Algorithms

— Lecture Notes for the Summer Term 2021 —

Prof. Dr. Karl Stroetmann

June 19, 2021

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 then be found in the file

`Lecture-Notes/algorithms.pdf`.

These lecture notes are still subject to some 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`. If you find any errors or typos in these lecture notes I would be grateful if you could report these errors via email or, if you are one of my students, via discord. Alternatively, you are welcome to send a pull request via 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. Correctness of algorithms

   An algorithm is useless unless it is correct. We discuss two methods to verify the correctness of an algorithm.

   (a) We first demonstrate the method of computational induction. This method can be used to prove the correctness of recursive functions.

   (b) Then we demonstrate the method of symbolic execution, which can be used to verify functions that are implemented with the help of loops.

3. 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) merge sort,
   (c) quicksort,
   (d) radix sort, and
   (e) heapsort.

4. Abstract data types

   Abstract data types enable us to describe the behaviour of a data structure in a concise way. Furthermore, abstract data types are part of the foundations of object-oriented programming.

5. Dictionaries and sets

   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

   A priority queue is a data structure that can best be described as a sorted list that remains sorted when elements are inserted or removed. 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. Data compression

   There are basically two algorithms for loss-less data compression: The algorithm of Lempel, Ziv, and Welch and Huffman's algorithm. For reasons of space we will only discuss the latter algorithm.

8. 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.

9. Monte Carlo Method

   Many important problems either do not have an exact solution at all or the computation of an exact solution would be prohibitively expensive. In these cases it is often possible to use 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 as 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. 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.

(a) 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.

(b) 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.

(c) Finally, we can specify an algorithm mathematically. The language of mathematics 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*. Furthermore, this has the added benefit that we are able to test our algorithm. For this reason we present *Python* implementations of all the algorithms covered in this lecture.

1.4 Desirable Properties of Algorithms

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

(a) Algorithms have to be correct.

(b) Algorithms should be efficient with respect to both computing time and memory.

(c) 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 very 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, maintenance might become a nightmare and it might be impossible to guarantee 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 the 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 exam. While I will most gladly  answer these questions, I should warn you that, unfortunately 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, or so I hope. 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 the discord server that I am using as a discussion forum for this lecture. All your questions are welcome since they give me valuable feedback ~~how stupid you really are~~ to improve my lecture.

Chapter 2

The Complexity of Algorithms

This chapter discusses the computational complexity of algorithms. In order to do so it introduces the big $\mathcal{O}$ notation, which is a notion that is used to describe the growth of a function. This notation is needed when analysing the running time of algorithms. Furthermore, the big $\mathcal{O}$ notation is part of the technical language of computer science and hence you need to understand this notation when conversing with your colleagues.

In order to illustrate the application of these notations, we will 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. After that, we discuss the master theorem, which enables us to estimate the growth rate of functions that are defined recursively. Finally, we learn how to solve recurrence relations. These are recursive equations that arise from the complexity analysis of functions that are recursively defined.

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. Additionally, we have to count all storage accesses.
3. We look up the amount of time that is needed for the different operations in the processor handbook.
4. Using the information discovered in the previous two steps we predict the running time of our algorithm for given input.

This approach is problematic for a number of reasons.

(a) It is both very time consuming and very complicated.

(b) 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.

(c) 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.

This 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:

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

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:

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

The definition of $\mathcal{O}(g)$ contains three nested quantifiers and may be difficult to understand when first encountered. Therefore, let us analyse this definition carefully. 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 a multiple of g

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

[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.

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

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

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

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

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

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

   holds for all values of n that are sufficiently big. This implies that, for example, if $f \in \mathcal{O}(g)$ holds, then 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 $f(x) \leq c \cdot g(x)$ 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 $c \cdot g(x)$ 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.

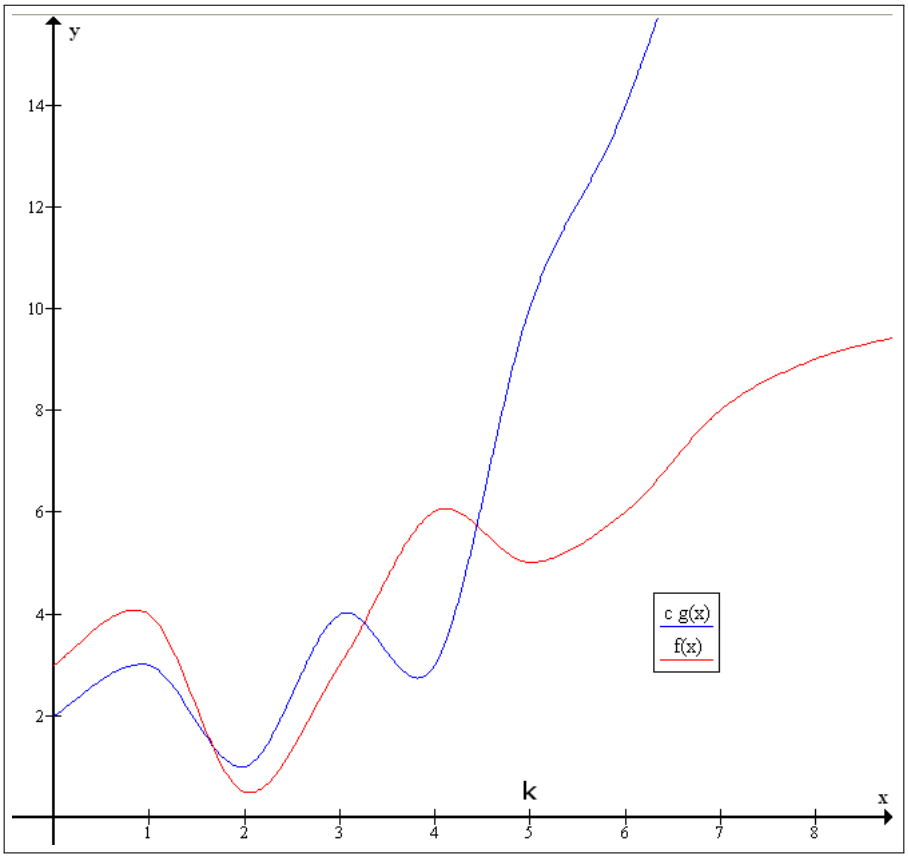

Figure 2.1: Example for $f \in \mathcal{O}(g)$.

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

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

for all $n \geq 1$ 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 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 $n^3 \in \mathcal{O}(2^n)$.

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 of the Big $\mathcal{O}$ Notation) 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

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

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

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

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

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

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

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

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

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

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

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

Proposition 4 (Stability of the Big $\mathcal{O}$ Notation under Addition) Assume that $f, g, h\colon \mathbb{N} \to \mathbb{R}_+$. Then we have

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

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

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

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

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

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

holds. Adding these inequalities we conclude that

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

holds for all $n \geq k$. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $f, g \colon \mathbb{N} \to \mathbb{R}_+$. Furthermore, assume that the limit

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

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

Proof: Define

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

Since the limit exists by our assumption, we know that

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

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

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

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

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

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

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

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:

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

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:

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

$$= \lim_{x\to\infty} \frac{\frac{\mathrm{d}^k\, x^k}{\mathrm{d}x^k}}{\frac{\mathrm{d}^k\, \lambda^x}{\mathrm{d}x^k}} = \lim_{x\to\infty} \frac{k!}{\ln(\lambda)^k \lambda^x} = \frac{k!}{\ln(\lambda)^k} \cdot \lim_{x\to\infty} \frac{1}{\lambda^x} = 0 \quad \text{because } \lambda > 0.$$

[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.

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

Remark: This example shows that any polynomial grows slower than any exponential function with a base greater than 1. ◇

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 have to 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:

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

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 $\bigl(\log_2(n)\bigr)^2 \in \mathcal{O}(n)$.

(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

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

(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

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

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

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

(h) Prove that $n^{\ln(n)} \in \mathcal{O}\bigl(b^n\bigr)$ 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 of functions. Therefore, for a given function $f : \mathbb{N} \to \mathbb{R}_+$ we can either have

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

but 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 $\texttt{abs}(f - g) \in \mathcal{O}(h)$. 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, $\texttt{power}(m, n)$ computes m^n. The basic idea is to compute the value of m^n according to the formula

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

```
1  def power(m, n):
2      r = 1
3      for i in range(n):
4          r = r * m
5      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 $5\,//\,2 = 2$. In general, if $a, b \in \mathbb{N}$ and $b > 0$, it can be shown that there exist natural numbers q and r such that

$$a = b \cdot q + r \quad \text{where } 0 \leq r < b. \tag{ID}$$

Furthermore, q and r are uniquely determined by the condition (ID). Then, q is called the integer division of a by b and r is called the remainder of the integer division of a by b and we write

$$q = a\,//\,b \quad \text{and} \quad r = a\,\%\,b.$$

```
1  def power(m, n):
2      if n == 0:
3          return 1
4      p = power(m, n // 2)
5      if n % 2 == 0:
6          return p * p
7      else:
8          return p * p * m
```

Figure 2.3: Computation of m^n for $m, n \in \mathbb{N}$.

Next, we want to analyse the computational complexity of our implementation of `power`. To this end, let us compute the number of multiplications that are performed when `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

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

The reason is that then

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

and hence $n\,//\,2$ is of the same form as n and, in particular, is odd again. To see this, note that for $n = 2^l - 1$ we have that

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

showing that $(2^l - 1)\,//\,2 = 2^{l-1} - 1$. Therefore, if $n = 2^l - 1$ the exponent n will be odd on every recursive call until it finally reaches 0. Let us assume $n = 2^l - 1$ and let us compute the number a_n of multiplications that are done when `power`(m, n) is evaluated:

$$a_n := \text{number of multiplications to compute } \texttt{power}(m, n).$$

First, we have $a_0 = 0$, because if we have $n = 2^0 - 1 = 0$, then the evaluation of `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[3]:

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

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 the recurrence relation $b_l = b_{l-1} + 2$ with initial condition $b_0 = 0$ is given by the formula

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

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 `power`(m, n) needs the least number of multiplications if the test `n % 2 == 0` always evaluates as true. In this case, n has to be a power of 2. Hence there has to 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 `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 `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

[3] A recurrence relation for a sequence a_n is any equation that relates a_n to preceding terms of the sequence a_n. For example, $a_n = a_{n-1} + 1$ for $n > 0$ is a very simple recurrence relation, while $a_n = a_{n\,//\,2} + n$ for $n > 0$ is a more complicated recurrence relation. In order for a recurrence relation to define the sequence a_n we also need to specify one or more initial values.

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

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 needed to compute $\texttt{power}(m, n)$ 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 `div_mod` that takes two numbers m and n and that returns the pair

$$\big(m \mathbin{//} n, m \mathbin{\%} n\big).$$

Therefore, if $\texttt{div\_mod}(m, n) = (q, r)$ we will have both

$$m = n \cdot q + r \quad \text{and} \quad 0 \leq r < n.$$

You should implement the function recursively and compute $\texttt{div\_mod}(m, n)$ by recursively computing $\texttt{div\_mod}(m \mathbin{//} 2, n)$. In your implementation of `div_mod` you are only permitted to use the operators // and % when the second argument of these operators is 2, i.e. you are permitted to use the expressions $m \mathbin{//} 2$ and $m \mathbin{\%} 2$ as these expressions can be implemented using bit operations.

Exercise 8: Given a natural number n, the integer square root of n is the largest number r such that r^2 is less or equal than n, i.e. we have

$$\texttt{isqrt}(n) := \max\big(\{r \in \mathbb{N} \mid r^2 \leq n\}\big).$$

(a) Develop a recursive algorithm that computes $\texttt{isqrt}(n)$ in terms of $\texttt{isqrt}(n \mathbin{//} 4)$.

(b) Implement this algorithm in *Python*.

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 in that case 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

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

where $n \mathbin{//} \beta$ denotes integer division[4] of n by β.

Then we have the following:

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

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

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

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

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

The general case is similar, but is technically much more involved. Observe that the equation $n = \beta^k$ implies $k = \log_\beta(n)$. We will need this equation later. 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. Without them, the proof would be much more involved and the main idea of the proof would be hidden. 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

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

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:

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

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

   In this case, all terms in the series $\sum\limits_{j=0}^{k-1} \big(\frac{\alpha}{\gamma}\big)^j$ 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 and hence we have established the validity of the master theorem in all three cases. □

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

$$f(n) = f(n \,//\, 2) + 2.$$

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

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

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

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

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

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

Since this time we have

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

the first case of the master theorem tells us that

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

Exercise 9: For each of the following recurrence relations, use the master theorem to give estimates of the growth of the function f.

(a) $f(n) = 4 \cdot f(n \,//\, 2) + 2 \cdot n + 3.$

(b) $f(n) = 4 \cdot f(n \,//\, 2) + n^2.$

(c) $f(n) = 3 \cdot f(n \,//\, 2) + n^3.$ ◇

2.5 Variants of Big $\mathcal{O}$ Notation*

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

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

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

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

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

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.

2.6 Recurrence Relations

In some cases knowing the growth rate of a function is not sufficient. Rather, we want to be able to compute the growth rate exactly. If the growth rate to be computed results from the complexity analysis of a recursive algorithm, the growth rate is often defined via a recurrence relation. In the literature, *recurrence relations* are also known as difference equations. In this section we discuss a simple program whose complexity analysis leads to a recurrence relation for the number of additions. We will then see how this recurrence relation can be solved explicitly. The program that we will analyse is shown in Figure 2.4. The function `fib`(n) computes the Fibonacci number F_n. These numbers are defined inductively as follows:

1. $F_0 := 0$,
2. $F_1 := 1$,
3. $F_{n+2} = F_{n+1} + F_n$.

   This last equation is an example of a recurrence relation, while the first two equations are the initial conditions.

```
1  def fib(n):
2      if n <= 1:
3          return n
4      return fib(n-1) + fib(n-2)
```

Figure 2.4: A *Python* program to compute the Fibonacci numbers.

If we run the program shown in Figure 2.4 , we will find that the runtimes grow very rapidly with growing input parameter n. To analyse this phenomenon, we investigate the number of additions that are used in the calculation of `fib`(n) for a given $n \in \mathbb{N}$. If we call this number a_n, we find the following:

1. $a_0 = 0$,
2. $a_1 = 0$,
3. $n \geq 2 \to a_n = a_{n-1} + a_{n-2} + 1$,
   because in the recursive calls `fib`$(n-1)$ and `fib`$(n-2)$ we have, respectively, a_{n-1} and a_{n-2} additions and, furthermore, the expression `fib(n-1) + fib(n-2)` adds another addition.

We take the equation $a_n = a_{n-1} + a_{n-2} + 1$ and substitute $i+2$ for n. This yields

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

This type of equation is called a linear inhomogeneous recurrence relation. The expression $+1$ is called the inhomogeneity of this recurrence relation. The associated homogeneous recurrence relation results from dropping this term in equation (1) and has the form

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

We solve this equation with the following ansatz:

$$a_i = \lambda^i.$$

Substituting this ansatz into equation (2) leads to the equation

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

If we divide both sides of this equation by λ^i, we get the quadratic equation

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

which we solve by completing the square:

$$\begin{aligned}
& \lambda^2 = \lambda + 1 && \mid -\lambda \\
\Leftrightarrow\ & \lambda^2 - 2 \cdot \frac{1}{2} \cdot \lambda = 1 && \mid +\frac{1}{4} \\
\Leftrightarrow\ & \lambda^2 - 2 \cdot \frac{1}{2} \cdot \lambda + \left(\frac{1}{2}\right)^2 = \frac{5}{4} && \mid \text{completing the square} \\
\Leftrightarrow\ & \left(\lambda - \frac{1}{2}\right)^2 = \frac{5}{4} && \mid \sqrt{\ } \\
\Leftrightarrow\ & \lambda - \frac{1}{2} = \pm\frac{\sqrt{5}}{2} && \mid +\frac{1}{2} \\
\Leftrightarrow\ & \lambda_{1/2} = \frac{1}{2} \cdot (1 \pm \sqrt{5})
\end{aligned}$$

We note that any linear combination of the form

$$a_n = \alpha \cdot \lambda_1^n + \beta \cdot \lambda_2^n$$

is a solution to the homogeneous recurrence equation (2). We also note that the following identities hold for the values λ_1 and λ_2:

$$\lambda_1 - \lambda_2 = \sqrt{5} \quad \text{and} \quad \lambda_1 + \lambda_2 = 1. \tag{3}$$

These equations will be needed later. From the last equation it follows immediately that

$$1 - \lambda_1 = \lambda_2 \quad \text{and} \quad 1 - \lambda_2 = \lambda_1. \tag{4}$$

To solve the original recurrence equation (1) we try the ansatz $a_i = c$, where c is an unknown constant that is to be determined. This leads us to the equation

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

which has the solution $c = -1$. Hence

$$a_n = -1$$

is a solution of the inhomogeneous recurrence relation (1). We call this solution a special solution. Although this is a solution of the recurrence relation $a_{i+2} = a_{i+1} + a_i + 1$, it does not satisfy the initial conditions $a_0 = 0$ and $a_1 = 0$. The general solution of the recurrence equation (1) is the sum of the general solution of the homogeneous recurrence equation and the special solution and therefore reads

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

where $\lambda_1 = \frac{1}{2} \cdot (1 + \sqrt{5})$ and $\lambda_2 = \frac{1}{2} \cdot (1 - \sqrt{5})$. The coefficients α and β have to be determined so that the initial conditions $a_0 = 0$ and $a_1 = 0$ are met. This leads to the following system of linear equations:

$$\begin{aligned} 0 &= \alpha \cdot \lambda_1^0 + \beta \cdot \lambda_2^0 - 1 \\ 0 &= \alpha \cdot \lambda_1^1 + \beta \cdot \lambda_2^1 - 1 \end{aligned}$$

Let's add 1 to both equations and simplify the powers λ_i^0 to 1 for $i = 1, 2$ and λ_i^1 to λ_i. This results in the following system of linear equations:

$$\begin{aligned} 1 &= \alpha + \beta \\ 1 &= \alpha \cdot \lambda_1 + \beta \cdot \lambda_2 \end{aligned}$$

The first of these two equations yields

$$\alpha = 1 - \beta.$$

If we insert this value of α into the second equation, we get

$$\begin{aligned} & 1 && = && (1 - \beta) \cdot \lambda_1 + \beta \cdot \lambda_2 \\ \Leftrightarrow \quad & 1 && = && \lambda_1 + \beta \cdot (\lambda_2 - \lambda_1) \\ \Leftrightarrow \quad & 1 - \lambda_1 && = && \beta \cdot (\lambda_2 - \lambda_1) \\ \Leftrightarrow \quad & \frac{1 - \lambda_1}{\lambda_2 - \lambda_1} && = && \beta \end{aligned}$$

Because of $\alpha = 1 - \beta$ we find

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

If we use the equations $\lambda_1 - \lambda_2 = \sqrt{5}$ and $\lambda_1 + \lambda_2 = 1$ next, we arrive at

$$\alpha = \frac{\lambda_1}{\sqrt{5}}.$$

As we have

$$\beta = \frac{1 - \lambda_1}{\lambda_2 - \lambda_1}$$

we conclude that

$$\beta = -\frac{\lambda_2}{\sqrt{5}}.$$

Therefore the solution of our recurrence relation is:

$$a_i = \frac{1}{\sqrt{5}} \cdot \left(\lambda_1^{i+1} - \lambda_2^{i+1}\right) - 1 \quad \text{for all } i \in \mathbb{N}.$$

Because of $\lambda_1 \approx 1.61803$ and $\lambda_2 \approx -0.61803$ the first term of this sum dominates the second term and the number of additions increases exponentially with the factor λ_1. This explains the strong increase in computing time.

Remark: The number λ_1 is also called the golden ratio and plays an important role in geometry and art. ◇

2.7 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.

The book "Calculus of Finite Differences and Difference Equations" by Murray R. Spiegel [Spi71] covers difference equations in much more detail than I was able to cover in my lecture. Furthermore, this book contains a large number of both solved and unsolved exercises concerning difference equations.

2.8 Check Your Understanding

If you are able to answer the questions below confidently, then you should have mastered the concepts introduced in this chapter.

1. What is the difference between a program and an algorithm.
2. How is the notion $\mathcal{O}(g)$ defined?
3. How would you prove $\ln^2(n) \in \mathcal{O}\big(\sqrt{n}\big)$?.
4. How can we compute m^n efficiently?
5. Are you able to recite the master theorem?
6. Why is the master theorem useful?
7. Do you know how to apply the master theorem?
8. Which formula about an infinite series is essential in order to prove the master theorem?
9. How are the Fibonacci numbers defined?
10. Can you derive an explicit formula for the Fibonacci numbers?
11. Describe the method to solve linear recurrence relations!
12. Which special cases might occur when solving a linear recurrence relation?

Chapter 3

Proving the Correctness of an Algorithm

In this chapter we will show two different methods that can be used to prove that an algorithm is correct.

(a) The method of computational induction can be used to verify the correctness of an algorithm that is defined recursively.

(b) In order to establish the correctness of an algorithm that is defined iteratively we use symbolic execution.

3.1 Computational Induction

Figure 3.1 shows the definition of the function power(m, n) that computes the value m^n. We have already analyzed the computational complexity of this program in the last chapter. In this chapter we show how the correctness of this function can be verified.

```
1  def power(m, n):
2      if n == 0:
3          return 1
4      p = power(m, n // 2)
5      if n % 2 == 0:
6          return p * p
7      else:
8          return p * p * m
```

Figure 3.1: 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 function that is defined recursively. The method of computational induction consists of three steps:

1. The base case.

   In the base case we have to show that the function definition is correct in all those cases where 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 where 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.

3. The termination proof.

   In this final step we have to prove that the recursive definition of the function is well founded, i.e. we have to prove that the recursive invocations terminate.

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

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

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 $n \mathbin{//} 2 = k$. Hence we have:

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

   (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 induction step is finished. $\surd$

3. **Termination proof**: Every time the function `power` is invoked as $\texttt{power}(m, n)$ and $n > 0$, the recursive invocation has the form $\texttt{power}(m, n \mathbin{//} 2)$ and, since $n \mathbin{//} 2 < n$ for all $n > 0$, the second argument is decreased. As this argument is a natural number, it must eventually reach 0. But if the second argument of the function `power` is 0, the function terminates. $\surd$ □

Exercise 10: Prove that the function `div_mod` that is shown in Figure 3.2 satisfies the specification

$$\texttt{div\_mod}(m, n) = \langle q, r \rangle \rightarrow m = q \cdot n + r \wedge r < n. \qquad \diamond$$

```
1  def div_mod(m, n):
2      if m < n:
3          return 0, m
4      q, r = div_mod(m // 2, n)
5      if 2 * r + m % 2 < n:
6          return 2 * q, 2 * r + m % 2
7      else:
8          return 2 * q + 1, 2 * r + m % 2 - n
```

Figure 3.2: The function `div_mod`.

3.2 Symbolic Execution

In the last chapter we have seen how to prove the correctness of a recursive function via computational induction. If a function is implemented via loops instead of recursion, then the method of computational induction is not applicable. Therefore, this section introduces the method of symbolic execution. Using this method it is possible to verify the correctness of programs that are implemented in an iterative fashion using loops. We will introduce this method via a simple example. Consider the program shown in Figure 3.3.

```
1  def power(x_1, y_1):
2      r_1 = 1
3      while y_n > 0:
4          if y_n % 2 == 1:
5              r_{n+1} = r_n * x_n
6          x_{n+1} = x_n * x_n
7          y_{n+1} = y_n \ 2
8      return r_N
```

Figure 3.3: An annotated programm to compute powers.

The main difference between a mathematical formula and a program is that in a formula all occurrences of a variable refer to the same value. This is different in a program because the variables change their values dynamically. In order to deal with this property of program variables we have to be able to distinguish the different occurrences of a variable. To this end, we index the program variables. When doing this we have to be aware of the fact that the same occurrence of a program variable can still denote different values if the variable occurs inside a loop. In this case we have to index the variables in a way that the index includes a counter that counts the number of loop iterations. For concreteness, consider the program shown in Figure 3.3. Here, in line 5 the variable `r` has the index n on the right side of the assignment, while it has the index r_{n+1} on the left side of the assignment in line 5. The index n denotes the number of times that the test y_n `> 0` of the `while` loop has been executed. After the loop finishes in line 10 the variable `r` is indexed as r_N, where N denotes the total number of times that the test `y > 0` has been executed. We show the correctness of the given program next. Let us define

$$a := x_1, \quad b := y_1.$$

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

$$r_n \cdot x_n^{y_n} = a^b. \tag{3.1}$$

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

B.C.: $n = 1$.

Since we have $r_1 = 1$, $x_1 = a$, and $y_1 = b$ we have

$$r_n \cdot x_n^{y_n} = r_1 \cdot x_1^{y_1} = 1 \cdot a^b = a^b.$$

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

We proof proceeds by a case distinction with respect to the expression $y \,\%\, 2$:

(a) $y_n \,\%\, 2 = 1$.

Then we have $y_n = 2 \cdot (y_n \,//\, 2) + 1$ and $r_{n+1} = r_n \cdot x_n$. Hence

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

(b) $y_n \,\%\, 2 = 0$.

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

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

This shows the validity of equation (3.1). If the `while` loop terminates, we must have $y_N = 0$. If $n = N$, then equation (3.1) yields:

$$\begin{aligned} & r_N \cdot x_N^{y_N} = a^b \\ \Leftrightarrow \;& r_N \cdot x_N^0 = a^b \\ \Leftrightarrow \;& r_N \cdot 1 = a^b \\ \Leftrightarrow \;& r_N = a^b \end{aligned}$$

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

Exercise 11: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 3.4 on page 34. During the proof you should make use of the fact that for all positive natural numbers x and y the equation

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

is valid.

```
1  def gcd(x, y):
2      while y != 0:
3          x, y = y, x % y
4      return x
```

Figure 3.4: The Euclidean algorithm.

3.3 Check Your Understanding

If you were able solve the exercises given in this chapter, then you should have mastered the concepts that have been introduced in this chapter.

Chapter 4

Sorting

In this chapter, we present various algorithms for solving the sorting problem. After a precise definition of the sorting problem, we start with a simple algorithm that is very easy to implement: insertion sort. However, the efficiency of this 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 considerably more involved. After that we show that any sorting algorithm that needs to compare the elements in order to sort them will perform $\Omega\big(n \cdot \log_2(n)\big)$ comparisons. The last algorithm presented is radix sort, which is an algorithm of linear complexity for sorting numbers of a fixed size. Finally, show how sorting is applied in the k-nearest-neighbour-algorithm.

4.1 The Sorting Problem

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. $\forall x \in S : x \leq x$, ($\leq$ is reflexive)
2. $\forall x, y \in S : \big(x \leq y \wedge y \leq x \rightarrow x = y\big)$, ($\leq$ is anti-symmetric)
3. $\forall x, y, z \in S : \big(x \leq y \wedge y \leq z \rightarrow x \leq z\big)$. ($\leq$ is transitive)

Definition 10 (Linear Order) 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. $\diamond$

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 two given employees $a, b \in P$

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

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

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

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

In the example 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 11 (Quasiorder)
A pair $\langle S, \preceq \rangle$ is a quasiorder iff $\preceq$ 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[1] $\approx$ on S by setting

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

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, i.e. we have:
   $$\forall i \in \{0, \cdots, \texttt{len}(S) - 2\} : S[i] \preceq S[i+1]$$
   Here, the length of the list S is denoted as $\texttt{len}(S)$ and $S[i]$ is the element at position i in S. We assume here that the first position is indexed with index 0.
   (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,
   $$\mathit{count}(x, L) := \texttt{card}\big(\{i \in \{0, \cdots, \texttt{len}(L) - 1\} \mid L[i] = x\}\big).$$
   Sometimes, this second requirement is changed as follows:
   $$\forall x \in S : \mathit{count}(x, S) \leq 1 \wedge \forall x \in M : \big(\mathit{count}(x, L) > 0 \leftrightarrow \mathit{count}(x, S) = 1\big).$$

[1] A pair $\langle M, \approx \rangle$ where M is a set and $\approx$ is a binary relation on M called an equivalence relation on M iff we have
(a) $\forall x \in M : x \approx x$, ($\approx$ is reflexive)
(b) $\forall x, y \in M : (x \approx y \rightarrow y \approx x)$, ($\approx$ is symmetric)
(c) $\forall x, y, z \in M : (x \approx y \wedge y \approx z \rightarrow x \approx z)$. ($\approx$ is transitive)

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 a 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 12: 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. ◇

4.2 Insertion Sort

Let us start our investigation of sorting algorithms with the algorithm insertion sort. We will describe this 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 can 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:

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

Figure 4.1 shows how the insertion-sort algorithm can be implemented in *Python*.

1. If `L` is empty, we return the empty list.

2. Otherwise, the assignment

   ```
   x, *R = L
   ```

   splits `L` into two parts: `x` is the first element of `L` and `R` is the rest of `L`, i.e. all elements of `L` with the exception of the first element. Next, the rest `R` is sorted and `x` is inserted into `R`.

3. In order to insert `x` into a list `L` that is already sorted, we first check whether `L` is the empty list. Inserting `x` into the empty list yields the list `[x]`.

4. Otherwise, `L` is split into two parts: `y` is the first element of `L` and `R` is the rest of `L`. Then, there are two cases:

```
1   def sort(L):
2       if L == []:
3           return []
4       x, *R = L
5       return insert(x, sort(R))
6
7   def insert(x, L):
8       if L == []:
9           return [x]
10      y, *R = L
11      if x <= y:
12          return [x] + L
13      else:
14          return [y] + insert(x, R)
```

Figure 4.1: Implementing insertion sort in *Python*.

(a) If `x` is less or equal than `y`, then, as `L` is sorted, `x` is less or equal than all elements of `L` and therefore `x` is prepended in front of `L`.

(b) Otherwise, we recursively insert `x` into the rest `R` and prepend `y` to the resulting list.

4.2.1 Complexity of Insertion Sort

We will compute the number of comparisons that are done in the implementation of `insert` in line 7 of Figure 4.1 in the worst case if we call `sort(L)` with a list `L` of length n. In order to do that, we have to compute the number of evaluations of the operator "`<=`" when `insert(x, L)` is evaluated for a list `L` of length n. Let us denote this number as a_n. The worst case happens if `x` is bigger than every element of `L` because in that case the test "`x <= y`" in line 11 of Figure 4.1 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

$$a_n = n.$$

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

Next, let us compute the number of comparisons that have to be done when calling `sort(L)` in the worst case for a list `L` of length n. Let us denote this number as b_n. The worst case happens if `L` is already sorted in reverse order, i.e. if `L` is sorted descendingly, because then the element `x` that is inserted in `sort(R)` is bigger than all elements of `R` and therefore also bigger than all elements of `sort(R)`. Then we have

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

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

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

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

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

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

because the sum of all natural numbers from 0 up to $n-1$ is given as

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

This last formula 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 has a quadratic complexity.

Next, let us consider the best case. The best case happens if the list L is already sorted ascendingly. Then, the call of `insert(x, sort(R))` needs a single comparison provided `R` is non-empty. 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 has a linear complexity. 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.

4.3 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 $\mathcal{O}\big(n \cdot \log_2(n)\big)$ 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, who was one of the most prominent mathematicians of the last century.

In order to sort a list L of length $n := \texttt{len}(L)$ 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 again is sorted.

Next, we need to specify how two sorted lists L_1 and L_2 are merged in a way that the resulting list is sorted.

1. If the list L_1 is empty, the result is L_2:
$$\texttt{merge}\big([], L_2\big) = L_2.$$
2. If the list L_2 is empty, the result is L_1:
$$\texttt{merge}\big(L_1, []\big) = L_1.$$
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 result of the comparison of x and y:
   (a) $x \preceq y$.
   In this case, we merge R_1 and L_2 and put x at the beginning of this list:
   $$x \preceq y \rightarrow \texttt{merge}\big([x] + R_1, [y] + R_2\big) = [x] + \texttt{merge}\big(R_1, [y] + R_2\big).$$
   (b) $\neg x \preceq y$.
   Now we merge L_1 and R_2 and put y at the beginning of this list:
   $$\neg x \preceq y \rightarrow \texttt{merge}\big([x] + R_1, [y] + R_2\big) = [y] + \texttt{merge}\big([x] + R_1, R_2\big).$$

Figure 4.2 shows how these equations can be implemented as a *Python* program.

```
1  def sort(L):
2      n = len(L)
3      if n < 2:
4          return L
5      L1, L2 = L[:n//2], L[n//2:]
6      return merge(sort(L1), sort(L2))
7
8  def merge(L1, L2):
9      if L1 == []:
10         return L2
11     if L2 == []:
12         return L1
13     x1, *R1 = L1
14     x2, *R2 = L2
15     if x1 <= x2:
16         return [x1] + merge(R1, L2)
17     else:
18         return [x2] + merge(L1, R2)
```

Figure 4.2: The merge sort algorithm implemented in *Python*.

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.
4. The implementation of the function `merge` is a straightforward translation of the equations given above.

4.3.1 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

$$\texttt{cmpCount} : \mathit{List}(M) \times \mathit{List}(M) \to \mathbb{N}$$

such that, given two lists L_1 and L_2 of elements from 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

$$\texttt{cmpCount}(L_1, L_2) \leq \texttt{len}(L_1) + \texttt{len}(L_2).$$

The proof is done by computational induction w.r.t. the definition of $\texttt{merge}(L_1, L_2)$. Hence there are four cases:

1. $L_1 = []$: Then we have
$$\texttt{cmpCount}(L_1, L_2) = \texttt{cmpCount}([], L_2) = 0 \leq \texttt{len}(L_1) + \texttt{len}(L_2).$$
2. $L_2 = []$: This case is analogous to the first case.
In the remaining cases we have
$$L_1 = [x_1] + R_1 \quad \text{and} \quad L_2 = [x_2] + R_2.$$
3. $x_1 \leq x_2$: Then we have
$$\begin{array}{lcl} \texttt{cmpCount}(L_1, L_2) & = & 1 + \texttt{cmpCount}(R_1, L_2) \\ & \overset{ih}{\leq} & 1 + \texttt{len}(R_1) + \texttt{len}(L_2) \\ & = & \texttt{len}(L_1) + \texttt{len}(L_2), \end{array}$$
where we have used the fact that
$$\texttt{cmpCount}(R_1, L_2) \leq \texttt{len}(R_1) + \texttt{len}(L_2)$$
holds by induction hypothesis.
4. $\neg(x_1 \leq x_2)$: This case is similar to the previous case. □

Exercise 13: 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

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

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

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

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 tells us 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 later, 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 less than $2^k + 2^k = 2^{k+1}$ 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

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

4.3.2 Implementing Merge Sort for Arrays

All the implementations of the *Python* programs presented up to now are quite inefficient. The reason is that, in *Python*, 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 have to take advantage of the fact that lists are represented as arrays. Figure 4.3 on page 43 presents an implementation of merge sort that treats the list `L` that is to be sorted as an array.

```
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     A[start:end] = L[start:end]
15     idx1 = start
16     idx2 = middle
17     i    = start
18     while idx1 < middle and idx2 < end:
19         if A[idx1] <= A[idx2]:
20             L[i]  = A[idx1]
21             idx1 += 1
22         else:
23             L[i]  = A[idx2]
24             idx2 += 1
25         i += 1
26     if idx1 < middle:
27         L[i:end] = A[idx1:middle]
28     if idx2 < end:
29         L[i:end] = A[idx2:end]
```

Figure 4.3: An array based implementation of merge sort.

We discuss the implementation shown in Figure 4.3 line by line.

1. The list `L` is sorted in place. Hence, 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].
```

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 8 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]` and `L[middle:end]`.

These two lists have approximately the same size which is about half the size of the list `L`.

6. Next, the lists `L[start:middle]` and `L[middle:end]` are sorted recursively in line 9 and 10, respectively.

7. The call to `merge` in line 11 merges these lists.

8. The procedure `merge` defined in line 13 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],
```

while the second sublist is

```
L[middle:end].
```

(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]` and `L[middle:end]`

are already sorted. The merging of these sublists works as follows:

(a) First, line 14 copies these 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 19 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 statement in line 27 covers the case that the second sublist is exhausted before the first sublist, while the statement in line 29 covers the case that the first sublist is exhausted before the second sublist.

4.3.3 An Iterative Implementation of Merge Sort

```
1  def sort(L):
2      A = L[:]  # A is a copy of L
3      mergeSort(L, A)
4
5  def mergeSort(L, A):
6      n = 1
7      while n < len(L):
8          k = 0
9          while n * (k + 1) + 1 <= len(L):
10             top = min(n * k + 2 * n, len(L))
11             merge(L, n * k, n * k + n, top, A)
12             k += 2
13         n *= 2
```

Figure 4.4: A non-recursive implementation of merge sort.

The implementation of merge sort shown in Figure 4.3 on page 43 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 can be less efficient than an iterative implementation. Therefore, we present an iterative implementation of merge sort in Figure 4.4 on page 45.

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:n*k+n]
```

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:n*k+n]` 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[0:n],
```

while the second sublist is given as

```
L[n: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 11 is the minimum of $n \cdot k + 2 \cdot n$ and `len(L)`.

4.4 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 the empty list:

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

2. Otherwise, L has the form $\texttt{L} = [\texttt{x}] + \texttt{R}$. 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 placing 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 4.5 on page 46 shows how these equations can be implemented in *Python*.

```
1  def sort(L):
2      if L == []:
3          return []
4      x, *R = L
5      S = [y for y in R if y <= x]
6      B = [y for y in R if y >  x]
7      return sort(S) + [x] + sort(B)
```

Figure 4.5: The quicksort algorithm.

4.4.1 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

$$\texttt{S} = [] \quad \text{and} \quad \texttt{B} = \texttt{R},$$

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 sort(B).

The initial condition is $a_1 = 0$, since we do not need any comparisons to sort a list containing only one element. Hence the recurrence relation can be solved as follows:

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

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, note that with quicksort the worst case occurs if we try to sort a list L that is already sorted. It is important to realize that this is a common case in practical applications of sorting. In order to avoid the worst case behaviour for lists that are either sorted or nearly sorted we use the element at position len(L) // 2 as the pivot element.

Average Complexity

By this time you probably wonder why the algorithm has been called quicksort since, in the worst case, it is much slower than merge sort. To understand what is really going on, we define

d_n := average number of comparisons to sort a list L of n 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 len(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 $i =$ len(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}:

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

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 it by the number $(n+1) \cdot (n+2)$ and are left with the equation

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

In order to simplify this equation, let us define

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

Then $d_n = (n+1) \cdot a_n$. Substituting this expression for d_n 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 we get

$$2 \cdot n = (\alpha + \beta) \cdot n + 2 \cdot \alpha + \beta. \tag{8}$$

Since this equation has to hold for $n = 0$ we see that

$$0 = 2 \cdot \alpha + \beta. \tag{9}$$

Subtracting this equation from equation (8) yields:

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

Setting $n = 1$ yields

$$2 = \alpha + \beta. \tag{10}$$

Taken together, the equations (9) and (10) are a system of equations that determine the values of the parameters α and β:

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

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, 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 is a telescoping recurrence equation. 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

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

The number γ in this formula 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.

4.4.2 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 4.6 on page 50 shows this implementation.

```
1  def sort(L):
2      quickSort(0, len(L) - 1, L)
3
4  def quickSort(start, end, L):
5      if end <= start:
6          return  # at most one element, nothing to do
7      m = partition(start, end, L)  # m is the split index
8      quickSort(start, m - 1, L)
9      quickSort(m + 1, end  , L)
10
11 def partition(start, end, L):
12     pivot = L[end]
13     left  = start - 1
14     for idx in range(start, end):
15         if L[idx] <= pivot:
16             left += 1
17             swap(left, idx, L)
18     swap(left + 1, end, L)
19     return left + 1
20
21 def swap(x, y, L):
22     L[x], L[y] = L[y], L[x]
```

Figure 4.6: 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 `start`, `end`, and `L`.

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

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

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

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

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

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

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

The implementation of the function `quickSort` is quite similar to the list based 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 `L[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 11. The `for` loop in line 14 satisfies the following invariants.

   (a) $\forall i \in \{\mathtt{start}, \cdots, \mathtt{left}\} : \mathtt{L}[i] \leq \mathtt{pivot}$.
   All elements in the subarray `L[start:left+1]` are less or equal than the pivot element.

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

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

   Figure 4.7 shows these invariants.

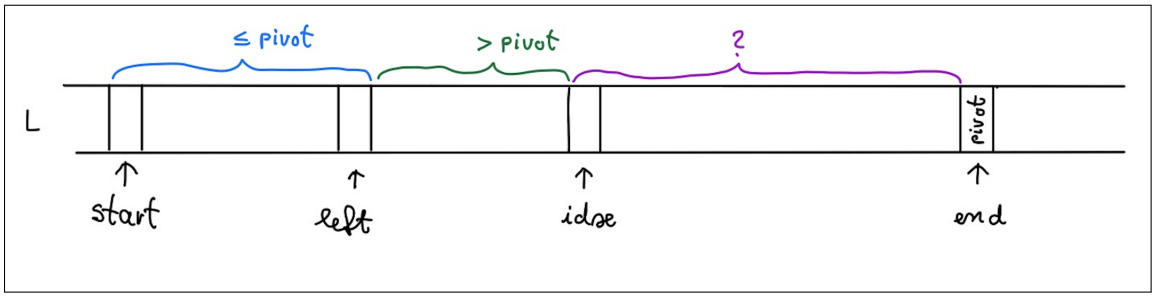

Figure 4.7: The invariants of the function `partition`.

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

$$\{\mathtt{start}, \cdots, \mathtt{left}\} = \{\mathtt{start}, \cdots, \mathtt{start}-1\} = \{\}$$

and

$$\{\mathtt{left}+1, \cdots, \mathtt{idx}-1\} = \{\mathtt{start}, \cdots, \mathtt{start}-1\} = \{\}.$$

(b) If the element `L[idx]` is less than the pivot element, it need to become part of the subarray $\mathtt{L}[\mathtt{start} : \mathtt{left}+1]$. In order to achieve this, it is placed at the position $\mathtt{L}[\mathtt{left}+1]$. The element that has been at that position is part of the subarray $\mathtt{L}[\mathtt{left}+1 : \mathtt{idx}]$ and therefore, most of the times,[2] it is greater than the pivot element. Hence we append this

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

element to the end of the subarray `L[left + 1 : idx]`. After incrementing the index `left`, both the placing of the element `L[idx]` at position `left + 1` and the appending of the element `L[left + 1]` to the end of the subarray `L[left + 1 : idx + 1]` is achieved by the statement

```
swap(left, idx, L).
```

Once the `for` loop in line 14 terminates, the call to `swap` in line 18 moves the pivot element into its correct position and returns the index where the pivot element has been placed.

4.4.3 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 the median of these elements, where the median of three elements is defined as the element that lies between the minimum and the maximum of these elements. In general, the median of a list L of length $2 \cdot n + 1$ is defined as the element x such that at least n elements of L are less or equal to x and at least n elements are bigger or equal than x.

   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 *Python* this is quite easy as the module `random` provides a predefined function `shuffle` that takes a list and shuffles it randomly in place. For example, the code

   ```
   L = list(range(10)); random.shuffle(L); print(L)
   ```

   might print the result

   ```
   [1, 9, 8, 5, 2, 0, 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 p_1 and p_2. For example, we can define

   $$x := \texttt{L}[0],\ y := \texttt{L}[-1], \quad \text{and then define} \quad p_1 := \min(x, y),\ p_2 := \max(x, y).$$

   Next, the list `L` is split into three parts:

   (a) The first part contains those elements that are less than p_1.

   (b) The second part contains those elements that are bigger or equal than p_1 but less or equal than p_2.

   (c) The third part contains those elements that are bigger than p_2.

   Figure 4.8 on page 53 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

```
def sort(L):
    if len(L) <= 1:
        return L
    x, y, *R   = L
    p1, p2     = min(x, y), max(x,y)
    L1, L2, L3 = partition(p1, p2, R)
    return sort(L1) + [p1] + sort(L2) + [p2] + sort(L3)

def partition(p1, p2, L):
    if L == []:
        return [], [], []
    x, *R      = L
    R1, R2, R3 = partition(p1, p2, R)
    if x < p1:
        return [x] + R1, R2, R3
    if x <= p2:
        return R1, [x] + R2, R3
    else:
        return R1, R2, [x] + R3
```

Figure 4.8: A list based implementation of dual pivot quicksort.

Exercise 14: Implement a version of dual pivot quicksort that is array-based instead of list-based. ◇

4.5 A Lower Bound for the Sorting Problem

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 the list $[a_1, a_2]$ is sorted ascendingly.
2. If $a_2 < a_1$, then the list $[a_2, a_1]$ 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 choose between at most four different possibilities. Next, we generalize this observation.

Theorem 12 Given a list L of n different elements, there are

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \prod_{i=1}^{n} i$$

different permutations of L.

Proof: The claim is proven by induction on n.

B.C.: $n = 1$:

There is only 1 way to arrange one element in a list. As $1! = 1$ the claim is true 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 `True` or `False` 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 $2 \cdot 2 = 2^2$ different cases.
3. In general, an easy induction shows that k tests can choose from at most 2^k different cases.

The last claim can also 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 a lower bound for the expression $\log_2(n!)$. First, we have

$$\log_2(n!) = \log_2\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log_2(i)$$

The crucial idea is to interpret this sum as an upper Riemann sum of the integral

$$\int_1^n \log_2(x)\,\mathrm{d}x$$

as is shown in Figure 4.9[3].

Therefore we have the following chain of inequations:

[3] I have created this figure using the highly recommended tool geogebra.

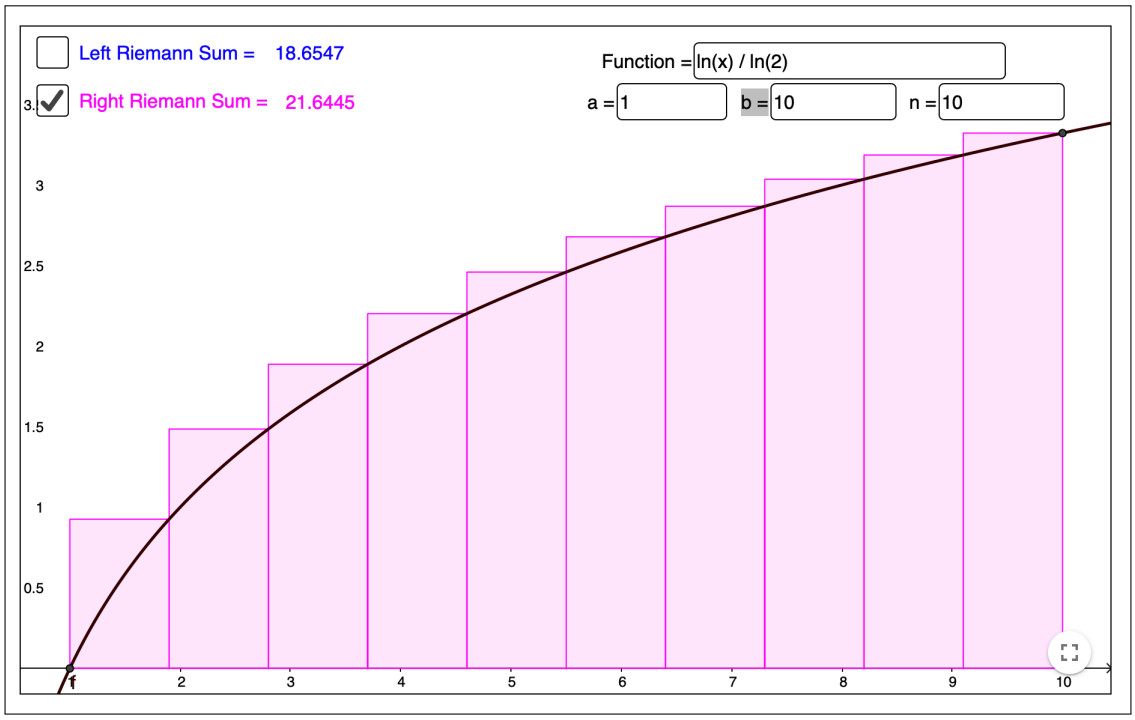

Figure 4.9: The upper sum for the integral $\int_1^{10} \log_2(x)\,\mathrm{d}x$.

$$
\begin{aligned}
k &\geq \log_2(n!) \\
&= \sum_{i=1}^{n} \log_2(i) \\
&\geq \int_1^n \log_2(x)\,\mathrm{d}x \\
&= \frac{1}{\ln(2)} \cdot \int_1^n \ln(x)\,\mathrm{d}x \\
&= \frac{1}{\ln(2)} \cdot \big[x \cdot \ln(x) - x\big]_1^n \\
&= \frac{1}{\ln(2)} \cdot \big(n \cdot \ln(n) - n + 1\big) \\
&= n \cdot \log_2(n) - \frac{n-1}{\ln(2)}
\end{aligned}
$$

This show that

$$k \geq n \cdot \log_2(n) - \frac{n-1}{\ln(2)}.$$

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 within a linear bound of size $(n-1)/\ln(2)$.

4.6 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 can not work by comparing the elements of the list that is to be sorted. This algorithm only works if the elements of the list that is to be sorted are natural numbers of a fixed size. The algorithm is called counting sort. We explain this algorithm via an example. Table 4.1 on page 56 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 4.2 on page 57 shows the table that has been sorted accordingly.

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 4.1: Students and their grades, sorted alphabetically.

We proceed to describe an algorithm that is capable of transforming Table 4.1 into Table 4.2. This 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 4.1 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. In this stage our goal is compute the indices of the sublists containing the different grades. For example, since there are 3 students with a grade of 1 and 1 is the smallest grade, we know that the students with grade 1 will be found in the sublist

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

`L[0:3]`. Similarly, as there are 8 students that have a grade of 2 and $3 + 8 = 11$, the sublist `L[3:11]` will contain the students with grade 2. All in all, we have the following:

(a) The sublist `L[0:3]` contains the students with grade 1.

(b) The sublist `L[3:11]` contains the students with grade 2.

(c) The sublist `L[11:17]` contains the students with grade 3.

(d) The sublist `L[17:20]` contains the students with grade 4.

(e) The sublist `L[20:22]` contains the losers.

The indexing stage computes the starting indices of these sublists. Of course, the first sublist has to start at the index 0. Since there are 3 students with a grade of 1, the second sublist starts at the index $0 + 3 = 3$. Since there are 8 students with a grade of 2, the third sublist starts at the index $0 + 3 + 8 = 11$. 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 4.10 on page 58 shows an implementation of counting sort. We proceed to discuss this algorithm line by line.

1. The procedure `countingSort` receives one argument. The list `Students` is a list of pairs of the form `(name, grade)` where

   (a) `name` is the name of a student, while

```
1  def countingSort(Students):
2      maxGrade =    255
3      Counts   =    [0] * (maxGrade+1)
4      Index    = [None] * (maxGrade+1)
5      Sorted   = [None] * len(Students)
6      # Phase 1: Counting
7      for _name, grade in Students:
8          Counts[grade] += 1
9      # Phase 2: Indexing
10     Index[0] = 0
11     for grade in range(maxGrade):
12         Index[grade+1] = Index[grade] + Counts[grade]
13     # Phase 3: Distribution
14     for name, grade in Students:
15         idx           = Index[grade]
16         Sorted[idx]   = (name, grade)
17         Index[grade] += 1
18     return Sorted
```

Figure 4.10: An implementation of counting sort.

(b) `grade` is the grade that this student has achieved.

The list `Students` is sorted alphabetically w.r.t. the names of the students. The purpose of the function `countingSort` is to sort this list according to their grades. Sublists of students with the same grade should still be sorted alphabetically.

2. In order for our function to generalize to arbitrary numbers we will assume that all grades are elements of the set $\{0, \cdots, 255\}$. Therefore we define `maxGrade` as 255. Of course, in the example discussed so far we know that the largest grade is 5. However, we will use the function `countingSort` later as an auxiliary function when implementing radix sort. Then the grades will be bytes and hence be natural numbers less or equal than 255.

3. Next, we initialize the auxiliary array `Count` to be an array of length `maxGrade` $+ 1$. 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 `Index` will contain the start indices of the different sublists. For a grade g, `Index`$[g]$ is the first index of the sublist containing those students that have achieved the grade g.

5. The list `Sorted` is the list that will be returned as the result. This list will contain pairs of the form `(name, grade)` where `name` is the name of a student and `grade` is her grade.

6. The `for`-loop in line 7-8 performs the counting stage. We iterate over all grades in the list `Students` and increment the counter `Counts`[`grade`] that is associated with the given grade.

7. Next, the index for the start of the sublist containing those students that have achieved the grade 0 is initialized as 0 in line 10.

8. Then, the `for`-loop in line 11-12 performs the indexing stage. As the number `Index`[`grade`] is the index of the start of the sublist for those students that have achieved the grade `grade` and the number of these students is `Counts`[`grade`], the sublist of the students with grade `grade` $+1$

has to start at index

$$\texttt{Index}[\texttt{grade}] + \texttt{Counts}[\texttt{grade}].$$

9. Finally, the `for`-loop in line 14-17 performs the distribution stage.
   (a) The `for`-loop iterates over all students.
   (b) We need to find where to put a student with grade `grade`. `Index[grade]` gives us the `index` of the next free entry in the result list `Sorted` corresponding for this grade.
   (c) In line 16 the students name and her grade are stored in the result lists `Sorted` at the index `idx`.
   (d) Finally, we need to increment the index stored at `Index[grade]` since we have just used this index and therefore the next student with the same grade needs to be stored at the subsequent location. This is done in line 17.

If the list `Students` has a length of n, then it is easy to see that counting sort has the complexity $\mathcal{O}(n)$. The first `for`-loop iterates over the list `Students` so its body is executed n times. The second for loop iterates 255 times and the last `for`-loop again iterates n times. Hence, counting sort is a linear sorting algorithm. Note that we do not compare grades in order to sort them.

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 to grade g before y and hence these sublists are still ordered alphabetically. This property is crucial for the development of our next sorting algorithm radix sort.

4.7 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: 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 defined via its four bytes $b_1, \cdots, b_4$ as follows:

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

Note that we 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 according to 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 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.

Figure 4.11 on page 60 shows an implementation of radix sort that implements this idea.

```
1  def extractByte(n, k):
2      return n >> (8 * (k-1)) & 0b11111111
3
4  def radixSort(L):
5      L = [(n, 0) for n in L]
6      for k in range(1, 4+1):
7          L = [(n, extractByte(n, k)) for (n, _) in L]
8          L = countingSort(L)
9      return [n for (n, _) in L]
```

Figure 4.11: An implementation of radix sort for sorting unsigned 32 integers.

1. The function `extractByte` is called with two arguments:

   (a) n is supposed to be an unsigned 32 bit number. Hence n is a integer satisfying $0 \leq n < 2^{32}$.

   (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`(n, k) extracts the k-th byte of the number n.

   `extractByte` works by shifting the number x by $(k-1) \cdot 8$ bits to the right using the shift operator "`>>`". This shift removes all bytes with index less than k. Then, the least significant byte of the remaining number is extracted using the bitwise and operator "`&`" with the mask 255. Note that 255 is written as 11111111_2 in binary and hence can be used to extract the eight least significant bits of a number.

2. `radixSort` takes a list of unsigned 32 bit integers L as its arguments.

3. This list is extended to a list of pairs where the first component of each pair is the number.

4. Then `radixSort` iterates over the four bytes of the numbers of the list L starting with the least significant byte.

5. In line 7 the k^{th} byte of the numbers of L is written into the second component of the pairs stored in the list.

6. With respect to `countingSort`, the elements of the list given to `countingSort` as its first argument are 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. Therefore in line 8 this list is sorted with respect to the k^{th} byte.

7. 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.

4.8 Application: Handwritten Digit Recognition

In the last section of this chapter we discuss an application of sorting: recognizing handwritten digits. We will use a training set of $50,000$ handwritten digits. Figure 4.12 shows the first 24 images of these handwritten digits. These images are given as grey scale images of 28×28 pixels. Each pixel p is a floating point number satisfying $0 \leq p \leq 1$. If $p = 1.0$, the pixel is completely black, while $p = 0.0$ if the pixel is white. Furthermore, for each of these images we also have been given a label $d \in \{0, 1, 2, \cdots, 9\}$, which specifies the digit that is represented in the image. Besides the training set there is also a test set of $10,000$ of handwritten images. Our task is to classify the images of this test set as digits, i.e. for each image of the test set we want to recognize the digit that is depicted.

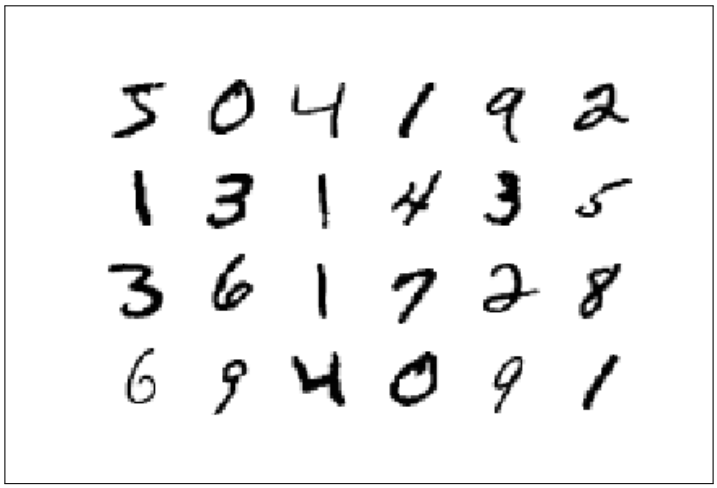

Figure 4.12: The first 24 digits of our dataset.

4.8.1 The k-Nearest Neighbour Algorithm

We will use the k-nearest neighbours algorithm to solve this task. Given an image x that is to be classified as a digit, we look at those images in the training set that are somehow close to x. If these images are all labelled with the same digit d, we conclude that x shows the digit d. In order to implement this algorithm, we have to specify what it means for an image $\mathbf{x}$ to be close to another image $\mathbf{y}$. As the images have a size of $28 \times 28 = 784$ pixels, they can be viewed as 784-dimensional vectors. We can compute the Euclidean distance $d(\mathbf{x}, \mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$ using the formula

$$d(\mathbf{x}, \mathbf{y}) := \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}\,.$$

This formula is a generalization of the distance between two points (x_1, x_2) and (y_1, y_2) in the the plane $\mathbb{R}^2$, which, according to the Pythagorean theorem, is given as

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}\,.$$

Given an image $\mathbf{y}$ that needs to be classified and a training set of n labelled images, the k-nearest neighbours algorithm works as follows:

(a) For every image $\mathbf{x}$ in the training set compute the Euclidean distance $d(\mathbf{x}, \mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$.

(b) Sort the images in the training set according to their distance to $\mathbf{y}$.

(c) Pick those k images that have the smallest distances to $\mathbf{y}$. These k images are called the k nearest neighbours.

(d) Among the k nearest neighbours, pick the label that occurs most frequently.

For example, if $k = 7$ and 3 of the 7 nearest neighbours are labelled as the digit 3, while 2 are labelled as the digit 2 and 2 are labelled as the digit 1, then we conclude that the image $\mathbf{y}$ shows the digit 3.

Figure 4.13 on page 62 shows a program that can be used to recognize a handwritten digit.

1. The images of the handwritten digits are stored in the file `mnist.pkl.gz`. This file is compressed using `gzip` and the images have been pickled using the module `pickle`. The module `pickle` supports the reading and writing of *Python* data structures and is therefore used to make data persistent, i.e. to store the data structures of a program in a file system. The counterpart of the function `load` is the function `dump`.

```
import gzip
import pickle
import numpy as np

def load_data():
    with gzip.open('mnist.pkl.gz', 'rb') as f:
        train, _, test = pickle.load(f, encoding="latin1")
    return (train[0], test[0], train[1], test[1])

X_train, X_test, Y_train, Y_test = load_data()

def distance(x, y):
    return np.sqrt(np.sum((x - y)**2))

def maxCount(L):
    Frequencies         = {}    # number of occurrences
    most_frequent       = L[0]  # most frequent digit so far
    most_frequent_count = 1
    for d in L:
        if d in Frequencies:
            Frequencies[d] += 1
        else:
            Frequencies[d]  = 1
        if Frequencies[d] > most_frequent_count:
            most_frequent       = d
            most_frequent_count = Frequencies[d]
    return most_frequent, most_frequent_count / len(L)

def digit(x, k):
    n          = X_train.shape[0]  # number of all training images
    Distances  = [(distance(X_train[i,:], x), i) for i in range(n)]
    Neighbours = [Y_train[i] for _, i in sorted(Distances)]
    return maxCount(Neighbours[:k])

digit(X_test[0,:], 13)
```

Figure 4.13: The k-nearest neighbours algorithm for digit recognition.

In order to read the images of the handwritten digits, we therefore have to import the modules `gzip` and `pickle`. The module `numpy` is needed as these images are stored as `numpy` arrays.

2. The function `load_data()` returns a tuple of the form

   $$(\texttt{X\_train}, \texttt{X\_test}, \texttt{Y\_train}, \texttt{Y\_test})$$

   where

   (a) `X_train` is a matrix storing the 50,000 training images of handwritten digits. For each $i \in \{0, \cdots, 49\,999\}$ the row `X_train`$[i,:]$ is an array of size 784 storing a single image.

   (b) `X_test` is a matrix containing 10,000 images of handwritten digits that we will use to test our implementation.

(c) `Y_train` is an array of size 50,000. For each $i \in \{0, \cdots, 49\,999\}$ the number `Y_train`$[i]$ specifies the digit shown in the ith training image.

(d) `Y_test` is an array of size 10,000. For each $i \in \{0, \cdots, 9\,999\}$ the number `Y_test`$[i]$ specifies the digit shown in the i^{th} test image.

The function `open` from the module `gzip` opens the specified file and automatically decompresses it. In the file `mnist.pkl.gz` the data is stored as a triple of pairs. For our purposes, we only need the first and the last component of this triple. Each of these components is a pair of the form (*data*, *label*), where *data* is an array of images and *labels* is an array specifying the digits represented in these images. The function `load_data` extracts the data stored in these pairs.

3. The function `distance`(x, y) computes the Euclidean distance between the images x and y. Given a vector x, the expression x`**`2 computes a vector containing the squares of the components of x. These squares can then be summed using the `numpy` function `sum`.

4. Given a list L of digits, the function `maxCounts`(L) returns a pair (d, p) where d is the digit that occurs most frequently in L and p is the fraction of occurrences of d in L. For example, we have

$$\texttt{maxCounts}([5, 2, 3, 5, 2, 5, 6, 5, 7, 8]) = (5, 0.4)$$

because the digit 5 is the most frequent digit in the list $[5, 2, 3, 5, 2, 5, 6, 5, 7, 8]$ and 40% of the digits in this list are fives. In detail, the function `maxCounts` works as follows:

(a) `Frequencies` is a dictionary that specifies how often a digit d occurs in the list L.

(b) `most_frequent` is the digit that so far is known to be the most frequent digit in L. Initially, this is assumed to be the first digit. As we iterate over the list L, this variable is updated.

(c) `most_frequent_count` is the number of occurrences of the digit `most_frequent`.

(d) As we iterate over the digits in L we update their frequencies. If we encounter a digit that is more common than the digit stored in the variable `most_frequent` we update this variable and the associated value `most_frequent_count`.

5. Given an image of a digit stored in the vector $\mathbf{x}$ and a number of neighbours k, the function `digit`$(\mathbf{x}, k)$ computes those k images in the training set `X_train` that are closest to the image $\mathbf{x}$, where closeness of images is defined in terms of the Euclidean distance of the vectors that store these images. From these k closest images of the training set the function chooses the digit that occurs most frequently. It returns a pair (d, p) where d is the digit that is most frequently occurring in the list of k neighbours and p is the percentage of images in the k neighbours of $\mathbf{x}$ that show the digit d. The implementation works as follows:

(a) `n` is the number of training examples. In our case, $n = 50,000$.

(b) `Distances` is a list of pairs of the form (d, i), where d is the distance of the i-th image in the training set from the given image x.

(c) These pairs are sorted with respect to their distance and the labels corresponding to the images are computed. Therefore, `Neighbours` is a list of the labels of all $50,000$ training images sorted according to their distance to the given image x.

(d) Finally, the function `maxCounts` takes the k closest images and computes the digit that is most frequently occurring.

6. The last line shows how the function `digit` can be used to classify the first image from the test set using the 13 closest neighbours.

Exercise 15: In the program shown in Figure 4.13 on page 62 we have sorted the list `Distances` in line 32. The only reason to do this was to be able to find the k smallest distances in this list. As k is a small number and the list `Distances` is rather large, this seems wasteful.

(a) Devise an algorithm Quickselect that selects the k smallest elements from a list L.

Hint: Try to adapt the algorithm Quicksort so that instead of sorting the list it finds the k smallest elements.

(b) Analyse the average complexity of your algorithm.

Hint: The recurrence equation resulting from an efficient implementation of Quickselect is quite complicated. Instead of solving this recurrence equation it is sufficient if you are able to prove by induction on the length n of the list L that your algorithm uses at most $4 \cdot n$ comparisons in order to compute the k smallest elements. ◇

4.9 Check Your Understanding

If you are able to answer the questions below confidently, then you can assume that you have mastered the concepts introduced in this chapter.

1. How have we defined the concept of a linear order?
2. What type of order do we need if we have to sort a list?
3. Describe the algorithm insertion sort on an abstract mathematical level.
4. What is the complexity of insertion sort?
5. Is there a case where insertion sort has only a linear complexity?
6. Describe the algorithm selection sort on an abstract mathematical level.
7. What is the complexity of selection sort?
8. Compare the complexity of selection sort with the complexity of insertion sort.
9. Describe the algorithm merge sort on an abstract mathematical level.
10. Describe an array-based, non-recursive implementation of merge sort.
11. Describe the algorithm quicksort on an abstract mathematical level.
12. Describe an array-based implementation of quicksort.
13. Compare the complexity of merge sort with the complexity of quicksort.
14. Compare the memory requierements of merge sort and quicksort.
15. Describe dual pivot quicksort.
16. In which case is dual pivot quicksort much faster than quicksort?
17. How does counting sort work?
18. How does radix sort work?
19. What is the complexity of radix sort?
20. How does the k-nearest neighbours algorithm work?

Chapter 5

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 implementation details of concrete data structures. Therefore, this notion enables us to separate algorithms from the data structures used in these algorithms. This chapter is structured as follows:

(a) We start with the formal definition of an abstract data types.

(b) As an example of an abstract data types we introduce stacks.

(c) Then we show how abstract data types are supported in *Python* via classes.

(d) Next, we demonstrate how stacks can be used to evaluate arithmetic expressions. To this end we build an operator precedence parser.

(e) Finally, we discuss the benefits of abstract data types.

Abstract data types were proposed by Barbara Liskov[1] and Stephen Zilles in 1974 [LZ74]. Abstract data type are one of the two main ingredients of object oriented programming. They were first implemented in the programming language CLU. The other ingredient of object oriented programming is inheritance.

5.1 A Formal Definition of Abstract Data Types

We define an abstract data type $\mathcal{D}$ formally as a 5-tupel of the form

$$\mathcal{D} = \langle N, P, \mathit{Fs}, \mathit{Ts}, \mathit{Ax} \rangle,$$

where the meaning of the components is as follows:

1. N is a string. This string 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 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. The function symbols itself are strings.

4. *Ts* is a set of type specifications. For every function symbol $f \in \mathit{Fs}$ the set *Ts* contains a type specification 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:

[1] Barbara Liskov received the 2008 Turing Award.

(a) We can have predefined data types like, e.g. "`int`" or "`str`".

(b) Furthermore, $T_1, \cdots, T_n$ and S 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 the argument t_i has type T_i for all $i \in \{1, \cdots, n\}$. Furthermore, the result of the function f is of type S.

Additionally, we must have either $T_1 = N$ or $S = N$. Therefore, either the first argument of f has to be of type N or the result of f has to be of type N, where N is the name of the abstract data types $\mathcal{D}$. If we have $T_1 \neq N$ and, therefore, $S = N$, then f is called a constructor of the abstract data type N. Otherwise, f is called a method.

5. Ax is a set of mathematical formulas. These formulas 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.

5.2 The Abstract Data Type Stack

We proceed to discuss the ADT 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 cafeteria. Usually, the clean plates are placed on top of each other and only the plate on top is accessible. Formally, we define the abstract data type `Stack` as follows:

1. The name of this abstract data type is `Stack`.
2. The set of type parameters is $\{\texttt{Element}\}$.
3. The set of function symbols is
$$\{\texttt{stack}, \texttt{push}, \texttt{pop}, \texttt{top}, \texttt{isEmpty}\}.$$
4. The type specifications for these function symbols are given as follows:
   (a) $\texttt{stack} : \texttt{Stack}$
   The function `stack` takes no arguments and produces an empty stack. Therefore, this function is a constructor. Intuitively, the function call $\texttt{stack}()$ creates an empty stack.
   (b) $\texttt{push} : \texttt{Stack} \times \texttt{Element} \to \texttt{Stack}$
   The function call $\texttt{push}(S, x)$ puts the element x on top of the stack S. In the following, we will use object oriented notation and write $S.\texttt{push}(x)$ instead of $\texttt{push}(S, x)$.
   (c) $\texttt{pop} : \texttt{Stack} \to \texttt{Stack} \cup \{\Omega\}$
   The function call $S.\texttt{pop}()$ removes the topmost element from the stack S and returns the resulting stack. If the stack S is empty, the return value is Ω, i.e. the implementation of this function will raise an exception in this case.
   (d) $\texttt{top} : \texttt{Stack} \to \texttt{Element} \cup \{\Omega\}$
   The function call $S.\texttt{top}()$ returns the element that is on top of the stack S. The stack S is left unchanged. If S is empty, then the result is undefined.

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

The Boolean function call $S.\texttt{isEmpty}()$ checks whether the stack S is empty and returns `True` or `False`.

The behaviour of a stack is specified by the following axioms.

1. $\texttt{stack}().\texttt{top}() = \Omega$

   Here, Ω denotes the undefined value[2]. In *Python*, Ω is represented as `None`. The expression $\texttt{stack}()$ creates an empty stack. Therefore, the given axiom expresses the fact that there is no element on top of the empty stack.

2. $S.\texttt{push}(x).\texttt{top}() = x$

   If we have a stack S and push an element x on top of S, then the element on top of the resulting stack is, unsurprisingly, x.

3. $\texttt{stack}().\texttt{pop}() = \Omega$

   Trying to remove an element from the empty stack yields an undefined result.

4. $S.\texttt{push}(x).\texttt{pop}() = S$

   If we have a stack S, 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. $\texttt{stack}().\texttt{isEmpty}() = \texttt{true}$

   This axiom expresses the fact that the stack created by the function call $\texttt{stack}()$ is empty.

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

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

When contemplating the axioms given above, we can recognize some structure. If we denote the functions `stack` and `push` as generators, then the axioms specify the behaviour of the remaining functions on the stacks created by these 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, we can build a parser for arithmetic expressions.

5.3 Implementing Abstract Data Types in Python

In object oriented programming languages, abstract data types are conveniently implemented as classes. 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 *Python* there is no way to neatly capture the signatures of an abstract data type. Therefore, the implementation of an abstract data type in *Python* merely consists of a class. At this point we note that classes are discussed in depth in Chapter 9 of the *Python* tutorial. These lecture notes only describe the most basic concepts of *Python* classes, since this is sufficient for this lecture. Further details can also be found in the *Python* online reference.

Figure 5.1 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 5.1 this name is `Stack`.

[2] 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.

```
class Stack:
    def __init__(self):
        self.mStackElements = []

    def push(self, e):
        self.mStackElements.append(e)

    def pop(self):
        assert len(self.mStackElements) > 0, "popping empty stack"
        self.mStackElements = self.mStackElements[:-1]

    def top(self):
        assert len(self.mStackElements) > 0, "top of empty stack"
        return self.mStackElements[-1]

    def isEmpty(self):
        return self.mStackElements == []

    def copy(self):
        C = Stack()
        C.mStackElements = self.mStackElements[:]
        return C

    def __str__(self):
        C = self.copy()
        result = C._convert()
        dashes = "-" * len(result)
        return '\n'.join([dashes, result, dashes])

    def _convert(self):
        if self.isEmpty():
            return '|'
        t = self.top()
        self.pop()
        return self._convert() + ' ' + str(t) + ' |'

def createStack(L):
    S = Stack()
    n = len(L)
    for i in range(n):
        S.push(L[i])
        print(S)
    return S

createStack(range(10))
```

Figure 5.1: An array based implementation of the ADT *Stack* in *Python*.

2. In *Python*, the constructor of a class has the name `__init__`. All methods defined in a class receive the object as their first argument. The convention is to name this parameter `self`,

but this is not mandatory. The name `self` is similar to the keyword `this` in the programming language *Java*. However, technically the name `self` is not a keyword in *Python*.

The constructor receives an uninitialized object as its first argument and has the task to initialize the member variables of this object. The class `Stack` uses only one member variable: This member variable is called `mStackElements`. My convention is to always start member variables with the letter '`m`'. Another convention is to use the underscore character '`_`'. These conventions facilitates the distinction of member variables from local variables.

In order to create an object of class `Stack` we invoke the constructor as follows:

```
s = Stack()
```

This statement creates an uninitialized object (in other words: an empty object) of class `Stack` and then invokes the constructor `__init__` to initialize the member variable `mStackElements` as an empty list. Next, object created is assigned to the variable `s`.

3. Line 3 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 the member variable `mStackElements` from an object o we use the following expression:

   o`.mStackElements`

   The implementation of stacks shown in Figure 5.1 is based on storing the elements of the stack in a *Python* list. In *Python*, 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. We will see how to do this later.

4. The rest of the class definition contains a number of function definitions. Functions defined inside a class are called methods. These methods are available in the class `Stack`. For example,

   ```
   stack.push
   ```

   refers to the method `push` defined in line 5 and 6. Every object of class `Stack` has 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`)`

5. Line 5 starts the definition of the method `push`. This method is called with two arguments:

   (a) `self` refers to the `Stack` object.

   (b) 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`.

   When invoking the method `push`, we 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:

   s`.push(`e`)`

6. 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.

7. 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.

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

9. Line 19 defines the method `copy`. The purpose of this method is to create an exact copy of the given stack. To this end the method creates a new object C of class `Stack`. Then the member variable `mStackElements` of the object `self` that was used to invoke the method `copy` is copied into the member variable `mStackElements` of the object C.

   Note that in order to create a copy C a stack object S it is not sufficient to use the assignment statement

   $$C = S$$

   because after this statement C is merely a new reference to the stack object S. Hence changing C would also change S and vice versa. For example, the method call

   $$C.\texttt{pop}()$$

   would then also pop the stack S and similarly the statement

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

   would push x onto the stack C. Since this is usually not what we want, we have to invoke the method `copy` as

   $$C = S.\texttt{copy}()$$

   in order to create a copy of the stack S.

10. Line 24 defines the method `__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 `__str__` is invoked automatically to perform this conversion.

    In order to understand the implementation of `__str__` we execute the following statements:

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

    These statements 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 `__str__` and has the following form:

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

    Hence, the topmost element of the stack is printed last.

    The implementation of the method `__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 is empty, the result of `_convert` will be the string "`|`". Otherwise we get the top element t of the stack using the method `top()` and remove it using `pop()`. Next, the remaining stack is converted to a string recursively and finally the element t is appended to this string.

    The name of the method `_convert` starts with an underscore because `_convert` is a private method of the class `Stack`, i.e. it should not be used from outside of the class `Stack`: Only methods defined in the class `Stack` are permitted to use the method `_convert`. However, this restriction is not enforced by the *Python* interpreter.

    (b) The method `__str__` creates a line of dashes in line 27. This line has the same length as the string produced by `_convert`. The result of `_convert` is then decorated with these dashes.

11. The function `createStack`(L) converts a list L into a stack and returns the resulting `Stack` object.

You should note that we were able to implement the method `__str__` without knowing anything about the internal representation of the stack. In order to implement `__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.

5.4 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.

5.5 Check Your Understanding

If you are able to answer the questions below confidently, then you can assume that you have mastered the concepts introduced in this chapter.

1. Explain the conceptual idea that underlies *abstract data types*.
2. How have we defined the concept of an *abstract data type* formally?
3. Can you give the formal definition of the abstract data type *Stack*?
4. What are the main benefits of using abstract data types?

Chapter 6

Sets and Maps

During the first term we have seen how useful sets and dictionaries are. The abstract concept that is implemented as a dictionary in *Python* is known as a map in theoretical computer science. The main idea behind a map is to have a function that is defined on a finite set of keys that maps these keys to values. A typical example of a map is a telephone book. In that case, the keys are names, while the values are the telephone numbers associated with these names.

In this chapter we will see how sets and maps can be implemented efficiently. We confine our attention to the implementation of maps. Once we know how to implement a map, it is easy to see that a set S can be represented by a function f_S that satisfies

$$x \in S \Leftrightarrow f_S(x) \neq \Omega.$$

The rest of this chapter is organized as follows:

(a) We begin with the definition of the abstract data type Map.

Following this definition we present several different implementations of maps.

(b) 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 finding the value associated with a key is $\mathcal{O}\big(\log_2(n)\big)$, where n is the number of key-value pairs stored in the binary tree. Unfortunately, the worst case complexity is $\mathcal{O}(n)$.

(c) Next, we discuss balanced ordered trees. In the case of balanced ordered trees the complexity of finding the value associated with a given key is guaranteed to be logarithmic in the number of keys, i.e. it is $\mathcal{O}\big(\log_2(n)\big)$. There are many different kinds of balanced ordered trees. In this lecture notes we will discuss AVL trees and 2-3 trees.

(d) After that, we explore a data structure that is known as a trie. This kind of data structure can be used if the keys of the map are strings.

(e) Finally, we discuss hash tables. Hash tables provide another way to implement a map. Hash tables are in wide spread use. For example, in *Python* sets and dictionaries are implemented via hash tables. Therefore, every computer scientist should have a good understanding of their inner workings.

6.1 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 name of a person with their telephone number. 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 13 (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
$$\{\texttt{map}, \texttt{find}, \texttt{insert}, \texttt{delete}\}.$$
4. The signatures of these function symbols are as follows:
   (a) $\texttt{map} : Map$
   Calling $\texttt{map}()$ generates a new empty map. Here, an empty map is a map that does not store any keys.
   (b) $\texttt{find} : Map \times Key \to Value \cup \{\Omega\}$
   The function call
   $$m.\texttt{find}(k)$$
   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) $\texttt{insert} : Map \times Key \times Value \to Map$
   The function call
   $$m.\texttt{insert}(k, v)$$
   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) $\texttt{delete} : Map \times Key \to Map$
   The function call
   $$m.\texttt{delete}(k)$$
   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) $\texttt{map}().\texttt{find}(k) = \Omega$.
   Calling $\texttt{map}()$ 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) $m.\texttt{insert}(k, v).\texttt{find}(k) = v$.
   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) $k_1 \neq k_2 \to m.\texttt{insert}(k_1, v).\texttt{find}(k_2) = m.\texttt{find}(k_2)$.
   If a value v 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) $m.\texttt{delete}(k).\texttt{find}(k) = \Omega$.
If the key k is deleted, then afterwards we won't find this key anymore.

(e) $k_1 \neq k_2 \rightarrow m.\texttt{delete}(k_1).\texttt{find}(k_2) = m.\texttt{find}(k_2)$,
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 *Python* it is very easy to implement the abstract data type *Map*. We just have to realize that a map is essentially the same thing as a dictionary. Now if d is a dictionary storing the value v for a key k, then the expression

$$d[k]$$

returns the value v. On the other hand we can insert a value v for a key k by writing

$$d[k] = v$$

and in order to delete the value stored for a key k we can use the `del` statement as follows:

`del` $d[k]$.

Figure 6.1 presents an implementation of the class `Map` that proceeds along these lines.

```
1  class Map:
2      def __init__(self):
3          self.mDictionary = {}
4
5      def find(self, k):
6          return self.mDictionary[k]
7
8      def insert(self, k, v):
9          self.mDictionary[k] = v
10
11     def delete(self, k):
12         del self.mDictionary[k]
13
14     def __str__(self):
15         return str(self.mDictionary)
```

Figure 6.1: A trivial implementation of the abstract data type *Map* in *Python*.

Note that the methods `insert` and `delete` do not return an object of type `Map`. Rather, for efficiency reasons, the existing `Map` object is updated.

6.2 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. In the average case, the complexity of the method `find` is logarithmic in the number of entries.

2. In the worst case, the complexity of the method `find` is linear in the number of entries.

   Unfortunately, the worst case occurs when the keys are inserted in sorted order.

Definition 14 (Ordered Binary Trees)
Assume a set *Key* and a set *Value* are given. The set $\mathcal{B}$ of ordered binary trees is defined inductively as the set of terms that are constructed from the function symbols `Nil` and `Node`, where the signatures of these function symbols are given as follows:

$$\texttt{Nil}\colon\ \mathcal{B} \qquad \text{and} \qquad \texttt{Node}\colon\ \mathit{Key} \times \mathit{Value} \times \mathcal{B} \times \mathcal{B} \to \mathcal{B}.$$

1. `Nil` is an ordered binary tree.

   This tree is called the empty tree since it does not store any information.

2. $\texttt{Node}(k, v, l, r) \in \mathcal{B}$ 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 is an ordered binary tree.
   l is the left subtree of the tree $\texttt{Node}(k, v, l, r)$.

   (d) r is a ordered binary tree.
   r is the right subtree of the tree $\texttt{Node}(k, v, l, r)$.

   (e) All keys that occur in the left subtree l are smaller than k.

   (f) All keys that occur in the right subtree r are bigger than k.

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

We will depict ordered binary trees graphically as follows:

1. The empty tree `Nil` is either shown as a black dot • or it is not shown at all.

2. A binary tree of the form $\texttt{Node}(k, v, l, r)$ 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 to the south-west of the node, while the right subtree r is depicted to the south-east of 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 6.2 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 a 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:

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

The method `find` has the signature

$$\texttt{find} : \mathcal{B} \times \mathit{Key} \to \mathit{Value} \cup \{\Omega\}$$

and is specified as follows:

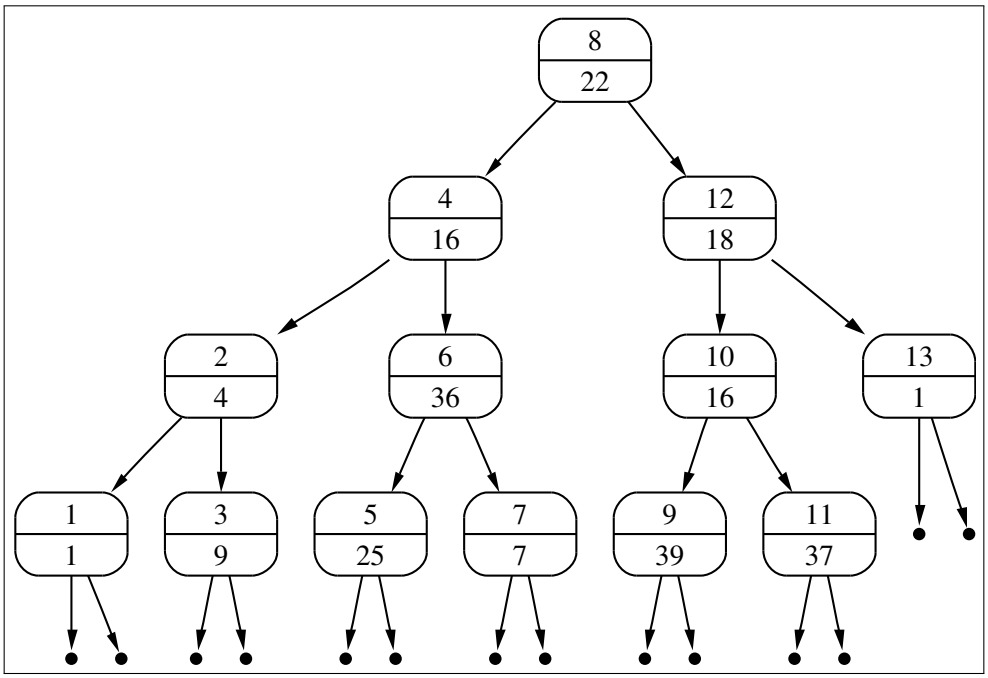

Figure 6.2: An ordered binary tree.

1. $\texttt{Nil.find}(k) = \Omega$,
   because the empty tree is interpreted as the empty map.
2. $\texttt{Node}(k, v, l, r).\texttt{find}(k) = v$,
   because the node $\texttt{Node}(k, v, l, r)$ stores the assignment $k \mapsto v$.
3. $k_1 < k_2 \rightarrow \texttt{Node}(k_2, v, l, r).\texttt{find}(k_1) = l.\texttt{find}(k_1)$,
   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. $k_1 > k_2 \rightarrow \texttt{Node}(k_2, v, l, r).\texttt{find}(k_1) = r.\texttt{find}(k_1)$,
   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 signature of `insert` is

$$\texttt{insert} : \mathcal{B} \times \mathit{Key} \times \mathit{Value} \rightarrow \mathcal{B}$$

and the definition of `insert` is given by the following equations.

1. $\texttt{Nil.insert}(k, v) = \texttt{Node}(k, v, \texttt{Nil}, \texttt{Nil})$,
   If the tree is empty, the information to be stored is placed at the root.
2. $\texttt{Node}(k, v_2, l, r).\texttt{insert}(k, v_1) = \texttt{Node}(k, v_1, l, r)$,
   If the key k is located at the root, we can just overwrite the old information.
3. $k_1 < k_2 \rightarrow \texttt{Node}(k_2, v_2, l, r).\texttt{insert}(k_1, v_1) = \texttt{Node}(k_2, v_2, l.\texttt{insert}(k_1, v_1), r)$,
   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. $k_1 > k_2 \rightarrow \texttt{Node}(k_2, v_2, l, r).\texttt{insert}(k_1, v_1) = \texttt{Node}(k_2, v_2, l, r.\texttt{insert}(k_1, v_1))$,
   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 signature is

$$\texttt{delete} : \mathcal{B} \times \mathit{Key} \to \mathcal{B}.$$

The specification of `delete` is more difficult than the specification of `find` and `insert`. If there is a tree of the form $t = \texttt{Node}(k, v, l, r)$ 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, $t.\texttt{delete}(k)$ can return the right subtree r, while if r is empty, $t.\texttt{delete}(k)$ can return the left subtree l. Things get more difficult when both l and r are non-empty. In this case, 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 $t = \texttt{Node}(k, v, l, r)$ and transform it into the node $t' = \texttt{Node}(k_{min}, v_{min}, l, r')$. 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 6.2, 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 6.3 on page 79.

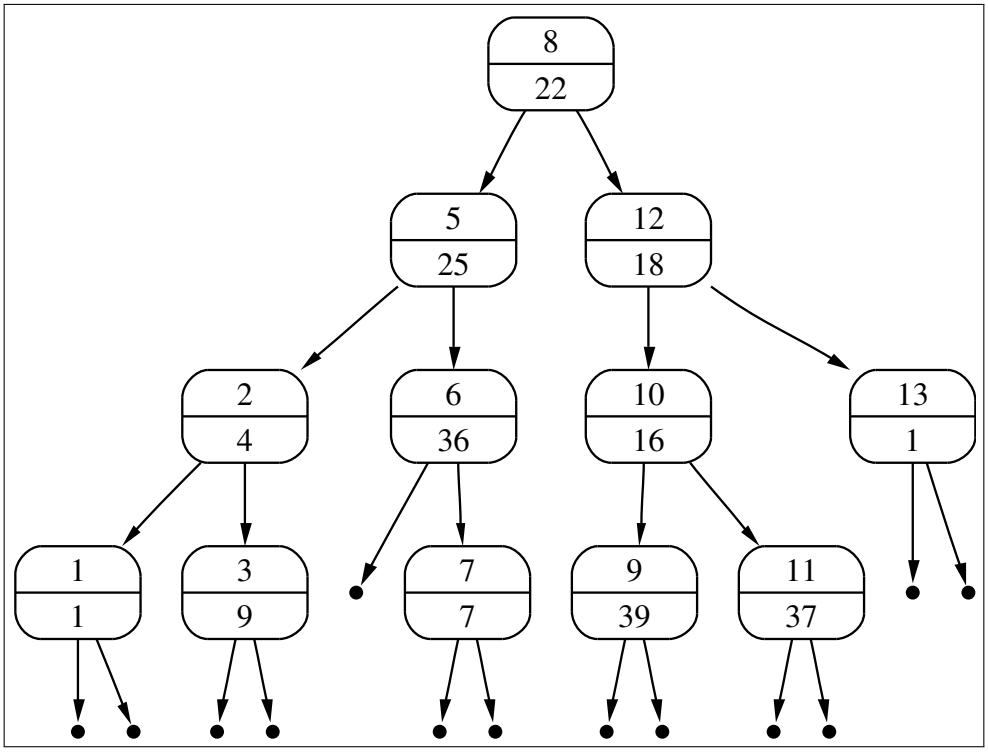

Figure 6.3: The ordered binary tree from Figure 6.2 after deleting the node with label $\langle 4, 16\rangle$.

Next, we specify the method `delMin`. The signature is

$$\texttt{delMin} : \mathcal{B} \to \mathcal{B} \times \mathit{Key} \times \mathit{Value}.$$

Hence, the call $t.\texttt{delMin}()$ returns a triple: If

$$t.\texttt{delMin}() = \langle r, k, v\rangle,$$

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. $\texttt{Node}(k, v, \texttt{Nil}, r).\texttt{delMin}() = \langle r, k, v \rangle$

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

2. $l \neq \texttt{Nil} \wedge l.\texttt{delMin}() = \langle l', k_{min}, v_{min} \rangle \rightarrow$

   $$\texttt{Node}(k, v, l, r).\texttt{delMin}() = \big\langle \texttt{Node}(k, v, l', r), k_{min}, v_{min} \big\rangle.$$

   If the left subtree l in the binary tree $t = \texttt{Node}(k, v, l, r)$ 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 $t' = \texttt{Node}(k, v, l', r)$.

Finally, we specify the method $\texttt{delete}()$.

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

   $$\texttt{Node}(k, v, l, r).\texttt{delete}(k) = \texttt{Node}(k_{min}, v_{min}, l, r').$$

   If the key k to be removed is found at the root of the tree and neither of its subtrees is empty, the call $r.\texttt{delMin}()$ 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. $k_1 < k_2 \rightarrow \texttt{Node}(k_2, v_2, l, r).\texttt{delete}\big(k_1\big) = \texttt{Node}(k_2, v_2, l.\texttt{delete}(k_1), r)$.

   If the key k_1 that is to be removed is less than the key k_2 stored at the root, the key k_1 can only be located in the left subtree l. Hence, k_1 is removed from the left subtree l recursively.

6. $k_1 > k_2 \rightarrow \texttt{Node}(k_2, v_2, l, r).\texttt{delete}(k_1) = \texttt{Node}(k_2, v_2, l, r.\texttt{delete}(k_1))$.

   If the key k_1 that is to be removed is greater than the key stored at the root, the key k_1 can only be located in the right subtree r. Hence, k_1 is removed from the right subtree r recursively.

6.2.1 Implementing Ordered Binary Trees in Python

Figure 6.4 and Figure 6.5 show how ordered binary trees can be implemented in *Python*. Objects of class `OrderedBinaryTree` represent ordered binary trees. We discuss the implementation of this class next.

1. The constructor is called without any argument. The expression

   ```
   OrderedBinaryTree()
   ```

   creates an empty tree that corresponds to `Nil`.

2. The class `OrderedBinaryTree` 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 `None`, which represents Ω.

   (b) `mValue` stores the value that is associated with `mKey`. For an empty node, `mValue` is `None`.

   (c) `mLeft` is the left subtree.

   (d) `mRight` is the right subtree.

3. The function `isEmpty` checks whether `self` represents an empty tree. The assumption is that if `mKey` is `None`, then the member variables `mValue`, `mLeft`, and `mRight` will also be `None`. Hence, in this case the object represents the empty tree `Nil`.
4. The implementation of `find` works as follows:
   (a) If the node is empty, there is no value to find and the function returns `None`.
   (b) If `key` == `mKey`, then the key we are looking for is stored at the root of this tree and hence the value we are looking for is `mValue`.
   (c) Otherwise, we have to compare the key `key`, which is the key we are looking for, with the key `mKey`, which is the key stored in this node. If `key` is less than `mKey`, the value associated with `key` can only be stored in the left subtree `mLeft`, while if `key` is greater than `mKey`, the value associated with `key` can only be stored in the right subtree `mRight`.

```
1  class OrderedBinaryTree:
2      def __init__(self):
3          self.mKey   = None
4          self.mValue = None
5          self.mLeft  = None
6          self.mRight = None
7
8      def isEmpty(self):
9          return self.mKey == None
10
11     def find(self, key):
12         if self.isEmpty():
13             return None
14         elif self.mKey == key:
15             return self.mValue
16         elif key < self.mKey:
17             return self.mLeft.find(key)
18         else:
19             return self.mRight.find(key)
20
21     def insert(self, key, value):
22         if self.isEmpty():
23             self.mKey   = key
24             self.mValue = value
25             self.mLeft  = OrderedBinaryTree()
26             self.mRight = OrderedBinaryTree()
27         elif self.mKey == key:
28             self.mValue = value
29         elif key < self.mKey:
30             self.mLeft.insert(key, value)
31         else:
32             self.mRight.insert(key, value)
```

Figure 6.4: Implementation of ordered binary trees in *Python*, 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 `key`, for which the value `value` 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 `value`. In this case, the current value of the variables `mValue` is overwritten with `value`.

(c) Otherwise, `key` is compared with `mKey` and the insertion is recursively continued in the appropriate subtree.

```
33      def delete(self, key):
34          if self.isEmpty():
35              return
36          if key == self.mKey:
37              if self.mLeft.isEmpty():
38                  self._update(self.mRight)
39              elif self.mRight.isEmpty():
40                  self._update(self.mLeft)
41              else:
42                  rs, km, vm = self.mRight._delMin()
43                  self.mKey   = km
44                  self.mValue = vm
45                  self.mRight = rs
46          elif key < self.mKey:
47              self.mLeft.delete(key)
48          else:
49              self.mRight.delete(key)
50
51      def _delMin(self):
52          if self.mLeft.isEmpty():
53              return self.mRight, self.mKey, self.mValue
54          else:
55              ls, km, vm = self.mLeft._delMin()
56              self.mLeft = ls
57              return self, km, vm
58
59      def _update(self, t):
60          self.mKey   = t.mKey
61          self.mValue = t.mValue
62          self.mLeft  = t.mLeft
63          self.mRight = t.mRight
```

Figure 6.5: Implementation of ordered binary trees in *Python*, part II.

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 59 show the implementation of the function `update`. When we delete the key at the root of the tree and either of the subtrees is empty, we would like to overwrite the current tree with the non-empty subtree, i.e. we would like to write something like

```
self = mLeft
```

However, we cannot replace the object `self` with another object. The only thing we can do is change the attributes of the object `self`. This is done in the method `update`. The expression `self.update(t)` overwrites the member variables of the object `self` with the corresponding member variables of the ordered binary tree `t`.

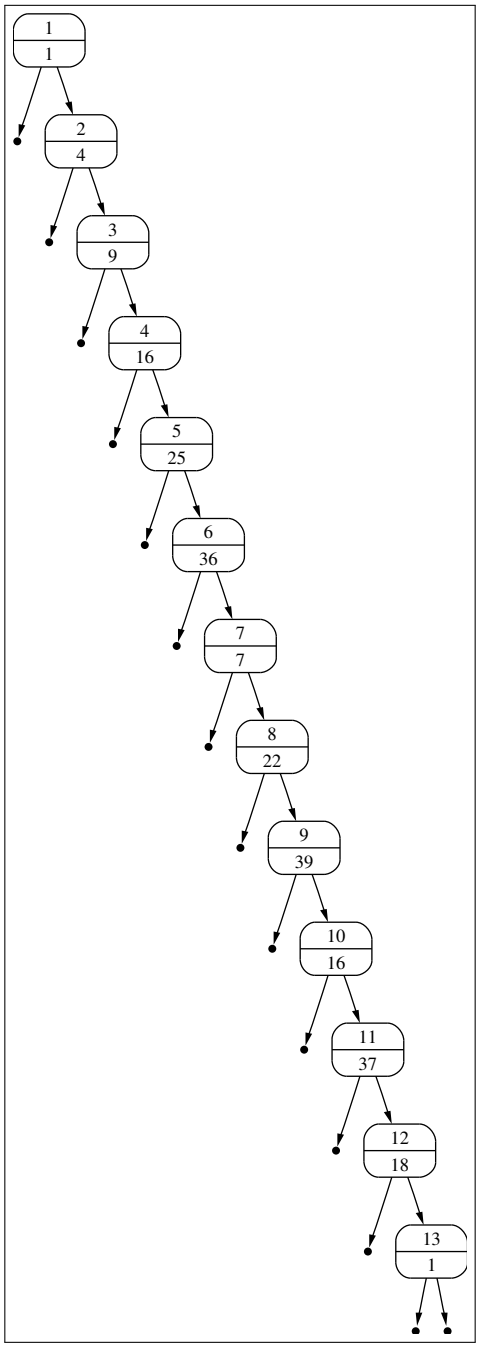

Figure 6.6: A degenerated binary tree.

6.2.2 Complexity Analysis

In this section we will first discuss the worst case complexity of ordered binary trees. Unfortunately, this complexity 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 containing n key-value pairs. After that, we investigate the average case complexity. We will show that the average case complexity is $\mathcal{O}\big(\ln(n)\big)$.

Worst Case Complexity

We begin our investigation of the complexity with an analysis of the complexity of $b.\texttt{find}(k)$ in the worst case. The worst case happens if the binary tree b degenerates into a list. Figure 6.6 on page 83 shows an ordered binary tree that is generated by inserting the keys 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 $b.\texttt{find}(k)$ 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 if the keys are inserted randomly[1]. 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 $b.\texttt{find}(k)$ 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 $b.\texttt{find}(k)$ 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 need exactly one key comparison. Next, imagine a binary tree b that contains $n+1$ keys. Then b can be written as

$$b = \texttt{Node}\big(\bar{k}, v, l, r\big),$$

where $\bar{k}$ 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 $\bar{k}$. If we have $\bar{k} = k_i$, then the left subtree of b contains i keys while the right subtree contains the remaining $n-i$ keys:

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

As b contains $n+1$ keys all together, there are $n+1$ different possibilities for the position of $\bar{k}$, 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 $b.\texttt{find}(k)$ provided the left subtree of b has i keys while b itself has $n+1$ keys as

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

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

[1] In practice, keys are not generated randomly. Therefore, in practice the worst case is not that unlikely to occur.

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

We proceed to compute $\texttt{numCmp}(i, n{+}1)$: 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 $\bar{k}$ 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 $\bar{k}$ 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
$$\frac{n-i}{n+1}.$$
In this case there are
$$d_{n-i} + 1$$
comparisons.

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

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

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

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

Here we have used the fact that the identity

$$\sum_{i=0}^{n} a_{n-i} = \sum_{i=0}^{n} a_i$$

is valid for any sequence $(a_n)_n$. We had verified this equation already when discussing the complexity of QuickSort in the average case. Next, we solve the recurrence equation

$$d_{n+1} = 1 + \frac{2}{(n+1)^2} \cdot \sum_{i=0}^{n} i \cdot d_i \tag{6.1}$$

with the initial condition $d_1 = 1$. In order to solve the equation (6.1) we perform the substitution $n \mapsto n+1$. This yields

$$d_{n+2} = 1 + \frac{2}{(n+2)^2} \cdot \sum_{i=0}^{n+1} i \cdot d_i \tag{6.2}$$

We multiply equation (6.1) with $(n+1)^2$ and equation (6.2) with $(n+2)^2$. We get

$$(n+1)^2 \cdot d_{n+1} = (n+1)^2 + 2 \cdot \sum_{i=0}^{n} i \cdot d_i, \tag{6.3}$$

$$(n+2)^2 \cdot d_{n+2} = (n+2)^2 + 2 \cdot \sum_{i=0}^{n+1} i \cdot d_i \tag{6.4}$$

We subtract equation (6.3) from equation (6.4) 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

$$c_{n+1} = c_n + \frac{2 \cdot n + 1}{(n+2) \cdot (n+1)} \quad \text{with the initial condition } c_1 = \frac{1}{2}.$$

A partial fraction decomposition shows that

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

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 shown that

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

Because of $d_n = \frac{n+1}{n} \cdot c_n$ this implies

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

This is our main result: On average, the operation b.`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`.

Exercise 16: Use ordered binary trees to implement a function `treeSort` that takes a list of numbers `L` as its input and returns a sorted list `S` that contains every element of `L` exactly once. If n is the length of `L`, the average complexity of your implementation should be $\mathcal{O}\big(n \cdot \log_2(n)\big)$. ◇

6.3 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 15 (Avl-Tree)
The set $\mathcal{A}$ of AVL trees is defined inductively:

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

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

In order to specify the methods find, insert, and delete for AVL trees we adapt the recursive equations that we have used to define these functions for 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. $|l.\texttt{height}() - r.\texttt{height}()| \leq 1 \rightarrow \texttt{Node}(k, v, l, r).\texttt{restore}() = \texttt{Node}(k, v, l, r)$.
   If the balancing condition is satisfied, then nothing needs to be done.
3. $$\begin{array}{ll} & l_1.\texttt{height}() = r_1.\texttt{height}() + 2 \\ \wedge & l_1 = \texttt{Node}(k_2, v_2, l_2, r_2) \\ \wedge & l_2.\texttt{height}() \geq r_2.\texttt{height}() \\ \rightarrow & \texttt{Node}(k_1, v_1, l_1, r_1).\texttt{restore}() = \texttt{Node}\big(k_2, v_2, l_2, \texttt{Node}(k_1, v_1, r_2, r_1)\big) \end{array}$$

The motivation for this equation can be found in Figure 6.7 on page 89. 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 6.7 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 6.7 the label below the horizontal line of each node shows the height of the tree corresponding to this node and not the value, because the values are irrelevant when it comes to rebalancing the trees. 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 either h or $h-1$.

The state shown in Figure 6.7 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.

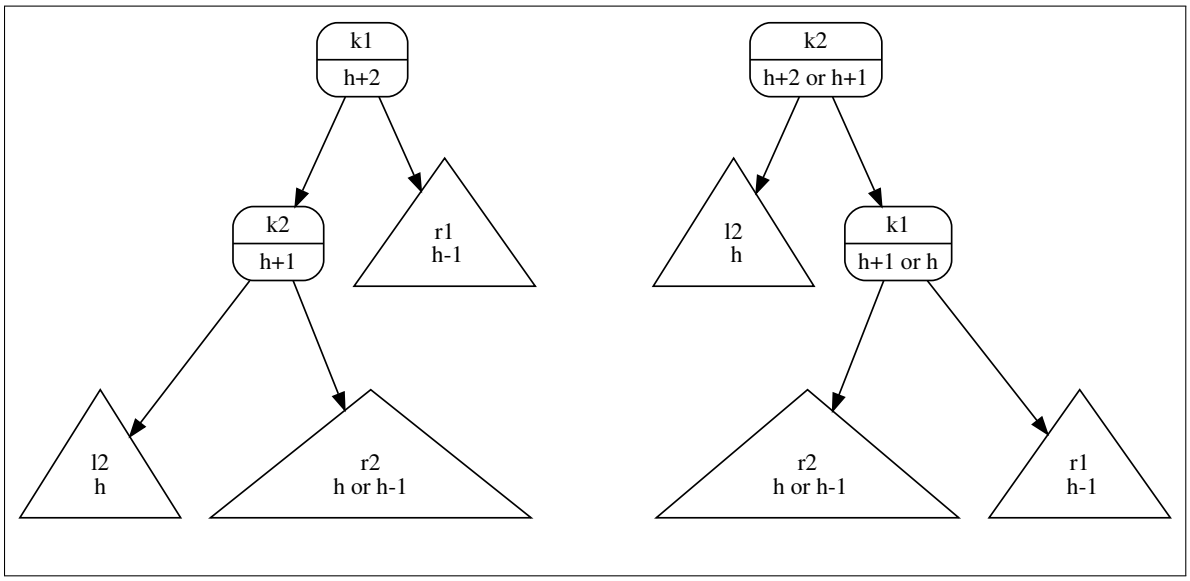

Figure 6.7: An unbalanced tree and the corresponding rebalanced tree.

We have to make sure that the tree shown in the right part of Figure 6.7 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 either h or $h-1$.

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 6.7 and in the right part of this figure. Hence, the tree shown in the right part of Figure 6.7 is ordered provided the tree in the left part is ordered to begin with.

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

The left hand side of this equation is shown in Figure 6.8 on page 90. 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 $h+1$.

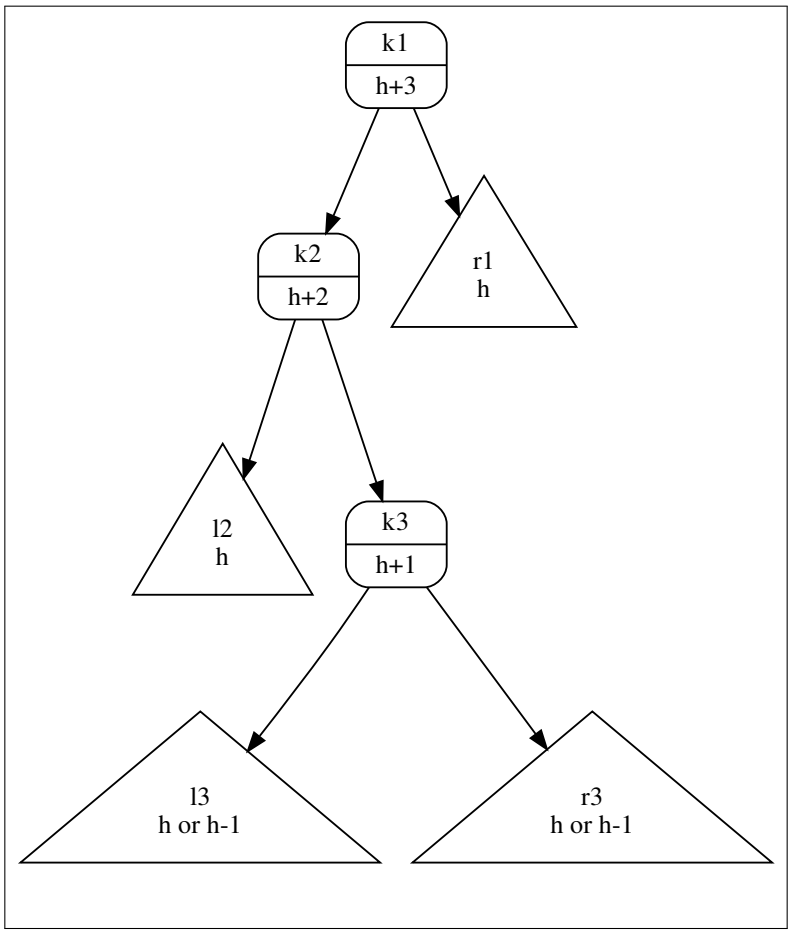

Figure 6.8: An unbalanced tree, second case.

Figure 6.9 on page 91 shows how the tree looks after rebalancing. The tree shown in this figure has the form

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

The inequation describing the ordering of the keys both in the tree in Figure 6.8 and the tree in Figure 6.9 is given as

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

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

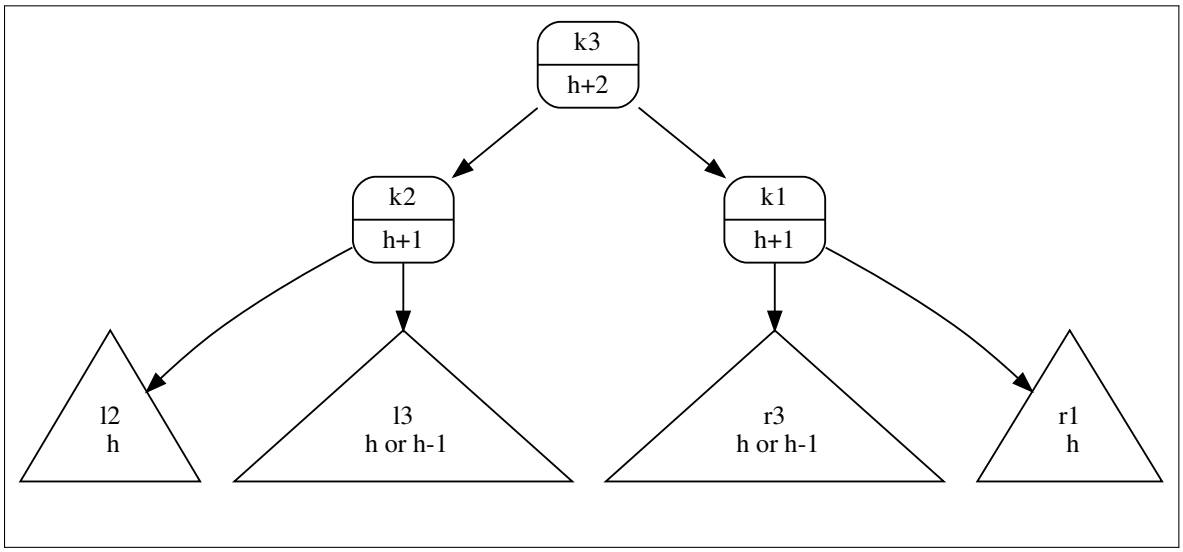

Figure 6.9: The rebalanced tree in the second case.

of the left subtree plus one. These two cases are completely analogous to the two cases discussed previously. The derivation of the corresponding equations is left as an exercise.

Exercise 17: Derive the two equations to compute $\texttt{Node}(k_1, v_1, l_1, r_1).\texttt{restore}()$ that have the precondition $r_1.\texttt{height}() = l_1.\texttt{height}() + 2$. You should start your derivation of these equations by drawing diagrams that are analogous to the diagrams shown Figure 6.7, Figure 6.8, and Figure 6.9. ⋄

Next, we 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. $k_1 > k_2 \rightarrow \texttt{Node}(k_2, v_2, l, r).\texttt{delete}(k_1) = \texttt{Node}\big(k_2, v_2, l, r.\texttt{delete}(k_1)\big).\texttt{restore}().$

6.3.1 Implementing AVL-Trees in Python

If we want to implement AVL-trees in *Python*, 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 6.10 shows the constructor of the class `AvlTree`. The variable `mHeight` is defined in line 7. It is initialised as 0 since the constructor `__init__` constructs an empty tree.

```
1  class AvlTree:
2      def __init__(self):
3          self.mKey    = None
4          self.mValue  = None
5          self.mLeft   = None
6          self.mRight  = None
7          self.mHeight = 0
```

Figure 6.10: Outline of the class `map`.

Figure 6.11 shows the implementation of the function `isEmpty`. A tree is empty iff its height is 0.

```
1  def isEmpty(self):
2      return self.mHeight == 0
```

Figure 6.11: Implementation of the method `isEmpyty`.

Figure 6.12 shows the implementation of the function `find`. Actually, this implementation is the same as the implementation that we have given earlier in Figure 6.4 for ordered binary trees. The reason is that every AVL tree is also an ordered binary tree and since searching for a key does not change the underlying tree there is no need to restore anything.

Figure 6.13 shows the implementation of the method `insert`. If we compare this implementation with the implementation for ordered 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 key-value pair 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 key-value pair into the right subtree `mRight`, we might have to rebalance the tree. This is done in line 15.

Figure 6.14 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.

Figure 6.15 shows the implementation of the method `delete` and the implementation of the auxiliary method `update`. Compared with Figure 6.5 there are only three differences:

```
def find(self, key):
    if self.isEmpty():
        return
    elif self.mKey == key:
        return self.mValue
    elif key < self.mKey:
        return self.mLeft.find(key)
    else:
        return self.mRight.find(key)
```

Figure 6.12: Implementation of the method `find`.

```
def insert(self, key, value):
    if self.isEmpty():
        self.mKey    = key
        self.mValue  = value
        self.mLeft   = AvlTree()
        self.mRight  = AvlTree()
        self.mHeight = 1
    elif self.mKey == key:
        self.mValue = value
    elif key < self.mKey:
        self.mLeft.insert(key, value)
        self._restore()
    else:
        self.mRight.insert(key, value)
        self._restore()
```

Figure 6.13: Implementation of the method `insert`.

```
def _delMin(self):
    if self.mLeft.isEmpty():
        return self.mRight, self.mKey, self.mValue
    else:
        ls, km, vm = self.mLeft._delMin()
        self.mLeft = ls
        self._restore()
        return self, km, vm
```

Figure 6.14: Implementation of `delMin`.

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 11.

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 14.

3. Similarly, if we delete a key in the right subtree, we have to restore the balancing condition. This is done in line 17.

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.

```
1   def delete(self, key):
2       if self.isEmpty():
3           return
4       if key == self.mKey:
5           if self.mLeft.isEmpty():
6               self._update(self.mRight)
7           elif self.mRight.isEmpty():
8               self._update(self.mLeft)
9           else:
10              self.mRight, self.mKey, self.mValue = self.mRight._delMin()
11              self._restore()
12      elif key < self.mKey:
13          self.mLeft.delete(key)
14          self._restore()
15      else:
16          self.mRight.delete(key)
17          self._restore()
18
19  def _update(self, t):
20      self.mKey    = t.mKey
21      self.mValue  = t.mValue
22      self.mLeft   = t.mLeft
23      self.mRight  = t.mRight
24      self.mHeight = t.mHeight
```

Figure 6.15: The methods `delete` and `update`.

Figure 6.16 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 the 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 31 and it recomputes `mHeight` according to the definition of the height of a binary tree.

```
1   def _restore(self):
2       if abs(self.mLeft.mHeight - self.mRight.mHeight) <= 1:
3           self._restoreHeight()
4           return
5       if self.mLeft.mHeight > self.mRight.mHeight:
6           k1,v1,l1,r1 = self.mKey,self.mValue,self.mLeft,self.mRight
7           k2,v2,l2,r2 = l1.mKey, l1.mValue, l1.mLeft, l1.mRight
8           if l2.mHeight >= r2.mHeight:
9               self._setValues(k2, v2, l2, createNode(k1,v1,r2,r1))
10          else:
11              k3,v3,l3,r3 = r2.mKey,r2.mValue,r2.mLeft,r2.mRight
12              self._setValues(k3, v3, createNode(k2, v2, l2, l3),
13                                      createNode(k1, v1, r3, r1))
14      elif self.mRight.mHeight > self.mLeft.mHeight:
15          k1,v1,l1,r1 = self.mKey,self.mValue,self.mLeft,self.mRight
16          k2,v2,l2,r2 = r1.mKey, r1.mValue, r1.mLeft, r1.mRight
17          if r2.mHeight >= l2.mHeight:
18              self._setValues(k2, v2, createNode(k1,v1,l1,l2), r2)
19          else:
20              k3,v3,l3,r3 = l2.mKey,l2.mValue,l2.mLeft,l2.mRight
21              self._setValues(k3, v3, createNode(k1, v1, l1, l3),
22                                      createNode(k2, v2, r3, r2))
23      self._restoreHeight()
24
25  def _setValues(self, k, v, l, r):
26      self.mKey   = k
27      self.mValue = v
28      self.mLeft  = l
29      self.mRight = r
30
31  def _restoreHeight(self):
32      self.mHeight = max(self.mLeft.mHeight, self.mRight.mHeight) + 1
```

Figure 6.16: The implementation of `restore` and `restoreHeight`.

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 5 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 6 and 7 so that these variable names correspond exactly to the variable names used in the Figures 6.7 and 6.8.

3. Next, the test in line 8 checks whether we have the case that is depicted in Figure 6.7. In this case, Figure 6.7 tells us that the key `k2` has to move to the root. The left subtree is now `l2`, while the right subtree is a new node that has the key `k1` at its root. This new node is created by the call of the function `createNode` in line 9. The function `createNode` is shown in Figure 6.17 on page 96.

4. If the test in line 8 fails, the right subtree is bigger than the left subtree and we are in the case that is depicted in Figure 6.8. We have to create the tree that is shown in Figure 6.9. 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 6.8. Next, we create the tree that is shown in Figure 6.9.

5. Line 14 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 5 to line 13, we won't discuss this case any further.

6. Finally, we recompute the variable mHeight since it is possible that the old value is no longer correct.

```
1   def createNode(key, value, left, right):
2       node         = AvlTree()
3       node.mKey    = key
4       node.mValue  = value
5       node.mLeft   = left
6       node.mRight  = right
7       node.mHeight = max(left.mHeight, right.mHeight) + 1
8       return node
```

Figure 6.17: Implementation of createNode.

The function createNode shown in Figure 6.17 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.

6.3.2 Analysis of the Complexity of AVL Trees

Next, we analyse the complexity of AVL trees in the worst case. In order to do this we have to know what the worst case actually looks like. Back when we only had ordered binary trees the worst case was the case where the tree had degenerated into a list. Now, the worst case is the case where the tree is as slim as it can possibly be while still satisfying the balancing condition 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:

(a) The height of $b_h(k)$ is h.

(b) The number of keys stored in $b_h(k)$ is minimal among all other AVL trees of height h.

(c) All keys stored in $b_h(k)$ are natural number that are bigger than k.

In order for the expression $b_h(k)$ to be unambiguously defined we demand that the keys in $b_h(k)$ are natural numbers that are as small as possible. Furthermore, as the values do not really matter, we define the values to be 0.

(d) If $h > 1$, then the height of the left subtree of $b_h(k)$ is bigger than the height of its right subtree.

For our investigation of the complexity, both the keys and the values do not really matter. The only problem is that we have to ensure that the tree $b_h(k)$ is an ordered tree. Before we can present the definition of $b_h(k)$ we need to define the auxiliary function maxKey(), which computes the largest key that is stored in a given tree. This function has the signature

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

Given an ordered binary tree b, the expression b.maxKey() returns the largest key that is stored in b. The expression b.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. $r \neq \texttt{Nil} \rightarrow \texttt{Node}(k, v, l, r).\texttt{maxKey}() = r.\texttt{maxKey}()$.

Now we are ready to define the tree $b_h(k)$ 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. $b_{h+1}(k).\texttt{maxKey}() = l \rightarrow b_{h+2}(k) = \texttt{Node}\big(l+1,\, 0,\, b_{h+1}(k),\, b_h(l+1)\big)$.

   In order to construct an AVL tree of height $h+2$ that contains the minimal number of keys 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 $b_{h+2}(k)$ is $b_{h+1}(k)$, while its 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 AVL 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 satisfy the balancing condition.

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

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

to be the number of keys in the tree $b_h(k)$. An easy induction on h shows 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. This is a linear, 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

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

To complete the square we add $\left(\frac{1}{2}\right)^2$ on both sides of this equation:

$$\lambda^2 - 2 \cdot \lambda \cdot \frac{1}{2} + \left(\frac{1}{2}\right)^2 = 1 + \frac{1}{4}.$$

This is equivalent to

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

From this we conclude

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

Let us therefore define

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

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

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

and therefore

$$\alpha = \frac{2 - \lambda_2}{\lambda_1 - \lambda_2}.$$

Now it just so happens that $\lambda_1 - \lambda_2 = \sqrt{5}$ and, furthermore, $2 - \lambda_2 = \lambda_1^2$. We prove only the second

claim, since the first is easy to verify. We have the following chain of equivalences:

$$\begin{aligned} & 2-\lambda_2 &=& \lambda_1^2 \\ \Leftrightarrow\ & 2-\lambda_2 &=& \lambda_1+1 \\ \Leftrightarrow\ & 1 &=& \lambda_1+\lambda_2 \\ \Leftrightarrow\ & 1 &=& \tfrac{1}{2}\cdot(1+\sqrt{5})+\tfrac{1}{2}\cdot(1-\sqrt{5}) \end{aligned}$$

Since the last equation is obviously true, the first is also true. 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:

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

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

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

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 yields the result

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

Above we have used the fact that

$$\log_2(n+1)-\log_2(n)\in\mathcal{O}(1).$$

As the height h is the maximal number of comparisons needed to find a given key the complexity of $b.\texttt{find}(k)$ for an AVL tree b is logarithmic even in the worst case. Figure 6.18 presents an AVL tree of height 6 where the number of keys is minimal.

6.3.3 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

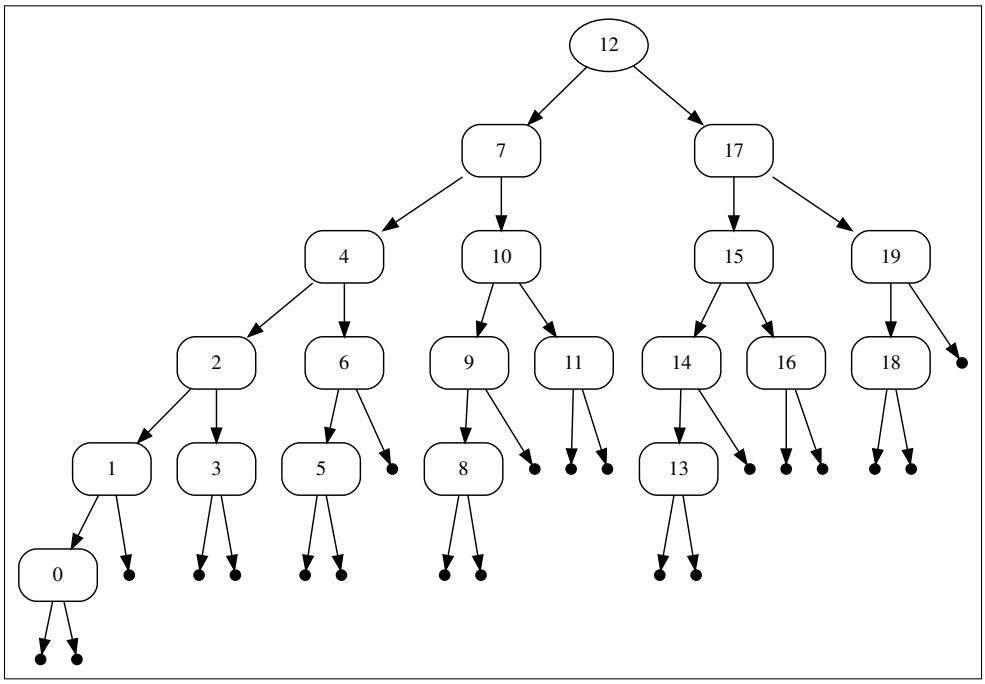

Figure 6.18: An AVL tree of height 6 that is as slim as possible. Values are not shown.

of a tree is defined. Red nodes do not contribute to the relaxed height of a tree. The left and right subtree of every node of a red-black tree are required to have the same relaxed height. A detailed and very readable exposition of red-black trees is given by Sedgewick [SW11b]. Red-black trees have been invented by Leonidas L. Guibas and Robert Sedgewick [GS78].

Exercise 18: 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. `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 `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 `Node`(k, v, l, r).
3. `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

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

where the right subtree is the 4-node $\texttt{Four}(\texttt{Nil}, 3, \texttt{Nil}, 4, \texttt{Nil}, 5, \texttt{Nil})$. 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 use three auxiliary functions to implement `insert`:

   (a) $t.\texttt{ins}(k)$ takes a 2-3 tree t and a key k and inserts the key k into t. It returns a 2-3-4 tree that has at most one 4-node. Unless the tree t has a height of 1, this 4-node has to be a child of the root node. The function `ins` is recursive and uses the function `restore` that is described next. Furthermore, the height of the tree $t.\texttt{ins}(k)$ has to be the same as the height of the tree t.

   (b) $t.\texttt{restore}()$ takes a 2-3-4 tree t that has at most one 4-node, which has to be a child of the root. It returns a 2-3-4 tree that has at most one 4-node. This 4-node has to be the root node. Furthermore, the height of the tree $t.\texttt{restore}()$ has to be the same as the height of the tree t.

(c) $t.\texttt{grow}()$ takes a 2-3-4 tree t that has at most one 4-node, which has to be the root node of the tree. It returns an equivalent 2-3 tree. The height of this tree is either the same as the height of t or it is the height of t plus 1.

Having defined these auxiliary functions, the function `insert` can then be computed as follows:

$$t.\texttt{insert}(k) = t.\texttt{ins}(k).\texttt{restore}().\texttt{grow}().$$

(c) Implement 2-3 trees in *Python*.

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

In order to implement the function `delete`, it is necessary to define 1-2-3 trees. In addition to both 2-nodes and 3-nodes, these trees also have 1-nodes. These nodes come into existence when a key is deleted from a 2-node: Deleting the key k from the node

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

creates the 1-node $\texttt{One}(\texttt{Nil})$.

Prof. Lyn Turbak has written a helpful paper describing 2-3 trees in more depth. This paper gives a graphical presentation of the `insert` and `delete` operations.

History: According to [CLRS09], 2-3 trees have been invented by John Hopcroft in 1970. John Hopcroft received the 1986 Turing Award.

6.4 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.
- `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

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

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

(a) $v \in \texttt{Value} \cup \{\Omega\}$,

(b) $Cs = [c_1, \cdots, c_n] \in \texttt{List}(\Sigma)$ is a list of different characters of length n and,

(c) $Ts = [t_1, \cdots, t_n] \in \texttt{List}(\mathbb{T})$ is a list of tries of the same length n,

then we have

$$\texttt{Trie}(v, Cs, Ts) \in \mathbb{T}.$$

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 Cs and Ts are both empty. Therefore, the empty trie has the form

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

In order to specify the semantics of a trie of the form $\texttt{Trie}(v, C, T)$ we specify a function

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

that takes a trie t and a string s as its arguments. The expression $t.\texttt{find}(s)$ returns the value that is associated with the key s in t. The value of the expression $t.\texttt{find}(s)$ is defined by induction on the length of the string s:

1. $\texttt{Trie}(v, Cs, Ts).\texttt{find}(\varepsilon) = v.$

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

2. $c = c_i \rightarrow \texttt{Trie}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\texttt{find}(cr) = t_i.\texttt{find}(r)$

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

3. $c \notin Cs \rightarrow \texttt{Trie}(v, Cs, Ts).\texttt{find}(cr) = \Omega$

   If c does not occur in the list Cs, then the trie $\texttt{Trie}(v, Cs, Ts)$ does not store a value for the key cr.

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

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

we draw a rectangle with rounded corners. This rectangle is split into two parts by a horizontal line. If the value v that is stored in this node is different from Ω, then v is written in the lower part of the rectangle. The label that we put in the upper half of the rectangle 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 rectangle that represents this node carries the label c_i in its upper half.

In order to clarify these ideas, Figure 6.19 on page 104 shows a trie mapping some strings to numbers. The mapping depicted in this tree can be written as a *Python* dictionary:

$$\big\{\texttt{'Stahl'} : 1, \texttt{'Stolz'} : 2, \texttt{'Stoeger'} : 3, \texttt{'Salz'} : 4, \texttt{'Schulz'} : 5,$$
$$\texttt{'Schulze'} : 6, \texttt{'Schnaut'} : 7, \texttt{'Schnupp'} : 8, \texttt{'Schroer'} : 9\big\}.$$

Since the node at the root has no parent, the upper half of the rectangle representing the root is always empty. In the example shown in Figure 6.19, the lower half of this rectangle 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

$$\texttt{Trie}(\Omega, [\texttt{'S'}], [t]).$$

Here, t denotes the trie that is labelled with the character `'S'` at its root. This trie can then be represented by the term

$$\texttt{Trie}(\Omega, [\texttt{'t'}, \texttt{'a'}, \texttt{'c'}], [t_1, t_2, t_3]).$$

This trie has three children that are labelled with the characters `'t'`, `'a'`, and `'c'`.

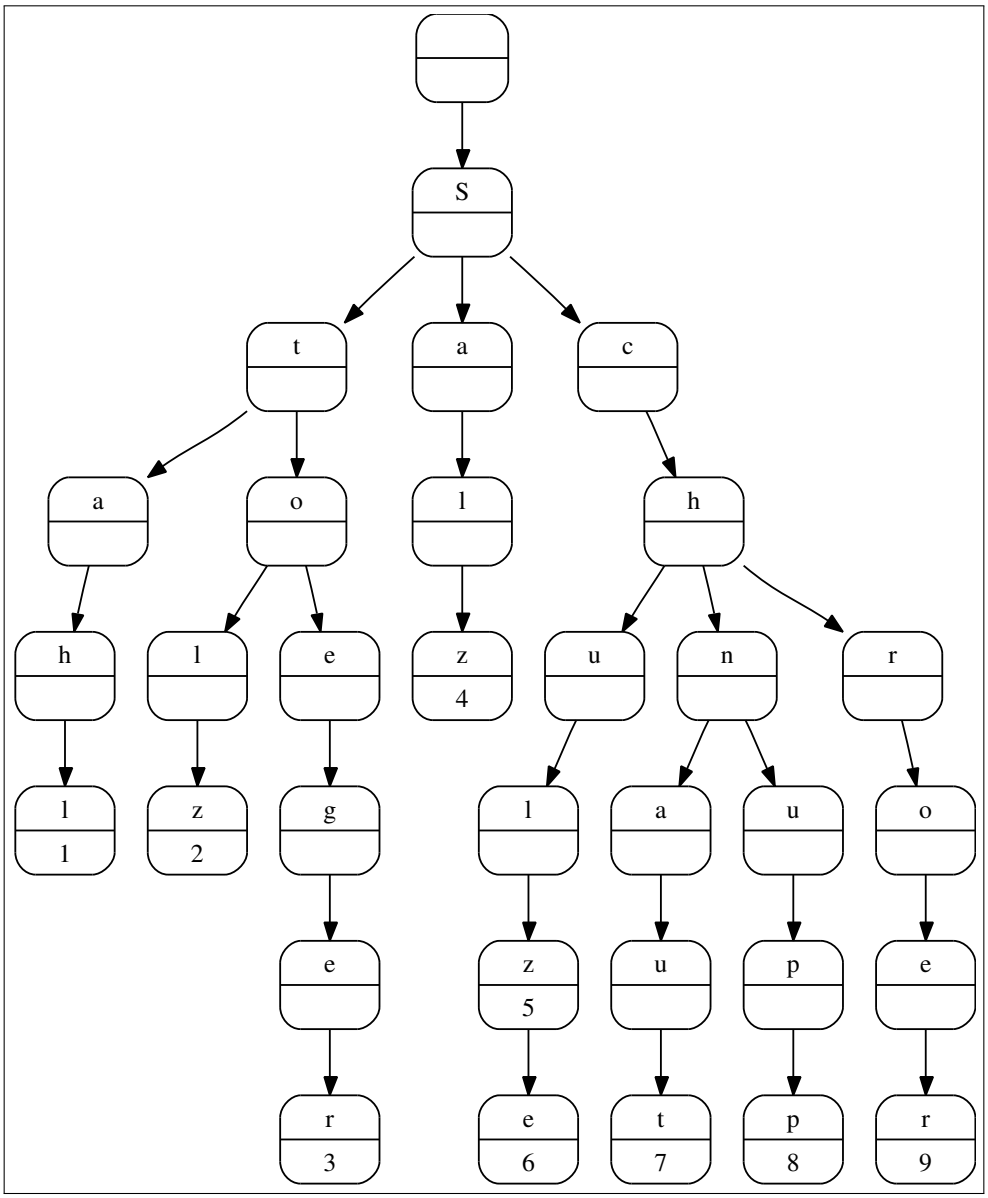

Figure 6.19: A trie storing some arbitrary numbers.

6.4.1 Insertion in Tries

Next, we present formulas 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4.2 Deletion in Tries

Finally, we present formulas 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 formula specifies the function $\texttt{isEmpty}$:

$$\texttt{Trie}(v, Cs, Ts).\texttt{isEmpty}() = \texttt{True} \stackrel{\text{def}}{\Longleftrightarrow} v = \Omega \wedge Cs = [] \wedge Ts = [].$$

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

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

is defined by induction on the length of w.

1. $\texttt{Trie}(v, Cs, Ts).\texttt{delete}(\varepsilon) = \texttt{Trie}(\Omega, Cs, Ts)$

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

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

   If the key that is to be deleted starts with the character c_i and if deletion of the key r in the i^{th} trie t_i yields an empty trie, then both the i^{th} character c_i and the i^{th} trie t_i are removed from their respective lists.

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

If the key that is to be deleted starts with the character c_i and if deletion of the key r in the i^{th} trie t_i yields a non-empty trie, then the key r has to be deleted recursively in the trie t_i and t_i has to be replaced by $t_i.\texttt{delete}(r)$.

4. $c \notin Cs \rightarrow \texttt{Trie}(v, Cs, Ts).\texttt{delete}(cr) = \texttt{Trie}(v, Cs, Ts)$.

   If the key that is to be deleted starts with the character c and 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.

6.4.3 String Completion

In addition to the functions specified in the abstract data type map, tries supports string completion: Given the prefix p of a string s that is known to be a member of some set S of strings, we can efficiently find all strings in S that start with the prefix p, provided we store the set S as a trie. Most modern programming environments offer some kind of string completion in their editors. In order to implement string completion we first define the function `allKeys`, which has the following signature:

$$\texttt{allKeys} : \mathbb{T} \times \Sigma^* \rightarrow \texttt{Set}(\Sigma^*)$$

Given a trie t and a string p, the function $t.\texttt{allKeys}(p)$ computes the set of all strings that are stored as keys in the trie t. Furthermore, the string p is added as a prefix to all these string. Therefore we specify the semantics of the function `allKeys` as follows:

$$t.\texttt{allKeys}(p) = \{p + w \mid w \in t\}.$$

Here, $p + w$ denotes the concatenation of the strings p and w and the expression $w \in t$ is true iff $t.\texttt{find}(w) \neq \Omega$. You might wonder why the function `allKeys` has two arguments. The reason is that this enables us to compute the value $t.\texttt{allKeys}(p)$ by induction on t. There are two cases in this inductive definition:

(a) $\texttt{Trie}(\Omega, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\texttt{allKeys}(p) = \bigcup_{i=1}^{n} t_i.\texttt{allKeys}(p + c_i)$,

(b) $v \neq \Omega \rightarrow \texttt{Trie}(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]).\texttt{allKeys}(p) = \{p\} \cup \bigcup_{i=1}^{n} t_i.\texttt{allKeys}(p + c_i)$.

The function `allKeys` is useful in itself because if we call it with an empty string as the second argument, then it returns the set of all keys that are stored in the given trie. The second argument is needed later when we implement string completion.

Next, we specify the auxiliary function `replacePrefix` that has the following signature:

$$\texttt{replacePrefix} : \mathbb{T} \times \Sigma^* \times \Sigma^* \rightarrow \texttt{Set}(\Sigma^*)$$

Given a trie t, a string s, and a string p, the expression $t.\texttt{replacePrefix}(s, p)$ returns the set of all strings that are used as keys in t and, furthermore, have the prefix s. Additionally, it replaces the prefix s with the string p. Therefore we have

$$t.\texttt{replacePrefix}(s, p) = \big\{p + r \mid s + r \in t\big\}.$$

The second argument p is needed in order to make the inductive definition of the function `replacePrefix` work out. The recursive definition of $t.\texttt{replacePrefix}(s, p)$ is given next:

(a) $t.\texttt{replacePrefix}(\varepsilon, p) = t.\texttt{allKeys}(p)$,

(b) $\texttt{Trie}\big(v, [c_1, \cdots, c_n], [t_1, \cdots, t_n]\big).\texttt{replacePrefix}(c_i r, p) = t_i.\texttt{replacePrefix}(r, p)$,

(c) $c \notin Cs \rightarrow \texttt{Trie}(v, Cs, Ts).\texttt{replacePrefix}(cr, p) = \{\}$.

Finally, we specify the function

$$\texttt{findPrefix} : \mathbb{T} \times \Sigma^* \rightarrow \texttt{Set}(\Sigma^*)$$

so that given a trie t and a prefix p, the expression $t.\texttt{findPrefix}(p)$ finds all strings $s \in t$ that have the prefix p, i.e. that can be written in the form $s = p + r$:

$$t.\texttt{findPrefix}(p) = \big\{p + r \mid p + r \in t\big\}.$$

Having defined the auxiliary function `replacePrefix`, the function `findPrefix` can be implemented as follows:

$$t.\texttt{findPrefix}(s) = t.\texttt{replacePrefix}(s, s).$$

6.4.4 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 that are stored in the trie. 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.

6.4.5 Implementing Tries in Python

```
1  class Trie():
2      def __init__(self):
3          self.mValue  = None
4          self.mChars  = []
5          self.mTries  = []
```

Figure 6.20: The constructor of the class `Trie`.

We proceed to discuss the implementation of tries. Figure 6.20 shows the definition of the class `Trie` and its constructor. This class supports three member variables. In order to understand these member variables, remember that a trie has the form

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

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 `Trie` 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.

```
1   def find(self, s):
2       if s == '':
3           return self.mValue
4       c, r = s[0], s[1:]
5       for i, ci in enumerate(self.mChars):
6           if c == ci:
7               return self.mTries[i].find(r)
```

Figure 6.21: Implementation of `find` for tries.

2. Otherwise, s can be written as $s = cr$ where c is the first character of s while r consists of the remaining characters. In order to check whether the trie has a value stored for s we first have to check whether there is an index i such that `mChars`$[i]$ is equal to c. If this is the case, the subtrie `mTries`$[i]$ contains the value associated with s. Then, we have to invoke `find` recursively on this subtrie.

   If the loop in line 5 does not find the character c in the list `mChars`, then the method `find` will just return the undefined value `None`. In *Python* this happens automatically when a function ends without explicitly returning a value.

```
1   def insert(self, s, v):
2       if s == '':
3           self.mValue = v
4           return
5       c, r = s[0], s[1:]
6       for i, ci in enumerate(self.mChars):
7           if c == ci:
8               self.mTries[i].insert(r, v)
9               return
10      t = Trie()
11      t.insert(r, v)
12      t.mParent = c # necessary for visualization
13      self.mChars.append(c)
14      self.mTries.append(t)
```

Figure 6.22: 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 and return.

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 recursively insert the value v in the trie `mTries`$[i]$.

(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   def delete(self, s):
2       if s == '':
3           self.mValue = None
4           return
5       c, r = s[0], s[1:]
6       for i, ci in enumerate(self.mChars):
7           if c == ci:
8               self.mTries[i].delete(r)
9               if self.mTries[i].isEmpty():
10                  self.mChars.pop(i)
11                  self.mTries.pop(i)
12              return
```

Figure 6.23: Implementation of delete for tries.

The method delete takes a string s 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 None, which represents the undefined value Ω in *Python*.

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 recursively delete the value associated with r in the trie mTries$[i]$. After this deletion, the subtrie mTries$[i]$ might well be empty. In this case, we remove the i^{th} character form mChars and also remove the i^{th} trie from the list mTries. This is done with the help of the function pop. Given a list L and an integer i, the statement L.pop(i) removes the i^{th} element from the list L.

```
1   def isEmpty(self):
2       return self.mValue == None and self.mChars == []
```

Figure 6.24: Implementation of isEmpty for tries.

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 6.24.

Exercise 19: (**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 tries are reduced to `Nil`. A binary trie b is defined to be empty iff $b.\texttt{find}(n) = \Omega$ for all $n \in \mathbb{N}$.

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

(b) Implement binary tries in *Python*.

(c) Test your implementation with a nontrivial example.

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

6.5 Hash Tables*

It is very simple to implement a function of the form

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

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

$$\texttt{Key} = \{0, 1, 2, \cdots, n-1\}.$$

In this case, we can implement the function f via an array of size n. Figure 6.25 shows how a map can be realized in this case.

```
1   class ArrayMap:
2       def __init__(self, n):
3           self.mArray = [None] * n
4
5       def find(self, k):
6           return self.mArray[k]
7
8       def insert(self, k, v):
9           self.mArray[k] = v
10
11      def delete(self, k):
12          self.mArray[k] = None
```

Figure 6.25: Implementing a map as an array.

1. The constructor takes a second argument n. This argument specifies the maximum size that a key is allowed to have. Therefore, it is assumed that the keys are elements of the set $\{0, 1, \cdots, n\}$.
2. The member variable maintained by the class ArrayMap is the array mArray. Initially, all entries of this array are None.
3. The method find looks up the value stored at the index k.
4. The method insert stores the value v at the index k.
5. The method delete removes the value stored at the index k by overwriting it with the undefined value.

If the domain $D := \texttt{dom}(f)$ of the function f is not a set of the form $\{0, 1, \cdots, n-1\}$, then we can instead try to find a one-to-one mapping of D onto a set of the form $\{0, 1, \cdots, n-1\}$. Let us explain the idea with a simple example: Suppose we want to implement a digital 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

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

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

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

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

as follows:

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

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 $\{0, 1, \cdots, 282\,429\,536\,481\}$.

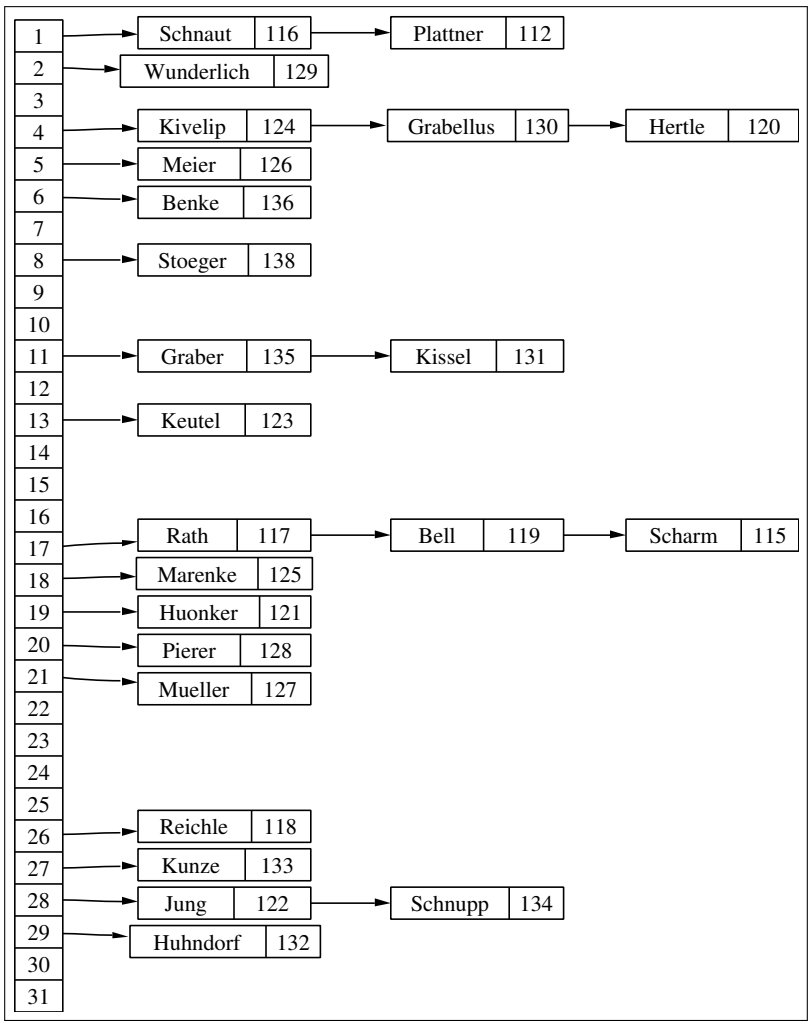

Figure 6.26: A hash table.

Unfortunately, this naive implementation has several problems:

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

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

entries. If every entry needs 8 bytes, we would need more than two terabyte to store this array.

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 the function `code` as

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

   then we will always have $\texttt{code}(c_0c_1\cdots c_n) < \texttt{size}$.

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

   The reason we have to use 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 list of key-value pairs. These 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, $\texttt{array}[\texttt{idx}]$ returns a 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.

6.5.1 Computing the Hash Function Efficiently

In this section we discuss how to compute the function

$$\texttt{hash\_code} : \Sigma^* \to \mathbb{N}$$

that takes a string s and returns a natural number less than some predefined size n. Given an ASCII string $s = c_0c_1c_2\cdots c_{k-1}$ of length k this function is defined as

$$\texttt{hash\_code}(c_0c_1\cdots c_{k-1}, n) = \left(\sum_{i=0}^{k-1} \texttt{ord}(c_i)\cdot 128^i\right) \,\%\, n. \tag{6.5}$$

However, a naive implementation of equation (6.5) is inefficient because the computation of 128^i produces very big numbers. In the programming language C, computing 128^i causes an overflow for a string that has more than five characters. In *Python*, integers are only bounded by the size of the available memory and therefore overflow ist not a problem. However, a computation involving numbers with hundreds of places is considerably slower than a computation that uses numbers that fit into a single memory word. Hence we should try to reorganize the computation that takes place in equation (6.5). This should be possible as the final result of $\texttt{hash\_code}(c_0c_1\cdots c_{k-1}, n)$ is bounded by n. In order to be able to reorganize this computation we need to discuss Euclidean division.

Theorem 16 (Euclidean Division) Given two natural numbers $a, n \in \mathbb{N}$ such that $n > 0$, there exist **unique** natural numbers q and r such that:

(a) $a = q \cdot n + r$ and

(b) $0 \leq r < n$.

The number q is called the quotient of a divided by n and denoted as $a \,//\, n$, while r is called the remainder of a divided by n and denoted as $a \,\%\, n$. ◇

The existential claim of this theorem can be established by providing a program that computes $a \,//\, n$ and $a \,\%\, n$. A trivial way to do compute $a \,//\, n$ is to successively subtract n from a as long as the result of the subtraction is non-negative. Then $a \,//\, n$ is the number of times n can be subtracted from a. Figure 6.27 shows an implementation of this idea.

```
1  def divide(a, n):
2      q = 0
3      while a >= n:
4          a -= n
5          q += 1
6      return q, a
```

Figure 6.27: A naive implementation of Euclidean division.

Exercise 20: Prove the uniqueness of the quotient and the remainder in the theorem on Euclidean division. In order to do this, assume that

1. $a = q_1 \cdot n + r_1 = q_2 \cdot n + r_2$ and
2. $0 \leq r_1 < n$ and $0 \leq r_2 < n$

holds and prove that this implies both $q_1 = q_2$ as well as $r_1 = r_2$. ◇

When computing a sum $(a + b) \,\%\, n$ and b is a big number, the following theorem can be used to simplify the computation.

Theorem 17 (Modulo Arithmetic: Addition) If $a, b \in \mathbb{N}$ and $n \in \mathbb{N}$ with $n > 0$, then we have

$$(a + b) \,\%\, n = (a + b \,\%\, n) \,\%\, n.$$

Proof: If we define $q_1 := (a + b) \,//\, n$, then according to the division theorem we have

$$a + b = q_1 \cdot n + (a + b) \,\%\, n.$$

Similarly, if $q_2 := b \,//\, n$ we have

$$b = q_2 \cdot n + b \,\%\, n,$$

which implies that $b \,\%\, n = b - q_2 \cdot n$. Therefore we have

$$\begin{array}{lcl} a + b \,\%\, n & = & a + b - q_2 \cdot n \\ & = & q_1 \cdot n + (a + b) \,\%\, n - q_2 \cdot n \\ & = & (q_1 - q_2) \cdot n + (a + b) \,\%\, n \end{array}$$

Hence we have shown that

$$a + b \,\%\, n = (q_1 - q_2) \cdot n + (a + b) \,\%\, n.$$

The right hand side of this equation has the same form as the first condition in the Euclidean division theorem. As we also have $0 \leq (a + b) \,\%\, n < n$, the uniqueness of Euclidean division implies that

$$(a + b \,\%\, n) \,\%\, n = (a + b) \,\%\, n$$

holds. □

Theorem 18 (Modulo Arithmetic: Multiplication)
If $a, b \in \mathbb{N}$ and $n \in \mathbb{N}$ with $n > 0$, then we have

$$(a \cdot b) \,\%\, n = \big(a \cdot (b \,\%\, n)\big) \,\%\, n.$$

Exercise 21: Prove the last theorem. ◇

Next, we assume to have been given a finite sequence of numbers a_0, a_1, a_2, $\cdots$, a_m and we want to compute the expression

$$p(x) := \left(\sum_{i=0}^{m} a_i \cdot x^i\right) \% n.$$

Horner's method for modular arithmetic is an efficient way to do this. This method is introduced in the following theorem.

Theorem 19 (Horner's Method for Modular Arithmetic) Assume that a_0, a_1, a_2, $\cdots$, a_m is a finite sequence of natural numbers, $x \in \mathbb{N}$, and $n \in \mathbb{N}$ such that $n > 0$. We define the sequence s_k for $k = 0, 1, \cdots, m$ inductively as follows:

B.C.: $k = 0$

$$s_0 := a_m \,\%\, n.$$

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

$$s_{k+1} := (s_k \cdot x + a_{m-(k+1)}) \,\%\, n.$$

Then we have

$$s_k = \left(\sum_{i=0}^{k} a_{m-i} \cdot x^{k-i}\right) \% n.$$

Proof: As the numbers s_m are defined by induction, the proof of this theorem is by induction on m.

B.C.: $k = 0$

$$\left(\sum_{i=0}^{0} a_{m-i} \cdot x^{0-i}\right) \% n = \big(a_{m-0} \cdot x^0\big) \,\%\, n = a_m \,\%\, n = s_0. \;\surd$$

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

$$\begin{aligned}
s_{k+1} &= \big(s_k \cdot x + a_{m-(k+1)}\big) \,\%\, n \\
&\overset{\text{ih}}{=} \left(\left(\left(\sum_{i=0}^{k} a_{m-i} \cdot x^{k-i}\right) \% n\right) \cdot x + a_{m-(k+1)}\right) \% n \\
&= \left(\left(\sum_{i=0}^{k} a_{m-i} \cdot x^{k-i}\right) \cdot x + a_{m-(k+1)}\right) \% n \\
&= \left(\sum_{i=0}^{k} a_{m-i} \cdot x^{k+1-i} + a_{m-(k+1)} \cdot x^{k+1-(k+1)}\right) \% n \\
&= \left(\sum_{i=0}^{k+1} a_{m-i} \cdot x^{k+1-i}\right) \% n \quad \surd
\end{aligned}$$

□

Setting $k := m$ in Horner's method we arrive at the following formula:

$$s_m = \left(\sum_{i=0}^{m} a_{m-i} \cdot x^{m-i}\right) \% n = \left(\sum_{i=0}^{m} a_i \cdot x^i\right) \% n = p(x).$$

This formula is used to implement the function `hash_code` efficiently. Figure 6.34 on page 120 shows an implementation of this formula. We have set x to 128 here since 128 is the size of the ASCII alphabet.

6.5.2 Implementing Hash Tables

In order to implement a has table we first need to implement three auxiliary classes:

1. We implement the class `ListNode` that represents the node of a linked list.
2. We implement the class `ListMap` that turns a linked list into a Map.
3. We implement the class `MapIterator` that is needed to iterate over a linked list.

The class `ListNode`

We start by defining the class `ListNode`. The implementation is shown in Figure 6.28 on page 116. An object of class `ListNode` stores a key, a value, and a pointer to the next node.

(a) `mKey` stores the key,

(b) `mValue` stores the value associated with this key, and

(c) `mNextPtr` stores a reference to the next node. If there is no next node, then `mNextPtr` is `None`.

```
1     class ListNode:
2         def __init__(self, key, value):
3             self.mKey     = key
4             self.mValue   = value
5             self.mNextPtr = None
```

Figure 6.28: The constructor of the class `ListNode`.

Given a `key`, the method `find` shown in Figure 6.29 traverses the given linked list that starts at the node `self` until it finds a node that stores the given `key`. In this case, it returns the associated value. Otherwise, `None` is returned.

```
def find(self, key):
2    while True:
3        if self.mKey == key:
4            return self.mValue
5        if self.mNextPtr != None:
6            self = self.mNextPtr
7        else:
8            return
```

Figure 6.29: The method `find` of the class `ListNode`.

Given the first node of a linked list L, the function L.`insert`(k, v) shown in Figure 6.30 on page 117 inserts the key-value pair (k, v) into the list L. If there is already a key value pair in L that has

the same key, then the old value is overwritten. It returns a boolean that is `True` if a new node has been allocated. If no new nodes needs to be allocated, `False` is returned instead. This information is needed later in order to keep track of the number of nodes in a linked list.

```
def insert(self, key, value):
    while True:
        if self.mKey == key:
            self.mValue = value
            return False
        elif self.mNextPtr != None:
            self = self.mNextPtr
        else:
            self.mNextPtr = ListNode(key, value)
            return True
```

Figure 6.30: The method `insert` of the class `ListNode`.

```
def delete(self, key):
    previous = None
    ptr      = self
    while True:
        if ptr.mKey == key:
            if previous == None:
                return ptr.mNextPtr, True
            else:
                previous.mNextPtr = ptr.mNextPtr
                return self, True
        elif ptr.mNextPtr != None:
            previous = ptr
            ptr      = ptr.mNextPtr
        else:
            return self, False
```

Figure 6.31: The method `delete` of the class `ListNode`.

Given the first node of a linked list L, the method L.`delete`(k) shown in Figure 6.31 on page 117 removes the first key-value pair of the form (k, v) from the list L. If there is no such pair, the list L is left unchanged. The expression L.`delete`(k) returns a pair:

(a) The first component of this pair is a pointer to the resulting list. If the list becomes empty, the first component is `None`.

(b) The second component is a Boolean that is `True` if a node has been deleted.

In the implementation, the variable `previous` is a pointer to the object of class `ListNode` that points to the node `self`. For the first node, this pointer is `None`, but when we step from one node to the next, this pointer is updated. When we find the `key` that is to be deleted, there are essentially two cases:

1. The `key` to be deleted is contained in the first node of the linked list. In this case, `previous` is still `None`. Therefore, we return a pointer to the rest of the list and set the Boolean that is returned to `True`.

2. Otherwise, we have to connect the node preceding the given node with the node succeeding the given node. This is done by setting `previous.mNextPtr` to `ptr.mNextPtr`. In this case, we return the pair (`self`, `True`) since `self` still points to the beginning of the list.

As long as we have not found the `key`, we keep following the pointer `ptr.mNextPtr` to iterate over the elements of the linked list. If we reach the end of the list without finding the `key`, the pair (`self`, `False`) is returned.

The class `ListMap`

```
1   class ListMap:
2       def __init__(self):
3           self.mPtr = None
4
5       def find(self, key):
6           if self.mPtr != None:
7               return self.mPtr.find(key)
8
9       def insert(self, key, value):
10          if self.mPtr != None:
11              return self.mPtr.insert(key, value)
12          else:
13              self.mPtr = ListNode(key, value)
14              return True
15
16      def delete(self, key):
17          if self.mPtr != None:
18              self.Ptr, flag = self.mPtr.delete(key)
19              return flag
20          return False
21
22      def __iter__(self):
23          return MapIterator(self.mPtr)
```

Figure 6.32: The class `ListMap`.

Now we are ready to present the class `ListMap` that implements the abstract data type Map using a linked list. Figure 6.32 on page 118 shows the implementation.

(a) The class maintains the member variable `mPtr`, which is a pointer to the first node of a linked list. As long as the list is empty, this pointer is `None`.

Basically, the class `ListMap` is a wrapper for the class `ListNode` that is needed to deal with empty lists.

(b) The method `find` is a wrapper of the method `find` from the class `ListNode`. If `mPtr` is set, then the linked list at `mPtr` is searched for the given `key`. Otherwise, the list is empty and `None` is returned.

(c) The method `insert` is a wrapper of the method `insert` from the class `ListNode`. If `mPtr` is set, then the given `key-value` pair is inserted into the linked list specified by `mPtr`. Otherwise, a new `ListNode` is created that stores the given `key-value`. In this case, the method has to return `True` since a new key has been inserted.

(d) Similarly, the method `delete` is a wrapper of the method `delete` from the class `ListNode`. If the given list is not empty, i.e. `mPtr` is not `None`, the implementation tries to delete the given `key` in the list pointed to by `self.mPtr`. Care has to be taken for the case that the `key` to be deleted is the first node of the list pointed to by `mPtr`. In this case, the value of `mPtr` will change.

(e) Furthermore, the class implements the function `__iter__` which returns a `MapIterator`. Therefore, an object of class `ListMap` is iterable: We can use a `for`-loop to iterate over the list of key-value pairs that is stored.

The class `MapIterator`

Next, we discuss the class `MapIterator` which makes `ListMap` objects iterable. This class is shown in Figure 6.33 on page 119. This class maintains a pointer to the next list node that is to be returned by this iterator. The constructor sets this pointer to the first node of the associated linked list. The method `__next__` is responsible for retrieving a key-value pair and updating this pointer. If the end of the list is reached, then this method raises a `StopIteration`, which signals that the list has been exhausted.

```
1   class MapIterator:
2       def __init__(self, ptr):
3           self.mPtr = ptr
4
5       def __next__(self):
6           if self.mPtr == None:
7               raise StopIteration
8           key   = self.mPtr.mKey
9           value = self.mPtr.mValue
10          self.mPtr = self.mPtr.mNextPtr
11          return key, value
```

Figure 6.33: The class `MapIterator`.

The class `HashMap`

In order to implement a hash table we need a hash function. Figure 6.34 on page 120 shows the function `hash_code`. Given a string w and the size n of the hash table, the function $\texttt{hash\_code}(w, n)$ calculates the hash code of w. For a string $w = c_0 c_1 \cdots c_{m-1}$ of length m, this function is defined as follows:

$$\texttt{hash\_code}(w, n) = \left(\sum_{i=0}^{m-1} \texttt{ord}(c_i) \cdot 128^i \right) \,\%\, n$$

In order to prevent overflows when computing the numbers 128^i we can define the partial sum s_k for $k = 0, 1, \cdots, m-1$ by induction:

(a) $s_0 = \texttt{ord}(c_{m-1}) \,\%\, n$,

(b) $s_{k+1} = \big(s_k \cdot 128 + \texttt{ord}(c_k)\big) \,\%\, n$.

Then we have

$$s_{m-1} = \left(\sum_{i=0}^{m-1} \texttt{ord}(c_i) \cdot 128^i \right) \,\%\, n.$$

```
1  def hash_code(w, n):
2      m = len(w)
3      s = 0
4      for k in range(m-1, -1, -1):
5          s = (s * 128 + ord(w[k])) % n
6      return s
```

Figure 6.34: The function `hash_code` to compute a hash code for a string.

Figure 6.35 on page 120 shows the class `HashMap`

```
1  class HashTable:
2      def __init__(self, n, code=hash_code):
3          self.mSize    = n
4          self.mEntries = 0
5          self.mArray   = [ ListMap() for i in range(self.mSize) ]
6          self.mAlpha   = 2
7          self.mCode    = code
8
9  HashTable.Primes = [ 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093,
10                      8191, 16381, 32749, 65521, 131071, 262139, 524287,
11                      1048573, 2097143, 4194301, 8388593, 16777213,
12                      33554393, 67108859, 134217689, 268435399,
13                      536870909, 1073741789, 2147483647
14                    ]
```

Figure 6.35: Definition of the class `HashTable`.

1. The constructor `__init__` is called with two arguments.

   (a) This argument n is the initial size of the array storing the `ListMaps` of key-value pairs. The constructor constructs an empty hash table of size n.

   (b) `code` is a function that takes an object and returns the hash code of this object. The default is to use the function `hash_code` defined above.

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 be increased later if more space is needed. 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. As the hash map is initially empty, all entries of `mArray` are initialized as empty `ListMaps`.

5. mAlpha is the load factor of our hash table. If at any point in time we have that

   $$\texttt{mEntries} > \texttt{mAlpha} \cdot \texttt{mSize},$$

   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 is too big, many entries in the array mArray will be empty and thus we will waste space. On the other hand, if mAlpha is too small, the key-value lists will become very long and hence it will take too much time to search for a given key in one of these lists.

6. Our implementation maintains the static variable Primes. This 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 the hash table 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 methods of the class HashTable.

```
1    def find(self, key):
2        index = self.mCode(key, self.mSize)
3        aList = self.mArray[index]
4        return aList.find(key)
```

Figure 6.36: Implementation of find.

Figure 6.36 shows the implementation of the method find.

1. First, we compute the index of the ListMap that is used to store the given key.
2. Next, we retrieve this ListMap from the array mArray.
3. Finally, we look up the information stored under the given key in this key-value list.

```
1    def insert(self, key, value):
2        if self.mEntries >= self.mSize * self.mAlpha:
3            self._rehash()
4        index = self.mCode(key, self.mSize)
5        aList = self.mArray[index]
6        self.mEntries += aList.insert(key, value)
```

Figure 6.37: Implementation of the method insert.

Figure 6.37 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.
2. Next, we compute the index of the ListMap that has to store mKey, retrieve the associated ListMap, and finally insert the key and the value into this ListMap. When doing this we take care to maintain the correct number of entries.

```
1   def _rehash(self):
2       for p in HashTable.Primes:
3           if p * self.mAlpha > self.mEntries:
4               prime = p
5               break
6       biggerTable = HashTable(prime, self.mCode)
7       for aList in self.mArray:
8           for k, v in aList:
9               biggerTable.insert(k, v)
10      self.mSize  = prime
11      self.mArray = biggerTable.mArray
```

Figure 6.38: Implementation of the method rehash.

Figure 6.38 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. This will assure that after rehashing the average 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 insert the key-value pairs from the given hash table in our newly created new hash table biggerTable.
3. Finally, the array stored in the new hash table is moved to the given hash table and the size is adjusted correspondingly.

```
12  def delete(self, key):
13      if 2 * self.mEntries <= self.mSize * self.mAlpha:
14          self._rehash()
15      index = self.mCode(key, self.mSize)
16      aList = self.mArray[index]
17      self.mEntries -= aList.delete(key)
```

Figure 6.39: Implementation of the procedure delete(map, key).

Finally, we discuss the implementation of the method delete that is shown in Figure 6.39. The implementation of this method is similar to the implementation of the method insert. When the number of elements drops below a threshold that is dependent on the site of mArray, the table shrinks in order to not use too much memory.

In the worst case, the complexity of the methods find, insert, and delete can grow linearly with the number of entries in the hash table. This happens if the function hash_code(k) returns the same number for all keys k. Although this case is highly unlikely, it is not impossible. If we have a good function to compute hash codes, then most of the 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`.

6.5.3 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.

6.6 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 another application of maps.

6.7 Check Your Understanding

If you are able to answer the questions below confidently, then you can assume that you have mastered the concepts introduced in this chapter.

1. How are sets related to maps?
2. What is the definition of the abstract data type Map?
3. Describe a practical application of the ADT Map.
4. How is the set $\mathcal{B}$ of *ordered binary trees* defined?
5. How are the functions `find`, `insert`, and `delete` defined for ordered binary trees?
6. What is the average complexity of the function `find` for an ordered binary tree? What is the worst case complexity?
7. How is the set $\mathcal{A}$ of AVL trees defined?
8. How did we define the function `restore` that is used to maintain the balancing condition in AVL trees?
9. What is the worst case complexity of AVL trees?
10. When can we use tries to implement a map? How are tries defined?
11. How are the functions `find`, `insert`, and `delete` defined for tries?
12. How are binary tries defined?
13. Define hash tables.
14. How is the load factor of a hash table defined? What happens if the load factor gets too big?
15. Describe a simple method to compute the hash value of a string.

Chapter 7

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 7.4.

7.1 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 shortest path algorithm of Dijkstra and the compression algorithm of Huffman.

Definition 20 (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 has to exist a binary relation $\leq$ on the set `Priority` such that the pair $\langle \texttt{Priority}, \leq \rangle$ is a linear order. 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 signatures 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 that 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 the pair $\langle \texttt{Priority}, \leq\rangle$ is a linear order. If $p_1 < p_2$, then we say that 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 `top`, `insert`, `remove`, and `isEmpty` we need to introduce two auxiliary functions:

   (a) The function `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 `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 `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 `insert` that we had defined when discussing the algorithm insertion sort.

Now we can specify the behaviour of the abstract data type `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) $Q.\texttt{toList}() \neq \texttt{[]} \rightarrow Q.\texttt{top}() = Q.\texttt{toList}()[0]$
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) $Q.\texttt{toList}() \neq \texttt{[]} \rightarrow Q.\texttt{remove}().\texttt{toList}() = Q.\texttt{toList}()[1:]$
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 off 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) Furthermore, the functions `top` and `isEmpty` are also observer functions 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 linked 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. In this implementation the complexity of `insert`() is only $\mathcal{O}\big(\log(n)\big)$, while the other operations still have the complexity $\mathcal{O}(1)$. To this end, we introduce a new data structure: Heaps.

7.2 The `Heap` Data Structure

We define the set $\mathcal{H}$ of heaps[1] inductively.

[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.

1. $\texttt{Nil} \in \mathcal{H}$
2. $\texttt{Node}(p, v, l, r) \in \mathcal{H}$ if and only if the following is true:
   (a) $p \in \texttt{Priority} \wedge v \in \texttt{Value}$
   (b) $l \in \mathcal{H} \ \wedge \ r \in \mathcal{H}$
   This condition ensures that all subtrees of a heap are heaps, too.
   (c) $p \leq l \ \wedge \ p \leq r$
   Here, the notation $p \leq l$ is used to express the fact that all priorities q occurring in l satisfy $p \leq q$ and $p \leq r$ is interpreted similarly. Therefore, 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.
   (d) $| \, l.\texttt{count}() - r.\texttt{count}() \, | \leq 1$
   $l.\texttt{count}()$ computes the number of nodes of the heap l and $r.\texttt{count}()$ is interpreted similarly. A formal definition of the function count is given below. The condition
   $$| \, l.\texttt{count}() - r.\texttt{count}() \, | \leq 1$$
   therefore states the fact that 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.

The function

$$\texttt{count} : \mathcal{H} \to \mathbb{N}$$

that has been used above is formally specified as follows:

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

The heap condition implies that in a non-empty heap the element with a highest priority is stored at the root. Figure 7.1 on page 129 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. $\texttt{Nil}.\texttt{top}() = \Omega$.
2. $\texttt{Node}(p, v, l, r).\texttt{top}() = \langle p, v \rangle$,
   because the heap condition ensures that the value with the highest priority is stored at the top.

Implementing the method isEmpty is straightforward:

1. $\texttt{Nil}.\texttt{isEmpty}() = \texttt{True}$,
2. $\texttt{Node}(p, v, l, r).\texttt{isEmpty}() = \texttt{False}$.

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

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

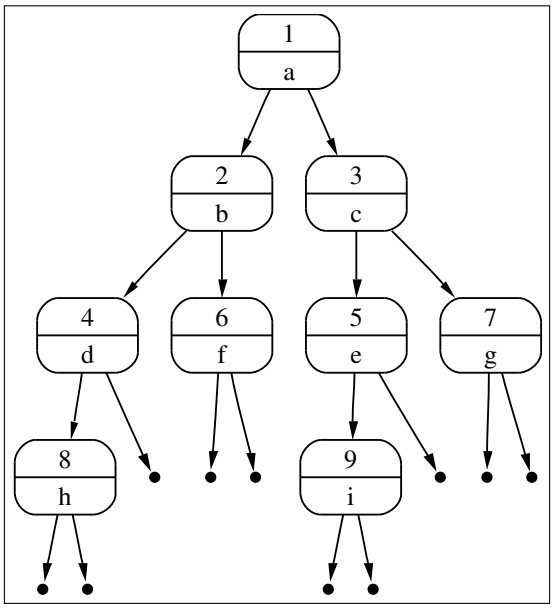

Figure 7.1: A heap.

2. $p_{\text{top}} \leq p \ \wedge \ l.\texttt{count}() \leq r.\texttt{count}() \ \rightarrow$
   $\texttt{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\texttt{insert}(p, v) = \texttt{Node}\big(p_{\text{top}}, v_{\text{top}}, l.\texttt{insert}(p, v), r\big).$
   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. $p_{\text{top}} \leq p \ \wedge \ l.\texttt{count}() > r.\texttt{count}() \ \rightarrow$
   $\texttt{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\texttt{insert}(p, v) = \texttt{Node}\big(p_{\text{top}}, v_{\text{top}}, l, r.\texttt{insert}(p, v)\big).$
   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. $p_{\text{top}} > p \ \wedge \ l.\texttt{count}() \leq r.\texttt{count}() \ \rightarrow$
   $\texttt{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\texttt{insert}(p, v) = \texttt{Node}\big(p, v, l.\texttt{insert}(p_{\text{top}}, v_{\text{top}}), r\big).$
   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. $p_{\text{top}} > p \ \wedge \ l.\texttt{count}() > r.\texttt{count}() \ \rightarrow$
   $\texttt{Node}(p_{\text{top}}, v_{\text{top}}, l, r).\texttt{insert}(p, v) = \texttt{Node}\big(p, v, l, r.\texttt{insert}(p_{\text{top}}, v_{\text{top}})\big).$
   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 was stored in the heap at any moment in time.

Exercise 22: Change the equations for the method `remove` so that the resulting heap satisfies the balancing condition.

7.3 Implementing Heaps in Python

Next, we present an implementation of heaps in *Python*. Figure 7.2 on page 131 shows the implementation of the class `Heap`. This class is a superclass of both the class `Nil` and the class `Node` which are used to represent heaps of the form `Nil` and $\texttt{Node}(p, v, l, r)$, respectively. These classes are presented later in Figure 7.3 and 7.4.

The class `Heap` maintains the static variable `sNodeCount`. This variable is used to attach a unique identifier to each node of a tree. The constructor of the class `Heap` increments this variable every time a new heap is constructed. Furthermore, the constructor initializes the member variable `mId`, which represents the unique identifier that is associated which each node. This member variable is later used by `graphviz` to present heaps as trees. However, we will not discuss the graphical presentation of heaps.

The class `Nil` shown in Figure 7.3 is a subclass of class `Heap` that creates an object representing an empty heap. The auxiliary method `_count` returns the number of values stored in this node which is, of course, zero. The implementation of the other methods is an obvious translation of the equations discussed in the previous section.

The class `Node` shown in Figure 7.4 on page 132 represents a node of the form $\texttt{Node}(p, v, l, r)$ using the following member variables:

1. `mPriority` is the priority of the value stored at this node,

```
class Heap:
    sNodeCount = 0

    def __init__(self):
        Heap.sNodeCount += 1
        self.mID = str(Heap.sNodeCount)

    def getID(self):
        return self.mID
```

Figure 7.2: The super class `Heap`.

```
class Nil(Heap):
    def __init__(self):
        Heap.__init__(self)

    def _count(self):
        return 0

    def top(self):
        return None

    def insert(self, p, v):
        return Node(p, v, Nil(), Nil())

    def remove(self):
        return self
```

Figure 7.3: The class `Nil`.

2. `mValue` stores the corresponding value,
3. `mLeft` and `mRight` represent the left and right subtree, respectively, while
4. `mCount` gives the number of nodes in the subtree rooted at this node.

The auxiliary method `_extract` returns a 4-tuple containing the first four member variables. The implementation of the methods `top`, `insert`, and `remove` is a direct translation of the equations presented earlier.

Exercise 23: 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 24: 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 $i \in \{0, \cdots, \texttt{len}(L) - 1\}$, 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 $2 \cdot i + 1$, while the right subtree is stored at the index $2 \cdot (i + 1)$. Develop an implementation of heaps that is based on this idea. ◇

```
class Node(Heap):
    def __init__(self, priority, value, left, right):
        Heap.__init__(self)
        self.mPriority = priority
        self.mValue    = value
        self.mLeft     = left
        self.mRight    = right
        self.mCount    = left._count() + right._count() + 1

    def _extract(self):
        return self.mPriority, self.mValue, self.mLeft, self.mRight

    def _count(self):
        return self.mCount

    def top(self):
        return self.mPriority, self.mValue

    def insert(self, p, v):
        p_top, v_top, l, r = self._extract()
        if p_top <= p:
            if l._count() <= r._count():
                return Node(p_top, v_top, l.insert(p, v), r)
            else:
                return Node(p_top, v_top, l, r.insert(p, v))
        else:
            if l._count() <= r._count():
                return Node(p, v, l.insert(p_top, v_top), r)
            else:
                return Node(p, v, l, r.insert(p_top, v_top))

    def remove(self):
        p, v, l, r = self._extract()
        if isinstance(l, Nil):
            return r
        if isinstance(r, Nil):
            return l
        p1, v1, l1, r1 = l._extract()
        p2, v2, l2, r2 = r._extract()
        if p1 <= p2:
            return Node(p1, v1, l.remove(), r)
        else:
            return Node(p2, v2, l, r.remove())
```

Figure 7.4: The class `Node`.

7.4 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 [Wil64] and improved by Robert W. Floyd [Flo64] in the same year. Robert Floyd received the Turing award 1978.

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:

$$\texttt{toHeap} : \texttt{List}(\mathbb{N}) \to \mathcal{H}$$

This function can be specified via the following equations:

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

Furthermore, we need an auxiliary function toList that has the signature

$$\texttt{toList} : \mathcal{H} \to \texttt{List}(\mathbb{N}).$$

This function transforms a heap in a list of numbers that is sorted ascendingly. This function is specified as follows:

1. $\texttt{Nil}.\texttt{toList}() = []$,
2. $h \neq \texttt{Nil} \wedge \langle p, \_ \rangle = h.\texttt{top}() \to h.\texttt{toList}() = [p] + h.\texttt{remove}().\texttt{toList}()$

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 7.5 on page 134. This implementation makes use of the class Heap that had been presented in the previous section.

1. In order to sort the list L that is given as argument to heap_sort, 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.
2. Next we create an empty list S in line 5. When the procedure heapSort finishes, this list will be a sorted version of the list L.
3. As long as the heap H is not empty, we take its top element and append it to S. We can test whether the heap is empty by checking that it has the type Node. Since the method top returns a pair of the form $\langle p, \texttt{None} \rangle$, 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 $\langle p, \texttt{None} \rangle$ is removed 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 7.5 can be improved by noting that a heap can be stored efficiently in an array A. If a node of the form $\texttt{Node}(p, v, l, r)$ is stored at index i, then the left subtree l is stored at index $2 \cdot i + 1$ while the right subtree r is stored at index $2 \cdot i + 2$:

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

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

```
def heap_sort(L):
    H = Nil()
    for p in L:
        H = H.insert(p, None)
    S = []
    while isinstance(H, Node):
        p, _ = H.top()
        S.append(p)
        H = H.remove()
    return S
```

Figure 7.5: Basic implementation of heapsort.

"The root of the heap 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 use index arithmetic to retrieve the subtrees. This results in memory savings as we no longer have to store the pointers.

```
def swap(A, i, j):
    A[i], A[j] = A[j], A[i]

def sink(A, k, n):
    while 2 * k + 1 <= n:
        j = 2 * k + 1
        if j + 1 <= n and A[j] > A[j + 1]:
            j += 1
        if A[k] < A[j]:
            return
        swap(A, k, j)
        k = j

def heap_sort(A):
    n = len(A) - 1
    for k in range((n + 1) // 2 - 1, -1, -1):
        sink(A, k, n)
    while n >= 1:
        swap(A, 0, n)
        n -= 1
        sink(A, 0, n)
```

Figure 7.6: An efficient implementation of Heapsort.

Figure 7.6 on page 134 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 i and j.
2. The procedure sink takes three arguments.

(a) A is the array representing the heap.

(b) k is an index into the array A.

(c) n is the upper bound of the part of this array that has to be transformed into a heap.
The array A itself might actually have more than $n+1$ elements, but for the purpose of the method `sink` we restrict our attention to the subarray $A[k:n+1]$.

When calling `sink`, the assumption is that $A[k:n+1]$ 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 $j+1$ is less than or equal to 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 the index j to point to the smaller child. Next, we check in line 9 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 at least one child, i.e. as long as $2 \cdot k + 1 \leq n$.

3. The function `heapSort` has the task to sort the array A and proceeds in two phases.

(a) In phase one 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 16. 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 $(n+1)//2-1$ are trivial, i.e. they only consist of their root node. This can be seen as follows: If $k = (n+1)//2$, which is the first natural number that is greater than $(n+1)//2-1$, then there are two cases: Either n is even or n is odd.

i. Case: n is even, i.e. there is an $m \in \mathbb{N}$ such that $n = 2 \cdot m$.
Then $k = (n+1)//2 = (2 \cdot m + 1)//2 = m$. Therefore, the left subtree of the tree rooted at k would be at position

$$2 \cdot k + 1 = 2 \cdot m + 1 = n + 1 > n.$$

As $2 \cdot k + 1$ is greater than n and n is the last index in an array of length $n+1$, the node at position $k = (n+1)//2$ does not have a left subtree. As $2 \cdot k + 2$ is even bigger than $2 \cdot k + 1$ there can be no right subtree either.

ii. Case: n is odd, i.e. there is an $m \in \mathbb{N}$ such that $n = 2 \cdot m + 1$.
Then $k = (n+1)//2 = (2 \cdot m + 1 + 1)//2 = m + 1$. Therefore, the left subtree of the tree rooted at k would be at position

$$2 \cdot k + 1 = 2 \cdot (m+1) + 1 = 2 \cdot m + 3 = n + 2 > n.$$

As $2 \cdot k + 1$ is greater than n and n is the last index in an array of length $n+1$, the node at position k does not have a left subtree and hence also no right subtree.

We conclude that if $k > (n+1)//2-1$ the node at position k has no subtrees and hence the heap condition is satisfied vacuously. This explains that we start with $k = (n+1)//2-1$ in line 16.

In order to maintain 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 its root. It is then the job of `sink` to establish the heap condition at 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.

This phase is also called the heapification of the array.

(b) In **phase two** 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 $A[0:n-1]$ represents a heap, except that the heap condition might now be violated at the root. Next, we decrement n in line 20, since the last element of the array A is already in its correct position in the list that is to be returned in the end. In order to reestablish the heap condition at the root, we call `sink` with index `0` in line 21.

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.

Prof. David Galles from the University of San Francisco has implemented a nice animation of heapsort that is available at the following address:

https://www.cs.usfca.edu/~galles/visualization/HeapSort.html.

However, he has defined heaps in a way that is different from our definition: In his version, `Node`(p, v, l, r) is a heap if p is bigger than all priorities occurring in l and r.

7.4.1 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].

Exercise 25: Develop an algorithm for sorting a list of n numbers that makes use of AVL-trees. Your algorithm should have the complexity $\mathcal{O}\big(n \cdot \log_2(n)\big)$. Specify the algorithm via recursive equations. ◇

7.5 Check Your Understanding

1. Which methods are provided by a priority queue?
2. Given these methods, how can we transform a priority queue into an ordered list?
3. Assume that we have a method `toList` that transforms a priority queue into an ordered list. How can this method be used to specify the behaviour of the methods provided by a priority queue?
4. How is the set $\mathcal{H}$ of heaps defined?
5. How can we define the methods `insert` and `remove` if we implement a priority queue as a heap?
6. How can we represent a heap as an array? What are the benefits of this representation.
7. Try to implement the algorithm heap sort in a language of your choice without consulting the text.
8. Compare heap sort and merge sort.

Chapter 8

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 smallest 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.

For reasons of time, we will only be able to discuss Huffman's algorithm.

8.1 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

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

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 8.1 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 8.1: Variable-length encoding of the characters.

In order to understand how this encoding works we represent this encoding in Figure 8.1 as a coding tree: The inner nodes of this tree do not contain any attributes and are therefore shown 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 8.1. 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 8.1.

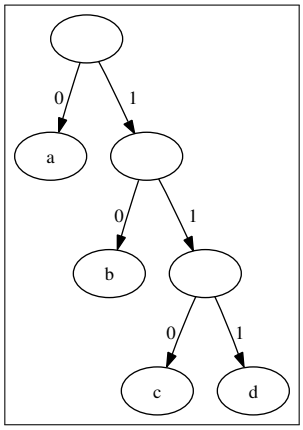

Figure 8.1: Tree representation of the encoding shown in Figure 8.1.

In order to see how a bit string can be decoded using the encoding shown in Figure 8.1 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".

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

   An expression of the form $\texttt{Leaf}(c, f)$ 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 8.1 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. $\texttt{Node}(l, r) \in \mathcal{K}$ if $l \in \mathcal{K}$ and $r \in \mathcal{K}$.

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

Next, we define the function

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

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:

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

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

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

Next we define the function

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

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 $\texttt{cost}(t)$ is given by induction on the coding tree t.

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

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

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

   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

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

   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 $l.\texttt{count}() + r.\texttt{count}()$ to the costs of l and r.

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

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

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

$$M = \big\{\texttt{Leaf}(c_1, f_1), \cdots, \texttt{Leaf}(c_k, f_k)\big\}. \tag{8.1}$$

Huffmann's algorithm combines two nodes a and b from M into a new node $\texttt{Node}(a, b)$ 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

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

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 $\texttt{Node}(a, b)$:

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

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

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

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 $\texttt{Node}(a, b)$. 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.
The function `codingTree` program shown in Figure 8.2 implements this algorithm.

1. *Python* has a module called `heapq`. This module implements a heap via a list. This module offers the following functions:

   (a) Given a heap H and an item x, the function call

   $$\texttt{heapq.heappush}(H, x)$$

   pushes the item x onto the heap H. The item x serves both as a priority and a value. In applications of `heapq` the items are usually pairs of the form (p, v) where p is a priority (often represented as a natural number) and v is some value.
   The function call $\texttt{heappush}(H, x)$ does not return a value. Rather, the heap H is modified in place.

   (b) Given a heap H, the function call

   $$\texttt{heapq.heappop}(H)$$

   returns the object from H that has the highest priority and removes this object from the heap H.

   (c) As heaps are represented as lists, the empty heap is represented by the empty list.

```
1  import heapq
2
3  class CodingTree:
4      pass
5
6  class Leaf(CodingTree):
7      def __init__(self, c, f):
8          self.mCharacter = c
9          self.mFrequency = f
10
11     def __lt__(self, other):
12         if isinstance(other, Node):
13             return True
14         return self.mCharacter < other.mCharacter
15
16 class Node(CodingTree):
17     def __init__(self, l, r):
18         self.mLeft  = l
19         self.mRight = r
20
21     def __lt__(self, other):
22         if isinstance(other, Leaf):
23             return False
24         return self.mLeft < other.mLeft
25
26 def codingTree(M):
27     H = []          # empty priority queue
28     for c, f in M:
29         heapq.heappush(H, (f, Leaf(c, f)))
30     while len(H) > 1:
31         a = heapq.heappop(H)
32         b = heapq.heappop(H)
33         heapq.heappush(H, (a[0] + b[0], Node(a[1], b[1])))
34     return H[0][1]
```

Figure 8.2: Huffman's algorithm implemented in *Python*.

2. We define a superclass `CodingTree` and two derived classes `Leaf` and `Node`. As we want to place objects of these classes onto a priority queue, we have to make them comparable. This is done by defining the method `__lt__` in these classes. This method is automatically called whenever two objects of class `CodingTree` are compared using the operator $<$. As this only happens when the corresponding counts are different, the precise implementation of this method does not matter as long as we take care to ensure that $<$ is a linear order.

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

   $$M = \big\{(c_1, f_1), \cdots, (c_k, f_k)\big\}.$$

   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.

4. Line 27 creates the empty heap.

5. The `for`-loop in line 28 turns pairs of the form (c, f) into pairs of the form

   $$\big(f, \texttt{Leaf}(c, f)\big)$$

   In the pairs $\big(f, \texttt{Leaf}(c, f)\big)$ the frequency f is stored first because when *Python* compares two pairs (x_1, y_1) and (x_2, y_2), the comparison is done lexicograhically, i.e. we have

   $$(x_1, y_1) < (x_2, y_2) \quad \text{iff} \quad x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2).$$

   Therefore the pair $\big(f_1, \texttt{Leaf}(c_1, f_1)\big)$ has a higher priority than the pair $\big(f_2, \texttt{Leaf}(c_2, f_2)\big)$ if $f_1 < f_2$. If the frequencies f_1 and f_2 are the same, then the priority is decided by comparing the objects $\texttt{Leaf}(c_1, f_1)$ and $\texttt{Leaf}(c_2, f_2)$. For the purpose of string compression the order does not matter in this case.

   The pairs $\big(f, \texttt{Leaf}(c, f)\big)$ are inserted into the heap H.

6. The `while` loop in line 2 reduces the number of nodes of the set M in every step by one. This is done by combining the nodes a and b into a single node $\texttt{Node}(a, b)$

   (a) Using the function `heappop` we compute those nodes a and b that have the lowest character count and remove them from the heap H.

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

7. The `while` loop terminates when H contains but a single element. This element is then 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 8.2: 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 8.2 on page 142.

1. Initially, the set M has the form

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

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

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

   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

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

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

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

3. The two pairs with the smallest frequencies are now

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

   These pairs are removed form M and replaced by

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

Then M is given as

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

4. Now the pairs

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

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

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

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

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

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

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

   This node is our result. Figure 8.3 shows the corresponding coding tree. Here every node n is labelled with its count. The resulting encoding is shown in table 8.3.

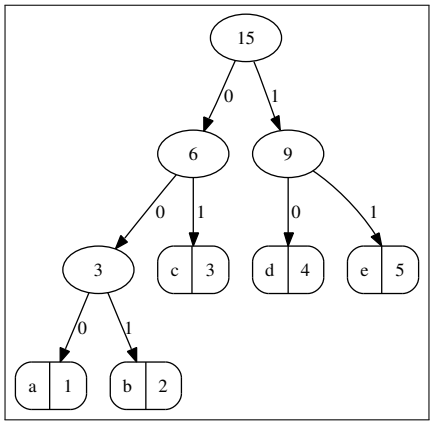

Figure 8.3: Tree representation of the coding tree.

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

Table 8.3: Variable length encoding of the letters "a" to "e".

Exercise 26:

(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 8.4: Letters with frequencies.

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

◇

8.3 Check Your Understanding

(a) What are the two principal ideas that can be used to compress a string?

(b) How is the set $\mathcal{K}$ of coding trees defined?

(c) Define a function `encode` : $\Sigma^* \times \mathcal{K} \to \{0,1\}^*$ that takes a string $s \in \Sigma^*$ and a coding tree $k \in \mathcal{K}$ and that produces a binary string b that encodes the string s using the coding tree k.

(d) Define a function `decode` : $\{0,1\}^* \times \mathcal{K} \to \Sigma^*$ that takes a binary string b and a coding tree k and decodes b according to the coding tree k.

(e) Assume you are given a dictionary `D` such that for every character $c \in \Sigma$ we have that $D[c]$ is the number of occurrences of the character c in a string $s \in \Sigma^*$. Are you able to implement a function `codingTree` that is able to compute the Huffman coding tree for this string?

Hint: We have implemented a similar function in this chapter.

Chapter 9

Graph Theory

In this chapter we are going to discuss four problems from graph theory.

(a) 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.

(b) 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.

(c) Then we discuss the problem of finding a shortest path in a *weighted directed graph*. We present Dijkstra's algorithm to solve this problem.

(d) Finally, we discuss topological sorting.

9.1 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:

(a) $\approx_R$ is an equivalence relation on M.

(b) 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 of equivalence classes generated by all $x \in M$

$$\big\{[x]_\approx \;\big|\; x \in M\big\}$$

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 partition 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    def union_find(M, R):
2        P = { frozenset({x}) for x in M }
3        for x, y in R:
4            Sx = find(x, P)
5            Sy = find(y, P)
6            if Sx != Sy:
7                P -= { Sx,  Sy }
8                P |= { Sx | Sy }
9        return P
10
11   def arb(S):
12       for x in S:
13           return x
14
15   def find(x, P):
16       return arb({ S for S in P if x in S })
```

Figure 9.1: A naive implementation of the union-find algorithm.

What we have sketched in the previous example is known as the union-find algorithm. Figure 9.1 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 $\approx_R$ 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 3 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 arb takes a set S as its argument and returns an arbitrary element from S.

4. 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.

9.1.1 A Tree-Based Implementation

The implementation shown in Figure 9.1 is not very efficient. The problem is the computation of the union

$$\texttt{Sx} \mid \texttt{Sy}.$$

If the sets Sx and Sy are represented as ordered 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

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

The reason is that every element of Sx has to be inserted into Sy and this insertion has a complexity of $\mathcal{O}\big(\log_2(\texttt{\#Sy})\big)$. 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 changing the parent pointer of ry to point to rx merges the sets Sx and Sy.

```
1   def find(x, Parent):
2       if Parent[x] == x:
3           return x
4       return find(Parent[x], Parent)
5
6   def union_find(M, R):
7       Parent = { x: x for x in M }
8       for x, y in R:
9           root_x = find(x, Parent)
10          root_y = find(y, Parent)
11          if root_x != root_y:
12              Parent[root_y] = root_x
13      Roots = { x for x in M if Parent[x] == x }
14      return [{y for y in M if find(y, Parent) == r} for r in Roots]
```

Figure 9.2: A tree-based implementation of the union-find algorithm.

Figure 9.2 on page 148 shows an implementation of this idea. In this implementation, the parent pointers are represented using the dictionary `Parent`.

1. The function `find` takes a node `x` and the dictionary `Parent` that represents the parent pointers. For a node `y` the expression `Parent[y]` returns the parent node of `y`. 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`. As I did not want to use `frozenset`s, this partition is represented by a list of sets.

   The dictionary[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 $\langle x, y \rangle$ from the binary relation `R`. In line 9 and 10 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 13 computes the set of all nodes that are at the root of some tree. Then, for every root `R` of a tree, line 14 computes the set of nodes corresponding to this tree. These sets are collected in a list, which is then returned.

9.1.2 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 attach this tree to the tree rooted at `parentY` via the assignment

```
parent[parentX] = parentY
```

otherwise we should use the assignment

```
parent[parentY] = parentX.
```

In order to be able to distinguish these cases, we store the height of the tree rooted at node `n` in the dictionary `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 9.3 on page 150. 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 `union_find` 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 27: We can speed up the implementation previously shown if the set `M` has the form

$$\texttt{M} = \{0, 1, 2, \cdots, n-1\} \quad \text{where } n \in \mathbb{N}.$$

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

```
def union_find(M, R):
    Parent = { x: x for x in M }
    Height = { x: 1 for x in M }
    for x, y in R:
        root_x = find(x, Parent)
        root_y = find(y, Parent)
        if root_x != root_y:
            if Height[root_x] < Height[root_y]:
                Parent[root_x] = root_y
            elif Height[root_x] > Height[root_y]:
                Parent[root_y] = root_x
            else:
                Parent[root_y]  = root_x
                Height[root_x] += 1
    Roots = { x for x in M if Parent[x] == x }
    return [{y for y in M if find(y, Parent) == r} for r in Roots]
```

Figure 9.3: A more efficient version of the union-find algorithm.

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

Exercise 28: For a natural number h, assume that t_h is a tree of height h that has been constructed with the union-find algorithm discussed in this section and, furthermore, assume that this tree has the smallest number of nodes that is possible for a tree of height h. Define c_h as the number of nodes in a tree t_h, set up an equation for c_h and solve this equation. ◇

9.1.3 Packaging the Union-Find Algorithm as a Class

Later, when we discus the problem of constructing a minimum spanning tree, 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 9.4 on page 151.

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

```
class UnionFind:
    def __init__(self, M):
        self.mParent = { x: x for x in M }
        self.mHeight = { x: 1 for x in M }

    def find(self, x):
        p = self.mParent[x]
        if p == x:
            return x
        return self.find(p)

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.mHeight[root_x] < self.mHeight[root_y]:
                self.mParent[root_x] = root_y
            elif self.mHeight[root_x] > self.mHeight[root_y]:
                self.mParent[root_y] = root_x
            else:
                self.mParent[root_y]  = root_x
                self.mHeight[root_x] += 1

def partition(M, R):
    UF = UnionFind(M)
    for x, y in R:
        UF.union(x, y)
    Roots = { x for x in M if UF.find(x) == x }
    return [{y for y in M if UF.find(y) == r} for r in Roots]
```

Figure 9.4: The class `unionFind`.

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`.

   (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`.

   This method is only needed to test the implementation of the class `UnionFind`.

9.2 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 a dense jungle, it is cheapest to build the connecting lines alongside existing roads. However, it is not necessary to have connection lines at every road. The task is to compute the set of roads that have to be equipped with connection lines so that all cities are connected and the overall cost of the connection is minimal.

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.

9.2.1 Basic Definitions

We start with the definition of an undirected graph, which is a set of vertices that are connected by undirected edges. The formal definition follows.

Definition 21 (Undirected Graph) An undirected graph is a pair $\mathcal{G} = \langle \mathbb{V}, \mathbb{E} \rangle$ 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 and therefore $\{x, y\} = \{y, x\}$, we have

   $$\{x, y\} \in \mathbb{E} \quad \text{if and only if} \quad \{y, x\} \in \mathbb{E}.$$

   Hence, if x is connected to y then y is also connected to x.

   We will further assume that if $\{x, y\}$ is an edge, then $x \neq y$, i.e. we do not admit loops.

Sometimes, the edges of an undirected graph have an associated weight. This leads to the notion of a weighted undirected graph that is given next.

Definition 22 (Weighted Graph) A weighted undirected graph is a triple $\langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ such that

1. $\langle \mathbb{V}, \mathbb{E} \rangle$ is an undirected graph.
2. $\|\cdot\| : \mathbb{E} \to \mathbb{N}$ 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. ◇

Given an undirected graph $\mathcal{G} = \langle \mathbb{V}, \mathbb{E} \rangle$, a path P in $\mathcal{G}$ is a list of the form

$$P = [x_0, x_1, x_2, \cdots, x_n]$$

such that we have :

$$\{x_i, x_{i+1}\} \in \mathbb{E} \quad \text{for all } i = 0, \cdots, n-1.$$

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 $P = [x_0, x_1, \cdots, x_n]$ connects the nodes x_0 and x_n. The weight of a path is defined as the sum of the weights of all of its edges.

$$\big\|[x_0, x_1, \cdots, x_n]\big\| := \sum_{i=0}^{n-1} \big\|\{x_i, x_{i+1}\}\big\|.$$

A graph is connected if for every $x, y \in \mathbb{V}$ there is a path connecting x and y, i.e. we have

$$\forall x, y \in \mathbb{V} : \exists P \in \mathbb{P} : \big(P[0] = x \wedge P[\texttt{len}(P) - 1] = y\big).$$

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

(a) the corresponding graph is connected **and**

(b) removing any edge from $\mathbb{E}$ would result in a graph that is no longer connected.

These conditions can be restated: A tree is a connected graph that does not have a cycle. Here, a cycle is a path of the form $[x_0, x_1, \cdots, x_n]$ where $n > 0$ and $x_0 = x_n$.

Definition 23 (Weight of a Tree) We define the weight of a tree as the sum of the weights of its edges, i.e. if the Graph $\mathcal{G} = (\mathbb{V}, \mathbb{E}, \|\cdot\|)$ is a tree, then

$$\texttt{weight}(\mathcal{G}) := \sum_{\{x,y\} \in \mathbb{E}} \big\|\{x, y\}\big\|. \qquad \diamond$$

Proposition 24 Assume that $\mathcal{T} = (\mathbb{V}, \mathbb{E})$ is an undirected graph that does not contain a cycle. Then $\mathcal{T}$ is a tree if and only if $\#\mathbb{E} = \#\mathbb{V} - 1$ holds. Here, $\#\mathbb{E}$ denotes the number of edges, while $\#\mathbb{V}$ denotes the number of vertices.

Proof: We will show the following claims:

(a) If the number of edges $\#\mathbb{E}$ is greater than $\#\mathbb{V} - 1$, then $\mathcal{T}$ has to contain a cycle.

(b) If the number of edges $\#\mathbb{E}$ is smaller than $\#\mathbb{V} - 1$, then $\mathcal{T}$ cannot be connected.

(c) If the number of edges $\#\mathbb{E}$ is equal to $\#V - 1$, then $\mathcal{T}$ is a tree.

We prove all of these claims simultaneously by induction on $n := \#V$.

B.C.: $n = 1$.

If the graph contains only 1 vertex, we have $\mathbb{V} = \{x\}$ and hence the set of edges $\mathbb{E}$ must be empty. Therefore we have

$$\#\mathbb{V} = 1, \quad \#\mathbb{E} = 0, \quad \text{and therefore} \quad \#\mathbb{E} = \#\mathbb{V} - 1.$$

Hence the first two claims are vacuously true because their precondition is never satisfied. Furthermore, in that case the graph trivially is a tree and thus the third claim is true.

I.S.: $1, 2, \cdots, n - 1 \mapsto n$

In this case we pick any edge $\{x, y\}$ from $\mathbb{E}$ and define a new graph $\mathcal{T}'$ that results from $\mathcal{T}$ by removing the edge $\{x, y\}$ from $\mathbb{E}$:

$$\mathcal{T}' := \big\langle \mathbb{V}, \mathbb{E} \backslash \big\{\{x, y\}\big\} \big\rangle$$

Next, we define A to be the set of those vertices that are connected to x in $\mathcal{T}'$, B to be the set of vertices that are connected to y in $\mathcal{T}'$, and C to contain those vertices that are neither connected to x nor to y:

(a) $A := \{z \in \mathbb{V} \mid z \text{ is connected to } x \text{ in } \mathcal{T}'\}$,

(b) $B := \{z \in \mathbb{V} \mid z \text{ is connected to } y \text{ in } \mathcal{T}'\}$, and

(c) $C := \{z \in \mathbb{V} \mid z \text{ is neither connected to neither } x \text{ nor to } y\}$.

The sets A and B have to be disjoint, for if there were a vertex z such that z were connected to both x and y, then the graph T would contain a cycle. If $\{u, v\} \in \mathbb{E}$, then there are three cases:

(a) u is connected to x. Because $\{u, v\} \in \mathbb{E}$ this implies that then v is also connected to x and hence both u and v are in A.

(b) u is connected to y. Then v is also connected to y and hence u and v are in B.

(c) u is neither connected to x nor to y. Then v can't be connected to x or y either. Therefore, both u and v are in the set C.

Accordingly, we can split the set $\mathbb{E}$ of edges into three disjoint parts:

(a) $\mathbb{E}_A := \big\{\{u, v\} \in \mathbb{E} \mid u \in A \wedge v \in A\big\}$,

(b) $\mathbb{E}_B := \big\{\{u, v\} \in \mathbb{E} \mid u \in B \wedge v \in B\big\}$,

(c) $\mathbb{E}_C := \big\{\{u, v\} \in \mathbb{E} \mid u \notin A \cup B \wedge v \notin A \cup B\big\}$.

Then, we can define three graphs $\mathcal{T}_A$, $\mathcal{T}_B$, and $\mathcal{T}_C$ as follows:

(a) $\mathcal{T}_A = \langle A, \mathbb{E}_A \rangle$,

(b) $\mathcal{T}_B = \langle B, \mathbb{E}_B \rangle$,

(c) $\mathcal{T}_C = \langle C, \mathbb{E}_C \rangle$,

As the graph $\mathcal{T}$ does not contain a cycle, the graphs $\mathcal{T}_A$, $\mathcal{T}_B$, and $\mathcal{T}_C$ can't contain a cycle either, since they are subgraphs of $\mathcal{T}$. Next, we define $a := \#A$, $b := \#B$, and $c := \#C$. Since $a < n$, $b < n$, and $c < n$ by induction hypothesis the claim holds true for $\mathcal{T}_A$, $\mathcal{T}_B$, and $\mathcal{T}_C$. Let us assume that $\#\mathbb{E}_A > a - 1$. By induction hypothesis we could then conclude that $\mathcal{T}_A$ contains a cycle. This would imply that also $\mathcal{T}$ contains a cycle and since this would contradict our assumption, we can conclude that

$$\#\mathbb{E}_A \leq a - 1.$$

In a similar way we must also have

$$\#\mathbb{E}_B \leq b - 1 \quad \text{and} \quad \#\mathbb{E}_C \leq c - 1.$$

Now there are three cases.

(a) Case: $C = \emptyset$, $\#\mathbb{E}_A = a - 1$, and $\#\mathbb{E}_B = b - 1$.
Then by induction hypothesis $\mathcal{T}_A$ and $\mathcal{T}_B$ are trees. In that case, $\mathcal{T}$ is a tree also because the edge $\{x, y\}$ connects the nodes from A and B. Since $\#\mathbb{E} = \#\mathbb{E}_A + \#\mathbb{E}_B + 1$ and $n = a + b$ we have

$$\#\mathbb{E} = (a - 1) + (b - 1) + 1 = n - 1 = \#\mathbb{V} - 1.$$

Hence in this case the claims (a) and (b) are trivially true as their assumptions are not given, while the last claim is true because $\mathcal{T}$ is a tree. $\surd$

(b) Case: $C = \emptyset$, $\#\mathbb{E}_A < a - 1$, or $\#\mathbb{E}_B < b - 1$.
Then by induction hypothesis either $\mathcal{T}_A$ or $\mathcal{T}_B$ is not connected. Of course, then $\mathcal{T}$ is not connected either. Furthermore, since we have $\#\mathbb{E}_A \leq a - 1$ and $\#\mathbb{E}_B \leq b - 1$ it follows that

$$\#\mathbb{E} = \#\mathbb{E}_A + \#\mathbb{E}_B - 1 < (a - 1) + (b - 1) = n - 1 = \#\mathbb{V} - 1.$$

Therefore in this case claim (a) and (c) are trivially true and (b) is true because $\mathcal{T}$ is not connected. $\surd$

(c) Case $C \neq \emptyset$.
In that case the graph $\mathcal{T}$ can't be connected since the vertices in C are not connected to either x or y. Furthermore, we have

$$\#\mathbb{E}_A + \#\mathbb{E}_B + \#\mathbb{E}_C \leq a + b + c - 3.$$

Since we have

$$\mathbb{E} = \mathbb{E}_A \uplus \mathbb{E}_B \uplus \mathbb{E}_C \uplus \big\{\{x, y\}\big\}$$

and, furthermore, $n = a + b + c$, we know that

$$\#\mathbb{E} = \#\mathbb{E}_A + \#\mathbb{E}_B + \#\mathbb{E}_C + 1.$$

Therefore it follows that $\mathbb{E} \leq n - 2 < n - 1 = \#\mathbb{V} - 1$.

Therefore in this case claim (a) and (c) are trivially true and (b) is true because $\mathcal{T}$ is not connected. $\surd$

Hence the claim has been established in all three cases. □

9.2.2 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.

(a) 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.

(b) 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 known as a greedy algorithm.

(c) We stop when the number of edges is one less than the number of nodes since, according to the proposition in the previous section, 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 9.5 on page 156. We discuss this program next.

1. Initially, we import the module `union_find_oo`, which is the object oriented implementation of the union-find algorithm that has been presented in Section 9.1.3.

2. Furthermore, we import the module `heapq` as it provides us with a priority queue.

3. 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 pairs of the form

   $$\big(\texttt{w}, (\texttt{x}, \texttt{y})\big).$$

   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 $\langle \texttt{V}, \texttt{E}\rangle$. It is assumed that this graph is connected.

4. The function `mst` first creates the union-find data structure `UF` in line 5. Initially, every node in `UF` is an equivalence class all by itself, i.e. no nodes are connected.

5. The spanning tree constructed by the algorithm is stored in the variable `MST`. The spanning tree is represented by the set of its weighted edges.

```
1   import heapq         as hq
2
3   def mst(V, E):
4       UF  = UnionFind(V)
5       MST = set()          # minimum spaning tree
6       H   = []             # empty priority queue for edges
7       for edge in E:
8           hq.heappush(H, edge)
9       while True:
10          w, (x, y) = hq.heappop(H)
11          root_x = UF.find(x)
12          root_y = UF.find(y)
13          if root_x != root_y:
14              MST.add((w, (x, y)))
15              UF.union(x, y)
16              if len(MST) == len(V) - 1:
17                  return MST
```

Figure 9.5: Kruskal's algorithm.

6. Next, the set of weighted edges `E` is turned into a priority queue. We use the weights of the edges as priorities. To this end, we create an empty priority queue `H` in line 7. The `for`-loop in line 8-9 inserts all edges of the set `E` into this priority queue.

7. The `while`-loop in line 10 checks for every edge (x, y) whether `x` and `y` are already connected. This is the case if both `x` and `y` are part of the same tree in the union-find data structure `UF`. In order to check whether `x` and `y` are part of the same tree we have to compute the roots of these trees. If these roots are identical, then `x` and `y` are part of the same tree and hence they are already connected.

8. If `x` and `y` are not yet connected, the edge $(w, (x, y))$ is added to the spanning tree and the trees containing `x` and `y` are connected by calling the function `union`.

9. The algorithm terminates once the tree `MST` has `#V-1` edges, since we know from the previous exercise that in this case all nodes are connected by the edges in `MST`.

9.3 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 25 (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 26 (Path, Path Length) A path P in a digraph is a list of the form

$$P = [x_0, x_1, x_2, \cdots, x_n]$$

such that

$$\langle x_i, x_{i+1}\rangle \in \mathbb{E} \quad \text{holds for all } i = 0, 1, \cdots, n-1.$$

The set of all paths is denoted as $\mathbb{P}$. The length of a path is the sum of the length of all edges:

$$\big\|[x_0, x_1, \cdots, x_n]\big\| := \sum_{i=0}^{n-1} \big\|\langle x_i, x_{i+1}\rangle\big\|.$$

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

$$\mathbb{P}(v, w) := \big\{[x_0, x_1, \cdots, x_n] \in \mathbb{P} \;\big|\; x_0 = v \wedge x_n = w\big\}.$$

Now we are ready to state the shortest path problem.

Definition 27 (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 $\|p\| = n$. ◇

9.3.1 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 discovered this algorithm independently and published their results in 1958 and 1956, respectively. Figure 9.6 on page 158 shows the implementation of a variant of this algorithm in *Python*. This variant of the algorithm was suggested by Edward F. Moore [Moo59] in 1959.

1. The function `shortestPath(source, Edges)` is called with two arguments:

   (a) `source` is the start node. The task of the function `shortestPath(source, Edges)` is to compute the distances of all vertices from the node `source`.

   (b) `Edges` is a dictionary that encodes the set of edges of the graph. For every node x the value of `Edges`$[x]$ has the form

   $$\big[(y_1, l_1), \cdots, (y_n, l_n)\big].$$

   This list is interpreted as follows: For every $i = 1, \cdots, n$ there is an edge (x, y_i) pointing from x to y_i and this edge has the length l_i.

   For example, the dictionary `Edges` given below defines a simple digraph. In that digraph, there is an edge from node `"a"` to node `"c"` which has a length of 2 and there is another edge from node `"a"` to the node `"b"` that has a length of 9.

```
def shortest_path(source, Edges):
    Distance = { source: 0 }
    Fringe   = [ source ]
    while len(Fringe) > 0:
        u = Fringe.pop()
        for v, l in Edges[u]:
            dv = Distance.get(v, None)
            if dv == None or Distance[u] + l < dv:
                Distance[v] = Distance[u] + l
                if v not in Fringe:
                    Fringe = Fringe + [v]
    return Distance
```

Figure 9.6: The Bellman-Ford algorithm to solve the shortest path problem.

```
Edges = { 'a': [('c', 2), ('b', 9)],
          'b': [('d', 1)],
          'c': [('e', 5), ('g', 3)],
          'd': [('f', 2), ('e', 4)],
          'e': [('f', 1), ('b', 2)],
          'f': [('h', 5)],
          'g': [('e', 1)],
          'h': []
        }
```

The directed graph defined by this dictionary is shown in Figure 9.7. In this Figure, every node is labelled both with its name and its distance from the source node a.

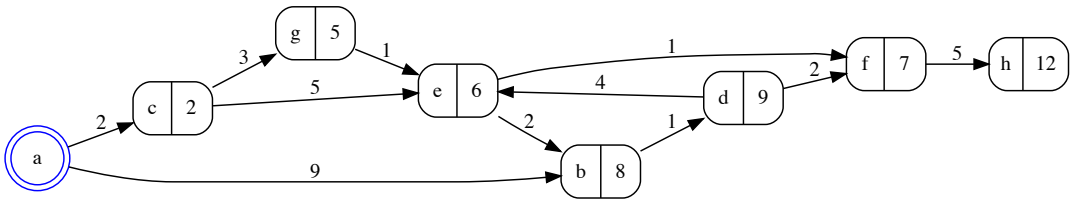

Figure 9.7: A directed graph.

2. The variable Distance is a dictionary. After the computation is finished, for every node x such that x is reachable from the node source, the value Distance$[x]$ records the length of the shortest path from source to x.

   The node source has distance 0 from the node source and initially this is all we know. Hence, the dictionary Distance is initialized as $\{\texttt{source} : 0\}$.

3. The variable Fringe is a list of those nodes that already have an estimate of their distance from the node source. Furthermore, if a node is in Fringe then this node might have neighbouring

node whose distance from source hasn't yet been computed. Initially we only know that the node source is connected to the node source. Therefore, we initialize the list Fringe as the list

$$[\texttt{source}].$$

4. Every iteration of the while-loop takes the first node u from the Fringe.

5. Next, we compute all nodes v that can be reached from u. For every node v that can be reached from u by an edge (u, v) we check whether reaching v from u results in a shorter path than previously known. There are two cases.

   (a) If there is an edge from node 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 $(\texttt{u}, \texttt{v})$.

   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, provided it isn't already a member of the Fringe.

6. The algorithm terminates when the list Fringe is empty because in that case we don't have any means left to improve our estimate of the distance function.

9.3.2 Dijkstra's Algorithm

The Bellman-Ford algorithm stores the nodes that are picked next to be examined in the list Fringe. This results in non-optimal behaviour of the algorithm: A node can be picked from Fringe and removed from the Fringe only to be reinserted into the Fringe later. In 1959 Edsger W. Dijkstra[2](1930 – 2002) [Dij59] published an algorithm that was more careful in picking the nodes from the Fringe. 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 that node from the Fringe that has the smallest distance to the node source. This has the effect that the algorithm visits nodes in increasing order of their distance from the node source. To implement this, we have to represent the variable Fringe as a priority queue where the priority of a node x is given by the distance of x to the node source. Figure 9.8 on page 160 shows an implementation of Dijkstra's algorithm in *Python*.

Since the priority queue that is implemented in *Python* as the module heapq does not provide a method to change the priority of an object that is already part of the queue, I have instead used the class Set, which I have implemented myself. The notebook Set.ipynb implements sets via AVL trees. In addition to the methods insert, delete, and isEmpty, the module Set provides the method pop. Given a set S, the call

$$S.\texttt{pop}()$$

returns the smallest element from the set S. Furthermore, this element is removed from S. Hence, objects of class Set can be used as priority queues. We proceed to discuss the program shown in Figure 9.8 line by line.

1. Distance is a dictionary mapping nodes to their estimated distance from the node source. If $d = \texttt{Distance}[x]$, then we know that there is a path of length d leading from source to x. However, we do not know whether there is a path shorter than d that also connects the source to the node x.

2. The program shown in Figure 9.8 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 Fringe and for which all

[2]Edsger Wybe Dijkstra received the 1972 Turing Award.

```
1   def shortest_path(source, Edges):
2       Distance = { source: 0 }
3       Visited  = { source }
4       Fringe   = Set()
5       Fringe.insert( (0, source) )
6       while not Fringe.isEmpty():
7           d, u = Fringe.pop()
8           for v, l in Edges[u]:
9               dv = Distance.get(v, None)
10              if dv == None or d + l < dv:
11                  if dv != None:
12                      Fringe.delete( (dv, v) )
13                  Distance[v] = d + l
14                  Fringe.insert( (d + l, v) )
15          Visited.add(u)
16      return Distance
```

Figure 9.8: Dijkstra's algorithm to solve the shortest path problem.

neighbouring nodes, i.e. those nodes y such that there is an edge (u, y), have been examined. We need the variable `Visited` in order to prove the correctness of the algorithm. This is explained in more detail later.

3. `Fringe` is a priority queue that contains pairs of the form (d, x), where x is a node and d is the distance that x has from the node `source`. This priority queue is implemented as a set, which in turn is represented by an ordered binary tree. The fact that we store the node x and the distance d as a pair (d, x) implies that the distances are used as priorities because pairs are compared lexicographically. Initially the only node that is known to be reachable from `source` is the node `source`. Hence `Fringe` is initialized as the set $\{(0, \texttt{source})\}$.

4. As long as the set `Fringe` is not empty, line 7 of the implementation removes that node u from the set `Fringe` that has the smallest distance d from the node `source`.

5. Next, all edges leading away from u are visited. If there is an edge (u, v) that has length l, then we check whether the node v has already a distance assigned. If the node v already has the distance `dv` assigned but the value $d + l$ is less than `dv`, then we have found a shorter path from `source` to v. This path leads from `source` to u and then proceeds to v via the edge (u, v).

6. If `v` had already been visited before and hence $\texttt{dv} = \texttt{Distance}[v]$ is defined, we have to update the priority of the v in the `Fringe`. The easiest way to do this is to remove the old pair $(\texttt{dv}, v)$ from the `Fringe` and replace this pair by the new pair $(d+l, v)$, because $d+l$ is the new estimate of the distance between `source` and v and $d + l$ is the new priority of v.

7. Once we have inspected all neighbours of the node u, u is added to the set of those nodes that have been `Visited`.

8. When the `Fringe` has been exhausted, the dictionary `Distance` contains the distances of every node that is reachable from the node `source`.

Theorem 28 (Distance Property of Dijkstra's Algorithm) If a node u is removed from the `Fringe` in line 7, then we have

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

and every node w such that $\texttt{sp}(w) \leq \texttt{sp}(u)$ satisfies

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

Proof: The claim is proven by induction on $n := \texttt{sp}(u)$.

Base case: $n = 0$.

The only node u satisfying $\texttt{sp}(u) = 0$ is the node $u = \texttt{source}$. Initially, the pair $(0, \texttt{source})$ is put onto the Fringe in line 5. The only way a pair on the Fringe could be changed is when a pair is removed in line 12 and then reinserted in line 14 with a smaller distance. As there is no smaller distance than 0, the pair $(0, \texttt{source})$ can not be altered. Therefore, when source is removed from the Fringe in line 8, we must have $d = 0$ and hence we have $\texttt{sp}(u) = d$. As all of our edges are assumed to be positive natural numbers, there can be no node w such that $\texttt{sp}[w] < 0$. Hence the second part of the claim is trivially true in this case.

Induction step: We assume that the claim holds for $\texttt{sp}[u] \in \{0, 1, \cdots, n-1\}$ and we prove that this implies that the claim also holds in case that $\texttt{sp}(u) = n$.

As $\texttt{sp}(u) = n$ there has to be a shortest path

$$P = [x_0, x_1, \cdots, x_{k-1}, x_k]$$

such that $x_0 = \texttt{source}$, $x_k = u$, and $\|P\| = n$. Since the lengths of all edges are positive natural numbers, this implies that the path $P' = [x_0, x_1, \cdots, x_{k-1}]$ has a length d' such that $d' < \texttt{sp}(u) = n$. By induction hypothesis this implies that

$$d' = \texttt{Distance}[x_{k-1}] = \texttt{sp}(x_{k-1}).$$

Immediately after $\texttt{Distance}[x_{k-1}]$ is computed in line 13, the pair (d', x_{k-1}) is put on the Fringe in line 14. Since $d' < d$, this pair is removed from the Fringe prior to the pair (d, u). After the pair (d', x_{k-1}) is removed from the Fringe in line 7, all edges starting in x_{k-1} are checked in line 8. Hence, the edge (x_{k-1}, x_k) has also been checked and, if $\texttt{Distance}[x_k]$ had been greater than d, it has been reduced to d at that time. Therefore,

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

A similar argument shows that for all nodes w such that $\texttt{sp}(w) \leq \texttt{sp}(u)$ we must have $\texttt{Distance}[w] = \texttt{sp}(w)$. □

Exercise 29: 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. ◇

9.3.3 Complexity

If a node u is removed from the priority queue Fringe, the node is added to the set Visited. The invariant that was just proven implies that in that case

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

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 $\texttt{Distance}[v]$ is still undefined or if the value $\texttt{Distance}[v]$ 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, the first insertion of a node 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 and then reinserting this node. 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

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

9.4 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[3], the set T of task is given as follows:

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

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 well. 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*.

Of course, the project of dressing up is trivial. However, complex projects can have ten thousands of tasks and even more dependencies between these tasks. For example, the US-lead invasion in the

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

second Irak war was a project consisting of more than $30\,000$ tasks. It is rumoured that the plan for the US invasion of the People's Republic of China contains an excess of 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.

In computer science, the utility program `make` uses topological sorting to execute the commands necessary to build a software system in the right order.

9.4.1 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 29 (Topological Sorting Problem, Solution of a Topological Sorting Problem)
A topological sorting problem is a finite directed graph $\langle T, D\rangle$ where

- T is a finte set that 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 $\textsc{Tsp}$. A solution of a topological sorting problem is a list S such that

(a) every task $t \in T$ appears exactly once in S, i.e. we have
$$\forall t \in T : \exists k \in \mathbb{N} : S[k] = t.$$

(b) If there are two tasks $t_1, t_2 \in T$ such that $t_1 \prec t_2$, then t_1 appears before t_2 in the list S, i.e. we have
$$\forall i, j \in \big\{1, \cdots, \texttt{len}(S)\big\} : \big(S[i] \prec S[j] \rightarrow i < j\big). \qquad \diamond$$

9.4.2 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. 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 D.
4. If T is not yet empty, go back to step 1.

When the algorithm terminates, the list `S` is a solution to the $\textsc{Tsp}$ $\langle T, D\rangle$.

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 a 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 9.9 on page 164.

```
1   def topo_sort(T, D):
2       Parents  = { t: set() for t in T }  # dictionary of parents
3       Children = { t: set() for t in T }  # dictionary of children
4       for s, t in D:
5           Children[s].add(t)
6           Parents [t].add(s)
7       Orphans = { t for t, P in Parents.items() if len(P) == 0 }
8       Sorted  = []
9       while len(T) > 0:
10          assert Orphans != set(), 'The graph is cyclic!'
11          t        = arb(Orphans)
12          Orphans -= { t }
13          T       -= { t }
14          Sorted.append(t)
15          for s in Children[t]:
16              Parents[s] -= { t }
17              if Parents[s] == set():
18                  Orphans.add(s)
19      return Sorted
```

Figure 9.9: 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`. This is the set of those tasks that have no `Parents`. We can start with any task from this set.
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. they depend on the completion of some other task, then the given TSP is cyclic and hence not solvable and therefore 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`.

9.4.3 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 9.9 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` also has a complexity of $\mathcal{O}\big(n \cdot \log_2(n)\big)$.

Chapter 10

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 volume of a three-dimensional object can be reduced to the computation of a multiple integral. However, if the object has a complicated surface, then computing the associated multiple integral exactly might be very difficult. 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 inherently difficult to predict. In many cases, their behaviour can 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 sometimes either impossible or at least very difficult to compute the 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.

10.1 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 10.1 on page 167 shows the resulting program.

```
1  def approximate_pi(n):
2      k = 0
3      for _ in range(n):
4          x = 2 * rnd.random() - 1
5          y = 2 * rnd.random() - 1
6          r = x * x + y * y
7          if r <= 1:
8              k += 1
9      return 4 * k / n
```

Figure 10.1: 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 $\langle$`x`, `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.

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 10.1: Results for the approximate computation of π using a Monte-Carlo simulation.

If we run this program, we get the results shown in table 10.1 on page 167. 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.

10.2 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 she 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 she wanted to stick with her first choice or whether, instead, she wanted to pick the remaining closed door. In order to better understand the problem you can try the animation at:

`http://math.ucsd.edu/~crypto/Monty/monty.html`

The question now is whether it is a good strategy to stick with the door chosen first or whether it is a better idea to switch doors. Mathematically, the reasoning is quite simple: As there are three doors initially and the probability that the car is behind any of them is the same, 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. Hence the best strategy is to switch the door.

Although the reasoning given above is straightforward, many people don't believe it, as is shown by this article. In order to convince them, the best thing is to run a Monte Carlo simulation. Figure 10.2 on page 169 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 *persistent strategy*.
2. The second strategy will always switch to the other door. This strategy is called the *wavering strategy*.

We discuss the implementation of the function `calculate_chances` 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 persistent strategy and how many cars are won by the wavering strategy.
2. The variable `success_persistent` counts the number of cars won by the persistent strategy.
3. The variable `success_wavering` counts the number of cars won by the wavering strategy.
4. The `for` loop extending from line 4 to line 12 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 wavering 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 persistent strategy wins the car. Therefore, we increment the variable `success_persistent` in this case.

   (b) If, instead, the car was placed behind the door that was neither chosen nor opened, then the wavering strategy wins the car. Hence, the variable `success_wavering` has to be incremented.

6. The function concludes by printing the results. Running the function for `n` equal to $1000\,000$ has yielded the following result:

```
The persistent strategy wins 333525 cars.
The wavering   strategy wins 666475 cars.
```

   This shows that, on average, the payoff from the wavering strategy is about twice as high as the payoff from the persistent strategy. This is just what we expect since $\frac{2}{3} = 2 \cdot \frac{1}{3}$.

```
1  def calculate_chances(n):
2      success_persistent = 0
3      success_wavering   = 0
4      for _ in range(n):
5          car    = rnd.randint(1, 3)
6          choice = rnd.randint(1, 3)
7          opened = rnd.choice(list({1,2,3} - {choice,car}))
8          last   = arb({ 1, 2, 3 } - { choice, opened })
9          if car == choice:
10             success_persistent += 1
11         if car == last:
12             success_wavering  += 1
13     print(f'The persistent strategy wins {success_persistent} cars.')
14     print(f'The wavering   strategy wins {success_wavering  } cars.')
```

Figure 10.2: A program to solve the Monty Hall problem.

10.3 Generation of Random Permutations

In this section we present an algorithm that can randomly permute a given list. Such a procedure can be likened to the shuffling of cards. The process is actually used for this purpose: When calculating the probability of winning in card games such as poker, the cards are shuffled using the algorithm presented here.

In order to permute a list $L = [x_1, x_2, \cdots, x_n]$ of length n we distinguish two cases:

1. The list L has length 1 and hence contains a single element, i.e. we have $L = [x]$. In this case, `permute`(L) returns the list unchanged:

   $$\#L = 1 \rightarrow \texttt{permute}(L) = L.$$

2. The length of L is bigger than 1. In this case, we randomly select an element to be the last in the permutation to be created. We remove this element from the list and then permute the remaining list. We append the initially selected element to the resulting permutation. Given two natural numbers a and b such that $a < b$ the *Python* Funktion

   $$\texttt{rnd.randint}(a, b)$$

   returns a random element from the set $\{a, \cdots, b\}$. Using this function we define:

   $$\#L = n \wedge n > 1 \wedge k := \texttt{rnd.randint}(0, n-1)$$
   $$\rightarrow \quad \texttt{permute}(L) = \texttt{permute}\big(L\texttt{[:}k\texttt{]} + L\texttt{[}k+1\texttt{:]}\big) + \texttt{[}L\texttt{[}k\texttt{]]}.$$

```
1  def permute(L):
2      if len(L) == 1:
3          return L
4      k = rnd.randint(0, len(L)-1)
5      return permute(L[:k] + L[k+1:]) + [L[k]]
```

Figure 10.3: Computation of random permutations of a list.

Figure 10.3 shows the implementation of this idea in Python. The method `permute` shown there creates a random permutation of the list L, which is passed as argument. The implementation directly implements the equations described above.

It can be shown that the algorithm presented above actually generates all permutations of a given list with the same probability. A proof of this claim can be found, for example, in Cormen et. al. [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.

[Flo64] Robert W. Floyd. "algorithm 245 - treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[For56] Lester R. Ford. Network flow theory. Technical report, RAND Corporation, Santa Monica, California, 1956.

[GS78] Leonidas L. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on the Foundations of Computer Science*, pages 8–21. IEEE, 1978.

[GS13] Hans-Peter Gumm and Manfred Sommer. *Einführung in die Informatik*. Oldenbourg-Verlag, 10th edition, 2013.

[Hoa61] C. Antony R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4:321, 1961.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, 2nd edition, 2006.

[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.

[Spi71] Murray R. Spiegel. *Calculus of Finite Differences and Difference Equations*. McGraw-Hill Education, 1971.

[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.

[Wil64] J. W. J. Williams. Algorithm 232 heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[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.

List of Figures

Index