

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?

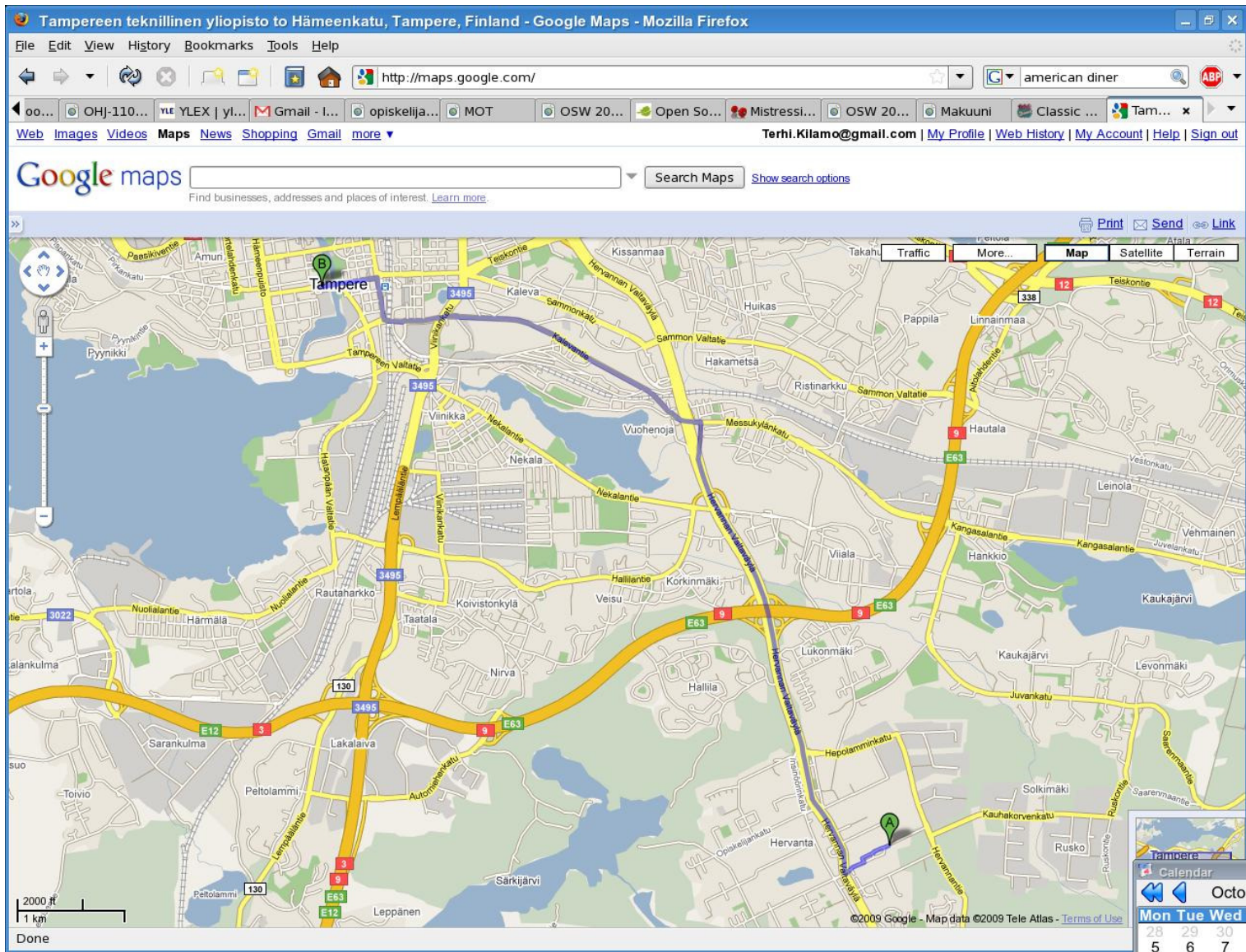


Picture: Chris Watt

There are no computer programs without algorithms

- algorithms make for example the following applications possible:





momondo

flights hotel car rental holiday rentals

Compare cheap flights and hotels

Tell us where you're going and we find the best prices on flights and hotels. Enjoy your trip! Psst ... we're a free service, not a travel agency and we don't add any booking fees.

We come recomm
CNN TRAVEL+ LEISURE

Flights ✈️ Hotel 🏨

One-way
 Return Trip
 Multiple destinations

From: Helsinki (HEL), Finland

To: Madrid (MAD), Spain

Departure date: 03/20/2014

Return date: 03/24/2014

Adults: 1 Children: 0 Ticket Class: Economy

SELECT YOUR END DATE

FEBRUARY 2014 MARCH 2014

Wk	Su	Mo	Tu	We	Th	Fr	Sa	Wk	Su	Mo	Tu
5							1	9			
6	2	3	4	5	6	7	8	10	2	3	4
7	9	10	11	12	13	14	15	11	9	10	11
8	16	17	18	19	20	21	22	12	16	17	18
9	23	24	25	26	27	28		13	23	24	25
10								14	30	31	

algorithms are at work whenever a computer is used

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

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

Modern programming languages provide easy to use library implementations for data structures (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 efficiency and thus the usability and user experience are not meaningless side remarks
- it is important to take memory- and time consumption into account when making decisions in program implementation
- using a library implementation seems more straightforward than is really is

This course discusses these issues

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.

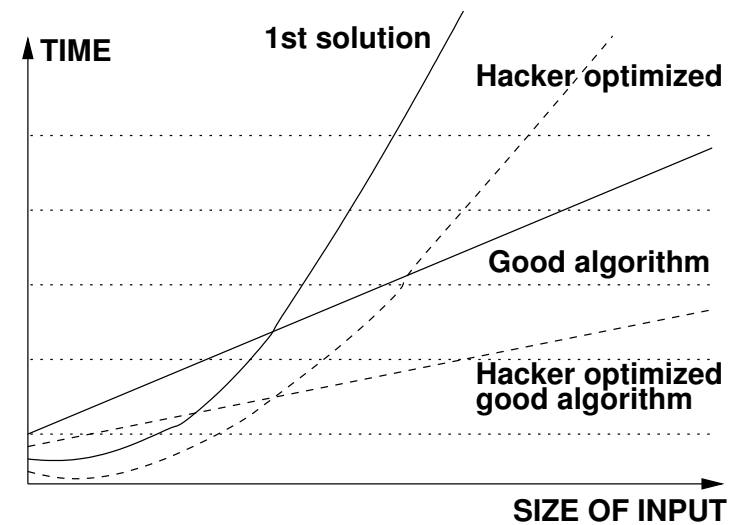
2.1 Goals of the course

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.

	10:20 HEL – 10:55 TKU suora 0 t 35 min	22:00 TKU – 22:35 HEL suora 0 t 35 min	331 € Varaa
Tiedot	tai varaa sivustoilta: travelstart 331 € Travellink 335 € Opodo 335 € Eticket.fi 336 € TravelPartner 337 € finnair 342 € budjet 343 € bravofly 348 € Flyhi 352 €		Supersaver Tarkistettu 1t sitten
	14:00 HEL – 14:35 TKU suora 0 t 35 min	17:10 TKU – 17:45 HEL suora 0 t 35 min	331 € Varaa
Tiedot	tai varaa sivustoilta: travelstart 331 € Travellink 335 € Opodo 335 € TravelPartner 337 € finnair 342 € budjet 343 € bravofly 348 €		Supersaver Tarkistettu 1t sitten

- well defined =
 - each step is detailed enough for the reader (human or machine) to execute
 - each step is 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, \dots, a_n
 - results:** numbers a_1, a_2, \dots, 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

SAS	06:15 HEL – 13:40 TKU 2 vaihtoa ARN,CPH 7 t 25 min	21:25 TKU – 14:30 HEL 2 vaihtoa ARN,CPH 17 t 05 min (+1)	3.200 € Varaa
Tiedot		tai varaa sivustolta: Flyhi 3.216 €	Eticket.fi ✓ päivitetty
SAS	07:45 HEL – 13:40 TKU 2 vaihtoa ARN,CPH 5 t 55 min	21:25 TKU – 23:10 HEL 2 vaihtoa ARN,CPH 25 t 45 min (+1)	3.200 € Varaa
Tiedot		tai varaa sivustolta: Flyhi 3.216 €	Eticket.fi ✓ päivitetty
SAS + KLM	06:15 HEL – 13:40 TKU 2 vaihtoa ARN,CPH 7 t 25 min	21:25 TKU – 23:55 HEL 2 vaihtoa ARN,AMS 26 t 30 min (+1)	3.605 € Varaa
Tiedot		tai varaa sivustolta: Flyhi 3.621 €	Eticket.fi ✓ päivitetty
SAS + KLM	07:55 HEL – 13:40 TKU 1 vaihto CPH 5 t 45 min	21:25 TKU – 23:55 HEL 2 vaihtoa ARN,AMS 26 t 30 min (+1)	3.683 € Varaa
Tiedot		tai varaa sivustolta: Flyhi 3.699 €	Eticket.fi ✓ päivitetty

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 useful one as long as a certain amount of errors is tolerated.

- for example, checking whether a number is prime

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

- usually algorithms are implemented as computer programs or in hardware
 - in practise, the implementation must take several “engineering viewpoints” into account
 - 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

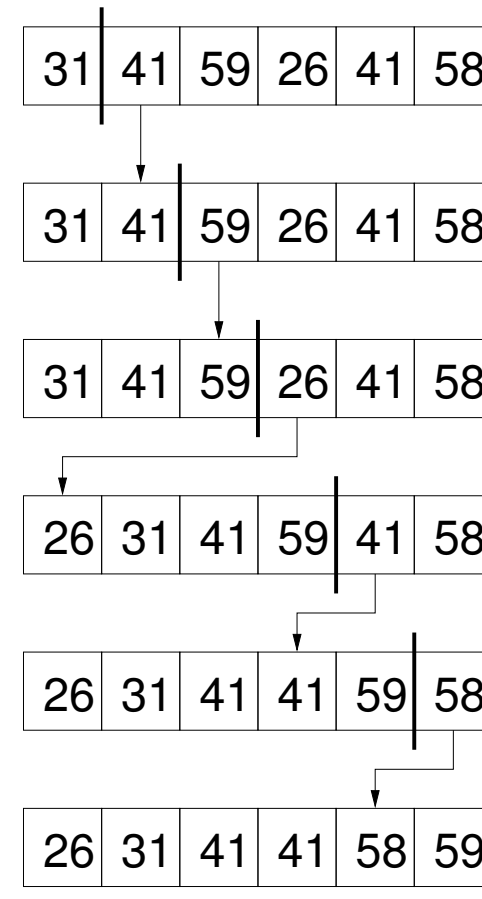
On this course we concentrate on the algorithmic ideas and therefore usually represent the algorithms in pseudocode without legality checks, error handling etc.

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→name*, *x→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
 - ⇒ 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 inside 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 above 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.

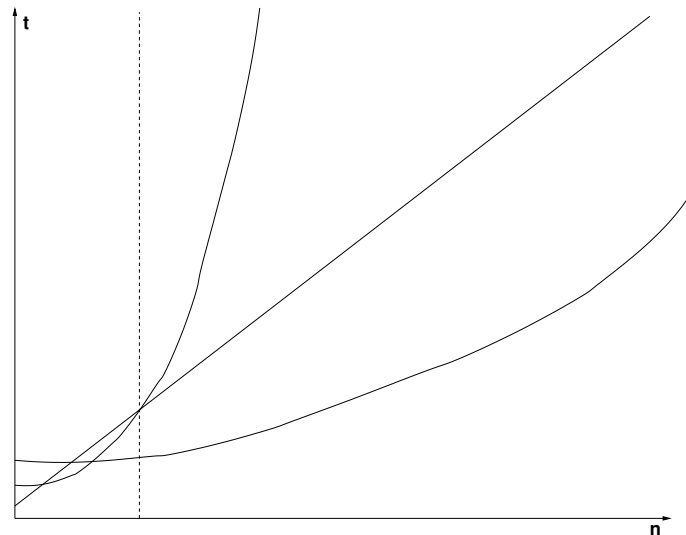
This course concentrates on the principles of algorithms and data structures. Therefore using program code doesn't serve the goals of the course.

⇒ From now on, program code implementations are not normally shown.

3 Efficiency and algorithm design

This chapter discusses 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*.



3.1 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.
 - ⇒ This can be done since as the input gets large enough the lower-order terms get insignificant when compared to the leading term.
 - ⇒ The approach naturally doesn't produce reliable results with small inputs. However, when the inputs are small, programs usually are efficient enough in any case.
- The final result is the efficiency of the algorithm and is denoted it with the greek alphabet theta, Θ .

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

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

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
⇒ 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.
⇒ 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
⇒ 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 occurring at the index i , is p/n , and the amount of comparisons needed is i
- the number of comparisons is:

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

- 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
⇒ 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
⇒ 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.
⇒ the efficiency is $\Theta(n^2)$ (or $\Theta(nm)$ if the arrays are of different lengths)

3.2 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
 - ⇒ 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)	<i>(input in array A)</i>
1 for $j := 2$ to $A.length$ do	<i>(move the limit of the sorted range)</i>
2 $key := A[j]$	<i>(handle the first unsorted element)</i>
3 $i := j - 1$	
4 while $i > 0$ and $A[i] > key$ do	<i>(find the correct location of the new element)</i>
5 $A[i + 1] := A[i]$	<i>(make room for the new element)</i>
6 $i := i - 1$	
7 $A[i + 1] := key$	<i>(set the new element to it's correct location)</i>

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

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

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

3.3 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

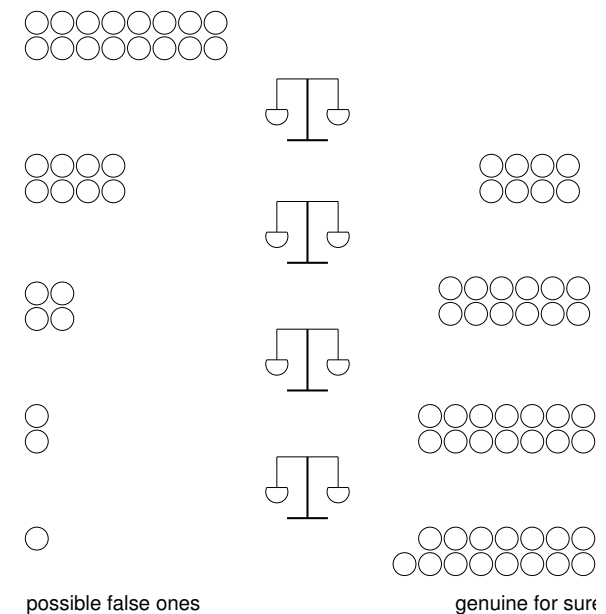
Let's get back to the claim made earlier about the complexity notation not being fixed to programs and take an everyday, concrete example

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

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

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

3.4 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, p, r)

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

The *partition algorithm* rearranges the subarray in place

PARTITION(A, p, r)	
1	$x := A[r]$ <i>(choose the last element as the pivot)</i>
2	$i := p - 1$ <i>(use i to mark the end of the smaller elements)</i>
4	for $j := p$ to $r - 1$ do <i>(scan to the second to last element)</i>
6	if $A[j] \leq x$ <i>(if $A[j]$ goes to the half with the smaller elements...)</i>
9	$i := i + 1$ <i>(... increment the amount of the smaller elements...)</i>
12	exchange $A[i] \leftrightarrow A[j]$ <i>(... and move $A[j]$ there)</i>
12	exchange $A[i + 1] \leftrightarrow A[r]$ <i>(place the pivot between the halves)</i>
13	return $i + 1$ <i>(return the location of the pivot)</i>

How fast is PARTITION?

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

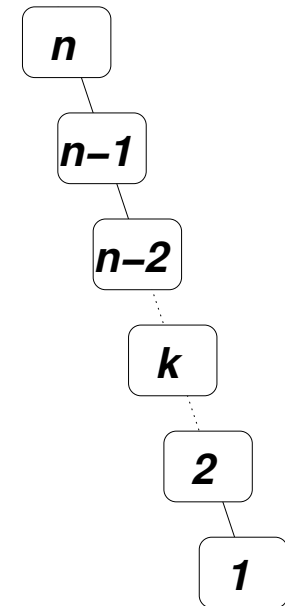
⇒ The running-time is $\Theta(n)$.

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

Worst-case running time

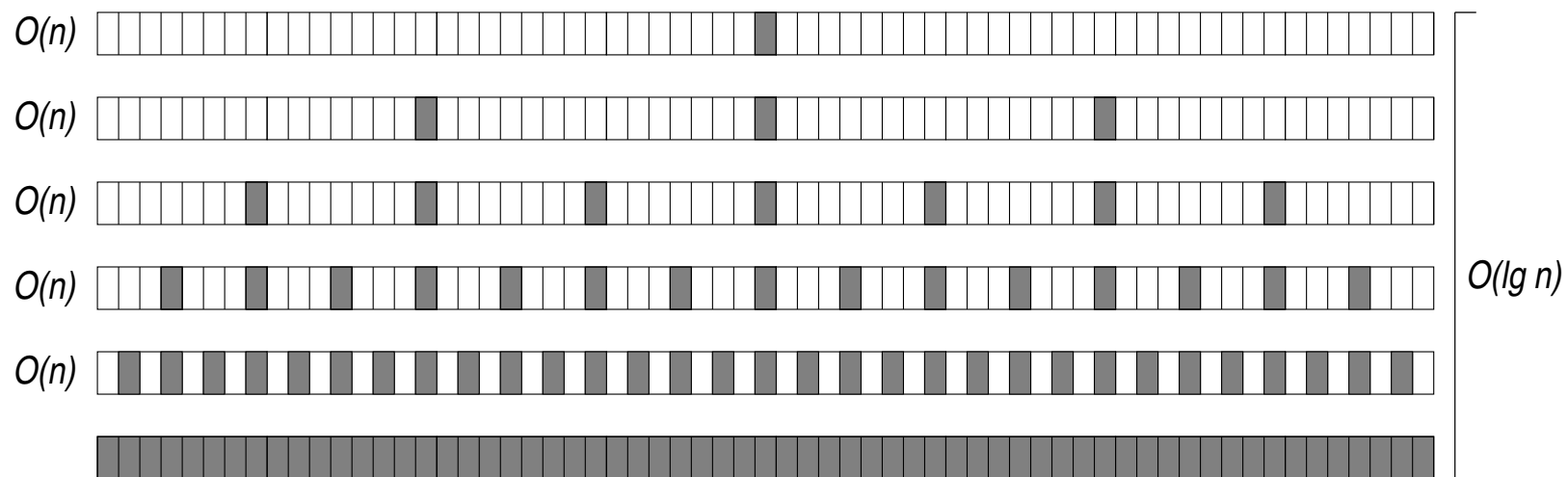
- The number of a node is always smaller than the number of its parent, since the pivot is already in its correct location and doesn't go into either of the sorted subarrays
⇒ there can be at most n levels in the tree
- the worst case is realized when the smallest or the largest element is always chosen as the pivot
 - this happens, for example, with an array already sorted
- the sum of the node numbers is $n + n - 1 + \dots + 2 + 1$

⇒ 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.

⇒ A way to decrease the risk of the systematic occurrence of the worst-case situation's likelihood is needed.

Randomization has proved to be quite efficient.

Advantages and disadvantages of QUICKSORT

Advantages:

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

Disadvantages:

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

3.5 Algorithm Design Technique: Randomization

Randomization is one of the design techniques of algorithms.

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

- Randomization is usually a good idea when
 - the algorithm can continue its execution in several ways
 - it is difficult to see which way is a good one
 - most of the ways are good
 - a few bad guesses among the good ones don't make much damage
 - For example, QUICKSORT can choose any element in the array as the pivot
 - besides the almost smallest and the almost largest elements, all other elements are a good choice
 - it is difficult to guess when making the selection whether the element is almost the smallest/largest
 - a few bad guesses now and then doesn't ruin the efficiency of QUICKSORT
- ⇒ 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, p, r)

- 1 $i := \text{RANDOM}(p, r)$ *(choose a random element as pivot)*
- 2 exchange $A[r] \leftrightarrow A[i]$ *(store it as the last element)*
- 3 **return** PARTITION(A, p, r) *(call the normal partition)*

RANDOMIZED-QUICKSORT(A, p, r)

- 1 **if** $p < r$ **then**
- 2 $q := \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3 RANDOMIZED-QUICKSORT($A, p, q - 1$)
- 4 RANDOMIZED-QUICKSORT($A, q + 1, r$)

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 likelihood holds for RANDOMIZED-QUICKSORT for sure.
- This holds for the normal QUICKSORT only if the data is evenly distributed.

⇒ 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 i th element belongs to and continue recursively from there.
- The other half does not need to be processed further.

RANDOMIZED-SELECT(A, p, r, i)

1	if $p = r$ then	<i>(if the subarray is of size 1...)</i>
2	return $A[p]$	<i>(... return the only element)</i>
3	$q :=$ RANDOMIZED-PARTITION(A, p, r)	<i>(divide the array into two halves)</i>
4	$k := q - p + 1$	<i>(calculate the number of the pivot)</i>
5	if $i = k$ then	<i>(if the pivot is the ith element in the array...)</i>
6	return $A[q]$	<i>(...return it)</i>
7	else if $i < k$ then	<i>(continue the search from the small ones)</i>
8	return RANDOMIZED-SELECT($A, p, q - 1, i$)	
9	else	<i>(continue on the large ones)</i>
10	return RANDOMIZED-SELECT($A, q + 1, r, i - k$)	

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 i th element and the execution ends.
- RANDOMIZED-PARTITION is run once for the entire array.

⇒ 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 i th element is left in the larger half.
- the amount of work is decreased only by one step on each level of recursion.

⇒ 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.

4 Sorting algorithms

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

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

Finally the maximum efficiency of comparison sorts, i.e. sorting based on the comparisons of the elements, is discussed. Sorting algorithms that use other approaches than comparisons are also examined.

4.1 Sorting with a heap

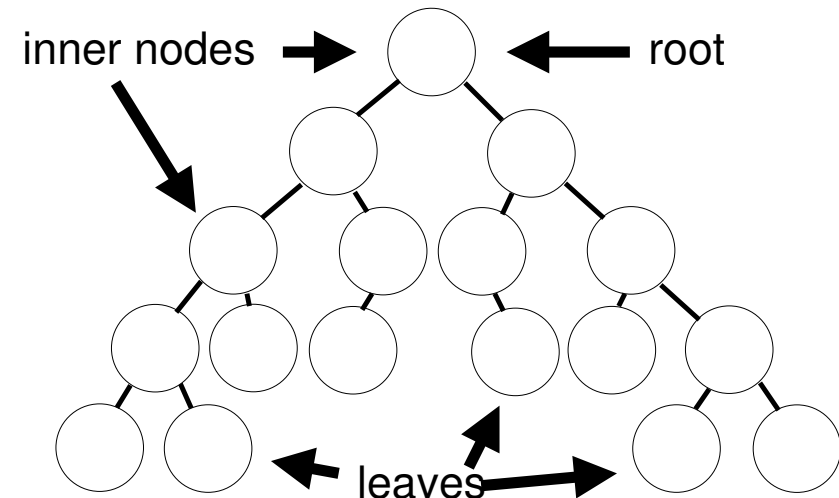
This chapter introduces a sorting algorithm HEAPSORT that uses a very important data structure, a *heap*, to manage data during execution.

Binary trees

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

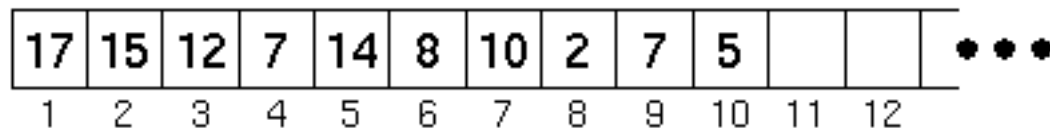
- a structure that consists of nodes who each have 0, 1 or 2 children
- the children are called *left* and *right*
- a node is the *parent* of its children
- a childless node is called a *leaf*, and the other nodes are *internal nodes*
- a binary tree has at most one node that has no parent, i.e. the *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 its root
- a binary tree is *completely balanced* if the difference between the height of the root's left and right subtrees is at most one and the subtrees are completely balanced
- the height of a binary tree with n nodes is at least $\lfloor \lg n \rfloor$ and at most $n - 1$
 - $\Rightarrow O(n)$ and $\Omega(\lg n)$



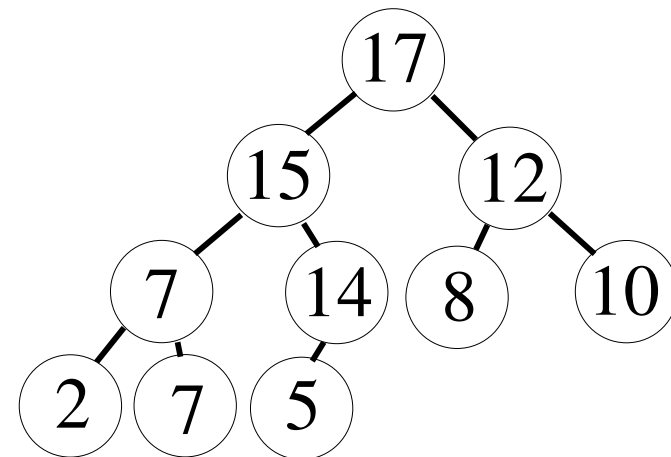
Heap

An array $A[1 \dots n]$ 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 $2i + 1$

⇒ Now the heap property can be given with:

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

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

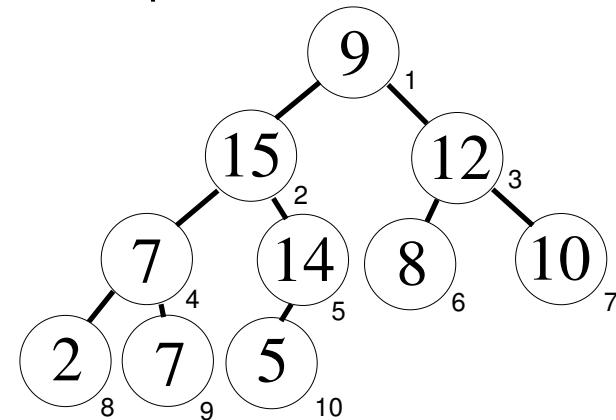
Due to the heap property, the largest element of the heap is always its root, i.e. at the first index in the array.

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

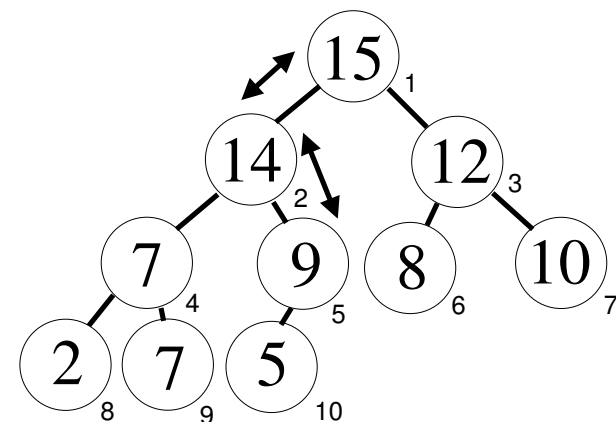
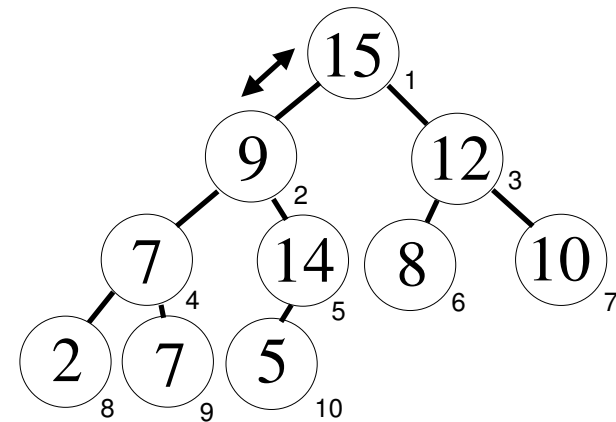
⇒ If there are n nodes in the heap its height is $\Theta(\lg n)$.

Adding an element to the heap from the top:

- let's assume that $A[1 \dots n]$ 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
 - ⇒ the tree becomes a heap



The same is 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      $l := \text{LEFT}(i)$ 
4      $r := \text{RIGHT}(i)$ 
5     if  $l \leq A.heapsize$  and  $A[l] > A[i]$  then   (the left child is larger than i)
6          $i := l$ 
7     if  $r \leq A.heapsize$  and  $A[r] > A[i]$  then   (right child is even larger)
8          $i := r$ 
9     if  $i \neq old\_i$  then                       (if a larger child was found...)
10        exchange  $A[old\_i] \leftrightarrow A[i]$    (...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  $i := \lfloor A.length/2 \rfloor$  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
- ⇒ 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	executions times of HEAPIFY
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 running time is $\frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \dots = \frac{n}{2} \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{n}{2} \cdot 2 = n \Rightarrow O(n)$
- \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:

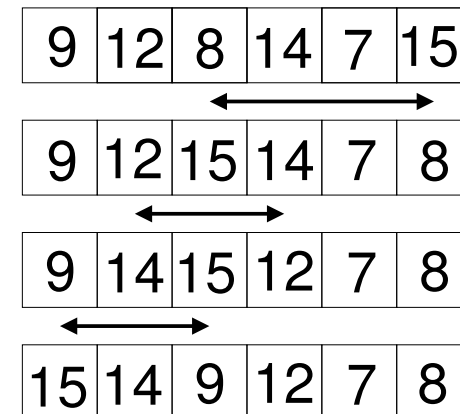
```

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)

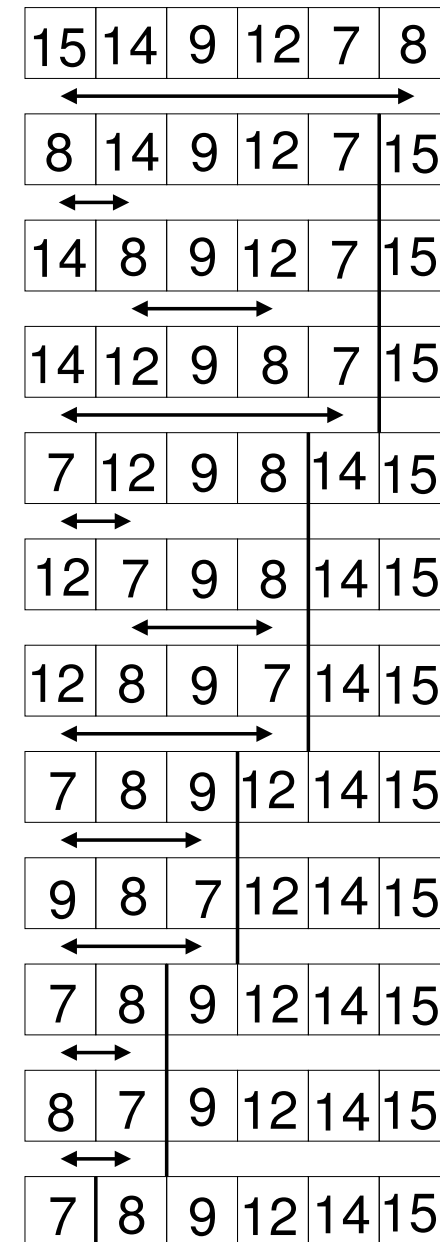
```

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 laborious
 - the heap property is obviously weaker than the order



- the picture shows how the sorted range at the end of the array gets larger until the entire array is sorted
- the heap property is fixed each time the sorted range gets larger
- the fixing process seems complex in such a small example
 - the fixing process doesn't take a lot of steps even with large arrays, only a logarithmic amount



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 above assume that the data structure used to store the heap provides a constant time indexing.

- Heap is worth using only when this is true

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

4.2 Priority queue

A *priority queue* is a data structure for maintaining a set S of elements, each associated with a *key* value. The following operations can be performed:

- $\text{INSERT}(S, x)$ inserts the element x into the set S
- $\text{MAXIMUM}(S)$ returns the element with the largest key
 - if there are several elements with the same key, the operation can choose any one of them
- $\text{EXTRACT-MAX}(S)$ removes and returns the element with the largest key
- alternatively the operations $\text{MINIMUM}(S)$ and $\text{EXTRACT-MIN}(S)$ can be implemented
 - there can be **only the maximum** or **only the minimum** operations implemented in the same queue

Priority queues can be used widely

- prioritizing tasks in an operating system
 - new tasks are added with the command INSERT
 - as the previous task is completed or interrupted the next one is chosen with EXTRACT-MAX
- action based simulation
 - the queue stores incoming (not yet simulated) actions
 - the key is the time the action occurs
 - an action can cause new actions
 - ⇒ they are added to the queue with INSERT
 - EXTRACT-MIN gives the next simulated action
- finding the shortest route on a map
 - cars driving at constant speed but choosing different routes are simulated until the first one reaches the destination
 - a priority queue is needed in 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  $max$ 
```

- $\text{INSERT}(S, x)$ 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 $O(\lg n)$

$\text{HEAP-INSERT}(A, key)$

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

\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)

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

4.3 QUICKSORT

This chapter covers a very efficient sorting algorithm QUICKSORT.

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

The division of the problem into smaller subproblems

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

Smaller subproblems:

- Subarrays of the size 0 and 1 are already sorted

Combining the sorted subarrays:

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

QUICKSORT-algorithm

QUICKSORT(A, p, r)

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

The *partition algorithm* rearranges the subarray in place

PARTITION(A, p, r)	
1	$x := A[r]$ <i>(choose the last element as the pivot)</i>
2	$i := p - 1$ <i>(use i to mark the end of the smaller elements)</i>
4	for $j := p$ to $r - 1$ do <i>(scan to the second to last element)</i>
6	if $A[j] \leq x$ <i>(if $A[j]$ goes to the half with the smaller elements...)</i>
9	$i := i + 1$ <i>(... increment the amount of the smaller elements...)</i>
12	exchange $A[i] \leftrightarrow A[j]$ <i>(... and move $A[j]$ there)</i>
12	exchange $A[i + 1] \leftrightarrow A[r]$ <i>(place the pivot between the halves)</i>
13	return $i + 1$ <i>(return the location of the pivot)</i>

How fast is PARTITION?

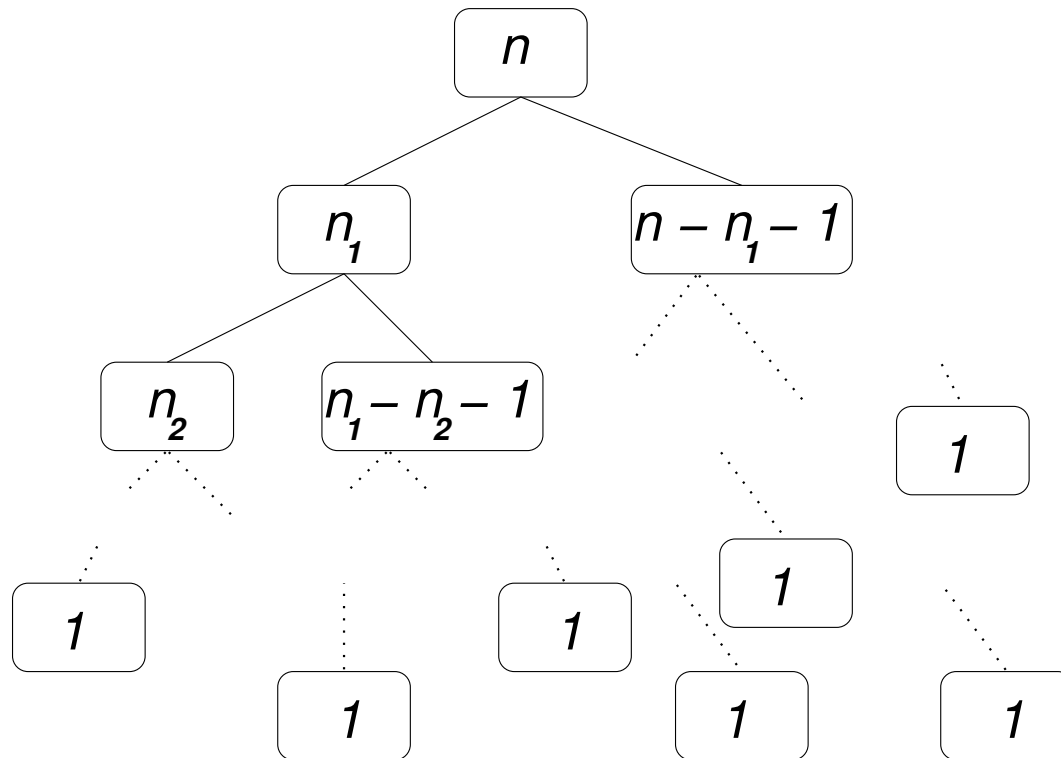
- The **for**-loop is executed $n - 1$ times when n is $r - p$
- All other operations are constant time.

⇒ The running-time is $\Theta(n)$.

Determining the running-time of QUICKSORT is more difficult.

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

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

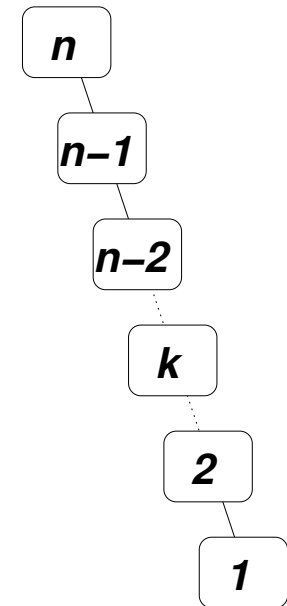


- The total time is the sum of the running times of the nodes in the picture above.
- The execution is constant time for an array of size 1.
- For the other the execution is linear to the size of the array.
⇒ The total time is Θ (the sum of the numbers of the nodes).

Worst-case running time

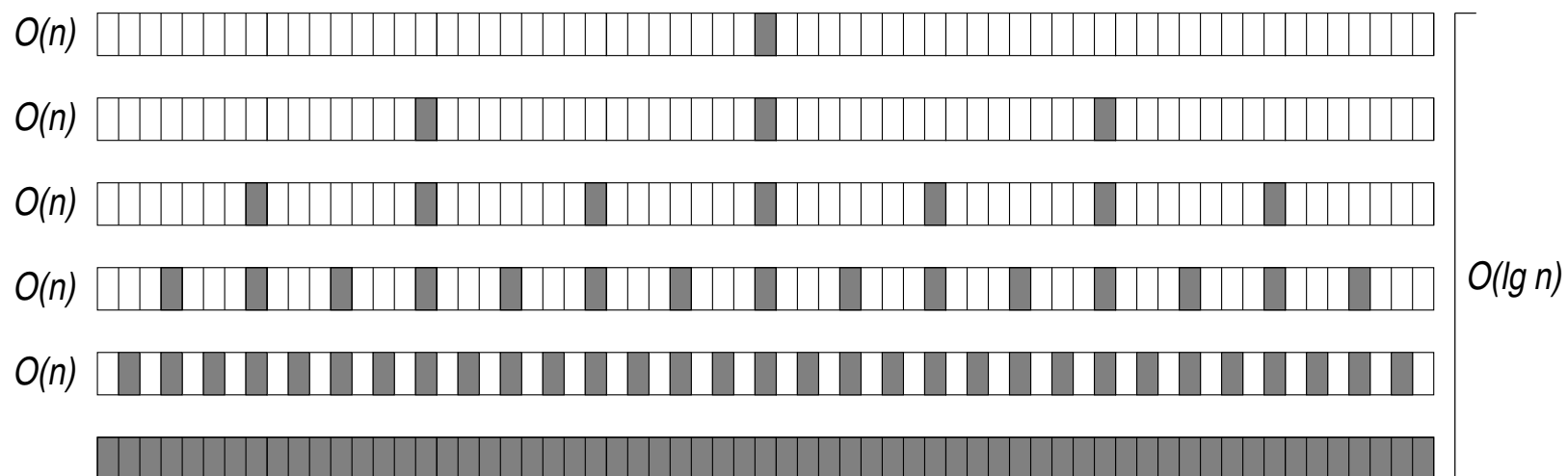
- The number of a node is always smaller than the number of its parent, since the pivot is already in its correct location and doesn't go into either of the sorted subarrays
⇒ there can be at most n levels in the tree
- the worst case is realized when the smallest or the largest element is always chosen as the pivot
 - this happens, for example, with an array already sorted
- the sum of the node numbers is $n + n - 1 + \dots + 2 + 1$

⇒ the running time of QUICKSORT is $O(n^2)$



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 $\Theta(n)$.
 - If the pivot would be kept either in the half with the smaller elements or with the larger elements the situation would be the same as in MERGE-SORT (page ??).
 - a pessimistic estimate on the height of the execution tree is in the best-case $\Rightarrow O(\lg n)$
- \Rightarrow The upper limit for the best-case efficiency is $O(n \lg n)$.



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.

⇒ A way to decrease the risk of the systematic occurrence of the worst-case situation's likelihood is needed.

Randomization has proved to be quite efficient.

Advantages and disadvantages of QUICKSORT

Advantages:

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

Disadvantages:

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

4.4 Randomization

Randomization is one of the design techniques of algorithms.

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

- Randomization is usually a good idea when
 - the algorithm can continue its execution in several ways
 - it is difficult to see which way is a good one
 - most of the ways are good
 - a few bad guesses among the good ones don't make much damage
 - For example, QUICKSORT can choose any element in the array as the pivot
 - besides the almost smallest and the almost largest elements, all other elements are a good choice
 - it is difficult to guess when making the selection whether the element is almost the smallest/largest
 - a few bad guesses now and then doesn't ruin the efficiency of QUICKSORT
- ⇒ 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, p, r)

- 1 $i := \text{RANDOM}(p, r)$ *(choose a random element as pivot)*
- 2 exchange $A[r] \leftrightarrow A[i]$ *(store it as the last element)*
- 3 **return** PARTITION(A, p, r) *(call the normal partition)*

RANDOMIZED-QUICKSORT(A, p, r)

- 1 **if** $p < r$ **then**
- 2 $q := \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3 RANDOMIZED-QUICKSORT($A, p, q - 1$)
- 4 RANDOMIZED-QUICKSORT($A, q + 1, r$)

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 likelihood holds for RANDOMIZED-QUICKSORT for sure.
- This holds for the normal QUICKSORT only if the data is evenly distributed.

⇒ 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 i th element belongs to and continue recursively from there.
- The other half does not need to be processed further.

RANDOMIZED-SELECT(A, p, r, i)

1	if $p = r$ then	<i>(if the subarray is of size 1...)</i>
2	return $A[p]$	<i>(... return the only element)</i>
3	$q :=$ RANDOMIZED-PARTITION(A, p, r)	<i>(divide the array into two halves)</i>
4	$k := q - p + 1$	<i>(calculate the number of the pivot)</i>
5	if $i = k$ then	<i>(if the pivot is the ith element in the array...)</i>
6	return $A[q]$	<i>(...return it)</i>
7	else if $i < k$ then	<i>(continue the search from the small ones)</i>
8	return RANDOMIZED-SELECT($A, p, q - 1, i$)	
9	else	<i>(continue on the large ones)</i>
10	return RANDOMIZED-SELECT($A, q + 1, r, i - k$)	

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 i th element and the execution ends.
- RANDOMIZED-PARTITION is run once for the entire array.

⇒ The algorithm's running-time is $\Omega(n)$.

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

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

⇒ The running-time of the algorithm is $O(n^2)$.

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

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

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

4.5 Other sorting algorithms

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

- They determine the correct order only based on comparing the values of the elements to each other.

It is possible to use information other than comparisons to sort the data.

Sorting by counting

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

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


```

COUNTING-SORT( $A, B, k$ )
1  for  $i := 1$  to  $k$  do
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  $\leq i$ )
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 a reverse order in order to guarantee 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 of the running-time is 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 a key with several parts.

- the list of exam results first based on the department and then into an alphabetical order
- dates first based on the year, then the month and then the day
- a deck of cards first based on the suit and then according to the numbers

The different criteris are taken into account as follows

- The most significant criterion according to which the values of the elements differs determines the result of the comparison.
- If the elements are equal with each criteria they are considered equal.

The problem can be solved with a comparison sort (e.g. by using a suitable comparison operator in QUICKSORT)

- 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 element and 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       $\triangleright$  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 criteria as is needed to determine the order
 - more complex to implement than LSD-RADIX-SORT
 - ⇒ 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
 - ⇒ total efficiency $\Theta(dn + dk)$
- k is usually constant
 - ⇒ total efficiency $\Theta(dn)$, or $\Theta(n)$, if d is also constant

RADIX-SORT appears to be a $O(n)$ sorting algorithm with certain assumptions.

Is it better than the comparison sorts in general?

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

- For example INSERTION-SORT is $O(n^2)$, 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 each other.

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
 - ⇒ $k = 2^8$ and $d = 4$
 - ⇒ 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.
 - ⇒ 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 $B[0] \dots B[n - 1]$.

BUCKET-SORT(A)

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

Implementation of the buckets:

- Operations INSERT and EXTRACT-MIN are needed.
 - ⇒ 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

⇒ an implementation that uses a heap would require $\Theta(n)$ for each bucket $\Theta(n^2)$ in total
- On the other hand, the implementation does not need to be very efficient for large buckets since they are rare.
 - ⇒ In practise the buckets should be implemented as lists.
 - INSERT links the incoming element to its correct location in the list, $\Theta(\text{list length})$ 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.
⇒ 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.

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

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

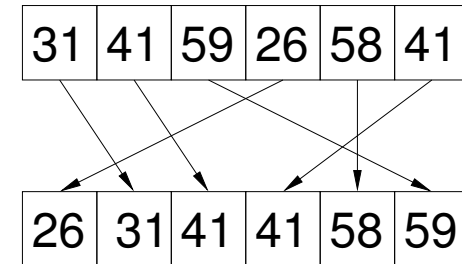
- INSERT takes a linear amount of time

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

4.6 How fast can we sort?

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

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



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

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

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

- An array of n elements of different values has $1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ i.e. $n!$ permutations.

- So many comparisons need to be made that the only correct alternative gets chosen from the set.
- Each comparison $A[i] \leq A[j]$ (or $A[i] < A[j]$) divides the permutations into two groups: those where the order of $A[i]$ and $A[j]$ must be switched and those where the order is correct so...
 - one comparison is enough to pick the right alternative from at most two
 - two comparisons is enough to pick the right one from at most four
 - ...
 - k comparisons is enough to pick the right alternative from at most 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 $\lceil \lg n! \rceil$?

- $\lceil \lg n! \rceil \geq \lg n! = \sum_{k=1}^n \lg k \geq \sum_{k=\lceil \frac{n}{2} \rceil}^n \lg \frac{n}{2} \geq \frac{n}{2} \cdot \lg \frac{n}{2} = \frac{1}{2}n \lg n - \frac{1}{2}n = \Omega(n \lg n) - \Omega(n) = \Omega(n \lg n)$
- 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 for real asymptotically more time consuming than finding the median value, which can be done in the slowest possible case in $O(n)$.

4.7 Hash table

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

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

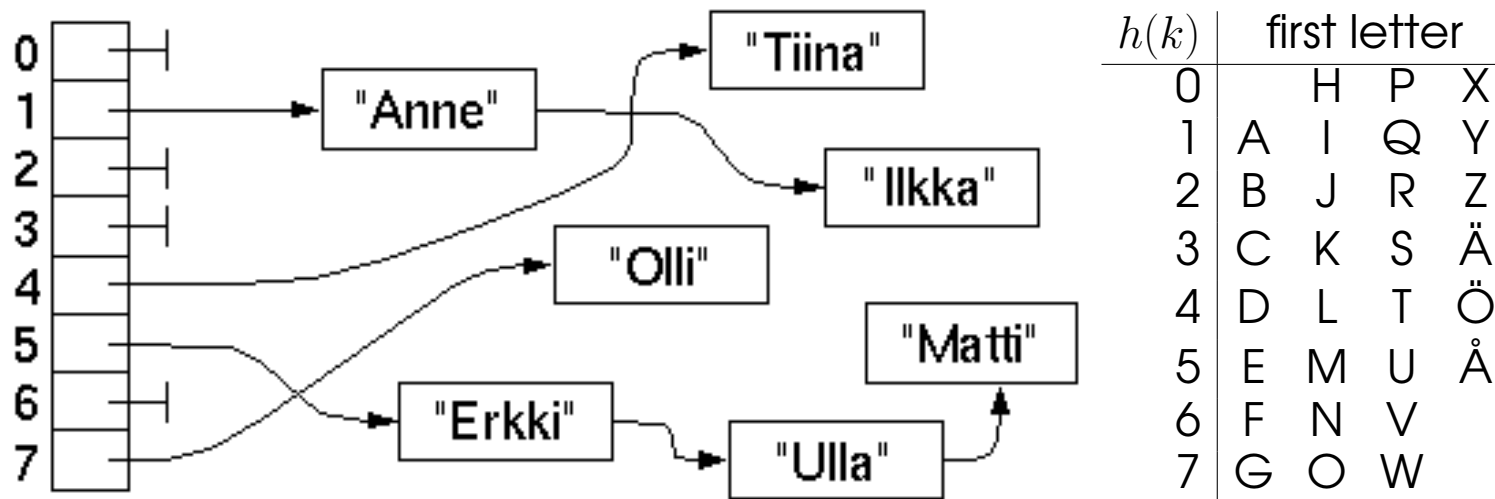
Reducing the range of the keys creates a problem: *collisions*.

- more than one element can hash into the same slot in the hash table

The most common way to solve the problem is called *chaining*.

- all the elements that hash to the same slot are put into a linked list
- there are other alternatives
 - in *open addressing* the element is put into a secondary slot if the primary slot is unavailable
 - 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.



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
 - ⇒ on average α elements need to be investigated
 - ⇒ the running-time is on average $\Theta(1 + \alpha)$
- if we assume that any of the elements in the list is the key with the same likelihood, on average half of the list needs to be searched through in the case where the key is found in the list
 - ⇒ the running-time is $\Theta(1 + \frac{\alpha}{2}) = \Theta(1 + \alpha)$ on average
- if the load factor is kept under some fixed constant (e.g. $\alpha < 50\%$), then $\Theta(1 + \alpha) = \Theta(1)$
 - ⇒ all operations of a chained hash table can be implemented in $\Theta(1)$ running-time on average
 - this requires that the size of the hash table is around the same as the amount of elements stored in the table

When evaluating the average-case running-time we assumed that the hash-function hashes evenly. However, it is in no way obvious that this actually happens.

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

Properties of a good hash function:

- the hash function must be deterministic
 - otherwise an element once placed into the hash table may never be found!
- despite this, it would be good that the hash function is as “random” as possible
 - $\frac{1}{m}$ 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 occurring in the input data, i.e. such patterns are broken by the function
 - for example, single letters are not investigated when hashing names but all the bits in the name are taken into account

- two methods for creating hash functions that usually behave well are introduced here
- lets assume that the keys are natural numbers $0, 1, 2, \dots$
 - 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 $b \in N = \{0, 1, 2, \dots\}$, then

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

- \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 choice, as then all the permutations of the same string end up into the same slot
- prime numbers are usually good choices for m , provided they are not close to a power of two
 - e.g. ≈ 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:

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

- which value should be chosen for A ?
 - all of the values of A work at least somehow
 - the rumor has it that $A \approx \frac{\sqrt{5}-1}{2}$ often works quite well

4.8 B-trees

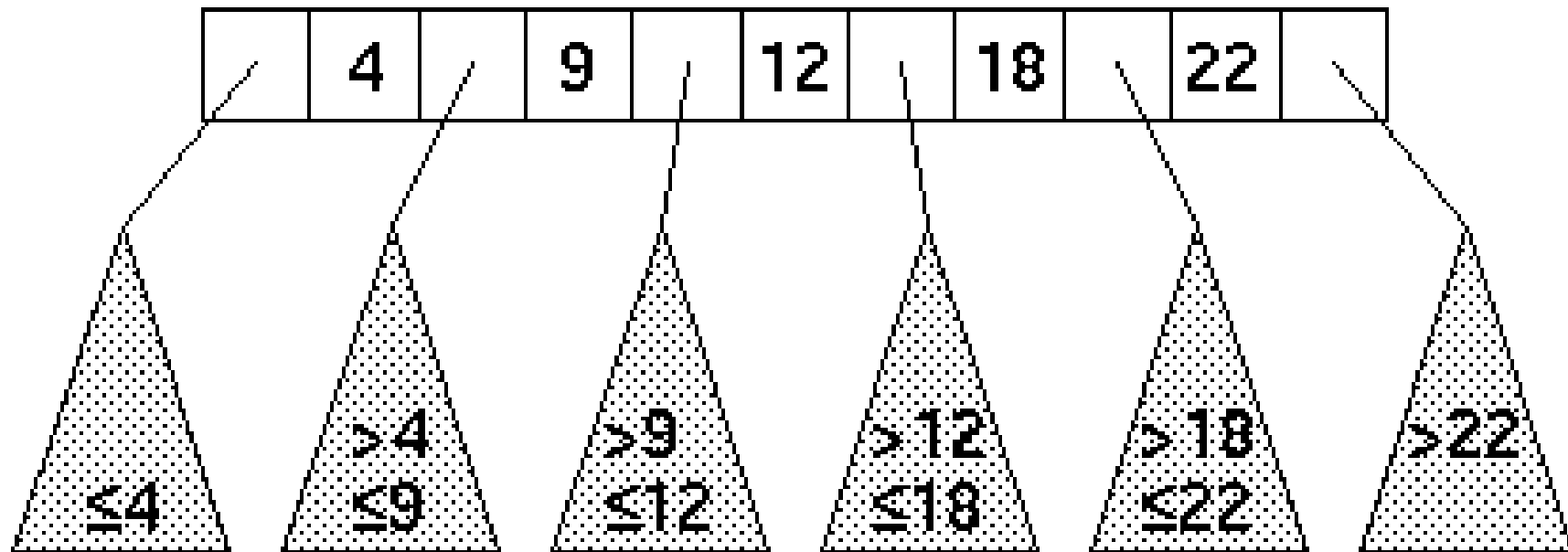
B-trees are rapidly branching search trees that are designed for storing large dynamic sets on a disk

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

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

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

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

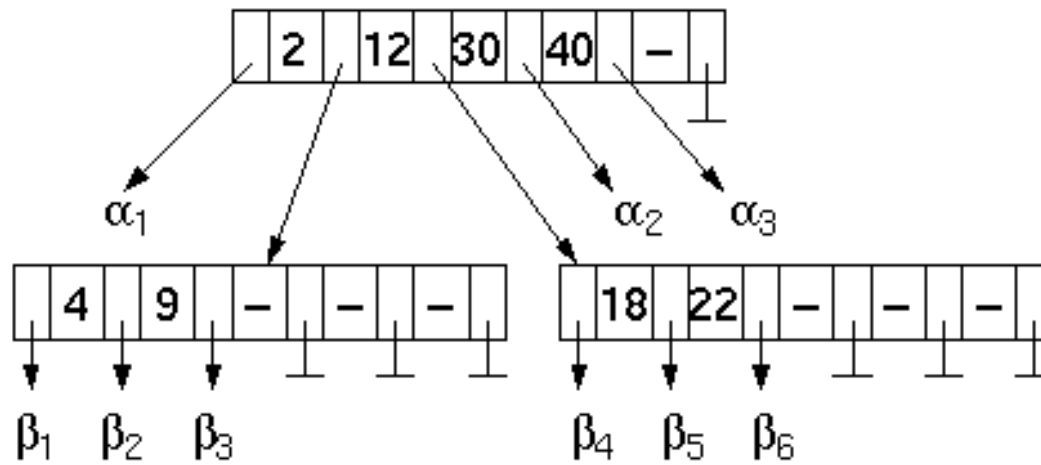
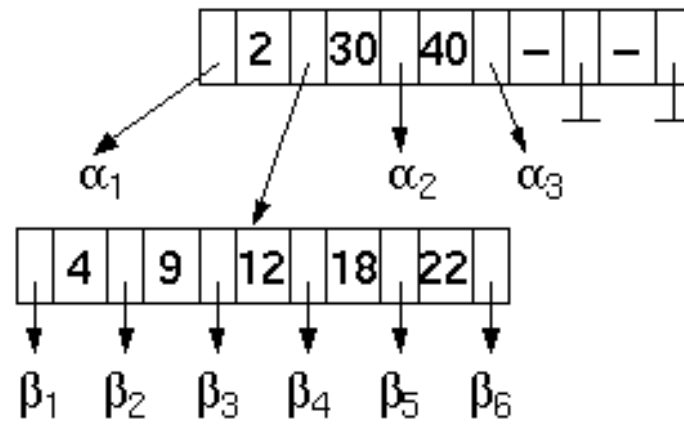


Inserting an element into a B-tree

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

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

The rest of the keys are split around the median key into a node with the smaller keys and a node with the larger keys.



Deleting a key from a B-tree is a similar operation to the addition.

- travel from the root to a leaf and always before entering a node make sure there is at least the minimum amount + 1 keys in it
 - this guarantees that the amount of the keys is kept legal although one is removed
- once the searched key is found, it is deleted and if necessary the node is 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