Algorithms

— Lecture Notes for the Summer Term 2017 —

Prof. Dr. Karl Stroetmann

May 2, 2017

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

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

The lecture notes itself can be found in the file

`Lecture-Notes/algorithms.pdf`.

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`. Using this command will be necessary as there will be frequent changes of these lecture notes.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Overview . . . 3
- 1.3 Algorithms and Programs . . . 4
- 1.4 Desirable Properties of Algorithms . . . 5
- 1.5 Literature . . . 5
- 1.6 A Final Remark . . . 6
- 1.7 A Request . . . 6

2 Big $\mathcal{O}$ Notation **7**
- 2.1 Motivation . . . 7
- 2.2 A Remark on Notation . . . 16
- 2.3 Computation of Powers . . . 17
- 2.4 The Master Theorem . . . 20
- 2.5 Variants of Big $\mathcal{O}$ Notation . . . 25
- 2.6 Further Reading . . . 26

3 Sorting **27**
- 3.1 Insertion Sort . . . 29
  - 3.1.1 Complexity of Insertion Sort . . . 30
- 3.2 Selection Sort . . . 31
  - 3.2.1 Complexity of Selection Sort . . . 32
- 3.3 Merge Sort . . . 33
  - 3.3.1 Complexity of Merge Sort . . . 35
  - 3.3.2 Implementing Merge Sort for Arrays . . . 36
  - 3.3.3 An Iterative Implementation of Merge Sort . . . 39
  - 3.3.4 Further Improvements of Merge Sort . . . 40
- 3.4 Quick Sort . . . 40
  - 3.4.1 Complexity . . . 41
  - 3.4.2 Implementing *Quick Sort* for Arrays . . . 45
  - 3.4.3 Improvements for *Quick Sort* . . . 46
- 3.5 A Lower Bound . . . 48

4 Abstract Data Types **50**
- 4.1 Formal Definition . . . 50
- 4.2 Implementation . . . 52
- 4.3 Evaluation of Arithmetic Expressions . . . 55
  - 4.3.1 A Simple Example . . . 55
  - 4.3.2 The Shunting-Yard-Algorithm . . . 58
- 4.4 Benefits of Abstract Data Types . . . 62

5 Sets and Maps **64**
5.1 ADT Map . . . 64
5.2 Ordered Binary Trees . . . 66
5.2.1 Implementing Ordered Binary Trees in SetlX . . . 70
5.2.2 Analysis of the Complexity . . . 72
5.3 AVL Trees . . . 78
5.3.1 Implementing AVL-Trees in SetlX . . . 82
5.3.2 Analysis of the Complexity of AVL Trees . . . 86
5.3.3 Improvements . . . 90
5.4 Tries . . . 91
5.4.1 Insertion in Tries . . . 93
5.4.2 Deletion in Tries . . . 94
5.4.3 Complexity . . . 95
5.4.4 Implementing Tries in SetlX . . . 95
5.5 Hash Tables . . . 99
5.5.1 Further Reading . . . 106
5.6 Applications . . . 106

6 Priority Queues **107**
6.1 Formal Definition . . . 107
6.2 Heaps . . . 109
6.3 Implementation . . . 111
6.4 Heap-Sort . . . 114

7 Data Compression **118**
7.1 Motivation . . . 118
7.2 Huffman's Algorithm . . . 120
7.3 LZW Algorithm* . . . 125
7.3.1 Implementing the LZW algorithm in SetlX . . . 127

8 Graph Theory **132**
8.1 The Union-Find Problem . . . 132
8.1.1 A Tree-Based Implementation . . . 135
8.1.2 Controlling the Growth of the Trees . . . 136
8.1.3 Packing Union-Find as a Data Structure . . . 136
8.2 Minimum Spanning Trees . . . 138
8.2.1 Basic Definitions . . . 138
8.2.2 Kruskal's Algorithm . . . 139
8.3 Shortest Paths Algorithms . . . 141
8.3.1 The Bellman-Ford Algorithm . . . 141
8.3.2 Dijkstra's Algorithm . . . 143
8.3.3 Complexity . . . 145

9 Monte-Carlo Method **146**
9.1 Computing π . . . 146
9.2 Theory . . . 148
9.3 The Monty Hall Problem . . . 150
9.4 Permutations . . . 152

Chapter 1

Introduction

1.1 Motivation

The previous lecture during the past 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: Five years after their students have graduated, 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. 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 the value $f(x)$ for a given 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) *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.

   (b) The notion of a *recurrence relation* is the discrete analogue of the notion of a differential equation. Recurrence relations occur naturally when analysing the runtime of algorithms. We present the *Master Theorem* that can be used to describe the growth of functions that are specified via recurrence relations.

2. Abstract data types.

   Abstract data types are a means to describe the behaviour of an algorithm in a concise way. Furthermore, abstract data types are part of the foundations of *object-oriented programming*.

3. Sorting algorithms.

   Sorting algorithm are the algorithms that are most frequently used in practice. Furthermore, these algorithms are easy to understand and easy to analyse. Therefore, we start our discussion of algorithms and their complexity with sorting algorithms. In this lecture, we discuss the following sorting algorithms:

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

4. Associative arrays.

   Associative arrays are a means to represent a function. Nowadays, all modern programming languages provide associative array. Mathematically, an associative array is nothing more than a functional relation. We discuss various data structures that can be used to implement associative arrays efficiently.

5. Priority queues.

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

6. 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. Furthermore, we discuss Kruskal's algorithm for finding the *minimum spanning tree* of a graph.

7. 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, my goal is to enable you to think algorithmically: At the end of these lectures, you should be able to develop your own algorithms. This is a process that requires a lot of creativity on your side. However, once you are acquainted with a fair number of algorithms, you should be able to develop new 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 usually 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 arbitrariness 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 *Python* 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 difficult to follow.
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 intended to be the main source for my lecture. Additionally, I want to mention those books that have inspired me most.

1. *Robert Sedgewick*: Algorithms, 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 an excellent course on algorithms that is available at coursera.org. This course is based on this book. Furthermore, all the algorithms discussed in this book are implemented in *Java*, so reading this book also strengthens your knowledge of *Java*.

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

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

3. *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 can be viewed as a reference work. This book requires more mathematical sophistication on the side of its readers than any of the other books referenced here.

4. Einführung in die Informatik, written by *Heinz-Peter Gumm* and `Manfred Sommer` [GS13].

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

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

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 free of charge.

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 most gladly answer these questions, I should warn you that, 50% of the time, my answers will be flat out lies. The other 50%, my answers will be some random rubbish. Please bear that in mind when evaluating my answers.

1.7 A Request

Computer science is a very active field of research. Furthermore, my comprehension of the English language is improving steadily. Therefore, these lecture notes are constantly evolving and hence might contain typos or even bugs. If you find a problem, typo or otherwise, please take the time and either send me an email or a message on discord. My email address is

karl.stroetmann@dhbw-mannheim.de.

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

Finally, if you have any questions regarding the material presented in this course, you are welcome to ask questions either by email, discord or *Skype*. My *Skype* name is `karlstroetmann`. If you think that others might have the same question, it is best if you ask your question via discord. All your questions are welcome since they give me valuable feedback how ~~stupid you really are~~ to improve my lecture.

Chapter 2

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 illustrate the application of these notions, we show how to implement the computation of powers efficiently, i.e. we discuss how to evaluate the expression a^b for given $a, b \in \mathbb{N}$ in a way that is significantly faster than the naive approach.

2.1 Motivation

Sometimes it is necessary to have a precise understanding of the complexity 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.

This approach is problematic for a number of reasons.

1. It is very complicated.
2. The execution time of the basic operations is highly dependent 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. 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 today's 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	$\frac{2 \cdot n^2 + 7}{3 \cdot n^3 + 2 \cdot n^2 + 7}$
1	0.75000000000000
10	0.06454630495800
100	0.00662481908150
1000	0.00066622484855
10 000	6.6662224852 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 the size of the input 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 $\mathbb{N}$ yielding a positive real number is defined as:

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

Definition 1 ($\mathcal{O}(g)$) Assume $g \in \mathbb{R}_+^{\mathbb{N}}$ is given. Let us define the set of all functions that *grow at most as fast* as the function g as follows:

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

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. Consider a given function g and try to understand what $f \in \mathcal{O}(g)$ means. Informally, this means that the function f does not grow faster than the function g.

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

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

   is only required for those values of n that are bigger than or equal to 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.
2. 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 2.1 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 2.1 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.

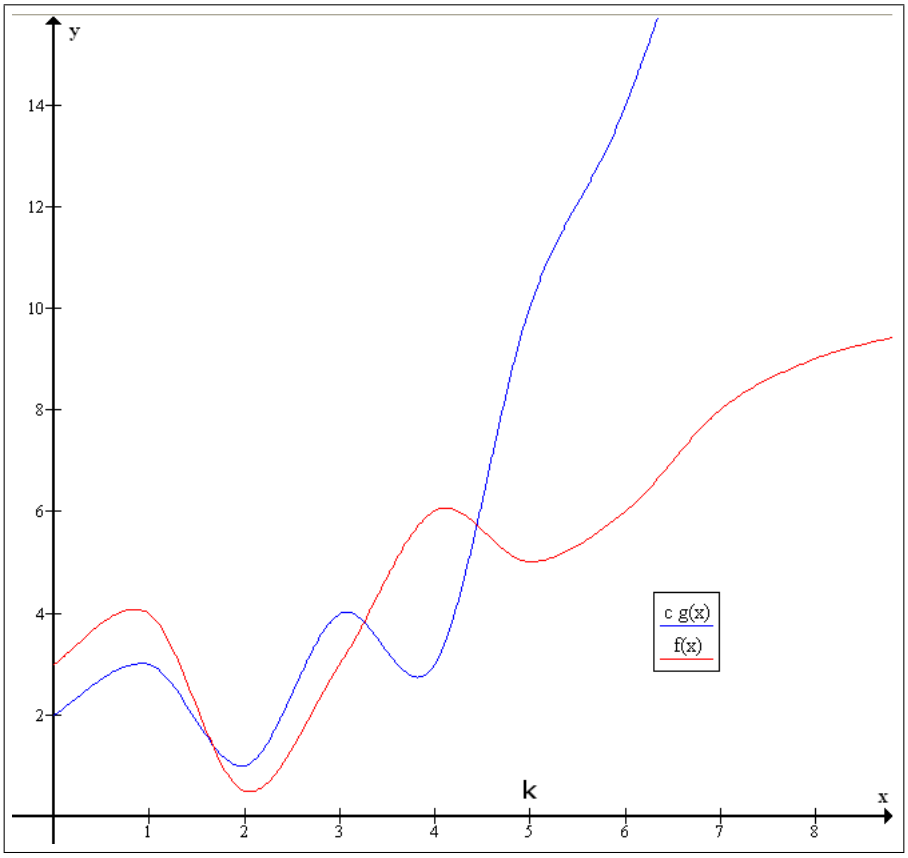

Figure 2.1: Example for $f \in \mathcal{O}(g)$.

We discuss some concrete 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 $n \in \mathbb{N}$ 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

$$1 \leq n \tag{2.1}$$

holds and we have to show that this implies

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3. \tag{2.2}$$

If we take the third power of both sides of the inequality (2.1) then we see that

$$1 \leq n^3 \tag{2.3}$$

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

$$7 \leq 7 \cdot n^3 \tag{2.4}$$

Furthermore, let us multiply the inequality (2.1) with the term $2 \cdot n^2$. This yields

$$2 \cdot n^2 \leq 2 \cdot n^3 \tag{2.5}$$

Finally, we obviously have

$$3 \cdot n^3 \leq 3 \cdot n^3 \tag{2.6}$$

Adding up the inequalities (2.4), (2.5), and (2.6) 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

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

We prove this claim by induction on n.

1. **Base case**: $n = 0$

   Obviously, $n = 0 \leq 1 = 2^0 = 2^n$ holds.

2. **Induction step**: $n \mapsto n + 1$

   By the induction hypothesis we have

   $$n \leq 2^n.$$

   Furthermore, a trivial induction shows that

   $$1 \leq 2^n.$$

   Adding these two inequalities yields

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

   □

Exercise 1:

(a) Prove that

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

(b) Prove that

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

(c) Prove that for every $\alpha \in \mathbb{N}$ we have

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

Hint:

1. Try to prove the following claim by induction on α: For every $\alpha \in \mathbb{N}$ there exists a number $c(\alpha)$ such that

   $$n^\alpha \leq c(\alpha) \cdot 2^n \quad \text{holds for all } n \in \mathbb{N}.$$

   In the induction step you will have to prove that there is a number $c(\alpha+1)$ such that

   $$n^{\alpha+1} \leq c(\alpha+1) \cdot 2^n \tag{$*$}$$

   holds. When proving this claim you may assume by induction hypothesis that for all $k \leq \alpha+1$ we have numbers $c(k)$ such that

   $$n^k \leq c(k) \cdot 2^n \quad \text{holds for all } n \in \mathbb{N}.$$

   You can prove the claim $(*)$ via a side induction on n.
2. The *binomial theorem* tells us that for all $n \in \mathbb{N}$ and all $a, b \in \mathbb{R}$ the equation

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

   holds. Here, the expression $\binom{n}{k}$ is read as n *choose* k and is defined for all $n \in \mathbb{N}$ and all $k \in \{0, 1, 2, \cdots, n\}$ via

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

   The binomial theorem should be used to expand the term $(n+1)^{\alpha+1}$ that occurs in the induction step of the side induction. ◇

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

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

Proof: Let us define $k := 0$ and $c := 1$. Then our claim follows immediately from the inequality

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

Proposition 3 (Multiplication with Constants)
Assume that we have functions $f, g\colon \mathbb{N} \to \mathbb{R}_+$ and a number $d \in \mathbb{R}_+$. Then we have

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

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

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

holds. If we multiply the inequality involving $g(n)$ with d, we get

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

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

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

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 4 (Addition) Assume that $f, g, h\colon \mathbb{N} \to \mathbb{R}_+$. 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 $k_1, k_2 \in \mathbb{N}$ and $c_1, c_2 \in \mathbb{R}$ such that both

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \to f(n) \leq c_1 \cdot h(n)\big) \quad \text{and}$$

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

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 2: Assume that $f_1, f_2, h_1, h_2\colon \mathbb{N} \to \mathbb{R}_+$. Prove that

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

Exercise 3: Assume that $f_1, f_2, h_1, h_2\colon \mathbb{N} \to \mathbb{R}_+$. Prove or refute the claim that

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

Proposition 5 (Transitivity) Assume $f, g, h\colon \mathbb{N} \to \mathbb{R}_+$. Then we have

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

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

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

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

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

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 6 (Limit Proposition) Assume that $f, g\colon \mathbb{N} \to \mathbb{R}_+$. 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

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

Since the limit exists by our assumption, we know that

$$\forall \varepsilon \in \mathbb{R}_+ : \exists k \in \mathbb{R} : \forall n \in \mathbb{N} : \left(n \geq k \to \left| \frac{f(n)}{g(n)} - \lambda \right| < \varepsilon \right)$$

Since this is valid for all positive values of ε, let us define $\varepsilon := 1$. Then there exists a number $k \in \mathbb{N}$ such that for all $n \in \mathbb{N}$ satisfying $n \geq k$ the inequality

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

holds. Let us multiply this inequality with $g(n)$. As $g(n)$ is positive, 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

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

holds. Combining the previous two inequalities yields

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

Therefore, we define

$$c := 1 + \lambda$$

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

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

Proof: We will show that

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = 0 \tag{2.7}$$

is true. Then the claim is an immediate consequence of the limit proposition. According to L'Hôpital's rule[1], the limit can be computed as follows:

[1] Basically, L'Hôpital's rule states that provided the limit $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists and $g'(x) \neq 0$, we have

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}.$$

Here f' and g' denote the derivatives of f and g. We will discuss L'Hôpital's rule in the lectures on analysis.

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

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

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

In the lecture on analysis we will see 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 4: Prove that $\sqrt{n} \in \mathcal{O}(n)$ holds. ◇

Exercise 5: 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

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

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 6:

(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

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

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

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

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 shows 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 $7 \cdot n + \big(\log_2(n)\big)^2 \in \mathcal{O}(n)$.

(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

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

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

2.2 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

$$\mathit{count}_A(n) = n \cdot \log_2(n) + \mathcal{O}(n) \quad \text{and} \quad \mathit{count}_B(n) = \tfrac{3}{2} \cdot n \cdot \log_2(n) + \mathcal{O}(n)$$

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

2.3 Case Study: Efficient Computation of Powers

Let us study an example to clarify the notions introduced so far. Consider the program shown in Figure 2.2. 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 2.2: Naive computation of m^n for $m, n \in \mathbb{N}$.

This program is obviously correct. The computation of m^n requires $n - 1$ multiplications if the function `power` is implemented as shown in Figure 2.2. 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 2.3. 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 2.3 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 2.3: Computation of m^n for $m, n \in \mathbb{N}$.

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 where 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 $k \in \mathbb{N}$ such that $n = 2 \cdot k$ and therefore $n/2 = k$. Then, we have the following:

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

   (b) $n \,\%\, 2 = 1$, therefore n is odd.
   Then there exists a number $k \in \mathbb{N}$ 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:

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

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 $l \in \mathbb{N}$ 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:

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

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

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

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

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

This claim is readily established via a trivial induction. Plugging in the definition $b_l = a_{2^l-1}$ we see that 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. Hence there must exist an $l \in \mathbb{N}$ 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 $\texttt{power}(m,n)$.

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 $\texttt{power}(m, 0)$ 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:

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

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

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

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

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

Obviously, the solution is

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

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 value 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 two numbers depends on the size of these numbers.

Exercise 7: Implement a procedure prod that multiplies two numbers: For given natural numbers m and n, the expression $\texttt{prod}(m, n)$ 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 $\texttt{prod}(m, n)$. This estimate should make use of the big $\mathcal{O}$ notation. $\diamond$

2.4 The Master Theorem

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

Theorem 7 (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 $f : \mathbb{N} \to \mathbb{R}_+$ 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*[2] of n by β.

Then we have the following:

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

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. Furthermore, in order to simplify our exposition even further, we assume that the recurrence relation for f has the form

$$f(n) = \alpha \cdot f\left(n \backslash \beta\right) + n^\delta,$$

i.e. instead of adding the term $\mathcal{O}\big(n^\delta\big)$ we just add n^δ. These simplifications do not change the proof idea. We start the proof by defining

$$a_k := f(n) = f\big(\beta^k\big).$$

Then the recurrence relation for the function f is transformed into a recurrence relation for the sequence a_k as follows:

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

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

$$a_k = \alpha \cdot a_{k-1} + \gamma^k.$$

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

$$a_{k-1} = \alpha \cdot a_{k-2} + \gamma^{k-1}.$$

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

[2] 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. In SetlX, integer division is available via the backslash operator "\".

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

We observe that

$$a_{k-2} = \alpha \cdot a_{k-3} + \gamma^{k-2}$$

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

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

Proceeding in this way we arrive at the general formula

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

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

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

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 $\sum\limits_{j=0}^{k-1} \big(\frac{\alpha}{\gamma}\big)^j$ 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

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

   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

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

Therefore, we have

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

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 analyse 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 8: 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 9: 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. ◇

2.5 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 8 ($\Omega(g)$) Assume $g \in \mathbb{R}_+^{\mathbb{N}}$ is given. Let us define the set of all functions that grow at least as fast as the function g as follows:

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

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 9 ($\Theta(g)$) Assume $g \in \mathbb{R}_+^{\mathbb{N}}$ 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 $\mathit{count}_A(n)$ 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

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

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

$$\tfrac{3}{2} \cdot \mathit{count}_A(n) \sim \mathit{count}_B(n)$$

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

2.6 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 $\mathcal{O}$ notation, while chapter 4 gives a more general version of the master theorem together with a detailed proof.

Chapter 3

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 and the relation $\leq$ is called a *total order* or a *linear order*.

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

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

   but obviously $\mathrm{Smith} \neq \mathrm{Robinson}$.

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 shows 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 10 (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,

$$\forall x, y \in S: \big(x \preceq y \vee y \preceq x\big) \qquad \textit{(linearity)}$$

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 linear 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

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

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 linear 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:
   $$\mathit{count}(x, l) := \#\big\{i \in \{1, \cdots, \#l\} \mid l[i] = x\big\}.$$
   Sometimes, this second requirement is changed as follows:
   $$\forall x \in s: \mathit{count}(x, s) \leq 1 \wedge \forall x \in M: \big(\mathit{count}(x, l) > 0 \rightarrow \mathit{count}(x, s) = 1\big).$$
   Hence, in this case we require that the sorted list s does not contain duplicate elements. Of course, an object x should only occur in s if it also occurs in l. If we change the second requirement in this way, then the main purpose of sorting is to remove duplicate elements from a list. This is actually one common application of sorting in practice. The reason this application is so common is the following: A list that contains every element at most once can be viewed as representing a set.

   Exercise 10: Assume a list s is sorted and contains every object at most once. Develop an efficient algorithm for testing whether a given object x is a member of the list s.
   Hint: Try to develop an algorithm that follows the *divide-and-conquer* paradigm.

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.

3.1 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 | L];
11          case [y | R] | !(x <= y) : return [y | insert(x, R)];
12      }
13  };
```

Figure 3.1: Implementing *insertion sort* in SetlX.

Figure 3.1 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 elements but the first element of L.

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.

3.1.1 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 3.2. In comparison to Figure 3.1, we have dropped the test

"`!(x <= y)`"

from line 5 since it is unnecessary: If control ever 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 3.2: 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 L 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 3.2 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 x 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`$(x, \texttt{sort(R)})$ 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`(R) 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 up to $n-1$ 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 possibly 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.

3.2 Selection Sort

Next, we discuss selection sort. In order to sort a given list L this algorithm 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 3.3 on page 32 shows an implementation of selection sort in SetlX. There is 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, if the algorithm tries to delete an element from the empty list, something must have gone wrong. 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 $L$");
11          case [y | R] | y == x: return R;
12          case [y | R] | y != x: return [y] + delete(x, R);
13      }
14  };
```

Figure 3.3: Implementing *selection sort* in SetlX.

3.2.1 Complexity of Selection Sort

In order to be able to analyse 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 now contains only n elements, is sorted recursively. We need b_n evaluations of the comparison 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*. Furthermore, in many practical applications of sorting the lists that have to be sorted are already partially sorted and have only a few elements that are out of place. In these cases, *insertion sort* can, in fact, be more efficient than any other sorting algorithm.

3.3 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.

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 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 3.4 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

$$\langle L_1, L_2 \rangle \quad \text{where } L_1 = \texttt{split}_1(L) \text{ and } L_2 = \texttt{split}_2(L).$$

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 3.4:

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`.

```
sort := procedure(L) {
    if (#L < 2) {
        return L;
    }
    [L1, L2] := split(L);
    return merge(sort(L1), sort(L2));
};
split := procedure(L) {
    match (L) {
        case []         : return [ [], [] ];
        case [x]        : return [ [x], [] ];
        case [x, y | R] : [R1, R2] := split(R);
                          return [ [x] + R1, [y] + R2];
    }
};
merge := procedure(L1, L2) {
    match ([L1, L2]) {
        case [      [],       L2] : return L2;
        case [      L1,       []] : return L1;
        case [[x | R1], [y | R2]] : if (x <= y) {
                                        return [x] + merge(R1, L2);
                                    } else {
                                        return [y] + merge(L1, R2);
                                    }
    }
};
```

Figure 3.4: The *merge sort* algorithm implemented in SETLX.

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:

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

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

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

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:

   $$\texttt{split}(R) = \langle R_1, R_2 \rangle \rightarrow \texttt{split}\big([x, y|R]\big) = \big\langle [x|R_1], [y|R_2] \big\rangle.$$

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

3.3.1 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 analyse 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 11: 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 $n\backslash 2$, 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 have $c_0 = 0$. 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

$$a_n = n \cdot \log_2(n).$$

3.3.2 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 quite 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 3.5 on page 37 presents an implementation of *merge sort* that treats the list L that is to be sorted as an array.

```
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]; idx1 += 1;
20         } else {
21             L[i] := A[idx2]; idx2 += 1;
22         }
23         i += 1;
24     }
25     while (idx1 < middle) {
26         L[i] := A[idx1]; idx1 += 1; i += 1;
27     }
28     while (idx2 < end   ) {
29         L[i] := A[idx2]; idx2 += 1; i += 1;
30     }
31     assert( idx1 == middle && idx2 == end,
32             "idx1 == middle && idx2 == end failed");
33 };
```

Figure 3.5: An array based implementation of *merge sort*.

We discuss the implementation shown in Figure 3.5 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 the entire list `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 has at most one element. In this case, there is nothing to do as any such list is already sorted.

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 about 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.

3.3.3 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 3.6: A non-recursive implementation of *merge sort*.

The implementation of *merge sort* shown in Figure 3.5 on page 37 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 considerably less efficient than an iterative implementation. Therefore, we present an iterative implementation of *merge sort* in Figure 3.6 on page 39.

Instead 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 pairs of these lists 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 the list 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.

3.3.4 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

   $$[x] + \mathtt{R} \quad \text{and} \quad \mathtt{L}_1 + [y] + \mathtt{L}_2$$

   and if y is less than x, then all elements of the list $\mathtt{L}_1$ 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. Originally, Tim Peters developed Timsort for the programming language *Python*. Today, *Timsort* is also 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.

3.4 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:

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

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`:

   $$\mathtt{partition}(x, r) = \langle s, b \rangle.$$

   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 $\mathtt{sort}(s)$, $[x]$, and $\mathtt{sort}(b)$:

   $$\mathit{partition}(x, r) = \langle s, b \rangle \rightarrow \mathit{sort}([x|r]) = \mathit{sort}(s) + [x] + \mathit{sort}(b).$$

Figure 3.7 on page 41 shows how these equations can be implemented in SetlX.

```
1   sort := procedure(l) {
2       match (l) {
3           case []   : return [];
4           case [x|r]:
5                [s,b] := partition(x, r);
6                return sort(s) + [x] + sort(b);
7       }
8   };
9   partition := procedure(p, l) {
10      match (l) {
11          case []   : return [ [], [] ];
12          case [x|r]: [ r1, r2 ] := partition(p, r);
13                      if (x <= p) {
14                          return [ [x|r1], r2 ];
15                      }
16                      return [ r1, [x|r2] ];
17      }
18  };
```

Figure 3.7: The *quick sort* algorithm.

3.4.1 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.

Worst Case Complexity

Let us denote the number of comparisons needed to evaluate $\texttt{sort}(l)$ 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 the execution of $\texttt{partition}(x, r)$ in line 4 of 3.7 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 is 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 are smaller than or equal to the pivot element x 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 the list b of those elements, that are bigger than x is $n-i$. Therefore, if $\#s = i$, then on average we need

$$d_i + d_{n-i}$$

comparisons to sort the lists s and b recursively. If we take the average over all possible values of $i = \#s$ then, since $i \in \{0, 1, \cdots, n\}$ and this set has $n+1$ elements, 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)
```

since we need to compare every element of l except the pivot x With the pivot x. In order to simplify the recurrence relation (1) we note that

$$\begin{aligned} \sum_{i=0}^{n} a_{n-i} &= a_n + a_{n-1} + \cdots + a_1 + a_0 \\ &= a_0 + a_1 + \cdots + a_{n-1} + a_n \\ &= \sum_{i=0}^{n} a_i \end{aligned}$$

holds for any sequence $(a_n)_{n \in \mathbb{N}}$. 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

$$d_n = n - 1 + \frac{2}{n} \cdot \sum_{i=0}^{n-1} d_i. \tag{3}$$

Next, we multiply equation (3) with n and equation (2) with $n+1$. This yields the equations

$$n \cdot d_n = n \cdot (n-1) + 2 \cdot \sum_{i=0}^{n-1} d_i, \tag{4}$$

$$(n+1) \cdot d_{n+1} = (n+1) \cdot n + 2 \cdot \sum_{i=0}^{n} d_i. \tag{5}$$

We take the difference of equation (5) and (4) and note that the summations cancel except for the term $2 \cdot d_n$. This leads to

$$(n+1) \cdot d_{n+1} - n \cdot d_n = (n+1) \cdot n - n \cdot (n-1) + 2 \cdot d_n.$$

This equation can be simplified as

$$(n+1) \cdot d_{n+1} = (n+2) \cdot d_n + 2 \cdot n.$$

In order to exhibit the true structure of this equation we divide by $(n+1) \cdot (n+2)$ and get

$$\frac{1}{n+2} \cdot d_{n+1} = \frac{1}{n+1} \cdot d_n + \frac{2 \cdot n}{(n+1) \cdot (n+2)}. \tag{6}$$

We proceed by computing the partial fraction decomposition of the fraction

$$\frac{2 \cdot n}{(n+1) \cdot (n+2)}.$$

In order to so, we use the ansatz

$$\frac{2 \cdot n}{(n+1) \cdot (n+2)} = \frac{\alpha}{n+1} + \frac{\beta}{n+2}.$$

Multiplying this equation with $(n+1) \cdot (n+2)$ yields

$$2 \cdot n = \alpha \cdot (n+2) + \beta \cdot (n+1).$$

Grouping like terms, this can be simplified as follows:

$$2 \cdot n = (\alpha + \beta) \cdot n + 2 \cdot \alpha + \beta.$$

Since this has to hold for every $n \in \mathbb{N}$ we must have:

$$\begin{aligned} 2 &= \alpha + \beta \\ 0 &= 2 \cdot \alpha + \beta \end{aligned}$$

If we subtract the first equation from the second equation we arrive at $\alpha = -2$. Substituting this into the first equation gives $\beta = 4$. Hence, the equation (6) can be written as

$$\frac{1}{n+2} \cdot d_{n+1} = \frac{1}{n+1} \cdot d_n - \frac{2}{n+1} + \frac{4}{n+2}.$$

In order to simplify this equation, let us define

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

Then the last equation is simplified to

$$a_{n+1} = a_n - \frac{2}{n+1} + \frac{4}{n+2}.$$

Substituting $n \mapsto n-1$ 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

$$H_n = \ln(n) + \gamma + \mathcal{O}\Big(\frac{1}{n}\Big).$$

We discuss this approximation in the lecture on calculus after we have covered integration. In the formula approximating the harmonic number H_n, γ is the Euler-Mascheroni constant and has the value

$$\gamma = 0.5772156649\cdots.$$

Therefore, we have found the following approximation for a_n:

$$a_n = -\frac{4 \cdot n}{n+1} + 2 \cdot \ln(n) + \mathcal{O}(1) = 2 \cdot \ln(n) + \mathcal{O}(1), \quad \text{as} \quad \frac{4 \cdot n}{n+1} \in \mathcal{O}(1).$$

Since we have $d_n = (n+1) \cdot a_n$ we can conclude that

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

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

$$2 \cdot \ln(2) \cdot n \cdot \log_2(n) + \mathcal{O}(n)$$

comparisons and hence on average *quick sort* needs $2 \cdot \ln(2) \approx 1.39$ times as many comparisons as *merge sort*.

3.4.2 Implementing *Quick Sort* for Arrays

Next, we show how *quick sort* is implemented using arrays instead of lists. Figure 3.8 on page 45 shows this implementation.

```
1   sort := procedure(rw lst) {
2       quickSort(1, #lst, lst);
3   };
4   quickSort := procedure(a, b, rw lst) {
5       if (b <= a) {
6           return; // at most one element, nothing to do
7       }
8       m := partition(a, b, lst);  // m is the split index
9       quickSort(a, m - 1, lst);
10      quickSort(m + 1, b, lst);
11  };
12  partition := procedure(start, end, rw lst) {
13      pivot := lst[end];
14      left  := start - 1;
15      for (idx in [start..end-1]) {
16          if (lst[idx] <= pivot) {
17              left += 1;
18              swap(left, idx, lst);
19          }
20      }
21      swap(left + 1, end, lst);
22      return left + 1;
23  };
24  swap := procedure(x, y, rw lst) {
25      [ lst[x], lst[y] ] := [ lst[y], lst[x] ];
26  };
```

Figure 3.8: An implementation of *quick sort* based on arrays.

1. Contrary to the array based implementation of *merge sort*, we do not need an 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 `lst`.
   (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) `lst` is the array that needs to be sorted.

   Calling `quickSort(a, b, lst)` sorts the subarray

   $$\texttt{lst[a]}, \texttt{lst[a+1]}, \cdots, \texttt{lst[b]}$$

   of the array `lst`, i. e. after that call we expect to have

   $$\texttt{lst[a]} \preceq \texttt{lst[a+1]} \preceq \cdots \preceq \texttt{lst[b]}.$$

   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 `lst`: All elements that are less or equal than the *pivot element* `lst[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) $\forall i \in \{\texttt{start}, \cdots, \texttt{left}\} : \texttt{lst}[i] \leq \texttt{pivot}$.
   All elements in the subarray `lst[start..left]` are less or equal than the pivot element.

   (b) $\forall i \in \{\texttt{left} + 1, \cdots, \texttt{idx} - 1\} : \texttt{pivot} < \texttt{lst}[i]$.
   All elements in the subarray `lst[left+1..idx-1]` are greater than the pivot element.

   (c) $\texttt{pivot} = \texttt{lst}[\texttt{end}]$
   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 `lst[idx]` is less than the pivot element, it need to become part of the subarray `lst[start..left]`. In order to achieve this, it is placed at the position `lst[left+1]`. The element that has been at that position is part of the subarray `lst[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 `lst[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.

3.4.3 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 `lst` 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 the worst case performance is much less likely 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 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].
   ```

[1] It is not always greater than the pivot element because the subarray `lst[left+1..idx-1]` might well be empty.

4. In 2009, Vladimir Yaroslavskiy introduced *dual pivot quick sort* [Yar09]. His paper can be downloaded at the following address:

   http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf

   The main idea of Yaroslavskiy is to use two pivot elements x and y. For example, we can define

   $$x := \texttt{lst}[1] \quad \text{and} \quad y := \texttt{lst}[\#\texttt{lst}],$$

   i. e. we take x as the first element of `lst`, while y is the last element of `lst`. Then, the list `lst` 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 3.9 on page 47 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

```
1   sort := procedure(lst) {
2       match (lst) {
3       case []     : return [];
4       case [x]    : return [x];
5       case [x,y|r]:
6            [p1, p2] := [min({x,y}), max({x,y})];
7            [lst1,lst2,lst3] := partition(p1, p2, r);
8            return sort(lst1) + [p1] + sort(lst2) + [p2] + sort(lst3);
9       }
10  };
11  partition := procedure(p1, p2, lst) {
12      match (lst) {
13      case []   : return [ [], [], [] ];
14      case [x|r]: [ r1, r2, r3 ] := partition(p1, p2, r);
15                  if (x < p1) {
16                      return [ [x|r1], r2, r3 ];
17                  } else if (x <= p2) {
18                      return [ r1, [x|r2], r3 ];
19                  } else {
20                      return [ r1, r2, [x|r3] ];
21                  }
22      }
23  };
```

Figure 3.9: A list based implementation of *dual pivot quick sort*.

Exercise 12: Implement a version of *dual pivot quick sort* that uses arrays instead of lists.

3.5 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

$$\Omega\big(n \cdot \log_2(n)\big)$$

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.
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 hypothesis 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 one test 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

$$\mathit{card}\big(\{0, 1, 2, \cdots, 2^k - 1\}\big) = 2^k.$$

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

$$\log_2(n!) = n \cdot \log_2(n) + \Theta(n).$$

Using this approximation we get

$$k \geq n \cdot \log_2(n) + \Theta(n).$$

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 4

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.

4.1 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 $\{\mathit{Element}\}$.
3. The set of function symbols is
$$\{\mathit{Stack}, \mathit{push}, \mathit{pop}, \mathit{top}, \mathit{isEmpty}\}.$$
4. The type specifications of these function symbols are given as follows:
   (a) $\mathit{stack} : \mathit{Stack}$
   The function *stack* takes no arguments and produces an empty stack. Therefore, this function is a constructor. Intuitively, the function call $\mathit{stack}()$ creates an empty stack.
   (b) $\mathit{push} : \mathit{Stack} \times \mathit{Element} \to \mathit{Stack}$
   The function call $\mathit{push}(S, x)$ puts the element x on top of the stack S. In the following, we will use an object oriented notation and write $S.\mathit{push}(x)$, instead of $\mathit{push}(S, x)$.
   (c) $\mathit{pop} : \mathit{Stack} \to \mathit{Stack}$
   The function call $S.\mathit{pop}()$ removes the first element from the stack S.
   (d) $\mathit{top} : \mathit{Stack} \to \mathit{Element}$
   The function call $S.\mathit{top}()$ returns the element that is on top of the stack S.
   (e) $\mathit{isEmpty} : \mathit{Stack} \to \mathbb{B}$
   The function call $S.\mathit{isEmpty}()$ checks whether the stack S is empty.

The intuition that we have of a stack is captured by the following axioms.

1. $\mathit{stack}().\mathit{top}() = \Omega$
   Here, Ω denotes the undefined value[1]. The expression $\mathit{stack}()$ creates an empty stack. Therefore, the given axiom expresses the fact that there is no element on top of the empty stack.
2. $S.\mathit{push}(x).\mathit{top}() = x$
   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. $\mathit{stack}().\mathit{pop}() = \Omega$
   Trying to remove an element from the empty stack yields an undefined result.
4. $S.\mathit{push}(x).\mathit{pop}() = S$
   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. $\mathit{stack}().\mathit{isEmpty}() = \texttt{true}$
   This axiom expresses the fact that the stack created by the function call $\mathit{stack}()$ is empty.

[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: As it cannot be defined, it is undefined. ☺

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

   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 behaviour 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.

4.2 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 signatures. 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 4.1 shows an 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 4.1 this name is `stack`.

   In SetlX, every class also defines a constructor which 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`.

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 4.1 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

```
class stack() {
    mStackElements := [];

  static {
    push := procedure(e) {
        this.mStackElements += [e];
    };
    pop := procedure() {
        assert(#mStackElements > 0, "popping empty stack");
        this.mStackElements := mStackElements[1 .. -2];
    };
    top := procedure() {
        assert(#mStackElements > 0, "top of empty stack");
        return mStackElements[#mStackElements];
    };
    isEmpty := procedure() {
        return mStackElements == [];
    };
    f_str := procedure() {
        result := convert(this);
        dashes := "-" * #result;
        return join([dashes, result, dashes], "\n");
    };
    convert := procedure(s) {
        if (s.isEmpty()) {
            return "|";
        }
        top := s.top();
        s.pop();
        return convert(s) + " " + top + " |";
    };
  }
}

createStack := procedure(l) {
    result := stack();
    n := #l;
    for (i in [n, n-1 .. 1]) {
        result.push(l[i]);
    }
    return result;
};
```

Figure 4.1: An array based implementation of the ADT *Stack* in SetlX.

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:

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

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:

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

   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. These 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, 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 30 and finally the element `top` is appended to this string.

(b) The method `f_str` creates a line of dashes in line 21. This line has the same length as the string produced by `convert` 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 33. 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.

4.3 Evaluation of Arithmetic Expressions

Next, in order to demonstrate the usefulness of stacks, we show how *arithmetic expressions* can be evaluated using stacks. To this end, we present the *shunting-yard algorithm* for parsing arithmetic expressions. An *arithmetic expression* is a string that is made up of numbers and the operator symbols "+", "-", "*", "/", "%", and "**". Here x % y denotes the modulus operator and x ** y denotes the power x^y. Furthermore, arithmetic expressions can use the parentheses "(" and ")".

Formally, the set of arithmetic expressions is defined by induction.

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.

4.3.1 A Simple Example

Before we dive into to the details of the shunting-yard algorithm, we present a simple example. Consider the arithmetic expression

"`1 + 2 * 3 - 4`".

First, this string is transformed into the list of *tokens*

`[1, "+", 2, "*", 3, "-" 4]`.

A *token* is either a number, an operator symbol, or a parenthesis. Notice that the space symbols that have been present in the original arithmetic expression string have been discarded. This list is processed from left to right, one token at a time. In order to process this list, we use three stacks.

1. The *token list* contains all the tokens of the arithmetic expression. It is initialized with the list of tokens resulting from the input string. Although the token list is really just a list we will represent this list as a stack and call this list the *token stack*. The first token of the arithmetic expression is on top of this stack.

2. The *argument stack* contains only numbers and is initially empty.

3. The *operator stack* contains only operator symbols and parentheses and is also initially empty.

The evaluation of `1 + 2 * 3 - 4` proceeds as follows:

1. In the beginning, the token stack contains the tokens of 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.

4.3.2 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 containing arithmetic operators. These operators are represented as strings.

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 the 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 using the arguments that have been previously removed from the argument stack. The resulting number is then pushed onto the argument stack.

An implementation of this algorithm in SetlX is shown in the Figures 4.2 and 4.3 on the following pages. We start our discussion of the class `calculator` by inspecting the method `evalBefore` that is defined in line 32. 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 4.1 on page 58 lists the precedences of our operators. This table is coded as the binary relation `prec` in line 33.

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

Table 4.1: 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 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`.

```
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 == ")" || evalBefore(stackOp, nextOp))
23                  popAndEvaluate();
24                  mTokenStack.push(nextOp);
25              } else {
26                  mOperators.push(nextOp);
27              }
28          }
29          while (!mOperators.isEmpty()) { popAndEvaluate(); }
30          return mArguments.top();
31      };
32      evalBefore := procedure(stackOp, nextOp) {
33          prec := {["+",1],["-",1],["*",2],["/",2],["%",2],["**",3]};
34          if (stackOp == "(") { return false;  }
35          if (prec[stackOp] > prec[nextOp]) {
36              return true;
37          } else if (prec[stackOp] == prec[nextOp]) {
38              if (stackOp == nextOp) {
39                  return stackOp in { "+", "-", "*", "/", "%" };
40              }
41              return true;
42          }
43          return false;
44      };
```

Figure 4.2: The class `calculator`, part 1.

```
45        popAndEvaluate := procedure() {
46            rhs := mArguments.top(); mArguments.pop();
47            lhs := mArguments.top(); mArguments.pop();
48            op  := mOperators.top(); mOperators.pop();
49            match (op) {
50                case "+" : result := lhs + rhs;
51                case "-" : result := lhs - rhs;
52                case "*" : result := lhs * rhs;
53                case "/" : result := lhs / rhs;
54                case "%" : result := lhs % rhs;
55                case "**": result := lhs ** rhs;
56                default: abort("ERROR: *** Unknown Operator *** $op$");
57            }
58            mArguments.push(result);
59        };
60      }
61    }
62
63    c := calculator("1+2*3**4-5/2");
64    c.evaluate();
```

Figure 4.3: The class `calculator`, part 2.

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 34.

2. If the precedence of `stackOp` is higher than the precedence of `nextOp`, we return `true` in line 36.

3. If the precedences of `stackOp` and `nextOp` are identical, there are two cases:

   (a) If both operators 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 39.

   (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 4.3 on page 60 shows the implementation of the method `popAndEvaluate`. This method works as follows:

1. It takes an operator from the operator stack (line 48),
2. it fetches the arguments of this operator from the argument stack (line 46 and line 47),
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 4.2.

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.

Exercise 13: In the following exercise, your task is to extend the program for evaluating arithmetic expressions in three steps.

(a) Extend the program discussed in these lecture notes so that it can also be used to evaluate arithmetic expressions containing the function symbols

`sqrt`, `exp`, and `log`.

(b) Extend the given program so that the arithmetic expressions may also contain the strings "`e`" and "`Pi`", where "`e`" stands for Euler's number while "`Pi`" stands for the mathematical constant π defined as the ratio of the circumference of a circle to its diameter.

Note that these constants can be accessed in SetlX via the function `mathConst`. For example, the function calls

`mathConst("e")` and `mathConst("Pi")`

yield the result 2.718281828459045 and 3.141592653589793 respectively.

(c) Extend the program so that it can be used to calculate a zero for a given function in a given interval $[a, b]$ provided that $f(a) < 0$ and $f(b) > 0$. ◇

4.4 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 have 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 the 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 implementation with an array based implementation of stacks that is more efficient. 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 (known as *profiling*) spot those data types that are performance bottlenecks.

(d) Finally, the implementations of those 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 our 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 and optimize.

Chapter 5

Sets and Maps

During the first term we have seen how important sets and functional relations are. In the following, *functional relations* will be called *maps*. In computer science, maps are also known as *associative arrays*, *dictionaries*, or *symbol tables*. In this chapter we show how sets and maps can be implemented efficiently. We confine our attention to the implementation of maps. The reason is that a set M can always be represented by its *characteristic function*: If M is a set, then the characteristic function χ_M is defined such that

$$x \in M \Leftrightarrow \chi_M(x) = \texttt{true}$$

holds true. In order to implement a set M we can therefore implement its characteristic function as a map. The rest of this chapter is organized as follows:

1. We begin with the definition of the abstract data type of a *map*.

   Following this definition we present several different implementations of maps.

2. We start our discussion with *ordered binary trees*. These trees can be used to implement maps, provided the keys are ordered. The average complexity of inserting an element into an ordered binary tree is logarithmic. Unfortunately, the worst case complexity is linear in the size of the ordered binary tree.

3. Next, we discuss *balanced ordered trees*. In the case of balanced ordered trees the complexity of insertion is always logarithmic.

4. After that, we discuss so called *tries*. These can be used as maps if the keys to be stored in the map are strings.

5. Finally, we discuss *hash tables*. Hash tables provide another way to implement a map. Although I personally think that hash tables are a bit overrated, they are in wide spread use and therefore every computer scientist should have a good understanding of their inner workings.

5.1 The Abstract Data Type *Map*

Many applications require the efficient maintenance of some mapping of *keys* to *values*. For example, in order to implement a software analogue of a telephone book we have to be able to associate numbers with names. In this case, the name of a person is regarded as a *key* and the telephone number is the value that gets associated with the key. The most important functions provided by a telephone directory are the following:

1. *Lookup*: We have to be able to look up a given name and return the telephone number associated with this name.

2. *Insertion*: We need to be able to insert a new name and the corresponding telephone number into our directory.
3. *Deletion*: The final requirement is that it has to be possible to delete names from the directory.

Definition 11 (Map)
The abstract data type of a map *is defined as follows:*

1. *The name is Map.*
2. *The set of type parameters is* $\{Key, Value\}$.
3. *The set of function symbols is*
   $$\{map, find, insert, delete\}.$$
4. *The signatures of these function symbols are as follows:*
   (a) $map : Map$
   Calling $map()$ *generates a new empty map. Here, an empty map is a map that does not store any keys.*
   (b) $find : Map \times Key \to Value \cup \{\Omega\}$
   The function call
   $$m.find(k)$$
   checks whether the key k *is stored in the map* m*. If this is the case, the value associated with this key is returned, otherwise the function call returns the undefined value* Ω*.*
   (c) $insert : Map \times Key \times Value \to Map$
   The function call
   $$m.insert(k, v)$$
   takes a key k *and an associated value* v *and stores this information into the map* m*. If the map* m *already stores a value associated with the key* k*, this value is overwritten.*
   The function call returns the resulting map.
   (d) $delete : Map \times Key \to Map$
   The function call
   $$m.delete(k)$$
   removes the key k *and any value associated with* k *from the map* m*. If the map* m *does not contain a value for the key* k*, then the map is returned unchanged.*
   The function call returns the new map.
5. *The behaviour of a map is specified via the following axioms.*
   (a) $map().find(k) = \Omega$.
   Calling $map()$ *generates an empty map which does not have any keys stored. Hence, looking up any key in the empty map will just return the undefined value.*
   (b) $m.insert(k, v).find(k) = v$.
   If a value v *is inserted for a key* k*, then when we look up this key* k *the corresponding value* v *will be returned.*
   (c) $k_1 \neq k_2 \to m.insert(k_1, v).find(k_2) = m.find(k_2)$.
   If a value is inserted for a key k_1*, then this does not change the value that is stored for any key* k_2 *different from* k_1*.*
   (d) $m.delete(k).find(k) = \Omega$.
   If the key k *is deleted, then afterwards we won't find this key anymore.*

(e) $k_1 \neq k_2 \rightarrow m.\mathit{delete}(k_1).\mathit{find}(k_2) = m.\mathit{find}(k_2)$,
If we delete a key k_1 and then try to look up the information stored under a key k_2 that is different from k_1, we will get the same result that we would have gotten if we had searched for k_2 before deleting k_1. ◇

In SetlX it is very easy to implement the abstract data type *map*. We just have to realize that a map is the same thing as a function and we have already seen that functions can be interpreted as binary relations. Now if r is a binary relation that, for a given key k, contains exactly one pair of the form $[k, v]$, then the expression

$$r[k]$$

returns the value v. On the other hand we can insert the pair $[k, v]$ into the relation r by writing

$$r[k] := v;$$

and in order to delete the value stored for a key k it is sufficient to assign the undefined value Ω to the key k as follows:

$$r[k] := \mathtt{om};.$$

This value Ω is also the value that is returned by the expression $r[k]$ if the binary relation r has no pair of the form $[k, v]$. Figure 5.1 presents an implementation of maps along these lines.

```
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 5.1: A trivial implementation of the abstract data type *Map* in SetlX.

5.2 Ordered Binary Trees

If the set *Key* is linearly ordered, i.e. if there exists a binary relation $\leq \subseteq \mathit{Key} \times \mathit{Key}$ such that the pair $\langle \mathit{Key}, \leq \rangle$ is a linear order, then the abstract data type *Map* can be implemented via *ordered binary trees* also known as *binary search trees*. The implementation of the ADT map that is based on ordered binary trees has the following performance characteristics:

1. The average case complexity of the lookup operation is logarithmic.
2. The worst case complexity of the lookup operation is linear.

In order to define *ordered binary trees* we introduce *binary trees* first.

Definition 12 (Binary Trees)
Assume a set *Key* and a set *Value* are given. The set $\mathcal{B}$ of binary trees is defined inductively as the set of terms that is build using the function symbols *Nil* and *Node*, where the signature of these function symbols is given as follows:

$$\mathit{Nil} : \mathcal{B} \qquad \text{and} \qquad \mathit{Node} : \mathit{Key} \times \mathit{Value} \times \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}.$$

1. *Nil* is a binary tree.
   This tree is called the *empty tree* since it does not store any information.

2. *Node*(k, v, l, r) is a binary tree if the following holds true:
   (a) k is a key from the set *Key*.
   (b) v is a value from the set *Value*.
   (c) l is a binary tree.
   l is the *left subtree* of the tree *Node*(k, v, l, r).
   (d) r is a binary tree.
   r is the *right subtree* of the tree *Node*(k, v, l, r). ◇

Next, we define the notion of an *ordered binary tree*.

Definition 13 (Ordered Binary Tree)
The set $\mathcal{B}_<$ of all *ordered binary trees* is defined inductively.

1. *Nil* $\in \mathcal{B}_<$
2. *Node*$(k, v, l, r) \in \mathcal{B}_<$ iff the following conditions hold:
   (a) k is a key from the set *Key*.
   (b) v is a value from the set *Value*.
   (c) l and r are ordered binary trees.
   (d) All keys that occur in the left subtree l are smaller than k.
   (e) All keys that occur in the right subtree r are bigger than k.

   The last two conditions are known as the *ordering conditions*. ◇

Graphically, ordered binary trees are depicted as follows:

1. The empty tree *Nil* is shown as a black circle.
2. A binary tree of the form *Node*(k, v, l, r) is represented by an oval. Inside of this oval, both the key k and the value v are printed. The key is printed above the value and both are separated by a horizontal line. This oval is then called a *node* of the binary tree. The left subtree l of the node is depicted both to the left and below the node, while the right subtree r is depicted both to the right and below the node. Both the left and the right subtree are connected to the node with an arrow that points from the node to the subtree.

Figure 5.2 shows an example of an ordered binary tree. The topmost node, that is the node that has the key 8 and the value 22 is called the *root* of the binary tree. A *path of length* k in the tree is list $[n_0, n_1, \cdots, n_k]$ of $k + 1$ nodes that are connected via arrows. If we identify nodes with their labels, we have that

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

is a path of length 3.

Next, we show how ordered binary trees can be used to implement the ADT *Map*. We specify the different methods of this ADT via conditional equations. The constructor *map*() returns the empty tree:

$$\mathit{map}() = \mathit{Nil}.$$

The method *find*() is specified as follows:

1. *Nil*.*find*$(k) = \Omega$,
   because the empty tree is interpreted as the empty map.

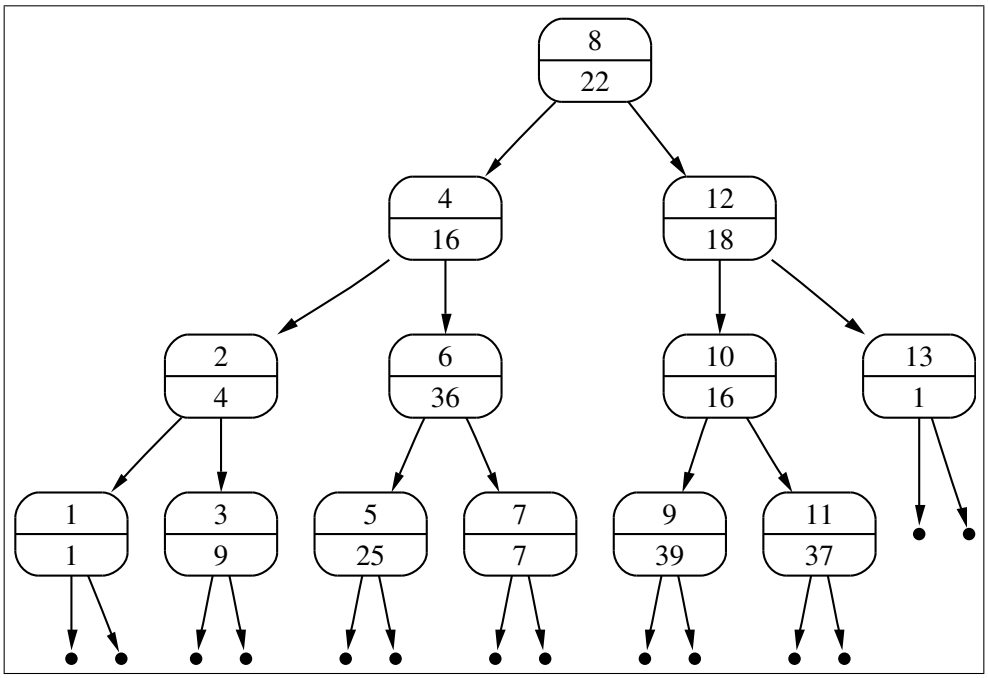

Figure 5.2: An ordered binary tree.

2. $\mathit{Node}(k, v, l, r).\mathit{find}(k) = v$,
   because the node $\mathit{Node}(k, v, l, r)$ stores the assignment $k \mapsto v$.
3. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v, l, r).\mathit{find}(k_1) = l.\mathit{find}(k_1)$,
   because if k_1 is less than k_2, then any mapping for k_1 has to be stored in the left subtree l.
4. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v, l, r).\mathit{find}(k_1) = r.\mathit{find}(k_1)$,
   because if k_1 is greater than k_2, then any mapping for k_1 has to be stored in the right subtree r.

Next, we specify the method *insert*. The definition of *insert* is similar to the definition of the method *find*.

1. $\mathit{Nil}.\mathit{insert}(k, v) = \mathit{Node}(k, v, \mathit{Nil}, \mathit{Nil})$,
   If the tree is empty, the information to be stored can be stored at the root.
2. $\mathit{Node}(k, v_2, l, r).\mathit{insert}(k, v_1) = \mathit{Node}(k, v_1, l, r)$,
   If the key k is located at the root, we can just overwrite the old information.
3. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{Node}(k_2, v_2, l.\mathit{insert}(k_1, v_1), r)$,
   If the key k_1, which is the key for which we want to store a value, is less than the key k_2 at the root, then we have to insert the information in the left subtree.
4. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{Node}(k_2, v_2, l, r.\mathit{insert}(k_1, v_1))$,
   If the key k_1, which is the key for which we want to store a value, is bigger than the key k_2 at the root, then we have to insert the information in the right subtree.

Finally we specify the method *delete*. The specification of *delete* is more difficult than the specification of *find* and *insert*. If there is a tree of the form $t = \mathit{Node}(k, v, l, r)$ and we want to delete the key k, then we have to check first whether either of the subtrees l or r is empty. If l is empty, $t.\mathit{delete}(k)$ can return the right subtree r while if r is empty, $t.\mathit{delete}(k)$ can return the left subtree l. Things

get more difficult when both l and r are non-empty. In this case, our solution is that we look for the smallest key in the right subtree r. This key and its corresponding value are removed from r. The resulting tree is called r'. Next, we take the node $t = \mathit{Node}(k, v, l, r)$ and transform it into the node $t' = \mathit{Node}(k_{min}, v_{min}, l, r')$. Here k_{min} denotes the smallest key found in r while v_{min} denotes the corresponding value. Note that t' is again ordered:

1. The key k_{min} is bigger than the key k and hence it is bigger than all keys in the left subtree l.
2. The key k_{min} is smaller than all keys in the subtree r', because k_{min} is the smallest key from the subtree r.

In order to illustrate the idea, let us consider the following example: If we want to delete the node with the label $\langle 4, 16\rangle$ from the tree shown in Figure 5.2, we first have to look for the smallest key in the subtree whose root is labelled $\langle 6, 36\rangle$. We find the node marked with the label $\langle 5, 25\rangle$. We remove this node and relabel the node that had the label $\langle 4, 16\rangle$ with the new label $\langle 5, 25\rangle$. The result is shown in Figure 5.3 on page 69.

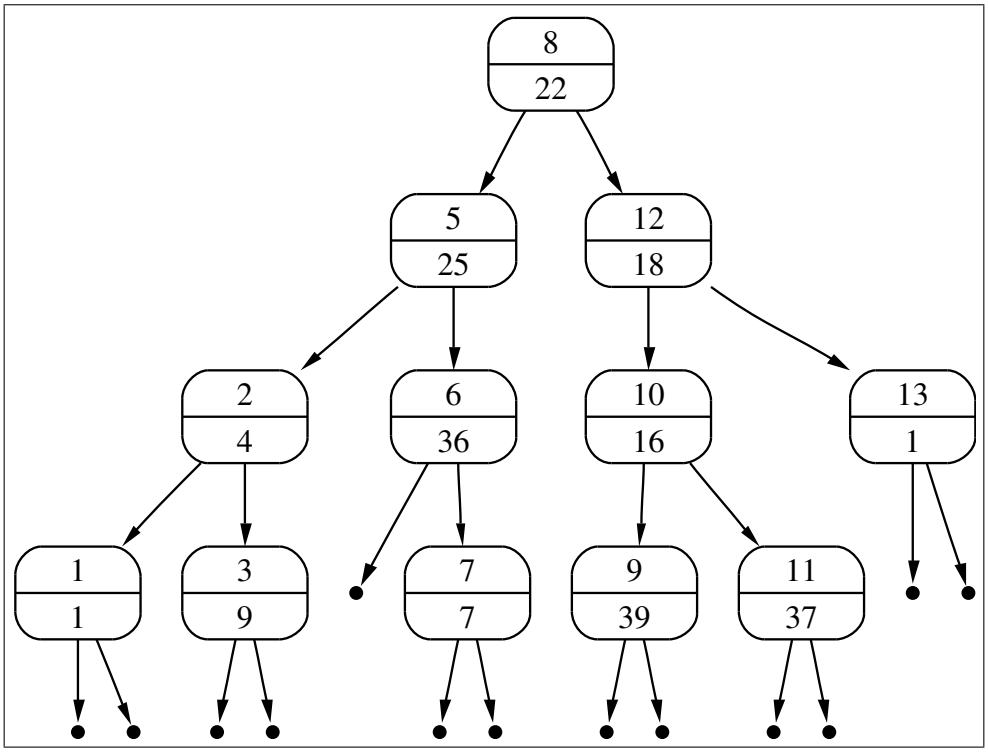

Figure 5.3: The ordered binary tree from Figure 5.2 after deleting the node with label $\langle 4, 16\rangle$.

Next, we specify the method *delMin*. The call $t.\mathit{delMin}()$ returns a triple. If

$$t.\mathit{delMin}() = [r, k, v],$$

then r is the tree that results from removing the smallest key in t, k is the key that is removed and v is the associated value.

1. $\mathit{Node}(k, v, \mathit{Nil}, r).\mathit{delMin}() = [r, k, v]$

   If the left subtree is empty, k has to be the smallest key in the tree $\mathit{Node}(k, v, \mathit{Nil}, r)$. If k is removed, we are left with the subtree r.

2. $l \neq \mathit{Nil} \wedge l.\mathit{delMin}() = [l', k_{min}, v_{min}] \;\rightarrow$
   $\mathit{Node}(k, v, l, r).\mathit{delMin}() = [\mathit{Node}(k, v, l', r), k_{min}, v_{min}].$

If the left subtree l in the binary tree $t = \mathit{Node}(k, v, l, r)$ is not empty, then the smallest key of t is located inside the left subtree l. This smallest key is recursively removed from l. This yields the tree l'. Next, l is replaced by l' in t. The resulting tree is $t' = \mathit{Node}(k, v, l', r)$.

Next, we specify the method `delete()`.

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. $l \neq \mathit{Nil} \wedge r \neq \mathit{Nil} \wedge r.\mathit{delMin}() = [r', k_{min}, v_{min}] \rightarrow$
   $\mathit{Node}(k, v, l, r).\mathit{delete}(k) = \mathit{Node}(k_{min}, v_{min}, l, r')$.

   If the key to be removed is found at the root of the tree and neither of its subtrees is empty, the call $r.\mathit{delMin}()$ removes the smallest key together with its associated value from the subtree r yielding the subtree r'. The smallest key from r is then stored at the root of the new tree.
5. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{delete}\big(k_1\big) = \mathit{Node}(k_2, v_2, l.\mathit{delete}(k_1), r)$.

   If the key that is to be removed is less than the key stored at the root, the key k can only be located in the left subtree l. Hence, k is removed from the left subtree l recursively.
6. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{delete}(k_1) = \mathit{Node}(k_2, v_2, l, r.\mathit{delete}(k_1))$.

   If the key that is to be removed is greater than the key stored at the root, the key k can only be located in the right subtree r. Hence, k is removed from the right subtree r recursively.

5.2.1 Implementing Ordered Binary Trees in SetlX

Figure 5.4 and Figure 5.5 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 that is associated with `mKey`. 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 only 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 5.4: Implementation of ordered binary trees SetlX, part (I).

5. The implementation of `insert` is similar to the implementation of `find`.

   (a) If the binary tree is empty, we set the member variables `mKey` and `mValue` to the appropriate values. The member variables `mLeft` and `mRight` are initialized as empty trees.

   (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.

6. The implementation of `delMin` and `delete` 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.

   There is however one caveat that should be mentioned. Line 55 show the implementation of the function `update`. When we delete the key at the root of the tree and either of the subtrees is

```
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 5.5: Implementation of ordered binary trees in SetlX, part (II).

empty, we would ideally just overwrite the current tree with the non-empty subtree, i.e. we would like to write something like

```
this := mLeft;
```

However, we cannot change the object `this`. The only thing we can do is change the attributes of the object `this`. This is done in the method `update`.

5.2.2 Analysis of the Complexity

In this section we will first discuss the worst case complexity, which is quite bad. In fact, in the worst case, the call $b.\texttt{find}(k)$ will perform $\mathcal{O}(n)$ key comparisons if b is an ordered binary search tree of n elements. After that, we investigate the average case complexity. We will show that the average case complexity is $\mathcal{O}\big(\ln(n)\big)$.

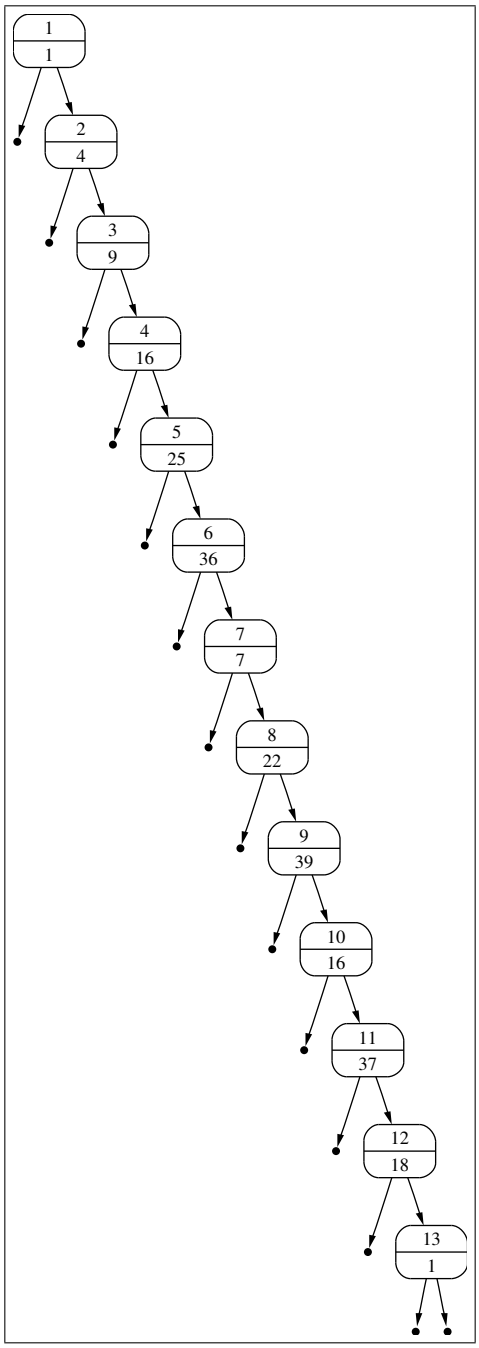

Figure 5.6: A degenerated binary tree.

Worst Case Complexity

We begin our investigation of the complexity with an analysis of the complexity of $b.\mathit{find}(k)$ in the worst case. The worst case happens if the binary tree b degenerates into a list. Figure 5.6 on page 73 shows the ordered binary tree that is generated if the keys are inserted in increasing order. If we then have to search for the biggest key, we have to traverse the complete tree in order to find this key. Therefore, if the tree b contains n different keys, we have to compare the key k that we are looking for to all of these n keys in the tree. Hence, in this case the complexity of $b.\mathit{find}(k)$ is $\mathcal{O}(n)$ and this is the same complexity that we would have gotten if we had used a linked list.

Average Case Complexity

Fortunately, the worst case has a very small probability to occur. On average, a randomly generated binary tree is quite well balanced. We will show next that the number of comparisons necessary for the function call $b.\mathit{find}(k)$ has the order $\mathcal{O}\big(\ln(n)\big)$.

In order to prove this claim, we have to introduce some definitions. We define the average number of comparisons that are needed for the function call $b.\mathit{find}(k)$ as d_n, where n is the number of keys stored in b. We assume that the key k is indeed stored in b. Our first goal is to derive a recurrence equation for d_n. First, we note that

$$d_1 = 1,$$

because if the tree b contains only one key we will do exactly one key comparison. Next, imagine a binary tree b that contains $n+1$ keys. Then b can be written as

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

where k' is the key at the root of b. If the keys of b are ordered as a list, then this ordering looks something like the following:

$$k_0 < k_1 < \cdots < k_{i-1} < k_i < k_{i+1} < \cdots < k_{n-1} < k_n.$$

Here are $n+1$ positions for the key k'. If we have $k' = k_i$, then the left subtree of b contains i keys while the right subtree contains the remaining $n-i$ keys:

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

As b contains $n+1$ keys all together, there are $n+1$ different possibilities for the position of k', as the number of keys in the left subtree l is i where

$$i \in \{0, 1, \cdots, n\}.$$

Of course, if the left subtree has i keys, the right subtree will have $n-i$ keys. Let us denote the average number of comparisons that are done during the function call $b.\mathit{find}(k)$ provided the left subtree of b has i keys while b itself has $n+1$ keys as

$$\mathit{numCmp}(i,\, n{+}1).$$

Then, since all values of i have the same probability, we have

$$d_{n+1} = \frac{1}{n+1} \cdot \sum_{i=0}^{n} \mathit{numCmp}(i,\, n{+}1).$$

We proceed to compute $\mathit{numCmp}(i, n+1)$: If l contains i keys while r contains the remaining $n-i$ keys, then there are three possibilities for the key k that we want to find in b:

1. k might be identical with the key k' that is located at the root of b. In this case there is only one comparison. As there are $n+1$ keys in b and the key we are looking for will be at the root in only one of these cases, the probability of this case is
$$\frac{1}{n+1}.$$
2. k might be identical to one of the i keys of the left subtree l. The probability for this case is
$$\frac{i}{n+1}.$$
In this case we need
$$d_i + 1$$
comparisons because in addition to the d_i comparisons in the left subtree we have to compare

the key k we are looking for with the key k' at the root of the tree.

3. k might be a key in the right subtree r. As there are $n-i$ keys in the right subtree and the total of keys is $n+1$, the probability that the key k occurs in the right subtree r is

$$\frac{n-i}{n+1}.$$

Hence, in this case there are

$$d_{n-i}+1$$

comparisons.

In order to compute $\mathit{numCmp}(i, n+1)$ we have to multiply the probabilities in every case with the number of comparisons and these three numbers have to added. This yields

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

Therefore, the recurrence equation for d_{n+1} is given as follows:

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

Here we have used the equation

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

We had verified this equation already when discussing the complexity of Quick Sort in the average case. Next, we solve the recurrence equation

$$d_{n+1} = 1 + \frac{2}{(n+1)^2}\cdot\sum_{i=0}^{n} i\cdot d_i \tag{5.1}$$

with the initial condition $d_1 = 1$. In order to solve the equation (5.1) we perform the substitution $n \mapsto n+1$. This yields

$$d_{n+2} = 1 + \frac{2}{(n+2)^2} \cdot \sum_{i=0}^{n+1} i \cdot d_i \tag{5.2}$$

We multiply equation (5.1) with $(n+1)^2$ and equation (5.2) with $(n+2)^2$. We get

$$(n+1)^2 \cdot d_{n+1} = (n+1)^2 + 2 \cdot \sum_{i=0}^{n} i \cdot d_i, \tag{5.3}$$

$$(n+2)^2 \cdot d_{n+2} = (n+2)^2 + 2 \cdot \sum_{i=0}^{n+1} i \cdot d_i \tag{5.4}$$

We subtract equation (5.3) from equation (5.4) and are left with

$$(n+2)^2 \cdot d_{n+2} - (n+1)^2 \cdot d_{n+1} = (n+2)^2 - (n+1)^2 + 2 \cdot (n+1) \cdot d_{n+1}.$$

To simplify this equation we substitute $n \mapsto n-1$ and get

$$(n+1)^2 \cdot d_{n+1} - n^2 \cdot d_n = (n+1)^2 - n^2 + 2 \cdot n \cdot d_n.$$

This can be simplified as

$$(n+1)^2 \cdot d_{n+1} = n \cdot (n+2) \cdot d_n + 2 \cdot n + 1.$$

Let us divide both sides of this equation by $(n+2) \cdot (n+1)$. We get

$$\frac{n+1}{n+2} \cdot d_{n+1} = \frac{n}{n+1} \cdot d_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)}.$$

We define

$$c_n = \frac{n}{n+1} \cdot d_n.$$

Then $c_1 = \frac{1}{2} \cdot d_1 = \frac{1}{2}$ and hence we have found the recurrence equation

$$c_{n+1} = c_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)}.$$

A partial fraction decomposition shows

$$\frac{2 \cdot n + 1}{(n+2) \cdot (n+1)} = \frac{3}{n+2} - \frac{1}{n+1}.$$

Hence we have

$$c_{n+1} = c_n + \frac{3}{n+2} - \frac{1}{n+1}.$$

Because of $c_1 = \frac{1}{2}$ this equation is also valid for $n = 0$ if we define $c_0 = 0$, since we have

$$\frac{1}{2} = 0 + \frac{3}{0+2} - \frac{1}{0+1}.$$

The recurrence equation for c_n can be solved using telescoping:

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

To simplify this equation we substitute $n \mapsto n-1$ and get

$$c_n = \sum_{i=2}^{n+1} \frac{3}{i} - \sum_{i=1}^{n} \frac{1}{i}$$

The harmonic number H_n is defined as $H_n = \sum_{i=1}^{n} \frac{1}{i}$. Therefore, c_n can be reduced to H_n:

$$c_n = 3 \cdot H_n - \frac{3}{1} + \frac{3}{n+1} - H_n = 2 \cdot H_n - 3 \cdot \frac{n}{n+1}$$

Because $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln(n) + \mathcal{O}(1)$ and $3 \cdot \frac{n}{n+1} \in \mathcal{O}(1)$ we therefore have

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

Because of $d_n = \frac{n+1}{n} \cdot c_n$ we have

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

This is our main result: On average, the operation $b.\mathit{find}(k)$ uses

$$2 \cdot \ln(n) = 2 \cdot \ln(2) \cdot \log_2(n) \approx 1.386 \cdot \log_2(n)$$

comparisons. Hence in the average case there are about 39 % more comparisons than there would be if the tree was optimally balanced. There are similar results for the operations *insert* and *delete*.

5.3 AVL Trees

If a binary tree is approximately *balanced*, i.e. if the left and right subtree of a binary tree b have roughly the same height, then the complexity of $b.\mathit{find}(k)$ will always be of the order $\mathcal{O}\big(\ln(n)\big)$. There are number of different variations of balanced binary trees. Of these variations, the species of balanced binary trees that is the easiest to understand is called an *AVL tree* [AVL62]. AVL trees are named after their inventors Georgy M. Adelson-Velsky and Evgenii M. Landis. In order to define these trees we need to define the *height* of a binary tree formally:

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 14 (AVL-Tree)
The set $\mathcal{A}$ of AVL trees *is defined inductively:*

1. $\mathit{Nil} \in \mathcal{A}$.
2. $\mathit{Node}(k, v, l, r) \in \mathcal{A}$ *iff*
   (a) $\mathit{Node}(k, v, l, r) \in \mathcal{B}_<$,
   (b) $l, r \in \mathcal{A}$ *and*
   (c) $|l.\mathit{height}() - r.\mathit{height}()| \leq 1$.
   This condition is called the balancing condition .

   According to this definition, an AVL tree is an ordered binary tree such that for every node $\mathit{Node}(k, v, l, r)$ in this tree the height of the left subtree l and the right subtree r differ at most by 1. □

In order to implement AVL trees we can start from our implementation of ordered binary trees. In addition to those methods that we have already seen in the class *Map* we will need the method

$$\mathit{restore} : \mathcal{B}_< \to \mathcal{A}.$$

This method is used to restore the balancing condition at a given node if it has been violated by either inserting or deleting an element. The method call $b.\mathit{restore}()$ assumes that b is an ordered binary tree that satisfies the balancing condition everywhere except possibly at its root. At the root, the height of the left subtree might differ from the height of the right subtree by at most 2. Hence, when the method $b.\mathit{restore}()$ is invoked we have either of the following two cases:

1. $b = \mathit{Nil}$ or
2. $b = \mathit{Node}(k, v, l, r) \wedge l \in \mathcal{A} \wedge r \in \mathcal{A} \wedge |l.\mathit{height}() - r.\mathit{height}()| \leq 2$.

The method *restore* is specified via conditional equations.

1. $\mathit{Nil}.\mathit{restore}() = \mathit{Nil}$,
   because the empty tree already is an AVL tree.
2. $|l.\mathit{height}() - r.\mathit{height}()| \leq 1 \rightarrow \mathit{Node}(k, v, l, r).\mathit{restore}() = \mathit{Node}(k, v, l, r)$,
   because if the balancing condition is already satisfied, then nothing needs to be done.
3. $$\begin{array}{lll} & l_1.\mathit{height}() = r_1.\mathit{height}() + 2 \\ \wedge & l_1 = \mathit{Node}(k_2, v_2, l_2, r_2) \\ \wedge & l_2.\mathit{height}() \geq r_2.\mathit{height}() \\ \rightarrow & \mathit{Node}(k_1, v_1, l_1, r_1).\mathit{restore}() = \mathit{Node}\big(k_2, v_2, l_2, \mathit{Node}(k_1, v_1, r_2, r_1)\big) \end{array}$$
   The motivation for this equation can be found in Figure 5.7 on page 79. The left part of this figure shows the state of the tree before it has been rebalanced. Therefore, this part shows the

tree

$$\mathit{Node}(k_1, v_1, \mathit{Node}(k_2, v_2, l_2, r_2), r_1).$$

The right part of Figure 5.7 shows the effect of rebalancing. This rebalancing results in the tree

$$\mathit{Node}\big(k_2, v_2, l_2, \mathit{Node}(k_1, v_1, r_2, r_1)\big).$$

In Figure 5.7 the label below the horizontal line of each node shows the height of the tree corresponding to this node. For subtrees, the height is given below the name of the subtree. For example, h is the height of the subtree l_2, while $h-1$ is the height of the subtree r_1. The height of the subtree r_2 is h' and we know that $h' \leq h$. As $\mathit{Node}(k_2, v_2, l_2, r_2)$ is an AVL tree and we know that $l_2.\mathit{height}() \geq r_2.\mathit{height}()$, we either have $h' = h$ or $h' = h - 1$.

The state shown in Figure 5.7 can arise if either an element has been inserted in the left subtree l_1 or if an element has been deleted from the right subtree r_1.

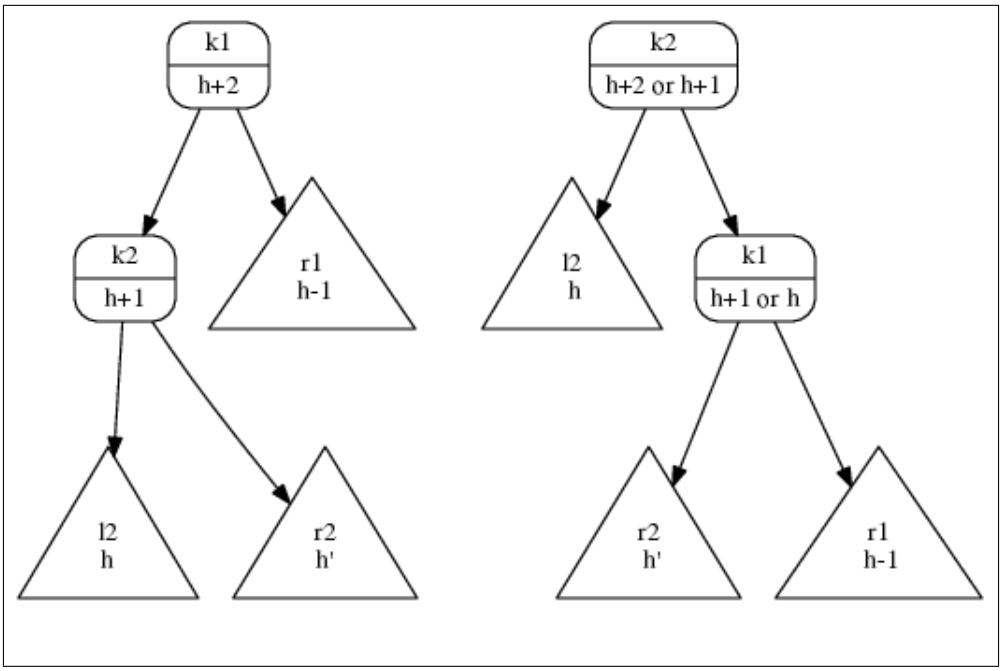

Figure 5.7: An unbalanced tree and the corresponding rebalanced tree.

We have to make sure that the tree shown in the right part of Figure 5.7 is indeed an AVL tree. With respect to the balancing condition this is easily verified. The fact that the node containing the key k_1 has either the height h or $h+1$ is a consequence of the fact that the height of r_1 is $h-1$ and that $h' \in \{h, h-1\}$.

In order to verify that the tree is ordered we can use the following inequation:

$$l_2 < k_2 < r_2 < k_1 < r_1. \qquad (\star)$$

Here we have used the following notation: If k is a key and b is a binary tree, then we write

$$k < b$$

in order to express that k is smaller than all keys that occur in the tree b. Similarly, $b < k$ denotes the fact that all keys occurring in b are less than the key k. The inequation $(\star)$ describes both the ordering of keys in the left part of Figure 5.7 and in the right part of this figure. Hence, the tree shown in the right part of Figure 5.7 is ordered provided the tree in the left part is ordered to begin with.

4. $l_1.\mathit{height}() = r_1.\mathit{height}() + 2$
   $\wedge$ $l_1 = \mathit{Node}(k_2, v_2, l_2, r_2)$
   $\wedge$ $l_2.\mathit{height}() < r_2.\mathit{height}()$
   $\wedge$ $r_2 = \mathit{Node}(k_3, v_3, l_3, r_3)$
   $\rightarrow$ $\mathit{Node}(k_1, v_1, l_1, r_1).\mathit{restore}() = \mathit{Node}\big(k_3, v_3, \mathit{Node}(k_2, v_2, l_2, l_3), \mathit{Node}(k_1, v_1, r_3, r_1)\big)$

   The left hand side of this equation is shown in Figure 5.8 on page 80. This tree can be written as

   $$\mathit{Node}\big(k_1, v_1, \mathit{Node}(k_2, v_2, l_2, \mathit{Node}\big(k_3, v_3, l_3, r_3)\big), r_1\big).$$

   The subtrees l_3 and r_3 have either the height h or $h-1$. Furthermore, at least one of these subtrees must have the height h for otherwise the subtree $\mathit{Node}(k_3, v_3, l_3, r_3)$ would not have the height $h+1$.

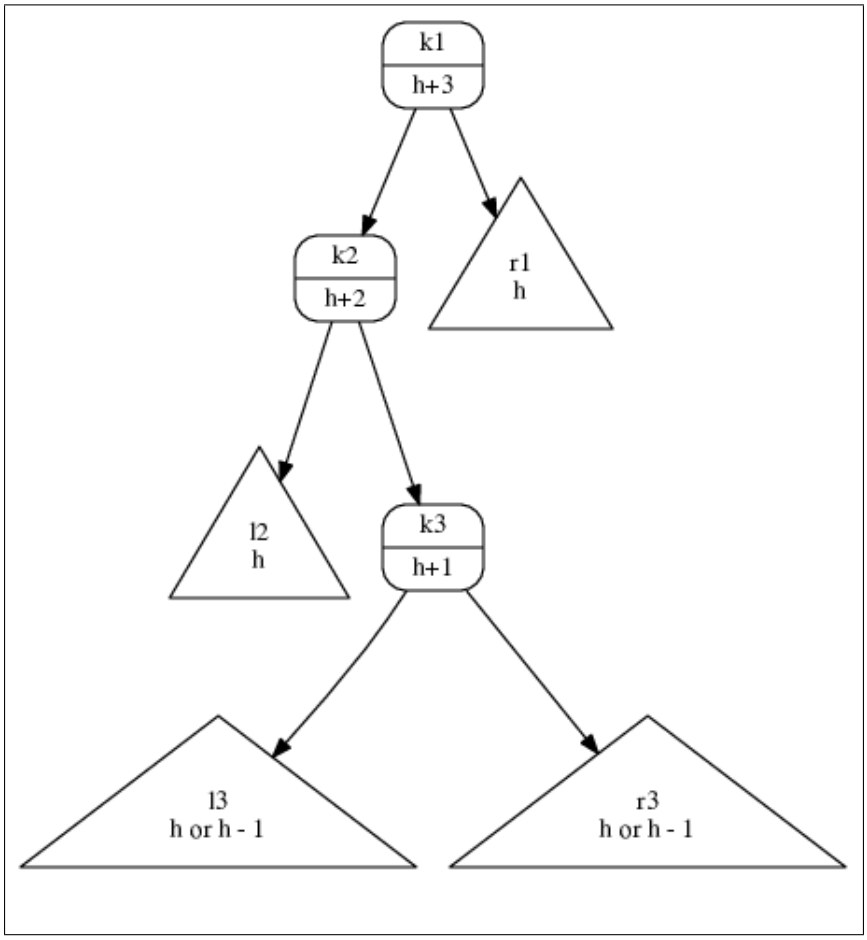

Figure 5.8: An unbalanced tree, second case.

Figure 5.9 on page 81 shows how the tree looks after rebalancing. The tree shown in this figure has the form

$$\mathit{Node}\big(k_3, v_3, \mathit{Node}(k_2, v_2, l_2, l_3), \mathit{Node}(k_1, v_1, r_3, r_1)\big).$$

The inequation describing the ordering of the keys both in the left subtree and in the right subtree is given as

$$l_2 < k_2 < l_3 < k_3 < r_3 < k_1 < r_1.$$

There are two more cases where the height of the right subtree is bigger by more than the height

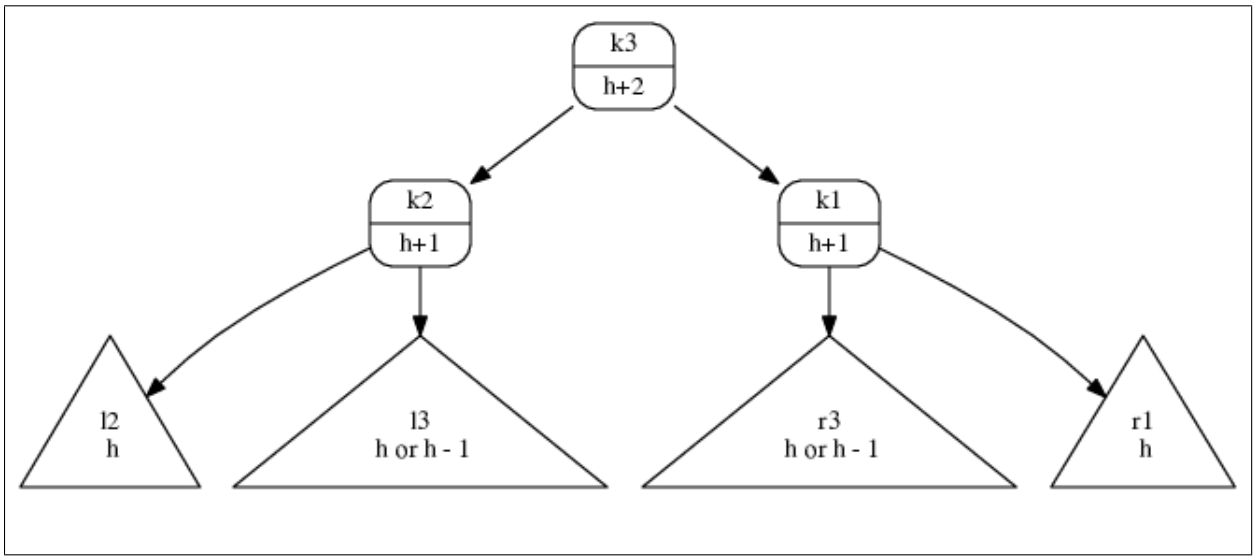

Figure 5.9: The rebalanced tree in the second case.

of the left subtree plus one. These two cases are completely analogous to the two cases discussed previously. Therefore we just state the corresponding equations without further discussion.

5. $r_1.\mathit{height}() = l_1.\mathit{height}() + 2$
   $\wedge \quad r_1 = \mathit{Node}(k_2, v_2, l_2, r_2)$
   $\wedge \quad r_2.\mathit{height}() \geq l_2.\mathit{height}()$
   $\rightarrow \quad \mathit{Node}(k_1, v_1, l_1, r_1).\mathit{restore}() = \mathit{Node}\big(k_2, v_2, \mathit{Node}(k_1, v_1, l_1, l_2), r_2\big)$

6. $r_1.\mathit{height}() = l_1.\mathit{height}() + 2$
   $\wedge \quad r_1 = \mathit{Node}(k_2, v_2, l_2, r_2)$
   $\wedge \quad r_2.\mathit{height}() < l_2.\mathit{height}()$
   $\wedge \quad l_2 = \mathit{Node}(k_3, v_3, l_3, r_3)$
   $\rightarrow \quad \mathit{Node}(k_1, v_1, l_1, r_1).\mathit{restore}() = \mathit{Node}\big(k_3, v_3, \mathit{Node}(k_1, v_1, l_1, l_3), \mathit{Node}(k_2, v_2, r_3, r_2)\big)$

Now we are ready to specify the method *insert*() via recursive equations. If we compare these equations to the equations we had given for unbalanced ordered binary trees we notice that we only have to call the method *restore* if the balancing condition might have been violated.

1. $\mathit{Nil}.\mathit{insert}(k, v) = \mathit{Node}(k, v, \mathit{Nil}, \mathit{Nil})$.
2. $\mathit{Node}(k, v_2, l, r).\mathit{insert}(k, v_1) = \mathit{Node}(k, v_1, l, r)$.
3. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) = \mathit{Node}\big(k_2, v_2, l.\mathit{insert}(k_1, v_1), r\big).\mathit{restore}()$.
4. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{insert}\big(k_1, v_1\big) = \mathit{Node}\big(k_2, v_2, l, r.\mathit{insert}(k_1, v_1)\big).\mathit{restore}()$.

The equations for *delMin*() change as follows:

1. $\mathit{Node}(k, v, \mathit{Nil}, r).\mathit{delMin}() = \langle r, k, v\rangle$.
2. $l \neq \mathit{Nil} \wedge \langle l', k_{min}, v_{min}\rangle := l.\mathit{delMin}() \ \rightarrow$
   $\mathit{Node}(k, v, l, r).\mathit{delMin}() = \langle \mathit{Node}(k, v, l', r).\mathit{restore}(), k_{min}, v_{min}\rangle$.

Then, the equations for *delete* are as follows:

1. $\mathit{Nil}.\mathit{delete}(k) = \mathit{Nil}$.

2. $\mathit{Node}(k, v, \mathit{Nil}, r).\mathit{delete}(k) = r$.
3. $\mathit{Node}(k, v, l, \mathit{Nil}).\mathit{delete}(k) = l$.
4. $l \neq \mathit{Nil} \wedge r \neq \mathit{Nil} \wedge \langle r', k_{min}, v_{min} \rangle := r.\mathit{delMin}() \rightarrow$
   $\mathit{Node}(k, v, l, r).\mathit{delete}(k) = \mathit{Node}(k_{min}, v_{min}, l, r').\mathit{restore}()$.
5. $k_1 < k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{delete}(k_1) = \mathit{Node}\big(k_2, v_2, l.\mathit{delete}(k_1), r\big).\mathit{restore}()$.
6. $k_1 > k_2 \rightarrow \mathit{Node}(k_2, v_2, l, r).\mathit{delete}(k_1) = \mathit{Node}\big(k_2, v_2, l, r.\mathit{delete}(k_1)\big).\mathit{restore}()$.

5.3.1 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 5.10 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 5.10: Outline of the class `map`.

Figure 5.11 shows the implementation of the function `find`. Actually, the implementation is the same as the implementation in Figure 5.4. 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.

Figure 5.12 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.

```
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 5.11: Implementation of the method `find`.

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 5.12: Implementation of the method `insert`.

Figure 5.13 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 5.13: Implementation of `delMin`.

Figure 5.14 shows the implementation of the method `delete` and the implementation of the auxiliary method `update`. Compared with Figure 5.5 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 shrink 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 5.14: The methods `delete` and `update`.

Figure 5.15 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

```
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 5.15: The implementation of `restore` and `restoreHeight`.

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 shrunk 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 5.7 and 5.8.

3. Next, the test in line 9 checks whether we have the case that is depicted in Figure 5.7. In this case, Figure 5.7 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 5.16 on page 86.

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 5.8. We have to create the tree that is shown in Figure 5.9. 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 5.8. Next, we create the tree that is shown in Figure 5.9.

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 5.16: Implementation of createNode.

The function createNode shown in Figure 5.16 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.

5.3.2 Analysis of the Complexity of AVL Trees

Next, we analyse the complexity of AVL trees in the worst case. In order to do this we have to know what the worst case actually looks like. Back when we only had ordered binary trees the worst case was the case where the tree had degenerated into a list. Now, the worst case is the case where the tree is as slim as it can possibly be while still satisfying the definition of an AVL tree. Hence the worst case happens if the tree has a given height h but the number of keys stored in the tree is as small as possible. To investigate trees of this kind, let us define $b_h(k)$ as an AVL tree that has height h and whose number of keys is minimal among all other AVL trees of given height h. Furthermore, we demand that all keys stored in $b_h(k)$ are bigger than k. For our investigation of the complexity, both the keys and the values do not really matter. The only problem is that we have to make sure that the tree $b_h(k)$ that we are going to construct in a moment is actually an ordered tree and for this reason we insist that all keys in $b_h(k)$ are bigger than k. We will use natural numbers as keys, while all values

will be 0. Before we can actually present the definition of $b_h(k)$ we need to define the auxiliary function *maxKey*(). This function has the signature

$$\mathit{maxKey} : \mathcal{B}_< \to \mathit{Key} \cup \{\Omega\}.$$

Given a non-empty ordered binary tree b, the expression $b.\mathit{maxKey}()$ returns the biggest key stored in b. The expression $b.\mathit{maxKey}()$ is defined by induction on b:

1. $\mathit{Nil}.\mathit{maxKey}() = \Omega$,
2. $\mathit{Node}(k, v, l, \mathit{Nil}).\mathit{maxKey}() = k$,
3. $r \neq \mathit{Nil} \to \mathit{Node}(k, v, l, r).\mathit{maxKey}() = r.\mathit{maxKey}()$.

Now we are ready to define the trees $b_h(k)$ by induction on h.

1. $b_0(k) = \mathit{Nil}$,

   because there is only one AVL tree of height 0 and this is the tree *Nil*.

2. $b_1(k) = \mathit{Node}(k+1, 0, \mathit{Nil}, \mathit{Nil})$,

   since, if we abstract from the actual keys and values, there is exactly one AVL tree of height 1.

3. $b_{h+1}(k).\mathit{maxKey}() = l \to b_{h+2}(k) = \mathit{Node}\big(l+1,\, 0,\, b_{h+1}(k),\, b_h(l+1)\big)$.

   In order to construct an AVL tree of height $h+2$ that contains the minimal number of keys possible we first construct the AVL tree $b_{h+1}(k)$ which has height $h+1$ and which stores as few key as possible given its height. Next, we determine the biggest key l in this tree. Now to construct $b_{h+2}(k)$ we take a node with the key $l+1$ as the root. The left subtree of this node is $b_{h+1}(k)$, while the right subtree is $b_h(l+1)$. Since l is the biggest key in $b_{h+1}(k)$, all key in the left subtree of $b_{h+2}(k)$ are indeed smaller than the key $l+1$ at the root. Since all keys in $b_h(l+1)$ are bigger than $l+1$, the keys in the right subtree are bigger than the key at the root. Therefore, $b_{h+2}(k)$ is an ordered binary tree.

   Furthermore, $b_{h+2}(k)$ is an AVL tree of height $h+2$ since the height of the left subtree is $h+1$ and the height of the right subtree is h. Also, this tree is as slim as any AVL tree can possibly get, since if the left subtree has height $h+1$ the right subtree must at least have height h in order for the whole tree to be an AVL tree.

Let us denote the number of keys stored in a binary tree b as $\#\, b$. Furthermore, we define

$$c_h := \#\, b_h(k)$$

to be the number of keys in the tree $b_h(k)$. We will see immediately that $\#\, b_h(k)$ does not depend on the number k and therefore c_h does not depend on k. Starting from the definition of $b_h(k)$ we find the following equations for c_h:

1. $c_0 = \#\, b_0(k) = \#\, \mathit{Nil} = 0$,
2. $c_1 = \#\, b_1(k) = \#\, \mathit{Node}(k+1, 0, \mathit{Nil}, \mathit{Nil}) = 1$,
3. $$\begin{array}{rcl} c_{h+2} & = & \#\, b_{h+2}(k) \\ & = & \#\, \mathit{Node}\big(l+1,\, 0,\, b_{h+1}(k),\, b_h(l+1)\big) \\ & = & \#\, b_{h+1}(k) + \#\, b_h(l+1) + 1 \\ & = & c_{h+1} + c_h + 1. \end{array}$$

Hence we have found the recurrence equation

$$c_{h+2} = c_{h+1} + c_h + 1 \quad \text{with initial values } c_0 = 0 \text{ and } c_1 = 1.$$

This also validates our claim that c_h does not depend on k. In order to solve the recurrence equation derived above we first solve the corresponding *homogeneous recurrence equation*

$$a_{h+2} = a_{h+1} + a_h$$

using the ansatz

$$a_h = \lambda^h.$$

Substituting $a_h = \lambda^h$ into the recurrence equation for a_h leaves us with the equation

$$\lambda^{h+2} = \lambda^{h+1} + \lambda^h.$$

Dividing by λ^h leaves the quadratic equation

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

which can be rearranged as

$$\lambda^2 - 2 \cdot \lambda \cdot \tfrac{1}{2} = 1.$$

Adding $\frac{1}{4}$ on both sides of this equation completes the square on the left hand side:

$$\left(\lambda - \tfrac{1}{2}\right)^2 = \tfrac{5}{4}.$$

From this we conclude

$$\lambda = \tfrac{1}{2} \cdot \left(1 + \sqrt{5}\right) \ \vee \ \lambda = \tfrac{1}{2} \cdot \left(1 - \sqrt{5}\right).$$

Let us therefore define

$$\lambda_1 = \tfrac{1}{2} \cdot (1 + \sqrt{5}) \approx 1.618034 \quad \text{and} \quad \lambda_2 = \tfrac{1}{2} \cdot (1 - \sqrt{5}) \approx -0.618034.$$

In order to solve the *inhomogeneous recurrence equation* for c_h we try the ansatz

$$c_h = d \quad \text{for some constant } d.$$

Substituting this ansatz into the recurrence equation for c_h yields

$$d = d + d + 1$$

from which we conclude that $d = -1$. Hence the solution for c_h has the form

$$c_h = \alpha \cdot \lambda_1^h + \beta \cdot \lambda_2^h + d = \alpha \cdot \lambda_1^h + \beta \cdot \lambda_2^h - 1.$$

Here, the values of α and β can be found by setting $h = 0$ and $h = 1$ and using the initial conditions $c_0 = 0$ and $c_1 = 1$. This results in the following system of linear equations for α and β:

$$0 = \alpha + \beta - 1 \quad \text{and} \quad 1 = \alpha \cdot \lambda_1 + \beta \cdot \lambda_2 - 1.$$

From the first equation we find $\beta = 1 - \alpha$ and substituting this result into the second equation gives

$$2 = \alpha \cdot \lambda_1 + (1 - \alpha) \cdot \lambda_2.$$

Solving this equation for α gives

$$2 - \lambda_2 = \alpha \cdot (\lambda_1 - \lambda_2)$$

You can easily verify that $\lambda_1 - \lambda_2 = \sqrt{5}$ and $2 - \lambda_2 = \lambda_1^2$ holds. Hence, we have found

$$\alpha = \frac{2 - \lambda_2}{\lambda_1 - \lambda_2} = \frac{1}{\sqrt{5}} \cdot \lambda_1^2.$$

From this, a straightforward calculation using the fact that $\beta = 1 - \alpha$ shows that

$$\beta = -\frac{1}{\sqrt{5}} \cdot \lambda_2^2.$$

Therefore, c_h is given by the following equation:

$$c_h = \frac{1}{\sqrt{5}} \left(\lambda_1^{h+2} - \lambda_2^{h+2}\right) - 1.$$

As we have $|\lambda_2| < 1$, the value of λ_2^{h+2} isn't important for big values of h. Therefore, for big values of h, the minimal number n of keys in a tree of height h is approximately given by the formula

$$n \approx \frac{1}{\sqrt{5}}\lambda_1^{h+2} - 1.$$

In order to solve this equation for h we take the logarithm of both side. Then we have

$$\log_2(n+1) = (h+2) \cdot \log_2(\lambda_1) - \tfrac{1}{2} \cdot \log_2(5).$$

Adding $\frac{1}{2} \cdot \log_2(5)$ gives

$$\log_2(n+1) + \tfrac{1}{2} \cdot \log_2(5) = (h+2) \cdot \log_2(\lambda_1).$$

Let us divide this inequation by $\log_2(\lambda_1)$. Then we get

$$\frac{\log_2(n+1) + \frac{1}{2} \cdot \log_2(5)}{\log_2(\lambda_1)} = h + 2.$$

Solving this equation for h gives the result

$$\begin{aligned} h &= \frac{\log_2(n+1) + \frac{1}{2} \cdot \log_2(5)}{\log_2(\lambda_1)} - 2 \\ &= \frac{1}{\log_2(\lambda_1)} \cdot \log_2(n) + \mathcal{O}(1) \\ &\approx 1,44 \cdot \log_2(n) + \mathcal{O}(1). \end{aligned}$$

However, the height h is the maximal number of comparisons needed to find a given key. Hence, for AVL trees the complexity of $b.\mathit{find}(k)$ is logarithmic even in the worst case. Figure 5.17 presents an AVL tree of height 6 where the number of keys is minimal.

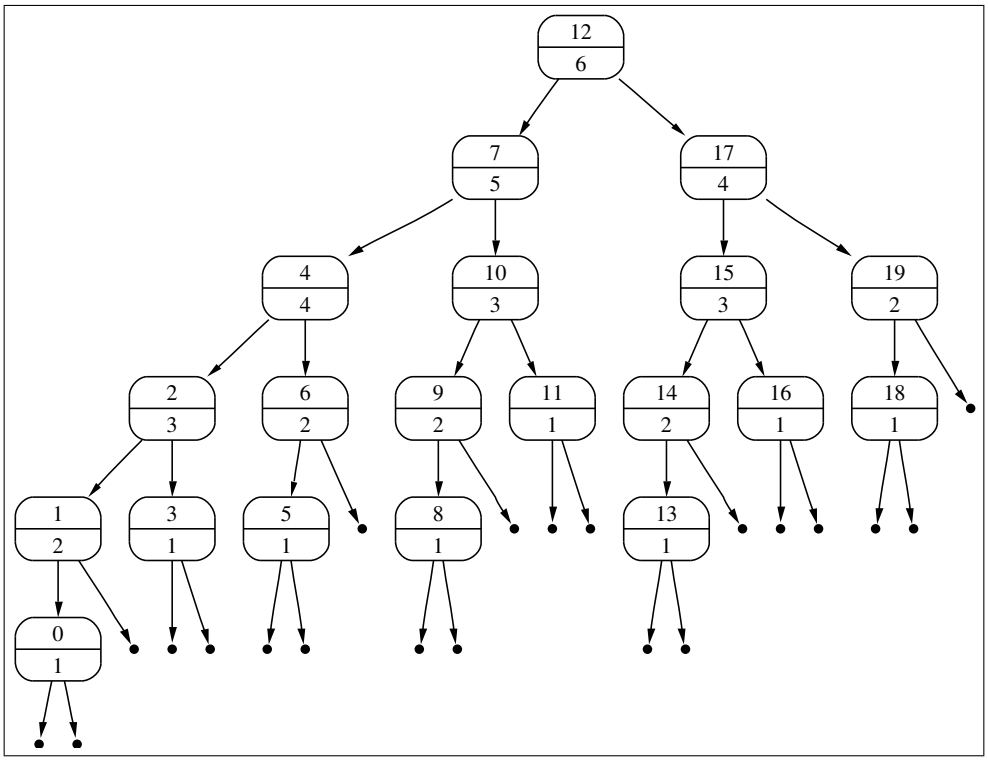

Figure 5.17: An AVL tree of height 6 that is as slim as possible.

5.3.3 Improvements

In practice, *red-black trees* are slightly faster than AVL trees. Similar to AVL trees, a red-black tree is an ordered binary tree that is approximately balanced. Nodes are either black or red. The children of a red node 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 every node 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 by Sedgewick [SW11b]. Red-black trees have been invented by Leonidas L. Guibas and Robert Sedgewick [GS78].

Exercise 14: Instead of using AVL trees, another alternative to implement a map is to use *2-3 trees*. Below we describe a simplified version of these trees. These trees do not store any values. Hence, instead of implementing maps, these trees implement sets. They are built using the following constructors:

1. `Nil` is a 2-3 tree that represents the empty set.
2. $\texttt{Two}(l, k, r)$ is a 2-3 tree provided
   (a) l is a 2-3 tree,
   (b) k is a key,
   (c) r is a 2-3 tree,
   (d) all keys stored in l are less than k and all keys stored in r are bigger than k, i.e. we have
   $$l < k < r.$$
   (e) l and r have the same height.

   A node of the form $\texttt{Two}(l, k, r)$ is called a *2-node*. Except for the fact that there is no value, a 2-node is interpreted in the same way as we have interpreted the term $\texttt{Node}(k, v, l, r)$.
3. $\texttt{Three}(l, k_1, m, k_2, r)$ is a 2-3 tree provided
   (a) l, m, and r are 2-3 trees,
   (b) k_1 and k_2 are keys,
   (c) $l < k_1 < m < k_2 < r$,
   (d) l, m, and r have the same height.

   A node of the form $\texttt{Three}(l, k_1, m, k_2, r)$ is called a *3-node*.

In order to keep 2-3 trees balanced when inserting new keys, we use a fourth constructor of the form

$$\texttt{Four}(l, k_1, m_l, k_2, m_r, k_3, r).$$

A term of the form $\texttt{Four}(l, k_1, m_l, k_2, m_r, k_3, r)$ is a *2-3-4* tree iff

1. l, m_l, m_r, and r are 2-3 trees,
2. k_1, k_2, and k_3 are keys,
3. $l < k_1 < m_l < k_2 < m_r < k_3 < r$,
4. l, m_l, m_r, and r all have the same height.

Nodes of this form are called 4-nodes and the key k_2 is called the *middle key*. Trees containing 4-nodes are called *2-3-4* trees. When a new key is inserted into a 2-3 tree, the challenge is to keep the tree balanced. The easiest case is the case where the tree has the form

$$\texttt{Two}(\texttt{Nil}, k, \texttt{Nil}).$$

In this case, the 2-node is converted into a 3-node. If the tree has the form

$$\texttt{three}(\texttt{Nil}, k_1, \texttt{Nil}, k_2, \texttt{Nil}),$$

the 3-node is temporarily transformed into a 4-node. Next, the middle key of this node is lifted up to its parent node. For example, suppose we insert the key 3 into the tree

```
Two(Two(Nil, 1, Nil), 2, Three(Nil, 4, Nil, 5, Nil)).
```

In this case, the key 3 needs to be inserted to the left of the key 4. This yields the temporary tree

```
Two(Two(Nil, 1, Nil), 2, Four(Nil, 3, Nil, 4, Nil, 5, Nil)).
```

Since this is not a 2-3 tree, we need to lift the middle key 4 to its parent node. This results in the new tree

```
Three(Two(Nil, 1, Nil), 2, Two(Nil, 3, Nil), 4, Two(Nil, 5, Nil)).
```

This tree is a 2-3 tree. In this example we have been lucky since the parent of the 4-node was a 2 node and therefore we could transform it into a 3-node. If the parent node instead is a 3-node, it has to be transformed into a temporary 4-node. Then, the middle key of this 4-node has to be lifted up recursively to its parent.

(a) Specify a method $t.\texttt{member}(k)$ that checks whether the key k occurs in the 2-3 tree t. You should use recursive equations to specify $t.\texttt{member}(k)$.

(b) Specify a method $t.\texttt{insert}(k)$ that inserts the key k into the 2-3 tree t. You should make use of an auxiliary function $t.\texttt{restore}()$ that takes a 2-3-4 tree and transforms it into an equivalent 2-3 tree.

(c) Implement 2-3 trees in SetlX.

According to [CLRS09], 2-3 trees have been invented by John Hopcroft in 1970.

5.4 Tries

Often, the keys of a map are strings. For example, when you search with Google, you are using a string as a key to lookup information that is stored in a gigantic map provided by Google. As another example, in an electronic phone book the keys are names and therefore are strings. There is a species of search trees that is particularly well adapted to the case that the keys are strings. These search trees are known as *tries*. The name is derived from the word *retrieval*. In order to be able to distinguish between *tries* and *trees* we have to pronounce *trie* so that it rhymes with *pie*. The data structure of tries has been proposed 1959 by René de la Briandais [dlB59].

Tries are also trees, but in contrast to a binary tree where every node has two children, in a trie a node can have as many children as there are characters in the alphabet that is used to represent the strings. In order to define tries formally we assume that the following is given:

- Σ is finite set of *characters*. Σ is called the *alphabet*.
- Σ^* is the set of all *strings* that are built from the characters of Σ. Formally, a string is just a list of characters. If we have $w \in \Sigma^*$, then we write $w = cr$ if c is the first letter of w and if r the string that remains if we remove the first character from w.
- ε denotes the empty string.
- *Value* is the set of all the values that can be associated with the keys.

The set $\mathbb{T}$ of all tries is defined inductively using the constructor

$$\text{Node} : \mathit{Value} \times \mathit{List}(\Sigma) \times \mathit{List}(\mathbb{T}) \to \mathbb{T}.$$

The inductive definition of the set $\mathbb{T}$ has only a single clause: If

1. $v \in \mathit{Value} \cup \{\Omega\}$
2. $C = [c_1, \cdots, c_n] \in \mathit{List}(\Sigma)$ is a list of different characters of length n and
3. $T = [t_1, \cdots, t_n] \in \mathit{List}(\mathbb{T})$ is a list of tries of the same length n,

then we have

$$\mathsf{Node}(v, C, T) \in \mathbb{T}.$$

As there is only one clause in this definition, you might ask how this inductive definition gets started. The answer is that the base case of this inductive definition is the case where $n = 0$ since in that case the lists C and T are both empty. Next, we specify the function that is represented by a trie of the form

$$\mathsf{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).$$

In order to do so, we specify a function

$$\mathit{find} : \mathbb{T} \times \Sigma^* \to \mathit{Value} \cup \{\Omega\}$$

that takes a trie and a string. For a trie t, the expression $t.\mathit{find}(s)$ returns the value that is associated with the string s in the trie t. The expression $\mathsf{Node}(v, C, T).\mathit{find}(s)$ is defined by induction on the length of the string s:

1. $\mathsf{Node}(v, C, T).\mathit{find}(\varepsilon) = v$.

   The value associated with the empty string ε is stored at the root of the trie.

2. $$\mathsf{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{find}(cr) = \begin{cases} t_1.\mathit{find}(r) & \text{if} \quad c = c_1; \\ \vdots & \\ t_i.\mathit{find}(r) & \text{if} \quad c = c_i; \\ \vdots & \\ t_n.\mathit{find}(r) & \text{if} \quad c = c_n; \\ \Omega & \text{if} \quad c \notin \{c_1, \cdots, c_n\}. \end{cases}$$

   The trie $\mathsf{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$ associates a value with the key cr if the list $[c_1, \cdots, c_n]$ has a position i such that c equals c_i and, furthermore, the trie t_i associates a value with the key r.

Graphically, tries are represented as trees. Since it would be unwieldy to label the nodes of these trees with the lists of characters corresponding to these nodes, we use a trick: In order to visualize a node of the form

$$\mathsf{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n])$$

we draw a circle. This circle is split into two parts by a horizontal line. If the value v that is stored in this node is different from Ω, then the value v is written in the lower part of the circle. The label that we put in the upper half of the circle depends on the parent of the node. We will explain how this label is computed in a moment. The node itself has n different children. These n children are the tries t_1, $\cdots$, t_n. The node at the root of the trie t_i is labelled with the character c_i, i.e. the circle that represents this node carries the label c_i in its upper half.

In order to clarify these ideas, Figure 5.18 on page 93 shows a trie mapping some strings to numbers. The mapping depicted in this tree can be written as a functional relation:

$$\big\{\langle\text{“Stahl”}, 1\rangle, \langle\text{“Stolz”}, 2\rangle, \langle\text{“Stoeger”}, 3\rangle, \langle\text{“Salz”}, 4\rangle, \langle\text{“Schulz”}, 5\rangle,$$
$$\langle\text{“Schulze”}, 6\rangle, \langle\text{“Schnaut”}, 7\rangle, \langle\text{“Schnupp”}, 8\rangle, \langle\text{“Schroer”}, 9\rangle\big\}.$$

Since the node at the root has no parent, the upper half of the circle representing the root is empty. The lower half of this circle is empty because the trie doesn't associate a value with the empty string. This root node corresponds to the term

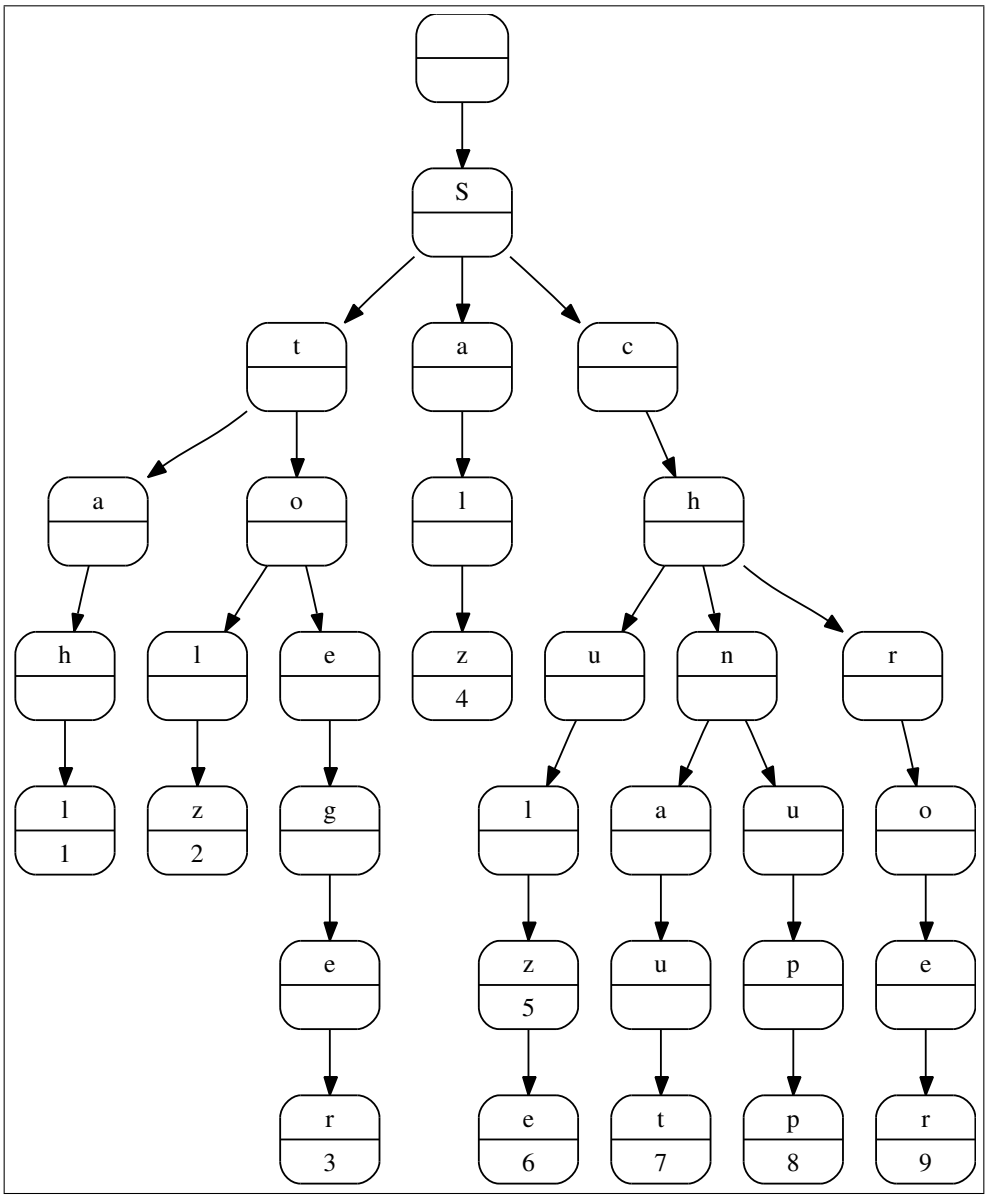

Figure 5.18: A trie storing some numbers.

$$\mathsf{Node}(\Omega, [\text{'S'}], [t]).$$

Here, t denotes the trie that is labelled with the character "S" at its root. This trie can then be represented by the term

$$\mathsf{Node}(\Omega, [\text{'t'}, \text{'a'}, \text{'c'}], [t_1, t_2, t_3]).$$

This trie has three children that are labelled with the characters "t", "a", and "c".

5.4.1 Insertion in Tries

Next, we present formulæ that describe how new values can be inserted into an existing tries, i.e. we specify the method `insert`. The signature of `insert` is given as follows:

$$\mathit{insert} : \mathbb{T} \times \Sigma^* \times \mathit{Value} \to \mathbb{T}.$$

The result of evaluating

$$\mathsf{Node}(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{insert}(w, v_2)$$

for a string $w \in \Sigma^*$ and a value $v_2 \in \mathit{Value}$ is defined by induction on the length of w.

1. $\mathsf{Node}(v_1, L, T).\mathit{insert}(\varepsilon, v_2) = \mathsf{Node}(v_2, L, T)$,

   If a new value v_2 is associated with the empty string ε, then the old value v_1, that had been stored at the root before, is overwritten.

2. $\mathsf{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big).\mathit{insert}(c_i r, v_2) =$

   $$\mathsf{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i.\mathit{insert}(r, v_2), \cdots, t_n]\big).$$

   In order to associate a value v_2 with the string $c_i r$ in the trie

   $$\mathsf{Node}\big(v_1, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]\big)$$

   we have to recursively associate the value v_2 with the string r in the trie t_i.

3. $c \notin \{c_1, \cdots, c_n\} \;\rightarrow\; \mathsf{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big).\mathit{insert}(cr, v_2) =$

   $$\mathsf{Node}\big(v_1, [c_1, \cdots, c_n, c], [t_1, \cdots, t_n, \mathsf{Node}(\Omega, [], []).\mathit{insert}(r, v_2)]\big).$$

   If we want to associate a value v with the key cr in the trie $\mathsf{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big)$ then, if the character c does not occur in the list $[c_1, \cdots, c_n]$, we first have to create a new empty trie. This trie has the form

   $$\mathsf{Node}(\Omega, [], []).$$

   Next, we associate the value v_2 with the key r in this empty trie. Finally, we append the character c to the end of the list $[c_1, \cdots, c_n]$ and append the trie

   $$\mathsf{Node}(\Omega, [], []).\mathit{insert}(r, v_2)$$

   to the end of the list $[t_1, \cdots, t_n]$.

5.4.2 Deletion in Tries

Finally, we present formulæ that specify how a key can be deleted from a trie. To this end, we define the auxiliary function

$$\mathit{isEmpty} : \mathbb{T} \rightarrow \mathbb{B}.$$

For a trie t, we have $t.\mathit{isEmpty}() = \texttt{true}$ if and only if the trie t does not store any key. The following formulæ specify the function *isEmpty*:

1. $\mathsf{Node}(\Omega, [], []).\mathit{isEmpty}() = \texttt{true}$,
2. $v \neq \Omega \rightarrow \mathsf{Node}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\mathit{isEmpty}() = \texttt{false}$,
3. $\mathsf{Node}(\Omega, L, T).\mathit{isEmpty}() = \mathit{isEmptyList}(T)$

In the last formula we have used the auxiliary function

$$\mathit{isEmptyList} : \mathit{List}(\mathbb{T}) \rightarrow \mathbb{B}.$$

For a list of tries, this function checks whether all tries in this list are empty. Formally, $\mathit{isEmptyList}(l)$ is defined by induction on the length of the list l.

1. $\mathit{isEmptyList}\big([]\big) = \texttt{true}$,
2. $\mathit{isEmptyList}\big([t] + r\big) = \big(t.\mathit{isEmpty}() \wedge \mathit{isEmptyList}(r)\big)$,

   because all tries in the list $[t] + r$ are empty if t is an empty trie and, furthermore, all tries in r are empty.

Now, we can specify the method

$$\mathit{delete} : \mathbb{T} \times \Sigma^* \to \mathbb{T}.$$

For a trie $t \in \mathbb{T}$ and a string $w \in \Sigma^*$ the value of

$$t.\mathit{delete}(w)$$

is defined by induction on the length of w.

1. $\mathsf{Node}(v, L, T).\mathit{delete}(\varepsilon) = \mathsf{Node}(\Omega, L, T)$

   The value that is associated with the empty string ε is stored at the root of the trie where is can be deleted without further ado.

2. $$\begin{array}{ll} t_i.\mathit{delete}(r).\mathit{isEmpty}() & \to \\ \mathsf{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]).\mathit{delete}(c_i r) & = \\ \quad \mathsf{Node}(v, [c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n], [t_1, \cdots, t_{i-1}, t_{i+1}, \cdots, t_n]). & \end{array}$$

   If the key that is to be deleted starts with the character c_i and if deletion of the key r in the ith trie t_i yields an empty trie, then both the ith character c_i and the ith trie t_i are deleted.

3. $$\begin{array}{ll} \neg t_i.\mathit{delete}(r).\mathit{isEmpty}() & \to \\ \mathsf{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i, \cdots, t_n]).\mathit{delete}(c_i r) & = \\ \quad \mathsf{Node}(v, [c_1, \cdots, c_i, \cdots, c_n], [t_1, \cdots, t_i.\mathit{delete}(r), \cdots, t_n]). & \end{array}$$

   If the key that is to be deleted starts with the character c_i and if deletion of the key r in the ith trie t_i yields a non-empty trie, then the key r has to be deleted recursively in the trie t_i.

4. $c \notin C \to \mathsf{Node}(v, C, T).\mathit{delete}(cr) = \mathsf{Node}(v, C, T).$

   If the key that is to be deleted starts with the character c and if c does not occur in the list of characters C, then the trie does not contain the key cr and therefore there is nothing to do: The trie is left unchanged.

5.4.3 Complexity

It is straightforward to see that the complexity of looking up the value associated with a string s of length k is $\mathcal{O}(k)$. In particular, it is independent on the number of strings n. As it is obvious that we have to check all k characters of the string s, this bound cannot be improved. Another advantage of tries is the fact that they use very little storage to store the keys because common prefixes are only stored once.

5.4.4 Implementing Tries in SetlX

We proceed to discuss the implementation of tries. Figure 5.19 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

$$\mathsf{Node}(v, C, T)$$

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.

```
1   class map() {
2       mValue := om;
3       mChars := [];
4       mTries := [];
5
6     static {
7       find    := procedure(s)    { ... };
8       insert  := procedure(s, v) { ... };
9       delete  := procedure(s)    { ... };
10      isEmpty := procedure()     { ... };
11    }
12  }
```

Figure 5.19: Outline of the class `trieMap`.

The class `trieMap` implements the abstract data type `map` and therefore provides the methods `find`, `insert`, and `delete`. Furthermore, 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.

```
1   find := procedure(s) {
2       match (s) {
3           case ""   : return mValue;
4           case [c|r]: for (i in [1 .. #mChars]) {
5                           if (mChars[i] == c) {
6                               return mTries[i].find(r);
7                           }
8                       }
9                       return;  // not found
10      }
11  };
```

Figure 5.20: 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 the character c in the list `mChars`, then the method `find` will just return `om` in line 9.

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.

```
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 5.21: Implementation of `insert` for tries.

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`.

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. In order to remove this value, the variable `mValue` is set to `om`.

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` 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`$[i]$.

```
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 5.22: Implementation of `delete` for tries.

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 5.24.

```
1   isEmpty := procedure() {
2       return mValue == om && mChars == [];
3   };
```

Figure 5.23: 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 5.24: The function to remove the i-th element from a list.

Finally, the implementation of `removeIthElement`, which is shown in Figure 5.24, is straightforward.

Exercise 15: **Binary Tries**: Let us assume that our alphabet is the binary alphabet, i.e. the alphabet only contains the two digits 0 and 1. Therefore we have $\Sigma = \{0, 1\}$. Every natural number can be regarded as a string from the alphabet Σ, so that numbers are effectively elements of Σ^*. The set BT of *binary tries* is defined by induction:

1. *Nil* $\in \mathrm{BT}$.

2. $\mathit{Bin}(v, l, r) \in \mathrm{BT}$ provided that

   (a) $v \in \mathit{Value} \cup \{\Omega\}$ and

   (b) $l, r \in \mathrm{BT}$.

The semantics of binary tries is fixed by defining the function

$$\mathit{find} : \mathrm{BT} \times \mathbb{N}_0 \to \mathit{Value} \cup \{\Omega\}.$$

Given a binary trie b and a natural number n, the expression

$$b.\mathit{find}(n)$$

returns the value in b that is associated with the number n. If there is no value associated with b, then the expression evaluates to Ω. Formally, the value of the expression $b.\mathit{find}(n)$ is defined by induction on b. The induction step requires a side induction with respect to n.

1. $\mathit{Nil}.\mathit{find}(n) = \Omega$,

   since the empty trie doesn't store any values.

2. $\mathit{Bin}(v, l, r).\mathit{find}(0) = v$,

   because 0 is interpreted as the empty string ε.

3. $\mathit{Bin}(v, l, r).\mathit{find}(2 \cdot n + 2) = l.\mathit{find}(n)$,

   because if a number is represented in binary, then the last bit of every even number is zero and zero chooses the left subtree.

4. $\mathit{Bin}(v, l, r).\mathit{find}(2 \cdot n + 1) = r.\mathit{find}(n)$,

   because if a number is represented in binary, then the last bit of every odd number is 1 and 1 is associated with the right subtree.

Solve the following exercises:

(a) Provide equations that specify the methods `insert` and `delete` in a binary trie. When specifying delete you should take care that empty binary trees are reduced to *Nil*.

Hint: It might be helpful to provide an auxiliary method that simplify those binary tries that are empty.

(b) Implement binary tries in SetlX.

Remark: Binary tries are known as *digital search trees*. ◇

5.5 Hash Tables

It is very simple to implement a function of the form

$$f : \mathit{Key} \to \mathit{Value}$$

provided the set *Key* is a set of natural numbers of the form

$$\mathit{Key} = \{1, 2, \cdots, n\}.$$

In this case, we can implement the function f via an array of size n. Figure 5.25 shows how a map can be realized in this case.

If the domain $D := \mathit{dom}(f)$ of the function f is not a set of the form $\{1, \cdots, n\}$, then we can instead try to find a one-to-one mapping of D onto a set of the form $\{1, \cdots, n\}$. Let us explain the idea with a simple example: Suppose we wanted to implement an electronic telephone book. To simplify things, let us assume first that all the names stored in our telephone dictionary have a length of 8

```
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 5.25: Implementing a map as an array.

characters. To achieve this, names that are shorter than eight characters are filled with spaces and if a name has more than eight characters, all characters after the eighth character are dropped.

Next, every name is translated into an index. In order to do so, the different characters are interpreted as digits in a system where the digits can take values starting from 0 up to the value 26. Let us assume that the function *ord* takes a character from the set

$$\Sigma = \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\}$$

and assigns a number from the set $\{0, \cdots, 26\}$ to this character, i.e. we have

$$\mathit{ord} : \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\} \to \{0, \cdots, 26\}.$$

Then, the value of the string $w = c_0 c_1 \cdots c_7$ can be computed by the function

$$\mathit{code} : \Sigma^* \to \mathbb{N}$$

as follows:

$$\mathit{code}(c_0 c_1 \cdots c_7) = 1 + \sum_{i=0}^{7} \mathit{ord}(c_i) \cdot 27^i.$$

Then, the function *code* maps the set of all non-empty strings with at most eight characters in a one-to-one way to the set of numbers $\{1, \cdots, 27^8\}$.
Unfortunately, this naive implementation has several problems:

1. The array needed to store the telephone dictionary has a size of

   $$27^8 = 282\,429\,536\,481$$

   entries. Even if every entry only needs 8 bytes, we still would need more than one terabyte of memory.

2. If two names happen to differ only after their eighth character, then we would not be able to store both of these names as we would not be able to distinguish between them.

These problems can be solved as follows:

1. We have to change the function `code` so that the result of this function is always less than or equal to some given number `size`. Here, the number `size` specifies the number of entries of the array that we intend to use. This number will be in the same order of magnitude as the number of key-value pairs that we want to store in our dictionary.

   There is a simple way to adapt the function *code* so that its result is never bigger than a given number `size`: If we define *code* as

   $$\mathit{code}(c_0 c_1 \cdots c_n) = \left(\sum_{i=0}^{n} \mathit{ord}(c_i) \cdot 27^i\right) \;\%\; \texttt{size} + 1,$$

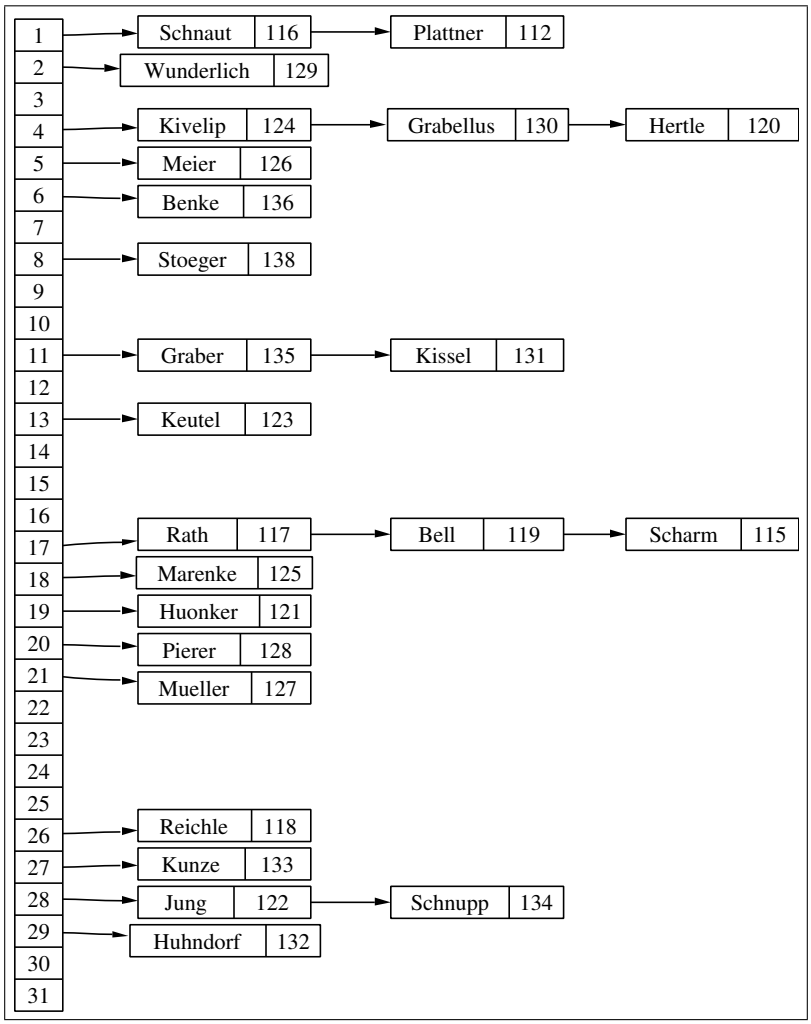

Figure 5.26: A hash table.

then we will always have $\mathit{code}(c_0 c_1 \cdots c_n) \leq \texttt{size}$. In order to prevent overflows we can define the partial sum s_k for $k = n, n-1, \cdots, 1, 0$ by induction:

(a) $s_n = \mathit{ord}(c_n)$,

(b) $s_k = \big(\mathit{ord}(c_k) + s_{k+1} \cdot 27\big) \;\%\; \texttt{size}$.

Then we have $$s_0 + 1 = \left(\sum_{i=0}^{n} \mathit{ord}(c_i) \cdot 27^i\right) \;\%\; \texttt{size} + 1.$$

2. Rather than storing the values associated with the keys in an array, the values are now stored in linked lists that contain key-value pairs. The array only stores pointers to these linked list.

   The reason we have to use linked lists is the fact that different keys may be mapped to the same index. Hence, we can no longer store the values directly in the array. Instead, the values of all keys that map to the same index are stored in a linked list of key-value pairs. These linked lists are then stored in the array. As long as these lists contain only a few entries, the look-up of a key is still fast: Given a key k, we first compute

   $$\texttt{idx} = \mathit{code}(k).$$

   Then, `array[idx]` returns a linked list containing a pair of the form $\langle k, v \rangle$. In order to find

the value associated with the key k we have to search this list for the key k.

Figure 5.26 on page 101 shows an array that contains linked lists. An array of this kind is called a *hash table*. We proceed to discuss the implementation of hash tables 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 5.27: Outline of the class `hashTable`.

Figure 5.27 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 empty, `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 is associated with at most one value, 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 this 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 increase the size of the array `mArray`. To determine the best value for `mAlpha`, we have to make a tradeoff: If `mAlpha` were too big, many entries in the array `mArray` would be empty and thus we would waste space. On the other hand, if `mAlpha` were 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` gets 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 5.28: Implementation of the method `hashCode`.

Figure 5.28 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 `hashCodeAux`(s). As the values returned by `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 `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 `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 5.29: Implementation of `find`.

Figure 5.29 shows 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.

```
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 5.30: Implementation of the method `insert`.

Figure 5.30 shows the implementation of the method `insert`. The implementation works as follows.

1. First, we check whether our hash 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 to `insert`.
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.

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 5.31: Implementation of the method `rehash`.

Figure 5.31 shows 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 hash 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 5.32: Implementation of the procedure `delete`(*map*, *key*).

Finally, we discuss the implementation of the method `delete` that is shown in Figure 5.32. 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 a certain threshold. 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 counterproductive.

If our implementation had used linked lists instead of functional relations then the complexity of the methods *find*, *insert* and *delete* could grow linearly with the number of entries in the hash table. This would happen if the function hashCode(k) would return the same number for all keys k. Of course, this case is highly unlikely, but it is not impossible. If we have a good function to compute hash codes, then most of the linked lists will have roughly the same length. The average length of a list is then

$$\alpha = \frac{\texttt{mEntries}}{\texttt{mSize}}.$$

Here, the number α is the *load factor* of the hash table. In practice, in order to achieve good performance, α should be less than 4. The implementation of the programming language *Java* provides the class *HashMap* that implements maps via hash tables. The default load factor used in this class is only `0.75`.

5.5.1 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. If, instead, a good implementation of red-black-trees is used, the program might be slower in general but is 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.

5.6 Applications

Both C++ and *Java* provide 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] maps 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.

Chapter 6

Priority Queues

In order to introduce *priority queues*, we first take a look at ordinary *queues*. Basically, a queue can be viewed as a list with the following restrictions:

1. A new element can only be appended at the end of the list.
2. Only the element at the beginning of the list can be removed.

This is similar to the queue at a cinema box office. There, a queue is a long line of people waiting to buy a ticket. The person at the front of the queue is served and thereby removed from the queue. New persons entering the cinema have to line up at the end of the queue. In contrast, a priority queue is more like a dentist's waiting room. If you have an appointment at 10:00 and you have already waited for an hour, suddenly a patient with no appointment but a private insurance shows up. Since this patient has a higher priority, he will then be attended next while you have to wait for another hour.

Priority queues have many applications in computer science. We will make use of priority queues, first, when implementing Huffman's algorithm for data compression and, second, when we implement Dijkstra's algorithm for finding the shortest path in a weighted graph. Furthermore, priority queues are used in simulation systems and in operating systems when scheduling processes. Finally, the sorting algorithm *heapsort* is based on priority queues.

6.1 Formal Definition of the ADT *PrioQueue*

Next, we give a formal definition of the ADT *PrioQueue*. Since the data type *PrioQueue* is really just an auxiliary data type, the definition we give is somewhat restricted: We will only specify those functions that are needed for Dijkstra's algorithm. These functions will also be sufficient for Huffman's algorithm.

Definition 15 (Priority Queue)
The abstract data type of priority queues is defined as follows:

1. The name is *PrioQueue*.
2. The set of type parameters is
   $$\{Priority, Value\}.$$
   Furthermore, there must exist a linear ordering $\leq$ on the set *Priority*. This is needed since we want to compare the priority of different elements.
3. The set of function symbols is
   $$\{prioQueue, insert, remove, top\}.$$
4. The type specifications of these function symbols is given as follows:

   (a) *prioQueue* : *PrioQueue*
   This function is the constructor. It creates a new, empty priority queue.
   (b) *top* : *PrioQueue* $\rightarrow$ (*Priority* $\times$ *Value*) $\cup \{\Omega\}$
   The expression Q.*top*() returns a pair $\langle p, v\rangle$. Here, v is any element of Q that has a maximal priority among all elements in Q, while p is the priority associated with v.
   (c) *insert* : *PrioQueue* $\times$ *Priority* $\times$ *Value* $\rightarrow$ *PrioQueue*
   The expression Q.*insert*(p, v) inserts the element v into the priority queue Q. Furthermore, the priority of v is set to be p.
   (d) *remove* : *PrioQueue* $\rightarrow$ *PrioQueue*
   The expression Q.*remove*() removes from Q the element that is returned by Q.top().

5. Before we are able to specify the behaviour of the functions implementing the function symbols given above, we have to discuss the notion of *priority*. We assume that there exists a set *Priority* and there is a linear order $<$ defined on this set. If $p_1 < p_2$, then the priority p_1 is *higher* than the priority p_2. This nomenclature might seem counter intuitive. It is motivated by Dijkstra's algorithm which is discussed later. In Dijkstra's algorithm, the priorities are distances in a graph and the priority of a node is higher if the node is nearer to the source node.

   Next, the behaviour of the functions is specified via a reference implementation. We demand that the behaviour of the functions making up the ADT *PrioQueue* is the same as the behaviour of the reference implementation.

   In our reference implementation a priority queue is implemented as a set of pairs of the form $\langle p, v\rangle$ where p is a priority, while v is a value. The reference implementation proceeds as follows:

   (a) *prioQueue*() $= \{\}$
   The constructor returns an empty priority queue.
   (b) Q.*insert*$(p, v) = Q \cup \{\langle p, v\rangle\}$,
   In order to insert an element v with priority p into the priority queue Q it is sufficient to add the pair $\langle p, v\rangle$ to the set Q.
   (c) If Q is empty, then Q.*top*() is undefined:
   $$Q = \{\} \;\rightarrow\; Q.\mathit{top}() = \Omega.$$
   (d) If Q is not empty, then the expression Q.*top*() returns a pair $\langle p_2, v\rangle$ from the set such that p_2 has the highest priority. Hence we must have:
   $$\langle p_1, v_1\rangle \in Q \;\wedge\; Q.\mathit{top}() = \langle p_2, v_2\rangle \;\rightarrow\; p_2 \leq p_1 \;\wedge\; \langle p_2, v_2\rangle \in Q.$$
   (e) If Q is empty, then calling *remove*() does not change this fact:
   $$Q = \{\} \rightarrow Q.\mathit{remove}() = Q.$$
   (f) Otherwise, Q.*remove*() removes the pair form Q that is returned by Q.*top*():
   $$Q \neq \{\} \rightarrow Q.\mathit{remove}() = Q \backslash \{Q.\mathit{top}()\}.$$

We could implement the ADT *PrioQueue* as a list of pairs that is sorted ascendingly. Then, the different methods of *PrioQueue* would be implemented as follows:

1. *prioQueue*() returns an empty list.
2. Q.*insert*(d) can be implemented via the function `insert` that we have used when implementing the algorithm "*insertion sort*".
3. Q.*top*() return the first element from the list Q.
4. Q.*remove*() removes the first element from Q.

The worst case complexity of this approach would be linear for the method *insert*(), i.e. it would have complexity $\mathcal{O}(n)$ where n is the number of elements in Q. All other operations would have the complexity $\mathcal{O}(1)$. Next, we introduce a more efficient implementation such that the complexity of *insert*() is only $\mathcal{O}\big(\log(n)\big)$. To this end, we introduce *Heaps* next.

6.2 The *Heap* Data Structure

We define the set *Heap*[1] inductively as a subset of the set $\mathcal{B}$ of binary trees. To this end, we first define a relation

$$\leq \subseteq \mathit{Priority} \times \mathcal{B}$$

For a priority $p \in \mathit{Priority}$ and a binary tree $b \in \mathcal{B}$ we have $p \leq b$ if and only if $p \leq q$ for every priority q occurring in b. The formal definition of $p \leq b$ is as follows:

1. $p \leq \mathit{Nil}$,

   because there are no priorities in the empty tree *Nil*.

2. $p \leq \mathit{Node}(q, v, l, r) \stackrel{\text{def}}{\longleftrightarrow} p \leq q \ \wedge \ p \leq l \ \wedge \ p \leq r$,

   because p is less than or equal to every priority in the binary tree $\mathit{Node}(q, v, l, r)$ iff $p \leq q$ and if, furthermore, p is less than or equal to every priority occurring in either l or r.

Next, we define a function

$$\mathit{count} : \mathcal{B} \to \mathbb{N}_0,$$

that counts the number of nodes occurring in a binary tree b. The definition of $b.\mathit{count}()$ is given by induction on b.

1. $\mathit{Nil}.\mathit{count}() = 0$.
2. $\mathit{Node}(p, v, l, r).\mathit{count}() = 1 + l.\mathit{count}() + r.\mathit{count}()$.

Now we are ready to define the set *Heap* by induction:

1. $\mathit{Nil} \in \mathit{Heap}$.
2. $\mathit{Node}(p, v, l, r) \in \mathit{Heap}$ if and only if the following is true:
   (a) $p \leq l \ \wedge \ p \leq r$

   The priority stored at the root is less than or equal to every other priority stored in the heap. This condition is known as the *heap condition*.

   (b) $| \, l.\mathit{count}() - r.\mathit{count}() \, | \leq 1$

   The number of elements in the left subtree differs from the number of elements stored in the right subtree by at most one. This condition is known as the *balancing condition*. It is similar to the balancing condition of AVL trees, but instead of comparing the heights, this condition compares the number of elements.

   (c) $l \in \mathit{Heap} \ \wedge \ r \in \mathit{Heap}$

   This condition ensures that all subtrees are heaps, too.

The *heap condition* implies that in a non-empty heap the element with a highest priority is stored at the root. Figure 6.1 on page 110 shows a simple heap. In the upper part of the nodes we find the priorities. Below these priorities we have the values that are stored in the heap. In the example given, the priorities are natural numbers, while the values are characters.

As heaps are binary trees, we can implement them in a fashion that is similar to our implementation of AVL trees. In order to do so, we first present equations that specify the methods of the data structure heap. We start with the method *top*.

1. $\mathit{Nil}.\mathit{top}() = \Omega$.

[1] In computer science, the notion of a *Heap* is used for two different concepts: First, a *heap* is a data structure that is organized as a tree. This kind of data structure is described in more detail in this section. Second, the part of main memory that contains dynamically allocated objects is known as *the heap*.

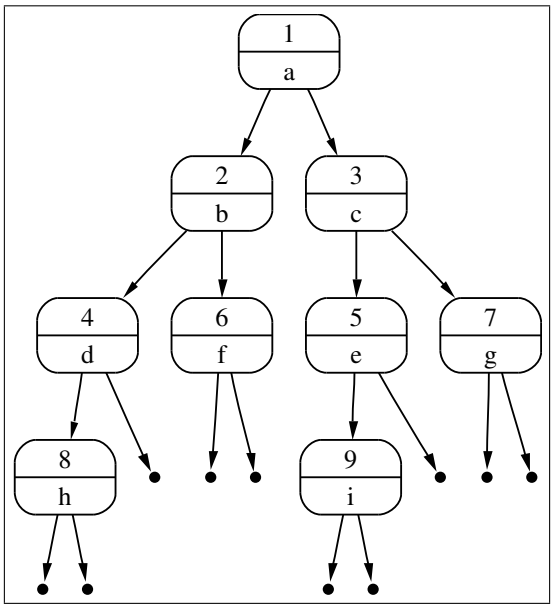

Figure 6.1: A heap.

2. $\mathit{Node}(p, v, l, r).\mathit{top}() = \langle p, v \rangle$,

   because the heap condition ensures that the value with the highest priority is stored at the top.

When implementing the method *insert* we have to make sure that both the balancing condition and the heap condition are maintained.

1. $\mathit{Nil}.\mathit{insert}(p, v) = \mathit{Node}(p, v, \mathit{Nil}, \mathit{Nil})$.

2. $p_{\text{top}} \leq p \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p_{\text{top}}, v_{\text{top}}, l.\mathit{insert}(p, v), r\big)$.

   If the value v to be inserted has a priority that is lower (or the same) than the priority of the value at the root of the heap, we have to insert the value v either in the left or right subtree. In order to maintain the balancing condition, we insert the value v in the left subtree if that subtree stores at most as many values as the right subtree.

3. $p_{\text{top}} \leq p \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p_{\text{top}}, v_{\text{top}}, l, r.\mathit{insert}(p, v)\big)$.

   If the value v to be inserted has a priority that is lower (or the same) than the priority of the value at the root of the heap, we have to insert the value v in the right subtree if the right subtree stores fewer values than the left subtree.

4. $p_{\text{top}} > p \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p, v, l.\mathit{insert}(p_{\text{top}}, v_{\text{top}}), r\big)$.

   If the value v to be inserted is associated with a priority p that is higher than the priority of the value stored at the root of the heap, then we have to store the value v at the root. The value v_{top} that was stored previously at the root has to be moved to either the left or right subtree. If the number of nodes in the left subtree is as most as big as the number of nodes in the right subtree, v_{top} is inserted into the left subtree.

5. $p_{\text{top}} > p \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
   $\mathit{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(p, v) = \mathit{Node}\big(p, v, l, r.\mathit{insert}(p_{\text{top}}, v_{\text{top}})\big)$.

If the value v to be inserted is associated with a priority p that is higher than the priority of the value stored at the root of the heap, then we have to store the value v at the root. The value v_{top} that was stored previously at the root has to be moved to the right subtree provided the number of nodes in the left subtree bigger than the number of nodes in the right subtree.

Finally, we specify our implementation of the method *remove*.

1. $\mathit{Nil}.\mathit{remove}() = \mathit{Nil}$,

   since we cannot remove anything from the empty heap.

2. $\mathit{Node}(p, v, \mathit{Nil}, r).\mathit{remove}() = r$,

3. $\mathit{Node}(p, v, l, \mathit{Nil}).\mathit{remove}() = l$,

   because we always remove the value with the highest priority and this value is stored at the root. Now if either of the two subtrees is empty, we can just return the other subtree.

   Next, we discuss those cases where none of the subtrees is empty. In that case, either the value that is stored at the root of the left subtree or the value stored at the root of the right subtree has to be promoted to the root of the tree. In order to maintain the heap condition, we have to take the value that is associated with the higher priority.

4. $p_1 \leq p_2 \ \wedge \ l = \mathit{Node}(p_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{Node}(p_2, v_2, l_2, r_2) \ \rightarrow$
   $\mathit{Node}(p, v, l, r).\mathit{remove}() = \mathit{Node}(p_1, v_1, l.\mathit{remove}(), r)$,

   because if the value at the root of the left subtree has a higher priority than the value stored at the right subtree, then the value at the left subtree is moved to the root of the tree. Of course, after moving this value to the root, we have to recursively delete this value from the left subtree.

5. $p_1 > p_2 \ \wedge \ l = \mathit{Node}(p_1, v_1, l_1, r_1) \ \wedge \ r = \mathit{Node}(p_2, v_2, l_2, r_2) \rightarrow$
   $\mathit{Node}(p, v, l, r).\mathit{remove}() = \mathit{Node}(p_2, v_2, l, r.\mathit{remove}())$

   This case is similar to the previous case, but now the value from the right subtree moves to the root.

The hawk-eyed reader will have noticed that the specification of the method delete that is given above violates the balancing condition. It is not difficult to change the implementation so that the balancing condition is maintained. However, it is not really necessary to maintain the balancing condition when deleting values. The reason is that the balancing condition is needed as long as the heap grows in order to guarantee logarithmic performance. However, when we remove values from a priority queue, the height of the queue only shrinks. Therefore, even if the heap would degenerate into a list during removal of values, this would not be a problem because the height of the tree would still be bounded by $\log_2(n)$, where n is the maximal number of values that a stored in the heap at any moment in time.

Exercise 16: Change the equations for the method `remove` so that the resulting heap satisfies the balancing condition.

6.3 Implementing *Heaps* in SetlX

Next, we present an implementation of heaps in SetlX. Figure 6.2 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

```
1   class heap() {
2       mPriority := om;
3       mValue    := om;
4       mLeft     := om;
5       mRight    := om;
6       mCount    := 0;
7
8     static {
9         top     := procedure()     { return [mPriority, mValue]; };
10        insert  := procedure(p, v) { ... };
11        remove  := procedure()     { ... };
12        update  := procedure(t)    { ... };
13        isEmpty := [] |-> mCount == 0;
14    }
15  }
```

Figure 6.2: Outline of the class heap.

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 6.3 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 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 lose 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 6.4 shows the implementation of the method remove. This method removes the value with the highest priority from the heap. Essentially, there are two cases.

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] 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   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 6.3: Implementation of the method `insert`.

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.

Figure 6.5 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 17: 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 18: 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.

```
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 6.4: Implementation of the method remove.

```
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 6.5: Implementation of the method update.

6.4 Heap-Sort

Heaps can be used to implement a sorting algorithm that is efficient in terms of both time and memory. While merge sort needs only $n \cdot \log_2(n)$ comparisons to get the job done, the algorithm uses an auxiliary array and is therefore not optimally efficient with regard to its memory consumption. The algorithm we describe next, heapsort has a time complexity that is $\mathcal{O}\big(n \cdot \log_2(n)\big)$ and that does not require an auxiliary array. Heapsort was invented by J. W. J. Williams and Robert W. Floyd in 1964.

Given an array a of keys to be sorted, the basic version of Heapsort proceeds as follows:

1. The elements of a are inserted in a heap h.
2. Now the smallest element of a is at the top of h. Therefore, if we remove the elements from h one by one, we retrieve these elements in increasing order.

A basic implementation of heapsort that is based on this idea is given in Figure 6.6 on page 115. This implementation makes use of the class heap that had been presented in the previous section.

1. In order to sort the list a that is given as argument to heapSort, we first create the empty heap h in line 2 and then proceed to insert all elements of a into h in line 4.

2. Next we create an empty list s in line 5. When the procedure heapSort finishes, this list will be a sorted version of the list a.

3. As long as the heap h is not empty, we take its top element and append it to s. Since the method top returns a pair of the form $\langle p, v \rangle$ and we are only interested in the priority p, we just add the first element of this pair to the end of the list s. After we have appended p to the list s, it is removed from the heap h.

4. When the heap h is empty, s contains all of the elements of the list a sorted ascendingly.

```
1   heapSort := procedure(a) {
2       h := heap();
3       for (x in a) {
4           h.insert(x, x);
5       }
6       s := [];
7       while (!h.isEmpty()) {
8           s += [ h.top()[1] ];
9           h.remove();
10      }
11      return s;
12  };
```

Figure 6.6: A basic version of heapsort.

The basic version of heapsort that is shown in Figure 6.6 can be improved by noting that a heap can be stored efficiently in an array a. If a node of the form $\mathsf{Node}(p, v, l, r)$ is stored at index i, then the left subtree l is stored at index $2 \cdot i$ while the right subtree r is stored at index $2 \cdot i + 1$:

$$\mathtt{a}[i] = \mathsf{Node}(p, v, l, r) \;\rightarrow\; \mathtt{a}[2 \cdot i] = l \;\wedge\; \mathtt{a}[2 \cdot i + 1].$$

If we store a heap in this manner, then instead of using pointers that point to the left and right subtree of a node we can just use index arithmetic to retrieve the subtrees.

Figure 6.7 on page 115 makes use of this idea. We discuss this implementation line by line.

1. The function swap exchanges the elements in the array a that are at the positions x and y.

2. The procedure sink takes three arguments.

   (a) k is an index into the array a.

   (b) a is an array.

   (c) n is the size of the array.

   The array a might actually have more than n elements, but for the purpose of the method sink we pretend that a has exactly n elements.

   When calling sink, the assumption is that a[k:n] should represent a heap that possibly has its heap condition violated at its root, i.e. at index k. The purpose of the procedure sink is to restore the heap condition at index k. To this end, we first compute the index j of the left subtree below index k. Then we check whether there also is a right subtree at position j+1, which is the case if j is less than n. Now if the heap condition is violated at index k, we have to exchange the element at position k with the child that has the higher priority, i.e. the child that is smaller. Therefore, in line 8 we arrange for index j to point to the smaller child. Next, we check in line 10 whether the heap condition is violated at index k. If the heap condition is satisfied, there is nothing to do and we return. Otherwise, the element at position k is swapped with the

```
1   swap := procedure(x, y, rw a) {
2       [a[x], a[y]] := [a[y], a[x]];
3   };
4   sink := procedure(k, rw a, n) {
5       while (2 * k <= n) {
6           j := 2 * k;
7           if (j < n && a[j] > a[j+1]) {
8               j += 1;
9           }
10          if (a[k] < a[j]) {
11              break;
12          }
13          swap(k, j, a);
14          k := j;
15      }
16  };
17  heapSort := procedure(rw a) {
18      n := #a;
19      for (k in [n\2, n\2-1 .. 1]) {
20          sink(k, a, n);
21      }
22      while (n > 1) {
23          swap(1, n, a);
24          n -= 1;
25          sink(1, a, n);
26      }
27  };
```

Figure 6.7: An implementation of Heapsort in SetlX.

element at position j. Of course, now it could happen that the heap condition is violated at position j. Therefore, we set k to j. As long as the node at position k has a child, i.e. as long as $2 \cdot k \leq n$ the while-loop continues.

3. The procedure heapSort has the task to sort the array a and proceeds in two phases.

   (a) In phase 1 our goal is to transform the array a into a heap that is stored in a.

   In order to do so, we traverse the array a in reverse using the for-loop starting in line 19. The invariant of this loop is that before sink is called, all trees rooted at an index greater than k satisfy the heap condition. Initially this is true because the trees that are rooted at indices greater than n\2 are trivial, i.e. they only consist of their root node. Then, since there are no children below these nodes, the heap condition is satisfied vacuously.

   In order to satisfy the invariant for index k, sink is called with argument k, since at this point, the tree rooted at index k satisfies the heap condition except possibly at the root. It is then the job of sink to establish the heap condition for index k. If the element at the root has a priority that is too low, sink ensures that this element sinks down in the tree as far as necessary.

   (b) In phase 2 we remove the elements from the heap one-by-one and insert them at the end of the array.

   When the while-loop starts, the array a contains a heap. Therefore, the smallest element is found at the bottom of the heap. Since we want to sort the array a descendingly, we move this element to the end of the array a and in return move the element from the end

of the array `a` to the front. After this exchange, the sublist `a[1..n-1]` represents a heap, except that the heap condition might now be violated at the root. Next, we decrement `n` in line 24, since the last element of the array `a` is already in its correct position. In order to reestablish the heap condition at the root, we call `sink` with index `1` in line 25.

The `while`-loop runs as long as the part of the array that has to be sorted has a length bigger than 1. If there is only one element left, this element is trivially sorted and the function returns.

Chapter 7

Data Compression

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 k verschiedenen Zeichen besteht und wir alle Buchstaben mit derselben Länge von b Bits kodieren wollen, dann muss für diese Zahl von Bits offenbar

$$k \leq 2^b$$

gelten, woraus

$$b = \mathit{ceil}\big(\log_2(k)\big)$$

folgt. Hier bezeichnet $\mathit{ceil}(x)$ die *Ceiling-Funktion*. Diese Funktion rundet eine gegebene reelle Zahl immer auf, es gilt also

$$\mathit{ceil}(x) = \min\{k \in \mathbb{Z} \mid x \leq k\}.$$

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 derselben 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.

7.1 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 seltener auftreten, mit einer größeren Anzahl Bits kodiert werden. Zur Verdeutlichung betrachten wir folgendes Beispiel: Unser Alphabet Σ bestehe nur aus vier Buchstaben,

$$\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}.$$

In dem zu speichernden String s trete der Buchstabe a insgesamt 990-mal auf, der Buchstabe b trete 8 mal auf und die Buchstaben c und d treten jeweils 1 mal auf. Dann besteht der String s aus

insgesamt $1\,000$ Buchstaben. Wenn wir jeden Buchstaben mit $2 = \log_2(4)$ Bits kodieren, dann werden also insgesamt $2\,000$ Bits benötigt um den String s abzuspeichern. Wir können den String aber auch mit weniger Bits abspeichern, wenn wir die einzelnen Buchstaben mit Bitfolgen unterschiedlicher Länge kodieren. In unserem konkreten Beispiel wollen wir versuchen den Buchstaben a, der mit Abstand am häufigsten vorkommt, mit einem einzigen Bit zu kodieren. Bei den Buchstaben c und d, die nur sehr selten auftreten, ist es kein Problem auch mehr Bits zu verwenden. Tabelle 7.1 zeigt eine Kodierung, die von dieser Idee ausgeht.

Buchstabe	a	b	c	d
Häufigkeit	990	8	1	1
Kodierung	0	10	110	111

Table 7.1: Kodierung der Buchstaben mit variabler Länge.

Um zu verstehen, wie diese Kodierung funktioniert, stellen wir sie in Abbildung 7.1 als *Kodierungs-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!

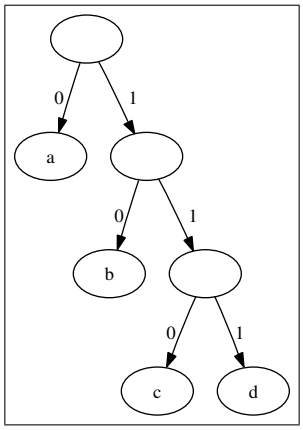

Figure 7.1: Baum-Darstellung der Kodierung.

Um zu sehen, wie mit Hilfe des Kodierungs-Baums ein String dekodiert werden kann, betrachten wir als Beispiel den String "100111". Wir beginnen mit der "1", die uns sagt, vom Wurzel-Knoten dem rechten Pfeil zu folgen. Die anschließende "0" spezifiziert dann den linken Pfeil. Jetzt sind wir bei dem mit "b" markierten Blatt angekommen und haben damit den ersten Buchstaben gefunden. Wir gehen wieder zur Wurzel des Baums zurück. Die folgende "0" führt uns zu dem Blatt, das mit "a" markiert ist, also haben wir den zweiten Buchstaben gefunden. Wir gehen wieder zur Wurzel zurück. Die Ziffern "111" führen uns nun zu dem Buchstaben "d". Damit haben wir insgesamt

"100111" $\simeq$ "bad".

7.2 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 7.1 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 *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 k verschiedenen Buchstaben, so haben wir dann eine Menge von Kodierungs-Bäumen der Form

$$M = \big\{\mathit{Leaf}(c_1, f_1), \cdots, \mathit{Leaf}(c_k, f_k)\big\}. \tag{7.1}$$

Es werden nun solange Knoten a und b aus M zu einem neuen Knoten $\mathit{Node}(a, b)$ zusammengefasst, bis die Menge M nur noch einen Knoten enthält. Offenbar gibt es im Allgemeinen sehr viele Möglichkeiten, die Knoten aus der Menge zu neuen Knoten zusammen zu fassen. Das Ziel ist es die Knoten so zusammen zu fassen, dass die Kosten der Menge M am Ende minimal sind. Um zu verstehen, welche Knoten wir am geschicktesten zusammenfassen können, betrachten wir, wie sich die Kosten der Menge durch das Zusammenfassen zweier Knoten ändert. Dazu betrachten wir zwei Mengen von Knoten M_1 und M_2, so dass

$$M_1 = N \cup \{a, b\} \quad \texttt{und} \quad M_2 = N \cup \{\mathit{Node}(a, b)\}$$

gilt, die Menge M_2 geht also aus der Menge M_1 dadurch hervor, dass wir die Knoten a und b zu einem neuen Knoten zusammenfassen 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 vergrößern 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 $\mathit{Node}(a, b)$ ersetzen. Dieser Prozess wird solange iteriert, bis die Menge M nur noch aus einem Knoten besteht. Dieser Knoten ist dann die Wurzel des gesuchten Kodierungs-Baums.
Die in Abbildung 7.2 gezeigte Funktion `codingTree`(m) implementiert diesen Algorithmus.

1. Die Funktion *codingTree* wird mit einer Menge m von Knoten aufgerufen, welche die Form

   $$m = \big\{\langle f_1, \mathit{Leaf}(c_1)\rangle, \cdots, \langle f_k, \mathit{Leaf}(c_k)\rangle\big\}$$

   hat. Hier bezeichnen die Variablen c_i die verschiedenen Buchstaben, während die Zahl f_i die Häufigkeit angibt, mit der der Buchstabe c_i auftritt.

   Wir haben hier die Information über die Häufigkeit an erster Stelle eines Paares gespeichert. Da SetlX diese Menge intern durch einen geordneten binären Baum abspeichert, ermöglicht uns diese Form der Darstellung einfach auf den Knoten mit der kleinsten Häufigkeit zuzugreifen, denn die Paare werden so verglichen, dass immer zunächst die erste Komponente zweier Paare zum Vergleich herangezogen werden. Nur wenn sich in der ersten Komponente kein Unterschied

```
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 7.2: Der Algorithmus von Huffman in SetlX.

ergibt, wird auch die zweite Komponente verglichen. Daher finden wir das Paar mit der kleinsten ersten Komponente immer am Anfang der Menge m.

Durch diesen Trick haben wir uns de facto die Implementierung einer Prioritäts-Warteschlange gespart:

(a) Die in SetlX vordefinierte Funktion `first`(m) liefert das erste Element der Menge m und entspricht damit der Funktion *top*(m) des abstrakten Daten-Typs *PrioQueue*.

(b) Anstelle von *insert*(m, p, v) können wir einfach

m `+= { [`p`,` v`] };`

schreiben um das Element v mit der Priorität p in die Prioritäts-Warteschlange m einzufügen.

(c) Die Funktion *remove*(m) realisieren wir durch den Aufruf

m `-= { first(`m`) };`

denn *remove*(m) soll ja das Element mit der höchsten Priorität aus m entfernen.

Das Elegante an diesem Vorgehen ist, dass damit sämtliche Operationen des abstrakten Daten-Typs *PrioQueue* eine logarithmische Komplexität haben. Das ist zwar im Falle der Operation *top*(m) nicht optimal, aber für die Praxis völlig ausreichend, denn in der Praxis kommt auf jeden Aufruf der Form *top*(m) auch ein Aufruf der Form *remove*(m) und der hat sowohl bei einer optimalen Implementierung als auch bei unserer Implementierung eine logarithmische Komplexität, die dann auch im Falle der optimalen Implementierung die gesamte Komplexität dominiert.

2. Die `while`-Schleife in Zeile 2 verringert 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 zurückgegeben.

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 $n.\mathit{cost}()$ 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 7.2: Buchstaben mit Häufigkeiten.

Wir illustrieren den Huffman-Algorithmus, indem wir ihn auf die Buchstaben, die in Tabelle 7.2 zusammen mit ihren Häufigkeiten angegeben sind, anwenden.

1. Zu Beginn hat die Menge m die Form

   $$m = \big\{\langle 1, \mathit{Leaf}(\mathtt{a})\rangle,\ \langle 2, \mathit{Leaf}(\mathtt{b})\rangle,\ \langle 3, \mathit{Leaf}(\mathtt{c})\rangle,\ \langle 4, \mathit{Leaf}(\mathtt{d})\rangle,\ \langle 5, \mathit{Leaf}(\mathtt{e})\rangle\big\}.$$

2. Die Häufigkeit ist hier für die Blätter mit den Buchstaben a und b minimal. Also entfernen wir diese Blätter aus der Menge und fügen stattdessen den Knoten

   $$\mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b}))$$

   in die Menge m ein. Die Häufigkeit dieses Knotens ergibt sich als Summe der Häufigkeiten der Buchstaben a und b. Daher fügen wir insgesamt das Paar

   $$\langle 3, \mathit{Node}\big(\mathit{Leaf}(\mathtt{a})\rangle, \mathit{Leaf}(\mathtt{b})\big)\rangle$$

   in die Menge m ein. Dann hat m die Form

   $$\big\{\langle 3, \mathit{Leaf}(\mathtt{c})\rangle,\ \langle 3, \mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b}))\rangle,\ \langle 4, \mathit{Leaf}(\mathtt{d})\rangle,\ \langle 5, \mathit{Leaf}(\mathtt{e})\rangle\big\}.$$

3. Die beiden Paare mit den kleinsten Werten der Häufigkeiten in m sind nun

   $$\langle 3, \mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b}))\rangle \quad \text{und} \quad \langle 3, \mathit{Leaf}(\mathtt{c})\rangle.$$

   Wir entfernen diese beiden Knoten und bilden aus diesen beiden Knoten den neuen Knoten

   $$\langle 6, \mathit{Node}(\mathit{Node}((\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})),\ \mathit{Leaf}(\mathtt{c}))\rangle,$$

   den wir der Menge m hinzufügen. Dann hat m die Form

   $$\Big\{\langle 4, \mathit{Leaf}(\mathtt{d})\rangle,\ \langle 5, \mathit{Leaf}(\mathtt{e})\rangle,\ \langle 6, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})),\ \mathit{Leaf}(\mathtt{c}))\Big\}.$$

4. Jetzt sind

   $$\langle 4, \mathit{Leaf}(\mathtt{d})\rangle \quad \text{und} \quad \langle 5, \mathit{Leaf}(\mathtt{e})\rangle$$

   die beiden Knoten mit dem kleinsten Werten der Häufigkeit. Wir entfernen diese Knoten und bilden den neuen Knoten

   $$\langle 9, \mathit{Node}(\mathit{Leaf}(\mathtt{d}), \mathit{Leaf}(\mathtt{e}))\rangle.$$

   Diesen fügen wir der Menge m hinzu und erhalten

   $$\Big\{\langle 6, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})),\ \mathit{Leaf}(\mathtt{c}, 3))\rangle,\ \langle 9, \mathit{Node}(\mathit{Leaf}(\mathtt{d}, 4), \mathit{Leaf}(\mathtt{e}, 5))\rangle\Big\}.$$

5. Jetzt enthält die Menge m nur noch zwei Knoten. Wir entfernen diese beiden Knoten und bilden daraus den neuen Knoten

$$\mathit{Node}\Big(\mathit{Node}\Big(\mathit{Node}\big(\mathit{Leaf}(\mathtt{a}), \mathit{Leaf}(\mathtt{b})\big),\ \mathit{Leaf}(\mathtt{c})\Big),\ \mathit{Node}\big(\mathit{Leaf}(\mathtt{d}), \mathit{Leaf}(\mathtt{e})\big)\Big)$$

Dieser Knoten ist jetzt der einzige Knoten in m und damit unser Ergebnis. Stellen wir diesen Knoten als Baum dar, so erhalten wir das in Abbildung 7.3 gezeigte Ergebnis. Wir haben hier jeden Knoten n mit dem Funktionswert $n.\mathit{count}()$ beschriftet.

Die Kodierung, die sich daraus ergibt, wird in Tabelle 7.3 gezeigt.

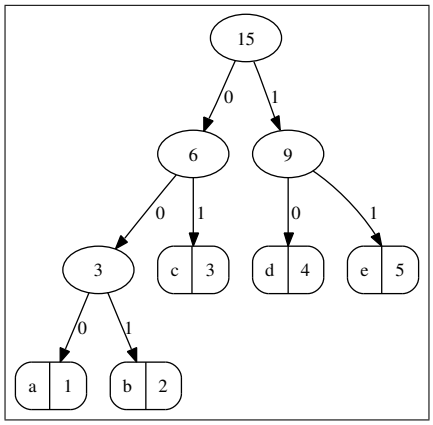

Figure 7.3: Baum-Darstellung der Kodierung.

Buchstabe	a	b	c	d	e
Kodierung	000	001	01	10	11

Table 7.3: Kodierung der Buchstaben mit variabler Länge.

Exercise 19:

(a) Berechnen Sie den Huffman-Code für einen Text, der nur die Buchstaben "a" bis "g" enthält und für den die Häufigkeiten, mit denen diese Buchstaben auftreten, durch die folgende Tabelle gegeben sind.

Buchstabe	a	b	c	d	e	f	g
Häufigkeit	1	1	2	3	5	8	13

Table 7.4: Buchstaben mit Häufigkeiten.

(b) Wie groß ist die Einsparung, wenn man die Buchstaben mit einem Huffman-Code kodiert gegenüber einer Kodierung mit drei Bits?

(c) 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.

(d) Wie groß ist die Einsparung im allgemeinen Fall?

7.3 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

1. s is a substring of the string that is encoded and
2. the substring $s[1..n-1]$ has already been entered into the dictionary.

The algorithm 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 known as the *look-ahead character*.
3. d is 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:

   $$d = d \cup \{\langle \texttt{ma}, 128\rangle\}.$$

   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:

   $$d = d \cup \{\langle \texttt{au}, 129\rangle\}.$$

   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:

   $$d = d \cup \{\langle \texttt{um}, 130 \rangle\}.$$

   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:

   $$d = d \cup \{\langle \texttt{mau}, 131 \rangle\}.$$

   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 that is slightly bigger than 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 of the dictionary constructed when starting to encode the string.
2. We initialize s as the empty string, which is denoted as ε:

   $$s := \varepsilon.$$

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 decode a number that has not yet been entered into the dictionary. To answer this question, we can reason as 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.

7.3.1 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
6      static {
7          compress      := procedure(s)     { ... };
8          uncompress    := procedure(l)     { ... };
9          longestPrefix := procedure(s, i)  { ... };
10     }
11 }
```

Figure 7.4: Outline of the class `lzw`.

Figure 7.4 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 method `longestPrefix`, which will be discussed later. The class `lzw` contains 3 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.

```
1   compress := procedure(s) {
2       result := [];
3       idx    := 1;
4       while (idx <= #s) {
5           p := longestPrefix(s, idx);
6           result += [ mDictionary[s[idx..p]] ];
7           if (p < #s) {
8               mDictionary[s[idx..p+1]] := mNextCode;
9               this.mNextCode += 1;
10          }
11          idx := p + 1;
12      }
13      return result;
14  };
```

Figure 7.5: The method `compress` encodes a string as a list of integers.

Figure 7.5 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 than 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 `mDictionary` and is then appended to the list `result`.

6. 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.

7. Since the code corresponding to the substring $s[\texttt{idx}..p]$ has been written to the list `result`, the index `idx` is set to $p+1$.

8. 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 7.6: Computing the longest prefix.

Figure 7.6 show the implementation of the auxiliary function `longestPrefix`. 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.

```
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 7.7: The method `uncompress` to decode a list of integers into a string.

Figure 7.7 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 20:

(a) Use the LZW algorithm to encode the string "abcabcabcabc". Compute the compression factor for this string.

(b) For all $n \in \mathbb{N}$ with $n \geq 1$ the string α_n is defined inductively as follows:

$$\alpha_1 := \mathsf{a} \quad \text{and} \quad \alpha_{n+1} = \alpha_n + \mathsf{a}.$$

Hence, the string α_n has the form $\underbrace{\mathsf{a} \cdots \mathsf{a}}_{n}$, i.e. it is the character a repeated n times. Encode the string α_n using the LZW algorithm. What is the compression rate?

(c) Decode the list

$$[97, 98, 128, 130]$$

using the LZW algorithm. ◇

Chapter 8

Graph Theory

In this chapter we are going to discuss three graph theoretical problems.

1. We present an algorithm to solve the *union-find problem*. In this problem, we are given a set M and a relation $R \subseteq M \times M$. Our task is then to find the smallest equivalence relation $R^{\approx}$ such that $R \subseteq R^{\approx}$.
2. The next problem we solve is the problem to compute the *minimum spanning tree* of a graph. Given a weighted graph, this problem asks to find the smallest tree that spans the graph.
3. Finally, we discuss the problem of finding a shortest path in a *weighted directed graph*.

8.1 The Union-Find Problem

Assume that we are given a set M together with a relation $R \subseteq M \times M$. The relation R is not yet an equivalence relation on M, but this relation *induces* an equivalence relation $\approx_R$ on M. This *induced equivalence relation* is defined inductively.

1. For every pair $\langle x, y \rangle \in R$ we have that $\langle x, y \rangle \in \approx_R$.

   This is the base case of the inductive definition. It ensures that the relation $\approx_R$ is an extension of the relation R.

2. For every $x \in M$ we have $\langle x, x \rangle \in \approx_R$.

   This clause ensures that the relation $\approx_R$ is reflexive on M.

3. If $\langle x, y \rangle \in \approx_R$, then $\langle y, x \rangle \in \approx_R$.

   This clause ensures that the relation $\approx_R$ is symmetric.

4. If $\langle x, y \rangle \in \approx_R$ and $\langle y, z \rangle \in \approx_R$, then $\langle x, z \rangle \in \approx_R$.

   This clause ensures that the relation $\approx_R$ is transitive.

Given this inductive definition, it can be shown that:

1. $\approx_R$ is an equivalence relation on M.
2. If Q is an equivalence relation on M such that $R \subseteq Q$, then $\approx_R \subseteq Q$.

Therefore, the relation $\approx_R$ is the smallest equivalence relation on M that extends R. In our lesson on linear algebra we had defined the transitive closure R^+ of a binary relation R in a similar way. In that lecture, we had then shown that R^+ is indeed the smallest transitive relation that extends R. This proof can easily be adapted to prove the claim given above.

It turns out that a direct implementation of the inductive definition of $\approx_R$ given above is not very efficient. Instead, we remind ourselves that there is are one-to-one correspondence between an equivalence relations $R \subseteq M \times M$ and a *partition* of M. A set $\mathcal{P} \subseteq 2^M$ is a *partition* of M iff the following holds:

1. $\{\} \notin \mathcal{P}$,
2. $A \in \mathcal{P} \wedge B \in \mathcal{P} \rightarrow A = B \vee A \cap B = \{\}$,
3. $\bigcup \mathcal{P} = M$.

Therefore, a partition $\mathcal{P}$ of M is a subset of the power set of M such that every element of M is a member of exactly one set of $\mathcal{P}$ and, furthermore, $\mathcal{P}$ must not contain the empty set. We have already seen in the lecture on Linear Algebra that an equivalence relation $\approx \subseteq M \times M$ gives rise to *equivalence classes*, where the equivalence class generated by $x \in M$ is defined as

$$[x]_\approx := \{y \mid \langle x, y\rangle \in \approx\}.$$

It was then shown that the set

$$\big\{[x]_\approx \;\big|\; x \in M\big\}$$

is a partition of M. It was also shown that every partition $\mathcal{P}$ of a set M gives rise to an equivalence relation $\approx_\mathcal{P}$ that is defined as follows:

$$x \approx_\mathcal{P} y \iff \exists A \in \mathcal{P} : (x \in A \wedge y \in A).$$

An example will clarify the idea. Assume that

$$M := \{1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Then the set

$$\mathcal{P} := \big\{\{1, 4, 7, 9\}, \{3, 5, 8\}, \{2, 6\}\big\}$$

is a partition of M since the three sets involved are disjoint and their union is the set M. According to this partition, the elements 1, 4, 7, and 9 are all equivalent to each other. Similarly, the elements 3, 5, and 8 are equivalent to each other, and, finally, 2 and 6 are equivalent.

It turns out that, given a relation R, the most efficient way to compute the induced equivalence relation $\approx_R$ is to compute the partition corresponding to this equivalence relation. In order to present the algorithm, we first sketch the underlying idea using a simple example. Assume the set M is defined as

$$M := \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

and that the relation R is given as follows:

$$R := \big\{\langle 1, 4\rangle, \langle 7, 9\rangle, \langle 3, 5\rangle, \langle 2, 6\rangle, \langle 5, 8\rangle, \langle 1, 9\rangle, \langle 4, 7\rangle\big\}.$$

Our goal is to compute a partition $\mathcal{P}$ of M such that the formula

$$\langle x, y\rangle \in R \rightarrow \exists A \in \mathcal{P} : \big(x \in A \wedge y \in A\big)$$

holds. In order to achieve this goal, we define a sequence of partitions $\mathcal{P}_1$, $\mathcal{P}_2$, $\cdots$, $\mathcal{P}_n$ such that $\mathcal{P}_n$ achieves our goal.

1. We start by defining

   $$\mathcal{P}_1 := \big\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\big\}.$$

   This is clearly a partition of M, but it is the trivial one since it induces an equivalence relation $\approx$ where we have $x \approx y$ only if $x = y$.

2. Next, we have to ensure to incorporate our given relation R into this partition. Since $\langle 1, 4\rangle \in R$ we replace the singleton sets $\{1\}$ and $\{4\}$ by their union. This leads to the following definition

of the partition $\mathcal{P}_2$:

$$\mathcal{P}_2 := \big\{\{1,4\},\{2\},\{3\},\{5\},\{6\},\{7\},\{8\},\{9\}\big\}.$$

3. Since $\langle 7,9\rangle \in R$, we replace the sets $\{7\}$ and $\{9\}$ by their union and define
$$\mathcal{P}_3 := \big\{\{1,4\},\{2\},\{3\},\{5\},\{6\},\{7,9\},\{8\}\big\}.$$
4. Since $\langle 3,5\rangle \in R$, we replace the sets $\{3\}$ and $\{5\}$ by their union and define
$$\mathcal{P}_4 := \big\{\{1,4\},\{2\},\{3,5\},\{6\},\{7,9\},\{8\}\big\}.$$
5. Since $\langle 2,6\rangle \in R$, we replace the sets $\{2\}$ and $\{6\}$ by their union and define
$$\mathcal{P}_5 := \big\{\{1,4\},\{2,6\},\{3,5\},\{7,9\},\{8\}\big\}.$$
6. Since $\langle 5,8\rangle \in R$, we replace the sets $\{3,5\}$ and $\{8\}$ by their union and define
$$\mathcal{P}_6 := \big\{\{1,4\},\{2,6\},\{3,5,8\},\{7,9\}\big\}$$
7. Since $\langle 1,9\rangle \in R$, we replace the sets $\{1,4\}$ and $\{7,9\}$ by their union and define
$$\mathcal{P}_7 := \big\{\{1,4,7,9\},\{2,6\},\{3,5,8\}\big\}$$
8. Next, we have $\langle 4,7\rangle \in R$. However, 4 and 7 are already in the same set. Therefore we do not have to change the partition $\mathcal{P}_7$ in this step. Furthermore, we have now processed all the pairs in the given relation R. Therefore, $\mathcal{P}_7$ is the partition that represents the equivalence relation $\approx$ induced by R. According to this partition, we have found that
$$1 \approx 4 \approx 7 \approx 9, \quad 2 \approx 6, \quad \text{and} \quad 3 \approx 5 \approx 8.$$

```
1   unionFind := procedure(m, r) {
2       p := { { x } : x in m };  // start with the trivial partition
3       // refine this partition to accommodate all pair [x,y] in r
4       for ([x, y] in r) {
5           sx := find(x, p);
6           sy := find(y, p);
7           if (sx != sy) {
8               p  -= { sx, sy };  // remove old sets
9               p  += { sx + sy }; // add their union
10          }
11      }
12      return p;
13  };
14  find := procedure(x, p) {
15      return arb({ s : s in p | x in s });
16  };
```

Figure 8.1: A naive implementation of the union-find algorithm.

What we have sketched in the previous example is known as the *union-find algorithm*. Figure 8.1 shows a naive implementation of this algorithm. The procedure `unionFind` takes two arguments: `m` is a set and `r` is a relation on `m`. The purpose of `unionFind` is to compute the equivalence relation induced by `r`. This equivalence relation is represented as a partition of `m`.

1. In line 2 we initialize `p` as the trivial partition that contains only singleton sets. Obviously, this is a partition of `m` but it does not take the relation `r` into account.

2. The `for`-loop in line 4 iterates over all pairs `[x,y]` from `r`. First, we compute the set `sx` that contains `x` and the set `sy` that contains `y`. If these sets are not the same, then `x` and `y` are not yet equivalent with respect to the partition `p`. Therefore, the equivalence classes `sx` and `sy` are joined and their union is added to the partition in line 9, while the equivalence classes `sx` and `sy` are removed in line 8.

3. The function `find` takes an element `x` of a set `m` and a partition `p` of `m`. Since `p` is a partition of `m` there must be exactly one set `s` in `p` such that `x` is an element of `s`. This set `s` is then returned.

8.1.1 A Tree-Based Implementation

The implementation shown in Figure 8.1 is not very efficient. The problem is the computation of

```
sx + sy.
```

If the sets `sx` and `sy` are represented as binary trees, then this computation takes time proportional to $\texttt{min}(\#\texttt{sx}, \#\texttt{sy})$. Here $\#\texttt{sx}$ denotes the size of `sx` and similarly $\#\texttt{sy}$ denotes the size of `sy`. A more efficient way to represent these sets is via *parent pointers*: The idea is that every set is represented as a tree. However, this tree is not a binary tree but is rather represented by pointers that point from the a node to its parent. The node at the root of the tree points to itself. Then, taking the union of two sets `sx` and `sy` is simple: If `rx` is the node at the root of the tree representing `sx` and `ry` is the node at the root of the tree representing `sy`, then we can just change the parent pointer of `ry` to point to `rx`.

```
1   find := procedure(x, parent) {
2       if (parent[x] == x) {
3           return x;
4       }
5       return find(parent[x], parent);
6   };
7   unionFind := procedure(m, r) {
8       parent := { [x, x] : x in m };
9       for ([x, y] in r) {
10          rootX := find(x, parent);
11          rootY := find(y, parent);
12          if (rootX != rootY) {
13              parent[rootY] := rootX;  // create union
14          }
15      }
16      roots := { x : x in m | parent[x] == x };
17      return { { y : y in m | find(y, parent) == r } : r in roots };
18  };
```

Figure 8.2: A tree-based implementation of the union-find algorithm.

Figure 8.2 on page 135 shows an implementation of this idea. In this implementation, the parent pointers are represented using the binary relation `parent`.

1. The function `find` takes a node `x` and the binary relation `parent` representing the parent pointers. The purpose of the call `find(x, parent)` is to return the root of the tree containing `x`.

   If `x` is its own parent, then `x` is already at the root of a tree and therefore we can return `x` itself in line 3.

Otherwise, we compute the parent of x and recursively compute the root of the tree containing this parent.

2. The function unionFind takes a set m and a relation r. It returns a partition of m that represents the equivalence relation generated by r on m.

   The binary relation[1] partition is initialized in line 8 so that every node points to itself. This corresponds to the fact that the sets in the initial partition are all singleton sets.

   Next, the function unionFind iterates over all pairs [x, y] from the binary relation r. In line 10 and 11 we compute the roots of the trees containing x and y. If these roots are identical, x and y are already equivalent. Otherwise, the parent pointer of the root of the tree containing y is changed so that it now points to the root of the tree containing x. Therefore, instead of iterating over all elements of the set containing y we just change a single pointer.

   Line 16 computes the set of all nodes that are at the root of some tree. Then, for every root r of a tree, line 17 computes the set of nodes corresponding to this tree.

8.1.2 Controlling the Growth of the Trees

As it stands, the algorithm shown in the previous section has a complexity that is $\mathcal{O}(n^2)$ in the worst case where n is the number of elements in the set m. The worst case happens if there is just one equivalence class and the tree representing this class degenerates into a list. Fortunately, it is easy to fix this problem. We just have to keep track of the height of the different trees. Then, if we want to join the trees rooted at parentX and parentY, we have a choice: We can either set the parent of the node parentX to be parentY or we can set the parent of the node parentY to be parentX. If the tree rooted at parentX is smaller than the tree rooted at parentY we should use the assignment

```
parent[parentX] := parentY;
```

otherwise we should use

```
parent[parentY] := parentX;
```

In order to be able to distinguish these case, we store the height of the tree rooted at node n in the relation height, i.e. if n is a node, then height[n] is the height of the tree rooted at node n. This yields the implementation shown in Figure 8.3 on page 137. Provided the size of the relation r is bounded by the size n of the set m, the complexity of this implementation is $\mathcal{O}\big(n \cdot \log(n)\big)$.

Exercise 21: We can speed up the implementation previously shown if the set m has the form

$$\mathtt{m} = \{1, 2, 3, \cdots, n\} \quad \text{where } n \in \mathbb{N}.$$

In this case, the relations parent and height can be implemented as arrays. Develop an implementation that is based on this idea. ◇

8.1.3 Packing Union-Find as a Data Structure

When discussing the minimum spanning tree problem, we will need the union-find algorithm as an auxiliary data structure. To this end we present a class that encapsulates the union-find algorithm. This class is shown in Figure 8.4 on page 138.

1. The constructor unionFind receives a set m as arguments. The class unionFind maintains two variables:

   (a) mParent is the dictionary implementing the pointers that point to the parents of each node. If a node n has no parent, then we have

   ```
   mParent[n] = n,
   ```

[1] In a language like C we would instead use pointers. Of course, this would be more efficient.

```
unionFind := procedure(m, r) {
    parent := { [x, x] : x in m };
    height := { [x, 1] : x in m };
    for ([x, y] in r) {
        rootX := find(x, parent);
        rootY := find(y, parent);
        if (rootX != rootY) {
            if (height[rootX] < height[rootY]) {
                parent[rootX] := rootY;
            } else if (height[rootX] > height[rootY]) {
                parent[rootY] := rootX;
            } else {
                parent[rootY] := rootX;
                height[rootX] += 1;
            }
        }
    }
    roots := { x : x in m | parent[x] == x };
    return { { y : y in m | find(y, parent) == r } : r in roots };
};
```

Figure 8.3: A more efficient version of the union-find algorithm.

i.e. the roots of the trees point to themselves. Initially, all nodes are roots, so all parent pointers point to themselves.

(b) `mHeight` is a dictionary containing the heights of the trees. If `n` is a node, then

```
mHeight[n]
```

gives the height of the subtree rooted at `n`. As initially all trees contain but a single node, these trees all have height 1.

2. The method `union` takes two nodes `x` and `y` and joins the trees that contain these nodes. This is achieved by finding their parents `parentX` and `parentY`. Then, the root of the smaller of the two trees is redirected to point to the root of the bigger tree.

3. The method `find` takes a node `x` as its argument and computes the root of the tree containing `x`.

4. The method `partition` is a client of the class `unionFind`. It takes a set `m` and a relation `r` on `m` and computes a partition that corresponds to the equivalence relation induced by `r` on `m`.

   (a) First, the method iterates over all pairs `[x,y]` in the relation `r` and joins the equivalence classes corresponding to `x` and `y`.

   (b) Next, the method collects all nodes `x` that are at the root of a tree.

   (c) Finally, for every root `r` the method collects those nodes `x` that are part of the tree rooted at `r`.

```
class unionFind(m) {
    mParent := { [x, x] : x in m };
    mHeight := { [x, 1] : x in m };

  static {
    union := procedure(x, y) {
        rootX := find(x);
        rootY := find(y);
        if (rootX != rootY) {
            if (mHeight[rootX] < mHeight[rootY]) {
                this.mParent[rootX] := rootY;
            } else if (mHeight[rootX] > mHeight[rootY]) {
                this.mParent[rootY] := rootX;
            } else {
                this.mParent[rootY] := rootX;
                this.mHeight[rootX] += 1;
            }
        }
    };
    find := procedure(x) {
        p := mParent[x];
        if (p == x) {
            return x;
        }
        return find(p);
    };
  }
}
```

Figure 8.4: The class `unionFind`.

8.2 Minimum Spanning Trees

Imagine a telecommunication company that intends to supply internet access to a developing country. The capital of the country is located at the coast line and is already connected to the internet via a submarine cable. It is the company's task to connect all of the towns and villages to the capital. Since most parts of the country are covered by jungle, it is cheapest to build the power lines alongside existing roads. Mathematically, this kind of problem can be formulated as the problem of constructing a minimum spanning tree for a given weighted undirected graph. Next, we provide the definitions of those notions that are needed to formulate the minimum spanning tree problem precisely. Then, we present Kruskal's algorithm for solving the minimum spanning tree problem.

8.2.1 Basic Definitions

Definition 16 (Weighted Graph) An *weighted graph* is a triple $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ such that

1. $\mathbb{V}$ is the set is a set of *nodes*.
2. $\mathbb{E}$ is the set of *edges*. An edge e has the form

   $$\{x, y\}$$

   and connects x and y. Since $\{x, y\}$ is a set, we have

$$\{x, y\} \in \mathbb{E} \quad \text{if and only if } \{y, x\} \in \mathbb{E}.$$

Hence, if x is connected to y then y is also connected to x.

3. $\|\cdot\| : \mathbb{E} \to \mathbb{N}$ is a function assigning a *weight* to every edge. □

A *path* P is a list of the form

$$P = [x_1, x_2, x_3, \cdots, x_n]$$

such that we have :

$$\{x_i, x_{i+1}\} \in \mathbb{E} \quad \text{for all } i = 1, \cdots, n-1.$$

The path P *connects* the nodes x_1 and x_n. The *weight* of a path is defined as the sum of the weights of all of its edges.

$$\big\|[x_1, x_2, \cdots, x_n]\big\| := \sum_{i=1}^{n-1} \big\|\{x_i, x_{i+1}\}\big\|.$$

A graph is *connected* if for every $x, y \in \mathbb{V}$ there is a path connecting x and y. A set of edges can be interpreted as graph since the set of nodes can be computed from the edges as follows:

$$\mathbb{V} = \bigcup \big\{ \{x, y\} \,\big|\, \{x, y\} \in \mathbb{E} \big\}.$$

Then, a set of edges $\mathbb{E}$ is called a *tree* if and only if

1. all of its nodes are connected and
2. removing an edge from $\mathbb{E}$ would result in a graph that is no longer connected.

The *weight* of a tree is the sum of the weights of its edges.

Exercise 22: Assume that the graph $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ is a tree. Prove that then

$$\mathit{card}(\mathbb{E}) = \mathit{card}(\mathbb{V}) - 1$$

must hold.

Hint: The easiest way to prove this is by induction on $n = \mathit{card}(V)$. You will need to prove the following auxiliary claim first: In a tree there is at least one node in $\mathbb{V}$ that has only one neighbouring node. ◇

8.2.2 Kruskal's Algorithm

In 1956 the mathematician Joseph Bernard Kruskal (1928 – 2010) found a very elegant algorithm for solving the minimum spanning tree problem. This algorithm makes use of the union-find algorithm that we have developed previously. The basic idea is as follows.

1. In the first step, we create a union-find data structure that contains *singleton trees* for all nodes in the graph. Here, a singleton tree is a tree containing just a single node.
2. Next, we iterate over all edges $\langle x, y \rangle$ in the graph in increasing order of their weight. If the nodes x and y are not yet connected, we join their respective equivalence classes.
3. We stop when the number of edges is 1 less than the number of nodes since, according to the previous exercise, the tree must then connect all the nodes of the graph.

The fact that we iterate over the edges in increasing order of their weight guarantees that the resulting tree has a minimal weight. The algorithm is shown in Figure 8.5 on page 140. We discuss this program next.

```
1   mst := procedure(v, e) {
2       uf     := unionFind(v);
3       result := {};
4       for([w, [x, y]] in e) {
5           px := uf.find(x);
6           py := uf.find(y);
7           if (px != py) {
8               result += { [w, [x, y]] };
9               uf.union(px, py);
10              if (#result == #v - 1) {
11                  return result;
12              }
13          }
14      }
15  };
```

Figure 8.5: Kruskal's algorithm.

1. The main function is the function `mst`. This function is provided with two arguments: `v` is a set of nodes and `e` is a set of edges. The edges are represented as triples of the form

   `[w, [x, y]]`.

   Here, `x` and `y` are the nodes connected by the edge and `w` is the weight of the edge. The function `mst` computes a minimum spanning tree of the graph that is specified via the sets `v` and `e`. It is assumed that the graph defined by `v` and `e` is connected.

2. The function `mst` first creates the union-find data structure `uf` in line 2. Initially, in `uf` every node is in an equivalence class all by itself, i.e. nothing is yet connected.

3. The spanning tree constructed by the algorithm is stored in the variable `result`. The spanning tree is represented by a set of edges.

4. Next, we iterate over the edges in `e`. Since the first part of each edge is its weight, the fact that in SetlX sets are stored as ordered binary trees ensures that we start with the edge that has the smallest weight.

5. For every edge `[x,y]` we check whether `x` and `y` are already connected. This is the case if both `x` and `y` are in the same tree.

6. If `x` and `y` are not connected, the corresponding edge is added to the spanning tree and the trees containing `x` and `y` are connected.

7. The algorithm returns if the tree has `#v` $- 1$ edges, since we know that then all nodes are connected.

8.3 Shortest Paths Algorithms

In this section we will show two algorithms that solve the *shortest path problem*, the Bellman-Ford algorithm and Dijkstra's algorithm. In order to explain the shortest path problem and the algorithm solving it, we introduce the notion of a weighted *directed graph*.

Definition 17 (Weighted Digraph) A *weighted directed graph* (a.k.a. a *weighted digraph*) is a triple $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ such that

1. $\mathbb{V}$ is the set of *nodes* (sometimes the nodes are known as *vertices*).
2. $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is the set of *edges*.
3. $\|\cdot\| : \mathbb{E} \to \mathbb{N}$ is a function that assigns a positive *length* to every edge. This length is also known as the *weight* of the edge. ◇

Remark: The main difference between a graph and a digraph is that in a digraph the edges are pairs of two nodes while in a graph the edges are sets of two nodes. Informally, the edges in a digraph model one-way streets, while in a graph they model streets that work both ways. ◇

Definition 18 (Path, Path Length) A *path* P in a digraph is a list of the form

$$P = [x_1, x_2, x_3, \cdots, x_n]$$

such that

$$\langle x_i, x_{i+1} \rangle \in \mathbb{E} \quad \text{holds for all } i = 1, \cdots, n-1.$$

The set of all paths is denoted as $\mathbb{P}$. The length of a path is the sum of the length of all edges:

$$\big\|[x_1, x_2, \cdots, x_n]\big\| \;:=\; \sum_{i=1}^{n-1} \big\|\langle x_i, x_{i+1}\rangle\big\|.$$

If $p = [x_1, x_2, \cdots, x_n]$ is a path then p *connects* the node x_1 with the node x_n. We denote the set of all paths that connect the node v with the node w as

$$\mathbb{P}(v, w) \;:=\; \big\{[x_1, x_2, \cdots, x_n] \in \mathbb{P} \;\big|\; x_1 = v \wedge x_n = w\big\}.$$

Now we are ready to state the shortest path problem.

Definition 19 (Shortest Path Problem)
Assume a weighted digraph $G = \langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ and a node $\texttt{source} \in \mathbb{V}$ is given. Then the *shortest path problem* asks to compute the following function:

$$\begin{aligned} &\mathsf{sp} : \mathbb{V} \to \mathbb{N} \\ &\mathsf{sp}(v) \;:=\; \texttt{min}\big\{\|p\| \;\big|\; p \in \mathbb{P}(\texttt{source}, v)\big\}. \end{aligned}$$

Furthermore, given that $\mathsf{sp}(v) = n$, we would like to be able to compute a path $p \in \mathbb{P}(\texttt{source}, v)$ such that $\|p\| = n$. ◇

8.3.1 The Bellman-Ford Algorithm

The first algorithm we discuss is the *Bellman-Ford algorithm*. It is named after the mathematicians Richard E. Bellman [Bel58] and Lester R. Ford Jr. [For56] who found this algorithm independently and published their results in 1958 and 1956, respectively. Figure 8.6 on page 142 shows the implementation of a variant of this algorithm in SetlX. This variant of the algorithm was suggested by Edward F. Moore [Moo59].

```
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 8.6: The Bellman-Ford algorithm to solve the shortest path problem.

1. The function `shortestPath`(*source*, *edges*) is called with two arguments:
   (a) `source` is the start node. Our intention is to compute the distance of all vertices from `source`.
   (b) `edges` is a binary relation that encodes the set of edges of the graph. For every node x we have that `edges`$[x]$ is a set of the form
   $$\{[y_1, l_1], \cdots, [y_n, l_n]\}.$$
   This set is interpreted as follows: For every $i = 1, \cdots, n$ there is an edge $\langle x, y_i \rangle$ pointing from x to y_i and this edge has the length l_i.
2. The variable `dist` is a functional binary relation. When the computation is successful, for every node x the relation will contain a pair of the form $[x, \mathtt{sp}(x)]$ showing that the node x has distance $\mathtt{sp}(x)$ from the node `source`.
   The node `source` has distance 0 from the node `source` and initially this is all we know. Hence, the relation `dist` is initialized as the set `{[source,0]}`.
3. The variable `fringe` is a set of those nodes where we already have an estimate of their distance. Our goal is to compute the distances of all those nodes y that are connected to a node in `fringe` by a singe edge $\langle x, y \rangle$. Since, initially we only know the distance of the node `source`, we initialize the set fringe as the set
   $$\{\mathtt{source}\}.$$
4. As long as there are nodes left in the set `fringe` we pick an arbitrary node `u` from `fringe` and remove it from `fringe`.
5. Next, we compute the set of all nodes`v` that can be reached from`u` and check, whether we have found a better path leading to these nodes`v`. There are two cases.
   (a) If there is an edge from `u` to a node `v` and `dist[v]` is still undefined, then we hadn't yet found a path leading to `v`.
   (b) Furthermore, there are those nodes `v` where we had already found a path leading from `source` to `v` but the length of this path is longer than the length of the path that we get when we first visit `u` and then proceed to `v` via the edge $\langle \mathtt{u}, \mathtt{v} \rangle$.
   We compute the distance of the path leading from `source` to `u` and then to `v` in both of these cases and add `v` to the fringe.
6. The algorithm terminates when the set `fringe` is empty because in that case we don't have any means left to improve our estimation of the distance function.

8.3.2 Dijkstra's Algorithm

```
1   shortestPath := procedure(source, edges) {
2       dist    := { [source, 0] };
3       fringe  := { [0, source] };
4       visited := { source };
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 8.7: Dijkstra's algorithm to solve the shortest path problem.

The Bellman-Ford algorithm doesn't specify how the nodes are picked from the set `fringe`. In 1959 Edsger W. Dijkstra (1930 – 2002) [Dij59] published an algorithm that removed this non-determinism. Dijkstra had the idea to always choose the node that has the smallest distance to the node `source`. To do this, we just have to implement the set `fringe` as a priority queue where the priority of a node is given by the distance of this node to the node `source`. Figure 8.7 on page 143 shows an implementation of Dijkstra's algorithm in SetlX.

The program shown in Figure 8.7 has an additional variable called `visited`. This variable contains the set of those nodes that have been *visited* by the algorithm. To be more precise, `visited` contains those nodes `u` that have been removed from the priority queue `fringe` and for which all neighbouring nodes, i.e. those nodes y such that there is an edge $\langle u, y\rangle$, have been examined. The set `visited` isn't used in the implementation of the algorithm since the variable is only written but is never read. I have introduced the variable `visited` in order to be able to formulate the invariant that is needed to prove the correctness of the algorithm. This invariant is

$$\forall \texttt{u} \in \texttt{visited} : \texttt{dist[u]} = \texttt{sp(u)},$$

i.e. for all nodes $\texttt{u} \in \texttt{visited}$, the functional relation `dist` already contains the length of the shortest path leading from `source` to x.

Proof: We prove by induction that every time that a node `u` is added to the set `visited` we have that

$$\texttt{dist[u]} = \texttt{sp(u)}$$

holds. To begin, observe that the program has only two lines where the set `visited` is changed. We only need to inspect these lines.

1. **Base Case:** Initially, the set `visited` contains only the node `source` and we have

   $$\texttt{sp(source)} = 0 = \texttt{dist[source]}.$$

   Hence the claim is true initially.

2. **Induction Step:** In line 13 we add the node `u` to the set `visited`. Immediately before `u` is inserted there are two cases: If `u` is already a member of the set `visited`, the claim is true by

induction hypothesis. Hence we only need to consider the case where u is not a member of the set `visited` before it is inserted.

The proof now proceeds indirect and we assume that we have

$$\texttt{dist[u]} > \texttt{sp(u)}$$

Then there must exist a shortest path

$$p = [x_0 = \texttt{source}, x_1, \cdots, x_n = \texttt{u}]$$

leading from `source` to u that has length `sp(u)` and this length is less than `dist[u]`. Define $i \in \{0, \cdots, n-1\}$ to be the index such that

$$x_0 \in \texttt{visited}, \cdots, x_i \in \texttt{visited} \quad \text{but} \quad x_{i+1} \notin \texttt{Visited}$$

holds. Hence x_i is the first node in the path p such that x_{i+1} is not a member of `visited`. Such an index i has to exist because $\texttt{u} \notin \texttt{visited}$. Before the node x_i is added to the set `visited`, all nodes y that are connected to x_i via an edge $\langle x_i, y \rangle$ are examined and for these nodes the function `dist` is updated if this edge leads to a shorter path. In particular, x_{i+1} has been examined and $\texttt{dist}[x_{i+1}]$ has been recomputed. At the latest, the node x_{i+1} has been added to the set `fringe` at this time, although, of course, it could have been added already earlier. Furthermore, we must have

$$\texttt{dist}[x_{i+1}] = \texttt{sp}(x_{i+1}),$$

because by induction hypothesis we have that $\texttt{dist}[x_i] = \texttt{sp}(x_i)$ and the edge $\langle x_i, x_{i+1} \rangle$ is part of a shortest path from x_i to x_{i+1}.

As we have assumed that $x_{i+1} \notin \texttt{visited}$, the node x_{i+1} still has to be an element of the priority queue `fringe` at this point. Therefore we must have $\texttt{dist}[x_{i+1}] \geq \texttt{dist}[u]$, since otherwise x_{i+1} would have been chosen from the priority queue `fringe` before u has been chosen, but then x_{i+1} would be a member of `visited`.

Since $\texttt{sp}(x_{i+1}) = \texttt{dist}[x_{i+1}]$ we now have the contradiction

$$\texttt{sp(u)} \geq \texttt{sp}(x_{i+1}) = \texttt{dist}[x_{i+1}] \geq \texttt{dist[u]} > \texttt{sp(u)}.$$

This shows that the assumption $\texttt{dist[u]} > \texttt{sp(u)}$ has to be wrong. On the other hand, we always have $\texttt{dist[u]} \geq \texttt{sp(u)}$ we can only conclude that

$$\texttt{dist[u]} = \texttt{sp(u)}$$

holds for all nodes $\texttt{u} \in \texttt{visited}$. □

Exercise 23: Improve the implementation of Dijkstra's algorithm given above so that the algorithm also computes the shortest path for every node that is reachable from `source`. ◇

8.3.3 Complexity

If a node u is removed from the priority queue `fringe`, the node is added to the set `visited`. The invariant that was just proven implies that in that case

$$\texttt{sp(u)} = \texttt{dist[u]}$$

holds. This implies that the node u can never be reinserted into the priority queue `fringe`, because a node v is only inserted in `fringe` if either `dist[v]` is still undefined or if the value `dist[v]` decreases. Inserting a node into a priority queue containing n elements can be bounded by $\mathcal{O}\big(\log_2(n)\big)$. As the priority queue never contains more than $\#\mathbb{V}$ nodes and we can insert every node at most once, insertion into the priority queue can be bounded by

$$\mathcal{O}\big(\#V \cdot \log_2(\#V)\big).$$

We also have to analyse the complexity of removing a node from the fringe. The number of times the assignment

```
fringe -= { [dvOld, v] };
```

is executed is bounded by the number of edges leading to the node v. Removing an element from a set containing n elements can be bounded by $\mathcal{O}\big(\log_2(n)\big)$. Hence, removal of all nodes from the fringe can be bounded by

$$\mathcal{O}\big(\#\mathbb{E} \cdot \log_2(\#\mathbb{V})\big).$$

Here, $\#\mathbb{E}$ is the number of edges. Hence the complexity of Dijkstra's algorithm can be bounded by the expression

$$\mathcal{O}\big((\#\mathbb{E} + \#\mathbb{V}) * \ln(\#\mathbb{V})\big).$$

If the number of edges leading to a given node is bounded by a fixed number, e.g. if there are at most 4 edges leading to a given node, then the number of edges is a fixed multiple of the number of nodes. In this case, the complexity of Dijkstra's algorithm is given by the expression

$$\mathcal{O}\big(\#\mathbb{V} * \log_2(\#\mathbb{V})\big)\ .$$

Chapter 9

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.

9.1 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

$$Q = \{\langle x, y\rangle \in \mathbb{R} \mid -1 \leq x \leq 1 \ \wedge \ -1 \leq y \leq 1\}$$

definiert ist, zufällig eine große Zahl n von Sandkörnern werfen. Wir notieren uns dabei die Zahl k der Sandkörner, die in den Einheits-Kreis fallen. Die Wahrscheinlichkeit p dafür, dass ein Sandkorn in den Einheits-Kreis fällt, wird nun proportional zur Fläche des Einheits-Kreises sein:

$$p = \frac{\mathit{Fläche}(E)}{\mathit{Fläche}(Q)}.$$

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. Anstatt mit Sand können wir dieses Experiment 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 9.1: Experimentelle Bestimmung von π mit Hilfe der Monte-Carlo-Methode.

Abbildung 9.1 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 9.1: Ergebnisse bei der Bestimmung von π mit der Monte-Carlo-Methode

Lassen wir das Programm laufen, so erhalten wir die in Tabelle 9.1 gezeigten Ergebnisse. Wir sehen, dass wir zur Berechnung von π mit einer Genauigkeit von $10^{-}3$ etwa $10\,000\,000$ Versuche brauchen, was angesichts der Rechenleistung heutiger Computer kein Problem darstellt. Die Berechnung weiterer Stellen gestaltet sich jedoch sehr aufwendig: Eine Genauigkeit von 10^{-4} erfordert schon $1\,000\,000\,000$ 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 grobe Abschätzungen zu berechnen, wird aber sehr aufwendig, wenn eine hohe Genauigkeit gefordert ist.

9.2 Theoretischer Hintergrund*

Wir diskutieren nun den theoretischen Hintergrund der Monte-Carlo-Methode. Da im zweiten Semester noch keine detaillierteren Kenntnisse aus der Wahrscheinlichkeitsrechnung vorhanden sind, beschränken wir uns darauf, die wesentlichen Ergebnisse anzugeben. Eine Begründung dieser Ergebnisse erfolgt dann in der Statistik-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 etwas mehr oder etwas 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 letzten Abschnitt durchgeführte Berechnung der Zahl π an, so erhalten wir wegen $p = \frac{\pi}{4}$ die in Abbildung 9.2 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 9.2: Genauigkeit der Bestimmung von π bei einer Sicherheit von $99,73\%$.

Exercise 24: Wie viele Versuche sind notwendig um π mit der Monte-Carlo-Methode auf 6 Stellen hinter dem Komma zu berechnen, wenn das Ergebnis mit einer Wahrscheinlichkeit von $99,73\%$ korrekt sein soll?

Hinweis: Um eine Genauigkeit von 6 Stellen hinter dem Komma zu erreichen, sollte der Fehler durch 10^{-7} abgeschätzt werden. ◇

Solution: Nach dem Hinweis soll

$$\varepsilon(n) = 10^{-7}$$

gelten. Setzen wir hier die Formel für $\varepsilon(n)$ ein, so erhalten wir

$$\begin{array}{rll} & 3 \cdot \sqrt{\dfrac{p \cdot (1-p)}{n}} & = \quad 10^{-7} \\ \Leftrightarrow & 9 \cdot \dfrac{p \cdot (1-p)}{n} & = \quad 10^{-14} \\ \Leftrightarrow & 9 \cdot p \cdot (1-p) \cdot 10^{14} & = \quad n \end{array}$$

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 Näherung von π den Wert 3, so dass p den Wert $\frac{3}{4}$ hat. Damit ergibt sich für n der Wert

$$n = 168.75 \cdot 10^{12}.$$

Das sind also mehr als fast 169 Billionen Versuche. □

Exercise 25: 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. ◇

9.3 The Monty Hall Problem

The Monty Hall problem is famous probability puzzle that is based on the TV show Let's Make a Deal, which was aired in the US from the sixties through the seventies. The host of this show was Monty Hall. In his show, a player had to choose one of three doors. Monty Hall had placed goats behind two of the doors but there was a shiny new car behind the third door. Of course, the player did not know the location of the door with the car. Once the player had told Monty Hall the door he had chosen, Monty Hall would open one of the other two doors. However, Monty Hall would never open the door with the car behind it. Therefore, if the player had chosen the door with the car, Monty Hall would have randomly chosen a door leading to a goat. If, instead, the player had chosen a door leading to a goat, Monty Hall would have opened the door showing the other goat. In either case, after opening the door Monty Hall would ask the player whether he wanted to stick with his first choice or whether, instead, he wanted to pick the remaining closed door.

The question now is whether it is a good strategy to stick with the door chosen first or whether it is better to switch doors. Mathematically, the reasoning is quite simple: The probability that the door chosen first leads to the car is $\frac{1}{3}$. Therefore, the probability that the car is behind the other unopened door has to be $\frac{2}{3}$, as the two probabilities have to add up to 1.

Although the reasoning given above is straightforward, many people don't believe it. In order to convince them, the best thing is to run a Monte Carlo simulation. Figure 9.2 on page 151 shows a function that simulates n games and compares the different strategies.

1. The first strategy is the strategy that does not switch doors. For obvious reasons, this strategy is called the *stupid strategy*.
2. The second strategy will always switch to the other door. This strategy is called the *smart strategy*.

We discuss the implementation of the function `calculateChances` line by line.

1. In order to compare the two strategies, the idea is to play the game offered by Monty Hall n times. Then we need to count how many cars are won by the stupid strategy and how many cars are won by the smart strategy.

2. The variable `successStupid` counts the number of cars won by the stupid strategy.
3. The variable `successSmart` counts the number of cars won by the smart strategy.
4. The `for` loop extending from line 4 to line 15 runs `n` simulations of the game.
   (a) First, in line 5 the car is placed randomly behind one of the three doors.
   (b) Second, in line 6 the player picks a door.
   (c) In line 7, Monty Hall opens a door that does not have a car behind it and that is different from the door chosen by the player.
   (d) When the player uses the smart strategy, she will then pick the remaining door in line 8.
5. Next, we check which of the two strategies actually wins the car.
   (a) If the car was placed behind the door originally chosen by the player, the stupid strategy wins the car. Therefore, we increment the variable `successStupid` in this case.
   (b) If, instead, the car was placed behind the door that was neither chosen nor opened, then the smart strategy wins the car. Hence, the variable `successSmart` has to be incremented.
6. The function concludes by printing the results. Running the function for `n` equal to $100\,000$ has yielded the following result:

   ```
   The stupid strategy wins 33262 cars.
   The smart  strategy wins 66738 cars.
   ```

   This shows that, on average, the payoff from the smart strategy is about twice as high as the payoff from the stupid strategy. This is just what we expect since $\frac{2}{3} = 2 \cdot \frac{1}{3}$.

```
1  calculateChances := procedure(n) {
2      successStupid := 0;
3      successSmart  := 0;
4      for (i in [1..n]) {
5          car    := rnd({1..3});
6          choice := rnd({1..3});
7          opened := rnd({1..3} - { choice, car });
8          last   := arb({1..3} - { choice, opened });
9          if (car == choice) {
10             successStupid += 1;
11         }
12         if (car == last) {
13             successSmart += 1;
14         }
15     }
16     print("The stupid strategy wins $successStupid$ cars.");
17     print("The smart  strategy wins $successSmart $ cars.");
18 };
```

Figure 9.2: A program to solve the Monty Hall problem.

9.4 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 *permute*(L) die Liste unverändert zurück:

   $$\#L = 1 \rightarrow \textit{permute}(L) = L$$

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

   $$\textit{random} : \mathbb{N} \rightarrow \mathbb{N},$$

   so dass der Aufruf *random*(n) zufällig eine Zahl aus der Menge $\{1, \cdots, n\}$ liefert, so können wir diese Überlegung wie folgt formalisieren:

$$\#L = n \wedge n > 1 \wedge k := \textit{random}(n) \rightarrow \textit{permute}(L) = \textit{permute}\big(\textit{delete}(L, k)\big) + \big[L(k)\big].$$

   Der Funktionsaufruf *delete*(L, k) löscht dabei das k-te Element aus der Liste L, wir könnten also schreiben

   $$\textit{delete}(L, k) = L(1 \,..\, k-1) + L(k+1 \,..\, \#L).$$

```
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 9.3: Berechnung zufälliger Permutationen eines Feldes

Abbildung 9.3 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 derselben 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.

[Bel58] Richard E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[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.

[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.

[For56] Lester R. Ford. Network flow theory. Technical report, RAND Corporation, Santa Monica, California, 1956.

[GS78] Leonidas L. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on the Foundations of Computer Science*, pages 8–21. IEEE, 1978.

[GS13] Hans-Peter Gumm and Manfred Sommer. *Einführung in die Informatik*. Oldenbourg-Verlag, 10th edition, 2013.

[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.

[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.

[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.

[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.