Technische Universität Wien

Vienna University of Technology

DIPLOMARBEIT

Analyse von Hash-Algorithmen

ausgeführt am Institut für

Diskrete Mathematik und Geometrie

der Technischen Universität Wien

unter Anleitung von a.o. Univ-Prof. Dipl.- Ing. Dr. techn. Michael Drmota

durch

Reinhard Kutzelnigg

Badsiedlung, 385

8250 Vorau

Datum

Unterschrift

Inhaltsverzeichnis

Vorwort **ii**

1 Einleitung **1**
1.1 Was ist ein Hashverfahren? . . . 1
1.2 Problemstellung . . . 2
1.3 Anwendungsbeispiele . . . 2
1.4 Hashfunktionen in der Kryptographie . . . 3
1.5 Verwendete Begriffe . . . 3
1.6 Kollisionen . . . 3
1.6.1 Strategien zur Behandlung von Kollisionen . . . 4
1.6.2 Vermeidung von Kollisionen . . . 4
1.7 Häufig verwendete Hashfunktionen . . . 5
1.7.1 Darstellung von Schlüsseln als natürliche Zahlen . . . 5
1.7.2 Divisionsmethode . . . 5
1.7.3 Multiplikationsmethode . . . 5
1.7.4 Weitere Verfahren . . . 5
1.7.5 Ausnützen von Nichtzufälligkeiten . . . 6
1.8 Ein Modell für die Eingangsdaten . . . 6

2 Mathematische Grundlagen **8**
2.1 Kombinatorik . . . 8
2.1.1 Erzeugende Funktionen . . . 8
2.1.2 Asymptotische Entwicklungen . . . 9
2.1.3 Fibonaccizahlen . . . 10
2.1.4 Identitäten . . . 11
2.1.5 Graphen . . . 12
2.2 Wahrscheinlichkeitstheorie . . . 13
2.2.1 Wahrscheinlichkeitserzeugende Funktionen . . . 13
2.2.2 Das Geburtstagsparadoxon . . . 13
2.2.3 Airy Verteilung . . . 14
2.3 Zufallszahlen . . . 14
2.3.1 Generatoren für Pseudozufallszahlen . . . 15
2.4 Zahlentheorie . . . 16
2.4.1 Der Chinesische Restsatz . . . 16
2.4.2 Darstellung einer natürlichen Zahl als Summe von Quadraten . . . 16
2.4.3 Kettenbrüche . . . 16

3 Hashing mit Verkettung **22**
3.1 Gewöhnliches Hashing mit Verkettung 22
3.2 Verbesserungen des Standardverfahrens 26
3.3 Coalesced Hashing 28
3.4 Verfahren mit mehreren Tabellen 35

4 Hashing mit offener Adressierung **40**
4.1 Lineares Sondieren 41
4.2 Quadratisches Sondieren 48
4.3 Uniform Hashing 49
4.4 Zufälliges Sondieren 51
4.5 Double Hashing 52
4.6 Brents Algorithmus 53
4.7 Binärbaum Sondieren 55
4.8 Robin Hood Hashing 55

5 Universelles Hashing **57**
5.1 Beispiele für universelle Klassen 57
5.2 Eigenschaften von universellem Hashing 60

6 Perfektes Hashing **62**
6.1 Elementare Ansätze 62
6.1.1 Trial and Error 62
6.1.2 Ein Verfahren mit dem Chinesischen Restsatz 63
6.1.3 Ein Kompressionsverfahren 64
6.2 Das FKS Verfahren 65
6.3 Cuckoo Hashing 68

7 Experimente am Computer **72**
7.1 Hard- und Software 72
7.2 Untersuchungen mit zufälligen Daten 73
7.2.1 Hashverfahren mit Verkettung 73
7.2.2 Offene Hashverfahren 78
7.2.3 Verfahren zur Begrenzung der maximalen Suchlänge 83
7.2.4 Vergleich der Verfahren 84
7.3 Erstellung eines Wörterbuchs 84

A Bemerkung zu „Cuckoo Hashing: Further Analysis” **85**

B Symboltafel **86**

C Software **87**

Index **89**

Literaturverzeichnis **91**

Vorwort

Die Erfindung der ersten Computer hat einen Prozess ausgelöst, der nicht nur unser tägliches Leben verändert hat, in dem praktisch kein Stein mehr auf dem anderem geblieben ist, auch der Einfluss auf die Mathematik war gewaltig. Vieles, was einstmals nur von theoretischem Interesse war, ist plötzlich zu einem für die Praxis relevanten Forschungsgebiet geworden. So ist zum Beispiel die Zahlentheorie auf Grund der Verwendung in der Kryptographie zu einem unverzichtbaren Baustein moderner Kommunikation geworden.
Nicht nur die Rechenleistung der Computer nimmt ständig zu, sondern auch die Anzahl der Daten, sei dies auf der eigenen Festplatte oder im Internet. Um in dieser Unmenge von Information nicht vollständig die Orientierung zu verlieren, bedarf es immer besserer Algorithmen, die bei der Suche und der Organisation helfen. Verzeichnisse bieten eine Möglichkeit Daten gut organisiert zu speichern. Diese Arbeit beschäftigt sich mit verschiedenen Verfahren, die es ermöglichen solche Verzeichnisse anzulegen.

Das erste Kapitel präzisiert die Aufgabenstellung, erklärt alle notwendigen Begriffe und liefert das gedankliche Fundament für die Analyse der Algorithmen. Daran anschließend werden für die Arbeit wesentliche mathematische Ergebnisse und Hilfsmittel präsentiert.
Die nächsten Kapitel befassen sich schließlich jeweils mit einer zusammengehörenden Gruppe von ähnlich aufgebauten Algorithmen. Insbesondere werden alle „klassischen" Verfahren eingehend untersucht. Es werden aber auch einige neue Variationen behandelt. Bei der Unmenge der Publikationen die es inzwischen zu diesem Thema gibt, ist es naturgemäß nicht möglich alles zu behandeln. So füllt eine aktuelle Zusammenfassung des im Kapitel 6 behandelten perfekten Hashings allein schon über 140 Seiten. Ich habe mich bemüht neben den bekannten und bewährten Verfahren auch einige interessante neuere Ansätze einfließen zu lassen.
Den Abschluss bildet ein Kapitel mit von mir ermittelten experimentellen Daten.

Ich möchte mich auf diesem Weg auch bei Herrn Prof. Drmota für den Vorschlag dieses interessanten Themas und seine Betreuung bedanken. Weiters gilt mein Dank Maria und meinem Vater für das Korrekturlesen, sowie meinen Eltern dafür, dass sie mir dieses Studium ermöglicht haben.
Wesentlich für die Entstehung dieser Arbeit ist aber auch die verwendete Software. So möchte schließlich noch allen danken, die durch ihre Arbeit an der Entstehung von Linux und LaTeX mitgewirkt haben, sowie dem Zentralen Informatikdienst und den Firmen, die Software wie MAPLE und Visual Studio als günstige Studentenversion anbieten.

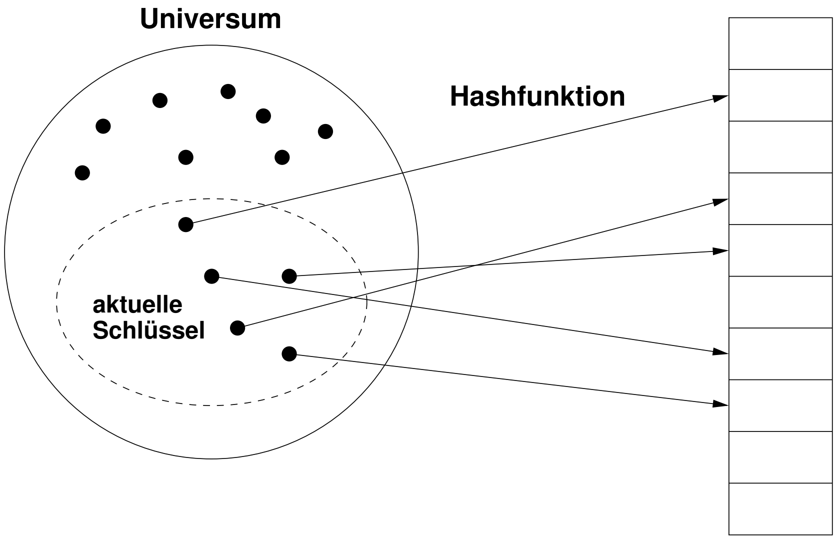
Kapitel 1

Einleitung

__hash__: zerhacken, zerkleinern, faschieren, Durcheinander
aus Langenscheidts Englischwörterbuch

1.1 Was ist ein Hashverfahren?

Viele Anwendungen in der Informatik benötigen Verzeichnisse, in die Daten schnell eingefügt, gefunden und auch wieder gelöscht werden können. Eine Möglichkeit, so ein Verzeichnis zu erzeugen, besteht darin, eine Liste anzulegen, die mindestens so viele Plätze enthält, wie Einträge aufgenommen werden sollen. Nun benötigt man eine Vorschrift, die die einzelnen Elemente auf die Plätze der Liste zuweist. Dies übernimmt die so genannte Hashfunktion:



Idealerweise ist die verwendete Funktion injektiv, allerdings ist es für praktische Zwecke im Normalfall nicht mit vertretbarem Aufwand möglich, eine solche Funktion zu finden. Deshalb wird es im Allgemeinen unterschiedliche Einträge geben, die den selben Platz in der Tabelle beanspruchen. Es gibt verschiedene Ansätze, wie dieses Problem gelöst werden kann. Darauf wird jedoch erst später eingegangen. In diesem Kapitel werden grundlegende Ideen und Begriffe bezüglich Hashing erläutert.

1.2 Problemstellung

Gegeben sei eine (endliche) Menge U, die als Universum bezeichnet wird. Jedes der Elemente von U besteht dabei aus dem Schlüssel S und der Information I, wobei jedes Element durch den Schlüssel eindeutig bestimmt wird. Dabei sind S und I Elemente aus einer jeweils vorgegebenen Menge. Oft wird auch der Schlüssel mit der Information gleichgesetzt.

Gesucht ist ein Verfahren, mit dem beliebige Elemente aus U in eine listenförmige Datenstruktur eingetragen werden können, sodass sowohl der Platzbedarf für die Liste, als auch der Aufwand für Einfüge-, Zugriffs- und eventuell auch Löschoperationen möglichst gering ist.

Definition 1.1 ([9])
Unter Hashing versteht man eine Methode, bei der Elemente des Universums U mit Hilfe einer auf den Schlüssel S angewendeten Funktion, der so genannten Hashfunktion, in eine listenförmige Datenstruktur eingetragen werden.

Hashing versucht also sowohl Laufzeit, als auch Speicherplatzbedarf gering zu halten, also einen „Kompromiss“ zu suchen. Spielt die Laufzeit keine Rolle, kann man die Daten einfach in einer Liste eintragen und jeden Suchvorgang sequentiell durchführen, wobei der Speicherplatzbedarf auf ein Minimum sinkt. Ist umgekehrt uneingeschränkt Speicherplatz vorhanden, so kann der Schlüssel direkt als Speicherplatzadresse verwendet werden. Eine Alternative zur Verwendung des Hashverfahrens ist die Benutzung einer sortierten Liste, in der die Einträge mit Binärsuche gesucht werden. Problematisch bei diesem Verfahren ist jedoch das Einfügen von Elementen, da stets davon auszugehen ist, dass ein großer Teil der Elemente verschoben werden muss.

1.3 Anwendungsbeispiele

Um die Bedeutung des Algorithmus zu unterstreichen, werden hier konkrete Beispiele angeführt, die zeigen, wie vielfältig Hashing angewendet werden kann.

Beispiel 1 ([9, 32]) *Der Compiler einer Programmiersprache verwaltet eine Symboltafel, in welche die in einem Programm vorkommenden Bezeichnungen (Identifier) eingetragen werden. Beim ersten Auftreten im Programm wird eine solche zusammen mit Informationen wie Typ, Lebensdauer oder zugewiesener Speicherplatz in die Symboltafel eingetragen. Bei wiederholtem Auftreten der Bezeichnung muss diese in der Symboltafel wieder gefunden werden. Identifier sind meist Zeichenketten über dem Alphabet $\{A, B, C, \ldots\}$. Im Allgemeinen ist die Wahrscheinlichkeit, dass eine bestimmte Folge von Buchstaben auftritt, nicht für alle möglichen Kombinationen gleich groß, da Zeichenketten wie $I1, I2, \ldots$ öfter verwendet werden als z.B. XYZ.*

Beispiel 2 *Zuordnungen von Namen und Zahlen wie z.B. bei der Sozialversicherungsnummer, Telefonnummer, Matrikelnummer, der Kontonummer bei einer Bank oder die Zuordnung von IP- Adressen zu Domainnamen im Internet mit dem so genannten Domain Name System (DNS) könnten mit Hilfe einer Hashtabelle realisiert werden.*

Beispiel 3 ([28]) *Im Kernel des Betriebssystems LINUX werden an mehreren Stellen (page cache, buffer cache, directory entry cache, inode cache) Hashtabellen verwendet. Die Leistungsfähigkeit des gesamten Betriebssystems hängt stark von diesen ab.*

Beispiel 4 *Damit ein Computer Daten auf einer Festplatte dauerhaft speichern kann, muss diese über ein so genanntes Dateisystem verfügen. Dabei handelt es sich um „Metadaten”, die die Struktur der Festplattendaten beschreiben. Das System ReiserFS, das eines der gebräuchlichsten*

Dateisysteme für das Betriebssystem LINUX ist, verwendet unter anderem einen Hashalgorithmus, um zu einem Dateinamen den Ort der Festplatte (die so genannte Blocknummer) zu finden, an dem die Datei physikalisch gespeichert ist.

Beispiel 5 *Auch ein Wörterbuch könnte mit Hilfe einer Hashtabelle aufgebaut werden. Die Menge der Schlüssel besteht dann aus allen zulässigen Wörtern der Sprache. Die zu einem Schlüssel gehörende Information ist die Beschreibung oder Übersetzung des Wortes.*

1.4 Hashfunktionen in der Kryptographie

Auch in der modernen Kryptographie kommen so genannte Hashfunktionen zum Einsatz. Manchmal werden diese auch als Einweg- Hashfunktionen bezeichnet. Darunter wird eine effizient berechenbare Funktion verstanden, die (binäre) Zeichenketten willkürlicher Länge auf (binäre) Zeichenketten vorgegebener Länge abbildet (siehe [33]). Von kryptographischem Interesse sind dabei Funktionen, für die es mit vertretbarem Aufwand unmöglich ist, zwei „sinnvolle“ Eingaben zu finden, für die die Hashfunktion den selben Wert hat. Diese Hashfunktionen werden vor allem für digitale Signaturen und zur Überprüfung der Datenintegrität verwendet. Auf Grund der fundamentalen Unterschiede werden Hashfunktionen für kryptographische Zwecke hier **nicht** weiter behandelt.

1.5 Verwendete Begriffe

Definition 1.2
Die Größe der Hashtabelle wird mit m bezeichnet, wobei die einzelnen Speicherplätze mit $0, 1, 2, \ldots, m-1$ adressiert werden.
Die Anzahl der (verschiedenen) einzufügenden Elemente bezeichnet man mit n.
Der Quotient $\alpha := \frac{n}{m}$ heißt Belegungsfaktor.

Es wird also davon ausgegangen, dass n Elemente aus dem Universum U in beliebiger Reihenfolge in eine Tabelle der Größe m, die meist Hashtabelle genannt wird, eingefügt werden. Die Umwandlung des Schlüssels S in die Tabellenadresse erfolgt dabei durch die Hashfunktion.

Im Allgemeinen soll ein Element nicht mehrfach in die Hashtabelle eingefügt werden, da dadurch unnötig Platz verbraucht wird.

Je nach Algorithmus wird deshalb vor dem Einfügen überprüft, ob das Element bereits vorhanden ist. In diesem Fall wird mit n die Anzahl der verschiedenen einzutragenden Elemente bezeichnet.

Definition 1.3 (Kollision [9, 25, 32, 40])
Sei h eine Hashfunktion. Falls zwei unterschiedliche, in die Tabelle einzutragende Schlüssel S_1 und S_2 den selben Hash- Wert besitzen, d.h. wenn $h(S_1) = h(S_2)$, spricht man von einer Kollision.

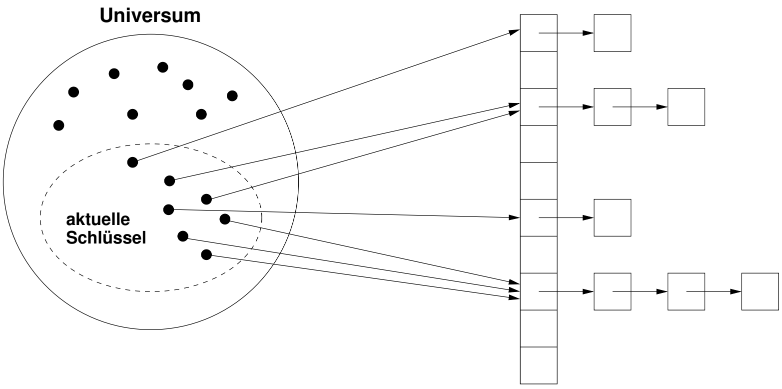
1.6 Kollisionen

Im normalen Anwendungsfall werden stets Kollisionen auftreten. Ein Beispiel hierfür liefert das bei den mathematischen Grundlagen erwähnte Geburtstagsparadoxon: Bereits bei 25 zufällig ausgewählten Personen ist die Wahrscheinlichkeit, dass es mindestens zwei unter ihnen gibt, die am selben Tag im Jahr Geburtstag haben größer als 50 Prozent. Nur beim später behandelten „perfekten Hashing“, wo eigens eine injektive Hashfunktion für eine vorgegebenene n- elementige Menge von U konstruiert wird, muss keine Rücksicht auf Kollisionen genommen werden.

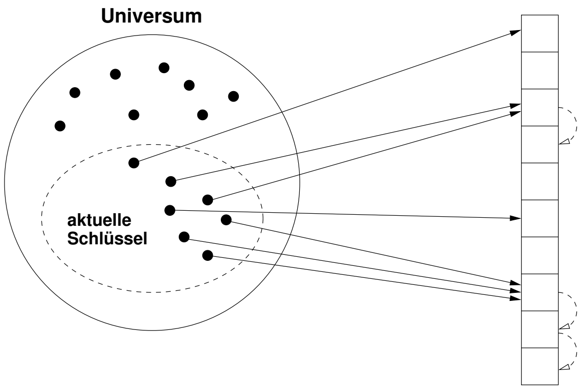
1.6.1 Strategien zur Behandlung von Kollisionen

Es gibt prinzipiell zwei unterschiedliche Ansätze:

- Hashing mit Verkettung
  Die Idee dazu stammt aus dem Jahr 1953 von H. P. Luhn (siehe [9]). Es handelt sich dabei um die einfachste Methode zur Auflösung von Kollisionen. Für jede Tabellenadresse wird eine verkettete Liste erzeugt, die all jene Datensätze aufnimmt, deren Schlüssel auf diese eine Tabellenadresse abbilden.



- Hashing mit offener Adressierung
  Etwa zur selben Zeit als Hashing mit Verkettung entstand, entwickelte G. M. Amdahl (vergleiche [9]) Hashing mit offener Adressierung. Bei diesem Ansatz werden alle Datensätze direkt in der Tabelle gespeichert. Ist der Platz, an dem ein Element eingefügt werden soll, bereits belegt, wird nach einem festgelegten Verfahren ein alternativer Platz bestimmt. Z.B. kann das der freie Platz mit der nächstgrößeren Adresse in der Hashtabelle sein.



1.6.2 Vermeidung von Kollisionen

Damit so wenig Kollisionen wie möglich auftreten, ist eine gute Wahl der Hashfunktion sehr wichtig. Würde man beispielsweise die ersten drei Stellen einer Matrikelnummer als Wert der Hashfunktion verwenden, so wäre das eine sehr schlechte Wahl, da die führenden beiden Stellen das Jahr der ersten Zulassung angeben und bei vielen Studenten gleich sein werden! Ob eine Funktion eine gute oder schlechte Hashfunktion ist, hängt also auch sehr stark von der Wahl der einzutragenden Daten ab.

1.7 Häufig verwendete Hashfunktionen

1.7.1 Darstellung von Schlüsseln als natürliche Zahlen

Nach der allgemeinen Definition müssen Schlüssel keine Zahlen sein, sondern es können z.B. beliebige Zeichenketten als Schlüssel aufgefasst werden. Für die folgenden Funktionen wird jedoch vorausgesetzt, dass der Schlüssel eine natürliche Zahl ist. Dies ist aber keine Beschränkung, da auf einfache Weise jedem Schlüssel eine eindeutige natürliche Zahl zugeordnet werden kann. (Z.B. durch den ASCII Code.)

Beispiel 6 *Jedem Großbuchstaben wird durch seine Reihenfolge im Alphabet eine Zahl zugeordnet. Weiters soll ein „Leerzeichen“ mit* 0 *codiert werden. Damit kann z.B.: „HALLO” der folgendem Wert zugeordnet werden:*
$8*27^4+1*27^3+12*27^2+12*27^1+15*27^0 = (((8*27+1)*27+12)*27)+12)*27+15 = 4280298$

1.7.2 Divisionsmethode

Bei dieser Methode [9, 25, 40] ist der Wert der Hashfunktion h für einen Schlüssel S gleich dem Rest der ganzzahligen Division von S durch m, d.h.

$$h(S) := S \mod m.$$

Bestimmte Werte für m sollten bei der Divisionsmethode aber meist vermieden werden. So ist z.B. eine Zweierpotenz meist eine schlechte Wahl für m, da $h(S)$ dann einfach aus den letzten Binärstellen von S besteht. Im Allgemeinen sollte stets darauf geachtet werden, dass $h(S)$ von allen Stellen des Schlüssels S abhängt. Eine gute Wahl für m sind z.B. nicht zu nahe an Zweierpotenzen liegende Primzahlen.

1.7.3 Multiplikationsmethode

Die Multiplikationsmethode [9, 25, 40] besteht aus zwei Schritten. Zuerst wird der Schlüssel mit einer Zahl A zwischen Null und Eins multipliziert und der gebrochene Anteil des Produkts bestimmt. Im zweiten Schritt wird dieser Wert mit m multipliziert und dieses Ergebnis abgerundet, d.h.:

$$h(S) := \lfloor m\{SA\}\rfloor.$$

Ein Vorteil der Multiplikationsmethode ist, dass der Wert für m unproblematisch ist. Knuth schlägt in [25] als Wert für A eine Approximation des „goldenen Verhältnisses“ $\phi^{-1} = \frac{\sqrt{5}-1}{2}$ vor. Durch diesen Wert wird eine besonders gleichmäßige Aufteilung erreicht. Der Beweis für diese Eigenschaft wird im Kapitel 2.4.3 nachgetragen.

1.7.4 Weitere Verfahren

Neben den bereits erwähnten Verfahren, die am häufigsten eingesetzt werden, gibt es unter anderem noch folgende Methoden:

Stellenanaylse

Für diese Methode wird vorausgesetzt, dass die Schlüssel Zeichenketten einer vorgegebenen Länge sind. Auch die einzelnen Plätze in der Hashtabelle werden mit Zeichenketten über dem selben Alphabet, das für die Schlüssel verwendet wird, adressiert, jedoch mit geringerer Länge. Man untersucht nun in der Menge aller einzufügenden Schlüssel an jeder Stelle die Verteilung der einzelnen Zeichen. Anschließend werden die Stellen mit den ungleichförmigsten Verteilungen solange gestrichen, bis die verbleibende Zeichenkette die Länge einer Tabellenadresse hat. Der Hashwert eines jeden Schlüssels wird dann durch Streichung von genau diesen Stellen ermittelt. Dieses Verfahren setzt voraus, dass die Menge der einzufügenden Schlüssel zum Zeitpunkt der Wahl der Hashfunktion zumindest größtenteils bekannt ist. Dadurch können ungünstige Verteilungen in den einzufügenden Schlüsseln zumindest teilweise umgangen werden. Experimentelle Resultate aus [29] zeigen jedoch, dass die Qualität der Stellenanalyse großen Schwankungen unterworfen ist.

Polynomielles Hashing

Polynomielles Hashing [25] verwendet Methoden der algebraischen Codierungstheorie, die hier kurz erläutert werden. Für einen fest gewählten endlichen Körper K ist ein (l, k)-Linearcode ein k- dimensionaler Unterraum von K^l. Durch ihn ist auf natürliche Weise eine lineare Abbildung von K^k nach K^l definiert, die jedem Klartextwort das entsprechende Codewort zuordnet. Wegen des Isomorphismus $v_1, \ldots, v_l \mapsto \sum_{i=1}^{l} v_i x^{l-i}$ kann jedes Wort $v_1, \ldots, v_l$ mit einem Polynom vom Grad kleiner l identifiziert werden. Für einen zyklischen Linearcode kann die Codierung durch Multiplikation modulo $x^l - 1$ mit dem so genannten Generatorpolynom $g(x)$ erfolgen, die Decodierung erfolgt analog mit Division durch $g(x)$. Für Wörter, die nicht im Code liegen, tritt bei dieser Division ein von Null verschiedener Rest auf. Wenn der verwendete Code Minimalgewicht w hat, d.h. die Hamming Distanz von zwei unterschiedlichen Codewörtern stets größer oder gleich w ist, dann ist für zwei Wörter aus K^l, die sich um weniger als w Stellen unterscheiden, sicher gestellt, dass sich verschiedene Reste bei der Division durch $g(x)$ ergeben. Dieser Rest kann nun als Hashwert verwendet werden, wobei die zugehörige Hashfunktion K^l auf K^{l-k} abbildet. Details über zyklische Linearcodes, insbesondere zur Wahl des Generatorpolynomes, finden sich z.B. in [4].

Beispiel 7 *Das Generatorpolynom $x^3 + x^2 + 1$ erzeugt über $\mathbb{Z}_2$ einen zyklischen (7,4)- Linearcode mit Minimalgewicht 3. Für die Wörter mit Gewicht 1 ergeben sich folgende Reste:*

Wort:	1	x	x^2	x^3	x^4	x^5	x^6
Rest:	1	x	x^2	x^2+1	x^2+x+1	$x+1$	x^2+x

Der Schlüssel $S = 0101011$ entspricht dem Polynom $S(x) = x^5 + x^3 + x + 1$. Als Hashwert ergibt sich $x^2 + 1$. Beispielsweise bildet auch das Polynom $x^5 + x^2 + x$ auf den selben Hashwert ab. Dieses unterscheidet sich aber bereits um 3 Stellen vom ursprünglichen Schlüssel.

1.7.5 Ausnützen von Nichtzufälligkeiten

In praktischen Anwendungen kommen oft arithmetische Folgen von Schlüsseln der Bauweise S, $S + d$, $S + 2d, \ldots$ wie z.B. $TYP1$, $TYP2$, $TYP3, \ldots$ vor [25, 32]. Sowohl die Multiplikations- als auch die Divisonsmethode können diese Tatsache ausnützen. Die Multiplikationsmethode konvertiert eine arithmetische Folge von Schlüsseln nämlich in eine annähernd arithmetische Folge von Hashwerten, bei der Divisionmethode gilt dies sogar exakt. Dadurch ist im Vergleich zu rein zufälligen Eingaben sogar mit einer Verringerung der Anzahl von Kollisionen zu rechnen.

1.8 Ein Modell für die Eingangsdaten

Der beste bzw. der schlechteste mögliche Fall, der bei einem Hashverfahren auftreten kann, ist unabhängig davon, wie Kollisionen behandelt werden, leicht zu erkennen. Ein solcher Extremfall

entsteht, wenn alle Schlüssel von der Hashfunktion auf verschiedene Plätze bzw. die selbe Tabellenposition abgebildet werden. Für die praktische Anwendung ist jedoch der „durchschnittliche" zu erwartende Aufwand interessant. Um diesen Durchschnitt bilden zu können, muss zunächst festgelegt werden, welche Eingangsdaten überhaupt möglich sind.
Im weiteren wird, falls nicht ausdrücklich etwas anderes erwähnt wird, stets folgendes [25, 32] für die Eingangsdaten vorausgesetzt:

Jede der m^n möglichen Folgen von Hashwerten $a_1, a_2, a_3, \ldots, a_n$ mit $0 \leq a_i < m$ für alle i von 1 bis n ist gleich wahrscheinlich.

Damit ist also $\mathbb{P}(a_i = k) = \frac{1}{m}$ für alle i von 1 bis n und alle k von 1 bis m.
Welche Bedingungen müssen Hashfunktion und einzufügende Elemente erfüllen, damit diese Annahme gilt? Offensichtlich gilt die Annahme, falls die beiden folgenden Punkte erfüllt sind:

1. Die Hashfunktion h verteilt das Universum gleichmäßig über die Menge $\{0, 1, 2, \ldots, m-1\}$, d.h. für alle a, b mit $0 \leq a < m$ und $0 \leq b < m$ ist $|h^{-1}(a)| = |h^{-1}(b)|$.

2. Alle Elemente des Universums U sind an jeder Stelle der Folge der einzufügenden Elemente gleich wahrscheinlich.

Die erste Annahme kann leicht erfüllt werden. Sei $|U| = N$, d.h. als Universum kann alternativ die Menge $\{0, 1, 2, \ldots, N-1\}$ betrachtet werden. Falls m ein Teiler von N ist, erfüllt die Hashfunktion $h(S) := S \mod m$ die Annahme exakt. Falls m kein Teiler von N, aber N wesentlich größer als m ist, was bei praktischen Anwendungen im Allgemeinen stets zutrifft, so wird die Voraussetzung fast erfüllt.
Die zweite Annahme ist jedoch wesentlich kritischer, da vom Benutzer ein bestimmtes Verhalten verlangt wird und bei der Wahl der Hashfunktion die Umstände der Benutzung oft noch nicht klar sind. Wie oben erwähnt wurde, muss eine solche Abweichung nicht unbedingt nur schlechte Effekte haben, da dadurch z.B. die Anzahl der Kollisionen vermindert werden kann.
Abweichungen von diesem Modell ergeben sich auch dadurch, dass im Allgemeinen vorausgesetzt wird, dass die n in die Tabelle einzutragenden Elemente alle verschieden sind. Sollen nämlich mehrfach vorkommende Elemente nur einmal eingetragen werden, so wird mit n die Anzahl der verschiedenen Elemente gezählt. Für diesen Fall ist die zweite Annahme verletzt, da ein Element nicht mehrfach ausgewählt werden kann. Der dadurch entstehende Fehler ist jedoch vernachlässigbar, wenn $|U|$ wesentlich größer als m ist.
Als alternative Verfahren, die ohne diese Annahmen auskommen, bieten sich perfektes Hashing und universelles Hashing an, die anschließend an die Analyse der „gewöhnlichen" Hashverfahren behandelt werden.

Kapitel 2

Mathematische Grundlagen

In diesem Kapitel werden die für die Arbeit wesentlichen mathematischen Grundlagen wiederholt bzw. vertieft. Größtenteils ist dies Stoff, der an der TU-Wien in den Vorlesungen Analyse von Algorithmen, Diskrete Methoden [14], Wahrscheinlichkeitstheorie[3] und Zahlentheorie für Technische Mathematiker[15] behandelt wird. Das Kapitel ist eher kurz gehalten, für weitere Details und nicht gebrachte Beweise wird auf das Literaturverzeichnis verwiesen.

2.1 Kombinatorik

2.1.1 Erzeugende Funktionen

Definition 2.1 (gewöhnliche Erzeugende Funktion [14])
Sei $(a_n)_{n\geq 0}$ eine Folge komplexer Zahlen, dann heißt die formale Potenzreihe $a(x) := \sum_{n\geq 0} a_n x^n$ gewöhnliche erzeugende Funktion (EF) der Folge $(a_n)_{n\geq 0}$.

Definition 2.2 (exponentielle Erzeugende Funktion [14])
Sei $(a_n)_{n\geq 0}$ eine Folge komplexer Zahlen, dann heißt die formale Potenzreihe $a(x) := \sum_{n\geq 0} a_n \frac{x^n}{n!}$ exponentielle erzeugende Funktion (EEF) der Folge $(a_n)_{n\geq 0}$.

Erzeugende Funktionen können im Fall eines positiven Konvergenzradius R auch als Funktionen im gewöhnlichen Sinn verstanden werden.

Satz 2.1 (Konvergenz von Potenzreihen [14, 24])
Sei $(a_n)_{n\geq 0}$ eine Folge komplexer Zahlen mit Konvergenzradius $R := \left(\limsup \sqrt[n]{|a_n|}\right)^{-1}$. Falls R positiv ist, so konvergiert die Potenzreihe $\sum_{n\geq 0} a_n x^n$ für $x \in \mathbb{C}$ mit $|x| < R$ absolut und stellt eine analytische Funktion dar.
Ist umgekehrt $a(x)$ eine Funktion, die durch die Potenzreihe $\sum_{n\geq 0} a_n x^n$ für $|x| < R$ dargestellt wird, so ist die Folge $(a_n)_{n\geq 0}$ durch $a_n = \frac{1}{n!} a^{(n)}(0)$ gegeben.

Erzeugende Funktionen verschiedener Folgen stellen demnach verschiedene Funktionen dar, wodurch es im konvergenten Fall möglich ist, nur die Funktion zu betrachten.

Satz 2.2 (Berechnung von Summen mit EF [14])
Sei $a(x)$ die erzeugende Funktion der Folge $(a_n)_{n\geq 0}$. Dann besitzt die Folge $\left(c_n = \sum_{k=0}^{n} a_k\right)_{n\geq 0}$ die erzeugende Funktion $c(x) = \frac{1}{1-x} a(x)$.

Satz 2.3 (wichtige Reihenentwicklungen [14])
Mit Konvergenzradius 1 *bzw.* ∞ *gilt:*

$$\frac{1}{1-x} = \sum_{n\geq 0} x^n \qquad (1+x)^\alpha = \sum_{n\geq 0} \binom{\alpha}{n} x^n$$

$$\frac{1}{(1-x)^m} = \sum_{n\geq 0} \binom{m+n-1}{m-1} x^n \qquad e^x = \sum_{n\geq 0} \frac{1}{n!} x^n$$

$$\ln(1+x) = \sum_{n\geq 1} \frac{(-1)^{n+1}}{n} x^n$$

Die folgende, nach Ramanujan benannte Funktion $Q(n)$ tritt bei vielen Laufzeitanalysen, insbesondere auch bei Hashing mit linearem Sondieren auf. Die Funktionen $Q_r(m,n)$ stellen eine Erweiterung der Q Funktion dar:[1]

Definition 2.3 (Ramanujans Q-Funktion [25])

$$Q(n) := \sum_{k=0}^{n} \frac{n^{\underline{k}}}{n^k} \qquad Q_r(m,n) := \sum_{k=0}^{n} \binom{r+k}{k} \frac{n^{\underline{k}}}{m^k}$$

Erzeugende Funktionen spielen auch im Zusammenhang mit kombinatorischen Konstruktionen eine wichtige Rolle, z.B. für die markierten Wurzelbäume. Ein solcher Baum besteht aus einem ausgezeichneten Knoten, der Wurzel, von der aus eine Menge von Ästen abzweigt, die jeweils für sich betrachtet, wieder die Struktur markierter Wurzelbäume haben. Besitzt der Baum n Knoten, so erhalten diese die Markierungen von $1, 2, \ldots, n$.

Satz 2.4 (markierte Wurzelbäume [14])
Die EEF der markierten Wurzelbäume, die im weiteren mit $T(x)$ bezeichnet wird, erfüllt die Beziehung $T(x) = xe^{T(x)}$. Mit Hilfe der Lagrangeschen Inversionsformel [14, 41] erhält man daraus:

$$T(x) = \sum_{n\geq 1} \frac{n^{n-1}}{n!} x^n \qquad T(x)^m = m \sum_{n\geq m} \frac{n^{n-m-1}}{(n-m)!} x^n$$

$$\frac{1}{1-T(x)} = 1 + \sum_{n\geq 1} n^n \frac{x^n}{n!} \qquad \ln \frac{1}{1-T(x)} = \sum_{n\geq 0} Q(n) n^{n-1} \frac{x^n}{n!}$$

2.1.2 Asymptotische Entwicklungen

Nicht für alle der Funktionen, die bei den Analysen in folgenden Kapiteln auftreten, ist das asymptotische Verhalten klar ersichtlich. Die später benötigten Resultate werden hier angeführt:

Definition 2.4 (Eulersche Konstante)

$$\gamma := \frac{1}{2} - \int_1^\infty \left(\{x\} - \frac{1}{2}\right) \frac{dx}{x^2} = 0.57721\ldots$$

Definition 2.5 (Harmonische Zahlen)
Die harmonische Zahl H_n entspricht der $n-$ten Partialsumme der harmonischen Reihe, analog kann für $i \geq 2$ die Bezeichnung $H_n^{[i]}$ verwendet werden:

$$H_n := \sum_{k=1}^{n} \frac{1}{k} \qquad H_n^{[r]} := \sum_{k=1}^{n} \frac{1}{k^i}$$

[1] mit $n^{\underline{k}}$ werden dabei die fallenden Faktoriellen bezeichnet, siehe Anhang B

Satz 2.5 (Stirlingsche Formel [21])

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Satz 2.6 (Asymptotische Entwicklungen [21, 19, 16, 41])
Die Funktion $T(x)$ besitzt eine dominante Singularität für $x = \frac{1}{e}$, mit $\delta(x) = \sqrt{2(1-ex)}$ gilt:

$$\begin{aligned}
T(x) &= 1 - \delta(x) + \frac{1}{3}\delta(x)^2 - \frac{11}{72}\delta(x)^3 + \frac{43}{540}\delta(x)^4 + O\big(\delta(x)^5\big)\\
H_n &= \gamma + \ln(n) + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \frac{1}{252n^6} + O\big(n^{-8}\big)\\
H_n^{[2]} &= \frac{\pi^2}{6} - \frac{1}{n} + \frac{1}{2n^2} - \frac{1}{6n^3} + \frac{1}{30n^5} - \frac{1}{42n^7} + O\big(n^{-8}\big)\\
Q(n) &\sim \sqrt{\frac{\pi n}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2n}} - \frac{4}{135n} + \ldots
\end{aligned}$$

2.1.3 Fibonaccizahlen

Definition 2.6 (verallgemeinerte Fibonaccizahlen)
Sei d eine natürliche Zahl größer oder gleich zwei. Dann heißen die durch $F_d(k) = \sum_{i=1}^d F_d(k-i)$ mit $F_d(k) = 0$ für $k \leq 0$ und $F_d(1) = 1$ rekursiv definierten Zahlen verallgemeinerte Fibonaccizahlen.

Satz 2.7 (von Rouché [24])
Sei G ein Gebiet, das eine abgeschlossene Kreisscheibe K enthält und f, g holomorph auf G. Falls f und g keine Nullstellen am Rand von K besitzen und dort die Ungleichung $|f-g| < |g|$ gilt, so liegen gleich viele Nullstellen von f und g im Inneren von K.

Satz 2.8 ([46])
Der Grenzwert $\phi_d := \lim_{k\to\infty} \sqrt[k]{F_d(k)}$ existiert stets. Es gilt $\phi_2 = \frac{1+\sqrt{5}}{2} < \phi_3 < \phi_4 < \ldots < 2$, $F_d(k) \geq \phi_d^{k-2}$ für $k \geq 1$ und $F_d(k) \leq \phi_d^k$.

Beweis:
Das charakteristische Polynom der homogenen linearen Rekursion mit konstanten Koeffizienten lautet

$$\chi(t) = t^d - t^{d-1} - t^{d-2} - \ldots - t - 1 = t^d - \frac{t^d - 1}{t-1} = \frac{t^{d+1} - 2t^d + 1}{t-1}$$

und besitzt paarweise verschiedene komplexe Nullstellen $t_1, t_2, \ldots, t_d$. Es gibt daher (komplexe) Konstanten $c_1, c_2, \ldots, c_d$ sodass

$$F_d(k) = c_1 t_1^k + c_2 t_2^k + \ldots + c_d t_d^k \qquad \text{für alle } k \in \mathbb{N}$$

gilt. Wegen $\chi(2) = 1$ und $\chi(1) = 1 - d < 0$ muss $\chi(t)$ eine Nullstelle im Intervall $(1, 2)$ besitzen, diese wird ab jetzt mit t_1 bezeichnet. Alle weiteren Nullstellen haben höchstens Betrag eins, denn für $0 < \varepsilon < \frac{1}{d}$ gilt $\varepsilon - \varepsilon^2 d = \varepsilon(1 - \varepsilon d) > 0$, woraus weiters folgt:

$$\begin{aligned}
&1 < 1 + \varepsilon - \varepsilon^2 d \leq 1 + \varepsilon(d-1) - \varepsilon^2 d = (1 + \varepsilon d)(1 - \varepsilon) \leq (1+\varepsilon)^d (1-\varepsilon)\\
&1 < (1+\varepsilon)^d(1-\varepsilon) \iff (1+\varepsilon)^d(-2+1+\varepsilon) < -1 \iff (1+\varepsilon)^{d+1} < 2(1+\varepsilon)^d - 1\\
&|t^{d+1} - 2t^d + 1 - (-2t^d + 1)| = |t|^{d+1} < 2|t|^d - 1 \leq |-2t^d + 1|
\end{aligned}$$

Damit sind die Voraussetzungen für den Satz von Rouché erfüllt, aus dem folgt, dass $t^{d+1} - 2t^d + 1$ und $t^d - \frac{1}{2}$ die selbe Anzahl von Nullstellen mit Betrag kleiner $1 + \varepsilon$ besitzen. Demnach müssen außer 1, das nicht Nullstelle von $\chi(t)$ ist, weitere $d - 1$ Nullstellen von $\chi(t)$ in diesem Bereich liegen. Damit gilt:

$$\frac{F^d(k)}{t_1^k} = c_1 + c_2\Big(\frac{t_2}{t_1}\Big)^k + c_3\Big(\frac{t_3}{t_1}\Big)^k + \ldots + c_d\Big(\frac{t_d}{t_1}\Big)^k \quad \to \quad c_1 \text{ für } k \to \infty$$

Demnach muss $\sqrt[k]{F_d(k)}$ gegen die eindeutig bestimmte reelle Nullstelle von $\chi(t)$ im Intervall $(1, 2)$ konvergieren. Für $k = 2, 3, \ldots, d + 1$ ist $F_d(k) = 2F_d(k - 1) = 2^{k-2} > \phi_d^{k-2}$. Für größere Werte von k folgt induktiv:

$$\begin{aligned} F_d(k+1) &= F_d(k) + F_d(k-1) + \ldots + F_d(k-d+1) \geq \phi_d^{k-2} + \phi_d^{k-3} + \ldots + \phi_d^{k-d-1} \\ &= \phi_d^{k-d-1}(\phi_d^{d-1} + \phi_d^{d-2} + \ldots + 1) = \phi_d^{k-d-1}\phi_d^d = \phi_d^{k-1} \end{aligned}$$

Analog folgt wegen $F_d(k) \leq \phi_d^k$ für $k = -d+2, -d+3, \ldots, 1$ dass $F_d(k) \leq \phi_d^k$ gilt. Die restlichen Aussagen sind trivial. ■

2.1.4 Identitäten

Satz 2.9 ([35])
Für natürliche Zahlen m, n und k gilt stets:

$$\sum_{m=0}^{k} \binom{m}{n} = \binom{k+1}{n+1}$$

Beweis: (mit vollständiger Induktion)

$$\underline{k = 0:} \quad \binom{0}{n} = \delta_{n,0} = \binom{1}{n+1}$$

$$\underline{k \to k+1:} \quad \sum_{m=0}^{k+1} \binom{m}{n} = \sum_{m=0}^{k} \binom{m}{n} + \binom{k+1}{n} = \binom{k+1}{n+1} + \binom{k+1}{n} = \binom{k+2}{n+1}$$

■

Satz 2.10 ([18])

$$\sum_{i=0}^{n} i^{\underline{s}} = \frac{(n+1)^{\underline{s+1}}}{s+1}$$

Beweis:
Die erzeugende Funktion der Folge $(i^{\underline{s}})_{i \geq 0}$ erhält man durch s maliges Ableiten der Funktion $\frac{1}{1-x}$ und anschließender Multiplikation mit x^s. Wie man durch vollständige Induktion leicht zeigen kann, gilt $\left(\frac{1}{1-x}\right)^{(s)} = \frac{s!}{(1-x)^{s+1}}$. Damit erhält man durch Verwendung von Satz 2.2:

$$\begin{aligned} \sum_{i=0}^{n} i^{\underline{s}} &= [x^{n-s}]\frac{1}{1-x}\Big(\frac{1}{1-x}\Big)^{(s)} = [x^{n-s}]\frac{s!}{(1-x)^{s+2}} \\ &= \binom{s+2+n-s-1}{s+2-1} s! = \binom{n+1}{s+1} s! = \frac{(n+1)^{\underline{s+1}}}{s+1} \end{aligned}$$

■

Satz 2.11
Für die Summe der Quadrate der ersten $k-1$ positiven ganzen Zahlen gilt:

$$\sum_{j=1}^{k-1} j^2 = \frac{k(k-1)(2k-1)}{6}$$

Beweis: (mit vollständiger Induktion)

$$\underline{k=2:} \quad \sum_{j=1}^{1} j^2 = 1 = \frac{2\cdot 1\cdot 3}{6}$$

$$\underline{k\to k+1:} \quad \sum_{j=1}^{k} j^2 = \frac{k(k-1)(2k-1)}{6} + k^2 = \frac{2k^3-3k^2+k+6k^2}{6} = \frac{(k+1)k(2k+1)}{6}$$

(alternativ mit EF)

$$\begin{aligned}\sum_{j=1}^{k-1} j^2 &= \sum_{j=0}^{k-1}\big(j(j-1)+j\big) = [x^{k-1}]\frac{1}{1-x}\left(\frac{2x^2}{(1-x)^3}+\frac{x}{(1-x)^2}\right) = [x^{k-1}]\frac{2x^2+x-x^2}{(1-x)^4}\\ &= \binom{4+k-2-1}{4-1}+\binom{4+k-3-1}{4-1} = \frac{(k+1)k(k-1)}{3!}+\frac{k(k-1)(k-2)}{3!}\\ &= \frac{k(k-1)(2k-1)}{6}\end{aligned}$$

■

Satz 2.12 (Abels Verallgemeinerung des binomschen Lehrsatzes [25])

$$(x+y)^n = \sum_{k=0}^{n}\binom{n}{k}x(x+k)^{k-1}(y-k)^{n-k}$$

2.1.5 Graphen

Definition 2.7 (Graph [14])
Ein ungerichteter bzw. gerichteter Graph G besteht aus der Knotenmenge V, der Kantenmenge E und aus einer Abbildung Γ, die jeder Kante e aus E ein ungeordnetes bzw. geordnetes Paar von Knoten v_1, v_2 aus V zuordnet, d.h. $e = (v_1, v_2)$ bzw. $e = \langle v_1, v_2\rangle$.

Nimmt man von einem gegebenen Graphen einzelnen Knoten und/oder Kanten weg, sodass wieder ein Graph entsteht, spricht man von einem Teilgraph. Auf diese Weise können leicht neue Graphen konstruiert werden. Die obige Definition lässt auch Kanten der Form $\langle v, v\rangle$ bzw. (v, v) zu. Diese Kanten werden Schlingen genannt. Weiters kann der Fall auftreten, dass es mindestens zwei unterschiedliche Kanten in einem Graph gibt, denen das selbe Knotenpaar zugeordnet ist. Ein solcher Graph besitzt eine Mehrfachkante. Ein Graph ohne Schlingen und Mehrfachkanten heißt schlicht.

Definition 2.8 (Kantenfolge, Kreis [14])
Eine Folge von Kanten $e_1, e_2, \ldots, e_n$ eines ungerichteten Graphen heißt Kantenfolge, wenn der Graph Knoten $v_0, v_1, \ldots, v_n$ besitzt, sodass $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_n = (v_{n-1}, v_n)$ gilt. Falls die Kantenfolge geschlossen ist, d.h. $v_0 = v_n$ und alle anderen Knoten paarweise verschieden sind, so nennt man sie einen Kreis.

Definition 2.9 (zusammenhängend, Zusammenhangskomponenten [14])
Ein ungerichteter Graph heißt zusammenhängend, wenn je zwei Knoten durch eine Kantenfolge verbunden sind. Die maximalen zusammenhängenden Teilgraphen eines ungerichteten Graphen werden Zusammenhangskomponenten genannt.

Definition 2.10 (Baum [14])
Ein schlichter, ungerichteter Graph, der keine Kreise enthält und zusammenhängend ist, wird Baum genannt.

Definition 2.11 (paarer Graph [14])
Ein ungerichteter Graph heißt paarer Graph, wenn dessen Knotenmenge in zwei disjunkte Teilmengen V_1, V_2 zerlegt werden kann, sodass alle Kanten nur zwischen V_1 und V_2 verlaufen.

2.2 Wahrscheinlichkeitstheorie

2.2.1 Wahrscheinlichkeitserzeugende Funktionen

Definition 2.12
Falls die zu einer Erzeugenden Funktion gehörende Folge $(a_n)_{n\in\mathbb{N}}$ von einer Wahrscheinlichkeitsverteilung stammt, d.h. jedes $a_n \geq 0$ und $\sum_{n\geq 0} a_n = 1$ ist, spricht man von einer Wahrscheinlichkeitserzeugenden Funktion (WEF).

Diese spezielle Art von erzeugenden Funktionen konvergiert sicher für alle $z \in \mathbb{C}$ mit $|z| \leq 1$. Falls η die Zufallsvariable mit $\mathbb{P}(\eta = n) = a_n$ bezeichnet, gilt für die zugehörige Erzeugende Funktion:

$$F(z) = \sum_{n\geq 0} a_n z^n = \mathbb{E}(z^\eta)$$

Satz 2.13 (Berechnung von Momenten [3])
Wenn $F(z)$ die zur Zufallvariable η gehörende WEF bezeichnet, so gilt:

$$F^{(k)}(1) = \mathbb{E}\big(\eta(\eta-1)\dots(\eta-k+1)\big)$$

Beweis:
Die Behauptung ergibt sich unmittelbar durch Differenzieren. ■
Damit können Erwartungswert und Varianz folgendermaßen berechnet werden:

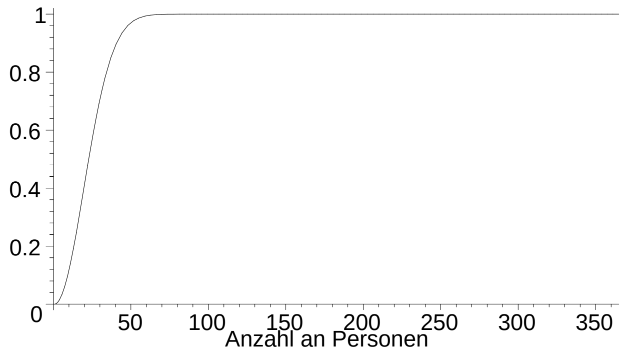
$$\mathbb{E}\eta = F'(1) \qquad \mathbb{V}\eta = F''(1) + F'(1) - (F'(1))^2$$

2.2.2 Das Geburtstagsparadoxon

Wie viele Personen müssen in einem Raum anwesend sein, damit die Wahrscheinlichkeit, dass mindestens zwei von ihnen am selben Tag im Jahr Geburtstag haben größer gleich $\frac{1}{2}$ ist?
Die Antwort auf diese Frage ist überraschend niedrig. Wenn man allgemein annimmt, dass ein Jahr T Tage besitzt, so beträgt die Wahrscheinlichkeit, dass eine Person nicht am selben Tag wie eine festgelegte zweite Geburtstag hat, $\frac{T-1}{T}$. Analog erhält man für die Wahrscheinlichkeit, dass $n < T$ Personen an lauter verschiedenen Tagen Geburtstag haben, den Ausdruck

$$\frac{T-1}{T}\frac{T-2}{T}\frac{T-3}{T}\dots\frac{T-(n-1)}{T} = \frac{n!}{T^n}\binom{T}{n}.$$

Die folgende Graphik [41] zeigt die Wahrscheinlichkeit, dass mindestens zwei Personen am selben Tag Geburtstag haben für $T = 365$.



Wie man der Graphik entnehmen kann, ist zum ersten Mal für $n = 25$ die gesuchte Wahrscheinlichkeit größer als $\frac{1}{2}$.

2.2.3 Airy Verteilung

Die Airy Verteilung tritt bei diversen kombinatorischen Problemen wie z.B. bei Pfadlängen in Bäumen, Hashing mit linearem Sondieren oder bei der Brownschen Bewegung als Grenzverteilung auf. Die Verteilung kann auf folgende Art über ihre Momente erklärt werden:

Definition 2.13 (Airy Verteilung [17, 18])
Eine Zufallsvariable A besitzt eine Airy Verteilung wenn für die Momente ($r \geq 1$)

$$\mathbb{E}(A^r) = \Omega_r \frac{-\Gamma\left(-\frac{1}{2}\right)}{\Gamma\left(\frac{3r-1}{2}\right)} = \Omega_r \frac{2\sqrt{\pi}}{\Gamma\left(\frac{3r-1}{2}\right)}$$

gilt, wobei die Konstanten Ω_r durch $\Omega_0 = -1$ und ansonsten durch die folgende Rekursion definiert werden:

$$2\Omega_r = (3r-4)r\Omega_{r-1} + \sum_{j=1}^{r-1} \binom{r}{j} \Omega_j \Omega_{r-j}$$

Eine ausführliche Untersuchung der Airy Verteilung findet sich in [17]. Die Werte einiger Konstanten und Momente sind auch in folgender Tabelle angeführt:

r	0	1	2	3	4	5	6	7
Ω_r	-1	$\frac{1}{2}$	$\frac{5}{4}$	$\frac{45}{4}$	$\frac{3315}{16}$	$\frac{25425}{4}$	$\frac{18635625}{64}$	18592875
$\mathbb{E}(A^r)$	1	$\sqrt{\pi}$	$\frac{10}{3}$	$\frac{15}{4}\sqrt{\pi}$	$\frac{884}{63}$	$\frac{565}{32}\sqrt{\pi}$	$\frac{662600}{9009}$	$\frac{19675}{192}\sqrt{\pi}$

2.3 Zufallszahlen

Um die im Kapitel 7 benötigten zufälligen Eingabedaten für Hashverfahren zu simulieren, ist ein Zufallszahlengenerator erforderlich. Zwar verfügen praktisch alle Programmiersprachen über einen solchen Generator, diese sind jedoch im Allgemeinen für die hier benötigten großen Zahlenmengen nicht gut genug.
„Echte" Zufallszahlen werden an Hand von physikalischen Experimenten erstellt, im einfachsten Fall denke man an das wiederholte Werfen einer Münze, um eine zufällige Binärfolge zu erzeugen. Da diese Experimente für die praktische Verwendung zu aufwändig sind, muss man sich mit so genannten Pseudozufallszahlen begnügen:

Definition 2.14 (Pseudozufallszahlen [34])
Eine Folge von Zahlen $x_1, x_2, x_3, \ldots$, die dadurch entsteht, dass jedes Folgenglied x_n durch einen deterministischen Algorithmus aus den Zahlen $x_{n-1}, \ldots, x_{n-r}$ berechnet wird, bezeichnet man als Pseudozufallszahlen, sofern sie mit statistischen Tests nicht von einer echten Zufallsfolge zu unterscheiden ist.

Grundsätzlich kann jeder Anpassungstest verwendet werden, um eine Folge von Pseudozufallszahlen zu testen. Da die einzelnen Generatoren unterschiedliche Schwachpunkte haben, empfiehlt es sich, eine Vielzahl von Tests durchzuführen, die speziell zu diesem Zweck entwickelt wurden. Details zu den wichtigsten Tests und ein fertiges Testprogramm finden sich in [31].

Da eine Hashfunktion Zufallszahlen als Funktionswerte annehmen soll, scheint die Verwendung eines Pseudozufallszahlengenerators auch zur Erzeugung der Hashwerte sinnvoll, z.B. um das später behandelte Uniform Hashing zu simulieren.

2.3.1 Generatoren für Pseudozufallszahlen

Lineare Schieberegister

Verwendet man $x_{n+l} = \sum_{i=1}^{l} c_i x_{n+i-1}$ als Übergangsfunktion, wobei die Berechnungen im endlichen Körper $\mathbb{Z}_q$ erfolgen, so spricht man von einem linearen Schieberegister der Länge l.

Satz 2.14 (lineare Schieberegister [34])
Ein lineares Schieberegister der Länge l mit der größtmöglichen Periode $q^l - 1$ kann mit Hilfe eines normierten irreduziblen Polynoms $p(x)$ konstruiert werden, welches $x^{q^l-1} - 1$ teilt, aber keines der Polynome $x^k - 1$ mit $k < q^l - 1$. Die Koeffizienten der Übergangsfunktion ermittelt man aus der Gleichung $p(x) = (-1)^l \left(x^l - \sum_{i=1}^{l} c_i x^{i-1}\right)$.

Beispiel 8 *Um ein Schieberegister der Länge 3 mit Periode $3^3 - 1 = 26$ über $GF(3)$ zu erhalten wählt man z.B. $p(x) = x^3 + x^2 + 2x + 1$ und erhält die Koeffizienten $c_1 = -1 = 2, c_2 = 1, c_3 = 2$.*

Kongruenzgeneratoren

Ein Kongruenzgenerator verwendet die Übergangsfunktion $x_{n+1} = ax_n + b \mod m$, wobei a und b geeignet gewählte Konstanten sind. Für den häufig vorkommenden Fall, dass eine Zweierpotenz als Modul verwendet wird, empfiehlt es sich $a \equiv 1 \mod 4$ und b ungerade zu wählen, da dann die maximal mögliche Periode m erreicht wird (siehe [39]).

Multiplikation mit Übertrag

Bei diesem Konzept handelt es sich im Prinzip um eine Verbesserung der Kongruenzgeneratoren. Sie verwenden die Übergangsfunktion $x_{n+1} = ax_n + c_n \mod m$. Der Multiplikator a wird wieder konstant gewählt, der Summand c_n ergibt sich als „Übertrag": $c_n = \lfloor (ax_{n-1} + c_{n-1})/m \rfloor$. Die Parameter a und m sollten dabei so gewählt werden, dass sowohl $am - 1$ als auch $\frac{am}{2} - 1$ Primzahlen sind (siehe [31]). Die mit diesem Generator erzeugte Pseudozufallsfolge scheint alle Tests aus [31] zu bestehen.

KISS - Generator

Um noch bessere Resultate zu erzielen, empfiehlt es sich, die Ergebnisse verschiedener Generatoren zu verknüpfen. Ein Beispiel hierfür ist der so genannte „Keep It Simple Stupid" Generator von George Marsaglia, der die Ergebnisse der drei oben vorgestellten Generatoren additiv verknüpft. Auch der KISS Generator scheint alle Tests aus [31] zu bestehen.

2.4 Zahlentheorie

2.4.1 Der Chinesische Restsatz

Satz 2.15 (Chinesischer Restsatz [15])
Zu paarweise teilerfremden, positiven ganzen Zahlen $m_1, m_2, \ldots, m_r$ und beliebigen ganzen Zahlen $a_1, a_2, \ldots, a_r$ gibt es stets eine ganzzahlige Lösung des Kongruenzsystems $x \equiv a_i \mod m_i$ mit $i = 1, \ldots, r$. Diese Lösung ist eindeutig modulo $M := m_1 m_2 \ldots m_r$.

Beweis: für $i = 1, \ldots, r$ gilt:

$$M_i := m_1 m_2 \ldots m_{i-1} m_{i+1} \ldots m_r \implies \mathrm{ggT}(m_i, M_i) = 1 \implies \exists b_i : b_i M_i \equiv 1 \mod m_i$$
$$x := \sum_{k=1}^{r} a_k b_k M_k \equiv a_i b_i M_i \equiv a_i \mod m_i$$

Sei nun y eine weitere Lösung des Kongruenzsystems:

$$\implies x - y \equiv 0 \mod m_i \implies m_i | x - y \implies M | x - y$$

■

2.4.2 Darstellung einer natürlichen Zahl als Summe von Quadraten

Satz 2.16 ([15])
Eine natürliche Zahl n ist genau dann als Summe zweier Quadrate natürlicher Zahlen darstellbar, wenn alle Primteiler p von n, für die $p \equiv 3 \mod 4$ gilt, in gerader Vielfachheit auftreten.

2.4.3 Kettenbrüche

Um den Beweis zu bringen, dass sich das „goldene Verhältnis“ $\phi^{-1} = \frac{\sqrt{5}-1}{2}$ besonders zur Verwendung in Hashfunktionen, die mit der Multiplikationsmethode generiert werden, eignet, werden einige Aussagen über Kettenbrüche benötigt, die deshalb in diese Arbeit aufgenommen wurden. Eine ausführliche Behandlung von Kettenbrüchen findet sich z.B. in [1] oder [23].

Definition 2.15 (endlicher Kettenbruch)
Unter einem endlichen Kettenbruch $[a_0, a_1, a_2, \ldots, a_k]$ mit $a_1 > 0$ für alle $i > 0$ versteht man einen Ausdruck der Form:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots \over a_{k-1} + \cfrac{1}{a_k}}}}$$

Satz 2.17 ([15])
Definiert man rekursiv die Polynome $p_k = p_k(a_0, \ldots, a_{k-1})$ und $q_k = q_k(a_0, \ldots, a_{k-1})$ durch $p_0 = 1, p_1 = a_0, q_0 = 0, q_1 = 1$ und $p_{k+1} = a_k p_k + p_{k-1}$, $q_{k+1} = a_k q_k + q_{k-1}$ für $k \geq 1$, so gilt:

$$[a_0, a_1, a_2, \ldots, a_{k-1}] = \frac{p_k}{q_k}$$

Beweis: (mit vollständiger Induktion)

$$\underline{k=0,1:}\quad [a_0]=\frac{p_1}{q_1},\qquad [a_0,a_1]=a_0+\frac{1}{a_1}=\frac{a_1a_0+1}{a_1}=\frac{p_2}{q_2}$$

$$\underline{k\to k+1:}\quad [a_0,a_1,a_2,\dots,a_k]=\left[a_0,a_1,a_2,\dots,a_{k-1}+\frac{1}{a_k}\right]$$

$$=\frac{\left(a_{k-1}+\frac{1}{a_k}\right)p_{k-1}+p_{k-2}}{\left(a_{k-1}+\frac{1}{a_k}\right)q_{k-1}+q_{k-2}}=\frac{p_k+\frac{1}{a_k}p_{k-1}}{q_k+\frac{1}{a_k}q_{k-1}}=\frac{a_kp_k+p_{k-1}}{a_kq_k+q_{k-1}}=\frac{p_{k+1}}{q_{k+1}}$$

■

Definition 2.16 (Näherungsbruch [15])
Die Brüche $\frac{p_k}{q_k}$ werden Näherungsbrüche genannt.

Satz 2.18 ([15])

$$p_{k+1}q_k-p_kq_{k+1}=(-1)^{k+1}\qquad\qquad p_{k+2}q_k-p_kq_{k+2}=(-1)^{k+1}a_{k+1}$$

Beweis: (mit vollständiger Induktion)

$$\underline{k=0:}\quad p_1q_0-p_0q_1=a_0\cdot 0-1\cdot 1=-1$$

$$\underline{k-1\to k:}\quad p_{k+1}q_k-p_kq_{k+1}=(a_kp_k+p_{k-1})q_k-p_k(a_kq_k+q_{k-1})=-(p_kq_{k-1}-p_{k-1}q_k)$$

Die zweite Formel wird analog gezeigt. ■

Satz 2.19 (unendlicher Kettenbruch [15, 23])
Wenn a_0 eine ganze Zahl ist und alle a_k für $k\geq 1$ positive natürliche Zahlen sind, so konvergiert die Folge $\big([a_0,a_1,a_2,\dots,a_k]\big)_{k\geq 0}$ gegen eine reele Zahl α. Diesen Grenzwert nennt man den unendlichen Kettenbruch $[a_0,a_1,a_2,\dots]$. Jede reelle Zahl α besitzt eine Kettenbruchentwicklung. Bei irrationalem α entsteht ein unendlicher Kettenbruch, der dieser Zahl umkehrbar eindeutig entspricht.

Definition 2.17 (beste Approximation)
Ein Bruch $\frac{a}{b}$ mit positivem Nenner heißt beste Approximation von $\alpha\in\mathbb{R}$, wenn für jeden von $\frac{a}{b}$ verschiedenen Bruch $\frac{c}{d}$ mit $0<d\leq b$ die Ungleichung $|d\alpha-c|>|b\alpha-a|$ erfüllt ist.

Satz 2.20 ([15])
Setzt man $\alpha_k:=[a_k,a_{k+1},a_{k+2},\dots]$, so gilt:

$$q_k\alpha-p_k=\frac{(-1)^{k+1}}{\alpha_kq_k+q_{k-1}}=\frac{(-1)^{k+1}\alpha_{k+1}}{\alpha_{k+1}q_{k+1}+q_k}$$

Beweis:

$$\frac{p_{k+1}}{q_{k+1}}-\frac{p_k}{q_k}=\frac{(-1)^{k+1}}{q_kq_{k+1}}\quad\text{(folgt aus Satz 2.18)}\qquad \alpha=[a_0,a_1,a_2,\dots,a_{k-1},\alpha_k]$$

$$\alpha=\frac{\alpha_kp_k+p_{k-1}}{\alpha_kq_k+q_{k-1}}=:\frac{\overline{p}_{k+1}}{\overline{q}_{k+1}}\qquad\Longrightarrow\qquad \alpha-\frac{p_k}{q_k}=\frac{(-1)^{k+1}}{q_k\overline{q}_{k+1}}$$

Die zweite Formel wird analog gezeigt. ■

Insbesondere folgt damit, dass für gerades k immer $\frac{p_{k+1}}{q_{k+1}}<\alpha<\frac{p_k}{q_k}$ gilt. Weiters folgt, dass stets $|q_{k+1}\alpha-p_{k+1}|<|q_k\alpha-p_k|$ gilt.

Satz 2.21 ([15])
Jeder Näherungsbruch (mit der möglichen Ausnahme $\frac{p_1}{q_1}$) ist beste Approximation und umgekehrt.

Beweis: Sei $0 < d < q_{k+1}$:
Aus $p_{k+1}q_k - p_kq_{k+1} = (-1)^{k+1}$ folgt mit der Cramerschen Regel, dass folgendes Gleichungssystem eine ganzzahlige in u und v Lösung besitzt:

$$\begin{aligned} c &= up_{k+1} + vp_k \\ d &= uq_{k+1} + vq_k \end{aligned}$$

Für $a_1 = 1$ ist $q_1 = q_2 = 1$ und $\frac{p_1}{q_1} = a_0$ ist nicht beste Approximation, weil $\frac{p_2}{q_2} = a_0+1$ gilt, sei also $a_1 > 1$. Falls $v = 0$ ist, erhält man einen Wert von d, der außerhalb des zulässigen Bereichs ist. Deswegen muss entweder $u = 0$, was zum Bruch $\frac{p_k}{q_k}$ führt, oder $\operatorname{sgn}(u) \neq \operatorname{sgn}(v)$ gelten. Im zweiten Fall gilt dann $\operatorname{sgn}(u(q_{k+1}\alpha - p_{k+1})) = \operatorname{sgn}(v(q_k\alpha - p_k))$ wegen $\operatorname{sgn}(q_{k+1}\alpha - p_{k+1}) \neq \operatorname{sgn}(q_k\alpha - p_k)$.

$$\begin{aligned} d\alpha - c &= (uq_{k+1} + vq_k)\alpha - (up_{k+1} + vp_k) = u(q_{k+1}\alpha - p_{k+1}) + v(q_k\alpha - p_k) \\ |d\alpha - c| &= |u(q_{k+1}\alpha - p_{k+1})| + |v(q_k\alpha - p_k)| > |q_k\alpha - p_k| \end{aligned}$$

■

Weiters gilt sogar $|q_k\alpha - p_k| < |(q_k + s)\alpha - t|$ für $s \neq q_{k+1} - q_k$, $0 \leq s < q_{k+1}$ und $t \in \mathbb{Z}$, denn die Gleichung $uq_{k+1} + vq_k = s + q_k$ besitzt dann keine Lösung mit $v = 0$, weshalb in diesem Fall wieder $\operatorname{sgn}(u) \neq \operatorname{sgn}(v)$ gelten muss.

Im Folgenden sei Θ eine irrationale Zahl mit $0 < \Theta < 1$, $\Theta = [0, a_1, a_2, a_3, \ldots]$ und p_k, q_k wie im Satz 2.17.

Satz 2.22 ([1, 25])
Jede positive natürliche Zahl n kann eindeutig in der Form $rq_k + q_{k-1} + s$ mit $k \geq 1$, $1 \leq r \leq a_k$ und $0 \leq s < q_k$ dargestellt werden.

Beweis:
Wegen $a_k > 0$ für $k > 0$ ist $(q_k)_{k\in\mathbb{N}} = (0, 1, a_1, a_2a_1 + 1, \ldots)$ streng monoton wachsend für $k > 1$. Damit ist auch die Folge $(q_k + q_{k+1})_{k\in\mathbb{N}} = (1, a_1 + 1, a_2a_1 + a_1 + 1, \ldots)$ streng monoton wachsend. Damit gilt:

$$\forall n \geq 1 : \exists^* k \geq 1 : q_{k+1} + q_k > n \geq q_k + q_{k-1}$$

Innerhalb dieser Grenzen liegen $q_{k+1} - q_{k-1} = a_kq_k$ natürliche Zahlen. Da offensichtlich jede zulässige Wahl für r und s eine andere Zahl innerhalb dieser Grenzen erzeugt und es genau a_kq_k Möglichkeiten gibt, r und s zu wählen, wird jede dieser Zahlen auf genau eine Art dargestellt.

■

Satz 2.23 ([1, 25])
Für alle r mit $0 \leq r \leq a_k$ und $k \geq 2$ gilt:

$$\left\{(-1)^k(rq_k + q_{k-1})\Theta\right\} = (-1)^k\left(r(q_k\Theta - p_k) + q_{k-1}\Theta - p_{k-1}\right)$$

Beweis: (mit vollständiger Induktion)

$$\underline{r = 0:}\quad \underline{k \text{ gerade:}}\quad \{q_{k-1}\Theta\} = \{\underbrace{q_{k-1}\Theta - p_{k-1}}_{>0}\} = q_{k-1}\Theta - p_{k-1}$$

Denn für $k=2$ gilt $\{\Theta\}<1$, ansonsten steht $\{q_{k-1}\Theta - p_{k-1}\}\geq 1$ im Widerspruch dazu, daß $\frac{p_{k-1}}{q_{k-1}}$ beste Approximation ist.

<u>k ungerade:</u> $\{-q_{k-1}\Theta\} = \{-(\underbrace{q_{k-1}\Theta - p_{k-1}}_{<0,>-1})\} = -(q_{k-1}\Theta - p_{k-1})$

<u>$r-1\to r$:</u> $\big\{(-1)^k q_k\Theta\big\} = \big\{-(-1)^{k+1}q_k\Theta\big\} = 1-\big\{(-1)^{k+1}q_k\Theta\big\}$

$= 1-(-1)^{k+1}(q_k\Theta - p_k) = 1+\underbrace{(-1)^k(q_k\Theta - p_k)}_{<0}$

$|q_k\Theta - p_k| < \Big|\Big(\underbrace{(r-1)q_k + q_{k-1}}_{<q_{k+1}}\Big)\Theta - \big(p_k(r-1)+p_{k-1}\big)\Big|$ da $\frac{p_k}{q_k}$ beste Approx.

$\big\{(-1)^k(rq_k + q_{k-1})\Theta\big\} = \big\{(-1)^k\big((r-1)q_k+q_{k-1}\big)\Theta + (-1)^k q_k\Theta\big\} =$

$\big\{\underbrace{(-1)^k\big((r-1)(q_k\Theta - p_k) + q_{k-1}\Theta - p_{k-1}\big) + 1 + (-1)^k(q_k\Theta - p_k)}_{>1,<2}\big\}$

$= (-1)^k\big(r(q_k\Theta - p_k) + q_{k-1}\Theta - p_{k-1}\big)$

■

Satz 2.24 (Three Distance Theorem [1, 25, 42])
Fügt man die Punkte $\{\Theta\},\{2\Theta\},\{3\Theta\},\ldots,\{n\Theta\}$ in der angegebenen Reihenfolge in das Intervall $[0,1]$ ein, so weisen die $n+1$ entstehenden Teilintervalle höchstens drei verschiedene Längen auf.

Beweis:
Von besonderem Interesse sind jene Werte für n, für die $\{n\Theta\} = \min_{1\leq m\leq n}\{m\Theta\}$ oder $\{n\Theta\} = \max_{1\leq m\leq n}\{m\Theta\}$ gilt. Diese werden von nun an min- bzw. max-Werte genannt.
Für <u>gerades $k\geq 2$</u> gilt:

$$\big\{(rq_k + q_{k-1})\Theta\big\} = r\underbrace{(q_k\Theta - p_k)}_{<0} + \underbrace{q_{k-1}\Theta - p_{k-1}}_{>0} > 0$$

Betrachtet man die Zahlen n für die $r=0$ gilt, erhält man: $\{q_1\Theta\}>\{q_3\Theta\}>\{q_5\Theta\},\ldots$
Erhöht man nun r um eins (wobei $r+1\leq a_k$) so ist wegen $q_k\Theta - p_k<0$:

$$\big\{\big((r+1)q_k + q_{k-1}\big)\Theta\big\} < \big\{(rq_k+q_{k-1})\Theta\big\}$$

Verwendet man nun $a_kq_k + q_{k-1} = q_{k+1}$ und obige Kette, so erhält man:

$$\{q_1\Theta\} > \big\{(q_2+q_1)\Theta\big\} > \big\{(2q_2+q_1)\Theta\big\} > \ldots > \big\{(a_2q_2+q_1)\Theta\big\} = \{q_3\Theta\} > \{q_4+q_3\Theta\},\ldots$$

Für dazwischen liegende Werte (d.h. $s<q_k$) von n tritt kein neuer min-Wert auf, denn für $a_1=1$ ist $q_1=q_2$, weshalb es kein zulässiges s gibt. Sonst gilt:

$$\big\{(rq_k+q_{k-1}+s)\Theta\big\} = \big\{r(q_k\Theta - p_k) + q_{k-1}\Theta - p_{k-1} + s\Theta\big\} > r(q_k\Theta - p_k) + q_{k-1}\Theta - p_{k-1}$$

Denn für $s\neq q_k - q_{k-1}$ ist

$$\big\{(q_{k-1}\Theta - p_{k-1} + s\Theta\big\} = \big\{(q_{k-1}+s)\Theta\big\} = \big|(q_{k-1}+s)\Theta - \big\lfloor(q_{k-1}+s)\Theta\big\rceil\big| > q_{k-1}\Theta - p_{k-1}$$

auf Grund der Bemerkung nach Satz 2.21. Für $s = q_k - q_{k-1}$ ist

$$\big\{(q_{k-1}+s)\Theta\big\} = \{q_k\Theta\} = 1-\{-q_k\Theta\} = 1+(q_k\Theta - p_k) > \frac{1}{2} > q_{k-1}\Theta - p_{k-1}.$$

Für <u>ungerades k</u> gilt:

$$\big\{-(rq_k+q_{k-1})\Theta\big\} = -r\underbrace{(q_k\Theta-p_k)}_{>0}-\underbrace{(q_{k-1}\Theta-p_{k-1})}_{<0}>0$$

Wenn $\{-n\Theta\}=1-\{n\Theta\}$ minimal ist, ist damit $\{n\Theta\}$ maximal. Analog zu oben folgt damit:

$$\{q_0\Theta\}<\big\{(q_1+q_0)\Theta\big\}<\big\{(2q_1+q_0)\Theta\big\}<\ldots<\big\{(a_1q_1+q_0)\Theta\big\}=\{q_2\Theta\}<\{q_3+q_2\Theta\},\ldots$$

und für dazwischen liegende Werte für n tritt kein neuer max-Wert auf. Zusammengefasst erhält man also folgende min- und max- Werte (der Größe nach sortiert):

$$\overbrace{q_1+q_0=0q_2+q_1}^{\text{min, max}},\overbrace{2q_1+q_0,\ldots,a_1q_1+q_0=q_2}^{\text{max}},\overbrace{q_2+q_1,\ldots,a_2q_2+q_1=q_3}^{\text{min}},$$
$$\underbrace{q_3+q_2,\ldots,a_3q_3+q_2=q_4}_{\text{max}},\underbrace{q_4+q_3,\ldots,a_4q_4+q3=q5}_{\text{min}},\underbrace{q5+q4,\ldots}_{\text{max}}$$

Der erste Wert $\{\Theta\}$ unterteilt das Intervall $[0,1]$ in zwei Teile. Tritt nun ein neuer min-Wert $t=rq_k+q_{k-1}$ (k gerade, $r\geq 1$) auf, so entsteht erstmals ein Intervall der Länge $\{t\Theta\}$. Dabei wird ein Intervall der Länge $\big\{((r-1)q_k+q_{k-1})\Theta\big\}$ unterteilt. Der zweite dadurch entstehende Teil hat die Länge $\{-q_k\Theta\}$ (q_k ist der aktuelle max-Wert), denn:

$$\big\{\big((r-1)q_k+q_{k-1}\big)\Theta\big\}=(r-1)(q_k\Theta-p_k)+q_{k-1}\Theta-p_{k-1}=$$
$$r(q_k\Theta-p_k)+q_{k-1}\Theta-p_{k-1}-(q_k\Theta-p_k)=\big\{(rq_k+q_{k-1})\Theta\big\}+\big\{-(0q_{k+1}+q_k)\Theta\big\}$$

Erhöht man nun s schrittweise um eins, wobei $s<q_k$ gelten soll, entstehen wieder ein Intervall der Länge $\{t\Theta\}$ und ein Intervall der Länge $\{-q_k\Theta\}$. Insgesamt entstehen so q_k Intervalle der Länge $\{t\Theta\}$. Für $r<a_k$ tritt anschließend wieder ein neuer min-Wert auf und die q_k Intervalle der Länge $\{t\Theta\}$ werden in q_k Intervalle der Länge $\{((r+1)q_k+q_{k-1})\Theta\}$ und q_k Intervalle der Länge $\{-q_k\Theta\}$ aufgeteilt. Für $r=a_k$ ist $t=q_{k+1}$, deshalb folgt als nächstes ein max-Wert. Durch diesen werden alle $a_kq_k+q_{k-1}=q_{k+1}$ Intervalle der Länge $\{-q_k\Theta\}$ unterteilt, wodurch q_{k+1} zusätzliche Intervalle der Länge $\{t\Theta\}$ entstehen. Der nächste min-Wert ist $q_{k+2}+q_{k+1}$ und dieser findet $q_k+a_{k+1}q_{k+1}=q_{k+2}$ Intervalle der Länge $\{t\Theta\}$ vor, usw.
Es treten damit zu jedem Zeitpunkt höchstens drei verschiedene Intervalllängen auf. Weiters wird durch einen neu eingefügten Punkt stets eines der Intervalle mit der größten vorhandenen Länge unterteilt. ■

Eine Hashfunktion soll die Eingangsdaten möglichst gleichmäßig auf die vorhanden Speicherplätze aufteilen. Die Aufteilung eines Intervalls in zwei Teilintervalle sollte deswegen immer zwei möglichst gleich große Teile erzeugen. Deshalb wird eine Unterteilung nun als **„schlecht“** bezeichnet, wenn eines der beiden entstehenden Intervalle mehr als doppelt so lange wie das andere ist.

Satz 2.25 ([25])
Die einzigen irrationalen Zahlen zwischen Null und Eins, die keine „schlechten“ Unterteilungen erzeugen, sind $\phi^{-1}=\frac{\sqrt{5}-1}{2}$ *und* $\phi^{-2}=\frac{3-\sqrt{5}}{2}$.

Beweis:
Die Zahl $\phi=\frac{1+\sqrt{5}}{2}$ besitzt die Kettenbruchentwicklung $[1,1,1,\ldots]$, wie mit der Gleichung $\phi=1+\frac{1}{\phi}$ leicht überprüft werden kann. Daraus erhält man $\phi^{-1}=[0,1,1,1,\ldots]$ und $\phi^{-2}=[0,2,1,1,1,\ldots]$.
Wegen $\{\Theta\}=\Theta<\frac{1}{a_1}$ folgt, dass für $a_1>2$ die erste Unterteilung schlecht ist. Für die bei

einer Unterteilung entstehenden Intervalllängen sind die folgenden Formeln aus dem Beweis des letzten Satzes bekannt:

$$\begin{aligned}
L_1 &= \big\{(-1)^k(rq_k + q_{k-1})\Theta\big\} = (-1)^k\big(r(q_k\Theta - p_k) + q_{k-1}\Theta - p_{k-1}\big)\\
L_2 &= \big\{(-1)^{k+1}(0q_{k+1} + q_k)\Theta\big\} = (-1)^{k+1}(q_k\Theta - p_k)\\
\frac{L_1}{L_2} &= -r - \frac{q_{k-1}\Theta - p_{k-1}}{q_k\Theta - p_k} = -r - \frac{\Theta_k q_k + q_{k-1}}{(-1)^{k+1}} \frac{(-1)^k\Theta_k}{\Theta_k q_k + q_{k-1}}\\
&= \Theta_k - r = a_k - r + \frac{1}{[a_{k+1}, a_{k+2}, a_{k+3}, \ldots]}
\end{aligned}$$

Für $r = a_k$ ist $\frac{L_1}{L_2} = \frac{1}{[a_{k+1},a_{k+2},a_{k+3},\ldots]} < \frac{1}{a_{k+1}} \leq \frac{1}{2}$ wenn $a_{k+1} \geq 2$. Deswegen muss $a_k = 1$ für alle $k \geq 1$ sein, um schlechte Aufteilungen zu verhindern. ■

Kapitel 3

Hashing mit Verkettung

Wie bereits erwähnt, wird bei diesen Verfahren für jede Tabellenadresse eine verkettete Liste erzeugt, die all jene Datensätze aufnimmt, deren Schlüssel auf diese eine Tabellenadresse abbilden. Der erste Schlüssel kann nun entweder in der Tabelle selbst gespeichert werden, oder die Tabelle kann so gestaltet werden, dass sie selbst keine Daten, sondern nur die Adresse des ersten Listenelements aufnimmt. Man spricht im zweiten Fall von **separater** Verkettung, ansonsten von **direkter** Verkettung. Direkte Verkettung hat den Nachteil, dass auch jeder leere Tabelleneintrag Speicherplatz verbraucht, gleichzeitig wird aber weniger Speicherplatz für Adressen verbraucht. Die Unterschiede zwischen diesen beiden Verfahren sind aber so gering, dass sie zusammen behandelt werden können.

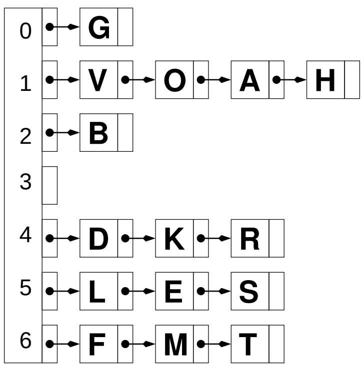
3.1 Gewöhnliches Hashing mit Verkettung

Die einfachste Möglichkeit einen neuen Schlüssel in eine bestehende Kette einzuhängen, besteht darin, ihn am Beginn der Liste als neues erstes (bei direkter Verkettung eventuell als zweites) Listenelement einzutragen. Das folgende einfache Beispiel erklärt den Ablauf:

Beispiel 9 (separate Verkettung) *Als Universum U werden die Großbuchstaben von A bis Z herangezogen, wobei als Schlüssel eines Buchstabens die durch die Reihenfolge im Alphabet zugeordnete Zahl verwendet wird. Weiters sei $m = 7$ und $h(S) := S \mod m$. In eine anfangs leere Liste wird nun die angeführte Folge eingetragen, wobei separate Verkettung verwendet wird.*

Information	H	S	A	O	G	T	E	R	V	L	B	K	M	F	D
Schlüssel	8	19	1	15	7	20	5	18	22	12	2	11	13	6	4
Hashwert	1	5	1	1	0	6	5	4	1	5	2	4	6	6	4

Dabei entsteht folgende Struktur: (Zur besseren Übersicht werden dabei Schlüssel nicht angeführt.)



Die Hashtabelle wird nun mit `T` und der Pointer auf den Beginn der i-ten Liste mit `T[i]` bezeichnet. Jedes Listenelement `z` soll nun eine Struktur sein, wobei `z.key` den Schlüssel und `z.info` die Information des abgespeicherten Elements aufnimmt, sowie `z.next` ein Pointer auf das nächste freie Listenelement ist. Der Nullpointer wird mit `NIL` bezeichnet.
Der Pseudocode in C++ ähnlicher Notation zur Beschreibung von Funktionen, die Einfüge-, Such- bzw. Löschoperationen in einer Hashtabelle mit separater Verkettung durchführen können, kann z.B. so ausschauen:

Algorithmus A: `search(S)`
Suche des Schlüssels `S` in einer Tabelle mit separater Verkettung

```
p = T[h(S)]                      // p: Pointer auf aktuelles Listenelement
q = NIL                          // q: Pointer auf Vorgänger von p
while (p != NIL)
    if (p->key == S)
        return(p,q,true)
    q = p
    p = p->next
return(p,q,false)
```

Algorithmus B: `insert(S,I)`
Einfügen des Eintrages `S,I` in einer Tabelle mit separater Verkettung

```
(p,q,statuts) = search(S)        // Eingefügt wird nur, wenn die Suche erfolglos ist
if (status == false)
    z = new Element              // Speicher wird allokiert
    z->info = I
    z->key = S
    z->next = T[h(S)]
    T[h(S)] = z
```

Algorithmus C: `delete(S)`
Löschen des Schlüssels `S` in einer Tabelle mit separater Verkettung

```
(p,q,status) = search(S)
if (status == true)
    if (q != NIL)
        q->next = p->next        // Element wird in Liste übersprungen
    else
        T[h(S)] = p->next        // es wird vom Listenanfang gelöscht
free(p)                          // Speicher wird freigegeben
```

Wird vor dem Einfügen eines neuen Elements nicht überprüft, ob das Element bereits vorhanden ist, kann eine Einfügeoperation in konstanter Zeit erfolgen. Der daraus entstehende Nachteil, dass dann mehrfach vorkommende Elemente auch mehrfach eingetragen werden, wiegt jedoch im Allgemeinen schwerer, da dadurch unnötig Speicherplatz verschwendet wird. Deshalb wird für die folgende Analyse angenommen, dass Elemente nicht mehrfach eingetragen werden.

Analyse

Für die gesuchten Erwartungswerte, in einer Tabelle mit momentan n Einträgen, wird in dieser Arbeit folgende Notation verwendet:

$\mathbb{E}C_n^{[l]}$ ... durchschnittliche Anzahl der notwendigen Vergleiche für erfolg**l**ose Suche
$\mathbb{E}C_n^{[r]}$ ... durchschnittliche Anzahl der notwendigen Vergleiche für erfolg**r**eiche Suche

Satz 3.1 (Hashing mit separater Verkettung [25])

$$\mathbb{E}C_n^{[r]} = 1 + \frac{n-1}{2m} \sim 1 + \frac{\alpha}{2}$$
$$\mathbb{E}C_n^{[l]} = \frac{n}{m} + \Big(1 - \frac{1}{m}\Big)^n \sim \alpha + e^{-\alpha}$$

Beweis:
P_{nk} ... Wahrscheinlichkeit, dass eine Liste die Länge k hat, wenn insgesamt n Elemente in die Tabelle eingetragen wurden

$$P_{nk} = \binom{n}{k}\frac{1}{m^k}\Big(1-\frac{1}{m}\Big)^{n-k} = \binom{n}{k}\frac{1}{m^n}(m-1)^{n-k}$$
$$P_n(z) := \sum_{k\ge 0} P_{nk}z^k = \sum_{k\ge 0}\binom{n}{k}\frac{1}{m^k}\Big(1-\frac{1}{m}\Big)^{n-k}z^k = \Big(\frac{z}{m}+1-\frac{1}{m}\Big)^n = \Big(1+\frac{z-1}{m}\Big)^n$$

Es wird nun davon ausgegangen, dass eine nicht erfolgreiche Suche in einer Kette der Länge $k > 0$ genau k Vergleiche und in einer leeren Kette einen Vergleich benötigt[1]. Damit gilt $\mathbb{P}\big(C_n^{[l]} = k\big) = \mathbb{P}\big(B\big(n, \frac{1}{m}\big) = k\big) + (\delta_{k,1} - \delta_{k,0})\mathbb{P}\big(B\big(n, \frac{1}{m}\big) = 0\big)$, wenn $B(n,p)$ eine Binomialverteilung mit den Parametern n und p bezeichnet.

$$\begin{aligned}\mathbb{E}C_n^{[l]} &= \sum_{k\ge 0}(k+\delta_{k0})P_{nk} = \sum_{k\ge 0} kP_{nk} + P_{n0} = P_n'(1) + P_n(0) = n\Big(1+\frac{z-1}{m}\Big)^{n-1}\frac{1}{m}\Big|_{z=1} + P_{n0}\\ &= \frac{n}{m} + \Big(1-\frac{1}{m}\Big)^n = \alpha + \Bigg(\underbrace{\Big(1-\frac{1}{m}\Big)^{-m}}_{\to e}\Bigg)^{\frac{-n}{m}} \sim \alpha + e^{-\alpha}\end{aligned}$$

Um $\mathbb{E}C_n^{[r]}$ zu bestimmen, wird die Anzahl von Vergleichen berechnet, die benötigt werden, um alle in der Tabelle vorhandenen Schlüssel zu suchen. Eine Liste der Länge k trägt zu dieser Summe $\sum_{j=1}^k j = \frac{k(k+1)}{2} = \binom{k+1}{2}$ bei.

$$\begin{aligned}\mathbb{E}C_n^{[r]} &= \frac{1}{m^n}\frac{1}{n}\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}\left(\binom{k_1+1}{2}+\ldots+\binom{k_m+1}{2}\right)\\ &= \frac{1}{nm^n}m\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}\binom{k_1+1}{2} = \frac{1}{nm^n}m\sum_{k_1=0}^{n}\binom{n}{k_1}(m-1)^{n-k_1}\binom{k_1+1}{2}\\ &= \frac{m}{n}\sum_{k=0}^{n}\binom{k+1}{2}\binom{n}{k}\frac{(m-1)^{n-k}}{m^n} = \frac{m}{n}\sum_{k\ge 0}\binom{k+1}{2}P_{nk} = \frac{m}{2n}\sum_{k\ge 0}k(k-1+2)P_{nk}\\ &= \frac{m}{2n}\Big(\sum_{k\ge 2}k(k-1)P_{nk}z^{k-2} + 2\sum_{k\ge 1}kP_{nk}z^{k-1}\Big)\Big|_{z=1}\\ &= \frac{m}{2n}\big(P_n''(1)+2P_n'(1)\big) = \frac{m}{2n}\left(n(n-1)\Big(1+\frac{z-1}{m}\Big)^{n-2}\frac{1}{m^2}\Big|_{z=1} + 2\frac{n}{m}\right)\\ &= \frac{m}{2n}\Big(\frac{n(n-1)}{m^2}+2\frac{n}{m}\Big) = 1 + \frac{n-1}{2m} \sim 1 + \frac{\alpha}{2}\end{aligned}$$

■

[1]Diese Annahme scheint gerechtfertigt, um die Unterschiede im Vergleich zu Hashing mit direkter Verkettung zu berücksichtigen vgl. [25].

Satz 3.2 (Hashing mit separater Verkettung [21])
Mit den oben verwendeten Bezeichnungen gilt weiters:

$$\begin{aligned}
\mathbb{V}C_n^{[r]} &= \frac{(n-1)(n-5)}{12m^2} + \frac{n-1}{2m} \sim \frac{\alpha^2}{12} + \frac{\alpha}{2} \\
\mathbb{V}C_n^{[l]} &= \frac{n(m-1)}{m^2} + \Big(1-\frac{1}{m}\Big)^n\Big(1-2\frac{n}{m} - \Big(1-\frac{1}{m}\Big)^n\Big) \sim \alpha + e^{-\alpha}(1-2\alpha-e^{-\alpha})
\end{aligned}$$

Beweis:

$$\begin{aligned}
\mathbb{E}(C_n^{[l]})^2 &= \sum_{k\ge 0}(k+\delta_{k0})^2 P_{nk} = \sum_{k\ge 0}(k^2+\delta_{k0})P_{nk} = P_n(0) + P_n''(1) + P_n'(1) \\
&= \Big(1-\frac{1}{m}\Big)^n + n(n-1)\frac{1}{m^2} + \frac{n}{m} \\
(\mathbb{E}C_n^{[l]})^2 &= \Big(\frac{n}{m} + \Big(1-\frac{1}{m}\Big)^n\Big)^2 \\
\mathbb{V}C_n^{[l]} &= \mathbb{E}(C_n^{[l]})^2 - (\mathbb{E}C_n^{[l]})^2 \\
&= \Big(1-\frac{1}{m}\Big)^n + \frac{n(n-1)}{m^2} + \frac{n}{m} - \frac{n^2}{m^2} - 2\frac{n}{m}\Big(1-\frac{1}{m}\Big)^n - \Big(1-\frac{1}{m}\Big)^{2n} \\
&= \frac{(n-1)n + nm - n^2}{m^2} + \Big(1-\frac{1}{m}\Big)^n\Big(1-2\frac{n}{m} - \Big(1-\frac{1}{m}\Big)^n\Big) \\
&= \frac{n(m-1)}{m^2} + \Big(1-\frac{1}{m}\Big)^n\Big(1-2\frac{n}{m} - \Big(1-\frac{1}{m}\Big)^n\Big) \sim \alpha + e^{-\alpha}(1-2\alpha-e^{-\alpha})
\end{aligned}$$

Ähnlich wie im Beweis des letzten Satzes verwendet man nun wieder bedingte Erwartungswerte und den Satz von der vollständigen Erwartung. Die Summe der Quadrate der ersten $k-1$ positiven Zahlen wird mit Satz 2.11 berechnet.

$$\begin{aligned}
\mathbb{E}(C_n^{[r]}-1)^2 &= \frac{1}{m^n}\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}\frac{1}{n}\Big(\sum_{j=1}^{k_1-1} j^2 + \sum_{j=1}^{k_2-1} j^2 + \ldots + \sum_{j=1}^{k_m-1} j^2\Big) \\
&= \frac{m}{m^n n}\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}\frac{k_1(k_1-1)(2k_1-1)}{6} \\
&= \frac{m}{6m^n n}\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}k_1(k_1-1)2\Big(k_1-2+\frac{3}{2}\Big)
\end{aligned}$$

Verwendet man nun $\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}{x_1}^{k_1}\ldots{x_m}^{k_m} = (x_1+\ldots+x_m)^n$ (Multinomialtheorem) und $k^{\underline{h}} = D^h x^k\big|_{x=1}$ wobei D den Differentialoperator bezüglich x bezeichnet, so ergibt sich:

$$\begin{aligned}
S(h) :=& \sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}k_1^{\underline{h}} = D^h\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}x^{k_1}\Big|_{x=1} \\
&= D^h(x+m-1)^n\Big|_{x=1} = n^{\underline{h}}(x+m-1)^{n-h}\Big|_{x=1} = \frac{n^{\underline{h}}m^n}{m^h} \\
\mathbb{E}(C_n^{[r]}-1)^2 &= \frac{m}{3m^n n}\sum_{k_1+\ldots+k_m=n}\binom{n}{k_1,\ldots,k_m}\Big(k_1(k_1-1)(k_1-2) + \frac{3}{2}k_1(k_1-1)\Big) \\
&= \frac{m}{3m^n n}\Big(S(3) + \frac{3}{2}S(2)\Big) = \frac{m}{3m^n n}\Big(\frac{n(n-1)(n-2)m^n}{m^3} + \frac{3}{2}\frac{n(n-1)m^n}{m^2}\Big)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{6m^2}(2n^2 - 6n + 4 + 3mn - 3m) \\
\mathbb{V}C_n^{[r]} &= \mathbb{E}(C_n^{[r]} - 1)^2 - (\mathbb{E}C_n^{[r]} - 1)^2 = \frac{2n^2 - 6n + 4 + 3mn - 3m}{6m^2} - \frac{(n-1)^2}{4m^2} \\
&= \frac{(n-1)(n-5)}{12m^2} + \frac{n-1}{2m} \sim \frac{\alpha^2}{12} + \frac{\alpha}{2}
\end{aligned}
$$

■

Satz 3.3 (Hashing mit direkter Verkettung [21])
Die Resultate bezüglich erfolgreicher Suche stimmen mit denen von Hashing mit separater Verkettung überein. Für erfolglose Suche gilt:

$$
\mathbb{E}C_n^{[l]} = \frac{n}{m} = \alpha \qquad \mathbb{V}C_n^{[l]} = \frac{n(m-1)}{m^2} \sim \alpha
$$

Beweis:
Der einzige Unterschied zwischen den beiden Verfahren besteht darin, dass jetzt eine erfolglose Suche in einer leeren Liste keinen Vergleich benötigt. Lässt man die entsprechenden Ausdrücke wegfallen, erhält man obige Resultate. ■

Satz 3.4 (Erwartete Anzahl nichtleerer Listen)
Der Erwartungswert für die Anzahl der Listen, die mindestens einen Eintrag besitzen, beträgt

$$
\mathbb{E}B_n := m\Big(1 - \Big(1 - \frac{1}{m}\Big)^n\Big) \sim m(1 - e^{-\alpha}).
$$

Beweis:
Im Beweis zu Satz 3.1 wurde bereits $P_{n0} = \big(1 - \frac{1}{m}\big)^n$ für die Wahrscheinlichkeit, dass eine Liste leer ist, ermittelt. Folgende Hilfsgröße erweist sich als nützlich:

$$
L_i := \begin{cases} 1 & \text{falls die } i\text{-te Liste leer ist,} \\ 0 & \text{sonst} \end{cases}
$$

$$
\mathbb{E}B_n = \mathbb{E}\Big(m - \sum_{i=1}^{m} L_i\Big) = m - \sum_{i=1}^{m} \mathbb{E}L_i = m - m\Big(1 - \frac{1}{m}\Big)^n
$$

■

3.2 Verbesserungen des Standardverfahrens

Sortiertes Einfügen

Der oben beschriebene Algorithmus hat den Nachteil, dass die Suche nach einem Element erst dann als erfolglos abgebrochen wird, wenn die gesamte Liste durchsucht wurde. Folgende Variante des ursprünglichen Algorithmus fügt die Elemente nach Schlüsseln sortiert ein. Dabei soll o.B.d.A. aufsteigend sortiert werden. Ist ein Element noch nicht in der Liste vorhanden, so liefert die erfolglose Suche nach ihm gleichzeitig den Platz in der Liste, an dem das Element eingefügt werden muss. Dadurch sinkt der durchschnittliche Aufwand, den eine erfolglose Suchoperation benötigt, allerdings auf Kosten der Einfügeoperation.

Algorithmus D:	`search_sorted(S)` Suche des Schlüssels `S` in einer Tabelle mit separater Verkettung und sortierten Listen

```
p = T[h(S)]                          // p: Pointer auf aktuelles Listenelement
q = NIL                              // q: Pointer auf Vorgänger von p
while (p != NIL)
    if (p->key = S)
        return (p,q,true)
    if (p->key > S)
        return (p,q,false)           // S kann nicht mehr in Liste enthalten sein
    q = p
    p = p->next
return (p,q,false)
```

Dieser Suchalgorithmus kann natürlich auch so konzipiert werden, dass auf jeden benötigten Schlüssel nur einmal zugegriffen werden muss, z.B. indem man die Differenz zwischen dem aktuellen und dem gesuchten Schlüssel ermittelt. Die tatsächlichen Vergleiche könnten dann durch Überprüfung des Vorzeichenbits bzw. des Zeroflags erfolgen, wofür der Aufwand vernachlässigbar ist.

Algorithmus E:	`insert_sorted` Sortiertes Einfügen des Eintrages `S,I` in einer Tabelle mit separater Verkettung

```
(p,q,status) = search_sorted(S)
if (status == false)
    z = new Element
    z->info = I
    z->key = S
    z->next = p
    if (q != NIL)
        q.next = z                   // es wurde nicht am Listenanfang einfügt
    else
        T[h(S)] = z                  // es wurde am Listenanfang eingefügt
```

Analyse

Satz 3.5 (sortierte Ketten [21])
Die Ergebnisse für erfolgreiche Suchoperationen bleiben unverändert. Für erfolglose Suche gilt:

$$\mathbb{E}C_n^{[l]} = \begin{cases} 1 + \frac{n}{2m} - \frac{m}{n+1}\left(1 - \left(1 - \frac{1}{m}\right)^{n+1}\right) + \left(1 - \frac{1}{m}\right)^n & \\ \qquad \sim 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha}) + e^{-\alpha} & \textit{für separate Verkettung} \\ 1 + \frac{n}{2m} - \frac{m}{n+1}\left(1 - \left(1 - \frac{1}{m}\right)^{n+1}\right) & \\ \qquad \sim 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha}) & \textit{für direkte Verkettung} \end{cases}$$

Beweis:
Die durchschnittliche Anzahl der Vergleiche für eine erfolgreiche Suchoperation hängt nicht von der Reihenfolge ab, in der die Elemente innerhalb der einzelnen Listen angeordnet sind. Deshalb bleiben die zuvor ermittelten Resultate für die erfolgreiche Suche weiterhin gültig.
Bei einer erfolglosen Suche ist damit zu rechnen, dass sich die Anzahl der benötigten Vergleiche etwa halbiert. Bei einer sortierten Kette der Länge k gibt es $k+1$ Möglichkeiten, das zu suchende Element einzuordnen, denn es kann kleiner als alle Elemente der Kette, kleiner als genau ein Element der Kette, ... sein. Dabei wird angenommen, dass jeder dieser Fälle gleichwahrscheinlich

ist. Es ist sichergestellt, dass das Element nicht in der Kette ist, wenn ein größeres Element gefunden wurde, oder die Kette vollständig abgesucht wurde. Demnach werden durchschnittlich

$$\frac{1+2+3+\ldots+k+k}{k+1} = \frac{\frac{1}{2}(k+1)(k+2)-1}{k+1} = 1 + \frac{k}{2} - \frac{1}{k+1}$$

Vergleiche benötigt, um festzustellen, dass das Element nicht in der Kette enthalten ist. Für direkte Verkettung stimmt diese Formel auch für eine leere Kette, bei separater Verkettung wird jedoch für eine leere Kette ein zusätzlicher Vergleich benötigt, weshalb in diesem Fall der Ausdruck $\delta_{k,0}$ addiert wird. Der einzige neue Ausdruck, der hier auftritt ist

$$\begin{aligned}\sum_{k\geq 0} \frac{1}{k+1} P_{nk} &= \sum_{k\geq 0} \int_0^1 P_{nk} z^k dz = \int_0^1 P_n(z) dz = \int_0^1 \Big(1 + \frac{z-1}{m}\Big)^n \\ &= \frac{m}{n+1}\Big(1 + \frac{z-1}{m}\Big)^{n+1}\Big|_0^1 = \frac{m}{n+1}\Big(1 - \Big(1 - \frac{1}{m}\Big)^{n+1}\Big).\end{aligned}$$

Summiert man die erhaltenen Ergebnisse, so erhält man obige Ausdrücke. ■

Verwendung abgekürzter Schlüssel

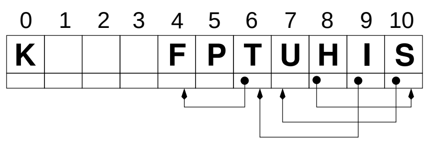
Alle Verfahren, die auf Hashing mit Verkettung beruhen, haben im Vergleich zu den anschließend behandelten Verfahren mit offener Adressierung den Nachteil, dass sie zusätzlichen Speicherplatz für Adressen verbrauchen. Dies kann eventuell dadurch wieder ausgeglichen werden, dass nicht der gesamte Schlüssel in der Hashtabelle gespeichert werden muss [25], sondern nur ein Teil des Schlüssels. Dabei muss die Abkürzung so gewählt sein, dass sie zusammen mit dem Hashwert den Schlüssel eindeutig bestimmt. Verwendet man die Hashfunktion $h(S) \mod m$, so ist $\lfloor \frac{S}{m} \rfloor$ eine geeignete Abkürzung. Wieviel Speicherplatz tatsächlich eingespart werden kann, hängt auch vom jeweils verwendeten Computersystem ab, da üblicherweise die kleinste adressierbare Speichereinheit ein Byte beträgt.

3.3 Coalesced Hashing

Das „gewöhnliche" Hashverfahren mit Verkettung hat den Nachteil, dass für jeden Schlüssel zusätzlicher Speicherplatz belegt werden muss. Beim Coalesced Hashing werden die Schlüssel zwar weiterhin verkettet, es wird jedoch kein zusätzlicher Speicherplatz belegt. Solange der Platz, auf den die Hashfunktion ein Element abbildet, nicht belegt ist, wird es direkt an dieser Stelle in die Tabelle eingetragen. Die „Überläufer" werden statt in neu allokiertem Speicher an jenem freien Tabellenplatz eingetragen, der die größte Hashadresse aller noch freien Plätze besitzt. Folgendes Beispiel verdeutlicht den Ablauf:

Beispiel 10 *Wie im Beispiel 9 sollen folgende Buchstaben in der angegebenen Reihenfolge in die Hashtabelle eingetragen werden, mit dem Unterschied dass jetzt $m = 11$ ist.*

Information	H	K	S	I	P	U	T	F
Schlüssel	8	11	19	9	16	21	20	6
Hashwert	8	0	8	9	5	10	9	6



In diesem Beispiel kann man auch schon das „Verschmelzen" (coalesce) der Überläuferketten erkennen: Der Buchstabe U befindet sich in der Kette, die beim Hashwert 8 startet, obwohl $h(U) \neq 8$ ist.

Such- Einfüge- und Löschoperationen können mit den folgenden Pseudocodes realisiert werden. Bereits vorhandene Elemente werden bei dieser Implementierung nicht noch einmal eingefügt. Die dabei verwendete Funktion `findempty()` sucht den freien Platz mit der größten Hashadresse, wobei bei einer realen Umsetzung noch geprüft werden muss, ob auch wirklich ein freier Platz gefunden werden konnte. Bei geschickter Umsetzung müssen, zumindest für den Fall, dass nicht besonders viele Löschoperationen erfolgt sind, nur wenige Plätze überprüft werden. Deshalb wird der Aufwand für diese Funktion vernachlässigt.

Algorithmus F: `search(S)`
Suche in Tabelle mit Coalesced Hashing

```
p = T[h(S)]                            // p: Pointer auf aktuelles Listenelement
q = NIL                                // q: Pointer auf Vorgänger von p
do
    if (p->key == S)
        return (p,q,true)
    q = p
    p = p->next
while (p != NIL)
return (p,q,false)
```

Algorithmus G: `insert(S,I)`
Einfügen in eine Tabelle mit Coalesced Hashing

```
(p,q,status) = search(S)
if (status == false)
    q = findempty()                    // liefert Pointer auf freien Platz
    q->info = I
    q->key = S
    p->next = q
```

Algorithmus H: `delete(S)`
Löschen in einer Tabelle mit Coalesced Hashing

```
(p,q,status) = search(S)
if (status == true)
    if (q != NIL)
        q->next = p->next              // zu löschendes Element nicht am Listenanfang
        setempty(p)                    // markiere Platz als leer
    else
        if (p->next != NIL)
            T[h(S)] = p->next
            setempty(p.next)
        else                           // Liste bestand nur aus einem Element
            setempty(p)
```

Coalesced Hashing mit Keller

Wenn viele Ketten miteinander verschmelzen, steigt dadurch der Aufwand für einen Suchvorgang stark an. Um dem vorzubeugen, kann zusätzlicher Speicherplatz am Ende der Tabelle reserviert werden, der dann ausschließlich dazu verwendet wird, Überläufer zu speichern. Dieses Verfahren wurde erstmals in [44] genau analysiert.

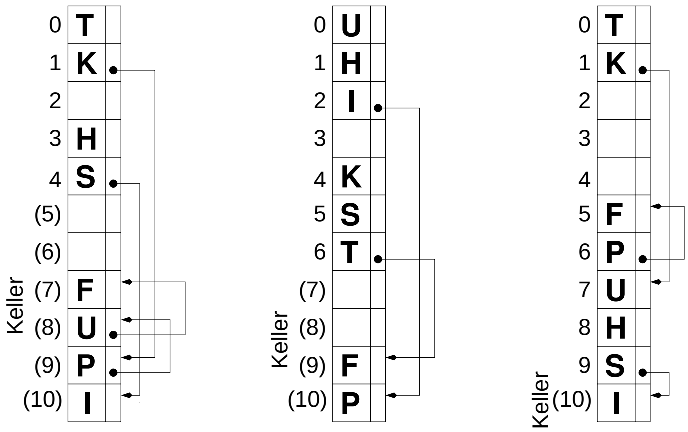
Definition 3.1 (Keller)
Der für die Überläufer reservierte Speicherplatz beim Coalesced Hashing wird Keller genannt.

Beispiel 11 illustriert dieses Verfahren:

Beispiel 11 *(Fortführung von Beispiel 10).*

Information	H	K	S	I	P	U	T	F
Schlüssel	8	11	19	9	16	21	20	6
mod 5	3	1	4	4	1	1	0	1
mod 7	1	4	5	2	2	0	6	6
mod 10	8	1	9	9	6	1	0	6

Folgende Graphik stellt die Hashtabelle für verschieden große Keller dar:



Da der Keller einfach am Ende der normalen Hashtabelle angehängt wird, ändern sich die Algorithmen für Such- Einfüge- und Löschoperationen nicht im Vergleich zum Fall ohne Keller. Wenn man den insgesamt verwendbaren Speicher als konstant ansieht, stellt sich die Frage, wie groß der Keller sein soll. Ist der Keller zu klein, müssen viele Überläufer in der eigentlichen Hashtabelle gespeichert werden. Wird umgekehrt der Keller vergrößert, verringert sich der direkt adressierbare Teil der Hashtabelle, und es treten mehr Kollisionen auf. Eine Antwort auf diese Frage gibt die folgende Analyse, für die noch einige Begriffe eingeführt werden.

Definition 3.2
Die Größe des Kellers wird mit k und der insgesamt zur Verfügung stehende Speicherplatz wird mit $\overline{m}$ bezeichnet.

Die direkt adressierbaren Speicherplätze der Tabelle werden weiterhin mit $0, 1, 2, \ldots, m-1$ bezeichnet. Damit gilt $\overline{m} = m + k$.

Definition 3.3
Die Quotienten $n/\overline{m}$ und $m/\overline{m}$ werden mit $\overline{\alpha}$ bzw. β bezeichnet.

Weiters werden die folgenden Bezeichnungen verwendet:

$\mathbb{E}C^{[l]}_{n,m,\overline{m}}$ ... durchschnittliche Anzahl der notwendigen Vergleiche für erfolg**l**ose Suche

$\mathbb{E}C^{[r]}_{n,m,\overline{m}}$ ... durchschnittliche Anzahl der notwendigen Vergleiche für erfolg**r**eiche Suche

Analyse

Satz 3.6 (Coalesced Hashing ohne Keller [25, 44])
Mit den eben eingeführten Bezeichnungen gilt für $m = \overline{m}$:

$$\mathbb{E}C^{[r]}_{n,m,m} = 1 + \frac{m}{8n}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) + \frac{n-1}{4m} \sim 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$
$$\mathbb{E}C^{[l]}_{n,m,m} = 1 + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) \sim 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$$

Beweis: Erfolgt als Spezialfall im Beweis des nächsten Satzes.

Satz 3.7 (Coalesced Hashing mit Keller [44])
Wenn λ die eindeutige nichtnegative Lösung der Gleichung $e^{-\lambda} + \lambda = \frac{1}{\beta}$ bezeichnet, gilt:

$$\mathbb{E}C^{[r]}_{n,m,\overline{m}} \approx \begin{cases} 1 + \frac{\alpha}{2} & \text{für } \overline{\alpha} \le \lambda\beta \\ 1 + \frac{1}{8\alpha}\big(e^{2(\alpha-\lambda)} - 1 - 2(\alpha-\lambda)\big)\big(3 - \frac{2}{\beta} + 2\lambda\big) & \\ \quad + \frac{1}{4}(\alpha+\lambda) + \frac{\lambda}{4}\big(1 - \frac{\lambda}{\alpha}\big) & \text{für } \overline{\alpha} > \lambda\beta \end{cases}$$
$$\mathbb{E}C^{[l]}_{n,m,\overline{m}} \approx \begin{cases} e^{-\alpha} + \alpha & \text{für } \overline{\alpha} \le \lambda\beta \\ \frac{1}{\beta} + \frac{1}{4}\big(e^{2(\alpha-\lambda)} - 1\big)\big(3 - \frac{2}{\beta} + 2\lambda\big) - \frac{1}{2}(\alpha - \lambda) & \text{für } \overline{\alpha} > \lambda\beta \end{cases}$$

Beweis:

$$\mathbb{E}C^{[l]}_{n,m,\overline{m}} = \frac{1}{m}\frac{1}{m^n} \sum_{0 \le a_1,\ldots,a_n < m-1} \sum_{a=0}^{m-1} C^{[l]}_a \qquad \text{wobei}$$

$C^{[l]}_a$ ... Anzahl von Vergleichen, die eine erfolglose Suche für eine festgelegte Hashfolge benötigt, die in Tabellenplatz a startet

Eine Suche kann erst dann als erfolglos abgebrochen werden, wenn das Ende der aktuellen Kette erreicht worden ist. Eine Kette der Länge l mit t Elementen im Keller wird im folgenden als (l,t)- Kette bezeichnet. Eine solche (l,t)- Kette trägt zur obigen Summe

$$l + (l-t-1) + (l-t-2) + \ldots + 1 = l + \binom{l-t}{2}$$

bei, da genau bei den Elementen der Kette, die nicht im Keller liegen, zu suchen begonnen werden kann. Weiters wird wieder angenommen, dass eine leere Kette genau einen Vergleich benötigt. Eine (l,t)- Kette trägt damit insgesamt $\delta_{l,0} + l + \binom{l-t}{2}$ zur Summe bei.

$c_n(l,t)$ ... Anzahl der (l,t)- Ketten in allen m^n möglichen Tabellen, wobei dieser Ausdruck für „unsinnige“ Werte von n, l und t wie z.B. $t > l$ oder $l > n$ gleich 0 sein soll

E_n ... Erwartete Anzahl an freien, direkt adressierbaren Plätzen in einer Tabelle

$$\begin{aligned} m^{n+1}\mathbb{E}C^{[l]}_{n,m,\overline{m}} &= \sum_{l,t\in\mathbb{N}} \left(\delta_{l,0} + l + \binom{l-t}{2}\right) c_n(l,t) \\ &= \underbrace{\sum_{l,t\in\mathbb{N}} l c_n(l,t)}_{=nm^n} + \underbrace{\sum_{l,t\in\mathbb{N}} \delta_{l,0} c_n(l,t)}_{=c_n(0,0)=m^n E_n} + \underbrace{\sum_{l,t\in\mathbb{N}} \binom{l-t}{2} c_n(l,t)}_{:=S_n} \end{aligned}$$

Die erste hier auftretende Summe kann einfach berechnet werden, wenn man alle Ketten gedanklich zusammenfasst, die zur selben der m^n möglichen Tabellen gehören. Da jedes Element der

Tabelle (inklusive Keller) in genau einer Kette liegt, umfassen alle Ketten einer Tabelle genau n Elemente.
Die dritte Summe ist am schwierigsten zu behandeln. Ist in einer Tabelle der Keller nicht voll, so ist $l-t=1$ und damit der Binomialkoeffizient in dieser Summe für diese Kette gleich Null. Deswegen tragen nur Ketten aus Tabellen mit vollem Keller zu S_n bei.

$(l,t)^+$	…	(l,t)- Kette in Tabelle mit vollem Keller
$(l,t)^-$	…	(l,t)- Kette in Tabelle deren Keller genau einen freien Platz besitzt
$c_n^+(l,t), c_n^-(l,t)$	…	Anzahl aller $(l,t)^+$- bzw. $(l,t)^-$- Ketten in allen m^n Tabellen

Eine $(l,t)^+$- Kette in einer Tabelle mit n Elementen kann auf folgende Arten aus einer Kette einer Tabelle mit $n-1$ Einträgen entstehen:

alte Kette	n- tes Element wird eingefügt …
$(l,t)^+$	… an einem der $m-l+t-\delta_{l,0}$ Plätze außerhalb der alten Kette
$(l-1,t)^+$	… an einem der $l-1-t+\delta_{l,1}\delta_{t,0}$ Plätze, die von der alten Kette belegt werden
$(l,t)^-$	… im Keller, die alte Kette bleibt unverändert
$(l-1,t-1)^-$	… im Keller, die Länge der alten Kette erhöht sich um 1

$$c_n^+(l,t) = (m-l+t)c_{n-1}^+(l,t) + (l-1-t)c_{n-1}^+(l-1,t) + c$$

Hierbei bezeichnet c einen Term, der $c_{n-1}^-(l,t)$, $c_{n-1}^-(l-1,t-1)$ und Korrekturausdrücke für die Sonderfälle $l=t=0$ und $l=1, t=0$ enthält. Dieser braucht nicht näher betrachtet werden, da er beim folgenden Einsetzen wegfällt, weil jeweils $\binom{l-t}{2}=0$ ist.

Für $n \geq k+2$ gilt damit:

$$\begin{aligned}
S_n &= \sum_{l,t\in\mathbb{N}} \binom{l-t}{2} c_n(l,t) = \sum_{l,t\in\mathbb{N}} \binom{l-t}{2} c_n^+(l,t) \\
&= \sum_{l,t\in\mathbb{N}} \binom{l-t}{2}(m-l+t)c_{n-1}^+(l,t) + \sum_{l,t\in\mathbb{N}} \binom{l-t}{2}(l-1-t)c_{n-1}^+(l-1,t) \\
&= \sum_{l,t\in\mathbb{N}} \binom{l-t}{2}(m-l+t)c_{n-1}^+(l,t) + \sum_{l,t\in\mathbb{N}} \binom{l+1-t}{2}(l-t)c_{n-1}^+(l,t) \\
&= \sum_{l,t\in\mathbb{N}} \left(m\binom{l-t}{2} + \frac{(l-t)(l-t-1)}{2}(t-l) + \frac{(l+1-t)(l-t)}{2}(l-t)\right)c_{n-1}^+(l,t) \\
&= \sum_{l,t\in\mathbb{N}} \left(m\binom{l-t}{2} + \frac{(l-t)^2}{2}2\right)c_{n-1}^+(l,t) = \sum_{l,t\in\mathbb{N}} \left((m+2)\binom{l-t}{2} + (l-t)\right)c_{n-1}^+(l,t) \\
&= (m+2)S_{n-1} + \sum_{l,t\in\mathbb{N}} (l-t)c_{n-1}^+(l,t)
\end{aligned}$$

Fasst man in der hier auftretenden Summe wieder all jene Ketten gedanklich zusammen, die zur selben Hashtabelle der Größe $n-1$ gehören, erhält man genau die $n-1-k$ Elemente, die nicht im vollen Keller liegen.

F_n … Anzahl an Hashtabellen mit vollem Keller unter allen m^n Möglichkeiten

$$S_n = (m+2)S_{n-1} + (n-1-k)F_{n-1}$$

Sei nun $n < k+2$. Für $n \leq k$ kann der Keller niemals voll werden. Mit $S_0 := 0$ und $F_0 := 0$ gilt wegen $S_n = S_{n-1} = F_{n-1} = 0$ die obige Rekursion für S_n ebenfalls.

$$\begin{aligned} S_n &= \sum_{i=1}^{n}(m+2)^{i-1}(n-i-k)F_{n-i} = \sum_{i=0}^{n-1}(m+2)^{n-1-i}(i-k)F_i \\ &= (m+2)^{n-1}\sum_{i=0}^{n-1}\Big(\frac{m}{m+2}\Big)^i(i-k)\frac{F_i}{m^i} \end{aligned}$$

Für den Sonderfall $k=0$ ist $F_i = m^i$, wodurch die Summe leicht berechnet werden kann. Setzt man weiters $q := \frac{m}{m+2}$, so ergibt sich:

$$\begin{aligned} S_n &= (m+2)^{n-1}\sum_{i=0}^{n-1} iq^i = (m+2)^{n-1}\sum_{j=1}^{n-1}\sum_{i=j}^{n-1} q^i = (m+2)^{n-1}\sum_{j=1}^{n-1}\frac{q^n-q^j}{q-1} \\ &= (m+2)^{n-1}\frac{nq^{n+1}-nq^n-q^{n+1}+q}{(q-1)^2} = \frac{1}{4}(m(m+2)^n - m^{n+1} - 2nm^n) \end{aligned}$$

Weiters ist $E_n = m-n$ wegen $k=0$, damit folgt:

$$\begin{aligned} m^{n+1}\mathbb{E}C^{[l]}_{n,m,m} &= nm^n + m^nE_n + S_n = nm^n + m^n(m-n) + \frac{1}{4}(m(m+2)^n - m^{n+1} - 2nm^n) \\ \mathbb{E}C^{[l]}_{n,m,m} &= \frac{n}{m} + \frac{m-n}{m} + \frac{1}{4}\Big(\Big(\frac{m+2}{m}\Big)^n - 1 - \frac{2n}{m}\Big) = 1 + \frac{1}{4}\Big(\Big(1+\frac{2}{m}\Big)^n - 1 - \frac{2n}{m}\Big) \end{aligned}$$

Im allgemeinen Fall können E_n und F_n nicht so einfach berechnet werden, sondern man muss sich mit Abschätzungen begnügen. Sei nun B_n wie in Satz 3.4 die Anzahl der nichtleeren Plätze in einer „gewöhnlichen" Hashtabelle mit Verkettung. Für Coalesced Hashing hat B_n folgende Bedeutung:

$B_n \leq n-k$	…	es gab mindestens k Kollisionen $\Longrightarrow$ der Keller ist voll
$B_n > n-k$	…	es gab weniger als k Kollisionen $\Longrightarrow$ Keller nicht voll

Für die weitere Berechnung wird nun die folgende Näherung verwendet:

$$\frac{F_n}{m^n} = \mathbb{P}(B_n \leq n-k) \approx \begin{cases} 1 & \text{für } B_n \leq n-k \\ 0 & \text{für } B_n > n-k \end{cases} \quad \text{denn:}$$

für $n-k \ll \mathbb{E}B_n$	…	der Anteil an Tabellen mit vollem Keller ist ≈ 0
für $n-k \gg \mathbb{E}B_n$	…	der Anteil an Tabellen mit vollem Keller ist ≈ 1

Sei nun l jene natürliche Zahl mit $l-1-k < \mathbb{E}B_{l-1}$ und $l-k > \mathbb{E}B_l$. Für $l > n-1$ ist $S_n \approx 0$, sonst gilt:

$$\begin{aligned} S_n &= (m+2)^{n-1}\sum_{i=l}^{n-1}\Big(\frac{m}{m+2}\Big)^i(i-k) = (m+2)^{n-1}\Big(\sum_{i=l}^{n-1} iq^i - \sum_{i=l}^{n-1} kq^i\Big) \\ &= \frac{(m+2)^{n-1}}{(q-1)^2}((n-1)q^{n+1} - nq^n - (l-1)q^{l+1} + lq^l) - \frac{k(m+2)^{n-1}}{q-1}(q^n - q^l) \\ &= \frac{m^{n+1}}{4}\Big(-\frac{2n}{m} - 1 + \frac{2k}{m} + \Big(\frac{m}{m+2}\Big)^{n-l}\Big(\frac{2l}{m} + 1 - \frac{2k}{m}\Big)\Big) \end{aligned}$$

Die folgende Gleichung kann zur Approximation von l verwendet werden:

$$x - k = \mathbb{E}B_n = m\big(1 - e^{-\frac{x}{m}}\big), \qquad f(x) := x - k - m\big(1 - e^{-\frac{x}{m}}\big)$$

Wegen $f(0) = -k < 0$ und $f(2m) > 0$, besitzt die Gleichung $f(x) = 0$ eine positive Lösung, wegen $f'(x) = 1 - e^{-\frac{x}{m}} > 0$ für $x > 0$ ist diese eindeutig. Setzt man $\lambda := \frac{l}{m}$, so wird die Bedingung $l \geq n$ zu $\overline{\alpha} \leq \lambda\beta$. Für diesen Fall ist $S_n \approx 0$. Für $\overline{\alpha} > \lambda\beta$ gilt:

$$\begin{aligned}
S_n &\approx \frac{m^{n+1}}{4}\Big(e^{2\frac{n-l}{m}}\Big(2\lambda + 1 - 2\frac{\overline{m} - m}{m}\Big) - 2\alpha - 1 + 2\frac{\overline{m} - m}{m}\Big) \\
&= \frac{m^{n+1}}{4}\Big(e^{2(\alpha-\lambda)}\Big(3 - \frac{2}{\beta} + 2\lambda\Big) - 2\alpha - 3 + \frac{2}{\beta}\Big) \\
&= \frac{m^{n+1}}{4}\Big(e^{2(\alpha-\lambda)} - 1\Big)\Big(3 - \frac{2}{\beta} + 2\lambda\Big) - \frac{m^{n+1}}{2}(\alpha - \lambda)
\end{aligned}$$

Um E_n zu berechnen, erweist es sich wieder als nützlich, auf B_n zurückzugreifen. Die Anzahl an freien, direkt adressierbaren Tabellenplätzen beträgt $m - B_n$, wenn der Keller nicht voll ist, und $\overline{m} - n$ sonst. Mit folgender Bezeichnungsweise ergibt sich dann:

I ... Summation über alle Hashfolgen, die eine Tabelle mit nicht vollem Keller ergeben
II ... Summation über alle Hashfolgen, die eine Tabelle mit vollem Keller ergeben

$$\begin{aligned}
E_n &= \frac{1}{m^n}\sum_{\mathrm{I}}(m - B_n) + \frac{1}{m^n}\sum_{\mathrm{II}}(\overline{m} - n) \\
&= m - \mathbb{E}B_n - \frac{1}{m^n}\sum_{\mathrm{II}}(n - k - B_n) = \overline{m} - n + \frac{1}{m^n}\sum_{\mathrm{I}}(n - k - B_n)
\end{aligned}$$

Für $n \ll l$ ist es sehr unwahrscheinlich, dass der Keller voll ist, umgekehrt ist für $n \gg l$ der Keller mit hoher Wahrscheinlichkeit voll. Deshalb erscheint folgende Näherung plausibel:

$$E_n \approx \begin{cases} m - \mathbb{E}B_n & \text{für } l \geq n \\ \overline{m} - n & \text{für } l < n \end{cases} \approx \begin{cases} me^{-\alpha} & \text{für } \overline{\alpha} \leq \lambda\beta \\ m\big(\frac{1}{\beta} - \alpha\big) & \text{für } \overline{\alpha} > \lambda\beta \end{cases}$$

Setzt man die Ergebnisse für E_n und S_n in $m^{n+1}\mathbb{E}C^{[l]}_{n,m,\overline{m}} = nm^n + m^n E_n + S_n$ ein, erhält man die allgemeine Formel für $\mathbb{E}C^{[l]}_{n,m,\overline{m}}$.

Die Anzahl der benötigten Vergleiche für eine erfolgreiche Suche kann nun einfach aus diesem Resultat berechnet werden. Eine erfolgreiche Suche nach dem $i + 1$- ten eingefügten Element, durchläuft die selben Schritte, wie beim Einfügen dieses Elements. Dies sind $\mathbb{E}C^{[l]}_{i,m,\overline{m}}$ Vergleiche, da vor dem Einfügen eine erfolglose Suche nach dem Element durchgeführt wird. Zusätzlich wird noch ein weiterer Vergleich benötigt, wenn das Element mit einem bereits vorhandenen kollidiert, also wenn das Element nicht am Beginn der Kette steht. Die Wahrscheinlichkeit hierfür ist $1 - \frac{E_n}{m}$. Setzt man nun obige Resultate in die folgende Summe ein, erhält man die noch fehlenden Ergebnisse.

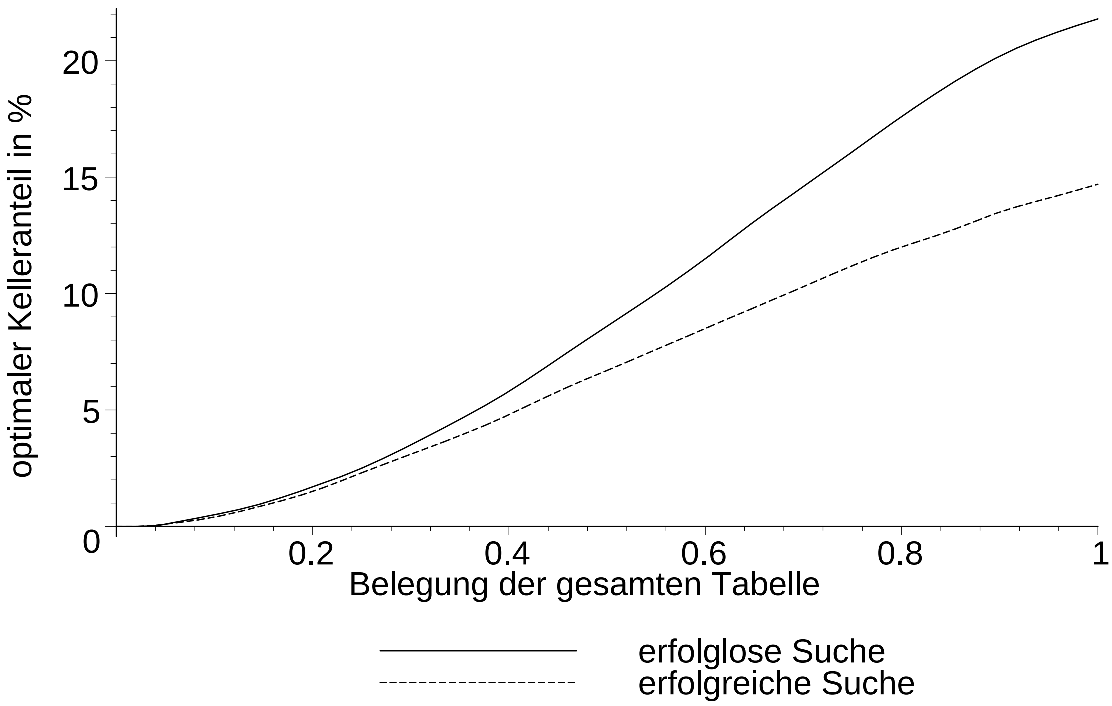
$$\mathbb{E}C^{[r]}_{n,m,\overline{m}} = \frac{1}{n}\sum_{i=0}^{n-1}\Big(\mathbb{E}C^{[l]}_{i,m,\overline{m}} + 1 - \frac{E_n}{m}\Big)$$

■

Die Ergebnisse dieses Satzes lassen sich auch anschaulich interpretieren. Für $\overline{\alpha} \leq \lambda\beta$ ist der Keller mit hoher Wahrscheinlichkeit nicht voll, wodurch die Ketten nicht verschmelzen. Deshalb stimmen die Ergebnisse für diesen Fall auch mit den Resultaten für „gewöhnliches“ Hashing mit Verkettung aus Satz 3.1 überein. Ist jedoch $\overline{\alpha} > \lambda\beta$ ist der Keller wahrscheinlich voll, die Formeln erinnern dann an Coalesced Hashing ohne Keller.

Satz 3.8 ([44])
Der Fehler durch die Abschätzungen im Satz 3.7 ist von der Größenordnung $O\left(\frac{\log \overline{m}}{\sqrt{\overline{m}}}\right)$ *für jeden festen Wert für* β*, unabhängig von* $\overline{\alpha}$*.*

Welche Schlüsse können aus dieser Analyse nun für die praktische Anwendung gezogen werden? Für vorgegebene Auslastung $\overline{\alpha}$ der Tabelle, ist es mit Hilfe der Formeln möglich, den Prozentsatz zu bestimmen, der um optimale Ergebnisse zu erhalten für den Keller reserviert werden sollte. Folgende Graphik zeigt den Wert für β in Abhängigkeit von $\overline{\alpha}$, für den der geringste Aufwand für erfolgreiche und erfolglose Suche auftritt. Für die Ermittlung der Kurven wurden die optimalen Werte für β an 20 regelmäßig verteilten Stützstellen numerisch berechnet und anschließend die erhaltenen Punkte mit kubischen Splines verbunden. Je nachdem, ob mehr erfolgreiche oder erfolglose Suchvorgänge zu erwarten sind, ist die Wahl von β unterschiedlich zu treffen. Als guter Kompromiss, um sowohl den Aufwand für erfolglose als auch erfolgreiche Suche gleichzeitig möglichst gering zu halten, empfiehlt sich laut [44] ein Wert von 0.86 für annähernd volle Tabellen. Weitere Untersuchungen zur optimalen Wahl von β erfolgen im Kapitel 7.



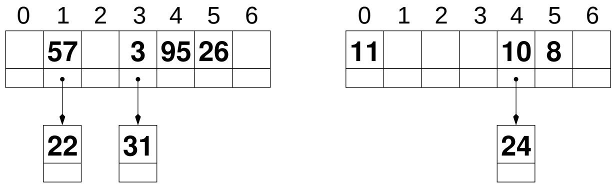
3.4 Verfahren mit mehreren Tabellen

Die Leistung eines Hashverfahrens kann unter Umständen verbessert werden, wenn man statt einer großen Tabelle mehrere kleine verwendet, wobei jede dieser Tabellen eine eigene Hashfunktion besitzt. Im Folgenden wird davon ausgegangen, dass diese kleineren Tabellen Teilbereiche einer einzigen Tabelle der Größe m sind. Diese werden auch als Subtabellen bezeichnet. Hier wird das so genannte d-Left Schema analysiert, das d Subtabellen der selben Größe verwendet. Ein weiteres, ähnlich aufgebautes Verfahren wird im Kapitel 6.3 behandelt.
Die Voraussetzungen des allgemeinen Modells für die Eingangsdaten sollen nun für jede einzelne

Subtabelle gelten, d.h. für jede von ihnen ist jede mögliche Folge von Hashwerten gleichwahrscheinlich. Alle d Untertabellen verwenden Verkettung der Überläufer zur Auflösung von Kollisionen, wobei direkte oder separate Verkettung möglich ist. Ein neu eingefügtes Element wird in jene Subtabelle eingetragen, in der die entsprechende Kette der Überläufer die kürzeste Länge aufweist, sofern diese eindeutig ist. Im Fall eines „Unentschiedens" wird in die am weitesten „links" stehende Untertabelle eingefügt.

Beispiel 12 *In ein 2-Left Schema, das zwei Subtabellen der Größe* $\overline{m} = 7$ *verwendet, sollen die Werte* $3, 57, 26, 10, 95, 31, 8, 11, 22$ *und* 24 *eingetragen werden, wobei zulässige Schlüssel natürliche Zahlen kleiner* 99 *sind. Dazu werden die Hashfunktionen* $h_1(x) = x \mod 7$ *und* $h_2(x) = (3x + 2 \mod 99) \mod 7$ *verwendet. Die Graphik zeigt Hashing mit direkter Verkettung.*

Schlüssel	3	57	26	10	95	31	8	11	22	24
h_1	3	1	5	3	4	3	1	4	1	3
h_2	4	3	3	4	5	4	5	0	5	4



Dieses Verfahren hat allerdings den Nachteil, dass bei der Suche nach einem Element bis zu d Ketten durchsucht werden müssen. Da die Abfragen jedoch unabhängig voneinander sind, kann dieser Nachteil durch parallele Berechnungen ausgeglichen werden, sofern entsprechende Hardware vorhanden ist. Wird vor jedem Einfügevorgang gesucht, ob das Element bereits vorhanden ist, wird auch kein zusätzlichen Speicher benötigt, um die Längen der aktuellen Ketten zu speichern, da diese Suche die Längen automatisch mitliefert.
Natürlich kann das Verfahren auch so abgeändert werden, dass im Fall von gleich langen Ketten die Kette, in die das neue Element eingefügt werden soll, zufällig gewählt wird. Durch diese Änderung werden die Elemente gleichmäßiger auf die einzelnen Untertabellen verteilt. Interessanterweise erweist sich jedoch die ungleichmäßigere Verteilung des ursprünglichen Verfahrens als vorteilhaft. Dies kann damit begründet werden, dass dann weniger unentschiedene Fälle auftreten, die für das Anwachsen der Ketten verantwortlich sind.

Analyse

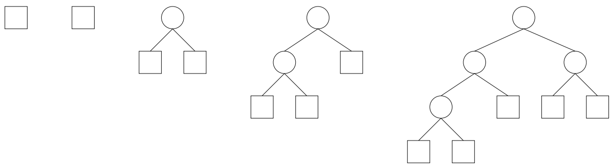
Satz 3.9 (d-Left Schema [46])
Sei h eine positive ganze Zahl. Wenn zu jedem Zeitpunkt höchstens $h \cdot m$ Schlüssel eingetragen sind, dann gibt es eine positive ganze Zahl c, sodass die Wahrscheinlichkeit, dass eine Kette mindestens die Länge $\frac{\ln \ln m}{d \ln \phi_d} + O(h)$ hat, kleiner als m^{-c} ist.

Beweis:
Wesentlicher Bestandteil diese Beweises sind so genannte Witness Trees, die im Fall eines ungünstigen Ereignisses stets konstruiert werden können. Die Existenz eines solchen Baumes ist daher eine notwendige Bedingung für das Auftreten einer langen Kette. Zuerst wird der Fall $h = 1$ behandelt, eine Verallgemeinerung für größere Werte erfolgt abschließend. Der Witness Tree $T_d(k)$ der Ordnung k wird folgendermaßen rekursiv definiert:

- $T_d(1)$ und $T_d(2)$ bestehen nur aus einem Knoten, nämlich der Wurzel.

- $T_d(k)$ für $3 \leq k \leq d$ besteht aus einem Wurzelknoten, der $k-1$ Kinder besitzt, welche die Wurzeln von Bäumen der Art $T_d(k-1), T_d(k-2), \ldots, T_d(1)$ bilden.
- $T_d(k)$ für $k > d$ besteht aus der Wurzel, deren d Kinder die Wurzeln von Bäumen der Art $T_d(k-1), T_d(k-2), \ldots, T_d(k-d)$ bilden.

Der Baum $T_d(k)$ hat entsprechend seiner Konstruktion $F_d(k) \geq \phi_d^{k-2}$ Blätter.[2] Offensichtlich gilt für beliebige Werte von d, dass $T_d(k)$ mindestens so viele Blätter wie innere Knoten besitzt. Die ersten Bäume für den Fall $d = 2$ haben folgende Gestalt:

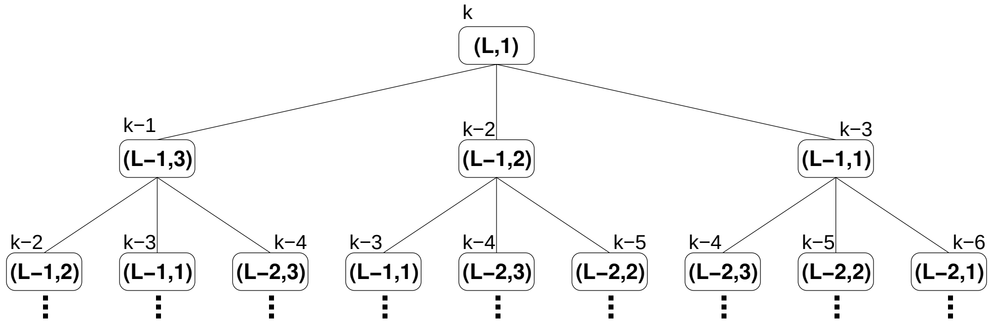


Jedem Knoten des Witness Trees wird nun ein in der Tabelle enthaltener Schlüssel zugewiesen, wobei diese Zuordnung nicht injektiv sein muss. Dabei entspricht jeder Knoten einem Schlüssel, der erst nach allen seinen Kindern zugeordneten Schlüsseln eingefügt wurde. Jede Kante und jedes Blatt des Baumes entspricht einem Ereignis, das auftreten kann, aber nicht auftreten muss:

- Sei (u, v) eine Kante des Baumes, wobei v das i-te Kind von u bezeichnet. Ein Kantenereignis tritt auf, wenn eine der möglichen Platzierungen des zu v gehörenden Schlüssels in der Tabelle gleich der i-ten Position des zu u gehörenden Schlüssel ist.
- Man spricht von einem Blattereignis, wenn jede der möglichen Positionen des dem Blatt zugeordneten Schlüssels beim Einfügen von mindestens drei Schlüsseln besetzt war.

Ein Witness Tree wird genau dann aktiv genannt, wenn alle Blatt- und Kantenereignisse auftreten. Falls eine Kette der Länge größer $L + 3$ in der Hashtabelle auftritt, so kann auf folgende Art ein aktiver Witness Tree konstruiert werden:
Um die Konstruktion besser beschreiben zu können, wird jedem Knoten des Baumes eine Markierung von einem Paar ganzer Zahlen (l, i) mit $0 \leq l \leq L$ und $1 \leq i \leq d$ zugeordnet. Dabei startet man, indem man der Wurzel $(L, 1)$ zuordnet und rekursiv die Kinder eines Knotens (l, i) mit $l \geq 1$ $(l, i-1), \ldots, (l, 1), (l-1, d), \ldots, (l-1, i)$ in der angegebenen Reihenfolge markiert. Die Kinder eines Knotens $(0, i)$ mit $i \geq 3$ bekommen die Markierungen $(0, i-1), \ldots, (0, 1)$. Diese Zuordnungen sind im Allgemeinen nicht injektiv. Die Markierung hat nun folgende Bedeutung: Ein einem Knoten (l, i) zugeordneter Schlüssel wurde beim Einfügen in die Tabelle an eine Kette, die mindestens die Länge $l + 3$ hatte, angehängt. Der Wert i gibt dabei an, welche Hashfunktion verwendet wurde. Nur für die Wurzel des Baumes hat i keine Bedeutung.
Der Wurzel des Baumes wird nun der als letztes eingefügte Schlüssel S einer beliebigen Kette, die mindestens $L + 4$ Schlüssel umfasst, zugewiesen. Vor dem Einfügen von S waren an jeder der d möglichen Positionen von S mindestens $L + 3$ Schlüssel vorhanden. Die jeweils zuletzt eingefügten Schlüssel dieser Ketten werden den Kindern der Wurzel zugewiesen und bekommen die Markierungen $(L-1, 1), (L-1, 2), \ldots, (L-1, d)$. Falls ein Schlüssel S der einem Knoten v mit Markierung (l, i) zugeordnete Schlüssel ist, so waren vor dem Einfügen von S mindestens $l + 3$ Schlüssel an den Plätzen $1, 2, \ldots, i-1$ und mindestens $l + 2$ an den Plätzen $i, i+1, \ldots, d$, usw.
Folgende Graphik zeigt einen Witness Tree für $d = 3$. Die Zahlen $k, k-1, k-2, \ldots$ geben dabei die Ordnung des Baumes an, wenn man den Knoten, bei dem sie stehen, als Wurzel auffasst und übergeordnete Knoten abschneidet. Es gilt $k = dL + 1$.

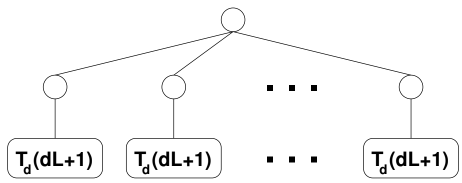
[2]siehe Satz 2.8



Sei nun K die Anzahl der Knoten des Witness Trees, dann gibt es höchstens m^K Möglichkeiten, Schlüssel zu den einzelnen Knoten zuzuordnen. Die Wahrscheinlichkeit, dass ein Kantenereignis auftritt, beträgt $\frac{d}{m}$, sofern die beiden zugehörigen Schlüssel verschieden sind. Setzt man nun vorübergehend voraus, dass die den Knoten zugeordneten Schlüssel alle paarweise verschieden sind, so erhält man einen Wert von $\left(\frac{d}{m}\right)^{K-1}$ für die Wahrscheinlichkeit, dass alle Kantenereignisse auftreten. Sei nun β_i der Anteil von Plätzen in der i-ten Subtabelle, der eine Kette enthält, deren Länge mindestens drei beträgt. Die Wahrscheinlichkeit, dass ein Blattereignis auftritt, ist demnach $\prod_{i=1}^{d} \beta_i$. Wegen $\sum_{i=1}^{d} \beta_i \frac{m}{d} \leq \frac{m}{3}$ gilt $\sum_{i=1}^{d} \beta_i \leq \frac{d}{3}$. Da obiges Produkt genau dann maximal wird, wenn alle β_i den Wert $\frac{1}{3}$ annehmen, gilt $\prod_{i=1}^{d} \beta_i \leq 3^{-d}$. Bezeichne nun B die Anzahl der Blätter. Die Wahrscheinlichkeit, dass ein aktiver Witness Tree mit paarweise verschiedenen Schlüsseln vorliegt, ist demnach durch folgenden Ausdruck beschränkt:

$$m^K \left(\frac{d}{m}\right)^{K-1} 3^{-dB} \leq md^{2B}3^{-dB} \leq m2^{-B} \leq m2^{-\phi_d^{dL-1}}$$

Dabei wurde $K \leq 2B$, $2d^2 \leq 3^d$ und $B = F_d(Ld+1) \geq \phi_d^{dL-1}$ verwendet. Wählt man nun $L \geq \frac{\ln \log_2 m + \ln(1+c)}{d \ln \phi_d} + 1$, ist die Wahrscheinlichkeit, dass so ein Baum aktiv ist, höchstens m^{-c}.
Da im Allgemeinen nicht davon ausgegangen werden kann, dass die den Knoten zugeordneten Schlüssel alle verschieden sind, wird eine Erweiterung der bisherigen Witness Trees, der so genannte vollständige Witness Tree, benötigt. Ein vollständiger Witness Tree besteht aus κ herkömmlichen Witness Trees und ist folgendermaßen aufgebaut:



Wesentliche Voraussetzung ist, dass den Kindern der Wurzel unterschiedliche Schlüssel zugeordnet werden. Sei nun x eine Position in der Tabelle, die eine Kette von mindestens $L+4+\kappa$ Schlüsseln enthält. Die κ zuletzt eingetragenen Schlüssel dieser Kette werden nun den Kindern der Wurzel des vollständigen Witness Trees zugeordnet. Der Wurzel selbst wird der selbe Schlüssel wie dem ersten Kind zugeordnet. Sei nun S der einem beliebigen Kind v der Wurzel zugeordneter Schlüssel. Vor dem Einfügen von S in die Tabelle hat jede mögliche Position, an die S eingefügt werden konnte, mindestens $L+4$ Schlüssel enthalten. Es wird aus diesen Positionen eine von x verschiedene zufällig ausgewählt und daraus der letzte vor S eingefügte Schlüssel ermittelt. Dieser wird dem einzigen Kind von v zugeordnet, welches der Wurzel eines gewöhnlichen Witness Trees entspricht.
Der so erhaltene vollständige Witness Tree wird nun niveauweise[3] durchsucht. Falls dabei ein

[3] d.h. mit Breitensuche

Knoten v gefunden wird, dem ein Schlüssel zugeordnet ist, der bereits einem vorher untersuchten, von der Wurzel verschiedenen Knoten zugeordnet war, wird der mit v beginnende Teilbaum abgeschnitten. Dies wird solange durchgeführt, bis entweder κ Abschneidungen durchgeführt wurden, oder der ganze Baum durchlaufen wurde. Der durchmusterte Witness Tree besteht dann aus allen untersuchten und nicht abgeschnittenen Knoten. Für einen gewöhnlichen Witness Tree der Ordnung $dL+1$ mit $L = \lceil \frac{\ln \log_2 m + \ln(1+c)}{d \ln \phi_d} \rceil + 1$ gilt folgende Schranke für die Anzahl der Knoten:

$$2F_d(dL+1) \leq 2\phi_d^{dL+1} \leq 2(1+c)(\log_2 m)\phi_d^{2d+1} \leq 2^{2d+2}(1+c)\log_2 m$$

Deshalb genügt es, vollständige Witness Trees zu betrachten, für die die Anzahl der Kanten M, die möglicherweise abgeschnitten werden, $2^{2d+2}\kappa(1+c)\log_2 m$ nicht übersteigt. In so einem Baum gibt es höchstens M^κ Möglichkeiten, κ abzuschneidende Kanten auszuwählen. Wenn nun $K+1$ die Anzahl der Knoten im durchmusterten Baum bezeichnet, gibt es ähnlich zu früher höchstens m^K Möglichkeiten Schlüssel zuzuordnen, da die Wurzel den selben Schlüssel wie ihr linkes Kind erhält. Analog beträgt die Wahrscheinlichkeit, dass alle Kantenereignisse eintreten $\left(\frac{d}{m}\right)^{K-1}$. Setzt man weiters B für die Anzahl der Blätter im durchmusterten Baum, so treten alle Blattereignisse höchstens mit Wahrscheinlichkeit 3^{-dB} auf. Als letztes wird noch eine Schranke für die Wahrscheinlichkeit benötigt, dass eine Kante (u,v) abgeschnitten wird. Dies ist der Fall, wenn der v zugeordnete Schlüssel bereits einem vorher inspizierten, von der Wurzel verschiedenen Knoten zugewiesen wurde, wodurch es höchstens M Möglichkeiten gibt, diesen Schlüssel zu wählen. Die Wahrscheinlichkeit, dass die abgeschnittene Kante aktiv war, beträgt $\frac{d}{m}$, wodurch sich insgesamt eine Schranke von $\frac{Md}{m}$ ergibt. Da die hier erfolgte Auswahl in keiner anderen Abschätzung verwendet wurde, ist die Unabhängigkeit der Ereignisse sichergestellt. Dadurch ergibt sich die folgende Schranke für die Wahrscheinlichkeit des Auftretens eines durchmusterten Witness Trees mit genau κ abgeschnittenen Kanten:

$$\begin{aligned} M^\kappa m^K \left(\frac{d}{m}\right)^{K-1} \frac{1}{3^{dB}} \left(\frac{Md}{m}\right)^\kappa &\leq m \frac{d^{2B}}{3^{dB}} \left(\frac{M^2 d^3}{m}\right)^\kappa \\ &\leq m \left(\frac{\left(2^{2d+2}\kappa(1+c)\log_2 m\right)^2 d^3}{m}\right) = m^{-\kappa+1+o(1)} \end{aligned}$$

Bei dieser Abschätzung wurden weiters $k \leq 2(B+K)$ und $d^2 \leq 3^d$ verwendet. Damit ist die Wahrscheinlichkeit, dass ein ungünstiges Ereignis eintritt durch $m^{-c} + m^{-\kappa+1+o(1)}$ beschränkt, wodurch der Beweis für den Fall $h=1$ abgeschlossen ist.

Um den Beweis für den Fall $h>1$ zu verallgemeinern, werden jedem Blatt eines Witness Trees h Schlüssel statt nur ein einziger zugewiesen. Für diese wird vorausgesetzt, dass sie an eine Kette, die mindestens die Länge β aufweist, angehängt wurden, statt wie bisher Länge drei. Die Anzahl der Möglichkeiten, einem Baum Schlüssel zuzuordnen, ist dann durch $\binom{hm}{h}^B (hm)^{K-B}$ beschränkt. Für die Wahrscheinlichkeit, dass alle Kantenereignisse eintreten, sofern lauter unterschiedliche Schlüssel vorausgesetzt werden, erhält man $\left(\frac{d}{m}\right)^{(h-1)B+K-1}$, da jede Kante zu einem Blatt jetzt h Ereignissen entspricht. Die Wahrscheinlichkeit, dass alle möglichen Positionen eines Schlüssels von mindestens β Schlüsseln besetzt sind, ist höchstens β^{-d}. Da jedoch nach Definition des Baumes alle zu einem Blatt gehörenden Schlüssel eine gemeinsame Tabellenposition besitzen müssen, bleiben nur $d + (h-1)(d-1)$ unabhängige Möglichkeiten für die Positionen der Schlüssel eines Blattes, wodurch sich insgesamt die folgende Schranke ergibt:

$$\binom{hm}{h}^B (hm)^{K-B} \left(\frac{d}{m}\right)^{(h-1)B+K-1} \beta^{-B(d+(h-1)(d-1))}$$

Wird β groß genug gewählt, so ist dieser Ausdruck durch $m2^{-B}$ beschränkt. Der restliche Beweis verläuft analog. ∎

Kapitel 4

Hashing mit offener Adressierung

Im Gegensatz zu Hashing mit Verkettung werden kollidierende Elemente nicht außerhalb der Hashtabelle gespeichert, sondern es wird versucht, das neue Element auf einem noch freien Platz der Tabelle zu speichern. Da nicht bekannt ist, welche der Speicherplätze der Tabelle noch frei sind, wird zu jedem Schlüssel eine Reihenfolge bestimmt, in der alle Plätze in der Tabelle überprüft werden. Dies muss solange durchgeführt werden, bis der erste freie Platz gefunden wird, auf dem das Element dann gespeichert wird. Die Folge, die angibt, in welcher Reichenfolge die einzelnen Plätze untersucht werden, wird Sondierungsfolge genannt. Bei der Suche nach einem Element in der Tabelle muss gegebenenfalls die gesamte Sondierfolge durchlaufen werden. Nur falls ein leerer Tabellenplatz bei der Sondierung gefunden wird, ist sicher gestellt, dass das Element nicht in der Tabelle enthalten ist, und die Suche bricht erfolglos ab.
Aus diesem Grund ist das Löschen von Tabelleneinträgen bei offenen Hashverfahren problematisch. Falls ein Schlüssel gelöscht wird, der einem anderen bei dessen Eintragung einen Platz versperrt hat, wird dieser Schlüssel nicht mehr gefunden, obwohl er noch in der Tabelle gespeichert ist. Bei der Suche nach diesem Schlüssel wird nämlich ein freier Platz in der Sondierfolge vor dem Platz gefunden, an dem der Schlüssel eigentlich gespeichert ist. Deshalb werden die Schlüssel oft nicht wirklich gelöscht, sondern nur als gelöscht markiert. Dadurch kann dieser Platz zur Speicherung eines anderen Schlüssels verwendet werden, eine Suche darf an dieser Stelle jedoch nicht erfolglos abgebrochen werden.
Ein weiterer Nachteil offener Verfahren im Vergleich zu Verfahren mit Verkettung liegt darin, dass ein Belegungsfaktor von eins nicht überschritten werden kann. Als vorteilhaft erweist sich vor allem der geringere Speicherplatzbedarf, da kein Platz zur Speicherung von Adressen verwendet werden muss.

Aus diesen Beschreibungen ergeben sich die folgenden Algorithmen für eine Hashtabelle `T` mit m Einträgen. Ein Tabellenplatz ist dabei eine Struktur, die Speicherung von Schlüssel, Information und Status des Platzes (`used`, `empty` oder `deleted`) übernimmt. Mit $h(S,i)$ wird die i-te Position in der Sondierfolge des Schlüssels S bezeichnet, wobei $h(S,1)$ der Wert der eigentlichen Hashfunktion h ist. Als Sondierfolge wird eine Permutation aller Hashadressen vorausgesetzt.

Algorithmus I: `search(S)`
Suche bei offener Adressierung

```
q = -1                                  // q: speichert erste gefundene gelöschte Stelle
for (i=1, i<=m, i++)
    p = h(S,i)                          // p: speichert speichert aktuelle Position
    if (T[p].key == S && T[p].status == used)
        return (p,q,true)
```

```
    if (T[p].status == empty)
        return (p,q,false)
    if(T[p].status == deleted && q == -1)
        q = p                             erste gelöschte Stelle gefunden
return (p,q,false)
```

Algorithmus J: `insert(S,I)`
Einfügen bei Hashing mit offener Adressierung

```
(p,q,status) = search(S)
if (q == -1)                     // q: Position an die eingefügt wird
    q = p                        // keine gelöschte Stelle gefunden: am Listenende einf.
if (T[q].status != used && status == false)
    T[q].status = used
    T[q].key = S
    T[q].info = I
    return true
return status
```

Algorithmus K: `delete(S)`
Löchen in einer Tabelle mit offener Adressierung

```
(p,q,status) = search(S)
if (status == true)
    T[p].status = deleted
```

4.1 Lineares Sondieren

Die einfachste Möglichkeit, im Fall einer Kollision einen freien Speicherplatz zu finden, besteht darin, immer den Speicherplatz mit der nächst höheren Adresse zu prüfen. Für einen Schlüssel S ergibt sich damit die Sondierfolge

$$h(S), h(S)+1, h(S)+2, \ldots, m-2, m-1, 0, 1, \ldots, h(S)-2, h(S)-1.$$

Beispiel 13 *In eine anfangs leere Tabelle der Größe* 11 *werden mit der Hashfunktion* $h(S) = S \bmod 11$ *die Schlüssel* $7, 12, 15, 53, 28, 3, 6, 70$ *und* 14 *in der angegebenen Reihenfolge eingetragen.*

0	1	2	3	4	5	6	7	8	9	10
	12		3	15	70	28	7	6	53	14

Wird in der obigen Tabelle noch ein weiterer, zufällig gewählter Schlüssel eingefügt, so wird dieser mit einer Wahrscheinlichkeit von $\frac{9}{11}$ *an Position* 0 *eingefügt.*

Das Problem bei diesem Verfahren liegt wie in obigem Beispiel angedeutet darin, dass lange, durchgehend belegte Teile der Hashtabelle schneller wachsen als kurze. Der Effekt wird dadurch zusätzlich verstärkt, dass durch das Schließen von Lücken zwischen großen Teilen noch größere Teile entstehen. Dieses Phänomen wird primäre Häufung genannt und ist besonders stark, wenn der Belegungsfaktor der Tabelle groß ist.

Analyse

Da jede der möglichen Hashfolgen mit der selben Wahrscheinlichkeit auftritt, ist die Wahrscheinlichkeit, dass ein Platz frei bleibt, für jeden der m Plätze der Tabelle gleich groß und beträgt $\frac{m-n}{m}$. Dies ist dadurch bedingt, dass jeder Hashfolge, die den i-ten Tabellenplatz frei lässt, durch zyklische Verschiebung eindeutig eine Folge zugeordnet werden kann, die den ersten Platz frei lässt. Soferne also eine nicht volle Tabelle vorliegt, kann aber sofort o.B.d.A. angenommen werden, dass die erste Position frei ist. Es gibt daher $m^{n-1}(m-n)$ Möglichkeiten, dass eine entsprechende Tabelle entsteht.

Satz 4.1 (Lineares Sondieren, erfolglose Suche [25])

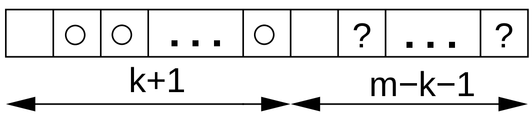
$$\begin{aligned}
\mathbb{E}C_n^{[l]} &= \frac{1}{2}\left(1+\sum_{k\geq 0}\frac{(k+1)n^{\underline{k}}}{m^k}\right) = \frac{1}{2}\big(1+Q_1(m,n)\big)\\
&= \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) - \frac{3\alpha}{2(1-\alpha)^4 m} + O\big(m^{-2}\big)\\
\mathbb{V}C_n^{[l]} &= \frac{1}{6}+\sum_{k\geq 0}\frac{(k+1)(k^2+3k+5)}{12}\frac{n^{\underline{k}}}{m^k} - \big(\mathbb{E}C_n^{[l]}\big)^2\\
&= \frac{3}{4(1-\alpha)^4} - \frac{2}{3(1-\alpha)^3} - \frac{1}{12} - \frac{\alpha(8\alpha+9)}{2(1-\alpha)^6 m} + O\big(m^{-2}\big)
\end{aligned}$$

Beweis: (klassische Analyse nach Knuth)
Für den Beweis werden folgende Begriffe eingeführt:

$f(m,n)$ ... Anzahl an Hashfolgen, die Position 0 freilassen

$g(m,n,k)$ ... Anzahl der Folgen aus $f(m,n)$, für die die Positionen 1 bis k besetzt sind und $k+1$ frei bleibt

$P_{n,k}$ ... Wahrscheinlichkeit, dass das n-te Element genau um k Plätze verschoben eingefügt wird



Jede Folge, die zu $g(m,n,k)$ gehört, kann wie in der Graphik ersichtlich, aus zwei Folgen, die die Längen $k+1$ und $m-k-1$ besitzen, erzeugt werden. Damit gilt:

$$\begin{aligned}
g(m,n,k) &= \binom{n}{k} f(k+1,k) f(m-k-1,n-k)\\
&= \binom{n}{k}(k+1)^{k-1}(m-n-1)(m-k-1)^{n-k-1}\\
P_{n,k} &= \frac{1}{m^{n-1}}\sum_{r\geq 0} g(m,n-1,k+r)\\
\mathbb{E}C_{n-1}^{[l]} &= \sum_{k\geq 0}(k+1)P_{n,k} = 1+\sum_{k\geq 0} k\frac{1}{m^{n-1}}\sum_{j\geq k} g(m,n-1,j)\\
&= 1+\frac{1}{m^{n-1}}\sum_{j\geq 0} g(m,n-1,j)\sum_{k=0}^{j} k = 1+\frac{1}{m^{n-1}}\sum_{j\geq 0} g(m,n-1,j)\frac{j(j+1)}{2}\\
&= 1+\frac{1}{2m^{n-1}}\sum_{j\geq 0}\binom{n-1}{j}(j+1)^{j-1}(m-n)(m-j-1)^{n-j-2}\big((j+1)^2-(j+1)\big)
\end{aligned}$$

$$=: 1 + S_1 + S_2$$

Substituiert man in S_2 für $n-1-j=k$, so erhält man mit Satz 2.12:

$$\begin{aligned}
S_2 &= -\frac{m-n}{2m^{n-1}} \sum_{j=0}^{n-1} \binom{n-1}{j} (j+1)^j (m-j-1)^{n-j-2} \\
&= -\frac{m-n}{2m^{n-1}} \sum_{k=0}^{n-1} \binom{n-1}{k} (n-k)^{n-1-k} (m-n+k)^{k-1} \\
&= -\frac{1}{2m^{n-1}} (m-n+n)^{n-1} = -\frac{1}{2}
\end{aligned}$$

Die Berechnung von S_1 erfordert die Hilfsgröße $s(n,x,y)$. Mit Satz 2.12 gilt nun:

$$\begin{aligned}
s(n,x,y) &:= \sum_{k=0}^{n} \binom{n}{k} (x+k)^{k+1} (y-k)^{n-k-1} (y-n) \\
&= \sum_{k=0}^{n} \binom{n}{k} x(x+k)^k (y-k)^{n-k-1} (y-n) + \sum_{k=0}^{n} \binom{n}{k} k(x+k)^k (y-k)^{n-k-1} (y-n) \\
&= \sum_{k=0}^{n} \binom{n}{k} x(x+n-k)^{n-k} (y-n+k)^{k-1} (y-n) + ns(n-1, x+1, y-1) \\
&= x(y-n+x+n)^n + ns(n-1,x+1,y-1) \\
&= x(x+y)^n + n(x+1)(x+y)^{n-1} + \ldots + n!s(0, x+n, y-n) \\
&= \sum_{k=0}^{n} (x+k) n^{\underline{k}} (x+y)^{n-k}
\end{aligned}$$

Nun kann S_1 aus $S_1 = (2m)^{-(n-1)} s(n-1, 1, m-1)$ bestimmt werden. Durch Zusammensetzen der Ergebnisse erhält man den gesuchten Erwartungswert. Die Varianz kann ausgehend von $\mathbb{E}\big(C_{n-1}^{[l]}\big)^2 = \sum_{k\geq 0} (k+1)^2 P_{n,k}$ berechnet werden. Analog zu $s(n,x,y)$ verwendet man dazu:

$$\begin{aligned}
t(n,x,y) &= \sum_{k\geq 0} \binom{n}{k} (x+k)^{k+2} (y-k)^{n-k-1} (y-n) \\
&= xs(n,x,y) + nt(n-1, x+1, y-1) \\
t(n-1, 1, m-1) &= m^{n-1} \big(3Q_3(m, n-1) - 2Q_2(m, n-1)\big)
\end{aligned}$$

■

Satz 4.2 (Lineares Sondieren, erfolgreiche Suche [18, 25])
Erwartungswert und Varianz erfolgreicher Suche betragen:

$$\begin{aligned}
\mathbb{E}C_n^{[r]} &= \frac{1}{2}\left(1 + \sum_{k\geq 0} \frac{(n-1)^{\underline{k}}}{m^k}\right) = \frac{1}{2}\big(1 + Q_0(m, n-1)\big) \\
&= \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) - \frac{1}{2(1-\alpha)^3 m} + O\big(m^{-2}\big) \\
\mathbb{V}C_n^{[r]} &= \frac{1}{6} + \frac{m}{n}\left(\sum_{k\geq 0} \frac{k^2+k+3}{6} \frac{n^{\underline{k}}}{m^k} - \frac{1}{2}\right) - \big(\mathbb{E}C_n^{[r]}\big)^2 \\
&= \frac{\alpha(\alpha^2 - 3\alpha + 6)}{12(1-\alpha)^3} - \frac{3\alpha+1}{2(1-\alpha)^5 m} + O\big(m^{-2}\big)
\end{aligned}$$

Beweis:
Die Resultate für erfolgreiche Suche können aus denen über erfolglose Suche gewonnen werden:

$$\mathbb{E}C_n^{[r]} = \frac{1}{n}\sum_{k=0}^{n-1}\mathbb{E}C_k^{[l]}$$

Alternativ können die Ergebnisse aus der anschließenden Untersuchung der Momente und der Grenzverteilungen gewonnen werden. ■

Grenzverteilungen

Die folgende Analyse stützt sich auf die Untersuchung der „Gesamtverschiebung" $d_{m,n}$ der einzelnen Einträge. Unter der Verschiebung eines Schlüssels S, der an der i-ten Tabellenposition liegt, wird dabei der Ausdruck $(i - h(s)) \mod m$ verstanden, und die Summe aller dieser Verschiebungen ergibt die Gesamtverschiebung. Aus dieser kann sofort der durchschnittliche Aufwand einer erfolgreichen Suche berechnet werden.
Für die Untersuchung werden folgende Operatoren verwendet:

Definition 4.1 ([18])
Sei $G(z,q) = \sum_{n\geq 0} g_n(q)\frac{z^n}{n!}$ eine beliebige Funktion. Dann bedeutet:

$$UG(z,q) = G(z,1) \qquad \partial_q G(z,q) = \frac{\partial G(z,q)}{\partial q}$$

$$ZG(z,q) = zG(z,q) \qquad \partial_z G(z,q) = \frac{\partial G(z,q)}{\partial z}$$

$$HG(z,q) = \frac{G(z,q) - qG(qz,q)}{1-q}$$

Für den eben definierten Operator H gilt:

$$\begin{aligned} HG(z,q) &= \frac{G(z,q) - qG(qz,q)}{1-q} = \frac{\sum_{n\geq 0} g_n(q)\frac{z^n}{n!} - q\sum_{n\geq 0} g_n(q)\frac{q^n z^n}{n!}}{1-q} \\ &= \sum_{n\geq 0} g_n(q)\frac{z^n}{n!}\frac{1-q^{n+1}}{1-q} = \sum_{n\geq 0} g_n(q)\frac{z^n}{n!}(1+q+q^2+\ldots+q^n) \end{aligned}$$

Satz 4.3 (Vertauschung von H und ∂_q [18])

$$U\partial_q^j H = \sum_{s=0}^{j}\binom{j}{s}\frac{1}{s+1}Z^s\partial_z^{s+1}ZU\partial_q^{j-s} \qquad \text{für } j \geq 0$$

Beweis: Unter Verwendung von Satz 2.10 erhält man:

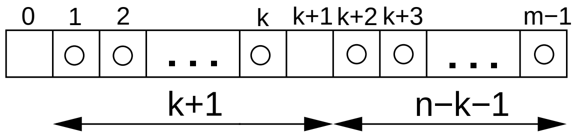
$$\begin{aligned} U\partial_q^j H(z^n q^k) &= U\partial_q^j z^n q^k \sum_{i=0}^{n} q^i = z^n \sum_{i=0}^{n}\sum_{s=0}^{j}\binom{j}{s} i^{\underline{s}}\, k^{\underline{j-s}} = z^n \sum_{s=0}^{j}\binom{j}{s}\frac{1}{s+1}(n+1)^{\underline{s+1}}\, k^{\underline{j-s}} \\ &= \sum_{s=0}^{j}\binom{j}{s}\frac{1}{s+1}z^s\partial_z^{s+1}ZU\partial_q^{j-s}(z^n q^k) \end{aligned}$$

■

Zuerst werden fast volle Tabellen behandelt, dabei bedeutet fast voll, dass genau ein Platz frei ist. Bezeichnet man die Anzahl der Möglichkeiten, eine solche Tabelle zu erzeugen mit $F_{n,k}$, so gilt für die zugehörige EF:

$$F(z,q) := \sum_{n,k\geq 0} F_{n,k} q^k \frac{z^n}{n!} = \sum_{n\geq 0} F_n(q)\frac{z^n}{n!} = 1 + z + (2+q)\frac{z^2}{2!} + (6+6q+3q^2+q^3)\frac{z^3}{3!} + \ldots$$

Betrachtet man eine fast volle Tabelle bevor das n-te Element eingefügt wird, so besitzt diese zwei freie Plätze:



Das n-te Element versucht anschließend auf einen der Plätze $1, 2, \ldots, k+1$ zu gelangen und landet an der Position $k+1$. Damit gilt folgende Beziehung:

$$F_n(q) = \sum_{k=0}^{n-1} \binom{n-1}{k} F_k(q)(1+q+q^2+\ldots+q^k)F_{n-1-k}(q)$$

Satz 4.4 (Fundamentalgleichung der EF [18])

$$\partial_z F = F \cdot HF$$

Beweis:

$$\begin{aligned}
\frac{\partial F(z,q)}{\partial z} &= \sum_{n\geq 1} F_n(q)\frac{z^{n-1}}{(n-1)!} = \sum_{n\geq 0} F_{n+1}(q)\frac{z^n}{n!} \\
&= \sum_{n\geq 0} \frac{z^n}{n!} \sum_{k=0}^{n} \binom{n}{k} F_k(q)(1+q+q^2+\ldots+q^k)F_{n-k}(q) \\
&= \left(\sum_{n\geq 0} F_n(q)\frac{z^n}{n!}\right)\left(\sum_{k=0}^{n} F_k(q)(1+q+q^2+\ldots+q^k)\frac{z^k}{k!}\right) = F(z,q)\cdot HF(z,q)
\end{aligned}$$

■

Führt man nun analog zu $F_{n,k}$ die Bezeichnung $C_{n,k}$ für volle Tabellen mit n Einträgen ein, so gilt für die zugehörige EF $C(z,q)$:

$$HF(z,q) = \sum_{n\geq 0} F_n(q)(1+q+\ldots+q^n)\frac{z^n}{n!} = \sum_{n\geq 0} C_{n+1}(q)\frac{z^n}{n!} = \frac{\partial C(z,q)}{\partial z}$$

Setzt man dieses Ergebnis in die Fundamentalgleichung ein, so erhält man:

$$F(z,q) = e^{C(z,q)} \qquad \text{bzw.} \qquad C(z,q) = \ln F(z,q)$$

Mit Hilfe der Operatoren ∂_q und U ist es jetzt leicht möglich, die faktoriellen Momente zu bestimmen:

$$f_n(z) := U\partial_q^r F(z,q) \qquad \Longrightarrow \qquad \mathbb{E}\big(d_{n,n-1}(d_{n,n-1}-1)\ldots(d_{n,n-1}-r+1)\big) = \frac{[z^n]f_n(z)}{[z^n]f_0(z)}$$

Durch die Anwendung des U Operators auf die Fundamentalgleichung und Satz 4.3 erhält man folgende Gleichung für f_0:

$$f_0'(z) = U\partial_z F(z,q) = UF(z,q)\cdot UHF(z,q) = f_0(z)\cdot \partial_z ZUF(z,q) = f_0(z)\big(zf_0(z)\big)'$$

Daraus kann $\frac{f_0'(z)}{f_0(z)} = \big(\ln f(z_0)\big)' = \big(zf_0(z)\big)'$ und $f_0(z) = e^{zf_0(z)}c$ gefolgert werden. Weiters muss $c = 1$ wegen $f_0(0) = 1$ gelten. Vergleicht man dies mit $T(z)$, der EEF der markierten Wurzelbäume, so erkennt man:

$$f_0(z) = \frac{1}{z}T(z) = e^{T(z)}$$

Analog erhält man für $r \geq 1$:

$$\begin{aligned} f_r'(z) &= U\partial_q^r \partial_z F(z,q) = U\partial_q^r\big(F(z,q)\cdot HF(z,q)\big) = U\sum_{j=0}^{r}\binom{r}{j}\partial_q^{r-j}F(z,q)\cdot\partial_q^j HF(z,q) \\ &= \sum_{j=0}^{r}\sum_{s=0}^{j}\binom{r}{j}\binom{j}{s}\frac{1}{s+1}f_{r-j}(z)z^s\big(zf_{j-s}(z)\big)^{(s+1)} \end{aligned}$$

Untersucht man diese Doppelsumme, so erkennt man, dass nur für $j = 0, s = 0$ und $j = r, s = 0$ Ausdrücke entstehen, die f_r enthalten. Diese lauten $f_r(z)\big(zf_0(z)\big)'$ bzw. $f_0(z)\big(zf_r(z)\big)' = f_0(z)\big(f_r(z) + zf_r'(z)\big)$. Bezeichnet nun $R_r(z)$ die restlichen Ausdrücke der Doppelsumme, so gilt mit dem folgendermaßen definierten Operator $\mathfrak{D}$:

$$\mathfrak{D}f_r(z) := f_r'(z)\big(1 - zf_0(z)\big) - f_r(z)\big(\big(zf_0(z)\big)' + f_0(z)\big) = R_r(z)$$

Verwendet man Variation der Konstanten zur Lösung dieser Differentialgleichung, so erhält man zusammen mit $f_r(0) = 0$:

$$f_r(z) = \frac{e^{T(z)}}{1 - T(z)}\int_0^z R_r(u)e^{-T(u)}du$$

Wegen $z = T(z)e^{-T(z)}$ und $dz = (1 - T(z))e^{-T(z)}du$ kann die Integration ausschließlich in T erfolgen und man erhält beispielsweise (empfehlenswerterweise mit MAPLE):

$$zf_1(z) = \frac{1}{2}\frac{T(z)^3}{\big(1-T(z)\big)^2} \qquad zf_2(z) = \frac{1}{12}\frac{T(z)^4\big(24 - 11T(z) + 2T(z)^2\big)}{\big(1-T(z)\big)^5}$$

$$zf_3(z) = \frac{T(z)^4}{8}\frac{8 + 144T(z) - 110T(z)^2 + 63T(z)^3 - 17T(z)^4 + 2T(z)^5}{\big(1-T(z)\big)^8}$$

Insbesondere können nun die ersten Momente von $d_{n,n}$ berechnet werden:

$$\begin{aligned} U\partial_q C(z,q) &= U\partial_q \ln F(z,q) = \frac{f_1(z)}{f_0(z)} = \frac{1}{2}\frac{T(z)^2}{\big(1-T(z)\big)^2} \\ U\partial_q^2 C(z,q) &= \frac{f_2(z)f_0(z) - f_1(z)^2}{f_0(z)^2} = \frac{1}{12}\frac{T(z)^3\big(24 - 14T(z) + 5T(z)^2\big)}{\big(1-T(z)\big)^5} \end{aligned}$$

Daraus können die bereits bekannten Ergebnisse für $\mathbb{E}d_{n,n}$ und $\mathbb{E}d_{n,n}^2$ ermittelt werden. Von den höheren Momenten interessiert in erster Linie das asymptotische Verhalten.

Satz 4.5 ([18])
Sei $\Omega_0 = -1$ und $2\Omega_r = (3r-4)r\Omega_{r-1} + \sum_{j=1}^{r-1}\binom{r}{j}\Omega_j\Omega_{r-j}$ für $r \geq 1$. Dann gilt für $r \geq 1$ und $z \to e^{-1}$ (d.h. $T(z) \to 1$):

$$zf_r(z) \sim \frac{\Omega_r}{\big(1-T(z)\big)^{3r-1}} \sim \frac{\Omega_r}{\big(2(1-ez)\big)^{\frac{3r-1}{2}}}$$

Beweis: (mit vollständiger Induktion)
Für $r = 1$ und 2 ist die Richtigkeit der Behauptung aus der exakten Lösung offensichtlich. Eine Anwendung von ∂_z auf $\big(1 - T(z)\big)^{-n}$ bewirkt eine Änderung des Exponenten um zwei:

$$\left(\frac{1}{\big(1 - T(z)\big)^n}\right)' = \frac{T'(z)}{\big(1 - T(z)\big)^{n+1}} = \frac{e^{T(z)}}{\big(1 - T(z)\big)^{n+2}}$$

Damit kann man induktiv die dominierenden Anteile von $f_r(z)$ aus folgender Darstellung bestimmen:

$$f_r'(z) = \sum_{j=0}^{r} \sum_{s=0}^{j} \binom{r}{j} \binom{j}{s} \frac{1}{s+1} f_{r-j}(z) z^s \big(z f_{j-s}(z)\big)^{(s+1)}$$

Für $j = 0, s = 0$ und $j = r, s = 0$ treten Ausdrücke mit f_r auf. Setzt man voraus, dass $r - j$ und $j - s$ je weder 0 noch r sind, so erhält man induktiv

$$f_{r-j}(z)\big(z f_{j-s}(z)\big)^{(s+1)} \sim \frac{c_1}{\big(1 - T(z)\big)^{3(r-j)-1+3(j-s)-1+2s+2}} = \frac{c_1}{\big(1 - T(z)\big)^{3r-s}},$$

weshalb nur Ausdrücke mit $s = 0$ weiter betrachtet werden müssen, da für diese der größte Exponent auftritt. Für $j = r, s \neq 0$ und $s \neq r$ gilt

$$f_{r-j}(z)\big(z f_{j-s}(z)\big)^{(s+1)} \sim \frac{c_2}{\big(1 - T(z)\big)^{3r-s+1}},$$

wobei der größte Exponent für $s = 1$ entsteht. In allen weiteren Sonderfällen treten kleinere Exponenten auf, wodurch diese Ausdrücke vernachlässigt werden können. Somit gilt:

$$f_r' = f_r(z f_0)' + f_0(z f_r)' + \frac{r}{2} f_0 z (z f_{r-1})'' + \sum_{j=1}^{r-1} \binom{r}{j} f_{r-j}(z f_j)' + O\left(\frac{1}{(1-T)^{3r-1}}\right)$$

Durch Weglassen und geschicktes Hinzufügen von Ausdrücken der Ordnung $O\big((1-T)^{-(3r-1)}\big)$ entsteht folgende Gleichung:

$$f_r'(1 - z f_0) - f_r(z f_0)' = \frac{r}{2}(z f_{r-1})'' + \sum_{j=1}^{r-1} \binom{r}{j} \frac{1}{2}\big((z f_{r-j})' f_j + f_{r-j}(z f_j)' - f_{r-j} f_j\big) + O\left(\frac{1}{(1-T)^{3r-1}}\right)$$

Integration und Multiplikation mit $2z$ liefert schließlich:

$$2 z f_r (1 - z f_0) = r z (z f_{r-1})' + \sum_{j=1}^{r-1} \binom{r}{j} (z f_{r-j})(z f_j) + O\left(\frac{1}{(1-T)^{3r-3}}\right)$$

Verwendet man $z(z f_{r-1})' \sim z \frac{\Omega_{r-1}(3(r-1)-1)}{(1-T)^{3(r-1)}}(-T') = \frac{\Omega_{r-1}(3r-4)}{(1-T)^{3r+1}} T$ erhält man daraus direkt die angegebene Rekursion für die Ω_r. ■

Die hier aufgetretenen Konstanten Ω_r sind bereits aus der Definition der Airy Verteilung über ihre Momente bekannt.
Mit Hilfe der Singularitätsanalyse [14, 19] kann man nun das asymptotische Verhalten der faktoriellen Momente bestimmen. Da diese klarerweise asymptotisch äquivalent zu den gewöhnlichen Momenten sind, gilt:

$$\mathbb{E} d_{n,n-1}^r \sim \frac{[z^n] f_r(z)}{[z^n] f_0(z)} \sim \frac{\Omega_r (n+1)^{\frac{3r-1}{2}-1} e^{n+1}}{2^{\frac{3r-1}{2}} \Gamma\big(\frac{3r-1}{2}\big)} \frac{(n+1)!}{(n+1)^n} \sim \frac{2\sqrt{\pi}}{\Gamma\big(\frac{3r-1}{2}\big)} \Omega_r \left(\frac{n}{2}\right)^{\frac{3r}{2}}$$

Satz 4.6 (Grenzverteilung für volle und fast volle Tabellen [18])
Die Verteilungen von $\frac{d_{n,n-1}}{(n/2)^{3/2}}$ *und* $\frac{d_{n,n}}{(n/2)^{3/2}}$ *konvergieren für alle* $x \geq 0$ *punktweise gegen eine airyverteilte Zufallsvariable* X*, d.h.:*

$$\mathbb{P}\left(\frac{d_{n,n}}{(n/2)^{3/2}} \leq x\right) \longrightarrow \mathbb{P}(X \leq x) \qquad \text{für } n \longrightarrow \infty$$

Beweis:
Für fast volle Tabellen lässt sich die Aussage sofort aus dem asymptotischen Verhalten der Momente schließen. Bezeichnet u_n eine aus $\{0, 1, 2, , \dots, n-1\}$ gleichverteilte Zufallsvariable, so gilt $\frac{d_{n,n}}{(n/2)^{3/2}} = \frac{d_{n,n-1}+u_n}{(n/2)^{3/2}}$, woraus die Aussage für volle Tabellen geschlossen werden kann. ■

Ausgehend von den zuvor ermittelten Ergebnissen kann jetzt auch der allgemeine Fall einer Tabelle mit $m - n$ freien Einträgen untersucht werden. Eine solche Tabelle kann stets durch Zusammensetzen von $m - n$ fast vollen Tabellen erzeugt werden, z.B.:

	○	○		○			○	○	○	○

Obige Tabelle entsteht durch fast volle Tabellen mit Länge $2, 1, 0$ und 4. Für $H_{m,n}(q)$, die zugehörige EF, die die Anzahl der Möglichkeiten eine Tabelle mit m Plätzen und n Einträgen zu erstellen zählt gilt:

$$H_{m,n}(q) = n![z^n]F(z,q)^{m-n}$$

Dabei kennzeichnet q wie zuvor die Gesamtverschiebung. Unter Anwendung von U und ∂_q kann analog zu vorher wieder auf die faktoriellen Momente geschlossen werden. Zur Ermittlung der entsprechenden Wahrscheinlichkeiten muss abschließend noch durch $H_{m,n}(1) = (m - n)m^{n-1}$ dividiert werden.

$$U\partial_q F(z,q)^{m-n} = (m-n)f_0(z)^{m-n-1}f_1(z)$$
$$U\partial_q^2 F(z,q)^{m-n} = (m-n)(m-n-1)f_0(z)^{m-n-2}f_1(z)^2 + (m-n)f_1(z)^{m-n-1}f_2(z)$$

Durch Ablesen der Koeffizienten kann nun auf Erwartungswert und Varianz der erfolgreichen Suche schließen (Details: siehe [18]).
Mit Hilfe von charakteristischen Funktionen kann gezeigt werden, dass eine Normalverteilung als Grenzverteilung auftritt.

Satz 4.7 (Grenzwertsatz für $\alpha < 1$ [18])
Für $\alpha < 1$ *gilt für* $n \to \infty$*:*

$$\mathbb{P}\left(\frac{d_{m,n} - \mathbb{E}d_{m,n}}{\sqrt{\mathbb{V}d_{m,n}}} \leq x\right) \longrightarrow \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{s^2}{2}} ds$$

Beweis: siehe [18]

4.2 Quadratisches Sondieren

Um die primäre Häufung zu vermeiden, die beim linearen Sondieren auftritt, wird beim quadratischen Sondieren mit quadratisch wachsendem Abstand von der ursprünglichen Position nach leeren Plätzen gesucht. Als Sondierfolge eines Schlüssels S ergibt sich damit die Folge

$$h(S), h(S) + 1, h(S) - 1, h(S) + 4, h(S) - 4, \dots.$$

Die in vielen Büchern vorgeschlagene Folge $h(S), h(S)+1, h(S)+4, \dots$ ist nicht so empfehlenswert, da sie wegen $h(S)+(m-1)^2 \equiv h(S)+(-1)^2 = h(S)+1 \mod m$ keinesfalls eine Permutation darstellt. Für die oben vorgeschlagene Folge gilt jedoch:

Satz 4.8 ([35])
Falls m eine Primzahl mit $m \equiv 3 \mod 4$ ist, so ist sichergestellt, dass die Folge $h(S), h(S)+1, h(S)-1, h(S)+4, h(S)-4, \dots$ eine Permutation aller Hashadressen bildet.

Beweis:
Die Zahlen, deren Quadrat zu $h(S)$ addiert bzw. subtrahiert werden, sind alle kleiner $\frac{m}{2}$. Offensichtlich gilt weiters $h(S) \mod m \neq h(S)+r \mod m$ für alle r mit $1 \leq r < \frac{m}{2}$.

$$h(S) \pm r^2 \equiv h(S) \pm t^2 \mod m \quad \Longrightarrow \quad r^2 - t^2 = (r+t)(r-t) \equiv 0 \mod m$$

Da $r+t \equiv 0 \mod m$ nicht gelten kann, muss wegen der Nullteilerfreiheit $r = t$ sein.

$$h(S) \pm r^2 \equiv h(S) \mp t^2 \mod m \quad \Longrightarrow \quad r^2 + t^2 = km$$

Wegen $r^2 + t^2 \leq \left(\frac{m}{2}\right)^2 + \left(\frac{m}{2}\right)^2 = \frac{m^2}{2}$ muss $m \nmid k$, im Widerspruch zu Satz 2.16 gelten. ■

Auch wenn bei dieser Methode keine primären Häufungen mehr auftreten, bleibt dennoch das Problem, dass die Sondierfolge nicht vom einzufügenden Element abhängt. Dadurch behindern sich kollidierende Elemente weiterhin auf den Ausweichplätzen. Dieser Effekt wird sekundäre Häufung genannt.
Unter der Voraussetzung, dass jede der von einem Tabellenplatz ausgehenden Sondierungsfolgen eine zufällige, aber fest gewählte Permutation bildet, gelten nach [25] die folgenden Laufzeiten:

$$\begin{aligned} \mathbb{E}C_n^{[r]} &\approx 1 + \ln\frac{1}{1-\alpha} - \frac{\alpha}{2} && \dots\text{erfolgreiche Suche} \\ \mathbb{E}C_n^{[l]} &\approx \frac{1}{1-\alpha} + \ln\frac{1}{1-\alpha} - \alpha && \dots\text{erfolglose Suche} \end{aligned}$$

Dieses Resultat stimmt auch in etwa mit praktisch ermittelten Daten für quadratisches Sondieren überein, obwohl obiges Modell an sich nicht erfüllt ist.

4.3 Uniform Hashing

Primäre und sekundäre Häufungen entstehen dadurch, dass die Sondierfolge nicht vom eingefügten Schlüssel abhängt. Beim Uniform Hashing wird für jeden Schlüssel eine zufällig gewählte Permutation aller Hashadressen als Sondierfolge verwendet. Dabei handelt es sich um ein theoretisches Modell, dessen direkte Umsetzung wegen des zu hohen Aufwands nicht möglich ist. Es gibt jedoch Verfahren, die es ermöglichen, annähernd zufällige Permutationen zu erzeugen. Ein Beispiel hierfür ist das im Anschluss behandelte Double Hashing.

Analyse

Satz 4.9 (Uniform Hashing [25])

$$\begin{aligned} \mathbb{E}C_n^{[r]} &= \frac{m+1}{n}(H_{m+1} - H_{m-n+1}) \sim \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \\ \mathbb{E}C_n^{[l]} &= \frac{m+1}{m-n+1} \sim \frac{1}{1-\alpha} \end{aligned}$$

Beweis:
Da jeder Schlüssel an einem zufällig gewählten Platz in der Hashtabelle abgespeichert wird, ist jede der $\binom{m}{n}$ möglichen Belegungen der m Plätze mit n Schlüsseln gleichwahrscheinlich. Bezeichne nun p_i die Wahrscheinlichkeit, dass beim Einfügen des $n+1$-ten Schlüssels genau i Plätze betrachtet werden müssen. Dies ist genau dann der Fall, wenn die ersten $i-1$ untersuchten Plätze belegt sind und der i-te nicht. Dementsprechend müssen die $n-i+1$ anderen Schlüssel auf die restlichen $m-i$ Plätze verteilt sein. Daraus erhält man:

$$\begin{aligned}
p_i &= \frac{1}{\binom{m}{n}}\binom{m-i}{n-i+1} = \frac{1}{\binom{m}{n}}\binom{m-i}{m-n-1} \\
\mathbb{E}C_n^{[l]} &= \sum_{i=1}^{m} ip_i = m+1-\sum_{i=1}^{m}(m+1)p_i - \sum_{i=1}^{m} -ip_i = m+1-\sum_{i=1}^{m}(m+1-i)p_i \\
&= m+1-\sum_{i=1}^{m}\frac{(m+1-i)}{\binom{m}{n}}\binom{m-i}{m-n-1} \\
&= m+1-\sum_{i=1}^{m}\frac{(m+1-i)}{\binom{m}{n}}\binom{m-i+1}{m-n}\frac{m-n}{m-i+1} \\
&= m+1-(m-n)\sum_{i=1}^{m}\frac{1}{\binom{m}{n}}\binom{m-i+1}{m-n}
\end{aligned}$$

mit Satz 2.9 folgt daraus

$$\begin{aligned}
&= m+1-\frac{m-n}{\binom{m}{n}}\binom{m+1}{m-n+1} = m+1-\frac{m-n}{\binom{m}{n}}\binom{m+1}{n} \\
&= m+1-(m-n)\frac{m+1}{m-n+1} = \frac{m+1}{m-n+1} \sim \frac{1}{1-\alpha} = 1+\alpha+\alpha^2+\alpha^3+\dots
\end{aligned}$$

Diese Reihe kann auch intuitiv interpretiert werden: Es muss mit Sicherheit ein Platz inspiziert werden, mit Wahrscheinlichkeit α mindestens ein weiterer, mit Wahrscheinlichkeit α^2 mehr als zwei Plätze usw.

$$\begin{aligned}
\mathbb{E}C_n^{[r]} &= \frac{1}{n}\sum_{i=0}^{n-1}\mathbb{E}C_i^{[l]} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m+1}{m-i+1} = \frac{m+1}{n}\left(\frac{1}{m+1}+\frac{1}{m}+\frac{1}{m-1}+\dots+\frac{1}{m-n+2}\right) \\
&= \frac{m+1}{n}(H_{m+1}-H_{m-n+1})
\end{aligned}$$

■

Satz 4.10 (Uniform Hashing [21])

$$\begin{aligned}
\mathbb{V}C_n^{[r]} &= \frac{2(m+1)}{m-n+2} - \mathbb{E}C_n^{[r]}(\mathbb{E}C_n^{[r]}+1) \sim \frac{2}{1-\alpha} + \frac{1}{\alpha}\ln(1-\alpha) - \frac{1}{\alpha^2}\ln^2(1-\alpha) \\
\mathbb{V}C_n^{[l]} &= \frac{(m+1)n(m-n)}{(m-n+1)^2(m-n+2)} \sim \frac{\alpha}{(1-\alpha)^2}
\end{aligned}$$

Beweis:
Die folgende Summe wurde unter Verwendung von MAPLE berechnet.

$$\mathbb{E}(C_n^{[l]})^2 = \sum_{i=1}^{m} i^2 p_i = \sum_{i=1}^{m} i^2 \frac{1}{\binom{m}{n}}\binom{m-i}{n-i+1} = \frac{(m+n+2)(m+1)}{(m-n+2)(m-n+1)}$$

Zusammen mit $\mathbb{E}C_n^{[l]}$ kann damit $\mathbb{V}C_n^{[l]}$ berechnet werden. Diese Summe ist aber auch der Ausgangspunkt zur Berechnung von $\mathbb{E}(C_n^{[r]})^2$.

$$\begin{aligned}
\mathbb{E}(C_n^{[r]})^2 &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{(m+i+2)(m+1)}{(m-i+2)(m-i+1)} = \frac{m+1}{n}\sum_{i=0}^{n-1}\Big(\frac{2m+3}{m+1-i}-\frac{2m+4}{m+2-i}\Big)\\
&= \frac{m+1}{n}\big((2m+3)(H_{m+1}-H_{m-n+1})-(2m+4)(H_{m+2}-H_{m-n+2})\big)\\
&= \frac{m+1}{n}\big((2m+4)(H_{m+1}-H_{m-n+1}-H_{m+2}+H_{m-n+2})-(H_{m+1}-H_{m-n+1})\big)\\
&= \frac{(m+1)(2m+4)}{n}\Big(\frac{1}{m-n+2}-\frac{1}{m+2}\Big)-\mathbb{E}C_n^{[r]}\\
&= \frac{2(m+1)(m+2)}{n}\frac{n}{(m+2)(m-n+2)}-\mathbb{E}C_n^{[r]} = \frac{2(m+1)}{m-n+2}-\mathbb{E}C_n^{[r]}
\end{aligned}$$

■

4.4 Zufälliges Sondieren

Auch bei zufälligem Sondieren handelt es sich um ein theoretisches Modell. Hierbei verwendet man für jeden Schlüssel eine zufällig gewählte Folge aller Hashadressen zur Sondierung. Im Gegensatz zum Uniform Hashing kann es daher vorkommen, dass ein Tabellenplatz beim Einfügen mehrfach inspiziert wird. Die praktische Realisierung ist unter Verwendung eines Pseudozufallszahlengenerators, der den Schlüssel als Initialwert verwendet, möglich.

Analyse

Es treten nur geringfügig schlechtere Resultate als beim Uniform Hashing auf:

Satz 4.11 (Zufälliges Sondieren [21])

$$\begin{aligned}
\mathbb{E}C_n^{[r]} &= \frac{m}{n}(H_m-H_{m-n}) = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}+O\Big(\frac{1}{m-n}\Big)\\
\mathbb{E}C_n^{[l]} &= \frac{1}{1-\alpha}
\end{aligned}$$

Beweis:
Die intuitive Interpretation der Formel bei Uniform Hashing gilt nun exakt, da bei jeder Sondierung der betrachtete Platz mit Wahrscheinlichkeit α bereits belegt ist.

$$\begin{aligned}
\mathbb{E}C_n^{[l]} &= \sum_{i\geq 0}\mathbb{P}(C_n^{[l]}>i) = \sum_{i\geq 0}\alpha^i = \frac{1}{1-\alpha}\\
\mathbb{E}C_n^{[r]} &= \frac{1}{n}\sum_{i=0}^{n-1}\mathbb{E}C_i^{[l]} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}(H_m-H_{m-n})
\end{aligned}$$

■

Satz 4.12 (Zufälliges Sondieren [21])

$$\begin{aligned}
\mathbb{V}C_n^{[r]} &= \frac{2m^2}{n}\big(H_m^{[2]} - H_{m-n}^{[2]}\big) - \mathbb{E}C_n^{[r]}\big(\mathbb{E}C_n^{[r]} + 1\big) \\
&= \frac{2}{1-\alpha} + \frac{1}{\alpha}\ln(1-\alpha) - \frac{1}{\alpha^2}\ln^2(1-\alpha) + O\Big(\frac{1}{m-n}\Big) \\
\mathbb{V}C_n^{[l]} &= \frac{\alpha}{(1-\alpha)^2}
\end{aligned}$$

Beweis:

$$\begin{aligned}
\mathbb{E}\big(C_n^{[l]}\big)^2 &= \sum_{i\geq 0}(i+1)^2(\alpha^i - \alpha^{i+1}) = (1-\alpha)\sum_{i\geq 0}(i+1)^2\alpha^i = (1-\alpha)\Big(\alpha\Big(\alpha\sum_{i\geq 0}\alpha^i\Big)'\Big)' \\
&= (1-\alpha)\Big(\alpha\frac{1}{(1-\alpha)^2}\Big)' = (1-\alpha)\frac{(1-\alpha)^2 + 2\alpha(1-\alpha)}{(1-\alpha)^4} = \frac{1+\alpha}{(1-\alpha)^2} \\
\mathbb{E}\big(C_n^{[r]}\big)^2 &= \frac{1}{n}\sum_{i=0}^{n-1}\mathbb{E}\big(C_i^{[l]}\big)^2 = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m(i+m)}{(m-i)^2} \\
&= \frac{m}{n}\sum_{i=0}^{n-1}\Big(\frac{2m}{(m-i)^2} - \frac{1}{(m-i)}\Big) = \frac{m^2}{n}\big(H_m^{[2]} - H_{m-n}^{[2]}\big) - \mathbb{E}C_n^{[r]}
\end{aligned}$$

■

4.5 Double Hashing

Dieses Verfahren verwendet eine zweite Hashfunktion zur Erzeugung einer von der ursprünglichen Hashfunktion unabhängigen Sondierfolge, wodurch annähernd die Leistungsfähigkeit von Uniform Hashing erreicht wird [25, 27]. Die Sondierfolge eines Schlüssels S besteht damit aus folgenden Positionen, jeweils modulo m:

$$h_1(S), h_1(S) + h_2(S), h_1(S) + 2h_2(S), h_1(S) + 3h_2(S), \ldots, h_1(S) + (m-1)h_2(S)$$

Dabei muss h_2 so gewählt sein, dass die Sondierfolge für jeden zulässigen Schlüssel eine Permutation aller Hashadressen bildet. Dies wird genau dann erfüllt, wenn h_2 nie Null oder eine zu m nicht teilerfremde Zahl annimmt. Die Größe der Tabelle m sollte deshalb unbedingt eine Primzahl sein. Dann eignet sich die Funktion $h_2(S) := 1 + (S \mod (m-2))$ gut als zweite Hashfunktion. Einerseits gewährleistet sie eine große Zahl von unterschiedlichen Sondierfolgen, andererseits ist sie besser geeignet als $1 + (S \mod (m-1))$, da $m-1$ stets eine gerade Zahl ist.

Beispiel 14 *Analog zu Beispiel 13 soll eine Hashtabelle mit* $h_1(S) := S \mod 11$ *erstellt werden. Jetzt sollen Kollisionen jedoch mit Double Hashing unter Verwendung von* $h_2(S) := 1 + (S \mod 9)$ *aufgelöst werden.*

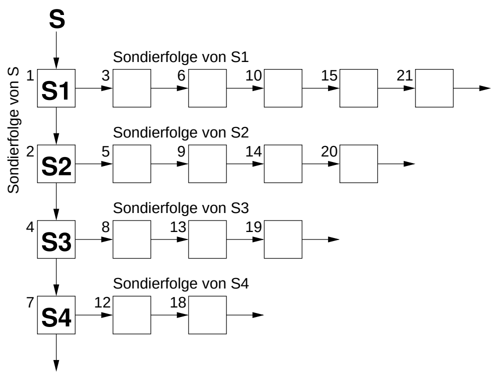
Schlüssel	7	12	15	53	28	3	6	70	14
h_1	7	1	4	9	6	3	6	4	3
h_2	8	4	7	9	2	4	7	8	6

0	1	2	3	4	5	6	7	8	9	10
70	**12**	**6**	**3**	**15**		**28**	**7**		**53**	**14**

Da Double Hashing nicht nur günstiges Laufzeitverhalten besitzt, sondern auch sehr leicht zu implementieren ist, ist es das am häufigsten verwendete offene Hashverfahren. Aus diesem Grund sind auch verschiedene Varianten bekannt, um den Aufwand einer erfolgreichen Suchoperation zu verkleinern, was allerdings auf Kosten der Effizienz der Einfügeoperation geschieht. Alle diese Verfahren beruhen auf der Tatsache, dass der durchschnittliche Aufwand für eine erfolgreiche Suche von der Reihenfolge abhängt, in der die Schlüssel in die Tabelle eingefügt werden. Im obigen Beispiel wurden sechs Schritte benötigt, um den Schlüssel 70 einzufügen. Vertauscht man jedoch die Schlüssel 15 und 70 und behält die weitere Reihenfolge bei, werden nur zwei Schritte benötigt, um 15 einzutragen.

4.6 Brents Algorithmus

Die von Brent in [5] vorgeschlagene Verbesserung ist für alle auf Uniform Hashing basiereneden Algorithmen geeignet und insbesondere zusammen mit Double Hashing in Verwendung. Ziel des Verfahrens ist es, die Gesamtanzahl der Schritte, die zur Suche aller in der Tabelle enthaltenen Schlüssel benötigt werden, möglichst klein zu halten. Die Grundidee basiert darauf, es zu versuchen den Schlüssel zu verschieben der S den Platz versperrt, falls beim Einfügen eines von S ein besetzter Platz gefunden wird. Dies wird aber nur dann durchgeführt, wenn dafür weniger Schritte benötigt werden, als gleichzeitig bei der Suche nach S gewonnen werden. Bei jedem Einfügevorgang wird höchstens ein anderer Schlüssel verschoben. Die folgende Graphik veranschaulicht das Prinzip. Die Nummern geben an, in welcher Reihenfolge die Plätze untersucht werden. Sobald ein freier Platz gefunden wurde, wird die Suche abgebrochen, falls nötig, ein Schlüssel verschoben und S eingefügt. Da die einzelnen Sondierfolgen natürlich nicht disjunkt sind, kann es vorkommen, dass ein Tabellenplatz bei dieser Suche mehrmals inspiziert wird.



Beispiel 15 *Wiederholung des Beispiels 14, allerdings wird jetzt mit Brents Einfügealgorithmus gearbeitet:*

Schlüssel	7	12	15	53	28	3	6	70	14
h_1	7	1	4	9	6	3	6	4	3
h_2	8	4	7	9	2	4	7	8	6

0	1	2	3	4	5	6	7	8	9	10
15	**12**	**6**	**3**	**70**		**28**	**7**		**53**	**14**

Im Vergleich zu gewöhnlichem Double Hashing werden nur mehr 14 *statt* 18 *Vergleiche für die Suche aller Schlüssel benötigt.*

Die Algorithmen für Such- und Löschoperationen bleiben unverändert, der Einfügealgorithmus wird folgendermaßen modifiziert:

Algorithmus L: `insert_brent(S)`
Einfügen nach Brent in eine Tabelle mit Double Hashing

```
(p,q,status) = search(S)
if (status == false)
    p = h_1(S)
    q = h_2(S)
    for (i=0; i<m; i++)              // i: zählt Gesamtverschiebung
        for (j=i; j>=0; j--)         // j: bestimmt Verschiebung von S
            pj = (p+j*q) % m         // pj: Position an der S eingefügt werden soll
            pi = (pj+h_2(T[pj].key)*(i-j)) % m
                                     // pi: Ausweichposition des Schlüssels auf pj
            if (T[pi].status != used)
                T[pi].key = T[pj].key
                T[pj].key = S
                T[pj].status = used
            return true
 return status
```

Falls im obigen Algorithmus `i=j` gilt, so stimmen auch die Positionen `pi` und `pj` überein. Dadurch wird effektiv kein Schlüssel verschoben, sondern `S` wird direkt eingefügt. Dies mag auf den ersten Blick verwirrend erscheinen, ist aber effizienter und übersichtlicher, als diesen Fall gesondert zu betrachten.

Analyse

Der durchschnittliche Aufwand einer erfolglosen Suche ändert sich durch Brents Variation nicht im Vergleich zu dem zu Grunde liegenden Verfahren. Bezeichne nun $\beta < 1$ die aktuelle Auslastung der Tabelle. Damit ist jeder der Plätze mit Wahrscheinlichkeit β belegt. Die Gesamtanzahl der Schritte zur Suche aller in der Tabelle enthaltenen Elemente steigt genau dann um mehr als d Schritte an, wenn die Plätze mit den niedrigsten $\frac{1}{2}d(d+1)$ Nummern in obiger Graphik alle besetzt sind. Unter der Voraussetzung, dass das Uniform Hashing Modell zutrifft, beträgt die Wahrscheinlichkeit hierfür auf Grund der Unabhängigkeit der einzelnen Ereignisse $\beta^{\frac{1}{2}d(d+1)}$, woraus sich durch Summation der Erwartungswert für den Zuwachs berechnen lässt. Damit erhält man:

$$\begin{aligned}\mathbb{E}C_n^{[r]} &\approx \frac{1}{\alpha}\int_0^\alpha \sum_{d=0}^{m} \beta^{\frac{1}{2}d(d+1)} d\beta \approx \sum_{d=0}^{\infty} \frac{1}{\alpha}\int_0^\alpha \beta^{\frac{1}{2}d(d+1)} d\beta \\ &= \sum_{d=0}^{\infty} \frac{\alpha^{\frac{1}{2}d(d+1)}}{1+\frac{1}{2}d(d+1)} = 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^6}{7} + \dots\end{aligned}$$

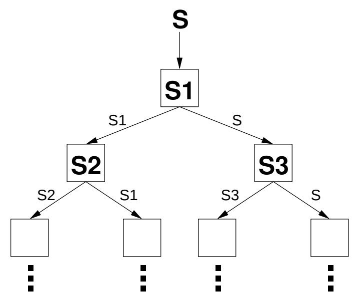
Verwendet man jedoch Double Hashing statt Uniform Hashing, so sind obige Ereignisse nicht mehr alle unabhängig voneinander. Brent gelang es, dies mit Hilfe eines Differentialgleichungssystems zu modellieren und durch numerische Lösung von diesem bestimmte er den Wert

$$\begin{aligned}\mathbb{E}C_n^{[r]} &\approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + 2\frac{\alpha^4}{15} - \frac{\alpha^5}{18} + \frac{\alpha^6}{15} + 9\frac{\alpha^7}{80} - 293\frac{\alpha^8}{5670} - 319\frac{\alpha^9}{5600} + \dots \\ &\leq 2.4941\dots \qquad \text{für } \alpha < 1,\end{aligned}$$

der geringfügig über dem oben ermittelten Wert liegt.

4.7 Binärbaum Sondieren

Ähnlich wie in Brents Variation wird hier durch Verschieben von bereits in der Tabelle vorhandenen Schlüsseln versucht, eine freie Position für das neue Element zu schaffen. Während bei Brents Algorithmus jedoch höchstens ein Schlüssel verschoben wird, können hier auch mehrere bewegt werden. Es wird nämlich nicht nur versucht, die in der Sondierreihenfolge des neuen Schlüssels befinden Einträge zu verschieben, sondern auch alle Schlüssel, die sich in der Sondierreihenfolge eines dieser Einträge befinden usw. Dadurch ergibt sich die folgende Baumstruktur:



In diesem Baum wird niveauweise nach dem ersten freien Platz gesucht. Sobald dieser gefunden wurde, erfolgen die entsprechenden Verschiebungen und der neue Schlüssel kann eingetragen werden.

Analyse

Die Analyse von Gonnet und Munro (vergleiche [22]) ergab

$$\begin{aligned}\mathbb{E}C_n^{[r]} &\approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + 2\frac{\alpha^6}{105} + 83\frac{\alpha^7}{720} + 613\frac{\alpha^8}{5760} - 69\frac{\alpha^9}{1120} + \dots \\ &\leq 2.13414\dots \qquad \text{für } \alpha < 1,\end{aligned}$$

was nur eine geringe Verbesserung gegenüber Brents Algorithmus darstellt. Da die Kosten für Einfügeoperationen, speziell bei Tabellen mit hoher Auslastung, deutlich steigen, ist eine Verwendung meist nur für nahezu statische Tabellen empfehlenswert.

4.8 Robin Hood Hashing

Bei Robin Hood Hashing handelt es sich um eine Technik, die versucht, die Varianz der für eine erfolgreiche Suche benötigten Vergleiche möglichst gering zu halten. Dies wird dadurch erreicht, dass vom bisher angewendeten „First Come First Serve” Prinzip abgegangen wird. Wie Robin Hood nimmt dieser Algorithmus von den „Reichen”, um es an die „Armen” zu verteilen. Bei einer Kollision wird jetzt nicht mehr automatisch das später eingefügte Element bewegt, sondern das Element, das momentan weniger weit verschoben ist. Unter der Verschiebung ist dabei die Anzahl der Vergleiche zu verstehen, die benötigt werden, um den Schlüssel S ausgehend von $h(S)$ zu finden. Um dies zu ermöglichen, muss die Verschiebung eines jeden Elements bekannt sein. Folgende Ansätze sind denkbar:

- Es wird zusätzlicher Speicherplatz verwendet, um die Verschiebung jedes einzelnen Elements festzuhalten.

- Die Verschiebung des Schlüssels S wird durch Suche von S jedesmal neu bestimmt.
- Wird Double Hashing mit den Hashfunktionen h_1 und h_2 mit der Primzahl m verwendet, so kann die Verschiebung des Schlüssels S an Position P mit folgender Formel errechnet werden:

$$v = \big(p - h_1(S)\big)h_2(S)^{-1} \mod m$$

Algorithmus M: `insert_hood(S)`
„Roobin Hood" Einfügen in eine Tabelle mit Double Hashing

```
(p,q,status) = search(S)
if (status == false)
    p = h1(S)
    v = 1                               // v: speichert Verschiebung von S
    while (v <= m)
        if (T[p].status != used)
            T[p].key = S
            T[p].status = used
            return true
        v2 = search(T[p].key)           // v2: Verschiebung des kollidierenden Schlüessels
        if (v > v2)                     // füge S ein, fahre mit anderem Schl"uessel fort
            S2 = T[p].key
            T[p].key = S
            S = S2
            v = v2
        p = (p + h2(S)) % m
        v++
return status
```

Beispiel 16 *Es wird wie in Beispiel 14 eine Tabelle mit Double Hashing erstellt, nur werden die Elemente jetzt nach dem Robin Hood Verfahren eingefügt:*

Schlüssel	7	12	15	53	28	3	6	70	14
h_1	7	1	4	9	6	3	6	4	3
h_2	8	4	7	9	2	4	7	8	6

0	1	2	3	4	5	6	7	8	9	10
15	**70**	**6**	**3**	**7**	**12**	**28**	**53**		**14**	

Analyse

Nimmt man an, dass zufälliges Sondieren erfolgt, so hat dieser Algorithmus keine Auswirkung auf die erwartete Anzahl der Vergleiche einer erfolgreichen Suche. Dies ist durch die Tatsache bestimmt, dass die verbleibende Sondierfolge zweier kollidierender Elemente die selbe Verteilung besitzt. Die Varianz ist jedoch konstant, wie die Analyse von Celis, Larson und Munro zeigt (siehe [8]). Für praktische Zwecke kann man von folgendem Wert ausgehen:

$$\mathbb{V}C_n^{[r]} \leq 1.883$$

Kapitel 5

Universelles Hashing

Die Resultate in den bisherigen Kapiteln zeigen, dass Hashing im durchschnittlichen Fall sehr geringen Aufwand benötigt. Im schlechtesten Fall jedoch, wenn alle Schlüssel auf den selben Hashwert abbilden, ist der Aufwand n. Eine fest gewählte Hashfunktion arbeitet für eine gemäß dem Modell in Abschnitt 1.8 zufällig gewählte Eingangsfolge mit hoher Wahrscheinlichkeit gut. Allerdings gibt es einige „schlechte" Eingaben. Diese stellen bei Verwendung von Hashing stets eine potentielle Gefahr dar, da nicht immer ausgeschlossen werden kann, dass diese schlechten Eingaben nicht gehäuft auftreten. Universelles Hashing, das erstmals in [7] erwähnt wird, bietet einen Ausweg von diesem Problem. Statt nur eine einzelne Hashfunktion zu verwenden, wird mit einer Menge H von Hashfunktionen gearbeitet, wobei die gerade benutzte zufällig ermittelt wird.

Beispiel 17 ([32]) *Verwendet der Compiler zur Verwaltung der Symboltafel universelles Hashing, so wird zu Beginn jedes Übersetzungsvorganges eine Hashfunktion zufällig aus der Menge H gewählt, und diese wird als Hashfunktion für den aktuellen Übersetzungsvorgang verwendet. Dadurch variiert die Zeit, die für die Übersetzung benötigt wird von Aufruf zu Aufruf. Durchschnittlich ist aber ein geringer Aufwand für die Bearbeitung der Symboltafel zu erwarten.*

Die zufällige Wahl der Hashfunktion ist vergleichbar mit der zufälligen Wahl des Pivotelementes in Quicksort[1]. Wenn H gut gewählt wurde, ist ein geringer mittlerer Aufwand zu erwarten. Wann aber ist H gut gewählt? Für zwei unterschiedliche Schlüssel S_1 und S_2 sollte es auf jeden Fall wenige Hashfunktionen h aus H geben, für die eine Kollision auftritt.

Definition 5.1 (c-universell [32])
Sei c eine reelle Zahl. Eine Menge H von Hashfunktionen heißt c-universell, wenn für je zwei ungleiche Schlüssel S_1 und S_2 folgendes gilt:

$$|\{h : h \in H \text{ und } h(S_1) = h(S_2)\}| \leq c\frac{|H|}{m}$$

Eine 1- *universelle Menge wird auch universell genannt.*

5.1 Beispiele für universelle Klassen

Satz 5.1 ([35])
Sei $\{0, 1, 2, \ldots, N-1\}$ die Menge der zulässigen Schlüssel und N eine Primzahl. Dann ist folgende Menge universell:

$$H := \{h_{a,b} : 0 < a < N, 0 \leq b < N \text{ und } h_{a,b}(S) = \big((aS + b) \mod N\big) \mod m\}$$

[1] Der Sortieralgorithmus Quicksort wird z.B. in [40] beschrieben.

Beweis:
Seien S und $\overline{S}$ zwei unterschiedliche (fest gewählte) Schlüssel. Setzt man für a und b jeweils alle möglichen Werte ein, so erhält man durch $r = aS+b \mod N$ und $q = a\overline{S}+b \mod N$ alle Paare (r,q) mit $0 \leq r < N$, $0 \leq q < N$ und $r \neq q$ denn:

$$\begin{aligned} S \neq \overline{S} \quad &\Longrightarrow \quad aS + b \neq a\overline{S} + b \mod N \\ r \neq q \quad &\Longrightarrow \quad a = (r-q)(S-\overline{S})^{-1} \mod N \neq 0, \qquad b = r - (r-q)(S-\overline{S})^{-1} S \mod N \end{aligned}$$

Es gilt nach Definition von $h_{a,b}$:

$$h_{a,b}(S) = h_{a,b}(\overline{S}) \iff (aS+b \mod N) = (a\overline{S}+b \mod N) \mod m$$

Um die Anzahl der Funktionen aus H zu bestimmen, die S und $\overline{S}$ auf den selben Hashwert abbilden, reicht es deswegen aus, die Anzahl der Paare (r,q) mit $0 \leq r < N$, $0 \leq q < N$ und $r \neq q$ zu bestimmen, für die $r = q \mod m$ gilt.
Für festgehaltenes r gibt es wegen $m \nmid N$ höchstens $\lfloor \frac{N}{m} \rfloor \leq \frac{N-1}{m}$ Möglichkeiten, q so zu wählen, dass $r = q \mod m$. Damit gilt:

$$|\{h \in H : h(S) = h(\overline{S})\}| \leq N\frac{N-1}{m} = \frac{|H|}{m}$$

■

Satz 5.2 ([9])
Sei r eine natürliche Zahl und m eine Primzahl. Weiters sei die Menge M aller zulässigen Schlüssel durch die Menge aller Folgen der Länge $r+1$ mit Elementen aus $\{0,1,2,\ldots,m-1\}$ gegeben und $a = (a_0, a_1, a_2, \ldots, a_r)$ eine solche Folge.
Dann ist $h_a(S) = \sum_{i=0}^{r} a_i S_i \mod m$ eine Hashfunktion und $H := \{h_a : a \in M\}$ eine universelle Menge von Hashfunktionen.

Beweis:
Seien S und $\overline{S}$ zwei verschiedene Schlüssel, wobei o.B.d.A. $S_0 \neq \overline{S}_0$. Falls $h_a(S) = h_a(\overline{S})$ so gilt weiters $\sum_{i=0}^{r} a_i S_i = \sum_{i=0}^{r} a_i \overline{S}_i \mod m$ und damit:

$$a_0(S_0 - \overline{S}_0) = \sum_{i=1}^{r} a_i(\overline{S}_i - S_i) \mod m$$

Da m eine Primzahl und $S_0 - \overline{S}_0 \neq 0 \mod m$ ist besitzt diese Gleichung für fest gewählte $a_1, a_2, a_3, \ldots, a_r$ eine eindeutige Lösung a_0.
Damit tritt für jedes Paar von unterschiedlichen Schlüsseln für genau m^r Werte von a eine Kollision auf, woraus mit $|H| = m^{r+1}$ obige Aussage folgt. ■

Beispiel 18 *Für $r = 1$ und $m = 3$ entsprechen die Folgen* 00, 01, 02, 10, 11, 12, 20, 21, 22 *den möglichen Schlüsseln. Es ergibt sich folgende Tabelle:*

Schlüssel	h_{00}	h_{01}	h_{02}	h_{10}	h_{11}	h_{12}	h_{20}	h_{21}	h_{22}
00	0	0	0	0	0	0	0	0	0
01	0	1	2	0	1	2	0	1	2
02	0	2	1	0	2	1	0	2	1
12	0	2	1	1	0	2	2	1	0
20	0	0	0	2	2	2	1	1	1

In [7] wird auch eine universelle Menge von Hashfunktionen angegeben, die zur Auswertung keine Multiplikationen benötigt. Dafür werden folgende Voraussetzungen gemacht:

- Ein Schlüssel ist eine Folge der Form $S = S_1, S_2, S_3, \ldots, S_r$, wobei $0 \leq S_i < B$ für alle S_i gilt.
- Die Größe der Hashtabelle ist eine Zweierpotenz, also $m = 2^t$, und die Tabellenplätze werden mit Binärzahlen adressiert.
- Eine Hashfunktion wird durch eine Binärfolge der Länge $r \cdot t \cdot B$ angegeben. Wenn f so eine Folge ist, dann bezeichne $f(k)$ die k- te Teilfolge der Länge t, d.h. $f = f(1), f(2), \ldots, f(Br)$.
- Mit $\oplus$ wird die bitweise Exklusiv- Oder Verknüpfung bezeichnet.

Satz 5.3 ([7])
Mit den obigen Bezeichnungen ist die Menge H aller wie folgt definierten Hashfunktionen h_f universell.

$$h_f(S) = f(S_1 + 1) \oplus f(S_1 + S_2 + 2) \oplus f(S_1 + S_2 + S_3 + 3) \oplus \ldots \oplus f\Big(\sum_{i=1}^{r} S_i + r\Big)$$

Beweis:
h_f ist wohldefiniert, da $\sum_{i=1}^{r} S_i + r \leq r(B-1) + r = rB$.
Seien S und $\overline{S}$ zwei unterschiedliche Schlüssel, $h_f(S) = f(k_1) \oplus f(k_2) \oplus f(k_3) \oplus \ldots \oplus f(k_r)$ und $h_f(\overline{S}) = f(l_1) \oplus f(l_2) \oplus f(l_3) \oplus \ldots \oplus f(l_r)$.

$$h_f(S) = h_f(\overline{S}) \iff f(k_1) \oplus \ldots \oplus f(k_r) \oplus f(l_1) \oplus \ldots \oplus f(l_r) = 0$$

Da $S \neq \overline{S}$ gilt, gibt es mindestens einen Index i, sodass $k_i \neq l_j \quad \forall j \in \{1, 2, 3, \ldots, r\}$.

$$h_f(S) = h_f(\overline{S}) \iff f(k_i) = f(k_1) \oplus \ldots \oplus f(k_{i-1}) \oplus f(k_{i+1}) \oplus \ldots \oplus f(l_r)$$

Für jede Wahl der $f(1), f(2), f(3), \ldots, f(k_{i-1}), f(k_{i+1}), \ldots, f(Br)$ gibt es also ein eindeutiges $f(k_i)$, sodass eine Kollision zwischen S und $\overline{S}$ auftritt.

$$|\{h \in H : h(S) = h(\overline{S})\}| = 2^{(Br-1)t} = \frac{2^{Brt}}{2^t} = \frac{|H|}{m}$$

■

Beispiel 19 *Setzt man $B = 3$, $r = 2$ und $m = 4$ (damit auch $t = 2$) so ergibt sich $Brt = 12$. Die 12- stellige Folge, die den Schlüssel beschreibt, wird hexadezimal angegeben.*

Schlüssel	h_{000}	h_{68E}	h_{A47}	h_{068}
00	00	11	00	00
01	00	00	11	01
12	00	01	11	10
20	00	10	01	11

5.2 Eigenschaften von universellem Hashing

Satz 5.4 (universelles Hashing mit Verkettung [32])
Sei H eine c-universelle Menge von Hashfunktionen und h aus H zufällig[2] gewählt. Dann hat eine Einfüge-, Such- oder Löschoperation in einer Tabelle mit n verschiedenen Einträgen den mittleren Aufwand $O(1 + c\alpha)$.

Beweis:
Der Aufwand für jede der obigen Operationen für ein Element S ist durch $1 + l_h(S)$ begrenzt, wenn $l_h(S)$ die Anzahl der Elemente in der Tabelle angibt, die bei Verwendung der Hashfunktion h den selben Hashwert wie S haben (siehe Kapitel 3.1). Die in der Hashtabelle eingefügten Werte werden nun mit $S_1, S_2, S_3, \ldots, S_n$ bezeichnet. Diese sind nach Voraussetzung paarweise verschieden.

$$\begin{aligned}\sum_{h\in H}(1 + l_h(S)) &= |H| + \sum_{h\in H}\sum_{i=1}^{n}\delta_{h(S),h(S_i)} = |H| + \sum_{i=1}^{n}\sum_{h\in H}\delta_{h(S),h(S_i)}\\ &\le |H| + \sum_{i=1}^{n} c\frac{|H|}{m} = |H|(1 + c\alpha)\end{aligned}$$

■

Der Aufwand ist also etwa von der selben Größenordnung wie bei den im Kapitel 3 erhaltenen Resultaten. Dieses Ergebnis wurde allerdings unter komplett unterschiedlichen Voraussetzungen erhalten.

Gilt ein ähnliches Resultat auch, wenn Hashing mit offener Adressierung verwendet wird?
Die Resultate für offene Adressierung wurden (wie auch die Resultate über Hashing mit Verkettung) unter der Voraussetzung ermittelt, dass jede der m^n möglichen Hashfolgen gleichwahrscheinlich ist (siehe Kapitel 1.8). Dies kann man auch so auffassen, dass für jeden Schlüssel S, der in die Tabelle eingetragen werden soll, **unabhängig** von allen anderen einzutragenden Schlüsseln gilt:

$$\mathbb{P}\big(h(S) = k\big) = \frac{1}{m} \quad \forall k \in \{0, 1, 2, \ldots, m-1\}$$

Für Hashing mit Verkettung wird dieses Modell verwendet, um die Verteilung der Länge der einzelnen Listen zu bestimmen. Der obige Satz verwendet hierfür einen anderen Zugang. Bei Hashing mit offener Adressierung fließt das Modell jedoch direkt in die eigentliche Berechnung ein.
Betrachtet man die universelle Menge aus Satz 5.2, so sieht man im Beispiel 18 für die beiden Schlüssel 01 und 02, dass nicht alle möglichen Hashfolgen mit der selben Wahrscheinlichkeit auftreten. Allgemein gilt, dass $|H| \ge m^n$ sein müsste, damit für eine vorgegebene n- elementige Menge m^n verschiedene Hashfolgen auftreten können, was aus Speicherplatzgründen viel zu groß wäre. Die folgenden beiden Sätze zeigen, dass eine gleichmäßige Verteilung der einzelnen Hashwerte auch bei universellem Hashing möglich ist, jedoch ohne die oben erwähnte Unabhängigkeit.

Satz 5.5
Für die universelle Menge von Hashfunktionen aus Satz 5.2 gilt für alle $S \neq 0$:

$$\mathbb{P}\big(h(S) = k\big) = \frac{1}{m} \quad \forall k \in \{0, 1, 2, \ldots, m-1\}$$

[2] Wobei für jedes Element aus H die Wahrscheinlichkeit, dass es gewählt wird $1/|H|$, sein soll.

Beweis:
Sei $S = S_0, S_1, S_2, \ldots, S_r$ wobei o.B.d.A. $S_0 \neq 0$ gelten soll. Für beliebig gewählte $a_1, a_2, a_3, \ldots, a_r$ gibt es ein eindeutiges a_0, sodass $h(S) = k \quad \forall k \in \{0, 1, 2, \ldots, m-1\}$:

$$a_0 = \Big(k - \sum_{i=1}^{r} a_i S_i\Big) {S_0}^{-1}$$

Deshalb gibt es für jedes k genau m^r Hashfunktionen, die S auf k abbilden. ■

Satz 5.6
Für die universelle Menge aus Satz 5.3 gilt für alle Schlüssel S:

$$\mathbb{P}\big(h(S) = \mathrm{bin}(k)\big) = \frac{1}{m} \quad \forall k \in \{0, 1, 2, \ldots, m-1\}$$

Beweis:
Sei $S = S_1, S_2, S_3, \ldots, S_r$ ein beliebiger Schlüssel. Wählt man nun beliebige Funktionswerte $f(1), f(2), \ldots, f(S_1), f(S_1+2), \ldots f(Br)$, so gibt es ein eindeutiges $f(S_1+1)$, für welches die Bedingung $h(S) = \mathrm{bin}(k) \quad \forall k \in \{0, 1, 2, \ldots, m-1\}$ erfüllt ist:

$$f(S_1+1) = \mathrm{bin}(k) \oplus f(S_1+S_2+2) \oplus f(S_1+S_2+S_3+3) \oplus \ldots \oplus f(k_r)$$

■

Der folgende Satz liefert eine obere Schranke für die Wahrscheinlichkeit, dass besonders schlechte Fälle bei universellem Hashing mit Verkettung auftreten:

Satz 5.7 ([32])
Unter den Voraussetzungen von Satz 5.4 ist die Wahrscheinlichkeit, dass eine Kette, deren Länge größer als k mal die mittlere Länge ist, auftritt, kleiner gleich $\frac{1}{k}$, falls k eine positive Zahl ist.

Beweis:
Mit $l_h(S)$ wird wieder die Anzahl der Elemente in der Tabelle bezeichnet, die den selben Hashwert wie S haben.

$$\begin{aligned} \mu :=& \frac{1}{|H|} \sum_{h \in H} l_h(S) \\ \overline{H} :=& \{h \in H : l_h(S) \geq k\mu\} \\ \mu \geq& \frac{1}{|H|} \sum_{h \in \overline{H}} l_h(S) \geq k\mu \frac{|\overline{H}|}{|H|} \end{aligned}$$

Damit gilt $|\overline{H}| \leq \frac{|H|}{k}$. ■

Für bestimmte Mengen von universellen Hashfunktionen sind noch bessere Schranken bekannt. So wird z.B. in [30] gezeigt, dass für die universelle Menge aus Satz 5.3 mit analoger Bezeichnungsweise wie im obigen Beweis, folgende Abschätzung gilt, wobei $c(\alpha)$ eine von α abhängige Konstante ist (z.B. $c(1) = 11$):

$$\mathbb{P}\big(|l_h(S) - \mu| > k\big) \leq \min\Big(\frac{\alpha}{k^2}, \frac{c(\alpha)}{k^4}\Big)$$

Kapitel 6

Perfektes Hashing

Perfektes Hashing bietet die Möglichkeit, Zugriffe auf in der Tabelle enthaltene Einträge mit konstantem Aufwand durchzuführen. Im Gegensatz zu den „gewöhnlichen“ Hashverfahren müssen auch die Eingangsdaten nicht gemäß dem Modell aus Kapitel 1.8 gewählt werden. Allerdings muss die Menge der Schlüssel, die in die Tabelle eingetragen werden soll, bereits vor der Erstellung der Hashfunktion bekannt sein. Im einfachsten Fall wird die Hashfunktion dann so konstruiert, dass sie auf dieser Menge injektiv ist. Der größte Nachteil des Verfahrens ist, dass, falls neue Einträge in die Tabelle eingefügt werden sollen, so gut wie immer eine komplett neue Hashfunktion berechnet werden muss. Deshalb eignet sich perfektes Hashing nur für statische Verzeichnisse, in die nach der Erstellung keine weiteren Daten eingetragen werden müssen. Eine Beschreibung nahezu aller perfekten Hashverfahren liefert [10].

Definition 6.1 (perfekt, minimal [10])
Eine Hashfunktion, die Zugriffe auf in der Tabelle gespeicherte Elemente mit konstantem Aufwand erlaubt, wird perfekt genannt. Falls zusätzlich $m = n$ gilt, d.h. die Tabelle den kleinst möglichen Speicherplatz beansprucht, so spricht man von minimalem perfekten Hashing.

Folgende Eigenschaften bestimmen die Qualität einer perfekten Hashfunktion:

- Der (konstante) Aufwand für den Zugriff auf ein in der Tabelle eingetragenes Element.
- Die Größe des von der Hashtabelle belegten Speicherplatzes m.
- Wie hoch der Aufwand ist, eine perfekte Hashfunktion zu generieren.
- Der Speicherplatz, der benötigt wird, um die Hashfunktion zu speichern.

6.1 Elementare Ansätze

Dieser Abschnitt beschreibt einige der einfachsten Verfahren zur Konstruktion perfekter Hashfunktionen. Diese sind nur für spezielle, meist kleine Datenmengen geeignet und deshalb nur von geringer praktischer Bedeutung, zeigen aber, mit welcher Vielfalt an Ideen hier versucht wurde, zum Ziel zu gelangen.

6.1.1 Trial and Error

Für den Fall, dass die Anzahl der Elemente, die in die Hashtabelle eingetragen werden soll, sehr klein ist, kann man versuchen, eine perfekte Hashfunktion durch Ausprobieren zu finden. Beispielsweise kann so eine Tabelle erstellt werden, die Kurzbezeichungen von Wochentagen oder mathematischen Funktionen wie „`mon`“ oder „`sin`“ aufnimmt. Man ordnet dabei jedem Buchstaben des Alphabets eine Zahl zu und verknüpft diese Werte mit einer einfachen Funktion, indem man beispielsweise nur den Wert der Buchstaben einer einzelnen Stelle heranzieht, oder die Summe der Werte von bestimmten Stellen bildet, und versucht so eine injektive Funktion zu finden. Zweckmäßigerweise ordnet man dabei den Buchstaben, die am häufigsten an den relevanten Stellen der einzutragenden Wörter vorkommen, zuerst die Werte zu, wobei man einzelne Werte auch mehrfach vergeben kann. Anschließend versucht man, die restlichen Werte so zu wählen, dass keine Lücken in der Tabelle entstehen. Gegebenenfalls muss dazu ein Gleichungssystem gelöst werden.

Beispiel 20 ([10]) *Ziel dieses Beispiels ist es, eine Hashtabelle mit den folgenden elf Funktionskürzeln einer fiktiven Programmiersprache zu erstellen:*

`exp, cos, log, atn, tng, int, abs, sgn, sqr, sin, rnd`

Bezeichnet nun f jene Funktion, die einen Buchstaben auf die ihm zugeordnete Zahl abbildet, so ergeben sich für die Hashfunktion eines Wortes xyz die Kandidaten $f(x), f(y), f(z), f(x)+f(y)$, $f(x)+f(z), f(y)+f(z), f(x)+f(y)+f(z)$. Versucht man die Funktion $f(x)$ zu verwenden entsteht dadurch eine Kollision zwischen den Wörtern `atn` *und* `abs`*. Streicht man alle Funktionen, für die nicht behebbare Kollisionen auftreten, bleiben nur mehr die Kandidaten $f(x) + f(y)$ und $f(x) + f(y) + f(z)$, wovon die erste wegen der einfacheren Form vorzuziehen ist.*

Zeichen	a	b	c	e	g	i	l	n	o	q	r	s	t	x
Häufigkeit	2	1	1	1	1	2	1	3	2	1	1	3	2	1

Beginnt man nun, den an den ersten beiden Stellen am häufigsten vorkommenden Buchstaben Werte zuzuordnen, kann man z.B. $f(\texttt{n}) = 0$, $f(\texttt{s}) = f(\texttt{i}) = f(\texttt{o}) = 1$ und $f(\texttt{a}) = 3$ setzen. Damit ist der Hashwert für die Wörter `sin` *und* `int` *bereits eindeutig festgelegt. Die folgende Tabelle veranschaulicht die Situation, wobei ein ? bedeutet, dass der Wert noch nicht festgelegt wurde:*

Wort	exp	cos	log	atn	tng	int	abs	sgn	sqr	sin	rnd
Hashwert	?+?	? + 1	? + 1	3+?	? + 0	1 + 0	3+?	1+?	1+?	1 + 1	? + 0

Setzt man nun noch $f(\texttt{t}) = 0$, wird auch der Hashwert von `tng` *und* `atn` *fixiert. Alle weiteren Buchstaben kommen nur mehr einfach vor und können problemlos so gewählt werden, dass eine minimale perfekte Hashfunktion entsteht. Buchstaben, die in keinem der Wörter vorkommen, wird abschließend ein beliebiger Wert zugewiesen.*

Natürlich kann durch schlecht gewählte Werte für einzelne Buchstaben eine Kollision entstehen. In diesem Fall muss wieder um einen Schritt zurückgegangen werden, und mit anderen Werten ein neuer Versuch gestartet werden. Dadurch kann der Aufwand zur Bestimmung einer perfekten Hashfunktion auch schon bei einer geringen Wortanzahl sehr groß sein. Auch kann leicht der Fall auftreten, dass keine Lösung existiert. Ersetzt man nur das Wort `tng` durch `tan` im obigen Beispiel, tritt für jede der obigen Funktionen eine nicht behebbare Kollision auf.

6.1.2 Ein Verfahren mit dem Chinesischen Restsatz

Chang schlug 1984 vor, den Chinesischen Restsatz zu verwenden, um eine einfache Funktion zu bestimmen, die die Schlüssel $S_1, S_2, S_3, \ldots, S_m$ auf die Zahlen $1, 2, 3, \ldots, m$ abbildet [10]. Da im Allgemeinen nicht davon ausgegangen werden kann, dass die einzelnen Schlüssel paarweise teilerfremd sind, wird zuerst eine Funktion verwendet, um sie in teilerfremde Zahlen zu transformieren. Chang schlug dazu die Verwendung einer „Primzahlformel“, wie zum Beispiel das bereits von Euler gefundene Polynom $x^2 + x + 41$ vor. Dieses liefert für die Werte $0, 1, 2, \ldots, 39$ voneinander verschiedene Primzahlen.

Beispiel 21 *Zu den Schlüsseln* $S_1 = 2, S_2 = 4, S_3 = 7, S_4 = 10$ *ergeben sich mit dem Eulerschen Polynom die Primzahlen* $p_1 = 47, p_2 = 61, p_3 = 97, p_4 = 151$. *Mit dem Chinesischen Restsatz erhält man daraus die Zahl* $C = 11262188$ *als Lösung des Kongruenzzsystems* $C \equiv i \mod p_i$ *für* $i = 1, 2, 3, 4$. *Als Hashfunktion erhält man damit insgesamt* $h(x) = 11262188 \mod (x^2+x+41)$.

Die erste große Schwachstelle dieses Verfahrens liegt darin, dass keine auch für größere Wertebereiche verwendbare Primzahlformel bekannt ist. Das Hauptproblem liegt jedoch darin, dass die mit dem Chinesischen Restsatz ermittelte Lösung zwar relativ effizient berechnet werden kann, aber schon bei wenigen Primzahlen sehr groß ist. So beträgt das kleinste positive C, wenn man die ersten 64 Primzahlen heranzieht, bereits ca. $1.91 \cdot 10^{124}$.
Es sind diverse Variationen dieses Verfahrens bekannt, unter anderem eines, das besser für aus Text bestehende Schlüssel geeignet ist. Auch gibt es eine Möglichkeit, auf die Verwendung einer Primzahlformel zu verzichten. Trotz all dieser Verbesserungen ist es nicht gelungen, die Größe der Konstanten C in akzeptabler Größe zu halten, weshalb dieses Verfahren nicht für praktischen Einsatz geeignet ist.

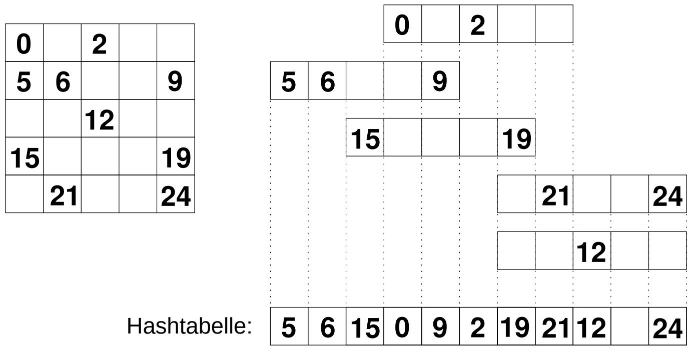
6.1.3 Ein Kompressionsverfahren

Von Trajan und Yao [10] kommt der Vorschlag eine perfekte Hashfunktion analog zu den in der Informatik verwendeten „packed Tries” aufzubauen. Dieses Verfahren ist auch für größere Tabellen geeignet, allerdings werden Voraussetzungen an die Größe des Universums U gestellt:

- $|U|$ darf n^2 nicht überschreiten
- $\sqrt{|U|} := t$ soll eine ganze Zahl sein
- zulässige Schlüssel sind alle natürlichen Zahlen kleiner $|U|$

Unter diesen Voraussetzungen kann man nun ein Feld der Größe $t \times t$ konstruieren, in das man die Schlüssel $S_1, S_2, S_3, \dots, S_n$ gemäß ihrem Wert eintragen kann. Der Schlüssel S wird dabei in Zeile $\lfloor \frac{S}{t} \rfloor + 1$ und Spalte $(S \mod t) + 1$ eingetragen. Nun trennt man dieses Feld in die einzelnen Zeilen auf und versucht diese so überlappend anzuordnen, dass eine einzelne Hashtabelle ohne Kollisionen, von möglichst kleiner Größe entsteht. Da dieses Optimierungsproblem NP-vollständig ist, wird üblicherweise eine „First-fit“ Heuristik angewendet. Dabei wird jede einzelne Zeile um die kleinste Zahl verschoben, die notwendig ist, damit in der entstehenden Tabelle keine Kollision auftritt. Vor dem Zusammenfügen ist es empfehlenswert, die Zeilen nach Anzahl der nichtleeren Einträge zu sortieren und sie in dieser Reihenfolge in der Heuristik heranzuziehen.
Um die Verschiebung der einzelnen Zeilen zu speichern, wird eine Tabelle der Größe t, welche im weiteren mit D bezeichnet wird, benötigt. Ein Schlüssel S ist demnach genau an der Stelle $D[\lfloor \frac{S}{t} \rfloor + 1] + (S \mod t) + 1$ in der fertigen Hashtabelle enthalten, oder überhaupt nicht.

Beispiel 22 *Es soll eine perfekte Hashtabelle erstellt werden, die die Werte* $0, 2, 5, 6, 9, 12, 15, 19, 21$ *und* 24 *enthält, wobei mögliche Schlüssel ganze Zahlen von* 0 *bis* 24 *sind.*



Die Tabelle D besteht demnach aus den Zahlen $3, 0, 6, 2, 6$. Die Zahl 21 stand im Feld in Zeile 5 und Spalte 2, weshalb sich die mögliche Tabellenadresse $D[5] + 2 = 8$ ergibt, an der die Zahl auch gespeichert ist. Für die Zahl 17 ergibt sich die Position $D[4] + 3 = 5$ in der Hashtabelle. An dieser Position ist jedoch 9 gespeichert, weshalb die gesuchte Zahl nicht in der Tabelle enthalten ist.

6.2 Das FKS Verfahren

Fredman, Kolmós und Szemerédi geben in [20] das später nach ihnen benannte FKS- Verfahren an, das ursprünglich entwickelt wurde, um eine konstruktive Schranke für die Eigenschaften von perfektem Hashing zu erhalten. Dieses Verfahren besitzt aber nicht nur eine theoretische Bedeutung, sondern eignet sich auch für die praktische Anwendung. Zunächst müssen jedoch noch einige Voraussetzungen gemacht und einige Begriffe erklärt werden.
Für dieses Verfahren wird vorausgesetzt, dass die Größe des Universums U eine Primzahl ist, welche im weiteren mit u bezeichnet wird. Als mögliche Schlüssel werden alle natürlichen Zahlen kleiner u zugelassen. Daraus werden n Schlüssel beliebig, aber fest gewählt, die in die Hashtabelle eingetragen werden sollen. Die Menge dieser Schlüssel wird mit S bezeichnet.

Definition 6.2
Eine Hashfunktionen der Bauart $x \mapsto (ax \mod u) \mod m$, wobei a eine ganze Zahl mit $1 \leq a < u$ ist, wird FKS- Hashfunktion genannt.

Bezeichnet man nun die Anzahl der Schlüssel von S, die von einer fest gewählten FKS- Hashfunktion auf den i-ten Speicherplatz der Hashtabelle abgebildet werden, mit l_i, so gelten folgende Sätze:

Satz 6.1 ([10, 20])
Falls die Hashtabelle über mindestens n Speicherplätze verfügt, gibt es eine FKS- Hashfunktion, für die die Ungleichung $\sum_{i=0}^{m-1} \binom{l_i}{2} \leq \frac{n(n-1)}{m}$ gilt.

Beweis:
Mit $\binom{l_i}{2}$ wird die Anzahl der Paare von Schlüsseln aus S gezählt, die am i-ten Speicherplatz kollidieren. Damit ergibt sich:

$$\begin{aligned}
\sum_{a=1}^{u-1}\sum_{i=0}^{m-1}\binom{l_i}{2} &= \sum_{a=1}^{u-1}\sum_{i=0}^{m-1} |\{(x,y) : x, y \in S, x \neq y, h(x) = h(y) = i\}| \\
&= \sum_{x,y \in S, x \neq y} |\{a : ((ax \mod u) - (ay \mod u)) \mod m = 0\}| \\
&= \sum_{x,y \in S, x \neq y} |\{a : a(x-y) \mod u \in \{m, 2m, 3m, \ldots, u-m, u-2m, \ldots\}\}|
\end{aligned}$$

Da es wegen $m \nmid u$ für jedes Paar von unterschiedlichen Schlüsseln aus S weniger als $\frac{2u}{m}$ mögliche Werte für a gibt, die diese Bedingung erfüllen, gilt damit:

$$\sum_{a=1}^{u-1} \sum_{i=0}^{m-1} \binom{l_i}{2} \leq \binom{n}{2} \frac{2(u-1)}{m} = \frac{(u-1)n(n-1)}{m}$$

Daraus folgt sofort, dass es ein a geben muss, für das die Behauptung des Satzes gilt. ■

Satz 6.2 ([10, 20])
Für eine Tabelle mit n Plätzen existiert eine FKS- Hashfunktion, für die $\sum_{i=0}^{n-1} l_i^2 < 3n$ gilt. Weiters gibt es eine perfekte FKS- Hashfunktion, wenn die Tabelle $n(n-1)+1$ Speicherplätze besitzt.

Beweis:
Verwendet man die Ungleichung des letzten Satzes für die beiden Spezialfälle $m = n$ und $m = n(n-1)+1$, so gibt es jeweils ein a, für das folgende Beziehung gilt:

$$\sum_{i=0}^{n-1} (l_i)^2 = 2\sum_{i=0}^{n-1} \binom{l_i}{2} + \sum_{i=0}^{n-1} l_i \leq 2\frac{n(n-1)}{n} + n = 2(n-1) + n < 3n$$

$$\sum_{i=0}^{n(n-1)} (l_i)^2 = 2 \sum_{i=0}^{n(n-1)} \binom{l_i}{2} + \sum_{i=0}^{n(n-1)} l_i \leq 2\frac{n(n-1)}{n(n-1)+1} + n < 2 + n$$

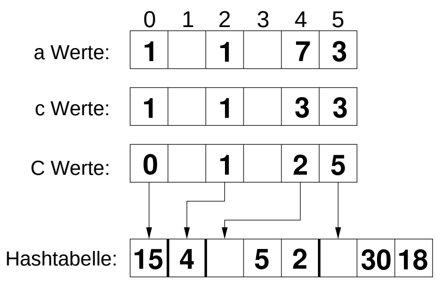
Falls eine Kollision am i-ten Speicherplatz auftritt, gilt $l_i \geq 2$. D.h., dass für eine nicht kollisionsfreie FKS- Hashfunktion $\sum_{i=0}^{n(n-1)} (l_i)^2 \geq n + 2$ gilt.

■

Das FKS- Verfahren verwendet nun die Hashfunktionen, deren Existenz eben gezeigt wurde, in zwei Schritten. Zuerst wird eine Funktion, die so genannte primäre Hashfunktion, gemäß dem ersten Teil des Satzes verwendet, um die Schlüssel der Menge S in n Segmente aufzuteilen. Für jedes der nichtleeren Segmente wird anschließend eine sekundäre Hashfunktion gemäß der zweiten Aussage von Satz 6.2 verwendet. Ein Segment der Größe l_i wird dazu auf einen Speicherplatz der Größe $c_i := l_i(l_i - 1) + 1$ abgebildet. Diese Speicherplätze werden in der Hashtabelle zusammengefasst, wobei der Speicherbereich für das i-te Segment an der Position $C_i := \sum_{j=0}^{i-1} c_j$ beginnt. Die Größe der entstehenden Hashtabelle ist dabei auf Grund der Wahl der primären Hashfunktion mit $3n$ begrenzt. Allerdings wird zur Speicherung der $a-$Werte der sekundären Hashfunktionen und der Werte c_i und C_i weiterer Speicherplatz der Größe $3n$ benötigt.

Beispiel 23 *Für $S = \{2, 4, 5, 15, 18, 30\}$, $n = 6$ und $u = 31$ soll das FKS- Verfahren angewendet werden. Dafür wird der Wert $a = 2$ für die primäre Hashfunktion $h(x) = (ax \mod 31) \mod 6$ ermittelt. Die folgende Graphik gibt die ermittelten Werte an:*

x	2	4	5	15	18	30
$h(x)$	4	2	4	0	5	5



Um zu prüfen, ob der Wert 5 in der Tabelle enthalten ist, wird zuerst das zugehörige Segment mit der primären Hashfunktion berechnet: $h(5) = (2 \cdot 5 \mod 31) \mod 6 = 4$. *Nun ermittelt man die zugehörigen Werte* $a_4 = 7$, $c_4 = 3$ *und berechnet* $h_4(5) = (7 \cdot 5 \mod 31) \mod 3 = 1$. *Der Eintrag steht demnach in der Hashtabelle an der Position* $C_4 + 1 = 3$. *Analog erhält man für den Schlüssel* 7 *die Tabellenadresse* 1, *an der jedoch der Schlüssel* 4 *gespeichert ist, weshalb* 7 *nicht in der Tabelle enthalten sein kann.*

Das größte Problem dieses Verfahrens liegt darin, dass die Suche nach den geeigneten FKS-Hashfunktionen im schlechtesten Fall den Aufwand $O(nu)$ besitzt. Der folgende Satz erlaubt eine schnellere, randomisierte Bestimmung der erforderlichen Konstanten, allerdings steigt dabei die Größe des benötigten Speicherplatzes.

Satz 6.3 ([10, 20])
Für mindestens die Hälfte aller zulässigen Werte für a *einer FKS- Hashfunktion ist die Ungleichung* $\sum_{i=0}^{n-1} l_i^2 < 5n$ *erfüllt. Weiters ist mindestens die Hälfte aller FKS- Hashfunktion perfekt, wenn die Tabelle* $2n(n-1)+1$ *Speicherplätze besitzt.*

Beweis:
Mit der im Beweis von Satz 6.1 erhaltenen Ungleichung $\sum_{a=1}^{u-1} \sum_{i=0}^{m-1} \binom{l_i}{2} \leq \frac{(u-1)n(n-1)}{m}$ gilt:

$$\sum_{a=1}^{u-1} \Big(\sum_{i=0}^{n-1} (l_i)^2 - n \Big) = 2 \sum_{a=1}^{u-1} \sum_{i=0}^{n-1} \binom{l_i}{2} \leq 2 \frac{(u-1)n(n-1)}{n} = 2(u-1)(n-1)$$

Da $\sum_{i=0}^{n-1} (l_i)^2 - n$ nie negativ sein kann, kann höchstens die Hälfte dieser Ausdrücke größer als der doppelte Mittelwert sein. Es gilt also für mindestens die Hälfte der Werte a von 1 bis $u-1$ die folgende Gleichung:

$$\sum_{i=0}^{n-1} (l_i)^2 - n \leq 4(n-1) \implies \sum_{i=0}^{n-1} (l_i)^2 < 5n$$

$$\sum_{a=1}^{u-1} \Big(\sum_{i=0}^{2n(n-1)} (l_i)^2 - n \Big) = 2 \sum_{a=1}^{u-1} \sum_{i=0}^{2n(n-1)} \binom{l_i}{2} \leq 2 \frac{(u-1)n(n-1)}{2n(n-1)+1} < u-1$$

Mit der selben Argumentation folgt daraus die zweite Behauptung des Satzes. ■

Ein beliebig gewählter Wert für a liefert damit mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ eine für dieses Verfahren verwendbare FKS- Hashfunktion.
Durch einen weiteren Transformationsschritt ist es möglich, das deterministische FKS Verfahren so zu modifizieren, dass der Konstruktionsaufwand auf $O(n^3 \ln u)$ sinkt. Falls u kleiner als $n^2 \ln u$ ist, ist diese Schranke schon vom ursprünglichen Verfahren eingehalten. Ansonsten wird versucht, eine Primzahl p, die kleiner als $n^2 \ln u$ ist und für die die Abbildung $x \mapsto x \mod p$ auf S injektiv ist, zu finden. Mit Hilfe dieser Funktion wird das bestehende Universum auf ein neues Universum der Größe p abgebildet, für das analog zum ursprünglichen FKS Verfahren vorgegangen wird. Der Konstruktionsaufwand beträgt dann $O(np) = O(n^3 \ln u)$. Der folgende Satz gibt Auskunft über die Existenz einer solchen Primzahl im asymptotischen Fall.

Satz 6.4 ([10, 20])
Es gibt bis auf höchstens endlich viele Ausnahmen stets eine Primzahl p, *die den Anforderungen des modifizierten FKS Verfahrens entspricht.*

Beweis:
Für $x_i \neq x_j$ ist die Bedingung $x_i \mod p \neq x_j \mod p$ äquivalent dazu, dass $p \nmid x_i - x_j$ gilt. Demnach ist es hinreichend, ein p zu finden, das das Produkt $t := \prod_{i<j}(x_i - x_j)$ nicht teilt. Durch einfache Abschätzung erhält man $\ln|t| \leq \binom{n}{2} \ln u \sim \frac{n^2}{2} \ln u$. Die Existenz von p ist sichergestellt, wenn das Produkt aller in Frage kommenden Primzahlen größer als t ist. Im Zusammenhang mit dem Primzahlsatz (siehe [23]) gilt $\vartheta(x) := \ln\left(\prod_{q \leq x, q Primz.} q\right) \sim x$. ■

Im Laufe der Zeit wurden weitere Variationen dieses Verfahrens vorgeschlagen. In [13] wird eine Erweiterung vorgestellt, die es ermöglicht dynamische Verzeichnisse zu erstellen, die auch Lösch- und Einfügeoperationen unterstützen. Darauf wird aber hier nicht näher eingegangen, denn das als nächstes vorgestellte Verfahren realisiert derartige Operationen einfacher.

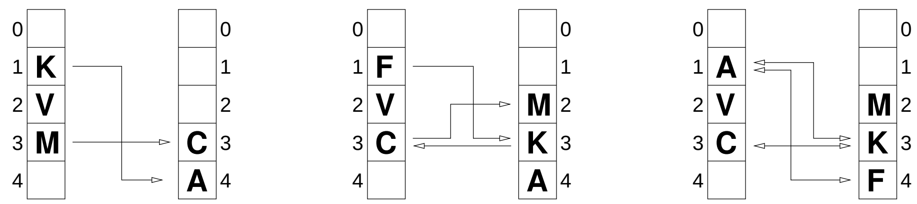
6.3 Cuckoo Hashing

Bei dem von Pagh und Rodler in [37] eingeführten Cuckoo Hashing handelt es sich um ein im Vergleich zu den meisten anderen Ansätzen für perfektes Hashing sehr einfaches Verfahren. Es benötigt zwei Tabellen T_1 und T_2 der Größe m, deren Speicherplätze wie üblich mit $0, 1, 2, \ldots, m-1$ bezeichnet werden. Für jede der Tabellen gibt es eine Hashfunktion h_1 bzw. h_2, die das Universum U auf die jeweilige Tabelle abbildet. Ein Schlüssel S kann in jeder Tabelle an genau einer Position, nämlich $h_1(S)$ bzw. $h_2(S)$ gespeichert werden. Er wird jedoch stets nur an einer der beiden möglichen Stellen gespeichert. Beim Einfügen des Schlüssels S wird zuerst geprüft, ob $h_1(S)$ frei ist. Wenn ja, wird der Schlüssel an der Position eingefügt, und der Einfügevorgang ist abgeschlossen. Falls an der Position $h_1(S)$ bereits ein Schlüssel T gespeichert war, wird S trotzdem in dieser Position eingefügt und für T wird ein neuer Platz gesucht. Dabei wird T an der Position $h_2(T)$ eingefügt und ein eventuell dort vorhandener Schlüssel entfernt, dieser wird anschließend in T_1 eingefügt usw.
Natürlich kann bei diesem Verfahren eine Endlosschleife auftreten, deshalb wird die maximale Anzahl der erlaubten Schritte bei einem Einfügevorgang (in Abhängigkeit von n) begrenzt. Wird diese Begrenzung überschritten, ist das Verfahren gescheitert, und es müssen zwei neue Hashfunktionen gewählt werden. Das folgende Beispiel veranschaulicht das Verfahren:

Beispiel 24 *Es werden zwei Tabellen verwendet, die je* 5 *Einträge aufnehmen können. Als Hashfunktionen werden* $h_1(S) = (S \mod 26) \mod 5$ *und* $h_2(S) = (2S + 7 \mod 26) \mod 5$ *verwendet.*

Information	C	A	K	V	M	F	H
Schlüssel	3	1	11	22	13	6	8
1. Hashwert	3	1	1	2	3	1	3
2. Hashwert	3	4	3	0	2	4	3

Fügt man die ersten fünf Werte der Reihe nach in ein anfangs leeres Tabellenpaar ein, so entsteht die im ersten Bild dargestellte Situation. Die Pfeile geben dabei die Positionsänderungen im Rahmen von Einfügeoperationen an.



Das zweite Bild veranschaulicht die Veränderungen, die sich durch zusätzliches Einfügen von F ergeben. Versucht man anschließend auch noch H einzufügen, entsteht eine Endlosschleife, was im letzten Bild angedeutet wird.

Cuckoo Hashing ermöglicht eine einfache Umsetzung von Such-, Einfüge- und Löschoperationen. Der Einfachheit halber wurde in folgender Realisierung auf eine Unterscheidung von Schlüssel und Information verzichtet.

Algorithmus N: `search(S)`
Suche für Cuckoo Hashing

```
if (T_1[h_1(S)] == S)
    return 1
if (T_2[h_2(S)] == S)
    return 2
return 0
```

Bei der folgenden Umsetzung der Suchoperation wird die Konstante `EMPTY` verwendet um einen leeren Eintrag zu kennzeichnen. Klarerweise muss `EMPTY` von allen möglichen Schlüsseln verschieden sein. Falls der Algorithmus nach `MAXLOOP` Iterationen immer noch nicht erfolgreich beendet werden konnte, ist mit hoher Wahrscheinlichkeit eine Endlosschleife aufgetreten. Das Element konnte daher nicht erfolgreich eingefügt werden, und es muss der gesamte Aufbau mit zwei neuen Hashfunktionen wiederholt werden.

Algorithmus O: `insert(S,I)`
Einfügen in eine Tabelle mit Cuckoo Hashing

```
if (search(S) != 0)
    return true
q = S                                // q: Element, das in T_1 eingefügt werden soll
for (i=1; i<=MAXLOOP; i++)
    p = T_1[h_1(q)]                  // p: Element, das in T_2 eingefügt werden soll
    T_1[h_1(q)] = q                  // q wird in T_1 eingefügt, verdrängt p
    if (p == EMPTY)                  // Einfügen abgeschlossen, falls p leerer Eintrag
        return true
    q = T_2[h_2(p)]
    T_2[h_2(p)] = p                  // p wird in T_2 eingefügt, verdrängt q
    if (q == EMPTY)                  // Einfügen abgeschlossen, falls q leerer Eintrag
        return true
return false
```

Algorithmus P: `delete(S)`
Löschen eines Eintrages bei Cuckoo Hashing

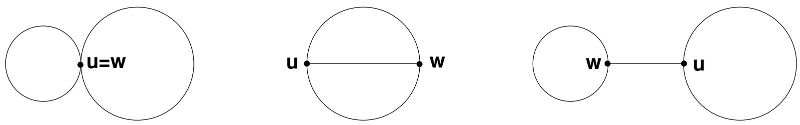
```
p = search(S)
if (p == 1)
    T_1[h_1(S)] = EMPTY
    return true
if (p == 2)
    T_2[h_2(S)] = EMPTY
    return true
return false
```

Analyse

Das ursprünglich von Pagh und Rodler [37] vorgeschlagene Verfahren setzt die Verwendung einer speziellen Familie von Hashfunktionen voraus. Es sind zwar geeignete Familien bekannt, die eine Auswertung in konstanter Zeit ermöglichen, diese sind jedoch für praktische Anwendungen zu aufwendig, wie die Experimente in [37] belegen. Ein für praktische Anwendungen vielversprechenderer Ansatz stammt von Devroye und Morin [12], der hier verfolgt wird. Im weiteren wird vorausgesetzt, dass die mit einem Paar von Hashfunktionen berechneten Hashwerte alle unabhängig voneinander und in der Menge $\{0, 1, 2, \ldots, m-1\}$ gleichverteilt sind. Falls ein neues Paar von Hashfunktionen gewählt werden muss, so sollen die neuen Hashwerte weiters unabhängig von allen bisherigen sein. Dies kann für praktische Zwecke annähernd erreicht werden, analog dazu, dass Double Hashing eine gute Approximation von Uniform Hashing darstellt.
Für gegebene Hashfunktionen h_1 und h_2 kann man nun einen paaren Graphen G definieren, indem man die Tabellenplätze als Knotenmenge auffasst und je zwei Knoten K_1, K_2 genau dann durch eine Kante verbindet, wenn für einen der n einzutragenden Schlüssel S die Bedingungen $h_1(S) = K_1$ und $h_2(S) = K_2$ erfüllt sind. Falls es mehrere solche Schlüssel gibt, wird die Kante entsprechend oft gezeichnet, d.h. Mehrfachkanten sind erlaubt. Mit Hilfe der Zusammenhangskomponenten dieses Graphen kann man nun erkennen, ob das zugehörige Cuckoo Hashing erfolgreich ist. Dies ist genau dann der Fall, wenn jede Zusammenhangskomponente höchstens einen Kreis enthält:

- Eine Zusammenhangskomponente mit k Knoten, die mehr als einen Kreis enthält, hat mindestens $k + 1$ Kanten. Da es nicht möglich ist, mehr als k Schlüssel auf k Plätze zu verteilen, schlägt Cuckoo Hashing in diesem Fall fehl.

- Für eine Komponente mit höchstens einem Kreis sei W die Kantenfolge im Graphen, die beim Einfügen eines Schlüssels durchlaufen wird. Falls W keinen Knoten mehrfach enthält, ist kein Kreis in W enthalten. Sonst sei u der erste wiederholt vorkommende Knoten und W besitzt einen Kreis C, der u enthält. Die Kante, die in W auf das zweite Auftreten von u folgt, kann nicht in C liegen, denn der Schlüssel, der in u gespeichert ist, hat seinen zweiten möglichen Speicherplatz außerhalb von C. Sei nun $\overline{W}$ der Teil von W, der direkt nach dem ersten Auftreten von u beginnt. Falls $\overline{W}$ einen Knoten w mehrfach enthält, so kann w in C liegen oder nicht. Die folgende Graphik zeigt, dass dadurch stets ein Widerspruch auftritt:



  Deshalb enthalten sowohl $W \setminus \overline{W}$ als auch $\overline{W}$ keine Knoten mehrfach. Insgesamt enthält W höchstens doppelt so viele Kanten wie die Komponente Knoten enthält und damit keinesfalls eine Endlosschleife.

Wie groß ist nun die Wahrscheinlichkeit, dass ein auf die oben angegebene Art konstruierter Graph eine Komponente mit mehr als einem Kreis enthält? Um dies festzustellen, wurden für verschiedene Lastfaktoren und Tabellengrößen je 10^4 Zufallsexperimente durchgeführt. Dabei wurden die Hashfunktionen zufällig aus der im Satz 5.1 eingeführten universellen Menge ermittelt, weshalb nur Primzahlen als Tabellengrößen herangezogen wurden. Alle benötigten Zufallszahlen wurden mit dem im Kapitel 2.3.1 beschriebenen KISS Generator erzeugt. Die folgende Tabelle gibt den Anteil der Graphen an, die mindestens eine Komponente mit mehr als einem Kreis enthalten:

	$n = 10^4$	$n = 5 \cdot 10^4$	$n = 10^5$	$n = 2 \cdot 10^5$
$\alpha_1 \approx 0.385$	$m = 13001$ 0%	$m = 65003$ 0%	$m = 130003$ 0.06%	$m = 260003$ 0.12%
$\alpha_2 \approx 0.455$	$m = 11003$ 0%	$m = 55001$ 0.15%	$m = 110017$ 0.21%	$m = 220009$ 0.84%
$\alpha_3 \approx 0.495$	$m = 10103$ 7.71%	$m = 50503$ 8.38%	$m = 101009$ 8.41%	$m = 202001$ 11.33%
$\alpha_4 \approx 0.5$	$m = 10009$ 13.14%	$m = 50023$ 15.15%	$m = 100019$ 18.44%	$m = 200009$ 23.31%
$\alpha_5 \approx 0.5$ $\alpha_5 > \alpha_4$	$m = 10007$ 11.47%	$m = 50021$ 17.53%	$m = 100003$ 17.37%	$m = 200003$ 24,76%

Diese Resultate zeigen, dass der Anteil der nicht für das Verfahren geeigneten Graphen relativ hoch ist.

Satz 6.5 (Cuckoo Hashing [11])
Sei $m = (1+\varepsilon)n$ für ein $\varepsilon \in [\varepsilon_1, M]$ mit festen $\varepsilon_1 > 0$ und $M < \infty$. Wenn weiters $C > 2/R(\varepsilon_1)$, wobei $R(t) := \ln(1+t) - \frac{t}{1+t}$ erfüllt ist und der Wert $C \ln n$ als `MAXLOOP` *verwendet wird, dann ist die Wahrscheinlichkeit, dass das Verfahren in einem Versuch fehlschlägt, höchstens $O(\ln^4 n/n)$. Die erwartete Zeit für den erfolgreichen Aufbau einer Hashtabelle beträgt dann insgesamt $O(n)$.*

Beweis: (skizziert)
Der Beweis besteht im wesentlichen aus zwei Schritten. Zuerst bestimmt man eine Schranke für die größte auftretende zusammenhängende Komponente im Cuckoo Graphen. Diese kann durch Untersuchung eines entsprechenden Galton-Watson Verzweigungsprozesses erhalten werden. Der zweite Schritt besteht darin, den Anteil der Graphen nach oben abzuschätzen, der mindestens eine Komponente mit mehr als einem Kreis enthält. Devroye und Morin geben in [12] hierfür die Schranke $O(1/n)$ an. Leider enthält jedoch der im Artikel angegebene Beweis dieser Abschätzung einen Fehler, wie im Anhang A ausgeführt wird. Eine Vorversion [11] dieses Artikel enthält jedoch einen Beweis, der zeigt, dass der Anteil höchstens $O(\ln^4 n/n)$ beträgt. Weitere Details stehen in [11, 12]. ∎

Kapitel 7

Experimente am Computer

Die in den bisherigen Kapiteln gewonnen Analysen können oft nicht direkt angewendet werden. Mögliche Gründe dafür sind zum Beispiel:

- Theoretische Annahmen über die Verteilung der Schlüssel sind nicht exakt erfüllt.
- Die verwendeten Hashfunktionen haben nicht die geforderten Eigenschaften.
- Aussagen gelten nur für den asymptotischen Fall.
- Es sind nur unvollständige Aussagen bekannt.
- Unterschiedliche benötigte Programmiertechniken bewirken andere Laufzeiten, auch wenn theoretisch die selbe Anzahl an Vergleichen notwendig ist.

Dieser Abschnitt soll vor allem eine praktische Bestätigung der theoretisch ermittelten Resultate bringen und theoretisch nicht erfasste Eigenschaften untersuchen. Ähnliche Untersuchungen finden sich z.B. in [6, 26, 29, 37, 38].

7.1 Hard- und Software

Die einzelnen Algorithmen wurden -angelehnt an die in der Arbeit angegebenen Pseudocodes- in Visual C++ 6 programmiert. Diese sind dabei größtenteils in Templateklassen umgesetzt, die eine flexible Verwendung ermöglichen. So sind z.B. sowohl natürliche Zahlen als auch Zeichenketten als Schlüssel möglich. Ausgehend von einer als Schnittstelle dienenden rein virtuellen Basisklasse entstand so eine vielseitig verwendbare Bibliothek, weitere Details im Anhang C.
Ein dialogfeldbasiertes Testprogramm ermöglicht die komfortable Verwendung der Bibliothek für die Experimente und auch zur Kontrolle der in dieser Arbeit angegebenen Beispiele für Hashtabellen.
Die hier angeführten Resultate wurden auf einem PC, der mit einem AMD Sempron 2500+ Prozessor und 256MB RAM Arbeitsspeicher ausgestattet ist, unter dem Betriebssystem Windows XP ermittelt.
Die verwendete Hard- und Software hat auch einen starken Einfluss auf die Wahl der Parameter für die Experimente ausgeübt:

- Laufzeitvergleiche ermöglichen einen direkten Vergleich der Geschwindigkeiten der einzelnen Algorithmen. Die Messung der Zeiten erfolgt dabei über die interne Systemzeit, welche in Millisekunden angegeben ist. Die tatsächliche Genauigkeit ist jedoch deutlich geringer, so dass ein Fehler von 10 ms je Messung zu befürchten ist. Da selbst alle Einträge einer

Hashtabelle mit 10^5 Einträgen oft in etwa 50ms gesucht werden können, war es notwendig, mehrere gleich aufgebaute Tabellen mit unterschiedlichen Einträgen hintereinander zu berechnen.

- Werden zu große Tabellen benutzt, wird am verwendeten System so viel Speicher benötigt, dass Daten in die Auslagerungsdatei abgelegt werden, was zu (punktuell auftretenden) Laufzeiteinbußen führt und die Resultate verfälscht.

7.2 Untersuchungen mit zufälligen Daten

Diese Untersuchungen erfolgen mit Daten, die dem Modell aus Kapitel 1.8 möglichst gut entsprechen. Deshalb wurden mit dem KISS Generator erstellte Zufallszahlen und die Divisionsmethode für die Hashfunktionen verwendet. Letztere erfordert, dass die Tabellengrößen stets Primzahlen sind. Die im Weiteren angeführten Graphen entstanden alle durch Berechnungen an 25 Tabellen mit je 10^5 Einträgen, wobei jeweils ca. 25 unterschiedliche Tabellengrößen herangezogen wurden. Die angegebenen Zeiten beziehen sich daher auf insgesamt 2.5 Millionen Zugriffe. Um die durchschnittlich benötigte Anzahl an Vergleichen für eine erfolglose Suche zu bestimmen, wurden je Tabelle 1000 zufällig bestimmte, nicht in der Tabelle enthaltene Elemente gesucht.

7.2.1 Hashverfahren mit Verkettung

Von den in der Arbeit behandelten Verfahren mit Verkettung lassen sich nur separate Verkettung und direkte Verkettung, jeweils mit unsortierter oder sortierter Einfügereihenfolge, unmittelbar vergleichen. Coalesced Hashing und das 2-left Schema werden später behandelt.

Laufzeit von Einfügeoperationen bei Hashing mit Verkettung

Abbildung 7.1 zeigt die aus den Daten erstellten Ausgleichsgeraden. Der Geschwindigkeitsvorteil von direkter Verkettung im Vergleich zu separater Verkettung ist vor allem darauf zurückzuführen, dass Zugriffe auf leere Tabellenbereiche schneller erfolgen. Weiters muss bei separater Verkettung mehr Speicher dynamisch allokiert werden, was zusätzliche Zeit benötigt. Ansonsten verlaufen die Geraden mit Steigungen von 160 bzw. 175ms annähernd parallel. Sortiertes Einfügen bewirkt einen deutlich flacheren Anstieg, da erfolglose Suchvorgänge, wie sie vor Einfügeoperationen durchgeführt werden, billiger werden. Die Resultate aus Kapitel 3 lassen eine Halbierung der Steigung erwarten, dies wird durch den hier auftretenden Wert von 75ms sogar etwas unterschritten.

Laufzeit erfolgreicher Suchoperationen bei Hashing mit Verkettung

Die theoretische Analyse lässt für alle drei hier untersuchten Verfahren die gleiche Laufzeit erwarten. Abbildung 7.2 zeigt jedoch ein deutlich davon abweichendes Resultat. Die Ausgleichsgeraden von direkter und separater Verkettung verlaufen zwar annähernd parallel (Steigung 142 bzw. 157ms), jedoch weist die zur sortierten Tabelle gehörende Gerade eine deutlich höhere Steigung (196ms) auf. Dies kann nur durch den geringfügig komplizierteren Suchalgorithmus bedingt sein, da stets zusätzliche Vergleiche durchgeführt werden müssen. Diese bringen jedoch jetzt keinen Vorteil, da das gesuchte Element in der Tabelle enthalten ist, und können nie zu einer Zeitersparnis führen.
Der etwa konstante Vorteil von direkter Verkettung im Vergleich zu separater Verkettung lässt sich durch den schnelleren Zugriff auf in der Tabelle selbst gespeicherte Daten zurückführen.

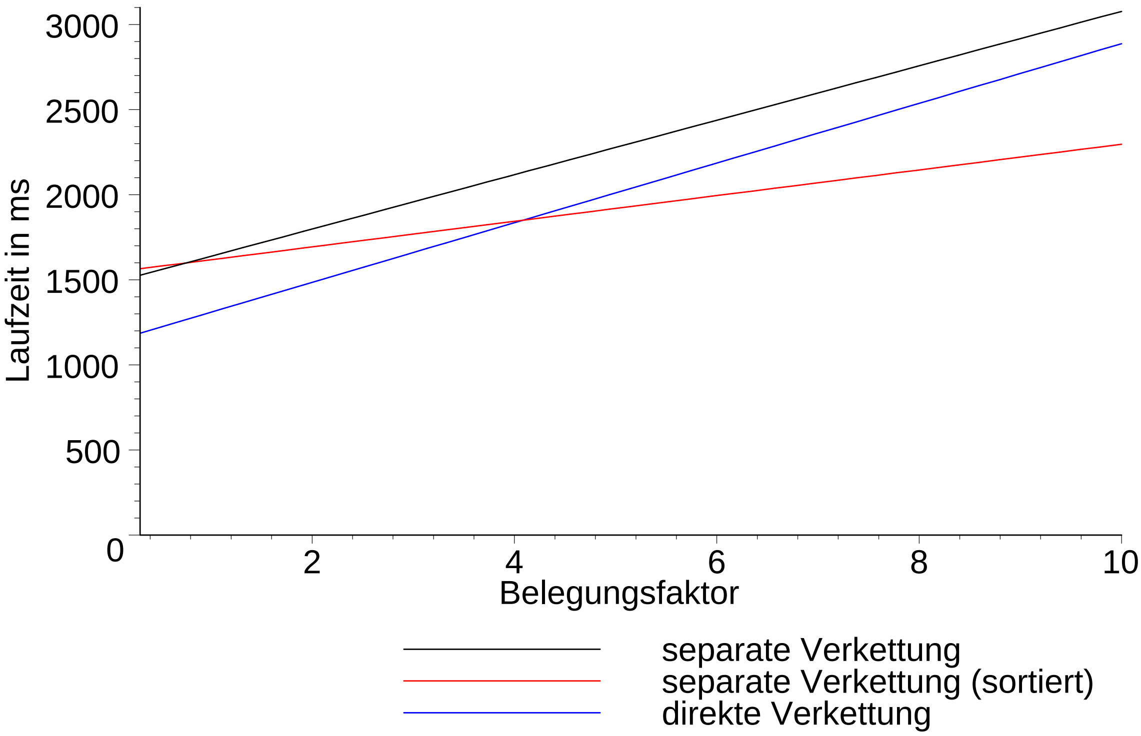


Abbildung 7.1: Laufzeit von Einfügeoperationen bei Hashing mit Verkettung

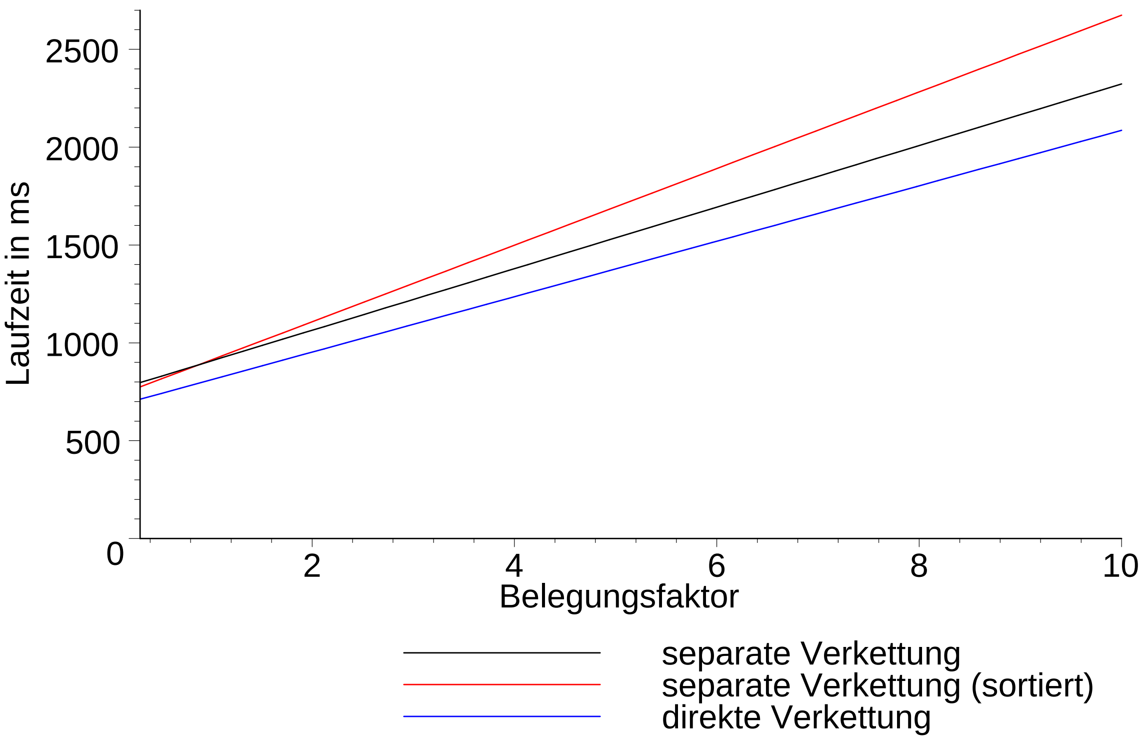


Abbildung 7.2: Laufzeit erfolgreicher Suchoperationen bei Hashing mit Verkettung

Durchschnittlich benötigte Anzahl an Vergleichen für erfolgreiche Suche bei Hashing mit Verkettung

Die Daten aus Tabelle 7.1 bestätigen, dass alle drei untersuchten Verfahren sich nicht wesentlich in der Anzahl der benötigten Vergleiche unterscheiden und die oben beobachteten Abweichungen in der Laufzeit nicht daraus resultieren können. Ähnlich geringe Abweichungen ergeben sich für die Varianzen, wie Tabelle 7.2 festhält.

		separate Verkettung		sep. Verk. (sort.)		direkte Verkettung	
α	$\mathbb{E}C_{10^5}^{[r]}$	Ergeb.	Abw.	Ergeb.	Abw.	Ergeb.	Abw.
0.4	1.2	1.2	−0.03%	1.2	−0.01%	1.2	−0.01%
0.7	1.35	1.35	0.03%	1.35	0.01%	1.35	0.00%
1.0	1.5	1.499	−0.04%	1.5	0.01%	1.5	−0.01%
1.6	1.8	1.8	0.03%	1.799	−0.03%	1.8	−0.01%
2.5	2.25	2.25	0.00%	2.25	0.01%	2.25	0.02%
4.0	2.999	2.998	−0.04%	2.999	0.00%	2.998	−0.02%
5.5	3.749	3.747	−0.04%	3.751	0.06%	3.748	−0.02%
7.0	4.498	4.499	0.01%	4.498	−0.00%	4.499	0.01%
8.5	5.246	5.245	−0.01%	5.245	−0.02%	5.244	−0.04%
10	5.996	5.997	0.01%	5.996	−0.01%	5.999	0.04%

Tabelle 7.1: Stichprobenmittel: Vergleiche für erfolgreiche Suche

		separate Verkettung		sep. Verk. (sort.)		direkte Verkettung	
α	$\mathbb{V}C_{10^5}^{[r]}$	Ergeb.	Abw.	Ergeb.	Abw.	Ergeb.	Abw.
0.4	0.2133	0.2129	−0.21%	0.2131	−0.11%	0.2129	−0.18%
0.7	0.3908	0.3913	0.12%	0.3908	−0.01%	0.3912	0.10%
1.0	0.5833	0.5832	−0.02%	0.5832	−0.02%	0.5838	0.09%
1.6	1.013	1.015	0.18%	1.013	−0.07%	1.012	−0.16%
2.5	1.77	1.771	0.04%	1.775	0.24%	1.773	0.15%
4.0	3.331	3.324	−0.20%	3.33	−0.01%	3.326	−0.14%
5.5	5.267	5.264	−0.06%	5.283	0.30%	5.263	−0.07%
7.0	7.577	7.579	0.03%	7.576	−0.01%	7.576	−0.02%
8.5	10.25	10.26	0.02%	10.25	−0.08%	10.23	−0.21%
10	13.32	13.32	0.03%	13.31	−0.04%	13.34	0.17%

Tabelle 7.2: Stichprobenvarianz: Vergleiche für erfolgreiche Suche

Durchschnittlich benötigte Anzahl an Vergleichen für erfolglose Suche bei Hashing mit Verkettung

Wie Tabelle 7.3 belegt, stimmen die in Abbildung 7.3 gezeigten Kurven fast exakt mit den theoretisch ermittelten Werten überein. Auch bei den in Tabelle 7.4 angeführten Varianzen sind nur geringe Abweichungen von den theoretischen Werten zu verzeichnen. Insgesamt scheint das Modell aus Kapitel 1.8 für „gute" Pseudozufallszahlen anwendbar.

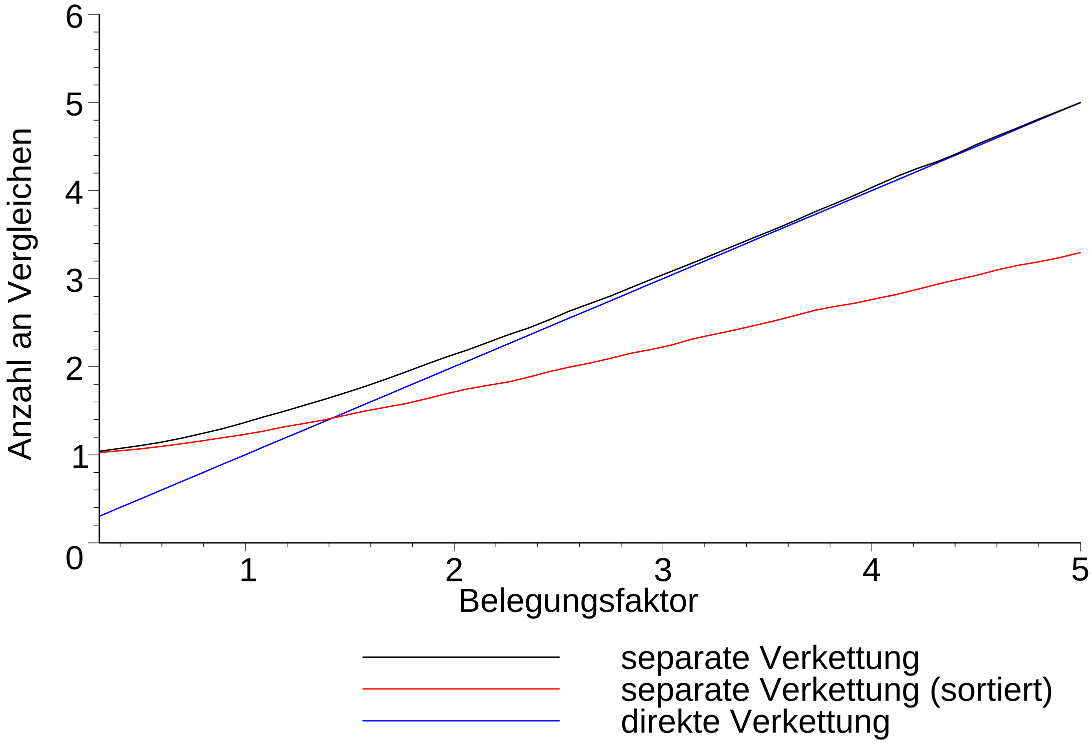


Abbildung 7.3: Stichprobenmittel: Vergleiche für erfolglose Suche

	sep. Verkettung			sep. Verk. (sort.)			dir. Verkettung		
α	$\mathbb{E}C_{10^5}^{[l]}$	Ergeb.	Abw.	$\mathbb{E}C_{10^5}^{[l]}$	Ergeb.	Abw.	$\mathbb{E}C_{10^5}^{[l]}$	Ergeb.	Abw.
0.4	1.07	1.072	0.11%	1.046	1.047	0.09%	0.4	0.4045	−1.13%
0.7	1.197	1.198	0.16%	1.127	1.129	0.13%	0.7	0.7026	−0.37%
1.0	1.368	1.362	−0.45%	1.236	1.234	−0.15%	1	0.9944	0.56%
1.6	1.802	1.802	0.01%	1.503	1.516	0.89%	1.6	1.606	−0.39%
2.5	2.582	2.585	0.14%	1.965	1.951	−0.67%	2.499	2.512	−0.49%
4.0	4.016	4.028	0.29%	2.772	2.785	0.47%	3.998	3.976	0.55%
5.5	5.501	5.492	−0.18%	3.572	3.574	0.08%	5.497	5.519	−0.40%
7.0	6.997	7.016	0.27%	4.356	4.36	0.08%	6.996	7.03	−0.47%
8.5	8.491	8.456	−0.41%	5.128	5.123	−0.10%	8.491	8.499	−0.09%
10	9.993	10.03	0.39%	5.896	5.879	−0.29%	9.993	9.993	0.00%

Tabelle 7.3: Stichprobenmittel: Vergleiche für erfolglose Suche

	sep. Verkettung			sep. sort.	dir. Verkettung		
α	$\mathbb{V}C_{10^5}^{[l]}$	Ergeb.	Abw.	Ergeb.	$\mathbb{V}C_{10^5}^{[l]}$	Ergeb.	Abw.
0.4	0.0847	0.0866	2.17%	0.0546	0.4	0.4032	−0.79%
0.7	0.2547	0.2596	1.90%	0.161	0.6999	0.7037	−0.53%
1.0	0.4968	0.4806	−3.26%	0.2997	1	0.9929	0.71%
1.6	1.115	1.117	0.15%	0.721	1.6	1.606	−0.38%
2.5	2.164	2.166	0.06%	1.404	2.4994	2.498	0.06%
4.0	3.869	3.886	0.44%	2.902	3.9979	3.977	0.53%
5.5	5.456	5.496	0.73%	4.762	5.4972	5.575	−1.42%
7.0	6.985	7.096	1.59%	7.089	6.9964	6.871	1.79%
8.5	8.488	8.273	−2.53%	9.531	8.491	8.481	0.12%
10	9.992	9.911	−0.82%	12.42	9.9929	10.04	−0.46%

Tabelle 7.4: Stichprobenvarianz: Vergleiche für erfolglose Suche

Coalesced Hashing

Coalesced Hashing unterscheidet sich von den oben behandelten Hashverfahren dadurch, dass die Anzahl der maximal in die Hashtabelle aufnehmbaren Einträge begrenzt ist. Von besonderem Interesse ist die optimale Wahl der Kellergröße. Die Analyse des optimalen Kelleranteils liefert Ergebnisse, die an sich nur für den asymptotischen Fall gelten. Hier wird untersucht, ob diese Ergebnisse allgemein herangezogen werden können. Die Betrachtung der Laufzeit erwies sich dabei als ungeeignet, da die auftretenden Schwankungen zu groß waren, um genaue Resultate zu liefern. Deshalb wurde hier die Anzahl der benötigten Vergleiche untersucht. Tabelle 7.5 dient zur Untersuchung der erfolgreichen Suche. Die für den jeweiligen Lastfaktor geringste durchschnittliche Anzahl an Vergleichen wurde dabei fett hervorgehoben. Betrachtet man den zugehörigen Kelleranteil, so ist dieser stets nahe am für den asymptotischen Fall bestimmten Wert. Analog untersucht Tabelle 7.6 erfolglose Suche, für die ebenfalls das asymptotische Optimum praktisch verwendbar scheint. Siehe auch [44].

α	opt. Wert	0%	3%	6%	9%	12%	15%	18%
0.25	2.3%	1.1366	**1.1290**	1.1326	1.1372	1.1417	1.1472	1.1522
0.40	4.8%	1.2328	1.2153	**1.2138**	1.2207	1.2268	1.2357	1.2441
0.50	6.7%	1.3042	1.2812	**1.2750**	1.2759	1.2840	1.2938	1.3049
0.65	9.4%	1.4260	1.3947	1.3819	**1.3778**	1.3800	1.3860	1.3973
0.75	11.2%	1.5169	1.4809	1.4654	1.4571	**1.4553**	1.4594	1.4673
0.85	12.7%	1.6199	1.5798	1.5584	1.5460	**1.5427**	1.5438	1.5486
0.90	13.5%	1.6770	1.6329	1.6090	1.5948	**1.5890**	1.5902	1.5953
0.95	14.1%	1.7347	1.6880	1.6634	1.6481	1.6428	**1.6401**	1.6446
0.99	14.6%	1.7853	1.7358	1.7090	1.6941	1.6858	**1.6826**	1.6879
1.00	14.7%	1.7968	1.7468	1.7186	1.7040	1.6952	**1.6927**	1.6952

Tabelle 7.5: Stichprobenmittel: Vergleiche für erfolgreiche Suche bei **Coalesced Hashing**

α	opt. Wert	0%	4%	8%	12%	16%	20%	24%
0.25	2.5%	1.0378	**1.0312**	1.0348	1.0363	1.0376	1.0433	1.0474
0.40	5.8%	1.1062	**1.0798**	1.0801	1.0912	1.0935	1.1052	1.1145
0.50	8.6%	1.1863	1.1440	**1.1290**	1.1348	1.1444	1.1583	1.1744
0.65	13.1%	1.3392	1.2820	1.2425	**1.2388**	1.2451	1.2586	1.2818
0.75	16.1%	1.4985	1.4132	1.3630	1.3510	**1.3405**	1.3427	1.3626
0.85	19.0%	1.6894	1.5849	1.5174	1.4930	**1.4790**	1.4792	1.4830
0.90	20.2%	1.8152	1.7000	1.6352	1.5897	1.5737	**1.5680**	1.5721
0.95	21.1%	1.9531	1.8332	1.7461	1.7043	1.6834	**1.6650**	1.6864
0.99	21.7%	2.0643	1.9293	1.8551	1.8068	1.7774	**1.7523**	1.7576
1.00	21.8%	2.0931	1.9645	1.8865	1.8400	1.7984	**1.7822**	1.7878

Tabelle 7.6: Stichprobenmittel: Vergleiche für erfolglose Suche bei **Coalesced Hashing**

7.2.2 Offene Hashverfahren

In diesem Abschnitt werden die vier wichtigsten Hashverfahren mit offener Adressierung, nämlich Lineares Sondieren, Double Hashing, Brents Variation von Double Hashing und Robin Hood Hashing untersucht.

Laufzeit von Einfügeoperationen bei Hashing mit offener Adressierung

Lineares Sondieren und Double Hashing sind so konzipiert, dass der Aufwand eines Einfügevorgangs gleich dem Aufwand einer entsprechenden erfolglosen Suche ist. Im Unterschied dazu wird bei Brents Variation und Robin Hood Hashing im Vergleich zur erfolglosen Suche eventuell zusätzlicher Aufwand für eine Einfügeoperation benötigt. Abbildung 7.4 zeigt die erhaltenen Resultate. Für geringe Auslastungen unterscheiden sich die Verfahren nicht besonders stark, für hohe Auslastungen ist jedoch Double Hashing am schnellsten. Hierbei ist aber auch die verwendete sekundäre Hashfunktion mit großer Sorgfalt auszuwählen. Ist nämlich die sekundäre Hashfunktion so gewählt, dass nur wenige und kleine Werte auftreten, so ist das Verfahren genau so langsam wie lineares Sondieren.[1] Brents Variation liegt bei der Einfügeoperation in der Laufzeit stets deutlich hinter gewöhnlichem Double Hashing zurück. Robin Hood Hashing weist insgesamt das schlechteste Laufzeitverhalten auf.

Laufzeit erfolgreicher Suche bei Hashing mit offener Adressierung

Auch hier unterscheiden sich die einzelnen Verfahren in der Laufzeit für Tabellen mit geringen Auslastungen nur unwesentlich, wie Abbildung 7.5 darstellt. Während jedoch der Aufwand bei Brents Variation nahezu konstant bleibt, steigt der Aufwand bei den anderen Verfahren stark mit der Auslastung an. Im direkten Vergleich von Brents Variation und gewöhnlichem Double Hashing erkennt man, dass Brents Variation bei einer Auslastung von ca. 80% rentabel ist, wenn mindestens doppelt so oft erfolgreich gesucht, als in die Tabelle eingefügt wird, was bei praktischer Anwendung oft der Fall ist, vergl. [5]. Hashing mit linearer Sondierung ist für praktische Umsetzungen nicht empfehlenswert, da der geringe Vorteil der etwas einfacheren Implementierung die Nachteile in der Laufzeit bei hohen Auslastungen nicht aufwiegt.

[1] Hier wurde stets $1 + (S \mod (m-2)) \mod m$ als zweite Hashfunktion gewählt.

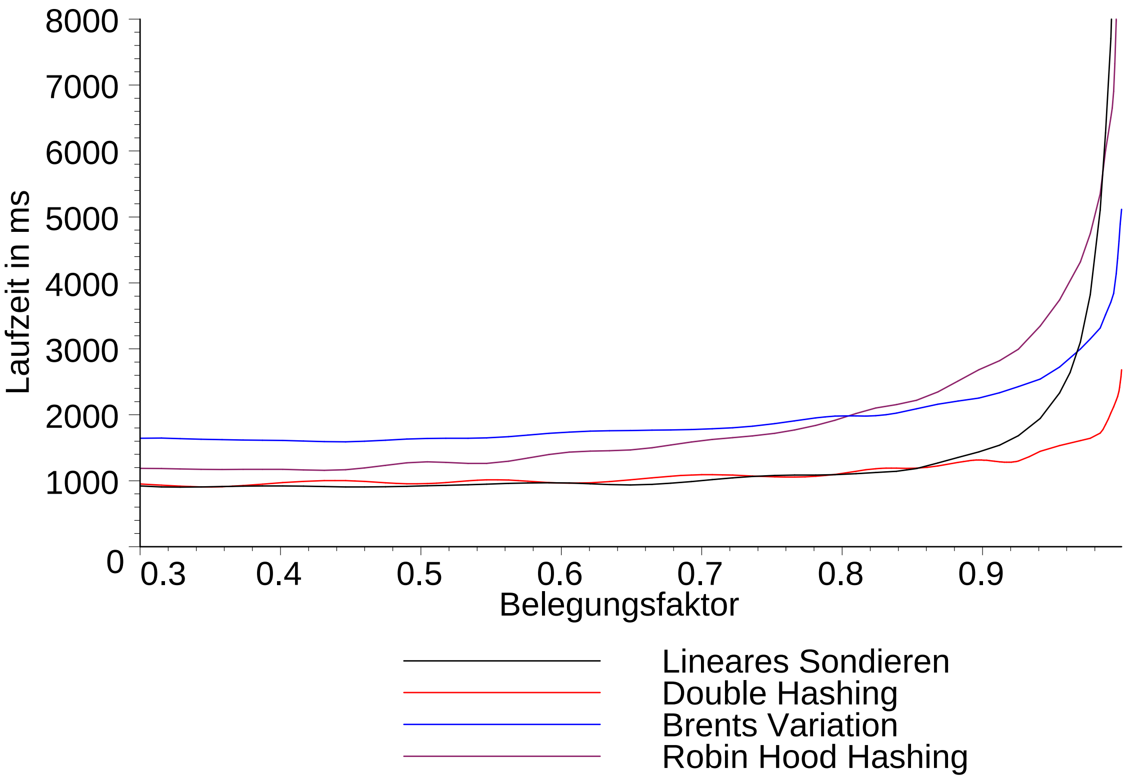


Abbildung 7.4: Laufzeit von Einfügeoperationen bei Hashing mit offener Adressierung

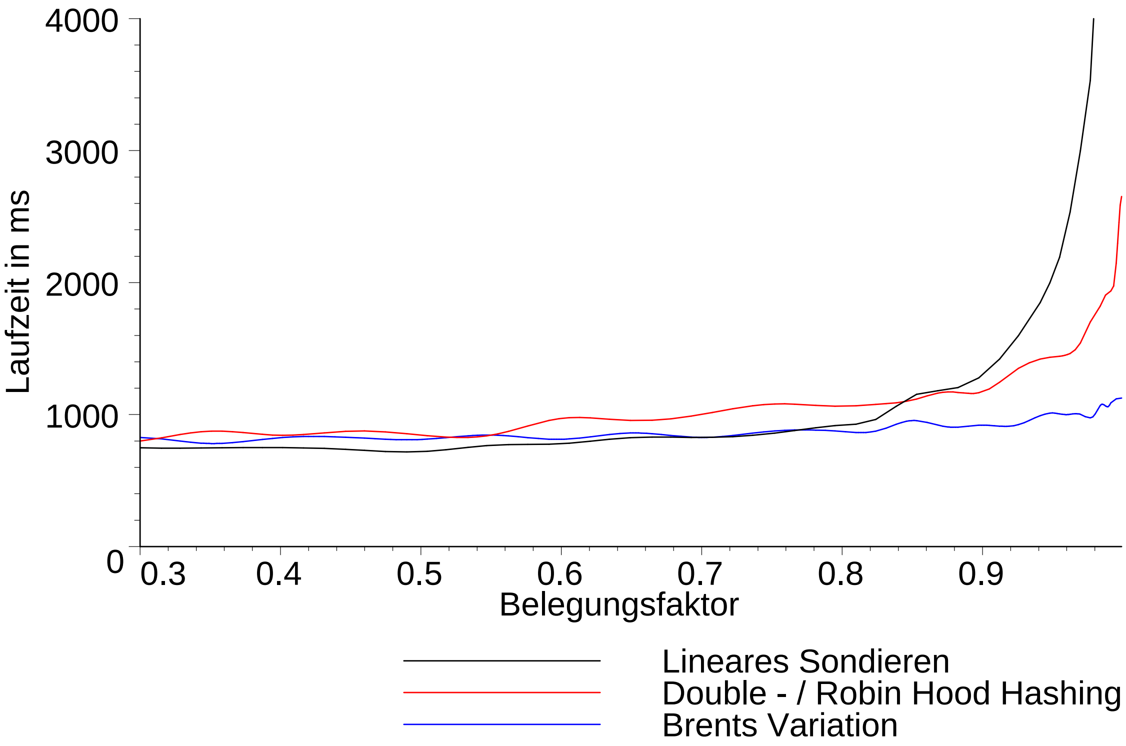


Abbildung 7.5: Laufzeit erfolgreicher Suche bei Hashing mit offener Adressierung

Durchschnittlich benötigte Anzahl an Vergleichen für erfolgreiche Suche bei Hashing mit offener Adressierung

Wie Tabelle 7.7 belegt, stimmen die in Abbildung 7.6 gezeigten Ergebnisse gut mit den theoretischen Resultaten überein. Insbesondere scheint die Annahme [25, 27, 38], dass Uniform Hashing von Double Hashing gut approximiert wird, gerechtfertigt. Weiters liegt bei Brents Variation die Anzahl der durchschnittlich benötigten Vergleiche stets unter der ermittelten Schranke von ca. 2, 5. Für einen Einfügevorgang werden jedoch stets etwas mehr Vergleiche als bei gewöhnlichem Double Hashing benötigt, stark wirkt sich dies allerdings erst bei hoher Auslastung aus. Der in Abbildung 7.4 ersichtliche größere Zeitbedarf ist demnach auch auf den komplizierteren Algorithmus zurückzuführen.
Während sich Double und Robin Hood Hashing beim Erwartungswert nicht unterscheiden, zeigt Tabelle 7.8 ein deutlich abweichendes Ergebnis bei der Varianz, die bei beim zweiten Algorithmus deutlich geringer ist und unter der Schranke [8] von 1.883 liegt. Auch Brents Variation bewirkt eine deutliche, aber nicht ganz so gravierende Reduktion der Varianz. Siehe weiters [5, 8, 22].

	Lineares Sondieren			Double - / R.H. Hashing			Brent	
α	$\mathbb{E}C_{10^5}^{[r]}$	Ergeb.	Abw.	$\mathbb{E}C_{10^5}^{[r]}$	Ergeb.	Abw.	Ergeb.	Einf.
0.250	1.167	1.167	0.02%	1.151	1.151	−0.02%	1.129	1.151
0.400	1.333	1.333	−0.04%	1.277	1.277	0.00%	1.218	1.279
0.500	1.5	1.502	0.10%	1.386	1.386	0.02%	1.287	1.392
0.650	1.928	1.928	−0.02%	1.615	1.614	−0.07%	1.415	1.635
0.750	2.5	2.497	−0.08%	1.848	1.85	0.07%	1.529	1.893
0.850	3.83	3.832	0.06%	2.231	2.231	−0.02%	1.69	2.336
0.900	5.492	5.523	0.56%	2.558	2.556	−0.06%	1.805	2.727
0.950	10.45	10.54	0.82%	3.152	3.153	0.00%	1.977	3.464
0.990	45.18	43.2	−4.37%	4.641	4.642	0.01%	2.246	5.341
0.999	198.2	155.6	−21.50%	6.886	6.883	−0.05%	2.417	8.052

Tabelle 7.7: Stichprobenmittel: Vergleiche für erfolgreiche Suche

	Lineares Sondieren		Double Hashing		Brent	R. Hood
α	$\mathbb{V}C_{10^5}^{[r]}$	Ergebnis	$\mathbb{V}C_{10^5}^{[r]}$	Ergebnis	Ergebnis	Ergebnis
0.250	0.2623	0.2635	0.1918	0.1921	0.1228	0.1334
0.400	0.7653	0.7642	0.4254	0.4256	0.2133	0.23
0.500	1.583	1.594	0.6919	0.6946	0.2924	0.3082
0.650	5.643	5.604	1.49	1.489	0.4665	0.4588
0.750	17.23	17.26	2.735	2.735	0.6626	0.6045
0.850	87.22	91.8	6.115	6.111	1.023	0.8209
0.900	306	315.3	10.89	10.84	1.359	0.9824
0.950	2500	2446	26.87	26.84	2.044	1.227
0.990	$134 \cdot 10^3$	$163 \cdot 10^3$	171.7	170.9	4.007	1.616
0.999		$3273 \cdot 10^3$	1889	1835	7.177	1.826

Tabelle 7.8: Stichprobenvarianz: Vergleiche für erfolgreiche Suche

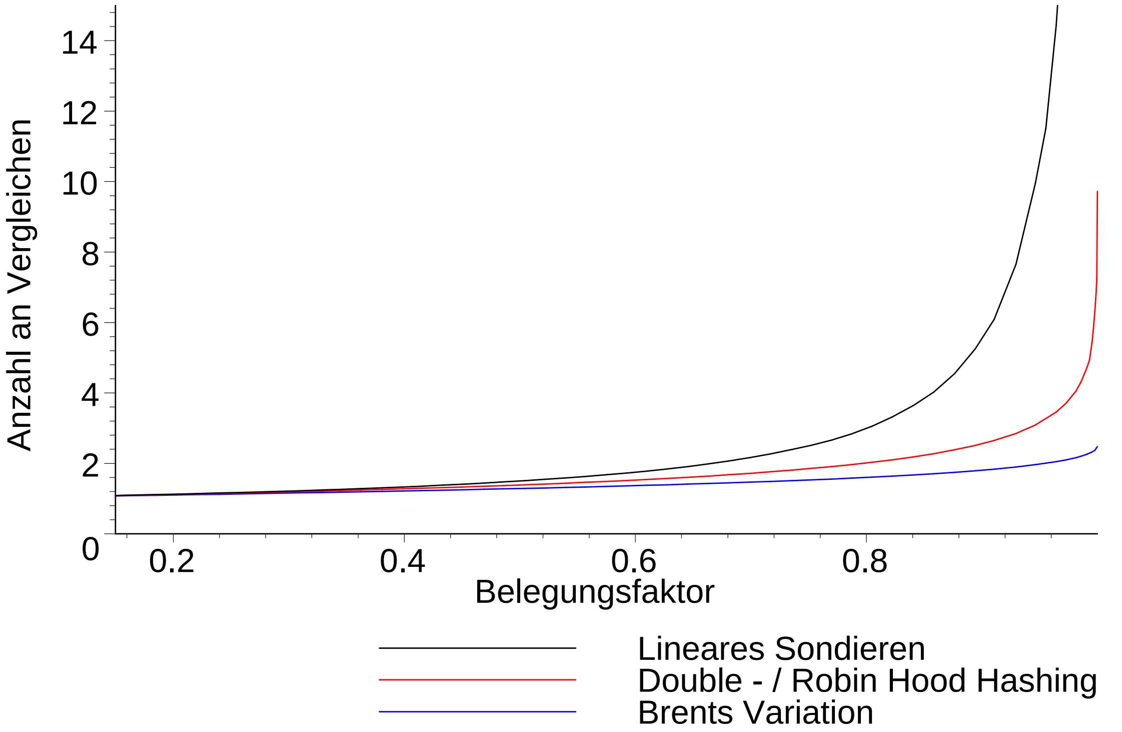


Abbildung 7.6: Stichprobenmittel: Vergleiche für erfolgreiche Suche

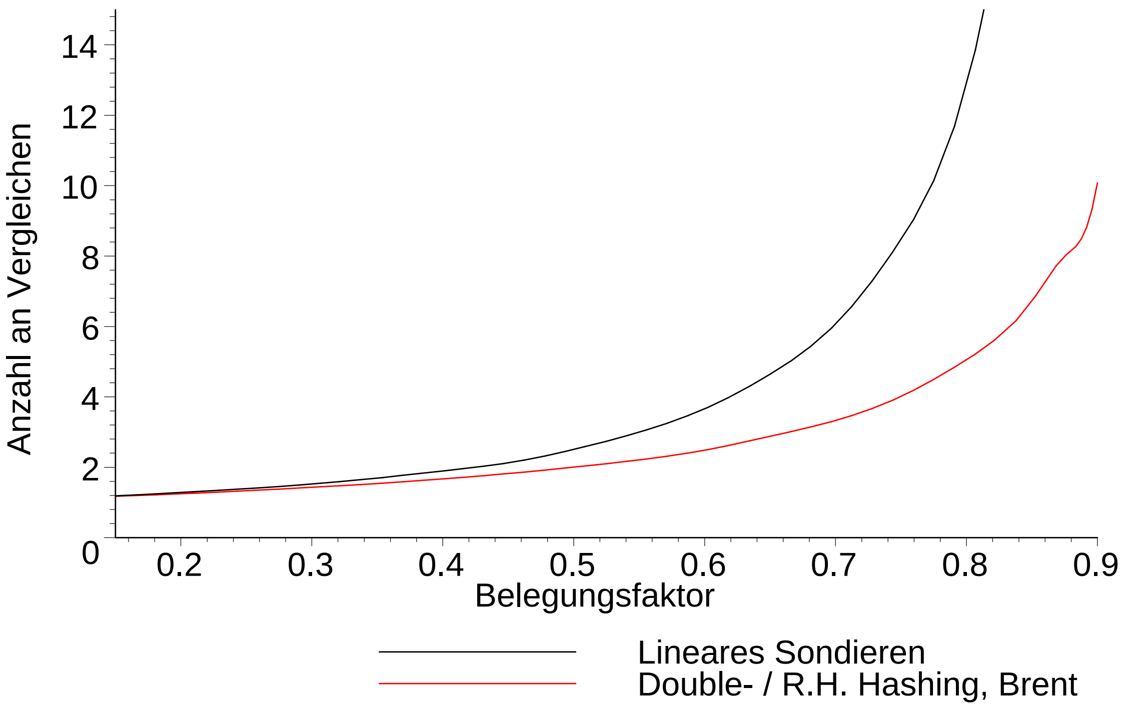


Abbildung 7.7: Stichprobenmittel: Vergleiche für erfolglose Suche

Durchschnittlich benötigte Anzahl an Vergleichen für erfolglose Suche bei Hashing mit offener Adressierung

Da bei dieser Untersuchung zwischen Double Hashing, Robin Hood Hashing und Brents Variation, von geringen zufälligen Schwankungen abgesehen, kein Unterschied festzustellen war, werden nur die Resultate von Double Hashing in Abbildung 7.7 und den Tabellen 7.9 und 7.10 angeführt. Auch weichen die Ergebnisse größtenteils nur geringfügig von den entsprechenden Erwartungswerten bzw. Varianzen ab, nur für fast volle Tabellen treten bei linearem Sondieren deutliche Abweichungen auf. Diese sind einerseits auf zufällige Schwankungen zurückzuführen, die sich in diesem Bereich besonders stark auswirken, und andererseits dadurch bedingt, dass auch das das zum Vergleich herangezogene, an sich exakte Ergebnis hier nur eine ungenaue Approximation der auftretenden Summen beinhaltet. Vergleichbare Resultate sind in [5, 8, 22] enthalten.

	Lineares Sondieren			Double - / R.H. Hashing, Brent		
α	$\mathbb{E}C_{10^5}^{[l]}$	Ergebnis	rel. Abw.	$\mathbb{E}C_{10^5}^{[l]}$	Ergebnis	rel. Abw.
0.250	1.389	1.392	0.23%	1.333	1.333	−0.02%
0.400	1.889	1.896	0.41%	1.667	1.67	0.22%
0.500	2.5	2.507	0.28%	2	2.005	0.25%
0.650	4.579	4.64	1.33%	2.856	2.879	0.80%
0.750	8.496	8.471	−0.29%	4	4.011	0.28%
0.850	22.68	22.73	0.23%	6.663	6.708	0.68%
0.900	50.31	51.68	2.72%	9.994	10.06	0.66%
0.950	197.9	194.3	−1.80%	19.98	19.88	−0.49%
0.990	3487	3319	−4.81%	98.94	99.08	0.14%
0.999		$35.26 \cdot 10^3$		971.8	962.1	−1.00%

Tabelle 7.9: Stichprobenmittel: Vergleiche für erfolglose Suche

	Lineares Sondieren			Double - / R.H. Hashing, Brent		
α	$\mathbb{E}C_{10^5}^{[l]}$	Ergebnis	rel. Abw.	$\mathbb{E}C_{10^5}^{[l]}$	Ergebnis	rel. Abw.
0.250	0.7067	0.7086	0.27%	0.4444	0.4479	0.78%
0.400	2.617	2.651	1.29%	1.111	1.129	1.60%
0.500	6.582	6.945	5.52%	2	2.02	1.01%
0.650	34.28	34.82	1.55%	5.302	5.386	1.59%
0.750	149	151.1	1.42%	12	12.27	2.27%
0.850	1276	1336	4.73%	37.73	38.34	1.61%
0.900	6750	7340	8.74%	89.88	93.61	4.15%
0.950	$109 \cdot 10^3$	$96 \cdot 10^3$	−12.26%	379.2	371.2	−2.11%
0.990		$20.28 \cdot 10^6$		9690	9648	−0.44%
0.999				$944 \cdot 10^3$	$892 \cdot 10^3$	−5.51%

Tabelle 7.10: Stichprobenvarianz: Vergleiche für erfolglose Suche

7.2.3 Verfahren zur Begrenzung der maximalen Suchlänge

Von besonderem Interesse bei allen verfahren ist auch die erwartete größte benötigte Anzahl an Vergleichen. Je nach Anwendung ist nämlich ein etwas höherer Gesamtaufwand gerechtfertigt, wenn dadurch der Aufwand für den schlechtest möglichen Fall verkleinert werden kann. Von den in dieser Arbeit behandelten Verfahren verfolgen Cuckoo Hashing (sowie auch andere perfekte Hashalgorithmen) und das 2-left Schema diese Strategie. Während bei Cuckoo Hashing die Suchlänge durch zwei begrenzt ist, treten bei anderen Verfahren deutlich größere Werte auf, wie Tabelle 7.11 belegt.[2] siehe auch [16, 26].

α	sep. Verk.	2-left	α	Lin. Sond.	Double H.	R. Hood H.	Brent
0.4	5	4	0.25	11	7	4	3
0.7	6	4	0.40	20	10	5	4
1.0	7	5	0.50	32	14	5	4
1.6	9	6	0.65	65	21	7	4
2.5	11	8	0.75	130	30	9	5
4.0	14	11	0.85	353	50	11	5
5.5	16	14	0.90	672	72	13	6
7.0	19	17	0.95	1949	136	17	6
8.5	21	20	0.99	16720	603	32	8
10.0	24	23	1.00	77000	3447	78	10

Tabelle 7.11: Maximale Suchlängen

2-left Schema

Wie Tabelle 7.11 zeigt sinkt die maximal auftretende Suchlänge nur unwesentlich im Vergleich zu gewöhnlichem Hashing mit Verkettung. Allerdings wurde dieses Ergebnis nicht unter Verwendung paralleler Berechnungen, wie sie sich für diesen Algorithmus anbieten, ermittelt, sondern durch sequentielle Behandlung der beiden Tabellen mit nur einem Prozessor. Stehen jedoch zwei Prozessoren zur Verfügung, so beträgt die maximale Suchlänge (je Prozessor) nur mehr die Hälfte des in der Tabelle angegebenen Wertes. In [6] finden sich vergleichbare Ergebnisse.
Ermittelt man nun Ausgleichsgeraden, so erhält man 2202ms $+\alpha$308ms bzw. 921ms $+\alpha$379ms für die Laufzeit eines Einfügevorgangs bzw. eines erfolgreichen Suchvorgangs. Die Steigung ist damit jeweils etwa doppelt so hoch wie bei Hashing mit Verkettung, und könnte durch parallele Berechnungen etwa die selbe Größenordnung annehmen. Dies wird auch durch die Resultate für die benötigten Vergleiche gestützt, diese betragen $0.814 + \alpha 0.972$ für erfolgreiche und $0.351 + \alpha 1.95$ für erfolglose Suche.
Das Verfahren ist also dann zu empfehlen, wenn man bereit ist, die doppelte Rechenleistung zur Verfügung zu stellen, wodurch sich bei ansonsten etwa gleicher Leistung der Aufwand des erwarteten Worst Cases etwa halbiert.

Cuckoo Hashing

Die Laufzeit des Verfahrens hängt stark davon ab, wieviele Versuche für den Aufbau der Tabelle benötigt werden. So betrug die Laufzeit von ca. 2500ms für geringe Auslastungen bis etwa 4500ms für Auslastungen in der Nähe von 0.5. Damit benötigt das Verfahren erwartungsgemäß

[2] Die angegebenen Werte entsprechen dabei dem Mittel der Maximalwerte von 25 Tabellen.

wesentlich mehr Zeit als alle anderen Verfahren bei gleicher Auslastung. Die Suche benötigte stets um 1250ms und liegt damit in der selben Größenordnung wie bei den anderen Algorithmen. Kein weiteres der hier behandelten Verfahren gewährleistet eine derart geringe maximal mögliche Suchlänge wie Cuckoo Hashing. Wie Tabelle 7.11 jedoch zeigt, weisen auch andere Verfahren geringe maximale Suchlängen speziell für Auslastungen bis 0.5 auf. Da diese Algorithmen ansonsten besseres Laufzeitverhalten besitzen, ist es in den meisten Fällen vorzuziehen, eines dieser Verfahren zu verwenden.
Weitere Untersuchungen mit ähnlichen Resultaten zu Cuckoo Hashing (allerdings mit anderen Hashfunktionen) sind in [37] enthalten.

7.2.4 Vergleich der Verfahren

Hashalgorithmen mit **offener Adressierung** und **Coalesced Hashing** sind vor allem für Anwendungen empfehlenswert, bei denen die Anzahl der benötigten Tabellenplätze gut vorhergesagt werden kann, oder zumindest keine starken Schwankungen auftreten. Weiters sind vor allem Anwendungen empfehlenswert, die Löschvorgänge nur selten erfordern. Generell empfiehlt es sich für diese Verfahren eine Auslastung der Tabellen von 70% bis 80% anzustreben (vergleiche [43]), da bis zu dieser Größe nur ein geringer Anstieg der Laufzeit zu bemerken ist, und noch Restkapazität für zusätzliche Einträge vorhanden ist. Bei einer Überschreitung dieses Bereichs, spätestens aber bei Auslastungen von 95% empfiehlt es sich eine neue, größere Hashtabelle anzulegen und die bisherigen Einträge zu übernehmen.
Für Anwendungen, für die obige Voraussetzungen nicht erfüllt sind empfiehlt sich die Verwendung einer **Hashtabelle mit Verkettung**.

7.3 Erstellung eines Wörterbuchs

Für praktische Anwendungen ist es im Allgemeinen eher unwahrscheinlich, dass die tatsächlichen Daten sich so zufällig wie die bis jetzt zu Testzwecken verwendeten Pseudozufallszahlen verhalten. Dies ist insbesondere bei einem Wörterbuch der Fall, da klarerweise nicht alle Wörter gleichwahrscheinlich auftreten, und ein Großteil der Buchstabenkombinationen keinen Sinn ergibt.
Ziel dieser Untersuchung ist die Erstellung eines Wörterbuchs aller vorkommender Wörter eines englisch sprachigen Artikels zum Thema Computer aus der Online-Enzyklopädie Wikipedia. Dieser weist insgesamt eine Länge von ca. 18000 Wörtern auf, darunter ca. 3400 von einander verschiedene. Für jedes Wort werden dabei höchstens die ersten 20 Stellen herangezogen, längere Wörter werden abgeschnitten. Weiters werden alle Sonderzeichen aus dem Text nicht berücksichtigt. Getestet werden hier zwei unterschiedliche Verfahren:

- Universelles Hashing mit separater Verkettung mit Hashfunktionen aus Satz 5.3
- Gewöhnliches Hashing mit separater Verkettung mit Divisionsmethode

Ein direkter Vergleich der erhaltenen Resultate ist schwer möglich, da das erste Verfahren Zweierpotenzen als Tabellengrößen erfordert, während für das andere die Verwendung von nicht zu nahe an diesen Potenzen liegenden Primzahlen empfehlenswert ist. Einzig möglich ist deshalb der Vergleich der aus den Experimentdaten gewonnenen Ausgleichsgeraden. Bei beiden Fällen sind hier nur minimale Abweichungen vom theoretischen Wert $1 + \frac{\alpha}{2}$ für die Anzahl der benötigten Vergleiche einer erfolgreichen Suche festzustellen. Dies zeigt einerseits, dass man auch bei „realen” Daten gute Ergebnisse mit herkömmlichen Verfahren erzielen kann, andererseits entsteht durch Verwendung von Universellem Hashing kein wesentlicher Mehraufwand bei der Laufzeit.

Anhang A

Bemerkung zu „Cuckoo Hashing: Further Analysis"

Wie bereits im Kapitel 6.3 erwähnt, ist der Beweis von Devroye und Morin aus [12] in der angegebenen Form nicht korrekt. Die Autoren verwenden den Ausdruck

$$\sum_{k=2}^{n-1}\sum_{l=1}^{k-1}\binom{m}{l}\binom{m}{k-l}\binom{n}{k+1}\left(\frac{l(k-l)}{m^2}\right)^{k+1}$$

um den Anteil der auftretenden Graphen abzuschätzen, der mindestens eine Komponente mit mehr als einem Kreis enthält. Sie folgern daraus, dass der gesuchte Anteil durch einen Ausdruck der Ordnung $O(1/n)$ beschränkt ist. Die oben auftretenden Summanden werden unter Verwendung der Stirlingformel weiter abgeschätzt, die anschließende Rechnung enthält jedoch einen Fehler, da ein Ausdruck fälschlicherweise im Nenner statt im Zähler angeschrieben wird. Der tatsächlich entstehende Ausdruck ist im Unterschied zum angegebenen nicht mehr konvergent. Das Problem lässt sich nicht durch eine bessere Abschätzung der oben angegebenen Doppelsumme beheben, da bereits einzelne Summanden gegen unendlich streben, so z.B. für $m = \frac{3}{2}n$, $k = \frac{n}{2}$ und $l = \frac{3}{10}n$:

$$\binom{m}{l}\binom{m}{k-l}\binom{n}{k+1}\left(\frac{l(k-l)}{m^2}\right)^{k+1} \to \infty \quad \text{für } n \to \infty$$

Anhang B

Symboltafel

Symbol	Bedeutung
$\mathbb{N}$	Menge der natürlichen Zahlen (inklusive Null)
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{C}$	Menge der komplexen Zahlen
$\mathbb{Z}_n$	Restklassenring modulo n
$\lvert M\rvert$	Mächtigkeit der Menge M
$\exists^*$	es gibt genau ein
$\mathbb{P}(A)$	Wahrscheinlichkeit, dass das Ereignis A auftritt
$\mathbb{P}(A\mid B)$	bedingte Wahrscheinlichkeit von A, gegeben B
$\mathbb{E}\eta$	Erwartungswert der Zufallsgröße η
$\mathbb{V}\eta$	Varianz der Zufallsgröße η
$\mathrm{ggT}(a,b)$	(positiver) größter gemeinsamer Teiler der Zahlen a und b
$a\mid b$	a teilt b, d.h. $\exists c$ mit $ac=b$
$a\nmid b$	a teilt b nicht
$x^{\overline{k}}$	$x(x+1)(x+2)\dots(x+k-1)$ (steigende Faktorielle)
$x^{\underline{k}}$	$x(x-1)(x-2)\dots(x-k+1)$ (fallende Faktorielle)
$\binom{n}{k_1,\dots,k_m}$	$\frac{n!}{k_1!\dots k_m!}$ wobei $\sum_{i=1}^m k_i = n$ (Multinomialkoeffizient)
$x \mod p$	ganzzahliger Rest bei der Division x durch p
$\delta_{i,j}$	$\begin{cases} 1 \text{ falls } i=j \\ 0 \text{ sonst} \end{cases}$ (Kronecker Delta)
$\lfloor x\rfloor$	größte ganze Zahl, die kleiner oder gleich x ist
$\lceil x\rceil$	kleinste ganze Zahl, die größer oder gleich x ist
$\{x\}$	$x-\lfloor x\rfloor$ (gebrochener Anteil von x)
$\mathrm{sgn}(x)$	Vorzeichen von x, wobei $\mathrm{sgn}(0)=0$
$f(n)\sim g(n)$	$\lim_{n\to\infty}\frac{f(n)}{g(n)}=1$ (asymptotisch äquivalent)
$f(n)=o(g(n))$	$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$
$f(n)=O(g(n))$	$\lvert f(n)\rvert\le c\lvert g(n)\rvert$ für $n\ge n_0$ mit $c>0$
$f(n)=\Theta(g(n))$	$c_1\lvert g(n)\rvert\le\lvert f(n)\rvert\le c_2\lvert g(n)\rvert$ für $n\ge n_0$ mit $c_1,c_2>0$
$f^{(n)}$	n-te Ableitung der Funktion f
D	Differentialoperator; ordnet jedem Polynom seine (formale) Ableitung zu
$[x^n]a(x)$	Ablesen des n-ten Koeffizienten der Potenzreihe $a(x)=\sum_{n\ge 0}a_nx^n$
$\oplus$	bitweise Exklusiv- Oder Verknüpfung
$\mathrm{bin}(x)$	Umwandlung der Dezimalzahl x in eine Binärzahl
$\blacksquare$	kennzeichnet das Ende eines Beweises

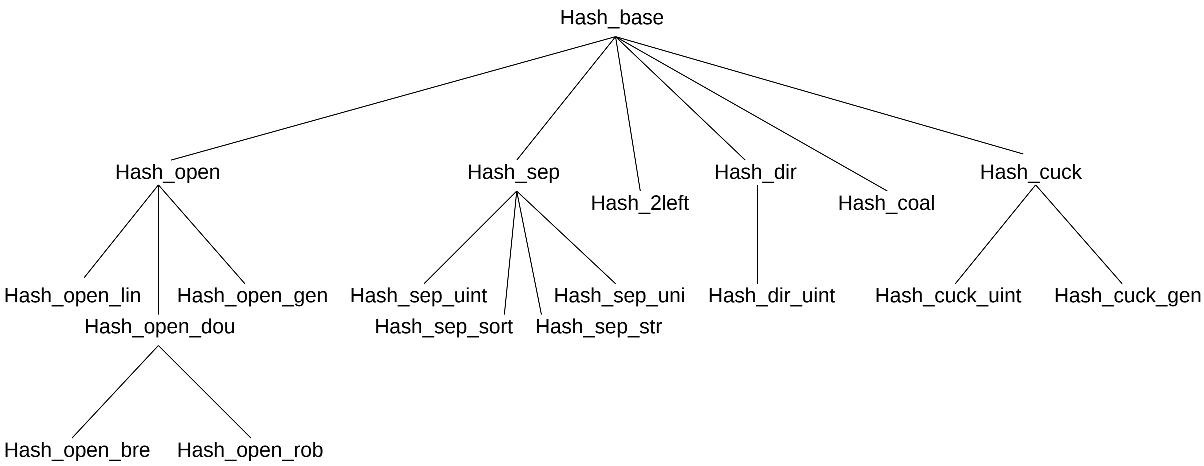
Anhang C

Software

Die der Arbeit beiliegende CD-ROM enthält neben den beiden ausführbaren MS-Windows Programmen `HashingTool.exe` und `Universal.exe` auch die Quellcodes der im Rahmen dieser Arbeit erstellten Bibliothek von Hashfunktionen. Weitere Details über die Verwendung der Programme enthält die Datei `readme.txt`. Der Aufbau der einzelnen Klassen der Software und wie diese verwendet werden können, wird hier kurz erläutert.

Hash_base

Diese abstrakte Klasse dient als Schnittstelle für die Anwendung der gesamten Bibliothek. Alle weiteren Klassen, die Hashverfahren umsetzen, sind von ihr abgeleitet.



Dem Anwender bleibt dadurch der reale Aufbau verborgen, entscheidend sind nur die von allen diesen Verfahren unterstützten Operationen:

- `search(S)` Suche nach dem Schlüssel `S`.
- `insert(S)` Einfügen des Schlüssel `S` in die Tabelle.
- `del(S)` Entfernen des Schlüssels `S`.
- `get_n()` Ermittelt die Anzahl der in der Tabelle enthaltenen Schlüssel.
- `out(std::ofstream*)` Bewirkt die Ausgabe des aktuellen Tabelleninhalts in eine Textdatei. Bei großen Tabellen wird nur der Anfang ausgegeben.

Weiters stehen die Operatoren `new` und `delete` zur Verfügung, welche bereits in der Basisklasse implementiert sind.
Diese Konzept bietet den Vorteil, eine Anwendung so programmieren zu können, dass eine Änderung des verwendeten Hashverfahrens sehr leicht möglich ist, es genügt nämlich, einen einzigen Befehl zu ändern. Insbesondere muss das Verfahren nicht fix festgelegt werden, es kann z.B. durch Benutzereingabe ausgewählt werden.

Name	Funktionalität	Datentyp
`Hash_open`	rein virtuelle Basisklasse für offene Adressierung	beliebig
`Hash_open_lin`	Lineares Sondieren, Hashfunktion `S%m`	`unsigned int`
`Hash_open_dou`	Double Hashing, Hashfunktionen `S%m`, `1+(S %(m-2))%m`	`unsigned int`
`Hash_open_bre`	Brents Variation von Double Hashing, Hashfunktionen wie bei `Hash_open_dou`	`unsigned int`
`Hash_open_rob`	Robin Hood Hashing, verwendet Hashfunktionen wie bei `Hash_open_dou`	`unsigned int`
`Hash_open_gen`	Double Hashing, beliebige Hashfunktionen über Funktionszeiger angebbar	beliebig
`Hash_sep`	rein virtuelle Basisklasse für Hashing mit separater Verkettung	beliebig
`Hash_sep_uint`	Hashing mit separater Verkettung, Hashfunktion `S%m`	`unsigned int`
`Hash_sep_sort`	wie `Hash_sep_uint`, jedoch sortiertes Einfügen	`unsigned int`
`Hash_sep_uni`	Universelles Hashing mit separater Verkettung, Hashfunktion nach Satz 5.3	`std::string`
`Hash_sep_str`	Hashing mit separater Verkettung, Zeichenkette wird in numerischen Wert umgerechnet, dann wird Divisionsmethode als Hashfunktion eingesetzt	`std::string`
`Hash_dir`	rein virtuelle Basisklasse für Hashing mit direkter Verkettung	beliebig
`Hash_dir_uint`	Hashing mit direkter Verkettung, Hashfunktion `S%m`	`unsigned int`
`Hash_2left`	2left Schema, Hashfunktionen nach Satz 5.1	`unsigned int`
`Hash_coal`	Coalesced Hashing, Hashfunktion `S%m`	`unsigned int`
`Hash_cuck`	rein virtuelle Basisklasse für Cuckoo Hashing	beliebig
`Hash_cuck_uint`	Cuckoo Hashing, Hashfunktionen nach Satz 5.1	`unsigned int`
`Hash_cuck_gen`	Cuckoo Hashing, beliebige Hashfunktionen über Funktionszeiger angebbar	beliebig

Index

c- universell, 57
d-Left Schema, 35

Abels Verallg. bin. Lehrsatz, 12
Airy Verteilung, 14
Asymptotische Entwicklungen, 10

Baum, 13
Belegungsfaktor, 3
beste Approximation, 17
Brents Algorithmus, 53

Chinesischer Restsatz, 16, 63

Divisionsmethode, 5

Erzeugende Funktion, 8
    exponentiell-, 8
    Wahrscheinlichkeits-, 13
Eulersche Konstante, 9

Fibonaccizahlen
    verallgemeinerte, 10, 37
FKS Verfahren, 65

Geburtstagsparadoxon, 3, 13
Graph, 12
    gerichteter, 12
    paarer, 13
    schlichter, 12
    Teilgraph, 12
    ungerichteter, 12
    zusammenhängender, 13

Häufung
    primäre, 41
    sekundäre, 49
Harmonische Zahlen, 9
Hashfunktion, 2
    FKS, 65
    häufig verwendete, 5
    in der Kryptographie, 3
Hashing, 2
    Coalesced, 28
    Cuckoo, 68, 85
    Double, 52
    mit direkter Verkettung, 22
    mit mehreren Tabellen, 35
    mit offener Adressierung, 4, 40, 60
    mit separater Verkettung, 22
    mit Verkettung, 4, 22, 60
    perfektes, 62
    polynomielles, 6
    Robin Hood, 55
    Uniform, 49
    universelles, 57

Information, 2

Kantenfolge, 12
Keller, 30
Kettenbruch
    endlicher, 16
    unendlicher, 17
KISS Generator, 15
Kollision, 3
Kollisionen
    Auflösung von, 4
Konvergenzradius, 8
Kreis, 12

markierte Wurzelbäume, 9
Modell für die Eingangsdaten, 6
Multiplikationsmethode, 5, 16

Näherungsbrüche, 17

perfekt, 62
    minimal, 62
Pseudozufallszahlen, 15
    Generatoren für, 15

Ramanujans Q-Funktion, 9
Reihenentwicklungen, 8

Satz
    von Rouché, 10
Schlüssel, 2

abgekürzte, 28
Sondieren
Binärbaum, 55
Lineares, 41
Quadratisches, 48
Zufälliges, 51
Sondierungsfolge, 40
Sortiertes Einfügen, 26
Stellenanaylse, 5
Stirlingsche Formel, 10

Three Distance Theorem, 19

universell, 57
Universum, 2

Witness Tree, 36
durchmusterter, 39
vollständiger, 38

Zusammenhangskomponenten, 13

Literaturverzeichnis

[1] Allouche, J.-P., and Shallit, J. *Automatic Sequences: Theory, Applications, Generalizations.* Cambridge University Press, 2003.

[2] Alon, N., Dietzfelbinger, M., Miltersen, P. B., Petrank, E., and Tardos, G. Is linear hashing good? *Comm. ACM* (1997), 465–474.

[3] Bernscherer, C., Hornik, K., and Schmidt, A. M. *Wahrscheinlichkeitstheorie.* Skriptum an der TU Wien, 1996.

[4] Betten, A., Fripertinger, H., Kerber, A., Wassermann, A., and Zimmermann, K. *Codierungstheorie, Konstruktion und Anwendung linearer Codes.* Springer, 1998.

[5] Brent, R. P. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM 16*, 2 (1973), 105–109.

[6] Broder, A. Z., and Mitzenmacher, M. Using multiple hash functions to improve ip lookups. *IEEE INFOCOM* (2001), 1454–1463.

[7] Carter, J. L., and Wegman, M. N. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2 (1979), 143–154.

[8] Celis, P., Larson, P., and Munro, J. I. Robin hood hashing. *Proceedings FOCS 26* (1985), 281–288.

[9] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[10] Czech, Z. J., Havas, G., and Majewski, B. S. Perfect hashing. *Theoretical Computer Science 182* (1997), 1–143.

[11] Devroye, L., and Morin, P. Cuckoo hashing: further analysis (preprint). http://jeff.cs.mcgill.ca/∼luc/devs.html, 9 2001.

[12] Devroye, L., and Morin, P. Cuckoo hashing: further analysis. *Information Processing Letters 86*, 4 (2003), 215–219.

[13] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing 24*, 4 (1994), 738–761.

[14] Drmota, M. *Diskrete Methoden.* Skriptum an der TU Wien, 2000.

[15] Drmota, M. *Zahlentheorie für TM.* Skriptum an der TU Wien, 2003.

[16] Flajolet, P., Grabner, P. J., Kirschenhofer, P., and Prodinger, H. On ramanujan's q-function. *Journal of Computional and Applied Mathematics 58*, 1 (1995), 103–116.

[17] Flajolet, P., and Louchard, G. Analytic variations on the airy distribution. *Algorithmica 31* (2001), 361–377.

[18] Flajolet, P., Poblete, P., and Viola, A. On the analysis of linear probing hashing. *Algorithmica 22*, 4 (1998), 490–515.

[19] Flajolet, P., and Sedgewick, R. *Analytic Combinatorics* (in Vorbereitung). http://algo.inria.fr/flajolet/Publications/books.html, 2005.

[20] Fredman, M. L., Kolmós, J., and Szemerédi, E. Storing a sparse table with o(1) worst case access time. *Journal of the ACM 31*, 3 (1984), 538–544.

[21] Gonnet, G., and Baeza-Yates, R. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[22] Gonnet, G., and Munro, J. I. The analysis of an improved hashing technique. *Proceedings STOC-SIGACT 9* (1977), 113–121.

[23] Hardy, G. H., and Wright, E. M. *An Introduction to the Theory of Numbers*. Oxford: Clarendon Press, 1979.

[24] Jähnich, K. *Funktionentheorie*. ?, 0.

[25] Knuth, D. *The Art of Computer Programming, Vol. 3 Sorting and Searching*. Adisson-Wesley, 1973.

[26] Larson, P.-A. Expected worst-case performance of hash files. *The Computer Journal 25*, 3 (1982), 347–352.

[27] Larson, P.-A. Analysis of uniform hashing. *Journal of the ACM 30*, 4 (1983), 805–819.

[28] Lever, C. Linux kernel hash tabel behavior: Analysis and improvements. Tech. Rep. 00-1, CITI, 2000.

[29] Lum, V. Y., Yuen, P. S. T., and Dodd, M. Key- to- adress transform techniques: A fundamental performance study on large existing formatted files. *Communications of the ACM 14*, 4 (1971), 228–239.

[30] Markowsky, G., Carter, J. L., and Wegman, M. N. Analysis of a universal class of hash functions. *Lecture Notes in Computer Sience 64*, 32 (1978), 345–354.

[31] Marsaglia, G. The marsaglia random number cdrom including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[32] Mehlhorn, K. *Datenstrukturen und effiziente Algorithmen, Band1 Sortieren und Suchen*. B.G. Teubner Stuttgart, 1988.

[33] Menezes, A., van Oorschot, P., and Vanstone, S. *Handbook of Applied Cryptographie*. CRC Press, 1996.

[34] Mlitz, R. *Algebraische Methoden*. Skriptum an der TU Wien, 2003.

[35] Ottmann, T., and Widmayer, P. *Algorithmen und Datenstrukturen*. BI-Wissenschaftsverlag, 1993.

[36] PAGH, R. *Hashing, randomness and dictionaries.* PhD thesis, University of Aarhus, 2002.

[37] PAGH, R., AND RODLER, F. F. Cuckoo hashing. Tech. Rep. RS-01-32, Basic Research in Computer Science, 2001.

[38] RAMAKRISHNA, M. V. Hashing in practice, analysis of hashing and universal hashing. *ACM* (1988), 191–199.

[39] RUPPERT, W. M. *Ausgewählte Kapitel aus der Kryptographie.* Skriptum an der Unversität Erlangen, 2001.

[40] SEDGEWICK, R. *Algorithmen in C++.* Addison-Wesley, 1992.

[41] SEDGEWICK, R., AND FLAJOLET, P. *Analysis of Algorithms.* Addison-Wesley, 1996.

[42] SÓS, V. T. On the theory of diophantine approximations. *Acta Math. Sci. Hung. 8* (1957), 461–471.

[43] STROUSTRUP, B. *Die C++ Programmiersprache.* Adisson-Wesley, 2000.

[44] VITTER, J. S. Analysis of the search performance of coalesced hashing. *Journal of the ACM 30*, 2 (1983), 231–258.

[45] VITTER, J. S., AND FLAJOLET, P. Average-case analysis of algorithms and data structures. In *Handbook of Theoretical Computer Sience*, J. van Leeuwen, Ed. North-Holland, 1990.

[46] VÖCKING, B. How asymmetry helps load balancing. *Proc. of the* 40^{th} *IEEE Symp. on Foundations of Computer Science* (1999), 131–141.