Poul Klausen

JAVA 18

Algorithms and data structures

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 18: ALGORITHMS AND DATA STRUCTURES

SOFTWARE DEVELOPMENT

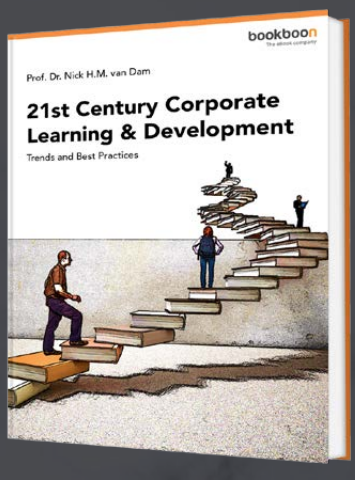
Java 18: Algorithms and data structures: Software Development
1st edition
© 2018 Poul Klausen & bookboon.com
ISBN 978-87-403-2616-1
Peer review by Ove Thomsen, EA Dania

CONTENTS

Free eBook on
Learning & Development

By the Chief Learning Officer of McKinsey

Download Now



FOREWORD

The current book is the eighteenth in this series of books on software development in Java. The book requires basic knowledge about Java and system development and here especially knowledge of the principles of object oriented programming as presented in the books Java 3 and Java 4. The book focuses on algorithms and presents several classic algorithms, but in conjunction with algorithms, the primary goal is to introduce algorithm analysis, and then allowing the reader to be able to compare and evaluate the complexity of algorithms with each other. In addition, the book treats the classic data structures such as lists and trees and shows how these data structures can be implemented in Java. In addition to shows the implementation of many of the classic data structures and discussing the applied algorithms, the book introduces several programming techniques that are useful/necessary to work as a professional software developer.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer

technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

1. NetBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
3. GlassFish as a web server and application server (from the book Java 11 onwards)
4. Android Studio, that is a tool to development of apps to mobil phones

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not "how to write" or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

ADT stands for *abstract data types* (as you can also think of as generic collections), and this book primarily deals with how classical collections such as lists, maps, etc. can be implemented in java. Instead of collections, I would like in this book to talk about *data structures*. Data structures must be implemented to be effective and therefore algorithms plays an important role, and the following will therefore focus on both data structures and algorithms.

You often hear that programming is difficult to learn, which is also correct. At least there are many who give up, but it is more because many not finds that programming is very exciting than because it is difficult, and in fact it has even become easier than it has been. It has become easier with the emergence of new programming languages, which can ease the work as you have to write less, and partly results in more robust programs, because the languages provide a much larger toolbox available than was previously the case. Conversely, the requirements for the programs to be created have grown correspondingly, and from the programmers, the challenges are really the same, to be able to write a good algorithm to solve a specific problem. All programs are not equally good written and there are lots of examples of programs that are ineffective and solve certain tasks in an inappropriate way and often the problem is that there has been no focus on the algorithm, but to a much greater extent the developers have had focus on getting the task solved without questioning on how the task was solved. Therefore, algorithms are a key issue in programming.

In fact, I should have started my history of programming with algorithms, as that is what it's all about. One should be good at algorithms to be a good programmer, but it does not work alone and another important thing is motivation, including the desire to work with programming and software development. You can learn about algorithms and methods, but you can not write finished programs that can run on a machine without having a programming language. Therefore, most entries to the programming and software development start with the language, as you quickly get ready to write completed programs. Most people want to quickly "put their fingers on the keys" and write programs that makes something interesting.

As mentioned, a data structure is just another name for a collection class. In addition to algorithms, the aim of the following is to explain the classic collection classes, but also to implement them as finished classes written in Java. You can ask why you need to do that, when they already exist as finished and tested classes, and there are two reasons for that. Firstly, it is important to know how the classes are made, so that you know how the classes' methods works in principle and have the characteristics that the documentation describes. Only with that knowledge available you are in practice able to make the right choices and choose the collection class that best suits the algorithm to be written. As another reason for

dealing with the implementation of classical collection classes, I also want to mention that studying these implementations introduces some techniques and methods that are important for practical programming. It is thus important to emphasize that it is not the goal of writing a new collection API and collection classes that will replace the classes that come as part of Java, but the goal is to explain how these classes are in principle implemented and by doing so, apply some techniques that are useful/necessary to master as a professional software developer.

Since many of the data structures and also the examples that will illustrate the applications of both algorithms and data structures make up a lot, I have only included a part of the code in the text of the book. In order to get the most out of it, therefore, like the other books in this series, it is important that you also study the finished programs, and especially how the data structures are implemented in Java. The most important algorithms and data structures have been added to a class library called *palib*, and this library is used by the book's examples.

The book is generally divided into two parts, the first part deals with algorithms, while the other part deals with data structures, but since the implementation of data structures is just about algorithms, one can also think of the entire book as a book on algorithms.

The goal of the first part about algorithms (the next 4 chapters) is to provide examples of some classic algorithms, including special algorithms for search and sorting. In addition, an introduction is given to how to ensure that an algorithm is correct and gives the right result, as well as an introduction to how to evaluate the effectiveness of algorithms in relation to each other.

The second part of the book comprises 5 chapters and deals with data structures and collection classes, the most important of which is the implementation of lists and trees and in connection therewith a variety of algorithms. The implementation of some of these data structures is both complex and comprehensive, and the book primarily describes the principles, while the code must be searched in the book's source files.

The source code, like the other books, contains a number of examples, but the largest part is implementations of algorithms and data structures collected in the class library called *palib*, and most of the examples use this library. The same goes for most of the book's exercises and tasks.

2 THE CONCEPT OF ALGORITHMS

An algorithm is a method that describes how to solve a problem through a final number of steps, and an algorithm is thus a solution that – usually from an input – solves a specific problem. Algorithms are formulated in an algorithm language where they can be expressed in a short and accurate way. A programming language is an example of an algorithm language, but it may often be easier to use a simpler language to be free from the many details of the programming language. In addition, many algorithms have a common nature in which a whole family of algorithms can be described from a common template. In addition to the programming language Java, this book does not use a formal algorithm language, but instead algorithms are sometimes described informally in a middle between common language and Java program code. In the following, I will typically describe algorithms informally using a common language and subsequently as finished algorithms in Java.

As an example of an algorithm, I will look at an algorithm that raises a number n in the power of p – thus calculating n^p. This task (algorithm) can be described as follows:

```
u = 1
while (p > 0)
{
 u = u * n
 --p
}
return u
```

The above is the informal version where the algorithm partially is written in Java, but I have not fully complied with the syntax of the language. The aim is to describe the solution method, but without having to deal with the many details of the programming language, and if you know a little about programming and Java, it is easy to understand what the above algorithm is doing.

For all algorithms, you has to answer two questions:

- do it solve that task – is the algorithm correct?
- how good is the algorithm?

The first is of course a requirement, while the other is a matter of quality.

If p is negative, the algorithm gives result 1 (the loop will not be executed), which is incorrect. If p is 0, the result is also 1, which is correct. If p positive, the loop is performed, and for every iteration p is decremented with 1 and the loop will exactly be repeated p times. For

each repeat, u is multiplied with n and since u from start is 1, the value of u – when the loop stops – is n multiplied by itself p times and thus has the value n^p. That is that the algorithm gives the correct result, assuming that p is not negative. Wee say that $p \geq 0$ is a precondition.

With regard to the quality of the algorithm, it is a goal that the algorithm should use as few resources as possible:

- that it minimizes the consumption of memory and thus does not use more variables than necessary
- that it minimizes the consumption of the processor and thus does not perform more statements than necessary

In this case, in addition to the input parameters n and p (the values the algorithm should be applied to), the algorithm uses a single variable for the result, and the important thing is that the requirement for memory is constant and independent of the two input parameters. Regarding the algorithm's consumption of the processor, the number of instructions is determined by the number of times the loop repeats. That is, the number of instructions grows with the input parameter p so that the calculation for large values of p requires more

instructions. In this case there is no problem in it – it is a quite effective algorithm, as it is limited how big *p* can be before an overflow occurs.

The algorithm is formulated so that it is independent of the data type, but it is easy to convert the algorithm to Java, so it works on a specific type. It is a very simple algorithm, but it expresses a method that can be used when just p is a non negative integer and n is a type (a number) for which multiplication is defined. Thus, the algorithm is partially formulated independently of a specific data type. For example, if you would realize it in Java for integers, it could be written as:

```
// It is assumed that p is not negative
public static long power(long n, int p)
{
 long u = 1;
 for (; p > 0; --p) u *= n;
 return u;
}
```

Here, it is utilized that different constructions in the specific language allow for the expression of the algorithm in a simpler way, which one should of course use.

You should note that an algorithm has input parameters, which are the values on which the algorithm works. To the extent there are preconditions attached to these parameters, I indicate it as a comment in front of the algorithm. Instead, you could let the method raise an exception if the preconditions have not been met and it's a matter of choice whether where you do the one or the other. If you just post a comment, it's a contract that you ask the user to adhere to and if the user do it, the one who has written the algorithm guarantees that it gives the right result while the programmer guarantees nothing if the algorithm is used with incorrect preconditions. You are talking about contract programming, and it is the user of the method that is responsible for compliance with the preconditions. Contract programming simplifies the code, while the use of exception handling helps the user always to performs the method with the right preconditions, but in return, the algorithm must test whether the preconditions is met.

The above algorithm is implemented to work for values of the type *lo*ng (the parameter *n*). If the algorithm also has to be implemented for values of the type *double*, one will often write an overridden version of the method, and where I this time have let the method tests the precondition:

```
/**
 * Calculates the p'th power of x by a loop that iterates p times.
 * @param x The number for the power calculation
 * @param p The exponent
```

```
 * @return The p'th power of x
 * @throws Exception If p is negative
 */
public static double power(double x, int p) throws Exception
{
 if (p < 0) throw new Exception("The exponent must be none negative");
 double y = 1;
 for (; p > 0; --p) y *= x;
 return y;
}
```

When writing algorithms whose input parameters are simple types, it is very common that you override the algorithm's method for different types, and it is up to the programmer who writes the algorithm to ensure the necessary overrides. In this case, it is an algorithm that works on numbers, and you typically have to write overrides for the different number types, but in other contexts there will be algorithms that work on class types, and here I will often write the algorithms as generic methods. Please note that this practice is widely used in Java.

In the review of algorithms, like the above, I will focus on both the algorithm and thus the how to solve the problem, but also how the algorithm can be implemented in Java.

2.1 EXAMPLES OF ALGORITHMS

The following continues the discussion of what an algorithm is, and also how to formulate algorithms using an informal language. In addition, I will give examples of classic algorithms.

With regard to writing an algorithm in an informal language or using a programming language, the choice is not clear. Modern programming languages such as Java is so self-documenting that you often can formulate an algorithm as clearly and easily in the language itself as using an informal pseudo language. In fact, there is even a tendency for algorithms formulated in an informally language to be longer than the corresponding algorithm written in a programming language, and if so, of course, it does not make sense to describe algorithms informally. If you are proficient in your language, you think and formulate equally well using the terms of the language, and many therefore avoid descriptions of algorithms informally. In this book, however, I would like to make some use of an informal language, because I think that at least in complex algorithms it is worth paying off the details of the programming language to better focus on the the most important steps in the algorithm. On the other hand, you should avoid making the informal algorithms too detailed and straightforward to use an informal language as a technique for problem decomposition. You should avoid writing informal algorithms, that in fact is the program written in English.

Basically, an algorithm contains four things:

- sequences that is a series of orders/instructions that are performed in the order in which they are written in the algorithm
- conditions where the execution of one or more operations are performed depending on a condition
- loops where one or more operations are performed as long as a condition is met
- performing a sub algorithm where the current algorithm starts another algorithm

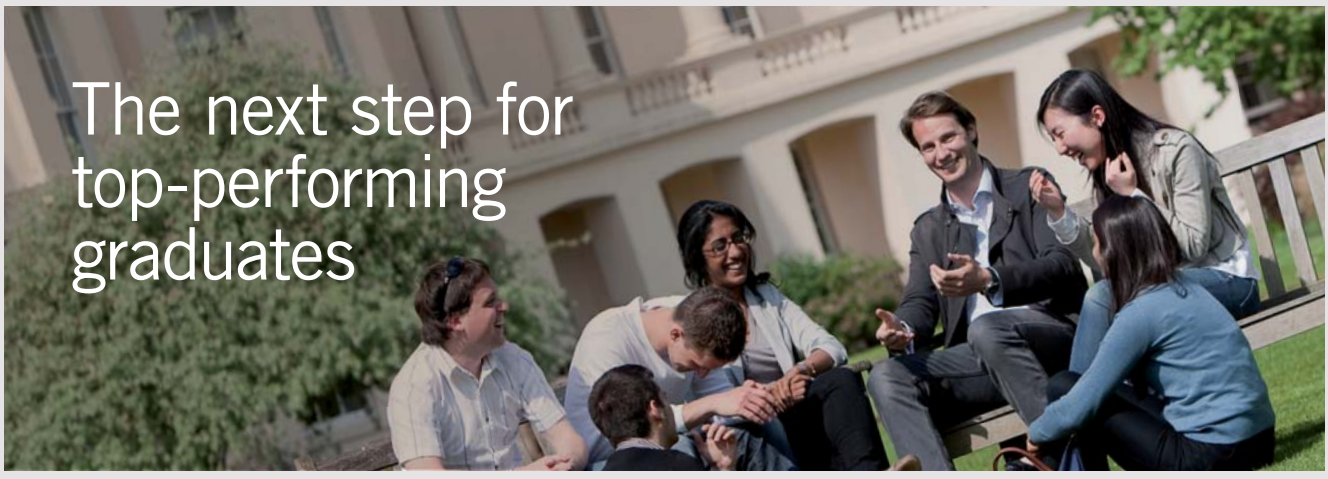
Difference series

By a difference series you should understand a series of number

$$a_1, a_2, a_3, \ldots, a_n$$

with the characteristic that you all the time go one step forward in the row by adding the same number, for example

4, 7, 10, 13, 16, 19, …



Masters in Management

Designed for high-achieving graduates across all disciplines, London Business School's Masters in Management provides specific and tangible foundations for a successful career in business.

This 12-month, full-time programme is a business qualification with impact. In 2010, our MiM employment rate was 95% within 3 months of graduation*; the majority of graduates choosing to work in consulting or financial services.

As well as a renowned qualification from a world-class business school, you also gain access to the School's network of more than 34,000 global alumni – a community that offers support and opportunities throughout your career.

For more information visit www.london.edu/mm, email mim@london.edu or give us a call on +44 (0)20 7000 7573.

* Figures taken from London Business School's Masters in Management 2010 employment report

London Business School

where you each time adds 3. It can also be expressed that the difference between a number and the previous number is constant (hence the name):

$$a_k - a_{k-1} = d$$

If so, you say that you have a difference series with the difference *d*. As other examples of difference series are:

5, 8, 11, 14, 17, 20, 23, …, 2 + 3n, ….
2, 4, 6, 8, 10, 12, …, 2n, ….
1, 2, 3, 4, 5, 6, …, n, ….

where the first has difference 3, the second difference 2 and the last difference 1.

The goal is now to write an algorithm that can determine the sum of the numbers in a difference series. To do that, you must know three things:

- the first number *a* (where the series starts)
- the difference *d* (how much you have to step forward to get to the next number)
- how many numbers *n*, that should be included in the sum

The algorithm can then be written as follows:

```
for (1 to n)
{
 adder a to sum
 adder d to a
}
```

Again, it's a simple algorithm, and it's easy to figure out that it gives the correct result: For each repeat, the sum is increased by the next number in the difference series, and *a* is then changed to reference the next number. It's easy to write the algorithm as a *static* method in Java:

```
public static long sum(int a, int d, int n)
{
 long s = 0;
 for (int i = 0; i < n; ++i)
 {
  s += a;
  a += d;
 }
 return s;
}
```

Note that the method uses the above algorithm. The algorithm basically consists of a loop which is repeated *n* times that is that the algorithm determines the sum of the *n* first numbers. Thus, the complexity of the algorithm, regarding the number of statements (operations), is determined by *n*, and for large values of *n*, the algorithm requires many operations. On the other hand, its space consumption is independent of *n*, since it only uses a single additional variable (the variable in the loop) regardless of the size of *n* in addition to the parameters.

If you consider the above method, you will decide in the last iteration of the loop the variable *a* is assigned a value which is not used. This can be easily overcome by changing the statements:

```
public static long sum(int a, int d, int n)
{
 long s = a;
 for (int i = 1; i < n; ++i)
 {
  a += d;
  s += a;
 }
 return s;
}
```

There is no big difference, but in the last version, the loop is performed once fewer, and the algorithm is therefore more efficient. However, the difference is hardly measurable, but it shows that, after writing an algorithm, you can often improve/optimize by critically referring to how it is written. However, I would like to warn some kind of optimizations, and be aware that they do not go beyond readability. This is not the case here, but it is important that an algorithm is easy to read and understand.

In one way or another, the above is the natural algorithm to determine the sum of a difference series as it directly depends on the definition, but there are other options, and the question is whether you can do it better – and you can. If you have a difference series

$$a_1, a_2, a_3, \ldots, a_n$$

with difference d, you can determine the sum of the n numbers based on the formula

$$s = (a_1 + a_n) * n/2$$

that is as the sum of the first and the last number and times the number of numbers divided by 2. It is a mathematical formula which is not particularly difficult to prove. You can therefore also write the algorithm as follows:

```
b = the last number in the row (the n'th number)
s = (a + b) * n / 2
```

Now the problem is reduced to determine the last number. If a difference series starts with the number *a*, and you want to determine the *n*th number, you must go *n-1* step forward and thus you can calculate the *n*th number as

$$a_n = a + (n-1)d$$

That is as the *n*th number based on the initial value and the difference, and the above algorithm can thus be implemented directly as:

```
public static long sum(int a, int d, int n)
{
 long b = a + (n - 1) * d;
 return (a + b) * n / 2;
}
```

Note that it also applies to the special case where *n* is 0. The last algorithm is the best. The first version solves the problem using a loop, and if *n* is large, it will lead to many operations (more, maybe simple operations has to be performed for each repeat of the

loop). In contrast, the last algorithm can solve the problem by only two calculations, which is far more efficient, and in addition, the algorithm is independent of the size of the problem – it does not depend on the size of *n*. It is said that the last algorithm has constant time complexity.

Note, in the case where $a = 1$ and $d = 1$, the algorithm determines the sum

```
1 + 2 + 3 + 4 + … + n
```

In this case, the difference between the two algorithms will only be measurable if *n* is large, but the example shows first that a problem can often be solved with very different algorithms and secondly that different algorithms are not equally beneficial for using the machine's resources. It is a goal to write algorithms that are as effective as possible, and that is, algorithms that minimize the resource consumption.

EXERCISE 1: A DIFFERENCE SERIES

You must write a class that can represent a difference series with numbers of the type double, when the constructor must initialize the first number and the difference. In addition, the class must implement two methods that return the *n*th number and the sum of the first *n* numbers, respectively:

```
public class DifferenceSeries
{
 /**
  * Calculates the n'th element in the difference series
  * @param n Index of the element
  * @return The n'th element
  * @throws Exception If n is less than 1
  */
 public double element(int n) throws Exception
 {
 }

 /**
  * Calculates the sum of the first n numbers in the difference series.
  * @param n The number of values in the sum
  * @return The sum of the first n values in the series
  * @throws Exception If n is less than 1
  */
 public double sum(int n) throws Exception
 {
 }
}
```

2.2 RECURSION

Among other things, within mathematics, it is very common to define formulas and definitions recursively, where a formula is expressed by itself. For example you can define a difference series with difference d as follows:

$$a_n = \begin{cases} a & n = 1 \\ a_{n-1} + d & n > 1 \end{cases}$$

Another example is the formula for a factorial

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

where the term is just expressed by itself: If you know *(n-1)!* can you also decide *n!*, and when *0!* is known, one can decide 1*!* and then you can decide *2!* and then again *3!* etc.

If you want to write an algorithm that can determine *n!*, You can immediately apply the above formula:

```
if (n == 0) return 1
return n * Factorial(n - 1);
```

Note that the algorithm is a direct rewriting of the mathematical definition, and the crucial thing is that the algorithm is written by itself – the algorithm calls itself, and an algorithm that calls itself is called recursive algorithm.

In Java, an algorithm is written as a method, and a method can, for example, call another method, and especially, itself. Thus, Java supports recursion, and the algorithm can therefore be implemented directly as follows:

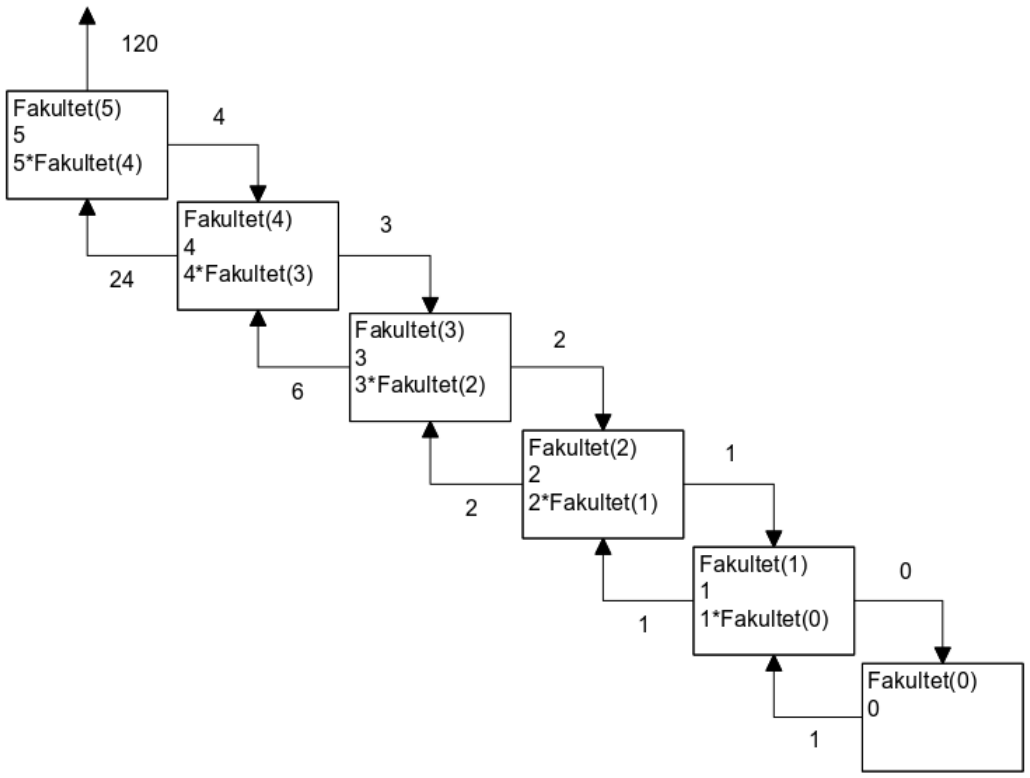
```
// n must be none negative
public static long factorial(int n)
{
 return n == 0 ? 1 : n * factorial(n - 1);
}
```

Recursive algorithms are often simpler than a corresponding non-recursive algorithm (also called an iterative algorithm) solving the same problem, but it is often more difficult to find out where a recursive algorithm is correct. In this case, however, it is easy because the algorithm is merely a copy of the mathematical definition.

It is also possible (easy) to implement the algorithm to determine the factorial without the use of recursion – thus as an iterative algorithm:

```
public static long factorial(int n)
{
 long u = 1;
 for (int i = 2; i <= n; ++i) u *= i;
 return u;
}
```

and also here it is easy to consider that the algorithm works and gives the correct result. In this case, the difference is not very clear, but actually the last one is more effective than the first one, as it uses less memory: Each time an algorithm calls itself, parameters need to be transferred and it takes memory and it takes time. Below is illustrated what happens if you use the recursive version of *factorial()* to determine the factorial of 5. The boxes are or illustrate activation blocks on the stack, and only when you reach the bottom and determine the last value for 0!, the calls begin to return and thus release the activation blocks.



In the vast majority of cases, it does not matter, but generally, an iterative algorithm will use less memory than a recursive.

There is always reason to be aware of what recursion means and below I will show an example where recursion is immediately the natural choice but where it's definitely the wrong choice and one ends with an algorithm that's so ineffective that it is unusable for large input values.

The advantage of recursion is rarely about ever efficiency, but instead of writing the algorithm shorter and sometimes also more readable. In fact, I will later show examples of algorithms that are difficult to write iterative, but where recursive solutions are both short and easy to understand.

As a last remark, there are many who find that recursion is difficult and that it is difficult to write and read recursive algorithms. It is undoubtedly correct, but vice versa, it helps with practice. Therefore, another example.

You have to write a method that has a string as a parameter and the method should returns the string with the characters in reverse order. That is, if the method is called with the string *"ABCD"* as parameter, it should return *"DCBA"*. The problem can of course be solved iterative with a loop that iterates through from behind the string's characters, but you can also write the solution recursively as:

```
public static String swap(String line)
{
 if (line.length() == 0) return "";
 return swap(line.substring(1)) + line.charAt(0);
}
```

Again, it is an example of an algorithm where recursion has no advantage, but the example nevertheless shows how recursive algorithms typically look. The algorithm consists of two parts, the first one being simple and can be immediately solve the problem (here it is the case if the string is empty). The other is the general case where the problem is solved using the algorithm itself, but applied to a smaller problem than the original – in this case *swap()* of a string, which is a character shorter than the original string. If you repeat it sufficiently often, you will finally reach the trivial problem that can be solved immediately.

EXERCISE 2: A PALINDROME

You should write a program that has a method that tests whether a string is a palindrome. It is a string that is spelled even from the front and from the back when spaces and special characters are to be ignored, and there should be no distinction between upper and lowercase letters. Examples of palindromes include:

- Otto
- A tin mug for a jar of gum, Nita.
- A nut for a jar of tuna
- A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal-Panama!

The method that tests if a string is a palindrome must be written as a recursive method:

- If you have the simple and trivial case where the string is empty or simply consists of a single character, it is a palindrome.
- Otherwise, if the first character should be ignored, then the remaining characters should be tested for a palindrome.
- Otherwise, if the last character should be ignored, almost the other characters should be tested for a palindrome.
- Else, if the first and last characters are the same, then the rest of the string is tested for a palindrome.

2.3 SEARCH

A classic issue is searching an element in an array. If you do not know anything about how the array's elements are sorted, you can not do much more than search linearly, that is, start from the beginning and pass through the array from start to finish and compare each element with what is searched for . The process can be expressed in the following algorithm where you search in an array *arr* after an element *elem*:

```
loop (over all elements element in arr)
{
 if (element == elem) element is found
}
element is not found
```

Note how the algorithm works by searching the entire array. If you find the element, you are done and the algorithm can terminate with a message that the element is found. If you get through the loop, it's because the element is not there and you can end up with a message that it does not exist. In the worst case, you must visit all elements to find out that an element is not available.

It is easy to implement linear search in *Java*. Below is a generic search method that searches for an element in an array. If the element exists, the method returns the index of the element's position in the array. If the element is not found, the method returns -1 (it may not be an index) as an indication that the element was not found.

```
public static <T> int linSearch(T[] arr, T elem)
{
 for (int i = 0; i < arr.length; ++i) if (arr[i].equals(elem)) return i;
 return -1;
}
```

You should note that the only requirement for parameter type *T* is that the type implements *equals()* with value semantics. Also note that if the same element is found multiple times, the method returns the index on the first element (the element with the smallest index).

The method *linSearch()* is defined as a *static* method in a class *Tools*, which is a class in a class library called *palib*. This class library will contain more methods and classes presented in this (and the next) book, and the goal is to write some utility classes that could be used as alternatives to corresponding methods and classes implemented in the package *java.util*.

Linear search is a widespread search algorithm, that you typically use, when the array is small or you do not know anything about the elements in the array. However, if you know that the array searched is already sorted, you can do it better using *binary search*. The algorithm is:

```
while (array is not empty)
{
 elem = the middle element
 if (elem is equal with element) the element is found
 if (elem is less than the element)
 {
  repeat the search, but only on the left half of the array
 }
 else
 {
  repeat the search, but only on the right half of the array
 }
}
the element is not found
```

The idea in this algorithm is that, for each repeat, you halve the number of elements to be searched, and in that way you get done quickly. The main problem with implementing the algorithm is to control what it is for a subset of the array to be searched. The problem can be solved using two indices *a* and *b*, which refer to the subset's left end and right end points respectively:

```
public static <T extends Comparable<T>> int binSearch(T[] arr, T elem)
{
 for (int a = 0, b = arr.length - 1; a <= b; )
 {
  int m = (a + b) / 2;
  if (arr[m].equals(elem)) return m;
  if (arr[m].compareTo(elem) < 0) a = m + 1; else b = m - 1;
 }
 return -1;
}
```

Here you should note the following:

- The method is a generic method, and since it is a requirement that the array's elements can be arranged, the parameter type implements the interface *Comparable*.
- That binary search only works if the array searched is sorted and then binary search can not be used as a general search strategy.
- That *a* from the beginning points to the first element and *b* on the last element in the array, and how one of these variables is changed for each iteration – if the element is not found.
- That the method, similar to the corresponding linear search method, returns the element's index if it exists and otherwise -1. If the element is found multiple times, it is the index of the first element found you will you encounter.

The question is then, how good binary search is compared to linear search. Binary search is not only better, it is simply much better, and I will later returns to what it means, under the mention of algorithm complexity. Now it is not the case that you do not have to apply linear search, and in smaller arrays the difference in efficiency is not pronounced, but in large arrays the difference can be significant.

If you look at the binary search algorithm, you can reformulate it a bit:

```
binSearch(arr, elem)
{
 if (the array is empty) return false // the element is not found
 if (elem is equal the midle element) return true // the element is found
 if (elem is less than the midle element)
 {
  binSearch(the left half of the array, elem)
 }
 else
 {
  binSearch(the right half of the array, elem)
 }
}
```

The algorithm is now recursive, and you can write binary search as a recursive method in Java as follows:

```
public static <T extends Comparable<T>> int binSearch1(T[] arr, T elem)
{
 return binSearch(arr, elem, 0, arr.length - 1);
}

private static <T extends Comparable<T>>
int binSearch(T[] arr, T elem, int a, int b)
{
 if (b < a) return -1;
 int m = (a + b) / 2;
 if (arr[m].compareTo(elem) == 0) return m;
 if (arr[m].compareTo(elem) < 0) return binSearch(arr, elem, m + 1, b);
 return binSearch(arr, elem, a, m - 1);
}
```

A recursive method requires two additional parameters that define the range to be searched for. But otherwise, there is not much to explain and it is just the above algorithm written in Java.

ericsson.
com

Shaping tomorrow's world – today
Our business is at the heart of a connected world – a world where communication is empowering people, business and society. Our networks, telecom services and multimedia solutions are shaping tomorrow. And this might just be your chance to shape your own future.

It's a people thing
We are looking for high-caliber people who can see the opportunities, people who can bring knowledge, energy and vision to our organization. In return we offer the chance to work with cutting-edge technology, personal and professional development, and the opportunity to make a difference in a truly global company.

We are currently recruiting both new graduates and experienced professionals in four areas: **Software, Hardware, Systems** and **Integration & Verification**.

Are you ready to shape **your** future? Begin by exploring a career with Ericsson. Visit www.ericsson.com/join-ericsson

ERICSSON

In order to get the same interface as the original algorithm, a version of the method is written that only calls the recursive version. This means that the algorithm is used in the same way, and that the user can not see if it is the recursive or non-recursive version. The recursive version is *private* and thus a private helper method, which is introduced alone to give the method the correct parameters. There is good reason for to be aware of that, as it is a technique that you will see used many times in the following. The public version of the two methods is referred to as a stub.

If you look at how binary search works by halving the number of elements to be searched in each iteration, just repeating the same operation, but at an interval that is half the length, it's obvious to consider a recursive implementation. In this case, the recursive solution has no advantages. It is neither shorter nor more readable than the iterative, and the efficiency is essentially the same, but with a small plus to the iterative if there are differences.

The project *Algorithms* has three test methods that uses the above search algorithms.

2.4 THE FIBONACCI NUMBERS

The first Fibonacci number is 0 and the next is 1. Then the next Fibonacci number is generated as the sum of the two previous ones. That is, it's the number series:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ….
```

Formally, you can define the Fibonacci numbers as follows:

$$\text{fibo}(n) = \begin{cases} n & n = 0,1 \\ \text{fibi}(n-2) + \text{fibo}(n-1) & n > 1 \end{cases}$$

and thus it is a recursive definition. If you want to write a method that returns the *n*th Fibonacci number, it is therefore obvious to try using a recursive method as it is a direct copy of the formal definition:

```
public static long fibo1(int n)
{
 if (n < 2) return n;
 return fibo1(n - 2) + fibo1(n - 1);
}
```

Note that the method is simple and easy to understand. If you try out the method, everything looks immediately fine, but if you try the method for large values of the input parameter *n* you will find that the method becomes slow for finally to stop completely. The reason is

that the number of recursive calls explodes, as each call leads to 2 new calls, and each of them again to two new calls and so on.

The problem is not that there is anything wrong with recursion, but simply that recursion is not the solution in this case, and it is also an example, that you should always think about recursive methods, as it can be difficult to spot the consequences. Should you write the above method, it should therefore be iterative, which is also easy, and the difference between the efficiency of the two algorithms is very clear when you tests the algorithms.

```
public static long fibo(int n)
{
 if (n < 2) return n;
 long f1 = 0;
 long f2 = 1;
 while (n-- > 1)
 {
  long f3 = f1 + f2;
  f1 = f2;
  f2 = f3;
 }
 return f2;
}
```

The project *Algorithms* has two test methods – *test04()* and *test05()* – which tests the above algorithms for the Fibonacci numbers.

Thus, the Fibonacci numbers are another example of the fact that algorithms can be written in several ways and it is far from certain that the most obvious is the best. Incidentally, the method used in the last iterative solution is referred to as dynamic programming, and it is a topic that I will return to in the next book in this series.

2.5 EUCLID'S ALGORITHM

One of the oldest algorithms is Euclid's algorithm, which determines the largest common divisor of two positive integers. It's a very simple algorithm. If *n* and *m* are positive integers, I want to denote with *gcd(m, n)* their largest common divisor. For example, considering the numbers 30 and 42, their largest common divisor is 6. That is, *gcd(30, 42) = 6*.

The idea of the algorithm is:

```
repeat
{
 if m and n are equal it is the greatest common divisor
 else subtract the smallest number from the greatest
}
```

In this case, the algorithm is not so easy to figure out. First of all, it is not clear that the algorithm will stop at all and, secondly, it is not easy to see that it gives the right result.

If you look at a single iteration of the loop, there are two options:

1. the two numbers are the same and you are done (the largest common divisor of two identical numbers is of course the numbers itself)
2. one of the two numbers becomes smaller than it was before, but not 0

The last can not be repeated forever, and sooner or later it must happen that the two numbers become the same. That is, that the algorithm stops.

To show that the algorithm is correct, suppose, for example, that $n >= m$. If d is divisor in both n and m, there are positive numbers a and b so $n = ad$ and $m = bd$ and hence $n - m = (a - b)d$. That is, that d is divisor in both m and $n - m$. Suppose that d is divisor in both m and $n - m$. Then there are positive numbers a and b so $n - m = ad$ and $m = bd$ and hence $n = (n - m) + m = (a + b)d$. That is, that d is divisor in both m and n. You can then conclude

```
if n >= m is gcd(n,m) = gcd(n-m,m)
```

From this, it can be concluded that the loop in the above algorithm does not change the largest common divisor of n and m and because the algorithm stops, it is thus correct.

Being able to determine the largest common divisor of two integers is interesting and although the Euclid algorithm in the above form is not the most obvious choice (due to ineffectiveness) the algorithm is nevertheless interesting as an algorithm for solving a none trivial problem, and as a very simple algorithm and as an example of an algorithm where it is not obvious that it solves the problem. Below I have shown the algorithm written in Java:

```
public static long gcd(long m, long n)
{
 if (m == 0) return n;
 if (n == 0) return m;
 while (n != m) if (n > m) n -= m; else m -= n;
 return n;
}
```

Above, I have stated that the arguments should be positive. If $a > 0$ is a divisor in both a and 0 and thus that $gcd(a, 0) = a$. That is, the largest common divisor also makes sense in the case where the arguments are 0. The algorithm therefore starts with a special case, and its precondition is that both arguments should not be negative.

The problem with the above algorithm is that there can be too many iterations. If the one number is very large (max int) and the second number is very small (1), you have to subtract many times and the algorithm will take a very long time, but you can do better.

Suppose again that $n >= m$ and assume d is divisor in both n and m. That is, that $n = ad$ and $m = bd$ for some positive numbers a and b. If $r = n \% m$, then $n = qm + r$, where q is the quotient of the division n / m. It follows that $r = n - qm = ad - qbd = (a - qb)d$ and hence d is divisor in r. Suppose that $n = qm + r$ where $r = n \% m$ and assume that d is divisor in both m and r. Then there are numbers a and b such that is, $m = ad$ and $r = bd$ and hence $n = (qa + b)d$ and d is divisor in both m and n. Thus it can be concluded that

d is divisor in n and m if and only if d is divisor in m and $n \% m$

and hence again that

```
gcd(n,m) = gcd(m,n%m)
```

when *n* >= *m*. You can therefore write an improved version of Euclid's algorithm as follows:

```
// Returns the largest common divisor of n and m.
// n and m must both be greater than 0
public static long gcd(long m, long n)
{
 return n >= m ? gcd1(n, m) : gcd1(m, n);
}

private static long gcd1(long n, long m)
{
 if (m == 0) return n;
 return gcd1(m, n % m);
}
```

Here the last method is a private helper method. It assumes that the first parameter is greater than or equal to the last one. It is a recursive method, and since *n % m* is always less than *m*, *m* must sooner or later become 0, and thus the method will stop. Corresponding to the above argumentation, it will exactly stop with the largest common divisor of *m* and *n*, and the method therefore gives the right result. The first method – *gcd()* – is the method that the user sees and is only intended to call *gcd1()* with the correct order of the parameters. If you try the method, you'll find that *gcd()* is not just faster than Euclid's *gcd()*, but it's simply much faster. Again, it is an example that algorithms that solve the same task can be very different in terms of efficiency and, in practice, it may be of great importance if you choose the one or the another version of an algorithm.

The test method *test06()* in the project *Algorithms* tests the last of the largest common divisor algorithms.

2.6 SUM OF DIGITS

In the book Java 1, I looked at the calculation of the cross sum for a number, which determines the sum of the digits of the number and repeats it until the sum is less than 10. The cross sum of a number is thus always a single digit and the cross sum has sometimes been used as a control digit for number systems.

An algorithm that determines a cross sum can be written as

```
while (n < 10)
{
 n = the sum of the digits in n
}
```

and the problem is therefore reduced to determine the sum of the digits in an integer:

```
// Returns the cross sum of digits in n, when n is not negative.
public static int crossSum(long n)
{
 while (n > 9) n = digitSum(n);
 return (int)n;
}

public static int digitSum(long n)
{
 if (n == 0) return 0;
 return (int)(n % 10) + digitSum(n / 10);
}
```

Here is the last method *digitSum()* written recursive – just to point out the syntax again. The test method is called *test07()*.

2.7 PRIME NUMBERS

As another example, the following algorithm determines whether a number is a prime number (it's also an algorithm that I've shown before):

```
if n is 2, 3, 5 or 7 n is a prime
if n is less than 11 or if n is even n is not a prime
k = 3
while (k is less than or equal the square root of n)
{
 if k divides n then n is not a prime
 k += 2
}
n is a prime
```

Note that the algorithm begins by processing a special case. You often see algorithms that works that way. When the special case is gone – here if the number is small or even – you can address the general case. To find out if a number is a prime, try to find a number less than the current number that goes up. If you can, it's not a prime, but if there is no number going up, it's a prime. Due to the special cases, it is enough to try the odd numbers (2 is the only equal prime). If a divisor and the number can be written as a product $n = ab$, then one of the numbers a and b will be less than (or equal to) the square root of n and it is therefore enough to try with all numbers less than or equal to the square root of n. The algorithm can now be written in Java as follows:

```
// Returns true if n is a prime and else false.
public static boolean isPrime(long n)
{
 if (n == 2 || n == 3 || n == 5 || n == 7) return true;
 if (n < 11 || n % 2 == 0) return false;
 for (long k = 3, m = (long)Math.sqrt(n) + 1; k <= m; k += 2)
  if (n % k == 0) return false;
 return true;
}
```

The algorithm is relatively effective, but to determine whether a number is a prime number, many divisions are required, and too large numbers, the algorithm takes a long time. To test if an integer is a prime, actually has a lot of practical interest, and, among other things, in terms of cryptography, it has the interest to determine large prime numbers. Therefore,

in the field of computer science (mathematics), the focus has been on the development of algorithms, which more effectively than the above can test if a positive integer is a prime and especially how to determine the sequence of prime numbers. The test method for the above algorithm is called *test08()*.

2.8 SWEEP

Sweep is not an algorithm, but it is a name for a family of common algorithms which are similar. For example, if you look at the linear search algorithm, it consists in run through an array from start to finish, and for each element, you should do something, and in linear search, compare the element with another. There are many algorithms that works completely after the same template:

- determine the largest element in an array
- determine the sum of elements in an array
- print all elements in an array

and the difference is solely what to do with the individual elements or whether it is necessary to pass through all the elements, or the iteration can be completed before reaching all the way (as with linear search). An algorithm of the above nature is sometimes called a sweep algorithm, which means it "sweeps" over the array's elements. In the next chapter I will look at the efficiency of algorithms, and here it will be seen that all sweep algorithms have the same efficiency, and in practice they are considered to be effective. If a sweep algorithm works on an array of a given size, the algorithm typically needs to do something with each element, and in principle it should perform the same number of operations as there are elements in the array. It is said that the complexity of the algorithm depends linearly on the array size.

Sometimes sweep is called for an algorithm template similar to that it is a common template for an entire family of algorithms. The template is simple and is just a loop over a collection:

```
for (e in collection)
{
 action(e)
}
```

Using Java's functional interfaces, you can write some general algorithms for sweep over a collection. As an example, the following method performs a sweep over a collection *values*, where the last parameter is an operation to be performed on the individual elements:

```
public static <T> void sweep(Iterable<T> values, Consumer<T> action)
{
 for (T t : values) action.accept(t);
}
```

The algorithm can be expanded with a parameter *ok*, which indicates which elements to be processed:

```
public static <T> void sweep(Iterable<T> values, Predicate<T> ok,
 Consumer<T> action)
{
 for (T t : values) if (ok.test(t)) action.accept(t);
}
```

Below is shown a method that uses the algorithm to prints all elements in a list and then only those elements that are primes:

```
private static void test09()
{
 List<Integer> list = Arrays.asList
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20);
 Tools.sweep(list, t -> System.out.print(t + " "));
```

```
 System.out.println();
 Tools.sweep(list, t -> Functions.isPrime(t), t -> System.out.print(t + " "));
 System.out.println();
}
```

Note that the test method assumes that the *sweep()* methods are implemented as static methods in the class *Tools* in the class library *palib*. The above method *sweep()* can not be used for so much, but below is shown a variant that returns a value:

```
public static <T,R> R sweepValue(Iterable<T> values, Predicate<T> ok,
 BiFunction<T,R,R> action)
{
 R value = null;
 for (T t : values) if (ok.test(t)) value = action.apply(t, value);
 return value;
}
```

The last parameter is this time a function that is applied to an element in the current collection *values* and another *value*, which is the return value. However, it gives a simple challenge, since *value* the first time is *null*, and that is something that function *action* has to handle. Below is a test method that uses the algorithm to determine the smallest number in a list and to determine the sum of the list's elements:

```
private static void test10()
{
 List<Integer> list = Arrays.asList
  (16, 19, 13, 17, 5, 6, 7, 11, 9, 20, 8, 12, 3, 14, 15, 1, 4, 18, 2, 10);
 Integer t1 = Tools.sweepValue(list, e -> true,
  (a,b) -> { return b == null ? a : a < b ? a : b; });
 System.out.println(t1);
 Integer t2 = Tools.sweepValue(list, e -> true,
  (a,b) -> { return b == null ? a : b + a; });
 System.out.println(t2);
}
```

You should especially note how the action methods take into account that the parameter *b* the first time is *null*.

As a final example, below is shown a sweep algorithm that returns a sub collection of a collection where the returned elements are optionally modified:

```
public static <T,R> List<R> sweepList(Iterable<T> values, Predicate<T> ok,
 Function<T,R> action)
{
 List<R> list = new ArrayList();
```

```
 for (T t : values) if (ok.test(t)) list.add(action.apply(t));
 return list;
}
```

The following test method uses the algorithm to first determine all prime numbers in a list and then determine the cross sum of all the numbers in a list:

```
private static void test11()
{
 List<Integer> list = Arrays.asList
  (16, 19, 13, 17, 5, 6, 7, 11, 9, 20, 8, 12, 3, 14, 15, 1, 4, 18, 2, 10);
 List<Integer> list1 = Tools.sweepList(list, e
-> Functions.isPrime(e), e -> e);
 Tools.sweep(list1, t -> System.out.print(t + " "));
 System.out.println();
 List<Integer> list2 = Tools.sweepList(list, e -> true,
  e -> Functions.crossSum(e));
 Tools.sweep(list2, t -> System.out.print(t + " "));
 System.out.println();
}
```

EXERCISE 3: SWEEP A LIST

Write a program that you can call *SweepProgram*. The program must have a method

```
private static List<Integer> createList()
{
}
```

which creates a list of all 2-digit integers when the list must contain the numbers in random order. In the *main()* method, you must use the method to create a list, and for this list, use the sweep methods to perform the following operations:

1. print the list on the screen
2. create and print a sub list with all numbers between 30 and 50
3. print the list's mean value (the value is 54.5)
4. print the list's theoretical standard deviation (the value is 25.979158313283875)

3 ALGORITHM ANALYSIS

Once you have written an algorithm, there are two outstanding issues

1. to argue that the algorithm is correct and solve the current task
2. to evaluate the efficiency of the algorithm with regard to resource consumption

Both can be done more or less formally. For example, with regard to the correctness of algorithms, mathematical evidence can often be used in the form of a proof by induction, but in other contexts, a more informal argumentation can explain the correctness of the algorithm. The same goes for analyzes of the algorithms efficiency. In the following, I will show both approaches, but I would primarily focus on informal reasoning because it provides a practical approach to the problems and better is a tool that can be used in the daily but also because a formal mathematical argumentation easily can become complex. However, it should be noted that there is a significant risk that the informal reasoning leads to incorrect results.

An analysis of the complexity of an algorithm relates to the algorithm's consumption of the machine's resources, and generally the goal is to minimizing resource consumption. It's

basically about CPU time and memory, which are two resources that are always limited. In principle, you can equip the machine with a faster CPU and increase the machine's memory, but whatever can not solve the problem of an inefficient algorithm. However, modern machines today have so much memory that too little memory is rarely a problem (at least as long as you talk about a regular PC), and often you perceive the CPU as a more restrictive resource. As for algorithms, the focus is primarily on writing algorithms that utilize the CPU as well as possible, that is the same that the algorithm and the program being executed as quickly as possible. It is said that the algorithm must have optimal *time complexity*.

If you have written a program you can of course measure its time complexity with a clock and thus find out how long it takes to perform an algorithm and it certainly has interest in practice and if nothing else so in connection with test. However, one should always be aware that such a timing depends on the machine that performs the algorithm and in many contexts it is more interesting to assess the overall nature of the algorithm and associate a functional expression with an algorithm that may in one way or another characterize its efficiency – among other things, because you to a much greater degree are able to compare algorithms with each other.

3.1 THE COMPLEXITY OF LINEAR SEARCH

To show what algorithm analysis is about, I will look at linear search, as an algorithm discussed above. At best, the element that is searched for is found at the first comparison. If so, the execution time – also called its time complexity – is independent of the size of the array. It is said that the algorithm has a best time complexity that is constant. The best time complexity does not depend on which array is searched and is the same every time. One writes that the best time complexity is *O(1)*, that is read "*big o of 1*".

At the worst case, you must compare with all elements in the array, and if the array has *N* elements, you must perform *N* operations where an operation consists in comparing the element that is searched for with the element in a particular place in the array. In this case, the execution time depends on the array size, so that the execution time grows as a linear function (and hence the algorithm's name). It is said that the worst time complexity depends linearly on the array size, which is indicated as *O(N)* – read "*big o of N*". It expresses that the worst execution time behaves as a linear function of the array's length.

If you do not know what elements the array contains, you should expect to search halfway through the array – you can not know if the element you are looking for is to be searched at the beginning or end – and if you have an array with *N* elements, you must expect to perform *½N* operations. It is still a linear function of the array size (just with a smaller increase), and the algorithm therefore also has an average time complexity that is *O(N)*.

The above is an analysis of the algorithm's time complexity, which has resulted in three complexities:

- best time complexity: *O(1)*
- average time complexity: *O(N)*
- worst time complexity: *O(N)*

It is important to note that complexity measures do not specify specific numbers regarding the execution time of an algorithm and how long it takes to perform a corresponding program but only tells you how an algorithm behaves as a function of one or more input parameters – here the array size. Complexity measures can be used to compare algorithms and to determine if one algorithm for solving a problem is better than another.

The execution time of an algorithm naturally depends on other factors than what appears from the above analysis. For example, time is used to transfer parameters and create variables, but all this is something that has a constant amount of time and does not depend on the size of the array. It does not change the overall principle of how the algorithm behaves for different sizes of input parameters. There are also environmental conditions that affect the execution time and are determined by the computer that performs the algorithm. Here, for example, there are CPU, bus and memory chips, but it's all very specific relationships that do not have anything to do with the current algorithm, and not relationships that can be used to compare algorithms.

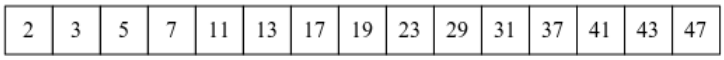
Specifically, the above analysis says that it is, not surprisingly, more costly to search in a large array than a smaller, and the analysis also says that the price grows as a linear function.

In the analysis, I have determined three complexity measures, which are often associated with an analysis of an algorithm's time complexity. The three measures can all be different, or they may all be the same, but often you are only interested in the average complexity, and you often only state one measure of an algorithm's time complexity, and that's the average complexity.

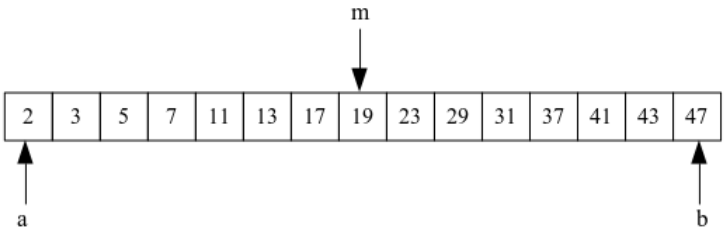
3.2 THE COMPLEXITY OF BINARY SEARCH

An algorithm is a solution to a problem, and there are often more algorithms that solve the same problem. If you look at the above problem to search an element in an array and if you do not know in advance what elements the array contains, you can not do much more than apply linear search, but if, for example, you know that the array is already sorted, you can do better with binary search.

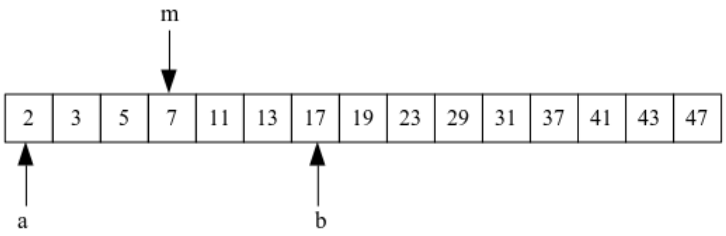
Suppose you have the following array:



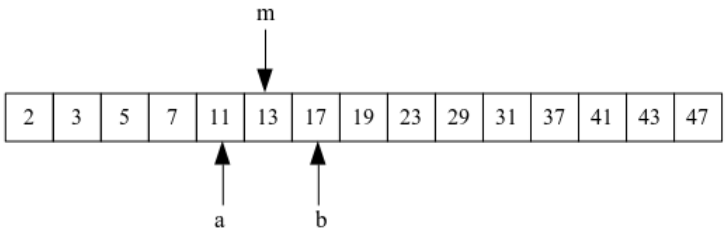
there is a sorted array and suppose you will search for the number 13. Instead of starting from the beginning, the idea is to start with the middle number:



That is, starting by comparing with the number 19. Since the array is sorted, and since 13 is less than 19, you can know that if 13 exists, it must be in the left half of the array. You can thus halve the number of elements to be searched. Next comparison should then be the middle number in the left half:



and the result is that you must compare with the number 7. Since 13 is greater than 7, only numbers to the right of 7 and up to the index *b* determined by the first half should be searched. In the next comparison, the number is found:



Note that the number was found after three comparisons. For linear search, 6 comparisons would be needed, a difference that would be even clearer if you had searched for example the number 41. Also note that the principle is that for each repeat, you halve the range of numbers to be searched.

In the same way as for linear search, the best time complexity of the algorithm is *O(1)*, since you can naturally find that the element you search already in the first comparison.

If you look at the worst time complexity, it will appear if the element searched is not found. In that case, the algorithm stops when the loop is run to end. This happens when the length of the interval to be searched is less than 1, so to determine the time complexity, you must determine when the length of the interval becomes 0. If the array has *N* elements, you must first search among

$$N = \frac{N}{2^0} \text{ elements}$$

At the next repeat, the interval is halved (actually the half minus 1, but in order to facilitate the calculations I will count on a halve since, 1 less in interval length does not make the big difference) so at the second comparison you must search among

$$\frac{N}{2} = \frac{N}{2^1} \text{ elements}$$

At the third comparison, the interval is halved again, so that you now have to searched among

$$\frac{N}{2}\Big/2 = \frac{N}{4} = \frac{N}{2^2} \text{ elements}$$

Then you proceed so that for each repeat the length of the interval is halved, and by that *k*th comparison, the interval length is

$$\frac{N}{2^{k-1}}$$

The loop therefore stops when

$$\frac{N}{2^{k-1}} < 1 \Rightarrow N < 2^{k-1} \Rightarrow \log(N) < \log(2^{k-1}) \Rightarrow \log(N) < k-1 \Rightarrow \log(N)+1 < k$$

where the logarithm function is the 2-number logarithm. That is, in worst case, the algorithm stops after *log(N)* comparisons, and the worst time complexity for binary search is therefore *O(log (N))*.

The mean complexity is somewhat more difficult to calculate and includes summation of a number of weighted averages, but it may appear to be *O(log (N))* and thus the same as the worst complexity. The result for binary search is as follows:

- best time complexity: *O(1)*
- average time complexity: *O(log(N))*
- worst time complexity: *O(log(N))*

The question is then what it means in relation to linear search, and to get a feel, you can consider the following table:

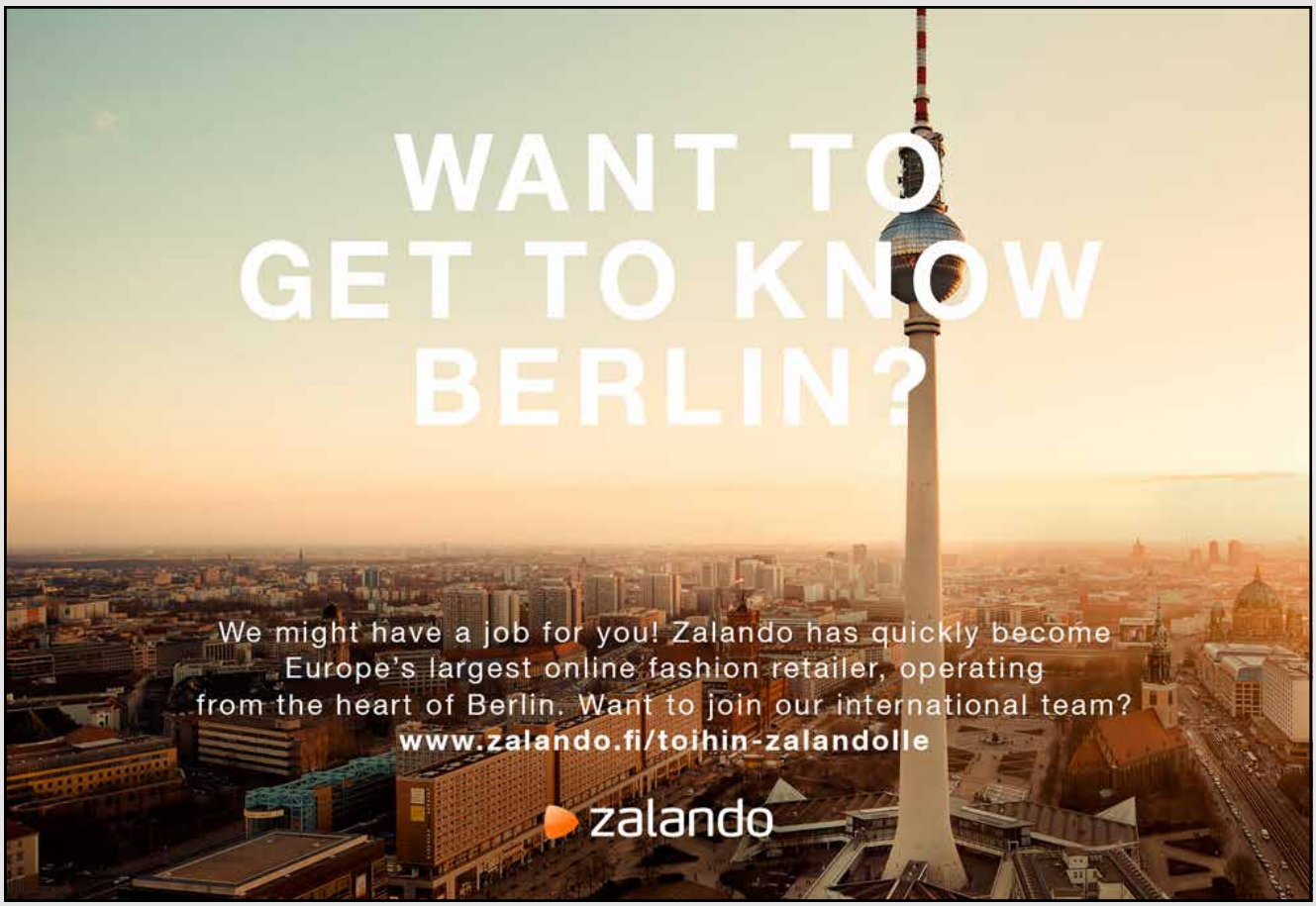
N	log(N)
64	6
256	8
1024	10
32768	15
1048576	20
33554432	25
4294967296	32
18446744073709551616	64

It clearly illustrates that binary search is not only faster than linear search, but for large arrays the speed difference is extreme.

Compared to linear search, it requires a little more details to implement binary search in Java, primarily because the comparison should be able to determine whether to search in the left or right half.

If you look at the table above, the difference is great, but in a concrete implementation there are other factors that matter. Looking at linear search, each repeat of the loop consists of a single comparison. In binary search, there is more in each repetition, which is both a calculation and two comparisons, and the last comparison even includes a call of the method *compareTo()*. The table expresses the difference in the number of operations, but the binary search operations are somewhat more complex than in linear search. Nevertheless, the difference corresponding to the table is a guideline and more than that, as it is something you can measure.

In the rest of this book, I will often measure how long it takes to perform an algorithm and I have therefore added the following class to the project *palib*:



```
package algoritms;

public class StopWatch
{
 private boolean running = false;
 private long start = 0;
 private long stop = 0;

 public boolean isRunning()
 {
  return running;
 }

 public void start()
 {
  if (!running)
  {
   start = System.nanoTime();
   running = true;
  }
 }

 public void stop()
 {
  if (running)
  {
   stop = System.nanoTime();
   running = false;
  }
 }

 public long getNanoSeconds()
 {
  if (isRunning()) return System.nanoTime() - start;
  return stop - start;
 }

 public int getMicroSeconds()
 {
  return (int) (getNanoSeconds() / 1000.0);
 }

 public int getMilliSeconds()
 {
  return (int) (getNanoSeconds() / 1000000.0);
 }
```

```
 public int getSeconds()
 {
  return (int) (getNanoSeconds() / 1000000000.0);
 }
}
```

The class represents a very simple stopwatch, which measures in nanoseconds. The class only have a method *start()* and a method *stop()*, as well as methods that can return the time between calls of *start()* and *stop()* (or from start to current time if the clock is not stopped). Consider as example the following method:

```
private static void test12()
{
 StopWatch sw = new StopWatch();
 sw.start();
 Integer[] table = Tools.createArray(200000000);
 for (int i = 0; i < table.length; ++i) table[i] = i;
 sw.stop();
 System.out.printf("%d\n", sw.getMicroSeconds());
 int elem = Integer.MAX_VALUE;
 sw.start();
 int n1 = Tools.linSearch(table, elem);
 sw.stop();
 System.out.printf("%d\n", sw.getMicroSeconds());
 sw.start();
 int n2 = Tools.binSearch(table, elem);
 sw.stop();
 System.out.printf("%d\n", sw.getMicroSeconds());
}
```

The method measures how long it takes to create a sorted array of 200 million elements, and then how long it takes to search for an element (which is undoubtedly not found in the array) using linear and binary search. Performing the method on my machine is the result:

```
42305691
241953
25
```

where time is indicated in micro seconds. From this you can read that it has taken about 42 seconds to create and initialize the array, approximate 250 milliseconds to use linear search and 25 microseconds to use binary search. Binary search is thus much faster than linear search. You could improve the test method by repeating the test several times (here 100 times) and searching for random numbers that is found in the array:

```
private static void test13()
{
 StopWatch sw = new StopWatch();
 Integer[] table = Tools.createArray(200000000);
 for (int i = 0; i < table.length; ++i) table[i] = i;
 for (int i = 0; i < 100; ++i)
 {
  int elem = rand.nextInt(table.length);
  sw.start();
  int n1 = Tools.linSearch(table, elem);
  sw.stop();
  long t1 = sw.getMicroSeconds();
  sw.start();
  int n2 = Tools.binSearch(table, elem);
  sw.stop();
  long t2 = sw.getMicroSeconds();
  System.out.printf("%d [%d %d] [%d %d]\n", elem, t1, n1, t2, n2);
 }
}
```

Also note the method *createArray()*, which is a method added to the library class *Tools*, which creates a generic array.

SIMPLY CLEVER

ŠKODA

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

The result of test methods of this kind will of course vary depending on which machine they are performed, but it is nevertheless a test that can be used to check the effectiveness of an algorithm and, in particular, such a test can be used to verify the result of an algorithm analysis.

3.3 MORE EXAMPLES

In this section I will show more examples of algorithms and their complexities. The goal is to show how you using simple arguments can derive an algorithm's time complexity without using advanced mathematical calculations.

I will start with an example that determines the sum of the smallest and largest numbers in an array. That is, given an array *table* with *N* numbers, you must determine the sum of the largest and the smallest number. This can be done as follows:

```
min = max = table[0]
for (i = 1 to N-1)
{
 if (table[i] < min) min = tabel[i]
 else if (table[i] > max) max = table[i]
 i = i + 1
}
return min + max
```

There is not so much mystery in the algorithm, and it is easy to figure out that it gives the right result. This is an example of a sweep algorithm, and in Java, for example, the algorithm can be written as follows:

```
private static int minMaxSum(int[] arr)
{
 int min = Integer.MAX_VALUE;
 int max = Integer.MIN_VALUE;
 for (int i = 0; i < arr.length; ++i)
 {
  if (min > arr[i]) min = arr[i];
  if (max < arr[i]) max = arr[i];
 }
 return min + max;
}
```

If you want to analyze the execution time, you can find the algorithm's operations and enter designations for their execution times:

t0 time consumption to call a method (parameter transfer, etc.)
t1 time consumption to initialize a local variable
t2 time consumption for a comparison
t3 time consumption for an assignment
t4 time consumption for an addition
t5 time consumption for a return

(and you could probably also point out other things that takes time). As the loop is repeated *N* times, the execution time of the algorithms can be determined as:

```
T = t0 + 3*t1 + N*(3*t2 + 2*t3 + t4) + t4 + t4
```

If you here defines

```
a = 3*t2 + 2*t3 + t4
b = t0 + 3*t1 + t4 + t5
```

the execution time is

```
T = a*N + b
```

and thus the algorithm's time complexity follows a linear function.

When looking at time complexities, they are only interesting in connection with "big" problems, and in this case, that is for large values of *N*. In minor problems all of the above measurements play a role, but if *N* is large, the constant *b* in the above function expression are not very important and you can ignore it. Thus, the time complexity of the algorithm will for large values of *N* follow the linear function

```
T = a*N
```

The constant *a* is the coefficient of the functions slope, and it tells how fast the linear function grows, but it does not change the basic characteristics of the function. When you talk about complexities, you therefore generally choose to ignore constants, and solely considering the general behavior of the complexity for great values of the parameters of the problem. Thus, in this case, the time complexity of the algorithm is *O(N)*.

Generally, you will not argue as above to determine the time complexity of an algorithm. In this case, it will be said that the algorithm consists primarily of a loop repeated *N* times and each repetition consists of a single (well-compounded) operation. The complexity is thus *O(N)*. Note that it applies to both the best, the medium and the worst complexity – every time you go through the loop.

Sortering

As the next example, I will look at sorting an array, and more precisely, I will look at the sorting of the elements in an array of *N* elements in ascending order. It is a very classic problem and there are many algorithms to solve the problem. In the next chapter, I will especially look at sorting algorithms, while I here just want to show an example called *bubble sort*. The algorithm is very simple. You pass through the array and compare the elements in pairs (each element with its successor). If you meet two elements that are incorrectly arranged – one element is larger than its successor – then you swap them. This process is repeated until you have completed an iteration without interchange. At that time, the array is sorted – then there are no element that are larger than the successor.

The algorithm can be written as follows:

```
do
{
 for (i = 0 to N-1)
 {
  if (table[i] < table[i + 1]) swap table[i] og table[i + 1]
 }
}
while (there was a swap)
```

Turning a challenge into a learning curve.
Just another day at the office for a high performer.

Accenture Boot Camp – your toughest test yet

Choose Accenture for a career where the variety of opportunities and challenges allows you to make a difference every day. A place where you can develop your potential and grow professionally, working alongside talented colleagues. The only place where you can learn from our unrivalled experience, while helping our global clients achieve high performance. If this is your idea of a typical working day, then Accenture is the place to be.

It all starts at Boot Camp. It's 48 hours that will stimulate your mind and enhance your career prospects. You'll spend time with other students, top Accenture Consultants and special guests. An inspirational two days packed with intellectual challenges and activities designed to let you discover what it really means to be a high performer in business. We can't tell you everything about Boot Camp, but expect a fast-paced, exhilarating and intense learning experience. It could be your toughest test yet, which is exactly what will make it your biggest opportunity.

Find out more and apply online.

Visit accenture.com/bootcamp
• Consulting • Technology • Outsourcing

accenture
High performance. Delivered.

The algorithm consists of two loops inside each other. The length of the inner loop is always *N-1*, and it thus performs *N-1* operations. The question is how many times the outermost loop is performed.

If the array has already been sorted, the inner loop does not change, and the outer loop is only executed once. The algorithm will therefore perform *N-1 + 1 = N* operations, and therefore it has the best time complexity, which is *O(N)*, that is, a linear time complexity.

When the inner loop is passed, the elements are compared in pairs, and in a comparison where the left-hand element is larger than the right, the largest moves a place to the right, and the smallest of the two elements a place to the left. The smallest element will therefore move to the left for each pass unless it is in the first place. It will thus be in place after no more than *N-1* iterations. The next smallest element also moves a place to the left unless it already is in place or the smallest element is to the left of it, but it only happens once, so after N-1 iterations is also the next smallest element in place. It also applies to the third least, which also moves a place to the left unless it is in place, or the smallest or second least is to the left of it, but it can only happens twice, and the element will then highest move N-2 places, and the element will be in place after no more than N-1 iterations of the inner loop. This argument holds for all array elements. As it requires an additional pass to find out that no exchanges have been made, the algorithm will in the worst case perform

$$N(N-1)+1 = N^2 - N + 1$$

operations. For large values of *N* you can ignore the constant 1, and the same applies to the part *N*, because N^2 for large values of *N* are much greater than *N*. The worst time complexity is thus

$$O(N^2)$$

The worst complexity occurs when the smallest elements from the start are to the right – many comparisons must be made before the elements move into place. In the mean, one must expect that less than N pass through the inner loop before the elements fall into place. For example, assuming that only *½N* iterations are necessary, the required number of operations are:

$$½N(N-1)+1 = ½N^2 - ½N + 1$$

When looking at the time complexity of an algorithm, as mentioned above, you only interest what happens to the algorithm for a large input – and here a large value of *N*. If *N* is large,

N^2 is much bigger and so big that you can ignore all parts that depends alone on N and constant parts, and you can therefore estimate the number of operations like

$$½N^2$$

Just as for the linear function, the constant just tells how fast the function grows, but it does not change the general characteristics of the function, here a parable. As in the case of complexity calculations, you will always ignore constants, and the result is that the average time complexity of the algorithm is

$$O(N^2)$$

There is nothing wrong with $O(N^2)$ algorithms, but the function N^2 grows fast and for large values of N, such algorithms will be so slow that they are often unusable.

The algorithm can be implemented in Java as follows:

```
public static <T extends Comparable<T>> void bubbleSort(T[] t)
{
 boolean ok;
 do
 {
  ok = true;
  for (int i = 0; i < t.length - 1; ++i)
   if (t[i].compareTo(t[i + 1]) > 0)
   {
    T e = t[i];
    t[i] = t[i + 1];
    t[i + 1] = e;
    ok = false;
   }
 }
 while (!ok);
}
```

It is not very strange because it is basically a translation of the above algorithm to Java. Note, however, that the parameter type must be comparable, as well as how to keep track of whether there have been exchanges at the inner loop.

Below is a test method (which uses the sweep algorithm from the previous chapter):

```
private static void test14()
{
 Integer[] arr =
  { 16, 19, 13, 17, 5, 6, 7, 11, 9, 20, 8, 12, 3, 14, 15, 1, 4, 18, 2, 10 };
 Tools.sweep(Arrays.asList(arr), t -> System.out.print(t + " "));
 System.out.println();
 bubbleSort(arr);
 Tools.sweep(Arrays.asList(arr), t -> System.out.print(t + " "));
 System.out.println();
}
```

To test the complexity, try the following method, that prints how many seconds it has taken to sort an array with random integers:

```
private static void test15()
{
 int N = 1000;
 StopWatch sw = new StopWatch();
 Integer[] table = Tools.createArray(N);
 for (int i = 0; i < table.length; ++i) table[i] = rand.nextInt(N);
```

```
 sw.start();
 bubbleSort(table);
 sw.stop();
 System.out.println(sw.getSeconds());
}
```

If you try the method, you will probably be told that it takes 0 seconds. Then try to make *N* larger, and you will find that eventually the algorithm takes a very long time to finally stop completely. The lesson is that for small values of *N*, the algorithm is quite effective, but for large values it is unusable.

3.4 DEFINITION OF COMPLEXITIES

In this section, I would like to define the concept of complexity a little more formally.

Let an algorithm be given where the execution time depends on a positive integer *N* and so that the execution time can be described by a function *T(N)*. If also *F(N)* is a function of *N*, it is said that the algorithm's time complexity is *O (F(N))* if there is a constant *c* and a positive integer so

$$T(N) \leq cF(N) \quad \forall N \geq N_0$$

It expresses that for large values of *N*, the execution time is not worse than a constant times *F(N)*. Saying a bit differently, the function *F(N)* defines an upper limit for how wrong it may be.

It should be added that there are other complexity measures than "*big o*", but I will be happy with that, as it is primarily the one you encounter in practice.

There are no specific requirements for the function *F(N)*, but in practice it will typically be one of the following functions, where the logarithm function (usually) is the 2-number logarithm and the sequence describing how fast the functions grows for large values of *N*:

- 1 Constant whose graph is a horizontal line
- $\log(N)$ The logarithm function whose graph is flat curve
- $\log^2(N)$ The square of the logarithm function
- $\sqrt{N}$ The square root function
- N A linear function, whose graph is a growing straight line
- $N\log(N)$ *N* times the logarithm function whose graph grows much smaller than N^2
- N^2 Square function whose graph is a parable
- N^3 The cubic function whose graph is a very fast growing function
- 2^N The eksponential function

The above functions are typical, but there may be others.

Above I have everywhere described complexities of functions in 1 variable, but an algorithm and its time complexity can of course depend on several parameters. It does not significantly change the definitions, but just think that N can consist of several values.

In practice, it can be quite difficult to deduce and argue for the complexity of an algorithm, and it will often require a lot of mathematics. However, there are some guidelines that can help.

1. If an algorithm does not contain loops, its execution time will usually be independent of the parameters of the algorithm and its complexity is constant. It is a $O(1)$ algorithm. You must be careful, however, because the algorithm can call a method that does not have constant time complexity. Algorithms with constant time complexity are considered to be optimal in many contexts.
2. If the algorithm contains a single loop and consists of a single pass of this loop, then its complexity will be $O(N)$. There are many algorithms that have this complexity and in many contexts they will be considered as good algorithms.
3. If an algorithm has two loops inside each other and if one loop is repeated N times and the other repeats M times, the complexity will typically be *O(MN)*. If specifically $M = N$ (what is often the case) the complexity will be $O(N^2)$.
4. Another common situation is (think of binary search) that you has a loop and for each pass of the loop the length of the loop is halved. If so, time complexity will be $O(\log(N))$.

You should be careful about the above rules as they can easily give a false picture, primarily because methods/algorithms call/use other methods that have a time complexity that is not constant, but conversely, the rules may be useful for quickly assessing a complexity of an algorithm. For example, think of an algorithm that has three nested loops, all of which are equal long. Such an algorithm will have the complexity $O(N^3)$, and in the vast majority of cases it will be a complexity that makes the algorithm unavailable for practical purposes.

3.5 MERGE

Given two arrays *table1* and *table2*, both of which are sorted in ascending order, one understands by *merge* the two arrays combined to an array that contains all elements in both *table1* and *table2* sorted in ascending order.

If *table1* has *M* elements and *table2* has *N* elements, you can write an algorithm as follows:

```
i = j = k = 0
while (i < M and j < N)
{
 if (table1[i] < M)
 {
  table3[k] = table1[i]
  ++i
 }
 else
 {
  table3[k] = table2[j]
  ++j
 }
 ++k
}
while (i < M)
{
 table3[k] = table1[i]
 ++i
 ++k
}
```

We have ambitions. Also for you.

SimCorp is a global leader in financial software. At SimCorp, you will be part of a large network of competent and skilled colleagues who all aspire to reach common goals with dedication and team spirit. We invest in our employees to ensure that you can meet your ambitions on a personal as well as on a professional level. SimCorp employs the best qualified people within economics, finance and IT, and the majority of our colleagues have a university or business degree within these fields.

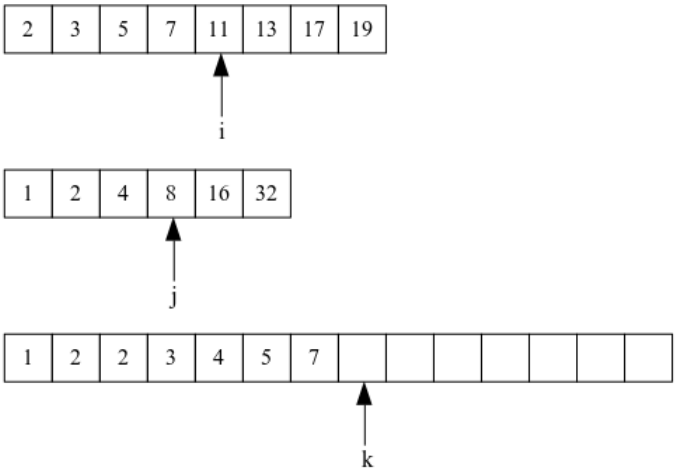
Ambitious? Look for opportunities at www.simcorp.com/careers

SimCorp

www.simcorp.com

```
while (j < N)
{
 table3[k] = table2[j]
 ++j
 ++k
}
return table3
```

The principle is that you have a pointer *i* pointing to the next element of *table1* that has to be moved to the result array, a pointer *j* pointing to the next element in *table2* to be moved to the result array and a pointer *k* as points to the place in the result array where the next element is to be placed. As long as the two pointers *i* and *j* point to something, you move the least of the two elements that the pointers point to. This continues until one of the two pointers does not point to anything – that is, all elements in one array are moved and they will in the result array be placed in ascending order. Then, the remaining elements of the two arrays must be moved, but there is only one of the two arrays where there are elements left, so only one of the last two loops is performing. A step in the algorithm can be illustrated as follows, where the next element to move is 8:



If you look at the complexity, the time consuming to move an element is the same, no matter what loop it is: A comparison, an assignment, and a ++ operations. A repeat of one of the three loops will always move exactly one element, and as all elements are moved, there are $M + N$ moves. That is that the complexity of the algorithm is $O(M + N)$.

The algorithm can be easily implemented in Java as follows:

```
 for (; i < t1.length && j < t2.length; ++k)
  if (t1[i].compareTo(t2[j]) < 0) t[k] = t1[i++]; else t[k] = t2[j++];
 for (; i < t1.length; ++k) t[k] = t1[i++];
 for (; j < t2.length; ++k) t[k] = t2[j++];
 return t;
}
```

which is nothing else than the algorithm translated to Java. The method is implemented in the class *Tools* in *palib*. Below is a test method:

```
private static void test16()
{
 Integer[] arr1 = { 2, 3, 5, 7, 11, 13, 17, 19 };
 Integer[] arr2 = { 1, 2, 4, 8, 16, 32 };
 Comparable[] arr3 = Tools.merge(arr1, arr2);
 Tools.sweep(Arrays.asList(arr3), t -> System.out.print(t + " "));
 System.out.println();
}
```

Here you should in particular note that because of type inference, the *merge()* method returns an array of the type *Comparable[]*.

3.6 TIME COMPLEXITY IN PRACTICE

Complexity calculations may seem theoretical, but they actually play a major role in practice. You often hear about programs where users complain that they are running slow. There may be several reasons, but the most frequent is actually unfortunate algorithms with poor time complexity. It is therefore important to be able to evaluate its algorithms with regard to the use of time and use time complexities as a measure of the algorithm's efficiency and as a tool for comparing algorithms and making the right choices.

In practice, it can be difficult (comprehensive) to determine the time complexity of algorithms, but you get well with the above informal guidelines and then a good deal of practice. Thus, it is a fixed part of writing an algorithm that you also relies on its time complexity.

As mentioned, it is not necessarily easy to determine a time complexity, and especially if you rely on informal arguments you can make mistakes. Therefore, there is also reasons to be interested in timing, which simply measures how much time an algorithm takes. Time measurements are especially well under test to validate if the theoretical time complexities that have been reached are now also sensible. In addition, time measurements can be important as documentation, where you can determine the length of time that has been addressed to solve the problem for different sizes of a problem.

EXERCISE 4: CALCULATING PRIMES

In the previous chapter, I looked at an algorithm to determine if an integer is a prime number. The algorithm basically consists of a loop, and if the algorithm is executed with a parameter N which is a prime number, you must go through the entire loop. That is, the loop should iterate $\sqrt{N}$ times, and therefore it has a complexity that is $O\left(\sqrt{(N)}\right)$.

Write a program that you can call *PrimesProgram* and copy the method *isPrime()* from the class *Functions* to the new project. You must then test if the above time complexity appears to be correct using the following algorithm:

```
p is a big positive odd number
n is a positive integer
loop i to n
{
 s = number of nano seconds to test where p is a prime
 if p is a prime
 {
```

```
  print p
  print s divided by the square root of p
  ++i
 }
 ++p
}
```

If that is the case, the value of the last quotient must be independent of the value of *p*. Below is a result where *p = 1000000000000001L* and *n = 10*:

```
1000000000000037 3,812190609270498
1000000000000091 3,853899533714737
1000000000000159 3,933701571081098
1000000000000187 3,898019347024227
1000000000000223 3,879385467797734
1000000000000241 3,859558745815084
1000000000000249 4,043631228548585
1000000000000259 3,843255212241957
1000000000000273 3,867859092217431
1000000000000279 3,849345442787636
```

3.7 MEMORY COMPLEXITY

Above, I have focused exclusively on the algorithm's time complexity, but algorithm's usage of memory obviously also plays a part. Here you use the same notation, and you can say that the algorithm's memory complexity is $O(N)$. This means that its space consumption increases linearly depending on an input parameter *N*, and a good example is where the algorithm creates an array or list. Similarly, if an algorithm has a memory complexity that is $O(\mathrm{MN})$, it could be because the algorithm creates a 2-dimensional array of *M* rows and *N* columns.

Memory complexity rarely has the same focus as time complexity, partly because the complexity is typically easier to understand, and partly because it is rarely as critical as time complexity, but conversely, the algorithm obviously does not need more memory than is necessary and, at worst, a too big consumption for memory can results in that the program going down.

As mentioned many times, you aim for an algorithm with a good time complexity, but often you will find that the price is a poor memory complexity. Although it is by no means a law, there is often a connection.

3.8 THE MEDIAN

Given an array

19 47 43 3 23 37 7 11 29 5 41 17 2 31 13

(as in this case has 15 elements), you understand by the median the middle element, and in this case it is element with index 7 in a corresponding sorted array. The median is therefore 19, since there are 7 numbers

3 7 11 5 17 2 13

which is less than 19 and 7 numbers greater than 19:

47 43 23 37 29 41 31

So long as the number of numbers is uneven, it's fine, but if the numbers is equal, a definition is required. Suppose there are 16 numbers instead

53 19 47 43 3 23 37 7 11 29 5 41 17 2 31 13

Then there are seven numbers that are less than 19 and 23, and there are seven numbers that are larger. There are therefore two options for the median. In practice, you will often choose the average, but when I look at arrays with integers, I want the result to be one of the numbers in the array and I therefore have to make a choice and if the number of elements is equal, I choose the least of the two options – in this case 19.

I will now look at an algorithm that can determine the median of an array with integers. In fact, it is not necessarily simple. However, there is a very direct solution where you simply start sorting the array:

```
public static <T extends Comparable<T>> T median(T[] arr)
{
 Arrays.sort(arr);
 return arr[(arr.length - 1) / 2];
}
```

The algorithm is based on sorting, and as will be seen in the next chapter, it is possible to write a sorting algorithm with an O(N log(N)) time complexity, and it also applies to the sorting method used by the framework. The above algorithm to determine the median therefore has a time complexity that is O(N log(N)). However, the algorithm has another

problem as it alters the array by sorting it. In many contexts, you are not interested in that and you can then solve the problem by letting the algorithm work on a copy:

```
public static <T extends Comparable<T>> T median(T[] arr)
{
 T[] tmp = Arrays.copyOf(arr, arr.length);
 Arrays.sort(tmp);
 return tmp[(tmp.length - 1) / 2];
}
```

It solves the problem, but now the algorithm has a memory complexity, which is $O(N)$.

You can also use a more direct route. Determining the median is actually a special case of a more general problem, which consists in determining the kth least number in an array, that is, a method that determines the least number, the second smallest number, the third smallest number and so on. If you have an array of 15 or 16 elements, and if you want to determine the median, you must determine the 7th smallest number in both cases. I want to start with a method that divides the elements of an array into two parts, where all elements on the left are less than a dividing point, while all elements to the right are greater than the dividing point. The result is a partition of an array where the lower index is a while the right index is b. The dividing point is determined by the index k:

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

```
public static <T extends Comparable<T>> int partition(T[] arr, int a, int
b, int k)
{
 T t = arr[k];
 swap(arr, k, b);
 int j = a;
 for (int i = a; i < b; ++i)
  if (arr[i].compareTo(t) < 0)
  {
   swap(arr, j, i);
   ++j;
  }
 swap(arr, b, j);
 return j;
}

private static <T> void swap(T[] arr, int i, int j)
{
 if (i != j)
 {
  T t = arr[i];
  arr[i] = arr[j];
  arr[j] = t;
 }
}
```

The algorithm is simple, but it works as follows:

1. The partition element is saved in a variable *t*, and it is then replaced with the value in the last place.
2. The index *j* is set to point after the end of the left part (that from the beginning is empty) – the elements that are smaller than the partition element.
3. Then the sub array is passed through. If you find an element that is smaller than the partition element, this element is added to the left part.
4. The last the partition element is put into place.

The partition element is also called a pivot point, and after the method is completed, the pivot point *t* will be located in the correct place, and all elements to the left of *t* are less than *t*, while all elements to the right of *t* are greater than (or equal to) *t*. Note that the method returns the index for the pivot point. Also note that it is a requirement that the index *k* falls within the range between *a* and *b*. Finally, note that the algorithm works by traversing the sub array and therefore it has linear complexity determined by the length of the sub array. You should note the method well, as I will apply it again in the next chapter about sorting.

When using the method, you are interested in dividing the array into two equal parts. It depends on how lucky you are with the choice of pivot point. Ideally, you should choose the median, but it's not easy to decide – that's the problem I'm in the process of solving. If you do not know anything about the array (the elements are arranged randomly), there is no reason to believe that one pivot point is better than the other, and you can then choose the last one (or the first one). This corresponds to the fact that in the above method, the index *k* can be omitted, and in practice you often use that strategy. However, if, for example, the array is already sorted (or almost sorted), it is the worst possible choice (since the array is not significantly divided). A better strategy is to choose the median of the endpoints and the middle point of the sub array:

```
private static <T extends Comparable<T>> int pivot(T[] arr, int a, int b)
{
 int c = (a + b) / 2;
 return arr[a].compareTo(arr[b]) <= 0 ? arr[b].compareTo(arr[c]) <= 0 ? b :
  arr[a].compareTo(arr[c]) < 0 ? c : a : arr[a].compareTo(arr[c]) <= 0 ? a :
  arr[b].compareTo(arr[c]) < 0 ? c : b;
}
```

Of course, there is no guarantee that it will produce a better result, but studies shows that in the vast majority of cases it is a good strategy, especially because it solves the problem of an array that has already been sorted and although the above method *pivot()* seems complicated, it's just a simple comparison.

With the method *partition()*, it is easy to write an algorithm that determines the *k*th smallest element in an array:

```
public static <T extends Comparable<T>> T leastOf(T[] arr, int k)
{
 T[] tmp = Arrays.copyOf(arr, arr.length);
 int a = 0;
 int b = tmp.length - 1;
 while (a < b)
 {
  int i = pivot(tmp, a, b);
  int j = partition(tmp, a, b, i);
  int p = j - a + 1;
  if (p == k) return tmp[j];
  if (k < p) b = j - 1;
  else
  {
   k -= p;
   a = j + 1;
  }
```

```
 }
 return tmp[a];
}
```

partition() swaps to the elements in the array, and since the array should remain unchanged after the method is completed, the method starts by creating a copy. Consequently, it has linear memory complexity. The method works by working on a partition (from the start it's all). Then, a pivot index is selected using the method *pivot()*, and the array is divided by the method *partition()*. Next, the location *p* of the pivot point is determined and if it is *k* the pivot point is the solution. If *p* is greater than k, the element must be found in the left part of the interval, and nothing else than the right end point of the sub array should be moved. Otherwise, the left end point is moved, and *k* is reduced by *p* corresponding to searching the (k-p)th smallest number in the right sub array.

With this method available, the median can be determined as follows:

```
public static <T extends Comparable<T>> T median(T[] arr)
{
 return leastOf(arr, arr.length / 2 + arr.length % 2);
}
```

If you look at *leastOf()*, it will (at worst) iterates until the length of the sub-interval from a to b is 1. If you are lucky with the division (and the sub-interval splits equal every time, then the loop stops after $\log(n)$ times. In each iteration you must divide the array, that consists of a sweep over the sub-interval to be divided, and as the length of this interval each halves, it corresponds to the number of operations being somewhat likes

$$\sum_{k=0}^{M} \frac{n}{2^k} = n \sum_{k=0}^{M} (½)^k = n \frac{(½)^M - 1}{½ - 1} = 2n(1 - (½)^M) < 2n$$

where $M = \log(n)$. That is, the time complexity of *leastOf()* is better than linear. The above calculation is not accurate, but it implies the magnitude of time complexity, and it is matched by time measurements.

The class *Tools* is expanded by the two methods *leastOf()* and *median()*, and the class *Algorithms* has a test method *test17()*.

EXERCISE 5: MEASUREMENTS FOR THE MEDIAN

If an algorithm has linear time complexity, and if $T(n)$ denotes the execution time as a function of the size of the problem, so is $T(n) = \mathrm{cn}$, and then that

$$\frac{T(n)}{n}$$

with approximation must be constant. Try applying this observation to validate the claim of linear time complexity of the median algorithm by writing a program that determines the median of arrays of different lengths and initialized with random integers. The result could be as shown in the following table, where the first column is the size of the array, the last the execution time in nanoseconds, and the middle is the quotient between the last and first columns:

```
    100000  55,2552          5525521
    200000  12,7290          2545796
    400000  17,5783          7031308
    800000  19,1032         15282593
   1600000  24,6048         39367651
   3200000  21,4919         68774160
   6400000  37,3755        239203352
  12800000  40,8904        523397121
  25600000  18,1694        465137509
  51200000  47,7779       2446228111
 102400000  27,7632       2842947688
```

PROBLEM 1: MAX CONTIGUOUS SUM

A classic problem is finding the largest contiguous sum in an array that contains both positive and negative numbers. A little different, the task is to determine the sub array whose elements results in the largest sum. For example, if you consider the array

-2 11 -4 13 -5 2

then the numbers 11, -4 and 13 form the largest contiguous sum which is 20. The problem can be solved by the following algorithm:

```
a = 0
b = arrayets længde – 1
sum = 0
for (i = 0 to the last index)
{
 s = 0
 for (j = i to the last index)
 {
  s += arr[j]
  if (s > sum)
  {
   sum = s
   a = i
   b = j
  }
 }
}
[a; b] is the sub array
```

Write a program called *MaxSumProgram* that implements this algorithm:

```
public static int[] maxSum1(int[] arr)
{
 …
}
```

Write a simple test method that can test the above method when it must print the array and the sum.

Consider the algorithm's time complexity.

Try writing an algorithm

```
public static int[] maxSum2(int[] arr)
{
 …
}
```

which performs the same but with the difference that time complexity is linear.

Write a test method that creates an array of *n* elements with random integers between *-n* and *n*. The method should then apply the above *maxSum* methods and measure how long the algorithms take and the method should print two lines (one line for each of the two methods). Below is an example where the test method has been performed 6 times:

```
1000             763        26051        2
1000             763        26051        0
10000           2481       452614       33
10000           2481       452614        0
50000          37950     12073907      379
50000          37950     12073907        1
100000         10459     19154645     1514
100000         10459     19154645        0
500000        134352    305079305    37804
500000        134352    305079305        0
1000000       392783    762382310   151914
1000000       392783    762382310        0
```

The first column is the array size (number of elements). The next column is the number of elements in the sub arrays that are determined and the third column is the sum of the elements in the sub-interval. The last column indicates how long it took to perform the algorithms measured in milliseconds.

3.9 CORRECTNESS OF ALGORITHMS

Another and often more difficult problem than to argue for the complexity of algorithms is to verify the correctness of an algorithm. Of course, it is, equally important as it does not make sense to write algorithms that do not give the correct result. In the vast majority of cases, one will argue for the correctness of an algorithm by means of informal argumentation, in which you by speech (or writing) explains how the algorithm works. After the algorithm is written, you will test it and apply the test results as an argument for the correctness of the algorithm. It's not always easy, and it's especially difficult to get all cases with and not overlook boundaries, where the algorithm may not take into account to.

However, you can also attack the correctness of algorithms more formally, and simply give a mathematical proof of the correctness of the algorithm. The following is a very brief introduction to the subject – it's almost just a hint of what to do. Partly because it is a relatively theoretical subject, and partly because the practical application is not so great.

An algorithm is correct if its method part used with correct pre-conditions gives the desired result with correct post-conditions. In addition, an algorithm must always stop after a final number of steps. The latter can only be a problem if the algorithm contains an iteration and, in general, argumentation for the correctness of algorithms is only interesting if the algorithm contains iterations. In the examples that I've seen so far, it's fairly easy to figure out that the algorithms are correct – maybe apart from Euclid's algorithm. To more formally prove

that algorithms are correct, you can associate statements (also called *assertions*) to specific locations in the algorithm. They express statements that must be *true* at those places and can as such be used to argue for the correctness of an algorithm. Assertions can in principle occurs anywhere in the algorithm, but they are typically used in selections and iterations where they are most interested, and they are in this context called for *invariants*. Looking at the power raising algorithm, you can introduce the following *invariant*:

```
// Pre: n > 0 og p >= 0
// Post: returnerer n opløftet i p'te
potens(n, p)
{
 r = 1
 while (p > 0) // Inv: r*n^p = n^p'
 {
  r = r * n
  p = p - 1
 }
 return r
}
```

Here is

```
Inv: r*n^p = n^p'
```

an invariant of the iteration, which must be met immediately before the iteration is performed and immediately after the iteration is performed. *p'* means the value of the parameter *p* as it was before the algorithm started (that is the value transferred to the algorithm). It is said that the invariant is valid if it is *true* whenever the condition of the iteration is checked, that is, before and after each iteration. Assume for a moment that that is the case. When the iteration stops, $p = 0$ and the invariant say that

$$r = \mathrm{rn}^0 = n^{p\prime}$$

and since the algorithm returns *r*, it just means that the algorithm is correct. To prove that the algorithm is correct is thus reduced to show that the invariant is preserved for each iteration. Typically, it is shown by induction:

When the loop starts is $r = 1$ and $p = p'$, so the invariant is met from the beginning and after the first iteration, $r = n$ and $p = p'-1$ are so

$$r * n^p = n * n^{p\prime-1} = n^{p\prime}$$

and the invariant is thus met. That is, the start of the induction is correct.

Assume that the invariant holds, that is to say that $\mathrm{rn}^p = n^{p\prime}$. Let *r"* and *p"* be the new values of *r* and *p* after the next iteration. That is that *r"* = *rm* and *p"* = *p-1*. then

$$r''n^{p''} = \mathrm{rnn}^{p-1} = \mathrm{rn}^p = n^{p\prime}$$

where the last equals are derived from the induction assumption. Thus, it is shown that the invariant applies after the next iteration, and it can be concluded that the algorithm is correct.

If you look at the concept of invariant in connection with a loop, it is a statement that is *true* both before and after each iteration in the loop:

while(B) // I

As the loop is iterating as long as the condition *B* is true, then after the loop is completed, you can conclude that:

$$I \wedge \neg B$$

and that is the statement used to conclude the correctness of an algorithm. Above is the statement

$$p = 0 \wedge \mathrm{rn}^p = n^{p\prime}$$

and then $r = \mathrm{rn}^0 = n^{p\prime}$.

As another example, in Euclid's algorithm one can express an invariant as follows:

```
Pre: n > 0 og m > 0
Post: n = gcd(n,m)
euklid(m, n)
{
 while (m != n) // Inv: gcd(n,m) = gcd(n',m')
  if (n > m) n -= m else m -= n
 return n
}
```

Above, during the presentation of the algorithm, I have explained that the algorithm stops, and that the invariant holds. Therefore, when the algorithm stops, you have

$$n = m \wedge \gcd(n, m) = \gcd(n', m')$$

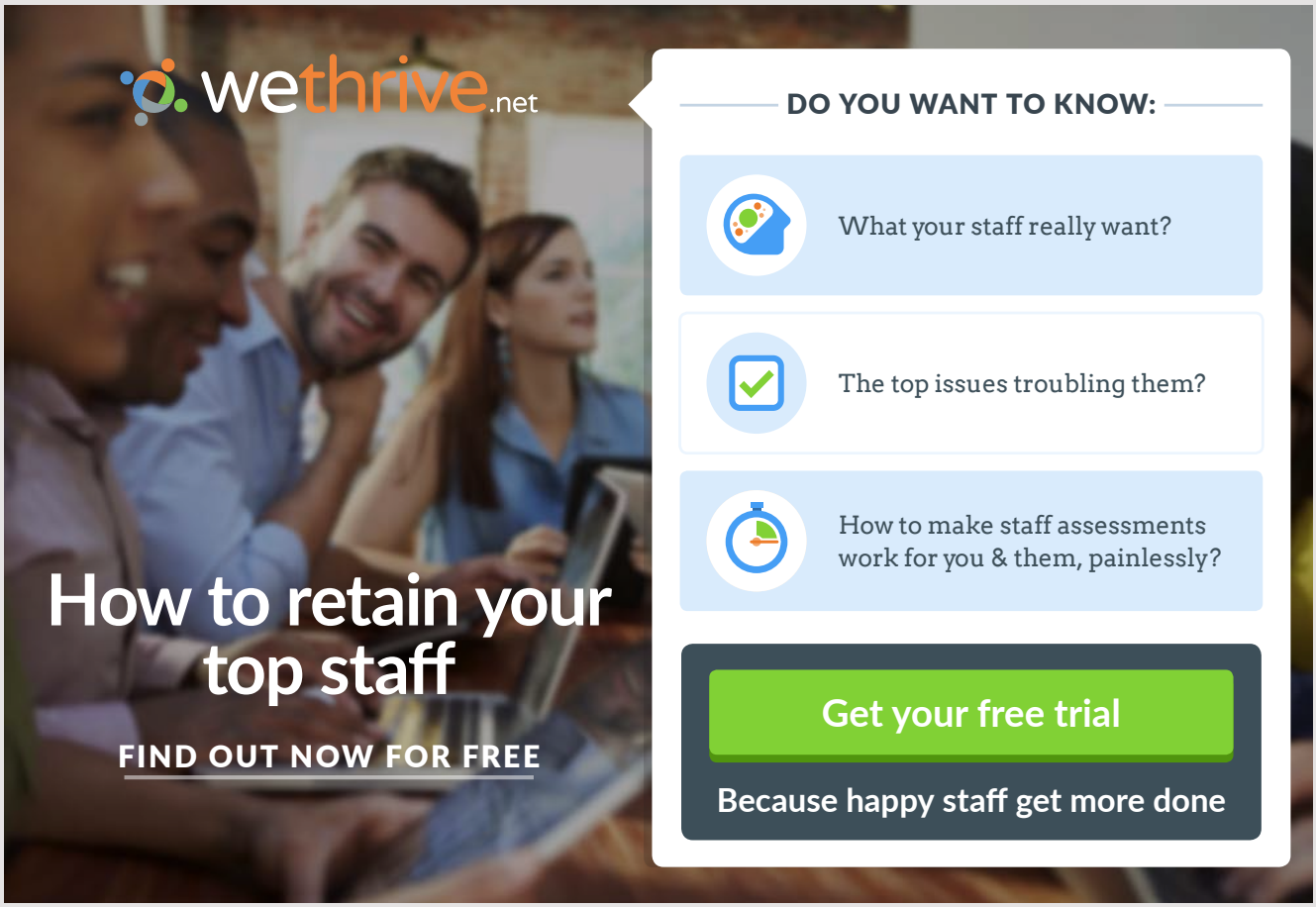
and then that the method returns $n = \gcd(n', m')$, which is the desired result.

4 SORTING

In this chapter I will mention a family of algorithms that are used to sort a collection of objects, and as an example of a collection I will use an array everywhere. I will look at the following algorithms:

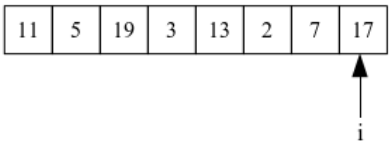
- Bubble sort $O(N^2)$
- Selection sort $O(N^2)$
- Insertion sort $O(N^2)$
- Shell sort $O(N(\log(N))^2)$
- Merge sort $O(N\log(N))$
- Quick sort $O(N\log(N))$

and as you can see from their time complexities, they differ a lot in terms of efficiency. In practice, the last two are the most widely used.

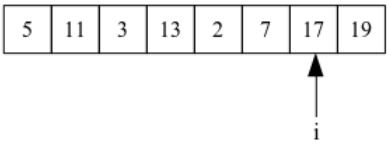


4.1 BUBBLE SORT

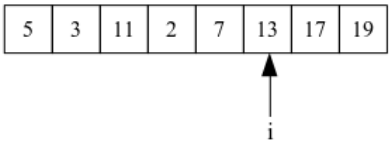
I've already mentioned this algorithm above, so the following is a bit of a repeat, but first a little bit about how bubble sort works (and can be implemented). If you have an array



then you run over the array from the back to the start. For each pass, you perform an inner loop that runs from start to the place where the arrow points. If you meet an item greater than its successor in this iteration, they are replaced. That is change 11 with 5, 19 change with 3, and then with 13, then with 2, 7 and finally with 17:



It is a step in the right direction, and 19 is now in place. Therefore, in the next iteration you should only look at the first 7 places and the result is after the second pass:



Now, the two major elements are with certainty in place and the process continues until the arrow reaches the last place – or until you have made a pass without exchanges – this is due to the fact that the array is sorted and the algorithm can stop. The algorithm is simple to implement and can be written in Java as follows:

```
public static <T extends Comparable<T>> void bubbleSort(T[] arr)
{
 boolean ok = false;
 for (int i = arr.length - 1; !ok && i >= 0; --i)
 {
  ok = true;
  for (int j = 0; j < i; ++j)
   if (arr[j].compareTo(arr[j + 1]) > 0)
   {
```

```
            swap(arr, j, j + 1);
            ok = false;
        }
    }
}
```

Note, first, that if the algorithm stops, an iteration has been made without exchanges. That is that you have not met two items that were wrong placed. Thus, the array must be sorted. The algorithm can stop in two ways. Either because an iteration has been made without exchanges or because the outer loop stops (and in the last pass there are no exchanges), but in any case, it stops and thus the algorithm is correct.

The next question is how effective the algorithm is. Assuming that the array is sorted from the start, there are no exchanges in the first pass and the algorithm stops. It provides a best time complexity that is *O(N)*. In the general case, an element incorrect relative to its successor will be moved one step to the right and repeat that until a major successor arrives. The largest element must therefore be in place after the first pass, after the next pass (which is 1 shorter), the second largest element will be in place and after all the other passes, all other elements will be replaced. The worst situation is where the smallest element from the start is at the last place. In this situation, the array requires *n* passes (the inner loop must be performed *n* times), which gives a complexity $O(n^2)$: The inner loop becomes one shorter each time (the first time there are *n-1* operations) and the number of operations is therefore:

$$(n-1)+(n-2)+\ldots+2+1=(n-1+1)\frac{n-1}{2}=½n^2-½n$$

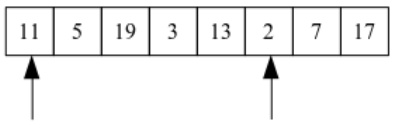
It corresponds exactly to a square time complexity. In the middle the least element must be moved *n/2* places that give the same complexity. The algorithm therefore has an average time complexity on $O(n^2)$. In return, the algorithm only uses memory for some simple variables, so its memory complexity is *O(1)*.

The algorithm is good if you have to sort a few elements and it is especially good if the array is almost sorted so that the smallest elements not are the last, but if it is a large array, it becomes slow and ultimately unusable. It is, on the other hand, simple to implement and, as mentioned, it does not use much memory.

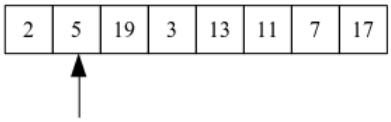
4.2 SELECTION SORT

Another sorting method is called selection sort. Here the array is traversed and the index is found on the smallest element. Once that has happened, the first element is replaced. The array is then traversed again, but only the last *n-1* elements are passed – the first one

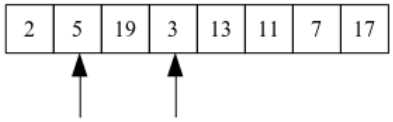
is put into place – and the index is found on the smallest element among the last *n-1* elements and it is replaced with the second element. Now the first two have been put in place. How to proceed. Each time you find the index on the smallest element that has not been put in place and it is placed. Each time the array to be passed becomes one element smaller. If the array is as above, you will find in the first pass the index for the number 2:



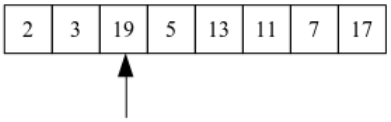
Then 11 and 2 are replaced:



and now 2 are in place. You then repeat the operation, but only on the last 7 elements, and you find the index for 3:



Then 5 and 3 must be replaced:



and the first two elements are in place. If you continue the process, you end after n-1 repeats with a sorted array. The algorithm can be written in Java as follows:

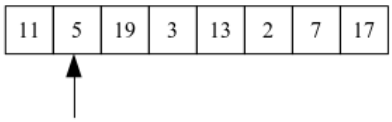
```
public static <T extends Comparable<T>> void selectionSort(T[] t)
{
 for (int i = 0; i < t.length - 1; ++i)
 {
  int k = i;
  for (int j = k + 1; j < t.length; ++j) if (t[k].compareTo(t[j]) > 0) k = j;
  if (k != i) swap(t, i, k);
 }
}
```

It is easy to see that the algorithm is correct, and because it consists of two nested loops, it is similar to bubble sort an $O(n^2)$ algorithm. The algorithm generally has the same advantages and disadvantages as bubble sort – both in terms of time and memory – and if I should mention an advantage, selection sort is, if possible, even easier to implement. If you make time measurements on the two algorithms, you will find that they behave in the same way (are both bad for large arrays), but bubble sort takes about twice as long as selection sort. The reason is that bubble sort tends to make too many exchanges.

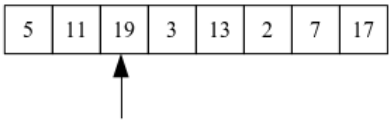
4.3 INSERTION SORT

A variant of the above is called *insertion sort*. Assume that the array is divided into a left part, which is sorted and a right part that is not sorted. You then take the first element in the right part and swap with the elements in the left part until the new element is in place – and the left part will still be sorted. Continue until you are finish. From the start

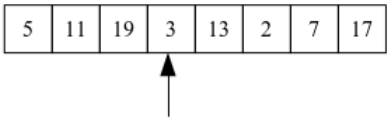
it is assumed that a left part, which consists only of an element, is always sorted. If the example is as above:



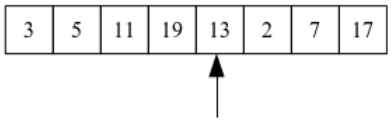
then you start with a left part, which contains only the element 11 and thus is sorted. You must put 5 in place in the left part:



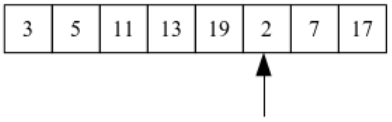
Now the left part consists of two elements, and the left part is sorted. As the next step, 19 must be put in place:



but it's simple for nothing should happen. Now 3 has to be put into place in the left array, and in return, it requires that all elements in the left array must be moved:



Next time, 13 will be in place, and it requires only a single exchange:



If you continue, you end up with a sorted array. The algorithm can be implemented as:

```
public static <T extends Comparable<T>> void insertionSort(T[] t)
{
 for (int i = 1; i < t.length; ++i)
 {
  boolean ok = false;
```

```
  for (int j = i; j > 0 && !ok; --j)
   if (t[j].compareTo(t[j - 1]) < 0) swap(t, j, j - 1); else ok = true;
 }
}
```

The algorithm is not much other than a variation of selection sort and it is clear that it has the same characteristics. However, insertion sort is fast for arrays that are almost sorted as an item falls into place after a few exchanges.

The above sorting algorithms primarily have theoretical interest and are good to get a grasp of how sorting is going on and how to write sorting algorithms. In the following, I will show three other sorting methods that are far more effective (and, on the other hand, much more complex) and thus have greater practical interest, but all sorting algorithms have a common nature that is about to loop over the array and all about to exchange the elements. The difference is primarily how clever you do it.

The above algorithms all have an average time complexity that is $O(n^2)$, and for large values of n (large arrays), the algorithms are too ineffective if not useless. However, you should not completely reject the algorithms in practice, as they as mentioned are simple to implement and are actually quite effective applied to small arrays.

4.4 SHELL SORT

The next algorithm is called shell sort. It is characterized by being far more effective than the above algorithms, having constant memory complexity, but also being more difficult both to implement and to review.

The idea is to divide the array into a number of sub-arrays called segments. Each of these segments is then sorted by one of the above methods, utilizing their effectiveness on small arrays. Once that is done, the process is repeated, but this time with fewer and larger segments, and here, utilizing that a method like insertion sort is effective if the array is already almost sorted. The process repeats until there is only one segment. The first problem is to select the number of segments that are determined by the following sequence:

$$a_k = \begin{cases} 0 & k = 0 \\ 2a_{k-1} + 1 & k > 0 \end{cases}$$

that is 1, 3, 7, 15, 31, 63, 127,…. Note that

$$a_k = 2a_{k-1} + 1 \Leftrightarrow a_{k-1} = ½(a_k - 1)$$

and thus it is easy to determine the number of segments for the next iteration. Also note that the sequence is essentially doubling, and thus it is a sequence that grows fast.

If there are *n* elements to be sorted, the number of segments is the largest value from the sequence that is less than or equal to *n*. For example, if there are *a* segments, the segments are defined as follows:

0th segment: t[0], t[a], t[2a], t[3a],….
1th segment: t[1], t[a+1], t[2a+1], t[3a+1], ….
……..
*i*th segment: t[i], t[a+i], t[2a+i], t[3a+i], ……
……
(*a-1*)th segment: t[a-1], t[2a-1], t[3a-1], t[4a-1], ….

If there are k elements in the *i*th segment, then

$$(k-1)a + 1 < n \Rightarrow k < \frac{n-1}{a} + 1$$

The above segments are sorted for example with insertion sort. In practice, it may be a bit complex, as the segment does not consist of continuous elements, so you need to control the indexing. In the next repeat, the number of segments is set to *(a – 1) / 2*, which is the previous item in the above sequence, and the repeat continues until the number of segments is 0.

Suppose that the following array is given:

41	11	29	3	37	2	7	17	31	5	19	43	13	23

Since there are $n = 14$ elements, there are 7 segments (the next number in the sequence is 15, which is greater than n). The contents of the segments are:

0. segment:	arr[0], arr[7]	41, 17
1. segment:	arr[1], arr[8]	11, 31
2. segment:	arr[2], arr[9]	29, 5
3. segment:	arr[3], arr[10]	3, 19
4. segment:	arr[4], arr[11]	37, 43
5. segment:	arr[5], arr[12]	2, 13
6. segment:	arr[6], arr[13]	7, 23

Now, each segment is sorted, which is easy, since each segment (in this case) has only 2 elements:

17	11	5	3	37	2	7	41	31	29	19	43	13	23

Note that there are actually only two places that have been replaced. Also note that large numbers move to the right while small numbers move to the left. Next time there will be 3 segments:

$$(7 - 1)/2) = 3$$

0. segment:	17, 3, 7, 29, 13
1. segment:	11, 37, 41, 19, 23
2. segment:	5, 2, 31, 43

Note that these segments are partially sorted. They are now sorted and the result is as follows:

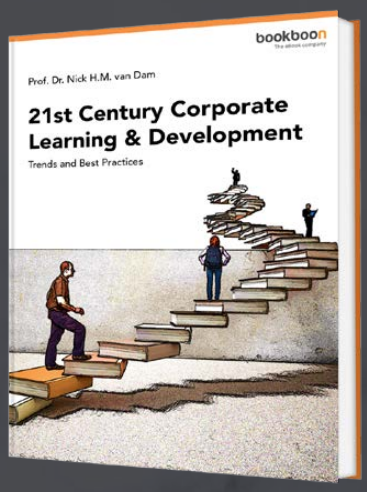
3	11	2	7	19	5	13	23	31	17	37	43	29	41

Next time there is only one segment and it is sorted:

2	3	5	7	11	13	17	19	23	29	31	37	41	43

The algorithm can be written in Java as follows:

```
public static <T extends Comparable<T>> void shellSort(T[] t)
{
 for (int a = segmentCount(t.length, 1); a > 0; a = (a - 1) / 2)
  segmentSort(t, a);
}

private static <T extends Comparable<T>> void segmentSort(T[] arr, int a)
{
 for (int k = 0; k < a; ++k)
  for (int i = a; i + k < arr.length; i += a)
   for (int j = k + i; j >= a && arr[j].compareTo(arr[j - a]) < 0; j -= a)
    swap(arr, j, j - a);
}

private static int segmentCount(int n, int a)
{
 int b = 2 * a + 1;
 return b > n ? a : segmentCount(n, b);
}
```

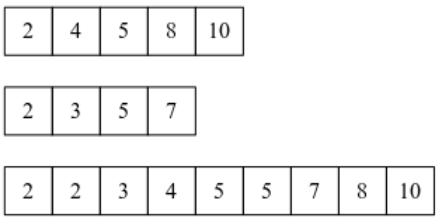
Free eBook on
Learning & Development
By the Chief Learning Officer of McKinsey
Download Now


First of all, note that the code does not fill up much, and that the method only uses simple helper variables and therefore has constant memory complexity. On the other hand, the code is not so easy to figure out. The help method *segmentCount()* is called from *shellSort()* and is used to determine how many segments to use. The method *segmentSort()* has an outer loop that iterates over the number of segments. Each iteration actually performs an insertion sort, and the hardest here is to control that a segment is not a continuous partition but is distributed over the array. You must therefore ensure that the correct places are addressed. Back there is the method *shellSort()* which repeats the execution of *segmentSort()* until the number of segments becomes 0.

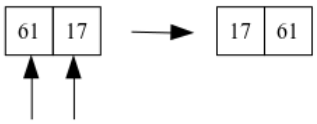
If you consider *segmentSort()*, it consists of three nested loops and it does not sound nice about the complexity, but it can be shown that the time complexity of shell sort with approximation is $O(n(\log(n))^2)$, which for big values of n are much better than $O(n^2)$. In fact, the effectiveness of shell sort depends on several things, including how to choose the sequence of segments, and not least how the array is already sorted. Shell sort is a very efficient sorting algorithm with a good memory complexity, and here it is the last thing that is the most interesting. The next two sorting algorithms are both faster than shell sort, but the price is an increased usage of memory, and in situations where memory is important, shell sort is a good option.

4.5 MERGE SORT

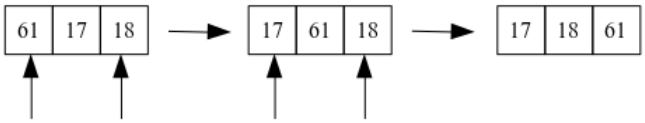
I have previously mentioned merging and how to merge two sorted arrays. Here you should note that if you merge two sorted arrays, the result is sorted array:



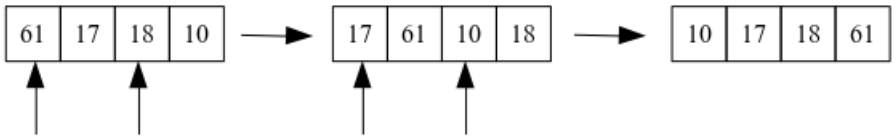
Merge can be used to sort an array. If you want to sort an array and it only has one element, the task is trivial. If there are two elements, you can perceive the array as consisting of two sub-arrays with each one element:



If you now merge these two arrays (the one consists of the number 61 and the other of the number 17), the result will be sorted. If instead there are three elements, you can consider the first two elements as a sub-array and sort it as shown above. Then you have two sub-arrays, one with 2 elements and one with 1 element, both of which are sorted. They can then be merged and the entire array is sorted:



If there are four elements, you start splitting into two sub-arrays with two elements in each. Each of these is sorted by merging. Then these sub-arrays rays are merged and the array is sorted:



This strategy is continued. An array is divided into sub-arrays, which are divided again until the length is 1, so that all sub-arrays are automatically sorted. After that, all sub-arrays are merged in pair until they are all reassembled to an array that is then sorted. This method is called *merge sort*.

An algorithm can be written as follows:

```
public static <T extends Comparable<T>> void mergeSort(T[] arr)
{
 mergeSort(arr, 0, arr.length - 1, createArray(arr.length));
}

private static <T extends Comparable<T>> void mergeSort(T[] t, int a, int
b, T[] v)
{
 if (a < b)
 {
  int m = (a + b) / 2;
  mergeSort(t, a, m, v);
  mergeSort(t, m + 1, b, v);
  merge(t, a, m, b, v);
 }
}
```

```
private static <T extends Comparable<T>> void
merge(T[] arr, int a, int b, int c,
 T[] tmp)
{
 int k = a; // index for the result array
 int i = a; // index for the left sun arrat
 int j = b + 1; // index for the right sub array
 while (i <= b && j <= c)
 {
  if (arr[i].compareTo(arr[j]) < 0)
  {
   tmp[k++] = arr[i];
   ++i;
  }
  else
  {
   tmp[k++] = arr[j];
   ++j;
  }
 }
 while (i <= b)
 {
```

```
  tmp[k++] = arr[i];
  ++i;
 }
 while (j <= c)
 {
  tmp[k++] = arr[j];
  ++j;
 }
 for (k = a, i = a; k <= c; ++k, ++i) arr[k] = tmp[i];
}
```

The algorithm has the same prototype as the other sorting methods, but works by calling a recursive version. As parameter, it has an additional array that is used to merge the result – it is sent as a parameter to the method *merge()* not having to create the array each time. The most complex is actually the method *merge()*, but in principle it is the same method that I have previously dealt with, and the new is to keep track of what it is for sub-arrays to be merge.

Each recursive call halves the length of the array and each time the entire array is to be passed. It gives a complexity that is $O(n\log(n))$ and is actually better than shell sort. On the other hand, the method requires a copy of the array to be sorted so its memory complexity is *O(n)* something that is a disadvantage of large arrays. That is that you pay for high efficiency with a large amount of space. In fact, Java (the classes *Arrays* and *Collections*) uses merge sort (exactly a variant called *TimSort*) as sorting algorithm.

4.6 QUICK SORT

Above I have looked at binary search. The idea is that you look at the middle element. This allows you to see if the element to be searched is in the left or right half. That is that you have halved the size of the problem. You continue this way and each time the size of the problem is halved. You are done when the element is found or the problem is trivial. Binary search is an example of a general problem solving technique, commonly called *divide and conquer*:

Given a problem *P* is the technique: *P* is divided into a number of sub-problems $P_1, P_2,..., P_k$, each of which is simpler than the original problem *P*. Then the sub-problems are solved and a number of sub-solutions are obtained. Finally, the partial solutions are combined to solve the original problem.

In connection with binary search, the problem *P* was to search an element *x* among *n* elements. This problem is divided into three sub-problems, each simpler than the original problem (it is easier to search an element among *n / 2* elements than among *n* elements):

1. Compare x with the middle element
2. Search *x* among element to the left of the middle element
3. Search *x* among element to the right of the middle element

Only one of these problems has a solution that is not empty, and since binary search relates to search in a sorted collection, one can decide which of the three sub-problems to proceed with: One solution is combining two empty solutions.

There are many examples of problems that are appropriate to solve that way and the method can be expressed as follows:

```
Pre: P is divisible
Post: S is the solution
divideAndConquer(P)
{
 if (P is simple)
  S = simple(P);
 else
 {
  divide(P, P1, P2, …, Pk);
  S1 = divideAndConquer(P1);
  S2 = divideAndConquer(P2);
  .....
  Sk = divideAndConquer(Pk);
  S = conquer(S1, S2, …, Sk);
 }
}
```

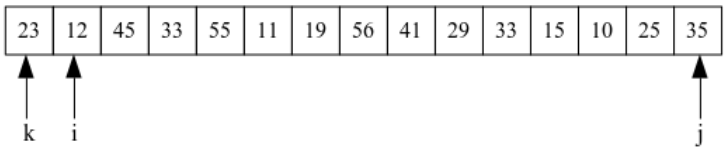
A prerequisite for the method's use is that the problem can be divided into sub-problems that are simpler than the original problem. *simple()* is an algorithm that solves a simple problem, that is a problem that it is not necessary to divide further. *divide()* divides the problem *P* into sub-problems, and *conquer()* is an algorithm that combines sub-solutions. Note that the algorithm is recursive.

When you implement a *divide and conquer* problem, you will typically use language options, which can make it difficult to recognize the above template. That is in conjunction with binary search:

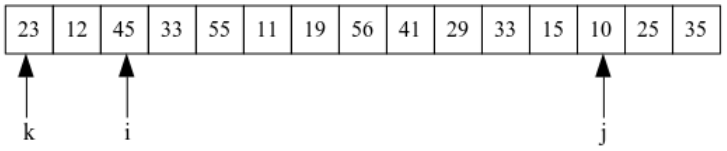
```
static boolean binSearch(int []tal, int a, int b, int x)
{
 // if the problem is simple, the solution is empty
 if (b < a) return false;
 // divide the problem
 int m = (a + b) / 2;
 // solve the first sub problem
 if (x == tal[m]) return true;
 // solve and conquer the two other problems
 return x < tal[m] ? binSearch(tal, a, m-1, x) : binSearch(tal, m+1, b, x);
}
```

Also merge sort is an example of an algorithm written by the *divide and conquer* template. The array to be sorted is divided into smaller arrays until the sorting problem becomes simple (when the arrays has only one element). Then, the sub-arrays are combined by merge them. The work lies in the merge, as the division into sub-arrays is just a matter of moving indexes appropriately.

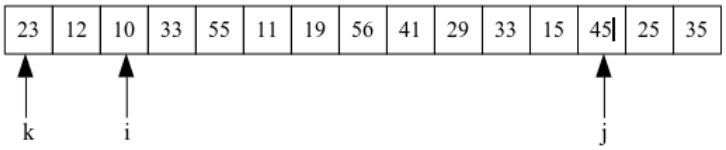
I want to show another sorting method, based on *divide and conquer*. Suppose you have to sort the following array:



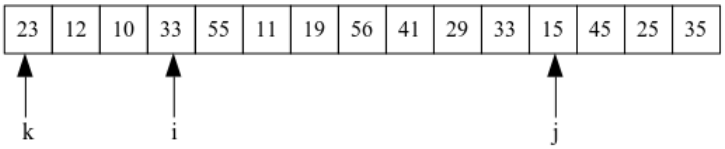
The method now is to select an element – for example the first – and then divide the array about this element: As long as the *i*th element is smaller than the *k*th moves *i* one step to the right, and as long as the *j*th element is larger than the *k*th moves *j* one step to the left:



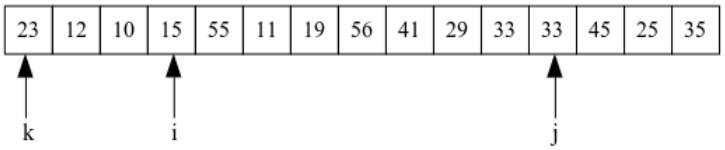
Now the *i*th and the *j*th element should be exchanged:



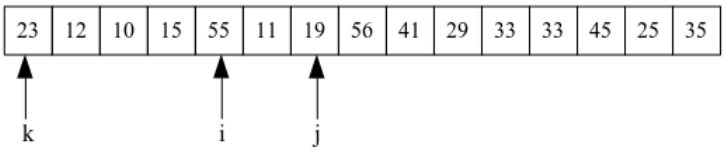
i and *j* are moved again according to the same rules:



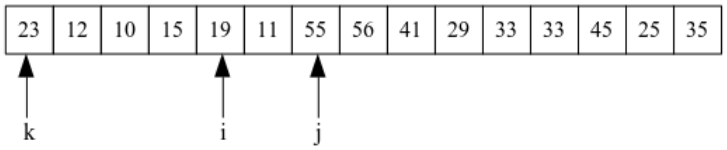
and the *i*th and the *j*th element must be exchanged:



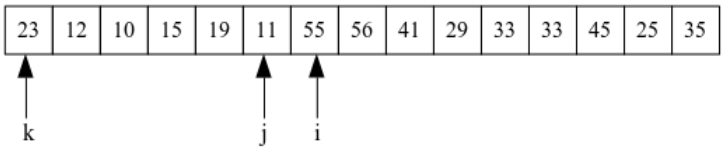
i and *j* are moved again:



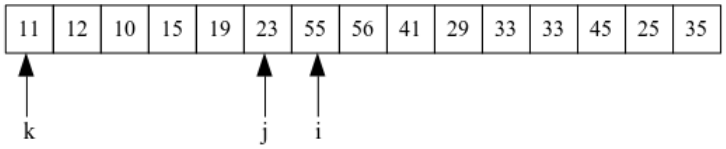
and elements are exchanged:



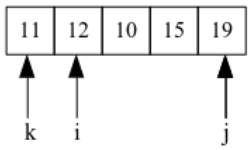
i and *j* are moved again, still after the same rules, but such that you stops when *j* becomes less than *i*:



Finally the *k*th and the *j*th element are exchanged:

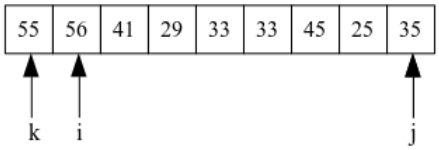


The array has not been sorted but all numbers to the left of 23 are less than 23 while all numbers to the right of 23 are greater than 23. That is, that 23 is in the right place. The problem has also been divided into two smaller sub-problems, consisting of sorting the left part of the array and the right part. If you divide the left part according to the same principles, you need to divide it around 11:



Once that is done, 11 is in place, and you get two sub-arrays, where the left has only one element – it's a simple problem – while the right has three elements. The method is repeated on these three elements, but nothing happens as this sub-array is sorted.

Now the right part of the original array must also be sorted according to the same principles:



and finally the array is sorted:

10	11	12	15	19	23	25	29	33	33	35	41	45	55	56

The method is called for *quick sort*. Its complexity depends on how lucky you are with the selection of the elements used to divide the arrays. Above I chose the first one each time. If you are lucky, and this means that the problem is divided into two equal problems each time, the length of the sub-problems is halved each time, which is proportional to *log(n)*. Each time, the entire array must be passed through to divides the problem, and the result is that quick sort has the best complexity there is $O(n\log(n))$. If the array is already sorted, the use of the first element as pivot point means that the left part always becomes empty and the number of times to divide is thus n. The result is a worst complexity that $O(n^2)$, thus the same as for the simple sorting methods. Assuming that the sorting of the array from the start is random, there is reason to believe that the division results in sub-problems, which are approximate of the same size, and it is possible to show that the mean complexity is also $O(n\log(n))$.

The idea behind quick sort is the same as I've previously shown in determining the median of an array. There I described a method that determines the kth smallest element in an array,

and it is actually a partially quick sort. Quick sort is generally a good sorting method, and it is even the method used most frequently in practice. As mentioned above, its effectiveness depends on how lucky you are with the selection of sub-arrays. If the array is already sorted or almost sorted, then the selection of the first element (or the last element) as a pivot point is unfortunate, as the result is an $O(n^2)$ algorithm. The problem is actually worse than that, because on a large array, there will be a sudden overflow due to the recursive implementation. You can therefore be interested in whether you can do better and do something that handle the case where the array is almost sorted. A simple strategy is each time selecting the middle element as a pivot element. It solves the problem of almost sorted arrays, but of course it does not provide a guarantee as you can also be so unfortunate that you can split up at two very different intervals each time. However, the strategy of choosing the middle element gives a much less probability of unlucky division of the array. Ideally, you should choose the median as a pivot point, but the problem is that it is difficult to determine the median, so you often go another way to choose the median of the first, middle and last element (the same way that I have previously described in connection with a method that determines the median). It provides a significant improvement of quick sort.

Quick sort is usually implemented recursively, where the array is split up, as described above, after which you sort each part separately – recursively with quick sort:

```
public static <T extends Comparable<T>> void quickSort(T[] t)
{
 quickSort(t, 0, t.length - 1);
}

private static <T extends Comparable<T>> void
quickSort(T[] arr, int a, int b)
{
 if (a < b)
 {
  int p = partition(arr, a, b, pivot(arr, a, b));
  quickSort(arr, a, p - 1);
  quickSort(arr, p + 1, b);
 }
}
```

Here are *partition()* and *pivot()* the same methods that I looked at in the section about the median.

PROBLEM 2: THE SORTING METHODS

In this task you should test the 6 sorting methods discussed in this chapter and compare them with the Java's sorting method.

You must first check whether the 6 methods sorts correctly. That is, you must create an array that is not sorted and for each of the 6 sorting methods print this array, sort it and print it again.

Next, you must use each of the 6 methods and Java's own sorting method to test how long it takes to sort an array of random integers.

The result could be as shown below:

```
Bubble sort:
31 43 29 23 41 7 61 13 3 67 37 59 19 73 5 97 17 71 11 2 89 47 53 79 83
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Selection sort:
31 43 29 23 41 7 61 13 3 67 37 59 19 73 5 97 17 71 11 2 89 47 53 79 83
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Insertion sort:
31 43 29 23 41 7 61 13 3 67 37 59 19 73 5 97 17 71 11 2 89 47 53 79 83
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Shell sort:
31 43 29 23 41 7 61 13 3 67 37 59 19 73 5 97 17 71 11 2 89 47 53 79 83
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Merge sort:
31 43 29 23 41 7 61 13 3 67 37 59 19 73 5 97 17 71 11 2 89 47 53 79 83
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Quick sort:
31 43 29 23 41 7 61 13 3 67 37 59 19 73 5 97 17 71 11 2 89 47 53 79 83
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Bubble sort       :     100000 elements,     34311 milliseconds
Selection sort    :     100000 elements,      8393 milliseconds
Insertion sort    :     100000 elements,     12337 milliseconds
Shell sort        :   10000000 elements,     18424 milliseconds
Merge sort        :   10000000 elements,      4318 milliseconds
Quick sort        :   10000000 elements,      3546 milliseconds
Java's sort       :   10000000 elements,      4121 milliseconds
```

5 DATA STRUCTURES

The rest of the book deals with classical data structures and thus what is known in Java as collection classes, but focusing on the implementation. As mentioned in the introduction, the goal is

- to present the most important collection classes for practical programming and their main characteristics
- to show how these classes can be implemented in Java, and especially how the classes are implemented in the Collection API
- to learn programming techniques that are useful in practice and especially how to work with linked objects
- to provide examples of how to implement your own collection classes that are not included in Java's API

It should be emphasized that the goal is not to write a new API as an alternative to Java's classes, but the aim is to show how these classes work and are implemented, which may be important knowledge in daily programming. Specifically, I will implement a number of collection classes, all of which are found in the library *palib* in the package *dk.data.*

torus.collections, but unlike Java's collection classes, the following classes do not constitute a common class hierarchy. For example, will the class *Stack* be a class with its own right and only offer the services that one would generally expect from a stack.

In addition, to many of the classes there will be exercises and problems that will illustrate how to instantiate objects of the classes and apply their most important methods, and in accordance with what is said in the introduction, I will only show the code to a limited extent.

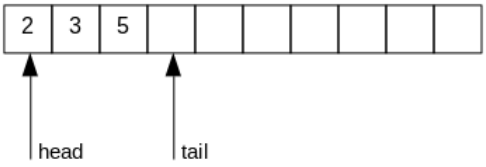
5.1 A BUFFER

To start somewhere, I will write a class *Buffer*, which is a class I have already looked at in connection with generic data types in the book Java 4.

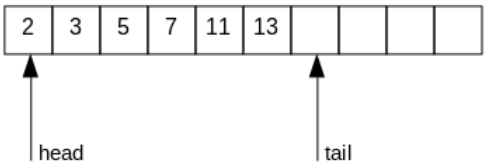
All collection classes are a form of container class which can contain a number of objects of a giventype. Internally, the classes can differ a lot as to how they represent or store the objects, but also with regard to their capacity, and whether a collection has a fixed size or, in principle, is unlimited and grows with the number of elements filled in it. The meaning is that whoever that the user of a particular collection class should not have (or need to have) knowledge of these details, but a particular class must be known by its methods and hence the features it provides. Therefore, you sometimes also talk about an abstract data type or ADT. In this section, I want to look at a collection *Buffer*, which is, in fact, nothing but an wrapper of an array, but its features, and what it's like and what applications it has is determined by the type's methods and properties. A buffer is characterized by three important features:

1. It has a method *add()*, so you can add an object to the buffer – so you can insert elements into the buffer.
2. It has a method *remove()*, so you can remove the oldest element in the buffer.
3. The buffer has a final size and thus a capacity that can not be expanded.

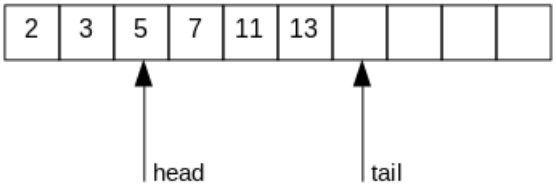
A buffer can therefore be illustrated as follows:



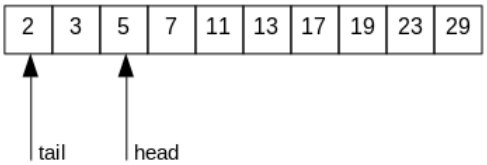
where the buffer holds up to 10 elements and currently contains 3. The buffer has two pointers, where the head indicates the oldest element, and thus the next element that can be removed from the buffer while the tail indicates where the next elements added to the buffer must be placed. Below is the buffer shown if additional 3 elements are inserted:



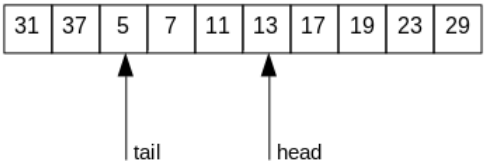
If you remove two elements then the result is:



Note that the elements removed are still in the buffer – there is just no access to them, as there is onlyaccess to the element that *head* points to. Below is the buffer shown after other 4 elements has been added:



That is, that *tail* now points to the first element, as the next available place where an element can be inserted. Thus, the two pointers must swap as they move to the right of the last element. Of course, you can only add to a buffer if there is room and when implementing a buffer, you must make sure that you will not overwrite elements that have not yet been removed. Below is the result of three *remove()* operations and followed by two *add()* operations:



A buffer is also called a FIFO (First In First Out) structure that corresponds to that the element that was first inserted into the buffer is the element to be removed first. If the buffer is implemented as illustrated above, it is often called for a circular buffer.

A buffer is typically used in situations where a container of a fixed size is needed and it will often be combined with tasks where performance is important. Therefore, it is necessary to aim for its operations to be effective and, in practice, it means that they should have constant time complexity.

In this book, a collection class will typically be implemented as a generic class, and in addition, the class will usually be defined by an interface. In this case, the class is defined by the following interface:

```
package dk.data.torus.collections;

import dk.data.torus.PaException;
/**
 * Defines a generic buffer for objects of an arbitrary type.
 * It is assumed that the buffer has a fixed size.
 */
```

```
public interface IBuffer<T>
{
 public static final String EmptyException = "The buffer is empty";
 public static final String FullException = "The buffer is empty";

 /**
  * @return The size of the buffer
  */
 public int getSize();
 /**
  * @return Number of elements in the buffer
  */
 public int getCount();

 /**
  * @return true, if the buffer is empty
  */
 public boolean empty();

 /**
  * @return true, if the buffer is full
  */
 public boolean full();

 /**
  * Returns the oldest (first) element in the buffer without removing it,
  * @return The oldest (first) element of the buffer
  * @throws PaException If the buffer is empty
  */
 public T peek() throws PaException;

 /**
  * Returns The oldest element in the buffer and remove it from the buffer.
  * @return The oldest (first) element of the buffer
  * @throws PaException If the buffer is empty
  */
 public T remove() throws PaException;

 /**
  * Adds an element to the buffer.
  * @param elem The element to be added
  * @throws PaException If the buffer is full
  */
 public void add(T elem) throws PaException;
}
```

which by the way is the same interface as in the class *Generic* from the book Java 4. Compared to what has been said above, a few methods have been added that can return properties

regarding a buffer's state. I want to use the convention that the name of an interface starts with an uppercase I, and here as *IBuffer*.

When implementing/defining collection classes there are many choices, and of course, what methods the class should have. As other important decisions I will mention, which methods should raise exceptions, and usually you should not raise exceptions, just catching exceptions raised by Java's own classes, and especially Java Runtime Exceptions, which indicates exceptions that are perceived as serious exceptions where it does not make sense to continue the execution of the program. In this case, there are three of the methods that can raise an exception and an exception type has been defined which is used in conjunction with all the following collection classes and, in general, classes in the library *palib*.

In addition to the interface *IBuffer*, the package *dk.data.torus.collections* has added a class *Buffer<T>* that implements the interface according to the guidelines described above.

EXERCISE 6: TEST BUFFER

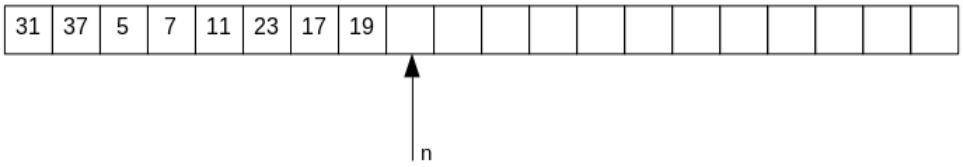
You must write the following program that can test the class *Buffer*. That is, you should write the code for the two methods:

```
public class BufferProgram
{
 private static final Random rand = new Random();

 public static void main(String[] args)
 {
  IBuffer<Integer> buffer = new Buffer(10);
  for (int i = 0; i < 100; ++i)
   if (rand.nextBoolean()) add(buffer); else remove(buffer);
 }

 // The method inserts a random number between 10 and 99 in the buffer.
 // If possible, the method must print which number has been inserted into
 // the buffer and otherwise the method must print an error message.
 private static void insert(IBuffer<Integer> buffer)
 {
 }

 // The method remove a number from the buffer and print the number.
 // If this is not possible, the method must print an error message.
 private static void remove(IBuffer<Integer> buffer)
 {
 }
}
```

5.2 ARRAYLIST

In practice, the most commonly used collection class is an *ArrayList* and in this section I will show how this class is implemented and why it is highly effective.

Internally, an *ArrayList* is an array with place for a given number of elements, and the elements are thus stored in places from index 0 and forward. An *ArrayList* thus defines an arrangement of the elements based on their location or number/index. An *ArrayList* can not have holes. That is, if the list contains an element at place *n*, there are also elements at the places 0, 1, 2, …, n-1. An *ArrayList* may be empty if it contains no elements at all. You can think of an *ArrayList* as a data structure, that is an array of elements, and an internal pointer referring to the first available empty place:

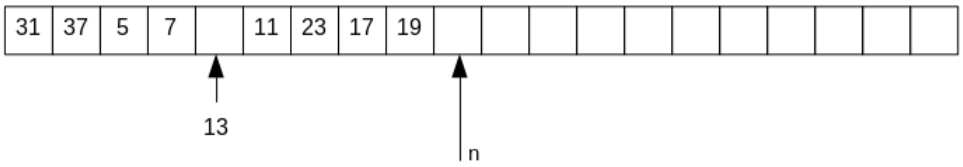


Here, there is room for 20 elements, while there currently are 8. The index *n* has thus the value 8, as where the next element has to be placed.

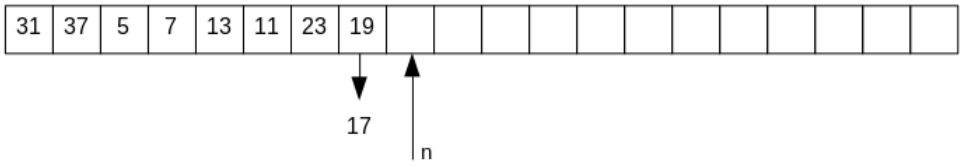
An *ArrayList* is similar to other data structures characterized by its properties in terms of properties and methods, and an *ArrayList* basically provides the following operations:

1. *add()* that adds an element to the end of the list
2. *insert()* that inserts an element at a specific position in the list
3. *remove()* that deletes an element with a specific index
4. *get()* that returns an element with a specific index
5. *getSize()* that returns the number of elements in the list

There are three of the operations – *insert()*, *remove()* and *add()* – that require a more detailed explanation. If you insert an element in place *n*, all subsequent elements must be moved a place to the right so that there is room for the new element. For example, if you insert item 13 into the place with index 4 in the above list, the result will be the following:



If you delete the *n*th element, all subsequent elements must be moved a place to the left. If you delete the element with index 7 (after 13 is inserted), the result will be the following:



Looking at these two operations, they mean that the list's elements must be moved and contains the list *N* elements, on average, *N / 2* elements must be moved. That is, both operations have a time complexity that is *O(N)*.

Then there is the method *add()* used to insert an element where the arrow points, and then move the arrow a position to the right. Thus, the operation does not depend on the number of elements and thus has constant time complexity, and it is even a very simple operation. It's at least true, as long as there is room, but if the list is filled up, there's more to do. The strategy is to create a new array with double capacity and then copy the old array to the new, after which the old is left to the garbage collector. It sounds extensive, and that it also is, but the whole idea is that it does not happen so often.

An extension consists of creating a new array and copying the content of the old array to the new array. This copy has linear time complexity, since it consists of a loop over an array and has the array *N* elements, the copy thus requires approximate *N* operations. Suppose an array contains exactly *N* elements. If you then perform the following operations:

```
for (int i = N; i < 2 * N; ++i) array[i] = x;
```

then the first assignment will require *N* operations as the array is expanded, while each of the next *N-1* operations has constant time complexity as they do not require the array to be expanded. The average complexity of these *N* operations is then

$$\frac{N+N-1}{N} \sim 1$$

That is, that the *add()* operation in the average has constant time complexity and it is the whole idea of an *ArrayList*, that it is highly effective to add elements to the end of the list and the list automatically is expanded without involving the user. However, the price is use of memory.

Above I have described how an *ArrayList* is expanded by doubling the array, but it is not a requirement, and several implementations use another factor. The result, however, is that constant time complexity of the method *add()*, and the choice is a weighing of performance in relation to memory consumption.

I have defined a list as a data structure with the following properties and methods:

```
package dk.data.torus.collections;

public interface IList<T> extends Iterable<T>
{
 public int getCapacity();
 public int getSize();
 public boolean isEmpty();
 public T get(int n);
 public boolean set(T elem, int n);
 public boolean contains(T elem);
 public void clear();
 public int find(T elem);
 public void add(T elem);
 public boolean insert(T elem, int n);
 public boolean remove(T elem);
 public boolean removeAt(int n);
}
```

The names of the methods should explain what they should be used for, but otherwise you can see in the code where there are comments. When implementing a data structure, there are many considerations, including what services it should make available. The above interface thus tells which services I have decided that a list should have, but there may be others and more, and the corresponding interface in *java.util* defines more services. You should also note that my interface extends *Iterable*<*T*> and an list thus will implement the iterator pattern. Also note that no methods raises exceptions. It is also a choice and means that all methods can be used without placing the code in a try/catch block. In return, several of the methods return a *boolean* so the user can test if the method was performed correctly. The interface is implemented by the class *ArrayList*, and you are encouraged to study the code, which is quite simple as well.

EXERCISE 7: TEST ARRAYLIST

You must write a program that can test the class *ArrayList*:

```
public class ArrayListTester
{
 public static void main(String[] args)
 {
 }

 private static void test1()
 {
  // Creates a IList<Integer> from an array with 13 numbers.
  // The method must print the list by use of an iterator.
 }

 private static void test2()
 {
  // Creates an ArrayList and add() the numbers 1 to 10000 in that order.
  // The method must then print the sum of the numbers in the list using the
  // method sum(). The result should be 50005000.
  // Then the method should do the folowing operations:
  // Remove the element with the value 1000
  // Remove the element with index 499
  // Insert the number 1000 at index 0
  // Insert the number 500 af index 1000
  // Print the sum of the list again. The result should be the same.
 }

 private static void test3(int n)
 {
  // Create an ArrayList<Integer> with n numbers (1 - n), where the list is
  // filled in a for loop with the method add().
  // The method must print the number of milli seconds for the above operations.
 }

 private static void test4(int n)
 {
  // The method must do exactly the same as test3() but with the difference
that
  // the list must be Java's ArrayList.
 }

 private static long sum(List<Integer> list)
 {
  // Returns the sum of all numbers in list by use of a for loop and get()
 }
}
```

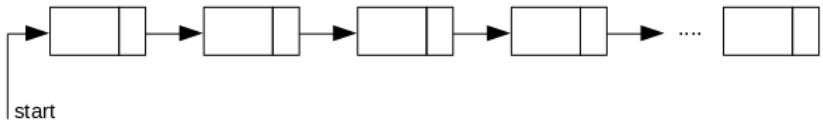
6 LINKED LISTS

A linked list is a collection that, like an *ArrayList*, can contain any number of objects organized in a particular order. It is a sequential structure where new elements are placed by directly or indirectly specifying the new element's predecessor or successor, and where you can traverse the list and process the objects in the order they are inserted. For example, you can browse the list from start to finish. A linked list is an alternative to an array or an *ArrayList*. An array (and also an *ArrayList*) is a coherent data structure in the machine's memory:

```
DataType[] t = new DataType[n];
```

and it is characterized by direct access to its elements via an index, that is, you can refer to the *i*th element as *t[i]*. An array thus gives direct access to the individual elements, which means that it is a very efficient data structure. For a linked list, the conditions are opposite. A linked list consists of a reference that refers to the list's start and each element in the list has a reference to the next element:

```
ListElement start = new ListElement();
```



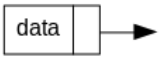
You can add new items to the list at any time or remove an existing item, so the list can grow or decrease as needed. On the other hand, to refer to a specific item in the list, you must start at *start* and search the list until you find the element. You must note that unlike an array (and an *ArrayList*), a linked list is not a coherent structure, and relative to an array, it's exactly both its pros and cons.

Lists can be organized in practice in many different ways, and below are discussed more methods of structuring lists and the associated algorithms that are used. The list illustrated above is an example of a *single linked list*.

6.1 SINGLE LINKED LISTS

The simplest list you can have is a single none ordered linked list. It is characterized by the list consisting of elements (also called nodes) that contains a data object and a reference to the next element/node:

```
class Node
{
 DataType data;
 Node next;
```

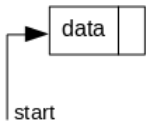


and a list is merely a reference to the element that is the first node in the list corresponding to the figure above. Basically, the following operations are required:

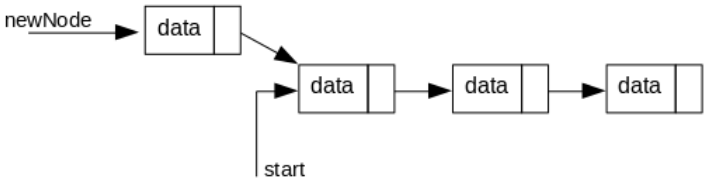
1. Insert a new element at the start of the list
2. Insert a new element after an existing element
3. Returns the number of elements in the list
4. Check if the list is empty

5. Traverse the list, so all elements in the list are processed
6. Remove the first element in the list
7. Remove a specific element in the list

An operation to insert an element at the start of the list is simple. If the list is empty, it should not happen other than creating a new node, and *start* (or *first*) is set to refer to this node:

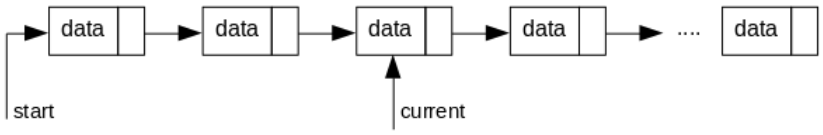


If the list is not empty, a new node should be created that refers to the node that before was the start:

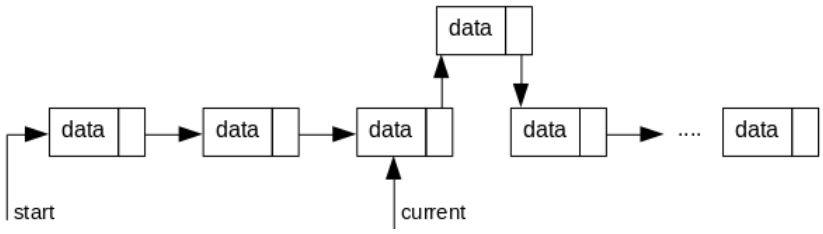


after which *start* is set to point to the new node:

Another option for insertion is to insert an element inside the list, for example, to insert an element as a successor to an existing element. An algorithm for this function will consist in a search to find the element *current* as the new node must follow

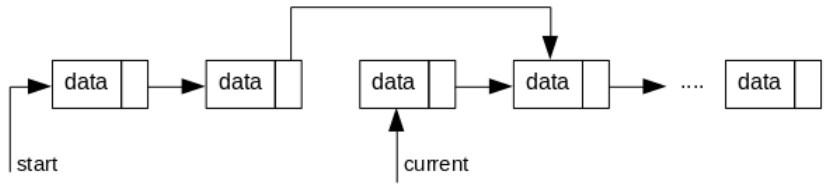


Then you can insert the new element as shown below, where the new element points to the successor of *current*, while *current* should point to the new element:



That is, after the element *current* is found, the new element can be inserted by creating a new node and setting two pointers and thus with constant time complexity.

If you want to remove an element from the list, you must find the element to be removed in a search, after which the element's predecessor should point to the element that the element to be removed points to:



After that, there is no longer a reference to the element to be removed, and the garbage collector takes care of the rest. However, it should be noted that remove of the first element is a special case where it is *start* that must point to the successor. Also, be aware that the algorithm should also work in the case that it is the last element to be removed, and finally, note that the list may become empty after an element has been removed.

The class *SingleList* implements a single linked list as described above, but with the extension that there also is a reference that refers to the last element in the list, as it may often be useful in practice. To implement the list in Java, you must represent a node:

```
package dk.data.torus.collections;

/**
 * Represents a node for a single linked list. The constructors and the method
 * setNext() has package visibility, which means that nodes can not be created
 * and node references can not be changed outside the package.
 */
public class SingleNode<T>
{
```

```
private T element; // the data element
private SingleNode<T> next; // pointer to the next node

SingleNode()
{
}

SingleNode(T element, SingleNode<T> next)
{
 this.element = element;
 this.next = next;
}

public T getElement()
{
 return element;
}

public void setElement(T element)
{
 this.element = element;
}
```

```
 public SingleNode<T> getNext()
 {
  return next;
 }

 void setNext(SingleNode<T> next)
 {
  this.next = next;
 }
}
```

The class is simple and is just an encapsulation of two variables, but you must note the comment and how the constructors and the method *setNext()* are defined with package visibility. Then the list can be defined as follows:

```
public interface ISingleList<T> extends Iterable<T>
{
 public SingleNode<T> getFirst();
 public SingleNode<T> getLast();
 public boolean isEmpty();
 public int getSize();
 public void add(T elem);
 public void insert(T elem);
 public void insertAfter(SingleNode<T> current, T elem);
 public boolean contains(T elem);
 public SingleNode<T> find(T elem);
 public void clear();
 public boolean remove(T elem);
 public boolean removeFirst();
 public boolean removeAfter(SingleNode<T> current);
}
```

Again, the methods corresponding to the above discussion of a single linked list should be self explanatory. The interface is implemented by the class *SingleList*.

EXERCISE 8: TEST SINGLELIST

You must write the following program that can test the class *SingleList*:

```
public class SingleLinkedListTest
{
 public static void main(String[] args)
 {
 }
```

```
 private static void test1()
 {
  // Create a SingleList<String> and do the following operations:
  // use insert() to insert a name add the start of the list
  // use insert() to insert a name add the start of the list
  // use add() to insert a name as the last name in the list
  // use insertAfter() to insert a name after the first name you have inserted
  // in the list
  // use insert() to insert a name as the last name in the list
  // use insertAfter() to insert a name after the last name in the list
  // Print the list by using the iterator.
 }

 private static void test2()
 {
  // Create a list and use add() to add the following numbers to the list:
  // 2 3 5 7 11 13 17 19
  // User the method print() to print the list
  // Remove the number 13
  // Remove the first number in the list
  // Remove the number after 11
  // User the method print() to print the list
  // Insert 23 after 7
  // Add 29 to the list
  // Insert 31 as the first element in the list
  // User the method print() to print the list
  // The result should be:
  // 2 3 5 7 11 13 17 19
  // 3 5 7 11 19
  // 31 3 5 7 23 11 19 29
 }

 private static void test3(int n)
 {
  // Create an SingleList<Integer> with n numbers (1 – n), where the list is
  // filled in a for loop and the method add().
  // The method must print the number of milli seconds for the above operations.
 }

 private static void print(ISingleList<Integer> list)
 {
  // print the list, when it must happens in a for loop of the form:
  // for (node = first node; node != null; node = next node) print element
 }
}
```

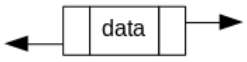
You must note the result of the method *test3()*, that compared with the previous exercise, shows that a linked list is less effective than an *ArrayList*. In fact, it does not have a better

memory complexity. An *ArrayList* fills in worst case twice as much as is necessary, and contains an *ArrayList* *N* elements, it can at worst contain *N* references that are not used. Each node in a linked list will always have a reference (to the next node), and contains the list *N* elements, it will also have assigned *N* node references, and therefore its memory complexity is not better than an *ArrayList*. You can therefore ask where the advantageous is, and it is the time complexity of *insert()* which inserts an element at the start of the list as it is *O(1)*.

6.2 DOUBLE LINKED LISTS

The above single linked list has its uses and is characterized by that it has a simple implementation, but often you choose to implement a double linked list, where each node not only keeps track of the successor, but also the predecessor. By giving the individual nodes a reference to the predecessor, you will also be able to search backward, for example, to start at the end of the list and traverse the list in the opposite direction. It is advantageous for many tasks, and some methods can actually be implemented more easily, but the price is that the individual nodes must contain an additional reference:

```
class Node
{
 DataType data;
 Node prev;
 Node next;
```



In order to access the individual elements, the list must consist of a reference *start* that can point to the first element, as well as a reference *end* than may point to the last element:

Insertion and deletion of elements takes place in principle in the same way as for a single linked list, but there are now several references to keep track of. For example, a method *insertAfter()* can be described as follows:

1. Find the node *current*, which the new element must follow
2. create a new node that points back to *current* and forward on *current*'s successor
3. *current*'s successor must point back to the new node
4. *current* should point forward to the new node

You should also consider whether the implementation also work if the element is inserted after the last element, which will often be a special case. Similarly, you can describe a method *insertBefore()*. A method to delete a node is the most complex and can be described as something like the following:

1. Find the node *current*, which contains the item to be deleted
2. The successor to *current* must point back to the predecessor to *current*
3. The predecessor to *current* must point forward on the successor to *current*

When deletion is more complex than insertion, it is because deletion of both the first and the last element will usually be a special cases. In addition, it should be noted that deleting an element may result in an empty list.

To implement a double-linked list, start with a node:

```
package dk.data.torus.collections;

/**
 * Represents a node for a double linked list. The constructors and the methods
 * setPrev() and setNext() has package visibility, which means that nodes can not
 * be created and node references can not be changed outside the package.
 */
public class DoubleNode<T>
{
 private T element; // the data element
 private DoubleNode<T> prev; // pointer to the previous node
 private DoubleNode<T> next; // pointer to the next node

 DoubleNode()
 {
 }

 DoubleNode(T element, DoubleNode<T> prev, DoubleNode<T> next)
 {
  this.element = element;
  this.prev = prev;
  this.next = next;
 }

 public T getElement()
 {
  return element;
 }

 public void setElement(T element)
 {
  this.element = element;
 }

 public DoubleNode<T> getPrev()
 {
  return prev;
 }

 void setPrev(DoubleNode<T> prev)
 {
  this.prev = prev;
 }

 public DoubleNode<T> getNext()
 {
  return next;
 }
```

```
 void setNext(DoubleNode<T> next)
 {
  this.next = next;
 }
}
```

The class looks like *SingleNode* simply expanded with an additional reference, and in fact the class could be written as a specialization of *SingleNode*. The benefits would be limited, and in some places it might be necessary to have a lot of typecasts. With this class available, you can define a double-linked list as follows:

```
public interface IDoubleList<T> extends Iterable<T>
{
 public DoubleNode<T> getFirst();
 public DoubleNode<T> getLast();
 public boolean isEmpty();
 public int getSize();
 public void insertFirst(T elem);
 public void insertLast(T elem);
 public void insertAfter(DoubleNode<T> current, T elem);
 public void insertBefore(DoubleNode<T> current, T elem);
 public T removeFirst();
```

```
 public T removeLast();
 public boolean remove(T elem);
 public T remove(DoubleNode<T> current);
 public boolean contains(T elem);
 public DoubleNode<T> find(T elem);
 public void clear();
 public Iterator<T> left();
 public Iterator<T> right();
}
```

Java has a class *LinkedList*, and it is a double linked list corresponding to the above description. It is relatively simple to implement a double linked list, but unlike a single list, there are several references to keep track of. The class *DoubleList* implements the above interface.

EXERCISE 9: TEST DOUBLELIST

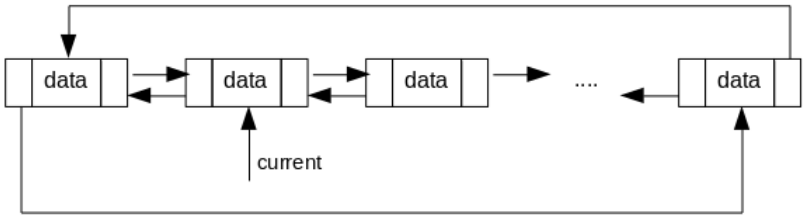
You must write a program that can test the class *DoubleList*:

```
public class DoubleLinkedListTest
{
 public static void main(String[] args)
 {
 }

 private static void test1()
 {
  // Create a DoubleList<String> and do the following operations:
  // use insertFirst() to insert a name at the start of the list
  // use insertFirst() to insert a name at the start of the list
  // use insertLast() to insert a name as the last name in the list
  // use insertAfter() to insert a name after the first name you have inserted
  // in the list
  // use insertLast() to insert a name as the last name in the list
  // use insertBefore() to insert a name before the last name in the list
  // Print the list first by using the iterator right() and then by using
  // the usual iterator.
 }

 private static void test2()
 {
  // Create a list and use insertLast() to add the following numbers to the
     list:
  //  2 3 5 7 11 13 17 19 23 29
  // User the method print() to print the list
  // Remove the number 13
```

```
  // Remove the first number in the list
  // Remove the last number in the list
  // Remove the node with value 23
  // Use the method print() to print the list
  // Insert 31 after 7
  // Insert 37 før 7
  // Insert 41 as the first number in the list
  // Insert 43 as the last number in the list
  // User the method print() to print the list
  // The result should be:
  // 2 3 5 7 11 13 17 19 23 29
  // 29 23 19 17 13 11 7 5 3 2
  // 3 5 7 11 17 19
  // 19 17 11 7 5 3
  // 41 3 5 37 7 31 11 17 19 43
  // 43 19 17 11 31 7 37 5 3 41
 }

 private static void test3(int n)
 {
  // Create an DoubleList<Integer> with n numbers (1 - n), where the list is
  // filled in a for loop and the method insertLast().
 }

 private static void test4(int n)
 {
  // The method must do exactly the same as test3(), but the list must be
     Java's
  // LinkedList and the elements should be added with the method add().
 }

 private static void print(DoubleList<Integer> list)
 {
  // print the list, when you must traverse the list using the iterator right()
  // print the list again but this time by using the iterator left()
 }
}
```

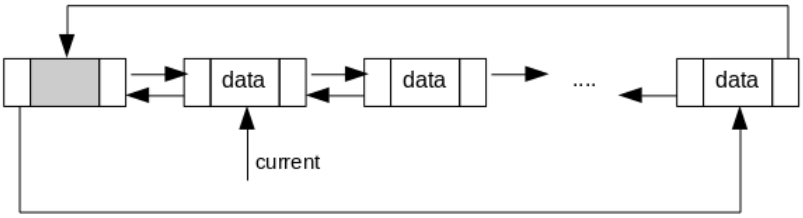
PROBLEM 3: A CIRCULAR LIST

A double linked list can be implemented in several ways, and one of the options is like a circular list, where the last node points to the first, while the first points back to the last one:

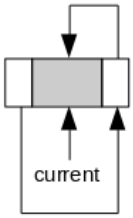


There is no longer any first and last element, and you refer to the list's elements using an internal reference, which refers to a current node. A class that implements such a data structure must therefore have methods that can move this internal reference and insertion in the list occurs before or after the node where *current* points. Similarly, deletion means that the node where *current* currently should be removed.

It is your task to implement such a list. In fact, it is not quite easy, but you can make it easier to introduce a dummy node, which is a node that is always there and does not contain data:



This means that an empty list has the form:



and in the case of an empty list, there is a node to insert before or after. The task is to write a class that implements the following interface as a circular double-ended list:

```
public interface ICircularList<T> extends Iterable<T>
{
 // Set the list to the start dummy node.
 public void reset();

 // Move the current node to the previous node, that could be the dummy node
 public void prev();

 // Move the current node to the next node, that could be the dummy node.
 public void next();

 // Returns the value in the node current.
 public T getValue();

 // Test if the list is empty.
 public boolean isEmpty();

 // Returns the number of elements in the list.
 public int getSize();

 // Inserts elem as new node in the list after the current node.
 public void insertAfter(T elem);
```

```
 // Inserts elem as new node in the list before the current node.
 public void insertBefore(T elem);

 // Remove the first occurrence of elem from the list.
 public boolean remove(T elem);

 // Remove current node and returns the element in the node.
 public T remove();

 // Test if elem is in then list, and if so that node is the current node.
 public boolean contains(T elem);

 // Returns an iterator that traverse the list backwards.
 public Iterator<T> left();

 // Returns an iterator that traverse the list onwards.
 public Iterator<T> right();
}
```

To write the class, use the class *DoubleNode*. It gives a problem as it is in another package. To resolve this issue, take a copy of the class and add it to your own project. My project is called *AnotherList*.

Remember to write a test program.

6.3 AN ORDERED LINKED LISTS

Both a single linked list and a double linked list can be implemented as an ordered list, which simply means that the elements are sorted according to a particular arrangement criterion. For example, if looking at an ordered double-linked list, the only difference compared to a general double-linked list is that insertion of elements should occur in another way. A class should now only have one *insert()* method that inserts an element corresponding to the current arrangements, while the other methods for insertion no longer make sense. Apart from inserting elements, all other methods must be implemented in exactly the same way as in the class *DoubleList<T>*.

The class *SortedList<T expands Comparable<T>>* implements an ordered double-linked list and is defined by the interface:

```
public interface ISortedList<T extends Comparable<T>> extends Iterable<T>
{
 public DoubleNode<T> getFirst();
 public DoubleNode<T> getLast();
 public boolean isEmpty();
 public int getSize();
 public void insert(T elem);
 public T removeFirst();
 public T removeLast();
 public boolean remove(T elem);
 public T remove(DoubleNode<T> current);
 public boolean contains(T elem);
 public DoubleNode<T> find(T elem);
 public void clear();
 public Iterator<T> left();
 public Iterator<T> right();
}
```

Aside from the methods for insert of elements is the same interface as *IDoubleList* – and that the data type *T* this time must implement *Comparable*. Therefore, the class *SortedList<T extends comparable<T>>* can basically be implemented in the same way as *DoubleList<T>*. Just are the four methods to insert elements been removed and replaced by one new method.

When the elements need to be *Comparable*, it does not make sense to insert an element at the beginning of the list, at the end of the list or after or before a particular node. Each time you insert an element, is it necessary with a search and find where the element should be inserted.

EXERCISE 10: DOUBLE LINKED LISTS

Create a new project that you can call *SortedLinkedListTest*. Copy the following classes (files) from the *palib* project to the new project:

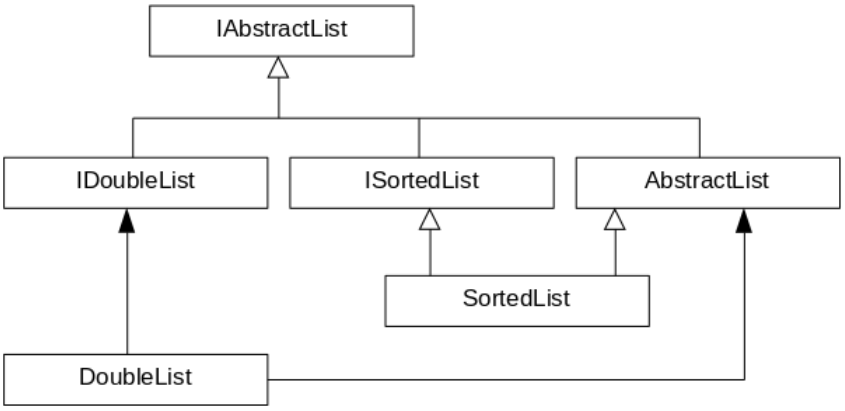
- *DoubleNode*
- *IDoubleList*
- *DoubleList*
- *ISortedList*
- *SortedList*

Add a simple model class *Name* that can represent the name of a person when a person is represented by a first name and a last name. Objects of the type *Name* must be comparable when ordered first by last name and then by first name.

You must then write the *main()* method of your project:

```
public class SortedLinkedListTest
{
 public static void main(String[] args)
 {
  // create a SortedList<Name>
  // insert 12 Name objects in list
  // print the list on the screen
  // remove a name by calling the method remove()
  // remove the name last added to the list by calling the method remove()
  // remove the first name added to the list by calling the method remove()
  // print the list again
 }

 private static void print(ISortedList<Name> list)
 {
 }

 private static void remove(ISortedList<Name> list, Name name)
 {
 }
}
```

Create an interface called *IAbstractList<T>*, which is identical to *IDoubleList<T>*, except that the insertion methods has been removed. Then write an abstract class *AbstractList<T>*, which implements the interface *IAbstractList<T>* when the class must be identical to *DoubleList<T>*, except that all of the insertion methods should be removed. In addition, you must define the three instance variables as *protected*. Modify the interface *IDoubleList<T>* to inherit *IAbstractList<T>*. That is, it should only define the four methods for inserting elements into a double linked list. You must then change the class *DoubleList<T>* so that it inherits *AbstractList<T>* and implements *IDoubleList<T>*. That is, the class alone must contain the four methods of inserting elements in the list. As the next step, change the interface *ISortedList<T extends Comparable<T>>* to inherit *IAbstractList<T>*, and the interface should thus only define the method *insert()*. Finally, change the class *SortedList<T extends Comrarable<T>>* so the class inherits *AbstactList<T>* and implements *ISortedList<T extends Comparable<T>>*. The result is that you have created the following type of hierarchy:



Run the test program again and the program should work as before.

6.4 A STACK

I have previously described the data structure a *Stack* as a LIFO structure, a data structure that basically have two operations *push()* and *pop()*. It's a pretty simple data structure, but useful, and it's similarly simple to implement. The only important decision is whether it should have a fixed size, or you should be able to put on the stack without thinking about the memory. Here I will look at a stack of the last kind and hence a data structure that can be defined as follows:

```
public interface IStack<T>
{
 public static final String EmptyError = "The stack is empty";
 public void push(T elem);
 public T pop() throws PaException;
 public T peek() throws PaException;
```

```
 public int getSize();
 public boolean isEmpty();
 public void clear();
}
```

Such a stack can be immediately implemented as an enclosure of an *ArrayList*, and the class *ArrayStack<T>* is an example.

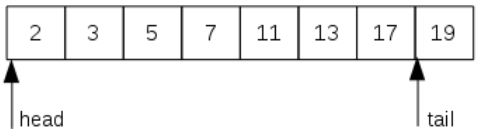
EXERCISE 11: A STACK OF FIXED SIZE

You must implement a stack with a fixed size. The interface must be as above except that there must now also be a method *isFull()* and the method *push()* can now raise an exception. You can implement the stack using a simple array, but this time there must be a constructor that specifies size.

Once you have written the stack, you must create a stack for integers and room for 10 numbers in a test program. You must then iterate a loop where you push numbers on the stack until it is full. Next, you should similarly iterate a loop where you pop the stack and print the values until the stack is empty.

6.5 A QUEUE

A stack is a data structure or abstract data type that can be manipulated by two operations *push()* and *pop()*. It is a LIFO structure (Last In First Out), as the element removed from the stack with *pop()* is the last element put on the stack. A data structure that is very similar to a stack is a queue, but it is a FIFO structure (First In First Out). You can think of a queue as a sequence of objects with two references



where *head* indicates the start of the queue while *tail* indicates the end. This is exactly the same as what I above have called a buffer and a class like *Buffer<T>* is a queue, and the difference is, among other things, that the operations are called something else:

1. *enqueue()* that inserts an element at the end of the queue – after the element that *tail* points to
2. *dequeue()* that removes the first or oldest element in the queue – the element that *head* points to

As mentioned, a queue can be implemented as a *Buffer<T>*. Doing so, the queue has a capacity, by setting the size of the buffer when it is created. It is not always advisable or desirable that the queue has a fixed size, and if not, it is difficult to implement the queue using an array, such that the above operations get constant time complexity. However, you can implement the queue using a linked list, allowing you to expand the queue as needed, such that both insertion and deletion have constant time complexity. A queue can be defined as follows:

```
public interface IQueue<T>
{
 public void enqueue(T elem); // add an element to the queue
 public T dequeue(); // remove the first element
 public T peek(); // return the first element in the queue
 public void clear(); // remove all elements in the queue
 public int getSize(); // return the number of elements
 public boolean isEmpty(); // test where the queue is empty
}
```

Note that the interface does not define the iterator pattern, for which there is no particular reason, but you can say that the basically it not the idea that you should be able to traverse the elements in a queue. Thus, it is a matter of choice whether it should be possible and Java's implementation makes it possible.

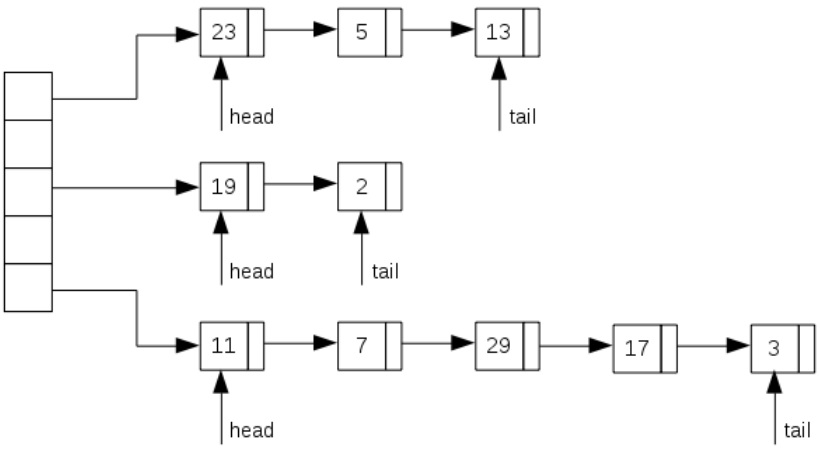
If you look at the class *SingleList<T>*, it has methods to delete the first element and to insert an element at the end of the list, and both with constant time complexity. You can therefore implement a queue as a class that wrapper a *SingleList*, what the class *Queue<T>* does.

A priority queue is a queue where you can insert elements corresponding to a given priority, which means that you do not necessarily have to insert at the end of the queue. For example, if having the queue as shown above, it should be possible to insert an element between 5 and 7. All other operations should work as for a regular queue. I do not want to implement a priority queue here, as I can do better after I have presented another data structure, but in the following exercise you must write a class that seems like a priority queue, but not with the desired time complexity.

EXERCISE 12: A PRIORITY QUEUE

Create a project that you can call *QueueTester*. The program must have a method *test1()* that creates a *Queue<String>*. Then the method must place 10–15 names in the queue, after which the method must remove all elements from the queue and print the names on the screen. Note the order.

There is a very simple implementation of a priority queue where a priority queue is implemented as an array of queues with a queue for each priority. Below is illustrated a priority queue with 5 priorities which contains 10 elements with three elements that have the priority 0, no elements with priority 1, two elements with priority 2, no elements with priority 3 five elements with priority 4:



Add the following interface to your project:

```
package queuetest;

import dk.data.torus.collections.IQueue;

public interface IPriorityQueue<T> extends IQueue<T>
{
 public void enqueue(T elem, int priority);
}
```

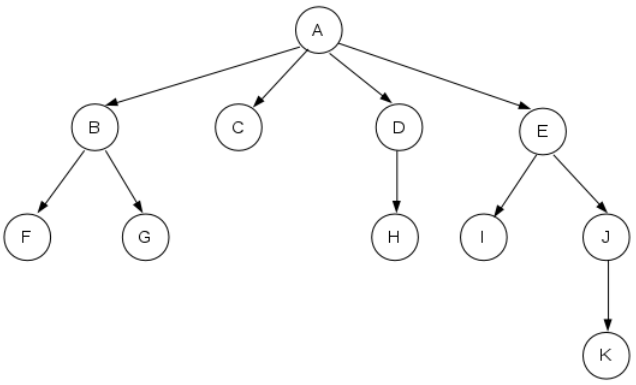
which defines a priority queue where the elements can be inserted in the queue corresponding to a given priority. You must then add a class that implements this interface and thus a priority queue when it is to be implemented corresponding to the above figure as an array of *Queue* objects. You should also consider the time complexity of each of the queue's methods.

After writing the queue, write a test method that is identical to the first test method *test1()*, but such that the names this time are inserted into a priority queue with 26 priorities and where the names are inserted with a priority determined by the first letter of the name, for example as:

```
queue.enqueue("Magnus den Gode", 'M' - 'A');
```

7 TREES

A tree is a collection that organizes objects in a hierarchical structure, for example:



that is a tree with 11 objects. The letters mean nothing but I can refer to the individual objects. It is said that a tree consists of nodes, where each node contains an object, and in this case there are 11 nodes. One of these nodes plays a special role and is the entrance to the data structure, and in this case it is the node A. The node is called the *root* of the tree. In order for the objects to form a tree, there must be a unique path from the root to any other node. That is, any node different from root has a unique predecessor, which is called its parent. For example, E has parent A while I has the parent E. This context is illustrated in the figure with lines from one node to another, and each node has a unique path or road from the root to the node. As examples of roads can be mentioned

[A, D]
[A, D, H]
[A, B, G]
[E, J, K]

where the first is a path from A to D, the next one a path from A to H, the third one a path from A to G and the last one a path from E to K. The number of lines in a road is called the road length (or path length). Here the first path has the length 1, while the other three have the length 2. Note that the path from A to K has the length 3. That there is a unique path from the root to a node, one can also express that each node is characterized by being root in a sub-tree. For example is B root in a 3-node sub-tree. Note that a tree can specially be empty.

There are many other definitions attached to a tree. A given node may have a number of direct successors. For example, E has two direct successors, which are I and J. They are also

called child nodes to E, and one say that I and J are *siblings*. Similarly, the root A has 4 child nodes (B, C, D, E), and they are all siblings. Some nodes have no child nodes. This applies, for example, to H and K. Such nodes are called *leaf* nodes. For each node is also associated a *degree* that is defined as the number of children. Thus, the root A has grade 4 while D has grade 1. Note that a leaf node is characterized by having degree 0. As the degree of the tree you should understand the greatest degree of all the nodes in the tree. Thus, the above tree has degree 4. For each node there is also a height which is the length of the path from the root of the tree to that node. That is, B has the height 1 while K has the height 3. As the height of the tree you should understand the largest height of all the leaf nodes in the tree and the above tree has the depth 3. Sometimes you speak instead of depth or level and you say, that nodes of the same depth are on the same level. In particular, it is said that an empty tree has the height -1, whereas a tree that only has the root has the height 0. There are several simple rules for trees, and for example, if a node N has parent P, then

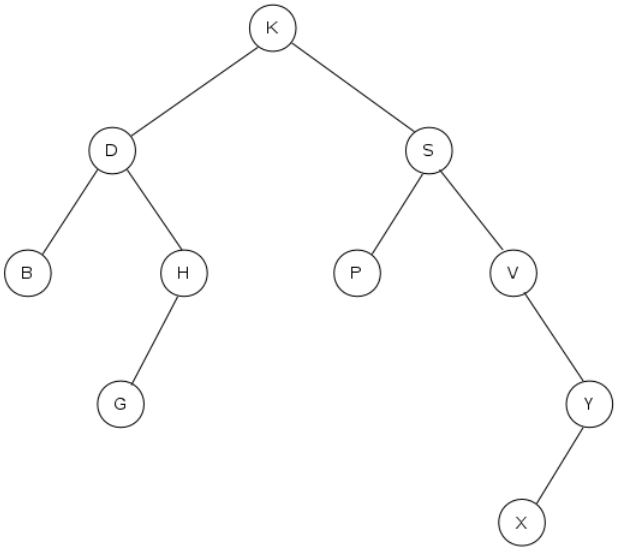
Depth(N) = Depth(P) + 1

Sometimes it is said that the line from one node to a successor is an edge. Since each node (apart from the root) has a unique predecessor (there is a unique line from the parent to the node), you can observe that the number of edges in a tree with n nodes is n-1.

Trees occurs in practice in a lot of contexts, and as the perhaps most well-known example, one can think of the file system on a hard disk where the files are stored in folders organized as a tree. Other examples include a parts list, and for example, the elements of an XML document are also organized as a tree. The question is then how to implement a tree as a collection class and it is not quite easy. To implement a general tree as described above is actually difficult and it is dealt with first in the next book. However, there are several special trees, called binary trees, and it is the subject of this chapter, including describing their main characteristics as well as demonstrating how they can be implemented in Java.

7.1 BINARY TREES

A binary tree is a tree of no more than degree 2. That means that for a given node, the number of child nodes can not exceed 2. As an example, below is shown a binary tree with 10 nodes:



A binary tree is characterized in that it is either empty or consists of a root and two sub-trees called the left and right sub-tree respectively, and where each sub-tree is a binary tree (and may in particular be empty). For example, S is root in a sub-tree, and the same applies to V. Note that the tree has 4 leaf nodes and that the height of the tree is 4 (the path from K to X is 4). Above the tree is drawn as if there is an order of its elements, but in a generally binary tree it is not the case.

It is limited which operations can be performed on a generally binary tree, indeed, it is limited what you can use a general binary tree for in addition to being used as the base class for other binary trees with more structure. I will mention four operations that are assigned to a generally binary tree. The first is an operation that determines the height of the tree. Above, I have defined the height of the tree as the largest depth of all tree leaf nodes, and correspondingly, the above binary tree has the height 4. Specifically, the height of an empty binary tree is calculated as -1 and you can then determine the height as follows

```
getHeight()
{
 if (the tree is empty) return -1
 return 1 + max(getHeight(the left subtree), getHeight(the right subtree))
}
```

Thus, the height of the tree can be determined by means of a recursive algorithm. Note that the algorithm gives the correct result – the result 0 – if the tree consists of a single node.

The other three operations concerns the traversing of a binary tree, which is an operation through which all the elements of the tree are traversed and visits (processed) in one order or another. Classically there are three ways called *in-order*, *pre-order* and *post-order*. An in-order traverse works as follows. You start in the root and then you proceed recursively according to the following pattern:

1. traverses the left sub-tree (as long as it's not empty)
2. perform the operation on the root
3. traverses the right sub-tree (as long as it is not empty)

This means that in the above tree you handle the elements in the following order:

B D G H K P S V X Y

A pre-order traverse uses the same recursive approach, but according to the following pattern:

1. perform the operation on the root
2. traverse the left sub-tree (as long as it is not empty)
3. traverse the right sub-tree (as long as it is not empty)

K D B H G S P V Y X

Finally there is post-order traversing:

1. traverse the left sub-tree (as long as it is not empty)
2. traverse the right sub-tree (as long as it is not empty)
3. perform the operation on the root

B G H D P X Y V S K

It is also possible to define other traverse sequences, but the above are the most commonly used. As an example, you can sometimes meet level order traversing, where you treat the elements on the same level from left to right:

K D S B H P V G Y X

As mentioned, it is not easy (and also perhaps not too interesting) to implement a generally binary tree, but I want to show two interfaces and two classes that implement a binary tree. A binary tree is implemented as a linked structure in the same way as the linked list, where a node must contain the data element and a reference to the left and right sub-tree respectively, and the starting point could be the following type:

```
class TreeNode<T>
{
 private T element;
 private TreeNode<T> left;
 private TreeNode<T> right;
 …
}
```

However, it is the same type as *DoubleNode<T>* except that this type denotes the nodes references *prev* and *next* instead of *left* and *right*. Since these names do not lead to significant problems regarding understanding, I will not introduce a new node type and also use *DoubleNode<T>* in conjunction with binary trees.

The interface *IBinTree<T>* defines a binary tree with the basic methods and properties and hence properties that all binary trees must have:

```
public interface IBinTree<T> extends Iterable<T>
{
 public boolean isEmpty(); // test if the tree is empty
 public int getSize(); // returns the number od elements
 public int getHeight(); // returns the height
 public int getLeafs(); // returns the number of leaf nodes
```

```
 public void clear(); // remove all elements
 public boolean contains(T elem); // test if the tree contains elem
 public Iterator<T> inOrder(); // returns an inorder iterator
 public Iterator<T> preOrder(); // returns a preorder iterator
 public Iterator<T> postOrder(); // returns a postorder iterator
}
```

In particular, you must note that there are iterators for the three traversing methods mentioned above, and then that a *contains()* method is defined, which is the central method of a binary tree. The whole purpose of a binary tree is actually to implement this method with logarithmic time complexity. Note that the interface does not define methods for inserting elements in the tree or deleting elements – they can first be defined when you have more knowledge about the tree. The interface is implemented by the class *ABinTree<T>*, but it is an abstract class as it does not implement *contains()*. On the other hand, it can implement the three iterators that are the whole purpose of the class as it happens in the same way (using a stack) independent of the current tree. To implement the three iterators, the class has inner classes for each iterator, and as an example, below is shown the class of an in-order traversing:

```
private class InorderIterator implements Iterator<T>
```

```
private IStack<DoubleNode<T>> stack = new ArrayStack();

public InorderIterator()
{
 for (DoubleNode<T> node = root; node != null; node = node.getPrev())
  stack.push(node);
}

@Override
public boolean hasNext()
{
 return !stack.isEmpty();
}

@Override
public T next()
{
 try
 {
  DoubleNode<T> node = stack.pop();
  T element = node.getElement();
  for (node = node.getNext(); node != null; node = node.getPrev())
   stack.push(node);
  return element;
 }
 catch(Exception ex)
 {
  return null;
 }
}
```

In in-order traversing, you start to go as far left as possible, and the class has a constructor where you go left as long as you can and put all the nodes you meet on a stack. The method *hasNext()* only has to test if the stack is empty. The method *next()* pops the stack, and the element at the top of the stack is the element that the iterator should return. Before you go to the node's right sub-tree, where you again pushes nodes on the stack as long as you can go left. The other two traversing iterators are, in principle, implemented in the same way, however, post-order require an extra effort, as you put the same node on the stack several times and therefore there is a need to count how many times a node has been placed on the stack. The problem is solved with a wrapper that encapsulates a node with the counter.

The interface *IBaseTree* defines two methods for finding a node in a binary tree and deleting an element, respectively:

```
public interface IBaseTree<T>
{
 public DoubleNode<T> find(T elem);
 public boolean remove(T elem);
}
```

The method *remove()* must be implemented to delete the hole sub-tree that has a particular element as root. The class *BinTree<T>* inherits *ABinTree<T>* and implements the interface *IBaseTree<T>*. Therefore, the class must also implement the method *contains()*, but for the general binary tree, it gets linear complexity, since you can do nothing but has to traverse all nodes and search for the element. The interface does not define methods for inserting elements into the tree, but the class *BinTree<T>* has a *static* method *merge()* that performs a merge of two binary trees (left and right sub-trees) about a new element that becomes root in a new binary tree. The method *merge()* can be used to build a binary tree. The class *BinTree<T>* has no particular practical interest, and its main application is the test of the traversing methods in the class *ABinTree<T>*, and then give an example of a concrete binary tree.

The class *BinTree<T>* also overrides *toString()*, which, using a static method in the class *TreePrinter*, which returns a text that visually illustrates the tree. This method has only interest in debug and testing and should not be part of a completed implementation of a binary tree. In addition, the class *TreePrinter* can generally be used to illustrate a binary tree as text.

EXERCISE 13: A GENERAL BINARY TREE

You must write a program that you can call *BinTreeTester*. The program should use the method *merge()* to construct the binary tree shown at the beginning of this section:

```
public class BinTreeTester
{
 public static void main(String[] args)
 {
  // build the tree

  show(tree);
  print(tree.inOrder());
  print(tree.preOrder());
  print(tree.postOrder());
  print(tree);

  // remove the substree with root S
```

```
  show(tree);
 }

 private static <T> void show(BinTree<T> tree)
 {
  // print the number of elements in the tree, the height of the tree and the
  // number of leafs in the tree
  // print a visual representation of the tree by calling toString()
 }

 private static <T> void print(Iterator<T> itr)
 {
  // print the elements in the tree by using the iterator itr
 }

 private static <T> void print(BinTree<T> tree)
 {
  // print the elements in the tree by using the default iterator
 }
}
```

7.2 BINARY SEARCH TREES

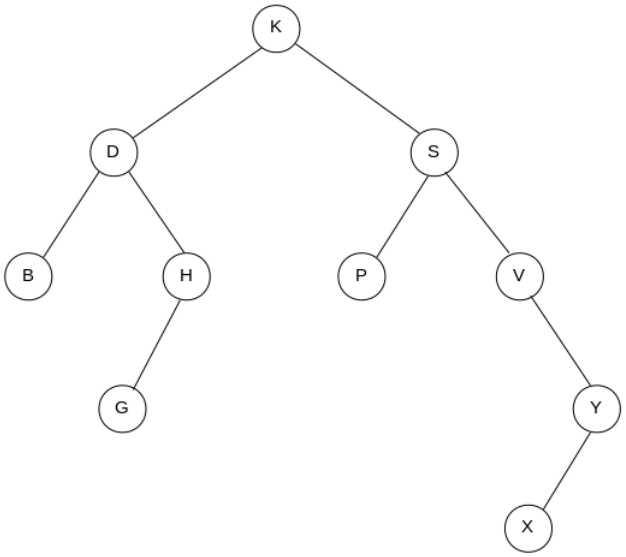
A binary search tree is a binary tree with an additional condition: For any node, its element is larger than all elements in the left sub-tree and less than all elements in the right sub-tree. If you consider the above example (beginning of section 8.1) of a binary tree, it is actually a binary search tree. In order to place elements in a binary search tree, the elements must be able to be arranged, and with Java terms, they should be comparable. I would assume that all elements in a binary search tree must be different, and in fact I have also requested it in the definition. It is not necessary and you can instead define the tree as follows: For any node, its element is greater than or equal to all elements in the left sub tree and less than all elements in the right sub tree (or: for any node its element is larger than all elements in the left sub tree and less than or equal to all elements in the right sub tree) and the implementation is primarily a matter on some places to use less than or equal to in instead of less than. However, in most practical applications, you are interested in search trees where an element may only occur once.

The binary search tree is defined as follows:

```
public interface ISearchTree<T extends Comparable<T>> extends IBinTree<T>
{
 public T getMin(); // returns the first element
 public T getMax(); // returns the last element
 public boolean insert(T elem); // inserts elem in the tree
 public boolean remove(T elem); // removes elem from the tree
}
```

A binary tree is *balanced* if it applies to any node that the left and right sub-tree "almost" has the same height. The task is to write a class *SearchTree<T>*, which inherits *ABinTree<T>* and implements the above methods, including the abstract method *contains()* from the base class. If the tree is reasonably balanced, all of these methods can be implemented with logarithmic time complexity, and that is exactly the goal of a binary tree. How to ensure that binary search tree is properly balanced is another question and is the subject of the next section, but if you think that the elements are added in random order, a binary search tree will often be quite well-balanced.

Consider the binary tree again:

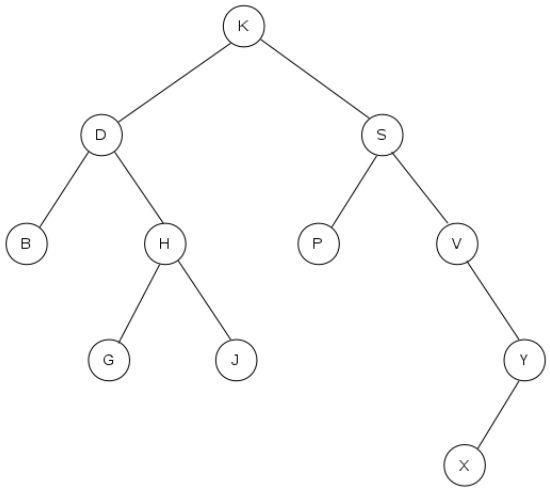


Suppose you want to search for *G*. You starts in the root, and since *G* is smaller than *K*, you must look in the left sub-tree to K. Since *G* is larger than *D*, you should look in the right sub-tree to *D*. Since *G* is smaller than *H*, you has to look for the left sub-tree to *H*. Hereafter, the element has been found. The principle is, therefore, that you start in the root, and is that the element you are looking for, you are done. Otherwise, repeat the search, but either in the left or right sub-tree. This continues until you either find the element or reach an empty tree. As with binary search, the principle is that the number of elements to be searched is halved every time, so it is an *O(log (N))* algorithm – assuming that the tree is reasonably balanced.

Inserting a new element goes a bit the same way. You start at the root and find out if the new element is to be inserted into the left or right sub-tree. If you have reached an empty sub-tree, you caninsert the element as a new child node. Otherwise, repeat the process. Again, you look down in the tree until you find the node that the new node has to be a successor to, and insertion can therefore also be implemented as an *O(log (N))* algorithm. For example, if you must insert *J* in the above tree, you get the tree below. The procedure is

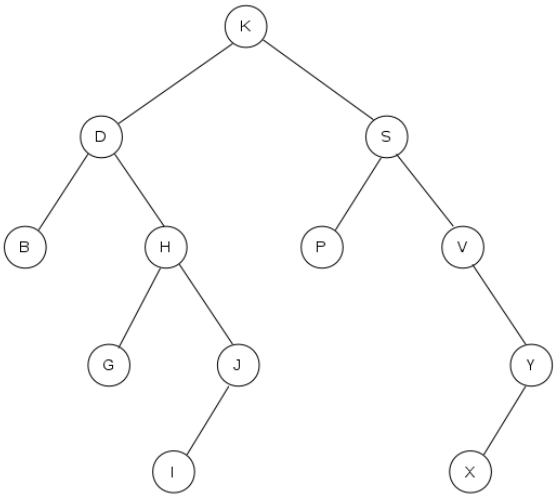
1. start in K
2. go to D
3. go to H
4. insert J as a right child to H

and the last operation corresponds to that the right sub-tree to *H* is empty, why the new element is inserted as a new leaf node to *H*.



If you next want to insert *I* you must:

1. start at K
2. go to D
3. go to H
4. go to J
5. insert I as a child to J

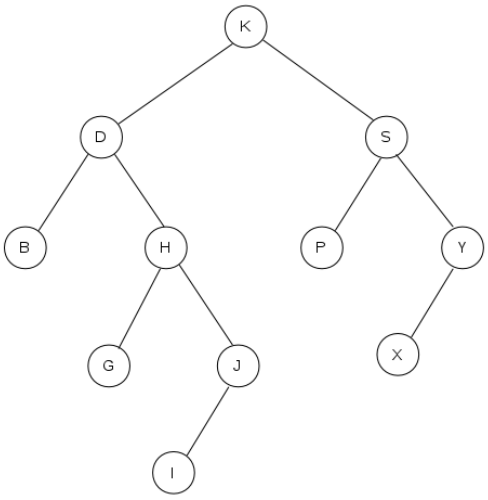


It is thus relatively simple to implement the method *insert()* and the most important thing is to note that the method can be implemented with logarithmic time complexity.

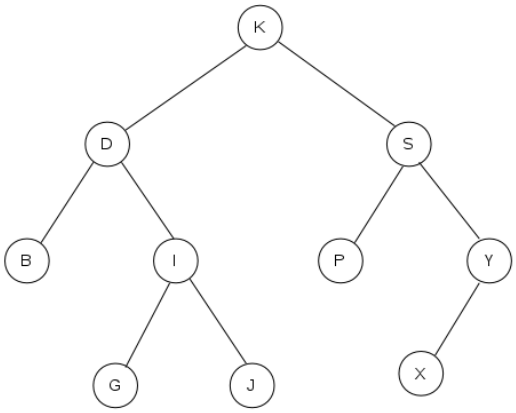
The hardest method to implement is *remove()*. You start in the root and look down the tree until you find the element to be deleted. There are now three options:

- If the element's right sub-tree is empty, the element must be replaced by the left sub-tree (for example, the elements *Y* and *X* in the above tree).
- Otherwise, if the element's left sub-tree is empty, the element must be replaced by the right sub-tree (for example, the element *V* in the above tree).
- Otherwise, if the element has both a left and a right sub-tree, it is replaced by the smallest element in the right sub-tree, and this node must then be deleted (for example, *H* in the above tree).

Again, the algorithm is searching down the tree, and it can therefore be implemented with logarithmic complexity. Below is the tree shown after deleting V:



and if you also delete *H* you get:

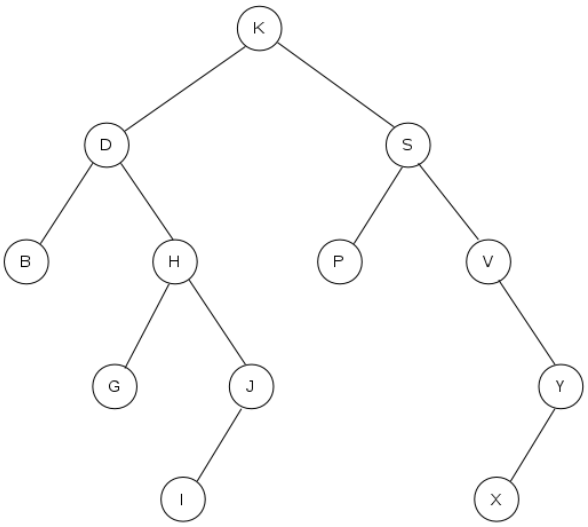


It is important to note that deletion in a binary tree retains the criterion for a binary search tree.

As shown by the interface, a binary search tree also has other services: Return the smallest and largest element. However, these methods are simple to implement and consist primarily of either walking left as far as you can, or going as far to the right as you can. When you study the code of the class *SearchTree<T>*, note that the methods *insert()* and *remove()* are both implemented recursively. They can also be implemented iterative, what I have done in the next implementation of a binary tree, which is an *AVL tree*.

EXERCISE 14: A BINARY SEARCH TREE

You must write a program that you can call *SearchTreeTester*. The program should tests the class *SearchTree<T>*. First, write a method that can create the binary tree:



That is, a tree:

```
ISearchTree<Character> tree = new SearchTree()
```

and where you add the element in the following order: K D B H G J I S P V Y X

The method must then perform the following statements:

```
print(tree.inOrder());
print(tree.preOrder());
print(tree.postOrder());
print(tree);
```

when

```
private static <T extends Comparable<T>> void print(ISearchTree<T> tree)
 throws Exception
{
 // prints the number of elements in the tree
 // prints the height of the tree
 // prints the number of leaf nodes in the tree
```

```
 // prints the smallest element in the tree
 // prints the largest element in the tree
}

private static <T> void print(Iterator<T> itr)
{
 // print the elements in tre tree
}
```

Write another test method *test2()* that performs the following:

1. creates the same tree as above
2. prints the tree using the above *print()* method
3. deletes the elements K D B H G J I S P V Y (in this order)
4. prints the tree again

Add a constant to the program:

```
private static int N = 1000000;
```

Then write a method *test3()* that performs the following:

1. creates a binary search tree for *Integer* objects
2. adds *N* random integers to the tree when the values of the numbers must be between 0 and *2N*
3. perform *N* searches in the tree on a random number between 0 and *2N*
4. as the last, the method should print:
    5. the height of the tree
    6. the number of leaf nodes
    7. the number of elements in the tree
    8. the number of searches
    9. how many of the above N searches that results in a match
    10. the number of milliseconds to execute the N searches

The result could for example be:

```
Height: 48
Leafs: 332930
1000000 elements
1000000 searches
499944 elements found
665 milli seconds
```

EXERCISE 15: LEVELORDER

Write a program that you can call *LevelProgram*. Add a class to the project, which you can call *MyBinTree* when the class must inherit *SearchTree*. The class should expand with a single method:

```
public Iterator<T> levelOrder()
{
}
```

which returns an iterator that performs a level order traversing (see above for definition of level order traversing). After writing the class, the program must create a *MyBinTree<Character>* and build the same binary tree as in the previous exercise. The program must then print the tree's elements in the level order.

7.3 BALANCED BINARY TREES

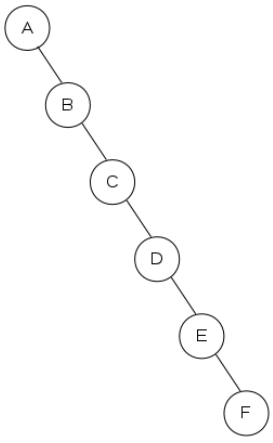
All uses of binary trees build in practice on a binary search tree (or a tree derived from a binary search tree). The tree has two very important features:

1. The search time for an element is (as mentioned) logarithmic.
2. An in-order traversing of the tree passes through the elements in ascending order.

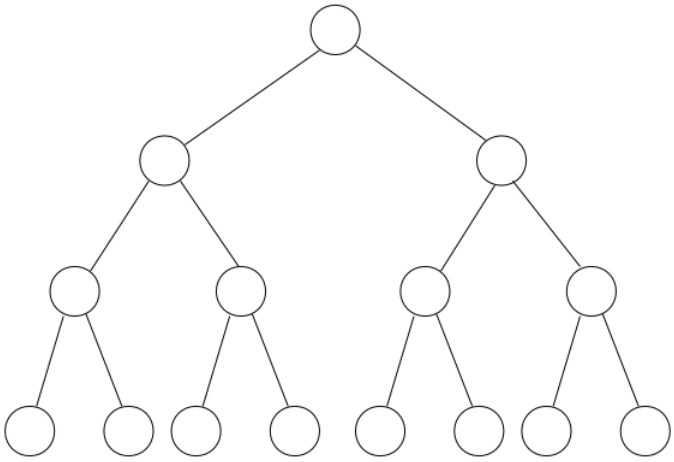
Here, however, the first one is not necessarily correct. Suppose, for example, you inserts the elements

A B C D E F

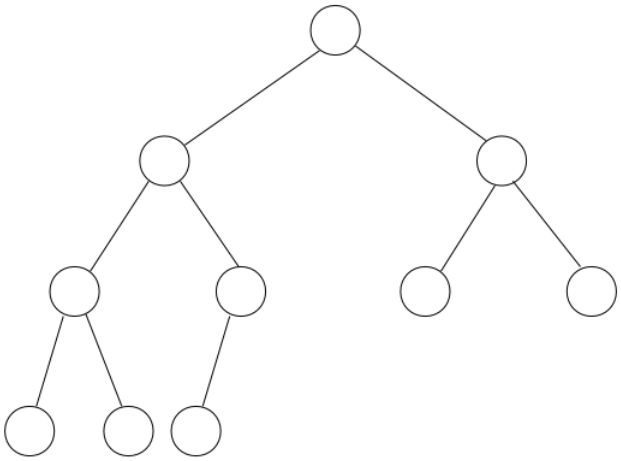
in this order. The result is still a binary search tree, but it is a degenerated tree that has the same characteristic as a list:



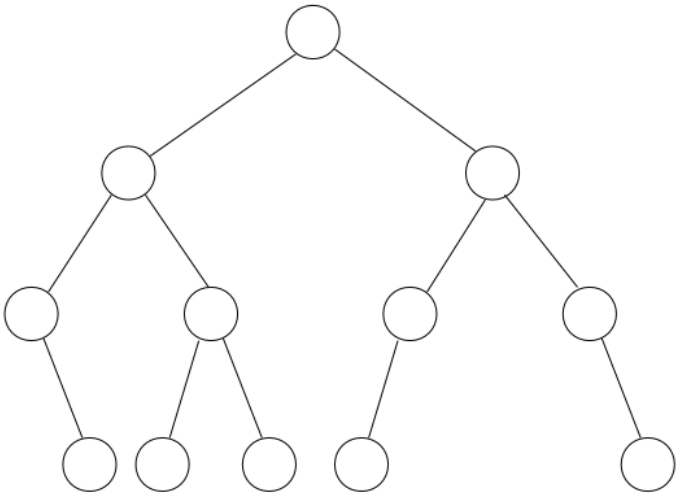
It is said that the tree is totally unbalanced. The result is for example, that *contains()* gets linear complexity, and the whole idea of a binary tree is thus lost. Therefore, it is necessary to make a few changes that ensure that the tree does not work out as illustrated above. The tree must be balanced and therefore insertion and deletion must be done in such a way that the tree after the operation is balanced. I want to show two approaches that define a specialized search tree, which is called an AVL tree, a red-black tree, where I first (in space consideration) will treat a red-black tree in the next book. But first a few more definitions. A binary tree, where all levels are filled up, is called a *full binary tree*:



If all levels near the bottom are filled up and the bottom is filled up from the left, one talks about a *complete binary tree*:



A binary tree is called balanced if it applies to any node that the difference between the number of nodes in the left sub-tree and the number of nodes in the right-sub tree is at most 1:



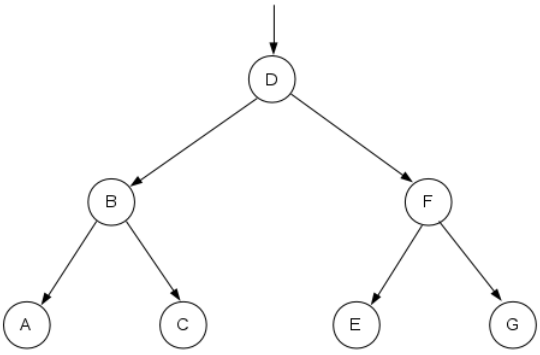
In order for the operations on a binary search tree to have logarithmic complexity, the tree must be balanced or at least fairly balanced, and in the next section I would like to show a balanced binary tree. The idea is to insert (and delete) elements as described by the binary search tree. The result may then be a binary tree that is not balanced, and the task is to restore the balance using special operations, called rotations.

7.4 AN AVL TREE

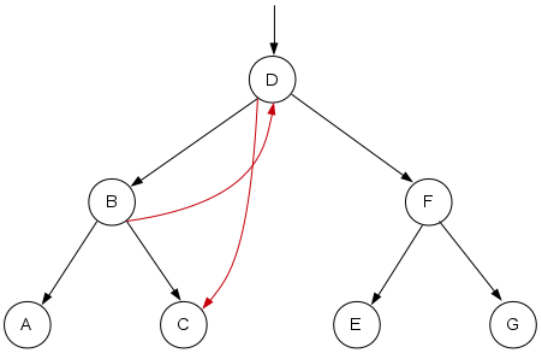
If you insert elements into a binary search tree, as outlined above, the result is not necessarily a balanced binary tree and, in the worst case, even a tree that look likes a list. It is therefore necessary to change the methods that insert and delete elements in the tree. The methods must ensure that the tree is balanced even after the operation has been completed. This is achieved through four operations, which I will now describe.

Right rotation

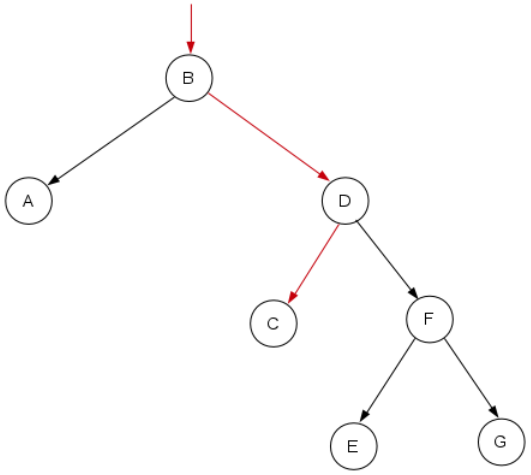
Consider the following binary tree (or sub-tree):



At a right rotation in D, one understand an operation where B becomes root (you have to change two pointers):

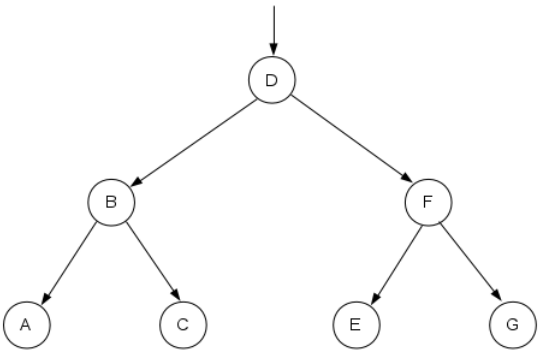


Here are the red arrows, the pointers that have been changed, and if the tree is drawn after the operation is completed, the result is:

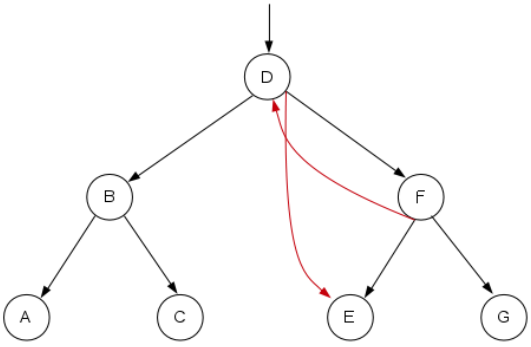


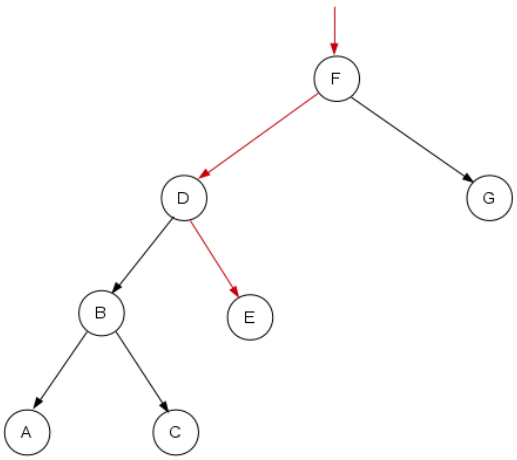
Left rotation

It is in principle the same as a right rotation, just rotate the other way. Consider again the tree:



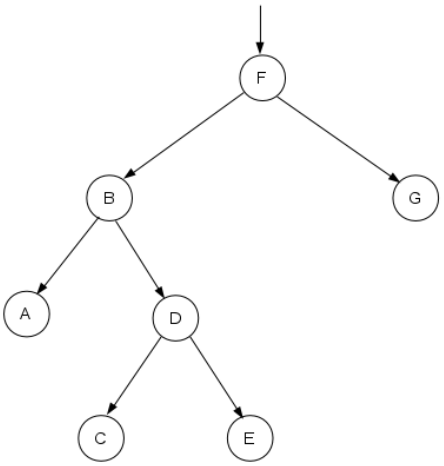
In this case, F must be a new root, and the left sub-tree for F must be the right sub-tree to D, whereas D will be left sub-tree to F:



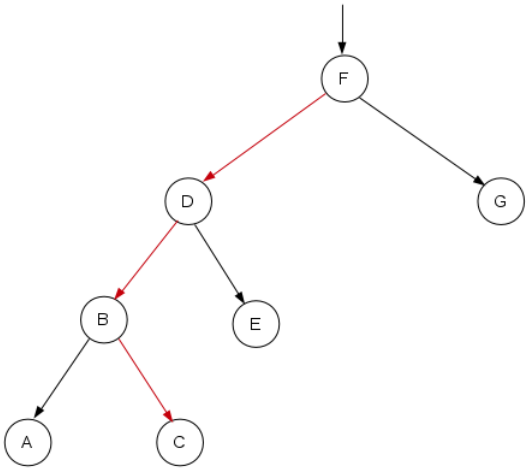


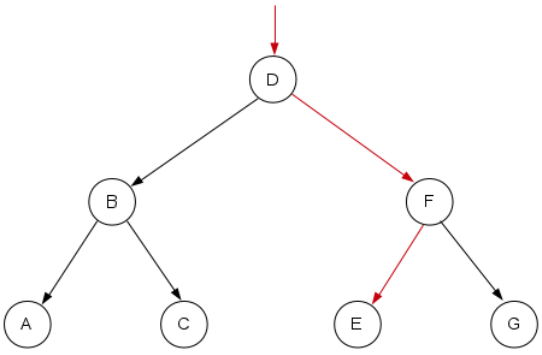
Left-right rotation

This operation is a double rotation, where first performing a left rotation of the left child of the root and then a right rotating of the root. Assume that the tree is the following and you want to perform a left-right rotation about the root F:



You start with a left rotation of B followed by a right rotation about F:



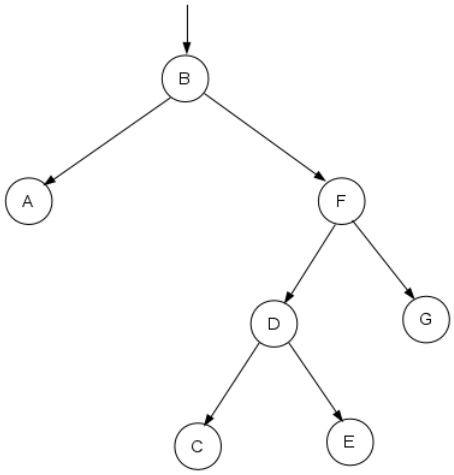


As a result, the right child to B has become root:

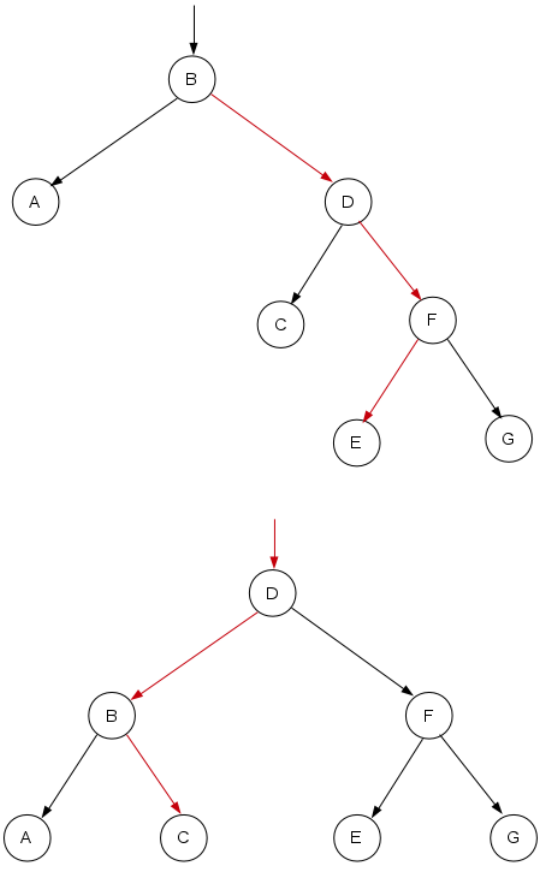
1. the first left rotation moves D one level up
2. the next right rotation further moves D up one level

Right-left rotation

The last operation is parallel to the above, where first rotates to the right of the right child of the root and then rotates to the left of the root. Suppose the tree is the following and you want to perform a right-left rotation about the root B:

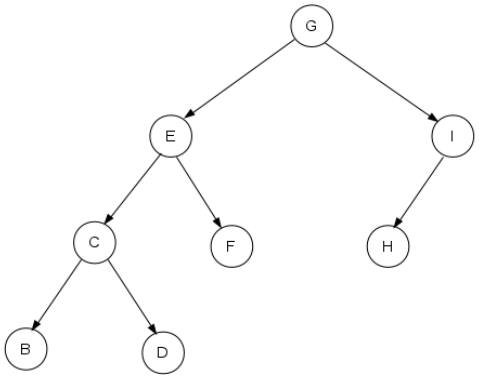


You start with a right rotation of F followed by a left rotation of B:

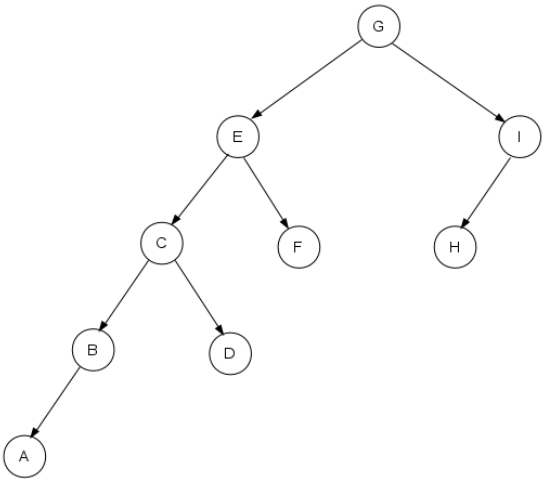


With the above operations in place, I can start to implement a balanced binary search tree. It is called an AVL tree, and it uses the above operations to ensure that the tree is balanced. An AVL tree is a binary search tree with the additional requirement that for any node is the difference between the height of the left sub-tree and the height of the right sub-tree not more than 1.

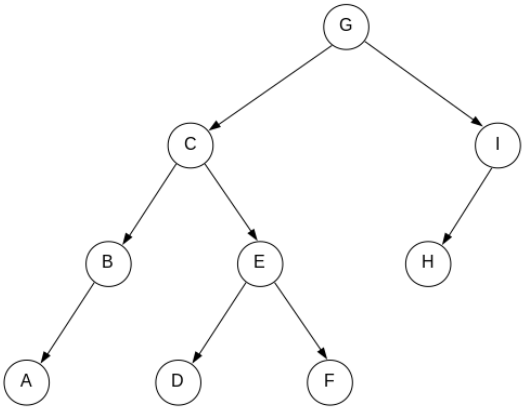
For example, the following tree is an AVL tree, but you should note that it is not a balanced tree, and the requirement for an AVL tree is thus slightly weaker than the requirement for a balanced requirement.



If you insert the element A into the above tree, you get:



but it is not an AVL tree since, according to the requirement for an AVL tree, there is an unbalance in *G* and *E*, but if you perform a right rotation about *E*, then the AVL unbalance is removed:



It shows what the purpose of rotations is, to restore the balance of an unbalanced tree. Therefore, you can implement the insertion of elements as in the regular binary search tree, but followed by a return where you test for each node on the path to the new node if the AVL condition is met and if not performing a rotation. It sounds simple, and it is, in principle, too, but there are more cases than the above example suggests, and this is where the other rotations come into play.

The algorithm is as follows:

1. You insert the new element in the same way as in the general binary search tree.
2. You go back in the tree until you reach the root.
3. If you get to an unbalance, you restore the balance using a rotation.

Initially, you should note that the insertion naturally results in a binary search tree. Once you have entered the new node, it is only in the nodes that are on its way where there may be an unbalance as only the heights of their sub-trees have changed.

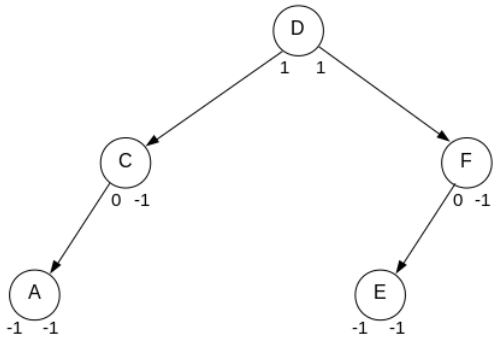
There are four options for an unbalance in a node X:

1. There is inserted in the left sub-tree for X's left child
2. There is inserted in the right sub-tree for X's left child
3. There is inserted in the left sub-tree for X's right child
4. There is inserted in the right sub-tree of X's right child

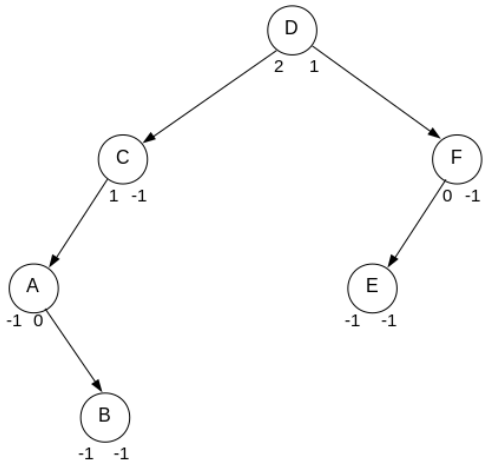
Looking at the above example, there is an unbalance in *E* after *A* is inserted into the left sub-tree to *E*'s left child *C*: The problem is solved with a right rotation in *E*.

That is, a left-left unbalance is solved with a right rotation about the node where the unbalance is detected.

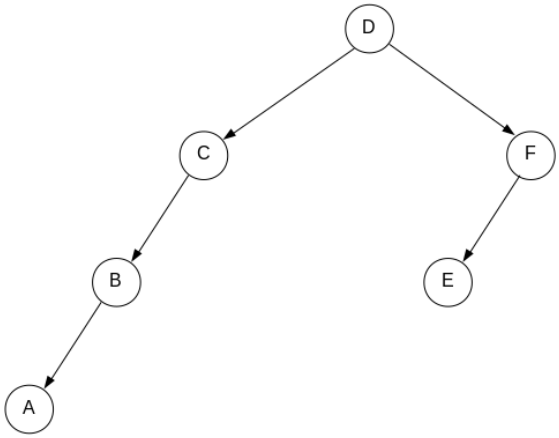
Consider the following tree, where the numbers indicate the heights of the respective sub-trees:



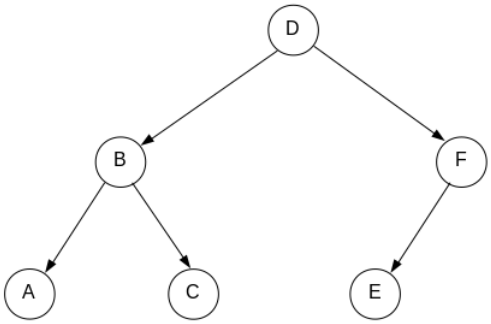
That is an AVL tree. Suppose inserting B, which is inserted as a right sub-tree to C's left child. The result is tree with an unbalance in C.



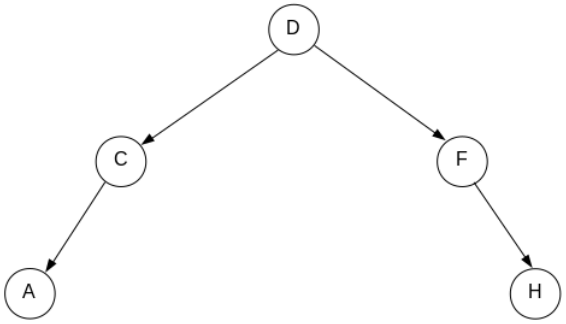
You have left-right imbalance, which is solved with a left-right rotation: Determine that there is an unbalance in C, rotate left in C's left child and right in C. Rotate left in A:



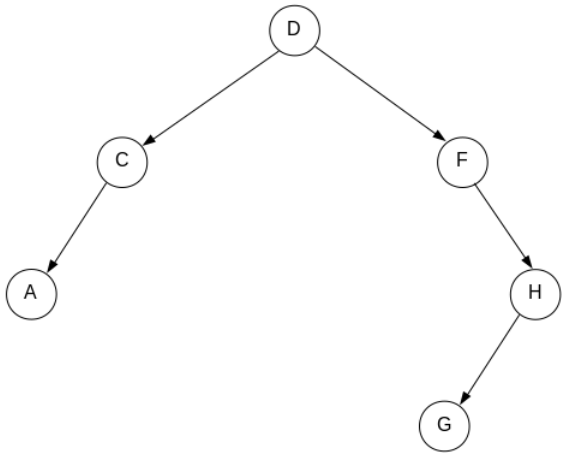
and then rotate to the right in *C*:



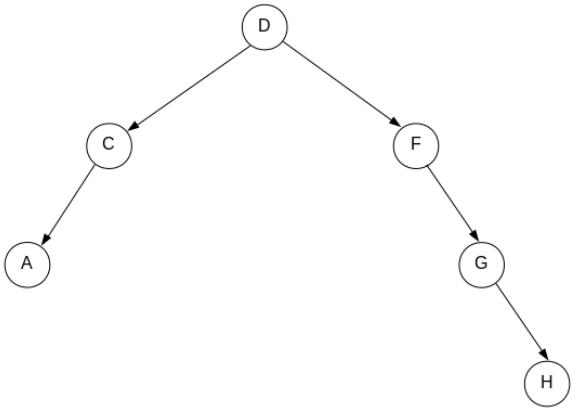
and the balance is restored. The following tree is also an AVL tree:



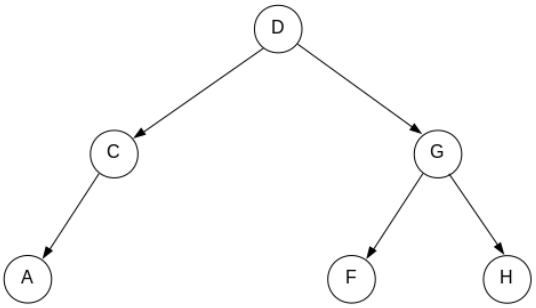
If you insert G, you get a binary search tree with an unbalance in F:



and thus are trees that are right-left unbalance. The problem is solved this time with a right-leftrotation in F. First, a right rotation in H:



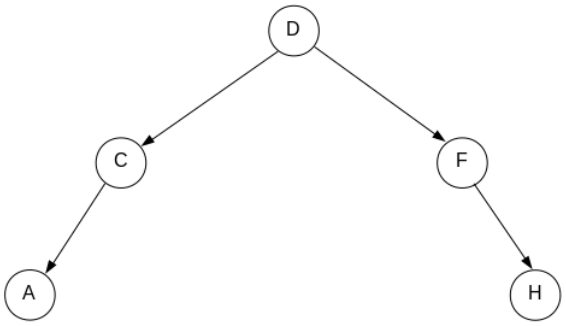
and then a left rotation in F:



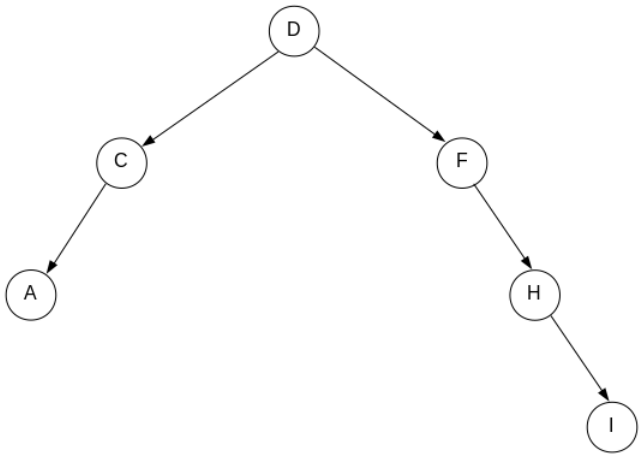
That is that a right-left unbalance is solved with a right-left rotation: Determines the unbalance in F, rotates to the right in F's right child and left in F.

The last option is a right-right unbalance where it is inserted into a right sub-tree for a right child. If the unbalance of X is detected, solve the problem with a left rotation in X.

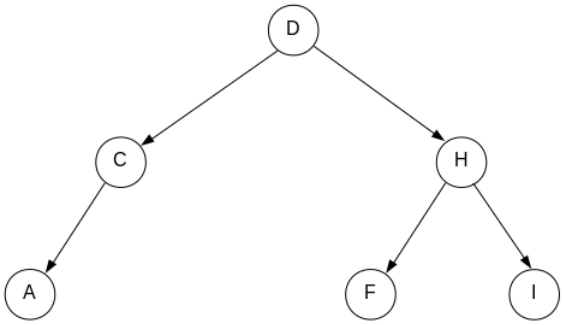
Consider again the tree:



If you insert *I*, you get an imbalance in *F*:



If you now make a left rotation in F, the tree is again in balance:



The above suggests what it takes to implement insertion into an AVL tree. You insert the item in the usual way, after which you go back to investigate whether an unbalance exists in the parent for the sub-tree into which the new node is inserted, and if so one of the four rotations, as described above, is performed after which the balance is restored. This procedure continues until you reach the root, and the result is an AVL tree.

Compared to the general binary search tree, insertion into an AVL tree is somewhat more complex, and you are encouraged to study the code, which is just a override of the corresponding method from the class *SearchTree<T>*.

Then there is deletion, where the problem is a bit the same that after deleting an element, an unbalance can be created, and it is necessary to restore the balance, but deletion is harder to implement as there are more situations to take into account. The method *remove()* deletes a specific element, and here the code is both long and complex.

Both for *insert()* and *remove()*, for each node, it is possible to determine whether there is an unbalance, and in principle it is simple because the class *SeachTree<T>* has a method *getHeight()* that returns the height of a sub-tree. The problem, however, is that it has linear time complexity, and if you test the height of sub-trees for each node on the way back, you end up with algorithms that have time complexes on *O(N log(N))* which are unfortunate. Now it is also not necessary, and instead you can attach a factor to each node, which you then adjust for all nodes on the road while looking down in the tree. The class *DoubleNode<T>* does not have such a factor, so an extension is required. When you go back to restore the balance, you need to control the parent's node's parent, and the problem can be solved by placing the individual nodes on stack, but another and easier approach is to associate a parent pointer to each node. It takes a little extra space, but makes insertion and deletion a bit easier, and you do not need to save nodes on a stack every time you insert or delete an element. For that, the class *AVLTree<T>* has the following inner class:

```
private class AVLNode<T> extends DoubleNode<T>
{
 public AVLNode<T> parent;
 private int height = 1;
```

```
 AVLNode(T elem, DoubleNode<T> prev, DoubleNode<T> next, AVLNode<T> parent)
 {
  super(elem, prev, next);
  this.parent = parent;
 }

 public int getBalance()
 {
  int leftHeight = getPrev() == null ? 0 : ((AVLNode<T>)getPrev()).height;
  int rightHeight = getNext() == null ? 0 : ((AVLNode<T>)getNext()).height;
  return rightHeight - leftHeight;
 }

 public int updateHeight()
 {
  int leftHeight = getPrev() == null ? 0 : ((AVLNode<T>)getPrev()).height;
  int rightHeight = getNext() == null ? 0 : ((AVLNode<T>)getNext()).height;
  if (leftHeight > rightHeight) height = leftHeight + 1;
  else height = rightHeight + 1;
  return height;
 }
}
```

The class has as a factor a variable *height*, and here you will notice the method *getBalance()* that returns the difference of the weights for the left and right sub-tree respectively. If this difference numerical is more than 1, it indicates an unbalance. The method *updateHeight()* update updates this factor to the largest factor for the left and right sub-tree plus 1, respectively, corresponding to that the respective sub-tree trees being changed.

However, with this class available it is, in principle, easy enough to implement the two methods *insert()* and *remove()* but many lines of code must be written and it is difficult to make sure that it are the correct references that are being changed. You should examine the code to get all the details.

EXERCISE 16: TEST THE AVLTREE

Write a program that you can call *AVLProgram*, which should test the class *AVLTree<T>* using 3 test methods. The first test method must place the letters A to Z in an *AVLTree<Character>* in random order, and then the method using the class *TreePrinter* must print the tree on the screen.

The next test method *test2()* must create the same *AVLTree<Character>*. Next, the method should in a loop print the tree, delete a random element and print what element has been deleted. The loop must be repeated until the tree is empty. The method *test2()* should do its work 10 times.

The last test method must create a *List<Integer>* with *M* random different integers. The method must then create three trees, each containing the numbers in the above list:

1. an *AVLTree<Integer>*
2. a *SearchTree<Integer>*
3. a *TreeSet<Integer>*

where the latter is one of Java's collection classes, which is actually implemented as a balanced binary tree (not an AVLTree, but a red-black tree). For each of the three trees, the method must measure how many milliseconds it takes to place the *M* elements in the tree. Subsequently, the method on each of the three trees must perform *N* searches for random numbers and measure how long the searches have taken. Finally, the method should print results as well as the height of the first two trees, and an example could be as follows:

```
M = 100000
N = 10000000
Insert elements
AVL tree                  39
SearchTree                25
SortedSet                 32
Search elements
AVL tree                2149
SearchTree              2153
SortedSet               1751
Hight of the trees
AVL tree                  19
SearchTree                37
```

Here you should especially note that the binary search tree when the elements are inserted randomly will typically be as good as the AVL tree.

8 HEAP

A heap is a data structure that in some way resembles a stack and a queue so that you can basically perform two operations

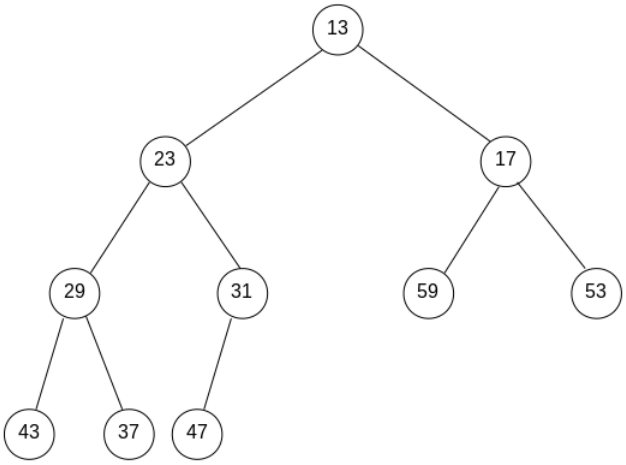
1. *add()*, which adds an element to the data structure
2. *remove()*, which returns and removes the smallest element in the data structure

From this it follows immediately that a heap is an ordered data structure so that its elements can be compared.

A heap can be organized as a binary tree that meets two additional features:

1. It is a complete binary tree.
2. Any node is less than or equal to its child nodes.

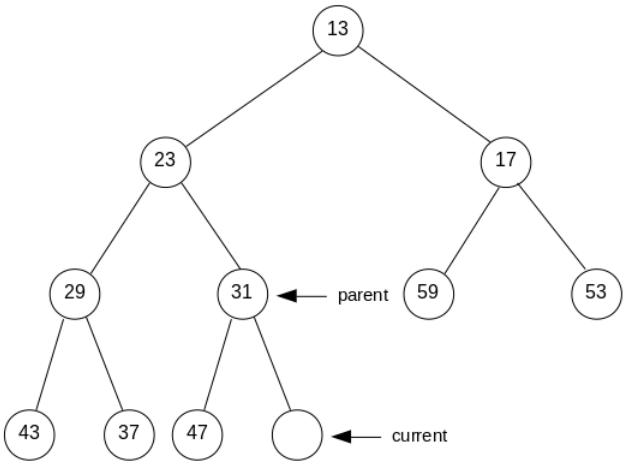
For example, the following data is organized as a heap:



Note in particular that for all nodes, its children are either empty (null) or they have a greater value than the (parent) node. It is also a completely binary tree, as level 1 is filled, while level 2 is filled out from the left. For example, if 47 had been a right child to 31, it would not be a complete binary tree.

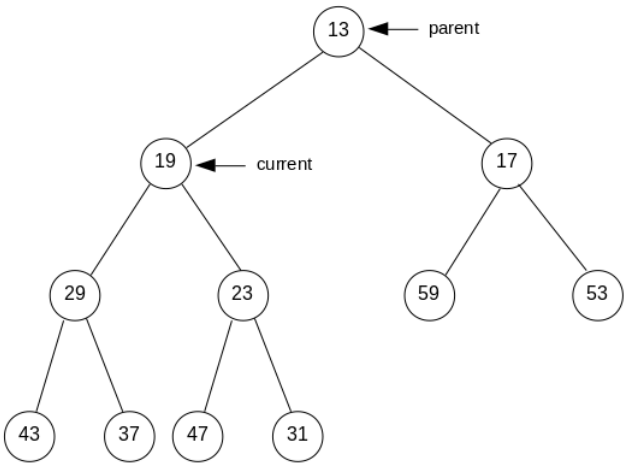
As a heap in that way organizes the elements like a binary tree, you sometimes also talk about a binary heap.

If you want to insert a new element into a heap, you expand the tree with a new node:

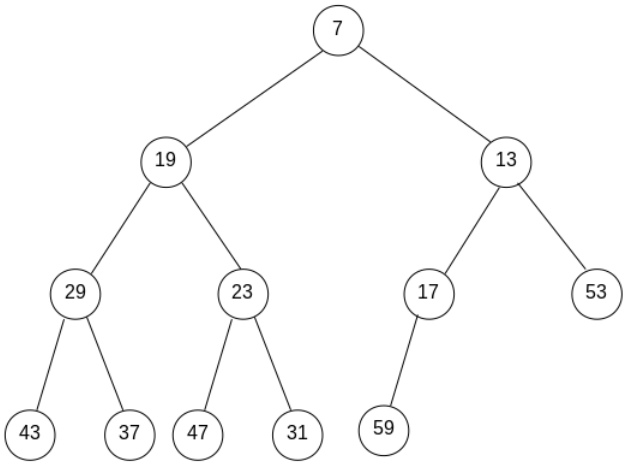


Note that it is still a complete tree. You start now in the new node called *current* and go up in the tree. For each step you look at parent. If it is larger than the new element to be inserted, the value of parent to *current* is exchanged, that is that current move one level up. When the value of parent is less than or equal to the new value, the process stops and

the new value is stored where *current* points. For example, assume that the new element is 19. After the elements are moved and 19 are inserted into *current*, the picture is as follows:



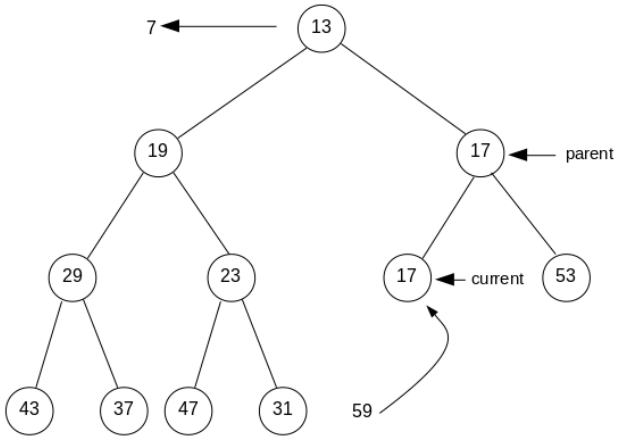
Below is how the heap looks like if you also inserts the element 7:



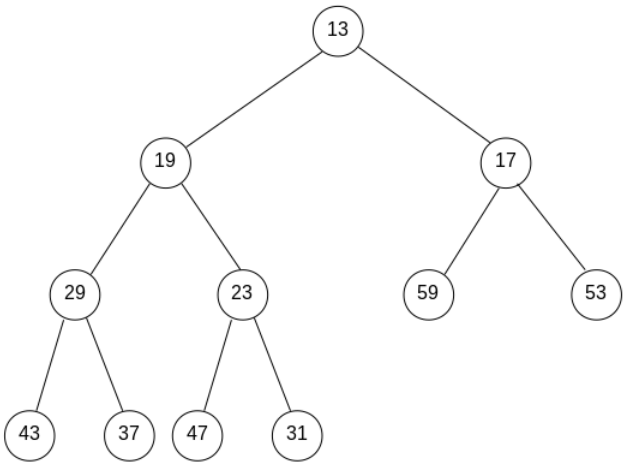
Note that, of course, the lower level may be filled up and you have a full binary tree. If that is the case, and if you want to insert another element, the tree will get another level, but as it still has to be a complete binary tree, you must start from the left with a new left child to the node at the far left of the previous level. Also note that after inserting a new node, you have to look up the tree to find the node where the new element must be placed. In the worst case, you have to go back to the root, but if you get it, the new element must be less than the root, and thus less than the children of the root, and the element can thus be inserted as a new root. This means that at worst you have to go back to the root, which means that the time complexity for inserting a new element is *O(log (N))*.

The result of inserting elements into a heap is, as shown above, that the smallest element is always in root, and it is therefore simple to implement *remove()* as it simply returns the value of the root. However, the problem is to ensure that the tree after removing the root is still a heap and it requires a process resembling insertion.

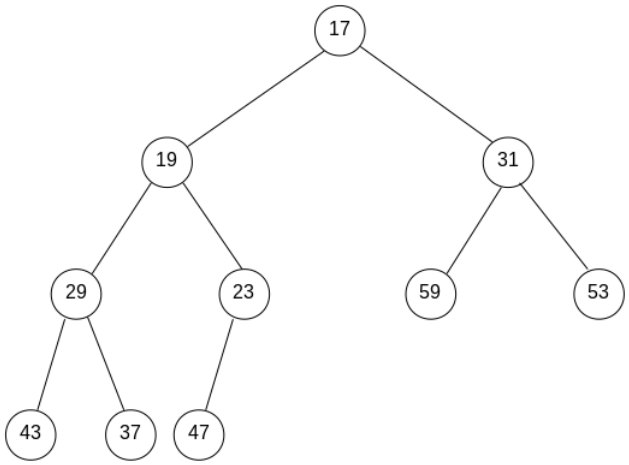
In the example above, *remove()* returns 7, and the node with 59 will be gone. You start at the root and go to the least child, here's 13. Since it's less than 59, it moves to the root. You are now repeating the process and going to the least child to 13, which is 17. Since it is less than 59, 17 moves a place up (where the parent points):



Since there are no children for *current* (the node with the value 17) 59 are put in that place:



The result is still a heap: It is a completely binary tree, and the value of each node is less than or equal to the value of the children (if they are not null). If you delete again, 13 are disappearing and this time it is 31 to be put in place:



and the result is again a heap.

When deleting, you start in the root and move down until you find the place where the value of the node to be deleted must be inserted. At worst, you have to go to the bottom of the tree, and the time complexity is therefore *O(log (N))*. The result is that both *add()* and *remove()* have logarithmic time complexity.

A heap also has operations other than *add()* and *remove()*, and as typical examples include:

1. *peek()*, which returns the smallest element (without removing it)
2. *clear()* which deletes all the content of the heap
3. *getSize()*, which returns the number of elements on the heap

You can also implement an iterator so you can traverse the heap, and it can be done in several ways as for the general binary tree, but a level order traverse is probably the natural choice. Correspondingly, I will define a heap as follows:

```
public interface IHeap<T extends Comparable<T>> extends Iterable<T>
{
 public void add(T elem);
 public T remove();
 public T peek();
 public void clear();
 public int getSize();
 public boolean isEmpty();
 public boolean contains(T elem);
 public boolean isMinHeap();
}
```

The methods should be self explanatory except the last. Above I have defined a heap like a binary tree with the property that every node is less than or equal to its children. Such a heap is sometimes called for a *minimum heap*. You can of course also define a *maximum heap* as a heap where each node is larger than or equal to its children, and the last method in the above interface should indicate which order of the elements to apply to the heap.

PROBLEM 4: A TREEHEAP

You must write a program that you can call *HeapProgram*. The program must have a class *TreeHeap<T>* that implements the above interface for a heap when it should happen as a binary tree following the guidelines described above. You can advantageously define the individual nodes with the following inner class:

```
private class Node<T>
{
 public T elem;
 public Node<T> left = null;
 public Node<T> right = null;
 public Node<T> parent = null;
 public Node(T elem)
 {
  this.elem = elem;
 }
}
```

where *left* refers to the left sub-tree, *right* to the right sub-tree and *parent* to the predecessor. The class can then be defined as follows:

```
public class TreeHeap<T extends Comparable<T>> implements Heap<T>
{
 private Node<T> root = null; // pointer to root
 private Node<T> last = null; // pointer to the last node created
 private int count = 0; // number of elements in the
 private boolean minHeap = true; // if true it is a min heap and else a max
heap
```

Remember that you also write test methods that can test your class.

8.1 THE CLASS HEAP

If you've solved the above problem, you've implemented a heap where both *add()* and *remove()* have logarithmic time complexity, which is quite good, but you can do better using an array.

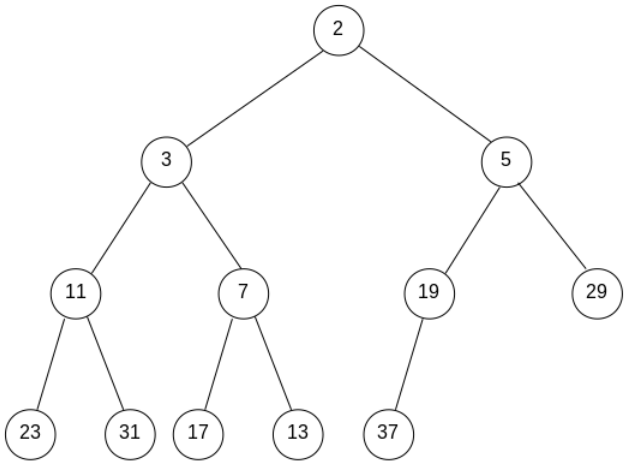
A heap is a completely binary tree. You can now place the root in place 1 and the root's children in places 2 and 3. Then you can place the root's grandchildren in places 4, 5, 6 and 7 and the next level (where there are 8 nodes) can then be placed on places 8, 9, 10, 11, 12, 13, 14 and 15. That is, that the elements of the tree are placed in the tree arranged

by level, and since it is a complete tree, it means that the tree has no holes. Put a little differently, you can calculates the indices the tree's nodes from the following formulas:

$$Index(root) = 1$$

$$\forall\ node \in heap : \begin{cases} Index(node.Left) = 2 * Index(node) \\ Index(node.Right) = 2 * Index(node) + 1 \\ Index(node.Parent) = Index(node) / 2 \end{cases}$$

Consider, for example, the following complete binary tree:



In fact, it's a heap. There are four levels where only the three are filled out and if you implement this tree as an array you get the result:

	2	3	5	11	7	19	29	23	31	17	13	37							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

The idea is that, based on the above formulas, you can directly calculate where an element's parent and children are. For example:

1. The right child to 3 is on plase 2 * 2 + 1 = 5
2. The left child to 5 is on place 2 * 3 = 6
3. Parent to 13 is on place 11 / 2 = 5

By implementing a complete binary tree this way, it is easy to implement a heap. It is done in the class *Heap*.

Looking at the above figures and the class *Heap*, you can see that I do not use the first place in the array. The reason is that both the implementation and indexing in the array will be

simpler thus resulting in a more efficient implementation. Of course, you can do that and instead start with the root in place 0, but it gives more complex index calculations and, in principle, a less effective code.

8.2 HEAP SORT

If you inserts the elements in an array in a heap and subsequently empty the heap, the result will be that the array has be sorted. You can therefore write a sorting algorithm as follows:

```
public static <T extends Comparable<T>> void heapSort(T[] t)
{
 Heap<T> heap = new ArrayHeap<T>(true, t);
 try
 {
  for (int i = 0; !heap.isEmpty(); ++i) t[i] = heap.remove();
 }
 catch (Exception ex)
 {
 }
}
```

The method is implemented in the class *Tools*.

Since both insertion and deletion in a heap have logarithmic complexity, the complexity of the algorithm is *O(N log (N))* and thus the same as for example quick sort. The method, however, has primarily theoretical interest. Firstly, time measurements will show that the method in most cases is less efficient than quick sort and then it should be noted that the memory complexity is not good as it is based on a heap and thus a copy of the array, that is on the same way as merge sort.

8.3 A PRIORITY QUEUE

As a last use of a heap, I would like to mention a priority queue. I have previously mentioned the implementation of a queue and in connection therewith also in a problem implementation of a priority queue as an array of queues where the number of queues was previously defined as a parameter for the constructor. If you consider a heap where the element's order is determined by the priority, then removing the elements from the heap will always get the element with the highest priority (the element with the least value for priority). The result is, that you have a priority queue. You can therefore very easily implement

a priority queue as a thin enclosure of a heap. An example is the class *PriorityQueue<T>*, which implements the following interface:

```
public interface IPriorityQueue<T> extends IQueue<T>
{
 public void enqueue(T elem, int p);
}
```

EXERCISE 17: TEST THE CLASS HEAP

You must write a program that you can call *HeapTester* when the program has three test methods that can test the class *Heap<T>*.

The first method must create a heap, where it randomly inserts the uppercase letters from A to Z. Next, the method must print the contents of the heap using an iterator, after which it should remove all elements from the heap and print the content again. The result could be:

```
ACBDFELTHNMGJYQXVSKWPZUOIR
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The next method must create an integer array of *N* elements and initialize the array with random integers. Next, the method must create a copy of this array, after which it must measure the time to sort the two arrays with Java's sort method and *heapSort()* respectively. Measured the time in milliseconds, the result could be:

```
10000000 elements
Arrays.sort(): 4190
Tools.heapSort(): 10318
```

The last method should test the class *PriorityQueue<T>*. The method must start by placing 10 random 3-digit integers in a priority queue when the priority must be the cross-sum of the number. Next, the method must inserts the numbers 2, 3, 5 and 7 in the queue, but without specifying a priority. Then again, 10 random 3-digit numbers must be added, where the priority is again the cross-sum. Finally, the method should remove all numbers from the queue and print the result, which could be:

```
298 839 686 822 534 660 462 652 733 391 923 626
771 619 547 817 116 504 486 270 3 5 7 2
```

PROBLEM 5: RADIX-SORT

Radix-sort is a sorting method that can be used to sort strings of the same length (think of the sorting of keys) and which in principle has linear time complexity. Suppose, for example, that you have 3-character keys that consist of digits, and an example might be the following:

```
235 681 925 105 311 229
```

The idea is now to create a queue for each character that can occur, and in this case it is 10. Next, you pass the array so that you place the numbers in the queues after the last digit, after which the numbers are moved back to the array by removing all elements from the queues. Then you repeat the process, but this time you place the numbers in the queues after the next last digit and continue until all digit positions are used. Then the array is sorted. With the above example, 3 iterations must be used (the key length is 3) and after the 1. iteration is the result:

```
Queue 0
Queue 1 681 311
Queue 2
Queue 3
Queue 4
Queue 5 235 925 105
Queue 6
```

```
Queue 7
Queue 8
Queue 9 229

681 311 235 925 105 229
```

Next, the process is repeated, but where the numbers are placed in the queues after the middle digit:

```
Queue 0 105
Queue 1 311
Queue 2 925 229
Queue 3 235
Queue 4
Queue 5
Queue 6
Queue 7
Queue 8 681
Queue 9

105 311 925 229 235 681
```

The last iteration places the numbers in the queues after the first digit:

```
Queue 0
Queue 1 105
Queue 2 229 235
Queue 3 311
Queue 4
Queue 5
Queue 6 681
Queue 7
Queue 8
Queue 9 925

105 229 235 311 681 925
```

after which the array is sorted. You should note that the method assumes that the keys have the same length and that the method basically only works for strings, but the result is that you can sort an array using a number of queues – or by using a priority queue. Each iteration consists of a pass of the array, where the elements are distributed on the queues, which can be performed in linear time. The number of iterations is determined by the key length, and if it is the M and if there are N elements in the array, the method can be implemented as an *O(MN)* algorithm. Since M is constant and in practice is small, one can therefore perceive the algorithm as a linear time algorithm.

You must write a program that you can call *RadixProgram*, which should sort an array of strings that represents fixed-length keys and where the individual characters are digits. The program must sort the array using radix-sort when using the class *PriorityQueue*<*String*>.

When you tests the program, try comparing with, for example, quick sort and measure how long it takes to sort an array. Although quick sort has a poorer time complexity a radix-sort, you'll probably find that quick sort wins. The reason is that radix-sort simply moves too much around the numbers.

9 HASHING

A collection class is a container of objects, and there are primarily two things that separate collection classes:

1. what services the class makes available, and thus what the users can do with the class
2. how the objects are internally stored and organized

The latter is only an internal class property, and it is basically the intention that users should not (need to) be aware of it. Users need to know the complexity of the individual methods, but whatever else happens, it is left to who implements the class to take care of.

Until this place I have mentioned two ways to store objects. It happens either by means of an array or by linking the elements together with references or pointers. Both ways have their pros and cons, and the choice will depend on the characteristics you want that collection to have. In fact, it is the only two ways you have to store objects, but in both cases it is possible to organize the objects appropriately, which is important for class' methods. For example, an *ArrayList* stores the objects in an array, but is organized in such a way that they are stored from the start and then to a last element. The *ArrayList* has an internal counter which it uses to keep track of where the last element is.

What characterizes an array as a container is that each object is located on a particular index, and with this index in hand you have direct access to the element with constant time complexity. An array is a highly efficient container, but it is up to the user (the collection class) to decide which index to use to find a particular element. The problem with it is that in many contexts (most) it is not very appropriate to refer to a particular object via a numerical index that directly corresponds to the physical location, but it would be better to find an object based on for example, a name (a key), but without it should be in the form of a search. The solution to that problem may be *hashing*.

Technically, a hashed structure is an array of objects that identify the individual objects using a key rather than an index. The idea is that the key is converted to an index, which indicates where in the array the object is located. It is this conversion that is called *hashing*. That is, when finding an object in the array, it does not happen with searching, but by calculating where the object is and thus by an operation that has constant time complexity.

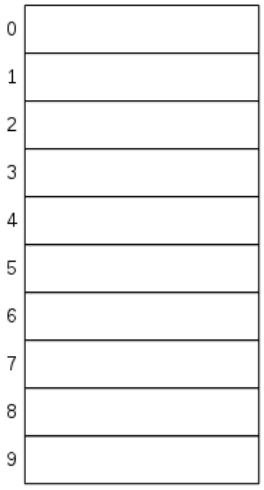
To illustrate the principle, I will look at an example where I want to save Danish zip codes in an array. An example of an object is

```
7800, Skive
```

Suppose that the starting point is an array of 10 places (see below). As mentioned above, I will use the zip code as a key, and the first task is to convert the zip code to an index. I will use the following function:

```
index = digetsum(code) % 10;
```

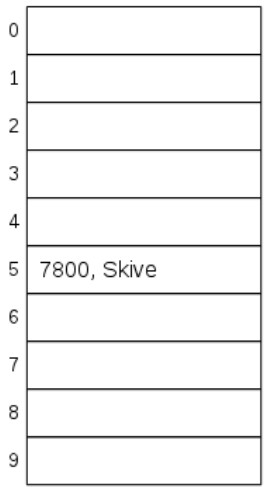
where *digetsum()* is a function that determines the sum of the digits in a zip code. The 10's come from the array size, and the value of index will therefore always be one of the numbers 0, 1, 2, …, 9 and thus a legal index. The method *digetsum()* is what is called a hash function and you should note that it is easy to implement this function in java regardless of whether the type of code is a *String* or an *int*.



If the key is *7800*, the zip code object must be placed in place

```
(7 + 8 + 0 + 0)%10 = 5
```

and below I have shown the array after *7800, Skive* is inserted:



Below I have shown the table after I have added another three objects:

8800, Viborg
7950, Erslev
8000, Aarhus C

and the important thing is to note that each time the object's place is found using a calculation and not a search:

0	
1	7950, Erslev
2	
3	
4	
5	7800, Skive
6	8800, Viborg
7	
8	8000, Aarhus C
9	

Suppose you want to further insert the object

7900, Nykøbing

The element should be placed in place 6, but it is busy and we are talking about a collision. There are two objects with different keys, which by the hash function determines the same location in the table. You can therefore conclude that there are two basic challenges to implement hashing of an array:

1. A hash function must be defined in a way that distributes the keys evenly across the array.
2. There must be a solution to the collision problem.

In terms of the hash function, it is in principle a question about that you have an array of a given size *n* and a collection of objects of one kind or another. If you have an object obj, the hash function *f* must from the object *obj* calculate an *int f(obj)* which is called the hash value or the hash code. You can then determine the object's index in the array as i = *f (obj) mod n*, which just specifies a value within the array boundaries.

The function *f* must of course be effective, but otherwise the most important feature is that the indexes are evenly distributed across the array. If this is not the case, the objects will clump into one or more locations in the array, which means that parts of the array will not be used and that the risk of collisions will increase.

There are many examples of good hash functions defined for numbers and strings, and they will typically work by looking at the individual characters as above with the zip code example and for numerical values by some form of calculation. Often the key is a string, or strings are included in the key. A hash function that works on a string therefore needs to convert a string to an integer, which typically takes place using the numerical codes for the individual characters. The calculation could, for example, take place as follows:

Let n denotes the number of characters in the string and let s_i be the character code for the character on position i. Then

$$\sum_{i}^{n-1} s_i * 31^{n-1-i}$$

is an example of a hash function that calculates the hash code for a string. If, for a given string, modulus is taken with the array length, the string is converted to an array index. The above hash function is simple to implement (and understand – the 31 are because it is a prime) and it is the function used in Java. I have shown the function here as an example, and that it's easy to write a hash function that works on strings.

As is known, the class *Object* has a method *hashCode()*, which returns a hash value for an object. It returns the object reference which can not be used to much (usually you need a code with value semantics), but the method can be overridden. The method is usually override together with *equals()* (overrides one of them and not the other you gets a warning), and the only requirement is that if two objects are *equal()*, they must also have the same hash code. Conversely, the reverse is not a requirement, and you may have two objects that have the same hash code but are not *equal()*.

hashCode() is a hash function, and the result is that each type in Java has its own hash function, but for custom types it is left to the programmer to implement the function so that you get a good hash function with value semantics. For most Java classes, including the class *String*, it is done, and it is not necessary to do any further. For example, *hashCode()* for an *Integer* just returns the number itself, while *hashCode()* of a *Long* returns a XOR of the top 32 bits with the last 32 bits. If *n* is the value a *Long*, the hash code is determined as:

```
(int)(n ^ (n >> 32))
```

Then there is the collision problem. In general, there will always be more possible keys than there are places in the array that will contain the objects. Therefore, collision will always occur in practice. The whole idea of hashing is that from the key you can find the location of an object with constant time complexity. To succeed, there must be few collisions and short collision chains. It is therefore a part of implementing a collection using hashing also to solve the collision problem. I want to mention four ways.

The first one is using an overlay area. To the data structure, a list of the objects that resulted in a collision is linked. This solution means that you can still insert objects into the data structure with constant time complexity, whether collision or not. If you add a new object and the result is a collision, then the object is added to the overlay area, and since both operations have constant time complexity, the time of the total operation is constant. The disadvantage is of course a growth in an access of an element in the data structure. Basically, the location is calculated, and if you do not find the object with the correct key, you must search the overlay area. If there are many collisions, the number of objects in the overlay area can be large, and the access time for an element will then have linear time complexity. It is simple to solve the collision problem using an overlay area, but conversely the access time can be long. However, it is worth paying attention to the technique, as it is actually a modified form that you typically use in practice.

Another simple way is to use the next available space, a method that is also called linear probing. If an object is to be inserted and the result is a collision, you check the next position and if this position are available, place the element there. If it is not available, try

the next again and continue until you find a free space. If you in the above table with zip codes want to insert

7900, Nykøbing

you get the result:

0	
1	7950, Erslev
2	
3	
4	
5	7800, Skive
6	8800, Viborg
7	7900, Nykøbing
8	8000, Aarhus C
9	

If you further add

6000, Kolding

it must also be placed in place 6 with collision as a consequence. Then try to place 7, then place 8 and until you find a free space in place 9. The advantage of linear probing is partly that the principle is easy to implement and that there is no need for any extra data structure for the overlay area. On the other hand, there are also some significant disadvantages. If the array is almost filled up, a collision can lead to many searches before finding a free space, and the same is true of access a particular element. Another problem is that it is difficult to delete an object in the data structure as it creates holes in the collision chains. Deletion must therefore be implemented by mark an object that has been deleted but without physically removing the element and releasing the space. It must then be combined with a form of rehashing, and reorganizing the data structure and thus also releasing the places used for deleted objects.

Studies of linear probing have also shown that the method tends to form clusters, where the keys refer to groups of indices while there are other groups of indices rarely used. It's unfortunate as it means to increase the number of collisions. However, one can improve the method by using square probing instead. The principle is the same as linear probing, where objects resulting in collisions are stored in the same array as the primary objects. The difference is that the places are not chosen by constantly taking the next one, but by taking the indexes from the following sequence:

$$i = (h + k^2)\ mod\ n\ k = 0,1,2,3,\ldots$$

where h is the primary hash index (the hash function's value in the key) and n is the array size. That is, if there is a collision (space h is busy), try $h + 1$, then place $h + 4$, then space $h + 9$ and so on. Studies shows that square probing is better than linear probing and does not lead to too many collisions, but it is not obvious that the above indices finds a free space, even if the array is not full and does not go into a ring. One can show that if the array size is a prime number, and if the fill rate (the proportion of array places used) does not exceed 50%, quadratic probing will always provide a solution. Quadratic probing is much better than linear probing, but conversely, the problems are also the same, and especially the issues of remove elements can cause holes in the collision chains. Finally, quadratic probing is not quite simple to implement.

As mentioned above, you can solve the collision problem using an overlay area. A very natural improvement is to tie an overlay area to each place in the array. In this way, each element has its own collision chain, which naturally becomes shorter, and if you have a good

hash function that distributes the objects evenly, all collision chains in the middle will be equal. Immediately, it may sound like an expensive solution, since each space in the array must be associated with a list (or similar structure) and it is also *true,* and compared to square probing, for example, this solution requires extra memory, but it is offset by that the solution is simple to implement. In Java, you have collection classes like *HashSet<T>* and *HashMap<K, T>*, that are based on hashing. They solve the collision problem just because each place has its own collision chain.

A hashed structure is internally an array and therefore it also has a capacity. If you save more objects than there is capacity for, the result will inevitably be collisions. Conversely, it is also clear that if almost all the places in the array are available, the likelihood of collision is very small. The question is, therefore, how full the array must be before collisions become a problem, and in other words, how much the fill rate must be. Regardless of what, it is necessary to extend the capacity of the array if the fill rate exceeds a given value. Here, as I have looked at in connection with an *ArrayList*, you can use array doubling so that if the fill rate exceeds the maximum set, you can double the array's capacity. However, it is not just a matter of creating a new array and copying the old one as a rehashing of all objects is required, as the size of the array is included in the index calculation. It also means that in the case of hashed structures, you can never assume anything about the physical position of the individual objects in the array. It also means that you can not assume anything about the objects order if you traverse the content of a hashed collection.

9.1 THE CLASS HASHING

I want to start with a data structure that stores objects in an array using hashing. It's a simple data structure, where you basically can do three things:

1. you can add (save) an object
2. you can remove an object
3. you can ask if an object exists in the data structure

The class is defined as follows:

```
public interface IHash<T> extends Iterable<T>
{
 public static final String FillError = "Illegal fill degree";
 public int getSize();
 public boolean contains(T elem);
 public boolean insert(T elem);
 public boolean remove(T elem);
```

```
 public T get(T elem);
 public void clear();
 public double getFill();
 public void setFill(double fill) throws PaException;
}
```

and here the methods should be self explanatory, perhaps not the last two, which respectively read and change the fill rate as a value between 0 and 1. The interface is implemented by the class *Hashing*:

```
public class Hash<T> implements IHash<T>
{
 protected static final int N = 11; // the default capacity
 protected double fill = 0.5;       // the fill degree
 protected IList<T>[] array;        // the internal array to elements
 protected int count = 0;           // the number of elements
```

The first constant indicates the size of the internal array. Here it is set to 11, but for practical use, it is reasonable to choose a larger starting value. The next variable is the fill rate, which is set to 0.5 corresponding to a fill rate of 50%. It is set low to reduce the number of collisions, and in practice I often choose the value to 0.8. Properties are attached to the

variable so it can be changed if you want to experiment with the fill rate. Next, it follows the internal array, which is an array of lists. It is a little expensive with a list for each entry in the array, especially if there is a low fill rate, and there are few collisions, and therefore only a list should be linked to the entries where there is an element. The last variable is a simple counter that keeps track of the number of elements in the data structure.

As example, I will show the method *insert()* that adds an object to the data structure:

```
public boolean insert(T elem)
{
 int n = index(elem);
 if (array[n].contains(elem)) return false;
 if (count + 1 > array.length * fill)
 {
  reHash();
  n = index(elem);
 }
 array[n].add(elem);
 ++count;
 return true;
}
```

It uses a help method *index()*, which returns the object's place in the array:

```
protected int index(T elem)
{
 return Math.abs(elem.hashCode()) % array.length;
}
```

It uses *hashCode()*, and it is therefore a requirement that the parameter type *T* implements *hashCode()* with value semantics and thus also *equals()*. Also note that I'm using the absolute value because you can not be sure that the value of *hashCode()* is non-negative. Finally, note that the method returns modulus of the hash code, so the result is an index within the array boundaries. After the method *insert()* has determined the index, it tests whether the data structure already contains the object, and if so, the operation is ignored and the method returns false. If this is not the case, the fill rate is tested and if the insertion results in the fill rate exceeding the selected max, a rehashing is performed. It expands the array by doubling in the same way that I have previously seen in connection with an *ArrayList*. Finally, the element is added to the list in the array that the index indicates. In case of a collision, it means that the list will contain more elements, and the goal is to run with such a low fill rate that the number of elements in the lists is few and preferably only 1.

The method *insert()* is relatively complex – at least if you also account for rehashing. Although the rehashing operation is complex and has linear complexity, it should only be performed occasionally, and in the same way with an *ArrayList*, the result is that *insert()* in the mean has a time complexity that is constant. That is, however, the condition that it is true that the lists have few elements, which are the same as few collisions.

EXERCISE 18: TEST HASHING

Create a test program that you can call *HashTester* when the program should test the class *Hash*. Start by adding a method:

```
private static <T> void print(IHash<T> hash)
{
}
```

which can print the content of a *IHash<T>*. Then add a simple test method *test1()* where you:

1. create a *IHash<String> names = new Hash()*
2. add 5 names
3. prints your *IHash<String>*
4. add one name more
5. prints your *IHash<String>*
6. add 5 names
7. prints your *IHash<String>*
8. add one name more
9. prints your *IHash<String>*

Note that this means that your *IHash<String>* needs to be expanded 3 times.

Add a class that you can call *Person* when the class must represent a person by a first and last name:

```
public class Person
{
 private String firstName;
 private String lastName;
```

Try to use NetBeans to generate

1. a constructor that has both values as parameters
2. *get* and *set* methods to both properties
3. a *toString()*
4. *equals()* and *hasCode()*

and study the code that NetBeans creates for you. You must then add another test method *test2()* whenever you need to

1. create a *IHash<Person>*
2. add 7 *Person* objects to your hash when two of objects must represent persons with the same
3. print your hash
4. remove a person when there should be a person with the same name as an object in your hash
5. print your hash

You must add another class to your project:

```
public class HashingTester<T> extends Hashing<T>
{
 public HashingTester() {}

 public HashingTester(int n) {}

 // Returns the capacity of the internal array.
 public int getCapacity() {}

 // Returns the current filling rate in percent.
 public double getFillDegree() {}

 // Returns the number of collisions
 public int getAllCollisions() {}

 // Returns the length of the largest collision chain.
 public int getMaxCollisions() {}
}
```

Then add the following print method to your test class *HashTester*:

```
private static <T> void print(HashingTester<T> hash)
{
 System.out.println("Capacity: " + hash.getCapacity());
 System.out.println("Size: " + hash.getSize());
 System.out.println("Fill degree: " + hash.getFillDegree());
 System.out.println("Collisions: " + hash.getAllCollisions());
 System.out.println("Max collision: " + hash.getMaxCollisions());
}
```

As the next step, write a test method *test3()* that creates a *Hash<Integer>*. The method should print how long it takes to place 1000000 elements in your hash, and as the last call the above print method. You must the write a similar method *test4()*, where the difference is that the type is *String* and that the method should places random strings in your hash:

```
private static String create(int n)
{
 StringBuilder builder = new StringBuilder();
 while (n-- > 0) builder.append((char)('A' + rand.nextInt(26)));
 return builder.toString();
}
```

Finally, write a test method *test5()* that is almost identical to *test4()* and only with the difference that you set the fill rate to 0.25. You may, if you like, experiment with the fill rate.

PROBLEM 6: QUADRATIC PROBING

Create a new project that you can call *HashTester2*. You must write a class *Hashing<T>*, which implements the interface *IHash<T>* and thus has the same methods as the class *Hash<T>*. The difference must be that the collision problem this time must be solved by means of quadratic probing.

Note that you must also write the necessary test methods, which should also include time measurements comparing with the class *Hash<T>*.

10 SETS

From mathematics you know the concept of a set, which is a well-defined container of objects. In other words, it is a collection. In a set, you can basically perform the following operations:

1. Ask if an element is included in the set
2. Create an intersection that creates a new set from two other sets, consisting of the elements that are included in both the other sets
3. Create an union that creates a new set from two other sets, consisting of the elements that are included in one of the two other sets and possibly both
4. Create a difference that creates a new set from two other sets, consisting of the elements that are included in the first of the two other sets but not in the last
5. Create a subset that creates a new set from another set, consisting of the elements from the other set which meets a condition

However, there will usually also be other operations, and you can, for example, define a set as follows:

```
public interface ISet<T> extends Iterable<T>
{
 public boolean insert(T elem);
 public boolean remove(T elem);
 public boolean contains(T elem);
 public boolean contains(Set<T> S);
 public Set<T> union(Set<T> S);
 public Set<T> intersection(Set<T> S);
 public Set<T> difference(Set<T> S);
 public Set<T> subset(Predicate<T> ok);
 public int getSize();
 public boolean isEmpty();
}
```

where the meaning of the individual methods should be self explanatory. A set does not mathematically guarantee anything about the order of the elements, and if you pass through all the elements of the set, you can not know the order in which they are processed. It is therefore obvious to implement a set as a collection using hashing, and the class *HashSet<T>* is not much more than a wrapper for a *Hash<T>*. Assuming that the methods *insert()*, *remove()* and *contains()* have constant time complexity, the same methods in *HashSet<T>* can be implemented with constant time complexity, while the methods *intersection(), union()* and *difference()* can be implemented with linear time complexity. The class *HashSet<T>* corresponds to Java's class with the same name.

In practice, however, one may be interested in how the elements of a *ISet<T>* being arranged, and instead, you can implement an *ISet<T>* using a binary tree. The class *TreeSet<T>* implements the interface *ISet<T>,* but using an *AVLTree<T>*. This, of course, means something about complexities as a constant time complexity is changed to logarithmic time complexity, and the complexity of *union()*, *intersection()* and *difference()* are no longer linear, but in practice the difference is not so great as a part of overhead is connected with the class *Hash<T>*.

EXERCISE 19: THE SET CLASSES

You must write a program that you can call *SetTester* to test the classes *HashSet<T>* and *TreeSet<T>*. Start by adding a method that can print the content of an *ISet<T>*:

```
private static <T> void print(ISet<T> S)
{
}
```

Next, add a test method:

```
private static void test1()
{
 ISet<Character> A = new HashSet('A', 'B', 'C', 'D');
 ISet<Character> B = new HashSet('C', 'D', 'E');
 // print A
 // print B
 // print the union of A and B
 // print the intersection of A and B
 // print the difference of A and B
 // print the subset of A that contains B and C
}
```

You must then write a test method *test2()*, which is identical to the above, but where the two sets instead have the type *TreeSet*. Also write a test method *test3()* where *A* is a *TreeSet* while *B* is a *HashSet*, and finally write a test method *test4()*, where *A* is a *HashSet*, while *B* is a *TreeSet*.

Add a method that initializes a set with *n* random integers between 0 and 2n when the method should return the set:

```
private static ISet<Integer> create(ISet<Integer> S, int n)
{
}
```

Add the following test method that prints how long it takes (measured in milliseconds) to perform *union()*, *intersection()* and *difference()*:

```
private static void test5()
{
 int N = 10000000;
 Set<Integer> A = create(new HashSet(), N);
 Set<Integer> B = create(new HashSet(), N);
 …
}
```

Write a similar test method *test6()*, which works the same way, but instead, the sets should have the type *TreeSet*.

11 MAPS

As the last data structure in this book, I would like to mention (and implement) a map. It is a collection that contains key/value pair where values (the value property) are identified by a key and thus a bit like a table in a database. A map can be defined as follows:

```
public interface IMap<K, T> extends Iterable<K>
{
 public T get(K key);
 public void set(K key, T value);
 public boolean contains(K key);
 public boolean insert(K key, T value);
 public T remove(K key);
 public int getSize();
 public boolean isEmpty();
}
```

where the methods should be self explanatory. Note that the type is parameterized with two parameters, where the first is the type of the keys, while the latter is the type for the values. A map must also implement the iterator pattern, which returns an iterator for the keys.

If you want to implement a map, there are several options, but basically you can implement it as a *HashMap*, which in principle means that the keys are saved using hashing. If you choose this solution, it means that by traversing the keys you find the objects in unpredictable order corresponding to traversing a *HashSet<K>*. Alternatively, you can implement a map as an *TreeMap*, which corresponds to the keys being stored in a binary search tree. This solution means that in a traverse the objects are sorted according to the key, but on the other hand, the methods of the class have a slightly poorer performance.

The specific class *HashMap<K, T>* is nothing but a hashing of elements of the form:

```
private class Entry<K, T>
{
 public K key;
 public T value;

 public Entry(K key)
 {
  this.key = key;
 }

 public Entry(K key, T value)
 {
  this.key = key;
```

```
  this.value = value;
 }

 @Override
 public boolean equals(Object obj)
 {
  if (getClass() == obj.getClass())
  {
   return ((Entry<K,T>)obj).key.equals(key);
  }
  else return false;
 }

 @Override
 public int hashCode()
 {
  return key.hashCode();
 }
}
```

which is an inner class, and the class *HashMap<K, T>* can then be defined as follows:

```
public class HashMap<K, T> implements IMap<K, T>
{
 private IHash<Element<K,T>> hash = new Hash();
```

The implementation of the individual methods are all trivial as the class *Hash<Entry<K, T>>* implements the necessary services.

For the class *TreeMap<K, T>*, the same applies. The parameter type K must be comparable this time, and the same must be the case for the class *Entry*, but else the class *TreeMapK, T>* is defined as:

```
public class TreeMap<K extends Comparable<K>,T> implements Map<K,T>
{
 private SearchTree<Entry<K,T>> tree = new AVLTree();
```

EXERCISE 20: THE MAP CLASSES

Together with the book's examples, there are four text files: *firstnames*, *lastnames*, *streetnames* and *zipcodes*, which contain some (Danish) first names and last names for persons, some street names and the Danish zip codes. These files should be used in this exercise, so copy them to where you easy can refer to them.

Write a program *MapTester*, where you should test the two classes *HashMap<K, T>* and *TreeMap<K, T>*. Add a simple model class *Zipcode*:

```
public class Zipcode
{
 private String code;
 private String city;
```

when in addition to a constructor there are no need to be anything else besides *get* methods for the two properties. Write correspondingly a model class:

```
public class Person
{
 private int key;
 private String firstname;
 private String lastname;
 private String street;
 private Zipcode zipcode;
```

when in the same way there only need to be a constructor and *get* methods. For the test program, define the following data structures:

```
public class MapTester
{
 private static IList<String> firstnames = new ArrayList();
 private static IList<String> lastnames = new ArrayList();
 private static IList<String> streetnames = new ArrayList();
 private static IList<Zipcode> zipcodes = new ArrayList();
```

and then add a method *initialize()* that initializes these data structures based on the contents of the four text files (860 first names, 778 last names, 401 street names and 1214 zip code). Then add a method

```
private static Person createPerson()
{
}
```

which can create a person when a person must have a key, which must be a random 9-digit integer, and when a *Person* has to be created with random values from the above data structures. Also write a method

```
private static void print(Person person)
{
}
```

which can print a line that contains information about a Person.

You must then write a test method

```
private static void test1(IMap<Integer, Person> map, int n)
{
}
```

which must creates and add *n* persons to a map. Next, the content of your map must be printed using an iterator. Perform the test method from *main()*, where you first transfer a *HashMap* as parameter and then a *TreeMap*. Write another test method

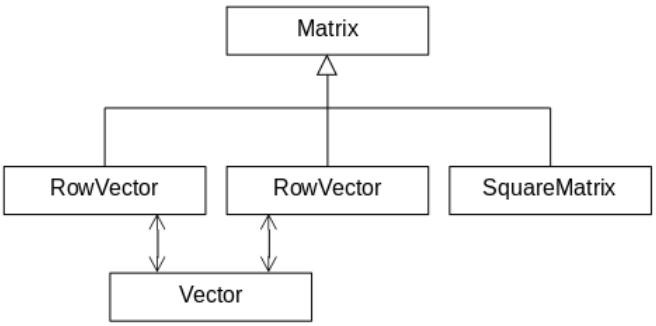
```
private static void test2(IMap<Integer, Person> map, int n, int m)
{
}
```

which creates and adds *n* persons to *map*. The method must then perform the *map.contains()* method *m* times, and you must measure how many milliseconds it takes. When you create the persons, you can save the keys to a list so that you can search both for existing people and non existing people. Test the method from *main()* both with a *HashMap* and an *TreeMap*.

12 A FINAL EXAMPLE

In mathematics, a vector is an array where the elements (usually) are real numbers, and you can perform different computational operations, such as adding two vectors, which simply means adding the elements in two arrays in pairs place for place. You can also determine the scalar product of two vectors, which is a real number. A similar term is a matrix that is nothing but a 2-dimensional array, and where the elements, like a vector, are real numbers. Also for matrices, a number of calculation rules apply, such as the addition of matrices, and you can (under certain conditions) multiply two matrices where the result again is a matrix. One of the classic tasks in programming is to write classes that represent vectors and matrices and the most common operations on these data structures, and there are many different solutions in the form of math libraries, and the subject of this chapter is to show yet another example of such a library. Perhaps it's not so much to do with collections, but it's data structures, and algorithms play a crucial role. The goal with this last example is addressing this issue, more than the practical application. The solution will additionally be an extension of the library *palib*, which contains a package *dk.data.torus.math*, with the classes relating to the implementation of classes to simple matrix algebra.

Specifically, I will implement the following classes:



When implementing such classes, it is necessary first to clarify when objects are compatible, and in accordance with the mathematical definitions, I generally agree that two vectors are compatible if they have the same number of coordinates. There are other options, and you might assume, for example, that missing coordinates should be perceived as 0. This would provide greater flexibility, but there would also be situations that would result in unintended results. Similarly, I assume that addition and subtraction of two matrices require that both matrices have the same number of rows and columns and that multiplication of two matrices requires that the first have the same number of columns as the other has rows.

Looking at the above class diagram, *Matrix* represents a *m x n* matrix with *m* rows and *n* columns, but the class *SquareMatrix* is a square matrix where the number of rows and

the number of columns is the same. *RowVector* is a *1 x n* matrix, while *ColVector* is *m x 1* matrix. Finally, there is the class *Vector*, which represents a vector with *n* coordinates. It's a very simple class, and it's mostly for the sake of completeness, and I do not want to use the class in the rest of this chapter, so you're referring to the completed code.

Basically, a *Matrix* is a rather simple data structure and is not much more than an encapsulation of a 2-dimensional array, but several of the operations are difficult to implement with satisfactory time complexity, especially for the methods in the class *SquareMatrix*. I want to start with the class *Matrix*, but first some comments regarding exceptions. Generally, the classes' methods do not test for index errors, and here I let the runtime system take care of the error handling. In situations where the classes' methods test for errors, they can raise a *MathException*, which is also a *RuntimeException*, and it is often due to an operation on vectors/matrices that are incompatible. Thus, the methods of the classes do not require a try/catch.

The class matrix can be defined as follows:

```
public class Matrix
{
 protected double[][] values; // backing array to coordinates
```

and should basically have methods for addition and subtraction of matrices and multiplication of a matrix with a constant. Additionally, there are also added methods that respectively add a constant to and subtract a constant from a matrix (although it is not classic matrix operations). Finally, there are multiplication of two matrices, which have always been the challenge, as for matrices of approximately the same size, they get a complexity that is $O(N^3)$. There are actually algorithms that do better, and one of them is the *Strassen algorithm* that has a complexity around $O(N^{2.7})$. Firstly, the algorithm is not quite simple to implement, and secondly, it only works for square matrices, and thirdly, for performance is gaining attention, you needs to work with very large matrices, before consider implementing the algorithm, maybe just like an override in the class *SquareMatrix* and might even be used for big matrices. I only implemented matrix multiplication in its classic form so there is room for improvements. There are also other algorithms for matrix multiplication, which have even better time complexity, but they primarily have theoretical interest.

Similar to the above, the implementation of the class *Matrix* is quite simple and only uses simple algorithms. The same applies to the two classes *RowVector* and *ColVector*, which do not need further comments here.

Back there is class *SquareMatrix*, and it is immediately more complex. The reason for the class is that particular matrix operations only make sense for square matrices, and it is primarily about calculating the determinant of a matrix, as well as determining the inverse of a matrix. Both of these operations are complex, which is also is reflected by the code. There are several ways to implement these operations, but overall, the basic definitions provide solutions with poor performance, especially because the definition of the determinant is recursive. Another way is to use LU decomposition and although I do not want to describe the algorithm here, the procedure is as follows:

If you have a square matrix, you have to switch to two rows without affecting the determinant, but where you have to multiplying the result with $(-1)^n$ where n is the number of exchanges. It is also allowed to subtract a multiple of one row from another row, thus reducing a square matrix A to an upper triangle matrix U and thus a matrix where all coordinates below the diagonal are 0. The determinant can then be determined as the product of all coordinates in the diagonal, but multiplied by the above sign. This results in many calculations and will therefore introduce significant uncertainties to the results, and it can be partially corrected by all the time make coordinates in the row with the highest value 0. This corresponds to the fact that during the process, a permutation of the rows corresponding to implicit substitutions is determined. When subtracting a multiple of a different row from a row, use a scale that is determined by the number of subtracts from being assigned to 0 at a particular space and all these scales can be collected in a lower triangle matrix L, which has 0 in diagonal. For the two matrices L and U, then A = L * U. The two matrices are assembled in a common matrix LU = L + U. The process is called LU decomposition and is a relatively complex process with a time complexity that is $O(N^3)$.

If you first performs a LU decomposition, it is simple to determine the determinant, as it is just the product of the coordinates in the diagonal. In fact, an LU decomposition consists of solving *n* equations with *n* variables and thus to solve a matrix equation *Ax = b*, where *x* and *b* are column vectors. The class *SquareMatrix* therefore also has a method *solve()* that can be used to solve such an equation system. In fact, it is not so much for solving equations, as because the method can immediately be used to determine the inverse matrix. If *v(i)* denotes a column vector, with a 1 on the *i*th place and the rest of the coordinates 0, you can determine the *i*th column in the inverse matrix as the solution to *A * x = v(i)* and thus using the method *solve()*.

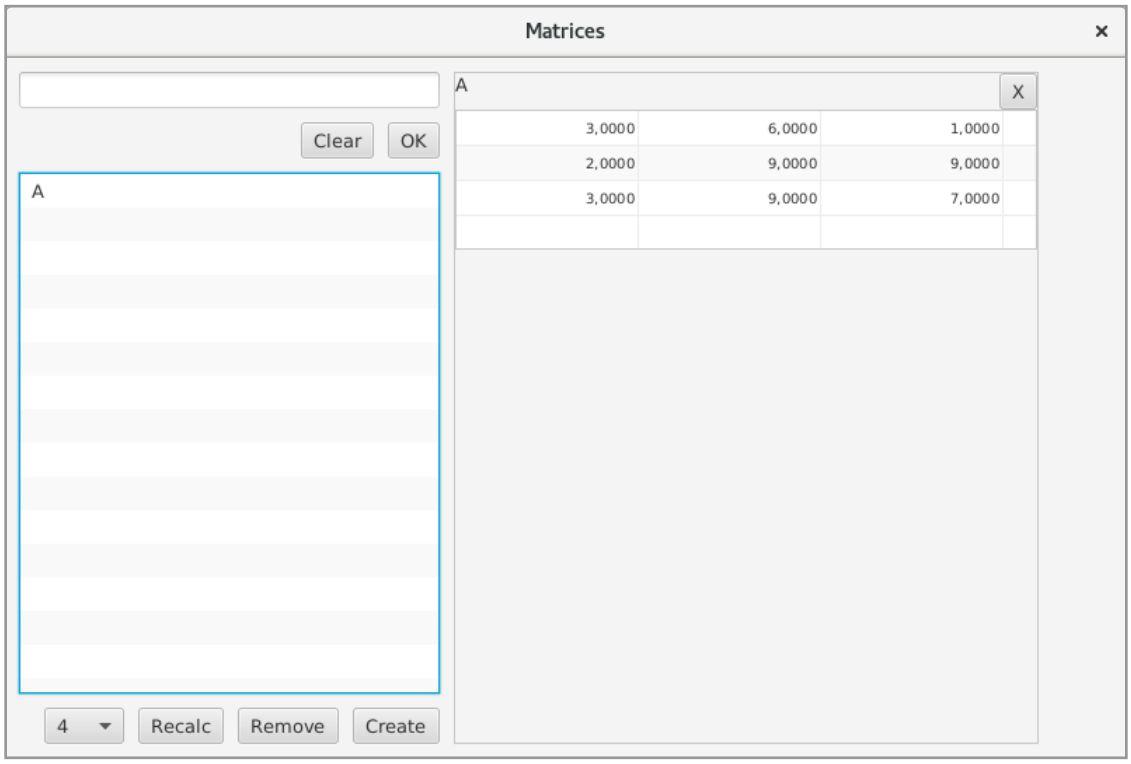
The class *SquareMatrix* is defined as follows:

```
public class SquareMatrix extends Matrix
{
 private SquareMatrix LU = null; // the decomposed matrix
 private int[] perm; // the permutations
 private boolean odd = false; // the sign of the determinant
```

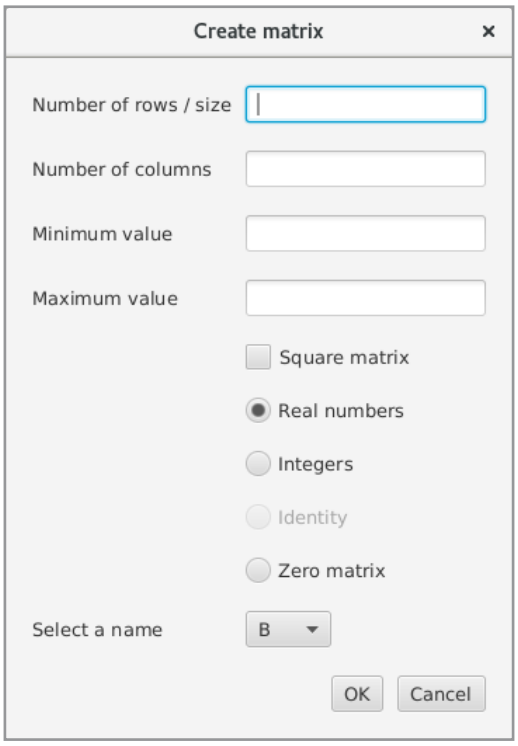
The meaning of the three variables is that if you first has performed a LU decomposition, it is not necessary to do it every time you determine the inverse, solve a linear equation or determine the determinant. Remember to invalidate the LU matrix if the current matrix is changed, what the class *SquareMatrix* also does.

12.1 A TEST PROGRAM

After I have written the above classes I have written a test program called *Matrices* which can be used to test some of the above classes. If you run the program, you will see the window below, where a single square 3×3 matrix has been created. To the left there is a *ListView* that shows an overview of the matrices that have been created and double-clicking on a matrix, opens a *TableView* that displays the matrix's values. You can also edit the values so that you can get a matrix with certain values. At the top of the *ListView* component there is an entry field and a few buttons. The input field is used to enter a matrix expression, such as *A + B*, and clicking *OK* will insert the expression into the *ListView* component, and you can see the result by double-clicking the expression (if the result is a matrix). Note that if you opens multiple matrices, they appear in a long horizontal row. Under the *ListView* component there are three buttons and a combo box. The last one is used to indicate how many decimals are to be used to display the results. If you edit values in an array (you can only edit the matrices created by clicking the *Create* button), the value of a formula may be incorrect and that is the goal of the button *Recalc* that recalculates the formula that is selected. You can use the button *Remove* to delete the formula for the matrix that is selected.


Matrices
Clear
OK
A
4
Recalc
Remove
Create
A
X
3,0000
6,0000
1,0000
2,0000
9,0000
9,0000
3,0000
9,0000
7,0000

Finally, the button *Create* and clicking on it you will get the following window:

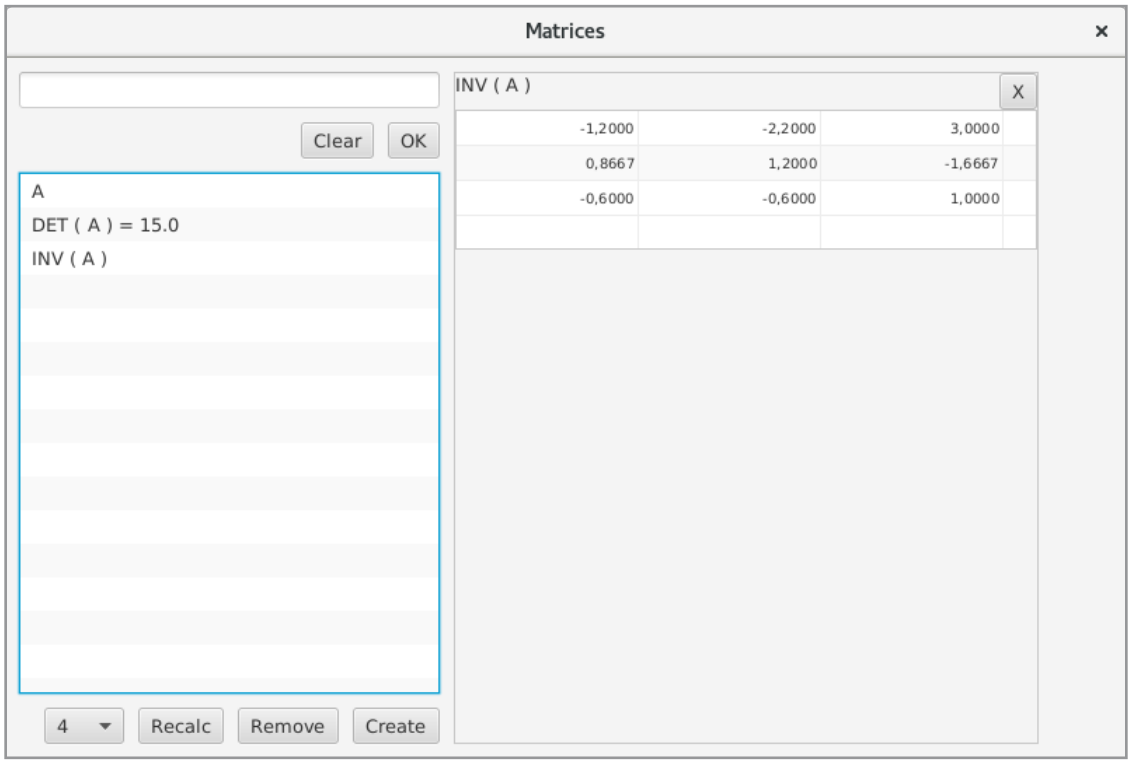


where you can create a new matrix. If the *Real numbers* or *Integers* radio button is clicked, the function will create a matrix of random numbers within the selected range. If you have clicked the bottom radio button get a 0 matrix and clicked on *Identity* (only available if *Square* matrix is selected), you get an identity matrix with 1 in the diagonal.

In terms of expressions, the program is relatively limited, but you can enter names of existing matrices and numerical constants. You can use +, – and * operators as well as parentheses. Additionally, you can use the functions:

- *DET(A)* which determines the determinant of A
- *INV(A)* which determines the inverse matrix for A
- *NORM(A)* which determines the norm of A
- *CON(A)* which determines the condition number for A
- *TRANS(A)* which determines the transposed matrix for A

but it will, of course, be easy to expand the program with more functions. Below I have shown the window after I have determined the determinant of A, determined the inverse matrix and opened it:


Matrices
Clear
OK
A
DET (A) = 15.0
INV (A)
4
Recalc
Remove
Create
INV (A)
X
-1,2000
-2,2000
3,0000
0,8667
1,2000
-1,6667
-0,6000
-0,6000
1,0000