Algorithms

— Lecture Notes for the Summer Term 2019 —

Prof. Dr. Karl Stroetmann

April 1, 2019

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`.

I am currently rewriting all SetlX programs contained in these lecture notes to *Python*. Hence, these lecture notes are subject to frequent changes. Provided the program `git` is installed on your computer, the repository containing the lecture notes and all of the accompanying programs can be cloned using the command

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

Once you have cloned the repository, the command

```
git pull
```

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

Contents

Chapter 1

Introduction

1.1 Motivation

The previous course has shown us how interesting problems can be solved with the help of sets and dictionaries. However, we did not discuss how these data structures can be implemented in an efficient way. This course will answer this question: We will develop a number of different data structures that can be used to implement both sets and dictionaries. 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 practical. 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. On Quora, the answers to the question

> "What are the 5 most important CS courses that every computer science student must take?"

consistently list the class algorithms and data structures among those courses that are most valuable for a professional career. 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 computing the value $f(x)$ for a given argument x. It is also important that the computation of $f(x)$ does not consume too much time or memory. Hence, we have to develop efficient algorithms. In order to be able to discuss the concept of efficiency we discuss the growth rate of functions. This notation is useful to abstract from unimportant details when discussing the runtime of algorithms.

2. Recurrence Relations

   The notion of a recurrence relation is the discrete analogue of the notion of a differential equation. For example, the equation

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

   is a recurrence relation. Together with the initial values $a_0 = 0$ and $a_1 = 1$, this equation defines a sequence of natural numbers. Later, we will see that this sequence can also be computed by the formula

$$a_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Recurrence relations occur naturally when analysing the runtime of algorithms. We present the Master Theorem that estimates the growth of recursive functions.

3. 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.

4. Sorting algorithms

   Sorting algorithm are among those 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 these algorithms. In this lecture, we discuss the following sorting algorithms:

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

5. Dictionaries

   A dictionary is a data structures that can be used to implement a function on a finite domain. Most modern programming languages provide dictionaries as basic data structures. We discuss various data structures that can be used to implement dictionaries efficiently. These data structures can also be used to implement sets.

6. Priority queues

   Some graph theoretical algorithms use priority queues as one of their basic building blocks. Therefore, our discussion of graph theory is preceded by a chapter on priority queues.

7. Graph theory

   This chapter discusses the following algorithms:

   (a) Dijkstra's algorithm for computing the shortest path in a graph,
   (b) Kruskal's algorithm for finding the minimum spanning tree of a graph,
   (c) topological sorting, and
   (d) the union-find problem.

8. 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 random simulations 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 have acquired the following capabilities:

1. You should be able to read and understand scientific literature describing algorithms.
2. You should have acquired the skill to develop your own algorithms and to analyse their complexity.

   Of course, developing an algorithm 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 similar 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 these two concepts. 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 just the interesting aspects. The rest can then be filled in by a competent programmer. Therefore, a specification of an algorithm often abstracts from minor details.

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 a philosophical idea and a text that describes the idea. If you have an philosophical idea, you can write it down to make it concrete. As you can write down the idea in English, or French or, preferably, in ancient Greek, the textual descriptions of the idea might be quite different. This is the same with an algorithm: We can code it in C or Python. 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. Therefore, this is our method of choice.

   However, after having presented an algorithm in the language of mathematics, it is often very straightforward to implement this algorithm in the programming language *Python*.

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 efficient with respect to both computing time and memory.
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 reason for the last goal is economical: If it takes too long to code an algorithm, the cost of the implementation might well be unaffordable. Furthermore, even if the time budget to implement an algorithm is next to 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.

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

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 mistakes. If you find a problem, 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 or discord. If you think that others might have the same question, it is best if you ask your question via discord in the channel called `algorithms`. 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 notations are useful to analyse the running time of algorithms. In order to illustrate the application of these notations, 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 size.
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 two years 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	$\dfrac{2 \cdot n^2 + 7}{3 \cdot n^3 + 2 \cdot n^2 + 7}$
1	0.75000000000000
10	0.06454630495800
100	0.00662481908150
1000	0.00066622484855
10 000	6.6662224852 e -05

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

Let us denote the set of all positive real numbers[1] 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. Let us consider a function $g \in \mathbb{R}_+^{\mathbb{N}}$. Informally, we have the following:

$f \in \mathcal{O}(g)$ if and only if f does not grow faster than g

We proceed to explain the definition of $\mathcal{O}(g)$ in more detail.

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

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

[1] In the literature, the set of positive real numbers is usually denoted as $\mathbb{R}_{>0}$. I prefer the notation $\mathbb{R}_+$ because it is shorter.

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 below 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.

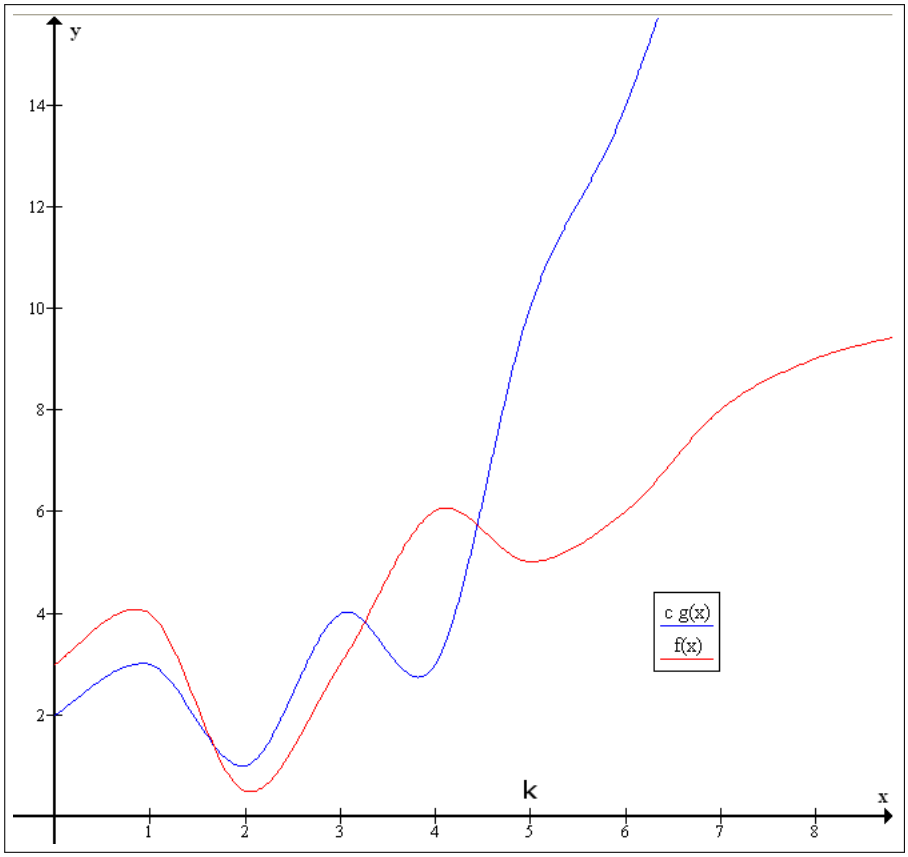

Figure 2.1: Example for $f \in \mathcal{O}(g)$.

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 $c \in \mathbb{R}_+$ and another constant $k \in \mathbb{N}$ 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) shows that

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

and therefore the proof is complete. □

Example: We have $n \in \mathcal{O}(2^n)$.
Proof: We have to provide a constant $c \in \mathbb{R}_+$ and a constant $k \in \mathbb{N}$ 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 $\beta \leq \alpha$ there exist numbers $c(\beta)$ such that

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

   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 $k \in \{0, 1, 2, \cdots, n\}$ as follows:

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

   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

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

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

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

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

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

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

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

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

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

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

holds. Adding these inequalities we conclude that

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

holds for all $n \geq k$. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The transitivity of the relation $\leq$ immediately implies $f(n) \leq c \cdot h(n)$ for $n \geq k$. □

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

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

holds. Combining the previous two inequalities yields

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

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[2], the limit can be computed as follows:

[2] 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. L'Hôpital's rule is discussed in the lectures on analysis.

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

The derivatives can be computed as follows:

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

We compute the second derivative and get

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

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

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

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

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

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

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 it is shown that

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

Therefore, we have

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

Exercise 4: Prove that $\sqrt{n} \in \mathcal{O}(n)$ holds. ◇

Exercise 5: Assume $\varepsilon \in \mathbb{R}$ and $\varepsilon > 0$. Prove that $\ln(n) \in \mathcal{O}\big(n^\varepsilon\big)$ 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 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 because if $b > 1$ and $c > 1$, then $\log_b(n)$ and $\log_c(n)$ only differ by a constant factor.

(b) Prove $\big(\log_2(n)\big)^2 \in \mathcal{O}(n)$.

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

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

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

(e) Assume that $f, g \in \mathbb{R}_+^{\mathbb{N}}$ and that, furthermore, $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0.$ holds. Prove that this implies

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

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

(g) Prove that we have that $n^k \in \mathcal{O}\big(n^{\ln(n)}\big)$ for all $k \in \mathbb{N}$. ◇

(h) Prove that $n^{\ln(n)} \in \mathcal{O}\big(b^n\big)$ for all $b > 1$. ◇

Remark: The last two examples show that the function $n^{\ln(n)}$ grows faster than any polynomial function but slower than any exponential function.

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 use this notation, too. 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

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

This is true because

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

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

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

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

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

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

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

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

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

```
def power(m, n):
    r = 1
    for i in range(n):
        r *= m
    return r
```

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 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:

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

In this formula $n\,//\,2$ denotes integer division by 2, e.g. we have $4\,//\,2 = 2$, but also $5\,//\,2 = 2$. Formally, for natural numbers $n \in \mathbb{N}$ and $k \in \mathbb{N}$ s.t. $k \geq 1$ the expression $n\,//\,k$ is defined as

$$n\,//\,k \;:=\; (n - n\,\%\,k)/k.$$

```
def power(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 $m, n \in \mathbb{N}$.

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 function definition. The method of computational induction consists of two steps:

1. The base case.

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

2. The induction step.

   In the induction step we have to prove that the function definition works in all those cases were the function 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

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

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

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

2. **Induction step**:

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

   $$\texttt{power}(m, n \mathbin{//} 2) = m^{n \mathbin{//} 2}$$

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

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

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

   (b) $n \mathbin{\%} 2 = 1$, therefore n is odd.
   Then there exists a number $k \in \mathbb{N}$ such that $n = 2 \cdot k + 1$ and we have $n \mathbin{//} 2 = k$. In this case we have:

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

   As we have shown that $\texttt{power}(m, n) = m^n$ 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

$$n \,//\, 2 = 2^{l-1} - 1 \quad \text{and therefore for } l > 1 \text{ we have} \quad n \,\%\, 2 = 1,$$

because in that case we have

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

Therefore, if $n = 2^l - 1$ the exponent n will be odd on every recursive call. 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 two multiplications in line 8 that have to be added to those multiplications that are performed in the recursive call in line 4. Therefore, we get the following recurrence relation:

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

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

$$b_l = a_{2^l-1} = a_{(2^l-1) \,//\, 2} + 2 = a_{2^{l-1}-1} + 2 = b_{l-1} + 2 \qquad \text{for all } l \in \mathbb{N} \text{ with } l > 0$$

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

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

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 8 yields two multiplications. Furthermore, in this case line 4 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 6 yields one multiplication that has to be added to those multiplications that are done during the recursive invocation of `power` in line 4. Therefore, we have the following recurrence relation:

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

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

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

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 integer 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

(a) $\alpha, \beta \in \mathbb{N}$ such that $\beta \geq 2$, $\delta \in \mathbb{R}$ such that $\delta \geq 0$ and

(b) the function $f : \mathbb{N} \to \mathbb{R}_+$ satisfies the recurrence relation

$$f(n) = \alpha \cdot f\left(n \,//\, \beta\right) + \mathcal{O}(n^\delta),$$

where $n \,//\, \beta$ denotes integer division[3] of n by β.

Then we have the following:

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

[3] For given integers $a, b \in \mathbb{N}$, the integer division $a \,//\, b$ is defined as the biggest number $q \in \mathbb{N}$ such that $q \cdot b \leq a$.

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

$$f(n) = \alpha \cdot f\left(n \mathbin{//} \beta\right) + n^\delta,$$

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:

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

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

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

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

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

If we take this formula and substitute $i := k$, 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:

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

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

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

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

   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

   $$a_k = \alpha^k \cdot a_0 + \gamma^k \cdot \frac{\big(\frac{\alpha}{\gamma}\big)^k - 1}{\frac{\alpha}{\gamma} - 1} = \alpha^k \cdot a_0 + \gamma \cdot \frac{\alpha^k - \gamma^k}{\alpha - \gamma} = \alpha^k \cdot a_0 + \mathcal{O}\big(\alpha^k\big),$$

   where we have used the fact that $\gamma < \alpha$ in the last step. 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 the following:

   $$\begin{array}{rrcl} & \alpha^{\log_\beta(n)} & = & n^{\log_\beta(\alpha)} \\ \Leftrightarrow & \log_\beta\big(\alpha^{\log_\beta(n)}\big) & = & \log_\beta\big(n^{\log_\beta(\alpha)}\big) \\ \Leftrightarrow & \log_\beta(n) \cdot \log_\beta(\alpha) & = & \log_\beta(\alpha) \cdot \log_\beta\big(n\big) \end{array}$$

   Using the first of these equations we conclude that

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

   holds. □

Examples:

1. Assume that f satisfies the recurrence relation

   $$f(n) = 9 \cdot f(n \mathbin{//} 3) + n.$$

   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

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

2. Assume that the function $f(n)$ satisfies the recurrence relation

$$f(n) = f(n \mathbin{//} 2) + 2.$$

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

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

Furthermore, we have

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

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

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

3. This time, f satisfies the recurrence relation

$$f(n) = 3 \cdot f(n \mathbin{//} 4) + n^2.$$

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

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

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

$$f(n) = 3 \cdot f(n \mathbin{//} 4) + n \cdot \log_2(n).$$

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 $\frac{1}{42}$ or 10^{-6}. 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

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

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

$$f(n) \in \mathcal{O}\big(n^{1+\frac{1}{42}}\big) \quad \text{or even} \quad f(n) \in \mathcal{O}\big(n^{1.000001}\big)$$

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. $f(n) = 4 \cdot f(n \mathbin{//} 2) + 2 \cdot n + 3$.
2. $f(n) = 4 \cdot f(n \mathbin{//} 2) + n^2$.
3. $f(n) = 3 \cdot f(n \mathbin{//} 2) + n^3$. ◇

Exercise 9: Consider the recurrence relation

$$f(n) = 2 \cdot f(n \mathbin{//} 2) + n \cdot \log_2(n).$$

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 $f(1) = 1$. Try to 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:

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

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

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

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

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

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

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

and this clearly shows that for big values of n, algorithm A is faster than algorithm B by a factor of 3.

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 pair $\langle P, \leq \rangle$ 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 much 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

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

If we extend the order $\preceq$ to the equivalence classes generated by the relation $\approx$, then it can be shown that this extension is a 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:
   $$\forall x \in s: \mathit{count}(x, S) \leq 1 \wedge \forall x \in M: \big(\mathit{count}(x, L) > 0 \leftrightarrow \mathit{count}(x, S) = 1\big).$$
   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   def sort(L):
2       if L == []:
3           return []
4       x, R = L[0], L[1:]
5       return insert(x, sort(R))
6
7   def insert(x, L):
8       if L == []:
9           return [x]
10      y, R = L[0], L[1:]
11      if x <= y:
12          return [x] + L
13      else:
14          return [y] + insert(x, R)
```

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` in line 11 of Figure 3.1 in the worst case if we call $\texttt{sort}(L)$ with a list L of length n. In order to do that, we have to compute the number of evaluations of the operator "<=" when $\texttt{insert}(x, L)$ 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 11 of Figure 3.1 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 $\texttt{insert}(x, L)$ 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 $\texttt{sort}(L)$ 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 $\texttt{insert}(x, \texttt{sort(R)})$ inserts the element x into $\texttt{sort}(R)$. We have previously seen that this takes n comparisons if x is bigger than all elements of $\texttt{sort}(R)$ 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

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

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

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

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 3.2 on page 33 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.

```
def sort(L):
    if L == []:
        return []
    x = min(L)
    return [x] + sort(delete(x, L))

def delete(x, L):
    if L == []:
        assert L != [], f'delete({x}, [])'
    if L[0] == x:
        return L[1:]
    return [L[0]] + delete(x, L[1:])
```

Figure 3.2: 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 $\texttt{min}$ is called $(n-1)$ times. Each of these calls of $\texttt{min}$ 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 $\texttt{sort}$.

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 is, in fact, more efficient than any other sorting algorithm.

3.3 Merge Sort

Next, we discuss merge sort. This algorithm is the first optimally efficient sorting algorithm that we encounter: We will see that merge sort only needs $\mathcal{O}(n \cdot \log_2(n))$ comparisons to sort a list of n elements. Later, we will prove that every algorithm that needs to compare its elements in order to sort them has at least this complexity. 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 of length $n := \#L$ the algorithm proceeds as follows:

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

   $$n < 2 \rightarrow \texttt{sort}(L) = L.$$

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:

   $$n \geq 2 \rightarrow \texttt{sort}(L) = \texttt{merge}\big(\texttt{sort}\big(\texttt{L[:n // 2]}\big), \texttt{sort}\big(\texttt{L[n // 2:]}\big)\big)$$

   Here, `L[:n // 2]` is the first part of the list, while `L[n // 2:]` is the second part. If the length of L is even, both part have the same number of elements, otherwise the second part has one element more than the first part. The function `merge` takes two sorted lists and combines their element in a way that the resulting list is sorted.

Figure 3.3 shows how these equations can be implemented as a SetlX program.

```
def sort(L):
    n = len(L)
    if n < 2:
        return L
    L1, L2 = L[:n//2], L[n//2:]
    return merge(sort(L1), sort(L2))

def merge(L1, L2):
    if L1 == []:
        return L2
    if L2 == []:
        return L1
    x1, R1 = L1[0], L1[1:]
    x2, R2 = L2[0], L2[1:]
    if x1 <= x2:
        return [x1] + merge(R1, L2)
    else:
        return [x2] + merge(L1, R2)
```

Figure 3.3: The merge sort algorithm implemented in SetlX.

1. If the list L has less than two elements, it is already sorted and, therefore, it can be returned as it is.

2. If the List L has n elements, then splitting L is achieved by putting the first n // 2 elements into the list L_1 and the remaining elements into the list L_2.

3. These lists are sorted recursively and the resulting sorted lists are then merged.

Finally, we need to 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:

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

   (b) $\neg x \preceq y$.
   Now we merge L_1 and R_2 and put y at the beginning of this list:

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

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. Define

$$f(n) \texttt{ := } \text{number of comparisons needed to sort a list } L \text{ of length } n.$$

The algorithm merge sort splits the list L into two lists that have the length of $n \texttt{ // } 2$ or $n \texttt{ // } 2 + 1$, then sorts these lists recursively, and finally merges the sorted lists. Merging the two lists can be done with at most n comparisons. Therefore, the function f satisfies the recurrence relation

$$f(n) = 2 \cdot f(n \texttt{ // } 2) + \mathcal{O}(n),$$

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,

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

and hence 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. 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. This is very wasteful. 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 3.4 on page 38 presents an implementation of merge sort that treats the list L that is to be sorted as an array.

We discuss the implementation shown in Figure 3.4 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. 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`.

```
1  def sort(L):
2      A = L[:]  # A is a copy of L
3      mergeSort(L, 0, len(L), A)
4
5  def mergeSort(L, start, end, A):
6      if end - start < 2:
7          return
8      middle = (start + end) // 2
9      mergeSort(L, start,  middle, A)
10     mergeSort(L, middle, end   , A)
11     merge(L, start, middle, end, A)
12
13 def merge(L, start, middle, end, A):
14     for i in range(start, end):
15         A[i] = L[i]
16     idx1 = start
17     idx2 = middle
18     i    = start
19     while idx1 < middle and idx2 < end:
20         if A[idx1] <= A[idx2]:
21             L[i]  = A[idx1]
22             idx1 += 1
23         else:
24             L[i]  = A[idx2]
25             idx2 += 1
26         i += 1
27     while idx1 < middle:
28         L[i]  = A[idx1]
29         idx1 += 1
30         i    += 1
31     while idx2 < end:
32         L[i]  = A[idx2]
33         idx2 += 1
34         i    += 1
```

Figure 3.4: An array based implementation of merge sort.

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(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 3.5: A non-recursive implementation of merge sort.

The implementation of merge sort shown in Figure 3.4 on page 38 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 3.5 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. These sublists have a length of n. 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*k+1 .. n*(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/jdk10/jdk10/jdk/file/ffa11326afd5/src/java.base/share/classes/java/util/ComparableTimSort.java

Timsort is also used on the Android platform.

3.4 Quicksort

In 1961, C.A.R. Hoare published the quicksort 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 $\texttt{L} = [\texttt{x}|\texttt{R}]$. In this case, we split R into two lists S and B. The list S (the letter S stands for small) contains all those elements of R that are less or equal than x, while B (the letter B stands for big) contains those elements of R that are bigger than x. These lists are computed as follows:

   (a) $\texttt{S} := [\texttt{y} \in \texttt{R} \mid \texttt{y} \leq \texttt{x}]$,

   (b) $\texttt{B} := [\texttt{y} \in \texttt{R} \mid \texttt{y} > \texttt{x}]$.

   The process of splitting the list R into the lists S and B is called partitioning. After partitioning the list R into the lists S and B, these lists are sorted recursively. Then, the result is computed by putting x between the lists $\texttt{sort}(\texttt{S})$ and $\texttt{sort}(\texttt{B})$:

$$\texttt{sort}([\texttt{x}|\texttt{R}]) = \texttt{sort}(\texttt{S}) + [\texttt{x}] + \texttt{sort}(\texttt{B}).$$

Figure 3.6 on page 41 shows how these equations can be implemented in SetlX.

```
def sort(L):
    if L == []:
        return L
    x, R = L[0], L[1:]
    S = [y for y in R if y <= x]
    B = [y for y in R if y >  x]
    return sort(S) + [x] + sort(B)
```

Figure 3.6: The quicksort algorithm.

3.4.1 Complexity

Next, we investigate the computational complexity of quicksort. Our goal is to compute the number of comparisons that are needed when $\texttt{sort}(\texttt{L})$ is computed for a list L of length n. In order to compute this number we first investigate how many comparisons are needed in order to partition the list R into the lists S and B. As these lists are defined as

$$\texttt{S} := [\texttt{y} \in \texttt{R} \mid \texttt{y} \leq \texttt{x}] \quad \text{and} \quad \texttt{B} := [\texttt{y} \in \texttt{R} \mid \texttt{y} > \texttt{x}],$$

it is obvious that each element of R has to be compared with x. Therefore, we need n comparisons to compute S and B. Since S and B are computed independently, the implementation given above would really need $2 \cdot n$ comparisons. However, a moments thought reveals that we can compute S and B with just n comparisons if the two lists are computed simultaneously. Next, we investigate the worst case complexity of quicksort.

Worst Case Complexity

Let us denote the number of comparisons needed to evaluate $\texttt{sort}(\texttt{L})$ for a list L of length n in the worst case as a_n. The worst case occurs if the partitioning returns a pair of lists S and B such that

$$\texttt{S} = [],$$

i.e. all elements of R are bigger than x. Then, we have

$$a_n = a_{n-1} + n - 1.$$

The term $n - 1$ is due to the $n - 1$ comparisons needed for the partitioning of R and the term a_{n-1} is the number of comparisons needed for the recursive evaluation of $\texttt{sort}(\texttt{B})$.

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:

$$\begin{aligned} a_n &= a_{n-1} + (n-1) \\ &= a_{n-2} + (n-2) + (n-1) \\ &= a_{n-3} + (n-3) + (n-2) + (n-1) \\ &= \vdots \\ &= a_1 + 1 + 2 + \cdots + (n-2) + (n-1) \\ &= 0 + 1 + 2 + \cdots + (n-2) + (n-1) \\ &= \sum_{i=0}^{n-1} i = \frac{1}{2} \cdot n \cdot (n-1) = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n \\ &\in \mathcal{O}(n^2) \end{aligned}$$

This shows that in the worst case, the number of comparisons is as big as it is in the worst case of insertion sort. However, with quicksort 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 quicksort since, in the worst case, it is much slower as merge sort. To understand what is really going on, we define

$$d_n \text{ := average number of comparisons to sort a list } L \text{ of } n \text{ elements via quicksort.}$$

We will show that $d_n \in \mathcal{O}\big(n \cdot \log_2(n)\big)$. 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 $\#\mathtt{S} = i$, 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 partition the list R. 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

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

In order to simplify this equation, let us define

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

Substituting this into equation (6) yields

$$a_{n+1} = a_n + \frac{2 \cdot n}{(n+1) \cdot (n+2)}. \tag{7}$$

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 similar 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

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

Then equation (7) 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

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

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 quicksort needs

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

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

3.4.2 Implementing Quicksort for Arrays

Next, we show how quicksort is implemented using arrays instead of lists. We are following the scheme of Nico Lomuto [CLRS09]. Figure 3.7 on page 45 shows this implementation.

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

Figure 3.7: An implementation of quicksort 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 quicksort over merge sort.

2. The function `sort` is reduced to a call of `quickSort`. This function takes the parameters `a`, `b`, and `L`.

   (a) `a` specifies the index of the first element of the subarray that needs to be sorted.

   (b) `b` specifies the index of the last element of the subarray that needs to be sorted.

   (c) `L` is the array that needs to be sorted.

   Calling $\texttt{quickSort}(\texttt{a}, \texttt{b}, \texttt{L})$ sorts the subarray

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

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

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

   The implementation of the function `quickSort` is quite similar to the list implementation. The main difference is that the function `partition`, that is called in line 8, redistributes the elements of `L`: All elements that are less or equal than the pivot element $\texttt{L}[\texttt{m}]$ are stored at indexes that are

smaller than the index `m`, while the remaining elements will be stored at indexes that are bigger than `m`. The pivot element itself will be stored at the index `m`.

3. The difficult part of the implementation of quicksort is the implementation of the function `partition` that is shown beginning in line 12. The `for` loop in line 15 satisfies the following invariants.

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

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

   (c) $\texttt{pivot} = \texttt{L}[\texttt{end}]$
   The pivot element itself is at the end of the array.

   Observe how the invariants (a) and (b) are maintained:

   (a) Initially, the invariants are true because the corresponding sets are empty. At the start of the `for`-loop we have
   $$\{\texttt{start}, \cdots, \texttt{left}\} = \{\texttt{start}, \cdots, \texttt{start} - 1\} = \{\}$$
   and
   $$\{\texttt{left} + 1, \cdots, \texttt{idx} - 1\} = \{\texttt{start}, \cdots, \texttt{start} - 1\} = \{\}.$$

   (b) If the element $\texttt{L}[\texttt{idx}]$ is less than the pivot element, it need to become part of the subarray $\texttt{L}[\texttt{start} \,..\, \texttt{left}]$. In order to achieve this, it is placed at the position $\texttt{L}[\texttt{left} + 1]$. The element that has been at that position is part of the subarray $\texttt{L}[\texttt{left} + 1 \,..\, \texttt{idx} - 1]$ and therefore, most of the times,[1] it is greater than the pivot element. Hence we append this element to the end of the subarray $\texttt{L}[\texttt{left} + 1 \,..\, \texttt{idx} - 1]$. After incrementing the index `left`, both the placing of the element $\texttt{L}[\texttt{idx}]$ at position $\texttt{left} + 1$ and the appending of the element $\texttt{L}[\texttt{left} + 1]$ to the end of the subarray $\texttt{L}[\texttt{left} + 1 \,..\, \texttt{idx} - 1]$ is achieved by the statement
   ```
   swap(left, idx, L.
   ```

   Once the `for` loop in line 15 terminates, the call to `swap` in line 21 moves the pivot element into its correct position and returns the index where the pivot element has been placed.

3.4.3 Improvements for Quicksort

There are a number of tricks that can be used to increase the efficiency of quicksort.

1. Instead of taking the first element as the pivot element, use three elements from the list `L` that is to be sorted. For example, take the first element, the last element, and an element from the middle of the list. Now compare these three elements and take that element as a pivot that is between the other two elements.

   The advantage of this strategy is that 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 quicksort 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.

[1] It is not always greater than the pivot element because the subarray $\texttt{L}[\texttt{left} + 1 \,..\, \texttt{idx} - 1]$ might well be empty.

For example, the expression

```
shuffle([1..10]);
```

might return the result

```
[1, 9, 8, 5, 2, 10, 6, 3, 4, 7].
```

4. In 2009, Vladimir Yaroslavskiy introduced dual pivot quicksort [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

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

   i.e. we take x as the first element of L, while y is the last element of L. Then, the list L is split into three parts:

   (a) The first part contains those elements that are less than x.

   (b) The second part contains those elements that are bigger or equal than x but less or equal than y.

   (c) The third part contains those elements that are bigger than y.

   Figure 3.8 on page 47 shows a simple list based implementation of dual pivot quicksort.

   Various studies have shown that, on average, dual pivot quicksort is faster than any other sorting algorithm. For this reason, the version 1.7 of *Java* uses dual pivot quicksort:

   http://www.docjar.com/html/api/java/util/DualPivotQuicksort.java.html

```
1  def sort(L):
2      match (L) {
3      case []     : return [];
4      case [x]    : return [x];
5      case [x,y|r]:
6           [p1, p2] := [min({x,y}), max({x,y})];
7           [L1,L2,L3] := partition(p1, p2, r);
8           return sort(L1) + [p1] + sort(L2) + [p2] + sort(L3);
9      }
10 };
11 partition := procedure(p1, p2, L) {
12     match (L) {
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 3.8: A list based implementation of dual pivot quicksort.

Exercise 12: Implement a version of dual pivot quicksort 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 not permitted to 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:

$$[a_1, a_2, a_3], \quad [a_1, a_3, a_2], \quad [a_2, a_1, a_3], \quad [a_2, a_3, a_1], \quad [a_3, a_1, a_2], \quad [a_3, a_2, a_1].$$

Therefore, 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.

B.C.: $n = 1$:

There is only 1 way to arrange one element in a list. As $1 = 1!$ the claim ist proven in this case.

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

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.

3.6 Counting Sort

In the last section of this chapter we introduce a sorting algorithm that has only a linear complexity. According to the result of the previous section this algorithm does not work by comparing the elements of the list that is to be sorted. Rather, this algorithm exploits the fact that the keys used for sorting are natural numbers. We explain counting sort with the help of an example. Table 3.1 on page 50 shows a table showing students and their grades. As it stands, the names of the students are ordered alphabetically. However, the teacher would like to sort the list of students according to their grades. Within a group of students that have achieved the same grade, the students should still be ordered alphabetically. Table 3.2 on page 51 shows the table that has been sorted accordingly.

We proceed to describe an algorithm that is capable of sorting the Table 3.1 into the table 3.2. The algorithm works in three stages.

1. The first stage is the counting stage. In this stage we count the number of students that have a specific grades. In the example from Table 3.1 we find the following:
   (a) 3 students have grade 1.
   (b) 8 students have grade 2.
   (c) 6 students have grade 3.
   (d) 3 students have grade 4.
   (e) 2 students have grade 5.
2. The second stage is the indexing stage. From the previous stage we know that the sorted list L that is returned must have the following outline.
   (a) The sublist `L[1..3]` contains the students with grade 1.
   (b) The sublist `L[4..11]` contains the students with grade 2.
   (c) The sublist `L[12..17]` contains the students with grade 3.
   (d) The sublist `L[18..20]` contains the students with grade 4.

Student	Grade
Alexander	4
Benjamin	2
Daniel	3
David	3
Elijah	2
Gabriel	1
Henry	2
Jacob	5
James	3
Joseph	2
Liam	2
Logan	3
Lucas	1
Mason	2
Matthew	5
Michael	3
Noah	4
Oliver	2
Owen	4
Samuel	3
Sebastian	2
William	1

Table 3.1: Students and their grades, sorted alphabetically.

(e) The sublist `L[21..22]` contains the students with grade 5.

The indexing stage computes the starting indices of these sublists. Of course, the first sublist has to start at the index 1. Since there are 3 students with a grade of 1, the second sublist starts at the index $1 + 3 = 4$. Since there are 8 students with a grade of 2, the third sublist starts at the index $1 + 3 + 8 = 12$. In general, if the sublist for the students with grade g starts at index i_g and there are n_g students that have achieved the grade g, then the sublist for the students with grade $g + 1$ starts at index i_{g+1} where

$$i_{g+1} = i_g + n_g.$$

3. The distribution stage iterates over the list of students and inserts them into the sublists corresponding to the grades of the students.

Figure 3.9 on page 52 shows an implementation of counting sort. We proceed to discuss this algorithm line by line.

1. The procedure `countingSort` receives two parameters:

   (a) `Names` is a list of the student names that is sorted alphabetically.

   (b) `Grades` is a list of the grades attached to the students.

   Of course, these lists are required to have the same number of elements. This is checked in the `assert` statement in line 2.

2. In order for our routine to generalize to arbitrary grades, we need to compute the maximum of all grades and store it the variable `maxGrade`. Of course, in the example discussed so far we know that the biggest grade is 5. However, in general the grades can be any numbers and the procedure `countingSort` is able to sort the `Names` according to these grades.

Student	Grade
Gabriel	1
Lucas	1
William	1
Benjamin	2
Elijah	2
Henry	2
Joseph	2
Liam	2
Mason	2
Oliver	2
Sebastian	2
Daniel	3
David	3
James	3
Logan	3
Michael	3
Samuel	3
Alexander	4
Noah	4
Owen	4
Jacob	5
Matthew	5

Table 3.2: Students and their grades, sorted with respect to the grade.

3. Next, we initialize the auxiliary array `Count` to be an array of length `maxGrade`. Later, for a grade g the number `Counts`$[g]$ will contain the number of students that have attained the grade g. Initially, all entries of the array `Count` are set to 0.

4. After the indexing stage, the array `Indices` will contain the start indices of the different sublists. For a grade g, `Indices`$[g]$ is the first index of the sublist containing those students that have achieved the grade g.

5. The array `SortedNames` and `SortedGrades` are the two lists that will be returned as the result. `SortedNames` will contain the names of the students sorted by their grades, while `SortedGrades` will contain the corresponding grades.

6. The `for`-loop in line 8 performs the counting stage. We iterate over all grades g in `Grades` and increment the counter `Counts`$[g]$ associated with the grade g.

7. Next, the index for the start of the sublist containing those students that have achieved the grade 1 is initialized as 1 in line 11.

8. Then, the `for`-loop in line 12 performs the indexing stage. As the number `Indices`$[g-1]$ is the index of the start of the sublist for those students that have grade $g-1$ and the number of these students is `Counts`$[g-1]$, the sublist of the students with grade g has to start at

   $$\texttt{Indices}[g-1] + \texttt{Counts}[g-1].$$

9. Finally, the `for`-loop in line 15 performs the distribution stage.

   (a) Conceptually, the `for`-loop iterates over all students. The index i refers to the i-th student in the original list.

```
1  def countingSort(Names, Grades):
2      assert(#Names == #Grades, "$#Names$ != $#Grades$");
3      maxGrade     := max(Grades);
4      Counts       := [0] * maxGrade;
5      Indices      := [];
6      SortedNames  := [];
7      SortedGrades := [];
8      for (g in Grades) {
9          Counts[g] += 1;
10     }
11     Indices[1] := 1;
12     for (g in [2 .. maxGrade]) {
13         Indices[g] := Indices[g-1] + Counts[g-1];
14     }
15     for (i in [1 .. #Names]) {
16         g     := Grades[i];
17         index := Indices[g];
18         SortedNames [index] := Names[i];
19         SortedGrades[index] := Grades[i];
20         Indices[g]          += 1;
21     }
22     return [SortedNames, SortedGrades];
23 };
```

Figure 3.9: An Implementation of counting sort.

(b) Next, we need to find where to put the i-th student. To this end we first look up the grade $g = \texttt{Grades}[i]$ of this student. Then, $\texttt{Indices}[g]$ gives us the `index` of the next free entry in the result list `SortedNames` corresponding to grade g.

(c) In line 17 and 18 the student and his grade are stored in the result lists `SortedNames` and `SortedGrades` at the computed `index`.

(d) Finally, we need to increment the index stored at $\texttt{Indices}[g]$ since we have just used this index and therefore the next student with the grade g needs to be stored at the subsequent location. This is done in line 20.

10. The procedure `countingSort` returns a pair of lists.

(a) The first list is the list of students sorted with respect to their grades.

(b) The second list is the list of the corresponding grades.

If the list `Names` has a length of n and the biggest grade is some fixed constant c, then it is easy to see that counting sort has the complexity $\mathcal{O}(n)$. The reason is that the first `for`-loop is iterated n times, the second `for`-loop is iterated just c times and the lust `for`-loop is again iterated n times. Hence, counting sort is a linear sorting algorithm. Note that we do not use any comparisons in this algorithm.

Another important fact is that counting sort is stable: In the resulting list, the sublists corresponding to the different grades are still sorted alphabetically. This is so because these sublists are filled by iterating over the original list that is sorted alphabetically. If two students x and y have the same grade g but the name of x is alphabetically before the name of y, then x will be inserted into the sublist corresponding grade g before y and hence these sublists are ordered alphabetically. This property is crucial for the development of our next sorting algorithm radix sort.

3.7 Radix Sort

The importance of the previous sorting algorithm, counting sort, stems from the fact that it is part of the implementation of radix sort. Radix sort was used as early as 1887 in tabulating machines constructed by Hermann Hollerith. He was the founder of the Tabulating Machine Company that later became IBM. To understand radix sort, suppose we want to implement an algorithm that sorts a large number of 32 bit unsigned integers. The easiest way to do this would be to use counting sort: In that case the names of the students would be unimportant and could be filled with an arbitrary string. However, since the grades could then have a maximum value of

$$2^{32} - 1 = 4,294,967,295,$$

the lists `Counts` and `Indices` that we have used in the function `countingSort` would become huge and would use 16 gigabyte. This size is prohibitive. The main idea of radix sort is to split the 32 bit numbers into four chunks of 8 bits each and to use each of these four chunks as a grade. To formulate this mathematically, a 32 bit unsigned integer x is decomposed into its four bytes as follows:

$$x = b_4 \cdot 256^3 + b_3 \cdot 256^2 + b_2 \cdot 256^1 + b_1.$$

Note that have numbered the bytes starting from the the least significant byte. Then, radix sort works as follows:

1. Sort the numbers by interpreting the byte b_1 as a grade.
2. Take the numbers that have been sorted by the byte b_1 and sort them by the byte b_2 next. Since counting sort is stable, numbers which happen to have the same byte b_2 will still be sorted with respect to by the byte b_1. Hence, after the sorting with respect to the byte b_2 is complete, in effect the numbers will then be sorted according to both b_2 and b_1.
3. Next, use counting sort to sort the numbers with respect to the byte b_3.
4. Finally, use counting sort to sort the numbers with respect to the byte b_4. By the stability of counting sort, the numbers are now sorted with respect to all of their byte, where b_4 is the most significant byte and b_1 is the least significant byte. Hence, the numbers are sorted.

```
1  load("counting-sort.stlx");
2
3  extractByte := [x, k] |-> (x \ 256 ** (k-1)) % 256;
4
5  radixSort := procedure(L) {
6      for (k in [1..4]) {
7          Grades := [extractByte(x, k) + 1: x in L];
8          L := countingSort(L, Grades)[1];
9      }
10     return L;
11 };
```

Figure 3.10: An implementation of radix sort for sorting unsigned 32 integers.

Figure 3.10 on page 53 shows an implementation of radix sort that implements these ideas.

1. The function `extractByte` is called with two arguments:
   (a) x is supposed to be an unsigned 32 bit number. With respect to the programming language SetlX this just means that x is a integer satisfying $0 \leq x < 2^{32}$.
   (b) k is the index of the byte that is to be extracted. It is supposed that the least significant byte has the index 1.

Hence, `extractByte`(x, k) extracts the k-th byte of the number x.

`extractByte` works by shifting the number x by $(k-1) \cdot 8$ bits to the left using an integer division. Then, the least significant byte of the resulting number is extracted using the modulus operator.

2. `radixSort` takes a list of unsigned 32 bit integers L as its arguments.
3. It iterates over the four bytes of these numbers starting with the least significant byte.
4. In order to be able to interpret a byte as a grade we have to add 1 to it, since counting sort assumes that the lowest grade is 1. The reason is that grades are used as array indices and in SetlX list indices start at index 1.
5. With respect to counting sort, the elements of the list given to counting sort as its first argument are just arbitrary objects. Therefore, it does not matter whether the first argument to the function `countingSort` is a list of strings interpreted as student names or a list of numbers. This list is sorted with respect to the second argument, the nature of the elements of this list is irrelevant. Therefore, in line 8 the list L is sorted with respect to the k-th byte.
6. When L is returned, this list is sorted with respect to all of the four bytes making up its numbers and hence, it is sorted.

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 the implementiation details of concrete data structures. Therefore, this 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. Abstract data types were proposed by Barbara Liskov and Stephen Zilles in 1974 [LZ74]. Abstract data type are one of the two main ingredients of object oriented programming. The other ingredient is inheritance.

4.1 A Formal Definition of Abstract Data Types

We define an abstract data type $\mathcal{D}$ formally 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 $\mathtt{Stack}$ as follows:

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

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

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

[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. ☺

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

   If we have a stack S, then push an element x of top of S, and finally remove the element on top of the resulting stack, then we are back at the original stack S.

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

   This axiom expresses the fact that the stack created by the function call `stack()` is empty.

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

   If we push an element x on top of a stack S, then the resulting stack cannot be empty.

When contemplating the axioms given above, we can recognize some structure. If we denote the functions `stack` and `push` as generators, then the axioms specify the 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 10 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 that starts with a capital letter.

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`.

```
1   class stack() {
2       mStackElements := [];
3
4     static {
5       push := procedure(e) {
6           this.mStackElements += [e];
7       };
8       pop := procedure() {
9           assert(#mStackElements > 0, "popping empty stack");
10          this.mStackElements := mStackElements[1 .. -2];
11      };
12      top := procedure() {
13          assert(#mStackElements > 0, "top of empty stack");
14          return mStackElements[#mStackElements];
15      };
16      isEmpty := procedure() {
17          return mStackElements == [];
18      };
19      f_str := procedure() {
20          result := convert(this);
21          dashes := "-" * #result;
22          return join([dashes, result, dashes], "\n");
23      };
24      convert := procedure(s) {
25          if (s.isEmpty()) {
26              return "|";
27          }
28          top := s.top();
29          s.pop();
30          return convert(s) + " " + top + " |";
31      };
32    }
33  }
34
35  createStack := procedure(l) {
36      result := stack();
37      n := #l;
38      for (i in [n, n-1 .. 1]) {
39          result.push(l[i]);
40      }
41      return result;
42  };
```

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

The static block itself contains a number of procedure definitions. These procedures are called methods. As these methods are defined inside the static block, they are not considered to be defined in the constructor and, therefore, are not member variables but are, instead, class variables. Hence, they are available in the class `stack`. For example,

```
stack.push
```

refers to the method `push` defined in line 5 to 7. Of course, every object of class `stack` will

also have access to these methods. For example, if s is an object of class `stack`, then we can invoke the method `push` by writing:

s`.push(`x`)`

The reason the methods in class `stack` are all inside the static block is the fact that these methods work the same way for all stacks. The only case where it is not a good idea to declare a method as static is when this method needs to be defined on a per-object-basis: If the code of the method differs for every object so that it is necessary to construct the method for every object differently, then the definition of the method has to be part of the constructor and not part of the static block.

4. Line 5 starts the definition of the method `push`. This method is called with one argument e, where e is the element that is to be pushed on the stack. In the array based implementation, this is achieved by appending e to the list `mStackElements`.

   If you are new to object-oriented programming, then at this point you might wonder why we do not need to specify the stack onto which the element e is pushed. However, we do have to specify the stack by prefixing it to the method invocation. That is, if s is a stack and we want to push e onto this stack, then we can do this by writing the following:

   s`.push(`e`)`

   There is another subtle point that needs to be discussed: When referring to the member variable `mStackElements` we had to prefix this variable with the string "`this.`". In the case that we just want to read a variable, it is not necessary to prefix the variable with "`this.`". However, once we want to change a member variable, the prefix "`this.`" is mandatory. If we just had written

   ```
   mStackElements += [e];
   ```

   then we would have created a new local variable with the name `mStackElements` and we would have changed this local variable rather than the member variable `mStackElements`.

5. Line 8 starts the implementation of the method `pop`, which has the task to remove one element from the stack. Of course, it would not make sense to remove an element from the stack if the stack is empty. Therefore, the `assert` statement in line 9 checks whether the number of elements of the list `mStackElements` is bigger than 0. If this condition is satisfied, the last element of the list `mStackElements` is removed.

6. Line 12 starts the definition of the method `top`. First, it is checked that the stack is non-empty. Then, the element at the end of the list `mStackElements` is returned.

7. Line 16 defines the method `isEmpty`. This method checks whether the list `mStackElements` is empty.

8. Line 19 defines the method `f_str`. This method serves a similar purpose as the method `toString` in a *Java* program: If an object of class `stack` needs to be converted into a string, then the method `f_str` is invoked automatically to perform this conversion.

   In order to understand the implementation of `f_str` we look at an example and type

   ```
   s := stack(); s.push(1); s.push(2); s.push(3); print(s);
   ```

   at the prompt of the interpreter. 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.

At this point it is **important** to notes that in functions and methods, objects are passed by value not by reference! If the argument s had been passed to the function by reference, then the original stack s would effectively by emptied since we pop it until it is empty. However, this is not what happens when convert is executed because `convert` is actually called with a copy of the stack s.

(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 remainder of the integer division of x by y. The expression x `**` y denotes the power of x to the y, i.e. it is the same as 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

$$s + t, \quad s - t, \quad s * t, \quad s \,/\, t, \quad s \,\%\, t, \quad \text{and} \quad s \mathbin{**} t$$

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 (1930-2002) 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. Hence, `mTokens` represents the token stack.
2. `mArguments` is a stack of rational numbers. Therefore, `mArguments` represents the argument stack.
3. `mOperators` is the operator stack containing arithmetic operators. These operators are represented as strings.

Considering the previous example we realize that the numbers appearing in the input tokens are always put onto the argument stack, while there are two cases for the operator symbols that are part of the input tokens:

1. We have to put an operator retrieved from the token stack 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 from the token stack has a higher precedence than the operator that is currently on top of the operator stack.
   (d) The operator from the token stack is the same operator as the operator that is on top of the operator stack and, furthermore, this 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 65 lists the precedences of our operators. This table is coded as the binary relation prec in line 33.

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

```
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.

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

Table 4.1: Precedences of the operators.

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`.

   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 65 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 number of the entries.

3. Next, we discuss balanced ordered trees. In the case of balanced ordered trees the complexity of insertion is guaranteed to be 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 the numbers with the 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 in the first term 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:

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

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.

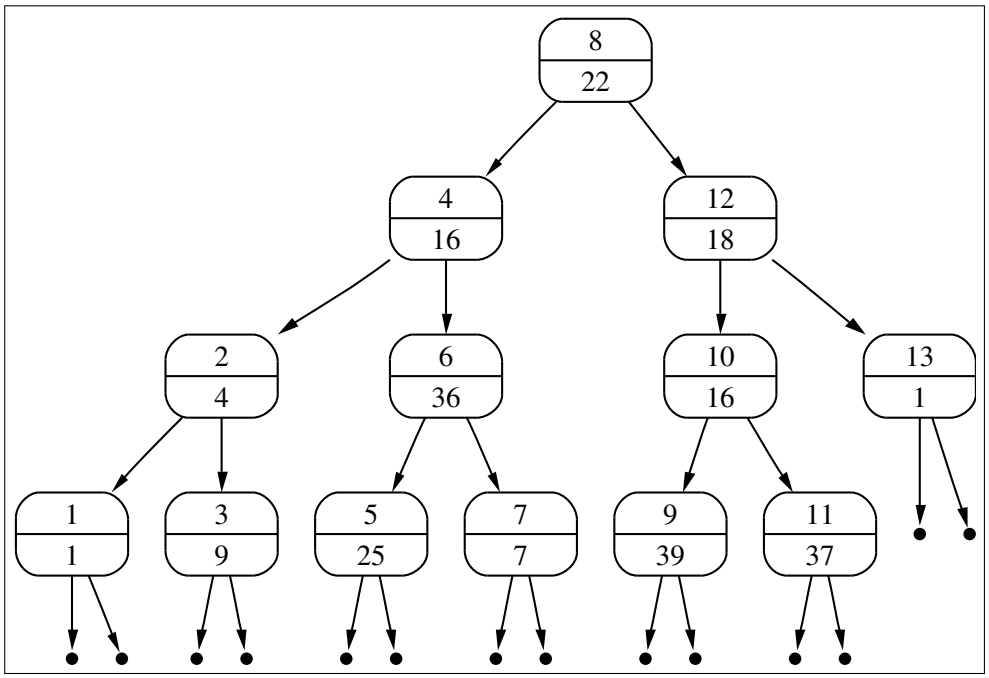

Figure 5.2: An ordered binary tree.

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 74.

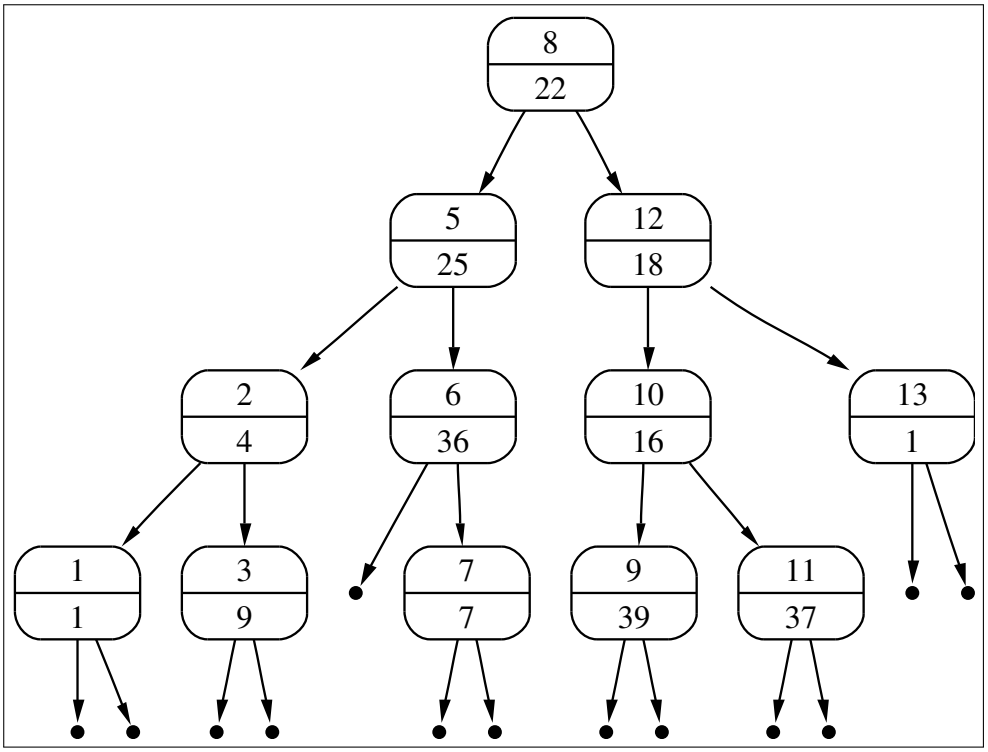

Figure 5.3: The ordered binary tree from Figure 5.2 after deleting the node with label $\langle 4, 16\rangle$.

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

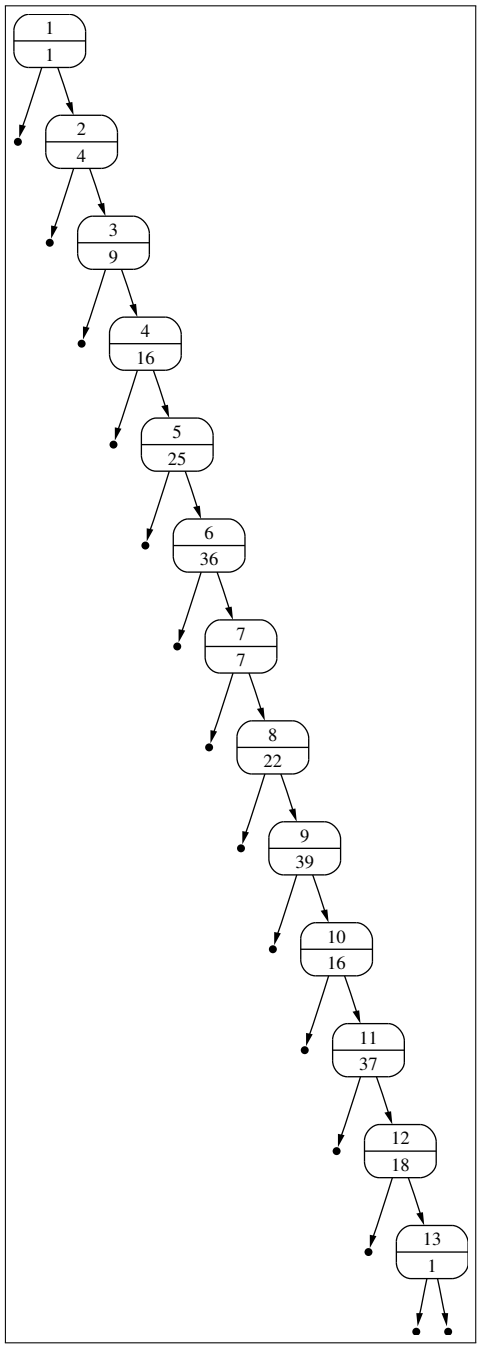

Figure 5.6: A degenerated binary tree.

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 78 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, there 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:

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

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 $b.\texttt{find}(k)$ will always be of the order $\mathcal{O}\big(\ln(n)\big)$. There are a 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 (1922 – 2014) and Evgenii M. Landis (1921 – 1997). In order to define these trees we need to define the height of a binary tree formally:

1. $\texttt{Nil}.\texttt{height}() = 0$.
2. $\texttt{Node}(k, v, l, r).\texttt{height}() = \max\big(l.\texttt{height}(), r.\texttt{height}()\big) + 1$. ◇

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

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

   According to this definition, an AVL tree is an ordered binary tree such that for every node $\texttt{Node}(k, v, l, r)$ 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

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

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 $b.\texttt{restore}()$ 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 $b.\texttt{restore}()$ is invoked we have either of the following two cases:

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

The method restore is specified via conditional equations.

1. $\texttt{Nil}.\texttt{restore}() = \texttt{Nil}$,
   because the empty tree already is an AVL tree.
2. $|l.\texttt{height}() - r.\texttt{height}()| \leq 1 \to \texttt{Node}(k, v, l, r).\texttt{restore}() = \texttt{Node}(k, v, l, r)$.
   If the balancing condition is satisfied, then nothing needs to be done.
3. $$\begin{array}{ll} & l_1.\texttt{height}() = r_1.\texttt{height}() + 2 \\ \wedge & l_1 = \texttt{Node}(k_2, v_2, l_2, r_2) \\ \wedge & l_2.\texttt{height}() \geq r_2.\texttt{height}() \\ \to & \texttt{Node}(k_1, v_1, l_1, r_1).\texttt{restore}() = \texttt{Node}\big(k_2, v_2, l_2, \texttt{Node}(k_1, v_1, r_2, r_1)\big) \end{array}$$

The motivation for this equation can be found in Figure 5.7 on page 84. The left part of this figure shows the state of the tree before it has been rebalanced. Therefore, this part shows the tree

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

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

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

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 $\texttt{Node}(k_2, v_2, l_2, r_2)$ is an AVL tree and we know that $l_2.\texttt{height}() \geq r_2.\texttt{height}()$, 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.

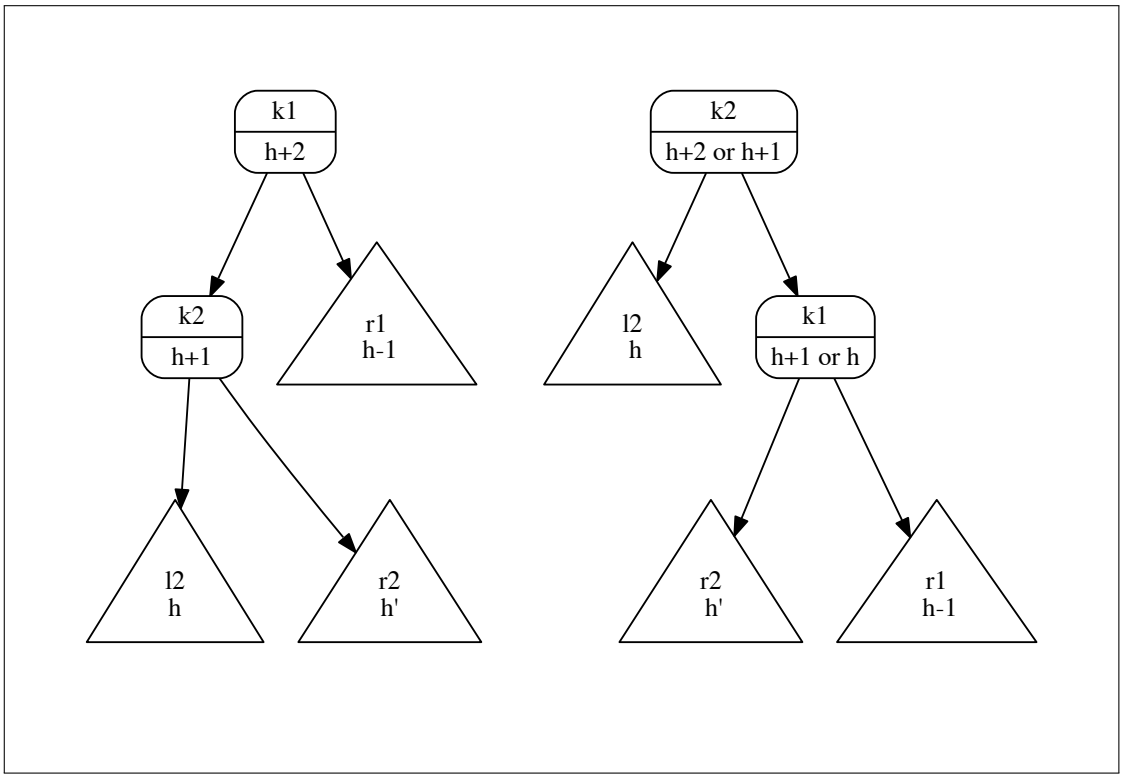

Figure 5.7: An unbalanced tree and the corresponding rebalanced tree.

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 $h-1$ while the height of r_2 is h' and we know that $h' \in \{h, h-1\}$.

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

$$l_2 < k_2 < r_2 < k_1 < r_1. \tag{$\star$}$$

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. $$\begin{array}{ll} & l_1.\texttt{height}() = r_1.\texttt{height}() + 2 \\ \wedge & l_1 = \texttt{Node}(k_2, v_2, l_2, r_2) \\ \wedge & l_2.\texttt{height}() < r_2.\texttt{height}() \\ \wedge & r_2 = \texttt{Node}(k_3, v_3, l_3, r_3) \\ \rightarrow & \texttt{Node}(k_1, v_1, l_1, r_1).\texttt{restore}() = \texttt{Node}\big(k_3, v_3, \texttt{Node}(k_2, v_2, l_2, l_3), \texttt{Node}(k_1, v_1, r_3, r_1)\big) \end{array}$$

The left hand side of this equation is shown in Figure 5.8 on page 85. This tree can be written as

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

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 $\texttt{Node}(k_3, v_3, l_3, r_3)$ would not have the height $h+1$.

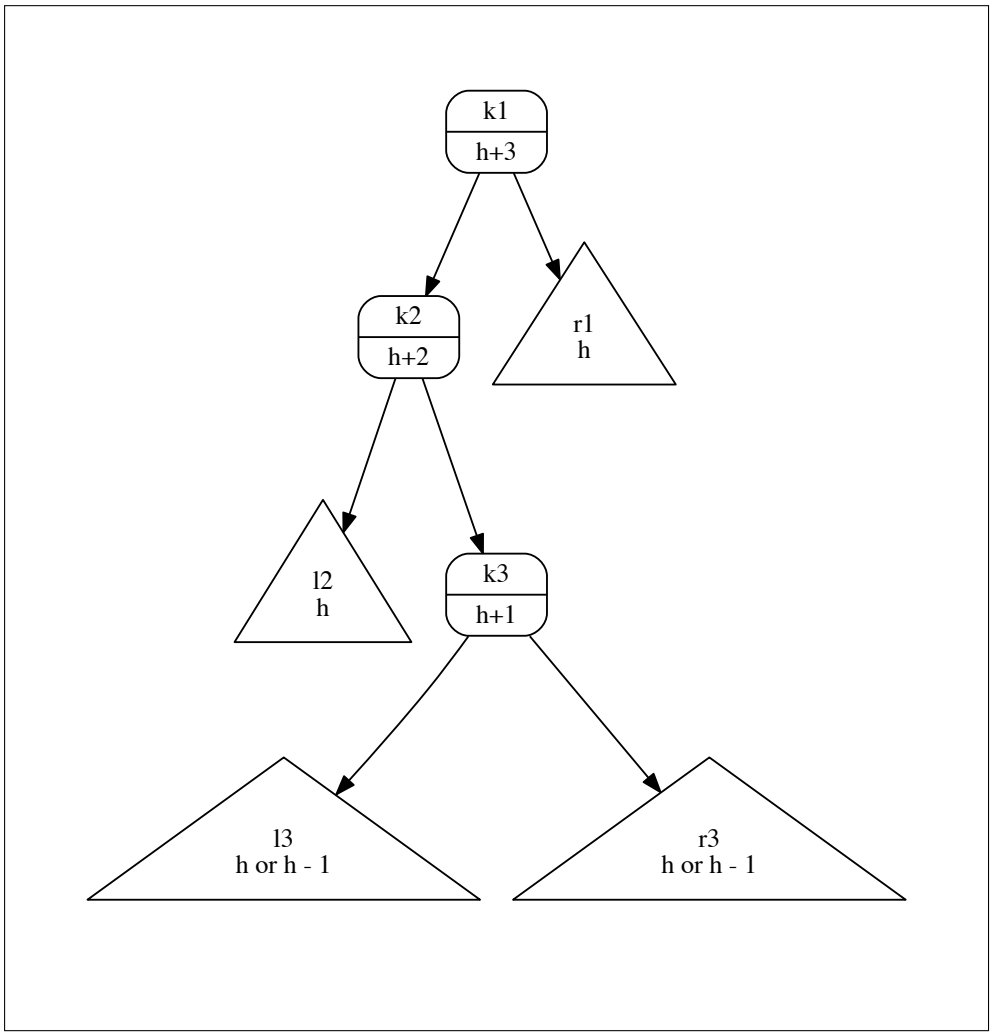

Figure 5.8: An unbalanced tree, second case.

Figure 5.9 on page 86 shows how the tree looks after rebalancing. The tree shown in this figure

has the form

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

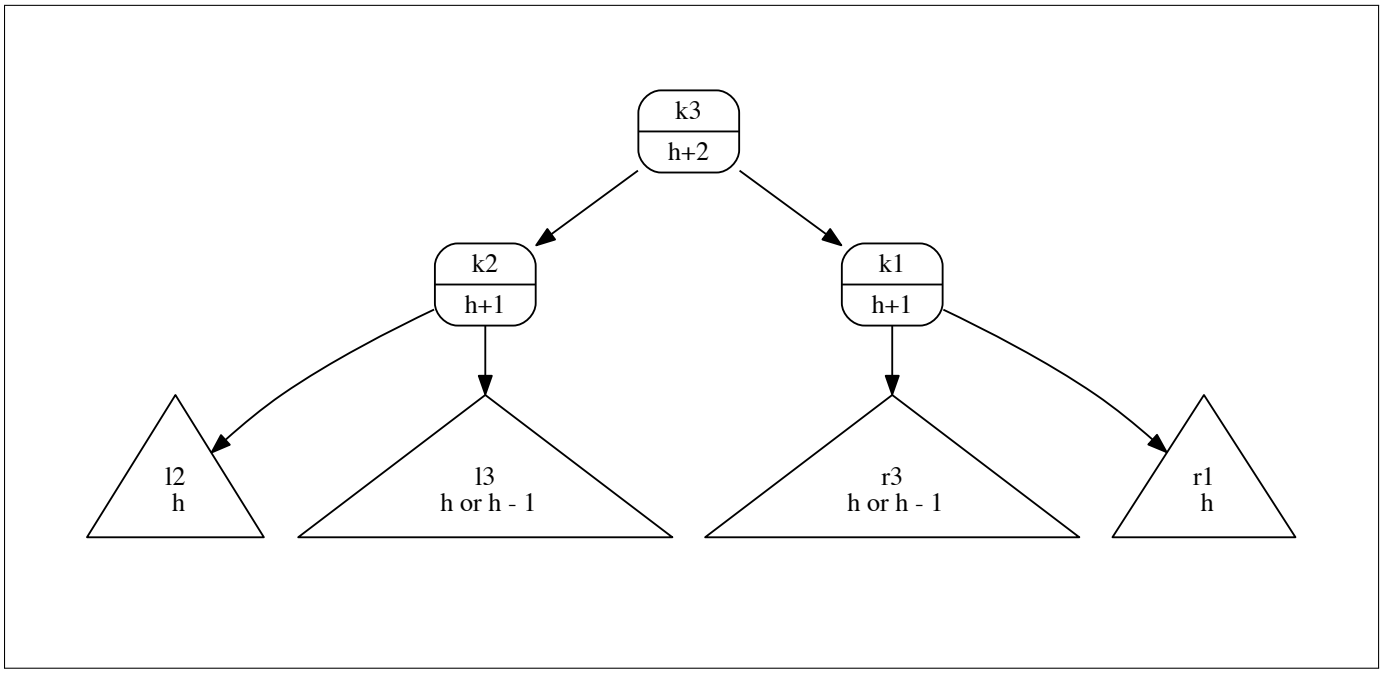

Figure 5.9: The rebalanced tree in the second case.

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 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. $$\begin{array}{ll} & r_1.\texttt{height}() = l_1.\texttt{height}() + 2 \\ \wedge & r_1 = \texttt{Node}(k_2, v_2, l_2, r_2) \\ \wedge & r_2.\texttt{height}() \geq l_2.\texttt{height}() \\ \rightarrow & \texttt{Node}(k_1, v_1, l_1, r_1).\texttt{restore}() = \texttt{Node}\big(k_2, v_2, \texttt{Node}(k_1, v_1, l_1, l_2), r_2\big) \end{array}$$

6. $$\begin{array}{ll} & r_1.\texttt{height}() = l_1.\texttt{height}() + 2 \\ \wedge & r_1 = \texttt{Node}(k_2, v_2, l_2, r_2) \\ \wedge & r_2.\texttt{height}() < l_2.\texttt{height}() \\ \wedge & l_2 = \texttt{Node}(k_3, v_3, l_3, r_3) \\ \rightarrow & \texttt{Node}(k_1, v_1, l_1, r_1).\texttt{restore}() = \texttt{Node}\big(k_3, v_3, \texttt{Node}(k_1, v_1, l_1, l_3), \texttt{Node}(k_2, v_2, r_3, r_2)\big) \end{array}$$

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

4. $k_1 > k_2 \rightarrow \texttt{Node}(k_2, v_2, l, r).\texttt{insert}\big(k_1, v_1\big) = \texttt{Node}\big(k_2, v_2, l, r.\texttt{insert}(k_1, v_1)\big).\texttt{restore}()$.

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

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

Finally, the equations for `delete` are as follows:

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

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.

```
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`.

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.
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.

Since the method `update` replaces the current tree with either its left or right subtree and this subtree is assumed to satisfy the balancing condition, there is no need for a call to `restore` in this method.

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 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`, `l1`, `r1`, `k2`, `v2`, `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`,

```
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`.

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 92.

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.

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

```
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`.

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 the following three properties:

1. The height of $b_h(k)$ is h.
2. The number of keys in $b_h(k)$ is minimal among all other AVL trees of height h.
3. All keys stored in $b_h(k)$ are bigger than k.

```
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`.

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 $\texttt{maxKey}()$. This function has the signature

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

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

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

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

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

   because there is only one AVL tree of height 0 and this is the tree $\texttt{Nil}$.

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

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

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

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

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. Due to the presence of the number 1 on the right hand side of this recurrence equation, this recurrence equation is a so called inhomogeneous recurrence equation. In order to solve this recurrence equation 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:

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

From this we conclude

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

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 $d = -1$. The solution for c_h is now a linear combination of the solutions for the corresponding homogeneous recurrence equation to which we have to add the solution for the inhomogeneous equation:

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

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 $b.\texttt{find}(k)$ is logarithmic even in the worst case. Figure 5.17 presents an AVL tree of height 6 where the number of keys is minimal.

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

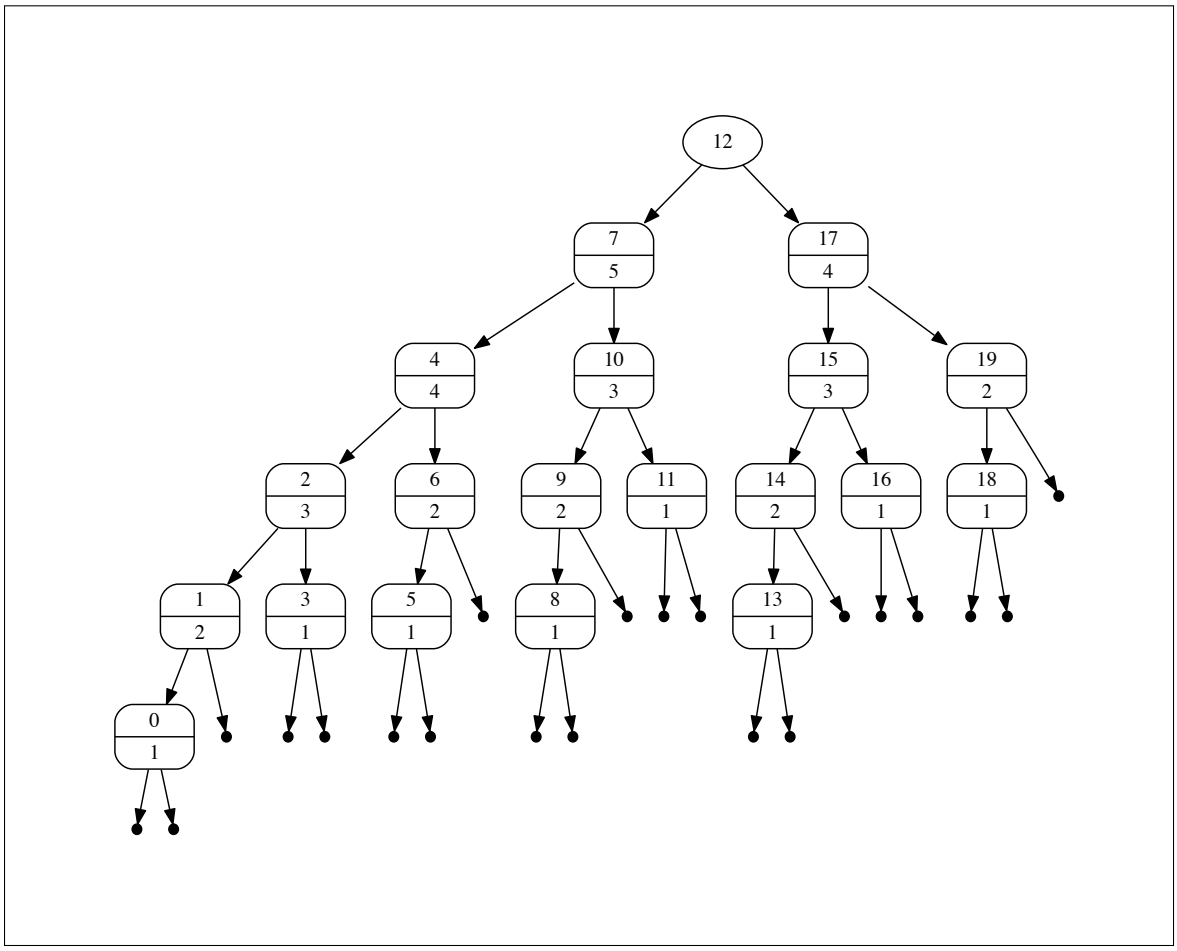

Figure 5.17: An AVL tree of height 6 that is as slim as possible.

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

$$\texttt{Two}(\texttt{Two}(\texttt{Nil}, 1, \texttt{Nil}), 2, \texttt{Three}(\texttt{Nil}, 4, \texttt{Nil}, 5, \texttt{Nil})).$$

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

$$\texttt{Two}(\texttt{Two}(\texttt{Nil}, 1, \texttt{Nil}), 2, \texttt{Four}(\texttt{Nil}, 3, \texttt{Nil}, 4, \texttt{Nil}, 5, \texttt{Nil})).$$

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

$$\texttt{Three}(\texttt{Two}(\texttt{Nil}, 1, \texttt{Nil}), 2, \texttt{Two}(\texttt{Nil}, 3, \texttt{Nil}), 4, \texttt{Two}(\texttt{Nil}, 5, \texttt{Nil})).$$

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.

(d) **Optional**: Specify a method $t.\texttt{delete}(k)$ that deletes the key k in the tree t.

There is a good description of 2-3-trees at

http://cs.wellesley.edu/~cs230/fall02/2-3-trees.pdf

by Franklyn Turbak. 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 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 character of w and if r the string that remains if we remove the first character from w.
- ε denotes the empty string.
- $\texttt{Value}$ 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

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

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

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

then we have

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

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

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

In order to do so, we specify a function

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

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

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

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

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

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

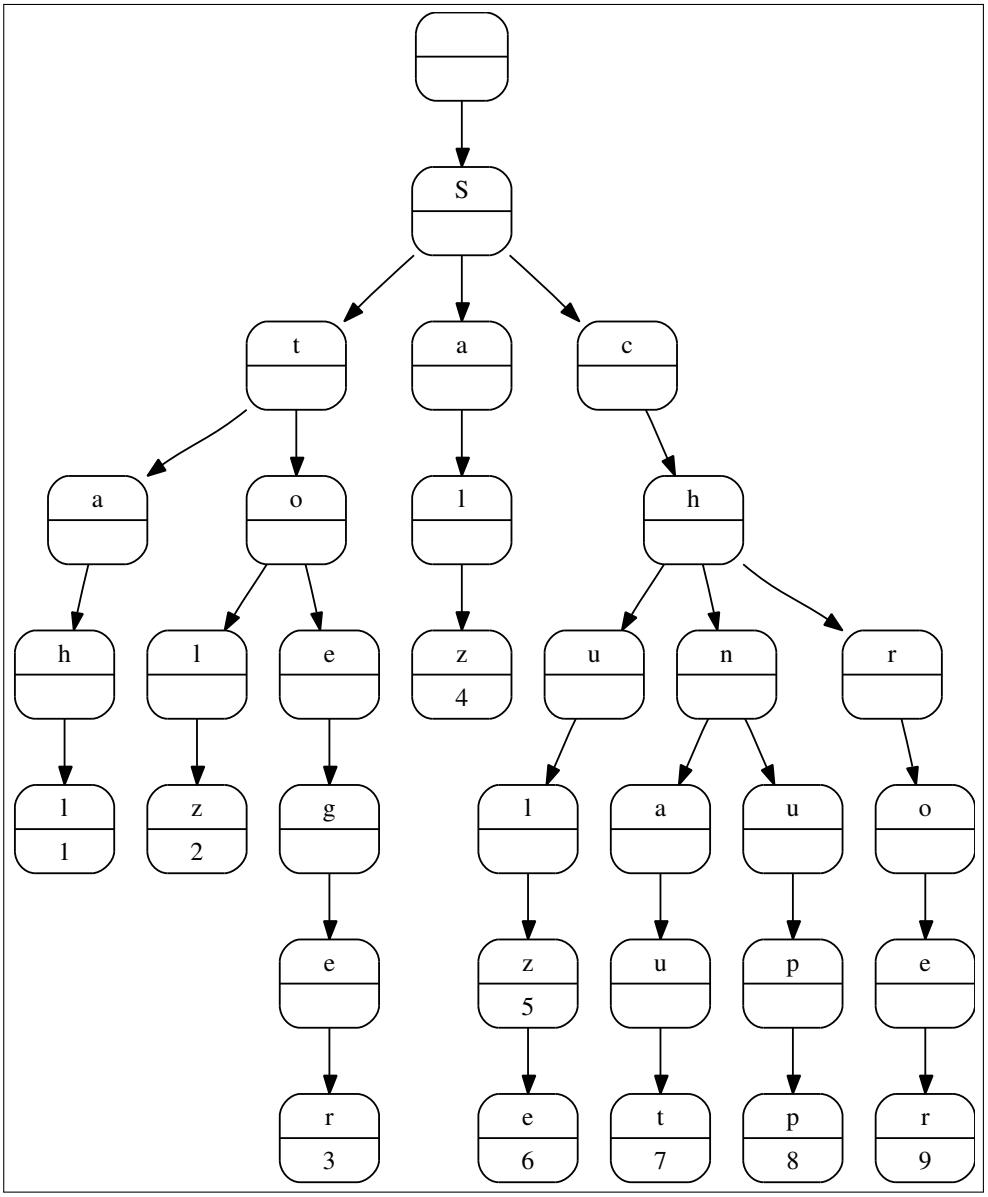

Figure 5.18: A trie storing some numbers.

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

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

we draw a oval. This oval 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 oval. The label that we put in the upper half of the oval 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 oval that represents this node carries the label c_i in its upper half.

In order to clarify these ideas, Figure 5.18 on page 98 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 oval representing the root is always empty. In the example shown in Figure 5.18, the lower half of this oval is also empty because the trie doesn't associate a value with the empty string. In this example, the root node corresponds to the term

$$\texttt{Node}(\Omega, [\text{‘S’}], [t]).$$

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

$$\texttt{Node}(\Omega, [\text{‘t’}, \text{‘a’}, \text{‘c’}], [t_1, t_2, t_3]).$$

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 existing tries, i.e. we specify the method `insert`. The signature of `insert` is given as follows:

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

The result of evaluating

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

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

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

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

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

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

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

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

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

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

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

   If we want to associate a value v with the key cr in the trie $\texttt{Node}\big(v_1, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big)$ 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

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

   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

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

   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

$$\texttt{isEmpty} : \mathbb{T} \to \mathbb{B}.$$

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

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

In the last formula we have used the auxiliary function

$$\texttt{isEmptyList} : \texttt{List}(\mathbb{T}) \to \mathbb{B}.$$

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

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

   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

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

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

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

is defined by induction on the length of w.

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

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

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

   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. $c \notin C \rightarrow \texttt{Node}(v, C, T).\texttt{delete}(cr) = \texttt{Node}(v, C, T)$.

   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

```
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`.

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

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

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.

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.

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.

```
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.

If the loop in line 4 does not find the character c in the list `mChars`, then the method `find` will just return the undefined value Ω in line 9.

```
1   insert := procedure(s, v) {
2       match (s) {
3           case ""   : this.mValue := v;
4           case [c|r]: for (i in [1 .. #mChars]) {
5                           if (mChars[i] == c) {
6                               t := mTries[i];
7                               t.insert(r,v);
8                               this.mTries[i] := t;
9                               return;
10                          }
11                      }
12                      newTrie := trieMap();
13                      newTrie.insert(r, v);
14                      this.mChars += [ c ];
15                      this.mTries += [ newTrie ];
16      }
17  };
```

Figure 5.21: Implementation of `insert` for tries.

The method `insert` takes a string s and an associated value v that is to be inserted into the given trie. The implementation of `insert` works somewhat similar to the implementation of `find`.

1. If the string s is empty, then the value v has to be positioned at the root of this trie. Hence we just set `mValue` to v.

2. Otherwise, s can be written as cr where c is the first character of s while r consists of the remaining characters. In this case, we need to check whether the list `mChars` already contains the character c or not.

   (a) If c is the i-th character of `mChars`, then we have to insert the value v in the trie `mTries`$[i]$. However, this is a little tricky to do. First, we retrieve the subtrie `mTries`$[i]$ and store this trie into the variable t. Next, we can insert the value v into the trie t using the rest r of the string s as the key. Finally, we have to set `mTries`$[i]$ to t. At this point, you might

wonder why we couldn't have just used the statement

$$\texttt{this.mTries}[i].\texttt{insert}(r, v);$$

to achieve the same effect. Unfortunately, this does not work because the expression $\texttt{this.mTries}[i]$ will create a temporary value and inserting v into this temporary value will not change the original list `mTries`.

(b) If c does not occur in `mChars`, things are straightforward: We create a new empty trie and insert v into this trie. Next, we append the character c to `mChars` and simultaneously append the newly created trie that contains v to `mTries`.

```
1   delete := procedure(s) {
2       match (s) {
3           case ""   : this.mValue := om;
4           case [c|r]:
5               for (i in [1 .. #mChars]) {
6                    if (mChars[i] == c) {
7                        t := mTries[i];
8                        t.delete(r);
9                        this.mTries[i] := t;
10                       if (this.mTries[i].isEmpty()) {
11                           this.mChars := removeIthElement(mChars, i);
12                           this.mTries := removeIthElement(mTries, i);
13                       }
14                       return;
15                   }
16              }
17      }
18  };
```

Figure 5.22: Implementation of `delete` for tries.

The method `delete` takes a string and, provided there is a value associated with s, this value is deleted.

1. If the string s is empty, the value associated with s is stored at the root of this trie. In order to remove this value, the variable `mValue` is set to `om`, which represents the undefined value Ω.

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 $\texttt{mTries}[i]$. Again, this is tricky to do. First, we retrieve the subtrie $\texttt{mTries}[i]$ and store this trie into the variable t. Next, the value associated with r is deleted in t and, finally, t is written to $\texttt{mTries}[i]$.

   After the deletion, the subtrie $\texttt{mTries}[i]$ 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.

In order to check whether a given trie is empty it suffices 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. Hence, there is no need to recursively check whether all tries in `mTries` are empty. The implementation is shown in Figure 5.23.

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

Figure 5.23: Implementation of `isEmpty` for tries.

```
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. $\texttt{Nil} \in \mathrm{BT}$.
2. $\texttt{Bin}(v, l, r) \in \mathrm{BT}$ provided that
   (a) $v \in \texttt{Value} \cup \{\Omega\}$ and
   (b) $l, r \in \mathrm{BT}$.

The semantics of binary tries is fixed by defining the function

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

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

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

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 $b.\texttt{find}(n)$ is defined by induction on b. The induction step requires a side induction with respect to the natural number n.

1. $\texttt{Nil}.\texttt{find}(n) = \Omega$,
   since the empty trie doesn't store any values.
2. $\texttt{Bin}(v, l, r).\texttt{find}(0) = v$,
   because 0 is interpreted as the empty string ε.
3. $n \neq 0 \to \texttt{Bin}(v, l, r).\texttt{find}(2 \cdot n) = l.\texttt{find}(n)$,
   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. $\texttt{Bin}(v, l, r).\texttt{find}(2 \cdot n + 1) = r.\texttt{find}(n)$,
   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 simplifies those binary tries that are empty.

(b) Implement binary tries in SetlX.

(c) Test your implementation with a meaningful example.

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

5.5 Hash Tables*

It is very simple to implement a function of the form

$$f : \texttt{Key} \rightarrow \texttt{Value}$$

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

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

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.

```
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.

If the domain $D := \texttt{dom}(f)$ 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 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

$$\texttt{ord} : \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\} \rightarrow \{0, \cdots, 26\}, \quad \text{where}$$

$$\texttt{ord}(\texttt{' '}) := 0, \quad \texttt{ord}(\texttt{'a'}) := 1, \quad \texttt{ord}(\texttt{'b'}) := 2, \quad \cdots, \quad \texttt{ord}(\texttt{'z'}) := 26.$$

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

$$\texttt{code} : \Sigma^* \rightarrow \mathbb{N} \backslash \{0\}$$

as follows:

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

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 $\{1, \cdots, 27^8\}$.

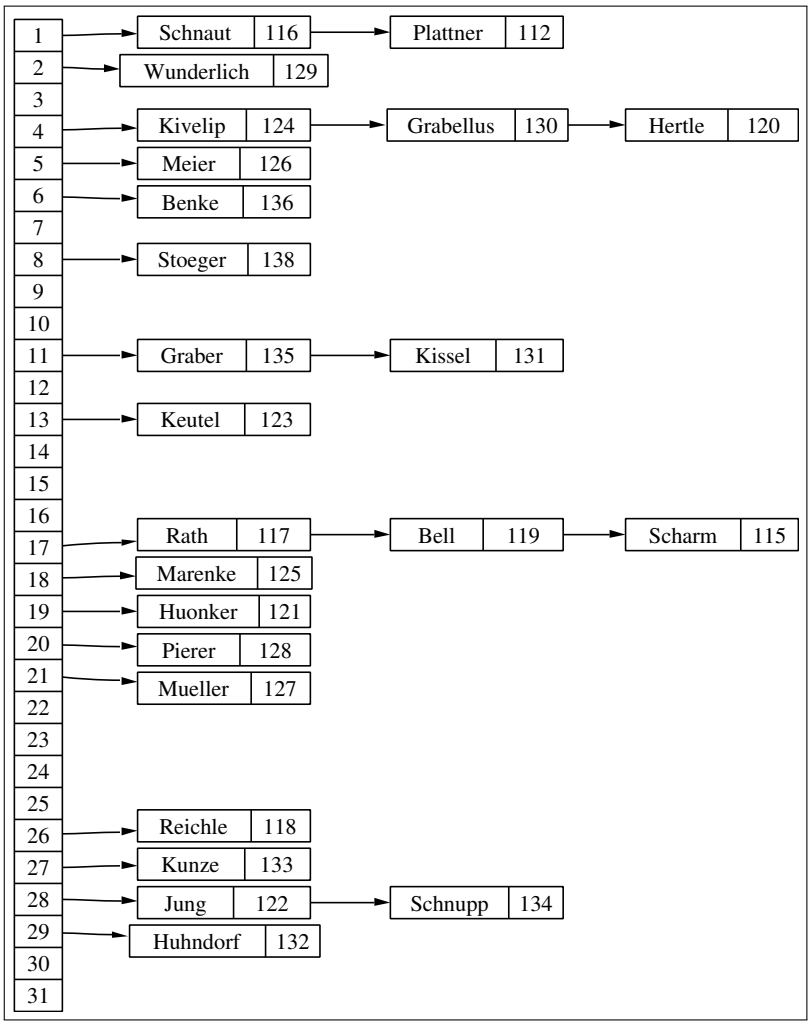

Figure 5.26: A hash table.

Unfortunately, this naive implementation has several problems:

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

$$1 + 26 \cdot \sum_{i=0}^{7} 27^i = 1 + 26 \cdot \frac{27^{7+1} - 1}{27 - 1} = 27^8 = 282\,429\,536\,481$$

   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

   $$\texttt{code}(c_0c_1\cdots c_n) = \left(\sum_{i=0}^{n} \texttt{ord}(c_i)\cdot 27^i\right) \texttt{ \% size} + 1,$$

   then we will always have $1 \leq \texttt{code}(c_0c_1\cdots c_n) \leq \texttt{size}$. In order to prevent overflows when computing the numbers 27^i we can define the partial sum s_k for $k = n, n-1, \cdots, 1, 0$ by induction:

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

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

   Then we have

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

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 itself 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

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

   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 106 shows an example of a map that is implemented along these lines. This data structure is called a hash table. We proceed to discuss the implementation of hash tables.

Figure 5.27 on page 108 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 of 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.

```
class hashTable(n) {
    mSize    := n;  // size of the array
    mEntries := 0;  // number of entries
    mArray   := [ {} : i in [1 .. mSize] ];
    mAlpha   := 2;  // load factor

    static {
        sOrd    := { [ char(i), i ] : i in [ 0 .. 127 ] };
        sPrimes := [ 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039,
                     4093, 8191, 16381, 32749, 65521, 131071,
                     262139, 524287, 1048573, 2097143, 4194301,
                     8388593, 16777213, 33554393, 67108859,
                     134217689, 268435399, 536870909, 1073741789,
                     2147483647
                   ];
        hashCode := procedure(s)          { ... };
        find     := procedure(key)        { ... };
        insert   := procedure(key, value) { ... };
        rehash   := procedure()           { ... };
        delete   := procedure(key)        { ... };
    }
}
```

Figure 5.27: Outline of the class `hashTable`.

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 $\texttt{char}(i)$. Given a number $i \in \{0, \cdots, 127\}$, the function $\texttt{char}(i)$ 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 the 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.
Figure 5.28 gives the implementation of the method `hashCode`.

1. The function $\texttt{hashCode}(s)$ takes a string s and computes a hash code for this string. This hash code satisfies

   $$\texttt{hashCode}(s) \in \{1, \cdots, \texttt{mSize}\}.$$

```
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`.

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.

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`.

```
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`.

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 you are sure that your hash function distributes the keys evenly.

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 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, she will 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 discrete event simulation and in operating systems for the scheduling of processes. Finally, the sorting algorithm heapsort uses a priority queue. We will discuss heapsort in Section 6.4.

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 to implement the algorithms of Dijkstra and Huffman.

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
   $$\{\texttt{Priority}, \texttt{Value}\}.$$
   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
   $$\{\texttt{prioQueue}, \texttt{insert}, \texttt{remove}, \texttt{top}, \texttt{isEmpty}\}.$$
4. The type specifications of these function symbols is given as follows:

(a) $\texttt{prioQueue} : \texttt{PrioQueue}$
This function is the constructor. It creates a new, empty priority queue.

(b) $\texttt{insert} : \texttt{PrioQueue} \times \texttt{Priority} \times \texttt{Value} \to \texttt{PrioQueue}$
The expression $Q.\texttt{insert}(p, v)$ inserts the element v into the priority queue Q. Furthermore, the priority of v is set to be p.

(c) $\texttt{remove} : \texttt{PrioQueue} \to \texttt{PrioQueue}$
The expression $Q.\texttt{remove}()$ removes from Q the element that is returned by $Q.\texttt{top}()$.

(d) $\texttt{top} : \texttt{PrioQueue} \to (\texttt{Priority} \times \texttt{Value}) \cup \{\Omega\}$
The expression $Q.\texttt{top}()$ 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.

(e) $\texttt{isEmpty} : \texttt{PrioQueue} \to \mathbb{B}$
The expression $Q.\texttt{isEmpty}$ checks whether the priority queue Q is empty.

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 the priority. We assume that there exists a set $\texttt{Priority}$ and there is a linear order $\leq$ 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 and in that case the distance to the source is smaller.

   In order to specify the behaviour of the functions $\texttt{top}$, $\texttt{insert}$, $\texttt{remove}$, and $\texttt{isEmpty}$ we need to introduce two auxiliary functions:

   (a) The function $\texttt{toList}$ turns a priority queue into a sorted list. It has the signature

   $$\texttt{toList} : \texttt{PrioQueue} \to \texttt{List}(\texttt{Priority} \times \texttt{Value}).$$

   This function takes a priority queue and turns this priority queue into a list of pairs that is sorted ascendingly according to the priorities. Once we have a working priority queue, we can implement the function $\texttt{toList}$ via the following conditional equations:

   i. $Q.\texttt{isEmpty}() \to Q.\texttt{toList}() = []$,
   ii. $\neg Q.\texttt{isEmpty}() \to Q.\texttt{toList} = [Q.\texttt{top}()] + Q.\texttt{remove}().\texttt{toList}()$.

   (b) The function $\texttt{insertList}$ takes a pair consisting of a priority and a value and inserts it into an ascendingly sorted list of priority-values-pairs such that the resulting list remains sorted. This function has the signature

   $$\texttt{insertList} : \texttt{Priority} \times \texttt{Value} \times \texttt{List}(\texttt{Priority} \times \texttt{Value}) \to \texttt{List}(\texttt{Priority} \times \texttt{Value}).$$

   This function can be specified as follows:

   i. $\texttt{insertList}(p, v, []) = [\langle p, v \rangle]$,
   ii. $p_1 < p_2 \to \texttt{insertList}\big(p_1, v_1, [\langle p_2, v_2 \rangle] + R\big) = [\langle p_1, v_1 \rangle, \langle p_2, v_2 \rangle] + R$,
   iii. $p_1 \geq p_2 \to \texttt{insertList}\big(p_1, v_1, [\langle p_2, v_2 \rangle] + R\big) = [\langle p_2, v_2 \rangle] + \texttt{insertList}\big(\langle p_1, v_1 \rangle, R\big)$.

   Conceptually, this function is the same as the function $\texttt{insert}$ that we had defined when discussing the algorithm insertion sort in Chapter 3.1.

Now we can specify the behaviour of the abstract data type $\texttt{PrioQueue}$.

(a) $\texttt{prioQueue}().\texttt{toList}() = []$
The constructor returns an empty priority queue.

(b) $Q.\texttt{insert}(p, v).\texttt{toList}() = \texttt{insertList}(p, v, Q.\texttt{toList}())$
If a pair $\langle p, v \rangle$ is inserted into a priority queue Q and the resulting priority queue is converted into a list, then the resulting list is the same as if this pair is inserted into $Q.\texttt{toList}()$.

(c) $Q.\texttt{isEmpty}() \leftrightarrow Q.\texttt{toList}() = \texttt{[]}$
A queue Q is empty iff converting Q to a list returns the empty list.

(d) $Q.\texttt{toList}() = \texttt{[]} \rightarrow Q.\texttt{top}() = \Omega$
If we try to retrieve the pair with the highest priority from an empty priority queue, the undefined value Ω is returned instead.

(e) $Q.\texttt{toList}() \neq \texttt{[]} \rightarrow Q.\texttt{top}() = Q.\texttt{toList}()[1]$
If we retrieve the pair with the highest priority from an non-empty priority queue Q, we get the pair that is the first element of the list $Q.\texttt{toList}()$.

(f) $Q.\texttt{toList}() = \texttt{[]} \rightarrow Q.\texttt{remove}().\texttt{toList}() = \texttt{[]}$
Trying to remove the top element from an empty queue Q results in a queue that is still empty.

(g) $Q.\texttt{toList}() \neq \texttt{[]} \rightarrow Q.\texttt{remove}().\texttt{toList}() = Q.\texttt{toList}()[2..]$
If we remove the top element from a non-empty queue Q and then transform the resulting queue into a sorted list, we get the same list that we get when we chop of the first element from the list $Q.\texttt{toList}()$.

The basic idea behind these axioms is the following:

(a) Priority queues are generated using the two constructors `prioQueue` and `insert`.

(b) The behaviour of priority queues is observed using the observer function `toList`. We do not really care how the priority queue works internally. We only care about the results that are observable via the function `toList`.

(c) The functions `top` and `isEmpty` are observers of the ADT `PrioQueue`: They do not change a priority queue, instead they return information about a priority queue. These observer function can be reduced to the more general observer function `toList`.

(d) The function `remove` is a mutator function: It does change a priority queue. The behaviour of this function is specified by describing the effects of these changes that can be observed using the function `toList`.

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. $Q.\texttt{insert}(p, v)$ is implemented by the function `insertList`.
3. $Q.\texttt{top}()$ returns the first element from the list Q.
4. $Q.\texttt{remove}()$ removes the first element from the list 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 a new data structure: Heaps.

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

[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 heap storage. The heap storage is the part of the memory system that is used to provide dynamically allocated memory.

$$\leq \subseteq \mathtt{Priority} \times \mathcal{B}.$$

For a priority $p \in \mathtt{Priority}$ 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. $p \leq \mathsf{Nil}$,

   because there are no priorities in the empty tree Nil.

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

   because p is less than or equal to every priority in the binary tree $\mathsf{Node}(q, v, l, r)$ 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

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

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

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

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

1. $\mathsf{Nil} \in \mathsf{Heap}$.
2. $\mathsf{Node}(p, v, l, r) \in \mathsf{Heap}$ 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) $| \, l.\mathtt{count}() - r.\mathtt{count}() \, | \leq 1$

   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. Therefore, this balancing condition is much stricter than the balancing condition for AVL trees.

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

   This condition ensures that all subtrees of a heap 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 117 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. $\mathsf{Nil}.\mathtt{top}() = \Omega$.
2. $\mathsf{Node}(p, v, l, r).\mathtt{top}() = \langle p, v \rangle$,

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

Implementing the method `isEmpty` is straightforward:

1. $\mathsf{Nil}.\mathtt{isEmpty}() = \mathtt{true}$,
2. $\mathsf{Node}(p, v, l, r).\mathtt{isEmpty}() = \mathtt{false}$.

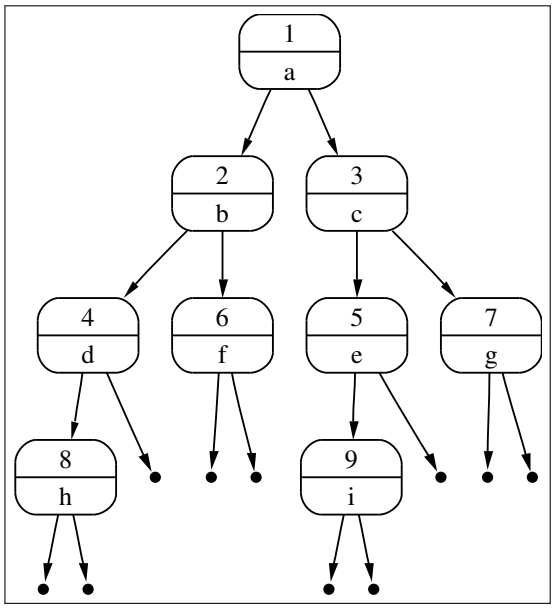

Figure 6.1: A heap.

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

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

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

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

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

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

   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 is bigger than the number of nodes in the right subtree.

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

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

   since we cannot remove anything from the empty heap.

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

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

   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 choose the value that is associated with the higher priority.

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

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

   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 can only shrink. 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

4. `mCount` gives the number of nodes in the subtree rooted at this node.

```
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.

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 simple: We just have to return the value stored at the root. In case that the heap is empty, both the member variables mPriority and mValue would be undefined. In SetlX, a list of the form

$$[\Omega, \Omega]$$

is still the empty list. For example, if x and y are undefined variables, the expression

```
[x,y]
```

returns the empty list []. Therefore, in case of an empty heap Q, the expression Q.top() returns the empty list. This is useful for our implementation of heapsort later.

The implementation of isEmpty is again easy: We just have to check whether the number of values stored into this heap, which is stored in the member variable mCount, 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.mCount is always well defined. If, however, we would allow mLeft to have the value om, then the evaluation of mLeft.mCount 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 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   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`.

1. If the left subtree is empty, we replace the given heap with the right subtree. Conversely, if the right subtree is empty, we replace the given heap with the left subtree.
2. Otherwise, we have to check which of the two subtrees contains the value with the highest priority. This value is then stored at the root of the given tree and, of course, it has to be removed from the subtree that had stored it previously.

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 heaps given in this section has been object-oriented. Your task is to implement heaps in SetlX in a functional way such that heaps are represented as terms.

(a) The empty heap should be represented as the term `@Nil()`.

(b) A heap storing the value v with priority p, left subtree L, right subtree R and c nodes should be represented as the term `@Node`(p, v, L, R, c).

In order to test your implementation, you should implement heapsort. Furthermore, you should implement a function that draws the underlying trees via `Graphviz`.

```
remove := procedure() {
    this.mCount -= 1;
    if (mLeft.isEmpty()) {
        update(mRight);
        return;
    }
    if (mRight.isEmpty()) {
        update(mLeft );
        return;
    }
    if (mLeft.mPriority < mRight.mPriority) {
        this.mPriority := mLeft.mPriority;
        this.mValue    := mLeft.mValue;
        mLeft.remove();
    } else {
        this.mPriority := mRight.mPriority;
        this.mValue    := mRight.mValue;
        mRight.remove();
    }
};
```

Figure 6.4: Implementation of the method `remove`.

```
update := procedure(t) {
    this.mPriority := t.mPriority;
    this.mValue    := t.mValue;
    this.mLeft     := t.mLeft;
    this.mRight    := t.mRight;
    this.mCount    := t.mCount;
};
```

Figure 6.5: Implementation of the method `update`.

Exercise 18: The implementation of the method `remove` given above violates the balancing condition. Modify the implementation of `remove` so that the balancing condition remains valid.

Exercise 19: Instead of defining a class with member variables `mLeft` and `mRight`, a binary tree can be stored as a list L. In that case, for every index $i \in \{1, \cdots, \#L\}$, the expression $L[i]$ stores a node of the tree. The crucial idea is that the left subtree of the subtree stored at the index i is stored at the index $2 \cdot i$, while the right subtree is stored at the index $2 \cdot i + 1$. Develop an implementation of heaps that is based on this idea.

6.4 Heapsort

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 does not require an auxiliary

array. Heapsort was invented in 1964 by J.W.J. Williams and improved by Robert W. Floyd in the same year.

The basic version of heapsort that was given by Williams takes an array A of keys to be sorted and then 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.

This algorithm can be described using an auxiliary function `toHeap` that takes a list of numbers and transforms this list into a heap. The signature of this function is as follows:

$$\texttt{toHeap} : \texttt{List}(\mathbb{N}) \rightarrow \texttt{Heap}$$

This function can be specified via the following equations:

1. $\texttt{toHeap}([\,]) := \texttt{Nil}$
2. $\texttt{toHeap}([x \mid R]) := \texttt{toHeap}(R).\texttt{insert}(x, x)$

Then, the function `heapSort` that takes a list of natural numbers and sorts them can be defined as follows:

$$\texttt{heapSort}(L) := \texttt{toHeap}(L).\texttt{toList}().$$

A basic implementation of heapsort along those lines is given in Figure 6.6 on page 123. 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 the list A into H in line 4. We use the elements of the list A both as priorities and as values.
2. Next we create an empty list S in line 6. 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 $\langle \mathsf{p}, \mathsf{p} \rangle$, 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, the pair is removed $\langle \mathsf{p}, \mathsf{p} \rangle$ from the heap H.
4. Once the heap H has become empty, S contains all of the elements of the list A and is sorted ascendingly.

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 $2 \cdot i + 1$:

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

Here, the expression $\mathsf{A}[i] \doteq \mathsf{Node}(p, v, l, r)$ is to be read as

"The root of the heap $\mathsf{Node}(p, v, l, r)$ is stored at index i in the array A".

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 123 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.

```
heapSort := procedure(A) {
    H := Heap();
    for (x in A) {
        H.insert(x, x);
    }
    S := [];
    while (!H.isEmpty()) {
        S += [ H.top()[1] ];
        H.remove();
    }
    return S;
};
```

Figure 6.6: A basic version of heapsort.

```
swap := procedure(x, y, rw A) {
    [A[x], A[y]] := [A[y], A[x]];
};
sink := procedure(k, rw A, n) {
    while (2 * k <= n) {
        j := 2 * k;
        if (j < n && A[j] > A[j+1]) {
            j += 1;
        }
        if (A[k] < A[j]) {
            return;
        }
        swap(k, j, A);
        k := j;
    }
};
heapSort := procedure(rw A) {
    n := #A;
    for (k in [n\2, n\2-1 .. 1]) {
        sink(k, A, n);
    }
    while (n > 1) {
        swap(1, n, A);
        n -= 1;
        sink(1, A, n);
    }
};
```

Figure 6.7: An implementation of Heapsort in SetlX.

(b) A is the array representing the heap.

(c) n is the size of the part of this array that has to be sorted.

The array A itself might actually have more than n elements, but for the purpose of the method `sink` we restrict our attention to the subarray `A[1..n]`.

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 left to do and the procedure returns. Otherwise, the element at position k is swapped with the element at position `j`. Of course, after this swap it is possible that the heap condition is violated at position `j`. Therefore, k is set to `j` and the `while`-loop continues as long as the node at position k has a child, i.e. as long as $2 \cdot k \leq n$.

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 $n\backslash 2$ 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 root 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 greater than 1. If there is only one element left in this part of the array, the array is sorted and the `while`-loop terminates.

6.4.1 Complexity

Heapsort uses fewer than $2 \cdot n \cdot \log_2(n)$ comparisons to sort a list of n elements. Since it does not need an auxiliary array, it is the algorithm that is to be chosen if there is not enough memory available to run merge sort [SW11b].

Chapter 7

Data Compression

In this chapter we investigate how a given string can be stored so that the amount of memory used to store the string is minimized. We assume that a set of characters Σ is given and that the string s is a finite sequence of characters from Σ, i.e. $s \in \Sigma^*$. The set of characters Σ is called the alphabet. If the alphabet Σ contains k different characters and if we use the same number of bits b for every character in Σ, then the number b of bits must satisfy the inequality

$$k \leq 2^b,$$

which entails that

$$b \geq \texttt{ceil}\big(\log_2(k)\big)$$

holds. Here, $\texttt{ceil}(x)$ denotes the ceiling function. Given a real number x, the expression $\texttt{ceil}(x)$ returns the samllest integer k that is as least as big as x, i.e. we have

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

If the string s has a length of m characters, then we have to use $m \cdot b$ bits in order to code s. There are two options to improve on this number.

1. If we drop the requirement to store all characters with the same amount of bits, then we can save some space. The idea is to code characters occurring very frequently with fewer than b bits while those characters that are very rare are encoded using more than b bits. This approach leads to Huffman's algorithm that was discovered 1952 by David A. Huffman (1925 – 1999) [Huf52].

2. Alternatively we can try to extend the alphabet by interpreting substrings that occur very frequently as new letters. For example, given an English text s, it is quite likely that the substring "*the*" occurs several times in s. If this substring is then coded as a single new character, we might save some space. The Lempel-Ziv-Welch algorithm [ZL77, ZL78, Wel84] was published in 1984 and is based on this idea.

This chapter discusses both Huffman's algorithm and the Lempel-Ziv-Welch algorithm.

7.1 Motivation of Huffman's Algorithm

The main idea of the algorithm developed by Huffman is that letters that occur very frequently are encoded with as few bits as possible, while letters that occur only rarely can be encoded with more bits. To clarify this idea we use the following example: Assume our alphabet Σ contains just four characters, we have

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

The string $s \in \Sigma^*$ that is to be encoded is assumed to contain the letter "a" 990 times, the letter "b" occurs 8 times and the letters "c" and "d" each occur once. Therefore, the string s has a length of

$1\,000$ characters. If we encode each letter with $2 = \log_2(4)$ bits, then we need a total of $2\,000$ bits to store the string s. We will now see that it is possible the store the string s with less than $2\,000$ bits. In our example, the character a occurs much more frequently that the other characters. Therefore, we encode "a" with a single bit. On the other hand, the characters "c" and "d" each occur only once. Therefore, it does no harm if we need more than two bits to encode these characters. Table 7.1 shows an encoding of the characters in Σ that is based on these considerations.

Character	a	b	c	d
Frequency	990	8	1	1
Encoding	0	10	110	111

Table 7.1: Variable-length encoding of the characters.

In order to understand how this encoding works we depict this encoding in Figure 7.1 as a coding tree: The inner nodes of this tree do not contain any attributes and are therefore depicted as empty circles. The leaves of this tree are labelled with characters. The encoding of a character is given by the labelling of the edges that lead from the root of the tree to the leaf containing that character. For example, there is an edge from the root of this tree to the leaf labelled with the digit "0". Hence, the character "a" is encoded by the bit string "0". To give another example we take the character "c". The path that starts at the root and leads to the leaf labelled with "c" consists of three edges. The first two of these edges are labelled with the bit "1", while the last edge is labelled with the bit "0". Therefore, the character "c" is encoded by the bit string "110".

If we now encode the string s that is made up from 990 occurrences of the character "a", 8 occurrences of the character "b" and a single occurrence of both "c" and "d", then we need

$$990 \cdot 1 + 8 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 1\,012$$

bits if we use the variable length encoding shown in Figure 7.1. Comparing this to the fixed width encoding that uses 2 bits per character and therefore uses $2\,000$ bits to store s, we see that we can save $49,4\%$ of the bits with the variable length encoding shown in Table 7.1.

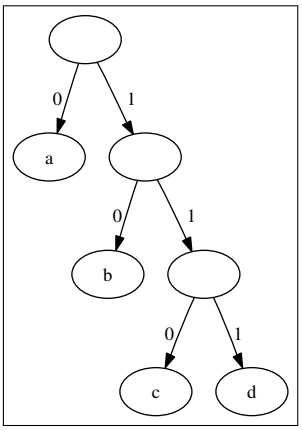

Figure 7.1: Tree representation of the encoding shown in Figure 7.1.

In order to see how a bit string can be decoded using the encoding shown in Figure 7.1 we consider the bit string "100111". We start with the bit "1" which commands us to take the right edge from the root of the coding tree. Next, the bit "0" specifies the left edge. After following this edge we arrive at the leaf labelled with the character "b". Hence we have found the first character. To decode the next character, we return to the root of the tree. The edge labelled "0" takes us to the leaf labelled with the character "a". Hence, we have found the second character. Again, we return to the root of the tree. Now the bits "111" lead us to the character "d". This ends the decoding of the given bit string and we have therefore found that this bit string encodes the string "bad", i.e. we have

"100111" $\simeq$ "bad".

7.2 Huffman's Algorithm

Suppose we have been given a string $s \in \Sigma^*$ where Σ is some alphabet. How do we find an encoding of the letters such that the encoding of s is as short as possible? Huffman's algorithm answers this question. In order to present this algorithm, we first define the set $\mathcal{K}$ of coding trees by induction.

1. $\mathsf{Leaf}(c, f) \in \mathcal{K}$ if $c \in \Sigma$ and $f \in \mathbb{N}$.

   An expression of the form $\mathsf{Leaf}(c, f)$ represent a leaf in a coding tree. Here c is a letter from the alphabet Σ and f is the number of times that the letter c occurs in the string s that is to be encoded.

   Compared to Figure 7.1 this representation adds the frequency f of the letters. This frequency information is needed since we intend to code frequent letters with fewer bits.

2. $\mathsf{Node}(l, r) \in \mathcal{K}$ if $l \in \mathcal{K}$ and $r \in \mathcal{K}$.

   The expressions $\mathsf{Node}(l, r)$ represent the inner nodes of the coding-tree.

Next, we define the function

$$\mathsf{count} : \mathcal{K} \to \mathbb{N}.$$

This function computes the sum of all frequencies of all letters occurring in a given coding tree.

1. For a leaf, the definition of count is obvious:

   $$\mathsf{Leaf}(c, f).\mathsf{count}() = f.$$

2. The sum of all frequencies of a coding tree of the form $\mathsf{Node}(l, r)$ is the sum of all frequencies in l plus the frequencies in r. Therefore we have

   $$\mathsf{Node}(l, r).\mathsf{count}() = l.\mathsf{count}() + r.\mathsf{count}().$$

Next we define the function

$$\mathsf{cost} : \mathcal{K} \to \mathbb{N}.$$

The function cost computes the number of bits that are necessary to encode a string s if all letters occurring in s occur in the the coding tree and if, furthermore, the frequencies of the letters in s are given by the frequencies stored in the coding tree. The definition of $\mathsf{cost}(t)$ is given by induction on the coding tree t.

1. $\mathsf{Leaf}(c, f).\mathsf{cost}() = 0$.

   As long as the coding tree has no edges, the resulting encoding has zero bits.

2. $\mathsf{Node}(l, r).\mathsf{cost}() = l.\mathsf{cost}() + r.\mathsf{cost}() + l.\mathsf{count}() + r.\mathsf{count}()$.

   If two coding trees l and r are combined into a new coding tree, the encoding of all letters occurring in either l or r grows by one bit: The encoding of a letter in l is prefixed with the bit "0", while the encoding of a letter from r is prefixed with the bit "1". The sum

   $$l.\mathsf{count}() + r.\mathsf{count}()$$

   counts the frequencies of all letters occurring in the coding tree. As the encoding of all these letters is lengthened by one bit, we have to add the term $l.\mathsf{count}() + r.\mathsf{count}()$ to the costs of l and r.

The function $\mathsf{cost}()$ is extended to sets of coding trees. If M is a set of coding trees, then we define

$$\mathsf{cost}(M) := \sum_{n \in M} n.\mathsf{cost}().$$

The algorithm that was published by David A. Huffman in 1952 [Huf52] starts with a set of pairs of the form $\langle c, f \rangle$ where c is a letter and f is the frequency of this letter on a given string s that is to be encoded. In the first step of this algorithm, these pairs are turned into leaves of a coding tree. Assume that the string s is built from the letters

$$c_1,\ c_2,\ \cdots,\ c_k$$

and that the frequencies of theses letters are givens as

$$f_1,\ f_2,\ \cdots,\ f_k.$$

Then the set of coding trees is given as

$$M = \{\mathsf{Leaf}(c_1, f_1), \cdots, \mathsf{Leaf}(c_k, f_k)\}. \tag{7.1}$$

Huffmann's algorithm combines two nodes a and b from M into a new node $\mathsf{Node}(a, b)$ until the set M contains just a single node. When combining the nodes of M into a single tree we have to take care that the cost of the resulting tree should be minimal. Huffman's algorithm takes a greedy approach: The idea is to combine those nodes a and b such that the cost of the set

$$M \backslash \{a, b\} + \{\mathsf{Node}(a, b)\}$$

is as small as possible. In order to choose a and b let us investigate how much the cost increases if we combine the two nodes into the new node $\mathsf{Node}(a, b)$:

$$\begin{array}{lll} & \mathtt{cost}\big(N \cup \{\mathsf{Node}(a,b)\}\big) - \mathtt{cost}\big(N \cup \{a,b\}\big) \\ = & \mathtt{cost}\big(\{\mathsf{Node}(a,b)\}\big) - \mathtt{cost}\big(\{a,b\}\big) \\ = & \mathsf{Node}(a,b).\mathtt{cost}() - a.\mathtt{cost}() - b.\mathtt{cost}() \\ = & a.\mathtt{cost}() + b.\mathtt{cost}() + a.\mathtt{count}() + b.\mathtt{count}() - a.\mathtt{cost}() - b.\mathtt{cost}() \\ = & a.\mathtt{count}() + b.\mathtt{count}() \end{array}$$

We see that if we combine a and b into the new node $\mathsf{Node}(a, b)$, the cost is increased by the sum

$$a.\mathtt{count}() + b.\mathtt{count}().$$

If our intention is to keep the cost small then it suggests itself to pick those nodes a and b from M that have the smallest count and replace them with the new node $\mathsf{Node}(a, b)$. This process is then iterated until the set M contains but a single node. It can be shown that this procedure yields a coding tree that codes the given string using the smallest number of bits.

```
1   codingTree := procedure(M) {
2       while (#M > 1) {
3           a := first(M);
4           M -= { a };
5           b := first(M);
6           M -= { b };
7           M += { [ a[1] + b[1], @Node(a, b) ] };
8       }
9       return arb(M);
10  };
```

Figure 7.2: Huffman's algorithm implemented in SetlX.

The function `codingTree` program shown in Figure 7.2 implements this algorithm.

1. The function `codingTree` is called with a set M of nodes. This set has the form

   $$M = \{[f_1, \mathsf{Leaf}(c_1)], \cdots, [f_k, \mathsf{Leaf}(c_k)]\}.$$

Here, c_1, $\cdots$, c_k are the different character that occur in the string s that is to be encoded. For every character c_i, the number f_i counts the number of times that c_i occurs in the string s.

In the pairs $[f_i, \texttt{Leaf}(c_i)]$ the frequency f_i is stored first. As SetlX stores this set internally as an ordered binary tree that is sorted ascendingly, this fact enables us to extract the node with the lowest frequency by calling the function `first`. The reason is that in SetlX pairs of the form $[f, \texttt{Leaf}(c)$ are compared lexicographically: In order to compare two pairs

$$[f_1, \texttt{Leaf}(c_1)] \quad \text{and} \quad [f_2, \texttt{Leaf}(c_2)]$$

SetlX first compares the frequencies f_1 and f_2. If f_1 is less than f_2, the pair $[f_1, \texttt{Leaf}(c_1)]$ is considered to be smaller than $[f_2, \texttt{Leaf}(c_2)]$, if f_2 is bigger than f_2, then the pair $[f_1, \texttt{Leaf}(c_2)]$ is considered to be bigger than $[f_2, \texttt{Leaf}(c_2)]$. Finally, if the frequencies f_1 and f_2 are the same, the characters c_1 and c_2 are compared to determine the order. Hence, the first element of the set M is the pair that has the smallest frequency. Effectively, this trick turns the set M into a priority queue:

(a) The function `first` that is predefined in SetlX returns the first element of the set M and thus the call $\texttt{first}(M)$ achieves the same as the call $M.\texttt{top}()$ would achieve if M had been implemented as a priority queue.

(b) In order to insert an element v with priority v into M we can use the expression

$$M \texttt{ += \{ [}p\texttt{, } v\texttt{] \};}$$

instead of writing $M.\texttt{insert}(p, v)$, which would have been necessary if M had been implemented as a priority queue.

(c) In order to remove the top element from the priority queue M we can use the expression

$$M \texttt{ -= \{ first(}M\texttt{) \};}$$

This achieves the same effect as the call $M.\texttt{remove}()$ would have achieved if M had been implemented as a priority queue.

What makes this approach elegant is the fact that with this implementation all operations of the abstract data type *PrioQueue* have logarithmic complexity. In the case of the operation $M.\texttt{top}()$ this is not optimal as the data structure heap achieves a constant complexity in this case. However, for every call of the form $M.\texttt{top}()$ there is a call of the form $M.\texttt{remove}()$ and this call has a logarithmic complexity even if we implement the priority queue as a heap. This complexity would dominate the overall complexity so that in the end the heap based implementation of the abstract data type *PrioQueue* has the same complexity as the set based implementation.

2. The `while` loop in line 2 reduces the number of nodes of the set M in every step by one.

   (a) Using the function $\texttt{first}()$, we compute those nodes a and b that have the lowest count.

   (b) These two nodes are removed from M .

   (c) Next a and b are combined into the new node $\texttt{Node}(a, b)$ that is added to M.

3. The `while` loop terminates when M contains but a single element. This element is then extracted using the function `arb` and is returned as the result.

The running time of Huffman's algorithm is given as $\mathcal{O}\big(n \cdot \ln(n)\big)$ where n denotes the number of different characters occurring in the string s. The reason is that all the operations inside the `while` loop have at most a logarithmic complexity in n and the loop is executed $n - 1$ times.

Character	a	b	c	d	e
Frequency	1	2	3	4	5

Table 7.2: Letters with their frequencies.

We demonstrate Huffman's algorithm by computing the coding tree that results from a string s containing the letters "a", "b", "c", "d", and "e" where the number of occurrence of these letters are given in table 7.2 on page 129.

1. Initially, the set M has the form

   $$M = \{\langle 1, \mathsf{Leaf}(\mathsf{a})\rangle, \langle 2, \mathsf{Leaf}(\mathsf{b})\rangle, \langle 3, \mathsf{Leaf}(\mathsf{c})\rangle, \langle 4, \mathsf{Leaf}(\mathsf{d})\rangle, \langle 5, \mathsf{Leaf}(\mathsf{e})\rangle\}.$$

2. Apparently, the characters "a" and "b" occur with the lowest frequency. Hence, these characters are removed from M and instead the node

   $$\mathsf{Node}(\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b}))$$

   is added to the set M. The frequency of this new node is given as the sum of the frequencies of the characters "a" and "b". Hence, the pair

   $$\langle 3, \mathsf{Node}\big(\mathsf{Leaf}(\mathsf{a})\rangle, \mathsf{Leaf}(\mathsf{b})\big)\rangle$$

   is inserted into the set M. The resulting form of M is

   $$\{\langle 3, \mathsf{Leaf}(\mathsf{c})\rangle, \langle 3, \mathsf{Node}(\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b}))\rangle, \langle 4, \mathsf{Leaf}(\mathsf{d})\rangle, \langle 5, \mathsf{Leaf}(\mathsf{e})\rangle\}.$$

3. The two pairs with the smallest frequencies are now

   $$\langle 3, \mathsf{Node}(\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b}))\rangle \quad \text{and} \quad \langle 3, \mathsf{Leaf}(\mathsf{c})\rangle.$$

   These pairs are removed form M and replaced by

   $$\langle 6, \mathsf{Node}(\mathsf{Node}((\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b})),\ \mathsf{Leaf}(\mathsf{c}))\rangle.$$

   Then M is given as

   $$\Big\{\langle 4, \mathsf{Leaf}(\mathsf{d})\rangle,\ \langle 5, \mathsf{Leaf}(\mathsf{e})\rangle,\ \langle 6, \mathsf{Node}(\mathsf{Node}(\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b})),\ \mathsf{Leaf}(\mathsf{c}))\Big\}.$$

4. Now the pairs

   $$\langle 4, \mathsf{Leaf}(\mathsf{d})\rangle \quad \text{and} \quad \langle 5, \mathsf{Leaf}(\mathsf{e})\rangle$$

   are the pairs with the smallest frequency. We remove them and construct the new node

   $$\langle 9, \mathsf{Node}(\mathsf{Leaf}(\mathsf{d}), \mathsf{Leaf}(\mathsf{e}))\rangle.$$

   This node is added to the set M and then M is

   $$\Big\{\langle 6, \mathsf{Node}(\mathsf{Node}(\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b})),\ \mathsf{Leaf}(\mathsf{c}, 3))\rangle,\ \langle 9, \mathsf{Node}(\mathsf{Leaf}(\mathsf{d}, 4), \mathsf{Leaf}(\mathsf{e}, 5))\rangle\Big\}.$$

5. Now the set M has just two elements. These are combined into the single node

   $$\mathsf{Node}\Big(\mathsf{Node}\big(\mathsf{Node}\big(\mathsf{Leaf}(\mathsf{a}), \mathsf{Leaf}(\mathsf{b})\big),\ \mathsf{Leaf}(\mathsf{c})\big),\ \mathsf{Node}\big(\mathsf{Leaf}(\mathsf{d}), \mathsf{Leaf}(\mathsf{e})\big)\Big).$$

   This node is our result. Figure 7.3 shows the corresponding coding tree. Here every node n is labelled with its count. The resulting encoding is shown in table 7.3.

Character	a	b	c	d	e
Encoding	000	001	01	10	11

Table 7.3: Variable length encoding of the letters "a" to "e".

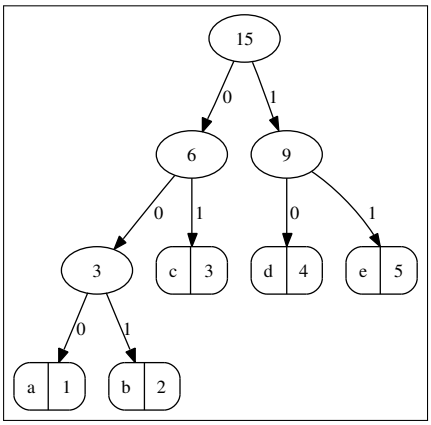

Figure 7.3: Tree representation of the coding tree.

Exercise 20:

(a) Compute the Huffman code for a string s that contains the letters "a" through "g" where the frequencies are given by the following table:

Character	a	b	c	d	e	f	g
Frequency	1	1	2	3	5	8	13

Table 7.4: Letters with frequencies.

(b) How many bits do we save when we use the Huffman encoding compared to a fixed with encoding?

(c) Try to guess the law that has been used to specify the frequencies in the table given above and try to compute the Huffman code in the general case where the string s has n different characters. ◇

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:

$$l = \texttt{lzw().compress}(s_1) \wedge s_2 = \texttt{lzw().uncompress}(l) \rightarrow s_1 = s_2.$$

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.

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.

```
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.

4. Next, the method `longestPrefix` computes the index of longest prefix of the substring s[`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[`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[`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.

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.

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

$$l[\texttt{idx}] = \texttt{code}$$

```
longestPrefix := procedure(s, i) {
   oldK := i;
   k    := i+1;
   while (k <= #s && mDictionary[s[i..k]] != om) {
       oldK := k;
       k    += 1;
   }
   return oldK;
};
incrementBitNumber := procedure() {
    if (2 ** mBitNumber <= mNextCode) {
        this.mBitNumber += 1;
    }
};
```

Figure 7.6: Computing the longest prefix.

```
uncompress := procedure(l) {
    result := "";
    idx    := 1;
    code   := l[idx];
    old    := mInverse[code];
    idx    += 1;
    while (idx < #l) {
        result += old;
        code := l[idx];
        idx  += 1;
        next := mInverse[code];
        if (next == om) {
            next := old + old[1];
        }
        mInverse[mNextCode] := old + next[1];
        this.mNextCode += 1;
        old := next;
    }
    result += old;
    return result;
};
```

Figure 7.7: The method `uncompress` to decode a list of integers into a string.

4. The variable `old` contains the substring associated with `code`. Therefore, the invariant

   $$\texttt{mInverse}[\texttt{code}] = \texttt{old}$$

   is maintained.

5. As long as the index `idx` still points inside the list, the substring that has just been decoded is appended to the string `result`.

6. Then, an attempt is made to decode the next number in the list l by looking up the code in the dictionary `mInverse`.

Now there is one subtle case: If the `code` has not yet been defined in the dictionary, then we can conclude that this code has been created when coding the substring `old` followed by some character c. However, as the next substring β corresponds to this code, the character c must be the first character of this substring, i.e. we have

$$c = \beta[1].$$

On the other hand, we know that the substring β has the form

$$\beta = \texttt{old} + c,$$

where the operator "+" denotes string concatenation. But then the first character of this string must be the first character of `old`, i.e. we have

$$\beta[1] = \texttt{old}[1]$$

and hence we have shown that

$$c = \texttt{old}[1].$$

Therefore, we conclude

$$\beta = \texttt{old} + \texttt{old}[1]$$

and hence this is the string encoded by a code that is not yet defined in the dictionary `mInverse`.

7. Next, we need to maintain the dictionary `mInverse` in the same fashion as the dictionary `mDictionary` is maintained in the method `compress`: Hence we take the string previously decoded and concat the next character of the string decoded in the current step. Of course, this string is

   $$\texttt{old} + \texttt{next}[1]$$

   and this string is then associated with the next available code value.

8. At the end of the loop, we need to set `old` to `next` so that `old` will always contain the string decoded in the previous step.

9. When the `while`-loop has terminated, we still need to append the final value of `old` to the variable `result`.

Now that we have discussed the implementation of the LZW algorithm I would like to encourage you to test it on several examples that are not too long. Time does not permit me to discuss examples of this kind in these lecture notes and, indeed, I do not think that discussing these examples here would be as beneficial for the student as performing the algorithm on their own.

Exercise 21:

(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 problems from graph theory.

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 equivalence relation that is generated by R. The equivalence relation generated by the relation R is the smallest equivalence relation $\approx_R$ such that $R \subseteq \approx_R$.

   Essentially, the union-find problem is a mathematical problem. Nevertheless, we will see that it has an important practical application in computer science.

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 subgraph that connects all vertices of the graph. We discuss Kruskal's algorithm for solving this problem.

3. Finally, we discuss the problem of finding a shortest path in a *weighted directed graph*. We present Dijkstra's algorithm to solve this problem.

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 generates an equivalence relation $\approx_R$ on M. This generated 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, i.e. it ensures that $R \subseteq \approx_R$.

2. For every $x \in M$ we have $\langle x, x \rangle \in \approx_R$.

   This 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 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 the equivalence relations $R \subseteq M \times M$ and the partitions 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 generated 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 generates an equivalence relation $\approx$ where we have $x \approx y$ if and 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 $\approx$ generated 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       for ([x, y] in R) {
4           Sx := find(x, P);
5           Sy := find(y, P);
6           if (Sx != Sy) {
7               P  -= { Sx, Sy };  // remove old sets
8               P  += { Sx + Sy }; // add their union
9           }
10      }
11      return P;
12  };
13  find := procedure(x, P) {
14      return arb({ S : S in P | x in S });
15  };
```

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 $\approx_{\mathsf{R}}$ that is generated by `R` on M. 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 yet 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 P in line 8, while the equivalence classes Sx and Sy are removed from P in line 7.

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 the union

$$Sx + Sy.$$

If the sets Sx and Sy are represented as binary trees and, for the sake of the argument, the set Sx contains at most as many elements as the set Sy, then the computational complexity of this operation is

$$\mathcal{O}\big(\#Sx \cdot \log_2(\#Sy)\big).$$

The reason is that every element of Sx has to be inserted into Sy and this insertion has a complexity of $\mathcal{O}\big(\log_2(\#Sy)\big)$. Here the expression #Sx denotes the size of the set Sx and similarly the expression #Sy denotes the size of the set 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 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 straightforward: 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 141 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 then 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] `Parent` 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, then `x` and `y` are already equivalent and there is nothing to do. However, if `x` and `y` are located in different trees, then these trees need to be merged. To this end, the parent pointer of the root of the tree containing `y` is changed so that it 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 if we 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`, then 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 143. Provided the size of the relation `R` is bounded by the size n of the set `M`, the complexity of this implementation is $\mathcal{O}\big(n \cdot \log(n)\big)$. However, this excludes the last two lines of the program. In practice, the implementation of the function `unionFind` would omit these two lines and, instead, return the relation `Parent` since this is all that is needed to determine whether two elements x and y are equivalent.

Exercise 22: We can speed up the implementation previously shown if the set `M` has the form

$$\texttt{M} = \{1, 2, 3, \cdots, n\} \quad \text{where } n \in \mathbb{N}.$$

In this case, the relations `Parent` and `Height` can be implemented as arrays. Develop an implementation that is based on this idea. ◇

[1] In a language like C we would instead use pointers. Of course, this would be more efficient.

```
1   unionFind := procedure(M, R) {
2       Parent := { [x, x] : x in M };
3       Height := { [x, 1] : x in M };
4       for ([x, y] in R) {
5           rootX := find(x, Parent);
6           rootY := find(y, Parent);
7           if (rootX != rootY) {
8               if (Height[rootX] < Height[rootY]) {
9                   Parent[rootX] := rootY;
10              } else if (Height[rootX] > Height[rootY]) {
11                  Parent[rootY] := rootX;
12              } else {
13                  Parent[rootY] := rootX;
14                  Height[rootX] += 1;
15              }
16          }
17      }
18      Roots := { x : x in M | Parent[x] == x };
19      return { { y : y in M | find(y, Parent) == r } : r in Roots };
20  };
```

Figure 8.3: A more efficient version of the union-find algorithm.

8.1.3 Packaging the Union-Find Algorithm as a Data Structure

When we later discus 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 144.

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,
   ```

   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 function `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 generated by `R` on `M`.

```
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);
    };
  }
}

partition := procedure(M, R) {
    uf := unionFind(M);
    for ([x, y] in R) {
        uf.union(x, y);
    }
    Roots := { x : x in M | uf.find(x) == x };
    return { { y : y in M | uf.find(y) == r } : r in Roots };
};
```

Figure 8.4: The class unionFind.

(a) First, the function constructs a union-find object uf for the set M.

(b) Then the method iterates over all pairs $[x, y]$ in the relation R and joins the equivalence classes corresponding to x and y.

(c) Next, the method collects all nodes x that are at the root of a tree.

(d) Finally, for every root R the method collects those nodes x that are part of the tree rooted at R.

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) A weighted undirected 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.
   In practical applications, the weight of an edge is often interpreted as the cost or the length of the 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 set of all paths is denoted as $\mathbb{P}$, i.e. we define

$$\mathbb{P} := \big\{P \mid P \text{ is a path}\big\}.$$

A path $P = [x_1, \cdots, x_n]$ 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, i.e. we have

$$\forall x, y \in \mathbb{V} : \exists P \in \mathbb{P} : \big(P[1] = x \wedge P[\#P] = y\big).$$

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. the corresponding graph is connected **and**
2. removing any 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 23: Assume that the graph $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ is a tree. Prove that the equation

$$\mathit{card}(\mathbb{E}) = \mathit{card}(\mathbb{V}) - 1$$

holds.

Hint: The easiest way to prove this is by induction on $n = \mathit{card}(\mathbb{V})$. 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 shown in the previous section. 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 by adding the edge $\langle x, y \rangle$ to the tree we are building.

   The fact that we investigate the edges in increasing order of their weight implies that we always choose the cheapest edge to connect two nodes that are not yet connected. Hence, Kruskal's algorithm is a greedy algorithm.

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 146. We discuss this program next.

```
1  mst := procedure(V, E) {
2      uf     := unionFind(V);
3      Result := {};
4      for([w, [x, y]] in E) {
5          rx := uf.find(x);
6          ry := uf.find(y);
7          if (rx != ry) {
8              Result += { [w, [x, y]] };
9              uf.union(rx, ry);
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`, which is short for minimum spanning tree. This function takes two arguments: V is the set of nodes and E is the set of edges of a given graph. 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 $\langle V, E \rangle$. It is assumed that this graph 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 by calling the function `union`.
7. The algorithm returns if the tree `Result` has #V $-$ 1 edges, since we know from the previous exercise that in that case all nodes have to be 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 algorithms to solve 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 can be viewed as one-way streets, while in a graph they represent streets that can be used in both directions. Hence, if a digraph has an edge $\langle a, b \rangle$, then this edge enables us to get from a to b but this edge does not enable us to go from b to a. On the other hand, if a graph has an edge $\{a, b\}$, then this edge enables us to go from a to b as well as to go from b to a. ◇

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:

$$\texttt{sp} : \mathbb{V} \to \mathbb{N}$$
$$\texttt{sp}(v) := \texttt{min}\big\{\|p\| \;\big|\; p \in \mathbb{P}(\texttt{source}, v)\big\}.$$

Furthermore, given that $\texttt{sp}(v) = n$, 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 148 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       Distance := { [source, 0] };
3       Fringe   := { source };
4       while (Fringe != {}) {
5           u := from(Fringe);
6           for ([v,l] in Edges[u]) {
7               if (Distance[v] == om || Distance[u]+l < Distance[v]) {
8                   Distance[v] := Distance[u] + l;
9                   Fringe      += { v };
10              }
11          }
12      }
13      return Distance;
14  };
```

Figure 8.6: The Bellman-Ford algorithm to solve the shortest path problem.

1. The function $\texttt{shortestPath}(\texttt{source}, \texttt{Edges})$ is called with two arguments:
   (a) source is the start node. Our intention is to compute the distance of all vertices from the node source.
   (b) Edges is a functional binary relation that encodes the set of edges of the graph. For every node x the set $\texttt{Edges}[x]$ has the form
   $$\big\{[y_1, l_1], \cdots, [y_n, l_n]\big\}.$$

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.

For example, the relation Edges defined below defines a simple digraph. In that digraph, there is an edge from node "a" to node "b" which has a length of 2 and there is another edge from node "a" to the node "c" that has a length of 3.

```
Edges := { ["a", {["b", 2], ["c", 3]}],
           ["b", {["d", 1]} ],
           ["c", {["e", 3]} ],
           ["d", {["e", 2], ["f", 4]} ],
           ["e", {["f", 1]} ],
           ["f", {} ]
         };
```

2. The variable Distance is a functional binary relation. When the computation is successful, for every node x this relation will contain a pair of the form $[x, \mathsf{sp}(x)]$ showing that the node x has distance $\mathsf{sp}(x)$ from the node source.

   The node source has distance 0 from the node source and initially this is all we know. Hence, the relation Distance is initialized as the set $\{[\mathtt{source}, 0]\}$.

3. The variable Fringe is a set of those nodes where we already have an estimate of their distance from the node source, but where we haven't yet computed the distances of their neighbours from the node source. Every iteration of the while loop computes the distances of all those nodes y that are connected to a node in Fringe by an edge $\langle x, y \rangle$. Initially we only know that the node source is connected to the node source. Therefore, 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 shorter path leading to any of these nodes. There are two cases.

   (a) If there is an edge from u to a node v and Distance[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 $\langle \mathsf{u}, \mathsf{v} \rangle$.

   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

The Bellman-Ford algorithm doesn't specify how the nodes are picked from the set Fringe. The result of this non-determinism is that a node can be picked from Fringe and removed from Fringe only to be reinserted into Fringe later. This is inefficient. In 1959 Edsger W. Dijkstra (1930 – 2002) [Dij59] published an algorithm that removed this non-determinism. Furthermore, Dijkstra's algorithm guarantees that it is never necessary to reinsert a node back into the set Fringe. Dijkstra had the idea to always choose the node that has the smallest distance to the node source. To do this, we

```
1  shortestPath := procedure(source, Edges) {
2      Distance := { [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]) {
9              if (Distance[v] == om || d + l < Distance[v]) {
10                 Fringe      -= { [Distance[v], v] };
11                 Distance[v] := d + l;
12                 Fringe      += { [d + l, v] };
13             }
14         }
15         Visited += { u };
16     }
17     return Distance;
18 };
```

Figure 8.7: Dijkstra's algorithm to solve the shortest path problem.

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 150 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

$$\forall u \in \texttt{Visited} : \texttt{Distance}[u] = \texttt{sp}(u),$$

i.e. for all nodes $u \in$ `Visited`, the functional relation `Distance` 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

$$\texttt{Distance}[u] = \texttt{sp}(u)$$

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

   $$\texttt{sp}(\texttt{source}) = 0 = \texttt{Distance}[\texttt{source}].$$

   Surely, the distance from `source` to `source` is 0. Hence the claim is true initially.

2. **Induction Step:** In line 15 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 by contradiction and we assume that we have

$$\texttt{Distance}[\mathsf{u}] > \mathsf{sp}(\mathsf{u})$$

Then there must exist a shortest path

$$p = [x_0 = \texttt{source}, x_1, \cdots, x_n = \mathsf{u}]$$

leading from source to u that has length $\mathsf{sp}(u)$ and this length is less than $\texttt{Distance}[\mathsf{u}]$. 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 $\mathsf{u} \notin \texttt{Visited}$. 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 Distance is updated if this edge leads to a shorter path. In particular, x_{i+1} has been examined and $\texttt{Distance}[x_{i+1}]$ 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

$$\texttt{Distance}[x_{i+1}] = \mathsf{sp}(x_{i+1}),$$

because by induction hypothesis we have that $\texttt{Distance}[x_i] = \mathsf{sp}(x_i)$ 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 $\texttt{Distance}[x_{i+1}] \geq \texttt{Distance}[u]$, 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 $\mathsf{sp}(x_{i+1}) = \texttt{Distance}[x_{i+1}]$ we now have

$$\mathsf{sp(u)} \geq \mathsf{sp}(x_{i+1}) = \texttt{Distance}[x_{i+1}] \geq \texttt{Distance[u]} > \mathsf{sp(u)},$$

i.e. we have shown the contradiction

$$\mathsf{sp(u)} > \mathsf{sp(u)},$$

This shows that the assumption $\texttt{Distance[u]} > \mathsf{sp(u)}$ has to be wrong. On the other hand, since we always have $\texttt{Distance[u]} \geq \mathsf{sp(u)}$ we can only conclude that

$$\texttt{Distance[u]} = \mathsf{sp(u)}$$

holds for all nodes $\mathsf{u} \in \texttt{Visited}$. □

Exercise 24: 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

$$\mathsf{sp(u)} = \texttt{Distance[u]}$$

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 Distance[v] is still undefined or if the value Distance[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

$$\mathcal{O}\big(\#\mathbb{V} * \log_2(\#\mathbb{V})\big).$$

8.4 Topological Sorting

Assume that you have a huge project to manage. In order to do so, you can use PERT, which is short for program evaluation and review technique. Using Pert, you break your project into a number of tasks

$$T := \{t_1, \cdots, t_n\}.$$

Furthermore, there are dependencies between the task: These dependencies take the form $s \prec t$, where s and t are tasks. A dependency of the form $s \prec t$ means that the task s has to be finished before the task t can start. For example, if the overall project is to get dressed in the morning[2], the set T of task is given as follows:

$$T := \{\textit{socks}, \textit{trousers}, \textit{shirt}, \textit{shoes}\}.$$

The elements of T are interpreted as follows:

- *socks*: Put on socks.
- *trousers*: Put on trousers.
- *shirt*: Put on shirt.
- *shoes*: Put on shoes.

There are some dependencies between these tasks: For example, putting on the shoes first and then trying to put on the socks does not work. In detail, we have the following dependencies between the different tasks:

1. *socks* $\prec$ *shoes*,
2. *trousers* $\prec$ *shoes*,
3. *shirt* $\prec$ *trousers*.

Now the problem is to order the tasks such that the dependencies are satisfied. For example, for the project of dressing up in the morning, the following schedule would work:

socks, *shirt*, *trousers*, *shoes*.

[2] In order to keep things manageably simple, I have made the assumption that the person that needs to get dressed is male.

For the simple task of dressing up in the morning, most of you would get the scheduling right most of the time[3]. However, complex projects can have ten thousands of tasks and even more dependencies between these tasks. For example, the US-lead invasion in the second Irak war was a project consisting of more than $30\,000$ tasks. The US invasion plan for the People's Republic of China even contain more than 2 million tasks! Ordering the tasks for a project of this size is very difficult to do by hand. Instead, a technique called topological sorting is used.

8.4.1 Formal Definition of Topological Sorting

In order to better grasp the idea of a topological sorting problem, we define it formally as a graph theoretical problem.

Definition 20 (Topological Sorting Problem, Solution of a Topological Sorting Problem)
A topological sorting problem is a directed graph $\langle T, D\rangle$ where

- T is called the set of tasks and
- $D \subseteq T \times T$ is called the set of dependencies.

  If $\langle s, t\rangle \in D$, then this is written as

  $$s \prec t, \quad \text{which is read as } t \text{ depends on } s.$$

For conciseness, we abbreviate "topological sorting problem" as $\textsc{Tsp}$.

Mathematically, a solution to a topological sorting problem is a linear ordering $\leq$ that satisfies

$$\forall s, t \in T : \big(s \prec t \rightarrow s < t\big),$$

where $s < t$ is defined in terms of $s \leq t$ as follows

$$s < t \stackrel{\text{def}}{\iff} s \leq t \wedge s \neq t.$$

In practical applications, the set of tasks T is always finite. Then, the linear ordering is computed as a list S of tasks. In order for a list S of tasks to be a solution of a $\textsc{Tsp}$ $\langle T, D\rangle$, we need to have

$$\forall i, j \in \{1, \cdots, \#S\} : \big(S[i] \prec S[j] \rightarrow i < j\big),$$

i.e. if the task $S[j]$ depends on the task $S[i]$, then $S[i]$ needs to precede $S[j]$ in the list S. Furthermore, every task $t \in T$ occurs exactly once in S. This requirement is equivalent to the equation

$$\mathsf{card}(T) = \mathtt{length}(S),$$

i.e. the number of elements of the set T is the same as the length of the list S. ◇

8.4.2 Computing the Solution of a Tsp

Next, we present Kahn's algorithm [Kah62] for solving a $\textsc{Tsp}$. The basic idea is very simple. Ask yourself: How can a list S that is supposed to be a solution to a *Tsp* $\langle T, D\rangle$ start? Of course, it can only start with a task that does not depend on any other task. Therefore, if we are unfortunate and every task depends on some other task, there is no way to solve the $\textsc{Tsp}$. Otherwise, we can just pick any task that does not depend on another task to be the first task and remove it from the set of tasks. After that it's just rinse and repeat. Therefore, Kahn's algorithm for solving a $\textsc{Tsp}$ $\langle T, D\rangle$ works as follows:

1. Initialize the list S of tasks to the empty list.
2. Pick a task $t \in T$ that does not depend on any other task and append this task to S.

   If there is no such task, the $\textsc{Tsp}$ is not solvable and the algorithm terminates with failure.

[3] If you have a bad hangover, the correct scheduling can turn out to be more challenging.

3. Remove the task t from both T and D, i.e. if there is a pair $\langle t, s\rangle \in T$ for some s, then this pair is removed from S.

4. If T is not yet empty, go back to step 1.

When the algorithm terminates, the list S is a solution to the TSP $\langle T, D\rangle$.

In order to implement this algorithm we need to think about the data structures that are needed. In order to be able to pick a task $t \in T$ that does not depend on any other task, we need a set of all those tasks that do not depend on any other task. Using graph theoretical manner of speaking we call this set the set of orphans, since in graph theory, if we have an edge $\langle s, t\rangle$, then s is called a parent of t and, therefore, a node with no parents is called an orphan. In order to maintain the set of orphans, we need to be able to compute the parents of each node. Furthermore, when we remove a node t from the graph after inserting it into the list S, we have to compute the set of children of that node. The reason is that we have to update the set of parents of the children of t. This leads to the algorithm shown in Figure 8.8 on page 154.

```
1   topoSort := procedure(T, D) {
2       Parents  := { [t, {}] : t in T };  // dictionary of parents
3       Children := { [t, {}] : t in T };  // dictionary of children
4       for ([s, t] in D) {
5           Children[s] += { t }; Parents[t] += { s };
6       }
7       Orphans := { t : [t, P] in Parents | P == {} };
8       Sorted  := [];
9       while (T != {}) {
10          assert(Orphans != {}, "The graph is cyclic!");
11          t := from(Orphans);  T -= { t };  Sorted += [t];
12          for (s in Children[t]) {
13              Parents[s] -= { t };
14              if (Parents[s] == {}) {
15                  Orphans += { s };
16              }
17          }
18      }
19      return Sorted;
20  };
```

Figure 8.8: Kahn's algorithm for topological sorting.

1. In order to compute the parents and children of a given node efficiently, Kahn's algorithm uses two dictionaries: `Parents` and `Children`. In lines 2 and 3 we initialize these dictionaries to be empty and the `for`-loop in line 4 fills both of these dictionaries: If there is a dependency $\langle$`s`, `t`$\rangle$, then `t` is a child of `s` and `s` is a parent of `t`. Therefore, `t` is added to the set `Children[s]` and `s` is added to the set `Parents[t]`.

2. In line 7, we compute the set `Orphans` of those tasks that have no `Parents`.

3. In line 8, the list `Sorted` is initialized. When the algorithm completes without failure, this list will contain a solution of the TSP.

4. As long as there are still tasks in the set T that have not been inserted into the list `Sorted` the algorithm keeps going.

5. If at any point the set of tasks T that need to be scheduled is not yet empty but all remaining tasks still have parents, i.e. depend on the completion of some other task, then the given TSP is not solvable and the algorithm terminates with a failure message.

6. Otherwise, we remove a task `t` from the set of orphaned tasks, remove this task from T and append it to the list of scheduled tasks `Sorted`.

7. Finally, we have to update the parent dictionary so that for all children `s` of the task `t`, the task `t` is no longer a parent of the task `s`. Furthermore, if this implies that the task `s` has no parents left, it needs to be added to the set `Orphans`.

8.4.3 Complexity

If the number of dependencies that any given task is involved in is bounded by some fixed constant and n is the number of tasks, then the complexity of Kahn's algorithm as given in Figure 8.8 is

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

The reason is that building the dictionary `Parents` as well as the dictionary `Children` both have the complexity $\mathcal{O}\big(n \cdot \log_2(n)\big)$. Furthermore, maintaining the set `Orphans` has a complexity of $\mathcal{O}\big(n \cdot \log_2(n)\big)$.

Chapter 9

The Monte-Carlo Method

Some problems are just too complex to be solved exactly. Often, these problem can be solved approximately via simulation.

1. The computation of the volumes of a three-dimensional object can be reduced to multiple integrals. However, if the object has a complicated surface, then computing these integrals exactly might be impossible. In this case, the Monte-Carlo method is often able to compute a rough approximation of the volume. Although the precision attained using the Monte-Carlo method is limited, the method is often very easy to implement.
2. Complex systems that are influenced by random events are often difficult to predict. Their behaviour can often only be understood via random simulations. For example, if a new underground transportation system is planned, the capacity of the system is estimated via simulation.
3. In games of chance it is often not possible to compute all probabilities exactly. However, Monte-Carlo simulations can be used to compute reasonable estimates of these probabilities.

This list can easily be extended. In this chapter, we investigate three applications of the Monte-Carlo method.

1. As an introductory example we show how the Monte-Carlo method can be used to compute the area of a circle. This way, the number π can be approximated via a simulation.
2. The Monty Hall problem is a brain teaser that has puzzled a lot of people. In my personal experience I have found it easiest to convince people via a simulation.
3. The final example shows how cards can be randomly shuffled. This can be used to compute the probability that a given Poker hand wins against a random hand.

9.1 How to Compute π via Simulation

The unit circle U is defined as the set

$$U = \big\{\langle x, y\rangle \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1\big\}.$$

The set U contains those points $\langle x, y\rangle$ that have distance of 1 or less from the origin $\langle 0, 0\rangle$. The unit circle is a subset of the square Q that is defined as

$$Q := \big\{\langle x, y\rangle \in \mathbb{R}^2 \mid -1 \leq x \leq +1 \wedge -1 \leq y \leq +1\big\}$$

A simple algorithm to compute π works as follows: We randomly throw n grains of sand into the square Q. Then we count the number of grains that end up in the unit circle U. Call this number k. It is reasonable to assume that approximately k is to n as the area of U is to the area of Q. As the area of Q is $2 \cdot 2$ and the area of U equals $\pi \cdot 1^2$, we should have

$$\frac{k}{n} \approx \frac{\pi}{4}.$$

Multiplying by 4 we get

$$\pi \approx 4 \cdot \frac{k}{n}.$$

Now instead of using grains of sand we can run a simulation. Figure 9.1 on page 157 shows the resulting program.

```
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.

1. The parameter n specifies the number of grains of sand that are thrown into the square Q.
2. In order to throw a grain of sand randomly into Q we first compute random numbers using the function `random()`. These random numbers are distributed uniformly in the interval $[0, 1]$. The transformation

   $$t \mapsto 2 \cdot t - 1$$

   maps the interval $[0, 1]$ into the interval $[-1, 1]$. Hence, the coordinates `x` und `y` that are computed in the lines 5 and 6 represent a random point $\langle \mathtt{x}, \mathtt{y} \rangle$ in the square Q.
3. Line 7 computes the square of the distance between $\langle x, y \rangle$ and $\langle 0, 0 \rangle$. If this distance is less or equal than 1, then the point $\langle x, y \rangle$ is inside the unit circle U. In this case, the counter k is incremented.

If we run this program, we get the results shown in table 9.1 on page 158. In order to compute π with a precision of 10^{-3} we need about $10\,000\,000$ trials. Given the computational power of modern computers this number can be achieved within seconds. However, if we need more precision, things get harder: In order to achieve an error less than 10^{-4} we already need $1\,000\,000\,000$ trials. Every time that we want to increase the precision by a factor of ten, we need to increase the number of trial by a hundred times!

Monte-Carlo simulations are efficient as long as only rough estimates are needed. However, if we need high precision results, the Monte-Carlo method gets inefficient.

n	approximation of π	absolute error
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: Results for the approximate computation of π using a Monte-Carlo simulation.

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 25: 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

$$\begin{aligned} & 3 \cdot \sqrt{\frac{p \cdot (1-p)}{n}} && = && 10^{-7} \\ \Leftrightarrow \quad & 9 \cdot \frac{p \cdot (1-p)}{n} && = && 10^{-14} \\ \Leftrightarrow \quad & 9 \cdot p \cdot (1-p) \cdot 10^{14} && = && n \end{aligned}$$

Um an dieser Stelle weitermachen zu können, benötigen wir den Wert der Wahrscheinlichkeit p. Der korrekte Wert von p ist für unser Experiment durch $\frac{\pi}{4}$ gegeben. Da wir π ja erst berechnen wollen, nehmen wir als 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 26: 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 a 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 161 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.

[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. *EinfÃijhrung 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.

[Kah62] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[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.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[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.

[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.