

Programmierung II – Algorithmen & Datenstrukturen

Prof. Dr. Andreas Schilling

www.ravensburg.dhbw.de

1. Grundlagen

Agenda

Algorithmen

Was ist ein Algorithmus und wie lässt sich die Güte dieser bewerten?

Datenstrukturen

Was sind zentrale Datenstrukturen und wie unterscheiden sie sich.

Laufzeitanalyse Datenstrukturen

Wie unterscheiden sich Datenstrukturen im Praxiseinsatz.

Algorithmen

Grundlagen

„Ein Algorithmus ist eine **endliche** Folge von Rechenschritten, die eine **Eingabe** in eine **Ausgabe** umwandelt.“
(Cormen et al. 2003)

- **Präzise Spezifikation** der Ein- und Ausgabe in Form von **Anzahl** und **Typ** der Parameter und Rückgabeformen.
- **Jeder Rechenschritt ist eindeutig** definiert und ausführbar.
- Die **Reihenfolge** der durchzuführenden Rechenschritte ist vorgegeben.
- Die **Spezifikation** von Ein- und Ausgabe, sowie der durchzuführenden Rechenschritte **ist endlich**.

„2 Eier in einer Schüssel aufschlagen. Mit Salz und Pfeffer würzen. Anschließend in einer Brandpfanne unter konstantem rühren und wenden anbraten.“

- Wieviel Salz und Pfeffer?
- Was ist aufschlagen?
- Wie lange konstant rühren?
- Welche Temperatur?
- Englischer Leser?

In diesem Kapitel befassen wir uns mit den Grundlagen von Algorithmen und Datenstrukturen. Fangen wir zunächst mit dem Begriff Algorithmus an. Gemäß Cormen ist ein Algorithmus eine endliche Folge von Rechenschritten, die eine Eingabe in eine Ausgabe umwandelt. Anhand dieser informalen Definition fällt Ihnen vielleicht schon auf, dass Algorithmen große Ähnlichkeit zu Anweisungen und Vorschriften besitzen, denen wir ganz natürlich in unserem Alltag folgen (bspw. Beipackzettel von Arzneimittel, EC-Automat, etc.). Nehmen wir als Beispiel das abgebildete Rezept für Rührei. Auch hier handelt es sich um Prozessschritte denen gefolgt wird, um eine Eingabe (Eier) in eine Ausgabe (Rührei) zu überführen. Allerdings fehlt es dem Rezept an der notwendigen Präzisierung, sodass auch fachfremde Kochnovizen den Prozessschritten folgen können. Wir können daher für Algorithmen in der Informatik festhalten, dass deren Anweisungen eindeutig spezifiziert werden müssen und der Typ und die Anzahl der Parameter unmissverständlich festgelegt sind. Ferner muss die Reihenfolge der einzelnen Rechenschritte klar spezifiziert werden und der Algorithmus muss enden.

Algorithmen

Eigenschaften

- **Determiniertheit:** Die gleiche Eingabe führt immer zur gleichen Ausgabe, allerdings sind andere Zwischenschritte möglich. *zwischenschritte irrelevant*
 - **Determinismus:** Die gleiche Eingabe führt immer zur gleichen Ausführung und zur gleichen Ausgabe. *auch die gleichen zwischenschritte*
 - **Terminierung:** Der Algorithmus endet für jede Eingabe.
 - **Partielle Korrektheit:** Der Algorithmus liefert bei einer gültigen Eingabe stets ein gültiges Ergebnis.
 - **Totale Korrektheit:** Der Algorithmus ist partiell korrekt und terminiert.
- 1 2 Eier von Hand öffnen und Inhalt in eine Schüssel geben.
 - 2 Mit Rührbesen Eier verquirlen bis sie leicht schaumig sind.
 - 3 Eier in Pfanne geben und auf mittlerer Hitze unter stetigem Rühren anbraten.
 - 4 Sobald die Eier eine goldbraune Farbe haben, vom Herd nehmen und servieren.
Algorithmus endet

Algorithmen werden in der Informatik basierend auf folgenden Eigenschaften klassifiziert:

- Die Determiniertheit sieht vor, dass ein Algorithmus bei gleicher Eingabe stets das gleiche Ergebnis zurückgibt. Bei der Durchführung der Prozessschritte sind allerdings abweichende Zwischenschritte erlaubt.
- Bei der Eigenschaft Determinismus wird erwartet, dass es bei gleicher Eingabe nicht nur stets zur gleichen Ausgabe kommt, sondern auch, dass hierbei stets die gleichen Zwischenschritte durchlaufen werden. Bitte beachten Sie, dass Determinismus stets Determiniertheit impliziert, d.h. wenn ein Algorithmus deterministisch ist, ist er zugleich auch determiniert. Dies gilt allerdings nicht in umgekehrter Richtung.
- Ein Algorithmus terminiert, wenn die durchgeführten Rechenschritte ein definiertes Ende haben.
- Die partielle Korrektheit liegt vor, wenn ein Algorithmus nach erfolgreichem Durchlaufen der Prozessschritte das richtige Ergebnis zurückgibt.
- Die totale Korrektheit liegt vor, wenn ein Algorithmus partiell korrekt ist und terminiert.

Algorithmen

Funktionsweise

Es gibt zwei grundlegende Formen die Funktionsweise von Algorithmen zu spezifizieren:



Iteration
(Schleife)

Ein Zähler gibt an wie oft eine Aktion durchlaufen wird.



Rekursion
(Selbstaufwurf)

Es gibt keinen Zähler, stattdessen ruft der Algorithmus sich immer wieder selbst auf.

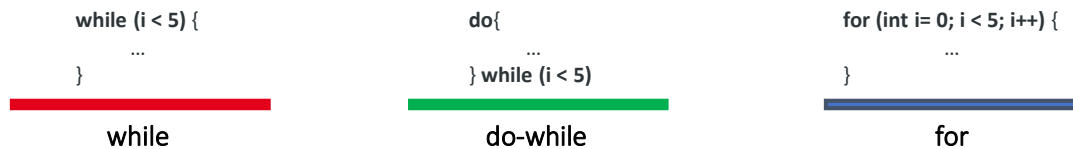
Die Funktionsweise von Algorithmen kann in zwei grundlegend unterschiedlichen Arten umgesetzt werden. Bei der Iteration wird der wiederholte Ablauf von Befehlen mit Hilfe einer Zählervariable bestimmt. Das heißt ein Zähler gibt an wie oft eine Schleife durchlaufen wird.

Bei der Rekursion findet dagegen ein Selbstaufwurf der Methode mit einer reduzierten Komplexität statt. Hierbei wird auf eine Zählervariable verzichtet, stattdessen definiert die Abbruchbedingung (Base Case) wann eine Lösung zurückgegeben wird und kein weiterer Selbstaufwurf stattfindet.

Algorithmen

Iteration

- Bei der Iteration werden Anweisungen mithilfe einer Schleife wiederholt.



- Grundlegend zur Ausführung sind:
 - **Prüfbedingung:** Entscheidet ob eine weitere Wiederholung durchgeführt wird
 - **Zählervariable:** Wird bei jedem Schleifendurchlauf verändert
 - **Anweisung:** Der Befehl der wiederholt wird

Das Wort Iteration stammt vom lateinischen Wort ‚iterare‘ das die Bedeutung von ‚sich wiederholen hat‘. Bei der Iteration werden also buchstäblich Aktionen wiederholt durchgeführt. In Java gibt es hierzu drei grundlegende Wege. Bei der while-Schleife wird die Prüfbedingung für eine Wiederholung der Schleife direkt nach dem Signalwort „while“ definiert. Bei der Do-While Schleife wird erst der do-Teil ausgeführt und dann abhängig von der Prüfbedingung des while-Teils erneut wiederholt. Die For-Schleife definiert nach dem Signalwort „for“ die Zählervariable, die Prüfbedingung und die Veränderung der Zählervariable nach jedem Schleifendurchlauf.

Für die Umsetzung einer Iteration sind drei Bestandteile entscheidend. Als erstes muss die Prüfbedingung klar formuliert werden. Sprich die Prüfung, die entscheidet ob eine weitere Wiederholung durchgeführt wird. Der zweite essentielle Bestandteil ist die Zählervariable, die meist im Rahmen der Prüfbedingung verglichen wird und die bei jedem Schleifendurchlauf verändert wird. Abschließend benötigt jede Iteration einen Anteil von Befehlen, der im Rahmen jeder Wiederholung ausgeführt wird.

Algorithmen

Exkurs Fakultät

- Die Fakultätsfunktion ist eine **grundlegende Funktion** in der Mathematik
- Die Fakultät wird unter anderem zur Ermittlung der Anzahl von Möglichkeiten eingesetzt **n unterscheidbare Gegenstände** in einer Reihe anzuordnen.
- **Berechnung:**

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

Bsp.:

- **Formel:**

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

$$3! = 3 \cdot 2 \cdot 1 = 6$$

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Lassen Sie uns die Umsetzung eines iterativen Algorithmus anhand der Implementierung der mathematischen Fakultätsfunktion veranschaulichen. Die Fakultät ist eine grundlegende mathematische Funktion, die im Rahmen der Kombinatorik eine entscheidende Rolle spielt. Konkret wird mit dieser Funktion die Anzahl der verschiedenen Anordnungen von n unterschiedlichen Gegenständen berechnet. Dies wird erzielt indem das Produkt aller natürlichen Zahlen bis zur Fakultätszahl berechnet wird. Beispielsweise wird die Fakultät zur Zahl 4 durch das Produkt der Zahlen $1 \cdot 2 \cdot 3 \cdot 4$ ermittelt.

Die mathematische Formel zur Berechnung der Fakultätsfunktion besagt, dass die Fakultät der Zahl 0 der Zahl 1 entspricht und für alle anderen Fälle berechnet sich die Fakultät aus dem Produkt der Fakultätszahl mit der Fakultät der um 1 reduzierten Fakultätszahl.

Algorithmen

Iteration

```
public static int calcFacultyIterative (int zahl){
    int result = 1;
    for (int i=1; i<=zahl; i++){
        result = result * i;
    }
    return result;
}
```

- **Zählervariable** *i* wird mit jedem Schleifendurchlauf um 1 erhöht.
- Die **Prüfbedingung** innerhalb der for-Schleife beinhaltet die Terminierungsvorschrift.
- Mit jedem **Schleifendurchlauf** wird das **Zwischenergebnis** mit der Zahl des Iterators multipliziert.

Bsp.: *zahl* = 4



Schauen wir uns nun an wie wir die Fakultätsfunktion mithilfe eines iterativen Algorithmus umsetzen können. Das dargestellte Codebeispiel beginnt mit der Definition der Variable „result“, die sowohl als Teilergebnis als auch das Endergebnis abbildet. Daran anschließend wird eine for-Schleife definiert, welche das Produkt aller natürlichen Zahlen bis zur Fakultätszahl berechnet. Dies wird dadurch erzielt indem mit jedem Schleifendurchlauf das Produkt des Zwischenergebnisses mit der nächsthöheren Zahl berechnet wird und als neuer Wert der Zwischenstandsvariable ‚result‘ abgespeichert wird. Nach Erreichen der Fakultätszahl wird die Schleife ein letztes Mal durchlaufen und das Ergebnis der Zwischenstandsvariable ‚result‘ wird zurückgegeben.

Algorithmen

Iteration

1. Schleife

Ist $i \leq \text{zahl}$?
 $\text{result} = i * \text{result}$
 erhöhe i um 1

2. Schleife

Ist $i \leq \text{zahl}$?
 $\text{result} = i * \text{result}$
 erhöhe i um 1

Jede Schleife umfasst die **vollständige Berechnung** des Zwischenergebnisses.

3. Schleife

Ist $i \leq \text{zahl}$?
 $\text{result} = i * \text{result}$
 erhöhe i um 1

4. Schleife

Ist $i \leq \text{zahl}$?
 $\text{result} = i * \text{result}$
 erhöhe i um 1

5. Schleife

Ist $i \leq \text{zahl}$?
 Gebe result zurück

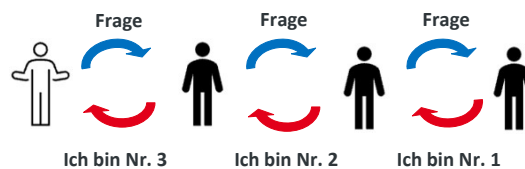
wir haben immer valide Zwischenergebnisse

Wenn wir uns den Ablauf des iterativen Algorithmus schematisch vor Augen führen, fällt auf, dass in jedem Schleifendurchlauf die vollständige Berechnung des Zwischenergebnisses abgeschlossen ist bevor der nächste Schleifendurchlauf startet. Die Terminierungsbedingung bestimmt ausschließlich ab wann kein weiterer Schleifendurchlauf vollzogen wird und der bisherige Zwischenstand zurückgegeben wird.

Algorithmen

Rekursion

- Bei der **Rekursion** werden ebenfalls Wiederholungen durchlaufen, allerdings ohne Zählervariable, stattdessen ruft sich die Methode immer wieder selbst auf.



- Grundlegend für die Ausführung sind:
 - **Abbruchbedingung (Base Case):** Grundlegendste Lösung des einfachsten Falls.
 - **Selbstaufwurf:** Die Methode ruft sich selbst mit einer reduzierten Komplexität auf.

Die zweite grundlegende Weise um einen Algorithmus umzusetzen ist die Rekursion. Auch hier werden wiederholte Abläufe eingesetzt um ein Problem zu lösen. Im Gegensatz zur Iteration kommt bei den Wiederholungen aber keine Zählervariable zum Einsatz sondern die Methode ruft sich immer wieder selbst auf.

Vergegenwärtigen wir uns diesen Ablauf anhand von folgendem Beispiel. Stellen Sie sich vor Sie sind am Ende einer Menschenkette und sehen nicht wie viele Leute vor Ihnen sind. Sie fragen nun Ihren Vordermann an welcher Position er steht. Ihr Vordermann weiß es jedoch auch nicht und fragt seinen Vordermann welche Position dieser hat. Dieses Vorgehen wiederholt sich solange bis die Frage den Ersten in der Schlange erreicht, der mit der Position 1 antwortet. Nun läuft die Antwort vom Anfang der Schlange bis zu deren Ende, und jede Person kann sich ihre Position erschließen indem sie 1 zur Antwort des Vordermanns addiert.

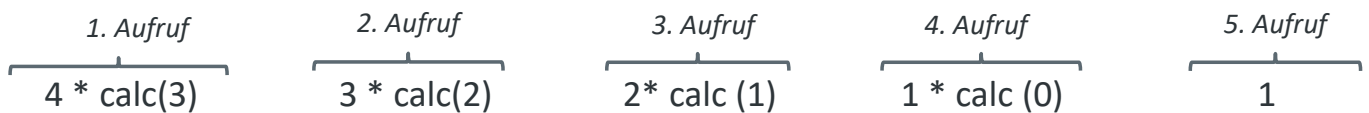
Dieses einfache Beispiel veranschaulicht die zwei essentiellen Teile die für eine rekursive Ausführung benötigt werden. Zum einen ist dies die Abbruchbedingung oder auch Base Case der die einfachste Lösung des Problems darstellt (in unserem Falle war dies die Person am Anfang der Schlange). Des Weiteren benötigt jede Rekursion den Selbstaufwurf mit einer reduzierten Problemkomplexität. In unserem Beispiel war dies die Weitergabe der Frage an die Personenschlange die jeweils um den Fragenden reduziert wird.

Algorithmen

Rekursion

```
public static int calcFacultyRecursive(int zahl){
    if (zahl == 0){
        return 1;
    }else{
        return zahl * calcFacultyRecursive( zahl: zahl -1);
    }
}
```

- Der **Base Case** ist wenn die übergebene **Zahl 0** ist.
- Im **else Teil** ruf die Methode sich selbst auf mit einer **um eins reduzierten Zahl**.
- Erst nachdem der Base Case erreicht wird, findet die Berechnung der Zwischenergebnisse rückwärtsgerichtet statt.



12

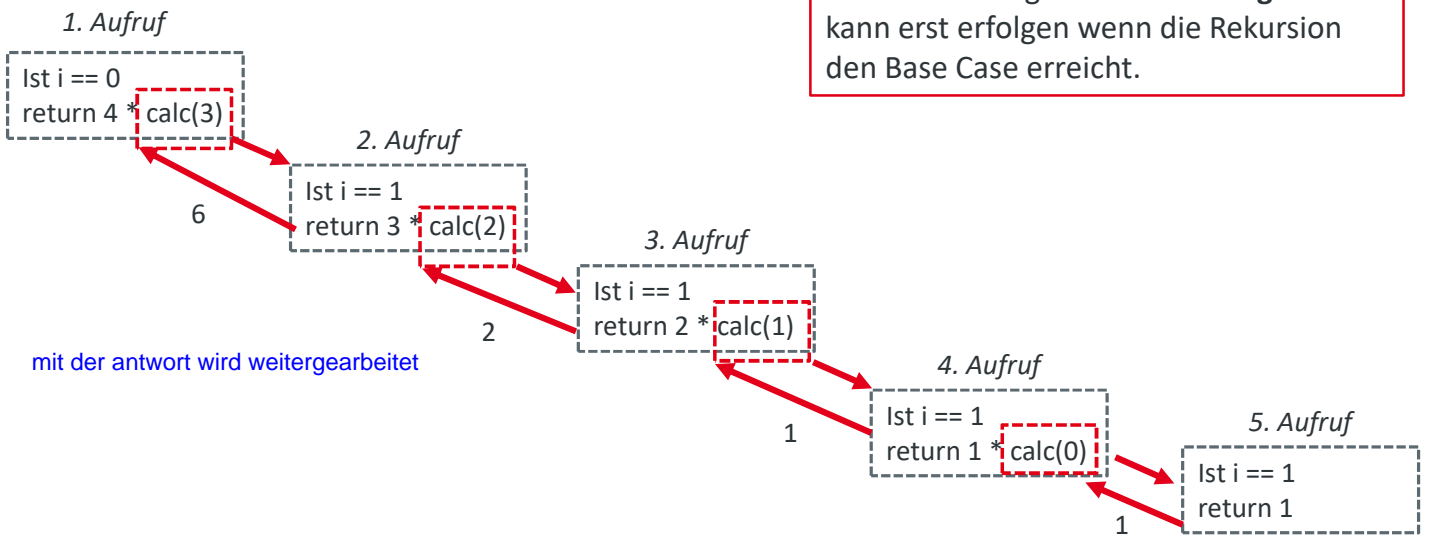
Die rekursive Berechnung der Fakultätsfunktion gestaltet sich wie folgt: Den Anfang macht die Abbruchsbedingung (Base Case) in unserem Fall ist dies wenn die Zahl 0 ist. Ist dies der Fall wird 1 zurückgegeben, in allen anderen Fällen wird der aktuelle Zahlenwert mit dem Ergebnis der Fakultätsfunktion der um 1 reduzierten Zahl multipliziert.

Veranschaulichen wir uns den Funktionsaufruf anhand des unten dargestellten Ablaufs. Wie schon im vorhergehenden Beispiel mit der Personenkette wird bei jedem Selbstaufruf die Komplexität der Problems um eins reduziert, sobald man den Base Case erreicht und die erste Rückgabe erfolgt, findet die Berechnung der Zwischenergebnisse in Gegenrichtung zum Aufruf statt.

Algorithmen

Rekursion

Die Berechnung der **Zwischenergebnisse** kann erst erfolgen wenn die Rekursion den Base Case erreicht.



13

Lassen Sie uns nun die Selbst-Aufruf Reihenfolge des Rekursionsablaufs schematisch skizzieren. Es fallen hierbei zwei wichtige Unterschiede im Vergleich zur Iteration auf. Zum Einen wird beim rekursiven Vorgehen die Berechnung des Zwischenergebnisses erst abgeschlossen, wenn durch wiederholten Selbstaufruf der Base Case erreicht wird. Der zweite grundlegende Unterschied, ist dass keine Zählervariable zur Steuerung des Rekursionsaufrufs genutzt wird, sondern das in der Komplexität reduzierte Ausgangsproblem.

Aufgabe 1

- Die Fibonacci-Zahlenreihe besteht aus den folgenden Anfangswerten:

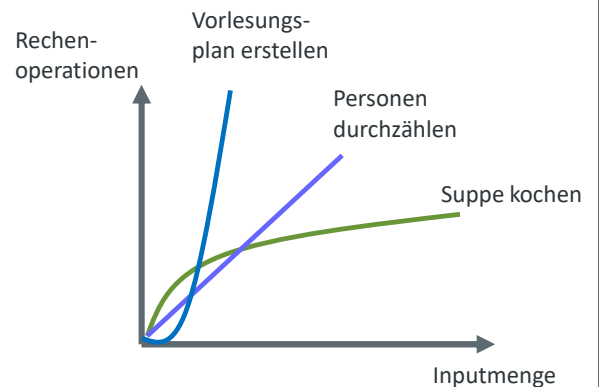
fibonacci(x)	0	1	1	2	3	5	8	13	21	34	55
x	0	1	2	3	4	5	6	7	8	9	10

- Die Anfangswerte $\text{fibonacci}(0) = 0$ und $\text{fibonacci}(1) = 1$ sind vorgegeben.
- Jede darauffolgende Zahl wird aus der Summe der beiden vorgehenden Fibonaccizahlen berechnet.
- Verfassen Sie die Methoden
 - `public static long fibo_rec(int orderNumber)` zur rekursiven Berechnung der Fibonacci-Zahl
 - `public static long fibo_iter(int orderNumber)` zur iterativen Berechnung der Fibonacci-Zahl (*Hausaufgabe*)

Algorithmen

Vergleich

- Neben der Korrektheit ist die **Effizienz** (Bedarf an Rechenoperationen und Speicher) **entscheidend** um Algorithmen zu beurteilen und zu vergleichen.
- Eine der entscheidenden Fragen hierbei ist, wie sich das **Laufzeitverhalten** des Algorithmus bei **steigender Inputmenge** verhält.
- Für die Analyse des Laufzeitverhaltens ist die **Klassifikation des Wachstumsverhaltens** viel wichtiger als die genaue Bestimmung des Ressourcenverbrauchs.
- Bei rechenintensiven Algorithmen können leistungsstärkere Computer nicht immer weiterhelfen.



Zur Beurteilung und Abgrenzung von Algorithmen ist neben der Korrektheit ihre Effizienz, d.h. der Bedarf an Speicher und Rechenoperationen, entscheidend. Hierbei wird insbesondere untersucht, wie sich der Algorithmus bei einer steigenden Inputmenge verhält. Nehmen wir als Beispiel folgende beispielhafte Algorithmen. Wenn Sie frisch eine Suppe kochen, ist der generelle Aufwand gleich egal ob Sie die Suppe für 1 Person oder 10 Personen zubereiten. Das heißt bei der Zubereitung der Suppe entsteht viel Aufwand für den 1 Teller, dann aber relativ wenig Aufwand für die zusätzlichen Teller. Ein anderes Beispiel ist das Durchzählen einer Personengruppe. Hier steigen die notwendigen Rechenoperationen mit der Anzahl der Personen, das heißt Sie benötigen zum Durchzählen von 20-Personen ca. doppelt so lange wie für 10-Personen. Abschließend befassen wir uns mit der Vorlesungsplanung, für die mit jedem Dozenten Abstimmungen geführt und Räume gebucht werden müssen. Der notwendige Aufwand hierfür steigt überproportional mit der Anzahl der Veranstaltungen. Für die Beurteilung von Algorithmen sind insbesondere große Inputmengen relevant anhand derer ihr Aufwandswachstum bewertet werden kann. Das Aufwandswachstum ist vor allem deshalb wichtig, da selbst heute leistungsstarke Computer bei komplexen Algorithmen an Ihre Grenzen kommen.

Algorithmen

Beispiel 1



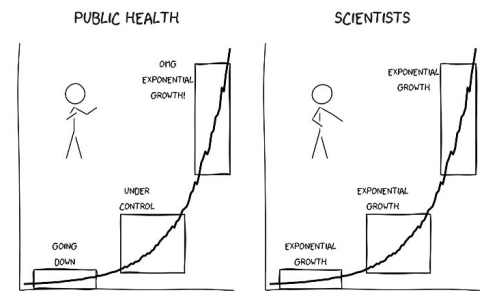
McGeddon, CC BY-SA 4.0,
<https://commons.wikimedia.org>

„Ich möchte nicht mehr als dass jedes Feld mit je der doppelten Anzahl an Reiskörnern aufgewogen wird.“

Beispiel 2

Gehen Sie von einem Bakterium aus, das jede Minute sein Wachstum verdoppelt. Um 12:00 Uhr ist die Petrischale zum ersten mal voll mit Bakterien, um wieviel Uhr war die Petrischale halb voll?

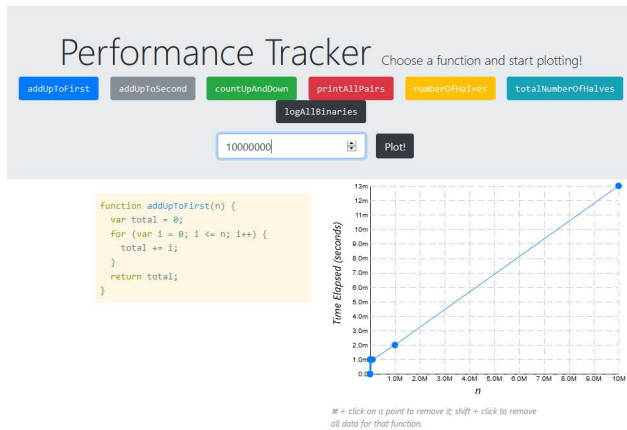
Beispiel 3



pic.twitter.com/a8LvlmZxT7

Algorithmen

Landau Notation: Hands-On



<https://rithmschool.github.io/function-timer-demo/>

Erkenntnisse:

- Bei kleinen Eingabemengen gibt es zu viele **Störgrößen**.
- Für den **Vergleich** von Algorithmen eignen sich insb. **große Eingabemengen**.
- Das Verhalten der Algorithmen kann anhand von **Komplexitätsklassen** kategorisiert werden.
- **Kleine Unterschiede** der Algorithmen können **zentrale Auswirkungen** auf die Laufzeit haben

Die anschaulichen Visualisierungen auf der verlinkten Website zeigen, dass bei kleinen Eingabegrößen sehr viel Störgrößen auf den Algorithmus wirken, sodass keine klare Entscheidung zur Vorteilhaftigkeit einzelner Algorithmen getroffen werden kann. Je größer die Eingabegröße aber gewählt wird, desto klarer zeigt sich die Vorteilhaftigkeit einzelner Algorithmen, teilweise mit gravierenden Unterschieden. Des Weiteren fällt auf, dass je größer der Betrachtungshorizont gewählt wird, desto unbedeutender werden konstante Faktoren. Es zeigt sich jedoch auch, dass schon kleine Codeteile gravierende Auswirkungen auf das Laufzeitverhalten haben können.

Algorithmen

Landau Notation: Grundlagen

- Mit der **Landau Notation („Groß O“)** wird das **Grenzverhalten** von Algorithmen charakterisiert, also wenn die unabhängige Variable ins Unendliche verläuft.
- Das Laufzeitverhalten der Algorithmen wird dabei für den **günstigsten (Best-Case)**, **schlechtesten (Worst-Case)** und **durchschnittlichen Fall (Avg.-Case)** bestimmt.
- Die Symbolik „ $O(n)$ “ steht für „**Ordnung von**“ und bezeichnet die obere Schranke der Anzahl von notwendigen Rechenoperationen für eine Inputmenge.
- $O(n)$ besagt, dass **ungefähr so viele Rechenoperationen** notwendig sind, **wie die Inputmenge Elemente** umfasst.



Geht man ein Telefonbuch mit n Einträgen von vorn nach hinten durch um einen Namen zu suchen benötigt man im besten Fall (bspw. Abt) eine Operation und im schlechtesten Fall n Operationen (bspw. Zander). In der Regel, ist der gesuchte Name jedoch in der Mitte.

18

Die Landau Notation ist eine formale Schreibweise um das Grenzverhalten von Algorithmen zu kategorisieren, also die zuvor besprochene Unterscheidung der Aufwandskomplexitäten. Hierzu wird das Grenzverhalten der notwendigen Aufwände für die günstigste, die durchschnittliche und die schlimmste Ausgangsbasis bestimmt und gegenübergestellt. Zur Beschreibung der Komplexitätsklasse wird die Symbolik „O“ verwendet, die ausgesprochen für „Ordnung von“ steht. Das heißt die Symbolik beschreibt formal eine obere Grenze der notwendigen Operationen eines Algorithmus, die bei steigender Inputmenge zum Tragen kommt. Die Notation „ $O(n)$ “ besagt beispielsweise, dass der Aufwand linear mit der Inputmenge wächst (z.B.: Durchzählen einer Personengruppe, oder sequenzielle Namenssuche im Telefonbuch). Somit lässt sich auch ohne grafische Darstellung der konkreten Aufwandsverläufe, sehr einfach die Komplexitätsklasse eines Algorithmus kommunizieren.

Algorithmen

Landau Notation: Komplexitätsklassen

Folgende Komplexitätsklassen werden unterschieden

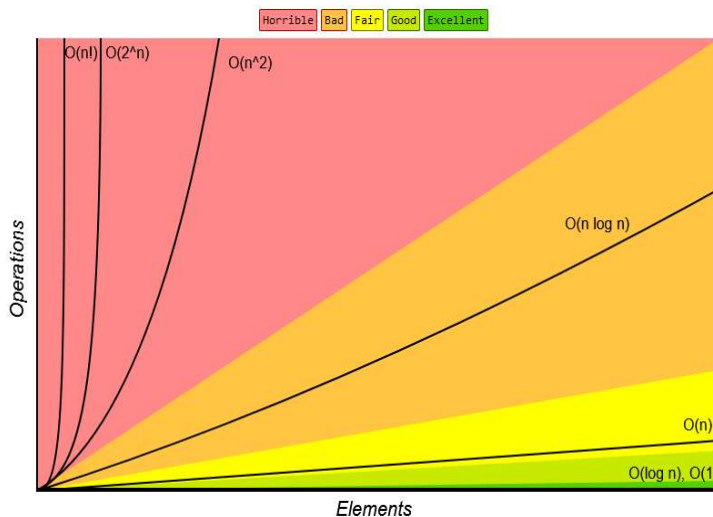
Bezeichnung	Schreibweise	
konstant	$\mathcal{O}(1)$ <small>input gröÙe ist irrelevant</small>	<div> <div>klein</div> <div>↓</div> <div>groß</div> </div>
logarithmisch	$\mathcal{O}(\log n)$	
polylogarithmisch	$\mathcal{O}(\log^k n)$ für $k \geq 1$	
linear	$\mathcal{O}(n)$	
linear-arithmetisch	$\mathcal{O}(n \log n)$	
quadratisch	$\mathcal{O}(n^2)$	
polynomial	$\mathcal{O}(n^k)$ für $k \geq 1$	
exponentiell	$\mathcal{O}(d^n)$ für $d > 1$	

19

Die folgende Tabelle listet die grundlegenden Komplexitätsklassen die bei der Landau-Notation unterschieden werden in aufsteigender Komplexität auf.

Algorithmen

Landau Notation: Komplexitätsklassen



Quadratisches Wachstum markiert generell den Übergang von akzeptablem zu inakzeptablem Laufzeitverhalten bei großem Input.



Bruce Dawson (Antifa)
@BruceDawson0xB

$O(n^2)$ is the sweet spot of badly scaling algorithms: fast enough to make it into production, but slow enough to make things fall down once it gets there

7:38 nachm. · 22. Apr. 2019 · Twitter Web Client



Office of Entroper Zero @EntroperZero · 9. Dez. 2019

Antwort an @BruceDawson0xB und @ben_a_adams
I recently fixed an $O(n^3)$ that made it to production. All of our input sizes were < 100 so it was "fine", until it wasn't when we added one that was ~ 1000 .

vgl. bigocheatsheet.com

20

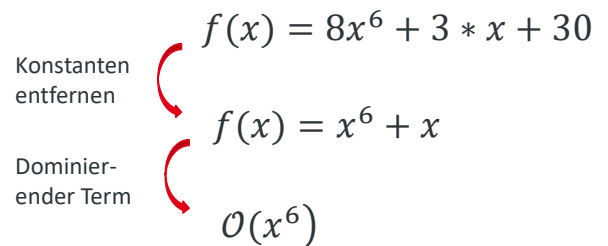
Die dargestellte Grafik veranschaulicht den drastischen Unterschied hinsichtlich der benötigten Operationen bei steigender Inputgröße. Sie sehen insbesondere wie steil die quadratischen und exponentiellen Komplexitätsklassen verlaufen und so eine sehr hohe Anzahl an Operationen verursachen. Im Vergleich hierzu führen lineare und logarithmische Komplexitätsklassen zu geringen zusätzlichen Operationen und somit auch zu einer schnellen Durchführung. Allgemein kann festgehalten werden, dass sich insbesondere das quadratische Wachstum als tückisch erweist, da hier oft der Aufwand unter Testbedingungen tolerabel ist, während sich im Praxiseinsatz derartige Algorithmen zu zentralen Flaschenhälsen entwickeln können.

Algorithmen

Landau Notation: Ableitung

Zur **Einteilung eines Algorithmus** in eine Komplexitätsklasse gelten folgende Vorschriften:

- Konstante Faktoren ignorieren
 - $\mathcal{O}(2n) \rightarrow \mathcal{O}(n)$
 - $\mathcal{O}(16 + n) \rightarrow \mathcal{O}(n)$
- Nur der dominierende Term wird betrachtet
 - $\mathcal{O}(4n^2 + 5n + 1) \rightarrow \mathcal{O}(n^2)$
 - $\mathcal{O}(5^n + n^4 + \log(n)) \rightarrow \mathcal{O}(5^n)$
- Unabhängige Inputvariablen werden mit unterschiedliche Variablen charakterisiert
 - $\mathcal{O}(4n^2 + 5k) \rightarrow \mathcal{O}(n^2 + k)$



$$f(x) = 8x^6 + 3 * x + 30$$

Konstanten entfernen

$$f(x) = x^6 + x$$

Dominierender Term

$$\mathcal{O}(x^6)$$

Zur Ermittlung einer Komplexitätsklasse in der Landau Notation gelten folgende drei Vorschriften:

- Zunächst werden konstante Faktoren weggelassen. Dadurch, dass die Landau Notation den asymptotischen Verlauf (den Grenzverlauf) beschreibt können konstante Faktoren ignoriert werden. Das betrifft konstante Faktoren genauso wie alleinstehende Konstanten.
- Die zweite Vorschrift besagt, dass nur der dominierende Teil einer Aufwandsgleichung von Bedeutung ist. Diese Regel ist insb. wichtig wenn der Aufwand eines Algorithmus aus verschiedenen zusammengesetzten Termen besteht. In derartigen Fällen wird nur jener Teil betrachtet, der im Grenzverlauf das höchste Wachstum aufweist.
- Zuletzt ist es wichtig, dass unabhängige Inputvariablen durch unterschiedliche Variablen in der Aufwandsgleichung berücksichtigt werden. In der Praxis erkennen Sie unabhängige Inputvariablen in Form von unterschiedlichen Parametern bei einem Methodenaufwurf.

Algorithmen

Konstantes Wachstum

- Die durchgeführten Operationen sind **unabhängig vom Umfang** der Eingabemenge n , bspw.:
 - Arithmetische Operationen (+, -, *, /)
 - Variablenzuweisung
 - Zugriff auf die Felder eines Arrays
- In der Landau-Notation wird bei einer konstanten Anzahl von Operationen $\mathcal{O}(1)$ geschrieben.
- **Beispiel:** Lesezeichen in einem Buch. Für das Öffnen der Seite ist es egal ob das Buch 30 oder 3000 Seiten hat.

```
private int constantRuntime(int[] array){  
    int result = array[0] + array[3];  
    return result;  
}
```

Die vorgenommenen Operationen sind unabhängig von der Länge des Arrays.

Auf den folgenden Folien befassen wir uns mit verschiedenen Praxisbeispielen zur Ableitung der Aufwandsgleichung und deren Überführung in die Landau-Notation. Algorithmen welche ein konstantes Wachstum aufweisen operieren oft unabhängig vom Umfang der Eingabemenge, z.B.: arithmetische Operationen, Variablenzuweisungen aber auch Zugriffe auf einzelnen Felder eines Arrays. Der dargestellte Code, zeigt ein Beispiel wie bestimmte Felder eines Arrays addiert werden, diese Operation ist jedoch unabhängig von der Größe von n , sodass man in diesem Fall dem Algorithmus ein konstantes Wachstum unterstellt. Gemäß der Landau-Notation wird das konstante Wachstum eines Algorithmus als $\mathcal{O}(1)$ formuliert.

Algorithmen

Lineares Wachstum

- Die durchgeführten Operationen **wachsen in linearem Verhältnis** (Proportion) mit der Eingabemenge.
- Zur Komplexitätsabschätzung einer Schleife wird die Arraylänge mit den Operationen innerhalb der Schleife multipliziert.
- **Beispiel:** Lesen eines Buches. Bei 1 Minute für 1 Seite benötigt das Lesen eines Buches mit 30 Seiten 30 Minuten und mit 100 Seiten 100 Minuten.

```
private int linearRuntime(int[] array){  
    int compareTo = 3;  
    int matches = 0;  
    for (int i=0; i < array.length -1; i++){  
        if (array[i] == compareTo){  
            matches = matches +1;  
        }  
    }  
    return matches;  
}
```

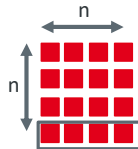
Die For-Schleife geht einmal über alle Elemente im Array. Bei zwei Operationen pro Schleifendurchlauf ergeben sich so $2 \cdot n$ Operationen $\rightarrow O(n)$

Algorithmen weisen ein lineares Wachstum auf, wenn ihr Aufwand linear mit der Inputgröße steigt. Ein typisches Beispiel für ein lineares Wachstum ist eine einfache for-Schleife, die über alle Felder eines Arrays iteriert. Hier hängt die Dauer eines Schleifendurchlaufs mit der Gesamtgröße des Arrays zusammen, sodass die Gesamtbearbeitungsdauer in linearem Verhältnis zur Gesamtgröße des Arrays verläuft. Ein typisches Beispiel hierzu ist das Lesen eines Buches. Hat das Buch einen 10-fach verringerten Umfang, so benötigen Sie auch 10mal kürzer zum Lesen des Buches.

Algorithmen

Quadratisches Wachstum

- Die durchgeführten Operationen steigen in **quadratischem Verhältnis** zur Eingabemenge.
- Für jedes Feld im Array wird bspw. der gesamte Array erneut durchlaufen



- Beispiel:** Überprüfung ob in einem Buch eine Seite doppelt vorkommt indem stets für jede Seite das gesamte Buch neu nach Übereinstimmungen überprüft wird.

```
private int[] quadraticRuntime(int[] array){  
    int[] matches = new int[array.length];  
    for (int i=0; i < array.length -1; i++){  
        for(int k=0; k < array.length -1; k++){  
            if(array[i] == array[k]){  
                matches[i] = matches[i] +1;  
            }  
        }  
    }  
    return matches;  
}
```

Geschachtelte for-Schleife bedeutet in diesem Fall, dass für jedes Element die Eingabemenge erneut durchlaufen wird -> $O(n^2)$

Ein Algorithmus wächst quadratisch, wenn die Anzahl der benötigten Operationen in quadratischem Verhältnis zur Eingabemenge steigt. Ein quadratisches Wachstum tritt oft im Falle von ineinander geschachtelten for-Schleifen ein. Sind mindestens zwei for-Schleifen ineinander geschachtelt, die über die gesamte Länge des Arrays iterieren, so heißt dies, dass für jedes Feld der Array erneut durchlaufen wird. In Folge dessen ergibt sich eine Gesamtkomplexität von n^2 . Sie können sich dies auch plastisch so vorstellen, als müssten Sie für jede Seite im Buch überprüfen ob diese im Buch doppelt vorkommt, sodass Sie für jede Seite das Buch komplett durchblättern müssen. Das Ergebnis dieser Überprüfung ist, dass Sie das Buch n-fach langsamer lesen können als ohne diese Duplikats Prüfung.

Algorithmen

Logarithmisches Wachstum $O(\log n)$

- Die durchgeführten Operationen stehen in einem **logarithmischen Verhältnis** zur Eingabemenge.
- Geringere Komplexität als lineares Wachstum aber höher als konstantes Wachstum.
- **Beispiel:** Begriffssuche in einem Wörterbuch. Annäherung indem die sortierte Reihenfolge der Begriffe beachtet wird.

```
private int logarithmicRuntime(int[] array){  
    int count = 0;  
    for (int i=2; i<array.length;i=i*2){  
        count = count + array[i];  
    }  
    return count;  
}
```

Schleife wird nicht für jedes Element durchlaufen sondern nimmt exponentielle Sprünge an.

Abschließend befassen wir uns mit dem logarithmischen Wachstum. Diese Art des Wachstums tritt beispielsweise ein, wenn nicht alle Elemente einer Inputmenge betrachtet werden, sondern nur eine Teilmenge, die im Gegensatz zur Inputmenge schrumpft. Ein Beispiel hierfür ist eine for-Schleife, die nicht sequenziell über alle Elemente eines Arrays iteriert sondern exponentielle Sprünge annimmt. Dadurch wird nicht jedes Element besucht sondern nur eine kleine Teilmenge aller Elemente. Ein konkretes Alltagsbeispiel hierzu ist die Begriffssuche in einem Wörterbuch, hier suchen Sie auch nicht jede Seite nach dem Begriff ab, sondern nähern sich anhand der Nähe des Anfangsbuchstabens und der folgenden Buchstaben schrittweise an das Wort an. Die Basis des Logarithmus spielt bei der Landau Notation keine Rolle.

Aufgabe 2

Teilaufgabe a

- I. Welche Komplexität weist der folgende Algorithmus auf?

quadratisch

$O(n^2)$

nur das dominierende besteht am ende

$O(1) + O(n) + O(n^2) \rightarrow$ dominierende bleibt übrig $O(n^2)$

nicht abhängig
von der länge

```
private int magicOperation(int[] array){
    int sum = array[0] + array[4] + array[6];
    for(int i=1; i<array.length; i++){
        sum = sum + array[i];
    }
    for(int i=1; i<array.length; i++){
        for(int b=1; b<array.length; b++){
            sum = sum + array[i] + array[b];
        }
    }
    return sum;
}
```

Aufgabe 2

Teilaufgabe a

- II. Welche Komplexität weist der folgende Algorithmus auf?

on
lineares wachstum

$O(10n)$
↓
 $O(n)$

$O(1)$ ↓

```
private int arrayMagic(int[] array){  
    int sum = 0;  
  
    for(int i=0; i<array.length; i++){  
        for(int k=0; k<10; k++){  
            sum = sum + array[k] * array[i];  
        }  
    }  
    return sum;  
}
```

Aufgabe 2

Teilaufgabe a

- III. Welche Komplexität weist der folgende Algorithmus auf?

zwei arrays

$n*m \rightarrow O(n*m)$

$\rightarrow O(1) + O(n*m)$

$O(1)$

```
private int arrayMagic2(int[] array1, int[] array2){  
    int sum = 0;  
  
    for (int i=0; i<array1.length; i++){  
        for (int m=0; m<array2.length; m++){  
            sum = sum + array1[i]*array2[m];  
        }  
    }  
    return sum;  
}
```

tabelle gibt wieder, welche werte eine komplexitätsklasse hat

Aufgabe 2

Teilaufgabe b

Bitte beurteilen Sie die Laufzeitverhalten der folgenden beiden Algorithmen:

- $g(x) = 0,0001x * 4x$ quadratisch? $O(x^2)$
- $b(x) = 360x$ linear

- I. Wie verhält sich die asymptotische Verhalten der beiden Algorithmen? Welche der beiden Algorithmen hat eine geringere Komplexitätsklasse?
- II. Wie verhalten sich die Algorithmen wenn x die Größe 4, 400, 400.000.000
100, 1000, 1000000 annimmt? 36000, 360.000, 360.000.000
- III. Welchen der beiden Algorithmen würden Sie für den Praxiseinsatz empfehlen?

Agenda

Algorithmen

Was ist ein Algorithmus
und wie lässt sich die
Güte dieser bewerten?

Datenstrukturen

Was sind zentrale
Datenstrukturen und
wie unterscheiden sie
sich.

Laufzeitanalyse Datenstrukturen

Wie unterscheiden sich
Datenstrukturen im
Praxiseinsatz.

Datenstrukturen

Grundlagen

- **Datentypen** beschreiben die **Art** und den **Wertebereich** in der Informationen innerhalb einer Anwendung abgelegt werden.
- **Datentypen** bestimmen die **Operationen** (Funktionalität) einer Variable.
- **Primitive Datentypen** können **nicht weiter zerlegt** werden (bspw. int, double, char, boolean).
- **Komplexe Datentypen** sind aus **komplexen** sowie **primitiven** Datentypen **zusammengesetzt**.

```
int[] test = new int[50];
```



Datentyp 'Array' -> Möglichkeit des Indexzugriffs auf Felder und Längenbestimmung via .length

Die Datentypen beschreiben die Art und den Wertebereich in der Informationen einer Anwendung abgelegt werden. Nehmen wir als Beispiel den Datentyp Integer (Ganzzahl) durch die Auswahl dieses Datentyps wird nicht nur der Wertebereich vorgegeben (-2^{31} bis 2^{31}), sondern auch was für Operationen auf diesen angewendet (+, -, *, etc.) und nicht angewendet werden können (bspw. size()).

In Java wird zwischen primitiven und komplexen Datentypen unterschieden. Primitive Datentypen zeichnen sich dadurch aus, dass sie nicht weiter zerlegt werden können (bspw. char, int, double, long, etc). Komplexe Datentypen können dagegen weiter in komplexe Datentypen oder in primitive Datentypen zerlegt werden.

Datenstrukturen

Grundlagen

- **Casting** umschreibt die **Umwandlung des Datentyps** einer Variable in einen anderen Datentyp.
- Ist der **Wertebereich** des **Zielfatentyps größer** werden **alle** Informationen des Ausgangstyps beibehalten.
- Ist der **Wertebereich** des **Zielfatentyps kleiner** gehen bei der Konvertierung **Informationen verloren**.
- Beim **impliziten Casting** kann in der Variablenzuweisung **wertebereich erweiterung, ganzzahl in komma** direkt ein neuer Wertebereich verwendet werden
- Beim **expliziten Casting** muss der Zielfatentyp in **wertebereich verkleinerung, werteverlust, komma in ganzzahl** Klammern geschrieben werden.

```
int ganzZahl = 2;
double dezimalZahl = 8.333333;

int ergebnis = (int) (ganzZahl/dezimalZahl)

double neueZahl = ganzZahl;
```

32

Die Umwandlung des Datentyps einer Variable in einen anderen wird auch als Casting bezeichnet. Hierbei gibt es zwei Möglichkeiten, entweder wird der ursprüngliche Wertebereich erweitert und so alle Informationen des Ausgangstyps werden beibehalten, oder der Zielwertebereich verkleinert den ursprünglichen Wertebereich wobei Information verloren gehen. In Java können diese Typumwandlungen entweder implizit oder explizit vorgenommen werden. Beim impliziten Casting wird ein neuer Datentyp mit dem alten Wert instanziiert werden. Der Name implizit kommt daher, da Java automatisch (implizit) die Umwandlung vornimmt. Beim expliziten Casting, wird dagegen der Zielfatentyp in Klammern vor der konkreten Wertzuweisung platziert. Explizites Casting ist insb. dann notwendig wenn eine Wertebereichverkleinerung vorgenommen wird (bspw. double -> int).

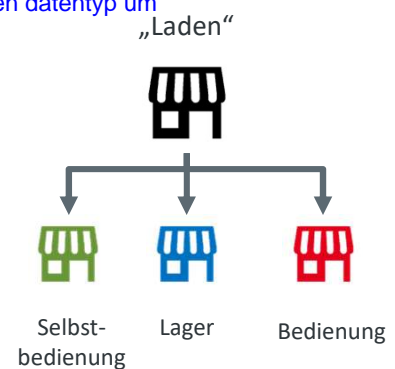
Datenstrukturen

Grundlagen

- Datenstrukturen beschreiben die **Zugriffs- und Speichermechanismen** von Informationen.
- Während Datentypen die Funktionalität beschreiben, **beschreiben Datenstrukturen wie diese Funktionen umgesetzt werden** („was“ vs. „wie“) **es geht um das was und das wie**
- **Statische Datenstrukturen** haben einen **festen Speichergröße** während **dynamische Datenstrukturen** ihre Größe **flexibel erweitern** können.
- Die Datenstruktur muss immer im Hinblick auf den Einsatzzweck ausgewählt werden.

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

Linus Torvalds



Während Datentypen das WAS (im Sinne von Art und Wertebereich) charakterisieren, beschreiben Datenstrukturen WIE der Zugriff und das Speichern von Informationen umgesetzt werden. Nehmen wir zur Verdeutlichung der Beziehung zwischen Datentyp und Datenstruktur das Beispiel eines Ladens. Aus Außensicht können wir klar einen Laden spezifizieren im Sinne eines Ortes an dem wir bestimmte Aktionen umsetzen können, wie den Einkauf von Waren (vgl. Datenstruktur). Diese abstrakte Beschreibung sagt aber nichts darüber aus wie dieser Laden konkret ausgestaltet ist bspw. ob Selbstbedienung herrscht oder man bedient wird, wie die Waren ausgestellt werden, etc. (vgl. Datenstruktur).

Wir unterscheiden zwischen statischen Datenstrukturen und dynamischen Datenstrukturen. Statische Datenstrukturen haben eine fixe Speicherkapazität, wohingegen dynamische Datenstrukturen flexibel ihren Umfang erweitern können.

Abschließend ist anzumerken, dass es keine „one-size fits all“ Datenstruktur gibt, sondern die beste Datenstruktur stets anhand der situativen Zugriffs- und Speicherbedarfe ausgewählt werden muss.

jede aufgabe, neue zeile
wir entscheiden, wie wir diese liste angehen

Datenstrukturen

Lineare Datenstruktur

- Bei linearen Datenstrukturen werden Elemente in einer **sequenziellen Reihenfolge** abgelebt.
- Jedes Element hat eine **eindeutige Position** in der Reihenfolge.
- Lineare Datenstrukturen unterscheiden sich hinsichtlich ihrer **Zugriffs- und Abarbeitungsreihenfolge**. individuell
- **Zentrale Operationen:**
 - Einfügen eines neuen Elements
 - Löschen eines bestehenden Elements
 - Ist-In, Abfrage nach einem Element
 - Länge, Länge der Reihenfolge



Bsp. ToDo-Liste welche auf einen Blatt geschrieben wird. Einträge werden von oben nach unten gelesen

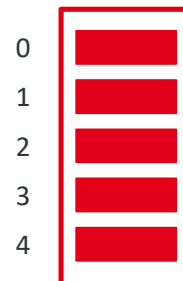
Lineare Datenstrukturen kategorisieren jene Datenstrukturen, die Elemente in einer sequenziellen Reihenfolge anordnen. Ein praktisches Beispiel hierfür ist eine To-Do Liste auf der Sie untereinander Ihre Aufgaben eintragen. Jede Aufgabe besitzt hierbei eine klare Position. Aufbauend auf dieser sequenziellen Anordnung kann es nun unterschiedliche Zugriffs- und Abarbeitungsstrategien geben. Manche von uns fangen bspw. an ihre ToDo-Liste von unten her abzuarbeiten, während andere von oben her die Abarbeitung beginnen. Eine andere Form der Abarbeitung kann in der Mitte beginnen. Wichtig hierbei ist, dass stets nur eine Aufgabe zur gleichen Zeit abgearbeitet wird.

Zentrale Operationen für lineare Datenstrukturen sind das Einfügen neuer Elemente, das Löschen von bestehenden Elementen, die Abfrage ob ein bestimmtes Element in der Liste ist und die Abfrage der Länge der Liste.

Datenstrukturen

Array

- Ein Array speichert Datenobjekte des **gleichen Typs** (bspw.: int, double, String, etc.).
- Der Array ist eine **statische Datenstruktur**, dessen Größe nachträglich **nicht erweitert** werden kann.
- Nach Überschreiten der **Kapazitätsgrenze** ist kein weiteres Einfügen von Elementen möglich.
- Die einzelnen Einträge im Array werden über einen **numerischen Index** angesprochen.



Ein Array kann man sich wie ein Büroschrank mit Fächern vorstellen. Ein bestimmtes Fachs kann über die Nummerierung ausgewählt werden.

Eine grundlegende lineare Datenstruktur ist der Array. Ein Array speichert Elemente des gleichen Typs in einer Datenstruktur mit einer fixen Kapazität ab. Diese fixe Kapazität macht den Array zu einer statischen Datenstruktur, da nach Überschreiten der Kapazitätsgrenze kein neues Element eingefügt werden kann. Eine wichtige Eigenschaft des Arrays ist, dass die einzelnen Elemente in ihm über einen numerischen Index angesprochen werden.

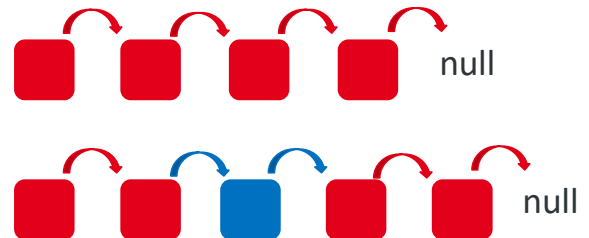
Ein gutes Beispiel um sich die Array Datenstruktur zu veranschaulichen ist ein Aktenschrank in einem Büro. Der Aktenschrank hat wie der Array eine feste Kapazität. Beim Überschreiten dieser Kapazität ist es erforderlich einen neuen Schrank anzuschaffen. Der Zugriff auf die einzelnen Fächer des Aktenschranks erfolgt direkt, wenn man weiß welches Fach man auswählen muss.

Datenstrukturen

Linked List

- Bei einer verketteten Liste referenziert **jedes Listenelement seinen direkten Nachfolger**.
- Das **letzte Element** einer verketteten Liste verweist auf **null**.
- Wie bei einer Perlenkette entsteht eine Sequenz von **aneinander gereihten Listenelementen**.
- Ein neues Element kann an jeder Position der Kette eingefügt werden. Hierzu muss die **Referenz des Vorgängers** und die **Referenz auf den Nachfolger** angepasst werden.

einzelne elemente verweisen auf ihren nachfolger
das letzte verweist auf nix = null



Beim Einfügen eines neuen Elements wird an der gewünschten Stelle die Referenz des direkten Vorgängers und des direkten Nachfolgers angepasst.

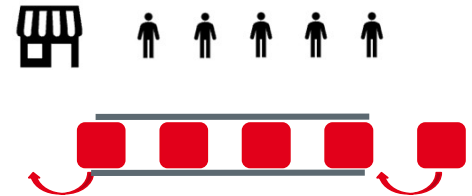
man kann dynamisch elemente hinzufügen

Die verkettete Liste (eng. Linked-List) ist eine lineare und dynamische Datenstruktur. Im Gegensatz zum Array gibt es hier keine Kapazitätsgrenze, sodass flexibel weitere Elemente an jeder Stelle hinzugefügt werden können. Dies wird umgesetzt indem jedes Element der Kette auf seinen direkten Nachfolger verweist, ähnlich wie die Glieder einer Kette. Da das letzte Element der Kette keinen Nachfolger hat, verweist dieses auf „null“. Das Ergebnis dieser schrittweisen Verkettung ist eine Listenstruktur, die dynamisch erweitert werden kann indem die Referenz des Vorgängers auf das neue Element und die Referenz des neuen Elements auf den neuen Nachfolger gesetzt wird. Im Gegensatz zum Array ist hier kein indizierter Zugriff auf ein Element möglich sondern nur indem man von Element zu Element „hüpft“.

Datenstruktur

Queue

- Die Queue ist eine Sonderform der verketteten Liste, die den **Elementzugriff nach dem FIFO (First-In-First-Out)-Prinzip** umsetzt.
- **FIFO-Prinzip:** neue Elemente werden am Ende hinzugefügt während stets das erste Element entfernt wird.
- „**enqueue**“ fügt ein neues Element am Ende ein, „**dequeue**“ entfernt das vorderste Element.
- Queues werden häufig zur **sequenziellen Bearbeitung** parallel eingehender Anfragen eingesetzt (bspw. Ticketreservierung).



Queues funktionieren ähnlich zu Schlangen an einer Supermarktkasse. Neue Kunden reihen sich am Ende ein und der vorderste Kunde verlässt als erster die Schlange sobald er gezahlt hat.

Die Queue ist eine spezielle Form einer linearen Liste, in der der Elementzugriff nach dem FIFO (First-In-First-Out) Prinzip umgesetzt ist. Das FIFO-Prinzip besagt, dass jene Elemente die als erstes abgelegt wurden auch als erstes die Liste wieder verlassen. Sie können sich dieses Prinzip sehr gut mit der Funktionsweise einer Warteschlange an einer Supermarktkasse vorstellen. Jene Person die als erstes an die Kasse kommt, wird auch als erstes bedient. Sobald die Person dann die Schlange verlässt wird die zweite Person bedient usw..

Die Operation zum Einfügen eines neuen Elements an einer Queue wird auch als enqueue und die Entnahme des ersten Elements aus der Warteliste als dequeue bezeichnet.

Im Praxiseinsatz werden Queues insbesondere dann eingesetzt wenn viele Anfragen parallel einströmen und eine Bearbeitung aufgrund eines Engpasses an Ressourcen sequenziell erfolgen muss.

Datenstrukturen

Stack

- Ist eine Sonderform der verketteten Liste, die den **Elementzugriff nach dem LIFO (Last-In-First-Out)-Prinzip** umsetzt.
- **LIFO-Prinzip:** neues Element wird stets am Ende hinzugefügt und es wird stets das letzte Element entfernt.
- „**push**“ legt ein neues Element auf den Stack und „**pull**“ entfernt das oberste Element, „**peek**“ liest das oberste Element aus ohne es zu entfernen.
- Stacks werden häufig als Zwischenspeicher eingesetzt (bspw. Verlauf von Webbrowser)



Wie ein Tellerstapel an einem Buffet, werden neue Teller stets oben aufgelegt und die Kunden nehmen stets den Teller von oben weg.

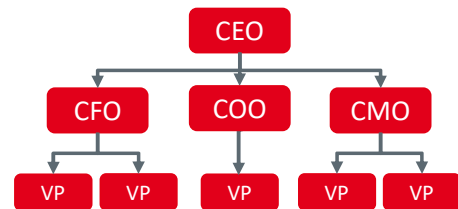
Der Stack ist eine weitere Sonderform einer linearen Liste, welche den Elementzugriff gemäß dem LIFO (Last-In-First-Out) Prinzip umsetzt. Bei diesem Zugriffsprinzip wird stets das zuletzt eingefügte Element der Liste entfernt. Ein Parallelen um sich dieses Zugriffsprinzip zu verinnerlichen sind Tellerstapel wie sie oft bei Buffets eingesetzt werden, bei denen stets die letzten dazugegebenen Teller als erstes an Gäste vergeben werden. Bei Verwendung eines Stacks wird die Operation mit der ein Element auf einen Stapel gelegt wird als „push“ bezeichnet und die Operation welche das letzte Element eines Stapels entfernt wird „pull“ genannt. Zusätzlich hierzu ermöglicht die Operation „peek“, dass das oberste Element angeschaut werden kann ohne es zu entfernen.

Die Stapel-Datenstruktur wird sehr oft als Zwischenspeicher für Anwendungen eingesetzt, beispielsweise zum Speichern des Verlaufs von Webbrowsern oder bei Java zum Auflösen von Variablen.

Datenstrukturen

Grundprinzip: Hierarchische Datenstruktur

- Hierarchische Datenstrukturen ordnen Elemente in einer **kontextspezifischen Rangordnung** an.
- Im Gegensatz zur linearen Datenstruktur können Elemente hier den **gleichen Rang besitzen** und **mehrere direkte Nachfolger** aufweisen.
- Hierarchische Datenstrukturen zeichnen sich dadurch aus, dass sie über eine Rangordnung die zu verwaltende **Komplexität reduzieren**.



Bsp. Organigramm einer Firma, jeder Bereich hat einen Verantwortlichen und ist selbst wieder unterteilt.

unterschied, keine sequentielle struktur, ein element kann mehrere nachfolger haben

Bei hierarchischen Datenstrukturen werden Elemente in einer hierarchischen und kontextspezifischen Rangordnung angeordnet. Diese hierarchische Rangordnung ist ein wichtiger Unterschied zu linearen Datenstrukturen, da hierdurch mehrere Elemente den gleichen Rang aufweisen können und ein Element auch mehrere direkte Nachfolger haben kann. Ein passendes Beispiel hierzu ist das Organigramm eines Unternehmens, bei dem jeder Bereich von einem eigenen Verantwortlichen geleitet wird. Hier unterstehen mehrere Arbeiter einem Bereichsleiter und der Bereichsleiter wiederum der Unternehmensführung.

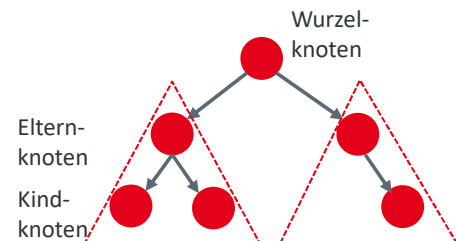
Ein wichtiger Vorteil von hierarchischen Datenstrukturen ist, dass sie die Gesamtkomplexität mit jeder Hierarchieebene in kleinere, voneinander getrennte, Teilbereiche unterteilen. Im Beispiel des Organigramms ist dies sichtbar indem ein Unternehmensbereich in Teilbereiche und diese wiederum in Aufgabengebiete unterteilt werden.

Datenstrukturen

Bäume: Grundlagen

ein element kann mehrere rollen umfassen
eltern sind zugleich kinder ihre großeltern

- Ein Baum beschreibt eine Datenstruktur, bei der Elemente **mehrere ausgehende aber nur eine eingehende Verbindung** besitzen.
- Die Klassifikation in **Eltern- und Kind** erfolgt stets **relativ** zu dem betrachteten Baumausschnitt.
- Ein Baum ist eine **rekursive Datenstruktur**, d.h. ein Baum besteht selbst aus Teilbäumen.
- Der oberste Knoten ist der **Wurzelknoten**.
- Knoten ohne Nachfolger werden als **Blätter** bezeichnet.



Ähnlich zum Stammbaum einer Familie, ist bei der Datenstruktur des Baum die Beziehung der Knoten relativ zueinander.

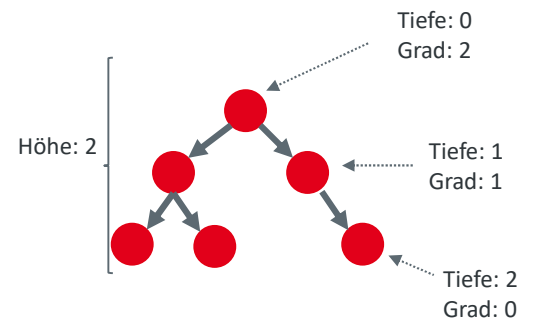
Eine der wichtigsten hierarchischen Datenstrukturen ist der Baum. Diese Datenstruktur hat ihren Namen daher, da ihre Struktur Ähnlichkeit zu einem auf den Kopf gedrehten realen Baum aufweist. Eine der wichtigsten Eigenschaften eines Baumes ist, dass hier ein Element mehrere ausgehende aber nur eine eingehende Verbindung aufweist. Wie bei einem Stammbaum erfolgt die Unterscheidung von Eltern- und Kindknoten stets relativ zu den direkten Vor- und Nachfolgern der Elemente. Der Baum ist eine rekursive Datenstruktur, das heißt der Baum selbst besteht wiederum aus mehreren Teilbäumen (bspw. einem linken und einem rechten). Jene Knoten die keinen direkten Nachfolger besitzen werden als Blätter des Baumes bezeichnet.

Ein typisches Einsatzgebiet der Baum Datenstruktur ist die Organisation des Dateisystems eines Computers.

Datenstrukturen

Bäume: Formale Aspekte

- Ein Baum hat insgesamt **$n-1$ Kanten**.
- Der **Grad** eines Knotens beschreibt die Anzahl seiner Kinder
- Die **Tiefe** eines Knotens bezeichnet den Abstand (Anzahl der Kanten) zur Wurzel.
- Die **Höhe** eines Baumes beschreibt die **maximale Tiefe** seiner Blätter.
- Knoten mit dem **gleichen Abstand zur Wurzel** befinden sich auf der gleichen **Ebene**.
- Ein Baum ist **balanciert**, wenn die Höhe des rechten und linken Teilbaums um höchstens 1 abweicht.

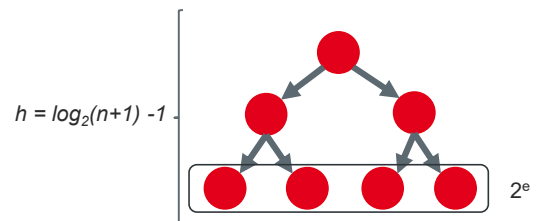


Schauen wir uns im Folgenden ein paar formale Aspekte von Bäumen an. Aufgrund dessen, dass Elemente in einem Baum stets nur einen Vorgänger besitzen, bestimmt sich die Gesamtzahl der Kanten in Abhängigkeit von der Anzahl der Elemente. Konkret gibt es in einem Baum $n-1$ Kanten, wobei n für die Gesamtanzahl der Elemente steht. Der **Grad** eines Knotens gibt an wie viele direkte Kinder dieser Elternknoten besitzt. Per Definition haben Blätter einen Grad von 0. Die Tiefe eines Knotens bestimmt sich durch die Gesamtanzahl an Kanten zwischen dem Knoten und der Wurzel des Baumes. Die Höhe eines Baumes ist die maximale Tiefe seiner Blätter. Knoten, welche die gleiche Tiefe aufweisen werden auch auf der gleichen Ebene bezeichnet. Abschließend bezeichnen wir einen Baum als balanciert, wenn sich die Tiefe seiner Blätter um maximal eine Höheneinheit unterscheidet.

Datenstrukturen

Binärbaum

- Der **Binärbaum** ist eine Sonderform eines Baumes bei dem jeder Knoten **maximal zwei Nachfolger** hat
- Die **Höhe (h)** eines vollständigen Binärbaums mit n Knoten berechnet sich via $h = \log_2(n+1) - 1$.
- Ein vollständiger Binärbaum der Höhe h umfasst **$2^{h+1}-1$ Knoten** *n herleiten*
- In einer Ebene (e) befinden sich **maximal 2^e Knoten**.

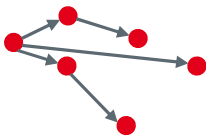


Der Binärbaum ist eine Spezialform eines Baumes bei der jeder Knoten maximal zwei Nachfolger besitzt. Aufgrund dieser Einschränkung ergeben sich verschiedene mathematische Schlussfolgerungen zur Höhe und Gesamtknotenanzahl. So lässt sich die Gesamthöhe eines vollständigen Binärbaumes über die Formel $h = \log_2(n+1) - 1$ berechnen, wobei h der Höhe und n der Gesamtknotenanzahl entspricht. Im Umkehrschluss gibt es in einem Binärbaum insgesamt $2^{h+1}-1$ Knoten und auf einer Ebene (e) befinden sich maximal 2^e Knoten.

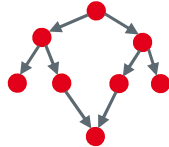
Aufgabe 3

Teilaufgabe a

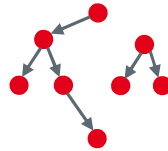
Handelt es sich bei den folgenden Skizzen um einen Baum?



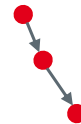
I.



II.



III.



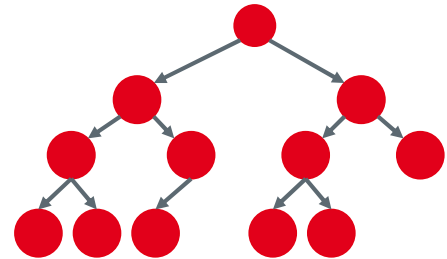
IV.



Aufgabe 3

Teilaufgabe b

- I. Ist der nebenstehende Baum balanciert? ✓
- II. Handelt es sich um einen Binärbaum? ✓
- III. Welche Höhe hat der Baum? 3
- IV. Markieren Sie jene Knoten welche eine Tiefe von 3 und einen Grad von 1 haben. keiner



Aufgabe 3

eine referenz ist ein verweis auf ein objekt

Teilaufgabe c

Implementieren Sie eine **Queue-Datenstruktur**. Mit folgenden Operationen:

- void enqueue(String value) – fügt ein neues Element am Ende der Queue ein.
- String dequeue() – entfernt das erste Element am Anfang der Queue und gibt es zurück
- int size() – gibt die aktuelle Anzahl an Elementen in der Queue zurück

Orientieren Sie sich hierzu an der vorgestellten Implementierung des Stack und nutzen Sie die **ListElem** Datenstruktur

Agenda

Algorithmen

Was ist ein Algorithmus und wie lässt sich die Güte dieser bewerten?

Datenstrukturen

Was sind zentrale Datenstrukturen und wie unterscheiden sie sich.

Laufzeitanalyse Datenstrukturen

Wie unterscheiden sich Datenstrukturen im Praxiseinsatz.

Laufzeitanalyse

Array

- **Feld-Zugriff:**

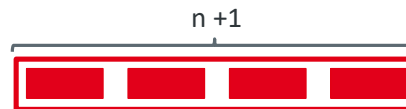
Jedes Feld kann über seinen Index, direkt angesprochen werden.



Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

- **Einfügen:**

Ein neuer Array muss angelegt werden in den die alten und der neue Wert eingefügt werden.



Best	Avg.	Worst
$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

- **Löschen:**

Ein neuer Array muss angelegt werden und die Werte, ohne das zu löschende Feld, kopiert werden.



Best	Avg.	Worst
$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

In diesem Kapitel untersuchen wir das asymptotische Zugriffsverhalten der zuvor besprochenen Datenstrukturen. Wir beginnen mit dem Array. Wie wir bereits gelernt haben, erfolgt beim Array ein indizierter Feldzugriff, das heißt jedes Feld kann gleich schnell angesprochen werden, genauso wie jede Seite eines Buches über ein Lesezeichen direkt angesprochen werden kann. Dadurch gilt für den Worst-, Avg-, und Best-Case ein konstantes Zugriffsverhalten gemäß Landau Notation.

Für das Einfügen eines neuen Elements gilt dagegen ein lineares Wachstum, da hierfür ein neuer Array angelegt wird in den die alten und der neue Wert hineinkopiert werden. Daraus folgt dass für jede Einfügeoperation bei einer Arraylänge von n insgesamt $n+1$ Kopierbefehle erfordert, sodass für den Worst-, Avg-, und Best-Case $\mathcal{O}(n)$ folgt.

Genauso wie beim Einfügen ist auch beim Löschen eines Feldes ein lineares Wachstum notwendig, da auch hier die alten Elemente kopiert werden müssen und somit für den Worst-, Avg-, und Best-Case $n-1$ Operationen erforderlich sind sodass $\mathcal{O}(n)$ folgt.

Laufzeitanalyse

Linked List

▪ Zugriff

Ausgehend vom Startknoten wird die Kette entlang der Referenzen gegangen.



Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

▪ Einfügen

An der entsprechenden Stelle wird die Referenz des Elements vor und nach der Einfügestelle angepasst



Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

▪ Löschen

An der entsprechenden Stelle wird die Referenz des zu entfernenden Elements angepasst.



Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Die nächste Datenstruktur die wir analysieren ist die verkettete Liste (Linked List). Im Gegensatz zum Array erfolgt hier ein sequenzieller Elementzugriff, das heißt es muss von Element zu Element gesprungen werden. Die Anzahl der benötigten Sprünge hängt von der Position des gesuchten Elements ab. Im günstigsten Fall ist das gesuchte Element das Anfangselement der Liste, sodass keine weiteren Sprünge erforderlich sind ($\mathcal{O}(1)$). Im durchschnittlichen Fall befindet sich das gesuchte Element in der Mitte also auf Position $n/2 \rightarrow \mathcal{O}(n)$ und im schlimmsten Fall am Ende der Liste ($\mathcal{O}(n)$).

Im Gegensatz zum Array erfordert die Einfügeoperation einen konstanten Aufwand im Best-, Avg.-, und Worst-Case da stets nur 2 Referenzen angepasst werden müssen.

Das Gleiche gilt für die Löschoption auch hier existiert durchweg ein konstanter Aufwand da stets nur 2 Referenzen angepasst werden müssen.

Laufzeitanalyse

Queue

best case: den ersten in der Schlange zurückgeben

▪ Zugriff

Um auf ein Element zuzugreifen, müssen alle Elemente davor entfernt werden.

Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

▪ Einfügen

Neue Elemente können nur am Ende eingefügt werden.



Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

▪ Löschen

Elemente können nur am Anfang entfernt werden

Best	Avg.	Worst
$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Wenden wir uns als nächstes der Queue zu. Für den Elementzugriff gilt dass im günstigsten Fall das gewünschte Element am Anfang der Warteschlange ist, sodass ein konstanter Aufwand benötigt wird $\mathcal{O}(1)$. Im durchschnittlichen Fall befindet sich das Element in der Mitte ($n/2$) und im schlimmsten Falle ganz am Ende der Schlange sodass in beiden Fällen $\mathcal{O}(n)$ gilt.

Ähnlich wie bei der verketteten Liste besteht bei der Queue ein konstanter Aufwand für das Einfügen neuer Elemente, wobei hier gilt, dass neue Elemente am Ende der Queue eingefügt werden (siehe FIFO-Prinzip). Das Gleiche gilt für das Entfernen von Elementen, da bei der Queue nur Elemente an der vordersten Position gelöscht werden.

Laufzeitanalyse

Stack

- Zugriff**

Es wird nur auf das letzte Element direkt zugegriffen, d.h. es müssen ggf. alle darüberliegenden Elemente durchlaufen werden.

Best	Avg.	Worst
$O(1)$	$O(n)$	$O(n)$

- Einfügen**

Neue Elemente können nur oben auf den Stapel hinzugefügt werden



Best	Avg.	Worst
$O(1)$	$O(1)$	$O(1)$

- Löschen**

Nur das oberste Element kann pro Operation entfernt werden

Best	Avg.	Worst
$O(1)$	$O(1)$	$O(1)$

Ähnlich wie bei der Queue gibt es auch beim Stack Ähnlichkeiten zur verketteten Liste. Der Zugriff auf die Elemente erfolgt auch hier sequenziell, wobei im besten Falle das gesuchte Element als letztes hinzugefügt wurde (vgl. LIFO-Prinzip) sodass hier ein konstanter Aufwand existiert und $O(1)$ folgt. Befindet sich das gesuchte Element jedoch in der Mitte (Avg.-Case) oder am Ende (Worst-Case) des Stack beträgt der Aufwand $O(n)$.

Das Einfügen von neuen Elementen am Ende des Stacks ist mit konstantem Aufwand verbunden, da diese stets am Ende eingefügt werden. Das gleiche gilt für das Entfernen von Elementen, da stets nur Elemente am Ende des Stacks entnommen werden können.

Laufzeitanalyse

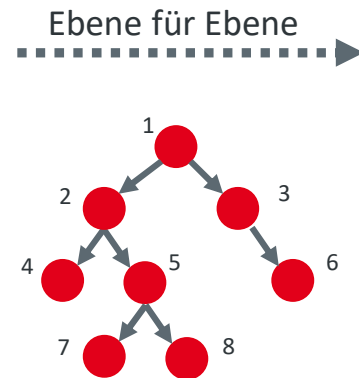
Binärbaum

▪ Zugriff

Es gibt **zwei grundlegend** unterschiedliche Arten für den Elementzugriff in Bäumen

- **Breitendurchlauf („breadth-first“)**

Der Algorithmus besucht die Knoten des Baumes der Breite nach geordnet, Zeile für Zeile, stets von links nach rechts.



Beim Breitendurchlauf durchläuft der Algorithmus den Baum Ebene für Ebene von links nach rechts.

Eine natürliche Rangordnung wie sie bei linearen Datenstrukturen besteht, gibt es bei Bäumen nicht. In Folge dessen gibt es mehrere Möglichkeiten einen Baum für den Elementzugriff zu durchlaufen. Im Folgenden schauen wir uns zwei grundlegende Arten für den Elementzugriff in einem Baum genauer an. Die erste Strategie ist der Breitendurchlauf. Hier wird der Baum Ebene für Ebene von links nach rechts durchlaufen. Die Grafik visualisiert die Reihenfolge in der die Knoten im Baum besucht werden.

Laufzeitanalyse

Binärbaum

- Zugriff**

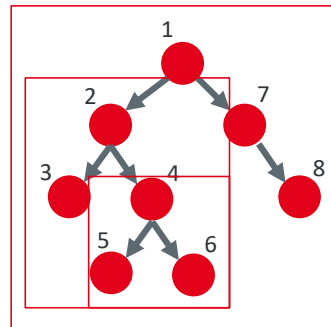
Die Anzahl der möglichen Felder reduziert sich mit jeder Hierarchieebene um ca. den Faktor 2

- Einfügen**

Im Falle, dass der Baum nach Einfügen balanciert werden muss, fallen Anpassungskosten an

- Entfernen**

Nach Entfernen eines Knoten können Anpassungen zur Balancierung anfallen



Best	Avg.	Worst
$O(1)$	$O(\log n)$	$O(n)$

Best	Avg.	Worst
$O(1)$	$O(\log n)$	$O(n)$

Best	Avg.	Worst
$O(1)$	$O(\log n)$	$O(n)$

Blicken wir zum Abschluss auf die Laufzeiteigenschaften des Binärbaums. Der Worst-Case hinsichtlich der Zugriffszeit, ist wenn der Binärbaum in einer listenähnlichen Struktur vorliegt. In einem derartigen Fall muss jedes Element wie bei einer verketteten Liste durchlaufen werden um am letzten Element anzukommen, hieraus resultiert $O(n)$ für den Elementzugriff. Beim Avg-Case ist der Baum balancierter, sodass mit jeder Ebene der Suchraum eingeschränkt werden kann. Sucht man bspw. die Position 5, würde man von der Wurzel kommend den linken Teilbaum besuchen da $5 < 7$ ist, als nächstes würde man den rechten Teilbaum besuchen, da $5 > 4$ ist und als nächstes könnte man schon auf das Element zugreifen. Das heißt mit jeder Ebene halbiert sich der Suchraum, sodass $O(\log n)$ besteht. Der Best-Case ist, dass das gesuchte Element an der Wurzelposition ist $O(1)$.

Hinsichtlich dem Einfügen sind im Worst-Case n Schritte notwendig, da komplette Teilbäume umgehängt und neu angeordnet werden müssen $O(n)$. Wie schon beim Zugriff zeigt sich jedoch auch hier im Avg-Case ein Laufzeitverhalten von $O(\log n)$. Im besten Fall ist der Baum im Vorfeld komplett balanciert sodass ein neues Element direkt angehängt werden kann.

Die gleiche Herleitung gilt für das Entfernen von Elementen. Im besten Falle, liegt bereits ein balancierter Baum vor, sodass die Entfernung eines Blattes keine Nachwirkungen hat. Im durchschnittlichen Falle jedoch führt das Entfernen von Elementen zu einem Aufwand von $O(\log n)$ und im schlimmsten Falle bei kompletter Umorganisation des Baumes zu einem Laufzeitverhalten von $O(n)$.

Laufzeitanalyse

Überblick

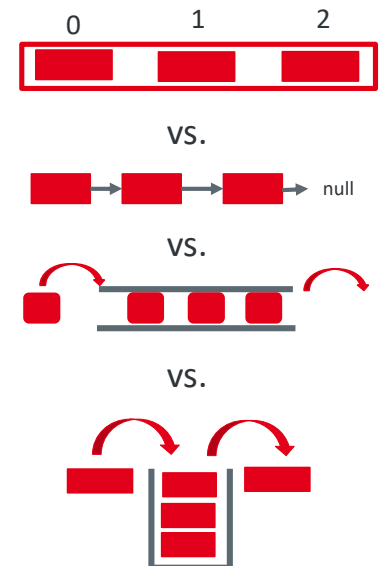
Daten- struktur	Zugriff (Worst)	Zugriff (Avg.)	Zugriff (Best)	Einfügen (Worst)	Einfügen (Avg.)	Einfügen (Best)	Löschen (Worst)	Löschen (Avg.)	Löschen (Best)
Array	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Linked- List	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Queue	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Stack	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Binär- baum	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Die folgende Tabelle stellt die einzelnen Laufzeiteigenschaften der untersuchten Datenstrukturen gegenüber.

Laufzeitanalyse

Überblick

- **Linked-List** sollte vor allem dann eingesetzt werden wenn **häufig Einfüge- und Löschoperationen** auf den Daten vorgenommen werden
- **Arrays** sollten verwendet werden, wenn **häufiger Zugriffsoperationen** als Einfüge- und Löschoperationen vorgenommen werden.
- **Binärbäume** sind die **zweite Wahl** bei Zugriffs- und Veränderungsoperationen.
- **Stack** und **Queue** sollten nur dann eingesetzt werden, wenn sie für den **Verwendungskontext** passen.



Aufbauend auf der Gegenüberstellung der letzten Folie sehen wir verschiedene Muster. Zum einen fällt auf, dass die verkettete Liste die präferierte Datenstruktur ist, wenn häufige Einfüge- und Löschoperationen vorgenommen werden, da diese Datenstruktur flexibel wachsen und schrumpfen kann. Der Nachteil der verketteten Liste ist das Zugriffsverhalten auf Elemente, genau in diesem Aspekt haben Arrays ihren Vorteil. Das heißt bei häufigen Zugriffsoperationen sollten Arrays verwendet werden. Wie wir gesehen haben stellen Binärbäume eine gute zweite Wahl zu Arrays bzgl. den Zugriffsoperationen und bzgl. Einfügeoperationen dar, doch aufgrund Ihres Worst-Case von $O(n)$ sollten sie nicht blind eingesetzt werden. Abschließend handelt es sich bei Stacks und Queues um Datenstrukturen die kontextspezifisch eingesetzt werden sollten.

Aufgabe 4

Laufzeitverhalten

- a) Beschreiben Sie 3 konkrete Einsatzszenarien für:
- a) Stack `strg c` `strg v` Strg Z, Verlauf im Browser
 - b) Queue Prozesse Anfragen für Buchungen, Anrufbeantworter, Instruktionsausführungen (Rezepte)
 - c) LinkedList Wagenreihung eines Zuges, Rundmail
 - d) Array Buch und Aktenregal, Sitzplätze eines Kinos, Arbeitsschichten an einem Tag

Aufgabe 4

tieftendurchlauf mit stack

- c) Konzipieren Sie einen Algorithmus zur Umsetzung des Breiten- und Tiefendurchlaufs auf Papier.

Überführen Sie anschließend Ihre Konzeption in Code und führen diesen aus.

Hinweis: Berücksichtigen Sie die Datenstrukturen Stack und Queue bei der Umsetzung der Durchlaufstrategien

