5 Hashing

Programme manipulieren Mengen von Objekten. Die Mengen werden in der Regel in den meisten Anwendungen laufend vergrößert, verkleinert oder nach bestimmten Elementen durchsucht. Jedes Element x einer Menge K besitzt einen Schlüssel k.

Typische Operationen auf diesen dynamischen Mengen sind

Search(K,k)	Suche ein Element x von K mit Schlüssel k.
Delete(K,x)	Entferne das Element x aus K.
Minimum(K)	Suche das (ein) Element von K mit minimalem Schlüssel.
Maximum(K)	Suche das (ein) Element von K mit maximalem Schlüssel.
Successor(K,x)	Suche das Element, dessen Schlüssel in der Ordnung von K auf den Schlüssel von x folgt (nächst größere Schlüssel).
Predessor(K,x)	Suche das Element, dessen Schlüssel in der Ordnung von K dem Schlüssel von x vorangeht (nächst kleinere Schlüssel).

Definition 5.1. Eine dynamische Menge, die diese Operationen unterstützt, bezeichnet man als **Wörterbuch** (**dictionary**).

Für typische Datenstrukturen erhalten wir folgendes Bild

Bemerkung 5.1. Keine der angegebenen Datenstrukturen ermöglicht gleichzeitig effizientes Suchen, Einfügen und Löschen!

Definition 5.2. Eine **Hash-Tabelle** ist eine Datenstruktur, die nur die Operationen Suchen, Einfügen und Entfernen “effizient“ unterstützt.

1 Sei U das **Universum** aller Schlüssel

2 Sei $K \supseteq U$ die Menge der zu speichernden Schlüssel.

3 Sei T ein Feld der Größe m.

4 Sei h eine Abbildung von U nach $\{0, 1, ..., m-1\}$.

h bezeichnet man als **Hash-Funktion**. $h(k_i)$ ist der **Hash-Wert** von k_i. Das Element k_i wird im Feldelement mit Index $h(k_i)$ von Feld T gespeichert.

	sortiertes Feld	nicht sortiertes Feld	sortierte doppelt-verkettete Liste	nicht sortierte doppelt verkettete Liste	(nicht sortierter) Stack/Queue
Search	$O(log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Delete	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Successor	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Predecessor	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Minimum	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Maximum	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Tabelle 5.1: Laufzeiten von Operationen bei verschiedenen Datenstrukturen

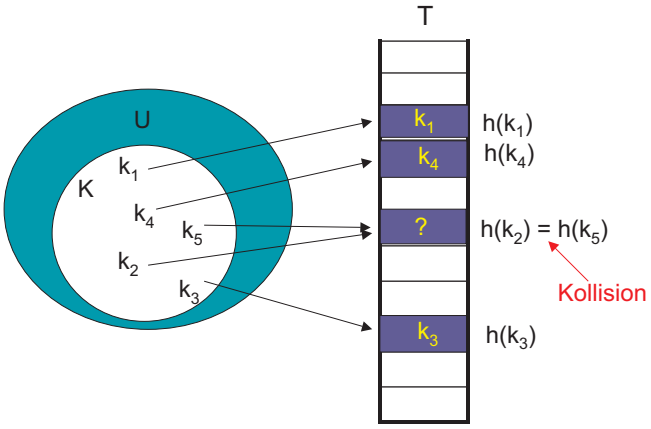

Abb. 5.1: Hashing mit Chaining-Techniken

5.1 Hashing Techniken

Hashing macht dann Sinn, wenn die Zahl $|K|$ der zu speichernden Elemente sehr viel kleiner als die Größe $|U|$ des Universums ist. Die Größe m des Feldes T sollte von der Größenordnung $O(|K|)$ sein. Wünschenswert sind Hash-Funktionen, die unter diesen Bedingungen wenig Kollisionen verursachen.

5.1.1 Hashing mit Chaining-Techniken

Kollisionen kann man unter anderem mit **Chaining-Techniken** auflösen. Hierbei arbeitet man mit einfach verketteten Listen. T enthält dann Zeiger auf Listen bzw. Zeiger auf NULL, falls

die entsprechenden Listen leer (weiße Felder) sind.

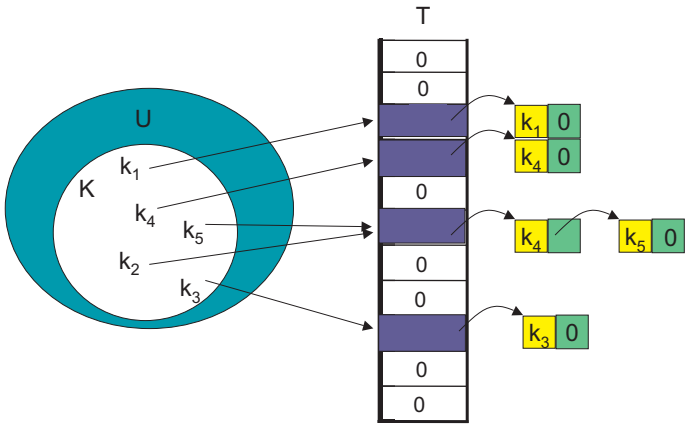

Abb. 5.2: Hashing mit Chaining-Techniken

Wir definieren eine Klasse von "ChainedHashTable" und betrachten die folgenden Methoden.

void insert(int k_i)	Füge k_i am Kopf der Liste ein, auf die $T[h(k_i)]$ zeigt
void search(int k_i)	Suche nach k_i in der Liste, auf die $T[h(k_i)]$ zeigt.
void delete(int k_i)	Lösche k_i aus der Liste, auf die $T[h(k_i)]$ zeigt.

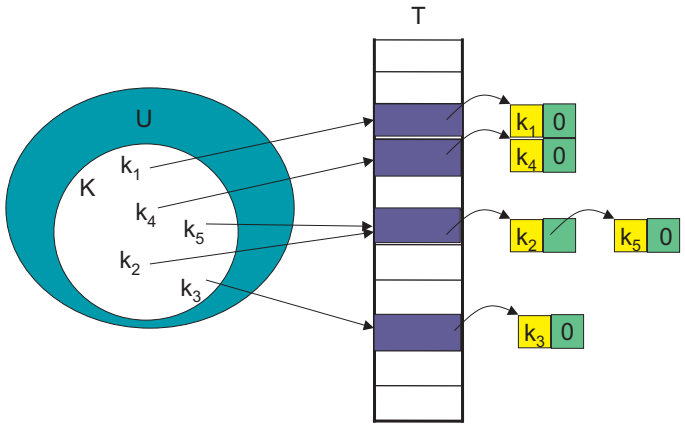

Abb. 5.3: Hashing mit Chaining-Techniken

Die worst-case Laufzeit von "insert" ist $O(1)$. Die worst-case Laufzeiten von "search" und "delete" sind proportional zu den Längen der entsprechenden Listen.

Effizients von Hashing mit Chaining

Gegeben eine Hash-Tabelle mit m Slots (also ein Feld T der Größe m), die $n = |K|$ Elemente speichert. Falls alle Elemente gleichmäßig über die Slots verteilt sind, dann ist jede Liste

$\alpha = \frac{n}{m}$ ($\alpha =$ **load factor**) lang. Im Worst case, d.h. alle n Elemente in die gleiche Liste, erhalten wir also $O(n)$. In einem nächsten Schritt wollen wir jetzt die Average Performance betrachten. Hierzu wenden wir uns der folgenden Fragestellung zu.

Wie gut verteilt die Hash-Funktion die Elemente über die Slots?

Im Weiteren sei immer eine Hash-Tabelle mit m Slots, die $n = |K|$ Elemente speichert, gegeben.

Einfaches uniformes Hashing Wir nehmen an, dass jedes vorgegebene Element mit der gleichen Wahrscheinlichkeit in einen der m Slots von der Hash-Funktion abgebildet wird. Für $j = 0, 1, ..., m-1$ bezeichnen wir mit l_j die Länge der Liste $T[j]$, d.h.

$$n = l_0 + l_1 + ... + l_{m-1}.$$

Der Durchschnittswert von l_j (erwartete Länge der Liste) ist

$$E[l_j] = \sum_{i=1}^{n} p(h(k_i) = j) \cdot 1 = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Wenn wir in der Hash-Tabelle nach k_i suchen, so müssen wir

(1) $h(k_i)$ berechnen: Zeit $O(1)$

(2) a) Falls k_i nicht in der Liste $T[h(k_i)]$ gespeichert ist, so müssen wir die gesamte Liste durchsuchen
⇒ erwartete Zeit $O(\alpha)$

b) Falls k_i in der Liste $T[h(k_i)]$ gespeichert ist, so müssen wir die Liste bis zum Schlüssel k_i durchsuchen
⇒ erwartete Zeit **?**

Wir nehmen an, dass jedes der n vorgegebenen Elemente mit gleicher Wahrscheinlichkeit das gesuchte Element k_i sein kann. Die Zahl der Elemente, die während einer erfolgreichen Suche nach einem Element k_i getestet werden, ist um eins größer als die Zahl der Elemente vor k_i in der Liste. Die Elemente vor k_i wurden alle später eingefügt, da man immer am Listenanfang einfügt. Um die erwartete Anzahl von untersuchten Elementen zu bestimmen, berechnen wir den Durchschnitt aller n gespeicherten Elemente. Sei k_i das i-te Element, das in die Hash-Tabelle eingefügt wurde. Wir definieren nun die folgenden Zufallsvariablen

$$X_{ij} = \begin{cases} 1, \text{ falls } h(k_i) = h(k_j) \\ 0, \text{ sonst} \end{cases},$$

es gilt nun

$$E[X_{ij}] = \frac{1}{m},$$

da $p(h(k_i) = h(k_j)) = \frac{1}{m}$. Wir erhalten somit für den Erwartungswert

$$\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})\right] &= \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) \\
&= \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}.
\end{aligned}$$

Satz 5.1. *In einer Hash-Tabelle, in der Kollisionen mit Chaining aufgelöst werden, benötigt eine Suche (unter der Annahme von einfachem, uniformem Hashing) im Durchschnitt eine erwartete Zeit von* $\Theta(1+\alpha)$.

Eine gute Hash-Funktion sollte die Annahme über einfaches uniformes Hashing erfüllen, d.h. jedes Element sollte ungefähr mit der gleichen Wahrscheinlichkeit in irgendeinen der m Slots abgebildet werden. Leider ist es in der Regel nicht möglich diese Bedingung zu überprüfen, da man selten die Wahrscheinlichkeitsverteilung der gezogenen Elemente (Schlüssel) kennt, da diese Schlüssel gewöhnlich nicht unabhängig voneinander gezogen werden.

5.1.2 Hashing mit der Divisions-Methode

Hierfür betrachten wir die folgende Hashfunktion

$$h(k) = k \mod m,$$

was uns zu dem nächsten Beispiel führt.

Beispiel 5.1. Sei $m = 12$ und $k = 100$ gegeben. Dann erhaten wir als Hashwert

$$\Rightarrow h(k) = 4.$$

Bemerkung 5.2.

1. Im Allgemeinen ist es sinnvoll $m = 2^p$ als Wahl zu vermeiden.
2. Eine Primzahl, die nicht zu nah bei einer exakten Zweierpotenz liegt, ist oft eine gute Wahl!

5.1.3 Hashing mit der Multiplikations-Methode

Die Multiplikations-Methode arbeitet in zwei Schritten. Zunächst multiplizieren wir den Schlüssel k mit einer Konstanten A, $0 < A < 1$, und extrahieren den "gebrochenen" Anteil $(kA - \lfloor kA \rfloor)$ aus kA. Dann multiplizieren wir mit m und runden das Resultat ab

$$h(k) = \lfloor m(kA \mod 1) \rfloor,$$

hierbei steht $kA \mod 1$ für den gebrochenen Anteil aus kA. Der Wert von m ist bei diesem Verfahren unkritisch! Gewöhnlich wählt man m als eine Zweierpotenz $m = 2^p$ ($p \in \mathbb{Z}$). Sei w die Wortgröße. Wir nehmen an, dass k in ein Wort passt

$$A = \frac{s}{2^w},$$

wobei s eine ganze Zahl mit

$$0 < s < 2^w$$

ist. Wir multiplizieren nun k mit s und erhalten einen 2^w-Bit Wert

$$r_1 2^w + r_0.$$

Der gesuchte Hash-Wert von k sind die ersten p Bits von r_0. Knuth schlägt für A den Wert

$$A \approx \frac{\sqrt{(5)} - 1}{2} = 0.6180339887...$$

vor.

5.1.4 Universelles Hashing

Ein "bösartiger" Gegner kann die Schlüssel immer so wählen, dass jede vorgegebene Hash-Funktion alle Schlüssel in den gleichen Slot abbildet (Suchzeit $\Theta(n)$). Jede fest vorgegebene Hash-Funktion kann durch einen solchen "Angriff" zum Worst-Case Verhalten gezwungen werden. Solche Angriffe können verhindert werden, in dem man die Hash-Funktion zufällig aus einer Menge von möglichen, sorgfältig entworfenen Hash-Funktionen (sie sollten "unabhängig" von der Schlüsselmenge sein) wählt. Dieser Ansatz wird als universelles Hashing bezeichnet.

Definition 5.3. Sei H eine endliche Menge von Hash-Funktionen, die das vorgegebene Universum U von Schlüsseln in die Menge $\{0, 1, ..., m-1\}$ abbilden. Eine solche Menge von Hash-Funktionen bezeichnet man als **universell**, wenn für jedes Paar k_i, k_j von verschiedenen Schlüsseln gilt, dass die Zahl der Funktionen $h \in H$ mit $h(k_i) = h(k_j)$ höchstens $\frac{|H|}{m}$ ist (oder bei zufälliger Wahl der Hash-Funktion gilt $p(h(k_i) = h(k_j)) \leq \frac{1}{m}$).

Satz 5.2. *Sei h eine Hash-Funktion, die aus einer universellen Menge von Hash-Funktionen zufällig gewählt wurde. Verwenden wir h, um eine Menge von n Schlüsseln in eine Hash-Tabelle T der Größe m mittels der Chaining-Technik zu speichern, so gilt, falls k_i nicht in der Tabelle gespeichert ist, dass die erwartete Länge $E[l_{h(k_i)}]$ der Liste in $h(k_i)$ höchstens α ist. Ist k_i in der Tabelle gespeichert, dann ist die erwartete Länge $E[l_{h(k_i)}]$ höchstens $1 + \alpha$.*

Beweis. Für jedes Paar k_i und k_j von Schlüsseln definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1, \text{ falls } h(k_i) = h(k_j) \\ 0, \text{ sonst} \end{cases} .$$

Aus der Definition einer universellen Menge von Hash-Funktionen folgt direkt, dass ein Paar von Schlüsseln mit einer Wahrscheinlichkeit $\leq \frac{1}{m}$ kollidiert, d.h.

$$p(h(k_i) = h(k_j)) \leq \frac{1}{m}.$$

Zu jedem Schlüssel k_i definieren wir uns eine Zufallsvariable

$$Y_i = \sum_{k_j \in T, j \neq i} X_{ij}.$$

Dann folgt

$$\begin{aligned} E[Y_i] &= E\left[\sum_{k_j \in T, j \neq i} X_{ij}\right] \\ &= \sum_{k_j \in T, j \neq i} E[X_{ij}] \\ &\leq \sum_{k_j \in T, j \neq i} \frac{1}{m}. \end{aligned}$$

Der Rest des Beweises hängt davon ab, ob k_i in der Tabelle T gespeichert ist oder nicht. Wir machen nun eine Fallunterscheidung

1. $k_i \notin T$:
   Dann ist $l_{h(k_i)} = Y_i$ und $|\{k_j : k_j \in T \wedge k_j \neq k_i\}| = n$. Wir erhalten

   $$E[l_{h(k_i)}] = E[Y_i] \leq \frac{n}{m} = \alpha.$$

2. $k_i \in T$:
   Da Y_i den Schlüssel k_i nicht einschließt, ist $l_{h(k_i)} = Y_i + 1$ und

   $$|\{k_j : k_j \in T \wedge k_j \neq k_i\}| = n - 1.$$

   Wir erhalten

   $$E[l_{h(k_i)}] = E[Y_i] + 1 \leq \frac{n-1}{m} + 1 = 1 + \alpha - \frac{1}{m} < 1 + \alpha.$$

■

Korollar 5.1. *Mit Hilfe der Kombination von universellem Hashing und Chaining kann man in einer Tabelle mit m Slots eine beliebige Folge von n INSERT-, SEARCH- und DELETE-Operationen, die $O(m)$ INSERT-Operationen enthält, in erwarteter Zeit $\Theta(n)$ ausführen.*

Beweis. Da die Zahl der Einfügungen von der Größenordnung $O(m)$ ist, ist $n = O(m)$ und $\alpha = O(1)$. Da INSERT- und DELETE-Operationen konstante Zeit benötigen und da nach Satz 5.2 auch SEARCH-Operationen erwartete Zeit von

$$O(1 + \alpha) = O(1)$$

unterhalb den obigen Annahmen erfordern, benötigt man für alle Operationen zusammen nur eine erwartete Zeit von $\Theta(n)$. ■

Eine universelle Klasse von Hash-Funktionen

Zunächst wählen wir eine Primzahl p, so dass jeder potentielle Schlüssel k kleiner als p ist. Im Weitern verwenden wir die üblichen mathematischen Bezeichnungen, d.h

$$\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z} = \{0, 1, ..., p-1\}$$

und die dazugehörige multipilkative Gruppe

$$\mathbb{Z}_p^* = (\mathbb{Z}/p\mathbb{Z})^* = \{1, ..., p-1\}.$$

Da das Universum erheblich größer als die Tabelle T sein soll, muss gelten

$$p > m.$$

Für jedes Paar (a, b) von Zahlen mit $a \in \mathbb{Z}_p^*$ und $b \in \mathbb{Z}_p$ definieren wir wie folgt eine Hash-Funktion

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

Beispiel 5.2. Sei nun $p = 17$ und $m = 6$ gegeben. Wir erhalten mit obiger Formel für den Hashwert bei einer Belegung von $a = 3$ und $b = 4$

$$h_{3,4}(8) = 5.$$

Satz 5.3. *Die Klasse*

$$H_{p,m} = \{h_{a,b} | a \in \mathbb{Z}_p^* \wedge b \in \mathbb{Z}_p\}$$

von Hash-Funktionen ist universell.

Beweis. Wir betrachten zwei beliebige, verschiedene Schlüssel k_i und k_j aus $\mathbb{Z}_p$ und eine gegebene Hash-Funktion $h_{a,b} \in H_{p,m}$. Dann setzen wir

$r = (ak_i + b) \mod p$,

$s = (ak_j + b) \mod p$.

Zunächst stellen wir fest, dass $r \neq s$ ist, da für $a \neq 0$, $(k_i - k_j) \neq 0$ und $p \in \mathbb{P}$ gilt

$$r - s \equiv (a(k_i - k_j)) \mod p.$$

Es gibt also modulo p keine Kollisionen. Darüber hinaus liefert jedes der möglichen $p(p-1)$ Paare (a, b) mit $a \neq 0$ ein anderes Paar (r, s) von Resultaten modulo p

$$a = (r - s)((k_i - k_j)^{-1} \mod p) \mod p,$$

$$b = (r - ak_i) \mod p.$$

Es gibt also eine eins-zu-eins Beziehung (Bijektion) zwischen den $p(p-1)$ Paaren (a, b) mit $a \neq 0$ und den Paaren (r, s) mit $r \neq s$. Wenn wir also für ein beliebiges vorgegebenes Paar k_i und k_j zufällig ein Paar (a, b) (eine Hash-Funktion) wählen, so ist das Resultatpaar (r, s) wieder ein "zufälliges" Paar von Zahlen modulo p (jedes Paar (r, s) kommt mit der gleichen Wahrscheinlichkeit vor). Die Wahrscheinlichkeit, dass verschiedene Schlüssel k_i und k_j kollidieren, ist gleich der Wahrscheinlichkeit, dass

$$r \equiv s \mod m$$

ist, wenn r und s zufällig modulo p gewählt werden. Wieviele Zahlen s kleiner als p mit $s \neq r$ und $s \equiv r \mod m$ gibt es für eine beliebige vorgegebene Zahl $r < p$. Es gilt

$$\frac{p}{m} - 1 \leq \frac{p + m - 1}{m} \leq \frac{p - 1}{m}$$

und wir erhalten, dass die Wahrscheinlichkeit, dafür dass s mit r modulo m kollidiert, höchstens

$$\frac{p - 1}{(p-1)m} = \frac{1}{m}$$

ist. Daher gilt für jedes beliebige Paar k_i, $k_j \in \mathbb{Z}_p$

$$p(h(k_i) = h(k_j)) \leq \frac{1}{m}.$$

■

5.1.5 Hashing mit offener Adressierung

1. Alle Elemente (Schlüssel) werden in der Hash-Tabelle selbst gespeichert, d.h. ein Slot enthält entweder einen Schlüssel oder -1.

2. Sucht man in der Tabelle nach einem Schlüssel, so durchmustert man die Slots in einer wohldefinierten Reihenfolge, die von dem gesuchten Schlüssel abhängt. Man sucht, bis man den Schlüssel gefunden hat oder bis man sicher ist, dass der Schlüssel nicht in der Tabelle gespeichert ist.

3. Auch beim Einfügen untersuchen wir die Tabelle iterativ, solange bis man einen freien Slot gefunden hat, in dem man den Schlüssel speichern kann.

4. Die Reihenfolge, in der die Slots beim Einfügen und beim Suchen durchmustert werden, hängt von dem entsprechenden Schlüssel und der Zahl der Iterationen ab. Wir arbeiten daher mit Hash-Funktionen der Form

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}.$$

5. Es ist notwendig, dass die sogenannte "Testfolge" für jeden Schlüssel k_i eine Permutation von $(0, 1, 2, ..., m-1)$ ist.

$$\text{Testfolge}(k_i) = (h(k_i, 0), h(k_i, 1), h(k_i, 2), ..., h(k_i, m-1)).$$

Wir betrachten eine Klasse "HashTable" mit zwei Datenelementen

```
1  int* T; // Zeiger auf ein C–Array von ganzen Zahlen
2  int  m; // aktuelle Groesse des C–Arrays (der Hash–Tabelle)
```

und betrachten eine Elementfunktion zum Einfügen von ganzen Zahlen, die eine Elementfunktion int HashTable::h(int k, int i) verwendet:

```
1  int HashTable::insert(int k)
2  {
3  int  i = 0, index = h(k, 0);
4  while (i < m && T[index] != -1)  index = h(k, ++i);
5  if (i < m)
6  {
7     T[index] = k;
8     return(index);
9  }
10 else
11 {
12    cout << "hash table overflow !!!" << endl;
13    return (-1);
14 }
```

Der Leser beachte bitte folgende Bemerkung.

Bemerkung 5.3. Beim Löschen eines Elements darf die Testfolge nicht unterbrochen werden. Deshalb sind weitere Strategien notwendig (siehe Übung).

Uniformes Hashing

Beim uniformen Hashing nimmt man an, dass jedem Schlüssel mit gleicher Wahrscheinlichkeit eine der $m!$ Permutationen von $(0, 1, ..., m-1)$ als Testfolge zugeordnet wird. Drei verschiedene Techniken werden beim uniformen Hashing in der Regel eingesetzt:

1. linear probing
2. quadratic probing

3. double hashing

Jede der Techniken garantiert, dass die Testfolge eine Permutation von

$$(0, 1, ..., m-1)$$

ist. Keine der Techniken erfüllt aber wirklich die eigenliche Bedingung des uniformen Hashings, dass alle Permutationen als Testfolgen gleichwahrscheinlich sind.

linear probing Sei $h' : U \to \{0, 1, ..., m-1\}$ eine gewöhnliche Hash-Funktion. Dann können wir eine neue Hash-Funktion wie folgt definieren

$$h(k, i) = (h'(k) + i) \mod m.$$

quadratic probing Sei $h' : U \to \{0, 1, ..., m-1\}$ eine gewöhnliche Hash-Funktion. Dann können wir eine neue Hash-Funktion definieren. Sei c_1 und $c_2 \neq 0$ Konstanten,

$$h(k, i) = (h'(k) + c_1 \cdot i + c_2 i^2) \mod m.$$

double hashing Seien $h_1 : U \to \{0, 1, ..., m-1\}$ und $h_2 : U \to \{0, 1, ..., m'-1\}$ gewöhnliche Hash-Funktionen. Dann können wir eine neue Hash-Funktion wie folgt definieren

$$h(k, i) = (h_1(k) + i h_2(k)) \mod m.$$

Damit man als Testfolge immer eine Permutation von $(0, 1, ..., m)$ erhält, muss m' relativ "prim" zu m sein. Zum Beispiel kann man m und m' wie folgt wählen:

1. m ist eine Primzahl und m' ist etwas kleiner als m, z.B. $m' = m - 1$
2. $h_1(k) = k \mod m$
3. $h_2(k) = 1 + (k \mod m')$

Satz 5.4. *Verwendet man für die Speicherung von n Elementen eine Hash-Tabelle der Größe m mit "load factor" $\alpha = \frac{n}{m} < 1$ und als Hashing-Technik offene Adressierung, dann gilt unter der Annahme, dass uniformes Hashing vorliegt, dass die erwartete Zahl der Tests (die Länge der erforderlichen Testfolge) bei einer nicht erfolgreichen Suche höchstens $\frac{1}{1-\alpha}$ ist.*

Beweis. Bei einer nicht erfolgreichen Suche sind alle Slots der Testfolge mit anderen Elementen besetzt, bis man dann schließlich einen leeren Slot findet. Sei X die Zahl der Tests bei einer nicht erfolgreichen Suche. A_i der i-te besuchte Slot ist besetzt und $\{X \geq i\}$ ist der Schnitt der Ereignisse $A_1 \cap A_2 \cap ... \cap A_{i-1}$. Mit der Formel

$$p(A_1 \cap A_2 \cap ... \cap A_{i-1}) = p(A_1) \cdot p(A_2|A_1) \cdot p(A_3|A_1 \cap A_2) \cdot ... \cdot p(A_{i-1}|A_1 \cap ...A_{i-2})$$

und den Wahrscheinlichkeiten

$$p(A_1) = \frac{n}{m}$$

und

$$p(A_{i-1}|A_1 \cap ...A_{i-2}) = \frac{n-i+2}{m-i+2}$$

erhalten wir direkt

$$\begin{aligned} p(\{X \geq i\}) &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot ... \cdot \frac{n-i+2}{m-i+2} \\ &\leq (\frac{n}{m})^{i-1} \\ &= \alpha^{i-1}. \end{aligned}$$

Der Erwartungswert ergibt sich somit zu

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} p(\{X = i\})i \\ &= \sum_{i=0}^{\infty} i(p(\{X \geq i\}) - (p(\{X \geq i+1\})) \\ &= \sum_{i=0}^{\infty} (p(\{X \geq i+1\}) \\ &\leq \sum_{i=0}^{\infty} \alpha^i \\ &= \frac{1}{1-\alpha}. \end{aligned}$$

■

Die obige Reihe für $E[X]$ können wir wie folgt interpretieren,

α^0 einen Test müssen wir immer durchführen,

α^1 mit ungefährer Wahrscheinlichkeit α ist der erste Slot besetzt,

α^2 mit ungefährer Wahrscheinlichkeit α^2 sind die ersten beiden Slots besetzt.

Aus Satz 5.4 folgt direkt

Korollar 5.2. *Unter den Annahmen von Satz 5.4 gilt, die erwartete Zahl der Tests für das Einfügen eines Elementes in eine Hash-Tabelle mit Belegungsfaktor α ist $\frac{1}{1-\alpha}$.*

Satz 5.5. *Unter den Annahmen von Satz 5.4 ist die erwartete Zahl von Tests bei einer erfolgreichen Suche kleiner gleich*

$$\frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\frac{1}{\alpha}}\right).$$

Beweis. Eine Suche nach dem Schlüssel k führt die gleichen Tests wie beim Einfügen des Schlüssels durch. Wurde k als $(i+1)$ Schlüssel in der Hash-Tabelle gespeichert, so folgt aus Korollar 5.2. dass die erwartete Zahl von Tests für

$$k \leq \frac{1}{1-\frac{i}{m}} \leq \frac{m}{m-i}$$

ist. Wir betrachten nun den Durchschnitt über alle Schlüssel. Sei H_i die i-te harmonische Zahl, dann gilt

$$\begin{aligned}
\frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i} \\
&= \frac{m}{n} \cdot (H_m - H_{m-n}) \\
&= \frac{1}{\alpha} \cdot \sum_{j=m-n+1}^{m} \frac{1}{j} \\
&\leq \frac{1}{\alpha} \cdot \int_{m-n}^{m} \frac{1}{x} dx \\
&= \frac{1}{\alpha} \cdot \ln(\frac{m}{m-n}) \\
&= \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\frac{1}{\alpha}}).
\end{aligned}$$

■

Ist die Hash-Tabelle halb gefüllt, so ist die erwartete Zahl von Tests 1.387. Ist die Hash-Tabelle zu 90% gefüllt, so ist die erwartete Testzahl 2.559.