Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation
4. Data Structures
   4.1. Introduction
   4.2. Graphs
   4.3. Use of Graphs
   4.4. Trees
   4.5. Use of Trees
   4.6. Linked Lists
   4.7. Queues and Stacks
5. Algorithms
The characteristics of the double linked lists allow for the implementation of 2 important data structures.

- **Queues:** FIFO – „First in First out“
  Elements are extracted in the order in which they enter the queue.
  Everyday examples: Bank counter, supermarket, waiting room at the doctor

- **Stacks:** LIFO – „Last in First out“
  Elements are extracted in reverse order of how they enter the line.
  Everyday examples: plate sequence, overcrowded busses, recursive problems
Queues

Operations:
• The first element is „served“
• A new element is added behind

Application:
example: Administration of print orders or processes
Minimum Operations:

- **write**
  insert (add) elements at the end
  
  ```java
  void enqueue(Object o)
  ```

- **delete**
  remove an element from the beginning
  
  ```java
  void dequeue()
  ```

- **Read and search**
  returns the beginning element, without removal
  
  ```java
  Object front()
  ```

Optional Operations:

- **Number of elements**
  
  ```java
  int size()
  ```

- **Queue empty?**
  
  ```java
  boolean empty()
  ```
Queues - Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Content of Queue</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Queue()</td>
<td>()</td>
<td>-</td>
</tr>
<tr>
<td>enqueue (1)</td>
<td>( 1 )</td>
<td>-</td>
</tr>
<tr>
<td>enqueue (3)</td>
<td>( 1, 3 )</td>
<td>-</td>
</tr>
<tr>
<td>front ()</td>
<td>( 1, 3 )</td>
<td>1</td>
</tr>
<tr>
<td>enqueue (7)</td>
<td>( 1, 3, 7 )</td>
<td>-</td>
</tr>
<tr>
<td>dequeue ()</td>
<td>( 3, 7 )</td>
<td>-</td>
</tr>
<tr>
<td>front ()</td>
<td>( 3, 7 )</td>
<td>3</td>
</tr>
<tr>
<td>dequeue ()</td>
<td>( 7 )</td>
<td>-</td>
</tr>
<tr>
<td>front ()</td>
<td>( 7 )</td>
<td>7</td>
</tr>
</tbody>
</table>
Operations:
- **Adding is only possible at one end (the back).** The oldest elements are at the front.
- **Removal at the same end**

Applications:
Examples:
- Calculations of multi-body systems and simulations (sequence of transformation matrices)
- Saving of local variables at function calls
Minimum Operations:

- **write**
  - inserts an element at the end
  - void push(Object o)

- **delete**
  - removes an element from the end
  - void pop()

- **Read and search**
  - returns the last element without deleting
  - Object peek()

Optional Operations:

- **Number of elements**
  - int size()

- **Stack empty?**
  - boolean empty()
## Stacks: Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Content of Stack</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Stack()</td>
<td>()</td>
<td>-</td>
</tr>
<tr>
<td>push (1)</td>
<td>(1)</td>
<td>-</td>
</tr>
<tr>
<td>push (3)</td>
<td>(1, 3)</td>
<td>-</td>
</tr>
<tr>
<td>peek ()</td>
<td>(1, 3)</td>
<td>3</td>
</tr>
<tr>
<td>push (7)</td>
<td>(1, 3, 7)</td>
<td>-</td>
</tr>
<tr>
<td>pop ()</td>
<td>(1, 3)</td>
<td>-</td>
</tr>
<tr>
<td>peek ()</td>
<td>(1, 3)</td>
<td>3</td>
</tr>
<tr>
<td>pop ()</td>
<td>(1)</td>
<td>-</td>
</tr>
<tr>
<td>peek ()</td>
<td>(1)</td>
<td>1</td>
</tr>
</tbody>
</table>
Recommended Literature


[FMUP05] Vorlesung „Formale Methoden und Programmierung“, 2005

[Schn98] H.-J. Schneider (Hrsg.), „Lexikon der Informatik“, Oldenbourg Verlag, 1998

[Wid02] Ottmann, Widmayer, „Algorithmen und Datenstrukturen“, Spektrum Akademischer Verlag, 2002


[Wald87] Waldschmidt, „Einführung in die Informatik für Ingenieure“, Oldenbourg Verlag, 1987
Recommended Literature

[Sen05] Sen, S.: "Bondgraphs – a formalism for modeling physical systems“, School of Computer Science, 2005


Introduction

- The term “algorithm” means, generally speaking, an **exact instruction for the solution of a problem** or a specific type of problems.

- Algorithms are one of the central topics in **computer science** and **mathematics**. As computer programs or electric circuits, they steer the actions of computers or other machines.

- There exist **various representations** for algorithms. They range from algorithms being the **abstract complement** to the concrete programs tailored for a machine, up to the idea to see algorithms as programs for ideal mathematical machines.

- Theoretical computer science uses the abstraction of these mathematical machines as a **basis for calculation models**. Thus definitions and conclusions about specific **problem classes** can be formalized.
In chapter 1, the term „algorithm“ was defined as follows:

**Definition 1.2:** An algorithm is a precise, e.g., drafted in an acceptable language, finite description of a general process with performing of executable elementary (processing) steps, whose task can be accomplished with one input [Bieh00].
Examples of Algorithms

Changing the tire on a car:

1. Loosen the wheel nut
2. Lift the vehicle
3. Unscrew the wheel nuts
4. Exchange the tire with another tire
5. Screw the nuts on again
6. Lower the vehicle to the ground
7. Tighten the wheel nuts

Other everyday examples:
Recipes, repair manuals, etc.
1. Operations

- For the **formal representation** of an algorithm, a **set of elementary operations** must be defined, which can be understood and performed by the user.

  for example: mechanic
  - (car) lift
  - (wheel nut) loosen
  - (wheel nut) unscrew
  - (tire) change
  - etc.

- The instructions of an algorithm are performed in a **specific order**.
2. Objects

- **According to the instruction, objects** are handled. Objects can be of **abstract** or of **concrete nature**.

  for example: car tires
  - car
  - tire
  - wheel nuts
  - etc.

- The involved objects are formalized in a “computer comprehensible“ way by describing their **characteristics through data**.

  for example: a single tire is described by its characteristics like manufacturer, size, type, etc.
1. Formal description of the problem area and boundary conditions

**example:**

A rational number $a$ shall be multiplied with a natural number $n$

**input:** $a \in \mathbb{R}$, $n \in \mathbb{N}$

**output:** $x = n \times a$

**boundary conditions:**

- **Elementary operations:**
  
  Assignment ($\equiv$), Addition (+), Subtraction (-), equal (=)

- **run structures:**
  
  Sequence, loop (WHILE)
2. Formulation of the algorithm with the help of existing elementary operations.

**example:**

input: \( a \in \mathbb{R}, n \in \mathbb{N} \)

\[
x := 0
\]
as long as \( n > 0 \)

\[
\{ \\
x := x + a \\
n := n - 1
\}
\]

output: \( x \)

Here the **multiplication** is represented by multiple **addition**. We need the elementary operations **assignment** and **addition**. **Subtraction** and **equal operations** are also needed in order to steer the **While-loop** of the algorithm and to test the abort conditions for the multiple **addition**.
3. The general and formally described algorithm is translated by the implementation into an understandable and executable language for the computer.

**Example:** (Java)

```java
double multiply(double a, int n) {
    double x = 0;
    while (n > 0) {
        x += a;
        n--;
    }
    return x;
}
```

**Simplifies the addition and the assignment with the operator +=**

**Decrements the loop variable n and steers the number of repetitions of the loop**
• The investigation and analysis of algorithms is one of the **main tasks in computer science**, and is mostly theoretically performed (without concrete transition into a programming language).

• **Possible criteria for classifying an algorithm are** (in order of importance):
  1. complexity of calculating time
  2. complexity of storage space for cache
  3. clarity and maintainability
  4. easy verification of correctness
  5. easy implementation

[Algo04]
Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation
4. Data Structures
5. Algorithms
   5.1. Introduction
   5.2. Characteristics
   5.3. Complexity
   5.4. Design Methods
   5.5. Examples of Algorithms
Classification of the characteristics of algorithms:

- **Problem independent characteristics:**
  They refer exclusively to the considered algorithm, but not to the problem or the specification.

  **Examples:**
  - Finiteness
  - Termination
  - Determination
  - Determinism
  - Recursivity
  - Parallelism

- **Problem related characteristics:**
  They refer both to the algorithm and to the problem or specification.

  **Examples:**
  - correctness
  - efficiency
Finiteness Condition:
An algorithm must be finitely describable - that means, it must be possible to formulate it with (finite) text.

- The condition for finiteness refers to the description of the algorithm, **not to its execution**
- The description consists of a finite number of **elementary operations**. But when the algorithm is executed, an arbitrary number of operations can be executed – e.g. caused by loops. This can lead to the situation that the algorithm does **not end** (not terminate).
A counter example:

Pi $\pi$ shall be shown on the screen:

```java
System.out.print("3");
System.out.print(',');
System.out.print("1");
System.out.print("4");
```

Since $\pi$ has an infinite number of positions after the decimal point, our program (when it's done as shown on the left) must consist of an infinite number of lines (1 line per position), too.

The sequential output of single numbers of $\pi$ leads to a program code that is just as long.

⇒ The condition for finiteness is not fulfilled.

[Algo04]
Termination:
An algorithm is terminating when it reaches a result for each valid input after a finite number of steps.

• An algorithm must terminate after a finite amount of time in a controlled way.

• The actual number of performed steps can be arbitrarily large.

• Generally, it’s not possible to decide for each arbitrary algorithm if it terminates or not. Many algorithms are too complex even to be described by a mathematical set of rules.
Note: an algorithm that is described with a final source text (program text), can still have an infinite run time.

Example: (Java)

```java
double multiply(double a, int n)
{
    double x = 0;
    while (n > 0)
    {
        x += a;
        n--;  
    }
    return x;
}
```

This is an infinite loop, because `n` is not decremented in the loop: the aborting conditions in the `while` instruction are never fulfilled. This is a typical beginner error in programming with Java (and C++).
Counter example:

input: $n$

repeat
    $n := n + 1$

     till $n = 50$

output: $n$

With an input of $n < 50$ the loop is exited after $n = 50$ and the algorithm then terminates.

With an input of $n \geq 50$ the aborting condition will never be fulfilled.

$\Rightarrow$ The condition for termination is not fulfilled.
Determination

An algorithm is determined if the same parameters and starting values always lead to the same result.

• The determination properties guarantee a clear dependence of the output data from the input data. Thus it's possible to describe functions with algorithms.

• An algorithm is not determined if it's output is partly based on coincidence.
Determinism

An algorithm is deterministic when in each step of execution there exists only exactly one possibility for the continuation of the program.

- **In practice**, non-deterministic algorithms are not of big importance. Non-deterministic machines cannot by realized practically.
- Non-deterministic algorithms find use in **theoretical computer science**, in order to estimate the complexity of problem areas or to describe **quantum computers**.

http://de.wikipedia.org/wiki/Quantencomputer

A **quantum computer** is any device for **computation** that makes direct use of distinctively **quantum mechanical** phenomena, such as **superposition** and **entanglement**, to perform operations on data. In a classical (or conventional) computer, the amount of **data** is measured by bits; in a quantum computer, it is measured by **qubits**.

The basic principle of quantum computation is that the quantum properties of particles can be used to represent and structure data, and that quantum mechanisms can be devised and built to perform **operations** with this data.
Recursivity

An algorithm is considered *recursive* when it uses itself again.

They are realised as follows:

**example:**
Calculate the factorial n!

```c
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return factorial(n-1)*n;
}
```

**Recursion anchor** (without reference to itself)
If the recursion anchor is reached, the result will be immediately returned.

factorial(0) = 1

**Recursion step** (with self reference)
This defines the function with help of a recursive call.

factorial(n) = factorial(n-1) * n

**Recursion anchor:** Contains the aborting conditions of the recursion.
There are 2 types of recursion:

### Direct Recursion:
An algorithm calls itself again.

**example:**
calculate the factorial n!

\[
\text{factorial}(0) = 1 \\
\text{factorial}(n) = \text{factorial}(n-1) \times n
\]

### Indirect Recursion:
Multiple different algorithms alternatively call each other.

**example:**
Test on even or odd numbers

\[
\begin{align*}
\text{even}(0) &= \text{true} \\
\text{even}(n) &= \text{odd}(n-1) \\
\text{odd}(0) &= \text{false} \\
\text{odd}(n) &= \text{even}(n-1)
\end{align*}
\]

The „even“ function calls the „odd“ function. In this call, the parameter is decremented by 1. The functions alternately call each other until the parameter is decremented to 0.
Recursivity (3)

**Everyday Recursion:**
- Mirror image between 2 mirrors
- Audio- or Video return coupling

**Mathematical examples:**
- Factorial functions
  
  \[
  0! = 1 \\
  n! = n \times (n - 1)
  \]

- Fibonacci-order \(\text{fib}(0), \text{fib}(1), \text{fib}(2), \ldots\)
  
  \[
  \text{fib}(0) = 0 \\
  \text{fib}(1) = 1 \\
  \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)
  \]

Sources:
- [http://www.oystersareevil.com](http://www.oystersareevil.com)
- [http://www.hyperkommunikation.ch](http://www.hyperkommunikation.ch)
Recursivity (4)

- Recursion is a popular solution strategy, since it allows to formulate complex problems through a small, „elegant“ description.
- Recursive algorithms are frequently shorter than non-recursive algorithms, but are not always more efficient.
- Each iterative (= loop based) algorithm can be expressed as a recursive algorithm. On the other hand, an arbitrary recursive algorithm can not be transferred into an iterative algorithm without further algorithms.

```cpp
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return factorial(n-1)*n;
}
```

```
int factorial(int n) {
    int a = 1;
    int i = 0;
    while (i < n) {
        ++i;
        a = a*i;
    }
    return a;
}
```

<table>
<thead>
<tr>
<th>recursive algorithms</th>
<th>iterative algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>int factorial(int n) {</td>
<td>int factorial(int n) {</td>
</tr>
<tr>
<td>if (n == 0)</td>
<td>int a = 1;</td>
</tr>
<tr>
<td>return 1;</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>else</td>
<td>while (i &lt; n) {</td>
</tr>
<tr>
<td>return factorial(n-1)*n;</td>
<td>++i;</td>
</tr>
<tr>
<td></td>
<td>a = a*i;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>Same result!</td>
<td>return a;</td>
</tr>
</tbody>
</table>

5. Algorithms
5.2. Characteristics
Parallelism

An algorithm is considered parallel if it is divided in sub-tasks that can be simultaneously run by different processors.

- A single processor allows only the strict sequential execution of elementary operations.
- Multiple processors allow for the execution of different parts at the same time in parallel algorithms.

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor 1:</strong></td>
<td><strong>Processor 1:</strong></td>
</tr>
<tr>
<td>( x := 3; )</td>
<td>( x := 3; )</td>
</tr>
<tr>
<td>( y := 4; )</td>
<td>( y := 4; )</td>
</tr>
<tr>
<td><strong>Processor 2:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• **Data dependence**
  The simultaneous execution of operations that use the same data leads to undefined results.

**Example:**
Different bookings in a bank can be performed by using different processors at the same time:

**Algorithms** for single bookings:
1. Read account balance $S$ from account $K$
2. Add the amount $B$ to the read-in account
3. Insert $X$ as the new account balance of $K$

Both processors return different account balances. That means, depending on the execution time, one booking is lost.

<table>
<thead>
<tr>
<th>Processor 1:</th>
<th>Processor 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K = 3089; B = 1000$</td>
<td>$K = 3089; B = 5000$</td>
</tr>
<tr>
<td>$X := S(K)$; (1.)</td>
<td>$X := S(K)$; (1.)</td>
</tr>
<tr>
<td>$X := X + B$; (2.)</td>
<td>$X := X + B$; (2.)</td>
</tr>
<tr>
<td>$S := X$; (3.)</td>
<td>$S := X$; (3.)</td>
</tr>
</tbody>
</table>

[Algo04]
Parallelism: Limitations (2)

- If the dependencies of the parallel algorithms on different processors cannot be avoided, a **deadlock** can occur. The processors then wait respectively for a calculation result or for a resource that the other processor must provide.

- Each parallel algorithm can also be run sequentially. Computer science is today still unclear of whether or not all algorithms are able to be parallelized or not.
Example for Deadlock – Dining Philosophers (1)

- 5 philosophers are sitting at a round table. Each one is alternatively eating and thinking. When a philosopher is hungry, he grabs the two forks (on his left and on his right) and starts eating. After he finishes eating, he puts the forks back on the table and continues his thinking (Dijkstra 1965).

- Abstract program for the philosophers:

```plaintext
repeat(
    think
    get right fork
    get left fork
    eat
    release right fork
    release left fork
)
```
Example for Deadlock – Dining Philosophers (2)

- Possible dead-lock:
  For all processes $i \in \{1\ldots5\}$ we have: process $i$ has the right fork and waits for the left fork.

- If all philosophers grab the right fork simultaneously, no one can grab his left fork and waits. → **Deadlock**