Fachhochschule Nordwestschweiz
Hochschule für Technik

Algorithmen und Datenstrukturen 2
Prof. Dr. C. Stamm, A. Remund christoph.stamm@fhnw.ch Tel.: 056 462 47 44

Übung 7

Ausgabe: KW 47
Besprechung: KW 50

1 Einführung in Hash-Verfahren

1.1 Übersicht und Einordnung

Bis jetzt haben Sie schon viele dynamische Datenstrukturen kennen gelernt und die grosse Aufgabe war praktisch immer, die drei Operationen *Einfügen*, *Suchen* und *Entfernen* möglichst schnell anzubieten. Alle Ansätze haben in die Richtung abgezielt, dass man durch Vergleichen von dem gesuchten Schlüssel und Elementen in der Datenstruktur immer näher zu dem gesuchten Element kommt. Das schnellste was Sie kennen gelernt haben sind die AVL-Bäume, die alle drei Operationen in O(log(n)) anbieten.

Im Gegensatz dazu steht das Array, welches alle drei Operationen in O(1) anbietet. Es kann direkt auf einzelne Speicherzellen zugegriffen werden ohne umständliches Vergleichen und/oder Suchen. Leider sind Arrays nicht dynamisch, d.h. es ist nicht möglich, ein unendlich langes Array zu haben, sondern es muss im Vorfeld auf eine fixe Grösse limitiert werden.

In diesem Teil der Vorlesung werden Sie einen prinzipiell anderen Ansatz kennen lernen, der versucht, das schnelle Array mit der Unlimitiertheit der dynamischen Datenstrukturen zu verbinden. In der Praxis sind Hash-Verfahren absolut zentral. Sie werden in Bereichen der *Kryptographie* und der *Berechnung von Prüfsummen* eingesetzt, sowie für *Datenbanken* respektive die *Speicherung von Daten*. Wir werden uns vor allem mit dem letzten der drei Bereiche auseinander setzen.

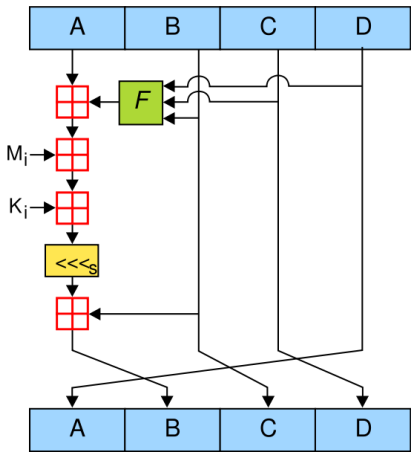
Zuerst lernen Sie das Prinzip von Hash-Verfahren an einem Beispiel aus der Praxis kennen. Danach gehen wir auf die mathematischen Grundlagen ein und befassen uns mit Hash-Funktionen, den Grundbausteinen von Hash-Verfahren. Dann ist alles vorhanden, um ganze Datenstrukturen zusammenzubauen und verschiedene Hash-Verfahren kennen zu lernen. Begleitet wird das immer durch konkrete Implementierungen in Java. Zum Abschluss werden wir uns anschauen, was die Java-Library in diesem Bereich bietet und wie deren Entwickler die Implementierungen realisiert haben.

1.2 Hash-Verfahren am Beispiel von MD5

Message-Digest Algorithm 5 (MD5) ist eine weit verbreitete, kryptographische Hash-Funktion, die aus einer beliebigen Nachricht einen 128-Bit-Hashwert erzeugt. Dieser Hash-Wert kann in praktischen Anwendungen als Prüfsumme verwendet werden, um entweder mit 100%iger Wahrscheinlichkeit die Unterschiedlichkeit zweier Nachrichten zu belegen oder mit hoher Wahrscheinlichkeit die Gleichheit zweier *sinnvoller* Nachrichten zu zeigen. Beachten Sie hier bitte die Asymmetrie, welche aus der Surjektivität der Abbildung einer sehr grossen Quellmenge auf eine kleine Zielmenge folgt.

MD5 wurde 1991 von Ronald L. Rivest am MIT entwickelt. Sie gilt inzwischen nicht mehr als sicher, da es mit überschaubarem Aufwand möglich ist, unterschiedliche Nachrichten zu erzeugen, die denselben MD5 Hash-Wert aufweisen. Wir sprechen in diesem Fall von Kollisionen. Heute werden ähnliche, aber sicherere Verfahren verwendet, welche unter den Namen SHA-1 („Secure Hash Algorithm“), SHA-2, SHA-256 usw. bekannt sind.

Ein MD5 Hash-Wert (auch Prüfsumme genannt) besteht aus 128 Bits, welche üblicherweise in Form von 32 Hexziffern notiert werden. Der MD5-Hashwert der Nachricht „Das Fach Algorithmen und Datenstrukturen 2 macht Spass!“ lautet beispielsweise: f82f66cc41959728ee887cf42c74ad91. Macht man sich bewusst, dass eine Hexziffer nur vier Bits repräsentiert, so sieht man, dass der generierte Hash-Wert deutlich kürzer ist als die ursprüngliche Nachricht, wo jedes ASCII-Zeichen 8 Bits repräsentiert. Von einer kryptographischen Hash-Funktion wird üblicherweise erwartet, dass zwei ähnliche Nachrichten zu komplett unterschiedlichen Prüfsummen führen, um Hackern keinen unnötigen Hinweis zu liefern. Ersetzen wir die ‚2‘ durch eine ‚1‘ in der Nachricht, so erhalten wir den folgenden Hash-Wert: f1975b75706191c63629214043b6d30e.



Die genaue Funktionsweise des Algorithmus wird in Wikipedia gut beschrieben: „Der Hauptalgorithmus von MD5 arbeitet mit einem 128-Bit-Puffer, der in vier 32-Bit-Wörter *A*, *B*, *C* und *D* unterteilt ist. Diese werden mit bestimmten Konstanten initialisiert. Auf diesen Puffer wird nun die Komprimierungsfunktion mit dem ersten 512-Bit-Block als Schlüsselparameter aufgerufen. Die Behandlung eines Nachrichtenblocks geschieht in vier einander ähnlichen Stufen, von Kryptographen ‚Runden' genannt. Jede Runde besteht aus 16 Operationen, basierend auf einer nichtlinearen Funktion ‚F', modularer Addition und Linksrotation. Es gibt vier mögliche ‚F'-Funktionen, in jeder Runde wird davon eine andere verwendet. Auf das Ergebnis wird dieselbe Funktion mit dem zweiten Nachrichtenblock als Parameter aufgerufen usw., bis zum letzten 512-Bit-Block. Als Ergebnis wird wiederum ein 128-Bit-Wert geliefert – die MD5-Summe."

Abbildung 1: Eine MD5-Operation. MD5 besteht aus 64 Operationen dieses Typs, gruppiert in 4 Durchläufen mit jeweils 16 Operationen. F ist eine nichtlineare Funktion, die im jeweiligen Durchlauf eingesetzt wird. M_i bezeichnet einen 32-Bit-Block des Eingabestroms und K_i eine für jede Operation unterschiedliche 32-Bit-Konstante; $<<<_s$ bezeichnet die bitweise Linksrotation um s Stellen, wobei s für jede Operation variiert. ⊞ bezeichnet die Addition modulo 2^{32}.

Übung 1
Studieren Sie den MD5-Algorithmus.

Quellmenge

Wenn wir von einer maximalen Nachrichtenlänge und einem endlichen Alphabet ausgehen, so ist die Menge aller möglichen Nachrichten endlich, aber unter Umständen extrem gross. Wir nennen diese Menge die **Quellmenge**. Man beachte, dass die Quellmenge im Allgemeinen auch sehr viele unsinnige Nachrichten enthalten kann, also beliebige Buchstabenfolgen, die keinen Sinn ergeben. Die Menge der sinnvollen Nachrichten wird also um ein Vielfaches geringer sein.

Zielmenge

Die **Zielmenge** besteht aus allen möglichen MD5 Hash-Werten, also 2^{128} verschiedenen Werten. Das Entscheidende ist nun, dass die Zielmenge im Allgemeinen viel kleiner als die Quellmenge ist und es zu einer surjektiven Abbildung kommt, wo mehrere Nachrichten aus der Quellmenge den gleichen MD5 Hash-Wert haben werden. Wenn zwei Nachrichten den gleichen Hash-Wert haben, so sprechen wir von einer **Kollision**.

Hash-Funktion

Der entscheidende Teil ist nun die Berechnung, die es uns erlaubt, von einer beliebigen Nachricht eine Zahl fixer Länge zu berechnen. Diese Berechnung ist nichts anderes als eine Funktion mit Parameter *Nachricht*. Wir nennen eine solche Funktion **Hash-Funktion**. Die damit errechnete Zahl heisst **Hash-Wert**. Von einer guten Hash-Funktion erwarten wir, dass unterschiedliche, sinnvolle Nachrichten zu möglichst unterschiedlichen Hash-Werten führen. Da die Menge der sinnvollen Nachrichten üblicherweise viel kleiner ist als die Quellmenge, darf erhofft werden, dass für fast alle sinnvollen Nachrichten unterschiedliche Hash-Werte generiert werden können.

Übung 2
In Java gibt es bereits Implementierungen der wichtigsten kryptographischen Hash-Algorithmen. Der nachfolgende Code-Ausschnitt zeigt die Verwendung von MD5.

```
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] dataBytes = …                // input

md.update(dataBytes);
byte[] mdbytes = md.digest();       // output
```

Schreiben Sie ein kleines Java-Programm, wo Sie eine Textnachricht eingeben und davon den MD5 Hash-Wert in hexidezimaler Schreibweise ausgeben können. Überprüfen Sie Ihr Programm mit den Beispielen auf der vorangehenden Seite.

1.3 Hash-Verfahren allgemein

Anhand dieses Beispiels können wir nun sämtliche Eigenschaften eines Hash-Verfahrens ableiten.

- Es gibt eine Quellmenge, die sehr gross, eventuell unendlich ist.
- Es gibt eine Zielmenge, die viel kleiner und endlich ist.
- Es gibt eine Hash-Funktion, die eine Abbildung von der Quellmenge auf die Zielmenge ermöglicht. Der errechnete Wert ist der Hash-Wert.
- Es kann zu Kollisionen kommen und es muss dafür eine Lösung gefunden werden.

Die Abbildung 2 veranschaulicht das. Links ist die Quellmenge, rechts die Zielmenge und die Pfeile geben an, wie die Hash-Funktion Elemente aus der Quellmenge auf die Zielmenge abbildet. Dort wo mehr als ein Element der Quellmenge auf die Zielmenge abgebildet werden, ist eine Kollision.

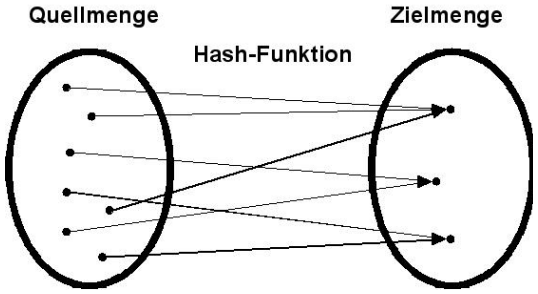


Abbildung 2: Hash-Verfahren Allgemein

Der grosse Vorteil von Hash-Verfahren ist, dass die Zielmenge beschränkt ist. Wir verwenden für die Grösse der Zielmenge die Variable *m*. Das erleichtert den Umgang extrem und ermöglicht z.B. ein schnelles Arrays als Datenspeicher zu verwenden. Um ein Element zu speichern, muss bloss sein Hash-Wert berechnet werden, welcher dann als Array-Index verwendet wird. So ist ein *Suchen*, *Einfügen* und *Löschen* mit O(1) möglich.

Übung 3
Beschreiben Sie, was die drei Begriffe *Quellmenge*, *Hash-Funktion* und *Zielmenge* im Beispiel der kryptographischen Hash-Algorithmen bedeuten.

Leider gibt es auch Probleme. Wie in unserem Beispiel gesehen, gibt es Elemente aus der Quellmenge, die nicht identisch sind, aber denselben Hash-Wert haben. Wenn wir davon ausgehen, dass die Quellmenge grösser ist als die Zielmenge, ist sofort klar, dass es gar nicht ohne Kollisionen gehen kann. Nebst den verschiedenen Hash-Funktionen, die wir kennen lernen werden, verlangt die Lösung dieses Problems unser Hauptaugenmerk.

Übung 4
Erklären Sie, weshalb Hash-Verfahren nicht in erster Linie auf Schlüsselvergleichen basieren wie z.B. Suchbäume.

1.4 Hash-Funktionen

In einem Array kann über den Index direkt auf einen Datensatz zugegriffen werden. Wenn ein Datensatz in ein Array eingefügt oder aus dem Array ausgelesen werden soll, muss also der Index des Datensatzes bekannt sein. Beim Hashing wird dieser Index mit Hilfe einer Hash-Funktion $h(k)$ aus einem oder mehreren Attributen berechnet. Üblicherweise wird bloss ein Attribut verwendet und dieses entsprechend *Suchschlüssel k* genannt. Alle möglichen Suchschlüssel bilden also unsere Quellmenge. Die daraus errechneten Array-Indizes sind entsprechend unsere Zielmenge und wir nennen diese Zielmenge auch Indexbereich.

Eine solche Hash-Funktion ist wie schon im vorhergehenden Kapitel erwähnt, üblicherweise nicht eineindeutig, das heisst, eine grosse Quellmenge K wird auf eine viel kleinere Zielmenge abgebildet. Somit kommt es vor, dass für verschiedene Suchschlüsselwerte x und y der gleiche Index $h(x) = h(y)$ berechnet wird. Suchschlüssel, welche mit einer gegebenen Hash-Funktion alle den gleichen Index erhalten, heissen **Synonyme**. Wichtig dabei ist zu realisieren, dass Synonyme nicht identisch sind. Wie beschrieben können sie sehr wohl verschieden sein und auch verschiedene Suchschlüssel haben, aber die Hash-Funktion errechnet für alle denselben Hash-Wert. Wenn zwei Objekte Synonyme sind, so sagen wir, dass sie **kollidieren**.

Eine gute Hash-Funktion sollte möglichst einfach und schnell berechnet werden können und die zu speichernden Datensätze möglichst gleichmässig auf den Indexbereich des Arrays verteilen, um Indexkollisionen (Synonyme) zu vermeiden. Dies soll selbst dann der Fall sein, wenn die Suchschlüssel alles andere als gleichverteilt sind. Dass dennoch Indexkollisionen, selbst bei einer optimal gewählten Hash-Funktion, wahrscheinlich sind, zeigt das *Birthday Paradox*: Wenn 23 oder mehr Personen in einem Raum sind, haben zu mehr als 50% Wahrscheinlichkeit zwei Personen am gleichen Tag des Jahres Geburtstag. Näherungsmässig gilt: Wenn eine Hash-Funktion $\sqrt{(\pi m/2)}$ Schlüssel auf einen Zielbereich der Grösse m abbildet, dann gibt es mit über 50% Wahrscheinlichkeit eine Indexkollision (für $m = 365$ ist $\lfloor\sqrt{(\pi m/2)}\rfloor = 23$).

Übung 5

a) Bestimmen Sie mit der oben aufgeführten Näherungsformel die Grösse der Zielmenge, wenn bereits eine Quellmenge von 100 Schlüsseln eine Kollisionswahrscheinlichkeit von 50% oder mehr zur Folge haben soll.
b) Mit welcher Formel lässt sich die Kollisionswahrscheinlichkeit berechnen?
c) Wie gross ist die Kollisionswahrscheinlichkeit für die Werte aus Teilaufgabe a?

1.5 Divisions-Rest-Methode

Ein nahe liegendes Verfahren zur Erzeugung eines Hash-Wertes ist es, den Rest einer ganzzahligen Division von k durch m zu nehmen:

$$h(k) = k \bmod m.$$

Für die Qualität dieser Hash-Funktion ist dann allerdings eine geschickte Wahl von m entscheidend. Ist etwa m eine gerade Zahl, so ist $h(k)$ gerade, wenn k gerade ist; ist k ungerade, so ist auch $h(k)$ ungerade. Das ist für viele Elemente aus der Quellmenge schlecht, z.B. dann, wenn die letzte Dualziffer einen Sachverhalt repräsentiert (0 = männlich, 1 = weiblich). Ebenfalls schlecht wäre die Wahl von m als Zweierpotenz, weil dadurch nur ein Teil der Binärziffern des Quellmengenelementes berücksichtigt wird: so liefert etwa $m = 2^i$ die letzten i Binärziffern von k für $h(k)$; die vorhergehenden Bits gehen überhaupt nicht in die Betrachtung ein.

Eine bessere Wahl ist, m als Primzahl zu wählen, welche keinen Teiler von $2^i \pm j$ ist, wobei i und j kleine natürliche Zahlen sind.

Übung 6

Berechnen Sie den Hash-Wert $h(k) = k \bmod m$, mit $k = 223$ und $m = 16$.
Ist 16 ein guter Wert für m? Begründen Sie Ihre Antwort.
Schlagen Sie selber einen guten Wert für m vor.

1.6 Multiplikative Methode

Der gegebene Schlüssel wird mit einer irrationalen Zahl multipliziert; der ganzzahlige Anteil des Resultats wird abgeschnitten. Auf diese Weise erhält man für verschiedene Schlüssel verschiedene Werte zwischen 0 und 1. Für Schlüssel 1, 2, 3, …, n sind diese Werte ziemlich gleichmässig im Intervall [0, 1) verstreut. Von allen irrationalen Zahlen führt der goldene Schnitt $\Phi = (\sqrt{5} - 1)/2$ zur gleichmässigsten Verteilung. Damit erhalten wir folgende Hash-Funktion:

$$h(k) = \lfloor m(k\Phi - \lfloor k\Phi \rfloor) \rfloor$$

Insbesondere bilden die Werte h(1), h(2), ... , h(10) für $m = 10$ gerade eine Permutation der Zahlen 0, 1, ... , 9.

Übung 7
Berechnen Sie den Hash-Wert von $k = 223$ mit $m = 16$ und Φ = goldener Schnitt.
Berechnen Sie der Reihe nach die Hash-Werte von $k = 1, ..., 10$ für $m = 10$ und Φ = goldener Schnitt.
Achten Sie dabei darauf, in welches Intervall (zwischen welche zwei bereits berechneten Hash-Werte) ein neuer Wert fällt. Es gibt dabei etwas Interessantes zu beobachten.
Bewerten Sie Ihre Beobachtung; ist diese Eigenschaft gut oder schlecht?

1.7 Perfektes und universelles Hashing

Ist die Anzahl der zu speichernden Schlüssel K nicht grösser als die Anzahl der zur Verfügung stehenden Speicherplätze m, so ist eine kollisionsfreie Speicherung immer möglich. Wenn wir K kennen und K fest bleibt, können wir leicht eine Abbildung h: $K \rightarrow \{0, ... , m - 1\}$ finden. Wir ordnen die Schlüssel in K lexikographisch und bilden jeden Schlüssel auf seine Ordnungsnummer ab. Wir haben damit eine **perfekte** Hash-Funktion, die Kollisionen gänzlich vermeidet. Im Allgemeinen ist K jedoch viel grösser als m und nur selten kennen wir den genauen Wert von K.

Übung 8
Ein Vertreter einer Software-Firma kommt zu Ihnen und präsentiert eine neu entwickelte Datenstruktur, die auf einem Hash-Verfahren basiert. Er schwärmt, dass sie eine Hash-Funktion entwickelt hätten, mit der man eine beliebig grosse Quellmenge auf eine fixe Zielmenge abbilden könne, und das kollisionsfrei! Was denken Sie über dieses Produkt?

Wir betrachten nun eine endliche Kollektion H von Hash-Funktionen. Jede Hash-Funktion aus H bildet alle denkbar möglichen Schlüsselwerte auf einen Index aus $\{0, 1, \ldots, m - 1\}$ ab. H heisst nun **universell**, wenn für je zwei verschiedene Schlüsselwerte x und y gilt:

$$\frac{\left|\{h \in H : h(x) = h(y)\}\right|}{|H|} \leq \frac{1}{m}$$

H ist also dann universell, wenn für jedes Paar von verschiedenen Schlüsseln höchstens der m-te Teil der Hash-Funktionen aus H zu einer Indexkollision für dieses Schlüsselpaar führen. Betrachten wir als Beispiel ein beliebiges, festes Paar von zwei verschiedene Schlüsseln x und y. Dann ist die Wahrscheinlichkeit dafür, dass x und y von einer zufällig aus H gewählten Funktion h auf denselben Hash-Wert abgebildet werden, höchstens $1/m$. Denn höchstens $1/m$ der Funktionen aus H führen zu einer Indexkollision bei x und y.

Solche universelle Kollektionen von Hash-Funktionen existieren und können sogar relativ leicht konstruiert werden. Dazu nehmen wir an, dass alle Schlüssel in der Menge $K = \{0, 1, \ldots, p - 1\}$ liegen, wobei p eine Primzahl ist. Für zwei beliebige Zahlen $a \in \{1, ... , p - 1\}$ und $b \in \{0, 1, \ldots, p - 1\}$ sei die Funktion $h_{a,b}(k)$ wie folgt definiert:

$$h_{a,b}(k) = ((a\,k + b) \bmod p) \bmod m.$$

Dann ist $H = \{ h_{a,b} \mid 1 \leq a < p \text{ und } 0 \leq b < p\}$ eine universelle Kollektion von Hash-Funktionen.

Mit Hilfe dieser Kollektion können wir uns folgende Strategie zur Wahl einer Hash-Funktion zu Grunde legen. Nehmen wir an, wir wissen, wie viele Schlüssel auf eine gegebene Hash-Tabelle der Grösse m abgebildet werden. Dann wählen wir eine Primzahl p, die grösser oder gleich der Anzahl der Schlüssel ist, und wählen zwei Zahlen a und b zufällig im entsprechenden Bereich. Dann ist $h_{a,b}$ eine „gute“ Hash-Funktion. Man beachte, dass wir (im Gegensatz zu der Divisions-Rest-Methode) keinerlei Voraussetzungen über die Grösse der Zielmenge m gemacht haben. Es schadet also nicht m beispielsweise als Zweierpotenz zu wählen.

Vielleicht ist Ihnen bis jetzt noch nicht klar geworden, für was wir das universelle Hashing überhaupt brauchen können. Beim Hashing kann es grundsätzlich vorkommen, dass alle Schlüssel im Worst-Case auf einen einzigen Hash-Wert abgebildet werden. Um diesen schlechtesten Fall zu umgehen, können wir nun eine zufällige Hash-Funktion aus der Menge H auswählen. Dadurch ist es nicht mehr möglich, mit Absicht eine spezielle Menge von Schlüsseln zu produzieren, die ein solches Worst-Case Verhalten hervorruft. Durch die zufällige Wahl der Hash-Funktion kann im vornherein nicht gesagt werden, welche Hash-Funktion verwendet wird und eine entsprechende Vorbereitung der Schlüssel ist nicht mehr möglich.

Übung 9
Gegeben sei eine Zielmenge der Grösse m = 10 und die Schlüsselfolge 1, 2, 3, ..., 20. Bestimmen Sie zuerst eine gute Hash-Funktion $h_{a,b}$ nach der Methode des universellen Hashings und verwenden Sie diese anschliessend.

1.8 Implementierung in Java

Nun haben Sie sämtliche Informationen beisammen, um eine Hash-Tabelle in Java zu programmieren. Wie zu Beginn beschrieben, wollen wir beliebige Objekte mit einem Schlüssel in unser Array einfügen. Wir wählen eine grosse Quellmenge von K = {0, 1, 2, ..., 10000} und ein Array der Grösse m mit m <<[1] K, also eine viel kleinere Zielmenge. Die Abbildung wird uns die Funktion der „Divisions-Rest-Methode“ $h(k) = k \bmod m$ ermöglichen.

Wir erstellen also eine neue Klasse `MyHashArray` mit den zwei privaten Member-Variablen `m_HashTable` und `m_m`. Die Erstere ist unsere Hash-Tabelle und die Letztere gibt die Grösse unserer Hash-Tabelle an. Im Konstruktor übergeben wir die Grösse und allozieren das Array.

```
public class MyHashArray<T> {
    private T[] m_HashTable;
    private int m_m;
    public MyHashArray(int size) {
        m_m = size;
        m_HashTable = (T[])new Object[m_m];
    }
}
```

Nun müssen wir Objekte in das Array einfügen können. Dazu verwenden wir die Methode `einfuegen`. Als Parameter hat sie natürlich das einzufügende Objekt, dazu aber auch noch einen Schlüssel aus der Quellmenge. Da unsere Quellemenge viel grösser ist als wir mögliche Array-Indizes haben, müssen wir hier nun zum ersten Mal die Hash-Funktion verwenden. Sie liefert von einem Schlüssel den Hash-Wert, welcher gerade der Index unseres Arrays ist. Diese Funktion haben wir noch nicht programmiert, aber wir bestimmen hier, dass diese Methode *h* heisst, logischerweise den Parameter *key* hat und einen *int* zurückgibt. Die ausprogrammierte Methode `einfuegen()` sieht dadurch so aus:

```
public void einfuegen(int key, T obj) {
    m_HashTable[h(key)] = obj;
}
```

Am besten wenden wir uns nun der Hash-Funktion *h* zu. Wir haben schon viele Annahmen gemacht und wissen deshalb, dass sie folgende Signatur haben wird:

```
private int h(int key)
```

Um in Java Modulo zu rechnen, gibt es den Operator *%*. Den Wert der Quellmenge bekommen wir als Parameter *key*, die Grösse des Arrays ist in der Member-Variable *m_m* gespeichert. Somit sieht die ganze Methode wie folgt aus:

```
private int h(int key) {
    return key % m_m;
}
```

Jetzt fehlen bloss noch die zwei Methoden `entfernen` und `suchen`. Die sind nun aber nicht mehr schwierig und so ergibt sich folgender Code:

```
public void entfernen(int key) {
    m_HashTable[h(key)] = null;
}
public T suchen(int key) {
    return m_HashTable[h(key)];
}
```

Um das Ganze noch abzurunden, möchten wir die Möglichkeit haben, unser Hash-Array schön auszugeben. Dazu überschreiben wir die Methode `toString()`. Diese Methode gibt einen `String` zurück, der jeden Array-Index plus seinen Inhalt beschreibt. Die Methode ist für die Implementierung des Hash-Arrays nicht nötig, hier aber trotzdem angegeben:

```
public String toString() {
    StringBuilder output = new StringBuilder();
    for (int i = 0; i < m_m; i++) {
        output.append("[Index \"" +i+"\", Object \""+m_HashTable[i]+"\"]; \n");
```

[1] << steht hier für „viel kleiner“ und nicht für den Shift-Operator

```
        }
        output.delete(output.length() - 3, output.length());
        return output.toString();
    }
```

Nun haben wir alles zusammen, um die `main`-Methode zu schreiben. Darin erzeugen wir zuerst eine Instanz dieser Klasse mit Grösse 7, fügen dann drei String-Objekte mit Schlüssel ein und geben danach das Array aus. Natürlich müssen wir auch noch `suchen` und `entfernen` testen. Dazu suchen wir zuerst ein Objekt, entfernen es und suchen noch einmal danach. Beim zweiten Mal sollte es natürlich nicht mehr gefunden werden. Die `main`-Methode sieht also so aus:

```
    public static void main(String[] args) {
        MyHashArray<String> myHashArray = new MyHashArray<String>(7);
        myHashArray.einfuegen(1, "eins");
        myHashArray.einfuegen(55, "fuenfundfuenfzig");
        myHashArray.einfuegen(87, "siebenundachtzig");
        System.out.println(myHashArray);
        System.out.println(myHashArray.suchen(1));
        myHashArray.entfernen(1);
        System.out.println(myHashArray.suchen(1));
    }
```

Hier zusammengefasst die ganze Klasse:

```
    public class MyHashArray<T> {
        private T[] m_HashTable;
        private int m_m;

        public MyHashArray(int size) {
            m_m = size;
            m_HashTable = (T[])new Object[m_m];
        }

        private int h(int key) {
            return key % m_m;
        }

        public void einfuegen(int key, T obj) {
            m_HashTable[h(key)] = obj;
        }

        public void entfernen(int key) {
            m_HashTable[h(key)] = null;
        }

        public T suchen(int key) {
            return m_HashTable[h(key)];
        }

        public String toString() {
            StringBuilder output = new StringBuilder();
            for (int i = 0; i < m_m; i++) {
                output.append("[Index \"" + i + "\", Object \"" + m_HashTable[i]
                        + "\"]; \n");
            }
            output.delete(output.length() - 3, output.length());
            return output.toString();
        }

        public static void main(String[] args) {
            MyHashArray<String> myHashArray = new MyHashArray<String>(7);
            myHashArray.einfuegen(1, "eins");
            myHashArray.einfuegen(55, "fuenfundfuenfzig");
            myHashArray.einfuegen(87, "siebenundachtzig");
            System.out.println(myHashArray);
            System.out.println(myHashArray.suchen(1));
            myHashArray.entfernen(1);
            System.out.println(myHashArray.suchen(1));
        }
    }
```

Übung 10
Sehen Sie in dieser Implementierung irgendwelche Probleme?
Ist diese Lösung in der Praxis so einsetzbar?
Was passiert, wenn Sie nach dem Einfügen mit Schlüssel 3 noch ein Objekt mit Schlüssel 10 einfügen?

2 Hash-Verfahren mit Verkettung der Überläufer

Im vorherigen Kapitel haben Sie zwei Hash-Funktionen kennen gelernt. Sie haben gesehen, wie es möglich ist, mit solchen Hash-Funktionen $h(k)$ aus dem zum Datensatz zugehörigen Suchschlüssel k einen Array-Index zu berechnen. In diesem Abschnitt geht es darum, noch ein letztes Problem zu lösen, um diese Idee als eine in der Praxis nutzbare Datenstruktur zu verwenden.

Das zu lösende Problem sind die Synonyme. Soll in ein Array, das bereits den Schlüssel k enthält, ein Synonym k' von k eingefügt werden, so ergibt sich eine Indexkollision. Der Platz $h(k) = h(k')$ ist bereits besetzt und k', ein **Überläufer**, muss anderswo gespeichert werden. Eine einfache Art, Überläufer zu speichern, ist die, sie ausserhalb des Arrays abzulegen, und zwar in dynamisch veränderbaren Strukturen. So kann man etwa die Überläufer zu jedem Array-Index in einer linearen Liste verketten; diese Liste wird an den Array-Eintrag angehängt, der sich durch Anwendung der Hash-Funktion auf die Schlüssel ergibt.

Je nachdem, ob die Listen nur für die Überläufer verwendet werden oder ob alle Datensätze in Listen abgespeichert werden, spricht man von separater Verkettung der Überläufer oder direkter Verkettung.

2.1 Separate Verkettung der Überläufer

Bei der separaten Verkettung der Überläufer ist jedes Element der Hash-Tabelle das Anfangselement einer Überlaufkette (verkettete lineare Liste). Angenommen wir hätten eine Klasse `List` mit einer inneren Klasse `List.Element`[2]. Somit können wir ein Array von solchen Elementen als unsere Hash-Tabelle verwenden. Die Klasse `HashSet` zur Speicherung einer Menge von Schlüsseln sieht dann wie folgt aus:

```
class HashSet {
    private List.Element[] m_hashTable;
    public HashSet(int m) {
        m_hashTable = new List.Element[m];
        // Einträge erstellen
    }
    public boolean search(int key) { … }
    public void insert(int key) { … }
    public void delete(int key) { … }
}
```

Beispiel:

Grösse des Arrays $m = 7$
$K = \{0, 1, \ldots, 500\}$
$h(k) = k \bmod m$

Nun werden in die leere Hash-Tabelle die Schlüssel 12, 53, 5, 15, 2, 19, 43 in dieser Reihenfolge eingefügt. Daraus ergibt sich folgende Situation.

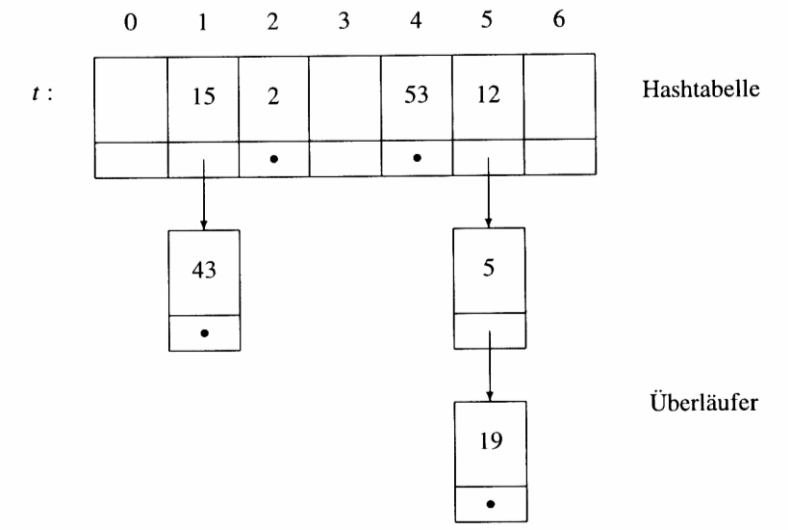


Abbildung 3: Hash-Tabelle mit separater Verkettung der Überläufer

[2] Sie wissen ja, dass jedes Element einen Zeiger auf das nächste Element hat und sich dadurch eine einfach verkettete Liste ergibt.

Betrachten wir nun die drei Basisoperationen Suchen, Einfügen und Entfernen für eine Hash-Tabelle *t*.

- Suchen nach Schlüssel *k*: Beginne bei *t*[*h*(*k*)] und folge den Verweisen der Überlaufkette, bis entweder *k* gefunden wurde (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist (erfolglose Suche).
- Einfügen eines Schlüssels *k*: Suche nach *k*; wird *k* gefunden, so fügen wir nichts ein; wird *k* nicht gefunden, so fügen wir den Schlüssel am Ende der Überlaufkette, welche in *t*[*h*(*k*)] beginnt, ein.
- Entfernen eines Schlüssels *k*: Suche nach *k*; wird *k* nicht gefunden, so gibt es nichts zu entfernen; wird *k* gefunden, so entferne den gefundenen Schlüssel.

Bei dieser Implementierung fällt allerdings auf, dass die unterschiedlichen Fälle (Schlüssel in Hash-Tabelle oder in Überlaufkette, gegebenenfalls Nachziehen des ersten Überläufers in die Hash-Tabelle oder beim Entfernen, usw.) einige Abfragen erfordert, die die Laufzeit der Operationen spürbar beeinträchtigen. Wenn man bereit ist, unter Umständen etwas Speicherplatz zu opfern, so kann man auch einfach alle Datensätze in den Überlaufketten speichern; in der Hash-Tabelle benötigt man dann nur Zeiger auf den Listenanfang. Diese Implementierung nennt sich „Direkte Verkettung der Überläufer“.

Übung 11
Fügen Sie in das vorhergehende Beispiel zusätzlich die Schlüssel 9 und 26 ein. Entfernen Sie danach den Schlüssel 5.

2.2 Direkte Verkettung der Überläufer

Bei der direkten Verkettung der Überläufer ist jedes Element der Hash-Tabelle eine eigenständige Liste. In der Hash-Tabelle werden also bloss Referenzen auf Listen gespeichert und die Datensätze in die Listen eingefügt. Somit wird die Klasse `HashSet` einfacher:

```
class HashSet {
    private List[] m_hashTable;
    public HashSet(int m) {
        m_hashTable = new List[m];
        // Einträge erstellen
    }
    public boolean search(int key) { … }
    public void insert(int key) { … }
    public void delete(int key) { … }
}
```

Schauen wir uns wieder das Beispiel aus dem vorherigen Abschnitt an: eine Hash-Tabelle der Grösse *m* = 7 und die Hash-Funktion *h*(*k*) = *k* mod *m*. Nach Einfügen der Schlüssel 12, 53, 5, 15, 2, 19, 43 in dieser Reihenfolge in die anfangs leere Hash-Tabelle ergibt sich die folgende Situation.

Die drei Basisoperationen Suchen, Einfügen und Entfernen laufen mehrheitlich gleich ab, wie bei der

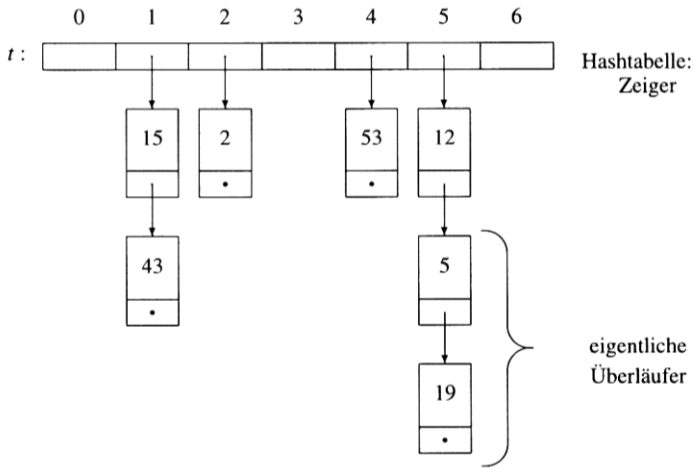


Abbildung 4: Hash-Tabelle mit direkter Verknüpfung der Überläufer

Hash-Tabelle mit separater Verkettung der Überläufer. Wir wollen an dieser Stelle noch ein Mass herleiten, das uns über die Speicherplatz-Verwendung einer Hash-Tabelle Auskunft gibt. Die Grösse

der Hash-Tabelle nennen wir jeweils *m*. Je nachdem, wie viele Datensätze gerade gespeichert sind ist die Tabelle mehr oder weniger belegt; wir bezeichnen die aktuelle Anzahl gespeicherter Datensätze mit *n*. Um nun anzugeben, wie stark eine Tabelle gefüllt ist, nennen wir den Quotienten aus *n* und *m*, $\alpha = n/m$, den **Belegungsfaktor** der Hash-Tabelle. Je grösser der Belegungsfaktor ist, desto „voller" ist unsere Hash-Tabelle. Die Anzahl der zum Suchen, Einfügen oder Entfernen eines Schlüssels benötigten Schritte hängt im Wesentlichen von diesem Faktor ab. Er wird also bei der kommenden Analyse eine grosse Rolle spielen.

Übung 12
Fügen Sie in das vorherige Beispiel zusätzlich die Schlüssel 10 und 26 ein. Entfernen Sie danach den Schlüssel 53.
Berechnen Sie jetzt den Belegungsfaktor α für das von Ihnen weitergeführte Beispiel.

2.3 Analyse

Wir nehmen an, dass die Hash-Funktion alle Hash-Werte (Indizes) mit gleicher Wahrscheinlichkeit (Gleichverteilung) und von Operation zu Operation unabhängig liefert; d.h. die Wahrscheinlichkeit, dass bei der *j*-ten Operation der Index *i* ausgewählt wird, ist unabhängig von *j* stets gleich $1/m$, für alle *i*.

Bei einer erfolglosen Suche nach *k* betrachten wir alle Einträge der bei $t[h(k)]$ beginnenden Überlaufkette. Die durchschnittliche Anzahl der Einträge in einer Kette ist n/m, wenn *n* Einträge auf *m* Ketten verteilt sind. n/m entspricht dem Belegungsfaktor α. Da dem Einfügen eines neuen Datensatzes im Normalfall eine erfolglose Suche vorangeht, ist die mittlere Zeitkomplexität von Einfügen proportional zu α.

Ist bei einer erfolgreichen Suche nach *k*, *k* um *d* Listenelemente vom Listenanfang $t[h(k)]$ entfernt, so werden gerade diese *d* Einträge betrachtet. Beim Einfügen des *j*-ten Schlüssels ist die durchschnittliche Listenlänge $(j - 1)/m$. Also betrachten wir bei einer späteren Suche nach dem *j*-ten Schlüssel gerade $1 + (j - 1)/m$ Einträge im Durchschnitt, wenn stets am Listenende eingefügt wird und kein Schlüssel entfernt wurde. Im Mittel ist die Anzahl der bei der erfolgreichen Suche nach einem Schlüssel betrachteten Einträge also

$$\frac{1}{n}\sum_{j=1}^{n}\left(1+\left(j-1\right)/m\right)=1+\frac{n-1}{2\mathrm{m}}\approx 1+\frac{\alpha}{2}$$

wenn nach jedem Schlüssel mit gleicher Wahrscheinlichkeit gesucht wird.

Übung 13
Sie hatten im vorhergehenden Abschnitt die Schlüssel 10 und 26 eingefügt sowie den Schlüssel 53 entfernt. Berechnen Sie nun, wie gross die allgemeine, mittlere Anzahl betrachteter Einträge für eine erfolgreiche Suche ist. Vergleichen Sie das erhaltene Ergebnis mit der konkreten, mittleren Anzahl betrachteter Einträge für eine erfolgreiche Suche.

2.4 Implementierung in Java

Eine schöne Aufgabe ist es nun, diese in der Theorie kennen gelernte Datenstruktur zu implementieren. Dazu werden wir jetzt zuerst ein Interface definieren, um die grundsätzlichen Möglichkeiten einer Hash-Map festzulegen. Eine Hash-Map ist ein Wörterbuch (Dictionary oder Map), wo zu jedem eindeutigen Schlüssel ein zusätzliches Datenobjekt abgespeichert werden kann. Eine Hash-Map ist also eine eindeutige Zuordnung von beliebigen Datenobjekten zu eindeutigen Schlüsselobjekten. Die Klasse `HashMap` soll mindestens die folgenden Methoden anbieten:

- `public int getSize()`: gibt die Anzahl Elemente der Map zurück;
- `public boolean contains(int key)`: gibt `true` zurück, falls ein Element mit Schlüssel `key` in der Map enthalten ist; ansonsten `false`;
- `public T get(int key)`: gibt das zum `key` passende Datenobjekt zurück, falls der `key` in der Map enthalten ist; ansonsten `null`;
- `public void put(int key, T data)`: fügt zu einem Schlüssel `key` ein Datenobjekt `data` in die Map ein; falls der Schlüssel `key` bereits enthalten ist, wird das Datenobjekt überschrieben;
- `public void remove(int key)`: entfernt den Schlüssel `key` und das zugehörige Datenobjekt aus der Map.

Das Interface sieht somit so aus:

```
public interface HashMap<T> {
    public int getSize();
```

```
    public boolean contains(int key);
    public T get(int key);
    public void put(int key, T data);
    public void remove(int key);

}
```

Wir möchten die „Direkte Verkettung der Überläufer“ implementieren. Dazu können wir nun eine Klasse `DirectLinkedHashMap` erstellen, die das Interface `HashMap` implementiert:

```
public class DirectLinkedHashMap<T> implements HashMap<T>
```

Um die Überlaufketten zu implementieren, müssen wir eine Liste verwenden. Sehr gut eignet sich dazu die `LinkedList` aus dem *Java Collection Framework*. Dabei muss man aber angeben, was für ein Typ diese Liste speichern soll. In der Hash-Map sollen Schlüssel-Daten-Paare abgespeichert werden können. Da wir aber in der Liste nur ein Objekt pro Eintrag abspeichern können, ist es notwendig, das Schlüssel-Daten-Paar in einem Objekt einer Klasse `Element` zusammenzufügen. Eine solche Klasse `Element` besitzt also genau zwei Instanzvariablen: einen Schlüssel (`m_key`) vom Typ `Integer` und die zugehörigen Daten (`m_data`) von einem beliebigen Datentyp.

Damit die Instanzen der Klasse `Element` in die Überlaufliste eingefügt werden können, ist es notwendig, dass die Klasse `Element` selber die Methode `boolean equals(Object o)` überschreibt. Dies ist aber denkbar einfach, weil die Umsetzung davon direkt an die Instanzvariable `m_key` delegiert werden kann. Die innere Klasse sieht also folgendermassen aus:

```
private class Element {
    private T m_data;
    private Integer m_key;
    public Element(int key, T data) {
        m_data = data;
        m_key = key;
    }
    public boolean equals(Object o) {
        return m_key.equals(((Element)o).m_key);
    }
}
```

Die Methode `getSize()` soll die Anzahl Elemente in der Hash-Map zurückgeben. Dazu ist es sinnvoll, eine Instanzvariable `m_size` auf dem aktuellen Stand zu halten und dann in der Funktion `getSize()` nur noch den Wert von `m_size` zurückzugeben.

In `contains(…)` muss lediglich entschieden werden, ob der gesuchte Schlüssel vorhanden ist oder nicht. Das ist der einfachste Fall. Wir berechnen einfach mit unserer Hash-Funktion die Adresse der Liste und schauen nach, ob unser Element dort vorkommt.

In `get(…)` muss nicht nur gesucht werden, sondern auch auf die gefundenen Daten zugegriffen werden können. Wir müssen also auch zuerst die richtige Liste mit Hilfe der Hash-Funktion finden, diese dann durchlaufen und das gesuchte Element finden. Sobald wir es gefunden haben, geben wir es zurück und brechen ab; falls nicht geben wir einfach `null` zurück.

Die Methode `put(…)` wiederum ist sehr einfach. Zuerst muss mittels der Hash-Funktion die richtige Liste berechnet werden und dann das Element eingefügt werden. Dabei dürfen wir bloss nicht vergessen, den Zähler `m_size` mitzuführen.

Zum Schluss bleibt noch die Methode `remove(…)`. Aber auch diese ist sehr ähnlich wie `contains(…)` zu implementieren: zuerst die richtige Liste bestimmen, dann ein neues `Element` erstellen, das als `key` den zu entfernenden Schlüssel hat und mit diesem neuen Element die `remove`-Methode der Liste ausführen. Da unsere selber implementierte `equals()`-Methode ja nur auf dem Vergleich der Schlüssel basiert, ist es egal, wenn wir als Datenobjekt `null` übergeben. Somit können wir einfach das Element aus der Liste löschen lassen und den Zähler `m_size` dekrementieren.

Bis jetzt haben wir noch gar nicht über die Hash-Funktion gesprochen. Im vorhergehenden Kapitel haben wir uns schon intensiv damit auseinander gesetzt und greifen nun darauf zurück. Wir verwenden die „Divisions-Rest-Methode“.

Um das Ganze noch abzurunden, möchten wir die Möglichkeit haben, unser Hash-Set schön auszugeben. Dazu überschreiben wir am besten die Methode `toString()`. Diese Methode gibt einen `String` zurück, der jeden Array-Index plus den Inhalt der Listen beschreibt. Die Methode ist für die Implementierung des Hash-Set nicht nötig, hier aber trotzdem angegeben:

```
public String toString() {
    StringBuilder output = new StringBuilder();
    for (int i = 0; i < m_m; i++) {
        output.append("[Index " + i + ", Elemente");
        for (Element element : m_HashTable[i]) {
            output.append("\n\t " + element.m_key);
        }
        output.append("];\n");
    }
    output.delete(output.length() - 2, output.length());
    return output.toString();
}
```

Unsere Hash-Map kann nun sehr schön ausprobiert werden:

```
HashMap<String> hashMap3 = new DirectLinkedHashMap<String>(7);
hashMap3.put(12, null);
hashMap3.put(53, null);
hashMap3.put(5, null);
hashMap3.put(15, null);
hashMap3.put(2, null);
hashMap3.put(19, null);
hashMap3.put(43, null);
System.out.println(hashMap3);
System.out.println(hashMap3.getSize());
hashMap3.remove(19);
System.out.println(hashMap3.getSize());
System.out.println(hashMap3.contains(19));
```

3 Offene Hash-Verfahren

Bis jetzt haben wir einfach die Hash-Tabelle aufgefüllt und im Falle einer Kollision die Überläufer ausserhalb der Hash-Tabelle gespeichert, typischerweise in einer einfach verketteten Liste. Wenn auf diese Art viele Datensätze ausgelagert werden müssen, wird der Datenzugriff immer langsamer und ihr Laufzeitverhalten unvorhersehbarer.

Mit der Idee von offenen Hash-Verfahren wird ein anderer Ansatz verfolgt, nämlich die Überläufer innerhalb der Hash-Tabelle unterzubringen. Wenn also beim Versuch den Schlüssel *k* in die Hash-Tabelle an Position *h*(*k*) einzutragen festgestellt wird, dass *t*[*h*(*k*)] bereits belegt ist, so muss man nach einer festen Regel einen anderen, nicht belegten Platz (eine offene Stelle) finden, an dem man *k* unterbringen kann. Da man von vornherein nicht weiss, welche Plätze belegt sein werden und welche nicht, definiert man für jeden Schlüssel eine Reihenfolge, in der alle Speicherplätze einer nach dem anderen betrachtet werden. Sobald dann ein betrachteter Platz frei ist, wird der Datensatz dort gespeichert. Die Magie liegt also darin, wie man abhängig vom jeweiligen Schlüssel, die Hash-Tabelle inspiziert. Diese Reihenfolge nennt sich **Sondierungsfolge**.

Die Sondierungsfolge ist von dem jeweiligen Schlüssel abhängig und möglichst individuell. Methoden, die diesem Schema folgen, werden offene Hash-Verfahren genannt. Bei diesen Verfahren geht es praktisch immer um die Wahl einer geeigneten Sondierungsfolge. Sie werden in diesem Kapitel zwei Varianten kennen lernen.

3.1 Lineares Sondieren

Beim linearen Sondieren ergibt sich für den Schlüssel *k* die Sondierungsfolge

h(k), *h(k)* - 1, *h(k)* - 2, *h(k)* - 3, ..., 0, *m* - 1, *m* - 2, *m* - 3, ..., *h(k)* + 1

Es wird einfach immer ein Array-Index kleiner versucht, bis der kleinste Index erreicht wird (also 0) und dann wird einfach vom höchsten Index an weiter gesucht.

Beispiel:

Grösse des Arrays *m* = 7
K = {0, 1, ... , 500}
h(k) = *k* mod *m*

Nun werden in die leere Hash-Tabelle die Schlüssel 12, 53, 5, 15, 2, 19 in dieser Reihenfolge eingefügt. Das ergibt nach Einfügen von 12, 53:

	0	1	2	3	4	5	6
t:					53	12	

Nach einfügen von *5: h(*5) = 5 mod 7 = 5 ist belegt; der nächste Index der Sondierungsfolge ist 4, ebenfalls belegt; der nächste Index ist 3, nicht belegt:

	0	1	2	3	4	5	6
t:				5	53	12	

Nach einfügen von 15, 2, 19 (Sondierungsfolge 5-4-3-2-1-0):

	0	1	2	3	4	5	6
t:	19	15	2	5	53	12	

Übung 14
Was passiert, wenn Sie jetzt noch ein Datenelement mit Schlüssel 13 einfügen?
Was passiert, wenn Sie anschliessend nach dem Datenelement mit Schlüssel 3 suchen?
Formulieren Sie eine Bedingung, damit solche Probleme nicht vorkommen!

Das lineare Sondieren ist zwar ein sehr einfaches Verfahren, hat aber auch einige Nachteile. Schauen wir uns die Wahrscheinlichkeit an, mit der ein Schlüssel an einer gewissen Hash-Adresse gespeichert wird. Im gezeigten Beispiel werden Elemente mit Schlüssel 12, 53 und 5 eingefügt. Im Eintrag *t*[2] werden nun alle Schlüssel *k* mit *h(k)* = 2 oder *h(k)* = 3 oder *h(k)* = 4 oder *h(k)* = 5 gespeichert, im Eintrag *t*[1] dagegen nur alle Schlüssel *k* mit *h(k)* = 1. Sie sehen, dass die Wahrscheinlichkeit in der Hash-Tabelle für die verschiedenen Hash-Adressen drastisch verschieden ist. Lange belegte Teilstücke der Hash-Tabelle haben also eine stärkere Tendenz zu wachsen als kurze. Dieser Effekt wird noch verstärkt, weil lange Teilstücke zu grösseren zusammenwachsen. Als Folge dieses Phänomens verschlechtert sich die Effizienz des linearen Sondierens drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert.

3.2 Entfernen-Problem bei offenen Hash-Verfahren

Leider gibt es noch ein weiteres Problem zu beklagen und dieses beschränkt sich nicht nur auf das lineare Sondieren. Für alle offenen Hash-Verfahren stellt sich die Frage: Wie kann ein Wert entfernt werden, ohne die Sondierungsfolge zu zerstören?

Beispiel:

Grösse des Arrays $m = 7$
$K = \{0, 1, \ldots, 500\}$
$h(k) = k \bmod m$
Sondierungsfolge: lineares Sondieren

Nach einfügen der Schlüssel 53 und 12 ergibt sich folgendes Bild:

	0	1	2	3	4	5	6
t:					53	12	

Nun möchten wir wie vorhin ein Datenelement mit Schlüssel 5 einfügen. Da *h(*5) = (5 mod 7) = 5 belegt ist, gehen wir gemäss unsere Sondierungsfolge zum Index 4, welcher ebenfalls belegt ist. Der nächste Index ist 3 und wir können das Datenelement dort ablegen. Es ergibt sich folgendes Bild:

	0	1	2	3	4	5	6
t:				5	53	12	

Nun kommen wir zum eigentlichen Knackpunkt dieses Beispiels: Wir möchten das Datenobjekt mit Schlüssel 12 entfernen. Natürlich ist das nicht sofort ein Problem, denn wir können einfach das Datenobjekt suchen und falls gefunden entfernen. Die Hash-Tabelle würde dann so aussehen:

	0	1	2	3	4	5	6
t:				5	53		

Wie finden wir nun aber das Datenobjekt mit Schlüssel 5? Wir errechnen den Array-Index, schauen dort nach und sehen kein Datenobjekt. Wir nehmen deshalb mit gutem Recht an, dass unser gesuchtes Datenobjekt dort sein würde, falls es in der Hash-Tabelle wäre. Es hätte ja Platz gehabt. Nun ist aber nichts dort und die Suchen-Methode entscheidet folglich auf: nicht vorhanden.

Lösung

Ein bereits in der Hash-Tabelle vorhandener Schlüssel *k* versperrt ja einem neu einzufügenden Schlüssel *k'* im Allgemeinen einen Platz. Der neue Schlüssel *k'* weicht also auf einen anderen Platz aus (gemäss seiner Sondierungsfolge). Wird nun *k* entfernt, so kann *k'* nicht wieder gefunden werden. Deshalb wird bei den meisten Verfahren *k* auch nicht wirklich entfernt, sondern lediglich **als entfernt markiert**. Wird ein neuer Schlüssel eingefügt, so wird der Platz von *k* als frei angesehen; wird ein Schlüssel gesucht, so wird der Platz von *k* als belegt angesehen.

In unserem Beispiel würde also der Schlüssel 12 nicht wirklich entfernt, sondern bloss als entfernt markiert:

	0	1	2	3	4	5	6
t:				5	53	*(12)*	

So kann erkennt werden, dass beim Suchen nach Schlüssel 5 der Sondierungsfolge entlang weiter gesucht werden muss. Wird nun ein Datenelement mit Schlüssel 19 eingefügt, kann die Position 5 im Array getrost überschrieben werden, der Platz ist ja eigentlich frei:

	0	1	2	3	4	5	6
t:				5	53	19	

Übung 15

Sie haben eine Hash-Tabelle mit folgenden Angaben:

Grösse des Arrays $m = 7$
$K = \{0, 1, ... , 500\}$
$h(k) = k \bmod m$
Sondierungsfolge: lineares Sondieren

Schreiben Sie alle Teilschritte der folgenden Operationen auf:

Fügen Sie nacheinander die Schlüssel 14, 8, 13, 10, 12 und 15 ein.
Entfernen Sie nun Schlüssel 10 und 13.
Wird Schlüssel 15 noch gefunden?
Fügen Sie nun zum Schluss noch Schlüssel 17 ein.

3.3 Schema für offene Hash-Verfahren

Wir definieren nun ein Schema für offene Hash-Verfahren, das sich für die meisten offenen Hash-Verfahren als Grundlage eignet.

Sei *s(j,k)* eine Funktion von *j* und *k* so, dass $(h(k) - s(j,k)) \bmod m$ für $j = 0, 1, ..., m - 1$ eine Sondierungsfolge bildet, d.h. eine Permutation aller Hash-Adressen. Es sei stets noch mindestens ein Platz in der Hash-Tabelle frei.

- Suche nach Schlüssel *k*: Beginne mit Hash-Adresse $i = h(k)$. Solange *k* nicht in $t[i]$ gespeichert ist und $t[i]$ nicht frei ist, suche weiter bei $i = (h(k) - s(j,k)) \bmod m$, für aufsteigende Werte von *j*. Falls $t[i]$ belegt ist, wird *k* gefunden; sonst ist die Suche erfolglos.
- Einfügen eines Schlüssels *k*: Wir nehmen an, dass *k* nicht schon in *t* vorkommt (das kann durch eine Suche festgestellt werden). Beginne mit Hash-Adresse $i = h(k)$. Solange $t[i]$ belegt ist, mache weiter bei $i = (h(k) - s(j,k)) \bmod m$, für steigende Werte von *j*. Trage *k* bei $t[i]$ ein.
- Entfernen eines Schlüssels k: Suche nach Schlüssel k. Verläuft die Suche erfolgreich und ist i die Adresse, an der *k* gefunden wird, dann markiere $t[i]$ als entfernt; sonst kommt *k* nicht in *t* vor und kann auch nicht entfernt werden.

Bei dem zu Beginn erklärten *linearen Sondieren* ist also die Funktion $s(j,k) = j$.

3.4 Double-Hashing

Beim linearen Sondieren sind doch gravierende Nachteile aufgetaucht, die wir mit Hilfe des *Double-Hashing* beheben möchten. Dabei wird für die Sondierungsfolge, wie der Name schon sagt, eine zweite Hash-Funktion verwendet. Die gewählte Sondierungsfolge für Schlüssel *k* ist

*h(k), h(k) – h'(k), h(k) – 2*h'(k), ..., h(k) – (m – 1)*h'(k)*

wenn *h'(k)* die zweite Hash-Funktion bezeichnet. Damit wir keine Indizes errechnen, die kleiner 0 sind, wird das Resultat jeweils noch modulo *m* gerechnet. Daraus folgt schliesslich die Sondierungsfolge:

h(k) mod *m*
(*h(k) – h'(k))* mod *m*
*(h(k) – 2*h'(k))* mod *m*
...
*(h(k) – (m – 1)*h'(k))* mod *m*

Für das soeben hergeleitete, allgemeine Schema ergibt sich: *s(j,k) = j * h'(k)*

Dabei muss *h'(k)* so gewählt werden, dass für alle Schlüssel *k* die Sondierungsfolge eine Permutation der Hash-Adressen bildet. Das bedeutet, dass *h'(k)* ≠ 0 sein muss und *m* nicht teilen darf. Wählen wir *m* als Primzahl, dann gilt dies sicher für jedes *h'(k)* und für alle *k*. Ist *m* eine Primzahl und *h(k)* = *k* mod *m*, so erfüllt *h'(k)* = 1 + *k* mod (*m* – 2) die obigen Anforderungen (das ist besser als 1 + *k* mod (*m* – 1), weil m – 1 gerade ist).

Beispiel:

Grösse des Arrays *m* = 7
K = {0, 1, ... , 500}
h(k) = *k* mod m
*h'(*k) = 1 + k mod 5

Nun werden in die leere Hash-Tabelle die Schlüssel 12, 53, 5, 15, 2, 19 in dieser Reihenfolgen eingefügt. Das ergibt nach einfügen von 12, 53:

	0	1	2	3	4	5	6
t:					53	12	

Nach einfügen von 5 (Sondierungsfolge ist *h*(5) = 5 mod 7 = 5, 5 – (1 + 5 mod 5) = 4, 5 – 2*(1 + 5 mod 5) = 3), 15, 2:

	0	1	2	3	4	5	6
t:		15	2	5	53	12	

Nach einfügen von 19 (Sondierungsfolge *h*(19) = 19 mod 7 = 5, 5 – (1 + 19 mod 5) = 0):

	0	1	2	3	4	5	6
t:	19	15	2	5	53	12	

Beim Einfügen des Schlüssels 19 müssen hier also lediglich zwei Plätze (nämlich *t*[5] und *t*[0]) inspiziert werden, während es beim linearen Sondieren sechs Plätze waren. Da Double-Hashing ein leicht zu implementierendes Verfahren ist, bietet es sich als praktisch einsetzbares, offenes Hash-Verfahren an.

Übung 16

Sie haben eine Hash-Tabelle mit folgenden Angaben:

Grösse des Arrays *m* = 7
K = {0, 1, ... , 500}
h(k) = *k* mod *m*
Sondierungsfolge: Double-Hashing
h'(k) = 1 + *k* mod 5

Schreiben Sie alle Teilschritte der folgenden Operationen auf:

Fügen Sie nacheinander die Schlüssel 14, 8, 13, 10, 12 und 15 ein.
Entfernen Sie nun Schlüssel 10 und 13.
Wird Schlüssel 15 noch gefunden?
Fügen Sie nun zum Schluss noch Schlüssel 17 ein.

Weshalb darf $h'(k)$ für beliebige Werte aus K nicht 0 geben?

3.5 Implementierung in Java

In diesem Abschnitt wird eine mögliche Implementierung in Java vorgestellt. Aus vorherigen Übungen ist schon viel vorhanden, auf das nun Rücksicht genommen werden muss. Als erstes soll unsere neue Version natürlich auch das Interface `HashMap<T>` implementieren.

Des Weiteren ist in der Theorie ein allgemeines Schema für offene Hash-Verfahren erklärt worden. Auch das möchten wir in unsere Design-Überlegungen einfliessen lassen. Doch wie ist das möglich? Wenn Sie den Text-Abschnitt des allgemeinen Schemas noch einmal anschauen, bemerken Sie, dass eigentlich alle Operationen, die unser Interface erfordert, beschrieben werden können. Dabei werden zwei Funktionen angenommen: *h(k)* und *s(j,k)*. Erstere errechnet den Hash-Wert für einen Schlüssel und letztere gibt uns die Sondierungsfolge. Genau dieses Prinzip können wir aber auch in der Java-Implementierung anwenden, indem wir eine abstrakte Klasse schreiben. Wir nennen diese abstrakte Klasse `OpenHashMap` und programmieren dort sämtliche Methoden bis auf die erwähnten zwei Methoden *h(k)* und *s(j,k)*. Die Klassen-Definition sieht also wie folgt aus:

```
abstract public class OpenHashMap<T> implements HashMap<T>
```

Die einzigen zwei Methoden, die in diesem Moment noch nicht programmiert werden sollen, definieren wir einfach als abstrakte Methoden. Die Klasse beginnt also so:

```
abstract public class OpenHashMap<T> implements HashMap<T> {

    abstract int h(int key);

    abstract int s(int j, int key);
}
```

Als nächstes müssen wir uns dem Problem zuwenden, dass die Einträge in unserer Hash-Tabelle die drei Zustände *frei*, *belegt* und *entfernt* haben können. Für Zustände bietet hier Java auch wieder ein Lösung an: Enumerations. Wir definieren also in der Klasse `OpenHashMap` den `enum`-Typ `Zustand`:

```
private static enum Zustand {
    FREI, BELEGT, ENTFERNT
};
```

Für die Elemente unserer Hash-Tabelle brauchen wir einen eigenen Element-Typ, um den Schlüssel, die Daten und den Zustand abzuspeichern. Dazu schreiben wir eine innere Klasse `Element`, deren Typ unsere Hash-Tabelle haben soll. Diese innere Klasse sieht wie folgt aus:

```
private class Element {
    private T m_data;
    private int m_key;
    private Zustand m_zustand;

    public Element(int key, T data) {
        m_data = data;
        m_key = key;
        zustand = Zustand.FREI;
    }
}
```

Wie üblich werden wir die Hash-Tabelle als private Member-Variable speichern und in einem Zähler die Anzahl verwendeter Elemente mitführen. Die Klasse `OpenHashMap` sieht bis jetzt wie folgt aus:

```
abstract public class OpenHashMap<T> implements HashMap<T> {

    private static enum Zustand {
        FREI, BELEGT, ENTFERNT
    };

    private class Element {
        private T m_data;
        private int m_key;
        private Zustand m_zustand;
```

```
        public Element(int key, T data) {
            m_data = data;
            m_key = key;
            m_zustand = Zustand.FREI;
        }
    }

    private Element[] m_HashTable;
    private int m_n;

    public OpenHashMap(int size) {
        m_n = 0;
        m_HashTable = new OpenHashMap.Element[size];
        for (int i = 0; i < size; i++) {
            m_HashTable[i] = new Element(-1, null);
        }
    }

    abstract int h(int key);

    abstract int s(int j, int key);
}
```

Jetzt sind die grundsätzlichen Design-Arbeiten gemacht und es geht nun darum, die vom Interface `HashMap` verlangten Methoden zu programmieren. Da die beiden Methoden `get()` und `contains()` ein Suchen verlangen, implementieren wir die Suche zuerst. Beim Suchen beginnen wir mit der Hash-Adresse $i = h(k)$. Solange k nicht in `m_HashTable[i]` gespeichert ist und `m_HashTable[i]` nicht frei ist, suchen wir weiter bei $i = h(k) - s(j,k)$ mod m, für aufsteigende Werte von j. Falls `m_HashTable[i]` belegt ist, wird k gefunden; sonst ist die Suche erfolglos. Programmiert sieht das wie folgt aus:

```
private Element find(int key) {
        int j = 0;
        int i;
        int hashValue = h(key);
        do {
            i = ((hashValue - s(j, key))%getTableSize()
                + getTableSize())%getTableSize();
            j++;
        } while (m_HashTable[i].m_zustand != Zustand.FREI
                && m_HashTable[i].m_key != key);
        if (m_HashTable[i].m_zustand == Zustand.BELEGT) {
            assert m_HashTable[i].m_key == key;
            return m_HashTable[i];
        } else {
            return null;
        }
    }
```

Sie sehen, dass eine Methode `getTableSize()` verwendet wird, um nicht immer umständlich die Array-Länge auslesen zu müssen. Diese Methode wird auch von Klassen benötigt werden, die `OpenHashMap` erweitern, weshalb wir sie als `protected` definieren:

```
    protected int getTableSize() {
        return m_HashTable.length;
    }
```

Übung 17
Programmieren Sie jetzt alle Methoden, die vom Interface `HashMap` verlangt werden.

Übung 18
Ihre programmierte Klasse ist eine abstrakte Klasse, das bedeutet, dass davon keine Objekte instanziiert werden können. Leiten Sie jetzt diese abstrakte Klasse ab und programmieren Sie die nötigen Methoden `int h(int key)` sowie `s(int j, int key)` nach der Methode des linearen Sondierens. Die Klasse sollte also irgendwie so aussehen:

```
public class LineareHashMap<T> extends OpenHashMap<T> {
    public LineareHashMap(int size) {
        super(size);
    }

    @Override
    int h(int key) {
        // TODO
    }

    @Override
    int s(int j, int key) {
        // TODO
    }
}
```

Übung 19
Erweitern Sie die Klasse `OpenHashMap` erneut und verwenden Sie nun Double-Hashing als Sondierungsverfahren.

Übung 20
Testen Sie beide Implementierungen. Kommt dasselbe heraus, wie Sie als Resultat der vorherigen Übungen von Hand bekommen haben?