

11. Hashing

Hashtabellen, Pre-Hashing, Hashing, Kollisionsauflösung durch Verketteten, Einfaches gleichmässiges Hashing, Gebräuchliche Hashfunktionen, Tabellenvergrößerung, offene Addressierung: Sondieren
[Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

Motivierendes Beispiel

Ziel: Effiziente Verwaltung einer Tabelle aller n ETH-Studenten

Mögliche Anforderung: Schneller Zugriff (Einfügen, Löschen, Finden) von Datensätzen nach Name.

Wörterbuch (Dictionary)

Abstrakter Datentyp (ADT) D zur Verwaltung einer Menge von Einträgen¹² i mit Schlüsseln $k \in \mathcal{K}$. Operationen

- **D.insert**(i): Hinzufügen oder Überschreiben von i im Wörterbuch D .
- **D.delete**(i): Löschen von i aus dem Wörterbuch D . Nicht vorhanden \Rightarrow Fehlermeldung.
- **D.search**(k): Liefert Eintrag mit Schlüssel k , wenn er existiert.

¹²Schlüssel-Wert Paare (k, v) , im Folgenden betrachten wir hauptsächlich die Schlüssel.

Wörterbücher in Python

Wörterbuch →

```
fruits = {  
    "banana": 2.95, "kiwi": 0.70,  
    "pear": 4.20, "apple": 3.95  
}
```

Einfügen →

```
fruits["melon"] = 3.95
```

Verändern →

```
fruits["banana"] = 1.90
```

Suchen →

```
print("banana", fruits["banana"])  
print("melon in fruits", "melon" in  
fruits)print("onion in fruits"  
, "onion" in fruits)
```

Löschen →

```
del fruits["strawberry"]
```

Iterieren →

```
for name,price in fruits.items():  
    print(name,"->",price)
```

Wörterbücher in Java

Wörterbuch → `Map<String,Double> fruits =
new HashMap<String,Double>();`

Einfügen → `fruits.put("banana", 2.95);
fruits.put("kiwi", 0.70);
fruits.put("strawberry", 9.95);
fruits.put("pear", 4.20);
fruits.put("apple", 3.95);`

Verändern → `fruits.put("banana", 2.90);`

Suchen → `Out.println("banana " + fruits.get("banana"));`

Löschen → `fruits.remove("banana");`

Iterieren → `for (String s: fruits.keySet())
Out.println(s+ " " + fruits.get(s));`

Motivation / Verwendung

Wahrscheinlich **die** gängigste Datenstruktur

- Unterstützt in vielen Programmiersprachen (C++, Java, Python, Ruby, Javascript, C# ...)
- Offensichtliche Verwendung
 - Datenbanken / Tabellenkalkulation
 - Symboltabellen in Compilern und Interpretern
- Weniger offensichtlich
 - Substring Suche (Google, grep)
 - Ähnlichkeit von Texten (Dokumentenvergleich, DNA)
 - Dateisynchronisation
 - Kryptographie: Filetransfer / Identifikation

1. Idee: Direkter Zugriff (Array)

Index	Eintrag
0	-
1	-
2	-
3	[3,wert(3)]
4	-
5	-
⋮	⋮
k	[k,wert(k)]
⋮	⋮

Probleme

1. Schlüssel müssen nichtnegative ganze Zahlen sein
2. Grosser Schlüsselbereich \Rightarrow grosses Array

Lösung zum ersten Problem: Pre-hashing

Prehashing: Bilde Schlüssel ab auf positive Ganzzahlen mit einer Funktion $ph : \mathcal{K} \rightarrow \mathbb{N}$

- Theoretisch immer möglich, denn jeder Schlüssel ist als Bitsequenz im Computer gespeichert
- Theoretisch auch: $x = y \Leftrightarrow ph(x) = ph(y)$
- In der Praxis: APIs bieten Funktionen zum pre-hashing an. (Java: `object.hashCode()`, C++: `std::hash<>`, Python: `hash(object)`)
- APIs bilden einen Schlüssel aus der Schlüsselmenge ab auf eine Ganzzahl mit beschränkter Grösse.¹³

¹³Somit gilt die Implikation $ph(x) = ph(y) \Rightarrow x = y$ **nicht** mehr für alle x, y .

Prehashing Beispiel: String

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ zu Schlüssel

$$ph(s) = \left(\sum_{i=0}^{l_s-1} s_{l_s-i} \cdot b^i \right) \bmod 2^w$$

b so, dass verschiedene Namen möglichst verschiedene Schlüssel erhalten.

w Wortgrösse des Systems (z.B. 32 oder 64).

Beispiel (Java), mit $b = 31$, $w = 32$ Ascii-Werte s_i .

Anna \mapsto 2045632

Jacqueline \mapsto 2042089953442505 $\bmod 2^{32} = 507919049$

Implementation Prehashing (String) in Java

$$ph_{b,m}(s) = \left(\sum_{i=0}^{l-1} s_{l-i+1} \cdot b^i \right) \bmod m$$

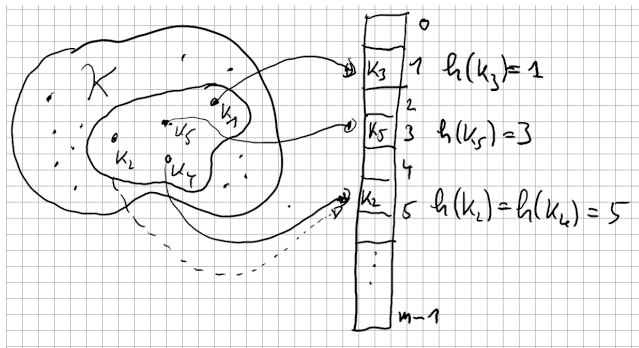
Mit $b = 31$ und $m = 2^{32}$ erhält man in Java¹⁴

```
int prehash(String s){
    int h = 0;
    for (int k = 0; k < s.length(); ++k){
        h = h * b + s.charAt(k);
    }
    return h;
}
```

¹⁴Machen Sie sich klar, warum das funktioniert

Lösung zum zweiten Problem: Hashing

Reduziere des Schlüsseluniversum: Abbildung (Hash-Funktion)
 $h : \mathcal{K} \rightarrow \{0, \dots, m - 1\}$ ($m \approx n =$ Anzahl Einträge in der Tabelle)



Kollision: $h(k_i) = h(k_j)$.

Nomenklatur

Hashfunktion h : Abbildung aus der Menge der Schlüssel \mathcal{K} auf die Indexmenge $\{0, 1, \dots, m - 1\}$ eines Arrays (**Hashtabelle**).

$$h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

Meist $|\mathcal{K}| \gg m$. Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (**Kollision**).

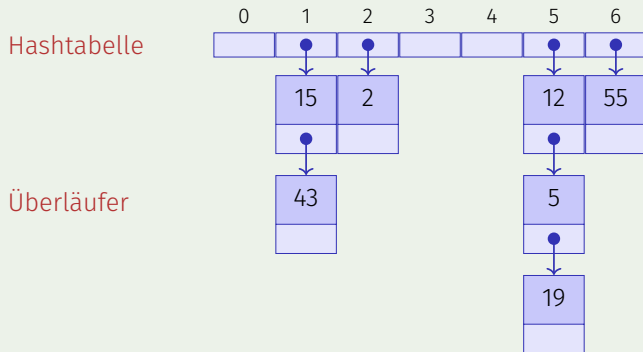
Eine Hashfunktion sollte die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle verteilen.

Behandlung von Kollisionen: Verkettung

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \bmod m.$

Schlüssel 12, 55, 5, 15, 2, 19, 43

Direkte Verkettung der Überläufer



Algorithmen zum Hashing mit Verkettung

- **insert**(i) Prüfe ob Schlüssel k vom Eintrag i in Liste an Position $h(k)$. Falls nein, füge i am Ende der Liste ein; andernfalls ersetze das Element durch i .
- **find**(k) Prüfe ob Schlüssel k in Liste an Position $h(k)$. Falls ja, gib die Daten zum Schlüssel k zurück. Andernfalls Rückgabe eines leeren Elements **null**.
- **delete**(k) Durchsuche die Liste an Position $h(k)$ nach k . Wenn Suche erfolgreich, entferne das entsprechende Listenelement.

Worst-case Analyse

Schlechtester Fall: alle Schlüssel werden auf den gleichen Index abgebildet.

⇒ $\Theta(n)$ pro Operation im schlechtesten Fall. 😞

Einfaches Gleichmässiges Hashing

Starke Annahmen: Jeder beliebige Schlüssel wird

- mit gleicher Wahrscheinlichkeit (Uniformität)
- und unabhängig von den anderen Schlüsseln (Unabhängigkeit)

auf einen der m verfügbaren Slots abgebildet.

Einfaches Gleichmässiges Hashing

Unter der Voraussetzung von einfachem gleichmässigen Hashing:
Erwartete Länge einer Kette, wenn n Elemente in eine Hashtabelle mit m Elementen eingefügt werden

$$\begin{aligned}\mathbb{E}(\text{Länge Kette } j) &= \mathbb{E}\left(\sum_{i=0}^{n-1} \mathbb{1}(k_i = j)\right) = \sum_{i=0}^{n-1} \mathbb{P}(k_i = j) \\ &= \sum_{i=1}^n \frac{1}{m} = \frac{n}{m}\end{aligned}$$

$\alpha = n/m$ heisst **Belegungsfaktor** oder **Füllgrad** der Hashtabelle.

Einfaches Gleichmässiges Hashing

Theorem 5

Sei eine Hashtabelle Verkettung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. Unter der Annahme vom einfachen gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von $\leq 1 + \alpha$.

Folgerung: ist die Anzahl der Slots m der Hashtabelle immer mindestens proportional zur Anzahl Elemente n in der Hashtabelle, $n \in \mathcal{O}(m) \Rightarrow$ Erwartete Laufzeit der Operationen Suchen, Einfügen und Löschen ist $\mathcal{O}(1)$.

Vor und Nachteile der Verkettung

Vorteile der Strategie:

- Belegungsfaktoren $\alpha > 1$ möglich
- Entfernen von Schlüsseln einfach

Nachteile

- Speicherverbrauch der Verkettung

Ein Beispiel einer gebräuchlichen Hashfunktion

Divisionsmethode

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

Ideal: m Primzahl, nicht zu nahe bei Potenzen von 2 oder 10

Aber oft: $m = 2^k - 1$ ($k \in \mathbb{N}$)

Andere Methode: Multiplikationsmethode (siehe Cormen et al, Kap. 11.3).

Tabellenvergrößerung

- Wissen nicht a priori, wie gross n sein wird.
- Benötigen $m = \Theta(n)$ zu jeder Zeit.

Grösse der Tabelle muss angepasst werden. Hash-Funktion ändert sich \Rightarrow

Rehashing

- Alloziere Array A' mit Grösse $m' > m$
- Füge jeden Eintrag von A erneut in A' ein (mit erneutem Hashing)
- Setze $A \leftarrow A'$.
- Kosten: $\mathcal{O}(n + m + m')$.

Wie wählt man m' ?

Tabellenvergrößerung

- 1.Idee $n = m \Rightarrow m' \leftarrow m + 1$

Bei jedem Einfügen vergrößern. Kosten $\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2)$ 😞

- 2.Idee $n = m \Rightarrow m' \leftarrow 2m$ Vergrößern nur wenn $m = 2^i$:

$$\Theta(1 + 2 + 4 + 8 + \dots + n) = \Theta(n)$$

Einige Einfügeoperationen kosten lineare Zeit, aber im Durchschnitt kosten sie $\Theta(1)$ 😊

Jede Operation vom Hashing mit Verketteten hat erwartet amortisierte Kosten $\Theta(1)$.

(\Rightarrow Amortisierte Analyse)

Amortisierte Analyse

Generelles Vorgehen bei dynamischen Arrays (z.B. Java: **ArrayList**, Python: **List**)

- Die Datenstruktur speichert neben dem Datenarray zwei Größen: Grösse des Arrays (Kapazität m) und Anzahl verwendete Einträge (Grösse n).
- Grösse verdoppeln und Einträge kopieren, wenn die Liste voll ist $n = m \Rightarrow m \leftarrow 2n$. Kosten $\Theta(m)$.
- Laufzeitkosten bei $n = 2^k$ Einfügeoperationen:
 $\Theta(1 + 2 + 4 + 8 + \dots + 2^k) = \Theta(2^{k+1} - 1) = \Theta(n)$.

Kosten pro Operation **gemittelt über alle Operationen** = **amortisierte Kosten** = $\Theta(1)$ pro Einfügeoperation.

Offene Adressierung

Speichere die Überläufer direkt in der Hashtabelle mit einer **Sondierfunktion** $s : \mathcal{K} \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$
Tabellenposition des Schlüssels entlang der **Sondierungsfolge**

$$S(k) := (s(k, 0), s(k, 1), \dots, s(k, m - 1)) \pmod{m}$$

Sondierungsfolge muss für jedes $k \in \mathcal{K}$ eine Permutation sein von $\{0, 1, \dots, m - 1\}$

Begriffsklärung: Dieses Verfahren nutzt **offene Adressierung** (Positionen in der Hashtabelle nicht fixiert), ist aber **ein geschlossenes Hashverfahren** (Einträge bleiben in der Hashtabelle)

Algorithmen zur offenen Addressierung

- **insert**(i) Suche Schlüssel k von i in der Tabelle gemäss Sondierungssequenz $S(k)$. Ist k nicht vorhanden, füge k an die erste freie Position in der Sondierungsfolge ein. Andernfalls Fehlermeldung.
- **find**(k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib die zu k gehörenden Daten zurück. Andernfalls Rückgabe eines leeres Elements **null**.
- **delete**(k) Suche k in der Tabelle gemäss $S(k)$. Wenn k gefunden, ersetze k durch den speziellen Schlüssel **removed**.

Lineares Sondieren

$$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \dots, h(k) + m - 1) \pmod{m}$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod{m}.$

Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

Diskussion

Beispiel $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 200 Tabelleneinträge! (hier ohne Herleitung).

Grund für die schlechte Performance?

Primäre Häufung: Ähnliche Hashadressen haben ähnliche Sondierungsfolgen \Rightarrow lange zusammenhängende belegte Bereiche.

Quadratisches Sondieren

$$s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod m$$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod m.$

Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
19	15	2		5	12	55

Diskussion

Beispiel $\alpha = 0.95$

Erfolgreiche Suche betrachtet im Durchschnitt 22 Tabelleneinträge (hier ohne Herleitung)

Grund für die schlechte Performance?

Sekundäre Häufung: Synonyme k und k' (mit $h(k) = h(k')$) durchlaufen dieselbe Sondierungsfolge.

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m - 1)h'(k)) \pmod m$

$m = 7, \mathcal{K} = \{0, \dots, 500\}, h(k) = k \pmod 7, h'(k) = 1 + k \pmod 5.$

Schlüssel 12, 55, 5, 15, 2, 19

0	1	2	3	4	5	6
5	15	2	19		12	55

Note: Red dots are present below the values 12 and 55 in the original image.

Double Hashing

- Sondierungsreihenfolge muss Permutation aller Hashadressen bilden. Also $h'(k) \neq 0$ und $h'(k)$ darf m nicht teilen, z.B. garantiert mit m prim.
- h' sollte möglichst unabhängig von h sein (Vermeidung sekundärer Häufung).

Unabhängigkeit weitgehend erfüllt von $h(k) = k \bmod m$ und $h'(k) = 1 + k \bmod (m - 2)$ (m prim).

Gleichmässiges Hashing

Starke Annahme: Die Sondierungssequenz $S(k)$ eines Schlüssels k ist mit gleicher Wahrscheinlichkeit eine der $m!$ vielen Permutationssequenzen von $\{0, 1, \dots, m - 1\}$.

(Double Hashing kommt dem am ehesten nahe)

Analyse gleichmässiges Hashing mit offener Addressierung

Theorem 6

Sei eine Hashtabelle mit offener Addressierung gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. Unter der Annahme vom gleichmässigen Hashing hat die nächste Operation erwartete Laufzeitkosten von $\leq \frac{1}{1-\alpha}$.

Ohne Beweis, siehe z.B. Cormen et al, Kap. 11.4