DHBW Ravensburg

Programmierung II – Algorithmen & Datenstrukturen

Prof. Dr. Andreas Schilling

www.ravensburg.dhbw.de

3. Hashing

Agenda

Grundlagen	Hashfunktion	Kollisionsauflösung
Was sind Hash-Tabellen und was ist ihr Vorteil.	Aufbau und Funktion der Hashfunktion.	Strategien zum Auflösen von Kollisionen.

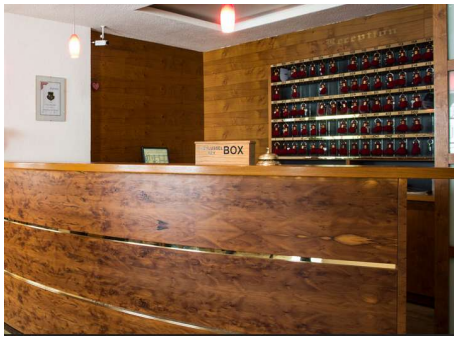
Grundlagen

Motivation

Ausgangsbeispiel:
Sie sind Gast in einem Hotel und fragen bei der Rezeption nach, ob eine Nachricht für Sie abgegeben wurde.

- **Unsortierter Stapel** von Nachrichten, der via linearer Suche durchgegangen wird (Laufzeit O(n)).
- **Sortierter Stapel** von Nachrichten, basierend auf dem Nachnamen, der binär durchsucht wird (Laufzeit O(log n)).
- Ein **Postkorb für jeden Buchstaben** des Alphabets, sodass der erste Buchstabe des Nachnamens den relevanten Postkorb identifiziert und so die Suche stark verkürzt.

→ Hash-Tabelle



Die Einordnung von Nachrichten für Gäste im Hotel in getrennte Postkörbe ist ein Beispiel für eine Hash-Tabelle

Um den Nutzen und die Funktionsweise einer Hashtabelle zu veranschaulichen, nehmen wir folgendes Ausgangsbeispiel: Sie sind Gast in einem Hotel und Sie fragen an der Rezeption, ob eine Nachricht für Sie abgegeben wurde. Mit unserem bisherigen Wissen zu Suchalgorithmen gibt es nun folgende Suchmöglichkeiten:

- Der Mitarbeiter an der Rezeption hat einen unsortierten Nachrichtenstapel vor sich den er mithilfe einer linearen Suche durchsucht, ob eine Nachricht darin für Sie adressiert ist. Wie wir im letzten Kapitel gelernt haben resultiert für dieses Suchvorgehen im Avg.-Case eine Laufzeit von O(n).
- Der Mitarbeiter an der Rezeption hat einen nach dem Nachnahmen sortierten Nachrichtenstapel vor sich und durchsucht diesen mithilfe einer binären Suche nach einer Nachricht für Sie. Dies resultiert in einer durchschnittlichen Laufzeit von O(log n).
- Eine dritte Möglichkeit ist, dass es für jeden Buchstaben des Alphabets ein eigenen Postkorb gibt und die Nachrichten basierend auf dem Anfangsbuchstaben des Nachnamens des Empfängers in einen dieser Postkörbe einsortiert wird. Bei einem derartigen Vorgehen muss der Mitarbeiter nun nur noch in dem Postkorb mit dem entsprechenden Anfangsbuchstaben nachschauen, ob es eine Nachricht für Sie gibt. Die Nutzung einer derartigen Datenstruktur wird in der Informatik als Hash-Tabelle bezeichnet.

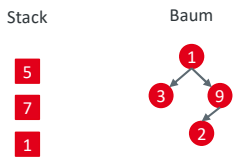
Grundlagen

Aufbau einer Hash-Tabelle

Hash-Tabellen werden zum effizienten Ablegen und Wiederfinden von **Schlüssel-Wert Paaren** eingesetzt**:**

[(Schlüssel, Wert), (Schlüssel, Wert)]

- Der **Schlüssel** ist ein Merkmal, das den Wert eindeutig zuordnen kann. Z.B.: Nachname, Kundennummer, Ausweis-Nr., Postleitzahl, Telefonnummer, E-Mail Adresse, etc.
- Der **Wert** entspricht dem konkreten Objekt, das abgelegt wird, dies kann sowohl ein Datensatz als auch ein reales Objekt sein.
- Die Werte werden in einem **Array** gespeichert, wobei der Zugriff auf die einzelnen Felder über den Schlüssel erfolgt.



Die bisherigen Datenstrukturen haben nur Werte gespeichert

Bei einer Hash-Tabelle werden **Schlüssel-Wert Paare** gespeichert.

Ein wichtiger Unterschied im Vergleich zu den bisher behandelten Datenstrukturen ist, dass in einer Hash-Tabelle nicht einzelne Werte abgelegt werden, sondern Schlüssel-Wert Paare. Der Schlüssel beschreibt ein Merkmal, das zum eindeutigen Wiederfinden des Wertes geeignet ist. Typische Beispiele für Schlüssel sind die Nutzung des Nachnamens, einer Kundennummer, einer Telefonnummer oder auch einer E-Mail Adresse.

Der Wert beim Schlüssel-Wert Paar steht für das abzulegende Objekt, also dem Datensatz der gespeichert und bei einer Abfrage wieder zurückgegeben werden soll. Die Werte werden bei einer Hash-Tabelle in ein Array geschrieben, wobei der Zugriff auf die einzelnen Felder nicht via Index erfolgt sondern über den Schlüssel.

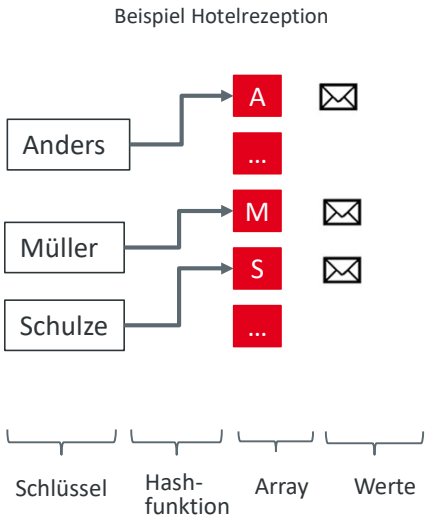
Grundlagen

nimmt den schlüssel und ordnet diesen einen wert zu

Aufbau einer Hash-Tabelle

Die Ablage der Werte in das Array der **Hash-Tabelle**, erfolgt mit Hilfe der Hashfunktion:

- Die **Hashfunktion** überführt den Schlüssel in den Array-Index an dem der Wert gespeichert wird.
- Die **Buckets** bezeichnen die einzelnen Speicherplätze des verwendeten Arrays.
- Die **Werte** der Schlüssel-Wert Paare werden in den einzelnen Buckets des Arrays gespeichert.



Im Folgenden werfen wir einen Blick auf die Funktionsweise der Hash-Tabelle. Im ersten Schritt wird auf Grundlage des Schlüsselwerts ein Speicherplatz im Array für den Wert-Teil identifiziert. Für diese Transformation des Schlüssels in einen numerischen Index ist die Hashfunktion verantwortlich, diese nimmt den Schlüsselwert entgegen und gibt den zu verwendenden Speicherplatzindex im Array zurück. Im Falle einer Hash-Tabelle werden die einzelnen Felder des Arrays, die zur Speicherung der Werte verwendet werden, Buckets genannt.

Agenda

Grundlagen

Was sind Hash-Tabellen und was ist ihr Vorteil.

Hashfunktion

Aufbau und Funktion der Hashfunktion.

Kollisionsauflösung

Strategien zum Auflösen von Kollisionen.

Hashfunktion

Eigenschaften

Eine **Hashfunktion** muss folgende Bedingungen zur Umsetzung einer Hash-Tabelle erfüllen:

- **Effizienz**: Nur wenn die Hashfunktion schnell berechnet wird bietet sie einen Vorteil ggü. andern Suchverfahren.
- **Gleichmäßige Verteilung**: Die Werte, sollten im Array gleichmäßig verteilt sein, um Clusterung zu vermeiden.
- **Deterministische Auflösung**: Bei mehrmaligem Aufruf wird für den gleichen Schlüssel stets der gleiche Wert zurückgegeben.
- **Uni-Direktional**: Der Hashwert kann nicht verwendet werden um den Schlüssel zu rekonstruiert.

M → Müller, Maier, Miller ?

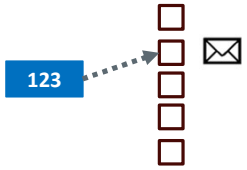
Die Hashfunktion ist von entscheidender Bedeutung für die Funktionsweise einer Hashtabelle. Folgende Bedingungen müssen von einer Hashfunktion erfüllt werden damit sie im Rahmen einer Hashtabelle verwendet werden kann:

- **Effizienz**: Entscheidend dafür dass eine Hashtabelle Laufzeitvorteile gegenüber anderen Datenstrukturen besitzt, ist, dass die Hashfunktion schnell den Schlüssel in ein Arrayindex zur Speicherung des Werts überführt.
- **Gleichmäßige Verteilung**: Die zugeordneten Indexfelder des Array sollten gleichmäßig verteilt sein, andernfalls steigt die Wahrscheinlichkeit dass es zu Kollisionen (der Zuweisung des gleichen Arrayindex für unterschiedliche Schlüssel kommt)
- **Deterministische Auflösung**: Die Hashfunktion muss bei wiederholtem Aufruf mit dem gleichen Schlüssel stets den gleichen Indexplatz zurückgeben, andernfalls kann der abgelegte Wert nicht wiedergefunden werden.
- **Uni-Direktiona**l: Die Hashfunktion funktioniert nur in eine Richtung: vom Schlüssel zum Indexplatz. Es darf nicht möglich sein, basierend auf dem Indexplatz den Schlüsselwert abzuleiten.

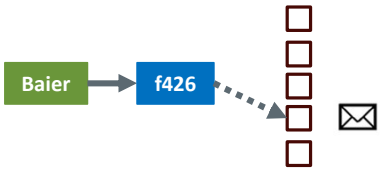
Hashfunktion

Klassen von Hashfunktionen

- Hashfunktionen für **numerische Schlüssel**
  - Unterstützen nur Schlüssel mit numerischen Wertebereichen.
  - Direkte Konvertierung von Schlüssel in den Arrayindex
  - Einsatz bei numerischen Indizes (bspw. Datenbanken)



- **Generische Hashfunktionen**
  - Keine Anforderung an den Wertebereich des Schlüssel (bspw. alphanumerisch, hexadezimal, binär, etc.).
  - Zuerst Transformation in eine Zahl notwendig.
  - Einsatz bei Verschlüsselung, Prüfsummen oder digitalem Fingerabdruck.



9

Es werden folgende zwei grundlegende Kategorien von Hashfunktionen basierend auf dem Typ des verwendeten Schlüssels unterschieden:

- Hashfunktionen für numerische Schlüssel. Diese Form der Hashfunktion nutzt mathematische Formeln zur Überführung des numerischen Schlüsselwerts in einen konkreten Arrayindex. Diese Klasse von Hashfunktionen kommen unter Anderem in Datenbanken zum Einsatz, bei denen vorwiegend numerische Indexwerte für Datensätze verwendet werden.

- Bei generische Hashfunktionen werden sämtliche Arten von Schlüsselwerten (bspw. binär, alphanumerisch, hexadezimal) unterstützt. Hierbei ist es erforderlich, dass zunächst eine Überführung des Schlüsselwertes in eine numerische Repräsentation erfolgt, welche anschließend in einen numerischen Index überführt wird. Generische Hashfunktionen werden unter Anderem bei der Verschlüsselung, bei der Sicherstellung von Nachrichtenübertragungen und bei digitalen Signaturen eingesetzt.

Hashfunktion

Numerische Schlüssel

Möglichkeiten zur Umsetzung der Hashfunktion:

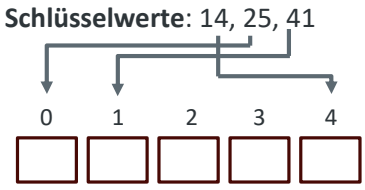
Modulo-Operation:

Bei **Gleichverteilung der Schlüsselwerte** ermöglicht die Divisionsrest-Methode eine ausgeglichene Auslastung der Hash-Tabelle.

Durch die **rotierende Slotzuweisung** ermöglicht die Modulo-Operation, dass der absolute **Schlüsselwert die maximale Slotanzahl übersteigen** kann.

Die Nutzung einer **Primzahl als Divisor**, reduziert die Anzahl an Kollisionen. *(Knuth 1998)*

Hashfunktion:
h(k) = k mod 5



Eine Möglichkeit zur Überführung von numerischen Schlüsseln in einen konkreten Array-Index ist die Modulo- (auch Divisionsrest-) Methode. Hierbei wird der numerische Schlüssel durch eine Zahl geteilt und der Rest der Ganzzahldivision wird als Index verwendet. So wird bei einem numerischen Index von 32 und der Hashfunktion h(k) = k mod 5 der Index 2 zugewiesen.

Ein Nachteil bei der Verwendung der Modulo Operation ist, dass eine Gleichverteilung über alle Buckets hinweg nur erreicht werden kann, wenn auch die numerischen Schlüssel gleichverteilt über ihren Wertebereich sind.

Ein wichtiger Vorteil der Modulo Operation ist, dass die Wertebereiche der Bucketindizes und der numerischen Schlüssel voneinander abweichen können.

Knuth (1998) hat in seiner Forschung herausgefunden das Primzahlen als Divisor im Rahmen der Modulo Operation eingesetzt werden sollten um eine gleichmäßige Verteilung über die verfügbaren Buckets zu erreichen.

Hashfunktion

Numerische Schlüssel

Multiplikative-Methode

Verallgemeinerung der Modulo-Methode, bei der das Produkt des Schlüsselwerts mit einer Konstanten (A) gebildet wird und der ganzzahlige Teil abgeschnitten wird. Das Ergebnis hieraus wird mit der Kapazität der Hashtabelle (m) multipliziert.

$$h(k) = \lfloor m * (k * A - \lfloor k * A \rfloor) \rfloor$$

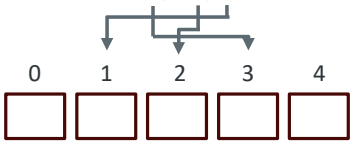
Floor operator rundet ab auf nächste Ganzzahl

Knuth (1997) zeigt, dass die Zahl des goldenen Schnitts (0,618) eine geeignete Konstante ist um eine hohe Streuung der Indexslots zu erreichen.

Hashfunktion:

$$h(k) = \lfloor 5 * (k * 0.618 - \lfloor k * 0.618 \rfloor) \rfloor$$

Schlüsselwerte: 14, 25, 41



n(14) = [5 * (14 * 0,618- [14*0,618])]
= [5 *(8,652 - 8)]
= [5 * 0,652]
= [3,26]
=3

11

Eine weitere Möglichkeit zur Umwandlung der numerischen Schlüsseln in eine Indexposition ist die Multiplikative Methode. Diese Methode ist eine Verallgemeinerung der Modulo-Methode im Rahmen derer mit einer Konstanten (A) gearbeitet wird. Konkret wird hier der ganzzahlige Teil der Multiplikation mit der Konstanten abgeschnitten. Die Gaußklammer (⌊ und ⌋) steht für die Abrundungsfunktion. Der nicht-ganzzahlige Restwert wird im Anschluss mit einer weiteren Konstante multipliziert.

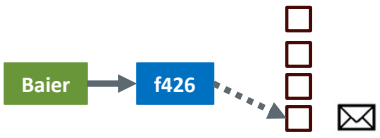
Knuth kommt in seinen Untersuchungen zum Schluss, dass sich insbesondere die Zahl des goldenen Schnitts (0,618) als geeignete Konstante für A anbietet um eine hohe Streuung über alle Buckets zu erzielen.

Vergegenwärtigen wir uns abschließend die Funktionsweise der multiplikativen-Methode mit der oben abgebildeten Hashfunktion, dem Schlüsselwert k von 14 und der Konstante A mit 0,618. Zunächst wird hier das Produkt von k und A gebildet das 8,652 ergibt. Im Anschluss daran wird der ganzzahlige Wert entfernt und der Rest mit 5 multipliziert (5*0,652) das 3,26 ergibt. Abschließend wird diese Zahl abgerundet um den endgültigen Arrayindex zu erhalten.

Hashfunktion

Generische Hashfunktion

- Bei generischen Hashfunktionen wird zunächst ein Hash-Code gebildet der anschließend dazu verwendet wird ein Arrayindex auszuwählen



- **MD5** ist ein populärer Algorithmus der ein 128 Stellen langen Hash-Code liefert. MD5 kann sehr schnell berechnet werden gilt allerdings als unsicher, da nachgewiesen wurde dass zwei Dateien den gleichen Hash-Code besitzen können

Einsatz: manuelle und automatische Integritätsprüfung

- **SHA-1** und **SHA-256** sind populäre Algorithmen, welche einen 160 bzw. 256 Stellen langen Hash-Code produzieren. SHA-256 gilt als sicher, jedoch behindert der lange Code einen schnellen manuelle Überprüfung

Einsatz: Passwortspeicher, Signatur automatische Integritätsprüfung.

Widmen wir uns abschließend den generischen Hashfunktionen, bei denen zunächst der Schlüsselwert in eine numerische Repräsentation überführt wird, bevor diese mithilfe einer Hashfunktion in ein Arrayindex überführt wird.

Ein etablierter Algorithmus um jegliche Form von Schlüssel (binär, textuell, numerisch) in einen Hash-Code zu überführen ist MD5. Dieser Algorithmus liefert einen 128-Stellen langen Hash-Code auf Grundlage dessen ein Bucket zugewiesen werden kann. Ein großer Vorteil von MD5 ist seine hohe Performance und seine leichte Verständlichkeit, jedoch wurde nachgewiesen, dass zwei Dateien den gleichen Hashcode besitzen können. Heute wird MD5 hauptsächlich zur manuellen Integritätsprüfung eingesetzt.

Zwei weitere sehr etablierte Algorithmen sind SHA-1 und SHA-256 welche einen 160 bzw. 256 Stellen langen Hash-Code erstellen. Während für SHA-1 nachgewiesen wurde dass es eine Überlagerung von Hash-Codes geben kann, ist dies bei SHA-256 bisher nicht der Fall. Daher gibt SHA-256 als sehr sicher und wird heute oft für sicherheitskritische Anwendungsfälle wie Passwörter und digitale Signaturen eingesetzt. Ein Nachteil der SHA-Algorithmen ist, dass sie eine längere Berechnungsdauer haben als MD5.

Agenda

Grundlagen

Was sind Hash-Tabellen und was ist ihr Vorteil.

Hashfunktion

Aufbau und Funktion der Hashfunktion.

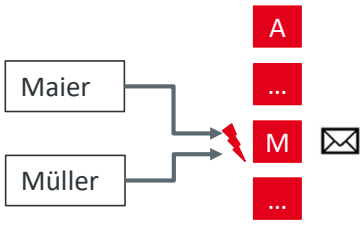
Kollisionsauflösung

Strategien zum Auflösen von Kollisionen.

Kollisionsauflösung

Kollisionen

- Unter einer **Kollision** versteht man das Ereignis, dass eine Hashfunktion für **unterschiedliche Schlüssel** den **gleichen Index** zurückgibt.
- Kollisionen können mithilfe einer großen Arraygröße und einer **gleichverteilten Hashfunktion** reduziert werden
- Es gibt unterschiedliche Strategien der Kollisions-behandlung:
  - **Geschlossenes Hashing** mit offener Adressierung
  - **Offenes Hashing** mit geschlossener Adressierung



Eine grundlegende Herausforderung von Hashtabellen sind Kollisionen, hierunter versteht man das Ereignis, dass eine Hashfunktion für **unterschiedlichen Input** den **gleichen Arrayindex** zurück gibt. Der Eintritt von Kollisionen bei Hashtabellen hängt maßgeblich von der verwendeten Hashfunktion und der verwendeten Arraygröße ab. Wird ein kleiner Array zur Speicherung der Objekte verwendet, sind auch bei einer gleichverteilten Hashfunktion Kollisionen unvermeidlich, da zu wenig Speicherplatz zur Verfügung steht um alle Objekte abzulegen. Im Gegenzug kann es auch bei großen Speicherarrays zu Kollisionen kommen, wenn die Hashfunktion die Werte nicht gleichverteilt und es zu einer Clusterung kommt.

Für die Auflösung von Kollisionen gibt es zwei Herangehensweisen, mit denen wir uns im Folgenden im Detail befassen: das geschlossene und das offene Hashing,

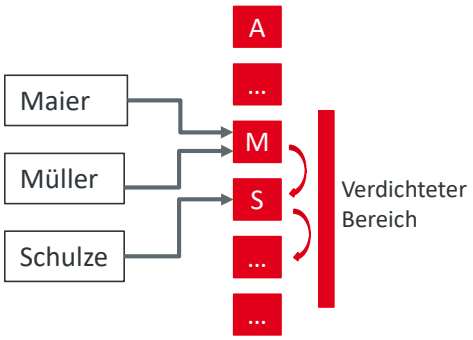
Kollisionsauflösung

Geschlossenes Hashing

Bei Verwendung des **geschlossenen Hashing mit offener Adressierung** wird beim Vorliegen einer Kollision nach einem alternativen Speicherplatz gesucht

- **Lineares Sondieren**: Sequenzielle Suche in gleicher Schrittgröße nach dem nächsten freien Speicherplatz.

  Herausforderung: Clustering von Werten, da sowohl die Kollisionen als auch die nächste Klassifikation der Hash-funktion um freien Speicher konkurrieren. In Folge dessen ist die Zugriffszeit umso länger.



Werden Kollisionen mit der Strategie des geschlossenes Hashing und offener Adressierung behandelt, wird beim Eintreten einer Kollision für den neu hinzukommenden Wert ein alternativer Speicherplatz im Array gesucht. Zur Suche eines alternativen Speicherplatzes gibt es drei grundlege Herangehensweisen:

Die erste Herangehensweise ist das lineare Sondieren. Hierbei wird in gleichgroßen Schritten nach dem nächsten freien Speicherplatz im Array gesucht. In der abgebildeten Kollision zwischen Maier und Müller, würde die Suche nach einem alternativer Platz zur Ablage des Wertes für Müller gestartet. Der nächste Speicherplatz im Array, ist jedoch durch den Schlüssel Schulze belegt, sodass der Wert für Maier im folgenden Speicherplatz abgelegt werden muss.

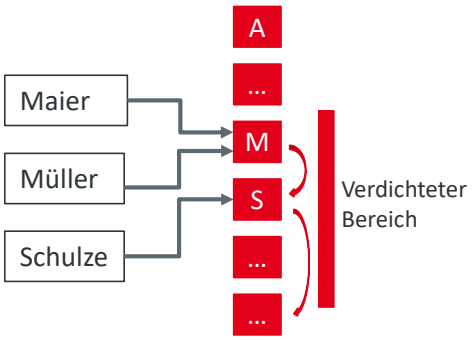
Die Herausforderung des linearen Sondieren ist, dass es hierdurch zu einem Clustering (Verdichtung) von Werten in einzelnen Abschnitten des Arrays kommt wodurch die Wahrscheinlichkeit für das Eintreten zukünftiger Kollisionen erhöht wird (selbstverstärkender Effekt). Infolgedessen steigt nicht nur die Zeit für die Ablage zukünftiger Werte sondern auch die notwendige Zeit für das Wiederfinden der passenden Werten zu einem Schlüssel.

Kollisionsauflösung

Geschlossenes Hashing

Quadratisches Sondieren: Auf der Suche nach einem freien Speicherplatz wird die Schrittgröße mit jedem Schritt verdoppelt.

Herausforderung: Clustering von Werten, jedoch deutlich besser verstreut als bei linearer Sondierung. Dennoch konkurrieren auch hier Kollisionen und Schlüsselwerte um Speicherplatz



sprungweite verdoppelt sich immer, erst dann 2 und danach 4

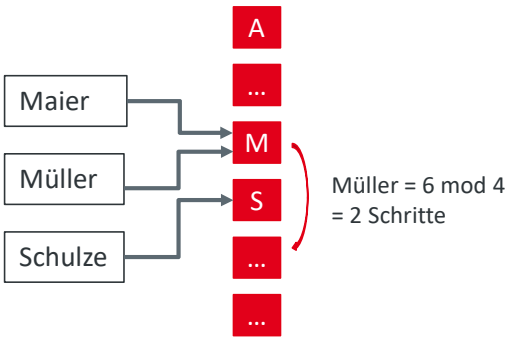
Beim quadratischen Sondieren, wird nicht eine statische Schrittanzahl bei der Suche nach dem nächsten freien Speicherplatz angewendet, sondern die Schrittgröße verdoppelt sich mit jedem Schritt. Durch die schnell wachsende Sprungweite verteilen sich die Werte gleichmäßiger als bei der linearen Sondierung. Allerdings entsteht auch hier (solange eine kleine Sprungweite besteht) die Gefahr eines verdichteten Bereichs im Array der zu längeren Speicher- und Zugriffszeiten der abzulegenden Werte führt.

Kollisionsauflösung

Geschlossenes Hashing

Doppeltes Hashing: Im Falle einer Kollision kommt eine weitere Hashfunktion zum Einsatz, deren Wert man mit der ursprünglichen Funktion kombiniert

Herausforderung: Die verwendeten Hash-funktionen müssen unabhängig voneinander sein, sodass eine Doppelkollision ausgeschlossen ist.



Beim doppelten Hashing wird bei einer Kollision das Ergebnis der zweiten Hash-funktion (hier Anzahl der Buchstaben modulo 4) auf die erste addiert.

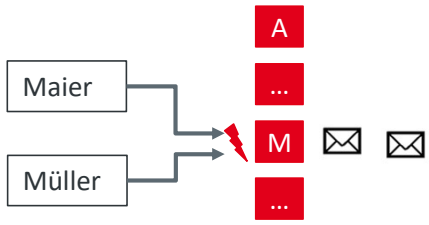
Die letzte Möglichkeit zum Umgang mit Kollisionen in einem Hasharray ist das doppelte Hashing. Bei dieser Strategie wird im Falle einer Kollision eine weitere Hashfunktion angewendet um für den neuen Schlüssel einen alternativen Speicherplatz zu finden. Im Vergleich zum linearen und quadratischen Sondieren entsteht durch den Einsatz einer weiteren Hashfunktion der Vorteil, dass verdichtete Bereiche in einem Array so gut wie möglich vermieden werden. Die Herausforderung beim Einsatz des doppelten Hashings ist jedoch, dass die hierfür verwendeten Hashfunktionen unabhängig voneinander sein müssen. Ansonsten entstehen wieder verdichtete Bereiche im Hasharray.

Kollisionsauflösung

Offenes Hashing

Bei Verwendung des **offenen Hashing mit geschlossener Adressierung** referenziert der verwendete Array keine konkreten Werte sondern eine verkettete Liste.

- Beim Eintreten einer **Kollision** wird kein alternativer Speicherplatz gesucht, sondern das neue Schlüssel-Wert Paar wird **an das bestehende** Schlüssel-Wert Paar **gehängt**.
- Die Verkettung von Schlüssel-Wert Paaren hat den Nachteil, dass die **Zugriffszeiten** rasant ansteigen können, im Extremfall in der Ordnung O(n).



Bei der Verwendung des offenen Hashing wird im Falle von Kollisionen das neue Schlüssel-Wert Paar an das Bestehende angehängt

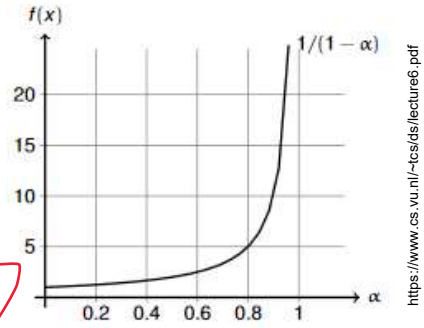
Zum Abschluss befassen wir uns mit der Möglichkeit des offenen Hashings mit geschlossener Adressierung. Hierunter versteht man die Herangehensweise, dass im Hasharray nicht konkrete Werte abgelegt werden, sondern stattdessen eine verkettete Liste mit Werten. Im Falle einer Kollision wird nun kein alternativer Speicherplatz im Hasharray für den konkreten Wert gesucht, sondern das Schlüssel-Wert Paar wird an die verkettete Liste angefügt.

Obwohl diese Herangehensweise den geringsten Aufwand für die Suche nach einem alternativen Speicherplatz hat, sollte sie mit Bedacht gewählt werden, da durch die verkettete Liste im Extremfall lineare Zugriffs- und Speicherzeiten für neue Elemente entstehen können.

Kollisionsauflösung

Laufzeitanalyse

- **Worst-Case**: Die Hashfunktion weist **allen Schlüsseln** den **gleichen Array-Slot** zu. Zugriffszeit dadurch analog zur verketteten List O(n)
- **Best-Case**: Die Hashfunktion **streut perfekt**, sodass es zu **keiner Kollision** kommt. Via Indexzugriff wird das gewünschte Element sofort gefunden -> O(1)
- **Avg.-Case**: Hierbei ist der **Auslastungsfaktor** α = n/m entscheidend, der das Verhältnis zwischen den zu speichernden Schlüsseln (n) und der Slot-Kapazität des Arrays beschreibt (m). Hierbei gilt für **offenes Hashing** $\mathcal{O}(1+\frac{n}{m})$ und **für geschlossenes Hashing** $\mathcal{O}(1+\frac{1}{1-n/m})$



https://www.cs.vu.nl/~tcs/ds/lecture6.pdf

Solange α unter 0,85 liegt können in der Praxis Zugriffszeiten der Ordnung O(1) erreicht werden

Blicken wir zum Abschluss auf das Laufzeitverhalten der Hash-Tabelle:

Beginnen wir mit dem Wort-Case. Hierbei gibt die Hashfunktion stets den gleichen Arrayslot für alle n-Schlüssel zurück. Infolge dessen entsteht beim offenen Hashing eine verkette Liste mit n-Elementen und im Falle des geschlossenen Hashing eine n-Schritt lange Suche nach dem gewünschten Element. In beiden Fällen entsteht so eine Laufzeit von O(n)

Im Best-Case verteilt die Hashfunktion die Schlüsselwerte gleichverteilt im Array, sodass ein konstanter Aufwand für den Zugriff und die Ablage der Schlüsselwerte entsteht -> O(1)

Im durchschnittlichen Fall hängen bei einer gleichverteilten Hashfunktion die Zugriffs- und Speicherzeiten von der aktuellen Auslastung des verwendeten Arrays ab. Hierzu muss der Auslastungsfaktor α bestimmt werden, der mit der Anzahl der abgelegten Werte (n) und der Kapazität des verwendeten Arrays (m) berechnet wird. Im Falle des offenen Hashings entstehen so mittlere Zugriffszeiten von O(1+n/m) und beim geschlossenen Hashing Zugriffszeiten von O(1+1/(1-n/m)). Hierbei lässt sich zeigen, dass solange α unter 0,85 liegt, im Durchschnitt konstante Zugriffszeiten erzielt werden können.

Aufgabe 7

k=Schlüssel
i= Anzahl vorliegender Kollisionen

a) Wenden Sie die unten genannten Sondierungsstrategien auf die nebenstehende Zahlenfolge und Hashtabelle an. Verwenden Sie hierzu die Hashfunktion h(k) = 3k mod 8.
Notieren Sie Versuche zur Identifikation eines freien Speicherplatzes bei Eintreten einer Kollision

- Lineare Sondierung (Schrittgröße 1)
- Quadratische Sondierung (verdoppelte Schrittgröße)
- Doppelte Hashfunktion mit
  $h(k,i) = (h(k) + i * h_1(k)) \bmod 8$
  $h_1(k) = 7k \bmod 3$

Zahlenfolge: 16, 5, 65, 75, 36, 8, 14

0	1	2	3	4	5	6	7
16	75	8	65	36	14		5
8		14					

quadr. 16 75 65 36 5

8 läuft unendlich durch, findet keinen slot in dem Array
bis zur 14 kommen wir gar nicht

Aufgabe 7

4 eigenschaften
deterministisch, effizienz, uni direktional, gleichmäßige verteilung
buchstaben sind nicht gleichmäßig verteilt

b) Handelt es sich bei der Hashfunktion aus dem Beispiel der Hotel-rezeption, bei der Nachrichten basierend auf dem ersten Buchstaben des Nachnamens ablegt werden um eine gute Hashfunktion?

Gehen Sie davon aus, dass der Anfangsbuchstabe ohne Probleme in eine numerische Zahl überführt werden kann.

Aufgabe 7

nicht so gut, viele dopplungen, lineares hasching, dauert irgendwann sehr lange, große lücken

c) Handelt es sich bei der folgenden Funktion um eine gute Hashfunktion?

```java
public int myHash(String key){
    //Startwert für große Streuung
    long start = System.currentTimeMillis();
    for(int i = 0; i<key.length();i++){
        //Multiplikation des Startwerts mit der numerischen Repräsentation der Buchstaben
        start = start * (int) key.charAt(i);
    }
    //Rückgabe des Modulowertes zur Einordnung in einem Array der Kapazität 8
    return (int) start%7;
}
```

Effizienz: linear, uni direktional: ja, deterministisch: nein, immer aktuelle uhrzeit, wenn man 100 mal den selben aufruf macht hat man ganz viel verschiedene ergebnisse, gleichwertig: nein baab hat den gleichen wert wie abba