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.
- Predessor(K,x) Suche das Element, dessen Schlüssel in der Ordnung von K dem Schlüssel von x vorangeht (nächst kleinere Schlüssel).
- Successor(K,x) Suche das Element, dessen Schlüssel in der Ordnung von K auf den Schlüssel von x folgt (nächst größere 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

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

- Sei U das Universum aller Schlüssel.
- Sei T ein Feld der Größe m.

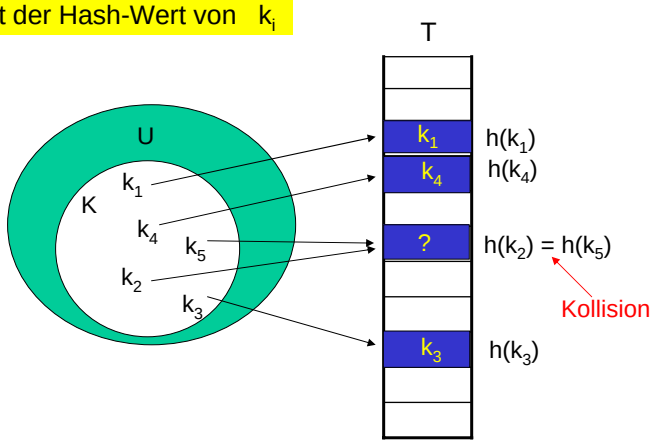
Eine Abbildung h von U nach T = { 0, 1, 2, …, m-1} heißt Hash-Funktion:

$$h : U \to \{0,1,2,\Lambda\ ,m-1\}$$

Hashing

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

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

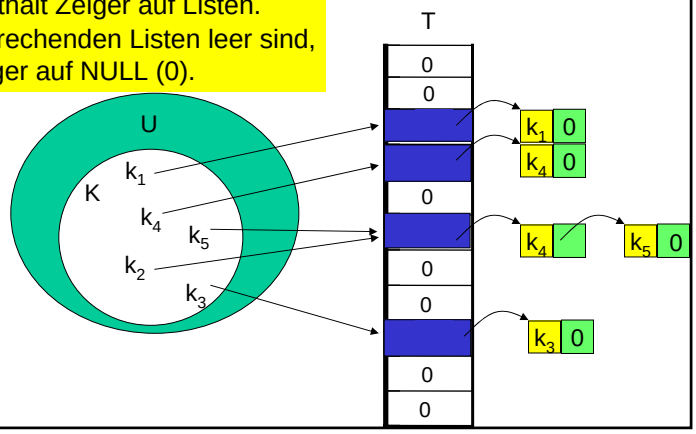


Das Element k_i wird im Feld T unter dem Index $h(k_i)$ gespeichert.

Hashing

- Hashing macht dann Sinn, wenn |K| << |U|.
- Für die Größe m des Feldes T sollte gelten: m = c |K|.
- Hash-Funktionen sollten keine oder nur wenige Kollisionen verursachen.
- Kollisionen kann man unter anderem mit Chaining-Techniken auflösen:

Chaining: T enthält Zeiger auf Listen.
Falls die entsprechenden Listen leer sind, zeigen die Zeiger auf NULL (0).

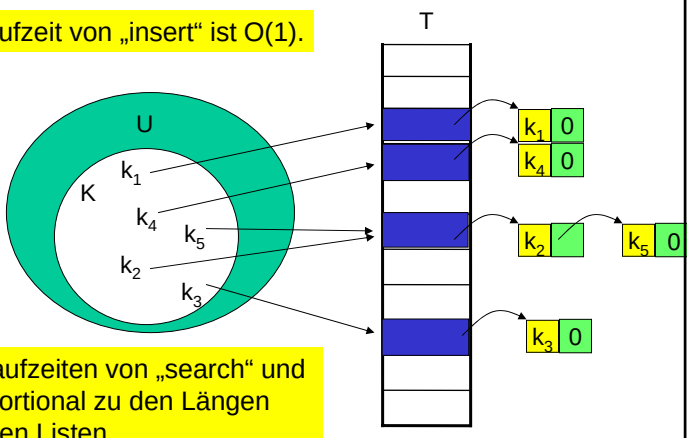


Hashing

Wir definieren eine Klasse „ChainedHashTable“ mit den 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

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-Analyse:

Alle n Elemente in die gleiche Liste ⇒ O(n)

Average Performance:

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

⇒

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]:

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

Der Durchschnittswert von l_j (erwartete Länge der Liste) ist

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

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.

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

Hashing mit Chaining

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

Einfaches uniformes Hashing:

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, betrachten 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

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

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

Wir berechnen nun den zu erwartenden durchschnittlichen Arbeitsaufwand für den Fall, dass der zu suchende Schlüssel k_i in der Liste enthalten ist:

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

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \quad \Rightarrow$$

Hashing mit Chaining

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

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

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

$$= 1+\frac{n-1}{2m} \quad = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Satz [1]:

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

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

arbeitet in zwei Schritten:

Schritt 1: Zunächst multiplizieren wir den Schlüssel k mit einer Konstanten A, 0 < A < 1, und extrahieren die Stellen hinter dem Komma (Punkt)

$$kA - \lfloor kA \rfloor = kA \bmod 1.$$

Schritt 2: Dann multiplizieren wir mit m und runden 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

$$m = 2^p \ (p \in Z).$$

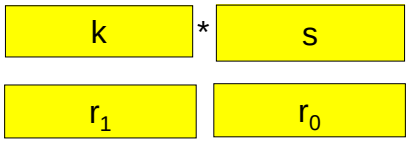
Hashing

Sei w die Wortgröße (Zahl der Bits). 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

$r_1 2^w + r_0$



Sei $m = 2^p$.

Die ersten p Bits von r_0 sind der Hash-Wert h(k) von k: h(k)

p Bits von r_0

Knuth schlägt für A den folgenden Wert vor:

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

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 Hash-Funktionen (diese sollten „unabhängig“ von der Schlüsselmenge sein) wählt.

Dieser Ansatz wird als universelles Hashing bezeichnet.

Definition [1]:

Sei H eine endliche Menge von Hash-Funktionen:

$$H = \{h \mid\mid h : U \to \{0,1,\Lambda\ ,m-1\}\}$$

H heißt universell, wenn für jedes Paar k_i , k_j von Schlüsseln mit $k_i \neq k_j$ gilt:

$$\#\{h \in H \mid\mid h(k_i) = h(k_j)\} \leq \frac{\#H}{m}$$

d.h., die Zahl der Funktionen $h \in H$ mit $h(k_i) = h(k_j)$ ist höchstens $\#H/m$.

oder anders formuliert: bei zufälliger Wahl der Hash-Funktion $h \in H$ gilt:

$$p(h(k_i) = h(k_j)) \leq 1/m$$

Satz [2]:

Sei H eine universelle Menge von Hash-Funktionen und sei h eine zufällig gewählte Hash-Funktion aus H.

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:

(a) Falls k_i nicht in der Tabelle gespeichert ist, so ist die erwartete Länge $E[l_{h(ki)}]$ der Liste von $h(k_i)$ höchstens α.

(b) Ist k_i in der Tabelle gespeichert, dann ist die erwartete Länge $E[l_{h(ki)}]$ höchstens $1+\alpha$.

Beweis von Satz [2]:

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 kollidiert mit Wahrscheinlichkeit kleiner gleich 1/m.

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

Zu jedem Schlüssel k_i definieren wir uns eine Zufallsvariable

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

Hashing

Es gilt:

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

Es folgt eine Fallunterscheidung: k_i in T gespeichert oder nicht?

Fall [1]: $k_i \notin T$:

$$l_{h(k_i)} = Y_i \quad \text{und} \quad |\{k_j : k_j \in T \wedge k_j \neq k_i\}| = n$$

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

Hashing

Fall [2]: $k_i \in T$: Da Y_i den Schlüssel k_i nicht einschließt, ist

$$l_{h(k_i)} = Y_i + 1 \quad \text{und} \quad |\{k_j : k_j \in T \wedge k_j \neq k_i\}| = n - 1$$

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

■

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 INSERTs von der Größenordnung O(m) ist, ist $\alpha = O(1)$.

INSERT-Operationen $\in O(1)$

DELETE-Operationen $\in O(1)$

Satz [2] $\Rightarrow$ SEARCH-Operationen $\in O(1+\alpha) = O(1)$ erwartet

Also benötigt man für alle n Operationen erwartete Zeit $\Theta(n)$.

■