Bibliography

These lecture notes have been translated from the material of the Finnish course: OHJ-2010 Tietorakenteiden käyttö. The lecture notes have been edited based on the material written by professor Antti Valmari for the course Tietorakenteet ja algoritmit. Minna Ruuska has done most of the editorial work for the Finnish lecture material. The editorial choices made for the English version have been done by Terhi Kilamo.

Most algorithms are originally from the book Introduction to Algorithms; Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

In addition the following books have been used when completing this material:
- Introduction to The Design & Analysis of Algorithms; Anany Levitin
- Olioiden ohjelmointi C++:lla; Matti Rintala, Jyke Jokinen
- Tietorakenteet ja Algoritmit; Ilkka Kokkarinen, Kirsti Ala-Mutka

Goals of the course 26
Why? 20
Terminology and conventions 25
Goals of the course 25
Terminology 26
Insertion-sort 33
Implementing algorithms 35
Efficiency and algorithm design 41
Asymptotic notations 42
Brute force 49
Divide and conquer 52
Merge-sort 55
Merge 56
Asymptotic notations 63

Contents

The course in 2012 9
Learning outcome 10
Taking the course 11
Exercises 12
Homework Assignments 13
byTheMark 14
Grading 15
Course staff 16
Course announcements 17
Material 18
Introduction 19
Why? 20
Terminology and conventions 25
Goals of the course 25
Terminology 26
Insertion-sort 33
Implementing algorithms 35
Efficiency and algorithm design 41
Asymptotic notations 42
Brute force 49
Divide and conquer 52
Merge-sort 55
Merge 56
Asymptotic notations 63

Θ-notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
Ο-notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Properties of asymptotic notations 70
Concepts of efficiency 74
Units of the running-time 75
Units of memory use 76
Choosing an algorithm 78
BIN-SEARCH 83
Sorting algorithms 86
Sorting with a heap 87
Binary trees 87
Heap 89
Heapify 93
Build-Heap 94
Sorting with a heap 96
Heap-sort 96
Priority queue 100
Heap-Maximum 102
Randomized-max 103
Heap-insert 104
Quick-sort 106
Partition 108
Randomization 115
Randomized-partition 117
Randomized-Quick-sort 117
Randomized-Select 119

Asymptotic notations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
Θ-notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Properties of asymptotic notations 70
Concepts of efficiency 74
Units of the running-time 75
Units of memory use 76
Choosing an algorithm 78
BIN-SEARCH 83
Sorting algorithms 86
Sorting with a heap 87
Binary trees 87
Heap 89
Heapify 93
Build-Heap 94
Sorting with a heap 96
Heap-sort 96
Priority queue 100
Heap-Maximum 102
Randomized-max 103
Heap-insert 104
Quick-sort 106
Partition 108
Randomization 115
Randomized-partition 117
Randomized-Quick-sort 117
Randomized-Select 119
1 The course in 2012

Let’s start with the practicalities

The intended learning outcomes, lecture- and exercise times and places, grading criteria and the contact information of the course staff is listed in this chapter. You can also find the information about the course material here.

1.1 Learning outcome

The aim is to gain knowledge in the commonly used data structures and the algorithms available for using them. The goal is to also be able to

• analyze the asymptotic efficiency of a simple program or an algorithm (time and memory consumption)
• represent the efficiency with the appropriate notation
• choose a data structure most suitable for the problem at hand
• use the libraries available in programming languages sensibly

The course assumes that the students already have the needed basic programming skills from the earlier programming courses.

1.2 Taking the course

The course consists of

• Lectures (Thursdays 2h/week)
• Exercises (Mondays 2h/week)
• Homework assignments (3, all compulsory)
• byTheMark-exercises (60% compulsory)

The focus of the homework assignments is to practise the key topics of the course in practise.

1.3 Exercises

The weekly exercises are held on Mondays 14-16 (2-4 PM) in TC128. Starting on 10.9.

It is not necessary to sign up for the exercises.

Model answers are not published after the exercises.
1.4 Homework Assignments
There are three compulsory, individual, programming assignments on the course.
Assignments completed during the previous years are not valid on this course.
The deadline is on the given day at midnight.

<table>
<thead>
<tr>
<th>HW</th>
<th>Soft deadline</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>26.10.2012</td>
<td>02.11.2012</td>
</tr>
</tbody>
</table>

The deadlines are not flexible without a valid reason. Contact the lecturer or the TA before the deadline in case you fall ill or something else preventing you from working turns up.

1.5 byTheMark
byTheMark is an interactive online collection of algorithm simulation exercises.
The deadline for the exercises is 15.1.2013 and 60% of the points is required.
You get the most out of the exercises by doing them during the course.

1.6 Grading
The final grade of the course is determined based on the points of the final exam and the points received from the three homework assignments.
- the maximum amount of points in the exam is 30
- the homework assignments are graded 0-6 points
\[ \Rightarrow \text{the maximum is 36 points} \]
3 bonus points are available from active attendance to the weekly exercises. These points can be used to upgrade a passed grade by one. A failed course cannot be upgraded to passed with bonus points.

1.7 Course staff
Lecturer: Terhi Kilamo
- terhi.kilamo@tut.fi
- office: TF206
- phone: 040 849 0723
Terhi is responsible for the contents of the course, lectures, materials, grading and course compensation.
Teaching assistant: Seppo Koivisto
- seppo.koivisto@tut.fi
- office: TF212
- phone: 03 3115 2699
Seppo is responsible for the practical arrangements on the course and for the homework assignments.
In addition, Mikko Airaksinen is responsible for the weekly exercises.
1.8 Course announcements
Mostly static information about completing the course is available on the course webpage www.cs.tut.fi/~uda. The weekly exercise problems and the specifications of the homework assignments are published there.
All current news are published on the course news group tut.ot.tiraka, which should be followed
The course has an irc channel #tiraka for peer support and for contacting the staff for help
In order to contact the staff on course related issues, you should use the email address of the course tiraka@cs.tut.fi

1.9 Material
These lecture notes act as the primary material for the course and the exam is based on the contents of these.
There are many available books that cover the course topics. Recommendable volumes are:
- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms. MIT Press

2 Introduction
Let's talk first about the motivation for studying data structures and algorithms

2.1 Why?
There are no computer programs without algorithms
- Algorithms make for example the following applications possible:
algorithms are at work whenever a computer is used

Data structures are needed to store and access the data handled in the programs easily

- there are several different data structures and not all are suitable for all tasks
  ⇒ it is the programmer’s job to know which to choose
  ⇒ the behaviour, strengths, and weaknesses of the alternatives must be known

Modern programming languages provide easy to use library implementations for data structures. Understanding the properties of these and the limitations there may be for using them requires knowledge of basic data structures.

Ever gotten frustrated on a program running slowly?

- functionality is naturally a top priority but efficiency and thus the usability and user experience are not meaningless side remarks
- it is important to take memory- and time consumption into account when making decisions in program implementation
- using a library implementation seems more straightforward than is really is

This course discusses these issues
3 Terminology and conventions
This chapter covers the terminology and the syntax of the algorithms used on the course.

The differences between algorithms represented in pseudocode and the actual solution in a programming language is discussed. The sorting algorithm INSERTION-SORT is used as an example.

3.1 Goals of the course
As discussed earlier, the main goal of the course is to provide a sufficient knowledge on and the basic tools for choosing the most suitable solution to a given programming problem. The course also aims to give the student the ability to evaluate the decisions made during the design process on a basic level.

- The course concentrates on choosing a suitable data structure for solving a given problem.
- In addition, common types of problems and the algorithms to solve them are covered.

3.2 Terminology
The data structures and algorithms commonly used in programming are covered on this course.

A data structure is a collection of related data items stored in a segment of the computer’s memory:

- data can be added and searched by using suitable algorithms.
- there can be several different levels in a data structure: a data structure can consist of other data structures.

An algorithm is a well defined set of instructions that takes in a set of input and produces a set of output, i.e. it gives a solution to a given problem.
An algorithm solves a well defined problem.

- The relation between the results and the given input is determined by the problem
- For example: sorting the contents of the array

**Input:** A sequence of numbers \( a_1, a_2, \ldots, a_n \)

**Results:** Numbers \( a_1, a_2, \ldots, a_n \) sorted into an ascending order

- Usually algorithms are implemented as computer programs or in hardware
- In practice, the implementation must take several engineering viewpoints into account
  - Accommodation to the situation and environment
  - Checking the legality of inputs
  - Handling error situations
  - Limitations of the programming language
  - Speed limitations and practicality issues concerning the hardware and the programming language
  - Maintenance issues ⇒ Modularity etc.
  ⇒ The idea of the algorithm may get lost under the implementation details

On this course we concentrate on the algorithmic ideas and therefore usually represent the algorithms in pseudocode without legality checks, error handling etc.

Let’s take, for example, an algorithm suitable for sorting small arrays called **INSERTION-SORT**:

### An Example

<table>
<thead>
<tr>
<th>31</th>
<th>41</th>
<th>59</th>
<th>26</th>
<th>41</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>31</td>
<td>59</td>
<td>26</td>
<td>41</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>31</td>
<td>41</td>
<td>26</td>
<td>41</td>
<td>58</td>
</tr>
<tr>
<td>58</td>
<td>31</td>
<td>41</td>
<td>26</td>
<td>41</td>
<td>58</td>
</tr>
</tbody>
</table>

- An instance of the problem is created by giving legal values to the elements of the problem’s input
  - For example: an instance of the sorting problem: 31, 41, 59, 26, 41, 58

An algorithm is correct, if it halts and gives the correct output as the result each time it is given a legal input.

- A certain set of formally possible inputs can be forbidden by the definition of the algorithm or the problem
- An algorithm can be incorrect in three different ways:
  - It produces an incorrect result
  - It crashes during execution
  - It never halts, i.e. has infinite execution
- An incorrect algorithm may sometimes be very useful one as long as a certain amount of errors is tolerated.
  - For example, checking whether a number is prime

In principle any method of representing algorithms can be used as long as the result is precise and unambiguous

### The Basic Idea

- During execution the leftmost elements in the array are sorted and the rest are still unsorted
- The algorithm starts from the second element and iteratively steps through the elements up to the end of the array
- On each step the algorithm searches for the point in the sorted part of the array, where the first element in the unsorted range should go to.
- Room is made for the new element by moving the larger elements one step to the right
- The element is placed to its correct position and the size of the sorted range in the beginning of the array is incremented by one.
In pseudocode used on the course **INSERTION-SORT** looks like this:

```
INSERTION-SORT(A)  (input in array A)
1 for j := 2 to A.length do  (increment the size of the sorted range)
2   key := A[j]  (handle the first unsorted element)
3   i := j - 1
4   while i > 0 and A[i] > key do  (find the correct location for the new element)
5      A[i + 1] := A[i]  (make room for the new element)
6      i := i - 1
7   A[i + 1] := key  (set the new element to its correct location)
```

- indentation is used to indicate the range of conditions and loop structures
- *(comments)* are written in parentheses in italics
- the " := " is used as the assignment operator (" = " is the comparison operator)
- the lines starting with the character > give textual instructions

### 3.3 Implementing algorithms

In the real world you need to be able to use theoretical knowledge in practise.

For example: apply a given sorting algorithm in a certain programming problem

- numbers are rarely sorted alone, we sort structures with
  - a **key**
  - **satellite data**
- the **key** sets the order
  - it is used in the comparisons
- the **satellite data** is not used in the comparison, but it must be moved around together with the key

- members of structure elements (or objects) are referred to with the dot notation.
  - e.g. student.name, student.number
- the members of a structure accessed through a pointer \( x \) are referred to with the \( \rightarrow \) character
  - e.g. \( x \rightarrow \text{name} \), \( x \rightarrow \text{number} \)
- variables are local unless mentioned otherwise
- a collection of elements, an array or a pointer, is a **reference** to the collection
  - larger data structures like the ones mentioned should always be passed by reference
- a pass-by-value mechanism is used for single parameters (just like C++ does)
- a pointer or a reference can also have no target: \( \text{NIL} \)

The **INSERTION-SORT** algorithm from the previous chapter would change as follows if there were some satellite data used:

```
for j := 2 to A.length do
   temp := A[j]
   i := j - 1
   while i > 0 and A[i].key > temp.key do
      A[i + 1] := A[i]
      i := i - 1
   A[i + 1] := temp
```

- An array of pointers to structures should be used with a lot of satellite data. The sorting is done with the pointers and the structures can then be moved directly to their correct locations.
In order to make an executable program, additional information is needed to implement **INSERTION-SORT**

- an actual programming language must be used with its syntax for defining variables and functions
- a main program that takes care of reading the input, checking its legality and printing the results is also needed
  - it is common that the main is longer than the actual algorithm

The implementation of the program described above in C++:

```cpp
#include <iostream>
#include <vector>
typedef std::vector<int> Array;

void insertionSort(Array & A ) {
    int key, i; unsigned int j;
    for( j = 1; j < A.size(); ++j ) {
        key = A.at(j); i = j-1;
        while( i >= 0 && A.at(i) > key ) {
            A.at(i+1) = A.at(i); --i;
        }
        A.at(i+1) = key;
    }
}

int main() {
    unsigned int i;
    // getting the amount of elements
    std::cout << "Give the size of the array 0...: "; std::cin >> i;
    Array A(i); // creating the array
    // reading in the elements
    for( i = 0; i < A.size(); ++i ) {
        std::cout << "Give A[" << i+1 << "]: ";
        std::cin >> A.at(i);
    }
    insertionSort( A ); // sorting
    // print nicely
    for( i = 0; i < A.size(); ++i ) {
        if( i % 5 == 0 ) {
            std::cout << std::endl;
        } else {
            std::cout << " ";
        }
        std::cout << A.at(i);
    }
    std::cout << std::endl;
}
```

The program code is significantly longer than the pseudocode. It is also more difficult to see the central characteristics of the algorithm.

This course concentrates on the principles of algorithms and data structures. Therefore using program code doesn’t serve the goals of the course.

⇒ From now on, program code implementations are not normally shown.
4 Efficiency and algorithm design

This chapter handles the analysis of algorithms. The efficiency of algorithms is discussed and the notations used to describe the asymptotic behavior of an algorithm are introduced. The asymptotic behavior describes the change in the algorithm’s use of the computer’s resources when the size of the input increases.

In addition the chapter introduces two algorithm design techniques: brute force and divide and conquer.

4.1 Asymptotic notations

Sometimes it is important to know the exact time it takes to perform a certain operation (in real-time systems for example). Often it is enough to know how the running time of the algorithm changes as the input gets larger.

• The advantage is that the calculations are not tied to a given processor, architecture or a programming language.
• In fact we will later learn that the analysis is not tied to programming at all but can be used to describe the efficiency of any behaviour that consists of successive operations.
• The time efficiency analysis is simplified by assuming that all operations that are independent of the size of the input take the same amount of time to execute.
• Furthermore, we don’t care how many times a certain operation is done as long as the amount is constant.

We investigate how many times each row is executed during the execution of the algorithm and add the results together.

The result is further simplified by removing any constant coefficients and lower-order terms.

⇒ This can be done since as the input gets large enough the lower-order terms get insignificant when compared to the leading term.

⇒ The approach naturally doesn’t produce reliable results with small inputs. However, when the inputs are small, programs usually are efficient enough in any case.

The final result is the efficiency of the algorithm and is denoted it with the greek alphabet theta, \( \Theta \).

Let’s take a few examples:

• example: addition of the elements in an array

```plaintext
1 for i := 1 to A.length do
2     sum := sum + A[i]
```

– if the size of the array \( A \) is \( n \), line 1 is executed \( n + 1 \) times
– line 2 is executed \( n \) times
– the running time increases as \( n \) gets larger:

<table>
<thead>
<tr>
<th>( n )</th>
<th>time = 2n + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
</tr>
<tr>
<td>100</td>
<td>201</td>
</tr>
<tr>
<td>1000</td>
<td>2001</td>
</tr>
<tr>
<td>10000</td>
<td>20001</td>
</tr>
</tbody>
</table>

⇒ notice how the value of \( n \) dominates the running time
⇒ let’s simplify the result as described earlier by taking away the constant coefficients and the lower-order terms.
we get \( \Theta(n) \) as the result, i.e. the running time depends linearly on the size of the input.

- example: searching from an unsorted array

```java
for i := 1 to A.length do
    if A[i] = x then
        return i
```

- the running time depends now not only on the size of the input but also on the order of the elements, i.e. the location of the searched element in the array.
- we must separately handle the best-case, worst-case and average-case efficiencies.
- in the best case the element we’re searching for is the first element in the array.
  \( \Rightarrow \) the element is found in constant time, i.e. the efficiency is \( \Theta(1) \)
- in the worst case the element is the last element in the array or there are no matching elements.
- now line 1 gets executed \( n + 1 \) times and line 2 \( n \) times
  \( \Rightarrow \) efficiency is \( \Theta(n) \).

- determining the average-case efficiency is not as straightforward
- first we must make some assumptions on the average, typical inputs:
  * the probability \( p \) that the element is in the array is \( 0 \leq p \leq 1 \)
  * the probability of finding the first match in each position in the array is the same
- we can find out the average amount of comparisons by using the probabilities
  * the probability that the element is not found is \( 1 - p \), and we must make \( n \) comparisons
  * the probability for the first match occuring at the index \( i \), is \( p/n \), and the amount of comparisons needed is \( i \)
  * the number of comparisons is:

\[
[1 \cdot \frac{p}{n} + 2 \cdot \frac{2}{n} + \ldots + i \cdot \frac{i}{n} + \ldots + n \cdot \frac{n}{n}] + n \cdot (1 - p)
\]

- if we assume that the element is found in the array, i.e. \( p = 1 \), we get \( (n+1)/2 \) which is \( \Theta(n) \)
  \( \Rightarrow \) since also the case where the element is not found in the array has linear efficiency we can be quite confident that the average efficiency is \( \Theta(n) \)
- it is important to keep in mind that all inputs are usually not as probable. Therefore, each case needs to be investigated separately.

- example: finding the common element in two arrays

```java
for i := 1 to A.length do
    for j := 1 to B.length do
        if A[i] = B[j] then
            return A[i]
```

- line 1 is executed \( 1 - (n + 1) \) times
- line 2 is executed \( 1 - (n \cdot (n + 1)) \) times
- line 3 is executed \( 1 - (n \cdot n) \) times
- line 4 is executed at most once
- the algorithm is fastest when the first element of both arrays is the same
  \( \Rightarrow \) the best case efficiency is \( \Theta(1) \)
- in the worst case there are no common elements in the arrays or the last elements are the same
  \( \Rightarrow \) the efficiency is \( 2n^2 + 2n + 1 = \Theta(n^2) \)
- on average we can assume that both arrays need to be investigated approximately half way through.
  \( \Rightarrow \) the efficiency is \( \Theta(n^2) \) (or \( \Theta(nm) \) if the arrays are of different lengths)
4.2 Brute force

The most straightforward algorithm design technique covered on the course is brute force.

- initially the entire input is unprocessed
- the algorithm processes a small piece of the input on each round so that the amount of processed data gets larger and the amount of unprocessed data gets smaller
- finally there is no unprocessed data and the algorithm halts

These types of algorithms are easy to implement and work efficiently on small inputs.

The Insertion-Sort seen earlier is a “brute force” algorithm.

- initially the entire array is (possibly) unsorted
- on each round the size of the sorted range in the beginning of the array increases by one element
- in the end the entire array is sorted

- in the best case the entire array is already sorted and the running time of the entire algorithm is at least \( \Theta(n) \)
- in the worst case the array is in a reversed order. \( \Theta(n^2) \) time is used
- once again determining the average case is more difficult:
- let’s assume that out of randomly selected element pairs half is in an incorrect order in the array
  ⇒ the amount of comparisons needed is half the amount of the worst case where all the element pairs were in an incorrect order
  ⇒ the average-case running time is the worst-case running time divided by two: \( (n - 1)n / 4 = \Theta(n^2) \)

4.3 Divide and conquer

We’ve earlier seen the brute force algorithm design technique and the algorithm INSERTION-SORT as an example of it.

Now another technique called divide and conquer is introduced. It is often more efficient than the brute force approach.

- the problem is divided into several subproblems that are like the original but smaller in size.
- small subproblems are solved straightforwardly
- larger subproblems are further divided into smaller units
- finally the solutions of the subproblems are combined to get the solution to the original problem

Let’s get back to the claim made earlier about the complexity notation not being fixed to programs and take an everyday, concrete example
Example: finding the false goldcoin

- The problem is well-known from logic problems.
- We have \( n \) gold coins, one of which is false. The false coin looks the same as the real ones but is lighter than the others. We have a scale we can use and our task is to find the false coin.
- We can solve the problem with Brute Force by choosing a random coin and by comparing it to the other coins one at a time.
- At least 1 and at most \( n - 1 \) weighings are needed. The best-case efficiency is \( \Theta(1) \) and the worst and average case efficiencies are \( \Theta(n) \).
- Alternatively we can always take two coins at random and weigh them. At most \( n/2 \) weighings are needed and the efficiency of the solution is still the same.

The same problem can be solved more efficiently with divide and conquer:

- Divide the coins into the two pans on the scales. The coins on the heavier side are all authentic, so they don’t need to be investigated further.
- Continue the search similarly with the lighter half, i.e. the half that contains the false coin, until there is only one coin in the pan, the coin that we know is false.
- The solution is recursive and the base case is the situation where there is only one possible coin that can be false.
- The amount of coins on each weighing is 2 to the power of the amount of weighings still required: on the highest level there are \( 2^0 = 1 \) coins, so based on the definition of the logarithm:

\[
2^\text{weighings} = n \Rightarrow \log_2 n = \text{weighings}
\]

- Only \( \log_2 n \) weighings is needed, which is significantly fewer than \( n/2 \) when the amount of coins is large.
- The complexity of the solution is \( \Theta(\log n) \) both in the best and the worst-case.

Sorting elements can also be done with divide and conquer. Let’s take the recursive sorting algorithm \textsc{Merge-Sort}, for example:

- divide the elements in the array into half.
- continue dividing the parts into half until the subarrays contain atmost one element
- arrays of 0 or 1 length are already sorted and require no actions
- finally merge the sorted subarrays

\[
\begin{array}{cccccccc}
8 & 1 & 6 & 3 & 6 & 5 \\
8 & 1 & 6 & 3 & 6 & 5 \\
8 & 1 & 6 & 3 & 6 & 5 \\
8 & 1 & 6 & 3 & 6 & 5 \\
1 & 8 & 6 & 3 & 6 & 5 \\
1 & 8 & 6 & 3 & 6 & 5 \\
1 & 3 & 5 & 6 & 6 & 8 \\
\end{array}
\]

- the \textsc{Merge} algorithm for merging the subarrays:

\[
\textsc{Merge}(A, p, q, r) \\
1 \textbf{for } i := p \textbf{ to } do \\
2 \text{\text{\textit{B}[i] := A[i]}} \\
3 i := p \\
4 j := p + 1 \\
5 \textbf{while } j \le q \textbf{ and } k \le r \textbf{ do} \\
6 \text{\textbf{if } \text{\textit{B}[j] \le B[k] \textbf{ then}} \\
7 \text{\text{\textit{A}[i] := B[j]}} \\
8 i := i + 1 \\
9 \text{\textbf{else}} \\
10 \text{\text{\textit{A}[i] := B[k]}} \\
11 k := k + 1 \\
12 i := i + 1 \\
13 \textbf{if } j > q \textbf{ then} \\
14 k := 0 \\
15 \text{\textbf{else}} \\
16 k := q - r \\
17 \textbf{for } j := i \textbf{ to } r \textbf{ do} \\
18 \text{\textit{A}[i] := B[j + k]}
\]

\[
\begin{array}{cccccccc}
8 & 1 & 6 & 3 & 6 & 5 \\
8 & 1 & 6 & 3 & 6 & 5 \\
8 & 1 & 6 & 3 & 6 & 5 \\
8 & 1 & 6 & 3 & 6 & 5 \\
1 & 8 & 6 & 3 & 6 & 5 \\
1 & 8 & 6 & 3 & 6 & 5 \\
1 & 3 & 5 & 6 & 6 & 8 \\
\end{array}
\]
• **Merge-Sort**

\[
\text{Merge-Sort}(A, p, r) \\
\text{1 if } p < r \text{ then} \\
\text{2 } q := \lfloor (p + r) / 2 \rfloor \\
\text{3 Merge-Sort}(A, p, q) \\
\text{4 Merge-Sort}(A, q + 1, r) \\
\text{5 Merge}(A, p, q, r)
\]

• Merge-Sort merges the arrays by using the “brute force” method.
  - the first for-loop uses a linear amount of time \( \Theta(n) \) relative to the size of the subarray
  - the while-loop scans through the upper and the lower halves at most once and at least one of the halves entirely \( \Rightarrow \Theta(n) \)
  - the second for-loop scans through at most half of the array and its running time is \( \Theta(n) \) in the worst case.
  - other operations are constant time

\[
\Rightarrow \text{we can concentrate on the time used by the instances of Merge, everything else is constant time.}
\]

\[
\begin{array}{cccccccc}
\text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} \\
2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\text{log}_2 \text{n} \\
\text{n}
\]

\[
\Rightarrow \text{the running time of the entire algorithm is obtained by combining the results. It’s } \Theta(n) \text{ both in the best and the worst case.}
\]

• The analysis of Merge-Sort is not as straightforward since it is recursive. Therefore the equation for its running time would also be recursive.

• Finding the recursive equation is, however, beyond the goals of this course so we’ll settle for a less formal approach

• Merge-Sort calls itself and Merge, all other operations are constant time

\[
\Rightarrow \text{the amount increases in powers of two, so the amount of instances on the last level is } 2^h, \text{ where } h \text{ is the height of the tree.}
\]

\[
\Rightarrow \text{on the last level there are approximately } n \text{ instances} \\
\Rightarrow 2^h = n \Leftrightarrow \text{log}_2 n = h, \text{ the height of the tree is } \text{log}_2 n
\]

\[
\Rightarrow \text{since a linear amount of work is done on each level and there are } \text{log}_2 n \text{ levels, the running time of the entire algorithm is } \Theta(n \text{ log}_2 n)
\]
**Merge-Sort** is clearly more complicated than **Insertion-Sort**. Is using it really worth the effort?

Yes, on large inputs the difference is clear.

- if \( n = 1000000 \) \( n^2 = 100000000000 \), while \( n \log n \) is about 19930000

**Practical examples:**

- **Merge-Sort** and **Insertion-Sort** have been implemented with C++ and their running times have been measured on a Solaris-workstation January 2001:
  - on very small inputs **Insertion-Sort** is more efficient
  - when \( n = 100 \), the programs are equally fast
  - when \( n = 1000 \), **Insertion-Sort** is already 10 times slower

**Merge-Sort** is a good algorithm based on the definition given in chapter 1.1.

### 4.4 Asymptotic notations

The \( \Theta \)-notation is defined in this chapter together with two other related notations.

The equation for the running time was simplified significantly:

- only the highest order term was used
- its constant coefficient was left out

⇒ studying the behaviour of the algorithm as the size of its input increases to infinity

- i.e. studying the asymptotic efficiency of algorithms

⇒ giving useful information only with inputs larger than a certain limit

- often the limit is rather low

⇒ the algorithm fastest according to \( \Theta \) and other notations is the fastest also in practice except on very small inputs

- **Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein** gives an example on the differences in the efficiency of **Insertion-Sort** and **Merge-Sort**:
  - **Insertion-Sort** is run on a computer 100 times faster than the computer running **Merge-Sort**
  - a top-notch programmer implements **Insertion-Sort** directly in machine code leaving the constant coefficient small, \( 2n^2 \) time is used
  - a mediocre programmer implements **Merge-Sort** in a high-level programming language and \( 50n \log n \) is used

⇒ sorting one million numbers takes 2000 seconds with **Insertion-Sort** and about 100 seconds with **Merge-Sort**

- If the amount of numbers that need to be sorted is increased to 10 million, the running time of **Insertion-Sort** increases to approximate 2.3 days while **Merge-Sort** produces results in under 20 minutes

**Merge-Sort** is a good algorithm based on the definition given in chapter 1.1.

#### \( \Theta \)-notation

- let \( g(n) \) be a function from a set of numbers to a set of numbers
  - \( \Theta(g(n)) \) is the set of those functions \( f(n) \) for which there exists positive constants \( c_1, c_2 \) and \( n_0 \) so that for all \( n \geq n_0 \)
    \[
    0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
    \]

- the function in the picture (c) \( f(x) = \Theta(g(n)) \)

- \( \Theta(g(n)) \) is a set of functions

⇒ we should write \( f(n) \in \Theta(g(n)) \), but this is not usually used

- let \( g(n) \) be a function from a set of numbers to a set of numbers
  - \( \Theta(g(n)) \) is the set of those functions \( f(n) \) for which there exists positive constants \( c_1, c_2 \) and \( n_0 \) so that for all \( n \geq n_0 \)
    \[
    0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
    \]

- the function in the picture (c) \( f(x) = \Theta(g(n)) \)

⇒ we should write \( f(n) \in \Theta(g(n)) \), but this is not usually used

- let \( g(n) \) be a function from a set of numbers to a set of numbers
  - \( \Theta(g(n)) \) is the set of those functions \( f(n) \) for which there exists positive constants \( c_1, c_2 \) and \( n_0 \) so that for all \( n \geq n_0 \)
    \[
    0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
    \]

- the function in the picture (c) \( f(x) = \Theta(g(n)) \)

⇒ we should write \( f(n) \in \Theta(g(n)) \), but this is not usually used

- let \( g(n) \) be a function from a set of numbers to a set of numbers
  - \( \Theta(g(n)) \) is the set of those functions \( f(n) \) for which there exists positive constants \( c_1, c_2 \) and \( n_0 \) so that for all \( n \geq n_0 \)
    \[
    0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
    \]

- the function in the picture (c) \( f(x) = \Theta(g(n)) \)

⇒ we should write \( f(n) \in \Theta(g(n)) \), but this is not usually used
Whether a function $f(n)$ is $\Theta(g(n))$ can be proven by finding suitable values for the constants $c_1$, $c_2$ and $n_0$ and by showing that the function stays larger or equal to $c_1 g(n)$ and smaller or equal to $c_2 g(n)$ with values of $n$ starting from $n_0$.

For example: $3n^2 + 5n - 20 = \Theta(n^2)$

- let's choose $c_1 = 3$, $c_2 = 4$ and $n_0 = 4$
- $0 \leq 3n^2 \leq 3n^2 + 5n - 20 \leq 4n^2$ when $n \geq 4$, since $0 \leq 5n - 20 \leq n^2$
- just as well we could have chosen $c_1 = 2$, $c_2 = 6$ and $n_0 = 7$ or $c_1 = 0,000,1$, $c_2 = 1,000$ and $n_0 = 1,000$
- what counts is being able to choose some positive $c_1$, $c_2$ and $n_0$ that fulfill the requirements

An important result: if $a_k > 0$, then

$$a_kn^k + a_{k-1}n^{k-1} + \ldots + a_1n + a_0 = \Theta(n^k)$$

- in other words, if the coefficient of the highest-order term of a polynomial is positive the $\Theta$-notation allows us to ignore all lower-order terms and the coefficient.

- an important result: if $k \leq m$, then $n^k = O(n^m)$
- if the running time of the slowest case is $O(g(n))$, then the running time of every case is $O(g(n))$

The $O$-notation is important in practise as, for example, the running times guaranteed by the C++-standard are often given in it.

Often some upper bound can be given to the running time of the algorithm in the slowest possible case with the $O$-notation (and every case at the same time).

The following holds for constant functions $c = \Theta(n^0) = \Theta(1)$

- $\Theta(1)$ doesn’t indicate which variable is used in the analysis
  - it can only be used if the variable is clear from the context

$O$-notation (pronounced “big-oh”)

The $O$-notation is otherwise like the $\Theta$-notation, but it bounds the function only from above.

$\Rightarrow$ asymptotic upper bound

Definition:

- $O(g(n))$ is the set of functions $f(n)$ for which there exists positive constants $c$ and $n_0$, such that, for all $n \geq n_0$ $0 \leq f(n) \leq c \cdot g(n)$

- the function in the picture (a) $f(x) = O(g(n))$
- it holds: if $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$
- the opposite doesn’t always hold: $n^2 = O(n^3)$, but $n^2 \neq \Theta(n^3)$

For example: INSERTION-SORT

```
for j := 2 to A.length do
  key := A[j]
  i := j - 1
  while i > 0 and A[i] > key do
    A[i + 1] := A[i]
    i := i - 1
  A[i + 1] := key
```

The worst case running time is $O(n) \cdot O(n) \cdot O(1) = O(n^2)$
**Ω-notation** (pronounced "big-omega")

The Ω-notation is otherwise like the Θ-notation but is bounds the function only from below. 
\[ \Rightarrow \text{asymptotic lower bound} \]

**Definition:**
\[ \Omega(g(n)) \] is the set of functions \( f(n) \) for which there exist positive constants \( c \) and \( n_0 \) such that, for all \( n \geq n_0 \)
\[ 0 \leq c \cdot g(n) \leq f(n) \]

- the function in the picture (b) function \( f(x) = \Omega(g(n)) \)
- the following result follows from the definitions: \( f(n) = \Theta(g(n)) \) if and only if \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).
- if the running time of the fastest case is \( \Omega(g(n)) \), the running time of every case is \( \Omega(g(n)) \)

The Ω-notation is useful in practise mostly in situations where even the best-case efficiency of the solution is unsatisfactory and the result can be rejected straightforward.

---

An example simplifying things a little:

- If an algorithm is \( \Omega(g(n)) \), its consumption of resources is at least \( g(n) \).
  - cmp. a book costs at least about 10 euros.
- If an algorithm is \( O(g(n)) \), its consumption of resources is at most \( g(n) \).
  - cmp. a book costs at most 10 euros.
- If an algorithm is \( \Theta(g(n)) \), its consumption of resources is always \( g(n) \).
  - cmp. a book costs about 10 euros

Note that the running time of all algorithms cannot be determined with the Θ-notation.

For example Insertion-Sort:

- the best-case is \( \Omega(n) \), but not \( \Omega(n^2) \)
- the worst-case is \( O(n^2) \), but not \( O(n) \)
\[ \Rightarrow \text{a } \Theta\text{-value common to all cases cannot be determined.} \]

---

**Properties of asymptotic notations**

\( f(n) = \Omega(g(n)) \) and \( f(n) = O(g(n)) \) \iff \( f(n) = \Theta(g(n)) \)

Many of the relational properties of real numbers apply to asymptotic notations:

\[ f(n) = O(g(n)) \quad a \leq b \]
\[ f(n) = \Theta(g(n)) \quad a = b \]
\[ f(n) = \Omega(g(n)) \quad a \geq b \]

i.e. if the highest-order term of \( f(n) \) whose constant coefficient has been removed \( \leq g(n) \)’s corresponding term, \( f(n) = O(g(n)) \) etc.

Note the difference: for any two real numbers exactly one of the following must hold:
\[ a < b, a = b \text{ ja } a > b. \]
However, this does not hold for all asymptotic notations.
\[ \Rightarrow \text{Not all functions are asymptotically comparable to each other (e.g. } n \text{ and } n^{1.001}. \]

---

**An example**

Let’s take a function \( f(n) = 3n^2 + 5n + 2 \).

- lower-order terms ignored
- constant coefficients ignored
\[ \Rightarrow f(n) = \Theta(n^2) \]

To be completely convinced we’ll determine the constants \( c_1 \) ja \( c_2 \): \( 3n^2 \leq 3n^2 + 5n + 2 \leq 4n^2 \), when \( n \geq 6 \)
\[ \Rightarrow c_1 = 3, \ c_2 = 4 \text{ and } n_0 = 6 \text{ work correctly} \]
\[ \Rightarrow f(n) = O(n^2) \text{ and } \Omega(n^2) \]
\[ \Rightarrow f(n) = \Theta(n^2) \]
Clearly the constant \( c_2 = 4 \) works also when \( g(n) = n^2 \), since when \( n \geq 6, n^2 > n^2 \) \[ \Rightarrow f(n) = O(n^2) \]

- the same holds when \( g(n) = n^2 \)...

And below the constant \( c_1 = 3 \) works also when \( g(n) = n \lg n \), since when \( n \geq 6, n^2 > n \lg n \) \[ \Rightarrow f(n) = \Omega(n \lg n) \]

- the same holds when \( g(n) = n \) or \( g(n) = \lg n \)

### 4.5 Concepts of efficiency

So far the efficiency of algorithms has been studied from the point of view of the running-time. However, there are other alternatives:

- We can measure the memory consumption or used bandwidth

In addition, at least the following need to be taken into account in practice:

- The unit used to measure the consumption of resources
- The definition of the size of the input
- Whether we’re measuring the best-case, worst-case or average-case resource consumption
- What kind of input sizes come in question
- Are the asymptotic notations precise enough or do we need more detailed information

#### Units of the running-time

When choosing the unit of the running-time, the “step”, usually a solution as independent of the machine architecture as possible is aimed for.

- Real units like a second cannot be used
- The constant coefficients become unimportant
  \[ \Rightarrow \text{We’re left with the precision of the asymptotic notations.} \]
- Any constant time operation can be used as the step
- Any operation whose time consumption is independent of the size of the input can be seen as constant time.
- This means that the execution time of the operation never exceeds a given time, independent of the size of the input
- Assignment of a single variable, testing the condition of a \texttt{if}-statement etc. are all examples of a single step.
- There is no need to be too precise with defining the step as \( \Theta(1) + \Theta(1) = \Theta(1) \).

#### Units of memory use

A bit, a byte (8 bits) and a word (if its length is known) are almost always exact units.

The memory use of different types is often known, although it varies a little between different computers and languages.

- an integer is usually 16 or 32 bits
- a character is usually 1 byte = 8 bits
- a pointer is usually 4 bytes = 32 bits
- an array \( A[1 \ldots n] \) is often \( n \cdot \text{<element size>} \)

\[ \Rightarrow \text{Estimating the exact memory use is often possible, although requires precision.} \]

Asymptotic notations are useful when calculating the exact number of bytes is not worth the effort.
If the algorithm stores the entire input simultaneously, it makes sense to separate the amount of memory used for the input from the rest of the memory consumption.

For example:
- **INSERTION-SORT** requires $\Theta(1)$ extra memory
- **MERGE-SORT** requires $\Theta(n)$ extra memory
  $\Rightarrow$ INSERTION-SORT is significantly better in its memory consumption than MERGE-SORT
- the total memory needed is $\Theta(n)$ in both cases
  $\Rightarrow$ if we measure the total memory needed INSERTION-SORT isn’t any better than MERGE-SORT when using the $\Theta$-notation
- It’s worth pointing out that searching for a character string from an input file doesn’t store the entire input but scans through it.

### 4.6 Choosing an algorithm

The most important factor in choosing an algorithm is usually its **efficiency in that situation**. However, there are other factors:

- implementation is easy
  - is there a suitable algorithm already available?
  - is the advantage of improved efficiency worth the effort of implementing a more complex algorithm?
  - simpler code may not contain errors as easily
  - a simpler solution is easier to maintain
- precision of the results
  - with real numbers round-off errors can be a significant problem
- variation in the running-time
  - e.g. in signal processing the running-time must not vary at all

The programming environment also sets its limitations:

- many languages require that a maximum size is defined for arrays
  $\Rightarrow$ algorithms using arrays get a compile time, artificial upper limit
  - with list structures the algorithm works as long as there is memory available in the computer
- the memory can suddenly run out with lists, but not with arrays of a fixed size
  $\Rightarrow$ list structures are not always suitable for embedded systems
- in some computers the space reserved for recursion is much smaller than the space for the rest of the data
  $\Rightarrow$ if a lot of memory is needed, a non-recursive algorithm (or implementation) must be chosen

If the efficiency is the primary factor in the selection, at least the following should be taken into account:

- Is the size of the input data so large that the asymptotic efficiency gives the right idea about the efficiency?
- Can the worst-case be slow if the efficiency is good in the average case?
- Is memory use a factor?
- Is there a certain regularity in the input that can be advantageous?
  - with one execution?
  - with several executions?
- Is there a certain regularity in the input that often is the worst-case for some algorithm?
Example: phonebook

- operations
  - finding the phonenumber based on the name
  - adding a new name to the phonebook
  - removing an old name and number

- assumptions
  - additions and removals are needed rarely
  - phonenumber queries are done often
  - additions and removals are done in groups

1st attempt: unsorted array

<table>
<thead>
<tr>
<th>Virtanen</th>
<th>Järvinen</th>
<th>Lahtinen</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 555</td>
<td>123 111</td>
<td>123 444</td>
</tr>
</tbody>
</table>

- Adding a new name to the end: $O(1)$.
- Searching by scanning the elements from the beginning (or end): $O(n)$.
- Removing by moving the last element to the place of the removed element: $O(1) + \text{search costs} = O(n)$.

⇒ The solution is not suitable since the operations that are needed often are slow.

2nd attempt: sorted array, 1st version

- Adding the new names directly to their correct location in the alphabetical order. The rest of the elements are moved one step forwards: $O(n)$.
- Removing by moving the elements one step backwards: $O(n)$.
- Searching with binary search:

```
BIN-SEARCH(A, 1, n, key)
1 low := 1; hi := n
2 while low < hi do
3     mid := \lfloor (low + hi)/2 \rfloor
4     if key ≤ A[mid] then
5         hi := mid
6     else
7         low := mid + 1
8     end if
9     if A[low] = key then
10        return low
11     else
12        return 0
13 end if

> requirement: n ≥ 1, the array is sorted
(initialize the search to cover the entire array)
(search until there are no more elements to cover)
(divide the search area in half)
(If the key is in the bottom half...)
(choose the bottom half as the new search area)
(else...)
(choose the upper half as the search area)
(the element is found)
(the element is not found)
```

- The running time of BIN-SEARCH is $Θ(\lg n)$.
- Now the search is efficient but the removal and addition are still slow.
- The solution seems better than the first attempt if our original assumption is correct and searches are done more frequently than additions and removals.

3rd attempt: an almost sorted array

- Keep most of the array sorted and a small unsorted segment at the end of the array (size 1).
  - cmp. Phonebook + additional pages
- Additions are done to the segment at the end: $O(1)$.
- Search is done first with binary search in the sorted part and then if needed by scanning the unsorted part at the end: $O(\lg n)$.
- Removal is done by leaving the name in and changing the number to 0: $O(1) + \text{search costs} = O(\lg n) + O(1)$.
- When the unsorted end segment has become too large, sort the entire array: $Θ(n \lg n)$.
- A mediocre solution, but
  - $Θ(1)$ can be large
  - sorting every now and then costs time
4th attempt: a sorted array, 2nd. version

- Let’s use the information that additions and removal are done in groups to our advantage.
- Sort the groups of additions and removals.
- Merge the array and the group of additions (like with \textsc{merge}) by removing the elements in the group of removals simultaneously.
- Now the search is still logarithmic.
- Addition and removal uses $O(l \lg l) + O(p \lg p) + O(n)$, when $l$ is the amount of additions and $p$ is the amount of removals
- Pretty good!

The problem could also be solved with dynamic sets covered later on the course or with the containers in the C++ standard library.

5 Sorting algorithms

This chapter covers two efficient sorting algorithms that sort the data in place.

In addition their central ideas are applied to solving two separate problems - priority queue and finding the median.

Finally the maximum efficiency of comparison sorts, i.e. sorting based on the comparisons of the elements, is discussed.

Sorting algorithms that use other approaches than comparisons are also examined.

5.1 Sorting with a heap

This chapter introduces a sorting algorithm \textsc{heapsort} that uses a very important data structure, a heap, to manage data during execution.

Binary trees

Before we get our hands on the heap, let’s define what a binary tree is

- a structure that consists of nodes who each have 0, 1 or 2 children
- the children are called \textit{left} and \textit{right}
- a node is the \textit{parent} of its children
- a childless node is called a \textit{leaf}, and the other nodes are \textit{internal nodes}
- a binary tree has at most one node that has no parent, i.e. the \textit{root}
- all other nodes are the root’s children, grandchildren etc.
Heap

An array $A[1 \ldots n]$ is a heap, if $A[i] \geq A[2i]$ and $A[i] \geq A[2i + 1]$ always when $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$ (and $2i + 1 \leq n$).

![Heap Example]

The structure is easier to understand if we define the heap as a completely balanced binary tree, where

- the root is stored in the array at index 1
- the children of the node at index $i$ are stored at $2i$ and $2i + 1$ (if they exist)
- the parent of the node at index $i$ is stored at $\lfloor i/2 \rfloor$

Thus, the value of each node is larger or equal to the values of its children.

Each level in the heap tree is full, except maybe the last one, where only some rightmost leaves may be missing.

In order to make it easier to see the heap as a tree, let’s define subroutines that find the parent and the children.

- they can be implemented very efficiently by shifting bits
- the running time of each is always $\Theta(1)$

```plaintext
PARENT(i) return \lfloor i/2 \rfloor
LEFT(i) return 2i
RIGHT(i) return 2i + 1
```

$\Rightarrow$ Now the heap property can be given with:

- $A[PARENT(i)] \geq A[i]$ always when $2 \leq i \leq A.heapsize$
- $A.heapsize$ gives the size of the heap (we’ll later see that it’s not necessarily the size of the array)

Due to the heap property, the largest element of the heap is always its root, i.e. at the first index in the array.

If the height of the heap is $h$, the amount of its nodes is between $2^h \ldots 2^{h+1} - 1$.

$\Rightarrow$ If there are $n$ nodes in the heap its height is $\Theta(\log n)$.

Adding an element to the heap from the top:

- let’s assume that $A[1 \ldots n]$ is otherwise a heap, except that the heap property does not hold for the root of the heap tree

  ![Adding Element Example]

  $\Rightarrow$ the tree becomes a heap
The same is pseudocode

\begin{verbatim}
HEAPIFY(A, i)  // i is the index where the element might be too small
repeat         // (repeat until the heap is fixed)
  old := i
  l := LEFT(i)
  r := RIGHT(i)
if l ≤ A.heapsize and A[l] > A[i] then  // (the left child is larger than i)
  i := l
if r ≤ A.heapsize and A[r] > A[i] then  // (right child is even larger)
  i := r
if i ≠ old then                           // (if a larger child was found...)
until i = old                             // (if the heap is already fixed, exit)
\end{verbatim}

- The execution is constant time if the condition on line 11 is true the first time it is met: \( \Omega(1) \).
- In the worst case the new element needs to be moved all the way down to the leaf.
  \( \Rightarrow \) The running time is \( O(h) = O(\log n) \).

- **BUILD-HEAP** executes the \textbf{for}-loop \( \lfloor \frac{n}{2} \rfloor \) times and HEAPIFY is \( \Omega(1) \) and \( O(\log n) \) so
  - the best case running time is \( \lfloor \frac{n}{2} \rfloor \cdot \Omega(1) + \Theta(n) = \Omega(n) \)
  - the program never uses more than \( \lfloor \frac{n}{2} \rfloor \cdot O(\log n) + \Theta(n) = O(n \log n) \)
- The worst-case running time we get this way is however too pessimistic:
  - HEAPIFY is \( O(h) \), where \( h \) is the height of the heap tree
  - as \( i \) changes the height of the tree changes

\[
\begin{array}{|c|c|}
\hline
\text{level} & \text{height} & \text{executions times of HEAPIFY} \\
\hline
\text{lowest} & 0 & 0 \\
2nd & 1 & \lfloor \frac{\log n}{2} \rfloor \\
3rd & 2 & \lfloor \frac{\log n}{4} \rfloor \\
\vdots & \vdots & \vdots \\
\text{topmost} & |\log n| & 1 \\
\hline
\end{array}
\]

- thus the worst case runnig time is \( \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{3}{2} \cdot \frac{5}{2} \cdot \frac{7}{2} \cdot \frac{9}{2} \cdot \cdots = \)

Building a heap

- the following algorithm converts an array into a heap:

\begin{verbatim}
BUILD-HEAP(A)
1 A.heapsize := A.length         // (the heap is built out of the entire array)
2 for i := A.length downto 1 do  // (scan through the lower half of the array)
3   HEAPIFY(A, i)                // (call Heaipy)
\end{verbatim}

- The array is scanned from the end towards the beginning and HEAPIFY is called for each node.
  - before calling HEAPIFY the heap property always holds for the subtree rooted at \( i \) except that the element in \( i \) may be too small
  - subtrees of size one don’t need to be fixed as the heap property trivially holds
  - after HEAPIFY(A, i) the subtree rooted at \( i \) is a heap
  \( \Rightarrow \) after HEAPIFY(A, 1) the entire array is a heap

\[
\frac{1}{2} \cdot \frac{3}{4} \cdot \frac{5}{8} \cdot \frac{7}{16} \cdot \cdots = n \Rightarrow O(n)
\]

\( \Rightarrow \) the running time of BUILD-HEAP is always \( \Theta(n) \)

Sorting with a heap

The following algorithm can be used to sort the contents of the array efficiently:

\begin{verbatim}
HEAPSORT(A)
1 BUILD-HEAP(A)          // (convert the array into a heap)
2 for i := A.length downto 2 do  // (scan the array from the last to the first element)
4   A.heapsize := A.heapsize - 1  // (move the largest element outside the heap)
5   HEAPIFY(A, 1)               // (fix the heap, which is otherwise fine...)
\end{verbatim}

(\text{.. except the first element may be too small})
Let’s draw a picture of the situation:
• first the array is converted into a heap
• it’s easy to see from the example, that the operation is not too laborious
  – the heap property is obviously weaker than the order
  – the picture shows how the sorted range at the end of the array gets larger until the entire array is sorted
  – the heap property is fixed each time the sorted range gets larger
  – the fixing process seems complex in such a small example
  – the fixing process doesn’t take a lot of steps even with large arrays, only a logarithmic amount

Advantages and disadvantages of HEAPSORT

Advantages:
• sorts the array in place
• never uses more than \( \Theta(n \lg n) \) time

Disadvantages:
• the constant coefficient in the running time is quite large
• instability
  – elements with the same value don’t maintain their order

The running time of HEAPSORT consists of the following:
• \textsc{Build-Heap} on line 1 is executed once: \( \Theta(n) \)
• the contents of the for-loop is executed \( n - 1 \) times
  – operations on lines 3 and 4 are constant time
  – \textsc{Heaify} uses \( \Omega(1) \) and \( O(lg \ n) \)
  \( \Rightarrow \) in total \( \Omega(n) \) and \( O(n \ lg \ n) \)
• the lower bound is exact
  – if all the elements have the same value the heap doesn’t need to be fixed at all and \textsc{Heaify} is always constant time
• the upper bound is also exact
  – proving this is more difficult and we find the upcoming result from the efficiency of sorting by counting sufficient

Note! The efficiency calculations above assume that the data structure used to store the heap provides a constant time indexing.
• Heap is worth using only when this is true

5.2 Priority queue

A priority queue is a data structure for maintaining a set \( S \) of elements, each associated with a key value. The following operations can be performed:
• \textsc{Insert}(\( S, x \)) inserts the element \( x \) into the set \( S \)
• \textsc{Maximum}(\( S \)) returns the element with the largest key
  – if there are several elements with the same key, the operation can choose any one of them
• \textsc{Extract-Max}(\( S \)) removes and returns the element with the largest key
  – alternatively the operations \textsc{Minimum}(\( S \)) and \textsc{Extract-Min}(\( S \)) can be implemented
  – there can be only the maximum or only the minimum operations implemented in the same queue
Priority queues can be used widely
- prioritizing tasks in an operating system
  - new tasks are added with the command INSERT
  - as the previous task is completed or interrupted the next one is chosen with EXTRACT-MAX
- action based simulation
  - the queue stores incoming (not yet simulated) actions
  - the key is the time the action occurs
  - an action can cause new actions
    ⇒ they are added to the queue with INSERT
  - EXTRACT-MIN gives the next simulated action
- finding the shortest route on a map
  - cars driving at constant speed but choosing different routes are simulated until the first one reaches the destination
- a priority queue is needed in practice in an algorithm for finding shortest paths, covered later

In practice, a priority queue could be implemented with an unsorted or sorted array, but that would be inefficient
- the operations MAXIMUM and EXTRACT-MAX are slow in an unsorted array
- INSERT is slow in a sorted array
A heap can be used to implement a priority queue efficiently instead.
- The elements of the set $S$ are stored in the heap $A$.
- MAXIMUM$(S)$ is really simple and works in $O(1)$ running-time

$\text{HEAP-MAXIMUM}(A)$
1. if $A\text{.heapsize} < 1$ then (there is no maximum in an empty heap)
2. error “heap underflow”
3. return $A[1]$ (otherwise return the first element in the array)

$\text{HEAP-INSERT}(A, \text{key})$
1. $A\.heapsize := A\.heapsize + 1$ (increment the size of the heap)
2. $i := A\.heapsize$ (start from the end of the array)
3. while $i > 1$ and $A[\text{PARENT}(i)] < \text{key}$ do (continue until the root or an parent with a larger value is reached)
4. $A[i]\assign A[\text{PARENT}(i)]$ (move the parent downwards)
5. $i := \text{PARENT}(i)$ (move upwards)
6. $A[i] := \text{key}$ (place the key into its correct location)

⇒ Each operation in the priority queue can be made $O(\lg n)$ by using a heap.
A priority queue can be thought of as an abstract data type which stores the data (the set S) and provides the operations
INSERT, MAXIMUM, EXTRACT-MAX.

- the user sees the names and the purpose of the operations but not the implementation
- the implementation is encapsulated into a package (Ada), a class (C++) or an independent file (C)

⇒ it's easy to maintain and change the implementation when needed without needing to change the code using the queue.

5.3 QUICKSORT

This chapter covers a very efficient sorting algorithm QUICKSORT.

Like MERGE-SORT, QUICKSORT is a divide and conquer algorithm. However, with MERGE-SORT the division is simple and combining the results is complex, with QUICKSORT it's vice versa.

The division of the problem into smaller subproblems
- Select one of the elements in the array as a pivot, i.e. the element which partitions the array.
- Change the order of the elements in the array so that all elements smaller or equal to the pivot are placed before it and the larger elements after it.
- Continue dividing the upper and lower halves into smaller subarrays, until the subarrays contain 0 or 1 elements.

Smaller subproblems:
- Subarrays of the size 0 and 1 are already sorted

Combining the sorted subarrays:
- The entire (sub) array is automatically sorted when its upper and lower halves are sorted.
  - all elements in the lower half are smaller than the elements in the upper half, as they should be

QUICKSORT-algorithm

QUICKSORT(A, p, r)
1  if p < r then (do nothing in the trivial case)
2     q := PARTITION(A, p, r) (partition in two)
3     QUICKSORT(A, p, q − 1) (sort the elements smaller than the pivot)
4     QUICKSORT(A, q + 1, r) (sort the elements larger than the pivot)

PARTITION(A, p, r)
1  x := A[r] (choose the last element as the pivot)
2  i := p − 1 (use i to mark the end of the smaller elements)
3  for j := p to r − 1 do (scan to the second to last element)
4      if A[j] ≤ x (if A[j] goes to the half with the smaller elements...)
5          i := i + 1 (... increment the amount of the smaller elements...)
6      end if
8    end for
9  exchange A[i + 1] ↔ A[r] (place the pivot between the halves)
10  return i + 1 (return the location of the pivot)

How fast is PARTITION?
- The for-loop is executed n - 1 times when n is r - p
- All other operations are constant time.

⇒ The running-time is O(n).
Determining the running-time of QUICKSORT is more difficult.

We'll analyze in the same way we did with MERGE-SORT

- As all the operations of QUICKSORT except PARTITION and the recursive call are constant time, let's concentrate on the time used by the instances of PARTITION.

Worst-case running time

- The number of a node is always smaller than the number of its parent, since the pivot is already in its correct location and doesn't go into either of the sorted subarrays
  \[ \Rightarrow \text{there can be at most } n \text{ levels in the tree} \]
- The worst case is realized when the smallest or the largest element is always chosen as the pivot
  - this happens, for example, with an array already sorted
- the sum of the node numbers is \( n + n - 1 + \cdots + 2 + 1 \)
  \[ \Rightarrow \text{the running time of QUICKSORT is } O(n^2) \]

The best-case is when the array is always divided evenly in half.

- The total time is the sum of the running times of the nodes in the picture above.
- The execution is constant time for an array of size 1.
- For the other the execution is linear to the size of the array.
  \[ \Rightarrow \text{the total time is } \Theta(\text{the sum of the numbers of the nodes}) \]

\[ \begin{array}{c}
  n \\
  n_1 \\
  n_2 \\
  \vdots \\
  1 \\
\end{array} \]

\[ \begin{array}{c}
  n - n_1 \\
  n - n - 1 \\
  n - 1 \\
  \vdots \\
  1 \\
\end{array} \]

\[ \begin{array}{c}
  n - n_2 \\
  n - n - 1 \\
  \vdots \\
  1 \\
\end{array} \]

\[ \begin{array}{c}
  n - n - 1 \\
  \vdots \\
  1 \\
\end{array} \]

\[ \begin{array}{c}
  1 \\
  \vdots \\
  1 \\
\end{array} \]
The best-case and the worst-case efficiencies of QUICKSORT differ significantly.

- It would be interesting to know the average-case running-time.
- Analyzing it is beyond the goals of the course but it has been shown that if the data is evenly distributed its average running-time is $\Theta(n \lg n)$.
- Thus the average running-time is quite good.

An unfortunate fact with QUICKSORT is that its worst-case efficiency is poor and in practise the worst-case situation is quite probable.

- It is easy to see that there can be situations where the data is already sorted or almost sorted.

$\Rightarrow$ A way to decrease the risk of the systematic occurrence of the worst-case situation’s likelihood is needed.

Randomization has proved to be quite efficient.

Advantages and disadvantages of QUICKSORT

Advantages:

- sorts the array very efficiently in average
  - the average-case running-time is $\Theta(n \lg n)$
  - the constant coefficient is small
- requires only a constant amount of extra memory
- if well-suited for the virtual memory environment

Disadvantages:

- the worst-case running-time is $\Theta(n^2)$
- without randomization the worst-case input is far too common
- the algorithm is recursive
  $\Rightarrow$ the stack uses extra memory
- instability

5.4 Randomization

Randomization is one of the design techniques of algorithms.

- A pathological occurrence of the worst-case inputs can be avoided with it.
- The best-case and the worst-case running-times don’t usually change, but their likelihood in practise decreases.
- Disadvantageous inputs are exactly as likely as any other inputs regardless of the original distribution of the inputs.
- The input can be randomized either by randomizing it before running the algorithm or by embedding the randomization into the algorithm.
  - the latter approach usually gives better results
  - often it is also easier than preprocessing the input.

$\Rightarrow$ Randomization is usually a good idea when

- the algorithm can continue its execution in several ways
- it is difficult to see which way is a good one
- most of the ways are good
- a few bad guesses among the good ones don’t make much damage

For example, QUICKSORT can choose any element in the array as the pivot

- besides the almost smallest and the almost largest elements, all other elements are a good choice
- it is difficult to guess when making the selection whether the element is almost the smallest/largest
- a few bad guesses now and then doesn’t ruin the efficiency of QUICKSORT

$\Rightarrow$ randomization can be used with QUICKSORT
With randomization an algorithm \textsc{Randomized-Quicksort} which uses a randomized \textsc{Partition} can be written

- \( A[r] \) is not always chosen as the pivot. Instead, a random element from the entire subarray is selected as the pivot
- In order to keep \textsc{Partition} correct, the pivot is still placed in the index \( r \) in the array
  \( \Rightarrow \) Now the partition is quite likely even regardless of the input and how the array has earlier been processed.

\textsc{Randomized-Partition}(\( A, p, r \))
1 \( i := \text{Random}(p, r) \) \hspace{1cm} (choose a random element as pivot)
2 exchange \( A[r] \leftrightarrow A[i] \) \hspace{1cm} (store it as the last element)
3 return \textsc{Partition}(\( A, p, r \)) \hspace{1cm} (call the normal partition)

\textsc{Randomized-Quicksort}(\( A, p, r \))
1 if \( p < r \) then
2 \( q := \text{Randomized-Partition}(A, p, r) \)
3 \text{Randomized-Quicksort}(A, p, q - 1)
4 \text{Randomized-Quicksort}(A, q + 1, r)

\( \Rightarrow \text{Randomized-Quicksort} \) is better than the normal \textsc{Quicksort} in general
\textsc{Quicksort} can be made more efficient with other methods:
- An algorithm efficient with small inputs (e.g. \textsc{InsertionSort}) can be used to sort the subarrays.
  - they can also be left unsorted and in the end sort the entire array with \textsc{InsertionSort}
- The median of three randomly selected elements can be used as the pivot.

The running-time of \textsc{Randomized-Quicksort} is \( \Theta(n \log n) \) on average just like with normal \textsc{Quicksort}.

- However, the assumption made in analyzing the average-case running-time that the pivot-element is the smallest, the second smallest etc. element in the subarray with the same likelihood holds for \textsc{Randomized-Quicksort} for sure.
- This holds for the normal \textsc{Quicksort} only if the data is evenly distributed.

\( \Rightarrow \text{Randomized-Quicksort} \) is better than the normal \textsc{Quicksort} in general

\textsc{Quicksort} can be made more efficient with other methods:
- It's always possible to use the median as the pivot.
  - The median can be found efficiently with the so called lazy \textsc{Quicksort}.
    - Divide the array into a "small elements" lower half and a "large elements" upper half like in \textsc{Quicksort}.
    - Calculate which half the \( i \)th element belongs to and continue recursively from there.
    - The other half does not need to be processed further.

\textsc{Randomized-Select}(\( A, p, r, i \))
1 if \( p = r \) then \hspace{1cm} (if the subarray is of size 1...)
2 return \( A[p] \) \hspace{1cm} (... return the only element)
3 \( q := \text{Randomized-Partition}(A, p, r) \) \hspace{1cm} (divide the array into two halves)
4 \( k := q - p + 1 \) \hspace{1cm} (calculate the number of the pivot)
5 if \( i = k \) then \hspace{1cm} (if the pivot is the \( i \)th element in the array...)
6 return \( A[q] \) \hspace{1cm} (... return it)
7 else if \( i < k \) then \hspace{1cm} (continue the search from the small ones)
8 return \text{Randomized-Select}(A, p, q - 1, i) \hspace{1cm} (continue on the large ones)
9 else \hspace{1cm} (continue on the large ones)
10 return \text{Randomized-Select}(A, q + 1, r, i - k)

\( \Rightarrow \text{The algorithm's running-time is } \Omega(n) \).

The lower-bound for the running-time of \textsc{Randomized-Select}:
- Again everything else is constant time except the call of \textsc{Randomized-Partition} and the recursive call.
- In the best-case the pivot selected by \textsc{Randomized-Partition} is the \( i \)th element and the execution ends.
- \textsc{Randomized-Partition} is run once for the entire array.

\( \Rightarrow \text{The algorithm's running-time is } \Omega(n) \).

The upper-bound for the running-time of \textsc{Randomized-Select}:
- \textsc{Randomized-Partition} always ends up choosing the smallest or the largest element and the \( i \)th element is left in the larger half.
- the amount of work is decreased only by one step on each level of recursion.

\( \Rightarrow \text{The running-time of the algorithm is } O(n^2) \).
The average-case running-time is however $O(n)$. The algorithm is found in STL under the name nth_element. The algorithm can also be made to always work in linear time.

5.5 Other sorting algorithms

All sorting algorithms covered so far have been based on comparisons.

- They determine the correct order only based on comparing the values of the elements to each other.
- It is possible to use information other than comparisons to sort the data.

Sorting by counting

Let’s assume that the value range of the keys is small, at most on the same scale with the amount of the elements.

- For simplicity we assume that the keys of the elements are from the set $\{1, 2, \ldots, k\}$, and $k = O(n)$.
- For each key the amount of elements with the given key is calculated.
- Based on the result the elements are placed directly into their correct positions.

```
COUNTING-SORT(A, B, k)
1 for i := 1 to k do       (initialize a temp array C with zero)
2     C[i] := 0
3 for j := 1 to A.length do    (calculate the amount of elements with key = i)
5 for i := 2 to k do          (calculate how many keys ≤ i)
6     C[i] := C[i] + C[i−1]
7 for j := A.length downto 1 do (scan the array from end to beginning)
8     B[C[A[j].key]] := A[j]   (place the element into the output array)
9     C[A[j].key] := C[A[j].key] − 1    (the next correct location is a step to the left)
```

The algorithm places the elements to their correct location in a reverse order in order to guarantee stability.

Running-time:
- The first and the third for-loop take $O(k)$ time.
- The second and the last for-loop take $O(n)$ time.
- The running time is $O(n + k)$.
- If $k = O(n)$, the running-time is $O(n)$.

COUNTING-SORT is not worth using if $k \gg n$.

- The memory consumption of the algorithm is $O(k)$.
- Usually $k \gg n$.
- For example: all possible social security numbers $\gg$ the social security numbers of TUT personnel

Sometimes there is a need to be able to sort based on a key with several parts.
- the list of exam results first based on the department and then into an alphabetical order
- dates first based on the year, then the month and then the day
- a deck of cards first based on the suit and then according to the numbers
The different criteria are taken into account as follows
- The most significant criterion according to which the values of the elements differs determines the result of the comparison.
- If the elements are equal with each criteria they are considered equal.

The problem can be solved with a comparison sort (e.g. by using a suitable comparison operator in \textsc{QuickSort})
- example: comparing dates

\begin{verbatim}
\textbf{DATE-COMPARE}(x, y)
1 \textbf{if} x.\text{year} < y.\text{year} \textbf{then return} "smaller"
2 \textbf{if} x.\text{year} > y.\text{year} \textbf{then return} "greater"
3 \textbf{if} x.\text{month} < y.\text{month} \textbf{then return} "smaller"
4 \textbf{if} x.\text{month} > y.\text{month} \textbf{then return} "greater"
5 \textbf{if} x.\text{day} < y.\text{day} \textbf{then return} "smaller"
6 \textbf{if} x.\text{day} > y.\text{day} \textbf{then return} "greater"
7 \textbf{return} "equal"
\end{verbatim}

Sometimes it makes sense to handle the input one criterion at a time.
- For example it’s easiest to sort a deck of cards into four piles based on the suits and then each suit separately.

The range of values in the significant criteria is often small when compared to the amount of element and \textsc{Counting-Sort} can be used.

There are two different algorithms available for sorting with multiple keys.
- \textsc{LSD-Radix-Sort}
  - the array is sorted first according to the least significant digit, then the second least significant etc.
  - the sorting algorithm needs to be stable - otherwise the array would be sorted only according to the most significant criterion
  - \textsc{Counting-Sort} is a suitable algorithm
  - comparison algorithms are not worth using since they would sort the array with approximately the same amount of effort directly at one go

\begin{verbatim}
\textbf{LSD-RADIX-SORT}(A, d)
1 \textbf{for} i := 1 \textbf{to} d \textbf{do} \hspace{1cm} (run through the criteria, least significant first)
2 \hspace{1cm} \textbf{sort} \ A \ with \ a \ stable \ sort \ according \ to \ criterion \ i
\end{verbatim}

- \textsc{MSD-Radix-Sort}
  - the array is first sorted according to the most significant digit and then the subarrays with equal keys according to the next significant digit etc.
  - does not require the sorting algorithm to be stable
  - usable when sorting character strings of different lengths
  - checks only as many of the sorting criterions as is needed to determine the order
  - more complex to implement than \textsc{LSD-Radix-Sort}
  \Rightarrow the algorithm is not given here

The efficiency of \textsc{Radix-Sort} when using \textsc{Counting-Sort}:
- sorting according to one criterion: $\Theta(n + k)$
- amount of different criteria is $d$
  \Rightarrow total efficiency $\Theta(dn + dk)$
- $k$ is usually constant
  \Rightarrow total efficiency $\Theta(dn)$, or $\Theta(n)$, if $d$ is also constant
**Radix-Sort** appears to be a $O(n)$ sorting algorithm with certain assumptions.

Is it better than the comparison sorts in general?

When analyzing the efficiency of sorting algorithms it makes sense to assume that all (or most) of the elements have different values.

- For example, **Insertion-Sort** is $O(n)$, if all elements are equal.
- If the elements are all different and the size of value range of one criterion is constant $k$, $k^d \geq n \Rightarrow d \geq \log_k n = \Theta(\log n)$

$\Rightarrow$ **Radix-Sort** is $\Theta(dn) = \Theta(\log n)$, if we assume that the element values are mostly different from each other.

**Radix-Sort** is asymptotically as slow as other good sorting algorithms.

- By assuming a constant $d$, **Radix-Sort** is $\Theta(n)$, but then with large values of $n$ most elements are equal to each other.

---

**Bucket sort**

Let’s assume that the keys are within a known range of values and the key values are evenly distributed.

- Each key is just as probable.
- For the sake of an example, we’ll assume that the key values are between zero and one.
- Let’s use $n$ buckets $B[0] \ldots B[n-1]$.

**Bucket-Sort(A)**

1. $n := \text{A.length}$
2. for $i := 1$ to $n$ do (go through the elements)
  3. $\text{Insert(B[(n - A[i])]}, A[i])$ (throw the element into the correct bucket)
4. $k := 1$ (start filling the array from index 1)
5. for $i := 0$ to $n - 1$ do (go through the buckets)
  6. while $B[i]$ not empty do (empty non-empty buckets...)
    7. $A[k] := \text{Extract-Min(B[i])}$ (... by moving the elements, smallest first...)
  8. $k := k + 1$ (... into the correct location in the result array)

---

**Advantages and disadvantages of Radix-Sort**

**Advantages:**

- **Radix-Sort** is able to compete in efficiency with **QuickSort** for example
  - if the keys are 32-bit numbers and the array is sorted according to 8 bits at a time
    $\Rightarrow k = 2^8$ and $d = 4$
    $\Rightarrow$ Counting-Sort is called four times
- **Radix-Sort** is well suited for sorting according to keys with multiple parts when the parts of the key have a small value range.
  - e.g. sorting a text file according to the characters on the given columns (cmp. Unix or MS/DOS sort)

**Disadvantages:**

- Counting-Sort requires another array $B$ of $n$ elements where it builds the result and a temp array of $k$ elements.
  $\Rightarrow$ It requires $\Theta(n)$ extra memory which is significantly larger than for example with QuickSort and Heapsort.

---

**Implementation of the buckets:**

- Operations **Insert** and **Extract-Min** are needed.
  $\Rightarrow$ The bucket is actually a priority queue.
- The size of the buckets varies a lot.
  - usually the amount of elements in a bucket is $\approx 1$
  - however it is possible that every element ends up in the same bucket
  $\Rightarrow$ an implementation that uses a heap would require $\Theta(n)$ for each bucket $\Theta(n^2)$ in total
- On the other hand, the implementation does not need to be very efficient for large buckets since they are rare.
  $\Rightarrow$ In practice the buckets should be implemented as lists.
  - **Insert** links the incoming element to its correct location in the list, $\Theta(1)$ time is used
  - **Extract-Min** removes and returns the first element in the list, $\Theta(l)$ time is used
the average efficiency of BUCKET-SORT:

- On average one element falls into each bucket and very rarely a significantly larger amount of elements fall into the same bucket.
- The first for-loop runs through all of the elements, \( \Theta(n) \).
- The second for-loop runs through the buckets, \( \Theta(n) \).
- The while-loop runs through all of the elements in all of its iterations in total once, \( \Theta(n) \).
- INSERT is on average constant time, since there is an average one element in the bucket.
- EXTRACT-MIN is constant time.

In the slowest case all elements fall into the same bucket in an ascending order.
- INSERT takes a linear amount of time
- The total running-time is \( \Theta(n^2) \) in the worst-case.

So many comparisons need to be made that the only correct alternative gets chosen from the set.

- Each comparison \( A[i] \leq A[j] \) (or \( A[i] < A[j] \)) divides the permutations into two groups: those where the order of \( A[i] \) and \( A[j] \) must be switched and those where the order is correct so...
  - one comparison in enough to pick the right alternative from atmost two
  - two comparisons in enough to pick the right one from atmost four
  - \( \ldots \)
  - \( k \) comparisons in enough to pick the right alternative from atmost \( 2^k \)
  - choosing the right one from \( x \) alternatives requires at least \( \lceil \log x \rceil \) comparisons
- If the size of the array is \( n \), there are \( n! \) permutations
  - At least \( \lceil \log n! \rceil \) comparisons is required
  - a comparison sort algorithm needs to use \( \Omega(\lceil \log n! \rceil) \) time.

5.6 How fast can we sort?

Sorting an array actually creates the permutation of its elements where the original array is completely sorted.

- If the elements are all different, the permutation is unique. \( \Rightarrow \) Sorting searches for that permutation from the set of all possible permutations.

For example the functionality of INSERTION-SORT, MERGE-SORT, HEAPSORT and QUICKSORT is based on comparisons between the elements.

- Information about the correct permutation is collected only by comparing the elements together.

What would be the smallest amount of comparisons that is enough to find the correct permutation for sure?

- An array of \( n \) elements of different values has \( 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \) i.e. \( n! \) permutations.

How large is \( \lceil \log n! \rceil \)?

- \( \lceil \log n! \rceil \geq \lceil \log n \rceil \geq \sum_{k=1}^{n} \log k \geq \sum_{k=2}^{n} \log k \geq \frac{1}{2} \cdot n \log \frac{n}{2} = \frac{1}{2} n \log n - \frac{n}{2} = \Omega(n \log n) - \Omega(n) = \Omega(n \log n) \)

- on the other hand \( \lceil \log n! \rceil < n \log n + 1 = O(n \log n) \)
  \( \Rightarrow \lceil \log n! \rceil = \Theta(n \log n) \)

Every comparison sort algorithm needs to use \( \Omega(n \log n) \) time in the slowest case.

- On the other hand HEAPSORT and MERGE-SORT are \( O(n \log n) \) in the slowest case.
  \( \Rightarrow \) In the slowest case sorting based on comparisons between elements is possible in \( \Theta(n \log n) \) time, but no faster.

- HEAPSORT and MERGE-SORT have an optimal asymptotic running-time in the slowest case.

- Sorting is for real asymptotically more time consuming than finding the median value, which can be done in the slowest possible case in \( O(n) \).
6 List- and array structures
The use of pointers and the basics of list structures are revised in this chapter. A few simple data structures that can be implemented with arrays and/or lists are also covered. In addition a data abstraction called a dynamic set is defined.

6.1 Definition of a dynamic set
A dynamic set maintains a collection of elements. Elements can be added, removed and searched according to different principles. 
From the user’s point of view an element consists of two parts: a key and satellite data
- a student number is the key and the satellite data contains the name, the department, credits etc.
- in a phonebook both the name and the telephone number can act as keys, so that searches can be done according to either of them
- the implementation usually contains extra fields in the structure that are used to maintain the data structure

Searching for data is done according to the key in dynamic sets.
- satellite data is handled only through a key and a handle ⇒ the satellite data is insignificant from the algorithms point of view
  - since satellite data may be moved around with the key, a large payload should be stored with a pointer to the data
- two elements can or cannot have the same key value, depending on the situation
- if several elements have the same key, an element can be chosen
  - arbitrarily
  - according to secondary criteria

- the set of possible key values can be sorted ⇒ the smallest element or an element larger than a given value can be searched for
  - some data structures cannot handle the order information efficiently (hash table)
  - whereas others actually insist that the set of elements is sorted (binary trees)

The search usually returns a handle to the element which identifies the element unambiguously from the others, even though they might have the same key.
- pointers, iterators and indices are typical handles
The operations of a dynamic set can be grouped into two categories: queries, which don’t change the structure but only return information and modifying operations

- **SEARCH**\((S, k)\) returns a handle to an element \(x\) in the set \(S\) such that \(x.key = k\), or NIL if no such element exists
- **INSERT**\((S, x)\) add the element \(x\) to the set \(S\)
- **DELETE**\((S, x)\) removes an element from the set \(S\) with the handle \(x\)
- **MINIMUM**\((S)\) returns a handle to the element of the set \(S\) with the smallest key value
  - sensible only if the set of keys is ordered
- **MAXIMUM**\((S)\) cmp. **MINIMUM**
- **SUCCESSOR**\((S, x)\) returns a handle to the element with the next largest key to the element \(x\)
- **PREDECESSOR**\((S, x)\) cmp. **SUCCESSOR**
- the elements can be stepped through e.g. with **SUCCESSOR**

- it’s worth a mention that minimum, maximum and scan operations are not necessarily unambiguous if several elements have the same key

The best way to implement a dynamic set depends on the operations needed and which of the operations need to be fast and which can be slow.
⇒ it’s useful to define some special cases that don’t allow all operations to be performed
- for example a dictionary provides only the operations **SEARCH**, **INSERT** and **DELETE**
  ⇒ the value set of the keys does not need to be sorted
- a priority queue is a dynamic set that provides operations **INSERT** and **MAXIMUM** in addition with the operation **EXTRACT-MAX**, that can be used to remove elements in a limited situation
- the queue can be seen as a dynamic set where the key has been selected in a unique way
  - only the operations **INSERT** and **EXTRACT-MIN**

6.2 Pointers and lists

One of the simplest dynamic sets is a linked list. It’s easy to implement records or objects in several programming languages and then handle them through pointers.
- the pointer indicates where the object can be found
  - a pointer can be seen as an arrow pointing to the object
  - a pointer can also point nowhere: "NIL"
- in linked structures there are pointer fields in the objects pointing to other objects
- the fields in the objects can be read and changed through the pointer
- when the value of the pointer is changed it turns to point to another object
  ⇒ a structure that uses pointers can be stepped through
- the entire structure can be changed by changing the pointer fields of an element in the structure
A lot can be done with pointers.

A doubly linked list is a basic linked structure.
- each element contains the fields `next`, `prev` and the actual payload
- for simplicity in the examples the only payload is the key

Searching from a list:

```
LIST-SEARCH(L, k)
1 x := L.head (start scanning from the beginning of the list)
2 while x != NIL and x->key != k do continue until the end is reached or the element found
3 x := x->next
4 return x (return the found element or NIL)
```
- the worst-case running-time is \( \Theta(n) \), when the element is the last one in the list or cannot be found at all

The algorithms above can be changed to handle a singly linked list by leaving out the field `prev` and all operations manipulating it.

Additionally the removing algorithms needs to be changed so that `x` points to the element infront of the element under removal in order to avoid searching for the previous element.

Handling the first and the last element in the list often need to be handled as special cases.
- this makes the algorithm more complex

Inserting an element `x` into the beginning of the list:

```
LIST-INSERT(L, x) (x points to the added element)
1 x->next := L.head
2 if L.head != NIL then (if the list is not empty . . .)
3 L.head->prev := x
4 L.head := x (make x the first element)
```
- running-time is always \( \Theta(1) \)

Removing an element:

```
LIST-DELETE(L, x) (x points to the element under removal)
1 if x->prev != NIL then (if there is a previous element . . .)
2 x->prev->next := x->next (link the next after the previous)
3 else
4 L.head := x->next (make the next the first element)
5 if x->next != NIL then (if the next element exists . . .)
6 x->next->prev := x->prev (link the previous infront of the next)
```
- running-time is always \( \Theta(1) \)

- sometimes it’s usefull to add an extra object to the end of the list, the sentinel, which is not used to store any payload
- a similar solution is a circular list

Typically the object `x` points to in the algorithms on the previous page need to
- reserved before calling `LIST-INSERT`
- freed after calling `LIST-DELETE` unless it needs to be saved for some other purpose
- some programming languages take care of the allocation and deallocation of objects automatically
  - e.g. Lisp, Scheme, Smalltalk, Java, (Eiffel)
  - garbage collection
  - algorithms are often heavy and can cause longish, unexpected delays
• when using other languages or when programming heavy duty programs the programmer must take care of the allocation and freeing of objects manually
  – this may cause errors that are difficult to find and may have unpleasant consequences
  – a object is freed too early
    ⇒ the memory of the object may be reallocated creating two uses for the same memory segments
    ⇒ wrong data is changed
  – freeing the object is forgotten causing a memory leak
    ⇒ the amount of free memory slowly decreases until it runs out
    ⇒ eventually the program crashes or starts acting weirdly

In general, reusing objects freed by the programmer is difficult and the implementation of the language handles it badly.
⇒ It’s often worth organizing the recycling of free objects by maintaining a list of free objects, a free list.

• the idea is not to free the objects with delete but collect the unused objects into a list
• once a new object is needed, it is taken from the free list provided it’s not empty
• objects are allocated with new only when the free list is empty
• a suitable pointer field is chosen from the objects for maintaining the free list

• allocating an object:

ALLOCATE-OBJECT
1  if free = NIL then (if the free list is empty, allocate a new object)
2    new(x)                  (else …)
3  else
4    x := free             (… make the second element in the list the first …)
5    free := x->next        (… and return the first)
6  return x

• freeing an object:

FREE-OBJECT(x)
1  x->next := free        (link the new element in front of the first)
2  free := x             (make the new element the first in the free list)

A linked list can be used in the implementation of a stack.
• a data abstraction with operations
  – STACK-EMPTY(S) tests whether the stack is empty
  – PUSH(S, x) adds the element x to the top of the stack
  – POP(S) removes and returns the topmost element in the stack
• there may be other operations depending on the circumstances
  – STACK-FULL(S)
  – STACK-COUNT(S) returns the amount of elements in the stack
• implements a lifo (last in, first out) policy
• each operation has always \(\Theta(1)\) running time
• there is no need to predefine a maximum size, the stack can get larger as long as there is memory available
An array can be used in the implementation of the stack.
- each operation has always \( \Theta(1) \) running-time
- the downsize is that the maximum size needs to be fixed beforehand or a flexible sized array must be used

```plaintext
STACK-EMPTY(S)
1 if S.top = nil then  (if the topmost element is nil)
2 return true        (.. stack is empty)
3 else
4 return false

PUSH(S, x)
1 r := ALLOCATE-OBJECT (allocate a list object)
2 if r = nil then    (memory allocation failed)
3 error "out of memory"
4 else
5 r.payload := x    (copy the data into the list object)
6 r.next := S.top   (mark the topmost element as next)
7 S.top := r       (set the new element to the top of the stack)

POP(S)
1 if STACK-EMPTY(S) then (empty is empty)
2 error "underflow"
3 else
4 x := S.top.payload  (copy the payload of the topmost element)
5 r := S.top.next     (store the location of the next element)
6 FREE-OBJECT(S.top) (free the popped element)
7 S.top := r         (make the next element topmost)
8 return x           (return the payload of the popped element)
```

Another common data structure implemented with a singly linked list is a queue.
- a data abstraction with operations
  - QUEUE-EMPTY(Q) tests whether the queue is empty
  - ENQUEUE(Q, x) adds the element \( x \) to the end of the queue
  - DEQUEUE(Q) removes and returns the first element in the queue
- implements a fifo (first in, first out) policy
- each operation has always \( \Theta(1) \) running time
- there is no need to predefined the maximum size, the queue can get larger as long as there is memory available

```plaintext
QUEUE-EMPTY(Q)
1 if Q.head = nil then (there is no first element)
2 return true
3 else
4 return false
```

The asymptotic running-time is the same in both cases but the constant coefficient is usually smaller in the array implementation.

The memory consumption of the solutions may the decisive factor in choosing between the alternatives.
- if the elements stored in the stack are large the list implementation uses less memory provided that an exact size is not known for the array
- if the elements are small, the list is better only if the array is kept less than 50% full
- if the elements exist anyway the stack can be created by adding an extra field into each object for maintaining the stack order
ENQUEUE(Q, x)
1 r := ALLOCATE-OBJECT
2 if r = NIL then 
3 error “out of memory” 
4 else 
5 r→payload := x 
6 if Q.head = NIL then 
7 Q.head := r 
8 else 
9 Q.tail→next := r 
10 Q.tail := r; 

DEQUEUE(Q)
1 if QUEUE-EMPTY(Q) then 
2 error “underflow” 
3 else 
4 s := Q.head→payload 
5 r := Q.head→next 
6 FREE-OBJECT(Q.head) 
7 Q.head := r; 
8 return s; 

The queue can also be implemented with an array as a ring buffer.

\[ Q_{head} \rightarrow Q_{tail} \rightarrow Q_{head} \]

- each operation has always \( \Theta(1) \) running-time
- the downside is that the maximum size needs to be known beforehand or a flexible sized array must be used.

Is keeping the list sorted worth it?

- scanning through a sorted list can be stopped once an element larger than the one under search has been met ⇒ it’s enough to scan through approximately half of the elements ⇒ scanning becomes more efficient by a constant coefficient
- in the addition, the correct location needs to be found ⇒ addition is \( \Theta(n) \)
- it’s usually not worth the effort to keep the list sorted unless it’s somehow beneficial to the other purposes where the list is also used
- if the same key cannot be stored into the list more than one the addition requires scanning anyway ⇒ maintaining the order becomes beneficial

The relations between the time and memory consumption of the list and array implementations is the same as with the stack implementation
**Skip list** is a randomized ordered linked list that can be used to implement the operations of a dynamic set efficiently

- it’s a relatively new idea: the main publication is from the year 1990
- each object has a *height* that is chosen randomly as follows:

<table>
<thead>
<tr>
<th>height</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>2</td>
<td>1/4</td>
</tr>
<tr>
<td>3</td>
<td>1/8</td>
</tr>
<tr>
<td>4</td>
<td>1/16</td>
</tr>
<tr>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>

- there is a list for each height $k$ containing the objects whose height is $\geq k$

- search is done by first stepping through the highest list until the next element is too large or NIL, then the next highest etc.
  - finally the element is found in the lowest list or we’re sure the element doesn’t exist
  - the average running-time $O(\lg n)$
- in the addition
  - a height $k$ is randomly selected for the element
  - the correct location is found: $O(\lg n)$
  - the element is linked into the lists: $1, \ldots, k$: $O(k)$
  - $k$ is on average 2 ⇒ the average running time is $O(\lg n)$
- removal

In tests skip lists have proven as efficient as many balanced tree structures but easier to implement.

---

### 6.3 Hash table

The basic idea behind hash tables is to reduce the range of possible key values in a dynamic set by using a hash function $h$ so that the keys can be stored in an array.

- the advantage of an array is the efficient, constant-time indexing it provides

Reducing the range of the keys creates a problem: collisions.
- more than one element can hash into the same slot in the hash table

The most common way to solve the problem is called chaining.
- all the elements that hash to the same slot are put into a linked list
- there are other alternatives
  - in *open addressing* the element is put into a secondary slot if the primary slot is unavailable
OHJ-2016 Utilization of Data Structures

– in some situations the range of key values is so small, that
it doesn’t need to be reduced and therefore there are no
 collisions either
– this direct-access table is very simple and efficient
– this course covers only hashing with chaining

The picture below shows a chained hash table, whose keys
have been hashed based on the first letter according the the
table given.

The chained hash table provides the dictionary operations
only, but those are very simple:

- **Chained-Hash-Search**($T, k$)
  ▷ find the element with key $k$ from the list $T[h(k)]$

- **Chained-Hash-Insert**($T, x$)
  ▷ add $x$ to the beginning of the list $T[h(x\rightarrow key)]$

- **Chained-Hash-Delete**($T, x$)
  ▷ remove $x$ from the list $T[h(x\rightarrow key)]$

Running-times:
- addition: $\Theta(1)$
- search: worst-case $\Theta(n)$
- removal: if the list is doubly-linked $\Theta(1)$; with a singly linked list
  worst-case $\Theta(n)$, since the predecessor of the element
  under removal needs to be searched from the list
  – in practise the difference is not significant since usually
  the element to be removed needs to be searched from
  the list anyway

The average running-times of the operations of a chained
hash table depend on the lengths of the lists.
- in the worst-case all elements end up in the same list and
  the running-times are $\Theta(n)$
- to determine the average-case running time we’ll use the
  following:
  - $m$ = size of the hash table
  - $n$ = amount of elements in the table
  - $\alpha = \frac{n}{m}$ = load factor i.e. the average length of the list
- in addition, in order to evaluate the average-case
efficiency an estimate on how well the hash function $h$
hashes the elements is needed
  - if for example $h(k) =$ the 3 highest bits in the name, all
    elements hash into the same list
  - it is often assumed that all elements are equally likely to
    hash into any of the slots
  - simple uniform hashing
  - we’ll also assume that evaluating $h(k)$ is $\Theta(1)$

- if an element that is not in the table is searched for, the
  entire list needs to be scanned through
  ⇒ on average $\alpha$ elements need to be investigated
  ⇒ the running-time is on average $\Theta(1 + \alpha)$
- if we assume that any of the elements in the list is the key
  with the same likelihood, on average half of the list needs
  to be searched through in the case where the key is found
  in the list
  ⇒ the running-time is $\Theta(1 + \frac{\alpha}{2}) = \Theta(1 + \alpha)$ on average
- if the load factor is kept under some fixed constant
  (e.g. $\alpha < 50 \%$), then $\Theta(1 + \alpha) = \Theta(1)$
  ⇒ all operations of a chained hash table can be
  implemented in $\Theta(1)$ running-time on average
  – this requires that the size of the hash table is around the
    same as the amount of elements stored in the table
When evaluating the average-case running-time we assumed that the hash-function hashes evenly. However, it is in no way obvious that this actually happens.

The quality of the hash function is the most critical factor in the efficiency of the hash table.

Properties of a good hash function:
- the hash function must be deterministic
  - otherwise an element once placed into the hash table may never be found!
- despite this, it would be good that the hash function is as "random" as possible
  - $\tfrac{1}{m}$ of the keys should be hashed into each slot as closely as possible
- unfortunately implementing a completely evenly hashing hash function is most of the time impossible
  - the probability distribution of the keys is not usually known
  - the data is usually not evenly distributed

Almost any sensible hash function hashes an evenly distributed data perfectly
- Often the hash function is created so that it is independent of any patterns occurring in the input data, i.e. such patterns are broken by the function
  - for example, single letters are not investigated when hashing names but all the bits in the name are taken into account
- two methods for creating hash functions that usually behave well are introduced here
- lets assume that the keys are natural numbers $0, 1, 2, \ldots$
  - if this is not the case the key can be interpreted as a natural number
  - e.g. a name can be converted into a number by calculating the ASCII-values of the letters and adding them together with appropriate weights

Creating hash functions with the division method is simple and fast.
- $h(k) = k \mod m$
- it should only be used if the value of $m$ is suitable
- e.g. if $m = 2^b$ for some $b \in N = \{0, 1, 2, \ldots\}$, then
  $h(k) = k'$s $b$ lowest bits
  $\Rightarrow$ the function doesn't even take a look at all the bits in $k$
- the function probably hashes binary keys poorly
  - for the same reason, values of $m$ in the format $m = 10^b$ should be avoided with decimal keys
- if the keys have been formed by interpreting a character string as a value in the 128-system, then $m = 127$ is a poor choice, as then all the permutations of the same string end up into the same slot
- prime numbers are usually good choices for $m$, provided they are not close to a power of two
  - e.g. $\approx 700$ lists is needed $\Rightarrow$ 701 is OK

The multiplication method for creating hash functions doesn't have large requirements for the values of $m$.
- the constant $A$ is chosen so that $0 < A < 1$
- $h(k) = [m(kA - \lfloor kA \rfloor)]$
- if $m = 2^w$, the word length of the machine is $w$, and $k$ and $2^w \cdot A$ fit into a single word, then $h(k)$ can be calculated easily as follows:
  $h(k) = \left\lfloor \frac{(2^w \cdot A) \cdot k \mod 2^w}{2^w - b} \right\rfloor$
- which value should be chosen for $A$?
  - all of the values of $A$ work at least somehow
  - the rumor has it that $A \approx \frac{\sqrt{5} - 1}{2}$ often works quite well
7 Trees
This chapter discusses some commonly used tree structures.
- first an ordinary binary search tree is covered and then it is balanced by converting it into a red-black tree.
- the B-tree is introduced as an example of a tree whose nodes can have more than two children
- Trie is introduced

7.1 Basic binary search tree
A binary tree is a finite structure of nodes that is either
- empty, or
- contains a node called the root, and two binary trees called the left subtree and the right subtree
- childless nodes are called leaves
- other nodes are internal nodes
- a node is the parent of its children
- the ancestors of a node are the node itself, its parent, the parents parent etc.
- a descendant is defined similarly

Additionally, all elements in a binary search tree satisfy the following property:
Let l be any node in the left subtree of the node x and r be any node in the right subtree of x then
\[ \text{l.key} \leq x.\text{key} \leq \text{r.key} \]
- the binary tree in the picture on the previous page is a binary search tree
- the heap described in chapter 3.1 is a binary tree but not a binary search tree

Usually a binary tree is represented by a linked structure where each object contains the fields key, left, right and p (parent).
- there may naturally be satellite data

The nodes of the binary tree can be handled in different orders.
- preorder
  - call preorder-tree-walk(T, root)
  - the nodes of the example are handled in the following order: 18, 13, 8, 5, 3, 6, 9, 15, 14, 25, 22, 23, 30, 26, 33, 32, 35
  ```c
  preorder-tree-walk(x)  
  1 if x != NIL then  
  2 process the element x  
  3 preorder-tree-walk(x->left)  
  4 preorder-tree-walk(x->right)  
  ```
- inorder
  - in the example 3, 5, 6, 8, 9, 13, 14, 15, 18, 22, 23, 25, 26, 30, 32, 33, 35
  - the binary search tree is handled in the ascending order of the keys when processing the elements with inorder
INORDER-TREE-WALK(x)
1 \textbf{if } x \neq \text{NIL} \textbf{ then}
2 \quad \text{INORDER-TREE-WALK}(x \rightarrow \text{left})
3 \quad \text{process the element } x
4 \quad \text{INORDER-TREE-WALK}(x \rightarrow \text{right})

• postorder
  - in the example 3, 6, 5, 9, 8, 14, 15, 13, 22, 26, 32, 35, 33, 30, 25, 18

POSTORDER-TREE-WALK(x)
1 \textbf{if } x \neq \text{NIL} \textbf{ then}
2 \quad \text{POSTORDER-TREE-WALK}(x \rightarrow \text{left})
3 \quad \text{POSTORDER-TREE-WALK}(x \rightarrow \text{right})
4 \quad \text{process the element } x

• running-time $\Theta(n)$
• extra memory consumption = $\Theta(\text{maximum recursion depth})$
  = $\Theta(h + 1) = \Theta(h)$

Searching in a binary search tree

• search in the entire tree \textbf{R-TREE-SEARCH}(T.\text{root}, k)
• returns a pointer $x$ to the node with $x \rightarrow \text{key} = k$, or \text{NIL} if there is no such node

\textbf{R-TREE-SEARCH}(x, k)
1 \textbf{if } x = \text{NIL} \textbf{ or } k = x \rightarrow \text{key} \textbf{ then}
2 \quad \text{return } x \quad \text{(the searched key is found)}
3 \quad \text{if } k < x \rightarrow \text{key} \textbf{ then}
4 \quad \quad \text{return } \text{R-TREE-SEARCH}(x \rightarrow \text{left}, k) \quad \text{(...search from the left subtree)}
5 \quad \text{else}
6 \quad \quad \text{return } \text{R-TREE-SEARCH}(x \rightarrow \text{right}, k) \quad \text{(...search from the right subtree)}

The algorithm traces a path from the root downward. In the worst-case the path is down to the leaf at the end of the longest possible path.

• running-time $O(h)$, where $h$ is the height of the tree
• extra memory requirement $O(h)$ due to recursion

Minimum and maximum:
• the minimum is found by stepping as far to the left as possible

\textbf{TREE-MINIMUM}(x)
1 \textbf{while } x \rightarrow \text{left} \neq \text{NIL} \textbf{ do}
2 \quad x := x \rightarrow \text{left}
3 \text{return } x

• the maximum is found similarly by stepping as far to the right as possible

\textbf{TREE-MAXIMUM}(x)
1 \textbf{while } x \rightarrow \text{right} \neq \text{NIL} \textbf{ do}
2 \quad x := x \rightarrow \text{right}
3 \text{return } x

• the running-time is $O(h)$ in both cases and extra memory requirement is $\Theta(1)$
The structure of the tree can be used to our advantage in finding the successor and the predecessor of the node instead of using comparisons.

- this way SCAN-ALL works correctly even if the elements in the tree all have the same key
  Ξ an algorithm that finds the node which is the next largest node in the inorder from the node given
- one can be build with the TREE-MINIMUM algorithm
- the successor of the node is either
  – the smallest element in the right subtree
  – or the first element on the path to the root whose left subtree contains the given node
- if such nodes cannot be found, the successor is the last node in the tree

```plaintext
TREE-SUCCESSOR(x)
1 if x→right ≠ NIL then
2     return TREE-MINIMUM(x→right) (if there is a right subtree...)
3     y := x→p
4     while y ≠ NIL and x = y→right do
5         x := y
6         y := y→p
7     return y (return the found node)
```

- note, the keys of the nodes are not checked!
- cmp. finding the successor from a sorted list
- running-time $O(h)$, extra memory requirement $\Theta(1)$
- TREE-PREDECESSOR can be implemented similarly

TREE-SUCCESSOR and TREE-MINIMUM can be used to scan the tree in inorder

### Insertion into a binary search tree:

```plaintext
TREE-INSERT(T, z) (starts to a structure allocated by the user)
1     y := NIL; x := T.root
2     while x ≠ NIL do
3         y := x
4         if z→key < x→key then
5             x := x→left
6         else
7             x := x→right
8         z→p := y (make the node we found the parent of the new node)
9     if y = NIL then
10         T.root := z (the root is the only node in the tree)
11     else if z→key < y→key then
12         y→left := z (make the new node its parent’s left...)
13     else
14         y→right := z (... or right child)
15     z→left := NIL; z→right := NIL
```

The algorithm traces a path from the root down to a leaf, a new node is always placed as a leaf.

⇒ running-time $O(h)$, extra memory requirement $\Theta(1)$
Deletion is more complicated as an internal node may also be removed:

TREE-DELETE(T, z)
if z→left = NIL or z→right = NIL then
  y := z
else
  y := TREE-SUCCESSOR(z)
if y→left ≠ NIL then
  x := y→left
else
  x := y→right
if x ≠ NIL then
  x→p := y→p
else
  T.root := x
if y→p ≠ NIL then
  if y = y→p→left then
    y→p→left := x
  else
    y→p→right := x
if y ≠ z then
  z→key := y→key
  z→satellitedata := y→satellitedata
return y

Note! It is really known on line 5 that y only has one child.
- if z has only one child y is z
- if TREE-SUCCESSOR is called on line 4, it is known that z has a right subtree, whose minimum is y
  - minimum cannot have a left child

The algorithms seems complex but all operations besides TREE-SUCCESSOR on line 4 are constant time.
⇒ running-time is $O(h)$ and the extra memory requirement $\Theta(1)$

All basic operations of a dynamic set can be implemented with a binary search tree in $O(h)$ time and with $\Theta(1)$ extra memory:
- SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR
- DELETION

How high the binary search trees usually are?
If we assume that the elements have been entered in a random order and every order is equally probable, the height of a binary search tree build solely with INSERT is on average $\Theta(lg n)$.
⇒ all operations are on average $\Theta(lg n)$

Unfortunately the result is not nearly as efficient if the keys are entered in an ascending order, as you can see in the picture.
- the height is $n - 1$, lousy!

The problem cannot be solved sensibly with randomization, if all the operations of a dynamic set need to be maintained.
The solution is keeping the tree balanced covered in the next chapter.

Binary search trees - like other data structures - can be made suitable for new tasks by adding new fields essential to the new problem into the structures.
- the basic operations also need to be updated to maintain the contents of the new fields.
- for example, adding a field to the nodes that tells the height of the subtree
  - a function that returns the element x in linear time relative to the height of the tree can be implemented
  - a function that in linear time relative to the height of the tree tells the sequence number of the element in the ascending order
  - the algorithms would be much more inefficient, linear to the amount of the nodes, without the extra fields
7.2 Red-black binary search trees

Red-black trees are balanced binary search trees.

The tree is modified when making additions and removals in order to keep the tree balanced and to make sure that searching is never inefficient, not even when the elements are added to the tree in an inconvenient order.

- A red-black tree can never be reduced to a list like with unbalanced binary search trees.

The basic idea of red-black trees:
- Each node contains one extra bit: its colour
  - Either red or black
- The other fields are the good old key, left, right and p
  - We're going to leave the satellite data out in order to keep the main points clear and to make sure they are not lost behind details

The colour fields are used to maintain the red-black invariant, which guarantees that the height of the tree is kept in $O(\lg n)$.

The invariant of red-black trees:
1. If the node is red, it either has
   - No children, or
   - It has two children, both of them black.
2. For each node, all paths from a node down to descendant leaves contain the same number of black nodes.
3. The root is black.

The black-height of the node $x$, $bh(x)$, is the amount of black nodes on the path from it down to the node with 1 or 0 children.

- By property 2 in the invariant, the black height of each node is well defined and unique
- All alternate paths from the root downwards have the same amount of black nodes.

The black height of the tree is the black height of its root.

The maximum height of a red-black tree:
- Denote the height $h$ and the amount of nodes $n$
- At least half the nodes on any path from the root down to a leaf ($\lfloor h/2 \rfloor + 1$) are black (properties 1 and 3 in the invariant)
- There is the same amount of black nodes on each path from the root down to a leaf (property 2 of the invariant)
  $\Rightarrow$ At least $\lfloor h/2 \rfloor + 1$ upmost levels are full
  $\Rightarrow$ $n \geq 2^{\lfloor h/2 \rfloor}$
  $\Rightarrow$ $h \leq 2 \lg n$

Therefore, the invariant does guarantee that the height of the tree is kept logarithmic to the number of the nodes in the tree.

$\Rightarrow$ The operations of a dynamic set SEARCH, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR can be made in $O(\lg n)$ time with red-black trees.

- The operations work with binary search trees in $O(h)$ time,
- And a red-black tree is a binary search tree with $h = \Theta(\lg n)$

The same addition and deletion algorithms from the binary search trees cannot be used with red-black trees as they might break the invariant.

Let's use the algorithms RB-INSERT and RB-DELETE instead.

- Operations RB-INSERT and RB-DELETE are based on rotations
- There are two rotations: to the left and to the right
- They modify the structure of the tree so that the basic properties of the binary search trees are maintained for each node.

- Rotation to the left
  - Assumes that the nodes $x$ and $y$ exist
- Right rotate similarly
  - Left and right have switched places

\[\begin{array}{c}
A & B & C \\
\end{array}\]

\[\begin{array}{c}
A & B & C \\
\end{array}\]
LEFT-ROTATE(T, x)
1 y := x->right; x->right := y->left
2 if y->left ≠ NIL then
3 y->left->p := x
4 y->p := x->p
5 if x->p = NIL then
6 T.root := y
7 else if y = x->p->left then
8 x->p->left := y
9 else
10 x->p->right := y
11 y->left := x; x->p := y

- the running-time of both is $Θ(1)$
- only the pointers are modified

The basic idea of the addition
- first a new node is added in the same way as into a ordinary binary search tree
- then it is coloured red

RB-INSERT(T, x)
1 TREE-INSERT(T, x)
2 x->colour := RED
3 while x ≠ T.root and x->p->colour = RED do
4 if x->p = x->p->p->left then
5 y := x->p->p->right
6 if y ≠ NIL and y->colour = RED then (move the violation upwards)
7 x->p->colour := BLACK
8 y->colour := BLACK
9 x->p->p->colour := RED
10 x := x->p->p
11 else (moving isn't possible -- fix the violation)
12 if x = x->p->right then
13 x := x->p->LEFT-ROTATE(T, x)
14 x->p->colour := BLACK
15 x->p->p->colour := RED
16 RIGHT-ROTATE(T, x->p->p)
17 else same as lines 5 . . . 16 except "left" and "right" have switched places (color the root black)

Moving the violation of property 1 upwards:
- the node $x$ and its parent are both red
- also the node $x$'s uncle/aunt is red and the grandparent is black.

⇒ the violation is moved upwards by coloring both the uncle of $x$ and the parent black and the grandparent red.

After the fixup:
- the property 1 may still be broken
  - the node $x$ and its parent parent may both be red
- property 2 isn't broken
  - the number of black nodes on each path stays the same
- the property 3 may be violated
  - if we’ve reached the root, it may have been colored red

- which basic properties of red-black trees could be violated if the addition is done this way?
  - 1 is broken by the node added if its parent is red, otherwise it cannot be broken
  - 2 doesn’t get broken since the amounts and the locations of the black node beneath any node don’t change, and there are no nodes beneath the new node
  - 3 gets broken if the tree was originally empty
- let’s fix the tree in the following way:
  - without ruining the property number 2 move the violation of property 1 upwards until it disappears
  - Finally fix property 3 by coloring the root black (this cannot break properties 1 and 2)

- violation of 1 = both the node and its parent are red
- moving is done by coloring nodes and making rotations
If there is no red uncle available, the violation cannot be moved upwards. A more complicated approach needs to be used instead:

• Make sure that \( x \) is the right child of its parent by making a rotation to the left

• then, colour the parent of \( x \) black and the grandparent red and make a rotation to the right

– the grandparent is black for sure since otherwise there would have been two successive red nodes in the tree before the addition

After the fixup:

• there no longer are successive red nodes in the tree

• the fixup operations together don’t break the property 2

⇒ the tree is complete and execution the fixup algorithm can be stopped

General characteristics of the deletion algorithm

• first the node is deleted like from an ordinary binary search tree

  – \( w \) points to the node deleted

• if \( w \) was red or the tree was made entirely empty, the red-black properties are maintained

  ⇒ nothing else needs to be done

• otherwise the tree is fixed with RB-DELETE-FIXUP starting from the possible child of \( w \) and its parent \( w \rightarrow p \)

  ⇒ TREE-DELETE ensures that \( w \) had atmost one child

 RB-DELETE(T, z)
 1 \( w := \) TREE-DELETE(T, z)
 2 if \( w \rightarrow colour = \) BLACK and \( T \cdot root \neq \) NIL then
 3 if \( w \rightarrow left \neq \) NIL then
 4 \( x := w \rightarrow left \)
 5 else
 6 \( x := w \rightarrow right \)
 7 RB-DELETE-FIXUP(T, x, w \rightarrow p)
 8 return \( w \)

 RB-DELETE-FIXUP(T, x, y)
 1 while \( x \neq T \cdot root \) and (\( x = \) Nil or \( x \rightarrow colour = \) BLACK) do
 2 if \( x \rightarrow y \rightarrow left \) then
 3 \( w := y \rightarrow right \)
 4 if \( w \rightarrow colour = \) RED then
 5 \( w \rightarrow colour := \) BLACK; \( y \rightarrow colour := \) RED
 6 LEFT-ROTATE(T, y); \( w := y \rightarrow right \)
 7 if \( (w \rightarrow left = \) NIL or \( w \rightarrow left \rightarrow colour = \) BLACK) and
 8 \( (w \rightarrow right = \) NIL or \( w \rightarrow right \rightarrow colour = \) BLACK) then
 9 \( w \rightarrow colour := \) RED; \( x := y \)
 10 else
 11 if \( w \rightarrow right = \) NIL or \( w \rightarrow right \rightarrow colour = \) BLACK then
 12 \( w \rightarrow left \rightarrow colour := \) BLACK
 13 \( w \rightarrow colour := \) RED
 14 RIGHT-ROTATE(T, w); \( w := y \rightarrow right \)
 15 \( w \rightarrow colour := y \rightarrow colour; y \rightarrow colour := \) BLACK
 16 \( x := T \cdot root \)
 17 else
 18 y := y \rightarrow p
 19 \( x \rightarrow colour := \) BLACK

... same as lines 3...16 except “left” and “right” have switched places
7.3 B-trees

B-trees are rapidly branching search trees that are designed for storing large dynamic sets on a disk

- the goal is to keep the number of search/write operations as small as possible
- all leaves have the same depth
- one node fills one disk unit as closely as possible

⇒ B-tree often branches rapidly: each node has tens, hundreds or thousands of children
⇒ B-trees are very shallow in practice
- the tree is kept balanced by alternating the amount of the node’s children between $t, \ldots, 2t$ for some $t \in \mathbb{N}$, $t \geq 2$
  - each internal node except the root always has at least $\frac{1}{2}$ children from the maximum amount

The picture shows how the keys of a B-tree divide the search area.

Searching in a B-tree is done in the same way as in an ordinary binary search tree.

- travel from the root towards the leaves
- in each node, choose the branch where the searched element must be in - there are just much more branches

Inserting an element into a B-tree

- travel from the root to a leaf and on the way split each full node into half
  ⇒ when a node is reached, its parent is not full
- the new key is added into a leaf
- if the root is split, a new root is created and the halves of the old root are made the children of the new root
  ⇒ B-tree gets more height only by splitting roots
- a single pass down the tree is needed and no passes upwards

A node in a B-tree is split by making room for one more key in the parent and the median key in the node is then lifted into the parent.

The rest of the keys are split around the median key into a node with the smaller keys and a node with the larger keys.
Deleting a key from a B-tree is a similar operation to the addition.

- travel from the root to a leaf and always before entering a node make sure there is at least the mimimum amount + 1 keys in it
  - this guarantees that the amount of the keys is kept legal although one is removed
- once the searched key is found, it is deleted and if necessary the node in combined with either of its siblings
  - this can be done for sure, since the parent node has at least one extra key
- if the root of the end result has only one child, the root is removed and the child is turned into the new root

7.4 Trie

A trie is a data structure better suited for storing character strings than the binary search tree.

The nodes of a trie are characters

- the character string the node represents is the string its parent represents + the character in the node itself
- the string the node represents is the prefix of the strings each of its children represent

In addition, each node contains one bit that tells whether the string represented by the node is in the set or not.

The alphabet of the example trie is \{a, b\}.

- since the truth value in the root is true, the empty string \(\varepsilon\) it represents is in the set
- plain a and b don’t belong into the set, since they’ve been marked false
- the strings aa, aba and ba belong into the set

\[
\text{TRIE-SEARCH}(p, A, n)
\]

1. \(\text{for } i := 1 \text{ to } n \text{ do} \) (scan through the characters of the word)
2. \(\text{if } p \rightarrow C[A[i]] = \text{NIL} \text{ then} \) (if the chosen branch ends...)
3. \(\text{return FALSE} \) (...the word is not found)
4. \(p := p \rightarrow C[A[i]] \) (move downwards)
5. \(\text{return } p \rightarrow \text{bit} \) (return the truth value of the found node)

The searched string is in an array \(A[1, \cdots, n]\) and \(p\) points to the root of the tree

The nodes contain an array \(C\), where there is a pointer \(C[a]\) for each possible character \(a\) and a bitfield \(\text{bit}\), that indicates whether the string is in the set or not.
The search travels down the tree until

- a NIL-pointer is met, and the word is not in the tree
- the word ends and the value of the bit-field is returned

**Trie-Insert**\((p, A, n)\)

1. \(i := 1\) to \(n\) do (go through the characters of the added word)
2. \(\text{if } p \rightarrow C[A[i]] = \text{NIL}\) then (add a new node if necessary)
   1. allocate a new node object and initialize each pointer \(new \rightarrow C[i]\) to NIL
3. \(p := p \rightarrow C[A[i]]\) (move downwards)
4. \(p \rightarrow \text{bit} := \text{TRUE}\) (set the truth value of the last node to TRUE)

When inserting a new node the tree is travelled downwards towards the leaf nodes according to the characters in the string.

- if a NIL-pointer is met a new node is created
- once the end of the word is reached the bit-field of the node is set to TRUE to indicate that the word belongs into the set

The operations of a dynamic set are very efficient with a trie.

\(⇒\) Regardless of the amount of the character strings stored, the operations are almost linear-time relative to the length of the string.

The downside is the large memory consumption if the number of possible characters is large.

- on the other hand, if the strings stored have long common prefixes, space is saved as the common parts are saved only once

The trie makes it possible to print the strings stored into it efficiently in the alphabetical order.

---

**Trie-Delete** \((p, A, n)\)

1. \(i := 1\) to \(n\) do (go through the characters of the word)
2. \(\text{if } p \rightarrow C[A[i]] = \text{NIL}\) then (if the chosen branch ends...)
   1. return \(FALSE\) (...the word is not found)
3. \(P[i] := p\) (save the parent of the node i)
4. \(p := p \rightarrow C[A[i]]\) (move downwards)
5. \(p \rightarrow \text{bit} := \text{FALSE}\) (set the truth value in the node to FALSE)
6. \(i := i - 1\)
7. \(\text{while } i > 0 \text{ and LeafNode}(p) \text{ and } p \rightarrow \text{bit} = \text{FALSE} \text{ do}\)
8. \(\text{free the element } p \text{ points to}\)
9. \(p := P[i]\) (continue freeing from the saved father)
10. \(p \rightarrow C[A[i]] := \text{NIL}\) (set the pointer to the removed node to nil)
11. \(i := i - 1\)
12. return \(TRUE\)

The string to be removed is searched from the tree and it is removed from the set by setting its bit to FALSE.

After this, the nodes made unnecessary by the removal are removed.

- since the nodes in the trie have no parent pointers, the pointers met on the way are saved in an array \(P\)

For example, the T9 text predicting text message writing system commonly used in mobile phones has been implemented with a trie structure.

- the difference to the model given here is that the characters behind the same key belong into the same node
- in addition, a probability has been determined for each letter, which is used to decide what is most likely going to be the end result of the typing
8 C++ standard library

This chapter covers the data structures and algorithms in the C++ standard library.

The emphasis is on things that make using the library purposeful and efficient.

Topics that are important from the library implementation point of view are not discussed here.

8.1 General information on the C++ standard library

The standard library, commonly called STL, was standardized together with the C++-language autumn 1998.

It contains the most important basic data structures and algorithms.

- most of the data structures and algorithms covered earlier in this material in one shape or another
- an odd exception: the hash table is missing from the standard
  - it is however available in some implementations
  - can be found in the C++ standards committee’s library extension report TR1 under the name unordered map/set

It also contains a lot more such as

- input / output: cin, cout, ...
- processing character strings
- minimum, maximum

- search and modifying operations of queues
- support for functional programming
- complex numbers
- arithmetic functions (e.g. sin, log10),
- vector arithmetics and support for matrix operations

The interfaces are carefully thought out, flexible, generic and type safe.

The efficiency of the operations the interfaces provide has been given with the $O$-notation.

The compile time C++ template mechanism has been used to implement the genericity.

- the loss in efficiency due to the interface is not significant
- error messages are long and difficult to read with the current compilers

Standard library suffers from the problems of C++ to some extent:

- type checking is incomplete
  - it is easy to get into a habit of implementing compiler dependent code

High-quality implementations are available for free.

- e.g. g++
  - is and will be used widely

Interesting elements in the standard library from the point of view of a data structures course are the containers, i.e. the data structures provided by the library and the generic algorithms together with the iterators, with which the elements of the containers are manipulated.
8.2 Iterators
We see all standard library data structures as black boxes with a lot of common characteristics. The only thing we know is that they contain the elements we’ve stored in them and that they implement a certain set of interface functions.

We can only handle the contents of the containers through the interface functions and with iterators.

Iterators are handles or "bookmarks" to the elements in the data structure.

- each iterator points to the beginning or the end of the data structure or between two elements.
- the element on the right side of the iterator can be accessed through it, except if the iterator in question is a reverse iterator which is used to access the element on the left side.
- the operations of moving the reverse iterator work in reverse, for example ++ moves the iterator one step to the left.
- the interface of the containers usually contains the functions begin() and end(), that return the iterators pointing to the beginning and the end of the container.
- functions rbegin() and rend() return the equivalent reverse iterators.
- an iterator can be used to iterate over the elements of the container, as the name suggests.
- an iterator can be used for reading and writing.
- the location of the elements added to or removed from a container is usually indicated with iterators.

Each container has its own iterator type.

- different containers provide different possibilities of moving the iterator from one place to another efficiently (cmp. reading an arbitrary element from a list/array)
- the design principle has been that all iterator operations must be performed in constant time to guarantee that the generic algorithms work with the promised efficiency regardless of the iterator given to them.
- iterators can be divided into categories based on which constant time operations they are able to provide.

An input iterator can only read element values but not change them.

- the value of the element the iterator points to can be read (*p)
- a field of the element the iterator points to can be read or its member function can be called (p->)
- the iterator can be moved one step forwards (++p or p++)
- iterators can be assigned to and compared with each other (p=q, p==q, p!=q)

An output iterator is like an input iterator but it can be used only to change the elements (*p=x)
A **forward iterator** is a combination of the interfaces of the input- and output iterators.

A **bidirectional iterator** is able to move one step at a time backwards (\(\text{p}--\text{p}\))

A **random access iterator** is like a bidirectional iterator but it can be moved an arbitrary amount of steps forwards or backwards.

- the iterator can be moved \(n\) steps forwards or backwards (\(\text{p}+=n, \text{p}==\text{p}+n, \text{q}==\text{p}+n\))
- an element \(n\) steps from the iterator can be read and it can be modified (\(\text{p}[n]\))
- the distance between two iterators can be determined (\(\text{p}-\text{q}\))
- the difference of two iterators can be compared, an iterator is "smaller" than the other if it's location is earlier than the other in the container (\(\text{p}<\text{q}, \text{p}<=\text{q}, \text{p}==\text{q}, \text{p}>=\text{q}\))

In addition to the ordinary iterators STL provides a set of iterator adapters.

- they can be used to modify the functionality of the generic algorithms
- the reverse iterators mentioned earlier are iterator adapters
- **insert iterators/inserters** are one of the most important iterator adapters.
  - they are output iterators that insert elements to the desired location instead of copying
  - an iterator that adds to the beginning of the container is given by the function call `front_inserter(container)`
  - an iterator that adds to the end is given by `back_inserter(container)`
  - an iterator that adds to the given location is given by `inserter(container, location)`

The syntax of the iterator operations is obviously similar to the pointer arithmetic of C++.

Iterators can be used with `#include <iterator>`

An iterator of a correct type can be created with the following syntax for example.

```cpp
container::stored::iterator p;
```

The additions and removals made to the containers may **invalidate** the iterators already pointing to the container.

- this feature is container specific and the details of it are covered with the containers

- **stream iterators** are input and output iterators that use the C++ streams instead of containers
  - the syntax for an iterator that reads from a stream with the type `cin` is `istream_iterator<type>(cin)`
  - the syntax for an iterator that prints the given data to the cout stream data separated with a comma is `ostream_iterator<type>(cout, ',')`
8.3 Containers

The standard library containers are mainly divided into two categories based on their interfaces:

- sequences
  - elements can be searched based on their number in the container
  - elements can be added and removed at the desired location
  - elements can be scanned based on their order in the container

- associative containers
  - elements are placed in the container to the location determined by their key
  - by default the operator < can be used to compare the key values of the elements stored (unordered map/set in TR1 is an exception)

Containers are passed by value.
- the container takes a copy of the data stored in it
- the container returns copies of the data it contains
  \[\Rightarrow\] changes made outside the container do not effect the data stored in the container
- all elements stored into the containers must implement a copy constructor and an assignment operator.
  - basic types have one automatically
  - elements of a type defined by the user should be stored with a pointer pointing to them
  - this is sensible from the efficiency point of view anyway
  - the shared_ptr available in TR1 and Boost is a handy tool for making the memory management easier
    - contains an inbuilt reference counter and deletes the element once the counter becomes zero
    - there is no need to call delete. Furthermore is must not be called.

Sequence containers:

Vector declaration:

\[\text{Vector } \text{vector}<\text{type}>\]

- vector is a flexible sized array which provides the same functionality as an ordinary array and a lot more.
- vector should rather be indexed with \texttt{.at(...) \_} than with \texttt{[ ... ]}, since then overindexing causes an exception
- legal indices are 0, \ldots, size – 1
- the size can be increased with the function \texttt{.resize(size, initial value)}
  - initial value is optional
  - the vector automatically allocates new memory when needed
  - memory can also be allocated in advance:
    - \texttt{.reserve(size), .capacity()}
In practise, the vector has been implemented as a normal array whose size is incremented in the following way:

- when there is no more room for the new element in the array, a new one, twice the size of the old one is allocated and the old elements are moved to it
- the array doesn’t ever get smaller
  ⇒ the memory allocation is not reduced except by copying a completely new content to the container
  ⇒ Adding an element to the end of the vector is amortized constant time
- the performance is analyzed as a unit, the efficiency of a sequence of operations is investigated instead of single operations
- each addition operation that requires expensive memory allocation is preceded by an amount of inexpensive additions relative to the price of the expensive operation
- the cost of the expensive operation can be equally divided to the inexpensive operations

- the savings can be used to pay for the expensive operation
  ⇒ all additions to the end of the vector are amortized constant time

This can be proven with the accounting method:

- charge three credits for each addition
- one credit is used for the real costs of the addition
- one credit is saved with the element added to \( i \)
- one credit is saved at the element at \( i - \frac{1}{2} \cdot \text{vector.capacity()} \)
- when the size of the array needs to be increased, each element has one credit saved and the expensive copying can be paid for with them

- now the inexpensive operations are still constant time, although they are a constant coefficient more inefficient than in reality

- adding an element elsewhere with \( \text{insert} \) takes \( O(n) \) time although addition to the end with \( \text{push_back} \) is amortized \( O(1) \)
- the running-time of the removal operation \( \text{erase} \) is linear and the running time of \( \text{pop_back} \) is constant time
- iterators are invalidated in the following situations
  - if the vector originally didn’t have enough space reserved for it any addition can cause the invalidation of all iterators
  - the removals only invalidate those iterators that point to the elements after the removal point
  - additions to the middle always invalidate the iterators after the addition point
- a special implementation has been given to \( \text{vector<bool>} \) which is different from the general implementation the templates would create
  - the goal: makes 1 bit / element possible where the ordinary implementation would probably use 1 byte /
element i.e. 8 bits / element

Deque `deque<type>`
- declaration: `#include <deque>`
- provides similar services and almost the same interface as `vector`
- in addition, provides an efficient \((O(1)\) amortized running-time) of addition and removal at both ends
  - `push_front(element)`
  - `pop_front()`
- a likely implementation: a group of memory cells accessed through a ring buffer
- iterators are invalidated in the following situations
  - all additions can invalidate the iterators
  - removals at the middle invalidate all iterators
  - all addition and removal operations elsewhere except at the ends can invalidate references and pointers

List `list<type>`
- `#include <list>`
- implementation is probably a doubly linked list
  - addition and removal is everywhere constant time, there is no indexing operation
- addition and removal don’t invalidate the iterators or references (except naturally to elements removed)
- list provides several special services
  - `.splice(location, other_list)`: makes the other list a part of the second list in front of the location
  - `.splice(location, list, element)`: moves an element from another list or the same list in front of the location
  - `.splice(location, list, first, end)`
  - `.own_merge(other_list) and .sort()`
  - `.reverse()`

Associative containers:
- `set<type>` and `multiset<type>`
  - `#include <set>`
  - dynamic set where
    - elements can be searched, added and deleted in logarithmic time
    - elements can be scanned in ascending order in amortized constant time so that the scan from the beginning to the end is always a linear time operation
  - the same element can be in the multiset several times, in the set the elements are unique
  - an order of size must be defined for the elements “<”
    - can be implemented separately as a part of the type or as a parameter of the constructor
  - determines the equality with \(\neg (x < y \lor y < x)\)
    - a sensible and efficient definition to “<” needs to be given
the likely implementation is a red-black tree
changing the value of the element directly has been prohibited
- the old element needs to be removed and a new one added instead
interesting operations:
- \texttt{.find} (element) finds the element (first of many in the multiset) or returns \texttt{}. \texttt{end()} if it isn’t found
- \texttt{.lower_bound} (element) finds the first element > element
- \texttt{.upper_bound} (element) finds the first > element
- \texttt{.equal_range} (element) returns \texttt{make_pair(}
  \texttt{.lower_bound} (element), \texttt{.upper_bound} (element) \texttt{)}, but needs only one search (the size of the range is 0 or 1 for the set)
- for sets \texttt{insert} returns a pair (location, added), since elements already in the set may not be added
the standard guarantees that the iterators are not invalidated by the addition or removal (except of course to the elements removed)

Unordered set/map/multiset/multimap

\begin{itemize}
  \item NOTE! not (yet) available in the standard but is in the draft TR1
  \item NOTE! all compilers don’t implement this interface
  \item the interfaces of unordered-map/set resemble map and set
  \item the most significant differences:
    \begin{itemize}
      \item only single \emph{bucket} of the hash table can be iterated through
      \item addition, removal and searching is on average constant time and in the worst-case linear
      \item the size of the hash table is automatically increased in order to keep the load factor of the buckets under a certain limit
      \item changing the size of the hash table (rehashing) is on average linear, worst-case quadratic
      \item rehashing invalidates all iterators but not pointers or references
    \end{itemize}
\end{itemize}

Additionally other containers are found in the Standard library:

\begin{itemize}
  \item \texttt{bitset<bit_amount>}
    \begin{itemize}
      \item \#include \texttt{<bitset>}
      \item for handling fixed sized binary bitsets
      \item provides typical operations (AND, OR, XOR, NOT)
    \end{itemize}
  \item \texttt{string}
    \begin{itemize}
      \item \#include \texttt{<string>}
      \item the character string has been optimized for other purposes in C++ and they are usually not perceived as containers. However, they are, in addition to other purposes, also containers.
      \item store characters but can be made to store other things also
      \item they provide, among others, iterators, \texttt{[...].at(...)}, \texttt{.size()}, \texttt{.capacity()} and \texttt{swap}
      \item string can get very large and automatically allocate new memory when necessary
    \end{itemize}
\end{itemize}

\begin{verbatim}
map<key_type, element_type> and
multimap<key_type, element_type>
\end{verbatim}

\begin{itemize}
  \item \#include \texttt{<map>}
  \item without satellite data would be almost like (multi)sets
  \item store (key, satellite data) pairs
    \begin{itemize}
      \item the type of the pair is \texttt{pair<type1, type2>}
      \item a pair can be created with the function \texttt{make_pair}
      \item the fields of the pair are returned by \texttt{.first()}, \texttt{.second()}
    \end{itemize}
  \item \texttt{map} can be exceptionally indexed with the key (indexing is not constant time)
  \item the iterators are not invalidated by the addition or removal
\end{itemize}
• Care should be taken with the modifying operations of strings (catenation, removal) since they allocate memory and copy elements, which makes them heavy operations for long strings
• it is often sensible anyway to store the strings with a pointer when for example storing them into the containers to avoid needless copying
• for the same reason strings should be passed by reference

In addition to containers, STL provides a group of container adapters, that are not containers themselves but that can be "adapted into a new form" through the interface of the container:

Stack `stack<element_type, container_type>`
• `#include <stack>`
• provides in addition to the normal class-operations only
  – stack-operations, `.push()`, `.pop()`, `.top()`
  – size queries, `.size()` and `.empty()`
  – comparisons `==`, `<` etc.
• `.pop()` doesn’t return anything, the topmost element is evaluated with `.top()`
• the topmost element of the stack can be changed in place:
  `stack.top() = 35;`
• what’s interesting from the course’s point of view is that the user can choose an implementation either based on an array or a list
  – `stack<type> array_stack;`
  – `stack<type, list<type>> list_stack;`

Queue `queue<element_type, container_type>`
• `#include <queue>`
• queue operations, `.push()`, `.pop()`, `.front()`, `.back()`()
• otherwise more or less like the stack

Priority queue `priority_queue<element_type, container_type>`
• `#include <queue>`
• has almost an identical interface as the queue
• elements have a different order: `.top()` returns the largest
• returns any of the equal elements
• the topmost element cannot be changed with `.top()` in place
• like with associative containers, the sorting criterion can be given as a parameter to `<>` as the constructor parameter
• implemented with a stack
• unlike with the others, the container used as the implementation of the priority queue must support random indexing
  – default implementation vector

<table>
<thead>
<tr>
<th>data-structure</th>
<th>addition in the end</th>
<th>addition elsewhere</th>
<th>removing 1st elem.</th>
<th>removing others</th>
<th>search the largest</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>list</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>deque</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>stack (queue)</td>
<td>O(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>priority</td>
<td>O(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue (priority)</td>
<td>O(log n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set (multiset)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>map (multimap)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

[1] constant-time with the last element, linear otherwise
[2] logarithmic if the container is sorted, else linear
[3] constant time if the data structure is sorted, else linear
[4] constant time with the first element, else linear
[5] possible only with the last element
[6] addition possible only at the beginning
[7] removal possible only at the end
8.4 Generic algorithms

The standard library contains most of the algorithms covered so far.

All algorithms have been implemented with function templates that get all the necessary information about the containers through their parameters.

The containers are not however passed directly as a parameter to the algorithms, iterators to the containers are used instead.

- parts of the container can be handled instead of the complete container
- the algorithm can get an iterator to containers of different type which makes it possible to combine the contents of a vector and a list and store the result into a set
- the functionality of the algorithms can be changes with iterator adapters

The standard library algorithms can be used with `<algorithm>`. A short description on some of the algorithms that are interesting from the course’s point of view (in addition there are plenty of straightforward scanning algorithms and such):
Binary search

- `binary_search(first, end, value)` tells if the value is in the sorted sequence
  - `first` and `end` are iterators that indicate the beginning and the end of the search area, which is not necessarily the same as the beginning and the end of the data structure
- there can be several successive elements with the same value
  ⇒ `lower_bound` and `upper_bound` return the limits of the area where the value is found
  - the lower bound is and the upper bound isn’t in the area
- the limits can be combined into a pair with one search: `equal_range`
- cmp. `Bin-Search` page 83

nth_element` (first, nth, end)

- finds the element that would be at index `nth` in a sorted container
- resembles `RANDOMIZED-SELECT`
- iterators must be random-access

Partition

- `partition(first, end, condition)` unstable
- `stable_partition(first, end, condition)` stable but slower and/or reserves more memory
- sorts the elements in the range `first-end` so that the elements for which the `condition`-function returns true come first and then the ones for which `condition` is false.
- cmp. `Quick-Sort’s PARTITION`
- the efficiency of partition is linear
- the iterators must be bidirectional

Sorting algorithms

- `sort(beg, end)` and `stable_sort(beg, end)`
  - the running-time of `sort`
    - on average `O(n log n)`
    - at most `O(n^2)`
    - the worst-case is also `O(n log n)` in the SGI implementation
  - characteristics of `stable_sort`
    - running-time `O(n log n^2)`
    - stable
  - the sorting algorithms require random access iterators as parameter
    ⇒ cannot be used with lists, but list provides a `sort` of its own (and an non-copying `merge`) as a member function
  - there is also a sort that ends once a desired amount of the first elements are sorted: `partial_sort(beg, middle, end)`

merge` (beg1, end1, beg2, end2, target)

- The algorithm merges the elements in the ranges `beg1-end1` and `beg2-end2` and copies them in an ascending order to the end of the iterator `target`
- the algorithm requires that the elements in the two ranges are sorted
- cmp. `MERGE` on page 55
- the algorithm is linear
- `beg-` and `end-`iterators are input iterators and `target` is an output iterator

Heaps

- Heap algorithms equivalent to those described in chapter 3.1. can be found in STL
- `push_heap(first, end) HEAP-INSERT`
Set operations

- The C++ standard library contains functions that support this
- \texttt{std::set}\_\texttt{set}\_\texttt{include}\_\texttt{first}, \texttt{end}, \texttt{first2}, \texttt{end2}) subset \subseteq
- \texttt{std::set}\_\texttt{set}\_\texttt{union}(\texttt{first}, \texttt{end}, \texttt{first2}, \texttt{end2}, \texttt{result}) union \cup
- \texttt{std::set}\_\texttt{set}\_\texttt{intersection}(\ldots) intersection \cap
- \texttt{std::set}\_\texttt{set}\_\texttt{difference}(\ldots) difference -
- \texttt{std::set}\_\texttt{set}\_\texttt{symmetric\_difference}(\ldots)
- \texttt{first-} and \texttt{end-} iterators are input iterators and \texttt{result} is an output iterator

\texttt{std::find}\_\texttt{first}\_\texttt{of}(\texttt{first1}, \texttt{end1}, \texttt{first2}, \texttt{end2})

- there is a condition in the end that limits the elements investigated
- finds the first element from the first queue that is also in the second queue
- the queue can be an array, a list, a set, 

8.5 Problems

Example 1:

Your task is to implement a data structure that stores the service requests made for the company’s helpdesk. The requests need to be handled fairly so that the first one in is dealt with first.

How must the result be changed to work so that regular customers can be handled before ordinary customers but in fifo-order inside the customer group? What if there are gold and platinum customers?
Example 2:
Students are given parallel tasks so that the student in a student list sorted according to the student number modulo 1 gets the task 1, modulo 2 the task 2 etc.
How to efficiently implement a program that can be asked if a student has registered and what is his sequence number in the list sorted based on the student number?

Example 3:
It is difficult to know when buying a house what is the correct going price on the market. To make this easier the union of real estate agents decides to start collecting up-to-date information on the prices. Each member sends in the information on a deal immediately.
How would you store the information when a graph of the month’s average price per square meter is wanted on the union’s web page? What about the median price per square meter during the current year?

Example 4:
Knowledge of the customer’s purchases is stored into the customer register of a supermarket of a large chain. The program must be able to check efficiently if for example Teemu Teekari has earlier bought frozen pizza from the market or if he is a new customer and store the information.
At the second stage the information of separate markets is stored in asciiform and sent onto the central computer of the supermarket chain each night. The central computer combines the information into one file.

9 Graph algorithms
A graph is a data structure that consists of nodes (or vertex), and edges (or arc) that link the nodes to each other.
A graph can be undirected or directed.

Graphs play a very important role in computer science.
• model pictures can often be seen as graphs
• relations between things can often be represented with graphs
• many problems can be turned into graph problems
  – search for direct and indirect prerequisites for a course
  – finding the shortest path on a map
  – determining the capacity of a road network when there are several alternate routes
### 9.1 Representation of graphs in a computer

In mathematics a graph $G$ is a pair $G = (V, E)$.

- $V$ = set of vertices
- $E$ = set of edges
- thus there can be only one edge to each direction between vertices
  - this isn’t always enough in a practical application
  - for example, there can be more than one train connection between two cities
  - this kind of graph is called a multigraph
- if only one edge to each direction between nodes is possible $\Rightarrow E \subseteq V^2$
  - for a directed graph $|E|$ can alter between $0, \ldots, |V|^2$
  - this is assumed when analyzing the efficiencies of graph algorithms

There are two standard ways to represent a graph in a computer \textit{adjacency list} and \textit{adjacency matrix}.

- the adjacency list representation is more commonly used and we concentrate on it on this course.
  - there is a linked list for each node that contains the list of the vertices to which there is an edge from the vertex
    - the order of the nodes in the adjacency list is irrelevant
  - the sum of the lengths of all the adjacency lists of the graph is $|E|$, if the graph is directed
    $\Rightarrow 2 \cdot |E|$, if the graph is undirected
  - the memory consumption of the adjacency list representation is $O(\max(V, E)) = O(V + E)$
  - the search for “is there an edge from vertex $v$ to $u$” requires a scan through one adjacency list which is $\Theta(V)$ in the worst case

- the adjacency matrix is a $|V| \times |V|$-matrix $A$, where the element $a_{ij}$ is
  - 0, if there is no edge from vertex $i$ to $j$
  - 1, if there is an edge from $i$ to $j$
- the adjacency matrices of the earlier example are

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 \\
5 & 1 & 0 & 0 & 1
\end{pmatrix}
\]

- memory consumption is always $\Theta(V^2)$
  - Each element uses only a bit of memory, to several elements can be store in one word $\Rightarrow$ the constant coefficient can be made quite small

The efficiency of a graph algorithm is usually given as a function of both $|V|$ and $|E|$

- we’re going to leave the absolute value signs out for simplicity inside the asymptotic notations, i.e. $O(V \cdot E) = O(|V| \cdot |E|)$

There are two standard ways to represent a graph in a computer \textit{adjacency list} and \textit{adjacency matrix}.

- the adjacency matrix is a $|V| \times |V|$-matrix $A$, where the element $a_{ij}$ is
  - 0, if there is no edge from vertex $i$ to $j$
  - 1, if there is an edge from $i$ to $j$
- the adjacency matrices of the earlier example are

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 \\
5 & 1 & 0 & 0 & 1
\end{pmatrix}
\]

- memory consumption is always $\Theta(V^2)$
  - Each element uses only a bit of memory, to several elements can be store in one word $\Rightarrow$ the constant coefficient can be made quite small
the adjacency matrix representation should be used with very dense graphs.

Let's analyze the implementation of the adjacency list representation a little closer:
- In practical solutions all kinds of information useful to the problem or the algorithm used is collected to the vertices:
  - name
  - a bit indicating whether the vertex has been visited
  - a pointer that indicates the vertex through which this vertex was entered
- the vertex should be implemented as a structure with the necessary member fields
- usually it's beneficial to make the vertices structures which have been linked to the vertices that lead to them
- the main principle:
  - store everything once

use pointers to be able to travel to the necessary directions

9.2 General information on graph algorithms

Terminology:
- step = moving from one vertex to another along an edge
  - the step needs to be taken to the direction of the edge in a directed graph
- the distance of the vertex \( v_2 \) from the vertex \( v_1 \) is the length of the shortest path from \( v_1 \) to \( v_2 \)
  - the distance of each vertex from itself is 0
  - denoted by \( \delta(v_1, v_2) \)
  - it is possible (and common) in a directed graph that \( \delta(v_1, v_2) \neq \delta(v_2, v_1) \)
  - if there is no path from \( v_1 \) to \( v_2 \) then \( \delta(v_1, v_2) = \infty \)

To make understanding the algorithms easier, we'll color the vertices.

- white = the vertex hasn't been discovered
- grey = the vertex has been discovered but hasn't been completely processed
- black = the has been discovered and is completely processed
- the color of the node changes from white → grey → black
- the color coding is a tool for thinking and it doesn’t need to be implemented fully. Usually it is sufficient to know whether the node has been discovered or not.
  - of this information can also be determined from other fields

Many graph algorithms go through the graph or a part of it in a certain order.
- there are two basic ways to go through the graph: breadth-first search and depth-first search
an algorithm that
- visits once all vertices in the graph or some part of it
- travels through each edge in the graph or some part of it once
is meant by “going through”

The search algorithms use some given vertex of the graph, the source, as the starting point of the search and search through all nodes that can be accessed through the source with 0 or more steps.

9.3 Breadth-first search

The breadth-first search can be used for example for:
- determining the distance of all the nodes from the source
- finding (one) shortest path from the source to each node

The breadth-first is so named as it investigates the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

The fields of the vertices:
- \( v \rightarrow d = \) if the vertex \( v \) has been discovered its distance from \( s \), else \( \infty \)
- \( v \rightarrow \pi = \) a pointer to the vertex through which \( v \) was found the first time, NIL for undiscovered vertices
- \( v \rightarrow colour = \) the color of vertex \( v \)
- \( v \rightarrow Adj = \) the set of the neighbors of \( v \)

The data structure \( Q \) used by the algorithm is a queue (follows the FIFO policy).

```
BFS(s)
  1 \( s \rightarrow colour \) := GRAY
  2 \( s \rightarrow d \) := 0
  3 \( s \rightarrow \pi \) := NIL
  4 \( Q \rightarrow s \) := 0
  5 \( i := 0 \)
  6 while \( i \neq 0 \) do
  7     \( u := \) POP(\( Q \))
  8     for each \( v \in u \rightarrow Adj \) do
  9         if \( v \rightarrow colour = \) WHITE then
 10             \( v \rightarrow colour \) := GRAY
 11             \( v \rightarrow d \) := \( u \rightarrow d \) + 1
 12             \( v \rightarrow \pi \) := \( u \)
 13             \( Q \rightarrow v \) := \( Q \rightarrow v \) + 1
 14     end for
```

The running-time in relation to the amount of vertices (\( V \)) and edges (\( E \)):
- before calling the algorithm the nodes need to be initialized
  - this can be done in \( O(V) \) in a sensible solution
- the algorithm scans the out edges of the vertex on line 7
  - can be done in linear time to the amount of the edges of the vertex with the adjacency list representation
- each queue operation is constant time
- the amount of loops in the while
  - only white vertices are pushed to the queue
  - the color of the vertex is changed into gray at the same time
  \( \Rightarrow \) each vertex can be pushed into the queue at most once
  \( \Rightarrow \) the while-loop makes at most \( O(V) \) rounds
- the amount of loops in the for

All fields of the vertex used by the algorithm are not necessarily needed in the practical implementation. Some can be determined based on each other.
the algorithm goes through each edge once into both directions
⇒ for-loop is executed at most $O(E)$ times in total
⇒ the running-time of the entire algorithm is thus $O(V + E)$

Once the algorithm has ended the $\pi$ pointers define a tree that contains the discovered vertices with the source $s$ as its root.

- breadth-first tree
- $\pi$ pointers define the edges of the tree “backwards”
  - point towards the root
  - $v \rightarrow \pi = v$'s predecessor, i.e. parent
- all nodes reachable from the source belong into the tree
- the paths in the tree are the shortest possible paths from $s$ to the discovered vertices

### Printing the shortest path
- once BFS has set the $\pi$ pointers the shortest path from the source $s$ to the vertex $v$ can be printed with:

```plaintext
PRINT-PATH(G, s, v)
1 if v = s then (base case of recursion)
2 print s
3 else if $v \rightarrow \pi = \text{Nil}$ then (the search didn't reach the vertex $v$ at all)
4 print “no path”
5 else
6 PRINT-PATH(G, s, $v \rightarrow \pi$) (recursive call . . .)
7 print $v$ (…print afterwards)
```

- A non-recursive version can be implemented for example by
  - collecting the numbers of the vertices into an array by walking through the $\pi$ pointers and printing the contents of the array backwards
  - walking through the path twice and turning the $\pi$ pointers backwards each time (the latter turning is not necessary if the $\pi$ pointers can be corrupted)

The pseudocode for the algorithm resembles greatly the breadth-first search. There are only a few significant differences:
- instead of a queue the vertices waiting to be processed are put into a stack
- the algorithm doesn’t find the shortest paths but a path

### 9.4 Depth-first search

Depth-first is the second of the two basic processing orders.

Where as the breadth-first search investigates the vertices across the entire breadth of the search frontier, the depth-first search travels one path forwards as long it’s possible.

- only vertices that haven’t been seen before are accepted into the path
- once the algorithm cannot go any further, it backtracks only as much as is needed in order to find a new route forwards
- the algorithm stops once it backtracks back to the source and there are no unexplored edges left there

The pseudocode for the algorithm resembles greatly the breadth-first search. There are only a few significant differences:
- instead of a queue the vertices waiting to be processed are put into a stack
- the algorithm doesn’t find the shortest paths but a path

#### DFS(s)

- (the algorithm gets the source $s$ as a parameter)

```plaintext
DFS(s)
1 b := in the beginning the color field of each (unprocessed) vertex is colour = WHITE
2 s := colour : GRAY (mark the source as discovered)
3 PUSH(S, s) (push the source into the stack)
4 while S ≠ \emptyset do (continue until the stack is empty)
5 v := POP(S) (pop the latest vertex added to the stack)
6 for each $v \in u \rightarrow Adj$ do (go through the neighbours of $v$)
7 if $v \rightarrow colour = \text{WHITE}$ then (if the vertex hasn’t been processed . . .)
8 $v \rightarrow colour := \text{GRAY}$ (…mark it discovered . . .)
9 PUSH(S, v) (…and push it into the stack for processing)
```
If the entire graph needs to be investigated, the depth-first search can be called for all nodes still unprocessed.

- this time the nodes are not colored white between the calls

An operation that needs to be done to all vertices in the graph could be added after line 5. We can for example
- investigate if the vertex is a goal vertex and quit if so
- store satellite data associated with the vertex
- modify the satellite data

The efficiency can be analyzed as with breadth-first search:
- before calling the algorithm the nodes need to be initialized
  - this can be done in \(O(V)\) in a sensible solution
- the algorithm scans the out edges of the vertex on line 6
  - can be done in linear time to the amount of the edges of the vertex with the adjacency list representation
- each stack operation is constant time

Coloring the vertex black may be necessary in some situations.
- this makes finding loops from the graph with DFS possible
- a recursive version is given here, since the non-recursive alternative is significantly more complex
- the algorithm can be changed into an iterative one by using a loop and simulating the recursion stack with a stack like in the previous algorithm

Note! before calling the algorithm all vertices must be initialized white!

```plaintext
DFS(u)
1 u→colour := GRAY (mark vertex as found)
2 for each v ∈ u→Adj do (go through the neighbours of u)
3   if v→colour = WHITE then (if v hasn't been visited . . . )
4     DFS(v) (...continue the search recursively from v)
5   else if v→colour = GRAY then (if has been visited but not fully processed . . . )
6     if a loop is found (...a loop is found)
7     u→colour := BLACK (mark node as fully processed)
```

Running-time:
- the recursive call is done only with white vertices
- a vertex is colored grey at the beginning of the function
  ⇒ DFS is called atmost \(O(V)\)
- like in the earlier version the for loop makes atmost two rounds per each edge of the graph during the execution of the entire algorithm
  ⇒ there are atmost \(O(E)\) rounds of the for loop
- other operations are constant time
  ⇒ the running time of the entire algorithm is still \(O(V + E)\)

Breadth-first search vs. depth-first search:
- the breadth-first search should be used for finding the shortest path
- if the state space of the graph is very large, the breadth-first search uses a significantly larger amount of memory
  - the size of the stack in depth-first search is usually kept smaller than the size of the queue is breadth-first search
in most applications for example in artificial intelligence the size of the queue makes using breadth-first search impossible
• if the size of the graph can be infinite, the problem arises that the depth-first search doesn’t necessarily ever find a goal state and doesn’t finish until it runs out of memory
  – this occurs if the algorithm starts investigating a futile, infinite branch
  – this is not a problem with finite graphs
• some more complicated problems like searching for loops in the graph can be solved with depth-first search
  – the grey nodes from a path from the source to the current vertex
  – only black or grey vertices can be accessed through a black vertex
  ⇒ if a grey vertex can be reached from the current vertex, there is a loop in the graph

9.5 Dijkstra’s algorithm
A property called the weight can be added to the edges of the graph.
• the weight can represent the length of the route or the cost of the state transition
This makes finding the shortest route significantly more complicated.
• the shortest route from the source to the vertex is the one whose combined weight of the edges is as small as possible
• if the weight of each edge is 1, the problem can be solved with scanning through the part of the graph reachable from the source with breadth-first search
• if the weights can be <0, it is possible that there is no solution to the problem although there are possible paths in the graph
  – if there is a loop in the graph for which the sum of the weights if the edges is negative, an arbitrarily small sum of the weights can be created by travelling the loop as many times as necessary

This chapter covers finding the shortest path on a weighted, directed graph where the weights of the edges are positive and can be other than one.
• more complicated algorithms are needed for managing negative edge weights, e.g. the Bellman-Ford algorithm

The Dijkstra’s algorithm is used for finding the shortest weighted paths.
• finds the shortest paths from the source \( s \) to all vertices reachable from \( s \) by weighing the lengths of the edges with \( w \)
• chooses in each situation the shortest path it hasn’t investigated yet
  ⇒ it is a greedy algorithm

The algorithm uses a data structure \( Q \), which is a priority queue (the chapter 3.2 in the lecture notes)
\( w \) contains the weights of all the edges
Dijkstra’s algorithm uses the algorithm RELAX
• the relaxation of the edge \((u, v)\) tests could the shortest path found to vertex \( v \) be improved by routing its end through \( u \) and does so if necessary
Otherwise the algorithm greatly resembles the breadth-first search
• it finds the paths in the order of increasing length
• once the vertex \( u \) is taken from \( Q \) its weighted distance from \( s \) is \( d[u] = d(s, u) \) for sure
  – if the vertex taken from the priority queue is the goal vertex, the execution of the algorithm can be ended
Dijkstra\(\text{(s, w)}\)

\begin{itemize}
\item in the beginning the fields of each vertex are \(\text{colour} = \text{WHITE}, d = \infty, \pi = \text{NIL}\)
\item (the distance from the source to itself is 0)
\item (mark the source found)
\item (push the source into the priority queue)
\item (continue while there are vertices left)
\item (take the next vertex from the priority queue)
\item (go through the neighbours of \(v\))
\item (if the node has not been visited . . .)
\item (mark it found)
\item (push the vertex into the queue)
\item (mark \(u\) as processed)
\end{itemize}

Relaxation:
\begin{itemize}
\item if \(v \rightarrow d > u \rightarrow d + w(u, v)\) then
\item (if a new shorter route to \(v\) was found...)
\item (...decrement the distance of \(v\) from the source)
\item (mark \(v\) was reached through \(u\))
\end{itemize}

Running-time:
\begin{itemize}
\item the while loop makes atmost \(O(V)\) rounds and the for loop atmost \(O(E)\) rounds in total
\item the priority queue can be implemented efficiently with a heap or less efficiently with a list
\item one \(O(\log V)\) operation is done on each round of the while and the for loop when using a heap implementation
\Rightarrow the running-time of the algorithm is \(O((V + E) \log V)\)
\end{itemize}

10 Greedy algorithms and dynamic programming

This chapter covers two algorithm design principles: greedy algorithms and dynamic programming.

The greedy algorithm is often the most natural starting point for people when searching a solution to a given problem. A greedy algorithm doesn’t always produce correct results as the problem doesn’t against initial expectations fulfill all the requirements for the functionality of greedy algorithms.

\Rightarrow Solution: dynamic programming

10.1 Storing results to subproblems

Techniques that store the results of subproblems are:
\begin{itemize}
\item memoization
\item dynamic programming
\end{itemize}

Let’s take the algorithm that calculates Fibonacci numbers as an example.

Fibonacci numbers are defined:
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}, \text{ kun} n > 1
\end{align*}

A divide and conquer recursive solution can be implemented based on the definition:

\begin{itemize}
\item \text{Very-Slow-Fibonacci}(n)
\item if \(n = 0\) or \(n = 1\) then (the solution is known with 0 and 1)
\item return \(n\)
\item else (the result is calculated from \(F_{n-1}\) ja \(F_{n-2}\))
\item return \text{Very-Slow-Fibonacci}(n - 1) + \text{Very-Slow-Fibonacci}(n - 2)
\end{itemize}
The algorithm is not really that efficient:
- Each VERY-SLOW-FIBONACCI runs in $\Theta(1)$ time
- The picture shows that the solution computes solutions to the subproblems more than once for no reason:

- The amount of function calls is $O(2^n)$.
- The running-time of VERY-SLOW-FIBONACCI is exponential
- when this happens it is possible that there is no efficient solution available
- There is one for Fibonacci numbers.

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} \hline \text{Number} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline \text{Times} & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 & 233 & 377 & 610 & 987 & 1597 & 2584 \\ \hline \end{array} \]

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

- The running-time of MEMOIZED-FIBONACCI is the running-time of initializing the array $\Theta(n) + \text{the running time of MEM-FIB-RECURS}$
- the running-time of lines 1 and 3 in MEM-FIB-RECURS is $\Theta(1)$
- Line 2 gets executed at most once per each element in the array
- MEM-FIB-RECURS calls itself at most $2n$ times.
- The running-time of MEM-FIB-RECURS and the maximum depth of the recursion is $\Theta(n)$.

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

\[ \begin{align*}
\text{MEMOIZED-FIBONACCI}(n) &= \begin{cases} 
F[0] := 0 \\
F[1] := 1 \\
\text{initialize rest of the elements -1} \\
\text{return MEM-FIB-RECURS}(n) 
\end{cases} \\
\text{MEM-FIB-RECURS}(i) &= \begin{cases} 
\text{if } F[i] < 0 \text{ then (if the result hasn't been calculated)} \\
F[i] := \text{MEM-FIB-RECURS}(i-1) + \text{MEM-FIB-RECURS}(i-2) \\
\text{return } F[i] 
\end{cases} 
\end{align*} \]

The algorithm calculates the solutions to the subproblems several times. Let’s use this information to our advantage and store the results as they get counted.

Thus when the result is needed again, it doesn’t need to be calculated but the stored value can be used instead.

⇒ memoization

Let’s implement an algorithm based on memoization to Fibonacci numbers.

- when determining $F_n$, the possible solutions to subproblems are $F_0, F_1, \ldots, F_{n-1}$.
- The algorithm can be made simpler by storing $F_n$ also.
- Let’s use an array $F[0 \ldots n]$ to store the solutions of the subproblems
- The algorithm needs to be able to recognize whether the solution has already been calculated or not
  - Since Fibonacci numbers cannot be negative, let’s mark the uncalculated numbers in the array with -1

- When using memoization:
  - The algorithm calculates the solutions to subproblems several times, and it’s efficient to store calculated solutions.
  - This is particularly useful for recursive algorithms.

- About algorithms that use memoization:
  - Basic idea:
    - form an upper-bound to the set of subproblem results needed
    - design a data structure than can efficiently store said set
    - the parameters of the subproblem solutions act as the key to the data structure (i in the Fibonacci example)
  - Most of the time the data structure can be a simple array
  - A dynamic set with operations INSERT and SEARCH is needed in a general case
    - i.e. a dictionary without delete
  - The programmer doesn’t need to worry about the calculation order of the subproblems
    - recursion and the “not counted” mark make sure all results to the subproblems get calculated exactly once
These algorithms are:
- easy to make efficient
- difficult to analyze

- The calculation order of the results is unknown
  - all subproblem solutions need to be stored until the work is completed
  - a lot of memory is consumed

- naturally subproblem results can be deleted if it is sufficiently clear that the amount of work needed to recount them if necessary is kept reasonable

---

### 10.2 Dynamic programming

Dynamic programming is also based on memoization of subresults, but the order of calculation is fixed.

The subresults are usually stored in an ordinary array ⇒ “dynamic tabulation”, or something, would be a better name.

- typically used in optimization problems:
  - there are several alternate solutions to the problem
  - each solution has a price (cost function)
  - the cheapest solution is needed (or one of the cheapest)

The basic principles of an algorithms that uses dynamic programming:

1. Figure out the optimal structure of the solution
2. Define the optimal solution with the solutions to its subproblems
3. Calculate the result for the optimal solution by calculating the results of the subproblems from the smallest onwards
   - it may be useful to store information about the optimal solution simultaneously
   - the bottom-up approach is used
4. If the result of the optimal solution is not enough but the actual solution is also needed, it is calculated based on the information gathered in stage 3

A dynamic programming algorithm for Fibonacci numbers:

\[
\text{DYNAMIC-FIBONACCI}(n) \\
1 \quad F[0] := 0 \quad \text{(store the base cases in an array)} \\
2 \quad F[1] := 1 \\
3 \quad \text{for } i := 2 \text{ to } n \text{ do} \\
4 \quad \quad F[i] := F[i-1] + F[i-2] \quad \text{ (F is determined based on the previous results in the array)} \\
5 \quad \text{return } F[n]
\]

- The running-time of lines 1 and 2 in DYNAMIC-FIBONACCI is \(\Theta(1)\)
- Line 4 gets executed once per each element in the array ⇒ \(\Theta(n)\)
  - the running-time of DYNAMIC-FIBONACCI is \(\Theta(n)\).
- as the results to all previous subproblems are stored into the array the memory consumption of the algorithm is \(\Theta(n)\).

When should dynamic programming be used?

- the problem needs to have two properties:
  - the optimal solution can be given easily enough with the function of the optimal solutions to the subproblems: optimal substructure
  - the results to the subproblems can be used several times: overlapping subproblems
- for comparison: “memoization” requires only “overlapping subproblem”
  - it can be used more generally than dynamic programming
- “memoization” is better also when the tabulation of dynamic programming is needed only for a small amount of
the subproblems
⇒ dynamic programming calculates subproblems needlessly
– if all results to the subproblems are needed, dynamic
  programming is better than memoization by a constant
  coefficient
• dynamic programming makes certain improvements
  possible that cannot be done with memoization
  – reason: the order of the calculation of subresults is known
    and maybe it can even be effected
• if even “overlapping subproblems” doesn’t apply, “divide
  and conquer” may be the best solution
Let’s go through some of the properties of the optimal
substructure:
• the substructure can be proven optimal in the following
  way:
  – let’s assume that the optimal solution to the problem
    contains a non-optimal subresult

⇒ prove that replacing it with an optimal one would
improve the solution to the entire problem
⇒ the solution wasn’t optimal after all ⇒ contradiction
⇒ the initial assumption was incorrect ⇒
  the substructure is optimal
• the problem can be split into subproblems in several ways
• all of them are usually not as suitable for dynamic
  programming
• the set of subproblems should be kept as small as possible
  – all subproblems must be calculated and stored into the
    array
⇒ it’s worth testing the division plan by breaking the problem
  into subproblems and into their subproblems on 2...4 levels
  and see if the same subproblems start repeating
• this often clarify which subproblem needs to be taken into
  account when designing the array

• for example the \( n \)th Fibonacci number:
  \[
  F_n = F_{n-1} + F_{n-2}
  = F_{n-2} + F_{n-3} + F_{n-3} + F_{n-4}
  = F_{n-3} + F_{n-4} + F_{n-4} + F_{n-5} +
  F_{n-4} + F_{n-5} + F_{n-5} + F_{n-6}
  \]
⇒ same subproblems are clearly repeated and the result
  can be represented based on the subproblems solved
  earlier

Improving the solution:
• It is often easy to improve an algorithm implemented based
  on one principle once it has been developed
• dynamic programming algorithms can sometimes be
  improved by
  – changing the calculation order
  – destroying the subresults that are not going to be used
    again
• destroying futile subresults saves memory
• for example, when calculating Fibonacci numbers only the
  two latest results are needed at each point
⇒ instead of \( F[0...n] \) it is enough to maintain a “window” of
  two subresults which is then slid over the array
Memor-saving fibonacci(n)
1 if \( n = 0 \) then
2 return 0
3 previous := 0 (store the base cases)
4 newest := 1
5 for \( i := 2 \) to \( n \) do
6 \( x := \) newest (calculate \( F_i \) and update the values)
7 newest := newest + previous
8 previous := \( x \)
9 return newest

• this is how most of you would have solved this in the first place!
• the running-time of the algorithm is still \( \Theta(n) \), but the memory consumption is only \( \Theta(1) \)

10.3 Greedy algorithms
The most natural way to approach a problem is often the greedy algorithm. Ratkaisun etsimiseen.

• the idea of a greedy algorithm is to choose the best appearing choice at that moment and ignore the others completely.

Example: making a timetable for a concert hall

• the owner of the concert hall has received \( n \) reservations
  - for simplicity we'll mark the reservations with 1, 2, \ldots, \( n \)
  - the set of reservations are marked with \( S \)
• only one event can be held at a time in the hall
• each reservation \( i \) has a starting time \( s_i \) and an finishing time \( f_i \) such that \( f_i > s_i \)
  - the finishing time indicates the time when the next event can start (the hall is empty and clean etc.)
• the rent of the hall is almost independent of the length of the event \( \Rightarrow \) the goal is to book as many events as possible into the hall

• the algorithm:

\[
\text{GREEDY-ACTIVITY-SELECTOR}(S, f) \quad \text{(uses a queue } Q)\]
1 sort the reservations so that \( f_1 \leq f_2 \leq \ldots \leq f_n \)
2 \( n := S.\text{length} \)
3 \( Q.\text{push}(1) \) (push the first event into the queue)
4 \( j := 1 \)
5 for \( i := 2 \) to \( n \) do (on each step add the event which)
6 \( \text{if } s_i \geq f_j \text{ then } \) (starts as early as possible)
7 \( Q.\text{push}(i) \)
8 \( j := i \)
9 return \( Q \)

• the running-time
  - sorting \( O(n \log n) \)
  - \text{GREEDY-ACTIVITY-SELECTOR} \( \Theta(n) \), sillä \( Q.\text{push}() \) on \( \Theta(1) \).
  \( \Rightarrow \) quite efficient

When does a greedy algorithm produce correct results?
• there is no general test that could be used to test whether the greedy approach produces correct results or not
• a greedy algorithm is most likely good when the problem has two properties
  - greedy choice property
  - optimal substructure
• the optimal substructure was discussed with dynamic programming
  - it was a requirement also there
  - the event management problem has an optimal substructure: after choosing the first event the optimal solution to the rest of the problem combined with the solution to the first part is the optimal solution to the entire problem
• the greedy choice properly is that the greedy choice may not divert the algorithm away from the global optimal solution
• A lot of effort and skill is sometimes needed to prove that
the problem fulfills the greedy choice property
• Sometimes at first glance the problem seems like it fulfills the
  greedy choice property
• Example: minimizing the amount of coins
  – Always choose the largest possible coin that still fits the
    sum
• The greedy algorithm works with familiar euro coins
• The coin values in the kingdom of Poldavia are due to the
  monarchs superstition his lucky numbers
⇒ the greedy algorithm doesn’t produce the correct result in
  Poldavia: the coins to the money sum 60 cannot be
  minimized with a greedy selection if the coins are 1, 20 and
  50
• Solution: dynamic programming

⇒ the task is divided into two depending on whether the
sum is larger or smaller than \( K[k]\).
  - If the sum \( m < K[k]\) the coin \( K[k]\) cannot be in it and the
    amount of coins needed is \( P(m, k) = P(m, k - 1)\)
  - If \( m \geq K[k]\), \( K[k]\) can either be chosen into the sum or
    it can be left out
  - If \( K[k]\) is chosen in, the search for the solution if
    continued for the sum \( m - K[k]\)
  - If the coin is not taken in, the solution is \( P(m, k - 1)\)
• Since the amount of coins needs to be minimized the
  alternative that produces that smaller
  \( P(m, k) = \min(1 + P(m - K[k]), P(m, k - 1))\).

A dynamic programming algorithm:

- First let’s form the optimal solution recursively:

  Coins (K):
  \[
  \begin{array}{cccccccccc}
  1 & 2 & 5 & 10 & 20 & 50 \\
  1 & 2 & 5 & 10 & 20 & 50 \\
  1 & 2 & 5 & 10 & 20 & 50 \\
  1 & 2 & 5 & 10 & 20 & 50 \\
  1 & 2 & 5 & 10 & 20 & 50 \\
  \end{array}
  \]

  ⇒ the task is divided into two depending on whether the sum is larger or smaller than \( K[k]\).
  - If the sum \( m < K[k]\) the coin \( K[k]\) cannot be in it and the amount of coins needed is \( P(m, k) = P(m, k - 1)\)
  - If \( m \geq K[k]\), \( K[k]\) can either be chosen into the sum or it can be left out
  - If \( K[k]\) is chosen in, the search for the solution if continued for the sum \( m - K[k]\)
  - If the coin is not taken in, the solution is \( P(m, k - 1)\)

  Since the amount of coins needs to be minimized the alternative that produces that smaller
  \( P(m, k) = \min(1 + P(m - K[k]), P(m, k - 1))\).

MIN-COIN(m, k)
1 for \( i := 0 \) to \( m \) do
2 \( P[i, 1] := i \) \hspace{1cm} \text{(first base case)}
3 for \( i := 1 \) to \( k \) do
4 \( P[0, i] := 0 \) \hspace{1cm} \text{(second base case)}
5 for \( i := 1 \) to \( m \) do
6 \( P[i, 1] := 0 \) \hspace{1cm} \text{(fill the array based on the recursive equation)}
7 \( P[i, j] := 1 + P[i - K[j], j] \) \hspace{1cm} \text{(top to bottom, left to right)}
8 \( P[i, j] := \min(P[i - K[j], j], P[i, j - 1]) \) \hspace{1cm} \text{(coin j too large)}
9 else \( P[i, j] := \min(1 + P[i - K[j], j], P[i, j - 1]) \) \hspace{1cm} \text{(chose the alternative that requires less coins)}
10 \( P[i, j] := \min(1 + P[i - K[j], j], P[i, j - 1]) \)

Running-time:
- The result array is filled from the simple base cases towards the larger ones until the desired solution is reached
- An element in the array is determined based on previously calculated results
⇒ The array needs to be filled entirely
⇒ Running-time and memory consumption \( \Theta(km) \)
11 What to do when there is no efficient solution?

Even a remotely efficient solution is not known to all problems.

- a polynomial solution hasn’t been found
- a solution with $O(n^k)$ is not known, where $k$ is some constant
- these problems belong into some other complexity class than P, problems solvable in polynomial time
- the next complexity class is NP where it is not necessary to know an efficient way to solve the problem but there must be an efficient (in P) way to verify the correctness of a solution

Many, also in practise important, problems are in NP and a polynomial time solution is not known to them.

- with these, the worst-case running-time of known algorithms is exponential
- Graph coloring: the task is to find out is each vertex of a graph can be colored with one color out of $k$ so that the ends of each edge have different colors

- it is easy to see that checking the correctness of the solution is easy but finding the solution isn’t
- The travelling salesman problem: the vertices of a graph are seen as cities and the edges as roads between cities. The problem is to find out if there are routes almost a certain length that travel through each node at most once and in the end returns back to the source vertex
- Hamiltonian cycle: a special case of the previous problem. The length of each edge is 1 and the upper bound for the length of the route is the amount of vertices in the graph
- Hamiltonian path: as the previous but the requirement to reach the source has been left out
- The barrel problem: How many barrels is needed to pack $n$ items? The barrels can carry only a certain amount of weight and the items weigh $W[1, ..., n]$

The problems above are all NP-complete, i.e. all problems in NP can be reduced to them.

If you notice that the solution to your problem would solve for example the Hamiltonian path, it probably is a good idea to rethink the situation and to try to find alternative approaches.

- it is possible in principle that there is a polynomial solution available but since it hasn’t been found so far, despite all the effort, it is unlikely that you would find one either

It’s worth considering if an approximate solution could be used.

- some usually reach the correct result and sometimes, rarely, the incorrect one
  - the Monte Carlo -algorithm
  - is not a well defined algorithm defined in chapter 1.2
  - for example, testing is a large prime is a prime can be done efficiently with an approximate algorithms when is commonly used in cyphering
- additionally there are algorithms that don’t always find the solution but if the do, it is the correct one
  - the Las Vegas -algorithm

- some approximate solutions find an almost optimal solution that is often usable
  - there are this types of solution to the travelling salesman problem