Algorithmen und Datenstrukturen

Institut für Informatik

Wolfgang Mulzer

Sommersemester 2024

Preface

This is an evolving draft of the lecture notes for „Algorithmen und Datenstrukturen“, a new class that will be taught from the summer semester 2024 at the department of Computer Science of Freie Universität Berlin.

Inhaltsverzeichnis

I

Introduction

KAPITEL 1

Introduction

Data and Algorithms. The main topic of this class will be ways to deal with *data*.

In particular, we will learn how to organize data in a computer memory (*data structures*), and we will look at methods to manipulate the data and to answer questions about it efficiently (*algorithms*).

Our goal will be not only to see how these methods work, but also to reason about them rigorously and to develop ways for analyzing their properties. These properties include, for example, the *correctness* (does the method always work?), the *running time* (how fast is the method?), or the *space usage* (how much memory is required?). For our analysis, we will use mathematical methods and aim for rigorous proofs. As such, the topics of this class belong to the field of *theoretical* computer science, but we will always try to keep the applications in mind.

Algorithms. Before we begin, let us take a closer look at the notion of an algorithm and at ways to evaluate the quality of a given algorithm.

Definition: An algorithm is a *finitely described*, *general*, and *effective* procedure that transforms an *input* into an *output*.

Let us see what these terms mean:

- **input/output**: an algorithm always receives an *input* and produces an *output*. Depending on the algorithm, the input can be of different types, e.g., a text string, a sequence of numbers, a graph, or a map, and the same holds for the output. Some algorithms produce only two types of output: `Yes` or `No`. Such algorithms are called *decision algorithms*.

- **general**: an algorithm typically does not work only for a specific input, but it has a large class of possible inputs, usually infinitely many.[1] The algorithm must work for all of these inputs.

- **finitely described**: an algorithm has a finite description, either in a semi-formal (verbal description, pseudo-code) or a formal language (programming language,

[1] Of course, this is only in theory. In practice, there are limitations on the input size, due to the physical properties of the hardware and the time at our disposal. However, saying that there can be infinitely many inputs is a good mathematical abstraction for the fact that the inputs can be *arbitrarily large*.

Turing machine). Ideally, the algorithm is very short and fits on a single piece of paper. Even though the algorithm is finite, it usually can deal with arbitrarily large inputs that can be much larger than the algorithm itself.

- **effective**: an algorithm usually works by prescribing a sequence of *steps* on a given input. Each such step must be *effective*, i.e., it must be possible to carry out theses steps in "the physical world".[2]

A *good* algorithm usually has (some of) the following properties:

- **correct**: for every input, the algorithm terminates after a finite number of steps, and it provides the specified result.
- **fast**: for every input, the algorithm provides the answer after a short time. Of course, we expect that the algorithm will take longer as the inputs get larger, but this increase in the running time should be "reasonable".
- **space efficient**: similarly to the previous point, the memory usage of the algorithm should be small and increase "reasonably" as the input grows.
- **easy to understand**: it should not be too difficult to understand how and why a given algorithm works.
- **easy to implement**: it should not be too difficult to program the algorithm in a given programming language.

How can we evaluate the quality of an algorithm?

Possibility A: Benchmarking (using experiments). We run the algorithm on several "representative" input instances and observe how it behaves (by measuring the running time, the space requirement, etc.)

The big advantage of benchmarking is that it gives the most direct information about how an algorithm behaves "in the real world". However, there are several disadvantages. First, using only experiments, we cannot prove that an algorithm is correct for every possible input. Experiments can only show definitely that an algorithm is *incorrect*. Second, it is not clear how we choose the "representative" input instances for our experiments. How do we know that they are representative? For which scenario? Third, when performing an experiment, we typically do not just evaluate the algorithm, but other factors also come into play, such as the skill of the programmer who implemented the algorithm, the quality of the programming language, the quality of the compiler, the underlying hardware, etc. Fourth, it is not clear how to compare the results of different experiments over time, since a lot of factors can vary. When we have a new algorithm that we would like to compare to

[2] Of course, what we can do in the physical world depends on the current state of our technology. About 200 years ago, it was not possible to build precise electronic computers that we take for granted today. In the future, we may be able to construct quantum computers that allow for additional *efficient* operations that are not available today.

existing ones, how do we make sure that we can recreate the conditions of the previous experiments?

Thus, even though benchmarking is a good approach, it has many drawbacks. For a satisfactory theoretical understanding of our algorithms, we need to proceed differently.

Possibility B: Theoretical Analysis (using Mathematics). To analyze our algorithm theoretically, it is sufficient to have an abstract description of the algorithm in *pseudo-code* or for an *abstract machine model*. An actual implementation is not necessary.

Typically, the goal of a theoretical analysis understand how the performance (running time, space requirement) of an algorithm behaves as a function of the *input size*. Thus, for each possible input size, we would like to assign a number that represents the behavior of the algorithm for this input size, and we analyze how this function behaves as the input size grows. There are many different ways to define such a number, but typically we focus on the *worst-case performance* of the algorithm, that is, the maximum possible running time/space requirement/etc. for any given input size.

There are many different abstract machine models that can used to describe an algorithm, e.g., Turing machines, λ-calculus, Markov algorithms, formal grammars of Type-0, etc. For our purposes, the most useful abstract machine model is the *random access machine* (RAM). The RAM is a mathematical abstraction of an actual (simple) computer with a small instruction set and infinite memory that is structured in cells and that allows for random access.

In principle, once we have formally specified the properties of a RAM, we can formulate our algorithm as code for the RAM and then, for every given input, count the number of RAM instructions that the algorithm executes (notably, this is done in Donald Knuth's famous books on *The Art of Computer Programming*). This gives a detailed theoretical picture on the performance of the algorithm. However, this analysis can still be very cumbersome and may be full of unnecessary details

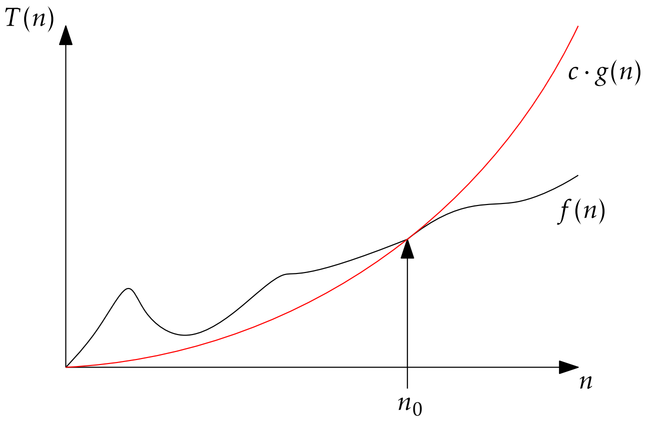
Thus, we typically represent our algorithm in *pseudo-code*, an abstraction of an imperative programming language that allows for "shortcuts" when it helps in understanding the algorithm. When analyzing the pseudo-code, we have an implementation of a RAM in mind, and we expect that one line of pseudo-code corresponds to a constant number of RAM operations.

Thus, when analyzing the running time of an algorithm theoretically as a function of the input size, constant factors do not matter, because they can depend on the specifics of the pseudo-code or the chosen instruction set of the RAM. In order to not be distracted by these effects (and also in order to be able to ignore terms that become less influential as the input size grows), we use *O-notation* to represent the performance concisely.

Definition: Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. Then, we write

- $f(n) = O(g(n)$ if and only if there exists a constant $c > 0$ and a number $n_0 \in \mathbb{N}$, such that for all $n \geq n_0$, we have $f(n) \leq c \cdot g(n)$.
- $f(n) = \Omega(g(n)$ if and only if there exists a constant $c > 0$ and a number $n_0 \in \mathbb{N}$, such that for all $n \geq n_0$, we have $f(n) \geq c \cdot g(n)$.

- $f(n) = \Theta(g(n)$ if and only we have $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.



To conclude, the goal of a theoretical analysis of an algorithm is to derive a function in O-notation that represents the worst-case performance of the algorithm as the input size grows. This yields a short and easily understood performance measure for an algorithm, and it allows for an easy comparison between algorithms.. However, this simplicity comes at a cost. The statement may be too concise, and it may hide too many details. We only analyze the worst-case, so it may be possible that for “many interesting” inputs, the algorithm is much better than our analysis suggests. Also, the O-notation gives only an *asymptotic estimate* of the performance of an algorithm, and it is thus only useful for inputs that are ”large enough”. Many efforts have been made to address these issues, and we will see some of them in later classes. For now, however, this approach will be a very good way to obtain a first understanding of how different algorithms behave.

Data structures and Abstract Data Types. A *data structure* is a way to organize data in a computer memory in order to allow for efficient operations on the data. Typically, for a given task, there are many ways how the data can be organized, with different advantages and disadvantages concerning, e.g., the memory requirement or the time needed for supporting different operations (where different choices can result in some operations being faster and other operations being slower).

Abstract Data Types (ADTs) are a way hide these choices behind a layer of abstraction, separating the desired functionality from the choice of implementation. To understand the motivation, we take a digression into the realm of good software design.

When undertaking a large programming task, our goal is to develop software that is

- **easy to maintain:** when new features need to be added or when existing bugs should be fixed it should be possible to effect these changes without the need to change too much of the existing code.
- **easy to understand:** when new people are added to the project (or when the original developers revisit the project after a longer absence), it should be possible to get a quit overview of the functionality and of the structure of code.

- **flexible:** it should be easy to change the functionality and to develop different versions of the software.
- **free from bugs:** The software should not have any serious flaws. If there are flaws (as will almost certainly be the case), they should have only a limited effect on the whole system, and it should be easy to locate and fix them once they are discovered.

The field of *software engineering* tries to address these issues and to find ways to develop better software. One very important principle from software engineering that is relevant in our context was identified in 1972 by David Parnas. It is called "information hiding" and calls for a strict separation of the code in different, isolated parts. Though controversial at the time, it is now a well-accepted method in software development.

The idea of information hiding is to partition a large software project into multiple parts. Each part has a clearly defined purpose and a precisely defined *interface* that allows it to communicate with the rest of the system. Crucially, the communication with the outside happens *only* through the interface. The interior of the part remains hidden and can be modified and exchanged at will, as long as the interface is not affected.

In the realm of data structures, this idea is specialized in the notion of an *abstract data type*.

Definition: An *abstract data type* (ADT) is a collection of data whose objects can be accessed only through a clearly specified interface that provides a fixed set of operations. The implementation cannot be accessed directly and can be replaced, if desired. The individual operations are specified precisely by giving a precondition, and effect and a return value for each operation.

There are many different abstract data types, depending on the kind of data and on the specific operations that are supported. In *Konzepte der Programmierung*, we have already encountered some examples of abstract data types, such as `Stack` (Last-In-First-Out storage), `Queue` (First-In-First-Out storage) or `Priority Queue`.

By using abstract data types in our algorithms, we gain several advantages. First, it increases the safety of our code, because the inner workings of the data structure cannot be disturbed from the outside by accidental programming errors. Second, we gain flexibility, because it is easy to replace one implementation of an ADT by another, which makes it easy to adapt our code to different needs. Third, ADTs make it easier to understand an algorithm, because the make the design more modular and let us separate the data structure logic from the rest of the algorithm. Fourth, ADTs are easy to use, because we only need to understand a simple interface.

Since information hiding and abstract data types are by now well-accepted notions in software design, most modern programming languages have constructs to support these ideas, e.g., traits in Scala, interfaces in Java, or virtual classes in C++.

We will now talk about the implementation of two specific ADTs.

II

Ordered Dictionaries

KAPITEL 2

Ordered Dictionaries

We now describe the ADT *ordered dictionary*, also called *ordered map*. The goal of an ordered dictionary is to maintain a set of *entries*. An entry consists of two parts, a *key* and a *value*. For each key, there is at most one entry, and there is an order among the keys. We would like to be able to create new entries for new keys, update values for existing keys, lookup the value for a given key, delete existing entries, and to determine the predecessor and the successor entry for a given key.

More precisely, in an ordered dictionary, we have two underlying sets, the *key set* K and the *value set* V. The key set is *ordered*, that is, for any two distinct keys, we can determine which one is smaller (and the results of these comparisons are consistent). An *entry* is an ordered pair (k,v) from the Cartesian product $K \times V$, and our goal is to maintain a dynamic set $S \subseteq K \times V$ such that each key appears at most once in an entry of S.

In an ordered dictionary, we would like to support the following operations:

- `put`(k,v), where $k \in K$ is a key and $y \in V$ is a value. This operation has no precondition and does not have a return value. The effect is as follows: if S does not contain an entry with key k, we create add to S the new entry (k,v); if S already has an entry with key k, then this entry is deleted and replaced by the entry (k,v). Mathematically, the desired effect of `put` can be described as $S_{\text{new}} = (S_{\text{old}} \setminus (\{k\} \times V)) \cup (k,v)$.

- `get`(k), where $k \in K$ is a key. The precondition is that S contains an entry with key k (otherwise, `get`) will raise an exception). There is no effect. The return value is the value v of the (unique) entry $(k,v) \in S$ that has key k.

- `remove`(k), where $k \in K$ is a key. There is no precondition and no return value. The effect is as follows: if S contains an entry with key k, then this entry is removed in S. Otherwise, S remains unchanged. Mathematically, the desired effect of `remove` can be described as $S_{\text{new}} = S_{\text{old}} \setminus (\{k\} \times V)$.

- `isEmpty`:, There is no precondition and no effect. The operation returns `true`, if $S = \emptyset$, and `false`, if $S \neq \emptyset$.

- `size`. There is no precondition and no effect. The operation returns $|S|$, the number of entries in S.

- `min`. The precondition is that S is not empty, $S \neq \emptyset$. There is no effect. The operation returns the smallest key that appears in S, $k_{\text{min}} = \min\{k \mid (k, v) \in S\}$.

- `max`. The precondition is that S is not empty, $S \neq \emptyset$. There is no effect. The operation returns the largest key that appears in S, $k_{\text{max}} = \max\{k \mid (k, v) \in S\}$.

- `pred`(k), where $k \in K$ is a key. The precondition is that k is larger than the smallest key in S, $k > \min\{k \mid (k, v) \in S\}$. There is no effect. The operation returns the *predecessor* of k in S, that is, the largest key that appears in S and that is smaller than k. Formally, the return value is $k_{\text{pred}} = \max\{k' \mid (k', v) \in S, k' < k\}$.

- `succ`(k), where $k \in K$ is a key. The precondition is that k is smaller than the largest key in S, $k < \max\{k \mid (k, v) \in S\}$. There is no effect. The operation returns the *successor* of k in S, that is, the smallest key that appears in S and that is larger than k. Formally, the return value is $k_{\text{succ}} = \min\{k' \mid (k', v) \in S, k' > k\}$.

In the following chapters, we will see several ways of implementing the ADT ordered dictionary.

KAPITEL 3

Binary Search Trees

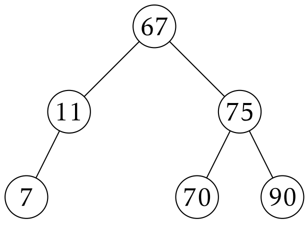
A simple way to implement the ADT ordered dictionary is via binary search trees.

Definition: An (ordered) *binary tree* is defined inductively as follows: (i) the *empty tree* $\perp$ is a binary tree; and (ii) if T_ℓ and T_r are binary trees, the we can obtain a binary tree by creating a *root node* v and by making T_ℓ the *left subtree* of v and T_r the *right subtree* of v. If T_ℓ is not empty, we create an *edge* between v and the root node w_ℓ of T_ℓ, and we call w_ℓ the *left child* of v. Similarly, if T_ℓ is not empty, we create an edge between v and the root node w_r of T_r, and we call w_r the *right child* of v. We call v the *parent node* of w_ℓ and w_r (if they exist).

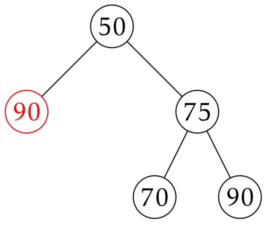
Unravelling the definition, we see that a binary tree T consists of nodes and edges, that are connected in a hierarchical fashion. A node v that does not have any children (that is, both subtrees of v are empty) is called a *leaf*. A node v that is not a leaf (that is, v has at least one child) is called an *interior node*. For every node v in T, the number of edges that connect v to the root of T is called the *depth* of v or the *level* of v (note that the root has level 0). The maximum depth over all nodes in T is called the *height* of T. If a node w lies in a subtree of a node v, then w is called a *descendant* node of v, and v is called an *ancestor* node of w. Otherwise, if w does not lie in in a subtree of v, and if v does not lie in a subtree of w, we call v and w *unrelated*.

Definition: Let K be an ordered set of keys and V be a set of entries. A *binary search tree* is an (ordered) binary tree T in which the nodes store entries from $K \times V$. The keys of the entries must fulfill the *binary search tree property*: For every node v of T, let k be the key of the entry that is stored in v. Then, for all keys k' in the left subtree of v, we have $k' < k$, and for all keys k' in the right subtree of v, we have $k' > k$.

this is a binary search tree



this is **not** binary search tree



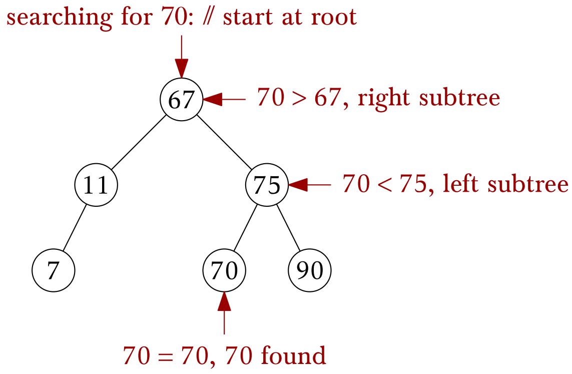
Implementation. We can implement a binary search tree as a collection of *node objects* that are connected through references. A node object has `n` has the following attributes:

- `k`, `v`: the key and the value stored in `n`.
- `parent`: a reference to the node object for the parent `n` (NULL if `n` is the root).
- `left`, `right`: references to the objects for the left and the right child of `n` (NULL if these children do not exist).

Now we can implement the operations as follows:

The operation get. To search for a key k in the binary search tree, we start at the root, and in each step, we compare k to the key k' in the current node `n`. If $k = k'$, we have found the desired entry. If $k < k'$, we proceed to the left child of `n`, and if $k > k'$, we proceed to the right child of `n`. If we reach an empty tree (that is, if the current node becomes NULL, we know that k is not present in the tree. The pseudo-code is as follows:

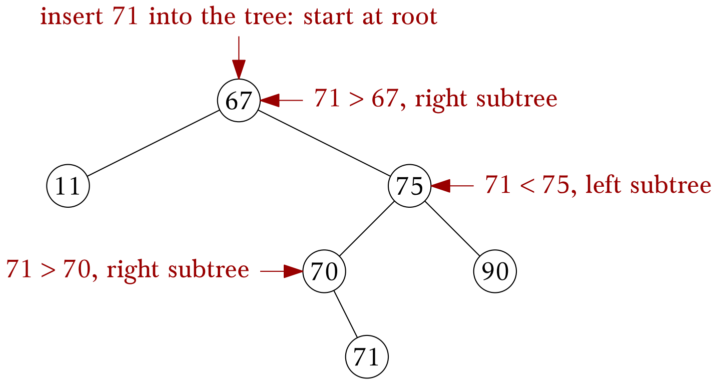
```
get(k)
    // start at the root
    n <- root
    // as long as we have not reached an empty tree
    while n != NULL do
        // is k in the current node?
        if n.k == k then
            // return the corresponding value
            return n.v
        // is k smaller than the current key?
        if k < n.k then
            // go to the left subtree
            n <- n.left
        // otherwise, k is larger than the current key?
        else
           // go to the right subtree
           n <- n.right
  // we have reached an empty tree. This means that k is not in the tree
  throw NoSuchElementException
```



The operation put. To update/insert an entry (kv), we first locate the key as in get. If we find k, we simply update the value and return. If we do not find k, the search ends at and empty tree that marks precisely the location where the entry (k,v) should be inserted. We create a new node for (k,v) and add it to the tree.

```
put(k,v)
    // is the tree empty?
    if root == NULL then
        // create a new root node and return
        root <- new node for (k, v)
       return
    // otherwise, we locate the position for k, starting at the root
    n <- root
    while true do
        // is k in the current node?
        if n.k == k then
            // update the value and return
            n.v <- v
            return
         // is k smaller than the current key?
         if k < n.k then
             // if the left subtree is not empty, go there
             if n.left != NULL then
                 n <- n.left
             // if not, create a new node for (k, v) and return
             else
                 n.left <- new node for (k, v)
                 return
         // otherwise, k is larger than the current key
         else
             // if the right subtree is not empty, go there
             if n.right != NULL then
                 n <- n.right
```

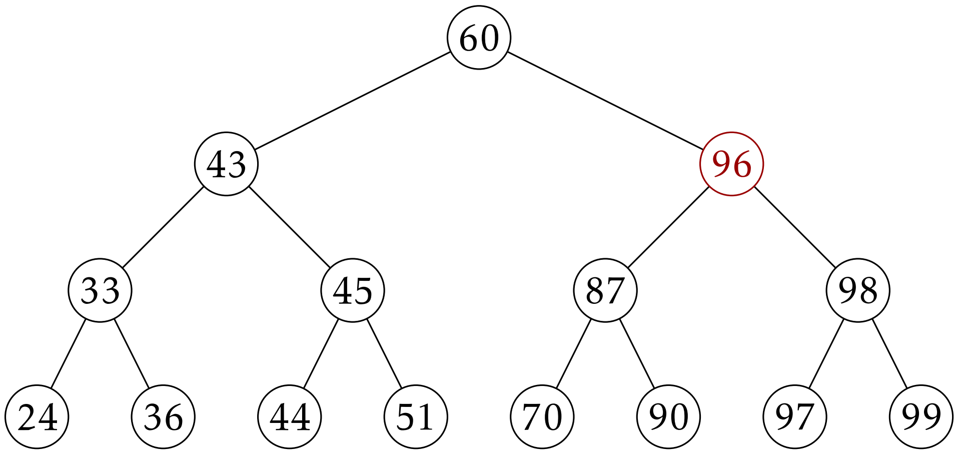
```
// if not, create a new node for (k, v) and return
else
   n.right <- new node for (k, v)
   return
```



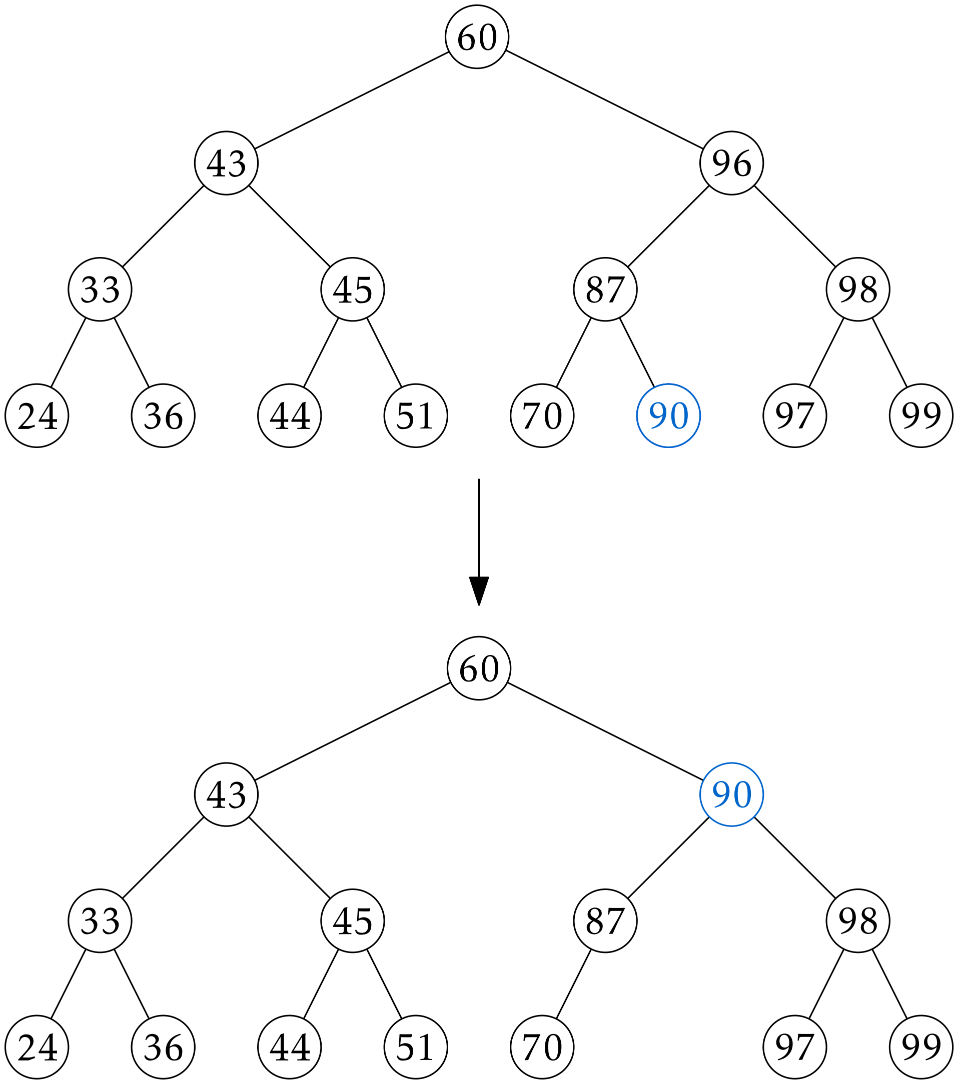
The operation remove. To delete an entry for a key k from a binary search tree T, we first locate the node n that stores the entry for k. If no such node exists, k is not in T, and we raise an error. If n is a leaf, we can simply remove n from T by replacing it with an empty tree (if n is also the root, this means that T becomes empty). If n has only one non-empty subtree, we replace n by this subtree. If n has two non-empty subtrees, we find the node m with the predecessor entry for k in the left subtree of n. We replace the entry in n with the entry in m, and we remove m from T. The latter removal operation is easy, because now we know that m has at most one non-empty subtree.

```
remove(k)
    // locate the node n that contains k
    n <- root
    while n != NULL && n.k != k do
        if k < n.k then
            n <- n.left
        else
            n <- n.right
    if n == NULL then
        throw NoSuchElementException
    // if at least one subtree of n is empty, replace n by the other one
    if n.left == NULL then
        // replace n by the right subtree
        // we show the necessary pointer operations only once
        // Is the right subtree of n nonempty?
        if n.right != NULL then
            // If so, update the parent pointer for the right child
            n.right.parent <- n.parent
        // Does n have a parent?
        if n.parent != NULL then
            // If so, update the correct child of this parent
            if n.parent.left == n then
                n.parent.left <- n.right
            else
                n.parent.right <- n.right
        // if n does not have a parent, it is the root
        else
            root = n.right
        return
    else if n.right == NULL then
        replace n by its left child
        return
    // now, n does not have an empty subtree
    // we locate the node m that contains the
    // predecessor entry for k
    m <- n.left
    while m.right != NULL do
        m <- m.right
    // now, m contains the predecessor entry for k,
    // and m does not have a right child
    move the predecessor entry from m to n
      (overwriting the entry for k)
    replace m by its left subtree
```

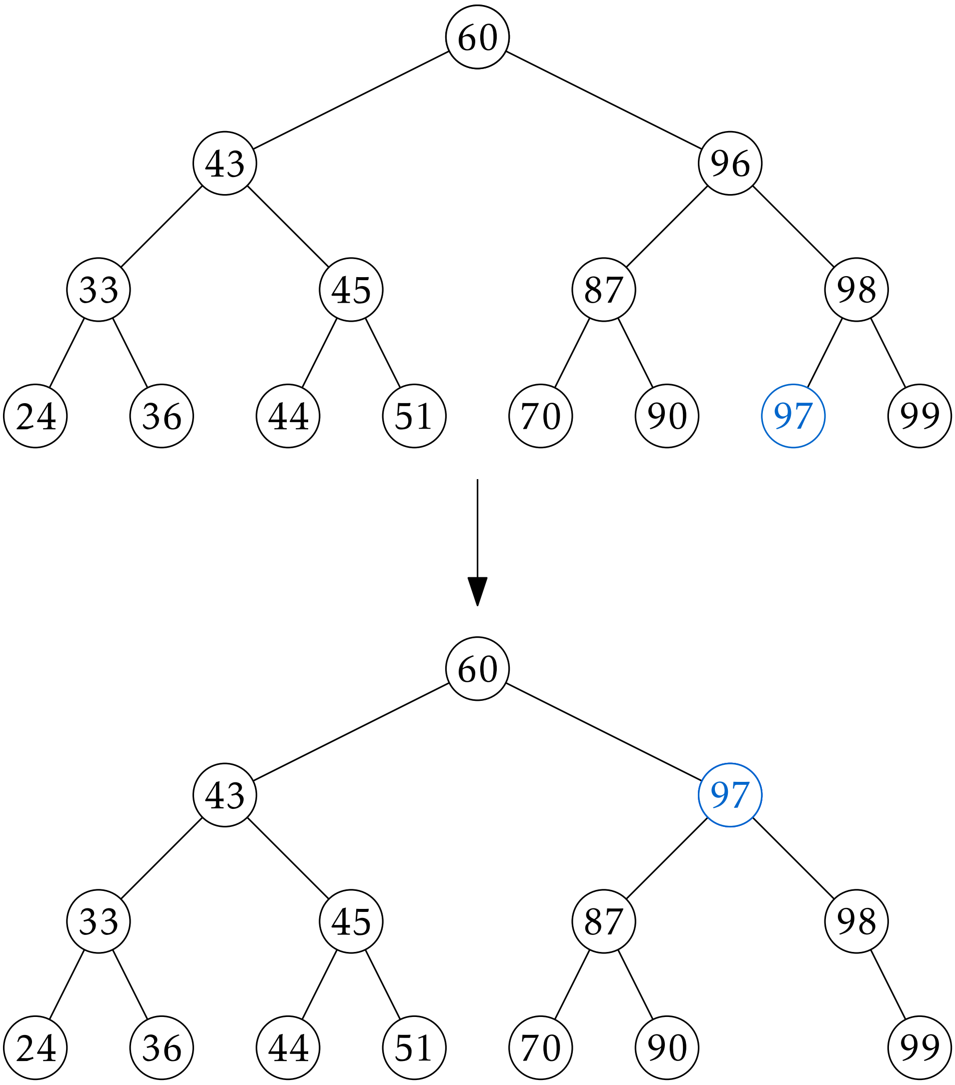
remove 96:



option 1: use the biggest child of left subtree

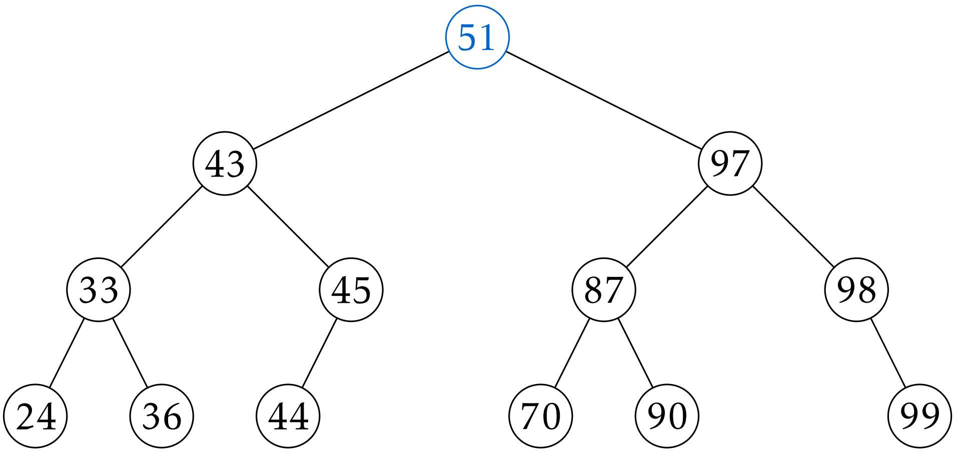


option 2: use the smallest child of right subtree

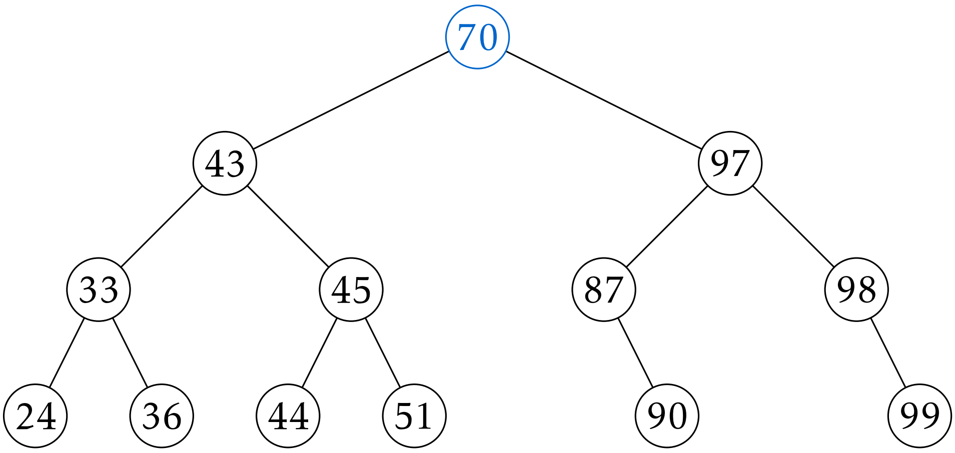


now remove 60 out of this tree:

option 1:

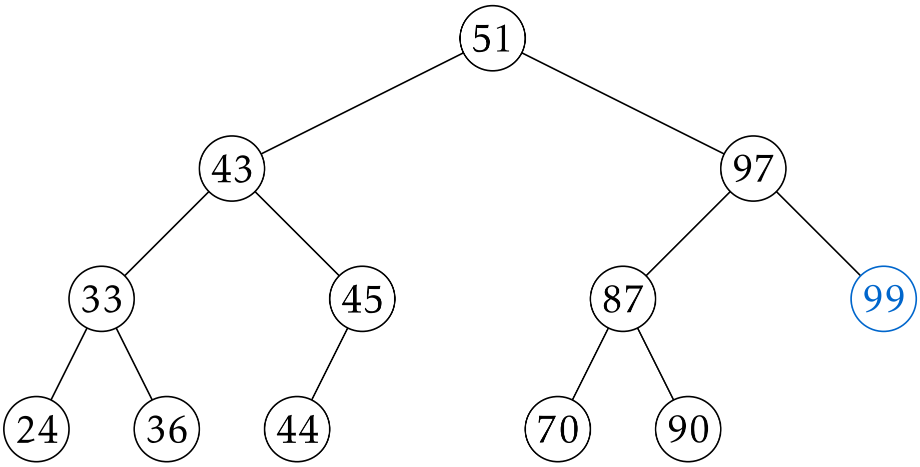


option 2:



now remove 98 out of this tree:

there is only one option, since 98 only has one child:

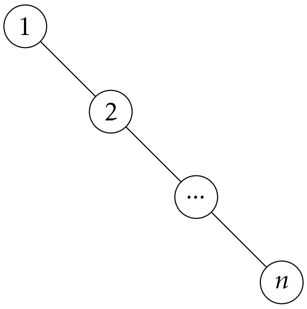


The details of the implementation for the remaining operations of the ADT ordered dictionary on a binary search tree are left as an exercise.

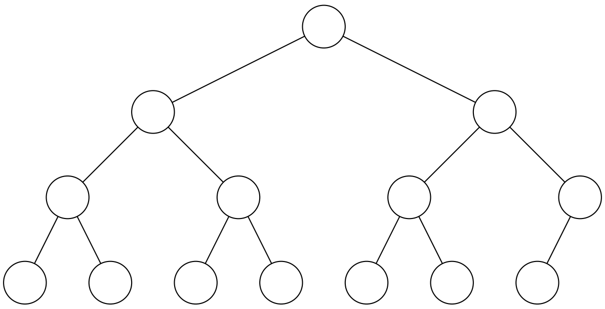
Analysis. Let us now analyze the performance of the binary search tree operations. First, correctness can be shown in a relatively easy manner: to see that the look-up operation `get` is correct, we can proceed by an induction on the height of the binary search tree to show that `get` will find the entry for the given k, if it exists in the tree. From this, we can also derive that `put` is correct, using that the position of the empty tree where the new node is inserted preserves the binary search tree property. Finally, to show correctness of `remove`, we distinguish cases to show that after the deletion, the binary search tree property is preserved. The details are left as an exercise.

What can we say about the running time of the binary search tree operations? An inspection of the code shows that all three operations `get`, `put`, `remove` proceed by a single descent down the tree, possibly backtracking along the path in the case of `remove`. The running time of the operations is proportional to the number of nodes along this path. Thus, in the worst case, we need to proceed to a leaf that has the largest depth in the tree. In this case, the running time is proportional to the *height* of the tree. Thus, we can conclude that in the worst case, the running time of `get`, `put` and `remove` in a binary search tree T is $\Theta(h)$, where h is the height of T.

What can we say about the height of a binary search tree T that stores n entries? In the worst-case, not much: it can happen that every node in T has only one child, in which case the height is $\Theta(n)$. For example, this happens if we insert entries with keys $1, 2, \dots, n$ in this order into an initially empty binary search tree. In this case, we might as well store our entries in a linked list and do without the additional complications of implementing a binary search tree.



However, the situation can be much better: we say that T is a *perfect* binary tree if all levels of T (except possibly the last level) contain as many nodes as possible (we already saw perfect binary trees when we talked about binary heaps in *Konzepte der Programmierung*). In this case, the height of T will be $\Theta(\log n)$, where n is the number of entries, and the running time for our operations will be *exponentially faster* than in the worst case. Even more encouragingly, we can show that, e.g., if the keys inserted in a *random order* into an initially empty binary search tree, the *expected* height of T will be $\Theta(\log n)$.



The last row can also possibly be full!

Now, we have two options: first, we can accept that the *worst-case* performance of our ordered dictionary implementation can be bad, and seek solace in the knowledge that the *average case* will be much better and hope that our operation sequence will contain enough randomness so that the average case is more representative for what happens. From this, we gain a simpler implementation, but at the cost of more uncertainty of the algorithmic performance. Second, we can try to adapt our binary search tree implementation to make sure that the worst-case cannot occur. In the next few chapters, we will see ways of how to achieve this.

KAPITEL 4

AVL-Trees

We will now look at a way to implement a binary search tree in such a way that the height of the tree is $O(\log n)$ at all times, where n is the number of entries that are stored in the tree. The idea is as follows: in the previous chapter, we saw that a perfect binary search tree always has height $O(\log n)$. Thus, it would be nice if we could ensure that we have a perfect binary search tree after each insertion and deletion, possibly by restructuring the tree (as we did, e.g., with binary heaps in *Konzepte der Programmierung*). However, it turns out that perfect binary search trees are too rigid for this. If we try to keep the binary search tree perfect after each operation, the height would be $O(\log n)$ at all times, but at the cost of a very expensive restructuring operation, defeating the purpose of an overall fast running time for all operations.

The solution is that we do not keep a perfect binary search tree, but a binary search tree that is only *approximately perfect*. The trick is now to find a notion of "approximately perfect" that simultaneously ensures that the height is $O(\log n)$ at all times and that can be maintained with a restructuring operation that is not too expensive.

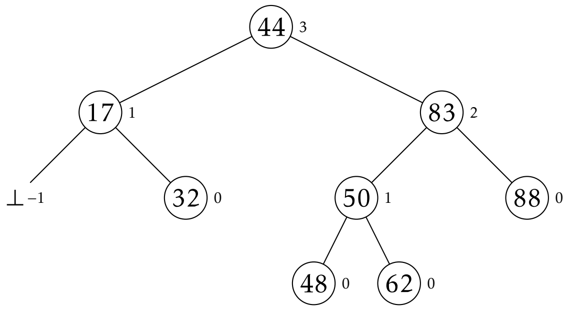
One way to achieve this was given by Adelson-Velski and Landis. They present a variant of binary search trees that are now called AVL-trees, according to the initials of their inventors. The idea is to relax the requirement of the perfect binary search tree that all levels (except for possibly the last one) should be filled completely to the requirement that for every node v of the tree, the subtrees of v should have approximately the same height. The precise definition is as follows:

Definition: Let T be a binary search tree. We say that T is an *AVL- tree* if for every node v of T, the following holds: let h_ℓ be the height of the left subtree of v and h_r be the height of the right subtree of v (we define the height of the empty tree as -1. Then, we have

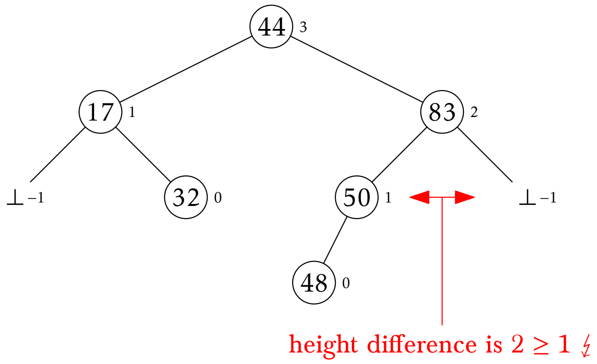
$$|h_\ell - h_r| \leq 1$$

, that is, the heights must be either the same or differ by at most 1.

this is an AVL-tree



this is **not** an AVL-tree



We will now see that this property ensures that an AVL-tree behaves approximately like a perfect tree.

Satz 4.1. *Let T be an AVL-tree with n nodes. Then, T has height $O(\log n)$.*

Beweis. Let h be a height. Our first goal is to determine a lower bound on the *minimum* number F_h of nodes that an AVL-tree of height h can have. In a binary tree that is essentially only a path, this quantity may grow only linearly with h, whereas in a perfect tree, this quantity grows exponentially with h. We will show that in an AVL-tree, F_h grows exponentially with h. First, we observe that for all $h \geq 1$, we have

$$F_h > F_{h-1}, \tag{4.1}$$

Second, we note that

$$F_0 = 1, \tag{4.2}$$

because there is only one tree of height 0, the tree that consists only one node that is simultaneously the root and a leaf. Third, we have

$$F_1 = 2, \tag{4.3}$$

because all three possible binary trees with height 1 are valid AVL-trees, and two of them consist of two nodes, while the third one has three nodes. Finally, for all $h \geq 2$, we have

$$F_h = F_{h-2} + F_{h-1} + 1. \tag{4.4}$$

To see why, let T be an AVL-tree of height h that has the minimum number of nodes. Then, T consists of a root v and a left and a right subtree of of v, such that (i) both subtrees have height at most $h-1$, (ii) one subtree has height exactly $h-1$; and (iii) the height of the two subtrees differs by at most 1. Since the two subtrees are independent of each other(and of the root), it must be the case that each subtree contains the minimum number of nodes for its respective height. Due to (4.1), the second subtree of v must have height exactly $h-1$.

Now, we use induction on h to show that for all $h \in \{0, 1, \dots, \}$, we have

$$F_h \geq \left(\frac{3}{2}\right)^h. \tag{4.5}$$

For the base of the induction, we check the cases $h \in \{0, 1\}$. By (4.2, 4.3), we get

$$F_0 = 1 \geq \left(\frac{3}{2}\right)^0 \text{ and } F_1 = 2 \geq \left(\frac{3}{2}\right)^1.$$

Now, for the inductive step, fix an $h \geq 2$. Using (4.4) and the inductive hypothesis, we get

$$\begin{aligned} F_h = F_{h-2} + F_{h-1} + 1 &\geq \left(\frac{3}{2}\right)^{h-2} + \left(\frac{3}{2}\right)^{h-1} + 1 \\ &= \left(\frac{3}{2}\right)^{h-2}\left(1 + \frac{3}{2}\right) + 1 \\ &> \left(\frac{3}{2}\right)^{h-2} \cdot \frac{9}{4} \\ &= \left(\frac{3}{2}\right)^{h}, \end{aligned}$$

as desired. Thus, we have proven (4.5).

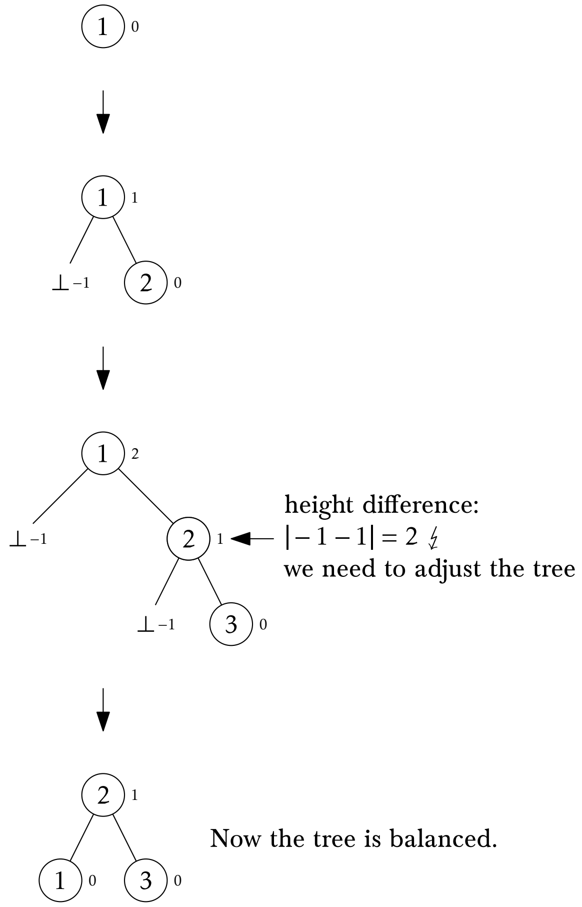
Now, consider an AVL-tree T with n nodes, and suppose that T has height h. Then, by the definition of F_h, we have $n \geq F_h$. By (4.5), we also have $F_h \geq (3/2)^h$. Thus, by combining the inequalities, we derive $n \geq (3/2)^h$, or, equivalently, $h \leq \log_{3/2} n \approx 1.71 \log_2 n = O(\log n)$, as claimed. □

Rotations on AVL-trees Now that we have seen that the AVL-tree property is useful to ensure that the height of tree is $O(\log n)$, our goal is to extend the binary search tree operations such that the AVL-tree property is maintained.

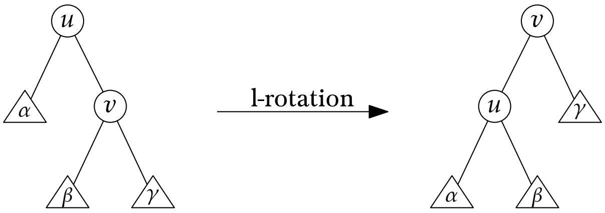
For this, we augment our binary search tree T such that in each node v of T, we store the height of the subtree that is rooted at v. During an operation (insertion/deletion), we can update this information, and we can use it to check if the AVL-tree property is violated. If this happens, we need to restructure the tree in order to restore the AVL-tree property. All of this needs to be done efficiently, since otherwise AVL-trees do not yield an advantage.

Let us first look at the case of insertions. If we perform an insertion, the height of a subtree an AVL-tree T can only *increase*. We can say more: an insertion in a binary search tree always adds a new leaf w to the tree, and the only nodes v whose subtrees can have a larger height are those nodes that lie on the path π from the root of T to w (all other subtrees remain unchanged). Even more, the AVL-tree property can only be violated at a parent node of a node whose subtree height has changed. Since the path π goes from w to the root of T, all these parent nodes automatically also lie on the path π. Thus, it is enough to consider only the nodes along π, and to do something about the *unbalanced* nodes, i.e., those nodes where the AVL-tree property is violated.

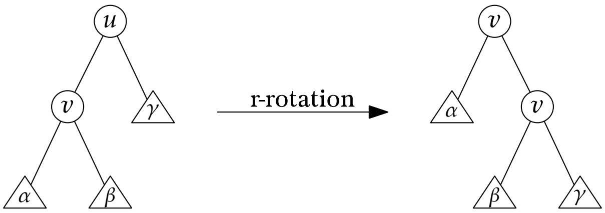
inserting $1, 2, 3$ into an empty tree:



The operations that we use in order to fix the unbalanced nodes are called *rotations*. There are several kinds of rotations. First, we describe the *left rotation* (ℓ-rotation): let u be a node and v be the right child of v, and let T_α be the left subtree of u and T_β and T_γ the left and right subtree of v. In a left rotation on u, we change the subtree at u such that v becomes the root, u becomes the left child of v, T_α and T_β become the left and right subtree of u, and T_γ becomes the right subtree of v. An ℓ-rotation is *local* (it can be performed by updating only a constant number of edges in the tree that are close together), and it preserves the binary search tree property.

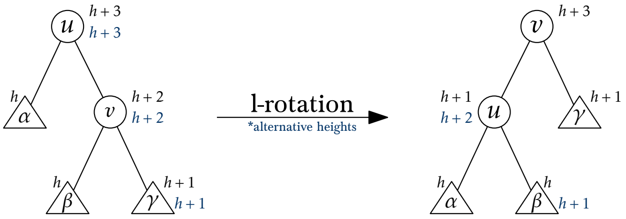


Symmetrically, there is also the *right rotation* (r-rotation): let u be a node and v be the left child of v, and let T_α and T_β be the left and the right subtree of v , andT_γ the right subtree of u. In a right rotation on u, we change the subtree at u such that v becomes the root, u becomes the right child of v, T_α becomse the left subtree of v, and T_β and T_γ become the left and right subtree of u. Again, the r-rotation is local and preserves the binary search tree property.



Now, we can examine how an (ℓ- or r-)rotation affects the heights of the subtrees that are involved. First, we see that the subtrees T_α, T_β, and T_γ are not changed, so their heights remain the same. The subtrees of interest are those that are rooted at u and at v.

Let us consider an ℓ-rotation, and suppose that the node u is unbalanced at such a way that T_α has height h and such that v has height $h+2$, for some h. Let us further assume that v is balanced (fulfills the AVL-property). Then, one of the subtrees T_β and T_γ must have height $h+1$, and the other subtree must have height h or $h+1$. The height at u is $h+3$. Now, after performing an ℓ-rotation, we see that we obtain a balanced situation in all cases except for the case where T_β has height $h+1$ and T_γ has height h. A symmetric situation occurs for an r rotation.



Thus, it still remains to find a solution for the final case. We look again at the situation for an ℓ-rotation. The difficult case is that T_α has height h, T_β has height $h+1$, and T_γ has height h. In this case, a simple ℓ-rotation on u will not lead to a balanced situation. However, we already know that if we were lucky and T_β has height h and T_γ has height $h+1$, then an ℓ-rotation on u would work. Thus, the idea is to create this situation by performing *first* an r-rotation on v, and *second* an ℓ-rotation of u. Such an operation is called an *$r\ell$-rotation*. By examining the cases we see that an ℓr-rotation resolves our final difficult case. Symmetrically, there is also an $r\ell$-rotation that resolves the difficult case for the ℓ-rotation.

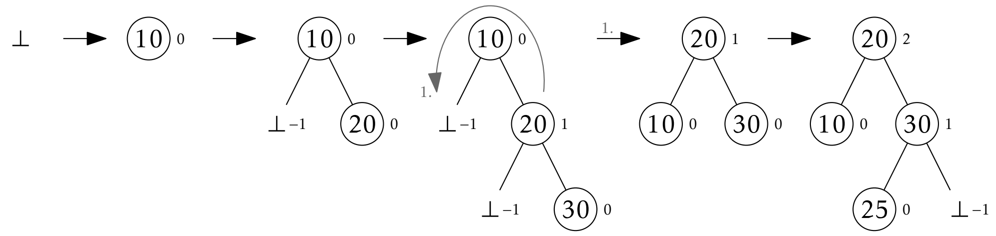
Collectively, ℓ-rotations and r-rotations are also called *simple* rotations, and $r\ell$-rotations and ℓr-rotations are also called *double* rotations.

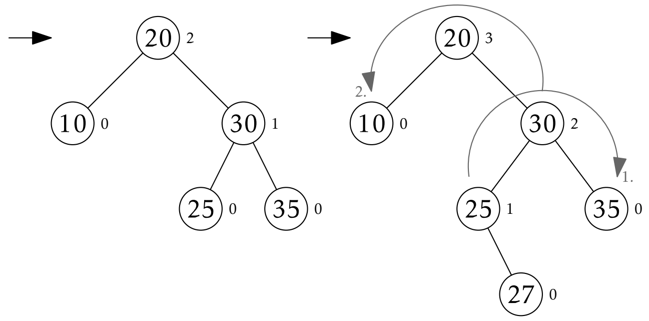
Insertion Now, the insertion algorithm for an AVL-tree is as follows: we perform the usual insertion in a binary search tree, adding a new leaf to the tree. Then, we go up along the path from this leaf to the root, and whenever we encounter an unbalanced node, we apply the appropriate (ℓ-, r-, $r\ell$-, or ℓr-)rotation to rebalance the subtree at this node. In such a way, whenever we perform a rotation, we know that all nodes in the subtree (except for the root) are balanced. Since all the unbalanced nodes must lie on the path to the root, and since rebalancing does not increase the height of a subtree, this procedure restores the AVL-tree property. Furthermore, the running time for the insertion is $O(\log n)$, because it consists of a single walk down and up the tree, with a constant number of operations being performed at each level of the tree that is visited.

Deletion The deletion procedure is similar: we perform the usual deletion in a binary search tree, and then we go back up to the root, from the location of the node that was deleted, to the root. Along this path, we perform the appropriate rotations to rebalance the nodes that we encounter. Now, it may happen that a rotation decreases the height of subtree. However, if this happens, the height of the subtree before the rotation must have been the same as the height of the subtree before the deletion, so it cannot happen that we ever create a situation where the heights of two subtrees of a given node differ by more than 2. Since the deletion operation consists of a single pass down and up the tree, performing $O(1)$ operations at each level, the running time is $O(\log n)$.

A closer examination of the algorithms shows that during an insertion, we perform at most one (single or double) rotation, whereas a deletion may perform many rotations, all the way up the tree.

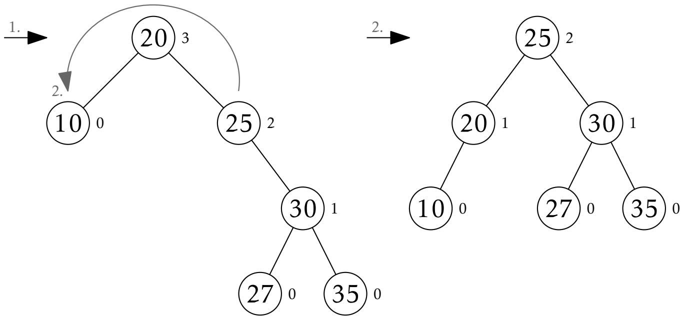
inserting: 10,20,30,25,35,27



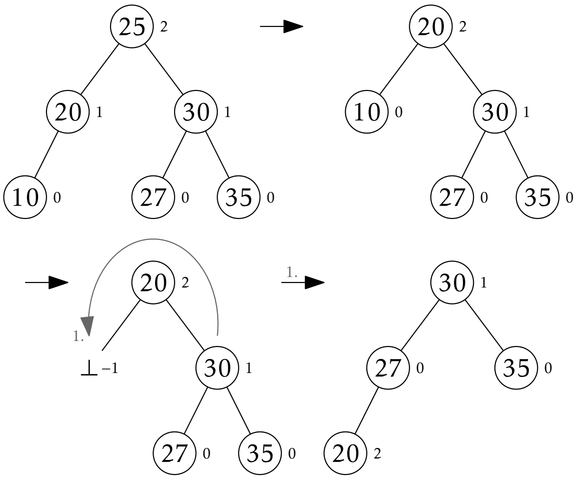


$h_r - h_l = 1 - 0 = 1 \not\leq 0$

$\Rightarrow$ right-left-rotation



deleting: 25,10



Conclusion. To conclude, AVL-trees are a useful and efficient implementation of the ADT ordered dictionary. They achieve $O(\log n)$ worst-case running time and have a memory-efficient representation. On the other hand, the implementation of an AVL-tree can be cumbersome (the rebalancing operations require a relatively long case distinction, with a lot of symmetric cases that lead easily to copy-paste errors), and a deletion operation may require a lot of rotations.

Next, we will see a simpler implementation of the ADT ordered dictionary that does not use search trees, but that needs randomness to be efficient.

Bonus remark (for the mathematically inclined reader). By solving the recurrence relation (4.2, 4.3, 4.4) exactly, we can derive the precise result that $h \le \log_\varphi(n+2) \approx 1.44 \log(n+2)$, where $\varphi = (1+\sqrt{5})/2$ is the golden ratio.

Let us see the proof. For this, we use the technique of *generating functions*. We represent the sequence F_h, $h \ge 0$ as a function, and we use simple algebraic methods, to obtain an explicit representation of the elements of the sequence. Thus, let us write

$$F(x) = \sum_{h=0}^{\infty} F_h x^h.$$

Now, we can compute

$$\begin{aligned}
F(x) &= 1 + 2x + \sum_{h=2}^{\infty} F_h x^h && \text{(by (4.2, 4.3))}\\
&= 1 + 2x + \sum_{h=2}^{\infty} (F_{h-2} + F_{h-1} + 1) x^h && \text{(by (4.4))}\\
&= 1 + 2x + \sum_{h=2}^{\infty} F_{h-2} x^h + \sum_{h=2}^{\infty} F_{h-1} x^h + \sum_{h=2}^{\infty} x^h && \text{(splitting the summands)}\\
&= 1 + 2x + x^2 \left(\sum_{h=0}^{\infty} F_h x^h \right) + x \left(\sum_{h=1}^{\infty} F_h x^h \right) + \left(\sum_{h=0}^{\infty} x^h \right) - 1 - x && \text{(shifting the index)}\\
&= 1 + 2x + x^2 F(x) + x(F(x) - 1) + \left(\sum_{h=0}^{\infty} x^h \right) - 1 - x && \text{(definition of } F(x)\text{)}\\
&= x^2 F(x) + x F(x) + \frac{1}{1-x}. && \text{(simplify, geometric series)}
\end{aligned}$$

Solving for $F(x)$, we obtain the following functional equation

$$F(x) = \frac{1}{(x^2 + x - 1)(x - 1)}.$$

Now, write $\varphi = (1+\sqrt{5})/2$ and $\widehat{\varphi} = (1-\sqrt{5})/2$. A quick calculation shows that the numbers φ and $\widehat{\varphi}$ have the following properties:

(i) $\varphi + \widehat{\varphi} = 1$; (ii) $\varphi - \widehat{\varphi} = \sqrt{5}$; (iii) $\varphi \cdot \widehat{\varphi} = -1$; (iv) $\varphi^2 = \varphi + 1$; and (v) $\widehat{\varphi}^2 = \widehat{\varphi} + 1$.

From (i, iii), it follows that $(x+\widehat{\varphi})(x+\varphi) = x^2 + x - 1$. Thus,:

$$F(x) = \frac{1}{(x+\widehat{\varphi})(x+\varphi)(x-1)}.$$

Now, the goal is to bring the functional equation for $F(x)$ into an explicit form that yields a formula for the elements F_h. For this, we would like to write the functional equation in terms of the geometric series $1/(1-ax) = \sum_{h=0}^{\infty} a^h x^h$. To achieve this, we use the method of decomposing the functional equation into *partial fractions*. WE write

$$F(x) = \frac{1}{(x+\widehat{\varphi})(x+\varphi)(x-1)} = \frac{\alpha}{x+\widehat{\varphi}} + \frac{\beta}{x+\varphi} + \frac{\gamma}{x-1},$$

where α, β, γ need to be determined. If we multiply this with $x+\widehat{\varphi}$ and set $x = -\widehat{\varphi}$, we obtain, due to (ii, v, iii),

$$\alpha = \frac{1}{(-\widehat{\varphi}+\varphi)(-\widehat{\varphi}-1)} = \frac{1}{\sqrt{5}(-\widehat{\varphi}^2)} = -\frac{\varphi^2}{\sqrt{5}}.$$

Similarly, using (ii, iv, iii), we see that

$$\beta = \frac{1}{(-\varphi+\widehat{\varphi})(-\varphi-1)} = \frac{1}{-\sqrt{5}(-\varphi^2)} = \frac{\widehat{\varphi}^2}{\sqrt{5}},$$

and using (iv, v, iii), we see that

$$\gamma = \frac{1}{(1+\varphi)(1+\widehat{\varphi})} = \frac{1}{(\varphi\widehat{\varphi})^2} = 1.$$

Now, we can derive the partial fraction decomposition for $F(x)$, and we use it as follows:

$$\begin{aligned}
F(x) &= -\frac{\varphi^2}{\sqrt{5}} \cdot \frac{1}{x+\widehat{\varphi}} + \frac{\widehat{\varphi}^2}{\sqrt{5}} \cdot \frac{1}{x+\varphi} + \frac{1}{x-1} \\
&= -\frac{\varphi^2}{\sqrt{5}} \cdot \frac{1}{\widehat{\varphi}(1+x/\widehat{\varphi})} + \frac{\widehat{\varphi}^2}{\sqrt{5}} \cdot \frac{1}{\varphi(1+x/\varphi)} - \frac{1}{1-x} && \text{(rearranging)} \\
&= \frac{\varphi^2}{\sqrt{5}} \cdot \frac{\varphi}{1-\varphi x} - \frac{\widehat{\varphi}^2}{\sqrt{5}} \cdot \frac{\widehat{\varphi}}{1-\widehat{\varphi}x} - \frac{1}{1-x} && \text{(by (iii))} \\
&= \frac{\varphi^3}{\sqrt{5}} \cdot \sum_{h=0}^{\infty} \varphi^h x^h - \frac{\widehat{\varphi}^3}{\sqrt{5}} \cdot \sum_{h=0}^{\infty} \widehat{\varphi}^h x^h - \sum_{h=0}^{\infty} x^h - \frac{1}{1-x} && \text{(geometric series)} \\
&= \sum_{h=0}^{\infty} \left(\frac{\varphi^{h+3} - \widehat{\varphi}^{h+3}}{\sqrt{5}} - 1 \right) x^h. && \text{(simplyfying)}
\end{aligned}$$

By comparing the coefficients, we obtain the precise formula

$$F_h = \frac{1}{\sqrt{5}} \left(\varphi^{h+3} - \widehat{\varphi}^{h+3} \right) - 1,$$

for all $h \geq 0$. Thus, we have

$$n \geq F_h \geq \frac{1}{\sqrt{5}}\varphi^{h+3} - 2$$

and hence

$$h \leq \log_\varphi(n+2) + \log_\varphi(\sqrt{5}) - 3,$$

as claimed.

KAPITEL 5

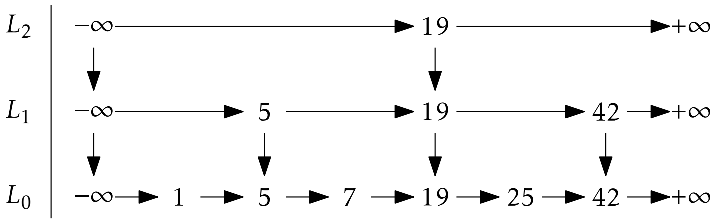
Skip Lists

Motivation. The main goal of skip lists is to implement the ADT ordered dictionary with the help of a linked list, as we saw them in *Konzepte der Programmierung*. In *Konzepted der Programmierung*, we saw an implementation of ordered dictionaries using an unsorted and a sorted linked list. The implementation is relatively straightforward, but the performance is not very good: to access an element of the list, we may need to traverse almost the whole list, taking $\Omega(n)$ steps in the worst case.

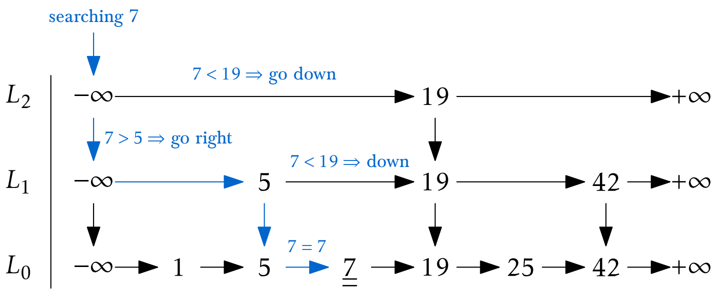
To improve this, we take an idea from public transportation. On some train routes in Germany, there are two kinds of trains: the *express train* that services only the "important" stops (e.g., the EC81 that stops only at München Hauptbahnhof, München Ostbahnhof, Rosenheim, and Kufstein) and the *local train* that stops everyhwere (e.g., the RB 54 that stops at München Hauptbahnhof, München Ostbahnhof, Grafing, Aßling, Ostermünchen, Großkarolinenfeld and Rosenheim, Raubling, Brannenburg, Flintsbach, Oberaudorf, Kiefersfelden and Kufstein). Thus, ignoring time-table issues, and efficient way to get from München Hauptbahnhof to Brannenburg would be to take the EC81 from München Hauptbahnhof to Rosenheim, and then the RB54 from Rosenheim to Brannenburg.

Thus, in public transportation, we have an option of *skipping* certain stops by taking a different line, expediting the travel to our desired destination. Can we find an analogue of this idea in the realm of linked lists?

Express list structure. In principle, this can be done in a relatively straightforward manner: we begin with a sorted linked list L_0 that contains our entries, and we add two *pseudo-nodes*: one at the beginning, storing the key $-\infty$ (and no value), and one at the end, storing the key ∞ (and no value). Then, we create the *first express list* L_1 that contains the pseudo-nodes and a node for every second key from L_0 (the complete entries that contain also the values are stored only in L_0), a *second express list* L_2 that contains the pseudo-nodes and a node for every second key from L_1 (and hence for every fourth key from L_0), a *third express list* L_3 that contains the pseudo-nodes and a node for every second key from L_2 (and hence for every eighth key from L_0), and so on. We stop when our current express list consists only of two pseudo-nodes. Between consecutive lists, we create *down-pointers* from each node in the higher list to the corresponding node (with the same key) in the lower list. These down-pointers allow us to change from one express list to the next lower one.



Now, the look-up procedure for a key k is as follows: we start in the highest express list from the left pseudo-node, and we walk right until the last node whose key is smaller than or equal to k. Then, we follow the down pointer to the corresponding node in the next express list. From there, we walk right until the last node whose key is smaller than or equal to k, and so on, until we reach the node with the entry for k in the lowest list L_0, or determine that L_0 does not contain k (pseudo-code is given below).



We call this structure the *express list structure*. To analyze it, suppose that we would like to store n entries. First, we analyze the number of express lists. The first express list L_1 contains every second key from L_0. Thus L_1 contains $\lfloor n/2 \rfloor$ keys. The second express list contains every fourth key from L_0, i.e., $\lfloor n/4 \rfloor$ keys. In general, the i-th express list contains every 2^i-th key from L_0, i.e., $\lfloor n/2^i \rfloor$ keys. The procedure stops for the first i where $\lfloor n/2^i \rfloor$ is 0, i.e., for the first i where $n/2^i$ is less than 1. This is exactly $i_{\max} = \lfloor \log n \rfloor + 1$. Thus, the total number of lists is $\Theta(\log n)$. During the look-up procedure, we visit a constant number of nodes in each list (we make at most one step to the right), to the time for a look-up is $\Theta(\log n)$.

Finally, we consider the total number of nodes in our lists. Let $|L_i|$ denote the total number of nodes in the list L_i. Then, the total number of nodes is

$$\begin{aligned}\sum_{i=0}^{i_{\max}} |L_i| &= \sum_{i=0}^{i_{\max}} (2 + \lfloor n/2^i \rfloor) && \text{(two pseudo-nodes and the keys)}\\ &\le 2 \cdot (i_{\max} + 1) + \sum_{i=0}^{i_{\max}} \frac{n}{2^i} && \text{(splitting the sum, bounding } \lfloor \cdot \rfloor \text{ from above)}\end{aligned}$$

$$\le O(\log n) + n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} \qquad \text{(bound on } i_{\max}\text{, bound the sum from above)}$$

$$= O(\log n) + n \cdot 2 \qquad \text{(geometric sum)}$$

$$= O(n). \qquad \text{(the linear term dominates)}$$

Thus, all the lists combined are larger only by a constant factor than the initial list L_0.

This express list structure looks very much like a perfect binary search tree, and just like a perfect binary search tree, it is difficult to maintain under insertions and deletions: adding or deleting a single entry can lead to significant changes in the overall structure, because we might need to rebuild all the express lists completely.

Skip lists. However, there is a way to maintain the express list structure *approximately*, by using *randomness*. In the express list structure, we insist that each express list L_i contains exactly every second key from the list L_{i-1} below it. Instead, suppose we have a fair coin that comes up heads with probability 1/2 and tails with probability 1/2. Then, we can derive L_i from L_{i-1} as follows: we create two pseudo-nodes that will be the first and the last node of L_i. Then, we go through the keys in L_{i-1} from left to right, and for each key k we flip our coin. If the coin comes up heads, we create a node for $k$in L_i. If the coin comes up tails, we skip k. Then, we *expect* that L_i contains every second key from L_{i-1}, but there may be some errors due to the random process. Below, we will see that this will not affect the (expected) running time.

The main insight now is that due to the independence of the coin flips, this procedure is completely equivalent to the following procedure: for each key k in L_0, we flip our coin until it comes up tails for the first time. We count the number of heads j, that we see before this, and we add key to the express lists $L_1, \dots, L_j$ (if $j = 0$, then k is stored only in L_0). Even more, if does not matter in which order we perform this experiment for each key: if we add a new key k to L_0 and perform the coin flipping experiment for k, we get the same distribution on the lists as if k hat been there all along. If we delete a key k from L_0 and all the express lists that lie above it, we get the same distribution as if k hat never been there. Thus, randomness gives us a way to treat each entry individually and to still get a structure that is "balanced" overall (at least in some expected sense, to be made precise below.

Thus, the main strategy is now clear: we implement a data structure that approximates the express list structure, using randomness. Lookups work exactly as before, with the only difference that we may look at more than two nodes in a single express list. To delete an entry, we perform a search for the key and then remove the corresponding nodes from all lists in which the key occurs. The perform an insertion, we perform a lookup to determine the insertion locations in all the lists. Then, we keep flipping a fair coin until the first occurrence of tails, and we insert the key into as many lists as we have used coin flips (we insert always in the bottom list, and in as many express lists as the number of heads).

Next, we look at the details.

Implementation Details. In memory, a skip list consists of several linked lists. Each linked lists consists of nodes. A node `n` is represented by an object that contains the following fields:

- `k`, `v`: the key and value stored in this node,
- `pred`, `succ`: the predecessor and successor node of `n` (in the same linked list, horizontally);
- `down`: the corresponding node to `n` in the list below (vertically). The `down`-pointer is NULL in the lowest list L_0.

Each list has two *pseudo-nodes*, one at the beginning and one at the end, storing the pseudo-keys $-\infty$ and $+\infty$. Initially, the skip list consists of a single linked list (the lowest list), that stores only the two pseudo-nodes.

Obtaining a list of predecessors. First, we describe an internal helper operation that determines for a given key k the list of predecessor nodes for k in all the constituent lists lists of the skip list. More precisely, the function `search`(k) produces a stack `Q` that contains the predecessor nodes for the key k, (or, if it exists, the nodes that contain the key k from all lists that are part of the skip list (ordered from the node in the lowest list to the node in the highest list).

The operation starts in the highest list and walks to the right until the last node whose key is smaller or equal to k. Then, it stores this node in the stack, and follows the `down`-pointer to the next list. This continues until the lowest list.

```
search(k)
  Q <- new Stack
  n <- -INFTY pseudo-node of the highest list
  do
    while n.next.k <= k do
      n <- n.next
    Q.push(n)
    n <- n.down
  while n != NULL
  return Q
```

Lookup To perform a lookup for a key k, we use the operation `search`(k). W consider the top-most node in the resulting stack `Q` and check if it stores the key k. If so, we can return the corresponding value. If not, we know that they key k does not occur.

```
get(k)
  Q <- search(k)
  n <- Q.pop
  if n.k == k then
    return n.v
  else
    throw NoSuchElementException
```

Insertion To perform an insertion, we first look for the key k, using `search`(k). If the key exists, we just update the corresponding value. If not, we use the stack of predecessor nodes that is returned by `search` to insert k into the express lists. The number of express lists into which we insert is determined by flipping a coin, as described above. It may happen that the coin shows heads more often than the number of lists in our current structure. In this case, we create more express lists that are added at the top of the list.

```
put(k, v)
  Q <- search(k)
  n <- Q.pop
  if n.k == k then
    n.v <- v
    return
  insert a new node for (k, v) after n
  while coinFlip == heads do
    if Q.isEmpty then
      create a new list with -INFTY, a node for k, +INFTY
      create down links
    else
      n <- Q.pop
      insert a new node for k after n, add a down link
```

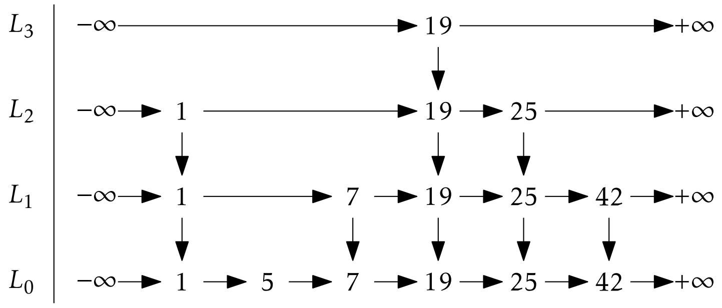
Deletion. To perform a deletion, we first look for the key k, using `search`(k). If the key does not exist, we raise an exception. Otherwise, we use the stack of nodes that is returned by `search` to remove the node for k from the express lists. It may happen that some express lists at the top of the structure become empty (consisting only of the pseudo-nodes). If so, they are removed.

```
remove(k)
  Q <- search(k)
  n <- Q.pop
  if n.k != k then
    throw NoSuchElementException
  while n != NULL and n.k == k do
    remove n from the list
    if !Q.isEmpty do
      n <- Q.pop
    else
      n <-NULL
  remove pseudonodes for empty express lists, if necessary
```

The operations `pred`(k), `succ`(k), `min` and `max` are left as an exercise.

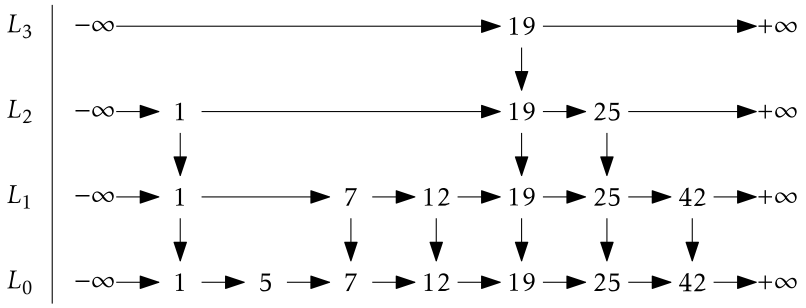
$1_{1,1,1,0}$ 5_0 $7_{1,1,0}$ $19_{1,1,1,1,0}$ $25_{1,1,1,0}$ $42_{1,0}$

indices indicate the coin results

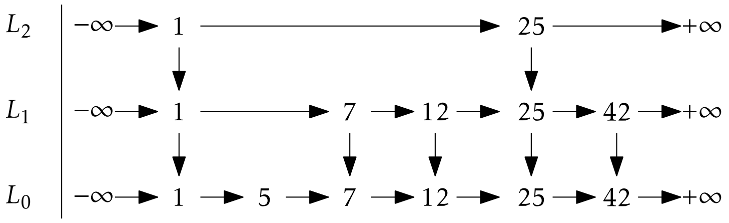


adding 12 to the list, we need to flip a coin again

$12_{1,1,0}$



removing 19, just delete the column and this time we can also delete L_3



Analysis . Suppose that we have a skip list L that stores n entries. Since we use randomness when maintaining L, the exact structure of L, and hence the performance guarantees for L (i.e., running time of the operations, space requirement) are *random variables* whose exact value depends on the random choices of the insertion algorithm. As such, we need to rely on notions from probability theory to understand these random variables. In probability theory, we see many different ways to analyze a random variable, e.g., expectation, variance, tail bounds, etc. We will focus on the most basic way: computing the *expectation* of a random variable.

Thus, our goal will be to computed the *worst-case expected* performance of a skip list with n entries. That is, for every possible set of n entries, we analyze the expected value of the running time of the operations and of the space requirement, and we take the maximum. This is a *worst-case* guarantee, because we take the maximum over all possible inputs, and it is an *expected* guarantee, because for every input we estimate the expected value.

First, we state that for any set of n entries, the expected number of key-nodes that are stored in the skip list is $O(n)$. Furthermore, the expected number of express lists is $O(\log n)$. The proof of these two facts are left as an exercise.

Here, we will analyze the expected number of steps that are needed to locate a key k in the skip list.

Satz 5.1. *Let $S \subseteq K \times V$ a set of n entries, and let Ls be a skip list that stores S. Let $k_0 \in K$ be a key. Then, the expected number of steps (to the right and down) to locate k_0 in L is $O(\log n)$.*

Beweis. For $i = 0, 1, \dots$, let L_i be the random variable that counts the number of steps that the search performs in L_i. Note that we have infinitely many random variables, because we cannot know in advance how many lists there are in L. If a list L_i does not appear in L, we set L_i to 0.

Now, the total number of steps is $T = \sum_{i=0}^{\infty} T_i$. To compute the expected value of T, we can use linearity of expectation, as follows:

$$\mathbf{E}[T] = \mathbf{E}\left[\sum_{i=0}^{\infty} T_i\right] = \sum_{i=0}^{\infty} \mathbf{E}[T_i].$$

We split the sum into two parts:

$$\sum_{i=0}^{\infty} \mathbf{E}[T_i] = \sum_{i=0}^{\lceil \log n \rceil - 1} \mathbf{E}[T_i] + \sum_{i=\lceil \log n \rceil}^{\infty} \mathbf{E}[T_i].$$

We analyze the two sums separately. For the first sum, we consider the first $\lceil \log n \rceil$ lists of L. We argue that in each such list, we expected number of steps will be $O(1)$. This gives a total of $O(\log n)$ steps for the first sum. For the second sum, we will argue that it is increasingly unlikely that there is a list in L that has level $i \geq \lceil \log n \rceil$. Thus, even if we bound the expected number of steps in such a list by its length, this will result in only a negligible contribution to the expected value.

We begin with the second sum. Fix an $i \geq \lceil \log n \rceil$. Let S_i be the number of keys from S that are stored in L_i (S_i is 0 if L_i is not present). Then, the number of steps in L_i is at most $2S_i$ (we take one step to the right for each key that is smaller than k, plus one step down, so at most 2 steps for each key from S that is present in S_i. Thus, using linearity of expectation, we have

$$\mathbf{E}[T_i] \leq \mathbf{E}[2 \cdot S_i]2 \cdot \mathbf{E}[S_i],$$

and we must compute $\mathbf{E}[S_i]$. For this, we again use linearity of expectation. For each key k' that appears in S, define an *indicator random variable* X_k to be 1, if k' appears in L_i, and 0, otherwise. Then, we have $S_i = \sum_{k \in S} X_k$, and, by linearity of expectation,

$$\mathbf{E}[S_i] = \mathbf{E}\left[\sum_{k \in S} X_k\right] = \sum_{k \in S} \mathbf{E}[X_k].$$

Now, using the definition of an expected value and the definition of the random variable X_k, we have

$$\mathbf{E}[X_k] = 0 \cdot \Pr[X_k = 0] + 1 \cdot \Pr[X_k = 1] = \Pr[X_k = 1] = \Pr[k \text{ appears in } L_i].$$

The probability that k appears in L_i is exactly the probability of the even that then inserting k, we our coin came up at least L_i times heads. This probability is exactly $1/2^i$, because this is the probability that when flipping a coin i times, we see i heads. Thus, we have

$$\mathbf{E}[X_k] = \frac{1}{2^i},$$

and going back to T_i, we get

$$\mathbf{E}[T_i] \leq 2 \cdot \mathbf{E}[S_i] = 2 \cdot \sum_{k \in S} \mathbf{E}[X_k] = 2 \cdot \sum_{k \in S} \frac{1}{2^i} = \frac{2n}{2^i},$$

since $|S| = n$. Hence, we can bound the second sum as

$$\sum_{i=\lceil \log n \rceil}^{\infty} \mathbf{E}[T_i] \leq \sum_{i=\lceil \log n \rceil}^{\infty} \frac{2n}{2^i} = \frac{2n}{2^{\lceil \log n \rceil}} \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} \leq \frac{2n}{n} \cdot 2 = 4,$$

since $2^{\lceil \log n \rceil} \geq 2^{\log n} = n$, and $\sum_{i=0}^{\infty} 1/2^i = 2$, by a geometric series.

Next, we bound the first sum. For this, fix an i between 0 and $\lceil \log n \rceil - 1$. Our goal is to bound the probability $\Pr[T_i \geq j]$ that T_i, the number of steps in L_i, is at least j, for every $j \geq 1$. Then, we can estimate the expected value of T_i using the well known formula

$$\mathbf{E}[T_i] = \sum_{j=1}^{\infty} \Pr[T_i \geq j]$$

which holds for every random variable whose range are the natural numbers.

First, suppose that $i > 0$ (the situation for T_0 is only slightly different, see below). The search path takes always at least one step in L_i, namely the down step from the last node. Thus, we have $\Pr[T_i \geq 1] = 1$. Now, fix an integer $j \geq 2$, and suppose that the search path takes at least j steps in L_i. Consider the last $j-1$ nodes of the search path in L_i. Since the search path enters each of these nodes by following at step in L_i, it must be the case that none of these $j-1$ nodes contains a key that is also present in L_{i+1} (because otherwise the search path would enter such a node through a step from L_{i+1}). The probability that this happens is at most $1/2^{j-1}$, since the coin flips in the insertion procedures are independent.[1] Thus, for all $j \geq 1$, we have $\Pr[T_i \geq j] \leq 1/2^{j-1}$.

For T_0, the situation is similar, except that the final down step does not exist. Thus, we even get $\Pr[T_0 \geq j] \leq 1/2^j$, for all $j \geq 1$, and since $1/2^j \leq/ 2^{j-1}$, we can treat T_0 in the same was as the other T_i.

Thus, we get for all $i \geq 0$.

$$\mathbf{E}[T_i] = \sum_{j=1}^{\infty} \Pr[T_i \geq j] \leq \sum_{j=1}^{\infty} \frac{1}{2^{j-1}} = \sum_{j=0}^{\infty} \frac{1}{2^j} = 2$$

using the geometric series formula.

In total, we get

$$\mathbf{E}[T] = \sum_{i=0}^{\lceil \log n \rceil - 1} \mathbf{E}[T_i] + \sum_{i=\lceil \log n \rceil}^{\infty} \mathbf{E}[T_i] \leq \sum_{i=0}^{\lceil \log n \rceil - 1} 2 + 4 = 2 \cdot \lceil \log n \rceil + 4 = O(\log n),$$

as claimed. □

To conclude, skip-lists have the advantage of a relatively simple implementation and a running time that is as good as AVL-trees. Possible disadvantages are that the running times are only in the expected sense (there is always a certain probability that everything can be very slow), and that we need additional space for the express lists, while, say, AVL-trees need only one node per entry.

In the next chapter, we will get back to data structures that are based on binary search trees, showing another way to relax the notion of a perfect binary tree to achieve an efficient implementation of ordered dictionaries.

[1] It may happen that this probability is 0, of the search path enters L_i through a node that is too close to the left pseudo-node.

KAPITEL 6

Splay Trees

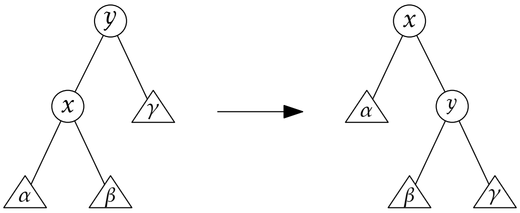
Splay trees are a *self-organizing* data structure that tries to adapt automatically to a sequence of operations. In AVL-trees, restructuring is rare and occurs only in order to enforce a fixed balancing condition on the tree structure. In splay trees, the tree is reorganized after *every* operation, not just after `put` and `remove`, but also after `get`. The main idea is that if an operation accesses a node x in a search tree T, then it may be likely that x, or a node that is close to x, will be accessed not much later. Thus, it can be helpful to reorganize T such that x becomes the new root of T, because then subsequent operations can access x very quickly.

In order to make x the root, we can use the search path for x in the tree, since we have computed this path already when accessing x. A natural idea would be to use the simple (ℓ- and r-) rotations that we know from AVL-trees, and to apply them repeatedly to the nodes along the search path (from the parent of x to the root). In this way, x moves up the tree and becomes the new root. This is called the *move-to-root* heuristic.

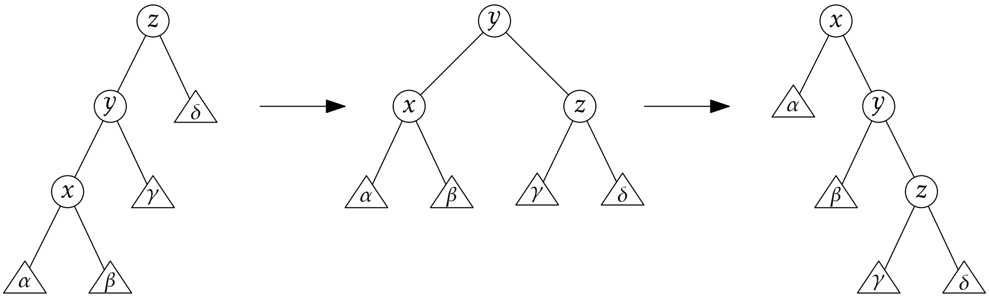
However, already simple examples show that this heuristic does not result in a good performance. The main innovation of Sleator and Tarjan, who presented the splay tree method in 1985, was to replace this simple move-to-root heuristic by a slightly more elaborate *splay*-operation. This operation works amazingly well and leads to probably good guarantees.

The splay-operation. Let T be a binary search tree with root r, and let x be a node in T. The splay operation, `splay`(x), applies a sequence of local operations along the search path from x to r in order to move x to the root of T. If x is the root of T, there is nothing to do. Otherwise, let y be the current parent of x. There are three possible cases, depending on the situation of y and x:

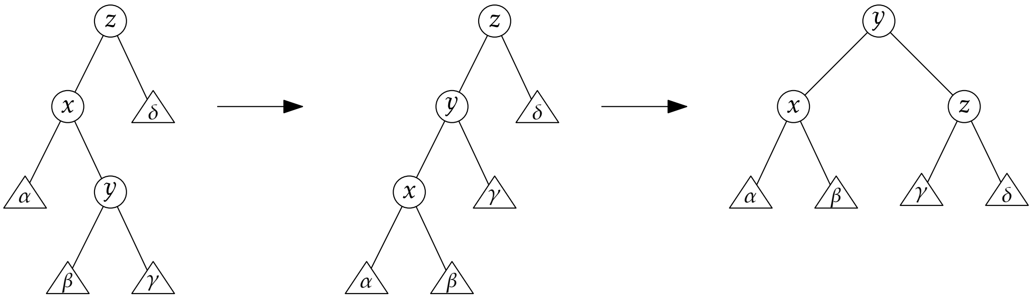
- **zig-case**: If y is the root of T, we apply an appropriate simple (ℓ- or r)-rotation to y in order to make x the root of T. This case occurs only once and finishes the splay-operation.



- **zig-zig-case**: If y is not the root of T, let z be the parent of y (and the grandparent of x). If y and x are one the same side of their respective parents (i.e., either y is the left child of z and x is the left child of y, or y is the right child of z and x is the right child of y), the *zig-zig-case* applies: First, we perform an appropriate rotation on z, to make y the new root of the subtree of z, and then we rotate on y, to bring x to the root.



- **zig-zag-case**: Suppose again that y is not the root of T, and let z be the parent of y. If y and x are one different sides of their respective parents (i.e., either y is the left child of z and x is the right child of y, or y is the right child of z and x is the left child of y), the *zig-zag-case* applies: First, we perform an appropriate rotation on y, to make x the new root of the subtree of y, and then we rotate on z, to bring x to the root.



We note that the only place where `splay`(x) differs from the move-to-root heuristic lies in the zig-zig-case: in this case, move-to-root would first rotate on y and then on z. The main trick of splay trees is to reverse this order, making sure that not just x, but also the subtrees of x move up the tree. Those nodes that were moved down in a single step are moved up again in the next step(s), so splaying ensures that the nodes in the subtrees of x and along the search path to x move much closer to the root, while the other nodes only move down by a constant number of levels./

Splay tree operations. The operations on a splay tree work like in a binary search tree, with an additional splay step after each operation: in a `get`(k) operation, we follow the search path for the key k. If k is present in the tree, we perform a splay operation on the node that contains it. If k is not present, we perform a splay operation on the leaf-node that has the empty subtree where k should lie (i.e., the last node on the search path). For `put`(k,v), the situation is similar: after locating/creating the node x for k, we perform a splay operation on x. For a deletion, we splay the parent node of the node that was removed.

Analysis. Unlike AVL-trees, splay trees do not enforce a global invariant on the structure of the tree. This comes at the benefit that we do not need to store any additional information in the nodes (in AVL-trees, we must keep track of the heights of the subtrees).

However, throughout the lifetime of a splay tree, it can happen that the tree becomes very unbalanced. It may well be the case that at some point a node lies at level $\Omega(n)$, and that a single access and splay operation can take $\Omega(n)$ steps. However, these situations are very rare, and they can occur only after a long sequence of operations that were very efficient. Thus, even though it may happen that *in the worst-case*, splay trees are very slow in a single operation, *in the amortized sense* (that is, over a sequence of operations) splay trees behave very well. Let us mention here that the notion of *amortized analysis* of a data structure is a general concept that has wide applicability in the design and analysis of data structures. We will look at it more closely (and with precise definitions) later in this class.

Splay trees have been analyzed extensively, and there are many theoretical results that make the statements from the previous paragraph precise. Here, we state (but do not prove) two theorems about splay trees that give an idea of the kinds of results that are possible. They both consider only the case that we have a splay tree that contains a fixed set of n entries and that we perform `get`-operations to only these entries. The first theorem shows that (in the amortized sense) splay trees are never worse than AVL-trees:

Satz 6.1. *Let T be a splay tree that contains a set S of n entries, and suppose we perform an arbitrary sequence π of m `get`-operations to keys that appear in S. Then, the total running time for π is $O(m \log n + n \log n)$.*

We can say even more: if there is structure in the sequence π of operations, then splay trees can learn how to exploit this structure. For example, if some entries are accessed much more frequently than others, splay trees will adapt to make accessed to the more popular elements faster. This is made precise in the second theorem:

Satz 6.2. *Let T be a splay tree that contains a set S of n entries, and suppose we perform an arbitrary sequence π of m `get`-operations to keys that appear in S. For an entry $x \in S$, let m_x be the number of times that x is accessed in π. Suppose that $m_x \geq 1$, for all*

$x \in S$, i.e., that every entry is accessed at least once. Then, the total running time for π is $O(m + \sum_{x \in S} m_x \log \frac{m}{m_x})$.

The term m/m_x becomes smaller as m_x is larger. More precisely, we can think of m_x/m as the *probability* that a randomly chosen `get` in the access sequence π concerns the entry x. The larger this probability becomes (i.e., the more popular the entry x is), the faster the query becomes. There are many more interesting properties of splay trees, but these will be discussed in more advanced classes (together with the proofs for these two theorems).

To conclude, splay trees are an easy and efficient variant of binary search trees. They are easier to implement than AVL-trees and yield similar performance bounds. Even more, since splay trees adapt in response to the query sequence, they can learn from and adapt to structure that occurs in the input. The main drawback is that splay-trees provide only amortized guarantees: a single operation can be slow, but an average operation in any sequence of operations will be efficient.

In the next chapter, we will see yet another tree structure that yields good worst-case performance and that is optimized for very large data sets.

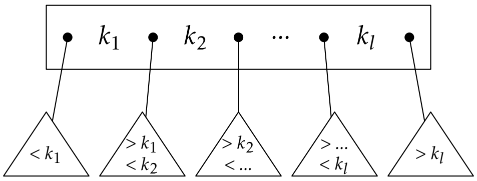
KAPITEL 7

(a,b)-Trees

We will now see a different way to approximate a perfect binary tree. Recall that in AVL-trees, we have relaxed the requirement that all leaves are at the same level, and we allow (controlled) differences in the heights. In (a,b)-trees, however, we insist that all leaves are at the same level, just as in a perfect tree. Instead, to ensure more flexibility, we relax the requirement that the tree must be *binary*. Now, a node can have more than two children, and the number of children can vary among different nodes.

We now describe the details. There are in fact many different kinds of (a,b)-trees, depending on the choice of two parameters: $a,b \in \mathbb{N}$. We require that $a \geq 2$ and that $b \geq 2a-1$. Once a and b are chosen, we can define the conditions for a valid (a,b)-tree:

- All leaves are on the same level.
- Different inner nodes can have a different number of children, according to the following rules: if the root is not a leaf, it has at least 2 children and at most b children. Every other inner node has at least a children and most b children.
- The entries are stored in the inner nodes and in the leaves. In an inner node, the number of entries is exactly the number of children minus one. In a leaf, the number of entries must be between $a-1$ and $b-1$, except if the leaf is also the root, in which case the number of entries must be between 1 and $b-1$.
- The children and entries in every inner node are ordered such that the *generalized search-tree property* is fulfilled for every inner node v of the tree: write v as $v = (w_1, k_1, w_2, k_2, \dots, w_\ell, k_{\ell-1}, w_\ell)$, where $w_1, w_2, \dots, w_\ell$ are the children of v, and $k_1, \dots, k_{\ell-1}$ are the keys that are stored in v. We assume that the keys and the children appear in this order in v. Then, we require that (i) $k_1 < k_2 < \cdots < k_{\ell-1}$; (ii) all keys in the subtree rooted at w_1 are smaller in k_1; (iii) for $i = 2, \dots, \ell-1$, all keys in the subtree rooted w_i are larger than k_{i-1} and smaller than k_i; and (iv) all keys in the subtree rooted at w_ℓ are larger than $k_{\ell-1}$.



Depending on the choice of a and b, there are different kinds of (a,b)-trees. Popular variants are $(2,3)$-trees and $(2,4)$-trees, which can be used as alternatives to AVL-trees. Another common variant is to choose a and b very large (e.g., a $(1024,2048)$-tree) and to store the tree in external memory. See below for more details.

First, we explain how the operations in an (a,b)-tree are implemented.

Lookup. A lookup operation for a key k in an (a,b)-tree T is a straightforward generalization of a lookup in a binary search tree. Starting at the root, suppose we are currently at a node v of T.

If v is an inner node of T, let $k_1,\dots,k_{\ell-1}$ be the keys in v, ordered from smallest to largest. If $k < k_1$, we go to the leftmost child w_1 of v. If $k > k_{\ell-1}$, we go to the rightmost child w_ℓ. Otherwise, we scan $k_1,\dots,k_\ell$, until we find the largest key k_{i-1} that is smaller than or equal to k. If $k_{i-1} = k$, we return the value that is stored with k. Otherwise, we go to the child w_i that is stored to the right of k_{i-1}.

If v is a leaf, we scan the entries in v for k. If k is present, we report the corresponding value. Otherwise, we report that the key k is not present. The pseudocode is as follows:

```
get(k)
  v <- root
  while v is not a leaf do
    write v as (w[1], k[1], w[2], k[2], ...., w[l-1], k[l-1], w[l])
    if k < k[1] then
      v <- w[1]
    else if k > k[l-1] then
      v <- w[l]
    else if there is an i such that k[i-1] < k < k[i] then
      v <- w[i]
    else // there is an i with k = k[i]
      return the value stored with k[i]
  // now v is a leaf
  if v contains k
    return the value stored with k
  else
    throw new NoSuchElementException
```

Update. To update an entry $(k,\texttt{val})$ in an (a,b)-tree T, we use the lookup-procedure to locate the entry for the key k. If this entry is found, we update the value and return. If not, the search ends in a leaf v where the key k should be located. We add the entry $(k,\texttt{val})$ to v.

Now, if the leaf v still has at most $b-1$ entries, there is nothing more to do. Otherwise, the leaf v contains exactly b entries (because before creating the entry, there (a,b)-tree invariant was fulfilled). We say that there is an *overflow* at v. In this case, we perform a *split*-operation that splits the entries in v into two new nodes v' and v'' and an entry $(k',\texttt{val})$, such that (i) all keys in v' are smaller than k'; (ii) all keys in v'' are larger than

k'; and (iii) v' and v'' each contain at least $a-1$ entries. The key k' and references to the new nodes v' and v'' are inserted into the parent node p of v, replacing the reference to v. This operation may in turn lead to and overflow in the parent node p, and we repeat the split operation on p. This process continues until we reach a parent node that does not have an overflow, or until we reach the root. In the latter case, we create a new root with a single entry, and T grows by one more level.

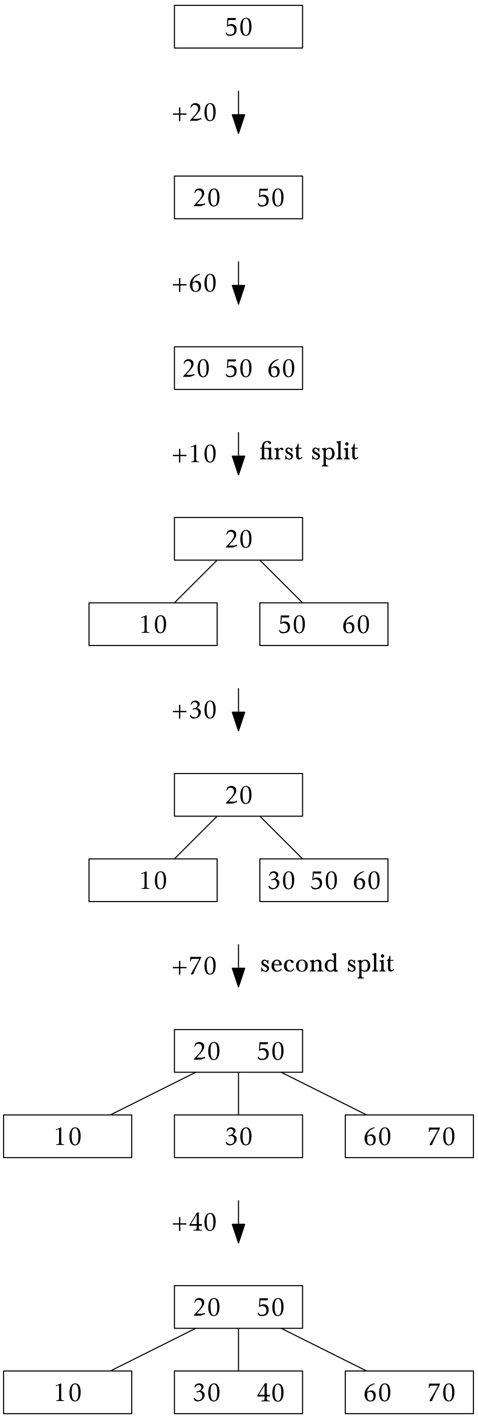
Note that our requirements on a and b ensure that a split is always possible: to obtain new nodes v' and v'' and an entry $(k', \texttt{val})$ that fulfill the requirements above, we need that v contains at least $(a-1)+(a-1)+1 = 2a-1$ entries. This is the case, because an overflow occurs only if v contains b entries, and we require that $b \geq 2a-1$. The pseudocode is as follows:

```
put(k, val)
  v <- root
  while v is not a leaf do
    write v as (w[1], k[1], w[2], k[2], ...., w[l-1], k[l-1], w[l])
    if k < k[1] then
      v <- w[1]
    else if k > k[l-1] then
      v <- w[l]
    else if there is an i such that k[i-1] < k < k[i] then
      v <- w[i]
    else // there is an i with k = k[i]
      update the entry for k with value val
      return
  // now v is a leaf
  if v contains k
    update the entry for k with value val
    return
  insert the entry (k, val) into v
  // now we need to split the nodes that overflow
  while v contains b keys do
    split v into v', k', v'' // see below
    if v is not the root then
      let p be the parent of v
      insert v',k',v'' into p // see below
      v <- p
    else
      create a new root (v', k', v'')
      return
```

The details of the split operation and of the insertion into the parent node can be implemented as follows: write $v = (w_1, k_1, \dots, w_{b-1}, k_b, w_{b+1})$ and set $m = \lfloor (b+1)/2 \rfloor$. Then, the new nodes are given by $v' = (w_1, k_1, \dots, w_m)$ and $v'' = (w_{m+1}, \dots, w_{b+1})$, and the middle key is $k' = k_m$. The parent node of v is updated as $p = (\ldots, k_{r-1}, v, k_r, \ldots) \rightarrow (..., k_{r-1}, v', k', v'', k_r, \ldots)$

$(2,3)$-tree

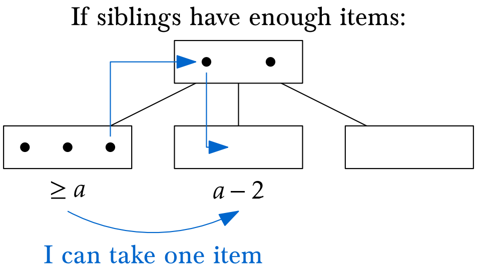
inserting: $50, 20, 60, 10, 30, 70, 40$

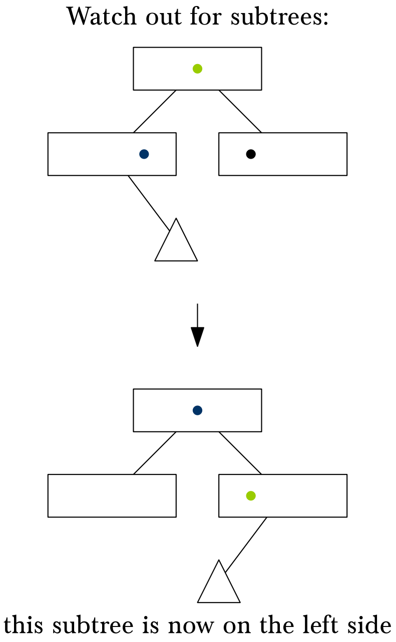


Deletion. To delete a key k from an (a,b)-tree T, we first locate k in T, using the lookup-method from above. If k lies in an inner node, we determine the predecessor k' of k by finding the largest key in the subtree to the left of k, we replace k by k', and we remove k' from the leaf that contains it. Note that k' must always be stored in a leaf (since k' is located by repeatedly following the rightmost child until reaching a leaf, starting in the subtree to the left of k). Note also that k' must always exist (because if k lies in an inner node, there is a subtree to the left of k, and this subtree contains k').

Thus, we can focus on the case that the key k must be deleted from a leaf node v of T. For this, we simply remove the entry for k from v. Afterwards, if v contains at least $a-1$ entries, there is nothing left to do. Otherwise, if v contains precisely $a-2$ entries, we must find another entry that we can add to v, in order to restore the (a,b)-tree invariant. We say that there is an *underflow* in v.

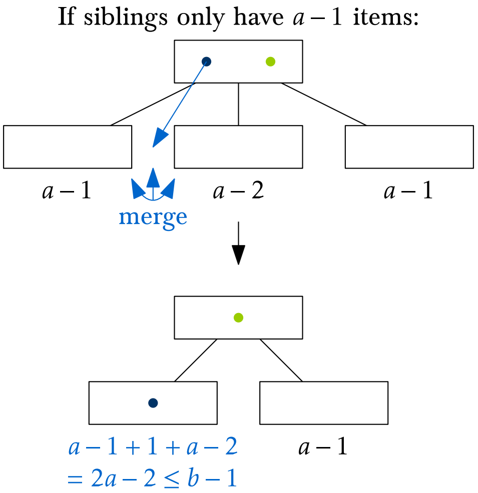
To fix the underflow, we first consider the *direct siblings* of v in T, i.e., the nodes that lie directly to the left or directly to the right of v in the parent node of v. The node v has at most two direct siblings. If one of the siblings contains at least a entries, we can use one of these entries to fix the underflow in v. This is called a *borrowing operation*.



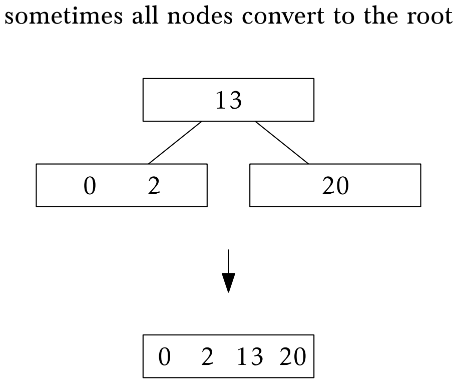


The details of this step are given below.

If each direct sibling of v contains only $a-1$ entries, we need to *join* v with a direct sibling. For concreteness, say that this is the right sibling v' of v. We take the entry $(k', \texttt{val})$ in the parent of v that separates v from v', and we merge v, $(k', \texttt{val})$ and v' into a node with $(a-1)+1+(a-2) = 2a-2 \leq b-1$ entries. This operation reduces the number of entries in the parent node by one, so is is possible that we have created another underflow.



Thus, we proceed up the root, fixing the underflows with join operations, until we can either apply a successful borrowing operation, or until we reach the root. If we perform a join operation with an entry from the root, it may happen that the root becomes empty (does not contain any more entries). In this case, we delete the empty root and make the (single) child of the old root the new root of the tree.



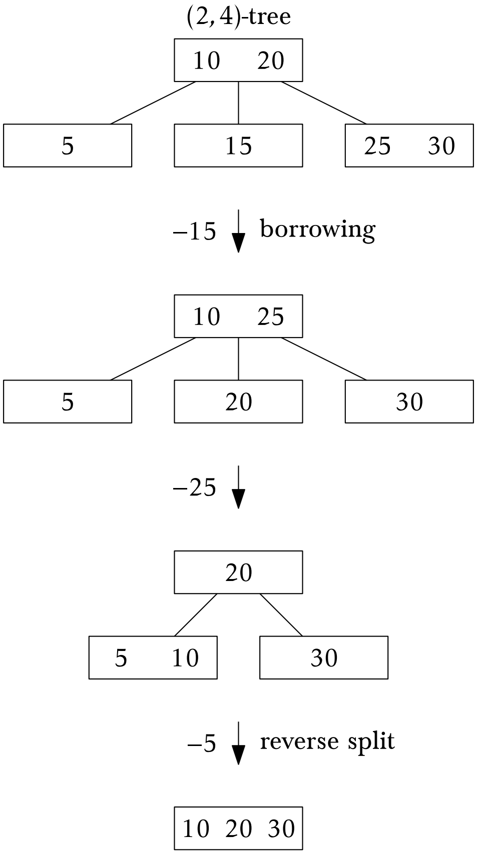
The pseudocode is as follows:

```
remove(k)
  find the node v that contains k (as above)
  if v is not a leaf then
    find the successor or predecessor k' of k // it does not matter
                                              // which one we take
    replace k by k' in v
    let k <- k' and v <- the leaf that contains k'
  // now v is a leaf
  remove k from v
  // now we need to fix the underflow
  while v contains a-2 keys and v is not the root do
    if v has a sibling with >=a keys then
      borrow a key from the sibling // see below
      return
    else
      // both siblings contain a-1 keys
      join v with a sibling // either left or right sibling is fine. See below█
      v <- parent of v
  if v is the root and v contains no keys then
    remove v and let the new root be the child of v
```

We describe the details of a borrowing operation with the right direct sibling v' of v: suppose the parent node p of v and v' contains $(\ldots,v,k'',v',\ldots)$, where $v = (w_1,k_1,\ldots,k_{a-2},w_{a-1})$ and $v' = (w_1',k_1',\ldots)$ has at least a entries. Then, we update $v \leftarrow (w_1,k_1,\ldots,k_{a-2},w_{a-1},k'',w_1')$ and $v' \leftarrow (w_2',k_2',\ldots)$, and p changes to $p \leftarrow (\ldots,v,k_1',v',\ldots)$. In words: the entry in p between v and v' moves to v, the leftmost child of v' becomes the leftmost child of v, and the leftmost entry from v' moves to p, between v and v'. A borrowing operation with the left direct sibling works analogously.

Next, we describe the details of a join operation with the direct right sibling v' of v: suppose the parent node p of v and v' contains $(\ldots,v,k'',v',\ldots)$, and write $v = (w_1,k_1,\ldots,k_{a-2},w_{a-1})$ and $v' = (w_1',k_1',\ldots,k_{a-1}',w_a')$. We join v, k'', and v' to $v'' = (w_1,k_1,\ldots,k_{a-2},w_{a-1},k'',w_1',k_1',\ldots,k_{a-1}',w_a')$. The parent node p changes to $p \leftarrow (\ldots,v'',\ldots)$. In words, we concatenate v and v', and the entry that separates v from v' in p moves down. A join operation with the left direct sibling works analogously.

Attention: If the parent node p has two children (and hence one entry), it may happen that a join operations leads to p being empty (containing no entries), with exactly one child. In this case, if p is the root, we can simply delete it. If p is in inner node (this may happen if $a = 2$), we proceed according to the algorithm and restore the entry in p using another borrowing or join operation.



Analysis. We analyze the running time of the operations on an (a,b)-tree. For this, just in the case of AVL-trees, we will compute the minimum and the maximum number of entries that an (a,b)-tree of a given height h can store.

The minimum number of entries is attained if every node in T stores as few entries as possible. That is, the root of T stores only one entry, and all other nodes store exactly $a-1$ entries. If this is the case, the root has exactly two children (if it is not a leaf), and every other inner node has exactly a children. Thus, at level 0, we have one node, at level 1, we have two nodes, and at every other level $i = 2,\dots,h$, we have $2 \cdot a^{i-1}$ nodes, because then at each additional level the number of nodes is multiplied by a factor of a. Multiplying the number of nodes with the number of entries (1 for the root, $a-1$ for every other node), we obtain the total number of entries as

$$\begin{aligned}
&1 + 2\cdot(a-1) + \sum_{i=2}^{h} 2\cdot a^{i-1}\cdot(a-1)\\
&= 1 + 2(a-1)\cdot\sum_{i=0}^{h-1} a^i\\
&= 1 + 2\cdot(a-1)\cdot\frac{a^h-1}{a-1}\\
&= 2\cdot a^h - 1.
\end{aligned}$$

Similarly, the maximum number of entries is attained if every node stores as many entries as possible, $b-1$. Then, every node has exactly b children, and the number of nodes at level i, for $i = 0,\dots,h$, if b^i, since we start with one root and since the number of nodes increases by a factor of b at every level. Multiplying the number of nodes with the number of entries $b-1$, we obtain the total number of entries as

$$\begin{aligned}&(b-1)\cdot\sum_{i=0}^{h} b^i\\ &= (b-1)\cdot\frac{b^{h+1}-1}{b-1}\\ &= b^{h+1}-1.\end{aligned}$$

Thus, for the number of entries n_h that can be stored in an (a,b)-tree of height h, we have

$$2\cdot a^h - 1 \le n \le b^{h+1}-1.$$

Solving for h, we obtain the minimum and the maximum possible height that and (a,b)-tree with n entries can have:

$$n \ge 2\cdot a^h - 1 \Rightarrow h \le \log_a \frac{n+1}{2} = O(\log_a n),$$

and

$$n \le b^{h+1}-1 \rightarrow h \ge \log_b(n+1)-1 = \Omega(\log_b n).$$

Considering the operations above on an (a,b)-tree T that stores n entries, we see that each operation visits a constant number of nodes at each level, and in each node v, an operation performs a constant number of scans keys and child references that are stored at v. Multiplying the maximum number of levels with the maximum number of entries that a node can have, we see that every operation takes at most $O(b\log_a n)$ time. On the other hand, multiplying the minimum number of levels with the minimum number of entries that a node can have, we see that for every (a,b)-tree, there are operations that take $\Omega(a\log b_n)$ time.

B-Trees. The variant of (a,b)-trees where we fix $b = 2a-1$, for a given parameter a are called *B-trees*. Historically, they were developed before the general notion of (a,b)-trees. B-trees are applied widely in computer science, e.g., to represent indexes in database management systems or in file-systems (e.g., in the Linux-file system `btrfs` = *B-tree file system*).

For large choices of a, B-trees (and (a,b)-trees) are useful for *external data structures*, data structures for data that is so large that it cannot be stored in main memory but needs to be kept in external memory (i.e., the hard disk). In this case, each node of the tree is stored at a different location of the external memory, and every visit to a node requires a read/write-operation on the external memory. Accessing the external memory is much slower than manipulating data in internal memory, so it is advantagous to keep the height of the tree small (whereas, in comparison, the cost for scanning a large number of keys in a given node, which can be done in internal memory, is negligible). This is exactly what B-trees achieve.

III

Dictionaries

KAPITEL 8

Dictionaries

We now describe the ADT *dictionary*, also called *map*. Dictionaries are very similar to the ordered dictionaries that we saw in the previos section, but without the assumption that we have an order on the key set. Instead, we require only that given two keys, we can determine whether the two keys are equal, or not. More precisely, just as in the ordered case, the goal of a dictionary is to maintain a set of *entries*. An entry consists of two parts, a *key* and a *value*. For each key, there is at most one entry. We would like to be able to create new entries for new keys, update values for existing keys, lookup the value for a given key. and delete existing entries.

More precisely, we have two underlying sets, the *key set* K and the *value set* V. We assume that we can check two given keys for equality, but nothing else. As before, an *entry* is an ordered pair (k,v) from the Cartesian product $K \times V$, and our goal is to maintain a dynamic set $S \subseteq K \times V$ such that each key appears at most once in an entry of S.

For completeness, we mention the details for the operations on the ADT dictionary. This is a subset of the operations for the ADT ordered dictionary, and the operations work in exactly the same way.

- `put`(k,v), where $k \in K$ is a key and $y \in V$ is a value. This operation has no precondition and does not have a return value. The effect is as follows: if S does not contain an entry with key k, we create add to S the new entry (k,v); if S already has an entry with key k, then this entry is deleted and replaced by the entry (k,v). Mathematically, the desired effect of `put` can be described as $S_{\text{new}} = (S_{\text{old}} \setminus (\{k\} \times V)) \cup (k,v)$.

- `get`(k), where $k \in K$ is a key. The precondition is that S contains an entry with key k (otherwise, `get`) will raise an exception). There is no effect. The return value is the value v of the (unique) entry $(k,v) \in S$ that has key k.

- `remove`(k), where $k \in K$ is a key. There is no precondition and no return value. The effect is as follows: if S contains an entry with key k, then this entry is removed in S. Otherwise, S remains unchanged. Mathematically, the desired effect of `remove` can be described as $S_{\text{new}} = S_{\text{old}} \setminus (\{k\} \times V)$.

- `isEmpty`:, There is no precondition and no effect. The operation returns `true`, if $S = \emptyset$, and `false`, if $S \neq \emptyset$.

- `size`. There is no precondition and no effect. The operation returns $|S|$, the number of entries in S.

We can use the implementations of the ADT ordered dictionary from the previous section to implement dictionaries, by imposing an arbitrary order on the keys (in a practical context, this is usually possible, because the keys are represented as bit strings, and we an define an order on bit strings, e.g., the lexicographic order).

Since dictonaries need less functionality than ordered dictionaries, there are faster and simpler methods to implement them. We will talk about those in the following chapters.

KAPITEL 9

Hash Tables

To develop a fast an simple implementation for the ADT dictionary, we start by considering a simple case: Suppose that our key set K consists of the integers $\{0,\dots,99\}$, and suppose that our values V is the set of strings. In this scenario, a simple implementation of a dictionary is as follows: we create an array T of Strings, with 100 positions.

Initially, all positions of T are set to the null value, $\perp$. Now, the idea is as follows: for any key $k \in \{0,\dots,99\}$, if there is an entry (k,v) for k in S, we store the value v at the array location $T[k]$. If there is no entry for k, the array location $T[k]$ contains the null value $\perp$.

The dictionary operations can easily be implmeneted as follows:

```
put(k, v): T[k] <- v
get(v):    if T[k] != NULL then
             return T[k]
           else
             throw NoSuchElementException
remove(k): T[k] <- NULL
```

All operations take $O(1)$ time, the space requirement is N plus the total size for all entries in the data structure. The main drawback is that we need to provide an array that has space for all possible keys. Thus, if the number of entries that we store is much smaller than the number of keys, we are wasting space.

Now, let us see what we can do if the key set becomes more complicated. We look at some examples for inspiration:

- Suppose that $K = \{1,2,\dots,100\}$. If we use the keys from K directly as indices in the table T, we waste space (position 0 in T would never be used). Thus, instead of storing the value for key k in $T[k]$, we store k in position $T[k-1]$. We need to perform an additional operation, but the space is used optimally.
- Suppose that $K = \{-10,-8,-6,\dots,0,2,4,\dots,90\}$. There are two issues: first, we cannot use the keys directly to index the table T, because there are no negative indices. Second, we are again wasting space, because there all the keys are even numbers. The solution is similar to the solution before: instead of storing the

value for key k in $T[k]$, we store k in position $T[(k+10)/2]$. This is a somewhat more complex transformation, but it allows us to use the space in the table T optimally.

- Suppose that $K = \{\text{red}, \text{blue}, \text{yellow}, \text{cyan}, \dots, \text{black}\}$, for a (small) selection of colors. Now, the issue is that the keys are not numbers, so at first glance it is not clear how to use them to refer to places in the array T. However, to represent the colors in our computer, we anyway need to use a numeric encoding. For example, one can encode the numbers using the RGB-encoding, which represents each color by giving the *red*, *green*, and *blue* components of the color. If we encode each color component by, say, three bits (which is feasible for a small set of colors), we can encode the colors with 9 bits overall, and we can use a table T with $2^9 = 512$ positions, which is quite feasible.

These three examples lead to the following insight: even if the key set K is a bit more complicated, it is often possible to get our scheme to work. All we need is a way to map the keys from K to positions in the table T. Such a way is called *hash function*.

Definition: Let K be a set of keys, and let $N \in \mathbb{N}$ be a number. A function $h : K \to \{0, \dots, N-1\}$ is called *hash function* for K (with table size N).

Now, the discussion so far shows: Suppose that given a set K of keys and a number N such that we can find a hash function $h : K \to \{0, \dots, N-1\}$ that (i) is injective (i.e., different keys are mapped to different table locations, $k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$); and (ii) can be evaluated quickly. Then, we have a fast and simple implementation for the ADT dictionary with key set K and arbitrary value set V. We create an array T with N entries, and we implement the operations as follows:

```
put(k, v): T[(h(k)] <- v
get(v):    if T[h(k)] != NULL then
             return T[h(k)]
           else
             throw NoSuchElementException
remove(k): T[h(k)] <- NULL
```

The running time for all operations is $O(1)$ plus the time to compute the hash function. The space requirement is N plus the total size for all entries in the data structure.

Now, the question is how to extend this idea to more general scenarios. The main requirements on the hash function h are that h is *injective* and that N is *small*. Typically, we cannot achieve both requirements at the same time. It may well be the case that K is very large (e.g., if K consists of all 64-bit integers) or even infinite (e.g., if K is the set of all Strings). Then, we cannot expect to obtain an injective function for reasonably small values of N (i.e., values of N such that we are willing to create an array of size N with potentially many empty positions.) However, if K is large, we typically do not want to store entries for all possible keys in K, but our set of actual entries will be quite small, compared to thee set of all possible keys K. Thus, there is still hope that we can get away with hash-function that is not injective, because for the actual set of keys in our table, if h is chosen well (and if N is reasonably large compared to the number of entries that are present in the table), the function h could behave like a function that is “almost injective”. Still, we typically need to commit to a hash function before we know the actual set of entries. Thus, we need to be prepared for dealing with the effects of missing injectivity. For the, we use the following definition:

Definition: Let K be a key set, $N \in \mathbb{N}$, and $h : K \to \{0,\dots,N-1\}$ a hash function. Let $k_1, k_2 \in K$, $k_1 \neq k_2$, be two distinct keys from K. Then, we call the keys k_1, k_2 a *collision* (for h) if and only if they hash to the same location in the table, i.e., if $h(k_1) = h(k_2)$.

Thus, in the general setting, we must be prepared to handle collisions in our current set of entries. There are different strategies how this can be done, with different names. We briefly introduce these strategies. More details will be given in the following chapters.

- First, we could store multiple entries at a given location of the table. That is, we simple store *all* entries (k,v) with $h(k) = i$ at $T[i]$. For this, the fields of T do not directly store the entries, but they store (a reference to) an auxiliary data structure that contains all the entries that hash to the respective field. This auxiliary structure is typically a linked list. This collision resolution strategy is called *chaining*.

- Second, if, upon inserting an entry (k,v), we see that the location $T[h(k)]$ in the hash table is already occupied, we can search through T for a “close-by” location that is still free. This strategy is called *open addressing*, because the location where an entry is stored is not fixed. There are different strategies for finding the “close-by” free location, with different names (e.g., *linear probing*, *quadratic probing*, *double hashing*).

- Third, if the location $T[h(k)]$ for an entry (k,v) is already occupied, we could still insist on storing the entry (k,v) in this location. In this case, the entry that is already present will have to move somewhere else. There are different ways to implement this strategy, again with different names (e.g., *cuckoo*, *Robin Hood*, *hopscotch*).

Note that once we allow for collisions, it is not enough to just store the value in a given entry in T. Instead, we now need to store the whole entry (k,v), and during any operation, we must compare the given key with the key in the entry that is store in the table (because it may be possible that our current location is occupied by an entry whose key hashes to the same location). This is where the requirement comes from that we can check elements of K for equality.

To summarize, a hash table consists of three parts:

- an array T with N entries, where N is called the *size* of the hash table (and is typically chosen when creating the hash table);

- a suitable hash function $h : K \to \{0,\dots,N-1\}$. The hash function depends on N, the table size, and it is typically chosen when creating the hash table; and

- a strategy for collision resolution.

In the next chapters, we will discuss in more detail how a hash function is chosen and how different strategies for collision resolution work.

KAPITEL 10

Hash Functions

We will now take a closer look at hash functions. Recall that given a set of keys K, and a table size N, a hash function is a function $h : K \to \{0, \ldots, N-1\}$ that maps every key from K to a position in the hash table.

When defining a hash function, there are two main goals:

- It should be possible to compute the hash function efficiently; and
- the hash function should behave like a "random" function. That is, for a "typcial" set S of entries, the keys in S should be distributed "evenly" over the hash table.

Typically, a hash function consists of two parts: a *hash code* and a *compression function*:

- **Hash code:** Given a set of keys K, a *hash code* for K is a function $\mathrm{hc} : K \to \mathbb{Z}$ that converts the keys from K to integer numbers.
- **Compression function:** Given a table size N, a compression function $c : \mathbb{Z} \to \{0, \ldots, N-1\}$ is a function that maps (arbitrary) integer numbers to locations in the hash table.

Once a hash code hc for K and a compression function c for a table size N are chosen, we can define the hash function as $h : k \mapsto c(\mathrm{hc}(k))$.

The reason why we distinguish these two parts is a separation of concerns, to make the design of our hash function modular. The *hash code* depends only on the set of keys. The main purpose of the hash code is to provide an encoding for our keys as integer numbers, and we can use a single hash code for a given set of keys, even if we need to implement multiple hash tables of different sizes. Since the hash code maps a key to the integers, it can be (and typically is) injective.[1] The compression function does not depend on the keys, but only on the table size. We can use the same compression function for different key sets, and it is possible to choose the compression function dynamically when creating a new hash table at runtime (whereas the hash

[1] Of course, there are practical limitations to this claim, since actual computers do not work with arbitrary integers but only with integers that can be represented by, say, 64 bits. However, the number of possible 64-bit integers is huge, compared to the table size, that we can typically ignore this issue in our considerations.

code is typically fixed when defining the key set, during the design phase). It is in the compression function that we usually try to ensure that the hash function behaves "randomly".

Hash codes. We give some examples of how a hash code code could be implemented, for different kinds of key sets.

- Suppose that K is the set of integers (i.e., in Scala, this would be $K = \texttt{Int}$). Then, there is nothing to do, we can directly use the key k itself as a hash code for k: $\mathrm{hc}(k) = k$.
- Suppose that K is the set of floating point numbers (i.e., in Scala, this would be $K = \texttt{Float}$. Now, we need to do something to convert a key k to an integer. There are different ways how this could be done. The simplest method would be simply to round k down to the nearest integer: $\mathrm{hc}(k) = \lfloor k \rfloor$. However, this might not be a good choice, for example, if (for some reason) we expect that all of our keys lie between 0 and 1. In this case, a slightly better method would be to first multiply k with a sufficiently large number, before rounding down to the nearest integer, e.g., $\mathrm{hc}(k) = \lfloor 1000 \cdot k \rfloor$. This makes it more likely that different keys receive different hash codes, and it could work quite well for certain situations. The most reliable method would be to consider the *bit representation* of the floating point number k, and to interpret this bit representation as an integer, e.g. $\mathrm{hc}(k) = \mathrm{ieee}(k)$, where $\mathrm{ieee}(k)$ yields the bit representation of the floating point number according to the IEEE standard.
- Suppose that K is the set of strings over some *alphabet* Σ.[2] Typical cases would be $\Sigma = \{0, 1\}$ (the case of bit-strings), $\Sigma = \{A, B, C, \ldots, Z\}$ (the usual Latin alphabet, with no additional characters), $\Sigma = \{C, G, T, A\}$ (the most common alphabet in computational biology), or $\Sigma = \mathrm{UNICODE}$ (the most common standard for a wide set of characters and glyphs). For example, typical strings over the Latin alphabet $\Sigma = \{A, B, C, \ldots, Z\}$ would be "HELLO" or "WORLD", whereas typical strings over the binary alphabet $\Sigma = \{0, 1\}$ would be "1100101" or "10101‘.

  To define a hash code for this case, we first pick an arbitrary, but fixed numbering $\|\cdot\|$ of the symbols in Σ with the numbers from 0 to $|\Sigma| - 1$. For example, for the Latin alphabet, we could take $\|A\| = 0, \|B\| = 1, \|C\| = 2, \ldots, \|Z\| = 25$, or for the binary alphabet, we could take $\|0\| = 0, \|1\| = 1$. Now, given a string τ over an alphabet Σ, we write τ as a sequence as symbols: $\tau = \sigma_0 \sigma_1 \ldots \sigma_{\ell-1}$, where ℓ is the total number of symbols in τ. For example, if $\tau = \mathrm{HELLO}$, we have $\ell = 5$ and $\sigma_0 = \mathrm{H}, \sigma_1 = \mathrm{E}, \sigma_2 = \mathrm{L}, \sigma_3 = \mathrm{L}, \sigma_4 = \mathrm{O}$. Then we define

  $$\mathrm{hc}(\tau) = \sum_{i=0}^{\ell-1} \|\sigma_i\| \cdot |\Sigma|^i.$$

[2] An *alphabet* is a non-empty finite set that is used to represent possible symbols in a String.

In other words, we interpret τ as a “number“ to base $|\Sigma|$, where the ”digits” are represented by the symbols of Σ. For example,

$$\begin{aligned}\text{hc}(\text{HALLO}) &= \|\text{H}\| \cdot 26^0 + \|\text{A}\| \cdot 26^1 + \|\text{L}\| \cdot 26^2 + \|\text{L}\| \cdot 26^3 + \|\text{O}\| \cdot 26^4 \\ &= 7 \cdot 1 + 0 \cdot 26 + 11 \cdot 26^2 + 11 \cdot 26^3 + 14 \cdot 26^4 \\ &= 6.598.443.\end{aligned}$$

This is an injective method for representing strings as numbers. We see that these numbers can become very big very quickly. In practice, the calculations are down with finite precision, e.g., with 64-bit integers, but the principle remains the same.

- We consider again the case that K consists of Strings, but this time we focus on the binary alphabet, e.g., we assume that K is the set of bit strings.[3] Then, there is a different way to define a hash code, using *cryptographic hash functions*. Cryptographic hash functions are hash codes that are especially engineered to map arbitrary bit strings to (large) integers (nowadays, typically of 256 or 512 bits) in a way that makes it very hard to find collisions. Even though we know that collisions must exist (because there is an infinite number of finite bit strings, much more than 2^{256}), the range of 2^{256} possible values is huge and, without additional structure, it will be very hard to find two bit strings that are hashed to the same position. The idea is that cryptographic hash functions should behave randomly enough that there is not enough structure to find a collision, while still being easy to compute. We know of no constructions that provably have these properties, but there are many heuristic candidates that are used in practice (e.g., SHA-2, SHA-3). However, these hash codes are relatively slow compared to their (non-cryptographic) alternatives, which is why they are not used in a typical hash table implementation.

In the JVM-context, objects must provide a function `hashCode` that is used in order to compute hash codes for storing the object in a hash table.

Compression function. Now, we give some examples of possible compression functions, for mapping an integer to a position in a hash table of size N.

- The simplest way to implement a compression function $c : \mathbb{Z} \to \{0, \ldots, N-1\}$ is to use modulo arithmetic: $c(z) = z \bmod N$. This is very fast to compute, and widely used in practice. However, this does not introduce much randomness into the process, and the distribution of keys is very predictable.
- A slightly more complicated way is to choose a prime number $p > N$ and to compute the compression function as $c(z) = (z \bmod p) \bmod N$. This creates a

[3] Actually, since all data in a contemporary computer is represented as bit strings, this is the most general practical case.

more unpredictable pattern. We choose p as a prime number to ensure that there are no common non-trivial factors between p and N, leading to a more "unpredictable" pattern.

- We can extend the previous method to obtain a choice of compression functions that is provably good. Here, we exploit the fact that we can choose the compression function when the hash table is constructed, and that we can use randomness in doing so: we choose a prime number $p > N$. When the hash table is constructed, we pick two random numbers $a \in \{1, \dots, p-1\}$ and $b \in \{0, \dots, p-1\}$, and we use the compression function $c(z) = ((a \cdot z + b) \bmod p) \bmod N$. This compression function is used throughout the whole lifetime of the hash table, but when a new hash table is created, new values a, b are chosen. This choice of compression function is called *universal hashing*.

- Another method that uses a random process during the construction of the hash table to determine the compression function is called *tabulation hashing*. Tabulation hashing works for the situation where the hash code is presented as a finite bit string (as is usually the case).

  For concreteness, suppose that the input hash code for the compression function is given as a 64-bit string. Then, a possible tabulation hashing scheme works as follows: at construction time, we create 8 arrays $A_1, \dots, A_8$ with 256 entries each. We initialize these entries with $8 \cdot 256 = 2048$ random numbers, each chosen uniformly from $\{0, \dots N-1\}$.

  To compute the compression function, we proceed as follows: given an input hash code z with 64-bits, we partition z into 8 blocks $B_1, \dots, B_8$ of 8 bits each. We can interpret the bits in each block as a number between 0 and 255, which we use to index the array $A_1, \dots, A_8$. We look up the corresponding entries, and we take the logical XOR of the results. This gives a number between 0 and $N-1$, the result of the compression function:

  $$c(z) = A_1[B_1] \texttt{ xor } A_2[B_2] \texttt{ xor } \cdots \texttt{ xor } A_8[B_8].$$

  Of course, the choice of 8 blocks of 8 bits is arbitrary, and we could define the compression function in different ways, e.g., using 4 blocks of 16 bits each (in which case we need to prepare $4 \cdot 2^{16} = 262.144$ random entries) or using 16 blocks with 4bits each (in which case we need to prepare $16 \cdot 2^4 = 256$ random entries). The idea of tabulation hashing is to approximate the notion that h is a random function, but instead of storing a random function for all 2^{64} possible input values, we compose it of (smaller) random functions, with a trade-off between the number of random tables (more tables means less space) and the block size (larger block size means more randomness).

  Just like universal hashing, tabulation hashing leads to a construction of hash functions that is provably good for hash tables.

KAPITEL 11

Hashing with Chaining

In hashing with chaining, collisions are resolved by storing all the entries (k, v) that hash to a given location i in the hash table at the same position $T[i]$. Since we cannot know in advance how many entries will end up at a given position of the hash table, we need an auxiliary data structure to store these entries. Typically, this auxiliary data structure is a singly linked list, the simplest dynamic data structure that can store an arbitrary number of entries. In the terminology of hash tables, such a linked list is called a *chain*, and the collision resolution method is called *chaining*.

Thus, in chaining, we have an array T of size N, and each position i of T stores a reference to (the head of) a linked list that stores all the entries $(k, v) \in S$ with $h(k) = i$. The operations are implemented as follows:

- `put`(k, v): We set i to $h(k)$, and we go through all the entries that are stored in the linked list at $T[i]$, checking for an entry with key k. If we find such an entry, we update the value to v. Otherwise, we append to the linked list at $T[i]$ a new node for the entry (k, v).

- `get`(k): We set i to $h(k)$, and we go through all the entries that are stored in the linked list at $T[i]$, checking for an entry with key k. If we find such an entry, we return the value stored with the entry. Otherwise, we throw a `NoSuchElementException`.

- `remove`(k): We set i to $h(k)$, and we go through all the entries that are stored in the linked list at $T[i]$, checking for an entry with key k. If we find such an entry, we remove the corresponding node from the linked list. Otherwise, we throw a `NoSuchElementException`.

Analysis. Now we analyze the performance of hashing with chaining. First, we note that the space requirement for the whole structure is $O(N + \text{size}(S))$, where $\text{size}(S)$ denotes the total space that is needed for the entries in S. Thus, as long as the number of positions in the hash table is not much larger than the total size for the entries in S, hashing with chaining is quite space efficient.

To analyze the running time, it is necessary to make an assumption on the hash function h. Without any such assumption, we cannot make a nontrivial statement about the running time, because otherwise, the hash function h could, e.g., be a constant function that maps all the keys to 0. In such a case, the running time of the operations would be $\Omega(n)$, where n is the number of entries in the structure.

Thus, for our analysis, we will assume that h behaves like a random function. That is, we imagine that when the hash table is constructed, we create a new hash function by assigning to each key in k a random position in the hash table, uniformly distributed and independent of the others. Formally, this means that for every $k \in K$ and for every $i \in \{0, \dots, N-1\}$, we have

$$\mathrm{Pr}_h[h(k) = i] = 1/N,$$

independently of the values for the other keys $k' \in K$. This assumption is not very realistic, since in practice we will never be able to generate and store a completely random function on the key set with reasonable effort. However, the assumption leads to clean calculations and a first idea of the performance guarantees that we can expect from a hash table. It often turns out that similar results can then also be derived for more realistic "random" hash functions (e.g., universal hashing, tabulation hashing), but with somewhat more complicated calculations.

Thus, consider the following situation: suppose that the hash table stores a fixed set S of n entries, and suppose that next we perform an operation on the hash table (get, put, or remove) that involves a key $k \in K$. The key k may or may not be present in S. We assume further that S and k are independent of the choice of h.

Now the cost of the operation on the key k is proportional to the time to evaluate the hash function $h(k)$, plus the number of entries in the list that is stored at $T[h(k)]$ (in the worst case, the operations get, put, remove involve a single scan through the list at $T[h(k)]$). Assuming that h is a good hash function, the time to compute $h(k)$ is $O(1)$. We denote the length of the list at $T[h(k)]$ by L. Then, the running time of the next operation is $O(1 + L)$.

What can we say about L? We know that L is a random variable that depends on the random choice of h. As we did in the case of skip lists, our way to understand L is by computing its expected value, $\mathbf{E}[L]$. This gives us an idea of the "typical" behavior of L.

To compute $\mathbf{E}[L]$, we notice that L is exactly the number of entries $(k', v') \in S$ with $h(k') = h(k)$, i.e., the number of entries in S that hash to the same position as the given key k.

To compute this quantity, we define *indicator random variables $L_{k'}$*, for every key $k' \in K$. The indicator random variable $L_{k'}$ is set to 1 if and only if $h(k') = h(k)$, and 0, otherwise. Then, by our definition of L, we have

$$L = \sum_{(k',v') \in S} L_{k'},$$

since the right hand side counts exactly the number of entries in S whose keys are hashed to the same position as k.

Using linearity of expectation, we get

$$\mathbf{E}_h[L] = \mathbf{E}_h\left[\sum_{(k',v')\in S} L_{k'}\right] = \sum_{(k',v')\in S} \mathbf{E}_h[L_{k'}].$$

Thus, we need to compute the expected value $\mathbf{E}_h[L_{k'}]$, for $k' \in K$. By definition, we have

$$\mathbf{E}_h[L_{k'}] = 0 \cdot \Pr_h[L_{k'} = 0] + 1 \cdot \Pr_h[L_{k'} = 1] = \Pr_h[L_{k'} = 1] = \Pr_h[h(k') = h(k)].$$

Now, there are two cases: first, if $k' = k$, then clearly $Pr_h[h(k') = h(k)] = 1$. Otherwise, if $k' \neq k$, we claim that $Pr_h[h(k') = h(k)] = 1/N$. Indeed, there are N^2 possible ways how k' and k can be hashed by h, and since we assume independence, each such way occurs with probability $1/N^2$. Of these configurations, there are exactly N that lead to a collision. Thus, $Pr_h[h(k') = h(k)] = N/N^2 = 1/N$, as claimed.[1] Thus, getting back to L, we have

$$\begin{aligned}
\mathbf{E}_h[L] &= \sum_{\substack{(k',v')\in S\\ k'=k}} \mathbf{E}_h[L_{k'}] + \sum_{\substack{(k',v')\in S\\ k'\neq k}} \mathbf{E}_h[L_{k'}]\\
&\leq 1 + \sum_{\substack{(k',v')\in S\\ k'\neq k}} \frac{1}{N}\\
&\leq 1 + \frac{|S|}{N}.
\end{aligned}$$

In conclusion, the expected running time for each operation on our hash table is $O(1 + |S|/N)$.

The quantity $|S|/N$ is called the *load factor* of the hash table. It changes over the life-time of the hash table, and it measures how many entries there are currently per position in the hash table. Our analysis shows that if the load factor is $O(1)$, then the expected time for the operations on the hash table is $O(1)$. The load factor in hashing with chaining can be any non-negative rational number between 0 and ∞.

If the load factor is small, the hash table is not space efficient, since many positions in the table are unused. If the load factor is too large, the performance suffers, because the linked lists in the table become too long. In the literature, it is recommended to keep the load factor between 1 and 3. This can be done by rebuilding the hash table with a new table size, whenever the load factor becomes too large or too small. This rebuilding operation is called *rehashing*. It entails the choice of a new compression function (since the table size changes), and it takes time linear in the size of the hash table and in the current number of entries, because all the entries must be placed into the new table. As in the case of implementing a stack with a dynamic array (as

[1] This probability can also be argued in a slightly different way that avoids a calculation: by assumption, the positions of k' and k' in the hash table are independent. Once the position of k' is chosen, there is exactly one possibility how k can collide with k'. Since there are N possible positions for k, the collision probability is thus $1/N$.

discussed in *Konzepte der Programmierung*, one can show that the total cost for this rehashing operation is negligible to the total cost of updating the hash table, provided that an appropriate strategy is chosen for determining the time for the next rehash.

In the next chapter, we will see another collision strategy that is based on *open addressing*: all entries are stored directly in the hash table, and if a position is already occupied, we search for a different place for the entry.

KAPITEL 12

Hashing with Linear Probing

Now, we consider a collision resolution strategy that falls into the category of *open addressing*. This means that the position of a entry (k,v) in the hash table is not only determined by the hash value for the key k, but also by the entries that are already present in the table.

More precisely, in an open addressing scheme, all entries are stored directly in the hash table T. There is no secondary data structure as in chaining. If we try to insert a new entry (k,v) and the position $T[h(k)]$ is already taken, the entry is stored in a different location, according to a certain strategy.

In *linear probing*, the strategy consists in a linear scan through the T starting at $T[h(k)]$, until the first free position in T is discovered. This is the position where the entry (k,v) will be stored. Accordingly, when performing a lookup for a key k, but it is not enough to consider only the position $T[h(k)]$. Instead, if $T[h(k)]$ is occupied by another entry, we start a linear scan until either the entry for k is found or until we encounter the first free position. Only when we see the free position can we be sure that k is not present in the table.

Now, when performing a deletion, we must be careful. To delete an entry (k,v), we cannot simply replace the entry (k,v) in the table by the null element $\perp$. If we did this, it might happen that the lookups no longer work properly: we might finish a linear scan, thinking that a key k is not present in T, even though k might appear after a $\perp$ that is due to a deletion after the entry for k was inserted. (**TODO**: Example)

To resolve this, we will introduce a special third state that a position $T[i]$ in the hash table T can have. In addition to (i) being *occupied* by an entry ($T[i] = (k,v)$; and (ii) being *free* ($T[i] =\perp$), we now have the third possibility (iii) that $T[i]$ has been *deleted* ($T[i] = *$). We use the third state to mark the situation that $T[i]$ was once occupied by an entry that has been removed due to a later deletion. This third state plays different roles during lookups and insertions. For a lookup, it is treated like an *occupied* position, and the scan continues over the deleted position. For an insertion, it is treated like a free position, and it can be used for the entry to be inserted.

We now give the details for the operations.

Lookup. For a lookup of a key k, we computed the hash value $h(k)$ for k. Starting from $h(k)$, we scan through T in a linear fashion (wrapping around if we encounter the end of the array). The scan stops when either (i) the key is discovered (in which case we return the corresponding value); or (ii) we encounter the first free position (in which case the key does not exist in the table); or (iii) we scanned the whole table without seeing k (in this case, the key is also not present in T, and T does not have any free positions). Note that the table may contain deleted positions, they do not need to be handled in a special way by the algorithm.

```
get(k):
  pos <- h(k)
  // the scan takes at most N steps
  for i := 1 to N do
    // if we see an empty position, k is not there
    if (T[pos] == NULL)
      throw NoSuchElementException
    // if we see the key, we return the value
    if (T[pos].k == k)
      return T[pos].v
    // we go to the next position, wrapping around if necessary
    pos <- (pos + 1) mod N
  // we have inspected all positions, k is not there
  throw NoSuchElementException
```

Put. To perform a put on a key k, we first compute the hash value $h(k)$. We start a scan from this position, in order to find the key k or the first empty position. If, along the way, we see a deleted position, we remember this position. Now, if we encounter the key k, we just update the value v according to the new entry. If we reach an empty position, we know that the key k is not present, and we insert he new entry either in the empty position, or in the first deleted position that we have encountered.

```
put(k, v):
  pos <- h(k)
  // the first deleted position
  delPos <- NULL
  // the scan takes at most N steps
  for i := 1 to N do
    // if we find k, we update the value and are done
    if (T[pos].k == k)
      T[pos] <- (k,v)
      return
    // if we see a deleted position, we remember it, if
    // it is the first one
    if (T[pos] == DELETED && delPos == NULL)
      delPos <- pos
    // if we see an empty position, k is not there
    if (T[pos] == NULL)
      break
    pos <- (pos + 1) mod N
  // At this point, we know that k is not there
  // if there was a deleted position, put k there
  if (delPos != NULL)
    T[delPos] <- (k,v)
  // if we stopped at a free position, put k there
  else if (T[pos] == NULL) then
    T[pos] <- (k,v)
  // otherwise, there are no free positions.
  else
    throw TableFullException
```

Alternatively, we can always put the entry into the first deleted position that we encounter. Then, we must scan further for an entry for *k*, and mark it as deleted, if necessary.

```
put(k, v):
  pos <- h(k)
  // has the entry already been inserted?
  inserted <- False
  // the scan takes at most N steps
  for i := 1 to N do
    if (T[pos].k == k)
      // if k has not been inserted,
      // we add the entry
      if (not inserted)
        T[pos] <- (k,v)
      // otherwise, we mark the position as deleted
      else
        T[pos] <- DELETED
      return
    // it is the first deleted position, we insert
    // the entry, but we must continue, in case
    // the k key comes later
    if (T[pos] == DELETED && not inserted)
      T[pos] <- (k,v)
      inserted <- True
    if (T[pos] == NULL)
      // if we see an emptry position,
      // we add the entry, unless we have
      // already done so
      if (not inserted)
```

```
        T[pos] <- (k,v)
      return
    pos <- (pos + 1) mod N
  // At this point, we have scanned
  // the whole table
  if (not inserted)
    throw TableFullException
```

Deletion. To delete an entry for a key k, we proceed as in a lookup. If we find the entry, we just mark the position as deleted.

```
remove(k):
  pos <- h(k)
  for i := 1 to N do
    if (T[pos] == NULL)
      throw NoSuchElementException
    if (T[pos].k == k)
      T[pos] <- DELETED
      return
    pos <- (pos + 1) mod N
  throw NoSuchElementException
```

Analysis. How good is linear probing? As with chaining, the space requirement is $O(N + \text{size}(S))$, where $\text{size}(S)$ is the total size space for the entries in S. In practice, the space requirement will actually be slightly better than for chaining, because there is no overhead for a secondary data structure (i.e., we do not need any space for maintaining the linked lists).

To analyze the running time, we again assume that the hash function h behaves like a random function that assigns to each key $k \in K$ a uniformly random position in the hash table T, independently of the other keys. We again consider the scenario that the hash table stores a fixed set S of n entries, and we suppose that next we perform an operation on the hath table that involves a fixed key $k \in K$. To simplify the proof, we further assume that $N = 6n$, i.e., that the hash table contains six times as many positions as there are entries.[1] Finally, we assume that there are no deleted positions in the hash table, either because we use the second deletion strategy that moves entries upon deletion, or because no deletions have taken place so far.

To analyze the cost of the operation on the key k, we need the notion of a *cluster*: let $\ell \in \{0,\dots,n\}$ and $i \in \{0,\dots,N-1\}$. A *cluster* of length k with starting point i in the hash table T is a sequence $i, i+1, i+2, \dots, i+\ell-1$ of ℓ consecutive indices, such that the positions $T[i], T[i+1], \dots, T[i+\ell-1]$ in ℓ are all occupied by entries from S, while $T[i-1]$ and $T[i+\ell]$ are free.[2]

[1] This assumption can be relaxed, at the cost of a slightly more complicated proof.

[2] We take all the indices modulo N, e.g., $T[-1] = T[N-1]$ and $T[N] = T[0]$.

Now, the time for an operation on the key k is proportional to the time to evaluate the hash function $h(k)$, plus the length of the cluster (if any) that contains $h(k)$. That is, the running time is $O(1 + \text{length of the cluster that contains } h(k))$.

Thus, we need to analyze the length of the clusters in T, after S was inserted. For this, fix a length ℓ and a starting position i. If T contains a cluster of length ℓ with starting point i, then necessarily there must be a subset of ℓ entries S' in S such that all keys in S' are hashed by h to $i, i+1, \ldots, i+\ell-1$.[3] Thus, we can say that

$$\Pr_h[\text{there is a cluster of length } \ell \text{ that starts at } i] \leq \binom{n}{\ell}\left(\frac{\ell}{N}\right)^{\ell}.$$

This is because there are $\binom{n}{\ell}$ ways to choose the subset S', and for each key k' in S', the probability that $h(k') \in \{i, i+1, \ldots, i+\ell-1\}$ is ℓ/N, independently of the other keys.

Using the bound $\binom{n}{\ell} \leq (ne/\ell)^{\ell}$ and the fact that $N = 6n$, we conclude that

$$\begin{aligned}\Pr_h[\text{there is a cluster of length } \ell \text{ that starts at } i] &\leq \left(\frac{ne}{\ell}\right)^{\ell} \cdot \left(\frac{\ell}{6n}\right)^{\ell}\\ &\leq \left(\frac{e}{6}\right)^{\ell}\\ &\leq \frac{1}{2^{\ell}},\end{aligned}$$

since $e = 2.71\ldots \leq 3$. Thus, we can also estimate the probability that there is a cluster of length *at least* ℓ that starts of i:

$$\begin{aligned}\Pr_h[\text{there is a cluster of length at least } \ell \text{ that starts at } i] &\leq \sum_{m=\ell}^{\infty} \frac{1}{2^m}\\ &= \frac{1}{2^{\ell}} \cdot \sum_{m=0}^{\infty} \frac{1}{2^m}\\ &= \frac{1}{2^{\ell}} \cdot 2\\ &= \frac{1}{2^{\ell-1}},\end{aligned}$$

using the geometric series. Thus, the probability that we have a cluster of length at least $2\log n$ anywhere in the table can be bounded by

$$\begin{aligned}&\Pr_h[\text{there is a cluster of length at least } 2\log n]\\ &\leq \sum_{i=0}^{N-1} \Pr_h[\text{there is a cluster of length at least } 2\log n \text{ that starts at } i]\end{aligned}$$

[3] This is just a necessary condition, but this is ok, because we are only looking for an upper bound.

$$
\begin{aligned}
&\leq \sum_{i=0}^{N-1} \frac{1}{2^{2\log n - 1}} \\
&= N \cdot \frac{2}{n^2} \\
&= \frac{12}{n},
\end{aligned}
$$

since $N = 6n$ and $2^{2\log n} = n^2$. Thus, with probability at least $1 - 12/n$, the operations in T take time at most $O(\log n)$.

To analyze the expected time for an operation on the key k, we first consider the case that $k \notin S$. Then the expectd time for the operation on k is proportional to by the definition of the expected value.

$$\sum_{\ell=0}^{n} \ell \cdot \Pr_h[h(k) \text{ is in a cluster of length } \ell]$$

by distinguishing the starting point of the cluster, we see that this sum is equal to

$$= \sum_{\ell=0}^{n} \sum_{i=0}^{N-1} \ell \cdot \Pr_h[\text{a cluster of length } \ell \text{ starts at } i \text{ and } h(k) \text{ is in this cluster}]$$

As we saw, the probability that a cluster of length ℓ starts at i is at most $2^{-\ell}$. Furthermore, the probability that $h(k)$ falls into this cluster is at most ℓ/N. Since $k \notin S$, the two events are independent of each other, so we can multiply the probabilities, to see that this is at most

$$\leq \sum_{\ell=0}^{n} \sum_{i=0}^{N-1} \ell \cdot \frac{1}{2^\ell} \cdot \frac{\ell}{N}$$

Since we add the same term N times, this is

$$= \sum_{\ell=0}^{n} N \cdot \frac{1}{2^\ell} \cdot \frac{\ell}{N}$$

The N cancels, and we add the same term ℓ times, so this becomes

$$= \sum_{\ell=0}^{n} \frac{\ell^2}{2^\ell} = O(1),$$

using the fact that $\sum_{i=0}^{\infty} i^2/x^i = O(1)$, for all $x \in (-1,1)$. The case that $k \in S$ is similar, only that we need to be a bit more careful in estimating the probabilites. **TODO: Add details** Thus, the expected time for the operations is $O(1)$.

Conclusion. Hashing with linear probing is a simple and efficient hashing scheme that performs very well in practice. In fact, on modern hardware, hashing with linear probing can be more efficient than hashing with chaning, because the memory architecture in today's computer systems favors algorithms that perform a linear scan through memory (as opposed to following a sequence of pointers).

As with chaining, the load factor plays an important role for the efficiency of a hash table. Since all the entries are stored directly in the hash table, the load factor in linear probing can be at most 1. In practice, it seems that it is a good strategy to keep the load factor at around 0.6.

There are many variants of the basic idea of linear probing (e.g., *quadratic probing*, *double hashing*, etc). The main idea is to vary the sequence in which the positions in the hash table are scanned for the first free position, in the hope that it will be less likely that clusters occur.

In the next section, we will see a more recent strategy for resolving collisions that tries to combine the advantages of chaining and linear probing.

KAPITEL 13

Cuckoo Hashing

In *cuckoo hashing*, we try to combine the advantages of chaining and of linear probing. Like in linear probing, all the entries are stored directly in the hash table T. There is no secondary data structure. Like in chaining, the position of the entries is determined by the hash function. It cannot happen that an entry ends up arbitrarily far from its assigned position. The main idea for making this possible is to use *two* hash functions h_1 and h_2 that assign possible positions to the keys. That is, an entry with key k can be stored *either* at position $h_1(k)$ *or* at position $h_2(k)$ (and nowhere else).

The option of having two possible positions for each key adds flexibility to the structure and allows for very efficient lookups and deletions. The main challenge is to implement insertions. Here, the approach is as follows: when inserting a new entry (k,v), we first consider the position $h_1(k)$ that is assigned to k by the first hash function. If $T[h_1(k)]$ is free, we simply write the entry and finish. Otherwise, $T[h_1(k)]$ is already occupied by an entry (k',v'). Now, we take inspiration from the cuckoo bird: we remove the entry (k',v') from $T[h_1(k)]$ to make space for our own entry (k,v), which is now stored in this position. Unlike the cuckoo, however, we cannot just discard the entry (k',v'), but we need to insert it at another position in T. For this, we compute $h_1(k')$, and if $h_1(k') \neq h_1(k)$, we relocate (k',v') to $h_1(k')$. Otherwise, we relocate (k',v') to $h_2(k')$. If this other position is empty, we are done. If it is again occupied, we need to relocate the entry at this other position to a new position. This continues until we either find an empty position or until we can be certain that our relocation scheme cannot be successful. In the latter case, the complete structure is rebuilt, using new hash functions h_1 and h_2.

We now look at the pseudocode for the operations.

Lookup. As mentioned above, a lookup for a key k is easy: we only need to consider the positions $h_1(k)$ and $h_2(k)$ to see if the key is present. If so, we return the value, if not, we report that k is not present.

```
get(k):
  if (T[h1(k)].k == k)
    return T[h1(k)].v
  if (T[h2(k)].k == k)
    return T[h2(k)].v
  throw NoSuchElementException
```

Deletion. Deletion works in the same way as a lookup. We check the positions $h_1(k)$ and $h_2(k)$. If key is present, we delete the entry by overwriting it with $\perp$. If not, we report that the key has not been found.

```
remove(k):
  if (T[h1(k)].k == k)
    T[h1(k)] <- NULL
    return
  if (T[h2(k)].k == k)
    T[h2(k)] <- NULL
    return
  throw NoSuchElementException
```

Put. In a put, we first check if the key k is present in the table. If so, we simply update the value. If not, we start the insertion process. The insertion process is implemented in a `for`-loop that makes N attempts of inserting the current entry. In each iteration of the loop, we have a current insertion position `pos` and a current entry (k,v). We check if the position `pos` in T is empty. If so, we simply put (k,v) into $T[$`pos`$]$ and are done. If not, we exchange our current entry (k,v) with the entry (k',v') that is already present in $T[$`pos`$]$. We set `pos` the second possible position for (k',v'), and we proceed to the next iteration of the loop.

Note that it may happen for a key k that $h_1(k) = h_2(k)$, i.e., that the two possible positions coincide. This is not necessarily a problem, the insertion algorithm can still succeed. Note also that we make N attempts for the insertion. This is just for simplicity, it may well happen that the insertion algorithm is caught in a look before that, but it would be more complicated to check for this.

If the insertion does not succeed, we reconstruct the whole table with new hash functions h_1 and h_2. This means that if we want to use cuckoo hashing, we need to use a compression function that is chosen randomly from a larger set (as, e.g., in universal hashing or in tabulation hashing), so that this step can be carried out.

```
put(k, v):
  // if k is present, we simply update the value
  if (T[h1(k)].k == k)
    T[h1(k)].v <- v
    return
  if (T[h2(k)].k == k)
    T[h2(k)].v <- v
    return
  // if not, we must insert (k, v)
  // if the table is already full, we cannot insert $k$
  if (|S| == N)
```

```
    throw TableFullException
  // we try to insert the entry at the first position
  pos <- h1(k)
  for i := 1 to N do
    // if the position is empty, we insert the entry
    // and are done
    if (T[pos] == NULL)
      T[pos] <- (k,v)
      return
    // otherwise, we exchange the current entry and
    // determine the second possible position
    (k,v) <-> T[pos]
    if (pos == h1(k))
      pos <- h2(k)
    else
      pos <- h1(k)
  // if the insertion does not succeed, we need to rebuild the table
  Choose new hash functions and rebuild the table
  put(k,v)
```

Analysis. A major advantage of cuckoo hashing is that lookups (and deletions) take $O(1)$ time *in the worst case*. Thus, in a scenario where insertions are rather infrequent, but lookups occur very often, cuckoo hashing can be superior to chaining or linear probing.[1]

One can show that if the load factor is small enough (e.g., 1/2), and if the hash functions behave randomly, then the expected time for an insertion operation is $O(1)$, so that in theory, cuckoo hashing performs as well as chaining and linear probing. This analysis is beyond the scope of these lecture notes and is deferred to later classes.

There are several variants of cuckoo hashing, e.g., we could use more than two hash functions, or we could introduce a *stash*, by storing more than one entry in a position of the hash table.

Cuckoo is a relatively new technique for collision resolution. After that, there have been further variants and improvements, e.g., Robin Hood hashing or hopscotch hashing. We will not discuss these here, but the interested reader is invited to investigate them further.

This chapter concludes our discussion of collision resolution strategies. In the next chapter, we will talk a bit more about further uses of hash functions beyond the implementation of hash tables.

[1] Such a situation occurs, e.g., in network routers where the next hop for a given address needs to be determined very quickly, while changes in the network topology are relatively rare.

KAPITEL 14

More Applications of Hash Functions

One major concept in computer science that we have encountered in this section are *hash functions*. Hash functions have turned out to be very useful, with powerful applications well beyond the implementation of hash tables.

Here, we will look at further uses of hash functions. Let us recall the main properties of a hash function $h : K \to \{0, \dots, N-1\}$. First, the main purpose of h is to map a large (potentially infinite) universe of keys to a (relatively) small set of numbers Second, the function value $h(k)$ of a given key k should be “easy to compute”. Third, the hash function should behave *randomly*. That is, the hash value of a given key k should be “unpredictable” and independent of the other keys. Collisions should be “unlikely”.

Now, suppose that we have a hash function h whose range is of “medium size”, e.g., $N = 2^{128}$ or $N = 2^{256}$. That is, the hash values can be represented by 128 or by 256 bits. Such a range is not useful for creating a hash table (even with today’s hardware, it seems very difficult to create an array with 2^{256} positions). However, since 2^{256} is a very large number, we see that even though, mathematically, collisions for h exist (if K is infinite, say), it is very unlikely that we will ever encounter a collision for h in practice. In other words, if N is of “medium size”, and if h behaves sufficiently randomly, we can consider h to be “practically injective”.

Since the notion of a “sufficiently random” function is not a rigorous mathematical concept, we can use ideas from complexity theory to replace the idea of a “random function” by the idea of a “difficult function”. This leads to the notion of a *cryptographic hash function*.

Suppose that our key set K consists of the set of all finite binary strings, denoted by $\{0,1\}^*$. A function $h : K \to \{0, \dots, N-1\}$ is called a *cryptographic hash function* if (i) given a key $k \in \{0,1\}^*$, it is “easy” to compute the hash value $h(k)$ (h is easy to compute); (ii) given a hash value $x \in \{0, \dots, N-1\}$, it is “difficult” to find a key $k \in \{0,1\}^*$ such that $h(k) = x$ (h is *resistant* to finding pre-images); and (iii) it is “difficult” to find two distinct keys $k, k' \in \{0,1\}^*, \neq k'$, such that $h(k) = h(k')$ (h is (strongly) resistant to finding collisions). Note that (iii) also implies that given a key $k \in \{0,1\}^*$, it is “difficult” to compute another key $k' \neq k$ such that $h(k) = h(k')$ (this property is also called “weakly resistant to finding collisions”).[1]

[1] The precise definition of a cryptographic hash function is more technical and will not be given here. It will be treated in a class on cryptography.

Even though we do not have any theoretical construction of a hash function that is *provably* cryptographic, there are several *heuristic* constructions that are used in practice (e.g., SHA-2, SHA-3).

Cryptographic hash functions can be used to create short (a few bytes) *fingerprints* for arbitrary data that (i) determine the data uniquely and (ii) do not allow for any conclusions on the original data. We stress that, mathematically, this is impossible: of course, even a cryptographic hash function will have collisions, and of course knowing the function value determines with preimages are possible. The point is that it should be beyond our computational capabilities to obtain this information within a reasonable time frame. Then, it may as well be impossible, for all practical purposes.

One application of cryptographic hash functions is the *secure storage of passwords*. Passwords are typically used to authenticate a user who wants to access a system. For this, the system must know the password for each user. However, storing these passwords in a plain text file poses a security risk. If an attacker gets hold of this file, the attacker could use it to impersonate every single user. Thus, most systems do not store the passwords directly. Instead, they fix a cryptographic hash function, and they store only the cryptographic hash values of the password. Once a user enters a password, the system computes the hash value of the entered password and compares it to the hash value that is stored in the passwords file. Since cryptographic hash functions are collision resistant, an adversary will not be able to construct a password that is accepted, *even if the contents of the passwords file are known*.

A second application of cryptographic hash functions is the generation of *message digests*. That is, we can use cryptographic hash functions to certify that two versions of a file are identical. For example, suppose we would like to download a very large file from the internet. After the file is downloaded, we would like to verify if our version of the file really matches the original. For this, we can locally compute a cryptographic hash value for the download, and we can quickly compare it to a hash value that the provider of the file has posted on the internet. If these two values are identical, we can be fairly certain that our download matches the original. Since the hash code is very small (only a few bytes), the check can be performed manually and the data transfer is much more reliable than for the large file. As a second example, suppose that at a certain deadline, we expect that a large number of users each would like to upload a large file onto our server (e.g., when turning in a large programming assignment). To save server capacity, we could instead require each user to upload only a cryptographic hash code of the respective file. The real files can then be uploaded later, after the deadline, at a more leisurely pace. This saves server capacity. Since cryptographic hash functions are collision resistant, it is not possible for the users to “cheat” and to use the additional time to produce a (supposedly improved) version of their file that differs from the version at the deadline.

A third application of cryptographic hash functions is the creation of *hash references* that make it possible to have *tamper-proof* data structures. Recall from *Konzepte der Programmierung* that in an object-oriented programming language, data is modeled by *objects* that reside in memory. The objects are accessed via *references*, variables in the code that can be associated with the different objects at different times (and the same object can have multiple references associated with it).

In a hash reference, we do not just store a reference to an object, but also a cryptographic hash value that is derived from all the attributes of the object. This makes it possible, when using the reference to access the object, to verify that the attributes of the object have not changed since the last access (e.g., via a different reference that refers to the same object). Since the hash value can be small compared to the original object, it is feasible to store this information directly with the reference.

A major advantage of hash references is that they can be used in *chained* data structures. For example, suppose that we have a singly linked list L whose `next`-references are implemented with hash references. For example, suppose that the list consists of four nodes n_1, n_2, n_3, n_4, in this order, and a `head`-reference that points to n_1. Then, the hash reference in n_3 determines all the attributes in n_4, the hash reference in n_2 determines all the attributes in n_3, and so on. Crucially, among the attributes in n_3 that are determined by n_2, there is also the hash value for n_4. Thus, it follows that the hash reference in n_2 does not only determine the attributes in n_3, but also the attributes in n_4. Similarly, the hash reference in n_1 determines all the attributes in n_2, and, via the hash reference from n_2, also all the attributes in n_3 and n_4. The hash reference in the `head` determines the attributes in the whole list.

In other words, given the hash value for the head reference, we can verify that the whole linked list has not been changed, by recursively checking the hash values for all the nodes in the list. Even more, we can add additional nodes at the head of the list, while keeping the structure intact (for adding nodes, we do not need to know the hole list, just the hash value for the head reference).

A singly linked list that uses hash references is called a *block-chain*. The main application of a block-chain is to realize a *tamper-free write-only log*, a sequence of entries that can be extended continuously such that possible manipulations of past entries can be detected easily (using the supposed history and the hash value for the current head).

An example application for such a tramper-free write-only log is as follows: suppose that students take a class in algorithms, and each week, the students are supposed to hand in an assignment. The assignments are graded, and only if the students achieve at least 60 % of the possible assignment credits do they get credit for the class. Once the class is finished, the result (pass or fail) is entered into the system, and, for data privacy reasons, the assignment credits are deleted from the central server. Now, suppose that a year later a student realizes that the official entry in the transcript is wrong, and would like to effect a correction. How can the student support this claim? The official data has been deleted. The student can show the supposed grades, but how can we verify that they are correct? This problem can be solved with a block-chain: each time an assignment is graded, the teaching assistant adds a node to the block-chain with the result. After the class is over, the grades are removed from the central system, only the hash code for the head remains. This fulfills the privacy requirement. The students can keep their individual block-chains with the grades. If, later on, a student wants to contest an erroneous entry, the student can produce the block-chain as a proof for the claimed mistake. The block-chain can then be checked using the hash code in the central system, and the goals of verifiability and privacy are achieved.

As second, much more popular application is the implementation of crypto-currencies For example, the crypto-currency Bitcoin relies on a block-chain that contains all the transactions in the bit-coin universe. The hash-references in the Block-chain ensure that even transactions that lie far in the past can be verified (and are resistant to modification) if the hash-code of the current node is known.

Talking about Bitcoin, we mention another application of cryptographic hash functions that is embedded in the Bitcoin block-chain: proof of work. In the bitcoin universe, any participant can add a new block at the head of the block-chain. The only requirement is that the hash code for the new had ends with k zeroes, where k is a parameter that is adjusted regularly. Since the value of the hash code is determined by the contents of the new head node, we need to add a new attribute to the head node whose value can be chosen arbitrarily and whose sole purpose is to influence the value of the hash code. This value is called *nonce*. By choosing the nonce appropriately, it will be possible to make sure that the hash value for the head block ends with a certain number of zeroes. However, for a cryptographic hash function, the only way to find such a value for the nonce is by trying many possible nonces and by checking the resulting hash values (if the hash function behaves randomly, we expect that we need to try 2^k nonces until we find the right value). Thus, we can only add a new block after investing enough computational effort into the Bitcoin system (this process is also called *mining*).

IV

Strings

KAPITEL 15

Strings

We will now turn our focus to data of a certain form: *strings*. Strings represent arbitrarily long sequences of *symbols*, e.g., text, source code, DNA information, or binary data. When dealing with strings, we face a number of specific algorithmic tasks and structural constraints. Thus, it make sense to study algorithms and data structures for strings in their own right. The present section presents only a glimpse at the wide variety of techniques that are available for dealing with string data. Later on, particularly in the bioinformatics curriculum, we will see much more on this topic.

Formal definitions and notation. First, we need to define some basic terms and fix the notation. When talking about strings, we always have an underlying *alphabet* that contains all the symbols that may be present in a given string.

An *alphabet* is a non-empty finite set. It is usually denoted by Σ. Typical examples of alphabets are $\Sigma = \{0,1\}$, the *binary alphabet*, containing only the symbols 0 and 1; the alphabet $\Sigma = \{a,\dots,z\}$, the *Latin alphabet* that consists of 26 letters; or $\Sigma = \{C,G,T,A\}$, the alphabet that is used to represent genome information.

Given an alphabet Σ, a *string over* Σ is a finite (ordered) sequence $w = \sigma_1\sigma_2\dots\sigma_n$ of symbols from Σ. The number of symbols in w is called the *length* of w. It is denoted by $|w|$. We specifically allow the string of length 0 that contains no symbols, the *empty string*. It is denoted by ε.[1] The set of all strings over Σ is denoted by Σ^*. This includes the empty string. If we would like to exclude the empty string, the set of all *non-empty* strings over Σ is denoted by Σ^+.

For example, possible strings over $\Sigma = \{0,1\}$ are 01001, 1101001, or 1, where $|01001| = 5$, $|1101001| = 7$, and $|1| = 1$. Possible strings over $\Sigma = \{C,G,T,A\}$ are $CGGGT$ or CGT, where $|CGGGT| = 5$ and $|CGT| = 3$.

We always have $|\varepsilon| = 0$, and for any alphabet Σ, we have $\varepsilon \in \Sigma^*$, $\varepsilon \notin \Sigma^+$ and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The set

$$\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \dots\}$$

is the set of all *bit strings* (including the empty bit string), and the set

$$\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, \dots\}$$

[1] It is important to note that ε is a symbol from the mathematical metalanguage. It denotes the string that contains no symbols, *not* the string that consists of the single symbol ε.

is the set of all *non-empty bit strings*. These two sets play an important role in computer science, since internally, all information in a modern computer is represented by bit strings.

TODO: Definitions substring, subsequence, prefix suffix

Problems on strings. In the next chapters, we will study the following chapters for strings:

- **Dictionaries for strings.** We have already talked in great detail about the problem of implementing an (ordered) *dictionary*. There are special dictionary structures for the case that the key set is a set of strings over a fixed alphabet. These dictionaries are called *tries* (or *digital search trees*). We will discuss them in the next chapter.

- **Efficient storage of strings.** Data in string form is typically very large. Given a string, is there a way to store Σ in such a way that no (or only little) information gets lost, while using as little space as possible? Methods that address this question are called *data compression*. We will see one approach to data compression in Chapter 17.

- **String search.** The string search problem is the following: given a string w and a pattern p, both over the same alphabet, we would like to determine whether p occurs in w as a substring. For example, the pattern “hund” appears in the string “schundliteratur”, but not in the string “schuhband”. The problem of string search comes in many variants and needs to be solved in many contexts. We will talk about it in Chapter 18.

- **String similarity.** Given two strings w_1 and w_2 over the same alphabet, how *similar* are w_1 and w_2? This is a natural questions that occurs, e.g., in bioinformatics when comparing two genome sequences, or when comparing two versions of a text file (say., in the linux utility `diff`). In order to answer this question, we first need a formal definition of mathematical similarity, and then we need an algorithm to implement this definition. In Chapter 19, we will see a way how this can be done.

KAPITEL 16

Dictionaries for Strings – Tries

In the previous sections, we saw many different ways to implement the abstract data types dictionary and ordered dictionary. Now, we will look at the problem of implementing the ADT ordered dictionary for the case that the key set $K = \Sigma^*$, for a fixed alphabet Σ.

TODO: Say something about lexicographic order.

Of course, we could just use any of the existing implementations that we saw in the previous chapters, e.g., AVL-trees, skiplists, (a,b)-trees, etc. These implementations work for general ordered key sets, so they can be used also for strings.

However, there are several reasons why one might want to construct special data structures for the case $K = \Sigma^*$. First, strings can become very long. Thus, it is not a realistic assumption that a comparison between two strings takes constant time, and we would like to have a dictionary structure that takes this special structure into account (and that is faster if the strings become very long). Second, it may be possible to obtain a simpler solution for this special case. Since the case of string keys is quite common, it makes sense to have special structures that are easier to understand and implement. Third, once we have a special dictionary structure for strings, one may extend it for more operations and applications. Since this case occurs often in application (particularly in computational biology), this can be very useful.

Tries. *Tries* are a simple way to implement an ordered dictionary for strings.[1]

Let Σ be an alphabet. A trie for Σ is a rooted multi-way tree T. Each inner node in T has at least 1 child and at most $|\Sigma|$ children, were the number of children can differ from node to node. For each inner node v, each edge that connects v to a child is labeled with a symbol from Σ, so that each symbol from Σ occurs at most once among the edges for v. For each key s that is stored in T, there is exactly one leaf w in T that represents s. We can obtain s by concatenating the labels along the path from the root of T to w. The value for the key s is stored in w.

TODO: Example

The structure as described has an obvious problem: suppose we have two keys s_1 and s_2 such that s_1 is a *prefix* of s_2, i.e., that s_1 coincides with a substring at the

[1] The name *trie* derives from the word re*trie*val, but it is pronounced like "try" and not like "tree". Tries are sometimes also called *radix trees* or *digital search trees*.

beginning of s_2. For example, this is the case for s_1 = HAND and s_2 = HANDSCHUH. Now, if we try to store both s_1 and s_2 in a trie T, we encounter a problem, because T is supposed to contain a leaf w for s_1, but then, the leaf w would also lie on the path to the leaf for s_2, thus we expect that w is both a leaf and an inner node, which is impossible.

There are at least two ways to solve this problem. First, we could simply abandon the requirement that the keys correspond to the leaves in the trie. Instead, we introduce an additional bit in each node that determines whether the node is the endpoint of a string or not. Then, the values can be stored both in the leaves and in the inner nodes. This is a feasible solution, but it creates special cases and complicates the algorithms. Thus, we prefer to use a second solution.

The second solution is to introduce a special symbol (usually denoted by \$) that does not appear in the original alphabet Σ. Whenever we insert a key s into the trie, we add the symbol \$ to the end of s. This ensure that now key can be a prefix of another key (because if $s_1\$$ were a prefix of $s_2\$$, it would be the case that s_2 contains the symbol \$ in the middle, which is impossible, since $\$ \notin \Sigma$). The advantage of this solution is that it keeps the structure and the algorithms simple. It is another example of the *sentinel technique* that we have already encountered in *Konzepte der Programmierung* when we talked about different implementations of linked lists.[2]

TODO: Add an example of a trie with \$.

Implementation issues. There are different ways to implement a trie. In particular, we need to have a way to represent the children of a node. If Σ is small, it is feasible to store a fixed array that has a position for every symbol in Σ and that stores a reference to the child node for that symbol, if it exists, and $\perp$, otherwise. If Σ is large, this solution becomes infeasible, and we may need to store a small dictionary that stores entries of the form (σ, v), where σ is a symbol for which a child exists, and v is a reference to the child node for σ. When analyzing our algorithms, we assume that in each node, we spend $|\Sigma|$ steps to locate a given child node. Using a better dictionary structure, this running time can be be improved.

Operations. Tries support all the operations of the ADT ordered dictionary. The details are left as an exercise.

Analysis. Suppose we have a trie T that stores a set S of entries. Then, the total space for T is at most $O(\sum_{(s,v)\in S} |s|)$, the total length of the keys that appear in S. The space can be better if the keys in S share many prefixes.

The operations $\texttt{put}(s,v)$, $\texttt{get}(s)$, and $\texttt{remove}(s)$ can be implemented in time $O(|s|\cdot|\Sigma|)$. The operations $\texttt{pred}(s)$ and $\texttt{succ}(s)$ can be implemented in time $O((|s|+|s'|)\cdot|\Sigma|)$, where

[2] The symbol \$ is used in several places in computer science to mark the end of a string. Historically, this comes from the old PDP-minicomputers that used a very restricted character set, so that \$ was a very natural choice for this purpose.

s' is the predecessor or successor of s, depending on whether we consider `pred` or `succ`.

Compressed tries. One possible issue with tries is that they create a node for every symbol in an input key, even if there are no branches in the tree. Thus, when resolving a key, may need to follow a long chain of node objects, even though there is no branch that needs to be resolved. This may cause unnecessary overhead in practice, because following an object reference can be slow in modern computer systems (compared to, say, scanning an array).

Compressed tries are a way to address this issue. In a compressed trie, we contract every long path that consists of nodes that have only one child each into a single edge. This edge is labelled with the whole substring that corresponds to the path. Nodes are only introduced when a branch happens. As before, we introduce a special symbol $\$ \notin \Sigma$ to mark the end of the strings in the compressed trie, to make sure that every key corresponds to a leaf.

The operations are implemented as before, but now we may need to split an edge after an insertion, if a new branch becomes necessary, or to merge an edge after a deletion, if a branch disappears. Since we still need to store all the keys, the space requirement for a compressed trie that stores a set S of entries is still $O(\sum_{(s,v)\in S}|s|)$, but now the tree has only $O(|S|)$ nodes, which is potentially more efficient in practice.

To distinguish them from compressed tries, we also refer to the usual tries from before as *uncompressed* tries. An alternative name for compressed tries is PATRICIA tries, where PATRICIA is an acronym for Practical Algorithm To Retrieve Information Coded In Alphanumeric.

In the bioinformatics lectures you will learn about string matching for multiple strings. One algorithm, the Aho-Corasick Algoriothms uses a trie for searching a string simultaneously for many patterns.

Suffix trees. Let $s \in \Sigma^*$ be a long string over some alphabet Σ. A *suffix tree* for s is a compressed trie that stores all the suffixes (i.e., all end strings) of the string $s\$$.

TODO: Example

Suffix trees are a very useful data structure that allow us to solve many tasks that concern the string s efficiently. For example, suppose that s is a very long string, and that we would like to carry out many *string searches* on s. That is, we would like to be able to perform the following query: given a string $t \in \Sigma^*$, does t occur in s as a substring?[3] If a suffix tree T for s is available, this is easy: the crucial observation is that t is contained in s if any only if s has a suffix that begins with t. Thus, all we need to do is search for t in T. If this search ends in an inner node (or a leaf), then t appears in s. Even more, the leaves under the node where our search ends give us the locations of *all* occurrences of t in s. This is only one example, there are many other applications of suffix trees that will be treated in later classes.

[3] We will talk more about string searching in Chapter 18.

Since the suffix tree T is a compressed trie and since $s\$$ as $|s|+1$ suffixes, the number of nodes in T is $O(|s|)$. However, if we store the edge labels explicitly, the space requirement for T can become $\Omega(|s|^2)$, which is too much if s is long. To fix this, we observe that all edge labels in T are substrings of T. Thus, instead of storing the edge labels explicitly, it suffices to store the start and the end position of the corresponding substring in s. This reduces the space requirement for the suffix tree to $O(|s|)$, which is asymptotically the same as storing s itself.

Given s, the naive construction algorithm that successively inserts all suffixes of s into a compressed trie may take time $\Omega(|s|^2)$. However, there are much better ways to construct a suffix tree for s. In fact, it is possible to construct the tree in time $O(|s|)$, which is as good as we can hope for. This method will be discussed in a later class.

Note that the suffix tree has a rather high memory consumption in practice. As an alternative for it, the bioinformaticians will learn about the suffix array and later about compact suffix arrays.

KAPITEL 17

Data Compression

Now we consider the problem of representing a string over some alphabet Σ "efficiently" in memory.

For example, consider the alphabet $\Sigma = \{a, b, c, d\}$ and the string $w = aaaaabbaaacd$ over Σ.

What is a "good" way to represent w in a computer? For this, we first note that in today's computer systems, data is represented in binary. That is, we need to find a way to encode w as a bit string (an element from $\{0,1\}^*$) that is "good". What exactly we mean by "good" depends on the context, but here are a few typical goals that we would like to achieve:

- **Space efficiency:** The bit string that encodes w should be as short as possible.
- **Fast encoding and decoding:** If should be possible to derive quickly the contents of w from the bit encoding.
- **Uniqueness:** It should be possible to reconstruct w from its bit encoding without ambiguities.

A simple encoding scheme would proceed like this: suppose that Σ consists or $k \geq 2$ symbols, and let $\ell = \lfloor \log(k-1) \rfloor + 1$. We number the elements from Σ arbitrarily from 0 to $k-1$. These numbers can be represented with ℓ bits. Now encode each symbol with the ℓ-bit binary representation of its number, filled with leading zeroes. To encode the whole string w, we encode each symbol individually with the corresponding bit string, and we concatenate the strings.

In our example, we have $k = 4$ and $\ell = 2$. We might number $a : 0, b : 1, c : 2, d : 3$ and assign the binary encodings $a : 00, b : 01, c : 10, d : 11$. The representation for $w = aaaaabbaaacd$ is then 000000000001010000000111.

This bit representation is called the *block code*, and it has many advantages: it is simple, regular, and fast. In many scenarios, it cannot be improved. This is why it is used widely throughout computer science. However, with our example string w, one could also imagine different, more efficient representations.

One idea would be to abandon the idea all the binary encodings for the symbols of Σ need to have the same length. Then, we could, e.g., use a shorter bit string to represent a, which occurs very frequently, at the expense of using longer bit strings for, say, c

and d, which occur only once. For example, if we assign $a : 0, b : 10, c : 110, d : 111$, we could represent $w = aaaaabbaaacd$ as 000001010000110111, which needs only 18 instead of 26 bits. The disadvantage is that the encoding is less regular: without know the contents of w, we cannot predict where in the bit string the, say, tenth symbol of w will lie. But for pure representation purposes, this is certainly more efficient than the block code.

Another idea would be to exploit patterns in the string w. For example, we notice that w contains long sequences where the symbol $a$is repeated (such a sequence is called a *run*. Instead of storing w explicitly, we could store the information that w consists of 5 a's, followed by 2 b's, followed by 3 a's, followed by 1 c and 1 d. Again, the representation is not as explicit and regular as in the block code, but if the input string contains many long runs, the space savings may be significant.

The field of computer science that explores these ideas in more details is called *data compression*.

Data compression. The general goal of data compression is this: we are given a string w over some alphabet Σ. The goal is to represent w as a bit string that is as short as possible, while retaining the goals of efficiency and generality for the encoding and decoding algorithms. The string w can be of many different types: it might represent text, program code, audio, pictures (photos and computer generated), videos, etc. Depending on the type of data, different approaches are possible:

- **Vary encoding length:** Encode the symbols of Σ individually by bit strings. Vary the length of the bit strings depending on the frequency with which a symbol occurs in w: more frequent symbols receive shorter bit strings, less frequent symbols receive longer bit strings.

- **Exploit redundancies:** Look for repeating patterns in w and try to use shortcuts to represent them more efficiently. One such pattern are long *runs* (the same symbol is repeated many times), and one may represent the pattern by simply storing the length of the run and the symbol. This method is called *run-length encoding*, and it is often used to represent image data for images with few colors). Another pattern occurs if the same word or phrase is used very often in a text-file. Then, one might introduce a special symbol to represent this phrase, along with a dictionary that translates between these symbols and the corresponding phrases. This idea is exploited, e.g., by the Lempel-Ziv-Welch family of compression algorithms, and is one of the main methods for file compression.

- **Lossy compression:** In certain kinds of data (photos, videos, sound), it may be acceptable if some information gets lost in the compression, as long as the main features are preserved. In this case, we can save a lot of space by discarding "unnecessary" information. The main task is now to identify which information is relevant and which information is redundant. There are many ways to do this, typically exploiting mathematical operations like the Fourier transformation, the discrete cosine transformation, or the wavelet transformation. This techniques are the main way to represent photos (e.g., JPEG), sound (e.g., MP3) and videos (e.g., MPEG-4).

In this class, we will look closer at Huffman-codes, a data compression method that exploits the first idea: varying the coding length. The other methods are treated in later classes on data compression.

Prefix-free codes and Huffman codes. Before discussing Huffman-codes, we first need some basic definitions and facts about codes. Let Σ be an alphabet, such that Σ contains at least two symbols.[1]

A *code* for Σ is an injective function $C : \Sigma \to \{0,1\}^+$ that assigns to each symbol of Σ a unique non-empty bit string. The images of a code are called *codewords*.

For example, for code for the alphabet $\Sigma = \{a,b,c,d\}$ could be $C(a) = 0$, $C(b) = 10$, $C(c) = 110$, $c(d) = 111$.

Given an alphabet Σ and a code $C : \Sigma \to \{0,1\}^+$, we encode a string $w \in \Sigma^*$ by concatenating the codes for the individual symbols in w: if $w = \sigma_1\sigma_2\dots\sigma_\ell$, then $C(w) = C(\sigma_1)C(\sigma_2)\dots C(\sigma_\ell)$. This defines a function $\widehat{C} : \Sigma^+ \to \{0,1\}^+$.

For example, using the code above, we would encode $w = aabc$ as $\widehat{C}(w) = C(a)C(a)C(b)C(c) = 0010110$.

To have a useful code, it must be possible to reconstruct the original string w from the encoding $C(w)$. For this, the code needs to be *uniquely decodable*, that is, for any two distinct strings $w_1, w_2 \in \Sigma^*$, $w_1 \neq w_2$, it must be the case that $C(w_1) \neq C(w_2)$. In other words, not only should the code $C : \Sigma \to \{0,1\}^+$ be injective, but also the extended encoding function $\widehat{C} : \Sigma^+ \to \{0,1\}^+$.

For example, let $\Sigma = \{a,b\}$ and let $C(a) = 0$ and $C(b) = 00$. Then, the code C is not uniquely decodable, because $C(ab) = C(ba) = 000$, so the encodings of ab and ba are identical, even though $ab \neq ba$.

A useful condition that ensures that a code is uniquely decodable is *prefix-freeness*. A code $C : \Sigma \to \{0,1\}^+$ is called *prefix-free* if for any pair $\sigma, \tau \in \Sigma$ of distinct symbols in Σ, we have that neither $C(\sigma)$ is a prefix of $C(\tau)$ nor $C(\tau)$ is a prefix of $C(\sigma)$.[2]

For example, let $\Sigma = \{a,b,c,d\}$. Then, the code $C_1 : \Sigma \to \{0,1\}^+$ with $C_1(a) = 00$, $C_1(b) = 101$, $C_1(c) = 11$, and $C_1(d) = 0101$ is prefix-free, but the code $C_2 : \Sigma \to \{0,1\}^+$ with $C_2(a) = 010$, $C_2(b) = 0110$, $C_2(c) = 01010$ and $C(d) = 0$ is not.

Satz 17.1. *Let Σ be an alphabet and let $C : \Sigma \to \{0,1\}^+$ be a prefix-free code. Then, C is uniquely decodable.*

Beweis. We will prove the contrapositive of the statement: if C is not uniquely decodable, then C is not prefix-free.

[1] If Σ has only one symbol, data compression is not too interesting, since then we could represent each string by its length, leading to exponential space savings.

[2] A prefix-free code is sometimes also called a *prefix code*.

Thus, suppose that C is not uniquely decodable. Then, by definition, there are two distinct strings $w_1, w_2 \in \Sigma^+$, $w_1 \neq w_2$, such that $C(w_1) = C(w_2)$. Let σ be the first symbol of w_1 and τ be the first symbol of w_2. We can assume that $\sigma \neq \tau$, because if $\sigma = \tau$, we can remove the first symbol from w_1 and from w_2, and we would still get two words $w_1 \neq w_2$ with $C(w_1) = C(w_2)$.

Now, since C is injective, $C(\sigma)$ and $C(\tau)$ must be different. Furthermore, since $C(w_1) = C(w_2)$, it must be the case that $C(w_1)$ starts with both $C(\sigma)$ and $C(\tau)$. This implies that $C(\sigma)$ is a proper prefix of $C(\tau)$, or vice versa. Hence, C is not prefix-free. □

The proof also suggests an efficient greedy algorithm for decoding a bit string $s \in \{0,1\}^+$ that has been encoded with a prefix-free code C: we read s from the beginning, collecting bits until we have obtained a codeword from C. We output the corresponding symbol, and we repeat this process with the remaining bits. Since the code is prefix-free, we can be sure that when we output a symbol, there is no other codeword that could have started at the beginning of s.

TODO: Example

We emphasize that every prefix-free code is uniquely decodable, but not every uniquely decodable code is prefix-free. For example, let $\Sigma = \{a, b\}$ and let $C : \Sigma \to \{0,1\}^+$ with $C(a) = 0$ and $C(b) = 010$. Then, the code C is not prefix-free (0 is a prefix of 010), but it is uniquely decodable (in an encoded bitstring, we can use the 1's to identify the codewords for b. The remaining 0's must be the codewords for a).

Prefix-free codes naturally correspond to rooted binary trees: let $C : \Sigma \to \{0,1\}^+$ be a prefix-free code, and represent all the codewords $C(\sigma)$, $\sigma \in \Sigma$, in an uncompressed trie T (without adding the trailing \$-symbol). Since T is over the binary alphabet $\{0,1\}$, it is a binary tree. We order the children such that the edges to the left children are labeled with 0 and the edges to the right children are labeled with 1. Since C is prefix-free, we can be sure that the leaves of T correspond exactly to the codewords of C. The tree T is called the *code tree* for C.

Huffman-Codes. We can now formally state the algorithmic problem: we are given an alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$, with $n \geq 2$, and associated *frequencies* $h_{\sigma_1}, h_{\sigma_2}, \dots, h_{\sigma_n} \in \mathbb{N}_0$. The frequencies indicate that we have a string in which σ_1 appears h_1 times, σ_2 appears h_2 times, etc. The goal is to construct a prefix-free code $C : \Sigma \to \{0,1\}^+$ such that the *total length* of C,

$$\ell(C) = \sum_{\sigma \in \Sigma} h_\sigma \cdot |C(\sigma)|$$

is minimum among all possible prefix-free codes for Σ. Recall that $|C(\sigma)|$ denotes the number of bits for $C(\sigma)$ and hence $\ell(C)$ is simply the number of bits that are needed to encode a string that leads to the given frequencies. In the following, we will also use the notation $\ell(T)$ to denote the *total length* of a code tree T, which is the total length of the prefix-free code that corresponds to T.

TODO: Example

The idea of the Huffman-algorithm is to construct a code tree for the desired code C bottom-up, in a greedy fashion.

We create a leaf-node for each symbol in Σ, and we annotate the leaf-node with the symbol and the corresponding frequency. In the following, we will not distinguish between the leaf-node and the corresponding symbol, to make the notation easier.

We choose two nodes σ, τ with the smallest frequencies (that is, in $\Sigma \setminus \{\sigma, \tau\}$ there is no symbol with a frequency that is smaller than h_σ or h_τ), and we join them together to a partial code tree. More precisely, we create a new inner node v, and we make the node for σ the left child of v and the node for τ the right node of v. We annotate v with the frequency $h_\sigma + h_\tau$.

We repeat this process, generalizing from nodes to trees: in each step, we choose two partial code trees with smallest frequencies, and we unite them to a new partial code tree (by adding a new parent node), whose frequency is the sum of the frequencies for its subtrees.

The process ends once only one tree remains. This is a code tree for Σ, and we can derive a prefix-free code $C_H : \Sigma \to \{0,1\}^*$. This prefix-free code is called the *Huffman code*. We will see that it is an optimal prefix-free code for Σ and the frequencies h_σ.

Before we can prove that Huffman codes are actually optimal, we first need two lemmas that show that the first step of the Huffman algorithm is correct. More precisely, in the first step, the Huffman algorithm takes two leaves with the smallest frequencies and unites them to a partial code tree. The lemmas state that there is always an optimal code tree in which this partial code tree occurs. First, we argue that there is always an optimal code tree in which every leaf has a sibling node.

Lemma 17.2 (Sibling lemma). *Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be an alphabet with $n \geq 2$, and let $h_{\sigma_1}, \ldots, h_{\sigma_n} \in \mathbb{N}_0$ be associated frequencies.*

There is an optimal code tree T^* for Σ and the frequencies h_σ such that every leaf in T^* has a sibling (this sibling may be an inner node or another leaf).

Beweis. Let T_1 be an optimal code tree for Σ and the frequencies h_σ. If every leaf in T_1 has a sibling, we set $T^* = T_1$ and are done. Otherwise, the tree T_1 contains a leaf that does not have a sibling. We call this leaf σ. Since $n \geq 2$, the leaf σ must have a parent node w, and σ is the only child of w. We remove the parent node w from T_1, and we put the leaf σ in its place. We call the resulting tree T_2. Then, T_2 is a valid code tree for Σ. In T_2, the length of the code word for σ has decreased by one, while all other code words are unchanged. Thus, we have $\ell(T_2) \leq \ell(T_1)$, and T_2 is also an optimal code tree for Σ and the frequencies h_σ.[3]

Now, if every leaf in T_2 has a sibling, we set $T^* = T_2$ and are done. Otherwise, we repeat. Eventually, this will terminate, because in each step, a leaf moves closer to the root. In the end, we obtain an optimal code tree T^* in which every leaf has a sibling, as desired. □

[3] In fact, this situation can occur only if $h_\sigma = 0$, since otherwise we would get $\ell(T_2) < \ell(T_1)$, which is impossible if T_1 is optimal.

Second, we show that there is an optimal code in which the first two nodes that are chosen by the Huffman algorithm are siblings.

Lemma 17.3 (Exchange lemma). *Let $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ be an alphabet with $n \geq 2$, and let $h_{\sigma_1}, \dots, h_{\sigma_n} \in \mathbb{N}_0$ be associated frequencies. Let $\alpha, \beta \in \Sigma$ be two symbols such that $h_\alpha = \min_{\sigma\in\Sigma} h_\sigma$ and $h_\beta = \min_{\sigma\in\Sigma\setminus\{\alpha\}} h_\sigma$, that is, α has a smallest frequency and β has a second smallest frequency in Σ.*

Then, there exists an optimal code tree T^* for Σ and the frequencies h_σ such that α and β occur as siblings in T^*.

Beweis. By Lemma 17.2, there exists an optimal code T in which every leaf has a sibling. Let γ be a leaf in T that has maximum depth (i.e., no leaf in T is deeper than γ). By assumption, γ has a sibling, and because γ was chosen with maximum depth, the sibling must also be a leaf. We call this sibling leaf δ, and we assume that the names are chosen such that $h_\gamma \leq h_\delta$. We exchange in T_1 the leaf for α with the leaf for γ, and the leaf for β with the leaf for δ. We call the resulting tree T^*. Then, T^* is a valid code tree for Σ.

In T^*, the lengths of the codewords for α and β may get larger. In return, the lengths of the codewords for γ and δ may get smaller. We will see that since α and β are two symbols with the smallest frequencies, the total length cannot get larger, so that T^* is still optimal.

More precisely, we will show by a calculation that $\ell(T^*) - \ell(T) \leq 0$. For this, let C be the code for T, and C^* the code for T^*. We have obtained C^* from C by exchanging the codewords of α and γ, and the codewords for β and δ. All other codewords are unchanged. Thus, we have

$$\begin{aligned}
&\ell(T_2) - \ell(T_1) \\
&= \sum_{\sigma\in\Sigma} h_\sigma \cdot |C^*(\sigma)| - \sum_{\sigma\in\Sigma} h_\sigma \cdot |C(\sigma)| && \text{(by definition)} \\
&= \sum_{\sigma\in\Sigma} h_\sigma \cdot (|C^*(\sigma)| - |C(\sigma)|) && \text{(rearranging the sums)} \\
&= \sum_{\sigma\in\{\alpha,\beta,\gamma,\delta\}} h_\sigma \cdot (|C^*(\sigma)| - |C(\sigma)|) && \text{(only } \alpha,\beta,\gamma,\delta \text{ changed)}
\end{aligned}$$

By construction of T^*, we have $C^*(\alpha) = C(\gamma)$, $C^*(\gamma) = C(\alpha)$, and $C^*(\beta) = C(\delta)$, $C^*(\delta) = C(\beta)$. Thus, this is

$$\begin{aligned}
&= h_\alpha \cdot (|C(\gamma)| - |C(\alpha)|) + h_\beta \cdot (|C(\delta)| - |C(\beta)|) \\
&\quad + h_\gamma \cdot (|C(\alpha)| - |C(\gamma)|) + h_\delta \cdot (|C(\beta)| - |C(\delta)|) \\
&= h_\alpha \cdot (|C(\gamma)| - |C(\alpha)|) + h_\beta \cdot (|C(\delta)| - |C(\beta)|) \\
&\quad - h_\gamma \cdot (|C(\gamma)| - |C(\alpha)|) - h_\delta \cdot (|C(\delta)| - |C(\beta)|) && \text{(switching signs)} \\
&= (h_\alpha - h_\gamma) \cdot (|C(\gamma)| - |C(\alpha)|) \\
&\quad + (h_\beta - h_\delta) \cdot (|C(\delta)| - |C(\beta)|). && \text{(rearranging)}
\end{aligned}$$

Since α is a symbol with minimum frequency, we have

$$h_\alpha \leq h_\gamma \quad \Rightarrow \quad h_\alpha - h_\gamma \leq 0.$$

Since γ is a leaf of maximum depth in T, we have

$$|C(\gamma)| \geq |C(\alpha)| \quad \Rightarrow \quad |C(\gamma)| - |C(\alpha)| \geq 0.$$

Together, this gives

$$(h_\alpha - h_\gamma) \cdot (|C(\gamma)| - |C(\alpha)|) \leq 0.$$

Similarly, since β is a symbol with second smallest frequency, and since $h_\delta \geq h_\gamma$ we have

$$h_\beta \leq h_\delta \quad \Rightarrow \quad h_\beta - h_\delta \leq 0.$$

Since γ is a leaf of maximum depth in T, we have

$$|C(\delta)| \geq |C(\beta)| \quad \Rightarrow \quad |C(\delta)| - |C(\beta)| \geq 0.$$

Together, this gives

$$(h_\beta - h_\delta) \cdot (|C(\delta)| - |C(\beta)|) \leq 0.$$

In total, we have

$$\ell(T^*) - \ell(T) \leq 0 \quad \Rightarrow \quad \ell(T^*) \leq \ell(T).$$

Thus, T^* is optimal, and the lemma follows. □

We can now show that the Huffman-algorithm is indeed optimal.

Satz 17.4. *Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be an alphabet with $n \geq 2$, and let $h_{\sigma_1}, \ldots, h_{\sigma_n} \in \mathbb{N}_0$ be associated frequencies. Then, the Huffman-algorithm produces a prefix-free code that minimizes the total length for the given frequencies, among all prefix-free codes for Σ.*

Beweis. The proof uses induction on n, the number of symbols in Σ. For the base case, we have $n = 2$. In this case, Σ consists of two symbols, say $\Sigma = \{\sigma, \tau\}$. The Huffman-code for Σ could be either C_1 with $C_1(\sigma) = 0$ and $C_1(\tau) = 1$, or C_2 with $C_2(\sigma) = 1$ and $C_2(\tau) = 0$. In either case, both codewords have length 1, and the total length is $h_\sigma + h_\tau$. Let C be an arbitrary prefix-free code for Σ. By definition of a code, all codewords in C must be nonempty, so they have length at least 1. Thus, the total length of C is $h_\sigma \cdot |C(\sigma)| + h_\tau \cdot |C(\tau)| \geq h_\sigma + h_\tau$. It follows that the Huffman-code is optimal.

For the inductive step, we go from $n-1$ to n. Our inductive hypothesis states that for every alphabet with $n-1$ symbols and for every sequence of associated frequencies, the Huffman-algorithm constructs an optimal prefix-free code. We need to show that the same holds for every alphabet with n symbols and for every set of associated frequencies.

Thus, let Σ be an alphabet with n symbols, and let h_σ, $\sigma \in \Sigma$ be a sequence of associated frequencies. Suppose we run the Huffman-algorithm for Σ and the h_σ. Let T_H be the resulting code tree, and let α, β be the two symbols that are chosen first by the Huffman-algorithm. By Lemma 17.3, there is an optimal code tree T_O for Σ and the h_σ such that α and β occur as siblings in T_O.

Now, we need to apply the inductive hypothesis. For this, let $\rho \notin \Sigma$ be a new symbol that does not occur in Σ. Let Σ' be the alphabet is obtained by removing α and β from Σ and by adding ρ. That is,

$$\Sigma' = (\Sigma \setminus \{\alpha, \beta\}) \cup \{\rho\}.$$

Furthermore, we set $h'_\rho = h_\alpha + h_\beta$, and $h'_\sigma = h_\sigma$ for all $\sigma \in \Sigma \setminus \{\alpha, \beta\}$. Then, Σ' is an alphabet with $n-1$ symbols, and with associated frequencies h'_σ.

Now, we modify the tree T_H as follows: we remove the children for α and β. Since α and β were chosen first by the Huffman-algorithm, the leaves for α and β are siblings in T_H, and they have a common parent node w. Now, after removing the leaves for α and β, the parent node w is a leaf, and we label it with the new symbol ρ. We call the resulting tree T'_H. Then, T'_H is a code tree for Σ'. Even more, crucially, the tree T'_H is actually a Huffman tree for Σ' and the frequencies h'_σ. This is because if we run the Huffman-algorithm on Σ' and the frequencies h'_σ, it behaves in the same way as the Huffman-algorithm on Σ and the frequencies h_σ after the first step.

Next, we modify the tree T_O in the same way: since the leaves for α and β are siblings in T_O, we can remove them and turn their common parent into a leaf for the new symbol ρ. We call the resulting tree T'_O. It is a code tree for Σ'.

We can now apply the inductive hypothesis. Since T'_H is a Huffman-tree for Σ' and the frequencies h'_σ, we can conclude by induction that T'_H is optimal. In particular, since T'_O is also a code tree for Σ', this means that

$$\ell(T'_H) \leq \ell(T'_O).$$

Now, by construction of T'_H and T'_O, we have that $\ell(T_H) = \ell(T'_H) + h_\alpha + h_\beta$, and $\ell(T_O) = \ell(T'_O) + h_\alpha + h_\beta$. Indeed, in both T_H and T_O, the codewords for α and β have the same length, and in T'_H and T'_O, they are replaced by a single code word that is shorter by one bit and that has frequency $h_\alpha + h_\beta$. Thus, we can conclude that

$$\ell(T_H) = \ell(T'_H) + h_\alpha + h_\beta \leq \ell(T'_O) + h_\alpha + h_\beta = \ell(T_O).$$

Since T_O is an optimal tree, it now follows that $\ell(T_H) = \ell(T_O)$, and hence the Huffman tree is also optimal. This concludes the proof. □

KAPITEL 18

String Search

Now, we turn to the problem of *string search*. This is the following problem: let Σ be an alphabet. We are given a (relatively short) string $t = \tau_1\tau_2\dots\tau_\ell$, the *pattern*, and a (much longer) string $s = \sigma_1\sigma_2\dots\sigma_k$, the *text*. Both s and t are strings over Σ, and we have $\ell \le k$. The goal is to determine whether t *occurs as a substring* in s, and, if so, to find the index of the first such occurrence. Formally, this means that we would like to find the smallest index i between 1 and $k - \ell + 1$ such that $\sigma_{i+j-1} = \tau_j$, for all $j = 1,\dots,\ell$, or to determine that no such index exists.

The naive algorithm. The string search problem can be solved by a very simple algorithm that is immediate from the problem definition – the *naive* algorithm. For each index i between 1 and $k - \ell + 1$, we check whether t occurs as a substring in s *at position* i. For this, we compare the characters of s, starting with σ_i, with the corresponding characters in t, starting with τ_1, one by one. If two corresponding characters do not match, we stop and continue with the next i (this is called a *mismatch*). Otherwise, if all characters match, we know that $\sigma_i \dots \sigma_{i+\ell-1} = t$, and we report that the first occurrence of t in s is at index i. If we find a mismatch for each index i, we report that t does not occur in s.

The pseudo-code is as follows:

```
for i := 1 to k - l + 1 do
    // Does s contain t at position i?
    j := 1
    while j <= l and s[i + j - 1] == t[j] do
        j++
    // If yes, return position i
    if j == l + 1 then
         return i
// Not found, return -1
return -1
```

The **for**-loop of the naive algorithm has $O(k)$ iterations, and in each such iteration, the **while**-loop runs until the first mismatch is found. Since in each step of the **while**-loop we test check another character of t, the number of iterations of the **while**-loop is $O(\ell)$. Thus, the running time of the naive algorithm is $O(k\ell)$.

This worst case can actually occur. For example, consider the case that $\Sigma = \{0, 1\}$ and that $s = 0^{k-1}1$ and $t = 0^{\ell-1}1$. That is, s consists of $k - 1$ zeroes, followed by a single 1, and t consists of $\ell - 1$ zeroes, followed by a single 1. Then, t occurs as a substring in s, but only at the very end. However, each execution of the **while**-loop runs for ℓ iterations, because the mismatch occurs at the very end of the pattern t. Thus, in the worst case, the naive algorithm has running time $\Theta(k\ell)$. We will now see two algorithms that improve this.

Knuth-Morris-Pratt. When looking at the naive algorithm, we see that it has an obvious inefficiency: whenever a mismatch is found, the naive algorithm simply goes to the next index i and begins to compare t with the substring $\sigma_i\sigma_{i+1}\dots$ from scratch. However, suppose that the **while**-loop finds a mismatch after j iterations. Then, we have learned that $\sigma_i \dots \sigma_{i+j-1} = \tau_1 \dots \tau_{j-1}$. Can we somehow take advantage of this information?

For example, suppose that t = STUDENT and that s = STUDIENSTUDENT. Then, the **while**-loop will encounter the first mismatch when comparing σ_5 = I to τ_5 = E. Now, because we started comparing at σ_1 and because we countered the first mismatch at σ_5, we have learned that $\sigma_1 \dots \sigma_4 = \tau_1 \dots \tau_4$ = STUD. Furthermore, the first character of t, τ_1 = S does not occur anywhere else in t. Thus, from the comparisons so far, we can not only conclude that the pattern t does not occur at position 1 in t, but also that t cannot occur at positions 2, 3, and 4. Thus, we can save the comparisons between τ_1 and σ_2, σ_3, and σ_4 that the naive algorithm would make. Instead, we can continue with comparing σ_5 to τ_1, because the next possible index where t can occur in s is 5 (this possibility was not excluded by the information that $\sigma_5 \neq \tau_5$).

Now, to generalize this idea, suppose that t = BARBARA and s = BARBARBARBARA. The first mismatch occurs when comparing σ_7 = B to $\tau_7 = A$. Now, we know that t cannot occur at index 1 in s. What is the smallest index where t could occur? Since we have learned that $\sigma_1 \dots \sigma_6 = \tau_1 \dots \tau_6$ = BARBAR, we can deduce that the first possible occurrence of t in s can be at index 4: this is the first position after 1 where there is a prefix of t = BARBARA that goes until the end of substring of successful comparisons. Thus, we can continue by comparing σ_7 = B with τ_4 = B. If we do this, the next mismatch occurs when comparing σ_{10} = B with τ_7 = A. Similarly to before, we have learned from the comparisons so far that $\sigma_4 \dots \sigma_{10} = \tau_1 \dots \tau_6$ = BARBAR. From this, we can now conclude that the next possible occurrence of t in s can only be at index 7, and we can continue by comparing σ_{11} to τ_4. From here, no more mismatches occur, and we have discovered t in s at index 7.

Thus, the idea is as follows: when we encounter a mismatch, we use the information from the successful comparisons to deduce the next possible occurrence for t in s. We adjust the current character in t accordingly and we continue at the same position in s where the mismatch occurred.

To formalize this idea, we need the following definition:

Definition 18.1. *Let $w, r \in \Sigma^*$. We say that r is a* **boundary** *of w if r is both a prefix and a suffix of w. Trivially, the string w itself and the empty string ε are always boundaries of w. A boundary r of w is called* **proper** *if $r \neq w$. We use $\partial(w)$ to denote the longest proper boundary of w.*

For example, we have $\partial(\text{BARBAR}) = \text{BAR}$, $\partial(\text{UNGLEICHUNG}) = \text{UNG}$, as well as $\partial(\text{AABBAABBAA}) = \text{AABBAA}$.

Now, suppose that we encounter a mismatch when comparing σ_i with τ_j. Then, if $j = 1$, we simply continue by comparing σ_{i+1} with τ_1. Otherwise, we know that $\sigma_{i-j+1} \dots \sigma_{i-1} = \tau_1 \dots \tau_{j-1}$, and the next possible match between σ and τ could start with $\partial(\tau_1 \dots \tau_{j-1})$ at position $i - |\partial(\tau_1 \dots \tau_{j-1})|$. To continue, we need to compare σ_i with $\tau_{|\partial(\tau_1 \dots \tau_{j-1})|+1}$.

To implement this, we define a function bd as $\text{bd}(j) = -1$, if $j = 1$, and $\text{bd}(j) = |\partial(\tau_1 \dots \tau_{j-1})|$, for $j = 2, \dots, \ell$. Using bd, we can write pseudocode for the algorithm as follows:

```
// we start with the first character in t
j := 1
for i := 1 to k - l + 1 do
    // if we have a mismatch, we use bd to try the next boundary
    // of t[1:j-1]. We repeat until the first successful
    // comparison or until we have  a mismatch with the first
    // character of t
    while j > 0 and s[i] <> t[j] do
        j := bd[j] + 1
    // advance in t
    j++
    // if we have reached the end of t, we return the index
    // of the occurrence
    if j == l + 1 then
        return i - l + 1
// Not found, return -1
return -1
```

To analyze the running time, we note that the **for**-loop needs at most k iterations. In each iteration of the **for**-loop, we spend constant time, plus the time for the iterations of the **while**-loop. The iterations of the while loop are due to mismatches $\sigma_i \neq \tau_j$. Every time we encounter a mismatch, the pattern t is moved to the right, i.e., the start index $i - j + 1$ of the pattern t increases. This can happen at most k times, since this is the total length of s. Thus, the total running time of the algorithm is $O(s)$, provide that the function bd is known.

To determine bd, we can use a recursive strategy. By definition, we have $\text{bd}(1) = -1$ and $\text{bd}(2) = 0$. Now, suppose that we know $\text{bd}(i) = j - 1$, for some i between 2 and $\ell - 1$. This means that $\tau_1 \dots \tau_{j-1}$ is the longest boundary of $\tau_1 \dots \tau_{i-1}$. Now, if $\tau_j = \tau_i$, then it follows that we can just extend the boundary and that $\tau_1 \dots \tau_j$ is the longest boundary of $\tau_1 \dots \tau_i$. Thus, in this case, we can set $\text{bd}(i + 1) = j$. On the other hand, if $\tau_j \neq \tau_i$, we need to try the *second* longest boundary of $\tau_1 \dots \tau_{i-1}$ to see if it can be extended by τ_i. The crucial observation is that the second longest boundary of $\tau_1 \dots \tau_{i-1}$ is the longest boundary of $\tau_1 \dots \tau_{j-1}$. This is already computed in $\mathrm{bd}(j)$. Thus, we can set j to $\mathrm{bd}(j)+1$, and we can continue by comparing τ_j to τ_i.

```
bd[1] := -1
bd[2] :=  0
j := 1
for i := 2 to l - 1 do
    while j > 0 and t[i] <> t[j] do
        j := bd[j] + 1
    bd[i+1] := j
    j++
```

This algorithm is actually almost identical to the string search algorithm above, with the main difference that the pattern t is compared to itself. Thus, the running time for computing bd is $O(\ell)$, and the total running time of the KMP-algorithm is $O(k+\ell)$.

Rabin-Karp. An alternative method for the string search problem was proposed by Rabin and Karp.

The bottleneck in the naive algorithm is that in each iteration of the **for**-loop, we might compare almost the complete substring $\sigma_i \dots \sigma_{i+\ell-1}$ with t, because the mismatch can occur very late. Now, to improve this, we could first perform a fast *heuristic* test that indicates whether it is *likely* that $\sigma_i \dots \sigma_{i+\ell-1} = t$. This test should fulfill three properties: (i) it should be *fast*, much faster than comparing the substring and t directly; (ii) it should be *complete*, that is, if $\sigma_i \dots \sigma_{i+\ell-1} = t$, then the test should always be positive; and (iii) it should be *sound*, that is, if $\sigma_i \dots \sigma_{i+\ell-1} \neq t$, then the test should only have a small probability of being positive.

Such a test can be implemented by using a *hash function* $h : \Sigma^* \to \mathbb{Z}$. Such a hash function assigns integer numbers to $\sigma_i \dots \sigma_{i+\ell-1}$ and to t. These two numbers can be compared quickly, in one step. Furthermore, a good hash function will make collisions unlikely, so that it will happen only rarely that two different strings $\sigma_i \dots \sigma_{i+\ell-1}$ and t receive the same hash value. This leads to the following approach, the algorithm by Rabin-Karp.

```
for i := 1 to k - l + 1 do
    // Quick test using a hash function
    if h(s[i...i+l-1]) == h(t) then
        while j <= l and s[i+j-1] == t[j] do
            j++
        // If yes, return position i
        if j == l + 1 then
            return i
// Not found, return -1
return -1
```

If collisions under h are rare, the time for the comparison $\sigma_i \dots \sigma_{i+\ell-1} \overset{?}{=} t$ after the heuristic test $h(\sigma_i \dots \sigma_{i+\ell-1}) \overset{?}{=} h(t)$ should be negligible. However, as presented, the algorithm poses a new problem: how to compute $h(\sigma_i \dots \sigma_{i+\ell-1})$ quickly? (If it takes too long to evaluate h, the whole approach would be pointless).

As a reminder: when talking about hash functions, we described the following way for computing a hash function h' for a string $a = \alpha_0\alpha_1 \dots \alpha_{\ell-1}$: pick a prime number p and interpret the individual symbols α_i as numbers $\|\alpha_i\|$ between 0 and $|\Sigma|-1$. Then, set

$$h'(a) = \left(\sum_{j=0}^{\ell-1} \|\alpha_j\| |\Sigma|^j \right) \bmod p.$$

For Rabin-Karp, it turns out to be advantageous to define the hash function slightly differently:

$$h(\sigma_i \dots \sigma_{i+\ell-1}) = \left(\sum_{j=0}^{\ell-1} \|\sigma_{i+j}\| |\Sigma|^{\ell-1-j} \right) \bmod p.$$

There are two main differences between h and h':

1. the index of σ is increased by i everywhere (because we are considering a substring of s); and

2. the powers of $|\Sigma|$ are decreasing instead of increasing (this makes the next formula slightly simpler).

Now, the main point is that *given $h(\sigma_i \dots \sigma_{i+\ell-1})$, we can determine $h(\sigma_{i+1} \dots \sigma_{i+\ell})$ in $O(1)$ steps*. Indeed, we have

$$h(\sigma_{i+1} \dots \sigma_{i+\ell}) = \Big(|\Sigma| \cdot h(\sigma_i \dots \sigma_{i+\ell-1}) - |\Sigma|^\ell \cdot \|\sigma_i\| + \|\sigma_{i+\ell}\| \Big) \bmod p.$$

(Note that we can precompute $|\Sigma|^\ell$ in advance and can reuse it every time we update the hash function). We call h a *rolling hash*.

From this, it follows that we can compute all the hash values $h(\sigma_1 \dots \sigma_\ell)$, $h(\sigma_2 \dots \sigma_{\ell+1})$, ..., $h(\sigma_{k-\ell+1} \dots \sigma_\ell)$ as well as $h(t)$ in total time $O(k+\ell)$.[1] Thus, if h is a "good" hash function were collisions are rare, we expect that the total running time for the Rabin-Karp algorithm is $O(k+\ell)$. In fact, we can make this intuition precise.

Satz 18.1. *Suppose that p is a random prime number between 2 and $\ell^2 \log(|\Sigma|\ell)$. Then, the expected running time of the Rabin-Karp algorithm is $O(k+\ell+\log|\Sigma|)$*

Beweis. First, we need $O(\ell)$ time to compute $|\Sigma|^\ell \bmod p$, $h(t)$, and $h(\sigma_1 \dots \sigma_\ell)$.

Now, for $i = 1, \dots, k-\ell+1$, let X_ℓ be the random variable that indicates the time that is needed for the i-th iteration of the **for**-loop. As discussed, we can find $h(\sigma_i \dots \sigma_{i+\ell-1})$

[1] We should make sure in every step to take all the intermediate results modulo p, in order to make sure that we will deal only with numbers that are small enough.

in $O(1)$ time. Thus, we have that $X_\ell = O(1)$, if $h(\sigma_i \dots \sigma_{i+\ell-1}) \neq h(t)$, and $X_\ell = O(\ell)$, if $h(\sigma_i \dots \sigma_{i+\ell-1}) = h(t)$.

Thus, we need to estimate the probability that $h(\sigma_i \dots \sigma_{i+\ell-1}) = h(t)$, over the choice of p. By construction, we have $h(\sigma_i \dots \sigma_{i+\ell-1}) = h(t)$ if and only if

$$\begin{aligned} \left(\sum_{j=0}^{\ell-1} \sigma_{i+j} |\Sigma|^{\ell-1-j}\right) &\equiv \left(\sum_{j=0}^{\ell-1} \tau_{1+j} |\Sigma|^{\ell-1-j}\right) \pmod{p} \\ \Leftrightarrow \left(\sum_{j=0}^{\ell-1} \big(\sigma_{i+j} - \tau_{1+j}\big) |\Sigma|^{\ell-1-j}\right) &\equiv 0 \pmod{p} \end{aligned}$$

Now, let $A = \sum_{j=0}^{\ell-1} \big(\sigma_{i+j} - \tau_{1+j}\big)|\Sigma|^{\ell-1-j}$. Then, A is a fixed number between $-|\Sigma|^\ell$ and $|\Sigma|^\ell$, and we have that $A \equiv 0 \pmod{p}$ if and only if p is a prime factor of A. Now, we observe that a natural number n can have at most $\log n$ distinct prime factors: each prime factor is at least 2, and if n has k distinct prime factors, then $n \geq 2^k$. Thus, it follows that $h(\sigma_i \dots \sigma_{i+\ell-1}) = h(t)$ if and only if p happens to be one of the at most $\log|A| \leq \log\left(\Sigma^\ell\right) = \ell \log|\Sigma|$ many prime factors of A. Now, by the prime number theorem, the number of distinct prime numbers between 2 and $\ell^2 \log(|\Sigma|\ell)$ is $\Omega(\ell^2 \log|\Sigma|)$. Thus, the probability that a random prime number between 2 and $\ell^2 \log(|\Sigma|\ell)$ is a prime factor of A is $O(1/\ell)$. It follows that $\mathbf{E}[X_i] = O(1)$. By linearity of expectation, the total running time of the algorithm is $O(\ell + k)$.

It remains to explain how to find a random prime number between 2 and $\ell^2 \log(|\Sigma|\ell)$. For this, we simply take a random number between 2 and $\ell^2 \log(|\Sigma|\ell)$ and check if it is a prime number. If the test fails, we repeat. By the prime number theorem, the probability that we find a prime number is $\Omega(1/\log(\ell|\Sigma|))$. Thus, we need $O(\log(\ell|\Sigma|))$ attempts in expectation to find such a number. There are very efficient algorithms testing whether a given number is a prime number. Thus, the time for this step is negligible. □

The topic of string search is a vast topic in algorithms, and many other variants and solutions for the problem exist. This will be covered in a later class, in particular in the classes that deal with algorithmic bioinformatics. There you will learn about the Horspool-Algorithm as well as bitvector-accelerated algorithms like the Shift-Or Algorithm.

KAPITEL 19

String Similarity–Longest Common Subsequence

Suppose we are given two strings $s, t \in \Sigma^*$, and we would like to determine how *similar* the two strings are. This task occurs often in applications, e.g., when comparing DNA or protein sequences of different species or when trying to determine the history of a text file. The bioinformaticians will address many variations of computing similarity or distance between biological sequences in the next semester. They will especially learn about the evolutionary context of similarity and distance.

Lets look at one possible, basic formulation for determining the similarity of two strings.

To answer this question algorithmically, we first need to have a formal mathematical definition of *similarity*. Only when we know exactly what we are talking about will we be able to design an efficient algorithm for the problem.

There are different ways to define the notion of string similarity. One approach is to identify a large portion of text that occurs in both s and t. This is formalized be the notion of a *common subsequence*.

Definition 19.1. *Let s and t be two strings over Σ. A* **common subsequence** *of s and t is a sequence of symbols from Σ that appears both in s and t in the same order, but not necessarily consecutively. Formally, let $s = \sigma_1 \dots \sigma_k$ and $t = \tau_1 \dots \tau_\ell$. Then, a common subsequence of s and t of length n consists of a sequence of n index pairs $(i_1, j_1), (i_2, j_2), \dots, (i_n, j_n)$ such that*

1. $1 \le i_1 < i_2 < \cdots < i_n \le k$;
2. $1 \le j_1 < j_2 < \cdots < j_n \le \ell$; *and*
3. $\sigma_{i_a} = \tau_{j_a}$, *for all* $a = 1, \dots, n$.

A common subsequence of s and t is a **longest common subsequence** *if there exists no common subsequence that contains more pairs.*

This definition leads to a simple way to determine the similarity of two strings s and t: first, we compute a longest common subsequence of s and t. Second, we find the symbols that occur in s but not in t, and the symbols that occur in t and not in s. Then, we can visualize the strings next to each other, marking the symbols that do not occur in the longest common subsequence with special colors. This approach is implemented, for example, in the standard operating system utilities `diff`, `fc`, or `cmp`.

We will now discuss an efficient algorithm to compute the longest common subsequence of s and t.

Finding the length of a longest common subsequence. Let $s = \sigma_1\sigma_2\dots\sigma_k$ and $t = \tau_1\tau_2\dots\tau_\ell$ two strings. First, we develop an algorithm to compute the *length* of the longest common subsequence of s and t. After that, we will discuss how to extend this to determine an actual LCS.

The main idea is to use a recursive approach. The main challenge is to find an appropriate recursion that can be used to solve the problem. For the LCS-problem, it is useful to consider the following subproblems: given two integers $i \in \{0,\dots,k\}$ and $j \in \{0,\dots,\ell\}$, find the length of the longest common subsequence for the *prefixes* $\sigma_1\dots\sigma_i$ and $\tau_1\dots\tau_j$ of s and t (if $i = 0$ or $j = 0$, the respective prefix is the empty string ε). We call this function $\mathrm{llcs}(i,j)$.

Now, we would like to find a recursion for $\mathrm{llcs}(i,j)$. First, we have that $\mathrm{llcs}(0,j) = 0$, for all $j = 0,\dots,\ell$, since the empty string does not contain any non-empty subsequence. Similarly, we have $\mathrm{llcs}(i,0) = 0$, for all $i = 0,\dots,k$.

For $i,j \geq 1$, we need to consider two cases: first, if $\sigma_i = \tau_j$, then we can obtain an LCS for $\sigma_1\dots\sigma_i$ and $\tau_1\dots\tau_j$ by taking an LCS for $\sigma_1\dots\sigma_{i-1}$ and $\tau_1\dots\tau_{j-1}$ and by adding (i,j). Second, if $\sigma_i \neq \tau_j$, then we can find an LCS for $\sigma_1\dots\sigma_i$ and $\tau_1\dots\tau_j$ as follows: take an LCS for $\sigma_1\dots\sigma_{i-1}$ and $\tau_1\dots\tau_j$ and an LCS for for $\sigma_1\dots\sigma_i$ and $\tau_1\dots\tau_{j-1}$. Then, the longer of the two LCS's will also be an LCS for $\sigma_1\dots\sigma_i$ and $\tau_1\dots\tau_j$.

This means that llcs fulfills the following recurrence:

$$\begin{aligned}\mathrm{llcs}(i,j) &= 1 + \mathrm{llcs}(i-1,j-1), \text{ if } \sigma_i = \tau_j, \text{ and}\\ \mathrm{llcs}(i,j) &= \max\{\mathrm{llcs}(i,j-1), \mathrm{llcs}(i-1,j)\}, \text{ if } \sigma_i \neq \tau_j.\end{aligned}$$

We can implement this recursion directly. However, this will not be very efficient. In fact, the running time could become *exponential* in k and ℓ, because the second case in the recursion performs *two* recursive calls instances that are only smaller by a single character. Nonetheless, when unravelling the recursion tree, we see that there are actually only $(k+1)\cdot(\ell+1)$ subproblems, and that many subproblems are considered repeatedly during the recursion.

One way to take advantage of this structure is called *memoization*: we implement the recursion directly, but we maintain a dictionary that contains the solutions to all the subproblems that we have already considered. Before performing a recursive call, we first consult the dictionary, and if the answer has already be computed, we refrain from repeating the calculation. This leads to an algorithm with a polynomial running time, using a dictionary structure.

A second, conceptually simpler way to reduce the running time is called *dynamic programming*. Here, we do not implement the recursion directly, but we systematically build a table with the solutions for all the subproblems, from small to large. Let `LLCS` be a two-dimensional array of **Integer**'s with dimension $(k+1)\times(\ell+1)$. We fill `LLCS` systematically bottom-up, using the recurrence relation from above. This can be implemented with simple nested **for**-loops.

```
// Initialization
for i := 0 to k do
    LLCS[i,0] := 0
for j := 0 to l do
    LLCS[0,j] := 0
// Implementing the recursion
for i := 1 to k do
    for j := 1 to l do
        if s[i] == t[j] then
            LLCS[i,j] := 1 + LLCS[i-1,j-1]
        else
            LLCS[i,j] := max(LLCS[i-1,j], LLCS[i,j-1])
```

The result is in `LLCS`$[k,\ell]$. The running time is $O(k\ell)$.

Computing an actual LCS After `LLCS`$[k,\ell]$ has been found, we can determine the longest common subsequence by reconstructing the path through `LLCS` that led to the solutions. This is done from the back of the table, starting at `LLCS`$[k,\ell]$.

```
// a stack to store the result
S := new Stack
// start at the end
(i,j) := (k,l)
while i>0 and k>0 do
    if s[i] == t[j] then
        // if the current symbols are equal, we can include
        // them into the sequence
        (i,j) := (i-1, j-1)
        S.push(s[i])
    else
        // otherwise we need to determine which of the two
        // choices led to the maximum
        if LLCS[i,j-1] > LLCS[i-1,j] then
            (i,j) := (i,j-1)
        else
            (i,j) := (i-1,j)
// now S contains an optimal subsequence
while not S.isEmpty() do
    output S.pop()
```

Since in each iteration of the **while**-loop at least one index `i` or `j` decreases by one, the running time is $O(k+\ell)$.

Remark. The dynamic programming algorithm for finding an LCS takes time $O(k\ell)$. As with string search, one may ask if this can be improved: in string search, we had the naive algorithm that took $O(k\ell)$ time, but using more sophisticated methods like, e.g., Knuth-Morris-Pratt, this could be improved to $O(k+\ell)$. However, it turns out that in the case of computing the LCS, such an improvement is unlikely: one can show that, assuming the *strong exponential time hypothesis*, a popular complexity-theoretic assumption, there is no algorithm for computing the LCS that takes time $O((k\ell)^{1-\varepsilon})$, for any fixed $\varepsilon > 0$.

Dynamic programming. The algorithm for computing the LCS is a typical example of a general algorithmic paradigm called *dynamic programming*.[1]

The general idea of dynamic programming is to solve an algorithmic problem by *recursion*. Typically, in a dynamic programming scenario, the subproblems overlap and throughout the recurrence, the same subproblem occurs multiple times. Thus, in dynamic programming, we create a table for all possible subproblems and we fill the table systematically from the smallest to the largest subproblem. The resulting algorithms are often quite elegant, consisting only of a few nested **for**-loops.

Dynamic programming is applicable to a wide array of problems, and we will see several other examples throughout this class and in other classes (e.g., the algorithm of Floyd-Warshall for solving the all-pairs-shortest-paths problem in graphs, the algorithm of Kleene for converting a deterministic finite automaton to a regular expression, the algorithm of Cocke-Younger-Kasami for the word-problem in context-free languages, etc.). The main challenge in dynamic programming is to come up with a good recursive formulation for the problem at hand. Once the recursion is found, the rest is usually routine.

[1] The term *dynamic programming* was coined by Bellman in the early days of computer science. Here, "programming" does not refer to "writing code for a computer", but to solving an optimization problem (as in, "creating a TV program"). "Dynamic" is a meaningless word that was chosen to impress bureaucrats that make decisions on research funding. Nowadays, the technique might be called "interdisciplinary optimization epistemology".

V

Graphs

KAPITEL 20

Graphs

In the final section of this class, we will look at algorithms for *graphs*. A graph is a mathematical structure that formalizes *relationships* between objects. Formally, a graph consists of *nodes* and *edges*. Nodes are arbitrary objects that represent entities of interest. Edges represent relationships between nodes and show that two nodes are connected in some way. Small graphs can be visualized by a diagram: the nodes are typically represented by small disks, sometimes annotated with additional information such as the name or content of the node. The edges are represented by line segments or curves that connect two nodes.

Graphs are very general mathematical objects. They are used in many places in computer science and mathematics to model a wide variety of real world scenarios. Here are a few examples:

- **Road networks**. Graphs can be used to model all kinds of road networks. For example, a simple way to represent the road network of Berlin would be as follows: the nodes are all intersections between roads in Berlin, and the edges are the direct connections between the intersections.

  One can imagine a wide variety of information that could be associated with the nodes and the edges: for example, for each intersection, we could store the name, the geographic coordinates, the elevation, etc. For each edge we could store the distance, the inclination, whether it is a one-way street, whether the road has special regulations for bikes, children, noise-protection, etc.

- **Public transportation networks**: Graphs can also model public transportation networks. For example, the nodes could be all subway stops in Berlin, and the edges could be direct subway connections between the stops.

  Again, a lot of information could be stored with the individual nodes and edges, e.g., the name of the station, the name of the subway lines that service a connection, the travel time between to stops, etc.

- **Computer networks**: The nodes in a computer network could be individual computers or servers, the edges could represent direct connections between servers, e.g., through physical communication cables.

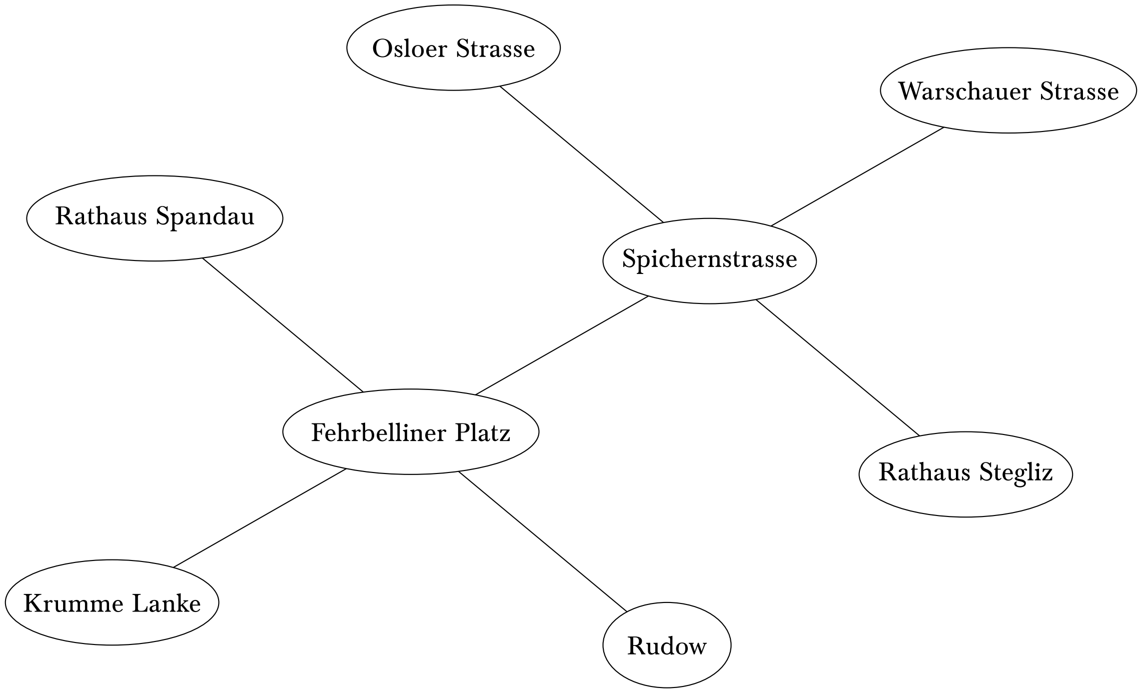


Abbildung 20.1: An example graph.

- **Web graph**: In the web graph, the nodes represent individual websites. The edges represent links between websites. Note that in the web graph, edges are *directed*: we can follow a link, but when we are on a website, we cannot see which other sites link to this website (unless, we use additional tools that provide this information).[1]

- **The bridges of Königsberg (Kaliningrad)**: The foundational problem of graph theory is the following: the city of Königsberg is traversed by the river Pregel. Within the city, there are two islands and seven bridges that connect the islands to each other and to the sides of the river. The question is to determine whether there is a way to walk through Königsberg so that each of the seven bridges is crossed exactly once. In 1736, Leonhard Euler showed that this is not possible. For this, he developed the formalism that now forms the basis of graph theory: The nodes correspond to the two sides of the river Pregel and the two islands, the edges represent the bridges.

  Note that in this problem, there are nodes that are connected by more than one edge (since there may be two bridges between an island and a river side). Note also that the bridges in today's Kaliningrad are different than in the 18th century. Today, it is actually possible to walk through the city so that every bridge is

[1] Interestingly, *Project Xanadu*, an early attempt to develop a networked hypertext model that predates the world-wide-web by 30 years, envisioned *bidirectional* links between individual pages. To date, Project Xanadu is not finished.

crossed exactly once. However, this walk will need to have different start and end points.

- **Bank transactions**: Banks use graphs to analyze transactions between their customers. The nodes are bank accounts, and the edges represent transactions between the accounts.

- **Social graph**: Graphs can also represent relationships between humans. A typical social graph is as follows: the nodes represent individual people, and edges correspond to a relationship between the people, e.g., that person *A knows* person *B*.

- **Protein interaction graphs**: In bioinformatics, we often encounter graphs that reflect some biological reality. For example in *protein interaction graphs*, the nodes are certain proteins that occur in the human body, and there is an edge between two proteins if an only if the two proteins typically engage in a chemical reaction.

- **Linked data structures**: Throughout this class, we have already encountered many examples of graphs: linked lists, binary trees, and tries are all special types of graphs with additional structure (e.g., directed edges, certain additional information in the vertices and in the edges, a designated root node).

Formal definitions, notation, and terminology. Formally, a graph G has two components: a finite, nonempty set V of *vertices*, and a set E of *edges*. The set E consists of two-element subsets of distinct vertices, i.e.,

$$E \subseteq \{\{v,w\} \mid v,w \in V, v \neq w\}.$$

The vertices are sometimes also called *nodes*. The edges are sometimes also called *arcs*. We write $G = (V,E)$ to indicate that the graph G consists of vertex set V and edge set E.

We say that two vertices $v,w \in V$ are *adjacent* if and only if there is an edge between v and w. For an edge $e = \{v,w\} \in E$, we call v and w the *endpoints* of e, and we say that e and the endpoints u, v are *incident* to each other.

Given a node $v \in V$, the *neighborhood* $\Gamma(v)$ of v consists of all nodes that are adjacent to v:

$$\Gamma(v) = \{w \mid \{v,w\} \in E\}.$$

The *degree* of v, denoted by $\deg(v)$, is the number of neighbors of v:

$$\deg(v) = |\Gamma(v)|.$$

The well-known hand-shake lemma states that the total degree in a graph is twice its number of edges:

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Beweis. For a vertex $v \in V$ and an edge $e \in E$, let $I_{ve} = 1$, if v is an endpoint of e, and $I_{ve} = 0$, otherwise. Then, we have

$$\begin{aligned}\sum_{v\in V}\deg(v) &= \sum_{v\in V}\sum_{e\in E} I_{ve} \\ &= \sum_{e\in E}\sum_{v\in V} I_{ve} \\ &= \sum_{e\in E} 2 \\ &= 2|E|.\end{aligned}$$

□

We saw that there are many objects that can be modeled with graphs. Thus, it will come as no surprise that there are many variants of the definitions that adapt the notion to different scenarios. The graphs that we have defined so far are also called *simple*, *undirected*, *unweighted* graphs. Common variations of the definition (some of which we will use later) include:

- **Directed graphs.** In a *directed* graph (as opposed to an *undirected* graph), the edges have an orientation, going *from* one endpoint *to* the other.

  Formally, the edge set of a directed graph E is a subset of the Cartesian product $E \times E$, so the edges are ordered pairs of vertices. An edge $e = (v, w)$ is *directed* from v to w. This is visualized as an arrow from v to w. We call v the *tail* of e and w the *head* of e (and, collectively, we still call v and w the *endpoints* of e).

  Given a vertex $v \in V$, the *outgoing neighbors* $\Gamma^+(v)$ of v are the heads of the edges that have v as a tail:
  $$\Gamma^+(v) = \{w \mid (v, w) \in E\}.$$
  The *outdegree* of v, denoted by $\deg^+(v)$, is the number of outgoing neighbors of v:
  $$\deg^+(v) = |\Gamma^+(v)|.$$
  Similarly, the *incoming neighbors* $\Gamma^-(v)$ of v are the tails of edges that have v as a head:
  $$\Gamma^-(v) = \{w \mid (w, v) \in E\}.$$
  The *indegree* of v, denoted by $\deg^-(v)$, is the number of incoming neighbors of v:
  $$\deg^-(v) = |\Gamma^-(v)|.$$
  Since every edge has exactly one head and one tail, we have
  $$\sum_{v\in V}\deg^-(v) = \sum_{v\in V}\deg^+(v) = |E|.$$

- **Multigraphs.** In a *multigraph* (as opposed to a *simple* graph), there can be multiple edges between two vertices. Furthermore, in a mutigraph there can also be *loops*, i.e., edges that connect a vertex to itself.

  Formally, the edge set E of a multigraph is a *multiset* of subsets of vertices, such that every in E has *at most* two elements. In a multset, the same element can occur multiple times. A set in E that contains only a single vertex v represents a loop that connects v to itself.

- **Weighted graphs.** In a *weighted* graph (as opposed to an *unweighted* graph), we have additional information associated with the parts of the graph.

  There are several variants, but in this class, we will focus on *edge-weighted* graphs where each edge has an associated real number (e.g., travel time, distance, inclination, etc.). This is formalized by a *weight function* $w : E \to \mathbb{R}$, where $w(e)$ is called the *weight* of edge e.[2]

Algorithmic problems on graphs. Graphs are a versatile modeling tool in Computer Science, Mathematics, and beyond. As such, there are many interesting algorithmic problems that arise in the study of graphs. Here are a few examples, some of which we will consider more closely in the following chapters.

- **Connectivity.** Let $G = (V, E)$ be a graph, and let $v, w \in V$ be two vertices. The general connectivity problem asks whether v and w are *connected* in G, i.e., whether there is a *path* in G that goes between v and w. A path between v and w is a sequence of vertices that starts with v, ends with w, and that has the property that two consecutive vertices are adjacent in G.

  There is also a directed version of the problem. Here, G is a directed graph, and the question is whether there is a *directed* path from v to w. A directed path from v to w is a sequence of vertices that starts at v, ends at w, and that has the property that for any two consecutive vertices x, y on the path, there is an edge from x to y in G.

- **Unweighted shortest path.** Let $G = (V, E)$ be a graph, and let $s, t \in V$ be two nodes in G. In the *shortest path* problem, we would like to find a path from s to t that has the minimum number of edges among all such paths.

  As with connectivity, this problem can be asked in both undirected and directed graphs.

- **Weighted shortest path.** Let $G = (V, E)$, $w : E \to \mathbb{R}$ be a weighted graph, and let $s, t \in V$ two nodes in G. Let π be a path from s to t. The *total weight* of π is the sum of the weights of the edges in π. In the *weighted shortest path* problem, we would like to find a path from s to t that minimizes the total weight. As usual, there is both an undirected and a directed version.

[2] Depending on the context, the value $w(e)$ may also be called differently, e.g., the *length* of edge e or the *cost* of edge e).

- **Minimum cut.** Let $G = (V, E)$ be a simple undirected graph. Let $s, t \in V$ be two nodes in G. The task is to find a minimum set $F \subseteq E$ of edges such that by deleting F from G, we can disconnect s from t in G.
- **Minimum spanning tree.** Let $G = (V, E)$, $w : E \to \mathbb{R}$ be a weighted, undirected, connected graph. The task is to find set $F \subset E$ such that (V, F) is a tree and such that the sum of the edge weights in F is minimum.

In the following chapters, we will look at some of these problems in more detail. First, however, we need to talk about how graphs are represented in a computer.

KAPITEL 21

Representing a Graph

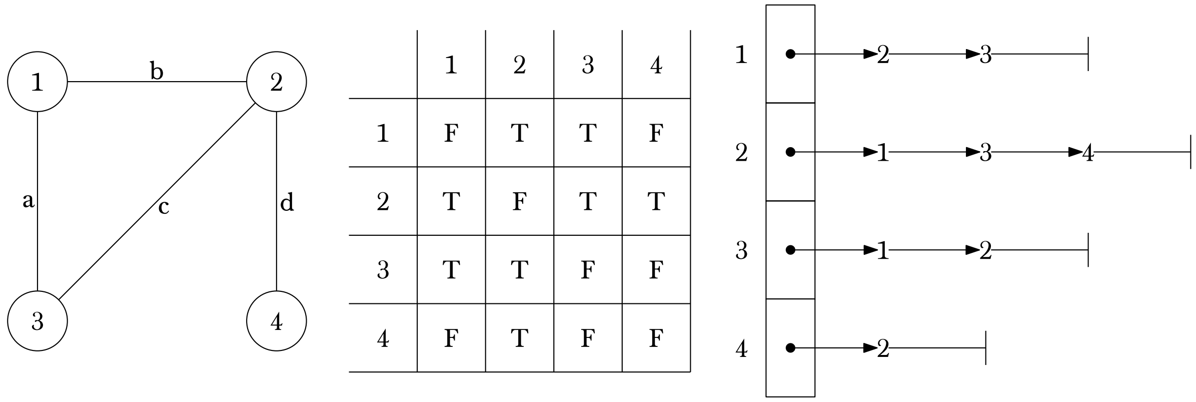
We now discuss several ways to represent a graph in a computer.

Representing the vertices. Recall that a graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. The vertices are an arbitrary nonempty finite set. For representing graphs in a computer, we will simply assume that $V = \{1, \ldots, n\}$, i.e., that the vertices are numbered from 1 to n, where n is the number of vertices in G. Then, we can represent a vertex by an integer-variable, and we can use it as an index in an array. If we need to store additional information with the vertices, we can use an additional array that has, at position i, the data for vertex i.

The main challenge when using a graph in a computer lies in representing the edges. There are many ways to do this. Here are three:

Adjacency matrix. In an *adjacency matrix representation* of a graph, the edges are represented by a two-dimensional array `A` that has n columns and n rows, such that the rows and the columns of `A` corresponds to the vertices of V. We call `A` the *adjacency matrix* for G.

To represent a simple, undirected, unweighted graph, we let `A` a matrix of Booleans. Then, for any two vertices $v, w \in V$, we have `A`$[v, w]$ = `A`$[w, v]$ = `true`, if there is an



	1	2	3	4
1	F	T	T	F
2	T	F	T	T
3	T	T	F	F
4	F	T	F	F

Abbildung 21.1: Adjacency maxtrix and adjacency list.

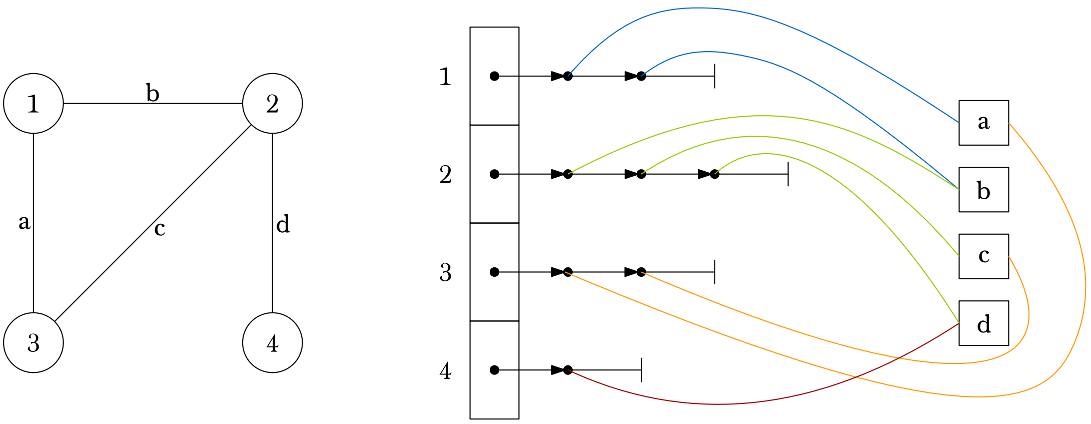


Abbildung 21.2: Incidence list.

undirected edge between v and w, and $\mathtt{A}[v,w] = \mathtt{A}[w,v] = \mathtt{false}$, otherwise. Thus, is G is undirected, then A is a *symmetric* matrix, i.e., for any given vertex v, the row and the column for v are identical.

To represent a directed, unweighted graph, we again let A be a matrix of Booleans. Now, for any two vertices $v,w \in V$, we let $\mathtt{A}[v,w] = \mathtt{true}$, if there is a directed edge from v to w, and $\mathtt{A}[v,w] = \mathtt{false}$, otherwise. The matrix A is not necessarily symmetric. For a vertex $v \in V$, the row $\mathtt{A}[v,\star]$ corresponds to the outgoing edges from v, and the column $\mathtt{A}[\star,v]$ corresponds to the incoming edges of v.

To represent a multigraph, we can let A be a matrix of Integers. Then, for two vertices $v,w \in V$, the entry $\mathtt{A}[v,w]$ indicates the *number* of edges from v and w (and 0, if there are none). A loop at a vertex v can be represented by a positive entry in $\mathtt{A}[v,v]$. If the multigraph is undirected, then A is symmetric.

For weighted graphs, we can use the adjacency matrix A to store the weights: we let A be a matrix of integer or floating point values, and the entry $\mathtt{A}[v,w]$ is the weight of the edge from v to w. In this case, we need a special value to indicate that there is no edge between v and w, e.g., $-\infty$, or NaN.

Adjacency list. In an *adjacency list representation* of a graph, the edges are represented by an array A. The array A has n positions, one for each vertex $v \in V$.

To represent a simple, undirected, unweighted graph, the position $\mathtt{A}[v]$ for a vertex $v \in V$ stores a linked list that contains exactly those nodes in V that are adjacent to v. We call $\mathtt{A}[v]$ the *adjacency list* for v. In an undirected graph, we have that w appears in the adjacency list $\mathtt{A}[v]$ if and only if v appears in the adjacency list $\mathtt{A}[w]$.

To represent a directed graph, we can proceed in the same manner, but now the adjacency list $\mathtt{A}[v]$ stores the *outgoing* edges from v. For a multigraph, we can allow a vertex to occur multiple times in an adjacency list. For a weighted graph, we can store the weights together with the endpoints in the adjacency lists.

Incidence list. An *incidence list representation* of a graph is very similar to an adjacency list. Again, we have an array A. The array A has n positions, one for each vertex $v \in V$. In addition, for every edge $e \in E$, we have an object that represents e and that stores additional information for e (e.g., the endpoints, the name or the edge, a weight, etc.).

For an $v \in V$, the position A$[v]$ of A stores a linked list that contains references to the objects that represent the edges that are incident to v. In a directed graph, the list A$[v]$ contains the *outgoing* edges of v. We do not need any additional data for weighted graphs of mulitgraphs, since the relevant information can be stored with the edges.

Discussion. All three representations of graphs have advantages and disadvantages.

If we represent the graph as an adjacency matrix, the space requirement is always $\Theta(|V|^2)$, independent of the number of edges. Adjacency matrices are particularly useful if we have many queries of the following form: given two vertices $v, w \in V$, is there an edge between v and w? An adjacency matrix can answer such a query in constant time: simply check the entry A$[v, w]$. On the other hand, suppose we have many queries of the following form: given a vertex $v \in V$, list all the vertices that are adjacent to v. In an adjacency matrix, this takes $\Theta(|V|)$ steps, irrespective of the actual number of neighbors: we must scan the whole row A$[v, \star]$, and collect all vertices w for which A$[v, w]$ is `true`. If we have a graph that contains many vertices and in which every vertex has only few neighbors (e.g., in a social graph), this may be unacceptably slow.

If we represent the graph as an adjacency list or an incidence list, the space requirement is $O(|V| + |E|)$ because every edge appears in exactly one (undirected case) or two (directed case) lists. If we want to check whether two given vertices $v, w \in V$ are adjacent, we must scan the lists. We can scan the lists for v and w simultaneously. If we find the other vertex in one of the two lists, we know that the v and w are adjacent. If a list ends, we know that v and w are not adjacent. Thus, the running time is $O(\min\{\deg(v), \deg(w)\}$. This can be much slower than in an adjacency list, if the degrees are large. On the other hand, suppose that we would like to find all neighbors of a given vertex v. Now, the information is directly available in the list for v, and we can output the neighbors in time $O(\deg(v))$, which can be much faster than in an adjacency matrix.[1]T

The differences between the adjacency list representation and the incidence list representation are small. Adjacency lists have slightly less overhead, because they do not need additional objects for the vertices. In contrast, incidence lists are more convenient if we want to store significant additional information with the edges.

[1] In undirected graphs, this works only for the *outgoing* edges for v, since the lists do not directly represent the incoming edges.

KAPITEL 22
Searching a Graph

We will now consider two algorithms to search through the nodes of a graph G in a systematic manner. The precise task is as follows: given a graph $G = (V, E)$ and a starting node $s \in V$, visit all nodes that can be reached from v. This makes sense in both undirected and directed graphs, and our algorithms work for both variants.

The two algorithms are *depth-first search* (DFS) and *breadth-first search* (BFS). In a DFS, we start from s and walk into the graph as far as possible, until there is no new node to be discovered. Then, we back up to the most recent node where an unvisited neighbor is still available, and we follow this path until there are no new nodes, and so on. We continue until all nodes have been explored. In contrast, a BFS tries to explore all directions simultaneously: first, we visit all nodes that can be reached in one step from s, then we visit all nodes that can be reached in two steps from s, and so on. Again, we continue until all nodes have been visited.

We now give the details of the two algorithms. The DFS is usually implemented recursively, using a function that calls itself to visit all the nodes. It can also be realized with an explicit stack, without recursion. For a BFS, we usually maintain the nodes that need to be visited in a FIFO-queue.

Recursive DFS. There is a straightforward implementation of DFS that uses a recursive function. Every node $v \in G$ has an additional attribute v`.found` of type Boolean. The attribute v`.found` indicates whether v has already been visited by the DFS. Initially, we have v`.found` = `false` for all vertices $v \neq s$, and s`.found` = `true`. The function `dfs`(v) explores all neighbors of v and immediately proceeds to every neighbor that has not been visited before.

```
dfs(v)
  //process v
  // Recursively visit all neighbors of v
  // that have not yet been visited
  for w in v.neighbors() do
    if not w.found then
      w.found <- true
      dfs(w)
```

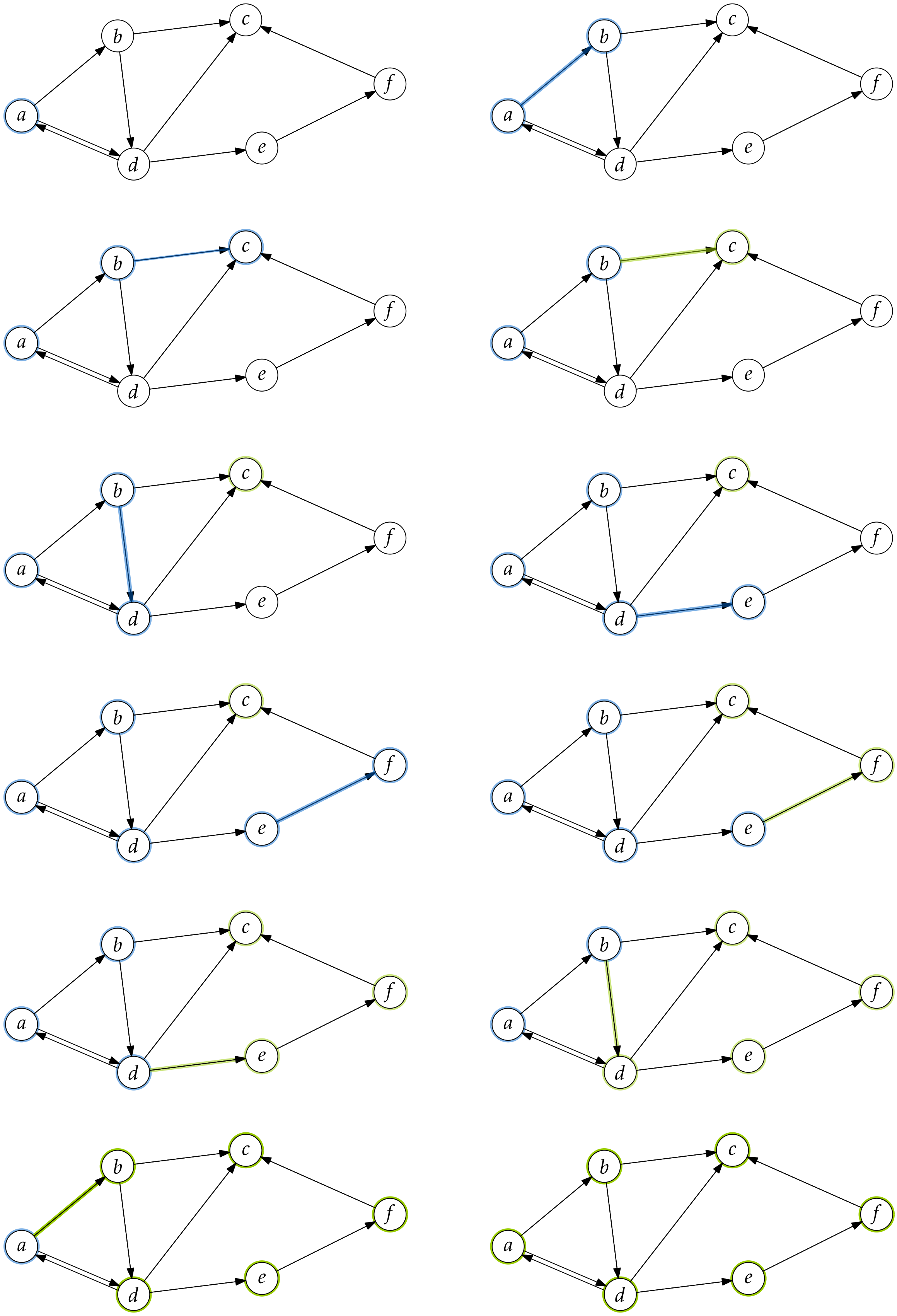


Abbildung 22.1: DFS example.

The DFS is started by invoking the function dfs(s). Since every node is visited at most once, and since every edge is inspected at most twice in the undirected case and at most once in the directed case, the running time is $O(|V| + |E|)$, if G is given as an adjacency list.

TODO: Add story about Ariadne and Theseus TODO: Example TODO: Check with MafI 1 for consistency

Iterative DFS. We can also implement DFS *iteratively*, using an explicit stack. The stack stores those vertices of G that we have already been visited, but for which we have not yes inspected all the out-neighbors. Together with each vertex w, the stack stores an *iterator* for the out-neighbors of w. This iterator is used to continue the traversal of the out-neighbors when the DFS returns to the vertex. Initially, we have v.found = false, for all $v \in V$.

```
// Initialize S
S <- new Stack
// process s
s.found <- true
S.push((s, s.neighbors()))
while not S.isEmpty() do
  (v, neighors) <- S.pop()
  // find the next unvisited neighbor
  // of the current node
  while neighbors.hasNext() do
    w <- neighbors.next()
    // if the neighbor has not yet been found,
    // go there
    if not w.found then
      w.found <- true
      S.push(v, neighbors)
      v <- w
      neighbors <- v.neighbors()
  // if there are no neighbors left, go back
```

The stack corresponds to the “Ariadne-thread” that is maintained by the DFS:: the nodes in S, from bottom to top, constitute the path from s to v that has been traversed by the DFS. Again, the running time of the iterative DFS is $O(|V| + |E|)$, if G is given as an adjacency list.

DFS is a simple and surprisingly effective algorithm to search a graph G. It has many applications, and it forms the basis for many efficient grkaph algorithms. This was first realized in a seminal paper by Robert E. Tarjan from 1972. In this class, we will not pursue this further, but these algorithms will be discussed in more advanced classes.

Sometimes, DFS is also called *backtracking*. This is also the name of a general optimization technique that is closely related to DFS. In backtracking, we construct a solution to an optimization problem by adding elements one by one. If we get stuck in this process, we discard the last element of the solution, and try another one. We continue, until a solution is found.

We emphasize that DFS does not necessarily visit all nodes in G, only the nodes that are *reachable* from s, i.e., those nodes $v \in V$ for which there exists a (directed) path from s to v in G.

Breadth-first search. BFS is implemented using a queue. The queue stores all nodes that have been discovered by the algorithm but that have not been visited yet. First, the queue contains only s. Then, we add all nodes that are one step away from s, then all nodes that are two steps away from s, and so on. We again use an attribute `found` for each vertex $v \in v$ that indicates whether v has already been visited by the BFS. Initially, we have v`.found` = `false` for all vertices $v \in V$.

```
Q <- new Queue
for v in vertices() do
  v.found <- false
s.found <- true
Q.enqueue(s)
while not Q.isEmpty() do
  v <- Q.dequeue()
  //process v
  for w in v.neighbors() do
    if not w.found then
      w.found <- true
      Q.enqueue(w)
```

Since every node is visited at most once, and since every edge is inspected at most twice in the undirected case and at most once in the directed case, the running time is $O(|V|+|E|)$, if G is given as an adjacency list.

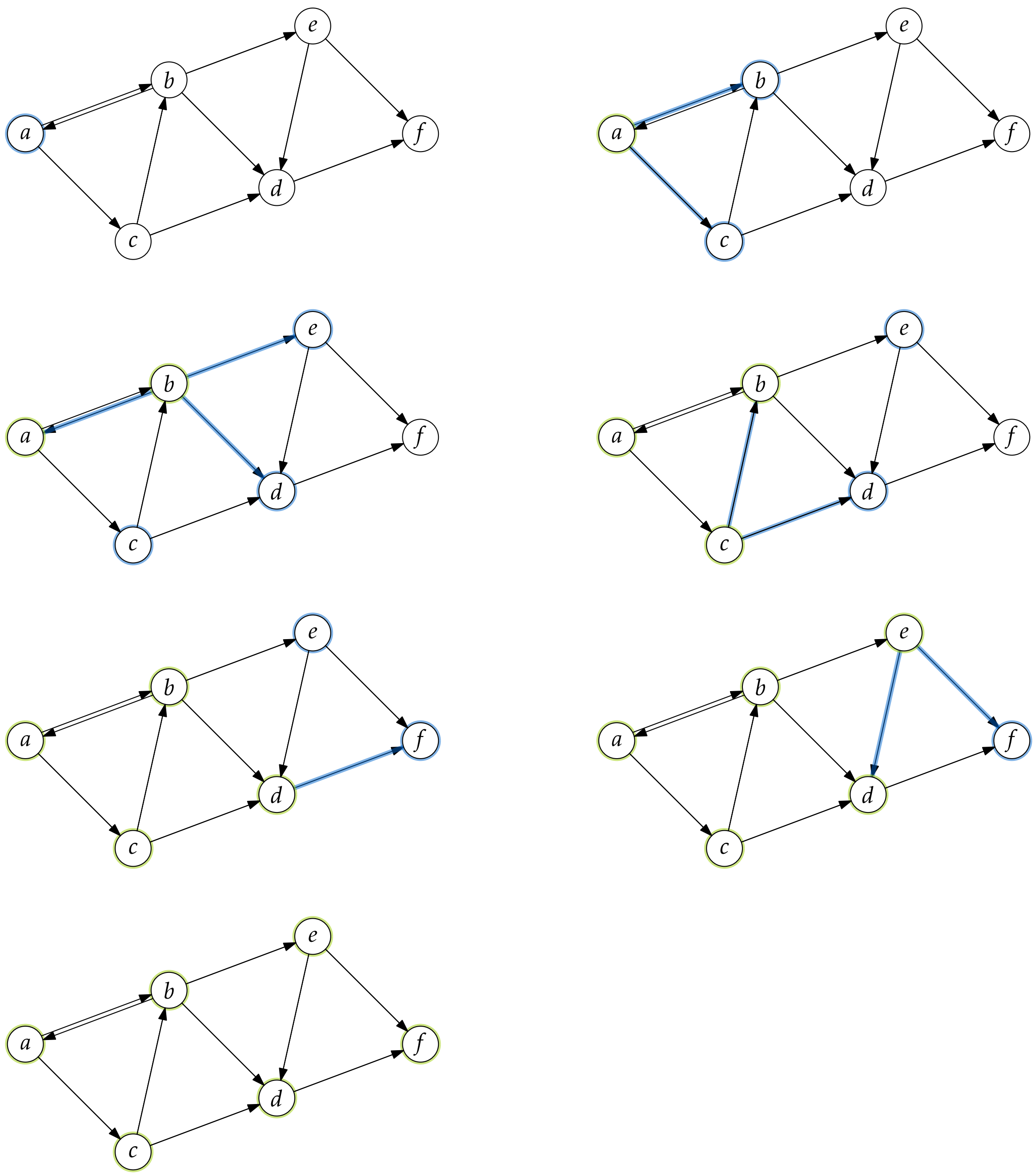


Abbildung 22.2: BFS example.

KAPITEL 23

Shortest Paths–Definitions and Unweighted Case

We will now look at the problem of finding a *shortest path* in a graph G.

For starters, we consider the following scenario: let G be an *unweighted* and *directed*, and let $s, t \in V$ be two nodes in V. We call s the *source* and t the *target*. The task is to find a *path* in G from s to t that has the minimum *length*.

A path π from s to t is a sequence $\pi : s = v_0, v_1, \dots, v_k = t$ of nodes that (i) starts with $v_1 = s$; (ii) ends with $v_k = t$; and (iii) has the property that for each pair v_i, v_{i+1} of consecutive nodes in π, there is the directed edge (v_i, v_{i+1}) from v_i to v_{i+1} in E.[1] The *length* of π, denoted by $|\pi|$, is the number of edges of π (i.e., $|\pi| = k$). The *distance* between s and t in G, denoted by $d_G(s,t)$, is the minimum length of any path from s to t.

TODO: Example

Remarks:

- We can also consider shortest paths in *undirected* graphs. The directed case is more general, because we can turn every undirected graph into a directed graph by replacing each undirected edge with two opposing directed edges.
- There is also a weighted version of the shortest path problem. We will look at it in the following chapters.
- This problem is also called the *single-pair-shortest-path* problem (SPSP), because we are looking for a shortest path between to given vertices s and t. A related problem is the *single-source-shortest-paths* problem (SSSP), which we will describe next.

[1] Note that in our definition, a vertex may appear multiple times along the path. In different texts on graph theory, this is handled differently. Since our focus here is on *shortest* paths, this choice does not matter much. In a shortest path, every vertex appears at most once: if a vertex is repeated, we can omit the part between the two repetitions and we obtain a shorter path. However, this is no longer true when have graphs with negative edge weights. We will discuss this more in the next chapter.

Single-source-shortest-path problem. In the single-source-shortest-paths problem (SSSP), we are given an *unweighted*, *directed* graph G and a *source* vertex $s \in V$. The task is to find, for every $v \in V$, a shortest path from s to v.

The SSSP is more general than the SPSP: if the know a shortest path from s to every other vertex of G, then we also know a shortest path from s to a desired target vertex t. Conversely, even if we are only interested in only a single shortest path, it may be inevitable to solve a version of the SSSP. The reason for this is toe following important property of shortest paths, the *subpath-optimality-property*: subpaths of shortest paths are also shortest paths. More formally:

Observation: Let $G = (V, E)$ be a graph, and let $s = v_0, v_1, \dots, v_k = t$ be a shortest path from s to t. Then, for every $j = 0, \dots, k$, we have that $s = v_0, v_1, \dots, v_j$ is a shortest path from s to v_j.

The reason for this observation is simple: if there were a $j \in \{0, \dots, k\}$ and a path π' from s to v_j that is shorter than the path $v_0, \dots, v_j$, then we would have a shorter path from s to t by taking the path π' followed by the suffix $v_j, \dots, v_k$. This is a contradiction to the fact that $v_0, \dots, v_k$ is a shortest path from s to t.

The subpath-optimality-property implies that to find a shortest path from s to t, we also need to find all shortest paths for the intermediate vertices. Thus, in the following, we will focus on solving the SSSP-problem in an unweighted graph.

From the discussion in the previous chapter, it is plausible that BFS is well suited to solve the SSSP-problem: we start at the source s, then we visit all the vertices that can be reached from s with a single edge, then we visit all vertices that can be reached from s with two exactly two edges, etc. Thus, BFS already encounters all the vertices in the correct order. All that we still need to do is to keep track of the distances to s and of the shortest paths to s throughout the algorithm.

Before we can do this, there is one final question: when solving the SSSP-problem, how do we keep track of all the different shortest paths? Of course, we could store, individually for each vertex $v \in V$, an explicit representation of a shortest path from s to v. However, this is not very efficient: in a graph with n vertices, we may need $\Omega(n^2)$ space to represent all the shortest paths explicitly.

To solve this problem, we use the subpath-optimality property to represent all the shortest paths from s *implicitly*: suppose that π is a shortest path from s to v, for a vertex $v \in V$, and suppose that we know that w is the last vertex on π before v. Then, we know that the prefix of π from s to w is also a shortest path. Thus, we can obtain a shortest path from s to v by constructing a shortest path from s to v and by adding the edge (v, w). In other words, we need to know only that w is the *predecessor* of v on a shortest path from s to v. The, we can proceed recursively.

After this discussion, it is straightforward to extend BFS in order to solve the SSSP-problem from s. With each vertex $v \in V$, we store two additional properties: the *distance* v.d from s to v, and the *predecessor* v.pred of v. The distance and the predecessor are set as soon as a vertex is encountered for the first time by the BFS.

```
Q <- new Queue
for v in vertices() do
```

```
  v.found <- false
  // NEW: Initially, the distances
  // from s are infinite and the predecessor
  // does not exist (we have not yet found a
  // path from s to v)
  v.d <- INFTY
  v.pred <- NULL
// we have seen s, and the distance from s to
// itself is 0
s.found <- true
s.d <- 0
Q.enqueue(s)
while not Q.isEmpty() do
  v <- Q.dequeue()
  for w in v.neighbors() do
    if not w.found then
      w.found <- true
      // NEW: Update the distance
      // and the predecessor
      w.d <- v.d + 1
      w.pred <- v
      Q.enqueue(w)
```

TODO: Add example

As before, the running time of the modified version of BFS is $O(|V|+|E|)$, if the graph is given as an adjacency list representation.

One can show that BFS solves the SSSP-problem in unweighted graphs: when the BFS-algorithm terminates, for every $v \in V$, we have that (i) $v.\mathtt{d} = d_G(s,v)$ (the distances have been computed correctly); and (ii) $v.\mathtt{pred}$ is the predecessor of v on a shortest path from s to v (if v is reachable from s). If we draw all the $v.\mathtt{pred}$-pointers in G, we obtain a rooted tree with root s that encodes all the shortest paths in G. This tree is called a *shortest path tree* for s in G.

We will not do the correctness proof here. Instead, in the next chapter, we look at at Dijkstra's algorithm, a generalization of BFS for graphs with nonnegative edge weights. We will prove that Dijkstra's algorithm is correct. This also implies the correctness of BFS.

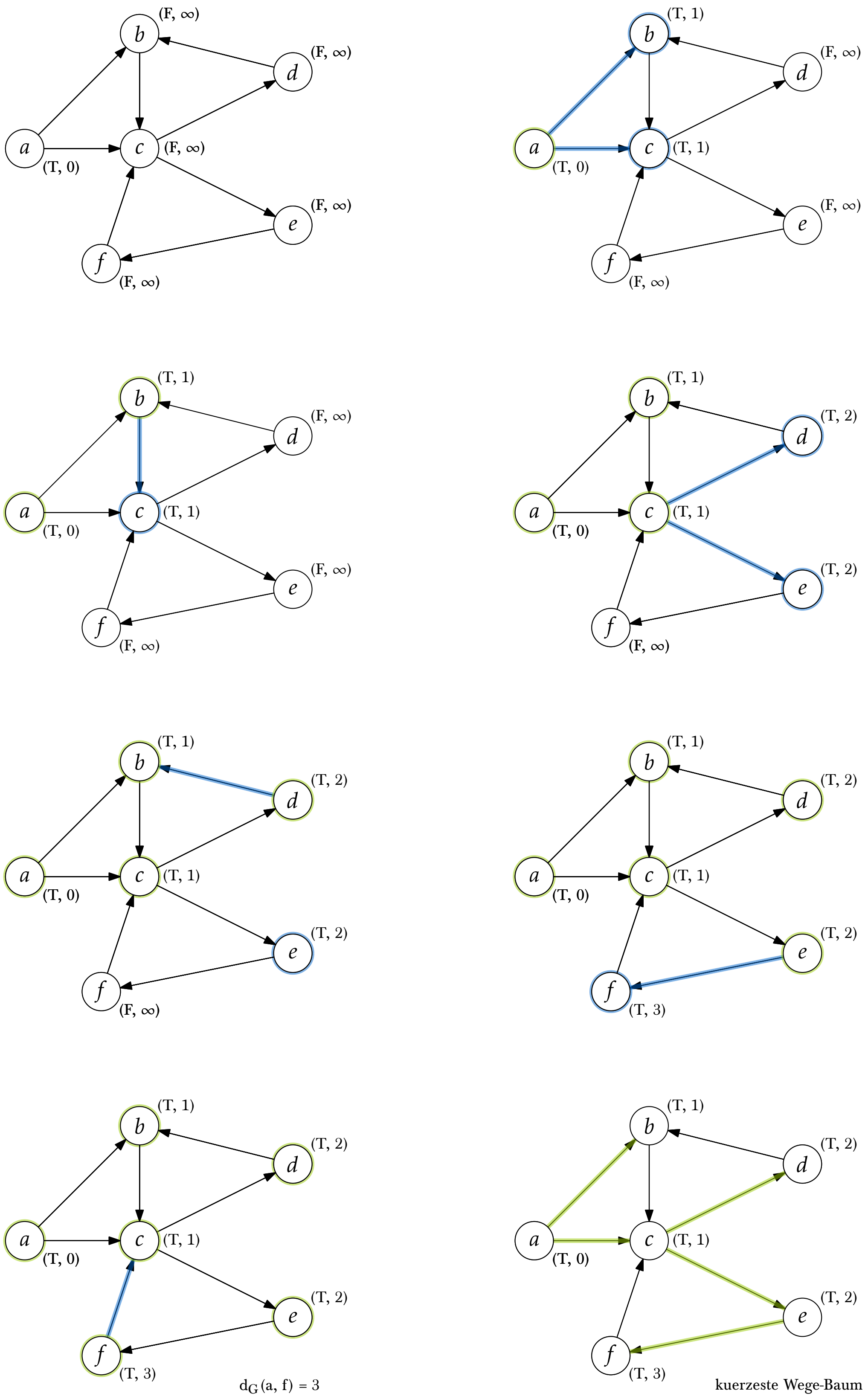


Abbildung 23.1: SSSP example.

KAPITEL 24

Shortest Paths in Weighted Graphs

We now consider the single source shortest path problem in a *weighted* graph. The setting is as follows: we are given a directed graph $G = (V, E)$ and a source vertex $s \in V$. In addition, we have a function $\ell : E \to \mathbb{R}$ that assigns a *length* $\ell(e)$ to every edge $e \in E$. The goal is to find, for every vertex $v \in V$, a shortest path from s to v *with respect to the edge lengths* $\ell(e)$.

To make this formal, we need to define the notion of the length of a path. Let $\pi : s = v_0, v_1, \dots, v_k = t$ be a path from s to t (with k edges). The *length* of π is defined as

$$|\pi| = \sum_{i=1}^{k} \ell(v_{i-1}, v_i).$$

Note that this definition is a generalization of the definition for the unweighted case in the previous chapter (just set $\ell(e) = 1$, for every edge $e \in E$). The *distance* between s and t in G, denoted by $d_G(s,t)$, is the smallest length of any path from s to t, and a *shortest path* from s to t is a path that realizes this distance.

In general, a shortest path from s to t may not exist. Suppose that G contains a vertex w such that (i) s can reach v; (ii) v can reach t; and (iii) there is a path π' that starts at v, ends at v, and that has $|\pi'| < 0$. Then, we can make the distance between s and t arbitrarily small by first going from s to v, then following the path π' as long as we like, and finally going from v to t. We call π' a *negative cycle*.[1] In general, it holds that if G contains a negative cycle that can be reached from s, then the SSSP-problem from s is not well-defined. Conversely, one can show that if s does not contain a negative cycle that is reachable from s, then all the shortest paths from s are well-defined.[2]

Thus, in order to solve the SSSP problem from s in a weighted graph G, we need to make sure that G does not contain a negative cycle that is reachable from s. One simple way to do this is to consider only graphs where all edges have nonnegative edge weights. This is a simple and well-motivated scenario that occurs often in practice

[1] More precisely, it should be called a *cycle of negative total weight*, but we use the more loose terminology for the sake of brevity.

[2] One could also address this issue by defining a *path* in such a way that every node may occur at most once. Then, a shortest path always exists, even in the presence of negative cycles. However, with this definition, the shortest path problem becomes much more difficult, and we do not have any efficient algorithms for it.

(e.g., in a road network, all edge weights are nonnegative). We will take this approach in the current chapter. In the next chapter we will see how to deal with the more general case.

Dijkstra's algorithm. We are faced with the following task: given a directed graph $G = (V, E)$, a nonnegative weight function $\ell : E \to \mathbb{R}_0^+$ and a source vertex s, compute the shortest paths from s to all other vertices in V.

Our approach is to generalize breadth-first search. As we saw, BFS proceeds uniformly in all directions, starting from s. More precisely, BFS maintains a queue Q that contains all the vertices that may have edges to vertices that we have not seen yet. In each step, the BFS extracts a vertex v from Q that is closest to s, and it uses v to identify new vertices that are one more step away from v. In a weighted graph, the main difference is that now edges may have different lengths. Thus, when computing the distances of the new vertices, we should take the edge lengths into account. Also, in order to process the vertices according to their distance from s, we should use a *priority queue* instead of a simple queue. This leads to the following attempt for a generalized BFS:

```
// NEW: Use a priority queue instead of a queue
Q <- new PrioQueue
for v in vertices() do
  v.found <- false
  v.d <- INFTY
  v.pred <- NULL
s.found <- true
s.d <- 0
// NEW: When inserting into Q, we use the distance
//      as the key.
Q.insert(s, s.d)
while not Q.isEmpty() do
  // NEW: We take the vertex in Q with the
  //     smallest distance from s
  v <- Q.extractMin()
  for w in v.neighbors() do
    if not w.found then
      w.found <- true
      // NEW: We compute w.d using the
      //   length of the edge (v, w)
      w.d <- v.d + l(v, w)
      w.pred <- v
      //NEW: Again, we use w.d as the key for w
      Q.insert(w, w.d)
```

If we try this algorithm on a simple example, we see that is does not quite work. The problem is that unlike in a BFS, we cannot be sure that the first time we encounter a vertex w from a vertex v, we have also found a shortest path to w. If may be the case that we see w again from another vertex v', and that the shortest path to w goes through v' instead of v. There is a simple fix: *every time* we see a vertex w from a vertex v, we check if the path through v is better than the best path we have seen so far. If so, we update the distance and the predecessor for w to account for the better path. This leads to the following attempt:

```
Q <- new PrioQueue
for v in vertices() do
  v.found <- false
  v.d <- INFTY
  v.pred <- NULL
s.found <- true
s.d <- 0
Q.insert(s, s.d)
while not Q.isEmpty() do
  v <- Q.extractMin()
  for w in v.neighbors() do
    if not w.found then
      w.found <- true
      w.d <- v.d + l(v, w)
      w.pred <- v
      Q.insert(w, w.d)
    // NEW: If the path through v is better than
    // the best path we have found so far, we
    // update the information for w
    else if v.d + l(v, w) < w.d then
      w.d <- v.d + l(v, w)
      w.pred <- v
      Q.decreaseKey(w, w.d)
```

The operation `decreaseKey` receives an element that is already present in the priority queue and a new key that is smaller than the current key associated with the element, and it updates the key accordingly.

It turns out that this new algorithm is now correct and computes all the shortest paths for s. Before we discuss the algorithm further, we slightly simplify it using the sentinel technique: instead of having a special `found`-attribute for all the vertices, we can indicate that a vertex v has not been discovered yet by setting v.d to ∞. This simplifies a few things: (i) we can store all the nodes in the priority queue from the beginning; and (ii) we do not need to distinguish between the first and the later encounters of a vertex w. The streamlined pseudocode is as follows:

```
Q <- new PrioQueue
for v in vertices() do
  // This indicates that v has not been found yet
  v.d <- INFTY
  v.pred <- NULL
  // All vertices are inserted into Q
  Q.insert(v, v.d)
// Initially, only s has been discovered
s.d <- 0
Q.decreaseKey(s, s.d)
while not Q.isEmpty() do
  (*)
  v <- Q.extractMin()
  for w in v.neighbors() do
    // This test now covers both the
    // case that w is encountered for
    // the first time and that we find
    // an improved path to w
    if v.d + l(v, w) < w.d then
      w.d <- v.d + l(v, w)
      w.pred <- v
      Q.decreaseKey(w, w.d)
```

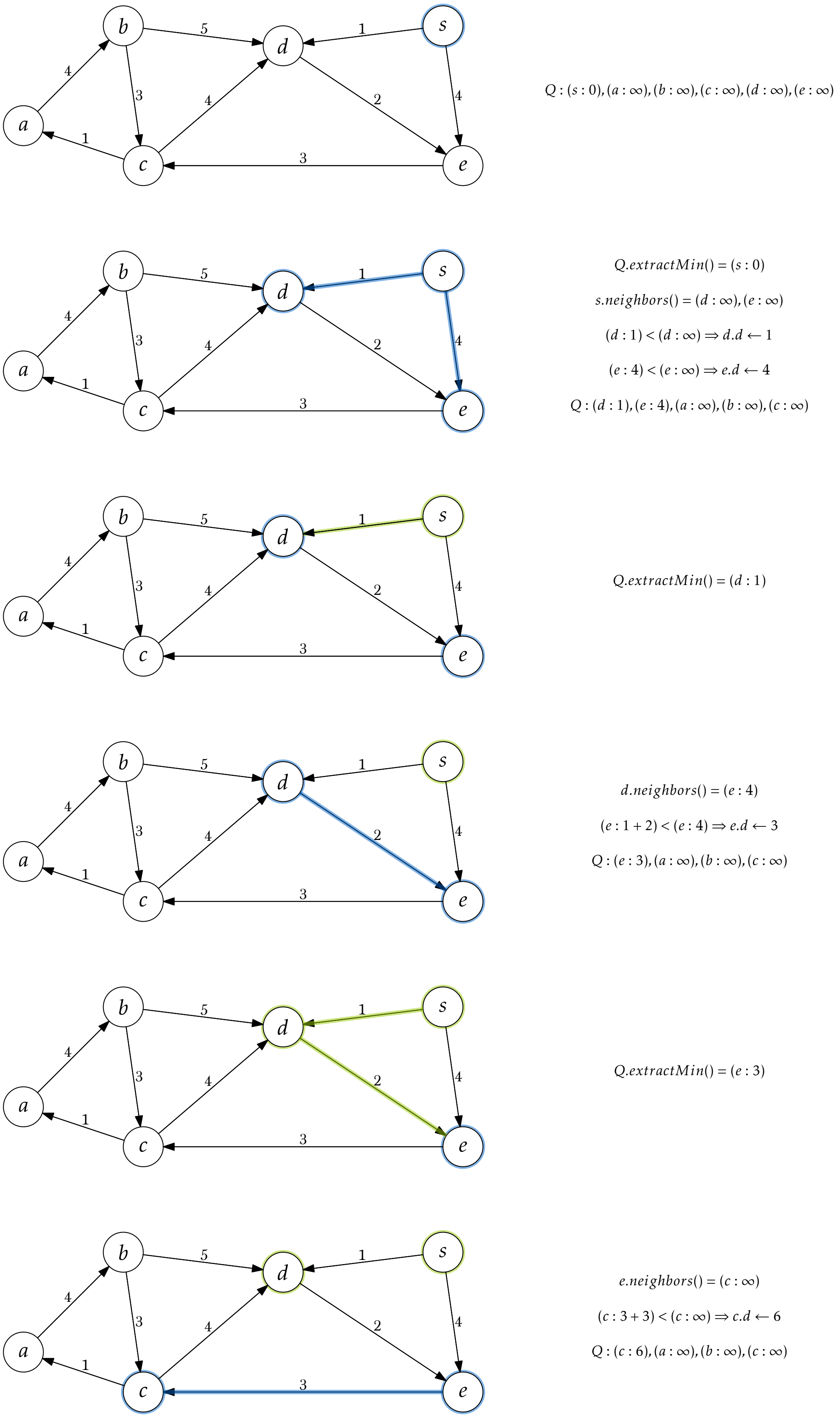
This streamlined algorithm is called the algorithm of Dijkstra.

Let us first analyze the running time of the algorithm: in the `for`-loop, every node is inserted into the priority queue, the additional time for each iteration of the loop is constant.. Then, we call a `decreaseKey`-operation on s. In the `while`-loop, each node is removed exactly once from the priority queue. The inner `for`-loop is executed exactly once for each directed edge. In each iteration of the inner `for`-loop, we may perform one `decreaseKey`-operation. The remaining overhead is constant. Thus, the running time of Dijkstra's algorithm is dominated by running time of the operations on the priority queue. We have $|V|$ `inserts`, $|V|$ `extractMins`, and at most $1 + |E|$ `decreaseKeys`. If the priority queue is implemented with a binary heap, each operation takes $O(\log|V|)$ time, and the total running time is $O(|V|\log|V|+|E|\log|V|)$. This can be improved by using a more sophisticated priority queue implementation. For example, *Fibonacci heaps* support `inserts` and `decreaseKeys` in $O(1)$, and `extractMins` in $O(\log|V|)$ *amortized* time. (**TODO**: Explain "amortized"). This means that the total running time of Dijkstra's algorithm with a Fibonacci heap is $O(|V|\log|V| + |E|)$.

Now, we prove the correctness of Dijkstra's algorithm. For this, we first define for each node $v \in V$ the *tentative path* $\Pi(v)$ as follows: (i) if $v = s$, then $\Pi(s) = s$; (ii) if v.`pred` $=\perp$, then $\Pi(v) =\perp$; (iii) if v.`pred` $\neq\perp$, then $\Pi(v) = \Pi(v.\texttt{pred}), v$. We claim that the algorithm maintains the following invariant:

Invariant: Whenever the algorithm reaches the beginning of the `while`-loop (position (*) in the pseudocode), the following properties hold:

1. For every node $v \in V \setminus Q$ and for every node $w \in Q$, we have $v.\mathsf{d} \leq w.\mathsf{d}$.



$Q : (s : 0), (a : \infty), (b : \infty), (c : \infty), (d : \infty), (e : \infty)$

$Q.extractMin() = (s : 0)$

$s.neighbors() = (d : \infty), (e : \infty)$

$(d : 1) < (d : \infty) \Rightarrow d.d \leftarrow 1$

$(e : 4) < (e : \infty) \Rightarrow e.d \leftarrow 4$

$Q : (d : 1), (e : 4), (a : \infty), (b : \infty), (c : \infty)$

$Q.extractMin() = (d : 1)$

$d.neighbors() = (e : 4)$

$(e : 1 + 2) < (e : 4) \Rightarrow e.d \leftarrow 3$

$Q : (e : 3), (a : \infty), (b : \infty), (c : \infty)$

$Q.extractMin() = (e : 3)$

$e.neighbors() = (c : \infty)$

$(c : 3 + 3) < (c : \infty) \Rightarrow c.d \leftarrow 6$

$Q : (c : 6), (a : \infty), (b : \infty), (c : \infty)$

Abbildung 24.1: Dijkstra example.

2. For every node $v \in V$, we have that (i) if $\Pi(v) \neq \perp$, then $\Pi(v)$ is a path from s to v with length $v.\mathtt{d}$ and such that all vertices of $\Pi(v)$ except for possibly v lie in $Q \setminus V$; and (ii) if $\Pi(v) = \perp$, then $v.\mathtt{d} = \infty$;

3. For every node $v \in V \setminus Q$, we have $v.\mathtt{d} = d_G(s,v)$.

Initially, this invariant holds: for (1) and (3), note that initially all vertices are in Q, so $V \setminus Q = \emptyset$. For (2), note that the only node that has $\Pi(v) \neq \perp$ is s, and $\Pi(s) = s$ is a path from s to s of length $s.\mathtt{d} = 0$. For all other nodes $v \in V \setminus \{s\}$, we have $\Pi(v) = \perp$ and $v.d = \infty$.

Now, we show that the invariant is maintained by the `while`-loop. First, we consider invariant (1). In one iteration of the `while`-loop, there is exactly one node that moves from Q to $V \setminus Q$: the node v that is returned by the `extractMin`-operation. Since invariant (1) holds at the beginning of the `while`-loop, we have $v'.\mathtt{d} \leq v.\mathtt{d}$, for all $v' \in V \setminus Q$. Furthermore, since v is the element in Q with minimum key, we have that $v.\mathtt{d} \leq w.\mathtt{d}$, for all $w \in Q$. Thus, invariant (1) holds immediately after the `extractMin`-operation. We still need to show invariant (1) also holds at the end of the `while`-loop. For this, we note that the `d`-attributes of some nodes w may change during the inner `for`-loop. However, an inspection of the code shows that the `d`-attribute can only get smaller. Thus, the only critical nodes are nodes $w \in Q$ whose `d`-attribute is decreased in the inner `for`-loop. An inspection of the code shows that in this case, the new value of $w.\mathtt{d}$ will be $v.\mathtt{d} + \ell(v,w) \geq v.\mathtt{d}$, since $\ell(v,w) \geq 0$. This means that after the inner `for`-loop, we still have for all nodes $w \in Q$ that $v.\mathtt{d} \leq w.\mathtt{d}$. It follows that invariant (1) also holds at the end of the `while`-loop.

For invariant (2), we first argue that during an iteration of the `while`-loop, the `pred`- and `d`-attributes change only for nodes w that are still in Q. Indeed, we change the attributes only for nodes w where $w.\mathtt{d} > v.\mathtt{d} + \ell(v,w) \geq v.\mathtt{d}$, where v is the node that is returned by the `extractMin`-operation. By invariant (1), we have $v'.\mathtt{d} \leq v.\mathtt{d}$ for all $v' \in V \setminus Q$, so no such v' is affected. Now, it follows that invariant (2) is maintained for all nodes $v \in Q \setminus V$, because the invariant holds at the beginning of the `while`-loop and nothing changes. Similarly, Invariant (2) is also maintained for the node v that is returned by `extractMin`, because it holds at the beginning and the attributes of v are not changed. Finally, let w be a node from Q. There are two possibilities: (i) the `pred`- and `d`-attributes of w are changed in the `while`-loop. Then, $w.\mathtt{pred}$ is set to v, and $w.\mathtt{d}$ is set to $v.\mathtt{d} + \ell(v,w)$. Invariant (2) holds for w, because we already saw that invariant (2) holds for v, and because v has moved to $V \setminus Q$. (ii) the `pred`- and `d`-attributes of w do not change: either $w.\mathtt{pred} = \infty$, in which case everything holds, or $w.\mathtt{pred} \in V \setminus Q$, and we saw that in this case nothing changes for $w.\mathtt{pred}$, so $\Pi(w)$ is still a path from s to w of length $w.\mathtt{d}$. In summary, we have shown that invariant (2) is maintained for all vertices.

Finally, we consider invariant (3). Let w be a vertex in $V \setminus Q$, at the beginning of the `while`-loop. Since invariant (3) holds for w at the beginning, and since we already saw that the `d`-attribute of v does not change, we have that invariant (3) also holds for w at the end. Thus, the main task is to show that $v.\mathtt{d} = d_G(s,v)$, where v is the node returned by `extractMin`. In the first iteration of the `while`-loop, we have $v = s$, and $s.\mathsf{d} = 0 = d_G(s,s)$. Thus, suppose that $v \neq s$. Consider the situation just before v is removed from Q. First, suppose that $d_G(s,v) < \infty$, and let π be an arbitrary path from s to v. Since $v \neq s$, we know that s has already been removed from Q. Thus, the path π starts at a vertex in $Q \setminus V$, and it ends at a vertex in Q. It follows that π must move from $V \setminus Q$ to Q. Let (a,b) the first edge on π such that $a \in V \setminus Q$ and $b \in Q$, and let π' be the prefix of π that goes from s to a. Then, we have

$$\begin{aligned}
|\pi| &\geq |\pi'| + \ell(a,b) && \text{(all edge lengths are nonnegative)}\\
&\geq d_G(s,a) + \ell(a,b) && \text{(π' is a path from s to a)}\\
&= a.\mathsf{d} + \ell(a,b) && \text{(Invariant (4) holds for a)}\\
&\geq b.\mathsf{d} && \text{(b was considered when a was removed from Q)}\\
&\geq v.\mathsf{d}. && \text{(v has a minimum key in Q)}
\end{aligned}$$

Since π is an arbitrary path from s to v, it follows that $v.\mathsf{d} \leq d_G(s,v)$. By invariant (2), we have that $\Pi(v)$ is a path from s to v of length $v.\mathsf{d}$, and hence $v.\mathsf{d} \geq d_G(s,v)$. Thus, we have $v.\mathsf{d} = d_G(s,v)$, as desired. Finally, if $d_G(s,v) = \infty$, then there is no path from s to v, and by invariant (2), if follows that $v.\mathsf{d} = \infty$. In any case, invariant (3) holds for v.

Since the invariant is maintained throughout the algorithm, it follows that in the end, Dijkstra's algorithm correctly solves the SSSP-problem.

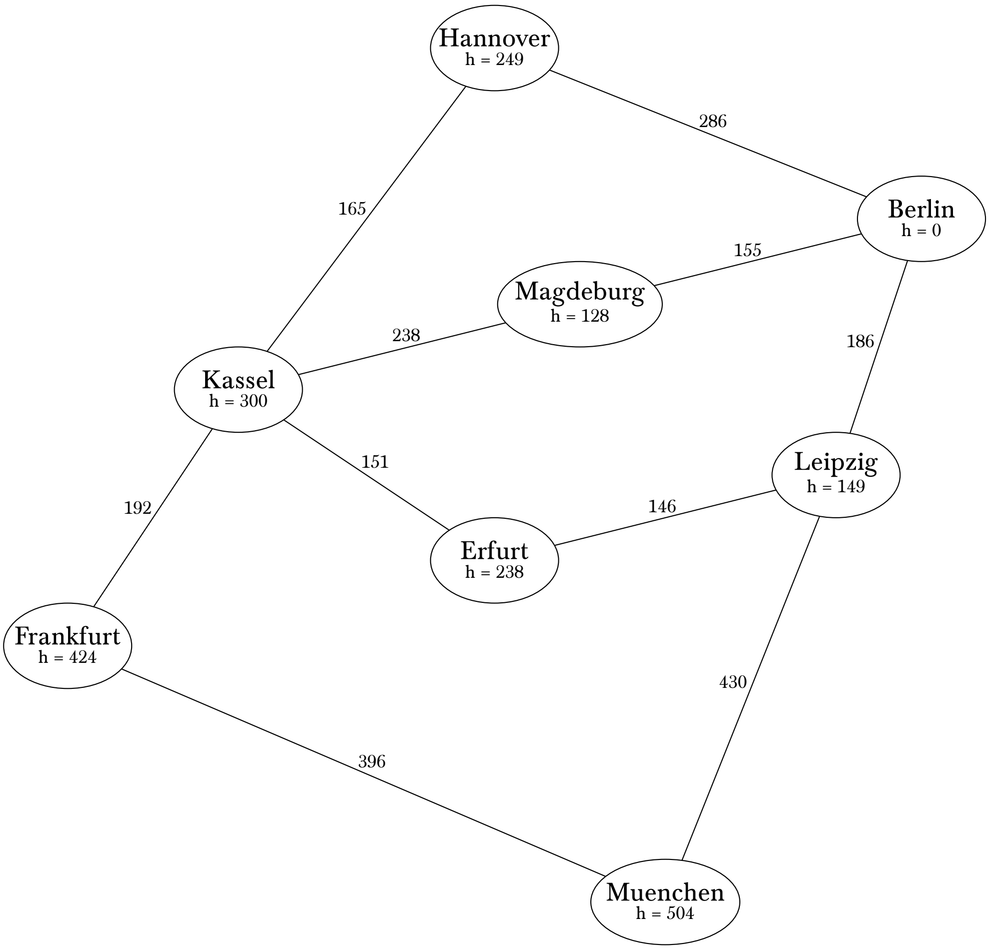
Remark: If we actually want to solve the SPSP-problem for two vertices s and t, we can run Dijkstra's algorithm from s and stop as soon as t is removed from Q. Our proof shows that then we have found a shortest path from s to t.

The A*-Algorithm. The A*-algorithm is a variant of Dijkstra's algorithm that is applicable for the following scenario: suppose we are given a directed graph $G = (V,E)$ with nonnegative edge weights $\ell : E \to \mathbb{R}_0^+$, and suppose we are given two vertices $s,t \in V$. We would like to find a shortest path from s to t. Now, however, we have some *additional* information on the shortest paths distances in G that comes from some understanding of the underlying structure in G.

For example, suppose that we would like to find a shortest path in a highway network G, where the vertices are cities. Then, we have a simple and fast way of estimating the distance between two cities v and w in the road network: simply take the geographic distance between two cities. Of course, this is not an accurate estimate. The distance in the highway network will usually be larger than the geographic distance. However, the geographic distance is much faster to compute, and we expect it to be reasonably close. Even more, when looking for a shortest path from s to t, the geographic distance could help us direct the search towards t. In its original version, Dijkstra's algorithm does not take this information into account, but looks in all directions simultaneously, even those that take us *away* from the target

TODO: Example

The A*-algorithm is supposed to incorporate this additional information into account and to help direct Dijkstra's algorithm into the direction of the target t. For this,



$Frankfurt \longrightarrow Muenchen : 396 - (424 - 504) = 396 + 80 = 476$

$Frankfurt \longrightarrow Kassel : 192 - (424 - 300) = 192 - 124 = 68$

$Kassel \longrightarrow Hannover : 165 - (300 - 249) = 165 - 51 = 114$

$Kassel \longrightarrow Magdeburg : 238 - (300 - 128) = 238 - 172 = 66$

$Kassel \longrightarrow Erfurt : 151 - (300 - 238) = 151 - 62 = 89$

$Kassel \longrightarrow Frankfurt : 192 - (300 - 424) = 192 + 124 = 316$

$Magdeburg \longrightarrow Berlin :$ 155 - (128 - 0) = 27

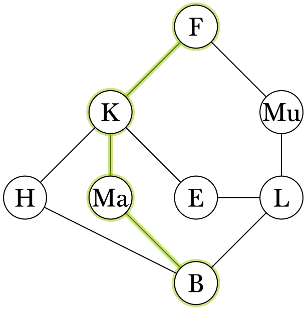


Abbildung 24.2: A*-example.

we first need to formalize the notion of *additional information*. This is done in the form of a *heuristic function* $h : V \to \mathbb{R}_0^+$ that assigns a nonnegative number to every vertex $v \in v$. The interpretation is that $h(v)$ constitutes an estimate of the distance from v to the target t. Using h, we would like to modify Dijkstra's algorithm so that it favors edges that take us closer to t over edges that takes us away from t. For this to work, the heuristic h needs to satisfy a simple mathematical property.

Definition: Let $h : V \to \mathbb{R}_0^+$ be a heuristic function. We say that h is *consistent* for G if for every edge $(v,w) \in E$, we have

$$h(v) \leq \ell(v,w) + h(w).$$

Consistency means that the estimated distance to t is compatible with the edges in G: if there is an edge from v to w, then our estimate for the distance from v to t is not larger than the estimate we get by taking the edge from v to w and adding the estimated distance from w to t.

Now, we can describe the A*-algorithm. Suppose we are given a directed graph $G = (V,E)$ with nonnegative edge weights $\ell : E \to \mathbb{R}_0^*$ and two vertices $s,t \in V$. Suppose further that we have a heuristic function $h : V \to \mathbb{R}_0^+$ that is consistent with G. Then, we define a function $\widehat{\ell} : V \to \mathbb{R}$ of *reduced edge weights* for G as follows: for $(v,w) \in E$, set

$$\widehat{\ell}(v,w) = \ell(v,w) + (h(w) - h(v)).$$

Then, we run Dijkstra's algorithm on G, using the reduced edge lengths $\widehat{\ell}$ instead of the original edge lengths ℓ. We stop the algorithm as soon as t is removed from the priority queue, and we return the resulting shortest path.

Intuitively, the reduced edge weights are supposed to favor edges that take us closer to t and to penalize edges that take us away from t: the term $h(w) - h(v)$ represents how much the estimated distance to t changes when following the edge (v,w). If we get closer to t, then $h(w) - h(v) < 0$. If we move away from t, then $h(W) - h(v) > 0$. Thus, in the reduced weights, edges go away from t get longer, and edges that go towards t get shortest. Since Dijkstra's algorithm discovers the shortest paths in the order of their lengths, the shortest path to t will be computed earlier.

To verify that A*-search is correct, we must check two things: (i) all the reduced edge weights are nonnegative, i.e., Dijkstra's algorithm is applicable; and (ii) a shortest path for the reduced edge weights corresponds to a shortest path from the original edge weights.

For (i), we note that since h is consistent, for every edge $(v,w) \in E$, we have

$$h(v) \leq \ell(v,w) + h(w) \Rightarrow 0 \leq \ell(v,w) + (h(w) - h(v)) = \widehat{\ell}(v,w),$$

so the nonnegativity of the reduced edge lengths is a direct consequence of consistency.

For (ii), let $pi : s = v_0, v_1, \ldots, v_k = t$ be an arbitrary path from s to t. Then, the length if π in the reduced weights is

$$\sum_{i=1}^{k} \widehat{\ell}(v_{i-1}, v_i)$$

$$
\begin{aligned}
&= \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + (h(v_i) - h(v_{i-1})) && \text{(by definition)} \\
&= \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \sum_{i=1}^{k} h(v_i) - \sum_{i=1}^{k} h(v_{i-1}) && \text{(rearranging the sum)} \\
&= \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \sum_{i=1}^{k} h(v_i) - \sum_{i=0}^{k-1} h(v_i) && \text{(shifting the index)} \\
&= \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + h(v_k) + \sum_{i=1}^{k-1} h(v_i) - \sum_{i=1}^{k-1} h(v_i) - h(v_0) && \text{(rearranging the sums)} \\
&= \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + h(v_k) - h(v_0) && \text{(two sums cancel)} \\
&= \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + h(t) - h(s). && (v_k = t \text{ and } v_0 = s)
\end{aligned}
$$

Thus, the reduced length of π is the original length of π, *plus* the term $h(t) - h(s)$. Crucially, this term is *the same* for *all* paths from s to t, independent of the inner vertices. Since the lengths of all paths from s to t are shifted by the same amount $h(t) - h(s)$, their relative order remains the same. Thus, a shortest path for the original weights is also a shortest path for the reduced weights, and vice versa.

Thus, the A*-algorithm is correct, and its worst-case behavior is never worse than for Dijkstra's algorithm. The hope is that if the heuristic h is chosen well, then the A*-algorithm will be much faster, because it is goes directly towards t. The challenge now is to find a good heuristic that is (i) meaningful and (ii) fast to compute.

To extreme heuristics are as follows: (a) the *trivial* heuristic $h_1 : V \to \mathbb{R}_0^+$ sets $h(v) = 0$, for all $v \in V$. Then, h_1 is consistent, and h_1 can be computed very quickly. However, h_1 is not meaningful: the reduced weights are the same as the original weights, and the A*-algorithm with h_1 is the same as Dijkstra's algorithm; (b) the *perfect* heuristic $h_2 : V \to \mathbb{R}_0^*$ sets $h_2(v) = d_G(v, t)$, for all $v \in V$. Then, h_2 is consistent, and h_2 is meaningful: the reduced length of a shortest path from s to t is 0, and it will be discovered almost immediately by the A*-algorithm. However, h_2 is not easy to compute. In fact, if we had h_2 available, we would not need to run the A*-algorithm in the first place.

Thus, we need to find a heuristic that lies somewhere between the two extremes h_1 and h_2. This depends on the special structure of the graph at hand, and it requires some creativity. As we saw, a classic example of a good heuristics is the geographic distance in a road network. Other examples come from the field of artificial intelligence, where the A*-search is often used to solve planning problems. **TODO**: Elaborate on this. **TODO**: Say something about A* with an admissible heuristic.

KAPITEL 25

Shortest Paths with Negative Weights

We now consider the SSSP-problem with negative edge weights.

Currency exchange. First, we consider an example where negative edge weights occur naturally: suppose we have a set of n currencies, $C_1, c_2, \dots, C_n$ (e.g., Euro, British Pound, Danish Kroner, Japanese Yen, Turkish Lira, etc.). For each pair of currencies C_i, C_j, we have an *exchange rate* $r_{ij} \geq 0$ that tells us how man units of currency C_j we get for one unit of currency C_i (the exchange rate r_{ji} is not necessarily related to the exchange rate r_{ij}.[1] Suppose further that we have one unit of currency C_1, and we would like to change this to currency C_n.

Of course, we can do this with a direct exchange, obtaining r_{1n} units of currency C_n. However, conceivably, there may be a better way of doing this. Possibly, one can realize a better exchange rate by going through a longer sequence of exchanges.[2]

In fact, we could be very lucky and discover a sequence of transactions that starts and ends in the same currency and that results in more units of currency than we

[1] An exchange rate of 0 tells us that for some reason a direct change is not possible.

[2] In reality, this would not be possible for a private person, because the transaction fees would eliminate the advantage of a sequence of multiple transactions, and the rates could fluctuate too quickly for the transactions to go through on time. For larger entities with a better access to the market, however, this may be a feasible strategy.

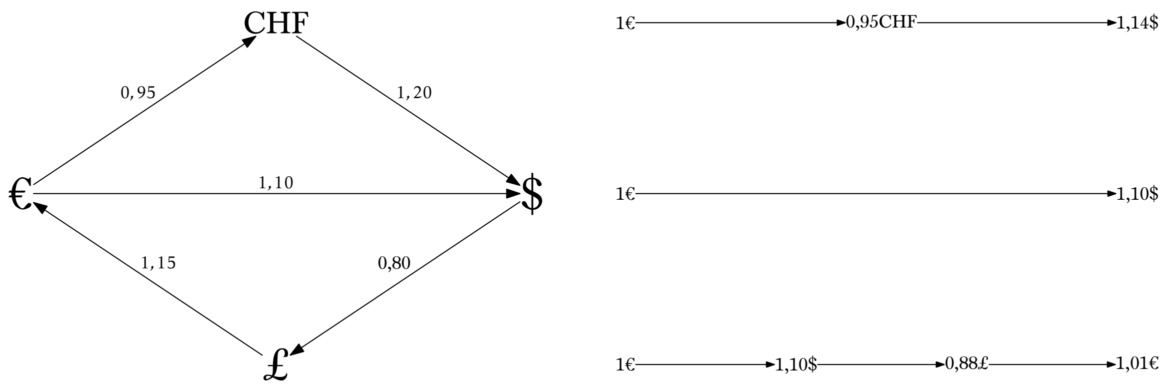


Abbildung 25.1: Arbitrage.

started with. This would, at least in theory, allows us to generate an infinite amount of money.

Now, let us formalize this problem. We define a weighted directed graph $G = (V,E)$ whose vertex set V is the set of currencies: $V = \{C_1, C_2, \dots, C_n\}$. There is a directed edge (C_i, C_j) for every pair of currencies that have a positive exchange rate $r_{ij} > 0$: $E = \{(C_i, C_j) \mid r_{ij} > 0\}$. The edges are weighted by the exchange rates r_{ij}. Now, a *transaction sequence* π from C_1 to C_n is a sequence of currencies $C_0 = C_{i_0}, C_{i_1}, \dots, C_{i_k} = C_n$ that starts with C_1, ends with C_n, and such that there is an edge from $C_{i_{j-1}}$ to C_{i_j}, for all $j = 1, \dots, k$. The *overall exchange rate* of π, denoted by $r(\pi)$ is the product

$$r(\pi) = \prod_{j=1}^{k} r_{i_{j-1}i_j}.$$

By assumption, we always have $r(\pi) > 0$. The goal is to find a transaction sequence from C_1 to C_n that *maximizes* the overall exchange rate. This problem is very similar to a shortest path problem, but not quite. There are two differences: (i) our goal is to *maximize* the weight of a path, not *minimize* it; and (ii) the weight of a path is the *product* of the individual weights, not the *sum*. However, it turns out that these differences are only superficial, and that we can interpret this problem as a shortest path problem in exactly the sense that we have seen in the previous chapters (but with negative edge weights).

First, we address difference (i) and show how to go from a maximization problem to a minimization problem. For this, we define a new weight function $r' : E \to \mathbb{R}^+$ as

$$r'(C_i, C_j) = \frac{1}{r(C_i, C_j)}, \quad \text{for all } (C_i, C_j) \in E.$$

Since our graph G contains directed edges only for pairs of currencies with positive weights, the weight function r' is well defined. Furthermore, let $\pi : C_0 = C_{i_0}, C_{i_1}, \dots, C_{i_k} = C_n$ be a transaction sequence from C_1 to C_n, and define the modified weight $r'(\pi)$ of π as

$$r'(\pi) = \prod_{j=1}^{k} r'(C_{i_{j-1}}, C_{i_j}).$$

By definition, we have

$$r'(\pi) = \prod_{j=1}^{k} r'(C_{i_{j-1}}, C_{i_j}) = \prod_{j=1}^{k} \frac{1}{r(C_{i_{j-1}}, C_{i_j})} = \frac{1}{\prod_{j=1}^{k} r(C_{i_{j-1}}, C_{i_j})} = \frac{1}{r(\pi)},$$

and for any two transaction sequences π, π', we have $r'(\pi_1) \leq r'(\pi_2)$ if and only if $r(\pi_1) \geq r(\pi_2)$.

This means that $\pi*$ is a transaction sequence with *maximum* weight with respect to r, then $\pi*$ is a transaction sequence with *minimum* weight with respect to r'. By changing the weights from r to r', we have changed our original problem into an equivalent minimization problem. This addresses (i).

To address (ii), we need to go from multiplication to addition. To do that, we again define a new weight function $r'' : E \to \mathbb{R}$ as

$$r''(C_i, C_j) = \log r'(C_i, C_j), \quad \text{for all } (C_i, C_j) \in E.$$

Now, we may have negative edge weights, because the logarithm of a number strictly between 0 and 1 is negative. Let $\pi : C_0 = C_{i_0}, C_{i_1}, \dots, C_{i_k} = C_n$ be a transaction sequence from C_1 to C_n, and define the weight $r''(\pi)$ of π as

$$r''(\pi) = \sum_{j=1}^{k} r''(C_{i_{j-1}}, C_{i_j}).$$

With these weights, we have

$$r''(\pi) = \sum_{j=1}^{k} r''(C_{i_{j-1}}, C_{i_j}) = \sum_{j=1}^{k} \log r'(C_{i_{j-1}}, C_{i_j}) = \log\left(\prod_{j=1}^{k} r'(C_{i_{j-1}}, C_{i_j})\right) = \log r'(\pi).$$

Since the logarithm is monotone increasing, it follows for any two transaction sequences π_1 and π_2 that $r''(\pi_1) \le r''(\pi_2)$ if and only if $r'(p_1) \le r'(p_2)$. This means that a shortest path with respect to the weight function r'' is also a shortest path with respect to the weight function r'. So finding an additive shortest path for r'' is equivalent to finding a multiplicative shortest path for r'. Thus, we have also addressed (ii) and turned our original problem into a classic shortest path problem with negative edge weights.[3]

Remark: In the second step, we have seen a general trick that uses the logarithm in order to turn multiplication into addition, based on the well-known rule $\log(a \cdot b) = \log a + \log b$. In other words, instead of multiplying two numbers, we can take their logarithms and then perform an addition. This trick is used extensively in the field of artificial intelligence. There, we often need to deal with probabilities, and it happens often that probabilities need to be multiplied together. The resulting numbers can get very small very quickly, leading to problems with floating point precision. Thus, we often deal with the logarithms of the probabilities instead. This has two advantages: (i) we can use addition instead of multiplication; and (ii) the magnitudes of the numbers involved do not get too small.

Bellman-Ford-Algorithm. Now, let us see how to solve the SSSP-algorithm for graphs with negative edges weights. Let $G = (V, E)$ be a directed graph, and let $s \in V$ be the source node. Let $\ell : E \to \mathbb{R}$ be a weight functions that may have negative edge weights, and suppose that G does not contain any negative cycles.

[3] This general process of transforming an instance of a new problem by stepwise modifications into an equivalent instance of a known problem is called a *reduction*. We will learn much more about reductions in *Grundlagen der Theoretischen Informatik*.

In this setting, Dijkstra's algorithm does not work. The reason is that in Dijkstra's algorithm, we require that once a vertex v is removed from the priority queue, the shortest path to v is known. However, this does not need to be the case if we have negative edge weights. Now, there may be a shorter path to v that has a prefix that is longer than $d_G(s,v)$, followed by a negative part. In this case, the shortest path to v is discovered only after the prefix has been computed, which may be after v has been removed from the priority queue. In such a case, it may be that shortest paths that involve v are not computed correctly.

To address this, we take a more abstract view of Dijkstra's algorithm, as follows: for every vertex $v \in V$, we have two attributes: the *tentative distance* v.d and the *tentative predecessor* v.pred. Initially, all tentative predecessors are $\perp$, the tentative distance s.d is 0, and all other tentative distances v.d are ∞. Our goal is to slowly improve the tentative distances and the predecessors until we have a shortest path tree for s. The main tool for this is a function improve(v, w), that can be called for any edge $(v,w) \in E$. The function improve uses the fact that there is an edge from v to w to improve our current guess for the shortest path to w, given our current guess for the shortest path to v:

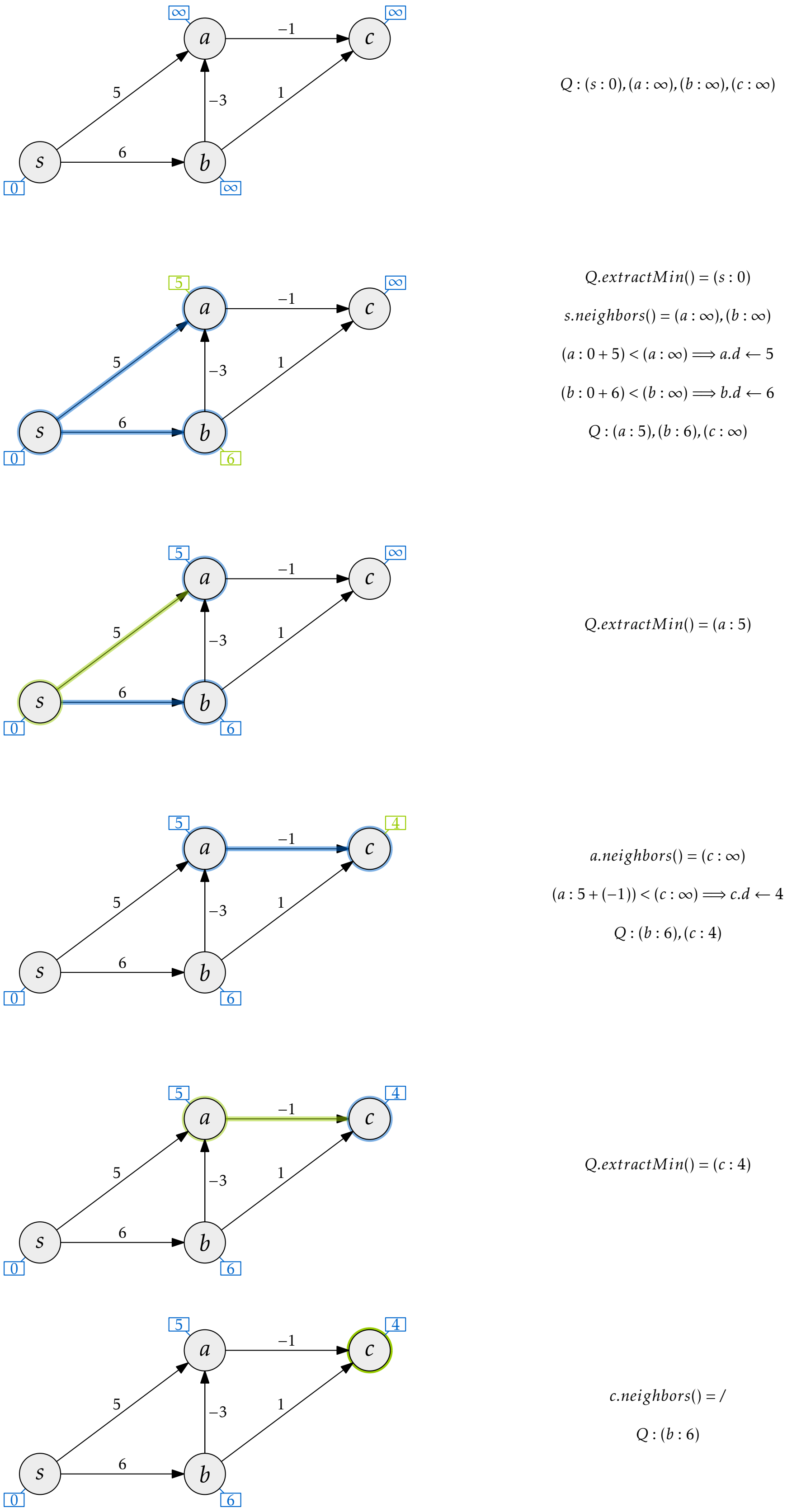
```
improve(v, w):
    if v.d + l(v, w) < w.d then
        w.d <- v.d + l(v, w)
 w.pred <- v
```

If, according to our current guesses, it is better to first go to v and then follow the edge (v,w), then we update the attributes for w accordingly. Otherwise, we do nothing.

We can think of Dijkstra's algorithm as a clever way to coordinate the calls to improve. We use a priority queue and the fact that the edge weights are nonnegative to call improve exactly once for every edge $(v,w) \in E$, namely at the time when we can be sure that a shortest path to v has been found.

If negative edge weights are allows, it is no longer clear how to identify this moment for an edge $(v,w) \in E$ when the shortest path for v has been found. But the solution is simple: simply call improve *multiple times* for each edge $e \in E$. A simple strategy is to just call improve for every edge, and to repeat this until no more changes take place. The pseudocode is as follows:

```
// Initialization, all distances are INFTY,
// all predecessors are NULL
for v in vertices() do
  v.d <- INFTY; v.pred <- NULL
// only s has distance 0
s.d <- 0
do
  // call improve on every edge
  for e = (v, w) in edges() do
     improve(v, w)
```



$Q : (s : 0), (a : \infty), (b : \infty), (c : \infty)$

$Q.extractMin() = (s : 0)$

$s.neighbors() = (a : \infty), (b : \infty)$

$(a : 0 + 5) < (a : \infty) \Longrightarrow a.d \leftarrow 5$

$(b : 0 + 6) < (b : \infty) \Longrightarrow b.d \leftarrow 6$

$Q : (a : 5), (b : 6), (c : \infty)$

$Q.extractMin() = (a : 5)$

$a.neighbors() = (c : \infty)$

$(a : 5 + (-1)) < (c : \infty) \Longrightarrow c.d \leftarrow 4$

$Q : (b : 6), (c : 4)$

$Q.extractMin() = (c : 4)$

$c.neighbors() = /$

$Q : (b : 6)$

Abbildung 25.2: Dijkstra fail.

```
while at least one call to improve had an effect
```

Each iteration of the `do-while`-loop takes $O(|E|)$ time. To analyze the correctness, and the running time, we need to understand how many iterations are necessary until the attributes converge (and to show that they actually correspond to shortest paths). For this, we first need an invariant for the `do-while`-loop. Recall from the previous chapter the definition of $\Pi(v)$ for a vertex $v \in V$: the path that is obtained by following the `pred`-pointers from v. With this definition, we can state and prove the invariant.

Lemma 25.1. *The* `do-while`*-loop maintains the following invariant for all* $v \in V$:

1. *we always have* $v.\mathsf{d} \geq d_G(s,v)$;
2. *if* $v.\mathsf{d} = d_G(s,v)$, *then* $v.\mathsf{d}$ *and* $v.\mathtt{pred}$ *do not change again;*
3. *if* $v.\mathsf{d} = d_G(s,v)$ *and if* $d_G(s,v) < \infty$, *then* $\Pi(v)$ *is a shortest path from* s *to* v, *and for all nodes* $w \in \Pi(v)$, *we have* $w.\mathsf{d} = d_G(s,w) < \infty$.

Beweis. The invariant holds initially: for the start node s, we have $s.\mathsf{d} = 0 = d_G(s,s)$ and $\Pi(s) = s$, which is a shortest path from s to s. For all other nodes $v \in V \setminus \{s\}$, we have $v.\mathsf{d} = \infty \geq d_G(s,v)$, and invariants (2) and (3) are vacuously true at the beginning.

Now, we show that the invariant is maintained throughout the loop. For this, we need to consider the effect of an invocation of `improve`. Thus, we suppose that the invariant holds for all nodes in V, and that we call the function `improve`(u,v) on an edge $(u,v) \in E$. The call changes only the attributes of v, and we show that it maintains the invariant.

First, we consider Invariant (1). Since only the attributes of v can change in the call to `improve`, Invariant (1) is maintained for all other nodes in $V \setminus \{v\}$, because it held before. If the call does not change anything, this also holds for v. Thus, it remains to consider the case that the call changes the attributes of v, and we need to show that the invariant is maintained for v. If the attributes of v change, then after `improve`(u,v), we have

$$v.\mathsf{d} = u.\mathsf{d} + \ell(u,v). \tag{25.1}$$

Since Invariant (1) holds before the call, we know that the other endpoint u fulfills

$$u.\mathsf{d} \geq d_G(s,u). \tag{25.2}$$

Combining (25.1) and (25.2), we get that after the call, we have

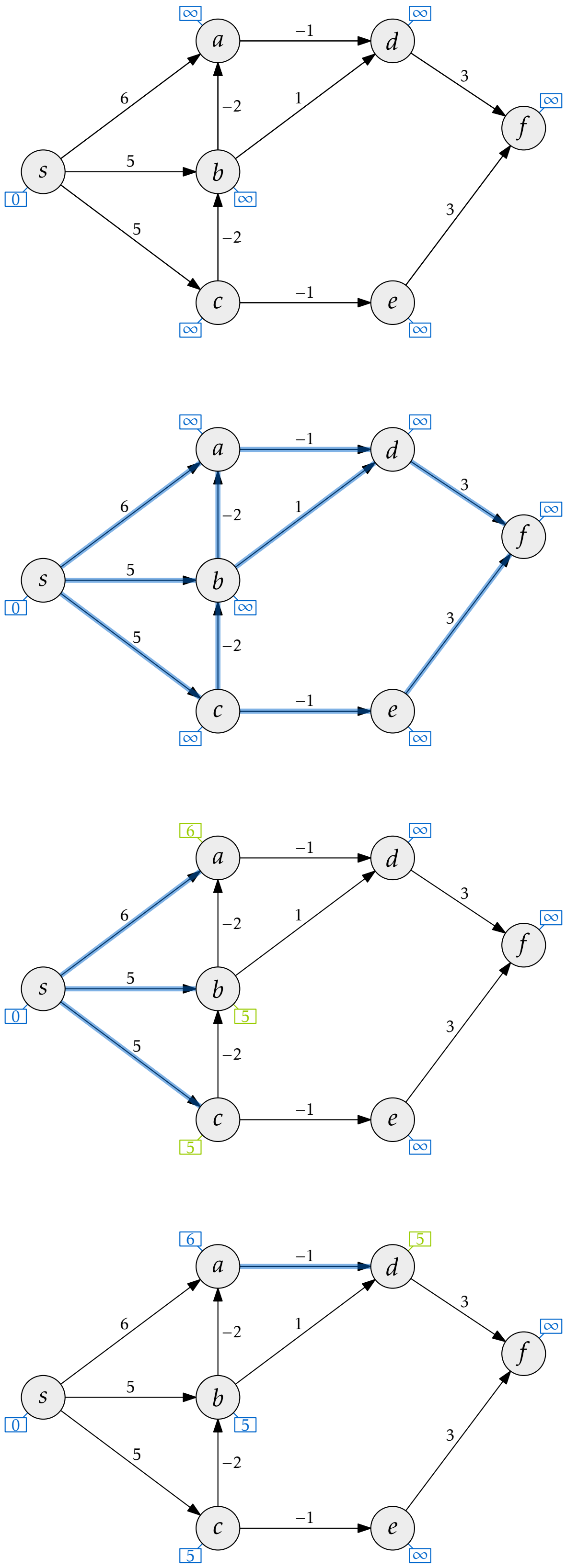
$$v.\mathsf{d} \geq d_G(s,u) + \ell(u,v). \tag{25.3}$$

Furthermore, the shortest path distance $d_G(\cdot,\cdot)$ fulfills

$$d_G(s,u) + \ell(u,v) \geq d_G(s,v), \tag{25.4}$$

because a shortest path from s to v is not longer than the path obtained by following a shortest path from s to u and then taking the edge from u to v. Thus, combining (25.3) and (25.4), we get that after the call, we have

$$v.\mathsf{d} \geq d_G(u,v).$$



Check all edges :

$(s,a),(s,b),(s,c),(a,d),(b,a),(b,d),(c,b),(c,e),(d,f),(e,f)$

$improve(s,a) = (a : 0+6) < (a : \infty) \Longrightarrow a.d \leftarrow 6$

$improve(s,b) = (b : 0+5) < (b : \infty) \Longrightarrow b.d \leftarrow 5$

$improve(s,c) = (c : 0+5) < (c : \infty) \Longrightarrow c.d \leftarrow 5$

$improve(a,d) = (d : 6+(-1)) < (d : \infty) \Longrightarrow d.d \leftarrow 5$

Abbildung 25.3: Bellman-Ford.

This means that Invariant (1) is also maintained for v.

Second, we look at Invariant (2). It is certainly maintained for all nodes in $V \setminus \{v\}$, because these nodes do not change. Furthermore, if $v.\mathsf{d} = d_G(s,v)$ before `improve`(u,v), then $v.\mathsf{d}$ and $v.$`pred` do not change: otherwise, the call `improve`(u,v) would strictly decrease $v.\mathsf{d}$. By Invariant (1), this is not possible if $v.\mathsf{d} = d_G(s,v)$.

Third, we consider Invariant (3). First, we note that for all nodes $a \in V$, the following holds: suppose that before `improve`(u,v), we have that (i) $a.\mathsf{d} = d_G(s,a) < \infty$ and (ii) $\Pi(a)$ is a shortest path from s to a with $b.\mathsf{d} = d_G(s,b) < \infty$, for all nodes $b \in \Pi(a)$. Then, this also holds after the call (by Invariant (2), none of these attributes changes). Thus, we only need to consider the situation that $v.\mathsf{d} > d_G(s,v)$ before the call and $v.\mathsf{d} = d_G(s,v)$ after the call. In this case, after the call, we have

$$\begin{aligned} d_G(s,u) + \ell(u,v) &\geq d_G(s,v) && \text{(by (25.4))} \\ &= v.\mathsf{d} && \text{(by assumption)} \\ &= u.\mathsf{d} + \ell(u,v). && (v.\mathsf{d} \text{ was changed by } \texttt{improve}), \end{aligned}$$

Subtracting $\ell(u,v)$, this gives

$$d_G(s,u) \geq u.\mathsf{d}.$$

By Invariant (1), we also have $d_G(s,u) \leq u.\mathsf{d}$, and hence $d_G(s,u) = u.\mathsf{d}$. Applying Invariant (3) to u, we see that $\Pi(u)$ is a shortest path from s to u with $w.\mathsf{d} = d_G(s,w)$ for all $w \in \Pi(u)$.. After `improve`(u,v), we have that $\Pi(v)$ is obtained by extending $\Pi(u)$ by the edge (u,v). This is a shortest path from s to v, as claimed, and Invariant (3) holds also for v. □

Now, the main insight is that after i iterations of the `do-while`-loop, we have found all shortest paths that consist of at most i edges:

Lemma 25.2. *For all vertices $v \in V$, the following holds: suppose that there is a shortest path from s to v that consists of at most i edges. Then, after i iterations of the* `do-while`*-loop, we have that $v.d = d_G(s,v)$.*

Beweis. The proof proceeds by induction on i, the number of iterations of the `do-while`-loop.

For the base case, we have $i = 0$, and we consider the situation right before the first iteration of the `do-while`-loop. The only node v for which the shortest path from s to v consists of 0 edges is $v = s$, and by the initialization we have that $s.d = 0 = d_G(s,s)$.

Now, suppose that the claim holds after i iterations of the `do-while`-loop, and we need to show that it also holds after $i+1$ iterations. Thus, let v be a vertex such that there is a shortest path π from s to v that has at most $i+1$ edges. If π has at most i edges, then, by the inductive hypothesis, the claim was already true before iteration $i+1$ of the `do-while`-loop, and by Invariant (2), it also holds after iteration $i+1$. Thus, suppose that π has $i+1$ edges, and let (u,v) be the last edge of π, i.e., u is the predecessor of v on π. Then, by the subpath-optimality property, the prefix of π that goes from s to u is a shortest path. This shortest path has i edges. Thus, be the inductive hypothesis, at the beginning of iteration $i+1$, we have $u.\mathsf{d} = d_G(s,u)$.

During iteration $i+1$, we call improve(u,v), and after this call will, we can be sure that $v.\mathsf{d} \leq u.\mathsf{d} + \ell(u,v) = |\pi| = d_G(s,v)$. By Invariant (1), we always have $v.\mathsf{d} \geq d_G(s,v)$, so at the end of iteration $i+1$, we have $v.\mathsf{d} = d_G(s,v)$, as claimed. □

The following theorem summarizes the properties of the Bellman-Ford algorithm.

Satz 25.3. *Let $G=(V,E)$ be a directed graph with weight function $\ell : E \to \mathbb{R}$, such that G contains no negative cycles. Set $s \in V$ be a source node. Then, the Bellman-Ford algorithm terminates after at most $|V|-1$ iterations and computes a shortest path tree for s. The running time is $O(|V| \cdot |E|)$.*

Beweis. For any node $v \in V$, a shortest path from s to v has at most $|V|-1$ edges. Thus, the claim follows from Lemma 25.2 and the fact that each iteration of the do-while-loop takes $O(|E|)$ steps. □

The algorithm can also be adapted to find a negative cycle in G, if it exists. **TODO:** Explain this.

KAPITEL 26

All-Pairs-Shortest-Paths

The *all-pairs-shortest-paths* problem is the variant of the shortest paths problem were we would like to find shortest paths for all pairs v, w of vertices in a given graph $G = (V, E)$.

More precisely, the problem is as follows: given a directed graph $G = (V, E)$ with edge weights $\ell : E \to \mathbb{R}$ such that G contains no negative cycles. The goal is to find a shortest path tree for every vertex $v \in V$.

Given the discussion in the previous chapters, there is a simple solution to this problem: simply run a SSSP-algorithm individually for each node $v \in V$. If all weights in G are nonnegative, we can use Dijkstra's algorithm for this, and we obtain a total running time of $O(|V| \cdot (|V| \log |V| + |E|) = O(|V|^2 \log |V| + |V| \cdot |E|)$. If G is connected, then we $|E|$ is between $\Theta(|V|)$ and $\Theta(|V|^2)$. Thus, the resulting running time is between $O(|V|^2 \log |V|)$ and $O(|V|^3)$, depending on the number of edges.

If G has general weights, then we need to use the Bellman-Ford algorithm, resulting in a running time of $O(|V| \cdot (|V| \cdot |E|)) = O(|V|^2 |E|)$. Depending on the number of edges, this is between $O(|V|^3)$ and $O(|V|^4)$.

We will now consider an algorithm that always requires $O(|V|^3)$ steps, independent of the number of edges. This is never worse than running Bellman-Ford for every vertex, and it is comparable to using Dijkstra, if the graph is *dense* (i.e., if the graph has $\Theta(|V|^2)$ edges. The algorithm is very simple and, unlike Dijkstra's algorithm, it does not need any sophisticated data structures.

The algorithm of Floyd-Warshall. The algorithm of Floyd-Warshall uses *dynamic programming* in order to solve the APSP-problem in a directed graph $G = (V, E)$ with weights $\ell : E \to \mathbb{R}$. The key to a dynamic-programming-solution is to find a suitable recursive formulation of the problem. In the Floyd-Warshall algorithm, this is done as follows: let $n = |V|$ and suppose that $V = \{1, \dots, n\}$.[1] Then, we define the following subproblems: for any $v, w \in V$, and for any $k \in \{0, \dots, n\}$, set

$$d_{vw}^{(k)} = \begin{array}{l} \text{the shortest length of a path from } v \text{ to } w \text{ that} \\ \text{uses only intermediate vertices from } \{1, \dots, k\}. \end{array}$$

[1] In the chapter on graph representation, we have already discussed that we can assume that our vertex set consists of numbers; otherwise, can simply fix an arbitrary numbering of the vertices and use this to identify them.

The *intermediate* vertices of a path π are all vertices of the path except for the first and the last vertex. In particular, if $k = 0$, we have $\{1,\dots,k\} = \emptyset$, and no intermediate vertices are allowed. If no path from v to w with the desired property exists, we define $d_{vw}^{(k)}$ to be ∞, Now, we can write down a recursive formula for $d_{vw}^{(k)}$, using recursion on k. We begin with $k = 0$. In this case, no intermediate vertices are allowed. If $v = w$, then $\pi : v$ is a path from v to v of length 0. If $v \neq w$ and if $(v,w) \in V$, then the only path from v to w without intermediate vertices consists of the edge (v,w). The length is $\ell(v,w)$. Otherwise, if $v \neq w$ and $(v,w) \notin V$, no such path exists, and the length is ∞. Thus, we have, for all $v,w \in E$:

$$d_{vw}^{(0)} = \begin{cases} 0, & \text{if } v = w, \\ \ell(v,w), & \text{if } v \neq w \text{ and } (v,w) \in E, \\ \infty, & \text{otherwise.} \end{cases}$$

Next, we discuss how to go from $k-1$ to k. Let $v,w \in V$, and let π be a shortest path from v to w among all paths that use only intermediate vertices from $\{1,\dots,k\}$. Now, there are two possibilities: (i) the path π does not use the vertex k. Then, π is also a shortest path from v to w among all paths with intermediate vertices from $\{1,\dots,k-1\}$. Thus, the length of π is $d_{vw}^{(k-1)}$; (ii) the path π uses the vertex $k+1$. Then, π uses k exactly once. By the subpath-optimality-property, we know that π consists of a shortest path π_1 from v to k, followed by a shortest path π_2 from k to w. Crucially, π_1 and π_2 do not have k as an intermediate vertex, but only vertices from $\{1,\dots,k-1\}$. Thus, we conclude that the length of π_1 is $d_{vk}^{(k-1)}$, and the length of π_2 is $d_{kw}^{(k-1)}$. In total, the length of π is $d_{vk}^{(k-1)} + d_{kw}^{(k-1)}$.

In conclusion, we now have two possibilities for what $d_{vw}^{(k)}$ could be. Since we are looking for a shortest path, the true value is the smaller of the two. Thus, we have, for all $v,w \in V$, and for all $k \in \{1,\dots n\}$.

$$d_{vw}^{(k)} = \min\left\{d_{vw}^{(k-1)}, d_{vk}^{(k-1)} + d_{kw}^{(k-1)}\right\}.$$

Once $k = n$, all vertices are allowed as intermediate vertices, Thus, the values $d_{vw}^{(n)}$ represent the true distances between the vertices in G.

As is common with dynamic programming algorithms, once the recurrence is known, it is quite straightforward to write the corresponding code. We just need a program that systematically computes all the $d_{vw}^{(k)}$, for $k = 0,\dots,n$. We can use a small observation to save space: since $d_{vw}^{(k+1)}$ relies only on the values for k, we do not need to store all the previous values. It suffices to have an $n \times n$ array that represents the values from the previous round. Now, the pseudocode is as follows:

```
// initialize an (n x n)-array A with the
// distances for k = 0
for i := 1 to n do
```

```
  for j := 1 to n do
    if i = j then
      A[i][j] <- 0
    else if (i, j) is an edge then
      A[i][j] <- length(i, j)
    else
      A[i][j] <- infty
for k := 1 to n do
  // B contains the distances for k - 1,
  // and our goal is to fill out A with the
  // distances for k
  copy A to B
  for i := 1 to n do
    for j := 1 to n do
      if B[i][j] <= B[i][k] + B[k][j] then
        A[i][j] <- B[i][j]
      else
        A[i][j] <- B[i][k] + B[k][j]
return A
```

The running time of the algorithm is $O(n^3)$, since the main work is done in three nested `for`-loops, each with n iterations. Since we use only two $(n \times n)$-arrays, the space is $O(n^2)$.

To obtain the shortest path trees for all the vertices $v \in V$, we can use an analogous recursion: for any $v, w \in V$, and for any $k \in \{0, \dots, n\}$, set

$$\pi_{vw}^{(k)} = \begin{array}{c}\text{the predecessor of } w \text{ on a shortest from } v \text{ to } w \text{ that}\\ \text{uses only intermediate vertices from } \{1, \dots, k\}.\end{array}$$

For $k = 0$, there are three cases for two vertices $v, w \in V$: (i) if $v = w$, then the shortest path from v to v consists only of v, and there is no predecessor; (ii) if $v \neq w$ and $(v, w) \in E$, then the shortest path from v to w with no intermediate vertices consists only of the edge (v, w), and the predecessor is v; and (iii) if $v \neq w$ and $(v, w) \notin E$, then there is no shortest path with no intermediate vertices. In particular, no predecessor exists. This gives:

$$\pi_{vw}^{(0)} = \begin{cases} v, & \text{if } v \neq w \text{ and } (v, w) \in E, \\ \perp, & \text{otherwise.} \end{cases}$$

To go from $k-1$ to k, we need to distinguish whether the shortest path from v to w with intermediate vertices from $\{1, \dots, k\}$ is obtained by (i) taking the shortest path from v to w with intermediate vertices from $\{1, \dots, k-1\}$; or by (ii) concatenating a shortest path from v to k with a shortest path from k to w, both with intermediate vertices from $\{1, \dots, k-1\}$. In both cases, the predecessor of w is the same as the predecessor on the shortest path that ends in w. We can distinguish which case occurs by looking at the d-values for $k-1$. We thus have

$$\pi_{vw}^{(k)} = \begin{cases} \pi_{vw}^{(k-1)}, & \text{if } d_{uv}^{(k-1)} \le d_{vk}^{(k-1)} + d_{kw}^{(k-1)}, \\ \pi_{kw}^{(k-1)}, & \text{otherwise.} \end{cases}$$

It is straightforward to extend the pseudocode for computing the π-values:

```
// NEW: Add an (n x n)-array C for the predecessors
for i := 1 to n do
  for j := 1 to n do
    if i = j then
      A[i][j] <- 0
      C[i][j] <- NULL
    else if (i, j) is an edge then
      A[i][j] <- length(i, j)
      C[i][j] <- i
    else
      A[i][j] <- infty
      C[i][j] <- NULL
for k := 1 to n do
  // D contains the predecessors for k - 1,
  // and our goal is to fill out C with the
  // predecessors for k
  copy A to B
  copy C to D
  for i := 1 to n do
    for j := 1 to n do
      if B[i][j] <= B[i][k] + B[k][j] then
        A[i][j] <- B[i][j]
        C[i][j] <- D[i][j]
      else
        A[i][j] <- B[i][k] + B[k][j]
        C[i][j] <- D[k][j]
return A, C
```

The running time is still $O(n^3)$, and the space is still $O(n^2)$.

Note: In *Grundlagen der Theoretischen Informatik*, we will see the Algorithm of Kleene. It is closely related to the algorithm of Floyd-Warshall and uses a very similar idea to construct an equivalent regular expression for a given deterministic finite automaton (we will learn in GTI what these words mean).

TODO: Add discussion of matrix multiplication, four Russians, fine-grained complexity.

KAPITEL 27

Minimum Spanning Trees

We turn to another classic algorithmic problem on graphs. Suppose we have a collection of cities $C_1, C_2, \dots, C_n$, and we would like to build a railroad network. In principle, we can build a connection between any pair of cities, but due to geographic realities, some connections can be much cheaper than others. Suppose that we have already hired a construction company, and for each pair C_i, C_j of cities, the company has determined an estimate ℓ_{ij} of the cost for building a direct railroad connection between i and j. Now, our task is as follows: we would like to select a set of railroad connections such that (i) it is possible to travel from each city to each other; and (ii) the total cost of the railroad connections is as small as possible.[1]

This problem can be formalized as follows: we are given an undirected, connected graph $G = (V, E)$ and a *weight function* $\ell : E \to \mathbb{R}^+$ that assigns a positive cost to each edge. The goal is to select a set $T \subseteq E$ of edges such that (i) the subgraph (V, T) of G is connected; and (ii) to total cost $\sum_{e \in T} \ell(e)$ of T is as small as possible, among all $T \subseteq E$ that lead to a connected subgraph.

Since all weights are positive, we see that the subgraph (V, T) cannot have any cycles: it there were a cycle C in (V, T), we could remove an arbitrary edge e of C. The resulting graph $(V, T \setminus \{e\})$ would still be connected, and the total edge weight would be strictly smaller than for (V, T). Thus, the subgraph (V, T) is a *spanning tree* for G. Because it has the smallest weight, it is called a *minimum spanning tree* (MST) for G.

We would now like to design an algorithm to construct a minimum spanning tree for G. Our approach is to try a *greedy* strategy. We would like to start with the empty set $A = \emptyset$, and to add edges to A, one by one, until A is a minimum spanning tree. We capture this idea in the following definition:

Definition: Let $G = (V, E)$ be an undirected, connected graph, and let $\ell : E \to \mathbb{R}^+$ be a positive weight function. A set $A \subseteq E$ of edges is called *safe* if there exists an MST $T \subseteq E$ such that $A \subseteq T$.

In other words, an edge set A is safe if and only if it is still possible to extend A to an MST for G by adding more edges. Since G is connected, an MST for G always exists, and hence $A = \emptyset$, the set that contains no edges, is safe. This is our starting point.

[1] In this version of the problem, we just care about connectivity. We do not consider the travel time between cities.

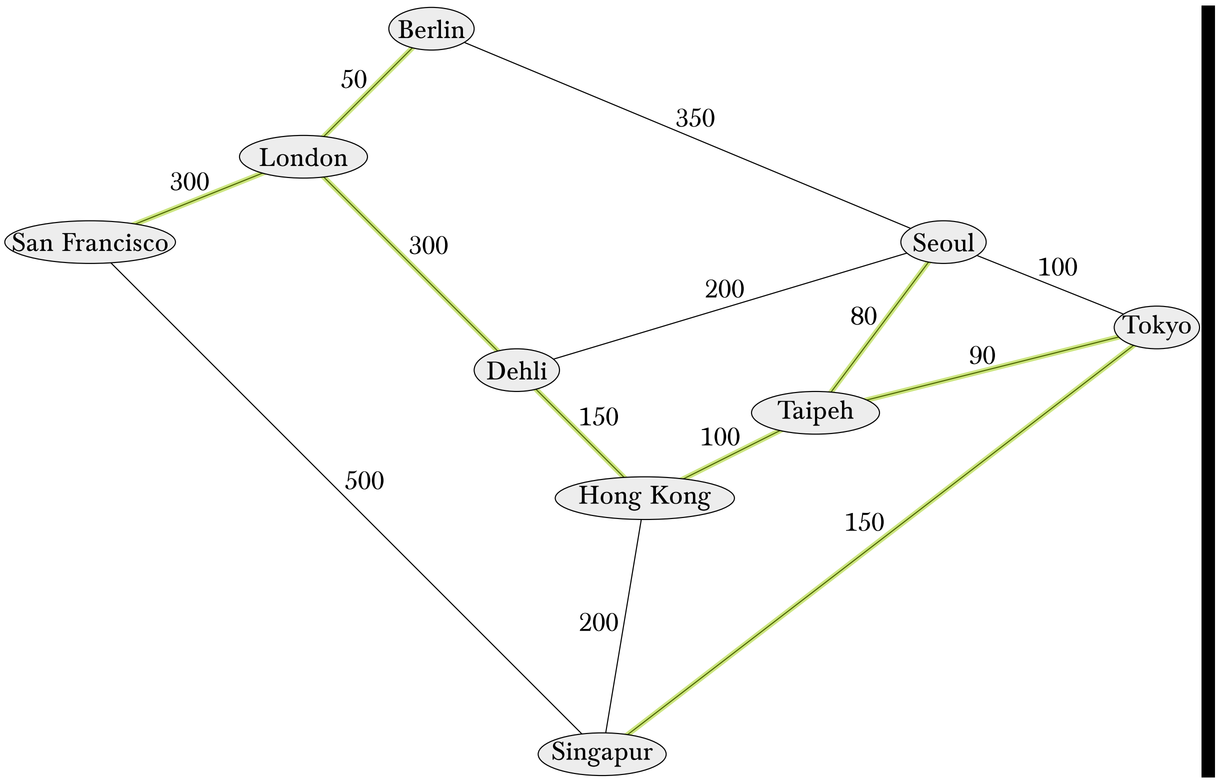


Abbildung 27.1: MST.

Now, all we need is a criterion that allows us to extend a given safe set by one more edge, resulting in a larger safe set. To state such a criterion, we need one more definition:

Definition: Let $G = (V, E)$ be an undirected, connected graph. Let $S \subset V$, $S \neq \emptyset, V$ be a nontrivial subset of vertices in G. Then, the partition $(S, V \setminus S)$ is called a *cut* of G. We say that an edge $e \in E$ *crosses* the cut $(V, V \setminus S)$ if e has exactly one endpoint in S and one endpoint in $V \setminus S$. Otherwise, we say that e *respects* the cut.

The following lemma tells us when we can add an edge to a safe set A: we choose a cut $(S, V \setminus E)$ that respects all edges in A, and we choose an edge of minimum weight that crosses the cut. More formally, the statement is as follows:

Lemma 27.1. *Let $G = (V, E)$ be an undirected, connected graph, and let $\ell : E \to \mathbb{R}^+$ be a positive weight function. Let $A \subseteq E$ be a safe set of vertices. Let $S \subset V, S \neq \emptyset, V$ be a nontrivial subset of vertices in V, such that all edges of A respect the cut $(S, V \setminus S)$.*

Now, the following holds: among all edges f that cross the cut $(S, V \setminus S)$, let e be an edge such that the weight $\ell(e)$ is minimum. Then, the set $A \cup \{e\}$ is safe.

Beweis. Since A is safe, there exists an MST $T \subseteq E$ such that $A \subseteq T$.

If $e \in T$, then we are done: in this case, we have $A \cup \{e\} \subseteq T$, and $A \cup \{e\}$ is safe, as witnessed by T.

Thus, suppose that $e \notin T$. Consider the suggraph $(V, T \cup \{e\})$. Since T is a tree, and since $e \notin T$, it follows that adding e to T creates a cycle. We call this cycle C, and we note that e is an edge of C. Since (i) C is a cycle; (ii) e is an edge of C; and (iii) e crosses the cut $(S, V \setminus S)$, it follows that C contains another edge $f \neq e$ that also crosses the cut $(S, V \setminus S)$. By the choice of e, we have $\ell(e) \leq \ell(f)$.

Now, consider the edge set $T' = (T \cup \{e\}) \setminus \{f\}$. Then, since e and f lie in a common cycle, the subgraph (V, T') is connected, and hence a spanning tree for G. Furthermore, because $\ell(e) \leq \ell(f)$, the total weight of T' is not larger than the total weight of T. Thus, T' is also an MST for G. Finally we have that $A \cup \{e\} \subseteq T'$. Thus, $A \cup \{e\}$ is safe, as claimed. □

As in Huffman codes, Lemma 27.1 is an *exchange lemma*: it shows that if we have an MST T, we can locally modify T to obtain a new MST T' that contains the desired edge e.

With Lemma 27.1 at hand, the strategy for a general MST-algorithm is clear: we start with the empty set $A = \emptyset$, and we successively add edges to A, until the MST is complete. In each step, we must construct a suitable cut $(S, V \setminus S)$ that respects A, and to find an edge of minimum weight that crosses this cut. There are different ways how this can be done. Depending on our choice of the cut, there are different concrete MST-algorithms that follow this general strategy. We will look at a few of them in more detail.

The algorithm of Prim-Jarník-Dijkstra. The algorithm of Prim-Jarník-Dijstra lets the MST grow from a single source vertex $s \in V$. The strategy is as follows: we pick an arbitrary vertex $s \in V$, and we set $A = \emptyset$ and $S = \{s\}$. In each step, we select the lightest edge e that crosses the cut $(S, V \setminus S)$. Let w be the endpoint os e that lies in $V \setminus S$. We add e to A and w to S.

In this way, we have that A defines a tree on the vertex set S, and in each step, we add one more vertex and one more edge to the tree. Since all edges of A have their endpoints in S, it follows that the cut $(S, V \setminus S)$ respects A. Since in each step, we pick a lightest edge that crosses the cut, we ensure that A stays safe throughout the algorithm. Thus, the resulting tree T is an MST for G.

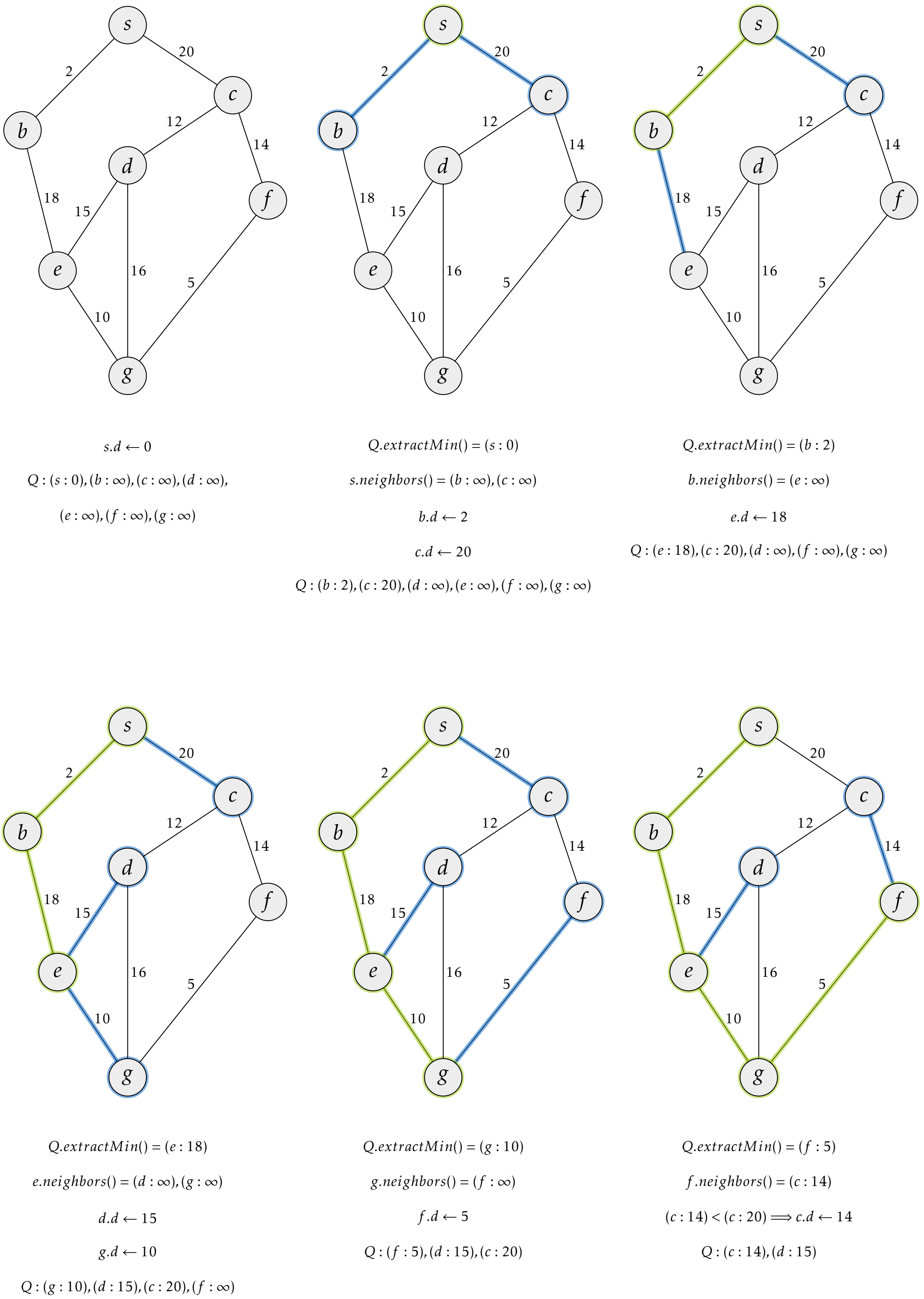
It remains to discuss the details of the implementation. We need to maintain the cut $(S, V \setminus S)$. Furthermore, in each step, we must be able to determine the lightest edge e that crosses the current cut. The idea is to store the vertices of $V \setminus S$ in a priority queue Q. The key for a vertex v in Q is the weight of the lightest edge that connects v to a vertex in S. We call this attribute $v.\mathtt{d}$. Furthermore, for each vertex v in Q, we store the other endpoint of the lightest edge that connects v to S. We call this attribute $v.\mathtt{pred}$. Initially, all the d-attributes are set to ∞, and all the pred-attributes are set to $\perp$.

In each step, we remove the vertex v with the smallest key from the Q. This means that we add v to S and the edge $\{v.\mathtt{pred}, v)\}$ to A.[2] Since v moves from $V \setminus S$ to S, there are new edges that cross the cut. These are exactly those edges e that have v as one endpoint and whose other endpoint remains in Q. For each such edge $e = \{v, w\}$, we must check whether e has now become the lightest edge that connects w to S. If this is the case, we must update the attributes $w.\mathtt{d}$ and $w.\mathtt{pred}$ for w. In the end, the edges of A, and hence the MST, can be deduced from the pred-attributes.

The pseudocode is as follows:

```
Q <- new PrioQueue
// initially, we put all vertices into Q
for v in vertices() do
  v.d <- INFTY; v.pred <- NULL
  Q.insert(v, v.d)
// We set s.d to 0, to ensure
// that s is removed first from Q
s.d <- 0
Q.decreaseKey(s, s.d)
while not Q.isEmpty() do
  v <- Q.extractMin()
  // we implicitly add v to S and the edge (v.pred, v) to A
  // we must check all the edges that go from v to vertices in Q
  for w in v.neighbors() do
    if w is in Q and l(v, w) < w.d then
      w.d <- l(v, w)
      w.pred <- v
      Q.decreaseKey(w, w.d)
```

[2] Actually, the situation in the first iteration is slightly different: there, we remove s from Q and add it to S. In this iteration, we do not add any edge to A. This is done to simplify the algorithm, so that no special treatment for s is necessary.



$s.d \leftarrow 0$

$Q : (s:0),(b:\infty),(c:\infty),(d:\infty),$

$(e:\infty),(f:\infty),(g:\infty)$

$Q.extractMin() = (s:0)$

$s.neighbors() = (b:\infty),(c:\infty)$

$b.d \leftarrow 2$

$c.d \leftarrow 20$

$Q : (b:2),(c:20),(d:\infty),(e:\infty),(f:\infty),(g:\infty)$

$Q.extractMin() = (b:2)$

$b.neighbors() = (e:\infty)$

$e.d \leftarrow 18$

$Q : (e:18),(c:20),(d:\infty),(f:\infty),(g:\infty)$

$Q.extractMin() = (e:18)$

$e.neighbors() = (d:\infty),(g:\infty)$

$d.d \leftarrow 15$

$g.d \leftarrow 10$

$Q : (g:10),(d:15),(c:20),(f:\infty)$

$Q.extractMin() = (g:10)$

$g.neighbors() = (f:\infty)$

$f.d \leftarrow 5$

$Q : (f:5),(d:15),(c:20)$

$Q.extractMin() = (f:5)$

$f.neighbors() = (c:14)$

$(c:14) < (c:20) \Longrightarrow c.d \leftarrow 14$

$Q : (c:14),(d:15)$

Abbildung 27.2: Prim.

We note that this algorithm is *almost exactly* the same as Dijkstra's algorithm for the SSSP-problem. There are only two differences in the inner `for`-loop: (i) in the `if`-test, we explicitly check whether w is in the priority queue, and the condition for $w.\mathsf{d}$ is slightly different ($\ell(v,w) < w.\mathsf{d}$ instead of $v.\mathsf{d}.\ell(v,w) < w.\mathsf{d}$); and (ii) the new value of $w.\mathsf{d}$ is computed differently ($\ell(v,w)$ instead of $v.\mathsf{d} + \ell(v,w)$).

In Dijsktra's algorithm, we are greedily growing a shortest path tree from s, in the Prim-Jarnḱ-Dijkstra-algorithm, we are greedily growing a minimum spanning tree from s. Due to the different definitions of these two trees, the algorithms are slightly different, but the main strategy is the same in both cases.

The analysis for the Prim-Jarnḱ-Dijkstra-algorithm is the same as for Dijkstra's algorithm. The main bottleneck lies in the implementation of the priority queue. Using the best available results for this, we achieve a running time if $O(|V|\log|V| + |E|)$.

Kruskal's algorithm. Kruskal's algorithm uses a more global strategy for selecting the next safe edge. The idea is to take in each step the *lightest* edge in E that crosses a cut that respects the current set A. How does such an edge look like? In general, the subgraph (V,A) is a *forest*: a graph on V that does not contain any cycles, but that consists of several connected components (some of them may be a single vertex). For an edge $e \in E \setminus A$, there is a cut $(S, V \setminus S)$ that respects A and that is crossed by e if and only if the endpoints of e lie in different components of the forest (V,A) (this cut must separate the components that contain the endpoints of e, and the other components can be assigned arbitrarily).

Thus, Kruskal's algorithm works as follows: sort the edges in E by weight, from the lightest edge to the heaviest edge. Set $A = \emptyset$. For each edge e in this order, check if e connects two different components of the current forest (V,A). If so, add e to A. Otherwise, do nothing. Continue, until all edges have been considered. In other words: in each step, Kruskal's algorithm adds the lightest edge to A that adds a new connection. The pseudocode is as follows:

```
A <- {}
Sort e by weight
for each edge e in sorted order do
  if e connects two different components of (V, A) then
    A <- A + {e}
return A
```

We still need to explain how to check whether an edge e connects two different components of (V,A). For this, we need a way to track the connected components of (V,A), as edges are added to A. More precisely, we need to support the following process: initially, we have $A = \emptyset$, and each component of (V,A) consists of a single vertex. Then, in each round, we are given an edge $\{v,w\}$ in E, and we need to *find* the components of (V,A) that contain the endpoints v and w. If these two components are distinct, we need to construct the *union* of the two connected components for v and w.

This is called the *union-find-problem*. The abstract problem is as follows: we are given a set $\mathcal{U} = \{1,\dots,n\}$ with n elements, and our task is to maintain a *partition*

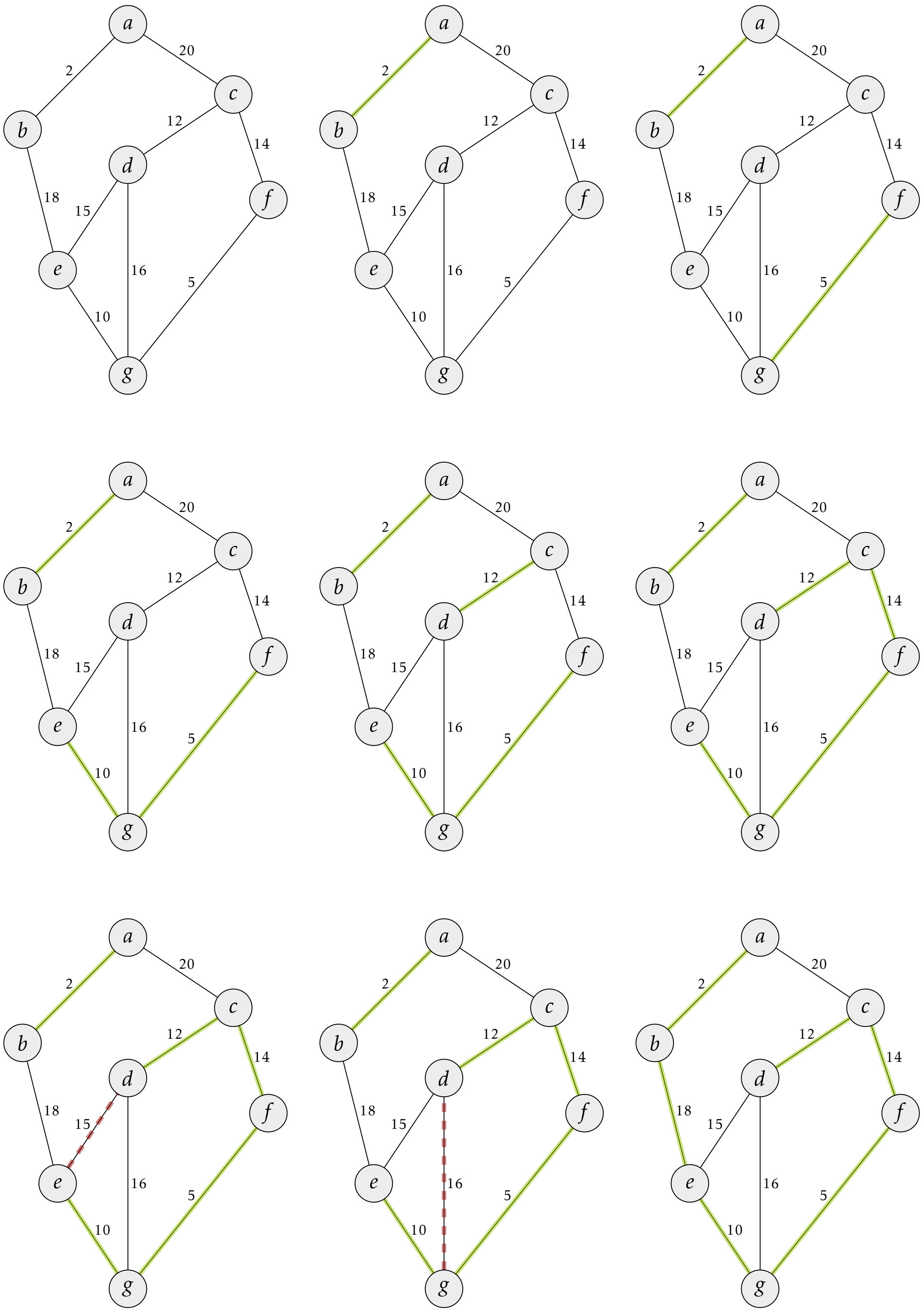


Abbildung 27.3: Prim.

$\mathcal{P} = \{U_1, U_2, \ldots, U_k\}$ of U under the following operations[3]: (i) `initialize`(U): create a new partition $\mathcal{P} = \{\{1\}, \{2\}, \ldots, \{n\}\}$ in which every set consists of a single element; (ii) `find`(u), for $u \in \mathcal{U}$: return a *representative element* $v \in U_i$, where U_i is the set in the current partition $\mathcal{P}$ that contains u. Crucially, we require that for all $u \in U_i$, the function `find`(a) returns *the same* representative $v \in U_i$ (note that it also follows that for $u_1, u_2 \in U$ that lie in *different* sets of the current partition, `find`(u_1) and `find`(u_2) return *different* representatives); and (iii) `union`(v_1, v_2), where $v_1 \in U$ is the representative element for a set U_i in the current partition and v_2 is the representative element for a different set U_j in the current partition: replace the sets U_i, U_j in the current partition $\mathcal{P}$ by their union $U_i \cup U_j$. That is, the new partition after `union`(v_1, v_2) is $\big(\mathcal{P} \setminus \{U_i, U_j\}\big) \cup \{U_i \cup U_j\}$.

Given a data structure that supports the operations of the union-find problem, we can now provide a more detailed pseudo-code for Kruskal's algorithm:

```
A <- {}
initialize(V)
Sort E by weight
for each edge e = {v, w} in sorted order do
  a = find(v)
  b = find(w)
  if a != b then
    A <- A + {e}
    union(a, b)
return A
```

Now, we analyze the running time of Kruskal's algorithm: it takes $O(|E| \log |E|)$ time to sort the edges in E. Then, we consider all edges. For each edge, we make two calls to `find`. Each time we perform a `union`-operation, we reduce the number of connected components by one. Thus, there are $2|E|$ calls to `find` and $|V|-1$ calls to `union` (we start with $|V|$ connected components, and we end with one connected component). Below, we will see how to implement the union-find structure such that each call to `find` takes $O(1)$ time and such that the total time for all the calls to `union` in $O(|V| \log |V|)$. Since G is connected, we have $|E| \geq |V| - 1$, so the total running time for Kurskal's algorithm is dominated by the sorting step: $O(|E| \log |E|)$.

The union-find-structure. We describe a simple way to implement the union-find structure. Let $\mathcal{U} = \{1, \ldots, n\}$ be given. We store the sets of the current partition $\mathcal{P}$ as a collection of linked lists, such that for each set, there is a linked list that contains the elements of the sets. We assume that all the linked lists have pointers to the first and to the last element in the list, so that we can join/concatenate two linked lists in constant time. Additionally, each element $u \in \mathcal{U}$ stores the following attributes:

[3] Recall that a partition of U is a set $\mathcal{P} = \{U_1, U_2, \ldots, U_k\}$ of nonempty, pairwise disjoint subsets of U such that every element of U occurs in exactly one set. Formally, we have (i) $U_i \subseteq U$ and $U_i \neq \emptyset$, for $i = 1, \ldots, k$; (ii) $U_i \neq U_j$, for $i, j = 1, \ldots, k$, $i \neq j$; and (iii) $\bigcup_{i=1}^{k} U_i = U$.

- u.rep: the *representative* of the set that contains u;
- u.setSize: the *size* of the set that u represents. This field is valid only if u is a representative of a set.
- u.set: the linked list that stores all the elements of the set that u represents. This field is valid only if u is a representative of a set.

Initially, each set in $\mathcal{P}$ consists of a single element. For each such set $\{u\}$, we create a linked list that contains only u. The representative of the set $\{u\}$ is u, and the size of the set is 1. Thus, the pseudocode for initialize(U) is as follows:

```
initialize(U):
  for u in U do
    u.rep <- u
    u.setSize <- 1
    u.set <- linked list that contains only u
```

The running time for initialize is $O(n)$. The find-operation is easy: since each element in U stores a direct reference to the representative of the current set that contains it, we can directly return the representative:

```
find(u):
  return u.rep
```

The running time for find is $O(1)$. The union-operation is slightly more complicated. We need to choose a representative for the new set, and we need to update all the representatives accordingly. To make union efficient, we choose the representative of the *larger* set as the new representative. In this way, we only need to update the representatives for the elements in the smaller set. Finally, we also need to concatenate the linked lists into a new list for the union. The pseudocode is as follows:

```
union(v1, v2):
  // make sure that v1 represents the larger set
  if v1.setSize < v2.setSize then
      swap(v1, v2)
  // update all the representatives in the smaller set
  for u in v2.set do
      u.rep <- v1
  // update the attribute for the new set
  v1.setSize <- v1.setSize + v2.setSize
  v1.set <- join(v1.set, v2.set)
  // invalidate the atributes of v2, since
  // v2 is no longer a representative of any set
  v2.setSize <- -1
  v2.set <- NULL
```

The running time of `union` is $O(1)$, except for the time that is needed to update the representatives for the set of v_2. In a single call, this may take a long time. However, we claim that over all calls to `union`, the total time to update the representatives is $O(n \log n)$.

Indeed, consider an element $u \in U$. Every time that u.`rep` changes, this means that the set U_i that contains u is united with a set U_j that is at least as large as U_i. In other words, if u.`rep` changes, then the size of the set that contains u increases by a factor of at least 2. Since u starts out in a set of size 1, and since u ends in a set of size at most n, this can happen at most $O(\log n)$ times. It follows that for each element in U, the representative changes at most $O(\log n)$ times, and hence the total time for all calls to `union` is $O(n \log n)$.

TODO: Say something about the inverse Ackermann function.

Boruvka's algorithm. Boruvka's algorithm is the third classic MST-algorithm. It is in fact the oldest MST-algorithm, and it can be seen as a combination of Kruskal's algorithm and the algorithm of Prim-Jarnḱ-Dijkstra. Like Kruskal's algorithm, Boruva's algorithm maintains a forest (V, A), instead of a single tree. Like Prim's algoirthm, Boruvka's algorithm makes sure that each tree in the forest (VA) grows in each step.

The details are as follows: we assume that all edge weights in G are pairwise distinct.[4] As usual, we start with the empty edge set $A = \emptyset$. In each step, we have a current forest (V, A), and for each connected component C of A, we pick a lightest edge e_C among all edges that have exactly one endpoint in C. Then, we add all these edges e_C to A.

In other words, Boruvka's algorithm picks one safe edge for each connected component of the current forest (V, A), and it adds all of them to A. Indeed, if e_C is a lightest edge that has exactly one endpoint in a connected component C of (V, A), then the cut $(C, V \setminus C)$ respects A and has e_C as a lightest crossing edge. This means that e_C is safe for A. One can show that if all edge weights in G are pairwise distinct, then the MST of G is unique, so that it is correct to add all these safe edges simultaneously. The pseudo-code for the algorithm is as follows:

```
A <- {}
while |A| < n - 1 do
  B <- {}
  for each connected component C of (V, A) do
    find the lightest edge e with exactly one endpoint in C
    add e to B (if e is not in B yet)
  add all edges in B to A
```

We argue that each iteration of the `while`-loop can be implemented in $O(|E|)$ time:

[4] By a standard trick, we can make sure that this is always the case: we number the edges of G arbitrarily from 1 to $|E|$, and if two edges have the same weight, we break the tie by preferring the edge with the smaller index.

- First, we use BFS in (V, A), in order to find the connected components. With each vertex $v \in V$, we store the id of the connected component that contains v. This takes $O(|V| + |A|) = O(|E|)$ time, since $|V|, |A| = O(|E|)$.

- Second, for each connected component C if (V, A), we find the lightest edge e_C that has exactly one endpoint in C. For this, we iterate over all edges $e \in C$, and we use the component ids that are stored with the endpoints of e to identify the incident components of e. If these components are distinct, we use the component ids as indices in an array in which we store the lightest edge that is incident to each connected component, updating this edge if necessary. All this can be done in $O(|E|)$ time.

Now, we show that the number of iterations in Boruvka's algorithm is at most $O(\log |V|)$.

Lemma 27.2. *After i iterations of the `while`-loop, each connected component of (V, A) has at least 2^i nodes.*

Beweis. We use induction on i. Initially, before the first iteration of the `while`-loop, we have $A = \emptyset$, and each connected component of (V, A) has exactly $1 = 2^0$ vertices. Thus, the statement holds for $i = 0$.

For the inductive step, we assume that before iteration $i + 1$ of the `while`-loop, each connected component in (V, A) has at least 2^i nodes. Then, during the iteration, each component is united with at least one other component (it may be more, but certainly at least one). Thus, after iteration $i + 1$, each remaining connected component has at least $2 \cdot 2^i = 2^{i+1}$ vertices. This finished the inductive step from i to $i + 1$. □

Since a connected component can have it most $|V|$ vertices, Lemma 27.2 implies that after $O(\log |V|)$ iterations of the `while`-loop, there is only one connected component in (V, A), which means that we are done. It follows that the running time of Boruvka's algorithm is $O(|E| \log |V|)$.

More on MST-algorithms. **TODO**

KAPITEL 28

Graphs and Games

Graph algorithms can be used to solve simple puzzles and to play games. There are many kinds of games, and there are many approaches for solving them with a computer. We will look at a few simple examples to illustrate the ideas, more details will be covered in advanced classes on artificial intelligence.

As a first example we will look at *deterministic*, *one-player* games with *perfect information*. Here, "deterministic" means that there is no randomness involved (i.e., no die is cast), and "perfect information" means that at each point of the game, the player has all the information about the current state (as opposed to, e.g., the card game solitaire, where some cards are hidden). Typical examples include:

- `peg solitaire`: we a given a board with pegs. Initially, there is a single hole. In each move, the player can jump with one peg over an adjacent peg. The peg that was jumped over is removed from the board, creating a new hole. The goal is to find a sequence of moves that results in a board that contains only a single peg.

- `Soduko`: we are given a 9x9 grid that contains a few digits between 1 and 9 in its cells . In each move, the player can write a new digit into a grid cell. The goal is to find a way to put the digits such that each row, column, and small 3x3 grid contains every digit from 1 to 9 exactly once.

- *sliding puzzles*: we are given a board that has the form of a 4x4 grid, with 15 movable blocks and one hole. In each move, the player can move a block that is adjacent to the hole into the hole. The blocks contain a picture that is scrambled. The goal is to find a sequence of moves that reconstitutes the picture in its original shape.

- *Sokoban*: a warehouse worker needs to shift crates in a labyrinthine warehouse. In each move, the worker can push a crate in a certain direction. The goal is to arrange the crates in a given configuration.

- *FreeCell* a well-known card-game that was shipped with Windows 95.

We can notice that all these games have a common abstraction: there is a *board* that contains all the information about the game, and at each point in time, the board can be in a certain *configuration*. The game proceeds in *moves* that are executed by the player, affecting the configuration of the board. There goal is to find a sequence of moves that transforms a *starting* configuration into a *winning* configuration.

With this abstraction in mind, it is easy to interpret these games as a reachability problem in graphs: let $G = (V, E)$ be the *game graph*. The nodes of G are all possible configurations of the board. There is an edge from configuration v to configuration w if and only if a single move transforms v into w. Depending on the game, the graph G can be directed or undirected (e.g., in peg solitaire, the graph is directed, because a move cannot be undone; in the sliding puzzle, the graph is undirected, because the possible block slides are symmetric).

Now, the problem is as follows: given a game graph G and a starting configuration s, find a path of moves in G that least to a winning configuration t. With our knowledge of graph algorithms, this now seems like a very easy task: for example we could use BFS, DFS, A*-search, or any other graph search algorithm.

However, a closer consideration shows that things are not that easy. Up to now, when describing our graph algorithm, we have always assumed that the graphs are given *explicitly*, as an adjacency list or an adjacency matrix. However, in the setting of games, this assumption is no longer realistic. Game graphs can be huge, and the time for explicitly generating all vertices and edges will be prohibitive. There are several strategies to deal with this:

First, when executing a graph search, the typical situation is that we are at a current node v, and that we need to consider all outgoing edges from v. So far, this was done by simply looking at the adjacency list for v or at the row for v in the adjacency matrix. Now, however we need an algorithm to generate the out-neighbors of v, once v is given. In the context of games, this is usually easy to do: we represent v in such a way that the corresponding configuration of the board can be deduces easily, and then we use our knowledge of the rules of the game to execute all possible moves from v, generating all the neighbors. Thus, we typically do not need to know the whole graph in advance, but we can generate the edges whenever they are needed.

Second, our graph search algorithms so far have assumed that we can store explicit information with all the vertices in the graph (e.g., the `visited`-attribute in BFS or DFS). In a game graph, this is no longer feasible, since the number of possible vertices is just too large. Thus, we need another strategy to maintain this information. For example we could use a dictionary to store all the vertices that have actually been visited, hoping that the graph search will succeed before too many vertices have been explored. This could be combined with an A*-search with a good heuristic, again in the hope of reducing the number of vertices that are explored. Another approach would be to give up on the attributes altogether, e.g., by doing a variant of DFS that can visit vertices multiple times (this is typically called *backtracking*.

There are many possible heuristics and optimizations that we can try, but in the end the underlying problems can be very difficult to solve. The art of finding the right heuristic for a given problem lies at the core of the field of artificial intelligence, and it requires a lot of experience and creativity. In later classes, you will see more of this (and also learn more about complexity theory that tries to explain why these problems are so hard).

Next, we consider a more general class of games: deterministic, *two-player* games with perfect information. The main difference now is that there are *two* players that play against each other. Again, *deterministic* means that there is no randomness (i.e., no die, unlike, say in Mensch-Ärgere-Dich-Nicht) and that the complete state of the game is known to all the players at any point in time (unlike, e.g., in UNO). Typical examples of such games are

1. Chess,
2. Checkers,
3. Go,
4. Tic-Tac-Toe,
5. Vier gewinnt.

Again, we can give a common abstraction for all these games: as before, there is a board that contains all the information about the game, and at each point in time, the board can be in a certain *configuration*. The game proceeds in *moves* that are executed by one of the players, affecting the configuration of the board. The players take *turns*, i.e., the moves alternate between Player 1 and Player 2. The information which player moves next is part of the configuration. There is a fixed *starting configuration* that describes the initial configuration of the board, and we assume that Player 1 moves first (i.e., in the starting configuration, it is Player 1's turn). There are certain configurations that are designated as *final configurations*. Once a final configuration is reached, the game is over. To determine who one, there is a function Ψ that assigns an integer to every final configuration. Typically, this integer represents the *score* for Player 1: if it is positive, then Player 1 wins, if it is negative, then Player 1 loses, and if it is zero, the game has finished in a draw. We assume that the game is *zero-sum*, i.e., the score for Player 2 is the negation of the score for Player 1.

These games are typically modelled as *game trees*: the root represents the starting configuration, and it is Player 1's turn. The children of the root are given by all possible configurations that can be obtained by a single move of Player 1 in the starting configuration. In every such child, it is Player 2's turn. For each node v at the second level, the children of v consist of all configurations that result from a single move of Player 2, and in each such child, it is Player 1's turn. This continues further down the tree, the layers alternating between Player 1 and Player 2. Once a final configuration is reached, there are no more children; these are the leaves of the tree.

In principle, the game tree can be infinite. This can happen if the game allows for circular sequences of moves that do not result in any progress. In the following, however, we will assume that the game tree is finite, e.g., by imposing a rule that a game is a draw if there is no progress within a certain number of moves (e.g., this kind of rule is present in chess).

Now, with the definition of the game tree, we can formally state an algorithmic problem that we need to solve: suppose we are given a node v of the game tree where it is the turn of Player j. Determine the best possible *score* that Player j can achieve, *assuming that the opponent plays optimally*. Furthermore, determine an *optimal move* that achieves this score.

Some explanations are in order to understand what this means: first, note that since we consider zero-sum-games, a best possible score for Player 1 is a score that is *as large as possible*, whereas a best possible score for Player 2 is a score that is *as small as possible*. In other words, the goal of Player 1 is to *maximize* the score, whereas the goal of Player 2 is to *minimize* the score. For this reason, the nodes in the game tree where it is the turn of Player 1 are called *max-nodes*, and they are represented by upward triangles; whereas the nodes where it is the turn of Player 2 are called *min-nodes*, and they are represented by downward triangles. Second, let us explain the notion of a "best possible score". Suppose we are in a node v, and it is the turn of Player 1. Then, Player 1 can *achieve* score k in v if and only if there is a move in node v that leads into a node v_2 such that no matter which move Player 2 chooses in node v_2, there is always a counter-move for Player 1 (depending on Player 2) for which Player 1 can achieve *at least* score k. Unrolling the definition, this means that *there exists* a child v_2 of v such that *for every* child v_3 of v_2, *there exists* a child v_4 of v_3, such that *for every* child v_5 of v_4, *there exists* a child v_6, etc, such that eventually every such sequence of configuration ends up in a final configuration with score at least k. For Player 2, the definition is similar, the only difference being that the final score should be *at most* k. Now, the best possible score for node v is the best possible score that the current player can achieve at node v.

After this discussion, we can now derive a simple recursive algorithm that determines the best possible score that can be achieved for a given node v in the tree. The recursion is very simple: first, suppose that v represents a final configuration. Then, the best possible score for v is given by the final score $\Psi(v)$ for v. Next, suppose that v is a max-node. This means that it is the turn of Player 1, and the goal is to achieve a score that is as large as possible. Let $w_1, \dots, w_j$ be the children of v. All these children are min-nodes, where it is the turn of Player 2. Suppose that for each child $w_1, \dots, w_j$, we can recursively compute the lowest possible score $s_1, \dots, s_j$ that Player 2 can achieve when playing from this child. Then, the best score that Player 1 can achieve from v is by making a move that maximizes this score, i.e., the move that is as bad for Player 2 as possible. Similarly, if v is a min-node, it is the turn of Player 2, and the goal is to achieve a score that is as small as possible. For each child of v, we can recursively determine the largest possible score that Player 1 can achieve from this child, and we pick the child that minimizes the score. The pseudocode for this algorithm is as follows:

```
// visit a final node
final-visit(v):
    // simply return the final score for v
    return psi(v)
```

```
//visit a max-node
max-visit(v):
    max = -infty
    // for each child w, determine the lowest possible score
    // that Player 2 can achieve from w, and pick the child
    // where this lowest possible score is as large as possible
    for each child w of v do
        if w is a final configuration then
            child_score <- final-visit(w)
        else
            child_score <- min-visit(w)
        if child-score > max then
            max <- child-score
    return max

//visit a min-node
min-visit(v):
    min = infty
    // for each child w, determine the largeest possible score
    // that Player 1 can achieve from w, and pick the child
    // where this largest possible score is as small as possible
    for each child w of v do
        if w is a final configuration then
            child_score <- final-visit(w)
        else
            child_score <- max-visit(w)
        if child-score < min then
            min <- child-score
    return min
```

This algorithm is called the *minimax-algorithm*. It constitutes of a simple post-order traversal of the tree and determines for each possible configuration in the game tree the optimal score. Given the optimal scores, it is also easy to determine the best possible move for each given configuration.

As in the single-player case, this solves the problem completely, except for the fact that the game trees for realistic games are prohibitively large, so that it is far from feasible the execute the minimax-algorithm in its entirety.

Again, there is a whole array of possible tricks that we can use to improve the running-time of the minimax-algorithm and to reduce the size of the search space. We list a few such tricks:

First, we can try to avoid unnecessary duplication of work: it can happen that certain configurations are repeated throughout the game tree,because the same configuration can be reached by different sequences of moves from the starting configuration. Instead of recomputing the optimal score for each such repetition from scratch, we can maintain a dictionary with all the configurations that we have processed so far, computing the optimal scores only for new configurations. Going even further, there may be *symmetries* between configurations, i.e., there may be configurations that look different superficially, but that are essentially the same (e.g., they can be obtained from each other by mirroring the board). By checking for such symmetries, and by avoiding a recomputation if possible, we can further reduce the number of distinct configurations that need to be processed.

Second, we can try to limit the search depth. Instead of searching the whole game tree until a final configuration is reached, we can cut off the search after a certain (pre-determined) number of moves. Every configuration that is reached after a certain number of moves is treated as a final configuration in the minimax-algorithm. The obvious problem is now, we do not have a final score for these configurations. Instead, we need to introduce a *heuristic function* that assigns to each possible configuration of the board a value that can be used as an estimator for the final score. Instead of the final score, we use the heuristic score, and the minimax-algorithm only provides a way to reach a configuration that achieves the best possible heuristic score for the current player. This means in particular that the minimax-algorithm is no longer guaranteed to be optimal; the quality of the result depends on the quality of the heuristic. We achieve a faster algorithm at the cost of the quality of the solution.

Third, there is a way to eliminate moves from consideration, without sacrificing the quality of the solution. This strategy is called (α, β)*-pruning*. The idea is as follows: suppose we explore the game tree starting from the root r, and suppose that we are currently visiting a max-node v. For each node w along the path from r to v, we are currently searching for a best possible move, and we have already processed some children of w and have a *tentative* value for the best possible score for w (more precisely, this is represented by the current values of the `max`- and the `min`-variables in the `max-visit`/`min-visit` calls from r to v). Now, suppose that we have just finished processing a child of the current max-node v, and that this results in increasing the tentative score of v to k. Suppose further that along the path from r to v, there is a min-node whose tentative score is smaller than k. Then, we claim that we this means that we can immediately stop our exploration of v and return to the parent node. The reason is as follows: given that the tentative score for node v is at least k, we know that if the game reaches configuration v, Player 1 will certainly have a move that ensures a final score of at least k. However, we know that in a configuration w that is encountered on the way to configuration v, there exists a move for Player 2 that ensures a score that is less than k. Thus, we know that Player 2 can always force a score that is less than k, and hence we will never reach configuration v, if Player 2 plays optimally.

To implement this idea, we introduce two additional parameter that are passed along during the search of the game tree: α and β. Here, α is the highest possible score that Player 1 was able to achieve so far, whiel β is the lowest posible score that Player 2 was able to achieve so far. While considering a max-node, we can abort the search as soon as we find a move whose score is higher than β, and while considering a min-node, we can abort as soon as we find a move whose score is lower than α. The pseudo-code is as follows:

```
// visit a final node
final-visit(v):
    // simply return the final score for v
    return psi(v)


//visit a max-node
max-visit(v, alpha, beta):
    max = -infty
    for each child w of v do
        if w is a final configuration then
            child_score <- final-visit(w)
        else
            child_score <- min-visit(w, alpha, beta)
        if child-score > max then
            max <- child-score
            // if we have found a move that is better than
            // the best move so far, we update alpha
            if max > alpha then
                alpha <- max
        // if the move is better than the best move that
        // Player 2 can achieve so far, we abort
        if max > beta then
           break
    return max

//visit a min-node
min-visit(v, alpha, beta):
    min = infty
        if w is a final configuration then
            child_score <- final-visit(w)
        else
            child_score <- max-visit(w, alpha, beta)
        if child-score < min then
            min <- child-score
            if min < beta then
                beta <- min
        if min < alpha then
           break
    return min
```

If we use (α, β)-pruning, it does make a difference in which order the children w of a node v are evaluated. If we investigate the more promising moves first, it becomes likely that in later stagest we abort the search for a less favorable move. Thus, (α, β)-prunign is often combined with a heuristic that determines the order in which the children in a game tree are evaluated. In practice, we combine (α, β)-pruning with a bounded search depth, in order to make sure that the number of moves under investigation is not too large.

The techniques described so far represent the state of the art in the late 1990s and early 2000s. The pinnacle was reached in 19XX, when the chess computer DeepBlue, constructed by IBM, managed to win a tournament against a ruling chess champion, Wladimir Kramnik. DeppBlue had a special hardware to optimize the search of the game tree, and it used a variant of (α, β)-pruning that was powered by heuristics that IBM developed with several chess grandmasters. Furthermore, DeepBlue had large look-up tables with known game sequences, e.g., standard openings and endgames. The game was very close, but it was the first time that a computer had decisively beaten a human at chess. At the time, AI researchers were very satisfied with the success, but harder and less structured games like Go seemed to be completely out of reach for the current paradigm.

However, in 2016, a stunning reversal took place: AlphaGo, a computer go program developed by DeepMind, decisively beat the reigning Go-champion. Unlike with DeepBlue two decades earlier, the result was very clear. Furthermore, AlphaGo did not rely on special hardware and did not have extensive lookup-tables and libraries for known game sequences. Instead, AlphaGo relied on a very simple technique for evaluating the game tree, called Monte-Carlo-tree-search (**TODO: add more details**). The heuristics were not hardcoded, but obtained in an extensive training phase using *deep learning*. The victory of AlphaGo was one of the first impressive successes of a new paradigm in artificial intelligence the we still see today: instead of building intricate specialized models that try to capture the knowledge of human experts, we use abstract, general-purpose models that are trained using a massive amount of data. You will learn more about this development in later classes.

VI

Conclusion

KAPITEL 29

Conclusion and Outlook

This concludes our course on algorithms and data structures. In future classes, you will learn a lot more about these topics and about other aspects of theoretical computer science.