

ID1020 Algorithms and Data structures

Description of central algorithms, data structures and ideas.

General instructions, assignments and useful background

2020-08-16

Table of Contents

1	Introduction.....	6
1.1	What is an algorithm and a data structure	6
1.2	Programming languages relations to algorithms and data structures	7
1.3	Why should you learn about algorithms and data structures	8
1.4	Disposition	8
2	What you should know from previous courses	9
3	Development environments.....	10
3.1	IDE	10
3.2	Command line interface	10
3.3	Recommended environments.....	10
4	Requirements on your code	11
4.1	Comments – documentation	11
4.2	Requirement: Comments in your code	11
4.3	Why you should design tests for your code.....	11
4.4	Requirement: Tests	12
4.5	Requirement: Where your test code should be placed.....	14
5	On the assignments	14
6	Assignment 1: A text changing filter in C	15
7	Assignment 2 a, b: Placing data in an array.....	15
8	Queues.....	16
8.1	Queues and Queueing policy	16
8.2	FIFO-queue.....	16
8.3	LIFO-queue or stack.....	16
8.4	Priority queue	16
9	Assignment 3 a, b: Basic data structures - Linked lists	17
9.1	Single linked list implementation of a LIFO-queue	17
9.1.1	Insert/enqueue.....	18
9.1.2	isEmpty	19
9.1.3	remove/dequeue	20
9.1.4	JAVA design of a LIFO-queue as an Abstract Data Type.....	22
9.2	Single linked list implementation of a FIFO-queue	23
9.2.1	isEmpty	23
9.2.2	insert/enqueue	23
9.2.3	remove/dequeue	24
9.3	Assignment 3a: Single linked lists	25
9.4	Double linked circular lists	27
9.5	Comparing single linked and double linked lists.....	30
9.5.1	Memory usage	30
9.5.2	Execution time.....	30
9.5.3	How to choose the proper list implementation	30

10	Complexity	32
10.1.1	Notations for complexity	33
11	Binary search trees	34
12	Queues and resizing arrays	37
12.1	Stack/LIFO-queue implemented by an array	37
12.2	LIFO-queue implemented by resizing arrays	38
12.2.1	Analysis	40
12.2.2	Comparison to a linked list implementation	40
12.3	FIFO-queue implemented as a fixed size circular buffer	41
12.4	Circular FIFO-queue implemented by resizing arrays	43
13	Ordering and sorting	45
13.1	Orders	45
13.2	Measures of the ordering of elements/sortedness of a set of elements	45
13.3	Basic properties of sorting algorithms	46
13.3.1	Stability	46
13.3.2	In-place	46
13.4	Comparison methods/functions	47
13.5	Priority queues	48
13.5.1	Priority queue data structures	48
13.6	Sorting algorithms	50
13.6.1	Some basics: <code>swap()</code> and how to generalize a sorting method	50
13.6.2	Insertionsort	51
13.6.3	Mergesort	52
13.6.4	Quicksort	53
13.6.5	Non-comparison based sorting	55
14	Searching	56
14.1	Binary search	56
14.2	Binary search trees	57
14.3	Searching by array indexing – hash tables	57
14.3.1	Hashing with separate chaining	59
15	Basic graph algorithms	61
15.1	Data structures for representing graphs	61
15.2	Common graph problems	63
15.3	Basic path finding algorithms in digraphs without edge weights	63
15.3.1	Depth first search	63
15.3.2	Breadth first search	64
16	Complexity revisited	65

17	Background: Structure your development - The best pieces of advice one can give!	67
17.1	Polya's problem solving method.....	67
17.1.1	Understand the problem	67
17.1.2	Make a plan	68
17.1.3	Carry out the plan	68
17.1.4	Look back.....	68
17.1.5	Where problems come from	68
17.2	Top-down and Bottom-up design.....	69
17.3	Some guiding principles when designing methods, data structures and classes	69
17.4	Proposed basic method for structured problem solving for programmers.....	70
18	Background: How do you know that your program is correct?	71
18.1	Verification	71
18.2	Validation and testing	71
18.2.1	Testing	71
18.2.2	Black-box and white-box testing.....	72
18.2.3	Goal of testing - exhaustive testing – the limits of testing.....	72
18.2.4	Basic strategies for testing.....	72
18.2.5	The role of testing in system development	72
18.2.6	On the design of unit/module tests for the course.....	73
19	Finding flaws in your design - debugging	74
19.1	Performance related errors.....	74
19.2	Logical errors.....	74
19.2.1	The analytic approach.....	74
19.2.2	Trace printouts.....	74
19.2.3	Debuggers.....	75
19.2.4	Comparing different debugging techniques	76
20	Background: Redirecting input/output.....	77
20.1	I/O to from the terminal.....	77
20.2	Redirecting I/O.....	78
20.3	Redirecting output	78
20.4	Redirecting input.....	80
20.5	Pipeing: Redirecting the output from one process to be the input of another process	81
20.6	An example how to pipe output/input between processes	81
21	Background: Some useful commands/things in UNIX/LINUX.....	83
21.1	Changing directory, path to a directory and listing the contents of a directory	83
21.2	man pages	83
21.3	Shells and shellscript.....	84
21.4	Accessing the LINUX servers at KTH Kista.....	84
22	Background: Programming language concepts	85
22.1	Syntax and semantics.....	85
22.2	Other language related concepts	85
23	Background: General concepts	88
24	Background: Object orientation classes, objects, class and instance variables/methods	93

25	Background: Part of the C- programming language.....	94
26	Background: Basic functions for I/O in C	99
27	Background: References, pointers, addresses and memory management	100
27.1	Pointers and addresses in C	100
27.2	A case for why we need pointers (references).....	101
27.3	Memory management in C	101
27.4	Dynamic memory allocation in C	102
27.5	References in JAVA	104
28	Background: How to build linked data structures in C	105
29	Background: Common pitfalls when programming C.....	107
29.1	Test for equality in the condition of <code>if</code> , <code>for</code> or <code>while</code> statements	107
29.2	Confusing bit-wise and logical operators	108
29.3	Indexing out of bounds and addressing errors.....	108
30	Index.....	109

1 Introduction

This text has been written in the strange times influenced by the Covid-19 pandemic. It is intended to help students understand both the basics of programming and computer science as well as giving an alternative, and simpler, introduction to the central concepts in the course. Hopefully you will find it easier to study the material in the course book and lecture notes if you first read the corresponding sections in this text.

How should you study this material? As a novice on programming it is easy to get stuck on details if you start at the wrong end of the problem, such as where to place semicolons and other syntax related problems. While the proverb “*The devil is in the details*” is true, it is equally true that focusing on details to early generate problems: “*You do not see the forest for all the trees*”. So when studying this material you should start by focusing on the bigger picture:

- 1. Start by understanding the problem the algorithm/data structure is aiming to solve**
- 2. Focus on understanding the underlying ideas and general structure of the algorithm/data structure**
- 3. Only when you understand the problem and the general structure of the algorithm should you start looking at details of how the algorithm, implementations and code are designed and work.**

The goal of the Algorithms and Data structures course is that you should learn about some of the basic algorithms and data structures used to address common problems encountered when designing software. In many cases there are more than one algorithm/data structure that could be used to solve a specific problem. The best choice of algorithm/data structure may depend on parameters such as the size and characteristics of the input, memory usage and execution time etc.

With this said it should be evident that this course is not a programming primer but a course which builds on that you know how to program. The content is not programming language dependent in the sense that the algorithms and data structures can be implemented in different languages. However, as different programming languages have different pros and cons you should be able to select the appropriate language among those you have encountered at KTH to solve a specific problem.

In this course we will use the JAVA language for most of the assignments as an example of an object oriented language. But we will also use the C programming language in cases where its lower overhead and more direct access to system calls¹ can result in more efficient solutions.

1.1 What is an algorithm and a data structure

An algorithm is a process or description of how to solve a problem or a class of problems. Algorithms can be described mathematically, in pseudo code, flow charts, programming languages or in a natural language. Algorithms are part of our everyday life. A recipe for how to bake a cake or a manual for how to change the battery in cell phone are examples of algorithms. Each program you have written is also an algorithm. Thus algorithms are the very core of programming. In the context of programming we often encounter problems that are common for many problems we solve. Typical examples are storing and accessing data, queues, sorting, and searching in data sets. For these there are well-researched algorithms and data structure which is what you will study in the course.

¹ System calls are the functions that implements the interface to the Operating system, i.e. the services a user program can get from the operating system. Examples of such services can be to read/write files or to allocate memory

Ideally an algorithm should be provably correct and efficient. We should also know its properties such as how it scales with the problem size or how its performance is affected by properties of the input. To achieve this one should be able to describe the algorithm mathematically. As a program also is an algorithm it would be preferable to describe programs mathematically. However, most programming languages are not well-defined (i.e. has a well-defined semantics and syntax) enough to allow them to be used to specify and prove the correctness of algorithms.

A data structure is a way of organizing data, store it and manage it. Or in other words, a data structure is a collection of data values, the relationship among the data and the operations that can be applied to the data structure. Ideally data structures should be selected so that they support the algorithms used in an effective manner and allow for the algorithm (program) to be further developed to meet future needs.

Many (most) of the algorithms and data structures used to solve such problems are available as library routines, as abstract data types (ADT) or as services on the Internet all readily available for use. So why should you bother to learn about them and their properties?

Learning about algorithms and data structures will allow you to:

- Select the appropriate algorithms and data structures for different problems
- Adapt algorithms and data structures for special purposes
- By seeing well designed algorithms and data structures you may also learn how to improve the design of your own algorithms and data structures.
- You can also learn what problems you can solve effectively and what problems you cannot solve at all or solve effectively (i.e. what problems that are *intractable*)

1.2 Programming languages relations to algorithms and data structures

Algorithms and data structures have also influenced the design of programming languages. Object oriented languages are based on the idea that data structures and algorithms should be merged into objects which hold data and methods to access/manipulate the data that implement the API (Application Programming Interface). This is one way of implementing Abstract Data Types (ADTs) where the internal representation of the data structures and internal methods (functions) are hidden from the user (this is called encapsulation). ADT is a useful idea which can simplify the reuse of code. However, it is worthwhile to point out that the idea of ADTs can be useful and can be implemented in programming languages which are not object-oriented. In C it is possible to hide functions so that they cannot be accessed from outside the compilation unit. The compilation unit is the source code file in which the functions are defined (written).

1.3 Why should you learn about algorithms and data structures

Many of the algorithms and data structures you will learn about on this course are readily available in libraries and many services such as data storage, sorting, indexing, data bases, AI etc. are available as services in the Internet. Moreover there is a trend towards “no-code” systems which allows users to develop applications without any prior knowledge of coding. Without writing a single line of code people (one person) have developed quite advanced applications such as (nearly) all functionality of Twitter in less than a week. So why should you learn about algorithms and data structures? The market is probably changing but there will always be a need for qualified programmers who can:

- Design and implement solutions for systems where there are no, and probably never will be “no-code” environments such as for embedded systems.
- Who can design and implement the underlying systems which are the basis for distributed services and “no-code” systems.
- Who can re-design “no-code” applications into more efficient and scalable code.
- Who can tailor specific solutions to problems which cannot be solved by library methods/functions, services on the Internet or “no-code” systems.
- Can understand what problems are possible to find solutions to and how to design solutions which will scale to meet demand.

1.4 Disposition

The first part of this text contains sections covering prerequisites and requirements for the lab assignments in the course.

This is followed by introductory assignments that will be useful further on in the course and introductions to the areas covered in the.

Finally there are several sections covering background material. The purpose of these are both to provide specific information targeted to help you solve the assignments but there are also sections covering more general knowledge/concepts that are part of the vocabulary for IT-engineers. This vocabulary and concepts are not always covered in courses, yet it is often assumed that one at least has some basic understanding of them. Hopefully, these sections can help demystify some concepts...

2 What you should know from previous courses

When designing courses and chains of courses one base the course design on the assumption that students entering the course have met the learning outcomes and know the content of pre-requisite courses. However, time is a factor and knowledge may fade.

For ID1020 you should have working knowledge of the following:

Concept	JAVA	C	Example
Development environment	X	X	IntelliJ, Eclipse, command shell with <code>gcc</code>
Simple debugging	X	X	Eg. what is built in to the IDE ² you have used (for example Eclipse) and/or <code>gdb</code>
Basic datatypes and variables in C and Java	X	X	<code>int</code> , <code>double</code> , <code>char</code> , <code>char *</code> , <code>int * string</code>
Arrays	X	X	
Flow control	X	X	<code>if</code> , <code>while</code> , <code>for</code>
Methods, functions	X	X	
Parameter passing	X	X	value, reference, pointer
Classes	X		
Instantiating objects	X		
References, pointers, addresses	X		Build linked data structures by using objects, C: <code>struct</code> and pointers
Pointers, addresses, dereferencing		X	Parameter passing, calculating addresses, dereferencing. Build linked data structures by using <code>struct</code> and pointers
Simple I/O	X	X	Read and write numbers, characters and text to/from a terminal (window)
Memory management		X	<code>malloc()</code> , <code>free()</code>
Arguments to <code>main()</code>	X	X	

² IDE – Integrated Development Environment

3 Development environments

Before you start the ID1020 course you should have a working development environment. Our recommendation is that you have both an integrated development environment (IDE) and a command line environment.

3.1 IDE

An IDE integrates things such as text editor, syntax control, compiler, debugger, run-time environment etc. There are certainly advantages to using an IDE during the development of a programs/systems but there are also things that are best/easiest done in a command line environment.

3.2 Command line interface

A command line interface is implemented by a user-level program called a shell. The shell reads commands from the user, parses (interprets) them and performs the commands by either executing them directly in the shell by using system calls³ or creating processes in which the command (in case it is a program that should be started) is executed. As the shell is a user level program there are often different shells that the user can chose from. A commonly used shell is `bash` (Bourne-again shell) which is the standard shell for GNU and most Linux distributions.

The idea of interacting with the operating system through a shell originates from the UNIX operating system and LINUX uses the same concept. Even Microsoft has had to recognize that professional developers and system administrators requires this type of interface and has introduced a LINUX shell-like interface in newer versions of their operating systems.

A command line environment has other advantages such as that you can select different components as your editor, debugger etc. according to your own choice. It allows you to make your programs read and write input/output from and to files through redirection without having to know how to open/read/write files. In the course this is a distinct advantage as you will have to read large volumes of input data from files and in some cases also generate large data files.

This type of environment normally allows you to program⁴ sequences of commands that you want to be performed by the shell, such as to start several executions of a program with different inputs and save the outputs to files. Once you are able to use a command line interface you will discover that it is a very powerful tool and it is often preferred by professional developers. At least you would expect a professional developer to be able to use a command line interface.

Furthermore, you cannot expect a user to have to install the same IDE you have used to execute an application you have developed in your IDE – the user would normally expect to use your application as a stand-alone application that could be executed directly by the operating system.

3.3 Recommended environments

IDE: IntelliJ for JAVA by JetBrains. This is a freely available professional IDE also used in many courses at KTH campus Valhallavägen

Command line interface: If you use a UNIX/Linux operating system you will have access to a command line interface. Note that MacOS is a UNIX operating system. For users of relatively modern Microsoft OSes there is something called Powershell which implements a `bash`-like command line interface.

³ The system calls are the methods that the operating system provides for the user, i.e. the interface to the operating system. For most operating systems the system calls are implemented in C.

⁴ A shell normally implements an interpreted programming language referred to as shellscrip (the syntax and semantics of the shellscrip may vary in different shells)

4 Requirements on your code

The requirements on your code in this course are basic professional requirements. The code that you will develop as an engineer will be used in applications that people depend on – hence you should make sure that it is correct, efficient and possible to maintain. This means that all code you write should be well documented and tested.

4.1 Comments – documentation

Comments and documentation serve two purposes. The most intuitive is to serve the purpose of explaining to others what problem your code is supposed to solve and how it does it. *Equally important is that by writing documentation, you will understand your code better and possibly see where it can be improved or even where it might be incorrect.*

Documentation may come as: i) documentation that is separate from the code; and ii) documentation integrated into the code in the form of comments. In real projects you will have both types of documentation. However, in this course you are only required to write well-commented code.

The advantage of having well-commented code is that by not separating the code from the documentation (comments) the documentation is more likely to reflect any changes to the code. Hence, it should not come as a surprise that there are systems available, such as **Javadoc**, which can collect specially formatted comments from (JAVA) code and generate separate documentation from the comments. Javadoc can parse JAVA code to generate HTML-documents with pre-formatted headings from comments in the code. Javadoc is integrated in many IDE's such as Eclipse. While it is not a requirement to use Javadoc in this course, it is recommended that you learn how to use it.

- How to write comments for Javadoc is explained in:
<https://www.oracle.com/technetwork/java/javase/tech/index-137868.html>.

4.2 Requirement: Comments in your code

The minimum requirements on your code in this course is that you have inserted the following comments into it:

- Start each source code file with a comment stating:
 - Who is the author
 - When the code was generated, when it has been updated
 - What problem the code solves
 - How it is used (how it is executed, what input it takes, what it outputs)
 - What it has been based on (if you have “borrowed” code, then you must state what code and from what source)
- Each class and method should have a comment describing it and its purpose
- Each “non-intuitive” declaration and statement should have an explanatory comment

4.3 Why you should design tests for your code

One of the things that distinguishes an engineer from an amateur is that we would expect the engineer to create solutions that are both efficient and correct. Faulty or erroneous software can at best be annoying. But incorrect code could easily lead to more severe consequences and in the worst case threaten people's lives. Lives may be jeopardized if the safety systems of an auto pilot systems in an aircraft does not work correctly or if a defibrillator fails to restart a heart. For an example of the perils bad code opens for in our society, how it could be attacked and what costs it could inflict you may want to google for “*the persistence of chaos*”. Correctness and efficiency are key elements of this course. So what can we do to build systems that work correctly?

Ideally we would like to guarantee the correctness of our systems by proving it. This is referred to as *verification* and is in most cases out of scope to do for complete systems. We usually cannot verify the whole chain necessary to execute a program consisting not only of our code, but also of computer hardware, operating system, compilers, run-time environments and networks. However, we can in many cases prove the correctness of algorithms by using mathematics and we should do this.

In most cases we have to stick to the second best alternative which is to validate our systems.

*Validation*⁵ means that we try to ensure/show that it is plausible that a system works properly. In software development this is a process which starts already in the planning stages of a project where we should study the problem and try to understand it in detail before we start designing solutions for it. In the design process we should also design tests early on for how to test that the system we implement works as intended. Testing is one method for validating the implementation of a system. Testing and tests are central to most system development and project methods such as Agile project methods and Test Driven Development (TDD).

What is important to realize as a student is that to be able to design proper tests you also have to gain a thorough understanding of how the systems you design and the algorithms you use work in detail. Thus testing the algorithms in the course and how you use them is a good way to learn and understand the internals of how the algorithms and data structures work.

4.4 Requirement: Tests

In this course all code you write should have proper unit (module) tests. Unit testing refers to the testing of the individual units a system is built from. In particular the tests you design should perform/allow for glass/white box testing of your code. This means that the tests should allow for testing the code with all possible/plausible inputs and should test all parts of the code (i.e. all methods/branches). The input should not only be the expected or correct input but also non-expected and incorrect input. Testing all parts of the code means that if you detect that parts of the code never can be reached, (dead code), then these parts of the code should be removed.

In this course you can in many cases simply have your program repeatedly prompt for input, which is read from the keyboard (i.e. `stdin`), until the input provided indicates that the test should end. The output is printed to the screen (i.e. `stdout`). This would allow you to redirect input and output so that it could be read from a file and printed to another file. Thus you can run the tests manually with input from the keyboard or create one or more input files with inputs for the tests. If you are not familiar with how to redirect input/output you should read Section 20.

⁵ It is quite common that the concepts of verification and validation are confused on the WWW

Example: Assume we would like to test the correctness of a method with two parameters of type *double* which divides the parameters *a* and *b* and returns the result (*a/b*). For this simple example we could test with a number of combinations of *a* and *b* to cover the cases of using positive, negative and zero values. We could also test with different datatypes such integer or floating point values as parameters (i.e. we need to know if the method should work properly with both integer and floating point values as parameters?). The values used should generate results where we know what to expect as results. In this case we could make a list of the combinations of values for *a* and *b* and the expected results. Testing with *b=0* should probably result in an error. Moreover, one could design tests to test the accuracy of the division and the maximal/minimal values of the parameters that can be used while the output is still correct.

a	b	result
1	2	0.5 (assuming the method should work properly with integer input)
-1	2	-0.5
1	-2	-0.5
-1	-2	0.5
1.0	2	0.5
1	2.0	0.5
1.0	2.0	0.5
-1.0	2.0	-0.5
1.0	-2.0	-0.5
-1.0	-2.0	0.5
1	0	error
1	0.0	error
1.0	0	error
1.0	0.0	error

What one should learn from the above is that it is difficult and in many cases impossible to (manually) do exhaustive testing⁶ of even simple code with few parameters. When hardware is designed it is described in a type of programming language. And there has been an example of where a large hardware manufacturer shipped many thousands (millions) of processors used in PC's that had an error in the floating point division.

However, that testing is hard should not prevent you from designing good tests!

⁶ Exhaustive testing is hard/impossible to do even with automated tools

4.5 Requirement: Where your test code should be placed

So where should you place your test code? In JAVA each class can have a class-method `main()`. Thus you should place your test code for each class (unit) in the class main-method.

In C you cannot have more than one `main()` function in a program. In fact the names of globally visible functions and variables need to be unique in a program to avoid name-collisions. So in this course you place your test code in a function declared as `static void test()`⁷ for each assignment.

5 On the assignments

By solving the assignments found in this text you will create solutions you can build on for many of the other lab assignments based on problems from the course book. If you plan on taking the ADK course you may want to go the extra mile and try to solve all assignments in both JAVA and C.

⁷The static declaration of the function `test()` will make it invisible outside the source code file (i.e. the unit of compilation) in which it is declared (i.e. the name/identifier will not appear in the symbol table)

6 Assignment 1: A text changing filter in C

Assignment 1: Create a filter written in C that reads characters from `stdin` until an end-of-file marker (`EOF`) is read. For every character read, the filter should check if the character is the character `'a'` in which case it should output an `'X'` to `stdout` otherwise it should output the character read to `stdout`.

Tests:

- 1) test the filter by manually insert text from the keyboard. `EOF` is normally represented by `<ctrl>-d`
- 2) write a small text containing a test text. Feed the text file as input to the filter by redirecting the input (see Section 18) to be read from the text file

Hints: Use the functions `getchar()` and `putchar()` (see Section 0) , a `main()` function containing one `while`-loop and one `if-else` statement (it need not be more complex). The code will look the same regardless if you use Windows or UNIX/LINUX

Extra: Try to implement the same program in JAVA and test it in the same way as the C-program.


7 Assignment 2 a, b: Placing data in an array

Assignment 2a: Create a JAVA-program which:

- 1) Reads a positive number from `stdin` into an `int` variable named `nrElements`. The value is the number of elements in a dynamically allocated array of integers.
- 2) Creates an array of integers with `nrElements` elements.
- 3) Reads `nrElements` integers from `stdin` and stores them in the array.
- 4) Prints the elements of the array to `stdout` in reverse order compared to how they were input.

Execute the program reading the input from the keyboard.

Create a text file containing the input (see the example below of a file where the first number is the number of elements followed by the five elements) and execute the program feeding the text file as input by redirecting the input to be read from the file instead of from the keyboard.



5
27
55
2
1

38

Assignment 2b: Redo the assignment by writing a C-program which does the same thing as the JAVA-program and execute it reading input from the keyboard and from the text file respectively. (See Section 0 for how to manage memory allocation in C).

8 Queues

Queues are one basic example of ADTs used in many applications. Typically we use them for scheduling processes in operating systems or jobs in administrative systems. They are also used in many other applications such as route finding algorithms in navigation systems.

We use and encounter many algorithms and data structures such as queues everywhere, not only in programs, but also in real-life. We have all had to stand in lines to pay at our grocery shop or wait to be seated at a restaurant. When learning and reasoning about algorithms and data structures it is often useful to relate to the real world.

8.1 Queues and Queueing policy

A queue supports two basic operations: i) to insert or enqueue a new element to the queue; and ii) to remove/dequeue and return an element from the queue. Which element returned by a remove/dequeue operation is defined by the queueing policy. Different queues are defined by their queueing policies. The three most common policies are: i) first-in first-out (FIFO); ii) last-in first-out (LIFO or stack); and iii) priority queues.

8.2 FIFO-queue

A FIFO-queue is probably the most intuitive queueing policy. It is the kind of queue we encounter when we stand in line waiting to be served by for example a cashier. The cashier will serve (remove/dequeue) the customer waiting the longest in the queue thus implementing a First-In First-Out (FIFO) policy.

What we should observe is that the queueing policy only defines which element to return when a remove/dequeue operation is performed. It does not, and should not, state anything on how the elements are stored in the queue. That is, we separate the algorithm (queueing policy) from the data structure (implementation). ***Separating policy from implementation is one of the basics to achieve good designs.*** However, the easiest and probably most efficient data structure to implement a FIFO-queue is one which mimics how we stand in line. That is the elements are ordered by the order in which they have been inserted/enqueued to the queue. In such an implementation we would insert elements to the back –end of the queue and remove elements from the front-end (often referred to as the *head* of the queue)

8.3 LIFO-queue or stack

In a stack, also called a Last-In First-Out queue, elements are removed so that the last element inserted/enqueued is returned by a remove/dequeue operation. The most natural data structure in this case is a structure where elements are inserted/removed at the front-end (the front-end is often called head if we talk of a queue or if we call it a stack that element is often called the top) of the queue.

8.4 Priority queue

In a priority queue the element returned in a remove/dequeue operation is the element in the queue that has the highest priority. Priority queues are used in applications such as simulations, scheduling of processes in an operating system and in route/path finding algorithms in navigation applications.

Priorities may consist of one or several fields (dimensions). Priorities may be explicit values in the element or implicit (not a value in the elements). For instance are processes in an operating system scheduled by their priority (operating system processes usually have higher priority than user processes). If there are more than one processes with the same priority in the queue they will be ordered by their arrival time – that is FIFO-order. In this case the priority the operating system assigns to the process and stores in the data structure representing the process is an explicit, while the arrival time (which is not stored) is implicit.

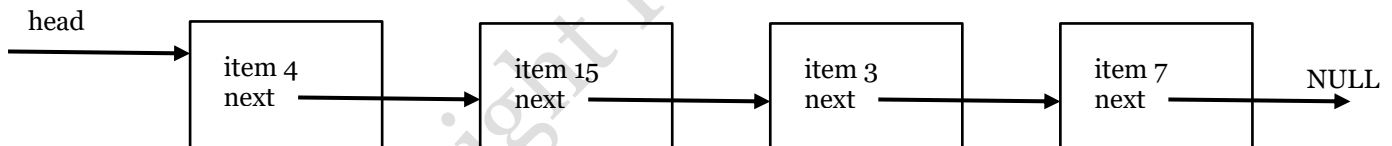
9 Linked lists

In many applications there is a need to handle and store sets of data that may vary in size also during the execution. One commonly used technique to build data structures to hold such data sets is to use linked data structures. Linked lists are suitable candidates to implement queues.

In JAVA a linked data structure is built from objects which contain data and references (see Section o) to other objects in the data structure. Such structures can be built in most languages. In C you would use structs instead of objects, pointers (variables containing addresses) instead of references and you would have to manage memory allocation/deallocation yourself. For further details on how to do this in C see Section o

Programming tip: Finding errors in linked data structures by executing and/or debugging the code with a debugger is inherently difficult and tedious. So the advice is as always to follow Polya's method for problem solving and make sure you understand the problem properly, design simple methods that do one thing good to build the solution from and test them one-by-one as you implement them. When designing the methods it is helpful to draw pictures of elements in the list and how references/pointers should be updated.

The simplest example of a linked data structure is a single linked linear list. The essential parts of code for a class that can be used to instantiate the elements (nodes) of such a data structure is found in Section o. Below you find an illustration of a linked list built from such objects where `item` is used to store an integer value and `next` is a reference to the next element in the list (or `NULL` if the element is the last element in the list). `head` is a reference to the first element in the list. The list in the figure contains elements with the data 4, 15, 3, 7.



9.1 Single linked list implementation of a LIFO-queue

To implement a LIFO-queue by a single linked list is relatively straightforward as all operations on the queue operates on the first element in the list (the list referred/pointed to by `head`).

Initially the queue is empty and the reference/pointer to the first element in the list/queue, `head`, should refer/point to `null`.



Observe: if `head` is global variable (accessible from the methods/functions such as: `enqueue()`, `isEmpty()` ...) we need not have it as a parameter to the methods/functions.

9.1.1 Insert/enqueue

Pseudo code for the enqueue method/function to insert a new element into the queue where the value is of type item is as follows:

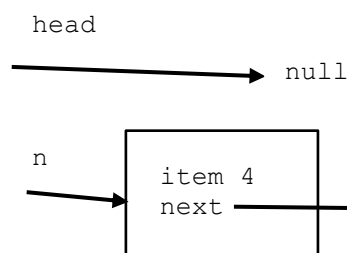
```
void enqueue(element head, itemType x)
    element n = new element(x)    //create a new element containing x ,
                                   // in JAVA element is an object, in C
                                   // it would be a struct
    n->next = head                // next reference/pointer of the new
                                   // element should point/refer to what head
                                   // referred to
    head = n                      // finally update head to point/refer to
                                   // the new first element in the list
```

Example: The figures below show how a first element with value 4 is inserted in the queue followed by the insertion of an element with value 33 (in this example we assume the data type of item is the type int).

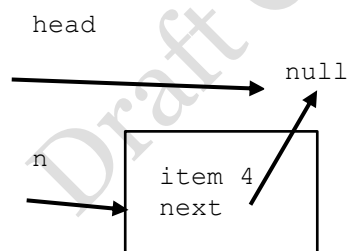
Initial state



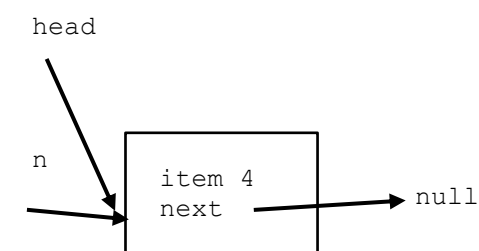
Create a new element



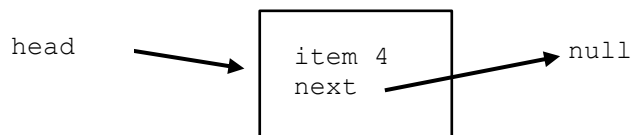
Set next in the new element to refer/point to what head is referring/pointing to.



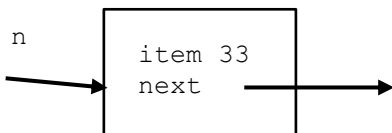
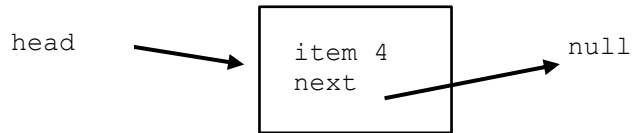
The last step is to set head to refer/point to the new element.



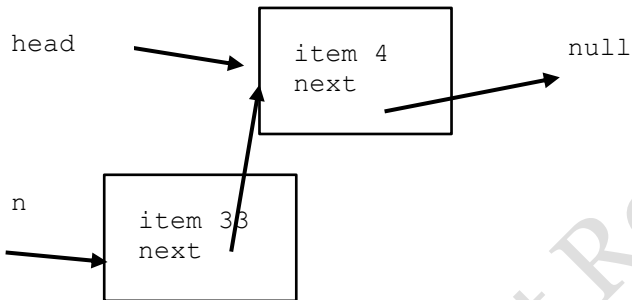
If we then insert another element with value 33 for `item` we will have the following sequence



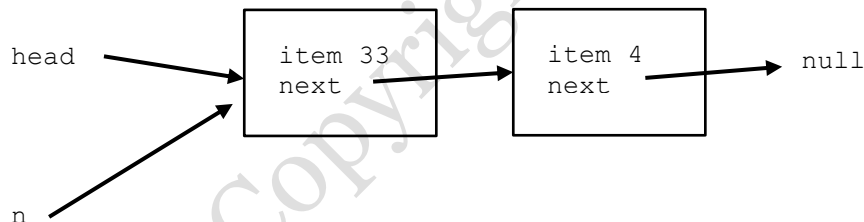
Create the new element



Set `next` in the new element to refer/point to what `head` is referring/pointing to.



And finally set `head` to refer/point to the new element.



Even though it may seem as the elements are moving around in the drawings above, they do remain on the same locations (addresses) in memory.

9.1.2 isEmpty

It may occur that we want to check whether the queue is empty or not. This is a simple check to implement as `head` will refer/point to `null` in this implementation of a LIFO-queue if it is an empty queue.

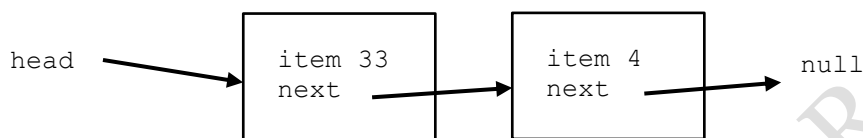
```
boolean isEmpty(element head)
    return head == null
```

9.1.3 remove/dequeue

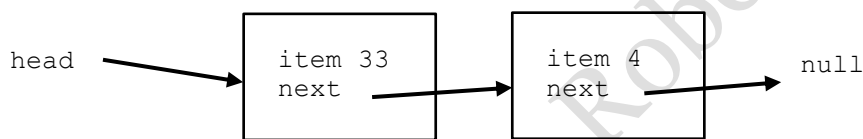
Implementation of `dequeue` is straightforward. If the queue is not empty, you simply store the value of the first element in the queue (if any), remove that element (the first) from the queue and update `head` to point to the next element.

```
itemType dequeue(element head)
if(head == null) error handling //In JAVA throw an exception, in C you
                                // would probably return a value
                                // indicating the queue was empty
else
    itemType x = head->item      // save value to return
    head = head->next            // set head to point to the next element
    return x
```

As an example we remove (dequeue) an element from the example above.

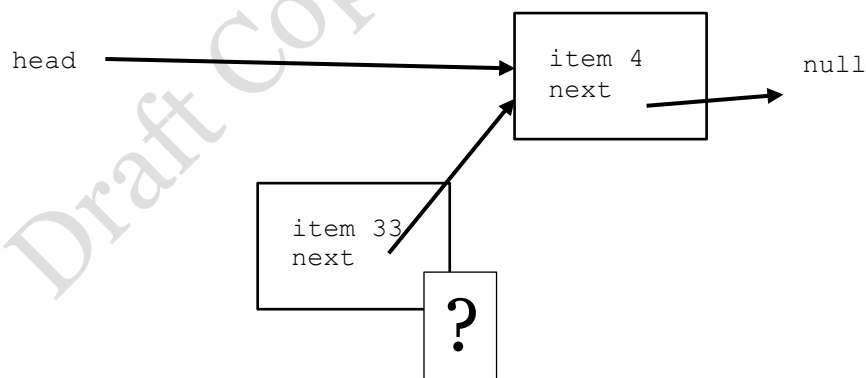


Check that the queue is non-empty. Then save the value to return



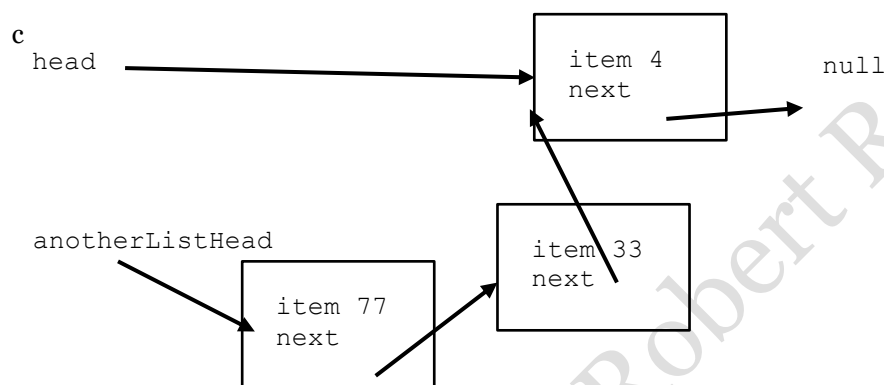
```
itemType x = head->item
```

Then reset `head` to point to what the first elements `next` reference/pointer is referring/pointing to. This will be the second element if the queue contained at least two elements or `null` if the queue contained only one element. Finally return what is stored in `x`.



What remains to understand is what happens or how we should deal with the element containing `item == 33`? In a language like JAVA where memory management for objects are automated we do not have to do anything. In this case the element is an object and when we no longer can reach it through any reference in the program the memory allocated for it will, eventually, be reclaimed by the garbage collector.

However, it is advisable that all reference/pointer variables which we no longer will use always be set to refer/point to null/NULL. Thus for the example above, we should set the `next` reference/pointer in the element containing 33 to `null/NULL`. A reason valid for both C and JAVA is that if the object/data structure (in the case above the element containing 33) is still in use in the program after having been removed from the list we could accidentally assume that the `next` reference/pointer still is a correct reference/pointer to something valid unless it is `null/NULL`. For example if we at a later stage in the execution of the program by accident use the element containing 33 as if it still was part of a list it would connect to the list we removed the element from (see example below). Moreover, if the element containing 33 is still used by the program, the rest of the list (i.e. the element containing 4) cannot be garbage collected by the JAVA garbage collector until both `head` and the element containing 33 no longer refers to it.



So the lesson learned is that in JAVA we should also always set references no longer used to `null` to avoid leaving references to objects we no longer use but where a dangling reference to the object may prevent the garbage collector to reclaim the memory.

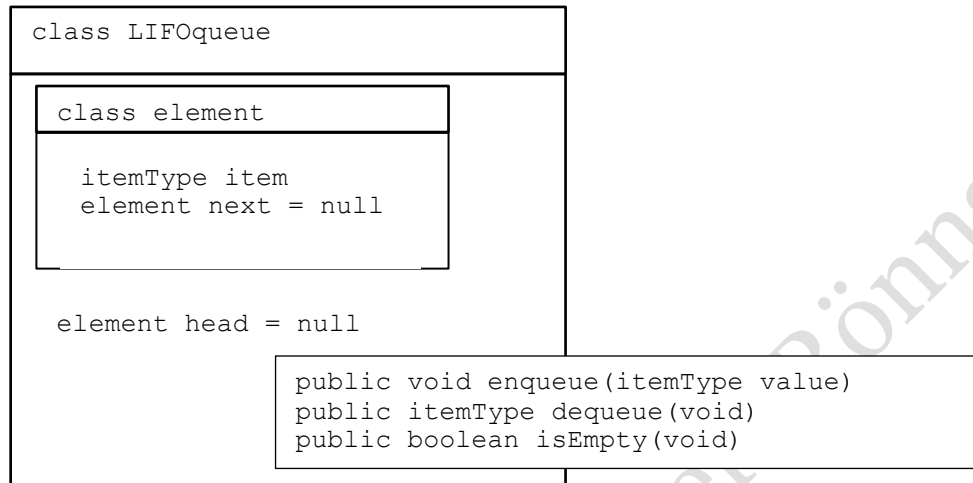
In languages like C where the user has to explicitly manage memory for dynamically created data areas such as the elements in the case above, the user has to deallocate the memory explicitly when it is no longer used. In JAVA the system will allocate memory whenever a new object is instantiated by **new** and the memory is reclaimed by the garbage collector when there no longer is any way to reach the object through any chain of references.

Failure to properly manage memory allocation in C and to leave dangling pointers (pointers/references referring to objects/data which we no longer use) in JAVA which prevents the garbage collector to reclaim memory⁸ will lead to **memory leaks** (also known as **loitering**). If a program executes for a long period, memory leaks may lead to that the memory available for the program is exhausted and that the program may freeze or simply crash.

⁸ The garbage collector reclaims memory which no longer can be reached from the program when it is executing. Thus as long as there is a direct reference or a chain of references leading to an object it cannot be garbage collected (i.e. the memory cannot be reclaimed to be reused for another purpose)

9.1.4 JAVA design of a LIFO-queue as an Abstract Data Type

In JAVA it is natural to design an implementation of a LIFO-queue as a class. The class could contain an inner class to implement the elements and a reference to the head of the list. What would be accessible from the outside would be the methods to insert/enqueue, remove/dequeue elements and the method to check if the list is empty (isEmpty). These three methods make up the Application Programming Interface (API) of the class. The class itself could be characterized as an Abstract Data Type (ADT) where the exact details of the implementation are encapsulated (hidden) from the user.



9.2 Single linked list implementation of a FIFO-queue

The only difference between a FIFO-queue and a LIFO-queue is that in the most natural implementation of a FIFO-queue new elements are inserted/enqueued at the end of the list instead of at the beginning/head of the list. This may seem as a minor difference but it complicates the implementations.

9.2.1 isEmpty

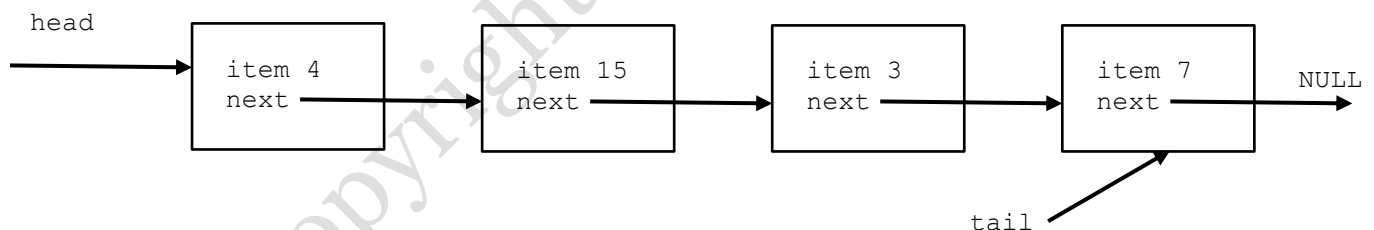
If we keep the queue/list ordered so that the oldest element is at the head of the list and the most recently (newest) inserted element is at the end of the list we can use the same method/function for the `isEmpty` method/function as for the LIFO-queue.

9.2.2 insert/enqueue

The problem with the `enqueue` operation is how to find the end of the list where new elements are to be inserted to preserve the FIFO-order in the queue. A naïve implementation of `enqueue` would search for the end of the queue starting from the beginning/head.

```
naiveEnqueue (element head, itemType item)
    if(head == null) head = new element(item)
    else
        element p = head
        while(p->next != null) p = p->next      //search for the last element
        p->next = new element(item)
```

The problem with the `naiveEnqueue` method/function is the search for the last element in the queue. The search quickly becomes inefficient as the number of elements in the queue grows. To avoid having to search for the last element of the queue/list we can introduce a reference/pointer to be able to access the last element directly. We call this reference/pointer `tail`.



To implement a more efficient `enqueue` operation on the queue we need to update the `tail` reference/pointer properly.

```
void enqueue(itemType item)
    if(head == null)                //insert to an empty queue
        head = tail = new element(item)
    else
        tail->next = new element(item) //insert to a non-empty queue
        tail = tail->next
```

9.2.3 remove/dequeue

We could potentially use exactly the same method/function for the `dequeue` operation as for the LIFO-queue. However, to make sure we avoid loitering/dangling pointers we should remember to reset the `tail` reference/pointer to `null` when we remove the last element from the queue.

```
itemType dequeue()
if(head == null) error handling
else
    itemType x = head->item
    head = head->next
    if(head == null) tail = null //update tail if the last element is removed
    return x
```

In the course book⁹ there is a nice description of how to build single linked lists to implement stacks and FIFO-queues on pages 142-154 (a PDF of this chapter is found here <https://algs4.cs.princeton.edu/13stacks/>) The following assignment builds on this.

⁹ The complete book is available online together with many other resources relevant for the course at <https://algs4.cs.princeton.edu/>

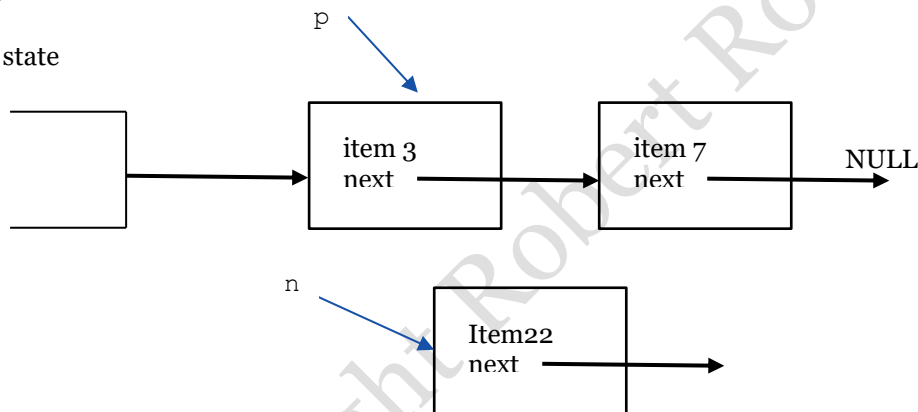
9.3 Assignment 3a: Single linked lists

Assignment 3a:

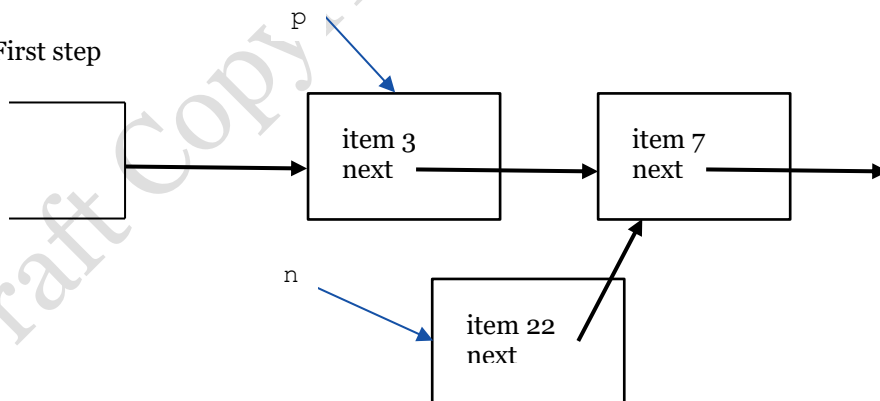
- 1) Write code that implements a single linked list implementing a stack (i.e. a LIFO queue) based on the code from the book (see above). The code should have a method to insert a new element and a method to remove and return the first element in the list. For the sake of simplicity you can assume that the data stored in the elements are integer numbers.
- 2) Implement a method to print the data in the elements in the list (without removing elements). For instance if you were to print the list in the example above you should print: 4 15 3 7
- 3) Create a method to insert a new element (pointed to by the reference **n** in the example below) after an element in the list pointed to by a reference **p**. The first step is to set the **next** reference in the element pointed to by **n** to point to the same element as the next reference in the element pointed to by **p**. The second step is to make the **next** reference in the element pointed to by **p** set to point to the element pointed to by the reference **n**.

Example:

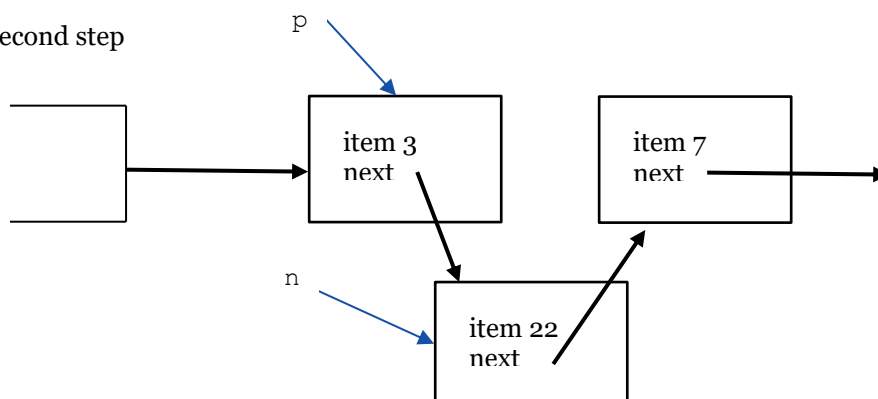
Initial state



First step



Second step



- 4) Write a method to insert new elements sorted in ascending order in the list according to the values of the data stored in the elements. Inserting elements in such an order implements an example of a simple priority queue¹⁰.
- 5) Implement test code in `main()` to test your implementation.
- 6) Create test files, i.e. files containing data that you can use to test your implementation
- 7) Try to change the declaration of the data in an element by adding `static` to the declaration: `static int data;` What happens if you insert a five different integer values to the queue and then prints the elements in the queue? Explain!

¹⁰ A priority queue is a queue for which the element with the highest priority in the queue is always returned when an element is removed (dequeued) from the queue.

9.4 Double linked circular lists

Operations on single linked linear lists can be a little bit tricky to implement. For instance it may require that you use auxiliary pointers when inserting elements in an ordered fashion and they may require that one handle special cases for the first and last elements a little bit differently.

If we try to follow the proposed method to analyse and model problems (i.e. Polya's method see Section 17) the first step is to understand the problem:

In this case we have already identified the problem as that in the case of an empty list, we have to handle the first and last element as special cases. So the problem is if it is possible to find a structure where we do not have to take special consideration to the first/last or in other words the beginning/end of the list?

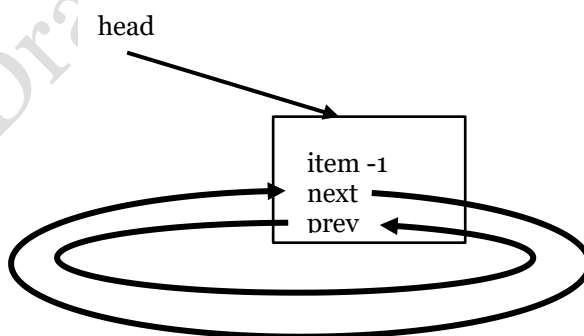
A structure which do not have a beginning/end is a circle. A list could be implemented as being circular. However, we must be able to identify the start/end of the list, i.e. the first and last elements of the list to be able to implement FIFO and LIFO queues. A possible solution is to build a circular list with a special element which denotes the start/end of the list. This special element does not contain any data and is only used for the purpose of being a marker for the start/end of the list. An empty list will only contain this special element. Such an element is often referred to as a **sentinel element**. We also introduce a second reference/pointer in the elements of the list which refers to the preceding/previous element in the list to facilitate inserting/removing elements to/from the list.

What we have identified is that for single linked lists we may have to handle special cases for inserting and removing elements at arbitrary positions in the list as well as having an extra reference/pointer to the end of the list in case we implement a FIFO queue. Double linked circular lists is an example of how we can remove some of the special cases at the expense of using more memory (for the extra reference/pointer in each element) and having to update more references/pointers when inserting/removing elements.

What can we learn from this simple analysis and solution?

Best practice: *If possible one should try to avoid having to handle special cases in code. In other words: **Always strive for simplicity!** Simple designs/code will be easier to understand, maintain and will contain less errors/bugs. To come up with simple designs/code requires a thorough understanding of the problem and how it can be solved. Thus, it always pays off in the end to spend some extra time and care on the design!*

The smallest double linked circular list contains only one element where the `next` and `prev` references point to the element itself:



Whether or not one should use the element pointed to by `head` as only a marker of where the circular list ends/starts (referred to as a **sentinel element**), or as a regular element of the list where we can

store data, is an issue that there are different opinions on. I prefer to use this element only as a marker, as it otherwise would create unnecessary problems on how to update the `head` reference/pointer if we try to insert a new element at the beginning of the list (i.e. think of how you would handle the case if we tried to insert a new element which should be the new start/end of the list, i.e. replacing the element in the figure above. In which case the `head` reference/pointer should point to the new element instead of the element containing -1).

Inserting a new element in a list after another element involves resetting four references/pointers:

- 1) Set the `next` reference of the new element to point to the same element as the `next` reference in the element after which the new element is to be inserted
- 2) Set the `prev` reference of the new element to point to the element after which it is inserted
- 3) Set the `next` reference of the element after which the new element is to be inserted to point to the new element.
- 4) Set the `prev` reference in the element pointed to by the `next` reference in the new element to point to the new element.

Example: Assume we insert a new element after the element pointed to by `head`. Assume we have a reference `n` which points to the new element. In pseudo code the operations will look like this:

```
n->next = head->next
n->prev = head
head->next = n
n->next->prev = n
```

Assignment 3b: Double linked circular list

- 1) Extend the class implementing elements (nodes) for the single linked list to implement a circular double linked list with a sentinel element. That is add an extra reference `prev` and change the method to instantiate a new object from the class so that both the `next` and `prev` references point to the newly created object, i.e. a new object is in itself a minimal circular list.
- 2) Implement a method, `insertAfter()`, to insert a new element into the list after an element pointed to by a reference. This method should have two parameters, a reference to the element after which the new element is to be inserted and a reference to the new element.
- 3) Based on the `insertAfter()` method implement methods to insert an element as the first element in the list, `insertFirst()` and as the last element in the list, `insertLast()`. Done properly, these methods need only to have one line of code.
- 4) Assume the data in an element is an integer value that can be used as a key. Implement a method which inserts new elements to the list sorted in ascending order by the value of the data/key.
- 5) Implement a method to print the content of the list. Start from the `head` element (the first element in the list) and traverse the list following the `next` references until you reach the element pointed to by `head` again.
- 6) Implement a method to remove an element from a list. This method need only one parameter, a reference to the element being removed. Return a reference to the element removed. The `prev` and `next` references in the removed element should be set to point to the removed element. You may need to think extra on how to handle the case if one tries to remove the `head` of the list while the list contains other elements also.
- 7) Implement test code in `main()` to test your implementation
- 8) Create test files, i.e. files containing data that you can use to test your implementation

Hint: When trying to understand how and in which order to set the references it is really helpful to draw pictures explaining step by step how to do this.

9.5 Comparing single linked and double linked lists

We have seen two different linked list implementations: single linked and double linked lists. So how should we select between them?

9.5.1 Memory usage

In terms of memory usage the double linked list uses two extra (auxiliary) references/pointers (`prev` and `next`) per element while a single linked list only uses one extra reference/pointer (`next`). Both use a reference/pointer to identify the start of the list (`head`). The single linked list use an extra pointer to identify the last element (`tail`) of the list while an empty (sentinel) element is used in the double linked list. Thus if we look at a list with N elements:

- a single linked list will use: 1 `head` reference/pointer, 1 `tail` reference/pointer and one `next` reference/pointer per element of the list (i.e. in total N `next` references/pointers).
A total of $(2+N)$ references/pointers
- a double linked list will use one `head` reference/pointer, 1 sentinel element and two references/pointers (`next` and `prev`) per element in the list (i.e. in total N `next` and N `prev` references/pointers).
A total of $(1+2+2N)$ references/pointers, that is $(3+2N)$.

When N grows large we can omit the constant terms as they will become insignificant for the total cost. So we estimate a double linked list to use double the amount of auxiliary references/pointers compared to a single linked list (i.e. $2N$ references/pointers compared to N references/pointers).

9.5.2 Execution time

When the lists are used to implement FIFO- and LIFO-queues we can access the first (`head`) and last (`tail`) elements directly for both single and double linked lists. The time consumed to insert/remove an element at either end of the list will in the case of a single linked list be that of having to update one or in the case of the element being the last (`tail`) two references to insert/remove an element to/from the queue. For a double linked circular list we update four reference/pointers. Thus the time used to update the references/pointers in a double linked list will be double or quadruple compared to if a single linked list is used.

9.5.3 How to choose the proper list implementation

In the example of the linked list implementations, the double linked list will use more memory (references/pointers) and it will have to update more references/pointers for operations on the lists. However, the double linked list implementations have less special cases to consider.

Best practice: Best practice is to always start by as simple design/implementation as possible. This will make the design easier to understand and less error prone. Memory or execution time optimizations should be left to when one has determined that it is necessary to improve the performance of the application. And one has to make certain that one can correctly identify where the optimizations should be performed in the design/code and that the proposed optimizations/improvements will lead to the expected result.

So if the choice is between the use of a single linked or a double linked list you should start by selecting the one which you understand and feel comfortable to use.

Best practice: Use abstraction layers! To facilitate to change the implementation of algorithms and data structures you should use abstractions. By this we mean that you should avoid hard-coding things into your solutions. For instance in the case of using a single or double linked list for the implementation of a FIFO queue you should not code things as `insertSingleLinkedListFIFO()` or `insertDoubleLinkedListFIFO()`. Rather use an abstraction layer which hides the details of the implementation. Either as abstract data types in object oriented languages or as an extra layer of methods/functions such as `insertFifo()` which in turn may call either `insertSingleLinkedListFIFO()` or `insertDoubleLinkedListFIFO()` depending on which type of implementation you want to use. By such an approach you only need to change at one (or possibly a few places) to change the implementation. Similarly you should use possibilities to define your own names for existing datatypes such as `int`, `Int`, `float`, `Float` etc. Consider an example where you are to code an application for handling distances. Initially you might want to use meters as the unit for the lengths and only represent distances as integral meters. Thus you probably select `int` as data type. There are two drawbacks with such an approach: 1) you probably have lots of variables of type `int` in your code. And to see that a specific `int` is used to hold a distance is harder than if you had defined a new name such as `distance` for the type `int` and declared variables as `distance`; 2) if you at a later stage discover that you will have to accommodate distances that are not only measured in integral meters but as meters and centimeters you may want to change the datatype for the distances to be of type `float`. If you have defined your own name for the data type or your own datatype for distances you can more easily change the representation.

10 Complexity

How much memory a data structure/algorithm uses is called **memory complexity**. Likewise the execution time used is referred to as **time complexity**. Often we can omit many details as it often is sufficient to understand how fast the memory usage and/or execution time grows as the problem size grows to be large. This means that if we can analyse an algorithm/data structure and define memory and/or time complexities as a mathematical function of the problem size, $f(N)$ (the problem size is called N as a convention) it is sufficient to only consider the fastest growing term as the less fast growing terms will be insignificant for large problem sizes (i.e. large N s).

What the problem size is and what operations we are interested in differs depending on what type of algorithm/data structure we are studying. For a queue (ex. LIFO, FIFO etc.) we are most likely interested in how long time it takes to insert or remove a single element from a queue with N elements and how much memory is used for either a single element or possibly for N elements. For a sorting algorithm we are instead interested how much memory is used to sort N elements and how long execution time is needed to sort N elements.

For complexity we can be interested in at least four different measures of complexity for memory and execution time:

1. Worst case
2. Best case
3. Average or probabilistic performance bounds
4. Amortized complexity

Worst case describes the maximum memory usage or execution time to solve a problem of a specific size. A worst case analysis gives a strict upper bound performance guarantee. That is it can be used to determine what the largest problem size is that we can be guaranteed to solve within a specific time or within a specific amount of memory. It is the most commonly used complexity measure.

Best case describes the best performance, i.e. the minimum memory usage or minimum execution time to solve a problem of size N . It may be interesting to understand the best cases for several reasons.

If one can prove that the worst case and best cases are the same within a constant factor we call the algorithm optimal. Note that optimality in this sense does not imply that there are no other algorithms that can be faster or use less memory to solve the same problem.

Moreover, if we know that the best case will occur frequently or that the input is such that the algorithm will exhibit near best case performance it could be suitable to select an algorithm with a good best case performance for this type of input even if there are other algorithms for the same problem with better worst case performance but worse best case performance. Examples of such algorithms are found for instance in sorting where there are algorithms which can sort nearly sorted data in very short time (good best case) but which have much worse worst cases than other algorithms.

Average or probabilistic performance bounds describe what performance we could expect on average or for most problems. There are commonly used algorithms for example for sorting, searching and for priority queues which will exhibit very good performance for most input one could expect to encounter in practice, but that have bad worst cases for specific types of input. Such algorithms/data structures are often used in applications where we can tolerate that the algorithm behaves poorly for a few cases which are unlikely to occur. Such algorithms can be used for many applications except for hard real-time systems. For example Quicksort is a much used algorithm with a probabilistic performance guarantee (i.e. we can expect it to be fast in most cases) which is one of the most

important sorting algorithms. This kind of performance bound can be determined either by applying statistical analyses or by experimental evaluations.

Amortized complexity is normally used for time complexity and applies to algorithms/data structures where most operations are fast and only a few are slow. Amortized complexity bounds are similar in vein to average complexity with the difference that an amortized complexity gives a strict performance guarantee. A common example for algorithms with this type of performance bound are algorithms with two types of operations: low cost and high cost where $(N-1)$ low cost operations are performed per each high cost operation. The amortized (or average) complexity per operation is thus $((N-1) * \text{cost of low cost operation} + 1 * \text{cost of high cost operation})/N$. Thus the amortized cost per operation is approximately: $\text{cost of low cost operation} + \text{cost of high cost operation}/N$. Thus the cost of the high cost operations is amortized (spread out) over many low cost operations. Resizing arrays are examples of algorithms with this type of performance bound. They are applicable for applications which can tolerate that some, few individual operations are costly (take long time). Which holds for most applications apart from hard real-time systems.

10.1.1 Notations for complexity

A commonly used notation for complexity is asymptotic notation. In asymptotic analysis we are only interested in approximating how fast the complexity grows when the problem size grows large. That is we identify the fastest growing term of the analysis of the algorithm and remove all other terms and any constants. For the worst case this is called big-Oh notation (sv. stora ordo), the best case is called big-Omega.

The memory complexity (overhead memory used) for a queue with N elements for the single linked list and double linked list, the big-Oh and big-Omega are $O(N)$ and $\Omega(N)$ for both data structures. That is they are both on the order of N .

If the worst and best cases are within a constant of each other, then the algorithm/data structure has big-Theta meaning that it is optimal. In this case they have memory complexity which also is big-Theta, $\Theta(N)$. Far from all algorithms/data structures have big-Theta defined.

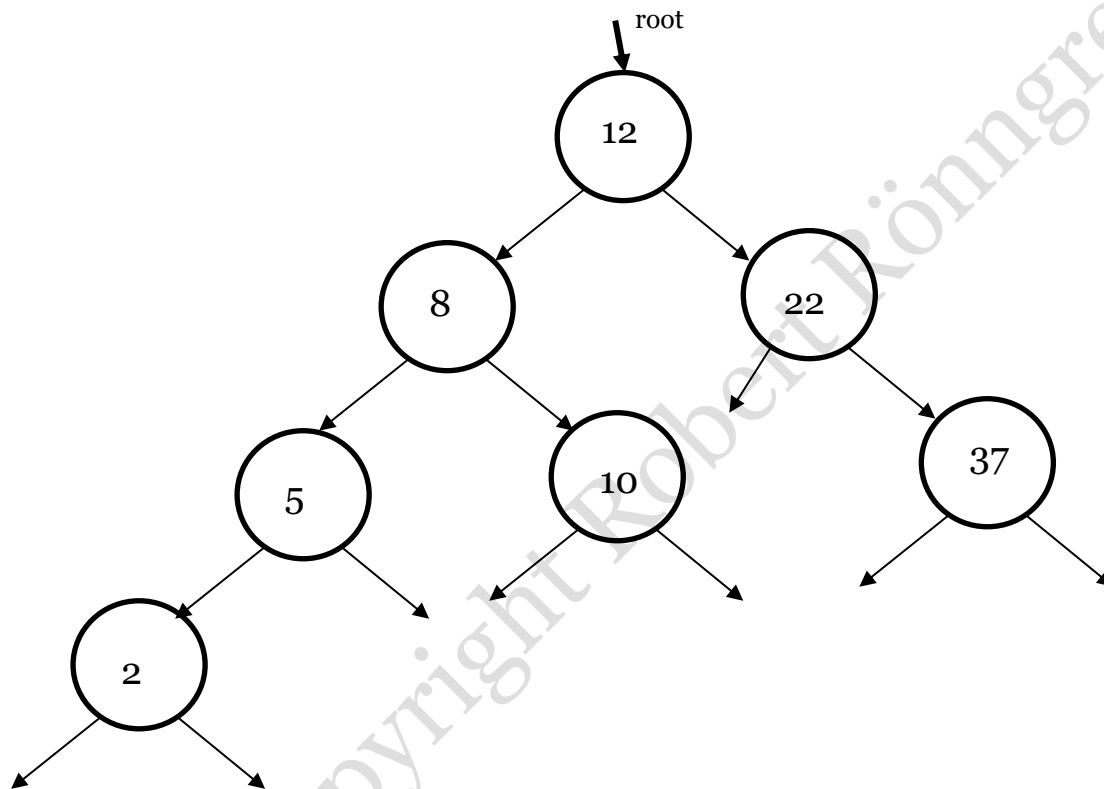
There are some things to notice about this notation:

- It is only possible to classify algorithms/data structures by this notation. It says nothing about the relative performance of algorithms/data structures with the same worst/best cases. In the example of the linked lists we cannot determine that the double linked list uses more memory than the single linked list as they fall in the same complexity classes for both best and worst case.
- Big-Theta is not the same thing as average or amortized complexity
- The complexity definitions only hold for large problem sizes (large N). An implication of this is for example that an algorithm which is $O(N^2)$ may be faster than an $O(N)$ algorithm for problem sizes smaller than N .
- It says nothing about how large N is (the complexity analysis may hold for different sizes of N for different algorithms)

An alternative notation (which we will not use in this course) is Tilde notation where the constant of the fastest growing term in the analysis is kept. Thus the memory overhead for a queue with N elements is $\sim N$ for the single linked list while it is $\sim 2N$ for the double linked list. The advantage being that by using tilde-notation it is possible to compare the complexity for algorithms/data structures even if they would belong to the same big-Oh classes.

11 Binary search trees

A binary search tree¹¹ is a linked data structure where each node (element) can have up to two descendants normally named `left` and `right`. The first node in a tree is called the `root`. In a binary search tree all nodes also contains a `key` by which the nodes in the tree are ordered. The normal ordering of the nodes in a binary tree is that all nodes in the left sub-tree of the root has keys which are smaller (in some sense) than the key in the root, while all elements in the right sub-tree of the root has elements with keys larger or equal to the key in the root. This order should hold for all sub-trees in the tree. Below is an example of an ordered tree.



Trees are interesting by many standards. Trees can allow the user to efficiently look-up/search for elements in the data structure and have inspired the design of some of the most efficient comparison based sorting algorithms known, i.e. Quicksort.

The easiest way to implement operations on trees often is to implement recursive methods. A recursive method is a method which calls itself.

You will encounter many problems in “programming” courses which are easiest, and in many cases most efficiently, solved by recursive methods. In real life we often use recursion to solve problems such as sorting a deck of cards without ever spending a second thought on that we use recursion. However, when we learn how to program, we often start by learning how to program sequential solutions without recursion. This often unconsciously prevents us from finding the most natural solutions to problems which may well be recursive solutions. So what properties does a description of an algorithm have which indicates that the algorithm (solution to the problem) could be recursive?

If you discover that you can describe a solution to a problem in a way that you can handle part of the problem directly and that the remaining part of the problem, which now is **smaller** than the original problem, can be solved in the same way as the original problem – then you have described a recursive solution.

¹¹ Binary trees have at most two branches/sub-trees connected to a node. In general trees may have any number of branches connected to a node.

Recursion is similar in many ways to proofs by induction. The main difference is that while proofs by induction starts by a base case and then performs an induction step typically to prove that if a property $P(N)$ is true it is also true for $P(N+1)$. Recursion start by a large problem and move towards smaller sub-problems until it meets a base case and the recursion terminates.

That each problem solved in each iteration (recursive call) should be smaller than the problem from where the recursive call is made is important. If this does not hold the recursion will not terminate as we never reach the base case(s) where we end the recursion.

A method to insert an element in an ordered tree can be described in pseudo code as:

```
insert (root, newNode)
  if newNode->key < root->key then
    if root->left == NULL then root->left = newNode    //Base case
    else insert(root->left, newNode)
  else
    if root->right == NULL then root->right = newNode  //Base case
    else insert(root->right, newNode)
```

So how can we show that the problem is smaller in each recursion-step in the above method/function? We know that we cannot have trees of infinite depth, nor should there be any circular structures in a tree. Thus when we do a recursive call we will traverse the tree along some branch and we know that the depth of the sub-tree (the sub-tree of the parameter `root` in the method/function) will be less than the depth of the tree we came in to the function with. Thus we will eventually reach a node which has no sub-tree where we will insert the new node.

When traversing a tree to process its content there are three distinct ways of doing it referred to as *prefix*, *infix* and *postfix*. These names refer to in which order we process the content of a node in comparison to when we process the left and right sub-trees of the node respectively.

Prefix traversal can be described in pseudo code as:

```
prefixTraversal(root)
  if root != NULL then
    process root->data
    prefixTraversal(root->left)
    prefixTraversal(root->right)
```

Infix traversal can be described in pseudo code as:

```
infixTraversal(root)
  if root != NULL then
    infixTraversal(root->left)
    process root->data
    infixTraversal(root->right)
```

Postfix traversal can be described in pseudo code as:

```
postfixTraversal(root)
  if root != NULL then
    postfixTraversal(root->left)
    postfixTraversal(root->right)
    process root->data
```

Assignment 4:

- 1) Assume the processing of data is to print the key. Show what the output would be if the example tree above is traversed in prefix, infix and postfix order.
- 2) Show how a binary search tree would be built if you insert data that are: i) un-ordered; ii) ordered in ascending order; iii) ordered in descending order
- 3) Create a class that can be used to implement a binary search tree where the keys are integer values. When a new object is instantiated the `left` and `right` references should be set to `NULL`
- 4) Implement a method to insert elements in the binary search tree.
- 5) Implement methods to traverse and print the key values in the nodes to `stdout` which traverses the tree in prefix, infix and postfix order respectively.
- 6) Implement a method to search for a key in your binary tree. The method should take a reference to the root and an integer key value to search for as parameters. If the key is found in the tree then the method should return `TRUE` else it should return `FALSE`.
- 7) Implement test code in `main()`
- 8) Create test files, i.e. files containing data that you can use to test your implementation

Note: There are two problems associated with binary search trees:

1. As you should have noticed by the assignments above a simple binary tree may become unbalanced and in the worst case deteriorate to a linked list. This means that if we build binary search trees with the intention to perform effective searches in the data in the tree we may not get the fast logarithmic search times we strive for if the tree is not well balanced. There are other, more complex trees structures where the tree is always built/kept balanced at little additional cost. The two most commonly used balanced trees are AVL-trees and Red-Black Trees.
2. It is inherently difficult (impossible?) to implement efficient removal of elements in search trees.

A final question is if trees always have to be built as binary trees, that is that a node only may have [0, 1, 2] descendants. The answer to this is that there is nothing that prevents a tree to be built so that a node may have an arbitrary number of descendants. If such trees are effective or not depends on the application for which they are used.

12 Queues and resizing arrays

Queues, such as stacks or LIFO queues, and FIFO-queues can be implemented by not only linked lists but also by arrays. The easiest queueing strategy to implement by an array is a stack or LIFO-queue.

12.1 Stack/LIFO-queue implemented by an array

To implement a fixed size LIFO-queue by an array is straightforward. The basic idea is to allocate an array holding a fixed number of elements of the type of elements we will store in the LIFO-queue. Elements will be added to the LIFO-queue from low indices, starting from index zero and upwards. Elements will be removed from the highest non-empty index. If we try to add yet another element to an array which is already full, an error will occur which means we should throw an exception if we program in JAVA.

Pseudo code for the operations to create the queue, insert an element and remove an element is as follows. We assume the data type of the elements is “element”

```
element stack[]
int top = 0.           //next empty index in array
int size = 0           //size of the stack

void createStack(stackSize)
    element stack[] = new array(stackSize)
    size = stackSize

void insertElement(element e)
    if(top < size) stack[top++] = e //insert element at top and increment top
    else error                     // stack is full

element removeElement()
    if(top == 0) return null       // empty stack return null (alternatively
                                   // throw exception
    else return stack[--top]      // last element inserted is at index top-1
```

To understand how this works we encourage you to draw a figure of a minimal stack that could fit in the range of 2-4 elements of type int and update it as you insert and remove elements.

One problem with the array based stack implementation compared to a linked list implementation is that the array based implementation has a fixed size and it may be hard to determine in advance how large the stack could become (i.e. how many elements we could add). A solution to this problem is to base the stack on a resizing array, that is the size of the array is implemented to adapt to the amount of data inserted into it.

12.2 LIFO-queue implemented by resizing arrays

To remove the limitation of having a fixed sized array we can “resize” the array to fit how many elements we insert in the array. In this case resizing means that we allocate a new array of appropriate size and copy elements from the old array to the new and replace the old array by the new.

We can identify two cases when we should resize the array:

- **Grow:** The old array becomes full and we will need to replace the old array with a larger new array to accommodate the growth of the LIFO-queue (stack). The elements from the old array are copied to the new array.
- **Shrink:** If elements are removed from the old array (LIFO-queue) then the old array may become unnecessarily large (i.e. have unnecessarily much unused space), in which case we should allocate a new smaller array and copy the elements from the old array to the new array.

There are several issues to consider if we should arrive at an effective design. Both growing and shrinking the array involves copying elements from the old array to the new array. Copying/moving data is very time consuming and should be avoided or at least kept to a minimum to keep performance within tolerable limits. Thus, to design an effective implementation of the resizing array we need to consider the following:

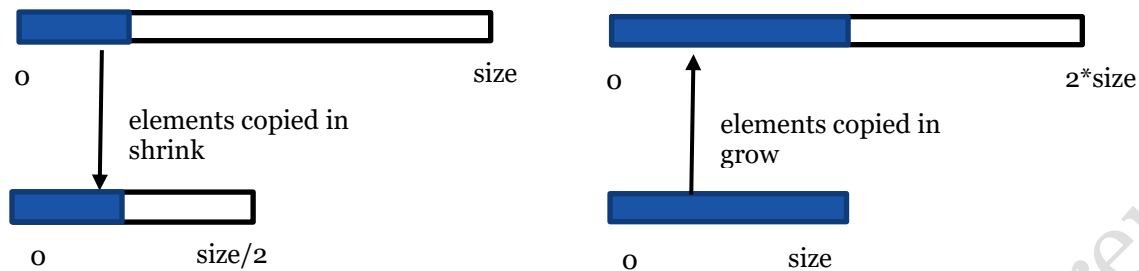
1. When we grow the array – how much should the size be increased?
2. What should trigger that the array is shrunk? And how much should the size be reduced?
3. Which elements to copy from the old array to the new and how they should be copied.

For the resizing of the array we will use a simple principle which is the basis for many efficient algorithms: **To halve or double**. When we grow the array we double its size and when we shrink the array we halve the size.

The trigger to grow the array is that the array is full when we attempt to insert yet another element. What we should use as a trigger to shrink the size of the array is not that obvious. To find a good trigger we need to use a simple technique to understand the algorithm – we should consider different operating conditions. We cannot halve the size of the array unless it is less than half full. So what if we trigger a halving (shrink) of the array when it becomes half full? This is actually a strategy that will fail to achieve reasonable performance. The reason is simply that if we remove one element from the queue it may trigger a halving of the array, if this is immediately followed by that we insert an element into the queue we will have to increase the array size. That is it would introduce a worst case scenario which would be a series of interchanging grow, shrink, grow, shrink... resulting in copying of all elements in each operation. This would lead to extremely poor performance.

Thus the problem is that if we use a half-full array as trigger for halving the size of the array is that it leaves no empty space to grow the queue. The solution is to only halve the array size when the array is quarter-full. This means that we will have half the array empty after a shrink – a situation similar to what we have after we have triggered a grow. This will effectively remove the aforementioned worst case of repeated, consecutive shrink and grow resizes.

What elements should we copy from the old array to the new array and where should they be placed in the new array? In a LIFO-queue (stack) all elements of the queue will be placed from array index zero and upwards in a continuous sequence. This makes the copying simple as we can copy all elements used from the old array to the new.



What remains to settle is how many elements the array should be able to hold from the start (and possibly define a minimum size). We should select this size as a power of two, typically 8,16,32 or 64 not to run into future problems when resizing the array.

Before we show the updated pseudo-code for the resizing array implementation of the LIFO-queue (stack) we should notice that we can use the same method for both growing and shrinking the queue.

```
minStackSize 16      //constant for the min stack size
element stack[]
int top = 0.         //next empty index in array
int size = 0         //size of the stack

void createStack()
    element stack[] = new array(minStackSize)
    size = minStackSize

void insertElement(element e)
    if(top < size) stack[top++] = e //insert element at top and increment top
    if(top == size) resizeStack(2*size) //stack is full - grow size

element removeElement()
    if(top == 0) return null        // empty stack return null (alternatively
                                    // throw exception
    if(top < size/4 && size >= 2*minStackSize) resizeStack(size/2)
    return stack[--top]             // last element inserted is at index top-1

resizeStack(int newSize)
    element newStack[] = new array[newSize]
    int i
    for(i=0; i< size; i++) newStack[i] = stack[i]
    stack = newStack
    size = newSize
```

Note: We have assumed that we always can allocate a larger array in case we need to grow the size of the array (queue). Since physical memory is limited and a process (a program executing) may have further limitations on how much memory it can use, we should check that we actually can allocate a larger array and otherwise handle the error (in many cases the runtime system or the operating system will terminate the process if it runs out of memory – i.e. it might generate an error we cannot handle)

12.2.1 Analysis

So how costly is it to base a LIFO-queue (stack) on resizing arrays? What we want to assess is the cost of copying elements. We will investigate two cases: i) how many times, on average, will an element in the queue have been copied if we start inserting elements into the queue from an empty queue to a queue with N elements; and ii) if we remove elements from a stack of size N to empty it.

If adding element $N+1$ will trigger a grow of the stack then after the grow: $N/2$ elements will have been copied once, $N/4$ elements will have been copied twice, $N/8$ elements will have been copied three times, ... etc. Assuming the cost to copy an element is C this will result in an average cost per element (N elements, i.e. we divide the sum by N to get the cost per element) is: $C \sum_{i=1}^N i / 2^i$ which sums to $2C$ as N goes to infinity. This means that on average an element will not be copied more than twice. The analysis for a shrinking queue is similar. That is in a worst case scenario the cost to insert or remove an element from the LIFO-queue is bounded by a constant. Thus the worst case (execution) time complexity is $O(1)$ to insert or remove an element from the queue (stack).

In terms of memory complexity the analysis is simple, in the worst case after the array has been shrunk we use three empty indices per full index in the array. Again this gives a constant bound and the memory complexity (worst case) which is also $O(1)$.

Note: the cost to resize the array (grow/shrink) is substantially larger than for operations where we do not have to resize the array. When we calculate the overhead per operation for the resize operations we spread the cost of resize operations over many low cost operations. Hence this analysis is an example of an analysis for an algorithm/data structure with an amortized complexity.

12.2.2 Comparison to a linked list implementation

If we compare the performances of a linked list implementation to the array based implementation of the array based implementation of a LIFO-queue we see that:

- Both have constant access times for insertion/removal of elements
- Both use extra memory per element that is bounded by a constant
- An array has its elements placed in sequence in the memory and could thus be assumed to exhibit better locality than the linked list where elements can be more dispersed in memory
- The cost of an operation in a linked list is constant while most of the operations in the array based implementation has a constant cost (and are fast) the cost for an operation involving a resize has a time complexity that is $O(N)$
- In terms of memory usage we would assume the difference to be small

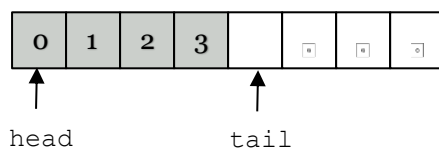
Thus we could expect the array based implementation to be faster for many applications though it may not be suitable for applications with hard real-time requirements. To determine more precisely when one or the other implementation is a better choice one should do performance measurements on the system where it is to be used, as performance may depend on the hardware, operating system and programming language used.

12.3 FIFO-queue implemented as a fixed size circular buffer

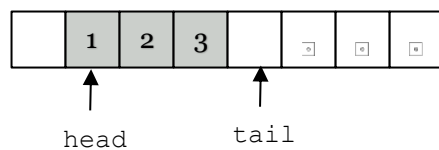
If we know how to implement a LIFO-queue by an array then it should be easy to implement a FIFO-queue by an array or could it be more complicated? There is actually one major difference between LIFO and FIFO-queues and that is that in a LIFO-queue all operations, insertion/removal of elements, operate on one end of the queue while in a FIFO-queue removals operate on the head end and insertions at the tail end.

To understand why this is a problem consider the following example: Assume we have a fixed array size of 8 indices and that we initially insert four elements to the queue. Then we start doing mixed sequences of removals and insertions. After having done eight insertions the tail end of the queue has reached the end of the array (index 7) while the removals have freed-up indices at the beginning of the array (indices 0-3). Thus there is free space in the array while we cannot continue inserting elements at the end. Resizing the array to double the size would not solve the problem but temporarily.

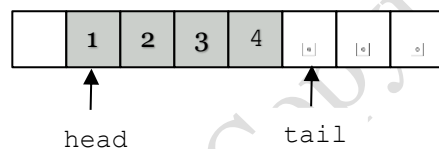
Example: insert four elements to an eight space queue



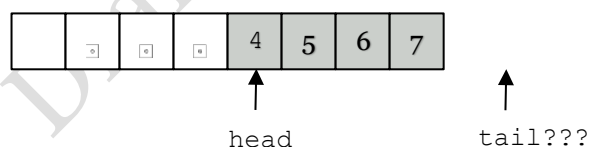
Remove first element



Insert an element

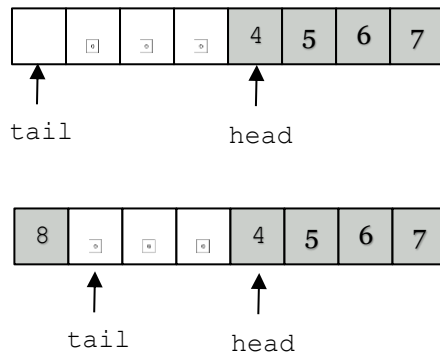


After three additional removals and insertions we would have four indices free but it is not obvious how to insert more elements.



The trick to being able to use the free indices (space) is to treat the data structure as a circular data structure by updating the head and tail modulus the size of the array.

Thus we would update tail as $(tail + 1) \% arraysize$. In the case above adding a new (9th) element to the queue we would update tail as: $tail = (tail + 1) \% 8$ which is zero (0).



Before moving on to pseudocode we notice that the elements in the queue might not be placed on consecutive indices which is something to bear in mind if a circular FIFO-queue is to be implemented by resizing arrays.

Pseudocode for a circular FIFO-queue of fixed size (also known as a circular buffer)

```

element queue[]
int head = 0, tail = 0.
int nrElements = 0          //to facilitate we keep track of the present
                             //number of elements in the queue
int size = 0                //size of the array (queue)

void createQueue(queueSize)
    element queue[] = new array(queueSize)
    size = queueSize

void insertElement(element e)
    if(nrElements >= size) error //attempt to insert to a full queue
                                //(alternatively throw exception)
    queue[tail] = e              //insert element at tail
    tail = (tail + 1) % size     //update tail
    nrElements++                //update number of elements in the queue

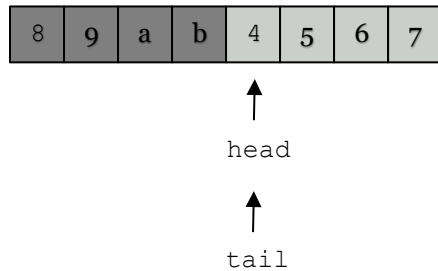
element removeElement()
    element e
    if(nrElements == 0) return null // empty queue return null (alternatively
                                    // throw exception)
    nrElements--
    e = queue[head]              // oldest element in the queue
    head = (head + 1) % size     // update head
    return e

```

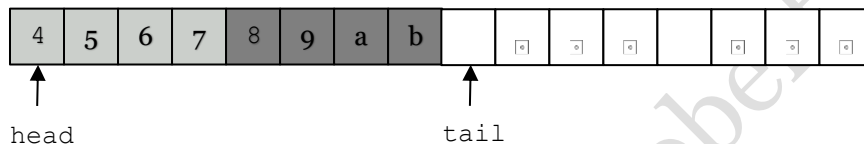
12.4 Circular FIFO-queue implemented by resizing arrays

When implementing a FIFO-queue by a resizing array the main problem compared to the implementation of a LIFO-queue is to place the elements in the correct order when resizing the array. The problem arises as the elements in the queue may not be placed in the same physical order (on indices) as the logical order (in FIFO-order).

Consider the following example of growing the size of the queue consisting of the elements (in FIFO-order) [4,5,6,7,8,9,a,b]:



After a resize to double the size of the array the elements should be placed as:



Pseudocode for a FIFO-queue based on resizing array:

```
minQueueSize 16          //constant for the min queue size element queue[]
int head = 0, tail = 0.
int nrElements = 0        //to facilitate we keep track the present number
                           //of elements in the queue
int size = 0              //size of the array (queue)

void createQueue()
    element queue[] = new array(minQueueSize)
    size = queueSize

void insertElement(element e)
    if(nrElements >= size) resizeQueue(2*size) //full queue - resize to grow
    queue[tail] = e                          //insert element at tail
    tail = (tail + 1) % size                  //update tail
    nrElements++                             //update number of elements in the queue
```

```

element removeElement()
    element e
    if(nrElements == 0) return null // empty queue return null (alternatively
                                    // throw exception)

    nrElements--
    e = queue[head]                // oldest element in the queue
    head = (head + 1)%size         // update head

if(nrElements < size/4 && size >= 2*minQueueSize)
    resizeQueue(size/2)
    return e

resizeQueue(int newSize)
    element newQueue[] = new array[newSize]
    int i
    for(i=0; i< nrElements; i++) //place elements in FIFO order from index 0
        newQueue[i] = queue[head]
        head = (head+1)%size
    queue = newQueue
    head = 0
    tail = nrElements
    size = newSize

```

13 Ordering and sorting

In many applications we order elements. Ordering may be implicit such as in the FIFO and LIFO-queues. In these queues we order elements by the order in which they are inserted to the queues. These orders are implicit in the sense that there is no key in the elements which we can inspect to determine in which order the elements should be stored in the queues. In an explicit order we would have keys associated with the elements that can be used to determine the order.

13.1 Orders

From childhood we learn how to order things such as numbers, text strings etc. by their natural order. In programs we may have to order also other types of elements for which a natural order may be less obvious such as data about persons (should we order by name or social security number or by income...) or lists of inventories. So how do we define an order as a mathematical entity.

To be able to order elements we want a **total order**, which is what interfaces such as `compareTo()` in JAVA should implement. A total order should fulfill the following:

- Antisymmetric: $\forall x, y: (x \leq y \ \&\& \ y \leq x) \rightarrow x = y$
- Transitive: $\forall x, y, z: (x \leq y \ \&\& \ y \leq z) \rightarrow x \leq z$
- Total: $x \leq y \ || \ y \leq x$

A special case which is worthwhile considering is when all keys¹² (keys are what we use in an element to compare by) are unique, that is if x and y are different elements the following hold: either $x < y$ or $y < x$. This may be important when selecting an appropriate sorting algorithm.

13.2 Measures of the ordering of elements/sortedness of a set of elements

When understanding how sorting/ordering algorithms work and to be able to reason about their effectiveness we need some kind of measure on how ordered/sorted a set of elements is. We define an inversion as two elements (a,b) which in relation to each other are sorted out of order. Example (i): the array below has two inversions: (2,1) and (3,1).

2	3	1	4
---	---	---	---

Another example (ii) with inversions (4,2), (4,3), (4,1), (2,1), (3,1):

4	2	3	1
---	---	---	---

We can use the number of inversions as a measure of the orderliness/sortedness of a set of elements. When a set is sorted the number of inversions will be zero (0). Measures of order/un-order are also referred to as entropy.

¹² A key and an element may be the same thing as in the case of elements which are integer numbers or it could be a field or a number of fields of an element such as name and social security number of an element representing a person

Sorting algorithms are (normally) based on swapping, exchanging the place, of two elements. Considering example (ii) above, we can observe that in some cases we can remove several inversions by swapping two non-adjacent elements (adjacent elements are placed on indices i and $i+1$ in an array, non-adjacent elements are placed on indices i and $i+a$ where $a > 1$). This means that sorting algorithms which can swap non-adjacent elements could be more efficient than algorithms which only can swap adjacent elements.

13.3 Basic properties of sorting algorithms

Two important properties of sorting algorithms are:

- Stability
- In-place

13.3.1 Stability

Stability means that the algorithm preserves the relative order of elements with identical keys. Consider the following example:

b_1	a_1	c_1	a_2
-------	-------	-------	-------

When a stable sorting algorithm is used to sort the array above it is guaranteed that the relative order between elements with identical keys are preserved. For this example the result of applying a stable sorting algorithm will always be:

a_1	a_2	b_1	c_1
-------	-------	-------	-------

However if we applied a non-stable sorting algorithm the result could be either of the following:

a_1	a_2	b_1	c_1
-------	-------	-------	-------

or

a_2	a_1	b_1	c_1
-------	-------	-------	-------

Worth noting is also that if an algorithm has the stability property or not is of no consequence if:

- All keys are unique
- Or if we cannot distinguish between two elements with equal keys. An example is when sorting integer numbers in which case we cannot distinguish on 4 from another 4 or one 17 from another 17

Typically library algorithms to sort *reference types* (objects such as `String`) in JAVA use stable sorting algorithms while sorting algorithms for *primitive types* (example: `int`) often are non-stable¹³.

13.3.2 In-place

A sorting algorithm which is in-place use at most $O(\log(N))$ extra memory to sort input of size N .

¹³ The fastest (for most cases) general purpose sorting algorithm, Quicksort, is not stable

13.4 Comparison methods/functions

While we can use operators such as `<`, `>`, `<=`, `>=`, `==` and `!=` to compare numbers in JAVA and C we have to use methods/functions to compare more complex data types (reference types in JAVA). These methods/functions, such as `compareTo()` methods in JAVA or `strcmp()` in C (`strcmp()` is one function of several used to compare text strings) typically return the following:

```
if(a < b) return -1 // (or a negative number)
if(a > b) return 1  // (or a positive number)
else return 0      // a == b
```

So why do these methods/functions return such numbers? The simple reason is that comparisons often can be implemented by subtraction.

To implement a comparison method for integers adhering to the convention above is very simple:

```
int cmpInt(int a, int b)
    return a - b
```

If we compare text strings we can build on the same idea. Characters are normally encoded by an industry standard called Unicode where a character is encoded as 8-bit (UTF-8), 16-bit (UCS-2) or in some cases 32-bit (UTF-32) integers. A string is stored as an array of characters. This means that to compare strings we can subtract character by character from the two strings we compare until we either get a result which is non-zero or we reach the end of one string or we reach the end of both strings (in which case the strings are equal). Example: compare the strings `abcd` and `abef`:

```
  a b c d
-a b e f
-----
  0 0 -2
```

The comparison in this example stops when the subtraction returns a non-zero number, i.e. when we find `c - e = -2` in which case the result (-2) is returned.

13.5 Priority queues

A priority queue is a queue using explicit priorities (keys) of the elements (in contrast to FIFO and LIFO-queues which use implicit priorities). A priority queue is defined so that it always returns the element with the highest priority present in the queue when an element is removed from the queue.

Different conventions are used to name the insert and remove operations. Insert are commonly called `insert()` or `enqueue()`, while the remove operation can be called `delMax()` or `dequeue()`.

Priority queues are used in many contexts such as for: process/thread scheduling, scheduling events in simulations, graph algorithms etc. In many (most) applications, such as process scheduling and event scheduling, a high priority has a low numerical value.

In many applications, such as scheduling, it is desirable that the priority queue preserves FIFO-order among elements with equal priorities. As an example we can consider scheduling processes for execution. Typically there are a limited number of priorities for processes (Linux use priorities 0-40) and for the scheduler to be fair we expect it to select the longest waiting process within a priority to execute, i.e. FIFO-scheduling within each priority.

13.5.1 Priority queue data structures

There are many data structures implementing priority queues. This is not surprising as they are an important part of many applications and one can often optimize them to take advantage of the specifics of the applications where they are used. Some of the best data structures are described briefly below.

Linked list: To implement a priority queue by a linked list is straightforward. When inserting an element (`enqueue`), the element is inserted sorted in decreasing priority order. That is the list is traversed from the head to insert the element at the correct place. When removing an element from the list (`dequeue`) one simply removes the first element in the list which can be done in constant time ($O(1)$). The drawback of a linked list implementation is that the worst case is that the whole list has to be traversed when inserting a new element which makes the insertion (`enqueue`) operation $O(N)$.

A linked list implementation is a good choice for small queue sizes (< 50 -100 elements) even though there are data structures/algorithms with better worst case performances. This observation also holds for many other types of applications. The big-Oh, big-Omega and big-Theta complexities only tells us what approximate performance to expect for large problem sizes. We encounter this in many situations in real life also. If you are to visit a neighbour living on the other side of the street where you live, it is probably faster to cross the street walking rather than retrieve your bike from a storage room and cross the street by bicycle. However, for larger problem sizes such as visiting a friend at the other end of the city the bicycle will be faster than walking.

Heap: A heap data structure is a binary tree structure where each node may have 0-2 descendants. The heap order is that each node has higher priority than its descendants. That is the highest priority element will be at the root. It cannot be used for searching as a binary search tree. In a BST we know whether what we search for should be searched for in the left or right subtree, while the heap order does not say anything about what subtree to search in. Heaps can be kept perfectly balanced and can be effectively implemented using arrays. However, the basic implementation do not exhibit the best locality and there are implementations that has better locality such as the B-heap.

Heap and stack are words also used to denote memory areas in the memory space for processes. While the execution stack is implemented as a stack (LIFO-queue) the memory area of a process memory space referred to as the heap (used to dynamically allocate memory for objects etc.) is not implemented as a the heap data structure described here.

Splay tree: A splay tree is a self-balancing tree structure based on what is referred to as “splaying” which has a fast and stable $O(\log(N))$ performance for both `enqueue()` and `dequeue()` operations. Splay tree is a good choice for a general purpose priority queue.

Calendar queue: Is a heuristic design of a data structure which bears similarities to how hash tables with separate chaining is implemented. As for hash tables it will exhibit a near $O(1)$ amortized access time for both `enqueue()` and `dequeue()` operations if the priorities are not very unevenly distributed. That is the $O(1)$ complexity is a probabilistic performance bound meaning that in most cases it is what we will experience. However, it has resize operations that are quite time consuming so it does not lend itself for applications with hard real time requirements.

Skip list: A Skip list is a probabilistic data structure. It can be characterized as a circular single linked list with additional pointers implementing a probabilisticly balanced search tree. The average access time is $O(\log(N))$ for both `enqueue()` and `dequeue()` operations. It is interesting since it has been used to implement distributed, fault tolerant, peer-to-peer services such as distributed hash tables.

13.6 Sorting algorithms

In this section we will describe the ideas behind three sorting algorithms that are commonly used and which will cover most of the situations where one has need for sorting. We will assume that the elements to sort are stored in an array and that they should be sorted in ascending order by the keys used.

13.6.1 Some basics: `swap()` and how to generalize a sorting method

Sorting elements in arrays can be done in two different ways: i) we could swap elements in the array; or ii) possibly allocate a new array and copy/move elements to the correct indices in the new array. The first approach uses less memory, i.e. it is in-place, which may be important when sorting large volumes of data.

Sorting elements in an array is based on exchanging/swapping elements in the array. This method/function is often called `exchange()` or `swap()`.

```
void swap(element array[], int i, int j) //swap elements at indices i and j
    element tmp = array[i]              //in the array
    array[i] = array[j]
    array[j] = tmp
```

Sorting is based on comparing elements according to their natural order. Elements that we compare should have data types that are comparable (example: we can compare cars as cars while they in reality probably would have different types/models/brands). For numbers (and individual characters) we would use the operator `<` for comparisons, i.e. perform checks such as `a < b`. However, this operator is not necessarily, depending on what programming language we use, defined for datatypes such as classes (JAVA) or `structs` (in C). For a library it is infeasible to implement specific versions for each data type for a sorting algorithm, in particular as we cannot know which user defined data types, such as classes/`structs` the user will implement.

The solution to this problem is to perform comparisons by calling a method/function to compare elements. That is instead of implementing comparisons in the algorithms as `a < b` we will call a method/function `less(a, b)` to do the comparisons¹⁴.

In JAVA we solve this problem by having classes implement the `comparable` interface which means that objects will have methods with standardized names by which one can compare an object to another object. This technique, by which the sorting method calls methods in the object it is sorting, is referred to as a *call-back mechanism*. In C one uses another solution where a pointer/reference to the method to compare elements that should be used by the sorting algorithm to compare elements is provided as a parameter to the sorting method/function.

¹⁴ The method for comparisons need not be called `less()` but it is a name which self-explanatory.

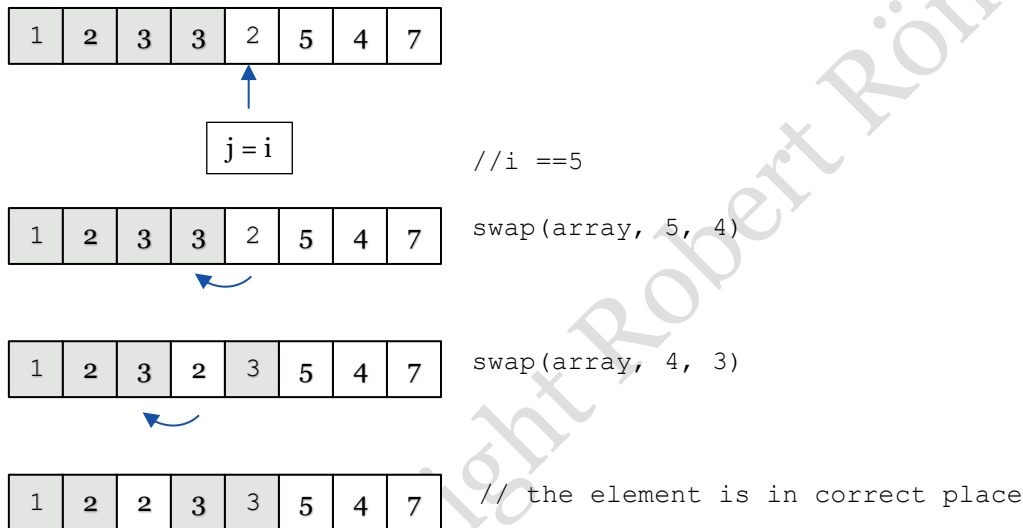
13.6.2 Insertionsort

Insertionsort is a relatively simple/intuitive sorting algorithm which is in-place and stable. It has a worst case performance of $O(N^2)$ but for data with few inversions (i.e. nearly sorted) it will sort the data in near $O(N)$ time. It is faster than most sorting algorithms for small data sets ($N < \sim 10$ elements). Thus it is a good choice for sorting small data sets and for data with few inversions.

The idea behind insertion sort is to keep the elements at low indices sorted and work towards higher indices and sort element by element to the correct place. A basic version of the algorithm is:

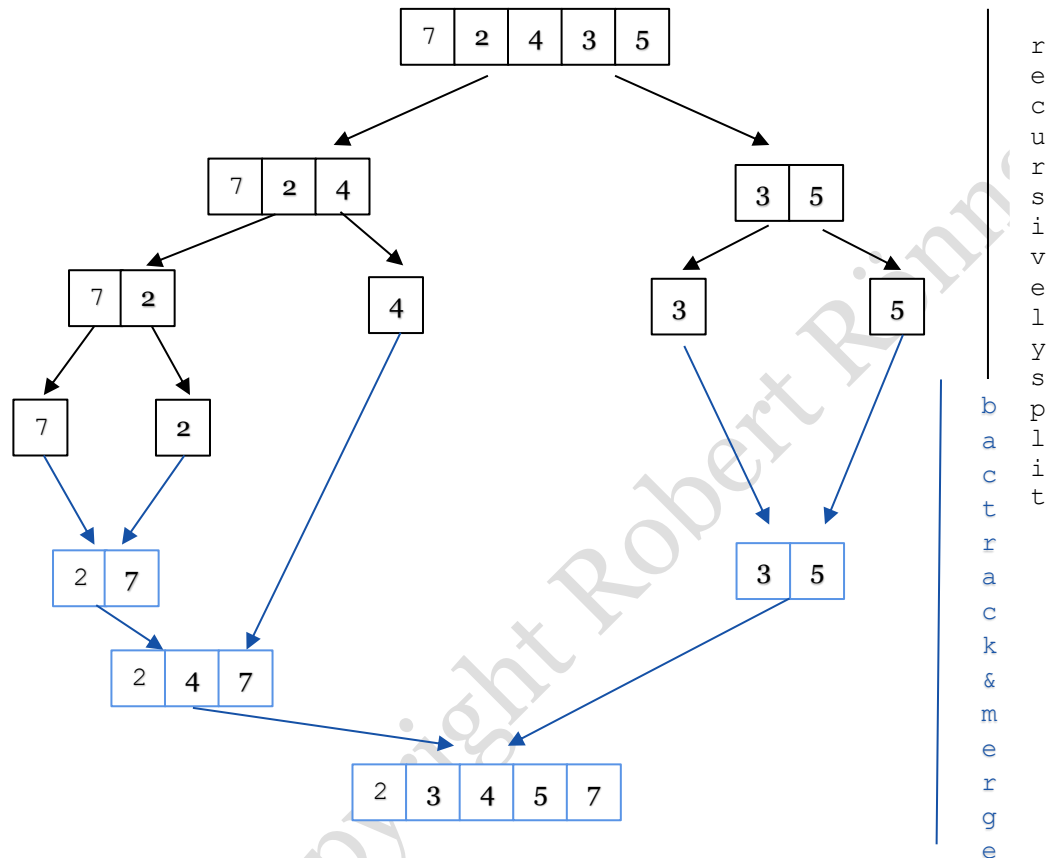
```
void insertionSort(element array[], int arrayLength)
    int i, j
    for(i = 1; i < arrayLength; i++)
        for(j = i; j > 0 && less(array[j], array[j-1]); j--)
            swap(array, j, j-1)
```

We show one step in the sorting of an array when sorted by insertionsort:



13.6.3 Mergesort

When discussing resizing arrays we encountered the idea of doubling/halving arrays and we noticed that this is an idea on which other types of algorithms can be based. Mergesort is an algorithm based on this observation. In its purest form mergesort sorts elements by recursively dividing the array into smaller and smaller subarrays until each subarray is sorted, i.e. contains one element. It then backtracks the recursion and successively merge adjacent subarrays into larger and larger sorted subarrays until the whole array is sorted. We illustrate mergesort by an example:



We should observe the following. When we merge subarrays we will in practice merge the subarrays by placing the elements in a new array of the same length as the array that is sorted. Hence, mergesort is not in-place but uses $O(N)$ extra memory. Merge sort is stable. It is not effective to continue subdividing the array into very small subarrays, instead we should change sorting strategy for short (less than ~ 10 elements) subarrays and sort them with insertionsort. Insertionsort will be faster for short arrays and it is stable so it will preserve the stability of mergesort.

As mergesort is stable it is often the primary choice of algorithm to implement library sorting methods for complex data types, such as objects in JAVA.

Theoretically one can show that lowest number of comparisons needed to sort N elements by a comparison based sorting algorithm is $O(N \log(N))$ ¹⁵. It is possible to show that the worst case of mergesort is $O(N \log(N))$. That is we cannot find a comparison based sorting algorithm which can sort any data with less comparisons. In this respect mergesort is optimal. However, the computer hardware on which we execute algorithms are not perfect and to achieve the best possible performance we need good cache performance. Mergesort does not have the best locality and it uses $O(N)$ extra (auxiliary) memory so there is room for improvement.

¹⁵ The actual limit is $\lg(N!)$ which is $\sim N \lg(N)$ for large N

13.6.4 Quicksort

Quicksort is in practice the fastest general purpose sorting algorithm. It can be in-place as it is possible to implement not using more than $O(\log(N))$ extra memory. It has better locality and consequently exhibits better cache performance than mergesort. It has a probabilistic execution time performance which is $O(N\log(N))$. This means that for the vast majority of inputs it will have $O(N\log(N))$ execution time but it could perform much worse, $O(N^2)$, for specific types of input which are not very probable to occur in practice. Quicksort is not stable. Based on these properties it is the algorithm used to implement the library sorting method for primitive data types in JAVA.

Quicksort is based on the principle to divide the array into smaller subarrays when sorting. However, it is based on another principle than mergesort. By this principle it is not guaranteed to sub-divide the array by halving the size. This is what can cause the execution time to grow faster than $O(N\log(N))$ for some specific types of input.

To understand the idea behind how quicksort works we need to understand how a binary search tree (BST) works. In a binary search tree elements with keys less than the key of the element in the root are found in the left sub-tree while elements with larger keys are found in the right sub-tree. That is the data set in the tree is partitioned into two sub-sets. For the tree to be perfectly balanced each root in any sub-tree should have a key which is the median key of the set of keys found in all elements in that sub-tree.

If we translate the analysis of a binary search tree into a sorting algorithm we will get quicksort.

In quicksort we select an element to be the element to partition the elements in the array by. Elements with a smaller key than that of the partitioning element are placed in the left-hand side (low indices) sub-array and other elements in the right-hand side (high indices) sub-array. We then recursively partition the left and right sub-arrays to sort the whole array. Observe that the partitioning element will be placed at the correct, final index in the array and need not be further sorted.

Some issues occur with this approach:

- To get a good partitioning (which gives us a partitioning which is close to halving the array) we need to select a partitioning element which is close to, and ideally is, the element with the median key of the elements in the sub-array. Observe that failure to select reasonably good partitioning elements will be detrimental to the performance of the algorithm. (If we understand what sequences of data can cause a binary search tree to become unbalanced we can see what could cause problems for quicksort)
- What should we do with elements that have the same key as the key of the partitioning element?
- For short sub-arrays we should switch to (cut-off) to insertion sort as it is more effective for short sub-arrays

To precisely identify the median element is too costly as it would mean that we would have to check all elements in the sub-array. A simple, yet effective approach (among many suggested), which will yield good results in most cases is to select the partitioning element as **the median-of-three**. That is, we select the partitioning element as the element with median key of the first, last and middle element in the sub-array.

The second problem, how to handle elements with identical keys to that of the partitioning element identifies another problem. To understand what the problem is we could consider an array where all elements have identical keys. Such an array is sorted. However, quicksort as described above would not be able to take advantage of this fact as it would continue to sub-divide the array recursively with

all elements going into one sub-array leading to $O(N^2)$ performance. A solution to this problem is to implement a **three-way partitioning**. Three way partitioning will divide the array into three sub-arrays: i) in the left-hand subarray (on low indices) one place elements with keys strictly less than that of the partitioning element; ii) in the middle sub-array elements with keys equal to that of the partitioning element is placed (note these are placed in the correct, final place in the array); iii) elements with keys strictly larger than that of the partitioning element will be placed in the right-hand sub-array (on high indices). When the partitioning is done we will recursively continue sorting the left-hand and the right-hand subarrays respectively.

Three-way partitioning illustration

keys < partitioning key	keys == partitioning key	keys > partitioning key
-------------------------	--------------------------	-------------------------

Finally we exemplify how quicksort with median-of-three selection of partitioning element works without touching the details on how the partitioning algorithm works in detail:

Median of 7,3,1 is 3

7	2	4	3	6	5	1
---	---	---	---	---	---	---

swap miss-placed elements (7,1) and (4,3)

1	2	3	4	6	5	7
---	---	---	---	---	---	---

Partition and sort the left-hand sub-array (1,2)

1	2	3	4	6	5	7
---	---	---	---	---	---	---

Partition the right-hand sub-array (4,6,5,7), median of 4,6,7 is 6,
swap miss placed elements (6,5)

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Partitioning the left-hand sub-array (4,5) and the right-hand (7) completes the sorting

13.6.5 Non-comparison based sorting

Sorting is for most of us intrinsically intertwined with performing comparisons. But in the discussion on the optimality of mergesort we indicated that there may be sorting algorithms which are not based on comparisons. While this may be hard to imagine there are many special cases where we can exploit properties of the elements/data types/data sets that we sort to design non-comparison based sorting algorithms. One such example is how we can sort N integer numbers in $O(N)$ time.

Assume we have an array containing N integer numbers with values in the range of $[0, K]$. The algorithm works by counting the frequencies (the number of occurrences) of each unique integer number in the array. Pseudocode for the algorithm is as follows:

```
frequencySort(int array, int K)
    int frequency[K+1]           //note we assume all elements are zero
    int i, j, k=0
    for(i=0; i< K+1; i++)
        frequency[array[i]]++    //found an occurrence of the value array[i]
    for(i=0; i< K+1; i++)
        if(frequency[i] > 0)
            for(j=0; j < frequency[i]; j++)
                array[k++] = i
```

14 Searching

Searching means finding an element with a specific key in a data set. We assume we search in data with $\langle \text{key}, \text{value} \rangle$ -pairs. In a random data set we would have to perform a brute force search checking each element from the first and onwards. This means that we would have to search $N/2$ elements on average to find the element we are searching for while we would have to inspect all N elements in the worst case. This is way too time consuming to be a workable solution. We are used to fast searches and we even expect a search in all data found on the Internet to return a result nearly instantaneously. So how could we achieve this? The solution lies in preparing the data we want to search into clever data structure which supports fast searches. This is not a new idea, books have had alphabetically sorted indices which allows us to rapidly look-up on which pages in the book we will find what we search for.

We should also note that in the index of a book we find only one occurrence of a key (keyword). If there are more than one occurrence of the keyword in the text there are not several entries for the keyword in the index but a list of all pages where the keyword occurs. In the following we will assume that a key is unique in the data structures, i.e. there will only be one entry for a key in the data structures (cp. book index). If a key is associated with several we can solve this by having for example a linked list containing these elements (cp. list of pages in the entry for a keyword in the index of a book).

14.1 Binary search

If we have the data we search in sorted in an array the most effective search is a binary search. Again this builds on halving the interval of the data we search in. The algorithm works as follows: Compare the key we search for to the key in the element ($\langle \text{key}, \text{value} \rangle$ -pair) on the middle of the array. If the key in the middle element is less than the key we search for we continue our search in the right (high indices) subarray else we search in the left (low indices) sub array. With this method we will find what we are searching for in $O(\log_2(N))$ time.

Pseudocode for an iterative binary search returning the index where the key was found or in case the key is not found, it returns the index where the key would have been found if it had been in the array. That is for this implementation of a binary search one need to check the element on the index returned to see if this $\langle \text{key}, \text{value} \rangle$ -pair has the key we searched for.

We assume the `compare()` method/function returns `compare(a,b)`: $a < b$ negative, $a == b$ zero, $a > b$ positive:

```
int bSearch(array a[], int arraySize, key k)
    int lo = 0, hi = arraySize - 1
    while (lo <= hi)
        int mid = lo + (hi-lo)/2
        int cmp = compare(key, array[mid])    //cmp: negative, 0 or positive
        if(cmp < 0) hi = mid - 1               //search low indices subarray
        else if(cmp > 0) lo = mid + 1          //search high indices subarray
        else return mid
    return lo
```

Note that an $O(\log_2(N))$ algorithm is substantially faster than a brute-force linear search. For a data set of one billion entries (2^{30}) we would find what we search for (or determine that the key does not exist in the data) by 30 comparisons ($\log_2(2^{30}) = 30$) while a brute-force linear search would need to do half a billion comparisons on the average. Keeping data sorted also allows us to effectively find ranges of keys and associated values (eg. the n^{th} to $n+k$ keys) and other so called ordered operations.

The drawback of keeping data sorted in an array is that in the case we want to add a new element ($\langle \text{key}, \text{value} \rangle$ -pair) we have to move a lot of data. Note that we need not sort all data again which would be a waste of time. To insert a new element/key we simply have to search for the place where

the new element should be inserted (with a binary search which is $O(\log(N))$) and then move all data after this point to one index higher. On average we would have to move $N/2$ of the elements. This is time consuming and may also create a need to increase the size of the array in which case all elements have to be copied to the new array.

14.2 Binary search trees

Instead of doing a binary search in a sorted array we could build a data structure based on a binary tree. A binary search tree is an ordered data structure and supports ordered operations such as range queries. It also allows for simple insertions of new elements to the data structure. However, a standard binary search tree has two main disadvantages:

- It can easily become unbalanced
- It is hard (impossible?) to implement efficient deletions of elements in the queue which preserves the balance

For operations on a binary search tree (searches, range queries, insertions etc.) to be $O(\log(N))$ the tree must be kept balanced (i.e. all branches from the root to a leaf have no more than $\log(N)+1$ nodes). However, simple binary search trees can easily become unbalanced resulting in a worst case performance of $O(N)$. If you attempt to insert a sequence of keys in ascending or descending order you will quickly see how this problem arises.

The problem of keeping the tree balanced is solved by algorithms/data structures for balanced trees such as AVL or Red-Black trees. These data structures/algorithms maintain the balance of the tree by reordering elements on insertions to keep the tree balanced.

However, there is no solution to do effective deletion of elements in any type of binary search tree.

So to summarize, balanced binary search trees allow for efficient ordered operations such as range queries which occur in many application. This it should be the choice for applications that need efficient range queries.

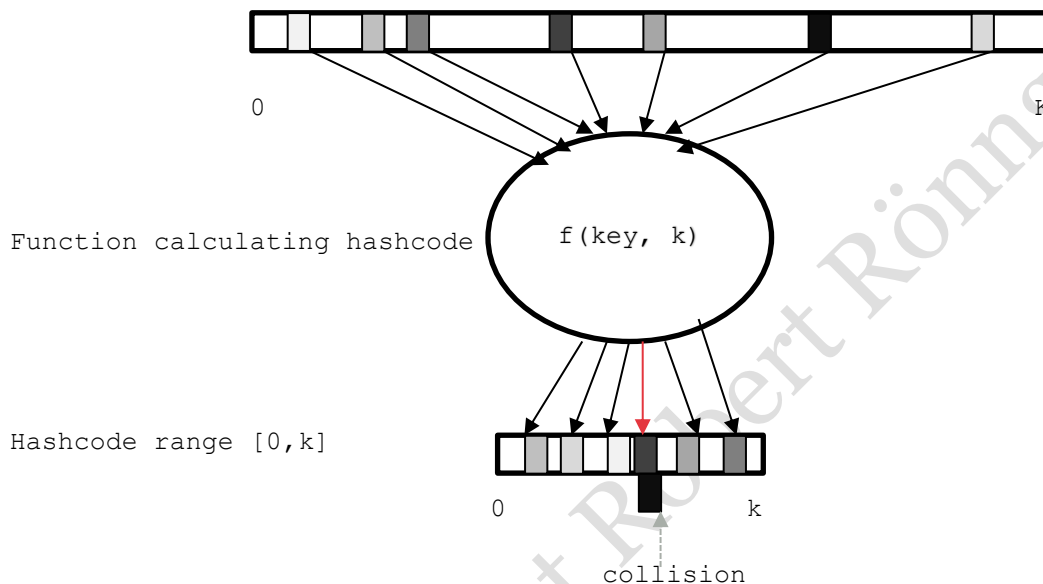
14.3 Searching by array indexing – hash tables

When we were discussing sorting we noticed that we, for some data types/sets, could do sorting without comparisons. The question is if we can do the same for searching? Could we find what we search for by indexing into an array instead of searching by doing comparisons?

If we had access to a memory which was infinite we could use any type of key as an index into a very large array. While this may seem like a strange idea at first glance we should notice that anything we store in the memory of a computer is stored as a sequence of bits with values zero or one. How we interpret such a sequence of bits is up to us and we could interpret any such sequence as a (huge) binary integer number which we could use to index into an array. In such case we could find the entry associated with a key in constant time by simply using the key as index into the array. This is of course not realistic as we do not have infinite amounts of memory. But we can borrow ideas from this to achieve near constant time searches (look-ups).

The observation we should make is that while keys may fall in a large range $[0, K]$ where K can be a very large number, this range is in many cases sparsely populated. That is the total number of keys in the range are in many cases not that high compared to the size of the range. Thus it should be possible to map the keys from the large range $[0, K]$ to a smaller range $[0, k]$ by some function $f()$ where $k < K$ or in many cases $k \ll K$. **We call the mapped value of the original key a hashcode.** Ideally each hashcode should be unique or at least there should be few keys that map to the same hashcode.

Original key range $[0, K]$



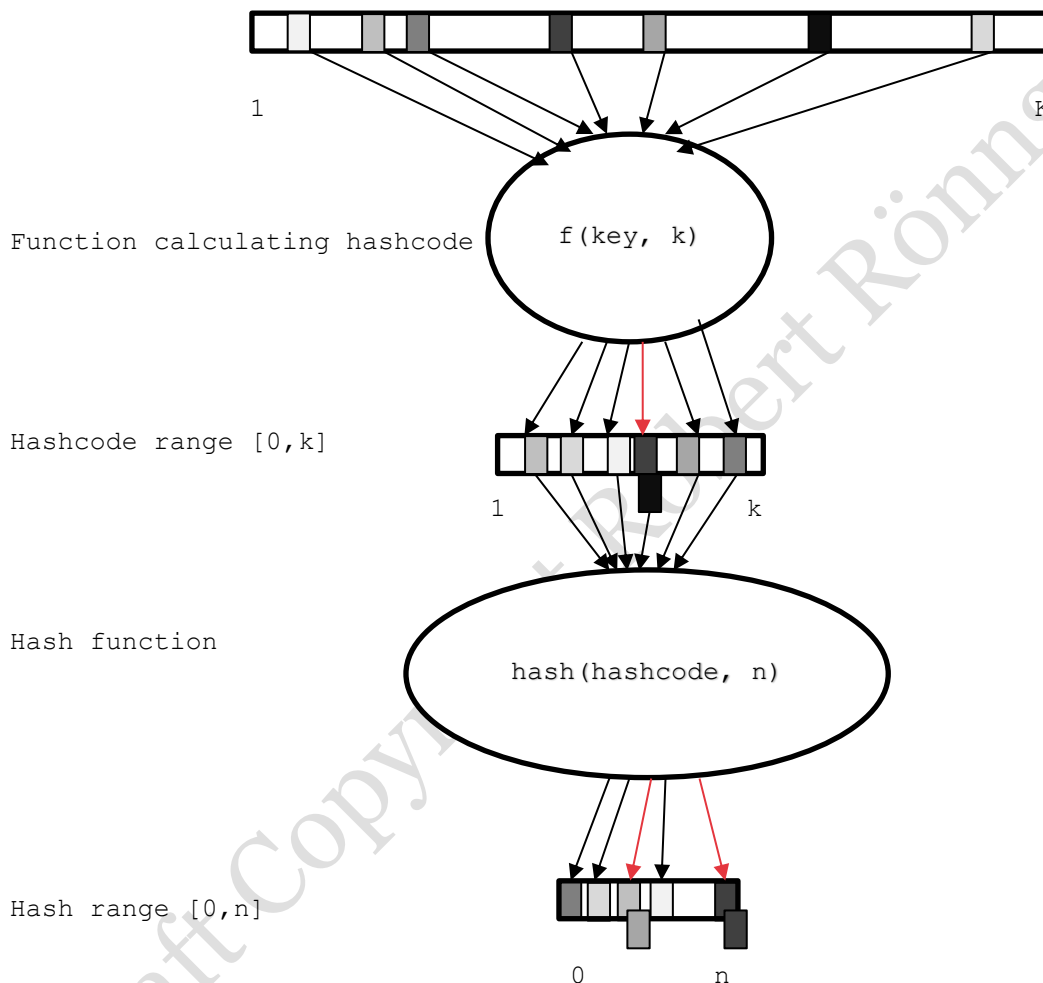
If possible we want to avoid or at least have few collisions (a collision is when more than one key is mapping to the same hashcode). The desired property of the function calculating the hashcode is that it should spread the hashcode mappings evenly over the whole range of hashcodes. This is referred to as uniform hashcode.

The way we calculate the hashcode depends on the data type and is likely to be different for different data types. However, they should use as much of the original key (as many of the bits of the key) as possible when encoding the hashcode. The range of the hashcodes is often limited to a of integer numbers. In JAVA hashcodes are 32-bit integer values. Calculating the hashcode for an object can be relatively expensive. Hence, for JAVA built-in data types, the hashcode for an object is only calculated when the hashcode for that object is used by the program¹⁶. Moreover, to avoid having to redo the calculation of the hashcode if it is used again, the hashcode is stored in the object. This is a technique called **software caching** where increased memory usage is traded for execution time efficiency.

¹⁶ To access the hashcode of an object one calls a method in the object. This method has a flag which tells if the hashcode has been calculated or not. When the method is called for the first time the flag is not yet set and the hashcode will be calculated and stored in the object while the flag is set. In consecutive calls the flag will be set and the saved hashcode will be returned

So can we use the hashcodes as indices into an array? The answer is yes in theory but no in practice. If we were to use 32-bit numbers (as the size of the hashcodes is in JAVA) we would have to allocate an array with four billion entries (2^{32}) which is very (too) large for many (most) practical uses. So for practical uses we do a second mapping from **hashcode** to **hash** and use the hash as index into a smaller array. Further reducing the range of possible values will increase the probability that more keys are mapped to the same hash. Hence it is important that the hash-function is designed to achieve uniform hashing (spreading the possible hash values evenly in the range, in the example below the range $[0, n]$) Moreover, the hash-function should be relatively easy (short execution time) to compute as we normally do not software cache the hash value for an element.

Original key range $[0, K]$



14.3.1 Hashing with separate chaining

So finally we have laid the groundwork for describing what is known as **hashing with separate chaining**.

The idea is quite simple. We create an array of linked lists of size n designed to hold $\langle \text{key}, \text{value} \rangle$ pairs. We only allow one instance of a key to be present at any time in the data structure. We should realize that the `value` associated with a `key` in itself could be whatever we choose such as a number, a reference to an object or a data structure such as a list containing several values associated with the `key`.

The reason for each entry in the array to be a list is to be able to cater for collisions, in this case more than one hashcode mapping to the same hash. Each linked list hold information on the $\langle \text{key}, \text{value} \rangle$ -pairs mapped to the index in the array.

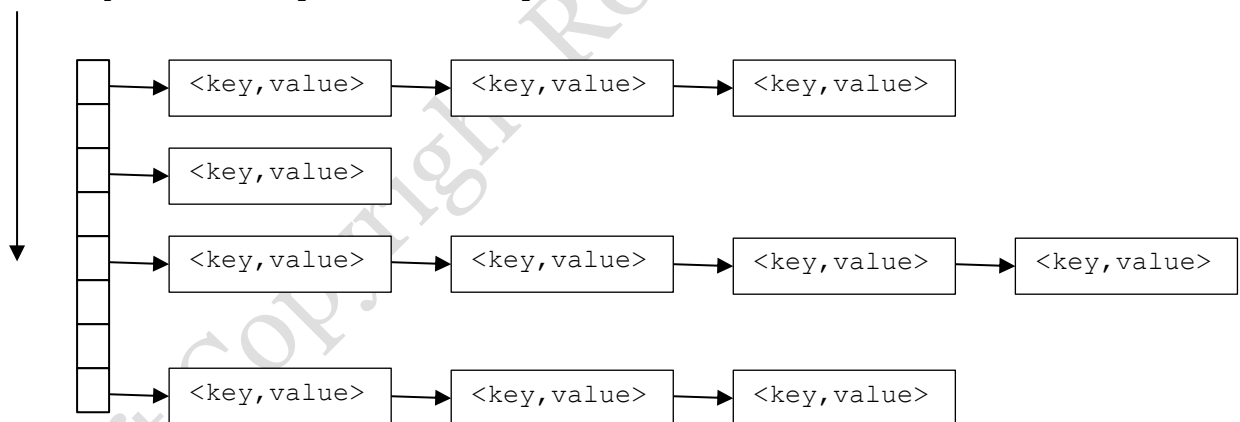
If the lists on average are short we can do very fast look-ups of keys. When doing a look-up we simply calculate the hash from the hashcode for the key, use the hash to index into the array and then search in the list for that index to see if we find an element ($\langle \text{key}, \text{value} \rangle$ -pair) with the key we search for.

When we insert a new element into the data structure we calculate the hash from the hashcode for the key and insert the $\langle \text{key}, \text{value} \rangle$ -pair using the hash as an index in the array. The $\langle \text{key}, \text{value} \rangle$ -pair is inserted into the list of the index with the hash value. When inserting the $\langle \text{key}, \text{value} \rangle$ -pair into the list we search it and if we find another $\langle \text{key}, \text{value} \rangle$ -pair with the same key we replace that with the new $\langle \text{key}, \text{value} \rangle$ -pair.

For the look-up in the hash table to be fast it is essential that the average length of the linked lists are short. Preferably they should not be longer than a few elements, typically 4 or less. Our ability to achieve this heavily depends on the hash function resulting in a uniform hash (i.e. uniform distribution of the hash values in the range $[0, n]$ where $n+1$ is the size of the array in the hash table). If we have inserted N $\langle \text{key}, \text{value} \rangle$ -pairs in the table, then the average number of elements in a list is $N/(n+1)$ where $n+1$ is the size of the array. If the average number of elements per list (or in other words per index) falls below this (i.e. typically below 2) we could resize the array by shrinking it or if the average list length goes above some threshold such as 8 we should increase the size of the array (cmp. the resizing array linked lists). What we should note is that if we resize the array we need to recalculate the hash values for all elements ($\langle \text{key}, \text{value} \rangle$ -pairs) in the hash table and re-insert them into the new, resized hash table. This is because the hash value calculation (the mapping of hashcode to hash) depends on the size of the array.

A schematic representation of a hash table with separate chaining:

at look-up index array with $\text{hash}(\text{key}, n)$



Hashing with separate chaining provide fast $O(1)$ look-up access and insertion times provided that hashcodes and hash values are uniformly distributed. However, one cannot implement ordered operations such as range queries by a hash table.

As an alternative to having linked lists for the indices of the array one could build the lists of the $\langle \text{key}, \text{value} \rangle$ -pairs in the array itself. This technique is known as *hashtables with linear probing*.

15 Basic graph algorithms

While we have not explicitly called them graphs we have already discussed data structures that are special cases of graphs. Both linked lists and trees are special cases of graphs.

A graph is a structure built from nodes, which are called **vertices**, and links connecting the vertices. The links are called **edges**.

A vertex v_a is said to be **adjacent** to another vertex v_b if v_a is directly connected to v_b by an edge

A **path** in a graph is a set of vertices connected by edges leading from a starting vertex to a finishing vertex with no repeated edges. A single linked linear list is an example of a path from the head element (vertex) to the end in the tail element (vertex).

A **simple path** is a path with no repeated vertices. In many (most) applications one is interested in simple paths and one commonly omit the “simple” and refer to them as paths. Simple paths are natural for instance in routing algorithms for navigation apps. If there are several parallel edges, in this example roads from one city to another, a routing algorithm should only select one of them. That is we expect a routing algorithm to return a route which only visit each vertex (city, crossroad etc.) once and return a unique route.

A **cycle** is a path with at least one edge and where the start and end vertices are the same.

The edges in a graph can be **undirected** (i.e. they can be traversed in any direction) or **directed** (an edge is one-way). Graphs with directed edges are called **digraphs**. Edges can have an associated **weight** or not. The weights can represent lengths, times, capacities, or costs etc. associated with traversing the edge. There are four possible combinations of these properties of the edges suitable for modelling different problems:

Combinations of edge types	Undirected edges	Directed edges
Edges without weights	Undirected weightless	Directed weightless
Weighted edges	Undirected weighted	Directed weighted

A **complete graph** is a graph where each vertex is directly connected to all other vertices by an edge. The density of a graph is defined as the pairs of vertices connected by edges to the total number of vertices. A **sparse** graph has relatively few of the possible edges present while a **dense** graph has most of these edges present.

The **length** of a path is defined as: i) in case of edges without weights – the number of edges in the path; or ii) in case of edges with weights – the sum of the edge weights in the path.

The number of edges leading directly to a vertex is called the **in-degree** of the vertex. Similarly, the number of edges leading out from a vertex is called the **out-degree** of the vertex.

15.1 Data structures for representing graphs

To implement effective graph algorithms we must design data structures that have low memory usage and allows us to effectively find what information used by the algorithms. This is a good example of why it is important to carefully design the data structures used to solve a problem.

To understand how to design suitable data structures to implement graphs we need to understand the characteristics of the problems we will model as graph problems.

Graphs are used to model many different problems including: maps, electric and electronic circuits, scheduling problems, computer networks, software, social networks, financial markets etc.

An important observation is that the vast majority of graphs resulting from modelling these problems are sparse graphs.

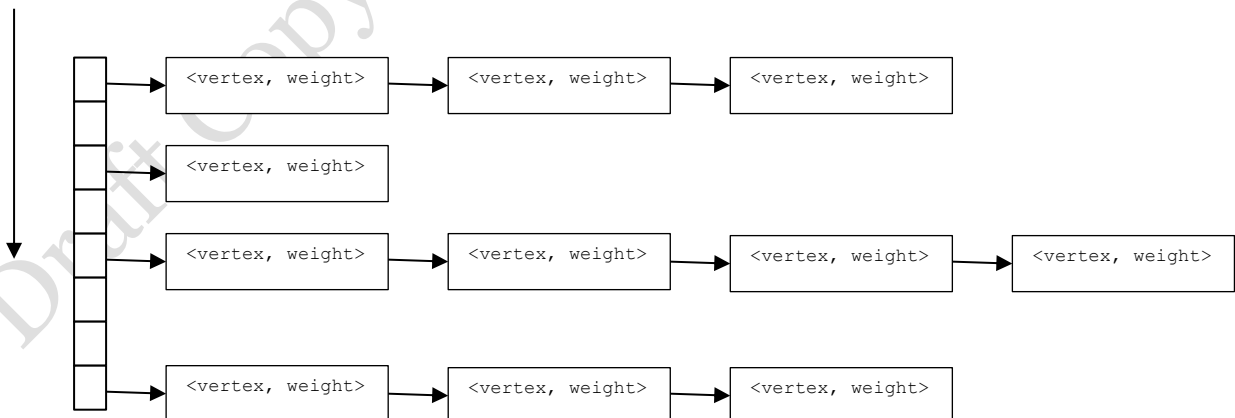
For graph problems we typically start from the vertices, i.e. to go from one city to another, to find a route on the Internet from one site to another etc. This indicates that we should be able to do fast look-ups based on the vertex we are looking for. As a simplifying assumption we will assume that we know the number of vertices V and that we will number all vertices from 0 to $V-1$.

We also note that most problems are sparse. That is the number of edges will be relatively low to the possible number of edges. The possible number of edges (not accounting for parallel edges) in a graph grows like $O(V^2)$ where V is the number of vertices. In a sparse graph the number of edges is $O(V)$, i.e. the number of edges E is within a constant of the number of vertices V .

We can associate each edge with where it starts. That is per vertex we could keep a list (or bag) of all edges originating from that vertex. Each edge could be represented as a pair of $\langle \text{end vertex}, \text{weight} \rangle$. We call a list of edges represented in this way an *adjacency list*, since it lists adjacent vertices of the vertex. For unweighted graphs we can skip the weight and for more complex graphs, where we associate the edge with more info, we have more complex weights. In a map, an edge would probably represent a section of a road and the weight could be $\langle \text{length}, \text{speed limit} \rangle$.

What we have is a situation similar to that of doing fast searches in large sets of sparse data. We can see the vertex numbers as the hash values and the (on average) small number of edges associated a vertex as “collision data”. That is, we can borrow ideas from how the data structure for a hash table with separate chaining is implemented. An effective representation for a sparse graph is an array of adjacency lists. We have a one dimensional array which we index with the vertex number. For each index (vertex) we have a list (or bag) of edges represented as indicated above.

array is indexed by vertex number



In this data structure we can easily see which vertices that we can reach by direct links from a specific vertex. That is, the edges represent directed edges. This structure can also be used to represent undirected edges. For an undirected edge between vertices v and w , i.e. an edge that is a two-way edge, we simply introduce two directed (one-way) edges, one from v to w and another in the opposite direction from w to v .

15.2 Common graph problems

Typical problems that we find in applications based on graphs are:

- Finding a path from vertex v to vertex w
- Finding the shortest path from vertex v to vertex w
- Finding cycles in a graph
- Finding a minimum spanning tree (a minimum spanning tree is a tree with its root in a vertex v by which all other, reachable vertices in the graph can be reached for which there is no other spanning tree with lower sum of the weights of the edges)
- Shortest paths tree (a tree with root in a vertex v containing all the shortest paths to all other vertices reachable)

Observe that a minimum spanning tree is not the same thing as a shortest paths tree. A minimum spanning tree identifies how to reach all other vertices at the lowest total cost (lowest sum of the edge weights). A shortest paths tree describes how to reach each individual vertex to the lowest cost. A consequence is that the sum of the edge weights in a minimum spanning tree is less or equal to that of the sum of the edge weights in a shortest paths tree.

15.3 Basic path finding algorithms in digraphs without edge weights

If we assume that there exists a path from vertex v to vertex w how can we then find such a path (any path will do) and how can we find the shortest such path? Here we outline the ideas of algorithms to solve these problems. For details refer to the book.

15.3.1 Depth first search

Depth first search (DFS) is easiest described as a recursive algorithm. The algorithm works as follows: We create a boolean array of size V which we call `marked[]` and another array of size V which we call `from[]` (the book uses the name `edgeTo[]`) where we store from which vertex we reached a vertex. We start our search from the source vertex v . For each vertex which we have not visited before we update the `marked[]` array to capture that we have visited the vertex. Before doing the recursive call to visit an unmarked vertex we set the `from[]` for the vertex we are about to visit to capture from what vertex we came. We stop iterating when we have reached the target we are aiming to reach or in case there is no such path when we have searched all edges we can reach.

```
boolean marked[V]
int from[V]

void dfs(graph G, int v, constant int w) //find path from v to w
{
    marked[v] = true
    for(int n = firstEdgeDest(v); //n is the index of the first vertex reached from v
        n != -1 && !marked[w] && !marked[n]; // assume -1 shows there were no more
        edges
    )
    {
        n = nextEdgeDest(v) //n is set to the index of the next vertex reached from v
        marked[n] = true
        from[n] = v
        dfs(graph G, n, w)
    }
}
```

When the method `dfs()` terminates we can track the path to w backwards in the array `from[]` starting at `from[w]` if `marked[w]` is set to true.

15.3.2 Breadth first search

DFS finds one path from a source vertex to a target vertex if such a path exists. DFS does not necessarily find the shortest path.

To find the shortest path to the target we should not go deep first in our search. Instead we should increase the search area by first see if we can reach the target by following a single edge (at distance 1). If the target vertex is not reachable at distance one we continue our search at distance two etc. To keep track of from which vertices to continue the search we store these in a FIFO-queue. Thus, when we visit a vertex we store all edges reachable from this vertex which have not been visited in the queue. To select the next vertex to check we simply remove the next vertex from the queue. If the queue becomes empty before we have reached the target vertex then there is no path to the vertex from our starting vertex. When we check a vertex we update the from() array with information on how we reached it as we know that we have reached it through the shortest possible path from the source.

In pseudo code the algorithm:

```
int from[V]
boolean marked[v]
fifoQueue q

void bfs(graph g, constant int v, constant int w)
    marked[v] = true
    enqueue(q, v)
    while(notEmpty(q) && v != w)
        v = dequeue(q)
        for(int n = firstEdgeDest(v); //n is the index of the first vertex reached from v
            n != -1; // assume -1 shows there were no more edges
            n = nextEdgeDest(v)) //n is set to the index of the next vertex reached from v
            if(!marked[n])
                from[n] = v
                marked[n] = true
                enqueue(q, n)
```

During the execution of the algorithm the FIFO-queue will contain vertices at distances k and $k+1$ from the starting vertex.

vertices with distance k	vertices with distance $k+1$
----------------------------	------------------------------

16 Complexity revisited

Now that we have looked at some algorithms it is time to revisit the concept of complexity. Understanding how to calculate the time and memory complexities is essential when selecting the appropriate algorithm/data structure to use in an application or when one needs to tailor an existing algorithm/data structure or design a new.

Understanding how to calculate the memory complexity of a data structure/algorithm is relatively easy. It is simply to check the maximum amount of memory during allocated execution or (in Java) how many and how large objects are created and exist simultaneously.

Calculating the execution time complexity is harder. It requires a thorough understanding of how the algorithm and code works.

To determine the best and worst cases you need to understand what properties of the input will generate these behaviours. This requires some creativity in addition to the understanding of the algorithm/data structures.

When determining the time complexity one needs to identify where the most time is spent in the code during execution.

What we are looking for depends on the algorithm. Examples we have encountered are:

- For a sorting algorithm we are interested in how long time and how much auxiliary memory is used to sort an input of size N . In a sorting algorithm based on comparisons, the performance is typically bounded by the number of comparisons made and possibly also the number of elements moved (swapped).
- To insert an element in a linked list depends on how many comparisons we have to make and how many links (references/pointers) we have to follow to find the place to insert the element.
- For tree-like search structures we are interested in understanding how long time it would take to insert or search for a key in a tree with N elements. The performance normally depends on the depth of tree, which for a linked implementation determines the number of links we have to follow to insert or find an element.

For most algorithms there are small parts of the code which are heavily used during execution. The execution time spent in these parts will dominate the total execution time. For large problem sizes one can ignore the contributions to the execution time from less used parts of the code. To understand this we can look at the most common complexity classes we will encounter:

constant $O(1)$, logarithmic $O(\log(N))$, linear $O(N)$, linear logarithmic $O(N\log(N))$ and quadratic $O(N^2)$.

When the problem size N grows to be large we have the following relations for the complexity: $O(1) \ll O(\log(N)) \ll O(N) \ll O(N\log(N)) \ll O(N^2)$. As a consequence of this the contribution of less fast growing terms will have little significance to the performance for large problem sizes and we can normally ignore them when calculating the complexity.

This means that for large problem sizes we only need to consider the fastest growing term in our model of the time or memory complexity for an algorithm/data structure.

We should note that we can solve problems of size N in a reasonable amount of time when using algorithms with time complexity which is $\leq O(N\log(N))$ also for very large problem sizes. For algorithms with time complexities of $O(N^2)$ and larger we can only expect to solve smaller problems within a reasonable time frame.

Furthermore one should recognize that only considering the worst case performance of an algorithm is not enough to understand if the algorithm is a good choice to use to solve a specific problem. One has to weigh in issues such as if: i) one need guaranteed worst case performance bound for all inputs; or ii) if it is acceptable to have good average performance; or iii) if we know characteristics of the input which allows us to select an algorithm that is particularly good for that kind of input.

Examples of such decisions are:

- Quicksort has a probabilistic performance bound of $O(N\log(N))$ which means that for the vast majority of inputs it will likely be the fastest sorting algorithm for larger problems – but for inputs with specific characteristics it may need $O(N^2)$ time; ii)
- If the input is nearly sorted (i.e. has few inversions) insertionsort will be faster than both merge- and quicksort even though insertionsort has a worst case performance of $O(N^2)$ while merge- and quicksort has a worst case of $O(N\log(N))$.

What sometimes creates problems to understand is how to identify logarithmic performance. The simplistic answer is that this occurs when we halve the problem size in each iteration (or rather when we reduce the problem size by a factor n in each step where $n \geq 2$).

Examples:

- The number of iterations performed by
`for(int i = 0; i < N; i = i * a) ...` where a is a constant and $a \geq 2$ will be $\sim \log_a(N)$
- When performing a binary search the remaining number of elements to search in will be halved in each iteration. The number of times that we can halve the problem size if it initially was of size N is $\sim \log_2(N)$. Hence the performance of binary search is bounded by $O(\log_2(N))$

17 Background: Structure your development - The best pieces of advice one can give!

When writing programs solving non-trivial problems or designing more elaborate data structures, such as linked data structures, it is a truly bad idea to immediately start coding and experimenting with the code to see what happens. In any other area of engineering it would be totally unconceivable not to bring structure into the process by starting by understanding the problem and then create a design or a blueprint for how to solve the problem. That the lack of structure is a problem and is often and nearly only accepted in the IT area is a problem recognized by successful IT- entrepreneurs and prominent researchers alike. To get a feeling for what expectations you will meet as an engineer read:

- <https://cacm.acm.org/blogs/blog-cacm/242740-talented-programmers-dont-tolerate-chaos/fulltext>
- <https://cacm.acm.org/magazines/2015/4/184705-who-builds-a-house-without-drawing-blueprints/abstract>

As engineers we should also adhere to the best-practice of modelling problems and solutions and use a more structured approach for our problem solving.

17.1 Polya's problem solving method

George Polya was a Hungarian mathematician who was a professor at Princeton and Stanford universities. He published a book on problem solving methods, *How to Solve It*, in 1945 in which he describes a heuristic method for problem solving in four steps. The book has sold in millions of copies and has inspired several generations of engineers and scientists, including Nobel price laureates. It is a seminal work and still inspires scientists and engineers. A testimony of its importance is that it still is in print (i.e. you can still buy newly printed copies of it). In the following we will try to summarize and translate the 253 pages of the book into a simple, yet effective structured approach to problem solving for programmers.

Polya's method consists of four steps:

1. Understand the problem
2. Make a plan
3. Carry out the plan
4. Look back - reflect

17.1.1 Understand the problem

The first step, *Understand the problem*, is probably the most important step. Without proper understanding of the problem it is not possible to solve or create good quality solutions to the problem. In this phase you need to ask a lot of questions such as: *Do I understand what it is that I am supposed to solve? Do I understand the problem formulation? Can I rephrase the problem in my own words? What is it that I am supposed to deliver? What is it that the user wants? How would the user want to use the solution? What knowledge is required to solve the problem? Do I have the necessary knowledge?*

17.1.2 Make a plan

In the second phase, *Make a plan*, you should devise a plan for how to solve the problem and a plan for *how to test the correctness of the solution*. Making a plan for how to solve the problem (in our case how to implement it) and a test plan goes hand-in-hand. The reason for this duality is simply:

If you do not understand how to test your solution you do not understand the problem, while if you understand how to test the solution you probably understand the problem.

When making the plan you often need to break down the problem in smaller sub-problems, possibly in several steps/levels, which you can solve and build the complete solution from. There are two general design methods to model a problem which you can apply to most problems: Top-Down and Bottom-Up design which are described later.

When designing your plan and more specifically when you describe how you have modelled your solution, you should not describe it in a specific programming language. Instead you should use paper and pen to describe it in pseudo code and graphic descriptions. To describe methods/functions one can use Jackson Structured Programming (JSP) and to describe how classes/objects interact one can use the Unified Modelling Language (UML).

For this course it is often sufficient to draw simple pictures like the descriptions of linked lists and trees that you find in this document and use JSP to illustrate algorithms. UML is a very large framework which goes beyond what you actually need in this course.

17.1.3 Carry out the plan

In the third phase, *Carry out the plan*, you implement and *test* the solution described in the plan. A good advice is to test continuously as you implement parts of the solution.

17.1.4 Look back

The fourth and final phase, *Look back*, means that you should reflect on what you have done and learn from the process.

17.1.5 Where problems come from

While all of these steps are natural and reasonable steps in the process of solving a problem, without enforcing you to spend too much time while yet reaching good quality solutions, three of the steps are often neglected by students and “professionals” alike.

The first, second and fourth phases of Polya’s method are often neglected and has often been neglected by past and present students and “professionals” of all times. The reason is that without proper experiences it may be hard to see why it is worthwhile investing time in these phases. The problem is that one is often eager to get on with the “real” problem solving as fast as possible. For IT-students and “engineers” this often means that one wants to get on with the programming as fast as possible.

However, starting to write code without properly understanding the problem, how it could/should be solved and without a proper plan generally leads to poor solutions, waste of time and in the worst case scenario pure misery.

As a teacher you see countless examples of students rushing to create solutions and failing to do so. I have seen students spend in excess of 200-300 hours, writing several hundred lines of code and adding code to save completely unstructured programs they have not spent enough effort to understand. When they eventually have realized that their approach was doomed to fail and having understood the problem properly, they could solve the problem in less than an hour with less than ten lines of code. Such experience is possibly educational but it is unnecessarily hard earned lessons.

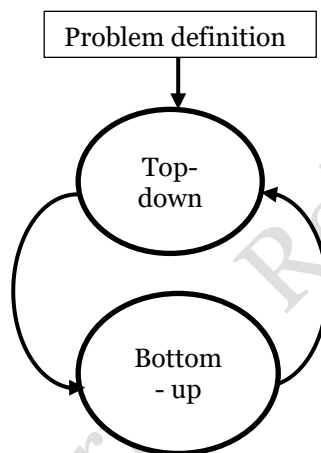
17.2 Top-down and Bottom-up design

Two basic design principles for designing solutions to programming problems are Top-down and Bottom-up design.

Top-down means that we start from the problem definition and try to recursively break it into smaller pieces to find a solution. The process is stopped when we reach small enough sub-problems that can easily be solved/implemented. This could be likened to an architect who starts the design of a building by making an overall drawing/blue print of the whole building and then breaking this down into drawings of floor plans, rooms down to what building material is used.

Bottom-up means that we start from the most basic blocks, design them and then iteratively see how these can be combined to create larger parts of the solution. For an architect this could be likened to see what or how we could build something from a set of building materials such as bricks, windows, doors etc.

In programming we often combine the methods iteratively, starting by a Top-down break-down of the problem followed by a Bottom-up design where we refine the basic building-blocks. The goal is to find as simple, clean, extendable and efficient solutions.



17.3 Some guiding principles when designing methods, data structures and classes

The most basic advice is that you should avoid creating monolithic solutions where you try to solve many or all problems in a single method or class. Monolithic structures are inflexible, hard to get an overview of and hard to adapt to changing requirements.

Instead you should focus on making your basic building blocks as simple and clean as possible. Methods and classes that do one thing only and does it well are to prefer. They are easier to understand and could be reused easier.

An example of how to think when designing such methods is found in the assignments of the double linked circular list where you are to implement a method to insert a new element into the list after a specific element, `insertAfter()`. The beauty of such a simple method is that you can easily implement specialized methods to insert elements first or last in the list. While if you had chosen to implement specialized methods for inserting elements first and last you would have had more code, with more errors. Moreover, designing methods to insert elements ordered based on priority (sorted by the integer in the elements) would have become more complicated.

17.4 Proposed basic method for structured problem solving for programmers

So to summarize, the method we propose you should use, or at least try, to solve the non-trivial problems you encounter in the course is:

1. Make sure you understand the problem properly.
2. Try to break-down the problem into manageable pieces, i.e. pieces which you can solve and use to build the complete solution from
3. ***Describe your design in some kind of pseudo-code, draw pictures of your data structures, how algorithms work and what the methods/functions operating on them should result in.*** Use simple descriptive methods such as pseudo code and Jackson Structured Programming diagrams. Make sure you build your design on general building blocks rather than specialized.
4. Design tests to test your solution/your design¹⁷.
5. When you understand the problem and how you could solve it – only then should you start to finalize your break-down of the problem into well-defined classes and methods in JAVA and/or functions and datatypes (such as `structs`) in C.

Each method/function should preferably solve a single, well-defined problem efficiently. This will make it easier to check that the methods/functions are correct. Once you know that the building-blocks are correct, you can use them to build more complex functionality from.

6. Once you have a solution on paper which is simple and clean – only then and not earlier should you try to implement it in code.
7. A good advice, both when designing and coding, is to always strive for simplicity. If your design/code becomes too complex you should redesign your solution to simplify it. While most of us are hesitant to discard the work we have put into something, the fact is that trying to add even more complexity (code) to solve the problem in most cases simply adds more errors and makes the solution even harder to understand¹⁸.

Spending a lot of time on understanding the problem and designing solutions on paper may seem like a waste of time! And why delay the fun of coding and executing the code to see how (if) it works? The simple answer to this is:

Code is important but without you understanding the problem in detail and basing your code on a good design – you cannot expect your solutions/code to be of good quality.

You will save lots of time and create better, more professional solutions by spending more (often much more!) time on understanding the problems and properly designing your solutions as compared to simply spending time mainly on coding.

Creating good and correct solutions is what being an engineer is about!

¹⁷ If you cannot design tests for your solution to the problem you probably do not understand your solution well enough. I.e. then you need to work more with your solution.

¹⁸ To simply keep adding code to something that does not work or which one does not understand is a common mistake made by many novices to programming. Such an approach actually only makes the task of learning how to program unnecessarily difficult.

18 Background: How do you know that your program is correct?

When we develop programs we do it with the intention that it should be used by users (at least if we develop professionally). The users expect our programs to be correct and efficient. Thus questions we have to ask ourselves include: How do you know that your program is correct? That it meets the specification? That the user/customer will be satisfied?

We can answer these questions by two techniques: **verification** and **validation**. These are two related but distinct concepts that too often are confused and mixed-up (especially on the Internet).

18.1 Verification

Verification means that we prove the correctness of a program or algorithm. This will be discussed further in the course (i.e. you will not use it yet).

To prove correctness relies on being able to construct a mathematically sound proof. Constructing a mathematical proof for a complete application/program is hard due to many reasons. These reasons involve that many popular programming languages does not have a well-defined semantics, the size of the program, the fact that programs are executed on operating systems and hardware which also may not be well-defined etc. However we can and should prove the correctness of algorithms.

In this course you will construct proofs for algorithms based on what essentially is a proof by induction. In a proof by induction you will have a statement $S(n)$ that is true from the beginning (a base-case, typically for $n=0$ or $n=1$). In the induction step we assume that $S(n)$ is true and then we prove that it holds for $S(n+1)$.

For algorithms that are based on loops (typically a `for`- or `while`-loop) we can apply a similar technique to prove the correctness. In this case we define what is referred to *loop-invariant* which should hold from the beginning and after each iteration in the loop. If properly constructed the loop-invariant should tell us that each iteration of the loop will lead us closer to the expected end result. That is, after enough iterations we will reach the expected, correct result. For example a properly defined loop-invariant for a sorting algorithm could be used to show that each iteration will make more of the input correctly sorted.

18.2 Validation and testing

Validation means that we make it plausible that an algorithm or program is correct. That is, instead of proving that something is correct we try to make a convincing case that it is correct. Validation is thus a weaker argument for correctness than verification.

18.2.1 Testing

There are several techniques for validation where *testing* is the most commonly used technique to validate the correctness of programs. Testing means that we test the response of a system/module/piece of code by feeding combinations of parameter values to it and checking that its behaviour/output is correct (plausible).

Typically testing is done on several levels when a system is developed. Testing should be an integral part of the development process.

Testing should start by the testing of methods/functions, classes and sets of methods/functions/classes that makes up well-defined building-blocks, i.e. units/modules, of the system. This is referred to as **unit testing** or **module testing**.

The next step of testing is referred to as **integration testing**. This means that we integrate two or several units/modules to see that they work together and interact properly.

The, hopefully, final step is to test that the whole system works/behaves as intended and is referred to as **system test**.

However, during the development and testing of a system we may encounter issues in units/modules that has to be addressed and corrected. In such cases we have to re-test all parts of the system already tested that could be affected by the corrections performed. This is referred to as **regression testing**.

18.2.2 Black-box and white-box testing

There are two approaches to unit testing referred to as black-box testing and white-box also referred to as glass-box testing.

In black-box testing we do not need to know, or should bother about, anything of the internals of the unit/module we are testing. We only supply different values to the parameters of the unit and observe the output from the unit.

In white-box testing we are also observing the internals of the unit we are testing. More precisely we want to make sure that all parts of the code is traversed during the testing. One thing we can do with white-box testing compared to black-box testing is to identify so called “dead” code which is code which never can be reached and thus could (and should) be removed.

18.2.3 Goal of testing - exhaustive testing – the limits of testing

The ultimate goal of testing is that we, to the best of our abilities, should assert that the system or module/unit tested is correct, i.e. behaves correctly for any input.

In the best of worlds we should do what is called exhaustive testing. Exhaustive testing means that we have tested the system under any possible condition, which for code means we have to test it with any possible combination of input values for all parameters of the system.

Exhaustive testing is not possible to achieve other than for small systems with a limited number of input parameters.

18.2.4 Basic strategies for testing

What we have to resort whenever we cannot do exhaustive testing is to test our systems/units with smart selections of parameter values and combinations. A good base to start from is to test the system with a combination of:

- “Normal” values for the input parameters. What normal values are depends on the application.
- Extreme values such as the largest/smallest values for input parameters. Examples could be maximum/minimum integer values, inserting or removing elements from a queue with zero elements, sorting ascending/descending sequences of elements, sorting sequences of equal elements etc.

When performing the tests it is advisable to start by varying one input parameter at a time to facilitate the detection of any flaws in the design. The next step is to start varying two or more parameters at a time to find flaws which depend on “co-variance” of the input.

18.2.5 The role of testing in system development

Testing is an integral part of system development. The test plan should be developed simultaneously to the plan for how to solve the problem. As stated above: *If you do not understand how to test your solution you do not understand the problem and vice versa if you understand how to test the solution you understand the problem.*

18.2.6 On the design of unit/module tests for the course

In this course you will face two kinds of problems which should be tested in two different ways:

- Algorithms that take an input and process it to produce an output – such as a sorting algorithm which takes unsorted data as input and produces a sorted sequence of data as output
- Algorithms/applications which allows different operations to be performed – such as inserting, removing and printing contents of a queue

In both of these cases your tests should allow the user to provide the input. Typically you would read the input from `stdin` and write results to `stdout`. This will allow you to redirect the input and the output if you run the program from a command line.

For a sorting algorithm you could have different input files with different sequences of input (you will typically use input sizes ranging from 10^1 - 10^n where n may be 6,7,8,9,10 – that is quite large inputs which you want to be able to produce by a program and store in a file). The result from the test could be a printout of the sorted sequence and/or a means (program) to check that the output is properly sorted.

For an application with several different operations you should design the test so that it repeatedly asks the user what operation he/she wants to perform and then perform the operation until the user indicates that the program/test should be terminated. The easiest way is to encode the different operations so the user provides an integer $[0, n]$ where zero indicates that the test should be terminated and a non-zero value indicates which operation to be performed. In pseudo code such a test could be described as:

```
prompt user for a new command
read command
if the command is not a valid command take care of error
while(command not equal to zero)
begin
    switch19(command)
        case x: do command
            break
        ....other commands/cases
        default
            read next command
end
```

¹⁹ `switch` is a statement found in both C and JAVA which is suitable when you want to select cases matching on an integer value. The same problem could be solved by a number of nested `if`-statements. However, the `switch`-statement results in cleaner and more readable code when there are more than a few cases to select from

19 Finding flaws in your design - debugging

While the best thing is to carefully design your solutions and fully understand what you are doing when programming the reality is that most programs will have flaws/errors. To find and remove these errors/flaws are called debugging. Errors in programs come in several flavours: It may be logical errors or it may be performance related errors.

19.1 Performance related errors

A performance related error is when a program uses more resources than expected. It may be memory or execution time related errors. For the algorithms and data structures in this course you will learn how much memory and time we can expect them to use as a function of the problem size, so called memory and time complexity.

To detect that there is a performance related problem one monitor memory consumption and/or execution time. As an example: If we expect an algorithm to use an execution time proportional to the size of the problem, N , and if we detect that it uses more than twice the execution time to solve a problem of double the size we could assume that the algorithm is not properly implemented.

For memory related problems one example is memory leaks. A memory leak occurs when memory allocated (reserved) for data structures that are no longer used by the program is not properly handled so that the memory could be reused. For programs that execute for a long time a memory leak could eventually result in that all available memory is used and the program halts or crashes. Everyone who has experienced that one has to reboot a computer which has become unresponsive after being up and running for a long time knows what memory leaks can lead to.

19.2 Logical errors

When you detect logical errors during testing there are mainly three approaches to find and remove the errors:

- Analytic debugging: Go back to the design and the corresponding implementation (the code) of the program to understand what it does and where it may fail
- Trace printouts: Add printouts to the code to trace the execution
- Use of a debugger. A debugger is a program that allows ne to follow the execution of a program and inspect values of variables during the execution

The use of debuggers are considered common knowledge for programmers but unfortunately not taught in any courses.

19.2.1 The analytic approach

An *analytic* approach is almost all cases the preferable method. If you can find and correct flaws by understanding the design/code chances are that you will address the problem properly and not simply patch the code/problem without really understanding what you do. This is view shared by many (most) prominent programmers as for instance Linus Torvalds the creator of LINUX.

19.2.2 Trace printouts

Trace printouts is probably the first method used by new programmers and it is sometimes useful. The problem is that one has to add the printouts to the source code. Often the code becomes sprinkled with such statements. And eventually, when one believes that one has found and addressed all problems one has to remove the printouts, only to discover that there are more problems which requires adding printouts again. There are tools which allows you to do this in a more orderly manner. If you want to use printouts and other checks in your code during the development phase but not leave them in

production code (version of the code which is the basis for the version of the application used by the user) it may be worthwhile learning how to use the C-pre-processor (**cpp**). `cpp` can be used to process any text file regardless if it contains C, JAVA or any other type of text (code). By adding pre-processor commands²⁰ to the text one can set up the text so that `cpp` can remove or retain text such as printouts depending on what parameters one provides to `cpp` when it processes a text. We will not cover this in detail in the text but will leave the details to the interested to ask the teachers assistants on how to do this.

19.2.3 Debuggers

A debugger is a useful tool that allows you to trace the execution of a program. IDEs like IntelliJ have built-in debuggers while you have stand-alone debuggers such as `gdb` (the GNU²¹ debugger) that can be used from command line interfaces in UNIX/Linux.

Most debuggers allow for some basic operations:

- Run a program in the debugger. Often the program has to be compiled in a specific way which inserts instructions in the code to allow the debugger to stop/resume the execution and correlate the execution (of byte or assembly code) to the source code lines (such as JAVA or C source code)
- Insert break points at specific lines of the source code or on entry when a method/function is called
- Show surrounding lines of code where the execution has been stopped
- Resume execution of the program
- Resume execution line by line (often called “step”). Often the program can be stepped by following execution in to methods/functions (step-in) called or not (step). Note: it is normally not possible to follow execution into library methods/function as these are pre-compiled without the necessary additional code which allows the debugger to execute them step-by-step (line-by-line)
- Show the value of variables

A debugger can be a useful tool, not the least for a novice to understand how a program executes. If no break points have been inserted the execution of a program in a debugger will run the program until it terminates. Termination can be normal termination (the program is run until it ends) or the execution may be caused by an error. In the latter case the debugger allows the user to inspect where the error occurred and possibly also to inspect values of variables/parameters.

In LINUX/UNIX environments one often uses tools from the GNU-project where the debugger is named `gdb`. For Mac, Windows and LINUX users there are many Integrated Development Environments with built in debuggers. One such IDE for JAVA preferred by many students is IntelliJ which is free to download for students.

As there is no single development environment used by all students it is not possible to go into any details on how to use a debugger. Rather we have to direct you to online material describing the specific environment of your preference.

²⁰ A `cpp` command is started by `#include<stdio.h>` is an example of a command that tells `cpp` to insert a copy of the text found in `stdio.h` into the output file that `cpp` produces

²¹ The GNU project founded by Richard Stallman at MIT provides much, high quality, open source software such as compilers, editors etc.

A typical debugging session follows a simple pattern for starters: (In the following typical commands to the debugger are printed in **courier bold**) Typically you would start the debugging by running the program in the debugger until it crashes. When the program crashes you can see **where** it was in the execution when it crashed. It also allows you to set **break** points when entering a method or at specified lines in the source code. When the debugger executes the program and the execution reaches a break point the execution halts. Then the user may inspect variable values by **printing** them, **step** the execution line by line or simply **continue** the execution.

19.2.4 Comparing different debugging techniques

While debuggers and trace printouts may be of use for finding simpler errors there are many examples of where they are inadequate.

One such example is if when there are big data structures. Assume a single linked list where we insert N elements and when one of the elements is inserted a link is broken. This is a type of error which will not manifest itself before the list is traversed and the broken link is reached. This may occur at a much later stage in the execution compared to when the cause, broken link, was introduced. Such an error may be very hard to find. It is easy to see that the link is broken, but to detect when and why the link was broken may be very hard regardless if a one uses a debugger or trace printouts.

The best way to find and to avoid introducing bugs are to carefully design and implement the code. And to make sure one understands the design in detail. Should an error such as the one described above the best way to resolve it is in most cases to go back to the design and analyse the design/code to find the error.

In fact more than one prominent programmer, such as Linus Torvalds, argue against using debuggers as their usage may invite to patching the code rather than properly identify and correct problems.

Best practice: *Make sure you fully understand the problem and make a carefully thought out design (possibly including pictures/drawings) of data structures and algorithms before you start to implement. And go back to the design to properly identify and correct errors rather than patching code.*

20 Background: Redirecting input/output

In this course you will learn about and work with algorithms and data structures designed to be able to work also with large input data sets. The input data is normally stored in files as is often the output. Thus for a program to work on such a data set it needs to be able to read and possibly write files. However, you have not learned how to read and write files in any of the courses which are pre-requisites for this course. So do you need to learn how to access files from your JAVA and C-programs?

The answer to this is no – you need not to learn how to do this!²² It is sufficient that you know how to read and write from/to the keyboard/terminal/window. In most operating systems you can redirect input/output to/from a process²³ to read/write from/to files or other processes. The following is a description of how I/O is done in UNIX/LINUX. However, most of these principles also apply to other operating systems such as the Microsoft operating systems.

What is interesting with this is that the principles introduced here for redirecting I/O works for all programming languages. This is because the operating system does not and should not know what programming language a program executing in a process has been written in. It just deliver input to the process from the terminal/window through `stdin` and transfers output from the process through `stdout` to the terminal/window. Again this is an example of abstractions necessary to get systems to work.

20.1 I/O to from the terminal

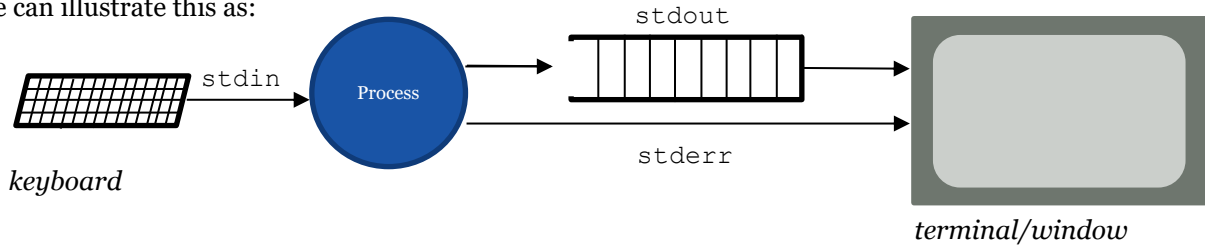
All I/O goes through the operating system. In order to have one simple interface for a process to communicate with its environment (terminal, keyboard, files etc.) all I/O is performed as if the process was reading/writing from/to files. However, these need not be files on a disk and are referred to as streams. Streams may lead to/from all types of I/O devices such as disks, terminals, keyboards, mouses etc.

When a process starts to execute it has three streams open, one for input and two for output. The input stream is referred to as standard input (`stdin`) and takes its input from the keyboard. The first output stream is referred to as standard output (`stdout`). `stdout` leads to the terminal/window in which the process was started. The standard output stream is buffered which means that when you output (write) something to this stream the data is actually written to a buffer in the operating system. This allows the operating system to output the content of the buffer when it is not occupied with other tasks. There is a second output stream open referred to as standard error (`stderr`). By default this also leads to the terminal/window in which the process was started. The difference between `stdout` and `stderr` is that `stderr` is not buffered which means that when something is output from the process on `stderr` the operating system has to output it to the terminal/window immediately without delay. Thus, if you have written an error message to `stdout` it may reside in the output buffer when a process crashes in which case the user never will see it. Had it instead been written to `stderr` before the process crashes, then it would also have been output to the terminal.

²² Although learning how to read and write files is quite simple. The basic system calls you need to learn are: `open()`, `close()`, `read()`, `write()`, `lseek()`. And the related methods in languages like JAVA and the higher level functions in the C/C++ libraries are very similar to these system calls.

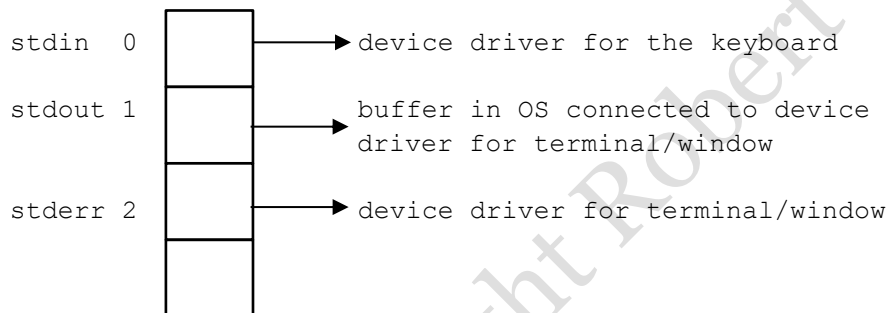
²³ A process is a program being executed

We can illustrate this as:



The open streams are implemented by that the operating system keeps a small table of what is called file descriptors for the open streams of the process. Each entry for an open stream identifies the device driver for the I/O-unit that the stream is attached to. The device driver implements functions corresponding to the system calls used to `read()`, `write()` data which the process calls when it wants to read/write data from/to the stream. For example, if a process calls `write()` to output something on a stream it results in that the operating system looks up the `write()` function in the device driver referred to by the entry in the file descriptor table corresponding to that stream and it will call that function with the parameters provided in the `write()` call made by the process. We can illustrate the file descriptor table for the example in the figure above as:

filedescriptor table for the process



So for example, what happens when we read something from the keyboard (`stdin`) is that we read from the device that is referred to from file descriptor 0 which normally would be the keyboard. And when we write to the terminal/window (`stdout`) we write to the device referred to from file descriptor 1.

20.2 Redirecting I/O

So what if we would like to have our program read from a file instead of the keyboard when it is executed, or write its output to another file rather than the terminal/window. Does this mean that we have to change the code for our program? The answer to this is no!

What we can do is to instruct the command interpreter²⁴ (normally referred to as a shell) to redirect the input/output to be read from/written to files. When redirecting input the command interpreter will set the `stdin` file descriptor to point to the device driver responsible for the file we want to read from and when redirecting output it will set the `stdout` file descriptor to point to the device driver responsible for the file we want to output to.

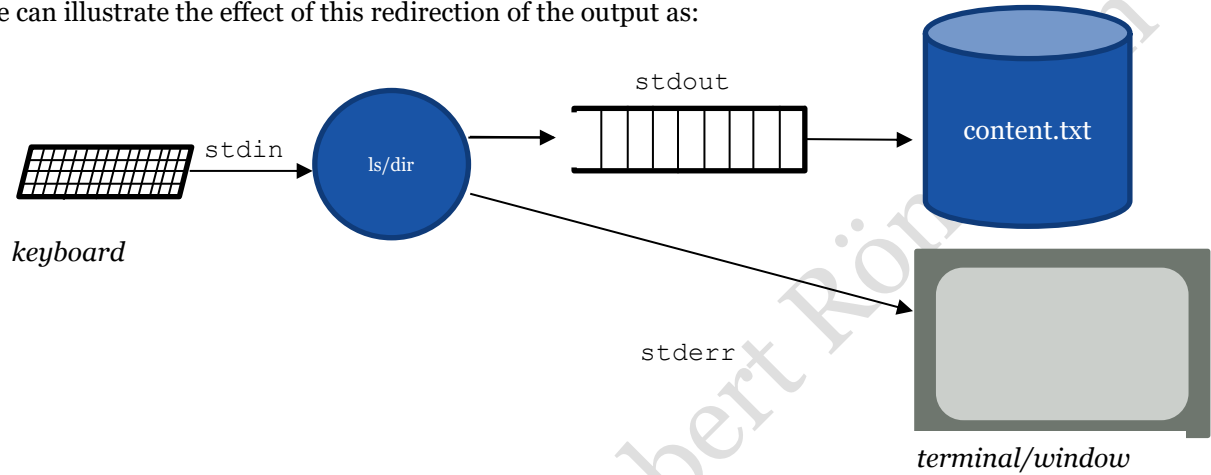
20.3 Redirecting output

To redirect the output from a process one uses `>`. The commands to redirect output from a process executing program to a file are:

²⁴ The command interpreter is the program running when you interact with the operating system using the command line. In other words, it is the program which outputs the prompt and then reads and interpret/performs the commands you write on the command line.

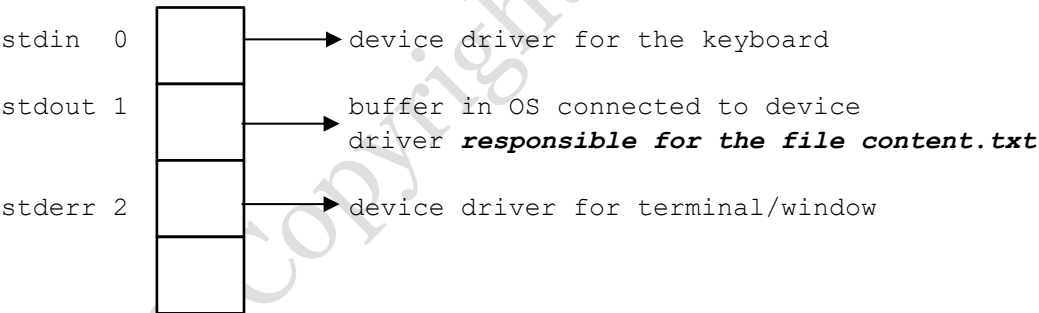
UNIX/LINUX	Windows
<pre>program > file</pre> <p>example: output the listing of the directory²⁵ to a file named <code>content.txt</code></p> <pre>ls > content.txt</pre>	<pre>program > file</pre> <p>example: output the listing of the directory²⁶ to a file named <code>content.txt</code></p> <pre>dir > content.txt</pre>

We can illustrate the effect of this redirection of the output as:



What actually happens in the above examples is that the device driver associated with `stdout` is reset in the process in which the program (`ls/dir`) is executed:

filedescriptor table for the process executing `ls/dir`



²⁵`ls` is the program executed to list the contents of a directory in UNIX/LINUX

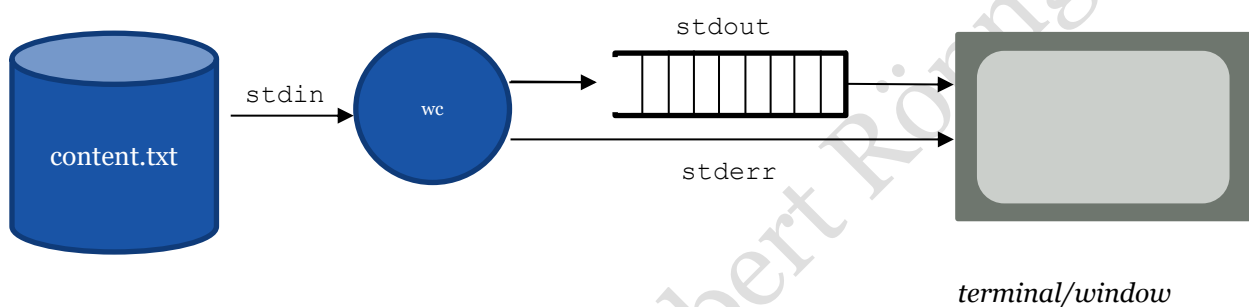
²⁶`dir` is the command/program executed to list the contents of a directory in Windows

20.4 Redirecting input

To redirect the input to a process one uses `<`. The commands to redirect input to the process executing `program` to be read from a file are:

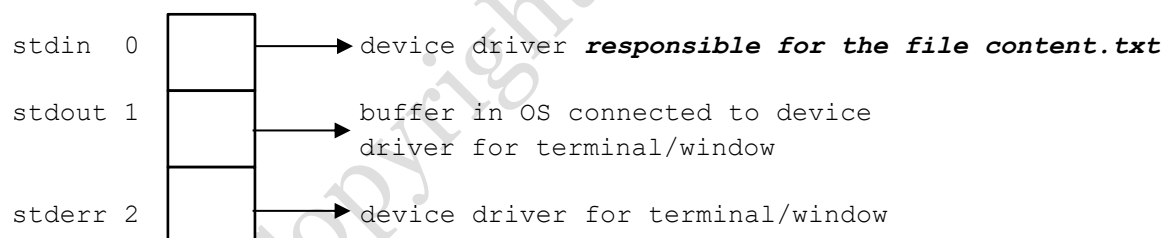
UNIX/LINUX	Windows
<pre>program < file</pre> <p><i>example: set the input for a process counting chars etc. to come from the file <code>content.txt</code></i></p> <pre>wc < content.txt</pre>	<pre>program < file</pre>

We can illustrate this as:



What actually happens in the above example is that the device driver associated with `stdin` is reset in the process in which the program (`wc`) is executed:

filedescriptor table for the process executing `wc`



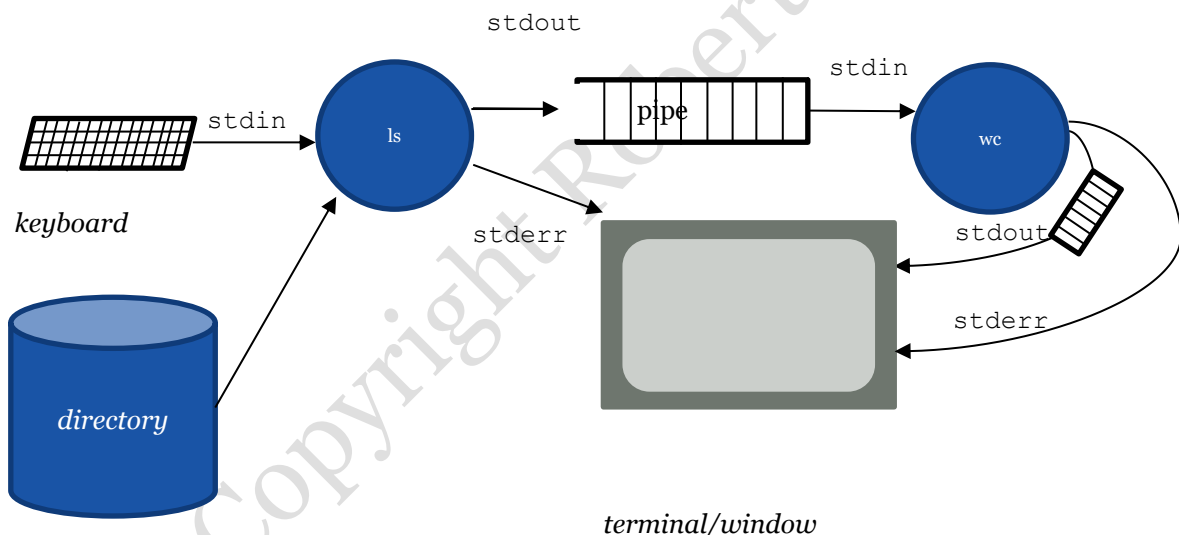
Of course it is possible to redirect both the input and output (`stdin` and `stdout`) for one process.

20.5 Piping: Redirecting the output from one process to be the input of another process

It is also possible to redirect the output from one process to be the input of another process. This is done by `|`. What actually happens in this case is that the output (`stdout`) from the first process is redirected to a small buffer in the operating system called a pipe, while the input (`stdin`) for the second process is redirected to read from the pipe

UNIX/LINUX
<code>process1 process2</code>
example: set the output from a process listing the content of a directory to be the input of another process counting characters, words and lines
<code>ls wc</code>

We can illustrate the effect of this as (a *directory* is a special type of file found in the file system which `ls` reads and displays in a user friendly format):



20.6 An example how to pipe output/input between processes

When logged on to a UNIX/LINUX computer your environment is set up by a number of *environment variables*²⁷. The environment variables determine things such as which directory is your home directory (`HOME`) which is the directory you will be attached to when logging on to the machine, which directories to search²⁸ when you issue a command (`PATH`) or what pager (`PAGER`) to display pages in.

²⁷ On a Windows machine the corresponding information is found in the “registry”

²⁸ Many (most) commands that you issue such as `ls`, `cd` etc. are programs that will be executed. `PATH` contains a list of directories that are searched to find the executable (such as the executables named `ls`, `cd`) when you issue the command.

If you want to inspect your environment variables you can execute:

```
printenv
```

This will result in a long listing of your environment variables. The listing is not ordered which makes it less useful. To order the listing in alphabetic order we can send the output from `printenv` to `sort`.

```
printenv | sort
```

The listing will now be sorted but may still be too long to fit in one page. To have one page at a time displayed we can send the output from `sort` to a pager such as `less` (to exit `less` press the 'q'-key).

```
printenv | sort | less
```

`printenv`, `sort` and `less` are examples of programs called *filters*. A filter reads input from `stdin` until an end-of-file marker (EOF) is found, manipulates what it has read and outputs the result to `stdout`. Well-designed filters which solve a well-defined problem (i.e. a single thing) efficiently, are powerful and can be combined to solve more complex problems by piping.

21 Background: Some useful commands/things in UNIX/LINUX

Working in a UNIX/LINUX environment has some advantages. Such as UNIX/LINUX systems having command interpreters capable of interpreting a programming language which makes it possible to issue complex commands and they have an excellent built in manual.

21.1 Changing directory, path to a directory and listing the contents of a directory

cd *path to directory* changes directory to the directory with *path to directory*.
path to directory may be an absolute path (a path from the root directory) or a relative path to a sub-directory of the current directory
.
(a dot) is a reference to the current directory
..
(two dots) is a reference to the parent directory
~
(tilde) is a reference to your home directory
Example: `cd /home/b/bob` changes directory to `/home/b/bob`

pwd displays the path to the current (working) directory (**p**ath to **w**orking **d**irectory)

ls lists the contents (files and directories) of the current directory. **ls** can take several parameters, to find out more type: `ls -help`

21.2 man pages

The built-in manual is often referred to as “*man pages*”. In the man pages you can find detailed information on user commands, system calls and library functions for different languages. For a library function you will get information on things such as: parameters and return values, which header files to include, possible problems associated with the function, related functions etc.

The man pages are divided into several sections. Section 1 contains built-in and user commands implemented by the shell, Section 2 contains the system calls and Sections 3 contains library functions for different programming languages.

To search for information in the man pages you give the command:

```
man -k item
```

Where *item* is the keyword you are searching for. There is a limitation to the man command in that you can only search for one keyword in a search.

To display the information on a specific command/function you simply give the command `man keyword` to the command interpreter, where *keyword* is the name of the command/function. `man` will display information on what it finds from the first section where it finds a match for the *keyword*. This means that if you are searching for something that exists under the same name as both a user command in Section 1 and a system call in Section 2 and possibly as a library function in Section 3, then `man` will only display the man page from Section 1.

If we for example search for the system call to send signals to a process which is named `kill` with the command `man kill`, then `man` will display information on the user command `kill` from Section 1 where it finds the first match. The Section is indicated by the section number in parenthesis after the name in the page header of the man page, i.e. `kill(1)` refers to Section 1 while `kill(2)` refers to Section 2.

To instruct `man` to display²⁹ information from a specific Section of the man pages you can specify the Section you want `man` to search in, in the command by adding the section number. The following command will search for `kill` in Section 2:

```
man 2 kill
```

Note: in some shells you would give the section number with a slightly different syntax such as `-s2` instead of only 2. To find out the exact syntax you should check the man pages for `man` with the command `man man` . While you can find man pages on-line on the web, the built-in man pages have the distinct advantage of being up-to-date and relevant for the version of the operating system and programming languages libraries installed (provided the installation has been done correctly).

21.3 Shells and shellscript

Other advantages of using UNIX/LINUX and its command interpreters, the shell(s), is that the shells normally can interpret a small programming language called shellscript. By using shellscript you can program your own more advanced commands, series of commands such as detailed searches, or have the shell run several instances of programs/sets of executions in experiments etc.

21.4 Accessing the LINUX servers at KTH Kista

The EECS School has several LINUX-servers available to students. The names of the LINUX-servers are listed here:

```
atlantis.sys.ict.kth.se
avril.sys.ict.kth.se
malavita.sys.ict.kth.se
subway.sys.ict.kth.se
```

You can access them by several tools such as `putty` or `X-Win32`. `X-Win32` is a program which allows the servers to open windows on your computer. The application is free to download and install for KTH-students from: <https://www.kth.se/student/kth-it-support/software/download>

You will need a tool to get tickets in the Kerberos authentication system to be able to access the servers. On Windows you would use a tool called Network Identity manager found in the “Kerberos for Windows” package. To get new tickets in the tool you select `Credential`, to obtain new credentials (tickets), and then give your KTH user name, `KTH.SE` as realm and then your KTH-password.

In `X-Win32` you set up a connection by setting the host (vård) to one of the names in the listing above, your identity (inloggningsnamn) as your KTH-identity, command should be: `xterm -ls` and password is your KTH-password.³⁰

How to log on to the servers using different tools are described here: <https://intra.kth.se/it/arbete-pa-distans/unix/how-to-connect-1.82881>

²⁹ `man` will display the man page in an application called a `pager`. Typically it will be either `more` or `less` that is used for the pager, where `less` is more powerful than `more`. Which pager to use is determined by the environment variable `PAGER`. To exit the pager hit the ‘q’-key.

³⁰ The Linux servers use a distributed file system which you can access from a Windows machine also. That requires that you download and use OpenAFS.

22 Background: Programming language concepts

This section gives brief explanations of concepts related to programming languages and the definition of such languages. Many of the concepts are borrowed from how we define natural (spoken/written) languages. Understanding these concepts greatly facilitates learning new programming languages as one can learn them by studying their definition rather than having to read lengthy text books.

22.1 Syntax and semantics

The syntax and semantics of a programming language defines the language. These are the two most basic and important concepts.

Syntax (sv. *syntax*) writing rules, i.e. grammar, that describe how proper expressions/statements/methods/functions/programs are written

Semantics (sv. *semantik*) describes the meaning of, or in other words what the result should be when executing/evaluating, expressions/statements/programs. If a programming language has a well-defined one could (at least in theory) verify the correctness of programs written in the language.

However, many of the popular programming languages used today do not have well-defined semantics. A simple example may be the following:

```
int a = 1, b = 0;
b = a + a++;
```

What value will **b** be assigned? In a programming language with a well-defined semantics the value of **b** would be well defined. However, many programming languages simply implement how addition is performed/defined in mathematics. Given an expression $x + y$ we could evaluate the values of x and y in any order, i.e. either evaluate (determine the value of) x first and then y or evaluate y first and then x , before we do the addition. In pure mathematics the evaluation order does not matter as we do not have operators with side effects. This is not true in the example above where we have an operator, `++`, which has the side effect of assigning a new value to its operand. Thus, depending on if we evaluate the left hand operand of the addition (`a`) first or the right hand operand (`a++`), **b** would be assigned the value two (2) or three (3). In many programming languages the evaluation order of the operands of an addition is not well-defined which means that the end result could differ from system to system (compiler to compiler, virtual machine to virtual machine)

22.2 Other language related concepts

Evaluate (sv. *evaluera*) compute the value of an expression/operand

Operand (sv. *operand*) an argument to an operand (eg. a constant, value of a variable, value returned from a method/function call)

Operator (sv. *operator*) mathematical or logical operator such as `+`, `-`, `++`, `>=`, `&&`, `||`

Imperative programming language (sv. *imperativt språk*) a programming language based on variable which can be assigned values more than once³¹ and functions/methods/procedures, C is an example of an imperative language.

Object oriented programming language (sv. *objektorienterat språk*) a programming language based on the concept that everything is represented by objects with inheritance, encapsulation and polymorphism. JAVA is an example of a language that is a mainly, but not completely, object oriented language.

³¹ There are programming languages such as functional languages where “items” holding values only can be assigned a value once

Data type/type (sv. *datatyp/typ*) Variables, constants, functions and objects in imperative and object oriented languages normally has a type. The type determines how to interpret something (remember that the only thing stored in the memory of a computer are bits which can hold only the values zero and one). Associated with a type is also a semantics which determines how operations involving elements with the type should be interpreted.

Basic examples of types are integer numbers such as `int` and `Integer`, and floating point numbers such as `float` and `Float`. Objects also have types.

Different types normally have different semantics. The semantics for integer numbers differ from the semantics for floating point numbers. For instance will a division of two integer numbers normally result in a result which is an integer (normally by simply omitting the decimals) while division of two floating point numbers will result in a floating point number (retaining the decimals). The semantics for similar data types may differ between programming languages

Strongly typed language (sv. *starkt typat språk*) In a strongly typed language it is not allowed to mix types in expressions, though sometimes the user can instruct the language (compiler) that it should be allowed in an expression.

Weakly typed language (sv. *svagt typat språk*) A weakly typed language does not enforce strong type checking which means that the user can choose to mix data types in expressions and interpret data as he/she wants. This implies that the semantics may not be as strong (well-defined) as for other types of languages. C is an example of a weakly typed language intended for applications where the user sometimes need to control how memory content/data is interpreted.

Variable (sv. *variabel*) An element with a name that can contain a value. When a variable is used it is normally referred to by its name. A variable normally has a type.

Declaration (sv. *deklaration*) A variable or function/method in imperative and object oriented languages must (normally) be declared/defined before it can be created/used. In the declaration one defines the type and name of the variable/function/method and the types and names of parameters to the function/method.

Scope (sv. *synlighet*) Scope is where a variable/function/method is accessible and can be used in a program.

Local variable (sv. *lokal variabel*) A local variable is declared inside a block and its visibility, i.e. scope, is limited to the block.

Global variable (sv. *global variabel*) In some languages, as in C, it is possible to declare variables outside functions/methods. Such a variable is visible from where it is declared and to the end of the text file (compilation unit) in which it was declared. In C it is possible to make an extern declaration of a global variable making it visible in other compilation units.

Assignment (sv. *tilldelning*) To assign a variable a (new) value.

Function/procedure/method (sv. *funktion/procedur/metod*) These are related very similar concepts. It is code that can be called, in many cases with parameters. The difference is that a function normally returns a value while a procedure normally do not return a value. Method is name for functions/procedures in object oriented languages.

Parameter (sv. *parameter*) A method to pass a value to a function/procedure/method when it is called. The parameter is declared in the head of the function/procedure/method in the declaration of the function/procedure/method declaration. The parameter is essentially a local variable visible in the body of the function/procedure/method.

Function/procedure/method call (sv. *funktions-/procedur-/metodanrop*) Is performed when the execution reaches a function/procedure/method name with values for the parameters in executable code. It results in that the execution continues in the function/procedure/method. After the execution in the function/procedure/method is finished, the control of the execution returns to the next

executable statement after the function/procedure/method call (unless the function/procedure/method has caused the execution of the program to terminate)

Parameter passing (sv. *parameteröverföring*) The basic method for parameter passing in functions/procedures/methods calls is **pass-by-value** (sv. *värdeöverföring*) sometimes also called call-by value. This is a direct mapping of how this is performed on low level in the processor where parameter values are stored in registers before control is transferred to the function/procedure/method called.

Two potential issues with passing parameters by value are:

- It is infeasible to copy large data structures as arrays or objects as parameters due to both efficiency reasons as well as how the hardware is organized.
- It is not possible to have a function update a variable passed as a parameter as only the value is passed. This would for instance make it impossible to have functions for reading input into variables.

The way to work around this issue is to pass a reference, i.e. an address to where the data structure, variable, array or object is stored in memory and access the data structure by the address from the function/procedure/method. This is referred to as **pass-by-reference** (sv. *referensöverföring*).

In C, arrays are referenced by address but for other types of data the programmer has to calculate the address, pass the address as parameter and program the function to use the addresses to achieve pass-by-reference.

In other programming languages pass-by-reference is implicitly implemented by the language (i.e. in the compiler or virtual machine). In JAVA objects are accessed by references and thus passing an object as parameter is implicitly done as pass-by-reference.

23 Background: General concepts

Bootting/to boot To boot a computer means that we start up the computer and its operating system. On a PC this is done in three steps: i) first the BIOS (basic input/output system) starts; ii) the BIOS starts the boot loader which is a program which starts the operating system; iii) the last step is that the operating system takes control and starts up.

Operating system (sv. *operativsystem*) The operating system (OS) is the program running on a computer system when it is up and running. The operating system manages the resources of the computer system as processor(s), memory, storage, I/O etc. The purpose of the operating system is to support the intended use of the computer system. Thus there are different types of or tailored operating systems for PCs, real-time-systems, embedded systems, servers, super computers. The main tasks for most operating systems is to manage users, their processes, file systems, communication, I/O, security etc.

Kernel-level and user-level For security reasons one normally distinguishes between two levels for the execution of programs – kernel-level and user-level. The operating system typically executes in what is referred to as kernel-level. To execute in kernel-level means that one has full access to handle the resources of the computer which only the operating system should have. User programs are executed in user-level with very restricted, if any, privileges to handle the resources of the computer. Thus, whenever a user-level program needs to perform something which involves managing resources such as: creating new processes, allocate memory, perform I/O etc. it has to do this by performing system calls. In most cases the system calls are performed implicitly by the library routines the program calls.

System call (sv. *systemanrop*) System calls are the functions that the operating system provides as an interface (API) for user processes to call to access the services of the operating system. The system calls include functions to manage processes, allocate memory and perform I/O etc. The system calls are normally implemented in C as most operating systems are programmed in C. Other programming languages typically can access the system calls by wrapper methods in libraries.

API (*application programmer interface*) An API is the set of functions/methods provided by a service/abstract data type (ADT) to be used by user programs for them to use the service. The system calls of an operating system is one example of an API.

Shell/command interpreter (sv- *kommandotolk*) A shell is a user level program executed in a process on top of the operating system. A command line oriented shell prompts the user for commands, reads what the user inputs on the command line and then checks what to do. Either the command is a built-in command, a command performed by starting another program such as a program (ls) to list the content of a directory or it is to start a process in which a program is executed. Most of these actions involves using system calls.

Execute (sv. *exekvera*) to run a program in a process.

Runtime system When a program is executed it is normally executed in a context called a runtime-system. The purpose of the runtime-system is to start the execution of the program by calling the first function/method in the program, in C and JAVA this is a `main()` method/function. The runtime-system also handles some error situations and the termination of the program. For JAVA the runtime-system is called the Java Runtime Environment (JRE) which includes the JVM.

Process (sv. *process*) a program is executed in a process. Processes are created and handled by the operating system. User-level programs can ask the operating system to create and terminate processes, and change the program to be executed in a process by system calls

Editor (sv. *editor*) A text editor is a program which allows the user to write (edit) text. Text editors should not be confused with word processing programs. A word processing program such as MS-Word, is used to both edit and format text. The formatting will add information in the file which is not directly visible (other than how it affects the formatting). This additional information would be visible to a compiler if a source code file was created by a word processing program. In such case the compiler

could not interpret what it reads in the source code file and thus it normally would make the assumption that the program is corrupt and cannot be compiled.

Source code/source (sv. *källkod*) Code/program written in a programming language.

To compile (sv. *kompilera*) To translate a program from source code to code that can be executed by the computer.

Compiler (sv. *kompilator*) A program which compiles (translates) source code to code that can be executed by the computer. The process of compilation often is performed in several steps. When a C-program is compiled the first step is pre-processing of the code, next syntax checks are performed and the code is translated to assembly code which then could be translated into a format that could be loaded for execution.

Pre-processor (sv. *preprocessor*) A pre-processor is an application/program which does some kind of processing to its input before this is furthered to the actual processing. You have likely used the C-pre-processor (in UNIX/LINUX this is called `cpre`) without even knowing it when compiling C-programs. The C-pre-processor reads the input file and looks for pre-processor commands which are started by a `#`, such as `#include <stdio.h>` and interprets these commands and replaces them with text (or removes text). The C-pre-processor can be used on any text file and it can be quite useful to be able to use it also for other languages than C.

Flag (sv. *flagga*) Flags are directives one can give to many programs/applications, particularly in UNIX/Linux, to prompt their behaviours. As an example one can instruct `ls` to provide more (full) information on the contents of a directory by starting it with the flag `-la`, i.e. to start it as `ls -la`. Flags are normally started by a dash ('-'). One can normally provide more than one flag to an application. The flag `-help` is implemented by many programs and should print some, hopefully, useful help to the user. Flags are transmitted to the program using the argument vector (i.e. the parameter(s) to the `main()` function in C and JAVA) Flags provided to a compiler such as `gcc` are also referred to as compiler directives.

Linker (sv. *länkare*) A program which joins several separately compiled programs into a single executable file. Larger programs often have their code divided into several code files which can be separately compiled into object code (or JVM byte code for JAVA). The linker (or link editor) checks that there is code for all methods/functions called in the program and builds a single executable file which can be loaded into memory for execution (or sent to the JVM). That code is compiled separately is standard – if you have used a library function such as `printf()` in C – you have used code that has been compiled separately.

Loader (sv. *laddare*) A program which loads an executable into memory for execution.

Object code/object file (sv. *objektkod/fil*) Compiled source code which has not yet been linked, i.e. it is not in an executable format. Typical file endings are `.o` or `.obj`. Note: this should not be confused with object orientation.

Separate compilation (sv. *separatkompilering*) For larger projects or for functions/methods that are often reused such as library functions/methods one often divide the project into smaller parts (files) which are compiled separately into object code (eg. C) or Java bytecode. When an executable is generated these files are linked together.

Executable (sv. *exekverbar fil*) compiled and linked code (machine code) complete and ready to be executed. On UNIX/Linux systems the standard format for executables and object code is ELF.

Executable and Linkable Format (ELF) On UNIX/Linux systems the standard format for executables and object code is ELF. While most engineers need not know any details about ELF two things are useful to know about ELF. A file in ELF format begins with a 32-bit integer number (four bytes) which is a **magic number**. This is a specific number that any valid ELF file should begin with. If you try to load any arbitrary file for execution where you have set the execution access permission it will not execute. This is because the loader will detect, with high probability, that the file does not start by the correct magic number. The other thing that can be useful to understand is that ELF files will

have a **symbol table**. The symbol table contains the names (symbols) of the functions/methods and global variables which should be available. Names not present in the symbol table cannot be used from other files, (in case of object files) or when loaded for execution (To test this you could try to write a C-program and declare the main-function as `static main()` which will remove the symbol name from the symbol table and then try to compile and execute it. It will fail to execute as the main function will not be found when you try to load and start the execution)

Processor (sv. *processor*) The hardware unit which executes (machine) code.

CPU (*central processing unit*) The hardware unit which executes code. In a normal computer design there are several auxiliary processors which off-loads work from the CPU by managing buses, I/O-units etc.

Processor core or core (sv. *processorkärna* eller *kärna*) Modern processors often has several units, cores, which can execute machine code. This allows several processes to be executed in parallel or the parallelization of processes (i.e. the parallelization of programs).

Primary memory or core memory (sv. *primärminne*) The main memory of a computer where processes are loaded for execution. Sometimes also referred to as RAM-memory after the memory technique used to implement the memory (Random Access Memory) Core memory refers to an older memory technology (used c:a 1955-1975) where each bit was implemented as a small ring (magnetic core) made of a magnetic material.

Cache A Cache is a small fast memory (faster than the primary memory) close to the processor in which in which a computer system stores recently used data and instructions in an effort to speed-up memory access times. Programs with poor locality cannot use the caches efficiently which results in poor execution time performance.

Locality (sv. *lokalitet*) There are two types of locality: i) **Spatial locality** (rumslokalitet) means that data used are stored relatively close in memory and; ii) **Temporal locality** (tidslokalitet) means that data are reused relatively close in time during execution. Both types of locality are important to achieve good performance from the memory systems on a computer. Good spatial locality means that when data is brought into a cache we increase the probability that we bring in data that will be used to the cache. Good spatial locality means that we increase the probability that data brought into caches will be reused (instead of replaced). By being aware of how data is stored in memory, for instance when using multi-dimensional matrices, and organizing the code to achieve good locality it is often possible to reduce the execution times by several orders of magnitude compared to a program solving the same problem by the same algorithm but with poor locality. Understanding locality is important for programmers solving large scale problems.

Bit The smallest unit of memory which can represent the values zero or one [0, 1]

Byte An 8-bit unit, normally the smallest unit one can address.

Word/word length (sv. *ord/ordlängd*) The number of bits/bytes the processor normally works with. Normally processors work with 8, 16, 32 or 64 bit word length (eg. for a 64 bit processor one can add two 64 bit integers in one clock cycle).

To allocate (sv. *allokera*) To allocate a resource means that one reserves the resource. An example is when memory is allocated for variables or resources. In JAVA memory allocation and de-allocation is done implicitly while the user has to explicitly allocate and manage memory in some cases in C. Other resources that could be allocated by the operating system are disk space, I/O-units etc.

I/O (*input/output*) I/O refers to the input and/or output of data to and from I/O units such as the keyboard, mouse, screen, disks, memory, network interfaces etc.

Compiled language(sv. *kompilerat språk*) A language that is compiled to machine instructions to be executed directly on the processor. C is typically compiled while JAVA originally was compiled to Java Byte Code which was interpreted by the Java Virtual Machine (i.e. a mixture of compilation and interpretation).

Interpreted language (sv. *interpreterat språk*) A language which is not compiled but where the language is interpreted by an interpreter. The interpreter is a program which reads and performs the instructions in programs written in the interpreted language. Shellscript is an example of an interpreted language. It is interpreted by the command shell on UNIX/Linux systems. JAVA is a language which can be compiled to machine code or compiled to Java Byte Code which is interpreted by the Java Virtual Machine (JVM).

Interpreter (sv. *interpretator*) A program which interprets an interpreted language.

Virtualization (sv. *virtualisering*) Virtualization means that there is software (program) that simulates a resource so that it can be used as if it was a real resource by another program. A simple example of virtualization is when an operating system allows several processes to execute in parallel by dividing the time it allows the processes to execute on the processes in small pieces. By this way of dividing time over several processes each process gets the illusion of it executing on its own slower processor

Virtual machine (sv. *virtuell maskin*) A virtual machine simulates a machine the user program can be run/executed on. There are two common types of virtual machines. Machines simulating a complete physical computer on which one can run real operating systems and applications as if they were run on a real physical machine (it may be somewhat slower though due to the overhead of simulating the physical machine). Examples of such systems are VMWare and VirtualBox. Another common type of virtual machines are machines interpreting a programming language without simulating a complete physical machine with an operating system executing on it. An example of this type is the Java Virtual Machine (JVM) which interpret JAVA byte-code.

Address space (sv. *Adressrymd*) Address space can refer to a physical address space which primary memory or addresses in disk space. There are also logical or virtual address spaces which are not physical addresses but the perceived addresses a program/process uses. (virtual/logical addresses are translated to physical addresses before or during execution)

Physical address (sv. *fysisk adress*) An address in the physical memory, normally primary memory but could also be a physical address on disk (secondary storage).

Logical address (sv. *logisk adress*) Typically a program uses names of variables and methods/functions as logical addresses. This is an abstraction which allows the program to be developed without the programmer having to bother about exactly where, on which particular physical addresses variables, methods, functions objects will be placed when the program is executed. Address translation from logical addresses to physical addresses are performed before or during execution of the program (translation could occur at compile time, when loading the program for execution or during execution).

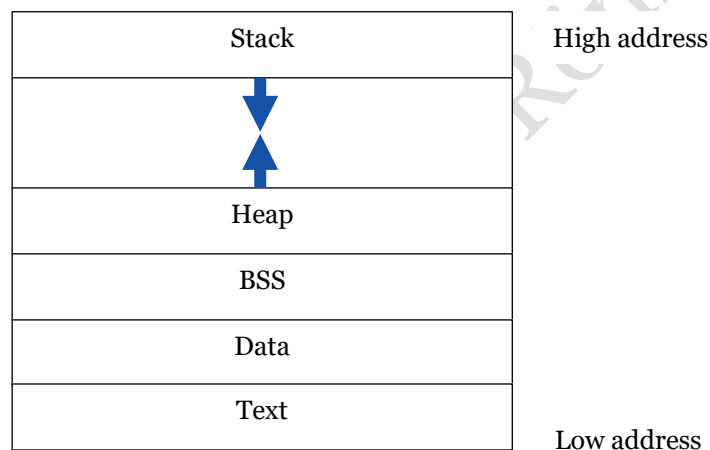
The logical address space for a process is normally divided into several areas used for different purposes:

Stack The execution stack is used to place activation records on when a method/function is called. The activation record contains parameters and local variables for the method/function, program counter, stack pointer and could contain other information also. The activation record is removed when the method/function returns (i.e. when it is finished). The stack normally occupies the highest addresses in memory growing downwards towards lower addresses.

Heap The heap is used for dynamically, during the execution, allocated data such as objects or structs which are not local variables. That is, it is used for data structures with longer life-span than local variables in methods or functions.

Data, BSS is used to store global variables and constants. The Data area is used for items with an initial value which is non-zero while the BSS (Block Started by Symbol) is used for items with an initial value of zero. The reason for distinguishing between these are that the initial values of items to be stored in the data area have to be stored in the executable (see for instance ELF Executable and Linkable Format which is the file format (organisation) used for executables on LINUX) while it is sufficient to store how many bytes the BSS area will occupy in the executable file.

Text The text area is used to store the instructions for the methods/functions of the program/process. The text area is located at low addresses.



Logical address space

Virtual memory/virtual addresses (sv. *virtuellt minne/adresser*) Most multi-user operating systems implements virtual memory. Virtual memory means that only the parts of a process that is currently being used when the process executes need to be in primary memory to execute the process. Unused parts of the process can be moved out to disk (second storage). Using secondary storage (disks) enables the parallel execution of more processes and processes which are larger than the physical memory. When a part of a process is needed for execution (the process logical memory is divided into equal size pages, typically 4kB) it is retrieved from the disk and stored in an unused part of the primary memory. This means that the exact physical addresses where things are located cannot be known until the execution and that parts of the process may move to new physical memory locations during execution. While this may seem complicated the basic ideas are not very complicated. Virtual memory is implemented by the operating system with hardware support

Recursion (sv. *rekursion*) Recursion occurs when a method/function calls itself. For the recursion to terminate there must be one or several base cases which terminates the recursive calls and one of the must be guaranteed to eventually be reached.

24 Background: Object orientation classes, objects, class and instance variables/methods

What sometimes is hard to understand is what the concepts Class, Object and Instance mean in an object oriented language. In (pure) object orientation everything is an object. Instance and Object are synonymous with the slight difference that when one use the term Instance it means one specific object. To create an object from a class is called *instantiation* or to *instantiate* an object.

While most find it easy to understand that an object is an entity which encapsulates data and methods and is instantiated from a class, it may be harder to understand what a class is. In true object orientation everything is an object. Consequently a class is also an object with the property that you can instantiate (create) objects (instances) from it.

During execution (parts of) the class is stored in memory. The class contains class methods and class variables which are stored in memory during execution. Class variables and class methods are shared by all objects from the class. This means that there is only one instance of a class variable (or class method). If one object updates a class variable all other objects belonging to the class will see the new value. Typically class variables are used for things like keeping track of how many objects have been instantiated from the class.

Class variables and class methods are declared as `static` in JAVA.

Instance variables and instance methods are part of the object/instance. That is each object has its own instance of an instance variable or instance method.

To exemplify we could look at how the population (humans) of Sweden could be modelled. We could have a class “human” which models a person. In that class we could have a class variable which counts how many instances of human, i.e. persons, there are. We could have class methods that are the same for all persons to for example return the length of a person. Information which is unique for each instance/person such as their length, social security number (sv. *personnummer*) etc. would be modelled as instance variables. Each instance (person) may also, in the general case, have its own methods to do things. For instance different persons may speak different languages.

25 Background: Part of the C- programming language

Mathematical operators: +, -, *, /, %

Comparison operators: >, <, >=, <=, ==, != (OBS! do not mix up test for equality and assignment)

Logical operators: &&, ||, ! (negation/not)

Bitwise operators: & (and), | (or), ^ (exor), ~ (bitvis complement), << (left shift), >> (right shift) (a << k left shift the bit pattern of the unsigned integer variable or constant a, k steps filling up with zero bits. Normally you operate on unsigned integers when performing bitwise operations as the sign bit of signed integers is treated specially (complicating things))

Assignment and assignment combined with mathematical operators: =, +=, -=, *=, /=, %= Obs! The left-hand side of an assignment is referred to as the lvalue (left value) it should be a variable. The expression on the right-hand side is referred to as the rvalue.

Size in bytes for the operand: sizeof(operand)

Comma operator: , (in an expression of the structure A, B the expression A will be evaluated first and then B is evaluated. The value of the expression will be that of B)

Conditional operator: expression1 ? expression2 : expression3 (if expression1 is true (has a value which is non-zero) the the value of the whole expression will be that of expression2 else it will be that of expression3)

Increment/decrement operators: ++, -- (these operators can be applied as pre- or postfix, example: a * i++ (i is incremented after its value has been used in the expression), a * ++i (i is incremented before its value is used in the expression)) OBS! Do not use these operators on variables which occur more than once in an expression as the value easily can become undefined.

Expression (sv. uttryck) An expression consists of a number of operands and operators, it can be empty. Examples. i + 1, tal > 4, i > 1 && j < 20.

All expressions in C have a numerical value. An expression is true if it has a value which is non-zero, if the value is zero the expression is false. An empty expression has a truth value of true.

A logical expression is evaluated from left to right only until the truth value of the whole expression is known.

Statement (sv. sats) A statement is an element of the language which is to be (can be) executed in a program. In C there are five types of statements. Examples:

Declaration: int tal; short i = 4;

Assignment: tal = a + 4711;

Function call: printf("hello\n");

Control statement: while(i < 2) statement;

Block: {
zero or more statments
}

Common integer data types in C

long	>= 32 bits
int	>= 16 bits (normally mapped to the length of a word of the processor architecture)
short	>= 16 bits
char	8 bits

The integer data types are signed (can hold/represent both positive and negative numbers)- It is possible to use a prefix, `unsigned`, to declare non-signed integer variables, ex: `unsigned int tal;`

Declaration of variables, global and local variables

Variables in C could originally only be declared in two ways:

Either the variable is declared outside functions in which case it is a **global** variable which is visible (can be used) from the point in the source code to the end of the source code module (file). It is allocated in the Data/BSS area and it is initialized to zero unless it has been initialized to another value in its declaration. A global variable can, unless it has been declared as **static**, be accessed from other source code modules from which the executable is built if it is declared as **extern** in the source code module where it is accessed.

Local variables are declared at the beginning of a block and are visible inside the block from the point where they were declared. Local variables are not initialized (i.e. has unknown starting values) unless they are explicitly initialized in their declarations.

Type definitions – to create your own names for data types

Type definitions is a way to use your own names for existing data types. It is something which can greatly increase the legibility and ease of understanding the code. It is best practice to use this opportunity!

typedef *datatype name* *name* can be used as a new name for *datatype*

`typedef int weight;` `weight` can be used as a new name for `int`

Typecast (or simply cast) to interpret the type of data as you intend

Sometimes it is necessary to be able to interpret the type of variables, constants or functions in a specific way in C. This is achieved by putting the data type we want to enforce in parentheses before the variable, constant or function. Ibland kan man i C behöva tolka data (variabler, konstanter eller funktioner) på ett visst sätt.

Example: `(int) 3.75` means that we interpret the floating point constant `3.75` as an integer number by removing the decimals, i.e. as `3`

C sometimes performs automatic type conversions (casts) when data of different types are mixed in an expressions. It can also be performed for parameters in function calls. Normally the types of the data used as parameters are converted to the types of the parameters in the function prototype/declaration. If the types of the parameters are unknown, that is if there has been no visible function prototype or declaration before the function is called, all parameters will be converted to the type `int`.

Common control (conditional) statements

Control statements are used to control the path of execution through the source code. The most common control statements are: `if-else`, `while` och `for`.

<code>if(expression) statement</code>	if <i>expression</i> is true then <i>statement</i> is executed
<code>if(expression) statement1</code>	if <i>expression</i> is true then <i>statement1</i> is executed else <i>statement2</i> is executed
<code>while(expression) statement</code>	as long as <i>expression</i> is true <i>statement</i> will be executed
<code>for(expression1; expression2; expression3) statement</code>	as long as <i>expression2</i> is true <i>statement</i> will be executed. <i>expression1</i> is evaluated once when the <code>for</code> -statement is entered before <i>expression2</i> is evaluated for the first time. <i>expression3</i> is evaluated after each time <i>statement</i> has been executed.

Functions

Functions are used to partition the source code into manageable pieces and to enable reuse of code. Functions are declared in *function declarations (definitions)*. *Function prototypes* are used to make the interface of a function (return type and parameter types) accessible other source code modules than where they are declared (defined). All functions in C returns a value, that is all functions has return type. `void` is used as the return type for functions not returning any value. If the return type is not defined in the function declaration (definition) C assumes the function returns a value of type `int`. If there is no visible function declaration (definition) or function prototype the C compiler cannot know the return type or types of the parameters in which case the compiler assumes these are of type `int`.

Function declaration

```
returntype functionname (parameter type- and name list)
block
```

Function prototype

```
returntype funktionnname (parameter type list)
```

In a function prototype it is sufficient to only provide the types of the parameters (the names are not useful in this case)

To return a value from a function

`return` används för att returnera ett värde från en funktion. När `return` exekveras avbryts exekveringen av funktionen och värdet av *uttryck* returneras och "ersätter" funktionsanropet.

```
return uttryck;
```

void

`void` is a keyword which can have different meanings in C depending on the context in which it is used. `void` is used as the return type of functions which do not return values (in other languages such functions are often called procedures). It is also used to declare empty parameter lists. Ex:

```
void noReturntypeNoParameters (void)
```

`void` is also used as generic type for pointers (`void *`)

Parameter passing

All parameters to functions are passed by value. This means that it is the values of the parameters supplied in a function call that are passed (copied) to the local variables which are the parameters in the function.

Pointers (sv. pekare)

Pointers are a type of variables where the value of the variable is an address. The type of the pointer identifies how to interpret what is found at the address the pointer contains. Pointers are declared by a star (*) before the variable name. One can declare pointers in several layers, that is one can have pointers to pointers to pointers... This is achieved by adding a star (*) per pointer level in the declaration. In the example below a pointer to `int` is declared. That is the variable `intPtr` will contain an address to a position in memory where an integer value is stored. Thus it has the type pointer to `int`.

```
int * intPtr;
```

To calculate the address to a variable (or function) &

The address (normally the logical address) of a variable or function is calculated by placing an ampersand (&) in front of the variable or function name. In the example below the variable (pointer) `intPtr` initialized to contain the address of `intNum` (or in other words: point to `intNum`).

```
int intNum, * intPtr = &intNum;
```

That C allows us to have pointers to functions means that we can call functions by dereferencing pointers to functions and that we can pass pointers to functions as parameters to functions (which for example allows us to implement generic sorting functions)

To dereference a pointer

To dereference a pointer means that we go to the address pointed to by the pointer and either fetch a value from the address by interpreting what we find at the address by the type of the pointer or that we store (write) a new value to the memory at the address. To dereference a pointer one puts a star (*) in front of the pointer in an executable statement (statements but declarations are executable). One can dereference pointers in several layers by adding stars (useful to dereference pointers to pointers to pointers ...) In the example below `intNum` is set to the value four (4) by accessing it through the pointer `intPtr`.

```
int intNum, * intPtr = &intNum;
```

```
*intPtr = 4;
```

Reading (interpreting) declarations

To determine the type of a variable in a declaration in C you should always read the declaration from right to left (unless there are parentheses in the declaration which alter the order). Reading the declaration below yields: 1) pointer to; 2) `int`

```
int * intPtr;  
  ^  ^  
  2  1
```

Pointers in C and references in JAVA

A reference in JAVA is similar to a pointer in C in the sense that it is a reference to where in memory we find an object (i.e. it is address). However, while we have to calculate addresses and dereference pointers explicitly in C this is done implicitly by JAVA.

Why do we need pointers?

Pointers (or references) are needed to enable us to do many of the things we expect to be able to do in languages like C or JAVA. For example we need them to be able to implement linked data structures but also some very basic things. Assume that we want to read a value to an integer variable from the keyboard. To do this we would call a function like `scanf()`.

```
int a = 0;

scanf("%d", a);
```

If we try to call `scanf()` with the parameters above we will pass a pointer to a string containing `"%d"` as the first parameter and the value zero as the second parameter. The value zero is completely useless to be used in `scanf()` when it should update the variable `a`. To update the variable `a` it needs to write the new value into the memory address where `a` is stored. To enable `scanf()` to do this we need to pass the address of `a` as parameter (`&a`) instead of the value of `a`.

```
int a = 0;

scanf("%d", &a);
```

26 Background: Basic functions for I/O in C

To use the appropriate (correct) tools is important for all engineers. For a programmer it is important to be able to use an appropriate programming language to solve a specific problem. In some cases where execution speed and/or resource usage (such as memory) is important it may be an advantage to use C instead of JAVA. C has less overhead than JAVA due to a less complex run-time system and it has efficient implementations of library functions to access the services of the operating system by system calls. However, the drawback of using C is that it may be harder to develop your code and you need to thoroughly understand what you are doing to avoid errors.

In this course you will find that particularly for I/O heavy applications C will have an advantage over JAVA. In many cases it is easier to program I/O in C compared to doing it in JAVA and the execution time can often be heavily reduced. Reduction of the execution time by up to a factor 10 is not unrealistic³² for specific problems.

In this course you will find it useful to be able to use the following functions for I/O in C:

Function	Application
<code>getchar()</code> , <code>getc()</code>	Read and return a character from <code>stdin</code>
<code>putchar()</code> , <code>putc()</code>	Write a character to <code>stdout</code>
<code>scanf()</code>	Read formatted input from <code>stdin</code> such as numbers, strings etc.
<code>printf()</code>	Write formatted output to <code>stdout</code> such as numbers, strings etc.
<code>isalpha()</code> , ...	<code>isalpha</code> is one of a family of functions to determine the type of a character such as if the character is alphanumerical, low-case, upper-case, white space etc.

To find out the details of the above functions we recommend that you use the `man` facility on a UNIX/LINUX machine.

³² The possibility to drastically reduce the execution time by using C is highly relevant for or anyone planning to take the "ADK"-course in the Computer Science Master

27 Background: References, pointers, addresses and memory management

References, pointers and addresses are related concepts which are important when working with data structures. References and pointers are needed if we want a method/function to be able to change the value of a variable in C or a member of an object in JAVA. It also enables us not to copy large amounts of data between methods/functions when we use arrays or objects.

27.1 Pointers and addresses in C

In C you may calculate the addresses of variables and functions, i.e. where in memory a variable or a function is stored. This is done by the `&` operator. By placing an `&` in front of the name of a variable or function we will calculate its address.

```
int a = 47;           // declare a variable a and initialize it to 47

&a                   // the address of the variable a
```

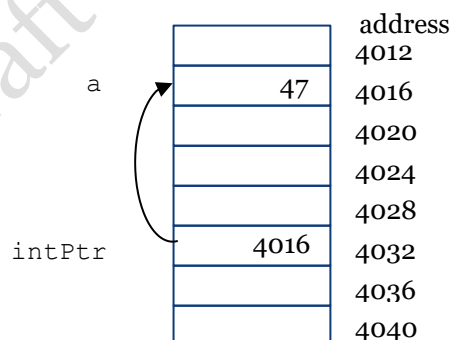
A variable containing the address of something is called a *pointer* in C. A pointer should always have a well specified type defining how to interpret what is found at the address the pointer holds. A pointer is declared by placing an asterisk in front of the name of a variable in the declaration.

```
int a, *intPtr;      // intPtr has the type "pointer to int"

intPtr = &a;         // intPtr is set to point to a,
                    // i.e. intPtr will contain the address of a
```

When reading a declaration in C to determine the type of a variable it is easiest to read the declaration right-to-left (provided there are no parentheses changing the order in which to assess the type). In the above example we start by reading: `intPtr` is `a`, then we find an asterisk meaning, *pointer to*, and finally we see `int`. Thus we can conclude that the type is: `intPtr` is a *pointer to int*.

To understand how pointers work it is a good idea to try to draw a picture of what is happening in the memory. For the above example there are memory allocated for two variables: `a` which most likely is a 32 or 64³³ bit integer and `intPtr` which is a 32 or 64³⁴ bit pointer to `int`. In the figure below we only display addresses which are a multiple of four bytes, assuming a 32 bit system (though normally we would expect to be able to address each byte individually).



Dereferencing a pointer means that we follow the pointer to go to the address it contains/points to. This is done by using the asterisk in front of a pointer variable in an executable statement (i.e. not in a

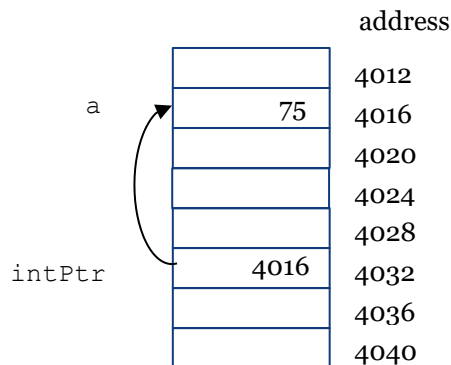
³³ This is normally determined by if we have a 32bit or 64bit processor

³⁴ This is determined by if we have a 32 bit or 64 bit operating system

declaration of a variable). The following example shows how we can use our `intPtr` to change the value stored in the variable `a`.

```
*intPtr = 75;
```

This means that we follow the pointer to the address 4016 where `a` is stored and interprets what starts at the address 4016 as an `int` and changes (i.e. writes) the value 75 to this address. Resulting in the following situation.



27.2 A case for why we need pointers (references)

Skriv om parameteröverföring I C scanf

27.3 Memory management in C

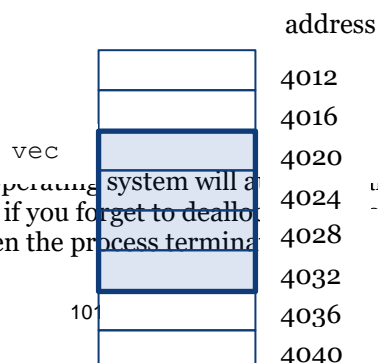
In JAVA the run-time system will handle memory allocation/deallocation. When you instantiate an object in JAVA the underlying system will reserve (allocate) memory for the object. When the object no longer can be reached through any direct or indirect reference, the system will automatically free (deallocate) the memory by a mechanism called garbage collection³⁵.

In C on the other hand there are no built in mechanisms that automatically allocates or deallocates (garbage collects) memory for dynamically generated objects (in fact there is no such thing as objects as defined in JAVA in C either). This means that the programmer explicitly has to manage memory allocation in many situations.

Assume we want to create an array with a number of elements which is not known a priori (before execution starts). How can we do that? Before we consider that problem we need to understand how arrays with a size known at compile-time works.

In C, memory is automatically allocated for an array which has a size which is known at compile time. The name of the array will be a constant pointer to the first element in the array. Array indices always start at zero (0) to facilitate addressing in the array. Consider the following example:

```
int vec[4];
```



³⁵ Note that when a process terminates the operating system will automatically reclaim all memory it has allowed the process to use. That is, even if you forget to deallocate memory, it will be reclaimed when the process terminates.

In the above example we have reserved the addresses from 4020 through 4035 to store the four elements of the array `vec`. `vec` is a static pointer (constant which cannot be changed) with the value 4020 which is the starting address of the array.

When referring to an element of the array such as `vec[2]` the C language will find this element by calculating its starting address and dereferencing that address in the following way:

```
*(vec + 2 * sizeof36 (int)) -> *(4020 + 2*4) -> *(4028)
```

`vec[X]` means that we should access the X th element of the array. In our case X is two, which is the third element which is located at address 4028. This address is calculated by taking the starting address of the array, 4020, and adding X times the size of the element in bytes. Note that to be able to calculate addresses in this way we have to know the type of the pointer, i.e. what type of data it points to. JAVA and virtually any other programming language implementing arrays use address calculations as the above when indexing into arrays.

Warning: You should also note that it is possible to index out of bounds in C without any run-time system detecting this. Thus in the above example we could for example do operations such as `vec[20] = 0;` or `vec[-2] = 14;` which would overwrite memory spaces possibly used for other purposes without any run-time errors. This could lead to erroneous behavior of your program (if for instance the value of a variable has been changed un-intentionally) or that the program eventually (at a later time) crashes. Thus these kind of errors can be very hard to detect. And it is virtually impossible to detect them by printing trace messages or using debuggers. The best way to avoid such problems is to be careful and fully understand what one is doing.

27.4 Dynamic memory allocation in C

So what if we want to create an array where we do not know the size of the array until we execute the program? In this case we would have to allocate memory dynamically (on the heap³⁷). Dynamic memory allocation is done by calls to the library function `malloc()`.

```
void * malloc(size_t size)
```

`malloc` will allocate a contiguous memory space of `size` bytes and return the address to where the space starts. If `malloc` could not find a free memory space of the desired `size`, i.e. there is no available free memory, it will return a `NULL` pointer. `void *` is a generic pointer type which can be used to address (point to) any valid address. In an assignment you should cast a pointer of type `void *` to the type of the pointer you assign it to.

³⁶ `sizeof()` is an operator in C which calculates the size of an element/type in bytes

³⁷ If you do not know what the heap is you need not worry – you will learn about that in the Operating systems course. However, a process normally has four memory areas: a **stack** which is used for activation records created when functions/methods are called, **BSS** which is used for static/global data, the **heap** used for dynamically (during run-time) allocated data and the **text** area where the instructions for the methods/functions of the program is stored.

When a memory space allocated by `malloc()` no longer is used it should be returned to the run-time system³⁸ by a call to `free()`. The `ptr` parameter to `free()` should be a pointer to an area allocated by `malloc()`. Note that `free()` does not check that this condition holds and you can call `free()` with any valid address which very likely could result in that your program no longer would function correctly.

```
void free(void * ptr)
```

Now we have all the necessary elements to be able to dynamically allocate memory for an array in C which has a size that is not known until execution time. The following code snippet shows how this can be done (note that we do not show how to free the memory when we no longer have use for the array):

```
int size, //number of elements in the array
    *dynVec; // a pointer to the allocated memory

printf("give the number of elements: ");

scanf("%d", &size); // read the number of elements and store
                    // at the address where size has been
                    // allocated

assert(size > 0); // make sure size is > 0, else throw error

//allocate memory, cast the pointer returned to type int *
dynVec = (int *)malloc(size*sizeof(int));

dynVec[0] = 14; //use dynVec as a normal array
```

³⁸ Think of the run-time system as a set of functions/methods used to implement different services a process needs. In this case it is the library functions to manage dynamically allocated memory, `malloc()`, `free()`, `realloc()`, which keeps track of and manages memory obtained from the operating system through system calls such as `mmap()` or `brk()` (`brk()` is old and no longer used).

27.5 References in JAVA

JAVA does not give the programmer the same freedom of calculating addresses and accessing any address for the reason of avoiding what could be the potential cause of errors and possible security risks.

In fully object oriented languages³⁹ there is no need to have references other than for being able to point to objects. Thus we will think of a reference in JAVA is a variable that contains the address of where an object is stored in memory. Consider the following example:

```
private class Node
{
    Item item;
    Node next; // reference variable that contains a
}             //reference to a Node object

//instantiate an object from the class node
//and set first to be a reference to the new object

private Node first = new Node();
```

In the case above where we instantiate a `Node` object by `new`, the JAVA run-time system (virtual machine) will allocate memory for the new object and return a reference to the new object and store the reference (~address) to it in `first`. What goes on behind the scenes is similar to allocating memory by `malloc()` in C and then initiating the fields in the new object (`item`, `next`) to zero/`NULL` (in this case) and then returning the address to the memory allocated for the object.

³⁹ JAVA is not a fully object oriented language as it allows for basic data types such as `int`, `double` etc. for performance reasons where the variables of such types are not objects in an object oriented meaning.

28 Background: How to build linked data structures in C

In C there are no concepts of objects or classes. Instead you have to rely on something called `struct` to implement elements/nodes. A `struct` is a way of managing several fields of data as one item. The syntax is as follows:

```
struct name {  
  
    field variable declarations  
  
} declaration of structs ;
```

Items set in boldface are mandatory, items in italics are optional.

A `struct` representing a person with the name and date of birth could be declared as:

```
struct person {  
    char * name;  
    int year;  
    int month;  
    int day;  
};
```

The following would create a `struct` variable from the data type we defined above:

```
struct person kalle;
```

The fields of `kalle` can be accessed by the dot (.) operator:

```
kalle.year = 97;
```

However, in most cases structs are handled by pointers. The following declares a pointer to a person `struct` and initiates it to point to `kalle`.

```
struct person *ptr = &kalle;
```

To access the fields of the `struct` `kalle` that `ptr` points to, we can use two methods:

```
(*ptr).month = 12;    // dereference ptr and then select a field  
  
ptr->month = 12;       // same as the above but uses the "arrow" operator
```

When working with structs it is often convenient and also helps the readability of the code to use the possibility to create your own names for data types by using `typedef`. The following creates new names for the data types `struct person` (`aPerson`) and a pointer to a `struct person` (`personPtr`).

```
typedef struct person {
    char * name;
    int year;
    int month;
    int day;
} aPerson, *personPtr;
```

That is, if we had had the above type definitions we could have used them to declare `kalle` and `ptr` as:

```
aPerson kalle;           // same as struct person kalle;
personPtr ptr = &kalle;  // same as struct person *ptr = &kalle;
```

If we want to dynamically create structures, for instance when creating a linked list, we have to dynamically allocate memory for the structs. The following code snippet defines a `struct` and associated names for `struct` elements and pointers suitable for implementing a single linked list and allocates memory for a new element, updates the fields in the element and then deallocates the element:

```
typedef struct x {
    int data;
    struct x *next;
} elem, *elemPtr;

elemPtr ny = (elemPtr) malloc(sizeof(elem)); //note: the memory space
                                              //allocated by malloc may
                                              //contain old data, i.e. you
                                              //cannot assume all
                                              //fields will be 0 (NULL)
                                              //(unless you initiate them)

ny->data = 4;

ny->next = NULL;

free((void *) ny);
```

29 Background: Common pitfalls when programming C

There are some common errors that inexperienced (and sometimes also experienced) C programmers may encounter. Below you find some of these.

29.1 Test for equality in the condition of `if`, `for` or `while` statements

In C all statements have a value which can be used for assignments but also for tests on whether or not it evaluates to TRUE/FALSE. In C a value of zero (0) evaluates to FALSE.

An error which may occur is that one confuses test for equality which is written by two equality signs `==`, with assignment which is a single equality sign. Assume we want to execute a `while`-loop as long as an integer variable `x` has the value 4962, *i.e. the condition has the form of a comparison of a variable and a constant*. We naïvely write the `while`-statement as:

```
while (x = 4962) ... //no error from the compiler as this is allowed in C
```

What we have created in the above example is an infinite loop (i.e. the loop will never terminate) as we assign the value 4962 to the variable `x` in the condition part of the `while` statement before we check the condition in each iteration of the loop. We then test to see if `x` is zero, in which case we break the loop, or in case it is non-zero we will execute the statement in the loop. As the variable `x` is always assigned the value 4962 before we check the condition, the value of `x` that we check in the condition, will always be non-zero and the loop will never terminate.

A simple way to avoid this is to always place the variable to the right of the equality signs and the constant to the left. Had we made the same mistake as above the compiler would have recognized it and issued an error telling us that we cannot assign a new value to a constant. This is often somewhat cryptically expressed as that we have: “an illegal lvalue”. In C-parlance an lvalue is the value to the left in an assignment and an rvalue is the value to the right.

```
while (4962 = x) ... //compilation error as we cannot assign to a constant
```

This error message would alert us so that we could correct the code to:

```
while (4962 == x) ...
```

29.2 Confusing bit-wise and logical operators

As C was designed also to cover low-level programming controlling hardware, it has not only logical but also bitwise operators. This may lead to hard to understand results if one confuses the bitwise operators for *or* which is written as a single vertical bar (|) and the bitwise *and* operator which is written as a single ampersand (&) with the logical *or* written as double vertical bars (||) and logical *and* written as double ampersands (&&). Consider the following example:

```
int x = 1, y = 2;

if(x && y) statement // statement will be executed as both x and y are
                      // true (non-zero)

if(x & y) statement  // statement will not be executed as bitwise AND of
                      // 1 & 2 is zero

x    0 1
y    & 1 0 bitwise and
-----
      0 0
```

29.3 Indexing out of bounds and addressing errors

C allows you to access basically any valid memory location in the (virtual) address space of the process in which your program executes. This is to allow C-code to control hardware and enable it to be used for implementing operating systems etc. Thus most C-compilers assume that you know what you are doing and allows for code to pass the compilation without error or warning messages that would occur in many other languages. There are also few run-time controls checking for instance if you index out-of-bounds of an array and overwrite things. That is if you are not careful you may overwrite data in your program, which in general only will result in faulty behavior. But you may also overwrite things vital for the execution of the program, such as the execution stack, which in most cases would cause your program to crash. In both these circumstances the effects of the overwriting, in the form of execution errors or a crash, may occur much stage in the execution which often means that it may be hard to find the cause of the problem. So how do you avoid such problems?

There are tools which can check your code before you compile it and issue warnings when it finds parts of the code which it suspects could be erroneous and there are other tools which may allow for more extensive run-time tests including searching for memory leaks. It may also be useful to know how to use a debugger.

However, the best way to avoid such problems (and most problems encountered when programming in any language) is to truly understand what you are doing.

30 Index

&, 100
*, 100
., 83
.., 83
|, 108
||, 108
~, 83
Address space, 91
addresses, 100
Agile, 12
allocate, 90
Analytic debugging, 74
API, 88
assert, 103
Assignment, 86
asterisk, 100
binary search tree, 34
Bit, 90
bitwise operators, 108
black-box testing, 72
boot, 88
break point, 76
buffer, 77
Byte, 90
cache, 90
cd, 83
class methods, 93
class variables, 93
command interpreter, 78, 88
Comments, 11
compiled language, 91
compiler, 89
core, 90
core memory, 90
cpp, 74
C-preprocessor, 74
CPU, 90
current directory, 83
Data type, 86
debugger, 75
debugging, 73
declaration, 100
Declaration, 86
dereference, 105
Dereferencing, 101
device driver, 80
double linked circular list, 27
Dynamic memory allocation, 102
editor, 88
ELF, 89
environment variables, 81
EOF, 15
Evaluate, 85
Executable, 89
Executable and Linkable Format, 89
execute, 88
exhaustive testing, 72
file descriptors, 78
filter, 82
Flag, 89
free(), 103
function, 86
function call, 86
garbage collection, 101
getc(), 99
getchar(), 15, 99
glass-box testing, 72
Global variable, 86
head, 27
HOME, 81
home directory, 81, 83
I/O, 91
Imperative programming language, 85
Indexing out of bounds, 108
infinite loop, 107
infix, 35
input, 78, 80
instance methods, 93
Instance variables, 93
instantiate, 93, 104
instantiation, 93
integration testing, 71
interpreted language, 91
interpreter, 91
isalpha(), 99
Jackson Structured Programming, 68
Javadoc, 11
JSP, 68
Kerberos, 84
kernel-level, 88
less, 82
linked data structures, 17
Linker, 89
LINUX-servers, 84
Loader, 89
Local variable, 86
locality, 90
logical errors, 74
loop-invariant, 71
ls, 83
lvalue, 94, 107
magic number, 89
malloc(), 102
man pages, 83
memory allocation, 101
method, 86
method call, 86
module testing, 71
Network Identity manager, 84
Object code, 89
Object oriented programming language, 85

- operand*, 85
- operating system**, 88
- operator*, 85
- ordered tree, 34
- output, 78
- PAGER, 81
- Parameter**, 86
- parameter passing**, 87
- pass-by-reference, 87
- pass-by-value, 87
- PATH, 81
- path to working directory**, 83
- performance related error, 74
- Physical address, 91
- pipe, 81
- pointer*, 100
- pointers, 100
- Polya's method, 67
- postfix*, 35
- prefix*, 35
- Pre-processor**, 89
- Primary memory**, 90
- printenv, 82
- printf()*, 99
- priority queue, 26
- procedure**, 86
- procedure call**, 86
- process**, 88
- Processor**, 90
- Processor core*, 90
- production code, 74
- proof by induction, 71
- putc(), 99
- putchar(), 15, 99
- pwd**, 83
- Recursion, 92
- recursive methods, 34
- redirect, 77
- redirecting input, 78
- references, 104
- References, 100
- regression testing**, 72
- runtime system**, 88
- rvalue, 94, 107
- scanf(), 99
- Scope**, 86
- sections, 83
- Semantics**, 85
- sentinel element, 27
- Separate compilation**, 89
- shell, 78, 84, 88
- shellscrip, 84
- side effect, 85
- single linked linear list, 17
- sizeof(), 102
- sort, 82
- source code**, 89
- spatial locality, 90
- stderr, 77
- stdin, 15, 73, 77, 99
- stdout, 15, 73, 77
- streams, 77
- Strongly typed language**, 86
- struct, 105
- symbol table**, 90
- Syntax**, 85
- system call**, 88
- system calls, 78
- system test**, 72
- temporal locality, 90
- Test Driven Development, 12
- Test for equality, 107
- testing*, 71
- tests, 12
- to compile**, 89
- Top-down, 69
- Trace printouts*, 74
- type**, 86
- typedef, 106
- UML, 68
- Unified Modelling Language, 68
- unit testing**, 71
- user-level**, 88
- validation*, 12, 71
- Variable**, 86
- Weakly typed language**, 86
- verification*, 12, 71
- white-box, 72
- virtual addresses, 92
- Virtual machine, 91
- Virtual memory, 92
- virtualization, 91
- void *, 103
- word, 90
- word length, 90
- X-Win32, 84