Algorithmen und Datenstrukturen 2023

bergery@student.ethz.ch

Oktober 2023

ETH zürich

Contents

1 Vorwort

Dieses Skript dient dazu eine grobe Übersicht für die Vorlesung "Algorithmen und Datenstrukturen" zu geben. Jegliche Informationen dieses Skriptes sollte man an der Prüfung können.
Für Fehler in diesem Skript wird keine Verantwortung übernommen.

2 Mathematische Grundlagen

2.1 Induktion

Die vollständige Induktion ist eine mathematische Beweismethode, nach der eine Aussage für alle natürlichen Zahlen bewiesen wird, die größer oder gleich einem bestimmten Startwert sind. Da es sich um unendlich viele Zahlen handelt, kann eine Herleitung nicht für jede Zahl einzeln erbracht werden. Sie ist ein deduktives Verfahren.
Der Beweis, dass die Aussage $A(n)$ für alle $n \geq n_0 (n_0$ meist 0 oder 1), gilt, wird dabei in zwei Etappen durchgeführt:

1. Im *Induktionsanfang* wird die Gültigkeit der Aussage $A(n_0)$ für eine kleinste Zahl n_0 gezeigt.
2. Im *Induktionsanfang* wird für ein beliebiges $n \geq n_0$ die Gültigkeit der Aussage $A(n+1)$ aus der Gültigkeit von $A(n)$ geschlussfolgert.

Beispiel:
Aussage: $A(n) := 1 + 2 + 3 + ... + n = \frac{n \cdot (n+1)}{2}$ für $n \geq 1$
Base Case: Für $n = 1$ gilt: $1 = \frac{1 \cdot (1+1)}{2} \rightarrow 1 = 1$ ✓
Induction hypothesis: "We now assume that it is true for $n = k$, i.e., $1 + 2 + 3 + ... + k = \frac{k \cdot (k+1)}{2}$.
Induction step: $k \rightarrow k + 1$
$1 + 2 + 3 + ... + k + (k + 1) \overset{\text{I.H}}{=} \frac{k \cdot (k+1)}{2} + (k + 1)$, where $\frac{k \cdot (k+1)}{2} + (k + 1) = \frac{k^2 + 3k + 2}{2} = \frac{(k+1) \cdot (k+2)}{2}$ □.
By the principle of mathematical induction, we conclude that $1 + 2 + 3 + ... + n = \frac{n \cdot (n+1)}{2}$ is true for all $n \in \mathbb{N}$.

2.2 Wichtige Annäherungen

Damit wir schnelle Resultate erhalten können, sind Approximationen ein fundamentaler Bestandteil. Aus diesem Grund folgende Liste:

- $n! \leq n^n$, for $n \geq 1$
- $\left(\frac{n}{2}\right)^{n/2} \leq n!, \ for \ n \geq 1$
- $\sum_{i=0}^{n} \log n = \ \log \prod_{i=1}^{n} n = \log n! \ \approx n \log n \rightarrow \mathcal{O}(n \log n)$
- $\sum_{i=0}^{n} 1 = (n + 1) \rightarrow \mathcal{O}(n)$
- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \rightarrow \mathcal{O}(n^2)$
- $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \rightarrow \mathcal{O}(n^3)$
- $\sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4} \rightarrow \mathcal{O}(n^4)$
- binomial coefficient: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
- $\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1} \qquad \binom{n}{n-k} = \binom{n}{k}$
- $\sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k$

Notation	Definition	Mathematische Definition
$f \in o(g)$	asymptotisch gegenüber g vernachlässigbar	$\lim_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert = 0$
$f \in \mathcal{O}(g)$	asymptotische obere Schranke	$\limsup_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$
$f = \Omega(g)$	asymptotische untere Schranke, f ist nicht in $o(g)$	$\limsup_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert > 0$
$f \in \Theta(g)$	scharfe Schranke, sowohl $f \in \mathcal{O}(g)$ als auch $f \in \Omega(g)$	$0 < \liminf_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert \leq \limsup \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$
$f \in \omega(g)$	asymptotisch dominant, $g \in o(f)$	$\lim_{x\to a} \frac{f(x)}{g(x)} \right\rvert = \infty$

Table 1: Definition der $\mathcal{O}$-Notation

2.3 Big-O Notation

Landau-Symbole (engl. Big-O notation) werden verwendet, um das asymptotische Verhalten von Funktionen und Folgen zu beschreiben. In der Informatik werden sie bei der Analyse von Algorithmen verwendet und geben ein Maß für die Anzahl der Elementarschritte oder der Speichereinheiten in Abhängigkeit von der Größe des gegebenen Problems an.
Für Annäherungen wird auch sehr oft die **Regel von de l'Hôpital** angewendet:
Let $f, g : \mathbb{R} \to \mathbb{R}$ be differentiable functions with $f(x) \to \infty, g(x) \to \infty$ for $x \to \infty$.
If $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists, then $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$

2.3.1 Definition O-Notation

In der Tabelle (1) sehen wir jegliche Definition der $\mathcal{O}$-Notation. Wir unterscheiden zwischen, Ω (Omega) [lower-bound], Θ (Theta) [tight-bound] und $\mathcal{O}$ [upper-bound]. Damit dies noch mathematisch formuliert ist haben wird folgendes:
Let $f, g : \mathbb{R} \to \mathbb{R}^+$ such that the limit of $\frac{f}{g}$ exists. Then:

$$\lim_{x\to\infty} \frac{f}{g} = \infty \Rightarrow g \in \mathcal{O}(f) \text{ and } f \in \Omega(g)$$

$$\lim_{x\to\infty} \frac{f}{g} = C \in \mathbb{R}^+ \setminus \{0\} \Rightarrow f \in \Theta(g) \text{ and } g \in \Theta(f)$$

$$\lim_{x\to\infty} \frac{f}{g} = 0 \Rightarrow f \in \mathcal{O}(g) \text{ and } g \in \Omega(f)$$

2.3.2 *Upper bound (big-O)

$$\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

$$\mathcal{O}(f) \leq O(g) \Leftrightarrow \exists c, n_0 . \forall n \geq n_0 . f(n) \leq c \cdot g(n)$$

Lower bound (big-Omega)

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

$$\Omega(f) \geq \Omega(g) \Leftrightarrow \exists c, n_0 . \forall n \geq n_0 . f(n) \geq c \cdot g(n)$$

Tight bound (big-Theta)

$$\Theta(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$$\Theta(f) = \Theta(g) \Leftrightarrow \exists c_1, c_2 n_1, n_2 . \forall n \geq n_1 . f(n) \leq c \cdot g(n) \wedge \forall n \geq n_2 . f(n) \geq c \cdot g(n)$$

Therefore:

$$\Theta(f) = \Theta(g) \Leftrightarrow O(f) \leq O(g) \text{ and } \Omega(f) \geq \Omega(g)$$

Wichtiger Hinweis: Implikationspfeil anschauen: $\mathcal{O}(f) \implies \lim_{x\to\infty} \frac{f}{g} = \infty$ ↯

2.3.3 Master Theorem

Damit das Rechnen mit solchen Grenzwerten schneller geht, haben wir das Master-Theorem, welches sehr praktische Anwendung mit sich bringt:

$$T(n) \leq aT(n/2) \ + Cn^b := \begin{cases} T(n) \leq \mathcal{O}(n^b), & \text{b} > \log_2(a) \\ T(n) \leq \mathcal{O}(n^b \log(n)) & \text{b} = \log_2(a) \\ T(n) \leq \mathcal{O}(n^{\log_2(a)}) & \text{b} < \log_2(a) \end{cases} \tag{1}$$

2.3.4 Overview

Damit wir nun einmal sehen können wie die asympotischen Grenzen sich verhalten, folgende Abbildung: Die graphische Darstellung der Funktionen im Bild (1) zeigt uns leider nur sechs Funktionen.

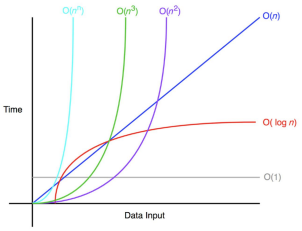


Figure 1: Wachstum verschiedener Funktionen

Um möglichst viele Funktionen in aufsteigendem, asymptotischen Wachstum zu sehen, folgende Abbildung:

$\mathcal{O}(1) \in [\mathcal{O}(\frac{1}{n^2}) \in \mathcal{O}(\frac{1}{n}) \in]\ \mathcal{O}(\log\log n) \in \mathcal{O}(\log n) \in \mathcal{O}(\sqrt{n}) \in \mathcal{O}(n) \in \mathcal{O}(n \log n) \in [\mathcal{O}(n^c) \in \mathcal{O}(c^n) \in]\mathcal{O}(n^2) \in \mathcal{O}(2^n) \in \mathcal{O}(2^{n^2}) \in \mathcal{O}(n!)$

2.4 Erster Algorithmus

2.4.1 Maximum Subarray Sum

Für einen ersten Algorithmus schauen wir den `Maximum Subarray Sum` Algorithmus an. Dieser Algorithmus enthält ein Array von n rationalen Zahlen $a_1, ..., a_n$ und gesucht ist ein Teilstück mit maximaler Summe.
Beispiel:
Sei ein Array [-2, -5, 6, -2, -3, 1, 5, -6] gegeben, so ist die maximale Teilsumme dieses Array 7, nämlich $6 + (-2) + (-3) + 1 + 5 = 7$.

Algorithm 1 Maximum Subarray Sum $(a_1, ..., a_n)$

```
1: function FindMSS(Array [1...n])
2:     S_0 ← 0
3:     for i ← 1, ..., n do
4:         S_i ← S_{i-1} + a_i
5:     end for
6:     maxS ← 0
7:     for i ← 1, ..., n do
8:         for j ← 1, ..., n do
9:             S ← S_j − S_{j-1}
10:            Merke maximales S
11:        end for
12:    end for
13: end function
```

Dieser naive Algorithmus führt insgesamt $\Theta(n^3)$ viele Additionen aus, was sehr schlecht ist für unseren Algorithmus.

Algorithm 2 MSS divide and conquer $(a_1, ..., a_n)$

```
1: function MSSDivAndConquer(Array [1, ..., n])
2:     Wenn n = 1 ist, dann gib max a_1, 0 zurück.
3:     Wenn n > 1 ist:
4:     Teile die Eingabe in A_1 = ⟨a_1, ..., a_{n/2}⟩ und A_2 = ⟨a_{n/2+1}, ..., a_n⟩ auf
5:     Berechne rekursiv den Wert W_1 einer besten Lösung für das Array A_1
6:     Berechne rekursiv den Wert W_2 einer besten Lösung für das Array A_2
7:     Berechne grösste Suffixsumme S in A_1
8:     Berechne grösste Suffixsumme P in A_2
9:     Setzte W_3 ← S + P
10:    Gib max{W_1, W_2, W_3} zurück
11: end function
```

Wir haben schon eine Verbesserung des Algorithmus von $\Theta(n^3)$ vielen Additionen zu $\Theta(n \log n)$ vielen Additionen. In einem letzten Schritt können wir diesen sogar noch einmal verbessern nämlich bis zu $\Theta(n)$ viele Additionen.

Algorithm 3 MSS-Induktiv $(a_1, ..., a_n)$

```
1: function MSSInduktiv(Array [1, ..., n])
2:     randmax ← 0
3:     maxS ← 0
4:     for i ← 1, ..., n do
5:         randmax ← randmax + a_i
6:         if randmax > maxS then
7:             maxS ← randmax
8:         end if
9:         if randmax < 0 then
10:            randmax ← 0
11:        end if
12:        Gib maxS zurück
13:    end for
14: end function
```

3 Suchen und Sortieren

3.1 Suchen

In diesem Abschnitt schauen wir wie schnell wir in abstrakten Daten Strukturen suchen, einfügen und löschen können. Die meist gebrauchten Datenstrukturen sind somit in folgender Tabelle aufgeführt mit ihrer korrespondierender Laufzeit:

Data structure	Search	Insert	Delete
Unsorted Array	$O(n)$	$O(1)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$
Unsorted List	$O(n)$	$O(1)$	$O(n)$
Sorted List	$O(n)$	$O(n)$	$O(n)$
Unbalanced Tree	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

Wir stellen fest, dass es im Allgemeinen einen Kompromiss gibt zwischen einfachen Datenstrukturen, die ein einfaches Einfügen und Löschen ermöglichen, aber die Einträge nicht vorverarbeiten, um spätere Suchvorgänge zu erleichtern (ungeordnetes Array), und komplexeren Datenstrukturen mit höheren Einfüge- und Löschkosten, die effizienter abgefragt werden können.

3.1.1 Binary Search

Binary Search ist der standart Suchalgorithmus für **sortierte** Arrays, da es einen effizienten, logarithmische Laufzeitkomplexität hat.

Algorithm 4 Binary search

```
1: function FindIndex(A, e)                ▷ Search item e in sorted array A
2:     l, r ← 0, A.length − 1
3:     while r > l do
4:         m ← ⌊(l+r)/2⌋
5:         if A[m] = e then
6:             return m
7:         else if A[m] > e then
8:             r ← m − 1
9:         else
10:            l ← m + 1
11:        end if
12:    end while
13:    return "not found"
14: end function
```

Obwohl bei der binären Suche die Kosten für die Suche in geordneten Arrays logarithmisch werden, sind Einfügung und Löschung immer noch linear: im schlimmsten Fall, d.h. wenn das einzufügende oder zu löschende Element das erste Element des Arrays ist, müssen wir alle Elemente um einen Schritt nach rechts/links verschieben.

3.1.2 Heaps

Noch effizienter als geordnete Arrays sind daher baumartige Strukturen, bei denen die Kosten für das Einfügen und Löschen ebenfalls logarithmisch sein können. Heaps sind eine spezielle Klasse von baumbasierten Strukturen, die eine effiziente (zeitkonstante) Methode zur Extraktion des kleinsten oder größten Elements bieten.

Genauer gesagt sind Min- (bzw. Max-) Heaps baumbasierte Datenstrukturen, die die folgende Heap-Invariante erfüllen: Wenn A das Elternteil von B ist, dann ist der Wert von Knoten A kleiner (bzw. größer) als der Wert von Knoten B. Wir betrachten hier binäre Min-Heaps, was bedeutet, dass ein Elternteil einen kleineren Wert hat als seine (höchstens zwei) Kinder. Für binäre Min-Haufen gilt zusätzlich die folgende Forminvariante: Der betrachtete Haufen ist immer ein vollständiger binärer Baum, d. h. alle Schichten des Baums sind von oben nach unten und von links nach rechts gefüllt.

Figure 2: Beispiel eines Min-Heaps

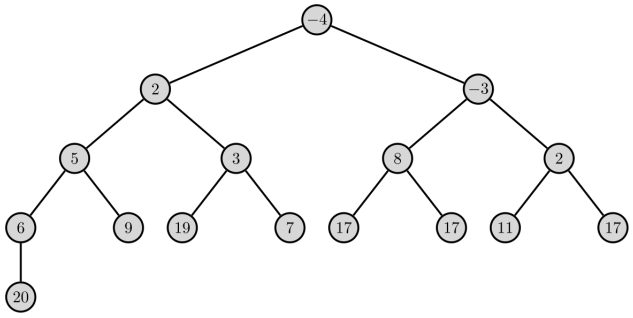
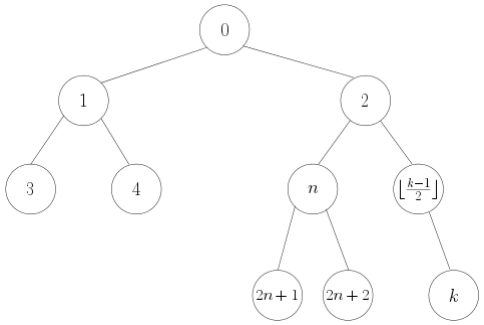


Figure 3: Beispiel 0-Heaps



Implementation

Die gebräuchlichste Implementierung ist ein Array (mit fester oder dynamischer Größe). Geht man von einem 0-basierten Array aus (vgl. Figure 3), so sind die *children* des Knotens n, $2n+1$ und $2n+2$ und die *parent* des Knotens k ist der Knoten $\lfloor \frac{k-1}{2} \rfloor$.

Common operations

Here are the most common operations that a min-heap must support:

- **Basic operations**
  - Find min,
  - Delete min,
  - Insert,
  - Find min and delete it (pop);
- **Initialization**
  - Create empty heap,
  - Heapify (transform array into heap);
- **Inspection**
  - Return size,
  - Test if empty;
- **Other**
  - Increase/decrease,
  - Delete,
  - Restore heap invariant,
  - Merge/union.

Preserving the heap invariant

Die Heap Invariante sagt uns eigentlich aus:

> "Das Label jedes Knoten n ist kleiner oder gleich den Labels der Kinder von n."

Bei der Aktualisierung eines Heaps ist es wichtig, die oben beschriebene Invariante beizubehalten. Nach einer Löschung oder einer Einfügung kann es vorkommen, dass ein (einzelnes) Element kleiner wird als eines seiner Kinder; wir können dann die Heap-Invariante in logarithmischer Zeit wiederherstellen, indem wir dieses Element nach unten durchsickern lassen. (Vgl. Durchsickeralgorithmus 5)

Costs

In der folgenden Tabelle sind die Kosten von Standard-Heap-Operationen für drei Arten von Heaps zusammengefasst: binäre Heaps, binomische Heaps und Fibonacci-Heaps.

Operation	Binary	Binomial	Fibonacci
search min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
increase key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
union	$\Theta(m \log n)$	$O(\log n)$	$\Theta(1)$

Zusammenfassung- Youtube

Kurze Zusammenfassung anschauen

Algorithm 5 Restore heap invariant by percolating element down

```
1: function PercolateDown(H, i)                          ▷ Percolate element i in H
2:     e ← H[i]
3:     if 2i + 2 = H.length then
4:         if e > H[2i + 1] then
5:             Swap H[i] and H[2i + 1]
6:         end if
7:     else if 2i + 2 < H.length then
8:         l, r ← H[2i + 1], H[2i + 2]
9:         if l < r then
10:            if l < e then
11:                Swap H[i] and H[2i + 1]
12:                PercolateDown(H, 2i + 1)
13:            end if
14:        else
15:            if r < e then
16:                Swap H[i] and H[2i + 2]
17:                PercolateDown(H, 2i + 2)
18:            end if
19:        end if
20:    end if
21: end function
```

3.2 Binärer Suchbaum

3.2.1 Begrifflichkeiten

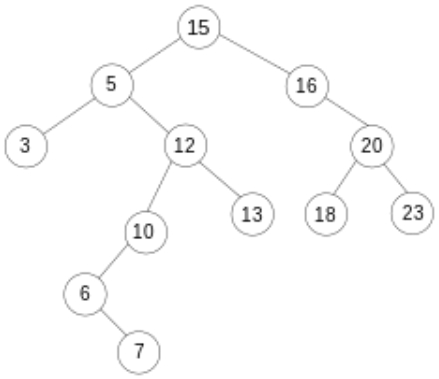
Bäume im Allgemeinen gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen. Seien es Entscheidungsbäume, Syntaxbäume, Ableitungsbäume, Kodebäume oder auch Suchbäume. Bäume sind Listenstrukturen, ein Element referenziert einen **Knoten**. Jegliche Nachfahren die einen Knoten beinhalten nennt man **Kinder-Knoten**. Die **Wurzel** des Baumes ist, von welchem der Baum aus startet. Zusätzlich gilt, dass von jeder Wurzel der verschiedenen Knoten eines Baumes durch genau einen Pfad mit der Wurzel verbunden. Ein **Blatt** oder **Blätter** eines Baumes sind die Knoten welche keine Nachfolger haben. Im Beispiel von 4 wären die Blätter dieses Baumes: [3, 7, 13, 18, 23]. Man nennt einen Baum **geordnet** wenn unter den Kinderknoten eines jeden Knotens eines Baumes eine **Anordnung definiert** ist, sodass man vom ersten, zweiten, dritten usw. Kinderknoten eines Knotens sprechen kann, so nennt man den Baum geordnet (Vgl. Figure 4). Im diesem Beispiel ist zu sehen, dass die definierte Anordnung wie folgt ist: Wenn der Wert eines Knotens kleiner ist als die Wurzel, so wird dieser links platziert, und dann wieder mit dem nächsten Knoten verglichen. Sofern dieser nun grösser ist, wie z.B Knoten 12, wird dieser rechts vom Knoten 5 platziert.
Ein weiterer Begriff, der wichtig ist im Kontext mit Bäumen ist die **Höhe** und **Tiefe** eines Baumes. Diese beiden Begriffe sind wie folgt definiert:

- $depth(v)$ = distance v to root (along unique path)
- $height(T) = max_{v \in V} depth(v) + 1$

Man fasst die Knoten eines Baumes gleicher Tiefe zu **Niveaus** zusammen. Die Knoten auf dem Niveau i sind alle Knoten der Tiefe i. Die Höhe des Beispiels (4) wäre also der Pfad von $(15 - 5 - 12 - 10 - 6 - 7) + 1 = 6$ und die Tiefe vom Knoten 20 wäre: 2.
Zusätzlich gilt: die Tiefe eines binary tree ist gleich wie die Höhe eines binary trees.[1] Des weiteren heisst ein Baum **vollständig** wenn er auf jedem Niveau die maximal mögiche Knotenanzahl hat und sämtliche Blätter dieselbe Tiefe haben. Auch wie bei dem Heap gelten die "Basic operations": Einfügen, löschen, suchen eines Elementes und das Traversieren des Baumes (Vgl. 3.1.2).

[1] weitere Beispiele & Erklärungen

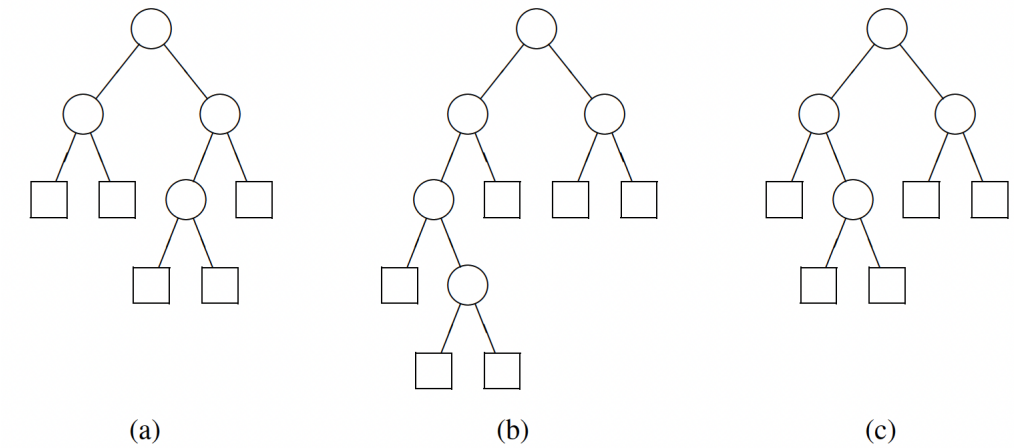
Figure 4: Beispiel eines geordneten Binärbaumes



3.3 AVL-Trees

Ein binärer Suchbaum ist *AVL-ausgegelichen* oder höhenbalanciert. Kurz: Ein AVL Baum, wenn für jeden Knoten p des Baumes gilt, dass sich die Höhe des linken Teilbaumes von der Höhe des rechten Teilbaumes von p um höchstens 1 unterscheidet. In der folgenden Abbildung(5) ist in (a) und (c) ein AVL-Baum zu sehen. (b) hingegen ist kein AVL-Baum.[2]

Figure 5: Beispiel von zwei AVL-Bäume



AVL-Bäume mit N inneren Knoten und $N+1$ Blättern haben eine höhe von $\mathcal{O}(\log N)$. Man überlegt sich was die minimale Blatt-und Knotenzahl eines AVL-Baumes gegebener Höhe h ist. Es gilt: AVL-Baum der Höhe 1 hat 2 Blätter und ein AVL-Baum der Höhe 2 mit minimaler Blattzahl hat 3 Blätter (Vgl. Figure 6). Ein AVL-Baum der Höhe $h+2$ mit minimaler Blattzahl erhält man, wenn man je einen AVL-Baum mit der Höhe $h+1$ und h mit minimaler Blattzahl wie in Figure (7) zu einem Baum der Höhe $h+2$ zusammenfügt.

[2] Animationen von AVL-Trees

Figure 6: AVL-Baum mit Höhe 1 und 2

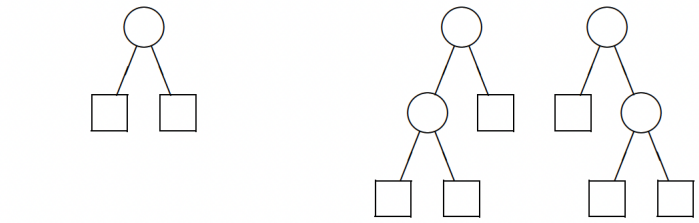
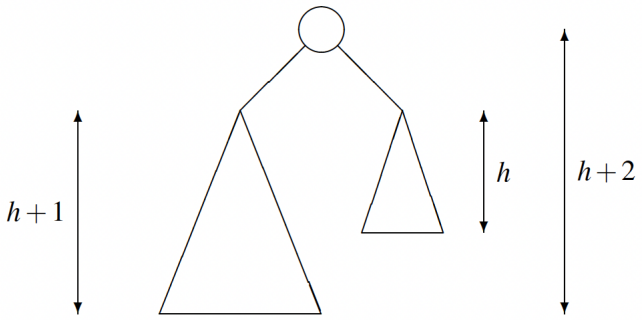


Figure 7: AVL-Baum



3.3.1 Suchen, Einfügen und Entfernen in AVL-Bäumen

Da AVL-Bäume eigentlich binäre Suchbäume sind, kann man ihnen nach einem Schlüssel genauso suchen wie in einem natürlichen Baum. Im schlechtesten Fall ist die Suche nach einem Schlüssel von Wurzel bis zu einem Blatt. Da die Höhe logarithmisch beschränkt ist, wird dies im worst case $\mathcal{O}(\log N)$ Schritte benötigen, wobei der worst case in einem normalen binären Suchbaum $\mathcal{O}(N)$ ist (alle links oder rechts an der Wurzel).

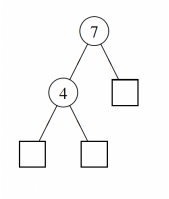
3.3.2 Einfügen

Damit der Schlüssel eingefügt werden kann, überprüfen wir in einem ersten Schritt nach dem Schlüssel im Baum. Wenn dieser Schlüssel bereits vorhanden ist, endet der Vorgang. Sofern dieser aber nicht in dem Baum ist fügen wird diesen ein wie bei einem normalen Baum.

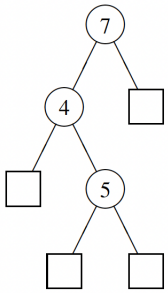
Das Problem welches wir nun erhalten ist die Aufrechterhaltung der AVL-Eigenschaft. Wenn wir bei Figure (8a) den Key 5 einfügen wollen, wird im nächsten Schritt Figure (8b) die Bedingung verletzt, da die Höhen des linken und rechten Teilbaumes sich um mehr als 1 unterscheiden. Man muss also die AVLAusgeglichenheit wieder herstellen. Dazu läuft man von der Einfügestelle den Suchpfad entlang zur Wurzel zurück und prüft an jedem Knoten (Vgl. 2, ob die Höhendifferenz zwischen linkem und rechtem Teilbaum noch innerhalb der vorgeschriebenen Grenzen liegt. Ist das nicht der Fall, führt man eine so genannte Rotation oder eine Doppelrotation durch (Vgl. 3.3.4), die die Sortierung der Schlüssel nicht beeinflusst, aber die Höhendifferenzen in den richtigen Bereich bringt.

Man könnte vermuten, dass man zur Prüfung der Höhenbedingung an einem Knoten im Baum die Höhen der Teilbäume des Knotens kennen muss. Das ist jedoch glücklicherweise nicht der Fall. Es genügt, an jedem inneren Knoten p den so genannten **Balancefaktor** $bal(p)$ mitzuführen, der wie folgt definiert ist:

$$bal(p) = \text{Höhe des rechten Teilbaumes von } p - \text{Höhe des linken Teilbaumes von } p \qquad (2)$$



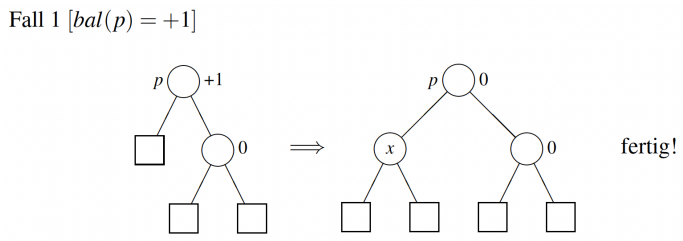
(a) Ausgangslage



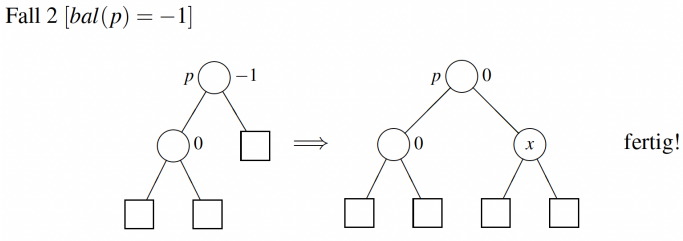
(b) AVL-Bedingung verletzt

Figure 8: Wie die AVL-Eigenschaft verletzt werden kann

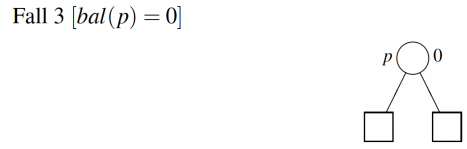
AVL-Bäume sind offenbar gerade dadurch charakterisiert, dass für jeden inneren Knoten p gilt: $bal(p) \in \{-1, 0, +1\}$. Um diese drei Fälle anzusehen, vergleichen wir diese in den folgenden Abbildungen 9:



(a) $bal(p) = +1$



(b) $bal(p) = -1$



(c) $bal(p) = 0$

Figure 9: Veranschaulichung von $bal(p) \in \{-1, 0, +1\}$ und dessen Balancierung

3.3.3 Entfernen

Wie beim Einfügen in den AVL-Tree gehen wir zuerst so vor wie in natürlichen Suchbäumen. Man sucht den zu entfernenden Schlüssel, findet man diesen nciht, so ist das Entfernen beendet. Falls dies nicht der Fall ist, haben wir drei Fälle der Entfernung.

Fall 1

Knoten n hat zwei blätter als Kinder. Sei p der Elternknoten von $n \implies$ Anderer Teilbaum hat Höhe $h' = 0$, 1 oder 2.

- h' =1: $bal(p)$ anpassen
- h' =0: $bal(p)$ anpassen. Aufruf von $upout(p)$
- h' =2: Rebalancieren des Teilbaumes. Aufruf von $upout(p)$

Fall 2

Knoten n hat einen inneren Knoten k als Kind

- Ersetze n durch k - $upout(k)$ 10

Fall 3

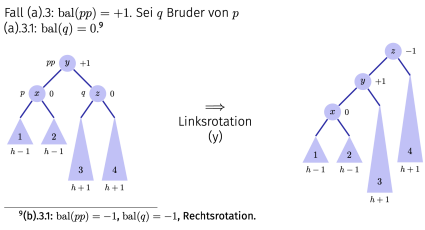
Knoten n hat zwei inneren Knoten als Kinder

- Ersetze n durch symmetrischen Nachfolger- $upout(k)$ 10
- Löschen des symmetrischen Nachfolgers wie in Fall 1 oder 2.

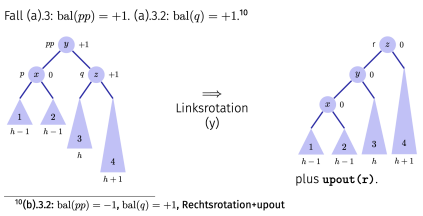
upout(p)

Sei pp der Elternknoten von p

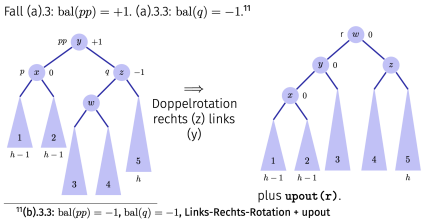
- p linkes Kind von pp
  1. $bal(pp) = -1 \implies bal(s) \leftarrow 0$ $upout(pp)$
  2. $bal(pp) = 0 \implies bal(s) \leftarrow 1$
  3. $bal(pp) = +1 \implies$ (Vgl. unten 10)
- p rechtes Kind von pp: Symmetrische Fälle unter Veratuschung von -1 und $+1$



(a) Erster Schritt



(b) Zweiter Schritt



(c) Dritter Schritt

Figure 10: $upout(p)$, Mit (a) ist der Erste bullet point (p linkes Kind von pp) gemeint

3.3.4 Rotationen

In den vorherigen Abschnitten (3.3.2, 3.3.3) haben wir bereits **Rotationen** gesehen. Wir unterscheiden insgesamt 4 Fälle: LR, RL, LL, RR.
LL und RR müssen jeweils nur eine Operation durchführen, wohingegen LR und RL zwei Operationen durchführen müssen.

Figure 11: Alle Fälle zum AVL-Tree zu balancieren

3.3.5 Operationen und Komplexität

Der Baum braucht genau $\Theta(n)$ Speicher. Er verfügt über die Standartoperationen von Suchen, Einfügen und Entfernen, die eine Komplexität von $\mathcal{O}(\log n)$ haben.

3.4 Sortier-Algorithmen

Zunächst wollen wir das Sortierproblem genauer fixieren: Wir nehmen an, es sei eine Menge von Sätzen gegeben; jeder Satz besitzt einen Schlüssel. Zwischen Schlüsseln ist eine Ordnungsrelation „$<$" oder „$\leq$" erklärt. Außer der Schlüsselkomponente können Sätze weitere Komponenten als „eigentliche" Information enthalten.

3.4.1 Eigenschaften von Sortier-Algorithmen

Stability Zwei Objekte mit gleichen Schlüsseln erscheinen in der (sortierten) Ausgabe in der gleichen Reihenfolge, wie sie in der Eingabe erschienen.

In-place oder in-situ beschreiben jene Algorithmen, die die Eingabe ohne zusätzliche Datenstruktur und damit ohne zusätzlichen Speicherplatz umwandeln. In einigen Fällen ist zusätzlicher Speicherplatz zur Speicherung einiger Variablen zulässig.

Table 2: Properties of sorting algorithms

	bubble	**insert**	**select**	**quick**	**merge**	**heap**
stable	Yes	Yes	No	No	Yes	No
in-situ	Yes	Yes	Yes	Yes	No	Yes

3.4.2 Bogo sort

Bogo sort $\mathcal{O}(n \cdot n!)$ - This is a fairly inefficient sorting algorithm that generates a random permutation until the elements are sorted. An analogy with a deck of card would be to shuffle the deck, then check if it is sorted and loop if it is not.

Algorithm 6 Bogo Sort

```
while ¬isSorted(A) do
    A ← randomPermutation(A)
end while
```

This algorithm doesn't have a worst case asymptotic time as it is not guaranteed to terminate within a given time.

3.4.3 Stalin-Sort

The Stalin sort is a sort in which elements that are out of order get removed from a list. For example, [1, 2, 5, 3, 6, 4, 10] becomes [1, 2, 5, 6, 10]. It is said to runs in $\mathcal{O}(n)$ time but if coded that way, the final list may have fewer than the maximum number of elements possible. For example [10, 1, 2, 3, 4] would become [10].

3.4.4 Bubblesort

Bubblesort $\mathcal{O}(n^2)$ - ist ein Verfahren, welches nach dem Motto *Sortierten durch Einfügen* leben. Bubblesort ist ein Sortierverfahren, das solange zwei jeweils benachbarte, nicht in der richtigen Reihenfolge stehende Elemente vertauscht, bis keine Vertauschungen mehr nötig sind. Um ein genaueres Verständnis zu erlangen ein Beispiel:

1. (Schritt 6 & 5)	**2.**	**3.**	**4.**
6 5 3 1 8 7 2 4	3 5 1 6 7 2 4 8	1 3 5 6 2 4 7 8	1 3 2 5 4 6 7 9
5 6 3 1 8 7 2 4	3 1 5 6 7 2 4 8	1 3 5 6 2 4 7 8	1 3 2 4 5 6 7 9
5 3 6 1 8 7 2 4	3 1 5 6 7 2 4 8	1 3 5 2 6 4 7 8	1 3 2 4 5 6 7 9
5 3 1 6 8 7 2 4	3 1 5 6 7 2 4 8	1 3 5 2 4 6 7 8	1 3 2 4 5 6 7 9
5 3 1 6 8 7 2 4	3 1 5 6 2 7 4 8	1 3 5 2 4 6 7 8	1 3 2 4 5 6 7 9
5 3 1 6 7 8 2 4	3 1 5 6 2 4 7 8	1 3 5 2 4 6 7 8	1 3 2 4 5 6 7 9
5 3 1 6 7 2 8 4	3 1 5 6 2 4 7 8	1 3 5 2 4 6 7 9	1 2 3 4 5 6 7 9
5 3 1 6 7 2 4 8	1 3 5 6 2 4 7 8	1 3 5 2 4 6 7 9	

Table 3: Tabelle wird von oben nach unten, links nach rechts interpretiert

Algorithm 7 Bubble sort

```
swapped ← true
while swapped do
    swapped ← false
    for i ∈ {0, ..., A.length − 1} do
        if A[i] > A[i + 1] then
            Swap A[i] and A[i + 1]
            swapped ← true
        end if
    end for
end while
```

3.4.5 InsertionSort

InsertionSort $\mathcal{O}(n^2)$ [worst and average case] - ist ein Algorithmus, der der Art und Weise, wie ein Mensch ein Kartenspiel sortiert, sehr ähnlich ist. Er geht so vor, dass er eine Folge von bereits sortierten Elementen behält und immer das nächste Element in der Folge berücksichtigt. Dieses Element wird dann an der richtigen Stelle in der sortierten Folge eingefügt, wobei alle größeren Elemente nach rechts verschoben werden.

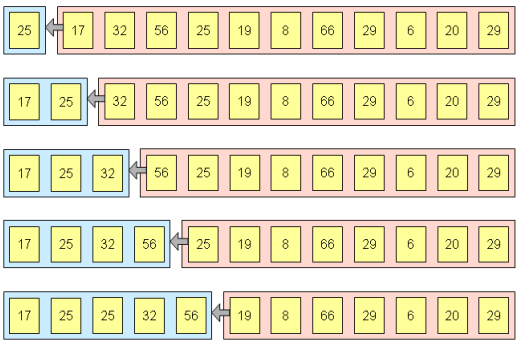


Figure 12: InsertionSort-Beispiel

Algorithm 8 Insertion sort

for $i \in \{0, \ldots, A.\texttt{length} - 1\}$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i - 1$
  while $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  end while
  $A[j+1] \leftarrow v$
end for

3.4.6 SelectionSort

Selection Sort $\Theta(n^2)$ - Dieser Algorithmus behält ebenfalls eine Folge von sortierten Elementen bei und erweitert sie iterativ um das größte der verbleibenden unsortierten Elemente. Dann tauscht er dieses Element mit demjenigen aus, das den sortierten Elementen am nächsten liegt. Dadurch wird das nächste Element ausgewählt, daher der Name.

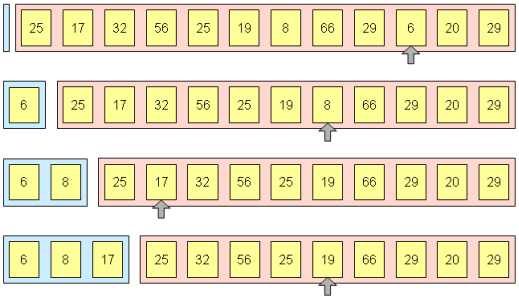


Figure 13: SelectionSort-Beispiel

Algorithm 9 Selection sort

```
for i ∈ {0, A.length − 2} do
    m, v ← i, A[i]
    for j ∈ {i + 1, ..., A.length − 1} do
        if A[j] < v then
            m, v ← j, A[j]
        end if
    end for
    if m ≠ i then
        Swap A[i] and A[m]
    end if
end for
```

Complexity

Worst case: $\mathcal{O}(n^2)$, Best case: $\mathcal{O}(n^2)$:
$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2-n}{2} = \frac{n^2-n}{2} = \Theta(n^2)$

3.4.7 Quicksort

Quicksort $\mathcal{O}(n \log n)$ - Dieser Algorithmus funktioniert, indem er rekursiv einen Pivot (ein Element, normalerweise an der ersten oder letzten Position im Array) wählt und alle anderen Elemente in Bezug auf diesen aufteilt. Er erstellt zunächst die Mengen S^- und S^+, in denen alle Elemente, die kleiner/größer als der Drehpunkt sind, gespeichert werden. Dann ruft es sich selbst rekursiv auf diesen beiden Mengen auf, und nachdem sie sortiert zurückgegeben wurden, fügt es sie einfach zusammen, wobei es den Drehpunkt in der Mitte hinzufügt. Die Rekursion endet im Basisfall, in dem das Array nur ein Element enthält. Es ist zu beachten, dass auch eine In-Place-Implementierung möglich ist, die den Speicher-Overhead reduziert. Dieser Algorithmus hat eine schlechtere obere Schranke für die Operationszeit, wird aber in der Praxis häufig verwendet, da die durchschnittliche Zeit besser ist als bei Merge und Heap Sort. Beachten Sie, dass der ·-Operator hier Verkettung bedeutet.

Algorithm 10 Quicksort

```
function QuickSort(A)
    if A.length ≤ 1 then
        return A
    end if
    Pick pivot p ← A[0]
    for i ∈ {1, ..., A.length − 1} do
        if A[i] ≤ p then
            S⁻.add(A[i])
        else
            S⁺.add(A[i])
        end if
    end for
    S⁺ ← QuickSort(S⁺)
    S⁻ ← QuickSort(S⁻)
    return S⁻ · {p} · S⁺
end function
```

3.4.8 Mergesort

Mergesort $\mathcal{O}(n \log n)$ - Dieser Algorithmus arbeitet ebenfalls rekursiv. Er teilt die Eingabe in zwei Mengen auf und ruft sich dann selbst rekursiv auf diesen Mengen auf. Nachdem die beiden Mengen sortiert zurückgegeben wurden, führt er sie in linearer Zeit zusammen. Der Name leitet sich von diesem letzten Teil des Algorithmus ab.

Algorithm 11 Merge sort

```
function MergeSort(A)
    if A.length ≤ 1 then
        return A
    end if
    n ← A.length
    n_1 ← n/2
    n_2 ← n − n_1
    for i ∈ {0, ..., A.length − 1} do
        if i < n_1 then
            S^−.add(A[i])
        else
            S^+.add(A[i])
        end if
    end for
    S^+ ← MergeSort(S^+)
    S^− ← MergeSort(S^−)
    i ← 0                                          ▷ Merge S^+ and S^−
    j ← 0
    while i < n_1 and j < n_2 do
        if S^−[i] ≤ S^+[j] then
            R.add(S^−[i])
            i ← i + 1
        else
            R.add(S^+[j])
            j ← j + 1
        end if
    end while
    Concatenate remaining elements to R
    return R
end function
```

3.4.9 Heapsort

Heapsort $\mathcal{O}(n \log n)$ - Dieser Sortieralgorithmus arbeitet, indem er alle Schlüssel in einen Min-Heap einfügt und iterativ das Minimum (die Wurzel) in konstanter Zeit extrahiert und an die Liste der sortierten Knoten anhängt. Er macht dies n Mal; die Gesamtlaufzeit hängt von der Komplexität der Standard-Heap-Operationen ab.

Algorithm 12 Heap sort

```
H ← CreateMinHeap()
for i ∈ {0, ..., A.length − 1} do
    H.insert(A[i])
end for
for i ∈ {0, ..., A.length − 1} do
    v ← H.pop()
    R.add(v)
end for
return R
```

3.4.10 Kosten von Sortieralgorithmen

Hier werden die minimalen und maximalen Kosten für einige Sortieralgorithmen auf der Grundlage ihrer Eingabegröße n angegeben, sowie die besondere Art der Eingabe, die zu dieser besonderen Komplexität führt.

Table 4: Best and worst case costs für sortier Algorithmen

	bubblesort		**insertion sort**		**selection sort**		**quicksort**	
	best case	worst case	best case	w.c.	best case	w.c.	best case	w.c.
# comparisons	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$
# permutations	0	$\Theta(n^2)$	0	$\Theta(n^2)$	0	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
corresponding order	A	B	A	B	A	C	C	C

- A = already ordered
- B = inverse order
- C = special order

3.4.11 Lower-bound für sortier Algorithmen

Jegliche Sortieralgorithmen haben eine worst-case runtime von $\Omega(n \log n)$. Mehr dazu in *Algorithmen und Wahrscheinlichkeiten, ConvexHull*

ERGÄNZUNGEN MACHEN ZU HEAPSORT, RSP BEISPIELE WIE SICH DIES DURCHSPIELEN LÄSST

4 GraphenTheorie

ERGÄNZUNGEN MACHEN ZU HEAPSORT, RSP BEISPIELE WIE SICH DIES DURCHSPIELEN LÄSST

4.1 Grundlagen

ERGÄNZUNGEN MACHEN ZU HEAPSORT, RSP BEISPIELE WIE SICH DIES DURCHSPIELEN LÄSST