$$

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

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

so gibt es $n+1$ verschiedene Positionen, an denen der Schlüssel k' auftreten kann. Wenn $k' = k_i$ ist, so enthält der linke Teilbaum i Schlüssel und der rechte Teilbaum enthält $n-i$ Schlüssel:

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

Da b insgesamt $n+1$ Schlüssel enthält, gibt es $n+1$ Möglichkeiten, wie die verbleibenden n Schlüssel auf die beiden Teilbäume l und r verteilt sein können, denn l kann i Schlüssel enthalten, wobei

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

gilt. Entsprechend enthält r dann $n-i$ Schlüssel. Bezeichnen wir die durchschnittliche Anzahl von Vergleichen bei einer Suche in einem Baum mit $n+1$ Schlüsseln, dessen linker Teilbaum i Elemente hat, mit

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

so gilt

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

denn wir haben ja angenommen, dass alle Werte von i die gleiche Wahrscheinlichkeit, nämlich $\frac{1}{n+1}$, haben.

Berechnen wir nun den Wert von $anzVgl(i, n{+}1)$: Falls l aus i Schlüsseln besteht und die restlichen $n-i$ Schlüssel in r liegen, so gibt es für den Schlüssel k, nach

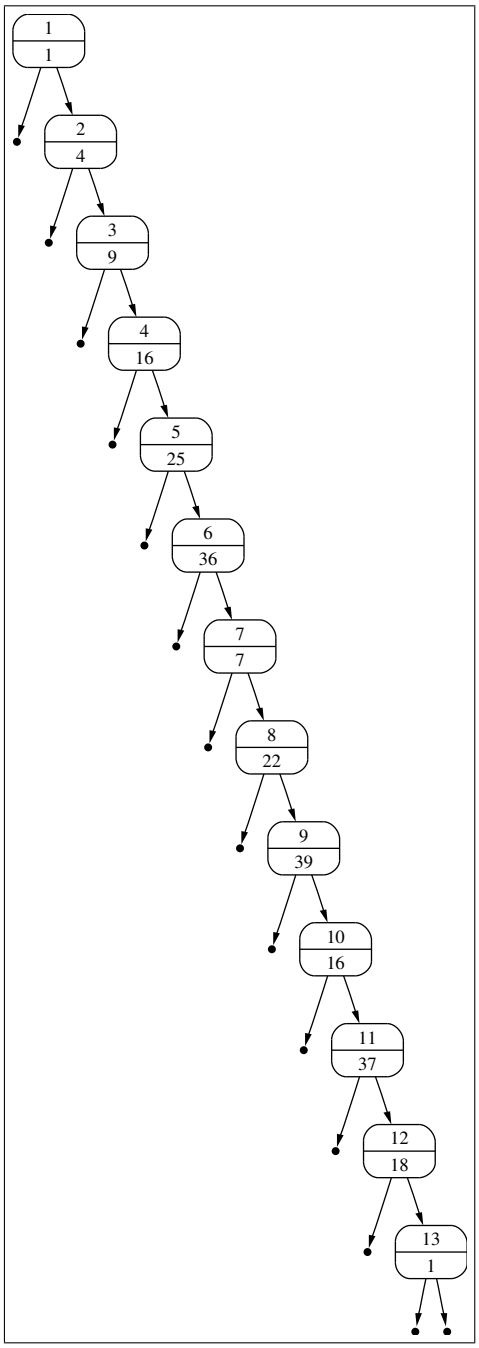

Figure 7.6: Ein entarteter geordneter binärer Baum.

dem wir in dem Aufruf $b.\mathit{find}(k)$ suchen, 3 Möglichkeiten:

1. k kann mit dem Schlüssel k' an der Wurzel des Baums übereinstimmen. In diesem Fall führen wir nur einen Vergleich durch. Da es insgesamt $n+1$ Schlüssel in dem Baum gibt und nur in einem dieser Fälle der Schlüssel, den wir suchen, an der Wurzel steht, hat dieser Fall die Wahrscheinlichkeit

$$\frac{1}{n+1}.$$

2. k kann mit einem der i Schlüssel im linken Teilbaum l übereinstimmen. Da der linke Teilbaum i Schlüssel enthält und es insgesamt $n+1$ Schlüssel gibt, hat die Wahrscheinlichkeit, dass k in dem linken Teilbaum l auftritt, den Wert

$$\frac{i}{n+1}.$$

In diesem Fall werden

$$d_i + 1$$

Vergleiche durchgeführt, denn außer den d_i Vergleichen mit den Schlüsseln aus dem linken Teilbaum muss der Schlüssel, der gesucht wird, ja noch mit dem Schlüssel an der Wurzel verglichen werden.

3. k kann mit einem der $n-i$ Schlüssel im rechten Teilbaum r übereinstimmen.

Da der rechte Teilbaum $n-i$ Schlüssel enthält und es insgesamt $n+1$ Schlüssel gibt, hat die Wahrscheinlichkeit, dass k in dem rechten Teilbaum r auftritt, den Wert

$$\frac{n-i}{n+1}.$$

Analog zum zweiten Fall werden diesmal

$$d_{n-i} + 1$$

Vergleiche durchgeführt.

Um nun $anzVgl(i, n+1)$ berechnen zu können, müssen wir in jedem der drei Fälle die Wahrscheinlichkeit mit der Anzahl der Vergleiche multiplizieren und die Werte, die sich für die drei Fälle ergeben, aufsummieren. Wir erhalten

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

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

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

Bei der letzten Umformung haben wir die Gleichung

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

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

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

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

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

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

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

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

Subtrahieren wir Gleichung (7.3) von Gleichung (7.4), so erhalten wir

$$(n+2)^2 \cdot d_{n+2} - (n+1)^2 \cdot d_{n+1} = (n+2)^2 - (n+1)^2 + 2 \cdot (n+1) \cdot d_{n+1}.$$

Zur Vereinfachung substituieren wir hier $n \mapsto n-1$ und erhalten

$$(n+1)^2 \cdot d_{n+1} - n^2 \cdot d_n = (n+1)^2 - n^2 + 2 \cdot n \cdot d_n.$$

Dies vereinfachen wir zu

$$(n+1)^2 \cdot d_{n+1} = n \cdot (n+2) \cdot d_n + 2 \cdot n + 1.$$

Bei dieser Gleichung teilen wir auf beiden Seiten durch $(n{+}2){\cdot}(n{+}1)$ und bekommen

$$\frac{n+1}{n+2} \cdot d_{n+1} = \frac{n}{n+1} \cdot d_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)}.$$

Nun definieren wir

$$c_n = \frac{n}{n+1} \cdot d_n.$$

Dann gilt $c_1 = \frac{1}{2} \cdot d_1 = \frac{1}{2}$ und wir haben die Rekurrenz-Gleichung

$$c_{n+1} = c_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)}.$$

Durch Partialbruch-Zerlegung finden wir

$$\frac{2 \cdot n + 1}{(n+2) \cdot (n+1)} = \frac{3}{n+2} - \frac{1}{n+1}.$$

Also haben wir

$$c_{n+1} = c_n + \frac{3}{n+2} - \frac{1}{n+1}.$$

Wegen $c_1 = \frac{1}{2}$ ist die Gleichung auch für $n = 0$ richtig, wenn wir $c_0 = 0$ setzen, denn es gilt

$$\frac{1}{2} = 0 + \frac{3}{0+2} - \frac{1}{0+1}.$$

Die Rekurrenz-Gleichung für c_n können wir mit dem Teleskop-Verfahren lösen:

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

Wir substituieren $n \mapsto n-1$ und vereinfachen dies zu

$$c_n = \sum_{i=2}^{n+1} \frac{3}{i} - \sum_{i=1}^{n} \frac{1}{i}$$

Die harmonische Zahl H_n ist als $H_n = \sum_{i=1}^{n} \frac{1}{i}$ definiert. Wir können c_n auf H_n zurückführen:

$$\begin{aligned} c_n &= 3 \cdot H_n - \frac{3}{1} + \frac{3}{n+1} - H_n \\ &= 2 \cdot H_n - 3 \cdot \frac{n}{n+1} \end{aligned}$$

Wegen $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln(n) + \mathcal{O}(1)$ gilt dann

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

Berücksichtigen wir $d_n = \frac{n+1}{n} \cdot c_n$, so finden wir für große n ebenfalls

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

Das ist unser zentrales Ergebnis: Im Durchschnitt erfordert das Suchen in einem zufällig erzeugten geordneten binären Baum für große Werte von n etwa

$$2 \cdot \ln(n) = 2 \cdot \ln(2) \cdot \log_2(n) \approx 1.386 \cdot \log_2(n)$$

Vergleiche. Damit werden etwa 39 % mehr Vergleiche ausgeführt als bei einem optimal balancierten binären Baum. Ähnliche Ergebnisse können wir für das Einfügen oder Löschen erhalten.

7.3 AVL-Bäume

Es gibt verschiedene Varianten von geordneten binären Bäumen, bei denen auch im schlechtesten Fall die Anzahl der Vergleiche nur logarithmisch von der Zahl der Schlüssel abhängt. Eine solche Variante sind die *AVL-Bäume* [AVL62], die nach ihren Erfindern G. M. Adel'son-Vel'skiĭ und E. M. Landis benannt sind. Diese Variante stellen wir jetzt vor. Dazu definieren wir zunächst die *Höhe* eines binären Baums:

1. $\mathit{Nil}.\mathit{height}() = 0$.
2. $\mathit{Node}(k, v, l, r).\mathit{height}() = \max\big(l.\mathit{height}(), r.\mathit{height}()\big) + 1$.

Definition 19 (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 Balancierungs-Bedingung wiederherstellen können, wenn diese beim Einfügen oder Löschen eines Elements verletzt wird. Der Aufruf $b.\mathit{restore}()$ setzt voraus, dass b ein geordneter binärer Baum ist, für den außer an der Wurzel überall die Balancierungs-Bedingung erfüllt ist. An der Wurzel kann die Höhe des linken Teilbaums um maximal 2 von der Höhe des rechten Teilbaums abweichen. Beim Aufruf der Methode $b.\mathit{restore}()$ liegt also einer der beiden folgenden Fälle vor:

1. $b = \mathit{Nil}$ oder
2. $b = \mathit{Node}(k, v, l, r) \wedge l \in \mathcal{A} \wedge r \in \mathcal{A} \wedge |l.\mathit{height}() - r.\mathit{height}()| \leq 2$.

Wir spezifizieren die Methode *restore* durch bedingte Gleichungen.

1. $\mathit{Nil}.\mathit{restore}() = \mathit{Nil}$,
   denn der leere Baum ist ein AVL-Baum.
2. $|l.\mathit{height}() - r.\mathit{height}()| \leq 1 \to \mathit{Node}(k, v, l, r).\mathit{restore}() = \mathit{Node}(k, v, l, r)$,
   denn wenn die Balancierungs-Bedingung bereits erfüllt ist, braucht nichts getan werden.

3. $l_1.height() = r_1.height() + 2$
   $\wedge \quad l_1 = Node(k_2, v_2, l_2, r_2)$
   $\wedge \quad l_2.height() \geq r_2.height()$
   $\rightarrow \quad Node(k_1, v_1, l_1, r_1).restore() = Node\big(k_2, v_2, l_2, Node(k_1, v_1, r_2, r_1)\big)$

   Um diese Gleichung zu verstehen, betrachten wir Abbildung 7.7 auf Seite 101. 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.

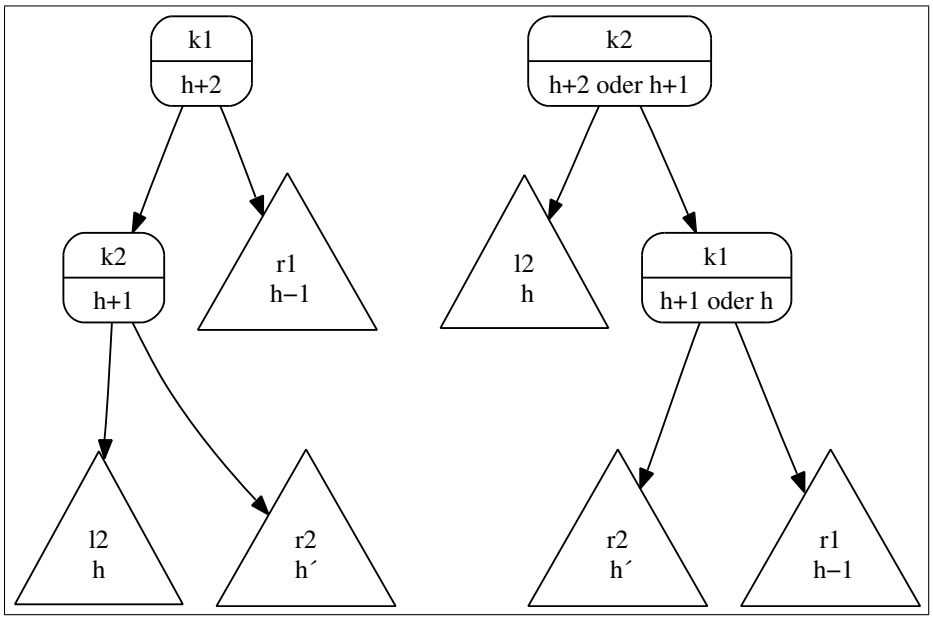

Figure 7.7: Ein unbalancierter Baum und der rebalancierte Baum

Wir müssen uns davon überzeugen, dass der im rechten Teil von Abbildung 7.7 gezeigte Baum auch tatsächlich ein AVL-Baum ist. Was die Balancierungs-Bedingung angeht, so rechnet man dies sofort nach. Die Tatsache, dass der mit k_1 markierte Knoten entweder die Höhe h oder $h + 1$ hat folgt daraus, dass r_1 die Höhe $h - 1$ hat und dass $h' \in \{h, h - 1\}$ gilt.

Um zu sehen, dass der Baum geordnet ist, können wir folgende Ungleichung hinschreiben:

$$l_2 < k_2 < r_2 < k_1 < r_1. \qquad (\star)$$

Dabei schreiben wir für einen Schlüssel k und einen binären 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. $$\begin{array}{ll} & l_1.height() = r_1.height() + 2 \\ \wedge & l_1 = Node(k_2, v_2, l_2, r_2) \\ \wedge & l_2.height() < r_2.height() \\ \wedge & r_2 = Node(k_3, v_3, l_3, r_3) \\ \rightarrow & Node(k_1, v_1, l_1, r_1).restore() = Node\big(k_3, v_3, Node(k_2, v_2, l_2, l_3), Node(k_1, v_1, r_3, r_1)\big) \end{array}$$

   Die linke Seite der Gleichung wird durch die Abbildung 7.8 auf Seite 103 illustriert. Dieser Baum kann in der Form

   $$Node\big(k_1, v_1, Node(k_2, v_2, l_2, Node\big(k_3, v_3, l_3, r_3)\big), r_1\big)$$

   geschrieben werden. Die Teilbäume l_3 und r_3 haben hier entweder die Höhe h oder $h-1$, wobei mindestens einer der beiden Teilbäume die Höhe h haben muß.

   Die Situation der rechten Seite der obigen Gleichung zeigt Abbildung 7.9 auf Seite 104. Der auf dieser Abbildung gezeigte Baum hat die Form

   $$Node\big(k_3, v_3, Node(k_2, v_2, l_2, l_3), Node(k_1, v_1, r_3, r_1)\big).$$

   Die Ungleichung, die die Anordnung der Schlüssel sowohl im linken als auch rechten Baum wieder gibt, lautet

   $$l_2 < k_2 < l_3 < k_3 < r_3 < k_1 < r_1.$$

   Es gibt noch zwei weitere Fälle die auftreten, wenn der rechte Teilbaum um mehr als Eins größer ist als der linke Teilbaum. Diese beiden Fälle sind aber zu den beiden vorherigen Fällen völlig analog, so dass wir die Gleichungen hier ohne weitere Diskussion angeben.

5. $$\begin{array}{ll} & r_1.height() = l_1.height() + 2 \\ \wedge & r_1 = Node(k_2, v_2, l_2, r_2) \\ \wedge & r_2.height() \geq l_2.height() \\ \rightarrow & Node(k_1, v_1, l_1, r_1).restore() = Node\big(k_2, v_2, Node(k_1, v_1, l_1, l_2), r_2\big) \end{array}$$

6. $$\begin{array}{ll} & r_1.height() = l_1.height() + 2 \\ \wedge & r_1 = Node(k_2, v_2, l_2, r_2) \\ \wedge & r_2.height() < l_2.height() \\ \wedge & l_2 = Node(k_3, v_3, l_3, r_3) \\ \rightarrow & Node(k_1, v_1, l_1, r_1).restore() = Node\big(k_3, v_3, Node(k_1, v_1, l_1, l_3), Node(k_2, v_2, r_3, r_2)\big) \end{array}$$

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. $Nil.insert(k, v) = Node(k, v, Nil, Nil)$.
2. $Node(k, v_2, l, r).insert(k, v_1) = Node(k, v_1, l, r)$.
3. $k_1 < k_2 \rightarrow Node(k_2, v_2, l, r).insert(k_1, v_1) = Node\big(k_2, v_2, l.insert(k_1, v_1), r\big).restore()$.
4. $k_1 > k_2 \rightarrow Node(k_2, v_2, l, r).insert\big(k_1, v_1\big) = Node\big(k_2, v_2, l, r.insert(k_1, v_1)\big).restore()$.

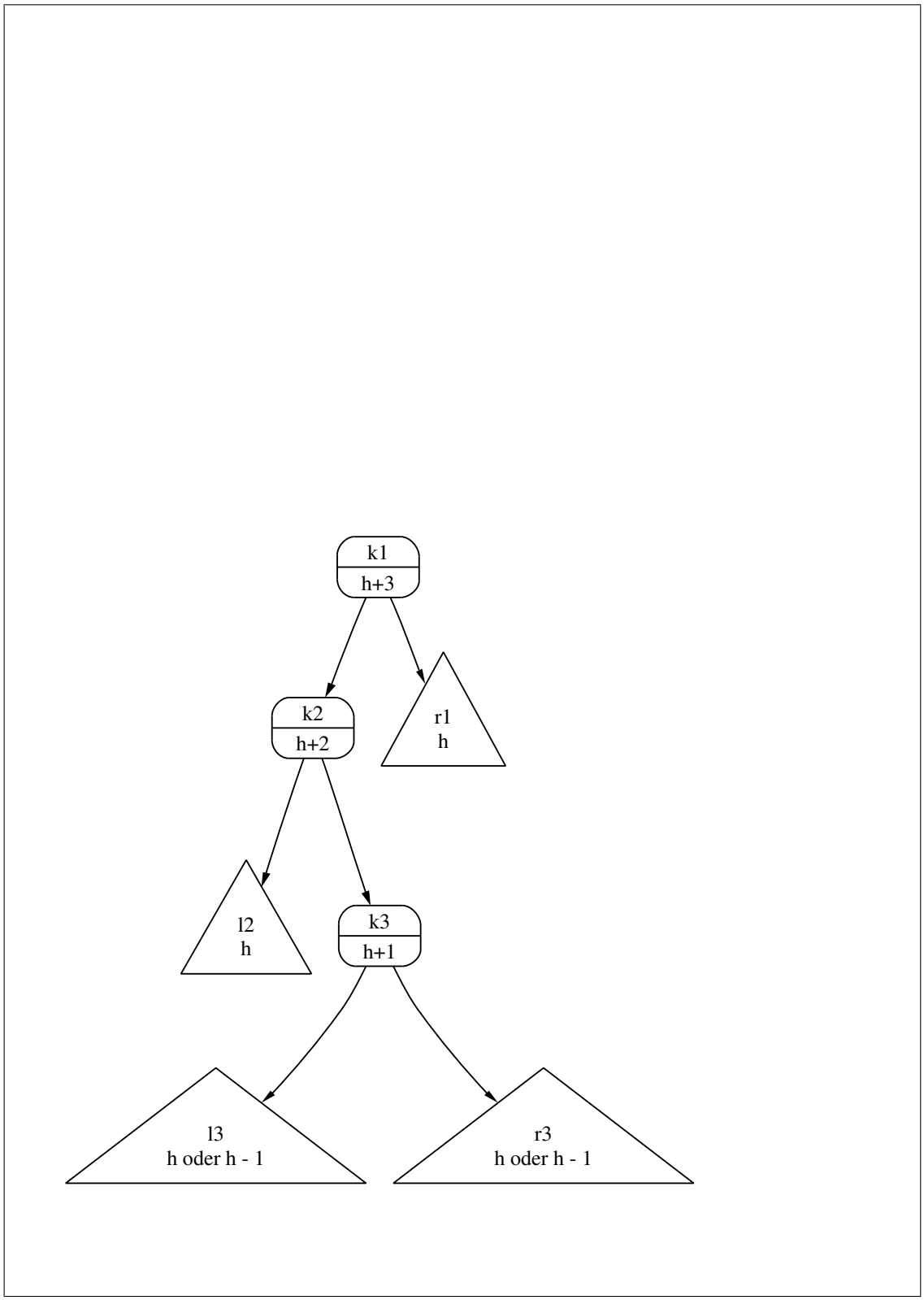

Figure 7.8: Ein unbalancierter Baum: 2. Fall

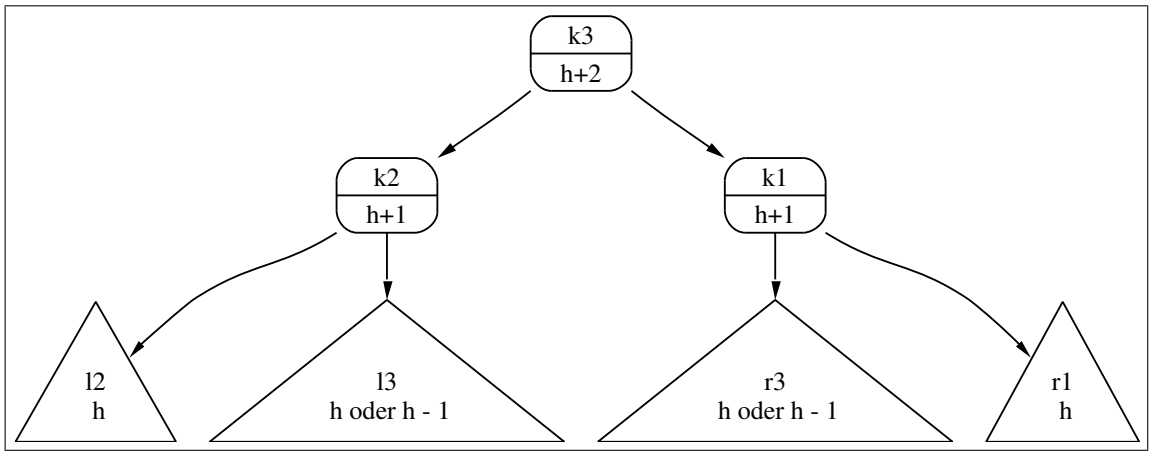

Figure 7.9: Der rebalancierte Baum im 2. Fall

Analog ändern sich die Gleichungen für *delMin*() wie folgt:

1. $Node(k, v, Nil, r).delMin() = \langle r, k, v\rangle$.
2. $l \neq Nil \wedge \langle l', k_{min}, v_{min}\rangle := l.delMin() \rightarrow$
   $Node(k, v, l, r).delMin() = \langle Node(k, v, l', r).restore(), k_{min}, v_{min}\rangle$.

Damit können wir die Gleichungen zur Spezifikation der Methode `delete`() angeben.

1. $Nil.delete(k) = Nil$.
2. $Node(k, v, Nil, r).delete(k) = r$.
3. $Node(k, v, l, Nil).delete(k) = l$.
4. $l \neq Nil \wedge r \neq Nil \wedge \langle r', k_{min}, v_{min}\rangle := r.delMin() \rightarrow$
   $Node(k, v, l, r).delete(k) = Node(k_{min}, v_{min}, l, r').restore()$.
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. $k_1 > k_2 \rightarrow Node(k_2, v_2, l, r).delete(k_1) = Node\big(k_2, v_2, l, r.delete(k_1)\big).restore()$.

7.3.1 Implementing AVL-Trees in SetlX

If we want to implement AVL-trees in SetlX then we have to decide how to compute the height of the trees. The idea is to store the height of every subtree in the corresponding node since it would be inefficient if we would recompute this height every time we need it. Therefore, we add a member variable `mHeight` to our class map. Figure 7.10 shows an outline of the class `map`. The variable `mHeight` is defined in line 6. It is initialised as 0 since the constructor `map` constructs an empty node.

```
1  class map(cmp) {
2      mKey    := om;
3      mValue  := om;
4      mLeft   := om;
5      mRight  := om;
6      mHeight := 0;
7      mCmpFct := cmp;
8
9    static {
10     isEmpty       := [] |-> mKey == om;
11     find          := procedure(k)          { ... };
12     insert        := procedure(k, v)       { ... };
13     delMin        := procedure()           { ... };
14     delete        := procedure(k)          { ... };
15     update        := procedure(t)          { ... };
16     restore       := procedure()           { ... };
17     setValues     := procedure(k, v, l, r) { ... };
18     restoreHeight := procedure()           { ... };
19   }
20 }
```

Figure 7.10: Outline of the class `map`.

Figure 7.11 show the implementation of the function `find`. Actually, the implementation is the same as the implementation in Figure 7.4. The reason is that every AVL tree is also an ordered binary tree and since searching for a key does not change the underlying tree there is no need to restore anything.

```
1  find := procedure(k) {
2      if      (isEmpty())        { return;                }
3      else if (mKey == k)        { return mValue;         }
4      else if (mCmpFct(k, mKey)) { return mLeft .find(k); }
5      else                       { return mRight.find(k); }
6  };
```

Figure 7.11: Implementation of the method `find`.

Figure 7.12 shows the implementation of the method `insert`. If we compare this implementation with the implementation for binary trees, we find three differences.

1. When inserting into an empty tree, we now have to update the member variable `mHeight` to 1. This is done in line 7.
2. After inserting a value into the left subtree `mLeft`, it might be necessary to rebalance the tree. This is done in line 12.
3. Similarly, if we insert a value into the right subtree `mRight`, we have to rebalance the tree. This is done in line 15.

```
1  insert := procedure(k, v) {
2      if (isEmpty()) {
3          this.mKey    := k;
4          this.mValue  := v;
5          this.mLeft   := map(mCmpFct);
6          this.mRight  := map(mCmpFct);
7          this.mHeight := 1;
8      } else if (mKey == k) {
9          this.mValue := v;
10     } else if (mCmpFct(k, mKey)) {
11         mLeft.insert(k, v);
12         restore();
13     } else {
14         mRight.insert(k, v);
15         restore();
16     }
17 };
```

Figure 7.12: Implementation of the method `insert`.

Figure 7.13 shows the implementation of the method `delMin`. The only change compared to the previous implementation for ordered binary trees is in line 7, where we have to take care of the fact that the balancing condition might be violated after deleting the smallest element in the left subtree.

```
1  delMin := procedure() {
2      if (mLeft.isEmpty()) {
3          return [ mRight, mKey, mValue ];
4      } else {
5           [ ls, km, vm ] := mLeft.delMin();
6          this.mLeft := ls;
7          restore();
8          return [ this, km, vm ];
9      }
10 };
```

Figure 7.13: Implementation of `delMin`.

Figure 7.14 shows the implementation of the method `delete` and the implementation of the auxiliary method `update`. Compared with Figure 7.5 there are only three differences:

1. If we delete the key at the root of the tree, we replace this key with the smallest key in the right subtree. Since this key is deleted in the right subtree, the height of the right subtree might shrunk and hence the balancing condition at the root might be violated. Therefore, we have to restore the balancing condition. This is done in line 12.

2. If we delete a key in the left subtree, the height of the left subtree might shrink. Hence we have to rebalance the tree at the root in line 16.

3. Similarly, if we delete a key in the right subtree, we have to restore the balancing condition. This is done in line 19.

Of course, since the method `update` only sets the member variables of the tree, it does not change the structure of the tree. Hence there is no need for a call to `restore` in this method.

```
1   delete := procedure(k) {
2       if (isEmpty())  {
3           return;
4       } else if (k == mKey) {
5           if (mLeft.isEmpty()) {
6               update(mRight);
7           } else if (mRight.isEmpty()) {
8               update(mLeft);
9           } else {
10              [ rs, km, vm ] := mRight.delMin();
11              [this.mKey,this.mValue,this.mRight ] := [km,vm,rs];
12              restore();
13          }
14      } else if (mCmpFct(k, mKey)) {
15          mLeft.delete(k);
16          restore();
17      } else {
18          mRight.delete(k);
19          restore();
20      }
21  };
22  update := procedure(t) {
23      this.mKey    := t.mKey;
24      this.mValue  := t.mValue;
25      this.mLeft   := t.mLeft;
26      this.mRight  := t.mRight;
27      this.mHeight := t.mHeight;
28  };
```

Figure 7.14: The methods `delete` and `update`.

```
1  restore := procedure() {
2      if (abs(mLeft.mHeight - mRight.mHeight) <= 1) {
3          restoreHeight();
4          return;
5      }
6      if (mLeft.mHeight > mRight.mHeight) {
7          [ k1,v1,l1,r1 ] := [ mKey,mValue,mLeft,mRight ];
8          [ k2,v2,l2,r2 ] := [ l1.mKey,l1.mValue,l1.mLeft,l1.mRight ];
9          if (l2.mHeight >= r2.mHeight) {
10             setValues(k2,v2,l2,createNode(k1,v1,r2,r1,mCmpFct));
11         } else {
12             [ k3,v3,l3,r3 ] := [r2.mKey,r2.mValue,r2.mLeft,r2.mRight];
13             setValues(k3,v3,createNode(k2,v2,l2,l3,mCmpFct),
14                             createNode(k1,v1,r3,r1,mCmpFct) );
15         }
16     }
17     if (mRight.mHeight > mLeft.mHeight) {
18         [ k1,v1,l1,r1 ] := [ mKey,mValue,mLeft,mRight ];
19         [ k2,v2,l2,r2 ] := [ r1.mKey,r1.mValue,r1.mLeft,r1.mRight ];
20         if (r2.mHeight >= l2.mHeight) {
21             setValues(k2,v2,createNode(k1,v1,l1,l2,mCmpFct),r2);
22         } else {
23             [ k3,v3,l3,r3 ] := [l2.mKey,l2.mValue,l2.mLeft,l2.mRight];
24             setValues(k3,v3,createNode(k1,v1,l1,l3,mCmpFct),
25                             createNode(k2,v2,r3,r2,mCmpFct) );
26         }
27     }
28     restoreHeight();
29 };
30 setValues := procedure(k, v, l, r) {
31     this.mKey   := k;
32     this.mValue := v;
33     this.mLeft  := l;
34     this.mRight := r;
35 };
36 restoreHeight := procedure() {
37     this.mHeight := 1 + max({ mLeft.mHeight, mRight.mHeight });
38 };
```

Figure 7.15: The implementation of `restore` and `restoreHeight`.

Figure 7.15 shows the implementation of the function `restore`. It is this method that makes most of the difference between ordered binary trees and AVL trees. Let us discuss this method line by line.

1. In line 2 we check whether the balancing condition is satisfied. If we are lucky, this test is successful and hence we do not need to restore the structure of the tree. However, we still need to maintain the height of the tree since it is possible that variable `mHeight` no longer contains the correct height. For example, assume that the left subtree initially has a height that is bigger by one than the height of the right subtree. Assume further that we have deleted a node in the left subtree so that its height shrinks. Then the balancing

condition is still satisfied, as now the left subtree and the right subtree have the same height. However, the height of the complete tree has also shrink by one and therefore, the variable `mHeight` needs to be decremented. This is done via the auxiliary method `restoreHeight`. This method is defined in line 36 and it recomputes `mHeight` according to the definition of the height of a binary tree.

2. If the check in line 2 fails, then we know that the balancing condition is violated. However, we do not yet know which of the two subtrees is bigger.

   If the test in line 6 succeeds, then the left subtree must have a height that is bigger by two than the height of the right subtree. In order to be able to use the same variable names as the variable names given in the equations discussed in the previous subsection, we define the variables `k1`, `v1`, ···, `l2`, and `r2` in line 7 and 8 so that these variable names correspond exactly to the variable names used in the Figures 7.7 and 7.8.

3. Next, the test in line 9 checks whether we have the case that is depicted in Figure 7.7. In this case, Figure 7.7 tells us that the key `k2` has to move to the root. The left subtree is now `l2`, while the right subtree is a new node that has the key `k1` at its root. This new node is created by the call of the function `createNode` in line 10. The function `createNode` is shown in Figure 7.16 on page 109.

4. If the test in line 9 fails, the right subtree is bigger than the left subtree and we are in the case that is depicted in Figure 7.8. We have to create the tree that is shown in Figure 7.9. To this end we first define the variables `k3`, `v3`, `l3`, and `r3` in a way that these variables correspond to the variables shown in Figure 7.8. Next, we create the tree that is shown in Figure 7.9.

5. Line 17 deals with the case that the right subtree is bigger than the left subtree. As this case is analogous to the case covered in line 6 to line 16, we won't discuss this case any further.

6. Finally, we recompute the variable `mHeight` since it is possible that the old value is no longer correct.

```
1  createNode :=  procedure(key, value, left, right, cmp) {
2      node         := map(cmp);
3      node.mKey    := key;
4      node.mValue  := value;
5      node.mLeft   := left;
6      node.mRight  := right;
7      node.mCmpFct := cmp;
8      node.mHeight := 1 + max({ left.mHeight, right.mHeight });
9      return node;
10 };
```

Figure 7.16:

The function `createNode` shown in Figure 7.16 constructs a node with given left and right subtrees. In fact, this method serves as a second constructor for the class `map`. The implementation should be obvious.

7.3.2 Analyse der Komplexität

Wir analysieren jetzt die Komplexität von AVL-Bäumen im schlechtesten Fall. Der schlechteste Fall tritt dann ein, wenn bei einer vorgegebenen Zahl von Schlüsseln die Höhe maximal wird. Das ist aber das selbe wie wenn in einem Baum gegebener Höhe die Zahl der Schlüssel minimal wird. Wir definieren daher $b_h(k)$ als einen AVL-Baum der Höhe h, der unter allen AVL-Bäumen der Höhe h die minimale Anzahl von Schlüsseln hat. Außerdem sollen alle Schlüssel, die in $b_h(k)$ auftreten, größer als der Schlüssel k sein. Sowohl die Schlüssel als auch die Werte sind in diesem Zusammenhang eigentlich unwichtig, wir müssen nur darauf achten, dass die Ordnungs-Bedingung für binäre Bäume erfüllt ist. Wir werden für die Schlüssel natürliche Zahlen nehmen, für die Werte nehmen wir immer die Zahl 0. Bevor wir mit der Definition von $b_h(k)$ beginnen können, benötigen wir noch eine Hilfs-Funktion *maxKey*() mit der Signatur

$$maxKey : \mathcal{B}_< \to Key$$

Für einen gegebenen geordneten nicht-leeren binären Baum b berechnet $b.maxKey()$ den größten Schlüssel, der in b auftritt. Die Definition von $b.maxKey()$ ist induktiv:

1. $Node(k, v, l, Nil).maxKey() = k$,
2. $r \neq Nil \to Node(k, v, l, r).maxKey() = r.maxKey()$.

Damit können wir nun die Bäume $b_h(k)$ durch Induktion nach der Höhe h definieren.

1. $b_0(k) = Nil$,

   denn es gibt genau einen AVL-Baum der Höhe 0 und dieser enthält keinen Schlüssel.

2. $b_1(k) = Node(k + 1, 0, Nil, Nil)$,

   denn es gibt genau einen AVL-Baum der Höhe 1.

3. $b_{h+1}(k).maxKey() = l \to b_{h+2}(k) = Node\big(l + 1,\ 0,\ b_{h+1}(k),\ b_h(l + 1)\big)$,

   denn um einen AVL-Baum der Höhe $h + 2$ mit einer minimalen Anzahl an Schlüsseln zu konstruieren, erzeugen wir zunächst den AVL-Baum $b_{h+1}(k)$ der Höhe $h+1$. Dann bestimmen wir den maximalen Schlüssel l in diesem Baum, der Schlüssel $l + 1$ kommt nun an die Wurzel des zu erzeugenden Baums der Höhe $h + 2$ und schließlich erzeugen wir noch den Baum $b_h(l + 1)$ der Höhe h, den wir als rechten Teilbaum in den neu zu erzeugenden Baum der Höhe $h + 2$ einfügen.

Für einen beliebigen binären Baum b bezeichne $\# b$ die Anzahl der Schlüssel, die in b auftreten. Dann definieren wir

$$c_h := \# b_h(k)$$

als die Anzahl der Schlüssel des Baums $b_h(k)$. Wir werden sofort sehen, dass $\# b_h(k)$ nicht von k abhängt. Für c_h finden wir in Analogie zu der induktiven Definition von $b_h(k)$ die folgenden Gleichungen.

1. $c_0 = \# b_0(k) = \# Nil = 0$,
2. $c_1 = \# b_1(k) = \# Node(k + 1, 0, Nil, Nil) = 1$,
3. $$\begin{array}{rcl} c_{h+2} & = & \# b_{h+2}(k) \\ & = & \# Node\big(l + 1,\ 0,\ b_{h+1}(k),\ b_h(l + 1)\big) \\ & = & \# b_{h+1}(k) + \# b_h(l + 1) + 1 \\ & = & c_{h+1} + c_h + 1. \end{array}$$

Also haben wir zur Bestimmung von c_h die Rekurrenz-Gleichung

$$c_{h+2} = c_{h+1} + c_h + 1 \quad \text{mit den Anfangs-Bedingungen } c_0 = 0 \text{ und } c_1 = 1$$

zu lösen. Das ist eine Rekurrenz-Gleichung, die wir, allerdings mit leicht veränderten Anfangs-Bedingungen, bereits im dritten Kapitel gelöst haben. Sie können leicht nachrechnen, dass die Lösung dieser Rekurrenz-Gleichung wie folgt lautet:

$$c_h = \frac{1}{\sqrt{5}}\left(\lambda_1^{h+2} - \lambda_2^{h+2}\right) - 1 \quad \text{mit}$$

$$\lambda_1 = \frac{1}{2}(1+\sqrt{5}) \approx 1.62 \quad \text{und} \quad \lambda_2 = \frac{1}{2}(1-\sqrt{5}) \approx -0.62.$$

Da $|\lambda_2| < 1$ ist, spielt der Wert λ_2^{h+2} für große Werte von h praktisch keine Rolle und die minimale Zahl n der Schlüssel in einem Baum der Höhe h ist durch

$$n \approx \frac{1}{\sqrt{5}}\lambda_1^{h+2} - 1$$

gegeben. Um diese Gleichung nach h aufzulösen, bilden wir auf beiden Seiten den Logarithmus zur Basis 2. Dann erhalten wir

$$\log_2(n+1) = (h+2) \cdot \log_2(\lambda_1) - \tfrac{1}{2} \cdot \log_2(5)$$

Daraus folgt nach Addition von $\frac{1}{2} \cdot \log_2(5)$

$$\log_2(n+1) + \tfrac{1}{2} \cdot \log_2(5) = (h+2) \cdot \log_2(\lambda_1)$$

Jetzt teilen wir durch $\log_2(\lambda_1)$. Dann erhalten wir

$$\frac{\log_2(n+1) + \frac{1}{2} \cdot \log_2(5)}{\log_2(\lambda_1)} = h + 2$$

Lösen wir diese Gleichung nach h auf, so haben wir für große n das Ergebnis

$$\begin{aligned} h &= \frac{\log_2(n+1) + \frac{1}{2} \cdot \log_2(5)}{\log_2(\lambda_1)} - 2 \\ &= \frac{1}{\log_2(\lambda_1)} \cdot \log_2(n) + \mathcal{O}(1) \\ &\approx 1,44 \cdot \log_2(n) + \mathcal{O}(1) \end{aligned}$$

gewonnen. Die Größe h gibt aber die Zahl der Vergleiche an, die wir im ungünstigsten Fall bei einem Aufruf von *find* in einem AVL-Baum mit n Schlüsseln durchführen müssen. Wir sehen also, dass bei einem AVL-Baum auch im schlechtesten Fall die Komplexität logarithmisch bleibt. Abbildung 7.17 zeigt einen AVL-Baum der Höhe 6, für den das Verhältnis von Höhe zur Anzahl der Knoten maximal wird. Wie man sieht ist auch dieser Baum noch sehr weit weg von dem zur Liste entarteten Baum aus der Abbildung 7.6.

7.3.3 Further Reading

In practice, *red-black trees* are slightly faster than *Avl* trees. Similar to *Avl* trees, a red-black is an ordered binary tree that is approximately balanced. Nodes are either black or red. The children of a red tree have to be black. In order to keep red-black trees approximately balanced, a *relaxed height* of a tree is defined. Red nodes do not contribute to the relaxed height of a tree. The left and right subtree of a red-black tree are required to have the same relaxed height. A detailed and very readable exposition of red-black trees is given in [SW11b].

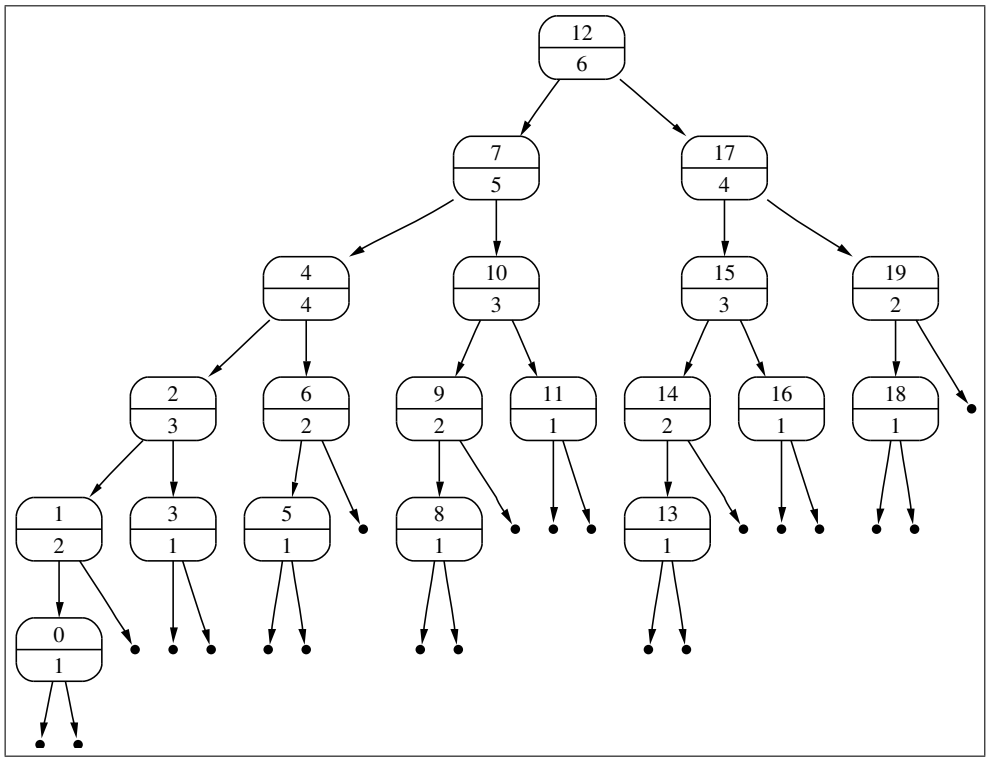

Figure 7.17: Ein AVL-Baum mit dem ungünstigsten Verhältnis von Höhe zur Anzahl an Knoten

7.4 Tries

In der Praxis kommt es häufig vor, dass die Schlüssel des ADT *Map* Strings sind. In dem einführenden Beispiel des elektronischen Telefon-Buchs ist dies der Fall. Es gibt eine Form von Such-Bäumen, die auf diese Situation besonders angepaßt ist. Diese Such-Bäume haben den Namen *Tries*. Dieses Wort ist von dem Englischen Wort *retrieval* abgeleitet. Damit man *Tries* und *Trees* unterscheiden kann, wird *Trie* so ausgesprochen, dass es sich mit dem Englischen Wort *pie* reimt. Diese Datenstruktur wurde 1959 von René de la Briandais [dlB59] vorgeschlagen.

Die Grundidee bei der Datenstruktur *Trie* ist ein Baum, an dem jeder Knoten nicht nur zwei Nachfolger hat, wie das bei binären Bäumen der Fall ist, sondern statt dessen potentiell für jeden Buchstaben des Alphabets einen Ast besitzt. Um Tries definieren zu können, nehmen wir zunächst an, dass folgendes gegeben ist:

1. Σ ist eine endliche Menge, deren Elemente wir als *Buchstaben* bezeichnen. Σ selbst heißt das *Alphabet*.

2. Σ^* bezeichnet die Menge der *Wörter* (engl. *strings*), die wir aus den Buchstaben des Alphabets bilden können. Mathematisch können wir Wörter als Listen von Buchstaben auffassen. Ist $w \in \Sigma^*$ so schreiben wir $w = cr$, falls c der erste Buchstabe von w ist und r das Wort ist, das durch Löschen des ersten Buchstabens aus w entsteht.

3. ε bezeichnet das leere Wort.

4. *Value* ist eine Menge von *Werten*.

Die Menge $\mathbb{T}$ der Tries definieren wir nun induktiv mit Hilfe des Konstruktors

$$\texttt{Node} : \mathit{Value} \times \mathit{List}(\Sigma) \times \mathit{List}(\mathbb{T}) \to \mathbb{T}.$$

Die induktive Definition besteht nur aus einer einzigen Klausel: Falls

1. $v \in \mathit{Value} \cup \{\Omega\}$
2. $C = [c_1, \cdots, c_n] \in \mathit{List}(\Sigma)$ eine Liste von Buchstaben der Länge n ist,
3. $T = [t_1, \cdots, t_n] \in \mathit{List}(\mathbb{T})$ eine Liste von Tries der selben Länge n ist,

dann gilt

$$\texttt{Node}(v, C, T) \in \mathbb{T}.$$

Als erstes fragen Sie sich vermutlich, wo bei dieser induktiven Definition der Induktions-Anfang ist. Der Induktions-Anfang ist der Fall $n = 0$, denn dann sind die Listen L und T leer. Als nächstes überlegen wir uns, welche Funktion von dem Trie

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]) \in \mathbb{T}$$

dargestellt wird. Wir beantworten diese Frage, indem wir rekursive Gleichungen für die Methode

$$\mathit{find} : \mathbb{T} \times \Sigma^* \to \mathit{Value} \cup \{\Omega\}$$

angeben. Wir werden den Ausdruck $\texttt{Node}(v, L, T).\mathit{find}(s)$ durch Induktion über den String s definieren:

1. $\texttt{Node}(v, C, T).\mathit{find}(\varepsilon) = v$.

   Der dem leeren String zugeordnete Wert wird also unmittelbar an der Wurzel des Tries abgespeichert.

2. $$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{find}(cr) = \begin{cases} t_1.\mathit{find}(r) & \text{falls} \quad c = c_1; \\ \vdots & \\ t_i.\mathit{find}(r) & \text{falls} \quad c = c_i; \\ \vdots & \\ t_n.\mathit{find}(r) & \text{falls} \quad c = c_n; \\ \Omega & \text{falls} \quad c \notin \{c_1, \cdots, c_n\}. \end{cases}$$

   Der Trie $\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$ enthält also genau dann einen Wert zu dem Schlüssel cr, wenn es in der Liste $[c_1, \cdots, c_n]$ eine Position i gibt, so dass der Buchstabe c mit dem Buchstaben c_i übereinstimmt wenn außerdem der Trie t_i einen Wert zu dem Schlüssel r enthält.

Zum besseren Verständnis wollen wir Tries graphisch als Bäume darstellen. Nun ist es nicht sinnvoll, die Knoten dieser Bäume mit langen Listen zu beschriften. Wir behelfen uns mit einem Trick. Um einen Knoten der Form

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$$

darzustellen, zeichnen wir einen Kreis, den wir durch einen horizontalen Strich in der Mitte aufteilen. Falls v von Ω verschieden ist, schreiben wir den Wert v in die untere Hälfte des Kreises. Das, was wir über dem Strich schreiben, hängt von dem Vater des jeweiligen Knotens ab. Wie genau es vom Vater abhängt, sehen wir gleich. Der Knoten selber hat n Kinder. Diese n Kinder sind die Wurzeln der Bäume, die die Tries t_1, $\cdots$, t_n darstellen. Außerdem markieren wir die diese Knoten darstellenden Kreise in den oberen Hälfte mit den Buchstaben c_1, $\cdots$, c_n.

Zur Verdeutlichung geben wir ein Beispiel in Abbildung 7.18 auf Seite 114. Die Funktion, die hier dargestellt wird, läßt sich wie folgt als binäre Relation schreiben:

$$\big\{\langle \text{“Stahl”}, 1\rangle, \langle \text{“Stolz”}, 2\rangle, \langle \text{“Stoeger”}, 3\rangle, \langle \text{“Salz”}, 4\rangle, \langle \text{“Schulz”}, 5\rangle,$$

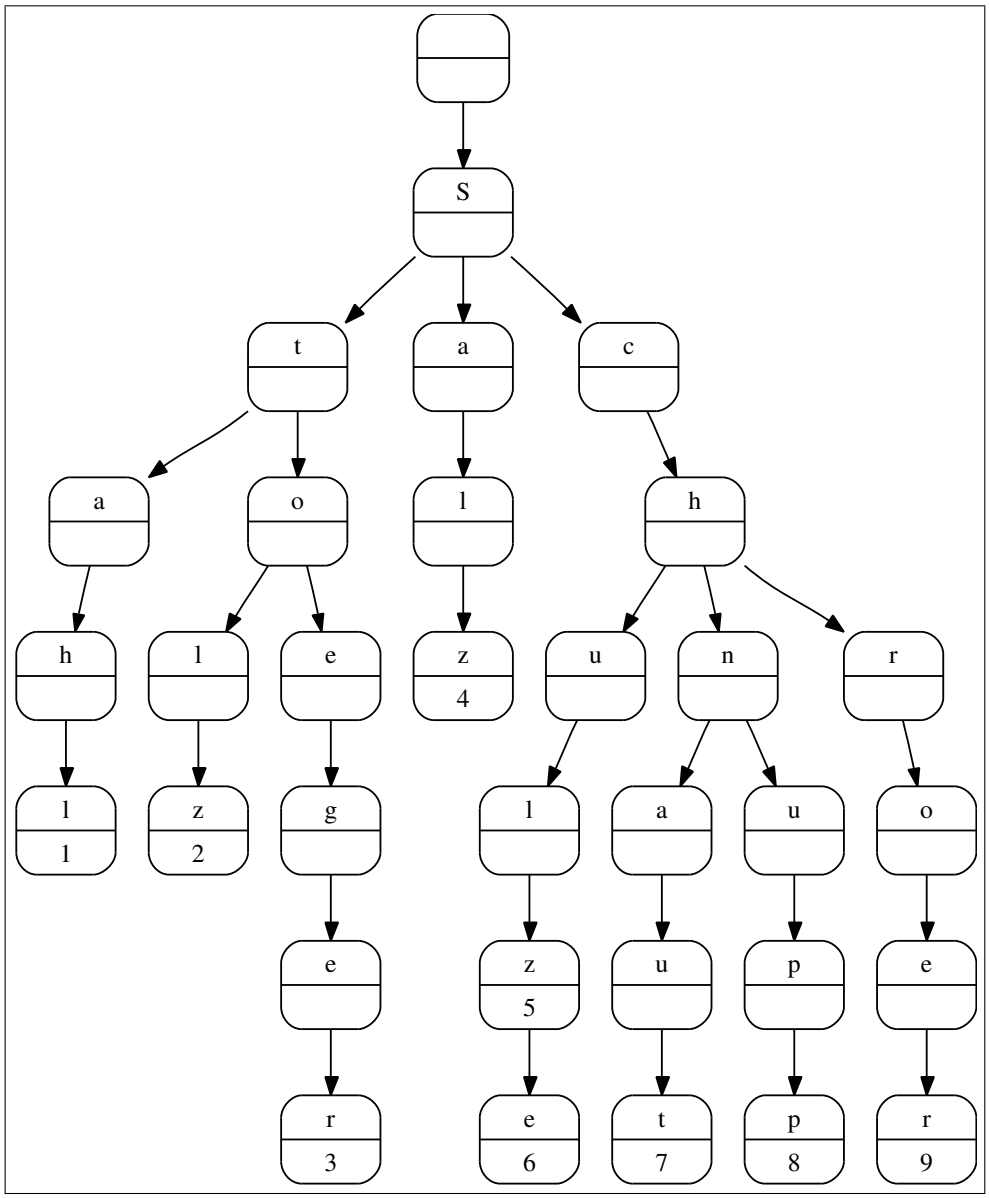

Figure 7.18: Ein Beispiel Trie

$$\langle \text{“Schulze”}, 6\rangle, \langle \text{“Schnaut”}, 7\rangle, \langle \text{“Schnupp”}, 8\rangle, \langle \text{“Schroer”}, 9\rangle\}.$$

Der Wurzel-Knoten ist hier leer, denn dieser Knoten hat keinen Vater-Knoten, von dem er eine Markierung erben könnte. Diesem Knoten entspricht der Term

$$\texttt{Node}(\Omega, [\text{‘S’}], [t]).$$

Dabei bezeichnet t den Trie, dessen Wurzel mit dem Buchstaben ‘S’ markiert ist. Diesen Trie können wir seinerseits durch den Term

$$\texttt{Node}(\Omega, [\text{‘t’}, \text{‘a’}, \text{‘c’}], [t_1, t_2, t_3])$$

darstellen. Daher hat dieser Knoten drei Söhne, die mit den Buchstaben ‘t’, ‘a’ und ‘c’ markiert sind.

7.4.1 Einfügen in Tries

Wir stellen nun bedingte Gleichungen auf, mit denen wir das Einfügen eines Schlüssels mit einem zugehörigen Wert beschreiben können. Bezeichnen wir die Methode für das Einfügen mit *insert*(), so hat diese Methode die Signatur

$$\mathit{insert} : \mathbb{T} \times \Sigma^* \times V \to \mathbb{T}.$$

Wir definieren den Wert von

$$\texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{insert}(s, v)$$

für ein Wort $w \in \Sigma^*$ und einen Wert $v \in V$ durch Induktion nach der Länge des Wortes w.

1. $\texttt{Node}(v_1, L, T).\mathit{insert}(\varepsilon, v_2) = \texttt{Node}(v_2, L, T)$,

   Einfügen eines Wertes mit dem leeren String als Schlüssel überschreibt einfach den an dem Wurzel-Knoten gespeicherten Wert.

2. $\texttt{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big).\mathit{insert}(c_i r, v_2) =$

   $$\texttt{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i.\mathit{insert}(r, v_2), \cdots, t_n]\big).$$

   Wenn in dem Trie $\texttt{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big)$ ein Wert v_2 zu dem Schlüssel cr eingefügt werden soll, und falls der Buchstabe c in der Liste $[c_1, \cdots, c_n]$ an der Stelle i vorkommt, wenn also gilt $c = c_i$, dann muß der Wert v_2 rekursiv in dem Trie t_i unter dem Schlüssel r eingefügt werden.

3. $c \notin \{c_1, \cdots, c_n\} \;\to\; \texttt{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big).\mathit{insert}(cr, v_2) =$

   $$\texttt{Node}\big(v_1, [c_1, \cdots, c_n, c], [t_1, \cdots, t_n, \texttt{Node}(\Omega, [], []).\mathit{insert}(r, v_2)]\big).$$

   Wenn in dem Trie $\texttt{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big)$ ein Wert v_2 zu dem Schlüssel cr eingefügt werden soll, und falls der Buchstabe c in der Liste $[c_1, \cdots, c_n]$ nicht vorkommt, dann wird zunächst ein Trie erzeugt, der die leere Abbildung repräsentiert. Dieser Trie hat die Form

   $$\texttt{Node}(\Omega, [], []).$$

   Anschließend wird in diesem Trie der Wert v_2 rekursiv unter dem Schlüssel r eingefügt. Zum Schluß hängen wir den Buchstaben c an die Liste $[c_1, \cdots, c_n]$ an und fügen den Trie

   $$\texttt{Node}(\Omega, [], []).\mathit{insert}(r, v_2)$$

   am Ende der Liste $[t_1, \cdots, t_n]$ ein.

7.4.2 Löschen in Tries

Als letztes stellen wir die bedingten Gleichungen auf, die das Löschen von Schlüsseln und den damit verknüpften Werten in einem Trie beschreiben. Um diese Gleichungen einfacher schreiben zu können, definieren wir zunächst eine Hilfs-Funktion

$$\mathit{isEmpty} : \mathbb{T} \to \mathbb{B},$$

so dass $t.\mathit{isEmpty}()$ genau dann `true` liefert, wenn der Trie t die leere Funktion darstellt. Wir definieren also:

1. $\texttt{Node}(\Omega, [], []).\mathit{isEmpty}() = \texttt{true}$,
2. $v \neq \Omega \to \texttt{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{isEmpty}() = \texttt{false}$,
3. $\texttt{Node}(\Omega, L, T).\mathit{isEmpty}() = \mathit{isEmptyList}(T)$

In der letzten Gleichung haben wir eine weitere Hilfs-Funktion benutzt, die wir noch definieren müssen. Die Funktion

$$\mathit{isEmptyList} : \mathit{List}(\mathbb{T}) \rightarrow \mathbb{B}$$

prüft für eine gegebene Liste von Tries, ob alle in der Liste vorhandenen Tries leer sind. Die Definition dieser Funktion erfolgt durch Induktion über die Länge der Liste.

1. $\mathit{isEmptyList}\big([]\big) = \texttt{true}$,
2. $\mathit{isEmptyList}\big([t] + r\big) = \big(t.\mathit{isEmpty}() \wedge \mathit{isEmptyList}(r)\big)$,

   denn alle Tries in der Liste $[t] + r$ sind leer, wenn einerseits t ein leerer Trie ist und wenn andererseits auch alle Tries in r leer sind.

Nun können wir die Methode

$$\mathit{delete} : \mathbb{T} \times \Sigma^* \rightarrow \mathbb{T}$$

spezifizieren: Wir definieren den Wert von

$$t.\mathit{delete}(w)$$

für einen Trie $t \in \mathbb{B}$ und ein Wort $w \in \Sigma^*$ durch Induktion nach der Länge des Wortes w.

1. $\texttt{Node}(v, L, T).\mathit{delete}(\varepsilon) = \texttt{Node}(\Omega, L, T)$,

   denn der Wert, der unter dem leeren String ε in einem Trie gespeichert wird, befindet sich unmittelbar an der Wurzel des Tries und kann dort sofort gelöscht werden.

2. $$\begin{array}{ll} t_i.\mathit{delete}(r).\mathit{isEmpty}() & \rightarrow \\ \texttt{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]).\mathit{delete}(c_i r) & = \\ \quad \texttt{Node}(v, [c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n], [t_1, \cdots, t_{i-1}, t_{i+1}, \cdots, t_n]). & \end{array}$$

   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. $$\begin{array}{ll} \neg t_i.\mathit{delete}(r).\mathit{isEmpty}() & \wedge \\ \mathit{delete}\big(\texttt{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]), c_i r\big) & = \\ \quad \texttt{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i.\mathit{delete}(r), \cdots, t_n]). & \end{array}$$

   Wenn der zu löschende String mit dem Buchstaben c_i anfängt, und wenn der Baum t_i, der durch das Löschen des Schlüssels r in dem i-ten Trie t_i entsteht nicht leer ist, dann löschen wir rekursiv in dem Baum t_i den Schlüssel r.

4. $c \notin C \rightarrow \texttt{Node}(v, C, T).\mathit{delete}(cr) = \texttt{Node}(v, C, T)$.

   Wenn der zu löschende String mit dem Buchstaben c anfängt und wenn der Buchstabe c gar kein Element der Buchstaben-Liste C des Tries ist, dann verändert das Löschen den Trie nicht.

7.4.3 Implementing Tries in SetlX

We proceed to discuss the implementation of tries. Figure 7.19 shows the outline of the class `trie`. This class supports three member variables. In order to understand these member variables, remember that a trie has the form

$$\texttt{Node}(v, C, T)$$

```
class map() {
    mValue := om;
    mChars := [];
    mTries := [];

  static {
    find    := procedure(s)    { ... };
    insert  := procedure(s, v) { ... };
    delete  := procedure(s)    { ... };
    isEmpty := procedure()     { ... };
  }
}
```

Figure 7.19: Outline of the class `trieMap`.

where v is the value stored at the root, C is the list of characters, and t is a list of tries. Therefore, the member variables have the following semantics:

1. `mValue` represent the value v stored at the root of this trie,
2. `mChars` represent the list C of characters. If there is a string cr such that the trie stores a value associated with this string, then the character c will be an element of the list C.
3. `mTries` represent the list of subtries T.

The methods `find`, `insert`, and `delete` are inherited from the abstract data type `map`. The method `isEmpty` is an auxiliary method that is needed in the implementation of the method `delete`. This method checks whether the given trie corresponds to the empty map. The implementation of all these methods is given below.

```
find := procedure(s) {
    match (s) {
        case ""   : return mValue;
        case [c|r]: for (i in [1 .. #mChars]) {
                        if (mChars[i] == c) {
                            return mTries[i].find(r);
                        }
                    }
                    return;  // not found
    }
};
```

Figure 7.20: Implementation of `find` for tries.

The method `find` takes a string s as its sole argument and checks whether the given trie contains a value associated with the string s. Essentially, the are two cases:

1. If s is the empty string, the value associated with s is stored in the member variable `mValue` at the root of this trie.

2. Otherwise, s can be written as $s = cr$ where c is the first character of s while r consists of the remaining characters. In order to check whether the trie has a value stored for s we first have to check whether there is an index i such that `mChars[`i`]` is equal to c. If this is the case, the subtrie `mTries[`i`]` contains the value associated with s. Then, we have to invoke `find` recursively on this subtrie.

   If the loop in line 4 does not find c in the list `mChars`, then the method `find` will just return `om` in line 9.

```
1  insert := procedure(s, v) {
2      match (s) {
3          case ""   : this.mValue := v;
4          case [c|r]: for (i in [1 .. #mChars]) {
5                          if (mChars[i] == c) {
6                              t := mTries[i];
7                              t.insert(r,v);
8                              this.mTries[i] := t;
9                              return;
10                         }
11                     }
12                     newTrie := trieMap();
13                     newTrie.insert(r, v);
14                     this.mChars += [ c ];
15                     this.mTries += [ newTrie ];
16     }
17 };
```

Figure 7.21: Implementation of `insert` for tries.

The method `insert` takes a string s and an associated value v that is to be inserted into the given trie. The implementation of `insert` works somewhat similar to the implementation of `find`.

1. If the string s is empty, then the value v has to be positioned at the root of this trie. Hence we just set `mValue` to v.

2. Otherwise, s can be written as cr where c is the first character of s while r consists of the remaining characters. In this case, we need to check whether the list `mChars` already contains the character c or not.

   (a) If c is the i-th character of `mChars`, then we have to insert the value v in the trie `mTries[`i`]`. However, this is a little tricky to do. First, we retrieve the subtrie `mTries[`i`]` and store this trie into the variable t. Next, we can insert the value v into the trie t using the rest r of the string s as the key. Finally, we have to set `mTries[`i`]` to t. At this point, you might wonder why we couldn't have just used the statement

   ```
   this.mTries[i].insert(r,v);
   ```

   to achieve the same effect. Unfortunately, this does not work because the expression `this.mTries[i]` will create a temporary value and inserting v into this temporary value will not change the original list `mTries`.

   (b) If c does not occur in `mChars`, things are straightforward: We create a new empty trie and insert v into this trie. Next, we append the character c to

`mChars` and simultaneously append the newly created trie that contains v to `mTries`.

```
1  delete := procedure(s) {
2      match (s) {
3          case ""   : this.mValue := om;
4          case [c|r]:
5              for (i in [1 .. #mChars]) {
6                   if (mChars[i] == c) {
7                       t := mTries[i];
8                       t.delete(r);
9                       this.mTries[i] := t;
10                      if (this.mTries[i].isEmpty()) {
11                          this.mChars := removeIthElement(mChars, i);
12                          this.mTries := removeIthElement(mTries, i);
13                      }
14                      return;
15                  }
16              }
17     }
18 };
```

Figure 7.22: Implementation of `delete` for tries.

The method `delete` takes a string and, provided there is a value associated with s, this value is deleted,

1. If the string s is empty, the value associated with s is stored at the root of this trie. Hence, this value is set to `om`.

2. Otherwise, s can we written as cr where c is the first character of s while r consists of the remaining characters. In this case, we need to check whether the list `mChars` contains the character c or not.

   If c is the i-th character of `mChars`, then we have to delete the value associated with r in the trie `mTries[`i`]`. Again, this is tricky to do. First, we retrieve the subtrie `mTries[`i`]` and store this trie into the variable t. Next, the value associated with r is deleted in t and, finally, t is written to `mTries[`i`]`.

   After the deletion, the subtrie `mTries[`i`]` might well be empty. In this case, we remove the i-th character form `mChars` and also remove the i-th trie from the list `mTries`. This is done with the help of the function `removeIthElement`, which is shown in Figure 7.24.

```
1  isEmpty := procedure() {
2      return mValue == om && mChars == [];
3  };
```

Figure 7.23: Implementation of `isEmpty` for tries.

In order to check whether a given trie is empty we have to check that no value is stored at the root and that the list `mChars` is empty, since then the list `mTries` will also be empty.

```
1  removeIthElement := procedure(l, i) {
2      return l[1 .. i-1] + l[i+1 .. #l];
3  };
```

Figure 7.24: The function to remove the i-th element from a list.

Finally, the implementation of `removeIthElement`, which is shown in Figure 7.24, is straightforward.

Exercise 16: **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. $\mathit{Nil} \in \mathrm{BT}$.
2. $\mathit{Bin}(v, l, r) \in \mathrm{BT}$ falls
   (a) $v \in \mathit{Value} \cup \{\Omega\}$.
   (b) $l, r \in \mathrm{BT}$.

Die Semantik legen wir fest, indem wir eine Funktion

$$\mathit{find} : \mathrm{BT} \times \mathbb{N} \to \mathit{Value} \cup \{\Omega\}$$

definieren. Für einen binären Trie b und eine natürliche Zahl n gibt $\mathit{find}(b, n)$ den Wert zurück, der unter dem Schlüssel n in dem binären Trie b gespeichert ist. Falls in dem binären Trie b unter dem Schlüssel n kein Wert gespeichert ist, wird Ω zurück gegeben. Formal definieren wir den Wert von $\mathit{find}(b, n)$ durch Induktion nach dem Aufbau von b. Im Induktions-Schritt ist eine Neben-Induktion nach n erforderlich.

1. $\mathit{find}(\mathit{Nil}, n) = \Omega$,
   denn in dem leeren binären Trie finden wir keine Werte.
2. $\mathit{find}\big(\mathit{Bin}(v, l, r), 0\big) = v$,
   denn der Schlüssel 0 entspricht dem leeren String ε.
3. $n \neq 0 \to \mathit{find}\big(\mathit{Bin}(v, l, r), 2{\cdot}n\big) = \mathit{find}(l, n)$,
   denn wenn wir Zahlen im Binärsystem darstellen, so hat bei geraden Zahlen das letzte Bit den Wert 0 und die 0 soll dem linken Teilbaum entsprechen.
4. $\mathit{find}\big(\mathit{Bin}(v, l, r), 2{\cdot}n{+}1\big) = \mathit{find}(r, n)$,
   denn wenn wir Zahlen im Binärsystem darstellen, so hat bei ungeraden Zahlen das letzte Bit den Wert 1 und die 1 soll dem rechten Teilbaum entsprechen.

Bearbeiten Sie nun die beiden folgenden Teilaufgaben:

1. Stellen Sie Gleichungen auf, die das Einfügen und das Löschen in einem binären Trie beschreiben. Achten Sie beim Löschen darauf, dass binäre Tries der Form $\mathit{Bin}(\Omega, \mathit{Nil}, \mathit{Nil})$ zu Nil vereinfacht werden.
   Hinweis: Um die Gleichungen zur Spezifikation der Funktion *delete*() nicht zu komplex werden zu lassen ist es sinnvoll, eine Hilfsfunktion zur Vereinfachung von binären Tries zu definieren.
2. Implementieren Sie binäre Tries in SetlX.

Bemerkung: Binäre Tries werden auch als *digitale Suchbäume* bezeichnet.

7.5 Hash-Tabellen

Eine Abbildung

$$f : Key \to Value$$

kann dann sehr einfach implementiert werden, wenn

$$Key = \{1, 2, \cdots, n\}$$

gilt, denn dann reicht es aus, ein Feld der Größe n zu verwenden. Abbildung 7.25 zeigt eine Umsetzung dieser Idee.

```
1  class map(n) {
2      mArray := [1..n];
3    static {
4      find   := k |-> mArray[k];
5      insert := procedure(k, v) { this.mArray[k] := v;  };
6      delete := procedure(k)    { this.mArray[k] := om; };
7      f_str  := procedure()     { return str(mArray);   };
8    }
9  }
```

Figure 7.25: Implementing a map as an array.

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 $w = c_0 c_1 \cdots c_7$ durch eine Funktion

   $$code : \Sigma^* \to \mathbb{N}$$

   berechnen, die wie folgt definiert ist:

   $$code(c_0 c_1 \cdots c_7) = 1 + \sum_{i=0}^{7} ord(c_i) \cdot 27^i.$$

   Die Menge *code* bildet die Menge aller Wörter mit 8 Buchstaben bijektiv auf die Menge der Zahlen $\{1, \cdots, 27^8\}$ ab.

Leider hat diese naive Implementierung mehrere Probleme:

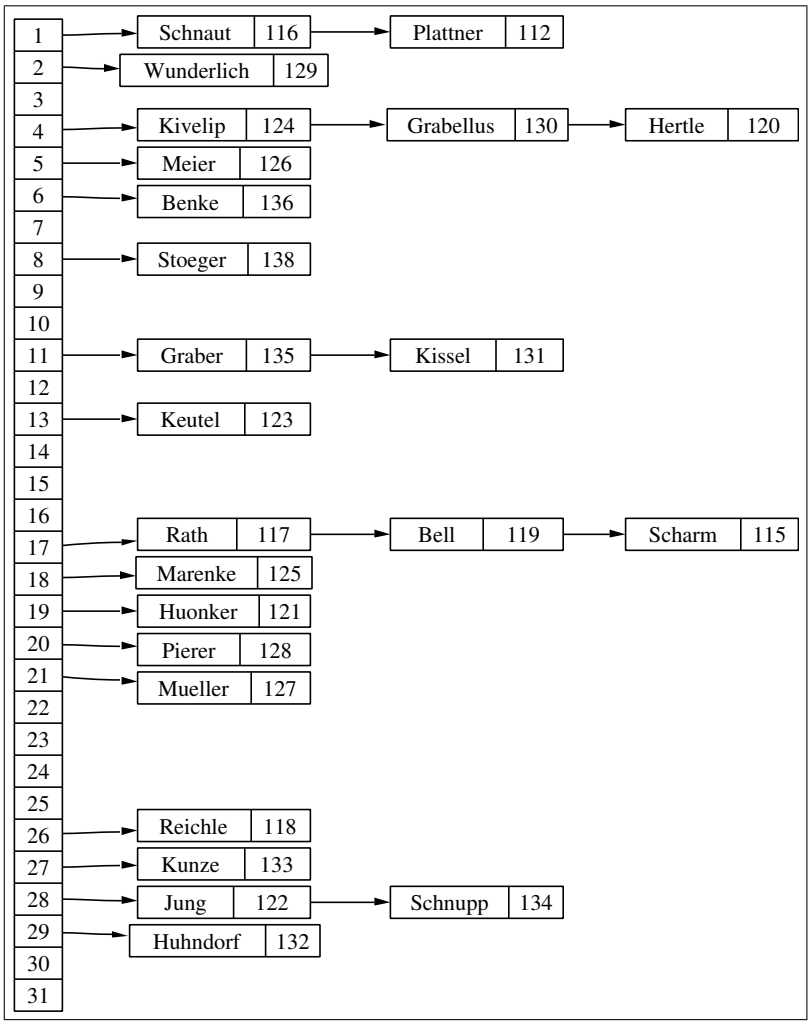

Figure 7.26: Eine Hash-Tabelle

1. Das Feld, das wir anlegen müssen, hat eine Größe von

   $$27^8 = 282\,429\,536\,481$$

   Einträgen. Selbst wenn jeder Eintrag nur die Größe zweier Maschinen-Worte hat und ein Maschinen-Wort aus 4 Byte besteht, so bräuchten wir etwas mehr als ein Terabyte um eine solche Tabelle anzulegen.

2. Falls zwei Namen sich erst nach dem 8-ten Buchstaben unterscheiden, können wir zwischen diesen Namen nicht mehr unterscheiden.

Wir können diese Probleme wir folgt lösen:

1. Wir ändern die Funktion `code` so ab, dass das Ergebnis immer kleiner-gleich einer vorgegebene Zahl `size` ist. Die Zahl `size` gibt dabei die Größe eines Feldes an und ist so klein, dass wir ein solches Feld bequem anlegen können.

   Eine einfache Möglichkeit, die Funktion *code* entsprechend abzuändern, besteht in folgender Implementierung:

   $$code(c_0c_1\cdots c_n) = \left(\sum_{i=0}^{n} ord(c_i)\cdot 27^i\right) \;\%\; \texttt{size} + 1.$$

Um einen Überlauf zu vermeiden, können wir für $k = n, n-1, \cdots, 1, 0$ die Teilsummen s_k wie folgt induktiv definieren:

(a) $s_n = ord(c_n)$

(b) $s_k = (ord(c_k) + s_{k+1} \cdot 27) \ \%$ `size`

Dann gilt $$s_0 + 1 = \left(\sum_{i=0}^{n} ord(c_i) \cdot 27^i \right) \ \% \texttt{size} + 1.$$

2. In dem Feld der Größe `size` speichern wir nun nicht mehr die Werte, sondern statt dessen Listen von Paaren aus Schlüsseln und Werten. Dies ist notwendig, denn wir können nicht verhindern, dass die Funktion `code`() für zwei verschiedene Schlüssel den selben Index liefert.

Abbildung 7.26 auf Seite 122 zeigt, wie ein Feld, in dem Listen von Paaren abgebildet sind, aussehen kann. Ein solches Feld bezeichnen wir als Hash-Tabelle. Wir diskutieren nun die Implementierung dieser Idee in SETLX.

```
1  class hashTable(n) {
2      mSize    := n;
3      mEntries := 0;  // number of entries
4      mArray   := [ {} : i in [1 .. mSize] ];
5      mAlpha   := 2;  // load factor
6
7      static {
8          sOrd    := { [ char(i), i ] : i in [ 0 .. 127 ] };
9          sPrimes := [ 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039,
10                      4093, 8191, 16381, 32749, 65521, 131071,
11                      262139, 524287, 1048573, 2097143, 4194301,
12                      8388593, 16777213, 33554393, 67108859,
13                      134217689, 268435399, 536870909, 1073741789,
14                      2147483647
15                    ];
16         hashCode := procedure(s)          { ... };
17         find     := procedure(key)        { ... };
18         insert   := procedure(key, value) { ... };
19         rehash   := procedure()           { ... };
20         delete   := procedure(key)        { ... };
21     }
22 }
```

Figure 7.27: Outline of the class `hashTable`.

Figure 7.27 shows the outline of the class `hashTable`.

1. The constructor is called with one argument. This argument n is the initial size of the array storing the different key-value lists. The constructor constructs an empty hash table with the given capacity.

2. `mSize` is the actual size of the array that stores the different key-value lists. Although this variable is initialized as n, it can later be increased. This happens if the hash table becomes overcrowded.

3. `mEntries` is the number key-value pairs that are stored in this hash map. Since, initially, this map is entry, `mEntries` is initialized as 0.

4. `mArray` is the array containing the list of key value pairs.

   In our implementation, the key-value pairs are not stored in a list but, instead, they are stored in a set. Since every key occurs at most once, this set can be interpreted as a functional relation. Therefore, looking up a key is more efficient than it would be if we had used a list. Although we actually use a relation instead of a list, we will still call the relation the *list of key-value pairs*.

   As the hash map is initially empty, all entries of `mArray` are initialized as empty sets.

5. `mAlpha` is the *load factor* of our hash table. If at any point in time, we have

   $$\texttt{mEntries} > \texttt{mAlpha} \cdot \texttt{mSize},$$

   then we consider our hash table to be *overcrowded*. In that case, we would increase the size of the array `mArray`. To determine the best value for `mAlpha`, we have to make a tradeoff: If `mAlpha` is too big, many entries in the array `mArray` would be empty and thus we would waste space. On the other hand, if `mAlpha` is too small, the key-value lists would become very long and hence it would take too much time to search for a given key in one of these lists.

6. Our implementation maintains two static variables.

   (a) `sOrd` is a functional relation mapping characters to Ascii codes. This relation is needed for the efficient computation of the method `hashCode` discussed below.

   In `SetlX` there is no function that returns the Ascii code of a given character. Fortunately, it is easy to implement this function as a binary relation via the function `char`(i). Given a number $i \in \{0, \cdots, 127\}$, the function `char`(i) returns the character that has Ascii code i. The relation `sOrd` is the inverse of the function `char`.

   (b) `sPrimes` is a list of prime numbers. Roughly, these prime numbers double in size. The reason is that performance of a hash table is best if the size of `mArray` is a prime number. When `mArray` get overcrowded, the idea is to, more or less, double the size of `mArray`. To achieve this, the variable `sPrimes` is needed.

Next, we discuss the implementation of the various methods.

```
1       hashCode := procedure(s) {
2           return hashCodeAux(s) + 1;
3       };
4       hashCodeAux := procedure(s) {
5           if (s == "") {
6               return 0;
7           }
8           return (sOrd[s[1]] + 128 * hashCodeAux(s[2..])) % mSize;
9       };
```

Figure 7.28: Implementation of the method `hashCode`.

Figure gives the implementation of the method `hashCode`.

1. The function `hashCode`(s) takes a string s and computes a hash code for this string. This hash code satisfies

$$\texttt{hashCode}(s) \in \{1, \cdots, \texttt{mSize}\}.$$

Therefore, the hash code can be used to index into `mArray`. The implementation of `hashCode` works by calling $\texttt{hashCodeAux}(s)$. As the values returned by $\texttt{hashCodeAux}(s)$ are elements of the set

$$\texttt{hashCode}(s, n) \in \{0, \cdots, \texttt{mSize} - 1\}$$

we have to add 1 so that the hash code is an element of

$$\texttt{hashCode}(s, n) \in \{1, \cdots, \texttt{mSize}\}.$$

2. The function $\texttt{hashCodeAux}(s)$ is defined by induction on the string s. If the string s has length m we have

$$\texttt{hashCodeAux}(s, n) := \left(\sum_{i=1}^{m} \texttt{ord}(s[i]) \cdot 128^{i-1}\right) \,\%\, \texttt{mSize}.$$

Here, given an Ascii character c, the expression $\texttt{ord}(c)$ computes the Ascii code of c.

```
1       find := procedure(key) {
2            index := hashCode(key);
3            aList := mArray[index];
4            return aList[key];
5       };
```

Figure 7.29: Implementation of `find`.

Figure 7.29 show the implementation of the method `find`.

1. First, we compute the index of the key-value list that is used to store the given `key`.
2. Next, we retrieve this key-value list from the array `mArray`.
3. Finally, we look up the information stored under the given `key` in this key-value list. Remember, that the key-value list is not an actual list but rather a binary relation. We can use the notation `aList[key]` to retrieve the value associated with the given key.

Figure 7.30 shows the implementation of the method `insert`. The implementation works as follows.

1. First, we check whether our has table is already overcrowded. In this case, we *rehash*, which means we roughly double the size of `mArray`. How the method `rehash` works in detail is explained later. After rehashing, the `key` is inserted via a recursive call.
2. If we don't have to rehash, we compute the index of the key-value list that has to store `mKey`, retrieve the associated key-value list, and finally associate the `value` with the given key. When inserting the given key-value pair into the key-value list there can be two cases.
   (a) The key-value list already stores information for the given `key`. In this case, the number of entries of the hash table is not changed.
   (b) If the given `key` is not yet present in the given key-value list, the number of entries needs to be incremented.

```
1       insert := procedure(key, value) {
2            if (mEntries > mSize * mAlpha) {
3                rehash();
4                insert(key, value);
5                return;
6            }
7            index      := hashCode(key);
8            aList      := mArray[index];
9            oldSz      := #aList;
10           aList[key] := value;
11           newSz      := #aList;
12           this.mArray[index] := aList;
13           if (newSz > oldSz) {
14               this.mEntries += 1;
15           }
16      };
```

Figure 7.30: Implementation of the method `insert`.

In order to distinguish these two cases, we compare the size of the key-value list before the insertion with the size after the insertion.

```
1       rehash := procedure() {
2            prime  := min({ p in sPrimes | p * mAlpha > mEntries });
3            bigMap := hashTable(prime);
4            for (aList in mArray) {
5                for ([k, v] in aList) {
6                    bigMap.insert(k, v);
7                }
8            }
9            this.mSize  := prime;
10           this.mArray := bigMap.mArray;
11      };
```

Figure 7.31: Implementation of the method `rehash`.

Figure 7.31 show the implementation of the method `rehash`(). This method is called if the hash table becomes overcrowded. The idea is to roughly double the size of `mArray`. Theoretical considerations that are beyond the scope of this lecture show that it is beneficial if the size of `mArray` is a prime number. Hence, we look for the first prime number `prime` such that `prime` times the load factor `mAlpha` is bigger than the number of entries since this will assure that, on average, the number of entries in each key-value list is less than the load factor `mAlpha`. After we have determined `prime`, we proceed as follows:

1. We create a new empty hash table of size `prime`.
2. Next, we move the key-value pairs from the given has table to the new hash table.
3. Finally, the array stored in the new hash table is moved to the given hash table and the size is adjusted correspondingly.

```
12      delete := procedure(key) {
13          index      := hashCode(key);
14          aList      := mArray[index];
15          oldSz      := #aList;
16          aList[key] := om;
17          newSz      := #aList;
18          this.mArray[index] := aList;
19          if (newSz < oldSz) {
20              this.mEntries -= 1;
21          }
22      };
```

Figure 7.32: Die Funktion `delete`(*map*, *key*).

Finally, we discuss the implementation of the method `delete` that is shown in Figure 7.32. The implementation of this method is similar to the implementation of the method `insert`. The implementation makes use of the fact that in order to delete a key-value pair from a function relation in SetlX it is possible to assign the value `om` to the `key` that needs to be deleted. Note, that we have to be careful to maintain the number of entries since we do not know whether the list of key-value pairs has an entry for the given `key`.

However, there is one crucial difference compared to the implementation of `insert`. We do not rehash the hash table if the number of entries falls under certain thresh hold. Although this could be done and there are implementations of hash tables that readjust the size of the hash table if the hash table gets underpopulated, we don't do so here because often a table will grow again after it has shrunk and in that case rehashing would be counter productive.

Falls wir in unserer Implementierung tatsächlich mit Listen und nicht mit Mengen arbeiten würden, dann könnte die Komplexität der Methoden *find*, *insert* und *delete* im ungünstigsten Fall linear mit der Anzahl der Einträge in der Hash-Tabelle wachsen. Dieser Fall tritt dann auf, wenn die Funktion `hashTable`(k, n) für alle Schlüssel k den selben Wert berechnet. Dieser Fall ist allerdings sehr unwahrscheinlich. Der Normalfall ist der, dass alle Listen etwa gleich lang sind. Die durchschnittliche Länge einer Liste ist dann

$$\alpha = \frac{\texttt{count}}{\texttt{size}}.$$

Hierbei ist `count` die Gesamtzahl der Einträge in der Tabelle und `size` gibt die Größe der Tabelle an. Das Verhältnis α dieser beiden Zahlen ist genau der *Auslastungs-Faktor* der Hash-Tabelle. In der Praxis zeigt sich, dass α kleiner als 4 sein sollte. In *Java* gibt es die Klasse *HashMap*, die Abbildungen als Hash-Tabellen implementiert. Dort hat der per Default eingestellte maximale Auslastungs-Faktor sogar nur den Wert `0.75`.

7.5.1 Further Reading

In this section, we have discussed hash tables only briefly. The reason is that, although hash tables are very important in practice, a thorough treatment requires quite a lot of mathematics, see for example the third volume of Donald Knuth's "The Art of Computer Programming" [Knu98]. For this reason, the design of a hash function is best left for experts. In practice, hash tables are quite a bit faster than *AVL*-trees or *red-black* trees. However, this is only true if the hash function that is used is able to spread the keys uniformly. If this assumption is violated,

the use of a hash table can lead to serious performance bugs[1]. If, instead, a good implementation of red-black-trees is used, the program might be slower in general but are certain to be protected from the ugly surprises that can result from a poor hash function. My advice for the reader therefore is to use hashing only if the performance is really critical and you are sure that your hash function is distributing the keys nicely.

7.6 Applications

Both `C++` and *Java* supply maps. In `C++`, maps are part of the standard template library, while *Java* offers the interface `Map` that is implemented both by the class `TreeMap` and the class `HashMap`. Furthermore, all modern script languages provide maps. For example, in *Perl* [WS92], maps are known as *associative arrays*, in *Lua* [Ier06, IdFF96] maps are called tables, and in *Python* [vR95, Lut09] map are called *dictionaries*.

Later, when we discuss Dijkstra's algorithm for finding the shortest path in a graph we will see an application of maps.

[1] Da habe ich schon Pferde kotzen sehen, direkt vor der Apotheke.

Chapter 8

Priority Queues

Um den Begriff der *Prioritäts-Warteschlange* zu verstehen, betrachten wir zunächst den Begriff der *Warteschlange*. Dort werden Daten hinten eingefügt und vorne werden Daten entnommen. Das führt dazu, dass Daten in der selben Reihenfolge entnommen werden, wie sie eingefügt werden. Anschaulich ist das so wie bei der Warteschlange vor einer Kino-Kasse, wo die Leute in der Reihenfolge bedient werden, in der sie sich anstellen. Bei einer Prioritäts-Warteschlange haben die Daten zusätzlich Prioritäten. Es wird immer das Datum entnommen, was die höchste Priorität hat. Anschaulich ist das so wie im Wartezimmer eines Zahnarztes. Wenn Sie schon eine Stunde gewartet haben und dann ein Privat-Patient aufkreuzt, dann müssen Sie halt noch eine Stunde warten, weil der Privat-Patient eine höhere Priorität hat.

Prioritäts-Warteschlangen spielen in vielen Bereichen der Informatik eine wichtige Rolle. Wir werden Prioritäts-Warteschlangen später sowohl in dem Kapitel über Daten-Kompression als auch bei der Implementierung des Algorithmus zur Bestimmung kürzester Wege in einem Graphen einsetzen. Daneben werden Prioritäts-Warteschlangen unter anderem in Simulations-Systemen und beim Scheduling von Prozessen in Betriebs-Systemen eingesetzt.

8.1 Definition des ADT *PrioQueue*

Wir versuchen den Begriff der Prioritäts-Warteschlange jetzt formal durch Definition eines abstrakten Daten-Typs zu fassen. Wir geben hier eine eingeschränkte Definition von Prioritäts-Warteschlangen, die nur die Funktionen enthält, die wir später für den Algorithmus von Dijkstra benötigen.

Definition 20 (Prioritäts-Warteschlange)
Wir definieren den abstrakten Daten-Typ der Prioritäts-Warteschlange *wie folgt:*

1. *Als Namen wählen wir PrioQueue.*
2. *Die Menge der Typ-Parameter ist*
   $$\{Priority, Value\}.$$
   Dabei muß auf der Menge Priority eine totale Quasi-Ordnung $<$ existieren, so dass wir die Prioritäten verschiedener Elemente vergleichen können.
3. *Die Menge der Funktions-Zeichen ist*
   $$\{prioQueue, insert, remove, top\}.$$
4. *Die Typ-Spezifikationen der Funktions-Zeichen sind gegeben durch:*

(a) *prioQueue* : *PrioQueue*
Der Aufruf "prioQueue()" erzeugt eine leere Prioritäts-Warteschlange.

(b) *top* : *PrioQueue* $\rightarrow$ (*Priority* $\times$ *Value*) $\cup \{\Omega\}$
Der Aufruf Q.top() liefert ein Paar $\langle p, v\rangle$. Dabei ist v ein Element aus Q, das eine maximale Priorität hat. p ist die Priorität des Elements v.

(c) *insert* : *PrioQueue* $\times$ *Priority* $\times$ *Value* $\rightarrow$ *PrioQueue*
Der Aufruf Q.insert(p, v) fügt das Element v mit der Priorität p in die Prioritäts-Warteschlange Q ein.

(d) *remove* : *PrioQueue* $\rightarrow$ *PrioQueue*
Der Aufruf Q.remove() entfernt aus der Prioritäts-Warteschlange Q das Element, das durch den Ausdruck Q.top() berechnet wird.

5. *Bevor wir das Verhalten der einzelnen Methoden axiomatisch definieren, müssen wir noch festlegen, was wir unter den* Prioritäten *verstehen wollen, die den einzelnen Elementen aus Value zugeordnet sind. Wir nehmen an, dass die Prioritäten Elemente einer Menge Priority sind und dass auf der Menge Priority eine totale Quasi-Ordnung $\leq$ existiert. Falls dann $p_1 < p_2$ ist, sagen wir, dass p_1 eine höhere Priorität als p_2 hat. Dies erscheint im ersten Moment vielleicht paradox. Es wird aber später verständlich, wenn wir den Algorithmus zur Berechnung kürzester Wege von Dijkstra diskutieren. Dort sind die Prioritäten Entfernungen im Graphen und die Priorität eines Knotens ist um so höher, je näher der Knoten zu einem als* Startknoten *ausgezeichneten Knoten ist.*

   Wir spezifizieren das Verhalten der Methoden nun dadurch, dass wir eine einfache Referenz-Implementierung *des ADT PrioQueue angeben und dann fordern, dass sich eine Implementierung des ADT PrioQueue genauso verhält wie unsere Referenz-Implementierung. Bei unserer Referenz-Implementierung stellen wir eine Prioritäts-Warteschlange durch eine Menge von Paaren von Prioritäten und Elementen dar. Für solche Mengen definieren wir unserer Methoden wie folgt.*

   (a) *prioQueue*$() = \{\}$,
   der Konstruktor erzeugt also eine leere Prioritäts-Warteschlange, die als leere Menge dargestellt wird.

   (b) Q.*insert*$(p, v) = Q \cup \{\langle p, v\rangle\}$,
   Um ein Element v mit einer Priorität p in die Prioritäts-Warteschlange Q einzufügen, reicht es aus, das Paar $\langle p, v\rangle$ zu der Menge Q hinzuzufügen.

   (c) *Wenn Q leer ist, dann ist Q.top() undefiniert:*
   $$Q = \{\} \;\rightarrow\; Q.\mathit{top}() = \Omega.$$

   (d) *Wenn Q nicht leer ist, wenn es also ein Paar $\langle p_1, v_1\rangle$ in Q gibt, dann liefert Q.top() ein Paar $\langle p_2, v\rangle$ aus der Menge Q, so dass die Priorität p_2 minimal wird. Dann gilt also für alle $\langle p_1, v_1\rangle \in Q$, dass $p_2 \leq p_1$ ist. Formal können wir schreiben:*
   $$\langle p_1, v_1\rangle \in Q \;\wedge\; Q.\mathit{top}() = \langle p_2, v_2\rangle \;\rightarrow\; p_2 \leq p_1 \;\wedge\; \langle p_2, v_2\rangle \in Q.$$

   (e) *Falls Q leer ist, dann ändert remove() nichts daran:*
   $$Q = \{\} \rightarrow Q.\mathit{remove}() = Q.$$

   (f) *Sonst entfernt Q.remove() das Paar, dass von Q.top() berechnet wird:*
   $$Q \neq \{\} \rightarrow Q.\mathit{remove}() = Q \backslash \big\{Q.\mathit{top}()\big\}.$$

Wir können den abstrakten Daten-Typ *PrioQueue* dadurch implementieren, dass wir eine Prioritäts-Warteschlange durch eine Liste realisieren, in der die Elemente aufsteigend geordnet sind. Die einzelnen Operationen werden dann wie folgt implementiert:

1. *prioQueue*() erzeugt eine leere Liste.
2. $Q.insert(d)$ kann durch die Prozedur `insert` implementiert werden, die wir beim "*Sortieren durch Einfügen*" entwickelt haben.
3. $Q.top()$ gibt das erste Element der Liste zurück.
4. $Q.remove()$ entfernt das erste Element der Liste.

Bei dieser Implementierung ist die Komplexität der Operation *insert*() 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*() den Wert $\mathcal{O}\big(\log(n)\big)$ hat. Dazu führen wir die Daten-Struktur eines *Heaps* ein.

8.2 Die Daten-Struktur *Heap*

Wir definieren die Menge *Heap*[1] induktiv als Teilmenge der Menge $\mathcal{B}$ der binären Bäume. Dazu definieren wir zunächst für eine Priorität $p_1 \in \mathit{Priority}$ und einen binären Baum $b \in \mathcal{B}$ die Relation $p_1 \leq b$ durch Induktion über b. Die Intention ist dabei, dass $p_1 \leq b$ genau dann gilt, wenn für jede Priorität p_2, die in b auftritt, $p_1 \leq p_2$ gilt. Die formale Definition ist wie folgt:

1. $p_1 \leq \mathit{Nil}$,

   denn in dem leeren Baum treten überhaupt keine Prioritäten auf.

2. $p_1 \leq \mathit{Node}(p_2, v, l, r) \stackrel{\text{def}}{\longleftrightarrow} p_1 \leq p_2 \wedge p_1 \leq l \wedge p_1 \leq r$,

   denn p_1 ist genau dann kleiner-gleich als alle Prioritäten, die in dem Baum $\mathit{Node}(p_2, v, l, r)$ auftreten, wenn $p_1 \leq p_2$ gilt und wenn zusätzlich p_1 kleiner-gleich als alle Prioritäten ist, die in l oder r auftreten.

Als nächstes definieren wir eine Funktion

$$\mathit{count} : \mathcal{B} \to \mathbb{N},$$

die für einen binären Baum die Anzahl der Knoten berechnet. Die Definition erfolgt durch Induktion:

1. $\mathit{Nil}.\mathit{count}() = 0$.
2. $\mathit{Node}(p, v, l, r).\mathit{count}() = 1 + l.\mathit{count}() + r.\mathit{count}()$.

Mit diesen Vorbereitungen können wir nun die Menge *Heap* induktiv definieren:

1. $\mathit{Nil} \in \mathit{Heap}$.
2. $\mathit{Node}(p, v, l, r) \in \mathit{Heap}$ g.d.w. folgendes gilt:

   (a) $p \leq l \wedge p \leq r$,

   Die Priorität an der Wurzel ist also kleiner-gleich als alle anderen Prioritäten. Diese Bedingung bezeichnen wir auch als die *Heap-Bedingung*.

[1] Der Begriff *Heap* wird in der Informatik für zwei völlig unterschiedliche Dinge verwendet: Zum einen wird die in diesem Abschnitt beschriebene Daten-Struktur als *Heap* bezeichnet, zum anderen wird der Teil des Speichers, in dem dynamisch erzeugte Objekte abgelegt werden, ebenfalls als *Heap* bezeichnet.

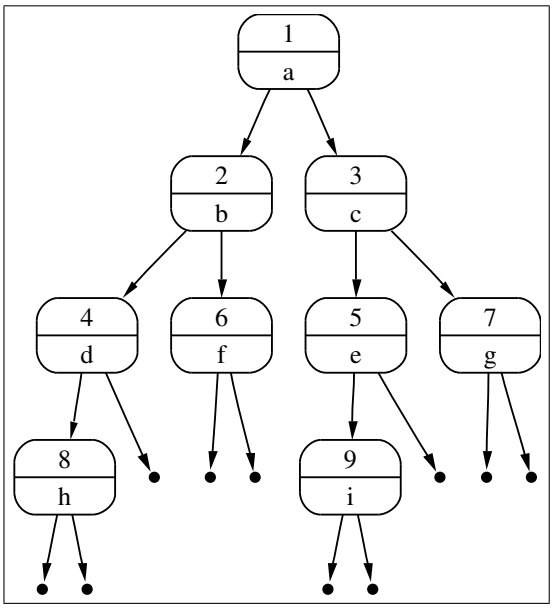

Figure 8.1: Ein Heap

(b) $| l.count() - r.count() | \leq 1$,
Die Zahl der Elemente im linken Teilbaum ist also höchstens 1 größer oder kleiner als die Zahl der Elemente im rechten Teilbaum. Diese Bedingung bezeichen wir als die *Balancierungs-Bedingung*. Sie ist ganz ähnlich zu der Balancierungs-Bedingung bei AVL-Bäumen, nur dass es dort die Höhe der Bäume ist, die verglichen wird, während wir hier die Zahl der im Baum gespeicherten Elemente vergleichen.

(c) $l \in \mathit{Heap} \wedge r \in \mathit{Heap}$.

Aus der *Heap-Bedingung* folgt, dass ein nicht-leerer Heap die Eigenschaft hat, dass das Element, welches an der Wurzel steht, immer die höchste Priorität hat. Abbildung 8.1 auf Seite 132 zeigt einen einfachen Heap. In den Knoten steht im oberen Teil die Prioritäten (in der Abbildung sind das natürliche Zahlen) und darunter stehen die Elemente (in der Abbildung sind dies Buchstaben).

Da Heaps binäre Bäume sind, können wir Sie ganz ähnlich wie geordnete binäre Bäume implementieren. Wir stellen zunächst Gleichungen auf, die die Implementierung der verschiedenen Methoden beschreiben. Wir beginnen mit der Methode *top*. Es gilt:

1. $\mathit{Nil}.\mathit{top}() = \Omega$.

2. $\mathit{Node}(p, v, l, r).\mathit{top}() = \langle p, v \rangle$,
denn aufgrund der Heap-Bedingung wird das Element mit der höchsten Priorität an der Wurzel gespeichert.

Die Methoden *insert* müssen wir nun so implementieren, dass sowohl die Balancierungs-Bedingung als auch die Heap-Bedingung erhalten bleiben.

1. $\mathit{Nil}.\mathit{insert}(p, v) = \mathit{Node}(p, v, \mathit{Nil}, \mathit{Nil})$.

2. $p_{\text{top}} \leq p \wedge l.count() \leq r.count() \rightarrow$
$\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p_{\text{top}}, v_{\text{top}}, l.\mathit{insert}(p, v), r\big)$.

Falls das einzufügende Paar eine geringere oder die selbe Priorität hat wie das Paar, welches sich an der Wurzel befindet, und falls zusätzlich die Zahl der Paare im linken Teilbaum kleiner-gleich der Zahl der Paare im rechten Teilbaum ist, dann fügen wir das Paar im linken Teilbaum ein.

3. $p_{\text{top}} \leq p \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p_{\text{top}}, v_{\text{top}}, l, r.\mathit{insert}(p, v)\big).$

   Falls das einzufügende Paar eine geringere oder die selbe Priorität hat als das Paar an der Wurzel und falls zusätzlich die Zahl der Paare im linken Teilbaum größer als die Zahl der Paare im rechten Teilbaum ist, dann fügen wir das Paar im rechten Teilbaum ein.

4. $p_{\text{top}} > p \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p, v, l.\mathit{insert}(p_{\text{top}}, v_{\text{top}}), r\big).$

   Falls das einzufügende Paar eine höhere Priorität hat als das Paar an der Wurzel, dann müssen wir das neu einzufügende Paar an der Wurzel positionieren. Das Paar, das dort vorher steht, fügen wir in den linken Teilbaum ein, falls die Zahl der Paare im linken Teilbaum kleiner-gleich der Zahl der Paare im rechten Teilbaum ist.

5. $p_{\text{top}} > p \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p, v, l, r.\mathit{insert}(p_{\text{top}}, v_{\text{top}})\big).$

   Falls wir das einzufügende Paar an der Wurzel positionieren müssen und die Zahl der Paare im linken Teilbaum größer als die Zahl der Paare im rechten Teilbaum ist, dann müssen wir das Paar, das vorher an der Wurzel stand, im rechten Teilbaum einfügen.

Als nächstes beschreiben wir die Implementierung der Methode *remove*.

1. $\mathit{Nil}.\mathit{remove}() = \mathit{Nil}$,

   denn aus dem leeren Heap ist nichts mehr zu entfernen.

2. $\mathit{Node}(p, v, \mathit{Nil}, r).\mathit{remove}() = r$,

3. $\mathit{Node}(p, v, l, \mathit{Nil}).\mathit{remove}() = l$,

   denn wir entfernen immer das Paar mit der höchsten Priorität und das ist an der Wurzel. Wenn einer der beiden Teilbäume leer ist, können wir einfach den anderen zurück geben.

   Jetzt betrachten wir die Fälle, wo keiner der beiden Teilbäume leer ist. Dann muss entweder das Paar an der Wurzel des linken Teilbaums oder das Paar an der Wurzel des rechten Teilbaums an die Wurzel aufrücken. Welches dieser beiden Paare wir nehmen, hängt davon ab, welches der Paare die höhere Priorität hat.

4. $p_1 \leq p_2 \ \wedge \ l = \mathit{Node}(p_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{Node}(p_2, v_2, l_2, r_2) \ \rightarrow$
   $\mathit{Node}(p, v, l, r).\mathit{remove}() = \mathit{Node}(p_1, v_1, l.\mathit{remove}(), r),$

   denn wenn das Paar an der Wurzel des linken Teilbaums eine höhere Priorität hat als das Paar an der Wurzel des rechten Teilbaums, dann rückt dieses Paar an die Wurzel auf und muss folglich aus dem linken Teilbaum gelöscht werden.

5. $p_1 > p_2 \ \wedge \ l = \mathit{Node}(p_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{Node}(p_2, v_2, l_2, r_2) \rightarrow$
   $\mathit{Node}(p, v, l, r).\mathit{remove}() = \mathit{Node}(p_2, v_2, l, r.\mathit{remove}()),$

   denn wenn das Paar an der Wurzel des rechten Teilbaums eine höhere Priorität hat als das Paar an der Wurzel des linken Teilbaums, dann rückt dieses Paar an die Wurzel auf und muss folglich aus dem rechten Teilbaum gelöscht werden.

```
class heap() {
    mPriority := om;
    mValue    := om;
    mLeft     := om;
    mRight    := om;
    mCount    := 0;

  static {
      top     := procedure()     { return [mPriority, mValue]; };
      insert  := procedure(p, v) { ... };
      remove  := procedure()     { ... };
      update  := procedure(t)    { ... };
      isEmpty := [] |-> mCount == 0;
  }
}
```

Figure 8.2: Outline of the class `heap`.

An dieser Stelle wird der aufmerksame Leser bemerken, dass die obige Implementierung der Methode *remove*() die Balancierungs-Bedingung verletzt. Es ist nicht schwierig, die Implementierung so abzuändern, dass die Balancierungs-Bedingung erhalten bleibt. Es zeigt sich jedoch, dass die Balancierungs-Bedingung nur beim Aufbau eines Heaps mittels *insert*() wichtig ist, denn dort garantiert sie, dass die Höhe des Baums in logarithmischer Weise von der Zahl seiner Knoten abhängt. Beim Löschen wird die Höhe des Baums sowieso nur kleiner, also brauchen wir uns da keine Sorgen machen.

Exercise 17: Change the equations for the method `remove` so that the resulting heap satisfies the balancing condition.

8.3 Implementing *Heaps* in SetlX

Next, we present an implementation of heaps in SetlX. Figure 8.2 shows an outline of the class `heap`. An object of class heap represents a node in a heap data structure. In order to do this, it maintains the following member variables:

1. `mPriority` is the priority of the value stored at this node,
2. `mValue` stores the corresponding value,
3. `mLeft` and `mRight` represent the left and right subtree, respectively, while
4. `mCount` gives the number of nodes in the subtree rooted at this node.

The constructor initializes these member variables in a way that the resulting object represents an empty heap. Since a heap stores the value with the highest priority at the root, implementing the method `top` is trivial: We just have to return the value stored at the root. The implementation of `isEmpty` is easy, too: We just have to check whether the number of values stored into this heap is zero.
Figure 8.3 show the implementation of the method `insert`. Basically, there are two cases.

1. If the given heap is empty, then we store the value to be inserted at the current node. We have to make sure to set `mLeft` and `mRight` to empty heaps. The

```
1   insert := procedure(priority, value) {
2       if (isEmpty()) {
3           this.mPriority := priority;
4           this.mValue    := value;
5           this.mLeft     := heap(this);
6           this.mRight    := heap(this);
7           this.mCount    := 1;
8           return;
9       }
10      this.mCount += 1;
11      if (priority < mPriority) {
12          if (mLeft.mCount > mRight.mCount) {
13              mRight.insert(mPriority, mValue);
14          } else {
15              mLeft.insert(mPriority, mValue);
16          }
17          this.mPriority := priority;
18          this.mValue    := value;
19      } else {
20          if (mLeft.mCount > mRight.mCount) {
21              mRight.insert(priority, value);
22          } else {
23              mLeft.insert(priority, value);
24          }
25      }
26  };
```

Figure 8.3: Implementation of the method `insert`.

reason is that, for every non-empty node, we want `mLeft` and `mRight` to store objects. Then, we can be sure that an expression like `mLeft.isEmpty()` is always well defined. If, however, we would allow `mLeft` to have the value `om`, then the evaluation of `mLeft.isEmpty()` would result in an error.

2. If the given heap is non-empty, we need another case distinction.

   (a) If the `priority` of the `value` to be inserted is higher[2] than `mPriority`, which is the priority of the value at the current node, then we have to put `value` at the current node, overwriting `mValue`. However, as we do not want to loose the value `mValue` that is currently stored at this node, we have to insert `mValue` into either the left or the right subtree. In order to keep the heap balanced we insert `mValue` into the smaller subtree and choose the left subtree if both subtrees have the same size.

   (b) If the `value` to be inserted has a lower `priority` than `mPriority`, then we have to insert `value` into one of the subtrees. Again, in order to maintain the balancing condition, `value` is stored into the smaller subtree.

Figure 8.4 shows the implementation of the method `remove`. This method removes the value with the highest priority from the heap. Essentially, there are two cases.

[2] Remember that we have defined a priority p_1 to be *higher* than a priority p_2 iff $p_1 < p_2$. I know that this sounds counter intuitive but unfortunately that is the way priorities are interpreted. You will understand the reason for this convention later on when we discuss Dijkstra's *shortest path algorithm*.

```
1   remove := procedure() {
2       this.mCount -= 1;
3       if (mLeft.isEmpty()) {
4           update(mRight);
5           return;
6       }
7       if (mRight.isEmpty()) {
8           update(mLeft );
9           return;
10      }
11      if (mLeft.mPriority < mRight.mPriority) {
12          this.mPriority := mLeft.mPriority;
13          this.mValue    := mLeft.mValue;
14          mLeft.remove();
15      } else {
16          this.mPriority := mRight.mPriority;
17          this.mValue    := mRight.mValue;
18          mRight.remove();
19      }
20  };
```

Figure 8.4: Implementation of the method `remove`.

1. If the left subtree is empty, we replace the given heap with the right subtree. Conversely, if the right subtree is empty, we replace the given heap with the left subtree.

2. Otherwise, we have to check which of the two subtrees contains the value with the highest priority. This value is then stored at the root of the given tree and, of course, it has to be removed from the subtree that had stored it previously.

```
1   update := procedure(t) {
2       this.mPriority := t.mPriority;
3       this.mValue    := t.mValue;
4       this.mLeft     := t.mLeft;
5       this.mRight    := t.mRight;
6       this.mCount    := t.mCount;
7   };
```

Figure 8.5: Implementation of the method `update`.

Figure 8.5 shows the implementation of the auxiliary method `update`. Its implementation is straightforward: It copies the member variables stored at the node `t` to the node `this`. This method is needed since in SetlX, assignments of the form

```
this := mLeft;   or   this := mRight;
```

are not permitted.

Exercise 18: The implementation of the method `remove` given above violates the balancing condition. Modify the implementation of `remove` so that the balancing condition remains valid.

Exercise 19: Instead of defining a class with member variables `mLeft` and `mRight`, a binary tree can be stored as a list l. In that case, for every index $i \in \{1, \cdots, \#l\}$, the expression $l[i]$ stores a node of the tree. The crucial idea is that the left subtree of the subtree stored at the index i is stored at the index $2 \cdot i$, while the right subtree is stored at the index $2 \cdot i + 1$. Develop an implementation of heaps that is based on this idea.

Chapter 9

Daten-Kompression

In diesem Kapitel untersuchen wir die Frage, wie wir einen gegebenen String s möglichst platzsparend abspeichern können. Wir gehen davon aus, dass der String s aus Buchstaben besteht, die Elemente einer Menge Σ sind. Die Menge Σ bezeichnen wir als unser *Alphabet*. Wenn das Alphabet aus n verschiedenen Zeichen besteht und wir alle Buchstaben mit der selben Länge von b Bits kodieren wollen, dann muss für diese Zahl von Bits offenbar

$$n \leq 2^b$$

gelten, woraus

$$b = \mathit{ceil}\big(\log_2(n)\big)$$

folgt. Hier bezeichnet $\mathit{ceil}(x)$ die *Ceiling-Funktion*. Diese Funktion rundet eine gegebene reelle Zahl immer auf, es gilt also

$$\mathit{ceil}(x) = \min\{k \in \mathbb{N} \mid x \leq k\}.$$

Besteht der String s aus m Buchstaben, so werden zur Kodierung des Strings insgesamt $m \cdot b$ Bits gebraucht. Nun gibt es zwei Möglichkeiten, weiterzumachen:

1. Lassen wir die Forderung, dass alle Buchstaben mit der selben Anzahl von Bits kodiert werden, fallen, dann ist es unter Umständen möglich, den String s mit weniger Bits zu kodieren. Dies führt zu dem 1952 von David A. Huffman angegebenen Algorithmus, den wir in den nächsten beiden Abschnitten vorstellen und analysieren.

2. Alternativ können wir versuchen, Buchstabenkombinationen, die häufig auftreten, als neue Buchstaben aufzufassen. Beispielsweise kommen Wörter wie "*der*", "*die*" und "*das*" in deutschsprachigen Texten relativ häufig vor. Es ist daher sinnvoll, für solche Buchstaben-Kombinationen neue Codes einzuführen. Ein Algorithmus, der auf dieser Idee basiert, ist der Lempel-Ziv-Welch Algorithmus, den wir im letzten Abschnitt dieses Kapitels diskutieren werden.

9.1 Motivation des Algorithmus von Huffman

Die zentrale Idee des von Huffman entwickelten Algorithmus ist die, dass Buchstaben, die sehr häufig auftreten, mit möglichst wenig Bits kodiert werden, während Buchstaben, die sehr selten auftreten, mit einer größeren Anzahl Bits kodiert werden. Zur Verdeutlichung betrachten wir folgendes Beispiel: Unser Alphabet Σ bestehe nur aus vier Buchstaben,

$$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}.$$

In dem zu speichernden String s trete der Buchstabe `a` insgesamt 990 mal auf, der Buchstabe `b` trete 8 mal auf und die Buchstaben `c` und `d` treten jeweil 1 mal auf. Dann besteht der String s aus insgesamt 1 000 Buchstaben. Wenn wir jeden Buchstaben mit $2 = \log_2(4)$ Bits kodieren, dann werden also insgesamt 2 000 Bits benötigt um den String s abzuspeichern. Wir können den String aber auch mit weniger Bits abspeichern, wenn wir die einzelnen Buchstaben mit Bitfolgen unterschiedlicher Länge kodieren. In unserem konkreten Beispiel wollen wir versuchen den Buchstaben `a`, der mit Abstand am häufigsten vorkommt, mit einem einzigen Bit zu kodieren. Bei den Buchstaben `c` und `d`, die nur sehr selten auftreten, ist es kein Problem auch mehr Bits zu verwenden. Tabelle 9.1 zeigt eine Kodierung, die von dieser Idee ausgeht.

Buchstabe	`a`	`b`	`c`	`d`
Kodierung	`0`	`10`	`110`	`111`

Table 9.1: Kodierung der Buchstaben mit variabler Länge.

Um zu verstehen, wie diese Kodierung funktioniert, stellen wir sie in Abbildung 9.1 als Baum dar. Die inneren Knoten dieses Baums enthalten keine Attribute und werden als leere Kreise dargestellt. Die Blätter des Baums sind mit den Buchstaben markiert. Die Kodierung eines Buchstabens ergibt sich über die Beschriftung der Kanten, die von dem Wurzel-Knoten zu dem Buchstaben führen. Beispielsweise führt von der Wurzel eine Kante direkt zu dem Blatt, das mit dem Buchstaben "`a`" markiert ist. Diese Kante ist mit dem Label "`0`" beschriftet. Also wird der Buchstabe "`a`" durch den String "`0`" kodiert. Um ein weiteres Beispiel zu geben, betrachten wir den Buchstaben "`c`". Der Pfad, der von der Wurzel zu dem Blatt führt, das mit "`c`" markiert ist, enthält drei Kanten. Die ersten beiden Kanten sind jeweils mit "`1`" markiert, die letzte Kante ist mit "`0`" markiert. Also wird der Buchstabe "`c`" durch den String "`110`" kodiert. Kodieren wir nun unseren ursprünglichen String s, der aus 990 `a`'s, 8 `b`'s, einem `c` und einem `d` besteht, so benötigen wir insgesamt

$$990 \cdot 1 + 8 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 1\,012$$

Bits. Gegenüber der ursprünglichen Kodierung, die 2 000 Bits verwendet, haben wir $49,4\%$ gespart!

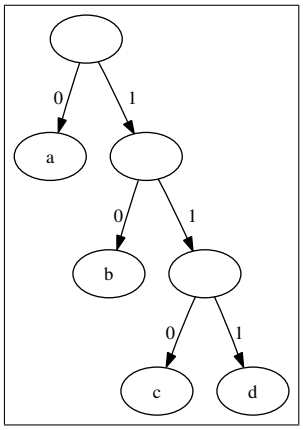

Figure 9.1: Baum-Darstellung der Kodierung.

Um zu sehen, wie mit Hilfe des Kodierungs-Baums ein String dekodiert werden kann, betrachten wir als Beispiel den String "`100111`". Wir beginnen mit der "`1`",

die uns sagt, vom Wurzel-Knoten dem rechten Pfeil zu folgen. Die anschließende "0" spezifiziert dann den linken Pfeil. Jetzt sind wir bei dem mit "b" markierten Blatt angekommen und haben damit den ersten Buchstaben gefunden. Wir gehen wieder zur Wurzel des Baums zurück. Die folgende "0" führt uns zu dem Blatt, das mit "a" markiert ist, also haben wir den zweiten Buchstaben gefunden. Wir gehen wieder zur Wurzel zurück. Die Ziffern "111" führen uns nun zu dem Buchstaben "d". Damit haben wir insgesamt

$$\text{“100111”} \simeq \text{“bad”}.$$

9.2 Der Algorithmus von Huffman

Angenommen, wir haben einen String s, der aus Buchstaben eines Alphabets Σ aufgebaut ist. Wie finden wir dann eine Kodierung für die einzelnen Buchstaben, die mit möglichst wenig Bits auskommt? Der Algorithmus von Huffman gibt eine Antwort auf diese Frage. Um diesen Algorithmus präsentieren zu können, definieren wir die Menge $\mathcal{K}$ der *Kodierungs-Bäume* induktiv.

1. $\mathit{Leaf}(c, f) \in \mathcal{K}$ falls $c \in \Sigma$ und $f \in \mathbb{N}$.

   Ausdrücke der Form $\mathit{Leaf}(c, f)$ sind die Blätter eines Kodierungs-Baums. Dabei ist c ein Buchstabe aus unserem Alphabet Σ und f gibt die Häufigkeit an, mit der dieser Buchstabe in dem zu kodierenden String auftritt.

   Gegenüber Abbildung 9.1 kommen hier bei den Blättern noch die Häufigkeiten hinzu. Diese benötigen wir, denn wir wollen ja später Buchstaben, die sehr häufig auftreten, mit möglichst wenig Bits kodieren.

2. $\mathit{Node}(l, r) \in \mathcal{K}$ falls $l \in \mathcal{K}$ und $r \in \mathcal{K}$.

   Ausdrücke der Form $\mathit{Node}(l, r)$ sind die inneren Knoten eines Kodierungs-Baums.

Als nächstes definieren wir eine Funktion

$$\mathit{count} : \mathcal{K} \to \mathbb{N},$$

welche die Gesamt-Häufigkeiten aller in dem Baum auftretenden Buchstaben aufsummiert.

1. Die Definition der Funktion *count* ist für Blätter trivial:

   $$\mathit{Leaf}(c, f).\mathit{count}() = f.$$

2. Die Gesamt-Häufigkeit des Knotens $\mathit{Node}(l, r)$ ergibt sich als Summe der Gesamt-Häufigkeiten von l und r. Also gilt

   $$\mathit{Node}(l, r).\mathit{count}() = l.\mathit{count}() + r.\mathit{count}().$$

Weiter definieren wir auf Kodierungs-Bäumen die Funktion

$$\mathit{cost} : \mathcal{K} \to \mathbb{N}.$$

Die Funktion *cost* gibt an, wie viele Bits benötigt werden, um mit dem gegebenen Kodierungs-Baum einen String zu kodieren, wenn die Häufigkeiten, mit denen ein Buchstabe verwendet wird, mit den Häufigkeiten übereinstimmen, die an den Blättern des Baums notiert sind. Die Definition dieser Funktion ist induktiv:

1. $\mathit{Leaf}(c, f).\mathit{cost}() = 0$,

   denn solange nur ein einziger Buchstabe vorhanden ist, ist noch nichts zu kodieren.

2. $\mathit{Node}(l, r).\mathit{cost}() = l.\mathit{cost}() + r.\mathit{cost}() + l.\mathit{count}() + r.\mathit{count}()$.

   Wenn wir zwei Kodierungs-Bäume l und r zu einem neuen Kodierungs-Baum zusammenfügen, verlängern sich die Kodierungen für alle Buchstaben, die in l oder r auftreten, um ein Bit. Die Summe

   $$l.\mathit{count}() + r.\mathit{count}()$$

   gibt die Gesamt-Häufigkeiten aller Buchstaben an, die in dem linken und rechten Teilbaum auftreten. Da sich die Kodierung aller dieser Buchstaben durch die Bildung des Knotens $\mathit{Node}(l, r)$ gegenüber der Kodierung in l und r jeweils um 1 verlängert, müssen wir zu den Kosten der Teilbäume l und r den Term $l.\mathit{count}() + r.\mathit{count}()$ hinzuaddieren.

Wir erweitern die Funktion $\mathit{cost}()$ auf Mengen von Knoten, indem wir die Kosten einer Menge M als die Summe der Kosten der Knoten von M definieren:

$$\mathit{cost}(M) = \sum_{n \in M} n.\mathit{cost}().$$

Ausgangs-Punkt des von David A. Huffman (1925 – 1999) im Jahre 1952 angegebenen Algorithmus [Huf52] ist eine Menge von Paaren der Form $\langle c, f \rangle$. Dabei ist c ein Buchstabe und f gibt die Häufigkeit an, mit der dieser Buchstabe auftritt. Im ersten Schritt werden diese Paare in die Blätter eines Kodierungs-Baums überführt. Besteht der zu kodierende String aus n verschiedenen Buchstaben, so haben wir dann eine Menge von Kodierungs-Bäumen der Form

$$M = \big\{\mathit{Leaf}(c_1, f_1), \cdots, \mathit{Leaf}(c_k, f_k)\big\} \tag{9.1}$$

Es werden nun solange Knoten a und b aus M zu einem neuen Knoten $\mathit{Node}(a, b)$ zusammen gefasst, bis die Menge M nur noch einen Knoten enthält. Offenbar gibt es im Allgemeinen sehr viele Möglichkeiten, die Knoten aus der Menge zu neuen Knoten zusammen zu fassen. Das Ziel ist es die Knoten so zusammen zu fassen, dass die Kosten der Menge M am Ende minimal sind. Um zu verstehen, welche Knoten wir am geschicktesten zusammenfassen können, betrachten wir, wie sich die Kosten der Menge durch das Zusammenfassen zweier Knoten ändert. Dazu betrachten wir zwei Mengen von Knoten M_1 und M_2, so dass

$$M_1 = N \cup \{a, b\} \quad \texttt{und} \quad M_2 = N \cup \{\mathit{Node}(a, b)\}$$

gilt, die Menge M_1 geht also aus der Menge M_2 dadurch hervor, dass wir die Knoten a und b zu einem neuen Knoten zusammen fassen und durch diesen ersetzen. Untersuchen wir, wie sich die Kosten der Menge dabei verändern, wir untersuchen also die folgende Differenz:

$$\begin{array}{lll}
 & \mathit{cost}\big(N \cup \{\mathit{Node}(a, b)\}\big) - \mathit{cost}\big(N \cup \{a, b\}\big) \\
= & \mathit{cost}\big(\{\mathit{Node}(a, b)\}\big) - \mathit{cost}\big(\{a, b\}\big) \\
= & \mathit{Node}(a, b).\mathit{cost}() - a.\mathit{cost}() - b.\mathit{cost}() \\
= & a.\mathit{cost}() + b.\mathit{cost}() + a.\mathit{count}() + b.\mathit{count}() - a.\mathit{cost}() - b.\mathit{cost}() \\
= & a.\mathit{count}() + b.\mathit{count}()
\end{array}$$

Fassen wir die Knoten a und b aus der Menge M zu einem neuen Knoten zusammen, so vergößern sich die Kosten der Menge um die Summe

$$a.\mathit{count}() + b.\mathit{count}().$$

Wenn wir die Kosten der Menge M insgesamt möglichst klein halten wollen, dann ist es daher naheliegend, dass wir in der Menge M die beiden Knoten a und b suchen, für die die Funktion *count*() den kleinsten Wert liefert. Diese Knoten werden wir aus der Menge M entfernen und durch den neuen Knoten $Node(a, b)$ ersetzen. Dieser Prozess wird solange iteriert, bis die Menge M nur noch aus einem Knoten besteht. Dieser Knoten ist dann die Wurzel des gesuchten Kodierungs-Baums.

```
1  codingTree := procedure(m) {
2      while (#m > 1) {
3          a := first(m);
4          m -= { a };
5          b := first(m);
6          m -= { b };
7          m += { [ count(a) + count(b), Node(a, b) ] };
8      }
9      return arb(m);
10 };
11 count := p |-> p[1];
```

Figure 9.2: Der Algorithmus von Huffman in SetlX.

Die in Abbildung 9.2 gezeigte Funktion `codingTree`(m) implementiert diesen Algorithmus.

1. Die Funktion *codingTree* wird mit einer Menge m von Knoten aufgerufen, welche die Form

   $$m = \{\langle f_1, Leaf(c_1)\rangle, \cdots, \langle f_k, Leaf(c_k)\rangle\}$$

   hat. Hier bezeichnen die Variablen c_i die verschiedenen Buchstaben, während die Zahl f_i die Häufigkeit angibt, mit der der Buchstabe c_i auftritt.

   Wir haben hier die Information über die Häufigkeit an erster Stelle eines Paares gespeichert. Da SetlX diese Menge intern durch einen geordneten binären Baum abspeichert, ermöglicht uns diese Form der Darstellung einfach auf den Knoten mit der kleinsten Häufigkeit zuzugreifen, denn die Paare werden so verglichen, dass immer zunächst die erste Komponente zweier Paare zum Vergleich herangezogen werden. Nur wenn sich in der ersten Komponente kein Unterschied ergibt, wird auch die zweite Komponente verglichen. Daher finden wir das Paar mit der kleinsten ersten Komponente immer am Anfang der Menge m.

   Durch diesen Trick haben wir uns de facto die Implementierung einer Prioritäts-Warteschlange gespart:

   (a) Die in SetlX vordefinierte Funktion `first`(m) liefert das erste Element der Menge m und entspricht damit der Funktion *top*(m) des abstrakten Daten-Typs *PrioQueue*.

   (b) Anstelle von *insert*(m, p, v) können wir einfach

   $$m \texttt{ += \{ [}p\texttt{, }v\texttt{] \};}$$

   schreiben um das Element v mit der Priorität p in die Prioritäts-Warteschlange m einzufügen.

   (c) Die Funktion *remove*(m) realisieren wir durch den Aufruf

   $$m \texttt{ -= \{ first(}m\texttt{) \};}$$

denn *remove*(m) soll ja das Element mit der höchsten Priorität aus m entfernen.

Das Elegante an diesem Vorgehen ist, dass damit sämtliche Operationen des abstrakten Daten-Typs *PrioQueue* eine logarithmische Komplexität haben. Das ist zwar im Falle der Operation *top*(m) nicht optimal, aber für die Praxis völlig ausreichend, denn in der Praxis kommt auf jeden Aufruf der Form *top*(m) auch ein Aufruf der Form *remove*(m) und der hat sowohl bei einer optimalen Implementierung als auch bei unserer Implementierung eine logarithmische Komplexität, die dann auch im Falle der optimalen Implementierung die gesamte Komplexität dominiert.

2. Die `while`-Schleife in Zeile 2 veringert die Anzahl der Knoten in der Menge m in jedem Schritt um Eins.

   (a) Dazu werden mit Hilfe der Funktion *first*() die beiden Knoten a und b berechnet, für die der Wert von *count*() minimal ist. Die Funktion `count`(p) ist in Zeile 11 definiert und liefert einfach die erste Komponente des Paares p, denn dort speichern wir die Häufigkeit der Buchstaben ab.

   (b) Die beiden Knoten a und b mit der geringsten Häufigkeit werden in Zeile 4 und 6 aus der Menge m entfernt.

   (c) Anschließend wird aus den beiden Knoten a und b ein neuer Knoten *Node*(a, b) gebildet. Dieser neue Knoten wird zusammen mit der Gesamthäufigkeit der Knoten a und b in Zeile 7 der Menge m hinzugefügt.

3. Die `while`-Schleife wird beendet, wenn die Menge m nur noch ein Element enthält. Dieses wird mit der Funktion `arb` extrahiert und als Ergebnis zurück gegeben.

Die Laufzeit des Huffman-Algorithmus hängt stark von der Effizienz der Funktion *first*() ab. Eine naive Implementierung würde die Knoten aus der Menge m in einer geordneten Liste vorhalten. Die Knoten n wären in dieser Liste nach der Größe n.*cost*() aufsteigend sortiert. Dann ist die Funktion *first*() zwar sehr effizient, aber das Einfügen des neuen Knotens, dass wir oben über den Befehl

```
m += { [ count(a) + count(b), Node(a, b) ] };
```

realisieren, würde einen Aufwand erfordern, der linear in der Anzahl der Elemente der Menge m ist. Dadurch, dass wir mit einer Menge m arbeiten, die in SetlX intern durch einen Rot-Schwarz-Baum dargestellt ist, erreichen wir, dass alle in der `while`-Schleife durchgeführten Operationen nur logarithmisch von der Anzahl der Buchstaben abhängen. Damit hat der Huffman-Algorithmus insgesamt die Komplexität $\mathcal{O}\big(n \cdot \ln(n)\big)$.

Buchstabe	a	b	c	d	e
Häufigkeit	1	2	3	4	5

Table 9.2: Buchstaben mit Häufigkeiten.

Wir illustrieren den Huffman-Algorithmus, indem wir ihn auf die Buchstaben, die in Tabelle 9.2 zusammen mit ihren Häufigkeiten angegeben sind, anwenden.

1. Zu Beginn hat die Menge m die Form

$$m = \big\{\langle 1, \mathit{Leaf}(\texttt{a})\rangle,\ \langle 2, \mathit{Leaf}(\texttt{b})\rangle,\ \langle 3, \mathit{Leaf}(\texttt{c})\rangle,\ \langle 4, \mathit{Leaf}(\texttt{d})\rangle,\ \langle 5, \mathit{Leaf}(\texttt{e})\rangle\big\}.$$

2. Die Häufigkeit ist hier für die Blätter mit den Buchstaben `a` und `b` minimal. Also entfernen wir diese Blätter aus der Menge und fügen statt dessen den

Knoten

$$\mathit{Node}(\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b}))$$

in die Menge m ein. Die Häufigkeit dieses Knotens ergibt sich als Summe der Häufigkeiten der Buchstaben `a` und `b`. Daher fügen wir insgesamt das Paar

$$\langle 3, \mathit{Node}\big(\mathit{Leaf}(\texttt{a})\rangle, \mathit{Leaf}(\texttt{b})\big)\rangle$$

in die Menge m ein. Dann hat m die Form

$$\big\{\langle 3, \mathit{Node}(\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b}))\rangle,\ \langle 3, \mathit{Leaf}(\texttt{c})\rangle,\ \langle 4, \mathit{Leaf}(\texttt{d})\rangle,\ \langle 5, \mathit{Leaf}(\texttt{e})\rangle\big\}.$$

3. Die beiden Paare mit den kleinsten Werten der Häufigkeiten in m sind nun

   $$\langle 3, \mathit{Node}(\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b}))\rangle \quad \text{und} \quad \langle 3, \mathit{Leaf}(\texttt{c})\rangle.$$

   Wir entfernen diese beiden Knoten und bilden aus diesen beiden Knoten den neuen Knoten

   $$\langle 6, \mathit{Node}(\mathit{Node}((\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b})),\ \mathit{Leaf}(\texttt{c}))\rangle,$$

   den wir der Menge m hinzufügen. Dann hat m die Form

   $$\Big\{\langle 4, \mathit{Leaf}(\texttt{d})\rangle,\ \langle 5, \mathit{Leaf}(\texttt{e})\rangle,\ \langle 6, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b})),\ \mathit{Leaf}(\texttt{c}))\Big\}.$$

4. Jetzt sind

   $$\langle 4, \mathit{Leaf}(\texttt{d})\rangle \quad \text{und} \quad \langle 5, \mathit{Leaf}(\texttt{e})\rangle$$

   die beiden Knoten mit dem kleinsten Werten der Häufigkeit. Wir entfernen diese Knoten und bilden den neuen Knoten

   $$\langle 9, \mathit{Node}(\mathit{Leaf}(\texttt{d}), \mathit{Leaf}(\texttt{e}))\rangle.$$

   Diesen fügen wir der Menge m hinzu und erhalten

   $$\Big\{\langle 6, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b})),\ \mathit{Leaf}(\texttt{c}, 3))\rangle,\ \langle 9, \mathit{Node}(\mathit{Leaf}(\texttt{d}, 4), \mathit{Leaf}(\texttt{e}, 5))\rangle\Big\}.$$

5. Jetzt enthält die Menge m nur noch zwei Knoten. Wir entfernen diese beiden Knoten und bilden daraus den neuen Knoten

   $$\mathit{Node}\Big(\mathit{Node}\big(\mathit{Node}\big(\mathit{Leaf}(\texttt{a}), \mathit{Leaf}(\texttt{b})\big),\ \mathit{Leaf}(\texttt{c})\big),\ \mathit{Node}\big(\mathit{Leaf}(\texttt{d}), \mathit{Leaf}(\texttt{e})\big)\Big)$$

   Dieser Knoten ist jetzt der einzige Knoten in m und damit unser Ergebnis. Stellen wir diesen Knoten als Baum dar, so erhalten wir das in Abbildung 9.3 gezeigte Ergebnis. Wir haben hier jeden Knoten n mit dem Funktionswert $n.\mathit{count}()$ beschriftet.

   Die Kodierung, die sich daraus ergibt, wird in Tabelle 9.3 gezeigt.

Buchstabe	a	b	c	d	e
Kodierung	000	001	01	10	11

Table 9.3: Kodierung der Buchstaben mit variabler Länge.

Exercise 20:

1. Berechnen Sie den Huffman-Code für einen Text, der nur die Buchstaben "`a`" bis "`g`" enthält und für den die Häufigkeiten, mit denen diese Buchstaben auftreten, durch die folgende Tabelle gegeben sind.

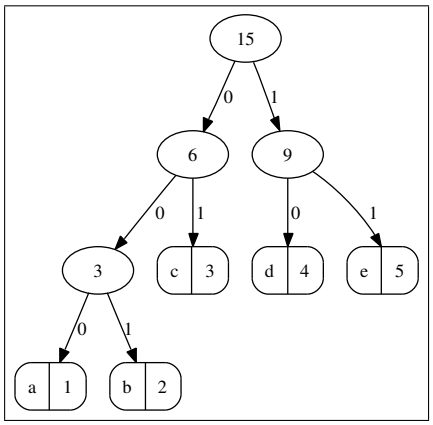

Figure 9.3: Baum-Darstellung der Kodierung.

Buchstabe	a	b	c	d	e	f	g
Häufigkeit	1	1	2	3	5	8	13

Table 9.4: Buchstaben mit Häufigkeiten.

2. Wie groß ist die Einsparung, wenn man die Buchstaben mit einem Huffman-Code kodiert gegenüber einer Kodierung mit drei Bits?

3. Versuchen Sie das Gesetz zu erkennen, nach dem die Häufigkeiten in der obigen Tabelle gebildet wurden und versuchen Sie, den Huffman-Code für den allgemeinen Fall, in dem n Buchstaben gegeben sind, anzugeben.

4. Wie groß ist die Einsparung im allgemeinen Fall?

9.3 The Algorithm of Lempel, Ziv, and Welch

The algorithm developed by Abraham Lempel, Jacob Ziv [ZL77, ZL78] and Terry A. Welch [Wel84], which is also known as the *LZW algorithm*, is based on the idea that in most texts certain combinations of letters are quite frequent. Therefore, it should pay of to view these combinations of letters as new letters and insert them into the alphabet. This is the main idea of the LZW algorithm. However, since counting the occurrences of all words would be too time consuming, the LZW algorithm works with a *dynamic* coding dictionary. Initially, this dictionary contains only the Ascii characters. Then, the idea is to extend this dictionary dynamically: Every time a new string is encountered, it is entered into the dictionary and a code is assigned to the corresponding string. However, since it would not make sense to add arbitrary strings to the dictionary, a new string s of length $n = \#s$ is only added to the dictionary if the substring $s[1..n-1]$ has already been entered into the dictionary. This procedure is best explained via an example. The basic working of the algorithm is explained with the help of four variables:

1. α is the last substring that has been encoded. Initially, this is the empty string ε.

   The encoding of a string s by the LZW algorithm works by encoding substrings of s as numbers and α denotes the last of theses substrings.

2. c is the next character of the string that is inspected. This is also know as the *look-ahead character*.

3. d ist the dictionary mapping strings to numbers. Initially, d maps all ASCII characters to their respective ASCII codes.

4. `nextCode` is the number assigned as code to the next string that is entered into the dictionary d. Since the ASCII codes are the numbers from 0 up to 127, initially `nextCode` is equal to 128.

To describe the working of the algorithm, let us encode the string "`maumau`".

1. Initially, we have

   $$\alpha = \varepsilon \quad \text{and} \quad c = \texttt{m}.$$

   Since the ASCII code of the character "`m`" is 109, we output this number.

2. After reading the next character "`a`" we have

   $$\alpha = \texttt{m} \quad \text{and} \quad c = \texttt{a}.$$

   Now, the substring αc, which is "`ma`", is entered into the dictionary and assigned to the code 128:

   $$d{:=}f \cup \{\langle \texttt{ma}, 128\rangle\}.$$

   Furthermore, we output the ASCII code of "`a`", which is 97.

3. After reading the next character "`u`" we have

   $$\alpha = \texttt{a} \quad \text{and} \quad c = \texttt{u}.$$

   Now, the substring αc, which is "`au`". is entered into the dictionary and assigned to the next available code, which is 129:

   $$d{:=}f \cup \{\langle \texttt{au}, 129\rangle\}.$$

   Furthermore, we output the ASCII code of "`u`", which is 117.

4. After reading the next character, which is the character "`m`", we have

   $$\alpha = \texttt{u} \quad \text{and} \quad c = \texttt{m}.$$

   Next, the substring αc, which is "`um`", is entered into the dictionary and assigned to the next available code, which is 130:

   $$d{:=}f \cup \{\langle \texttt{um}, 130\rangle\}.$$

   Since our dictionary already contains the substring "`ma`" and the character "`a`" is indeed the character following the character "`m`", we output 128, which is the code assigned to the string "`ma`".

5. The next character to be read is now the final character "`u`". We have

   $$\alpha = \texttt{ma} \quad \text{and} \quad c = \texttt{u}.$$

   Next, the substring αc, which is "`mau`", is entered into the dictionary and assigned to the next available code, which is 131:

   $$d{:=}f \cup \{\langle \texttt{mau}, 131\rangle\}.$$

   Furthermore, we output the ASCII code of "`u`", which is 117.

Putting everything together, we have coded the string "`maumau`" as the list

$$[109, 97, 117, 128, 117]$$

If we had encoded this string in ASCII we would have used $6 \cdot 7 = 42$ bits. Since the dictionary that we have built on the fly uses codes starting at 128 we now have to

use 8 bits to encode the numbers. However, we have only used 5 numbers to encode the string "`maumau`". Hence we have only used $5 \cdot 8 = 40$ bits. Of course, in this tiny example the compression factor is quite low. However, for texts that are longer and have more repetitions, the compression factor is usually higher: On average, the experience shows that text corresponding to natural language is compressed by a factor of 2.

If we use the LZW algorithm there is no need to add the dictionary to the encoded string. The reason is that the recipient of an encoded string can construct the dictionary using exactly the algorithm that is used when encoding the string.

Let us summarize the algorithm seen in the previous example:

1. The dictionary is initialized to map all Ascii characters to their Ascii codes.
2. Next, we search for the longest prefix β of s that is in the dictionary. This prefix is removed from s.
3. We emit the code stored for β in the dictionary.
4. Let α be the string that has been encoded in the previous step. Append the first character c of β to α and enter the resulting string αc to the dictionary.

   This step expands the dictionary dynamically.
5. Go to step 2 and repeat as long as the string s is not empty.

Decoding a list of numbers l into a string s is quite similar to the encoding and works as follows.

1. This time, the dictionary is initialized to map all Ascii codes to their corresponding Ascii characters. Hence, the dictionary constructed in this step is just the inverse as the dictionary constructed when starting to encode the string.
2. We initialize s as the empty string[1]:

   $$s := \varepsilon.$$
3. We remove the first number n from the list l and look up the corresponding string β in the dictionary. This string is appended to s.
4. Assume that α is the string decoded in the previous iteration and that c is the first character of β. Enter the resulting string αc into the dictionary.
5. Goto step 2 and repeat as long as the list l is not empty.

The third step of this algorithm needs to refined: The problem is that it might happen that the dictionary does not have an entry for the number n. This can occur because the encoder is one step ahead of the decoder: The encoder encodes a substring and enters a code corresponding to the previous substring into the dictionary. Now if the next substring is identical to the substring just entered, the encoder will produce a code that is not yet in the dictionary of the decoder when he tries to decode it. The question then is: How do we encode a number that has not yet been entered into the dictionary. To answer this question, we can reason an follows: If the encoder outputs a code that it has just entered into the dictionary, then the string that is encoded starts with the string that has been output previously, followed by some character. However, this character must be the first character of the string encoded now. The string encoded now corresponds to the code and hence this string is the same as the string previously decoded plus one character. Therefore, if the previous string is α, then the string corresponding to an unknown code must be $\alpha\alpha[1]$, i.e. α followed by the first character of α.

[1] In computer science, the empty string is denoted as ε.

9.3.1 Implementing the LZW algorithm in SetlX

In order to gain a better understanding of a complex algorithm it is best to code this algorithm. Then the resulting program can be run on several examples. Since humans tend to learn better from examples than from logical reasoning, inspecting these examples deepens the understanding of the algorithm. We proceed to discuss an implementation of the LZW algorithm.

```
1  class lzw() {
2      mDictionary := { [ char(i), i ] : i in [32 .. 127] };
3      mInverse    := { [ i, char(i) ] : i in [32 .. 127] };
4      mNextCode   := 128;
5      mBitNumber  := 8;
6      mCodeSize   := 0;
7
8      static {
9          compress      := procedure(s)     { ... };
10         uncompress    := procedure(l)     { ... };
11         longestPrefix := procedure(s, i)  { ... };
12         incrementBitNumber := procedure() { ... };
13     }
14 }
```

Figure 9.4: Outline of the class `lzw`.

Figure 9.4 shows the outline of the class `lzw`. This class contains both the method `compress` that takes a string s and encodes this string into a list of numbers and the method `uncompress` that takes a list of numbers l and decodes this list back into a string s. These methods are designed to satisfy the following specification:

$$l = \texttt{lzw}().\texttt{compress}(s_1) \wedge s_2 = \texttt{lzw}().\texttt{uncompress}(l) \rightarrow s_1 = s_2.$$

Furthermore, the class `lzw` contains the auxiliary methods `longestPrefix` and `incrementBitNumber`, which will be discussed later. The class `lzw` contains 5 member variables:

1. `mDictionary` is the dictionary used when encoding a string. It is initialized to map the Ascii characters to their codes. Remember that for a given number i, the expression `char`(i) returns the Ascii character with code i.

2. `mInverse` is a binary relation that associates the codes with the corresponding strings. It is initialized to map every number in the set $\{0, 1, 2, \cdots, 127\}$ with the corresponding Ascii character. The binary relation `mInverse` is the inverse of the relation `mDictionary`.

3. `mNextCode` gives the value of the next code used in the dictionary. Since the codes up to and including 127 are already used for the Ascii character, the next available code will be 128.

4. `mBitNumber` is the number of bits needed to encode the next number to be entered into the dictionary. Since the codes inserted into the dictionaries start at 128 and this number needs 8 bits to be encoded, `mBitNumber` is initialized to 8 bits.

5. `mCodeSize` is the total number of bits used to encode the string s given to the method `compress`. This member variable is only needed to keep track of the performance of the algorithm; maintaining this variable is not necessary for the algorithm to work.

```
1  compress := procedure(s) {
2      result := [];
3      idx    := 1;
4      while (idx <= #s) {
5          p := longestPrefix(s, idx);
6          result[#result+1] := mDictionary[s[idx..p]];
7          this.mCodeSize += mBitNumber;
8          if (p < #s) {
9              mDictionary[s[idx..p+1]] := mNextCode;
10             this.mNextCode += 1;
11             incrementBitNumber();
12         }
13         idx := p + 1;
14     }
15     return result;
16 };
```

Figure 9.5: The method `compress` encodes a string as a list of integers.

Figure 9.5 shows the implementation of the method compress. We discuss this implementation line by line.

1. The variable `result` points to the list that encodes the string s given as argument. Initially, this list is empty. Every time a substring of s is encoded, the corresponding code is appended to this list.

2. The variable `idx` is an index into the string s. The idea is that the substring $s[1..\texttt{idx}-1]$ has been encoded and the corresponding codes have already been written to the list `result`, while the substring $s[\texttt{idx}..]$ is the part of s that still needs to be encoded.

3. Hence, the `while`-loop runs as long as the index `idx` is less or equal that the length `#`s of the string s.

4. Next, the method `longestPrefix` computes the index of longest prefix of the substring $s[\texttt{idx}..]$ that can be found in the dictionary `mDictionary`, i.e. p is the maximal number such that the expression `mDictionary[s[idx..p]]` is defined.

5. The code corresponding to this substring is looked up in the dictionary `mDictionary` and is then appended to the list `result`.

6. Since we want to keep track of the number of bits needed to encode the string we increment the variable `mCodeSize` by the number of bits currently used to write a code.

7. Next, we take care to maintain the dictionary `mDictionary` and add the substring $s[\texttt{idx}..p+1]$ to the dictionary. Of course, we can only do this if the upper index of this expression, which is $p+1$, is an index into the string s.

Therefore we have to check that $p < \#s$. Once we have entered the new string with its corresponding code into the dictionary, we have to make sure that the variable `mNextCode` is incremented so that every string is associated with a unique code. When we increment `mNextCode` it can happen that the new value of `mNextCode` has grown so large that the number of bits `mBitNumber` is no longer sufficient to store `mNextCode`. In this case, `mBitNumber` needs to be incremented. The function `incrementBitNumber` increments the variable `mBitNumber` if this is necessary.

8. Since the code corresponding to the substring $s[\texttt{idx}..p]$ has been written to the list `result`, the index `idx` is set to $p + 1$.

9. Once the while loop has terminated, the string s has been completely encoded and the list containing the codes can be returned.

```
1  longestPrefix := procedure(s, i) {
2     oldK := i;
3     k    := i+1;
4     while (k <= #s && mDictionary[s[i..k]] != om) {
5         oldK := k;
6         k    += 1;
7     }
8     return oldK;
9  };
10 incrementBitNumber := procedure() {
11     if (2 ** mBitNumber <= mNextCode) {
12         this.mBitNumber += 1;
13     }
14 };
```

Figure 9.6: Computing the longest prefix.

Figure 9.6 show the implementation of the auxiliary functions `longestPrefix` and `incrementBitNumber`.

1. The function `longestPrefix`(s, i) computes the maximum value of k such that

   $$i \leq k \wedge k \leq \#s \wedge \texttt{mDictionary}[s[i..k]] \neq \Omega.$$

   This value is well defined since the dictionary is initialized to contain all strings of length 1. Therefore, `mDictionary`$[s[i..i]]$ is known to be defined: It is the Ascii code of the character $s[i]$.

   The required value is computed by a simple `while`-loop that tests all possible values of k. The loop exits once the value of k is too big. Then the previous value of k, which is stored in the variable `oldK` is returned as the result.

2. The function `incrementBitNumber` tests whether `mNextCode` can be stored in with `mBitNumber` bits. This is the case if

   $$\texttt{mNextCode} < 2^{\texttt{mBitNumber}}$$

   holds true. Otherwise, `mBitNumber` is incremented so that the inequality given above is reestablished.

```
1  uncompress := procedure(l) {
2      result := "";
3      idx    := 1;
4      code   := l[idx];
5      old    := mInverse[code];
6      idx    += 1;
7      while (idx < #l) {
8          result += old;
9          code := l[idx];
10         idx  += 1;
11         next := mInverse[code];
12         if (next == om) {
13             next := old + old[1];
14         }
15         mInverse[mNextCode] := old + next[1];
16         this.mNextCode += 1;
17         old := next;
18     }
19     result += old;
20     return result;
21 };
```

Figure 9.7: The method `uncompress` to decode a list of integers into a string.

Figure 9.7 shows the implementation of the method `uncompress` that takes a list of numbers and decodes it into a string s.

1. The variable `result` contains the decoded string. Initially, this variable is empty. Every time a code of the list l is deciphered into some string, this string is added to `result`.

2. The variable `idx` is an index into the list l. It points to the next code that needs to be deciphered.

3. The variable `code` contains the code in l at position `idx`. Therefore, we always have

   $$l[\texttt{idx}] = \texttt{code}$$

4. The variable `old` contains the substring associated with `code`. Therefore, the invariant

   $$\texttt{mInverse}[\texttt{code}] = \texttt{old}$$

   is maintained.

5. As long as the index `idx` still points inside the list, the substring that has just been decoded is appended to the string `result`.

6. Then, an attempt is made to decode the next number in the list l by looking up the code in the dictionary `mInverse`.

   Now there is one subtle case: If the `code` has not yet been defined in the dictionary, then we can conclude that this code has been created when coding the substring `old` followed by some character c. However, as the next substring β corresponds to this code, the character c must be the first character of this

substring, i.e. we have

$$c = \beta[1].$$

On the other hand, we know that the substring β has the form

$$\beta = \texttt{old} + c,$$

where the operator "+" denotes string concatenation. But then the first character of this string must be the first character of `old`, i.e. we have

$$\beta[1] = \texttt{old}[1]$$

and hence we have shown that

$$c = \texttt{old}[1].$$

Therefore, we conclude

$$\beta = \texttt{old} + \texttt{old}[1]$$

and hence this is the string encoded by a code that is not yet defined in the dictionary `mInverse`.

7. Next, we need to maintain the dictionary `mInverse` in the same fashion as the dictionary `mDictionary` is maintained in the method `compress`: Hence we take the string previously decoded and concat the next character of the string decoded in the current step. Of course, this string is

   $$\texttt{old} + \texttt{next}[1]$$

   and this string is then associated with the next available code value.

8. At the end of the loop, we need to set `old` to `next` so that `old` will always contain the string decoded in the previous step.

9. When the `while`-loop has terminated, we still need to append the final value of `old` to the variable `result`.

Now that we have discussed the implementation of the `LZW` algorithm I would like to encourage you to test it on several examples that are not too long. Time does not permit me to discuss examples of this kind in these lecture notes and, indeed, I do not think that discussing these examples here would be as beneficial for the student as performing the algorithm on their own.

Exercise 21:

1. Use the LZW algorithm to decode the string "`abcabcabcabc`". Compute the compression factor for this string.

2. For all $n \in \mathbb{N}$ with $n \geq 1$ the string α_n is defined inductively as follows:

   $$\alpha_1 := \texttt{a} \quad \text{and} \quad \alpha_{n+1} = \alpha_n + \texttt{a}.$$

   Hence, the string α_n has the form $\underbrace{\texttt{a}\cdots\texttt{a}}_{n}$, i.e. it is the character `a` repeated n times. Encode the strings α_n using the LZW algorithm. What is the compression rate?

3. Decode the list

   $$[97, 98, 128, 130]$$

   using the LZW algorithm.

Chapter 10

Graph Theory

Wir wollen zum Abschluß der Vorlesung wenigstens ein graphentheoretisches Problem vorstellen: Das Problem der Berechnung kürzester Wege.

10.1 Die Berechnung kürzester Wege

Um das Problem der Berechnung kürzester Wege formulieren zu können, führen wir zunächst den Begriff des *gewichteten Graphen* ein.

Definition 21 (Gewichteter Graph) Ein *gewichteter Graph* ist ein Tripel $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ so dass gilt:

1. $\mathbb{V}$ ist eine Menge von *Knoten*.
2. $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ ist eine Menge von *Kanten*.
3. $\|\cdot\| : \mathbb{E} \to \mathbb{N}\backslash\{0\}$ ist eine Funktion, die jeder Kante eine positive *Länge* zuordnet.

□

Ein *Pfad* P ist eine Liste der Form

$$P = [x_1, x_2, x_3, \cdots, x_n]$$

so dass für alle $i = 1, \cdots, n-1$ gilt:

$$\langle x_i, x_{i+1} \rangle \in \mathbb{E}.$$

Die Menge aller Pfade bezeichnen wir mit $\mathbb{P}$. Die Länge eines Pfads definieren wir als die Summe der Länge aller Kanten:

$$\big\|[x_1, x_2, \cdots, x_n]\big\| \;:=\; \sum_{i=1}^{n-1} \big\|\langle x_i, x_{i+1}\rangle\big\|.$$

Ist $p = [x_1, x_2, \cdots, x_n]$ ein Pfad, so sagen wir, dass p den Knoten x_1 mit dem Knoten x_n verbindet. Die Menge alle Pfade, die den Knoten v mit dem Knoten w verbinden, bezeichnen wir als

$$\mathbb{P}(v, w) \;:=\; \big\{[x_1, x_2, \cdots, x_n] \in \mathbb{P} \mid x_1 = v \wedge x_n = w\big\}.$$

Damit können wir nun das Problem der Berechnung kürzester Wege formulieren.

Definition 22 (Kürzeste-Wege-Problem) Gegeben sei ein gewichteter Graph $G = \langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ und ein Knoten $\texttt{source} \in \mathbb{V}$. Dann besteht das *kürzeste-Wege-Problem* darin, die folgende Funktion zu berechnen:

$$\texttt{sp} : \mathbb{V} \to \mathbb{N}$$
$$\texttt{sp}(v) \;:=\; \texttt{min}\big\{\|p\| \mid p \in \mathbb{P}(\texttt{source}, v)\big\}. \qquad \square$$

10.1.1 Der Algorithmus von Moore

Wir betrachten zunächst den Algorithmus von Moore [Moo59] zur Berechnung des kürzeste-Wege-Problems. Abbildung 10.1 zeigt eine Implementierung dieses Algorithmus' in SetlX.

```
1  shortestPath := procedure(source, edges) {
2      dist   := { [source, 0] };
3      fringe := { source };
4      while (fringe != {}) {
5          u := from(fringe);
6          for ([v,l] in edges[u] | dist[v] == om || dist[u]+l < dist[v]) {
7              dist[v] := dist[u] + l;
8              fringe  += { v };
9          }
10     }
11     return dist;
12 };
```

Figure 10.1: Algorithmus von Moore zur Lösung des kürzeste-Wege-Problems.

1. Die Funktion `shortestPath`(*source*, *edges*) wird mit zwei Parametern aufgerufen:
   (a) *source* ist der Start-Knoten, von dem aus wir die Entfernungen zu den anderen Knoten berechnen.
   (b) *edges* ist eine binäre Relation, die für jeden Knoten x eine Menge von Paaren der Form
   $$\{[y_1, l_1], \cdots, [y_n, l_n]\}$$
   speichert. Die Idee dabei ist, dass für jeden in dieser Menge gespeicherten Knoten y_i eine Kante $\langle x, y_i \rangle$ der Länge l_i existiert.
2. Die Variable *dist* speichert die Abstands-Funktions als binäre Relation, also als Menge von Paaren der Form $[x, \texttt{sp}(x)]$. Hierbei ist $x \in \mathbb{V}$ und $\texttt{sp}(x)$ ist der Abstand, den der Knoten x von dem Start-Knoten *source* hat.
   Der Knoten *source* hat von dem Knoten *source* offenbar den Abstand 0 und zu Beginn unserer Rechnung ist das auch alles, was wir wissen. Daher wird die Relation *dist* in Zeile 2 mit dem Paar `[source, 0]` initialisiert.
3. Die Variable *fringe* enthält alle die Knoten, von denen ausgehend wir als nächstes die Abstände benachbarter Knoten berechnen sollten. Am Anfang wissen wir nur von *source* den Abstand und daher ist dies der einzige Knoten, mit dem wir die Menge *fringe* in Zeile 3 initialisieren.
4. Solange es nun Knoten gibt, von denen ausgehend wir neue Wege berechnen können, wählen wir in Zeile 5 einen beliebigen Knoten aus der Menge *fringe* aus und entfernen ihn aus dieser Menge.

5. Anschließend berechnen wir die Menge aller Knoten v, für die wir jetzt einen neuen Abstand gefunden haben:

   (a) Das sind einerseits die Knoten v, für welche die Funktion $dist(v)$ bisher noch undefiniert war, weil wir diese Knoten in unserer bisherigen Reechnung noch gar nicht gesehen haben.

   (b) Andererseits sind dies aber auch Knoten, für die wir schon einen Abstand haben, der aber größer ist als der Abstand des Weges, den wir erhalten, wenn wir die Knoten von u aus besuchen.

   Für alle diese Knoten berechnen wir den Abstand und fügen diese Knoten dann in die Menge *fringe* ein.

6. Der Algorithmus terminiert, wenn die Menge *fringe* leer ist, denn dann haben wir alle Knoten abgeklappert.

10.1.2 Der Algorithmus von Dijkstra

```
1  shortestPath := procedure(source, edges) {
2      dist    := { [source, 0] };
3      fringe  := { [0, source] };
4      visited := {};
5      while (fringe != {}) {
6          [d, u]  := first(fringe);
7          fringe  -= { [d, u] };
8          for ([v,l] in edges[u] | dist[v]==om || d+l<dist[v]) {
9              fringe  -= { [dist[v], v] };
10             dist[v] := d + l;
11             fringe  += { [d + l, v] };
12         }
13         visited += { u };
14     }
15     return dist;
16 };
```

Figure 10.2: Der Algorithmus von Dijkstra zur Lösung des kürzeste-Wege-Problems.

Im Algorithmus von Moore ist die Frage, in welcher Weise die Knoten aus der Menge *fringe* ausgewählt werden, nicht weiter spezifiziert. Die Idee bei dem von Edsger W. Dijkstra (1930 – 2002) im Jahre 1959 veröffentlichten Algorithmus [Dij59] besteht darin, immer den Knoten auszuwählen, der den geringsten Abstand zu dem Knoten *source* hat. Dazu wird die Menge *fringe* nun als eine Prioritäts-Warteschlange implementiert. Als Prioritäten wählen wir die Entfernungen zu dem Knoten `source`. Abbildung 10.2 auf Seite 155 zeigt die Implementierung des Algorithmus von Dijkstra zur Berechnung der kürzesten Wege in der Sprache SetlX.

In dem Problem in Abbildung taucht noch eine Variable mit dem Namen *visited* auf. Diese Variable bezeichnet die Menge der Knoten, die der Algorithmus schon *besucht* hat. Genauer sind das die Knoten u, die aus der Prioritäts-Warteschlange `fringe` entfernt wurden und für die dann anschließend in der `for`-Schleife, die in Zeile 8 beginnt, alle zu u benachbarten Knoten untersucht wurden. Die Menge *visited* hat für die eigentliche Implementierung des Algorithmus keine Bedeutung, denn die Variable *visited* wird nur in Zeile 4 und Zeile 13 geschrieben, aber sie

wird an keiner Stelle gelesen. Ich habe die Variable *visited* nur deshalb eingeführt, damit ich eine Invariante formulieren kann, die für den Beweis der Korrektheit des Algorithmus zentral ist. Diese Invariante lautet

$$\forall u \in \texttt{visited} : dist[u] = sp(u).$$

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 dem Programm gibt es genau zwei Stellen, an denen die Menge *visited* verändert wird.

I.A.: Zu Beginn wird die Menge *visited* mit der leeren Menge initialisiert und daher ist die Behauptung für alle Elemente, die zu Beginn in der Menge *visited* sind, trivialerweise richtig.

I.S.: In Zeile 13 fügen wir den Knoten u in die Menge *visited* ein. Wir betrachten nun die Situation unmittelbar vor dem Einfügen von u. Falls u bereits ein Elemente der Menge *visited* sein sollte, gilt die Behauptung nach Induktions-Voraussetzung. Wir brauchen also nur den Fall betrachten, dass u vor dem Einfügen noch kein Element der Menge *visited* ist.

Wir führen den weiteren Beweis nun indirekt und nehmen an, dass

$$dist(u) > sp(u)$$

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

$$x_0 \in visited, \cdots, x_i \in visited \quad \text{aber} \quad x_{i+1} \notin \texttt{Visited},$$

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 $x_{i+1} \notin visited$ ist, muss x_{i+1} immer noch in der Prioritäts-Warteschlange *fringe* liegen. Also muss $dist[x_{i+1}] \geq dist[u]$ gelten, denn sonst wäre x_{i+1} vor u aus der Prioritäts-Warteschlange entfernt worden. Wegen $sp(x_{i+1}) = dist[x_{i+1}]$ haben wir dann aber den Widerspruch

$$sp(u) \geq sp(x_{i+1}) = dist[x_{i+1}] \geq dist[u] > sp(u). \qquad \square$$

10.1.3 Komplexität

Wenn ein Knoten u aus der Warteschlange *fringe* entfernt wird, ist er anschließend ein Element der Menge *visited* und aus der oben gezeigten Invariante folgt, dass dann

$$sp(u) = dist[u]$$

gilt. Daraus folgt aber notwendigerweise, dass der Knoten u nie wieder in die Prioritäts-Warteschlange *fringe* eingefügt werden kann, denn ein Knoten v wird nur dann in *fringe* neu eingefügt, wenn entweder die Funktion $dist[v]$ noch undefiniert ist, oder sich der Wert von `dist`$[v]$ verkleinert. Das Einfügen eines Knoten in eine Prioritäts-Warteschlange mit n Elementen kostet eine Rechenzeit, die durch $\mathcal{O}\big(\log_2(n)\big)$ abgeschätzt werden kann. Da die Warteschlange sicher nie mehr als $\#\mathbb{V}$ Knoten enthalten kann und da jeder Knoten höchstens einmal eingefügt werden kann, liefert das einen Term der Form

$$\mathcal{O}\big(\#V \cdot \log_2(\#V)\big)$$

für das Einfügen der Knoten. Neben dem Einfügen eines Knotens durch den Befehl

```
fringe += { [dvNew, v] };
```

müssen wir auch die Komplexität des Aufrufs

```
fringe -= { [dvOld, v] };
```

analysieren. Die Anzahl dieser Aufrufe ist durch die Anzahl der Kanten begrenzt, die zu dem Knoten v hinfügen. Da das Entfernen eines Elements aus einer Menge mit n Elementen eine Rechenzeit der Größe $\mathcal{O}\big(\log_2(n)\big)$ erfordert, haben wir die Abschätzung

$$\mathcal{O}\big(\#\mathbb{E} \cdot \log_2(\#\mathbb{V})\big)$$

für diese Rechenzeit. Dabei bezeichnet $\#\mathbb{E}$ die Anzahl der Kanten. Damit erhalten wir für die Komplexität von Dijkstra's Algorithmus insgesamt den Ausdruck

$$\mathcal{O}\big((\#\mathbb{E} + \#\mathbb{V}) * \ln(\#\mathbb{V})\big).$$

Ist die Zahl der Kanten, die von den Knoten ausgehen können, durch eine feste Zahl begrenzt (z.B. wenn von jedem Knoten nur maximal 4 Kanten ausgehen), so kann die Gesamt-Zahl der Kanten durch ein festes Vielfaches der Knoten-Zahl abgeschätzt werden. Dann ist die Komplexität für Dijkstra's Algorithmus zur Bestimmung der kürzesten Wege durch den Ausdruck

$$\mathcal{O}\big(\#\mathbb{V} * \log_2(\#\mathbb{V})\big)$$

gegeben.

Chapter 11

Die Monte-Carlo-Methode

Bestimmte Probleme sind so komplex, dass es mit vertretbarem Aufwand nicht möglich ist, eine exakte Lösung zu berechnen. Oft läßt sich jedoch mit Hilfe einer Simulation das Problem zumindest näherungsweise lösen.

1. Das Problem der Berechnung der Volumina von Körpern, die eine große Zahl von Begrenzungsflächen haben, läßt sich auf die Berechnung mehrdimensionaler Integrale zurückführen. In der Regel können diese Integrationen aber nicht analytisch ausgeführt werden. Mit der Monte-Carlo-Methode läßt sich hier zumindest ein Näherungswert bestimmen.

2. Die Gesetzmäßigkeiten des Verhaltens komplexer Systeme, die zufälligen Einflüssen einer Umgebung ausgesetzt sind, können oft nur durch Simulationen bestimmt werden. Wird beispielsweise ein neues U-Bahn-System geplant, so wird die Kapazität eines projektierten Systems durch Simulationen ermittelt.

3. Bei Glückspielen ist die exakte Berechnung bestimmter Wahrscheinlichkeiten oft nicht möglich. Mit Hilfe von Simulationen lassen sich aber gute Näherungswerte bestimmen.

Die obige Liste könnte leicht fortgesetzt werden. In diesem Kapitel werden wir zwei Beispiele betrachten.

1. Als einführendes Beispiel zeigen wir, wie sich mit Hilfe der Monte-Carlo-Methode Flächeninhalte bestimmen lassen. Konkret berechnen wir den Flächeninhalt eines Kreises und bestimmen auf diese Weise die Zahl π.

2. Als zweites Beispiel zeigen wir, wie sich Karten zufällig mischen lassen. Damit kann beispielsweise die Wahrscheinlichkeit dafür berechnet werden, dass im Texas Hold'em Poker eine gegebene Hand gegen eine zufällige Hand gewinnt.

11.1 Berechnung der Kreiszahl π

Eine sehr einfache Methode zur Berechnung einer Approximation der Zahl π funktioniert wie folgt. Wir betrachten in der reellen Ebene den Einheits-Kreis E, der als die Menge

$$E = \{\langle x, y\rangle \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1\}$$

definiert ist. Der Ausdruck $\sqrt{x^2 + y^2}$ gibt nach dem Satz des Pythagoras gerade den Abstand an, den der Punkt $\langle x, y\rangle$ vom Koordinatenursprung $\langle 0, 0\rangle$ hat. Der

Einheits-Kreis hat offenbar den Radius $r = 1$. Damit gilt für die Fläche dieses Kreises

$$\mathit{Fläche}(E) = \pi \cdot r^2 = \pi.$$

Wenn es uns gelingt, diese Fläche zu berechnen, dann haben wir also π bestimmt. Eine experimentelle Methode zur Bestimmung dieser Fläche besteht darin, dass wir in das Quadrat Q, dass durch

$$Q = \{\langle x, y\rangle \in \mathbb{R} \mid -1 \leq x \leq 1 \ \wedge \ -1 \leq x \leq 1\}$$

definiert ist, zufällig eine große Zahl n von Sandkörnern werfen. Wir notieren uns dabei die Zahl k der Sandkörner, die in den Einheits-Kreis fallen. Die Wahrscheinlichkeit p dafür, dass ein Sandkorn in den Einheits-Kreis fällt, wird nun proportional zur Fläche des Einheits-Kreises sein:

$$p = \frac{\mathit{Fläche}(E)}{\mathit{Fläche}(Q)}.$$

Da das Quadrat die Seitenlänge 2 hat, gilt für die Fläche des Quadrats Q die Formel

$$\mathit{Fläche}(Q) = 2^2 = 4.$$

Auf der anderen Seite wird bei einer hohen Anzahl von Sandkörnern das Verhältnis $\frac{k}{n}$ gegen diese Wahrscheinlichkeit p streben, so dass wir insgesamt

$$\frac{k}{n} \approx \frac{\pi}{4}$$

haben, woraus sich für π die Näherungsformel

$$\pi \approx 4 \cdot \frac{k}{n}$$

ergibt. Während die alten Ägypter bei dieser historischen Methode zur Berechung von π noch Tonnen von Sand benötigten, können wir dieses Experiment heute einfacher mit Hilfe eines Computers durchführen.

```
1  approximatePi := procedure(n) {
2      k := 0;
3      i := 0;
4      while (i < n) {
5          x := 2 * random() - 1;
6          y := 2 * random() - 1;
7          r := x * x + y * y;
8          if (r <= 1) {
9              k += 1;
10         }
11         i += 1;
12     }
13     return 4.0 * k / n;
14 };
```

Figure 11.1: Experimentelle Bestimmung von π mit Hilfe der Monte-Carlo-Methode.

Abbildung 11.1 zeigt die Funktion `approximatePi`, die mit dem oben beschriebenen Verfahren einen Näherungswert für π berechnet.

1. Der Parameter n gibt die Anzahl der Sandkörner an, die wir in das Quadrat Q werfen.

2. Um ein Sandkorn zufällig zu werfen, werden mit Hilfe der Funktion *random*() zunächst Zufallszahlen erzeugt, die in dem Intervall $[0, 1]$ liegen. Mit Hilfe der Transformation

   $$t \mapsto 2 \cdot t - 1$$

   wird das Intervall $[0, 1]$ in das Intervall $[-1, 1]$ transformiert, so dass die in den Zeilen 5 und 6 berechneten Koordinaten `x` und `y` ein zufällig in das Quadrat Q geworfenes Sandkorn beschreiben.

3. Wir berechnen in Zeile 7 das Quadrat des Abstandes dieses Sandkorns vom Koordinatenursprung und überprüfen in Zeile 8, ob das Sandkorn innerhalb des Kreises liegt.

n	Näherung für π	Fehler der Näherung
10	2.40000	-0.741593
100	3.28000	+0.138407
1 000	3.21600	+0.074407
10 000	3.13080	-0.010793
100 000	3.13832	-0.003273
1 000 000	3.13933	-0.002261
10 000 000	3.14095	-0.000645
100 000 000	3.14155	-0.000042
1 000 000 000	3.14160	+0.000011

Table 11.1: Ergebnisse bei der Bestimmung von π mit der Monte-Carlo-Methode

Lassen wir das Progamm laufen, so erhalten wir die in Tabelle 11.1 gezeigten Ergebnisse. Wir sehen, dass wir zur Berechnung von π auf eine Genauigkeit von zwei Stellen hinter dem Komma etwa 100 000 Versuche brauchen, was angesichts der Rechenleistung heutiger Computer kein Problem darstellt. Die Berechnung weiterer Stellen gestaltet sich jedoch sehr aufwendig: Die Berechnung der dritten Stelle hinter dem Komma erfordert 100 000 000 Versuche. Grob geschätzt können wir sagen, dass sich der Aufwand bei der Berechnung jeder weiteren Stelle verhundertfacht! Wir halten folgende Beobachtung fest:

> Die Monte-Carlo-Methode ist gut geeignet, um einfache Abschätzungen zu berechnen, wird aber sehr aufwendig, wenn eine hohe Genauigkeit gefordert ist.

11.2 Theoretischer Hintergrund

Wir diskutieren nun den theoretischen Hintergrund der Monte-Carlo-Methode. Da im zweiten Semester noch keine detailierteren Kenntnisse aus der Wahrscheinlichkeitsrechnung vorhanden sind, beschränken wir uns darauf, die wesentlichen Ergebnisse anzugeben. Eine Begründung dieser Ergebnisse erfolgt dann in der Mathematik-Vorlesung im vierten Semester.

Bei der Monte-Carlo-Methode wird ein Zufalls-Experiment, im gerade diskutierten Beispiel war es das Werfen eines Sandkorns, sehr oft wiederholt. Für den

Ausgang dieses Zufalls-Experiments gibt es dabei zwei Möglichkeiten: Es ist entweder erfolgreich (im obigen Beispiel landet das Sandkorn im Kreis) oder nicht erfolgreich. Ein solches Experiment bezeichnen wir als *Bernoulli-Experiment*. Hat die Wahrscheinlichkeit, dass das Experiment erfolgreich ist, den Wert p und wird das Experiment n mal ausgeführt, so ist die Wahrscheinlichkeit, dass genau k dieser Versuche erfolgreich sind, durch die Formel

$$P(k) = \frac{n!}{k! \cdot (n-k)!} \cdot p^k \cdot (1-p)^{n-k}$$

gegeben, die auch als *Binomial-Verteilung* bekannt ist. Für große Werte von n ist die obige Formel sehr unhandlich, kann aber gut durch die *Gauß-Verteilung* approximiert werden, es gilt

$$\frac{n!}{k! \cdot (n-k)!} \cdot p^k \cdot (1-p)^{n-k} \approx \frac{1}{\sqrt{2 \cdot \pi \cdot n \cdot p \cdot (1-p)}} \cdot \exp\left(-\frac{(k-n \cdot p)^2}{2 \cdot n \cdot p \cdot (1-p)}\right)$$

Wird das Experiment n mal durchgeführt, so erwarten wir im Durchschnitt natürlich, dass $n \cdot p$ der Versuche erfolgreich sein werden. Darauf basiert unsere Schätzung für den Wert von p, denn wir approximieren p durch die Formel

$$p \approx \frac{k}{n},$$

wobei k die Anzahl der erfolgreichen Experimente bezeichnet. Nun werden in der Regel nicht genau $n \cdot p$ Versuche erfolgreich sein: Zufallsbedingt werden ein Paar mehr oder ein Paar weniger Versuche erfolgreich sein. Das führt dazu, dass unsere Schätzung von p eine Ungenauigkeit aufweist, deren ungefähre Größe wir irgendwie abschätzen müssen um unsere Ergebnisse beurteilen zu können.

Um eine Idee davon zu bekommen, wie sehr die Anzahl der erfolgreichen Versuche von dem Wert $\frac{k}{n}$ abweicht, führen wir den Begriff der *Streuung* σ ein, die für eine binomialverteilte Zufallsgröße durch die Formel

$$\sigma = \sqrt{n \cdot p \cdot (1-p)}$$

gegeben ist. Die Streuung gibt ein Maß dafür, wie stark der gemessene Wert von k von dem im Mittel erwarteten Wert $p \cdot n$ abweicht. Es kann gezeigt werden, dass die Wahrscheinlichkeit, dass k außerhalb des Intervalls

$$[p \cdot n - 3 \cdot \sigma,\, p \cdot n + 3 \cdot \sigma]$$

liegt, also um mehr als das dreifache von dem erwarteten Wert abweicht, kleiner als 0.27% ist. Für die Genauigkeit unserer Schätzung $p \approx \frac{k}{n}$ heißt das, dass dieser Schätzwert mit hoher Wahrscheinlichkeit (99.73%) in dem Intervall

$$\left[\frac{p \cdot n - 3 \cdot \sigma}{n},\, \frac{p \cdot n + 3 \cdot \sigma}{n}\right] = \left[p - 3 \cdot \frac{\sigma}{n},\, p + 3 \cdot \frac{\sigma}{n}\right]$$

liegt. Die Genauigkeit $\varepsilon(n)$ ist durch die halbe Länge dieses Intervalls gegeben und hat daher den Wert

$$\varepsilon(n) = 3 \cdot \frac{\sigma}{n} = 3 \cdot \sqrt{\frac{p \cdot (1-p)}{n}}.$$

Wir erkennen hier, dass zur Erhöhung der Genauigkeit um den Faktor 10 die Zahl der Versuche um den Faktor 100 vergrößert werden muss.

Wenden wir die obige Formel auf die im letzen Abschnitt durchgeführte Berechnung der Zahl π an, so erhalten wir wegen $p = \frac{\pi}{4}$ die in Abbildung 11.2 gezeigten Ergebnisse.

Anzahl Versuche n	Genauigkeit $\varepsilon(n)$
10	0.389478
100	0.123164
1 000	0.0389478
10 000	0.0123164
100 000	0.00389478
1 000 000	0.00123164
10 000 000	0.000389478
100 000 000	0.000123164
1 000 000 000	3.89478e-05
10 000 000 000	1.23164e-05
100 000 000 000	3.89478e-06

Table 11.2: Genauigkeit der Bestimung von π bei einer Sicherheit von $99,73\%$.

Exercise 22: Wieviel Tonnen Sand benötigte Pharao Ramses II, als er π mit der Monte-Carlo-Methode auf 6 Stellen hinter dem Komma berechnet hat?

Hinweise:

1. Ein Sandkorn wiegt im Durchschnitt etwa $\frac{1}{1000}$ Gramm.
2. Um eine Genauigkeit von 6 Stellen hinter dem Komma zu haben, sollte der Fehler durch 10^{-7} abgeschätzt werden. Das Ergebnis soll mit einer Wahrscheinlichkeit von $99,7\%$ korrekt sein.

Solution: Nach dem Hinweis soll

$$\varepsilon(n) = 10^{-7}$$

gelten. Setzen wir hier die Formel für $\varepsilon(n)$ ein, so erhalten wir

$$\begin{aligned} & 3 \cdot \sqrt{\tfrac{p\cdot(1-p)}{n}} && = && 10^{-7} \\ \Leftrightarrow\quad & 9 \cdot \tfrac{p\cdot(1-p)}{n} && = && 10^{-14} \\ \Leftrightarrow\quad & 9 \cdot p \cdot (1-p) \cdot 10^{14} && = && n \end{aligned}$$

Um an dieser Stelle weitermachen zu können, benötigen wir den Wert der Wahrscheinlichkeit p. Der korrekte Wert von p ist für unser Experiment durch $\frac{\pi}{4}$ gegeben. Da wir π ja erst berechnen wollen, nehmen wir als Abschätzung von π den Wert 3, so dass p den Wert $\frac{3}{4}$ hat. Da eine Tonne insgesamt 10^9 Sandkörner enthält, bekommen wir für das Gewicht g das Ergebnis

$$\begin{aligned} g &\approx 9 \cdot \frac{3}{4} \cdot \frac{1}{4} \cdot 10^5 \text{ Tonnen} \\ &\approx 168\,750 \text{ Tonnen} \end{aligned}$$

Die Cheops-Pyramide ist 35 mal so schwer. □

Exercise 23: Berechnen Sie mit Hilfe der Monte-Carlo-Methode eine Näherung für den Ausdruck $\ln(2)$. Ihre Näherung soll mit einer Wahrscheinlichkeit von 99.73% eine Genauigkeit von $\varepsilon = 10^{-3}$ haben.

11.3 Erzeugung zufälliger Permutationen

In diesem Abschnitt lernen wir ein Verfahren kennen, mit dem es möglich ist, eine gegebene Liste zufällig zu permutieren. Anschaulich kann ein solches Verfahren mit dem Mischen von Karten verglichen werden. Das Verfahren wird auch tatsächlich genau dazu eingesetzt: Bei der Berechnung von Gewinn-Wahrscheinlichkeiten bei Kartenspielen wie Poker wird das Mischen der Karten durch den gleich vorgestellten Algorithmus erledigt.

Um eine n-elementige Liste $L = [x_1, x_2, \cdots, x_n]$ zufällig zu permutieren, unterscheiden wir zwei Fälle:

1. Die Liste L hat die Länge 1 und besteht folglich nur aus einem Element, $L = [x]$. In diesem Fall gibt die Funktion $\mathit{permute}(L)$ die Liste unverändert zurück:

   $$\#L = 1 \rightarrow \mathit{permute}(L) = L$$

2. Die Liste L hat eine Länge, die größer als 1 ist. In diesem Fall wählen wir zufällig ein Element aus, das hinterher in der zu erzeugenden Permutation an der letzten Stelle stehen soll. Wir entfernen dieses Element aus der Liste und permutieren anschließend die verbleibende Liste. An die dabei erhaltene Permutation hängen wir noch das anfangs ausgewählte Element an. Haben wir eine Funktion

   $$\mathit{random} : \mathbb{N} \rightarrow \mathbb{N},$$

   so dass der Aufruf $\mathit{random}(n)$ zufällig eine Zahl aus der Menge $\{1, \cdots, n\}$ liefert, so können wir diese Überlegung wie folgt formalisieren:

   $$\#L = n \wedge n > 1 \wedge k := \mathit{random}(n) \rightarrow \mathit{permute}(L) = \mathit{permute}\big(\mathit{delete}(L, k)\big) + \big[L(k)\big].$$

   Der Funktionsaufruf $\mathit{delete}(L, k)$ löscht dabei das k-te Element aus der Liste L, wir könnten also schreiben

   $$\mathit{delete}(L, k) = L(1 \,..\, k-1) + L(k+1 \,..\, \#L).$$

```
1  permute := procedure(l) {
2      if (#l == 1) {
3          return l;
4      }
5      k := rnd([1..#l]);
6      return permute(l[..k-1] + l[k+1..]) + [l[k]];
7  };
```

Figure 11.2: Berechnung zufälliger Permutationen eines Feldes

Abbildung 11.2 zeigt die Umsetzung dieser Idee in SetlX. Die dort gezeigte Methode *permute* erzeugt eine zufällige Permutation der Liste l, die als Argument übergeben wird. Die Implementierung setzt die oben beschriebenen Gleichungen unmittelbar um.

Es kann gezeigt werden, dass der oben vorgestellte Algorithmus tatsächlich alle Permutationen einer gegebenen Liste mit der selben Wahrscheinlichkeit erzeugt. Einen Beweis dieser Behauptung finden Sie beispielsweise in [CLRS01].

Bibliography

[AHU87] Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[AP10] Adnan Aziz and Amit Prakash. *Algorithms for Interviews*. CreateSpace Independent Publishing Platform, 2010.

[AVL62] Georgii M. Adel'son-Vel'skiĭ and Evgenii M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[BM93] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[dlB59] René de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of the Western Joint Computer Conference*, pages 195–298, 1959.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[GS08] Hans-Peter Gumm and Manfred Sommer. *Einführung in die Informatik*. Oldenbourg-Verlag, 8th edition, 2008.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[Hoa61] C. Antony R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4:321, 1961.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, 2nd edition, 2006.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Lut09] Mark Lutz. *Learning Python*. O'Reilly, 4th edition, 2009.

[MGS96] Martin Müller, Thomas Glaß, and Karl Stroetmann. Automated modular termination proofs for real prolog programs. In Radhia Cousot and David A. Schmidt, editors, *SAS*, volume 1145 of *Lecture Notes in Computer Science*, pages 220–237. Springer, 1996.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.

[SW11a] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[SW11b] Robert Sedgewick and Kevin Wayne. *Algorithms in Java*. Pearson, 4th edition, 2011.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[vR95] Guido van Rossum. Python tutorial. Technical report, Centrum Wiskunde & Informatica, Amsterdam, 1995.

[Wel84] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[WS92] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Assoc., 1992.

[Yar09] Vladimir Yaroslavskiy. Dual-pivot quicksort. Technical report, Mathematics and Mechanics Faculty, Saint-Petersburg State University, 2009. This paper is available at `http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf`.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.