Introduction to Algorithms and Data Structures

Markus Bläser
Saarland University

Draft—Thursday 22, 2015 and forever

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Contents

1 Introduction

1.1 Binary search

Consider the following—nowadays pretty outdated—example: You are looking for a particular CD of a particular interpret. Because you are well-bred, you store your CD alphabetically sorted in your shelf. So you can scan all your CDs from the left to the right until you find the one that you are looking for. If your collection were not alphabetically sorted, then this would be the best that you can do. But since your CDs are sorted, you can look at the CD (roughly) in the middle of the shelf. If the CD you are looking for is alphabetically smaller, then you know that it is in the left half of your collection. Otherwise, it is in the right half of the collection. In this way, you halved the search space by looking at only one CD. This is a huge improvement over the scanning from left to right method. Here looking at one CD decreases your search space only by one.

From this example, we will learn a lot about the spirit of this course. We will

- find algorithmic solutions that are faster than the "obvious" ones,
- prove that the algorithms terminate and produce the correct result, and
- rigorously analyze the running time of the algorithms.

Let us abstract a little bit from the CD example. Our input is an array $a[1..n]$ and the entries in the array are sorted, i.e., $a[1] < a[2] < \cdots < a[n]$. We do not care about the exact type of the array, we only need that we can compare the values. Algorithm 1 implements the method sketched above. The variables ℓ and r store the current left and right border of the search space and m is the index of the current element in the middle. If $a[m] = x$, then we found the element. Otherwise, m becomes the new left or right border.

The text between /* ... */ in line 2 is a *loop invariant*. The loop invariant holds before each iteration of the loop. This is of course not automatically true, we have to design the program in such a way and, of course, we have to prove this.

In the case above, we have to declare what $a[0]$ and $a[n+1]$ is. We will set $a[0] = -\infty$ and $a[n+1] = +\infty$. These are fictitious elements, we need them for the invariant to be true and for our proofs to work, but the algorithm can never access them or compute with the values $-\infty$ or ∞.

Algorithm 1 Binary-search

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x

Output: $\begin{cases} m & \text{if there is an } 1 \leq m \leq n \text{ with } a[m] = x \\ -1 & \text{otherwise} \end{cases}$

```
1: ℓ := 0; r := n + 1;
2: while ℓ + 1 < r do   /* 0 ≤ ℓ < r ≤ n + 1 AND a[ℓ] < x < a[r] */
3:    m := ⌊(ℓ+r)/2⌋;
4:    if a[m] = x then
5:       return m;
6:    if a[m] < x then
7:       ℓ := m
8:    else
9:       r := m;
10: return −1;
```

If we start with this, then before the first iteration of the while loop, our invariant is certainly true. Now we show that if the invariant is true before the current iteration, then it will also be true after the iteration (that is, before the next iteration). Note that $\ell \leq m \leq r$ and in lines 7 and 8, either ℓ or r is set to m. Thus $0 \leq \ell \leq r \leq n+1$ holds after the iteration. If $\ell = r$, then this would mean that $\ell + 1 = r$ before the iteration. But in this case, the loop would not have been executed. Thus the first part of the invariant is true. For the second part, note that it is certainly true in the beginning, since by our convention, $a[0] = -\infty$ and $a[n+1] = +\infty$.[1] Now assume that the invariant was true before the current iteration. If $a[m] = x$, then we stop and there is no next iteration. If $a[m] \neq x$, then in lines 6 to 9, ℓ and r are chosen in such a way that the invariant holds at the end of the iteration.

If the while loop is left, then either $a[m] = x$ or $\ell + 1 = r$. In the first case, the algorithm returns m. In the second case, the algorithm returns -1. But we know that $a[\ell] < x < a[r]$ from our invariant. But since there are no further array elements between $a[\ell]$ and $a[r]$, x is not in the array. Thus the algorithm is correct.

The algorithm also terminates: In each iteration either ℓ is increased or r is reduced. Hence, the algorithm will terminate.

Theorem 1.1 *Binary-search always terminates and returns the correct output.*

Often, it is very convenient to state algorithms in a recursive fashion. While we can transform an iterative algorithm into a recursive one and vice

[1] If you still feel uneasy with this convention, you can replace it by a case distinction. The second part of the invariant then reads $((\ell = 0) \vee (a[\ell] < x)) \wedge ((r = n+1) \vee (x < a[r]))$. I hope that this convinces you of the usefulness of the fictitious elements.

versa, the recursive one is often easier to analyze, but might be slower in practice.

Algorithm 2 Binary-search (recursive)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x

Output: $\begin{cases} m & \text{if there is an } 1 \leq m \leq n \text{ with } a[m] = x \\ -1 & \text{otherwise} \end{cases}$

```
 1: if n = 0 then
 2:    return −1;
 3: m := ⌊(n+1)/2⌋;
 4: if a[m] = x then
 5:    return m;
 6: if a[m] < x then
 7:    s := Binary-search(a[m + 1..n], x);
 8:    if s = −1 then
 9:       return −1
10:    else
11:       return m + s;
12: else
13:    return Binary-search(a[1..m − 1], x);
```

If $n = 0$, then our array is empty and there is nothing to do. Otherwise, we compute m as before. (0 and $n+1$ are the borders.) If $a[m]$ is our element, we return m. If not, we call Binary search recursively. In lines 7–11, we deal with the case that x is (potentially) in the second half of the array. If the recursive call returns -1, that is, x was not found, then we return -1. If something else is returned, that is, x was found, then we return $m + s$, since we searched in the second half. In line 13, we deal with the case that x is (potentially) in the first half. Here we can just pass the value s, since no index shift is required.

Note that we rely on an intelligent compiler (you!) when writing our Pseudocode. For instance, Binary-search$(a[m + 1...n], x)$ calls the routine with the subarray of length $n - m$. Inside the recursive call, the entries are again indexed from $1, \ldots, n - m$. Furthermore, we always know the length of the array.

1.2 Machine model

We formulated the algorithms for binary search in *Pseudocode*. Pseudocode has no precise syntax. There are while and for loops, if-then-else statements and returns statement. We only need to informally declare variables. For the moment, the only built-in type are integers and the Boolean values true and false. We saw comparison operations like $=$ or $<$, we will allow arithmetic

operations like $+$, $-$, $\cdot$, $/$, and rem, where the latter two are integer division and remainder. $:=$ denotes the assignment operator. Pseudocode is very intuitive. For instance, you should be able to transfer the Pseudocode of binary search into real code of the programming language of your choice without any problems.

When we want to analyze the running time of an algorithm written in Pseudocode, we have to charge costs for different operations. Since we want to abstract from different programming languages and architectures, there is no point in discussing whether an addition is more costly than a multiplication, so all statements in Pseudocode take one unit of time.[2]

There is one exception: We have to bound the values that we can store in integer variables. Otherwise we could create huge numbers by repeated squaring: We start with 2, multiply it with itself, we get 4, multiply it with itself and get 16 and so on. With n multiplications, we can create the number of 2^{2^n}, which has an exponential number of bits. We could even have efficient algorithms for factoring integers, a problems that is supposed to be hard; modern cryptographic systems are based on this assumption. We can even have efficient algorithms for so-called NP-hard problems, which you will encounter in the lecture "Grundzüge der Theoretischen Informatik". (And even this is not the end of the story...) So we will require that our integer variables can only store values that are polynomially large in the size of the data that we want to process. In the binary search algorithm, n is the size of the input (or $n+1$ if you also want to count x.) Our integers can hold values up to some polynomial in n (of our choice), say, n^2. This is enough to address all the elements of the array, so this is a reasonable model. If we want to have larger integers, we have to simulate them by an array of small integers (like bignum packages do this in reality). In our binary search example, the values of the variables ℓ, m, and r are always bounded by $n+1$. The values of the $a[i]$ and x are part of the input. If they were large, then n would not be an appropriate measure for the size of the input but rather the sum of the bit lengths of the array elements and x. But the array elements and x play a special role in binary search, since we only access them via comparisons. So their actual size does not matter at all, we could even assume that they are real numbers or something else, as long as there is a total order on them.

The underlying architecture is the so-called *Word RAM*, RAM stands for *random access machine*, the prefix Word indicates that we can store numbers that are polynomially large in the input size in the cells. This is a mathematical abstraction of the *von Neumann architecture*, a pioneering computer architecture that still influences modern architectures. But since

[2]This oversimplification turned out to be very fruitful, since we now can concentrate on finding better algorithms and not trying to avoid multiplications because they are so costly. Nevertheless, if we want to do an implementation in a concrete programming language of an algorithm given in Pseudocode, you have to think about such details.

Pseudocode is so intuitive, we do not need to go into further details here. If you want to learn more about the underlying model, it is a good idea to read Section 2.2 of the book by Mehlhorn and Sanders.

Modern computer architectures are more complicated, they have different levels of caches, which one should exploit when implementing these algorithms. But so-called *external memory algorithms* are beyond the scope of this lecture.

1.3 Asymptotic growth of functions

Next, we want to analyze the running time of binary search. Usually we do not care too much about constant factors. First of all, this has been a very fruitful approach in the past decades. It is usually much better to first care about the asymptotic behavior. For example, if looking at the CD in the middle of the shelf takes three times longer than looking at a CD when you scan them one after an other, because you first have to determine the middle, even for small-sized CD collections, binary search is preferable.

Second, it is very hard to precisely calculate the exact running times. Do we count the number of actual cycles? The number of statements? Do different statements have different costs? By only caring about the asymptotic behavior, we are abstracting away from these details and we can concentrate on finding good abstract algorithms. Of course, when it comes down to implementing an algorithm, we have to think about the underlying architecture. And it might turn out that for your concrete architecture and typical input size, an asymptotically slower algorithm is preferable.

Let $\mathbb{R}_{\geq 0}$ denote the set of all non-negative real numbers. We call a function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ *asymptotically positive* if it is positive almost everywhere, i.e., $f(n) = 0$ only holds for finitely many n. Whenever we investigate the asymptotic growth of functions, we will tacitly assume that they are asymptotically positive, which is fine, since we are estimating the running time of algorithms. The number of steps that an algorithm can make is always in integer, so running times are always functions $\mathbb{N} \to \mathbb{N}$. By allowing functions $\mathbb{N} \to \mathbb{R}_{\geq 0}$, some calculations will become easier .

Definition 1.2 *Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function that is asymptotically positive.*

1. $\mathrm{O}(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid$ *there are $c > 0$ and $n_0 \in \mathbb{N}$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$*$\}$,
2. $\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid$ *there are $c > 0$ and $n_0 \in \mathbb{N}$ such that $g(n) \geq cf(n)$ for all $n \geq n_0$*$\}$,
3. $\Theta(f) = \mathrm{O}(f) \cap \Omega(f)$,
4. $\mathrm{o}(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid$ *for all $c > 0$ there is an $n_0 \in \mathbb{N}$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$*$\}$,

5. $\omega(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid$ *for all* $c > 0$ *there is an* $n_0 \in \mathbb{N}$ *such that* $g(n) \geq cf(n)$ *for all* $n \geq n_0\}$.

Above c can be an arbitrary positive real number.

If $g \in \mathrm{O}(f)$, then g does not asymptotically grow faster than f. If $g \in \Omega(f)$, then g grows asymptotically at least as fast as f. If $g = \Theta(f)$, then the asymptotic growth of f and g is the same. If $g \in \mathrm{o}(f)$, then f grows asymptotically strictly faster than g; if $g \in \omega(f)$, then g grows asymptotically strictly faster than f.

Example 1.3 1. $2n^2 + 1 \in \mathrm{O}(n^2)$, *since* $2n^2 + 1 \leq 3n^2$ *for all* $n \geq 1$.

2. $2n^2 + 1 \in \Omega(n^2)$, *because* $2n^2 + 1 \geq n^2$ *for all* $n \geq 0$. *Thus* $2n^2 + 1 = \Theta(n^2)$.

Remark 1.4 *A little warning: In some books, a different definition of* $\Omega(f)$ *is used, namely,* $g(n) \geq cf(n)$ *has to hold infinitely often (instead of almost everywhere). The idea is that the* O*-notation is used to give upper bounds for the running time. A reasonable upper bound should hold for almost all inputs. The* Ω*-notation is used to state lower bounds. One can consider an algorithm bad if it has a particular running time on infinitely many inputs (and not on almost all).*

Lemma 1.5 1. *If* $f \in \mathrm{O}(g)$, *then* $g \in \Omega(f)$ *and vice versa.*

2. *If* $f \in \mathrm{o}(g)$, *then* $g \in \omega(f)$ *and vice versa.*

Proof. We only prove the first statement, the second is proven in the same way. If $f \in \mathrm{O}(g)$, then there is a constant c and an n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$. But this means that $g(n) \geq \frac{1}{c} f(n)$ for all $n \geq n_0$. Hence $g \in \Omega(f)$. This argument is reversible, which shows the "vice versa"-part (replace c by $\frac{1}{c}$). ■

Exercise 1.1 *Prove:* $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid$ *there are* $c, c' > 0$ *and* $n_0 \in \mathbb{N}$ *such that* $c'f(n) \leq g(n) \leq cf(n)$ *for all* $n \geq n_0\}$.

Syntactic sugar 1.6 *Although* $\mathrm{O}(f)$ *and all the other classes are sets of functions, we will often treat them like a function. "Algorithm A has running time* $\mathrm{O}(n^2)$*" means that the running time of A is a function in* $\mathrm{O}(n^2)$*. Many books also write* $g = \mathrm{O}(f)$ *instead of* $g \in \mathrm{O}(f)$ *and* $\mathrm{O}(g) = \mathrm{O}(f)$ *instead of* $\mathrm{O}(g) \subseteq \mathrm{O}(f)$*. I will try to avoid this.*

1.4 Running time analysis

1.4.1 On foot

To prove that binary search terminates, we showed that $r - \ell$ decreases in every iteration. To analyze the running time, we have to bound the amount by which $r - \ell$ decreases.

But what does it mean "to analyze the running time"?. We can of course count the number of steps for each possible input. But this can get very complicated. Therefore, we will always puts inputs into groups of the same size and measure the running time for them. For binary search, n, the length of the array, is a natural choice. We will consider *worst case* running times. This means that we try to find an upper bound for all running times of the binary search algorithm on inputs of length n. In other words, we want to bound the maximum of all running times on inputs of length n.[3]

Lemma 1.7 *Let λ and ρ be the value of ℓ and r before an iteration of the loop, and λ' and ρ' after the iteration.*[4] *Then*

$$\rho' - \lambda' - 1 \leq \lfloor (\rho - \lambda - 1)/2 \rfloor.$$

Proof. Let $\mu = \lfloor (\rho+\lambda)/2 \rfloor$. In one iteration, either the value of r becomes μ or the value of λ becomes μ. Therefore,

$$\begin{aligned}
\rho' - \lambda' - 1 &\leq \max\{\rho - \mu - 1, \mu - \lambda - 1\} \\
&\leq \max\{\rho - ((\rho + \lambda)/2 - \frac{1}{2}) - 1, (\rho + \lambda)/2 - \lambda - 1\} \\
&= \max\{(\rho - \lambda - 1)/2, (\rho - \lambda)/2 - 1\} \\
&= (\rho - \lambda - 1)/2.
\end{aligned}$$

Since $\rho' + \lambda' - 1$ is a natural number, we can even round down the right hand side. ∎

Exercise 1.2 *Convince yourself that for all $n \in \mathbb{N}$,*

[3] Considering worst case running times has been proven very fruitful. It is usually easier to analyze and it bounds the running time of *every* input. Nevertheless, we will occasionally consider average case running times, that is, we will try to estimate the average of the running times of all inputs of length n.

[4] You should have learned that it is important to distinguish between the variable (part of the syntax) and its value (part of the semantic). ℓ is the variable name, λ and λ' are two values. Since you got older, we will often use the same symbol for the variable and for its value, if there is no or only little danger of confusion, and sometimes even if there is real danger of confusion. For instance, at the beginning of this paragraph, we wrote that $r - \ell$ decreases. Correctly, we had to write the value of r minus the value of ℓ decreases. But since it is sooo convenient, we will often write the name of the variable and mean the value. Take this chance to write all the things you ever wanted to write in the "Programmierung 1+2" lectures and still get credits for it.

1. $\lceil \frac{n}{2} \rceil = \lfloor \frac{n+1}{2} \rfloor$,
2. $\frac{n}{2} - \frac{1}{2} \leq \lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$.

Thus in every step, the quantity $r - \ell - 1$ is halved. In the beginning, we have $r - \ell - 1 = n$. If $n = 0$, then there are no iterations. Assume that $n > 0$. After i iterations, $r - \ell - 1 \leq \lfloor n/2^i \rfloor$. We stop, if $\ell + 1 \geq r$, that is $r - \ell - 1 \leq 0$. This is equivalent to $n/2^i < 1$. ($r - \ell - 1$ is an integer!) The smallest integer i_0 such that $n/2^{i_0} < 1$ is $i_0 = \lfloor \log_2 n \rfloor + 1$. Therefore, Binary-search performs at most $\lfloor \log_2 n \rfloor + 1$ iterations. Since every iteration consists of a constant number of operations, we get the following theorem.

Theorem 1.8 *The running time of Binary-search is* $O(\log_2 n)$.

Exercise 1.3 *Binary-search looks at* $\lfloor \log_2 n \rfloor + 1$ *elements of the array. Prove that this is optimal, i.e., every algorithm that correctly decides whether* x *is in the array or not has to look at* $\geq \lfloor \log_2 n \rfloor + 1$ *elements.*

1.4.2 By solving recurrences

We now invest a little more to get a nicer derivation of the running time. We will essentially only analyze one step of the binary search algorithm and then let our theorems do the rest.

Syntactic sugar 1.9 *For* $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, $f \circ g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ *is the function given by* $n \mapsto f(\lceil g(n) \rceil)$. *We extend this to functions of larger arity in the obvious way.*

Lemma 1.10 *Let* $g_1, \ldots, g_\ell : \mathbb{N} \to \mathbb{R}_{\geq 0}$ *be functions such that* $g_\lambda(m) \leq m-1$ *for all* $m > 0$ *and* $1 \leq \lambda \leq \ell$. *Let* $c \in \mathbb{R}_{\geq 0}$ *and* $h : \mathbb{N} \to \mathbb{R}_{\geq 0}$. *Then the recurrence*

$$f(n) = \begin{cases} c & \text{if } n = 0 \\ f(g_1(n)) + \cdots + f(g_\ell(n)) + h(n) & \text{if } n > 0 \end{cases}$$

has a unique solution $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

Proof. We construct the solution f inductively.
Induction base: $f(0) := c$ fulfills the equation and is the only possible solution.
Induction step: We assume that we already defined all $f(m)$ with $m < n$ and that there was only one possible solution. Since $g_\lambda(n) \leq n - 1$ for all λ, $f(g_\lambda(n))$ is defined and it is unique. Thus there is only one way to define $f(n)$, namely $f(n) := f(g_1(n)) + \cdots + f(g_\ell(n)) + h(n)$. ∎

Remark 1.11 *The same is true (with the same proof), if the values of* $f(0), f(1), \ldots, f(i)$ *are given. In this case* $g_\lambda(m) \leq m - 1$ *has to hold only for* $m > i$.

Remark 1.12 *Again with the same proof, one can show that if $\hat{f}$ satisfies the inequalities $\hat{f}(n) \leq \hat{f}(g_1(n)) + \cdots + \hat{f}(g_\ell(n)) + h(n)$ if $n > 0$ and $\hat{f}(0) \leq c$, then $\hat{f}(n) \leq f(n)$ for all n where f is the unique solution of the equation.*

Throughout this lecture, $\log : \mathbb{N} \to \mathbb{R}_{\geq 0}$ will denote the function

$$n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ \lfloor \log_2(n) \rfloor + 1 & \text{if } n > 0 \end{cases}$$

Here $\log_2 : \mathbb{R} \to \mathbb{R}$ is the "ordinary" logarithm to base 2. $\log n$ essentially is the number of bits of the binary expansion of n. (0 is represented by the empty word.)

Exercise 1.4 *Show that* $\log n = \lceil \log_2(n+1) \rceil$.

Lemma 1.13 *The function* log *is the unique solution of the equation*

$$f(n) = \begin{cases} f(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

Proof. By Lemma 1.10, the equation has a unique solution, since $\lfloor \frac{n}{2} \rfloor < n$ for $n > 0$. The function log is monotonically increasing. It is constant on the sets $\{2^\nu, \ldots, 2^{\nu+1} - 1\}$ for all ν and $\log(2^{\nu+1}) = \log(2^{\nu+1} - 1) + 1$.

We claim that if $n \in \{2^\nu, \ldots, 2^{\nu+1} - 1\}$, then $\lfloor \frac{n}{2} \rfloor \in \{2^{\nu-1}, \ldots, 2^\nu - 1\}$. Since $n \to \lfloor \frac{n}{2} \rfloor$ is a monotonically increasing function, it is sufficient to check this for the borders:

$$\lfloor 2^\nu / 2 \rfloor = \lfloor 2^{\nu-1} \rfloor = 2^{\nu-1},$$
$$\lfloor (2^{\nu+1} - 1)/2 \rfloor = \lfloor 2^\nu - 1/2 \rfloor = 2^\nu - 1.$$

Therefore,

$$\log(\lfloor n/2 \rfloor) + 1 = \log(n)$$

for all $n > 0$. ∎

Corollary 1.14 *Let $c \in \mathbb{R}_{\geq 0}$. The function $c \cdot \log n$ is the unique solution of*

$$f(n) = \begin{cases} f(\lfloor \frac{n}{2} \rfloor) + c & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

Proof. By dividing the equations by c, we see that $\frac{1}{c} \cdot f = \log$. ∎

Now we look at the recursive version of binary search. Let $t : \mathbb{N} \to \mathbb{N}$ denote an upper bound for the number of comparisons[5] made by binary search. t fulfills

$$t(0) = 0$$
$$t(n) \leq t(\lfloor n/2 \rfloor) + 1.$$

The second equation follows from Lemma 1.7. Thus by Lemma 1.13, $t(n) \leq \log(n)$ for all n.

Disclaimer

Algorithms and data structures are a wide field. There are way more interesting and fascinating algorithms and data structures than would ever fit into this course. There are even way more important and basic algorithms and data structures than would fit into this course. Therefore, I recommend that you study related material in text books to get a broader understanding. The books by Cormen, Leiserson, Rivest, and Stein and by Mehlhorn and Sanders are excellent, but there are many more.

[5]For the ease of presentation, we count the "$a[m] = x$" and "$a[m] < x$" comparisons as one comparison. Think of it as a three-valued comparison predicate that returns the results "smaller", "equal", or "larger".

2 Sorting

2.1 Selection-sort

To work properly, binary search requires that the array is sorted. How fast can we sort an array? A simple algorithm works as follows: Look for the minimum element of the array, put it at the front. Then sort the rest of the array in the same way. Look at the minimum element of the remaining array and put it at the front, and so on. In this way, after m iterations, the first m elements are sorted and contain the m smallest elements of the array. Algorithm 3 shows one implementation of this. It is written recursively, but one can easily replace the recursion by one for loop (does not look so nice, but usually yields a faster code).

The correctness of the algorithm is easily shown by induction.

Lemma 2.1 *After executing* Selection_sort($a[1..n]$), $a[1] \leq a[2] \leq \cdots \leq a[n]$ *holds.*

Proof. By induction on n.
Induction base: If $n \leq 1$, then there is nothing to sort and selection sort does nothing.
Induction step: By the induction hypothesis, invoking Selection_sort($a[2..n]$) sorts the subarray $a[2..n]$. Since $a[1]$ is the smallest among the elements in the array, the complete array $a[1..n]$ is sorted, too. ∎

Algorithm 3 Selection-sort

Input: array $a[1..n]$
Output: afterwards, $a[1] \leq a[2] \leq \cdots \leq a[n]$ holds

```
1: if n > 1 then
2:     i := Select-minimum(a[1..n])
3:     h := a[1]; a[1] := a[i]; a[i] := h;
4:     Selection-sort(a[2..n]);
```

The subroutine Select-minimum can be implemented as follows. The index i stores the index of the current minimum. In the for loop, we compare every element with the current minimum and update the index i if a smaller element is found.

The correctness follows from the invariant of the for loop. Again, this invariant can be easily shown by induction. To train a little bit, we will do

the proof formally. Later, we will often just state the invariant and give a hint how to prove it, assuming that you can do the proof on your own.

Lemma 2.2 *The invariant in Algorithm 4 is fulfilled before every iteration of the for loop.*

Proof. The proof is by induction on j.
Induction base: When $j = 2$, then the invariant is trivially fulfilled, since $a[1]$ is the minimum of $a[1..j-1]$.
Induction step: Assume that $a[i]$ is the minimum of $a[1..j-1]$, that is, the invariant was true before the $(j-1)$th execution of the loop. If $j > n$, there is nothing to prove. Otherwise, in the loop, $a[i]$ is compared with $a[j]$ and i is set to j if $a[j] < a[i]$. Thus $a[i]$ is now the minimum of $a[1..j]$. Therefore, the invariant is true before the jth execution, too. ∎

Algorithm 4 Select-minimum

Input: array $a[1..n]$
Output: index i such that $a[i] = \min_{1 \le j \le n} a[j]$

```
1: i = 1
2: for j = 2, ..., n do   /* i ≤ j − 1 ∧ a[i] IS THE MIN OF a[1..j − 1] */
3:    if a[j] < a[i] then
4:       i := j
5: return i;
```

How many comparisons does Selection-sort do? Select-minimum does $n-1$ comparisons when selecting the minimum of an array of length n. Thus we get the following recursive equation for the number $c(n)$ of comparisons of selection sort:

$$c(n) = \begin{cases} 0 & \text{if } n = 0, 1 \\ n - 1 + c(n-1) & \text{if } n > 1 \end{cases}$$

We can determine $c(n)$ by the so-called *iteration method*, an "on foot" method, which accidentally turns out to be elegant here:

$$\begin{aligned} c(n) &= n - 1 + c(n-1) \\ &= n - 1 + n - 2 + c(n-2) \\ &= n - 1 + n - 2 + n - 3 + \cdots + 1 + c(1) \\ &= \sum_{i=1}^{n-1} i \\ &= \frac{1}{2} n(n-1) = \mathrm{O}(n^2). \end{aligned}$$

Thus the overall running time of selection sort is $\mathrm{O}(n^2)$.

Excursus: Gauß's formula for consecutive integer sums

Karl Friedrich Gauß (1777–1855) was a German mathematician, astronomer, and geodesist. If he had lived today, he would also have been a computer scientist for sure.

One hears the following anecdote quite often:[1] At the age of seven, Gauß attended the Volksschule. To have some spare time, the teacher Büttner told the pupils to sum up the numbers from 1 to 100. Gauß quickly got the answer 5050 by recognizing that $1 + 100 = 101$, $2 + 99 = 101$, $\ldots 50 + 51 = 101$, in other words, $\sum_{i=1}^{100} i = \frac{1}{2} \cdot 100 \cdot 101$. Poor Büttner.

2.2 Merge sort

There are other algorithms that sort. Insertion sort is another natural algorithm. It works the way people sort playing cards. We take one element after another and insert it into the correct position. To find the position, we compare it with the elements sorted so far until we find the right position. Then we copy the rest of the array one place to the right and store the current element into the free position. Inserting the ith element requires $O(i)$ steps, since either every element in the list of already sorted elements is either compared to the current element or the element is moved. Therefore, the number of steps is bounded by $O(n^2)$ (by the same analysis as for selection sort).

Can we sort faster? Merge sort is a faster algorithm. We start with the following subproblem. We have two arrays that are already sorted and want to combine them into one sorted array. Algorithm 5 does the job. We assume that our two arrays are stored in one larger array $a[1..n]$. The index t marks the "border" between these two arrays, i.e., $a[1..t]$ is sorted and $a[t+1..n]$ is sorted. We have two indices i and j which mark the smallest element of the two arrays that have not been processed so far. In the beginning, $i = 1$ and $j = t + 1$. Then we compare $a[i]$ with $a[j]$ and add the smaller one to the array b. The condition "$j = n + 1$ or ($i \leq t$ and $a[i] < a[j]$)" also checks whether one of the two arrays $a[1..t]$ and $a[t+1..n]$ is already empty. We assume that Pseudocode does lazy evaluation. In this way, we do not try to access array elements that are out of the boundary.

We can easily show by induction that the invariant of the for loop is correct. From this it follows that $a[1..n]$ is sorted in the end. For every comparison that merge does, one element is added to b. For the last element, we do not need a comparison. Thus merge does at most $n - 1$ comparisons and the overall running time is $O(n)$.

[1] In this lecture notes, you occasionally find anecdotes about mathematicians and computer scientists. I cannot guarantee that they are true but they are entertaining. This particular one can be found on Wikipedia, which indicates that it is false.

Algorithm 5 Merge

Input: array $a[1..n]$, integer t such that $a[1..t]$ and $a[t+1..n]$ are sorted
Output: afterwards, $a[1..n]$ is sorted

```
i := 1; j := t + 1
for k = 1, ..., n do
    /* b[1..k−1] is sorted and all elements in b[1..k−1] are smaller
   than the elements in a[i..t] and a[j..n] */
   if j = n + 1 or (i ≤ t and a[i] < a[j]) then
      b[k] := a[i];
      i := i + 1;
   else
      b[k] := a[j];
      j := j + 1
a[1..n] := b[1..n];
```

Now we can sort as follows (Algorithm 6): If $n \leq 1$, there is nothing to sort. Otherwise, we divide the input array $a[1..n]$ into two halves of (almost) the same size. Then we recursively sort these two arrays. An easy induction in n shows that merge sort indeed sorts.

Algorithm 6 Merge-sort

Input: array $a[1..n]$
Output: afterwards, $a[1] \leq a[2] \leq \cdots \leq a[n]$ holds

```
if n > 1 then
   m := ⌊n/2⌋;
   Merge-sort(a[1..m]);
   Merge-sort(a[m + 1..n]);
   Merge(a[1..n], m);
```

What can we say about the running time? Let us estimate the number of comparisons $c(n)$ that merge sort does on arrays of length n in the worst case. We have

$$c(n) \leq \begin{cases} 0 & \text{if } n \leq 1 \\ c(\lfloor \frac{n}{2} \rfloor) + c(\lceil \frac{n}{2} \rceil) + n - 1 & \text{if } n > 1 \end{cases}$$

since we split the array into two halves and need an additional $n-1$ comparisons for merging. Since c is certainly monotonically increasing, we can replace the second inequality by

$$c(n) \leq 2c(\lceil \tfrac{n}{2} \rceil) + n - 1.$$

Let us divide the inequality by $n-1$ and set $\hat{c}(n) = \frac{c(n)}{n-1}$ for $n \geq 1$. We get

$$\begin{aligned}\hat{c}(n) = \frac{c(n)}{n-1} &\leq 2 \cdot \frac{c(\lceil \frac{n}{2} \rceil)}{n-1} + 1 \\ &\leq 2 \cdot \frac{c(\lceil \frac{n}{2} \rceil)}{2(\lceil \frac{n}{2} \rceil - 1)} + 1 \\ &= \hat{c}(\lceil \frac{n}{2} \rceil) + 1.\end{aligned}$$

We "almost" know $\hat{c}$, since it "almost" fulfills the same recursive inequality as log. In particular, $\hat{c}(n) \leq \log(n-1)$ for $n > 0$, because the function $n \mapsto \log(n-1)$ fulfills the same recurrence:

$$\log(n-1) = \log(\lfloor (n-1)/2 \rfloor) + 1 = \log(\lceil n/2 \rceil - 1) + 1 = \log(\lceil n/2 \rceil - 1) + 1.$$

Therefore, merge sorts makes $c(n) = (n-1)\hat{c}(n) = (n-1)\log(n-1)$ comparisons in the worst case and the overall running is $\mathrm{O}(n \log n)$.

Divide and Conquer

Divide and conquer is a general design principle. It consists of three steps:

Divide: Split the problem into smaller subproblems.
In the case of merge sort, these are the two sublists.

Conquer: Solve the subproblems recursively.
This is done by the two recursive calls.

Combine: Combine the solutions of the subproblems to get a solution of the original problem.
This is done by the procedure merge.

Of course, the structure of the problem has to be suitable to allow a divide and conquer approach.

Excursus: Sorting in practice

In practice, one does not just sort numbers. Sorting algorithms are for instance used to sort database records according to some *key*. The rest of the record is so-called satellite data. Instead of storing the whole records/objects in the array, one usually only store a reference or pointer to the records/objects and just moves the references in the array.

When sorting really large array, memory management can be more important than minimizing the number of comparisons, since fetching blocks from and writing them to the hard disk is a really costly task. There are algorithms for this, too, but we will not talk about them in this lecture.

3 More sorting procedures

A sorting algorithms is called *in place* if it only stores a constant number of values outside of the array. Merge sort is not an in place sorting algorithm, since the merge procedure uses a second array. So merge sort needs twice as much storage to run as is needed to store the data. For large data sets, this can be problematic.

3.1 Heap sort

Heap sort is a sorting algorithm that is in place and whose running time is also $O(n \log n)$ in the worst case. We introduce another important principle of algorithm design, namely, *data structures*. Data structures are used to manage the data during the execution of the algorithm. The data is structured in such a way that operations that are crucial for the algorithms can be performed in an efficient manner. You can think of it as a class. We have several methods which are described by their input/output behavior and we have upper bounds for their running times.

3.1.1 Heaps

A *(binary) heap* A can be viewed as an ordered binary tree all levels of which are completely filled except for the bottom one. Each node of the tree stores one element of our array. We have three basic functions

Parent(i) returns the parent node of i, if i has a parent.

Left(i) returns the left child of i, if i has a left child

Right(i) return the right child of i, if i has a right child

So far, there is nothing special about heaps. The crucial thing is that they satisfy the *heap property*:

Heap property

$A[\text{Parent}(i)] \geq A[i]$ for all i except the root

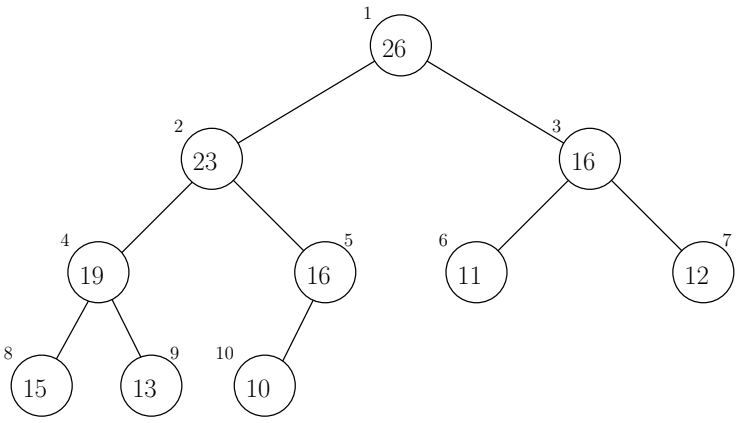

Figure 3.1: A heap. The large numbers in the circles are the elements stored in the heap. The small numbers next to the circle is the corresponding position in the array.

That is, the value of every node is larger than the values of its two children. Using induction, we get that for every node i in the heap, all nodes that are below $A[i]$ have a value that is at most $A[i]$. **Warning!** Do not confuse our heaps with the heap in a JAVA environment.

Heaps can be efficiently implemented: Assume we want to store n elements in an array A. We just store them into in an array $A[1..n]$. We have a variable *heap-size* which stores the number of elements in the heap.[1] The nodes are just the indices $\{1, \ldots, \textit{heap-size}\}$. The array may contain more elements (with larger index), since later on, we will build a heap step by step starting with a heap of size 1.

The value of the root is stored in $A[1]$. It has two children, we store their value in $A[2..3]$. These two children have altogether four children. We can store their values in $A[4..7]$. In general, the ith "layer" of the tree is stored in $A[2^i..2^{i+1}-1]$. (The root is the 0th layer.) The last layer might be shorter and is stored in $A[2^h..\textit{heap-size}]$. Here $h = \log(\textit{heapsize}) - 1$ is the height of the tree, the number of edges on a longest path that goes from the root to a leaf. (Because of the structure of heaps, every path from the root to a leaf has length either $h-1$ or h.)

The methods Parent, Left, and Right can be easily implemented as follows:

It is easy to verify that the functions Left and Right map the set $\{2^i, \ldots, 2^{i+1}-1\}$ (level i) into the set $\{2^{i+1}, \ldots, 2^{i+2}-1\}$ (level $i+1$). Left is nothing

[1] Yes, I know, never ever have a global variable floating around somewhere. But for Pseudocode, this is okay. We just want to *understand* how heaps work. I am pretty sure that with your knowledge from the "Programmierung 1+2" lectures, you can do a very nice state-of-the art implementation.

Algorithm 7 Parent(i)

1: return $\lfloor i/2 \rfloor$

Algorithm 8 Left(i)

1: return $2i$

else but a left shift and adding the least significant bit 0, Right does the same but adds a 1. In particular these implementations are very efficient and short, so in reality, you could declare them as inline. Since we use the Left, Right, and Parent procedures only inside other procedure, we do not check whether the result is still a valid node, i.e, whether it is in $\{1, \ldots, \textit{heap-size}\}$; this has to be done by the calling procedures.

3.1.2 Establishing the Heap property

The function Heapify is a crucial procedure to ensure the heap property. It gets a node i as input. We promise that the two subtrees that have the children Left(i) and Right(i) as roots satisfies the heap property, but the subtree with root i may not. After execution, also the tree with root i satisfies the heap property.

After the execution of lines 1–7, $h \in \{i, \ell, r\}$ is chosen such that

$$A[h] = \max\{A[i], A[\ell], A[r]\}.$$

Note that we check in lines 4 and 6 whether the children are indeed present. If $h = i$, then the heap property is also satisfied by the subtree with root i and we do not have to do anything. Otherwise, we exchange $A[i]$ with $A[h]$. Now $A[i]$ contains the largest element of the subtree with root i. The subtree whose root is not h is not affected at all and still satisfies the heap property. The other subtree with root h almost fulfills the heap property: The two subtrees Left(h) and Right(h) fulfill the heap property but $A[h]$ might not be the largest element. But this is precisely the input situation of Heapify, so we can apply recursion.

What's the running time of Heapify? Since every call results in at most one recursive call, the number of recursive calls is bounded by the height of the subtree with root i. In each call, only a constant number of operations are executed. Thus, the total running time is $O(\log n)$.

Exercise 3.1 *Analyze the running time of* Heapify *by formulating a recurrence and solving it.*

Algorithm 9 Right(i)

1: return $2i + 1$

Algorithm 10 Heapify

Input: heap A, index i such that the heap property is fulfilled for the subheaps with roots Left(i) and Right(i)
Output: afterwards, the heap property is also fulfilled at i

```
1: ℓ := Left[i]
2: r := Right[i]
3: h := i
4: if ℓ ≤ heap-size and A[ℓ] > A[i] then
5:    h = ℓ
6: if r ≤ heap-size and A[r] > A[h] then
7:    h = r
8: if h ≠ i then
9:    swap(A[i], A[h])
10:   Heapify(A, h)
```

3.1.3 Building a heap from scratch

Now assume we have an array $A[1..n]$ and we want to convert it into a heap. We can use the procedure Heapify in a bottom-up manner. Because the indices $\{\lfloor n/2+1 \rfloor, \ldots, n\}$ are all leaves, the subtree with a root $j > \lfloor n/2 \rfloor$ is a heap. Then we apply Heapify and ensure the heap property in a layer by layer fashion. The correctness of the approach can be easily proven by reverse induction on i.

Algorithm 11 Build-heap

Input: array $A[1..n]$
Output: afterwards, A satisfies the heap property

```
1: heap-size := n
2: for i = ⌊n/2⌋, ..., 1 do
3:    Heapify(A, i)
```

What is the running time of Build-heap? An easy upper bound can be obtained as follows: The running time of each Heapify call is $O(\log n)$ and there are $\lfloor n/2 \rfloor$ such calls, therefore, the total running is bounded by $O(n \log n)$. This bound is correct, since it is an upper bound, but we can do better.

The crucial observation is for a lot of subtrees, the height is not $\log n$ but much smaller. $\lceil n/2 \rceil$ nodes are leaves, the corresponding subtrees have height 0. The leaves have $\lceil n/4 \rceil$ parents, the height of the corresponding trees is 1, etc.

Exercise 3.2 *Show that in general, a heap with n nodes has at most $\lceil n/2^{h+1} \rceil$ subtrees of height h.*

Therefore, we can bound the running of build-heap by

$$\sum_{h=0}^{\log n-1} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \mathrm{O}(h) \subseteq \mathrm{O}\left(n \cdot \sum_{h=0}^{\log n-1} \frac{h}{2^h}\right) \subseteq \mathrm{O}\left(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h}\right).$$

We know that

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}.$$

for all x with $|x| < 1$ (geometric series). Taking derivatives with respect to x, and multiplying with x, we get

$$\sum_{h=0}^{\infty} h \cdot x^h = \frac{x}{(1-x)^2}.$$

Therefore, $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$ and

$$\mathrm{O}\left(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \subseteq \mathrm{O}(n)$$

3.1.4 Heap sort

To sort an array $A[1..n]$ using heaps, we first build a heap. Now we can easily get the largest element of the array. It is $A[1]$. We exchange it with the last element of the array. $A[1]$ is now in the right place. The array $A[1..n-1]$ is almost a heap, only the element $A[1]$ may violate the heap property. But this is exactly the situation for Heapify! After the call in line 5, $A[1..n-1]$ is a heap and we can repeat the process.

Algorithm 12 Heap sort

Input: array $A[1..n]$
Output: afterwards, A is sorted

```
1: Build-heap(A)
2: for i := n, ..., 2 do
3:     Swap(A[1], A[i])
4:     heap-size := heap-size − 1
5:     Heapify(A, 1)   /* A[i..n] IS SORTED AND CONTAINS THE n − i + 1
       LARGEST ELEMENTS */
```

The correctness is again easily shown by induction. The analysis of the running time is not hard either. The for loop is executed $n-1$ times, the Heapify call needs time $\mathrm{O}(\log n)$. Therefore, the total running times is $\mathrm{O}(n \log n)$. Heap sort is a clever implementation of the selection sort idea.

Exercise 3.3 *In the analysis of Build-heap, we got a better time bound when estimating the running times of the Heapify calls more carefully. Does the same work for Heap sort?*

3.2 Quick sort

Quick sort is an in-place sorting algorithm that is widely used in practice. Its worst case running time is quadratic, but it performs very well in practice. One can also show that the average running time is $O(n \log n)$. Since we need some probability theory to prove this, we defer this proof until you learn these things in the Mathematics for Computer Scientists 3 lecture and just state the algorithm.

Quick sort gets an array A and bounds ℓ and r. After execution, the subarray $A[\ell..r]$ is sorted. Quick sort works in a divide and conquer manner. We divide the array into two parts and sort them recursively.

Algorithm 13 Quick sort

Input: array $A[1..n]$, indices ℓ, r
Output: afterwards, $A[\ell..r]$ is sorted

```
1: if ℓ < r then
2:     q := Partition(A, ℓ, r)
3:     Quicksort(A, ℓ, q − 1)
4:     Quicksort(A, q + 1, r)
```

Compared to merge sort, instead of just cutting the array into two halves, sort recursively, and then merge, quick sort divides the array into two parts in a more clever way such that no merging is needed after the recursive calls.

The function Partition does the following. It chooses a *pivot* element p , that is, any element of $A[\ell..r]$. We simply choose $p := A[\ell]$.[2] Then we rearrange A in such a way that $A[1..q-1]$ only contains elements $\leq p$, $A[q]$ contains p, and $A[q+1..r]$ contains only elements $\geq p$. Finally, we return q. We now simply need to sort $A[1..q-1]$ and $A[q+1..r]$ recursively and are done.

The function Partition can easily implement in time $O(n)$. With a little care, we can even do it in-place.

In the first line, the *pivot* element is chosen. We take the first element. Then we move the pivot temporarily to the end. Then we run through the array and compare whether $A[i] \leq p$. If this is the case, then $A[i]$ has to be moved to the left. So we swap it with $A[s]$. The index s stores the current border between the "lower" part and the "upper" part. Finally, we just put the pivot again in the middle.

[2] Choosing a pivot element in practice is somewhere between art and magic, since it shall be chosen in such a way that the two parts are roughly balanced. Taking the first element is often a bad choice in practice, in particular, if the array is partly sorted.

Algorithm 14 Partition

Input: array $A[1..n]$, indices $\ell \leq r$

Output: afterwards, there is an index q such that all elements in $A[\ell..q-1]$ are $\leq A[q]$ and the ones in $A[q+1..r]$ are $\geq A[q]$. q is returned.

```
1: p := A[ℓ]
2: swap(A[ℓ], A[r])
3: s := ℓ
4: for i := ℓ, ..., r − 1 do
5:    if A[i] ≤ p then
6:       swap(A[i], A[s])
7:       s := s + 1
8: swap(A[s], A[r])
9: return s;
```

4 Selection problems

Our procedure for finding the minimum or maximum needs $n-1$ comparisons. We want to show that this is optimal.

4.1 Lower bounds

Let X be a finite set. Recall that a (binary) relation is a subset $R \subseteq X \times X$. R is *irreflexive*, if for all $x \in X$, $(x,x) \notin R$. R is *antisymmetric* if for all $x \neq y$, $(x,y) \in R$ implies $(y,x) \notin R$. R is *transitive*, if for all $x,y,z \in X$, $(x,y) \in R$ and $(y,z) \in R$ implies $(x,z) \in R$. A *partial order* on X is a relation $R \subseteq X \times X$ that is irreflexive, antisymmetric, and transitive. R is a *total order* if in addition, $(x,y) \in R$ or $(y,x) \in R$ for all $x \neq y$.

Exercise 4.1 *1. Show that the usual relation $<$ on $\mathbb{N}$ is a total order.*

2. What about $\leq$?[1]

3. Show that "a is a nontrivial divisor of b" is a partial order. (A divisor is called nontrivial if $a \neq 1$ and $a \neq b$.)

Instead of writing $(x,y) \in R$, we will often write xRy. With an R, this does not look nice, on the other hand $x < y$ looks nicer than $(x,y) \in <$.

An element x is *minimal* with respect to R if for all $y \neq x$, $(y,x) \notin R$. x is *maximal* with respect to R, if for all $y \neq x$, $(x,y) \notin R$.

We can represent relations as directed graphs, more precisely, (X,R) is a directed graph. Since R is irreflexive, G has no self-loops. Antisymmetry means that there is no cycle of length two, that is, two edges (x,y) and (y,x). Transitivity means that whenever there is a directed path from x to y, there is also an edge (x,y). (From the definition of transitivity, this follows directly for paths of length 2. The rest follows by induction on the path length.) Altogether, this means that G is acyclic, i.e., it does not contain any directed cycles. Thus, every order induces an acyclic graph. On the other hand, every acyclic graph induces a partial order by taking the transitive closure.

A *comparison tree* is an ordered binary tree $T = (V,E)$. Each node that is not a leaf is labeled with a comparison $x?y$, where x,y are two variables from a set $X = \{x_1, \ldots, x_n\}$ of variables. For reasons that will become clear in one or two sentences, the left child of a node v is denoted by $v_<$ and the

[1] Some authors prefer orders to be reflexive, that is, $(x,x) \in R$ for all $x \in X$, instead of irreflexive.

right child is denoted by $v_>$. With every node v of the graph, we associate a relation $R(v)$, which is defined inductively. If v is the root of the tree, then $R(v) = \emptyset$, that is, we do not know anything about the relation between the elements in X. Assume that we have defined $R(v)$ for some node v and that v is labeled with the comparison $x?y$. Then $R(v_<)$ is the transitive closure of $R(v) \cup \{(x, y)\}$. In the same way, $R(v_>)$ is the transitive closure of $R(v) \cup \{(y, x)\}$. So $R(v)$ stores all the information about the relation between the variables in X that the comparison tree has gathered so far (including what it can deduce by transitivity). We always assume that the comparison tree does nothing stupid, that is, it will never compare an x with itself nor does it compare an x with a y with $(x, y) \in R(v)$ or $(y, x) \in R(v)$.

Let π be a permutation in S_n. For a relation R on x, the relation R_π is defined by

$$(x_i, x_j) \in R_\pi \quad :\Longleftrightarrow \quad (x_{\pi(i)}, x_{\pi(j)}) \in R$$

for all i, j, that is, the relation is the same up to changing the roles of the $x_1, \ldots, x_n$.

What does it mean that a comparison tree computes a particular relation R on X? We have our variables $X = \{x_1, \ldots, x_n\}$. There is a hidden total order, that is, there is a permutation $\pi \in X$ such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$ but the comparison tree does not know this permutation. Now we "run" the comparison tree in the obvious way. At every node v, two elements $x_i, x_j \in X$ are compared. If $x_i < x_j$ according to the order given by π, we go to $v_<$, otherwise we go to $v_>$. Eventually, we get to a leaf ℓ. We require that $R \subseteq R(\ell)_\pi$, that is, the comparisons tree finds the relation R between the elements up the the renaming given by π.

In the case of sorting, the relation computed at every leaf is simply a total order on X. That is, the relation computed by the tree is $S = \{(x_i, x_j) \mid i < j\}$.

In the case of minimum computation, the comparison tree shall be able to tell at every leaf which one of the x_j is the minimum. That is, the relation computed by the tree is $M = \{(x_1, x_i) \mid 2 \leq i \leq n\}$. This means that x_1 is minimal and no other element is.

Every algorithm that we studied so far was comparison-based, that is, we only accessed the elements of the array by comparisons. Therefore, every algorithm yields a comparison tree by just running the algorithm and whenever two elements are compared, we add a node to the comparison tree and branch on the two possible outcomes. (If the algorithms does something stupid and compares two elements where the outcome is already known, then we do not add a node and there is no need to branch.) The number of comparisons that our algorithm does in the worst case is nothing else than the height of the constructed comparisons tree.

Exercise 4.2 *Formally prove the following: Let A be a sorting algorithm*

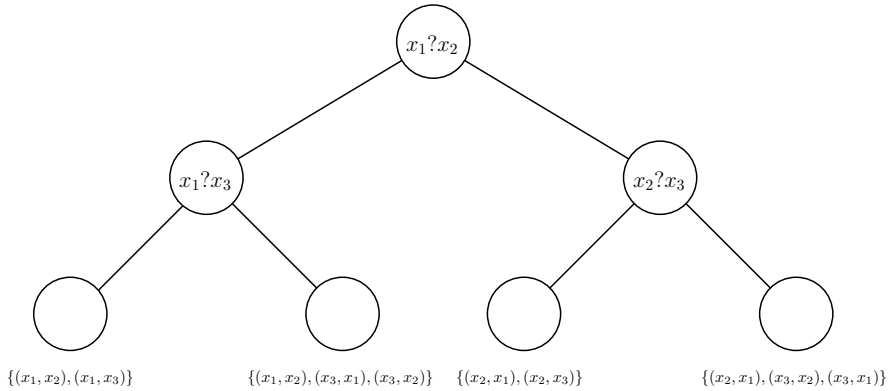

Figure 4.1: A comparison tree which finds the minimum of three elements. The left child of a node v is the child $v_<$, right child is the node $v_>$. (Node names are not drawn.) Each node that is not a leaf is labeled with a comparison. Below each leaf, there is the relation that is computed at the leaf. At two leaves, the tree learns the complete order between the three elements, at the other two, it only knows that one is smaller than the other two, but it does not know the relation between the other two. But this is fine, since the tree shall only find the minimum.

which makes at most $t(n)$ comparisons. Then there is a comparison tree with height bounded by $t(n)$ which sorts.

Comparison trees itself are not a good machine model for sorting, since they are *non-uniform*: For most comparison trees, there is no compact way to represent them! But we will use comparison trees only to show lower bounds. And since every sorting algorithm in Pseudocode gives a comparison tree, this also yields lower bounds for the number of comparisons made by algorithms in Pseudocode.

4.1.1 Sorting

Theorem 4.1 *Every comparison that sorts the elements in $X = \{x_1, \ldots, x_n\}$ has at least $n!$ leaves.*

Proof. If a comparison tree sorts, then two different permutations π and π' obviously have to lead to different leaves ℓ and ℓ', because $S(\ell)_\pi$ and $S(\ell')_{\pi'}$ have to be the same relation S, which is a total ordering. There are $n!$ permutations of $\{1, \ldots, n\}$. ■

Since a tree with N leaves has at height at least $\log N - 1$, every Pseudocode algorithm that sorts by using comparisons needs $\log(n!) - 1$ compar-

isons. A rough estimate for $n!$ is

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

(why?). Because $\log_2 n \leq \log n \leq \log_2 n + 1$, we get the lower bound

$$\log(n!) \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log_2 n - \frac{n}{2} \in \Omega(n \log n).$$

A more precise estimate for $n!$ is *Stirling's formula*, which says that

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\lambda_n}$$

for some $\lambda_n \in (\frac{1}{12n+1}, \frac{1}{12n})$. With this, we get

$$\log(n!) \geq n\log_2 n - n\log_2 e + \frac{1}{2}\log_2(2\pi n) + \frac{\log_2 e}{12n+1} \geq n\log n - n(\log_2 e + 1).$$

Merge sort uses $(n-1)\log(n-1) \leq n \log n$ comparisons. So up to a linear lower order term, merge sort uses the minimum number of comparisons.

Let us first make the proof a little more complicated. In this way, the method becomes applicable to a wider range of problems. Consider a comparison tree T with node set V. A function $w : V \to \mathbb{R}$ is a *potential function* if $w(v) \leq w(v_<) + w(v_>)$ for all nodes v that are not a leaf.

Lemma 4.2 *Let T be a comparison tree and w be a potential function. If $w(v) \leq B$ for all leaves v, then the height of T is $\geq \log_2(\frac{w(r)}{B})$ where r is the root of T.*

Proof. We prove the following more general statement: For every node v, the height of the subtree with root v is $\geq \log_2(\frac{w(v)}{B})$. The proof is by induction on the height of the subtree.
Induction base: If v is a leaf, then $w(v) \leq B$, hence $w(v)/B \leq 1$ and $\log_2(\frac{w(r)}{B}) \leq 0$.
Induction step: Let v be any node that is not a leaf. Since $w(v) \leq w(v_<) + w(v_>)$, we have $\frac{w(v)}{2} \leq \max\{w(v_<), w(v_>)\}$. W.l.o.g. let $w(v_<) \geq w(v_>)$. By the induction hypothesis, we know that the height of the subtree with root $v_<$ is at least $\log(\frac{w(v_<)}{B})$. The height of the subtree with root v therefore is at least

$$\log_2\left(\frac{w(v_<)}{B}\right) + 1 = \log_2\left(2 \cdot \frac{w(v_<)}{B}\right) \geq \log_2\left(\frac{w(v)}{B}\right). \qquad \blacksquare$$

We call a permutation π *consistent* with an order R on X, if, whenever $(x_i, x_j) \in R$, then $x_i < x_j$ in the hidden total order given by the permutation π. That is, R does not contradict the total order induced by π.

Consider the following function $s : V \to \mathbb{R}$ defined by

$$s(v) = \text{the number of permutations that are consistent with } R(v).$$

We claim that s is a potential function: If π is consistent with $R(v)$, then π is either consistent with $R(v_<)$ or $R(v_>)$. Thus $s(v) = s(v_<) + s(v_>)$. We have $s(r) = n!$ for the root r and $s(\ell) = 1$ for every leaf ℓ. Therefore, the height of the tree is a least $\log_2(n!) - 1$.

4.1.2 Minimum selection

We can use potential functions to show that selecting the minimum requires at least $n - 1$ comparisons. While this is intuitively clear, since one "has to look at every element", we could for instance just compare pairs of elements ($n/2$ comparisons) and have looked at every element and still do not know the minimum. Consider the following function

$$m(v) = 2^{\text{number of minima in } R(v)}.$$

Assume that at a node v, $R(v)$ has i minima. Then both $R(v_<)$ and $R(v_>)$ have at least $i - 1$ minima, since a comparison can kill at most one minimum. Because $2^i = 2^{i-1} + 2^{i-1}$, m is a potential function. At the root r, we have $m(r) = n$, since every element is a minimum. At a leaf v, $m(v) = 1$, since our tree finds the minimum. Therefore, the height is $\geq \log_2(\frac{2^n}{2}) = n - 1$.

Theorem 4.3 *Every comparison tree that finds the minimum has height* $\geq n - 1$.

4.2 Upper bounds for selecting the median

Selecting the minimum can be done with $n - 1$ comparisons. What about selecting the *median*, the element in middle. Given n elements and a total order on it, the median is the unique element x such that $\lfloor (n-1)/2 \rfloor$ elements are smaller than x and $\lceil (n-1)/2 \rceil$ elements are larger than x. Or what about selecting the ith largest elements for any $i \in \{1, \ldots, n\}$? We can of course sort the elements using any sorting algorithm and then pick the ith largest element. But we can do it with $O(n)$ comparisons for any i.

The algorithm above is written in Pseudopseudocode. Pseudopseudocode uses natural language to describe the algorithm. Pseudopseudocode is a very advanced language, since the user has to be very careful. Whenever you are in doubt whether some piece of Pseudopseudocode is o.k., ask yourself whether you can write a program in Pseudocode that does the same. When analyzing the running time of a program in Pseudopseudocode, you have to analyze the running time of the corresponding program in Pseudocode.

Algorithm 15 Select

Input: array $a[1..n]$, integer i
Output: the ith largest element in a

1: If $n \leq 60$, then find the ith largest element by sorting.
2: Divide the n elements in $\lfloor n/5 \rfloor$ groups of 5 elements. At most 4 elements remain.
3: Find the median of each of the $\lfloor n/5 \rfloor$ groups by sorting.
4: Recursively call Select to find the median m of the $\lfloor n/5 \rfloor$ medians. (Once we found m, we forget about the groups.)
5: Use the procedure Partition on a with m as the pivot element.
6: Let q be the position of m in the array.
7: If $q = i$, then return m.
8: If $i < q$, then call Select$(a[1..q-1], i)$. Else call Select$(a[q+1..n], i-q)$.

The algorithm Select first groups the elements into $\lfloor n/5 \rfloor$ groups of five elements. From each of the groups it computes the median for instance just by sorting. (With Mergesort this takes 12 comparisons; there is also an (optimal) comparison tree for selecting the median with height 6). We use Select recursively to find the median m of the $\lfloor n/5 \rfloor$ medians and use m as a pivot. m is a good pivot, because it yields a rather balanced partition.

Lemma 4.4 *$q - 1 \geq \frac{3}{10}n - 3$ and $n - q \geq \frac{3}{10}n - 3$.*

Proof. Let $r = \lfloor n/5 \rfloor$ be the number of groups with 5 elements.

Since m is the median of the medians, $\lceil \frac{1}{2}(r-1) \rceil$ medians are larger and $\lfloor \frac{1}{2}(r-1) \rfloor$ medians are smaller than m. For every median, there are two elements that are larger than the median and two that are smaller (in his group). And there are two elements that are larger than m and two elements that are smaller. Altogether, there are at least $3 \cdot \lceil \frac{1}{2}(r-1) \rceil + 2$ elements that are larger than m, namely the larger medians, the two elements in each group that are larger than the median, and the two larger elements in the group of m, and $3 \cdot \lfloor \frac{1}{2}(r-1) \rfloor + 2$ elements that are smaller. Since the later quantity is smaller, we only have to bound this one:

$$\begin{aligned} 3 \cdot \left\lfloor \frac{1}{2}(r-1) \right\rfloor + 2 &\geq 3 \cdot \left(\frac{1}{2}(r-1) - \frac{1}{2} \right) + 2 \\ &\geq 3 \cdot \left(\frac{1}{2} \left(\frac{n}{5} - \frac{4}{5} - 1 \right) - \frac{1}{2} \right) + 2 \\ &\geq \frac{3}{10}n - 3. \qquad \blacksquare \end{aligned}$$

By selecting the median of medians, we found a good pivot for the Partition procedure. Since the size of each partition is at least $\frac{3}{10}n - 3$, it

can be, on the other hand, at most $\frac{7}{10}n + 2$ (we can subtract one more for the median of medians).

Let $t(n)$ be the number of comparisons made by select on arrays of length n. We have

$$t(n) = t\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + t\left(\left\lceil \frac{7}{10}n \right\rceil + 2\right) + 12\left\lfloor \frac{n}{5} \right\rfloor + n, \tag{4.1}$$

if $n > 60$, since we have one recursive call with an array of length $\lfloor \frac{n}{5} \rfloor$ for finding the median of medians and another recursive call on the partition that contains the true median. We need $12 \cdot \lfloor \frac{n}{5} \rfloor$ comparisons to find the medians in the groups and $n - 1$ comparisons within partition. If $n \leq 60$, we have $t(n) \leq (n-1)\log(n-1) \leq 8n$.

Lemma 4.5 *Let $\epsilon_1, \ldots, \epsilon_\ell \in \mathbb{Q}$ be positive. Let $d, e \in \mathbb{Q}$ be positive. Let $N \geq \frac{1}{1-\epsilon_\lambda}$ for all $1 \leq \lambda \leq \ell$. Assume that $\epsilon_1 + \cdots + \epsilon_\ell + \frac{\ell}{N} < 1$. If*

$$t(n) \leq \begin{cases} t(\lceil \epsilon_1 n \rceil) + \cdots + t(\lceil \epsilon_\ell n \rceil) + d \cdot n & \text{if } n > N \\ e \cdot n & \text{otherwise} \end{cases}$$

then $t(n) \leq c \cdot n$ where $c = \max\{\frac{d}{1-(\epsilon_1+\cdots+\epsilon_\ell+\frac{\ell}{N})}, e\}$.

Proof. Induction base: If $n \leq N$, then $t(n) = e \cdot n \leq c \cdot n$.
Induction step: Let $n > N$. By the induction hypothesis, we can assume that $t(m) \leq cm$ for all $m < n$. By the choice of N, $\lceil \epsilon_\lambda n \rceil < n$. Thus we can apply the induction hypothesis to all $t(\lceil \epsilon_\lambda N \rceil)$. We have

$$\begin{aligned} t(n) &\leq t(\lceil \epsilon_1 n \rceil) + \cdots + t(\lceil \epsilon_\ell n \rceil) + d \cdot n \\ &\leq c \cdot \lceil \epsilon_1 n \rceil + \cdots + c \cdot \lceil \epsilon_\ell n \rceil + d \cdot n \\ &\leq c \cdot (\epsilon_1 + \cdots + \epsilon_\ell)n + c \cdot \ell + d \cdot n \\ &\leq c \cdot n \cdot \underbrace{\left(\epsilon_1 + \cdots + \epsilon_\ell + \frac{\ell}{N} + \frac{d}{c}\right)}_{\leq 1}. \end{aligned}$$

■

We can use Lemma 4.5 to solve (4.1). We can bound $\frac{7}{10}n + 2$ from above by $\frac{11}{15}n$ for $n > 60$. Since $\frac{1}{5} + \frac{11}{15} + \frac{2}{60} = \frac{29}{30} < 1$, we get that $t(n) \leq c \cdot n$ with $c = 102$.

Remark 4.6 *The actual value for c is rather coarse. We can get the term $\frac{\ell}{N}$ as small as we want by making N larger. In the same way, we can bound $\frac{7}{10}n + 2$ by $h \cdot n$ for an h that is arbitrarily close to $\frac{7}{10}$.*

5 Elementary data structures

Data structures store data in a clever way to support certain operations. We already saw heaps that stored the data in such a way that we could quickly find the minimum (or maximum).

In this chapter we will review some of the basic data structures. I am sure that you have seen them in the "Programmierung 1+2" lectures.

5.1 Stacks

Stacks provide two operations. We can insert an element and we can remove it. These two operations are usually called Push and Pop. Stacks use the last-in-first-out principle. They work like stacks in real life: When you want to put something on the stack, you can only put it on the top. And when you want to remove something from the stack, you can only take the element on the top.

To implement a stack, all you need is an array $S[1..N]$ and a variable *top*. This variable stores the index of the last element put on the stack. In the beginning, we set *top* to 0.

The algorithm IsEmpty checks, whether there are elements on the stack.

Algorithm 16 IsEmpty

Input: stack S
Output: 1 if S is empty, 0 otherwise

```
1: if top = 0 then
2:     return 1
3: else
4:     return 0
```

Push pushes an element x on the stack. It first increases *top* by one and then stores x in the new position. It also checks whether the stack is full. (If for some reason, we know that there will be no stack overflow, one can also dismiss this check for efficiency reasons.)

Pop removes an element from the stack and returns it. It also checks whether there is an element to pop.

All the operations supported by the stack take O(1) time. Often, a Peek procedure is provided in addition. Peek returns the top element without removing it. Peek can be simulated by popping the top element and pushing it back.

Algorithm 17 Push

Input: stack S, element x
Output: adds x to S

```
1: if top ≥ N then
2:     "error"
3: else
4:     top = top + 1;
5:     S[top] = x;
```

Algorithm 18 Pop

Input: stack S
Output: returns and removes the top element of S

```
1: if IsEmpty(S) then
2:     "error"
3: else
4:     top := top − 1
5:     return S[top+1]
```

5.2 Queues

Queues do the same as stacks: We can store elements and retrieve them. However, the storing principle is first-in-first-out. Queues work like queues in real life, say, in the mensa. If a student arrives, he or she goes to the end of the queue. Whenever a meal is handed out, the first student in the queue leaves happily (?) with his or her meal.

To implement a queue, we need an array $Q[1..N]$. We have two variables, *head* and *tail*. The variable *head* stores the index of the elements that is the first in the queue, *tail* the index of position after the queue where the next arriving element will be put. In the beginning, *head* and *tail* will be set to 1.

The queue will be arranged in a circular way in Q. If $head \leq tail$, then the queue consists of the elements $Q[head..tail-1]$. If $tail < head$, then the queue are the elements $Q[head..N]$ together with the elements $Q[1..tail-1]$. In this way, we do not have to shift the queues by one whenever an element is dequeued. (Unlike queues in real life, which usually move forward when the first element is removed. One exception might be very bad meals.)

The queue is empty, when $head = tail$. The queue is full, when $head = tail + 1$. In the latter case, there is still one free cell, but if we occupied this one, then $head = tail$, which we could not distinguish from the empty queue. (This is an information theoretic problem: there are $n+1$ possible numbers of elements that can be stored in the queue, namely $0, 1, \ldots, n$. On the other hand, the difference between *head* and *tail* can only attain n different values.) If we want to use this last cell, then we should introduce a variable *count* which counts the length of the queue. However, the first solution usually

gives a slightly more efficient code.

Procedure IsEmpty checks whether the queue is empty.

Algorithm 19 IsEmpty

Input: queue Q
Output: 1 if Q is empty, 0 otherwise

```
1: if head = tail then
2:    return 1
3: else
4:    return 0
```

Procedure Enqueue appends x to the queue. First, it checks whether the queue if full. The second part of the disjunction checks the case that the queue occupies $Q[1..N-1]$. If the queue is not full, then we store x in $Q[tail]$ and increase *tail* by 1. If *tail* will get larger than N, then we set it to 1.

Algorithm 20 Enqueue

Input: queue Q, element x
Output: adds x to Q

```
1: if head = tail + 1 or head = 1 and tail = N then
2:    "error"
3: else
4:    Q[tail] = x
5:    if tail = N then
6:       tail = 1
7:    else
8:       tail = tail + 1
```

Procedure Dequeue returns the first element and works in a similar manner like Enqueue. All three procedures have running time O(1).

Algorithm 21 Dequeue

Input: queue Q
Output: the first element of Q

```
1: if head = tail then
2:    "error"
3: else
4:    x := Q[head]
5:    if head = N then
6:       head := 1
7:    else
8:       head := head + 1
9:    return x
```

5.3 Linked lists

A linked list stores elements in a linear order. Unlike an array, we do not have direct access to the elements via the indices. The elements are linked via references. This yields a very flexible structure though not every operation is very efficient. There are singly linked list and doubly linked list. In a singly linked list, for every element in the list, we only store a reference to the next object in the list. In a doubly linked list, we also store a reference to the previous object in the list.

For every element x in a doubly linked list, there are three basic methods:

Prev(x)**:** is a reference to the previous element in the list (not present in a singly linked list).

Next(x)**:** is a reference to the next element

Key(x)**:** is the key of the element

Usually, there is more data to store than the key. So we also store a reference to the element data. But this is not important for our considerations. For the first element h in the list, the head, Prev(h) = NULL is a null reference. For the last element in the list, the tail, Next(x) = NULL. There is a reference *head* which points to the first element of the list.

The procedure List-search finds an element in the list with a given key, if there is such an element, otherwise it returns the NULL reference. In the worst case, it has to scan the whole list, therefore, the running time is O(n). Note that the while loop either stops when we found the key k or we are sure that there is no node with key k.

Algorithm 22 List-search

Input: a list L, a key k

Output: (a reference to) an element x with Key(x) = k, if there exists one, NULL otherwise

```
1: x := head
2: while x ≠ NULL and Key[x] ≠ k do
3:     x := Next(x)
4: return x.
```

The procedure List-insert adds an element to the front of the list. It takes O(1) time. If we want to insert the element at a specific place, say after an element y (maybe we want to keep the list sorted), the same algorithm works, we just have to replace the reference *head* by Next(y).

Finally, the procedure List-delete deletes an element given by a reference from a list. Its running time is O(1). If, however, the element is given by a key, then we first have to use List-search to find the element.

Algorithm 23 List-insert

Input: a list L, an element x
Output: appends x to the front of the list

```
1: Next(x) := head
2: if head ≠ NULL then
3:     Prev(head) := x
4: head := x
5: Prev(x) := NULL
```

Algorithm 24 List-delete

Input: a list L, an element x
Output: removes x from the list

```
1: if Prev(x) ≠ NULL then
2:     Next(Prev(x)) := Next(x)
3: else
4:     head := Next(x)
5: if Next(x) ≠ NULL then
6:     Prev(Next(x)) := Prev(x)
```

Stacks

IsEmpty: O(1) time, checks whether the stack is empty

Push: O(1) time, adds an element to the stack

Pop: O(1) time, removes an element to the stack

Stacks use the *last-in-first-out* principle.

Queues

IsEmpty: O(1), checks whether queue is empty

Enqueue: O(1), adds an element to the queue

Dequeue: O(1), removes an element from the queue

Queues use the *first-in-first-out* principle.

Doubly linked lists

List-search: O(n), finds an element with a given key

List-insert: O(1), adds an element to the front of the list

List-delete: O(1), removes an element given by a reference

6 Binary search trees

In many applications, like data bases, the data is dynamic: Elements are inserted or deleted and we try to maintain the data in such a way that other operations can be supported efficiently. Linked lists for instance support such operations but the search is not efficient in lists.

Binary search trees are a simple data structure that support many dynamic set operations quite efficiently, in particular

Search: Finds an element with a particular key value (or detects that there is no such element).

Minimum: Finds the element with the smallest key

Maximum: Finds the element with the largest key

Pred: Finds the element before a given element (according to the keys)

Succ: Finds the element after a given element

Insert: Inserts a new element

Delete: Deletes a given element

Binary search trees are rooted and ordered binary trees that fulfill the binary search tree property (see below). The running time of the methods above is bounded by the height of the trees. The height of binary search trees "behaves like" the running time of quick sort. In the good cases and even on the average, the height is logarithmic. In the worst case, however, it is linear. There are variations of binary search trees that are guaranteed to have height that is bounded by $O(\log n)$, n being the number of nodes in the tree (= number of elements stored). We will study one of them in the next chapter.

We represent trees as a linked data structure. There are four basic methods:

Parent(v) returns the parent of v (and NULL, if v is the root).

Left(v) returns the left child of v (and NULL, if v has no left child).

Right(v) returns the right child of v (and NULL, if v has no right child).

Key(v) returns the key of v according to which the elements will be sorted.

Binary search trees fulfill the *binary search tree property*:

Binary search tree property

$\text{Key}(v) \leq \text{Key}(w)$ for all nodes w in the right subtree of v,
$\text{Key}(v) \geq \text{Key}(w)$ for all nodes w in the left subtree if v.

Every subtree of a binary search tree T is again a binary search tree. If x is a node in T, we denote the subtree of T with root x by $T(x)$.

The binary search tree property enables us to output the elements sorted by key. This is done via a so-called *inorder walk*.

Algorithm 25 Inorder-walk

Input: A node x of a binary search tree T
Output: prints the elements of $T(x)$ in increasing key order

```
1: if x ≠ NULL then
2:    Inorder-walk(Left(x))
3:    output x;
4:    Inorder-walk(Right(x))
```

Exercise 6.1 *1. Prove that* Inorder-walk *is correct.*

2. Prove that the running time of Inorder-walk *is* $O(n)$ *where* n *is the number of elements in* T*.*

Exercise 6.2 *1. Can you reconstruct the binary search tree from the output of* Inorder-walk*?*

2. A preorder walk *first outputs the root and then deals with the left and right subtrees, i.e., lines 2 and 3 of the procedure* Inorder-walk *are exchanged. Can you reconstruct the binary search tree from the output of a preorder walk?*

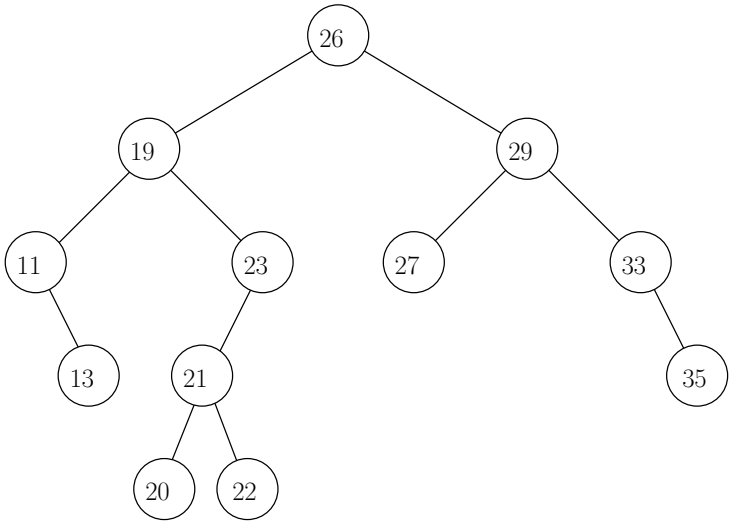

Figure 6.1: A binary search tree

6.1 Searching

If given a key k, we can easily check whether there is a node v in T with $\text{Key}(v) = k$. We just start at the root r of T. If $\text{Key}(r) = k$, then we are done. If $\text{Key}(r) < k$, then we have to look in the left subtree and otherwise in the right subtree.

Algorithm 26 BST-search

Input: node x, key k

Output: a node y with $\text{Key}(y) = k$ if such a y exists, NULL otherwise

```
1: if x = NULL or k = Key(x) then
2:     return x
3: if k < Key(x) then
4:     return BST-search(Left(x), k)
5: else
6:     return BST-search(Right(x), k)
```

Lines 1 and 2 deal with case when we either found our node or a node with key k is not in the tree. In the latter case, $x = \text{NULL}$, i.e, we were in a node v without any left or right child and called BST-search with $\text{Left}(v)$ or $\text{Right}(v)$, respectively, which is NULL. The running time of BST-search is bounded by $O(h)$ where h is the height of T.

6.2 Minimum and maximum

We can find the minimum in a binary search tree by always going to the left child until we reach a node that has no left child.

Algorithm 27 BST-minimum

Input: a node x in a binary search tree T
Output: the node in $T(x)$ with minimum key

```
1: if Left(x) ≠ NULL then
2:     return BST-minimum(Left(x))
3: return x
```

Lemma 6.1 BST-minimum *is correct.*

Proof. The proof is by induction in the height of $T(x)$.
Induction base: All elements in $T(\text{Right}(x))$ have a key that is $\geq \text{Key}(x)$. Thus, if $\text{Left}(x) = \text{NULL}$, then x must be the element with the smallest key.
Induction step: Assume that $\text{Left}(x) \neq \text{NULL}$. By the induction hypothesis, BST-minimum(Left(x)) returns the minimum element y of $T(\text{Left}(x))$. But since x and all elements in $T(\text{Right}(x))$ have a larger key by the binary search property, y is also the minimum element of $T(x)$. ∎

The correctness of the procedures in the next sections can be shown in a similar way, though the proofs get more complicated. But since binary search trees occurred in the "Programmierung 1+2" lectures, we will skip these proofs here.

Exercise 6.3 *Write the corresponding procedure for the Maximum.*

6.3 Successor and predecessor

Often it is useful to find the successor of a given node x. This means that we are looking for the node with the smallest key that is greater than $\text{Key}(x)$ (assuming that all keys are distinct). This successor is completely determined by the position in the tree, so we can find it without comparing any elements. (And this also shows that everything works if the keys are not all different...)

If a node x has a right child, that is $\text{Right}(v) \neq \text{NULL}$, then the successor simply is the node in $T(\text{Right}(x))$ with minimum key. If x does not have a right child, then the successor of x is the lowest ancestor of x that has a left child that is also an ancestor of x.

BST-successor computes the successor of a node x. The second part deals with the case that x has no right child. We go up the paths from x to the root. y is always the parent of x. We stop when either $y = \text{NULL}$ or $x = \text{Left}(y)$. In the first case, x has no successor and NULL is returned. In

Algorithm 28 BST-successor

Input: node x in a binary search tree T
Output: the successor of x, if x has a one, NULL otherwise

```
1: if Right(x) ≠ NULL then
2:    return BST-minimum(Right(x))
3: y := Parent(x)
4: while y ≠ NULL and x ≠ Left(y) do
5:    x := y
6:    y := Parent(y)
7: return y
```

the second case, y is the lowest ancestor the left child of which is also an ancestor. The running time is again linear in the height of T.

Exercise 6.4 *Write the corresponding procedure for finding the predecessor.*

6.4 Insertion and deletion

Now we want to insert a node z. Key(z) is initialized, but Left(z), Right(z) and Parent(z) are all NULL. To insert a node in a binary search tree, similar to the procedure BST-search, we trace down a path from the root until we would have to go to a child that is not present. This is the place where z has to be inserted.

Algorithm 29 BST-insert

Input: node x of a binary search tree T, new node z
Output: inserts z into $T(x)$ maintaining the binary search tree property

```
1: if x = NULL then
2:    z becomes the root of T
3: else
4:    if Key(z) < Key(x) then
5:       if Left(x) ≠ NULL then
6:          BST-insert(Left(x), z)
7:       else
8:          Parent(z) := x
9:          Left(x) := z
10:   else
11:      if Right(x) ≠ NULL then
12:         BST-insert(Right(x), z)
13:      else
14:         Parent(z) := x
15:         Right(x) := z
```

In lines 1 and 2, we check whether the tree is empty. In this case, z is the new root of T. This check is only done once, since we ensure that in all recursive calls that x is not NULL. So one gets a somewhat more efficient code by checking this in the beginning once and then removing this check in the recursive calls. In line 3, we test whether z belongs to the left or right subtree of x. Then there are two cases: If x has a corresponding child, we do recursion. Otherwise, we found the place to insert z.

For deletion, we are given a tree T and a node x in T. We consider three different cases:

x **has no child:** In this case, we can just remove x.

x **has one child:** In this case, we can just remove x and connect the child of x to the parent of x. (If x does not have a parent, i.e., is the root, then the child of x becomes the new root.

x **has two children:** Then we first search the successor of x. The successor y of x has at most one child. So we first delete y as in one of the first two cases and then replace x by y.

In lines 1 to 4, we select the node to delete. In line 19, we also have to copy anything else that is stored into the node.

Exercise 6.5 *Prove that the insertion and deletion procedures keep the binary search tree property.*

Theorem 6.2 *Searching, minimum and maximum computation, predecessor and successor computation and insert and delete run in time* $O(h)$ *on binary search trees of height* h.

Algorithm 30 BST-delete

Input: node x in a binary search tree T
Output: deletes x from T

```
1: if Left(x) = NULL or Right(x) = NULL then
2:    y := x
3: else
4:    y := BST-successor(x)
5: if Left(y) ≠ NULL then
6:    v := Left(y)
7: else
8:    v := Right(y)
9: if v ≠ NULL then
10:   Parent(v) := Parent(y)
11: if Parent(y) = NULL then
12:   v becomes the root of T
13: else
14:   if y = Left(Parent(y)) then
15:      Left(Parent(y)) := v
16:   else
17:      Right(Parent(y)) := v
18: if y ≠ x then
19:   Key(x) := Key(y)
```

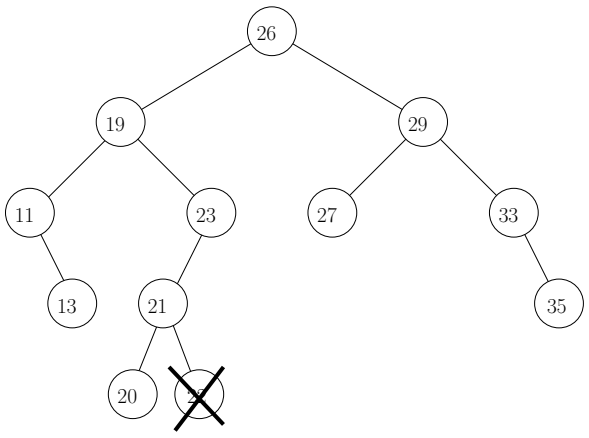

Figure 6.2: Deletion of a leaf (with key 22). It is simply removed.

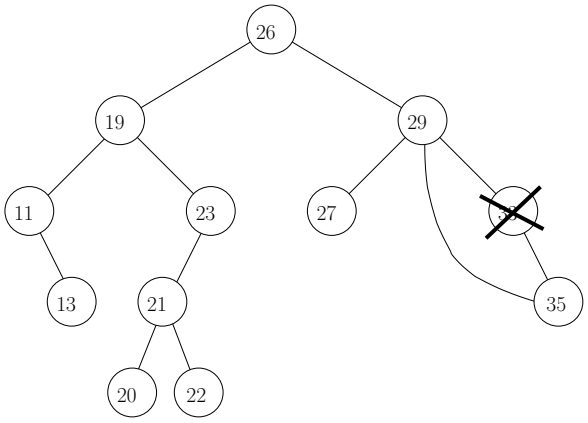

Figure 6.3: Deletion of a node with one child (with key 33). The node is removed and the child becomes a child of the parent.

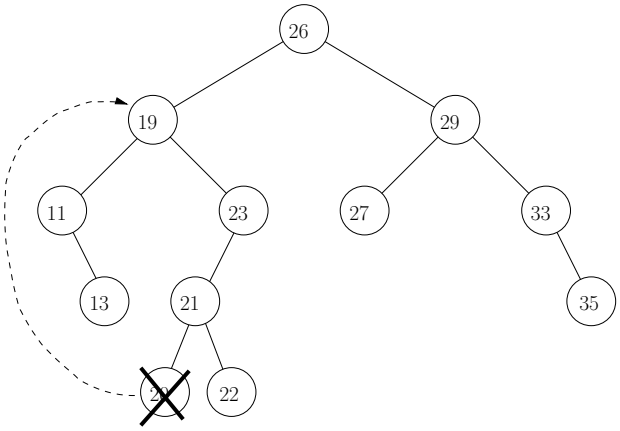

Figure 6.4: Deletion of a node with two children (with key 19). We search the successor (key 20). The successor is removed as in the first two cases and its content is copied into the node that shall be deleted.

7 AVL trees

Binary search trees of height h support a lot of operations in time $O(h)$. While one can show that on the average, a binary search tree with n elements has height $O(\log n)$, this need not be true in the worst case, in particular, if we insert elements one after another that are already partially sorted. There are several ways to create search trees that have logarithmic height even in the worst case. One such an example are *AVL trees*.

Let T be a binary search tree. Whenever a child of a node x is missing, say $\text{Left}(x) = \text{NULL}$, then we add a virtual leaf to this node. In this way, every node has either degree two or is a virtual leaf. Nodes that are not virtual leaves (i.e., the nodes from the original binary search tree) are called internal nodes. Only internal nodes carry data. We need these virtual leaves for the analysis, they are not needed in the implementation (a NULL reference is interpreted as a virtual leaf).

For every node x in T, $\text{Height}(x)$ is the height of the subtree $T(x)$ with root x. The virtual leaves have height 0. For every internal node x, the balance of x is defined as

$$\text{Bal}(x) = \text{Height}(T(\text{Left}(x))) - \text{Height}(T(\text{Right}(x))).$$

An AVL tree is a binary search tree with the extra property that for every internal node x, the height of the subtree with root $\text{Left}(x)$ and the height of the subtree with root $\text{Right}(x)$ differ by at most one.

AVL tree property

1. An AVL tree fulfills the binary search tree property.
2. For every internal node v, $\text{Bal}(v) \in \{-1, 0, 1\}$.

For the static operations like searching, finding minimal elements, etc. we can use the implementations for binary search trees. We will also use the insert and delete procedure of binary search trees, but afterwards, we have to ensure that the AVL property is restored.

AVL trees are named after their inventors, G. M. Adelson-Velsky and E. M. Landis.

If we want to implement AVL trees, we do not have to store the heights, it is sufficient to store the balance. This information can be stored in two bits which can often be squeezed somewhere in.

7.1 Bounds on the height

Definition 7.1 *The* Fibonacci numbers F_n *are defined recursively by*

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

Exercise 7.1 *Show that*

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

for all n (hint: induction on n). This is called the Moivre-Binet formula.

Excursus: Fibonacci numbers

Fibonacci numbers described the growth of a population living under ideal (?) conditions. Assume you start with a pair of rabbits. After one month, they bear another pair of rabbits at the end of every month. Every new pair does the same after one month of waiting and rabbits never die. Then F_i is the number of pairs of rabbits in the ith month.

Note that the base $\frac{1-\sqrt{5}}{2}$ of the second term is in $(-1, 0)$. Therefore, this term goes to 0 very fast and oscillates around zero. The growth of the Fibonacci numbers is dominated by the first term, the base of which is in $(1, 2)$. Thus, the growth is exponential.

Lemma 7.2 *For every AVL tree T with n internal nodes, the height of T is bounded by* $\mathrm{O}(\log n)$.

Proof. We show by induction on h that every AVL tree of height h has at least F_{h+1} virtual leaves.
Induction base: An AVL tree of height 0 has 1 virtual leaf. An AVL tree of height 1 has 2 virtual leaves.
Induction step: Let T be an AVL tree with height h. Let T_1 and T_2 be its subtrees. One of them, say T_1, has height $h-1$, the other one has height $\geq h-2$ by the AVL tree property. By the induction hypothesis, T_1 has at least F_h leaves and T_2 has at least F_{h-1} leaves. Therefore, T has a least $F_h + F_{h-1} = F_{h+1}$ leaves.

The claim of the lemma follows from this: Let T be an AVL tree with n internal nodes (and $n+1$ virtual leaves) and height h. We have

$$n+1 \geq F_{h+1} = \frac{1}{\sqrt{5}} \left(\left(\tfrac{1+\sqrt{5}}{2} \right)^{h+1} - \left(\tfrac{1-\sqrt{5}}{2} \right)^{h+1} \right) \geq \frac{1}{2\sqrt{5}} \left(\tfrac{1+\sqrt{5}}{2} \right)^{h+1}.$$

Therefore $h \in \mathrm{O}(\log n)$. ∎

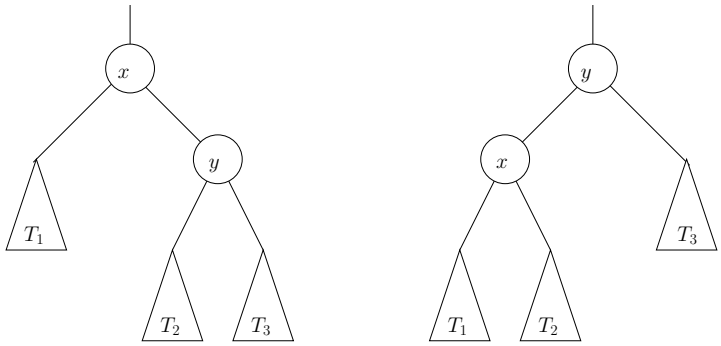

Figure 7.1: A left rotation around x transforms the tree on the left hand side into the tree on the right hand side. A right rotation around y transforms the tree on the right hand side into the tree on the left hand side.

7.2 Restoring the AVL tree property

Since every AVL tree is a binary search tree, it automatically supports the static methods Search, Minimum, Maximum, Pred, and Succ. Since their running time is $O(h)$ and the height h of any AVL tree is $O(\log n)$, their running time is $O(\log n)$. Insert and Deletion are also performed in the same way, however, after inserting or deleting a node, the AVL tree property might be violated and we have to restore it.

7.2.1 Rotations

Rotations are used to restructure binary search trees. A *left rotation* around x transforms the tree on the left hand side in Figure 7.1 into the tree on the right hand side. A *right rotation* around y transforms the tree on the right hand side in Figure 7.1 into the tree on the left hand side. In this sense, left and right rotations are inverse operations.

If the tree on the left hand side fulfills the binary search tree property, then the tree T_1 contains all nodes the key of which is smaller than Key(x). The tree T_2 contains all node with a key between Key(x) and Key(y). And T_3 has all nodes with a key greater than Key(y). But then the tree on the right hand side fulfills the binary search tree property, too. The same is true for the right rotation.

Lemma 7.3 *Left and right rotations preserve the binary search tree property.*

7.2.2 Insertion

When a node is inserted, it always becomes a leaf in the tree. But because we added the virtual leaves, inserting a node means that a virtual leaf is

replaced by an internal node with height 1 having two virtual leaves. So a tree of height 0 is replaced by a tree of height 1. This means that potentially the AVL tree property is violated.

Observation 7.4 *After inserting a node y, the AVL tree property can only be violated at nodes on the path from y to the root.*

Algorithm 31 restores the AVL property. y is the current node the height of which is increased. If y is the left child of its parent x, then we have to increase the balance of x by 1 because the height of the left subtree of x increased. If y is the right child, then the balance of x is decreased by 1. If the balance of x becomes 0 in this way, then we are done, because the height of $T(x)$ did not change.[1]

Algorithm 31 AVL-insert-repair

Input: AVL tree T, a node y inserted into T
Output: afterwards, the AVL property is restored

```
1:  x := Parent(y)
2:  while x ≠ NULL do
3:    if y = Left(x) then
4:      Bal(x) := Bal(x) + 1
5:    else
6:      Bal(x) := Bal(x) − 1
7:    if Bal(x) = 0 then
8:      return
9:    if Bal(x) = 2 or Bal(x) = −2 then
10:     Restore the AVL property using a rotation or double rotation (see
        Figure 7.1 and 7.2)
11:     return
12:   y := x; x := Parent(y)
```

If the balance of x became $+1$ or -1, then the height of $T(x)$ was increased by 1, so we need to inspect the parent of x in the next iteration of the while loop. If the balance of x became $+2$ or -2, then the AVL tree property is violated and we have to repair it. We only consider the case when the balance of x is -2, the other case is completely symmetric. If the balance of x became -2, then $y = \text{Right}(x)$ and the balance of y is either $+1$ or -1. We distinguish between these two cases:

$\text{Bal}(y) = -1$ after the insertion: In this case, we perform a left rotation as in Figure 7.1. The following table show all relevant parameters

[1] The balance of x was $+1$ or -1 before the insertion, that is, the height of one of its subtrees was one less than the height of the other. After the insertion, both of them have the same height now. Since the nodes above x do not "see" this difference, we can stop.

before the insertion, after the insertion, and after the rotation. We set $h := \text{Height}(T_1)$. The height of the other subtrees are then determined by the balances of x and y. Because we assumed that $\text{Bal}(y) = -1$ after the insertion, the insertion took place in T_3.

	before insertion	after insertion	after rotation
$\text{Bal}(x)$	-1	-2	0
$\text{Bal}(y)$	0	-1	0
$\text{Height}(T_1)$	h	h	h
$\text{Height}(T_2)$	h	h	h
$\text{Height}(T_3)$	h	$h+1$	$h+1$
$\text{Height}(T(x))$	$h+2$	$h+3$	$h+1$
$\text{Height}(T(y))$	$h+1$	$h+2$	$h+2$

After the rotation, all balances are in $\{+1, 0, -1\}$. Furthermore, since $\text{Height}(T(x))$ before the insertion is the same as $\text{Height}(T(y))$ after the insertion, we can stop, because there cannot be any further violations above y.

$\text{Bal}(y) = 1$ after the insertion: In this case, we perform a left double rotation as depicted in Figure 7.2. We set $h := \text{Height}(T_1)$. The height of the other subtrees are then determined by the balances of x, y, and z. Because we assumed that $\text{Bal}(y) = 1$ after the insertion, the insertion took place in $T(z)$:

	before insertion	after insertion	after rotation
$\text{Bal}(x)$	-1	-2	0 or $+1$
$\text{Bal}(y)$	0	1	-1 or 0
$\text{Bal}(z)$	0	$+1$ or -1	0
$\text{Height}(T_1)$	h	h	h
$\text{Height}(T_2)$	$h-1$	$h-1$ or h	$h-1$ or h
$\text{Height}(T_3)$	$h-1$	$h-1$ or h	$h-1$ or h
$\text{Height}(T_4)$	h	h	h
$\text{Height}(T(x))$	$h+2$	$h+3$	$h+1$
$\text{Height}(T(y))$	$h+1$	$h+2$	$h+1$
$\text{Height}(T(z))$	h	$h+1$	$h+2$

After the double rotation, all balances are in $\{+1, 0, -1\}$. Furthermore, the height of $T(x)$ before the insertion is the same as the height of $T(z)$ after the insertion. Therefore, the procedure can stop.

In both cases, the balances are restored and moreover, we can leave the procedure.

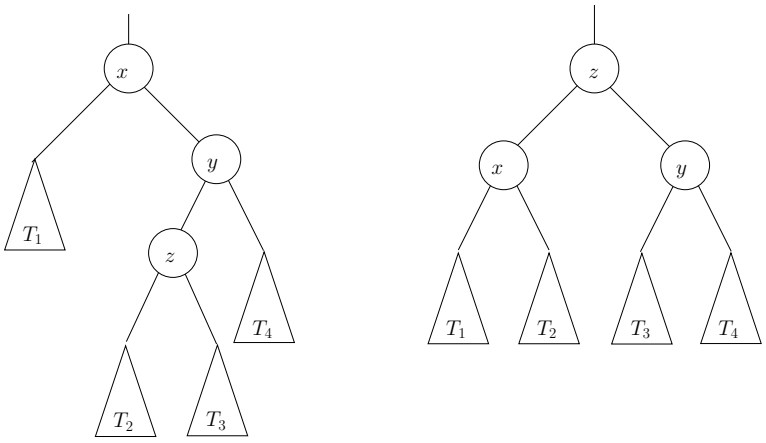

Figure 7.2: A left double rotation. You can think of a double left rotation as a right rotation around y followed by a left rotation around x. This explains the name double rotation. A double rotation also preserves the binary search tree property.

7.2.3 Deletion

Deletion is performed similar to insertion. We use the delete method from binary search trees and then move the path up to the root and try to restore the AVL tree property

When we delete a node in a binary search tree, three cases can occur. Either it is a leaf in the binary tree, that means, it is an internal node with two virtual leaves in our AVL tree. Or it is an internal node with only one child. In an AVL tree, this means that one of its children is a virtual leaf.[2] Or it is a node with two children that are internal nodes, too. Then we delete its successor and copy the content of the successor to the node.

Let v be the node deleted. We assume that there are no references from nodes in the tree to v, but that Parent(v) still references to the former parent of v in T. After deletion, we call the method AVL-delete-repair with v. The procedure works in a similar way to AVL-insert-repair by going up the path from v to the root. However, some things have to be changed: First of all, the balance of x has to be adapted in the opposite way in lines 4 and 6. Then, we can stop if the balance of x becomes -1 or 1. This means that the balance was 0 before and the height of one of the subtrees was decreased. But since this does not change the total height of the tree, then we stop.

If the balance becomes 2 or -2, then things get more complicated. Assume that Bal(x) $= -2$, the other case is treated similarly. In this case, v is a left child of x. Let y be the right child of x. We distinguish three cases:

[2]Note that by the AVL tree property, the other child then is an internal node with two virtual leaves.

Warning! We use the same figures as in the case of insertion. However, the node deleted is in the left subtree of x. Moreover, y is *not* the node currently inspected by AVL-delete-repair. (This is v which is the root of T_1.)

$\mathrm{Bal}(y) = -1$: We use a left rotation like in Figure 7.1. Let $h = \mathrm{Height}(T_1)$.

	before deletion	after deletion	after rotation
$\mathrm{Bal}(x)$	-1	-2	0
$\mathrm{Bal}(y)$	-1	-1	0
$\mathrm{Height}(T_1)$	h	$h-1$	$h-1$
$\mathrm{Height}(T_2)$	$h-1$	$h-1$	$h-1$
$\mathrm{Height}(T_3)$	h	h	h
$\mathrm{Height}(T(x))$	$h+2$	$h+2$	h
$\mathrm{Height}(T(y))$	$h+1$	$h+1$	$h+1$

Since the height of $T(x)$ before the deletion is larger than the height of $T(y)$ after the rotation, we have to go on and move up the path to the root.

$\mathrm{Bal}(y) = 0$: Again, we do a left rotation. Let $h := \mathrm{Height}(T_1)$.

	before deletion	after deletion	after rotation
$\mathrm{Bal}(x)$	-1	-2	0
$\mathrm{Bal}(y)$	0	0	1
$\mathrm{Height}(T_1)$	h	$h-1$	$h-1$
$\mathrm{Height}(T_2)$	h	h	h
$\mathrm{Height}(T_3)$	h	h	h
$\mathrm{Height}(T(x))$	$h+2$	$h+2$	$h+1$
$\mathrm{Height}(T(y))$	$h+1$	$h+1$	$h+2$

Since the height of $T(x)$ before the deletion is the same as the height of $T(y)$ after the rotation, we can stop in this case.

$\mathrm{Bal}(y) = +1$: In this case, we perform a left double rotation. Now we look at Figure 7.2. Let $h := \mathrm{Height}(T_1)$.

	before deletion	after deletion	after rotation
$\mathrm{Bal}(x)$	-1	-2	0 or $+1$
$\mathrm{Bal}(y)$	$+1$	$+1$	0 or -1
$\mathrm{Bal}(z)$	-1, 0, or $+1$	-1, 0, or $+1$	0
$\mathrm{Height}(T_1)$	h	$h-1$	$h-1$
$\mathrm{Height}(T_4)$	$h-1$	$h-1$	$h-1$
$\mathrm{Height}(T(x))$	$h+2$	$h+2$	h
$\mathrm{Height}(T(y))$	$h+1$	$h+1$	h
$\mathrm{Height}(T(z))$	h	h	$h+1$

All balances are restored, but since the height of $T(x)$ before the deletion is larger than the height of $T(z)$ after the rotation, we have to go up.

Algorithm 32 AVL-delete-repair

Input: AVL tree T, a node v deleted from T
Output: afterwards, the AVL property is restored

```
1:  x := Parent(v)
2:  while x ≠ NULL do
3:    if v = Left(x) then
4:      Bal(x) := Bal(x) − 1
5:    else
6:      Bal(x) := Bal(x) + 1
7:    if Bal(x) = 1 or Bal(x) = −1 then
8:      return
9:    if Bal(x) = 2 or Bal(x) = −2 then
10:     Restore the AVL property using a rotation or double rotation
11:   v := x; x := Parent(v)
```

Both repair procedures trace a path of length $O(\log n)$. Each rotation takes time $O(1)$, since we have to update a constant number of references.

Theorem 7.5 *Searching, minimum and maximum computation, predecessor and successor computation and insert and delete run in time* $O(\log n)$ *on a AVL tree with* n *internal nodes.*

8 Binomial Heaps

Mergeable heaps are heaps that support a union operation. That is, given heaps H and H', we can merge them into one big heap that contains all elements of H and H' (and, of course, satisfies the heap property). Take our ordinary heaps from heap-sort. To implement the union operation, we can just concatenate the two arrays and call the Heapify procedure. However, this takes time $O(n)$ which is too slow for large sets of data. In this chapter, we will introduce *binomial heaps* which support all heap operations in time $O(\log n)$ and, in addition, the union operation in time $O(\log n)$, too. In particular, binomial heaps support the following operations:

Make-heap(): creates a new heap that is empty.

Minimum(H): returns a pointer/reference to an element in H that has a minimum key (but leaves the element in H).

Extract-min(H): removes an element with the minimum key from H and returns (a pointer/reference to) the node.

Insert(H, x): inserts x into H.

Delete(H, x): deletes x from H.

Union(H, H'): creates a new heap that contains all elements from H and H'.

Decrease-key(H, x, k): assigns the new key value k to the element x provided that k is smaller than the current value.

The Decrease-key procedure is useful if we want to implement mergeable priority queues.

8.1 Binomial trees

A *binomial tree* B_k of order k is an ordered tree that is defined recursively:

1. B_0 is the tree with one node.
2. The tree B_k consists of two copies of B_{k-1}. The root of one copy becomes the the leftmost child of the root of the other copy.

Note that B_k only "describes a structure", later we will fill the nodes with data.

Lemma 8.1 *The following statements are true:*

1. *B_k has 2^k nodes.*
2. *The height of B_k is k.*
3. *There are $\binom{k}{i}$ nodes at depth i, $0 \leq i \leq k$.*
4. *The root has k children and all other nodes have strictly less.*

Proof. The proof of each of the statements is by induction on k.

1. B_0 has $1 = 2^0$ nodes. Now assume that B_{k-1} has 2^{k-1} nodes (induction hypothesis). B_k consists of two copies of B_{k-1} and has $2 \cdot 2^{k-1} = 2^k$ nodes.
2. B_0 has height 0. Assume that B_{k-1} has height $k-1$. The height of B_k is the height of B_{k-1} plus 1, hence it is $k-1+1 = k$.
3. In B_0, there is exactly one node at depth 0. Now assume that in B_{k-1}, there are exactly i nodes at depth $\binom{k-1}{i}$. How many nodes are at depth i in B_k? There are $\binom{k-1}{i}$ in the copy of B_{k-1} that contains the root of B_k and $\binom{k-1}{i-1}$ in the one that does not contain the root. Altogether, these are
$$\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}.$$[1]
4. In B_0, the root has no children and there are no further nodes. Assume that in B_{k-1}, the root has $k-1$ children and all other nodes have less children. The root in B_k has k children. All other nodes have less children, since they belong to one copy of B_{k-1} and no further edges are added except the one joining the two copies. ∎

Binary trees were easy to store. Every node had three fields containing pointers/references to the left and right child and the parent. Binomial trees have an unbounded number of children. We can of course keep a list of all children at every node. More compact is the so-called *left-child-right-sibling representation*: Every node has a pointer/reference to its parent, its left-most child, and its sibling to the right. If the node is the root, then the parent

[1] How does one prove this identity? Either you use brute force or you can do the following: The right hand side is the number of possibilities to choose i out of k items. Now mark one of the k items. To take i of the k items, you either can take the marked one and $i-1$ out of the $k-1$ remaining items or you do not take the marked one but i out of the $k-1$ remaining ones. But this is precisely the sum on the left-hand side.

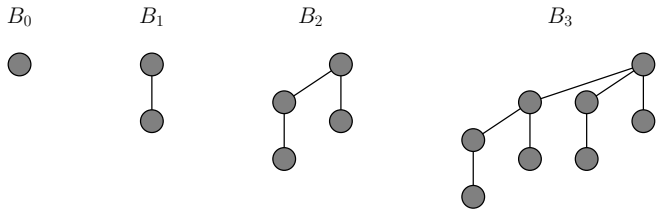

Figure 8.1: The binomial trees B_0, B_1, B_2, and B_3

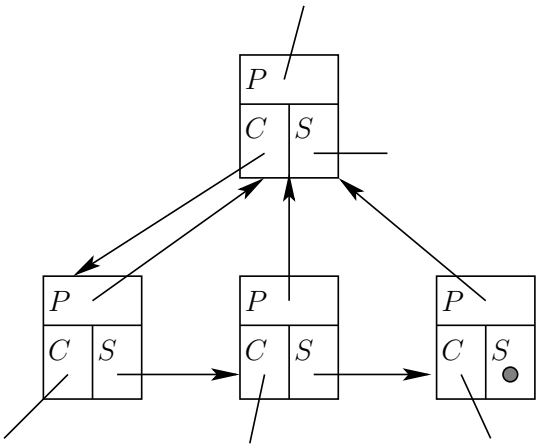

Figure 8.2: The left-child-right-sibling representation of a node in a tree with three children. The three types of pointers are marked with P (parent), C (child), and S (sibling).

pointer is NULL. If the node is a leaf, then the pointer to the left-most child is NULL. If it is the last (right-most) child, then the pointer to the right sibling is NULL. We will also store at each node x its number of children and denote this quantity by Degree(x).

8.2 Binomial heaps

A *binomial heap* is a set of binomial trees. Every tree in this set has the structure of a binomial tree but it contains additional data like the key and a pointer/reference to satellite data. A binomial heap satisfies the *binomial heap property*:

Binomial heap property

1. Every binomial tree in the heap is *heap-ordered*: the key of the parent is smaller than or equal to the key of the children.
2. The binomial heap contains at most one copy of B_k for every k.

Technically, we store the trees in a linked list. It turns out to be useful to store the trees in a list in ascending order with respect to their sizes (or degrees of the roots, which gives the same order). To link the trees together in the list, we can use the Sibling pointer/reference of the roots, since it is not used otherwise. There is one distinguished pointer/reference Head(H) which points to the first tree in the list.

The number n of elements in the binomial heap H uniquely determines the orders of the binomial trees in the heap. If $n = \sum_{i=0}^{\ell} b_i 2^i$ is the binary expansion of n, then H will contain the trees B_i with $b_i = 1$, since the binary expansion is unique. In particular, every binomial heap with n elements has at most $\log n$ trees.

8.3 Operations

8.3.1 Make heap

To build a heap, we just initialize H as the empty list. The running time is obviously O(1).

Algorithm 33 Make-BH

Input: binomial heap H
Output: empties H

Head(H) := NULL (Old-fashioned languages like C/C++ require that you free the memory)

8.3.2 Minimum

By item 1 of the binomial heap property, the minimum is stored in one of the roots of the trees. Therefore we only have to search the roots of the trees. The next algorithm is a simple minimum selection algorithm on lists. n runs through the list and in x and *min*, we store the current minimum and its value. Since a binomial heap contains $\leq \log n$ trees, the running time of BH-Minimum is O($\log n$).

Algorithm 34 BH-Minimum

Input: binomial heap H
Output: the minimum element stored in H

```
n := Head(H)
x := n
if n ≠ NULL then
   min := Key(n)
   n := Next(n)
while n ≠ NULL do
   if Key(n) < min then
      min := Key(n)
      x := n
   n := Next(n)
return x
```

8.3.3 Union

The union routine is the heart of binomial heaps. All further operations are somehow based on this routine. As a subroutine, we need a procedure that takes two B_{k-1} and produces a B_k. We move the references/pointers in the obvious way.

Algorithm 35 BT-union

Input: the roots x and y of two copies of B_{k-1}
Output: the trees are linked together to form a B_k with root y

```
Parent(x) := y
Sibling(x) := Child(y)
Child(y) := x
Degree(y) := Degree(y) + 1
```

Remark 8.2 *If* $\text{Key}(y) \leq \text{Key}(x)$ *and both trees above are heap-ordered, then the resulting tree is heap-ordered, too.*

To union two binomial heaps H_1 and H_2, we first merged the two lists of trees into one list. The trees are ordered with ascending degrees. By the binomial heap property, every tree B_k occurs at most once in each H_i, thus it occurs at most twice in the merged list. We will go through the list from left to right. Whenever we encounter two copies of B_k, we will merge them into one B_{k+1} by exploiting BT-union. In this way, we will ensure that left to the current position, every B_k appears at most once and right to the current position, every B_k appears at most twice. The only exception is the current position: The B_k in this position may appear three times! How can this happen when in the beginning, every B_k only appears at most twice? Well,

we can have two B_{k-1} followed by two B_k in the merged list. The two B_{k-1} will be merged to one B_k and then there are three B_k's in a row. We leave the first one of them in the list and merge the other two to one B_{k+1} in the next step.

There will be three pointers/references in our procedure, *prev*, *cur*, and *next*. They point to the previous, current, and next tree in the merged list. We distinguish three cases. The third one splits into two subcases.

Degree(***cur***) $\neq$ Degree(***next***)**:** In this case, the degrees are different. We do not have to merge the current tree and can move on with the next tree.

Degree(***cur***) = Degree(***next***) = Degree(Sibling(***next***))**:** In the second case, we have three trees of the same degree in a row and we are standing on the first one. We go one step to the right and then will merge the next two trees in the next step.

Degree(***cur***) = Degree(***next***) $\neq$ Degree(Sibling(***next***))**:** We merge the two trees of the same degree to form a tree of larger degree. This case has two subcases: If the first tree has the root with smaller key, then it becomes the root of the new tree. Otherwise, the root of the second tree, becomes the new root.

If case 2 is executed, then case 3 is executed in the next iteration of the while-loop.

Lemma 8.3 *The following invariant holds after executing cases 1 or 3:*

1. *The degree of all trees left to cur is* $<$ Degree(*cur*).
2. *The degrees of all trees left to cur are distinct.*
3. *Right to cur, the degree of all trees is* $\geq$ Degree(*cur*).
4. *Right to cur, every degree appears at most twice,* Degree(*cur*) *might appear three times.*

Proof. The proof is by induction on the number of executions of the body of the while loop. Before the first execution of the while loop, the invariant is certainly fulfilled.

If case 1 is executed, then Degree(*cur*) $<$ Degree(*next*), so the invariant is still valid after setting *cur* := *next*.

If case 3 is executed, then Degree(*prev*) $<$ Degree(*cur*) = Degree(*next*) or Degree(*prev*) = Degree(*cur*) = Degree(*next*). In the latter case, case 2 has been executed before. Now *cur* and *next* are merged into one tree of bigger degree. After this, Degree(*prev*) $<$ Degree(*cur*). It might now be the case that Degree(*cur*) = Degree(*next*) = Degree(Sibling(*next*)). But right to

Algorithm 36 BH-union

Input: binomial heaps H_1, H_2
Output: H_1 and H_2 merged

```
H := Make-BH()
Merge the two lists of binomial trees of H_1 and H_2 into one list with
ascending degrees.
prev := NULL; cur := Head(H); next := Sibling(cur);
while next ≠ NULL do
  if (Degree(cur) ≠ Degree(next)) or (Sibling(next) ≠ NULL) and
  (Degree(Sibling(next)) = Degree(cur)) then   /* CASES 1,2 */
    prev := cur
    cur := next
  else
    if Key(cur) ≤ Key(next) then   /* CASE 3A */
      Sibling(cur) := Sibling(next)
      BT-union(next, cur)
    else   /* CASE 3B */
      if prev = NULL then
        Head(H) := next
      else
        Sibling(prev) := next
      BT-union(cur, next)
      cur := next
  next := Sibling(cur)
```

this, every degree occurs at most twice. Hence the invariant is still fulfilled after setting $cur := next$. ■

Thus, BH-union is correct. It runs in time $O(\log n)$ where n is the number of elements in the output heap, since the heaps only have $\leq \log n$ trees and we only spend $O(1)$ time in the while loop per tree.

8.3.4 Insert

With the union operation, inserting an element x into a binomial heap H is very easy: We pack x into a binomial heap and then merge this heap with H. The overall running time is obviously $O(\log n)$.

Algorithm 37 BH-insert
Input: binomial heap H, element x
Output: inserts x into H
$H_1 :=$ Make-BH()
Parent(x) := Child(x) := Sibling(x) := NULL
Degree(x) := 0
Head(H_1) := x
$H :=$ BH-union(H, H_1)

8.3.5 Extract-min

We saw already that an element with a minimum key can only be a root of a tree. But if we extract the minimum, then we destroy the binomial tree. The next lemma states that the remaining parts of the tree still form a binomial heap. Thus we can merge the resulting two heaps.

Lemma 8.4 *The tree B_k is obtained by attaching $B_{k-1}, \ldots, B_0$ to a single node (the root) as children from left to right.*

Proof. The proof is by induction on k.
Induction base: If $k = 0$, then there is nothing to prove.
Induction step: Assume that the statement is true for B_{k-1}. We obtain B_k by taking two copies of B_{k-1} and making the root of one of them the left-most child of the other. By the induction hypothesis, we can replace the B_{k-1} the root of which became the root of B_k by a root with $B_{k-2}, \ldots, B_0$ attached to it. It follows that B_k is a node with $B_{k-1}, B_{k-2}, \ldots, B_0$ attached to it. ■

The algorithm is written in Pseudopseudocode. The running time is dominated by the BH-union call. Thus it is $O(\log n)$.

Algorithm 38 BH-extract-min

Input: binomial heap H
Output: returns an element with minimum key and removes it from H
Use the Minimum procedure to find a root x with minimum key.
Remove the corresponding tree from H.
$H_1 :=$ Make-BH()
$H_1 :=$ list of subtrees of x in reverse order
$H :=$ BH-union(H, H_1)
return x

8.3.6 Delete

Exercise 8.1 *Implement the Decrease-key procedure.*

With the Decrease-key operation, deleting given elements is easy. We set its key to $-\infty$ and just call Extract-minimum. $-\infty$ is a value that is smaller than any other key. Either one has to artificially increase the range of the key values. Or one just merges the code of Decrease-key and Extract-minimum explicitly. In the latter case, we do not need the value $-\infty$. We just introduced it to produce shorter code.

Algorithm 39 BH-delete

Input: binomial heap H, element x in H
Output: x is removed from H
BH-decrease-key$(H, x, -\infty)$
BH-extract-min(H)

9 Fibonacci heaps

Fibonacci heaps provide the same operations as binomial heaps, but with a better running time for most of the operations. For some of the operations, this is only achieved on the average in a sense made precise in the next section. Fibonacci heaps achieve this improvement by being lazy; binomial heaps ensure that after every operation, the binomial heap property is fulfilled. Fibonacci heaps, on the other hand, tidy up only once after a while.[1]

9.1 Amortized analysis

Consider the following problem: Given an array $a[0..\ell-1]$ which we interpret as an ℓ bit binary counter, what is the running time of increasing the counter by 1? The algorithm below performs this increase. Each entry $a[i]$ either holds the value 0 or 1 and $a[0]$ is the least significant bit. Of course, in the worst case, every bit is set to 1 and we have to flip all n bits (and get an overflow error). But on the average, these costs are much lower. The average costs are also called the *amortized costs*.

Algorithm 40 Increasing a binary counter

Input: a binary counter $a[0..\ell-1]$
Output: a is increased by 1

```
1: i := 0
2: while a[i] = 1 and i ≤ ℓ − 1 do
3:     a[i] := 0
4:     i := i + 1
5: if i ≤ ℓ − 1 then
6:     a[i] := 1
7: else
8:     output "overflow"
```

9.1.1 The aggregate method

How can we determine the amortized costs? When using the aggregate method, we show that for all n, every sequence of n operations takes $\leq t(n)$ time. The amortized costs per operation is therefore $\frac{t(n)}{n}$. Here $t(n)$ is the number of times we increase an entry of a. We assume that $n \leq 2^\ell$.

[1] As opposed to real life, being lazy sometimes pays off for data structures.

Assume our counter a is initialized with 0 in all entries. The least significant bit $a[0]$ is changed every time we perform an increase. $a[1]$, however, is only changed every second time, namely, when $a[0] = 1$. $a[2]$ is changed only every fourth time, when $a[0] = a[1] = 1$. In general, $a[i]$ is changed only every 2^ith time. Therefore, the total time is

$$t(n) = \sum_{i=0}^{n} \lfloor \frac{n}{2^i} \rfloor \leq n \cdot \sum_{i=0}^{n} \lfloor \frac{1}{2^i} \rfloor \leq 2n$$

and the amortized costs are

$$\frac{t(n)}{n} \leq 2.$$

You saw this method already when analyzing the running time of Build-heap for binary heaps. Now you know a fancy name for this, too.

9.1.2 The potential method

The potential method is an advanced method to estimate the amortized costs of an operation. For the binary counter example, it is overkill, but it will be useful for analyzing Fibonacci heaps.

With every "state" of the counter, we associate a *potential*. (Do not confuse this with the potential function used when analyzing comparison trees!) Let c_i be the true costs of the ith operation and Φ_i be the potential after the ith operation, then the amortized costs are defined as

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

(Think of this as the "true costs" plus the "potential difference".) Then the sum of the amortized costs is

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \Phi_n - \Phi_0 + \sum_{i=1}^{n} c_i,$$

in other words,

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i + \Phi_0 - \Phi_n.$$

If we design the potential function in such a way that $\Phi_0 - \Phi_n \leq 0$, then the sum of the a_i bounds the total costs (the sum of the c_i). Often, we will choose the potential function in such a way that it is always non-negative and $\Phi_0 = 0$. Then $\Phi_0 - \Phi_n \leq 0$ trivially holds.

It is up to you to design a good potential function. In the case of a binary counter, the potential Φ_i of the counter is the number of ones in it. Let t_i be the number of ones in the counter after the ith increase. We set $\Phi_i = t_i$. Then $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i. We have

$$\Phi_i - \Phi_{i-1} \leq -c_i + 2$$

because if the ith operations takes c_i steps, then $c_i - 1$ ones are replaced by zeros and one zero is replaced by a one. Thus

$$a_i = c_i + \Phi_i - \Phi_{i-1} \leq 2.$$

With the potential method, it is also very easy to bound the costs of n increase operations when we start in any counter state, it is simply

$$2n + \Phi_0 - \Phi_n \leq 2n + \Phi_0.$$

Since $\Phi_0 \leq \ell$, the total costs are $O(n)$ as soon as $n = \Omega(\ell)$.

Warning! We do not have to implement the potential function, we only need it for the analysis! There is no need to keep a variable *potential* in Algorithm 40 and update it every time!

9.2 Fibonacci heaps

Fibonacci heaps are a collection of unordered trees. For the moment it is okay to think of them as binomial trees with the only difference that the tree is unordered. The trees are heap-ordered. We extend the left-child-right-sibling representation: The children of a node is not stored in a singly linked linear list but in a doubly linked circular list. (You could call this "some-child-left-sibling-right-sibling" representation.) Also the roots of the trees in the Fibonacci heap are stored in a circular list. We do, however, not require that every type of tree appears at most once in the tree. We will be lazy and try to "clean up" the heap as few times as possible. There will be a pointer *min* to one of the roots in the circular list and it will always point to the root with the minimum key.

With this structure, it is easy (that means in O(1) time) to perform the following operations:

Link: Given the roots of two trees, we combine them into one tree making the root with the smaller key the root of the new tree and the other root one of its children. Note that the new tree is heap-ordered if both input trees are. See Figure 9.1 for an illustration.

Unlink: is the opposite of Link. Given a the root r of a tree and one of the children x of the root, we remove the subtree with root x from the tree with root r.

Splice: It takes two circular lists of trees and combines them into one circular list of trees. See Figure 9.2.

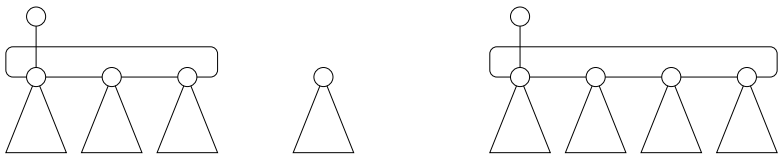

Figure 9.1: The link procedure. The link operation takes two trees and makes the the root with the larger key (on the right-hand side) the child of the other root (on the left hand side). Note that we did not draw the children of the root on the right-hand side. The unlink procedure is the opposite operation. It removes a particular subtree from the doubly linked list.

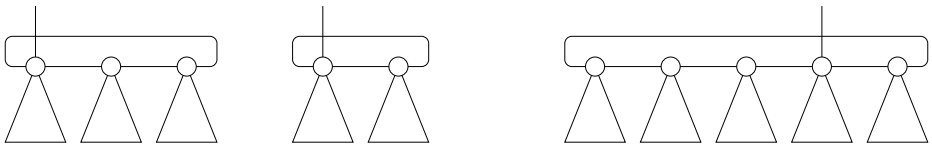

Figure 9.2: The splice procedure.

Furthermore, nodes can be marked in a Fibonacci heap. A node is marked if it has already lost one of its children. The potential of a Fibonacci heap consisting of t trees with m marked nodes is

$$t + 2m.$$

We will analyze the amortized cost of an arbitrary sequence of operations. $\Phi_i = t_i + 2m_i$ denotes the potential after the ith operation, where t_i is the number of roots after the ith operation and m_i is the number of marked nodes. If several heaps are involved in the operation, the potential is the sum of the individual potentials.

More precisely, the potential function will be some constant time the potential defined above to compensate for constant factors in the running times.

9.3 Operations

We first implement the procedures Insert, Union, Minimum, and Extract-min. If we only perform these operations, then the trees in the Fibonacci heaps will always be (unordered) binomial trees and no nodes will be marked. Nevertheless, we will explicitly write the term m_i in the potential, because everything is also valid if there are marked nodes.

9.3.1 Minimum

The procedure Minimum is easy to implement, we just have to return *min*. The actual costs for this are O(1) and there is no change in the potential. Therefore the amortized costs are O(1), too.

9.3.2 Union

The procedure Union takes the two heaps and splices them together to one heaps as in Figure 9.2. Then we choose *min* as the minimum of the two minima. That's it! We do not tidy everything up as we did for binomial heaps. We even do not sort the sizes of the trees. The actual costs are O(1). There is no change in potential, since the number of trees in the union is the sum of the number of heaps in the input heaps. Therefore, the amortized costs are O(1), too.

9.3.3 Insert

To insert an element, we create a heap of size one (using Make-heap, which we frankly did not implement so far) and then use union. This takes O(1) time. The new root has one child more, so the potential goes up by one. The amortized costs are $O(1 + 1) = O(1)$.

9.3.4 Extract-min

Extract-min is the first interesting procedure. We can find the minimum using *min*. We unlink the corresponding tree T from the heap F. When we remove the root of T, we get a ring of trees, which is nothing else than a Fibonacci heap. If we splice this ring together with F. The following step is the only time when we tidy up: Whenever we find two trees in the heap the root of which has the same degree, we link them. Note that if the two trees are unordered binomial trees, then we will get an unordered binomial tree again. So after each linking, we still have a ring of unordered binomial trees.

Algorithm 41 FH-Extract-min

Input: Fibonacci heap F
Output: the minimum is extracted and returned

1: Unlink the tree T with root *min* from F
2: Remove the root of T to obtain a circular list of trees.
Splice this list with F.
3: As long as there are two roots with the same degree in F, we link them together.
4: Recompute *min* and return the old minimum.

How fast can we implement this scheme? Steps 1 ans 2 take time O(1). It is very easy to implement step 3 by first sorting the trees by size (recall that a Fibonacci heap is lazy and does not store the trees sorted) and then proceed in a way similar to the union procedure for binomial heaps. But Step 3 can be even implemented in time linear in the number of roots after splicing: We have an array D. $D[i]$ stores a reference/pointer to a root with degree i if we have found one so far. For each root r, we run the following program:

1: $i :=$ Degree$[r]$
2: **while** $D[i] \neq$ NULL **do**
3: $\quad r' := D[i]$
4: $\quad D[i] :=$ NULL
5: $\quad r :=$ Link(r, r')
6: $\quad i := i + 1$
7: $D[i] := r$

If $D[i] =$ NULL in the beginning, then we simply set $D[i] := r$. Otherwise, we link r and $D[i]$. Then we get a tree with degree $i + 1$. If it is the first one of this degree, then we set $D[i + 1] := r$, otherwise we repeat the process. Since every root it put into D exactly once, the running time is linear in the number of roots.

Let $d(n)$ be an upper bound for the degree of any node in any Fibonacci heap on n nodes. Since every tree currently is an unordered binomial heap, $d(n) \leq \log n$. Before we extract the minimum, we have t_{i-1} roots. After splicing, we have $\leq t_{i-1} + d(n)$ many roots. After linking all roots of the same degree, we have t_i roots. Since all roots have different degrees then, $t_i \leq d(n)$. The total cost of FH-Extract-min is

$$\mathrm{O}(t_{i-1} + d(n)).$$

The potential difference is

$$t_i + 2m_i - t_{i-1} - 2m_{i-1} = t_i - t_{i-1} \leq d(n) - t_{i-1}.$$

Note that $m_i = m_{i-1}$. By scaling the potential function appropriately (to compensate for the constant in the O-notation), we get that the amortized costs are proportional to $t_{i-1} + d(n) + d(n) - t_{i-1} = 2d(n)$ and thus they are

$$\mathrm{O}(d(n)) \leq \mathrm{O}(\log n).^2$$

Proposition 9.1 *For a Fibonacci heap, the operations Minimum, Union, and Insert take constant time, even in the worst case. The operation Extract-min takes amortized time* $\mathrm{O}(\log n)$.

[2] We will later show that even with deletions and delete-min operations, $d(n) \in \mathrm{O}(\log n)$. Therefore, this bound remains valid.

9.4 More operations

The Decrease-key and the Delete operation work in a similar way. They both use a subroutine called Cascading-cut.

9.4.1 Cascading cuts

Algorithm 42 Cascading-cut

Input: Fibonacci heap F, node x

```
1: if x is not marked then
2:     Mark(x)
3:     return
4: y := Parent(x).
5: if y ≠ NULL then
6:     Unmark(x)
7:     Unlink T(x)
8:     Link T(x) to the cycle of roots of F
9:     Update min
10:    Cascading-cut(F, y)
```

Cascading-cut is called when a node lost one of its children. Cascading-cut gets a Fibonacci heap F and a node x. If x is not marked (that is, it did not loose a child so far), then x is simply marked. If x is marked, then we remove the tree $T(x)$ from its current tree and link it to the cycle of roots and x is unmarked. Then we proceed recursively with the parent of x (which now lost one of its children, namely x) until we reach a root.

9.4.2 Decrease-key

Let x be the node, the key of which shall be decreased. We remove the subtree $T(x)$ and link it to the cycle of roots. Since x is now a root, we can decrease its key without violating the heap property. Maybe we need to update *min*. Since the former parent of x lost a child, we call Cascading-cut.

Let c_i be the number of cuts performed. This includes the initial cut of $T(x)$ and all cuts performed by Cascading-cut. The costs of Decrease-key is proportional to c_i. After calling Decrease-key, the number of trees in the cycle of roots increases by c_i, there is one new tree for every cut. Therefore

$$t_i = t_{i-1} + c_i.$$

Furthermore, at most one node is newly marked and $c_i - 1$ nodes are unmarked. Hence,

$$m_i = m_{i-1} - (c_i - 1) + 1 = m_i - c_i + 2.$$

Algorithm 43 FH-decrease-key

Input: Fibonacci heap F, node x, key k with $k \leq \text{Key}(x)$
Output: Key(x) is set to k

```
1: y = Parent(x)
2: Key(x) := k
3: if y ≠ NULL then
4:    Unlink the tree T(x)
5:    Link T(v) to the cycle of roots
6: Update min
7: if y ≠ NULL then
8:    Cascading-cut(F, y)
```

The potential difference is

$$\Phi_i - \Phi_{i-1} = t_i - t_{i-1} + 2(m_i - m_{i-1}) = -c_i + 4.$$

Therefore, the amortized costs are proportional to

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 4$$

and thus they are constant. (Because we choose the factor 2 in the term $2m_i$, we get a constant here!)

9.4.3 Delete

Delete is very similar to Decrease-key. Let x be the node to be deleted. If x is the minimum, we just use extract-min. Otherwise, we first remove $T(x)$ as before. Instead of adding $T(x)$ to the cycle of roots, we first remove x and splice the cycle of the children of x with the cycle of roots of F. *min* needs not to be updated, since none of the new roots can be minimal, because the trees are heap-ordered. Since the former parent of x lost a child, we call Cascading-cut.

Let c_i be the number of cuts performed. This includes the initial cut of $T(x)$ and all cuts performed by Cascading-cut. The costs of Delete is proportional to $d(n) + c_i$. $d(n)$ for adding the new trees to the root cycle (references/pointers need to be updated) and c_i for performing the cuts. After calling Delete, the number of trees in the cycle of roots increases by $d(n) + c_i$, there is one new tree for every cut. Therefore

$$t_i \leq t_{i-1} + d(n) + c_i.$$

Furthermore, at most one node is marked and $c_i - 1$ nodes are unmarked. Hence,

$$m_i = m_{i-1} - (c_i - 1) + 1.$$

Algorithm 44 FH-delete

Input: Fibonacci heap F, node x
Output: x is deleted

```
1: if x = min then
2:    FH-extract-min(x)
3: else
4:    y = Parent(x)
5:    Unlink the tree T(x)
6:    Remove x
7:    Splice the cycle of children of x with the cycle of roots of F
8:    if y ≠ NULL then
9:       Cascading-cut(F, y)
```

The potential difference is

$$\Phi_i - \Phi_{i-1} = t_i - t_{i-1} + 2(m_i - m_{i-1}) = d(n) - c_i + 4.$$

Therefore, the amortized costs are proportional to

$$a_i = d(n) + c_i + \Phi_i - \Phi_{i-1} = 2d(n) + 4.$$

We will prove that $d(n) \in \mathrm{O}(\log n)$ in the next section. Therefore, deletion has amortized costs $\mathrm{O}(\log n)$.

Our analysis of the four other operations are still valid if we allow the operations Decrease-key and Delete. too. Therefore, we obtain the final result.

Theorem 9.2 *For a Fibonacci heap, the operations Minimum, Union, Insert, and Decrease-key take constant time, the first three even in the worst case. The operations Extract-min and Delete take amortized time* $\mathrm{O}(\log n)$.

9.5 A bound on the number of leaves

Lemma 9.3 *Let x be a node in a Fibonacci heap with* $\mathrm{Degree}[x] = d$. *Let $y_1, \ldots, y_\ell$ be the children of x in order in which they were attached to x. Then* $\mathrm{Degree}[y_1] \geq 0$ *and* $\mathrm{Degree}[y_i] \geq i - 2$ *for all* $i \geq 2$.

Proof. Let $i \geq 2$. When y_i was linked to x, then x had $y_1, \ldots, y_{i-1}$ as children and hence, $\mathrm{Degree}[x] \geq i - 1$. Two nodes are only linked together, if they have the same degree. Therefore, $\mathrm{Degree}[y_i] \geq i - 1$ at that time. Since y_i still is a child of x, it has lost at most one child so far. Therefore, $\mathrm{Degree}[y_i] \geq i - 2$. ∎

Let F be a Fibonacci heap. Let s_k denote the minimum size of any subtree of F having a root of degree k.

Lemma 9.4 *For all $k \geq 2$, $s_k \geq F_{k+2}$.*

Proof. The proof is by induction on k.
Induction base: We have $s_0 = 1$, $s_1 = 2$, and $s_2 = 3$.
Induction step: Let $k \geq 3$. By Lemma 9.3 and the induction hypothesis,

$$s_k \geq s_0 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}. \qquad \blacksquare$$

Exercise 9.1 *Prove that $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ for all k. (Hint: Induction on k)*

Corollary 9.5 *$d(n) \in \mathrm{O}(\log n)$ for all Fibonacci heaps with n nodes*

Proof. For every node of degree k, we have $F_{k+2} \leq n$. As we have seen before, $k \in \mathrm{O}(\log n)$. $\blacksquare$

10 Master theorem

We solved recurrence relations twice, when analyzing merge sort and the linear time selection algorithm. Everything we learned there is essentially sufficient to show the following general purpose theorem. It is usually called "master theorem" (but should be called "slave theorem" since it encourages you to turn your brain off).

Theorem 10.1 (Master theorem) *Let $a \geq 1$, $b > 1$, c, and n_0 be natural numbers, let $f : \mathbb{N} \to \mathbb{N}$ be a function and let t be defined by*

$$t(n) = \begin{cases} a \cdot t(\lceil n/b \rceil) + f(n) & \text{if } n > n_0 \\ c & \text{if } n \leq n_0. \end{cases}$$

Then the following holds:

1. *If $f(n) = \mathrm{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $t(n) = \mathrm{O}(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $t(n) = \mathrm{O}(n^{\log_b a} \log n)$.*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and a $f(\lceil n/b \rceil) \leq df(n)$ for some constant $d < 1$ and all sufficiently large n, then $t(n) = \mathrm{O}(f(n))$.*

Proof overview: Consider the recursion tree. At the root, we have work $f(n)$ to do. The root has a children. At every children, we have work roughly $f(n/b)$, given a total work of $af(n/b)$. In general, at recursion depth i, the total work amounts to $a^i f(n/b^i)$.

The function $f(n) = n^{\log_b a}$ fulfills $n^{\log_b a} = a(n/b)^{\log_b a}$. For this function, the total work at every recursion level is the same, namely $n^{\log_b a}$. There are $\log_b n$ levels yielding a total amount of $n^{\log_b a} \log_b n$.

If f grows (polynomially) slower than $n^{\log_b a}$, then the work at the leaves dominates, since the work increases geometrically. There are $a^{\log_b n} = n^{\log_b a}$ leaves. The work at each leaf is constant.

If f grows (polynomially) faster than $n^{\log_b a}$, then the work at the root dominates. It is $f(n)$.

Exercise 10.1 *Show that if f fulfills $f(\lceil n/b \rceil) \leq df(n)$ for some constant $d < 1$ and all sufficiently large n, then $f(n) = \Omega(n^{\log_b a + \epsilon})$*

Lemma 10.2 *Let $f, f' : \mathbb{N} \to \mathbb{N}$ and let t and t' be the solutions of the recurrence in the master theorem for f and f', respectively.*

1. *Assume that $f(n) \leq f'(n)$ for all n. Then $t(n) \leq t'(n)$ for all n.*
2. *If f is monotone, so is t.*

Exercise 10.2 *Proof the lemma above (Hint: standard induction on n).*

Let $B = \{b^i \mid i \in \mathbb{N}\}$ be the set of all powers of b. If $n \in B$, then $\lceil n/b \rceil = n/b \in B$, too. Therefore, the restriction of $t|_B$ to B is also defined by the recurrence of the master theorem.

For the first two cases of the master theorem, our strategy will be to solve the recurrence only for $t|_B$ and $f|_B$. The next lemma shows that this solutions also bounds t on $\mathbb{N}$.

Lemma 10.3 *If t is monotone and $t(n) \leq \gamma \cdot n^e$ for all $n \in B$ and constants γ and e, then $t \in \mathrm{O}(n^e)$.*

Proof. Let $n \in \mathbb{N}$. Let $i = \lceil \log_b n \rceil$. Since t is monotone, $t(n) \leq t(b^i)$. Therefore,

$$t(n) \leq \gamma b^{e \cdot i} \leq \gamma b^{e \log_b n + e} = \gamma b^e \cdot n^e.$$

Since γb^e is a constant, we are done. ∎

Proof of Theorem 10.1. We start with the first two cases. Let $e := \log_b a$, respectively. By Lemmas 10.2 and 10.3, it is sufficient if we show that $t|_B(n) \leq \gamma \cdot n^e$ (first case) and $t|_B(n) \leq \gamma \cdot n^e \log n$.

By dividing by $\gamma \cdot n^e$, the recurrence relation becomes

$$\frac{t(n)}{\gamma \cdot n^e} \leq a \cdot \frac{t(n/b)}{\gamma \cdot n^e} + \frac{f(n)}{\gamma \cdot n^e}$$

Let $\hat{t}(n) = \frac{t(n)}{\gamma \cdot n^e}$.

We have

$$\gamma n^e = \gamma b^e \left(\frac{n}{b}\right)^e = a \left(\frac{n}{b}\right)^e$$

In the first case, the recurrence becomes

$$\hat{t}(n) \leq \hat{t}(n/b) + \alpha \cdot n^{-\epsilon}$$

for some constant α. By Exercise 10.3, $\hat{t}(n) = \mathrm{O}(1)$. Therefore, $t(n) = \mathrm{O}(n^{\log_b a})$.

In the second case, the recurrence becomes

$$\hat{t}(n) \leq \hat{t}(n/b) + 1$$

Using a proof similar to the one of Lemma 1.13, we get $\hat{t}(n) = \mathrm{O}(\log n)$. Hence, $t(n) = \mathrm{O}(n^{\log_b a} \log n)$.

In the third case, we do not restrict ourselves to B. We divide the recurrence by $f(n)$ and get

$$\frac{t(n)}{f(n)} \leq a \cdot \frac{t(\lceil n/b \rceil)}{f(n)} + 1$$

Let $\hat{t}(n) = \frac{t(n)}{f(n)}$. By assumption, $f(n) \leq a \cdot f(\lceil n/b \rceil)/d$. Thus, the recurrence becomes

$$\hat{t}(n) = d \cdot \hat{t}(\lceil n/b \rceil) + 1.$$

As in the proof of Lemma 4.5, we get $\hat{t}(n) = \mathrm{O}(1)$. Thus $t(n) = \mathrm{O}(f(n))$. ∎

Exercise 10.3 *Show that if $s(n) = s(n/b) + n^{-\epsilon}$ for all $n \in B \setminus \{1\}$ and $s(1) = 1$, then $s(n) = \mathrm{O}(1)$. (Hint: Show by induction on n, that $s(n) = n^{\epsilon} \sum_{i=0}^{\log_b n - 1} b^{\epsilon i}$.)*

11 Some algorithm design techniques

In this chapter, we will learn two algorithm design techniques that are well-suited for optimization problems with a certain "substructure property".

11.1 Dynamic programming

Let $G = (V, E)$ be an undirected graph. Let $n = |V|$ and $m = |E|$. A *node coloring* is a mapping $c : V \to \mathbb{N}^+$. If $\operatorname{im} c \subseteq \{1, \ldots, k\}$, then c is called a k-coloring. A coloring c of a graph $G = (V, E)$ is called *proper* if $c(u) \neq c(v)$ for all $\{u, v\} \in E$. The chromatic number $\chi(G)$ of a graph G is the smallest number k such that there is a k-coloring of G.

Exercise 11.1 *Let $G = (V, E)$ be a graph with $|V| = n$. Prove the following:*

1. $1 \leq \chi(G) \leq n$
2. $\chi(G) = 1$ *if and only if* $E = \emptyset$
3. $\chi(G) = 2$ *if and only if G is bipartite.*
   (A graph is bipartite if we can partition $V = L \cup R$ with $L \cap R = \emptyset$ such that for every edge $\{u, v\} \in E$, $u \in L$ and $v \in R$ or vice versa.)

Given a graph $G = (V, E)$ and an integer k as an input, how can we decide whether G has a proper k-coloring? First, we can try all proper k-colorings. We can bound their number by k^n. The total running time of the naive algorithm is $\mathrm{O}(k^n \cdot m)$, since for each coloring, we have to check whether it is proper or not. Backtracking is an alternative, however, it is not clear how to bound the running time.

For a set $U \subseteq V$, the induced subgraph $G[U] = (U, E[U])$, where $E[U]$ is the set of all edges $\{u, v\} \in E$ with $u, v \in U$. We call a set $U \subseteq V$ *independent* if $G[U]$ has no edges. If $U = \emptyset$, then $G[U]$ is the empty graph. Its chromatic number is 0 by definition.

Dynamic programming computes the chromatic numbers of induced subgraphs of G, starting with the subgraphs of size one and then increasing the size of the subgraphs step by step. We set

$$\begin{aligned} C(G[\emptyset]) &= 0 \\ C(G[\{v\}]) &= 1 \qquad \text{for all } v \in V \end{aligned}$$

For any set U we define

$$C(G[U]) = \min\{C(G[U \setminus I]) + 1 \mid \emptyset \subsetneq I \subseteq U \text{ is an independent set}\} \quad (11.1)$$

These two equations immediately yield an algorithm: Run over all sets U, starting with the sets of size 0 and 1, then of size 2, and so on. For each set U with $|U| > 1$ run over all subsets I and check whether it is independent. If yes, then check whether $C(G[U \setminus I]) + 1$ is smaller than the smallest number of colors we found so far.

Lemma 11.1 *For all U, $\chi(G[U]) = C(G[U])$.*

Proof. The proof is by induction on $|U|$.
Induction base: If $|U| = 0$ or $|U| = 1$, then this is obvious.
Induction step: Let $|U| > 1$. Let c be an optimum proper coloring for $G[U]$, that is, a coloring with $\chi(G[U]) =: k$ colors. Every color class forms an independent set, in particular $c^{-1}(k) =: I$ is independent. c restricted to $U \setminus I$ is a $(k-1)$-coloring and therefore, $\chi(G[U \setminus I]) \leq \chi(G[U]) - 1$. By the induction hypothesis, $C(G[U \setminus I]) = \chi(G[U \setminus I])$. We have

$$C(G[U]) \leq C(G[U \setminus I]) + 1 \leq \chi(G[U])$$

Since there is also a coloring with $C(G[U])$ colors by construction, $C(G[U]) = \chi(G[U])$. ∎

The running time is proportional to

$$\sum_{i=0}^{n} \binom{n}{i} 2^i \cdot m = m \cdot 3^n,$$

which is independent of k![1] $\binom{n}{i}$ is the number of sets U of size i and 2^i is the number of subsets I of a set U of size i. (We have included the empty set to get a nicer formula for the running time.)

11.1.1 Finding the sequence of operations

The algorithm above only computes the number of colors needed. What can we do if we want to compute a corresponding coloring? $C(G[U])$ is computed by looking at all $C(G[U \setminus I])$. If we store a set I_U for which the minimum is attained for every $C(G[U])$, then we can easily construct a coloring. We start at $C(G[V])$. We give the set I_V color 1. Then we look at $C(G[V \setminus I_V])$. The set stored there gets color 2 and so on.

[1] There are even faster algorithms, for instance, one can show that there are at most $3^{n/3}$ independent sets in any graph.

11.1.2 When does dynamic programming help?

Dynamic programming helps whenever we can divide our problems (here computing $C(G[U])$) into subproblems (here $C(G[U \setminus I])$) From the optimal solutions of the subproblems, we must be able to reconstruct an optimal solution of our original problem.

In order to be efficient, we also need that the subproblems "overlap". In this way, the overall work is dramatically reduced.

The term "programming" in "dynamic programming" does not mean designing a program or writing some code, it means solving a problem by using a tableau.

11.1.3 Memoization

The dynamic programming approach is bottom-up. If we want to go top-down by making recursive calls, we can use a technique called memoization. We have a global array $C(G[U])$. Whenever we compute $C(G[U])$ during a recursive call, we store the result in $C(G[U])$. Before making a recursive call, we look in $C(G[U])$ whether this value has already been computed. This avoids unnecessary recursive calls. This essentially fills the table T top-down instead of bottom-up.

11.2 Greedy algorithms

11.2.1 Interval scheduling and interval graph coloring

Consider the following problem: We have a set of intervals $I_i = [s_i, f_i)$, $1 \leq i \leq n$. In the interval scheduling problem, we want to find as many intervals as possible that are pairwise disjoint. You can think of each I_i being a lecture with starting time s_i and finishing time f_i. You want to attend as many lectures as possible. [2]

A related problem is the following: Instead of finding a maximum set of non-intersecting lectures, we want to assign the lectures to lecture halls in such a way that at most one lecture takes place in a lecture hall at any point in time. Of course, we want to use the least number of lecture halls. The *interval graph coloring problem* formalizes this. Given a set V of intervals $[s_i, f_i)$, $1 \leq i \leq n$, the corresponding interval graph $G = (V, E)$ is obtained as follows: There is an edge between two intervals I and J if and only if $I \cap J \neq \emptyset$. (This means that the corresponding activities cannot take place in the same location, since their time slots intersect.) A proper coloring of an interval graphs is a solution to our scheduling problem: Each color corresponds to a

[2] The intervals are closed on the one side and are open on the other to reflect the fact that a lecture that goes from 2 to 4 and a lecture from 4 to 6 are usually assumed to be compatible.

lecture hall. All lectures with the same color can be scheduled to the same location, since by the definition of proper coloring, they are not connected by an edge and hence their time slots do not intersect. To minimize the number of locations, we have to seek for a proper coloring with a minimum number of locations.

11.2.2 The greedy method

The interval scheduling and graph coloring problem can be solved by a so-called *greedy algorithm*. Greedy algorithms are greedy in a literal sense, they always make the choice that looks best to them at the moment. In the interval scheduling problem, we will always take the interval with the smallest finishing time that does not intersect with the intervals already chosen, that is, occupies the least amount of time.

Algorithm 45 Greedy-scheduling

Input: a set of intervals with starting and finishing times $s_i \leq f_i$, $1 \leq i \leq n$
Output: a subset T of non-intersecting intervals that is as large as possible

1: $T := \emptyset$
2: **while** there is an interval that does not intersect any interval in T **do**
3: Choose the non-intersecting interval I that has the smallest finishing time.
4: $T := T \cup \{I\}$
5: return T

If we sort S in advance with respect to finishing times, then we only look at each activity once and we can achieve a running time of $O(n \log n)$ (including sorting).

The next lemma shows that the greedy approach here works.

Lemma 11.2 *Let I_{j_i} be the interval that is selected in the ith iteration of the while loop, $1 \leq i \leq t$. If there is a set $S_i = \{j_1, \ldots, j_i, h_1, \ldots, h_\ell\}$ such that $I_p \cap I_q = \emptyset$ of all $p, q \in S_i$, $p \neq q$, then after a suitable permutation of the $h_1, \ldots, h_\ell$, $\hat{S}_i = \{j_1, \ldots, j_i, j_{i+1}, h_2, \ldots, h_\ell\}$ is also a set of indices such that $I_p \cap I_q = \emptyset$ for all $p, q \in \hat{S}_i$, $p \neq q$.*

Proof. Since $I_{j_{i+1}}$ was chosen by Greedy-schedule in the $(i+1)$th iteration, we have $f_{j_{i+1}} \leq f_{h_s}$, $1 \leq s \leq \ell$. Each I_{h_s} does not intersect with any I_{j_r}, $1 \leq r \leq i$. Hence if $f_{j_{i+1}} > f_{h_s}$, then Greedy-schedule would have chosen I_{h_s} instead. $I_{j_{i+1}}$ can have a nonempty intersection with at most one I_{h_s}. Otherwise two of the $I_{h_1}, \ldots, I_{h_\ell}$ would intersect, which cannot be true by assumption. Let w.l.o.g. I_{h_1} be this interval. Since $I_{j_{i+1}}$ does not intersect with any of the other intervals, also not with I_{j_r}, since $I_{j_{i+1}}$ was chosen by Greedy-schedule. Therefore, $\hat{S}_i$ is a set of indices such that the corresponding intervals do not intersect. ∎

Theorem 11.3 *Greedy schedule is correct, that is, if there is a schedule S with k intervals, then Greedy-schedule will find a schedule with at least k intervals.*

Proof. Let $j_1, \ldots, j_m$ be the intervals chosen by Greedy-schedule and let $S = \{q_1, \ldots, q_k\}$. By applying Lemma 11.2 repeatedly and swapping in j_1, j_2, and so on, we get that $S' = \{j_1, \ldots, j_m, q_{k-m+1}, \ldots, q_k\}$ is a schedule with k intervals, too. Can $m < k$? No, because S' is a schedule and because $f_{j_i} \leq f_{q_{k-m+1}}$ by the greedy choice, greedy would also have taken $I_{q_{k-m+1}}$, which it has not. Therefore, $m = k$. ∎

Next we come to the interval graph coloring problem. For the ease of presentation, we assume that the starting and the finishing times are natural numbers between 0 and N. Let

$$C := \max_{0 \leq x \leq N} \#\{i \mid x \in I_i\}.$$

C is the maximal number of intervals that overlap in a single point. All these intervals are connected with an edge in the interval graph, therefore, each of these intervals needs a different color. Therefore, every proper coloring of an interval graph needs at least C colors.

On the other hand, there is always a coloring with C colors and we can find it by a greedy algorithm. We consider the intervals by ascending starting time and always give the next interval a color that does not create a conflict.

Algorithm 46 Greedy-coloring

Input: an interval graph $G = V, E$
Output: a proper coloring of the nodes in V

1: sort the intervals by starting time.
2: initially, all intervals are uncolored
3: **for** $j = 1, \ldots n$ **do**
4: Let F be the colors of intervals from $I_1, \ldots, I_{j-1}$ that intersect I_j.
5: Choose a color from $\{1, \ldots, C\} \setminus F$ and color I_j with this color.

Since we sort the intervals by starting time, it is also very easy to keep track of the intervals that intersect with I_j. In this way, we get an algorithm with $O(n \log n)$ running time.

The algorithm will never color intervals that are connected by an edge with the same color. It might happen that an interval remains uncolored, since there are no colors left in $\{1, \ldots, C\} \setminus F$. Can this happen? Since the intervals are sorted by starting time, all the intervals that intersect I_j intersect in the point s_j. By the definition of C, $\#F < C$. Therefore, I_j is properly colored.

Theorem 11.4 *Greedy-coloring is correct.*

11.2.3 When does the greedy approach work?

Greedy algorithms are quite similar to dynamic programming. We reduce our problem to the solution of a subproblem. But in dynamic programming, we reduce the solution of our problem to the solution of several problems and we do not know which of them will give the best solution to our original problem. So we have to solve them all. If greedy algorithms works, it can reduce the solution to the original problem to the solution of *one* problem. So usually it runs much faster, typically in linear time.

12 Graph algorithms

A(n undirected) *graph* G is a pair (V, E), where V is a finite set, the set of *nodes* or *vertices*, and E is a set of two-element subsets (= unordered pairs) of V, the so-called *edges*. If $e = \{u, v\} \in E$, we say that u and v are connected by the edge e or that u and v are *adjacent*. The edge e is called *incident on* u and v.

In a *directed* graph $G = (V, E)$, edges are ordered pairs, i.e., $E \subseteq V \times V$. We say that an edge $e = (u, v)$ *leaves* u and *enters* v or is incident from u and incident to v. e is called an edge from u to v. An edge of the form (v, v) is called a self-loop. We can also allow self loops in undirected graphs, they are sets of size one. Note that in a directed graph, there can be two edges between two nodes, one from u to v and the other one from v to u.

A sequence $v_0, v_1, \ldots, v_k$ is called a *walk* from v_0 to v_k in $G = (V, E)$ if $\{v_i, v_{i+1}\} \in E$ for all $0 \leq i < k$. (In the case of a directed graph, $(v_i, v_{i+1}) \in E$, that is, all edges in the path have to point "in the same direction".) A walk is called a (simple) *path* if all nodes in the path are distinct. k is called the *length* of the path. (Some books use path as a synonym for walk and simple path for path.)

Exercise 12.1 *Show that if two vertices are connected by a walk, then they are connected by a path.*

A *cycle* is a walk such that $v_0 = v_k$, $k > 0$ (if G is directed) or $k > 1$ (if G is undirected), and the vertices $v_0, \ldots, v_{k-1}$ are pairwise distinct. (Some books call this a simple cycle.) A graph is called *acyclic* if it contains no cycle.

A node is u is reachable from v if there is a path from v to u. If G is undirected, then the "is reachable from" relation is symmetric.

Exercise 12.2 *Show that in undirected graph, the "is reachable from" relation is an equivalence relation.*

The *connected components* of an undirected graph are the equivalence classes of the "is reachable from" relation. Two nodes u and v are in the same component if and only if there is a path between u and v. A graph is called *connected* if there is only one equivalence class.

If G is directed, then the "is reachable from" relation is no longer an equivalence relation. But the relation "u is reachable from v and v is reachable from u" is an equivalence relation. (Prove this!) The equivalence classes

of this relation are the *strongly connected components*. A directed graph is *strongly connected* if there is only one equivalence class. It is called *weakly connected* if the underlying undirected graph (the graph that we get by replacing every directed edge (u, v) by $\{u, v\}$ and possibly removing double edges) is connected.

A *tree* is an acyclic connected undirected graph. A *forest* is an acyclic undirected graph. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A tree that is a subgraph of G and that contains all nodes of G is called a *spanning tree*.

12.1 Data structures for representing graphs

How can we store a graph? It is very convenient to represent the nodes in V as numbers $1, 2, \ldots, n$ where $n = |V|$. There are two standard ways of storing the edges of a graph, the *adjacency-list* representation and the *adjacency-matrix* representation.

In the adjacency-list representation, for every node i, there is a list L_i. If there is an edge $e = \{i, j\}$ or $e = (i, j)$, we add the element j to L_i. In other words, if there is an edge from i to j, then we store the vertex j in the list L_i. In the case of an undirected graph, we either store the edge $\{i, j\}$ twice, namely in L_i and L_j, or we make the convention that an edge is only stored in the list L_i, if $i < j$, and in L_j, if $j < i$, respectively.

In the adjacency-matrix representation, we store the edges in a matrix (or two-dimensional array) A:

$$A[i, j] = \begin{cases} 1 & \text{if } \{i, j\} \text{ or } (i, j) \in E, \text{ respectively} \\ 0 & \text{otherwise} \end{cases}$$

If G is undirected, then A is symmetric.

First of all, the adjacency-list representation needs $O(|V| + |E|)$ cells of memory, so the space consumption is asymptotically optimal. (Note that $|E|$ can be much smaller than $|V|$ if the graph is not connected. Therefore, we cannot just write $O(|E|)$.) The adjacency-matrix representation always needs $O(|V|^2)$ cells of memory. If the graph has relatively few edges (so called sparse graphs), then the adjacency-list representation is preferably with respect to space consumption.

If we want to find out whether there is an edge between two nodes (i, j), we can answer this question in $O(1)$ time with an adjacency-matrix representation. With an adjacency-matrix representation, however, we have to scan the whole list L_i (or L_j) in the worst case, so the time is $O(d_i)$ where d_i is the *degree* of i, that is, the number of edges that are incident with i.

In short, if one can afford it, the adjacency-matrix representation is advantageous because of its simplicity. However, often we are confronted

with huge sparse graphs (think of web pages as nodes and links as edges). Here adjacency-matrices are way too large.

Both representation can be easily extended to weighted graphs. A weighted graph $G = (V, E, w)$ has in addition a function $w : E \to \mathbb{Q}$ that assigns each edge a weight. In the adjacency-list representation, we simply store the corresponding weight $w(i, j)$ together with j in the list. In the adjacency-matrix, we just store the weight $w(i, j)$ instead of 1. If an edge does not exist, we can, depending on the application, either store 0 or ∞ or a special value indicating that there is no edge.

12.2 Breadth first search

Breadth first search and depth first search are two important way to explore a graph and they are the basis of many other graph algorithms. We are given a graph $G = (V, E)$ and a node $s \in V$, the start node or source. We want to explore the graph in the sense that we want to discover all nodes that are reachable from s by a path.

We start in s. The nodes that we first can discover from s are the nodes v such that there is an edge $(s, v) \in E$, the so-called neighbors of s.[1] Then we can discover the neighbors of the neighbors of s and so on until all nodes reachable from s are discovered.

Breadth first search first explores *all* nodes that are close to s. Nodes have one of three possible states: undiscovered, discovered, finished. In the beginning, s is discovered and all other nodes are undiscovered. A node becomes discovered if we encounter it for the first time. Then its state changes to discovered. If all neighbors of a node have been discovered, its state finally changes to finished.

Breadth first search first marks every node v as *undiscovered*, the distance $d[v]$ to s is set to ∞, and the predecessor $p[v]$ is set to NULL. Since the only node that we know so far is s, we set $state[s] = discovered$ and $d[s] := 0$. We initialize a queue Q with s, Q will contain all discovered nodes that are not finished. As long as Q is not empty, we remove the next node v from it. All its neighbors that have not been discovered before are marked *discovered* and are put into Q. v becomes the predecessor of these nodes and their distance is set to $d[v] + 1$. Since all of its neighbors are discovered, v is marked *finished*.

[1] Breadth first search works for undirected and directed graphs. We only present it for directed graphs, since every undirected graph can be represented by a directed graph by replacing each undirected edge $\{u, v\}$ be two directed edges (u, v) and (v, u).

Algorithm 47 Breadth first search

Input: graph $G = (V, E)$, source $s \in V$
Output: every node that is reachable from s is marked as *finished*
$p[v]$ contains the node from which v was discovered
$d[v]$ contains the distance from s to v

```
 1: for each v ∈ V do
 2:    state[v] := undiscovered
 3:    p[v] := NULL
 4:    d[v] := ∞
 5: state[s] := discovered
 6: d[s] := 0
 7: let Q be an empty queue
 8: Enqueue(Q, s)
 9: while Q is not empty do
10:    v := Dequeue(Q)
11:    for each node u adjacent to v do
12:       if state[u] = undiscovered then
13:          state[u] := discovered
14:          d[u] := d[v] + 1
15:          p[u] := v
16:          Enqueue(Q, u)
17:    state[v] := finished
```

12.2.1 Correctness

Let $G = (V, E)$ be a graph. The shortest path distance $\delta(x, y)$ of two nodes $x, y \in V$ is

$$\delta(x, y) = \text{the length of a shortest path from } x \text{ to } y.$$

If there is no path from x to y, then $\delta(x, y) = \infty$.

Exercise 12.3 *Show that δ fulfills the triangle inequality, i.e,*

$$\delta(x, z) \leq \delta(x, y) + \delta(y, z)$$

for all $x, y, z \in V$.

Lemma 12.1 *Let $G = (V, E)$ be a graph and $s \in V$. Then*

$$\delta(s, y) \leq \delta(s, x) + 1$$

for all $(x, y) \in E$.

Proof. If there is a path p from s to x, then there is also one from s to y, namely, p extended by y. Therefore $\delta(s, y) \leq \delta(s, x) + 1$. If there is no path from s to x, then $\delta(s, x) = \infty$ and the inequality is trivially fulfilled. ∎

Let $U_i = \{v \in V \mid \delta(s, v) = i\}$ for $i \in \mathbb{N}$ be the set of all vertices that have distance i from s. We have $U_0 = \{s\}$.

Lemma 12.2 *If BFS is executed on $G = (V, E)$, then all vertices of U_i are inserted into the queue before all vertices in U_{i+1} and after all vertices in U_{i-1}, if $i > 0$.*

Proof. The proof is by induction on i.
Induction base: Clear, since $U_0 = \{s\}$ and s is the first vertex in the queue and every vertex enters the queue at most once.
Induction step: For the induction step, we assume that the statement is true for all $0 \leq j \leq i$. By the induction hypothesis (applied repeatedly), first all vertices of U_0 enter the queue, then all of U_1 and finally all of U_i.

Now new nodes are put into the queue if a node is removed from the queue. If a node v is removed from the queue, all its undiscovered neighbors are put into the queue. If v is in U_j, then all its neighbors are in $U_0 \cup U_1 \cup \cdots \cup U_{j+1}$. Neighbors in $U_0 \cup \cdots \cup U_{j-1}$ were put before v in the queue, therefore they are already finished and not put into the queue. Neighbors in U_j are also neighbors of some node $u \in U_{j-1}$. Hence they were already put into the queue by u or some other node in U_{j-1}, since all nodes in U_{j-1} were removed from the queue before v. Therefore, the only nodes that can be put into the queue by v are its neighbors in U_{j+1}.

Now we apply the argument above for $j = i$. Nodes from U_{i+1} can only be put into the queue when we remove vertices from U_i. When we remove the first vertex of U_i, only vertices from U_i are in the queue and we start putting the vertices from U_{i+1} into the queue. Since every vertex in U_{i+1} has a neighbor in U_i, all vertices of U_{i+1} are in the queue after the last vertex of U_i is removed. ∎

Theorem 12.3 *BFS works correctly, that is, after termination, $d[v] = \delta(s, v)$ for all $v \in V$. Moreover, for every node v with $\delta(s, v) < \infty$, a shortest path from s to $p[v]$ together with the edge $(p[v], v)$ is a shortest path from s to v.*

Note that $d[v] = \delta(s, v)$ is sufficient to prove the correctness, since a node v is marked as finished if and only if $d[v] < \infty$ and if a node is marked as finished, then $p[v]$ contains the node v was discovered from.

Proof. If v is not reachable from s, then v is never put into the queue, hence $d[v] = \infty$.

We now show by induction on i that $d[v] = i$ for all $v \in U_i$.
Induction base: This is true by construction for U_0.
Induction step: Assume that $d[v] = i$ for all $v \in U_i$. All nodes u in U_{i+1} are put into the queue by a node in U_i by Lemma 12.2. This means that $d[u] = i + 1$. This completes the induction step.

Finally note that for every node $v \in U_{i+1}$, $p[v] \in U_i$. Therefore a shortest path to $p[v]$ has length i. If we extend this path by $(p[v], v)$, then we get a path of length $i + 1$, which must be a shortest path. ∎

Corollary 12.4 *Let $V' = \{v \in V \mid p[v] \neq \text{NULL}\} \cup \{s\}$ and $E' = \{(p[v], v) \mid v \in V' \setminus \{s\}\}$. Then $G' = (V', E')$ is a tree such that every path from s to v is a shortest path.*

The proof of the corollary follows from repeatedly using the "moreover"-part of Theorem 12.3. Such a tree G' is called a *shortest path tree*.

12.2.2 Running time analysis

The initialization phase (lines 1–6) is linear in the number of nodes. Since we only put undiscovered nodes into the Q and mark them as *finished* when we remove them, every node is put at most once into the queue. Since every node is queued only once, we look at every edge also at most once. If we have a adjacency-list representation, then the running time of breadth first search is $O(|V| + |E|)$. In other words, the running time is linear in the size of the graph. If we have an adjacency-list representation, then the running time is $O(|V|^2)$.

12.3 Depth first search

Depth first search also explores all nodes that are reachable from a given node s. But instead of storing the discovered node in a queue, it uses a stack (or recursion). Everything else is the same, our vertices can have one of three states *undiscovered*, *discovered*, and *finished*. And for every node v, we store the node $p[v]$ from which it was discovered.

Since nodes will not be discovered along shortest paths, we will not store the distances to s. Instead, there is a global variable t which is used as a clock. It is increased by one whenever a new vertex is discovered or one is finished. Every vertex gets two time stamps, the discovery time $d[v]$ and the finishing time $f[v]$.

Algorithm 48 Depth-first-search

Input: graph $G = (V, E)$, node x
uses a global variable t as clock
Output: marks all nodes reachable from x as *finished*
stores for each such node the discovery time $d[v]$ and finishing time $f[v]$

```
1: state[x] := discovered;
2: t := t + 1; d[x] := t
3: for each y adjacent to x do
4:    if state[y] = undiscovered then
5:       p[y] = x
6:       Depth-first-search(y)
7: state[x] := finished
8: t := t + 1; f[x] := t;
```

Before we call Depth-first-search(G, s), we have to initialize the arrays *state* and p. Since we look at every vertex and every edge at most once, the running time is $O(|V| + |E|)$.

To explore all nodes in a graph, for example, when we want to compute connected components, we can pack the depth-first-search procedure into a for loop. The procedure DFS-explore does not only explore what is reachable from a single node, but when it explored all reachable nodes, it starts with another undiscovered vertex. (We could have done the same with BFS.) In this way, we do not only get a depth first search tree but a depth first search forest. The trees in these forests are essentially the recursion trees.

Theorem 12.5 *After execution of DFS-explore, for any two vertices $x \neq y$, the intervals $[d[x], f[x]]$ and $[d[y], f[y]]$ are either disjoint or one is contained in the other.*

Proof. W.l.o.g. assume that $d[x] < d[y]$. If $f[x] < d[y]$ then the intervals are disjoint and we are done.

Algorithm 49 DFS-explore

Input: a graph G
Output: every node is visited
start and finishing times are computed

```
1: for each v ∈ V do
2:    state[v] := undiscovered
3:    p[v] := NULL
4: t := 0
5: for each x ∈ V do
6:    if state[x] = undiscovered then
7:       Depth-first-search(x)
```

If $d[y] < f[x]$, then x's state is *discovered* when y is discovered. Therefore x is a predecessor in the recursion tree of y. Therefore $f[x] < f[y]$. ■

13 Minimum spanning trees

Let $G = (V, E, w)$ be a connected weighted undirected graph. A tree T that is a subgraph of G and contains every node of G is called a *spanning tree*. Let E_T be the edges of T. The weight of T is defined as

$$w(T) = \sum_{e \in E_T} w(e).$$

A spanning tree is a *minimum spanning tree* of G if its weight is minimal among all spanning trees of G.

If we think of the vertices as towns and the cost on the edges as the costs of building a road between the corresponding towns, then a minimum spanning tree is the cheapest way of building streets such that any pair of towns is connected. (We do not allow any junctions outside the towns. If this is allowed, then we get so-called Steiner trees. But this is a more difficult problem.)

A *cut* of G is a partition of the vertices into two nonempty sets $(S, V \setminus S)$. An edge $e = \{u, v\}$ *crosses* the cut, if $u \in S$ and $v \in V \setminus S$ or vice versa. A cut *respects* a set E' of edges if no edge in E' crosses the cut.

The following theorem will be useful, when analyzing algorithms for minimum spanning trees.

Theorem 13.1 *Let F be a set of edges that is contained in a minimum spanning tree of $G = (V, E, w)$. Let $(S, V \setminus S)$ be a cut that respects F. Let e be an edge of minimum weight crossing the cut. Then $F \cup \{e\}$ is also contained in a minimum spanning tree.*

Proof. Let $T = (V, E_T)$ be a minimum spanning tree that contains F. Let $e = \{u, v\}$ an edge of minimum weight crossing the cut $(S, V \setminus S)$. If T contains e, too, then we are done.

Otherwise, the graph $(V, E_T \cup \{e\})$ contains exactly one cycle consisting of the edge e and a path P from u to v in T. Since u and v are on opposite sides of the cut, there is a least one edge $f = \{x, y\}$ in P crossing the cut.

Because $(V, E_T \cup \{e\})$ contains only one cycle, $T' = (V, E_T \cup \{e\} \setminus \{f\})$ is again acyclic. It is still connected, since the nodes x and y are connected by the remaining edges of P together with e. Therefore, T' is a spanning tree. By the choice of e, $w(e) \leq w(f)$. Thus

$$w(T') = w(T) - w(f) + w(e) \leq w(T).$$

Since T is a minimum spanning tree, $w(T') = w(T)$. Because F respects the cut, $f \notin F$. By construction, T' contains $F \cup \{e\}$. Hence, T' is a minimum spanning tree as required. ■

13.1 Data structures for (disjoint) sets

For one of the algorithms that we will design in this chapter, we need a data structure to maintain so-called disjoint dynamic sets. We have a finite universe U of n elements and we want to store a collection of disjoint subsets $S_1, \ldots, S_\ell \subseteq U$. Each of the sets S_i is represented by a distinguished element in it, the so-called representative. Since the sets are disjoint, the representative is unique. The following operations shall be supported:

Make-set(x)**:** Creates a new set that contains x (and no other element). x is not allowed to appear in any other set that we created so far.

Union(x, y)**:** Computes the union of the two sets that contain x and y, respectively. We assume that x and y are in different sets. The old sets are destroyed.

Find(x)**:** Finds the representative of the unique set that x belongs to. If x is not in any set, then NULL is returned.

One easy implementation is to use linked lists. Each set is stored in a linked list, the first element in the list is the representative. Make-set can be implemented with constant running time and also Union needs only constant running time since we just have to append the lists. (We store an additional pointer to the last element.) However, finding the representative needs $\Theta(n)$ time in the worst case since we have to go from element to element until we reach the head of the list. To speed up the Find procedure, we can add to each element of the list a pointer to the first element. Then we can find the representative in time $O(1)$. But now the union needs time $\Theta(n)$ in the worst case, since we might have to redirect $\Theta(n)$ pointers.

A very good compromise are binomial heaps. Recall that binomial heaps are a lists of binomial trees. Make-set (called Make-BH) and Union are already implemented and have running times $O(1)$ and $O(\log n)$. As the representative element, we can take the minimum of the heap, which can be found in time $O(\log n)$.

13.2 Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm. In each step it chooses the lightest edge that does not create a cycle. We start with n trees, each consisting of a

Algorithm 50 Kruskal-MST

Input: connected weighted undirected graph $G = (V, E, w)$
Output: a minimum spanning tree T of G

```
1: Sort the edges by increasing weight
2: for each vertex v ∈ V do
3:    Make-set(v)
4: E_T := ∅
5: for each edge e = {u, v} ∈ E, in order by increasing weight do
6:    if Find(u) ≠ Find(v) then
7:       E_T := E_T ∪ {e}
8:       Union(u, v)
9: return (V, E_T)
```

single vertex. Then trees are joined until in the end, we have a single tree. We use a disjoint set data structure to store the intermediate forests.

Theorem 13.2 *Kruskal's algorithm returns a minimum spanning tree.*

Proof. We prove by induction on the number of edges that E_T is a subset of some minimum spanning tree. Since in the end, (V, E_T) is connected, we obtained a minimum spanning tree.
Induction base: In the beginning, E_T is empty.
Induction step: Now assume that $|E_T| > 0$. By the induction hypothesis, E_T is contained in some minimum spanning tree. If $|E_T| = n - 1$, we are done. Otherwise, let $e = \{u, v\}$ be the edge that is chosen next. Let S be the set of all nodes in the set of u. The set S respects E_T by construction and e crosses the cut $(S, V \setminus S)$. By Theorem 13.1, there is a spanning tree that contains $E_T \cup \{e\}$. ∎

Sorting the edges takes time $O(|E| \log |E|)$. Initialization of the forest takes time $O(|V|)$. In the second for-loop, we iterate over all edges. For each edge, we perform a constant number of disjoint set operations, each taking time $O(\log |V|)$. Therefore, the total running time is $O(|E| \log |E|)$. (Note that $|E| \geq |V| - 1$, since G is connected.)

13.3 Prim's algorithm

Prim's algorithm also works in a greedy fashion, too. But it does not grow several components, it just extends one component by successively adding new nodes. (Prim's algorithm was invented, as the name suggests, by R.C. Prim. But it was also invented earlier by V. Jarnik.)

We store all vertices in a priority queue. As long as a vertex y is not adjacent to a vertex of the spanning tree selected so far, its key is set to ∞.

Algorithm 51 Prim-MST

Input: connected weighted undirected graph $G = (V, E, w)$
Output: a minimum spanning tree T of G

```
1: Choose an arbitrary root r ∈ V.
2: Key[r] := 0; Key[v] := ∞ for all v ∈ V \ {r}.
3: Let Q be a min-priority queue filled with the vertices in V.
4: p[r] := NULL
5: while Q is not empty do
6:    x := Extract-min(Q)
7:    if p[x] ≠ NULL then
8:       E_T := E_T ∪ {p[x], x}
9:    for each vertex y adjacent to x do
10:      if y ∈ Q and w({x, y}) < Key[y] then
11:         p[y] := x
12:         Key[y] := w({x, y})
13: return (V, E_T)
```

If it becomes adjacent to some vertex x, then its key becomes the weight of $w(\{x, y\})$. If it becomes adjacent to a new vertex, then we only set the key to the new weight if this will decrease the key. (In particular, we can always use the Decrease-key procedure for this.) In this way, Key$[x]$ is the cost of adding x to the tree grown so far. In $p[y]$ we always store the vertex x such that Key$[y] = w(\{x, y\})$.

What do we achieve by this? Well, Prim's algorithm is supposed to work as follows: We always choose the minimum weight edge such that extends the current tree, that is, joins a new node to the tree. If $x :=$ Extract-min(Q), then the edge $\{p[x], x\}$ will be precisely this edge. By using a priority queue, we get an efficient implementation of this process. The only exception is in the beginning, when we extract the root r from the queue. In this case, we do not add an edge to the tree.

Why is Prim's algorithm correct? The edge chosen is always a minimum weight edge that crosses the cut consisting of the vertices chosen so far and the remaining vertices in the Q. Now, the same proof as the one of Theorem 13.2 works.

Assume that we implement the priority queue using binary heaps. The initialization needs time $O(|V|)$. The outer for-loop is executed $|V|$ times. An Extract-min operation needs time $O(\log |V|)$, so the cost of all Extract-min operations is $O(|V| \log |V|)$. The inner for-loop is executed $|E|$ times altogether. The test $y \in Q$ can be implemented in constant time by using a Boolean array. The Decrease-Key operation has running time $O(\log |V|)$. Thus the overall running time is $O(|V| \log |V| + |E| \log |V|) \subseteq O(|E| \log |V|)$. Note that $\log |V| \in \Theta(\log |E|)$, so this is not an improvement over Kruskal's algorithm (at least not theoretically). However, if we use Fibonacci heaps to

implement the queue, we can perform a Decrease-key operation in amortized time O(1). This means that all $|E|$ Decrease-key operations need running time O($|E|$) altogether. This gives a total running time of O($|V| \log |V| + |E|$).

14 Shortest paths

Let $G = (V, E, w)$ be an edge-weighted directed graph and let $s, t \in V$. Let $P = (v_0, \ldots, v_\ell)$ be a path from s to t, that is, $v_0 = s$, $v_\ell = t$, and for all $0 \leq \lambda < \ell$, $(v_\lambda, v_{\lambda+1}) \in E$. The *(shortest path) weight* of such a path is $w(P) = \sum_{\lambda=0}^{\ell-1} w((v_\lambda, v_{\lambda+1}))$. How can we compute the shortest[1] path from s to t? (This is an important problem if you develop a navigation system or a travel planer.)

First there is the question whether the shortest walk from s to t is always a path. This is certainly the case if all weights are non-negative. What if there are negative weights? Consider a walk from s to t that is not a (simple) path. Then this walk contains a cycle C. If the weight of C is non-negative, then we can remove this cycle and get a new walk whose weight is equal to (if $w(C) = 0$) or shorter than (if $w(C) > 0$) the weight of the original walk. If the weight of C is negative, then the walk gets shorter if we go through C once more. In this case, there is no shortest walk from s to t. In summary, if on every walk from s to t there is no negative cycle, then the weight $\delta(s, t)$ of shortest walk from s to t is well-defined and is always attained by a path (which has at most $|V| - 1$ edges). If there is a walk with a negative cycle from s to t, then we set $\delta(s, t) = -\infty$. Luckily, in many practical applications, non-negative weights do not appear at all.

Exercise 14.1 *Show that if $(v_0, v_1, \ldots, v_\ell)$ is a shortest path from v_0 to v_ℓ, then $(v_i, v_{i+1}, \ldots, v_j)$ is a shortest path from v_i to v_j.*

Exercise 14.2 *Show that for any three nodes u, v, and x, $\delta(u, v) \leq \delta(u, x) + w((x, v))$.*

14.1 Relaxing an edge

All algorithms that we know for computing a shortest path from s to t, also compute a shortest path from s to v for every other node $v \in V$. This seems wasteful at a first glance, but if you think about it, there could be a path with many edges which nevertheless has small weight. And to be sure that there is no such path you have to look at all nodes. (In practical applications, it is important to find stop-criteria and heuristics which edges to consider first. If you want to go from Saarbrücken to Homburg, you usually only have

[1] "shortest" means a path with minimum weight in this chapter and not minimum number of edges.

to consider edges to cities that are geographically near to Saarbrücken to Homburg.) Such a type of problem is called *single source shortest path*, we want to compute the shortest path from a given start node (source) to all other nodes. In the all-pair shortest path problem, we want to compute a shortest path between every pair of vertices.

Our algorithms will be quite similar to breadth-first-search. We have an array d and an array p. $d[v]$ will always be an upper bound for $\delta(s, v)$. In the beginning, we initialize $d[v] := \infty$ for $v \neq s$ and $d[s] := 0$. $p[v]$ will be a predecessor of v, in particular, we will always have

$$d[v] \leq d[p[v]] + w((p[v], v)). \tag{14.1}$$

provided that $p[v] \neq$ NULL. We initialize $p[v] :=$ NULL for all v.

The following procedure Relax[2] tests whether we can improve the current estimate $d[u]$:

Algorithm 52 Relax

Input: nodes u and v with $(u, v) \in E$
Output: $d[v]$ and $p[v]$ are updated
  if $d[v] > d[u] + w((u, v))$ **then**
    $d[v] := d[u] + w((u, v))$
    $p[v] := u$

If we initialize d and p as above and then perform any number of calls to Relax, then the following statements are true:

1. For every edge $(u, v) \in E$, $d[v] \leq d[u] + w((u, v))$ holds after every call of Relax(u, v).
2. For every $v \in V$, $\delta(s, v) \leq d[v]$.
3. The values $d[v]$ decrease monotonically (over time).
4. If $d[u] = \delta(s, u)$ and there is a shortest path from s to v such that the edge (u, v) is on this path, then $d[v] = \delta(s, v)$ after the call of Relax(u, v).

All the statements are true in the beginning. A call of Relax(u, v) only changes $d[v]$. When $d[v]$ is changed, then $d[v] \leq d[u] + w((u, v))$ now holds. When $d[v]$ is not changed, then $d[v] \leq d[u] + w((u, v))$ was true even before.

For the second statement, assume that the call of Relax(u, v) is the first call such that $d[v] < \delta(s, v)$ for some node v. Then

$$d[u] + w((u, v)) = d[v] < \delta(s, v) \leq \delta(s, u) + w((u, v)).$$

[2]The term Relax is used mostly for historical reasons. We rather tighten the upper bound.

The last inequality follows from Exercise 14.2. Therefore, $d[u] < \delta(s,u)$ which contradicts the assumption that Relax(u,v) was the first call with $d[v] < \delta(s,v)$.

The third statement is clear by the property of Relax.

For the fourth statement, note that after executing Relax(u,v), then $d[v]$ is $\delta(s,u) + w((u,v))$ which is the weight of a shortest path from s to v.

14.2 Dijkstra's algorithm

Dijkstra's algorithm assumes that all weights are non-negative. It is maybe the most popular single source shortest path algorithm.

Algorithm 53 Dijkstra

Input: edge-weighted directed graph $G = (V,E,w)$, $w(e) \geq 0$ for all e, source $s \in V$

Output: $d[v] = \delta(s,v)$ for all v and p encodes a shortest path tree

1: Initialize $d[s] := 0$, $d[v] := \infty$ for all $v \neq s$, and $p[v] :=$ NULL for all v.
2: Let Q be a min-priority queue filled with all vertices from V using $d[v]$ as keys.
3: **while** Q is not empty **do**
4: $\quad x :=$ Extract-min(Q)
5: $\quad$ **for** each y with $(x,y) \in E$ **do**
6: $\quad\quad$ Relax(x,y)

Note that the Relax procedure now involves a Decrease-min call. Before we come to the correctness, let us first analyze the running time. If we implement Q by an ordinary array, then the Insert and Decrease-key operations take time O(1) while Extract-min takes O($|V|$). Every node is inserted and extracted once. For every edge, we have one Decrease-key (= Relax) operation. Thus, the total running time is O($|V|^2 + |E|$) which is linear in the size of the graph if the graph is dense, that is, $|E| = \Theta(|V|^2)$. If we implement Q with binary heaps, then every operation takes time O($\log |V|$) giving a total running time of O($|V| \log |V| + |E| \log |V|$). And finally, we could also use Fibonacci heaps; then we get a running time of O($|V| \log |V| + |E|$).

The correctness of Dijkstra's algorithm follows immediately from the following lemma.

Lemma 14.1 *For all v that are extracted from Q, $d[v] = \delta(s,v)$.*

Proof. The first node that is extracted is s. We have $d[s] = 0 = \delta(s,s)$. Now let u be the first node that will be extracted such that $d[u] \neq \delta(s,u)$. We have $u \neq s$. By property 2 in Section 14.1, $d[u] > \delta(s,u)$.

Consider a shortest path P from s to u. Since s is not in the queue and u is (still) in the queue, there must be an edge (x, y) on the path such that x is not in the queue but y is. (x can be s and y can be u.)

Since x is removed from the queue before u, we have $d[x] = \delta(s, x)$. Since Relax(x, y) was called when x was removed, we have

$$d[y] \leq d[x] + w((x, y)) = \delta(s, x) + w((x, y)) = \delta(s, y).$$

The second equality follows from the fact that the subpath from s to y of P is a shortest path. Thus $d[y] = \delta(s, y)$ by property 2.

Since the weights are non-negative, $\delta(s, y) \leq \delta(s, u)$. Therefore,

$$d[y] = \delta(s, y) \leq \delta(s, u) < d[u].$$

But u is removed before y from Q, therefore $d[u] \leq d[y]$, a contradiction. ∎

14.3 Bellman-Ford algorithm

The Bellman-Ford algorithm also works with negative weights. If it detects an negative cycle, it will return an error message.

Algorithm 54 Bellman-Ford

Input: edge-weighted directed graph $G = (V, E, w)$, source $s \in V$
Output: $d[v] = \delta(s, v)$ for all v and p encodes a shortest path tree

```
1: Initialize d and p
2: for i := 1, ..., |V| − 1 do
3:    for each e = (u, v) ∈ E do
4:       Relax(u, v)
5: for each (u, v) ∈ E do
6:    if d[v] > d[u] + w((u, v)) then
7:       error "negative cycle"
```

The running time of the Bellman-Ford algorithm is obviously $O(|V| \cdot |E|)$. Why is it correct? Assume that there is no path with a negative cycle from s to v. Consider a shortest path $P = (v_0, \ldots, v_\ell)$ from s to v ($s = v_0$ and $v = v_\ell$). Before the first iteration of the outer for-loop, $\delta(s, v_0) = d[v_0]$. After the first iteration of the outer loop, the edge (v_0, v_1) is relaxed in the inner loop, since every edge is relaxed there. By property 4, $\delta(s, v_1) = d[v_1]$. After the second iteration, (v_1, v_2) is relaxed and $\delta(s, v_2) = d[v_2]$. Since P has at most $|V| - 1$ edges, $\delta(s, v_\ell) = d[v_\ell]$ holds in the end.

Intuitively speaking, if there is a negative cycle, then we can relax along this cycle an infinite number of times ($|V|$ is sufficient) and lower the values $d[v]$ every time.

Exercise 14.3 *Formally prove that if there is a negative cycle in G, then the Bellman-Ford algorithm will detect this.*

15 Hash tables

Often we only want to support the three basic operations insert, search, and delete. This is called a dynamic set or dictionary problem.

Insert(S, x)**:** adds x to S

Search(S, k)**:** finds an element with key k

Delete(S, x)**:** removes x from S.

All these operations should run extremely fast, that is, O(1). And they should be even fast in practice.

15.1 Direct-address tables

The simplest way to store the elements is *direct addressing*. Our keys come from a universe $U = \{0, 1, \ldots, m-1\}$. We assume that m is not too large and that no two elements have the same key. We store the elements in a *direct address table*, that is, an array $T[0..m-1]$. We have

$$T[\text{Key}(x)] = \begin{cases} \text{Pointer/Reference to } x & \text{if } x \in S \\ \text{NULL} & \text{otherwise} \end{cases}$$

So we store (a pointer/reference to) an element x in $T[\text{Key}(x)]$. In this way, it is very easy to realize the three operations above in constant time. But there is a problem: If the number of possible keys is much larger than the size of a typical set S that we want to store, then we are wasting a lot of space. Consider for instance a phone book. Our keys are strings of, say, length 15. So there are 26^{15} possible keys! But only a very tiny fraction of them corresponds to real names.

15.2 Hash tables

If our universe U of keys is large, say $\#U = N$, but we want to store a relatively small number of elements, say n, then direct addressing performs poorly. Instead, we use a so-called *hash function*

$$h : U \to \{0, \ldots, m-1\}.$$

m usually will be of roughly the same size as n. Instead of storing an element x in $T[\text{Key}(x)]$, like in direct addressing, we store it in $T[h(\text{Key}(x))]$. In this way, an array $T[0..m-1]$ is sufficient. Since m is comparable to n, we do not waste too much space. T is called a *hash table*.

Does this solve our problem? The answer is no (or not yet), since there might be keys $k \neq k'$ with $h(k) = h(k')$, so two elements would be stored in the same entry of T. In fact, this is unavoidable if $N > m$ by the pigeonhole principle. Even stronger, there is an index $i \in \{0, \ldots, m-1\}$ such that $h(k) = i$ holds for at least $\lceil N/m \rceil$ keys k.

One possible solution to this problem is *chaining*. Instead of storing a pointer/reference to an element in $T[i]$, we store a pointer/reference to a list that contains all elements x in S with $h(\text{Key}(x)) = i$.

We can implement our three operations as follows (in Pseudopseudocode).

Algorithm 55 CH-insert

Input: hash table T, element x
Output: inserts x into T
1: insert x at the head of $T[h(\text{Key}(x))]$.

Insertion runs in time O(1), since we insert x at the head of $T[h(\text{Key}(x))]$. We do not test whether x is already in T.

Algorithm 56 CH-search

Input: hash table T, key k
Output: returns an element with key k if one exists, NULL otherwise
1: search for key k in $T[h(k)]$

Searching might take time O(number of elements in $T[h(k)]$). In the worst case (there is no element with key k), we have search the whole list $T[h(k)]$.

Algorithm 57 CH-delete

Input: hash table T, element x
Output: deletes x from T
1: delete x from the list $T[h(\text{Key}(x)]$

Deletion can be done in time O(1), if we assume that the lists are doubly linked. Note that we assume that the element x is given. If we only have the key, we first have to search for the element.

The total space requirement for T is $O(m+n)$, the number of slots in T plus the number of elements stored in the table.

The worst case running time for searching, however, is $O(n)$, which is unacceptable. The worst case occurs if all elements hash to the same entry

$T[i]$. This can indeed happen, since by the pigeon hole principle, there is an i such that $\lceil N/m \rceil$ keys hash to $T[i]$.

But can this happen often? Of course, there are degenerate hash functions h that map all keys to the same slots $T[i]$. We assume that h distributes the keys evenly among the table, that is, $\#\{k \in U \mid h(k) = i\} \leq \lceil N/m \rceil$.

Lemma 15.1 *Assume that h distributes the keys evenly among the table. If a random set $S \subseteq U$ of size n is stored in T, then the expected number of elements in any entry of T is $\leq \lceil \frac{N}{m} \rceil \cdot \frac{n}{N}$.*

Proof. Let i be an index in $\{0, \ldots, m-1\}$. Consider the following random variable

$$X(S) = \#\{k \in S \mid h(k) = i\}.$$

We want to bound its expected value. For a key $k \in U$, let

$$Y_k(S) = \begin{cases} 1 & \text{if } k \in S \text{ and } h(k) = i \\ 0 & \text{otherwise} \end{cases}$$

Obviously,

$$X(S) = \sum_{k \in U} Y_k(S)$$

and

$$E[X] = E\left[\sum_{k \in U} Y_k\right] = \sum_{k \in U} E[Y_k].$$

The last equality follows from the fact the expected value is linear. Next, we just have to compute $E[Y_k]$. But since Y_k is $\{0,1\}$-valued (such random variables are also called *indicator variables*), we have

$$E[Y_k] = \Pr[Y_k = 1],$$

where the probability is taken over all choices of S. If $h(k) \neq i$, then $\Pr[Y_k = 1] = 0$. Therefore, only keys k with $h(k) = i$ can contribute to the expected value. If $h(k) = i$, then $\Pr[Y_k = 1] = \frac{n}{N}$, since we choose n elements out of N at random and the probability that we hit k is $\frac{n}{N}$. Hence,

$$E[X] = \sum_{k \in U} E[Y_k] = \sum_{k:h(k)=i} E[Y_k] \leq \left\lceil \frac{N}{m} \right\rceil \cdot \frac{n}{N}. \qquad \blacksquare$$

The quantity $\lceil \frac{N}{m} \rceil \cdot \frac{n}{N}$ is approximately $\frac{n}{m}$. This is called the *load factor*. Under the assumption of the lemma above, then running time of searching is $O(1 + \frac{n}{m})$. If the number of slots in T and the number of elements are linearly related, i.e., $n = \Theta(m)$, then searching can be done in constant time.

Finding a good hash function is an art, since inputs usually are not random. So hash functions in practice often include some knowledge of the input distribution.

Example 15.2 *We interpret our keys as natural numbers* $\in \{0, \ldots, N-1\}$.

1. $h(k) = k \mod m$
   This function distributes the keys evenly. If $m = 2^t$ *is a power of two, then it is often a bad hash function, since* $h(k)$ *are just the lower order bits of* k*. If* m *is a prime "not too close to a power of two", then it usually performs well.*

2. $h(k) = \lfloor m(k \cdot C - \lfloor k \cdot C \rfloor) \rfloor$*, where* $C \in (0,1)$ *is a constant.*
   The expression in the brackets multiplies k *by* C *and then removes the integral part. Experiments show that some constants work better than others, a good choice seems to be* $C = (\sqrt{5}-1)/2$*.*

16 More on hashing

16.1 Universal hashing

If the set S being inserted is chosen uniformly at random, then hashing performs well on the average under mild conditions on h (see Lemma 15.1). However, real data is usually not chosen uniformly at random. To overcome this, we now choose the hash function at random! This will have the same effect if the family of hash function we choose our function from is "nice".

Definition 16.1 *Let H be a family of functions $U \to \{0, \ldots, m-1\}$. H is called* universal *if for any two keys $x \neq y$,*

$$\Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}.$$

Above, h is drawn uniformly at random from H.

Exercise 16.1 *Show that the set of all functions $U \to \{0, \ldots, m-1\}$ is universal.*

Universal hashing gives good upper bounds on the expected running time of the Search procedure.

Theorem 16.2 *Let H be a universal family of hash functions. If a set $S \subseteq U$ with n elements is stored in a hash table T of size m with chaining using a function h drawn uniformly at random from H, then the expected list length is $\leq 1 + \frac{n-1}{m}$.*

Proof. Let x be a key from S. Let X be the random variable defined by

$$X(h) = \{k \in S \mid h(k) = h(k)\}.$$

We want to show that $E[X] \leq 1 + \frac{n-1}{m}$. For each $k \in S$, let Y_k be the indicator variable

$$Y_k(h) = \begin{cases} 1 & \text{if } h(k) = h(x), \\ 0 & \text{otherwise.} \end{cases}$$

Obviously

$$X(h) = \sum_{k \in S} Y_k(h).$$

and

$$E[X] = E\left[\sum_{k\in S} Y_k\right] = \sum_{k\in S} E[Y_k].$$

The last equality follows from the fact that the expected value is linear. Since Y_k is an indicator variable,

$$E[Y_k] = \Pr_{h\in H}[Y_k = 1] = \Pr_{h\in H}[h(k) = h(x)] \begin{cases} \leq \frac{1}{m} & \text{if } k \neq x \\ = 1 & \text{if } k = x \end{cases}$$

The last equation follows from the fact that H is universal. Thus,

$$E[X] = E[Y_x] + \sum_{k\in S\setminus\{x\}} E[Y_k] \leq 1 + \frac{n-1}{m}. \qquad \blacksquare$$

While the set of all functions forms a universal family, it is a bad family for practical purposes: The family is huge; it has size m^N. Furthermore, there is no better way than representing the functions by a table of its values. A good family of universal should have the following properties.

- The family should not be too large.
- We should be able to efficiently choose a function a random from the family. (We enhance our Pseudocode language by a function Rand(n) which returns a random integer between 0 and $n-1$.)
  Efficiently means in time polynomial in $\log N$. This induces a bound on the size of the family.
- There should be *one* algorithm that efficiently evaluates *every* function of the family.

Example 16.3 *Here is a better family of universal hash functions. We assume that $U = \{0,\ldots,2^\nu - 1\}$ and $m = 2^\mu$. Alternatively, we can interpret U as bit strings or $\{0,1\}$-vectors of length ν and $\{0,\ldots,m-1\}$ as bits string or $\{0,1\}$-vectors of length μ. Recall that $\{0,1\}$ together with addition and multiplication modulo 2 forms a field. A matrix $M \in \{0,1\}^{\mu\times\nu}$ induces a linear mapping $\{0,1\}^\nu \to \{0,1\}^\mu$. Our family of hash functions H_{lin} will be the set of linear functions $\{0,1\}^\nu \to \{0,1\}^\mu$ or alternatively, the set of all $\mu\times\nu$-matrices over $\{0,1\}$.*

Choosing a function at random from H_{lin} means that we pick a random matrix from a big bag containing all $\mu\times\nu$-matrices with $\{0,1\}$ entries. The following "local" process generates the same distribution: We flip a random coin for each entry and set it to 0 or 1 depending of the outcome. (This works since H_{lin} contains all matrices.)

Theorem 16.4 *H_{lin} is universal.*

Proof. Let $x, y \in \{0,1\}^\nu$, $x \neq y$. Let $x_1, \ldots, x_\nu$ and $y_1, \ldots, y_\nu$ be the entries of x and y, respectively. W.l.o.g. $x_1 \neq y_1$. W.l.o.g. $x_1 = 0$ and $y_1 = 1$. Let h be a function in H_{lin} with matrix M. Let $c_1, \ldots, c_\nu$ be the columns of M. We have

$$h(x) = Mx = \sum_{i=1}^{\nu} c_i x_i = \sum_{i=2}^{\nu} c_i x_i$$
$$h(y) = My = \sum_{i=1}^{\nu} c_i y_i = c_1 + \sum_{i=2}^{\nu} c_i y_i$$

We have

$$h(x) = h(y) \iff c_1 = \sum_{i=2}^{n} c_i (x_i - y_i).$$

How many matrices M fulfill $c_1 = \sum_{i=2}^{n} c_i(x_i - y_i)$? The right-hand side determines c_1 completely. Hence out of the 2^μ possible choices for c_1, only one will be a solution of $c_1 = \sum_{i=2}^{n} c_i(x_i - y_i)$. Consequently, $\Pr[h(x) = h(y)] = \frac{1}{2^\mu} = \frac{1}{m}$. ∎

To choose a random function form H_{lin}, we just have to fill a matrix with 0 and 1 chosen randomly. We can evaluate the hash function by multiplying the matrix with the key we want to hash.

16.2 Open addressing

Like chaining, open addressing is a way to resolve conflicts. All elements are stored directly in the hash table T; there are no lists stored outside the table. In this way, our hash table can become full and no further elements can be added. The load factor can never exceed 1, as it can with chaining. How do we resolve conflicts? When we want to insert an element in an occupied table entry, we look for another place in the table. More specifically, we compute a sequence of positions and check or *probe* the hash table until we find an empty slot to put the key. (Note: hashing occasionally has an alchemy-flavor; this is one of these sections.)

Formally, we are given a function

$$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}.$$

We assume that for all keys $k \in U$, the sequence $h(k,0), h(k,1), \ldots, h(k,m-1)$ is a permutation of the number $0, 1, \ldots, m-1$. We first try to store the element in $T[h(k,0)]$. If this field is occupied, we try $T[h(k,1)]$ and so on. For this reason, the sequence $h(k,0), h(k,1), \ldots, h(k,m-1)$ is also called a

probing sequence. If we do not find an empty field, we claim that the table is full.

The procedure Insert and Search can be easily realized as follows.

Algorithm 58 OA-Insert

Input: hash table T, key k
Output: inserts k into T, if T is not full

```
1: i := 0
2: j := h(k, 0)
3: while i < m do
4:    if T[j] = NULL then
5:       T[j] := k
6:       return
7:    else
8:       i := i + 1
9:       j := h(k, i)
9: error "Table full"
```

Algorithm 59 OA-Search

Input: hash table T, key k
Output: j if k is stored in $T[j]$, NULL if no such j exists

```
1: i := 0
2: j := h(k, 0)
3: while i < m and T[j] ≠ NULL do
4:    if T[j] = k then
5:       return j
6:    else
7:       i := i + 1
8:       j := h(k, i)
8: return NULL
```

What about deletion of elements? This is a real problem. We cannot just set $T[i] :=$ NULL to delete an element, since afterwards, the search procedure might not be able to find elements that are still in the table. One solution is to not remove elements from the table but mark them as deleted by using an extra flag. If there are many deletions, then the table is quite full although it only stores a small number of keys. But if only a few deletion operations are expected (or none at all), this works well.

How can one construct functions $h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$. Here are some examples:

Linear problem: Let $h' : U \to \{0, \ldots, m-1\}$ be an "ordinary" hash

function. We set

$$h(k,i) := h'(k) + i \mod m.$$

$h(k,0),\ldots,h(k,m-1)$ is a permutation of $0,\ldots,m-1$, so h meets our requirements. In fact, $h(k,0),\ldots,h(k,m-1)$ is a cyclic shift. This means that altogether, h produces at most m different probing sequences.

A big problem of linear probing is *primary clustering*: Assume that h' distributes the keys evenly among the entries of T. Consider an empty table and we insert an element. Since h' distributes the keys evenly, every cell of T has the same chance of getting the element. Let i be the cell to which the element goes. Now we add a second element. Every cell of T again has a $1/m$ chance of getting the element except for one, $i+1$. This cell has a chance of $2/m$, since the element goes there if it goes to cell i (which is occupied) or cell $i+1$. In general, if j cells are occupied before a particular cell, then the probability that an element will be stored in this cell is $(j+1)/m$. Such long runs are bad since they increase the running time of search.

Quadratic probing: Quadratic probing uses functions of the form

$$h(k,i) = h'(k) + ai + bi^2 \mod m$$

with appropriate constants a and b. It avoids primary clustering (and replaces it by a milder form called, guess what, secondary clustering). On the other hand, it also generates at most m different probe sequences.

Double hashing: Here we use two hash functions $h_1, h_2 : U \to \{0,\ldots,m-1\}$ and set

$$h(k,i) = h_1(k) + ih_2(k) \mod m$$

If h_1 and h_2 are carefully chosen, then we get $\Theta(m^2)$ different probe sequences.

16.3 Perfect hashing

Up to now, all our upper bounds on the search time are only true in expectation. For universal hashing, the expected search time for any element is bounded by $O(1+(n-1)/m)$ but that does not exclude that for a particular choice of h, the search time is much higher.

In the following, we assume that we are given a fixed set of keys S and deletions are not allowed. (This is not an unrealistic scenario, think of a dictionary or phone book.) A hash function h is called *perfect* for S if every call of the Search procedure has running time $O(1)$.

Lemma 16.5 *Let H be a universal family of hash functions and $m = n^2$. We have*

$$\Pr[\textit{for all } x \neq y\textit{: } h(x) \neq h(y)] \geq \frac{1}{2}.$$

Proof. For every pair $x \neq y$, we have $\Pr[h(x) = h(y)] \leq 1/m$. Therefore,

$$\begin{aligned}
\Pr[\text{for all } x \neq y\text{: } h(x) \neq h(y)] &= 1 - \Pr[\text{there are } x \neq y\text{: } h(x) = h(y)] \\
&\geq 1 - \sum_{x \neq y} \Pr[h(x) \neq h(y)] \\
&\geq 1 - \binom{n}{2} \cdot \frac{1}{m} \\
&\geq \frac{1}{2},
\end{aligned}$$

where the sum in the second line is over all unordered pairs. ∎

If we can accept the fact that the size of the table is quadratic in the number of stored keys, then half of the functions $h \in H$ are perfect, since they map every key to a distinct cell. However, if n is large, then we want m to be linear in n. We can achieve this by a two stage construction (see below).

How can we construct such an h? We can just draw a function at random. With probability 1/2, it will be perfect. We can check this, since we know the set S of keys explicitly. If h is not perfect, we draw another function and so on. The probability that we do not find a perfect function after i such steps is $\leq \frac{1}{2^i}$ which goes to zero rapidly.

To get down to linear space, we first choose a hash function $h : U \to \{0, \ldots, n-1\}$ from a universal family. h will have collisions, let m_0 be the number of keys that are mapped to 0 by h, m_1 be the number of keys that are mapped to 1 and so on. Let S_i be the set of keys that are hashed to i. We use a hash function $h_i : S_i \to \{0, \ldots, m_i^2 - 1\}$ as constructed in Lemma 16.5. A key x with $h(x) = i$ is hashed to $(i, h_i(x))$. Finally the hash values (i, j), $0 \leq i \leq n-1$ and $0 \leq j \leq m_i$ are identified with $\{0, \ldots, M-1\}$ where $M = \sum_{i=0}^{n-1} m_i^2$. Altogether, we have a two-stage process: First apply h and then the appropriate h_i. We have no collisions by construction.

We are done if we can choose h in such a way that

$$\Pr_h[\sum_{i=0}^{n-1} m_i^2 > 4n] < \frac{1}{2}.$$

In this case, we can choose h such that the total table length is $\leq 4n$. It is sufficient to show the following lemma. (Use Markov inequality! Or do it on foot: If for at least half of the hash functions, the length is $> 4n$, then the expected length is $> \frac{1}{2} \cdot 4n = 2n$, a contradiction.)

Lemma 16.6 $E[\sum_{i=0}^{n-1} m_i^2] < 2n$.

Proof. For $x, y \in$, let $Z_{x,y}$ be the random variable defined by

$$Z_{x,y}(h) = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

We have

$$E\left[\sum_{i=0}^{n-1} m_i^2\right] = E\left[\sum_{x \in S} \sum_{y \in S} Z_{x,y}\right],$$

because for every $0 \leq i \leq n-1$, every pair $(x, y) \in S_i \times S_i$ contributes 1 to the sum on the right hand side yielding a total contribution of m_i^2. By the linearity of expectation,

$$\begin{aligned} E\left[\sum_{x \in S} \sum_{y \in S} Z_{x,y}\right] &= n + \sum_{x \in S} \sum_{y \neq x} E[Z_{x,y}] \\ &\leq n + n(n-1) \cdot \frac{1}{n} \\ &< 2n \qquad \blacksquare \end{aligned}$$