

# Inhaltsverzeichnis

<b>1 Grundlagen</b>	<b>3</b>
1.1 Einführung	3
1.1.1 insertionsort	3
1.1.2 Analyse der Laufzeit	4
1.2 Asymptotische Schranken	6
1.2.1 Direkte Definition der asymptotischen Schranken	6
1.2.2 Grenzwertkriterium	9
1.2.3 Definition der asymptotischen Schranken über den Limes	9
1.2.4 Rechenregeln für die $\mathcal{O}$ -Notation	10
1.2.5 Beispiele	11
1.2.6 Die gängigsten Wachstumsklassen	11
1.3 Elementare Datenstrukturen	12
1.3.1 Lineares Feld (Array)	13
1.3.2 Listen	13
1.3.3 Stack	15
1.3.4 Queue	16
1.3.5 Übersichtstabelle zu den asymptotischen Aufwänden	17
1.4 Rekursion	18
1.4.1 Die Fakultätsfunktion $n!$	18
1.4.2 Die Fibonacci-Zahlen $\text{fib}(n)$	19
1.4.3 Binärsuche	20
1.4.4 Divide et impera sowie der Hauptsatz der Laufzeitfunktionen	21
<b>2 Sortieralgorithmen</b>	<b>23</b>
2.1 Mergesort	23
2.2 Quicksort	26
2.2.1 Partition	26
2.2.2 Quicksort	27
2.2.3 Randomisierter Quicksort	29
2.3 Die untere Laufzeitschranke für vergleichsbasierte Sortieralgorithmen	32
2.3.1 Beweisskizze	32
2.4 Radixsort	34
2.5 Eigenschaften von Sortieralgorithmen	35
2.5.1 Laufzeit	35
2.5.2 worst-case-optimal	35
2.5.3 Stabilität	35

2.5.4	in-place . . . . .	35
2.5.5	Adaptivität . . . . .	35
2.5.6	Parallelisierbarkeit . . . . .	36
<b>3</b>	<b>Datenstrukturen</b>	<b>37</b>
3.1	Dynamische Arrays . . . . .	37
3.1.1	Operationen auf dynamischen Arrays . . . . .	37
3.1.2	Speicherverbrauch . . . . .	39
3.1.3	Amortisierte Laufzeitanalyse . . . . .	39
3.2	Hashtabellen . . . . .	43
3.2.1	Grundidee: Gestreute Speicherung . . . . .	43
3.2.2	Kollisionsbehandlung mit Überlauferliste (Chaining) . . . . .	44
3.2.3	Kollisionsbehandlung durch Offene Adressierung . . . . .	44
3.2.4	Echte Hashfunktionen . . . . .	47
3.2.5	Asymptotische Aufwände und Speicherbedarf . . . . .	48
3.3	Binärbäume . . . . .	49
3.3.1	Bäume . . . . .	49
3.3.2	Binärbaum . . . . .	52
3.3.3	Binäre Suchbäume . . . . .	53
3.4	Halden (Heaps) . . . . .	56
3.4.1	Verhalden ( <code>heapify</code> ) . . . . .	57
3.4.2	Aufbau einer Halde aus einem Array/Baum . . . . .	57
3.4.3	Sortieren mit Halden (Heapsort) . . . . .	58
3.4.4	Wartschlange (Priority Queue)* . . . . .	61
<b>A</b>	<b>Programmcode</b>	<b>62</b>
A.1	<code>quicksort</code> in Python . . . . .	62

# Kapitel 1

## Grundlagen

### 1.1 Einführung

#### 1.1.1 insertionsort

Eine ungeordnete Folge von Zahlen soll aufsteigend sortiert werden. Ein erster intuitiver Ansatz wäre, von links nach rechts gehend, jede einzelne Zahl so weit nach links zu verschieben, bis sie am richtigen Platz angekommen ist.

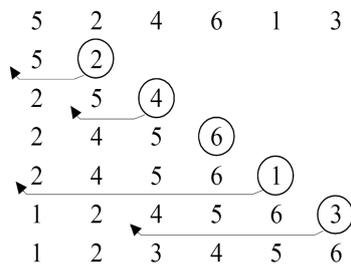


Abbildung 1.1: Ein Beispiellauf von `insertionsort`

Die verbale Formulierung kann auch in Form eines *Pseudocodes* ausgedrückt werden. Ein Pseudocode ist eine an höhere Programmiersprachen angelehnte Formulierung. Der vorgeschlagene Algorithmus wird in der Fachliteratur *insertion sort* (*Sortieren durch Einfügen*) genannt.

---

**Algorithmus 1** : `insertionsort(A)`

---

**Input** : A finite sequence  $A = [A_1, \dots, A_n]$  of integers

- [1] **for**  $i \leftarrow 2$  **to**  $n$  **do**
  - [2]      $h \leftarrow A_i$
  - [3]      $j \leftarrow i - 1$
  - [4]     **while**  $A_j > h$  **and**  $j > 0$  **do**
  - [5]          $A_{j+1} \leftarrow A_j$
  - [6]          $j \leftarrow j - 1$
  - [7]      $A_{j+1} \leftarrow h$
-

### 1.1.2 Analyse der Laufzeit

Wir suchen nun die Abhängigkeit der Laufzeit  $T$  von der Inputgröße  $n$  (Anzahl der zu sortierenden Zahlen). Im obigen Zahlenbeispiel ist  $n = 6$ .

Die Grundidee ist einfach: Wir zählen die einzelnen Schritte, multiplizieren sie mit konstanten Zeitfaktoren  $c_i$  (als Platzhalter für die Zeit die sie benötigen) und versuchen sie in Abhängigkeit der Inputgröße  $n$  darzustellen. Die Zeitfaktoren  $c_i$  sind je nach Implementierung (Sprache, Hardware, etc.) unterschiedlich. Sie können aber dann immer als konstant angesehen werden. Für unseren ersten Algorithmus ergibt sich also:

Pseudocode	Zeitkonstante	Wiederholungen
<b>insertionsort(A)</b>	$c_0$	1
<b>for</b> $i \leftarrow 2$ <b>to</b> $n$ <b>do</b>	$c_1$	$n$
$h \leftarrow A_i$	$c_2$	$(n - 1)$
$j \leftarrow i - 1$	$c_3$	$(n - 1)$
<b>while</b> $A_j > h$ <b>and</b> $j > 0$ <b>do</b>	$c_4$	$\sum_{i=2}^n t_i$
$A_{j+1} \leftarrow A_j$	$c_5$	$\sum_{i=2}^n (t_i - 1)$
$j \leftarrow j - 1$	$c_6$	$\sum_{i=2}^n (t_i - 1)$
$A_{j+1} \leftarrow h$	$c_7$	$n - 1$

Anmerkung: Die Zeit um den Algorithmus aufzurufen wird als unabhängig von der Inputgröße angenommen. In den zukünftigen Betrachtungen wird sie deshalb nie berücksichtigt. Als optisches Kennzeichen trägt der Algorithmuskopf daher auch keine eigene Zeilennummer.

Bemerkung zu den einzelnen Zeilen:

1. In der Schleife werden die einzelnen Elemente abgearbeitet. Pro Schleife wird geschaut, ob der Index  $i$  schon  $n$  erreicht hat oder nicht. Falls nicht wird  $i$  um eins erhöht. Benötigte Zeit:  $c_1$ . Der Body der *for*-Schleife wird  $(n - 1)$  mal ausgeführt ( $i$  läuft von 2 bis  $n$ ). Der Schleifenkopf wird einmal öfter ausgeführt als der Body, da die Abbruchbedingung noch einmal überprüft werden muss. Das gilt auch für die *while*-Schleife in Zeile 4.
2. Die Zuweisung benötigt eine konstante Zeit  $c_2$ .
3. Ebenfalls eine Zuweisung mit einer Subtraktion:  $c_3$
4. Die Konstante  $c_4$  beinhaltet die beiden Vergleiche als auch deren logische Verknüpfung. Wenn man sich ein Zahlenbeispiel genauer anschaut, sieht man, dass es von der Verteilung der Zahlen abhängt, wie oft man die *while*-Schleife durchlaufen und damit die Abfrage durchführen muss. Man definiert eine allgemeine Variable  $t_i$ :  $t_i$  ist die Anzahl der *while*-Abfragen, die der Algorithmus für das  $i$ -te Element des Inputs machen muss.
5. sowie 6. und 7. sind ebenfalls Zuweisungen:  $c_5, c_6$  und  $c_7$

Wenn man nun einfach alle Zeiten zusammenzählt, erhält man die Laufzeit  $T$  in Abhängigkeit der Inputgröße  $n$ :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^n t_i + c_5 \sum_{i=2}^n (t_i - 1) + c_6 \sum_{i=2}^n (t_i - 1) + c_7(n-1)$$

Offensichtlich hängt die Laufzeit  $T$  nicht nur von der Inputgröße  $n$  und den Zeitkonstanten  $c_i$  ab, sondern auch davon wie die Zahlen in der ungeordneten Folge  $A$  vorliegen (= Einfluss von  $t_i$ ). Dadurch kann man drei wichtige Fälle unterscheiden: der *beste Fall* (*best case*), der *schlechteste Fall* (*worst case*) und der *durchschnittliche Fall* (*average case*).

### Bester Fall (best case)

Der beste (schnellste) Fall liegt vor, wenn der Algorithmus *nie* in die **while**-Schleife hineingehen muss, die Zahlen also bereits sortiert vorliegen. Das heißt, die logische Abfrage  $A[j] > h$  and  $j > 0$  in Zeile 5 wird jeweils nur einmal pro Element durchgeführt:  $t_i = 1$  für alle  $i = 2, 3, \dots, n$ .

Mit  $\sum_{i=2}^n t_i = \sum_{i=2}^n 1 = (n-1)$  ergibt sich für die Laufzeit:

$$T_{\text{best}}(n) = n \cdot (c_1 + c_2 + c_3 + c_4 + c_7) - (c_2 + c_3 + c_4 + c_7) = c_{\text{lin}} n + c_{\text{const}}$$

Das heißt, die Laufzeit nimmt *linear* mit der Inputgröße zu.

### Schlechtester Fall (worst case)

Der schlechteste (langsamste) Fall liegt vor, wenn der Algorithmus das  $i$ -te Element um  $i-1$  Plätze nach vorne verschieben muss, die Sequenz also bereits andersrum sortiert ist. Das heißt:  $t_i = i$  für alle  $i = 2, 3, \dots, n$ . Die Laufzeit beträgt:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \cdot \sum_{i=2}^n i + c_5 \cdot \sum_{i=2}^n (i-1) + c_6 \cdot \sum_{i=2}^n (i-1) + c_7(n-1)$$

Mit  $\sum_{i=2}^n i = \frac{n \cdot (n+1)}{2} - 1$  und  $\sum_{i=2}^n (i-1) = \frac{n \cdot (n-1)}{2}$  erhält man folgende Form:

$$T_{\text{worst}}(n) = c_{\text{quad}} n^2 + c_{\text{lin}} n + c_{\text{rest}}$$

Das heißt, die Laufzeit nimmt *quadratisch* mit der Inputgröße zu.

### Durchschnittlicher Fall (average case)

Alle  $n$  Zahlen sind zufällig verteilt (gleichverteilt). Im Durchschnitt ist die Hälfte der Elemente in  $A_{1, \dots, i-1}$  größer als das  $i$ -te Element. Das heißt  $t_i \approx \frac{i}{2}$ . Mit

$$T_{\text{avg}}(n) = \sum_{i=2}^n \left(\frac{i}{2}\right) = \frac{1}{2} \sum_{i=2}^n (i) = -\frac{1}{2} + \frac{n \cdot (n+1)}{4}$$

erhält man wieder eine *quadratische* Abhängigkeit der Laufzeit von der Inputgröße.

## 1.2 Asymptotische Schranken

Wir haben im letzten Kapitel gesehen, dass die konkrete Laufzeit eines Programms stark abhängig ist von der konkreten Hardware (ausgedrückt durch die Parameter  $c_1, \dots, c_7$ ). Zudem können Formeln und ihre Analyse unhandlich und kompliziert werden. Daher werden sowohl die Laufzeit  $T(n)$  eines Programms als auch der Speicherbedarf  $S(n)$  meist durch *asymptotische Schranken* angegeben.

Wir schließen das Kapitel mit einer formalen Definition, wie wir die Laufzeit (und Speicherverbrauch) eines Programms in Abhängigkeit von der Inputlänge  $n$  beschreiben:

**Definition 1** (Laufzeit  $T(n)$  und Speicherverbrauch  $S(n)$ ). Sei  $I$  der Input für unseren Algorithmus,  $|I| < \infty$  die Größe des Inputs. Sei  $T(I) \in \mathbb{R}^+$  die Laufzeit, die unser Algorithmus für Input  $I$  benötigt. Dann definieren wir

- $T_{\text{worst}}(n) = \max\{T(I) : |I| = n\}$  ist die *worst case* Laufzeit.
- $T_{\text{best}}(n) = \min\{T(I) : |I| = n\}$  ist die *best case* Laufzeit.
- $T_{\text{avg}}(n) = \frac{1}{|I_n|} \sum_{I \in I_n} T(I)$  wobei  $I_n = \{I : |I| = n\}$  ist die *average case* Laufzeit.

Sei analog  $S(I)$  der Speicherverbrauch des Algorithmus mit Input  $I$ . Dann definieren wir  $S_{\text{worst}}$ ,  $S_{\text{best}}$  und  $S_{\text{avg}}$  analog.

Gelegentlich verwenden wir einfach nur  $T(n)$ , insbesondere dann, wenn die Laufzeit nur von der Inputlänge, nicht vom konkreten Input abhängt (beispielsweise  $T(n) = 5n^2$ ).

Die Grundidee für asymptotische Schranken ist, dass bei einem immer weiter wachsendem Input von Länge  $n$  die Konstanten vernachlässigbar werden. Desweiteren können Terme mit niedriger Ordnung als verschwindend angesehen werden.

Ein Beispiel: Unser Programm `insertionsort` hatte im average case die Laufzeit  $T_{\text{avg}}^{\text{is}}(n) = a_2 n^2 + a_1 n + a_0$  für gewisse  $a_0, a_1, a_2 \in \mathbb{R}^+$ . Der Zugänglichkeit zuliebe wurde in dem Beispiel  $T_{\text{avg}}^{\text{is}}(n)$  nicht korrekt nach unserer Definition berechnet. Nun kann man, bei sehr großem  $n$ , sowohl die Konstanten  $a_0$  und  $a_2$  als auch den Term  $a_1 n$  vernachlässigen.

Mathematisch präzisiert wird das in der sogenannten  $\mathcal{O}$ -,  $\Theta$ - und  $\Omega$ -Notation.

### 1.2.1 Direkte Definition der asymptotischen Schranken

Wir betrachten grundsätzlich Funktionen  $f(n), g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ . Eine Funktion  $g(n)$  ist eine asymptotisch obere Schranke für  $f(n)$ , wenn eine positive Konstante  $c$  und eine natürliche Zahl  $n_0$  existieren, sodass ab diesem  $n_0$  die Funktion  $f(n)$  für alle  $n > n_0$  kleiner als  $c \cdot g(n)$  bleibt.

**Definition 2** (Asymptotisch obere Schranke:  $\mathcal{O}$ -Notation). Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  zwei positive Funktionen. Wir definieren die Menge der von  $g$  asymptotisch nach oben beschränkten Funktionen

$$\mathcal{O}(g) := \{f \mid \exists c \in \mathbb{R}^+, \exists n_o \in \mathbb{N}, \forall n > n_o : f(n) \leq cg(n)\}.$$

Wir sagen wahlweise  $f$  liegt in  $\mathcal{O}$  von  $g$ ,  $f$  ist asymptotisch von  $g$  nach oben beschränkt, etc., aber meinen damit stets  $f \in \mathcal{O}(g)$ . Es wird oftmals auch die Notation  $f = \mathcal{O}(g) :\Leftrightarrow f \in \mathcal{O}(g)$  benutzt.

Analog dazu die untere Schranke: Wieder müssen  $c \in \mathbb{R}^+$  und  $n_0 \in \mathbb{N}$  existieren, sodass für alle fortfolgenden  $n \in \mathbb{N}$   $f(n)$  stets *größer* ist  $cg(n)$ .

**Definition 3** (Asymptotisch untere Schranke:  $\Omega$ -Notation). Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  zwei positive Funktionen. Wir definieren die Menge der von  $g$  asymptotisch nach unten beschränkten Funktionen

$$\Omega(g) := \{f \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n > n_0 : f(n) \geq cg(n)\}.$$

Wir sagen wahlweise  $f$  liegt in  $\Omega$  von  $g$ ,  $f$  ist asymptotisch von  $g$  nach unten beschränkt, etc., aber meinen damit stets  $f \in \Omega(g)$ . Es wird oftmals auch die Notation  $f = \Omega(g) :\Leftrightarrow f \in \Omega(g)$  benutzt.

Die Definition der beidseitigen Schranke überrascht nun nichtmehr:

**Definition 4** (Asymptotisch exakte Schranke:  $\Theta$ -Notation). Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  zwei positive Funktionen. Wir definieren die Menge der von  $g$  asymptotisch exakt beschränkten Funktionen

$$\Theta(g) := \{f \mid \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n > n_0 : c_1g(n) \leq f(n) \leq c_2g(n)\}.$$

Wir sagen wahlweise  $f$  liegt in  $\Theta$  von  $g$ ,  $f$  ist asymptotisch (exakt) von  $g$  beschränkt, etc., aber meinen damit stets  $f \in \Theta(g)$ . Es wird oftmals auch die Notation  $f = \Theta(g) :\Leftrightarrow f \in \Theta(g)$  benutzt.

Man sieht direkt, dass

$$f \in \Theta(g) \Leftrightarrow f \in \mathcal{O}(g) \text{ und } f \in \Omega(g).$$

Wir können nun direkt mit dieser Definition den Asymptotischen Aufwand von  $T_{\text{avg}}^{\text{is}}(n)$  klassifizieren:

**Beispiel 1** ( $T_{\text{avg}}^{\text{is}} \in \Theta(n^2)$ ). Zuerst zeigen wir, dass  $T_{\text{avg}}^{\text{is}} = a_2n^2 + a_1n + a_0 \in \mathcal{O}(n^2)$ :

Gesucht werden  $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  sodass für alle  $n > n_0$ :

$$a_2n^2 + a_1n + a_0 \leq cn^2.$$

Wähle mit  $m = \max(\{a_0, a_1, a_2\})$  die Konstante  $c = 3m$ . Dann gilt für  $n > 1$ :

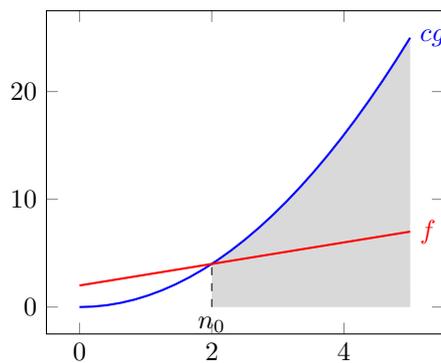
$$\begin{aligned} cn^2 &= mn^2 + mn^2 + mn^2 \\ &\geq a_2n^2 + a_1n + a_0, \end{aligned}$$

da  $n^2 \geq n$  und  $n^2 \geq 1$  für alle  $n > 1$ .

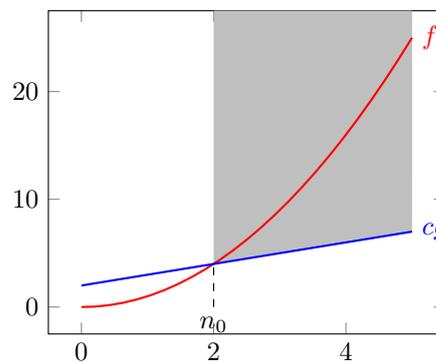
Für  $T_{\text{avg}}^{\text{is}} \in \Omega(n^2)$  nehmen wir noch an, dass  $a_0, a_1, a_2 > 0$ . Dann ist mit  $c = a_2$  klar:

$$cn^2 \leq a_2n^2 + a_1n + a_0 \text{ für alle } n \geq 0.$$

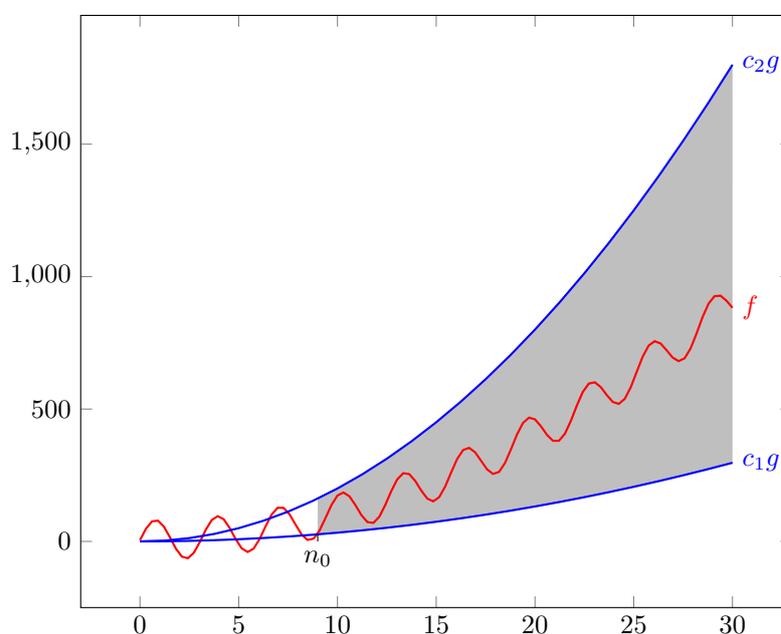
Auch ohne die Einschränkung  $a_0, a_1, a_2 > 0$  gilt  $T_{\text{avg}}^{\text{is}} \in \Theta(n^2)$ , aber es ist eine für Laufzeitfunktionen sinnvolle Einschränkung.



(a)  $f \in \mathcal{O}(g)$ , mit  $n_0 = 2$



(b)  $f \in \Omega(g)$ , mit  $n_0 = 2$



(c)  $f \in \Theta(g)$ , wobei  $g(n) = n^2$ ,  $c_1 = 0.3$ ,  $c_2 = 2$  und  $f(n) = n^2 + 75 \sin(2n)$ . Ab  $n_0 = 9$  wird  $f$  von  $c_1g$  und  $c_2g$  abgeschätzt.

**Beispiel 2** ( $6n^3 \notin \mathcal{O}(n^2)$ ). Als nächstes schauen wir, ob  $6n^3$  asymptotisch langsamer wächst als  $n^2$ . Gehen wir also davon aus, dass es  $c \in \mathbb{R}^+$  und  $n_0 \in \mathbb{N}$  gäbe, sodass  $6n^3 \leq cn^2$  für alle  $n > n_0$ . Dann können wir die Ungleichung aber mit  $\frac{1}{6n^2}$  multiplizieren und hätten  $n \leq \frac{c}{6}$  für alle  $n > n_0$ . Da  $c$  eine Konstante ist, ist dies ein Widerspruch.

Betrachten wir nun ein spannenderes Beispiel: Ist  $\ln(n) \in \mathcal{O}(\sqrt{n})$ ? Mit unserer Quantorendefinition kommen wir erstmal nicht weiter. Aber gefühlt wächst  $\ln$  langsamer! Wir können ja die kontinuierlichen Varianten  $\ln(x)$  und  $\sqrt{x}$  betrachten, die die Ableitung  $\frac{1}{x}$  respektive  $\frac{1}{2\sqrt{x}}$  haben. Anscheinend wächst der Logarithmus ja wirklich bedeutend langsamer! Das Grenzwertkriterium (und

die Regel von L'Hôpital) hilft uns dabei enorm:

### 1.2.2 Grenzwertkriterium

Oftmals ist Nachweis gewissen asymptotischen Verhaltens auf diese direkte Art recht umständlich. Existiert der Grenzwert  $\frac{f(n)}{g(n)}$  im Infinitesimalen, so können wir das Grenzwertverhalten direkt ablesen:

**Lemma 1.** *Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  zwei positive Funktionen. Existiert der Grenzwert  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = K$ , so gilt:*

- Ist  $0 \leq K < \infty$ , so folgt  $f \in \mathcal{O}(g)$
- Ist  $0 < K \leq \infty$ , so folgt  $f \in \Omega(g)$
- Ist  $0 < K < \infty$ , so folgt  $f \in \Theta(g)$

Damit können wir das Logarithmusbeispiel lösen:

**Beispiel 3** ( $\ln(n) \in \mathcal{O}(\sqrt{n})$ ). Wir betrachten weiterhin die kontinuierlichen Versionen. Dann gilt:

$$\lim_{x \rightarrow \infty} \frac{\ln(x)}{\sqrt{x}} = \frac{\infty}{\infty},$$

wir müssen uns also um die Regel von L'Hôpital bemühen. Nach dieser existiert obiger Limes, solange der Limes der Ableitungen existiert.

$$\lim_{x \rightarrow \infty} \frac{\ln(x)}{\sqrt{x}} = \lim_{x \rightarrow \infty} \frac{\frac{\partial \ln(x)}{\partial x}}{\frac{\partial \sqrt{x}}{\partial x}} = \lim_{x \rightarrow \infty} \frac{2\sqrt{x}}{x} = \lim_{x \rightarrow \infty} \frac{2}{\sqrt{x}} = 0.$$

Zu beachten ist, dass die Umkehrung nicht unbedingt gilt, z.B.,  $f(n) = \sin(\frac{n}{100}) + 1$ . Es gilt  $f(n) = \mathcal{O}(1)$ , aber der Grenzwert  $\lim_{n \rightarrow \infty} \frac{f(n)}{1}$  existiert nicht. Das ist die Motivation für die folgende Alternativdefinition:

### 1.2.3 Definition der asymptotischen Schranken über den Limes

Wir erinnern uns an die Definition des Limes superior bzw Limes inferior:

**Definition 5** (Limes superior und Limes inferior). Sei  $g : \mathbb{N} \rightarrow \mathbb{R}^+$  eine positive Funktion. Wir definieren

$$\limsup_{n \rightarrow \infty} g(n) := \inf_{n_0 \in \mathbb{N}} \sup_{n \geq n_0} g(n)$$

und analog

$$\liminf_{n \rightarrow \infty} g(n) := \sup_{n_0 \in \mathbb{N}} \inf_{n \geq n_0} g(n)$$

Für uns der größte Vorteil ist, dass unter den gegebenen Umständen sowohl der Limes superior als auch der Limes inferior stets existieren und Werte in  $\mathbb{R}^+ \cup \{\infty\}$  annehmen.

Damit können wir nicht nur Kriterien für asymptotisches Wachstum benennen, sondern sogar Äquivalenzen.

**Satz 1.** Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$  positive reelle Funktionen. Es gilt

- $f \in \mathcal{O}(g) \Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$ ,
- $f \in \Omega(g) \Leftrightarrow \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$ ,
- $f \in \Theta(g) \Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$  und  $\liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$ .

### 1.2.4 Rechenregeln für die $\mathcal{O}$ -Notation

**Lemma 2.** Seien  $f, f', g, g', h : \mathbb{N} \rightarrow \mathbb{R}^+$ .

*Addition* Seien  $f \in \mathcal{O}(g)$  und  $f' \in \mathcal{O}(g')$ . Dann ist  $f + f' \in \mathcal{O}(g) \cup \mathcal{O}(g')$ .

*Subtraktion* Seien  $f, f' \in \mathcal{O}(g)$ . Dann gilt  $f - f' \in \mathcal{O}(g)$ .

*Multiplikation* Seien  $f \in \mathcal{O}(g)$  und  $f' \in \mathcal{O}(g')$ . Dann gilt  $ff' \in \mathcal{O}(gg')$ .

*Transitivität* Seien  $f \in \mathcal{O}(g)$  und  $g \in \mathcal{O}(h)$ . Dann ist  $f \in \mathcal{O}(h)$ .

Zur Additionsregel sei angemerkt, dass sie durch die Transitivitätsregel meist weiter vereinfachen lässt. Wachsen  $g$  und  $g'$  gleich schnell, also  $g \in \Theta(g')$ , so ist  $\mathcal{O}(g) = \mathcal{O}(g')$ . Wächst o.B.d.A.  $g'$  stärker als  $g$ , also  $g \in \mathcal{O}(g')$ , so ist  $\mathcal{O}(g) \cup \mathcal{O}(g') = \mathcal{O}(g')$ . Nur für den seltenen Fall, das  $g \notin \mathcal{O}(g')$  und  $g' \notin \mathcal{O}(g)$  bleibt es bei der Meingenvereinigung.

*Beweis.* Wir betrachten zur Veranschaulichung die Transitivität genauer. Nach Voraussetzung gibt es  $c, c' \in \mathbb{R}^+$  und  $n_0, n'_0 \in \mathbb{N}$ , sodass  $f \leq cg(n)$  für alle  $n > n_0$  und  $g \leq c'h(n)$  für alle  $n > n'_0$ . Wir können also mit  $N_0 := \max(n_0, n'_0)$  und  $C := cc'$  zeigen:  $f \leq cg(n) \leq Ch(n)$  für alle  $n > N_0$ .

Die anderen Rechenregeln verlaufen meist analog und sind eine gute Fingerübung.  $\square$

**Lemma 3.** Die Basis des Logarithmus ist für asymptotische Betrachtung nicht interessant. Seien  $b, b' \in \mathbb{R}^+$ , dann gilt:

$$\log_b \in \Theta(\log_{b'})$$

Insbesondere gilt für den häufig vorkommenden Logarithmus zur Basis 2:

$$\log_2 \in \Theta(\ln)$$

*Beweis.* Es gilt

$$\log_{b'}(n) = \frac{1}{\log_b(b')} \cdot \log_b(n),$$

also unterscheiden sie sich nur um den konstanten Faktor  $\frac{1}{\log_b(b')}$ , welcher asymptotisch gesehen vernachlässigbar ist.  $\square$

### 1.2.5 Beispiele

Wir betrachten nun einige Beispiele, die die Relation gängiger Wachstumskategorien im Beispiel belegen:

**Beispiel 4** (Polynomielles versus exponentielles Wachstum). Wir zeigen  $n^a \in \mathcal{O}(b^n)$  für  $1 < b \in \mathbb{R}$ ,  $a > 0$ . Der Einfachheit halber nehmen wir  $a \in \mathbb{N}$  an.

Beim Betrachten des Limes des kontinuierlichen Falles  $\lim_{x \rightarrow \infty} \frac{x^a}{b^x}$  stellen wir schnell fest, dass wir  $a$ -mal die Regel von L'Hôpital anwenden müssen. Mit  $\frac{\partial^a x^a}{\partial x^a} = a!$  und  $\frac{\partial^a b^x}{\partial x^a} = b^x \log(b)^a$  sehen wir:

$$\lim_{x \rightarrow \infty} \frac{x^a}{b^x} = \lim_{x \rightarrow \infty} \frac{a!}{b^x \log(b)^a} = 0.$$

Damit gilt auch für die diskrete Variante  $n^a \in \mathcal{O}(b^n)$ . Exponentielles Wachstum trumpft also polynomielles Wachstum.

**Beispiel 5** (Exponentielles Wachstum versus die Fakultätsfunktion). Wir betrachten als Beispiel  $2^n \in \mathcal{O}(n!)$ . Dies geschieht wieder über die Quantorendefinition: Wir wählen  $c = 1$  und  $n_0 = 4$ . Damit haben wir

$$2^n \leq n! \text{ für alle } n > n_0.$$

Warum? Wir benutzen eine natürliche Induktion. Basisfall ist  $n = 4$ , dann gilt:  $2^4 = 16 \leq 24 = 4!$ . Im Induktionsschritt  $n \mapsto n + 1$ :

$$2^{n+1} = 2 \cdot 2^n \leq (n + 1) \cdot (n!) = (n + 1)!,$$

da  $(n + 1) > 2$  für  $n \geq 1$  und  $2^n \leq n!$  durch die Induktionsvoraussetzung.

### 1.2.6 Die gängigsten Wachstumsklassen

Hier ist eine Auflistung der gebräuchlichsten Wachstumsklassen im Kontext der Informatik. Die Liste ist aufsteigend gestaltet, also  $\mathcal{O}(1) \subseteq \mathcal{O}(\log(n)) \subseteq \dots \subseteq \mathcal{O}(n!)$ .

Name	$\mathcal{O}$ -Notation	Bsp.-Funktion	Bsp.-Algorithmus
konstant	$\mathcal{O}(1)$	3	Addition
logarithmisch	$\mathcal{O}(\log(n))$	$\log_2(n), \ln(n)$	Suchen
Wurzelfunktion	$\mathcal{O}(n^c), 0 < c < 1$	$\sqrt{n}, n^{\frac{1}{3}}$	Primzahltest
linear	$\mathcal{O}(n)$	$3n + 4$	Maximum finden
linearlogarithmisch	$\mathcal{O}(n \cdot \log(n))$	$2n \cdot \ln(n)$	Sortieren
polynomial	$\mathcal{O}(n^c), c > 1$	$4n^2 + 7n + 10$	Matrizenoperationen
exponential	$\mathcal{O}(c^n), c > 1$	$2^n, 10^n$	NP-vollständige Alg.
Fakultät	$\mathcal{O}(n!)$	$n!$	

### 1.3 Elementare Datenstrukturen

Was ist eine Datenstruktur? Diesen Begriff mathematisch präzise zu fassen ist sehr schwer. Wir überlassen das daher anderen Vorlesungen und begnügen uns mit einer unpräziseren Definition: Datenstrukturen sind *Daten* mit einer *Struktur*.

Daten meint dabei sowohl elementare Datentypen wie `int`, `float`, `char`, aber auch komplexere oder abstraktere Gebilde wie die Repräsentation eines Gegenstandes oder  $x \in \mathbb{R}$ .

Struktur kann viel bedeuten. In diesem Kapitel befassen wir uns hauptsächlich mit Strukturen, die eine lineare Ordnung ermöglichen, die uns also erlauben zu sagen, in welcher “eindeutigen Reihenfolge” sich die Daten befinden. Datenstrukturen, in denen mehr als ein Nachfolger erlaubt ist, behandeln wir in späteren Kapiteln.

**Die Menge: Eine Datenstruktur fast ohne Struktur** Ein kurzes, einführendes Beispiel betrachtet aber eine Datenstruktur mit noch viel weniger Struktur, die sogenannte Menge (set). Ganz wie die naive Definition einer mathematischen Menge ist das eine “Zusammenfassung bestimmter, wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens zu einem Ganzen” (Cantor).

Vorstellen kann man es sich als Sack, in welchem verschiedene Dinge sind. Die einzige Struktur, die die Menge auferlegt, ist die Tatsache, das ein Element nicht mehrfach vorkommen darf (das wäre dann ein Multiset). Ansonsten haben wir keinerlei Reihenfolge oder sonstige Struktur. Das Entfernen eines Elements geschieht in konstanter Zeit ( $\mathcal{O}(1)$ ), das Suchen und Einfügen muss durch elementweises Vergleichen geschehen ( $\mathcal{O}(n)$ , wobei  $n$  die Anzahl der Elemente ist). Sortieren ist schlicht nicht möglich.

**Datenstrukturen mit einer Reihenfolge** Im Folgenden betrachten wir die vier bekanntesten Datenstrukturen, die uns zumindest eine Reihenfolge geben. Das mathematische Äquivalent wäre also nicht mehr die Menge, sondern ein Tupel  $(d_0, d_1, d_2, \dots)$  mit  $d_i$  als das Nutzdatum an der  $i$ -ten Stelle. Damit einher geht die Einführung des Index  $i \in \mathbb{N}_0$ , wir fangen also (um Verwirrung beim Programmieren vorzubeugen) mit 0 an zu zählen. Dieser Index erlaubt uns, den Index  $i$  (manchmal auch Schlüssel) und die Daten der Datenstruktur  $\mathcal{D}$  zu unterscheiden. Die Daten der Datenstruktur an der Stelle  $i$  nennen wir  $\mathcal{D}[i]$ .

Wir werden zusätzlich Implementierungen dieser Datenstrukturen angeben. Diese sind in imperativer Perspektive für ein Maschinenmodell mit Random-Access-Memory (RAM)  $\mathcal{M}$  zu verstehen. Die definierende Eigenschaft dieses Maschinenmodells ist, dass der Zugriff  $\mathcal{M}[i]$  auf ein Speicherfeld stets gleich lange braucht, egal wo es liegt. Der Einfachheit halber nehmen wir zudem an, unendlich viel Speicher zu haben, also  $i \in \mathbb{N}$ .

**Notation** In der Hoffnung, die folgende Mischung aus mathematischer Notation, Pseudocode und Erklärungen verständlicher zu gestalten, benutzen wir folgende notationelle Konventionen: Indexe  $i$ , sowie Pointer auf Speicheradressen  $p$  und (für uns abstrakte) Nutzlast  $d$  wird in mathematischem *italic* geschrieben. Datenstrukturen wie beispielsweise das Array  $\mathcal{A}$  oder die Liste  $\mathcal{L}$  werden

in  $\mathcal{M}$  gesetzt, auch der RAM  $\mathcal{M}$  gehört notationell zu den Speicherstrukturen. Systemfunktionen wie `reference_of` oder sehr triviale zu implementierende Funktionen wie `len` für die Länge von Speicherstrukturen werden in `true_type` gesetzt, um ihre Maschinennähe darzustellen. Ebenso bekommen all unsere in Psudeocode notierten Algorithmen diese Notation.

### 1.3.1 Lineares Feld (Array)

Das Array  $\mathcal{A}$  ist eine Datenstruktur von fixer Größe  $n \in \mathbb{N}$ . Aus logischer Perspektive stehen benachbarte Elemente direkt nebeneinander. Das Array  $\mathcal{A}$  von Größe  $n$  ist also nichts weiter als  $\mathcal{M}[s, \dots, s + n - 1]$ , wobei  $s \in \mathbb{N}$  ein gewisser Versatz ist. Durch die Natur des RAM ist Lese- oder Schreibzugriff auf das  $i$ -te Feld in konstanter Zeit  $\mathcal{O}(1)$  möglich. Das ist das Alleinstellungsmerkmal des Arrays!

Es gibt aber keine kanonische Möglichkeit, die Größe des Arrays zu verändern, also Elemente hinzuzufügen oder zu entfernen. Das sogenannte *dynamische Array* bietet Möglichkeiten dazu, ist aber keine elementare Datenstruktur. Wir geben daher später keine Laufzeiten für das Einfügen oder Entfernen von Elementen an.

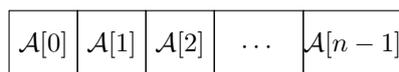


Abbildung 1.3: Ein Array  $\mathcal{A}$  von Länge  $n$

### 1.3.2 Listen

Nicht immer können wir vorhersagen, wie viele Datensätze wir im Laufe des Programmablaufs entgegennehmen. Daher benötigen wir Datenstrukturen, in welche wir Daten einfügen, aber auch wieder löschen können. Ein elementarer Prototyp ist die einfach verkettete Liste:

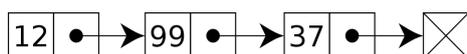


Abbildung 1.4: Graphische Darstellung einer einfach verketteten Liste: 12, 99 und 37 sind hier die Nutzdaten, der Schlusspointer `void` wird mit dem gekreuzten Quadrat dargestellt.

Eine Liste  $\mathcal{L}$  besteht dabei aus Elementen  $(d, p_{\text{next}})$ , wobei  $d$  die Nutzlast und  $p_{\text{next}}$  einen Zeiger (also die Speicheradresse bzw der Index in unserem RAM) auf das nächste Listenelement oder den sogenannten Nullpointer `void` darstellt.

Haben wir also einen Zeiger  $p$  zu einem Element, so bekommen wir mit der Funktion `data( $\mathcal{M}[p]$ )` die Datennutzlast  $d$  und mit `next( $\mathcal{M}[p]$ )` den Zeiger  $p_{\text{next}}$  zum nächsten Element. Ist der Zeiger  $p_{\text{next}} = \text{void}$ , so handelt es sich also um das letzte Element dieser Liste. Sowohl die Anfrage `next( $\mathcal{M}[\text{void}]$ )` als auch `data( $\mathcal{M}[\text{void}]$ )` bringen uns einen Fehler.

Nun hat unsere Liste ein Ende, sie braucht nur noch einen Anfang. Wir haben dafür ein spezielles Element `head( $\mathcal{L}$ )`, welches wir als nutzlastfreies Element verstehen, welches nur die Referenz auf das erste (richtige) Element enthält.

Eine neue, leere Liste ist also lediglich dieses Kopfelement (`void, void`): In diesem Fall haben wir keine Nutzlast, weil wir das Headelement sind (hier ebenfalls `void` ausgedrückt) und der Pointer auf das nächste Element ist der Nullpointer `void`.

Der Bequemlichkeit halber führen wir auch für Listen eine Indexnotation ein:  $\mathcal{L}[0]$  referiert auf das erste Element der Liste,  $\mathcal{L}[2]$  auf das dritte, etc. Dabei ist die Abkürzung definiert als

$$\begin{aligned} \mathcal{L}[i] &= \text{next}^i(\text{head}(\mathcal{L})) \\ &:= \underbrace{\text{next}(\text{next}(\text{next} \dots (\text{head}(\mathcal{L})) \dots ))}_{i \text{ mal}} \end{aligned}$$

Offensichtlich ist hier, anders als beim Array, die Zugriffszeit linear abhängig von  $i$ .

### Listenoperationen

Beginnen wir mit einem einfachen Check, ob unsere Liste  $\mathcal{L}$  leer ist oder nicht:

---

**Algorithmus 2 : `is_empty`( $\mathcal{L}$ )**

---

**Input :** A list  $\mathcal{L}$   
**Output :** A binary value. `True` in case  $\mathcal{L}$  is empty, `False` otherwise

```
[1] if next(head( $\mathcal{L}$ )) = void then
[2]   return True
[3] else
[4]   return False
```

---

Anders als in einem Array können wir nun an jeder Stelle der Liste etwas einfügen, siehe Abbildung 1.5. Beachtenswert ist hierbei, dass bei Kenntnis der Adresse des vorherigen Listenelements nur  $\mathcal{O}(1)$  Aufwand vonnöten ist.

---

**Algorithmus 3 : `insert_after`( $p, d$ )**

---

**Input :** A pointer  $p$  to the list element after which to insert the new list element storing  $d$   
**Output :** Side effects in the memory  $\mathcal{M}$

```
[1] new_element = ( $d, \text{next}(\mathcal{M}[p])$ )
[2] next( $\mathcal{M}[p]$ ) ← reference_of(new_element)
```

---

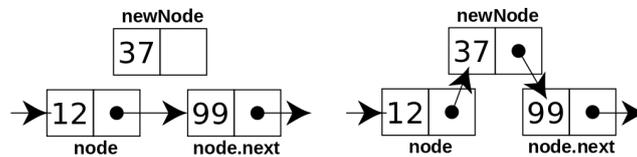


Abbildung 1.5: Einfügen eines Listenelements an arbiträrer Stelle

Auch für das Löschen ist nur das Umbiegen eines einzelnen Zeigers nötig, siehe Abbildung 1.6:

---

**Algorithmus 4 : delete\_after(p)**

---

**Input :** A pointer  $p$  to the list element whose successor shall be removed from the list

**Output :** Side effects in the memory  $\mathcal{M}$

[1]  $\text{next}(\mathcal{M}[p]) \leftarrow \text{next}(\text{next}(\mathcal{M}[p]))$

---

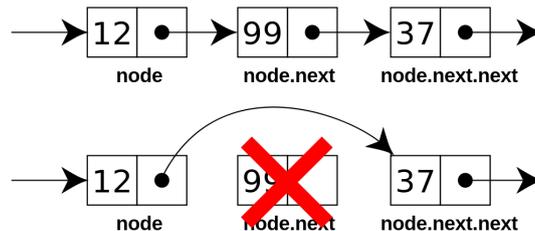


Abbildung 1.6: Löschen eines Listenelements an arbiträrer Stelle

**Varianten von Listen**

Es gibt Listen in fast allen Geschmacksrichtungen. Häufig verwendet werden beispielsweise die sogenannten doppelt verketteten Listen (Abbildung 1.7). Deren Elemente  $(d, p_{\text{prev}}, p_{\text{next}})$  haben zwei Pointer, sodass in beide Richtungen traversiert werden kann.

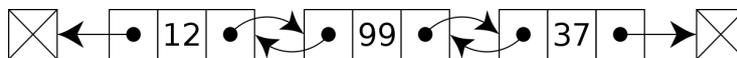


Abbildung 1.7: Graphische Darstellung einer doppelt verketteten Liste

Frei kombinierbar gibt es aber auch zirkulär verketteten Listen (Abbildung 1.8), bei denen anstelle der Referenz auf void wieder auf den Anfang referenziert wird.

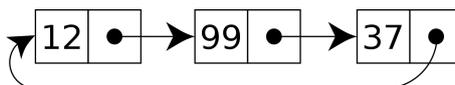


Abbildung 1.8: Graphische Darstellung einer zirkulär verketteten Liste

**1.3.3 Stack**

Ein Stack  $\mathcal{S}$  ist wie ein Stapel Teller: Hinzufügen oder Entnehmen von weiteren Tellern ist nur an der Spitze möglich.

Viele Anwendungen brauchen genau das, aber auch nicht mehr. Beispielsweise das Umdrehen der Reihenfolge einer Sequenz, das Umformen von Prefix in Postfixnotation, oder allgemeiner: Syntax parsen. Ebenso braucht man zur Verwaltung rekursiver Funktionsaufrufe (siehe Kapitel 1.4) einen Stack. Solche Algorithmen arbeiten mit der sogenannten LIFO-Strategie, was für last in, first out steht.

Aber auch aus rein abstrakter Sicht ist ein Stack interessant. Während wir gleich einen Stack mithilfe einer verketteten Liste implementieren (aus Bequemlichkeit, nicht aus Notwendigkeit), können umgekehrt auch Listen mit zwei Stacks implementiert werden. Es ist also, in diesem Sinne, eine ebenso fundamentale Datenstruktur.

Ein Stack hat zwei Operationen: `push`, was dem dazulegen eines weiteren Tellers entspricht, sowie `pop`, welches den obersten Teller entfernt. Die LIFO-Bedingung wird dann mit dem Axiom

$$x = \text{pop}(\text{push}(x))$$

beschrieben.

### Stackoperationen

Wir implementieren den Stack  $\mathcal{S}$  als oben definierte verkettete Liste. Ein leerer Stack ist daher auch einfach eine leere Liste  $\mathcal{L}$ .

Der Test auf Inhalt ist somit ident zum entsprechenden `is_empty` für Listen. Möchten wir nun ein Element mit Nutzlast  $d$  hinzufügen, so ist dies auch auf Listenoperationen zurückführbar:

$$\text{push}(\mathcal{S}, d) := \text{insert\_after}(\text{head}(\mathcal{L}), d)$$

Wir haben die gleiche Nebenwirkung im Speicher. Das Entfernen eines Elements besteht aus zwei elementaren Schritten, dem Auslesen, und dem Löschen:

---

#### Algorithmus 5 : `pop`( $\mathcal{S}$ )

---

**Input** : A Stack  $\mathcal{S}$

**Output** : The top element of the stack  $\mathcal{S}$

**Output** : Side effects in the memory  $\mathcal{M}$ : Removal of the returned top element

[1] `ret` ← `data`( $\mathcal{M}[\text{next}(\text{head}(\mathcal{S}))]$ )

[2] `delete_after`(`head`( $\mathcal{S}$ ))

[3] **return** `ret`

---

Charmant für uns ist, dass alle Operationen auf Stacks jeweils  $\mathcal{O}(1)$  Zeit brauchen, meist also vernachlässigbar sind.

### 1.3.4 Queue

Anders als der Stapel, der die LIFO-Strategie verfolgt, beschreibt die Queue eine (faire) Warteschlange: Wer als erstes da war, kommt als erstes dran. Man nennt es auch die FIFO-Strategie (first-in, first-out). Ansonsten hat auch sie zwei Funktion, `put`, das Hintanstellen, sowie `get`, das Drankommen.

Anwendungsfälle sind vor allem die namensgebenden Warteschlangen. Man könnte also beispielsweise einen sehr primitiven Betriebssystemscheduler damit beschreiben. Häufiger kommt es aber beispielsweise bei der Arbeitsverteilung eines aus vielen unabhängigen Einzeloperationen bestehenden Programms auf einzelne Threads vor.

### Queueoperationen

Wir implementieren eine Queue  $Q$  wieder über eine verkettete Liste  $\mathcal{L}$ , aber diesmal müssen wir noch eine Referenz auf das letzte Element mitführen (sonst müssten wir immer mit  $\mathcal{O}(n)$  Aufwand zum Ende der verketteten Liste laufen). Die Queue  $Q$  besteht also aus  $Q = (\mathcal{L}, p_{\text{last}})$ . Wir fügen Elemente am Ende der Liste hinzu und lesen sie vom Anfang der Kette ab. Der Test auf Leere ist wieder analog wie bei Listen.

---

#### Algorithmus 6 : put( $Q, d$ )

---

**Input :** A queue  $Q$ , data  $d$

**Output :** Side effects in the memory  $\mathcal{M}$ : Insertion of an Element at the end of the list  $\mathcal{L}$  as well as change of the value  $p_{\text{last}}$  of  $Q$

[1] insert\_after( $p_{\text{last}}, d$ )

[2]  $p_{\text{last}} \leftarrow \text{next}(\mathcal{M}[p_{\text{last}}])$

---

Die Funktion get entspricht genau dem Stackpendant pop:

---

#### Algorithmus 7 : get( $Q$ )

---

**Input :** A queue  $Q$

**Output :** The oldest element of the queue  $Q$

**Output :** Side effects in the memory  $\mathcal{M}$ : Removal of the returned top element

[1] ret  $\leftarrow$  data( $\mathcal{M}[\text{next}(\text{head}(\mathcal{L}))]$ )

[2] delete\_after(head( $\mathcal{L}$ ))

[3] return ret

---

Wiederum benötigen alle Operationen lediglich  $\mathcal{O}(1)$  Zeit.

### 1.3.5 Übersichtstabelle zu den asymptotischen Aufwänden

Datenstruktur $\mathcal{D}$	$\mathcal{D}[1]$	$\mathcal{D}[i]$	$\mathcal{D}[n]$	Einfügen	Löschen	Suchen
Array	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$			$\mathcal{O}(n)$
Liste	$\mathcal{O}(1)$	$\mathcal{O}(i)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Stack	$\mathcal{O}(1)$			$\mathcal{O}(1)$	$\mathcal{O}(1)$	
Queue			$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	

Beim Einfügen oder Löschen in Listen wird vorausgesetzt, dass die Adresse  $p$  bekannt ist. Ist das Element nur durch seine Position oder seinen Inhalt bekannt, muss zuerst mit entsprechendem Aufwand zum Element traversiert werden.

## 1.4 Rekursion

Warum Rekursion?

Rekursion ist beliebt, weil sie oftmals eine sehr knappe, präzise, mathematische Formulierung von Lösungen erlaubt. Korrektheitsbeweise sind oft einfacher zu führen (nämlich induktiv), ebenso lassen sich meist recht schnell Laufzeitfunktionen finden und ihre Komplexitätsklasse bestimmen. Allerdings kann es auch vorkommen, dass eine (naive) rekursive Lösung deutlich langsamer läuft und mehr Speicher benötigt.

### 1.4.1 Die Fakultätsfunktion $n!$

Wir berechnen  $! : \mathbb{N} \rightarrow \mathbb{N}$ .

Algorithmus 8 : <code>facit(n)</code>	Algorithmus 9 : <code>facrec(n)</code>
<b>Input</b> : $n \in \mathbb{N}$	<b>Input</b> : $n \in \mathbb{N}$
<b>Output</b> : $n!$	<b>Output</b> : $n!$
[1] $r \leftarrow 1$	[1] <b>if</b> $n = 1$ <b>then</b>
[2] <b>for</b> $i \leftarrow 1$ <b>to</b> $n$ <b>do</b>	[2] <b>return</b> 1
[3] $r \leftarrow r * i$	[3] <b>else</b>
[4] <b>return</b> $r$	[4] <b>return</b> $n * \text{facrec}(n - 1)$

Die Zeitanalyse ist bei beiden Versionen sehr einfach. Mit  $f \in \mathcal{O}(1)$  als Ausführzeit von Zeile 3 gilt:

$$T^{\text{facit}}(n) = nf \in \mathcal{O}(n).$$

Der Speicherverbrauch ist konstant. Die rekursive Variante hat Aufwand  $x \in \mathcal{O}(1)$ , wenn sie mit  $n = 1$  aufgerufen wird, ansonsten hat sie Aufwand  $g \in \mathcal{O}(1)$  sowie den Aufwand des rekursiven Funktionsaufwandes in Zeile 4. Damit

$$\begin{aligned} T^{\text{facrec}}(n) &= g(n) + T^{\text{facrec}}(n - 1) \\ &= g(n) + (g(n - 1) + T^{\text{facrec}}(n - 2)) \\ &= \dots \\ &= (n - 1)g(n) + x(n) \in \mathcal{O}(n). \end{aligned}$$

In der Laufzeit unterscheiden sich die Algorithmen also nicht. Aber was ist mit ihrem Speicherverbrauch?

Naiv würde man sich das Ausführen des Codes etwa so vorstellen: Der Computer arbeitet das Programm ab, sieht also:

$$\begin{aligned} \text{facrec}(n) &= n * \text{facrec}(n - 1) \\ &= n * ((n - 1) * \text{facrec}(n - 2)) \\ &= n * ((n - 1) * ((n - 2) * \text{facrec}(n - 3))) \\ &= n * ((n - 1) * (\dots (2 * \text{facrec}(1)) \dots)) \\ &= n * ((n - 1) * (\dots (2 * 1) \dots)) \end{aligned}$$

Aber irgendwo muss die Information, dass gerade  $n$  Aufrufe von `facrec` stattfinden, welche Parameter die Aufrufe haben und wo mit ihrem Rückgabewert

weitergerechnet werden soll abgespeichert werden. Das findet üblicherweise auf einem Stack statt, wo für jeden Rekursionsaufruf ein neues Objekt draufgelegt wird, welches Funktion, Funktionsparameter, sowie Rücksprungadresse abspeichert<sup>1</sup>. Wir brauchen auf diesem Wege also  $\mathcal{O}(n)$  Speicher.

Compiler moderner (funktionaler) Sprachen können solche Rekursionen aber direkt in ihre iterativen Pendant umschreiben<sup>2</sup>, sodass bei ihrem Einsatz Speicherverbrauch/Geschwindigkeit eigentlich kein ausschlaggebendes Kriterium mehr sein sollte. Momentan (Stand 2020) führen beispielsweise Haskell und Scala solche Umwandlungen durch, während beispielsweise C, Java und Python mit call stacks arbeiten.

### 1.4.2 Die Fibonacci-Zahlen $\text{fib}(n)$

Wir berechnen die Fibonacci-Zahlen durch die Funktion  $\text{fib} : \mathbb{N} \rightarrow \mathbb{N}$ , welche wie folgt definiert ist:  $\text{fib}(n) := \text{fib}(n-1) + \text{fib}(n-2)$ .

Algorithmus 10 : $\text{fibr}(n)$	Algorithmus 11 : $\text{fibrec}(n)$
<b>Input</b> : $n \in \mathbb{N}$	<b>Input</b> : $n \in \mathbb{N}$
<b>Output</b> : $\text{fib}(n)$	<b>Output</b> : $\text{fib}(n)$
[1] $x_{-1} \leftarrow 1$	[1] <b>if</b> $n \in \{0, 1\}$ <b>then</b>
[2] $x \leftarrow 1$	[2] <b>return</b> 1
[3] <b>for</b> $i \leftarrow 3$ <b>to</b> $n$ <b>do</b>	[3] <b>else</b>
[4] $x_{-2} \leftarrow x_{-1}$	[4] <b>return</b>
[5] $x_{-1} \leftarrow x$	$\text{fibrec}(n-2) + \text{fibrec}(n-1)$
[6] $x \leftarrow x_{-2} + x_{-1}$	
[7] <b>return</b> $x$	

Die Korrektheit der rekursiven Variante ist sofort ablesbar, es handelt sich lediglich um die in Code gegossene Mathematische Definition. Das hat aber seinen Preis: Während die iterative Variante Laufzeit  $\mathcal{O}(n)$  und konstanten Speicherverbrauch hat, erzeugt jeder Aufruf von  $\text{fibrec}$  zwei neue Funktionen. Damit haben wir exponentielles Wachstum, also Laufzeit in  $\mathcal{O}(2^n)$ . Je nach konkreter Implementierung und Abarbeitung des Stacks haben wir zudem  $\mathcal{O}(2^n)$  (Breitensuche) oder auch "nur"  $\mathcal{O}(n)$  (Tiefensuche) Speicherverbrauch.

Auch funktionale Sprachen haben hier keine wirklich Hebel, die Laufzeit zu verbessern. Diese naive Implementierung ist somit leider für größere  $n$  unbrauchbar. Selbst eine moderne CPU berechnet  $\text{fibrec}(1000)$  nichtmehr in unserer Lebenszeit.

### Die Türme von Hanoi\*

Ein weiteres klassisches Problem sind die Türme von Hanoi. Wir haben folgende Spielregeln:

- 3 Stäbe  $A, B$  und  $C$ .
- Am Stab  $A$  liegen zu Beginn  $n \in \mathbb{N}$  unterschiedliche große Scheiben, geordnet nach Größe (kleinste oben).

<sup>1</sup>Stichwort: "call stack" führt zu Details gängiger Implementierungen

<sup>2</sup>Stichwort: tail recursion bzw Endrekursion

- Am Ende sollen die Scheiben in dieser Reihenfolge auf Stab  $B$  liegen.
  - Es darf jeweils nur eine Scheibe bewegt werden.
  - Es darf nie eine größere auf einer kleineren Scheibe liegen.
- TODO: Wird noch ausgeführt. Oder eine Übungsaufgabe.

### 1.4.3 Binärsuche

Ein sehr praxisnahes Beispiel für Rekursion ist die Binärsuche auf sortierten Arrays. Wir suchen in einem sortierten Array, welches Objekte enthält, die nach dem Schlüssel  $k$  sortiert sind, das Objekt mit dem Schlüssel  $k_0$ . Die Funktion  $\text{key}(x)$  verrät uns hierbei den Schlüssel eines Objekts  $x$ . Wir nehmen dabei an, dass das gesuchte Objekt existiert.

---

#### Algorithmus 12 : `binsearchrec`

---

**Input :** Nach Schlüsseln sortiertes Array  $\mathcal{A}$ , sowie den gesuchten Schlüssel  $k_0$   
**Output :** Das Objekt mit dem Schlüssel  $k_0$

```

[1]  $m \leftarrow \lfloor \frac{\text{len}(\mathcal{A})}{2} \rfloor$ 
[2] if  $\text{key}(\mathcal{A}[m]) > k_0$  then
[3]   return binsearchrec( $\mathcal{A}[0, \dots, m]$ )
[4] else if  $\text{key}(\mathcal{A}[m]) < k_0$  then
[5]   return binsearchrec( $\mathcal{A}[m + 1, \dots, \text{len}(\mathcal{A}) - 1]$ )
[6] else
[7]   return  $\mathcal{A}[m]$ 

```

---

Machen wir zuerst eine Aufwandsanalyse per Hand, bevor wir im nächsten Kapitel eine schnellere Methode sehen.

Die Laufzeit von `binsearchrec` hat Unterschiede im best-case oder worst-case Szenario. Best-case wäre, wenn wir schon beim ersten Aufruf zufällig unser gesuchtes Element finden (Aufwand  $f \in \mathcal{O}(1)$ ). Im pessimistischen Fall teilen wir unser Array so lange, bis nur noch ein Element, das Gesuchte, übrig bleibt. Würden wir auch die Suche nach nicht-enthaltenen Schlüsseln erlauben, wären wir auch immer im worst-case. Betrachten wir hier also  $T(n) = T_{\text{worst}}^{\text{binsearchrec}}$ . Ein Durchlauf von `binsearchrec` hat dabei Aufwand  $f \in \mathcal{O}(1)$  sowie den Aufwand rekursiver Aufrufe.

$$\begin{aligned}
 T(n) &= f(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\
 &= f(n) + f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor\right) \\
 &= \dots \\
 &\simeq f(n) + f\left(\frac{n}{2}\right) + f\left(\frac{n}{4}\right) + \dots + f(1) \\
 &\leq \log_2(n) f(n) \in \mathcal{O}(\ln(n))
 \end{aligned}$$

Der asymptotische Aufwand liegt damit in  $T_{\text{worst}}^{\text{binsearchrec}} \in \mathcal{O}(\log n)$ , siehe Hauptsatz der Laufzeitfunktionen. Damit ist überhaupt erst die Motivation für Sortieralgorithmen gegeben: Suchen in sortierten Arrays geht verdammt schnell.

Die Binärsuche ist zudem ein Randbeispiel des folgenden Prinzips:

#### 1.4.4 Divide et impera sowie der Hauptsatz der Laufzeitfunktionen

Eine ganze Klasse rekursiver Algorithmen folgen dem sogenannten “divide et impera”-Prinzip.

Grundidee ist immer, ein Problem der Größe  $n$  in  $b \in \mathbb{N}$  Teilprobleme zu Zerlegen und deren Teillösungen mit Aufwand  $f : \mathbb{N} \rightarrow \mathbb{N}$  wieder zusammenzufügen. Das Zerlegen geschieht rekursiv immer weiter, bis ein gewisses Abbruchkriterium gefunden wird. Damit ergibt sich folgende (rekursive) Aufwandsbeschreibung der Funktion:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Ein klassisches Beispiel ist der **mergesort**-Algorithmus (Details im nächsten Kapitel). Das zu sortierende Array wird mittig geteilt, die beiden Teilarrays werden (wieder rekursiv) sortiert. Die beiden sortierten Teilarrays werden nun verschmolzen. Dabei müssen immer nur die jeweils kleinsten Elemente miteinander verglichen werden.

Aber auch die eben erwähnte Binärsuche fällt in dieses Muster: Hier haben wir  $a = 1$  Teilprobleme, welches jeweils in  $b = 2$  Teilprobleme zerlegt wird. Sowohl das zerteilen als auch das rekursive Zurückgeben des Algorithmus geschieht mit  $f \in \mathcal{O}(1)$ .

Dank des Hauptsatz der Laufzeitfunktionen können wir die asymptotische Laufzeit in vielen Fällen direkt ablesen:

**Satz 2** (Hauptsatz der Laufzeitfunktionen). *Sei  $T$  die Laufzeitfunktion eines divide-et-impera Algorithmus, welcher die Laufzeit (best case bzw. worst case)*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

*hat. Hierbei ist  $a \in \mathbb{N}$  die Anzahl der Unterprobleme in der Rekursion,  $b$  der Teiler des Originalproblems, welcher im Unterproblem betrachtet wird und  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  die Kostenfunktion für das Teilen des Problems und Kombination der Teillösungen.*

*Gilt zudem weiterhin eine der folgenden Voraussetzungen, so können wir die asymptotische Laufzeit sofort bestimmen:*

- 1. Gibt es ein echt positives  $\varepsilon \in \mathbb{R}^+$ , sodass  $f \in \mathcal{O}(n^{\log_b(a)-\varepsilon})$ , so liegt die Laufzeit  $T \in \Theta(n^{\log_b a})$ .*
- 2. Ist  $f \in \Theta(n^{\log_b a})$ , so liegt die Laufzeit  $T \in \Theta(n^{\log_b a} \log n)$ .*
- 3. Gibt es ein echt positives  $\varepsilon \in \mathbb{R}^+$ , sodass  $f \in \Omega(n^{\log_b(a)+\varepsilon})$  und weiterhin für ein  $c \in \mathbb{R}$  mit  $0 < c < 1$  und alle hinreichend großen  $n$  die Abschätzung  $af(\frac{n}{b}) \leq cf(n)$ , so liegt die Laufzeit  $T \in \Theta(f)$ .*

Greift keiner der drei Fälle, so muss die Laufzeit anders bestimmt werden. Insbesondere **facrec** und **fibrec** lassen sich nicht mit dem Hauptsatz der Laufzeitfunktionen behandeln, da ihre Teilprobleme kein Bruchteil  $\frac{n}{b}$  des Hauptproblems sind.

Wir untersuchen nun als Beispiel die asymptotische Laufzeit von **binsearchrec** noch einmal. Wir haben pro Rekursionsschritt ein ( $a = 1$ ) halb so großes Unterproblem ( $b = 2$ ), damit also  $\log_b a = 0$ . Die konstante Funktion  $f$  liegt für ein  $\varepsilon \in \mathbb{R}^+$  gerade nichtmehr in  $\mathcal{O}(n^{\log_b(a)-\varepsilon})$ , aber definitiv in  $\mathcal{O}(n^{\log_b a}) = \mathcal{O}(n^0)$ , also haben wir den zweiten Fall und unser Aufwand liegt in  $\Theta(n^{\log_b a} \log n) = \Theta(\log n)$ .

## Kapitel 2

# Sortieralgorithmen

In diesem Kapitel werden wir uns mit Sortieralgorithmen beschäftigen. Den Anfang machen zwei Sortierverfahren für Arrays, `quicksort` und `mergesort`. Danach werden wir uns der theoretischen unteren Grenze vergleichsbasierter Sortieralgorithmen widmen, nur um sie kurz darauf (mit erweiterten Voraussetzungen an die zu sortierenden Objekte) mit `radixsort` wieder zu brechen.

### 2.1 Mergesort

Diese Sortierverfahren ist ein Paragon der sogenannten *divide et impera*-Strategie.

Ausgehend von einem linearen Feld  $\mathcal{A}$  der Größe  $n$  unterteilen wir das Problem rekursiv immer weiter in kleinere Probleme, welche superlinear schneller gelöst werden können. Der Basisfall, ein Array mit einem einzigen Element, ist dann immer sortiert. Im letzten Schritt vereinen wir alle sortierten Teilarrays zu einem großen sortierten Array. Siehe Abbildung 2.1 für ein Beispiel.

Im folgenden Pseudocode verwenden wir die Funktion `Array( $n$ )`, welches uns einfach nur ein leeres (also z.B. mit 0 gefülltes) Array der Länge  $n$  gibt, sowie die Funktion `len( $\mathcal{A}$ )`, welche die Länge des arrays  $\mathcal{A}$  zurückgibt.

---

**Algorithmus 13** : `mergesort( $\mathcal{A}$ )`

---

**Input** : An array  $\mathcal{A}$  of size  $n$

**Output** : The sorted array  $\mathcal{A}$

```
[1] if  $n = 1$  then  
[2]   return  $\mathcal{A}$   
[3] else  
[4]    $k \leftarrow \lfloor \frac{n}{2} \rfloor$   
[5]   return merge(mergesort( $\mathcal{A}[0, \dots, k]$ ), mergesort( $\mathcal{A}[k + 1, \dots, n]$ ))
```

---

Wobei `merge` folgendermaßen definiert wird:

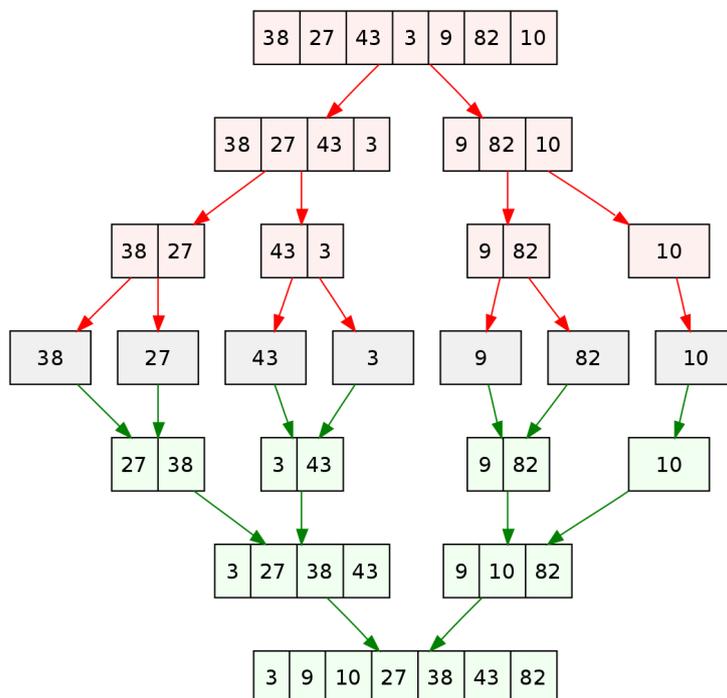


Abbildung 2.1: Ein Beispiellauf des mergesort-Algorithmus auf den Input  $\mathcal{A} = (38, 27, 43, 3, 9, 82, 10)$

---

**Algorithmus 14 : merge( $\mathcal{A}_1, \mathcal{A}_2$ )**

---

**Input :** Two sorted arrays  $\mathcal{A}_1$  and  $\mathcal{A}_2$

**Output :** The sorted union array  $\mathcal{A}$  of the input

- [1]  $\mathcal{A} \leftarrow \text{Array}(\text{len}(\mathcal{A}_1) + \text{len}(\mathcal{A}_2))$
  - [2]  $i_1, i_2 \leftarrow 0$
  - [3] **for**  $i \leftarrow 0$  **to**  $\text{len}(\mathcal{A})$  **do**
  - [4] **if**  $\mathcal{A}_1[i_1] \leq \mathcal{A}_2[i_2]$  **then**
  - [5]  $\mathcal{A}[i] \leftarrow \mathcal{A}_1[i_1]$
  - [6]  $i_1 \leftarrow i_1 + 1$
  - [7] **else**
  - [8]  $\mathcal{A}[i] \leftarrow \mathcal{A}_2[i_2]$
  - [9]  $i_2 \leftarrow i_2 + 1$
  - [10] **return**  $\mathcal{A}$
- 

**Laufzeit**

Die Laufzeit der Algorithmen ist nicht vom konkreten Input, sondern nur von der Länge des Inputs abhängig. Worst, best und average case sind also gleich. Für merge ist die Länge des Inputs  $n = \text{len}(\mathcal{A}_1) + \text{len}(\mathcal{A}_2)$  und es gilt:  $T_m = T_{\text{merge}} = f \in \mathcal{O}(n)$ , da Zeile 3 alles dominiert.

Damit ist die Laufzeitfunktion für mergesort für  $T_{\text{ms}}(1) \in \mathcal{O}(1)$  und für

$n > 1$

$$T_{\text{ms}}(n) = 2T_{\text{ms}}\left(\frac{n}{2}\right) + T_{\text{m}}(n) + d,$$

wobei  $d \in \mathcal{O}(1)$ . Mit dem Hauptsatz der Laufzeitfunktionen haben wir damit Fall 2, die asymptotisch exakte Laufzeit beträgt ( $\log_2 2 = 1$ ) also

$$T^{\text{ms}} \in \Theta(n \log n).$$

Um die Methodik der rekursiven Laufzeitberechnungen zu demonstrieren, berechnen wir die eine Seite der Abschätzung elementar. Um den Beweis übersichtlicher zu halten, nehmen wir an, dass  $n = 2^k$  für ein  $k \in \mathbb{N}$ . Für Werte von  $n$ , welche keine Zweierpotenz sind, können wir aufgrund der Monotonie von  $T_{\text{ms}}$  dann abschätzen.

$$\begin{aligned} T_{\text{ms}} &= 2T_{\text{ms}}\left(\frac{n}{2}\right) + T_{\text{m}}(n) + d \\ &= 4T_{\text{ms}}\left(\frac{n}{4}\right) + 2T_{\text{m}}\left(\frac{n}{2}\right) + 2d + T_{\text{m}}(n) + d \\ &= 8T_{\text{ms}}\left(\frac{n}{8}\right) + 4T_{\text{m}}\left(\frac{n}{4}\right) + 4d + 2T_{\text{m}}\left(\frac{n}{2}\right) + 2d + T_{\text{m}}(n) + d \\ &\vdots \\ &= \underbrace{nT_{\text{ms}}(1)}_{\in \mathcal{O}(n)} + \sum_{l=0}^{\log_2\left(\frac{n}{2}\right)} 2^l T_{\text{m}}\left(\frac{n}{2^l}\right) + \underbrace{\sum_{l=0}^{\log_2\left(\frac{n}{2}\right)} 2^l d}_{\in \mathcal{O}(n)} \end{aligned}$$

Wir müssen also nun noch den asymptotischen Aufwand für die rekursiven Aufrufe von  $T_{\text{m}}$  finden. Wir wissen, da  $T_{\text{m}} \in \mathcal{O}(n)$ , dass es ein  $n_0 \in \mathbb{N}$  und  $c \in \mathbb{R}$  gibt, sodass  $T_{\text{m}}(n) \leq cn$  für alle  $n > n_0$ . Insbesondere also für die nächstgrößere Zweierpotenz, sodass wir o.B.d.A. annehmen können:  $n_0 = 2^{k_0}$  für ein  $k_0 \in \mathbb{N}$ . Für alle  $n \leq n_0$  können wir den Aufwand von  $T_{\text{m}}(n)$  zudem durch die Konstante  $m = \max(\{T_{\text{m}}(i) \mid i \in \{0, \dots, n_0\}\})$  abschätzen.

Das erlaubt es uns, obige Summe aufzuspalten:

$$\sum_{l=0}^{k-1} 2^l T_{\text{m}}\left(\frac{n}{2^l}\right) = \sum_{l=0}^{k-k_0} 2^l T_{\text{m}}\left(\frac{n}{2^l}\right) + \underbrace{\sum_{l=k-k_0+1}^{k-1} 2^l T_{\text{m}}\left(\frac{n}{2^l}\right)}_{\leq 2^k m \in \mathcal{O}(n)}.$$

Es bleibt noch der erste Teil. Aber hier greift endlich die Abschätzung  $T_{\text{m}}(n) \leq cn$ , und damit

$$\sum_{l=0}^{k-k_0} 2^l T_{\text{m}}\left(\frac{n}{2^l}\right) = c \sum_{l=0}^{k-k_0} 2^l \frac{n}{2^l} = c \sum_{l=0}^{k-k_0} n \in \mathcal{O}(n \log_2 n).$$

Dieser Term dominiert also alle anderen Terme, und damit  $T_{\text{ms}} \in \mathcal{O}(n \log n)$ .

### Speicherverbrauch

In der hier vorgestellten Variante muss bei jedem Aufruf von `merge` ein neues Array angelegt werden. Damit braucht `mergesort` asymptotisch  $\mathcal{O}(n)$  zusätzlichen Speicher.

Es gibt allerdings auch Weiterentwicklungen, die mit  $\mathcal{O}(1)$  Speicher auskommen (z.B. TimSort oder blocksort).

## 2.2 Quicksort

Quicksort ist einer der meist verwendeten Sortieralgorithmen. Die Idee ist wieder das divide-et-impera-Prinzip, der Kern des Algorithmus ist die Zerlegung des Arrays, die sogenannte Partition.

### 2.2.1 Partition

Für eine Partition eines Arrays  $\mathcal{A}$  wählen wir zuerst ein Pivotelement  $p$ . Nun sortieren wir das Array so, dass links von  $p$  nur Elemente stehen, die kleiner als  $p$  sind, rechts nur Elemente, die größer sind. Vorerst nehmen wir als Pivotelement einfach das letzte Arrayelement.

Die Funktion  $\text{swap}(a, b)$  im folgenden Pseudocode tauscht hierbei die Werte von  $a$  und  $b$ .

---

#### Algorithmus 15 : $\text{partition}(\mathcal{A})$

---

**Input :** An unsorted array  $\mathcal{A}$  of length  $n$

**Output :** The partition of  $\mathcal{A}$  wrt. the pivot  $p$  and its index  $k$

```
[1]  $p \leftarrow \mathcal{A}[n - 1]$ 
[2]  $i \leftarrow -1$ 
[3] for  $j \leftarrow 0$  to  $n - 2$  do
[4]     if  $\mathcal{A}[j] \leq p$  then
[5]          $\text{swap}(\mathcal{A}[i + 1], \mathcal{A}[j])$ 
[6]          $i \leftarrow i + 1$ 
[7]  $\text{swap}(\mathcal{A}[i + 1], \mathcal{A}[n - 1])$ 
[8]  $i \leftarrow i + 1$ 
[9] return  $(\mathcal{A}, i)$ 
```

---

#### Korrektheitsüberlegungen:

Was passiert hier? Das Pivotelement  $p$  ist das letzte Element von  $\mathcal{A}$ , siehe Zeile [1]. Wie oben erwähnt, partitionieren wir das Array in Elemente, welche kleiner oder gleich  $p$  sind, sowie die Elemente, welche größer als  $p$  sind. Das Pivotelement  $p$  dient erstmal nur zum Vergleichen ([4]), bleibt aber ansonsten außen vor, bis es in Zeile [7] an seine endgültige Position getauscht wird.

Die endgültige Position von  $p$  wird von dem Index  $i$  bestimmt. Dabei erfüllt  $i$  zu jedem Zeitpunkt die Bedingung, dass alles, was sich links von  $i$  befindet ( $i$  eingeschlossen), stets kleiner oder gleich  $p$  ist (Zeile [4] und [5])! Unter den Elementen, die bereits mit  $p$  verglichen wurden (siehe Laufindex  $j$ ) nimmt  $i$  dabei den maximalen Wert ein (Nach Ausführung von Zeile [6], bzw Zeile [8]). Zu beachten ist außerdem, dass durch das rechtzeitige Addieren von  $i + 1$  nie ein Arrayzugriff an undefinierter Stelle geschieht ([5]), selbst wenn  $i$  mit  $-1$  initialisiert wurde ([2]).

Als letztes muss nur noch die Rolle des Laufindex  $j$  geklärt werden. Definiert in Zeile [3] startet  $j$  beim ersten Element und geht bis zum vorletzten (also exklusiv  $p$ ). Da pro Schleifendurchgang  $i$  um maximal eins inkrementiert werden kann, ist  $j$  also stets größergleich  $i$ . Dabei zeigt  $j$  an, welche Elemente des Arrays bereits mit  $p$  verglichen wurden. Findet  $j$  mit Zeile [4] ein Element, welches kleiner ist als  $p$ , wird dieses in den von  $i$  markierten Bereich vertauscht ([5]).

Dabei wird eine Sortierung innerhalb einer Partition zwar vielleicht zerstört, aber das ist für die Korrektheit des Algorithmus nicht relevant.

Man kann also feststellen: Das Array ist stets in vier (möglicherweise leere) Teilbereiche unterteilt. Der Speicherort von  $p$  (Anfangs  $n-1$ ), die noch nicht verglichenen Elemente  $j < \text{index} \leq n-2$ , die kleiner als  $p$  eingestuften Elemente  $0 \leq \text{index} \leq i$  und die als größer als  $p$  eingestuften Elemente  $i < \text{index} \leq j$ .

Da ein Schleifendurchlauf die Schleifeninvarianten (also der Programmzustand exakt vor dem Inkrementieren von  $j$ )

- Elemente mit Index  $x$ , wobei  $x \leq i$ , sind kleiner oder gleich  $p$ ,
- Elemente mit Index  $x$ , wobei  $i < x$  und  $x \leq j$ , sind größer  $p$ ,
- Elemente mit Index  $x$ , wobei  $j < x$ , sind noch nicht überprüft,

erhält, aber die Anzahl der nicht überprüften Elemente pro Schleifendurchlauf um eins schrumpft, terminiert der Algorithmus. Mit Zeile [7] wird am Ende noch  $p$  an seinen richtigen Platz verschoben, damit eine korrekte Partition zurückgegeben werden kann.

#### **Laufzeit:**

Die Schleife in Zeile [3] bis [6] hat an sich eine Laufzeit in  $\mathcal{O}(1)$ , wird aber  $\mathcal{O}(n)$  mal aufgerufen. Dadurch ergibt sich, dass die Laufzeit von  $T_p = T_{\text{partition}}(\mathcal{A})$  in  $\Theta(n)$  liegt, wobei  $n = \text{len}(\mathcal{A})$ . Es gibt keinen asymptotischen Unterschied zwischen best/worst-case.

### **2.2.2 Quicksort**

Auf der Basis von `partition` kann der Sortieralgorithmus `quicksort` konstruiert werden.

---

#### **Algorithmus 16 : quicksort( $\mathcal{A}$ )**

---

**Input :** An unsorted array  $\mathcal{A}$  of length  $n$

**Output :** The same array  $\mathcal{A}$ , but sorted

```
[1] if  $n > 1$  then
[2]    $(\mathcal{A}, k) \leftarrow \text{partition}(\mathcal{A})$ 
[3]   return concat(quicksort( $\mathcal{A}[0, \dots, k-1]$ ),  $[A[k]]$ , quicksort( $\mathcal{A}[k+1, \dots, n-1]$ ))
[4] else
[5]   return  $\mathcal{A}$ 
```

---

Die Funktion `concat` ist hierbei eine  $\mathcal{O}(1)$ -Operation, da sie vom Compiler wegoptimiert werden kann. Aber auch als  $\mathcal{O}(n)$ -Operation wäre sie asymptotisch irrelevant, da `partition` bereits  $\mathcal{O}(n)$  Zeit braucht. Sollten wir Arrays auf Grenzen wie  $[0, -1]$  aufrufen, so ist damit das leere Array gemeint.

#### **Korrektheitsüberlegungen:**

Die Korrektheit von `quicksort` ist schneller einsehbar als die von `partition`. Ist die Partitionseigenschaft bezüglich des Pivotelements in Position  $k$  erfüllt, so haben wir links und rechts davon echt kleinere Unterarrays, welche durch Rekursion sortiert werden (der Basisfall von einem Element ist trivial sortiert). Der

Algorithmus `quicksort` terminiert als spätestens nach  $n$  rekursiven Aufrufen und arbeitet dabei korrekt.

**Laufzeit:**

Die allgemeine Rekursionsgleichung für  $T_{\text{qs}} = T_{\text{quicksort}}$  lautet

$$T_{\text{qs}}(n) = T_{\text{qs}}(m) + T_{\text{qs}}(n - m - 1) + T_p(n) + f(n), \text{ wobei } f \in \mathcal{O}(1).$$

Dabei verschwindet  $f$  völlig unter  $T_p \in \mathcal{O}(n)$ . Der Parameter  $m \in \{0, \dots, n - 1\}$  beschreibt dabei die Position des Pivotelements. Die Laufzeit des Algorithmus hängt nun stark von  $m$  ab:

**Best case** Im besten Fall treffen wir mit dem Pivotelement  $p$  genau den Median von  $\mathcal{A}$ . Dann haben wir durch  $m \simeq \frac{n}{2}$  pro Rekursionsschritt eine balancierte Aufteilung des Rekursionsbaums, die Rekursionsgleichung wird zu

$$T_{\text{qs}}(n) = 2T_{\text{qs}}\left(\frac{n}{2}\right) + T_p(n).$$

Nach dem Hauptsatz der Laufzeitfunktionen ist mit  $a = 2, b = 2, f \in \mathcal{O}(n)$  die Laufzeit im best-case damit in  $\mathcal{O}(n \log n)$ .

**Worst case** Im schlechtesten Fall teilen wir das Array sehr ungünstig: Das Pivotelement ist immer das Maximum oder das Minimum, unser Array wird also aufgeteilt in ein  $n - 1$  großes Array,  $m = 1$  in jedem Rekursionsschritt. Damit haben wir keinen echten Bruchteil pro Rekursionsschritt und das Master-Theorem lässt sich nicht anwenden. Also rechnen wir per Hand:

$$\begin{aligned} T_{\text{qs}}(n) &= T_{\text{qs}}(1) + T_{\text{qs}}(n - 1) + T_p(n) \\ &= 2T_{\text{qs}}(1) + T_{\text{qs}}(n - 2) + T_p(n - 1) + T_p(n) \\ &= 3T_{\text{qs}}(1) + T_{\text{qs}}(n - 3) + T_p(n - 2) + T_p(n - 1) + T_p(n) \\ &\quad \vdots \\ &= \underbrace{nT_{\text{qs}}(1)}_{\in \Theta(n)} + \underbrace{\sum_{i=1}^n T_p(i)}_{\in \Theta(n^2)?} \end{aligned}$$

Dabei gibt uns die Eulersche Summenformel  $\sum_{i=1}^n i = \frac{n^2-n}{2} \in \mathcal{O}(n^2)$  einen Hinweis, in welcher Aufwandsklasse der Summenterm liegen könnte. Wir formalisieren jetzt also die Idee, dass wir  $T_p(n)$  durch  $c_2n$  von oben und  $c_1n$  von unten abschätzen können, sobald die  $n$  groß genug werden.

Betrachten wir also  $\sum_{i=1}^k T_p(i)$ . Mit  $T_p \in \Theta(n)$  wissen wir, dass  $n_0, c_1, c_2$  existieren, sodass  $c_1n' \leq T_p(n') \leq c_2n'$  für alle  $n' > n_0$ . Dieses  $n_0$  ist aber fix, d.h. für ein groß genuges  $k$  (und  $k$  soll später gegen unendlich gehen) betrachten wir also  $\sum_{i=n_0}^k T_p(i)$ . Hier gilt:

$$\underbrace{c_1 \sum_{i=n_0}^k i}_{\in \Omega(k^2)} \leq \sum_{i=n_0}^k T_p(i) \leq \underbrace{c_2 \sum_{i=n_0}^k i}_{\in \mathcal{O}(k^2)}$$

wie uns die Eulersche Summenformel verrät. Mit einer beidseitigen Abschätzung haben wir hier also obige Vermutung bewiesen.

Wir würden nun gerne noch eine average-case Analyse durchführen. Bei einer Gleichverteilung des Inputs ergibt sich nämlich auch eine average-case Aufwand von  $T_{\text{qs}}^{\text{avg}} \in \mathcal{O}(n \log n)$ . Allerdings benötigt eine average-case Analyse Annahmen über die Verteilung des Inputs.

Wir analysieren daher eine stark verwandte Variante, den randomisierten Quicksort.

### 2.2.3 Randomisierter Quicksort

Grundidee des randomisierten Quicksort Algorithmus ist es, nicht mehr ein fixes Pivotelement in der Partition zu wählen (wie bei uns das erste Element), sondern ein zufälliges. Dadurch kann man eine gute *durchschnittliche* Performance erreichen.

Das Wort *durchschnittlich* bekommt hier aber eine andere andere Bedeutung als in der average-case-Analyse! In der average-case-Analyse betrachtet man die durchschnittliche Laufzeit über alle möglichen Inputs (gewichtet mit der Wahrscheinlichkeit des entsprechenden Inputs), hier hingegen betrachtet man die durchschnittliche Laufzeit *über die verschiedenen zufälligen Läufe des nichtdeterministischen Algorithmus* (gewichtet nach Wahrscheinlichkeit des entsprechenden Durchlaufs).

Definieren wir zuerst die randomisierte Partition:

---

**Algorithmus 17** : `randomized_partition`( $\mathcal{A}$ )

---

**Input** : An unsorted array  $\mathcal{A}$  of length  $n$

**Output** : The partition of  $\mathcal{A}$  wrt. the randomized pivot  $p$  and its post-partitioning-index  $k$

[1]  $i \leftarrow \text{random\_uniform}(n - 1)$

[2] `swap`( $\mathcal{A}[0]$ ,  $\mathcal{A}[i]$ )

[3] **return** `partition`( $\mathcal{A}$ )

---

Neu ist also nur das Vertauschen des ersten Elements von  $\mathcal{A}$  mit einem durch `random_uniform` zufällig (gleichverteilt) gewählten Elements des Arrays, der Rest ist wie bei `partition`. Dabei hat `random_partition` die gleichbleibende Laufzeit  $T_{\text{rp}} \in \Theta(n)$  mit  $n = \text{len}(\mathcal{A})$ . Es ist auch nicht schwer einzusehen, dass hier  $n_0 = 2$  gewählt werden kann.

Der Algorithmus `randomized_quicksort` hat dabei genau den gleichen Pseudocode wie `quicksort`, nur dass `randomized_partition` statt `partition` in Zeile [2] gerufen wird. Die Laufzeit von `randomized_quicksort` mit Input der Länge  $n$  bezeichnen wir mit  $T_{\text{rqs}}(n)$ . Wir werden nun zeigen, dass  $T_{\text{rqs}} \in \mathcal{O}(n \log n)$  liegt.

#### Herleitung der average case Laufzeit von `randomized_quicksort`:

Für die erwartete Laufzeit gilt

$$T_{\text{rqs}}(n) = \sum_{k=1}^{n-1} P(k) (T_{\text{rqs}}(k) + T_{\text{rqs}}(n - k - 1) + T_{\text{rp}}(n)),$$

wobei  $P(k)$  die Wahrscheinlichkeit ist, dass `randomized_partition`( $\mathcal{A}$ ) den Index  $k$  liefert. Wir nehmen mit  $k \in \{0, \dots, n - 1\}$  eine Gleichverteilung  $P(k) = \frac{1}{n}$

an.

$$\begin{aligned}
T_{\text{rqs}} &= \sum_{k=0}^{n-1} \left( \frac{1}{n} (T_{\text{rqs}}(k) + T_{\text{rqs}}(n-k-1) + T_{\text{rp}}(n)) \right) \\
&= \frac{n}{n} T_{\text{rp}}(n) + \frac{1}{n} \sum_{k=0}^{n-1} T_{\text{rqs}}(k) + T_{\text{rqs}}(n-k-1) \\
&= T_{\text{rp}}(n) + \frac{2}{n} \sum_{k=0}^{n-1} T_{\text{rqs}}(k) \\
&\leq T_{\text{rp}}(n) + d + \frac{2}{n} \sum_{k=2}^{n-1} T_{\text{rqs}}(k).
\end{aligned}$$

Wobei  $d := 2T_{\text{rqs}}(0) + 2T_{\text{rqs}}(1) \leq \frac{2}{n}(T_{\text{rqs}}(0) + T_{\text{rqs}}(1))$  ist für  $n \geq 1$ .

Wir zeigen nun vermöge einer vollständigen Induktion, dass mit  $c = \max\{T_{\text{rq}}(2) + d, 8(d + c_{\text{rp}})\}$  gilt:

$$T_{\text{rqs}}(n) \leq c \cdot n \log_2 n \text{ für alle } n \geq 2.$$

Hierbei ist  $c_{\text{rp}}$  die Konstante mit welcher  $T_{\text{rp}}(n) \leq c_{\text{rp}}n$  gilt. Streng genommen gilt das erst ab irgendeinem  $n_0$ , aber den Aspekt vernachlässigen wir hier, um die Beweisstruktur etwas übersichtlicher zu gestalten. Es ist bei `randomized_partition` aber auch leicht ersichtlich, dass  $n_0 = 2$  gewählt werden kann.

Induktionsanfang  $n = 2$ :

$$\begin{aligned}
T_{\text{rqs}}(2) &\leq d + T_{\text{rp}}(2) \\
&\leq cn \log_2 2
\end{aligned}$$

Für  $c = \max(T_{\text{rqs}}(1) + T_{\text{rp}}(2), 8c_{\text{rp}}) \geq T_{\text{rqs}}(1) + T_{\text{rp}}(2)$  ist die Abschätzung definitiv erfüllt.

Induktionsschritt  $n-1 \mapsto n$ : Zu zeigen ist, dass mit mit der Aussage wahr für alle  $n \in \{2, \dots, n-1\}$  gilt:  $T_{\text{rqs}}(n) \leq cn \log_2 n$ . Wir rechnen:

$$\begin{aligned}
T_{\text{rqs}}(n) &\leq T_{\text{rp}}(n) + d + \frac{2}{n} \sum_{k=2}^{n-1} T_{\text{rqs}}(k) \\
&\stackrel{\text{IV}}{\leq} T_{\text{rp}}(n) + d \frac{2c}{n} \sum_{k=2}^{n-1} k \log_2 k \\
&\stackrel{(*)}{\leq} T_{\text{rp}}(n) + d + \frac{2c}{n} \left( (\log_2 n) \left( \frac{n(n-1)}{2} \right) - \frac{\frac{n}{2}(\frac{n}{2}-1)}{2} \right) \\
&= c(n-1) \log_2 n - c \left( \frac{n}{4} - \frac{1}{2} \right) + T_{\text{rp}}(n) + d \\
&\stackrel{(**)}{\leq} c \cdot n \log_2 n.
\end{aligned}$$

Betrachten wir zuerst (\*\*\*) genauer: Das gilt genau dann, wenn

$$T_{\text{rp}}(n) + d \leq c \left( \frac{n}{4} - \frac{1}{2} \right) + \log_2 n.$$

Mit  $n \geq 2$  ist  $\log_2 n > \frac{1}{2}$  und  $T_{\text{rp}}(n)$  lässt sich nach Annahme von oben durch  $c_{\text{rp}}n$  abschätzen. Damit vereinfacht sich die Ungleichung auf

$$4(c_{\text{rp}}n + d) \leq cn,$$

was mit obigen  $c$  ab  $n \geq 2$  erfüllt ist.

Jetzt gilt es nun noch, die in Schritt (\*) getroffene Abschätzung der Summe  $\sum_{k=1}^{n-1} k \log_2 k$  zu beweisen:

$$\begin{aligned} \sum_{k=2}^{n-2} k \log_2 k &\leq \sum_{k=1}^{n-1} k \log_2 k \\ &= \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} \underbrace{k \log_2 k}_{\leq \log_2 \frac{n}{2}} + \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} \underbrace{k \log_2 k}_{\leq \log_2 n} \\ &\leq \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k(\log_2 n - 1) + \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \log_2 n \\ &= \log_2 n \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log_2 n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \\ &= \log_2 n \underbrace{\sum_{k=1}^{n-1} k}_{=\frac{n(n-1)}{2}} - \underbrace{\sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k}_{\geq \frac{\frac{n}{2}(\frac{n}{2}-1)}} \\ &\leq (\log_2 n) \left( \frac{n(n-1)}{2} \right) - \frac{\frac{n}{2}(\frac{n}{2}-1)}{2} \end{aligned}$$

Unsere vollständige Induktion ist damit bewiesen, und wir haben gezeigt, dass die durchschnittliche Laufzeit von `randomized_quicksort` in  $\mathcal{O}(n \log n)$  liegt.

## 2.3 Die untere Laufzeitschranke für vergleichsbasierte Sortieralgorithmen

Wir haben in der analyse von `mergesort` gesehen, dass die worst-case-Laufzeit  $\mathcal{O}(n \log n)$  nicht unterschreitet. Es ist auch die best-case-Laufzeit von `quicksort`. Schneller als  $\mathcal{O}(n)$  kann ein Sortieralgorithmus sicherlich nicht sortieren: Er muss ja jeden Input mindestens einmal angeschaut haben. Aber geht es, im worst-case, schneller als  $\mathcal{O}(n \log n)$ ?

Die Antwort ist - für vergleichsbasierte Algorithmen - nein, und wir werden in diesem Kapitel den Beweis dazu skizzieren.

Doch was ist ein vergleichsbasierter Algorithmus? All unsere bisherigen Sortieralgorithmen verglichen stets zwei Elemente, um ihre entsprechende Reihenfolge untereinander zu bestimmen. Das impliziert, dass die Elemente eine sogenannte totale Quasiordnung haben mussten;

**Definition 6** (totale Quasiordnung). Sei  $A$  eine Menge. Eine totale Quasiordnung auf  $X$  ist eine binäre Relation  $\leq$ , für welche gilt:

- $a \leq a$  für alle  $a \in A$  (*Reflexivität*).
- $a \leq b$  und  $b \leq c$  impliziert  $a \leq c$  für alle  $a, b, c \in A$  (*Transitivität*).
- $a \leq b$  oder  $b \leq a$  für alle  $a, b \in A$  (*Totalität*).

Insbesondere erfüllt jede totale Ordnung (also beispielsweise die normale Ordnung auf den ganzen oder reellen Zahlen) die Axiome der totalen Quasiordnung. Für eine "echte" Quasiordnung betrachten wir  $X = \{x, y, z\}$ , mit  $\leq$  definiert durch  $y \leq x$ ,  $y \leq z$ ,  $x \leq z$  und  $z \leq x$ . Nun sind  $x$  und  $z$  "gleich", wir benutzen das Symbol  $x \equiv z$ . Formaler aufgeschrieben:  $x \leq z \wedge z \leq x \implies x \equiv z$ .

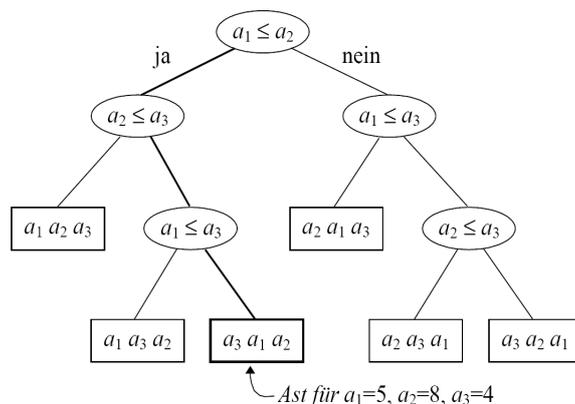
Für ein praktisches Beispiel können wir Spielkarten anschauen, die wir allein nach den numerischen Werten ordnen. Dann ist mit  $x = \text{Herz } 8$ ,  $y = \text{Herz } 7$  und  $z = \text{Pik } 8$  klar, dass  $x \equiv z$ , aber nicht unbedingt  $x = z$ , wie es bei einer totalen Ordnung der Fall wäre.

Vergleichsbasierte Sortieralgorithmen vergleichen daher nur immer genau zwei Elemente miteinander. Dies geschieht bei `quicksort` beispielsweise in Zeile 4 der Partition, oder bei `mergesort` in Zeile 4 von `merge`.

### 2.3.1 Beweisskizze

Wir betrachten im Folgenden die Anzahl der nötigen Vergleiche für das Sortieren des Inputs. Da der Aufwand eines vergleichsbasierten Sortieralgorithmus mindestens so schnell wächst wie die Anzahl der nötigen Vergleiche, haben wir damit eine untere Schranke für den zeitlichen Aufwand.

Der Kontrollfluss eines vergleichenden Sortierverfahrens kann mittels eines sogenannten *Entscheidungsbaummodells* dargestellt werden. In einem Entscheidungsbaummodell werden alle möglichen und nötigen Entscheidungen dargestellt, um ein sortiertes lineares Feld zu erhalten. Beispiel: Es liegt eine Sequenz von 3 Zahlen vor  $[a_1, a_2, a_3]$ , ganz konkret also möglicherweise  $[5, 8, 2]$ . Dann schaut der zugehörige Entscheidungsbaum folgendermaßen aus:



Die Blätter stellen alle möglichen Permutationen des Inputs dar. Die Anzahl der Blätter ist daher gleich  $n!$ .

Das worst-case Verhalten eines Algorithmus entspricht dem längsten Ast im Entscheidungsbaum, die Anzahl der inneren Knoten ist dabei genau die Anzahl der notwendigen Vergleiche.

Ein idealer Sortieralgorithmus entspricht einem vollständigen, ausgeglichenen Baum. Dadurch wird der längste Ast, also der worst case, minimiert. Die Höhe beträgt  $h \geq \log_2(n!)$ . Aber wir haben bereits in einer Übungsaufgabe berechnet, dass gilt:

$$h \geq \log_2(n!) \in \Theta(n \log n).$$

Also sind wir damit fertig.

**Satz 3.** Für vergleichsbasierte Sortieralgorithmen liegt die worst-case Laufzeit immer in

$$\Omega(n \log_n).$$

## 2.4 Radixsort

TODO: Das Kapitel braucht mehr Bilder

Der Sortieralgorithmus **radixsort** ist ein Beispiel für ein nicht vergleichsbasiertes Sortierverfahren. Nicht vergleichsbasierte Sortierverfahren arbeiten ohne Vergleiche und müssen daher zusätzliche Annahmen über den Input machen. Wir fordern also, dass unser aus  $n$  Elementen bestehender Input jeweils eine gleiche Anzahl von Stellen  $d \in \mathbb{N}$  hat. Pro Stelle haben wir dabei  $b$  verschiedene Möglichkeiten. Beispielsweise besteht ```beispiel''` aus  $d = 8$  Buchstaben, von denen es  $b = 26$  gibt. Die Dezimalzahl 9215 hat hingegen  $d = 4$  Stellen mit jeweils  $b = 10$  Möglichkeiten. Wir referenzieren auf die  $i$ . Stelle von  $x$  mit  $x_i$ . Beispielsweise ist mit  $x = 423$ :  $x_0 = 4$ ,  $x_2 = 3$ .

Hat man Zahlen/Wörter von unterschiedlicher Länge, muss man sie von der passenden Seite mit Lückenfüllern auffüllen, im Fall von Zahlen also von links mit 0.

Die Idee von **radixsort** ist nun, von der unsignifikantesten Stelle (bei uns an letzter Stelle) ausgehend bis zur signifikantesten (ersten) Stelle immer jeweils einmal zu *streuen* und dann zu *sammeln*. In der Streuphase werden die Elemente in die jeweiligen Kategorien einsortiert (unter strikter Einhaltung einer vorher vorhandenen Reihenfolge). In der Sammelphase werden alle Kategorien entsprechend ihrer Ordnung untereinander wieder konkateniert.

---

### Algorithmus 18 : radixsort( $Q$ )

---

**Input** : An unsorted Queue  $Q$  of length  $n$   
**Output** :  $Q$ , but sorted

```

[1] Initialize Queues  $Q_0, \dots, Q_{b-1}$ 
[2] for  $i \leftarrow d - 1$  to 0 do
[3]   while  $Q$  is not empty do
[4]      $x \leftarrow \text{get}(Q)$ 
[5]     put( $Q_{x_i}, x$ )
[6]   for  $j \rightarrow 0$  to  $b - 1$  do
[7]     while  $Q_j$  is not empty do
[8]       put( $Q, \text{get}(Q_j)$ )
[9] return  $Q$ 

```

---

Die Zeilen 3-5 werden *Streuphase* genannt, und Zeile 6-8 bezeichnet man auch als *Sammelphase*. Der Algorithmus **radixsort** ist korrekt, da nach den ersten  $i$  Durchläufen von Zeile 2 die Elemente, eingeschränkt auf die letzten  $i$  Stellen, sortiert sind. Eine wichtige Bedingung für die Korrektheit der Sammelphase ist, dass die vorige Reihenfolge innerhalb der Fächer aufrechterhalten wird.

Die Laufzeit von **radixsort** beträgt  $T_{\text{rs}}(n) = \mathcal{O}(d \cdot n)$ , ist also linear, wenn die Anzahl der Stellen  $d$  als konstant betrachtet wird. In diesem Fall ist **radixsort** asymptotisch schneller als vergleichsbasierte Sortierverfahren, hat aber einen deutlich erhöhten Speicherbedarf, welcher insbesondere linear in der Anzahl der Fächer  $b$  wächst.

## 2.5 Eigenschaften von Sortieralgorithmen

### 2.5.1 Laufzeit

Wir haben bereits umfangreiche Laufzeitanalysen von Sortieralgorithmen durchgeführt. Die Qualität eines Sortieralgorithmus anhand seines best/worst/average-case-Verhaltens zu beurteilen, scheint also naheliegend.

### 2.5.2 worst-case-optimal

Im Zusammenhang von vergleichsbasierten Sortieralgorithmen bedeutet die Eigenschaft *worst-case-optimal*, dass die Untergrenze von  $\Omega(n \log n)$  immer eingehalten wird.

### 2.5.3 Stabilität

Ein Sortieralgorithmus ist stabil, wenn durch das Sortieren die Reihenfolge innerhalb bezüglich der Quasiordnung gleicher Elemente gewährleistet ist. Formaler:

**Definition 7** (Stabilität). Sei  $[x_0, x_1, \dots, x_{n-1}]$  ein Array bzw eine Sequenz von  $n$  Elementen aus der Menge  $X$ . Unterhalte  $X$  eine totale Quasiordnung, unter derer ‘‘Gleichheit’’ von  $x, x' \in X$  mit  $x \equiv x'$  bezeichnet wird. Sei das Wirken des Sortieralgorithmus  $f$  eine Permutation  $\pi \in \text{Perm}(n)$ , also  $f([x_0, \dots, x_{n-1}]) = [x_{\pi(0)}, x_{\pi(1)}, \dots, x_{\pi(n-1)}]$ .

Dann ist unser Sortieralgorithmus stabil, wenn gilt:

$$x_i \equiv x_{i'} \text{ und } i \leq i' \text{ impliziert } \pi(i) \leq \pi(i').$$

### 2.5.4 in-place

Als *in-place* bezeichnet man die Eigenschaft eines Sortieralgorithmus, neben dem Speicherverbrauch des Inputarrays nur einen konstanten zusätzlichen Speicherverbrauch zu haben. Es ist also  $S_{\text{alg}} \in \mathcal{O}(1)$ . Gelegentlich wird auch Speicherverbrauch von  $\mathcal{O}(\log n)$  noch als *in-place* bezeichnet.

### 2.5.5 Adaptivität

Als adaptiv bezeichnet man einen Sortieralgorithmus, der kürzere Laufzeiten hat, wenn Teile des Arrays bereits sortiert sind. Insbesondere muss der best-case also in  $\mathcal{O}(n)$  liegen. Diese Definition lässt sich mittels Fehlstellungen präzisieren:

**Definition 8** (Fehlstände und Fehlstandszahl). Sei  $A$  eine geordnete Menge und sei  $[a_1, \dots, a_n] \in A^n$  eine Folge von Elementen in  $A$ . Die Anzahl der Fehlstände bezüglich  $a_i$  ist

$$f_i := |\{a_j : j > i, a_j < a_i\}|,$$

also alle Elemente die rechts von  $a_i$  liegen, obwohl sie bei einer geordneten Folge links von  $a_i$  liegen sollten. Die Summe aller Fehlstände einer solchen Folge ist die Fehlstandszahl

$$F([a_1, \dots, a_n]) := \sum_{i=1}^n f_i.$$

Die Fehlstandsanzahl ist  $F([1, 2, 3, \dots, n]) = 0$ , falls wir eine vollständig sortierte Folge haben, ist bei einer einzelnen Vertauschung von zwei Nachbarn 1 und wir bei einem invers sortierten Array maximiert:  $F([n, n-1, \dots, 1]) = \sum_{i=1}^{n-1} i = \frac{n^2-n}{2}$ .

Adaptiv im strengeren Sinne bedeutet nun, dass sich die (asymptotische) Laufzeit auch in Abhängigkeit zu den Fehlständen ausdrücken lässt und geringer wird, wenn weniger Fehlstände vorliegen.

Allerdings wird die Definition von adaptiv landläufig auch etwas lockerer gehandhabt und umfasst auch das Erkennen invers sortierter Teilsequenzen. Diese stellen in unserer Definition aber gerade den maximal schlimmsten Fall dar.

TODO: Beispiel `insertionsort`.

### 2.5.6 Parallelisierbarkeit

Sowohl auf dem PC, als auch im Smartphone und insbesondere in einem Rechenzentrum wird zusätzliche Rechenleistung fast nur noch durch parallele Architekturen erreicht. Damit gewinnt auch die Frage nach Parallelisierbarkeit größere Bedeutung. Aus einer theoretischen Perspektive nimmt man dabei oftmals unbegrenzt viele parallele Prozessoren an.

TODO: Tabelle

# Kapitel 3

## Datenstrukturen

### 3.1 Dynamische Arrays

Wir haben in Kapitel 1.3 bereits Arrays kennengelernt. Arrays sind aber sehr statisch: Am Anfang wird ihre Größe  $n$  festgelegt, danach verändert sich diese nichtmehr.

Wir können zwar auf alle Elemente  $\mathcal{A}[0], \dots, \mathcal{A}[n-1]$  mit konstantem Aufwand zugreifen und diese auch verändern. Aber neue Elemente hinzufügen ist unmöglich.

In einer Liste  $\mathcal{L}$  hingegen ist das Hinzufügen von neuen Elementen kein Problem. Dafür kostet der Zugriff auf das  $i$ . Listenelement  $\mathcal{O}(i)$  Zeit, da wir uns vom Anfang der Liste bis zu  $i$  durcharbeiten müssen.

Was ist aber, wenn wir beides wollen? Zugriff in konstanter Zeit und dynamisches Wachstum? Das dynamische Array ist eine mögliche Lösung für das Problem.

Das Grundprinzip ist dabei sehr einfach. Ein dynamisches Array ist ein Array  $\mathcal{A}$ , welches immer größer ist ( $n_{\text{cap}}$  Felder) als das Array, was wir gerade brauchen ( $n$  Felder). Wir schreiben also auf die ersten  $n$  Plätze wie gewohnt. Wollen wir nun ein neues Element hinzufügen, schreiben wir es auf das  $n+1$ -te Feld und inkrementieren  $n$  um eins. Irgendwann ist jedoch auch das benutzte Array so groß wie das darunterliegende Array ( $n = n_{\text{cap}}$ ). Bevor wir nun also weitere Elemente hinzufügen können, müssen wir das echte Array im Hintergrund durch ein größeres (zum Beispiel  $n_{\text{cap}} \leftarrow 2n_{\text{cap}}$ ) ersetzen. Diese Operation ist aufwendig, geschieht aber selten. Diesen Umstand machen wir uns dann später in der sogenannten amortisierten Analyse zunutze.

**Anwendungen** Dynamische Arrays sind weit verbreitet. Beispielsweise ist die Implementierung der List-Datenstruktur in Python ein dynamisches Array. Aber auch in C++, Java und Rust ist verstecken sich hinter `vectors` oder generischen Listenobjekten dynamische Arrays.

#### 3.1.1 Operationen auf dynamischen Arrays

**Definition 9** (dynamisches Array). Ein dynamisches Array  $\mathcal{DA}$  ist ein Tupel  $(\mathcal{A}, n)$ , wobei  $\mathcal{A}$  ein Array von Länge  $n_{\text{cap}}$  ist und  $n \in \{0, \dots, n_{\text{cap}}-1\}$  ein Zeiger auf das aktuell letzte Element.

### Zugriff auf Arrayelemente

Möchten wir nun auf das  $i$ . Element zugreifen, wird zuerst geprüft, ob das zulässig ist. Falls ja, fallen wir auf die normale Arrayoperation zurück:

$$\mathcal{DA}[i] = \begin{cases} \mathcal{A}[i] & \text{if } i \in \{0, \dots, n-1\}, \\ \text{error: out\_of\_bounds} & \text{otherwise.} \end{cases}$$

### Anhängen neuer Elemente

Wir können in einem dynamischen Array auch Elemente hinzufügen. Hier wird nur das Einfügen am Ende behandelt. Möchte man beliebig (Index  $i$ ) einfügen, muss man mit Aufwand  $\mathcal{O}(n-i)$  alles nach hinten verschieben.

Wie obig erwähnt, gibt es zwei Fälle. Im ersten reicht der reservierte Platz noch aus ( $n \leq n_{\text{cap}}$ ). Dann schreiben wir einfach das neue Element ins Array und inkrementieren  $n$ . Diese Operation hat Aufwand  $\mathcal{O}(1)$ . Im zweiten Fall müssen wir zuerst ein neues, doppelt so großes Array anlegen ( $n_{\text{cap}} \leftarrow 2n_{\text{cap}}$ ) und alle bisherigen Elemente ins neue Array kopieren. Das braucht  $\mathcal{O}(n)$  Zeit. Danach tritt der erste Fall ein.

---

#### Algorithmus 19 : $\text{add}(\mathcal{DA}, x)$

---

**Input :** A dynamic array  $\mathcal{DA}$  and the element  $x$  to be added

**Output :**  $\mathcal{DA}$  with  $x$  appended at the end

```
[1] if  $n < n_{\text{cap}}$  then
[2]    $\mathcal{A}[n] \leftarrow x$ 
[3]   return  $(\mathcal{A}, n + 1)$ 
[4] else
[5]    $\mathcal{A}_{\text{old}} \leftarrow \mathcal{A}$ 
[6]    $\mathcal{A} = \text{new Array of size } 2n_{\text{cap}}$ 
[7]    $\mathcal{A}[0, \dots, n-1] \leftarrow \mathcal{A}_{\text{old}}[0, \dots, n-1]$ 
[8]   free $(\mathcal{A}_{\text{old}})$ 
[9]   return  $\text{add}((\mathcal{A}, n), x)$ 
```

---

Der von  $\mathcal{A}_{\text{old}}$  belegte Speicherplatz kann nun freigegeben werden (manuell via **free** oder durch einen garbage collector).

### Löschen

Das Löschen geschieht analog zum Einfügen nur am Ende. Möchten wir in der Mitte löschen, müssen wir danach alle Elemente nach vorne schieben (Wieder mit Aufwand  $\mathcal{O}(n-i)$ ).

Um nicht unnötig Speicherplatz zu blockieren, verkleinern wir das Array auf die Hälfte, sobald es zu weniger als einem Viertel belegt ist. Die Verkleinerungsoperation braucht wieder  $\mathcal{O}(n)$  Zeit.

---

**Algorithmus 20** : delete( $\mathcal{DA}$ )

---

**Input** : A dynamic array  $\mathcal{DA}$ **Output** :  $\mathcal{DA}$  without the last element

```

[1] if  $n > \frac{n_{\text{cap}}}{4}$  then
[2]   return ( $\mathcal{A}, n - 1$ )
[3] else
[4]    $\mathcal{A}_{\text{old}} \leftarrow \mathcal{A}$ 
[5]    $\mathcal{A} = \text{new Array of size } \frac{n_{\text{cap}}}{2}$ 
[6]    $\mathcal{A}[0, \dots, n - 1] \leftarrow \mathcal{A}_{\text{old}}[0, \dots, n - 1]$ 
[7]   free( $\mathcal{A}_{\text{old}}$ )
[8]   return delete( $(\mathcal{A}, n)$ )

```

---

### 3.1.2 Speicherverbrauch

Durch den Algorithmus `add` ist garantiert, dass  $n \leq n_{\text{cap}}$ . Durch den Algorithmus `delete` ist garantiert, dass  $n_{\text{cap}} \leq 4n$ . Dadurch ist garantiert, dass  $n \leq n_{\text{cap}} \leq 4n$ , wir “verschwenden” also stets höchstens  $\Theta(n)$  Speicherplatz.

### 3.1.3 Amortisierte Laufzeitanalyse

Die amortisierte Laufzeitanalyse betrachtet die Laufzeit von  $k$  konsekutiven Operationen hintereinander und errechnet damit die durchschnittliche amortisierte Laufzeit für eine Einzeloperation. Prinzipiell ähnelt sie damit also der average-case-Analyse, jedoch ohne die Ungewissheit durch den involvierten Zufall.

Wie wir gerade gesehen haben, haben sowohl `add` als auch `delete` einen worst-case-Aufwand von  $\mathcal{O}(n)$ . Nehmen wir nun an, dass wir mit einem leeren dynamischen Array starten (die anfängliche Größe setzen wir auf  $n_{\text{cap}} = n_0 = 1$ ).

Welches Laufzeitverhalten würden wir nun von  $k$  hintereinander durchgeführten Einfügeoperation erwarten? Pessimistisch, rein nach dem worst-case-Verhalten, wären es  $\sum_{n=0}^k f(n)$  mit  $f \in \mathcal{O}(n)$ . Damit würden die Kosten für  $k$  Einfügeoperationen in  $\mathcal{O}(k^2)$  liegen. Wir wissen jedoch mit Sicherheit, dass wir nach einer Einfügeoperation mit Kopieraufwand der Größe  $n$  weitere  $n$  Elemente mit konstantem Aufwand einfügen können, da das Array  $\mathcal{A}$  hinter  $\mathcal{DA}$  ja nun doppelt so viel Platz hat. Damit kommen wir bei genauerer Betrachtung auf einen amortisierten Aufwand von  $\mathcal{O}(k)$  für  $k$  Operationen.

Im Unterschied zur average-case-Analyse haben wir bei einer amortisierten Analyse also *garantiert*, dass  $k$  subsequente Operationen nicht jedes mal im worst case landen.

**Definition 10** (Amortisierte Laufzeit). Seien  $o_1, o_2, \dots$  endlich viele verschiedene Operationen. Der *amortisierte* Aufwand von Operationenfolgen  $f_{\bullet} \in \{o_1, o_2, \dots\}^{\mathbb{N}}$  der Länge  $n$  wird definiert als:

$$T_{\text{amo}}(n) = \frac{\max(\{\sum_{i=1}^n T(f_{\bullet_i}) \mid f_{\bullet} \in \{o_1, o_2, \dots\}^{\mathbb{N}}\})}{n}.$$

Im Kontext der dynamische Arrays haben wir zwei Operationen,  $o_1 = \text{add}(\mathcal{DA}, x)$  und  $o_2 = \text{delete}(\mathcal{DA})$ , wobei der hinzugefügte Wert  $x$  keinerlei Auswirkungen auf die Laufzeit hat.

### Aggregierte Analyse

Zuerst betrachten wir die Einzeloperationen `add` und `delete` separat. Das entspricht also Funktionsfolgen  $f_\bullet = (\text{add}, \text{add}, \dots)$  bzw.  $f_\bullet = (\text{delete}, \text{delete}, \dots)$ .

Der Einfachheit halber weisen wir den Zeilen [1] bis [3] von `add` den Aufwand  $f(n) = 1$  und den Zeilen [4] bis [8] den Aufwand

$$g(n) = \begin{cases} n - 1 & \text{falls } n = 2^x + 1 \text{ für ein } x \in \mathbb{N} \\ 0 & \text{sonst} \end{cases}$$

zu, siehe Abbildung 3.1.

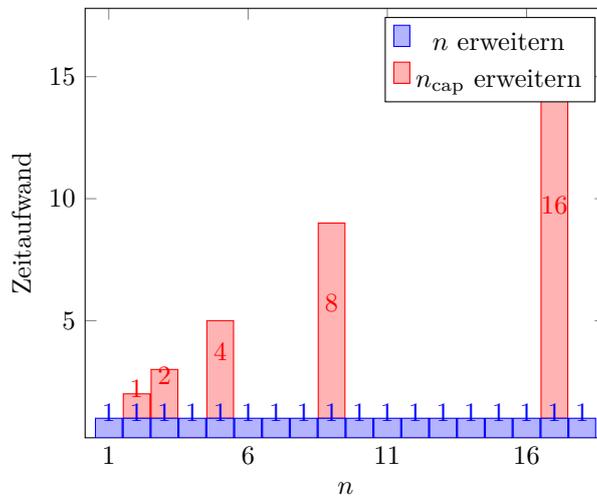


Abbildung 3.1: Wir sehen die Laufzeit für das Einfügen in ein dynamisches Array. Auf der  $x$ -Achse die Anzahl der Elemente, auf der  $y$ -Achse die Laufzeit. Der Einfachheit halber wurde ein Aufwand von 1 für das einfache Einfügen und ein Aufwand von  $n$  für das Kopieren von  $n$  Elementen angenommen.

Wenn wir mit einem leeren dynamischen Array anfangen, haben nun  $k$  Aufrufe von `add` den Aufwand:

$$\begin{aligned} \sum_{n=0}^k T_{\text{add}}(n) &= \sum_{n=0}^k f(n) + \sum_{n=0}^k g(n) \\ &= k + \sum_{i=0}^{\lfloor \log_2(k-1) \rfloor} 2^i \\ &= k + 2^{1+\lfloor \log_2(k-1) \rfloor} \\ &\leq k + 2(k-1) \\ &\leq 3k \in \mathcal{O}(k) \end{aligned}$$

Ein einzelner Aufruf von `add` hat damit einen amortisierten Aufwand von  $\mathcal{O}(1)$ .

Diese Rechnung lässt sich analog auf  $k$  konsekutive `delete`-Operation übertragen, solange wir voraussetzen, dass das dynamische Array zu Beginn  $k \leq n_{\text{cap}} \leq 4k$  groß ist.

**Accountingmethode**

Wir wollen nun untersuchen, was passiert, wenn wir zwischen Löschen und Einfügen nach Belieben wechseln.

In der aggregierten Analyse von **add** haben wir einfach  $k$  Operationen von **add** subsequent ausgeführt, den kumulativen Aufwand berechnet und dann durch  $k$  geteilt, um den Aufwand pro Schritt zu berechnen. Das ging einfach und direkt, da es nur eine Operationenfolge  $f_\bullet$  gab:  $(\text{add}, \text{add}, \dots, \text{add})$ . Nun sind wir aber an gemischten Operationenfolgen von **add** und **delete** interessiert. Für  $k = 3$  gibt es schon 8 mögliche Operationenfolgen:  $(\text{add}, \text{add}, \text{add})$ ,  $(\text{add}, \text{add}, \text{delete})$ , .... Über all diese Möglichkeiten zu gehen und den maximalen durchschnittlichen Wert einer Sequenz zu bestimmen, wäre sehr mühselig.

Wir gehen daher einen anderen Weg, die Accountingmethode. Hier wird der Datenstruktur selber ein Zeitkonto  $K$  zugeordnet, in welche Operationen ein- und auszahlen. Schnelle Operationen wie ein **add** ohne Erweiterung von  $\mathcal{A}$  zahlen dann mehr ein als sie kosten, wird dann aber eine Erweiterung von  $\mathcal{A}$  fällig, wird sie aus dem angespartem Guthaben bezahlt.

Wir vereinfachen die Laufzeit von **add** und **delete** wie bei der aggregierten Analyse und haben damit folgende Tabelle für die Kosten und Einzahlungen:

Laufzeit der $i$ -ten Operation $f_i$	Laufzeitkosten $a_i$	Einzahlung $e_i$
<b>add</b> (ohne Umstrukturierung)	1	3
<b>add</b> (mit Umstrukturierung)	$n_i$	2
<b>delete</b> (ohne Umstrukturierung)	1	3
<b>delete</b> (mit Umstrukturierung)	$n_i$	2

Wir betrachten dabei alles zu Zeitpunkten  $i \in \mathbb{N}$ . Dabei ist  $f_i$  die Operation zum  $i$ . Zeitpunkt,  $n_i$  bezeichnet die Anzahl der Elemente im  $\mathcal{DA}$  zum Zeitpunkt  $i$  (vor dem Ausführen der Operation  $f_i$ ),  $a_i$  sind die dazu assoziierten (echten) Laufzeitkosten und  $e_i$  die dafür eingezahlten Beträge. Der Kontostand schwankt auch mit der Zeit und wird daher indiziert.

Dadurch, dass das Konto der Datenstruktur zugerechnet wird, und nicht nur einer speziellen Operation, können wir jetzt auch gemischte Operationsreihenfolgen analysieren. Solange der Kontostand nicht negativ wird, sind die Einzahlungen wohl eine ausreichende Abschätzung für die tatsächlichen Laufzeitkosten.

Wir beobachten zuerst: Die eingezahlten Kosten vor einer Umstrukturierung decken die echten Kosten dieser Umstrukturierung:

**Lemma 4.** *Sei eine Operationenfolge  $f_\bullet$  gegeben. Seien  $i < j$  zwei Zeitpunkte, bei denen eine Umstrukturierung stattfindet. Dann ist  $K_i \leq K_j$ .*

*Beweis.* Wir beobachten zuerst: Direkt nach einer Umstrukturierung, als zu Beginn von Zeitpunkt  $i + 1$  gilt:

$$n_{i+1} = \frac{n_{\text{cap}}}{2}. \tag{3.1}$$

Nun gibt es zwei Fälle für die Umstrukturierung zum Zeitpunkt  $j$ :

Fall 1: Die Umstrukturierung ist eine Erweiterung, es ist also  $f_j$  gleich **add**. Dann gibt es mindestens  $n_{i+1}$  **add**-Operationen zwischen Zeitpunkt  $i$  und  $j$ , da

ansonsten  $n$  nicht genug gewachsen ist, um eine Erweiterung anzustoßen. Der Kontostand wächst daher bis zum Zeitpunkt  $j - 1$ :

$$K_{j-1} = K_i + \sum_{t=i+1}^{j-1} (e_t - a_t) \geq K_i + 3(n_{i+1} - 1) - 1(n_{i+1} - 1) = K_i + 2n_{i+1} - 2.$$

Die Abschätzung  $\geq$  rechtfertigt sich dadurch, dass zwischen  $i$  und  $j$  durchaus mehr als  $n_{i+1} - 1$  Zeitschritte liegen können. Beispielsweise dann, wenn in der Operationenfolge auch mal ein `delete` auftaucht, was dann ein weiteres `add` zum Ausgleich braucht.

Im Schritt  $j$  wird nach Voraussetzung eine Umstrukturierung fällig. Daher betragen die Kosten  $a_j = n_{\text{cap}} = 2n_{i+1}$ , während die Einzahlung  $e_j = 2$  beträgt. Das entspricht aber genau der vorher angesparten Summe:

$$K_j = K_{j-1} + e_j - a_j \geq K_i.$$

Fall 2, die Umstrukturierung in Form einer Verschmälerung verläuft analog und ist eine Übungsaufgabe.  $\square$

Damit können wir zeigen, dass das dynamische Array einen amortisierten Aufwand von  $\mathcal{O}(1)$  hat:

**Satz 4.** *Das dynamische Array hat einen asymptotischen Aufwand von  $\mathcal{O}(1)$  für `add`- und `delete`-Operationen, auch bei gemischter Reihenfolge.*

*Beweis.* Sollten keinerlei Umstrukturierungen nötig sein, ist  $a_i < e_i$  klar. Aber auch bei Umstrukturierungen ist aus dem vorherigen Lemma klar: Ist  $K_0 = 0$  und damit nicht negativ, bleibt  $K_i \leq 0$  für alle  $i \in \mathbb{N}$ . Da die tatsächliche Laufzeit  $T(f_i) = a_i$ , haben wir also

$$\sum_{i=1}^n T(f_i) = \sum_{i=1}^n a_i \leq \sum_{i=1}^n e_i \leq 3n.$$

Das gilt für alle möglichen Operationenfolgen, also auch für die zeitaufwendigste. Nach Definition ist damit ist Laufzeit pro Operation

$$T_{\text{amo}}(n) = \frac{\max(\{\sum_{i=1}^n T(f_i) \mid f_{\bullet} \in \{\text{add}, \text{delete}\}^{\mathbb{N}}\})}{n} \leq \frac{3n}{n} = 3 \in \mathcal{O}(1).$$

$\square$

## 3.2 Hashtabellen

Die Datenstruktur einer Hashtabelle wird zur Lösung des sogenannten Wörterbuchproblems verwendet. Ein Wörterbuch ist eine Menge an Elementen  $(s, d)$ , bei welchem die Nutzdaten  $d$  über einen eindeutigen Schlüssel  $s$  identifizierbar sind.

Das Wörterbuchproblem ist durch die drei Anforderungen *Einfügen*, *Suchen* und *Löschen* definiert. Beispiele wären Telefonbücher, Wörterbücher, Adressbücher, aber auch Symboltabellen beim Compilieren oder allgemeine Zuordnungstabellen.

### 3.2.1 Grundidee: Gestreute Speicherung

Statt aufwendig zu suchen und zu sortieren, findet man den gewünschten Ort der Daten durch ihren Schlüssel. Hierbei wird nicht, wie beim Sortieren und Suchen in Kapitel 2, nur eine Quasiordnung auf dem Schlüssel ausgenutzt, sondern der (ungefähre) Speicherort in einem Array der Größe  $m$  wird vermöge einer Hashfunktion direkt, mit Aufwand  $\mathcal{O}(1)$ , aus dem Schlüssel  $s$  berechnet.

**Definition 11** (Hashfunktionen). Sei  $S$  eine Menge von Schlüsseln und  $m \in \mathbb{N}_+$  eine positive natürliche Zahl. Dann ist eine Hashfunktion eine Funktion

$$h : S \rightarrow \{0, \dots, m - 1\},$$

welche mit konstantem Aufwand berechnet werden kann.

Eine ideale Hashfunktion  $h_i$  ist gleichverteilt (Die Funktion  $\Pr$  gibt die Wahrscheinlichkeit des Ereignisses zurück):

$$\Pr(h_i(s) = k) = \frac{1}{m} \text{ unabhängig von } s \in S \text{ und } k \in \{0, \dots, m - 1\}.$$

Eine perfekte Hashfunktion  $h_p$  ist zusätzlich injektiv.

Den Wert  $h(s)$  nennt man den Hash von  $s$ , bzw den Hash vom Element  $(s, d)$ . Die Nutzdaten werden nun üblicherweise in einem Array von Länge  $m$  abgespeichert, der Hash ist also der Index in diesem Array.

Das Gegenstück zu einer idealen Hashfunktion, die ihrer Werte gleichverteilt über ihrem Wertebereich streut, ist die konstante Hashfunktion  $h_c(s) = 0$  für alle  $s \in S$ . Von (nicht perfekten) Hashfunktionen wird nämlich nicht erwartet, dass sie injektiv sind! Das führt zu einem grundlegenden Problem von Hashtabellen, den Kollisionen. Eine Kollision ist gegeben, wenn zwei unterschiedliche Schlüssel den gleichen Hash haben, also  $h(s) = h(s')$ , obwohl  $s \neq s'$ . Auch bei einer idealen Hashfunktion kommt es spätestens dann zu diesem Effekt, wenn die Anzahl der Elemente  $n$  sich der Anzahl der Felder  $m$  annähert. Wir nennen den Quotienten

$$\alpha := \frac{n}{m}$$

den *Belegungsfaktor*.

Wir betrachten nun die zwei bekanntesten Strategien um mit Kollisionen umzugehen, Überläuferlisten und offene Adressierung mittels Sondierung. Danach betrachten wir, warum und wie wir die Wahrscheinlichkeit für Kollisionen minimieren sollten und können.

### 3.2.2 Kollisionsbehandlung mit Überlauferliste (Chaining)

Die Idee liegt darin, Elemente, die zum gleichen Index führen, als verkettete Liste zu organisieren. Das heißt, dass in der Hashtabelle in Feld  $j$  nur der Kopf einer Liste steht. Konzeptionell ist eine Hashtabelle mit Überläuferlisten der Größe  $m$  also eigentlich kein Array, sondern eine Komposition aus einer  $m$ -wertigen Hashfunktion  $h$  und  $m$  Listen:  $\mathcal{T} = (h, \mathcal{L}_0, \dots, \mathcal{L}_{m-1})$ . In einer konkreten Implementierung, insbesondere dann, wenn die Kollisionen nur sehr selten auftreten, können wir ein Array nutzen und die Listen nur im Kollisionsfall dazuschalten.

Die drei nötigen Funktionen zur Lösung des Wörterbuchproblems können folgendermaßen implementiert werden:

- Einfügen von  $(s, d)$ : Der Index  $j = h(s)$  wird vermöge der Hashfunktion aus dem Schlüssel  $s$  berechnet. Der Datensatz  $(s, d)$  wird dann am Anfang der Liste abgespeichert. Die Laufzeit ist immer  $\mathcal{O}(1)$ .
- Suchen nach  $s$ : Der Index  $j = h(s)$  wird vermöge der Hashfunktion aus dem Schlüssel  $s$  berechnet. Die lokale Liste  $\mathcal{L}_j$  wird nach dem Schlüssel durchsucht. Die Laufzeit des Suchens ist dabei abhängig von der Länge  $l_j := \text{len}(\mathcal{L}_j)$  der lokalen Überläuferliste. Da wir keinerlei Ordnung auf der Liste haben, müssen wir alle Elemente nacheinander abgehen und den Schlüssel vergleichen. Der asymptotische Aufwand ist  $\mathcal{O}(l_j)$ .
- Löschen des Elements mit Schlüssel  $s$ : Das Löschen basiert auf dem Suchen. Das tatsächliche Löschen aus der Liste ist eine Operation mit konstantem Aufwand. Die Laufzeit orientiert sich daher am Suchen und beträgt ebenfalls  $\mathcal{O}(l_j)$ .

Im Laufe der Zeit werden in unser Hashtable  $\mathcal{T}$  Elemente eingefügt und gelöscht, die derzeitige Anzahl der Elemente bezeichnen wir mit  $n \in \mathbb{N}$ . Wir betrachten nun den durchschnittlichen Aufwand beim Suchen in Abhängigkeit von dem Belegungsfaktor  $\alpha$ . Wir nehmen an, dass wir eine ideale Hashfunktion  $h$  vorliegen haben.

Der worst case wäre, dass alle Elemente durch großes Pech beim hashen in einer einzigen Überläuferliste gespeichert sind, welche wir mit  $\Theta(n)$  Aufwand linear durchsuchen müssen. Dieser Fall hat eine Wahrscheinlichkeit von  $\left(\frac{1}{m}\right)^{n-1}$ .

Im durchschnittlichen Fall erwarten wir eine (ungefähre) Gleichverteilung der Länge der Überläuferlisten und wir suchen in einer Liste mit Länge  $l_j = \alpha$ . Dann ist der zu erwartende Aufwand  $\Theta\left(1 + \frac{\alpha}{2}\right)$ , falls das Element bereits vorhanden ist, und  $\Theta(1 + \alpha)$ , falls es nicht vorhanden ist. In beiden Fällen liegt der Aufwand asymptotisch in  $\Theta(1 + \alpha)$ .

### 3.2.3 Kollisionsbehandlung durch Offene Adressierung

Bei der offenen Adressierung werden keine Pointer und keine Listen verwendet, um den dadurch freigewordenen Speicher in ein größeres Array zu investieren. Alle Elemente werden nun direkt in der Hashtabelle gespeichert. Bei einer Kollision muss also ein anderer Speicherplatz gefunden werden.

Wir konstruieren uns daher eine derivierte Hashfunktion  $h'(s, i)$ , welche nicht nur vom Schlüssel, sondern von der Anzahl  $i$  der fehlgeschlagenen Versuche einen Platz zu finden abhängt. Anfangs ist  $h'(s, 0) = h(s)$ , wir suchen

also zuerst mit einer herkömmlichen Hashfunktion und sondieren dann iterativ weiter, falls der Platz schon belegt war. Falls  $i = m$  ist unsere Hashtabelle offensichtlich voll und wir können nichts mehr abspeichern.

Um zu testen, ob ein Platz belegt ist, müssen wir auf einen Arrayplatz als leer markieren können. Das geschieht mittels eines speziellen markers, denn wir `null` nennen.

Es gibt drei übliche Sondierungsverfahren, das linear probing, quadratic probing und double hashing.

- Lineares Sondieren: Falls es zu einer Kollision kommt, wird einfach der danach liegende Wert genommen:

$$h'(s, i) = (h(w) + i) \bmod m.$$

Diese Methode ist sehr schnell, aber es kommt zu einem sogenannten Primärcluster: Benachbarte Positionen sind mit höherer Wahrscheinlichkeit belegt. Je nach Hashfunktion  $h$  kann das ein großer Nachteil sein.

- Quadratisches Sondieren: Indem man die Sprünge beim Versuch eine freie Stelle zu finden quadratisch ansteigen lässt, kann man den primären Ballungseffekt verhindern. Die derivierte Hashfunktion  $h'$  ist also

$$h'(s, i) = (h(s) + c_2 i^2 + c_1 i) \bmod m.$$

Analog können auch andere Polynome verwendet werden. Der Nachteil ist, dass es zu einem sogenannten sekundären Ballungseffekt kommen kann. Hierbei gibt es zwar Lücken zwischen den Clustern, aber der Orbit kann sehr schnell geschlossen werden. Insbesondere bei einer ungeschickten Wahl der Parameter ist nicht garantiert, dass der Hashtable voll ist, während wir durch unser Quadratisches Sondieren uns nur auf einem Teilbereich der Felder bewegen!

- Zwei Hashfunktionen: Um auch den sekundären Ballungseffekt zu verhindern, werden die Sprünge mithilfe einer zweiten Hashfunktion  $h_2$  vom Wert  $s$  abhängig gemacht.

$$h'(s, i) = (h(s) + i h_2(s)) \bmod m$$

Einfügen, Suchen und Löschen geschieht dann analog zu den Überläuferlisten, mit dem Unterschied, dass wir uns nicht entlang der lokalen Liste entlangbewegen, sondern entlang der sondierten Felder. Außerdem fügen wir, anders als bei Überläuferlisten, nicht am Anfang “der Liste” an, sondern am Ende, sobald wir einen freien Platz gefunden haben. In Pseudocode:

---

**Algorithmus 21** : `insert( $\mathcal{T}, (s, d)$ )`

---

**Input** : A hashtable  $\mathcal{T} = (h', \mathcal{A})$  with open addressing, an element  $(s, d)$

**Output** : The modified hashtable  $\mathcal{T}$

```
[1] for  $i \leftarrow 0$  to  $m - 1$  do
[2]    $j \leftarrow h'(s, i)$ 
[3]   if  $\mathcal{A}[j] = \text{null}$  then
[4]      $\mathcal{A}[j] \leftarrow (s, d)$ 
[5]   return  $\mathcal{T}$ 
[6] error(The hashtable  $\mathcal{T}$  is already full)
```

---

Man beachte, dass die asymptotische Zeit zum Einfügen nun nichtmehr konstant ist, sondern vom Belegungsgrad  $\alpha$  abhängt! Ebenso kann es, anders als bei Überläuferlisten, vorkommen, dass die Hashtabelle voll ist.

---

**Algorithmus 22 : search( $\mathcal{T}, s$ )**


---

**Input :** A hashtable  $\mathcal{T} = (h', \mathcal{A})$  with open addressing, a key  $s$   
**Output :** The element  $(s, d)$ , if it can be found

```

[1] for  $i \leftarrow 0$  to  $m - 1$  do
[2]    $j \leftarrow h'(s, i)$ 
[3]   if  $\text{key}(\mathcal{A}[j]) = s$  then
[4]     return  $(s, d)$ 
[5]   else if  $\mathcal{A}[j] = \text{null}$  then
[6]     error(Key  $s$  not found)
[7] error(Key  $s$  not found)

```

---

Der Algorithmus für das Löschen beginnt wie bei Überläuferliste erst aus dem Suchen. Wird ein Element mit passendem Schlüssel gefunden, so wird der letzte Eintrag mit dem gleichen Hash an diese Stelle geschrieben und das nun nichtmehr benötigte Arrayfeld durch überschreiben mit **null** freigegeben.

Eine Hashtabelle dieser Art definiert sich also mit  $\mathcal{T} = (h', \mathcal{A})$ , wobei  $\mathcal{A}$  ein gewöhnliches Array der Länge  $m$  ist.

Wieder wollen wir die Aufwände für das Suchen betrachten. Wir konzentrieren uns auf den Fall, dass das gesuchte Element nicht in der Tabelle vorhanden ist. In diesem Fall ist der Aufwand für das Suchen gleich dem Aufwand für das Einfügen. Wie zuvor ist der Belegungsgrad  $\alpha$  ausschlaggebend. In Fall von offener Adressierung gilt stets  $\alpha \leq 1$ , wir wollen hier aber  $\alpha < 1$  annehmen, da sonst die Aufwände ins Unendliche wachsen. Weiterhin nehmen wir wieder die Verwendung einer idealen Hashfunktion an:

Ist unser Element nicht vorhanden, so ist (tautologisch) jeder Versuch bis auf den letzten besetzt. Sei  $p_i$  die Wahrscheinlichkeit, dass genau  $i$  Versuche besetzt sind. Dann ist die, im durchschnittlichen Fall, zu erwartete Anzahl der Versuche

$$\text{Erwartungswert der Anzahl der Versuche} = \underbrace{1}_{\text{frei}} + \underbrace{\sum_{i=1}^{\infty} i \cdot p_i}_{\text{besetzt}}$$

Die Summe konvergiert, da  $p_i = 0$  für  $i \geq n$ . Um die Summe abzuschätzen, definieren wir mit  $q_i$  die Wahrscheinlichkeit, dass mindestens  $i$  Versuche besetzt sind:

$$q_1 = P(1. \text{ Versuch besetzt}) = \frac{n}{m} = \alpha$$

$$q_2 = P(1. \text{ und } 2. \text{ Versuch besetzt}) = \frac{n}{m} \cdot \frac{n-1}{m-1} < \alpha^2$$

$$q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \dots \cdot \frac{n-i+1}{m-i+1} < \left(\frac{n}{m}\right)^i = \alpha^i$$

Weiters gilt  $p_i = q_i - q_{i+1}$ . Damit ergibt sich für die Summe

$$\begin{aligned}
\sum_{i=1}^{\infty} i \cdot p_i &= \sum_{i=1}^{\infty} i \cdot q_i - \sum_{i=1}^{\infty} i \cdot q_{i+1} && (p_i \text{ eingesetzt}) \\
&= \sum_{i=1}^{\infty} i \cdot q_i - \sum_{i=2}^{\infty} (i-1) \cdot q_i && (\text{Indexverschiebung}) \\
&= q_1 + \sum_{i=2}^{\infty} i \cdot q_i - \sum_{i=2}^{\infty} (i-1) \cdot q_i && (\text{erstes Element herausgeholt}) \\
&= q_1 + \sum_{i=2}^{\infty} i \cdot q_i - (i-1) \cdot q_i && (\text{Summen zusammenfügt}) \\
&= q_1 + \sum_{i=2}^{\infty} (i - (i-1)) \cdot q_i \\
&= q_1 + \sum_{i=2}^{\infty} q_i \\
&= \sum_{i=1}^{\infty} q_i
\end{aligned}$$

und damit für die erwartete Anzahl der Versuche

$$\text{Erwartungswert der Anzahl der Versuche} = 1 + \sum_{i=1}^{\infty} q_i < 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha},$$

da  $\alpha < 1$  (geometrische Reihe).

Der durchschnittlich zu erwartene Suchaufwand in einer Hashtabelle  $\mathcal{T}$  mit offener Adressierung ist also  $\Theta\left(\frac{1}{1-\alpha}\right)$ , falls der gesuchte Wert  $s$  nicht in der Tabelle vorhanden ist. Im Fall, dass der gesuchte Wert  $w$  in der Hash-Tabelle  $T$  vorhanden ist, ist (hier ohne Beweis) der zu erwartende Suchaufwand  $\Theta\left(\frac{1}{\alpha} \ln \frac{1}{1-\alpha}\right)$ , tatsächlich in einer echt kleineren Aufwandskategorie.

### 3.2.4 Echte Hashfunktionen

Bisher haben wir nur eine ideale Hashfunktion benutzt, die aber so ein theoretisches Konstrukt ist. Hier daher ein paar reale Hashfunktionen:

- Die Identitätsfunktion: Ist der Schlüssel  $s$  bereits eine Zahl in  $\{0, \dots, m-1\}$ , sowie gleichverteilt und kollisionsarm, so kann man als Hashfunktion einfach die Identität  $h(s) = s$  nehmen. Das ist zweifelsohne am schnellsten.
- Division mit Rest:

$$h(s) = s \bmod m$$

Insbesondere für den häufigen Fall  $m = 2^k$  gibt es bei Zahlen in Binärdarstellung keine schnellere Methode, die noch einigermaßen gut funktioniert.

- Multiplikationsmethode: Der Schlüssel  $s$  wird zuerst mit einem fixen reellwertigen Parameter  $0 < A < 1$  multipliziert, dann nur die Nachkommastellen betrachtet (welche in  $[0, 1]$  liegen), dann mit  $m$  multipliziert und schließlich abgerundet:

$$h(s) = \lfloor m \cdot (sA \bmod 1) \rfloor$$

Diese Hashfunktion ist immernoch sehr schnell und liefert bei guter Wahl von  $A$  ausreichend gute Resultate.

- Fibonacci-Hash: Das ist die Multiplikationsmethode mit  $A = \frac{\sqrt{5}-1}{2} \approx 0.6180339\dots$  verwendet. Durch seine Irrationalität erreicht er eine gute Verteilung.
- Kryptographische Hashfunktionen: Im Prinzip kann auch eine beliebige kryptographische Hashfunktion genutzt werden. Diese haben nahezu ideale Eigenschaften (Gleichverteilung, Schwierigkeit eine Hashkollision zu erzwingen, etc), sind aber aufgrund ihres Designs auch sehr rechenintensiv. Außerdem gibt es Einschränkungen in der Größe von  $m$ .

Die Wahl einer guten Hashfunktion ist in der echten Welt nicht immer einfach. Insbesondere bei so grundlegenden Entscheidungen wie “Was ist die Standardhashfunktion für die mitgelieferte Hashtable unserer Programmiersprache” wird sehr viel und gründlich nachgedacht. Im Falle von Python kann man das in PEP 456 gut nachvollziehen:

<https://www.python.org/dev/peps/pep-0456/>.

### 3.2.5 Asymptotische Aufwände und Speicherbedarf

Wir betrachten lineare Listen, Arrays, Hashtables mit Überläuferlisten und Hashtables mit offener Adressierung im Vergleich. Zu beachten ist, dass  $\alpha = \frac{n}{m}$  für Hashtables mit Überläuferlisten durchaus deutlich größer als 1 sein darf, im Fall der offenen Adressierung jedoch  $\alpha < 1$  gelten muss:

	Listen	Arrays	Hashtables (Listen)	Hashtables (off. Adr.)
Einfügen	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1/(1-\alpha))$
Suchen	$\Theta(n)$	$\Theta(n)$	$\Theta(1+\alpha)$	$\Theta(1/(1-\alpha))$
Löschen ( $s$ gegeben)	$\Theta(n)$	$\Theta(n)$	$\Theta(1+\alpha)$	$\Theta(1/(1-\alpha))$
maximales $n$	$\infty$	$m$	$\infty$	$m$
Speicherbedarf	$\Theta(n)$	$\Theta(m)$	$\Theta(\max(m, n))$	$\Theta(m)$

Der eklatante Nachteil einer fixen Größe von Hashtabellen mit offener Adressierung kann durch sogenannte dynamische Arrays angegangen werden: In dem Fall wächst  $m$  bei größer werdendem  $n$  mit. Das vergrößern ist eine kostspielige Operation, die aber nur selten passiert. Man versucht dabei,  $\alpha$  im Bereich von ca  $\alpha \approx 0.7$  zu halten.

### 3.3 Binärbäume

#### 3.3.1 Bäume

**Definition 12** (Baum). Ein Baum  $\mathcal{B}$  ist eine abzählbare Menge an Knoten  $k$ . Ein Knoten ist ein Tupel  $(p, c_1, \dots, c_o, d)$ , wobei  $d$  die Nutzdaten des Knoten sind,  $p$  ein Pointer zu seinem Vaterknoten (parent node) ist und  $c_1, \dots, c_o$  Zeiger zu den Kindknoten (child nodes) sind. Pointer können anstatt der Speicheradresse auch den Wert `void` haben, das sind dann Nullpointer. Die größte Zahl  $o$  über alle Knoten von  $\mathcal{B}$  nennen wir die Ordnung des Baumes  $\mathcal{B}$ .

Die Relation zwischen Vater- und Kindzeiger ist definiert als

$$\text{parent}(\text{child}_i(k)) = k \text{ für alle } i \in \{1, \dots, o\} \text{ mit } c_i \neq \text{void}$$

die Funktion `parent` ist also das Linksinverse zu `childi`, solange der Pointer  $c_i \neq \text{void}$ .

Desweiteren gibt es in jedem nichtleeren Baum  $\mathcal{B}$  genau einen Knoten, der anstelle eines echten parent-pointers den Nullpointer `void` hat. Diesen Knoten nennen wir die Wurzel von  $\mathcal{B}$  und finden ihn mit der Funktion `root`( $\mathcal{B}$ ).

Folgende Funktionen erleichtern den Umgang mit den Pointern und damit das Traversieren im Baum:

- `parent`( $k = (p, c_1, \dots, c_o, d)$ ) =  $\mathcal{M}[p]$
- `childi`( $k = (p, c_1, \dots, c_o, d)$ ) =  $\mathcal{M}[c_i]$
- `data`( $k = (p, c_1, \dots, c_o, d)$ ) =  $d$

Verfolgen `childi` oder `parent` einen Nullpointer, so geben sie den `void_node` zurück, das Equivalent von `void` für Knoten. Im Untenstehenden Pseudocode benutzen wir manchmal auch `void_node = (void, void, ..., void_data)` als “leeren”, frisch befüllbaren Knoten.

Man nennt Knoten ohne Kinder auch Blätter, Knoten mit einer maximalen Anzahl an Kindern voll. Ein Baum bezeichnet man als voll, wenn jeder Knoten entweder ein Blatt ist oder voll. Die Eigenschaft geordnet vergibt man, wenn pro Knoten  $k$  alle Kinder von links nach rechts besetzt werden, also `childi ≠ void_node` impliziert, dass `childj ≠ void_node` für alle  $j \leq i$ .

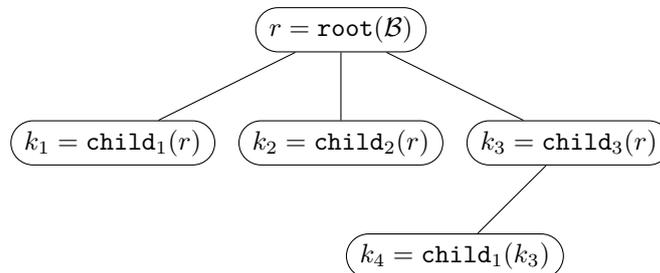


Abbildung 3.2: Ein allgemeiner Baum  $\mathcal{B}$  mit  $n = 5$  Knoten und Ordnung  $o = 3$ . Dabei ist  $r := \text{root}(\mathcal{B})$  die Wurzel des Baumes. In der Ebene 0 befindet sich nur  $r$ , in der Ebene 1 befinden sich  $k_1, k_2, k_3$  und in der Ebene 2 nur  $k_4$ . Weiterhin ist  $r$  der einzige volle Knoten,  $k_1, k_2$  sowie  $k_4$  sind Blätter.

Meistens wird in der Definition eines Baumes verlangt, dass für jeden Knoten  $k$  ein *eindeutiger* Weg von der Wurzel  $w$  zum Knoten  $k$  existiert. Mit Weg ist hierbei das verkettete Aufrufen von `child`-Funktionen gemeint. Die längstmögliche Kette an Aufrufen ist dabei die Höhe des Baumes, kurz  $h$ . Formaler: Für alle Knoten  $k \in \mathcal{B}$  existiert genau eine Knotenfolge (der Länge  $l \in \mathbb{N}$ )  $k_0 := \text{root}(\mathcal{B}), k_1, \dots, k_l$ , sodass  $k_{i-1} = \text{parent}(k_i)$  für alle  $i \in \{1, \dots, l-1\}$ . Damit ist  $l(k)$  als die Länge der Knotenfolge von der Wurzel zu  $k$  wohldefiniert. Die Höhe  $h \in \mathbb{N}$  eines Baums  $\mathcal{B}$  ist dann  $h := \max_{k \in \mathcal{B}} \{l(k)\}$

Alle Knoten, die den gleichen Abstand zur Wurzel haben liegen dabei auf einer Ebene. Die Wurzel liegt in Ebene 0, ihre direkten Kinder auf Ebene 1, ihre Enkel auf Ebene 2, etc. .

Da bei uns jeder Knoten nicht mehr als einen Elternknoten haben kann und dank der garantierten Existenz von  $w$  keine Kreisstrukturen (Zykel) vorhanden sein können, ergibt sich diese sonst als Axiom geforderte Eigenschaft von alleine (siehe Abbildung 3.3 bis Abbildung 3.5 .

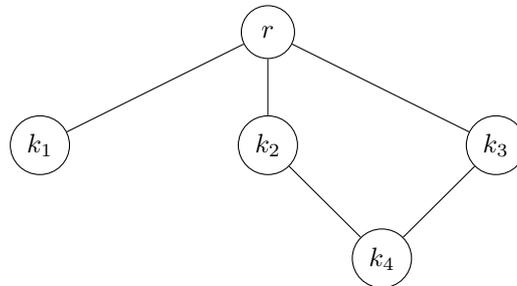


Abbildung 3.3: Kein Baum: Knoten  $k_4$  hat zwei Elternknoten,  $k_2$  und  $k_3$ . Damit ist die Eindeutigkeit des Weges von der Wurzel  $r$  zu  $k_4$  nichtmehr gegeben.

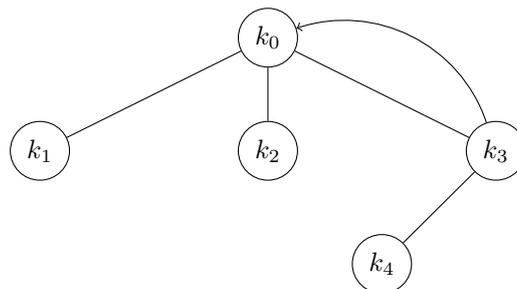


Abbildung 3.4: Kein Baum: Es gibt keinen Knoten ohne Elternknoten:  $k_0 := \text{child}_3(k_3)$  ist Kind von  $k_3$ . Damit ist der Weg zwischen  $k_0$  und  $k_4$  nicht eindeutig.

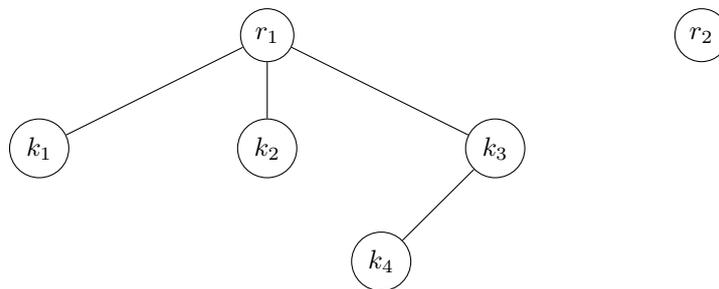


Abbildung 3.5: Kein Baum: Es gibt zwei Knoten ohne Elternknoten:  $r_1$  und  $r_2$ . Damit gibt es zwei Wurzeln, was verboten ist. Zusätzlich existiert kein Weg zwischen  $r_1$  und  $r_2$ .

Ist die Ordnung des Baumes festgelegt, nennen wir einen Baum *vollständig*, wenn jede Ebene, bis auf möglicherweise die letzte, voll besetzt ist. Voll besetzt bedeutet dabei, dass jeder Knoten so viele Kinder besitzt wie die Ordnung des Baumes ermöglicht. Blätter dürfen nur in der letzten Ebene auftauchen. Desweiteren müssen Ebenen von links nach rechts belegt werden. Siehe Abbildung 3.6.

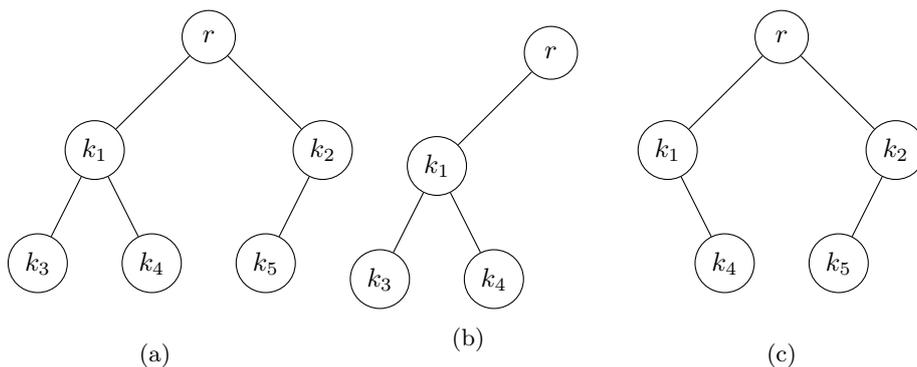


Abbildung 3.6: Wir sehen hier drei Bäume mit Ordnung 2. In (a) haben wir einen vollständigen Baum, jede (bis auf die Letzte) Ebene ist vollständig und die Letzte ist von links nach rechts aufgefüllt. In (b) ist die 1. Ebene nicht vollständig ( $k_2$  fehlt). In (c) ist die Bedingung verletzt, die letzte Ebene von links nach rechts aufzufüllen ( $k_3$  fehlt).

Desweiteren ist ein Baum degeneriert, wenn er die Struktur einer doppelt verketteten Liste hat. Formaler: Die Höhe des Baumes  $h$  entspricht der Anzahl der Elemente (abzüglich der Wurzel)  $n - 1$  (Abbildung 3.7).

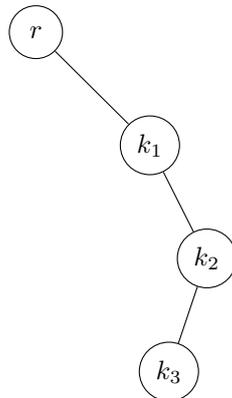


Abbildung 3.7: Ein degenerierter Baum. Von den Endpunkten abgesehen hat jeder Knoten genau einen Vorgänger und Nachfolger, wie eine doppelt verkettete Liste.

### Anwendungen

Bäume sind die ideale Struktur um hierarchische Strukturen zu modellieren. Beispielsweise eine Kommandostruktur in der Armee, einen Stammbaum einer Person oder den Phylogenetischen Baum aller Arten.

Auch in der Informatik sind Bäume sehr beliebt. Beispielsweise als Entscheidungsbäume (siehe Abbildung 2.3.1), Syntaxbäume (siehe beispielsweise XML trees), in unixoiden Dateisystemen (zumindest ohne `link`). Wir werden uns im kommenden vor allem mit binären Suchbäumen, Halden und (2-4)-Bäumen beschäftigen, da sie häufig sehr gute, balancierte Wahlen von Datenstrukturen sind.

### 3.3.2 Binärbaum

**Definition 13** (Binärbaum). Ein Binärbaum ist ein Baum mit Ordnung  $o = 2$ .

Bei einem Binärbaum gibt es also kein  $c_3, \dots$  und damit auch keine Funktionen `child3, ...`. Das erste Kind bekommt der Einfachheit halber den Namen `left := child1` und das zweite `right := child2`.

#### Traversierung / Auslesen

Wie können wir nun systematisch alle Knoten eines Baumes ablaufen? Dazu gibt es verschiedene Methoden, welche alle in einer unterschiedlichen Reihenfolge münden. Wir betrachten die häufigsten Auslesereihenfolgen:

- Die symmetrische Reihenfolge:

$$\text{in\_order}(k) = \text{concat}(\text{in\_order}(\text{left}(k)), \text{data}(k), \text{in\_order}(\text{right}(k)))$$

- Die Hauptreihenfolge:

$$\text{pre\_order}(k) = \text{concat}(\text{data}(k), \text{pre\_order}(\text{left}(k)), \text{pre\_order}(\text{right}(k)))$$

- Die Nebenreihenfolge:

$$\text{post\_order}(k) = \text{concat}(\text{post\_order}(\text{left}(k)), \text{post\_order}(\text{right}(k)), \text{data}(k))$$

- Die Reihenfolge der Breitensuche: Hier werden Ebene für Ebene alle Knoten von links nach rechts abgeschritten. Mit  $r = \text{root}(\mathcal{B})$  besuchen wir die Knoten also in der Reihenfolge

$$[r, \text{left}(r), \text{right}(r), \text{left}(\text{left}(r)), \text{right}(\text{left}(r)), \text{left}(\text{right}(r)), \dots].$$

### Umschreiben von allgemeinen Bäumen in Binärbäume

TODO

#### 3.3.3 Binäre Suchbäume

Binäre Suchbäume speichern Elemente mit totaler Ordnung als Nutzdaten (TODO: Auf Quasiordnung erweitern). Die “Sortiertheitsgarantie” ist, dass alle Werte im linken Unterbaum eines Knotens  $k$  kleiner als  $\text{data}(k)$  sind und alle Werte im rechten Teilbaum größer oder gleich  $\text{data}(k)$ .

Betrachtet man also die symmetrische Reihenfolge eines sortierten Binärbaums, so ist diese sortiert.

Binäre Suchbäume können alle Wörterbuchoperationen effektiv implementieren, aber auch effektiv das Minimum, das Maximum, das nächstgrößere und nächstkleinere Element finden, solange der Baum nicht allzu entartet ist.

Alle Funktionen laufen dabei in  $\mathcal{O}(h)$ , im Idealfall also  $\mathcal{O}(\log_2 n)$ . Im schlechtesten Fall, in dem der Baum zu einer doppelt verketteten Liste degeneriert ist, liegt die Laufzeit in  $\mathcal{O}(n)$ .

**Suchen in Binärbäumen:** Der Algorithmus ähnelt einer Binärsuche.

---

**Algorithmus 23 : search( $k, v$ )**

---

**Input :** A node  $k$  and a data value  $v$

**Output :** The node  $k'$  of the subtree of  $k$  which has  $\text{data}(k') = v$

```
[1] if  $k \neq \text{void\_node}$  and  $\text{data}(k) \neq \text{void}$  then
[2]     return  $k$ 
[3] else
[4]     if  $v \leq \text{data}(k)$  then
[5]         search( $\text{left}(k), v$ )
[6]     else
[7]         search( $\text{right}(k), v$ )
```

---

**Finden des Minimums und Maximums in  $\mathcal{B}$**  Hier gehen wir so lange nach links, bis es kein linkes Kind mehr gibt. Da wir nach symmetrischer Reihenfolge geordnet sind, finden wir damit das Minimum des Baums, wenn wir  $\text{minimum}(\text{root}(\mathcal{B}))$  aufrufen.

---

**Algorithmus 24 :  $\text{minimum}(k)$** 

---

**Input :** A node  $k$  of a binary search tree**Output :** The node with the smallest value within the subtree of  $k$ 

```

[1] if  $\text{left}(k) = \text{void\_node}$  then
[2]     return  $k$ 
[3] else
[4]     return  $\text{minimum}(\text{left}(k))$ 

```

---

Diese Rekursion ist eine Endrekursion und braucht keinen zusätzlichen Speicher. Die Laufzeit ist von  $\mathcal{O}(h)$  begrenzt.

Analog finden wir das Maximum mit  $\text{maximum}(\text{root}(\mathcal{B}))$  wenn wir immer nach rechts laufen.

**Finden des Vorgängers/Nachfolgers eines Knotens:** Übungsaufgabe.

**Einfügen in den Binärbaum:** In den Binärbaum  $B$  wird der Wert  $w$  eingefügt. Wir müssen dabei darauf achten, dass der Wert an der passenden Stelle eingefügt wird, also der Baum in symmetrische Reihenfolge weiterhin sortiert bleibt. Wir gehen also immer dann nach links, wenn der Wert im Knoten größer ist und andernfalls rechts entlang.

---

**Algorithmus 25 :  $\text{insert}(B, v)$** 

---

**Input :** A binary tree  $\mathcal{B}$  and a new value  $v$ **Output :** Side effects in  $\mathcal{M}$ 

```

[1]  $x_{\text{last}} \leftarrow \text{void\_node}$ 
[2]  $x \leftarrow \text{root}(\mathcal{B})$ 
[3] while  $x \neq \text{void\_node}$  do
[4]      $x_{\text{last}} \leftarrow x$ 
[5]     if  $v < \text{data}(x)$  then
[6]          $x \leftarrow \text{left}(x)$ 
[7]     else
[8]          $x \leftarrow \text{right}(x)$ 
[9]  $\text{data}(x) \leftarrow v$ 
[10]  $\text{parent}(x) \leftarrow x_{\text{last}}$ 
[11]  $\text{change\_parents\_reference}(\mathcal{B}, x, x)$ 

```

---

Wobei folgende Hilfsfunktion den richtigen Kindpointer von einem alten Kindknoten auf einen neuen umbiegt. In unserem Fall benutzten wir sie nur, um eine Referenz von  $\text{parent}(x)$  auf  $x$  zu legen.

---

**Algorithmus 26** : `change_parents_reference( $\mathcal{B}, k, t$ )`

---

**Input** : A binary tree  $\mathcal{B}$ , the old child node  $k$  and the new target child node  $t$ **Output** : Side effects in  $\mathcal{B}$ : The parent of  $k$  changes the respective child pointer from  $k$  to  $t$ . In case of  $k$  being the root,  $t$  is set as the new root.

```

[1]  $p \leftarrow \text{parent}(k)$ 
[2] if  $p = \text{void\_node}$  then
[3]    $\text{root}(\mathcal{B}) \leftarrow t$ 
[4] else if  $\text{data}(k) < \text{data}(p)$  then
[5]    $\text{left}(p) \leftarrow t$ 
[6] else
[7]    $\text{right}(p) \leftarrow t$ 

```

---

**Löschen eines Werts:** Soll der Knoten  $k$  gelöscht werden, müssen drei Fälle unterschieden werden, damit die Sortierung erhalten bleibt:

1. Ist  $k$  kinderlos, so entfernen wir den Knoten, indem wir an der passenden Stelle des Elternknotens die Referenz auf `void` umschreiben.
2. Hat  $k$  nur ein Kind,  $c$ , so entfernen wir den Knoten  $k$  indem wir  $k$  überspringen. Dazu biegen wir die Referenz des Elternknotens von  $k$  auf das Kind von  $c$  um.
3. Im letzten Fall hat  $k$  zwei Kinder. Wir ersetzen die Daten von  $k$  durch die Daten des nächstgrößeren Knotens  $t$  und löschen dann den Knoten  $t$ . Da dieser nur maximal ein Kind hat (er kann kein linkes Kind haben, sonst wäre das nächstgrößere) terminiert dieser Rekursive Aufruf von `delete` nach einem Schritt, da wir einem der letzten beiden Fälle landen. Ist das Verschieben der Daten sehr teuer, so können wir alternativ auch alle Pointer so umbiegen, dass  $t$  in der Baumstruktur den Platz von  $k$  einnimmt (nicht gezeigt).

---

**Algorithmus 27** : `delete( $\mathcal{B}, k$ )`

---

**Input** : A binary tree  $\mathcal{B}$  and the unwanted node  $k$ **Output** : Side effects in  $\mathcal{M}$ : The node  $k$  is removed from  $\mathcal{B}$ , but child nodes of  $k$  are preserved.

```

[1] if  $\text{left}(k) = \text{void\_node}$  and  $\text{right}(k) = \text{void\_node}$  then
[2]   change_parents_reference( $\mathcal{B}, k, \text{void\_node}$ )
[3] else if  $\text{left}(k) \neq \text{void\_node}$  XOR  $\text{right}(k) \neq \text{void\_node}$  then
[4]   if  $\text{left}(k) \neq \text{void\_node}$  then
[5]      $c \leftarrow \text{left}(k)$ 
[6]   else
[7]      $c \leftarrow \text{right}(k)$ 
[8]   change_parents_reference( $\mathcal{B}, k, c$ )
[9] else
[10]   $t \leftarrow \text{minimum}(\text{right}(k))$ 
[11]   $\text{data}(k) \leftarrow \text{data}(t)$ 
[12]  delete( $\mathcal{B}, t$ )

```

---

### 3.4 Halden (Heaps)

Halden sind eine Datenstrukturen, welche insbesondere zur Implementierung von priority queues geeignet sind, aber auch einen worst-case optimalen in-place Sortieralgorithmus bereitstellen. Es gibt zwei Perspektiven auf Halden: Einmal als Array und einmal als vollständiger, sortierter Binärbaum:

**Definition 14** (Halde: Arraydefinition). Eine Halde  $\mathcal{H}$  ist ein Array  $\mathcal{A}$  von  $n$  Elementen mit einer Quasiordnung, welches der folgenden Haldenbedingung genügt:

$$\mathcal{A}[i] \geq \max\{\mathcal{A}[2i + 1], \mathcal{A}[2i + 2]\} \text{ für alle } i \in \left\{0, 1, \dots, \left\lfloor \frac{n}{2} \right\rfloor - 1\right\}$$

**Definition 15** (Halde: Baumdefinition). Eine Halde ist ein vollständiger binärer Baum  $\mathcal{H}$ , welcher in seinen Knoten Elemente mit einer Quasiordnung enthält. Zudem gilt für Knoten  $k \in \mathcal{H}$ :

$$\text{data}(k) \geq \max\{\text{data}(\text{left}(k)), \text{data}(\text{right}(k))\}.$$

Dabei ist `void_data` kleiner als jedes Element der Quasiordnung.

Direkt aus der Definition ergibt sich sofort, dass  $\mathcal{H}[0]$  beziehungsweise  $\text{root}(\mathcal{H})$  das Maximum der Halde ist. Ebenso ist jeder Knoten das Maximum seines Teilbaums. Die Höhe einer Halde ist dabei für  $n > 0$  definiert als  $h(n) := \lceil \log_2 n \rceil$ . Eine leere Halde hat die Höhe  $h(0) = -1$ .

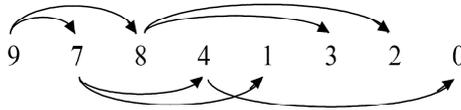


Abbildung 3.8: Eine Halde aus Arrayperspektive. Pfeile zeigen die Beziehung zwischen  $i$  und  $2i + 1$  beziehungsweise  $2i + 2$  an.

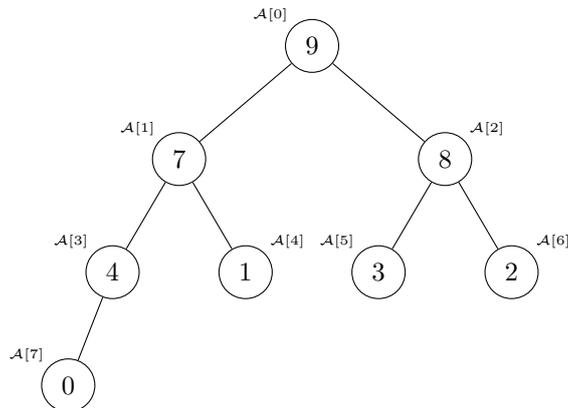


Abbildung 3.9: Eine Halde aus Baumperspektive. Kindbeziehungen anstelle der Pfeile aus Abbildung 3.8

### 3.4.1 Verhalden (heapify)

Der Algorithmus `heapify` repariert eine Fast-Halde, bei dem die Haldenbedingung an genau einer Stelle  $i$  verletzt ist. Wir nehmen dabei an, dass der linke und rechte Teilbaum von  $i$  Halden sind.

Um die Haldenbedingung zu erzwingen, muss das  $i$ . Element also mit dem Maximum seiner beiden Kinder getauscht werden. Ist das  $i$ . Element aus Baumperspektive ein Blatt (also  $i \geq \frac{n}{2}$ ), so ist die Haldenbedingung immer erfüllt.

Andernfalls vertauschen wir das  $i$ . Elemente mit maximum seiner Kindknoten. Damit ist bei  $i$  die Haldenbedingung erfüllt, aber möglicherweise bei einem Kindknoten nun verletzt. Also rufen wir den Algorithmus dort noch einmal auf.

In Pseudocode aus der Array-Perspektive sieht das folgendermaßen aus (Der Pseudocode in Baumperspektive ist eine Übungsaufgabe):

---

#### Algorithmus 28 : `heapify( $\mathcal{H}, i$ )`

---

**Input :** An almost-heap  $\mathcal{H}$  of size  $n$  with above conditions and the index  $i \in \{0, \dots, n-1\}$

**Output :** A modified  $\mathcal{H}$  with its heap structure intact

```

[1]  $l \leftarrow 2i + 1$ 
[2]  $r \leftarrow 2i + 2$ 
[3]  $j \leftarrow i$ 
[4] if  $l < n$  and  $\mathcal{H}[l] > \mathcal{H}[i]$  then
[5]      $j \leftarrow l$ 
[6] if  $r < n$  and  $\mathcal{H}[r] > \mathcal{H}[j]$  then
[7]      $j \leftarrow r$ 
[8] if  $i \neq j$  then
[9]     swap( $\mathcal{H}[i], \mathcal{H}[j]$ )
[10]  $\mathcal{H} \leftarrow \text{heapify}(\mathcal{H}, j)$ 
[11] return  $\mathcal{H}$ 

```

---

**Korrektheit:** Unter obig beschriebenen Bedingungen repariert der Algorithmus den Knoten  $i$ , möglicherweise auf Kosten der Haldenbedingung auf in einer seiner Kindknoten. Da in dem Fall der Algorithmus rekursiv aufgerufen wird und der Basisfall (Knoten ist ein Blatt) die Haldenbedingung trivial erfüllt, terminiert der Algorithmus und arbeitet korrekt.

**Laufzeit:** Hier hilft es, die Perspektive des Baums einzunehmen: Wir erkennen, dass der Algorithmus schlimmstenfalls  $h = \log_2(n)$  mal aufgerufen wird. Weniger oft, wenn wir in einer tieferen Ebene starten. Da der Aufruf von `heapify` ohne seine rekursiven Unteraufrufe  $c \in \mathcal{O}(1)$  dauert, liegt die Laufzeit von `heapify` damit in  $\mathcal{O}(\log n)$ .

### 3.4.2 Aufbau einer Halde aus einem Array/Baum

Wir können aus einem beliebigen Array (oder vollständigen Binärbaum) eine Halde machen, indem wir, angefangen an den Blättern bis hin zur Wurzel, immer wieder `heapify` rufen. Bei den Blättern können wir es uns sparen, da sie eh schon Heaps sind. Damit haben wir (in Array-Perspektive) folgenden Pseudocode

---

**Algorithmus 29 : heapify\_array( $\mathcal{A}$ )**

---

**Input :** An array  $\mathcal{A}$  of size  $n$ **Output :** The same Array but resorted as a heap

```

[1] for  $i \leftarrow \lfloor \frac{n}{2} \rfloor$  to 0 do
[2]   heapify( $\mathcal{A}, i$ )
[3] return  $\mathcal{A}$ 

```

---

**Korrektheit:** Dadurch, dass wir von unten nach oben arbeiten, sind Teilbäume bereits Halden. Dadurch sind die Bedingungen von `heapify` erfüllt und der Algorithmus arbeitet korrekt.

**Laufzeit:** Die Funktion `heapify` wird  $\frac{n}{2}$  mal aufgerufen und hat selber einen Aufwand von  $\mathcal{O}(\log n)$ . Dadurch ist die Abschätzung für den asymptotischen Aufwand von `heapify_array` mit  $\mathcal{O}(n \log n)$  schnell getroffen. Allerdings stellt sich heraus, dass man auch exakter abschätzen kann, da die Laufzeit von `heapify` nicht immer der Höhe des kompletten Baumes entspricht. Dadurch erreicht man eine genauere Abschätzung: Die Laufzeit von `heapify_array( $\mathcal{A}$ )` liegt in  $\Theta(n)$ , wobei  $n$  die Länge des Array  $\mathcal{A}$  ist (Übungsaufgabe).

### 3.4.3 Sortieren mit Halden (Heapsort)

Mit Hilfe der beiden vorherigen Algorithmen kann man auch einen Sortieralgorithmus konstruieren. Wir starten mit einem Array/Baum und formen ihn in eine Halde um. An der Wurzel des Baumes bzw dem Anfang des Arrays haben wir dann das Maximum aller Elemente. Dieses Element gilt als sortiert und wird mit dem letzten Platz des Arrays vertauscht, welcher nun nichtmehr angefasst wird. Ein erneuter Aufruf von `heapify` findet wieder das Maximum, wir vertauschen mit dem vorletzten Feld, etc. Damit haben wir ein worst-case optimalen in-place Sortieralgorithmus:

---

**Algorithmus 30 : heapsort( $\mathcal{A}$ )**

---

**Input :** An array  $\mathcal{A}$  of size  $n$ **Output :** The same Array, but sorted

```

[1]  $\mathcal{A} \leftarrow \text{heapify\_array}(\mathcal{A})$ 
[2] for  $i \leftarrow n - 1$  to 1 do
[3]   swap( $\mathcal{A}[i], \mathcal{A}[0]$ )
[4]    $\mathcal{A}[0, \dots, i] \leftarrow \text{heapify}(\mathcal{A}[0, \dots, i], 0)$ 
[5] return  $\mathcal{A}$ 

```

---

**Korrektheit:** Nach dem initialen Verhalten des Arrays mit `heapify_array( $\mathcal{A}$ )` ist nun sichergestellt, dass der linke und rechte Teilbaum der Wurzel Halden sind. Dadurch können wir `heapify` auf das Wurzelement aufrufen und haben danach eine garantierte Halde und damit stets das Maximum an der Spitze. Durch das Vertauschen wird die Haldenbedingung nur an einem Knoten verletzt, `heapify` kann also weiter korrekt arbeiten. Da der "haldifizierte" Anteil des Arrays mit jedem Schleifendurchlauf schrumpft und sein Maximum an den sortieren hinteren Teil abgibt, terminiert der Algorithmus und sortiert das Array.

**Laufzeit:** Es reicht hier obige Abschätzung, dass `heapify_array` einen asymptotischen Aufwand von  $\mathcal{O}(n \log n)$  hat. Die Laufzeit wird von den  $\frac{n}{2}$  aufrufen von `heapify`, welches  $\mathcal{O}(\log n)$  Aufwand hat, eh dominiert. Also liegt die Laufzeit in  $\mathcal{O}(n \log n)$ . Damit ist `heapsort` also worst-case optimal.

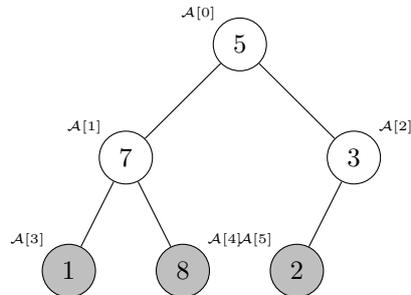


Abbildung 3.10: Der Algorithmus `heapsort` auf den Array  $\mathcal{A} = [5, 7, 3, 1, 8, 2]$ . Im ersten Schritt wird (nacheinander, in dieser Reihenfolge) `heapify` auf die Blätter mit den Werten 2, 8 und 1 ausgeführt, was jeweils keinerlei Effekt hat.

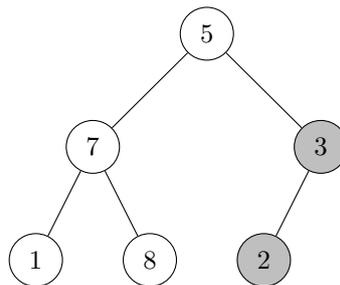


Abbildung 3.11: Der Algorithmus `heapsort` auf den Array  $\mathcal{A} = [5, 7, 3, 1, 8, 2]$ . Im zweiten Schritt wird `heapify` auf den Knoten mit dem Wert 3 ausgeführt, also die Haldenbedingung zu seinem Kind 2 überprüft. Da die Haldenbedingung für diesen Unterbaum nicht verletzt ist, passiert weiter nichts.

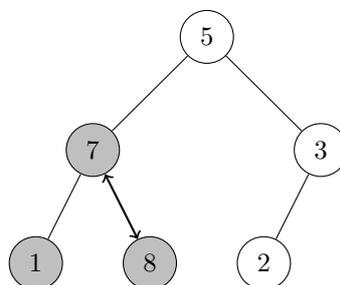


Abbildung 3.12: Der Algorithmus `heapsort` auf den Array  $\mathcal{A} = [5, 7, 3, 1, 8, 2]$ . Im dritten Schritt wird `heapify` auf den Knoten mit dem Wert 7 ausgeführt, also die Haldenbedingung zu seinem Unterbaum überprüft und gegebenenfalls repariert. Da 8 größer ist als 7, werden die beiden Werte getauscht.

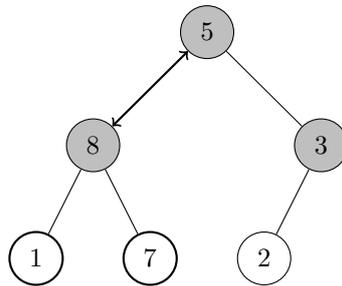


Abbildung 3.13: Der Algorithmus `heapsort` auf den Array  $\mathcal{A} = [5, 7, 3, 1, 8, 2]$ . Im dritten Schritt wird `heapify` auf den Knoten mit dem Wert 7 ausgeführt, also die Haldenbedingung zu seinem Unterbaum überprüft und gegebenenfalls repariert. Da 8 größer ist als 7, werden die beiden Werte getauscht.

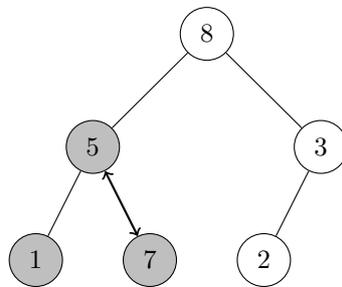


Abbildung 3.14: Der Algorithmus `heapsort` auf den Array  $\mathcal{A} = [5, 7, 3, 1, 8, 2]$ . Im dritten Schritt wird `heapify` auf den Knoten mit dem Wert 7 ausgeführt, also die Haldenbedingung zu seinem Unterbaum überprüft und gegebenenfalls repariert. Da 8 größer ist als 7, werden die beiden Werte getauscht.

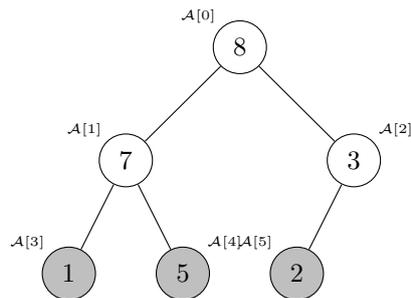


Abbildung 3.15: Der Algorithmus `heapsort` auf den Array  $\mathcal{A} = [5, 7, 3, 1, 8, 2]$ . Im dritten Schritt wird `heapify` auf den Knoten mit dem Wert 7 ausgeführt, also die Haldenbedingung zu seinem Unterbaum überprüft und gegebenenfalls repariert. Da 8 größer ist als 7, werden die beiden Werte getauscht.

### 3.4.4 Wartschlange (Priority Queue)\*

Eine der häufigsten Anwendungen von Halden: Warteschlangen mit einer Priorität, bei der immer das dringende (also das größte) Element gesucht wird. Da das sofort das erste Arrayelement ist, kommt der Vorteil von Halden voll zum tragen. Einfügen und Löschen erfolgt mit  $\mathcal{O}(\log n)$  Aufwand, wie bei Baumstrukturen üblich.

TODO: Details

# Anhang A

## Programmcode

### A.1 quicksort in Python

```
1 def swap(A,i,j):
2     A[i], A[j] = A[j], A[i]
3     return A
4
5 def partition(A):
6     p = A[-1]
7     print("A is {} as partition is called, pivot is {}".format(A, p))
8     i = -1
9     for j in range(len(A)-1):
10        if A[j] <= p:
11            A = swap(A,i+1,j)
12            i += 1
13            print("A is {} after swapping i={} and j={} during partition as {} <= {}".format(A,i,j,A[i],p))
14
15    A = swap(A,i+1,-1)
16    i += 1
17    print("A is {} after swapping {} and {} in final swap in partition, returning A
18        and pivot index {} to quicksort".format(A,i,len(A)-1,i))
19    return (A,i)
20
21 def quicksort(A):
22    print("A is {} as quicksort is called".format(A))
23    if len(A) > 1:
24        A,k = partition(A)
25        print("A is {} after partition, calculating now: quicksort({}) + [A[{}]] +
26            quicksort({}) ".format(A, A[:k], k, A[k+1:]))
27        return quicksort(A[:k]) + [A[k]] + quicksort(A[k+1:])
28    else:
29        return A
30
31 #A = [7,2,1,8,6,3,5,4] # see https://www.youtube.com/watch?v=MZaf_9IZCrc for step-
32     for-step explanation of the first partition here
33 # A = [34,56,99,46,23,16,23,11,23,56] # a more complex case, might be helpful ...
34 A = [3,3,2]
35 A = quicksort(A)
36
37 print("A was sorted to {}".format(A))
```