6. Hash Tables

6.1. Motivation

Das *Collection* Interface lässt bewusst viele Freiheiten für verschiedene Implementierungen. Diese unterscheiden sich durch unterschiedliche Eigenschaften über das für alle *Collections* Verbindliche hinaus. So erlauben *List*s Elemente gezielt zu positionieren und damit die Reihenfolge zu kontrollieren. *Queues* und *Stacks* wiederum erlauben rasches Auffinden von Elementen in Abhängigkeit der Reihenfolge, in der sie in die Datenstruktur eingefügt wurden. *Tree*s und *Priority Queues* ordnen die Elemente selbständig nach einem Ordnungskriterium.

Keine dieser Datenstrukturen kann jedoch die *contains*-Operation im Durchschnitt mit Aufwand *O*(1) ausführen. Wird keine der zuvor diskutierten Zusatzeigenschaften benötigt, sondern nur ein rasches Auffinden von Objekten anhand eines Schlüssels, sind *Hash-Tabellen* eine gute Lösung.

6.2. Lernziele

- Sie können die Grundidee und die Eigenschaften von *Hash Tables* im Vergleich mit anderen Datenstrukturen erklären und bewerten.
- Sie können *Hash-Funktionen* vorschlagen und ihre Eignung für konkrete Anforderungen beurteilen.
- Sie können Verfahren zur *Kollisionsbehandlung* vorschlagen und ihre Eignung für konkrete Anforderungen beurteilen.
- Sie können eine *Hash Table* effizient in Java implementieren.

6.3. Grundidee

Eine oft zu lösende Aufgabe ist eine Sammlung mehrerer Elemente zu unterhalten und darin einzelne Elemente anhand eines Schlüssels wiederzufinden (bzw. festzustellen, ob es überhaupt ein solches Element gibt). Beispielsweise möchte man zu Personen das Geburtsdatum speichern und anhand des Namens finden können.

Datenstrukturen wie geordnete Arrays und Bäume unterstützen das Wiederfinden, ohne dass man *alle* Elemente dafür einzeln betrachten muss. Dabei wird die *logische Anordnung* der Elemente in einer Sequenz oder einem Baum benutzt, um rasch die Menge der möglichen Speicherorte zu verkleinern. Das kann man sich vorstellen wie die Suche in einem alphabetisch geordneten Karteikasten.

Nochmal deutlich schneller ist das Wiederfinden anhand eines Schlüssels, wenn der direkt auf den Speicherplatz verweist, an dem sich das gesuchte Objekt befindet. Das ist vergleichbar mit Gegenständen, die wohldefiniert in Schubladen eingeräumt sind, und bei denen man genau weiss, welche Schublade man aufziehen muss, um einen bestimmten Gegenstand zu finden.

In Programmiersprachen entsprechen *Array*s eigentlich ganzen Schränken mit solchen Schubladen. Wenn man den Array-Index des gesuchten Objekts kennt, kann man es „mit einem Griff“ – also Aufwand *O*(1) – finden, bzw. prüfen, ob es im Array enthalten ist.

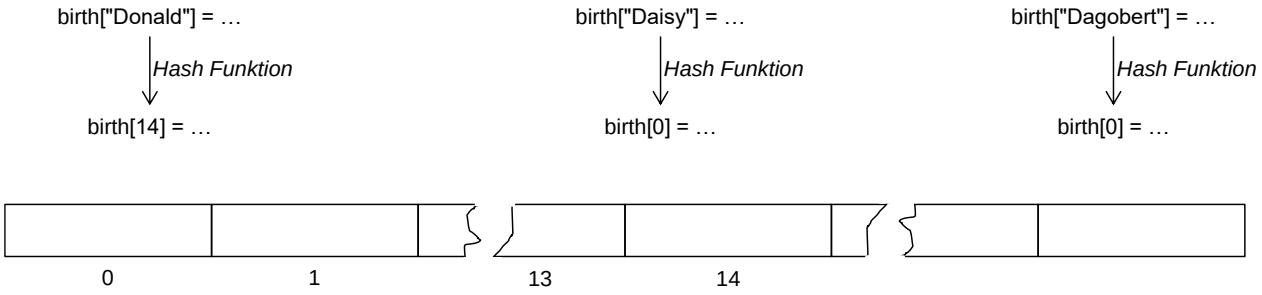
Ideal wäre also, wenn die Programmiersprache z.B. zulassen würde zu schreiben:[1]

```
birth["Donald"] = "9.Juni 1934";
if (birth["Donald"].equals(birth["Dagobert"])) ...
```

Objekte wie z.B. vom Typ *String* sind jedoch nicht geeignet, den Speicher eines Computers oder ein Array-Element direkt zu adressieren. Daher muss man das Objekt in eine Zahl transformieren, die dann als Index benutzt werden kann. Genau das leistet eine sogenannte *Hash Funktion*. Das Ergebnis des transformierten Strings wird als dessen *Hash Wert* bezeichnet. Und eine Datenstruktur, die solche Direktzugriffe umsetzt, wird *Hash Tabelle* (oder englisch: *Hash Table)* genannt.

In der Praxis besteht eine Hashfunktion sogar häufig aus zwei Schritten. Im ersten Schritt wird aus dem Objekt eine Zahl berechnet. Diese Zahl ist aber oft grösser als der höchste Index in einer Hashtabelle. In einem solchen Fall bietet es sich an, als Index den Rest eines Hash-Werts bei Division durch *m* zu verwenden, wobei *m* die Grösse der Hashtabelle bezeichnet. Für eine Zahl x ergibt sich der Index also als $x \bmod m$.

[1] Es gibt Programmiersprachen (z.B. *Perl* oder *Python*), die solche Schreibweisen ermöglichen. Tatsächlich handelt es sich dabei aber um *syntactic sugar* für Hash-Tabellen, also eine abkürzende Schreibweise für solche Datenstrukturen, um die es hier geht.



6.4. Datenstruktur *Map*

Das Interface *java.util.Map* des Java Collection Frameworks definiert eine abstrakte Datenstruktur, in der Schlüssel-Werte-Paare abgelegt werden können. Die Schlüssel können dabei beliebige Objekte sein, nicht nur Zahlen oder Namen. Java bietet damit den im Abschnitt 6.3 beschriebenen Mechanismus (Direktzugriff auf ein Objekt mit String-Schlüssel «Dagobert») an, wenn auch in der Library und nicht als Teil der Programmiersprache.

```
public interface Map<K,V> {
    V get(Object key);          // get value stored for key
    V put(K key, V value);      // store value with key as "index"
    V remove(Object key);       // remove value stored for key
    Set<K> keySet();            // get all keys currently stored in map
                                (keys must be unique, i.e., set semantics)
    Collection<V> values();     // get all values currently stored in map
                                (this may be a bag)
    ...
}
```

Das *interface Map* kann auf verschiedene Arten implementiert werden, beispielsweise mit ausgeglichenen Bäumen (*class TreeMap*) oder mit dem zuvor beschriebenen *Hashing* (*class HashMap*).

Passende Aufgabe Lösen Sie die Aufgabe 1 auf dem Arbeitsblatt zu Hash Tables.

6.5. Datenstruktur *HashMap*

Für *HashMaps* müssen die Schlüssel-Objekte zwei Bedingungen erfüllen:

1. Es muss feststellbar sein, ob zwei Objekte als *gleich* betrachtet werden sollen
2. Aus den für den Schlüssel relevanten Attributwerten muss ein *Hash-Wert* berechnet werden

Sowohl die *Gleichheit* zweier Objekte, als auch die *Hash-Werte* können nicht generell ermittelt werden, sondern hängen von Attributwerten und deren Bedeutung im Einzelfall ab. Java sieht deswegen in der Klasse *Object* zwei entsprechende Operationen vor, die dann für konkrete Objekte implementiert werden.

```
boolean equals(Object obj)      // obj is "equal to" this one
int hashCode();                 // hash value of this object
```

Damit *HashMaps* richtig funktionieren, muss für zwei Objekte *a* und *b* grundsätzlich gelten:

$$(a.equals(b)) \Rightarrow (a.hashCode() == b.hashCode())$$

In Worten ausgedrückt bedeutet dies, wenn die Objekte a und b gleich sind, dann sind auch ihre hashCodes identisch. Man beachte, dass gleiche HashCodes *nicht* Gleichheit der Objekte implizieren. Es wird also nicht ausgeschlossen, dass zwei unterschiedliche Objekte den gleichen Hash-Wert haben.

6.6. Implementierung von Hash Tabellen

Das Beispiel im Abschnitt 6.3 zeigt die grundsätzliche Funktionsweise von *Hash Tabellen*. Nach Berechnen eines Array-Index durch eine Hashfunktion können Elemente mit Aufwand *O*(1) eingefügt und gefunden werden – jedenfalls so lange es keine Kollisionen gibt. Daraus ergeben sich zwei grundsätzliche Überlegungen zu *Hash Tabellen*:

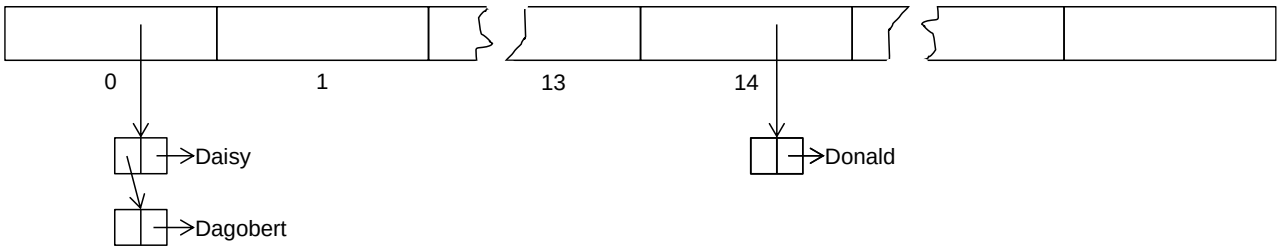
1. *Kollisionen*, (wie zum Beispiel beim Versuch, Daten mit dem Schlüssel „Dagobert“ einzufügen, nachdem die Hash Tabelle bereits Daten mit dem Schlüssel „Daisy“ enthält), können fast nie vollständig ausgeschlossen werden.[2] Oft hat man eine grosse oder gar prinzipiell unbegrenzte Menge von Schlüssel-Objekten (z.B. vom Typ *String*) und muss diese auf einen vergleichsweise kleinen Index-Bereich abbilden. Kollisionen – also gleiche Index-Werte für verschiedene Schlüssel – kann man dabei nicht systematisch ausschliessen. Man kann versuchen, Sie möglichst unwahrscheinlich zu machen, aber es braucht auch Strategien, um mit dann doch auftretenden Kollisionen umzugehen.

2. Als Anforderungen an *Hash Funktionen* ergeben sich folgende Punkte: Sie müssen
   a. schnell berechnet werden können und
   b. Werte erzeugen, die möglichst gleichmässig auf dem gesamten Index-Bereich verteilt sind und
   c. für Schlüssel, die logisch als gleich betrachtet werden (*equals*), denselben Wert erzeugen.

Im Rest dieses Kapitels werden verschiedene Verfahren zur *Kollisionsbehandlung* und dazu geeignete *Hash Funktionen* vorgestellt.

6.7. Eine einfache Strategie zum Umgang mit Kollisionen – Separate Chaining

Eine einfache Möglichkeit ist, mehrere verschiedene Objekte, die mit gleichem Array-Index abgelegt werden sollen, tatsächlich so zu speichern. Die Elemente des Arrays sind dann nicht Objekte sondern Listen von Objekten:



Diese Technik wird als *Separate Chaining* bezeichnet, weil die Elemente in (separaten) Ketten am Array hängen. Das Java Collection Framework benutzt Separate Chaining in der Klasse *HashMap*.[3] Darin sind u.a. folgende Deklarationen enthalten:

```
public class HashMap<K,V> implements Map<K,V> {
   Node<K,V>[] table;
   ...

   static class Node<K,V> implements Map.Entry<K,V> {
       final K key;
       V value;
       Node<K,V> next;
       ...
   }
}
```

[2] Die Ausnahme bildet sogenanntes *perfektes Hashing*, bei dem man für eine im Voraus genau bekannte Menge von Elementen Hash-Werte so vergeben kann, dass es keine Kollisionen gibt. Diese Randbedingungen lassen sich in der Praxis selten erfüllen. Mögliche Anwendungen sind Speicher mit unveränderlichen Daten.

[3] Seit Java 8 werden bei einer grossen Menge von Kollisionen anstelle von Listen Baumstrukturen benutzt.

6.8. Wahrscheinlichkeiten und Kollisionen

Bei rein zufälligen Verteilungen von *Hash-Werten* über den ganzen Index-Bereich ist es sehr wahrscheinlich, dass einzelne Kollisionen auftreten, aber unwahrscheinlich, dass es viele gibt. Das sollen folgende Überlegungen illustrieren.

Gehen wir davon aus, dass das Array ca. doppelt so vielen Elementen Platz bietet, als darin enthalten sind. Es ist dann sehr wahrscheinlich, dass es einzelne Kollisionen gibt.[4] Andererseits ist es sehr unwahrscheinlich, dass sehr viele Objekte am selben Ort landen und die Hash-Tabelle damit zu einer linearen Liste entartet.

Die Wahrscheinlichkeit, dass es zu keinen Kollisionen kommt, wenn man in ein Array der Länge *m* insgesamt *n* Elemente einfügt, kann man mit folgender Formel berechnen:

$$\left(\frac{m}{m}\right) \cdot \left(\frac{m-1}{m}\right) \cdot \left(\frac{m-2}{m}\right) \cdot \ldots \cdot \left(\frac{m-n+1}{m}\right) = \prod_{i=0}^{n-1} \left(\frac{m-i}{m}\right)$$

Dabei ist $(m-i)/m$ die Wahrscheinlichkeit, dass das (i+1)-te Objekt einen leeren Index erwischt, wenn bereits i Indizes besetzt sind.

Beispiel 1 Betrachten wir eine Tabelle der Grösse 13, in der 7 Elemente abgelegt werden sollen. Die Wahrscheinlichkeit, dabei mindestens eine Kollision zu verursachen, ist 1 minus die Wahrscheinlichkeit für keine Kollision:

keine Kollision: $\prod_{i=0}^{6} \left(\frac{13-i}{13}\right) = 0.138$

mindestens eine Kollision: $1 - \prod_{i=0}^{6} \left(\frac{13-i}{13}\right) = 0.862$

Aufgabe Um Zahlen in einer Hash-Tabelle der Länge *m* abzulegen, bietet es sich an, den Rest dieser Zahlen bei Division durch *m* als Index zu verwenden. Für eine Zahl x ergibt sich der Index als $x \bmod m$. Betrachten Sie 7 beliebig ausgewählte Zahlen (wählen Sie z.B. wilkürlich 4-stellige Zahlen), und füllen Sie die Zahlen nacheinander in eine Hash-Tabelle der Grösse 13.

Zahl *x*	***x* mod 13**

#Kollisionen:

Max #Kollisionen:

Durchschn. #Kollisionen:

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

[4] Ein beliebtes Beispiel hierfür ist das sogenannte *Geburtstags-Paradox*: Wenn 23 Personen in einem Raum sind – wie gross ist die Wahrscheinlichkeit, dass mindestens zwei davon am selben Tag eines Jahres Geburtstag haben? Übersetzt auf *Hash-Tabellen:* Wenn 23 Elemente in eine Tabelle mit 365 Plätzen eingefüllt werden – wie wahrscheinlich ist eine Kollision?

6.9. Berechnung von *hashCode*

Um den Hashwert zu berechnen, werden in Java die für den Schlüssel relevanten Attributwerte eines Objekts zuerst in eine Zahl umgewandelt. Die Ergebnisse dieser *hashCode*-Methode sollten möglichst zufällig über den ganzen Wertebereich des Typs *int* gestreut sein. Dies ermöglicht anschliessend eine gleichmässige Verteilung auf Index-Werte, beispielsweise bei einer modulo-Operation. Für Datentypen, deren Wertebereich ohnehin nicht grösser ist, als jener von *int*, verwendet man üblicherweise gerade den entsprechenden Wert selbst. Dies gilt für *boolean*, *byte*, *short*, *int*, *char* und *float*.

Für die 64-Bit Datentypen *long* und *double* ist eine *exclusive-oder*-Verknüpfung der beiden 32-Bit-Teile üblich, wie folgendes Beispiel aus der Klasse *Long* illustriert:

```
public int hashCode() {
     return (int)(value ^ (value >>> 32));
}
```

Um einen Hash-Wert über mehrere Variablen zu berechnen, die gemeinsam den Schlüssel-Wert ausmachen, könnte man die Hash-Werte dieser einzelnen Werte addieren. Dabei erhält man jedoch für verschiedene systematische Variationen immer dieselbe Summe.

Als Beispiel betrachten wie den Datentyp *String*. *String*-Werte lassen sich als Folge von *char*-Werten auffassen. Ein trivialer *String*-Hash-Wert wäre daher die Summe aller Zeichen-Codes.

Aufgabe Ergänzen Sie die nebenstehende Tabelle. Finden Sie auch andere Hash-Werte, wofür es viele verschiedene Wörter gibt?

Die vielen Kollisionen bei diesem Vorgehen entstehen, weil alle 26^3 Kürzel aus drei Buchstaben auf nur 78 verschiedene Hash-Werte abgebildet werden. Dies entspricht jeweils 255 oder 0 Wörtern mit gleichem Hash-Wert und ist damit keineswegs eine gleichmässige Verteilung über den gesamten Wertebereich des Typs *int*.

Wort	**Hash-Wert**
AUS	65 + 85 + 83 = 233
USA	
SAU	
OHR	
WEM	
OFT	

Tatsächlich ist in der Klasse *String* ein etwas komplizierteres Verfahren implementiert:[5]

```
public final class String {
   /** The value is used for character storage. */
   private final char value[];

   /** Cache the hash code for the string */
   private int hash; // Default to 0

   ...

   public int hashCode() {
      int h = hash;
      if (h == 0 && value.length > 0) {
         char val[] = value;
         for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
         }
         hash = h;
      }
      return h;
   }
   ...
}
```

[5] Das angegebene Programmfragment stammt aus Java 8. Seit Java 9 werden in der Klasse `String` intern Variablen vom Typ `byte[]` anstelle von `char[]` benutzt. Das Schema der Berechnung bleibt aber gleich.

Aufgabe Berechnen Sie die nun besser verteilten Hash-Werte:

Wort	Hash-Wert
AUS	65'183
USA	
SAU	

Betrachten wir obige Funktion für HashCode etwas genauer, stellen wir fest, dass die einzelnen chars im String (c_0, c_1, …, c_{l-1}) die Koeffizienten eines Polynoms p darstellen, welches mittels Horner-Schema für den Wert x=31 ausgewertet wird:

$$p(x) = x^{l-1} \cdot c_0 + x^{l-2} \cdot c_1 + \ldots\ldots + x \cdot c_{l-2} + c_{l-1} = x(x(\ldots x(c_0) + \ldots) + c_{l-2}) + c_{l-1}$$

Nach der Berechnung von HashCode für einen String gilt demnach hash = *p*(31). Generell gilt, dass die Verwendung einer Primzahl im Polynom (wie hier 31) zu einer guten Verteilung der Werte führt.

Passende Aufgaben: Lösen Sie die Aufgaben 2 bis 5 auf dem Arbeitsblatt zu Hash Tables.

6.10. Open Addressing

Separate Chaining zum Umgang mit Kollisionen ist ein einfaches Konzept (s. 6.7). Die Implementierung verursacht jedoch Aufwand z.B. in Form von zusätzlichem Speicher für die Listenverkettung. Ausserdem wird eine dynamische Speicherverwaltung vorausgesetzt, die es z.B. bei kleinen Geräten (*embedded Systems*, die z.B. mit *JavaCard* programmiert werden) nicht gibt.

Open Addressing vermeidet diesen Aufwand. Statt einer zusätzlichen Datenstruktur, werden noch freie Plätze in der Hash-Tabelle selbst benutzt. Kommt es zu einer Kollision, wird nach einer festen Regel ein neuer Index berechnet und die Suche dort fortgesetzt. Dieses Weitersuchen nennt man auch *Probing* oder *Sondieren*.

Es gibt verschiedene konkrete Umsetzungsmöglichkeiten von solchem *Open Addressing*. Zwei davon werden im Folgenden näher betrachtet: *Linear Probing* und *Double Hashing*.

6.10.1. Linear Probing [6]

Eine einfache Regel zur Suche nach einem freien Platz ist, jeweils den nächsthöheren Index zu überprüfen. Erreicht man dabei die obere Grenze des Indexbereichs, setzt man die Suche bei 0 fort. Als Java-Code könnte dies so aussehen:

```
public void add(T elem) {
   int i = (elem.hashCode() & 0x7FFFFFFF) % size;
   while (array[i] != null) {
      i = (i + 1) % size;
   }
   array[i] = elem;
}
```

Wir können dies auch als mathematische Funktion darstellen. Betrachten wir dazu den Java-Code etwas genauer. Für $x = elem.hashCode() \;\&\; 0x7FFFFFFF$ gilt, dass der Index *i* des Objekts *elem* im ersten Versuch $i = x \bmod m$ ist, wobei *m* die Grösse der Hash-Tabelle bezeichnet. Nach *k* Kollisionen gilt dann

$$i = (x + k) \bmod m.$$

Der Index ist also abhängig vom *hashCode* des Elements und von der Anzahl Kollisionen. Die Sondierungsreihenfolge *H* zur Berechnung des Index mit Linear Probing lässt sich wie folgt definieren

$$H(x, k) = (x + k) \bmod m.$$ [7]

[6] Auch: *lineares* oder *sequenzielles Sondieren*

Beispiel Fügt man Elemente mit den ersten Quadratzahlen als Hash-Wert in eine Tabelle der Länge 16 ein und zählt dabei jeweils, wie oft man Sondieren (weitersuchen) muss, ergibt sich folgendes Bild:

hashCode()	0	1	4	9	16	25	36	49	64	81
%16	0	1	4	9	0	9	4	1	0	1
Kollisionen	-	-	-	-	2	1	1	3	6	6

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	16	49	4	36	64			9	25					

Bei diesem Beispiel kann man sehr gut das sogenannte *Clustering* beobachten: Mit der Zeit bilden sich Klumpen (Cluster) aus belegten Stellen. Wird ein solcher Cluster «getroffen», muss mehrfach sondiert werden bis ein freier Platz dahinter gefunden wird. Dadurch wird der Cluster noch grösser. Je grösser aber ein Cluster, desto grösser ist die Wahrscheinlichkeit diesen zu treffen und dadurch eine lange Sondierungsfolge zu haben.

Passende Aufgabe: Lösen Sie Aufgabe 1 auf dem Arbeitsblatt 2 zu Open Adressing.

6.10.2. Double Hashing

Um die beim *Linear Probing* beobachtete Cluster-Bildung zu vermeiden, muss der Mechanismus durchbrochen werden, der die Cluster systematisch wachsen lässt. Dazu müssen die beim Sondieren gemachten Schritte vergrössert werden. Die besten Ergebnisse erreicht man, wenn man die Schrittweite für verschiedene Elemente unterschiedlich wählt, d.h. in Abhängigkeit des Hash-Codes:

```
public void add(T elem) {
     int i = (elem.hashCode() & 0x7FFFFFFF) % size;
     int step = ...?
     while (array[i] != null) {
        i = (i + step) % size;
     }
     array[i] = elem;
}
```

Die Regel zur Berechnung der Werte für `step` muss mit Sorgfalt gewählt werden. Zur Veranschaulichung, worauf es ankommt, untenstehend einige Beispiele für `step`-Werte (die Tabellengrösse sei 16).

Beispiel

step	**Sondierungsfolge bei Start mit *i* = 1**	**Beobachtung**
7	1, 8, 15, 6, 13, 4, 11, 2, 9, 0, 7, 14, 5, 12, 3, 10, 1, …	Alle Indizes werden versucht.
0	1, 1, 1, 1, 1, 1, . . .	Es wird überhaupt nicht sondiert.
16	1, 1, 1, 1, 1, 1, . . .	dito
4	1, 5, 9, 13, 1, 5, 9, 13, . . .	Es werden nicht alle Indizes versucht.
12	1, 13, 9, 5, 1, . . .	dito
9	1, 10, 3, 12, 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1	Alle Indizes werden versucht.

[7] Für F ist eigentlich jede Funktion geeignet, die Schlüssel in gleichmässig verteilte Zahlenwerte umwandelt. Für viele Objekte in Java existiert bereits die Funktion HashCode, welche diese Anforderungen erfüllt. Darum bietet sich diese Funktion für F direkt an.

Alle Werte von `step` müssen zwei Bedingungen erfüllen:

- `step` darf keine gemeinsamen Teiler haben mit der Tabellengrösse m
- `step` soll innerhalb von [1, … m-1] liegen

Es gibt zwei bekannte Strategien, um dafür zu sorgen, dass diese Bedingungen immer erfüllt sind:

a) Wähle für die Grösse der Tabelle *m* eine 2er-Potenz (2^m) und `step` ungerade in [1, …, m-1]

b) Wähle für die Grösse der Tabelle *m* eine Primzahl und `step` in [1, …, m-1]

Eine beliebte Lösung ist, als Tabellen-Grösse eine Primzahl zu wählen und die Schrittweite zum Sondieren dann zu bestimmen als:

```
int step = 1 + (elem.hashCode() & 0x7FFFFFFF) % (size - 2);
```

Der Name Double Hashing kommt also davon, dass nicht nur der erste Index ‹gehasht› wird, sondern der *HashCode* auch für den Sondierungsschritt gebraucht wird.

Bemerkung Grundsätzlich kann man in obiger Berechnung auch *% (size - 1)* anstelle von *% (size - 2)* verwenden. Der Grund, warum man das oft nicht tut, ist folgender:

Wenn *size* prim ist, ist es auch ungerade. Somit ist *size - 1* gerade. Für jede gerade Zahl *m* gilt, dass *x mod m* genau dann gerade ist, wenn *x* gerade ist. Hat man nun beispielsweise aus systematischen Gründen nur gerade Hash-Werte, so werden alle Schrittweiten ebenfalls gerade sein. In einem solchen Fall erhält man eine grössere Vielfalt an Schrittweiten, wenn *m* ungerade ist.

Auch bei Variante a) oben steht grundsätzlich nur die Hälfte der Zahlen (die ungeraden) für Schrittweiten zur Verfügung.

Beispiel Wir verwenden im Beispiel vereinfachend nur nicht-negative Hash-Werte. Obige Schrittweiten-Formel kann man dann für eine Tabellengrösse von 13 vereinfacht schreiben als:

```
int step = 1 + hashCode() % 11;
```

Fügt man damit Elemente mit den ersten Quadratzahlen als Hash-Wert in eine Tabelle der Länge 13 ein und zählt dabei jeweils, wie oft man Sondieren (weitersuchen) muss, ergibt sich folgendes Bild:

hashCode()	0	1	4	9	16	25	36	49	64	81
%13	0	1	4	9	3	12	10	10	12	3
step	*1*	*2*	*5*	*10*	*6*	*4*	*4*	*6*	*10*	*5*
Kollisionen	0	0	0	0	0	0	0	3	2	1

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	**1**	**49**	**16**	**4**		**64**		**81**	**8**	**36**		**25**

Herleitung mathematische Funktion:

Für den Index *i* des Objekts *elem* gilt im ersten Versuch wieder $i = x \bmod m$, wobei $x = elem.hashCode()$ & $0x7FFFFFFF$ und *m* die Grösse der Hash-Tabelle bezeichnet. Nach *k* Kollisionen gilt dann

$$i = \left(x + k \cdot \left(1 + x \bmod (m-2)\right)\right) \bmod m.$$

Mit $F(x) = 1 + x \bmod (m-2)$ lässt sich die Hashfunktion *H* zur Berechnung des Index mit Double Hashing wie folgt definieren:

$$H(x,k) = \left(x + k \cdot F(x)\right) \bmod m$$

Auch hier wird deutlich, dass die Sondierungsreihenfolge im Prinzip durch zwei ineinander verschachtelte Hashfunktionen berechnet wird.

Passende Aufgabe: Lösen Sie die Aufgabe 2 auf dem Arbeitsblatt 2 zu Open Addressing.

6.10.3. Schnellere Implementierung

Sowohl beim *Linear Probing* als auch beim *Double Hashing* verursacht die *Rest-* oder *Modulo-Operation* (*%*) in der Sondierungs-Schleife den grössten Laufzeitaufwand, denn diese benötigt eine Division und dafür werden im Vergleich zu den sonstigen sehr einfachen Operationen viele Einzelschritte in der Hardware benötigt. Es lohnt sich also, nach Alternativen für die Anweisung `i = (i + step) % size;` zu suchen:

- Wenn `size` = 2^k für ein beliebiges k – also `size` eine Zweierpotenz ist –, lässt sich die Modulo-Operation durch eine Bit-Maskierung in einem Zyklus berechnen:
  ```
  i = (i + step) & (size - 1);
  ```
- Anstelle einer Modulo-Rechnung kann man den Überlauf auch durch einen Vergleich erkennen:
  ```
  i = i + step; if (i >= size) i -= size;
  ```
- Im Prinzip ist ein Vergleich von `i` mit `0` schneller als einer mit einem beliebigen Zahlenwert `size`, denn letzterer wird eigentlich mit einer zusätzlichen Operation berechnet: `i-size >= 0`. Diese Überlegung legt als weitere Variante nahe, *rückwärts zu sondieren* und statt eines Überlaufs einen Unterlauf zu erkennen:
  ```
  i = i - step; if (i < 0) i += size;
  ```

Die ersten beiden Implementierungs-Varianten berechnen dieselben Indexwerte, wie die ursprüngliche Variante mit `%`. Die damit erzeugten Hash-Tabellen sehen ebenfalls gleich aus. Da die letzte Variante jedoch in der anderen Richtung sucht, werden die Elemente bei Kollisionen an anderen Plätzen abgelegt.

6.10.4. Load Factor [8]

Es ist zu erwarten und auch in den bisher betrachteten Beispielen erkennbar, dass Kollisionen immer öfter auftreten, je voller eine Tabelle ist. Um in diesem Zusammenhang quantifizierbare Aussagen machen zu können, verwendet man den *Load Factor* λ als Messgrösse:

$$\lambda := \frac{AnzahlElemente \in derTabelle}{Tabellengrösse}$$

Beispiel

Für die beiden Beispiel-Tabellen in 6.10.1 und 6.10.2 gilt

$\lambda_{6.10.1} = 10/16 = 0.625$ bzw. $\lambda_{6.10.2} = 10/13 = 0.769$

Für *Separate Chaining* gilt:

- Es gibt keine grundsätzliche Obergrenze für λ, weil die Überlauflisten beliebig lang werden können.
- Die durchschnittliche Länge der Listen ist gleich dem *Load Factor* λ.
- Um die Effizienz zu erhalten, empfiehlt sich: $\lambda < 1$.

Für *Open Addressing* gilt:

- Der Load Factor ist hier systembedingt begrenzt: $\lambda \leq 1$.
- Nur solange $\lambda < 1$ gilt, wird man bei jeder Sondier-Schleife irgendwann auf (allenfalls den) einen freien Platz treffen. Lässt man auch den Fall $\lambda = 1$ zu, muss man die Schleifenabbruchkriterien so formulieren, dass auch abgebrochen wird, wenn man alle Plätze einmal besucht hat.
- Um die Effizienz zu erhalten, empfiehlt sich bei *Linear Probing* $\lambda < 0.75$ und bei *Double Hashing* $\lambda < 0.9$.

Passende Aufgabe: Lösen Sie die Aufgabe 3 auf dem Arbeitsblatt 2 zu Open Addressing.

[8] Auch *Belegungsfaktor* oder *Belegungsgrad*

6.10.5. Rehashing

Kennt man die Anzahl Elemente, die eingefügt werden sollen, von vornherein, kann man unter Berücksichtigung des angestrebten *Load Factors* ein entsprechend grosses Array erzeugen.

Kennt man die Anzahl der Elemente nicht, empfiehlt es sich, ein neues grösseres Array zu erzeugen und die Elemente dort hinein zu kopieren, sobald der *Load Factor* eine gegebene Grenze überschreitet. Das Verfahren ist von halbdynamischen Array-Listen her bekannt. Jedoch kann man bei *Hash Tables* nicht mehr einfach die Positionen im Array beibehalten, sondern muss die Indices mit dem Hash-Mechanismus neu berechnen. Diesen Prozess nennt man deswegen auch *Rehashing*.

6.10.6. Entfernen von Elementen

Entfernt man aus der im Kapitel 6.10.2 gefüllten Tabelle am Schluss das Element *9* wieder, erhält man diese Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	*49*	16	4		*64*		81		36		25

Sucht man in dieser Tabelle (mit der gleichen Sondierungs-Strategie wie in Kapitel 6.10.2) nach *49* oder *64* erhält man das Ergebnis, dass die Elemente nicht in der Tabelle sind, weil ein leeres Feld gefunden wird, bevor man auf das Element trifft.

Man darf also in solchen Tabellen keine Werte einfach entfernen. Stattdessen setzt man in solchen „frei gewordenen“ Plätzen eine spezielle Markierung (*Sentinel*) ein. Trifft man beim Sondieren nach einem Element (z.B. bei *contains(x)*) auf das Sentinel, dann wird die Suche weitergeführt.

Soll ein neues Element eingefügt werden und man trifft beim Sondieren nach einem freien Platz auf das Sentinel, dann wird das Element dort eingefügt.

Implementierungstechnik für Sentinels

Benötigt man in einer objektorientierten Programmiersprache wie Java ein *Sentinel*, gibt es eine spezielle Implementierungstechnik: Man erzeugt ein fixes Objekt und setzt dieses (eigentlich eine Referenz darauf) als *Sentinel* ein. Dieses Objekt ist nie gleich irgendeinem tatsächlich in der Datenstruktur enthaltenen Objekt, aber auch nicht *null*. Per Referenzvergleich *o == sentinel* kann es klar identifiziert werden.

```
public class Table<T> {
    private final Object[] arr;
    private final Object sentinel = new Object();

    public Table(int size) {
        this.arr = new Object[size];
    }
    public void add(T elem) {
        ...
    }
    public boolean contains(T elem) {
        ...
    }
    public void remove(T elem) {
        assert elem != null;
        int i = (elem.hashCode()& 0x7FFFFFFF) % arr.length;
        int cnt = 0;
        while (arr[i] != null && !elem.equals(arr[i]) && cnt != arr.length) {
            i = (i + 1) % arr.length; // we use linear probing for simplicity
            cnt++;
        }
        if (elem.equals(arr[i])) arr[i] = sentinel;
    }
}
```

Passende Aufgabe: Lösen Sie die Aufgabe 4 auf dem Arbeitsblatt2 zu Open Addressing.