Hashtabellen

Das Problem

Man möchte Daten, die in der Form *(Schlüssel, Wert)* gespeichert sind, schnell wiederfinden. Beispiel:

ID	Mitarbeiter
42	Max Muster
79	Susi Sorglos
60	Rudi Ratlos

Eine effiziente Lösung dieser Aufgabe ist unter anderem bei folgenden Anwendungen von grosser Bedeutung.

- Datenbanken
- Websuche
- Symboltabellen in Programmiersprachen (`a=5`, `pi=3.14`, ...)

Lösung mit Arrays

Die Daten werden als List of Lists (LoL) gespeichert:

```
data = [
    [42, Max Muster],
    [79, Susi Sorglos],
    [60, Rudi Ratlos]
]
```

Wie viele Vergleiche benötigt man im Mittel, bei einer Liste der Länge n, um einen Schlüssel darin zu finden?

$n/2$ Vergleiche O(n)

Hashtabellen

Stelle zwei Listen mit jeweils n *Buckets* (Behältern) für die Schlüssel und die Werte bereit. Zum Beispiel $n = 11$:

0	1	2	3	4	5	6	7	8	9	10

Um ein Schlüssel-Wert-Paar in dieser Datenstruktur abzulegen, verwendet man eine Streuwertfunktion (*hash function*) $h(k)$, die den Schlüssel (hier k) auf die Adresse eines Buckets abbildet. Ist der Schlüssel k eine ganze Zahl, so kann man als Hashfunktion den Divisionsrest bei der Division durch n verwenden.

Im Beispiel $h(k) = (k \bmod 11)$:

$h(42) = 9$, $h(79) = 2$, $h(60) = 5$

Der Wert, den die Hashfunktion aus dem Schlüssel k berechnet, ist die Nummer des Zellenpaars, in die der Schlüssel und der Wert untergebracht werden. Aus Platzgründen sind die Werte mit „. . . “ gekennzeichnet.

0	1	2	3	4	5	6	7	8	9	10
		79			60				42	
		. . .			. . .				. . .	

Anforderungen

Gute Hashfunktionen (a) sind konsistent [kein Hashwert gehört zu zwei Schlüsseln] (b) sind effizient berechenbar und (c) verteilen die Schlüssel gleichmässig.

Kollisionen

Was geschieht, wenn man den neuen Mitarbeiter

ID	Mitarbeiter
35	Tim Thaler

zur Hashtabelle hinzufügen möchte?

$h(35) = 2$ diese Zelle ist bereits besetzt!

In einem solchen Fall spricht man von einer *Kollision.*

Um solche Kollisionen aufzulösen, gibt es verschiedene Strategien.

Lineares Sondieren

Beim *linearen Sondieren* (*linear probing*) handelt es sich um eine ganze Familie von Verfahren, die Kollisionen auflösen.

Im einfachsten Fall wählt man die nächste freie Zelle aus. Ist auch diese besetzt, geht man zur darauffolgenden freien Zelle. Dies wiederholt man so lange, bis man fündig wird. Sollte man dabei am Ende des Zellenbereichs ankommen, setzt man die Suche bei der ersten Zelle fort.

im Beispiel: $h(35) = 2$ (besetzt) $\rightarrow$ 3 (frei)

0	1	2	3	4	5	6	7	8	9	10
		79	*35*		60				42	
		. . .	. . .		. . .				. . .	

Bei der Suche nach einem Schlüssel (35), geht man analog vor:

1. Berechne den Hashwert $35 \bmod 11 = 2$
2. gehe zum Bucket mit diesem Index.

(a) Bucket ist leer ⇒ Schlüssel nicht in Tabelle

(b) Schlüssel = 35: ⇒ Wert zurückgeben

(c) Schlüssel ≠ 35: prüfe sukzessive und zyklisch den rechten Zellennachbarn, bis

2.3.1 der Schlüssel gefunden wird ⇒ Wert zurückgeben

2.3.2 man auf einen leeren Bucket trifft ⇒ Schlüssel nicht in Tabelle

Sondierungsvarianten

Ein Problem des linearen Sondierens besteht darin, dass Kollisionen zu Anhäufungen besetzter Zellen führen können. Dies verlangsamt das Einfügen und Wiederfinden von Daten.

Daher gibt es verschiedene Varianten des Sondierens:

- Anstelle der darauffolgenden Position werden bei der Suche nach dem nächsten freien Platz k Positionen (modulo n) übersprungen.
- Zuerst prüft man, ob die Zelle $h(k) + 1$ frei ist, danach, ob die Zelle $h(k) + 4$ frei ist, dann ob die Zelle $h(k) + 9$ frei ist usw. Dann spricht man jedoch nicht mehr von linearem, sondern von *quadratischem Sondieren*.

Grössenanpassung

Beim linearen Sondieren hängt die Leistung des Verfahrens vom *Lastfaktor* ab. Beim linearen Sondieren ist dies

$$\alpha = \frac{\text{Anzahl besetzter Tabelleneinträge}}{\text{Grösse der Tabelle}}$$

sicherlich gilt: $0 \leq \alpha \leq 1$

optimal: $\frac{1}{8} \leq \alpha \leq \frac{1}{2}$

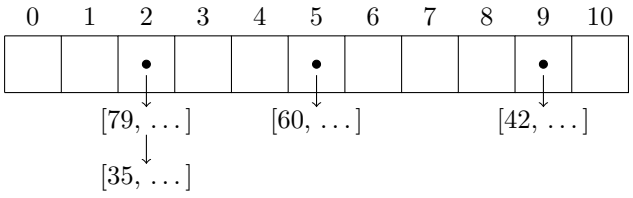
Wächst α über die obere Grenze hinaus, sollte die Tabelle vergrössert werden (verdoppeln). In diesem Fall müssen alle Schlüssel der zu kleinen Tabelle erneut gehasht werden, um sie in die grössere Tabelle zu übernehmen (*Rehashing*).

Beim Verkleinern der Hashtabelle geht man analog vor.

Hashing mit Verkettung

Eine Alternative zum Sondieren besteht darin, dass man die Schlüssel-Wert-Paare in einer erweiterbaren Liste ablegt.

Bei einer Kollision wird der neue Wert ans Ende dieser Liste angehängt.



Zum Auffinden des Wertes muss diese Liste zwar durchsucht werden, was aber bei wenig Kollisionen (und damit kurzen Listen) nicht so aufwändig ist.

Grössenanpassung

Beim Hashing mit Verkettung beträgt der Lastfaktor

$$\begin{aligned}\alpha &= \frac{\text{Anzahl Schlüssel in der Tabelle}}{\text{Anzahl der Listen}} \\ &= \text{Mittlere Anzahl Schlüssel pro Liste}\end{aligned}$$

„gute“ Lastfaktoren: $2 \leq \alpha \leq 8$

andernfalls sollte ein Rehashing durchgeführt werden.

Textschlüssel

Liegt der Schlüssel als String vor, kann man beispielsweise die Nummer der Zeichencodierung als Argument für die Hashfunktion verwenden

Beispiel: Der Schlüssel `'a'` würde bei der ASCII-Codierung der Zahl 97 entsprechen.

Besteht der Schlüssel aus mehreren Zeiche, könnte man die Codenummern der einzelnen Zeichen addieren.

Beispiel: Der Schlüssel `'cat'` würde bei der ASCII-Codierung der Zahl $99+97+116 = 312$ entsprechen.

Das Problem bei diesem Vorgehen ist, dass mehrere Zeichenketten auf die gleiche Zahl abgebildet werden. (z. B. 'cat' und 'tac')

Eine mögliche Lösung besteht darin, den Zeichencode positionsabhängig zu gewichten.

`'cat'`: $99 \cdot 1 + 97 \cdot 2 + 116 \cdot 3 = 641$

`'tac'`: $116 \cdot 1 + 97 \cdot 2 + 99 \cdot 3 = 607$

Oder wie im Stellenwertsystem (bei maximal 128 Zeichen):

`'cat'`: $99 \cdot 128^0 + 97 \cdot 128^1 + 116 \cdot 128^2 = 1\,913\,059$

`'tac'`: $116 \cdot 128^0 + 97 \cdot 128^1 + 99 \cdot 128^2 = 1\,634\,548$

Assoziative Arrays

Viele Programmiersprachen besitzen eine Datenstruktur, die es erlaubt, Schlüssel-Wert-Paare (*key-value-pairs*) effizient zu speichern und wiederzufinden. In den Grundzügen funktionieren sie wie oben beschrieben. Die Wahl einer geeigneten Hashfunktion ist ein Forschungsgegenstand.

- PHP: *Assoziatives Array*
- C++ und Java: *Map*
- Python: *Dictionary*