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.
Insert(K,x)	----	Füge das Element x in K ein.
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).

Eine dynamische Menge, die diese Operationen unterstützt, bezeichnet man als „Wörterbuch“ (dictionary).

Hashing

	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)
Mimimum	O(1)	O(n)	O(1)	O(n)	O(n)
Maximum	O(1)	O(n)	O(1)	O(n)	O(n)

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

Hashing

3

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

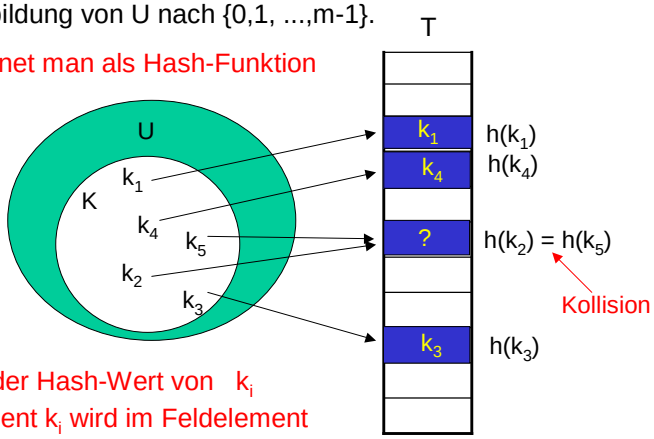
[1] Sei U das Universum aller Schlüssel.

[2] Sei K ⊇ 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



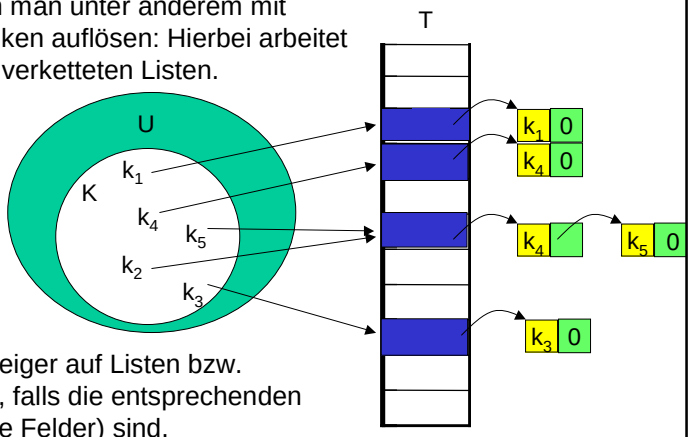
$h(k_i)$ ist der Hash-Wert von k_i

Das Element k_i wird im Feldelement mit Index $h(k_i)$ von Feld T gespeichert.

Hashing mit Chaining-Techniken

4

- 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.
- Kollisionen kann man unter anderem mit Chaining-Techniken auflösen: Hierbei arbeitet man mit einfach verketteten Listen.



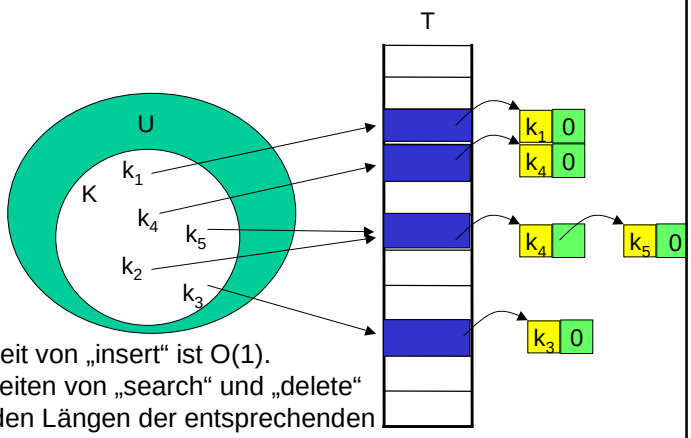
- T enthält dann Zeiger auf Listen bzw. Zeiger auf NULL, falls die entsprechenden Listen leer (weiße Felder) sind.

Hashing mit Chaining-Techniken

5

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.



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.

Hashing mit Chaining

6

Wie effizient ist 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 $\alpha = n/m$ (α = load factor) lang.

Worst case: alle n Elemente in die gleiche Liste ⇒ O(n)

Average Performance:

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

Wie effizient ist Hashing mit Chaining?

Gegeben eine Hash-Tabelle mit m Slots, die n = |K| Elemente speichert.

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 bezeichen wir mit l_j die Länge der Liste T[j]:

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

Der Durchschnittswert von l_j (erwartete Länge der Liste) ist

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

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

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

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

nächste Seite

Wie effizient ist Hashing mit Chaining?

Gegeben eine Hash-Tabelle mit m Slots, die n = |K| Elemente speichert.

Einfaches uniformes Hashing:

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:

nächste Seite

Hashing mit Chaining

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

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

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

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \quad = 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad = 1+\frac{n-1}{2m} \quad = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Satz [8]:

In einer Hash-Tabelle, in der Kollisionen mit Chaining aufgelöst werden, benötigt eine Suche (unter der Annahme: einfaches uniformes Hashing) im Durchschnitt erwartete Zeit $\Theta(1+\alpha)$.

Hashing: Die Multiplikations-Methode

Welche Eigenschaften sollte eine gute Hash-Funktion besitzen?

Eine gute Hash-Funktion sollte die Annahme über einfaches uniformes Hashing erfüllen, d.h., jedes Element sollte ungefähr mit der gleichen Wahrscheinlichkeit in irgend einen 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 und da diese Schlüssel gewöhnlich nicht unabhängig voneinander gezogen werden.

Die Divisions-Methode

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

Beispiel: m = 12 k = 100 => h(k) = 4.

Man vermeide $m = 2^p$ (Zweierpotenz) zu wählen!

Eine Primzahl, die nicht zu nah zu einer exakten Zweierpotenz ist, ist oft eine gute Wahl!

Hashing: Die Multiplikations-Methode

11

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:

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

Der Wert von m ist bei diesem Verfahren unkritisch!

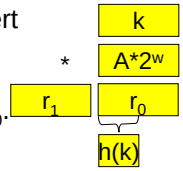
Gewöhnlich wählt man m als eine Zweierpotenz $m = 2^p$ ($p \in Z$).

Sei w die Wortgröße. Wir nehmen an, dass k in ein Wort passt.

$A = \frac{s}{2^w}$ wobei s eine ganze Zahl mit $0 < s < 2^w$

Wir multiplizieren nun k mit s und erhalten einen 2w-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 folgenden Wert vor:

$A \approx (\sqrt{5} - 1)/2 = 0.6180339887...$ ersten p Bits von r_0

Universelles Hashing

12

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 [6]:

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: Die Zahl der Funktionen $h \in H$ mit $h(k_i) = h(k_j)$ ist höchstens $|H|/m$.

(oder bei zufälliger Wahl der Hash-Funktion gilt: $p(h(k_i) = h(k_j)) \leq 1/m$)

Satz [9]:

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, so ist die erwartete Länge $E[l_{h(ki)}]$ der Liste in $h(k_i)$ höchstens α. Ist k_i in der Tabelle gespeichert, dann ist die erwartete Länge $E[l_{h(ki)}]$ 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:
Ein Paar von Schlüsseln kollidiert mit Wahrscheinlichkeit kleiner gleich 1/m.

$$p(h(k_i) = h(k_j)) \le \frac{1}{m} \quad \Rightarrow \quad E[X_{ij}] \le \frac{1}{m}$$

Zu jedem Schlüssel k_i definieren wir uns eine Zufallsvariable

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

$$E[Y_i] = E[\sum_{k_j \in T, j \neq i} X_{ij}] = \sum_{k_j \in T, j \neq i} E[X_{ij}] \le \sum_{k_j \in T, j \neq i} \frac{1}{m}$$

Der Rest des Beweises hängt davon ab, ob k_i in der Tabelle T gespeichert ist oder nicht.

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

$$\Rightarrow \quad E[l_{h(k_i)}] = E[Y_i] \le \frac{n}{m} = \alpha$$

[2] $k_i \in T$: Da Y_i den Schlüssel k_i nicht einschließt, ist $l_{h(k_i)} = Y_i + 1$ und $|\{k_j : k_j \in T \wedge k_j \neq k_i\}| = n-1$

$$\Rightarrow \quad E[l_{h(k_i)}] = E[Y_i] + 1 \le \frac{n-1}{m} + 1 = 1 + \alpha - \frac{1}{m} < 1 + \alpha$$ ■

Korollar [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 dem vorhergehenden Satz auch SEARCH-Operationen erwartete Zeit $O(1+\alpha) = O(1)$ unter den obigen Annahmen erfordern, benötigt man für alle Operationen zusammen nur erwartete Zeit $\Theta(n)$.

■

Eine universelle Klasse von Hash-Funktionen:

Zunächst wählen wir eine Primzahl p, so dass jeder potentielle Schlüssel k kleiner als p ist.

$$Z_p = \{0,1,\cdots, p-1\}$$

$$Z_p^* = \{1,\cdots, p-1\}$$

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

$$p > m$$

Für jedes Paar (a,b) von Zahlen mit $a \in Z_p^*$ und $b \in Z_p$ definieren wir wie folgt eine Hash-Funktion:

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

Beispiel: p = 17 m = 6 ⟹ $h_{3,4}(8) = 5$

Satz [10]:

Die Klasse

$$H_{p,m} = \{h_{a,b} \mid a \in Z_p^* \wedge b \in Z_p\}$$

von Hash-Funktionen ist universell.

Beweis:

Wir betrachten zwei beliebige Schlüssel k_i und k_j aus Z_p und eine gegebene Hash-Funktion $h_{a,b} \in H_{p,m}$:

$$r = (ak_i + b) \bmod p \qquad s = (ak_j + b) \bmod p$$

Zunächst stellen wir fest, dass $r \neq s$ ist, da

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

$a \neq 0$, $(k_i - k_j) \neq 0$ und p eine Primzahl ist.

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:

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

$$b = (r - ak_i) \bmod p$$

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 $r \equiv s \bmod m$ ist, wenn r und s zufällig modulo p gewählt werden.

Wieviele Zahlen s kleiner als p mit $s \neq r$ und $s \equiv r \bmod m$ gibt es für eine beliebige vorgegebene Zahl $r < p$.

$$\lceil p/m \rceil - 1 \leq ((p + m - 1)/m) - 1 \leq (p - 1)/m$$

Universelles Hashing

Die Wahrscheinlichkeit, dass s mit r modulo m kollidiert, ist höchstens

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

Daher gilt für jedes beliebige Paar k_i, $k_j \in Z_p$:

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

■

Hashing mit offener Adressierung

Offene 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:

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

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

Testfolge(k_i) = (h(k_i ,0), h(k_i ,1), h(k_i ,2), ... , h(k_i ,m-1))

Hashing mit offener Adressierung

Wir betrachten eine Klasse „HashTable“ mit zwei Datenelementen

```
int* T;      // Zeiger auf ein C-Array von ganzen Zahlen
int  m;      // aktuelle Größe 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:

```
int HashTable::insert(int k)
{
   int  i = 0, index = h(k, 0);
   while (i < m  && T[index] != -1)  index = h(k, ++i);
   if ( i < m )  {
      T[index] = k;
      return(index);
   }
   else {
      cout << “hash table overflow !!!“ << endl;
      return (–1);
   }
}
```

Uniformes Hashing

Beim uniformen Hashing nimmt man an, dass jedem Schlüssel mit gleicher Wahrscheinlichkeit eine der m! Permutationen von (0,1,...,m-1) als Test-folge zugeordnet wird.

Hashing mit offener Adressierung

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 Test-folgen gleichwahrscheinlich sind.

„linear probing“:

Sei h‘: U -> {0,1, ... , m-1} eine gewöhnliche Hash-Funktion. Dann können wir eine neue Hash-Funktion wie folgt definieren:

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

„quadratic probing“:

Sei h‘: U -> {0,1, ... , m-1} eine gewöhnliche Hash-Funktion. Dann können wir eine neue Hash-Funktion wie folgt definieren:

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

c_1 und $c_2 \neq 0$ Konstanten

„double hashing“

Seien h_1: U -> {0,1, ... , m-1} und h_2: U -> {0,1, ... , m‘-1} gewöhnliche Hash-Funktionen. Dann können wir eine neue Hash-Funktion wie folgt definieren:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

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:

- m ist eine Primzahl und m‘ ist etwas kleiner als m, z.B. m‘ = m-1
- $h_1(k) = k \bmod m$
- $h_2(k) = 1 + (k \bmod m‘)$

Satz [11]

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

Beweis:

Bei einer nicht erfolgreichen Suche sind alle Slots der Testfolge mit anderen Elementen besetzt, bis man dann schließlich einen leeren Slot findet.

X = die Zahl der Tests bei einer nicht erfolgreichen Suche.

A_i = der i-te besuchte Slot ist besetzt.

$\{ X \geq i \}$ ist der Schnitt der Ereignisse $A_1 \cap A_2 \cap ... \cap A_{i-1}$.

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

$p(A_1) = n/m$

$p(A_{i-1} | A_1 \cap A_2 \cap ... \cap A_{i-2}) = (n-i+2)/(m-i+2)$ ⇒

$$p(\{X \geq i\}) = \frac{n}{m}\frac{n-1}{m-1}\cdots\frac{n-i+2}{m-i+2}$$

Da $\frac{n-j}{m-j} \leq \frac{n}{m}$

$$\leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

für j ∈{0,1, ... , m-1}

$$E[X] = \sum_{i=0}^{\infty} p(\{X = i\})i = \sum_{i=0}^{\infty} i(p(\{X \geq i\}) - p(\{X \geq i+1\}))$$

$$= \sum_{i=0}^{\infty} p(\{X \geq i+1\}) \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

■

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

$\alpha^0 = 1$: einen Test müssen wir immer durchführen
α^1 : mit ungefährer Wahrscheinlichkeit α ist der erste Slot besetzt
α^2 : mit ungefährer WSK α^2 sind die ersten beiden Slots besetzt
usw.

Aus Satz [11] folgt direkt:

Korollar [2]:
Unter den Annahmen von Satz [11] gilt: Die erwartete Zahl der Tests für das Einfügen eines Elementes in eine Hash-Tabelle mit Belegungsfaktor α ist $1/(1-\alpha)$.

Satz [12]:
Unter den Annahmen von Satz [11] ist die erwartete Zahl von Tests bei einer erfolgreichen Suche kleiner gleich

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

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 [2]:

die erwartete Zahl von Tests für k $\leq \frac{1}{1-\frac{i}{m}} \leq \frac{m}{m-i}$ ⇨

Hashing mit offener Adressierung

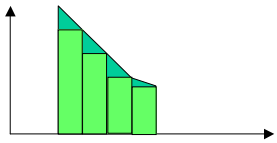
Wir betrachten nun den Durchschnitt über alle Schlüssel:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{m}{n}(H_m - H_{m-n})$$

H_i ist die i-te harmonische Zahl

$$= \frac{1}{\alpha}\sum_{j=m-n+1}^{m}\frac{1}{j} \le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx$$

$$H_i = \sum_{j=1}^{i}\frac{1}{j}$$



$$= \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right)$$

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

■

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.

Perfektes Hashing

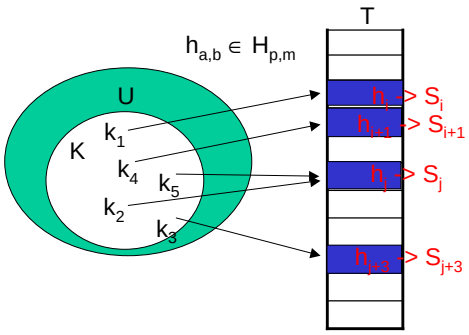
Perfektes Hashing

Man nennt eine Hashing-Technik „Perfektes Hashing“, falls die „worse case“ Zahl von Speicherzugriffen für eine Suche O(1) ist.

Idee:

Man schalte zwei universelle Hash-Funktionen hintereinander. Die erste Hash-Funktion bestimmt den Slot. Anstatt Chaining zur Auflösung von Kollisionen zu verwenden, setzt man eine zweite Hash-Funktion für alle Schlüssel eines Slots ein, die keine Kollisionen zulässt.

Die erste Hash-Funktion wird aus der universellen Klasse $H_{p,m}$ gewählt.



Für jeden Slot j gibt es eine Funktion h_j, die in die Hash-Tabelle S_j abbildet.

Um Kollisionen zu vermeiden, ist es erforderlich die Größe m_j von S_j als das Quadrat der Zahl l_j von Schlüsseln zu wählen, die in Slot S_j abgebildet werden.

$$h_j \in H_{p,m_j}$$

Satz [13]

Wenn man n Schlüssel in einer Hash-Tabelle der Größe $m = n^2$ mit einer zufällig gewählten Hash-Funktion $h_{a,b} \in H_{p,m}$ speichert, dann ist die Wahrscheinlichkeit einer Kollision kleiner als ½.

Beweis:

Es gibt n(n-1)/2 Paare, die kollidieren können. Für jedes Paar ist die WSK kleiner gleich $1/m = 1/n^2$.

Die Zufallsvariable X zählt die Kollisionen ($X_{ij} = 1$, falls $h_{a,b}(k_i) = h_{a,b}(k_j)$).

$$E[X] = E[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}] = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{1}{n^2} = \frac{n^2 - n}{2} \frac{1}{n^2} < \frac{1}{2}$$

Mit Hilfe der Markow Ungleichung (Modifikation von Satz [5])

$$p(X \geq t) \leq \frac{E[X]}{t}$$

für t = 1 folgt die Behauptung des Satzes. ■

Wir nehmen im folgenden an, dass die Menge K der betrachteten Elemente (Schlüssel) statisch ist, d.h., diese Menge ist vorgegeben und wird nicht durch Hinzufügen oder Entfernen von Elementen dynamisch verändert.

Gilt die obige Annahme und stellen wir $m = n^2$ Speicherplatz zur Verfügung, so folgt aus Satz [13]:

Mit hoher WSK können wir mit einigen wenigen Versuchen in der universellen Klasse $H_{p,m}$ von Hash-Funktionen eine Funktion finden, die bezüglich der statischen Menge K keine Kollisionen erzeugt.

Da man der Hash-Tabelle nicht quadratischen Speicherplatz $O(n^2)$ spendieren möchte, wenden wir diese Strategie nur für die sekundären Hash-Funktionen h_j der einzelnen Slots S_j an.

Die primäre Hash-Tabelle besteht aus $m = n$ Slots.

Satz [14]

Wenn man n Schlüssel in einer Hash-Tabelle der Größe m = n mit einer zufällig gewählten Hash-Funktion h aus einer universellen Klasse von Hash-Funktionen speichert, dann ist

$$E[\sum_{j=0}^{m-1} l_j^2] < 2n$$

Beweis:

Wir verwenden die folgende Identität für eine beliebige nicht-negative ganze Zahl:

$$a^2 = a + 2\binom{a}{2}$$

Es gilt:

$$E[\sum_{j=0}^{m-1} l_j^2] = E[\sum_{j=0}^{m-1}\left(l_j + 2\binom{l_j}{2}\right)] = E[\sum_{j=0}^{m-1} l_j] + 2E\left[\sum_{j=0}^{m-1}\binom{l_j}{2}\right] = E[n] + 2E\left[\sum_{j=0}^{m-1}\binom{l_j}{2}\right]$$

$$= n + 2E\left[\sum_{j=0}^{m-1}\binom{l_j}{2}\right] \quad \text{wobei} \quad \sum_{j=0}^{m-1}\binom{l_j}{2} \quad \text{die Gesamtzahl der Kollisionen ist.}$$

$$E[\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} X_{ij}] = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} E[X_{ij}] \le \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\frac{1}{m} = \frac{n^2-n}{2}\frac{1}{m} = \frac{n-1}{2}$$

da m = n

Hieraus folgt:

$$E[\sum_{j=0}^{m-1} l_j^2] \le n + 2\frac{n-1}{2} = 2n-1 < 2n$$

■

Aus Satz [14] folgt:

Korollar [3]:

Unter den Annahmen von Satz [14] gilt für eine zufällig gewählte Hash-Funktion aus einer universellen Klasse von Hash-Funktionen:
Ist die Größe m_j der sekundären Hash-Tabelle für jedes j = 0, 1, ... , m-1 gleich $m_j = l_j^2$, dann ist der erwartete Speicherplatzbedarf für alle sekundären Hash-Tabellen kleiner als 2n.

Korollar [4]:

Unter den Annahmen von Satz [14] gilt für eine zufällig gewählte Hash-Funktion aus einer universellen Klasse von Hash-Funktionen:
Ist die Größe m_j der sekundären Hash-Tabelle für jedes j = 0, 1, ... , m-1 gleich $m_j = l_j^2$, dann ist die Wahrscheinlichkeit, dass der Speicherplatz für alle sekundären Hash-Tabellen größer als 4n ist, kleiner als 1/2.

Beweis: Aus der Markow Ungleichung (Modifikation von Satz [5])

$$p(\{X \geq t\}) \leq \frac{E[X]}{t}$$

folgt für t = 4n

$$p(\{\sum_{j=0}^{m-1} l_j^2 \geq 4n\}) \leq \frac{E\left[\sum_{j=0}^{m-1} l_j^2\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}$$

■