



**Westfälische  
Hochschule**

# **Hashing (Kollisionsbehandlung)**

Lennart Pries

15. Mai 2024

Algorithmen und Datenstrukturen

Prof. Dr. Martin Guddat, Dipl.-Ing. Dirk Gouders

Sommersemester 2024

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen des Hashing</b>	<b>2</b>
2.1	Was ist eine Hash-Funktion? . . . . .	2
2.2	Aufbau eines Hash-Wertes . . . . .	3
2.3	Hashtabelle . . . . .	4
<b>3</b>	<b>Kollisionen im Hashing</b>	<b>6</b>
3.1	Was sind Kollisionen? . . . . .	6
3.2	Kollisionsresistenz . . . . .	6
3.3	Kollisionsangriff . . . . .	7
<b>4</b>	<b>Hash-Algorithmen</b>	<b>8</b>
4.1	Hashing durch Division . . . . .	8
4.2	Hashing durch Multiplikation . . . . .	9
4.3	Kryptographische Hashfunktionen . . . . .	10
4.3.1	MD5 . . . . .	11
4.3.2	SHA-1 . . . . .	11
4.3.3	SHA-2 . . . . .	11
4.3.4	SHA-3 . . . . .	12
<b>5</b>	<b>Methoden zur Kollisionsbehandlung</b>	<b>14</b>
5.1	Verkettung (Seperate Chaining) . . . . .	14
5.2	Offene Hashverfahren (Open Addressing) . . . . .	15

5.2.1	Lineares Sondieren . . . . .	16
5.2.2	Quadratisches Sondieren . . . . .	17
5.2.3	Double Hashing . . . . .	17
5.2.4	Uniformes Sondieren und zufälliges Sondieren . . . . .	18
5.2.5	Coalesced Hashing . . . . .	19
5.3	Dynamisches Hashing . . . . .	19
5.4	Implementierung . . . . .	20
<b>6</b>	<b>Anwendungsbeispiele</b>	<b>23</b>
<b>7</b>	<b>Fazit</b>	<b>25</b>
<b>8</b>	<b>Quellen</b>	<b>26</b>
<b>9</b>	<b>Anhang</b>	<b>28</b>

# 1 Einleitung

Hashing ist eine zentrale Technik in der Informatik und findet Anwendung in unzähligen Bereichen. Es ist entscheidend beim Speichern von Passwörtern oder dem Datenzugriff von effizienten Datenstrukturen. Besonders in der Informatik spielt es eine große Rolle in vielen Bereichen. Die effiziente Handhabung dieser Prozesse hängt stark von der Qualität der verwendeten Hash-Algorithmen und der effektiven Behandlung von Kollisionen ab.

Das Verständnis verschiedener Hashing-Methoden und die Fähigkeit, Kollisionen angemessen zu adressieren, sind daher für jeden, der in der Informatik tätig ist, von großer Bedeutung.

Begleitend zu diesem Thema gibt es ein Github Repository<sup>1</sup>, in dem alle verwendeten Code-Beispiele zu finden sind.

---

<sup>1</sup><https://github.com/lennart02/Hashing>

## 2 Grundlagen des Hashing

Ein zentraler Bestandteil des Hashings ist die Hash-Funktion, auch Streuwertfunktion genannt, die Daten beliebiger Größe auf eine kleine, feste Anzahl von Werten abbildet, den sogenannten Hash-Wert.

### 2.1 Was ist eine Hash-Funktion?

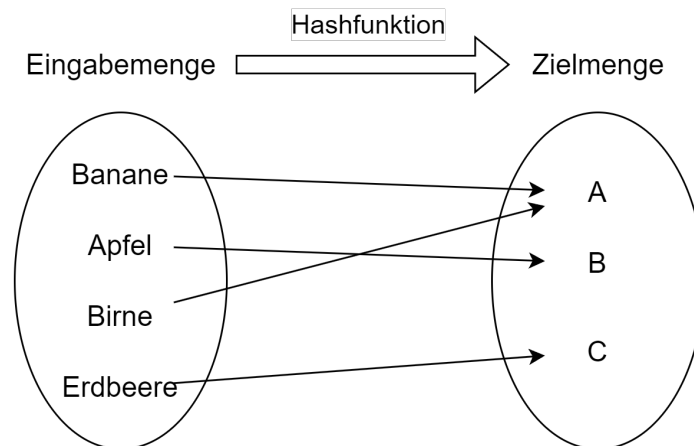


Abbildung 1: Beispiel für eine nicht injektive Funktion (eigene Darstellung)

Eine Hash-Funktion ist eine nicht injektive mathematische Funktion, die eine beliebige Eingabedatenmenge auf eine feste Länge von Bytes oder Bits abbildet, die als Hash-Wert bezeichnet wird (s. Abbildung 1). Eine Hash-Funktion ist eine Einweg-Funktion, weshalb aus dem Hashwert kein eindeutiger Rückschluss auf die ursprüngliche Eingabe gezogen werden kann. Zudem ist die Funktion deterministisch, was bedeutet, dass für jede bestimmte Eingabe stets derselbe Hash-Wert erzeugt wird [1].

Darüber hinaus soll ein Hash-Algorithmus streuend sein, was bedeutet, dass kleine Änderungen in der Eingabe zu drastischen Änderungen im Hash-Wert führen sollten. Aus diesem Grund wird sie auch Streuwertfunktion genannt. Die Quelldaten, aus denen ein Hash erstellt werden kann, können jede Art von Daten umfassen,

seien es Zahlen, Texte, Bilder oder Binärdaten.

Eine der einfachsten Hashfunktionen ist das Bilden der Quersumme einer Zahl.

Eingabe	Quersumme (Hash-Wert)
123	$1 + 2 + 3 = 6$
456	$4 + 5 + 6 = 15$
789	$7 + 8 + 9 = 24$
987	$9 + 8 + 7 = 24$

Tabelle 1: Beispiel für die Bildung der Quersumme als Hash-Wert (eigene Darstellung)

Dies wird auch Summen-Hash-Funktion genannt. Da sich mithilfe von ASCII, Zeichen auch als Zahlen darstellen lassen, kann man diese Hash-Funktion auch auf Zeichen anwenden. Für die Eingabe "Hallo" würde die additive Hash-Funktion zuerst ASCII-Wert aller Buchstaben berechnen: 'H' = 72, 'a' = 97, 'l' = 108, 'l' = 108, 'o' = 111. Die ASCII-Werte werden addiert:  $72 + 97 + 108 + 108 + 111 = 496$ . Man erhält den Hash-Wert 496. Die Summen-Hash-Funktion ist aber eine eher untypische Hashfunktion, da der Hash-Wert keine feste Länge hat.

## 2.2 Aufbau eines Hash-Wertes

Der Hash-Wert, der durch einen Hash-Algorithmus erzeugt wird, besteht typischerweise aus einer festgelegten Anzahl von Bytes oder Bits. Dieser Wert dient als einzigartiger Fingerabdruck für die Eingabedaten. Ein guter Hash-Wert sollte sich drastisch ändern, selbst wenn eine kleine Änderung an der Eingabe vorgenommen wird. Dies gewährleistet, dass zwei unterschiedliche Eingaben mit hoher Wahrscheinlichkeit verschiedene Hash-Werte erzeugen. Typischerweise werden Hash-Werte in hexadezimaler oder binärer Form dargestellt. Betrachten wir die folgenden

zwei Eingaben und ihre entsprechenden Hash-Werte:

Eingabe	MD5 Hash-Wert
Hello	8b1a9953c4611296a827abf8c47804d7
Hallo	d1bf93299de1b68e6d382c893bf1215f

Tabelle 2: Beispiel für MD5 Hash-Werte bei leicht unterschiedlichen Eingaben (eigene Darstellung)

Obwohl sich nur ein einziges Zeichen in der Eingabe von “e” zu “a” geändert hat, führt dies zu einem vollständig anderen Hash-Wert als Ergebnis. Dies verdeutlicht die Eigenschaft eines guten Hash-Algorithmus, dass selbst minimale Änderungen in der Eingabe zu erheblichen Veränderungen im Hash-Wert führen.

## 2.3 Hashtabelle

Eine Hashtabelle ist eine effiziente Datenstruktur, die es ermöglicht, Daten schnell zu finden, einzufügen und zu löschen. Jedem Element wird über eine Hashfunktion ein spezifischer Index zugewiesen. Dieser Index bestimmt den Speicherplatz des Elements in der Hashtabelle. Dadurch kann die Suchoperation mit einem konstanten Zeitaufwand realisiert werden ( $O(1)$ ).

Die Größe  $m$  der Hashtabelle sollte der Größenordnung der Anzahl der zu speichernden Elemente  $n$  entsprechen. Eine besondere Metrik dabei ist der Belegungsfaktor  $\alpha$ , definiert als:

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

Der Belegungsfaktor  $\alpha$  gibt den Prozentsatz der Hashtabelle an, der aktuell belegt ist [2]. Für optimale Performance sollte  $\alpha$  kleiner als 1 sein,  $\alpha < 1$ . Ein niedriger Belegungsfaktor minimiert die Wahrscheinlichkeit von Kollisionen (siehe Kapitel 3.1).

Durch ihre hohe Effizienz bei Zugriffsoperationen sind Hashtabellen in vielen Anwendungen verbreitet, wie z. B. Datenbanken, Caches und Assoziativspeichern.



## 3 Kollisionen im Hashing

Beim Hashing spricht man von einer Kollision, wenn zwei unterschiedliche Eingabemengen, zu dem gleichen Hashwert generiert werden. Dies ist unvermeidbar, da Hashfunktionen nicht injektiv sind und somit die Eigenschaft haben, dass die Zielmenge kleiner ist als die Eingabemenge. In der Abbildung 1 sieht man, dass zu den beiden Eingaben Banane und Birne jeweils der Hashwert A generiert wird.

### 3.1 Was sind Kollisionen?

Hash-Funktionen sind so konzipiert, dass sie eine schnelle Berechnung des Hash-Wertes ermöglichen und eine gleichmäßige Verteilung der Hash-Werte über den Hash-Raum gewährleisten. Dadurch wird die Wahrscheinlichkeit von Kollisionen minimiert, bei denen unterschiedliche Eingabedaten denselben Hash-Wert erhalten. Solche Kollisionen sind jedoch nie vollständig vermeidbar, daher ist es wichtig, effiziente Methoden zur Kollisionsbehandlung zu implementieren, wie zum Beispiel das Verketteten von Einträgen oder das offene Adressieren (s. Kapitel 5).

Ziel eines Hash-Algorithmus ist es, die Kollision möglichst gering zu halten, damit die Streuung somit möglichst verteilt ausfällt.

### 3.2 Kollisionsresistenz

Eine Hashfunktion wird als kollisionsresistent bezeichnet, wenn es nur mit einem erheblichen Aufwand möglich ist, Eingaben zu finden, die auf denselben Wert abgebildet werden.

Eine perfekte Kollisionsresistenz gibt es nur, wenn die Hashfunktion  $H$  injektiv ist. Dies steht aber im direkten Widerspruch mit den Anforderungen einer Hashfunktion. In diesem Fall hätte die Ausgabe, dieselbe Länge wie die Eingabe [3].

### 3.3 Kollisionsangriff

Einen Algorithmus zum Finden einer Kollision in einer Hashfunktion wird Kollisionsangriff genannt. Solche Angriffe zielen darauf ab, zwei unterschiedliche Eingaben zu finden, die den gleichen Hashwert erzeugen. Die Effektivität eines solchen Angriffs wird oft durch die kryptographische Sicherheit der Hashfunktion bestimmt.

Die Anzahl an Funktionsaufrufen einer Hashfunktion, um eine Kollision von zwei Hashwerten zu finden, wird folgendermaßen definiert:

$$C = 2^{n/2}$$

Dabei ist  $n$  die Hashlänge [4]. Für einen 128-bit Hash, wird eine Kollision nach durchschnittlich  $18,446 \cdot 10^{18}$  Aufrufen gefunden. Für einen Rechner, der 1 Milliarde Versuche pro Sekunde schafft, beträgt die Laufzeit hier ca. 585 Jahre.

Kollisionsangriffe spielen eine wichtige Rolle in der Kryptographie, weil sie die Zuverlässigkeit von digital signierten Dokumenten beeinträchtigen können. Wenn jemand zwei verschiedene Dokumente erstellen kann, die denselben Hashwert produzieren, könnte diese Person die Signatur eines Dokuments fälschlicherweise auf ein anderes übertragen.

## 4 Hash-Algorithmen

In der Entwicklung von Hashfunktionen sind heuristische Techniken ein wichtiger Bestandteil. Eine effektive Hashfunktion muss jedes Bit des Schlüssels berücksichtigen, um sicherzustellen, dass selbst geringfügige Unterschiede in den Schlüsseln zu deutlich verschiedenen Hashwerten führen. Die in diesem Kapitel beschriebenen heuristischen Methoden umfassen das Hashing durch Division und das Hashing durch Multiplikation. Beide Ansätze zielen darauf ab, die Schlüssel gleichmäßig über die Hash-Tabelle zu verteilen und Kollisionen von vornherein zu minimieren.

### 4.1 Hashing durch Division

Bei der Divisions-Rest-Methode wird der Schlüssel  $k$  durch eine feste Zahl  $m$  geteilt und der Modulo wird für die Berechnung des Hashes verwendet. Die Hashfunktion kann als

$$h(k) = k \mod m$$

definiert werden. Die Wahl von  $m$  ist kritisch, da eine schlechte Wahl zu vielen Kollisionen führen kann. Eine gute Wahl ist dabei eine Primzahl, die nicht zu nah bei einer Zweierpotenz liegt [1]. Dies hat den Grund, dass viele Computersysteme und Datenformate dazu neigen, Werte zu erzeugen, die in binärer Form gespeichert und manipuliert werden. Würde man die feste Zahl  $m$  mit einer 2er-Potenz festlegen, kommt es zu Cluster-Bildungen und in der Ergebnismenge kommt es zu häufigen Kollisionen. Bei der Implementierung einer Hashtabelle, ist  $m$  oft die Größe der Hashtabelle.

Im Folgenden ist eine Implementierung dieses Verfahrens in C zu sehen.

```
1 public class DivisionHash
2 {
3     private int tableSize;
4 }
```

```
5     public DivisionHash(int size)
6     {
7         tableSize = size;
8     }
9
10    public int Hash(int key)
11    {
12        return key % tableSize;
13    }
14 }
```

Code-Ausschnitt 1: Implementierung des Hashings durch Division in C# (eigene Darstellung)

## 4.2 Hashing durch Multiplikation

Das Hashing durch Multiplikation (auch Fibonacci Hashing) verwendet eine Konstante  $A$  ( $0 < A < 1$ ). Die Hashfunktion lautet

$$h(k) = \lfloor m(kA \bmod 1) \rfloor.$$

Die Funktion bietet eine gute Verteilung, wenn  $A$  sorgfältig gewählt wird. Oft ist es eine irrationale Zahl nahe  $(\sqrt{5} - 1)/2$ , der goldenen Ratio [5]. Der Vorteil gegenüber dem Hashing durch Division liegt darin, dass trotz einer beliebigen Hashtabellengröße  $m$  stets eine gleichmäßige Verteilung der Daten generiert wird.

Im Folgenden ist eine Implementierung des Hashings durch Multiplikation in C zu sehen.

```
1 public class MultiplicationHash
2 {
3     private readonly double A = (Math.Sqrt(5) - 1) / 2;
4     private int tableSize;
```

```
5
6  public MultiplicationHash(int size)
7  {
8      tableSize = size;
9  }
10
11 public int Hash(int key)
12 {
13     double fractionalPart = key * A % 1;
14     return (int)(tableSize * fractionalPart);
15 }
16 }
```

Code-Ausschnitt 2: Implementierung des Hashings durch Multiplikation in C# (eigene Darstellung)

### 4.3 Kryptographische Hashfunktionen

Kryptographische Hashfunktionen sind eine spezielle Art der Hashfunktionen, die für kryptographische Anwendungszwecke ausgelegt sind. Kryptographische Hashfunktionen gelten als irreversibel. Das heißt, aus dem Hash lassen sich kaum Rückschlüsse auf den Originalwert zurückführen. Darüber hinaus erfordert es einen erheblichen Aufwand, zwei unterschiedliche Nachrichten zu finden, die denselben Hashwert produzieren, was die Kollisionsresistenz zu einem entscheidenden Faktor macht [6].

Es gibt viele verschiedene Implementierungen von Hash-Funktionen. Unterschiedliche Algorithmen bieten verschiedene Eigenschaften hinsichtlich Geschwindigkeit, Sicherheit und Resistenz gegen Kollisionen. Während dieses Kapitel einige der am weit verbreiteten und aktuell relevanten kryptographischen Hash-Algorithmen wie SHA-256 und SHA-3 vorstellt, gibt es auch andere bekannte Algorithmen, die heutzutage von weniger Relevanz sind und deshalb nicht im Detail behandelt werden.

Dazu gehören z. B. die Algorithmen Blake, RipeMd, Whirlpool, Tiger, HAVAL und Skein.

#### **4.3.1 MD5**

MD5 (Message Digest Algorithm 5) ist ein Hashing-Algorithmus, welcher einen 128-Bit Hash Wert erzeugt. Dieser Algorithmus ist sehr verbreitet und wird häufig für die Integritätsprüfung verwendet. Früher hat man ihn auch für die Speicherung von Passwörtern verwendet, allerdings ist es aufgrund von Schwächen in seiner Kollisionsresistenz und der Möglichkeit von Brute-Force-Angriffen nicht mehr für sicherheitskritische Anwendungen, wie dem Password-Hashing geeignet. Obwohl der Algorithmus nicht mehr empfohlen wird, spielt es historisch eine wichtige Rolle in der Entwicklung von Hash-Funktionen und wird immer noch in älteren Systemen oder für weniger sicherheitskritische Anwendungen verwendet [7].

#### **4.3.2 SHA-1**

Die bekanntesten kryptografischen Algorithmen sind die der SHA Familie, welche SHA-1, SHA-2 und SHA-3 beinhalten. SHA-1 erzeugt einen 160-Bit-Hash-Wert und wurde 1993 von dem National Institute of Standards and Technology (NIST) zuerst als SHA-0 veröffentlicht und zwei später zu SHA-1 verbessert [8]. Seit 2017 gibt es einen "SSHattered prefix collision attack", welcher dazu führt, dass SHA-1 nicht mehr resistent gegenüber Kollisionsangriffen ist [9]. Es wird empfohlen daher nur noch SHA-2 oder SHA-3 zu verwenden.

#### **4.3.3 SHA-2**

Die SHA-2 Familie umfasst die Varianten SHA-224, SHA-256, SHA-384 und SHA-512. Sie wurden von der National Security Agency (NSA) entwickelt und veröffentlicht durch das NIST. Die Zahl im Namen der Varianten ist dabei immer äquivalent zu

der Schlüssellänge in Bits. Am populärsten ist davon der SHA-256 Algorithmus. Dieser produziert einen Hash-Wert von 256 Bits und ist weit verbreitet in der Sicherung digitaler Daten und der Implementierung von Blockchain-Technologien. Z. B. findet man ihn in Anwendungen wie digitale Signaturen oder TLS (Transport Layer Security) und ist somit für die Sicherheit von HTTPS Zertifikaten verantwortlich [10].

Der Algorithmus gilt als kollisionsresistent, denn bisher wurde kein Kollisionsangriff gefunden, der schneller als eine Brute-Force-Attacke ist [4].

#### 4.3.4 SHA-3

SHA-3 wurde als robuste Alternative zu den älteren SHA-1 und SHA-2 Algorithmen entwickelt. Die Auswahl des SHA-3 Algorithmus wurde 2012 von der NIST Organisation durch einen Wettbewerb getroffen. Dabei standen mehrere Algorithmen zur Auswahl, die die damals bekannten Schwachstellen ausbessern sollten. Gewonnen hat dabei der Keccak Algorithmus, der zum neuem SHA-3 Standard wurde [11]. SHA-3 ist genauso wie SHA-2 in einer Reihe von Varianten verfügbar, die unterschiedliche Digest-Größen bieten, einschließlich 224, 256, 384 und 512 Bits.

Name	Hash-Länge n	Nachrichten- blocklänge r	Kapazität c=1600-r	Sicherheit (Kollision)	Sicherheit (Urbild)	Padding-Schema
SHA3-224	224	1152	448	112	224	0110*1
SHA3-256	256	1088	512	128	256	
SHA3-384	384	832	768	192	384	
SHA3-512	512	576	1024	256	512	
SHAKE128	variabel	1344	256	$\min(n/2, 128)$	$\min(n, 128)$	111110*1
SHAKE256	variabel	1088	512	$\min(n/2, 256)$	$\min(n, 256)$	

Abbildung 2: Bitlänge der SHA3 Varianten [12]

Die beiden Varianten SHAKE128 und SHAKE256 sind Funktionen mit veränderbarer, bzw. erweiterbarer Ausgabe. Diese werden als XOFs (extendable-output functions) bezeichnet. Die Anzahl an Bits des Ausgabe-Hashes können also für jeden

Anwendungszweck verschieden festgelegt werden [12]. Bei Hashing-Algorithmen kann eine geringere Geschwindigkeit auch ein Vorteil für die Sicherheit sein, da sie die Dauer erhöht, die erforderlich ist, um erfolgreiche Brute-Force-Angriffe durchzuführen. Diese Zeitverlängerung für potenzielle Angriffe trägt dazu bei, die Sicherheit gegenüber Angriffen, die auf Geschwindigkeit angewiesen sind, zu erhöhen.

Diese robuste Konstruktion macht SHA-3 zu einer bevorzugten Wahl in Sicherheitsanwendungen, die höchste Ansprüche an die Integrität und Authentizität der Daten stellen.



## 5 Methoden zur Kollisionsbehandlung

Es gibt mehrere Verfahren, um Kollisionen zu behandeln, bzw. vorzubeugen. Im Folgenden wird beispielhaft davon ausgegangen, dass Kollisionen in Hashtabellen auftreten.

Das Problem sieht folgendermaßen aus: Wenn in eine Hashtabelle zwei Schlüssel  $k$  und  $k'$  eingefügt werden sollen, welche denselben Hashwert besitzen, kommt es zu einer Adresskollision. Der Adresse  $h(k) = h(k')$  ist schon besetzt. Der Überläufer  $k'$  muss entsprechend behandelt werden.

### 5.1 Verkettung (Seperate Chaining)

”So kann man etwa die Überläufer zu jeder Hashadresse in einer linearen Liste verketteten; diese Liste wird an den Hashtabelleneintrag angehängt, der sich durch Anwendung der Hashfunktion auf die Schlüssel ergibt” [13]. Diese Vorgehensweise wird auch *Hashverfahren mit Verkettung der Überläufer* genannt.

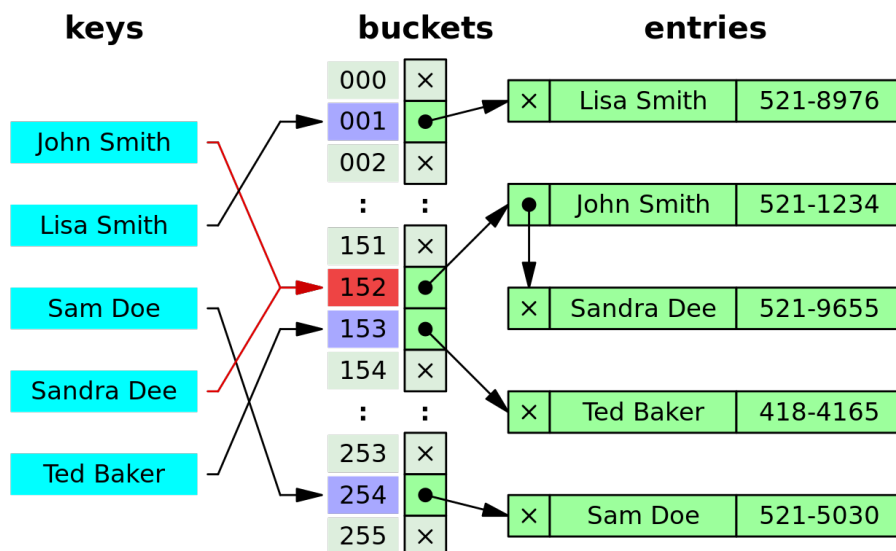


Abbildung 3: Kollisionsbehandlung durch Verkettung [2]

Die Hash-Tabelle besteht dabei aus mehreren Behältern (Buckets), die jeweils eine dynamische Datenstruktur enthalten (s. Abbildung 3). Dies ist meist eine verkettete Liste, kann aber auch als Binärbaum implementiert werden. Die Überläufer werden an das Ende der Liste angehängt. Jedes Element der Hashtabelle ist ein Zeiger auf eine Liste.

Operationen können in einer Zeitkomplexität von  $O(1)$  realisiert werden, wenn es eine sinnvolle Wahl der Tabellengröße gibt. Im Optimalfall gibt es wenige Elemente pro Bucket [2].

## 5.2 Offene Hashverfahren (Open Addressing)

Beim offenem Hashing werden alle Elemente in der Hashtabelle selbst gespeichert. Der Begriff "offenes Hashing" bezieht sich auf die Methode der offenen Adressierung [2]. Die Tabelle muss also größer-gleich der Anzahl aller Schlüssel sein. Während Verkettung Flexibilität bietet, können offene Hashverfahren den Speicherplatz effizienter nutzen. Allgemein wird ein Schlüssel  $k$ , dessen Position  $h(k)$  in der Hashtabelle bereits belegt ist, nach einer bestimmten Regel untergebracht.

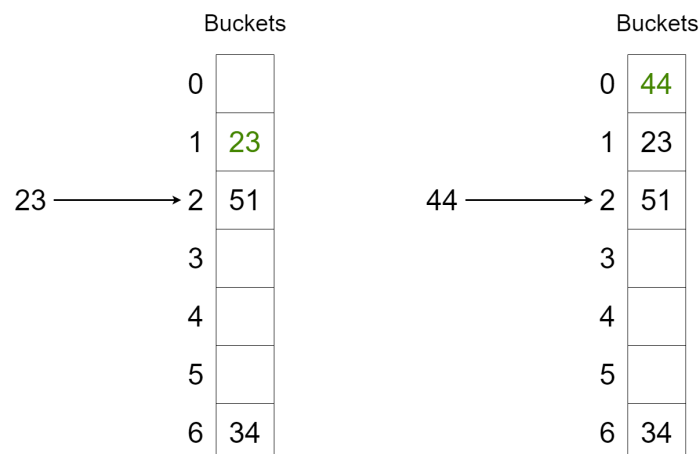


Abbildung 4: Offenes Hashverfahren mit linearem Sondieren (eigene Darstellung)

Ist der Platz eines Schlüssels  $k$  belegt, wird dieser z. B. immer davor einsortiert. In der Abbildung 4 müsste die Zahl 26 eigentlich in den Bucket 2 platziert werden. Da

dieses Feld aber besetzt ist, wird es davor einsortiert.

Das Entfernen von Schlüsseln ist problematisch. Das Entfernen eines Schlüssels kann dazu führen, dass andere Schlüssel, die aufgrund einer Kollision verschoben wurden, nicht mehr gefunden werden können. Entfernte Schlüssel werden daher nur als gelöscht markiert, bleiben aber technisch gesehen in der Tabelle, um Kollisionen und Suchprobleme zu vermeiden. Eine solche Markierung wird auch als Tombstone bezeichnet. Da dies die Effizienz beeinflusst, wird hauptsächlich nur eingefügt und gesucht [14].

Die *Sondierungsfolge* beschreibt die Methode und Reihenfolge, nach der alternative Positionen in einer Hashtabelle ausgewählt werden. Die Wahl einer geeigneten Sondierungsfolge ist dabei entscheidend.

### 5.2.1 Lineares Sondieren

Beim linearen Sondieren wird der Schlüssel bei einer Kollision um ein konstantes Intervall verschoben, bis eine freie Stelle gefunden wird (s. Abbildung 4). Meistens wird die Intervallgröße auf 1 festgelegt. Es ergibt sich die Sondierungsfunktion

$$s(j, k) = j.$$

Um ein Element  $x$  zu lokalisieren, startet die Suche an der durch  $h(x)$  bestimmten Position und setzt sich im Array fort, bis das Element gefunden wird oder ein leerer Slot auftritt.

Lineares Sondieren ist eine einfache Methode des Hashing, bei der jedoch das Problem auftritt, dass sich schnell Gruppen oder Ketten von Kollisionen bilden, was zu längeren Zugriffszeiten in diesen Bereichen führt. Die Häufigkeit von Kollisionen nimmt tendenziell mit der Anzahl der bestehenden Kollisionen zu. Dies wird als primäres Clustering bezeichnet. Sobald sich der Belegungsfaktor  $\alpha$  (Füllgrad der Tabelle) dem Wert 1 nähert, verschlechtert sich die Effizienz des linearen Sondierens drastisch [14].

### 5.2.2 Quadratisches Sondieren

Das quadratische Sondieren ist ein Sondierungsverfahren, das das Problem der primären Häufung des linearen Sondierens löst. Dabei wird mit quadratisch wachsendem Abstand nach einem freien Platz gesucht.

$$s(j, k) = ([j/2])^2(-1)^j$$

Mit dieser Formel erhält man die Sondierungsreihenfolge

$$h(s), h(s) + 1, h(s) - 1, h(s) + 4, h(s) - 4, h(s) + 16, h(s) - 16.$$

Der Nachteil dieses Verfahrens ist die sekundäre Häufung, bei der die Schlüssel mit gleichem Hashwert immer die gleiche Ausweichsequenz durchlaufen [15].

### 5.2.3 Double Hashing

Das Double Hashing ist ein optimiertes Verfahren, das primäre und sekundäre Häufung verhindert. Die Sondierungsreihenfolge wird mittels einer zweiten Hashfunktion  $h'$  generiert. Diese muss zur ersten Hashfunktion unabhängig sein. Die Sondierungsfunktion lautet:

$$s(j, k) = j * h'(k)$$

[14]

Betrachten wir eine Hashtabelle der Größe  $m$  und zwei Hashfunktionen,  $h(k)$  und  $h'(k)$ :

- $h(k) = k \bmod m$
- $h'(k) = 1 + (k \bmod (m - 2))$

In diese Hashtabelle sind bereits die Zahlen 12 und 50 eingefügt worden (s. Abbildung 5). Nun soll der Schlüssel  $k = 47$  eingefügt werden:

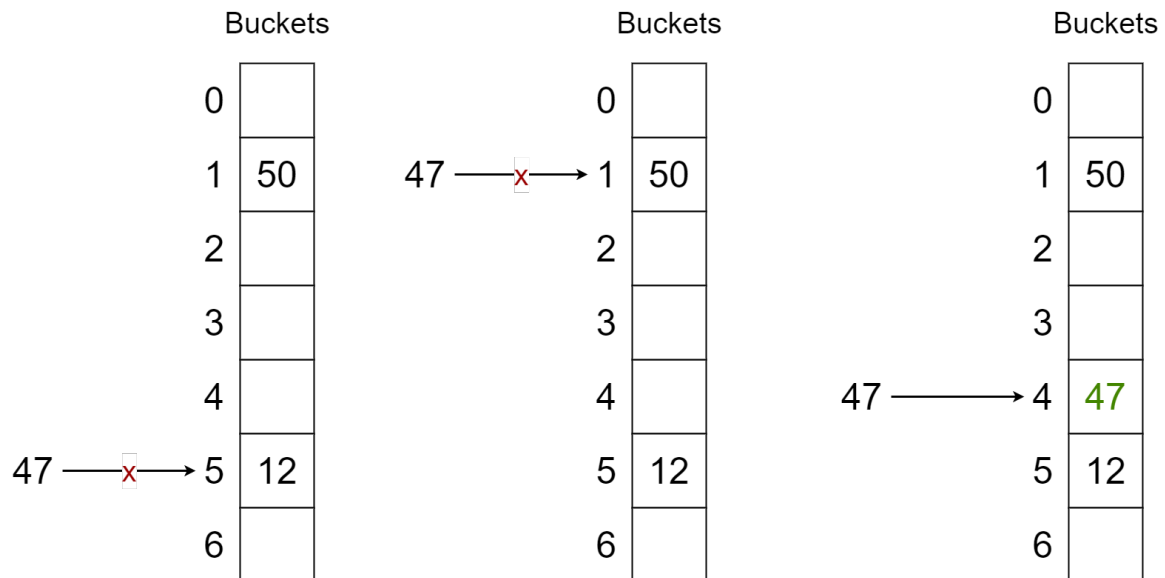


Abbildung 5: Offenes Hashverfahren mit Double Hashing (eigene Darstellung)

- Erster Hashwert:  $h(47) = 47 \bmod 7 = 5$
- Zweiter Hashwert:  $h'(47) = 1 + (47 \bmod 5) = 3$

### Sondierungssequenz:

1. Erste Probe:  $s(0, 47) = (5 + 0 \cdot 3) \bmod 7 = 5$
2. Zweite Probe (bei Kollision):  $s(1, 47) = (5 + 1 \cdot 3) \bmod 7 = 1$
3. Dritte Probe (bei weiterer Kollision):  $s(2, 47) = (5 + 2 \cdot 3) \bmod 7 = 4$

Nachdem in der Sondierungssequenz die Plätze 5 und 1 auf Kollisionen überprüft wurden, konnte die Zahl 47 in den Bucket 4 einsortiert werden.

### 5.2.4 Uniformes Sondieren und zufälliges Sondieren

Uniformes Sondieren verbessert die Verteilung der Schlüssel. Die Sondierungsreihenfolge ist vom Schlüssel unabhängig. Die Haupteigenschaft dieser Methode ist, dass jeder mögliche Index in der Hashtabelle genau einmal pro vollständigem

Durchlauf der Sondierungssequenz aufgerufen wird. Häufungen werden dadurch vermieden [15].

Da dies schwierig zu realisieren ist, spricht man auch vom zufälligem Sondieren. Die Hashtabelle wird durch eine zufällige Sondierungsfolge realisiert. Dort kann ein Index auch zweimal pro Sondierungsfolge auftreten. Es resultiert jedoch in einer ähnlichen Effizienz wie das uniforme Sondieren [14].

Diese Methode erfordert das Speichern der Permutationssequenz, was den Speicherbedarf erhöhen kann.

### 5.2.5 Coalesced Hashing

Das Coalesced Hashing verbindet das Prinzip des offenen Hashings mit dem der Verkettung der Überläufer. In einer Hashtabelle wird jeder Eintrag, der aus einem Schlüssel und dem dazugehörigen Datensatz besteht, zusammen mit einem Zeiger auf dem nächsten Eintrag gespeichert. Dieser Zeiger, der als Hashadresse realisiert ist, verbindet alle Einträge, die aufgrund von Kollisionen nicht direkt in der Hashtabelle gespeichert werden können, in einer Überlaufkette.

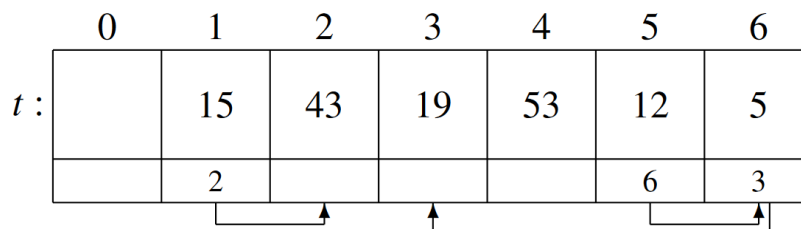


Abbildung 6: Offenes Hashverfahren mit Coalesced Hashing [14]

## 5.3 Dynamisches Hashing

Dynamische Hashverfahren sind Techniken, die es ermöglichen, Hash-Tabellen effizient zu vergrößern oder zu verkleinern. Um die optimierte Suche effizienz von  $O(1)$

zu beizubehalten, muss die Hashtabelle neu geordnet werden, sobald sich der Belegungsfaktor  $\alpha$  einem bestimmten Wert nähert. Wenn die Anzahl der Elemente einen bestimmten Schwellenwert überschreitet (z.B.  $\alpha > 0.75$ ), wird die Größe der Tabelle typischerweise verdoppelt und alle Elemente werden neu eingefügt. Häufig wird die Größe der Tabelle zusätzlich auf die nächste Primzahl erhöht, um die Wahrscheinlichkeit von Kollisionen weiter zu verringern. Man spricht auch vom Rehashing.

Alternativ kann man auch das lineare dynamisches Hashing benutzen. Bei diesem Verfahren wird die Hashtabelle schrittweise um einzelne Blöcke erweitert, wobei jedes Mal nur ein Teil der Elemente neu gehasht wird, um die Last des Rehashing zu verteilen (vgl. [14]).

## 5.4 Implementierung

Im Folgenden wird eine Implementierung der Einfüge-Operation einer Hashtabelle in C# vorgestellt. Als Hash-Funktion wird das Hashing durch Division verwendet. Das gezeigte Beispiel verwendet das lineare Sondieren, um Kollisionen zu behandeln.

```
1 public class HashTable
2 {
3     private KeyValuePair<int, string>?[] table;
4     protected int size;
5     protected int count;
6
7     public HashTableBase(int size)
8     {
9         this.size = size;
10        this.table = new KeyValuePair<int, string>?[size];
11    }
12
```

```
13 private int Hash(int key)
14 {
15     key % size;
16 }
17
18 public void Insert(int key, string value)
19 {
20     CheckLoadFactor();
21
22     int index = Hash(key);
23     int counter = 0;
24
25     while (table[index] != null)
26     {
27         counter++;
28         // If the key already exists, update the value
29         if (table[index]?.Key == key)
30         {
31             table[index] = new KeyValuePair<int, string>(key, value)
32             ;
33             return;
34         }
35         index = (key + ProbingFunction(counter, index)) % size;
36         if (counter == size)
37         {
38             Console.WriteLine("Table is full. Initializing rehashing.");
39             HandleTableFull();
40         }
41     }
42     // Insert the key-value pair at the hashed index
43     count++;
```



```
43     table[index] = new KeyValuePair<int, string>(key, value);  
44 }  
45  
46 public override int ProbingFunction(int j, int k){  
47     return j;  
48 }  
49 ...  
50 }
```

Code-Ausschnitt 3: Implementierung der Einfüge-Operation einer Hashtabelle in C# (eigene Darstellung)

Diese Implementierung verwendet lineares Sondieren, um Kollisionen zu behandeln. Wenn eine Kollision auftritt, wird der nächste freie Platz gesucht. Die *Insert* Methode fügt ein neues Schlüssel-Wert-Paar ein. Zusätzlich kann in der Methode *CheckLoadFactor* das dynamische Hashing realisiert werden. Dort würde die Auslastung der Tabelle überprüft werden und gegebenenfalls eine Rehashing-Operation durchgeführt werden.

## 6 Anwendungsbeispiele

Hashing-Technologien sind in vielen Bereichen der Informatik und Datenverarbeitung von zentraler Bedeutung. In diesem Abschnitt werden verschiedene Anwendungsfälle beschrieben, in denen das Hashing benutzt wird.

### **Password-Hash:**

Passwörter sollten niemals im Klartext gespeichert werden. Stattdessen wird nur der Hashwert eines Passworts gespeichert. Bei der Anmeldung wird das eingegebene Passwort gehasht und mit dem gespeicherten Hashwert verglichen. Diese Methode schützt Passwörter vor Diebstahl, da die Hashfunktionen irreversibel sind.

### **Integritätsprüfung:**

Integrität bedeutet, dass Daten unverändert und korrekt sind. Hashfunktionen sind essenziell, um die Integrität von Daten zu gewährleisten. Sie können schnell sicherstellen, dass bei einer Übertragung keine Daten verändert wurden, egal wie groß die Datei ist. Dazu wird der Hash vor der Übertragung mit dem nach der Übertragung verglichen. Beispiele hierfür sind digitale Signaturen und Integritätsprüfungen für Software-Downloads.

### **Prüfziffer und Fehlererkennung:**

Hashfunktionen können auch bei der Fehlererkennung eine Rolle spielen. Ähnlich zur Integritätsprüfung kann hier mithilfe von Prüfsummen die Korrektheit der Daten sichergestellt werden. So kann beim Validieren einer Eingabe überprüft werden, ob die Eingabe einen Tippfehler, Zahlendreher oder eine Ziffer zu viel enthält. Ein solcher Mechanismus kommt z. B. bei einer Überweisung per IBAN oder bei einer ISBN-Nummer zum Einsatz [16].

ISBN 3-446-19313-[?] (Fermats letzter Satz):

$$3 * 1 + 4 * 2 + 4 * 3 + 6 * 4 + 1 * 5 + 9 * 6 + 3 * 7 + 1 * 8 + 3 * 9 = 162; 162 \mod 11 = 8$$

Die ersten 10 Ziffern identifizieren das Buch. Die letzte angehängte Nummer ist eine

Prüfziffer. Diese wird über die Quersumme der anderen Ziffern und mit modulo 11 berechnet. Bei dem Beispiel des Buches "Fermats letzter Satz" wird hier die Prüfziffer 8 erzeugt. Ein weiteres Beispiel sind CRC-Checksummen (Cyclic Redundancy Check).

### **Deduplizierung:**

In der Datenverwaltung wird die Deduplizierung verwendet, um doppelte Kopien von Daten zu identifizieren und zu entfernen. Hashfunktionen können genutzt werden, um effizient zu prüfen, ob eine Datei oder ein Datensatz bereits existiert. Indem man den Hashwert einer Datei speichert, kann man schnell überprüfen, ob eine neu hinzugefügte Datei den gleichen Hashwert aufweist und somit ein Duplikat ist. Speziell in Speichernetzwerken oder Cloud-Speichern kann dies von Vorteil sein.

### **Index-Datenstrukturen:**

Hashfunktionen sind die Grundlage für viele Datenstrukturen wie Hash-Tabellen, Hash-Bäumen und Bloom-Filter. In Hash-Tabellen werden Schlüssel mithilfe von Hashfunktionen in Indizes umgewandelt, um schnellen Zugriff auf Daten zu ermöglichen. Hashtabellen sind besonders wichtig in Datenbanken, da ein Hashindex unter optimalen Bedingungen zu idealen Zugriffszeiten führen kann. Dies ist für die Effizienz und Leistung von Datenbanksystemen von entscheidender Bedeutung. Bloom-Filter verwenden Hashfunktionen, um effizient zu überprüfen, ob ein Element in einer Menge enthalten ist, ohne die Elemente selbst zu speichern.

### **Blockchain Technologien:**

In Kryptowährungen wie Bitcoin werden Hashfunktionen verwendet, um die Blockchain, eine Kette von Transaktionsblöcken, zu sichern. Jeder Block enthält einen Hash des vorherigen Blocks, was eine Manipulation der Blockchain äußerst schwierig macht. Hashfunktionen werden auch im Mining-Prozess verwendet, um neue Blöcke zu erzeugen.

## 7 Fazit

Hashing ist eine wichtige Technik in der Informatik, die in vielen Anwendungsfällen im Einsatz ist. Sie ermöglicht das effiziente Speichern, Abrufen und Verwalten von Daten. In dieser Ausarbeitung wurden die Grundlagen des Hashing und deren Funktionsweise erläutert. Besonderes Augenmerk lag auf der Behandlung von Kollisionen, die eine unvermeidliche Herausforderung beim Hashing darstellen. Die verschiedenen Methoden zur Kollisionsbehandlung, wie Verkettung und offene Hashverfahren, wurden ausführlich diskutiert. Zusätzlich wurden moderne Hash-Algorithmen wie MD5, SHA-1, SHA-2 und SHA-3 vorgestellt und ihre Anwendung in verschiedenen Bereichen wie Passwortsicherheit, Datenintegrität und Blockchain-Technologien vorgestellt.

Die verschiedenen Methoden zur Kollisionsbehandlung bieten jeweils spezifische Vorteile und Herausforderungen. Gerade bei Hashtabellen ist es eine große Herausforderung, trotz der möglichen Kollisionsbehandlungsmethoden eine gute und konstante Einfüge-Geschwindigkeit zu erreichen.

In der modernen Softwareentwicklung sind durch die Abstraktionsschichten der verwendeten Programmiersprachen und -bibliotheken oft die Implementierung solcher Kollisionsbehandlungsmethoden verborgen. Die meisten Hochsprachen wie Python, Java und C# bieten eingebaute Datenstrukturen wie Hash-Tabellen oder Dictionaries, die automatisch Kollisionsbehandlung implementieren, sodass sich Entwickler selten direkt um Kollisionsbehandlung kümmern müssen.

Zusammenfassend bleibt Hashing eine Schlüsseltechnologie in der Informatik, die entscheidend für die Entwicklung sicherer und effizienter IT-Systeme ist.

## 8 Quellen

### Literatur

- [1] Wikipedia, *Hashfunktion*. Adresse: <https://de.wikipedia.org/wiki/Hashfunktion> (besucht am 05.05.2024).
- [2] Wikipedia, *Hashtabelle*. Adresse: <https://de.wikipedia.org/wiki/Hashtabelle> (besucht am 10.05.2024).
- [3] Wikipedia, *Kollisionsresistenz*. Adresse: <https://de.wikipedia.org/wiki/Kollisionsresistenz> (besucht am 06.05.2024).
- [4] M. Stevens, E. Bursztein, P. Karpman, A. Albertini und Y. Markov, "The First Collision for Full SHA-1," in *Advances in Cryptology – CRYPTO 2017*, J. Katz und H. Shacham, Hrsg., Cham: Springer International Publishing, 2017, S. 570–596, ISBN: 978-3-319-63688-7.
- [5] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998, S. 516–520, ISBN: 0201896850.
- [6] Wikipedia, *Kryptographische Hashfunktion*. Adresse: [https://de.wikipedia.org/wiki/Kryptographische\\_Hashfunktion](https://de.wikipedia.org/wiki/Kryptographische_Hashfunktion) (besucht am 24.04.2024).
- [7] Wikipedia, *Message-Digest Algorithm 5*. Adresse: [https://de.wikipedia.org/wiki/Message-Digest\\_Algorithm\\_5](https://de.wikipedia.org/wiki/Message-Digest_Algorithm_5) (besucht am 08.05.2024).
- [8] Wikipedia, *SHA-1*. Adresse: <https://en.wikipedia.org/wiki/SHA-1> (besucht am 08.05.2024).
- [9] G. Leurent und T. Peyrin, "From Collisions to Chosen-Prefix Collisions Application to Full SHA-1," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai und V. Rijmen, Hrsg., Cham: Springer International Publishing, 2019, S. 527–555, ISBN: 978-3-030-17659-4.

- [10] Wikipedia, *SHA-2*. Adresse: <https://de.wikipedia.org/wiki/SHA-2> (besucht am 08.05.2024).
- [11] J. Guo, G. Liao, G. Liu, M. Liu, K. Qiao und L. Song, "Practical Collision Attacks against Round-Reduced SHA-3," *Journal of Cryptology*, Jg. 33, Nr. 1, S. 228–270, Jan. 2020, ISSN: 1432-1378. DOI: 10.1007/s00145-019-09313-3. Adresse: <https://doi.org/10.1007/s00145-019-09313-3>.
- [12] Wikipedia, *SHA-3*. Adresse: <https://de.wikipedia.org/wiki/SHA-3> (besucht am 05.05.2024).
- [13] A. D. Yusuf, S. Abdullahi, M. M. Boukar und S. I. Yusuf, "Collision Resolution Techniques in Hash Table: A Review," *International Journal of Advanced Computer Science and Applications*, Jg. 12, Nr. 9, 2021. DOI: 10.14569/IJACSA.2021.0120984. Adresse: <http://dx.doi.org/10.14569/IJACSA.2021.0120984>.
- [14] T. Ottmann und P. Widmayer, *Algorithmen und Datenstrukturen*. Springer Berlin Heidelberg, 2017, ISBN: 9783662556504. Adresse: <https://books.google.de/books?id=grEvDwAAQBAJ>.
- [15] M. Teschner, *Algorithmen und Datenstrukturen, Hashverfahren*, 2012. Adresse: [https://cg.informatik.uni-freiburg.de/course\\_notes/info2\\_11\\_hashverfahren.pdf](https://cg.informatik.uni-freiburg.de/course_notes/info2_11_hashverfahren.pdf) (besucht am 12.05.2024).
- [16] Lehrer Fortbildung Baden-Württemberg, *Hashing*, 2016. Adresse: [https://lehrerfortbildung-bw.de/u\\_matnatech/informatik/gym/bp2016/fb2/01\\_duc/1\\_hintergrund/1\\_infos/05\\_hashing/](https://lehrerfortbildung-bw.de/u_matnatech/informatik/gym/bp2016/fb2/01_duc/1_hintergrund/1_infos/05_hashing/) (besucht am 24.04.2024).

## 9 Anhang

### Tabellenverzeichnis

1	Beispiel für die Bildung der Quersumme als Hash-Wert (eigene Darstellung) . . . . .	3
2	Beispiel für MD5 Hash-Werte bei leicht unterschiedlichen Eingaben (eigene Darstellung) . . . . .	4

### Abbildungsverzeichnis

1	Beispiel für eine nicht injektive Funktion (eigene Darstellung) . . . .	2
2	Bitlänge der SHA3 Varianten [12] . . . . .	12
3	Kollisionsbehandlung durch Verkettung [2] . . . . .	14
4	Offenes Hashverfahren mit linearem Sondieren (eigene Darstellung)	15
5	Offenes Hashverfahren mit Double Hashing (eigene Darstellung) . .	18
6	Offenes Hashverfahren mit Coalesced Hashing [14] . . . . .	19

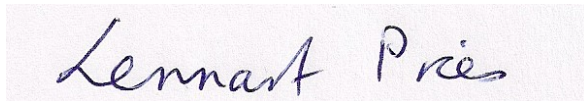
### Codeverzeichnis

1	Implementierung des Hashings durch Division in C# (eigene Darstellung) . . . . .	8
2	Implementierung des Hashings durch Multiplikation in C# (eigene Darstellung) . . . . .	9
3	Implementierung der Einfüge-Operation einer Hashtabelle in C# (eigene Darstellung) . . . . .	20

## Erklärung zur Arbeit

Ich erkläre, dass ich alle Quellen, die ich zur Erstellung dieser Arbeit verwendet habe, im Quellenverzeichnis aufgeführt habe und dass ich alle Stellen, an denen Informationen (Texte, Bilder, Tabelle, Quellcode) aus diesen Quellen in meine Arbeit eingeflossen sind, als Zitate mit Quellenangabe kenntlich gemacht habe. Mir ist bewusst, dass ein Verstoß gegen diese Regeln zum Ausschluss aus dem Seminar führt.

Lennart Pries

A handwritten signature in blue ink on a light-colored rectangular background. The signature reads "Lennart Pries" in a cursive script.