TIE-20106 DATA STRUCTURES AND ALGORITHMS

Bibliography

These lecture notes are based on the notes for the course OHJ-2016 Utilization of Data Structures. All editorial work is done by Terhi Kilamo and the content is based on the work of Professor Valmari and lecturer Minna Ruuska.

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

1 Introduction

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

Algorithms in the world

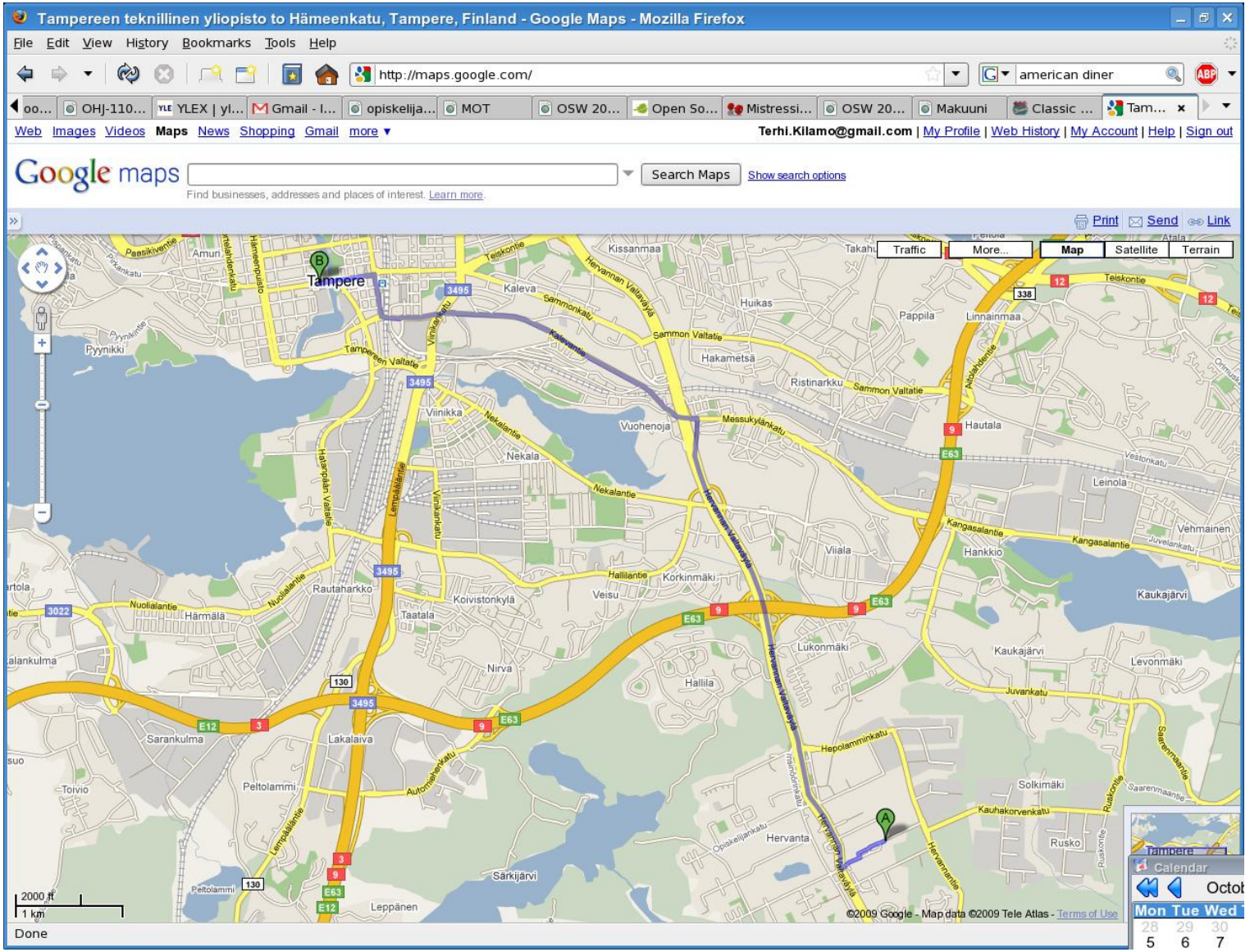
1.1 Why?

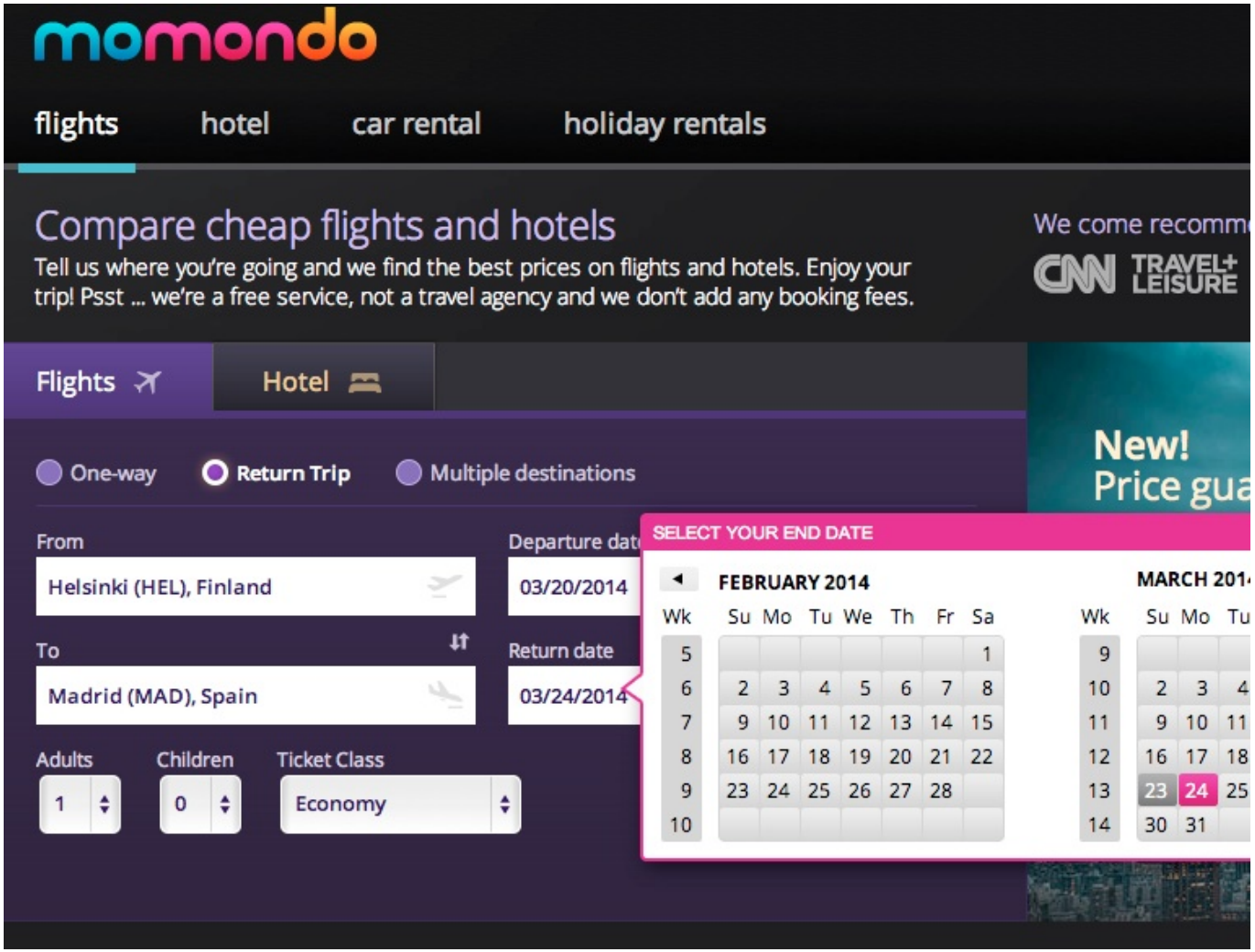
What are the three most important algorithms that affect YOUR daily life?

Picture: Chris Watt

There are no computer programs without algorithms

- algorithms make for example the following applications possible:


Tampereen teknillinen yliopisto to Hämeenkatu, Tampere, Finland - Google Maps - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://maps.google.com/
american diner
Web Images Videos Maps News Shopping Gmail more
Terhi.Kilamo@gmail.com | My Profile | Web History | My Account | Help | Sign out
Google maps
Search Maps
Show search options
Find businesses, addresses and places of interest. Learn more.
Print Send Link
Traffic More... Map Satellite Terrain
Tampere
Pyynikki
Kaleva
Viinikka
Nekala
Vuohenoja
Hakametsä
Pappila
Huikas
Linnainmaa
Ristinarkku
Hautala
Leinola
Vehmainen
Hankkio
Kaukajärvi
Viiala
Korkinmäki
Veisu
Rautaharkko
Koivistonkylä
Taatala
Härmälä
Nirva
Hallila
Lukonmäki
Levonmäki
Sarankulma
Lakalaiva
Peltolammi
Toivio
Hervanta
Rusko
Solkimäki
Särkijärvi
Leppänen
Tampere
Calendar
Mon Tue Wed
2000 ft
1 km
©2009 Google - Map data ©2009 Tele Atlas - Terms of Use
Done



algorithms are at work whenever a computer is used

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

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

Modern programming languages provide easy to use library implementations for data structures (C++ standard library, JCF). Understanding the properties of these and the limitations there may be for using them requires theoretical knowledge on basic data structures.

Ever gotten frustrated on a program running slowly?

- functionality is naturally a top priority but effiency 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

2 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.

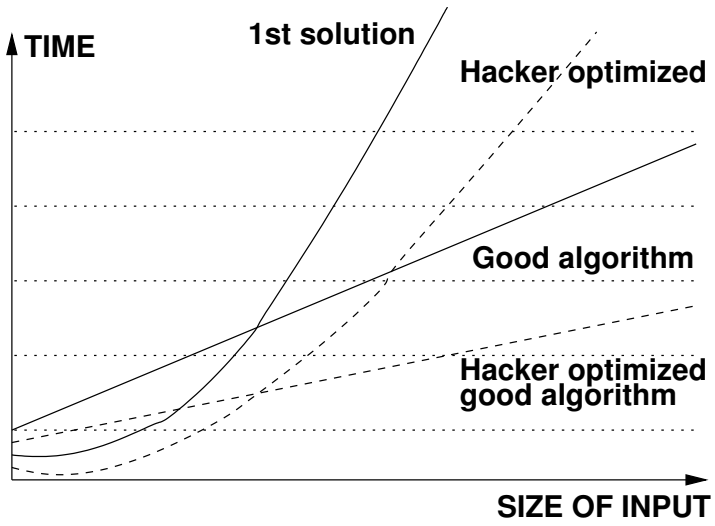
2.1 Goals of the course

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 data structures and algorithms commonly used in programming are covered.

- 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.

- The course concentrates on the so called "good algorithms" shown in the picture on the right.
- The emphasis is on the time the algorithm uses to process the data as the size of the input gets larger. Less attention is paid to optimization details.

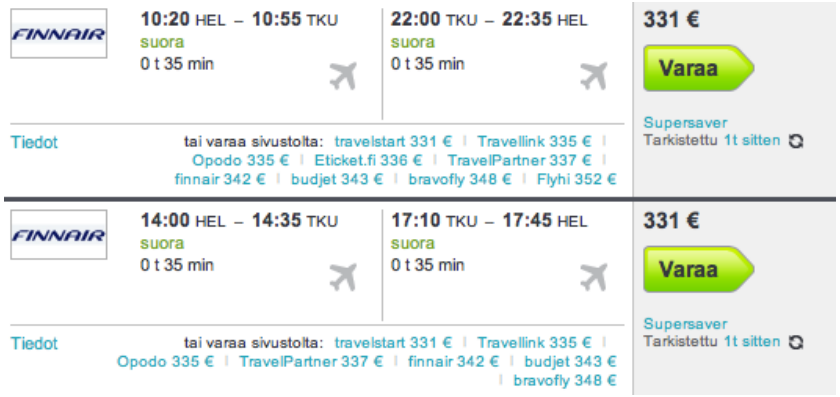


2.2 Terminology

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.



- well defined =
  - each step is detailed enough for the reader (human or machine) to execute
  - each step in unambiguous
  - the same requirements apply to the execution order of the steps
  - the execution is finite, i.e. it ends after a finite amount of steps.

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
  - finding flight connections

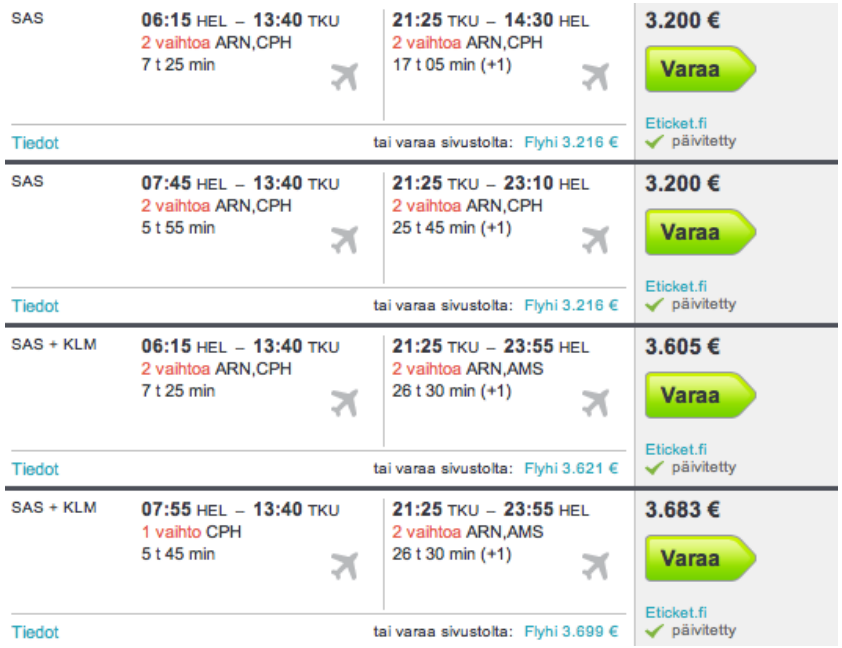
    input: a graph of flight connections, cities of departure and destination

    results: Flight numbers, connection and price information

- 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 a very usefull 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

- usually algorithms are implemented as computer programs or in hardware
- in practise, the implementation must take several "engineering viewpoints" into accout
  - accomodation 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 $\Rightarrow$ modularity etc.

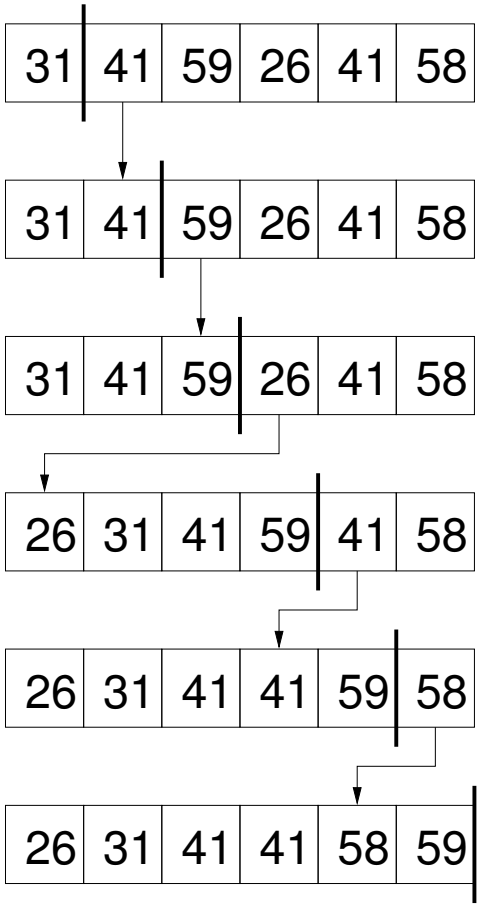
  $\Rightarrow$ the idea of the algorithm may get lost under the implementation details

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:

Figure 1: picture from Wikipedia

- 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 upto 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 it's 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

- 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. $x{\rightarrow}name$, $x{\rightarrow}number$
- 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: NIL

2.3 Implementing algorithms

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

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

- numbers are rarely sorted alone, we sort structures with
  - a *key*
  - *satellite data*
- the key sets the order
  $\Rightarrow$ 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

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

```
1  for j := 2 to A.length do
2      temp := A[j]
3      i := j − 1
4      while i > 0 and A[i].key > temp.key do
5          A[i + 1] := A[i]
6          i := i − 1
7      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.

The programming language and the problem to be solved also often dictate other implementation details, for example:

- Indexing starts from 0 (in pseudocode often from 1)
- Is indexing even used, or some other method of accessing data (or do we use arrays or some other data structures)
- (C++) Is the data really inside the array/datastructure, or somewhere else at the end of a pointer (in which case the data doesn't have to be moved and sharing it is easier). Many other programming languages always use pointers/references, so you don't have to choose.
- If you refer to the data indirectly from elsewhere, does it happen with
  - Pointers (or references)
  - Smart pointers (C++, `shared_ptr`)
  - Iterators (if the data is inside a datastructure)
  - Index (if the data is inside an array)
  - Search key (if the data is insde a data structure with fast search)

- Is recursion implemented really as recursion or as iteration
- Are algorithm "parameters" in pseudocode really parameters in code, or just variables

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 abowe in C++:

```
#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.

$\Rightarrow$ From now on, program code implementations are not normally shown.

3 Algorithm design techniques

3.1 Algorithm Design Technique: Decrease and conquer

The most straightforward algorithm *design technique* covered on the course is *decrease and conquer*.

- initially the entire input is unprocessed
- the algorithm processes a small piece of the input on each round

  $\Rightarrow$ 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 "decrease and conquer" 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

INSERTION-SORT

```
INSERTION-SORT(A)                          (input in array A)
1  for j := 2 to A.length do               (move the limit 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 of 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 it's correct location)
```

3.2 Algorithm Design Technique: Divide and Conquer

We've earlier seen the *decrease and conquer* 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 decrease and conquer 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

3.3 QUICKSORT

Let's next cover a very efficient sorting algorithm QUICKSORT.

QUICKSORT is a divide and conquer algorithm.

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$(A, left, right)$

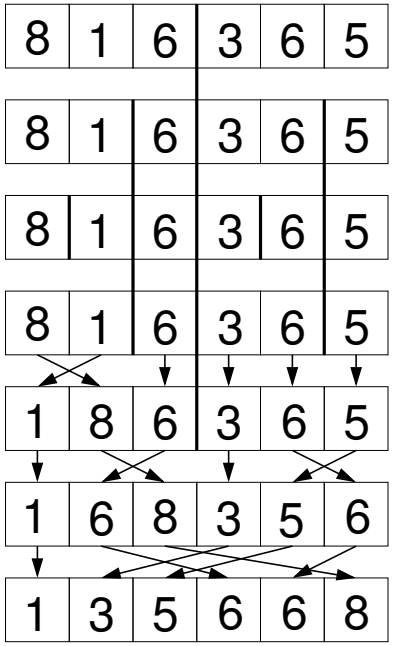
```
1  if left < right then                          (do nothing in the trivial case)
2      pivot := PARTITION(A, left, right)        (partition in two)
3      QUICKSORT(A, left, pivot − 1)             (sort the elements smaller than the pivot)
4      QUICKSORT(A, pivot + 1, right)            (sort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

The *partition algorithm* rearranges the subarray in place

```
PARTITION(A, left, right)
1   pivot := A[right]                    (choose the last element as the pivot)
2   i := left − 1                        (use i to mark the end of the smaller elements)
4   for j := left to right − 1 do        (scan to the second to last element)
6       if A[j] ≤ pivot                  (if A[j] goes to the half with the smaller elements...)
9           i := i + 1                   (... increment the amount of the smaller elements...)
12          exchange A[i] ↔ A[j]         (... and move A[j] there)
12  exchange A[i + 1] ↔ A[right]         (place the pivot between the halves)
13  return i + 1                         (return the location of the pivot)
```

MERGE-SORT

- divide the elements in the array into two halves.
- continue dividing the halves further in 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



MERGE-SORT$(A, left, right)$

1 **if** $left < right$ **then** *(if there are elements in the array...)*
2 $middle$:= $\lfloor (left + right)/2 \rfloor$ *(... divide it into half)*
3 MERGE-SORT$(A, left, middle)$ *(sort the upper half...)*
4 MERGE-SORT$(A, middle + 1, right)$ *(... and the lower)*
5 MERGE$(A, left, middle, right)$ *(merge the parts maintaining the order)*

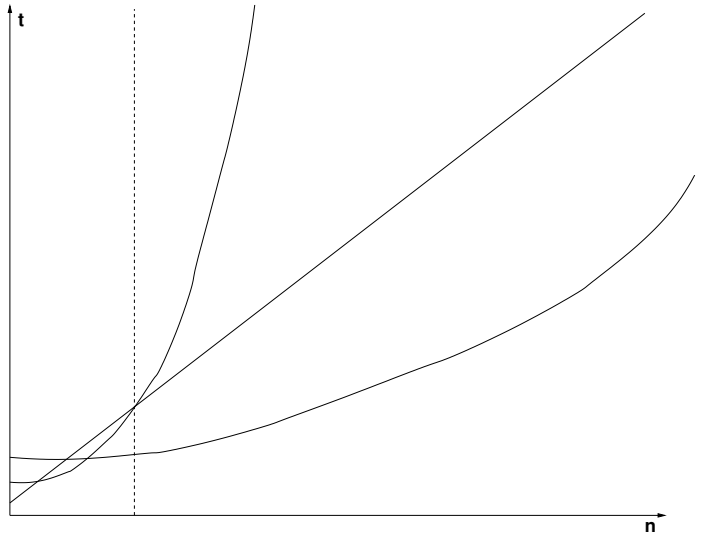
- the MERGE-algorithm for merging the subarrays:

```
MERGE(A, left, middle, right)
1   for i := left to right do      (scan through the entire array...)
2       B[i] := A[i]               (... and copy it into a temporary array)
3   i := left                      (set i to indicate the endpoint of the sorted part)
4   j := left; k := middle + 1     (set j and k to indicate the beginning of the subarrays)
5   while j ≤ middle and k ≤ right do      (scan until either half ends)
6       if B[j] ≤ B[k] then        (if the first element in the lower half is smaller...)
7           A[i] := B[j]           (... copy it into the result array...)
8           j := j + 1             (... increment the starting point of the lower half)
9       else                       (else...)
10          A[i] := B[k]           (... copy the first element of the upper half...)
11          k := k + 1             (... and increment its starting point)
12      i := i + 1                 (increment the starting point of the finished set)
13  if j > middle then
14      k := 0
15  else
16      k := middle − right
17  for j := i to right do         (copy the remaining elements to the end of the result)
18      A[j] := B[j + k]
```

4 Measuring efficiency

This chapter dicusses the analysis of algorithms: the efficiency of algorithms and the notations used to describe the *asymptotic* behavior of an algorithm.

In addition the chapter introduces two algorithm design techniques: *decrease and conquer* and *divide and conquer*.



4.1 Asymptotic notations

It is occasionally important to know the exact time it takes to perform a certain operation (in real time systems for example).

Most of the time it is enough to know how the running time of the algorithm changes as the input gets larger.

- The advantage: the calculations are not tied to a given processor, architecture or a programming language.
- In fact, 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, the amount of times a certain operation is done is irrelevant 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.

  $\Rightarrow$ This can be done since as the input gets large enough the lower-order terms get insigficant when compared to the leading term.

  $\Rightarrow$ 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, Θ.

$$f(n) = 23n^2 + 2n + 15 \Rightarrow f \in \Theta(n^2)$$

$$f(n) = \tfrac{1}{2} n \lg n + n \Rightarrow f \in \Theta(n \lg n)$$

Example 1: addition of the elements in an array

```
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:

n	time = $2n$ + 1
1	3
10	21
100	201
1000	2001
10000	20001

- 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:

  $f(n) = 2n + 1 \Rightarrow n$

  $\Rightarrow$ we get $f \in \Theta(n)$ as the result

  $\Rightarrow$ the running time depends *linearly* on the size of the input.

Example 2: searching from an unsorted array

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

- the location of the searched element in the array affects the running time.
- the running time depends now both on the size of the input and on the order of the elements
  $\Rightarrow$ 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:

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

- 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.

  $\Rightarrow$ each case needs to be investigated separately.

Example 3: finding the common element in two arrays

```
1   for i := 1 to A.length do
2       for j := 1 to B.length do
3           if A[i] = B[j] then
4               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)

RETURN OF INSERTION-SORT

```
INSERTION-SORT(A)                          (input in array A)
1   for j := 2 to A.length do              (move the limit 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 of 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 it's correct location)
```

- line 1 is executed n times
- lines 2 and 3 are executed n - 1 times
- line 4 is executed at least n - 1 and at most (2 + 3 + 4 + $\cdots$ + n - 2) times
- lines 5 and 6 are executed at least 0 and at most (1 + 2 + 3 + 4 + $\cdots$ + n - 3) times

- 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

  $\Rightarrow$ the amount of comparisons needed is half the amount of the worst case where all the element pairs were in an incorrect order

  $\Rightarrow$ the average-case running time is the worst-case running time divided by two: ((n - 1)n)/ 4 = $\Theta(n^2)$

4.2 Algorithm Design Technique: Divide and Conquer

We've earlier seen the *decrease and conquer* 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 decrease and conquer 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

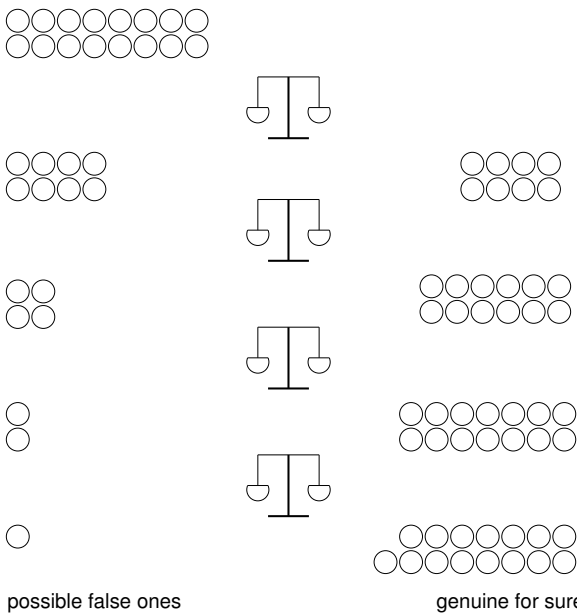
Let's get back to the claim made earlier about the complexity notation not being fixed to programs and take an everyday, concrete example

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 Decrease and conquer by choosing a random coin and by comparing it to the other coins one at a time.

  $\Rightarrow$ 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 $2^{weighings}$ coins, so based on the definition of the logarithm:

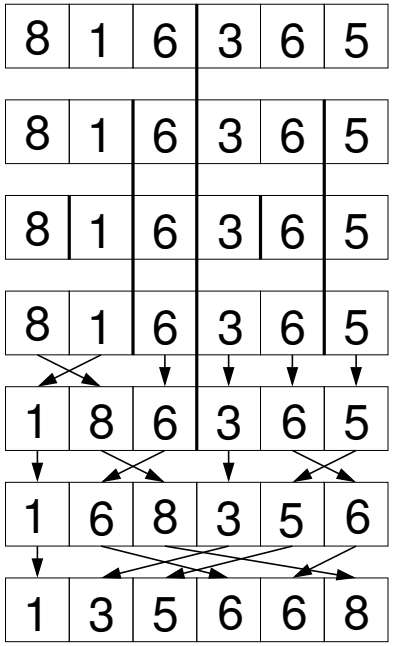
$$2^{weighings} = n \Rightarrow log_2 n = weighings$$

- Only $\log_2 n$ weighings is needed, which is significantly fewer than $n/2$ when the amount of coins is large.

  $\Rightarrow$ The complexity of the solution is $\Theta(\lg n)$ both in the best and the worst-case.

MERGE-SORT:

- divide the elements in the array into two halves.
- continue dividing the halves further in 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



- the MERGE-algorithm for merging the subarrays:

```
MERGE(A, left, middle, right)
1   for i := left to right do      (scan through the entire array...)
2       B[i] := A[i]               (... and copy it into a temporary array)
3   i := left                      (set i to indicate the endpoint of the sorted part)
4   j := left; k := middle + 1     (set j and k to indicate the beginning of the subarrays)
5   while j ≤ middle and k ≤ right do      (scan until either half ends)
6       if B[j] ≤ B[k] then        (if the first element in the lower half is smaller...)
7           A[i] := B[j]           (... copy it into the result array...)
8           j := j + 1             (... increment the starting point of the lower half)
9       else                       (else...)
10          A[i] := B[k]           (... copy the first element of the upper half...)
11          k := k + 1             (... and increment its starting point)
12      i := i + 1                 (increment the starting point of the finished set)
13  if j > middle then
14      k := 0
15  else
16      k := middle − right
17  for j := i to right do         (copy the remaining elements to the end of the result)
18      A[j] := B[j + k]
```

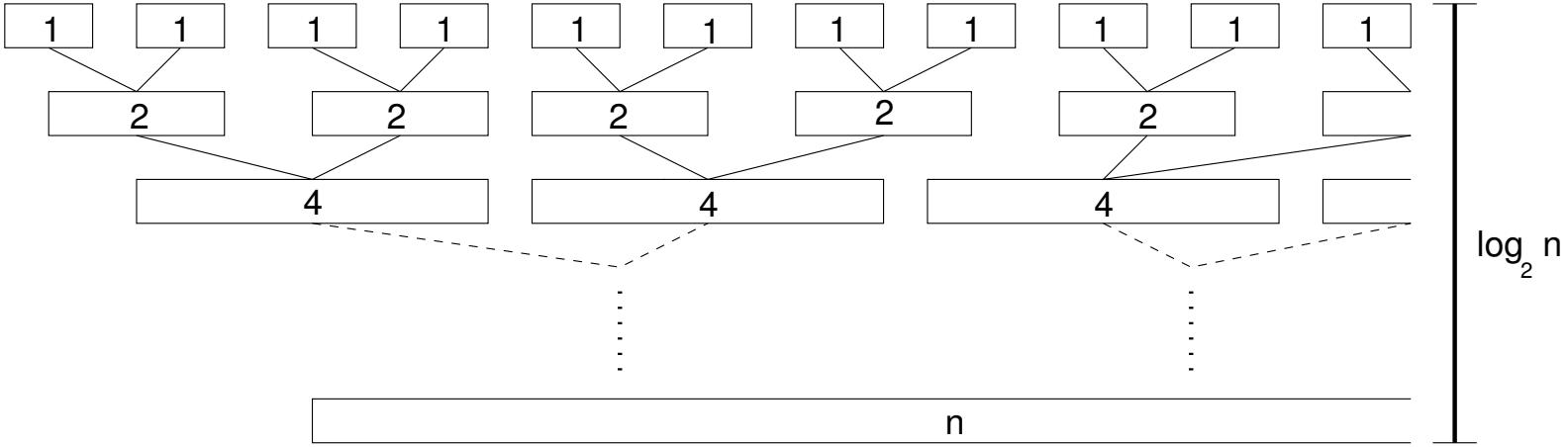
- MERGE-SORT

```
MERGE-SORT(A, left, right)
1  if left < right then                    (if there are elements in the array...)
2      middle := ⌊(left + right)/2⌋        (... divide it into half)
3      MERGE-SORT(A, left, middle)         (sort the upper half...)
4      MERGE-SORT(A, middle + 1, right)    (... and the lower)
5      MERGE(A, left, middle, right)       (merge the parts maintaining the order)
```

- MERGE merges the arrays by using the "decrease and conquer" 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

$\Rightarrow$ the running time of the entire algorithm is obtained by combining the results. It's $\Theta(n)$ both in the best and the worst case.

- Like with QUICKSORT, the analysis of MERGE-SORT is not as straghtforward since it is recursive.
- MERGE-SORT calls itself and MERGE, all other operations are constant time $\Rightarrow$ we can concentrate on the time used by the intances of MERGE, everything else is constant time.



- The instances of MERGE form a treelike structure shown on the previous page.
  - the sizes of the subarrays are marked on the instances of MERGE in the picture
  - on the first level the length of each subarray is 1 (or 0)
  - the subarrays on the upper levels are always two times as large as the ones on the previous level
  - the last level handles the entire array
  - the combined size of the subarrays on each level is n
  - the amount of instances of MERGE on a given level is two times the amount on the previous level.
    $\Rightarrow$ the amount increases in powers of two, so the amount of instances on the last level is 2^h, where h is the height of the tree.
  - on the last level there are approximately n instances
    $\Rightarrow 2^h = n \Leftrightarrow \log_2 n = h$, the height of the tree is $\log_2 n$
  - since a linear amount of work is done on each level and there are $\lg n$ levels, the running time of the entire algorithm is $\Theta(n \lg n)$

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 is 1000 000 n^2 is 1000 000 000 000, while n log n is about 19 930 000

Advantages and disadvantages of Mergesort

Advantages

- Running time $\Theta(n \lg n)$
- Stable

Disadvantages

- MERGE-SORT requires $\Theta(n)$ extra memory, INSERTION-SORT and QUICKSORT sort in place.
- Constant coefficient quite large

4.3 RETURN OF QUICKSORT

Let's next cover a very efficient sorting algorithm QUICKSORT.

QUICKSORT is a divide and conquer algorithm.

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$(A, left, right)$

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

The *partition algorithm* rearranges the subarray in place

```
PARTITION(A, left, right)
1   pivot := A[right]                    (choose the last element as the pivot)
2   i := left − 1                        (use i to mark the end of the smaller elements)
4   for j := left to right − 1 do        (scan to the second to last element)
6       if A[j] ≤ pivot                  (if A[j] goes to the half with the smaller elements...)
9           i := i + 1                   (... increment the amount of the smaller elements...)
12          exchange A[i] ↔ A[j]         (... and move A[j] there)
12  exchange A[i + 1] ↔ A[right]         (place the pivot between the halves)
13  return i + 1                         (return the location of the pivot)
```

How fast is PARTITION?

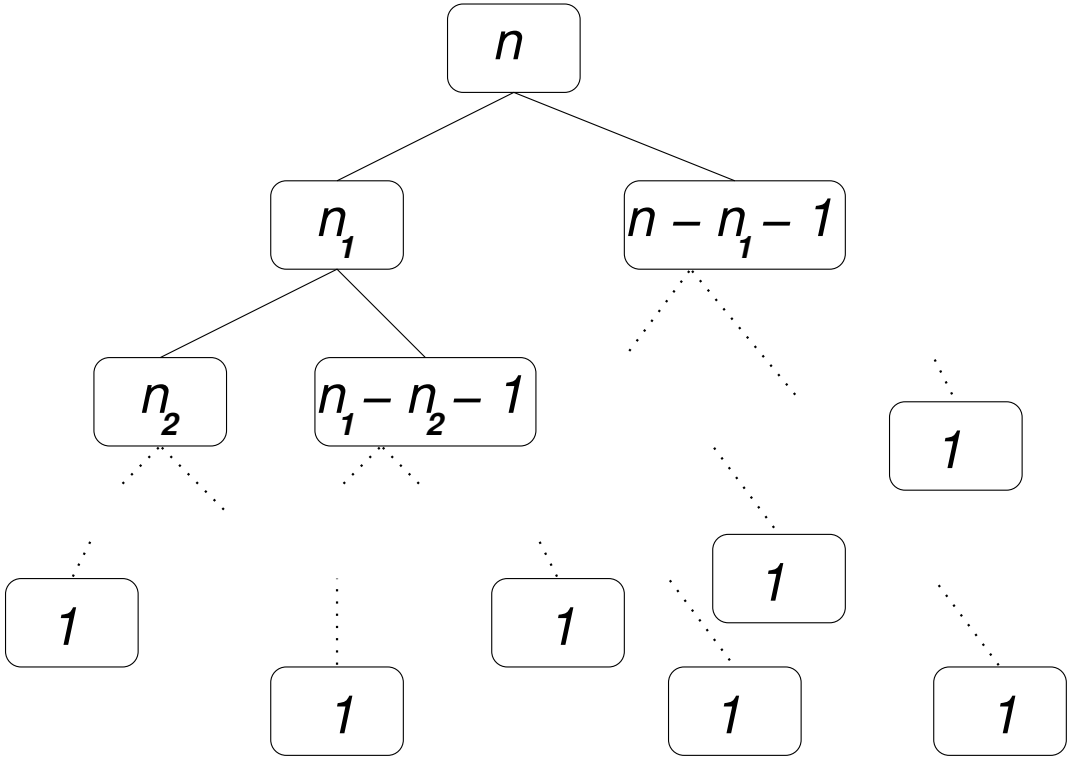
- The **for**-loop is executed n - 1 times when n is r - p
- All other operations are constant time.

$\Rightarrow$ The running-time is $\Theta(n)$.

Determining the running-time of QUICKSORT is more difficult since it is recursive. Therefore the equation for its running time would also be recursives.

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

- 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.



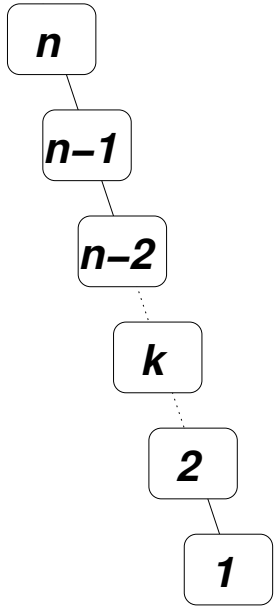
- The total time is the sum of the running times of the nodes in the picture abowe.
- The execution is constant time for an array of size 1.
- For the other the execution is linear to the size of the array.
  $\Rightarrow$ The total time is Θ(the sum of the numbers of the nodes).

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

  $\Rightarrow$ there can be atmost n 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

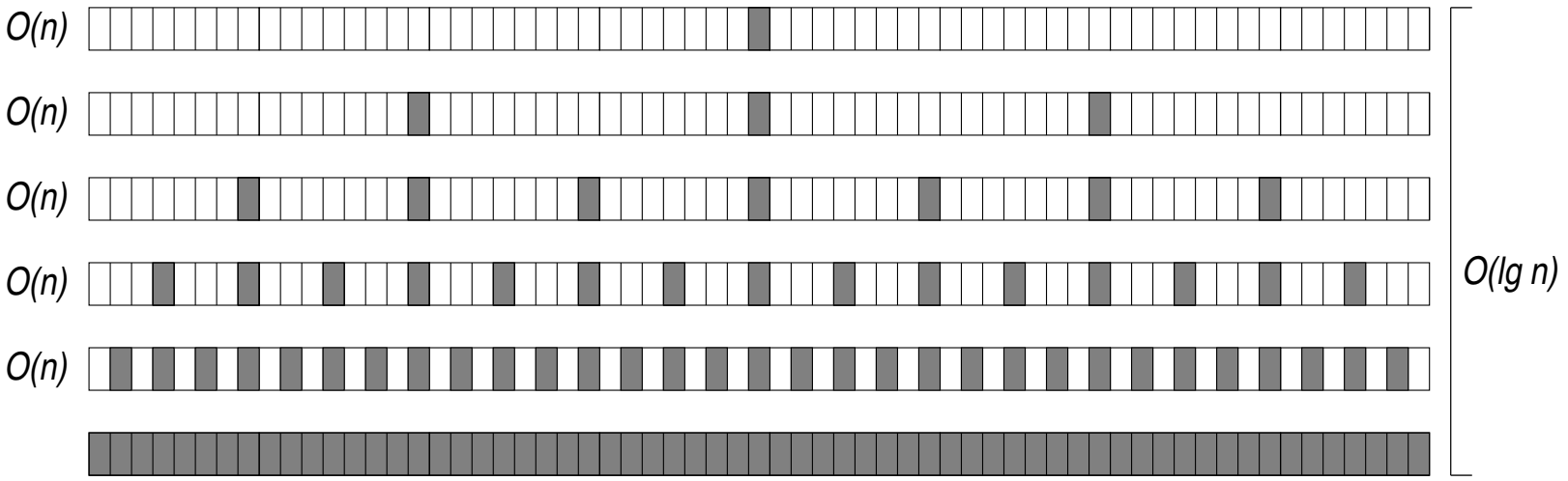


$\Rightarrow$ the worst case running time of QUICKSORT is $\Theta(n^2)$

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

- The picture below shows how the subarrays get smaller.
  - The grey boxes mark elements already in their correct position.
- The amount of work on each level is in $\Theta(n)$.
  - a pessimistic estimate on the height of the execution tree is in the best-case $\Rightarrow \Theta(\lg n)$

$\Rightarrow$ The upper limit for the best-case efficiency is $\Theta(n \lg n)$.



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 occurence of the worst-case situation's likelyhood 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

4.4 Algorithm Design Technique: Randomization

Randomization is one of the design techniques of algorithms.

- A pathological occurence 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 likelyhood 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.

- 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 choise
  - 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 RANDOMIZED-QUICKSORT which uses a randomized 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 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.

```
RANDOMIZED-PARTITION(A, left, right)
1  i := RANDOM(left, right)              (choose a random element as pivot)
2  exchange A[right] ↔ A[i]              (store it as the last element)
3  return PARTITION(A, left, right)      (call the normal partition)
```

```
RANDOMIZED-QUICKSORT(A, left, right)
1  if left < right then
2       pivot := RANDOMIZED-PARTITION(A, left, right)
3       RANDOMIZED-QUICKSORT(A, left, pivot − 1)
4       RANDOMIZED-QUICKSORT(A, pivot + 1, right)
```

The running-time of RANDOMIZED-QUICKSORT is $\Theta(n \lg n)$ on average just like with normal 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 likelyhood holds for RANDOMIZED-QUICKSORT for sure.
- This holds for the normal QUICKSORT only if the data is evenly distributed.

$\Rightarrow$ RANDOMIZED-QUICKSORT is better than the normal QUICKSORT in general

QUICKSORT can be made more efficient with other methods:

- An algorithm efficient with small inputs (e.g.INSERTIONSORT) can be used to sort the subarrays.
  - they can also be left unsorted and in the end sort the entire array with INSERTIONSORT
- The median of three randomly selected elements can be used as the pivot.
- It's always possible to use the median as the pivot.

The median can be found efficiently with the so called lazy QUICKSORT.

- Divide the array into a "small elements" lower half and a "large elements" upper half like in QUICKSORT.
- Calculate which half the ith element belongs to and continue recursively from there.
- The other half does not need to be processed further.

```
RANDOMIZED-SELECT(A, left, right, goal)
1  if left = right then                          (if the subarray is of size 1...)
2      return A[left]                            (... return the only element)
3  pv := RANDOMIZED-PARTITION(A, left, right)    (divide the array into two halves)
4  k := pv − left + 1                            (calculate the number of the pivot)
5  if k = goal then                              (if the pivot is the goalth element...)
6      return A[pv]                              (...return it)
7  else if goal < k then                         (continue the search from the small ones)
8      return RANDOMIZED-SELECT(A, left, pv − 1, goal)
9  else                                          (continue on the large ones)
10     return RANDOMIZED-SELECT(A, pv + 1, right, goal − k)
```

The lower-bound for the running-time of RANDOMIZED-SELECT:

- Again everything else is constant time except the call of RANDOMIZED-PARTITION and the recursive call.
- In the best-case the pivot selected by RANDOMIZED-PARTITION is the $goal$th element and the execution ends.
- RANDOMIZED-PARTITION is run once for the entire array.

$\Rightarrow$ The algorithm's best case running-time is $\Theta(n)$.

The upper-bound for the running-time of RANDOMIZED-SELECT:

- RANDOMIZED-PARTITION always ends up choosing the smallest or the largest element and the $goal$th element is left in the larger half.
- the amount of work is decreased only by one step on each level of recursion.

$\Rightarrow$ The worst case running-time of the algorithm is $\Theta(n^2)$.

The average-case running-time is however $\Theta(n)$.

The algorithm is found in STL under the name `nth_element`.

The algorithm can also be made to always work in linear time.

5 Complexity notations

This chapter discusses the notations used to describe the asymptotic behaviour of algorithms.

Θ is defined together with two other related useful notations O and Ω.

5.1 Asymptotic notations

The equation for the running time was simplified earlier significantly:

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

$\Rightarrow$ studying the behaviour of the algorithm as the size of its input increases to infinity

- i.e. the *asymptotic* efficiency of algorithms

$\Rightarrow$ usefull information **only with inputs larger than a certain limit**

- often the limit is rather low

  $\Rightarrow$ the algorithm fastest according to Θ- and other notations is the fastest also in practice, except on very small inputs

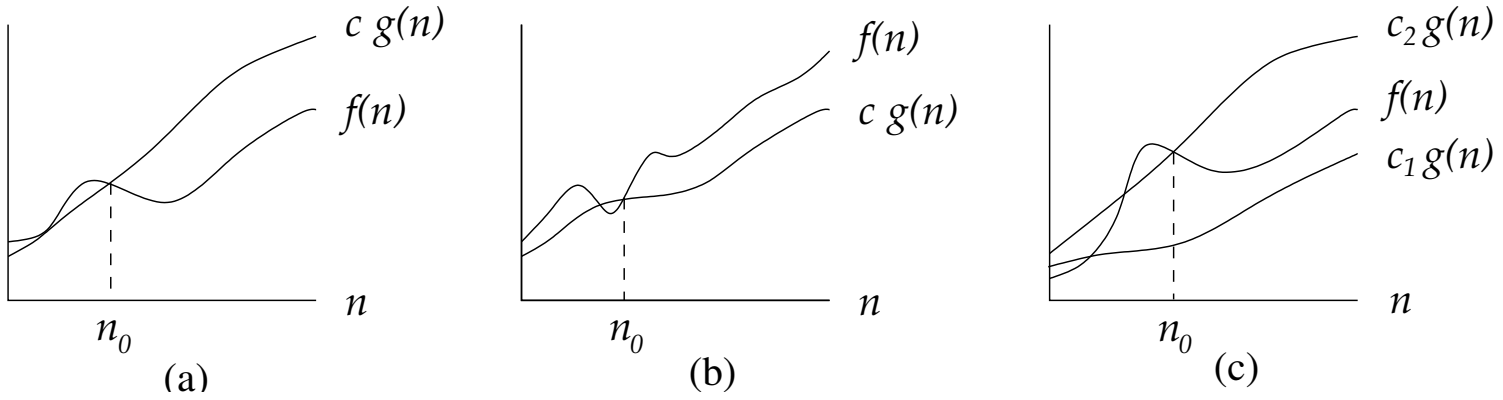
Θ-notation

- 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
    $\Rightarrow$ we should, and earlier did, write $f(n) \in \Theta(g(n))$, but usually $=$ is 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 $c_1 = 3$, $c_2 = 4$ ja $n_0 = 4$
- $0 \leq 3n^2 \leq 3n^2 + 5n - 20 \leq 4n^2$ 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 fullfill the requirements

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

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 = \Theta(n^k)$$

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

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

- $\Theta(1)$ doesn't indicate which variable is used in the analysis
  $\Rightarrow$ 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 Θ-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 $n^2 \neq \Theta(n^3)$
- 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 quaranteed 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).

We are often interested in the upper bound only.

For example: INSERTION-SORT

line	efficiency
for j := 2 **to** $A.length$ **do**	$O(n)$
key := $A[j]$	$\cdot \quad O(1)$
i := $j-1$	$\cdot \quad O(1)$
while $i > 0$ and $A[i] > key$ **do**	$\cdot \quad O(n)$
$A[i+1]$:= $A[i]$	$\cdot \quad \cdot \quad O(1)$
i := $i-1$	$\cdot \quad \cdot \quad O(1)$
$A[i+1]$:= key	$\cdot \quad O(1)$

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.
$\Rightarrow$ *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 usefull in practise mostly in situations where even the best-case efficiency of the solution is unsatisfactory and the result can be rejected straightaway

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:

$$\begin{array}{ll} f(n) = O(g(n)) & a \leq b \\ f(n) = \Theta(g(n)) & a = b \\ f(n) = \Omega(g(n)) & a \geq b \end{array}$$

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: $a < b$, $a = b$ ja $a > b$. However, this does not hold for all asymptotic notations.
$\Rightarrow$ Not all functions are asymptotically comparable to eachother (e.g. n and $n^{1+\sin n}$).

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 algorithms 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 $O(n)$
  $\Rightarrow$ a Θ-value common to all cases cannot be determined.

An example

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

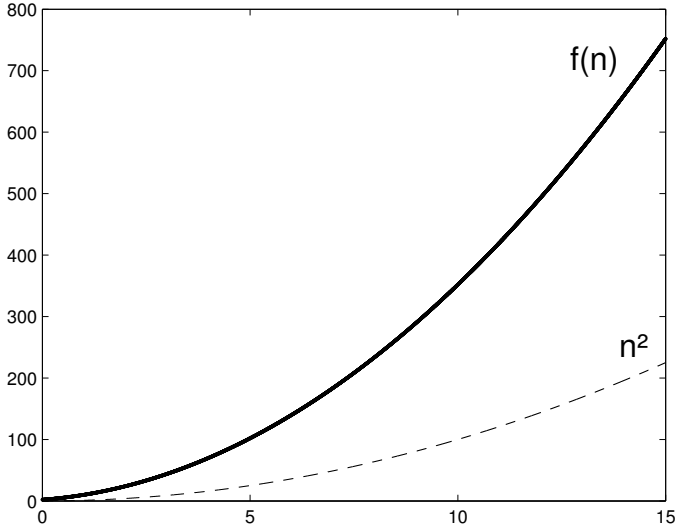
and simplify it according to the rules given earlier:

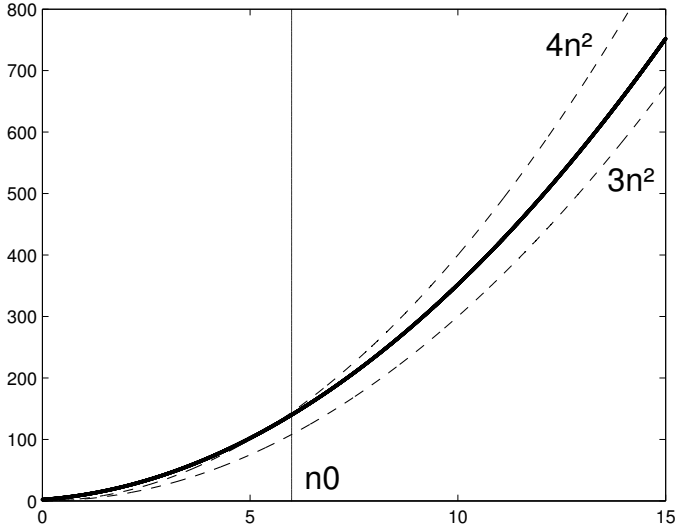
- lower-order terms ignored
- constant coefficients ignored

$\Rightarrow f(n) = \Theta(n^2)$

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 $n \geq 6$
$\Rightarrow c_1 = 3$, $c_2 = 4$ and $n_0 = 6$ work correctly
$\Rightarrow f(n) = O(n^2)$ and $\Omega(n^2)$
$\Rightarrow f(n) = \Theta(n^2)$



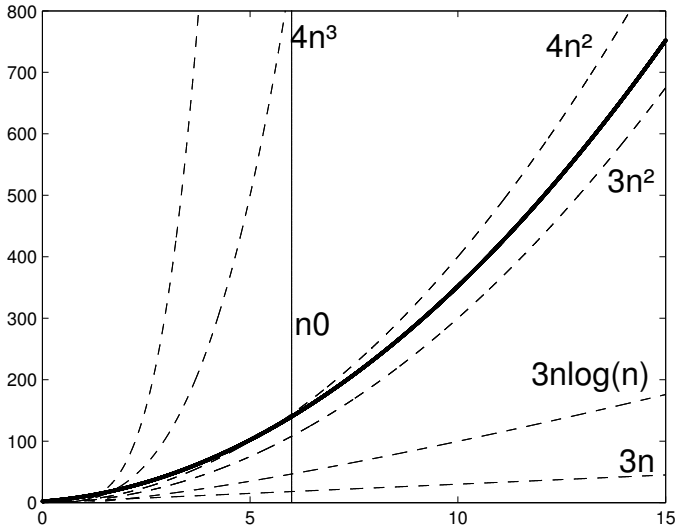


Clearly the constant $c_2 = 4$ works also when $g(n) = n^3$, since when $n \geq 6, n^3 > n^2$
$\Rightarrow f(n) = O(n^3)$

- the same holds when $g(n) = n^4$...

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 $g(n) = \lg n$



5.2 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 bandwidth usage

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

- 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
  $\Rightarrow$ We're left with the precision of the asymptotic notations.

$\Rightarrow$ 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 **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 (8bits) 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 $n \cdot$ <element size>

$\Rightarrow$ Estimating the exact memory use is often possible, although requires precision.

Asymptotic notations are usefull 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.

- $\Theta(1)$ extra memory vs. $\Theta(n)$ extra memory
- 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.

5.3 Binary Search

An efficient search in sorted data

Works by comparing a search key to the middle element in the data

- divide the search area into two
- choose the half where the key must be, ignore the other
- continue until only one element left
  - it is the key
  - the element is not in the data set

```
BIN-SEARCH(A, 1, n, key)        ▷ requirement: n ≥ 1, the array is sorted
1   low := 1; hi := n           (initialize the search to cover the entire array)
2   while low < hi do           (search until there are no more elements to cover)
3       mid := ⌊(low + hi)/2⌋   (divide the search area in half)
4       if key ≤ A[mid] then    (If the key is in the bottom half...)
5           hi := mid           (...choose the bottom half as the new search area)
6       else                    (else...)
7           low := mid + 1      (...choose the upper half as the search area)
8   if A[low] = key then
9       return low              (the element is found)
10  else
11      return 0                (the element is not found)
```

- The running time of BIN-SEARCH is $\Theta(\lg n)$.

6 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 purposefull and efficient.

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

6.1 General information on the C++ standard library

The standard library, commonly called STL, was standardized together with the C++-language autumn 1998. The latest version of the standard C++14 is from August 2014.

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

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 $\mathcal{O}$-notation.

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

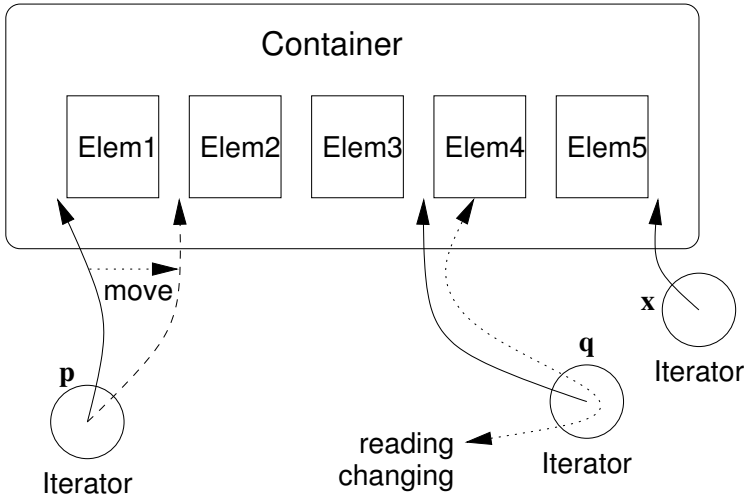
Interesting elements in the standard library from the point of view of a data structures course are the *containers*, i.e. the data structures provided by the library and the *generic algorithms* together with the *iterators*, with which the elements of the containers are manipulated.

Lambdas were introduced with C++11 and play a key role as well.

6.2 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

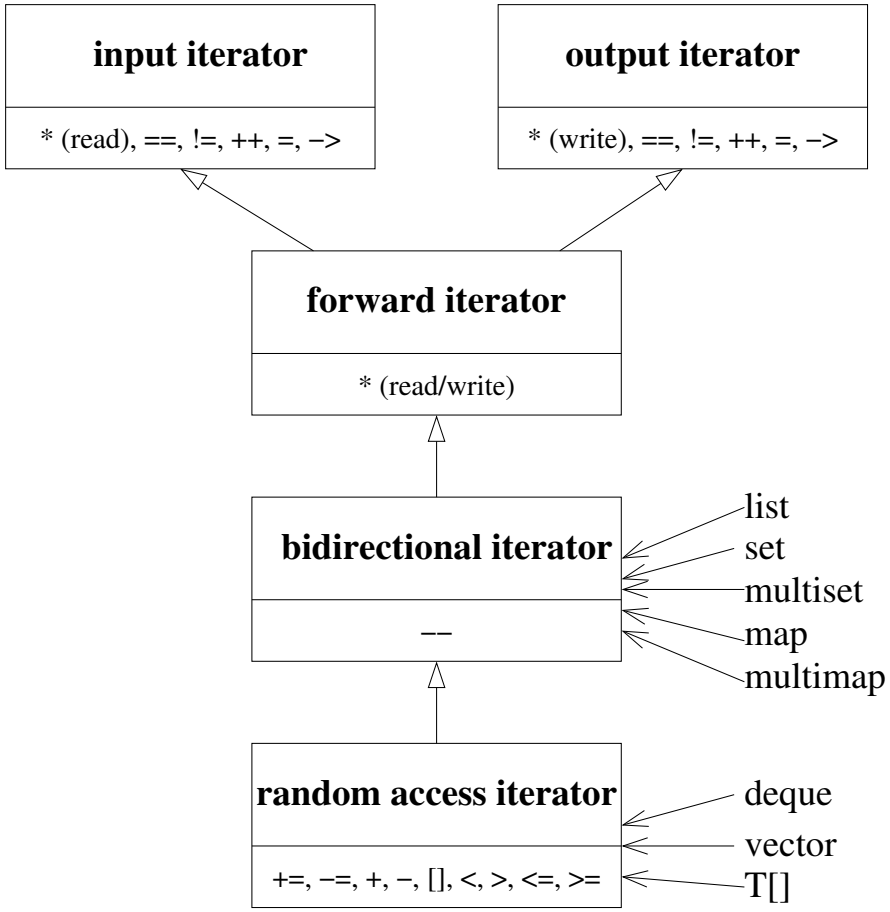
- the location of the elements added to or removed from a container is usually indicated with iterators

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 quarantee 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!=`)



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

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

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

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

```
container<type stored>::iterator p;
```

The key word `auto` is useful with iterators:

```
auto p = begin( cont );
// → std::vector<std::string>::iterator
```

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

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 iterator*s 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)

- *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, ','`)
- *move iterators* replace the copying of an element with a move.

6.3 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 to the container to the location determined by their *key*
  - by default the operator < can be used to compare the key values of the elements in ordered containers
- the interface's member functions indice how the container is supposed to be used.

The containers:

Container type	Library

Sequences	`array`
	`vector`
	`deque`
	`list`
	(`forward_list`)

Assosiative containers	`map` `set`
unordered assosiative	`unordered_map` `unordered_set`
Adapters	`queue` `stack`

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_pointer` is available in C++11 for easier management of memory is situations where several objects share a resource
  - contains an inbuild reference counter and deletes the element once the counter becomes zero
  - there is no need to call delete. Furthermore is must not be called.
  - definition: `auto pi = std::make_shared<Olio>(params);`
  - handy especially if a data structure ordered based on two different keys is needed
    * store the satellite data with a `shared_pointer`
    * store the pointers in two different containers based on two different keys

Sequence containers:

Array `array<type>` is a constant sized array.

- Initialization `std::array<type, size> a = {val, val,...};`
- Can be indexed with `.at()` or with `[]`. Functions `front()` and `back()` access the first and the last element.
- Provides iterators and reverse iterators.
- `empty()`, `size()` and `max_size()`
- The function `data()` accesses the underlying array.

Operations are constant time except `fill()` and `swap()` which are $O(n)$

Vector `vector<type>` is an flexible sized array where additions and removals are efficient at the end

- Initialization `vector<int> v {val, val, ...};`
- Constant time indexing with `.at()`, `[]` and a (amortized) constant time addition with `push_back()` and removal with `pop_back()` at the end of the vector.
- Inserting an element elsewhere with `insert()` and removal with `erase` is linear, $O(n)$
- `emplace_back(args);` builds the element directly into the vector
- The size can be increased with `.resize(size, initial_val);`
  - the initial value is voluntary
  - if needed, the vecor can allocate more memory automatically
  - memory can also be reserved in advance: `.reserve(size)`, `.capacity()`
- 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 `vector<bool>` which is different from the general implementation the templates would create in order to save memory
  - 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>` is an array open at both ends

- initialization: `deque<type> d {val, val, val...};`
- 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(val)`, `.emplace_front(args)`, `.pop_front()`
- 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 is a container that support bidirectional iteration

- initialization: `list<type> l {val, val, val };`
- 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, list2)` makes the other list a part of the second list infront of the location
  - `.splice(location, list, val)` moves an element from another list or the same list infront of the location
  - `.splice(location, list, beg, end)`
  - `.merge(list2)` and `.sort()`, stable, $O(nlogn)$ on average
  - `.reverse()`, linear

Amortized running time

Vector is a flexible sized array, i.e. the size increases when needed

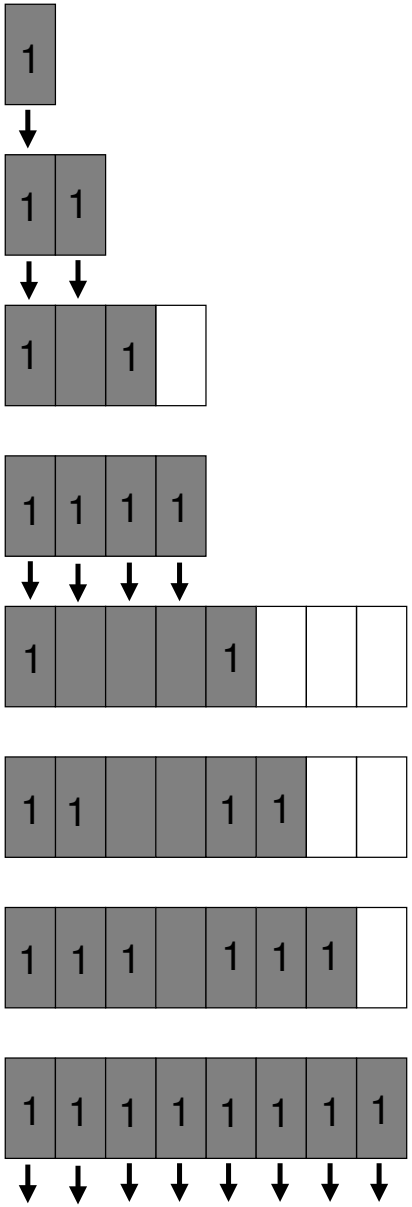
- when a new element no longer fits into the array, a new, larger one is allocated and all the elements are moved there
- the array never gets smaller $\Rightarrow$ the memory allocation is only reduced when a new contents are copied on top of the old one

$\Rightarrow$ Adding an element to the end of the vector was mentioned as *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 preceeded 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
- now the inexpensive operations are still constant time, although they are a constant coefficient more inefficient than in reality
- the savings can be used to pay for the expensive operation

$\Rightarrow$ all addition operations at the end of a vector can be done in 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 $i - \frac{1}{2} \cdot vector.capacity()$
- when the size of the array needs to be increased, each element has one credit saved and the expensive copying can be payed for with them

Associative containers:

`set<type>` and `multiset<type>` is a 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
- 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 $\neg(x<y \vee y<x)$

  $\Rightarrow$ a sensible and efficient definition to "<" needs to be given
- the same element can be in the multiset several times, in the set the elements are unique
- 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:
  - `.find(val)` finds the element (first of many in the multiset) or returns `.end()` if it isn't found
  - `.lower_bound(val)` finds the first element $\geq$ element
  - `.upper_bound(val)` finds the first $>$ element
  - `.equal_range(val)`returns `make_pair( .lower_bound(val), .upper_bound(val) )` but needs only one search (the size of the range is 0 or 1 for the set)
  - for sets `insert` returns a pair (location, added), since elements already in the set may not be added
- the standard quarantees that the iterators are not invalidated by the addition or removal (except of course to the elements removed)

`map<key_type, val_type>` and
`multimap<key_type, val_type>`

- store (key, satellite data) pairs
  - the type of the pair is `pair<type1, type2>`
  - a pair can be created with the function `make_pair`
  - the fields of the pair are returned by `.first(), .second()`
- initialization `std::map<key_type, val_type> m {{key1, val1}, {key2, val2}, key3, val3},...};`
  e.g.
  ```
  std::map<std::string,int> anim { {"bear",4}, {"giraffe",2}, {"tiger",7} };
  ```
- `map` can be exceptionally indexed with the key $O(nlogn)$
  - If the key is not found, a new valuepair key-type is added to the container
- the iterators are not invalidated by the addition or removal

Hash tables Unordered set/multiset is a structure that contains a set of elements and unodered map/multimap contains a set of key-value pairs.

- the interfaces of unordered-map/set resemble map and set
- the most significant differences:
  - the elements are not ordered (unordered)
  - addition, removal and searching is on average constant time and in the worst-case linear
  - a set of member function valuable for hashing, such as `rehash(size)`, `load_factor()`, `hash_function()` ja `bucket_size()`.
- the size of the hash table is automatically increased in order to keep the load factor of the buckets under a certain limit
  - changing the size of the hash table *(rehashing)* is on average linear, worst-case quadratic
  - rehashing invalidates all iterators but not pointers or references

Additionally other containers are found in the Standard library:

`bitset<bit_amount>`

- `#include<bitset>`
- for handling fixed sized binary bitsets
- provides typical operations (AND, OR, XOR, NOT)

Strings `string`

- `#include <string>`
- the character string has been optimized for other purposes in C++ and they are usually not percieved as containers. However, they are, in addition to other purposes, also containers.
- store characters but can be made to store other things also
- they provide, among others, iterators, `[...]`, `.at(...)`, `.size()`, `.capacity()` and `swap`
- string can get very large and automatically allocate new memory when necessary

- 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>`

- provides in addition to the normal class-operations only
  - stack-operations, `.push(...)`, `.top()`, `.pop()`
  - 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 based on different containers
  - any container that provides `back()`, `push_back()` and `pop_back()` is usable. Especially `vector`, `list` and `deque`.
  - `stack<type> basic_stack;` (`deque`)
  - `stack<type, list<type> > list_stack;`

Queue `queue<element_type, container_type>`

- queue operations `.push(...)`, `.pop()`, `.front()`, `.back()`(!)
- otherwise more or less like the stack

Priority queue `priority_queue<element_type, container\_type>`

- has an almost identical interface as the queue
- implemented with a heap
- any container that provides `front()`, `push_back()` and `pop_back()` and random access iteration can be used. Especially `vector` (default) and `deque`
- 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 <> or as the constructor parameter

data-structure	add to end	add elsewhere	remove 1st elem.	remove elem.	nth elem. (index)	search elem.	remove largest
array					$O(1)$	$O(n)_{[2]}$	
vector	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
list	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[3]}$
deque	$O(1)$	$O(n)_{[4]}$	$O(1)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
stack$_{[9]}$	$O(1)$			$O(1)_{[5]}$			
queue$_{[9]}$		$O(1)_{[6]}$		$O(1)_{[7]}$			
priority queue$_{[9]}$		$O(\log n)$ [10]					$O(\log n)_{[8]}$
set (multiset)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
map (multimap)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
unordered_ (multi)set		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$
unordered_ (multi)map		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$

[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] query constant time, removal logarithmic
[9] container adapter
[10] addition is done automatically based on the heap order

6.4 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 programmer can implement iterators for his/her own data structures which makes using algorithms possible also with them

Not all algorithms can be applied to all data structures efficiently.
$\Rightarrow$ part of the algorithms accept only iterators from one iterator category as parameters.

- this ensures the efficiency of the algorithms since all operations the iterator provides are constant time
- if the iterator is of an incorrect type, a compile-time error is given

  $\Rightarrow$ if the algorithm is given a data structure for which it cannot be implemented efficiently, the program won't even compile

The standard library algorithms are in `algorithm`. In addition the standard defines the C-language library `cstdlib`.

Algorithms are divided into three main groups: Non-modifying sequence operations, modifying sequence operations and sorting and related operations.

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`$(first, \quad end, \quad value)$ 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

  $\Rightarrow$ `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

Sorting algorithms

- `sort(`beg`,` end`)` and `stable_sort(`beg`,` end`)`
- the running-time of `sort` is $O(nlogn)$ and `stable_sort` is $O(nlogn)$ is enough memory is available and $O(nlog^2n)$ otherwise
- the sorting algorithms require random access iterators as parameter

  $\Rightarrow$ 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`)`
- in addition `is_sorted(`beg`,` end`)` and `is_sorted_until(`beg`,` end`)`

`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
- in addition `is_partitioned` and `partition_point`

`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
- 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

- `pop_heap(` $first$`,` end `)` makes the topmost element the last (i.e. to the location $end - 1$) and executes HEAPIFY to the range $first \ldots end - 1$
  - cmp. HEAP-EXTRACT-MAX
- `make_heap(` $first$`,` end`)` BUILD-HEAP
- `sort_heap(` $first$`,` end `)` HEAPSORT
- the iterators must be random-access
- in addition `is_heap` and `is_heap_until`

Set operations

- The C++ standard library contains functions that support this
- `includes(` $first1$, $end1$, $first2$, $end2$ `)` subset $\subseteq$
- `set_union(` $first1$, $end1$, $first2$, $end2$, $result$ `)` union $\cup$
- `set_intersection(...)` intersection $\cap$
- `set_difference(...)` difference -
- `set_symmetric_difference(...)`
- $first$- and end-iterators are input iterators and $result$ is an output iterator

`find_first_of(` $first1$, $end1$, $first2$, $end2$ `)`

- 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 , . . .

- a simple implementation is $\Theta(nm)$ in the worst case where n and m are the lengths of the queues
- the second queue is scanned for each element in the first queue
  – the slowest case is when nothing is found
  $\Rightarrow$ slow if both queues are long
- the implementation could be made simple, efficient and memorysaving by requiring that the queues are sorted

NOTE! None of the STL algorithms automatically make additions or removals to the containers but only modify the elements in them

- for example `merge` doesn't work if it is given an output iterator into the beginning of an empty container
- if the output iterator is expected to make additions instead of copying the iterator adapter insert iterator must be used (chapter 6.2)

6.5 Lambdas: ()(){}

Situations where a need to pass functionality on to the functions arise often with the algorithm library

- e.g. `find_if`, `for_each`, `sort`

Lambdas are nameless functions of an undefined type. They take parameters, return a value and can refer to the creation environment and change it.

Syntax: `[environment](parameters)->returntype {body}`

- `environment` is empty if the lamba does not refer to its environment
- `parameter` can be left out
- if there is no `->returntype` it is void. It can also be deducted from a simple return statement.
- e.g. `[](int x, int y){ return x+y;}`
  `for_each( v.begin(), v.end(),[] (int val) {cout<<val<<endl;});`
  `std::cin >> lim;` //local var
  `std:find_if(v.begin(), v.end(),[lim](int a){return a<lim;});`

STL algorithms can be seen as named special loops whose body is given in the lambda.

```
bool all = true;
for (auto i : v)
{
    if (i%10 != 0) {
        all = false;
        break;
    }
}
if (all) {...}

if (std::all_of(v.begin(), v.end(), [](int i){return i%10==0;}){...}
```

6.6 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 currect going price on the market. To make this easier the union of real estate agents decides to start collecting uptodate 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:

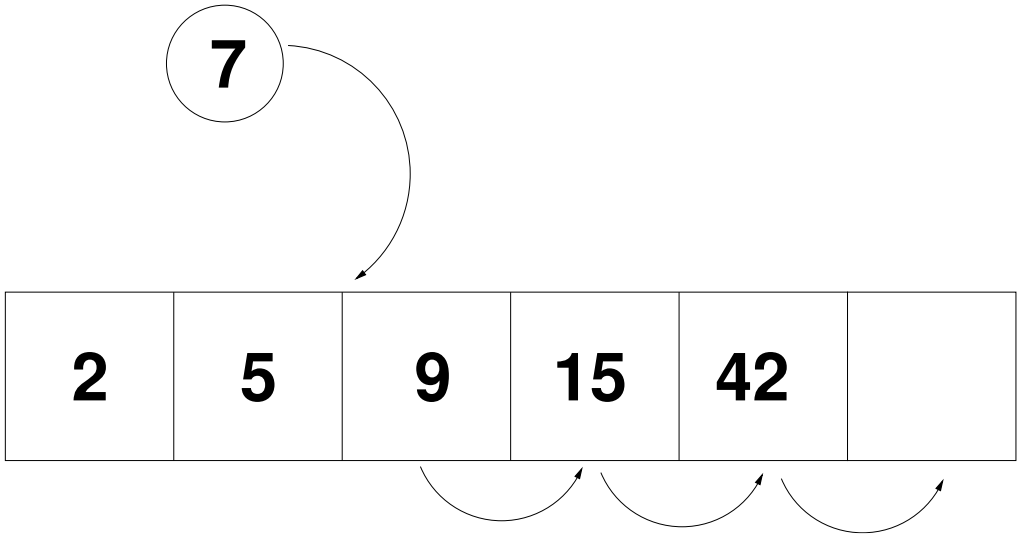
Knowledge of the customer's purchases is stored into the customer register of a supermarket of a large chain. The program must be able to check efficiently if for example Teemu Teekkari has earlier bought frozen pizza from the market or if he is a new customer and store the information.

At the second stage the information of separate markets is stored in ascii-form and sent onto the central computer of the supermarket chain each night. The central computer combines the information into one file.

7 Interlude: Is keeping the data sorted worth it?

When a sorted range is needed, one idea that comes to mind is to keep the data stored in the sorted order as more data comes into the structure

Is this an efficient approach to problems needing a sorted range?



- scanning through a sorted data range can be stopped once an element larger than the one under search has been met

  $\Rightarrow$ it's enough to scan through approximately half of the elements

  $\Rightarrow$ scanning becomes more efficient by a constant coefficient

- in the addition, the correct location needs to be found, i.e. the data needs to be scanned through

  $\Rightarrow$ addition of a new element into the middle of the data range is $\Theta(n)$

  $\Rightarrow$ addition becomes $\Theta(n^2)$

$\Rightarrow$ it's usually not worth the effort to keep the data sorted unless it's somehow beneficial to the other purposes and a data structure with constant time insert can be chosen

- if the same key cannot be stored more than once the addition requires searching anyway

  $\Rightarrow$ mainting the order becomes beneficial in a data structure with a constant time insert

8 Tree, Heap and Priority queue

This chapter deals with a design method *transform and conquer*

The notion of a *binary tree* and a *heap* are introduced

A sort based on the construction of a heap tree (HEAPSORT) is investigated

A *priority queue*, a set of elements with an orderable characteristic – a priority, is discussed.

8.1 Algorithm Design Technique: Transform and Conquer

Transform and conquer is a design technique that

- First modifies the problem's instance to be more amenable to solution – the transformation stage
- Second the problem is solved – the conquering stage

There are three major variations in the way the instance is transformed

- *Instance simplification*: a simpler or more convenient instance of the same problem
- *Representation change*: a different representation of the same instance
- *Problem reduction*: an instance of a different problem for which an algorihtm is already available.

8.2 Sorting with a heap

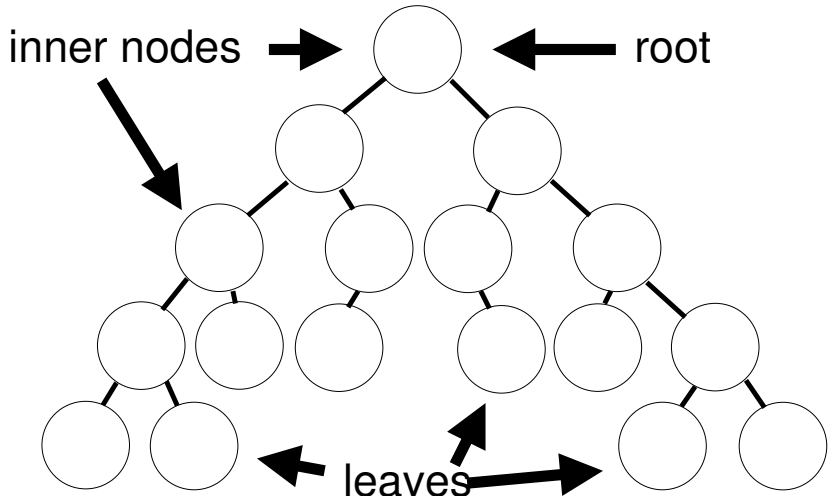
Let's discuss a sorting algorithm HEAPSORT that uses a very important data structure, a *heap*, to manage data during execution.

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 chidren
- the children are called *left* and *right*
- a node is the *parent* of its children
- a childless node is called a *leaf*, and the other nodes are *internal node*s
- a binary tree has at most one node that has no parent, i.e. the *root*
  - all other nodes are the root's children, grandchildren etc.

- the descendants of each node form the *subtree* of the binary tree with the node as the root
- The *height* of a node in a binary tree is the length of the longest simple downward path from the node to a leaf
  - the edges are counted into the height, the height of a leaf is 0
- the height of a binary tree is the height of it's root



- a binary tree is *completely balanced* if the difference between the height of the root's left and right subtrees is atmost one and the subtrees are completely balanced
- the height of a binary tree with n nodes is at least $\lfloor \lg n \rfloor$ and atmost n - 1

  $\Rightarrow O(n)$ and $\Omega(\lg n)$

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

```
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  if x ≠ NIL then
2      INORDER-TREE-WALK(x→left)
3      process the element x
4      INORDER-TREE-WALK(x→right)
```

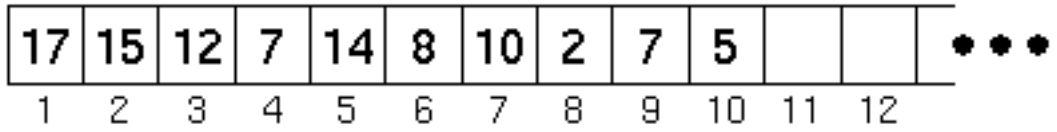
- *postorder*
  - in the example 3, 6, 5, 9, 8, 14, 15, 13, 23, 22, 26, 32, 35, 33, 30, 25, 18

```
POSTORDER-TREE-WALK(x)
1  if x ≠ NIL then
2       POSTORDER-TREE-WALK(x→left)
3       POSTORDER-TREE-WALK(x→right)
4       process the element x
```

- running-time $\Theta(n)$
- extra memory consumption = Θ(maximum recursion depth) = $\Theta(h+1) = \Theta(h)$

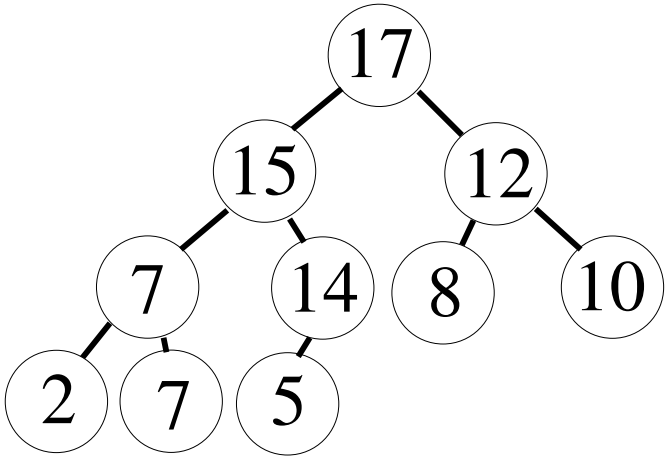
Heap

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 $2i + 1 \leq n$).



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 $\lfloor \frac{i}{2} \rfloor$



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 $\Theta(1)$

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:

$A[\text{PARENT}(i)] \geq A[i]$ *always when* $2 \leq i \leq A.heapsize$

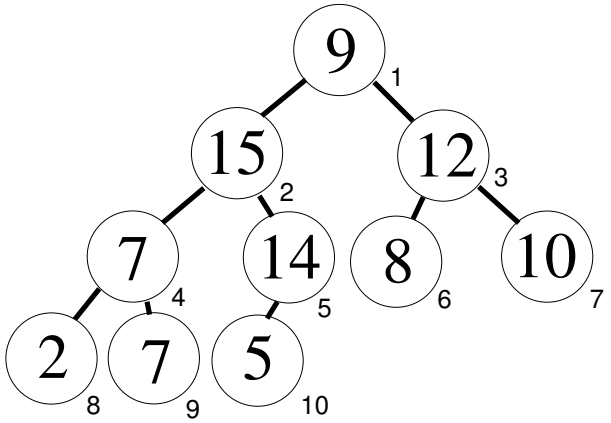
- $A.heapsize$ gives the size of the heap (we'll later see that it's not necessarily the size of the array)

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

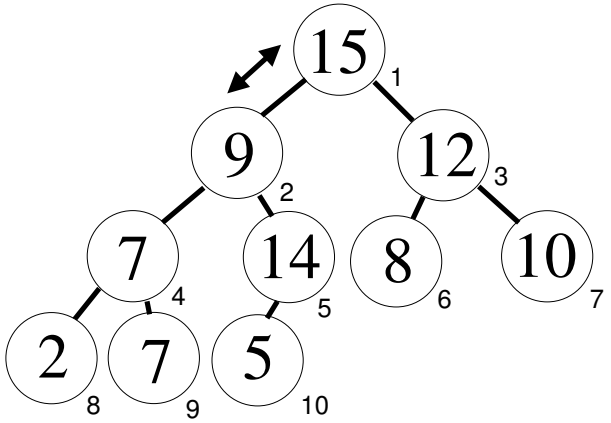
If the height of the heap is h, the amount of its nodes is between $2^h \ldots 2^{h+1} - 1$.
$\Rightarrow$ If there are n nodes in the heap its height is $\Theta(\lg n)$.

Adding an element from the top of the heap:

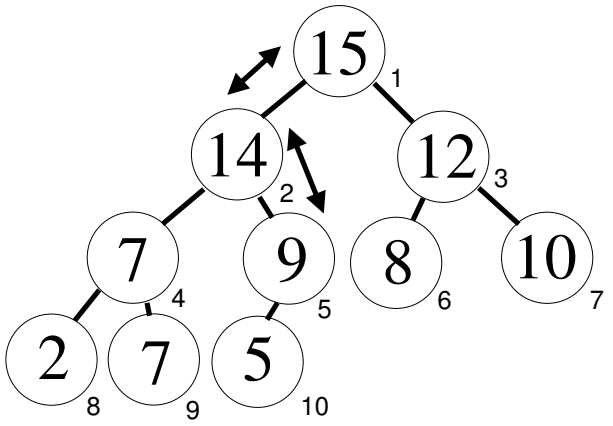
- let's assume that $A[1 \ldots n]$ is otherwise a heap, except that the heap property does not hold for the root of the heap tree
  - in other words $A[1] < A[2]$ or $A[1] < A[3]$



- the problem can be moved downwards in the tree by selecting the largest of the root's children and swapping it with the root
  - in order to maintain the heap property the largest of the children needs to be chosen - it is going to become the parent of the other child



- the same can be done to the subtree, whose root thus turned problematic and again to its subtree etc. until the problem disappears
  - the problem is solved for sure once the execution reaches a leaf
    $\Rightarrow$ the tree becomes a heap



The same as pseudocode

```
HEAPIFY(A, i)                    (i is the index where the element might be too small)
1  repeat                                          (repeat until the heap is fixed)
2      old_i := i                                  (store the value of i)
3      l := LEFT(i)
4      r := RIGHT(i)
5      if l ≤ A.heapsize and A[l] > A[i] then      (the left child is larger than i)
6          i := l
7      if r ≤ A.heapsize and A[r] > A[i] then      (right child is even larger)
8          i := r
9      if i ≠ old_i then                           (if a larger child was found...)
10         exchange A[old_i] ↔ A[i]                (...move the problem downwards)
11 until i = old_i                                 (if the heap is already fixed, exit)
```

- 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 $O(h) = O(\lg n)$.

Building a heap

- the following algorithm converts an array into a heap:

```
BUILD-HEAP(A)
1  A.heapsize := A.length          (the heap is built out of the entire array)
2  for i := n⌊A.length/2⌋ downto 1 do   (scan through the lower half of the array )
3      HEAPIFY(A, i)               (call Heapify)
```

- 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

- BUILD-HEAP executes the **for**-loop $\lfloor \frac{n}{2} \rfloor$ times and HEAPIFY is $\Omega(1)$ and $O(\lg n)$ 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 $\lfloor \frac{n}{2} \rfloor \cdot O(\lg n) + \Theta(n) = O(n \lg n)$
- 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

level	h	number of HEAPIFY calls
lowest	0	0
2nd	1	$\lfloor \frac{n}{4} \rfloor$
3rd	2	$\lfloor \frac{n}{8} \rfloor$
...	...	...
topmost	$\lfloor \lg n \rfloor$	1

  - thus the worst case runnign time is $\frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \cdots = \frac{n}{2} \cdot \Sigma_{i=1}^{\infty} \frac{i}{2^i} = \frac{n}{2} \cdot 2 = n \Rightarrow O(n)$

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

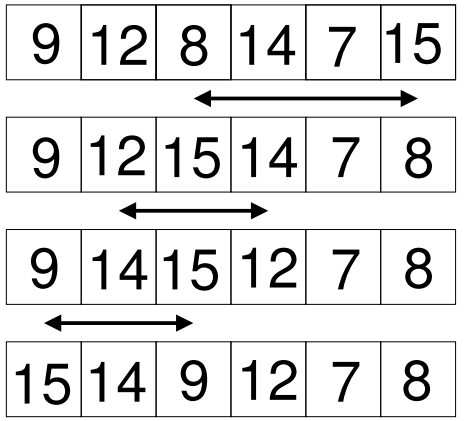
Sorting with a heap

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

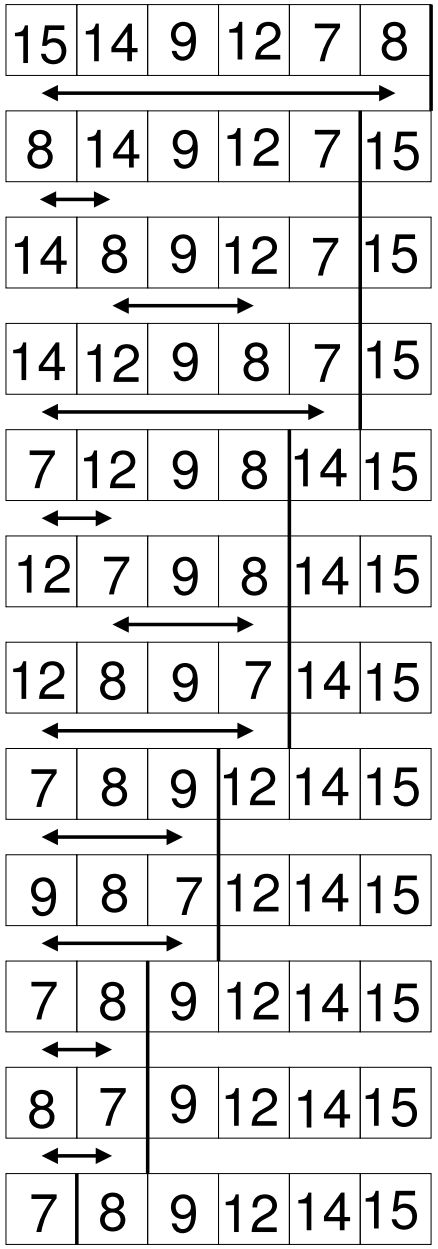
```
HEAPSORT(A)
1  BUILD-HEAP(A)                          (convert the array into a heap)
2  for i := A.length downto 2 do          (scan the array from the last to the first element)
3      exchange A[1] ↔ A[i]               (move the heap's largest element to the end)
4      A.heapsize := A.heapsize − 1       (move the largest element outside the heap)
5      HEAPIFY(A, 1)                      (fix the heap, which is otherwise fine...)
                                          (... 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 laborous
  - 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



The running time of HEAPSORT consists of the following:

- 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
  - HEAPIFY 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 HEAPIFY 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 abowe assume that the data structure used to store the heap provides a constant time indexing.

- Heap is worth using only when this is true

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

8.3 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:

- INSERT(S, x) inserts the element x into the set S
- 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
- EXTRACT-MAX(S) removes and returns the element with the largest key
- alternatively the operations MINIMUM(S) and EXTRACT-MIN(S) can be implemented
  - there can be **only the maximum** or **only the minimun** 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
    $\Rightarrow$ 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 practise in an algorithm for finding shortest paths, covered later

In practise, 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 $\Theta(1)$ running-time

```
HEAP-MAXIMUM(A)
1  if A.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)
```

- EXTRACT-MAX(S) can be implemented by fixing the heap after the extraction with HEAPIFY.
- HEAPIFY dominates the running-time of the algorithm: $O(\lg n)$.

```
HEAP-EXTRACT-MAX(A)
1  if A.heapsize < 1 then          (no maximum in an empty heap)
2      error "heap underflow"
3  max := A[1]                     (the largest element is at the first index)
4  A[1] := A[A.heapsize]           (make the last element the root)
5  A.heapsize := A.heapsize − 1    (decrement the size of the heap)
6  HEAPIFY(A, 1)                   (fix the heap)
7  return max
```

- INSERT(S, x) adds a new element into the heap by making it a leaf and then by lifting it to its correct height based on its size
  - is works like HEAPIFY, but from bottom up
  - in the worst-case, the leaf needs to be lifted all the way up to the root: running-time $O(\lg n)$

HEAP-INSERT(A, 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[PARENT(i)] < key do (continue until the root or ...)
                                         (... or a parent with a larger value is reached)
4      A[i] := A[PARENT(i)]              (move the parent downwards)
5      i := PARENT(i)                    (move upwards)
6  A[i] := key                           (place the key into its correct location)
```

$\Rightarrow$ 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)

$\Rightarrow$ It's easy to maintain and change the implementation when needed without needing to change the code using the queue.

9 Different Types of Sorting Algorithms

This chapter discusses sorting algorithms that use other approaches than comparisons to determine the order of the elements.

The maximum efficiency of comparison sorts, i.e. sorting based on the comparisons of the elements, is aldo examined.

Finally, the chapter covers the factors for choosing an algorithm.

9.1 Other Sorting Algorithms

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

- They determine the correct order of elements based on comparing their values to eachother.

It is possible to use other information besides comparisons to sort data.

Sorting by counting

Let's assume that the value range of the keys is small, atmost the same range than the amount of the elements.

- For simplicity we assume that the keys of the elements are from the set $\{1, 2, \ldots, k\}$, and $k = O(n)$.
- The amount of elements with each 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
2      C[i] := 0                          (initialize a temp array C with zero)
3  for j := 1 to A.length do
4      C[A[j].key] := C[A[j].key] + 1     (calculate the amount of elements with key = i)
5  for i := 2 to k do
6      C[i] := C[i] + C[i − 1]            (calculate how many keys ≤ i)
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 reverse order to quarantee stability.

Running-time:

- The first and the third **for**-loop take $\Theta(k)$ time.
- The second and the last **for**-loop take $\Theta(n)$ time.

$\Rightarrow$ The running time is $\Theta(n + k)$.

- If $k = O(n)$, the running-time is $\Theta(n)$.
- All basic operations are simple and there are only a few of them in each loop so the constant coefficient is also small.

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

- The memory consumption of the algorithm is $\Theta(k)$.
- 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 seeral keys or a key with several parts.

- the list of exam results: sort first based on the department and then those into an alphabetical order
- dates: first based on the year, then the month, and the day
- a deck of cards: first based on the suit and then those 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 differ 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 QUICKSORT or MERGESORT). Example: comparing dates:

```
DATE-COMPARE(x, y)
1  if x.year < y.year then return "smaller"
2  if x.year > y.year then return "greater"
3  if x.month < y.month then return "smaller"
4  if x.month > y.month then return "greater"
5  if x.day < y.day then return "smaller"
6  if x.day > y.day then return "greater"
7  return "equal"
```

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 elements and thus COUNTING-SORT can be used.

There are two different algorithms available for sorting with multiple keys.

- 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
  - 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

LSD-RADIX-SORT(A, d)
1 **for** i := 1 **to** d **do** *(run through the criteria, least significant first)*
2 ▷ sort A with a stable sort according to criterion i

- 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 LSD-RADIX-SORT
    $\Rightarrow$ the algorithm is not given here

The efficiency of RADIX-SORT when using 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 is 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, $k^d \geq n \Rightarrow d \geq log_k\ n$ = $\Theta(\lg n)$
  $\Rightarrow$ RADIX-SORT is $\Theta(dn) = \Theta(n \lg n)$, 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 eachother.

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.

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 := A.length
2  for i := 1 to n do                          (go through the elements)
3      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] := EXTRACT-MIN(B[i])           (... by moving the elements, smallest first...)
8          k := k + 1                          (... into the correct location in the result array)
```

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 the bucket is ≈ 1
  - however it is possible that every element end up in the same bucket

  $\Rightarrow$ an implementation that uses a heap would require $\Theta(n)$ for each bucket,s $\Theta(n^2)$ in total
- On the other hand, the implementation does not need to be very efficient for large buckets since they are rare. $\Rightarrow$ In practise the buckets should be implemented as lists.
  - INSERT links the incoming element to its correct location in the list, Θ(list length) time is used
  - EXTRACT-MIN removes and returns the first element in the list, $\Theta(1)$ time is used

the average efficiency of BUCKET-SORT:

- We assumed the keys are evenly distributed.
  $\Rightarrow$ 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 on average one element in the bucket.
- EXTRACT-MIN is constant time.

$\Rightarrow$ The total running-time is $\Theta(n)$ on average.

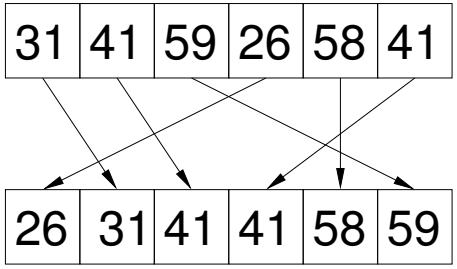
In the slowest case all elements fall into the same bucket in an ascending order.

- INSERT takes a linear amount of time

$\Rightarrow$ The total running-time is $\Theta(n^2)$ in the worst-case.

9.2 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.
- The amount of comparisons needed must find the only correct alternative.
- 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
  - ...
  - k comparisons in enough to pick the right alternative from atmost 2^k

    $\Rightarrow$ choosing the right one from x alternatives requires at least $\lceil \lg x \rceil$ comparisons

- If the size of the array is n, there are $n!$ permutations
  $\Rightarrow$ At least $\lceil \lg n! \rceil$ comparisons is required
  $\Rightarrow$ a comparison sort algorithm needs to use $\Omega(\lceil \lg n! \rceil)$ time.

How large is $\lceil \lg n! \rceil$?

- $\lceil \lg n! \rceil \geq \lg n! = \Sigma_{k=1}^{n} \lg k \geq \Sigma_{k=\lceil \frac{n}{2} \rceil}^{n} \lg \frac{n}{2} \geq \frac{n}{2} \cdot \lg \frac{n}{2} = \frac{1}{2} n \lg n - \frac{1}{2} n = \Omega(n \lg n) - \Omega(n) = \Omega(n \lg n)$
- on the other hand $\lceil \lg n! \rceil < n \lg n + 1 = O(n \lg n)$
  $\Rightarrow \lceil \lg n! \rceil = \Theta(n \lg n)$

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

- On the other hand HEAPSORT and MERGE-SORT are $O(n \lg n)$ in the slowest case.

  $\Rightarrow$ *In the slowest case sorting based on comparisons between elements is possible in* $\Theta(n \lg n)$ *time, but no faster.*
- HEAPSORT and MERGE-SORT have an optimal asymptotic running-time in the slowest case.
- Sorting is truly asymptotically more time consuming than finding the median value, which can be done in the slowest possible case in $O(n)$.

9.3 Choosing an algorithm

The key 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 and dynamic arrays 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 overall 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: contacts

- operations
  - finding a phonenumber based on the name
  - adding a new contact into the phone-book
  - removing a contact and all the related information
- assumptions
  - additions and removals are needed rarely
  - phonenumber queries are done often
  - additions and removals are done in groups

1st attempt: unsorted array

Virtanen	Järvinen		Lahtinen		
123 555	123 111		123 444		
1	2		n		

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

$\Rightarrow$ 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 BIN-SEARCH: $\Theta(\lg n)$.

$\Rightarrow$ 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 $O(1)$).
- 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: $O(\lg n) + O(l)$.
- Removal is done by leaving the name in and changing the number to 0: $O(1)$ + search costs = $O(\lg n) + O(l)$.
- When the unsorted end segment has become too large, sort the entire array: $\Theta(n \lg n)$.
- A mediocre solution, but
  - $\Theta(l)$ 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 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 and naturally, with programming languages' library implementations such as the C++ containers.

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

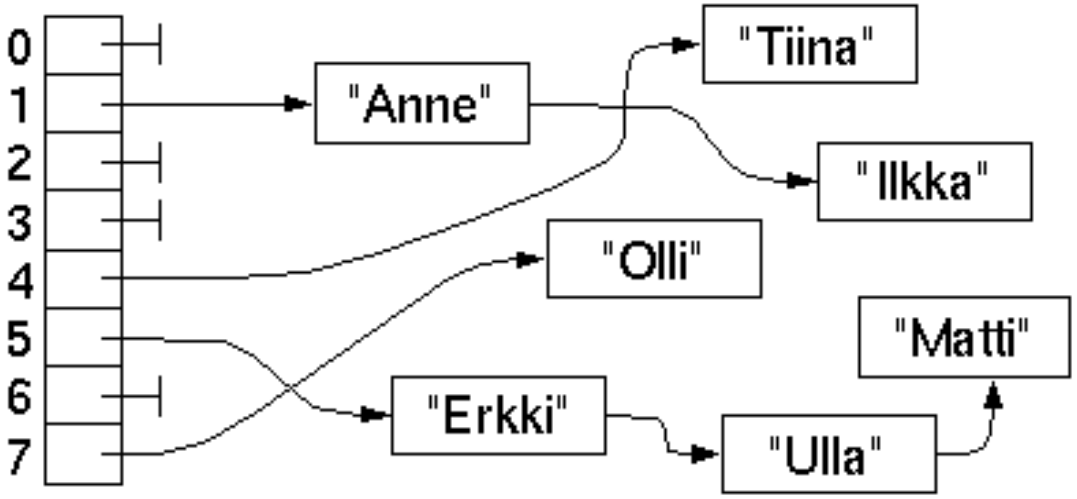
Reducing the range of the keys creates a problem: *collisions*.

- 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
  - 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 hashing with chaining only

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



$h(k)$	first letter			
0		H	P	X
1	A	I	Q	Y
2	B	J	R	Z
3	C	K	S	Ä
4	D	L	T	Ö
5	E	M	U	Å
6	F	N	V	
7	G	O	W	

Is this a good hash?

- No. Let's see why.

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

  $\Rightarrow$ on average α elements need to be investigated

  $\Rightarrow$ 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 likelyhood, on average half of the list needs to be searched through in the case where the key is found in the list

  $\Rightarrow$ 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. α < 50 %), then $\Theta(1+\alpha) = \Theta(1)$

  $\Rightarrow$ 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
  - $\frac{1}{m}$ 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 occuring 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.

- $h(k) = k \bmod m$
- 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

$$h(k) = k\text{'s } b \text{ lowest bits}$$

$\Rightarrow$ the function doesn't even take a look at all the bits in k
$\Rightarrow$ 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 choise, as then all the permutations of the same string end up into the same slot
- prime numbers are usually good choises for m, provided they are not close to a power of two
  - e.g. $\approx$ 700 lists is needed $\Rightarrow$ 701 is OK
- it's worth checking with a small "real" input data set whether the function hashes efficiently

The *multiplication method* for creating hash functions doesn't have large requirements for the values of m.

- the constant A is chosen so that $0 < A < 1$
- $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
- if $m = 2^b$, 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:

$$h(k) = \lfloor \frac{(((2^w \cdot A) \cdot k) \bmod 2^w)}{2^{w-b}} \rfloor$$

- 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

10 Trees

http://imgur.com/L77FY5X

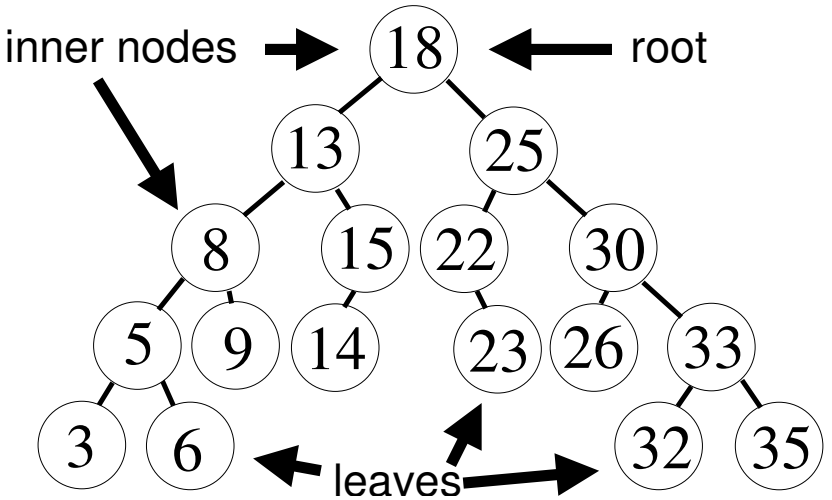
This chapter discusses the most commonly used tree structures.

- 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
- Splay trees and AVL-trees are mentioned

10.1 Basic binary search tree

A *binary tree* is a finite structure of *node*s 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 *ancestor*s 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*
>
> $$l.key \leq x.key \leq r.key$$

- 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

Searching in a binary search tree

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

```
R-TREE-SEARCH(x, k)
1  if x = NIL or k = x→key then
2      return x                          (the searched key is found)
3  if k < x→key then                     (if the searched key is smaller...)
4      return R-TREE-SEARCH(x→left, k)   (...search from the left subtree)
5  else                                  (else...)
6      return R-TREE-SEARCH(x→right, k)  (...search from the right subtree)
```

The algorithm traces a path from the root downward.

In the worst-case the path is down to a 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

The search can be done without the recursion (which is recommendable).

- now the extra memory requirement is $\Theta(1)$
- the running-time is still $O(h)$

```
TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x→key do      (until the key is found or a leaf is reached)
2      if k < x→key then               (if the searched key is smaller...)
3          x := x→left                 (...move left)
4      else                            (else...)
5          x := x→right                (...move right)
6  return x                            (return the result)
```

Minimum and *maximum*:

- the minimum is found by stepping as far to the left as possible

```
TREE-MINIMUM(x)
1  while x→left ≠ NIL do
2        x := x→left
3  return x
```

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

```
TREE-MAXIMUM(x)
1  while x→right ≠ NIL do
2        x := x→right
3  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 a node

- this way SCAN-ALL works correctly even if the elements in the tree all have the same key $\Rightarrow$ an algorithm that finds the node which is the next largest node in 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

```
TREE-SUCCESSOR(x)
1  if x→right ≠ NIL then                (if there is a right subtree...)
2      return TREE-MINIMUM(x→right)     (...find its minimum)
3  y := x→p                             (else step towards the root)
4  while y ≠ NIL and x = y→right do     (until we've moved out of the left child)
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

```
TREE-SCAN-ALL(T)
1  if T.root ≠ NIL then
2      x := TREE-MINIMUM(T.root)        (start from the tree minimum)
3  else
4      x := NIL
5  while x ≠ NIL do                     (scan as long as there are successors)
6      process the element x
7      x := TREE-SUCCESSOR(x)
```

- each edge is travelled through twice, into both directions
  $\Rightarrow$ TREE-SCAN-ALL uses only $\Theta(n)$ time, although it calls TREE-SUCCESSOR n times

- extra memory requirement $\Theta(1)$

  $\Rightarrow$ TREE-SCAN-ALL is asymptotically as fast as and has a better memory consumption than INORDER-TREE-WALK

  – the difference in constant coefficients is not significant

  $\Rightarrow$ it's worth choosing TREE-SCAN-ALL, if the nodes contain p-fields

- TREE-SCAN-ALL allows several simultaneous scans while INORDER-TREE-WALK doesn't

Insertion into a binary search tree:

```
TREE-INSERT(T, z)                          (z points to a structure allocated by the user)
1   y := NIL; x := T.root                  (start from the root)
2   while x ≠ NIL do                       (go downwards until an empty spot is located)
3       y := x                             (save the potential parent)
4       if z→key < x→key then              (move left or right)
5           x := x→left
6       else
7           x := x→right
8   z→p := y                               (make the node found the parent of the new one)
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             (make the new node its parent's left . . . )
12      y→left := z
13  else                                   (. . . or right child)
14      y→right := z
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.
$\Rightarrow$ running-time $O(h)$, extra memory requirement $\Theta(1)$

Deletion is more compilicated as an internal node may also be removed:

```
TREE-DELETE(T, z)                                  (z points to the node under deletion)
1  if z→left = NIL or z→right = NIL then           (if z has only one child . . . )
2       y := z                                     (. . . make z the stucture to be removed)
3  else
4       y := TREE-SUCCESSOR(z)                     (else remove the successor of z)
5  if y→left ≠ NIL then                            (save the only child)
6       x := y→left
7  else
8       x := y→right
9  if x ≠ NIL then                                 (if the child exists . . . )
10      x→p := y→p                                 (. . . link it into the place of the removed node)
11 if y→p = NIL then                               (if the removed node was the root . . . )
12      T.root := x                                (. . . make x the new root)
13 else if y = y→p→left then                       (set x into the place of the removed node . . . )
14      y→p→left := x                              (. . . as the left child . . . )
15 else
16      y→p→right := x                             (. . . or the right child or its parent)
17 if y ≠ z then                                   (if a node other than z was removed z . . . )
18      z→key := y→key                             (. . . switch the data of z and the removed noc
19      z→satellitedata := y→satellitedata
20 return y                                        (return a pointer to the removed node)
```

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.
$\Rightarrow$ 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:

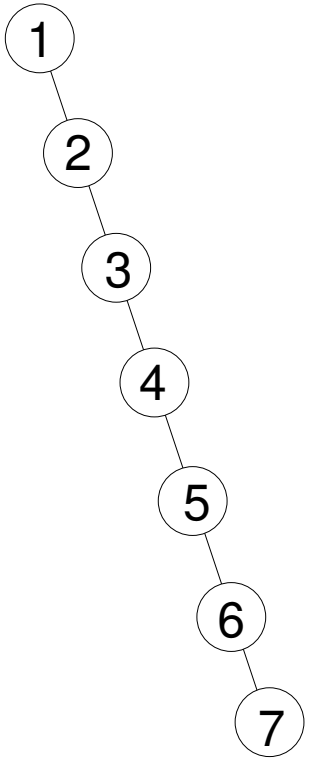
SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR

How high are binary search trees usually?

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 $\Theta(\lg n)$.
$\Rightarrow$ all operations are on average $\Theta(\lg n)$

Unfortunately the result is not neary 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, we'll return to that later.

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 i 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

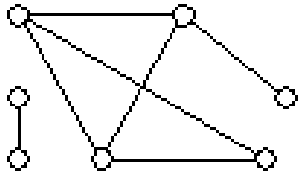
11 Graph algorithms

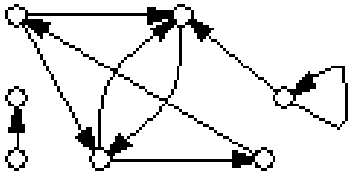
This chapter discusses the data structure that is a collection of points (called nodes or vertices) and connections between them (called edges or arcs) – a graph.

The common search algorithms related to graphs are discussed.

A graph is a data structure that consists of *node*s (or *vertex*), and *edge*s (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

11.1 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, . . . , $|V|^2$
  - this is assummed when analyzing the efficiencies of graph algorithms

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. $O(VE) = O(|V| \cdot |E|)$

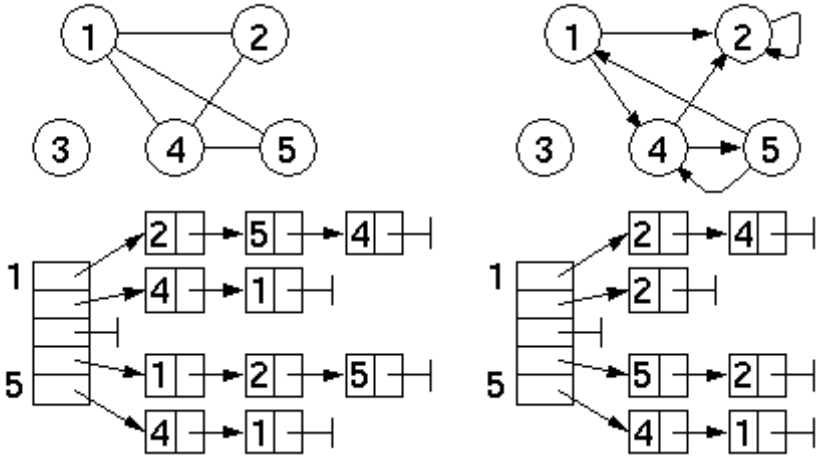
There are two standard ways to represent a graph in a computer *adjacency list* and *adjacency matrix*.

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
  - $2 \cdot |E|$, if the graph is undirected

  $\Rightarrow$ 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 question abowe can easily be answered with the adjacency matrix representation

- 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

	1	2	3	4	5
1	0	1	0	1	1
2	1	0	0	1	0
3	0	0	0	0	0
4	1	1	0	0	1
5	1	0	0	1	0

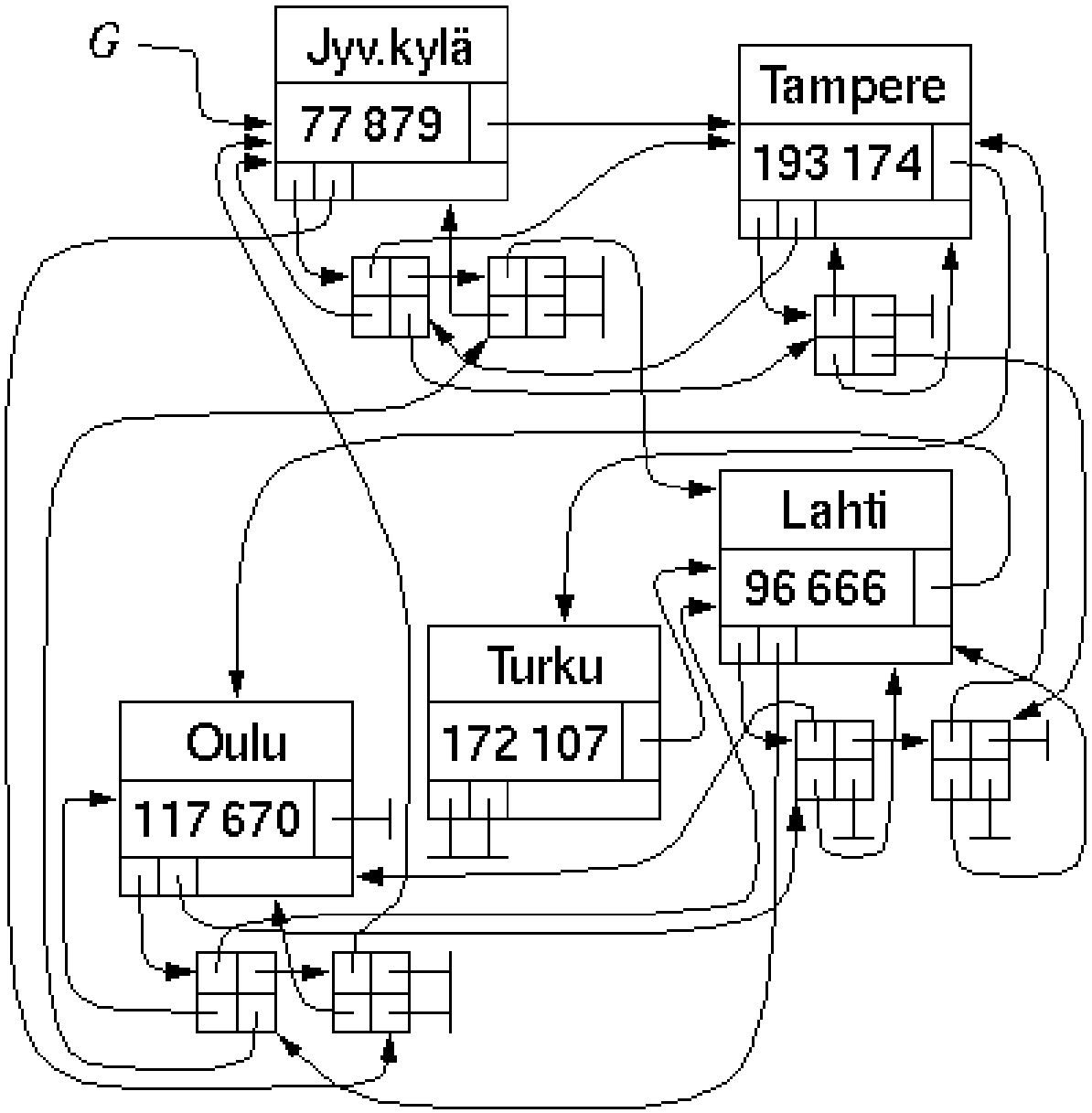
	1	2	3	4	5
1	0	1	0	1	0
2	0	1	0	0	0
3	0	0	0	0	0
4	0	1	0	0	1
5	1	0	0	1	0

- memory consumption is always $\Theta(V^2)$
  - Each element uses only a bit of memory, to several elements can be store in on 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 usefull 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

  $\Rightarrow$ 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


G
Jyv.kylä
77 879
Tampere
193 174
Lahti
96 666
Turku
172 107
Oulu
117 670

11.2 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 tov_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 $\rightarrow$ grey $\rightarrow$ 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.

11.3 Breadth-first search

The breadth-first search can be used for example for:

- determining the distance of all the nodes from the source
- finindh (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 ∞
- $v \rightarrow \pi$ = 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 neighbours of v

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

```
BFS(s)                                  (the algorithm gets the source s as a parameter)
1   ▷ in the beginning the fields of each vertex are colour = WHITE, d = ∞, π = NIL
2   s→colour := GRAY                    (mark the source as discovered)
3   s→d := 0                            (the distance of the source from the source is 0)
4   PUSH(Q, s)                          (push the source to the queue)
5   while Q ≠ ∅ do                      (repeat while there are vertices)
6       u := POP(Q)                     (take the next vertex from the queue)
7       for each v ∈ u→Adj do           (go through the neighbours of u)
8           if v→colour = WHITE then    (if the vertex hasn't been discovered . . . )
9               v→colour := GRAY        (. . . mark is as discovered)
10              v→d := u→d + 1          (increment the distance by one)
11              v→π := u                (vertex v was reached through u)
12              PUSH(Q, v)              (push the vertex into the queue to be processed)
13      u→colour := BLACK               (mark the vertex u as processed)
```

All fields of the vertex used by the algorithm are not necessarily needed in the practical implementaion. Some can be determined based on each other.

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 atmost once

  $\Rightarrow$ the while-loop makes atmost $O(V)$ rounds
- the amount of loops in the for

- the algorithm goes through each edge once into both directions

$\Rightarrow$ for-loop is executed atmost $O(E)$ times in total

$\Rightarrow$ the running-time of the entire algorithm is thus $O(V + E)$

Once the algorithm has ended the π pointers define a tree that containes the discovered vertices with the source s as its root.

- *breadth-first tree*
- π 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 π pointers the shortest path from the source s to the vertex v can be printed with:

```
PRINT-PATH(G, s, v)
1  if v = s then                    (base case of recursion)
2      print s
3  else if v→π = NIL then           (the search didn't reach the vertex v at all)
4      print "no path"
5  else
6      PRINT-PATH(G, s, v→π)        (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 π pointers and printing the contents of the array backwards
  - walking through the path twice and turning the π pointers backwards each time (the latter turning is not necessary if the π pointers can be corrupted)

11.4 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

  - for this reason the example pseudocode has been simplified by leaving out the π-fields
- this version of the algorithm doesn't color the vertices black
  - a little bit more complicated version is needed for that

The data structure S used by the algorithm is a stack (follows the LIFO-policy).

```
 DFS(s)                                    (the algorithm gets the source s as a parameter)
1   ▷ in the beginning the color field of eack (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 ≠ ∅ do                        (continue until the stack is empty)
5       u := POP(S)                       (pop the latest vertex added to the stack)
6       for each v ∈ u→Adj do             (go through the neighbours of u)
7           if v→colour = WHITE then      (if the vertex hasn't been processed . . . )
8               v→colour := GRAY          (. . . mark it discovered . . . )
9               PUSH(S, v)                (. . . and push it into the stack for prosessing)
```

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 satellita 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

- the amount of loops in the while
  - only white vertices are pushed into the stack
  - the color of the vertex is changed into gray at the same time
    $\Rightarrow$ each vertex can be pushed into the stack atmost once

  $\Rightarrow$ the while-loop makes atmost $O(V)$ rounds
- the amount of loops in the for
  - the algorithm goes through each edge once into both directions

  $\Rightarrow$ for-loop is executed atmost $O(E)$ times in total

$\Rightarrow$ the running-time of the entire algorithm is thus $O(V + E)$

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!

```
 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           ▷ 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
  $\Rightarrow$ 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
  $\Rightarrow$ there are atmost $O(E)$ rounds of the for loop
- other operations are constant time

$\Rightarrow$ 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

    $\Rightarrow$ if a grey vertex can be reached from the current vertex, there is a loop in the graph

11.5 Dijkstra's algorithm

A property called the *weight* can be added to the edges of the graph.

- the weigth 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 weigths 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
  $\Rightarrow$ 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 $d[u] = \delta(s, u)$ for sure
  - if the vertex taken from the priority queue is the goal vertex, the execution of the algorithm can be ended

```
DIJKSTRA(s, w)                          (the algorithm gets the source s as parameter)
1   ▷ in the beginning the fields of each vertex are colour = WHITE, d = ∞, π = NIL
2   s→colour := GRAY                    (mark the source found)
3   s→d := 0                            (the distance from the source to itself is 0)
4   PUSH(Q, s)                          (push the source into the priority queue)
5   while Q ≠ ∅ do                      (continue while there are vertices left)
6       u := EXTRACT-MIN(Q)             (take the next vertex from the priority queue)
7       for each v ∈ u→Adj do           (go through the neighbours of u)
8           if v→colour = WHITE then    (if the node has not been visited . . . )
9               v→colour := GRAY        (. . . mark it found)
10                  PUSH(Q, v)          (push the vertex into the queue)
11          RELAX(u, v, w)
12      u→colour := BLACK               (mark u as processed)
```

```
RELAX(u, v, w)
1   if v→d > u→d + w(u, v) then         (if a new shorter route to v was found...)
2       v→d := u→d + w(u, v)            (...decrement the distance of v from the source)
3       v→π := u                        (mark the v was reached through u)
```

Running-time:

- the while loop makes atmost $O(V)$ rounds and the for loop atmost $O(E)$ rounds in total
- the priority queue can be implemented efficiently with a heap or less efficiently with a list

with a heap:	with a list:	
line 4: $\Theta(1)$	$\Theta(1)$	(adding to an empty data structure)
line 5: $\Theta(1)$	$\Theta(1)$	(is the priority queue empty)
line 6: $O(\lg V)$	$O(V)$	(Extract-Min)
line 10: $\Theta(1)$	$\Theta(1)$	(the priority of a white vertex is infinite, its correct location is at the end of the queue)
line 11: $O(\lg V)$	$\Theta(1)$	(relaxation can change the priority of the vertex)

- one $O(\lg V)$ 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 $O((V + E) \lg V)$

11.6 A*algorithm

Dijkstra's algorithm finds the shortest weighted route by accessing the nodes in the order of shortest route. I.e.: Dijkstra only uses information about routes found so far.

A* algorithm enhances this by adding a heuristic (= assumption) about the shortest possible distance to the goal. (For example, the direct distance ("as the crow flies") in a road network.)

- find the shortest weighted route from starting node s to given goal node g. **Does not** find the shortest route to *all* nodes (like Dijkstra).
- assumes that edge weights are non-negative (like Dijkstra)
- assumes that the minimum possible distance from any node to the goal can be calculated (i.e., the found shortest route cannot be shorter).
- chooses in each situation a new node, whose (shortest path from start to node + estimated minimum remaining distance to goal) is smallest.

The only difference between A* and Dijkstran is in relaxation (and the fact that A* should be terminated immediately when the goal is found, since it doesn't calculate shortest routes to all nodes).

RELAX-A*(u, v, w)

```
1  if v→d > u→d + w(u,v) then        (if a shorter route to node v is found...)
2       v→d := u→d + w(u,v)          (...new route this far...)
3       v→de := v→d + min_est(v,goal) (...and minimum estimate for whole route)
4       v→π := u                     (mark the v was reached through u)
```

In the priority queue used by A* the whole route's distance estimate $v{\rightarrow}de$ is used as the priority.

(Dijkstra's algorithm is a special case of A*, where $min\_est(a, b)$ is always 0.)

11.7 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 quarantees that the height of the tree is kept in $\Theta(\lg n)$

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 contains 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 ($\lfloor \frac{h}{2} \rfloor + 1$) 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 $\lfloor \frac{h}{2} \rfloor + 1$ upmost levels are full

  $\Rightarrow n \geq 2^{\frac{h}{2}}$

  $\Rightarrow h \leq 2 \lg n$

Therefore, the invariant does quarantee 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.

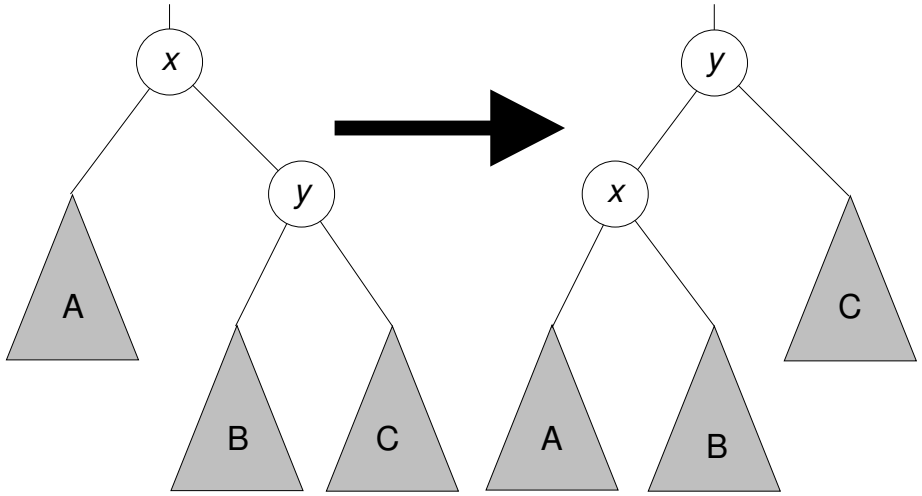
- the operations work with binary search trees in $O(h)$ time, and a red-black tree is a binary search tree with $h = \Theta(\lg n)$

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

The idea of a left rotation:



- rotation is done to the left
  - assumes that the nodes x and y exist
- right rotate similarly
  - $left$ and $right$ have switched places

```
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 x = 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 $\Theta(1)$
- only the pointers are modified

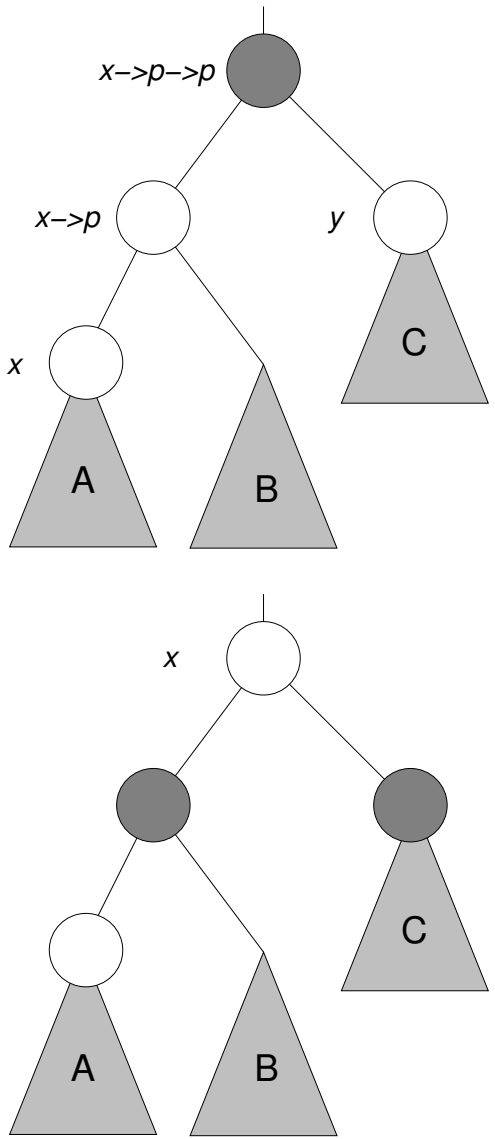
The basic idea of the addition

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

```
 RB-INSERT(T, x)
1   TREE-INSERT(T, x)
2   x→colour := RED
    (loop until the violation has disappeared or the root is reached)
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
30  T.root→colour := BLACK                      (color the root black)
```



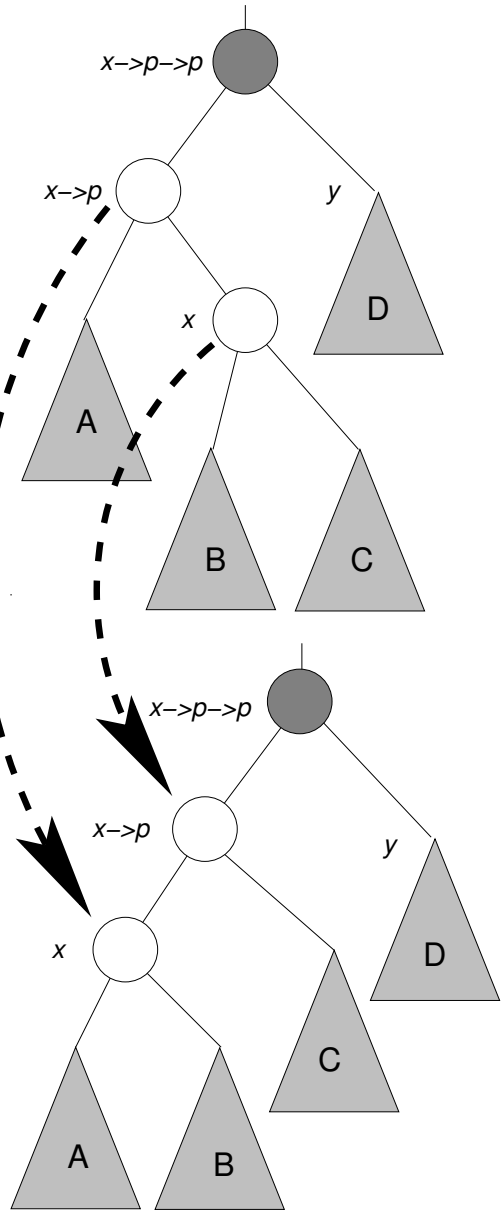
Moving the violation of property 1 upwards:

- the node x and its parent are both red
- also the node x's uncle/aunt is red and the grandparent is black.

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

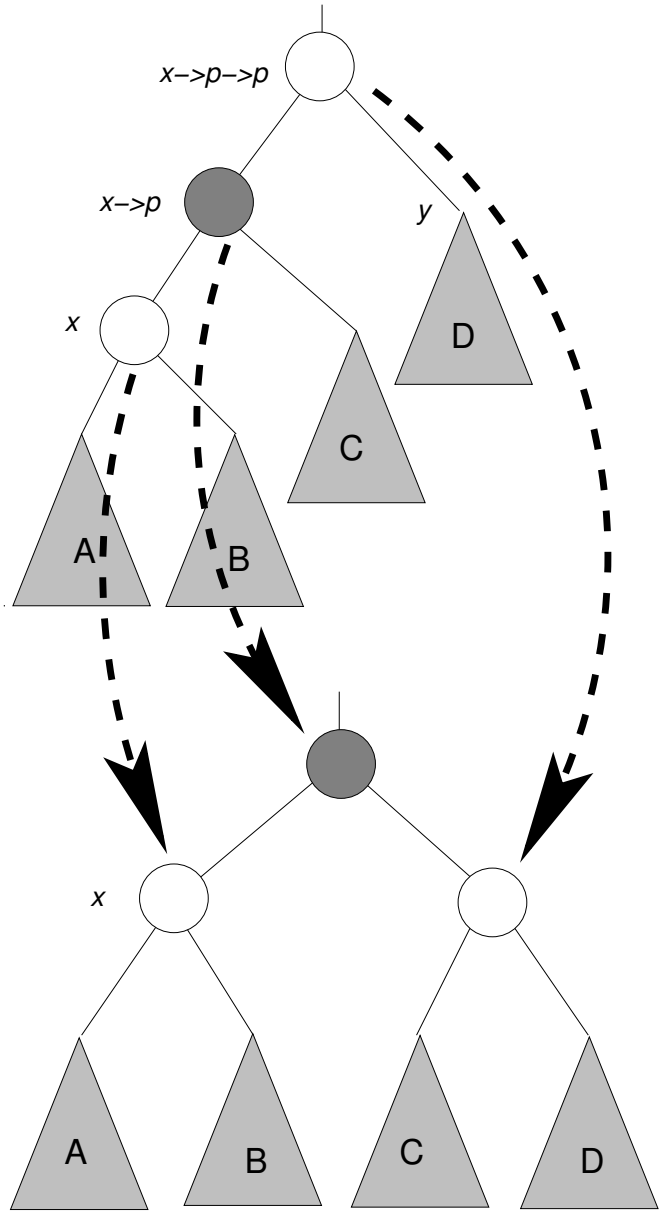
After the fixup:

- 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



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

  $\Rightarrow$ 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

  $\Rightarrow$ 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→colour = BLACK and T.root ≠ NIL then
3       if w→left ≠ NIL then
4           x := w→left
5       else
6           x := w→right
7       RB-DELETE-FIXUP(T, x, w→p)
8   return w
```

```
RB-DELETE-FIXUP(T, x, y)
1   while x ≠ T.root and (x = NIL or x→colour = BLACK) do
2       if x = y→left then
3           w := y→right
4           if w→colour = RED then
5               w→colour := BLACK; y→colour := RED
6               LEFT-ROTATE(T, y); w := y→right
7           if (w→left = NIL or w→left→colour = BLACK) and
            (w→right = NIL or w→right→colour = BLACK)
            then
8               w→colour := RED; x := y
9           else
10              if w→right = NIL or w→right→colour = BLACK then
11                  w→left→colour := BLACK
12                  w→colour := RED
13                  RIGHT-ROTATE(T, w); w := y→right
14              w→colour := y→colour; y→colour := BLACK
15              w→right→colour := BLACK; LEFT-ROTATE(T, y)
16              x := T.root
17      else
...         ▷ same as lines 3...16 except "left" and "right" have switched places
32      y := y→p
33  x→colour := BLACK
```

11.8 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

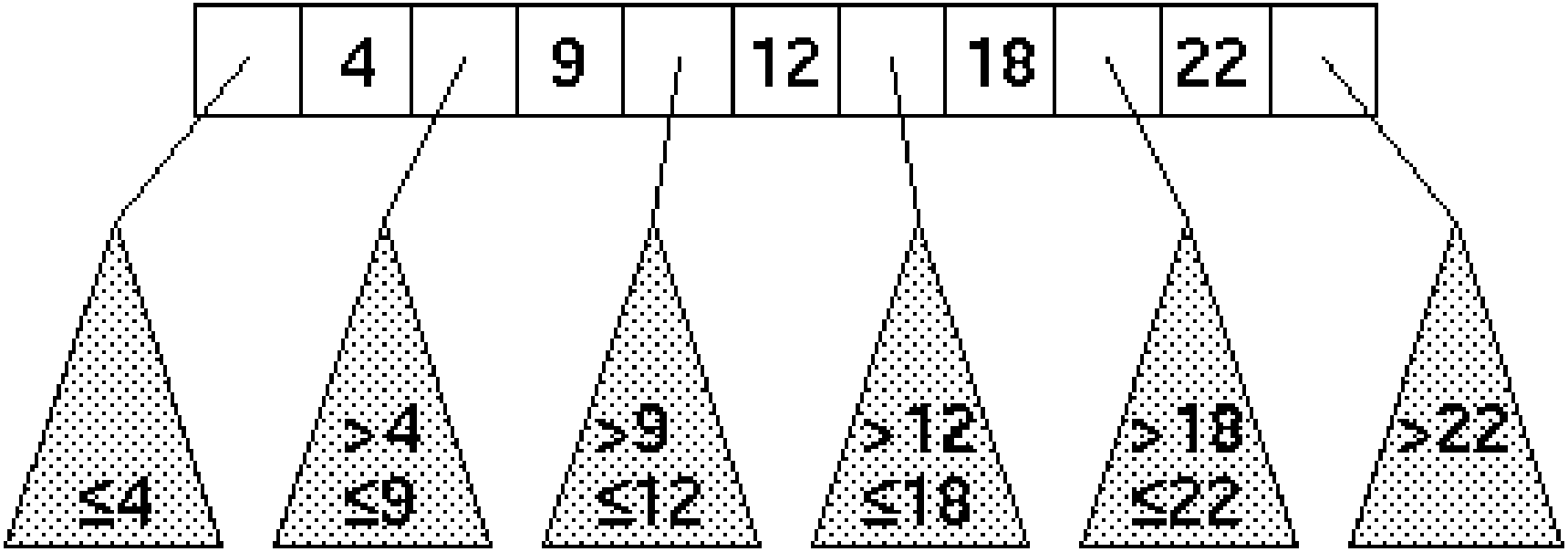
  $\Rightarrow$ B-tree often branches rapidly: each node has tens, hundreds or thousands of children

  $\Rightarrow$ B-trees are very shallow in practise
- the tree is kept balanced by alternating the amount of the node's children between $t, \ldots, 2t$ for some $t \in N, t \geq 2$
  - 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 brances

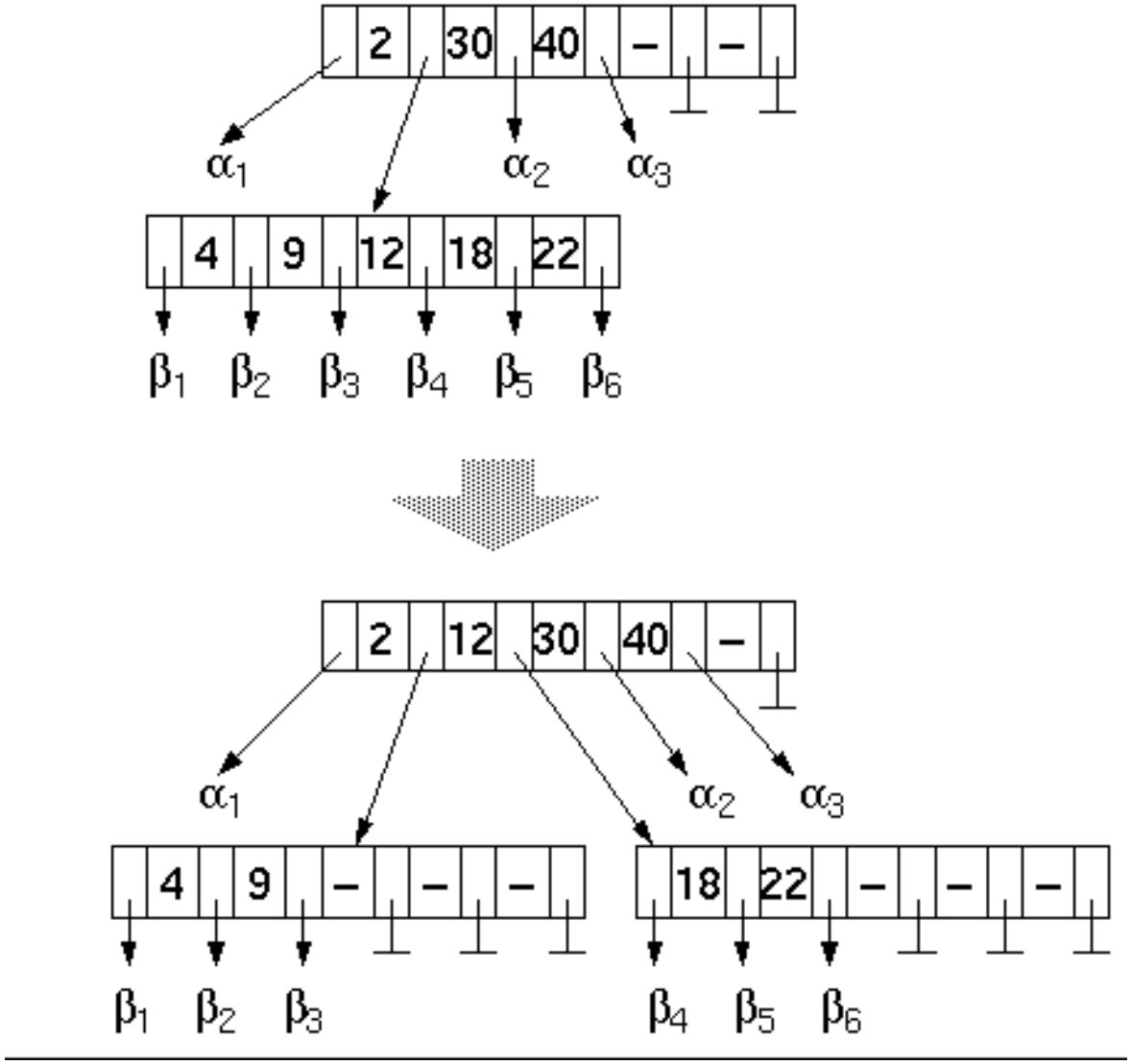


Inserting an element into a B-tree

- travel from the root to a leaf and on the way split each full node into half
  $\Rightarrow$ 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
  $\Rightarrow$ 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.


2
30
40
–
–
α_1
α_2
α_3
4
9
12
18
22
β_1 β_2 β_3 β_4 β_5 β_6
2
12
30
40
–
α_1
α_2
α_3
4
9
–
–
–
18
22
–
–
–
β_1 β_2 β_3 β_4 β_5 β_6

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 mimimun amount + 1 keys in it
  - this quarantees 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

11.9 AVL-trees and Splay-trees

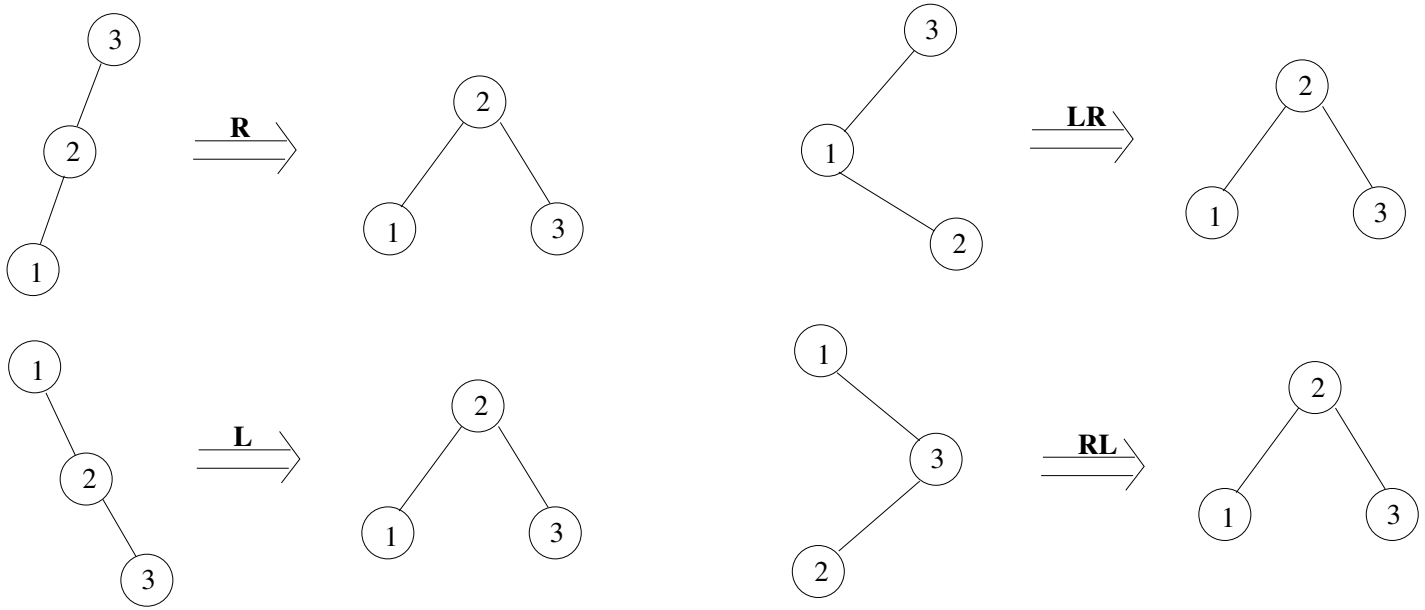
An *AVL tree* (after Adelson-Velsky, Landis) is a binary search tree where every node has a **balance factor** of either 0, +1, or -1.

- the factor is defined as the difference between the heights of the node's left and right subtree

In insertion of a new node makes and AVL tree unbalanced, rotations are used to transform the tree back to balance

There are four rotations:

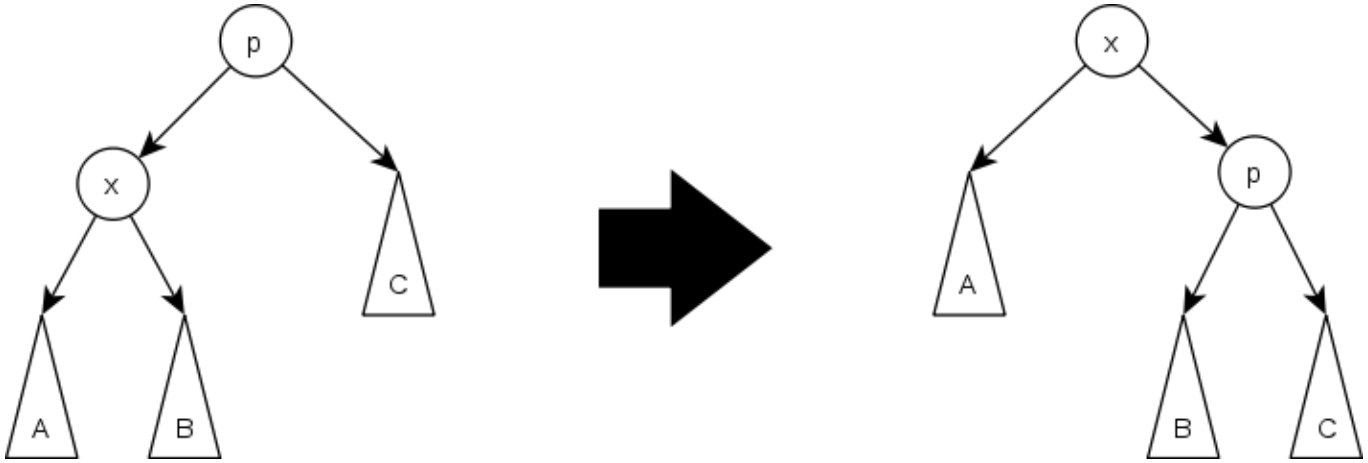
- Single right rotation
- Single left rotation
- Double left-right -rotation
- Double right-left -rotation



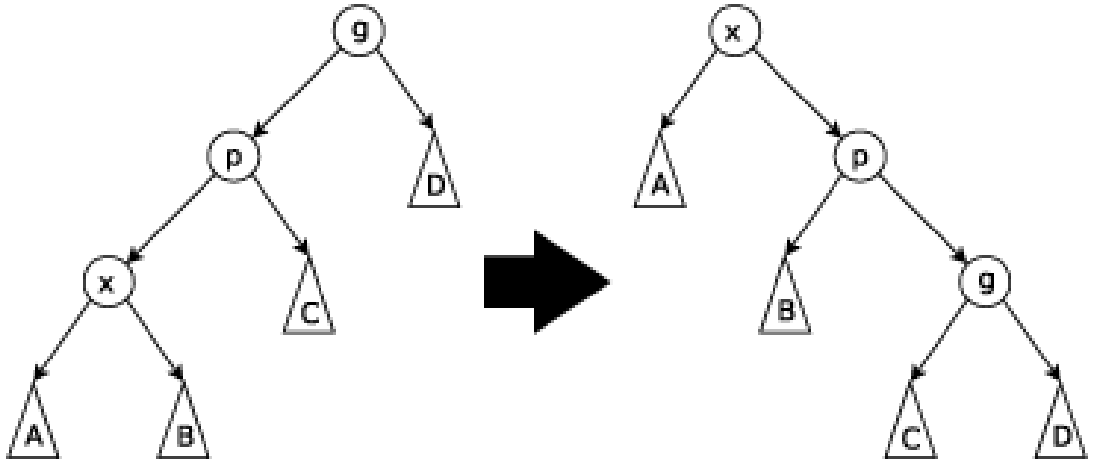
A *splay tree* is a binary search tree with the additional property that recently accessed elements are quick to access again.

A splay operation is performed on a node when accessed to move it to the root: a sequence of splay steps, each moving the node closer to the root.

- Zig Step:



- Zig-Zig Step:



- Zig-Zag Step: