Entwurf-Entwurf-Entwurf-Entwurf-Entwurf-Entwurf-Entwurf-Entwurf

Concise Algorithmics
or
Algorithms and Data Structures — The Basic Toolbox
or...

Kurt Mehlhorn and Peter Sanders

Entwurf-Entwurf-Entwurf-Entwurf-Entwurf-Entwurf-Entwurf-Entwurf

Foreword

Buy me not [25].

Contents

[amuse geule arithmetik. Bild von Al Chawarizmi] ⟸

Chapter 1

Amuse Geule: Integer Arithmetics

We introduce our readers into the design, analysis, and implementation of algorithms by studying algorithms for basic arithmetic operations on large integers. We treat addition and multiplication in the text and leave division and square roots for the exercises.

Integer arithmetic is interesting for many reasons:

- Arithmetic on long integers is needed in applications like cryptography, geometric computing, and computer algebra.
- We are familiar with the problem and know algorithms for addition and multiplication. We will see that the high school algorithm for integer multiplication is far from optimal and that much better algorithms exist.
- We will learn basic analysis techniques in a simple setting.
- We will learn basic algorithm engineering techniques in a simple setting.
- We will see the interplay between theory and experiment in a simple setting.

We assume that integers are represented as digit-strings (digits zero and one in our theoretical analysis and larger digits in our programs) and that two primitive operations are available: the addition of three digits with a two digit result (this is sometimes called a full adder) and the multiplication of two digits with a one digit result. We will measure the efficiency of our algorithms by the number of primitive operations executed.

We assume throughout this section that a and b are n-digit integers. We refer to the digits of a as a_{n-1} to a_0 with a_{n-1} being the most significant (also called leading) digit and a_0 being the least significant digit. [consistently replaced bit (was used
⟹ mixed with digit) by digit]

1.1 Addition

We all know how to add two integers a and b. We simply write them on top of each other with the least significant digits aligned and sum digit-wise, carrying a single bit
⟹ from one position to the next. [picture!]

c=0 : *Digit* // Variable for the carry digit
for $i := 0$ **to** $n-1$ **do** add a_i, b_i, and c to form s_i and a new carry c
$s_n = c$

We need one primitive operation for each position and hence a total of n primitive operations.

Lemma 1.1 *Two n-digit integers can be added with n primitive operations.*

1.2 Multiplication: The School Method

⟹ [picture!] We all know how to multiply two integers. In this section we will review the method familiar to all of us, in later sections we will get to know a method which is significantly faster for large integers.

The *school method* for integer multiplication works as follows: We first form partial products p_i by multiplying a with the i-th digit b_i of b and then sum the suitably aligned products $p_i \cdot 2^i$ to obtain the product of a and b.

p=0 : $\mathbb{N}$
for $i := 0$ **to** n **do** $p := a \cdot b_i \cdot 2^i + p$

Let us analyze the number of primitive operations required by the school method. We need n primitive multiplications to multiply a by b_i and hence a total of $n \cdot n = n^2$ primitive multiplications. All intermediate sums are at most $2n$-digit integers and hence each iterations needs at most $2n$ primitive additions. Thus there are at most $2n^2$ primitive additions.

Lemma 1.2 *The school method multiplies two n-digit integers with no more than* $3n^2$ *primitive operations.*

n	T
40000	0.3
80000	1.18
160000	4.8
320000	20.34

Table 1.1: The running time of the school method for the multiplication of n-bit integers. The running time grows quadratically.

⟹ [todo: proper alignment of numbers in tables] Table 1.1 shows the execution time of the school method using a C++ implementation and 32 bit digits. The time given is the average execution time over ??? many random inputs on a ??? machine. The quadratic growth of the running time is clearly visible: Doubling n leads to a four-fold increase in running time. We can interpret the table in different ways:

(1) *We can take the table to confirm our theoretical analysis.* Our analysis predicts quadratic growth and we are measuring quadratic growth. However, we analyzed the number of primitive operations and we measured running time on a ??? computer.Our analysis concentrates on primitive operations on digits and completely ignores all book keeping operations and all questions of storage. The experiments show that this abstraction is a useful one. We will frequently only analyze the number of "representative operations". Of course, the choice of representative operations requires insight and knowledge. In Section 2.2 we will introduce a more realistic computer model to have a basis for abstraction. We will develop tools to analyze running time of algorithms on this model. We will also connect our model to real machines, so that we can take our analysis as a predictor of actual performance. We will investigate the limits of our theory. Under what circumstances are we going to concede that an experiment contradicts theoretical analysis?

(2) *We can use the table to strengthen our theoretical analysis.* Our theoretical analysis tells us that the running time grows quadratically in n. From our table we may conclude that the running time on a ??? is approximately $??? \cdot n^2$ seconds. We can use this knowledge to *predict* the running time for our program on other inputs. [todo: redo numbers.] Here are three sample outputs. For n =100 000, the running ⟸
time is 1.85 seconds and the ratio is 1.005, for $n = 1000$, the running time is 0.0005147 seconds and the ratio is 2.797, for $n = 200$, the running time is $3.3 \cdot 10^{-5}$ seconds and the ratio is 4.5, and for n =1 000 000, the running time is 263.6 seconds and the ratio is 1.433. We see that our predictions can be far off. We simply were too careless. We started with the assumption that the running is cn^2 for some constant c, estimated c, and then predicted. Starting from the assumption that the running time is $cn^2 + dn + e$

would have lead us to different conclusions. We need to do our theory more carefully[1] in order to be able to predict. Also, when we made the prediction that the running time is approximately $???\cdot 10^{-10}\cdot n^2$ seconds, we did not make any restrictions on n. However, my computer has a finite memory (albeit large) and this memory is organized into a complex hierarchy of registers, first and second level cache, main memory, and disk memory. The access times to the different levels of the memory differ widely and this will have an effect on running times. None of the experiments reported so far requires our program to use disk memory. We should analyze the space requirement of our program in order to be able to predict for how large a value of n the program is able "to run in core[2]". In our example, we ran into both traps. The prediction is off for small n because we ignored the linear and constant terms in the running time and the prediction is off for large n because our prediction ignores the effects of the memory hierarchy.

(3) *We can use the table to conjecture quadratic growth of the running time of our algorithm.* Again, we need to be careful. What are we actually conjecturing? That the running time grows quadratically on random numbers? After all, we ran the algorithm only on random numbers (and even on very few of them). That the running time grows quadratically in the worst case, i.e., that there are no instances that lead to higher than quadratic growth? That the running grows quadratically in the best case, i.e., that there are no instances that lead to less than quadratic growth? We see that we need to develop more concepts and a richer language.

[dropped checking for now. If we use it we should do more in that direction later, e.g., pqs, flows, sorting] ⟹

1.3 A Recursive Version of the School Method

We derive a recursive version of the school method. This will be our first encounter of the *divide-and-conquer paradigm*, one of the fundamental paradigms in algorithm design. We will also learn in this section that constant factors can make a significant difference.

Let a and b be our two n bit-integers which we want to multiply. Let $k=\lfloor n/2\rfloor$. We split a into two numbers a_1 and a_0; a_0 consists of the k least significant bits and a_1 consists of the $n-k$ most significant bits. Then

$$a = a_1\cdot 2^k + a_0 \quad \text{and} \quad b = b_1\cdot 2^k + b_0$$

[1] Maybe this is not what you wanted to read, but it is the truth.

[2] Main memory was called core memory in ancient times (when one of the authors studied computer science).

Figure 1.1: todo: visuzlization of the school method and its recursive variant.

and hence

$$a\cdot b = a_1\cdot b_1\cdot 2^{2k} + (a_1\cdot b_0 + a_0\cdot b_1)\cdot 2^k + a_0\cdot b_0\ .$$

This formula suggests the following algorithm for computing $a\cdot b$:

a) Split a and b into a_1, a_0, b_1, and b_0.

b) Compute the four products $a_1\cdot b_1$, $a_1\cdot b_0$, $a_0\cdot b_1$, and $a_0\cdot b_0$.

c) Add the suitably aligned products to obtain $a\cdot b$.

Observe that the numbers a_1, a_0, b_1, and b_0 are $\lceil n/2\rceil$-bit numbers and hence the multiplications in step (2) are simpler than the original multiplication if $\lceil n/2\rceil < n$, i.e., $n>1$. The complete algorithm is now as follows: To multiply 1-bit numbers, use our multiplication primitive, and to multiply n-bit numbers for $n\geq 2$, use the three step approach above. [picture!] ⟸

It is clear why this approach is called divide-and-conquer. We reduce the problem of multiplying $a\cdot b$ into some number of *simpler* problems of the same kind. A divide and conquer algorithm always consists of three parts: In the first part, we split the original problem into simpler problems of the same kind (our step (1)), in the second part we solve the simpler problems using the same method (our step (2)), and in the third part, we obtain the solution to the original problem from the solutions to the subproblems. The following program implements the divide-and-conquer approach to integer multiplication.

What is the connection of our recursive integer multiplication to the school method? It is really the same. Figure **??** shows that the products $a_1\cdot b_1$, $a_1\cdot b_0$, $a_0\cdot b_1$, and $a_0\cdot b_0$ are also computed by the school method. Knowing that our recursive integer multiplication is just the school method in disguise tells us that the recursive algorithms uses a quadratic number of primitive operations. Let us also derive this from first principles. This will allow us to introduce recurrence relations, a powerful concept for the analysis of recursive algorithm.

Lemma 1.3 *Let $T(n)$ be the maximal number of primitive operations required by our recursive multiplication algorithm when applied to n-bit integers. Then*

$$T(n) \leq \begin{cases} 1 & \text{if } n=1,\\ 4\cdot T(\lceil n/2\rceil) + 3\cdot 2\cdot n & \text{if } n\geq 2.\end{cases}$$

Proof: Multiplying two 1-bit numbers requires one primitive multiplication. This justifies the case $n = 1$. So assume $n \geq 2$. Splitting a and b into the four pieces a_1, a_0, b_1, and b_0 requires no primitive operations[3]. Each piece has at most $\lceil n/2 \rceil$ bits and hence the four recursive multiplications require at most $4 \cdot T(\lceil n/2 \rceil)$ primitive operations. Finally, we need three additions to assemble the final result. Each addition involves two numbers of at most $2n$ bits and hence requires at most $2n$ primitive operations. This justifies the inequality for $n \geq 2$. ∎

In Section 2.5 we will learn that such recurrences are easy to solve and yield the already conjectured quadratic execution time of the recursive algorithm. At least if n is a power of two we get even the same constant factors. [exercise: induction proof ⟹ for n power of two?] [some explanation how this introduces the concept for the ⟹ next section?]

1.4 Karatsuba Multiplication

⟹ [check names and citations]

In 1962 the Soviet mathematician Karatsuba [51] discovered a faster way of multiplying large integers. The running time of his algorithm grows like $n^{\log 3} \approx n^{1.58}$. The method is surprisingly simple. Karatsuba observed that a simple algebraic identity allows one to save one multiplication in the divide-and-conquer implementation, i.e., one can multiply n-bit numbers using only three(!!) multiplications of integers half the size.

The details are as follows. Let a and b be our two n bit-integers which we want to multiply. Let $k = \lfloor n/2 \rfloor$. We split a into two numbers a_1 and a_0; a_0 consists of the k least significant bits and a_1 consists of the $n-k$ most significant bits. Then

$$a = a_1 \cdot 2^k + a_0 \quad \text{and} \quad b = b_1 \cdot 2^k + b_0$$

and hence (the magic is in the second equality)

$$\begin{aligned} a \cdot b &= a_1 \cdot b_1 \cdot 2^{2k} + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 2^k + a_0 \cdot b_0 \\ &= a_1 \cdot b_1 \cdot 2^{2k} + ((a_1 + a_0) \cdot (b_1 + b_0) - a_1 \cdot b_1 - a_0 \cdot b_0) \cdot 2^k + a_0 \cdot b_0 \end{aligned}$$

At first sight, we have only made things more complicated. A second look shows that the last formula can be evaluated with only three multiplications, namely, $a_1 \cdot b_1$, $a_1 \cdot b_0$, and $(a_1 + a_0) \cdot (b_1 + b_0)$. We also need six additions. That is three more than in the recursive implementation of the school method. The key is that additions are cheap

[3] It will require work, but it is work that we do not account for in our analysis.

compared to multiplications and hence saving a multiplication more than outweighs three additional additions. We obtain the following algorithm for computing $a \cdot b$:

a) Split a and b into a_1, a_0, b_1, and b_0.

b) Compute the three products $p_2 = a_1 \cdot b_1$, $p_0 = a_0 \cdot b_0$, and $p_1 = (a_1 + a_0) \cdot (b_1 + b_0)$.

c) Add the suitably aligned products to obtain $a \cdot b$, i.e., compute $a \cdot b$ according to the formula $a \cdot b = p_2 \cdot 2^{2k} + (p_1 - p_2 - p_0) \cdot 2^k + p_0$.

The numbers a_1, a_0, b_1, b_0, $a_1 + a_0$, and $b_1 + b_0$ are $\lceil n/2 \rceil + 1$-bit numbers and hence the multiplications in step (2) are simpler as the original multiplication if $\lceil n/2 \rceil + 1 < n$, i.e., $n \geq 4$. The complete algorithm is now as follows: To multiply 3-bit numbers, use the school method, and to multiply n-bit numbers for $n \geq 4$, use the three step approach above.

Table **??** shows the running time of the Karatsuba method in comparison with the school method. We also show the ratio of the running times (last column) and the ratio to the running time in the preceding iteration, i.e., the ratio between the running time for $2n$-bit integers and the running time on n-bit integers. We see that the ratio is around three in case of the Karatsuba method (doubling the size of the numbers about triples the running time) and is about four in case of the school method (doubling the size of the numbers about quadruples the running time). The latter statement requires some phantasy. We also see that the Karatsuba-method looses on short integers but wins on very large integers.

The lessons to remember are:

- Better asymptotics ultimately wins. However, it may loose on small inputs.
- An asymptotically slower algorithm can be faster on small inputs. You may only be interested in small inputs.

It is time to derive the asymptotics of the Karatsuba method.

Lemma 1.4 *Let $T(n)$ be the maximal number of primitive operations required by the Karatsuba algorithm when applied to n-bit integers. Then*

$$T(n) \leq \begin{cases} 3n^2 & \text{if } n \leq 3, \\ 3 \cdot T(\lceil n/2 \rceil + 1) + 6 \cdot 2 \cdot n & \text{if } n \geq 4. \end{cases}$$

Proof: Multiplying two n-bit numbers with the school method requires no more than $3n^2$ primitive operations by Lemma 1.2. This justifies the first line. So assume $n \geq 4$. Splitting a and b into the four pieces a_1, a_0, b_1, and b_0 requires no primitive operations[4]. Each piece and the sums $a_0 + a_1$ and $b_0 + b_1$ have at most $\lceil n/2 \rceil + 1$

[4] It will require work, but it is work that we do not account for in our analysis.

n	T_K	T_K/T'_K	T_S	T_S/T'_S	T_K/T_S
80000	5.85	–	1.19	–	4.916
160000	17.51	2.993	4.73	3.975	3.702
320000	52.54	3.001	19.77	4.18	2.658
640000	161	3.065	99.97	5.057	1.611
1280000	494	3.068	469.6	4.698	1.052
2560000	1457	2.95	1907	4.061	0.7641
5120000	4310	2.957	7803	4.091	0.5523

Table 1.2: The running time of the Karatsuba and the school method for integer multiplication: T_K is the running time of the Karatsuba method and T_S is the running time of the school method. T'_K and T'_S denote the running time of the preceding iteration. For an algorithm with running time $T(n) = cn^\alpha$ we have $T/T' = T(2n)/T(n) = 2^\alpha$. For $\alpha = \log 3$ we have $2^\alpha = 3$ and for $\alpha = 2$ we have $2^\alpha = 4$. The table was produced on a 300 MHz SUN ULTRA-SPARC.

bits and hence the three recursive multiplications require at most $3 \cdot T(\lceil n/2 \rceil + 1)$ primitive operations. Finally, we need two additions to form $a_0 + a_1$ and $b_0 + b_1$ and four additions to assemble the final result. Each addition involves two numbers of at most $2n$ bits and hence requires at most $2n$ primitive operations. This justifies the inequality for $n \geq 4$. ∎

The techniques introduced in Section 2.5 allow us to conclude that $T(n) \leq ??? n^{\log 3})$.
⟹ [fill in constant factor and lower order terms!]

1.5 Implementation Notes

Karatsuba integer multiplication is superior to the school method for large inputs. In our implementation the superiority only shows for integers with more than several million bits. However, a simple refinement can improve that significantly. Since the school method is superior to Karatsuba for short integers we should stop the recursion earlier and switch to the school method for numbers which have at n_0 bits for some yet to be determined n_0. In this way we obtain a method which is never worse than either the school method or the naive Karatsuba algorithm.

What is a good choice for n_0? We can answer this question analytically and experimentally. The experimental approach is easier here: we can simply time the revised Karatsuba algorithm for different values of n_0 and then take the value which gives the
⟹ smallest running time. On a ??? the best results were obtained for $n_0 \approx 512$.[check]

[report on switching time in LEDA] Note that this number of bits makes the Karat- ⟸
suba algorithm useful to applications in crytography where multiplying numbers up to 2048 bits is the most time consuming operation in some approaches [**?**].

1.6 Further Findings

Is the Karatsuba method the fastest known method for integer multiplication? Much faster methods are known. The asymptotically most efficient algorithm is due to Schönhage and Strassen [84]. It multiplies n-bit integers in time $O(n \log n \log\log n)$. Their method is beyond the scope of this book. [more? the generalization of Karatzuba is a bit heavy.] ⟸

Chapter 2

Introduction

[Alan Turing und John von Neumann? gothische Kathedrale (Chartres)?] ⟸

When you want to become a sculptor, you first have to learn your basic trade. Where to get the right stones, how to move them, how to handle the chisel, erecting scaffolding... These things alone do not make you a famous artist but even if you are a really exceptional talent, it will be very difficult to develop if you do not find a craftsman who teaches you the basic techniques from the first minute.

This introductory chapter attempts to play a similar role. It introduces some very basic concepts that make it much simpler to discuss algorithms in the subsequent chapters.

We begin in Section 2.1 by introducing notation and terminology that allows us to argue about the complexity of alorithms in a concise way. We then introduce a simple abstract machine model in Section 2.2 that allows us to argue about algorithms independent of the highly variable complications introduced by real hardware. Section 2.3 than introduces a high level pseudocode notation for algorithms that is much more convenient than machine code for our abstract machine. Pseudocode is also more convenient than actual programming languages since we can use high level concepts borrowed from mathematics without having to worry about how exactly they can be compiled to run on actual hardware. The Pseudocode notation also includes annotations of the programs with specifications of what makes a consistent state of the system. Section 2.4 explains how this can be used to make algorithms more readable and easier to prove correct. Section 2.5 introduces basic mathematical techniques for analyzing the complexity of programs, in particular, for analyzing nested loops and recursive procedure calls. Section 2.6 gives an example why concepts from probability theory can help analyzing existing algorithms or designing new ones. Finally, Section 2.8 introduces graphs as a convenient language to describe many concepts in

discrete algorithmics.

2.1 Asymptotic Notation

The main purpose of algorithm analysis is to give performance guarantees, e.g, bounds on its running time, that are at the same time accurate, concise, general, and easy to understand. It is difficult to meet all this criteria. For example, the most accurate way to characterize the running time T of an algorithm is to view T as a mapping from the set of possible inputs I to the set of nonnegative numbers $\mathbb{R}_+$. This representation is so cumbersome that we only use it to define more useable notations. We first group many inputs into a single class of "similar" inputs. The most typical way to group inputs is by their size. For example, for integer arithmetics, we said that we have an input of size n if both input numbers could be coded with at most n bits each. Now we can assign a single number to the set of inputs with identical size by looking at the maximum, minimum, or average of the execution times.

worst case: $T(n) = \max\{T(i) : i \in I, \text{size}(i) = n\}$
best case: $T(n) = \min\{T(i) : i \in I, \text{size}(i) = n\}$
average case: $T(n) = \frac{1}{|I|}\sum_{i \in I, \text{size}(i)=n} T(i)$

In this book we usually try to bound the worst case execution time since this gives us the strongest performance guarantee. Comparing the best case and the worst case helps us to estimate how much the execution time can vary for different inputs in the same set. If this discrepancy is big, the average case may give more insight into the true performance of the algorithm. Section 2.6 gives an example.

We have already seen in Chapter 1 that we can easily get lost in irrelevant details during algorithm analysis even if we only look at the worst case. For example, if we want to design a multiplication algorithm for n-bit integers, we do not care very much whether an algorithm takes $7n^2 - 6n$ or $7n^2 - 2n - 42$ elementary operations. We may not even care whether the leading constant is seven or six since implementation details can imply much larger differences in actual run time. However, it makes a fundamental difference whether we have an algorithm whose running time is proportional to n^2 or proportional to $n^{1.58}$ — for sufficiently large n, the algorithm with $n^{1.58}$ will win over the quadratic algorithm regardless of implementation details. We would like to group functions with quadratic behavior into the same category whereas functions that are "eventually much smaller" should fall into another category. The following definitions help to make this concept of *asymptotic behaviour* precise. Let $f(n)$ and $g(n)$ denote

functions that map nonnegative integers to nonnegative real numbers.

$$O(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \quad (2.1)$$
$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\} \quad (2.2)$$
$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)) \quad (2.3)$$
$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\} \quad (2.4)$$
$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\} \quad (2.5)$$

$O(f(n))$ is the set of all functions that "eventually grow no faster than" $f(n)$ except for constant factors. Similarly, $\Omega(f(n))$ is the set of all functions that "eventually grow at least as fast as" $f(n)$ except for constant factors. For example, the Karatsuba algorithm for integer multiplication has worst case running time in $O\left(n^{1.58}\right)$ whereas the school algorithm has worst case running time in $\Omega\left(n^2\right)$ so that we can say that the Karatsuba algorithm is asymptotically faster than the school algorithm. The "little-o" notation $o(f(n))$ denotes the set of all functions that "grow stricly more slowly than" $f(n)$. Its twin $\theta(f(n))$ is rarely used and only shown for completeness.

Handling the definitions of asymtotic notation directly requires some manipulations but is relatively simple. Let us consider one example that [abhaken] rids us of ⟸ future manipulations for polynomial functions.

Lemma 2.1 *Let $p(n) = \sum_{i=0}^{k} a_i n^i$ denote any polynomial with $a_k > 0$. Then $p(n) \in \Theta\left(n^k\right)$.*

Proof: It suffices to show that $p(n) \in O\left(n^k\right)$ and $p(n) \in \Omega\left(n^k\right)$.

First observe that for $n > 0$,

$$p(n) \leq \sum_{i=0}^{k} |a_i| n^i \leq n^k \sum_{i=0}^{k} |a_i| \quad .$$

Hence, $\forall n > 0 : p(n) \leq (\sum_{i=0}^{k} |a_i|) n^k$, i.e., $p(n) \in O\left(n^k\right)$.

Let $A = \sum_{i=0}^{k-1} |a_i|$. For $n > 0$ we get

$$p(n) \geq a_k n^k - A n^{k-1} = \frac{a_k}{2} n^k + n^{k-1}(\frac{a_k}{2} n - A) \geq \frac{a_k}{2} n^k$$

for $n > 2A/a_k$, i.e., choosing $c = a_k/2$ and $n_0 = 2A/a_k$ in the definition of $\Omega(n^k)$, we see that $p(n) \in \Omega\left(n^k\right)$. ∎

Since asymptotic notation is used a lot in algorithm analysis, mathematical notation is stretched a little bit to allow treating sets of functions (like $O\left(n^2\right)$) similar to

ordinary functions. In particular, we often write $h = F$ when we mean $h \in F$. If h is a function, F and G are sets of functions and "$\circ$" is an operand like $+, \cdot, /, \ldots$ then $F \circ G$ is a shorthand for $\{f+g : f \in F, g \in G\}$ and $h \circ F$ stands for $\{h\} + F$. For example, when we do care about constant factor but want to get rid of *lower order terms* we write expressions of the form $f(n) + o(f(n))$ or, equivalently $(1+o(1))f(n)$.

Most of the time, we will not bother to use the definitions directly but we use a rich set of rules that allows us to manipulate expressions without much calculation.

Lemma 2.2

$$cf(n) = \Theta(f(n)) \text{ for any constant } c \tag{2.6}$$
$$f(n) + g(n) = \Omega(f(n)) \tag{2.7}$$
$$f(n) + g(n) = o(f(n)) \text{ if } g(n) = o(f(n)) \tag{2.8}$$

$\Longrightarrow$ *[more rules!]*

Exercise 2.1 Prove Lemma 2.2.

To summarize this section we want to stress that there are at least three orthogonal choices in algorithm analysis:

- What complexity measure is analyzed? Perhaps time is most important. But we often also consider space consumption, solution quality, (e.g., in Chapter 12). Many other measures may be important. For example, energy consumption of a computation, the number of parallel processors used, the amount of data transmitted over a network,...
- Are we interested in worst case, best case, or average case?
- Are we simplifying bounds using $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, or $\omega(\cdot)$?

Exercise 2.2 Sharpen Lemma 2.1 and show that $p(n) = a_k n^k + o(n^k)$.

2.2 Machine Model

In 1945 John von Neumann introduced a basic architecture of a computer [76]. The design was very simple in order to make it possible to build it with the limited hardware technology of the time. Hardware design has grown out of this in most aspects. However, the resulting programming model was so simple and powerful, that it is still the basis for most programming. Usually it turns out that programs written with

Figure 2.1: John von Neumann born Dec. 28 1903 in Budapest, died Feb. 8, 1957, Washington DC.

the model in mind also work well on the vastly more complex hardware of todays machines.

The variant of von Neumann's model we consider is the *RAM (random access machine)* model. The most important features of this model are that it is *sequential*, i.e., there is a single processing unit, and that it has a uniform memory, i.e., all memory accesses cost the same amount of time. The memory consists of cells $S[0]$, $S[1]$, $S[2]$, ... The "..." means that there are potentially infinitely many cells although at any point of time only a finite number of them will be in use.

The memory cells store "small" integers. In Chapter 1 we assumed that small means one or two bits. It is more convenient to assume that "reasonable" functions of the input size n can be stored in a single cell. Our default assumption will be that polynomials in n like n^2 or perhaps $100n^3$ are still reasonable. Lifting this restriction could lead to absurdly overoptimistic algorithms. For example by repeated squaring, we could generate a number with 2^n bits in n steps.[mehr zu den komplexitaets-theoretischen Konsequenzen hier oder in further findings?] We should keep in mind however, that that our model allows us a limited form of parallelism. We can perform simple operations on $\log n$ bits in constant time. $\Longleftarrow$

In addition to the main memory, there is a small number of *registers* $R_1, \ldots, R_k$. Our RAM can execute the following *machine instructions.*

$R_i := S[R_j]$ *loads* the content of the memory cell with index R_j into register R_i.

$S[R_j] := R_i$ *stores* register R_i in memory cell $S[R_j]$.

$R_i := R_j \odot R_\ell$ is a binary register operation where '$\odot$' is a placeholder for a variety of operations. *Arithmetic* operations can be the usual $+$, $-$, and $*$ but also

the bit-wise operations |, &, >> <<, and $\oplus$ for exclusive-or. Operations **div** and **mod** stand for integer division and remainder respectively. *Comparison* operations $\leq$, $<$, $>$, $\geq$ encode *true* as 1 and *false* as 0. *Logical* operations $\wedge$ and $\vee$ further manipulate the *truth values* 0 and 1. We may also assume that there are operations which interpret the bits stored in a register as floating point numbers, i.e., finite precision approximations of real numbers.

$R_i := \odot R_j$ is a *unary* operation using the operators $-$, $\neg$ (logical not), or $\tilde{}$ (bitwise not).

$R_i := C$ assigns a *constant* value to R_i.

$R_i \in randInt(C)$ assigns a *random* integer between 0 and $C-1$ to R_i.

JZ j, R_i continues execution at memory address j if register i is zero.

Each instruction takes a certain number of time steps to execute. The total execution time of a program is the sum of the execution time of all the executed instructions.

It is important to remember that the RAM model is an abstract model. One should not confuse it with physically existing machines. In particular, real machines have finite memory and a fixed number of bits per register (e.g., 64). Our assumption of word sizes and memories scaling with the input size can be viewed as an abstraction of the practical experience that word sizes are large enough for most purposes and that memory capacity is more limited by your bank account than by physical limitations of available hardware.

Our complexity model is also a gross oversimplification: Modern processors attempt to execute many instructions in parallel. How well this works depends on factors like data dependencies between subsequent operations. Hence, we cannot assign a fixed cost to an operation. This effect is particularly pronounced for memory accesses. The worst case time for a memory access from main memory can be hundreds of times slower than the best case time. The reason is that modern processors attempt to keep frequently used data in *caches* — small, fast memories close to the processors. How well caches works depends a lot on their architecture, the program, and the particular input.

We could attempt to introduce a very accurate cost model but we would probably miss our point. We would end up with a very complex model that is almost impossible to handle. Even a successful complexity analysis would be a monstrous formula depending on many parameters that change with every new processor generation. We therefore go to the other extreme and eliminate any model parameters by assuming that each instruction takes exactly one unit of time. The result is that constant factors in our model are quite meaningless — one more reason to stick to asymptotic analysis most of the time. On the implementation site, our gross oversimplification will

be mitigated by occasinal informal discussions of implementation tradeoffs. We will now also introduce a very simple model for memory hierarchies that will be used in Sections 5.7 and **??**.

External Memory

The external memory model is like the RAM model except that the fast memory S is limited in size to M words. Additionally, there is an external memory with unlimited size. There are special *I/O operations* that transfer B consecutive words between slow and fast memory. For example, the external memory could be a hard disk, M would then be the main memory size and B would be a block size that is a good compromise between low latency and high bandwidth. On current technology $M = 1$GByte and $B = 1$MByte could be realistic values. One I/O step would then be around 10ms which is 10^7 clock cycles of a 1GHz machine. With another setting of the parameters M and B, we could model the smaller access time differences between a hardware cache and main memory.

2.3 Pseudocode

Our RAM model is an abstraction and simplification of the machine programs executed on microprocessors. But the model is still too low level for an algorithms textbook. Our programs would get too long and hard to read. Our algorithms will therefore be formulated in pseudocode that is an abstraction and simplification of imperative programming languages like C, C++, Java, Pascal... combined with a liberal use of mathematical notation. We now describe the conventions used in this book and give a rough idea how these high level descriptions could be converted into RAM machine instructions. But we do this only to the extend necessary to grasp asymptotic behavior of our programs. This would be the wrong place to worry about compiler optimization techniques since a *real* compiler would have to target *real* machines that are much more complex. The syntax of our pseudocode is similar to Pascal [47] because this is typographically nicer for a book than the more widely known Syntax of C and its descendents C++ and Java.

A *variable declaration* "$v{=}x : T$" introduces a variable v of type T that is initialized to value x. For example, "*answer*$=42 : \mathbb{N}$". When the type of a variable is clear from the context we sometimes omit the declaration. We can also extend numeric types by values $-\infty$ and ∞. Similarly, we use the symbol $\perp$ to denote an undefined value which we sometimes assume to be distinguishable from a "proper" element of T.

An declaration "$a : \textbf{Array}\ [i..j]\ \textbf{of}\ T$" yields an *array* a consisting of $j-1+1$ *elements* of type T stored in $a[i]$, $a[i+1]$, ..., $a[j]$. Arrays are implemented as con-

tiguous pieces of memory. To find element $a[i]$, it suffices to know the starting address of a. For example, if register R_a stores the starting address of the array $a[0..k]$ and elements of a have unit size, then the instruction sequence "$R_1 := R_a + 42$; $R_2 := S[R_1]$" loads $a[42]$ into register R_2. Section 3.1 gives more details on arrays in particular if they have variable size.

Arrays and objects referenced by pointers can be **allocate** d and **dispose** d of. For example, p:= **allocate Array** $[1..n]$ **of** $\mathbb{R}$ allocates an array of p floating point numbers. **dispose** p frees this memory and makes it available for later reuse. **allocate** and **dispose** cut the single memory array S into disjoint pieces that accommodate all data. ⟹ These functions can be implemented to run in constant time.[more in appendix?] For all algorithms we present they are also space efficient in the sense that a program execution that needs n words of memory for dynmically allocated objects will only touch physical memory locations $S[0..O(n)]$.[1]

From mathematics we borrow the composite data structures tuples, sequences, and sets. *Pairs*, *Triples*, and, more generally, *tuples* are written in round brackets, e.g., $(3,1)$, $(3,1,4)$ or $(3,1,4,1,5)$. Since tuples only contain a constant number of elements, operations on them can be broken into operations on their constituents in an obvious way. *Sequences* store elements in a specified order, e.g., "s=$\langle 3,1,4,1\rangle$: Sequence **of** $\mathbb{Z}$" declares a sequence s of integers and initializes it to contain the numbers 3, 1, 4, and 1 in this exact order. Sequences are a natural abstraction for many data structures like files, strings, lists, stacks, queues,... but our default assumption is that a sequence s is synonymous to an array $s[1..|s|]$. In Chapter 3 we will learn many additional ways to represent sequences. Therefore, we later make extensive use of sequences as a mathematical abstraction with little further reference to implementation details. We extend mathematical notation for sets to a notation for sequences in the obvious way, e.g., $e \in s$ if e occurs somewhere in s or $\langle i^2 : 1 \le i \le 9\rangle = \langle 1,4,9,16,25,36,49,64,81\rangle$. The empty sequence is written as $\langle\rangle$.

Sets play a pivotal rule in the expressive power of mathematical language and hence we also use them in high level pseudocode. In particular, you will see declarations like "M=$\{3,1,4\}$: **set of** $\mathbb{N}$" that are analogous to array or sequence declarations. Sets are usually implemented as sequences.

Numerical expressions can be directly translated into a constant number of machine instructions. For example, the pseudocode statement $a := a + bc$ can be translated into the RAM instructions "$R_1 := R_b * R_c$; $R_a := R_a + R_1$" if R_a, R_b, and R_v stand for the registers storing a, b, and c respectively. From C we borrow the shorthands ⟹ ++ , and −− . etc.[+= etc needed?] Assignment is also allowed for composite objects. For example, "(a,b):= (b,a)" swaps a and b.

The conditional statement

[1] Unfortunately, no memory management routines are known that are space efficient for *all* possible sequences of allocations and deallocations. It seems that there can be a factor $\Omega(\log n)$ waste of space [**?**].

if C **then**
  I
else I'
stands for the instruction sequence
C; JZ *sElse*, R_c; I; JZ *sEnd*, R_0; I′
where C is a sequence of instructions evaluating the expression C and storing its result in register R_c, I is a sequence of instructions implementing the statement I, I′ implements I', *sElse* addresses[Pfeile einbauen] the first instruction in I′, *sEnd* addresses ⟸ the first instruction after I′, and R_0 is a register storing the value zero.

Note that the statements affected by the **then** part are shown by indenting them. There is no need for the proliferation of brackets observed in programming languages like C that are designed as a compromise of readability for humans and computers. Rather, our pseudocode is designed for readability by humans on the small pages of a book. For the same reason, a line break can replace a ';' for separating statements. The statement "**if** C **then** I" can be viewed as a shorthand for "**if** C **then** I **else** ;", i.e., an if-then-else with an empty else part.

The loop "**repeat** I **until** C is equivalent to the instruction sequence I; C; JZ *sI*, R_c where I is an instruction sequence implementing the pseudocode in I, C computes the condition C and stores its truth value in register R_c, and *sI* adresses the first instruction in I. To get readable and concise programs, we use many other types of loops that can be viewed as shorthands for repeat-loops. For example:[todo: nicer alignment] ⟸

while C **do** $I \equiv$	**if** C **then repeat** I; **until** $\neg C$
for $i := a$ **to** b **do** $I \equiv$	$i := a$; **while** $i \le b$ **do** I; i++
for $i := a$ **downto** b **step** s **do** $I \equiv$	$i := a$; **while** $i \ge b$ **do** I; $i := i - s$
for $i := a$ **to** ∞ **while** C **do** $I \equiv$	$i := a$; **while** C **do** I; i++
foreach $e \in s$ **do** $I \equiv$	**for** $i := 1$ **to** $\lvert s\rvert$ **do** $e := s[i]$; I

[do we need a break loop construct? (so far not, Dec28,2003)] How exactly ⟸ loops are translated into efficient code is a complicated part of compiler construction lore. For us it only matters that the execution time of a loop can be bounded by summing the execution times of each of its iterations including the time needed for evaluating conditions.

Often we will also use mathematical notation for sequences or sets to express loops implicitly. For example, assume the set A is represented as an array and s is its size. Then $A := \{e \in B : C(e)\}$ would be a shorthand for "$s := 0$; **foreach** $e \in B$ **if** $C(e)$ **then** $A[++s] = e$". Similarly, the use of the logical quantifiers $\forall$ and $\exists$ can implicitly describe loops. For example, $\forall e \in s : e > 3$ could be a shorthand for "**foreach** $e \in s$ **do if** $e \le 3$ **then return** 0 **endfor return** 1".

A subroutine with name *foo* is declared in the form "**Procedure** *foo*(D) I" where I is the body of the procedure and D is a sequence of variable declarations specifying

the parameters of *foo*. For example,
Procedure $foo(a : \mathbb{Z}; b : \textbf{Array}\ [1..n])\ b[42] := a; a := 0$
Our default assumption is that parameters are passed "by value", i.e., the program behaves as if *foo* gets a copy of the passed values. For example, after a call $foo(a,c)$, variable a would still have its old values. However, complex objects like arrays are passed "by reference" to ensure that parameter passing only takes constant time. For example after the call $foo(a,c)$ we would have $c[42] = 2$.[check everywhere] These conventions are similar to the conventions used by C. ⟹

As for variable declarations, we sometimes omit type declarations for parameters if they are unimportant or clear from the context. Sometimes we also declare parameters implicitly using mathematical notation. For example, the **Procedure** $bar(\langle a_1, \ldots, a_n \rangle)$ is passed an sequence of n elements of unimportant type.

Most procedure calls could be compiled into machine code by just substituting the procedure body for the procedure call and making appropriate assignments to copy the parameter values into the local variables of the precedure. Since the compiler subsequently has many opportunities for optimization, this is also the most efficient approach for small procedures and procedures that are only called from a single place. However, this substitution approach fails for *recursive* procedures that directly or indirectly call themselves — substitution would never terminate. The program therefore maintains a *recursion stack* r: One register R_r always points to the last valid entry on this stack. Before a procedure call, the calling procedure *caller* pushs its local variables and the return address on the stack. Parameter values are put into preagreed registers,[2] and finally *caller* jumps to the first instruction of the called routine *called*. When *called* executes the **return** statement, it finds the next instruction of *caller* in $S[R_r]$ and jumps to this address. After popping the return address and its local variables from the stack, *caller* can continue with its normal operation. Note that recursion is no problem with this scheme since each incarnation of a routine will have its own stack area for its local variables.

Functions are similar to procedures except that they allow the return statement to return a value. For example,

Function $factorial(n) : \mathbb{Z}$ **if** $n = 1$ **then return** 1 **else return** $n \cdot factorial(n-1)$

declares a recursive function that returns $n!$. [picture. with a factorial example showing the state of the machine, after several levels of recursion.] ⟹

Our pseudocode also allows a simple form of object oriented programming because this is needed to separate the interface and implementation of data structures. We will introduce our notation by an example. The definition

Class $Complex(x,y : Element)$ **of** *Number*

[2] If there are not enough registers, parameters can also be passed on the stack.

Number $r := x$
Number $i := y$
Function abs : *Number* **return** $\sqrt{r^2 + i^2}$
Operator $+(c' : Complex) : Complex$ **return** $Complex(r + c'.r, i + c'.i)$

gives a (partial) implementation of a complex number type that can use arbitrary numeric types for real and imaginary parts. Very often, our class names will begin with capital letters.[somewhere else? drop? true?] The real and imaginary parts are stored in the *member variables* r and i respectively. Now, the declaration "$c : Complex(2,3)$**of** $\mathbb{R}$" declares a complex number c initialized to $2 + 3i$, $c.i$ is the imaginary part, and $c.abs$ returns the absolute value of c. We also allow a notation that views operators as ordinary functions. The object itself ["this" needed anywhere?] plays the role of the first (left) operand and the remaining operands are passed in the parameter list. Our complex number type uses this feature to define the operator $+$. ⟸ ⟸

In general, the type after the **of** allows us to parameterize classes with types in a similar fashion as using the more elaborate template mechanism of C++ [Java??]. Note that in the light of this notation the previously mentioned types "*Set* **of** *Element*" and "*Sequence* **of** *Element*" are ordinary classes. The combination of a parameter list for the class and initializations of the member variables is a simple replacement for a *constructor* that ensures that objects are immediately created in a consistent state. ⟸

Many procedures and functions within classes will only need the state of the object and hence need no parameter list. Note that this simplifies the task to hide the representation of a class. For example if we would decide to change the representation of complex numbers to store them in polar coordinates (absolute value and angle) the function *abs* would change into a member variable whereas r and i would be implemented as functions but a program using *abs* would not have to be changed.[drop this discussion?] ⟸

Exercise 2.3 Translate the following pseudocode for finding all prime numbers up to n in RAM machine code. This algorithm is known as the sieve of Eratosthenes.

$a = \langle 1, \ldots, 1 \rangle$: **Array** $[2..n]$ **of** $\{0,1\}$
for $i := 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**
  if $a[i]$ **then for** $j := 2i$ **to** n **step** i **do** $a[j] := 0$
for $i := 2$ **to** n **do** **if** $a[i]$ **then** output "i is prime"

2.4 Designing Correct Programs

[do we need old values anywhere else?] ⟸

```
Function power(a : ℝ; n0 : ℕ) : ℝ
    assert ¬(a = 0 ∧ n0 = 0)                    // It is not so clear what 0^0 should be
    p=a : ℝ;   r=1 : ℝ;   n=n0 : ℕ
    while n > 0 do
        invariant p^n r = a^n0
        if n is odd then n−− ; r:= r · p        // invariant violated between assignments
        else (n, p):= (n/2, p · p)              // parallel assignment maintains invariant
    assert r = a^n0                             // This is a consequence of the invariant and n = 0
    return r
```

Figure 2.2: An algorithm that computes integer powers of real numbers.

When a program solves a problem, it massages the state of the system — the input state is gradually transformed into an output state. The program usually walks on a narrow path of consistent intermediate states that allow it to function correctly. To understand why a program works, it is therefore crucial to characterize what is a consistent state. Pseudocode is already a big advantage over RAM machine code since it shapes the system state from a sea of machine words into a collection of variables with well defined types. But usually this is not enough since there are consistency properties involving the value of several variables. We explain these concepts for the algorithm in Figure 2.2 that computes powers.

We can require certain properties of the system state using an **assert**-statement. In an actual program we would usually check the condition at run time and signal an error if it is violated. For our pseudocode however, an assertion has no effect on the computation — it just declares that we expect the given property to hold. In particular, we will freely use assertions that would be expensive to check. As in our example, we often need *preconditions*, i.e., assertions describing requirements on the input of a function. Similarly, *postconditions* describe the result of a function or the effect of a procedure on the system state. One can view preconditions and postconditions as a contract between the caller and the called routine: If the caller passes consistent paramters, the routine produces a result with guaranteed properties. For conciseness, we will use assertions sparingly assuming that certain "obvious" conditions are implicit from the textual description of the algorithm. Much more elaborate assertions may be required for high quality programs or even formal verification.

Some particularly important consistency properties should hold at many places in the program. We use two kinds of such *invariants*: A *loop invariant* holds before and after each loop iteration. Algorithm 2.2 and many of the simple algorithms explained in this book have a very simple structure: A couple of variables are declared

and initialized to establish the loop invariant. Then a main loop manipulates the state of the program. When the loop terminates, the loop invariant together with the termination condition of the loop imply that the correct result has been computed. The loop invariant therefore plays a pivotal role in understanding why a program works correctly. Once we understand the loop invariant, it suffices to check that the loop invariant is true initially and after each loop iteration. This is particularly easy, if only a small number of statemtents are executed between points where the invariant is maintained. For example, in Figure 2.2 the parallel assignment $(n, p) := (n/2, p \cdot p)$ helps to document that the invariant is only maintained after both assignments are executed.[forward references to examples?] ⟸

More complex programs encapsulate their state in objects whose consistent representation is also governed by invariants. Such *data structure invariants* are declared together with the data type. They are true after an object is constructed and they are preconditions and postconditions of all methods of the class. For example, in order to have a unique representation of a two-element set of integers we might declare
Class IntSet2 $a, b : \mathbb{N}$; **invariant** $a < b$ ...[forward references to examples?] ⟸
[more on programming by contract?] ⟸

2.5 Basic Program Analysis

Let us summarize what we have already learned about algorithm analysis. First, we abstract from the complications of a real machine to the simplfied RAM model. Now, in principle, we want to count the number of instructions executed for all possible inputs. We simplify further by grouping inputs by size and focussing on the worst case. Using asymptotic notation, we additionally get rid of constant factors and lower order terms. This coarsening of our view also allows us to look at upper bounds on the execution time rather than the exact worst case as long as the asymptotic result remains unchanged. All these simplifications allow us to analyze the pseudocode directly. Let $T(I)$ denote the worst case execution time of a piece of program I:

- $T(I; I') = T(I) + T(I')$.
- $T(\textbf{if } C \textbf{ then } I \textbf{ else } I') = O(T(C) + \max(T(I), T(I')))$.
- $T(\textbf{repeat } I \textbf{ until } C) = O\left(\sum_{i=1}^{k} T(i)\right)$ where k is the number of loop iterations, and where $T(i)$ is the time needed in the i-th iteration of the loop.

2.5.1 "Doing Sums"

On the first glance, only the rule for loops seems nontrivial. To simplify the expressions generated due to loops, we will need to manipulate sums. Sums also show up

when we make an average case analysis or when we are interested in the expected execution time of the randomized algorithms introduced in Section 2.6.

For example, the insertion sort algorithm introduced in Section 5.1[move that here?] has two nested loops. The outer loop counts i from 2 to n. The inner loop performs at most $i-1$ iterations. Hence, the total number of iterations of the inner loop is at most

$$\sum_{i=2}^{n}\sum_{j=2}^{i} 1 = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n}(i-1) = -n + \sum_{i=1}^{n} i = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} \quad . \quad (2.9)$$

Since the time for one execution of the inner loop is $O(1)$, we get a worst case execution time of $\Theta(n^2)$. All nested loops with an easily predictable number of iterations can be analyzed in an analogous fashion: Work your way inside out by repeatedly finding a closed form expression for the innermost loop. Using simple manipulations like $\sum_i ca_i = c\sum_i a_i$, $\sum_i (a_i + b_i) = \sum_i a_i + \sum_i b_i$, or $\sum_{i=2}^{n} a_i = -a_1 + \sum_{i=1}^{n} a_i$ one can often reduce the sums to very simple forms that can be looked up in a formulary. A small sample of such formulas can be found in Appendix A. Since we are often only interested in the asymptotic behavior, one can also get rid of constant factors or lower order terms quickly. For example, the chain of equations 2.9 could be rewritten

$$\sum_{i=2}^{n}(i-1) \le \sum_{i=2}^{n}(n-1) = (n-1)^2 \le n^2 \text{ or, for even } n,$$
$$\sum_{i=2}^{n}(i-1) \ge \sum_{i=n/2+1}^{n} n/2 = n^2/4 \quad .$$

[more tricks, like telescoping, etc? but then we need good examples here. Alternative: forward references to places where these tricks are actually needed]

2.5.2 Recurrences

In our rules for analyzing programs we have so far neglected subroutine calls. Subroutines may seem to be easy to handle since we can analyze the subroutine separately and then substitute the obtained bound into the expression for the running time of the calling routine. But this gets interesting for recursive calls. For example, for the recursive variant of school multiplication and n a power of two we obtained $T(1) = 1$, $T(n) = 6n + 4T(n/2)$ for the number of primitive operations. For the Karatzuba algorithm, the corresponding expression was $T(1) = 1$, $T(n/2) = 12n + 3T(n/2)$. In general, a *recurrence relation* defines a function (directly or indirectly) in terms of the same function using smaller arguments. Direct definitions for small parameter

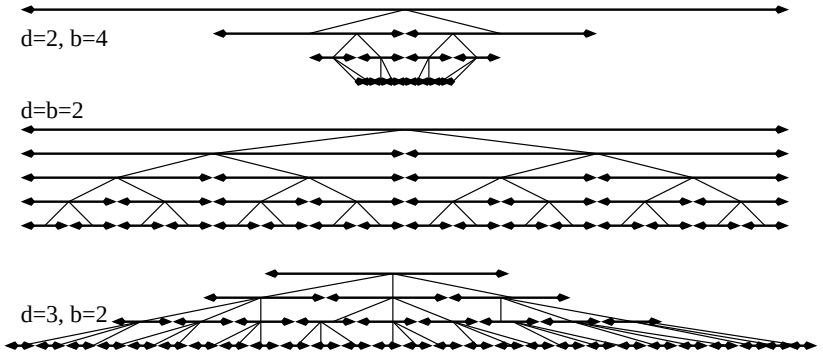


Figure 2.3: Examples for the three cases of the master theorem. Arrows give the size of subproblems. Thin lines denote recursive calls.

values make the function well defined. Solving recurrences, i.e., giving nonrecursive, closed form expressions for them is an interesting subject of mathematics.[sth more in appendix?] Here we focus on recurrence relations that typically emerge from divide-and-conquer algorithms. We begin with a simple case that already suffices to understand the main ideas:

Theorem 2.3 (Master Theorem (Simple Form)) *For positive constants a, b, and d, and $n = b^k$ for some integer k, consider the recurrence*

$$r(n) = \begin{cases} a & \text{if } n = 1 \\ cn + dT(n/b) & \text{else.} \end{cases}$$

then

$$r(n) = \begin{cases} \Theta(n) & \text{if } d < b \\ \Theta(n \log n) & \text{if } d = b \\ \Theta\left(n^{\log_b d}\right) & \text{if } d > b \end{cases}$$

Figure 2.3 illustrates the main insight behind Theorem 2.3: We consider the amount of work done at each level of recursion. If $d < b$, the amount of work in a recursive call is a constant factor smaller than in the previous level. Hence, the total work *decreases geometrically* with the level of recursion. Informally speaking, the *first* level of recursion already accounts for a constant factor of the execution time. If $d = b$, we have the same amount of work at *every* level of recursion. Since there are logarithmically many levels, the total amount of work is $\Theta(n \log n)$. Finally, if $d > b$

we have a geometrically *growing* amount of work in each level of recursion so that the *last* level accounts for a constant factor of the running time. Below we formalize this reasoning.

Proof: At level k, the subproblems have size one and hence we have to account cost $an = ad^k$ for the base case.

At the i-th level of recursion, there are d^i recursive calls for subproblems of size $n/b^i = b^k/b^i = b^{k-i}$. The total cost for the divide-and-conquer steps is

$$\sum_{i=0}^{k-1} d^i \cdot c \cdot b^{k-i} = c \cdot b^k \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i \quad .$$

Case $d = b$: we have cost $ad^k = ab^k = an = \Theta(n)$ for the base case and $cnk = cn\log_b n = \Theta(n\log n)$ for the divide-and-conquer steps.

Case $d < b$: we have cost $ad^k < ab^k = an = o(n)$ for the base case. For the cost of divide-and-conquer steps we use Formula A.9 for a geometric series and get

$$cn\frac{1-(d/b)^k}{1-d/b} < cn\frac{1}{1-d/b} = o(n) \text{ and}$$
$$cn\frac{1-(d/b)^k}{1-d/b} > cn = \Omega(n) \quad .$$

Case $d > b$: First note that

$$d^k = 2^{k\log d} = 2^{k\frac{\log b}{\log b}\log d} = b^{k\frac{\log d}{\log b}} = b^{k\log_b d} = n^{\log_b d} \quad .$$

Hence we get time $an^{\log_b d} = \Theta\left(n^{\log_b d}\right)$ for the base case. For the divide-and-conquer steps we use the geometric series again and get

$$cb^k\frac{(d/b)^k-1}{d/b-1} = c\frac{d^k-b^k}{d/b-1} = cd^k\frac{1-(b/d)^k}{d/b-1} = \Theta\left(d^k\right) = \Theta\left(n^{\log_b d}\right) \quad .$$

∎

How can one generalize Theorem 2.3 to arbitrary n? The simplest way is semantic reasoning. It is clear[3] that it is more difficult to solve larger inputs than smaller inputs and hence the cost for input size n will be less than the time needed when we round to $b^{\lceil \log_b n \rceil}$ — the next largest power of b. Since b is a constant, rounding can only affect the running time by a constant factor. Lemma [todo. The analysis of merge sort uses $C(n) \le C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \Rightarrow C(n) \le n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \le n \log n$] in the appendix makes this reasoning precise. ⟹

[3] Be aware that most errors in mathematical arguments are near occurrences of the word 'clearly'.

There are many further generalizations of the Master Theorem: We might break the recursion earlier, the cost for dividing and conquering may be nonlinear, the size of the subproblems might vary within certain bounds, the number of subproblems may depend on the input size, ... [how much of this are we doing? Theorem here, proof in appendix] ⟸
[linear recurrences? mention here, math in appendix?] ⟸
[amortization already here or perhaps in section of its own?] ⟸

***Exercise 2.4** Suppose you have a divide-and-conquer algorithm whose running time is governed by the recurrence

$$T(1) = a,\ T(n) = cn + \lceil \sqrt{n} \rceil\, T(\lceil n/\lceil \sqrt{n} \rceil \rceil) \quad .$$

Show that the running time of the program is $o(n \log\log n)$.

Exercise 2.5 Access to data structures is often governed by the following recurrence: $T(1) = a$, $T(n) = c + T(n/2)$.

$$T(1) = a,\ T(n) = c + T(\lceil n/b \rceil) \quad .$$

Show that $T(n) = o(\log n)$.

[Hier section on basic data structures and their "set theoretic" implementation? move sth from summary? Also search trees, sorting, priority queues. Then perhaps do binary search here?] [moved section Generic techniques into a summary chapter.] ⟸ ⟸

2.6 Average Case Analysis and Randomized Algorithms

[more? currently only one example. Introduce more concepts that are used more than once in this book?] ⟸

Suppose you are offered to participate in a tv game show: There are 100 boxes that you can open in an order of your choice. Box i contains an initially unknown amount of money m_i. No two boxes contain the same amount of money. There are strange pictures on the boxes and the show master is supposedly giving hints. The rules demand that after opening box i, you hold on to the maximum amount seen so far $\max_{j=1}^{i} m_i$, i.e., when you open a box that contains more money than you currently hold, you return what you hold and take the money in the current box. For example, if the first 10 boxes contain 3, 5, 2, 7, 1, 4, 6, 9, 2.5, 4.5 Euro respectively, you start with 3 Euro and swap them for 5 Euro in the second box. When you open the next box you see that it contains only 2 Euro and skip it. Then you swap the 5 Euro for

7 Euro in the fourth box and after opening and skipping boxes 5–7, you swap with the 9 Euro in the eigth box. The trick is that you are only allowed to swap money ten times. If you have not found the maximum amount after that, you lose everything. Otherwise you are allowed to keep the money you hold. Your Aunt, who is addicted to this show, tells you that only few candidates win. Now you ask yourself whether it is worth participating in this game. Is there a strategy that gives you a good chance to win? Are the hints of the show master useful?

Let us first analyze the obvious algorithm — you always follows the show master. The worst case is that he makes you open the boxes in order of increasing weight. You would have to swap money 100 times to find the maximum and you lose after box 10. The candidates and viewers, would hate the show master and he would be fired soon. Hence, worst case analysis is quite unrealistic in this case. The best case is that the show master immediately tells you the best box. You would be happy but there would be no time to place advertisements so that the show master would again be fired. Best case analysis is also unrealistic here.

So let us try average case analysis. Mathematically speaking, we are inspecting a sequence $\langle m_1, \ldots, m_n \rangle$ from left to right and we look for the number X of left-right maxima, i.e., those elements m_i such that $\forall j < i : m_i > m_j$. In Theorem 10.11 and Exercise 11.6 we will see algorithmic applications.

For small n we could use a brute force approach: Try all $n!$ permutations, sum up the number of left right maxima, and divide by $n!$ to obtain the average case number of left right maxima. For larger n we could use combinatorics to count the number of permutations c_i that lead to $X = i$. The average value of X is then $\sum_{i=1}^{n} c_i/n!$. We use a slightly different approach that simplfies the task: We reformulate the problem in terms of probability theory. We will assume here that you are familiar with basic concepts of probability theory but you can find a short review in Appendix A.2.

The set of all possible permutations becomes the probability space Ω. Every particular order has a probability of $1/n!$ to be drawn. The number of left-right maxima X becomes a random variable and the average number of left-right maxima X is the expectation $\mathrm{E}[X]$. We define indicator random variables $X_i = 1$ if m_i is a left-right maximum and $X_i = 0$ otherwise. Hence, $X = \sum_{i=1}^{n} X_i$. Using the linearity of expectation we get

$$\mathrm{E}[X] = \mathrm{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \mathrm{prob}(X_i = 1) \quad .$$

The probablity that the i-th element is larger than the elements preceeding it is simply $1/i$. We get

$$\mathrm{E}[X] = \sum_{i=1}^{n} \frac{1}{n} \quad .$$

Since this sum shows up again and again it has been given the name H_n — the n-th harmonic number. We have $H_n \leq \ln n + 1$, i.e., the average number of left-right maxima is much smaller than the worst case number.

Exercise 2.6 Show that $\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1$. Hint: first show that $\sum_{k=2}^{n} \frac{1}{k} \leq \int_1^n \frac{1}{x} dx$.

Let us apply these results to our game show example: For $n = 100$ we get less than $\mathrm{E}[X] < 6$, i.e., with ten opportunities to change your mind, you should have a quite good chance to win. Since most people lose, the "hints" given by the show master are actually contraproductive. You would fare much better by ignoring him. Here is the algorithm: Choose a random permutation of the boxes initially and stick to that order regardless what the show master says. The expected number of left-right maxima for this random choice will be below six and you have a good chance to win. You have just seen a *randomized algorithm*. In this simple case it was possible to permute the input so that an average case analysis applies. Note that not all randomized algorithms follow this pattern. In particular, for many applications there is no way to obtain an equivalent average case input from an arbitrary (e.g. worst case) input.[preview of randomized algorithms in this book. define Las Vegas and Monte Carlo? do we have any Monte Carlo algs?] ⟸

Randomized algorithms come in two main varieties. We have just seen a *Las Vegas algorithm*, i.e., an algorithms that always computes the correct output but where the run time is a random variable. In contrast, a *Monte Carlo* algorithm always has the same run time yet there is a nonzero probability that it gives an incorrect answer. [more?] For example, in Exercise 5.5 we outline an algorithm for checking whether ⟸ two sequences are permutations of each other that with small probability may may errorneously answer "yes".[primality testing in further findings?] By running a ⟸ Monte Calo algorithm several times using different random bit, the error probability can be made aribrarily small.

Exercise 2.7 Suppose you have a Las Vegas algorithm with expected execution time $t(n)$. Explain how to convert it to a Monte Carlo algorithm that gives no answer with probability at most p and has a deterministic execution time guarantee of $O\left(t(n) \log \frac{1}{p}\right)$. Hint: Use an algorithm that is easy to frustrate and starts from scratch rather than waiting for too long.

Exercise 2.8 Suppose you have a Monte Carlo algorithm with execution time $m(n)$ that gives a correct answer with probability p and a deterministic algorithm that verifies in time $v(n)$ whether the Monte Carlo algorithm has given the correct answer. Explain how to use these two algorithms to obtain a Las Vegas algorithm with expected execution time $\frac{1}{1-p}(m(n) + v(n))$.

Exercise 2.9 Can you give a situation where it might make sense to combine the two previous algorithm to transform a Las Vegas algorithm first into a Monte Carlo algorithm and then back into a Las Vegas Algorithm?

⟹ [forward refs to probabilistic analysis in this book]

2.7 Data Structures for Sets and Sequences

Building blocks for many algorithms are data structures maintaining sets and sequences. To underline the common aspects of these data structures, we introduce the most important operations already here. We give an abstract implementation in terms of mathematical notation. Chapters 3, 4, **??**, 6, and 7 explain more concrete implementations in terms of arrays, objects, and pointers.

In the following let e denotes an *Element* of a set sequence. $key(e)$ is the *key* of an element. For simplicity we assume that different elements have different keys. k denotes some other value of type *Key*; h is a *Handle* of an element, i.e., a reference or pointer to an element in the set or sequence.

Class *Set* **of** *Element*
Let M denote the set stored in the object
Procedure *insert*(e) $M := M \cup \{e\}$
Procedure *remove*(k) $\{e\} := \{e \in M : key(e) = k\}$; $M := M \setminus \{e\}$
Procedure *remove*(h) $e := h$; $M := M \setminus \{e\}$
Procedure *decreaseKey*(h,k) **assert** $key(h) \geq x$; $key(h) := k$
Function *deleteMin* $e := \min M$; $M := M \setminus \{e\}$ **return** e
Function *find*(k) $\{h\} := \{e : key(e) = k\}$; **return** h
Function *locate*(k) $h := \min\{e : key(e) \geq k\}$; **return** h

2.8 Graphs

[Aufmacher? Koenigsberger Brueckenproblem ist historisch nett aber didaktisch bloed weil es ein Multigraph ist und Eulertouren auch sonst nicht unbedingt gebraucht werden.] ⟹

Graphs are perhaps the single most important concept of algorithmics because they are a useful tool whenever we want to model objects (nodes) and relations between them (edges). Besides obvious applications like road maps or communication networks, there are many more abstract applications. For example, nodes could be tasks to be completed when building a house like "build the walls" or "put in the windows" and edges model precedence relations like "the walls have to be build before the windows can be put in". We will also see many examples of data structures where it

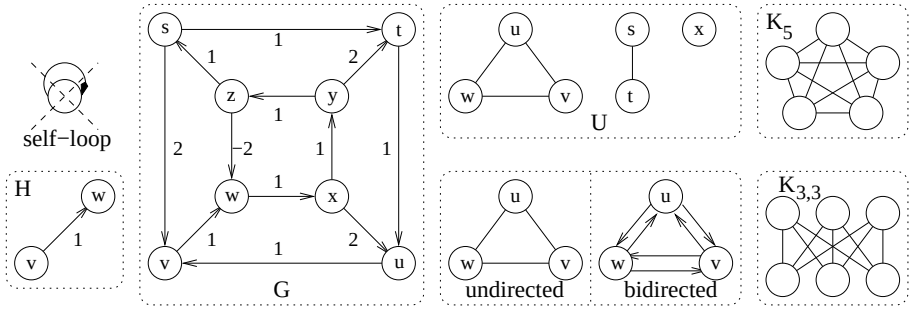


Figure 2.4: Some graphs.

makes sense to view objects as nodes and pointers as edges between the object storing the pointer and the object pointed to.

When humans think about graphs, they usually find it convenient to work with pictures showing nodes as bullets and lines and arrows as edges. For treating them algorithmically, a more mathematical notation is needed: A *directed graph* $G = (V,E)$ is described by a pair consisting of the *node set* (or *vertex* set) V and the *edge set* $E \in V \times V$. For example, Figure 2.4 shows the graph $G = (\{s,t,u,v,w,x,y,z\}, \{(s,t),(t,u),(u,v),(v,w),(w,x),(x,y),(y,z),(z,s),(s,v),(z,w),(y,t),(x,u)\})$. Throughout this book we use the convention $n = |V|$ and $m = |E|$ if no other definitions for n or m are given. An edge $e = (u,v) \in E$ represents a connection from u to v. We say that v is adjacent to u. The special case of a *self-loop* (v,v) is disallowed unless specifically mentioned. [do we need multigraphs and pseudographs?] [do we need head and tail?] ⟸

The the *outdegree* of a node v is $|\{(v,u) \in E\}|$ and its *indegree* is $|\{(u,v) \in E\}|$. ⟸ For example, node w in Figure 2.4 has indegree two and outdegree one.

A *bidirected graph* is a directed graph where for any edge (u,v) also the reverse edge (v,u) is present. An *undirected graph* can be viewed as a streamlined representation of a bidirected graph where we write a pair of edges (u,v), (v,u) as the two element set $\{u,v\}$. Figure 2.4 shows a three node undirected graph and its bidirected counterpart. Most graph theoretic terms for undirected graphs have the same definition as for their bidirected counterparts so that this sections concentrates on directed graphs and only mentions undirected graphs when there is something special about them. For example, the number of edges of an undirected graph is only half the number of edges of its bidirected counterpart. Since indegree and outdegree of nodes of undirected graphs are identical, we simple talk about their *degree*. Undirected graphs are important because directions often do not matter and because many problems are easier to solve (or even to define) for undirected graphs than for general directed graphs.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. Given $G = (V, E)$ and a subset $V' \subseteq V$, we get the subgraph *induced* by V' as $G' = (V', E \cap V'^2)$. In Figure 2.4, the node set $\{v, w\}$ in G induces the subgraph $H = (\{v, w\}, \{(v, w)\})$. A subset $E' \subseteq E$ of edges induces the subgraph $G' = (V, E')$.

Often additional information is associated with nodes or edges. In particular, we will often need *edge weights* or *costs* $c : E \to \mathbb{R}$ mapping edges to some numeric value. For example, the edge (z, w) in Figure 2.4 has weight $c((z, w)) = -2$. Note that edge $\{u, v\}$ of an undirected graph has a unique edge weight whereas in a bidirected graph
⟹ we can have $c((u, v)) \neq c((v, u))$.[needed?]

We have now seen quite a lot of definitions for one page of text. If you cannot wait how to implement this mathematics in a computer program, you can have a glimpse at Chapter 8 already now. But things also get more interesting here.

Perhaps the most central higher level graph theoretic concept is the connection of nodes by paths. A *path* $p = \langle v_0, \ldots, v_k \rangle$ of length k connects nodes v_0 and v_k if subsequent nodes in p are connected by edges in E, i.e, $(v_0, v_1) \in E$, $(v_1, v_2) \in E$, $\ldots$, $(v_{k-1}, v_k) \in E$. Sometimes a path is also represented by the resulting sequence of edges. For example, $\langle u, v, w \rangle = \langle (u, v), (v, w) \rangle$ is a path of length two in Figure 2.4.
⟹ If not otherwise stated[check] paths are assumed to be *simple*, i.e., all nodes in p
⟹ are different. Paths that need not be simple are called *walks*[check]. In Figure 2.4, $\langle z, w, x, u, v, w, x, y \rangle$ is such a non-simple path.

Cycles are paths that share the first and the last node. Analogously to paths, cycles are by default assumed to be simple, i.e, all but the first and the last nodes on the path are different. With $\langle s, t, u, v, w, x, y, z, s \rangle$, graph G in Figure 2.4 has a *Hamiltonian cycle*, i.e., a simple cycle visiting all nodes. A simple undirected cycle has at least three nodes since we also do not allow edges to be used twice.

The concept of paths and cycles helps us to define yet higher level concepts. A graph is *strongly connected*, if all its nodes are connected by some path, i.e,

$$G \text{ is strongly connected } \Leftrightarrow \forall u, v \in V : \exists \text{path } p : p \text{ connects } u \text{ and } v \ .$$

Graph G in Figure 2.4 is strongly connected. However, by removing edge (w, x), we get a graph without any directed cycles. In such a *directed acyclic graph (DAG)* every strongly connected component is a trivial component of size one.

For an undirected graph the "strongly" is dropped and one simply talks about connectedness. A related concept are *(strongly) connected components*, i.e., maximal subsets of nodes that induce a (strongly) connected subgraph. *Maximal* means that the set cannot be enlarged by adding more nodes without destroying the connectedness property. You should not mix up maxim*al* with maxim*um*. For example, a maxim*um* connected subgraph would be the largest connected component. For example, graph U in Figure 2.4 contains the connected components $\{u, v, w\}$, $\{s, t\}$, and $\{x\}$. All these subgraphs are maximal connected subgraphs but only $\{u, v, w\}$ is a maximum

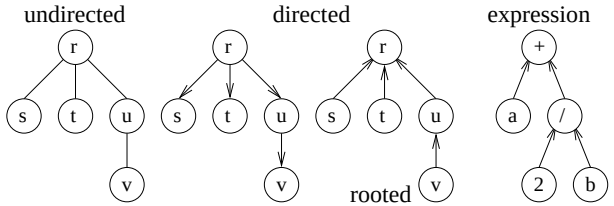


Figure 2.5: Different kinds of trees.

component. The node set $\{u, w\}$ induces a connected subgraph but it is not maximal and hence not a component.

An undirected graph is a *tree* if there is *exactly* one path between any pair of nodes.

Exercise 2.10 Prove that the following three properties of an undirected graph G are equivalent:

- G is a tree.
- G is connected and has exactly $n - 1$ edges.
- G is connected and contains no cycles.

Similarly, an undirected graph is a *forest* if there is *at most* one path between any pair of nodes. Note that each component of a forest is a tree.

When we look for the corresponding concept in directed graphs, the above equivalence breaks down. For example, a directed acyclic graph may have many more than $n - 1$ edges. A directed graph is a tree[check] if there is a *root* node r such that there ⟸
is exactly one path from r to any other node or if there is exactly one path from any other node to r. The latter version is called a *rooted tree*[check]. As you see in Fig- ⟸
ure 2.5, computer scientists have the peculiar habit to draw trees with the root at the top and all edges going downward.[do we need roots of undirected trees?] Edges ⟸
go between a unique *parent* and its *children*. Nodes with the same parent are *siblings*. Nodes without successors are *leaves*. Nonroot, nonleaf nodes are *interior* nodes. Consider a path such that u is between the root and another node v. Then u is an *ancestor* of v and v is a *descendant* of u. A node u and its descendants form a *subtree* rooted at v. For example, in Figure 2.5 r is the root; s, t, and v are leaves; s, t, and u are siblings because they are children of the same parent r; v is an interior node; r and u are ancestors of v; s, t, u and v are descendants of r; v and u form a subtree rooted at u.

We finish this barrage of definitions by giving at least one nontrivial graph algorithm. Suppose we want to test whether a directed graph is acyclic. We use the simple observation that a node v with outdegree zero cannot appear in any cycle. Hence, by deleting v (and its incoming edges) from the graph, we obtain a new graph G' that is acyclic if and only if G is acyclic. By iterating this transformation, we either arrive at the empty graph which is certainly acyclic, or we obtain a graph G^1 where every node has outdegree at least one. In the latter case, it is easy to find a cycle: Start at any node v and construct a path p by repeatedly choosing an arbitrary outgoing edge until you reach a node v' that you have seen before. Path p will have the form $(v,\ldots,v',\ldots,v')$, i.e., the part $(v',\ldots,v')$ of p forms a cycle. For example, in Figure 2.4 graph G has no node with outdegree zero. To find a cycle, we might start at node z and follow the walk $\langle z,w,x,u,v,w\rangle$ until we encounter w a second time. Hence, we have identified the cycle $\langle w,x,u,v,w\rangle$. In contrast, if edge (w,x) is removed, there is no cycle. Indeed, our algorithm will remove all nodes in the order w, v, u, t, s, z, y, x. [repeat graph G, with dashed (w,x), mark the walk $\langle w,x,u,v,w\rangle$, and give the order in which nodes are removed?] In Chapter 8 we will see how to represent graphs such that this algorithm can be implemented to run in linear time. See also Exercise 8.3.

Ordered Trees

Trees play an important role in computer science even if we are not dealing with graphs otherwise. The reason is that they are ideally suited to represent hierarchies. For example, consider the expression $a+2/b$. We have learned that this expression means that a and $2/b$ are added. But deriving this from the sequence of characters $\langle a,+,2,/,b\rangle$ is difficult. For example, it requires knowledge of the rule that division binds more tightly than addition. Therefore computer programs isolate this syntactical knowledge in *parsers* and produce a more structured representation based on trees. Our example would be transformed into the expression tree given in Figure 2.5. Such trees are directed and in contrast to graph theoretic trees, they are *ordered*, i.e., the order of successors matters. For example, if we would swap the order of '2' and 'b' in our example, we would get the expression $a+b/2$.

To demonstrate that trees are easy to process, Figure 2.6 gives an algorithm for evaluating expression trees whose leaves are numbers and whose interior nodes are binary operators (say +,-,*,/).

We will see many more examples of ordered trees in this book. Chapters 6 and 7 use them to represent fundamental data structures and Chapter 12 uses them to systematically explore a space of possible solutions of a problem.

[what about tree traversal, in/pre/post order?]


```
Function eval(r) : ℝ
    if r is a leaf then return the number stored in r
    else                                          // r is an operator node
        v1 := eval(first child of r)
        v2 := eval(second child of r)
        return v1 operator(r) v2                  // apply the operator stored in r
```

Figure 2.6: Evaluate an expression tree rooted at r.

Exercises

Exercise 2.11 Model a part of the street network of your hometown as a directed graph. Use an area with as many one-way streets as possible. Check whether the resulting graph is strongly connected.[in dfs chapter?]

Exercise 2.12 Describe ten sufficiently different applications that can be modelled using graphs (e.g. not car, bike, and pedestrian networks as different applications). At least five should not be mentioned in this book.

Exercise 2.13 Specify an n node DAG that has $n(n-1)/2$ edges.

Exercise 2.14 A *planar graph* is a graph that you can draw on a sheet of paper such that no two edges cross each other. Argue that a street network is *not* necessarily planar. Show that the graphs K_5 and K_{33} in Figure 2.4 are not planar.

2.9 Implementation Notes

[sth about random number generators, hardware generators]

Converting our pseudocode into actual programs may take some time and blow up the code but should pose little fundamental problems. Below we note a few language peculiarities. The Eiffel programming language has extensive support for assertions, invariants, preconditions, and postconditions. The Eiffel book [**?**] also contains a detailed discussion of the concept of programming by contract.

Our special values $\perp$, $-\infty$, and ∞ are available for floating point numbers. For other data types, we have to emulate these values. For example, one could use the smallest and largest representable integers for $-\infty$, and ∞ respectively. Undefined pointers are often represented as a null pointer. Sometimes, we use special values for

convenience only and a robust implementation should circumvent using them. [give example for shortest path or the like]

C++

Our pseudocode can be viewed as a concise notation for a subset of C++.

The memory management operations **allocate** and **dispose** are similar to the C++ operations `new` and `delete`. Note that none of these functions necessarily guarantees constant time execution. For example, C++ calls the default constructor of each element of a `new` array, i.e., allocating an array of n objects takes time $\Omega(n)$ whereas allocating an array n of `int`s "may" be done in constant time. In contrast, we assume that *all* arrays which are not explicitly initialized contain garbage. In C++ you can obtain this effect using the C functions `malloc` and `free`. However, this is a deprecated practice and should only be used when array initialization would be a severe performance bottleneck. If memory management of many small objects is performance critical, you can customize it using the `allocator` class of the C++ standard library.[more refs to good implementations? Lutz fragen. re-crosscheck with impl. notes of sequence chapter]

Our parameterization of classes using **of** is a special case of the C++-template mechanism. The parameters added in brackets after a class name correspond to parameters of a C++ constructor.

Assertions are implemented as C-macros in the include file `assert.h`. By default, violated assertions trigger a runtime error and print the line number and file where the assertion was violated. If the macro `NDEBUG` is defined, assertion checking is disabled.

Java

[what about genericity?] [what about assertions?] Java has no explicit memory mangement in particular. Rather, a *garbage collector* periodically recycles pieces of memory that are no longer referenced. While this simplfies programming enormously, it can be a performance problem in certain situations. Remedies are beyond the scope of this book.

2.10 Further Findings

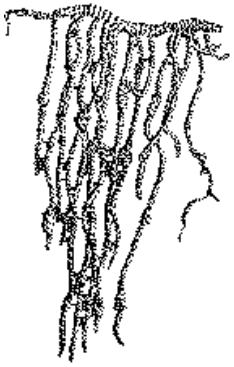
For a more detailed abstract machine model refer to the recent book of Knuth [58].

[results that say P=PSPACE for arbitrary number of bits?]

[verification, some good algorithms books,]

[some compiler construction textbooks R. Wilhelm fragen]

Chapter 3



Representing Sequences by Arrays and Linked Lists

Perhaps the world's oldest data structures were tablets in cuneiform script used more than 5000 years ago by custodians in Sumerian temples. They kept lists of goods, their quantities, owners and buyers. The left picture shows an example. Possibly this was the first application of written language. The operations performed on such lists have remained the same — adding entries, storing them for later, searching entries and changing them, going through a list to compile summaries, etc. The Peruvian quipu you see in the right picture served a similar purpose in the Inca empire using knots in colored strings arranged sequentially on a master string. Probably it is easier to maintain and use data on tablets than using knotted string, but one would not want to haul stone tablets over Andean mountain trails. Obviously, it makes sense to consider different representations for the same kind of logical data.

The abstract notion of a sequence, list, or table is very simple and is independent of its representation in a computer. Mathematically, the only important property is that the elements of a sequence $s = \langle e_0, \ldots, e_{n-1} \rangle$ are arranged in a linear order — in contrast to the trees and graphs in Chapters 7 and 8, or the unordered hash tables discussed in Chapter 4. There are two basic ways to specify elements of a sequence. One is to specify the index of an element. This is the way we usually think about arrays where $s[i]$ returns the i-th element of sequence s. Our pseudocode supports *bounded* arrays. In a *bounded* data structure, the memory requirements are known in advance, at the latest when the data structure is created. Section 3.1 starts with *unbounded arrays* that can adaptively grow and shrink as elements are inserted and removed. The analysis of unbounded arrays introduces the concept of *amortized analysis*.

Another way to specify elements of a sequence is relative to other elements. For example, one could ask for the successor of an element e, for the predecessor of an element e' or for the subsequence $\langle e,\ldots,e'\rangle$ of elements between e and e'. Although relative access can be simulated using array indexing, we will see in Section 3.2 that sequences represented using pointers to successors and predecessors are more flexible. In particular, it becomes easier to insert or remove arbitrary pieces of a sequence.

In many algorithms, it does not matter very much whether sequences are implemented using arrays or linked lists because only a very limited set of operations is needed that can be handled efficiently using either representation. Section 3.3 introduces stacks and queues, which are the most common data types of that kind. A more comprehensive account of the zoo of operations available for sequences is given in Section 3.4.

3.1 Unbounded Arrays

Consider an array data structure that besides the indexing operation $[\cdot]$, supports the following operations *pushBack*, *popBack*, and *size*.

$$\langle e_0,\ldots,e_n\rangle.\mathit{pushBack}(e) = \langle e_0,\ldots,e_n,e\rangle$$
$$\langle e_0,\ldots,e_n\rangle.\mathit{popBack} = \langle e_0,\ldots,e_{n-1}\rangle$$
$$\mathit{size}(\langle e_0,\ldots,e_{n-1}\rangle) = n$$

Why are unbounded arrays important? Often, because we do not know in advance how large an array should be. Here is a typical example: Suppose you want to implement
⟹ the Unix command `sort` for sorting[explain somewhere in intro?] the lines of a file. You decide to read the file into an array of lines, sort the array internally, and finally output the sorted array. With unbounded arrays this is easy. With bounded arrays, you would have to read the file twice: once to find the number of lines it contains and once to actually load it into the array.

In principle, implementing such an unbounded array is easy. We emulate an unbounded array u with n elements by a dynamically allocated bounded array b with $w \geq n$ entries. The first n entries of b are used to store the elements of b. The last $w-n$ entries of b are unused. As long as $w > n$, *pushBack* simply increments n and uses one of the previously unused entries of b for the new element. When $w = n$, the next *pushBack* allocates a new bounded array b' that is a constant factor larger (say a factor two). To reestablish the invariant that u is stored in b, the content of b is copied to the new array so that the old b can be deallocated. Finally, the pointer defining b is redirected to the new array. Deleting the last element with *popBack* looks even easier since there is no danger that b may become too small. However, we might waste a lot of space if we allow b to be much larger than needed. The wasted space can be kept small by shrinking b when n becomes too small. Figure 3.1 gives the complete pseudocode for an unbounded array class. Growing and shrinking is performed using the same utility procedure *reallocate*.

Amortized Analysis of Unbounded Arrays

Our implementation of unbounded arrays follows the algorithm design principle "make the common case fast". Array access with $[\cdot]$ is as fast as for bounded arrays. Intuitively, *pushBack* and *popBack* should "usually" be fast — we just have to update n. However, a single insertion into a large array might incur a cost of n. Exercise 3.2 asks you to give an example, where almost every call of *pushBack* and *popBack* is expensive if we make a seemingly harmless change in the algorithm. We now show that such a situation cannot happen for our implementation. Although some isolated procedure calls might be expensive, they are always rare, regardless of what sequence of operations we execute.

Lemma 3.1 *Consider an unbounded array u that is initially empty. Any sequence $\sigma = \langle\sigma_1,\ldots,\sigma_m\rangle$ of pushBack or popBack operations on u is executed in time $O(m)$.*

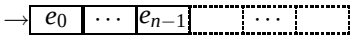
If we divide the total cost for the operations in σ by the number of operations, we get a constant. Hence, it makes sense to attribute constant cost to each operation. Such costs are called *amortized costs*. The usage of the term *amortized* is similar to its general usage, but it avoids some common pitfalls. "I am going to cycle to work every day from now on and hence it is justified to buy a luxury bike. The cost per ride is very small — this investment will amortize" Does this kind of reasoning sound familiar to you? The bike is bought, it rains, and all good intentions are gone. The bike has not amortized.

In computer science we insist on amortization. We are free to assign arbitrary *amortized costs* to operations but they are only correct if the sum of the amortized costs over any sequence of operations is never below the actual cost. Using the notion of amortized costs, we can reformulate Lemma 3.1 more elegantly to allow direct comparisons with other data structures.

Corollary 3.2 *Unbounded arrays implement the operation $[\cdot]$ in worst case constant time and the operations pushBack and popBack in amortized constant time.*

To prove Lemma 3.1, we use the *accounting method*. Most of us have already used this approach because it is the basic idea behind an insurance. For example, when you rent a car, in most cases you also have to buy an insurance that covers the ruinous costs you could incur by causing an accident. Similarly, we force all calls to *pushBack* and

```
Class UArray of Element
    Constant β=2 : ℝ+                                   // growth factor
    Constant α=β^2 : ℝ+                                 // worst case memory blowup
    w=1 : ℕ                                             // allocated size
    n=0 : ℕ          // current size. invariant n ≤ w
    b : Array [0..w−1] of Element                       // b →

    Operator [i : ℕ] : Element
        assert 0 ≤ i < n
        return b[i]

    Function size : ℕ return n

    Procedure pushBack(e : Element)                     // Example for n = w = 4:
        if n = w then                                   // b →
            reallocate(βn)                              // b →
        //For the analysis: pay insurance here.
        b[n] := e                                       // b →
        n++                                             // b →

    Procedure popBack                                   // Example for n = 5, w = 16:
        assert n > 0                                    // b →
        n−−                                             // b →
        if w ≥ αn ∧ n > 0 then                          // reduce waste of space
            reallocate(βn)                              // b →

    Procedure reallocate(w' : ℕ)                        // Example for w = 4, w' = 8:
        w := w'                                         // b →
        b' := allocate Array [0..w−1] of Element        // b' →
        (b'[0],...,b'[n−1]) := (b[0],...,b[n−1])        // b' →
        dispose b                                       // b →
        b := b'                                         // pointer assignment b →
```



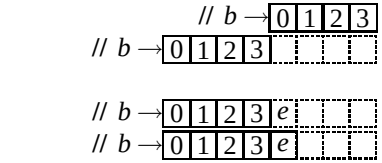


Figure 3.1: Unbounded arrays

popBack to buy an insurance against a possible call of *reallocate*. The cost of the insurance is put on an account. If a *reallocate* should actually become necessary, the responsible call to *pushBack* or *popBack* does not need to pay but it is allowed to use previous deposits on the insurance account. What remains to be shown is that the account will always be large enough to cover all possible costs.

Proof: Let m' denote the total number of elements copied in calls of *reallocate*. The total cost incurred by calls in the operation sequence σ is $O(m+m')$. Hence, it suffices to show that $m' = O(m)$. Our unit of cost is now the cost of one element copy.

For $\beta = 2$ and $\alpha = 4$, we require an insurance of 3 units from each call of *pushBack* and claim that this suffices to pay for all calls of *reallocate* by both *pushBack* and *popBack*. (Exercise 3.4 asks you to prove that for general β and $\alpha = \beta^2$ an insurance of $\frac{\beta+1}{\beta-1}$ units is sufficient.)

We prove by induction over the calls of *reallocate* that immediately after the call there are at least n units left on the insurance account.

First call of *reallocate***:** The first call grows w from 1 to 2 after at least one[two?] ⟸ call of *pushBack*. We have $n = 1$ and $3 - 1 = 2 > 1$ units left on the insurance account.

For the induction step we prove that $2n$ units are on the account immediately before the current call to *reallocate*. Only n elements are copied leaving n units on the account — enough to maintain our invariant. The two cases in which *reallocate* may be called are analyzed separately.

pushBack **grows the array:** The number of elements n has doubled since the last *reallocate* when at least $n/2$ units were left on the account by the induction hypothesis.[forgot the case where the last reallocate was a shrink.] The $n/2$ new ele- ⟸ ments paid $3n/2$ units giving a total of $2n$ units.

popBack **shrinks the array:** The number of elements has halved since the last *reallocate* when at least $2n$ units were left on the account by the induction hypothesis. ∎

Exercises

Exercise 3.1 (Amortized analysis of binary counters.) Consider a nonnegative integer c represented by an array of binary digits and a sequence of m increment and decrement operations. Initially, $c = 0$.

a) What is the worst case execution time of an increment or a decrement as a function of m? Assume that you can only work at one bit per step.

b) Prove that the amortized cost of increments is constant if there are no decrements.

c) Give a sequence of m increment and decrement operations that has cost $\Theta(m \log m)$.

d) Give a representation of counters such that you can achieve worst case constant time for increment and decrement. (Here you may assume that m is known in advance.)

e) Consider the representation of counters where each digit d_i is allowed to take values from $\{-1,0,1\}$ so that the counter is $c = \sum_i d_i 2^i$. Show that in this *redundant ternary* number system increments and decrements have constant amortized cost.

Exercise 3.2 Your manager asks you whether it is not too wasteful to shrink an array only when already three fourths of b are unused. He proposes to shrink it already when $w = n/2$. Convince him that this is a bad idea by giving a sequence of m *pushBack* and *popBack* operations that would need time $\Theta(m^2)$ if his proposal were implemented.

Exercise 3.3 (Popping many elements.) Explain how to implement the operation *popBack*(k) that removes the last k elements in amortized constant time independent of k. Hint: The existing analysis is very easy to generalize.

Exercise 3.4 (General space time tradeoff) Generalize the proof of Lemma 3.1 for general β and $\alpha = \beta^2$. Show that an insurance of $\frac{\beta+1}{\beta-1}$ units paid by calls of *pushBack* suffices to pay for all calls of *reallocate*.

***Exercise 3.5** We have not justified the relation $\alpha = \beta^2$ in our analysis. Prove that any other choice of α leads to higher insurance costs for calls of *pushBack*. Is $\alpha = \beta^2$ still optimal if we also require an insurance from *popBack*? (Assume that we now want to minimize the maximum insurance of any operation.)

Exercise 3.6 (Worst case constant access time) Suppose for a real time application you need an unbounded array data structure with *worst case* constant execution time for all operations. Design such a data structure. Hint: Store the elements in up to two arrays. Start moving elements to a larger array well before the small array is completely exhausted.

Exercise 3.7 (Implicitly growing arrays) Implement an unbounded array where the operation $[i]$ allows any positive index. When $i \geq n$, the array is implicitly grown to size $n = i+1$. When $n \geq w$, the array is reallocated as for *UArray*. Initialize entries that have never been written with some default value $\perp$.

Exercise 3.8 (Sparse arrays) Implement a bounded array with constant time for allocating the array and constant amortized time for operation $[\cdot]$. As in the previous exercise, a read access to an array entry that was never written should return $\perp$. Note that you cannot make any assumptions on the contents of a freshly allocated array. Hint: Never explicitly write default values into entries with undefined value. Store the elements that actually have a nondefault value in arbitrary order in a separate data structure. Each entry in this data structure also stores its position in the array. The array itself only stores references to this secondary data structure. The main issue is now how to find out whether an array element has ever been written.

3.2 Linked Lists

In this section we exploit the approach of representing sequences by storing pointers to successor and predecessor with each list element. A good way to think of such linked lists is to imagine a chain where one element is written on each link. Once we get hold of one link of the chain, we can retrieve all elements by exploiting the fact that the links of the chain are forged together. Section 3.2.1 explains this idea. In Section 3.2.2 we outline how many of the operations can still be performed if we only store successors. This is more space efficient and somewhat faster.

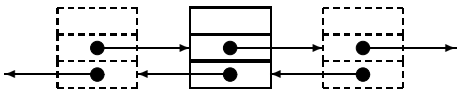
3.2.1 Doubly Linked Lists

Figure 3.2 shows the basic building block of a linked list. A list item (a link of a chain) stores one element and pointers to successor and predecessor. This sounds simple enough, but pointers are so powerful that we can make a big mess if we are not careful. What makes a consistent list data structure? We make a simple and innocent looking decision and the basic design of our list data structure will follow from that: The successor of the predecessor of an item must be the original item, and the same holds for the predecessor of a successor.

If all items fulfill this invariant, they will form a collection of cyclic chains. This may look strange, since we want to represent sequences rather than loops. Sequences have a start and an end, wheras loops have neither. Most implementations of linked lists therefore go a different way, and treat the first and last item of a list differently. Unfortunately, this makes the implementation of lists more complicated, more error-prone and somewhat slower. Therefore, we stick to the simple cyclic internal representation. Later we hide the representation from the user interface by providing a list data type that "simulates" lists with a start and an end.

For conciseness, we implement all basic list operations in terms of the single operation *splice* depicted in Figure 3.2. *splice* cuts out a sublist from one list and inserts

```
Type Handle = Pointer to Item
Function info(a : Handle) return a→e
Function succ(a : Handle) return a→next
Function pred(a : Handle) return a→prev

Class Item of Element                                  // one link in a doubly linked list
    e : Element
    next : Handle                                      //
    prev : Handle
    invariant next→prev=prev→next=this

    // Remove ⟨a,...,b⟩ from its current list and insert it after t
    // ...,a',a,...,b,b',...,t,t',...) ↦ (...,a',b',...,t,a,...,b,t',...)
    Procedure splice(a,b,t : Handle)
        assert b is not before a ∧ t ∉ ⟨a,...,b⟩
        // Cut out ⟨a,...,b⟩
        a' := a→prev
        b' := b→next
        a'→next := b'                                  //
        b'→prev := a'                                  //

        // insert ⟨a,...,b⟩ after t
        t' := t→next                                   //

        b→next := t'                                   //
        a→prev := t                                    //

        t→next := a                                    //
        t'→prev := b                                   //
```



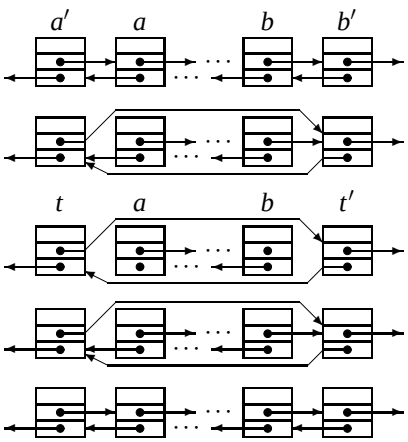


Figure 3.2: Low level list labor.

it after some target item. The target can be either in the same list or in a different list but it must not be inside the sublist.

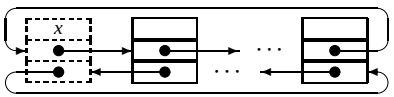
Since *splice* never changes the number of items in the system, we assume that there is one special list *freeList* that keeps a supply of unused elements. When inserting new elements into a list, we take the necessary items from *freeList* and when deleting elements we return the corresponding items to *freeList*. The function *checkFreeList* allocates memory for new items when necessary. We defer its implementation to Exercise 3.11 and a short discussion in Section 3.5.

It remains to decide how to simulate the start and end of a list. The class *List* in Figure 3.3 introduces a dummy item *h* that does not store any element but seperates the first element from the last element in the cycle formed by the list. By definition of *Item*, *h* points to the first "proper" item as a successor and to the last item as a predecessor. In addition, a handle *head* pointing to *h* can be used to encode a position before the first element or after the last element. Note that there are $n+1$ possible positions for inserting an element into an list with n elements so that an additional item is hard to circumvent if we want to code handles as pointers to items.

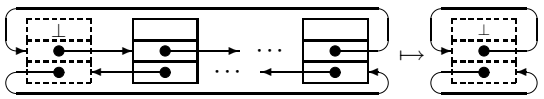
With these conventions in place, a large number of useful operations can be implemented as one line functions that all run in constant time. Thanks to the power of *splice*, we can even manipulate arbitrarily long sublists in constant time. Figure 3.3 gives typical examples.

The dummy header can also be useful for other operations. For example consider the following code for finding the next occurence of *x* starting at item *from*. If *x* is not present, *head* should be returned.

```
Function findNext(x : Element; from : Handle) : Handle
    h.e = x                                            // Sentinel
    while from→e≠ x do
        from := from→next
    return from
```



```
Class List of Element
    // Item h is the predecessor of the first element
    // Item h is the successor of the last element.
    h=( ⊥ this this ) : Item          // empty list ⟨⟩ with dummy item only
    // Simple access functions
    Function head() : Handle; return address of h// Pos. before any proper element

    Function isEmpty : {0,1}; return h.next = this                    // ⟨⟩?
    Function first : Handle; assert ¬isEmpty; return h.next
    Function last : Handle; assert ¬isEmpty; return h.prev

    // Moving elements around within a sequence.
    // (⟨...,a,b,c...,a',c',...⟩) ↦ (⟨...,a,c...,a',b,c',...⟩)
    Procedure moveAfter(b,a' : Handle) splice(b,b,a')
    Procedure moveToFront(b : Handle) moveAfter(b,head)
    Procedure moveToBack(b : Handle) moveAfter(b,last)

    // Deleting and inserting elements. // ⟨...,a,b,c,...⟩ ↦ ⟨...,a,c,...⟩
    Procedure remove(b : Handle) moveAfter(b, freeList.head)
    Procedure popFront remove(first)
    Procedure popBack remove(last)

    // ⟨...,a,b,...⟩ ↦ ⟨...,a,e,b,...⟩
    Function insertAfter(x : Element; a : Handle) : Handle
        checkFreeList        // make sure freeList is nonempty. See also Exercise 3.11
        a' := freeList.first
        moveAfter(a',a)
        a'↦e=x
        return a'

    Function insertBefore(x : Element; b : Handle) : Handle return insertAfter(e, pred(b))
    Procedure pushFront(x : Element) insertAfter(x, head)
    Procedure pushBack(x : Element) insertAfter(x, last)

    // Manipulations of entire lists
    // (⟨a,...,b⟩,⟨c,...,d⟩) ↦ (⟨a,...,b,c,...,d⟩,⟨⟩)
    Procedure concat(o : List)
        splice(o.first, o.last, head)

    // ⟨a,...,b⟩ ↦ ⟨⟩
    Procedure makeEmpty
        freeList.concat(this )    //
```



We use the header as a *sentinel*. A sentinel is a dummy element in a data structure that makes sure that some loop will terminate. By storing the key we are looking for in the header, we make sure that the search terminates even if x is originally not present in the list. This trick saves us an additional test in each iteration whether the end of the list is reached.

Maintaining the Size of a List

In our simple list data type it not possible to find out the number of elements in constant time. This can be fixed by introducing a member variable *size* that is updated whenever the number of elements changes. Operations that affect several lists now need to know about the lists involved even if low level functions like *splice* would only need handles of the items involved. For example, consider the following code for moving an element from one list to another:

```
// (⟨...,a,b,c...⟩,⟨...,a',c',...⟩) ↦ (⟨...,a,c...⟩,⟨...,a',b,c',...⟩)
Procedure moveAfter(b, a' : Handle; o : List)
    splice(b,b,a')
    size−−
    o.size++
```

Interfaces of list data types should require this information even if *size* is not maintained so that the data type remains interchangable with other implementations.

A more serious problem is that operations that move around sublists beween lists cannot be implemented in constant time any more if *size* is to be maintained. Exercise 3.15 proposes a compromise.

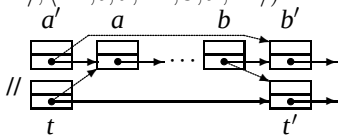
3.2.2 Singly Linked Lists

The many pointers used by doubly linked lists makes programming quite comfortable. Singly linked lists are the lean and mean sisters of doubly linked lists. *SItem*s scrap the *prev* pointer and only store *next*. This makes singly linked lists more space efficient and often faster than their doubly linked brothers. The price we pay is that some operations can no longer be performed in constant time. For example, we cannot remove an *SItem* if we do not know its predecessor. Table 3.1 gives an overview of constant time operations for different implementations of sequences. We will see several applications of singly linked lists. For example for hash tables in Section 4.1 or for mergesort in Section 5.2. In particular, we can use singly linked lists to implement free lists of memory managers — even for items of doubly linked lists.

We can adopt the implementation approach from doubly linked lists. *SItem*s form collections of cycles and an *SList* has a dummy *SItem h* that precedes the first proper element and is the successor of the last proper element. Many operations of *List*s can still be performed if we change the interface. For example, the following implementation of *splice* needs the *predecessor* of the first element of the sublist to be moved.

$//(\langle\ldots,a',a,\ldots,b,b'\ldots\rangle,\langle\ldots,t,t',\ldots\rangle)\mapsto(\langle\ldots,a',b'\ldots\rangle,\langle\ldots,t,a,\ldots,b,t',\ldots\rangle)$
Procedure *splice*(a',b,t : *SHandle*)

$$\begin{pmatrix} a'\!\rightarrow next \\ t\rightarrow next \\ b\rightarrow next \end{pmatrix} := \begin{pmatrix} b\rightarrow next \\ a'\!\rightarrow next \\ t\rightarrow next \end{pmatrix}$$



Similarly, *findNext* should not return the handle of the *SItem* with the next fit but its *predecessor*. This way it remains possible to remove the element found. A useful addition to *SList* is a pointer to the last element because then we can support *pushBack* in constant time. We leave the details of an implementation of singly linked lists to
⟹ Exercise 3.17. [move some exercises]

Exercises

Exercise 3.9 Prove formally that items of doubly linked lists fulfilling the invariant $next\rightarrow prev = prev\rightarrow next =$ **this** form a collection of cyclic chains.

Exercise 3.10 Implement a procudure *swap* similar to *splice* that swaps two sublists in constant time
$(\langle\ldots,a',a,\ldots,b,b',\ldots\rangle,\langle\ldots,c',c,\ldots,d,d',\ldots\rangle)\mapsto$
$(\langle\ldots,a',c,\ldots,d,b',\ldots\rangle,\langle\ldots,c',a,\ldots,b,d',\ldots\rangle)$.
Can you view *splice* as a special case of *swap*?

Exercise 3.11 (Memory mangagement for lists) Implement the function *checkFreelist* called by *insertAfter* in Figure 3.3. Since an individual call of the programming language primitive **allocate** for every single item might be too slow, your function should allocate space for items in large batches. The worst case execution time of *checkFreeList* should be independent of the batch size. Hint: In addition to *freeList* use a small array of free items.

Exercise 3.12 Give a constant time implementation for rotating a list right: $\langle a,\ldots,b,c\rangle\mapsto\langle c,a,\ldots,b\rangle$. Generalize your algorithm to rotate sequence $\langle a,\ldots,b,c,\ldots,d\rangle$ to $\langle c,\ldots,d,a,\ldots,b\rangle$ in constant time.

Exercise 3.13 (Acyclic list implementation.) Give an alternative implementation of *List* that does not need the dummy item *h* and encodes *head* as a null pointer. The interface and the asymptotic execution times of all operations should remain the same. Give at least one advantage and one disadvantag of this implementation compared to the algorithm from Figure 3.3.

Exercise 3.14 *findNext* using sentinels is faster than an implementation that checks for the end of the list in each iteration. But how much faster? What speed difference do you predict for many searches in a small list with 100 elements, or for a large list with 10 000 000 elements respectively? Why is the relative speed difference dependent on the size of the list?

Exercise 3.15 Design a list data type that allows sublists to be moved between lists in constant time and allows constant time access to *size* whenever sublist operations have not been used since the last access to the list size. When sublist operations have been used *size* is only recomputed when needed.

Exercise 3.16 Explain how the operations *remove*, *insertAfter*, and *concat* have to be modified to keep track of the length of a *List*.

Exercise 3.17 Implement classes *SHandle*, *SItem*, and *SList* for singly linked lists in analogy to *Handle*, *Item*, and *List*. Support all functions that can be implemented to run in constant time. Operations *head*, *first*, *last*, *isEmpty*, *popFront*, *pushFront*, *pushBack*, *insertAfter*, *concat*, and *makeEmpty* should have the same interface as before. Operations *moveAfter*, *moveToFront*, *moveToBack*, *remove*, *popFront*, and *findNext* need different interfaces.

3.3 Stacks and Queues

Sequences are often used in a rather limited way. Let us again start with examples from precomputer days. Sometimes a clerk tends to work in the following way: There is a *stack* of files on his desk that he should work on. New files are dumped on the top of the stack. When he processes the next file he also takes it from the top of the stack. The easy handling of this "data structure" justifies its use as long as no time-critical jobs are forgotten. In the terminology of the preceeding sections, a stack is a sequence that only supports the operations *pushBack*, *popBack*, and *last*. In the following we will use the simplified names *push*, *pop*, and *top* for these three operations on stacks. We get a different bahavior when people stand in line waiting for service at a post office. New customers join the line at one end and leave it at the other end.

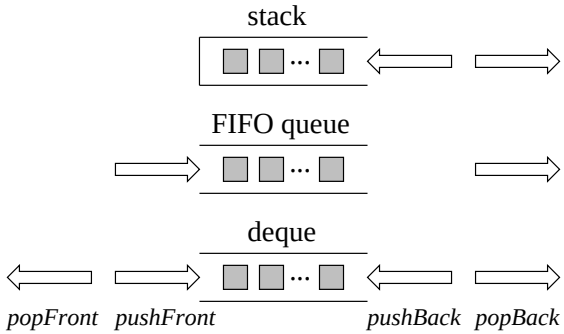


Figure 3.4: Operations on stacks, queues, and double ended queues (deques).

Such sequences are called *FIFO queues* (First In First Out) or simply *queues*. In the terminology of the *List* class, FIFO queues only use the operations *first*, *pushBack* and *popFront*.

The more general *deque*[1], or *double ended queue*, that allows operations *first*, *last*, *pushFront*, *pushBack*, *popFront* and *popBack* might also be observed at a post office, when some not so nice guy jumps the line, or when the clerk at the counter gives priority to a pregnant woman at the end of the queue. Figure 3.4 gives an overview of the access patterns of stacks, queues and deques.

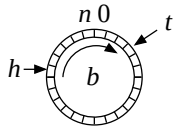
Why should we care about these specialized types of sequences if *List* can implement them all? There are at least three reasons. A program gets more readable and easier to debug if special usage patterns of data structures are made explicit. Simple interfaces also allow a wider range of implementations. In particular, the simplicity of stacks and queues allows for specialized implementions that are more space efficient than general *List*s. We will elaborate this algorithmic aspect in the remainder of this section. In particular, we will strive for implementations based on arrays rather than lists. Array implementations may also be significantly faster for large sequences because sequential access patterns to stacks and queues translate into good reuse of cache blocks for arrays. In contrast, for linked lists it can happen that each item access causes a cache miss.[klar? Verweis?]

Bounded stacks, where we know the maximal size in advance can easily be implemented with bounded arrays. For unbounded stacks we can use unbounded arrays. Stacks based on singly linked lists are also easy once we have understood that we can use *pushFront*, *popFront*, and *first* to implement *push*, *pop*, and *top* respectively.

Exercise 3.19 gives hints how to implement unbounded stacks and queues that support worst case constant access time and are very space efficient. Exercise 3.20 asks you to design stacks and queues that even work if the data will not fit in main memory. It goes without saying that all implementations of stacks and queues described here can easily be augmented to support *size* in constant time.

[1] Deque is pronounced like "deck".

Class *BoundedFIFO(n* : $\mathbb{N}$*)* **of** *Element*
  b : **Array** $[0..n]$ **of** *Element*
  h=*0* : $\mathbb{N}$ // index of first element
  t=*0* : $\mathbb{N}$ // index of first free entry



Function *isEmpty* : $\{0,1\}$*;* **return** $h = t$

Function *first* : *Element;* **assert** $\neg$*isEmpty;* **return** $b[h]$

Function *size* : $\mathbb{N}$*;* **return** $(t-h+n+1)$ **mod** $(n+1)$

Procedure *pushBack*$(x$: *Element*$)$
  assert *size*$< n$
  $b[t] := x$
  $t := (t+1)$ **mod** $(n+1)$

Procedure *popFront* **assert** $\neg$*isEmpty;* $h := (h+1)$ **mod** $(n+1)$

Figure 3.5: A bounded FIFO queue using arrays.

FIFO queues are easy to implement with singly linked lists with a pointer to the last element. Figure 3.5 gives an implementation of bounded FIFO queues using arrays. The general idea is to view the array as a cyclic structure where entry zero is the successor of the last entry. Now it suffices to maintain two indices delimiting the range of valid queue entries. These indices travel around the cycle as elements are queued and dequeued. The cyclic semantics of the indices can be implemented using arithmetics modulo the array size.[2] Our implementation always leaves one entry of the array empty because otherwise it would be difficult to distinguish a full queue from an empty queue. Bounded queues can be made unbounded using similar techniques as for unbounded arrays in Section 3.1.

Finally, deques cannot be implemented efficiently using singly linked lists. But the array based FIFO from Figure 3.5 is easy to generalize. Circular arrary can also support access using $[\cdot]$.

[2] On some machines one might get signifi cant speedups by choosing the array size as a power of two and replacing **mod** by bit operations.

Operator $[i : \mathbb{N}] : \mathit{Element}$; **return** $b[i+h \bmod n]$

Exercises

Exercise 3.18 (The towers of Hanoi) *In the great temple of Brahma in Benares, on a brass plate under the dome that marks the center of the world, there are 64 disks of pure gold that the priests carry one at a time between these diamond needles according to Brahma's immutable law: No disk may be placed on a smaller disk. In the beginning of the world, all 64 disks formed the Tower of Brahma on one needle. Now, however, the process of transfer of the tower from one needle to another is in mid course. When the last disk is finally in place, once again forming the Tower of Brahma but on a different needle, then will come the end of the world and all will turn to dust. [42].*[3]

Describe the problem formally for any number k of disks. Write a program that uses three stacks for the poles and produces a sequence of stack operations that transform the state $(\langle k,\ldots,1\rangle,\langle\rangle,\langle\rangle)$ into the state $(\langle\rangle,\langle\rangle,\langle k,\ldots,1\rangle)$.

Exercise 3.19 (Lists of arrays) Here we want to develop a simple data structure for stacks, FIFO queues, and deques that combines all the advantages of lists and unbounded arrays and is more space efficient for large queues than either of them. Use a list (doubly linked for deques) where each item stores an array of K elements for some large constant K. Implement such a data structure in your favorite programming language. Compare space consumption and execution time to linked lists and unbounded arrays for large stacks and some random sequence of pushes and pops

Exercise 3.20 (External memory stacks and queues) Design a stack data structure that needs $O(1/B)$ I/Os per operation in the I/O model from Section **??**. It suffices to keep two blocks in internal memory. What can happen in a naive implementaiton with only one block in memory? Adapt your data structure to implement FIFOs, again using two blocks of internal buffer memory. Implement deques using four buffer blocks.

Exercise 3.21 Explain how to implement a FIFO queue using two stacks so that each FIFO operations takes amortized constant time.

3.4 Lists versus Arrays

Table 3.1 summarizes the execution times of the most important operations discussed in this chapter. Predictably, arrays are better at indexed access whereas linked lists

[3] In fact, this mathematical puzzle was invented by the French mathematician Edouard Lucas in 1883.

Table 3.1: Running times of operations on sequences with n elements. Entries have an implicit $O(\cdot)$ around them.

Operation	*List*	*SList*	*UArray*	*CArray*	explanation of '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	1^*	1^*	1	1	not with inter-list *splice*
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	*insertAfter* only
remove	1	1^*	n	n	*removeAfter* only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache efficient

have their strenghts at sequence manipulation at arbitrary positions. However, both basic approaches can implement the special operations needed for stacks and queues roughly equally well. Where both approaches work, arrays are more cache efficient whereas linked lists provide worst case performance guarantees. This is particularly true for all kinds of operations that scan through the sequence; *findNext* is only one example.

Singly linked lists can compete with doubly linked lists in most but not all respects. The only advantage of cyclic arrays over unbounded arrays is that they can implement *pushFront* and *popFront* efficiently.

Space efficiency is also a nontrivial issue. Linked lists are very compact if elements are much larger than one or two pointers. For small *Element* types, arrays have the potential to be more compact because there is no overhead for pointers. This is certainly true if the size of the arrays is known in advance so that bounded arrays can be used. Unbounded arrays have a tradeoff between space efficiency and copying overhead during reallocation.

3.5 Implementation Notes

C++

Unbounded arrays are implemented as class *vector* in the standard library. Class *vector*⟨*Element*⟩ is likely to be more efficient than our simple implementation. It gives you additional control over the allocated size *w*. Usually you will give some initial estimate for the sequence size *n* when the *vector* is constructed. This can save you many grow operations. Often, you also know when the array will stop changing size and you can then force $w = n$. With these refinements, there is little reason to use the builtin C style arrays. An added benefit of *vector*s is that they are automatically destructed when the variable gets out of scope. Furthermore, during debugging you ⟹ can easily switch to implementations with bound checking.[where?]

There are some additional performance issues that you might want to address if you need very high performance for arrays that grow or shrink a lot. During reallocation, *vector* has to move array elements using the copy constructor of *Element*. In most cases, a call to the low level byte copy operation *memcpy* would be much faster. Perhaps a very clever compiler could perform this optimization automatically, but we doubt that this happens in practice. Another low level optimization is to implement *reallocate* using the standard C function *realloc*

```
b = realloc(b, sizeof(Element));
```

The memory manager might be able to avoid copying the data entirely.

A stumbling block with unbounded arrays is that pointers to array elements become invalid when the array is reallocated. You should make sure that the array does not change size while such pointers are used. If reallocations cannot be ruled out, you can use array indices rather than pointers.

The C++ standard library and LEDA offer doubly linked lists in the class *list*⟨*Element*⟩, and singly linked lists in the class *slist*⟨*Element*⟩. The implementations we have seen perform rather well. Their memory management uses free lists for all object of (roughly) the same size, rather than only for objects of the same class. Nevertheless, you might have to implement list like data structures yourself. Usually, the reason will be that your elements are part of complex data structures, and being arranged in a list is only one possible state of affairs. In such implementations, memory management is often the main challenge. Note that the operator *new* can be redefined for each class. The standard library class *allocator* offers an interface that allows you to roll your own memory management while cooperating with the memory managers of other classes.

The standard C++ library implements classes *stack*⟨*Element*⟩ and *deque*⟨*Element*⟩ for stacks and double ended queues respectively. C++ *deque*s also allow constant time indexed access using $[\cdot]$. LEDA offers classes *stack*⟨*Element*⟩ and *queue*⟨*Element*⟩ for unbounded stacks, and FIFO queues implemented via linked lists. It also offers bounded variants that are implemented as arrays.

Iterators are a central concept of the C++ standard library that implement our abstract view of sequences independent of the particular representation.[Steven: more?] ⟸

Java

The *util* package of the Java 2 platform provides *Vector* for unbounded arrays, *LinkedList* for doubly linked lists, and *Stack* for stacks. There is a quite elaborate hierarchy of abstractions for sequence like data types.[more?] ⟸

Many Java books proudly announce that Java has no pointers so that you might wonder how to implement linked lists. The solution is that object references in Java are essentially pointers. In a sense, Java has *only* pointers, because members of nonsimple type are always references, and are never stored in the parent object itself.

Explicit memory management is optional in Java, since it provides garbage collections of all objects that are not needed any more.

Java does not allow the specification of container classes like lists and arrays for a particular class *Element*. Rather, containers always contain *Object*s and the application program is responsible for performing appropriate casts. Java extensions for better support of generic programming are currently a matter of intensive debate.[Im Auge behalten] ⟸

3.6 Further Findings

Most of the algorithms described in this chapter are *folklore*, i.e., they have been around for a long time and nobody claims to be their inventor. Indeed, we have seen that many of the concepts predate computers.

[more refs?] ⟸

Amortization is as old as the analysis of algorithms. The accounting method and the even more general *potential method* were introduced in the beginning of the 80s by R.E. Brown, S. Huddlestone, K. Mehlhorn. D.D. Sleator und R.E. Tarjan [18, 43, 87, 88]. The overview article [91] popularized the term *amortized analysis*.

There is an array-like data structure, that supports indexed access in constant time and arbitrary element insertion and deletion in amortized time $O(\sqrt{n})$ [lower bound?] ⟸ The trick is relatively simple. The array is split into subarrays of size $n' = \Theta(\sqrt{n})$. Only the last subarray may contain less elements. The subarrays are maintained as cyclic arrays as described in Section 3.3. Element *i* can be found in entry i **mod** n' of subarray $\lfloor i/n' \rfloor$. A new element is inserted in its subarray in time $O(\sqrt{n})$. To repair the invariant that subarrays have the same size, the last element of this subarray is inserted

as the first element of the next subarray in constant time. This process of shifting the extra element is repeated $\mathcal{O}(n/n') = \mathcal{O}(\sqrt{n})$ times until the last subarray is reached. Deletion works similarly. Occasionally, one has to start a new last subarray or change n' and reallocate everything. The amortized cost of these additional operations can be kept small. With some additional modifications, all deque operations can be performed in constant time. Katajainen and Mortensen have designed more sophisticated implementations of deques and present an implementation study [53].

Chapter 4

Hash Tables

[Cannabis Blatt als Titelbild?] ⟸

If you want to get a book from the central library of the University of Karlsruhe, you have to order the book an hour in advance. The library personnel take the book from the magazine. You can pick it up in a room with many shelves. You find your book in a shelf numbered with the last *digits of your library card. Why the last digits and not the leading digits? Probably because this distributes the books more evenly about the shelves. For example, many first year students are likely to have the same leading digits in their card number. If they all try the system at the same time, many books would have to be crammed into a single shelf.*

The subject of this chapter is the robust and efficient implementation of the above "delivery shelf data structure" known in computer science as a *hash table*. The definition of "to hash" that best expresses what is meant is "to bring into complete disorder". Elements of a set are intentionally stored in disorder to make them easier to find. Although this sounds paradoxical, we will see that it can make perfect sense. Hash table accesses are among the most time critical parts in many computer programs. For example, scripting languages like `awk` [2] or `perl` [95] use hash tables as their *only* data structures. They use them in the form of *associative arrays* that allow arrays to be used with any kind of index type, e.g., strings. Compilers use hash tables for their *symbol table* that associates identifiers with information about them. Combinatorial search programs often use hash tables to avoid looking at the same situation multiple times. For example, chess programs use them to avoid evaluating a position twice that can be reached by different sequences of moves. One of the most widely used implementations of the *join*[ref some database book] operation in relational databases ⟸ temporarily stores one of the participating relations in a hash table. (Exercise 4.4 gives a special case example.) Hash tables can often be used to replace applications

of sorting (Chapter 5) or of search trees (Chapter 7). When this is possible, one often gets a significant speedup.

Put more formally, hash tables store a set M of elements where each element e has a unique key $key(e)$. To simplify notation, we extend operation on keys to operations on elements. For example, the comparison $e = k$ is a abbreviation for $key(e) = k$. The following *dictionary* operations are supported: [einheitliche Def. der Ops. Bereits in intro? move from summary?]

$\Longrightarrow$

$M.insert(e : Element)$**:** $M := M \cup \{e\}$

$M.remove(k : Key)$**:** $M := M \setminus \{e\}$ where e is the unique element with $e = k$, i.e., we assume that *key* is a one-to-one function.

$M.find(k : Key)$ If there is an $e \in M$ with $e = k$ return e otherwise return a special element $\perp$.

In addition, we assume a mechanism that allows us to retrieve all elements in M. Since this *forall* operation is usually easy to implement and is not severely affected by the details of the implementation, we only discuss it in the exercises.

In the library example, Keys are the library card numbers and elements are the book orders. Another pre-computer example is an English-German dictionary. The keys are English words and an element is an English word together with its German translations. There are many ways to implement dictionaries (for example using the *search trees* discussed in Chapter 7). Here, we will concentrate on particularly efficient implementations using hash tables.

The basic idea behind a hash table is to map keys to m array positions using a *hash function* $h : Key \to 0..m-1$. In the library example, h is the function extracting the least significant digits from a card number. Ideally, we would like to store element e in a table entry $t[h(e)]$. If this works, we get constant execution time for all three operations *insert*, *remove*, and *find*.[1]

Unfortunately, storing e in $t[h(e)]$ will not always work as several elements might *collide*, i.e., they might map to the same table entry. A simple fix of this problem in the library example allows several book orders to go to the same shelf. Then the entire shelf has to be searched to find a particular order. In a computer we can store a sequence of elements in each table entry. Because the sequences in each table entry are usually implemented using singly linked lists, this hashing scheme is known as *hashing with chaining*. Section 4.1 analyzes hashing with chaining using rather

[1] Strictly speaking, we have to add additional terms to the execution time for moving elements and for evaluating the hash function. To simplify notation, we assume in this chapter that all this takes constant time. The execution time in the more detailed model can be recovered by adding the time for one hash function evaluation and for a constant number of element moves to *insert* and *remove*.

optimistic assumptions about the properties of the hash function. In this model, we achieve constant expected time for dictionary operations.

In Section 4.2 we drop these assumptions and explain how to construct hash functions that come with (probabilistic) performance guarantees. Note that our simple examples already show that finding good hash functions is nontrivial. For example, if we applied the least significant digit idea from the library example to an English-German dictionary, we might come up with a hash function based on the last four letters of a word. But then we would have lots of collisions for words ending on 'tion', 'able', etc.

We can simplify hash tables (but not their analysis) by returning to the original idea of storing all elementsn in the table itself. When a newly inserted element e finds entry $t[h(x)]$ occupied, it scans the table until a free entry is found. In the library example, this would happen if the shelves were too small to hold more than one book. The librarians would then use the adjacent shelves to store books that map to the same delivery shelf. Section 4.3 elaborates on this idea, which is known as *hashing with open addressing and linear probing*.

Exercise 4.1 (Scanning a hash table.) Implement the *forall* operation for your favorite hash table data type. The total execution time for accessing all elements should be linear in the size of the hash table. Your implementation should hide the representation of the hash table from the application. Here is one possible interface: Implement an *iterator* class that offers a view on the hash table as a sequence. It suffices to support initialization of the iterator, access to the current element and advancing the iterator to the next element.

Exercise 4.2 Assume you are given a set M of pairs of integers. M defines a binary relation R_M. Give an algorithm that checks in expected time $O(|M|)$ whether R_M is symmetric. (A relation is symmetric if $\forall(a,b) \in M : (b,a) \in M$.) Space consumption should be $O(|M|)$.

Exercise 4.3 Write a program that reads a text file and outputs the 100 most frequent words in the text. Expected execution time should be linear in the size of the file.

Exercise 4.4 (A billing system:) Companies like Akamai[2] deliver millions and millions of files every day from their thousands of servers. These files are delivered for other E-commerce companies who pay for each delivery (fractions of a cent). The servers produce log files with one line for each access. Each line contains information that can be used to deduce the price of the transaction and the customer ID. Explain

[2] `http://www.akamai.com/index_flash.html`

Figure 4.1: Clever description ....

how to write a program that reads the log files once a day and outputs the total charges
⟹ for each customer (a few hundred).[check with Harald prokop.]

4.1 Hashing with Chaining

Hashing with chaining maintains an array t with m entries, each of which stores a sequence of elements. Assuming a hash function $h : Key \to 0..m-1$ we can imple-
⟹ ment the three dictionary functions as follows: [harmonize notation.][minibildchen.
⟹ vorher, nachher]

insert(e)**:** Insert e somewhere in sequence $t[h(e)]$.

remove(k)**:** Scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, remove it and return.

find(k) : Scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, return it. Otherwise, return $\perp$.

⟹ Figure 4.1 gives an example.[sequences. (or list with sentinel?)]

Using hashing with chaining, insertion can be done in worst case constant time if the sequences are implemented using linked lists. Space consumption is at most $O(n+m)$ (see Exercise 4.6 for a more accurate analysis).

The other two operations have to scan the sequence $t[h(k)]$. In the worst case, for example, if *find* looks for an element that is not there, the entire list has to be scanned. If we are unlucky, all elements could be mapped to the same table entry. For n elements, we could would get execution time $O(n)$. Can we find hash functions that never fail? Unfortunately, the answer is no. Once we fix h, it is always possible to find a set of $n < |Key|/m$ keys that all map to the same table entry.

To get a less pessimistic — and hopefully realistic — estimate, we now look at the *average case* efficiency in the following sense: We fix a set of n elements and analyze the execution time averaged over all possible hash functions $h : Key \to 0..m-1$. Equivalently, we can ask what the expected time of a *remove* or *find* is if we pick a hash function uniformly at random from the set of all possible hash functions.

Theorem 4.1 *If n elements are stored in a hash table with m entries using hashing with chaining, the expected execution time of remove or find is $O(1+n/m)$ if we assume a random hash function.*

The proof is very simple once we know the probabilistic concepts of random variables, their expectation, and the linearity of expectation described in Appendix **??**.

Proof: Consider the execution time of *remove* or *find* for a key k. Both need constant time plus the time for scanning the sequence $t[h(k)]$. Hence, even if this sequence has to be scanned completely, the expected execution time is $O(1+\mathrm{E}[X])$ where the random variable X stands for the length of sequence $t[h(k)]$. Let e_i denote element i in the table. Let $X_1,\ldots,X_n$ denote *indicator* random variables where $X_i = 1$ if $h(e_i) = h(k)$ and otherwise $X_i = 0$. We have $X = \sum_{i=1}^{n} X_i$. Using the linearity of expectation, we get

$$\mathrm{E}[X] = \mathrm{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \mathrm{prob}(X_i = 1) \ .$$

A random hash function will map e_i to all m table entries with the same probability, independent of $h(k)$. Hence, $\mathrm{prob}(X_i = 1) = 1/m$. We get $\mathrm{E}[X] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$ and overall an expected execution time of $O(n/m+1)$. ∎

We can achieve linear space requirements and constant expected execution time of all three operations if we keep $m = \Theta(n)$ at all times. This can be achieved using adaptive reallocation analogous to the unbounded arrays described in Section 3.1.

Exercise 4.5 (Unbounded Hash Tables) Explain how to implement hashing with chaining in such a way that $m = \Theta(n)$ at all times. Assume that there is a hash function $h' : Key \to \mathbb{N}$ and that you set $h(k) = h'(k)$ **mod** m.

Exercise 4.6 (Waste of space) Waste of space in hashing with chaining is due to empty table entries. Assuming a random hash function, compute the expected number of empty table entries as a function of m and n. Hint: Define indicator random variables $Y_0, \ldots, Y_{m-1}$ where $Y_i = 1$ if $t[i]$ is empty.

4.2 Universal Hash Functions

The performance guarantees for hashing in Theorem 4.1 have a serious flaw: It is very expensive to get hold of a truly random hash function $h : Key \to 0..m-1$. We would have to compute a table of random values with $|Key|$ entries. This is prohibitive; think of $Key = 0..2^{32}-1$ or even strings. On the other hand, we have seen that we cannot use a single fixed hash function. Some randomness is necessary if we want to cope with all possible inputs. The golden middle way is to choose a hash function randomly from some smaller set of functions. For the purposes of this chapter we only need the following simple condition:

[right aligned kleines Bildchen mit H nested in $\{0..m-1\}^{Key}$] ⟸

Definition 4.2 A family $\mathcal{H} \subseteq \{0..m-1\}^{Key}$ of functions from keys to table entries is *c-universal* if for all *x, y* in *Key* with $x \neq y$ and random $h \in \mathcal{H}$,

$$\text{prob}(h(x) = h(y)) \leq \frac{c}{m} \ .$$

For *c*-universal families of hash functions, we can easily generalize the proof of Theorem 4.1 for fully random hash functions.

Theorem 4.3 *If n elements are stored in a hash table with m entries using hashing with chaining, the expected execution time of remove or find is* $O(1+cn/m)$ *if we assume a hash function taken randomly from a c-universal family.*

Now it remains to find *c*-universal families of hash functions that are easy to compute. We explain a simple and quite practical 1-universal family in detail and give further examples in the exercises. In particular, Exercise 4.8 gives an family that is perhaps even easier to understand since it uses only simple bit operations. Exercise 4.10 introduces a simple family that is fast for small keys and gives an example where we have *c*-universality only for $c > 1$.

Assume that the table size *m* is a prime number. Set $w = \lfloor \log m \rfloor$ and write keys as bit strings. Subdivide each bit string into pieces of at most *w* bits each, i.e., view keys as *k*-tuples of integers between 0 and $2^w - 1$. For $\mathbf{a} = (a_1, \ldots, a_k)$ define

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \ \mathbf{mod}\ m$$

where $\mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{k} a_i x_i$ denotes the scalar product.

Theorem 4.4

$$H^{\cdot} = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k \right\}$$

is a 1-universal family of hash functions if m is prime.

In other words, we get a good hash function if we compute the scalar product between a tuple representation of a key and a random vector.[bild dotprod.eps. redraw in latex?]

Proof: Consider two distinct keys $\mathbf{x} = (x_1, \ldots, x_k)$ and $\mathbf{y} = (y_1, \ldots, y_k)$. To determine $\text{prob}(h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y}))$, we count the number of choices for **a** such that $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.

Fix an index *j* such that $x_j \neq y_j$. We claim that for each choice of the a_i's with $i \neq j$ there is exacly one choice of a_j such that $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$. To show this, we solve the equation for a_j:

$$\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \quad &\Leftrightarrow \quad \sum_{1\leq i\leq k} a_i x_i \equiv \sum_{1\leq i\leq k} a_i y_i && (\ \mathbf{mod}\ m) \\
&\Leftrightarrow \quad a_j(y_j - x_j) \equiv \sum_{i\neq j, 1\leq i\leq k} a_i(x_i - y_i) && (\ \mathbf{mod}\ m) \\
&\Leftrightarrow \quad a_j \equiv (y_j - x_j)^{-1} \sum_{i\neq j, 1\leq i\leq k} a_i(x_i - y_i) && (\ \mathbf{mod}\ m)
\end{aligned}$$

where $(x_j - y_j)^{-1}$ denotes the *multiplicative inverse* of $(x_j - y_j)$ (i.e., $(x_j - y_j) \cdot (x_j - y_j)^{-1} \equiv 1(\ \mathbf{mod}\ m)$). This value is unique **mod** *m* and is guaranteed to exist because we have chosen *j* such that $x_j - y_j \not\equiv 0(\ \mathbf{mod}\ m)$ and because the integers modulo a prime number form a field.

There are m^{k-1} ways to choose the a_i with $i \neq j$. Since the total number of possible choices for **a** is m^k, we get

$$\text{prob}(h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y})) = \frac{m^{k-1}}{m^k} = \frac{1}{m}.$$

∎

Is it a serious restriction that we need prime table sizes? On the first glance, yes. We cannot burden users with the requirement to specify prime numbers for *m*. When we adaptively grow or shrink an array, it is also not obvious how to get prime numbers for the new value of *m*. But there is a simple way out. Number theory tells us that there is a prime number just around the corner [41]. On the average it suffices to add an extra $O(\log m)$ entries to the table to get a prime *m*. Furthermore, we can afford to look for *m* using a simple brute force approach: Check the desired *m* for primality by trying to divide by all integers in $2..\lfloor\sqrt{m}\rfloor$. If a divisor is found, increment *m*. Repeat until a prime number is found. This takes average time $O(\sqrt{m}\log m)$ — much less than the time needed to initialize the table with $\perp$ or to move elements at a reallocation.

Exercise 4.7 (Strings as keys.) Implement the universal family $H^{\cdot}$ for strings of 8 bit characters. You can assume that the table size is at least $m = 257$. The time for evaluating a hash function should be proportional to the length of the string being processed. Input strings may have arbitrary lengths not known in advance. Hint: compute the random vector **a** lazily, extending it only when needed.

Exercise 4.8 (Hashing using bit matrix multiplication.) [Literatur? Martin fragen] For table size $m = 2^w$ and $Key = \{0,1\}^k$ consider the family of hash functions

$$H^{\oplus} = \left\{ h_M : M \in \{0,1\}^{w\times k} \right\}$$

where $h_M(\mathbf{x}) = Mx$ computes a matrix product using arithmetics **mod** 2. The resulting bit-vector is interpreted as an integer from $0..2^w - 1$. Note that multiplication **mod** 2 is the logical and-operation, and that addition **mod** 2 is the logical exclusive-or operation $\oplus$.

a) Explain how $h_M(\mathbf{x})$ can be evaluated using k bit-parallel exclusive-or operations.

b) Explain how $h_M(\mathbf{x})$ can be evaluated using w bit-parallel *and* operations and w *parity* operations. Many machines support a machine instruction *parity*($\mathbf{y}$) that is one if the number of one bits in y is odd and zero otherwise.

c) We now want to show that $H^{\oplus}$ is 1-universal. As a first step show that for any two keys $x \neq y$, any bit position j where x and y differ, and any choice of the columns M_i of the matrix with $i \neq j$, there is exactly one choice of column M_j such that $h_M(\mathbf{x}) = h_M(\mathbf{y})$.

d) Count the number of ways to choose $k-1$ columns of M.

e) Count the total number of ways to choose M.

f) Compute the probability $\text{prob}(h_M(\mathbf{x}) = h_M(\mathbf{y}))$ for $x \neq y$ if M is chosen randomly.

***Exercise 4.9 (More matrix multiplication.)** Define a class of hash functions

$$H^{\times} = \left\{ h_M : M \in \{0..p\}^{w \times k} \right\}$$

that generalizes class $H^{\oplus}$ by using arithmetics **mod** p for some prime number p. Show that $H^{\times}$ is 1-universal. Explain how $H^{\cdot}$ is also a special case of $H^{\times}$.

Exercise 4.10 (Simple linear hash functions.) Assume $Key = 0..p-1$ for some prime number p. Show that the following family of hash functions is $(\lceil |Key|/m \rceil / (|Key|/m))^2$-universal.

$$H^* = \left\{ h_{(a,b)} : a, b \in 0..p-1 \right\}$$

where $h_{(a,b)}(x) = ax + b$ **mod** p **mod** m.

Exercise 4.11 (A counterexample.) Consider the set of hash functions

$$H^{\text{fool}} = \left\{ h_{(a,b)} : a, b \in 0..p-1 \right\}$$

with $h_{(a,b)}(x) = ax + b$ **mod** m. Show that there is a set of $\lfloor |Key|/m \rfloor$ keys M such that

$$\forall x, y \in M : \forall h_{(a,b)} \in H^{\text{fool}} : h_{(a,b)}(x) = h_{(a,b)}(y)$$

even if m is prime.

Exercise 4.12 (Table size 2^ℓ.) Show that the family of hash functions

$$H^{\gg} = \left\{ h_a : 0 < a < 2^k \wedge\ a \text{ is odd} \right\}$$

with $h_a(x) = (ax \textbf{ mod } 2^k) \div 2^{k-\ell}$ is 2-universal. (Due to Keller and Abolhassan.)

Exercise 4.13 (Table lookup made practical.) Let $m = 2^w$ and view keys as $k+1$-tuples where the 0-th elements is a w-bit number and the remaining elements are a-bit numbers for some small constant a. The idea is to replace the single lookup in a huge table from Section 4.1 by k lookups in smaller tables of size 2^a. Show that

$$H^{\oplus[]} = \left\{ h_{(t_1,\ldots,t_k)\oplus} : t_i \in \{0..m-1\}^{\{0..w-1\}} \right\}$$

where

$$h_{\oplus(t_1,\ldots,t_k)}((x_0, x_1, \ldots, x_k)) = x_0 \oplus \bigoplus_{i=1}^{k} t_i[x_i] \quad .$$

is 1-universal.

4.3 Hashing with Linear Probing

Hashing with chaining is categorized as a *closed* hashing approach because each entry of the table $t[i]$ has to cope with all the elements with $h(e) = i$. In contrast, *open* hashing schemes open up other table entries to take the overflow from overloaded fellow entries. This added flexibility allows us to do away with secondary data structures like linked lists—all elements are stored directly in table entries. Many ways of organizing open hashing have been investigated. We will only explore the simplest scheme, which is attributed to G. Amdahl [57] who used the scheme in the early 50s. Unused entries are filled with a special element $\perp$. An element e is stored in entry $t[h(e)]$ or further to the right. But we only go away from index $h(e)$ with good reason, i.e., only if the table entries between $t[h(e)]$ and the entry where e is actually stored are occupied by other elements. The remaining implementation basically follows from this invariant. Figure 4.2 gives pseudocode. [todo: inline bildchen. linear2.fig aus vorlesung?] ⟸

To insert an element, we linearly scan the table starting at $t[h(e)]$ until a free entry is found, where d is then stored. Similarly, to find an element with key k, we scan the table starting at $t[h(k)]$ until a matching element is found. The search can be aborted when an empty table entry is encountered, because any matching element further to the right would violate the invariant. So far this sounds easy enough, but we have to deal with two complications.

```
// Hash table with m primary entries and an overflow area of size m'.
Class BoundedLinearProbing(m, m' : ℕ; h : Key → 0..m−1)
    t=⟨⊥,...,⊥⟩ : Array [0..m+m'−1] of Element
    invariant ∀i : t[i] ≠ ⊥ ⇒ ∀j ∈ {h(t[i])..i−1} : t[i] ≠ ⊥

    Procedure insert(e : Element)
        for i := h(e) to ∞ while t[i] ≠ ⊥ do ;
        assert i < m+m'−1                                   // no overflow
        t[i] := e

    Function find(k : Key) : Element
        for i := h(e) to ∞ while t[i] ≠ ⊥ do
            if t[i] = k then return t[i]
        return ⊥                                            // not found

    Procedure remove(k : Key)
        for i := h(k) to ∞ while k ≠ t[i] do
            if t[i] = ⊥ then return                         // nothing to do
        // Scan a cluster of elements.
        // i is where we currently plan for a ⊥ (a hole).
        for j := i+1 to ∞ while t[j] ≠ ⊥ do
            // Establish invariant for t[j].
            if h(t[j]) ≤ i then
                t[i] := t[j]                                // Overwrite removed element
                i := j                                      // move planned hole
        t[i] := ⊥                                           // erase freed entry
```

Figure 4.2: Hashing with linear probing.

What happens if we reach the end of the table during insertion? We choose a very simple fix by allocating m' table entries to the right of the largest index produced by the hash function h. For 'benign' hash functions it should be sufficient to choose m' much smaller than m in order to avoid table overflows. Exercise 4.14 asks you to develop a more robust albeit slightly slower variant where the table is treated as a cyclic array.

A more serious question is how *remove* should be implemented. After finding the element, we might be tempted to replace it with $\bot$. But this could violate the invariant for elements further to the right. Consider the example

$$t = [\ldots, \underset{h(z)}{x}, y, z, \ldots] \ .$$

When we naively *remove* element y and subsequently try to find z we would stumble over the hole left by removing y and think that z is not present. Therefore, most other variants of linear probing disallow *remove* or mark removed elements so that a subsequent *find* will not stop there. The problem with marking is that the number of nonempty cells (occupied or marked) keeps increasing, so searches eventually become very slow. This can only be mitigated by introducing the additional complication of periodic reorganizations of the table.

Here we give a different solution that is faster and needs no reorganizations [57, Algorithm R][check]. The idea is rather obvious—when an invariant may be violated, reestablish it. Let i denote the index of the deleted element. We scan entries $t[j]$ to the right of i to check for violations of the invariant. If $h(t[j]) > i$ the invariant still holds even if $t[i]$ is emptied. If $h(t[k]) \le i$ we can move $t[j]$ to $t[i]$ without violating the invariant for the moved element. Now we can pretend that we want to remove the duplicate copy at $t[j]$ instead of $t[i]$, i.e., we set $i := j$ and continue scanning. Figure **??** depicts this case[todo???]. We can stop scanning, when $t[j] = \bot$ because elements to the right that violate the invariant would have violated it even before the deletion.

Exercise 4.14 (Cyclic linear probing.) Implement a variant of linear probing where the table size is m rather than $m+m'$. To avoid overflow at the right end of the array, make probing wrap around.

- Adapt *insert* and *delete* by replacing increments with $i := i+1$ **mod** m.
- Specify a predicate *between*(i, j, k) that is true if and only if j is cyclically between i and j.
- Reformulate the invariant using *between*.
- Adapt *remove*.

Exercise 4.15 (Unbounded linear probing.) Implement unbounded hash tables using linear probing and universal hash functions. Pick a new random hash function whenever the table is reallocated. Let $1 < \gamma < \beta < \alpha$ denote constants we are free to choose. Keep track of the number of stored elements n. Grow the table to $m = \beta n$ if $n > m/\gamma$. Shrink the table to $m = \beta n$ if $n < m/\alpha$. If you do not use cyclic probing as in Exercise 4.14, set $m' = \delta m$ for some $\delta < 1$ and reallocate the table if the right end should overflow.

4.4 Chaining Versus Linear Probing

We have seen two different approaches to hash tables, chaining and linear probing. $\Longrightarrow$ Which one is better? We do not delve into the details of[pursue a detailed] a theoretical analysis, since this is complicated for the case of linear probing, and the results would remain inconclusive. Therefore, we only discuss some qualitative issues without detailed proof.

An advantage of linear probing is that, in each table access, a contiguous piece of memory is accessed. The memory subsystems of modern processors are optimized for this kind of access pattern, whereas they are quite slow at chasing pointers when the data does not fit in cache memory. A disadvantage of linear probing is that search times become very high when the number of elements approaches the table size. For chaining, the expected access time remains very small. On the other hand, chaining wastes space for pointers that could be used to support a larger table in linear probing. Hence, the outcome is not so clear.

To decide which algorithm is faster, we implemented both algorithms. The outcome is that both perform about equally well when they are given the same amount of memory. The differences are so small that details of the implementation, compiler, operating system and machine used can reverse the picture. Hence we do not report exact figures.

However, to match linear probing, chaining had to be implemented very carefully using the optimizations from Section 4.5. Chaining can be much slower than linear probing if the memory management is not done well or if the hash function maps many elements to the same table entry. [more advantages of chaining: referential $\Longrightarrow$ integrity, theoretical guarantees.]

4.5 Implementation Notes

Although hashing is an algorithmically simple concept, a clean, efficient, and robust implementation can be surprisingly nontrivial. Less surprisingly, the most important issue are hash functions. Universal hash functions could in certain cases work on the bit representation of many data types and allow a reusable dictionary data type that hides the existence of hash functions from the user. However, one usually wants at least an option of a user specified hash function. Also note that many families of hash functions have to be adapted to the size of the table so that need some internal state that is changed when the table is resized or when access times get too large.

Most applications seem to use simple very fast hash functions based on `xor`, shifting, and table lookups rather than universal hash functions[3]. Although these functions seem to work well in practice, we are not so sure whether universal hashing is really slower. In particular, family $H^{\oplus[]}$ from Exercise 4.13 should be quite good for integer keys and Exercise 4.7 formulates a good function for strings. It might be possible to implement the latter function particularly fast using the *SIMD-instructions* in modern processors that allow the parallel execution of several small precision operations.

Implementing Hashing with Chaining

Hashing with chaining uses only very specialized operations on sequences, for which singly linked lists are ideally suited. Since these lists are extremely short, some deviations from the implementation scheme from Section 3.2 are in order. In particular, it would be wasteful to store a dummy item with each list. However, we can use a single shared dummy item to mark the end of the list. This item can then be used as a sentinel element for *find* and *remove* as in function *findNext* in Section 3.2.1. This trick not only saves space, but also makes it likely that the dummy item is always in cache memory.

There are two basic ways to implement hashing with chaining. Entries of *slim* tables are pointers to singly linked lists of elements. *Fat* tables store the first list element in the table itself. Fat tables are usually faster and more space efficient. Slim tables may have advantages if the elements are very large. They also have the advantage that references to table entries remain valid when tables are reallocated. We have already observed this complication for unbounded arrays in Section 3.5.

Comparing the space consumption of hashing with chaining and linear probing is even more subtle than outlined in Section 4.4. On the one hand, the linked lists burden the memory management with many small pieces of allocated memory, in particular if memory management for list items is not implemented carefully as discussed in Section 3.2.1. On the other hand, implementations of unbounded hash tables based on chaining can avoid occupying two tables during reallocation by using the following method: First, concatenate all lists to a single list ℓ. Deallocate the old table. Only then allocate the new table. Finally, scan ℓ moving the elements to the new table.

[3] For example `http://burtleburtle.net/bob/hash/evahash.html`

C++

The current C++ standard library does not define a hash table data type but the popular implementation by SGI (`http://www.sgi.com/tech/stl/`) offers several variants: *hash$_s$et*, *hash$_m$ap*, *hash$_m$ultiset*, *hash$_m$ultimap*. Here "set" stands for the kind of interfaces used in this chapter wheras a "map" is an associative array indexed Keys. The term "multi" stands for data types that allow multiple elements with the same key. Hash functions are implemented as *function objects*, i.e., the class `hash<T>` overloads the operator "`()`" so that an object can be used like a function. The reason for this charade is that it allows the hash function to store internal state like random coefficients.

LEDA offers several hashing based implementations of dictionaries. The class $h_a rray\langle Key, Element\rangle$ implements an associative array assuming that a hash function $intHash(Key\&)$ is defined by the user and returns an integer value that is then mapped to a table index by LEDA. The implementation grows adaptively using hashing with chaining.[check. Kurt, was ist der Unterschied zu *map*?]

Java

The class *java.util.hashtable* implements unbounded hash tables using the function *hashCode* defined in class *Object* as a hash function.

***Exercise 4.16 (Associative arrays.)** Implement a C++-class for associative arrays. Support `operator[]` for any index type that supports a hash function. Make sure that `H[x]=...` works as expected if x is the key of a new element.

4.6 Further Findings

More on Hash Functions

Carter and Wegman [20] introduced the concept of universal hashing to prove constant *expected* access time to hash tables. However, sometimes more than that is needed. For example, assume m elements are mapped to m table entries and consider the expected *maximum occupancy* $\max_{M\subseteq Key,|M|=m} \mathrm{E}[\max_i |\{x \in M : h(x) = i\}|]$. A truly random hash function produces a maximum occupancy of $O(\log m/\log\log m)$ whereas there are universal families and sets of keys where the maximum occupancy of a table entry is $\Theta(\sqrt{m})$. [reference? Martin und Rasmus fragen], i.e., there can be be some elements where access takes much longer than constant time. This is undesirable for real time applications and for parallel algorithms where many processors work together and have to wait for each other to make progress. Dietzfelbinger

and Meyer auf der Heide [31][check ref] give a family of hash functions that [which bound, outline trick.]. [m vs n dependence?]

Another key property of hash functions is in what sense sets of elements are hashed independently. A family $\mathcal{H} \subseteq \{0..m-1\}^{Key}$ is *k-way* independent if $\mathrm{prob}(h(x_1) = a_1 \wedge \cdots \wedge h(x_k) = a_k) = m^{-k}$ for any set of k keys and k hash function values. One application of higher independence is the reduction of the maximum occupancy because k-wise independence implies maximum occupancy $O\left(m^{1/k}\right)$[check bound][**?**]. Below we will see an application requiring $O(\log m)$-wise independence. A simple k-wise independent family of hash functions are polynomials of degree $k-1$ with random coefficients [fill in details][**?**].

[strongly universal hashing]

[cryptographic hash functions]

Many hash functions used in practice are not universal and one can usually construct input where they behave very badly. But empirical tests indicate that some of them are quite robust in practice.[hash function studies http://www.cs.amherst.edu/~ccm/challenge5/] It is an interesting question whether universal families can completely replace hash functions without theoretical performance guarantees. There seem to be two main counterarguments. One is that not all programmers have the mathematical to background to implement universal families themselves. However, software libraries can hide the complications from the everyday user. Another argument is speed but it seems that for most applications there are very fast universal families. For example, if $|Key|$ is not too large, the families from Exercise 4.12 and 4.13 seem to be hard to beat in terms of speed.

More Hash Table Algorithms

Many variants of hash tables have been proposed, e.g. [57, 39]. Knuth [57] gives an average case analysis of many schemes. One should however be careful with interpreting these results. For example, there are many algorithms based on open addressing that need less probes into the table than linear probing. However, the execution time of such schemes is usually higher since they need more time per probe in particular because they cause more cache faults.

extensible hashing.

Worst Case Constant Access Time

Kurt? Rasmus?

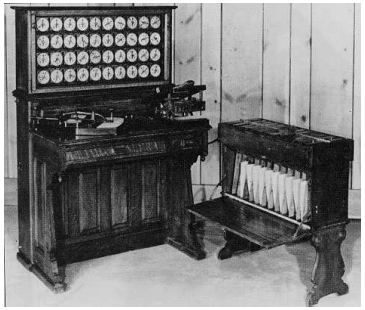
Rasmus fragen

cuckoo hashing

perfect hashing, shifting trick

Chapter 5

Sorting and Selection



A telephone directory book is alphabetically sorted by last name because this makes it very easy to find an entry even in a huge city. A naive view on this chapter could be that it tells us how to make *telophone books. An early example of even more massive data processing were the statistical evaluation of census data. 1500 people needed seven years to manually process the US census in 1880. The engineer Herman Hollerith*[1] *who participated in this evaluation as a statistician, spend much of the ten years to the next census developing counting and sorting machines (the small machine in the left picture) for mechanizing this gigantic endeavor. Although the 1890 census had to evaluate more people and more questions, the basic evaluation was finished in 1891. Hollerith's company continued to play an important role in the development of the information processing industry; since 1924 is is known as International Business Machines (IBM). Sorting is important for census statistics because one often wants*

[1] The picuture to the right. Born February 29 1860, Buffalo NY; died November 17, 1929, Washington DC.

to group people by an attribute, e.g., age and then do further processing for persons which share an attribute. For example, a question might be how many people between 20 and 30 are living on farms. This question is relatively easy to answer if the database entries (punch cards in 1890) are sorted by age. You take out the section for ages 20 to 30 and count the number of people fulfilling the condition to live on a farm.

Although we probably all have an intuitive feeling what *sorting* means, let us look at a formal definition. To sort a sequence $s = \langle e_1, \ldots, e_n \rangle$, we have to produce a sequence $s' = \langle e'_1, \ldots, e'_n \rangle$ such that s' is a permutation of s and such that $e'_1 \leq e'_2 \leq \cdots \leq e'_n$. As in Chapter 4 we distinguish between an *element* e and its *key* $key(e)$ but extend the comparison operations between keys to elements so that $e \leq e'$ if and only if $key(e) \leq key(e')$. Any key comparison relation can be used as long as it defines a *strict weak order*, i.e., a reflexive, transitive, and antisymmetric with respect to some equivalence relation $\equiv$. This all sounds a bit cryptic, but all you really need to remember is that all elements must be comparable, i.e., a *partial order* will not do, and for some elements we may not care how they are ordered. For example, two elements may have the same key or may have decided that upper and lower case letters should not be distinguished.

Although different comparison relations for the same data type may make sense, the most frequent relations are the obvious order for numbers and the *lexicographic order* (see Appendix A) for tuples, strings, or sequences.

Exercise 5.1 (Sorting words with accents.) Ordering strings is not always as obvious as it looks. For example, most European languages augment the Latin alphabet with a number of accented characters. For comparisons, accents are essentially omitted. Only ties are broken so that Mull$\leq$Müll etc.

a) Implement comparison operations for strings that follow these definitions and can be used with your favorite library routine for sorting.[2]

b) In German telephone books (but not in Encyclopedias) a second way to sort words is used. The umlauts ä, ö, and ü are identified with their circumscription ä=ae, ö=oe, and ü=ue. Implement this comparison operation.

c) The comparison routines described above may be too slow for time critical applications. Outline how strings with accents can be sorted according to the above rules using only plain lexicographic order within the sorting routines. In particular, explain how tie breaking rules should be implemented.

Exercise 5.2 Define a total order for complex numbers where $x \leq y$ implies $|x| \leq |y|$.

[2] For West European languages like French, German, or Spanish you can assume the character set ISO LATIN-1.

Sorting is even more important than it might seem since it is not only used to produce sorted output for human consumption but, more importantly, as an ubiquitous algorithmic tool:

Preprocessing for fast search: Not only humans can search a sorted directory faster than an unsorted one. Although hashing might be a faster way to find elements, sorting allows us additional types of operations like finding all elements which lie in a certain range. We will discuss searching in more detail in Chapter 7.

Grouping: Often we want to bring equal elements together to count them, eliminate duplicates, or otherwise process them. Again, hashing might be a faster alternative. But sorting has advantages since we will see rather fast deterministic algorithms for it that use very little space.

Spatial subdivision: Many divide-and-conquer algorithms first sort the inputs according to some criterion and then split the sorted input list.[ref to example? graham scan (but where? vielleicht als appetizer in further findings? or in einem special topic chapter?)???] ⟸

Establishing additional invariants: Certain algorithms become very simple if the inputs are processed in sorted order. Exercise 5.3 gives an example. Other examples are Kruskal's algorithm in Section 11.3, several of the algorithms for the knapsack problem in Chapter 12, or the scheduling algorithm proposed in Exercise 12.7. You may also want to remember sorting when you solve Exercise **??** on interval graphs.

Sorting has a very simple problem statement and in Section 5.1 we see correspondingly simple sorting algorithms. However, it is less easy to make these simple approaches efficient. With mergesort, Section 5.2 introduces a simple divide-and-conquer sorting algorithm that runs in time $O(n \log n)$. Section 5.3 establishes that this bound is optimal for all *comparison based* algorithms, i.e., algorithms that treat elements as black boxes that can only be compared and moved around. The quicksort algorithm described in Section 5.4 it is also based on the divide-and-conquer principle and perhaps the most frequently used sorting algorithm. Quicksort is also a good example for a randomized algorithm. The idea behind quicksort leads to a simple algorithm for a problem related to sorting. Section 5.5 explains how the k-th smallest from n elements can be found in time $O(n)$. Sorting can be made even faster than the lower bound from Section 5.3 by looking into the bit pattern of the keys as explained in Section 5.6. Finally, Section 5.7 generalizes quicksort and mergesort to very good algorithms for sorting huge inputs that do not fit into internal memory.

Exercise 5.3 (A simple scheduling problem.) Assume you are a hotel manager who has to consider n advance bookings of rooms for the next season. Your hotel has k identical rooms. Bookings contain arrival date and departure date. You want to find

out whether there are enough rooms in your hotel to satisfy the demand. Design an algorithm that solves this problem in time $O(n \log n)$. Hint: Set up a sequence of events containing all arrival and departure dates. Process this list in sorted order.

Exercise 5.4 (Sorting with few different keys.) Design an algorithm that sorts n elements in $O(k \log k + n)$ expected time if there are only k different keys appearing in the input. Hint: use universal hashing.

***Exercise 5.5 (Checking.)** It is easy to check whether a sorting routine produces sorted output. It is less easy to check whether the output is also a permutation of the input. But here is a fast and simple Monte Carlo algorithm for integers: Show that $\langle e_1, \ldots, e_n \rangle$ is a permutation of $\langle e'_1, \ldots, e'_n \rangle$ if and only if the polynomial identity $(z - e_1) \cdot (z - e_n) - (z - e'_1) \cdot (z - e'_n) = 0$ holds for all z. For any $\varepsilon > 0$ let p denote a prime such that $p > \max\{n/\varepsilon, e_1, \ldots, e_n, e'_1, \ldots, e'_n\}$. Now the idea is to evaluate the above polynomial mod p for a random value $z \in 0..p-1$. Show that if $\langle e_1, \ldots, e_n \rangle$ is *not* a permutation of $\langle e'_1, \ldots, e'_n \rangle$ then the result of the evaluation is zero with probability at most ε. Hint: A nonzero polynomial of degree n has at most n zeroes.

5.1 Simple Sorters

Perhaps the conceptually simplest sorting algorithm is *selection sort*: Start with an empty output sequence. Select the smallest element from the input sequence, delete it, and add it to the end of the output sequence. Repeat this process until the input sequence is exhausted. Here is an example

$$\langle\rangle, \langle 4,7,1,1 \rangle \rightsquigarrow \langle 1 \rangle, \langle 4,7,1 \rangle \rightsquigarrow \langle 1,1 \rangle, \langle 4,7 \rangle \rightsquigarrow \langle 1,1,4 \rangle, \langle 7 \rangle \rightsquigarrow \langle 1,1,4,7 \rangle, \langle\rangle \ .$$

Exercise 5.6 asks you to give a simple array implementation of this idea that works *in-place*, i.e. needs no additional storage beyond the input array and a constant amount of space for loop counters etc. In Section **??**[todo] we will learn about a more sophisticated implementation where the input sequence is maintained as a *priority queue* that supports repeated selection of the minimum element very efficiently. This algorithm runs in time $O(n \log n)$ and is one of the more useful sorting algorithms. In particular, it is perhaps the simplest efficient algorithm that is deterministic and works in-place.

⟹

Exercise 5.6 (Simple selection sort.) Implement a simple variant of selection sort that sorts an array with n elements in time $O(n^2)$ by repeatedly scanning the input sequence. The algorithm should be in-place, i.e., both the input sequence and the output sequence should share the same array.

Procedure *insertionSort*(a : **Array** $[1..n]$ **of** *Element*)
  for $i := 2$ **to** n **do**
    invariant $a[1] \leq \cdots \leq a[i-1]$ // a: 1..$i-1$: sorted | $i..n$: unsorted
    // Move $a[i]$ to the right place
    $e := a[i]$ // a: sorted | e | $i+1..n$
    if $e < a[1]$ **then** // new minimum
      for $j := i$ **downto** 2 **do** $a[j] := a[j-1]$ // a: | sorted $> e$ | $i+1..n$
      $a[1] := e$ // a: e | sorted $> e$ | $i+1..n$
    else // Use $a[1]$ as a sentinel
      for $j := i$ **downto** $-\infty$ **while** $a[j-1] > e$ **do** $a[j] := a[j-1]$
      $a[j] := e$ // a: $\leq e$ | e | $> e$ | $i+1..n$

Figure 5.1: Insertion sort

Selection sort maintains the invariant that the output sequence is always sorted by carefully choosing the element to be deleted from the input sequence. Another simple algorithm, *insertion sort*, maintains the same invariant by choosing an arbitrary element of the input sequence but taking care to insert this element at the right place in the output sequence. Here is an example

$$\langle\rangle, \langle 4,7,1,1 \rangle \rightsquigarrow \langle 4 \rangle, \langle 7,1,1 \rangle \rightsquigarrow \langle 4,7 \rangle, \langle 1,1 \rangle \rightsquigarrow \langle 1,4,7 \rangle, \langle 1 \rangle \rightsquigarrow \langle 1,1,4,7 \rangle, \langle\rangle \ .$$

Figure 5.1 gives an in-place array implementation of insertion sort. This implementation is straightforward except for a small trick that allows the inner loop to use only a single comparison. When the element e to be inserted is smaller than all previously inserted elements, it can be inserted at the beginning without further tests. Otherwise, it suffices to scan the sorted part of a from right to left while e is smaller than the current element. This process has to stop because $a[1] \leq e$. In the worst case, insertion sort is quite slow. For example, if the input is sorted in decreasing order, each input element is moved all the way to $a[1]$, i.e., in iteration i of the outer loop, i elements have to be moved. Overall, we get

$$\sum_{i=2}^{n} (i-1) = -n + \sum_{i=1}^{n} i = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} = \Omega\left(n^2\right)$$

movements of elements (see also Equation (A.7).

Nevertheless, insertion sort can be useful. It is fast for very small inputs (say $n \leq 10$) and hence can be used as the base case in divide-and-conquer algorithms for

sorting. Furthermore, in some applications the input is already "almost" sorted and in that case insertion sort can also be quite fast.

Exercise 5.7 (Almost sorted inputs.) Prove that insertion sort executes in time $O(kn)$ if for all elements e_i of the input, $|r(e_i) - i| \leq k$ where r defines the *rank* of e_i (see Section 5.5 or Appendix A).

Exercise 5.8 (Average case analysis.) Assume the input to the insertion sort algorithm in Figure 5.1 is a permutation of the numbers 1,...,n. Show that the average execution time over all possible permutations is $\Omega(n^2)$. Hint: Argue formally that about one third of the input elements in the right third of the array have to be moved to the left third of the array. Using a more accurate analysis you can even show that on the average $n^2/4 - O(n)$ iterations of the inner loop are needed.

Exercise 5.9 (Insertion sort with few comparisons.) Modify the inner loops of the array based insertion sort algorithm from Figure 5.1 so that it needs only $O(n \log n)$ comparisons between elements. Hint: Use binary search[ref].

Exercise 5.10 (Efficient insertion sort?) Use the data structure for sorted sequences from Chapter 7 to derive a variant of insertion sort that runs in time $O(n \log n)$.

Exercise 5.11 (Formal verification.) Insertion sort has a lot of places where one can make errors. Use your favorite verification formalism, e.g. Hoare calculus, to prove that insertion sort is correct. Do not forget to prove that the output is a permutation of the input.

5.2 Mergesort — an $O(n \log n)$ Algorithm

[kann man das Mergesoertbild kompakter machen? z.B. v. links nach rechts statt v. oben nach unten. Ausserdem "analog" zum quicksort bild?] Mergesort is a straightforward application of the divide-and-conquer principle. The unsorted sequence is split into two about equal size parts. The parts are sorted recursively and a globally sorted sequence is computed from the two sorted pieces. This approach is useful because merging two sorted sequences a and b is much easier than sorting from scratch. The globally smallest element is either the first element of a or the first element of b. So we move this element to the output, find the second smallest element using the same approach and iterate until all elements have been moved to the output. Figure 5.2 gives pseudocode and Figure 5.3 illustrates an example execution. The merging part is elaborated in detail using the list operations introduced in Section 3.2.


```
Function mergeSort(⟨e_1,...,e_n⟩) : Sequence of Element
    if n = 1 then return ⟨e_1⟩                                  // base case
    else return merge(mergeSort(e_1,...,e_⌊n/2⌋),
                      mergeSort(e_⌊n/2⌋+1,...,e_n))

// Merging for sequences represented as lists
Function merge(a,b : Sequence of Element) : Sequence of Element
    c := ⟨⟩
    loop
        invariant a, b, and c are sorted
        invariant ∀e ∈ c, e' ∈ a ∪ b : e ≤ e'
        if a.isEmpty then          c.concat(b) return c          // c ⟨ ⟨⟩ ⟩ a ⟩ b
        if b.isEmpty then          c.concat(a) return c          // c ⟨ ⟨⟩ ⟩ a ⟩ b
        if a.first ≤ b.first then  c.moveToBack(a.first)         // c ⟨ ⟨ ⟩ ⟩ a ⟩ b
        else                       c.moveToBack(b.first)         // c ⟨ ⟨ ⟩ ⟩ a ⟩ b
```

Figure 5.2: Mergesort

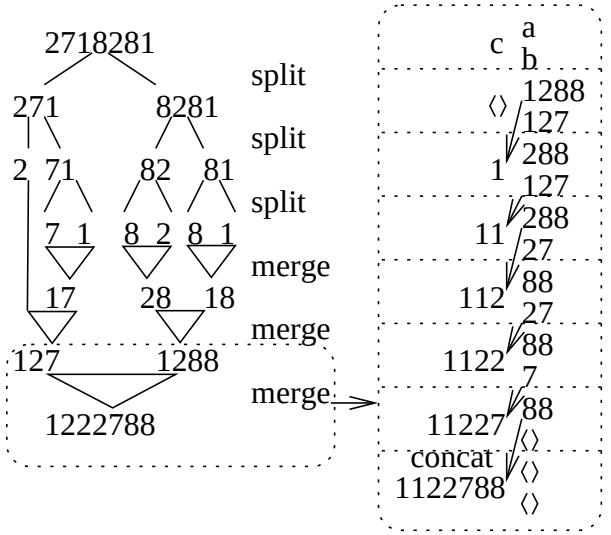


Figure 5.3: Execution of $mergeSort(\langle 2,7,1,8,2,8,1 \rangle)$.

Note that no allocation and deallocation of list items is needed. Each iteration of the inner loop of *merge* performs one element comparison and moves one element to the output. Each iteration takes constant time. Hence merging runs in linear time.

Theorem 5.1 *Function merge applied to sequences of total length n executes in time $O(n)$ and performs at most $n-1$ element comparisons.*

For the running time of mergesort we get.

Theorem 5.2 *Mergesort runs in time $O(n \log n)$ and performs no more than $n \log n$ element comparisons.*

Proof: Let $C(n)$ denote the number of element comparisons performed. We have $C(0) = C(1) = 0$ and $C(n) \leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1$ using Theorem 5.1. By
$\Longrightarrow$ Equation (**??**)[todo], this recurrence has the solution

$$C(n) \leq n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \leq n \log n \ .$$

The bound for the execution time can be verified using a similar recurrence relation. ∎

Mergesort is the method of choice for sorting linked lists and is therefore frequently used in functional and logical programming languages that have lists as their primary data structure. In Section 5.3 we will see that mergesort performs basically the minimal number of comparisons needed so that it is also a good choice if comparisons are expensive. When implemented using arrays, mergesort has the additional advantage that it streams through memory in a sequential way. This makes it efficient in memory hierarchies. Section 5.7 has more on that issue. Mergesort is still not the usual method of choice for an efficient array based implementation since *merge* does not work in-place. (But see Exercise 5.17 for a possible way out.)

Exercise 5.12 Explain how to insert k new elements into a sorted list of size n in time $O(k \log k + n)$.

Exercise 5.13 Explain how to implement the high level description of routine *mergeSort* with the same list interface from Chapter 3 that is used for *merge*.

Exercise 5.14 Implement mergesort in your favorite functional programming language.

Exercise 5.15 Give an efficient array based implementation of mergesort in your favorite imperative programming language. Besides the input array, allocate one auxiliary array of size n at the beginning and then use these two arrays to store all intermediate results. Can you improve running time by switching to insertion sort for small inputs? If so, what is the optimal switching point in your implementation?

Exercise 5.16 The way we describe *merge*, there are three comparisons for each loop iteration — one element comparison and two termination tests. Develop a variant using sentinels that needs only one termination test. How can you do it without appending dummy elements to the sequences?

Exercise 5.17 Exercise 3.19 introduces a list-of-blocks representation for sequences. Implement merging and mergesort for this data structure. In merging, reuse emptied input blocks for the output sequence. Compare space and time efficiency of mergesort for this data structure, plain linked lists, and arrays. (Constant factors matter.)

5.3 A Lower Bound

It is a natural question whether we can find a sorting algorithm that is asymptotically faster than mergesort. Can we even achieve time $O(n)$? The answer is (almost) no. The reason can be explained using the analogy of a sports competition. Assume your local tennis league decides that it wants to find a complete ranking of the players, i.e., a mapping r from the n players to $1..n$ such that $r(x) < r(y)$ if and only if player x is better than player y. We make the (unrealistic) assumption that a single match suffices to sample the 'plays-better-than' relation '$<$' and that this relation represents a total order. Now we ask ourselves what is the minimal number of matches needed to find the ranking. The analogy translates back to sorting if we restrict ourselves to algorithms that use comparisons only to obtain information about keys. Such algorithms are called *comparison based* algorithms.

The lower bound follows from two simple observations. Every comparison (tennis match) gives us only one bit of information about the ranking and we have to distinguish between $n!$ different possible rankings. After T comparisons, we can distinguish between at most 2^T inputs. Hence we need

$$2^T \geq n! \text{ or } T \geq \log n!$$

comparisons to distinguish between all possible inputs. The rest is arithmetics. By Stirling's approximation of the factorial (Equation (A.10)) we get

$$T \geq \log n! \geq \log \left(\frac{n}{e}\right)^n = n \log n - n \log e \ .$$

Theorem 5.3 *Any comparison based sorting algorithm needs $n\log n - O(n)$ comparisons in the worst case.*

Using similar arguments, Theorem 5.3 can be strengthened further. The same bound (just with different constants hidden in the linear term) applies on the average, i.e., worst case sorting problems are not much more difficult than randomly permuted inputs. Furthermore, the bound even applies if we only want to solve the seemingly simpler problem of checking whether an element appears twice in a sequence.

Exercise 5.18 Exercise 4.3 asks you to count occurences in a file.

a) Argue that any comparison based algorithm needs time at least $O(n\log n)$ to solve the problem on a file of length n.

b) Explain why this is no contradiction to the fact that the problem can be solved in linear time using hashing.

Exercise 5.19 (Sorting small inputs optimally.) Give an algorithm for sorting k element using at most $\lceil \log k! \rceil$ element comparisons.

a) For $k \in \{2,3,4\}$. Hint: use mergesort.

*****b) For $k = 5$ (seven comparisons). Implement this algorithm efficiently and use it as the base case for a fast implementation of mergesort.

c) For $k \in \{6,7,8\}$. Hint: use the case $k = 5$ as a subroutine.

$\Longrightarrow$ [ask Jyrky for sources]

$\Longrightarrow$ [nice geometric example for a lower bound based on sorting]

5.4 Quicksort

$\Longrightarrow$ [kann man die Bilder fuer merge sort und quicksort so malen, dass sie dual zueinander aussehen?] Quicksort is a divide-and-conquer algorithm that is complementary to the mergesort algorithm we have seen in Section 5.2. Quicksort does all the difficult work *before* the recursive calls. The idea is to distribute the input elements to two or more sequences that represent disjoint ranges of key values. Then it suffices to sort the shorter sequences recursively and to concatenate the results. To make the symmetry to mergesort complete, we would like to split the input into two sequences of equal size. Since this would be rather difficult, we use the easier approach to pick a random splitter elements or *pivot p*. Elements are classified into three sequences

Function *quickSort*(s : *Sequence* **of** *Element*) : *Sequence* **of** *Element*
  if $|s| \leq 1$ **then return** s // base case
  pick $p \in s$ uniformly at random // pivot key
  $a := \langle e \in s : e < p \rangle$ // (A)
  $b := \langle e \in s : e = p \rangle$ // (B)
  $c := \langle e \in s : e > p \rangle$ // (C)
  return *concatenation of quickSort*(a), *b, and quickSort*(c)

Figure 5.4: Quicksort

```
quickSort           qsort               i->  partition  <-j
3 6 8 1 0 7 2 4 5 9  3 6 8 1 0 7 2 4 5 9  3 6 8 1 0 7 2 4 5 9
1 0 2|3|6 8 7 4 5 9  2 0 1|8 6 7 3 4 5 9  2 6 8 1 0 7 3 4 5 9
0|1|2   4 5|6|8 7 9  1 0|2|5 6 7 3 4|8 9  2 0 8 1 6 7 3 4 5 9
        4|5   7|8|9  0 1| |4 3|7 6 5|8 9  2 0 1|8 6 7 3 4 5 9
0 1 2 3 4 5 6 7 8 9     | |3 4|5 6|7|         j i
                        | |   |5 6| |
                      0 1 2 3 4 5 6 7 8 9
```

Figure 5.5: Execution of *quickSort* (Figure 5.4) and *qSort* (Figure 5.6) on $\langle 2,7,1,8,2,8,1 \rangle$ using the first character of a subsequence as the piviot. The right block shows the first execution of the repeat loop for partitioning the input in *qSort*.

a, b, and c of elements that are smaller, equal to, or larger than p respectively. Figure 5.4 gives a high level realization of this idea and Figure 5.5 depicts an example. This simple algorithm is already enough to show expected execution time $O(n \log n)$ in Section 5.4.1. In Section 5.4.2 we then discuss refinements that make quicksort the most widely used sorting algorithm in practice.

5.4.1 Analysis

To analyze the running time of quicksort for an input sequence $s = \langle e_1, \ldots, e_n \rangle$ we focus on the number of element comparisons performed. Other operations contribute only constant factors and small additive terms in the execution time.

Let $C(n)$ denote the worst case number of comparisons needed for any input sequence of size n and any choice of random pivots. The worst case performance is easily determined. Lines (A), (B), and (C) in Figure 5.4. can be implemented in such a way that all elements except for the pivot are compared with the pivot once (we allow *three-way* comparisons here, with possible outcomes 'smaller', 'equal', and 'larger'). This makes $n-1$ comparisons. Assume there are k elements smaller than the pivot and k' elements larger than the pivot. We get $C(0) = C(1) = 0$ and

$$C(n) = n - 1 + \max\{C(k) + C(k') : 0 \le k \le n-1, 0 \le k' < n-k\} \quad .$$

By induction it is easy to verify that

$$C(n) = \frac{n(n-1)}{2} = \Theta(n^2) \quad .$$

The worst case occurs if all elements are different and we are always so unlucky to pick the largest or smallest element as a pivot.

The expected performance is much better.

Theorem 5.4 *The expected number of comparisons performed by quicksort is*

$$\bar{C}(n) \le 2n \ln n \le 1.4 n \log n \quad .$$

We concentrate on the case that all elements are different. Other cases are easier because a pivot that occurs several times results in a larger middle sequence b that need not be processed any further.

Let $s' = \langle e'_1, \ldots, e'_n \rangle$ denote the elements of the input sequence in sorted order. Elements e'_i and e'_j are compared at most once and only if one of them is picked as a pivot. Hence, we can count comparisons by looking at the indicator random variables X_{ij}, $i < j$ where $X_{ij} = 1$ if e'_i and e'_j are compared and $X_{ij} = 0$ otherwise. We get

$$\bar{C}(n) = \mathrm{E}[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{prob}(X_{ij} = 1) \quad .$$

The middle transformation follows from the linearity of expectation (Equation (A.2)). The last equation uses the definition of the expectation of an indicator random variable $\mathrm{E}[X_{ij}] = \mathrm{prob}(X_{ij} = 1)$. Before we can further simplify the expression for $\bar{C}(n)$, we need to determine this probability.

Lemma 5.5 *For any $i < j$, $\mathrm{prob}(X_{ij} = 1) = \dfrac{2}{j-i+1}$.*

Proof: Consider the $j-i+1$ element set $M = \{e'_i, \ldots, e'_j\}$. As long as no pivot from M is selected, e'_i and e'_j are not compared but all elements from M are passed to the same recursive calls. Eventually, a pivot p from M is selected. Each element in M has the same chance $1/|M|$ to be selected. If $p = e'_i$ or $p = e'_j$ we have $X_{ij} = 1$. The probability for this event is $2/|M| = 2/(j-i+1)$. Otherwise, e'_i and e'_j are passed to different recursive calls so that they will never be compared. ∎

Now we can finish the proof of Theorem 5.4 using relatively simple calculations.

$$\begin{aligned} \bar{C}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{prob}(X_{ij} = 1) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\le \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} = 2n \sum_{k=2}^{n} \frac{1}{k} = 2n(H_n - 1) \le 2n(\ln n + 1 - 1) = 2n \ln n \quad . \end{aligned}$$

For the last steps, recall the properties of the harmonic number $H_n := \sum_{k=1}^{n} 1/k \le \ln n + 1$ (Equation A.8). ∎

Note that the calculations in Section 2.6 for left-right maxima were very similar although we had a quite different problem at hand.

5.4.2 Refinements

[implement] ⟸

Figure 5.6 gives pseudocode for an array based quicksort that works in-place and uses several implementation tricks that make it faster and very space efficient.

To make a recursive algorithm compatible to the requirement of in-place sorting of an array, quicksort is called with a reference to the array and the range of array indices to be sorted. Very small subproblems with size up to n_0 are sorted faster using a simple algorithm like the insertion sort from Figure 5.1.[3] The best choice for the

[3] Some books propose to leave small pieces unsorted and clean up at the end using a single insertion sort that will be fast according to Exercise 5.7. Although this nice trick reduces the number of instructions executed by the processor, our solution is faster on modern machines because the subarray to be sorted will already be in cache.

```
// Sort the subarray a[ℓ..r]
Procedure qSort(a : Array of Element; ℓ,r : ℕ)
    while r − ℓ ≥ n0 do                                   // Use divide-and-conquer
        j := pickPivotPos(a,l,r)
        swap(a[ℓ],a[j])                                  // Helps to establish the invariant
        p := a[ℓ]
        i := ℓ; j := r
        repeat                                           // a:  ℓ      i→  ←j      r
            invariant 1: ∀i′ ∈ ℓ..i−1 : a[i′] ≤ p         // a: [∀ ≤ p |          ]
            invariant 2: ∀j′ ∈ j+1..r: a[j′] ≥ p          // a: [      |      ∀ ≥ p]
            invariant 3: ∃i′ ∈ i..r     : a[i′] ≥ p       // a: [      | ∃ ≥ p    ]
            invariant 4: ∃j′ ∈ ℓ..j     : a[j′] ≤ p       // a: [  ∃ ≤ p |          ]
            while a[i] < p do i++                        // Scan over elements (A)
            while a[j] > p do j−−                        // on the correct side (B)
            if i ≤ j then swap(a[i],a[j]); i++ ; j−−
        until i > j                                      // Done partitioning
        if i < (l+r)/2 then qSort(a,ℓ,j); ℓ := j+1
        else    qSort(a,i,r) ; r := i−1
    insertionSort(a[l..r])                               // faster for small r − l
```

Figure 5.6: Refined quicksort

constant n_0 depends on many details of the machine and the compiler. Usually one should expect values around 10–40.

The pivot element is chosen by a function *pickPivotPos* that we have not specified here. The idea is to find a pivot that splits the input more accurately than just choosing a random element. A method frequently used in practice chooses the median ('middle') of three elements. An even better method would choose the exact median of a random sample of elements. [crossref to a more detailed explanation of this concept?] ⟸

The repeat-until loop partitions the subarray into two smaller subarrays. Elements equal to the pivot can end up on either side or between the two subarrays. Since quicksort spends most of its time in this partitioning loop, its implementation details are important. Index variable i scans the input from left to right and j scans from right to left. The key invariant is that elements left of i are no larger than the pivot whereas elements right of j are no smaller than the pivot. Loops (A) and (B) scan over elements that already satisfiy this invariant. When $a[i] \geq p$ and $a[j] \leq p$, scanning can be continued after swapping these two elements. Once indices i and j meet, the partitioning is completed. Now, $a[\ell..j]$ represents the left partition and $a[i..r]$ represents the right partition. This sounds simple enough but for a correct and fast implementation, some subtleties come into play.

To ensure termination, we verify that no single piece represents all of $a[\ell..r]$ even if p is the smallest or largest array element. So, suppose p is the smallest element. Then loop A first stops at $i = \ell$; loop B stops at the last occurence of p. Then $a[i]$ and $a[j]$ are swapped (even if $i = j$) and i is incremented. Since i is never decremented, the right partition $a[i..r]$ will not represent the entire subarray $a[\ell..r]$. The case that p is the largest element can be handled using a symmetric argument.

The scanning loops A and B are very fast because they make only a single test. On the first glance, that looks dangerous. For example, index i could run beyond the right boundary r if all elements in $a[i..r]$ were smaller than the pivot. But this cannot happen. Initially, the pivot is in $a[i..r]$ and serves as a sentinel that can stop Scanning Loop A. Later, the elements swapped to the right are large enough to play the role of a sentinel. Invariant 3 expresses this requirement that ensures termination of Scanning Loop A. Symmetric arguments apply for Invariant 4 and Scanning Loop B.

Our array quicksort handles recursion in a seemingly strange way. It is something like "semi-recursive". The smaller partition is sorted recursively, while the larger partition is sorted iteratively by adjusting ℓ and r. This measure ensures that recursion can never go deeper than $\lceil \log \frac{n}{n_0} \rceil$ levels. Hence, the space needed for the recursion stack is $O(\log n)$. Note that a completely recursive algorithm could reach a recursion depth of $n-1$ so the the space needed for the recursion stack could be considerably larger than for the input array itself.

***Exercise 5.20 (Sorting Strings using Multikey Quicksort [12])** Explain why $mkqSort(s,1)$ below correctly sorts a sequence s consisting of n *different* strings. Assume that for any $e \in s$, $e[|e|+1]$ is an end-marker character that is different from all "normal" characters. What goes wrong if s contains equal strings? Fix this problem. Show that the expected execution time of *mkqSort* is $O(N + n\log n)$ if $N = \sum_{e\in s}|e|$.

```
Function mkqSort(s : Sequence of String, i : ℕ) : Sequence of String
    assert ∀e.e' ∈ s : e[1..i−1] = e'[1..i−1]
    if |s| ≤ 1 then return s                                  // base case
    pick p ∈ s uniformly at random                            // pivot character
    return concatenation of  mkqSort(⟨e ∈ s : e[i] < p[i]⟩, i),
                             mkqSort(⟨e ∈ s : e[i] = p[i]⟩, i+1), and
                             mkqSort(⟨e ∈ s : e[i] > p[i]⟩, i)
```

5.5 Selection

Often we want to solve problems that are related to sorting but do not need the complete sorted sequence. We can then look for specialized algorithms that are faster. For example, when you want to find the smallest element of a sequence, you would hardly sort the entire sequence and then take the first element. Rather, you would just scan the sequence once and remember the smallest element. You could quickly come up with algorithms for finding the second smallest element etc. More generally, we could ask for the k-th smallest element or all the k smallest elments in arbitrary order. In particular, in statistical problems, we are often interested in the *median* or $n/2$-th smallest element. Is this still simpler than sorting?

To define the term "k-th smallest" if elements can be equal, it is useful to formally introduce the notion of *rank* of an element. A *ranking function* r is a one-to-one mapping of elements of a sequence $\langle e_1,\ldots,e_n\rangle$ to the range $1..n$ such that $r(x) < r(y)$ if $x < y$. Equal elements are ranked arbitrarily. A k-th smallest element is then an element that has rank k for some ranking function.[is this the common def. I could not find this anywhere?]

Once we know quicksort, it is remarkably easy to modify it to obtain an efficient selection algorithm from it. This algorithm is therefore called quickselect. The key observation is that it is always sufficient to follow at most one of the recursive calls. Figure 5.7 gives an adaption for the simple sequence based quicksort from Figure 5.4. As before, the sequences a, b, and c are defined to contain the elements smaller than the pivot, equal to the pivot, and larger than the pivot respectively. If $|a| \geq k$, it suffices to restrict selection to this problem. In the borderline case that $|a| < k$ but $|a|+|b| \geq k$, the pivot is an element with rank k and we are done. Note that this also covers the case $|s| = k = 1$ so that no separate base case is needed. Finally, if $|a|+|b| < k$ the elements


```
// Find an element with rank k
Function select(s : Sequence of Element; k : ℕ) : Element
    assert |s| ≥ k
    pick p ∈ s uniformly at random                            // pivot key
    a := ⟨e ∈ s : e < p⟩
    if |a| ≥ k then return select(a,k)                        //  a
    b := ⟨e ∈ s : e = p⟩
    if |a|+|b| ≥ k then return p                              //  a | b
    c := ⟨e ∈ s : e > p⟩
    return select(c, k−|a|−|b|)                               //  a | b | c
```

Figure 5.7: Quickselect

in a and b are too small for a rank k element so that we can restrict our attention to c. We have to search for an element with rank $k-|a|+|b|$.

The table below illustrates the levels of recursion entered by $select(\langle 3,1,4,5,9,2,6,5,3,5,8\rangle, 6) = 5$ assuming that the middle element of the current s is used as the pivot p.

s	k	p	a	b	c
$\langle 3,1,4,5,9,2,6,5,3,5,8\rangle$	6	2	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3,4,5,9,6,5,3,5,8\rangle$
$\langle 3,4,5,9,6,5,3,5,8\rangle$	4	6	$\langle 3,4,5,5,3,4\rangle$	-	-
$\langle 3,4,5,5,3,5\rangle$	4	5	$\langle 3,4,3\rangle$	$\langle 5,5,5\rangle$	-

As for quicksort, the worst case execution time of quickselect is quadratic. But the expected execution time is only linear saving us a logarithmic factor over the execution time of quicksort.

Theorem 5.6 *Algorithm quickselect runs in expected time $O(n)$ for an input of size $|s| = n$.*

Proof: An analysis that does not care about constant factors is remarkably easy to obtain. Let $T(n)$ denote the expected execution time of quickselect. Call a pivot *good* if neither $|a|$ nor $|b|$ are larger than $2n/3$. Let $\rho \geq 1/3$ denote the probability that p is good. We now make the conservative assumption that the problem size in the recursive call is only reduced for good pivots and that even then it is only reduced by a factor of $2/3$. Since the work outside the recursive call is linear in n, there is an appropriate

constant c such that

$$T(n) \le cn + \rho T\left(\frac{2n}{3}\right) + (1-\rho)T(n) \text{ or, equivalently}$$
$$T(n) \le \frac{cn}{\rho} + T\left(\frac{2n}{3}\right) \le 3cn + T\left(\frac{2n}{3}\right) \quad .$$

$\Longrightarrow$ Now the master Theorem **??** [todo: need the version for arbitrary n.] for recurrence relations yields the desired linear bound for $T(n)$. ∎

$\Longrightarrow$ [approximate median plus filter algorithm in exercise?]

Exercise 5.21 Modify the algorithm in Figure 5.7 such that it returns the k smallest elements.

Exercise 5.22 Give a selection algorithm that permutes an array in such a way that the k smallest elements are in entries $a[1],\ldots,a[k]$. No further ordering is required except that $a[k]$ should have rank k. Adapt the implementation tricks from array based quicksort to obtain a nonrecursive algorithm with fast inner loops.

$\Longrightarrow$ [some nice applications]
$\Longrightarrow$ [in PQ chapter?]

Exercise 5.23 (Streaming selection.)

a) Develop an algorithm that finds the k-th smallest element of a sequence that is presented to you one element at a time in an order you cannot control. You have only space $O(k)$ available. This models a situation where voluminous data arrives over a network or at a sensor.

b) Refine your algorithm so that it achieves running time $O(n \log k)$. You may want to read some of Chapter 6 first.

*c) Refine the algorithm and its analysis further so that your algorithm runs in average case time $O(n)$ if $k = O\left(n/\log^2 n\right)$. Here, average means that all presentation orders of elements in the sequence are equally likely.

5.6 Breaking the Lower Bound

Lower bounds are not so much a verdict that something cannot be improved but can be viewed as an opportunity to break them. Lower bounds often hold only for a restricted

class of algorithms. Evading this class of algorithms is then a guideline for getting faster. For sorting, we devise algorithms that are not comparison based but extract more information about keys in constant time. Hence, the $\Omega(n \log n)$ lower bound for comparison based sorting does not apply.

Let us start with a very simple algorithm that is fast if the keys are small integers in the range $0..K-1$. This algorithm runs in time $O(n+K)$. We use an array $b[0..K-1]$ of *buckets* that are initially empty. Then we scan the input and insert an element with key k into bucket $b[k]$. This can be done in constant time per element for example using linked lists for the buckets. Finally, we append all the nonempty buckets to obtain a sorted output. Figure 5.8 gives pseudocode. For example, if elements are pairs whose first element is a key in range 0..3 and

$$s = \langle (3,a),(1,b),(2,c),(3,d),(0,e),(0,f),(3,g),(2,h),(1,i)\rangle$$

we get $b = [\langle (0,e),(0,f)\rangle,\ \langle (1,b),(1,i)\rangle,\ \langle (2,c),(2,h)\rangle,\ \langle (3,a),(3,d),(3,g)\rangle]$ and the sorted output $\langle (0,e),(0,f),(1,b),(1,i),(2,c),(2,h),(3,a),(3,d),(3,g)\rangle$.

Procedure *KSort*(s : *Sequence* **of** *Element*)
  b=$\langle\langle\rangle,\ldots,\langle\rangle\rangle$: **Array** $[0..K-1]$ **of** *Sequence* **of** *Element*
  foreach $e \in s$ **do** $b[key(e)].pushBack(e)$ // s e
  s := *concatenation of* $b[0],\ldots,b[K-1]$
b[0] b[1] b[2] b[3] b[4]

Figure 5.8: Sorting with keys in the range $0..K-1$.

Procedure *LSDRadixSort*(s : *Sequence* **of** *Element*)
  for $i := 0$ **to** $d-1$ **do**
    redefine *key*(x) as (x **div** K^i) **mod** K
    KSort(s)
    invariant s is sorted with respect to digits $0..i$

// x [d−1] ··· [i] ··· [1] [0]
digits
key(x)

Figure 5.9: Sorting with keys in the range $0..K^d-1$ using least significant digit radix sort.

KSort can be used as a building block for sorting larger keys. The idea behind *radix sort* is to view integer keys as numbers represented by digits in the range $0..K-1$. Then *KSort* is applied once for each digit. Figure 5.9 gives a radix sorting algorithm for keys in the range $0..K^d-1$ that runs in time $O(d(n+K))$. The elements are sorted

first by their least significant digit then by the second least significant digit and so on until the most significant digit is used for sorting. It is not obvious why this works. *LSDRadixSort* exploits the property of *KSort* that elements with the same $key(e)$ retain their relative order. Such sorting algorithms are called *stable*. Since *KSort* is stable, the elements with the same i-th digit remain sorted with respect to digits $0..i-1$ during the sorting process by digit i. For example, if $K = 10$, $d = 3$, and

$s = \langle 017, 042, 666, 007, 111, 911, 999 \rangle$, we successively get
$s = \langle 11\mathbf{1}, 91\mathbf{1}, 04\mathbf{2}, 66\mathbf{6}, 01\mathbf{7}, 00\mathbf{7}, 99\mathbf{9} \rangle$,
$s = \langle 0\mathbf{0}7, 1\mathbf{1}1, 9\mathbf{1}1, 0\mathbf{1}7, 0\mathbf{4}2, 6\mathbf{6}6, 9\mathbf{9}9 \rangle$, and
$s = \langle \mathbf{0}07, \mathbf{0}17, \mathbf{0}42, \mathbf{1}11, \mathbf{6}66, \mathbf{9}11, \mathbf{9}99 \rangle$.

The mechanical sorting machine shown on Page 75 basically implemented one pass of radix sort and was most likely used to run LSD radix sort.

Procedure *uniformSort*(s : *Sequence* **of** *Element*)
  n:= $|s|$
  b=$\langle\langle\rangle, \ldots, \langle\rangle\rangle$: **Array** $[0..n-1]$ **of** *Sequence* **of** *Element*
  foreach $e \in s$ **do** $b[\lfloor key(e) \cdot n \rfloor].pushBack(e)$
  for $i := 0$ **to** $n-1$ **do** sort $b[i]$ in time $O(|b[i]| \log |b[i]|)$
  s := *concatenation of* $b[0], \ldots, b[n-1]$

Figure 5.10: Sorting random keys in the range $[0,1)$.

Radix sort starting with the most significant digit (*MSD radix sort*) is also possible. We apply *KSort* to the most significant digit and then sort each bucket recursively. The only problem is that the buckets might be much smaller than K so that it would be expensive to apply *KSort* to small buckets. We then have to switch to another algorithm. This works particularly well if we can assume that the keys are uniformly distributed. More specifically, let us now assume that keys are real numbers with $0 \leq key(e) < 1$. Algorithm *uniformSort* from Figure 5.10 scales these keys to integers between 0 and $n-1 = |s|-1$, and groups them into n buckets where bucket $b[i]$ is responsible for keys in the range $[i/n, (i+1)/n$. For example, if $s = \langle 0.8, 0.4, 0.7, 0.6, 0.3 \rangle$ we get five buckets responsible for intervals of size 0.2 and

$$b = [\langle\rangle, \quad \langle 0.3 \rangle, \quad \langle 0.4 \rangle, \quad \langle 0.6, 0.7 \rangle, \quad \langle 0.8 \rangle]$$

and only $b[3] = \langle 0.7, 0.6 \rangle$ represents a nontrivial sorting subproblem.

We now show, that *uniformSort* is very efficient for *random* keys.

Theorem 5.7 *If keys are independent uniformly distributed random values in the range* $[0.1)$*, Algorithm uniformSort from Figure 5.10 sorts n keys in expected time $O(n)$ and worst case time $O(n \log n)$.*

Proof: We leave the worst case bound as an exercise and concentrate on the average case analysis. The total execution time T is $O(n)$ for setting up the buckets and concatenating the sorted buckets plus the time for sorting the buckets. Let T_i denote the time for sorting the i-th bucket. We get

$$E[T] = O(n) + E[\sum_{i<n} T_i] = O(n) + \sum_{i<n} E[T_i] = nE[T_0] \ .$$

The first "=" exploits the linearity of expectation (Equation (A.2)) and the second "=" exploits that all buckets sizes have the same distribution for uniformly distributed inputs. Hence, it remains to show that $E[T_0] = O(1)$. We prove the stronger claim that $E[T_0] = O(1)$ even if a quadratic time algorithm such as insertion sort is used for sorting the buckets. [cross ref with perfect hashing?] ⟸

Let $B_0 = |b[0]|$. We get $E[T_0] = O\left(E[B_0^2]\right)$. The random variable B_0 obeys a binomial distribution with n trials and success probability $1/n$. Using the definition of expected values (Equation (A.1)) and the binomial distribution defined in Equation (A.5) we get

$$E[B_0^2] = \sum_{i \leq n} i^2 \text{prob}(B_0 = i) = \sum_{i \leq n} i^2 \binom{n}{i} \frac{1}{n^i} \left(1 - \frac{1}{n}\right)^{n-i} .$$

It remains to show that this value is bounded by a constant independent of n. Using Inequality (A.6) and by estimating $(1-1/n)^{n-1} \leq 1$ we get

$$E[B_0^2] \leq \sum_{i \leq n} i^2 \left(\frac{ne}{i}\right)^i \frac{1}{n^i} = \sum_{i \leq n} i^2 \left(\frac{e}{i}\right)^i \ .$$

Finally, we can drop the restriction $i \leq n$ and get an infinite sum that is independent of n. It remains to show that this sum is bounded. By Cauchy's n-th root test, the sum is bounded if

$$\sqrt[i]{i^2 \left(\frac{e}{i}\right)^i} = i^{2/i} \frac{e}{i} < q$$

for some constant $q < 1$ and sufficiently large i. For $i \geq 6$ we get

$$i^{2/i} \frac{e}{i} \leq i^{1/3} \frac{e}{i} = \frac{e}{i^{2/3}} \leq \frac{e}{6^{2/3}} \leq 0.83 < 1 \ .$$

[More elegant proof? Kurt hatte was schoeneres?] ∎ ⟸

```
make_things_|as_simple_as|_possible_bu|t_no_simpler
 form run   | form run   | form run   | form run   |
__aeghikmnst|__aaeilmpsss|__bbeilopssu|__eilmnoprst
        \merge/                  \merge/
____aaaeeghiiklmmnpsssst |____bbeeiillmnoopprssstu
                      \merge/
_________aaabbeeeeghiiiiklllmmmnnooppprsssssssttu
```

Figure 5.11: An example of two-way mergesort with runs of length 12.

***Exercise 5.24** Implement an efficient sorting algorithm for elements with keys in the range $0..K-1$ that uses the data structure from Exercise 3.19 as input and output. Space consumption should be at most $n + O(n/B + KB)$ elements for n elements and blocks of size B.

5.7 External Sorting

Sorting large data sets that do not fit into internal memory is important. Not only because sorting is important anyway but also because sorting is a universal principle for coping with large data sets in particular. For example, it is no problem if you write the names of your ten best friends on a piece of paper without sorting them but you would be lost with an unsorted telephone book of Paris.

Recall that the external memory model introduced in Section 2.2 distinguishes between a fast internal memory of size M and a large external memory that is accessed with I/Os of block size B. As a starting point we look for internal memory sorting algorithms that can be implemented to run efficiently in external memory. Mergesort is a good candidate. Assume the input sequence is represented as an array in external memory. We describe a nonrecursive implementation for the case that the number of elements n is divisible by the number B of elements that can be stored in a block of external memory. We load *runs* of size M into internal memory, sort them using any algorithm we like and write them back. This *run formation phase* takes n/B blocks reads and n/B blocks writes, i.e., a total of $2n/B$ I/Os. Then we merge pairs of runs into larger runs in $\lceil \log(n/M) \rceil$ *merge phases* ending up with a single sorted run. Figure 5.11 gives an example for $n = 48$ and runs of length twelve.

It remains to explain how merging can be implemented I/O-efficiently. We have to support the following operations: inspecting the first element of the input sequences, removing the first element of an input sequence, and writing an element to the output sequence. This is easy to implement by just keeping one block of each sequence in internal memory. When the buffer block of an input sequence runs out of entries, we fetch the next block. When the buffer block of the output sequence fills up, we write it to the external memory. In each phase, we need n/B block reads and n/B block writes. Summing all I/Os we get $2(n/B + \lceil \log N/M \rceil)$ I/Os. The only requirement is that $M \geq 3B$.

Multiway Mergesort

We now generalize the external Mergesort to take full advantage of the available internal memory during merging. We can reduce the number of phases by merging as many runs as possible in a single phase. In *k-way merging*, we merge k sorted sequences into a single output sequence. Binary merging ($k = 2$) is easy to generalize. In each step we find the input sequence with the smallest first element. This element is removed and appended to the output sequence. External memory implementation is easy as long as we have enough internal memory for k input buffer blocks, one output buffer block, and a small amount of additional storage.

For each sequence, we need to remember which element we are currently considering. To find the smallest element among all k sequences, we keep their current element keys and positions in a *priority queue*. A priority queue maintains a set of elements supporting the operations insertion and deletion of the minimum. Chapter 6 explains how priority queues can be implemented so that insertion and deletion take time $O(\log k)$ for k elements. Figure 5.12 gives pseudocode for this algorithm. Figure 5.13 gives a snapshot of an execution of 4-way merging. This two-pass sorting algorithm sorts n elements using $4n/B$ I/Os — during run formation and merging everything is read and written exactly once. A single merging phase works if there is enough internal memory to store $\lceil n/M \rceil$ input buffer blocks, one output buffer block, and a priority queue with $\lceil n/M \rceil$ entries, i.e., we can sort up to $n \approx M^2/B$ elements. If internal memory stands for DRAMs and external memory stands for disks, this bound on n is no real restriction for all practical system configurations. For comparison with the I/O cost of binary mergesort it is nevertheless instructive to look at arbitrarily large inputs. Run formation works as before, but we now need $\left\lceil \log_{M/B}(N/M) \right\rceil$ merging phases to arrive at a single sorted run. Overall we need

$$2\frac{n}{B}\left(1 + \left\lceil \log_{M/B} \right\rceil \frac{N}{M}\right) \tag{5.1}$$

I/Os. The difference to binary merging is the much larger base of the logarithm.

Exercise 5.25 (Huge inputs.) Describe a generalization of *twoPassSort* in Figure 5.12 that also works for $n > M^2/B$, using $\lceil \log_{M/B} n/M \rceil$ merging phases.

Exercise 5.26 (Balanced systems.) Study the current market prices of computers, internal memory, and mass storage (currently hard disks). Also estimate the block size needed to achieve good bandwidth for I/O. Can you find any configuration where multi-way mergesort would require more than one merging phase for sorting an input filling all the disks in the system? If so, which fraction of the system cost would you have to spend on additional internal memory to go back to a single merging phase?

⟹ [something on implementing a join?]

⟹ [reinstate sample sort? otherwise say sth in further findings or perhaps in a big exercise]

5.8 Implementation Notes

Sorting algorithms are usually available in standard libraries so that you may not have to implement them yourself. Many good libraries use tuned implementations of quicksort. If you want to be faster than that you may have to resort to non comparison based algorithms. Even then, a careful implementation is necessary. Figure 5.14 gives an example for an array based implementation of the algorithm from Figure 5.8. Even this algorithm may be slower than quicksort for large inputs. The reason is that the distribution of elements to buckets causes a cache fault for every element.

To fix this problem one can use multi-phase algorithms similar to MSD radix sort. The number K of output sequences should be chosen in such a way that one block from each bucket is kept in the cache [4]. The distribution degree K can be larger when the subarray to be sorted fits in the cache. We can then switch to a variant of Algorithm *uniformSort* in Figure 5.10.

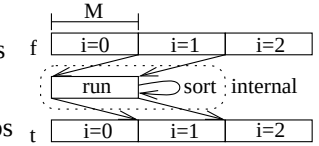
Another important practical aspect concerns the type of elements to be sorted. Sometimes, we have rather large elements that are sorted with respect to small keys. For example, you might want to sort an employee database by last name. In this situation, it makes sense to first extract the keys and store them in an array together with pointers to the original elements. Then only the key-pointer pairs are sorted. If the orginal elements need to brought into sorted order, they can be permuted accordingly in linear time using the sorted key-pointer pairs.

C/C++

Sorting is one of the few algorithms that is part of the C standard library. However, this function *qsort* is slower and less easy to use than the C++-function *sort*. The main reason is that *qsort* is passed a pointer to a function responsible for element comparisons.

[4] If there are M/B cache blocks this does *not* mean that we can use $k = M/B - 1$. A discussion of this issue can be found in [70].


```
Procedure twoPassSort(M : ℕ; a : external Array [0..n−1] of Element)
    b : external Array [0..n−1] of Element                    // auxiliary storage
    formRuns(M, a, b)
    mergeRuns(M, b, a)

// Sort runs of size M from f writing sorted runs to t
Procedure formRuns(M : ℕ; f, t : external Array [0..n−1] of Element)
    for i := 0 to n−1 step M do
        run := f[i..i+M−1]                    // M/B read steps
        sort(run)
        t[i..i+M−1] := run                    // M/B write steps

// Merge n elements from f to t where f stores sorted runs of size L
Procedure mergeRuns(L : ℕ; f, t : external Array [0..n−1] of Element)
    k := ⌈n/L⌉                                                // Number of runs
    next : PriorityQueue of Key × ℕ
    runBuffer := Array [0..k−1][0..B−1] of Element
    for i := 0 to k−1 do
        runBuffer[i] := f[iM..iM+B−1]
        next.insert(key(runBuffer[i][0]), iL))

    // k-way merging
    out : Array [0..B−1] of Element
    for i := 0 to n−1 step B do
        for j := 0 to B−1 do
            (x, ℓ) := next.deleteMin
            out[j] := runBuffer[ℓ div L][ℓ mod B]
            ℓ++
            if ℓ mod B = 0 then                                // New input block
                if ℓ mod L = 0 ∨ ℓ = n then
                    runBuffer[ℓ div L][0] := ∞                 // sentinel for exhausted run
                else runBuffer[ℓ div L] := f[ℓ..ℓ+B−1]          // refill buffer
            next.insert((runBuffer[ℓ div L][0], ℓ))
        write out to t[i..i+B−1]                               // One output step
```



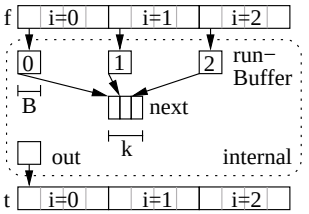


Figure 5.12: External k-way mergesort.

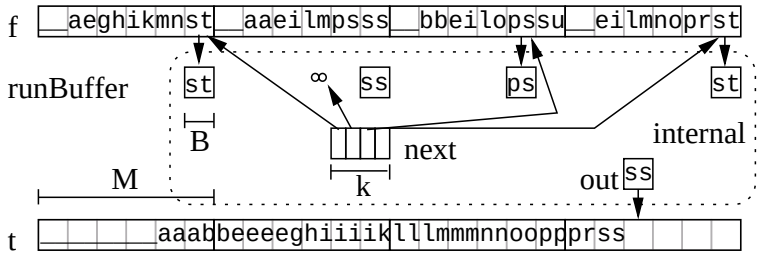


Figure 5.13: Execution of *twoPassSort* for the characters of "`make things as simple as possible but no simpler`". We have runs of size $M = 12$ and block size $B = 2$. Ties are broken so that the elements from the leftmost run is taken. The picture shows the moment of time just before a block with the third and fourth `s` is output. Note that the second run has already been exhausted.

```
Procedure KSortArray(a,b : Array [1..n] of Element)
    c=⟨0,...,0⟩ : Array [0..K−1] of ℕ          // counters for each bucket
    for i := 1 to n do c[key(a[i])]++          // Count bucket sizes
    C := 0
    for k := 0 to K−1 do (C,c[k]) := (C+c[k],C)   // Store ∑_{i<k} c[k] in c[k].
    for i := 1 to n do                         // Distribute a[i]
        b[c[key(a[i])]] := a[i]
        c[key(a[i])]++
```

Figure 5.14: Array based sorting with keys in the range $0..K-1$. The input is an unsorted array a. The output is b with the elements of a in sorted order.

This function has to be called for every single element comparison. In contrast, *sort* uses the template mechanism of C++ to figure out at compile time how comparisons are performed so that the code generated for comparisons is often a single machine instruction. The parameters passed to *sort* are an iterator pointing to the start of the sequence to be sorted and an iterator pointing after the end of the sequence. Hence, sort can be applied to lists, arrays, etc. In our experiments on an Intel Pentium III and `gcc` 2.95, sort on arrays runs faster than our manual implementation of quicksort. One possible reason is that compiler designers may tune there code optimizers until they find that good code for the library version of quicksort is generated.

Java

[todo: Sorting in the Java library] ⟸
4–8 way merging
more on streaming algorithms

Exercises

Exercise 5.27 Give a C or C++-implementation of the quicksort in Figure 5.6 that uses only two parameters. A pointer to the (sub)array to be sorted, and its size.

5.9 Further Findings

In this book you will find several generalizations of sorting. Chapter 6 discusses priority queues — a data structure that allows insertion of elements and deletion of the smallest element. In particular, by inserting n elements and then deleting them we get the elements in sorted order. It turns out that this approach yields some quite good sorting algorithm. A further generalization are the *search trees* introduced in Section 7 that can be viewed as a data structure for maintaining a sorted list supporting inserting, finding, and deleting elements in logarithmic time.

Generalizations beyond[check] the scope of this book are geometric problems on ⟸ higher dimensional point sets that reduce to sorting for special inputs (e.g., convex hulls or Delaunay triangulations [26]). Often, sorting by one coordinate is an important ingredient in algorithms solving such problems.

We have seen several simple, elegant, and efficient randomized algorithms in this chapter. An interesting theoretical question is whether these algorithms can be replaced by deterministic ones. Blum et al. [15] describe a deterministic median selection algorithm that is similar to the randomized algorithm from Section 5.5. This algorithm makes pivot selection more reliable using recursion: The pivot is the median of the $\lfloor n/5 \rfloor$ medians of $\langle e_{5i+1}, e_{5i+2}, e_{5i+3}, e_{5i+4}, e_{5i+5} \rangle$ for $0 \le i < n/5-1$. Working

out the resulting recurrences yields a linear time worst case execution time but the constant factors involved make this algorithms impractical. There are quite practical ways to reduce the expected number of comparisons required by quicksort. Using the median of three random elements yields an algorithm with about $1.188n\log n$ comparisons. The median of three three-medians brings this down to $\approx 1.094n\log n$ [10]. A "perfect" implementation makes the number of elements considered for pivot selection dependent on size of the subproblem. Martinez and Roura [62] show that for a subproblem of size m, the median of $\Theta(\sqrt{m})$ elements is a good choice for the pivot. The total number of comparisons required is then $(1+o(1))n\log n$, i.e., it matches the lower bound of $n\log n - O(n)$ up to lower order terms. A deterministic variant of quicksort that might be practical is proportion extend sort [21].

A classical sorting algorithm of some historical interest is *shell sort* [48, 85] — a quite simple generalization of insertion sort, that gains efficiency by also comparing nonadjacent elements. It is still open whether there might be a variant of Shellsort that achieves $O(n\log n)$ run time on the average [48, 63].

There are some interesting tricks to improve external multiway mergesort. The *snow plow* heuristics [57, Section 5.4.1] forms runs of size $2M$ on the average using a fast memory of size M: When an element e is read from disk, make room by writing the smallest element e' from the current run to disk. If $e \le e'$ insert e into the current run. Otherwise, remember it for the next run. Multiway merging can be slightly sped up using a *tournament tree* rather than general priority queues [57].[discuss in PQ chapter?]

To sort large data sets, we may also want to use parallelism. Multiway mergesort and distribution sort can be adapted to D parallel disks by *striping*, i.e., every D consecutive blocks in a run or bucket are evenly distributed over the disks. Using randomization, this idea can be developed into almost optimal algorithms that also overlaps I/O and computation [28]. Perhaps the best sorting algorithm for large inputs on P parallel processors is a parallel implementation of the sample sort algorithm from Section **??** [14].

[more lower bounds, e.g., selection, I/O? Or do it in detail?]

We have seen linear time algorithms for rather specialized inputs. A quite general model, where the $n\log n$ lower bound can be broken, is the *word model*. If keys are integers that can be stored in a memory word, then they can be sorted in time $O(n\log\log n)$ regardless of the word size as long as we assume that simple operations on[what exactly] words can be performed in constant time [5]. A possibly practical implementation of the distribution based algorithms from Section 5.6 that works almost in-place is *flash sort* [75].

Exercise 5.28 (Unix spell checking) One of the authors still finds the following spell checker most effective: Assume you have a dictionary consisting of a sorted sequence of correctly spelled words. To check a text, convert it to a sequence of words, sort it, scan text and dictionary simultaneously, and output the words in the text that do not appear in the dictionary. Implement this spell checker using any unix tools in as few lines as possible (one longish line might be enough).

[ssssort? skewed qsort? cache oblivious funnel sort? Cole's merge sort sort 13 elements? more than $\lceil \log n! \rceil$ vergleiche.]

Chapter 6

Priority Queues

Suppose you work for a company that markets tailor-made first-rate garments. Your business model works as follows: You organize marketing, measurements etc. and get 20% of the money paid for each order. Actually executing an order is subcontracted to an independent master taylor. When your company was founded in the 19th century there were five subcontractors in the home town of your company. Now you control 15 % of the world market and there are thousands of subcontractors worldwide.

Your task is to assign orders to the subcontractors. The contracts demand that an order is assigned to the taylor who has so far (this year) been assigned the smallest total amount of orders. Your ancestors have used a blackboard with the current sum of orders for each tailor. But for such a large number of subcontractors it would be prohibitive to go through the entire list everytime you get a new order. Can you come up with a more scaleable solution where you have to look only at a small number of values to decide who will be assigned the next order?

In the following year the contracts are changed. In order to encourage timely delivery, the orders are now assigned to the taylor with the smallest amount of unfinished *orders, i.e, whenever a finished order arrives, you have to deduct the value of the order from the backlog of the taylor who executed it. Is your strategy for assigning orders flexible enough to handle this efficiently?*

[Verweise auf Summary, Intro korrekt?] The data structure needed for the ⟸ above problem is a *priority queue* and shows up in many applications. But first let us look at a more formal specification. We maintain a set M of *Element*s with *Key*s. Every priority queue supports the following operations:

Procedure *build*$(\{e_1,\ldots,e_n\})$ $M := \{e_1,\ldots,e_n\}$
Procedure *insert*(e) $M := M \cup \{e\}$
Function *min* **return** $\min M$
Function *deleteMin* $e := \min M;$ $M := M \setminus \{e\};$ **return** e

⟹ [replaced findMin by min] [brauchen wir Min und size? Dann in allen Kapiteln
⟹ durchschleifen? Im Moment habe ich size weggelassen weil trivial] This is enough for the first part of our tailored example: Every year we build a new priority queue containing an *Element* with *Key* zero for each contract tailor. To assign an order, we delete the smallest *Element*, add the order value to its *Key*, and reinsert it. Section 6.1 presents a simple and efficient implementation of this basic functionality.

Addressable priority queues additionally support operations on arbitrary elements addressed by an element handle:

Function *remove*$(h : Handle)$ $e := h$; $M := M \setminus \{e\}$; **return** e
Procedure *decreaseKey*$(h : Handle, k : Key)$ **assert** $key(h) \geq k$; $key(h) := k$
Procedure *merge*(M') $M := M \cup M'$

⟹ [index terms: delete: see also remove, meld: see also merge].

In our example, operation *remove* might be helpful when a contractor is fired because it delivers poor quality. Together with *insert* we can also implement the "new contract rules": When an order is delivered, we remove the *Element* for the contractor who executed the order, subtract the value of the order from its *Key* value, and reinsert the *Element*. *DecreaseKey* streamlines this process to a single operation. In Section 6.2 we will see that this is not just convenient but that decreasing a *Key* can be implemented more efficiently than arbitrary element updates.

Priority queues support many important applications. For example, in Section 12.2 we will see that our tailored example can also be viewed as greedy algorithm for a very natural machine scheduling problem. Also, the rather naive selection sort algorithm from Section 5.1 can be implemented efficiently now: First insert all elements to be sorted into a priority queue. Then repeatedly delete the smallest element and output it. A tuned version of this idea is described in Section 6.1. The resulting *heapsort* algorithm is one of the most robust sorting algorithms because it is efficient for all inputs and needs no additional space.

In a *discrete event simulation* one has to maintain a set of pending events. Each event happens at some scheduled point in time [was execution time which can be misunderstood as the time the task/event takes to execute in the real world
⟹ or on the simulated computer.] and creates zero or more new events scheduled to happen at some time in the future. Pending events are kept in a priority queue. The main loop of the simulation deletes the next event from the queue, executes it, and inserts newly generated events into the priority queue. Note that priorities (times) of the deleted elements (simulated events) are monotonically increasing during the simulation. It turns out that many applications of priority queues have this monotonicity property. Section 10.5 explains how to exploit monotonicity for queues with integer keys.

Another application of monotone priority queues is the *best first branch-and-bound* approach to optimization described in Section 12.4.1. Here elements are partial solutions of an optimization problem and the keys are optimistic estimates of the obtainable solution quality. The algorithm repeatedly removes the best looking partial solution, refines it, and inserts zero or more new partial solutions.

We will see two applications of addressable priority queues in the chapters on graph algorithms. In both applications the priority queue stores nodes of a graph. Dijkstras algorithm for computing shortest paths in Section 10.4 uses a monotone priority queue where the keys are path lenghts. The Jarník-Prim algorithm for computing minimum spanning trees in Section 11.2 uses a (nonmonotone) priority queue where the keys are edge weights connecting a node to a spanning tree. In both algorithms, each edge can lead to a *decreaseKey* operation whereas there is at most one *insert* and *deleteMin* for each node. Hence, *decreaseKey* operations can dominate the running time if $m \gg n$. [moved this discussion here since it fits to the application overview] ⟸

Exercise 6.1 Show how to implement bounded non-addressable priority queues by arrays. The maximal size of the queue is w and when the queue has size n the first n entries of the array are used. Compare the complexity of the queue operations for two naive implementations. In the first implementation the array is unsorted and in the second implementation the array is sorted.

Exercise 6.2 Show how to implement addressable priority queues by doubly linked lists. Each list item represent an element in the queue and a handle is a handle of a list item. Compare the complexity of the queue operations for two naive implementations. In the first implementation the list is unsorted and in the second implementation the list is sorted.

6.1 Binary Heaps

[title was: Heaps] Heaps are a simple and efficient implementation of non-addressable ⟸
bounded priority queues. They can be made unbounded in the same way as bounded arrays are made unbounded in Section 3.1. Heaps can also be made addressable, but we will see better addressable queues in later sections.

We use an array $h[1..w]$ that stores the elements of the queue. The first n entries of the array are used. The array is *heap-ordered*, i.e.,

$$\text{if } 2 \leq j \leq n \text{ then } h[\lfloor j/2 \rfloor] \leq h[j].$$

[in the caption of Figure 6.2 the *tree* is heap ordered. Is this slight double meaning intended?] Figure 6.1 gives pseudocode for this basic setup. ⟸

Class *BinaryHeapPQ*($w : \mathbb{N}$) **of** *Element*
  h : **Array** $[1..w]$ **of** *Element* // The *heap* h is
  n=*0* : $\mathbb{N}$ // initially *empty* and has the
  invariant $\forall j \in 2..n : h[\lfloor j/2 \rfloor] \leq h[j]$ // *heap property* which implies that
  Function min **assert** $n > 0$ *;* **return** $h[1]$ // the *root* is the *min*imum.

Figure 6.1: Class declaration for a priority queue based on binary heaps whose size is bounded by w.

What does this mean? The key to understanding this definition is a bijection between positive integers and the nodes of a complete binary tree as illustrated in Figure 6.2. In a heap the minimum element is stored in the root (= array position 1). Thus min takes time $O(1)$. Creating an empty heap with space for w elements also takes
⟹ constant time [ps: was $O(n)$???] as it only needs to allocate an array of size w.

Although finding the minimum of a heap is as easy as for a sorted array, the heap property is much less restrictive. For example, there is only one way to sort the set $\{1,2,3\}$ but both $\langle 1,2,3\rangle$ and $\langle 1,3,2\rangle$ a legal representations of $\{1,2,3\}$ by a heap.

Exercise 6.3 Give all legal representations of $\{1,2,3,4\}$ by a heap.

We will now see that this flexibility makes it possible to implement *insert* and *deleteMin* efficiently. We choose a description which is simple and has an easy correctness proof. Section 6.3 gives some hints for a more efficient implementation[ps: alternative:
⟹ give hints in exercises].

An *insert* puts a new element e tentatively at the end of the heap h, i.e., e is put at a
⟹ leaf of the tree represented by h.[reformulated:more rendundancy, less ambiguity] Then e is moved to an appropriate position on the path from the leaf $h[n]$ to the root.

Procedure *insert*(e : *Element*)
  assert $n < w$
  n++ *;* *h[n]*:= e
  siftUp(n)

where *siftUp*(s) moves the contents of node s towards the root until the heap prop-
⟹ erty[was heap condition] holds, cf. Figure 6.2.

Procedure *siftUp*($i : \mathbb{N}$)
  assert the heap property holds except maybe for $j = i$
  if $i = 1 \vee h[\lfloor i/2 \rfloor] \leq h[i]$ **then return**

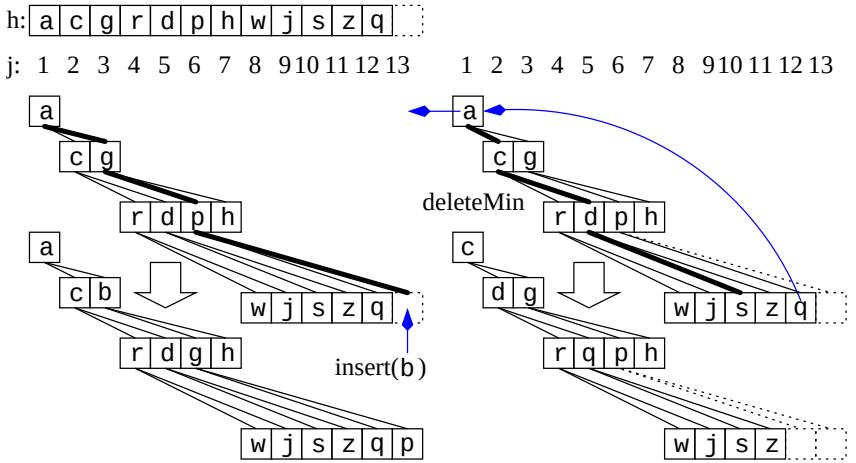


Figure 6.2: The top part shows a heap with $n = 12$ elements stored in an array h with $w = 13$ entries. The root has number one. The children of the root have numbers 2 and 3. The children of node i have numbers $2i$ and $2i+1$ (if they exist). The parent of a node i, $i \geq 2$, has number $\lfloor i/2 \rfloor$. The elements stored in this implicitly defined tree fulfill the invariant that parents are no larger than their children, i.e., the tree is heap-ordered. The left part shows the effect of inserting b. The fat edges mark a path from the rightmost leaf to the root. The new element b is moved up this path until its parent is smaller. The remaining elements on the path are moved down to make room for b. The right part shows the effect of deleting the minimum. The fat edges mark the path p starting at the root and always proceding to the child with smaller *Key*. *Element* q is provisorically moved to the root and then moves down path p until its successors are larger. The remaining elements move up to make room for q
.

  assert the heap property holds except for $j = i$
  swap($h[i], h[\lfloor i/2 \rfloor]$)
  assert the heap property holds except maybe for $j = \lfloor i/2 \rfloor$
  siftUp($\lfloor i/2 \rfloor$)

Correctness follows from the stated invariants.

Exercise 6.4 Show that the running time of *siftUp*(n) is $O(\log n)$ and hence an *insert* takes time $O(\log n)$

A *deleteMin* returns the contents of the root and replaces it by the contents of node n. Since $h[n]$ might be larger than $h[1]$ or $h[2]$, this manipulation may violate the heap property at positions 1 or 2. This possible violation is repaired using *siftDown*.

Function *deleteMin* : *Element*
  assert $n > 0$
  result=$h[1]$: *Element*
  $h[1] := h[n]$
  $n--$
  siftDown(1)
  return *result*

Procedure *siftDown*(1) moves the new contents of the root down the tree until the heap
⟹ property[ps:was condition] holds. More precisely, consider the path p starting at the root and always proceding to a child with minimal[was smaller which is wrong
⟹ for equal keys] *Key*, cf. Figure 6.2. We are allowed to move up elements along path p because the heap property with respect to the other successors (with maximal
⟹ *Key*) will be maintained.[new sentence] The proper place for the root on this path is the highest position where both its successors[was: where is also fulfills the heap property. This is wrong because this would only require it to be larger than
⟹ the parent.] fulfill the heap property. In the code below, we explore the path p by a recursive procedure.

Procedure *siftDown*(i) repairs the heap property at the successors of heap node i without destroying it elsewhere. In particular, if the heap property originally held for the subtrees rooted at $2i$ and $2i+1$, then it now holds for the the subtree rooted at i. (Let us say that the heap property holds at a subtree rooted at node i if it holds for
⟹ all descendants[check whether this is defined in intro] of i but not necessarily for i itself.)[new somewhat longwinded discussion. But some redundance might help and it looks like this is needed to explain what siftDown does in other
⟹ circumstances like *buildHeap*.]

⟹ [changed the precondition and postcondition so that the correctness proof of buildHeap works.]

Procedure *siftDown*$(i : \mathbb{N})$
  assert the heap property holds for the subtrees rooted at $j = 2i$ and $j = 2i+1$
  if $2i \le n$ **then** // i is not a leaf
    if $2i+1 > n \vee h[2i] \le h[2i+1]$ **then** $m := 2i$ **else** $m := 2i+1$
    assert the sibling of m does not exist or does not have a smaller priority than m
    if $h[i] > h[m]$ **then** // the heap property is violated
      swap$(h[i], h[m])$
      siftDown(m)
  assert heap property @ subtree rooted at i

Exercise 6.5 Our current implementation of *siftDown* needs about $2\log n$ element comparisons. Show how to reduce this to $\log n + O(\log\log n)$. Hint: use binary search. Section 6.4 has more on variants of *siftDown*.

We can obviously build a heap from n elements by inserting them one after the other in $O(n\log n)$ total time. Interestingly, we can do better by estabilishing the heap property in a bottom up fashion: *siftDown* allows us to establish the heap property for a subtree of height $k+1$ provided the heap property holds for its subtrees of height k. The following exercise asks you to work out the details of this idea:

Exercise 6.6 (*buildHeap*) Assume that we are given an arbitrary array $h[1..n]$ and want to establish the heap property on it by permuting its entries. We will consider two procedures for achieving this:

Procedure *buildHeapBackwards*
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** *siftDown*(i)

Procedure *buildHeapRecursive*$(i : \mathbb{N})$
  if $4i \le n$ **then**
    buildHeapRecursive$(2i)$
    buildHeapRecursive$(2i+1)$
  siftDown(i)

[smaller itemsep for our enumerations?] ⟸

a) Show that both *buildHeapBackwards* and *buildHeapRecursive*(1) establish the heap property everywhere.

b) Show that both algorithms take total time $O(n)$. Hint: Let $k = \lceil \log n \rceil$. Show that the cost of *siftDown*(i) is $O(k - \log i)$. Sum over i. [should we somewhere have this nice derivation of $\sum_i i2^i$?] ⟸

c) Implement both algorithms efficiently and compare their running time for random integers and $n \in \{10^i : 2 \le i \le 8\}$. It will be important how efficiently you implement *buildHeapRecursive*. In particular, it might make sense to unravel the recursion for small subtrees.

***d)** For large n the main difference between the two algorithms are memory hierarchy effects. Analyze the number of I/O operations needed to implement the two algorithms in the external memory model from the end of Section 2.2. In particular, show that if we have block size B and a fast memory of size $M = \Omega(B\log B)$ then *buildHeapRecursive* needs only $O(n/B)$ I/O operations.

The following theorem summarizes our results on binary heaps.

Theorem 6.1 *[*reformulated theorem such that ALL results on heaps are summarized*] With the heap implementation of bounded non-addressable priority queues, creating an empty heap and finding* min *takes constant time, deleteMin and insert take logarithmic time $O(\log n)$ and build takes linear time.*

Heaps are the basis of *heapsort*. We first *build* a heap from the elements and then repeatedly perform *deleteMin*. Before the i-th *deleteMin* operation the i-th smallest element is stored at the root $h[1]$. We swap $h[1]$ and $h[n+i+1]$ and sift the new root down to its appropriate position. At the end, h stores the elements sorted in decreasing order. Of course, we can also sort in increasing order by using a *max-priority queue*, i.e., a data structure supporting the operations *insert* and deleting the maximum. [moved deleteMin by binary search up to the general deleteMin since I see no reason to use it only for heapsort. Moved the bottom up part to further findings since new experiments indicate that it does not really help if the top-down algorithm is properly implemented.]

Heaps do not immediately implement the ADT addressable priority queue, since elements are moved around in the array h during insertion and deletion. Thus the array indices cannot be used as handles.

Exercise 6.7 (Addressable binary heaps.) Extend heaps to an implementation of addressable priority queues. Compared to nonaddressable heaps your data structure should only consume two additional pointers per element.[new requirement]

Exercise 6.8 (Bulk insertion.) We want to design an algorithm for inserting k new elements into an n element heap.

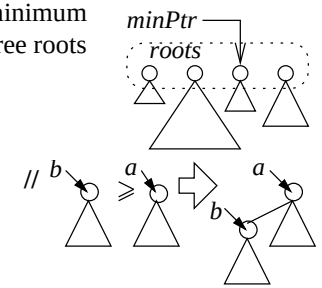
*a) Give an algorithm that runs in time $O\left(k+\log^2 n\right)$. Hint: Use a similar bottom up approach as for heap construction.

**b) Can you even achieve time $O(\min(\log n + k\log k, k+\log(n)\cdot\log\log n))$?

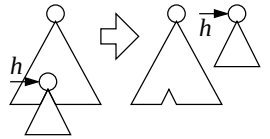
6.2 Addressable Priority Queues

[My current view is that binomial heaps, pairing heaps, fibonacci heaps, thin heaps and perhaps other variants have enough merits to not fix on Fibonacci heaps. Rather I tried to factor out their common properties first and then delve into Fibonacchi heaps as one particular example. Later exercises and Further findings outlines some alternatives.] Binary heaps have a rather rigid structure. All n elements are arranged into a single binary tree of height $\lceil \log n \rceil$. In order to obtain faster implementations of the operations *insert*, *decreaseKey*, *remove*, and *merge* we now look at structures which are more flexible in two aspects: The

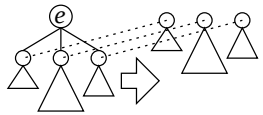
Class *AddressablePQ*
  minPtr : *Handle* // root that stores the minimum
  roots : **set of** *Handle* // pointers to tree roots
  Function min **return** element stored at *minPtr*



  Procedure *link*$(a,b : Handle)$
    remove b from *roots*
    make a the parent of b
  Procedure *combine*$(a,b : Handle)$
    assert *a and b are tree roots*
    if $a \le b$ **then** *link*(a,b) **else** *link*(b,a)
  Procedure *newTree*$(h : Handle)$
    roots:= *roots* $\cup \{i\}$
    if $e <$ min **then** *minPtr*:= i
  Procedure *cut*$(h : Handle)$
    remove the subtree rooted at h from its tree
    newTree(h)



  Function *insert*$(e : Element) : Handle$
    i:= a *Handle* for a new *Item* storing e
    newTree(i)
    return i
  Function *deleteMin* : *Element*
    e:= the *Element* stored in *minPtr*
    foreach *child h of the root at minPtr* **do** *newTree*(h)//
    dispose *minPtr*
    perform some rebalancing // uses *combine*
    return e



  Procedure *decreaseKey*$(h : Handle, k : Key)$
    key(h):= k
    if h is not a root **then**
      cut(h); possibly perform some rebalancing
  Procedure *remove*$(h : Handle)$ *decreaseKey*$(h,-\infty)$; *deleteMin*
  Procedure *merge*$(o : AddressablePQ)$
    if *minPtr* $>$ *o.minPtr* **then** *minPtr*:= *o.minPtr*
    roots:= *roots* $\cup$ *o.roots*
    o.roots:= $\emptyset$; possibly perform some rebalancing

Figure 6.3: Addressable priority queues.

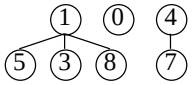


Figure 6.4: A heap ordered forest representing the set $\{0,1,3,4,5,7,8\}$.

single tree is replaced by a collection of trees — a forest. Each tree is still *heap-ordered*, i.e., no child is smaller than its parent. In other words, the sequence of keys along any root to leaf path is non-decreasing. Figure 6.4 shows a heap-ordered forest. But now there is no direct restriction on the height of the trees or the degrees of their nodes. The elements of the queue are stored in *heap items* that have a persistent location in memory. Hence, we can use pointers to address particular elements at any time. Using the terminology from Section 3, *handle*s for priority queue elements are implemented as pointers to heap items. The tree structure is explicitly defined using pointers between items. However we first describe the algorithms on an abstract level independent of the details of representation. In order to keep track of the current minimum, we maintain the handle to the root where it is stored.

Figure 6.3 gives pseudocode that expresses the common aspects of several addressable priority queues. The forest is manipulated using three simple operations: adding a new tree (and keeping *minPtr* up to date), combining two trees into a single one, and cutting out a subtree making it a tree of its own.

An *insert* adds a new single node tree to the forest. So a sequence of n inserts into an initially empty heap will simply create n single node trees. The cost of an insert is clearly $O(1)$.

A *deleteMin* operation removes the node indicated by *minPtr*. This turns all children of the removed node into roots. We then scan the set of roots (old and new) to find the new minimum. To find the new minimum we need to inspect all roots (old and new), a potentially very costly process. We make the process even more expensive (by a constant factor) by doing some useful work on the side, namely combining some trees into larger trees. We will use the concept of amortized analysis to charge the cost depending on the number of trees to the operations that called *newTree*. Hence, to prove a low complexity for *deleteMin*, it suffices to make sure that no tree root has too many children.

We turn to the *decreaseKey* operation next. It is given a handle h and a new key k and decreases the key value of h to k. Of course, k must not be larger than the old key stored with h. Decreasing the information associated with h may destroy the heap property because h may now be smaller than its parent. In order to maintain the heap property, we cut the subtree rooted at h and turn h into a root. Cutting out subtrees causes the more subtle problem that it may leave trees that have an awkward shape.

Therefore, some variants of addressable priority queues perform additional operations to keep the trees in shape.

The remaining operations are easy. We can *remove* an item from the queue by first decreasing its key so that it becomes the minimum item in the queue and then perform a *deleteMin*. To merge a queue o into another queue we compute the union of *roots* and o.*roots*. To update *minPtr* it suffices to compare the minima of the merged queues. If the root sets are represented by linked lists, and no additional balancing is done, a merge needs only constant time.

We will now look at several particular implementations of addressable priority queues:

6.2.1 Pairing Heaps

Pairing Heaps [36] are a minimalistic implementation of the above family of addressable priority queues. It seems that pairing heaps are what one should do in practice yet a tight theoretical analysis is still open[check a bit]. When Figure 6.3 says "possibly ⟸ do some rebalancing", pairing heaps do nothing. If $\langle r_1,\ldots,r_k\rangle$ is the sequence of root nodes stored in *roots* then *deleteMin combine*s r_1 with r_2, r_3 with r_4, etc., i.e., the *roots* are *paired*. Figure 6.5 gives an example.

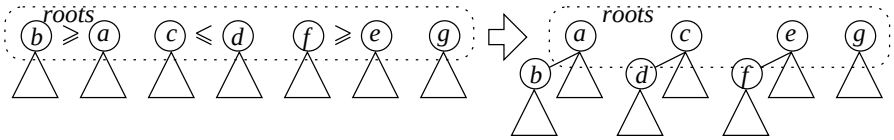


Figure 6.5: The *deleteMin* operation of pairing heaps combines pairs of root nodes.

***Exercise 6.9 (Three pointer items.)** Explain how to implement pairing heaps using three pointers per heap item: One to the youngest child, one to the next older sibling (if any), and one that either goes to the next younger sibling, or, if there is no younger sibling, to the parent. Figure 6.8 gives an example.

***Exercise 6.10 (Two pointer items.)** The three pointer representation from Exercise 6.9 can be viewed as a binary tree with the invariant that the items stored in left subtrees contain no smaller elements. Here is a more compact representation of this binary tree: Each item stores a pointer to its right child. In addition, a right child stores a pointer to its sibling. Left childs or right childs without sibling store a pointer to their parent. A right child without a sibling stores Explain how to implement pairing heaps using this representation. Figure 6.8 gives an example.

6.2.2 Fibonacchi Heaps

Fibonacchi Heaps use more aggressive balancing operations than pairing heaps that make it possible to prove better performance guarantees. In particular, we will get logarithmic amortized time for *remove* and *deleteMin* and worst case constant time for all other operations.

Each item of a Fibonacchi heap stores four pointers that identify its parent, one child, and two siblings (cf. Figure 6.8). The children of each node form a doubly-linked circular list using the sibling pointers. The sibling pointers of the root nodes can be used to represent *roots* in a similar way. Parent pointers of roots and child pointers of leaf nodes have a special value, e.g., a null pointer.

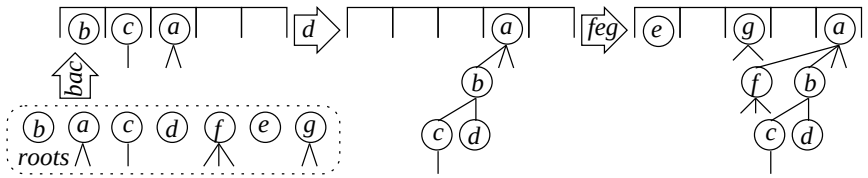


Figure 6.6: An example for the development of the bucket array while the *deleteMin* of Fibonacchi heaps is combining the roots. The arrows indicate which roots have been scanned. Node that scanning *d* leads to a cascade of three combination operations.

In addition, every heap item contains a field *rank*. The *rank* of an item is the number of its children. In Fibonacchi heaps, *deleteMin* links roots of equal rank r. The surviving root will then get rank $r+1$. An efficient method to combine trees of equal rank is as follows. Let *maxRank* be the maximal rank of a node *after* the execution of *deleteMin*. Maintain a set of buckets, initially empty and numbered from 0 to *maxRank*. Then scan the list of old and new roots. When a root of rank i is considered inspect the i-th bucket. If the i-th bucket is empty then put the root there. If the bucket is non-empty then combine the two trees into one. This empties the i-th bucket and creates a root of rank $i+1$. Try to throw the new tree into the $i+1$st bucket. If it is occupied, combine .... When all roots have been processed in this way, we have a collection of trees whose roots have pairwise distinct ranks. Figure 6.6 gives an example.

A *deleteMin* can be very expensive if there are many roots. However, we now show that in an amortized sense, *maxRank* is more important.

Lemma 6.2 *By charging a constant amount of additional work to every newTree operation, the amortized complexity of deleteMin can be made* $O(\mathit{maxRank})$.

Proof: A *deleteMin* first calls *newTree* at most *maxRank* times and then initializes an array of size *maxRank*. The remaining time is proportional to the number of *combine* operations performed. From now on we therefore only count *combine*s. To make things more concrete lets say that one *peanut* pays for a *combine*.[intro here or in search tree?] We make *newTree* pay one peanut stored with the new tree root. With these conventions in place, a combine operation performed by *deleteMin* is free since the peanut stored with the item that ceases to be a root, can pay for the *combine*. ∎

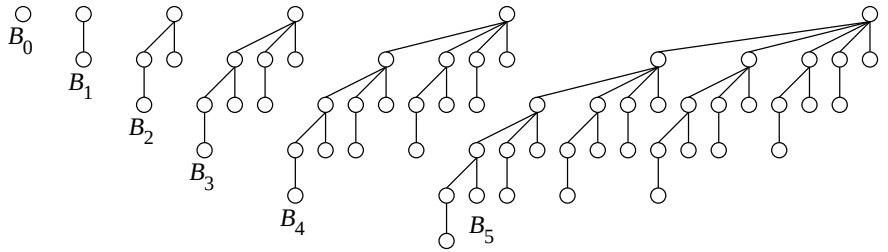


Figure 6.7: The binomial trees of rank zero to five.

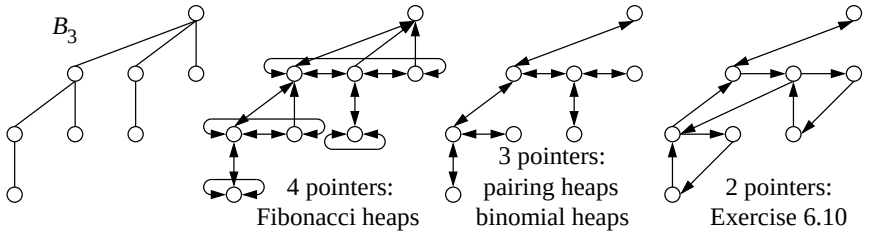


Figure 6.8: Three ways to represent trees of nonuniform degree. The binomal tree of rank three, B_3, is used as an example.

Lemma 6.2 tells us that in order to make *deleteMin* fast we should make sure that *maxRank* remains small. Let us consider a very simple situation first. Suppose that we perform a sequence of inserts followed by a single *deleteMin*. In this situation, we start with a certain number of single node trees and all trees formed by combining are so-called *binomial trees* as shown in Figure 6.7. The binomial tree B_0 consists of a single node and the binomial tree B_{i+1} is obtained by joining two copies of the tree B_i. This implies that the root of the tree B_i has rank i and that the tree B_i contains

exactly 2^i nodes. We conclude that the the rank of a binomial tree is logarithmic in the size of the tree. If we could guarantee in general that the maximal rank of any node is logarithmic in the size of the tree then the amortized cost of the *deleteMin* operation would be logarithmic.

Unfortunately, *decreaseKey* may destroy the nice structure of binomial trees. Suppose item v is cut out. We now have to decrease the rank of its parent w. The problem is that the size of the subtrees rooted at the ancestors of w has decreased but their rank has not changed. Therefore, we have to perform some balancing to keep the trees in shape.

An old solution suggested by Vuillemin [94] is to keep all trees in the heap binomial. However, this causes logarithmic cost for a *decreaseKey*.

***Exercise 6.11 (Binomial heaps.)** Work out the details of this idea. Hint: you have to cut v's ancestors and their younger siblings.

Fredman and Tarjan showed how to decrease its cost to $O(1)$ without increasing the cost of the other operations. Their solution is surprisingly simple and we describe it next.

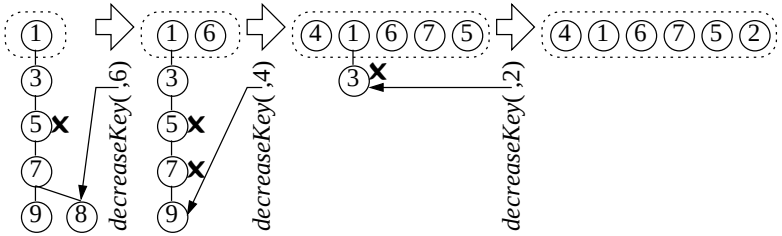


Figure 6.9: An example for cascading cuts. Marks are drawn as crosses. Note that roots are never marked.

When a non-root item x loses a child because *decreaseKey* is applied to the child, x is marked; this assumes that x has not already been marked. When a marked node x loses a child, we cut x, remove the mark from x, and attempt to mark x's parent. If x's parent is marked already then .... This technique is called *cascading cuts*. In other words, suppose that we apply *decreaseKey* to an item v and that the k-nearest ancestors of v are marked, then turn v and the k-nearest ancestors of v into roots and mark the $k+1$st-nearest ancestor of v (if it is not a root). Also unmark all the nodes that were turned into roots. Figure 6.9 gives an example.

Lemma 6.3 *The amortized complexity of decreaseKey is constant.*

Proof: We generalize the proof of Lemma 6.2 to take the cost of *decreaseKey* operations into account. These costs are proportional to the number of *cut* operations performed. Since *cut* calls *newTree* which is in turn charged for the cost of a *combine*, we can as well ignore all costs except the peanuts needed to pay for *combine*s.[alternative: also account cuts then marked items store two peanuts etc.] We assign one peanut to every marked item. We charge two peanuts for a ⟸ *decreaseKey* — one pays for the *cut* and the other for marking an ancestor that has not been marked before. The peanuts stored with unmarked ancestors pay for the additional *cut*s. ∎

How do cascading cuts affect the maximal rank? We show that it stays logarithmic. In order to do so we need some notation. Let $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$ be the sequence of Fibonacci numbers. It is well-known that $F_{i+1} \geq (1+\sqrt{5}/2)^i \geq 1.618^i$ for all $i \geq 0$.

Lemma 6.4 *Let v be any item in a Fibonacci heap and let i be the rank of v. Then the subtree rooted at v contains at least F_{i+2} nodes. In a Fibonacci heap with n items all ranks are bounded by* $1.4404 \log n$.

Proof: [start counting from zero here?] Consider an arbitrary item v of rank i. ⟸ Order the children of v by the time at which they were made children of v. Let w_j be the j-th child, $1 \leq j \leq i$. When w_j was made child of v, both nodes had the same rank. Also, since at least the nodes $w_1, \ldots, w_{j-1}$ were nodes of v at that time, the rank of v was at least $j-1$ at the time when w_j was made a child of v. The rank of w_j has decreased by at most 1 since then because otherwise w_j would be a root. Thus the current rank of w_j is at least $j-2$.

We can now set up a recurrence for the minimal number S_i of nodes in a tree whose root has rank i. Clearly $S_0 = 1$, $S_1 = 2$, and $S_i \geq 2 + S_0 + S_1 + \cdots + S_{i-2}$. The last inequality follows from the fact that for $j \geq 2$, the number of nodes in the subtree with root w_j is at least S_{j-2}, and that we can also count the nodes v and w_1. The recurrence above (with = instead of $\geq$) generates the sequence 1, 2, 3, 5, 8,... which is identical to the Fibonacci sequence (minus its first two elements).

Let's verify this by induction. Let $T_0 = 1$, $T_1 = 2$, and $T_i = 2 + T_0 + \cdots + T_{i-2}$ for $i \geq 2$. Then, for $i \geq 2$, $T_{i+1} - T_i = 2 + T_0 + \cdots + T_{i-1} - 2 - T_0 - \cdots - T_{i-2} = T_{i-1}$, i.e., $T_{i+1} = T_i + T_{i-1}$. This proves $T_i = F_{i+2}$.

For the second claim, we only have to observe that $F_{i+2} \leq n$ implies $i \cdot \log(1+\sqrt{5}/2) \leq \log n$ which in turn implies $i \leq 1.4404 \log n$. ∎

This concludes our treatment of Fibonacci heaps. We have shown.

Theorem 6.5 *The following time bounds hold for Fibonacci heaps:* min, *insert, and merge take worst case constant time; decreaseKey takes amortized constant time and remove and deleteMin take amortized time logarithmic in the size of the queue.*

6.3 Implementation Notes

There are various places where *sentinels* (cf. Chapter 3) can be used to simplify or (slightly) accelerate the implementation of priority queues. Since this may require additional knowledge about key values this could make a reusable implementation more difficult however.

- If $h[0]$ stores a *Key* no larger than any *Key* ever inserted into a binary heap then *siftUp* need not treat the case $i = 1$ in a special way.
- If $h[n+1]$ stores a *Key* no smaller than any *Key* ever inserted into a binary heap then *siftDown* need not treat the case $2i+1 > n$ in a special way. If such large keys are even stored in $h[n+1..2n+1]$ then the case $2i > n$ can also be eliminated.
- Addressable priority queues can use a special *NADA* item rather than a null pointer.

For simplicity we have formulated the operations *siftDown* and *siftUp* of binary heaps using recursion. It might be a bit faster to implement them iteratively instead.

Exercise 6.12 Do that.

However, good compilers might be able to do this *recursion elimination* automatically.

As for sequences, memory management for items of addressable priority queues can be critical for performance. Often, a particular application may be able to do that more efficiently than a general purpose library. For example, many graph algorithms use a priority queue of nodes. In this case, the item can be stored with the node. [todo ⟹ cross references to sssp and jarnikprimmst]

There are priority queues that work efficiently for integer keys. It should be noted that these queues can also be used for floating point numbers. Indeed, the IEEE floating point standard has the interesting property that for any valid floating point numbers a and b, $a \leq b$ if an only $bits(a) \leq bits(b)$ where $bits(x)$ denotes the reinterpretation of x as an unsigned integer.

C++

The STL class *priority_queue* offers nonaddressable priority queues implemented using binary heaps. LEDA implements a wide variety of addressable priority queues including pairing heaps and Fibonacchi heaps.

Java

The class *java.util.PriorityQueue* supports addressable priority queues to the extend that *remove* is implemented. However *decreaseKey* and *merge* are not supported. Also it seems that the current implementation of *remove* needs time $\Theta(n)$! JDSL offers an addressable priority queue *jdsl.core.api.PriorityQueue* which is currently implemented as a binary heap.[wo erklren wir was jdsl ist? in implmentation notes of intro?] ⟸

6.4 Further Findings

There is an interesting internet survey[1] on priority queues. It seems that the most important applications are (shortest) path planning (cf. Section **??**), discrete event simulation, coding and compression (probably variants of *Huffman coding*[irgendwas zitieren?]), scheduling in operating systems, computing maximum flows (probably ⟸ the highest label preflow push algorithm [68, Chapter 7.10] [AhuMagOrl auch zitieren?]), and branch-and-bound (cf. Section 12.4.1). About half of the respondents ⟸ used binary heaps.

In Section 6.1 we have seen an implementation of *deleteMin* by top-down search that needs about $2\log n$ element comparisons and a variant using binary search that needs only $\log n + o(\log\log n)$ element comparisons. The latter is of mostly of theoretical interest. Interestingly a very simple algorithm that first sifts the elment down all the way to the bottom of the heap and than sifts it up again can be even better. When used for sorting, the resulting *Bottom-up heapsort* requires $\frac{3}{2}n\log n + o(n)$ comparisons in the worst case and $n\log n + o(1)$ in the average case [96, 33, 83]. [are there more general average case results for bottom-up deleteMin or insertion?] While bottom-up heapsort is simple and practical, our own experiments ⟸ indicate that it is not faster than the usual top-down variant (for integer keys). This was unexpected for us. The explanation seems to be that the outcome of the comparisons saved by the bottom-up variant are easy to predict. Modern hardware executes such predictable comparisons very efficiently (see [81] for more discussion).

The recursive *buildHeap* routine from Exercise 6.6 is an example for a *cache oblivous algorithm* [37] [already introduce model further findings of intro or

[1] `http://www.leekillough.com/heaps/survey_results.html`

⟹ sort?] — the algorithm is efficient in the external memory model even though it does not explicitly use the block size or cache size.

Pairing heaps have amortized constant complexity for *insert* and *merge* [44] and logarithmic amortized complexity for *deleteMin*. There is also a logarithmic upper bound for *decreaseKey* but it is not known whether this bound is tight. Fredman [35] has given operation sequences consisting of $O(n)$ insertions and *deleteMin*s and $O(n \log n)$ *decreaseKey*s that take time $\Omega(n \log n \log\log n)$ for a family of addressable priority queues that includes all previously proposed variants of pairing heaps.

The family of addressable priority queues from Section 6.2 has many additional interesting members. Høyer describes additional balancing operations which can be used together with *decreaseKey* that look similar to operations more well known for search trees. One such operation yields *thin heaps* [50] which have similar performance guarantees than Fibonacchi heaps but do not need a parent pointer or a mark bit. It is likely that thin heaps are faster in practice than Fibonacchi heaps. There are also priority queues with worst case bounds asymptotically as good as the amortized bounds we have seen for Fibonacchi heaps [17]. The basic idea is to tolerate violations of the heap property and to continously invest some work reducing the violations. The *fat heap* [50] seem to be simple enough to be implementable at the cost of allowing *merge* only in logarithmic time.

Many applications only need priority queues for integer keys. For this special case there are more efficient priority queues. The best theoretical bounds so far are constant time *decreaseKey* and *insert* and $O(\log\log n)$ time for *deleteMin* [92, 71]. Using randomization the time bound can even be reduced to $O\left(\sqrt{\log\log n}\right)$ [97]. These algorithms are fairly complex but by additionally exploiting mononotonicity of the queues one gets simple and practical priority queues. Section 10.5 will give examples.[move ⟹ here?] The *calendar queues* [19] popular in the discrete event simulation community are even simpler variants of these integer priority queues. A practical monotone priority queue for integers is described in [6].

Chapter 7

Sorted Sequences

Searching a name in a telephone book is easy — if the person you look for has the telephone number for long enough. It would be nicer to have a telephone book that is updated immediately when something changes. The "manual data structure" used for this purpose are file card boxes. Some libraries still have huge collections with hundreds of thousands of cards collecting dust.

[erstmal binary search proper wahrscheinlich in intro] ⟸

People have looked for a computerized version of file cards from the first days of commercial computer use. Interestingly, there is a quite direct computer implementation of file cards that is not widely known. The starting point is our "telephone book" data structure from Chapter 5 — a sorted array $a[1..n]$. We can search very efficiently by binary search. To remove an element, we simply mark it as removed. Note that we should *not* overwrite the key because it is still needed to guide the search for other elements. The hard part is insertion. If the new element e belongs between $a[i-1]$ and $a[i]$ we somehow have to squeeze it in. In a file card box we make room by pushing other cards away. Suppose $a[j+1]$ is a free entry of the array to the right of $a[i]$. Perhaps $a[j+1]$ previously stored a removed element or it is some free space at the end of the array. We can make room for element e in $a[i]$ by moving the elements in $a[i..j]$ one position to the right. Figure 7.1 gives an example. Unfortunately, this shifting can be very time consuming. For example, if we insert n elements with decreasing

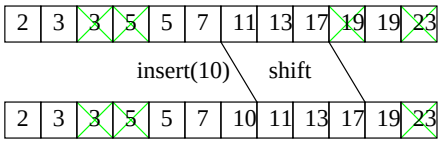


Figure 7.1: A representation of the sorted sequence $\langle 2,3,5,7,11,13,17,19\rangle$ using a sparse table. Situation before and after a 10 is inserted.

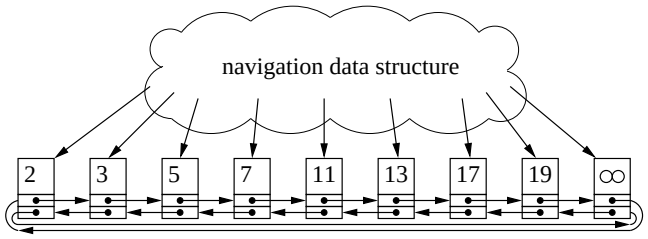


Figure 7.2: A sorted sequence as a doubly linked list plus a navigation data structure.

key value, we always have to shift all the other elements so that a total of $n(n-1)/2$ elements are moved. But why does the shifting technique work well with file cards? The trick is that one leaves some empty space well dispersed over the entire file. In the array implementation, we can achieve that by leaving array cells empty. Binary search will still work as long as we make sure that empty cells carry meaningful keys, e.g., the keys of the next nonempty cell. Itai et al. [45] have developed this idea of *sparse tables* into a complete data structure including rebalancing operations when things get too tight somewhere. One gets constant amortized insertion time "on the average", i.e., if insertions happen everywhere with equal probability. In the worst case, the best known strategies give $O\left(\log^2 n\right)$ amortized insertion time into a data structure with n elements. This is much slower than searching and this is the reason why we will follow another approach for now. But look at Exercise 7.10 for a refinement of sparse tables.

Formally, we want to maintain a *sorted sequence*, i.e., a sequence of *Element*s sorted by their *Key* value. The three basic operations we want to support are

Function *locate*$(k : Key)$ **return** $\min\{e \in M : e \geq k\}$
Procedure *insert*$(e : Element)$ $M := M \cup \{e\}$
Procedure *remove*$(k : Key)$ $M := M \setminus \{e \in M : key(e) = k\}$

where M is the set of elements stored in the sequence. It will turn out that these basic operations can be implemented to run in time $O(\log n)$. Throughout this section, n denotes the size of the sequence. It is instructive to compare this operation set with previous data structures we have seen. Sorted sequences are more flexible than sorted arrays because they efficiently support *insert* and *remove*. Sorted sequences are slower but also more powerful than hash tables since *locate* also works when there is no element with key k in M. Priority queues are a special case of sorted sequences because they can only locate and remove the smallest element.

Most operations we know from doubly linked lists (cf. Section 3.2.1) can also be

implemented efficiently for sorted sequences. Indeed, our basic implementation will represent sorted sequences as a sorted doubly linked list with an additional structure supporting *locate*. Figure 7.2 illustrates this approach. Even the dummy header element we used for doubly linked lists has a useful role here: We define the result of locate(k) as the handle of the smallest list item $e \geq k$ or the dummy item if k is larger than any element of the list, i.e., the dummy item is treated as if it stores an element with infinite key value. With the linked list representation we "inherit" constant time implementations for *first*, *last*, *succ*, and *pred*. We will see constant amortized time implementations for *remove*$(h : Handle)$, *insertBefore*, and *insertAfter* and logarithmic time algorithms for concatenating and splitting sorted sequences. The indexing operator $[\cdot]$ and finding the position of an element in the sequence also takes logarithmic time.

Before we delve into implementation details, let us look at a few concrete applications where sorted sequences are important because all three basic operations *locate*, *insert*, and *remove* are needed frequently.

Best First Heuristics: [useful connection to optimization chapter?] Assume we want to pack items into a set of bins. The items arrive one at a time and have to be put into a bin immediately. Each item i has a weight $w(i)$ and each bin has a maximum capacity. A successful heuristic solutions to this problem is to put item i into the bin that fits best, i.e., whose remaining capacity is smallest among all bins with residual capacity at least as large as $w(i)$ [23]. To implement this algorithm, we can keep the the bins in a sequence s sorted by their residual capacity. To place an item, we call $s.locate(w(i))$, remove the bin we found, reduce its residual capacity by $w(i)$, and reinsert it into s. See also Exercise 12.8. An analogous approach is used for scheduling jobs to machines [**?**] or in memory management. ⟸

Sweep-Line Algorithms: [todo: einheitlich myparagraph mit kleinerem Abstand] Assume you hava a set of horizontal and vertical line segments in the plane and want to find all points where two segments intersect. A sweep line algorithm moves a vertical line over the plane from left to right and maintains the set of horizontal lines that intersect the sweep line in sorted sequence s. When the left endpoint of a horizontal segment is reached, it is inserted into s and when its right endpoint is reached, it is removed from s. When a vertical line segment is reached at position x that spans the vertical range $[y, y']$, we call $s.locate(y)$ and scan s until we reach key y'. All the horizontal line segments discovered during this scan define an intersection. The sweeping algorithm can be generalized for arbitrary line segments [11], curved objects, and many other geometric problems[cite sth?]. ⟸ ⟸

Data Base Indexes: A variant of the (a,b)-tree data structure explained in Section 7.2 is perhaps the most important data structure used in data bases.

The most popular way to accelerate *locate* is using a *search tree*. We introduce search tree algorithms in three steps. As a warm-up, Section 7.1 introduces a simple variant of *binary search trees* that allow *locate* in $O(\log n)$ time under certain circumstances. Since binary search trees are somewhat difficult to maintain under insertions and removals, we switch to a generalization, (a,b)-trees that allows search tree nodes of larger degree. Section 7.2 explains how (a,b)-trees can be used to implement all three basic operations in logarthmic worst case time. Search trees can be augmented with additional mechanisms that support further operations using techniques introduced in Section 7.3.

7.1 Binary Search Trees

$\Longrightarrow$ [already in intro?]

Navigating a search tree is a bit like asking your way around a foreign city. At every street corner you show the address to somebody. But since you do not speak the language, she can only point in the right direction. You follow that direction and at the next corner you have to ask again.

More concretely, a *search tree* is a tree whose leaves store the elements of the sorted sequence.[1] To speed up locating a key k, we start at the root of the tree and traverse the unique path to the leaf we are looking for. It remains to explain how we find the correct path. To this end, the nonleaf nodes of a search tree store keys that guide the search. In a *binary* search tree that has $n \geq 2$ leaves, a nonleaf node has exactly two children — a *left* child and a *right* child. Search is guided by one *splitter* key s for each nonleaf node. The elements reachable through the left subtree have keys $k \leq s$. All the elements that are larger than s are reachable via the right subtree. With these definitions in place, it is clear how to ask for the way. Go left if $k \leq s$. Otherwise go right. Figure 7.5 gives an example. The length of a path from the root the a node of the tree is called the depth of this node. The maximum depth of a leaf is the *height* of the tree. The height therefore gives us the maximum number of search steps needed to *locate* a leaf.

Exercise 7.1 Prove that a binary search tree with $n \geq 2$ leaves can be arranged such that it has height $\lceil \log n \rceil$.

A search tree with height $\lceil \log n \rceil$ is *perfectly balanced*. Compared to the $\Omega(n)$ time needed for scanning a list, this is a dramatic improvement. The bad news is that it is

[1] There is also a variant of search trees where the elements are stored in all nodes of the tree.

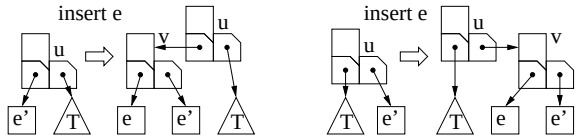


Figure 7.3: Naive insertion into a binary search tree. A triangles indicates an entire subtree.

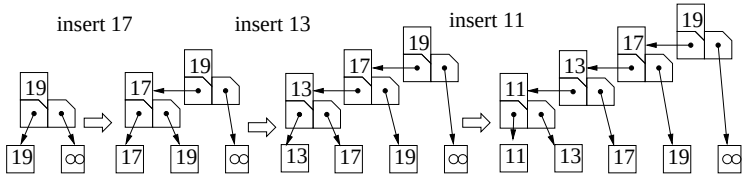


Figure 7.4: Naively inserting sorted elements leads to a degenerate tree.

expensive to keep perfect balance when elements are inserted and removed. To understand this better, let us consider the "naive" insertion routine depicted in Figure 7.4. We locate the key k of the new element e before its successor e', insert e into the list, and then introduce a new node v with left child e and right child e'. The old parent u of e' now points to v. In the worst case, every insertion operation will locate a leaf at maximum depth so that the height of the tree increases every time. Figure 7.4 gives an example that shows that in the worst case the tree may degenerate to a list — we are back to scanning.

An easy solution of this problem is a healthy portion of optimism — perhaps it will not come to the worst. Indeed, if we insert n elements in *random* order, the expected height of the search tree is $\approx 2.99 \log n$ [29]. [Proof of average height of a key? ($\approx \log_\Phi n$)] A more robust solution, that still can be implemented to run in $O(\log n)$ $\Longleftarrow$ time is to actively rebalance the tree so that the height always remains logarithmic. Rebalancing is achieved by applying the *rotation* transformation depicted in Figure 7.5 in a clever way. For more details on rebalancing techniques we refer to the literature surveyed in Section 7.6. Instead, in the following section we will generalize from binary search trees to search trees where the nodes have variable degree.

Exercise 7.2 Explain how to implement an *implicit* binary search tree, i.e., the tree is stored in an array using the same mapping of tree structure to array positions as in the binary heaps discussed in Section 6.1. What are advantages and disadvantages

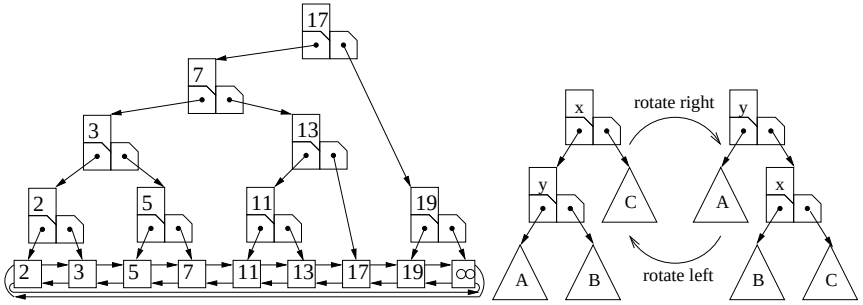


Figure 7.5: Left: Sequence $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a binary search tree. Right: Rotation of a binary search tree.

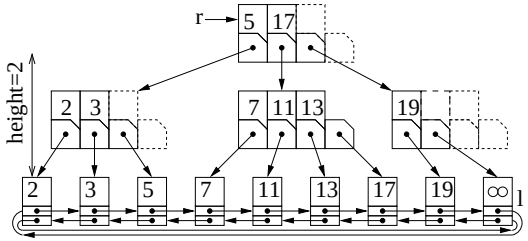


Figure 7.6: Sequence $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a $(2,4)$-tree.

compared to a pointer based implementation? Compare search in an implicit binary tree to binary search in a sorted array. We claim that the implicit tree might be slightly faster. Why? Try it.

7.2 Implementation by (a,b)-Trees

An (a,b)-tree is a search tree where interior nodes have degree[outdegree? oder degree von rooted trees geeignet definieren?] d between a and b. Let $c[1..d]$ denote an array of pointers to children. The root has degree one for a trivial tree with a single leaf. Otherwise the root has degree between 2 and b. We will see that see for $a \geq 2$ and $b \geq 2a-1$, the flexibility in node degrees allows us to efficiently maintain the invariant that *all leaves have the same depth*. Unless otherwise stated, we treat a and b as constants to simplify asymptotic notation, i.e., $a = O(b) = O(1)$. Search is

guided by a sorted array $s[1..d-1]$ of $d-1$ splitter keys. To simplify notation, we additionally define $s[0] = -\infty$ and $s[d] = \infty$. Elements e reachable through child $c[i]$ have keys $s[i-1] < key(e) \leq s[i]$ for $1 \leq i \leq d$. Figure 7.6 gives a $(2,4)$-tree storing the sequence $\langle 2,3,5,7,11,13,17,19\rangle$.

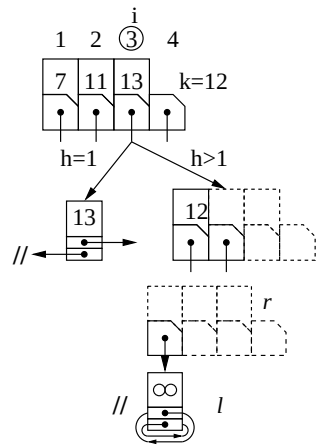
Exercise 7.3 Prove that an (a,b)-tree with $n \geq 2$ leaves has height at most $\lfloor \log_a n/2 \rfloor + 1$. Also prove that this bound is tight: For every height h give an (a,b)-tree with height $h = 1 + \log_a n/2$.

Searching an (a,b)-tree is only slightly more complicated than searching a binary tree. Instead of performing a single comparison at a nonleaf node, we have to find the correct child among up to b choices. If the keys in the node are stored in sorted order, we can do that using binary search, i.e., we need at most $\lceil \log b \rceil$ comparisons for each node we are visiting.

Class *ABHandle* : **Pointer** *to ABItem or Item*
Class *ABItem(splitters* : *Sequence* **of** *Key, children* : *Sequence* **of** *ABHandle)*
  d=|*children*| : 1..b // degree
  s=*splitters* : **Array** $[1..b-1]$ **of** *Key*
  c=*children* : **Array** $[1..b]$ **of** *ABItem*

  Function *locateLocally*(*k* : *Key*) : $\mathbb{N}$
    return $\min\{i \in 1..d : k \leq s[i]\}$

  Function *locateRec*(*k* : *Key, h* : $\mathbb{N}$) : *Handle*
    i:= *locateLocally(k)*
    if $h = 1$ **then return** $c[i]$
    else return $c[i]$→*locateRec(k, h* − 1*)*

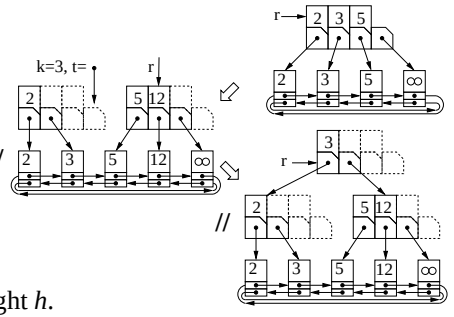


Class *ABTree(a* ≥ 1 : $\mathbb{N}$,*b* ≥ 2*a* − 1 : $\mathbb{N}$*)* **of** *Element*
  ℓ=⟨⟩ : *List* **of** *Element*
  r : *ABItem(*⟨⟩, ⟨ℓ.*head*()⟩*)*
  height=1 : $\mathbb{N}$

  // Locate the smallest Item with key $k' \geq k$
  Function *locate*(*k* : *Key*) : *Handle* **return** *r.locateRec*(*k*, *height*)

Figure 7.7: (a,b)-trees.

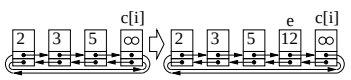
[where to explain "::" notation? Just here? in intro? in appendix?]

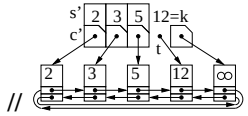


```
// Example:
// ⟨2,3,5⟩.insert(12)
Procedure ABTree::insert(e : Element)
    (k,t):= r.insertRec(e,height,ℓ)
    if t ≠ null then          // root was split
        r:= new ABItem(⟨k⟩, ⟨r,t⟩)
        height++
```



```
// Insert a new element into a subtree of height h.
// If this splits the root of the subtree, return the new splitter and subtree handle
Function ABItem::insertRec(e : Element, h : ℕ, ℓ : List of Element) : Key×ABHandle
    i:= locateLocally(e)
    if h = 1 then                                              // base case
        if key(c[i]→e) = key(e) then c[i]→e:= e;   return (⊥,null )    // just update
        else
            (k,t):= (key(e),ℓ.insertBefore(e,c[i]))
    else
        (k,t):= c[i]→insertRec(e,h−1,ℓ)
        if t = null then return (⊥,null )
    s′:= ⟨s[1],...,s[i−1],k,s[i],...,s[d−1]⟩
    c′:= ⟨c[1],...,c[i−1],t,c[i],...,c[d]⟩
    if d < b then          // there is still room here
        (s,c,d):= (s′,c′,d+1)
        return (⊥,null )
    else                                  // split this node
        d:= ⌊(b+1)/2⌋
        s:= s′[b+2−d..b]
        c:= c′[b+2−d..b+1]
        return (s′[b+1−d], newABItem(s′[1..b−d], c′[1..b+1−d]))
```





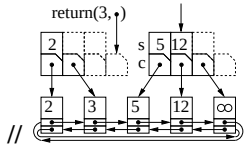


Figure 7.8: Insertion into (a,b)-trees.

[consistently replace .. with a range macro? In particular, put some white space around it?] Figure 7.7 gives pseudocode for the (a,b)-trees and the *locate* operation. Recall that we use the search tree as a way to locate items of a doubly linked list and that the dummy list item is considered to have key value ∞. This dummy item is the rightmost leaf in the search tree. Hence, there is no need to treat the special case of root degree 0 and a handle of the dummy item can serve as a return value when locating a key larger than all values in the sequence.

To *insert* an element e, the routine in Figure 7.8 first descends the tree recursively to find the smallest sequence element e' that is not smaller than e. If e and e' have equal keys, e replaces e'. Otherwise, e is inserted into the sorted list ℓ before e'. If e' was the i-th child $c[i]$ of its parent node v then e will become the new $c[i]$ and $key(e)$ becomes the corresponding splitter element $s[i]$. The old children $c[i..d]$ and their corresponding splitters $s[i..d-1]$ are shifted one position to the right.

The difficult part is what to do when a node v already had degree $d=b$ and now would get degree $b+1$. Let s' denote the splitters of this illegal node and c' its children. The solution is to *split* v in the middle. Let $d=\lceil (b+1)/2 \rceil$ denote the new degree of v. A new node t is allocated that accommodates the $b+1-d$ leftmost child pointers $c'[1..b+1-d]$ and the corresponding keys $s'[1..b-d]$. The old node v keeps the d rightmost child pointers $c'[b+2-d..b+1]$ and the corresponding splitters $s'[b+2-d..b]$.

The "leftover" middle key $k=s'[b+1-d]$ is an upper bound for the keys reachable from t now. Key k and the pointer to t is needed in the predecessor u of v. The situation in u is analogous to the situation in v before the insertion: if v was the ith child of u, t is displacing it to the right. Now t becomes the ith child and k is inserted as the i-th splitter. This may cause a split again etc. until some ancestor of v has room to accommodate the new child or until the root is split.

In the latter case, we allocate a new root node pointing to the two fragments of the old root. This is the only situation where the height of the tree can increase. In this case, the depth of all leaves increases by one, i.e., we maintain the invariant that all leaves have the same depth. Since the height of the tree is $O(\log n)$ (cf. Exercise 7.3), we get a worst case execution time of $O(\log n)$ for *insert*.

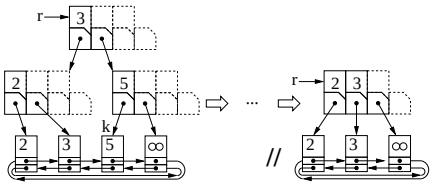
Exercise 7.4 It is tempting to streamline *insert* by calling *locate* to replace the initial descent of the tree. Why does this *not* work with our representation of the data structure?[2]

Exercise 7.5 Prove that for $a \geq 2$ and $b \geq 2a-1$ the nodes v and t resulting from splitting a node of degree $b+1$ have degree between a and b.

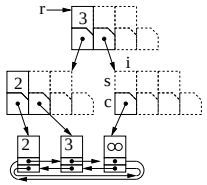
[consistently *concat*, *concatenate* → ∘?] The basic idea behind *remove* is similar ⟸ to insertion — it locates the element to be removed, removes it from the sorted list, and on the way back up repairs possible violations of invariants. Figure 7.9 gives

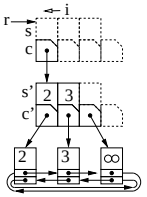
[2] Note that this approach becomes the method of choice when the more elaborate representation from Section 7.4.1 is used.

```
// Example: ⟨2,3,5⟩.remove(5)
Procedure ABTree::remove(k : Key)
    r.removeRec(k, height, ℓ)
    if r.d = 1 ∧ height > 1 then
        r' := r; r := r'.c[1]; dispose r'
```




```
Procedure ABItem::removeRec(k : Key, h : ℕ, ℓ : List of Element)
    i := locateLocally(k)
    if h = 1 then                                   // base case
        if key(c[i]→e) = k then // there is sth to remove
            ℓ.remove(c[i])
            removeLocally(i)
    else
        c[i]→removeRec(e, h−1, ℓ)
        if c[i]→d < a then                          // invariant needs repair
            if i = d then i−−                       // make sure i and i+1 are valid neighbors
            s' := concatenate(c[i]→s, ⟨s[i]⟩, c[i+1]→s))
            c' := concatenate(c[i]→c, c[i+1]→c)
            d' := |c'|
            if d' ≤ b then                          // fuse
                (c[i+1]→s, c[i+1]→c, c[i+1]→d) := (s', c', d')
                dispose c[i];   removeLocally(i)
            else                                    // balance
                m := ⌈d'/2⌉
                (c[i]→s, c[i]→c, c[i]→d) := (s'[1..m−1], c'[1..m], m)
                (c[i+1]→s,        c[i+1]→c, c[i+1]→d) :=
                (s'[m+1..d'−1], c'[m+1..d'], d'−m)
                s[i] := s'[m]
```






```
// Remove the i-th child from an ABItem
Procedure ABItem::removeLocally(i : ℕ)
    c[i..d−1] := c[i+1..d]
    s[i..d−2] := s[i+1..d−1]
    d−−
```

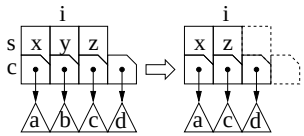


Figure 7.9: Removal from an (a,b)-tree.

pseudocode. When a parent u notices that the degree of its child $c[i]$ has dropped to $a-1$, it combines this child with one of its neighbors $c[i-1]$ or $c[i+1]$ to repair

the invariant. There are two cases. If the neighbor has degree larger than a, we can *balance* the degrees by transferring some nodes from the neighbor. If the neighbor has degree a, balancing cannot help since both nodes together have only $2a-1$ children so that we cannot give a children to both of them. However, in this case we can *fuse* them to a single node since the requirement $b \geq 2a-1$ ensures that the fused node has degree at most b.

To fuse a node $c[i]$ with its right neighbor $c[i+1]$ we concatenate their children arrays. To obtain the corresponding splitters, we need to place the splitter $s[i]$ from the parent between the splitter arrays. The fused node replaces $c[i+1]$, $c[i]$ can be deallocated, and $c[i]$ together with the splitter $s[i]$ can be removed from the parent node.

Exercise 7.6 Suppose a node v has been produced by fusing two nodes as described above. Prove that the ordering invariant is maintained: Elements e reachable through child $v.c[i]$ have keys $v.s[i-1] < key(e) \leq v.s[i]$ for $1 \leq i \leq v.d$.

Balancing two neighbors works like first fusing them and then splitting the result in an analogous way to the splitting operation performed during an *insert*.

Since fusing two nodes decreases the degree of their parent, the need to fuse or balance might propagate up the tree. If the degree of the root drops to one this only makes sense if the tree has height one and hence contains only a single element. Otherwise a root of degree one is deallocated an replaced by its sole child — the height of the tree decreases.

As for *insert*, the execution time of *remove* is proportional to the height of the tree and hence logarithmic in the size of data structure. We can summarize the performance of (a,b)-tree in the following theorem:

Theorem 7.1 *For any constants $2 \leq a$ and $b \geq 2a-1$, (a,b)-trees support the operations insert, remove, and locate on n element sorted sequences in time $O(\log n)$.*

Exercise 7.7 Give a more detailed implementation of *locateLocally* based on binary search that needs at most $\lceil \log b \rceil$ *Key* comparisons. Your code should avoid both explicit use of infinite key values and special case treatments for extreme cases.

Exercise 7.8 Suppose $a = 2^k$ and $b = 2a$. Show that $(1+\frac{1}{k})\log n + 1$ element comparisons suffice to execute a *locate* operation in an (a,b)-tree. Hint, it is *not* quite sufficient to combine Exercise 7.3 with Exercise 7.7 since this would give you an additional term $+k$.

***Exercise 7.9 (Red-Black Trees.)** A *red-black tree* is a binary search tree where the edges are colored either red or black. The *black depth* of a node v is the number of black edges on the path from the root to v. The following invariants have to hold:

a) All leaves have the same black depth.

b) Edges into leaves are black.

c) No path from the root to a leaf contains two consecutive red edges.

Show that red-black trees and $(2,4)$-trees are isomorphic in the following sense: $(2,4)$-trees can be mapped to red-black trees by replacing nodes of degree three or four by two or three nodes connected by red edges respectively[bild spendieren?]. Red-black trees can be mapped to $(2,4)$-trees using the inverse transformation, i.e., components induced by red edges are replaced by a single node. Now explain how to implement $(2,4)$-trees using a representation as a red black tree.[3] Explain how expanding, shrinking, splitting, merging, and balancing nodes of the $(2,4)$-tree can be translated into recoloring and rotation operations in the red-black tree. Colors should only be stored at the target nodes of the corresponding edges.

$\Longrightarrow$

***Exercise 7.10 (Improved Sparse Tables.)** Develop a refinement of sparse tables [45] that has amortized insertion time $O(\log n)$. Hint: Your sparse table should store pointers to small *pages* of $O(\log n)$ elements. These pages can be treated in a similar way as the leaves of an (a,b)-tree.

7.3 More Operations

In addition to *insert*, *remove*, and *locate*, search trees can support many more operations. We start with the operations that are directly supported by the (a,b)-trees introduced in Section 7.2. Section 7.4 discusses further operations that require augmenting the data structure.

min/max The constant time operations *first* and *last* on a sorted list give us the smallest and largest element in the sequence in constant time. In particular, search trees implement *double ended priority queues*, i.e., sets that allow locating and removing both the smallest and the largest element in logarithmic time. For example, in Figure 7.6, the header element stored in the list ℓ gives us access to the smallest element 2 and to the largest element 19 via its *next* and *prev* pointers respectively.

Range queries: To retrieve all elements with key in the range $[x,y]$ we first locate x and then traverse the sorted list until we see an element with key larger than y. This takes time $O(\log n + \text{output-size})$. For example, the range query $[4,14]$ applied to the search tree in Figure 7.6 will find the 5, subsequently outputs 7, 11, 13, and stops when it sees the 17.

[3] This may be more space effi cient than a direct representation, in particular if keys are large.

Build/Rebuild: Exercise 7.11 asks you to give an algorithm that converts a sorted list or array into an (a,b)-tree in linear time. Even if we first have to sort an unsorted list, this operation is much faster than inserting the elements one by one. We also get a more compact data structure this way.

Exercise 7.11 Explain how to construct an (a,b)-tree from a sorted list in linear time. Give the $(2,4)$-tree that your routine yields for $\langle 1..17\rangle$. Finally, give the trees you get after subsequently removing 4, 9, and 16.

Concatenation: Two (a,b)-trees $s_1 = \langle w,\ldots,x\rangle$ and $s_2 = \langle y,\ldots,z\rangle$ can be concatenated in time $O(\log\max(|s_1|,|s_2|))$ if $x < y$. First, we remove the dummy item from s_1 and concatenate $s_1.\ell$ and $s_2.\ell$. Now the main idea is to fuse the root of one tree with a node of the other in tree in such a way that the resulting tree remains sorted and balanced. If $s_1.\mathit{height} \geq s_2.\mathit{height}$, we descend $s_1.\mathit{height} - s_2.\mathit{height}$ levels from the root of s_1 by following pointers to the rightmost children. The node v we reach is then fused with the root of s_2. The required new splitter key is the largest key in s_1. If the degree of v now exceeds b, v is split. From that point, the concatenation proceeds like an *insert* operation propagating splits up the tree until the invariant is fulfilled or a new root node is created. Finally, the lists $s_1.\ell$ and $s_2.\ell$ are concatenated. The case for $s_1.\mathit{height} < s_2.\mathit{height}$ is a mirror image. We descend $s_2.\mathit{height} - s_1.\mathit{height}$ levels from the root of s_2 by following pointers to the leftmost children. These operations can be implemented to run in time $O(1+|s_1.\mathit{height} - s_2.\mathit{height}|) = O(\log n)$. All in all, a *concat* works in $O(\log n)$ time. Figure 7.10 gives an example.

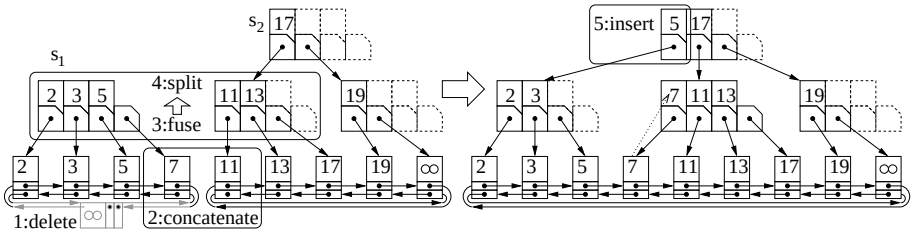


Figure 7.10: Concatenating $(2,4)$-trees for $\langle 2,3,5,7\rangle$ and $\langle 11,13,17,19\rangle$.

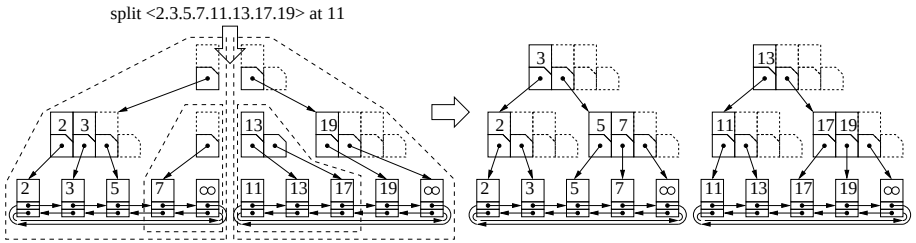


Figure 7.11: Splitting the $(2,4)$-tree for $\langle 2,3,5,7,11,13,17,19\rangle$ from Figure 7.6 yields the subtrees shown on the left. Subsequently concatenating the trees surrounded by the dashed lines leads to the $(2,4)$-trees shown on the right side.

Splitting: A sorted sequence $s = \langle w,\ldots,x,y,\ldots,z\rangle$ can be split into sublists $s_1 = \langle w,\ldots,x\rangle$ and $s_2 = \langle y,\ldots,z\rangle$ in time $O(\log n)$ by specifying the first element y of the second list. Consider the path from the root to the leaf containing the splitting element y. We split each node v on this path into two nodes. Node v_ℓ gets the children of v that are to the left of the path and v_r gets the children that are to the right of the path. Some of these nodes may be empty. Each of the nonempty nodes can be viewed as the root of an (a,b)-tree. Concatenating the left trees and a new dummy list element yields the elements up to x. Concatenating $\langle y\rangle$ and the right trees yields the list of elements starting from y. We can do these $O(\log n)$ concatenations in total time $O(\log n)$ by exploiting that the left trees have strictly decreasing height and the right trees have strictly increasing height. By concatenating starting from the trees with least height, $\Longrightarrow$ the total time (the sum of the height differences) is $O(\log n)$.[more gory details?] Figure 7.11 gives an example.

Exercise 7.12 Explain how to delete a subsequence $\langle e \in s : \alpha \le e \le \beta\rangle$ from an (a,b)-tree s in time $O(\log n)$.

7.3.1 Amortized Analysis of Update Operations

[somewhere mention that this is important for parallel processing where locking is expensive?] $\Longrightarrow$ The best case time for an insertion or removal is considerably smaller than the worst case time. In the best case, we basically pay to locate the affected element, the update of the list, and the time for updating the bottommost internal nodes. The worst case is much slower. Split or fuse operations may propagate all the way up the tree.

Exercise 7.13 Give a sequence of n operations on $(2,3)$-trees that requires $\Omega(n\log n)$ split operations.

We now show that the *amortized* complexity is much closer to the best case except if b has the minimum feasible value of $2a-1$. In Section 7.4.1 we will see variants of *insert* and *remove* that turn out to have constant amortized complexity in the light of the analysis below.

Theorem 7.2 *Consider an (a,b)-tree with $b \ge 2a$ that is initially empty. For any sequence of n insert or remove operations. the total number of split or fuse operations is $O(n)$.*

Proof: We only give the proof for $(2,4)$-trees and leave the generalization to Exercise 7.14. In contrast to the global insurance account that we used in Section 3.1 we now use a very local way of accounting costs that can be viewed as a *pebble game* using peanuts.[here or in priority queue section?] We pay one peanut for a *split* or $\Longleftarrow$ *fuse*. We require a *remove* to pay one peanut and an *insert* to pay two peanuts. We claim that this suffices to feed all the *split* and *fuse* operations. We impose an additional *peanut invariant* that requires nonleaf nodes to store peanuts according to the following table:

degree	1	2	3	4	5
peanuts	○○	○		○○	○○○○

Note that we have included the cases of degree 1 and 5 that violate the degree invariant and hence are only temporary states of a node.

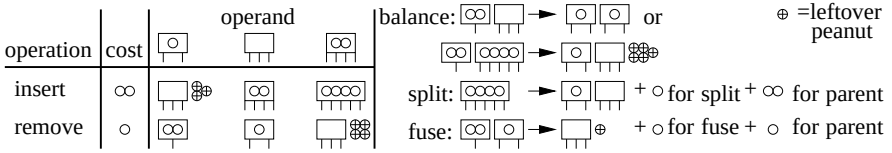


Figure 7.12: The effect of (a,b)-tree operations on the peanut invariant.

Since creating the tree inserts the dummy item, we have to pay two peanuts to get things going. After that we claim that the peanut invariant can be maintained. The peanuts paid by an *insert* or *remove* operation suffice to maintain the peanut invariant for the nonleaf node immediately above the affected list entries. A *balance* operation can only decrease the total number of peanuts needed at the two nodes that are involved.

A *split* operation is only performed on nodes of (temporary) degree five and results in left node of degree three and a right node of degree two. The four peanuts stored on the degree five node are spent as follows: One peanut is fed to the *split* operation itself. Two peanuts are used to maintain the peanut invariant at the parent node. One peanut is used to establish the peanut invariant of the newly created degree two node to the left. No peanut is needed to maintain the peanut invariant of the old (right) node that now has degree three.

A *fuse* operation fuses a degree one node with a degree two node into a degree three node. The $2+1=3$ peanuts available are used to feed one peanut to the *fuse* operation itself and to maintain the peanut invariant of the parent with one peanut.[4] Figure 7.12 summarizes all peanut pushing transformations. ∎

***Exercise 7.14** Generalize the proof for arbitrary $b \geq 2a$. Show that n *insert* or *remove* operations cause only $O(n/(b-2a+1))$ fuse or split operations.

Exercise 7.15 (Weight balanced trees.) Consider the following variant of (a,b)-trees: The node-by-node invariant $d \geq a$ is relaxed to the global invariant that the tree leads to at least $2a^{height-1}$ elements. Remove does not perform any *fuse* or *balance* operations. Instead, the whole tree is rebuild using the routine from Exercise 7.11 when the invariant is violated. Show that *remove* operations execute in $O(\log n)$ amortized $\Longrightarrow$ time. [check]

7.4 Augmenting Search Trees

By adding additional information to a search tree data structure, we can support additional operations on the sorted sequence they represent. Indeed, storing the sequence elements in a doubly linked list can already be viewed as an augmentation that allows us to accelerate *first*, *last*, and range queries. However, augmentations come at a cost. They consume space and require time for keeping them up to date. Augmentations may also stand in each others way. The two augmentations presented in the following are an example for useful augmentations that do not work well together.

Exercise 7.16 (Avoiding Augmentation.) Explain how to support range queries in time $O(\log n + \text{output-size})$ *without* using the *next* and *prev* pointers of the list items.

[4] There is one peanut left over. Please send it to the authors.

7.4.1 Parent Pointers

Suppose we want to remove an element specified by the handle of a list item. In the basic implementation from Section 7.2, the only thing we can do is to read the key k of the element and call $remove(k)$. This would take logarithmic time for the search although we know from Section 7.3.1 that the amortized number of *fuse* operations propagating up the tree will only be constant. This detour is not necessary if each node v of the tree stores a handle indicating its *parent* in the tree (and perhaps an index i such that $v.parent.c[i] = v$).

Exercise 7.17 Show that in an (a,b)-trees with parent pointers, the operations $remove(h : Item)$ and $insertAfter(h : Item)$ can be implemented to run in constant amortized time.

***Exercise 7.18 (Avoiding Augmentation.)** Outline the implementation of a class *ABTreeIterator* that represents a position in a sorted sequence in an ABTree *without* parent pointers. Creating an iterator I works like *search* and may take logarithmic time. The class should support operations *remove*, *insertAfter*, and operations for moving backward and forward in the host sequence by one position. All these operations should use constant amortized time. Hint: You may need a logarithmic amount of internal state.

***Exercise 7.19 (Finger Search.)** Explain how to augment a search tree such that searching can profit from a "hint" given in the form of the handle of a *finger element* e'. If the sought element has rank r and the finger element e' has rank r', the search time should be $O(\log|r-r_0|)$. Hint: One solution links all nodes at each level of the search tree into a doubly linked list.

***Exercise 7.20 (Optimal Merging.)** Explain how to use finger search to implement merging of two sorted sequences in time $O\left(n \log \frac{m}{n}\right)$ where n is the size of the shorter sequence and m is the size of the longer sequence.

7.4.2 Subtree Sizes

Suppose every nonleaf node t of a search tree stores its *size*, i.e., $t.size$ is the number of leaves in the subtree rooted at t. Then the k-th smallest element of the sorted sequence can be selected in time proportional to the height of the tree. For simplicity we describe this for binary search trees. Let t denote the current search tree node which is initialized to the root. The idea is to descend the tree while maintaining the invariant that the k-th element is contained in the subtree rooted at t. We also maintain the number i of the elements that are to the *left* of t. Initially, $i = 0$. Let i' denote the

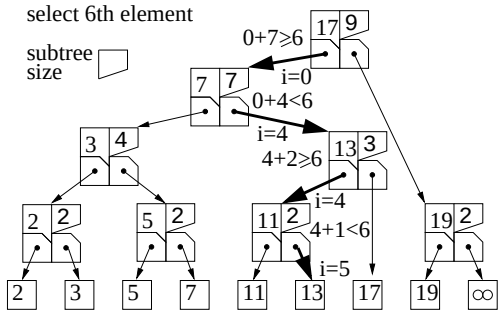


Figure 7.13: Selecting the 6th-smallest element from $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a binary search tree.

size of the left subtree of t. If $i+i' \geq k$ then we set t to its left successor. Otherwise t is set to its right successor and i is incremented by i'. When a leaf is reached, the invariant ensures that the k-th element is reached. Figure 7.13 gives an example.

Exercise 7.21 Generalize the above selection algorithm for (a,b)-trees. Develop two variants. One that needs time $O(b\log_a n)$ and stores only the subtree size. Another variant needs only time $O(\log_a n)$ and stores $d-1$ sums of subtree sizes in a node of degree d.

Exercise 7.22 Explain how to determine the rank of a sequence element with key k in logarithmic time.

Exercise 7.23 A colleague suggests to support both logarithmic selection time and constant amortized update time by combining the augmentations from Sections 7.4.1 and 7.4.2. What goes wrong?

7.5 Implementation Notes

⟹ Our pseudocode[einheitliche Schreibweise] for (a,b)-trees is quite close to an actual implementation in a language like C++ except for a few oversimplifications. The temporary arrays s' and c' in *insertRec* and *removeRec* can be avoided by appropriate case distinctions. In particular, a balance operation will not require calling the memory manager. A split operation of a node v might get slightly faster if v keeps the left half rather than the right half. We did not formulate it this way because then the cases

of inserting a new list element and splitting a node are no longer the same from the point of view of their parent.

For large b, *locateLocally* should use binary search. For small b, linear search might be OK. Furthermore, we might want to have a specialized implementation for small, fixed values of a and b that *unrolls*[5] all the inner loops. Choosing b to be a power of two might simplify this task.

Of course, the crucial question is how a and b should be chosen. Let us start with the cost of *locate*. There are two kinds of operations that (indirectly) dominate the execution time of *locate*: element comparisons (because they may cause branch mispredictions[needed anywhere else? mention ssssort in sort.tex?][6]) and pointer ⟸ dereferences (because they may cause cache faults). Exercise 7.8 indicates that element comparisons are minimized by choosing a as large as possible and $b \leq 2a$ should be a power of two. Since the number of pointer dereferences is proportional to the height of the tree (cf. Exercise 7.3), large values of a are also good for this measure. Taking this reasoning to the extreme, we would get best performance for $a \geq n$, i.e., a single sorted array. This is not so astonishing. By neglecting update operations, we are likely to end up with a *static* search data structure looking best.

Insertions and deletions have the amortized cost of one *locate* plus a constant number of node reorganizations (split, balance, or fuse) with cost $O(b)$ each. We get logarithmic amortized cost for update operations for $b = O(\log n)$. A more detailed analysis [64, Section 3.5.3.2] would reveal that increasing b beyond $2a$ makes split and fuse operations less frequent and thus saves expensive calls to the memory manager associated with them. However, this measure has a slightly negative effect on the performance of *locate* and it clearly increases *space consumption*. Hence, b should remain close to $2a$.

Finally, let us have a closer look at the role of cache faults. It is likely that $\Theta(M/b)$ nodes close to the root fit in the cache. Below that, every pointer dereference is associated with a cache miss, i.e., we will have about $\log_a(bn/\Theta(M))$ cache misses in a cache of size M provided that a complete search tree node fits into a single cache block. [some experiments?] Since cache blocks of processor caches start at addresses that ⟸ are a multiple of the block size, it makes sense to *align* the starting address of search tree nodes to a cache block, i.e., to make sure that they also start at an address that is a multiple of the block size. Note that (a,b)-trees might well be more efficient than binary search for large data sets because we may save a factor $\log a$ cache misses.

[5] *Unrolling* a loop "**for** $i := 1$ **to** K **do** *body*$_i$" means replacing it by the *straight line program* "*body*$_1$,...,*body*$_K$". This saves the overhead for loop control and may give other opportunities for simplifications.

[6] Modern microprocessors attempt to execute many (up to a hundred or so) instructions in parallel. This works best if they come from a linear, predictable sequence of instructions. The branches in search trees have a 50 % chance of going either way *by design* and hence are likely to disrupt this scheme. This leads to large delays when many partially executed instructions have to be discarded.

Very large search trees are stored on disks. Indeed, under the name *BTrees* [8], (a,b)-tree are the working horse of indexing data structures in data bases. In that case, internal nodes have a size of several KBytes. Furthermore, the linked list items are also replaced by entire data blocks that store between a' and b' elements for appropriate values of a' and b' (See also Exercise 3.19). These leaf blocks will then also be subject to splitting, balancing and fusing operations. For example, assume we have $a = 2^{10}$, the internal memory is large enough (a few MBytes) to cache the root and its children, and data blocks store between 16 and 32 KBytes of data. Then two disk accesses are sufficient to *locate* any element in a sorted sequence that takes 16 GBytes of storage. Since putting elements into leaf blocks dramatically decreases the total space needed for the internal nodes and makes it possible to perform very fast range queries, this measure can also be useful for a cache efficient internal memory implementation. However, note that update operations may now move an element in memory and thus will invalidate element handles stored outside the data structure.

There are many more tricks for implementing (external memory) (a,b)-trees that are beyond the scope of this book. Refer to [40] and [72, Chapters 2,14] for overviews. Even from the augmentations discussed in Section 7.4 and the implementation trade-offs discussed here you have hopefully learned that *the* optimal implementation of sorted sequences does not exist but depends on the hardware and the operation mix relevant for the actual application. We conjecture however, that in internal memory (a,b)-trees with $b = 2^k = 2a = O(\log n)$ augmented with parent pointers and a doubly linked list of leaves will yield a sorted sequence data structure that supports a wide range of operations efficiently.

Exercise 7.24 How do you have to choose a and b in order to guarantee that the number of I/O operations to perform for *insert*, *remove*, or *locate* is $O\left(\log_{B/4} \frac{n}{M}\right)$? How many I/O are needed to *build* an n elements (a,b)-tree using the external sorting algorithm from Section 5.7 as a subroutine? Compare this with the number of I/Os needed for building the tree naively using insertions. For example, try $M = 2^{29}$ byte, $B = 2^{18}$ byte[7] , $n = 2^{32}$, and elements have 8 byte keys and 8 bytes of associated information.

C++

The STL has four container classes *set*, *map*, *multiset*, and *multimap*. The prefix *multi* means that there may be several elements with the same key. A *map* offers an array-like interface. For example, *someMap*$[k]$:= x would insert or update the element with key k and associated information x.

[7] We are committing a slight oversimplification here since in practice one will use much smaller block sizes for organizing the tree than for sorting.

Exercise 7.25 Explain how our implementation of (a,b)-trees can be generalized to implement multisets. Element with identical key should be treated like a FIFO, i.e., *remove*(k) should remove the least recently inserted element with key k.

The most widespread implementation of sorted sequences in STL uses a variant of red-black trees with parent pointers where elements are stored in all nodes rather than in the leaves. None of the STL data types supports efficient splitting or concatenation of sorted sequences.

LEDA offers a powerful interface *sortseq* that supports all important (and many not so important) operations on sorted sequences including finger search, concatenation, and splitting. Using an implementation parameter, there is a choice between (a,b)-trees, red-black trees, randomized search trees, weight balanced trees, and skip lists.[8] [todo: perhaps fix that problem in LEDA? Otherwise explain how to declare this in LEDA. PS was not able to extract this info from the LEDA documentation.] ⟸

Java

The Java library *java.util*[check wording/typesetting in other chapters] offers the ⟸ interface classes *SortedMap* and *SortedSet* which correspond to the STL classes *set* and *map* respectively. There are implementation classes, namely *TreeMap* and *TreeSet* respectively based on red-black trees.

7.6 Further Findings

There is an entire zoo of sorted sequence data structures. If you just want to support *insert*, *remove*, and *locate* in logarithmic time, just about any of them might do. Performance differences are often more dependent on implementation details than on fundamental properties of the underlying data structures. We nevertheless want to give you a glimpse on some of the more interesting species. However, this zoo displays many interesting specimens some of which have distinctive advantages for more specialized applications.

The first sorted sequence data structure that supports *insert*, *remove*, and *locate* in logarithmic time were AVL trees [1]. AVL trees are binary search trees which maintain the invariant that the heights of the subtrees of a node differ by at most one. Since this is a strong balancing condition, *locate* is probably a bit faster than in most competitors. On the other hand, AVL trees do *not* support constant amortized update costs. Another small disadvantage is that storing the heights of subtrees costs additional space. In

[8] Currently, the default implementation is a remarkably inefficient variant of skip lists. In most cases you are better off choosing, e.g., $(4,8)$-trees [27].

comparison, red-black trees have slightly higher cost for *locate* but they have faster updates and the single color bit can often be squeezed in somewhere. For example, pointers to items will always store even addresses so that their least significant bit could be diverted to storing color information.

Splay trees [90] and some variants of randomized search trees [86] even work without any additional information besides one key and two successor pointers. A more interesting advantage of these data structures is their *adaptivity* to nonuniform access frequencies. If an element e is accessed with probability p then these search trees will over time be reshaped to allow an access to e in time $O\left(\log \frac{1}{p}\right)$. This can be shown to be asymptotically optimal for any comparison based data structure.

⟹ [sth about the advantages of weight balance?]

There are so many *search tree* data structures for *sorted sequences* that these two terms are sometimes used as synonyms. However, there are equally interesting data structures for sorted sequences that are *not* based on search trees. In the introduction, we have already seen sorted arrays as a simple *static* data structure and sparse tables [45] as a simple way to make sorted arrays dynamic. Together with the observation from Exercise 7.10 [9] this yield an data structure which is asymptotically optimal in an amortized sense. Moreover, this data structure is a crucial ingredient for a sorted sequence data structure [9] which is *cache oblivious* [37], i.e., cache efficient on any two levels of a memory hierarchy without even knowing the size of caches and cache blocks. The other ingredient are cache oblivious *static* search trees [37] — perfectly balance binary search trees stored in an array such that any search path will exhibit good cache locality in any cache. We describe the *van Emde Boas layout* used for this purpose for the case that there are $n = 2^{2^k}$ leaves for some integer k: Store the top 2^{k-1} levels of the tree in the beginning of the array. After that, store the 2^{k-1} subtrees of depth 2^{k-1} allocating consecutive blocks of memory for them. Recursively allocate the resulting $1+2^{k-1}$ subtrees. At least static cache oblivious search trees are practical in the sense that they can outperform binary search in a sorted array.

Skip lists [80] are based on another very simple idea. The starting point is a sorted linked list ℓ. The tedious task of scanning ℓ during *locate* can be accelerated by producing a shorter list ℓ' that only contains some of the elements in ℓ. If corresponding elements of ℓ and ℓ' are linked, it suffices to scan ℓ' and only descend to ℓ when approaching the searched element. This idea can be iterated by building shorter and shorter lists until only a single element remains in the highest level list. This data structure supports all important operations efficiently in an expected sense. Randomness comes in because the decision which elements to lift to a higher level list is made randomly. Skip lists are particularly well suited for supporting finger search.

Yet another familie of sorted sequence data structures comes into play when we no longer consider keys as atomic objects that can only be compared. If keys are numbers

in binary representation, we get faster data structures using ideas similar to the fast numeric sorting algorithms from Section 5.6. For example, sorted sequences with w bit integer keys support all operations in time $O(\log w)$ [93, 66]. At least for 32 bit keys these ideas bring considerable speedup also in practice [27]. Not astonishingly, string keys are important. For example, suppose we want to adapt (a,b)-trees to use variable length strings as keys. If we want to keep a fixed size for node objects, we have to relax the condition on the minimal degree of a node. Two ideas can be used to avoid storing long string keys in many nodes: *common prefixes* of keys need to be stored only once, often in the parent nodes. Furthermore, it suffices to store the *distinguishing prefixes* of keys in inner nodes, i.e., just enough characters to be able to distinguish different keys in the current node. Taking these ideas to the extreme we get *tries* [34][check], a search tree data structure specifically designed for strings ⟸ keys: Tries are trees whose edges are labelled by characters or strings. The characters along a root leaf path represent a key. Using appropriate data structures for the inner nodes, a trie can be searched in time $O(s)$ for a string of size s. [suffix tree and array wrden zu weit fhren?] ⟸

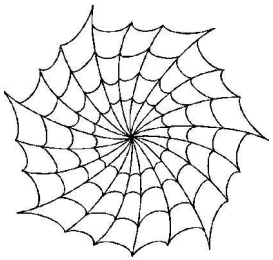
We get a more radical generalization of the very idea of sorted sequences when looking at more complex objects such as intervals or pointd in d-dimensional space. We refer to textbooks on geometry for this wide subject [26][cite more books?]. ⟸

Another interesting extension of the idea of sorted sequences is the concept of *persistence*. A data structure is persistent if it allows us to update a data structure and at the same time keep arbitrary old versions around. [more? what to cite?] ⟸

[what else] ⟸

Chapter 8

Graph Representation



[Definition grundlegender Begriffe schon in Intro. Weitere Begriffe dort wo sie das erste mal gebraucht werden. Zusaetzlich Zusammenfassung der Defs. im Anhang.] ⟸

Nowadays scientific results are mostly available in the form of articles in journals, conference proceedings, and on various web resources. These articles are not self contained but they cite previous articles with related content. However, when you read an article from 1975 with an interesting partial result, you often ask yourselves what is the current state of the art. In particular, you would like to know which newer papers cite the old paper. Projects like citeseer[1] *work on providing this functionality by building a database of articles that efficiently support looking up articles citing a given article.*

We can view articles as the nodes in a directed graph where[which terms need definition] an edge (u,v) means u cites v. In the paper representation, we know the ⟸ outgoing edges of a node u (the articles cited by u) but not the incoming edges (the articles citing u). We can see that even the most elementary operations on graphs can be quite costly if we do not have the right representation.

This chapter gives an introduction into the various possibilities of graph representation in a computer. We mostly focus on directed graphs and assume that an undirected graph $G = (V,E)$ is represented in the same way as the (bi)directed graph $G' = (V, \bigcup_{\{u,v\}\in E} \{(u,v),(v,u)\})$. Figure 8.1 gives an example. Most of the presented data structures also allow us to represent parallel edges and self-loops. The most important question we have to ask ourselves is what kind of operations we want to support.

Accessing associated information: Given a node or an edge, we want to access information directly associated to it, e.g., the edge weight. In many representations

[1] `http://citeseer.nj.nec.com/cs`

nodes and edges are objects and we can directly store this information as a member of these objects. If not otherwise mentioned, we assume that $V = \{1,\ldots,n\}$ so that information associated with nodes can be stored in arrays. When all else fails, we can always store node or edge information in a hash table. Hence, elementary accesses can always be implemented to run in constant time. In the remainder of this book we abstract from these very similar options by using data types *NodeArray* and *EdgeArray* to indicate an array-like data strucure that can be addressed by node or edge labels respectively.

Navigation: Given a node we want to access its outgoing edges. It turns out that this operation is at the heart of most graph algorithms. As we have seen in the scientific article example, we sometimes also want to know the incoming edges.

Edge Queries: Given a pair of nodes (u,v) we may want to know whether this edge is in the graph. This can always be implemented using a hash table but we may want to have something even faster. A more specialized but important query is to find the *reverse edge* (v,u) of a directed edge $(u,v) \in E$ if it exists. This operation can be implemented by storing additional pointers connecting edges with their reverse edge.

Construction, Conversion and Output: The representation most suitable for the algorithmic problem at hand is not always the representation given initially. This is not a big problem since most graph representations can be translated into each other
$\Longrightarrow$ in linear time. However, we will see examples[do we? in mst chapter?], where conversion overhead dominates the execution time.

Update: Sometimes we want to add or remove nodes or edges one at a time. Exactly what kind of updates are desired is what can make the representation of graphs a nontrivial topic.

$\Longrightarrow$ [somewhere a more comprehensive overview of operations?]

8.1 Edge Sequences

Perhaps the simplest representation of a graph is a sequence of edges. Each edge contains a pair of node indices and possibly associated information like an edge weight. Whether these node pairs represent directed or undirected edges is merely a matter of interpretation. Sequence representation is often used for input and output. It is easy to add edges or nodes in constant time. However, many other operations, in particular navigation take time $\Theta(m)$ which is forbiddingly slow. In Chapter 11 we will see an example where one algorithm can work well with an edge sequence representation whereas the other algorithm needs a more sophisticated data structure supporting fast access to adjacent edges.

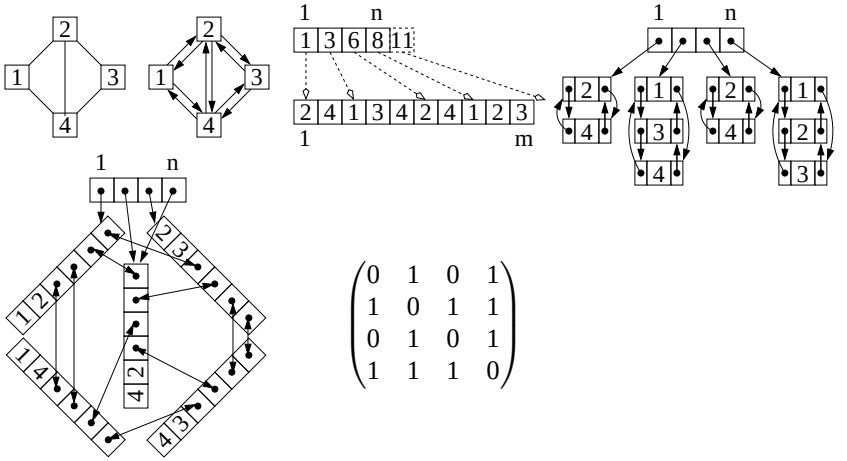


Figure 8.1: The top row shows an undirected graph, its interpretation as a bidirected graph, and representations of this bidirected graph by adjacency array and by adjancency list. The bottom part shows a direct representation of the undirected graph using linked edge objects and the adjacency matrix.

8.2 Adjacency Arrays — Static Graphs

To allow navigation in constant time per edge, we can store the edges leaving a node together in an array. In the simplest case without edge weights this array would just contain the indices of the target nodes. If the graph is *static*, i.e., it does not change, we can concatenate all these little arrays into a single edge array E. An additional array V stores the index of the first edge in E leaving node v in $V[v]$. It is convenient to add a dummy entry $V[n+1] = m+1$. The edges of node v are then easily accessible as $E[V[v]], \ldots, E[V[v+1]-1]$. Figure 8.1 shows an example.

The memory consumption for storing a directed graph using adjacency arrays is $n+m+\Theta(1)$ words. This is even more compact than the $2m$ words needed for an edge sequence representation.

Adjacency array representations can be generalized to store additional information: Information associated with edges can be stored in separate arrays or within the edge array. If we also need incoming edges, additional arrays V', E' can store $G' = (V, \{(u,v) : (v,u) \in E\})$.

Exercise 8.1 Design a linear time algorithm for converting an edge sequence representation of a directed graph into adjacency array representation. You should use only

$O(1)$ auxiliary space. Hint: View the problem as the task to sort edges by their source node and adapt the integer sorting algorithm from Figure 5.14.

8.3 Adjacency Lists — Dynamic Graphs

The main disadvantage of adjacency arrays as presented above is that it is expensive to add or remove edges. For example assume we want to insert a new edge (u,v) and there is enough room in the edge array E to accommodate it. We still have to move the edges associated with nodes $u+1,\ldots,n$ one position to the right which takes time $O(m)$.

Of course, with our knowledge of representing sequences from Chapter 3 the solution is not far away: we can associate a sequence E_v of edges with each node v. E_v may in turn be represented by an unbounded array or by a (singly or doubly) linked list. We inherit the advantages and disadvantages of the respective sequence representations. Unbounded arrays may be more cache efficient. Linked lists allow worst case constant time insertion and deletion of edges at arbitrary positions. It is worth noting that adjacency lists are usually implemented without the header item introduced in Section 3.2 because adjacency lists are often very short so that an additional item would consume too much space. In the example in Figure 8.1 we show circurlarly linked lists instead.

Exercise 8.2 Suppose the edges adjacent to a node u are stored in an unbounded array E_u and an edge $e=(u,v)$ is specified by giving its position in E_u. Explain how to remove $e=(u,v)$ in constant amortized time. Hint, you do *not* have to maintain the relative order of the other edges.

Exercise 8.3 Explain how to implement the algorithm for testing whether a graph a is acyclic from Chapter 2.8 in linear time, i.e., design an appropriate graph representation and an algorithm using it efficiently. Hint: maintain a queue of nodes with outdegree zero.

Linked Edge Objects

Sometimes it is more natural to store all the information associated with an edge at one place so that it can be updated easily. This approach can be applied to undirected graphs directly without the detour over a bidirected graph. If we want doubly linked adjacency information this means that each edge object for edge $\{u,v\}$ stores four pointers. Two are used for the adjacency information with respect to u, i.e., the edges incident to u form a doubly linked list based on these pointers. In an analogous way, the two other pointers represent the adjacency information with respect to

v. The bottom part of Figure 8.1 gives an example. A node u now stores a pointer to one incident edge $e=\{u,w\}$. To find the identity of w we have to inspect the node information stored at e. To find further nodes incident to u we have to inspect the adjacency information with respect to u which is also stored at e. Note that in order to find the right node or adjacency information we either need to compare u with the node information stored at e, or we need a bit stored with u that tells us where to look. (We can then use this bit as an index into two-element arrays stored at e.)

8.4 Adjacency Matrix Representation

An n-node graph can be represented by an $n\times n$ *adjacency matrix* $\mathbf{A}$. $\mathbf{A}_{ij}$ is 1 if $(i,j)\in E$ and 0 otherwise. Edge insertion or removal and edge queries work in constant time. It takes time $O(n)$ to get the edges entering or leaving a node. This is only efficient for very dense graphs with $m=\Omega\left(n^2\right)$. Storage requirement is n^2 bits which can be a factor up to 64 better than adjacency arrays for very dense graphs but a factor $\Omega(n)$ worse for the frequent case of very sparse graph with $m=O(n)$. Even for dense graphs, the advantage is small if we need additional edge information.

Exercise 8.4 Explain how to represent an undirected loopless graph with n nodes using $n(n-1)/2$ bits.

Perhaps more important than actually storing the adjacency matrix is the conceptual link between graphs and linear algebra introduced by the adjacency matrix. On the one hand, graph theoretic problems can be solved using methods from linear algebra. For example, if $\mathbf{C}=\mathbf{A}^k$, then $\mathbf{C}_{ij}$ counts the number of paths from i to j with exactly k edges.

Exercise 8.5 Explain how to store an $n\times n$ matrix $\mathbf{A}$ with m nonzero entries using storage $O(m+n)$ such that a matrix vector multiplication $\mathbf{Ax}$ can be performed in time $O(m+n)$. Describe the multiplication algorithm. Expand your representation so that products of the form $\mathbf{x}^T\mathbf{A}$ can also be computed in time $O(m+n)$.

On the other hand, graph theoretic concepts can be useful for solving problems from linear algebra. For example, suppose we want to solve the matrix equation $\mathbf{Bx}=\mathbf{c}$ where B is a symmetric matrix. Now consider the corresponding adjacency matrix $\mathbf{A}$ where $\mathbf{A}_{ij}=1$ if and only if $\mathbf{B}_{ij}\neq 0$. If an algorithm for computing connected components finds out that the undirected graph represented by $\mathbf{A}$ contains two disconnected components, this information can be used to reorder the rows and columns of

B such that we get an equivalent equation of the form

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} .$$

This equation can now be solved by solving $\mathbf{B}_1\mathbf{x}_1 = \mathbf{c}_1$ and $\mathbf{B}_2\mathbf{x}_2 = \mathbf{c}_2$ seperately.[2]

[Exercise: how to exploit strongly connected compontents? special case, acyclic graph?]

8.5 Implicit Representation

Many applications work with graphs of special structure. This structure can be exploited to get simpler and more efficient representations. For example, The *grid graph* $G_{k\ell}$ with node set $V = (\{0,\ldots,k-1\} \times \{0,\ldots,\ell-1\})$ and edge set

$$E = \big\{((i,j),(i,j')) \in V^2 : |j-j'| = 1\big\} \cup \big\{((i,j),(i',j)) \in V^2 : |i-i'| = 1\big\}$$

is completely defined by the two parameters k and ℓ . Figure 8.2 shows $G_{3,4}$. Edge weights could be stored in two two-dimensional arrays, one for the vertical edges and one for the horizontal edges.

[refs to examples for geometric graphs?]

Exercise 8.6 Nodes of *interval graphs* can be represented by real intervals $[v_l, v_r]$. Two nodes are adjancent iff their intervals overlap. You may assume that these intervals are part of the input.

a) Devise a representation of interval graphs that needs $O(n)$ storage and supports navigation in constant expected time. You may use preprocessing time $O(n \log n)$.

[2] In practice, the situation is more complicated since we rarely get disconnected Matrices. Still, more sophisticated graph theoretic concepts like cuts can be helpful for exploiting the structure of the matrix.

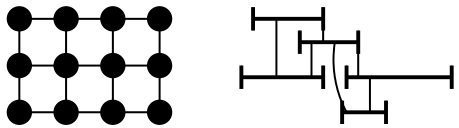


Figure 8.2: The grid graph G_{34} (left) and an interval graph with 5 nodes and 6 edges (right).

b) As the first part but you additionally want to support node insertion and removal in time $O(\log n)$.

c) Devise an algorithm using your data structure that decides in linear time whether the interval graph is connected.

8.6 Implementation Notes

If we want to maximum performance, we may have to fine tune our graph representation to the task. An edge sequence representation is good only in specialized situations. Adjacency matrices are good for rather dense graphs. Adjacency lists are good if the graph is changing frequently. Very often some variant of adjacency arrays is fastest. This may be true even if the graph is changing because often there are only few changes, changes can be agglomerated into an occasional rebuilding of the graph, or changes are avoided by building several related graphs.

But there are many variants of adjacency array representations. It can even matter whether information associated with nodes and edges is stored together with these objects or in separate arrays. A rule of thumb is that information that is frequently accessed should be stored with the nodes and edges. Rarely used data should usually be kept in separate arrays because otherwise it would often be uselessly moved through the cache even if it is not used. There can be other, more complicated reasons why separate arrays are faster. For example, if both adjacency information and edge weights are read but only the weights are changed then seperate arrays may be faster because the amount of data written back to the main memory is reduced.

Unfortunately, this wide variety of graph representations leads to a software engineering nightmare. We are likely to end up with a new representation for every new application. We might even have to change the representation when the requirements for one application change. This is so time consuming and error prone that it is usually not practical.

It can be more economical to use a general purpose library like LEDA most of the time and only decide to use selected custom-built components for time critical or space critical parts of an application. Often it will turn out that the graph theoretic part of the application is not the bottleneck so that no changes are needed.

Another possibility is generic programming. Graph algorithms can be implemented so that they are largely independent of the actual representation used. When we add a new representation, we only have to implement a small set of interface functions used by the generic algorithms. This concept is easiest to illustrate using a simpler data structure like sequences. The standard C++ function *sort* can sort any sequence regardless whether it is implemented by a list, an array or some more complicated data structure. For example, if we decide to use generic algorithms from the

C++ library for our unbounded array data structure from Section 3.1, we only have to implement the data type *iterator* which is basically an abstraction of a pointer into the sequence.

⟹ [sth on standard formats?]

C++

LEDA [67] has a very powerful graph data type that is space consuming but supports a large variety of operations in constant time. It also supports several more space efficient adjacency array based representations.

The Boost graph library [3] emphasizes a consistent separation of representation and interface. In particular, by implementing the Boost interface, a user can run Boost graph algorithms on her own (legacy) graph representation. With *adjacency_list* Boost also has its own graph representatotion class. A large number of parameters allow to choose between variants of graphs (directed, undirected, multigraph) type of available navigation (in-edges, out-edges,...) and representations of vertex and edge sequences (arrays, linked lists, sorted sequences,...). However, it should be noted that even the array representation is not what we call adjacency array representation here because one array is used for the adjacent edges of each vertex. [some qualified criticism?
⟹ Is this still easy to use?]

Java

⟹ [JGraphT looks promising but is still in version 0.sth.]

⟹ The library JDSL [4][todo: consistent citation of all libraries, stl,leda,boost,jdsl,java.util,???] offers rich support for graphs in *jdsl.graph*. JDSL has a clear separation between interfaces, algorithms, and representation. The class *jdsl.graph.ref.IncidenceListGraph* implements an adjacency list representation of graphs that allows a mix of (possibly parallel) directed and undirected edges.

8.7 Further Findings

Special classes of graphs may result in additional requirements for their representation. An important example are *planar graphs* — graphs that can be drawn on the plane without crossing edges. Here, the ordering of the edges adjacent to a node should be in counterclockwise order with respect to a planar drawing of the graph. In addition, the graph data structure should efficiently support iterating over the edges along a *face* of the graph — a cycle that does not enclose any other nodes.

[move bipartite and hypergraphs into intro chapter?] *Bipartite graphs* are ⟸ special graphs where the node set $V = L \cup R$ can be decomposed into two disjoint subsets L and R so that edges are only between nodes in L and R.

Hypergraphs $H = (V,E)$ are generalizations of graphs where edges can connect more than two nodes. Often hypergraphs are represented as the bipartite graph $B_H = (E \cup V, \{(e,v) : e \in E, v \in e\})$.

Cayley graphs [5] are an interesting example for implicitly defined graphs. Recall that a set V is a *group* if it has a associative multiplication operation $*$, a neutral element, and a multiplicative inverse operation. The *Cayley graph* (V,E) with respect to a set $S \subseteq V$ has the edge set $\{(u,u*s) : u \in V, s \in S\}$. Cayley graphs are useful because graph theoretic concepts can be useful in group theory. On the other hand, group theory yields concise definitions of many graphs with interesting properties. For example, Cayley graphs have been proposed as the interconnection networks for parallel computers [7].

In this book we have concentrated on convenient data structures for *processing graphs*. There is also a lot work on *storing* graphs in a flexible, portable, and space efficient way. Significant compression is possible if we have a priori information on the graphs. For example, the edges of a triangulation of n points in the plan can be representd with about $6n$ bits [24, 82].

[OBDDs??? or is that too difficult] [what else?] ⟸ ⟸

[3] www.boost.org
[4] www.jdsl.org

[5] We use the defi nition from Wikipedia.

Chapter 9

Graph Traversal

Suppose you are working in the traffic planning department of a nice small town with a medieval old town full of nooks and crannies. An unholy coalition of the shop owners who want more streeside parking opportunities and the green party that wants to discourage car traffic all together has decided to turn most streets into one-way streets. To avoid the worst, you want to be able to quickly find out whether the current plan is at least feasible in the sense that one can still drive from every point in the town to every other point.

Using the terminology of graph theory, the above problem asks whether the directed graph formed by the streets is strongly connected. The same question is important in many other applications. For example, if we have a communication network with unidirectional channels (e.g., radio transceivers with different ranges) we want to know who can communicate with whom. Bidirectional communication is possible within components of the graph that are strongly connected. Computing strongly connected components (SCCs) is also an important subroutine. For example, if we ask for the minimal number of new edges in order to make a graph $G = (V,E)$ strongly connected, we can regard each SCC as a single node of the smaller graph $G = (V',E')$ whose nodes are the SCCs of G and with the edge set

$$E = \left\{(u',v') \in V' \times V' : \exists (u,v) \in E : u \in u' \wedge v \in v'\right\} \quad .$$

In G' we have *contracted* SCCs to a single node. Since G' cannot contain any directed cycles (otherwise we could have build larger SCCs), it is a directed acyclic graph (DAG). This might further simplify our task. Exercise 9.9 gives an example, where a graph theoretic problem turns out to be easy since it is easy to solve on SCCs and easy to solve on DAGs.

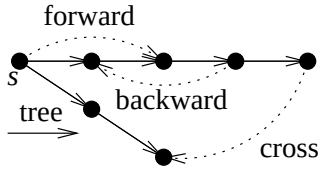


Figure 9.1: Classification of graph edges into tree edges, forward edges, backward edges, and cross edges.

We present a simple and efficient algorithm for computing SCCs in Section 9.2.2. The algorithm systematically explores the graph, inspecting each edge exactly once and thus gathers global information. Many graph problems can be solved using a small number of basic traversal strategies. We look at the two most important of them: Breadth first search in Section 9.1 and depth first search in Section 9.2. Both algorithms have in common that they construct trees that provide paths from a root node s to all nodes reachable from s. These trees can be used to distinguish between four classes of edges (u,v) in the graph: The *tree* edges themselves, *backward* edges leading to an ancestor of v on the path, *forward* edges leading to an indirect descendent of v, and *cross* edges that connect two different branches of the tree. Figure 9.1 illustrates the four types of edges. This classification helps us to gather global information about the graph.

9.1 Breadth First Search

A simple way to explore all nodes reachable from some node s is *breadth first search (BFS)* shown in Figure 9.2. This approach has the useful feature that it finds paths with a minimal number of edges. For example, you could use such paths to find railway connections that minimize the number of times you have to change trains. In some communication networks we might be interested in such paths because they minimize the number of failure-prone intermediate nodes. To encode the paths efficiently, we use a simple trick. Each node reachable from s just stores its parent node in the spanning tree. Thus, by following these parent pointers from a node v, we can easily reconstruct the path from s to v. We just have to keep in mind that we get the nodes on the path in reverse order.

Exercise 9.1 (FIFO BFS) Explain how to implement BFS using a single FIFO queue of nodes whose outgoing edges still have to be scanned.


```
Function bfs(s : Node) : NodeArray of Node
    parent=⟨⊥,...,⊥⟩ : NodeArray of Node          // all nodes are unexplored
    parent[s]:= s                                 // self-loop signals root
    q=⟨s⟩ : Set of Node                           // current layer of BFS tree
    q'=⟨⟩ : Set of Node                           // next layer of BFS tree
    for d := 0 to ∞ while q ≠ ⟨⟩ do               // explore layer by layer
        invariant q contains all nodes with distance d from s
        foreach u ∈ q do
            foreach (u,v) ∈ E do                  // scan edges out of u
                if parent(v) = ⊥ then             // found an unexplored node
                    q':= q' ∪ {v}                 // remember for next layer
                    parent(v):= u                 // update BFS tree
        (q,q'):= (q',⟨⟩)                          // switch to next layer
    return parent          // the BFS tree is now {(v,w) : w ∈ V, v = parent(w)}
```

Figure 9.2: Find a spanning tree from a single root node s using breadth first search.

Exercise 9.2 (Graph representation for BFS) Give a more detailed description of BFS. In particular make explicit how to implement it using the adjacency array representation from Section 8.2. Your algorithm should run in time $O(n+m)$.

Exercise 9.3 (Connected components) Explain how to modify BFS so that it computes a spanning forest of an undirected graph in time $O(m+n)$. In addition, your algorithm should select a *representative* node for each connected component of the graph and assign a value *component*$[v]$ to each node that identifies this representative. Hint: Start BFS from each node $s \in V$ but only reset the parent array once in the beginning. Note that isolated nodes are simply connected components of size one.

[example] ⟸

9.2 Depth First Search

If you view breadth first search (BFS) as a careful, conservative strategy for systematic exploration that looks at known things before moving on to unexplored ground [zu neuen ufern??], *depth first search (DFS)* is the exact opposite: Only look back ⟸ if you run out of options. Although this strategy leads to strange looking exploration trees compared to the orderly layers generated by BFS, the combination of eager exploration with the perfect memory of a computer leads to properties of DFS trees that

make them very useful. Figure 9.3 therefore does not just give one algorithm but an algorithm template. By filling in the routines *init*, *root*, *traverse*, and backtrack, we can solve several interesting problems.

```
init
foreach s ∈ V do
    if s is not marked then
        mark s
        root(s)                                   (s)
        recursiveDFS(s,s)

Procedure recursiveDFS(u,v : Node)
    foreach (v,w) ∈ E do
        traverse(v,w)                             (v)→(w)
        if w is not marked then
            mark w
            recursiveDFS(v,w)
    backtrack(u,v)                                // finish v (u)→(v)
```

Figure 9.3: A template for depth first search of a graph $G = (V,E)$.

Exercise 9.4 Give a nonrecursive formulation of DFS. You will need to maintain a stack of unexplored nodes and for each node on the stack you have to keep track of the edges that have already been traversed.

9.2.1 DFS Numbering, Finishing Times, and Topological Sorting

As a warmup let us consider two useful ways of numbering the nodes based on DFS:

```
init dfsPos=1 : 1..n;   finishingTime=1 : 1..n
root dfsNum[s]:= dfsPos++
traverse(v,w) if w is not marked then dfsNum[w]:= dfsPos++      (v)→(w)
backtrack(u,v) finishTime[v]:= finishingTime++                  (u)→(v)
```

dfsNum records the order in which nodes are marked and *finishTime* records the order in which nodes are finished. Both numberings are frequently used to define orderings of the nodes. In this chapter we will define "$\prec$" based on *dfsNum*, i.e., $u \prec v \Leftrightarrow$ *dfsNum*$[u] <$ *dfsNum*$[v]$. Later we will need the following invariant of DFS:

Lemma 9.1 *The nodes on the DFS recursion stack are sorted with respect to* $\prec$.

Finishing times have an even more useful property for directed acyclic graphs:

Lemma 9.2 *If G is a DAG then* $\forall (v,w) \in E$: *finishTime*$[v] >$ *finishTime*$[w]$.

Proof: We consider for each edge $e = (v,w)$ the event that *traverse*(v,w) is called. If w is already finished, v will finish later and hence, *finishTime*$[v] >$ *finishTime*$[w]$. Exercise 9.5 asks you to prove that this case covers forward edges and cross edges. Similarly, if e is a tree edge, *recursiveDFS*$[w]$ will be called immediately and w gets a smaller finishing time than v. Finally, backward edges are illegal for DAGs because together with the tree edges leading from w to v we get a cycle. ∎

An ordering of the nodes in a DAG by decreasing finishing times is known as *topological sorting*. Many problems on DAGs can be solved very efficiently by iterating through the nodes in topological order. For example, in Section 10.3 we will get a very simple algorithm for computing shortest paths in acyclic graphs.[more examples?] ⟸

Exercise 9.5 (Classification of Edges) Prove the following relations between edge types and possible predicates when edge (v,w) is traversed. Explain how to compute each of the predicates in constant time. (You may have to introduce additional flags for each node that are set during the execution of DFS.)

type	w marked?	w finished?	$v \prec w$	w on recursion stack?
tree	no	no	yes	no
forward	yes	yes	yes	no
backward	yes	no	no	yes
cross	yes	yes	no	no

Exercise 9.6 (Topological sorting) Design a DFS based algorithm that outputs the nodes as a sequence in topological order if G is a DAG. Otherwise it should output a cycle.

Exercise 9.7 Design a *BFS* based algorithm for topological sorting.

[rather use markTime, finishTime?] ⟸

Exercise 9.8 (Nesting property of DFS numbers and finishing times) Show that $\nexists u,v \in V$: *dfsNum*$[u] <$ *dfsNum*$[v] <$ *finishTime*$[u] <$ *finishTime*$[v]$

9.2.2 Strongly connected components (SCCs)

We now come back to the problem posed at the beginning of this chapter. Computing connected components of an *un*directed graph is easy. Exercise 9.3 outlines how to do it using BFS and adapting this idea to DFS is equally simple. For *directed* graphs it is not sufficient to find a path from u to v in order to conclude that u and v are in the same SCC. Rather we have to find a directed cycle containing both u and v.

Our approach is to find the SCCs using a single pass of DFS. At any time during DFS, we will maintain a representation of all the SCCs of the subgraph defined by marked nodes and traversed edges. We call such an SCC *open* if it contains any unfinished nodes and *closed* otherwise. We call a node open if it belongs to an open component and closed if it belongs to a closed component. Figure 9.4 gives an example for the development of open and closed SCCs during DFS. DFS is so well suited for computing SCCs because it maintains the following invariants at all times:

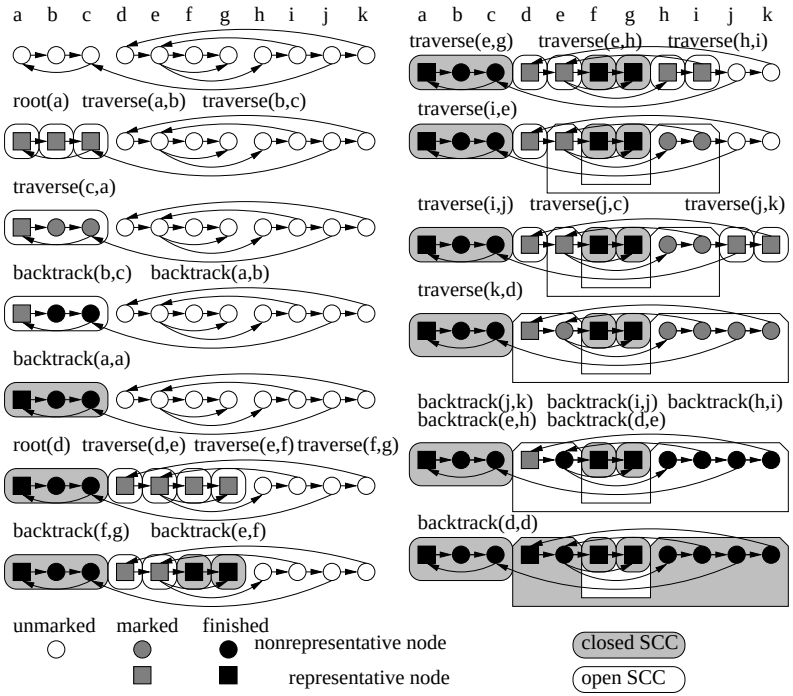


Figure 9.4: An example for the development of open and closed SCCs during DFS.

Lemma 9.3 *DFS maintains the following invariants:*

1. *Closed SCCs are also SCCs of the entire graph.*
2. *Open SCCs, can be arranged into a sequence $\langle S_1, \ldots, S_k \rangle$ such that the following properties hold for $1 \leq i < k$:*
   - *(a) There is a tree edge $(u,v) \in S_i \times S_{i+1}$ such that v is the first node in S_{i+1} that is marked, i.e., open components form a path.*
   - *(b) $\forall u \in S_i, v \in S_{i+1} : u \prec v$.*
   - *(c) When a node v finishes or an edge (v,w) is traversed, v is in S_k.*

Proof: [each paragraph as a minipage side by side with a picture?] First consider Invariant 2(c). By Lemma 9.2, v must be the open node with highest *dfsNum*. Since S_k is not closed, it must contain at least one open node and by Invariant 2(b) these nodes have larger DFS numbers than the open nodes in all other components. ⟸

We prove Invariants 1–2(b) inductively. Whenever DFS marks or finishes a node or traverses an edge, we show that the invariants are maintained. When DFS starts, the claim is vacuously true — no nodes are marked, no edges have been traversed and hence there are neither open nor closed components yet.

Before a new root is marked, all marked nodes are finished and hence there can only be closed components. Therefore, marking a new root s produces the trivial sequence of open components $\langle \{s\} \rangle$.

If a tree edge $e = (v,w)$ is traversed and w is marked, $\{w\}$ is appended to the sequence of open components.

Now suppose an non-tree edge $e = (v,w)$ out of S_k is traversed. If w is closed, e cannot affect the invariants because the component of w is maximal. If w is open, $w \in S_i$ for some $i \leq k$. Now $S_i \cup \ldots \cup S_k$ form a single SCC because together with the tree edges from Invariant 2(a), we get a cycle of components $S_i, \ldots, S_k$.

If the last node in S_k finishes, S_k becomes closed. Invariant 2 is maintained by removing S_k from the sequence of open components. To show that Invariant 1 is also maintained, assume it would be violated. This means that there must be a cycle C that contains nodes from both inside and outside S_k. Consider an edge $e = (v,v')$ on C such that $v \in S_k$ and $v' \notin S_k$.[bild?] Node v' cannot be in a closed component S' because by the induction hypothesis, all previously closed components are maximal whereas C would extend S'. Node v' cannot be unmarked either since DFS would have explored this edge before finishing v. Hence, v' must lie in some open component S_i, $i < k$. But this is impossible because the tree edges from Invariant 2(a) together with e would merge $S_i, \ldots, S_k$ into a single open component. ⟸ ∎

```
init
    component=⟨⊥,...,⊥⟩ : NodeArray of Node      // SCC representatives
    oReps=⟨⟩ : Stack of Node                    // representatives of open SCCs
    oNodes=⟨⟩ : Stack of Node                   // all nodes in open SCCs

root(s)                                                          (s)
    oReps.push(s)
    oNodes.push(s)

traverse(v,w)                                                    (v)→(w)
    if (v,w) is a tree edge then
        oReps.push(w)
        oNodes.push(w)
    else if w ∈ oNodes then                 // Collapse components on cycle
        while w ≼ oReps.top do              // picture???
            oReps.pop

backtrack(u,v)                                                   (u)→(v)
    if v = oReps.top then
        oReps.pop
        repeat
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

Figure 9.5: An instantiation of the DFS template that computes strongly connected components of a graph $G=(V,E)$.

The invariants guaranteed by Lemma 9.3 come "for free" with DFS without any additional implementation measures. All what remains to be done is to design data structures that keep track of open components and allow us to record information on closed components. The first node marked in any open or closed component is made its *representative*. For a node v in a closed component, we record its representative in $component[v]$. This will be our output. Since the sequence of open components only changes at its end, it can be managed using stacks. We maintain a stack *oReps* of representatives of the open components. A second stack *oNodes* stores all the open nodes ordered by $\prec$. By Invariant 2, the sequence of open components will correspond to intervals of nodes in *oNodes* in the same order.

Figure 9.5 gives pseudocode. When a new root is marked or a tree edge is explored, a new single node open component is created by pushing this node on both stacks. When a cycle of open components is created, these components can be merged by popping representatives off *oReps* while the top representative is not left of the node w closing the cycle. Since an SCC S is represented by its node v with smallest *dfsNum*, S is closed when v is finished. In that case, all nodes of S are stored on top of *oNodes*. Operation *backtrack* then pops v from *oReps* and the nodes $w \in S$ from *oNodes* setting their *component* to the representative v.

Note that the test $w \in oNodes$ in *traverse* can be done in constant time by keeping a flag for open nodes that is set when a node is first marked and that is reset when its component is closed. Furthermore, the while and the repeat loop can make at most n iterations during the entire execution of the algorithm since each node is pushed on the stacks exactly once. Hence, the exeuction time of the algorithm is $O(m+n)$. We get the following theorem:

Theorem 9.4 *The DFS based algorithm in Figure 9.5 computes strongly connected components in time $O(m+n)$.*

Exercises

***Exercise 9.9 (transitive closure)** The *transitive closure* $G^* = (V,E^*)$[check notation] of a graph $G=(V,E)$ has an edge $(u,v) \in E^*$ whenever there is a path from u to v in E. Design an algorithm that computes E^* in time $O(n+|E^*|)$. Hint: First solve the problem for the DAG of SCCs of G. Also note that $S \times S' \subseteq E^*$ if S and S' are SCCs connected by an edge. ⟸

Exercise 9.10 (2-edge connected components) Two nodes of an *undirected* graph are in the same 2-edge connected component (2ECC) iff they lie on a cycle. Show that the SCC algorithm from Figure 9.5 computes 2-edge connected components. Hint: first show that DFS of an undirected graph never produces any cross edges.

Exercise 9.11 (biconnected components) Two *edges* of an *undirected* graph are in the same biconnected component (BCC) iff they lie on a *simple* cycle. Design an algorithm that computes biconnected components using a single pass of DFS. You can use an analogous approach as for SCC and 2ECC but you need to make adaptations since BCCs are defined for edges rather than nodes.[explain why important?] ⟸

9.3 Implementation Notes

BFS is usually implemented by keeping unexplored nodes (with depths d and $d+1$) in a FIFO queue. We choose a formulation using two separate sets for nodes at

depth d and nodes at depth $d+1$ mainly because it allows a simple loop invariant that makes correctness immediately evident. However, an efficient implementation of our formulation is also likely to be somewhat more efficient. If q and q' are organized as stacks, we will get less cache faults than for a queue in particular if the nodes of a layer do not quite fit into the cache. Memory management becomes very simple and efficient by allocating just a single array a of n nodes for both stacks q and q'. One grows from $a[1]$ to the right and the other grows from $a[n]$ towards smaller indices. When switching to the next layer, the two memory areas switch their roles.

⟹ [unify marks and dfsnumbers]

⟹ [graph iterators]

C++

⟹ [leda boost graph iterators]

Java

⟹ [todo]

9.4 Further Findings

⟹ [triconnected components, planarity, parallel and external CC edge contraction ear decomposition?]

Chapter 10

Shortest Paths

The shortest or quickest or cheapest path problem is ubiquitous. You solve it all the time when traveling. Give more examples

- the shortest path between two given nodes s and t (single source, single sink)
- the shortest paths from a given node s to all other nodes (single source)
- the shortest paths between any pair of nodes (all pairs problem)

10.1 Introduction

Abstractly, we are given a directed graph $G = (V,E)$ and a cost function c that maps edges to costs. For simplicity, we will assume that edge costs are real numbers, although most of the algorithms presented in this chapter will work under weaker assumptions (see Exercise 10.13); some of the algorithms require edge costs to be integers. We extend the cost function to paths in the natural way. The cost of a path is the sum of the costs of its constituent edges, i.e., if $p = [e_1, e_2, \ldots, e_k]$ then $c(p) = \sum_{1 \le i \le k} c(e_i)$. The empty path has cost zero. For two nodes v and w, we define

todo???: redraw

Figure 10.1: **Single Source Shortest Path Problem:** source node s = fat blue node, yellow node has distance $+\infty$ from s, blue nodes have finite distance from s, square blue node has distance -1 from s. There are paths of length -1, 4, 9, ..., green nodes have distance $-\infty$ from s.

the least cost path distance $\mu(v,w)$ from v to w as the minimal cost of any path for v to w.

$$\mu(v,w) = \inf\{c(p) : p \text{ is a path from } v \text{ to } w\} \in \mathbb{R} \cup \{-\infty, +\infty\} \ .$$

The distance is $+\infty$, if there is no path from v to w, is $-\infty$, if there are paths of arbitrarily small cost[1] and is a proper number otherwise, cf. Figure 10.1 for an example. A path from v to w realizing $\mu(v,w)$ is called a shortest or least cost path from v to w. The following Lemma tells us when shortest paths exist.

[PS: avoid proofs early in the chapters? These properties here we could just state and give similar arguments in a less formal setting.] ⟹

Lemma 10.1 (Properties of Shortest Path Distances)

a) $\mu(s,v) = +\infty$ *iff* v *is not reachable from* s.

b) $\mu(s,v) = -\infty$ *iff* v *is reachable from a negative cycle* C *which in turn is reachable from* s.

c) If $-\infty < \mu(s,v) < +\infty$ *then* $\mu(s,v)$ *is the cost of a simple path from* s *to* v.

Proof: If v is not reachable from s, $\mu(s,v) = +\infty$, and if v is reachable from s, $\mu(s,v) < +\infty$. This proves part a). For part b) assume first that v is reachable from a negative cycle C which is turn is visible from s. Let p be a path from s to some node u on C and let q be a path from u to v. Consider the paths $p^{(i)}$ which first use p to go from s to u, then go around the cycle i times, and finally follow q from u to v. Its cost is $c(p) + i \cdot c(C) + c(q)$ and hence $c(p^{(i+1)}) < c(p^{(i)})$. Thus there are paths of arbitrarily small cost from s to v and hence $\mu(s,v) = -\infty$. This proves part b) in the direction from right to left.

For the direction from left to right, let C be the minimal cost of a simple path from s to v and assume that there is a path p from s to v of cost strictly less than C. Then p is non-simple and hence we can write $p = p_1 \circ p_2 \circ p_3$, where p_2 is a cycle and $p_1 p_3$ is a simple path. Then

$$C \leq c(p_1 p_3) = c(p) - c(p_2) < C$$

and hence $c(p_2) < 0$. Thus v is reachable from a negative cycle which in turn is reachable from s.

We turn to part c). If $-\infty < \mu(s,v) < +\infty$, v is reachable from s, but not reachable through a negative cycle by parts a) and b). Let p be any path from s to v. We decompose p as in preceding paragraph. Then $c(p_1 p_3) = c(p) - c(p_2) \leq c(p)$ since the cost of the cycle p_2 must be non-negative. Thus for every path from s to v there is a simple path from s to v of no larger cost. This proves part c). ∎

[1] $\min\{c(p) : p \text{ is a path from } v \text{ to } w\}$ does does not exist in this situation.

Exercise 10.1 Let p be a shortest path from from u to v for some nodes u and v and let q be a subpath of p. Show that q is a shortest path from its source node to its target node.

Exercise 10.2 Assume that all nodes are reachable from s and that there are no negative cycles. Show that there is an n-node tree T rooted as s such that all tree paths are shortest paths. Hint: Assume first that shortest paths are unique and consider the subgraph T consisting of all shortest paths starting at s. Use the preceding exercise to prove that T is a tree. Extend to the case when shortest paths are not unique.

The natural way to learn about distances is to propagate distance information across edges. If there is a path from s to u of cost $d(u)$ and $e = (u,v)$ is an edge out of u, then there is a path from s to v of cost $d(u) + c(e)$. If this cost is smaller than the best cost previously known, we remember that the currently best way to reach v is through e. Remembering the last edges of shortest paths will allow us to trace shortest paths.

More precisely, we maintain for every node v a label [PS: In a real implementation you store a predecessor node rather than an edge. Note this somewhere?] $\langle d(v), in(v)\rangle$ where $d(v)$ is the cost of the currently best path from s to v and $in(v)$ is the last edge of this path. We call $d(v)$ the *tentative distance* of v. If no path from s to v is known yet, $d(v) = \infty$ and $in(v)$ has the special value $\perp$. If $p(v)$ is the empty path (this is only possible for $v = s$), $d(v) = 0$, and $in(v) = \perp$. ⟸

A function $relax(e : Edge)$ is used to propate distance information.

Procedure $relax(e = (u,v) : Edge)$
if $d(u) + c(e) < d(v)$ set the label of v to $\langle d(u) + c(e), e\rangle$

At the beginning of a shortest path computation we know very little. There is a path of length zero from s to s and no other paths are known.

Initialization of Single Source Shortest Path Calculation:
$\langle d(s), in(s)\rangle := \langle 0, , \perp\rangle$
$\langle d(v), in(v)\rangle := \langle +\infty, \perp\rangle$ *for* $v \neq s$

Once the node labels are initialized we propagate distance informations.

Generic Single Source Algorithm
initialize as described above
relax edges until $d(v) = \mu(s,v)$ for all v

The following Lemma gives sufficient conditions for the termination condition to hold. We will later make the conditions algorithmic.

Lemma 10.2 (Sufficient Condition for Correctness) *We have $d(v) = \mu(s,v)$ if for some shortest path $p = [e_1, e_2, \ldots, e_k]$ from s to v there are times $t_1, \ldots, t_k$ such that $t_1 < t_2 < \ldots < t_k$ and e_i is relaxed at time t_i.*

Proof: We have $\mu(s,v) = \sum_{j=1}^{k} c(e_j)$. Let $t_0 = 0$, let $v_0 = s$, and let $v_i = target(e_i)$. Then $d(v_i) \leq \sum_{1 \leq j \leq i} c(e_j)$ after time t_i. This is clear for $i = 0$ since $d(s)$ is initialized to zero and d-values are only decreased. After the relaxation of e_i at time t_i for $i > 0$, we have $d(v_i) \leq d(v_{i-1}) + c(e_i) \leq \sum_{j=1}^{i} c(e_j)$. ∎

The Lemma above paves the way for specific shortest path algorithms which we discuss in subsequent sections. Before doing so, we discuss properties of graph defined by the in-edges. The set $\{in(v) : in(v) \neq \perp\}$ of in-edges form a graph on our node set with maximal indegree one; we call it the *in-graph*. The in-graph changes over time. Unreached nodes and s (except if it lies on a negative cycle) have indegree zero. Thus the in-graph consists of a tree rooted at s, isolated nodes, cycles and trees emanating from these cycles, cf. Figure 10.1. The tree rooted at s may be empty. We call the tree rooted at s the shortest path tree and use T to denote it. The next lemma justifies the name.

[PS: Hier wird es etwas schwierig. Bilder? Ausfuehlicher formulieren? Dijkstra und Bellmann-Ford ohne negative cycles kann man einfacher haben. Aber dann wird es weniger elegant. . . .]

Lemma 10.3 (Properties of In-Graph)

*a) If p is a path of in-edges from u to v then $d(v) \geq d(u) + c(p)$. [*needed anywhere outside the proof?*]*

*b) If v lies on a cycle or is reachable from a cycle of the in-graph, $\mu(s,v) = -\infty$.[*already follows from the invariant*]*

c) If $-\infty < \mu(s,v) < +\infty$ and $d(v) = \mu(s,v)$, the tree path from s to v is a shortest path from s to v.

Proof: for part a) let $p = v_1, v_2, v_3, \ldots, v_k$ with $e_i = (v_{i-1}, v_i) = in(v_i)$ for $2 \leq i \leq k$. After e_i, $2 \leq i \leq k$, was added to the in-graph, we have $d(v_i) \geq d(v_{i-1}) + c(e_i)$. We had equality, when $in(v_i)$ was set to e_i and $d(v_i)$ has not decreased since (because otherwise $in(v_i)$ would have been set to a different edge[what if it changed back and forth?]). On the other hand, $d(v_{i-1})$ may have decreased. Putting the inequalities together, we have

$$d(v_k) \geq d(v_1) + \sum_{i=2}^{k} c(e_i) = d(v_1) + c(p) .$$

For part b), let $C = p \circ e$[where is $\circ$ introduced?] with $e = (v,u)$ be a cycle of in-edges where e is the edge in the cycle which was added to the in-graph last; p is a path from u to v. Just before e was added, we had $d(v) \geq d(u) + c(p)$ by part a). Since e was added, we had $d(u) > d(v) + c(e)$ at this point of time. Combining both inequalities we obtain

$$d(u) > d(u) + c(e) + c(p) \quad \text{and hence} \quad c(C) < 0 .$$

For part c) consider any node v with $d(v) = \mu(s,v) < +\infty$. Then v has an in-edge and v is not reachable from a cycle of in-edges and hence $\mu(s,s) = d(s) = 0$ and v belongs to T. Let p be the tree path from s to v. Then $\mu(s,v) = d(v) \geq d(s) + c(p) = c(p)$ and hence $c(p) = \mu(s,v)$. We conclude that p is a shortest path from s to v. ∎

We are now ready for specializations of the generic algorithm.

10.2 Arbitrary Edge Costs (Bellman-Ford Algorithm)

It is easy to guarantee the sufficient condition of Lemma 10.2. We simply perform $n-1$ rounds and in each round we relax all edges.

Bellman-Ford Algorithm
initialize node labels
for $i := 1$ **to** $n-1$ **do**
  forall *edges* $e = (u,v) \in E$ **do** *relax*(e)
set $d(x)$ to $-\infty$ for all x that can be reached from an edge $e = (u,v)$ with $d(u) + c(e) < d(v)$

Theorem 10.4 *The Bellman-Ford algorithm solves the shortest path problem in time $O(nm)$.*

Proof: The running time is clearly $O(nm)$ since we iterate $n-1$ times over all edges. We come to correctness.

If $\mu(s,v) > -\infty$ then $\mu(s,v) = d(v)$ after termination of the do-loop. This follows from Lemma 10.2 and the fact that a shortest path consists of at most $n-1$ edges.

If $d(v)$ is not set to $-\infty$, we have $d(x) + c(e) \geq d(y)$ for any edge $e = (x,y)$ on any path p from s to p. Thus $d(s) + c(p) \geq d(v)$ for any path p from s to v and hence $d(v) \leq \mu(s,v)$. Thus $d(v) = \mu(s,v)$.

If $d(v)$ is set to $-\infty$, there is an edge $e = (x,y)$ with $d(x) + c(e) < d(y)$ after termination of the do-loop and such that v is reachable from y. The edge allows us to decrease $d(y)$ further and hence $d(y) > \mu(y)$ when the do-loop terminates. Thus $\mu(y) = -\infty$ by the second paragraph; the same is true for $\mu(s,v)$, since v is reachable from y. ∎

Exercise 10.3 Consider a round and assume that no node label changes in the round. Show that tentative shortest path distance are actual shortest path distances.

Exercise 10.4 Call a node *hot* at the beginning of a round if its tentative distance label was changed in the preceding round. Only s is hot at the beginning of the first round. Show that it suffices to relax the edges out of hot nodes.

Exercise 10.5 Design a network without negative cycles such that the refined version of the Bellman-Ford algorithm outlined in the preceding exercise takes time $\Omega(nm)$.

We will see much faster algorithms for acyclic graphs and graphs with non-negative edge weights in the next sections.

10.3 Acyclic Graphs

Let G be an acyclic graph and let $v_1, v_2, \ldots, v_n$ be an ordering of the nodes such that $(v_i, v_j) \in E$ implies $i \leq j$. A topological order can be computed in time $O(n+m)$ using depth-first search (cf. Section **??**). Lemma 10.2 tells us that if we first relax the edges out of v_1, then the edges out of v_2, ..., then shortest path distances will be computed correctly.

Single Source Shortest Path Algorithm for Acyclic Graphs:
intialize node labels
for $i := 1$ **to** n **do** relax all edges out of v_i // in increasing order

[PS: another place where we can simply state the result without deterring theorems and proofs.] ⟹

Theorem 10.5 *Shortest paths in acyclic graphs can be computed in time $O(n+m)$.*

Proof: It takes time $O(n+m)$ to compute a topological ordering. The algorithm iterates over all nodes and for each node over the edges out of the node. Thus its running time is $O(n+m)$. Correctnes follows immediately from Lemma 10.2. ∎

10.4 Non-Negative Edge Costs (Dijkstra's Algorithm)

We assume that all edge costs are non-negative. Thus there are no negative cycles and shortest paths exist for all nodes reachable from s. We will show that if the edges are relaxed in a judicious order, every edge needs to be relaxed only once.

What is the right order. Along any shortest path, the shortest path distances increase (more precisely, do not decrease). This suggests to scan nodes (to scan a node means to relax all edges out of the node) in order of increasing shortest path distance. Of course, in the algorithm we do not know shortest path distances, we only know tentative distances. Fortunately, it can be shown that for the unscanned node with minimal tentative distance, the true distance and tentative distance agree. This leads to the following algorithm.

Dijkstra's Algorithm
initalize node labels and declare all nodes unscanned
while $\exists$ unscanned node with tentative distance $\leq +\infty$ **do**
  u := the unscanned node with minimal tentative distance
  relax all edges out of u and declare u scanned

Theorem 10.6 *Dijkstra's algorithm solves the single source shortest path for graphs with non-negative edge costs.*

[PS: a picture here?] ⟸

Proof: Assume that the algorithm is incorrect and consider the first time that we scan a node with its tentative distance larger than its shortest path distance. Say at time t we scan node v with $\mu(s,v) < d(v)$. Let $p = [s = v_1, v_2, \ldots, v_k = v]$ be a shortest path from s to v and let i be minimal such that v_i is unscanned just before time t. Then $i > 0$ since s is the first node scanned (in the first iterations s is the only node whose tentative distance is less than $+\infty$) and since $\mu(s,s) = 0 = d(s)$ when s is scanned. Thus v_{i-1} was scanned before time t and hence $d(v_{i-1}) = \mu(s,v_{i-1})$ when v_{i-1} was scanned by definition of t. When v_{i-1} was scanned, $d(v_i)$ was set to $\mu(s,v_i)$ since any prefix of a shortest path is a shortest path. Thus $d(v_i) = \mu(s,v_i) \leq \mu(s,v_k) < d(v_k)$ just before time t and hence v_i is scanned instead of v_k, a contradiction. ∎

Exercise 10.6 Let $v_1, v_2, \ldots$ be the order in which nodes are scanned. Show $\mu(s,v_1) \leq \mu(s,v_2) \leq \ldots$, i.e., nodes are scanned in order of increasing shortest path distances.

We come to the implementation of Dijkstra's algorithm. The key operation is to find the unscanned node with minimum tentative distance value. Adressable priority queues (see Section **??**) are the appropriate data structure. We store all unscanned reached (= tentative distance less than $+\infty$) nodes in an addressable priority queue PQ. The entries in PQ are pairs $(d(u),u)$ with $d(u)$ being the priority. Every reached unscanned node stores a handle to its entry in the priority queue. We obtain the following implementation of Dijkstra's algorithm.

⟹ [todo: Dijkstra und Prim aehnlicher machen.]

```
PQ : PriorityQueue of Node                                   // Init
intialize the node labels          // this sets d(s) = 0 and d(v) = ∞ for v ≠ s
declare all nodes unscanned        // s is the only reached unscanned node at this point
PQ.insert(s,0)
while PQ≠ ∅ do
    select u ∈ PQ with d(u) minimal and remove it; declare u scanned   // delete_min
    forall edges e = (u,v) do
        if D = d(u) + c(e) < d(v) then
            if d(v) == ∞ then PQ.insert(v,D)                          // insert
            else PQ.decreaseKey(v,D)                                  // decreaseKey
            set the label of v to ⟨D,e⟩
```

Figure 10.2: Implementation of Dijkstra's Algorithm.

We need to say a few more words about the use of the priority queue. The insert operations create a entry for a particular node in the priority queue. A handle to this entry is stored with the node. The handle is used in the decrease priority operation to access the entry corresponding to a node. The *deleteMin* operation returns the pair $(d(u),u)$ with $d(u)$ minimal. The second component of the pair tells us the node.

Theorem 10.7 *Dijkstra's algorithm solves the single source shortest path problem in graphs with non-negative edge weights in time*

$$O(n+m+T_{init}+n\cdot(T_{isEmpty}+T_{delete\_min}+T_{insert})+m\cdot T_{decreaseKey})$$

⟹ [PS: move proof as explaining text before the theorem to avoid a deterring proof?]

Proof: Every reachable node is removed from the priority queue exactly once and hence we consider each edge at most once in the body of the while loop. We conclude that the running time is $O(n+m)$ plus the time spent on the operations on the priority queue. The queue needs to be initialized. Every node is inserted into the queue and deleted from the queue at most once and we perform one emptyness test in each iteration of the while-loop. The number of decrease priority operation is at most $m-(n-1)$: for every node $v \neq s$ we have at most $indeg(v)-1$ decrease priority operations and for s we have none. ∎

Exercise 10.7 Design a graph and and a non-negative cost function such that the relaxation of $m-(n-1)$ edges causes a *decreaseKey* operation.

In his original paper [32] Dijkstra proposed the following implementation of the priority queue. He proposed to maintain the number of reached unscanned nodes and two arrays indexed by nodes: an array d storing the tentative distances and an array storing for each node whether it is unscanned and reached. Then *init* is $O(n)$ and *is_empty*, *insert* and *decreaseKey* are $O(1)$. A *delete_min* takes time $O(n)$ since it requires to scan the arrays in order to find the minimum tentative distance of any reached unscanned node. Thus total running time is $O(m+n^2)$.

Theorem 10.8 *With Dijkstra's proposed implementation of the priority queue, Dijkstra's algorithm runs in time $O(m+n^2)$.*

Much better priority queue implementations were invented since Dijkstra's original paper, cf. the section of adressable priority queues (Section **??**).

Theorem 10.9 *With the heap implementation of priority queues, Dijkstra's algorithm runs in time $O(m\log n+n\log n)$.*

Theorem 10.10 *With the Fibonacci heap implementation of priority queues, Dijkstra's algorithm runs in time $O(m+n\log n)$.*

Asymptotically, the Fibonnacci heap implementation is superior except for sparse graphs with $m=O(n)$. In practice (see [22, 67]), Fibonacci heaps are usually not the fastest implementation because they involve larger constant factors and since the actual number of decrease priority operations tends to much smaller than what the worst case predicts. An average case analysis [77] sheds some light on this.

Theorem 10.11 *Let G be an arbitrary directed graph, let s be an arbitrary node of G, and for each node v let $C(v)$ be a set of non-negative real numbers of cardinality*

indeg(v). *For each v the assignment of the costs in $C(v)$ to the edges into v is made at random, i.e., our probability space consists of the $\prod_v$ indeg(v)! many possible assignments of edge costs to edges. Then the expected number of decreaseKey operations is* $O(n\log(m/n))$.

⟹ [PS: todo similar thing for analysis of Prim.]

Proof: Consider a particular node v and let $k = indeg(v)$. Let $e_1, \ldots, e_k$ be the order in which the edges into v are relaxed in a particular run of Dijkstra' algorithm and let $u_i = source(e_i)$. Then $d(u_1) \le d(u_2) \le \ldots \le d(u_k)$ since nodes are removed from U in increasing order of tentative distances. Edge e_i causes a *decreaseKey* operation iff

$$i \ge 2 \quad \text{and} \quad d(u_i) + c(e_i) < \min\left\{d(u_j) + c(e_j) : j < i\right\} .$$

Thus the number of operations *decreaseKey*$(v,\cdot)$ is bounded by the number of i such that

$$i \ge 2 \quad \text{and} \quad c(e_i) < \min\left\{c(e_j) : j < i\right\} .$$

Since the order in which the edges into v are relaxed is independent of the costs assigned to them, the expected number of such i is simply the number of left-right maxima in a permutation of size k (minus 1 since $i = 1$ is not considered). Thus the expected number is $H_k - 1$ by Theorem **??** and hence the expected number of decrease priority operations is bounded by [KM: CHECK first $\le$.] ⟹

$$\sum_v H_{indeg(v)} - 1 \le \sum_v \ln indeg(v) \le n\ln(m/n)) ,$$

where the last inequality follows from the concavity of the ln-function (see Appendix **??**). ∎

We conclude that the expected running time is $O(m + n\log(m/n)\log n)$ with the heap implementation of priority queues. This is asymptotically more than $O(m + n\log n)$ only for $m = \Omega(1)$ and $m = o(n\log n\,\mathrm{loglog}\,n)$.

Exercise 10.8 When is $n\log(m/n)\log n = O(m + n\log n)$? Hint: Let $m = nd$. Then the question is equivalent to $\log d \log n = O(d + \log n)$.

10.5 Monotone Integer Priority Queues

Dijkstra's algorithm does not really need a general purpose priority queue. It only requires what is known as a monotone priority queue. The usage of a priority queue is monotone if any insertion or decrease priority operation inserts a priority at least as large as the priority returned by the last *deleteMin* operation (at least as large as the first insertion for the operations preceding the first *deleteMin*). Dijkstra's algorithm uses its queue in a monotone way.

It is not known whether monotonicity of use can be exploited in the case of general edge costs. However, for integer edge costs significant savings are possible. We therefore assume for this section that edges costs are integers in the range $[0..C]$ for some integer C. C is assumed to be known at initialization time of the queue.

Since a shortest path can consist of at most $n-1$ edges, shortest path distances are at most $(n-1)C$. The range of values in the queue at any one time is even smaller. Let *min* be the last value deleted from the queue (zero before the first deletion). Then all values in the queue are contained in $[min\,..\,min + C]$. This is easily shown by induction on the number of queue operations. It is certainly true after the first insertion. A *deleteMin* does not decrease *min* and inserts and decrease priority operations insert priorities in $[min\,..\,min + C]$.

10.5.1 Bucket Queues

A bucket queue [30] is an array B of $C+1$ linear lists. A node $v \in U$ with tentative distance $d(v)$ is stored in the list $B[d(v) \bmod C+1]$. Since the priorities in the queue are contained in $[min\,..\,min + C]$ at any one time, all nodes in a bucket have the *same* distance value. Every node keeps a handle to the list item representing it. Initialization amounts to creating $C+1$ empty lists, an *insert*$(v, d(v))$ inserts v into the appropriate list, a *decreaseKey*$(v, d(v))$ removes v from the list containing it and inserts it into the appropriate bucket. Thus *insert* and *decreaseKey* take constant time.

A *deleteMin* first looks at bucket $B[min \bmod C+1]$. If this bucket is empty, it increments *min* and repeats. In this way the total cost of all *deleteMin* operations is $O(n + nC)$ since *min* is incremented at most nC times and since at most n elements are deleted from the queue.

Theorem 10.12 *With the bucket queue implementation of priority queues, Dijkstra's algorithm runs in time $O(m + nC)$. This assumes that edge costs are integers in the range $[0..C]$.*

[PS: Exercise with Dinitz refinement (bucket width= min edge weight)?] ⟸

10.5.2 Radix Heaps

Radix heaps [3] improve upon the bucket queue implementation by using buckets of different granularity. Fine grained buckets are used for tentative distances close to *min* and coarse grained buckets are used for tentative distances far away from *min*. The details are as follows.

TODO

Figure 10.3: The path represents the binary representation of *min* with the least significant digit on the right. A node $v \in U$ is stored in bucket B_i if its binary representation differs in the i-th bit when binary representations are scanned starting at the most significant bit. Distinguishing indices i with $i \geq K$ are lumped together.

Radix heaps exploit the binary representation of tentative distances. For numbers a and b with binary representations $a = \sum_{i\geq 0} \alpha_i 2^i$ and $b = \sum_{i\geq 0} \beta_i 2^i$ define the most significant distinguishing index $msd(a,b)$ as the largest i with $\alpha_i \neq \beta_i$ and let it be -1 if $a = b$. If $a < b$ then a has a zero bit in position $i = msd(a,b)$ and b has a one bit. A radix heap consists of a sequence of buckets $B_{-1}, B_0, \ldots, B_K$ where $K = 1 + \lfloor \log C \rfloor$. A node $v \in U$ is stored in bucket B_i where $i = \min(msd(min, d(v)), K)$. Buckets are organized as doubly linked lists and every node keeps a handle to the list item representing it. Figure 10.3 illustrates this definition. We assume that most distinguishing indices can be computed[2] in time $O(1)$ and justify this assumption in Exercise 10.10.

Exercise 10.9 There is another way to describe the distribution of nodes over buckets. Let $min = \sum_j \mu_j 2^j$, let i_0 be the smallest index greater than K with $\mu_{i_0} = 0$, and let $M_i = \sum_{j>i} \mu_j 2^j$. B_{-1} contains all nodes $v \in U$ with $d(v) = min$, for $0 \leq i < K$, $B_i = \emptyset$ if $\mu_i = 1$, and $B_i = \left\{ v \in U : M_i + 2^i \leq d(x) < M_i + 2^{i+1} - 1 \right\}$ if $\mu_i = 0$, and $B_K = \left\{ v \in U : M_{i_0} + 2^{i_0} \leq d(x) \right\}$. Prove that this description is correct.

We turn to the realization of the queue operations. Initialization amounts to creating $K+1$ empty lists, an *insert*$(v, d(v))$ inserts v into the appropriate list, a *decreaseKey*$(v, d(v))$ removes v from the list containing it and inserts it into the appropriate queue. Thus *insert* and *decreaseKey* take constant time.

A *deleteMin* first finds the minimum i such that B_i is non-empty. If $i = -1$, an arbitrary element in B_{-1} is removed and returned. If $i \geq 0$, the bucket B_i is scanned and *min* is set to the smallest tentative distance contained in the bucket. Afterwards, all elements in B_i are moved to the appropriate new bucket. Thus a *deleteMin* takes constant time if $i = -1$ and takes time $O(i + |B_i|) = O(K + |B_i|)$ if $i \geq 0$. The crucial observation is now that every node in bucket B_i is moved to a bucket with smaller index.

$\Longrightarrow$ [somewhere show that the other buckets need not be touched]

[2] For the built-in type *int* it is a machine instruction on many architectures.

Lemma 10.13 *Let i be minimal such that B_i is non-empty and assume $i \geq 1$. Let min be the smallest element in B_i. Then $msd(min, x) < i$ for all $x \in B_i$.*

Proof: We distinguish the cases $i < K$ and $i = K$. Let min' be the old value of *min*. If $i < K$, the most significant distinguishing index of *min*' and any $x \in B_i$ is i, i.e., min' has a zero in bit position i and all $x \in B_i$ have a one in bit position i. They agree in all positions with index larger than i. Thus the most signifant distinguishing index for *min* and x is smaller than i.

Let us next assume that $i = K$ and consider any $x \in B_K$. Then $min' < min \leq x \leq min' + C$. Let $j = msd(min', min)$ and $h = msd(min, x)$. Then $j \geq K$. We want to show that $h < K$. Observe first that $h \neq j$ since *min* has a one bit in position j and a zero bit in position h. Let $min' = \sum_l \mu_l 2^l$.

Assume first that $h < j$ and let $A = \sum_{l>j} \mu_l 2^l$. Then $min' \leq A + \sum_{l<j} 2^l \leq A + 2^j - 1$ since the j-th bit of min' is zero. On the other hand, x has a one bit in positions j and h and hence $x \geq A + 2^j + 2^h$. Thus $2^h \leq C$ and hence $h \leq \lfloor \log C \rfloor < K$.

Assume next that $h > j$ and let $A = \sum_{l>h} \mu_l 2^l$. We will derive a contradiction. min' has a zero bit in positions h and j and hence $min' \leq A + 2^h - 1 - 2^j$. On the other hand, x has a one bit in position h and hence $x \geq A + 2^h$. Thus $x - min' > 2^j \geq 2^K \geq C$, a contradiction. ∎

Lemma 10.13 allows us to account for the cost of a *deleteMin* as follows: We charge the time for the search for i to the operation itself (charge $O(K)$) and charge $O(1)$ to each node in B_i. In this way the cost of the operation is covered. Also, since each node in B_i is moved to a lower numbered bucket and since nodes start in bucket B_K and end in bucket B_{-1}, the total charge to any node is at most $2 + K = O(\log C)$. We conclude that the total cost of the n *deleteMin* operations is $O(nK + nK) = O(n \log C)$ and have thus proved:

Theorem 10.14 *With the Radix heap implementation of priority queues, Dijkstra's algorithm runs in time $O(m + n \log C)$. This assumes that edge costs are integers in the range $[0..C]$.*

Exercise 10.10 [PS: ich verstehe die Aufgabe im Moment nicht. Ich dachte Aufg. 10.9 ist eine *andere* Darstellung.] The purpose of this exercise is to show $\Longleftarrow$ that the assumption that the *msd*-function can be computed in amortized constant time is warranted. We assume inductively, that we have the binary representation of *min* and the description of bucket ranges given in Exercise 10.9 available to us. When we need to move a node from bucket i to a smaller bucket we simply scan through buckets B_{i-1}, B_{i-2} until we find the bucket where to put the node. When *min* is increased, we compute the binary representation of *min* from the binary representation of the old

minimum min' by adding $min - min'$ to the old minimum. This takes amortized time $O(K+1)$ by Theorem **??**.

Exercise 10.11 Radix heaps can also be based on number representations with base b for any $b \geq 2$. In this situation we have buckets $B_{i,j}$ for $i = -1, 0, 1, \ldots K$ and $0 \leq j \leq b$, where $K = 1 + \lfloor \log C / \log b \rfloor$. An unscanned reached node x is stored in bucket $B_{i,j}$ if $msd(min, d(x)) = i$ and the i-th digit of $d(x)$ is equal to j. We also store for each i, the number of nodes contained in buckets $\cup_j B_{i,j}$. Discuss the implementation of the priority queue operations and show that a shortest path algorithm with running time $O(m + n(b + \log C / log b))$ results. What is the optimal choice of b?

If the edge costs are random integers in the range $[0..C]$, a small change of the algorithm guarantees linear running time [**?**, 38, **?**]. For every node v let $min\_in\_cost(v)$ the minimum cost of an incoming edge. We divide U into two parts, a part F which contains nodes whose tentative distance label is known to be equal to their exact distance from s, and a part B which contains all other labeled nodes. B is organized as a radix heap. We also maintain a value min. We scan nodes as follows.

When F is non-empty, an arbitrary node in F is removed and the outgoing edges are relaxed. When F is empty, the minimum node is selected from B and min is set to its distance label. When a node is selected from B, the nodes in the first non-empty bucket B_i are redistributed if $i \geq 0$. There is a small change in the redistribution process. When a node v is to be moved, and $d(v) \leq min + min\_in\_cost(v)$, we move v to F. Observe that any future relaxation of an edge into v cannot decrease $d(v)$ and hence $d(v)$ is know to be exact at this point.

The algorithm is correct since it is still true that $d(v) = \mu(s, v)$ when v is scanned. For nodes removed from F this was argued in the previous paragraph and for nodes removed B this follows from the fact that they have the smallest tentative distance among all unscanned reached nodes.

Theorem 10.15 *Let G be an arbitrary graph and let c be a random function from E to $[0..C]$. Then the single source shortest path problem can be solved in expected time $O(n+m)$.*

Proof: We still need to argue the bound on the running time. As before, nodes start out in B_K. When a node v is moved to a new bucket, but not yet to F, $d(v) > min + min\_in\_cost(v)$ and hence v is moved to a bucket B_i with $i \geq \log min\_in\_cost(v)$. We conclude that the total charge to nodes in *deleteMin* and *decreaseKey* operations is

$$\sum_v (K - \log min\_in\_cost(v) + 1) \ .$$

Next observe that $mincost(v)$ is the minimum over $c(e)$ of the edges into v and hence

$$\sum_v (K - \log min\_in\_cost(v) + 1) \leq n + \sum_e (K - \log c(e)) \ .$$

$K - \log c(e)$ is the number of leading zeros in the binary representation of $c(e)$ when written as a K-bit number. Our edge costs are uniform random numbers in $[0..C]$ and $K = 1 + \lfloor \log C \rfloor$. Thus $\text{prob}(k - \log c(e)) = i) = 2^{-i}$. We conclude

$$\mathrm{E}[\sum_e (k - \log c(e))] = \sum_e \sum_{i \geq 0} i 2^{-i} = O(m)$$

We conclude that the total expected cost of *deleteMin* and $decrease_p$ operations is $O(n+m)$. The time spent outside these operations is also $O(n+m)$. ∎

10.6 All Pairs Shortest Paths and Potential Functions

The all-pairs problem is tantamount to n single source problems and hence can be solved in time $O(n^2 m)$. A considerable improvement is possible. We show that is suffices to solve one general single source problem plus n single source problems with nonnegative edge costs. In this way, we obtain a running time of $O(nm + n(m + n \log n)) = O(nm + n^2 \log n)$. We need the concept of a potential function.

A *node potential* assigns a number $pot(v)$ to each node v. For an edge $e = (v, w)$ define its *reduced cost* as:

$$\bar{c}(e) = pot(v) + c(e) - pot(w) \ .$$

Lemma 10.16 *Let p and q be paths from v to w. Then $\bar{c}(p) = pot(v) + c(p) - pot(w)$ and $\bar{c}(p) \leq \bar{c}(q)$ iff $c(p) \leq c(q)$. In particular, shortest paths with respect to $\bar{c}$ are the same as with respect to c.*

Proof: The second and the third claim follow from the first. For the first claim, let $p = [e_0, \ldots, e_{k-1}]$ with $e_i = (v_i, v_{i+1})$, $v = v_0$ and $w = v_k$. Then

$$\begin{aligned} \bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(e) = \sum_{0 \leq i < k} (pot(v_i) + c(e_i) - pot(v_{i+1})) \\ &= pot(v_0) + \sum_{0 \leq i < k} c(e_i) - pot(v_k) \\ &= pot(v_0) + c(p) - pot(v_k) \end{aligned}$$

∎

Exercise 10.12 Potential functions can be used to generate graphs with negative edge costs but no negative cycles: generate a (random) graph, assign to every edge e a (random) non-negative (!!!) weight $c(e)$, assign to every node v a (random) potential $pot(v)$, and set the cost of $e=(u,v)$ to $c(e)=pot(u)+c(e)-pot(v)$. Show that this rule does not generate negative cycles. Hint: The cost of a cycle with respect to c is the same as with respect to $\bar{c}$.

Lemma 10.17 *Assume that G has no negative cycles and that all nodes can be reached from s. Let $pot(v)=\mu(s,v)$ for $v\in V$. With this potential function reduced edge costs are non-negative.*

Proof: Since all nodes are reachable from s and since there are no negative cycles, $\mu(s,v)\in\mathbb{R}$ for all v. Thus the reduced costs are well defined. Consider an arbitrary edge $e=(v,w)$. We have $\mu(s,v)+c(e)\geq\mu(w)$ and hence $\bar{c}(e)=\mu(s,v)+c(e)-\mu(s,w)\geq 0$. ∎

All Pair Shortest Paths in the Absence of Negative Cycles

```
add a new node s and zero length edges (s,v) for all v   // no new cycles, time O(m)
compute μ(s,v) for all v with Bellman-Ford               // time O(nm)
set pot(v) = μ(s,v) and compute reduced costs            // time O(m)
forall nodes x do                                        // time O(n(m+n log n))
    solve the single source problem with source x and
    reduced edge costs with Dijkstra's algorithm
translate distances back to original cost function       // time O(m)
μ(v,w) = μ̄(v,w) + pot(w) − pot(v)
```

We have thus shown

Theorem 10.18 *Assume that G has no negative cycles. The all pairs shortest problem in graphs without negative cycles can be solved in time $O(nm)$[PS: $+n^2\log n$???].*

⟹

The assumption that G has no negative cycles can be removed [69].

⟹ [PS: what about heuristics like A^* in geometric graphs and road graphs.?]

10.7 Implementation Notes

Shortest path algorithms work over the set of extended reals $\mathbb{R}\cup\{+\infty,-\infty\}$. We may ignore $-\infty$ since it is only needed in the presence of negative cycles and even there it is only needed for the output, see Section **??**. We can also get rid of $+\infty$ by noting that

$in(v)=\perp$ iff $d(v)=+\infty$, i.e., when $in(v)=\perp$, we $d(v)=+\infty$ and ignore the number stored in $d(v)$.

[PS: More implementation notes: heuristics? store PQ items with nodes? Implementations in LEDA? BOOST?] ⟸

10.8 Further Findings

Exercise 10.13 A ordered semi-group is a set S together with an associative and commutative operation $+$, a neutral element 0, and a linear ordering $\leq$ such that for all x, y, and z: $x\leq y$ implies $x+z\leq y+z$. Which of the algorithms of this section work for ordered semi-groups? Which work under the additional assumption that $0\leq x$ for all x?

Chapter 11

Minimum Spanning Trees

??? nettes Bild The atoll Taka-Tuka-Land in the south seas asks you for help. They want to connect their islands by ferry lines. Since there is only little money available, the sum of the lengths of the connections openened should be minimal as long as it is possible to travel between any two islands even if you have to change ships several times.

More generally, we want to solve the following problem: Consider a connected[1] undirected graph $G = (V,E)$ with positive edge weights $c : E \to \mathbb{R}_+$. A *minimum spanning tree (MST)* of G is defined by a set $T \subseteq E$ of edges such that the graph (V,T) is connected and $c(T) := \sum_{e \in T} c(e)$ is minimized. It is not difficult to see that T forms a tree[2] and hence contains $n-1$ edges.[wo erklaert?] In our example, we have a ⟸ complete graph of the islands and the edge weights are the pairwise distances between the islands (say between their post offices).

Minumum spanning trees (MSTs) are perhaps the simplest variant of an important family of problems known as *network design problems*. Because MSTs are such a simple concept, they also show up in many seemingly unrelated problems such as clustering, finding paths that minimize the maximum edge weight used, or finding approximations for harder problems. Section 11.6 has more on that. An equally good reason to discuss MSTs in an algorithms text book is that there are simple, elegant, and fast algorithms to find them. The algorithms we discuss are greedy algorithms based on a simple property of MST edges introduced in Section 11.1. The Jarník-Prim algorithm from Section 11.2 applies this property to grow MSTs starting from some starting node. Kruskal's algorithm from Section 11.3 grows the tree by merging

[1] If G is not connected, we may ask for a *minimum spanning forest* —a set of edges that defi nes an MST for each connected component of G.

[2] In this chapter we often identify a set of edges T with a subgraph of (V,T).

small subtrees. This algorithm applies a generally useful data structure explained in Section 11.4: Maintain a partition of a set of elements. Operations are a to find out whether two elements are in the same subset and to join two subsets.

Exercises

Exercise 11.1 Develop an efficient way to find minimum spanning forests using a single call of a minimum spanning tree routine. Do not find connected components first. Hint: insert $n-1$ additional edges.

Exercise 11.2 Explain how to find minimum spanning sets of edges when zero and negative weights are allowed. Do these edge sets necessarily form trees?

Exercise 11.3 Explain how to reduce the problem of finding *maximum* weight spanning trees to the minimum spanning tree problem.

11.1 Selecting and Discarding MST Edges

All known algorithms for computing minimum spanning trees are based on the following two complementary properties.

Lemma 11.1 (Cut[3] Property:) *Consider a proper subset S of V and an edge $e \in \{(s,t) : (s,t) \in E, s \in S, t \in V \setminus S\}$ with minimal weight. Then there is an MST T of G that contains e.*

$\Longrightarrow$ [picture: $S, V-T, e, e', T, T'$]

Proof: Consider any MST T' of G. Since T' is a tree, T' contains a unique edge $e' \in T'$ connecting a node from S with a node from $V \setminus S$. Furthermore, $T' \setminus \{e'\}$ defines a spanning trees for S and $V \setminus S$ and hence $T = (T' \setminus \{e'\}) \cup \{e\}$ defines a spanning tree. By our assumption, $c(e) \leq c(e')$ and therefore $c(T) \leq c(T')$. Since T' $\Longrightarrow$ is an MST, we have $c(T) = c(T')$ and hence T is also an MST.[Bild. eleganter?] ∎

Lemma 11.2 (Cycle Property:) *Consider any cycle $C \subseteq E$ and an edge $e \in C$ with maximal weight. Then any MST of $G' = (V, E \setminus \{e\})$ is also an MST of G.*

Proof: Consider any MST T of G. Since trees contain no cycles, there must be some edge $e' \in C \setminus T$. If $e = e'$ then T is also an MST of G' and we are done. Otherwise, $T' = \{e'\} \cup T \setminus \{e\}$ forms another tree and since $c(e') \leq c(e)$, T' must also form an MST of G. ∎

Using the cut property, we easily obtain a greedy algorithm for finding a minimum spanning tree: Start with an empty set of edges T. While T is not a spanning tree, add an edge fulfilling the cut property.

There are many ways to implement this generic algorithm. In particular, we have the choice which S we want to take. We also have to find out how to find the smallest edge in the cut efficiently. We discuss two approaches in detail in the following sections and outline a third approach in Section 11.6.

Exercises

Exercise 11.4 Show that the MST is uniquely defined if all edge weights are different. Show that in this case the MST does not change if each edge weight is replaced by its rank among all edge weights.

11.2 The Jarník-Prim Algorithm

[pictures in pseudocode. Pseudocode nur in implementation notes?] ⟸

```
Function jpMST(V, E, w) : Set of Edge
    dist=⟨∞,...,∞⟩ : Array [1..n]          // dist[v] is distance of v from the tree
    pred : Array of Edge                    // pred[v] is shortest edge between S and v
    q : PriorityQueue of Node with dist[·] as priority
    q.insert(s) for any s_0 ∈ V
    for i := 1 to n − 1 do
        s := q.deleteMin()                                  // new node for S
        dist[s] := 0
        foreach (s,v) ∈ E do
            if c((s,v)) < dist[v] then
                dist[v] := c((s,v))
                pred[v] := (s,v)
                if v ∈ q then q.decreaseKey(v) else q.insert(v)
    return {pred[v] : v ∈ V \ {s_0}}
```

Figure 11.1: The Jarník-Prim MST Algorithm.

The Jarník-Prim (JP) Algorithm for MSTs is very similar to Dijkstra's algorithm for shortest paths.[4] Starting form an (arbitrary) source node s, the JP-algorithm grows a minimum spanning tree by adding one node after the other. The set S from the cut property is the set of nodes already added to the tree. This choice of S guarantees that the smallest edge leaving S is not in the tree yet. The main challenge is to find this edge efficiently. To this end, the algorithm maintains the shortest connection between any node $v \in V \setminus S$ to S in a priority queue q. The smallest element in q gives the desired edge. To add a new node to S, we have to check its incident edges whether they give improved connections to nodes in $V \setminus S$. Figure 11.1 gives pseudocode for
⟹ the JP-algorithm. [example. (JP plus Kruskal)] [harmonize with description of
⟹ Dijkstra's algorithm] Note that by setting the distance of nodes in S to zero, edges connecting s with a node $v \in S$ will be ignored as required by the cut property. This small trick saves a comparison in the innermost loop.

The only important difference to Dijkstra's algorithm is that the priority queue stores edge weights rather than path lengths. The analysis of Dijkstra's algorithm transfers to the JP-algorithm, i.e., using a Fibonacci heap priority queue, $O(n \log n + m)$ execution time can be achieved.

Exercises

Exercise 11.5 Dijkstra's algorithm for shortest paths can use monotonous priority queues that are sometimes faster than general priority queues. Give an example to show that monotonous priority queues do *not* suffice for the JP-algorithm.

⟹ [subsection of its own?]

****Exercise 11.6 (Average case analysis of the JP-algorithm)** Assume the edge weights 1,...,m are randoly assigned to the edges of G. Show that the expected number of *decreaseKey* operations performed by the JP-algorithm is then bounded by $O(??)$[Referenz?
⟹ Bezug zu SSSP.].

11.3 Kruskal's Algorithm

Although the JP-algorithm may be the best general purpose MST algorithm, we will now present an alternative algorithm. Kruskal's algorithm [60] does not need a full fledged graph representation but already works well if it is fed a list of edges. For sparse graphs with $m = O(n)$ it may be the fastest available algorithm.

[4] Actually Dijkstra also describes this algorithm in his seminal 1959 paper on shortest paths [32]. Since Prim described the same algorithm two years earlier it is usually named after him. However, the algorithm actually goes back to a 1930 paper by Jarn´ık [46].


```
Function kruskalMST(V, E, w) : Set of Edge
    T := ∅                                              // subforest of the MST
    foreach (u,v) ∈ E in ascending order of weight do
        if u and v are in different subtrees of T then
            T := T ∪ {(u,v)}                            // Join two subtrees
    return T
```

Figure 11.2: Kruskal's MST Algorithm.

The pseudocode given in Figure 11.2 looks almost trivial. Each edge (u,w) is considered once and it is immediately decided whether it is an MST edge. We will see that this decision is quite easy because edges are scanned in order of increasing weight. The set T of MST edges found so far forms a subforest of the MST, i.e., a collection of subtrees. If u and w lie in the same subtree $U \subseteq T$ then (u,w) is a heaviest edge on some cycle $C \subseteq U \cup \{(u,w)\}$. Hence, the cycle property tells us that (u,w) can safely be ignored. Otherwise, u and w lie in different subtrees U and W. Consider the cut defined by U and $V \setminus U$. No edge lighter than (u,w) can connect U to something else because otherwise this edge would have been selected before. Hence, the cut property ensures that (u,w) can be used for an MST.

The most interesting algorithmic aspect of Kruskal's algorithm is how to implement the test whether an edge bridges two subtrees in T. In the next section we will see that this can be implemented very efficiently so that the main cost factor is sorting the edges. This takes time $O(m \log m)$ if we use an efficient comparison based sorting algorithm. The constant factor involved is rather small so that for $m = O(n)$ we can hope to do better than the $O(m + n \log n)$ JP-algorithm.

Exercises

Exercise 11.7 Explain how Kruskal fits into the framework of the generic greedy algorithm based on the cut property, i.e., explain which set S must choosen in each iteration of the generic algorithm to find the MST edges in the same order as Kruskal's algorithm.

Exercise 11.8 (Streaming MST) Suppose the edges of a graph are presented to you only once (for example over a network connection) and you do not have enough memory to store all of them. The edges do *not* necessarily arrive in sorted order.

a) Outline an algorithm that nevertheless computes an MST using space $O(V)$.

*b) Refine your algorithm to run in time $O(m \log n)$. Hint: Use the *dynamic tree* data structure by Sleator and Tarjan [89].

11.4 The Union-Find Data Structure

A *partition* of a set M into subsets $M_1, \ldots, M_k$ has the property, that the subsets are disjoint and cover M, i.e., $M_i \cup M_j = \emptyset$ for $i \neq j$ and $M = M_1 \cup \cdots \cup M_k$. For example, in Kruskal's algorithm the forest T partitions V into subtrees — including trivial subsets of size one for isolated nodes. Kruskal's algorithms performs two operations on the partition: Testing whether two elements are in the same subset (subtree) and joining two subsets into one (inserting an edge into T).

```
Class UnionFind(n : ℕ)                                  // Maintain a partition of 1..n
    parent=⟨1,2,...,n⟩ : Array [1..n] of 1..n           // bild mit self loops???
    gen=⟨0,...,0⟩ : Array [1..n] of 0..log n            // generation of leaders

Function find(i : 1..n) : 1..n                          // picture 'before'
    if parent[i] = i then return i
    else i' := find(parent[i])
        parent[i] := i'                                 // path compression
        return i'                                       // picture 'after'

Procedure link(i, j : 1..n)                             // picture 'before'
    assert i and j are leaders of different subsets
    if gen[i] < gen[j] then parent[i] := j              // balance
    else
        parent[j] := i
        if gen[i] = gen[j] then gen[i]++
    Procedure union(i, j : 1..n)
        if find(i) ≠ find(j) then link(find(i), find(j))
```

Figure 11.3: An efficient Union-Find data structure maintaining a partition of the set $\{1, \ldots, n\}$.

The *union-find data structure* maintains a partition of the set $1..n$ and supports these two operations. Initially, each element is in its own subset. Each subset is assigned a *leader* element[term OK? 'representative' (CLR) or 'canonical element' are such long words. too much confusion between leader and parent?]. The function $find(i)$ finds the leader of the subset containing i; $link(i, j)$ applied to leaders of different partitions joins these two subsets. Figure 11.3 gives an efficient implementation of this idea. The most important part of the data structure is the array *parent*. Leaders are their own parents. Following parent references leads to the leaders. The *parent* references of a subset form a *rooted tree*[where else], i.e., a tree with all edges directed towards the root.[5] Additionally, each root has a self-loop. Hence, *find* is easy to implement by following the *parent* references until a self-loop is encountered.

Linking two leaders i and j is also easy to implement by promoting one of the leaders to overall leader and making it the parent of the other. What we have said so far yields a correct but inefficient union-find data structure. The *parent* references could form long chains that are traversed again and again during *find* operations.

Therefore, Figure 11.3 makes two optimizations. The *link* operation uses the array *gen* to limit the depth of the *parent* trees. Promotion in leadership is based on the seniority principle. The older generation is always promoted. We will see that this measure alone limits the time for *find* to $O(\log n)$. The second optimization is *path compression*. A long chain of parent references is never traversed twice. Rather, *find* redirects all nodes it traverses directly to the leader. We will see that these two optimizations together make the union-find data structure "breath-takingly" efficient — the amortized cost of any operation almost constant.

Analysis

[todo. trauen wir uns an Raimund's neue analyse?]

Exercises

Exercise 11.9 Describe a nonrecursive implementation of *find*.

Exercise 11.10 Give an example for an n node graph with $O(n)$ edges where a naive implementation of the union-find data structure without balancing or path compression would lead to quadratic execution time for Kruskal's algorithm.

[a separate section for improved Kruskal with filtering?]

11.5 Implementation Notes

Good minimum spanning tree algorithms are so fast that running time is usually dominated by the time to generate the graphs and appropriate representations. If an

[5] Note that this tree may have very different structure compared to the corresponding subtree in Kruskal's algorithm.

adjacency array representation of undirected graphs as described in Section 8.2 is used, then the JP-algorithm works well for all m and n in particular if pairing heaps [73, 54, **?**] are used for the priority queue. It might be good to store all the information related to a node (*dist*, *pred*, priority queue entry, reference to adjacency array) in a single record. Kruskal's algorithm may be faster for sparse graphs, in particular, if only a list or array of edges is available or if we know how to sort the edges very efficiently.

11.6 Further Findings

The oldest MST algorithm is also based on the cut property. Boruvka's [16, 74] algorithm goes back to 1926 and hence represents one of the oldest graph algorithms. The algorithm finds many MST edges in each *phase*. In the first phase, every node finds its lightest incident edge. The next phases are reduced to this simple case by *contracting* the MST edges already found. Contracting an edge (u,v) means to remove u and v from V and to replace them by a new node u'. Edge (u,v) is removed. Edges of the form (u,w) and (v,w) are replaced by an edge (u',w). If this procedure generates several parallel edges between u' and w, only the lightest one survives. Each phase of Boruvka's algorithm can be implemented to run in time $O(m)$. Since a phase at least halves the number of remaining nodes, only a single node is left after $O(\log n)$ phases. By keeping track of the original terminals of edges we can output the MST as a side product. Boruvka's algorithm is not used often because it is somewhat complicated to implement. It is nevertheless important as a basis for parallel and external memory MST algorithms.

There is a randomized linear time MST algorithm that uses phases of Boruvka's algorithm to reduce the number of nodes [52, 56]. The second ingredient of this algorithm reduces the number of edges to $O(\sqrt{mn})$: sample $O(\sqrt{mn})$ edges randomly; find an MST T' of the sample; remove edges $e \in E$ that are the heaviest edge on a cycle in $e \cup T'$. The last step is rather difficult to implement efficiently. But at least for rather dense graphs this approach can yield a practical improvement [54].

The theoretically best known *deterministic* MST algorithm [78] has the interesting property that it has optimal worst case complexity although it is not exactly known what this complexity is. Hence, if you come tomorrow with a completely different deterministic MST algorithm and prove that your algorithm runs in linear time, then we know that the algorithm by Pettie and Ramachandran [78] also runs in linear time.

Minimum spanning trees define a single path between any pair of nodes s and t. Interestingly, this path is a *bottleneck shortest path* [4, Application 13.3], i.e., it minimizes the maximum edge weight for all paths from s to t in the original graph. Hence, finding an MST amounts to solving the all-pairs bottleneck shortest path problem in time much less than for solving the all-pairs shortest path problem.

A related and even more frequently used application is clustering based on the MST [4, Application 13.5]: By dropping $k-1$ edges from the MST it can be split into k subtrees. Nodes in a subtree T' are far away from the other nodes in the sense that all paths to nodes in other subtrees use edges that are at least as heavy as the edges used to cut T' out of the MST.

Many applications of MSTs define complete graphs with $n(n-1)/2$ edges using a compact implicit description of the graph. Then it is an important concern whether one can rule out most of the edges as too heavy without actually looking at them. For example, if the nodes represent points in the plane and if edge weights are Euclidean distances, one can exploit the geometrical structure of the problem. It can be proven that the MST or the complete geometric graph is contained in various well know $O(n)$ subgraphs that have size $O(n)$ and can be computed in time $O(n \log n)$ (Delaunay triangulation, Gabriel traph [79], Yao graph [**?**]).

Delaunay-Triangulation of the point set [79]. The Delaunay-Triangulation is an $O(n)$ subset of the edges that can be found in time $O(n \log n)$. Hence, MSTs of 2D point sets with Euclidean distance function can be found in time $O(n \log n)$. We will see another example for implicitly defined complete graphs below.

Although we introduced MSTs as a network design problem, most network design problems of practical interest ask more general questions that are very hard to solve exactly. For example, a *Minumum Weight Steiner Tree* (MWST) $T \subseteq E$ connects a set of terminals $U \subseteq V$. The *Steiner nodes* $V \setminus U$ need not be connected but can be used to reduce the overall weight. In our cabling example, the Steiner nodes could represent uninhabited islands that do not need a telephone but can host a switch. MSTs can be used to approximate MWSTs. Consider the complete graph $G' = (U, E')$ where $c((s,t))$ is the shortest path distance between s and t in $G = (V,E)$. An MST in G' yields a Steiner tree spanning U in G by taking the edges in E from all paths used to connect nodes in G'. This Steiner tree hast at most twice the weight of an optimal Steiner tree. Although G' can have $\Omega(n^2)$ edges, its MST can be found efficiently in time $O(n \log n + m)$ [65].

[overview paper of other generalizations] ⟸

[TSP approximation? Held-Karp lower bound?] ⟸

[checkers for MST?] ⟸

Chapter 12

Generic Approaches to Optimization

A smuggler in the mountainous region of Profitania has n items in his cellar. If he sells item i across the border, he makes a profit $p_i \in \mathbb{N}$. However, the smuggler's trade union only allows him to carry knapsacks with maximum additional weight $M \in \mathbb{N}$. If item i has a weight of $w_i \in \mathbb{N}$, what items should he pack into the knapsack to maximize the profit in his next trip?

This *knapsack problem* has many less romantic applications like [nachschauen] ⟸ [61, 55]. In this chapter we use it as a model problem to illustrate several generic approaches to solve optimization problems. These approaches are quite flexible and can be adapted to complicated situations that are ubiquitous in practical applications. In the previous chapters we looked at specific very efficient solutions for frequently occurring simple problems like finding shortest paths or minimum spanning trees. Now we look at generic solutions that work for a much larger range of applications but may be less efficient.

More formally, an optimization problem can be described by a set $\mathcal{U}$ of potential solutions, a set $\mathcal{L}$ of *feasible* solutions, and an *objective function* $f : \mathcal{L} \to \mathbb{R}$. In a *maximization* problem, we are looking for a feasible solution $\mathbf{x}^* \in \mathcal{L}$ that maximizes $f(\mathcal{L}^*)$ among all feasible solutions. In a *minimization* problem, we look for a solution minimizing f. For example, the knapsack problem is a maximization problem with $\mathcal{U} = \{0,1\}^n$, $\mathcal{L} = \{\mathbf{x} = (x_1,\ldots,x_n) \in \mathcal{U} : \sum_{i=1}^n x_i w_i \leq M\}$, and $f(\mathbf{x}) = \sum_{i=1}^n x_i p_i$. Note that the distinction between minimization and maximization problems could be avoided because setting $f := -f$ converts a maximization problem into a minimization problem and vice versa. We will use maximization as our default simple because

our mode problem is more naturally viewed a maximization problem.[1]

We present six basic approaches to optimization roughly ordered by conceptual simplicity. Perhaps the easiest solution is to use an existing black box solver that can be applied to many problems so that the only remaining task is to formulate the problem in a language understood by the black box solver. Section 12.1 introduces this approach using *linear programming* as an example. The greedy approach that we have already seen in Section **??** is reviewed in Section 12.2. The *dynamic programming* approach discussed in Section 12.3 is a more flexible way to construct solutions. We can also systematically explore the entire set of potential solutions as described in Section 12.4. Finally we discuss two very flexible approaches to explore only a subset of the solution space. *Local search* discussed in Section 12.5 modifies a single solution until it is satisfied whereas the evolutionary algorithms explained in Section **??** simulate a population of solution candidates.

Exercises

Exercise 12.1 (Optimizing versus Deciding.) Suppose you have a routine $\mathcal{P}$ that outputs a feasible solution with $f(\mathbf{x}) \leq a$ if such a solution exists and signals failure otherwise.

a) Assume that objective function values are positive integers. Explain how to find an minimal solution $\mathbf{x}^*$ using $O(\log f(\mathbf{x}^*))$ calls of $\mathcal{P}$.

*****b) Assume that objective function values are reals larger than one. Explain how to find a solution that is within a factor $(1+\varepsilon)$ from minimal using $O(\log\log f(\mathbf{x}^*) + \log(1/\varepsilon))$ calls of $\mathcal{P}$. Hint: use the geometric mean $\sqrt{a \cdot b}$.

12.1 Linear Programming — A Black Box Solver

The easiest way to solve an optimization problem is to write down a specification of the solution space $\mathcal{L}$ and the objective function f and then use an existing software package to find an optimal solution. The question is of course for what kinds of problem specifications such general solvers are available. Here we introduce a black box solver for a particularly large class of problems.

⟹ [what about constraint solvers as another black box?]

Definition 12.1 A *Linear Program (LP)*[2] with *n variables* and *m constraints* is specified by the following maximization problem: A linear objective function $f : \mathbb{R}^n \to \mathbb{R}$ with $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$ where $\mathbf{c}$ is called the *cost vector* and where "$\cdot$" stands for the scalar product of two n-vectors. Constraints have the form $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$ where $\bowtie_i \in \{\leq, \geq, =\}$ and $\mathbf{a}_i \in \mathbb{R}^n$ for $i \in 1..m$. We have

$$\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n : \forall i \in 1..m : x_i \geq 0 \wedge \mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i\} \quad .$$

Let a_{ij} denote the j-th component of vector $\mathbf{a}_i$.

Figure 12.1 gives a simple example for a linear program. We have $n = 2$ variables (x and y) and $m = 3$ constraints. The cost vector is $\mathbf{c} = (1,4)$. The optimal solution is $(x,y) = (2,6)$.

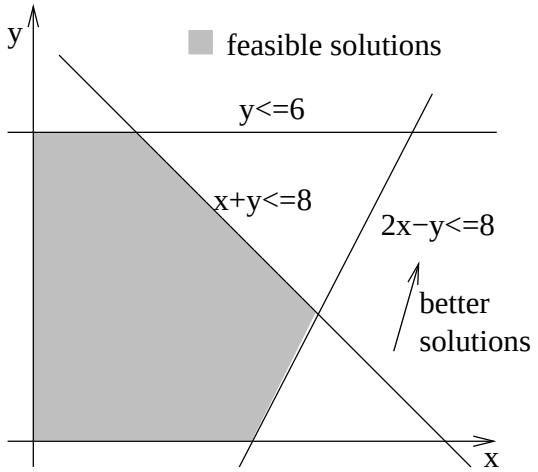


Figure 12.1: A simple two-dimensional linear program with three constraints.

Here is a classical application of linear programming: A farmer wants to mix food for his cows. There are n different kinds food on the market, say, corn, soya, fish meal,.... One kilogram of kind j costs c_j Euro. There are m requirements for a healthy nutrition, e.g., the cows should get enough calories, proteins, Vitamin C, ...One kilogram of kind i contains a_{ij} percent of the daily requirement of an animal with respect to requirement i. Then a solution of the corresponding linear program gives a cost optimal and (reasonably) healthy diet.

[max flow?] ⟸

Linear programming is so important because it is one of the most general problem formulations for which efficient solution algorithms are known. In particular:

[1] Be aware that most of the literature uses minimization as its default.

[2] The term "linear program" stems from the 1940s [**?**] and has nothing to do with with the modern meaning "computer program".

Theorem 12.2 *A linear program can be solved in polynomial time.*

The worst case execution time of these polynomial algorithms can still be rather high. However, most linear programs can be solved relatively quickly by several procedures. One, the simplex algorithm, is briefly outlined in Section 12.5.1. For now, the only thing we need to know is that there are efficient packages that we can use to solve linear programs. Infact, very few people in the world know every detail of efficient LP solvers.

12.1.1 Integer Linear Programming

Very often one would like to solve linear programs where the variables are only allowed to take integer values. Such problems are called *Integer Linear Programs (ILP)*. (Problems where only some variables must be integer are called *Mixed Integer Linear Programs (MILP)*.) For example, an instance of the knapsack problem is the 0-1 integer linear program

$$\text{maximize} \mathbf{p} \cdot \mathbf{x}$$

subject to

$$\mathbf{w} \cdot \mathbf{x} \leq M, x_i \in \{0,1\} \text{ for } i \in 1..n \ .$$

In a sense, the knapsack problem is the simplest 0-1 ILP since it has only a single additional constraint. Unfortunately, solving ILPs and MILPs is NP-hard[where explained?].

Nevertheless, ILPs can often be solved in practice using linear programming packages. In Section 12.4 we will outline how this is done. Even if we cannot solve an ILP exactly, linear programming can help us to find approximate solutions. In a *linear relaxation* of an ILP, we simply omit the integrality constraints and obtain an ordinary LP that can be solved efficiently. For example in the knapsack problem we would replace the constraint $x_i \in \{0,1\}$ by the constraint $x_i \in [0,1]$. The resulting *fractional* solutions of the linear relaxation often has only few variables set to *fractional* (i.e., noninteger) values. By appropriate rounding of fractional variables to integer values, we can often obtain good *integer feasible solutions*.

For example, the linear relaxation of the knapsack problem asks for a maximum profit solution if the items can be arbitrarily cut. In our smuggling "application" this *fractional knapsack problem* would even make sense if the smugglers are dealing with powdery substances. In Exercise **??** we ask you to show that the following $O(n \log n)$ greedy algorithm finds the optimal solution:
Renumber (sort) the items by *profit density* such that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n} \ .$$

Find the smallest index j such that $\sum_{i=1}^{j} w_i > M$ (if there is no such index, we can take all knapsack items). Now set

$$x_1 = \cdots = x_{j-1} = 1, x_j = M - \sum_{i=1}^{j-1} w_i, \text{ and } x_{j+1} = \cdots = x_n = 0 \ .$$

Figure **??** gives an example. For the knapsack problem we only have to deal with a single fractional variable x_j. Rounding x_j to zero yields a feasible solution that is quite good if $p_j \ll \sum_{i=1}^{j-1} p_i$. This fractional solution is the starting point for many good algorithms for the knapsack problem.

Exercises

Exercise 12.2 (More Animal Food.) How do you model the farmer's own supplies like hay which are cheap but limited? Also explain how to additionally specify upper bounds like "no more than 0.1mg Cadmium contamination per cow and day".

[min cost flow?, multi-commodity flow?]

Exercise 12.3 Explain how to replace any ILP by an ILP that uses only 0/1 variables. If U is an upper bound on the b_i values, your 0/1-ILP should be at most a factor $O(\log U)$ larger than the original ILP.

Exercise 12.4 Formulate the following *set covering* problem as an ILP: Given a set $M = 1..m$, n subsets $M_i \subseteq M$ for $i \in 1..n$ and a cost c_i for set M_i. Assume $\bigcup_{i=1}^{n} M_i = 1..m$. Select $F \subseteq 1..n$ such that $\bigcup_{i \in F} M_i = 1..m$ and $\sum_{i \in F} c_i$ is minimized.

Exercise 12.5 (Linear time fractional knapsacks.) Explain how to solve the fractional knapsack problem in linear expected time. Hint: use a similar idea as in Section 5.5.

12.2 Greedy Algorithms — Never Look Back

We have already seen greedy optimization algorithms for shortest paths in Section 10 and for minimum spanning trees in Chapter 11. There we were "lucky" in the sense that we got optimal solutions by carefully choosing edges of a graph. Usually, greedy algorithms only yield suboptimal solutions. For example, inspired by the fractional solution from Section 12.1.1, we could solve the knapsack problem by greedily including items of largest profit density that do not exceed its capacity. The example from Figure **??** shows that this does not lead to optimal solutions. Nevertheless, greedy

algorithms are often the best choice for getting a reasonable solution quickly. Let us look at another typical example.

Suppose you have m identical machines that can be used to process n *weighted* jobs of size $t_1,\ldots,t_j$. We are looking for a mapping $\mathbf{x} : 1..n \to 1..m$ from jobs to machines such that the *makespan* $L_{\max} = \max_{j=1}^{m} \sum \{t_i : \mathbf{x}(i)=j\ t_i\}$ is minimized. This is one of the simplest among an important class of optimization problems known as *scheduling* problems.

We give a simple greedy algorithm for scheduling *independent weighted jobs on identical machines* that has the advantage that we do not need to know the job sizes in advance. We assign jobs in the order they arrive. Algorithms with this property are known as *online* algorithms. When job i arrives we inspect the *loads* $\ell_j = \sum\{t_i : i < j, \mathbf{x}(i) = j\}$ and assign the new job to most the lightly loaded machine, i.e., $\mathbf{x}(i) := \min_{j=1}^{m} \ell_j$. This *shortest queue* algorithm does not guarantee optimal solutions but at least we can give the following performance guarantee:

Theorem 12.3 *The shortest queue algorithm ensures that* $L_{\max} \leq \frac{1}{m}\sum_{i=1}^{n} t_i + \frac{m-1}{m}\max_{i=1}^{n} t_i$.

Proof: We focus on the job $\hat{\imath}$ that is the last job being assigned to the machine with maximum load. When job $\hat{\imath}$ is scheduled, all m machines have load at least $L_{\max} - t_{\hat{\imath}}$, i.e.,

$$\sum_{i \neq \hat{\imath}} t_i \geq (L_{\max} - t_{\hat{\imath}}) \cdot m \ .$$

Solving this for $L_{\max}$ yields

$$L_{\max} \leq \frac{1}{m}\sum_{i \neq \hat{\imath}} t_i + t_{\hat{\imath}} = \frac{1}{m}\sum_{i} t_i + \frac{m-1}{m} t_{\hat{\imath}} \leq \frac{1}{m}\sum_{i=1}^{n} t_i + \frac{m-1}{m}\max_{i=1}^{n} t_i \ .$$

∎

In particular, if there are many rather small jobs, we get a solution that is very close to the obvious lower bound of $\sum_i t_i/m$ on $L_{\max}$. On the other hand, if there is a job that is larger than the average than $\sum_i t_i/m$, this is also a lower bound and the shortest queue algorithm will not be too far away from it either. In Exercise 12.6 we ask to show that we get the following guarantee.

Corollary 12.4 *The shortest queue algorithm computes a solution where* $L_{\max}$ *is at most a factor* $2 - 1/m$ *larger than for the optimal solution.*

When an approximation algorithm for a minimization problem guarantees solutions that are at most a factor a larger than an optimal solution, we say that the algorithm achieves *approximation ratio a*. Hence, we have shown that the shortest queue algorithm achieves an approximation ratio of $2 - 1/m$. This is tight because for $n = m(m-1)+1$, $t_n = m$, and $t_i = 1$ for $i < n$, the optimal solution has makespan $L_{\max} = m$ whereas the shortest queue algorithm produces a solution with makespan $L_{\max} = 2m-1$. Figure **??** gives an example for $m = 4$.

Similarly, when an online algorithm for a minimization problem guarantees solutions that are at most a factor a larger than solutions produced by an algorithm that knows the entire input beforehand, we say that the algorithm has *competetive ratio a*, i.e. the shortest queue algorithm has competitive ratio $2 - 1/m$.

Exercises

Exercise 12.6 Prove Corollary **??**. Hint: distinguish the cases $t_{\hat{\imath}} \leq \sum_i t_i/m$ and $t_{\hat{\imath}} > \sum_i t_i/m$.

***Exercise 12.7** Show that the shortest queue algorithm achieves approximation ratio $4/3$ if the jobs are sorted by decreasing size.

***Exercise 12.8 (Bin packing.)** Suppose a smuggler boss has perishable goods in her cellar. She has to hire enough porters to ship all items this night. Develop a greedy algorithm that tries to minimize the number of people she needs to hire assuming that they can all carry maximum weight M. Try to show an approximation ratio for your *bin packing* algorithm.

12.3 Dynamic Programming — Building it Piece by Piece

For many optimization problems the following *principle of optimality* holds: *An optimal solution can be viewed as constructed from optimal solutions of subproblems. Furthermore, for a given subproblem size it does not matter which optimal solution is used.*

The idea behind dynamic programming is to build an exhaustive table of optimal solutions starting with very small subproblems. Then we build new tables of optimal solutions for increasingly larger problems by constructing them from the tabulated solutions of smaller problems.

Table 12.1: A dynamic programming table for the knapsack problem "maximize $(10,20,15,20)\cdot\mathbf{x}$ subject to $(1,3,2,4)\cdot\mathbf{x}\leq 5$". Table entries have the form "$P(i,C)$, (x_i)". Bold face entries contribute to the optimal solution.

$i\setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	**0, (0)**	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	**20, (1)**	30, (1)	30, (1)
3	0, (0)	10, (0)	15, (1)	25, (1)	30, (0)	**35, (1)**
4	0, (0)	10, (0)	15, (0)	25, (0)	30, (0)	**35, (0)**

Again, we use the knapsack problem as an example. Define $P(i,C)$ as the maximum profit possible using only items 1 through i and total weight at most C. Set $P(0,C)=0$ and also $P(i,C)=0$ if $C\leq 0$.

Lemma 12.5 $\forall i\in 1..n : P(i,C)=\max(P(i-1,C),P(i-1,C-c_i)+p_i)$.

Proof: Certainly, $P(i,C)\geq P(i-1,C)$ since having a wider choice of items can only increase the obtainable profit. Furthermore $P(i,C)\geq P(i-1,C-c_i)+p_i$ since a solution obtaining profit $P(i-1,C-c_i)$ using items 1 through $i-1$ can be improved by putting item i into the knapsack. Hence,

$$P(i,C)\geq\max(P(i-1,C),P(i+1,C-c_i)+p_i)$$

and it remains to show the "$\leq$" direction. So, assume there is a solution $\mathbf{x}$ with $P(i,C)>P(i-1,C)$ and $P(i,C)>P(i-1,C-c_i)+p_i)$. We can distinguish two cases:

Case $x_i=0$: The solution $\mathbf{x}$ does not use item i and hence it is also a solution of the problem using item 1 through $i-1$ only. Hence, by definition of P, $P(i,C)\leq P(i-1,C)$ contradicting the assumption $P(i,C)>P(i-1,C)$.

Case $x_i=1$: If we remove item i from the solution we get a feasible solution of a knapsack problem using items 1 through $i-1$ and capacity at most $C-c_i$. By our assumption we get a profit larger than $P(i-1,C-c_i)+p_i-p_i=P(i-1,C-c_i)$. This contradicts our definition of P. ∎

Using Lemma 12.5 we can compute optimal solutions by filling a table top-down and left to right. Table 12.1 shows an example computation.

Figure **??** gives a more clever list based implementation of the same basic idea. Instead of computing the maximum profit using items $1..i$ and *all* capacities $1..C$, it


```
Function dpKnapsack
    L=⟨0⟩ : ℒ                    // candidate solutions. Initially the trivial empty solution
    for i := 1 to n do
        invariant L is sorted by total weight w(x) = Σ_{k<i} x_i w_i
        invariant L contains all Pareto optimal solutions for items 1..i
        L' := ⟨x ∪ {i} : x ∈ L, w(x) ≤ M⟩      // '∪' treats a solution as a set of items
        L := prune(merge(L,L'))                 // merge by w(x), see Figure 5.2
    return L.last

Function prune(L : Sequence of sol) : Sequence of ℒ
    p := −1                                     // best profit seen so far
    L' := ⟨⟩                                    // Pareto optimal solutions
    foreach x ∈ L do
        if p(x) > p then                        // p(x) = Σ_{k<i} x_i p_i is the total profit of the items in x
            p := p(x)
            L'.pushBack(x)
    return L'
```

Figure 12.2: A dynamic programming algorithm for the knapsack problem.

only computes *Pareto optimal* solutions. A solution $\mathbf{x}$ is Pareto optimal if there are no other solutions that achieve higher profit using no more knapsack capacity than $\mathbf{x}$. We cannot overlook optimal solutions by this omission since solutions of subproblems that are not Pareto optimal could be be improved by replacing them with solutions that have better profit.

Algorithm *dpKnapsack* needs $O(nM)$ worst case time. This is quite good if M is not too large. Since the running time is polynomial in n and M, *dpKnapsack* is a *pseudopolynomial* algorithm. The "Pseudo" means that this is not necessarily polynomial in the *input size* measured in bits — we can encode an exponentially large M in a polynomial number of bits. [say sth about average case complexity] ⟸

Exercises

Exercise 12.9 (Making Change.) Suppose you have to program a vending machine that should give exact change using a minimum number of coins.

a) Develop an optimal greedy algorithm that works in the Euro zone with coins worth 1, 2, 5, 10, 20, 50, 100, and 200 cents and in the Dollar zone with coins worth 1, 5, 10, 25, 50, and 100 cents.

b) Show that this algorithm would not be optimal if there were a 4 cent coin.

c) Develop a dynamic programming algorithm that gives optimal change for any currency system.

Exercise 12.10 (Chained Matrix Multiplication.) We want to compute the matrix product $M_1M_2 \cdot M_n$ where M_i is a $k_{i-1} \times k_i$ matrix. Assume that a pairwise matrix product is computed in the straight forward way using *mks* element multiplications for the product of an $m \times k$ matrix with an $k \times s$ matrix. Exploit the associativity of the matrix product to minimize the number of element multiplications needed. Use dynamic programming to find an optimal evaluation order in time $O(n^3)$.

Exercise 12.11 (Minimum edit distance.) Use dynamic programming to find the *minimum edit distance* between two strings s and t. The minimum edit distance is the minimum number of character deletions, insertions, and replacements applied to s that produces string t.

Exercise 12.12 Does the principle of optimality hold for minimum spanning trees? Check the following three possibilities for definitions of subproblems: Subsets of nodes, arbitrary subsets of edges, and prefixes of the sorted sequence of edges.

Exercise 12.13 (Constrained shortest path.) Consider a graph with $G = (V,E)$ where edges $e \in E$ have a *length* $\ell(e)$ and a *cost* $c(e)$. We want to find a path from node s to node t that minimizes the total cost of the path subject to the constraint that the total length of the path is at most L. Show that subpaths $[s',t']$ of optimal solutions are *not* necessarily optimal paths from s' to t'.

Exercise 12.14 Implement a table based dynamic programming algorithm for the knapsack problem that needs $nM + O(M)$ bits of space.

Exercise 12.15 Implement algorithm *dpKnapsack* from Figure 12.2 so that all required operations on solutions ($w(\mathbf{x})$, $p(\mathbf{x})$, $\cup$) work in constant time.

12.4 Systematic Search — If in Doubt, Use Brute Force

In many optimization problems, the universe of possible solutions $\mathcal{L}$ is finite so that we can in principle solve the optimization problem by trying all possibilities. If we apply this idea naively, there are only few cases where we can get away with it. But often, a few tricks can make it possible to systematically explore all *promising* candidates.

⟹ [section overview, basic ideas]

12.4.1 Branch-and-Bound

Function *bbKnapsack*($\langle p_1,\ldots,p_n\rangle, \langle w_1,\ldots,w_n\rangle, M$) : $\mathcal{L}$
  assert $p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$ // assume input is sorted by profit density
  $\hat{\mathbf{x}}$=*heuristicKnapsack*($\langle p_1,\ldots,p_n\rangle, \langle w_1,\ldots,w_n\rangle, M$) : $\mathcal{L}$ // best solution so far
  $\mathbf{x}$: $\mathcal{L}$ // current partial solution
  recurse$(1, M, 0)$

  // Find solutions assuming $x_1,\ldots,x_{i-1}$ are fixed, $M' = M - \sum_{k<i} x_i w_i$, $P = \sum_{k<i} x_i p_i$.
  Procedure *recurse*$(i, M', P : \mathbb{N})$
    if $P +$ *upperBound*$(\langle p_i,\ldots,p_n\rangle, \langle w_i,\ldots,w_n\rangle, M') > \sum_{i=1}^{n} \hat{\mathbf{x}}_i p_i$ **then**
      if $i > n$ **then** $\hat{\mathbf{x}} := \mathbf{x}$
      else // Branch on variable x_i
        if $w_i \leq M$ **then** $x_i := 1$; *recurse*$(i+1, M'-w_i, P+p_i)$
        $x_i := 0$; *recurse*$(i+1, M', P)$

Figure 12.3: A branch-and-Bound algorithm for the knapsack problem. Function *heuristicKnapsack* constructs a feasible solution using some heuristic algorithm. Function *upperBound* computes an upper bound for the possible profit.

Figure 12.3 gives pseudocode for a systematic search routine for the knapsack problem. The algorithm follows a pattern known as *branch-and-bound*. The "branch" is the most fundamental ingredient of systematic search routines. Branching tries all sensible setting of a part of the result — here the values 0 and 1 for x_i — and solves the resulting subproblems recursively. [Bild mit Beispielsuchbaum] Algorithms based ⟸ on branching systematically explore the resulting tree of subproblems. Branching decisions are internal nodes of this search tree.

"Bounding" is a more specific method to prune subtrees that cannot contain promising solutions. A branch-and-bound algorithm keeps an incumbent $\hat{\mathbf{x}}$ for the best solution. Initially, the incumbent is found using a heuristic routine. In the knapsack example we could use a greedy heuristic that scans the items by decreasing profit density and includes items as capacity permits. Later $\hat{\mathbf{x}}$ contains the best solution found at any leaf of the search tree. This lower bound on the best solutions is complemented by an upper bound that can be computed quickly. In our example the upper bound could be the profit for the fractional knapsack problem with items $i..n$ and capacity $M - \sum_{j<i} x_i w_i$. In Section 12.1.1 we have seen that the fractional knapsack problem can be solved quickly. In Exercise 12.16 we even outline an algorithm that runs in time $O(\log n)$. Upper and lower bound together allow us to stop searching if the up-

per bound for a solution obtainable from **x** is no better than the lower bound already known.

12.4.2 Integer Linear Programming

Integer linear programs are usually solved by an intimate connection between a solver for linear programs and branch-and-bound. We use the case of 0-1 ILPs to explain the ingredients of branch-and-bound in more detail.

What is a branch? An solver for 0-1 ILPs picks a variable y and produces two branches in the search tree by setting $y = 1$ in one subtree and $y = 0$ in the other subtree.

Bounding: The integrality constraints of the variables that are not fixed yet are relaxed and the optimal solution of the resulting linear relaxation is found using an LP solver. If this gives a worse solution than the best solution found so far, or if the linear relaxation has no feasible solution, then this subtree need not be further explored.

Where to branch: The ILP solver usually branches on a variable that is fractional in
⟹ a solution of the linear relaxation. [More heuristics?]

Finding Solutions: At the latest when all variables are fixed, a feasible solution is found. In many applications we are more lucky and the solution of the linear relaxation turns out to be an integer feasible solution already when few variables have been fixed. Application specific heuristics can additionally help to find good solutions quickly.

Order of Search Tree Traversal: In the knapsack example the search tree was traversed depth first and the 1-branch was tried first. In general, the search is free to choose any order of tree traversal. There are at least two considerations influencing this strategy. As long as no good feasible solutions are known, it is good to use a depth first strategy to fix enough variables that a feasible solution is found. Otherwise, a *best-first* strategy is better that explores those search tree nodes that are most likely to contain good solutions. Search tree nodes are kept in a priority queue and the next node to be explored is the most promising node in the queue. The priority could be the objective function value of the linear relaxation of the subproblem. Since this is expensive to evaluate, one sometimes settles for an approximation.

Branch-and-Cut: When an ILP solver branches too often, the size of the search tree explodes and it becomes too expensive to find an optimal solution. Therefore, it is important to avoid branching whenever possible. One possibility is to introduce additional constraints that cut off fractional solutions from the set of feasible solutions of the linear relaxation without changing the set of integer feasible solutions.

12.4.3 Shortest Paths Reconsidered

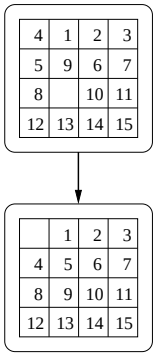
[mit sssp chapter abgleichen] ⟸
[Beispiel bei dem klar wird, dass es sich um einen Graphen handeln kann?] ⟸

Exercises

Exercise 12.16 (Logarithmic time upper bounds for the knapsack problem.) Explain how to implement the function *upperBound* in Figure 12.3 so that it runs in time $O(\log n)$. Hint: precompute prefix sums $\sum_{k\le i} w_i$ and $\sum_{k<i} p_i$ and use binary search[rather golden ratio search?]. ⟸

Exercise 12.17 (15-puzzle)

The 15-puzzle is a popular mechanical puzzle. You have to shift 15 scrambled squares in a 4×4 frame into the right order. Define a move as the action of moving one square into the hole. Implement a systematic search algorithm that finds a shortest move sequence from a given starting configuration to the order given in the picture to the right. Use *iterative deepening depth first search* [59]: Try all one move sequences first, then all two move sequences,... This should work for the simpler 8-puzzle. For the 15-puzzle use the following optimizations: Never undo the immediatly preceding move. Maintain the number of moves that would be needed if all pieces could be moved freely. Stop exploring a subtree if this bound proves that the current search depth is too small. Decide beforehand, whether the number of moves is odd or even. Implement your algorithm to run in constant time per move tried.



12.5 Local Search — Think Globally, Act Locally

Find some feasible solution $\mathbf{x} \in \mathcal{L}$
$\hat{\mathbf{x}} := \mathbf{x}$ // best solution found so far
while not satisfied with $\hat{\mathbf{x}}$ **do**
  $\mathbf{x}$:= some heuristically chosen element from $\mathcal{N}(\mathbf{x}) \cap \mathcal{L}$
  if $f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

Figure 12.4: Local search.

The optimization algorithms we have seen so far are only applicable in special circumstances. Dynamic programming needs a special structure of the problem and may require a lot of space and time. Systematic search is usually too slow for large inputs. Greedy algorithms are fast but often do not give very good solutions. *Local search* can be viewed as a generalization of greedy algorithms. We still solve the problem incrementally, but we are allowed to change the solution as often as we want possibly reconsidering earlier decisions.

Figure 12.4 gives the basic framework that we will later refine. Local search maintains a current feasible solution $\mathbf{x}$ and the best solution $\hat{\mathbf{x}}$ seen so far. We will see in Section 12.5.2 that the restriction to work with feasible solutions only can be circumvented. The idea behind local search is to incrementally modify the current solution $\mathbf{x}$. The main application specific design choice for a local search algorithm is to define how a solution can be modified. The *neighborhood* $\mathcal{N}(\mathbf{x})$ formalizes this concept. The second important design decision is which element from the neighborhood is chosen. Finally, some heuristic decides when to stop searching.

12.5.1 Hill Climbing

$\Longrightarrow$ [golden ration search? exercise?]

The most straightforward choice for a local search algorithm is to allow only new solutions which improve on the best solution found so far. This approach is known as *hill climbing*. In this case, the local search algorithm gets quite simple. The variables $\hat{\mathbf{x}}$ and $\mathbf{x}$ are the same and we stop when no improved solutions are in the neighborhood $\mathcal{N}$. The only nontrivial aspect of hill climbing is the choice of the neighborhood. For example, consider the case $\mathcal{L} = \{0,\ldots,k\}^2$. A natural choice seems to be $\mathcal{N}((x,y)) = \{(x+1,y),(x-1,y),(x,y-1),(x,y+1)\}$. However, this neighborhood fails even for the simple function $f(x,y) = 2y-x$ for $x > y$ and $f(x,y) = 2x-y$ for $x \leq y$. Figure 12.5[neu machen] depicts this function. Once x has climbed the ridge at at $x = y$ none of the coordinate directions leads to an increase of f.

$\Longrightarrow$

todo: a picture for the function $2y-x$ for $x > y$, $2x-y$ else.

Figure 12.5: An example where the orthogonal neighborhood does not find the global optimum.

An interesting example of hill climbing with a clever choice of the neighborhood function are algorithms for linear programming (see Section 12.1). We outline how the most widely used linear programming algorithm, the *simplex algorithm*, works. The set of feasible solutions $\mathcal{L}$ of a linear program is a *convex polytope*. Convex means that if you connect any two points in $\mathcal{L}$ by a line then all points on that line are also

in $\mathcal{L}$. A polytope is the n-dimensional generalization of a polygon. The border of the polytope is defined by areas where one or several of the constraints are satisfied with equality. *Corner points* of the polytope are points in $\mathcal{L}$ where n constraints are satisfied with equality. A nice thing about linear programs is that there must be a corner point which maximizes f (if there is a feasible solution at all). The simplex algorithm exploits this by constraining its search to the discrete set of corner points of $\mathcal{L}$. Corner points are neighbors if the set of n constraints fulfilled with equality differ by one constraint. Unless an optimal corner point has already been reached, there is always a neighboring corner point that decreases f. Moving between neighboring points can be performed using relatively simple methods from linear algebra.

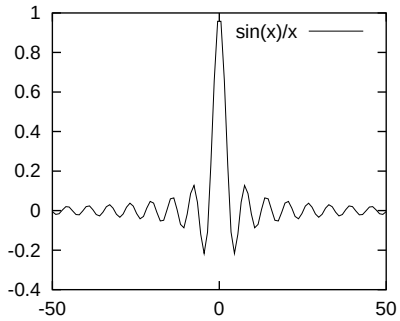


Figure 12.6: A function with many local optima.

Unfortunately, there are many optimization problems with less favorable behavior than linear programs. The main problem is that hill climbing may get stuck in *local optima* — feasible solutions where no solution in the neighborhood yields an improvement. This effect can even be seen in a one-dimensional example such as the one shown in Figure 12.6. The only cure for local optima in hill climbing is giving the neighborhood the right shape to eradicate local optima. However, in general this may involve impractically large neighborhoods. For example, consider the problem of finding he highest mountain in the world. If you do not start very near to the Mount Everest, even a viewing range of 500 km will not give you any hints about its whereabout.

12.5.2 Simulated Annealing —Learning from Nature

If we want to ban the bane of local optima in local search, we must find a way to escape from them. This means that we sometimes have to accept moves to feasible

solutions that make $f(\mathbf{x})$ worse. This implies a dilemma. We must be able to make many downhill steps to escape from wide local optima and at the same time we must be sufficiently goal directed to find a global optimum at the end of a long narrow ridge.

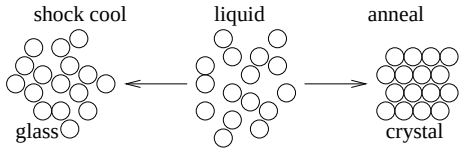


Figure 12.7: Annealing versus Shock Cooling.

One approach is inspired by behavior of physical systems. For example, consider a pot of molten quartz (SiO_2). If we cool it very quickly, we get glass — an amorphous substance where every molecule is in a local minimum of energy. This process of shock cooling is very similar to [translate min-max?]hill climbing. Every molecule simply drops into a state of locally minimal energy. But a much lower state of energy is reached by a quartz crystal where all the molecules are arranged in a regular way. This state can be reached (or approximated) by cooling the quartz very slowly. This process is called *annealing*. Figure 12.7 depicts this process. How can it be that molecules arrange into a perfect shape over a distance of billions of molecule diameters although they feel only very local forces?

Qualitatively, the explanation is that local energy minima have enough time to dissolve in favor of globally more efficient structures. For example, assume that a cluster of a dozen molecules approaches a small perfect crystal that already consists of thousands of molecules. Then with enough time the cluster will dissolve and its molecules can attach to the crystal. More formally, within a reasonable model of the system one can show that if cooling is sufficiently slow, the system reaches *thermal equilibrium* at every temperature. Equilibrium at temperature T means that a state $\mathbf{x}$ of the system with energy $E_\mathbf{x}$ is assumed with probability $\frac{\exp -(E_\mathbf{x}/T)}{\sum_{\mathbf{y}\in\mathcal{L}} \exp -(E_\mathbf{y}/T)}$ where T is the temperature of the system in Kelvin multiplied by the *Boltzmann constant* $k_B \approx 1.4 \cdot 10^{-23} J/K$. This energy distribution is called *Boltzmann* distribution. One can see that as $T \to 0$ the probability of states with minimal energy approaches one.

The same mathematics works for an abstract system corresponding to an optimization problem. We identify the cost function f with the energy of the system and a feasible solution with the state of the system. One can show that the system approaches a Boltzmann distribution for a quite general class of neighborhoods and the following rules for choosing the next state:

pick $\mathbf{x}'$ *from* $\mathcal{N}(\mathbf{x}) \cap \mathcal{L}$ *uniformly at random*
with probability $\min(1, \exp(\frac{f(\mathbf{x})-f(\mathbf{x}')}{T})$ **do** $\mathbf{x} := \mathbf{x}'$

The above theoretical considerations give some intuition why simulated annealing might work, but they do not provide an implementable algorithm. We have to get rid of two infinities: For every temperature, wait infinitely long to get equilibrium and do that for infinitely many temperatures. Simulated annealing algorithms therefore have to decide on a *cooling schedule*, i.e., how the temperature T should be varied over time. A simple schedule chooses a starting temperature T_0 that is supposed to be just large enough so that all neighbors are accepted. Furthermore, for a given problem instance there is a fixed number of iterations N used for each temperature. The idea is that N should be as small as possible but still allow the system to get close to equilibrium. After every N iterations, T is decreased by multiplying it with a constant $\alpha < 1$. Typically, α is between 0.8 and 0.99. When T has become so small that it is comparable to the smallest possible difference between two feasible solutions, T is finally set to 0, i.e, the annealing process concludes with a hill climbing search.

Better performance can be obtained with *dynamic schedules*. For example, the initial temperature can be determined by starting with a low temperature and increasing it quickly until the fraction of accepted transitions approaches one. Dynamic schedules base their decision how much T should be lowered on the actually observed variation in $f(\mathbf{x})$. If the temperature change is tiny compared to the variation, it has too little effect. If the change is too close or even larger than the variation observed, there is a danger that the system is prematurely forced into a local optimum. The number of steps to be made until the temperature is lowered can be made dependent on the actual number of moves made. Furthermore, one can use a simplified statistical model of the process to estimate when the system approaches equilibrium. The details of dynamic schedules are beyond the scope of this exposition.

Finally let us look at a concrete example.

Graph Coloring Using Simulated Annealing

To make a meaningful study of simulated annealing, we employ a more difficult problem than the knapsack problem: For an undirected graph $G = (V,E)$ find a *node coloring* $c : V \to 1..k$ such that no two adjacent nodes get the same color, i.e., $\forall \{u,v\} \in E : c(u) \neq c(v)$. We want to minimize k.

We will present two approaches to find good approximations to the graph coloring approach using simulated annealing. One is straightforward but requires a generally useful technique to make it work. The other requires some problem specific insight but often yields better solutions. For more details refer to [49].

The Penalty Function Approach

A generally useful idea for local search is to relax some of the constraints on feasible solutions to make the search more flexible and to be able to find a starting solution. In order to get feasible solutions in the end, the objective function is modified to penalize infeasible solutions. The constraints are effectively moved into the objective function.

We explain this idea using graph coloring as an example. Solutions $\mathbf{x}$ may now denote some arbitrary coloring of the nodes even if there are adjacent nodes with the same color. An initial solution is generated by guessing a number of colors needed and coloring the nodes randomly.

A neighbor is generated by picking a random color j and a random node v with this color $\mathbf{x}(v) = j$. Then, a random new color for node v is chosen among all the colors already in use plus one new color. Let $C_i = \{v \in V : \mathbf{x}(v) = i\}$ denote the set of nodes with color i and let $E_i = \{(u,v) \in E \cup C_i \times C_i\}$ denote the set of edges whose incident nodes are illegally both colored with color i. The objective is to minimize

$$f(\mathbf{x}) = 2\sum_i |C_i| \cdot |E_i| - \sum_i |C_i|^2.$$

The first term penalizes illegal edges and the second favors large color classes. Exercise 12.18 asks you to show that this cost function ensures feasible colorings at local optima. Hence, simulated annealing is guaranteed to find a feasible solution even if it
⟹ starts with an illegal coloring. [Bild!]

The Kempe Chain Approach

Now solutions are only legal colorings. The objective function simplifies to $f(\mathbf{x}) = -\sum_i |C_i|^2$. To find a candidate for a new solution, randomly choose two colors i and j and a node v with color $\mathbf{x}(v) = i$. Consider the maximal connected component K of G containing v and nodes with colors i and j. Such a component is called a *Kempe Chain*. Now exchange colors i and j in all nodes contained in K. If we start with a
⟹ legal coloring, the result will be a legal coloring again.[Bild!]

Experimental Results

Johnson et al. [49] have made a detailed study of algorithms for graph coloring with particular emphasis on simulated annealing. The results depend a lot on the structure of the graph. Many of the experiments use *random graphs*. The usual model for an undirected random graph picks each possible edge $\{u,v\}$ with probability p. The edge probability p can be used to control the the expected number $pn(n-1)/2$ of edges in the graph.

For random graphs with 1000 nodes and edge probability 0.5, Kempe chain annealing produced very good colorings given enough time. However, a sophisticated and expensive greedy algorithm, *XRLF*, produces even better solutions in less time. Penalty function annealing performs rather poorly. For very dense random graphs with $p = 0.9$, Kempe chain annealing overtakes XRLF.

For sparser random graphs with edge probability 0.1, penalty function annealing overtakes Kempe chain annealing and can sometimes compete with XRLF.

Another interesting class of random inputs are *random geometric graphs*: Associate the nodes of a graph with random uniformly distributed positions in the unit square $[0,1] \times [0,1]$. Add an edge (u,v) whenever the Euclidean distance of u and v is less than some range r. Such instances might be a good model for an application where nodes are radio transmitters, colors are frequency bands, and edges indicate possible interference between neighboring senders that use the same frequency. For this model, Kempe chain annealing is outclassed by a third annealing strategy not described here.

Interestingly, the following simple greedy heuristics is quite competitive:

- Given a graph $G = (V,E)$, we keep a subset V' of nodes already colored. Initially, $V' = \emptyset$.
- In every step we pick a node $v \in V \setminus V'$ that maximizes $|\{(u,v) \in E : u \in V'\}|$. Node v is then colored with the smallest legal color.

To obtain better colorings than using a single run, one simple takes the best coloring produced by repeated calls of the heuristics using a random way to break ties when selecting a node to be colored.

Exercises

Exercise 12.18 Show that the objective function for graph coloring given in Section 12.5.2 has the property that any local optimum is a correct coloring. Hint: What happens with $f(\mathbf{x})$ if one end of an illegally colored edge is recolored with a fresh color? Prove that the cost function of the penalty function approach does not necessarily have its global optimum at a solution that minimizes the number of colors used.

12.5.3 More on Local Search

The correspondence between principles observed in nature and optimization algorithms like simulated annealing is quite appealing. However, when we have to solve a concrete problem, we are interested in the most effective method available even if we

have to break an analogy. Here we want to summarize a number of refinements that can be used to modify or replace the approaches chosen so far.

Threshold Acceptance: There seems to be nothing magic about the particular form of the acceptance rule of simulated annealing. For example, a simpler yet also successful rule uses the parameter T as a threshold. New states with a value $f(\mathbf{x})$ below the threshold are accepted others are not.

Tabu Lists: Local search algorithms like simulated annealing sometimes tend to return to the same suboptimal solution again and again — they cycle. For example, simulated annealing might have reached the top of a steep hill. Randomization will steer the search away from the optimum but the state may remain on the hill for a long time. *Tabu search* steers away from local optima by keeping a *tabu list* of "solution elements" that should be "avoided" in new solutions for the time being. For example, in graph coloring a search step could change one color of a node v from c to c' and then store the tuple (v,c) in the tabu list to indicate that v should not be colored with color c again as long as (v,c) is in the tabu list. Usually, this tabu condition is not applied if the solution gets improved by coloring node v with color c. Tabu lists are so successful that they can be used as the core technique of an independent variant of local search called *tabu search*.

Restarts: The typical behavior of a well tuned local search algorithm is that it moves to an area in $\mathcal{L}$ with good solutions and explores this area trying to find better and better local optima. However, it might be that there are other, far away areas with much better solutions. In this situation it is a good idea to run the algorithm multiple times with different random starting solutions because it is likely that different starting point will lead to different areas of good solutions. Even if these restarts do not improve the average performance of our algorithm it may make it more robust in the sense that it is less likely to produce grossly suboptimal solutions. Several independent runs are also an easy source of parallelism. Just run the program on different workstations at once.

12.6 Evolutionary Algorithms

Perhaps the most ingenious solutions to real world problems are the adaptation of living beings to their circumstances of living. The theory of evolution tells us that the mechanisms obtaining these solutions is mutation, mating, and survival of the fittest. Let us translate this approach into our abstract framework for optimization problems. A genome describing an individual corresponds to the description of a feasible solution $\mathbf{x}$. We can also associate infeasible solutions with dead or ill individuals. In nature, it is important that there is sufficiently large *population* of living individuals. So, instead of one solution as in local search, we are now working with many feasible solutions at once.

The individuals in a population produce offspring. Because there is only a limited amount of resources, only the individuals best adapted to the environment survive. For optimization this means that feasible solutions are evaluated using the cost function f and only solutions with lower value are likely to be kept in the population.

Even in bacteria which reproduce by cell division, no offspring is identical to its parent. The reason is *mutation*. While a genome is copied, small errors happen. Although mutations usually have an adverse effect on fitness, some also improve fitness. The survival of the fittest means that those individuals with useful mutations will produce more offspring. On the long run, the average fitness of the population increases. An optimization algorithm based on mutation produces new feasible solutions by selecting a solution $\mathbf{x}$ with large fitness $f(\mathbf{x})$, copies it and applies a (more or less random) mutation operator to it. To keep the population size constant, a solution with small fitness f is removed from the population. Such an optimization algorithm can be characterized as many parallel local searches. These local searches are indirectly coupled by survival of the fittest.

In natural evolution, an even more important ingredient is *mating*. Offspring is produced by combining the genetic information of two individuals. The importance of mating is easy to understand if one considers how rare useful mutations are. Therefore it takes much longer to get an individual with two new and useful mutations than it takes to combine two individuals with two different useful mutations.

We now have all the ingredients needed for an evolutionary algorithm. There are many ideas to brew an optimization algorithm from these ingredients. Figure 12.8 presents just one possible framework. The algorithm starts by creating an initial population. Besides the population size N, it must be decided how to build the initial individuals. This process should involve randomness but it might also be useful to use heuristics for constructing some reasonable solutions from the beginning.

To put selection pressure on the population, it is important to base reproduction success on the fitness on the individuals. However, usually it is not desirable to draw a hard line and only use the fittest individuals because this might lead to a too uniform population and incest. Instead, one draws reproduction partners randomly and only biases the selection by choosing a higher selection probability for fitter individuals. An important design decision is how to fix these probabilities. One choice might be to sort the individuals by fitness and then to define $p(\mathbf{x}_i)$ as some decreasing function of the rank of $\mathbf{x}_i$ in the sorted order. This indirect approach has the advantage that it is independent on the shape of f and that it is equally distinctive in the beginning when fitness differences are large as in the end when fitness differences are usually small. The most critical operation is mate$(\mathbf{x}_i,\mathbf{x}_j)$ which produces two new offspring from two ancestors. The "canonical" mating operation is called crossover: Individuals are assumed to be represented by a string of k bits in such a way that every k-bit string

Create an initial population $\mathsf{pop} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
while not finished **do**
  compute probabilities $p(\mathbf{x}_1), \ldots, p(\mathbf{x}_N)$ based on the fitness values
  randomly select $N/2$ pairs of individuals
  $\mathsf{pop}' = \emptyset$ // new population
  for *each selected pair* $(\mathbf{x}_i, \mathbf{x}_j)$ **do**
    mutate $\mathbf{x}_i$ with probability p_{mutation}
    mutate $\mathbf{x}_j$ with probability p_{mutation}
    if a biased random coin throw shows 'head' **then** // with probability p_{mate}
      $\mathsf{pop}' := \mathsf{pop}' \cup \text{mate}(\mathbf{x}_i, \mathbf{x}_j)$
    else
      $\mathsf{pop}' := \mathsf{pop}' \cup \{\mathbf{x}_i, \mathbf{x}_j\}$
  $\mathsf{pop} := \mathsf{pop}'$
  optionally apply hill climbing to new individuals

Figure 12.8: A generic evolutionary algorithm.

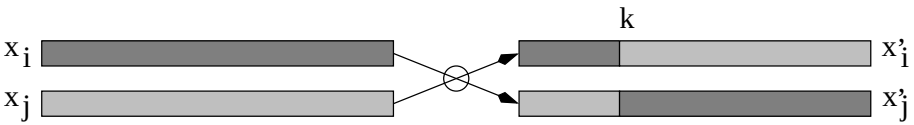


Figure 12.9: The crossover operator.

represents a feasible solution. Then crossover consists of choosing a position k' and producing a new individual $\mathbf{x}'_i$ from bits 0 through $k'-1$ from $\mathbf{x}_i$ and bits k' through of $k-1$ of $\mathbf{x}_j$. Conversely, the other new individual is defined by bits 0 through $k'-1$ from $\mathbf{x}_j$ and bits k' through $k-1$ of $\mathbf{x}_i$. Figure 12.9 shows this procedure. The main difficulty with crossover is that it often requires a very careful choice of encoding. Not only must every bit string represent a feasible solution, but also the mating of two individuals with high fitness must have a good chance of producing another individual with reasonably high fitness. Therefore, many of the more successful applications of evolutionary algorithms choose a carefully designed representation and an application specific mating operation. [Graph coloring example mating operation?]

12.7 Implementation Notes

We have seen several generic approaches to optimization that are applicable to a wide variety of problems. When you face a new application you are therefore likely to have the choice between more approaches than you can realistically implement. In a commercial environment you may even have to home in on a single approach quickly.

Here are a few rules of thumb that may help the decision:

- Look for previous approaches to related problems.
- Try black box solvers if this looks promising.
- If problem instances are small, systematic search or dynamic programming may allow you to find optimal solutions.
- If none of the above looks promising, implement a simple prototype solver using a greedy approach or some other simple and fast heuristics. The prototype helps you to understand the problem and might be useful as component of a more sophisticated algorithm.
- Develop a local search algorithm. Focus on a good representation of solutions and how to incorporate application specific knowledge into the searcher. If you have a promising idea for mating operator, you can also consider evolutionary algorithms. Use randomization and restarts to make the results more robust.

packages like CPLEX[more][3] or free packages like SoPlex[4] are used. Ernsts Paket as successor of Abacus. Constraint programming various Prologs, ILOG.

12.8 Further Findings

History of LP and dynamic programming

More information on linear programming can for example be found in textbooks [13] or on the web.[5]

MST and SSSP as linear programs, flows??

Bixbies runtime studies

matroids

scheduling literatur

nonclairvoyant scheduling

[3] http://www.cplex.com/

[4] http://www.zib.de/Optimization/Software/Soplex/

[5] http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html

better algorithms for identical machines LPT, FirstFitDecreasing, FPTAS, PTAS

interior point solvers — also local search but not (quite) hill climbing

A more elaborate analysis allows us to conclude that even after finitely many steps at a given temperature and using finitely many temperatures, an optimal solution is found with high probability. However, the number of iterations needed to obtain such guarantees is ridiculously high. For example, for the Traveling Salesman Problem, a bound of $O\left(n^{n^{2n-1}}\right)$ iterations can be shown. Just trying all possible tours needs "only" $(n-1)! \ll n^n$ iterations.

Lagrange relaxation as a general theory how to move constraints into the objective function?

ellipsoid method

Chapter 13

Summary: Tools and Techniques for Algorithm Design

When you design an algorithm, it is usually a good idea to first design it in terms of high level operations on sets and sequences. Section 13.1 reviews some basic approaches that we have seen. More likely than not, you can find an appropriate implementation among the many data structures that we have seen in this book. Section 13.2 gives some guidance how to choose among the many possibilities for representing sets and sequences. A data structure for maintaining partitions of sets is described in Section 11.4. To represent more application specific data structures you might still use patterns you have seen: Adjacency lists, adjacency arrays, and bit matrices for graph-like data structures (Chapter 8). Fixed degree trees can be implicityly defined in an array (Chapter 6.1). With bounded degree trees can use downward pointers (Chapter 7), and with variable degree they could use parent pointers (Chapter 11.4) or sibling pointers (Chapter **??**).

13.1 Generic Techniques

Divide-and-Conquer:

Perhaps the most natural way to solve a problem is to break it down into smaller problems, solve the smaller problems recursively, and then merge them to an overall solution. Quicksort (Section 5.4) and mergesort (Section 5.2) are typical examples for

this *divide-and-conquer* approach. These two quite different solutions for the same problem are also a good example that many roads lead to Rome[check]. For example, quicksort has an expensive divide-stragety and a trivial conquer-stragegy and it is the other way round for mergesort.

Dynamic Programming:

Whereas divide-and-conquer procedes in a top-down fashion, dynamic programming (Section 12.3) works bottom up — systematically construct solutions for small problem instances and assemble them to solutions for larger and larger inputs. Since dynamic programming is less goal directed than divide-and-conquer, it can get expensive in time and space. We might ask "why not always use a top-down approach?" The reason is that a naive top down approach might solve the same subproblems over and over again. [example: fibonacchi numbers?]

Randomization:

Making random decisions in an algorithm helps when there are many good answers and not too many bad ones and when it would be expensive to definitively distinguish good and bad. For example, in the analysis of the quick-sort like selection algorithm from Section 5.5 on third of all possible splitters were "good". Moreover, finding out whether a splitter is good is not much cheaper than simply using it for the divide step of the divide-and-conquer algorithm.

Precomputation:

[preconditioning for APSP?]
[nice example from Brassard Bradlay]

13.2 Data Structures for Sets

[move sth to intro?]

Table 13.1 gives an overview of the most important operations on sets and six different representations. As a rule of thumb, the data structures compared in Table 13.1 get more complicated but also more powerful from left to right. Hence, you get the most effeictive solution from the leftmost column that supports the required operations efficiently.

[ps: Bei mir hat ein addrssable priority queue kein $O(logn)$ time concat or split, weil die Datenstrukturen, die das untersttzen eigentlich Suchbaeume sind und kein effizientes merge oder decreaseKey unterstuetzen.] The simplest

Table 13.1: Basic operations on a *Set* M of *Element*s and their complexity (with an implicit $O(\cdot)$) for six different implementations. Assumptions: e an *Element*, h is an *Element Handle*, k a *Key*, and $n = |M|$. 'a' stands for an amortized bound, 'r' for a randomized algorithm. '$-$' means that the representation is not helpful for implementing the operation. All data structures support **forall** $e \in M \ldots$ in time $O(n)$ and $|\cdot|$ in constant time.

Procedure *insert*(e) $M := M \cup \{e\}$
Procedure *insertAfter*(h,e) **assert** $h = \max\{e' \in M : e > e'\}$; $M := M \cup \{e_2\}$
Procedure *build*$(\{e_1,\ldots,e_n\})$ $M := \{e_1,\ldots,e_n\}$
Function *deleteMin* $e := \min M$; $M := M \setminus \{e\}$; **return** e
Function *remove*(k) $\{e\} := \{e \in M : key(e) = k\}$; $M := M \setminus \{e\}$; **return** e
Function *remove*(h) $e := h$; $M := M \setminus \{e\}$; **return** e
Procedure *decreaseKey*(h,k) **assert** $key(h) \geq x$; $key(h) := k$
Function *find*(k) : *Handle* $\{h\} := \{e : key(e) = k\}$; **return** h
Function *locate*(k) : *Handle* $h := \min\{e : key(e) \geq k\}$; **return** h
Procedure *merge*(M') $M := M \cup M'$
Procedure *concat*(M') **assert** $\max M < \min M'$; $M := M \cup M'$
Function *split*(h) : *Set* $M' := \{e \in M : e \leq h\}$; $M := M \setminus M'$; **return** M'
Function *findNext*(h) : *Handle* **return** $\min\{e \in M : e > h\}$
Function *select*$(i : \mathbb{N})$ $\{e_1,\ldots,e_n\} := M$; **return** e_i

Operation	*List*	*HashTable*	sort-array	*PQ*	*APQ*	(a,b)-*tree*
insert	1	1^r	–	1	1	$\log n$
insertAfter	1	1^r	–	1	1	1^a
build	n	n	$n \log n$	n	n	$n \log n$
deleteMin	–	–	1	$\log n$	$\log n$	1^a
remove(*Key*)	–	1	–	–	–	$\log n$
remove(*Handle*)	1	1	–	–	$\log n$	1^a
decreaseKey	1	–	–	–	1^a	$\log n$
find	–	1^r	$\log n$	–	–	$\log n$
locate	–	–	$\log n$	–	–	$\log n$
merge	1	–	n	–	$\log n$	n
concat	1	–	n	–	–	$\log n$
split	–	–	1	–	–	$\log n$
findNext	–	–	1	–	–	1
select	–	–	1	–	–	$\log n$

representations just accumulates inserted elements in the order they arrive using a sequence data structure like (cyclic) (unbounded) arrays or (doubly) linked lists (Chapter 3). Table 13.1 contains a column for doubly linked lists but often even arrays do the job. For a more detailed comparison and additional operations for sequences you should refer to Section 3.4.

Hash tables (Chapter 4) are better if you frequently need to find, remove, or change arbitrary elements of the set without knowing their position in the data structure. Hash tables are very fast but have a few restrictions. They give you only probabilistic performance guarantees for some operations (there is some flexibility which operations). You need a hash function for the *Element* type (or better a universal family of hash functions). Hash tables are not useful if the application exploits a linear ordering of elements given by a key.

If you need to process elements in the order of key values or if you want to find the elements closest to a given key value, the simplest solution is a sorted array (Chapter 5). Now you can find or locate elements in logarithmic time using binary $\Longrightarrow$ search[where]. You can also merge sorted arrays in linear time, find elements of given rank easily and split the array into subarrays of disjoint key ranges. The main restriction for sorted arrays is that insertion and deletion is expensive.

Priority queues (Chapter 4) are a somewhat specialized yet frequently needed dynamic data structure. They support insertions of arbitrary elements and deletion of the smallest element. You can additionally remove elements from adressable priority queues (column APQ in Table lreftab:operations) and some variants allow you to merge two priority queues in logarithmic time. Fibonacchi heaps support *decreasKey* in constant amortized time.

Search trees like (a,b)-trees (Chapter 7) support almost all conceivable elementwise operations in logarithmic time or faster. They even support the operations *split* and *concat* in logarithmic time although they affect the entire sorted sequence. Search trees can be augmented with additional information to support more information. For example, it is easy to support extraction of the k-th smallest element (Function *select*) in logarithmic time if each subtree knows its size.

$\Longrightarrow$ [so far the only place where “this” is needed?] [todo: nicer alignment of
$\Longrightarrow$ setops]

[PS: more Possible appendixes: memory management, recurrences] $\Longleftarrow$

Appendix A

Notation

A.1 General Mathematical Notation

$\{e_0,\ldots,e_{n-1}\}$**:** Set containing the elements e_0,...,e_{n-1}.

$\{e : P(e)\}$**:** Set of all elements the fulfill the predicate P.

$\langle e_0,\ldots,e_{n-1}\rangle$**:** Sequence containing the elements e_0,...,e_{n-1}.

$\langle e \in S' : P(e)\rangle$**:** Subsequence of all elements of sequence S' that fulfill the predicate P.[bei Pseudocode?] ⟸

$\perp$**:** An undefined value.

$(-)\infty$**:** (Minus) infinity.

$\mathbb{N}$**:** Nonegative integers, $\mathbb{N} = \{0,1,2,\ldots\}$. Includes the zero! [check] ⟸

$\mathbb{N}_+$**:** Positive integers, $\mathbb{N} = \{1,2,\ldots\}$.

|, &, <<, >>, $\oplus$**:** Bit-wise 'or', 'and', right-shift, left-shift, and exclusive-or respectively.

$\sum_{i=1}^{n} a_i = \sum_{i\in\{1,\ldots,n\}} a_i$**:** $= a_1 + a_2 + \cdots + a_n$

$\prod_{i=1}^{n} a_i = \prod_{i\in\{1,\ldots,n\}} a_i$**:** $= a_1 \cdot a_2 \cdots a_n$

$n!$**:** $= \prod_{i=1}^{n} i$ — the *factorial* of n.

div**:** Integer division. $c = m$ div$n$ is the largest nonnegative integer such that $m - cn \geq 0$.[durch $\lfloor/\rfloor$ ersetzen?] ⟸

mod**:** Modular arithmetics, $m \bmod n = m - n(mrmdivn)$.

$a \equiv b (\bmod m)$**:** a and b are congruent modulo m, i.e., $\exists i \in \mathbb{Z} : a + im = b$.

$\prec$**:** Some ordering relation. In Section 9.2 it denotes the order in which nodes are marked during depth first search.

$i..j$**:** Short hand for $\{i, i+1, \ldots, j\}$.

A^B**:** When A and B are sets this is the set of all functions mapping B to A.

$A \times B$**:** The set of pairs (a,b) with $a \in A$ and $b \in B$.

$\lfloor x \rfloor$ The largest integer $\leq x$.

$\lceil x \rceil$ The smallest integer $\geq x$.

antisymmetric: A relation $\sim$ is *antisymmetric* with respect to an equivalence relation $\equiv$ if for all a and b, $a \sim b$ and $b \sim a$ implies $a \equiv c$. The no equivalence relation
$\Longrightarrow$ is specified, $\equiv$ is the equality relation $=$.[check. needed for sorting]

equivalence relation: A transitive, reflexive, symmetric relation.

false: a shorthand for the value 0.

field: In algebra a set of elements that support addition, subtraction, multiplication, and division by nonzero elements. Addition and multiplication are associative, commutative, and have neutral elements analogous to zero and one for the integers.

H_n**:** $= \sum_{i=1}^{n} 1/i$ the n-th *harmonic number*. See also Equation (A.8).

iff: A shorthand for "if and only if".

$\Longrightarrow$ **lexicographic order:** The most common way to extend a total order[cross ref] on a set of elements to tuples, strings, or sequences of these elements. We have $\langle a_1, a_2, \ldots, a_k \rangle < \langle b_1, b_2, \ldots, b_k \rangle$ if and only if $a_1 < b_1$ or $a_1 = b_1$ and $\langle a_2, \ldots, a_k \rangle < \langle b_2, \ldots, b_k \rangle$

$\log x$ The logarithm base two of x, $\log_2 x$.

median: An element with rank $\lceil n/2 \rceil$ among n elements.

multiplicative inverse: If an object x is multiplied with a *multiplicative inverse* x^{-1} of x, we get $x \cdot x^{-1} = 1$ — the neutral element of multiplication. In particular, in a *field* every elements but the zero (the neutral element of addition) has a unique multiplicative inverse.

$\Longrightarrow$ Ω**:** The sample space in probabilty theory or the set of functions [todo!!!].

prime numbers: $n \in \mathbb{N}$ is a prime number if there are no integers $a, b > 1$ such that $n = a \cdot b$.

Rank: A one-to-one mapping $r : Element \to 1..n$ is a ranking functions for elements of a sequence $\langle e_1, \ldots, e_n \rangle$ if $r(x) < r(y)$ whenever $x < y$.

reflexive: A relation $\sim \subseteq A \times A$ is reflexive if $\forall a \in A : (a,a) \in R$.

relation: A set of pairs R. Often we write relations as operators, e.g., if $\sim$ is relation, $a \sim b$ means $(a,b) \in \sim$.

strict weak order: A relation that is like a total order except the antisymmetry only needs to hold with respect to some equivalence relation $\equiv$ that is not necessarily the identity (see also `http://www.sgi.com/tech/stl/LessThanComparable.html`).

symmetric relation: A relation $\sim$ is *antisymmetric* if for all a and b, $a \sim b$ implies $b \sim a$.

total order: A reflexive, transitive, antisymmetric relation.

transitive: A relation $\sim$ is *transitive* if for all a, b, c, $a \sim b$ and $b \sim c$ imply $a \sim c$.

true A shorthand for the value 1.

A.2 Some Probability Theory

The basis of any argument in probability theory is a *sample space* Ω. For example, if we want to describe what happens if we roll two dice, we would probably use the 36 element sample space $\{1, \ldots, 6\} \times \{1, \ldots, 6\}$. In a *random experiment*, any element of Ω is chosen with the elementary *probability* $p = 1/|\Omega|$. More generally, the probability of an *event* $\mathcal{E} \subseteq \Omega$ is the sum of the probabilities of its elements, i.e, $\text{prob}(\mathcal{E}) = |\mathcal{E}|/|\Omega|$. [conditional probability needed?] A random variable is a $\Longleftarrow$ mapping from elements of the sample space to a value we obtain when this element of the sample space is drawn. For example, X could give the number shown by the first dice[check: wuerfel] and Y could give the number shown by the second dice. $\Longleftarrow$ Random variables are usually denoted by capital letters to differentiate them from plain values.

We can define new random variables as expressions involving other random variables and ordinary values. For example, if X and Y are random variables, then $(X + Y)(\omega) = X(\omega) + Y(\omega)$, $(X \cdot Y)(\omega) = X(\omega) \cdot Y(\omega)$, $(X + 3)(\omega) = X(\omega) + 3$.

Events are often specified by predicates involving random variables. For example, we have $\text{prob}(X \leq 2) = 1/3$ or $\text{prob}(X+Y=11) = \text{prob}(\{(5,6),(6,5)\}) = 1/18$.

Indicator random variables are random variables that can only take the values zero and one. Indicator variables are a useful tool for the probabilistic analysis of algorithms because they encode the behavior of complex algorithms into very simple mathematical objects.

The *expected* value of a random variable $X : \Omega \to A$ is

$$\text{E}[X] = \sum_{x \in A} x \cdot \text{prob}(X = x) \quad . \tag{A.1}$$

In our example $\text{E}[X] = \frac{1+2+3+4+5+6}{6} = \frac{21}{6} = 3.5$. Note that for an indicator random variable Z we simply have $\text{E}[Z] = \text{prob}(Z = 1)$.

Often we are interested in the expectation of a random variable that is defined in terms of other random variables. This is easy for sums due to *the linearity of expectation* of random variables: For any two random variables X and Y,

$$\text{E}[X+Y] = \text{E}[X] + \text{E}[Y] \quad . \tag{A.2}$$

In contrast, $\text{E}[X \cdot Y] = \text{E}[X] \cdot \text{E}[Y]$ only if X and Y are *independent*. Random variables $X_1, \ldots, X_k$ are independent if and only if

$$\forall x_1, \ldots, x_k : \text{prob}(X_1 = x_1 \wedge \cdots \wedge X_k = x_k) = \text{prob}(X_1 = x_1) \cdots \text{prob}(X_k = x_k) \tag{A.3}$$

[exercise?: let A,B denote independent indicator random variables. Let $X =$
$\Longrightarrow$ $A \oplus B$. Show that X, A, B are pairwise independent, yet not independent.]

We will often work with a sum $X = X_1 + \cdots + X_n$ of n indicator random variables $X_1, \ldots, X_n$. The sum X is easy to handle if $X_1, \ldots, X_n$ are independent. In particular, there are strong *tail bounds* that bound the probaility of large deviations from the expectation of X. We will only use the following variant of a *Chernoff bound*:

$$\text{prob}(X < (1-\varepsilon)\text{E}[X]) \leq e^{-\varepsilon^2 \text{E}[X]/2} \tag{A.4}$$

If the indicator random variables are also identically distributed with $\text{prob}(X_i = 1) = p$, X is *binomially distributed*,

$$\text{prob}(X = i) = \binom{n}{i} p^i (1-p)^(n-i) \quad . \tag{A.5}$$

A.3 Useful Formulas

$\Longrightarrow$ [separate section for recurrences? todo: linear recurrence, master theorem]

$$\binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k \tag{A.6}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{A.7}$$

$[\sum_{i=1}^{n} i^2]$ $\Longleftarrow$

$$\ln n \leq H_n = \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \tag{A.8}$$

Exercise 2.6 gives a hint how to prove this relation.

$$\sum_{i=0}^{n-1} q^i = \frac{1-q^n}{1-q} \text{ for } q \neq 1 \tag{A.9}$$

Stirling

$$\left(\frac{n}{e}\right)^n \leq n! = \left(1 + O\left(\frac{1}{n}\right)\right) \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ Stirling's equation} \tag{A.10}$$

Bibliography

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[3] R. Ahuja, K. Mehlhorn, J. Orlin, and R. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 3(2):213–223, 1990.

[4] R. K. Ahuja, R. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, pages 74–93, 1998.

[6] A. Andersson and M. Thorup. A pragmatic implementation of monotone priority queues. In *DIMACS'96 implementation challenge*, 1996.

[7] F. Annexstein, M. Baumslag, and A. Rosenberg. Group action graphs and parallel architectures. *SIAM Journal on Computing*, 19(3):544–569, 1990.

[8] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173 – 189, 1972.

[9] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious *b*-trees. In IEEE, editor, *41st IEEE Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[10] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software Practice and Experience*, 23(11):1249–1265, 1993.

[11] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, pages 643–647, 1979.

[12] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In ACM, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, January 5–7, 1997*, pages 360–369, New York, NY 10036, USA, 1997. ACM Press.

[13] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[14] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 3–16, 1991.

[15] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. of Computer and System Sciences*, 7(4):448, 1972.

[16] O. Boruvka. O jistém problému minimálním. *Pràce, Moravské Prirodovedecké Spolecnosti*, pages 1–58, 1926.

[17] G. S. Brodal. Worst-case efficient priority queues. In *Proc. 7th Symposium on Discrete Algorithms*, pages 52–58, 1996.

[18] M. Brown and R. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9:594–614, 1980.

[19] R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[20] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, Apr. 1979.

[21] J.-C. Chen. Proportion extend sort. *SIAM Journal on Computing*, 31(1):323–330, 2001.

[22] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. In D. D. Sleator, editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 516–525. ACM Press, 1994.

[23] E. G. Coffman, M. R. G. Jr., , and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 46–93. PWS, 1997.

[24] D. Cohen-Or, D. Levin, and O. Remez. rogressive compression of arbitrary triangular meshes. In *Proc. IEEE Visualization*, pages 67–72, 1999.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[26] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 2., rev. ed. edition, 2000.

[27] R. Dementiev, L. Kettner, J. Mehnert, and P. Sanders. Engineering a sorted list data structure for 32 bit keys. In *Workshop on Algorithm Engineering & Experiments*, New Orleans, 2004.

[28] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[29] L. Devroye. A note on the height of binary search trees. *Journal of the ACM*, 33:289–498, 1986.

[30] R. B. Dial. Shortest-path forest with topological ordering. *Commun. ACM*, 12(11):632–633, Nov. 1969.

[31] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *5th ACM Symposium on Parallel Algorithms and Architectures*, pages 110–119, Velen, Germany, June 30–July 2, 1993. SIGACT and SIGARCH.

[32] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[33] R. Fleischer. A tight lower bound for the worst case of Bottom-Up-Heapsort. *Algorithmica*, 11(2):104–115, Feb. 1994.

[34] E. Fredkin. Trie memory. *CACM*, 3:490–499, 1960.

[35] M. L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, July 1999.

[36] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[37] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[38] Goldberg. A simple shortest path algorithm with linear average time. In *ESA: Annual European Symposium on Algorithms*, 2001. INCOMPLETE.

[39] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, New York, 2 edition, 1991.

[40] G. Graefe and P.-A. Larson. B-tree indexes and cpu caches. In *ICDE*, pages 349–358. IEEE, 2001.

[41] J. F. Grantham and C. Pomerance. Prime numbers. In K. H. Rosen, editor, *Handbook of Discrete and Combinatorial Mathematics*, chapter 4.4, pages 236–254. CRC Press, 2000.

[42] R. D. Hofstadter. Metamagical themas. *Scientific American*, (2):16–22, 1983.

[43] S. Huddlestone and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[44] J. Iacono. Improved upper bounds for pairing heaps. In *7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *LNCS*, pages 32–45. Springer, 2000.

[45] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, volume 115 of *LNCS*, pages 417–431, Acre, Israel, July 1981. Springer.

[46] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[47] K. Jensen and N. Wirth. *Pascal User Manual and Report. ISO Pascal Standard*. Springer, 1991.

[48] T. Jiang, M. Li, and P. Vitányi. Average-case complexity of shellsort. In *ICALP*, number 1644 in LNCS, pages 453–462, 1999.

[49] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: Experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[50] H. Kaplan and R. E. Tarjan. New heap data structures. Technical Report TR-597-99, Princeton University, 1999.

[51] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics—Doklady*, 7(7):595–596, Jan. 1963.

[52] D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *J. Assoc. Comput. Mach.*, 42:321–329, 1995.

[53] J. Katajainen and B. B. Mortensen. Experiences with the design and implementation of space-efficient deque. In *Workshop on Algorithm Engineering*, volume 2141 of *LNCS*, pages 39–50. Springer, 2001.

[54] I. Katriel, P. Sanders, and J. L. Träff. A practical minimum spanning tree algorithm using the cycle property. Technical Report MPI-I-2002-1-003, MPI Informatik, Germany, October 2002.

[55] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

[56] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.

[57] D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[58] D. E. Knuth. *MMIXware: A RISC Computer for the Third Millennium*. Number 1750 in LNCS. Springer, 1999.

[59] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[60] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[61] S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations*. Wiley, 1990.

[62] C. Martínez and S. Roura. Optimal sampling strategies in Quicksort and Quickselect. *SIAM Journal on Computing*, 31(3):683–705, June 2002.

[63] C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. Using finite experiments to study asymptotic performance. In *Experimental Algorithmics — From Algorithm Design to Robust and Efficient Software*, volume 2547 of *LNCS*, pages 1–23. Springer, 2002.

[64] K. Mehlhorn. *Data Structures and Algorithms, Vol. I — Sorting and Searching*. EATCS Monographs on Theoretical CS. Springer-Verlag, Berlin/Heidelberg, Germany, 1984.

[65] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27(3):125–128, Mar. 1988.

[66] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[67] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.

[68] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[69] K. Mehlhorn, V. Priebe, G. Schäfer, and N. Sivadasan. All-pairs shortest-paths computation in the presence of negative cycles. IPL, to appear, `www.mpi-sb.mpg.de/~mehlhorn/ftp/shortpath.ps`, 2000.

[70] K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.

[71] R. Mendelson and U. Z. R. E. Tarjan, M. Thorup. Melding priority queues. In *9th Scandinavian Workshop on Algorithm Theory*, pages 223–235, 2004.

[72] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.

[73] B. M. E. Moret and H. D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Workshop Algorithms and Data Structures (WADS)*, number 519 in LNCS, pages 400–411. Springer, Aug. 1991.

[74] J. Nešetřil, H. Milková, and H. Nešetřilová. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233, 2001.

[75] K. S. Neubert. The flashsort1 algorithm. *Dr. Dobb's Journal*, pages 123–125, February 1998.

[76] J. v. Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945. `http://www.histech.rwth-aachen.de/www/quellen/vnedvac.pdf`.

[77] K. Noshita. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms*, 6(3):400–408, 1985.

[78] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. In *27th ICALP*, volume 1853 of *LNCS*, pages 49–60. Springer, 2000.

[79] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer, 1985.

[80] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[81] P. Sanders and S. Winkel. Super scalar sample sort. In *12th European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004.

[82] R. Santos and F. Seidel. A better upper bound on the number of triangulations of a planar point set. *Journal of Combinatorial Theory Series A*, 102(1):186–193, 2003.

[83] R. Schaffer and R. Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15:76–100, 1993. Also known as TR CS-TR-330-91, Princeton University, January 1991.

[84] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971.

[85] R. Sedgewick. Analysis of shellsort and related algorithms. *LNCS*, 1136:1–11, 1996.

[86] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[87] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[88] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[89] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[90] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[91] R. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[92] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *35th ACM Symposium on Theory of Computing*, pages 149–158, 2004.

[93] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. *Information Processing Letters*, 6(3):80–82, 1977.

[94] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.

[95] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[96] I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Comput. Sci.*, 118:81–98, 1993.

[97] M. T. Y. Han. Integer sorting in $O\left(n\sqrt{\log\log n}\right)$ expected time and linear space. In *42nd Symposium on Foundations of Computer Science*, pages 135–144, 2002.