Hash Tables

1. Einleitung

Wichtige Operationen auf Datensammlungen – also Datenstrukturen wie *Mengen* – sind

- *Einfügen* eines Elements,
- *Prüfen* ob ein Element enthalten ist,
- *Entfernen* eines Elements.

Um eine solche Datenstruktur zu realisieren, gibt es verschiedene Möglichkeiten basierend auf *Listen*. Aufwändig sind die *Such-Operationen*, also jene, bei denen es darum geht, zu prüfen, ob ein bestimmtes Element in der Menge enthalten ist. Wird ein Element gesucht, das nicht in der Menge ist, müssen entweder alle Elemente betrachtet werden (*Sequenzielle Suche* mit linearem Aufwand *O*(n)) oder die Elemente in der Liste müssen speziell angeordnet sein, um mit einzelnen Vergleichen ganze Gruppen von Elementen ausschliessen zu können (*Binäre Suche* mit logarithmischem Aufwand *O*(log n)). Das zweite verursacht jedoch zusätzlichen Aufwand (*O*(n)) beim *Einfügen* und *Entfernen* von Elementen, um dabei die Ordnung aufrecht zu erhalten.

Am schnellsten sind die beschriebenen Mengenoperationen, wenn jedes Element einen eigenen fest definierten Platz hat. Dann kann es genau dort gespeichert werden, und um zu prüfen, ob ein bestimmtes Element in der Menge enthalten ist, braucht nur auf diesen einen Platz zugegriffen zu werden. Beides sind Operationen mit *konstantem Aufwand O*(1).

Ist die Grundmenge sehr gross und enthält eine zu repräsentierende Menge nur einige wenige Elemente daraus, bleiben sehr viele der bereitzustellenden Speicherplätze leer. Der Speicherbedarf einer solchen Datenstruktur ist viel zu gross.

Soll beispielsweise zu einem Text die Menge aller darin enthaltenen Wörter ermittelt werden (z.B. um zu zählen, wie viele verschiedene Wörter der Text enthält), müsste ein riesiges Speichervolumen bereitgestellt werden, um den ganzen Wortschatz allein einer Sprache abzudecken. Das ist offensichtlich so noch nicht praktikabel.

2. Die Funktion hash()

Die Idee hinter *Hash Tables* ist, allen darin abzulegenden Elementen als Identifikation eine ganze Zahl zuzuweisen – einen *hash*-Wert. Dieser *hash*-Wert muss einen Bezug zum Wert des zu speichernden Elements haben, denn man möchte ja als *äquivalent* geltende Objekte in der Datenstruktur wiederfinden können. In *Python* existiert dafür die Funktion *hash*:[1]

```
>>> x, y, z = 3, 'abc', (3, 'abc')
>>> hash(x)
3
>>> hash(y)
5577162849320017501
>>> hash(z)
6551016547980811634
>>> hash(z) == hash((3, y))
True
>>> z == (3, y)
True
```

Es gilt für beliebige x und y: x == y ⇒ hash(x) == hash(y).

Abgesehen davon ist es das Ziel der *hash*-Funktion, möglichst gleichmässig verteilte möglichst verschiedene Werte zu liefern. Kleine Unterschiede können zu völlig anderen *hash*-Werten führen:

```
>>> hash('abcde')
-5776473217066292285
>>> hash('abcdf')
7872682686140329403
```

Die Werte der Funktion *hash* kann man idealisiert wie eine Zufallszahl betrachten, die aber abhängig von den Daten selbst reproduzierbar ist. Den Wert einer solchen *hash*-Funktion kann man benutzen, um einen Index in eine Speicher-Struktur zu berechnen: Stehen *n* Plätze zur Verfügung, würde ein Element `e` an der Stelle `hash(e) % n` gespeichert. Eine so genutzte Datenstruktur bezeichnet man als *Hash Table*.

[1] Die Zahlenwerte der Ergebnisse variieren bei jedem Neustart des Python-Interpreters, um zu vermeiden, dass sich Programmierer auf bestimmte Werte verlassen. Deswegen werden Ihr andere *hash*-Werte erhalten, wenn Ihr selbst experimentiert. Eine Ausnahme hiervon bilden die *hash*-Werte für ganze Zahlen.

Das Verfahren kann nicht verhindern, dass zwei verschiedene Elemente denselben Index zugeteilt erhalten. Eine solche Situation bezeichnet man als *Kollision*.

Der entscheidende Hebel von *Hash Tables* liegt darin, die Wahrscheinlichkeit von Kollisionen möglichst klein zu halten. Dazu trägt unter anderem eine möglichst breit streuende *hash-Funktion* bei. Ausserdem lässt sich die Wahrscheinlichkeit durch das Verhältnis der Anzahl an tatsächlich eingefügten Elementen und der Tabellengrösse *n* beeinflussen. Je grösser die Tabelle bei gleicher Elemente-Zahl, desto geringer die Wahrscheinlichkeit von Kollisionen.

Da sich Kollisionen nicht generell vermeiden lassen, braucht es in jedem Fall eine möglichst effiziente Strategie damit umzugehen. Hier gibt es verschiedene Wege.

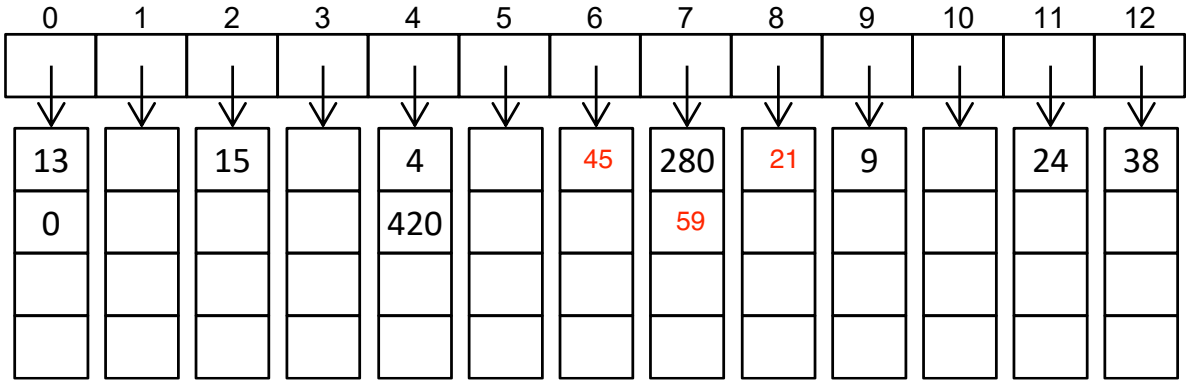
3. Separate Chaining

Eine einfache Kollisionsbehandlung besteht darin, an den einzelnen Speicherplätzen nicht einzelne Elemente, sondern *Listen von Elementen* abzulegen. So können einfach alle Elemente, denen derselbe Index zugeteilt wird, gemeinsam an dem entsprechenden Platz gespeichert und später wiedergefunden werden.

Enthalten diese Listen jeweils nur wenige Elemente, kann man sich eine einfache sequenzielle Suche leisten. Verteilt auf eine sehr grosse Anzahl von Zugriffen auf die *Hash Table* fällt der Aufwand dafür dann nicht mehr ins Gewicht.

Betrachten wir als Beispiel eine solche *Hash Table* der Grösse 13. Darin wurden bereits die Zahlen 15, 4, 38, 13, 24, 280, 9, 0 und 420 (in dieser Reihenfolge) eingefügt. Wir nehmen hier an, dass die *hash*-Werte gerade den jeweiligen Zahlenwerten entsprechen. In *Python* ist das ja auch tatsächlich so.

Für die Zahl 15 ergibt sich aus `hash(15) % 13` der Index 2. Deswegen wurde 15 als erstes Element in die Liste eingefügt, die in der Tabelle die Position 2 belegt.



Aufgabe 1: Wo wären in dieser bereits teilweise gefüllten Tabelle jetzt die Werte 45, 59 und 21 zu platzieren? Tragt diese Werte entsprechend ein.

Aufgabe 2: Nachdem die drei Werte gemäss Aufgabe 1 zu den bereits vorhandenen 9 Werten hinzugefügt wurden, enthält die Tabelle 12 Elemente. Wenn nun für jeden dieser 12 Werte einmal die Frage gestellt wird, ob er in der Tabelle enthalten ist, muss man in der richtigen indizierten Liste von oben nach unten die Elemente vergleichen, bis das Ende erreicht ist oder der gesuchte Wert gefunden wird. Wie viele Element-Vergleiche braucht es dann in diesem konkreten Beispiel hier? 9 Elemente an erster stelle + 3 *2 = 15

Wie viele braucht es, wenn dieselben 12 Werte unsortiert in einer einfachen Liste gespeichert sind und entsprechend für jeden Wert einmal sequenzielle Suche ausgeführt wird. 12*12/2 = 72

Aufgabe 3: In eine *Hash Table* der Grösse *m* werden nacheinander *n* Elemente eingefügt, wobei $n < m$. Mit welcher Wahrscheinlichkeit kommt es dabei zu *keiner einzigen Kollision*?

Gebt eine Formel dafür an.

Bemerkung: Setzt man versuchshalber verschiedene Zahlen in diese Formel ein, stellt man fest, dass die Wahrscheinlichkeit für Kollisionsfreiheit schon bei wenigen Elementen gering ist.[2] Andererseits ist aber die Wahrscheinlichkeit, dass es viele Kollisionen gibt, gering solange die Anzahl der gespeicherten Elemente kleiner ist als die Grösse der Tabelle und die Hash-Werte statistisch gleichverteilt sind.

[2] Ein bekanntes Beispiel zur Veranschaulichung dieses Phänomens ist das sogenannte *Geburtstagsparadox*. Das besagt, dass bereits von 23 Personen mit mehr als 50% zwei am selben Tag im Jahr Geburtstag feiern.

Aufgabe 4: In *Python* lässt sich eine *Hash Table* mit *Separate Chaining* leicht mit einem Tupel von Listen realisieren. Ein Tupel eignet sich, weil Elemente nur in die Listen eingefügt werden, die Listen selbst aber für die gesamte Lebensdauer der Datenstruktur dieselben bleiben: [3]

```
def new_hash_set(table_size):
  return tuple([] for _ in range(table_size))
```

Programmiert folgende Funktionen, die eine durch `new_hash_set` erzeugte *Hash Table* bearbeiten:

- `add_to_hash_set(table, elem)` – fügt `elem` in `table` ein, falls es nicht schon enthalten ist
- `remove_from_hash_set(table, elem)` – entfernt `elem` aus `table`
- `is_in_hash_set(table, elem)` – `True`, falls `elem` in `table` enthalten ist, sonst `False`
- `num_elems(table)` – Anzahl der aktuell in `table` enthaltenen Elemente
- `get_statistics(table)` – berechnet 2 Werte über die aktuelle Verteilung der Elemente in `table`

Die Funktion `get_statistics(table)` soll zwei Werte berechnen:

1. Die Anzahl Element-Vergleiche, die nötig wären, wenn jedes der in `table` enthaltenen Elemente einmal (mit `is_in_hash_set`) gesucht würde (vgl. Aufgabe 2).
2. Die maximale Anzahl Element-Vergleiche bei der Suche nach einem beliebigen Element (entspricht der Länge der längsten Liste in `table`).

Testet und untersucht das Laufzeitverhalten Eurer Implementierung mit verschiedenen Experimenten. Zur Prüfung von `get_statistics` könnt Ihr das Experiment aus Aufgabe 2 programmieren.

Aufgabe 5: Testet Euer bei Aufgabe 4 entstandenes Programm indem Ihr eine Tabelle mit vielen zufälligen Fliesskommazahlen füllt, wie sie `random.random()` generiert. Füllt 300'000 solche Werte in eine Tabelle mit 400'000 Plätzen.

Was fällt an den Werten von `get_statistics` auf?

Wie ändern sich diese, wenn Ihr anstelle von 400'000 die Tabellengrösse 400'009 verwendet?

Die verwendeten Zufallszahlen sind Fliesskommazahlen zwischen 0 und 1. Schaut Euch ein paar *Hash*-Werte solcher Zahlen an. Vielleicht hilft es, diese *Hash*-Werte in binärer Darstellung ausgeben zu lassen (Funktion `bin()` in *Python*). Was fällt Euch auf? Könnt Ihr damit die beobachteten Unterschiede erklären?

4. Open Adressing

Separate Chaining ist ein einfach zu programmierender und wirkungsvoller Weg mit Kollisionen umzugehen. Deswegen wird diese Idee auch oft verwendet.

Die Einfachheit erkauft man sich mit zusätzlichem Speicheraufwand für die Listen-Strukturen. Ausserdem braucht jede Suche nach einem Element zwei Schritte bzw. Zugriffe auf zwei Speicher-Objekte: Man findet zunächst die Liste und muss dann darin nach dem gesuchten Element schauen. Diese beiden Zugriffe können kaum optimierende Hardware wie Caches und Pipelines nutzen.

Programmiert man auf eingeschränkter Hardware (wie *embedded devices*), hat man bisweilen nur wenig Speicher und womöglich keine dynamische Speicherverwaltung zur Verfügung. Dann eignet sich *Separate Chaining* nicht gut; und wenn es auf maximale Geschwindigkeit ankommt, möchte man den zweiten Speicher-Zugriff lieber vermeiden.

In solchen Fällen ist es besser, alle Elemente direkt in der *Hash Table* zu speichern und bei Kollisionen freie Plätze an anderer Stelle der *Hash Table* zu benutzen. Dazu wird dann nach einer festen Regel ein neuer Index berechnet und die Suche dort fortgesetzt. Solche Kollisionsbehandlungs-Strategien werden als *Open Addressing* bezeichnet.

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

[3] Nebenbei: Warum funktioniert folgende Initialisierung des Tupels nicht, obwohl das Resultat auf den ersten Blick gleich aussieht? `([],)*n`

4.1. Linear Probing [4]

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: [5]

```
def add_to_hash_set(table, elem):
   h = hash(elem) % len(table)
   while table[h] is not None and table[h] != elem:
     h = (h + 1) % len(table)
   if table[h] is None:
     table[h] = elem
```

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*								
Kollisionen	*0*	*0*								

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	*1*														

Aufgabe 6: Vervollständigt die Tabellen oben, indem Ihr zu den angegebenen Quadratzahlen jeweils den Index (Zahlenwert modulo 16) berechnet, an dem zuerst sondiert wird. Tragt dann in der *Hash-Tabelle* darunter die Quadratzahl in dem Feld ein, das ihr gemäss dem angegebenen Programm zum *linearen Sondieren* zugewiesen wird. Zählt dabei die *Kollisionen*, also wie oft ein bereits besetztes Feld gewählt würde und weitergesucht werden muss, und tragt das in der untersten Zeile der oberen Tabelle ein.

Bei diesem Beispiel kann man sehr gut das sogenannte *Clustering* beobachten: Mit der Zeit bilden sich «Klumpen» (*Cluster)* von belegten Feldern. Wird beim Einfügen eines Elements ein solcher *Cluster* getroffen, muss mehrfach sondiert werden, bis ein freier Platz dahinter gefunden wird. Dort wird dann das neue Element platziert, wodurch der *Cluster* noch grösser wird. Damit steigt dann auch gerade wieder die Wahrscheinlichkeit für weitere «Treffer» in den *Cluster*.

4.2. Double Hashing

Um die beim *Linear Probing* beobachtete *Cluster*-Bildung zu vermeiden, muss der Mechanismus durchbrochen werden, der die *Cluster* systematisch wachsen lässt. Die besten Ergebnisse erreicht man, wenn man die Schrittweite für verschiedene Elemente unterschiedlich wählt, d.h. in Abhängigkeit des Hash-Wertes: [5]

```
def add_to_hash_set(table, elem):
   h = hash(elem) % len(table)
   step = calc_step(elem)         // function calc_step still needs to be defined
   while table[h] is not None and table[h] != elem:
     h = (h + step) % len(table)
   if table[h] is None:
     table[h] = elem
```

[4] Auch bezeichnet als *lineares* oder *sequenzielles Sondieren*

[5] Hier wurde der Übersichtlichkeit halber ein Programm gewählt, das einen freien Platz in der *Hash Table* voraussetzt. Wird versucht, ein weiteres Element in eine bereits volle Tabelle einzufügen, terminiert dieses Programm nicht. Es kann aber leicht so erweitert werden, dass es dann z.B. mit einer *Exception* einen Fehler anzeigt.

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

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 Plätze werden «besucht»*
0	*1,*	
16	*1,*	
4	*1,*	
12	*1,*	
9	*1,*	

Aufgabe 7: Vervollständigt obige Tabelle.

Wie die Beispiele zeigen, müssen alle Werte von `step` zwei Bedingungen erfüllen:

- `step` hat keinen gemeinsamen Teiler mit der Tabellengrösse
- `step` ∈ [1, … , Tabellengrösse – 1]

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

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

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

Eine beliebte Lösung ist, als Tabellen-Grösse eine Primzahl zu wählen und die Schrittweite zum Sondieren dann zu bestimmen als: [6]

```
step = 1 + hash(elem) % (len(table) - 2)
```

Aufgabe 8: Probiert dieses Verfahren mit den Quadratzahlen als Elementen aus, die in eine Tabelle der Grösse 13 eingefügt werden (vgl. Aufgabe 6). Ergänzt die Berechnungsformel für `step` dafür passend:

```
step = 1 + hash(elem) %
```

hashCode()	0	1	4	9	16	25	36	49	64	81
%13	*0*	*1*								
step	*1*	*2*								
Kollisionen	*0*	*0*								

Hash-Tabelle:

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

[6] Anstelle von «`% (len(table) - 2)`» könnte man auch «`% (len(table) - 1)`» verwenden. Der Grund, warum man das oft nicht tut, ist folgender:

Wenn `len(table)` prim ist, ist es auch ungerade. Somit ist `len(table)-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.

4.3. Load Factor [7]

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:

$$\lambda = \frac{Anzahl\ Elemente\ in\ der\ Tabelle}{Tabellengrösse}$$

Beispiel:

Für die beiden Beispiel-Tabellen in 4.1**Error! Reference source not found.** und 4.2 gilt

$\lambda_{4.1}$ = bzw. $\lambda_{4.2}$ =

Für *Separate Chaining* gilt:

- Es gibt keine grundsätzliche Obergrenze für λ.
- 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* $\lambda < 0.9$.

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. Allerdings muss man dabei die Positionen der Elemente neu berechnen, denn die Berechnungsformel ist ja von der Länge der Tabelle abhängig. Diesen Prozess nennt man *Rehashing*.

4.4. Entfernen von Elementen

Entfernt man aus der im Kapitel 4.2 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 4.2) nach *49* oder *64* erhält man das Ergebnis «Element nicht enthalten», weil die Suche beim zwischenzeitlich frei gewordenen Platz der 9 abgebrochen wird, denn der ist ja jetzt leer.

Man darf also in solchen *Hash Tables* mit *Open Addressing* keine Werte einfach entfernen. Stattdessen setzt man in solchen «frei gewordenen» Plätzen eine spezielle Markierung (*Sentinel*) ein. Trifft man beim Sondieren nach einem Element auf das Sentinel, dann setzt man die Suche einfach fort.

Soll ein neues Element eingefügt werden und man trifft beim Sondieren nach einem freien Platz auf das Sentinel, dann kann man diesen Platz wieder benutzen.

Als *Sentinels* eignen sich Objekte, von denen man sicher weiss, dass sie nicht als Werte verwendet werden. Eine sichere Möglichkeit – beispielsweise für eine Implementierung in einer Programmbibliothek – ist, ein Objekt einer spezielle Klasse zu instanziieren, das dann zu keinem anderen Objekt äquivalent ist:

```
class Sentinel: pass
sentinel = Sentinel()
```

[7] Auch *Belegungsfaktor* oder *Belegungsgrad*