Info Abi

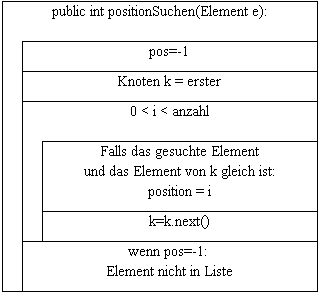
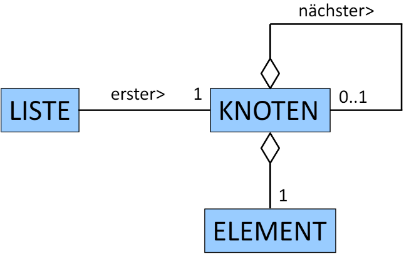
Vorallem bei HJ3 Formale Sprachen: Buch texte anschauen; Turing Machine?

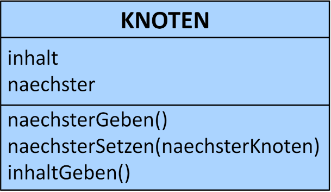
B Bäume nur für Thomas

Chomsky Hierarchie für mich schon wichtig

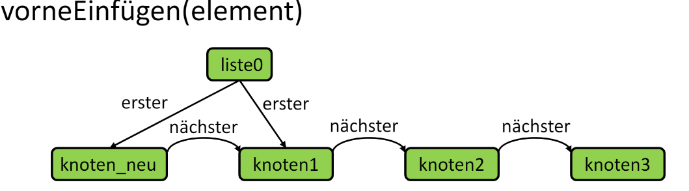
# HJ 1

## Basics

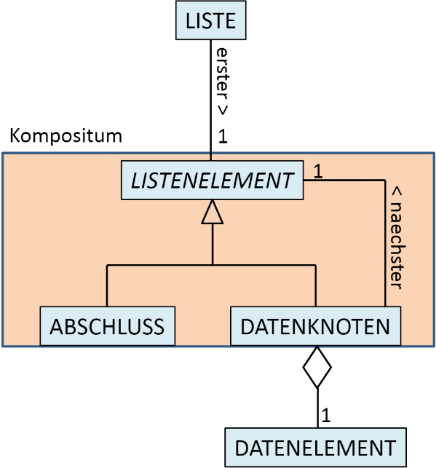
1. **Einfach verkettete Liste:** (Eine dynamische Datenstruktur -> Länge variabel)



* Iterative Lösung

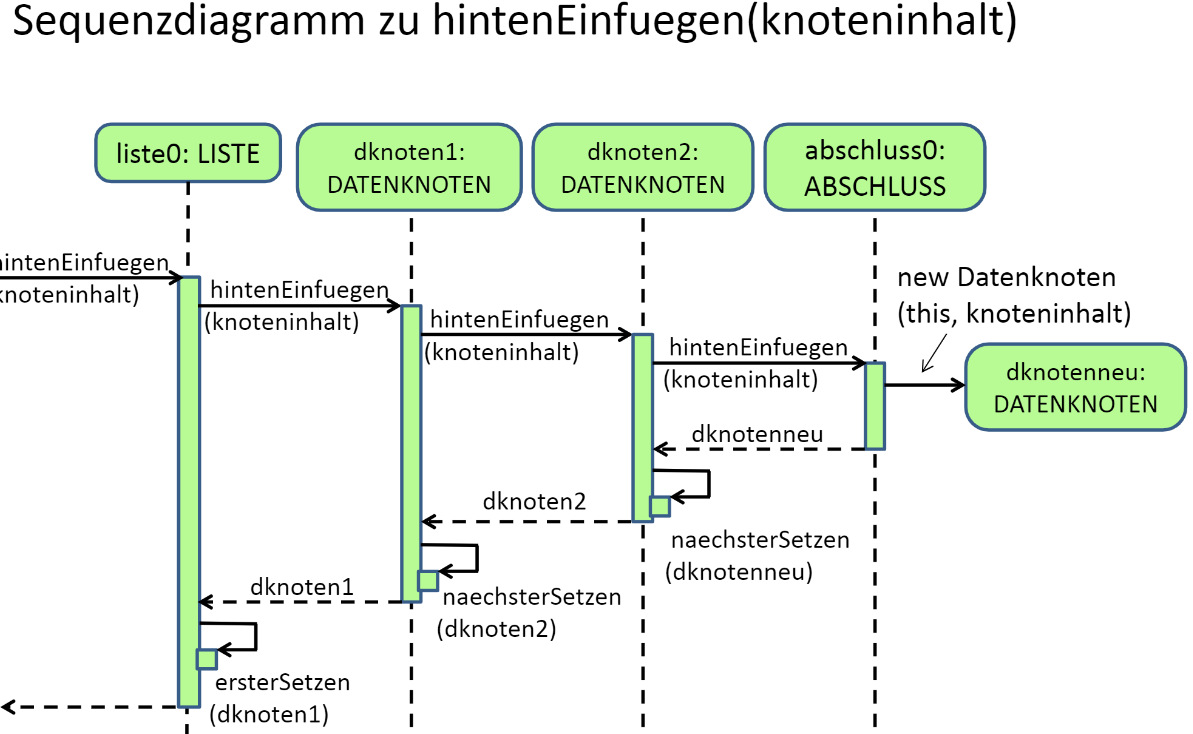


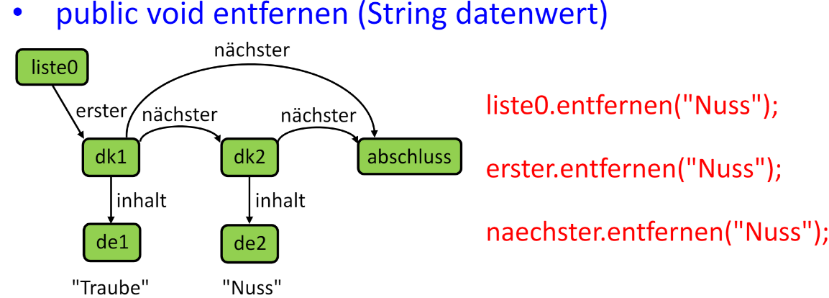
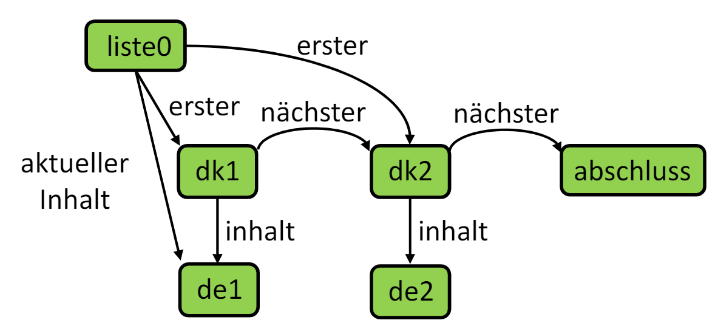
1. **Rekursive Liste:**

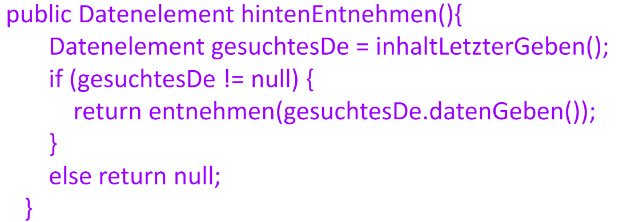


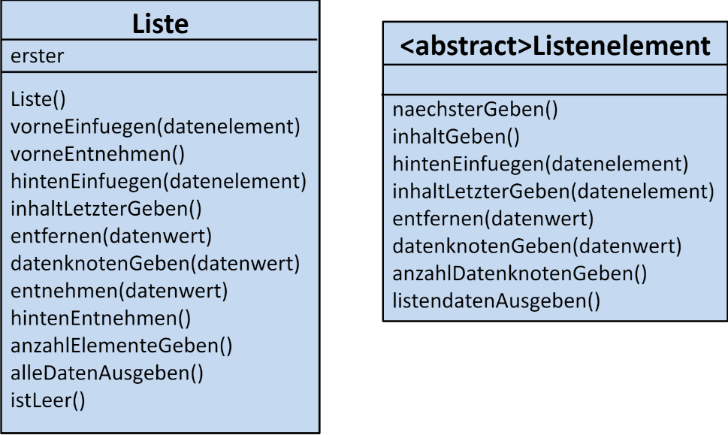
Alle Knoten bis auf den letzten verhalten sich gleich

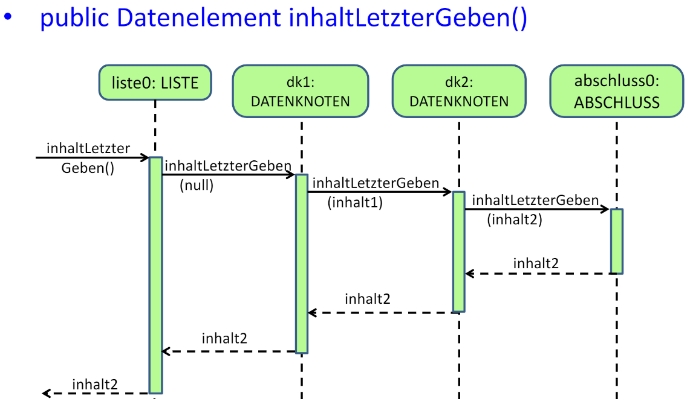
* Knoten reichen Inhalte meist nur an den nächsten weiter
* Arbeit macht oft der Abschluss



public Datenelement vorneEntnehmen()



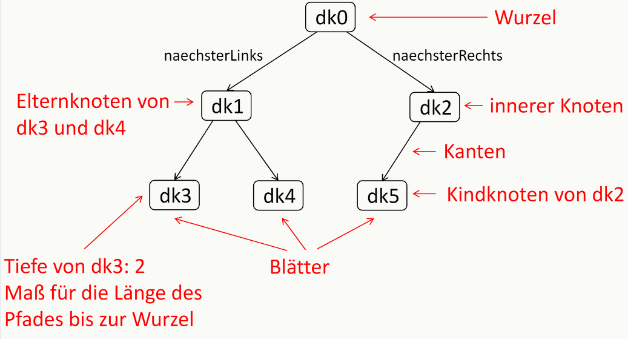




1. **Binärbäume:**

Eine Baumstruktur entsteht wenn jedes Objekt mehrere Nachfolger hat

* Genau ein Objekt wird innerhalb der Baumstruktur **nicht** referenziert (Wurzel)
* Von der Wurzel kann jedes Objekt über genau einen Weg erreicht werden

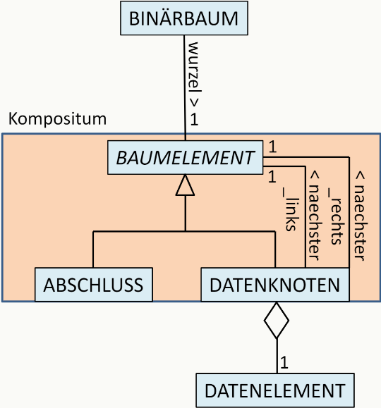
Bei Binärbäumen: Jedes Objekt hat **höchstens zwei** Nachfolger

Traversieren: Jeder Knoten muss 3 Aktionen ausführen:  
 sich selbst ausgeben (w), weiterreichen an den linken Teilbaum (LK) und weiterreichen an den rechten Teilbaum (RK)

Preorder: w-LK-RK Blätter sind ein Abschluss oder

Inorder: LK-w-RK haben zwei null Referenzen als

Postorder: LK-RK-w Nachfolger

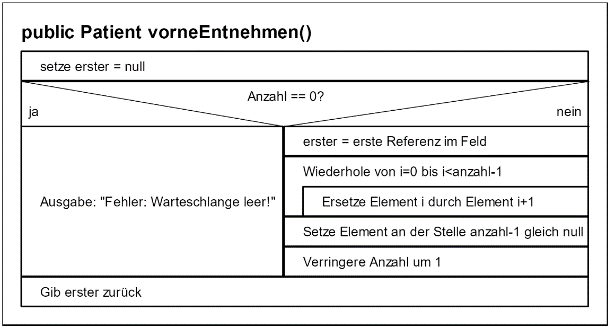
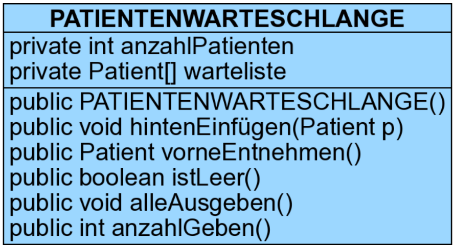


Kompositum ähnlich wie   
  
bei Liste

## HJ1\_1 Die Datenstrukturen „Warteschlange“ und „Stapel“

1. **Warteschlangen (queue)**

zb in Rechnernetzen die viele Anfragen bekommen

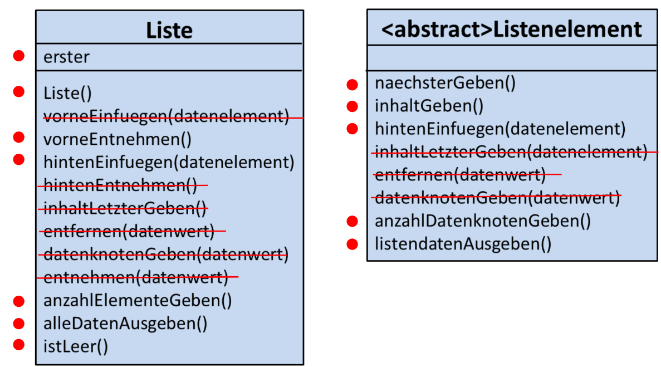
Grundprinzipien: FIFO (first in first out) -> Elemente in der Reihenfolge entnehmen, wie sie eingefügt wurden

Umsetzung zb mit einem Array:

* Problem:

Feldlänge statisch  
Nachrutschen bei entnehmen

* Lösung: Dynamische Datenstrukturen wie Listen:

Iterative Lösung mit einfach verketteter Liste aber besser: **Rekursive Liste**

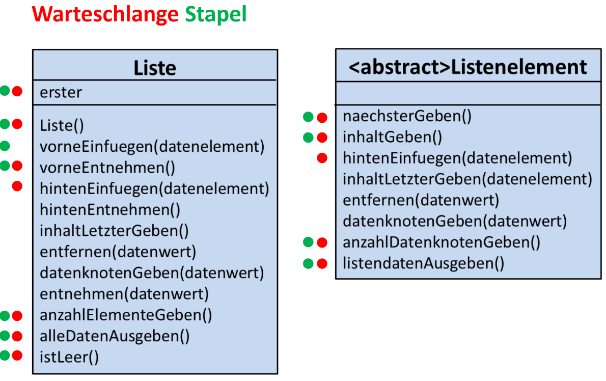
-> benötigte Einfügemethoden: hintenEinfügen, vorneEntnehmen **oder** hintenEntnehmen, vorneEinfügen

-> vorneEntnehmen & hintenEinfügen bei rekursiver Liste aber einfacher

1. **Stapel (stack)**

zb bei Solitaire; nur die oberste, zuletzt hingelegte Karte ist entnehmbar

Grundprinzipien: LIFO (last in first out)

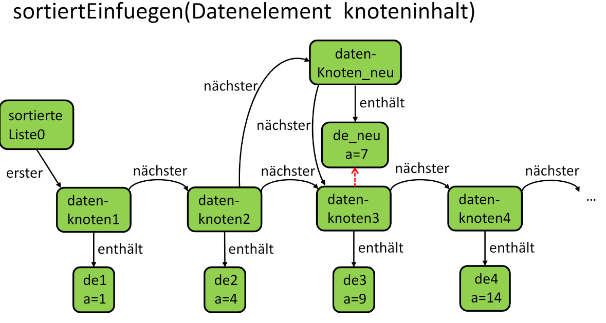
-> benötigte Einfügemethoden: vorneEinfügen, vorneEntnehmen **oder** hintenEinfügen, hintenEntnehmen

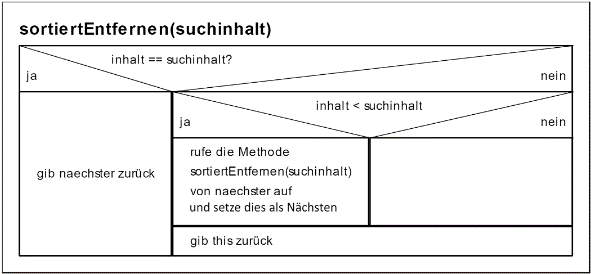
-> vorneEntnehmen & vorneEinfügen bei rekursiver Liste aber einfacher

## HJ1\_2 Die sortierte Liste

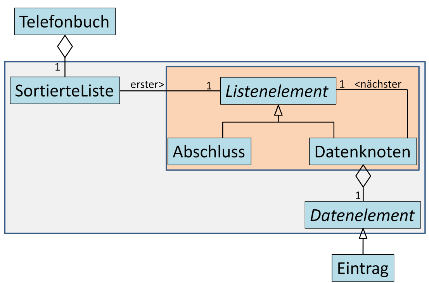
Die Datenwerte einer Liste können sortiert sein (zb alphabetisch, numerisch…)

Vergleich von Werten mit zb <>, mit eigen implementierten Methoden, oder bei Strings String.compareTo(string) -> (alphabetische Ordnung)

Beim Einfügen wird der Inhalt immer weitergegeben  
bis der Inhalt kleiner ist als der in der Liste.  
Hier wird dann eingefügt und der neue Datenknoten zurückgegeben.  
Wenn der neue Inhalt größer als alle anderen ist, fügt der Abschluss ein.

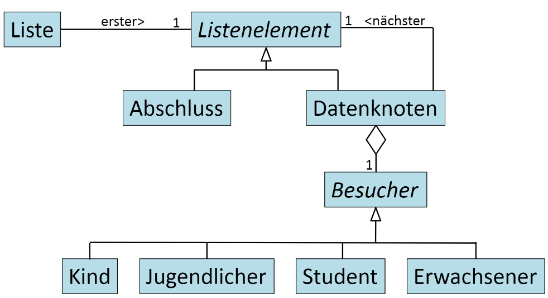


Beim Entfernen muss man nur so weit suchen, solange der Inhalt der Liste kleiner bleibt als der gesuchte Inhalt.   
Wenn der Inhalt größer ist als der Suchinhalt, kann der Suchinhalt aufgrund der Sortierung nicht mehr kommen



## HJ1\_3 Heterogene Listen

Liste mit mehreren verschiedenen Klassen als Datenelement -> dafür braucht man eine abstrakte Oberklasse aller Datenelemente, um alle Datenelement Klassen ansprechen zu können



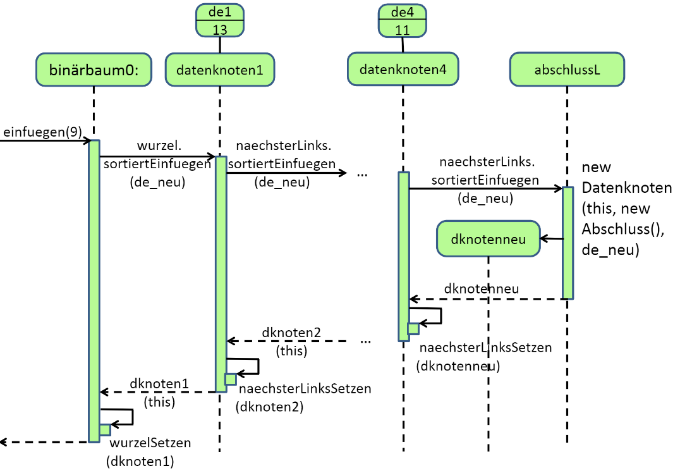
## HJ1\_4 Der geordnete Binärbaum

= binäre Suchbäume = BST (binary search tree)

Ein Binärbaum heißt geordnet, wenn der Schlüssel des LK kleiner und der Schlüssel des RK größer ist als die Wurzel

Einfügemethoden etc. sehr ähnlich zu Liste. Man muss nur den richtigen Teil-Ast unterscheiden:

public void sortiertEinfuegen(int i):



Datenknoten entfernen: (Pseudocode da es sehr kompliziert ist)

Suche im Baum nach dem Datenknoten  
 Wenn kein DK gefunden wurde: return  
 Wenn DK gefunden: ermittle Anzahl der Kindknoten  
 Anzahl 0: DK durch einen Abschluss ersetzen  
 Anzahl 1: DK durch den Kindknoten ersetzen  
 Anzahl 2: DK-Inhalt durch den kleinsten Inhalt im rechten Teil-Ast ersetzen; DK mit  
 dem kleinsten Inhalt im rechten Teil-Ast entfernen

# HJ 2

Gestrichen rip

# HJ 3

## HJ3\_1 Formale Sprachen (Referatsthema)

1. **Theorie**

Natürliche Sprache = zb Deutsch, Englisch…

Formale Sprache = zb Mathe, Chemieformeln, Programmiersprachen…

Alphabet ∑ := beliebige, endliche, nichtleere Menge, deren Elemente Zeichen oder Symbole genannt werden

Zeichenkette, Wort := endliche Folge von Zeichen des Alphabets

Leeres Wort Ɛ [„epsilon“] := Zeichenkette mit null Zeichen

Wortmenge ∑\* := Menge aller Wörter, die sich aus ∑ bilden lassen

∑ = {a,b}

∑\* = {Ɛ, a, b, aa, ab, ba, bb, aaa, …}

Teilmenge L von ∑\* := formale Sprache über dem Alphabet ∑

Wörter:

‚2+3=5‘ syntaktisch korrekt  
‚=+1-‘ syntaktisch falsch  
‚2+3=6‘ syntaktisch korrekt, aber semantisch (von der Bedeutung her) nicht sinnvoll

1. **Erzeugung**

S.13 Wetterkarte: ∑ = { ☀; ❄; 🌨; 1; 2; 3; 4; 5; 6; 7; 8; 9; 0; -; °C, %, /}

Produktionsregeln:

<S> -> <W> <T> ‘/‘ <L>  
<W> -> <WS> | <WS> <WS> | <WS> <WS> <WS>

<WS> -> ‚☀‘ | ‚❄‘ | ‚🌨‘  
<T> -> ‚-‘ <Z> ‚°C‘ | <Z> ‚°C‘ | ‚0’ ‚°C‘

<Z> -> <ziffernichtnull> | <ziffernichtnull> <ziffer>

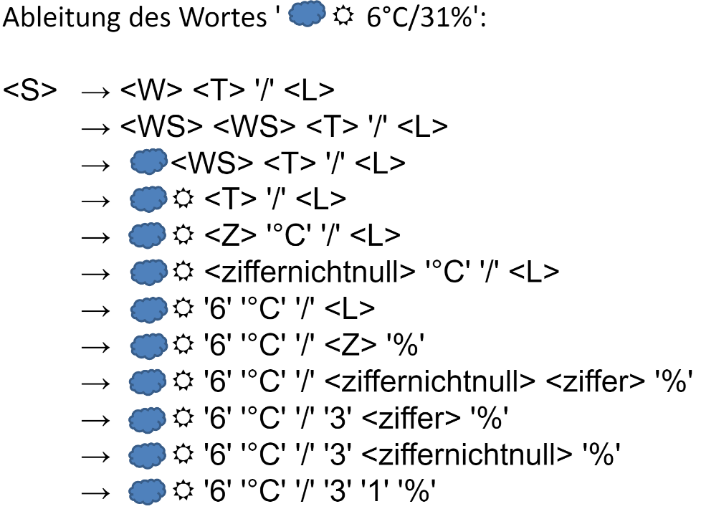
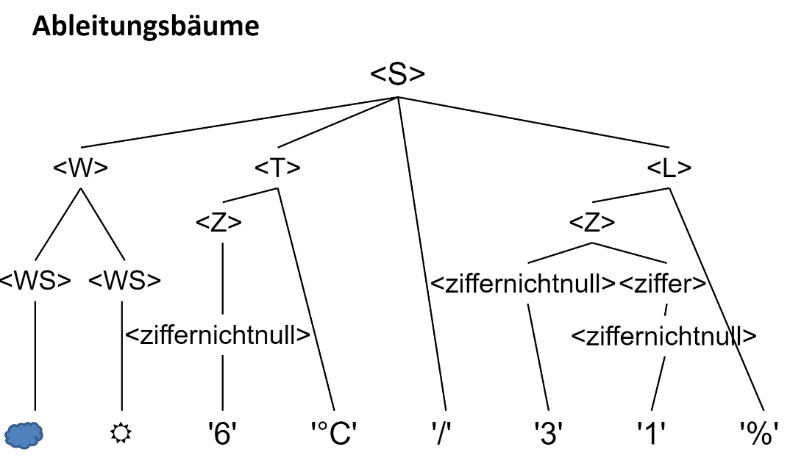
<ziffernichtnull> -> ‚1‘ | ‚2‘ | … | ‚9‘

<ziffer> -> <ziffernichtnull> | ‚0‘

<L> -> <Z> ‚%‘ | ‚0‘ ‚%‘

**Syntaktische Variablen** stehen in spitzen Klammern (zb <W>). Sie werden in der Ableitung eines Wortes ersetzt und heißen **Nichtterminale**. Sie gehören nicht zu den Symbolen der Sprache.

Alle Symbole, die zu ∑ gehören, werden nicht ersetzt (**Terminale**). Diese werden in Hochkommata gefasst (zb ‚1‘).

1. **Ableitungen**

Startvariable (zb <S>): syntaktische Variable, mit der alle Ableitungen einer Sprache beginnen

Wenn ein Wort, nur durch anwenden der Produktionsregeln (ausgehend von der Startvariable) erzeugt werden kann, gehört es zur Sprache, sonst nicht.

<A> -> ‚0‘ | ‚0‘ <A> : rekursive Produktion, theoretisch unendlich lange wörter

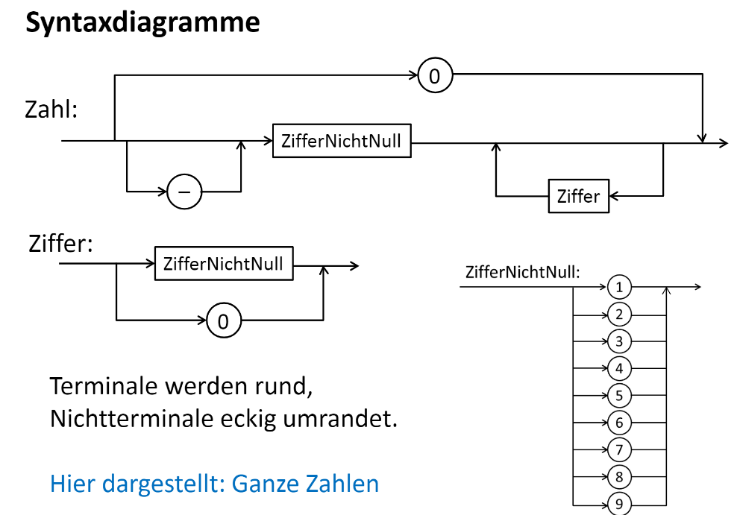
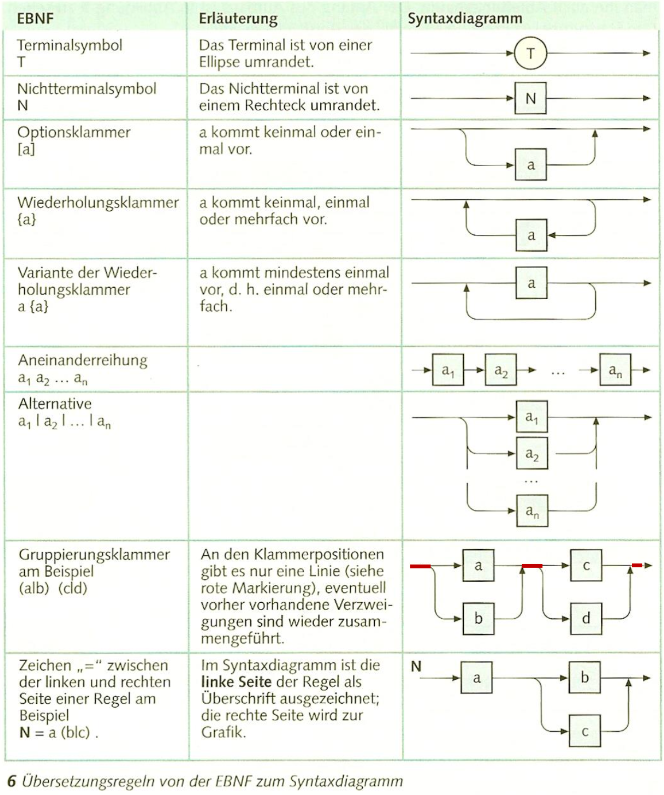
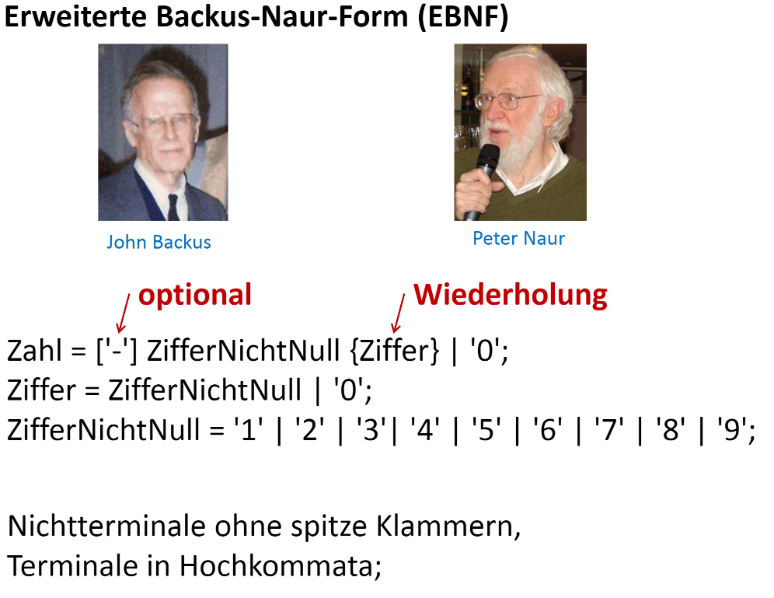
<A> -> b<B> | b verschränkte Rekursion

<B> -> a<A> | a

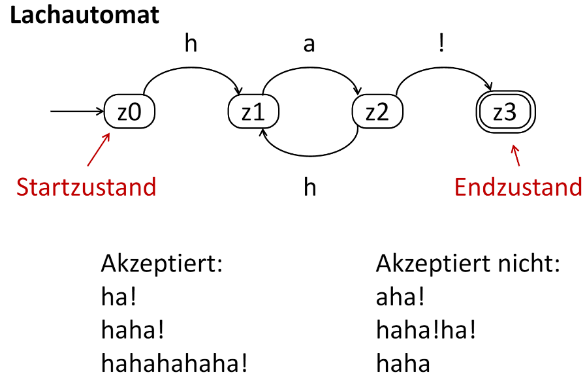
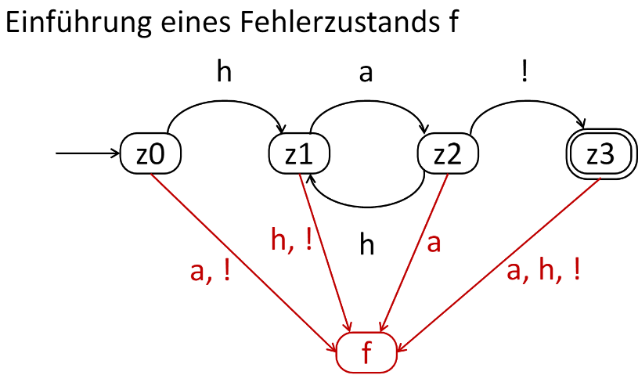
1. **Formale Grammatik**

Grammatik G = (V, ∑, P, S) <- Muss immer mit angegeben werden [sprich „VEPS“]

V := Menger der syntaktischen Variablen  
 ∑ := Alphabet  
 P: Menge der Produktionsregeln  
 S: Startvariable  
Dabei gilt: V ∩ ∑ = {} (V darf keine Nichtterminale enthalten, die Zeichen in ∑ sind)

1. **Notationsformen**

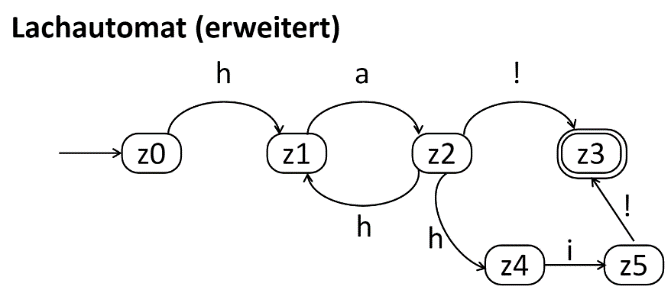
**Bei EBNF: Semikolon am Ende nicht vergessen**

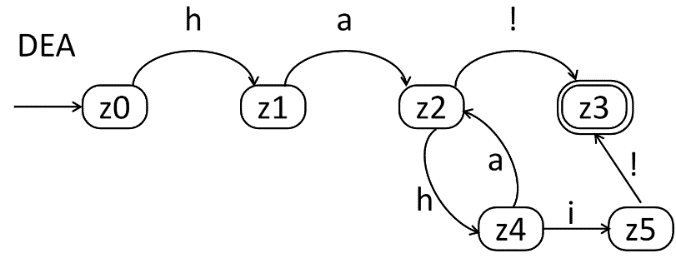
1. **Erkennende Automaten**

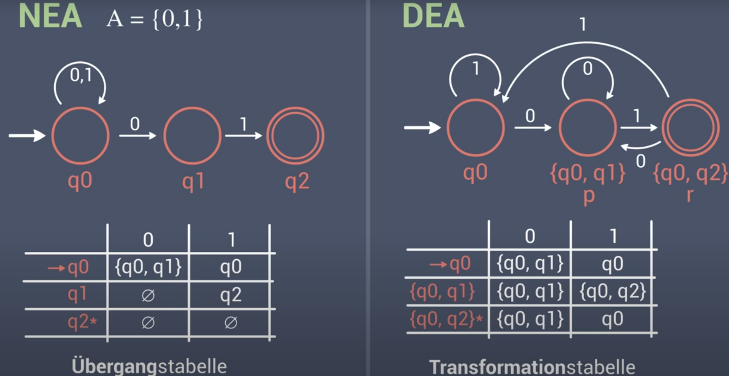
Ein Wort wird akzeptiert, wenn der Automat (beginnend im Startzustand) nach Abarbeitung des Wortes, in einem Endzustand ist.

Erkennender Automat, Akzeptor := Automat, der ein Wort auf   
 Korrektheit überprüft

Endlicher Automat := Automat mit endlich vielen Zuständen  
 und Verarbeitung einer endlich langen  
 Zeichenkette

Wenn es mindestens einen Zustand gibt (hier z2), von dem aus ein Eingabezeichen eine Transition zu verschiedenen Folgezuständen auslösen kann, ist der Automat **nichtdeterministisch (NEA)**

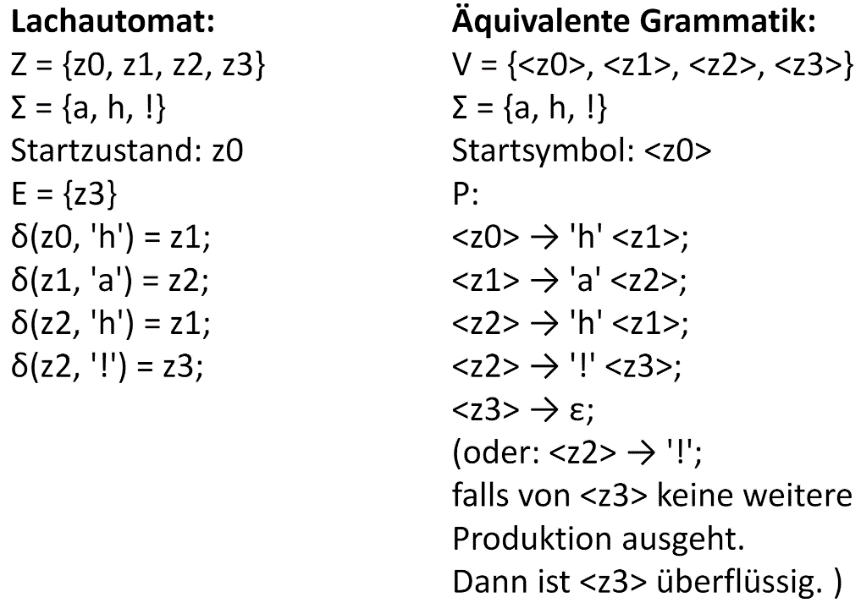
Wir implementieren aber nur DEA (deterministische, endliche Automaten)

Zu jedem NEA gibt es einen DEA, der die gleiche Sprache akzeptiert.

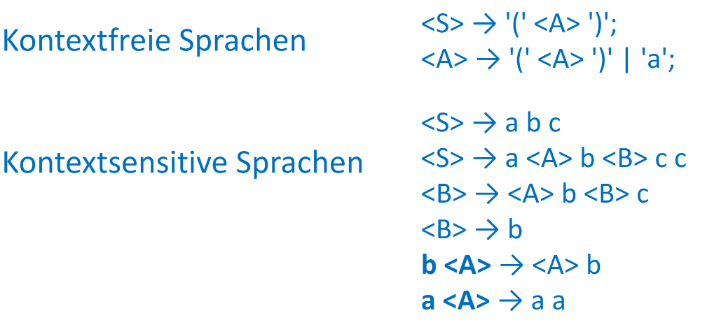
Definition DEA (**D**eterministischer **E**ndlicher **A**utomat)

1. Endliche Menge Z von Zuständen
2. Endliches Eingabealphabet ∑
3. Ein Startzustand
4. Menge E von Endzuständen
5. Zweistellige Übergangsfkt δ, die jedem Paar aus Zuständen und Eingabezeichen einen Folgezustand zuordnet

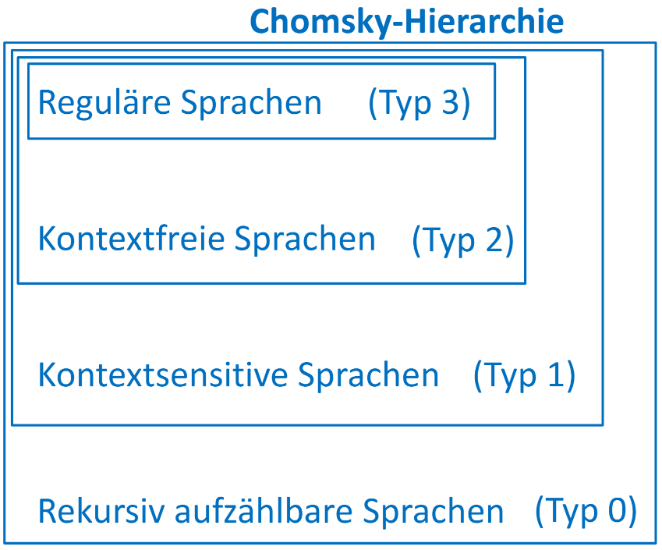
https://youtu.be/ArYI4dd7ffE



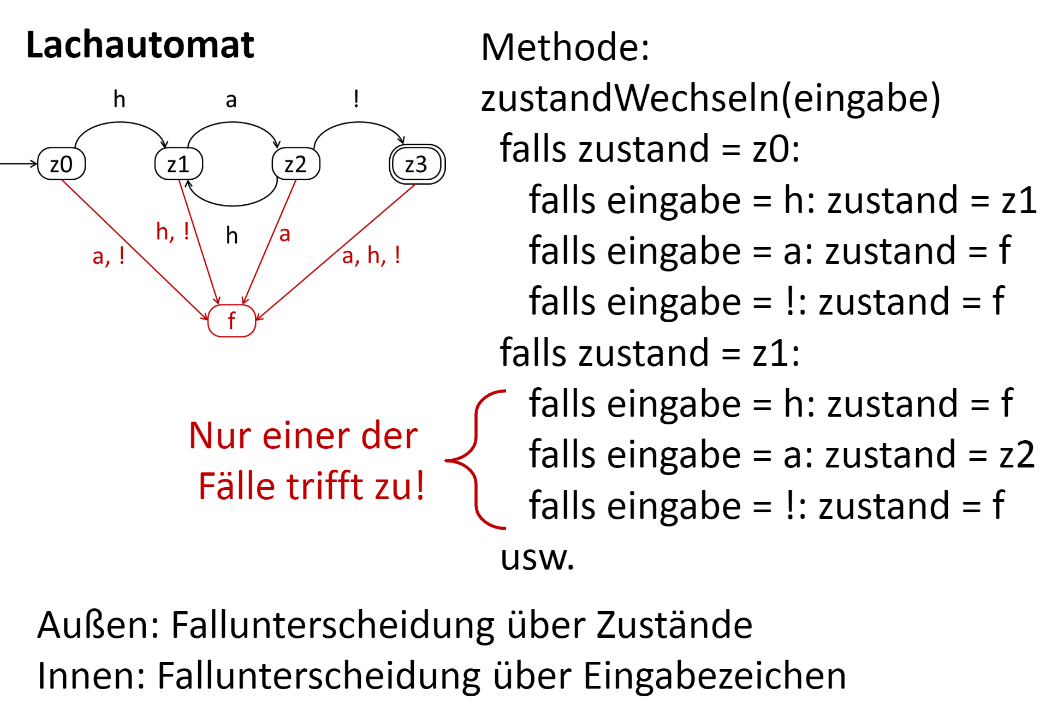
* Rechtslineare Grammatik

Reguläre Sprache := Jede Sprache, die durch einen endlichen Automaten erkannt werden kann

Zb Rekursive Produktionen können nicht erkannt werden, da sie potenziell unendlich lange Wörter bilden können



1. **Implementierung von endlichen Automaten**



* switch-Statement für Außen; if-Statements für Innen

Für implementierung wichtig: String funktionen String foo = „bar“;

foo.length() -> Länge des Strings

foo.charAt(int index) -> Buchstabe and position <index>

# HJ 4

## HJ4\_1 Aufbau und Funktionsweise eines Rechners, Zustandsübergänge der Registermaschine

1. **Aufbau eines Computersystems**

Hauptkomponenten (Zentraleinheit): CPU (Central Processing Unit)  
 RAM (Random Access Memory)  
Peripheriegeräte: Eingabegeräte: Tastatur, Maus…  
 Ausgabegeräte: Monitor, Drucker…  
 Datenspeichergeräte: Festplatte, CD, DVD, USB

Kommunikationsgeräte: Netzwerkkarte, Wlan-Karte

1. **Von-Neumann-Rechner** (John von Neumann (1903-1957))

Programme & Daten im selben Speicher -> Universalrechner: Von-Neumann-Rechner

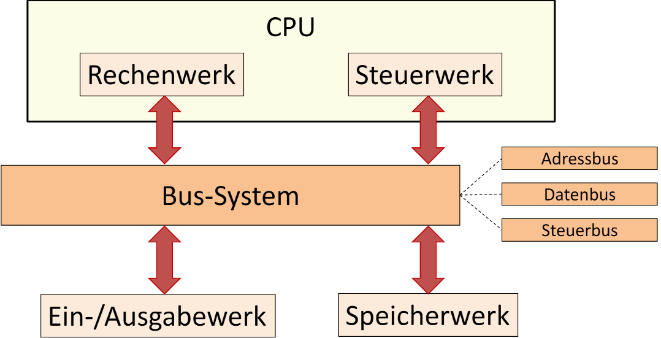
5 Grundeinheiten:  
- Rechenwerk (Arithmetic Logic Unit (ALU)) -> Rechnet

- Steuerwerk (Leitwerk, Control Unit) -> Steuert Ablauf von Befehlen

- Speicherwerk (RAM, Hauptspeicher) -> Speichert Programme & Daten

- Eingabewerk -> Steuert Eingabe von Daten & Programmen in den Speicher

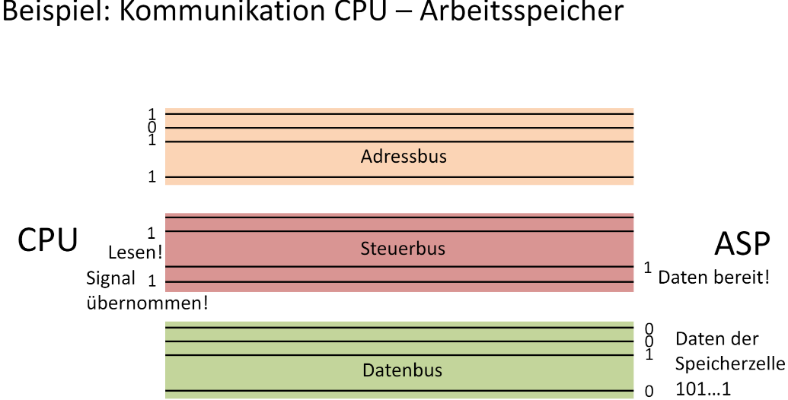
- Ausgabewerk -> Steuert Ausgabe von Daten nach außen



Bus-System:

-paralleler Bus := viele Einzelleitungen (zb Adressbus)

- serieller Bus := aufeinanderfolgende Signale (zb USB)

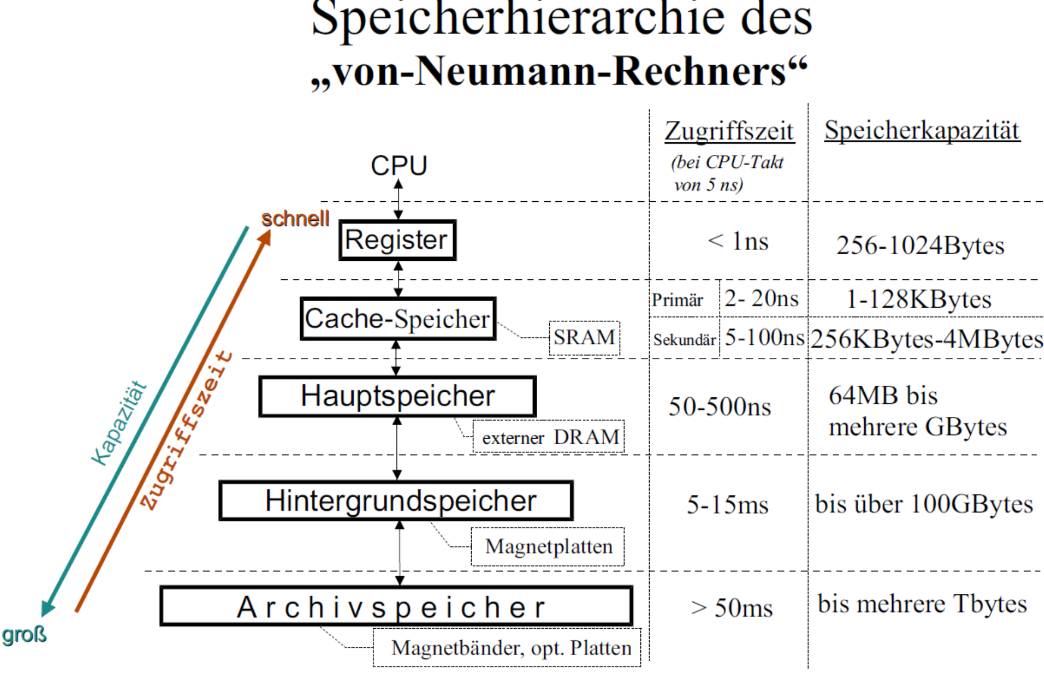


1. **Probleme des Von-Neumann-Rechner**

Früher: Rechen- & Steuerwerk langsam

Heute aber: Schnellere Prozessoren -> Verbindungssystem ist zu langsam (Von Neumannscher Flaschenhals)

* Lösung des Problems: schneller zugreifbarer Speicher (Cache)  
   Parallelverarbeitung

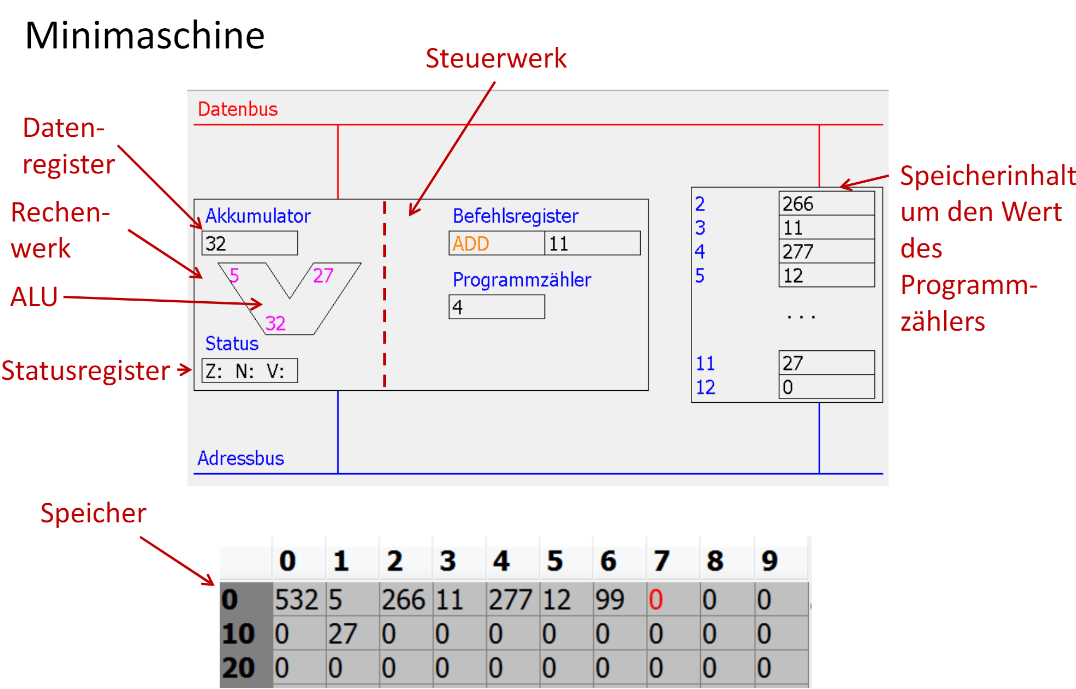


<- nur grob wissen

1. **Prinzipien des Von-Neumann-Rechners**
2. Der Rechner besteht aus den 5 Grundeinheiten (siehe oben)
3. Hauptspeicher ist in gleich große Zellen unterteilt, die fortlaufend nummeriert sind
4. Programme & Daten binär codiert im selben Speicher
5. Aufeinanderfolgende Befehle in aufeinanderfolgenden Speicherzellen und nacheinander abgearbeitet
6. Sprungbefehle zur abweichen von der linearen Verarbeitung
7. Operationen: Arithmetische Befehle, logische Befehle, Sprungbefehle, Transportbefehle (LOAD, STORE)

+-> AB Binärzahlen <- Binär rechnen kommt definitiv dran; ganze zahlen auch; floats nicht; addieren

1. **Die Registermaschine**



- Befehlsregister erhält momentan auszuführenden Befehl

- Befehlszähler hat die Speicheradresse des nächsten Befehls

- Datenregister := Speichereinheiten in CPU, schneller als RAM

- Minimaschine = 1-Registermaschine: Ein Operand aus Datenregister, einer aus RAM. Ergebnis im Datenregister (Akkumulator)

- Statusregister (Flags): Z=zero, N=negativ, V=overflow

- Maschinenbefehl: zb. LOAD 5

LOAD := Operationskennung (Opcode)

5 := Operandenteil, Adressteil

Befehlszyklus:

- Fetch-Phase I: Befehl holen (Adresse im Befehlszähler), Befehlszähler += 1

- Decode-Phase: Opcode des Befehls bestimmen

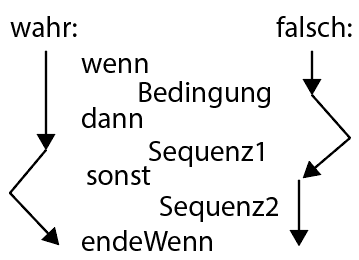
- Fetch-Phase II: Operand (-en) laden; BZ um die Anzahl der gelesenen Speicherzählen erhöhen

- Execute-Phase: Operanden für ALU bereitstellen; Befehl ausführen; Ergebnis zurückschreiben; bei Sprüngen evtl. BZ aktualisieren

* Frage: Warum 2 Fetchphasen -> möglicherweiße 2 Operanden abhängig vom Befehl (nicht in Minimaschine)

## HJ4\_2 Programmierung auf Maschinenebene

Befehlssatz: <https://schule.awiedemann.de/manualmini/assembler.html>

1. **Algorithmische Strukturelemente**
2. If/else bsp: if i < 10: i = -3 else: i=10

LOAD i

CMP i i <10

JGE else

<- Unbedingter Sprung

LOADI -3 i = -3

STORE i

JMP end # else teil überspringen

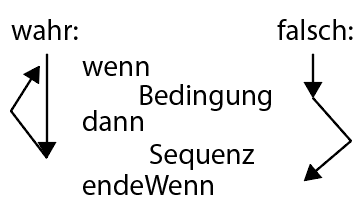
else:

LOADI 10 i = 10

STORE i

end: HOLD

1. While: bsp: while i >= 10: i—

start: LOAD i

CMPI 10 wenn i < 10: springen  
 JLT end

SUBI 1 i--

STORE i

JMP start

end: HOLD

1. For: bsp: for(int i = 0; i < 69; ++i): n\*=i

startFor: LOAD i

CMPI 69 Springen wenn i >= 69

JGE end

MUL n n \*= i

STORE n

LOAD i

ADDI 1 ++i

STORE i

JMP startFor

end: HOLD

1. Do-while: bsp: do: i++ while: i != 24

startDo: LOAD i

ADDI 1 i++

STORE i

CMPI 24 Springen wenn i nicht 24 ist

JNE startDo

HOLD

1. **Adressierungsarten**

- direkte Adressierung (ADD 100) -> Addieren von Wert in Zelle 100

- unmittelbare Adressierung (ADDI 100) -> Addieren von Wert 100

- Registeradressierung (R12)

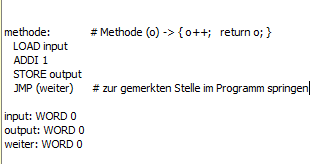
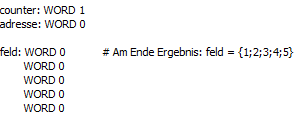
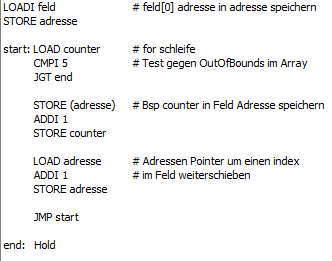
- Indirekte Adressierung (ADD (100)) -> Addieren des Wertes in der von Zelle 100 angegebenen Zelle

Zelle 100: 75

Zelle 75: 3

ADD (100) -> Addieren von 3 (Wert in Zelle 75)

Methoden, Felder nur grob wenn überhaupt:

1. **Methoden**
2. **Felder**

## HJ4\_3 Grenzen der Berechenbarkeit

1. **Laufzeit von Algorithmen**

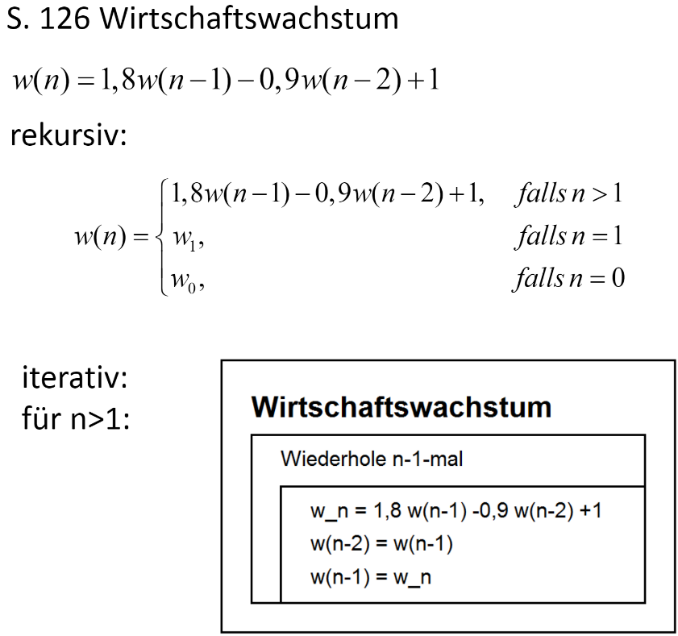
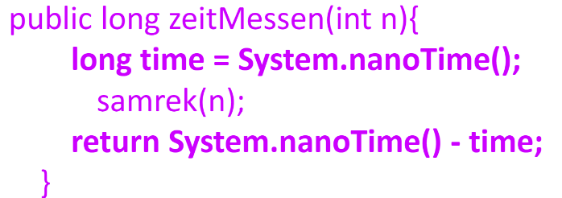
O(1) < O(log(n)) < O(n) < O(n\*log(n)) < O(n²) < O(exp(n))

[Flops/s] = Floating Point Operations / second

Schnelle Rechner: Leibniz-Rechenzentrum:

SuperMUC: 3 PFlops/s, ca. 150.000 Prozessorkerne, 300 TByte Hauptspeicher

SuperMUC NG: (schnellster Rechner in Dt) 26.7 PFlops/s, >300.000 Kerne, 700 TByte Hauptspeicher, 70 PByte Plattenspeicher

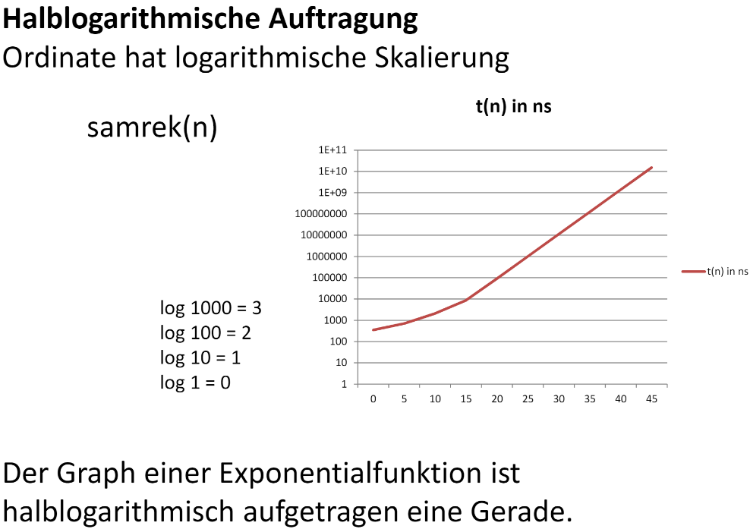
1. **Zeitmessung:**

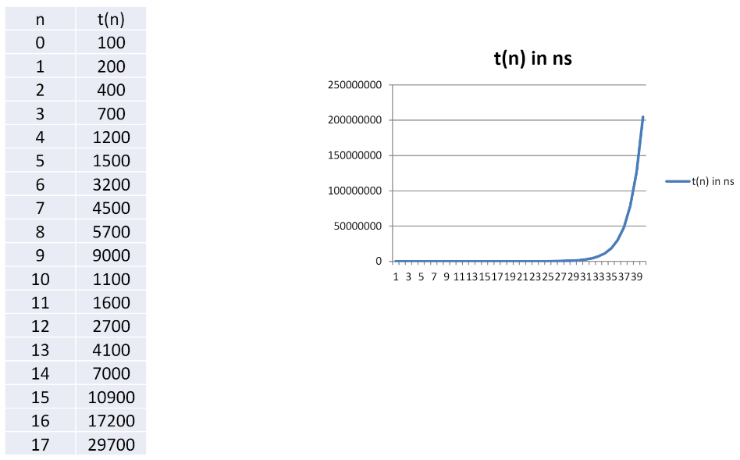
Fehlerquellen:

- Zeitmessung erfordert auch Zeit

- Prozess kann während Ausführung zb durch Garbage Collection unterbrochen werden

-> mehrmals messen und Minimum nehmen



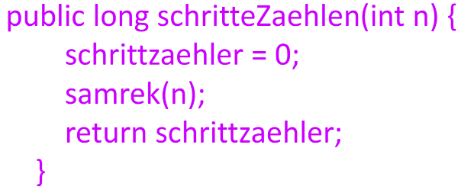
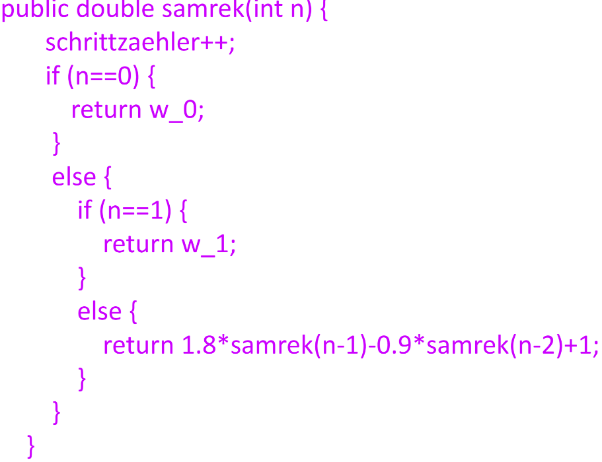


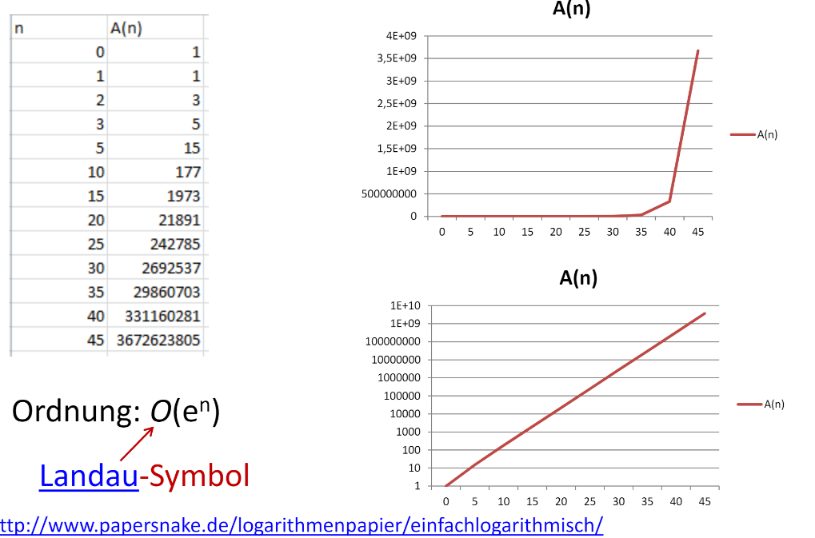
1. **Schritte zählen**

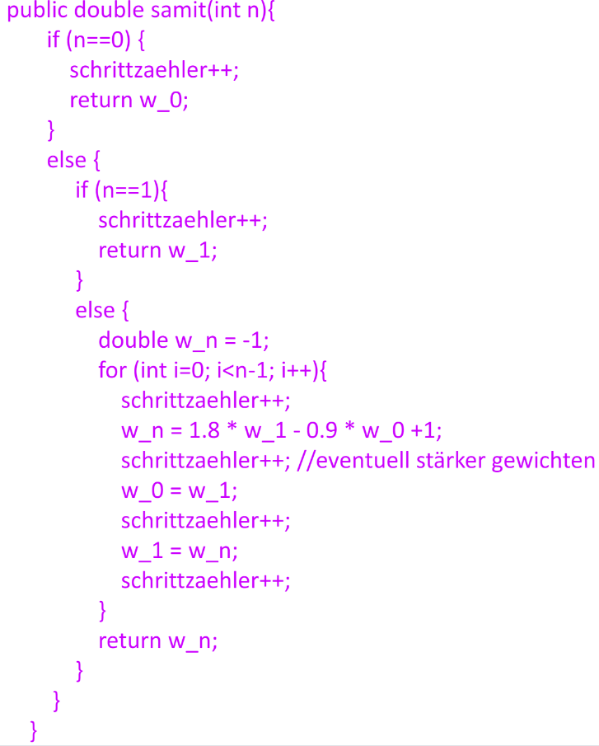
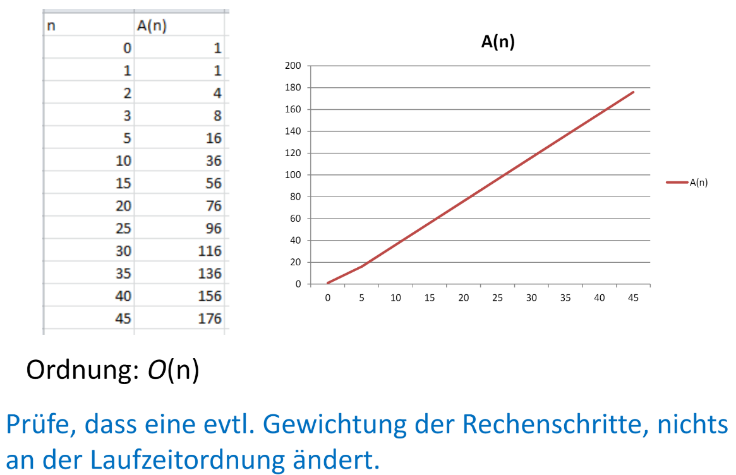
Nach jeder Anweisung wird ein schrittzähler um eins (evtl auch gewichtet) erhöht

- Sagt nichts über den absoluten Zeitverbrauch

- unabhängig von Rechnern und Unterbrechungen durch andere Prozesse

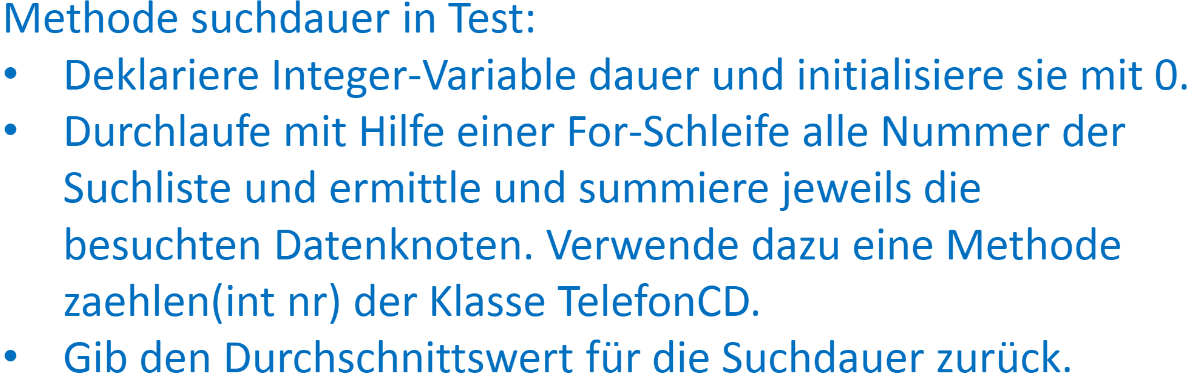


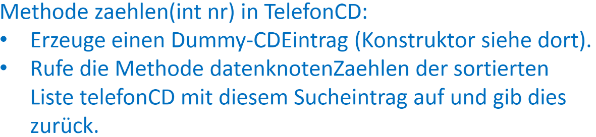
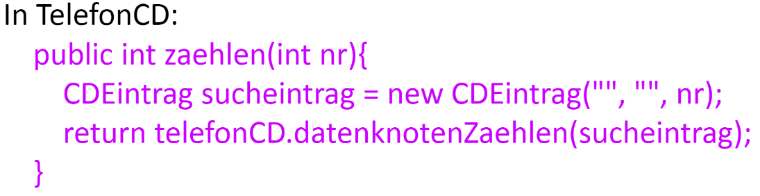


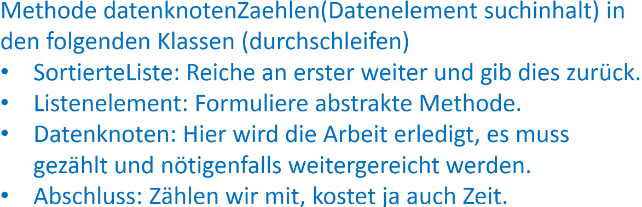
Iterativ:

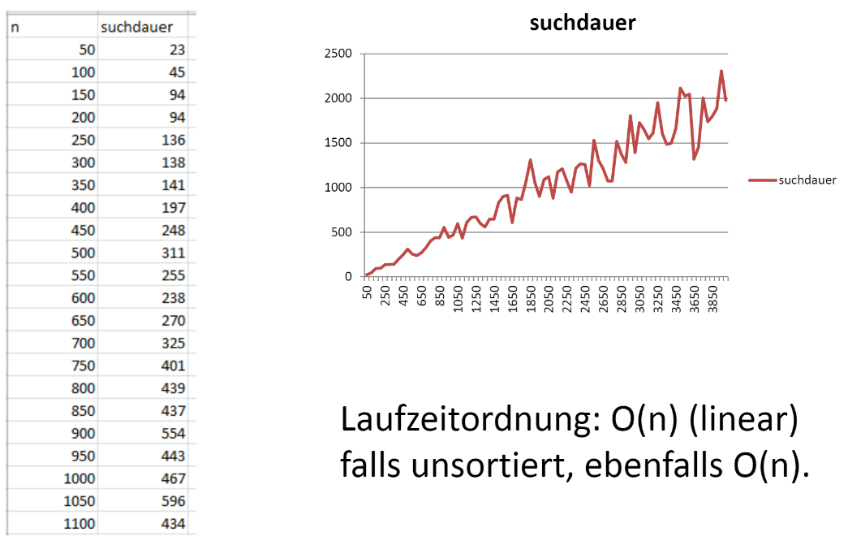
Vllt auch eigene Methode mit selben Aufbau, die nur schritte zählt um große Zahlen (& Overlows) zu vermeiden

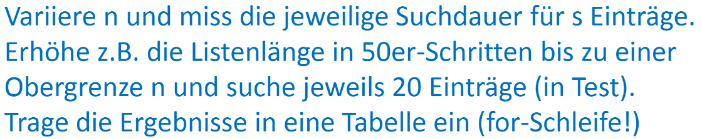
1. **Suchdauer in sortierten Listen**

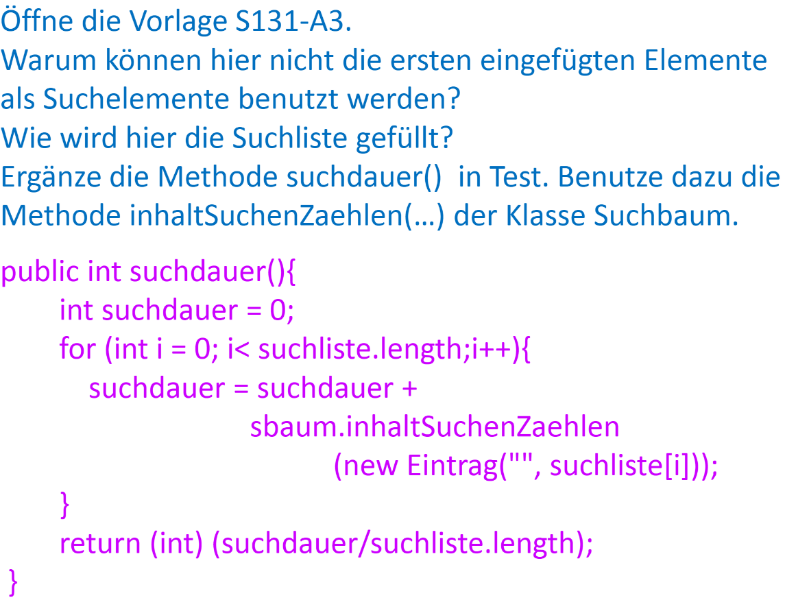
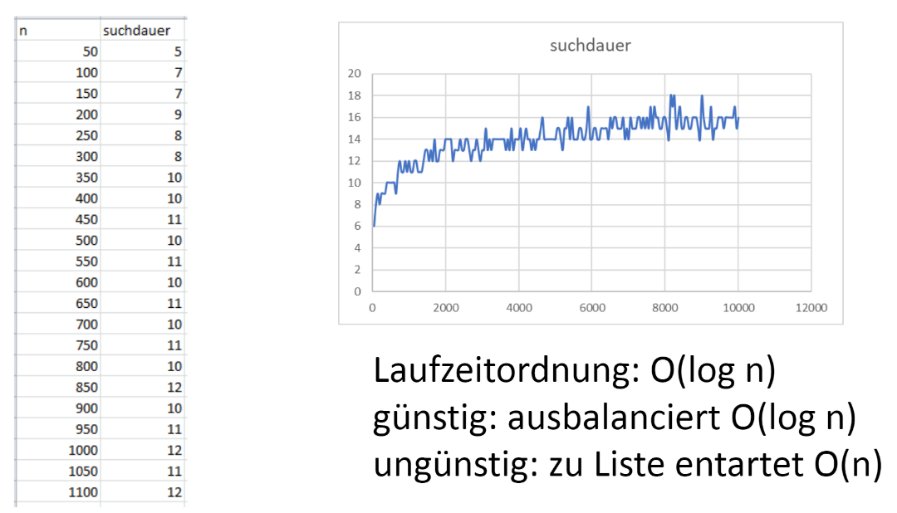
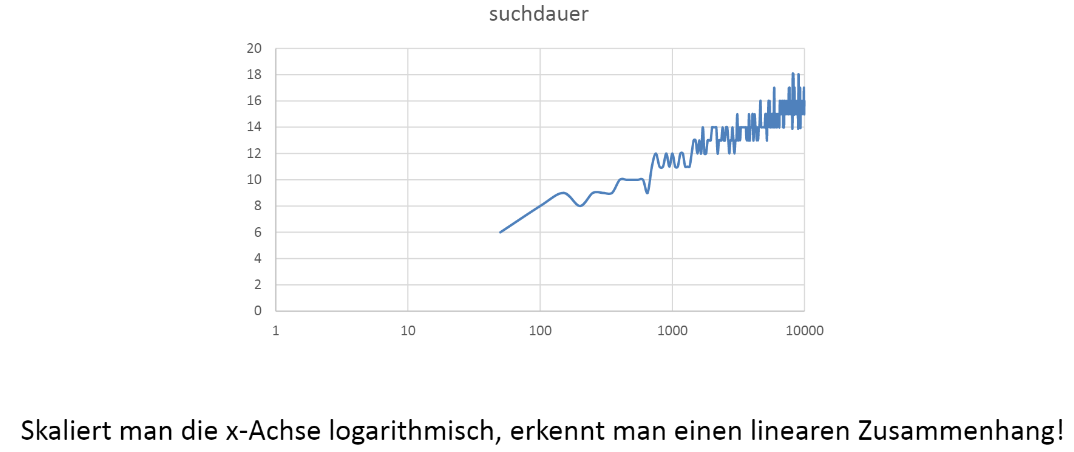


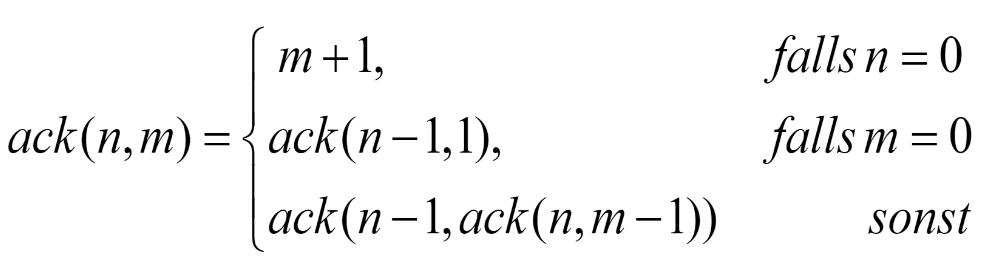
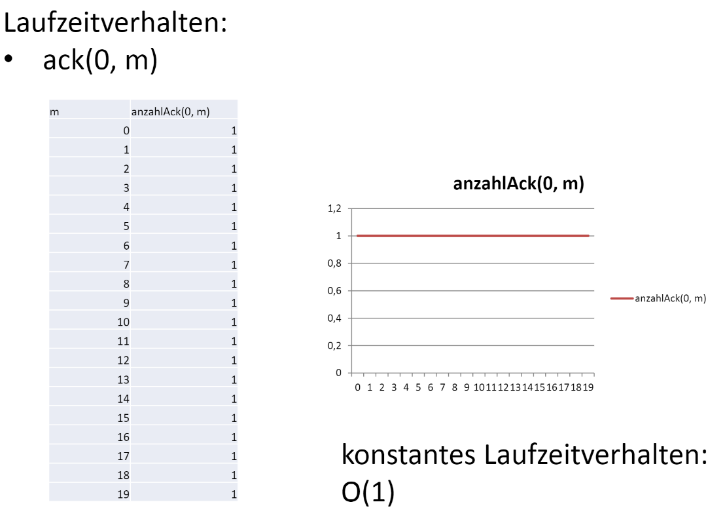


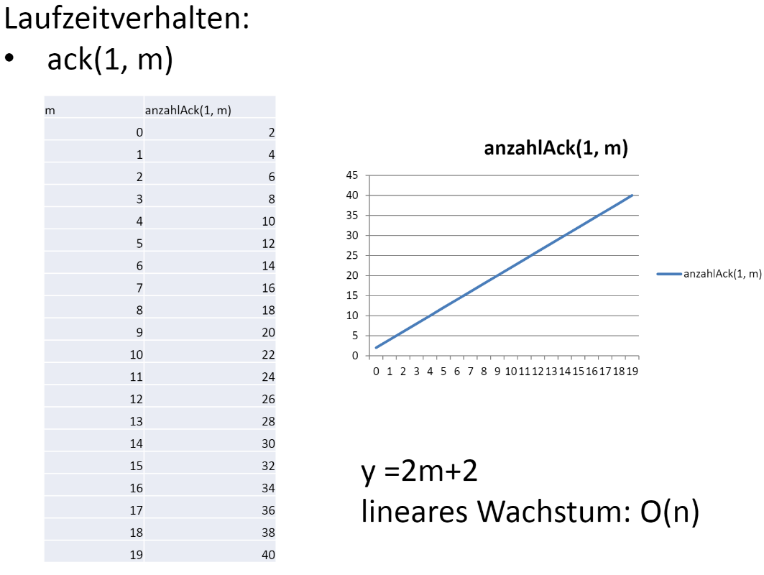


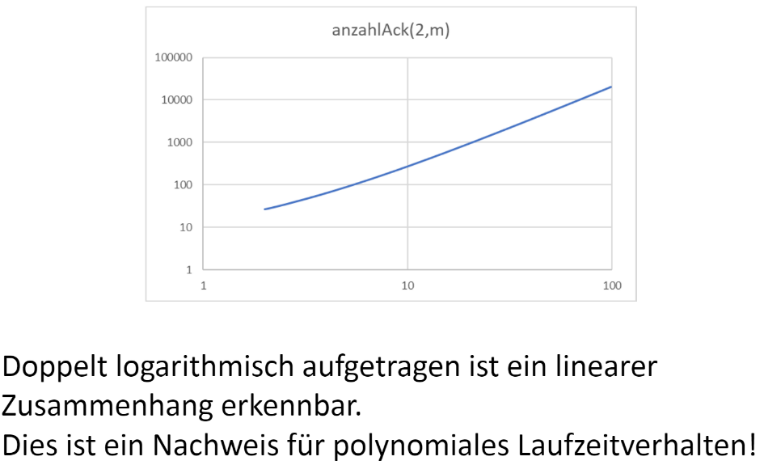
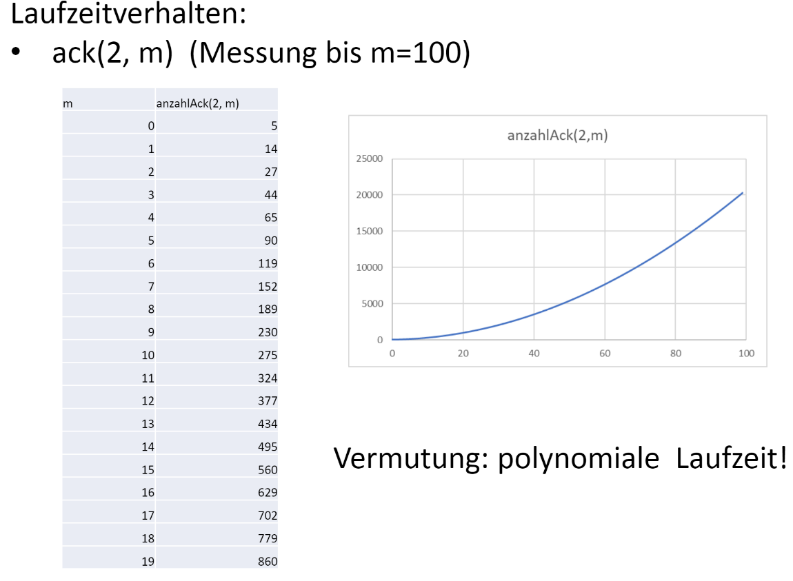
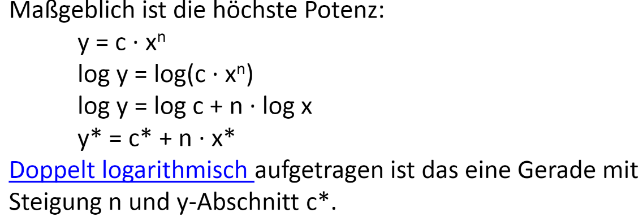


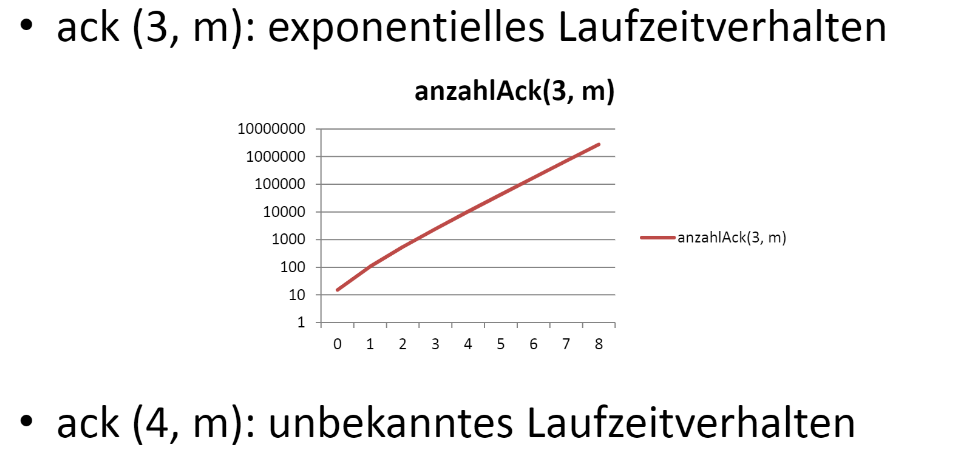


1. **Suchdauer im Binärbaum**
2. **Ackermann Funktion**

 <- Funktion nicht auswendig







Sortier algorithmen durchlesen

Laufzeit der sortier algorithmen nicht

Laufzeit graphen nur ganz grob eher nicht

Halte problem garnicht

Brute force mögl. Berechnen wichtig

Verschlüsselung