Vorlesung Programmieren II

Michael Bergau

Fortsetzung der Stoffeinheit

Hashing

Was ist Hashing?

Hashing ist eine Methode zur dynamischen Verwaltung von Daten, wobei die Daten durch einen Schlüssel (key) angesprochen werden.

Was sind Dictionary Operationen?

- Suchen nach einem Datensatz bei gegebenem Schlüssel.
- Einfügen eines neuen Datensatzes samt Schlüssel.
- Löschen eines Datensatzes von einem gegebenen Schlüssel.

Was versteht man unter einer Kollision?

Werden zwei verschiedene Schlüssel k und k‘ auf die gleiche Adresse abgebildet, d.h. h(k) = h(k‘) dann sprechen wir von einer Kollision.

Verschiedene Beispiele für Hash-Funktionen

- ~~h(k) = 0 für alle Schlüssel k (einfachste Hashfunktion).~~
- ~~h(k) = 2 * ASCII(k_0) + 3* ASCII(k_1) + 5*ASCII(k_2) + ... Für k =$k_0k_1k_2$.....~~

Wann kann man sagen, dass eine Hash-Funktion „gut“ ist?

Hashing

Wann kann man sagen, dass eine Hash-Funktion „gut“ ist?

- Eine Hash-Funktion muss einfach berechenbar sein.
- Eine Hash-Funktion muss die Schlüssel gleichmäßig über dem Indexbereich der Hash-Tabelle verteilen, damit Kollisionen möglichst selten auftreten.

Injektive Hash-Funktion

Das heißt, je zwei verschiedene tatsächlich verwendete Schlüssel k, k‘ haben verschiedene Hashwerte h(k) ≠h(k‘).

⇨ Dictionary Operationen in Zeit O(1) realisierbar.

Problem

Im Allgemeinen können wir nicht davon ausgehen, dass die Hashfunktion h auf der verwendeten Schlüsselmenge K injektiv ist.
Die Wahrscheinlichkeitsrechung zeigt, dass es sogar bei einer sehr kleinen Schlüsselmenge (|K| << |T|) schon sehr wahrscheinlich ist, dass Kollisionen auftreten.

⇨ Geburtstagsproblem (Geburtstagsparadoxon)

Hashing

Geburtstagsproblem (Geburtstagsparadoxon)

Bei wie vielen, zufällig ausgewählten Personen ist es wahrscheinlich, dass mindestens zwei Personen am gleichen Tag Geburtstag haben?

Antwort

Es ist bereits ab 23 Personen wahrscheinlich, dass 2 Personen am gleichen Tag Geburtstag haben.

Folgen für Hashing

Bei einer Hashtabelle der Größe m = 365 muss bereits nach 23 zufälligen Einträgen mit mindestens einer Kollision gerechnet werden.

⇨ Wir benötigen Strategien zur Kollisionsbehandlung

- Hashing mit Chaining (geschlossene Adressierung)
- Verfahren mit offener Adressierung

Weiteres Problem

Hashing hat ein sehr schlechtes worst-case Verhalten

⇨ Suche in einer linearen Liste (Komplexität O(n))

Universelles Hashing

Ein „bösartiger“ Gegner kann die Schlüssel immer so wählen, dass jede vorgegebene Hash-Funktion alle Schlüssel in den gleichen Slot abbildet (Suchzeit O(n)).

Jede fest vorgegebene Hash-Funktion kann durch einen solchen „Angriff“ zum Worst-Case-Verhalten gezwungen werden.

Solche Angriffe können verhindert werden, indem 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.

⇒ Universial Hashing ist also ein probabilistisches Verfahren

Eine universelle Klasse von Hash-Funktionen

Wir wählen eine Primzahl p, so dass jeder Schlüssel k kleiner als p ist.

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

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

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 ⇒ $h_{3,4}(8) = 5$

Hashing

Satz [3]:

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, verschiedene 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.

Hashing

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.

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

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

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 $k_i, k_j \in Z_p$:

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

■

Offene Adressierung

- Alle Elemente (Schlüssel) werden in der Hash-Tabelle selbst gespeichert, d.h., ein Slot enthält entweder einen Schlüssel oder -1 (NIL).
- Sucht man in der Tabelle nach einem Schlüssel, so durchmustert man die Slots in einer wohl-definierten 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.

Offene Adressierung

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

- Die Testfolge TF, in der die Slots beim Einfügen und beim Suchen durchmustert werden, hängt vom Schlüssel und der Zahl der Iterationen ab. Wir arbeiten daher mit Hash Funktionen der Form:

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

$$TF(k_i) = (h(k_i,0), h(k_i,1), h(k_i,2), \ldots, h(k_i,m-1))$$

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

Wir betrachten eine Klasse „HashTable“ mit zwei Datenelementen

```
vector<int> v;        // Vektor von ganzen Zahlen
int  m;               // aktuelle Größe m 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-1  && v[index] != -1)  index = h(k, ++i);
        if ( i < m )  {
                v[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 Testfolge zugeordnet wird.

Drei verschiedene Techniken:

- linear probing
- quadratic probing
- double hashing

Jede der Techniken garantiert, dass die Testfolge eine Permutation von (0,1, ... , m-1) ist. Keine der Techniken erfüllt aber die eigentliche Bedingung des uniformen Hashings, dass alle Permutationen als Testfolgen gleich wahrscheinlich sind.

„linear probing“:

Sei h‘: U -> {0,1, ... , m-1} eine gewöhnliche Hash-Funktion. Dann kann man neue Hash-Funktionen 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 kann man neue Hash-Funktionen 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 kann man neue Hash-Funktionen 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 [4]

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 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 $A_1 \cap A_2 \cap ... \cap A_{i-1}$.

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

$p(A_1) = n/m$

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

Hashing

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

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

für j $\in$ {0,1, ... , m-1}

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

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

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

■

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.

Hashing

Aus Satz [4] folgt direkt:

Korollar [2]:

Unter den Annahmen von Satz [4] gilt: Die erwartete Zahl der Tests für das Einfügen eines Elementes in eine Hash-Tabelle mit Belegungsfaktor α ist kleiner gleich 1/(1- α).

Satz [5]:

Unter den Annahmen von Satz [4] 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 $\leq \frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$ ⇨

Wir betrachten nun den Durchschnitt über alle n Schlüssel:

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

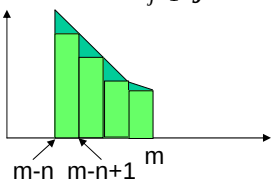
$$= \frac{1}{\alpha}\sum_{j=m-n+1}^{m}\frac{1}{j} \le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx$$

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

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

H_i ist die i-te harmonische Zahl

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



■

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.

Weiteres Sondierungsverfahren

Coalesced Hashing

Diese Methode verbindet die Platzvorteile des offenen Hashing mit den Zeitvorteilen des Chaining-Verfahrens.

Methode

Die Elemente der Hash-Tabelle bestehen aus Datensätzen mit einem Schlüssel und einem Kettungszeiger.

- Ein Schlüssel wird gesucht, indem man zunächst seine Hash-Adresse ermittelt und die dort beginnende lineare Liste solange durchläuft, bis man ihn gefunden hat oder das Ende der Überlaufliste erreicht wird.
- Ein Satz wird eingefügt, indem man zunächst nach seinem Schlüssel sucht. Wird dabei das Ende der Überlaufliste erreicht, ohne ihn zu finden, so reserviert man dasjenige freie Element der Hash-Tabelle mit der größten Adresse, trägt den neuen Satz dort ein und hängt ihn an das Ende der Überlauf-Liste an.
- Der Schlüssel des zu löschenden Satzes wird gesucht, das Element der Hashtabelle aus der Überlaufliste ausgehängt und freigegeben.

Allgemeine Probleme

Umwandlung nicht numerischer Schlüssel

Häufig haben wir Problemstellungen mit nicht numerischen Schlüsseln

- Direkte Interpretation der Zeichenfolge als Binärzahl ist ungünstig
  1. Zu große unhandliche Zahlen
  2. In der Regel keine zusammenhängenden Intervalle
- Bijektive Abbildung der in den Schlüsseln erlaubten Alphabetzeichen auf ein Intervall [1, q]. Also durch eine Abbildung

$$\sigma : \{k_0, ..., k_{q-1}\} \rightarrow \{1, ..., q\}$$

so kann man einer Zeichenfolge $k_0k_1...k_s$ die folgende Zahl

$$\sum_{i=0}^{s} \sigma(k_i)(q+1)^i$$

zuordnen.

Ende

Vielen Dank für Ihre Aufmerksamkeit
und ein schönes Wochenende