Vorlesungsmitschrieb

Algorithmen und Berechenbarkeit

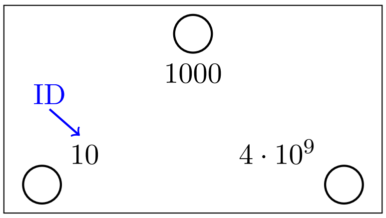
Vorlesung 07

Letztes Update: 2017/11/15 - 12:29 Uhr

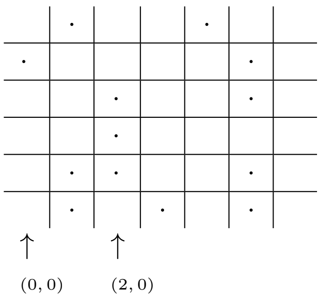
Hashing

Einleitung

Bei der Implementierung des Algorithmus vom Closest-Pair Problem wird überlicherweise auf Hashing zurückgegriffen. Hashing beschreibt dabei eine Funktion bzw. Abbildung, *die eine große Eingabemenge (die Schlüssel) auf eine kleinere Zielmenge (die Hashwerte) abbildet*[1].
Auch bei Telefon- und Wörterbüchern (vgl. dazu *Wörterbuchproblem*) oder für Anwendungen, die OpenStreetMap-Daten verwenden, wird oft auf Hashing zurückgegriffen.



Beim Closest-Pair-Algorithmus kann mittels Hashing die Gitterzelle errechnet werden. Jede Gitterzelle erhält Koordinaten.



Die Maschenweite des Gitters sei w. Dann fällt ein Punkt $P(p_x, p_y)$ in die Gitterzelle

$$\left(\left\lfloor \frac{P_x}{w} \right\rfloor, \left\lfloor \frac{P_y}{w} \right\rfloor\right)$$

abgerundet

[1] `https://de.wikipedia.org/wiki/Hashfunktion`

Beispiel

Sie $w = \frac{1}{2}$ und $P(p_x, p_y) = \left(\frac{15}{10}, \frac{7}{10}\right)$. Man erhält für das Gitter die Koordinaten

$$\left(\left\lfloor\frac{P_x}{w}\right\rfloor, \left\lfloor\frac{P_y}{w}\right\rfloor\right) = \left(\left\lfloor\frac{\frac{15}{10}}{\frac{1}{2}}\right\rfloor, \left\lfloor\frac{\frac{7}{10}}{\frac{1}{2}}\right\rfloor\right) = (3, 1)$$

Diese Koordinaten müssen noch gehasht werden.

Hashing formal

Gegeben sei **erstens** das *Universum* U, das immer sehr groß gewählt wird und typischerweise eine Teilmenge der natürlichen Zahlen darstellt, **zweitens** sei auch die vergleichsweise kleine Menge $S \subseteq U$ gegeben. Daraus folgt **drittens** $n = |S|$, also die Anzahl der Elemente in S.
Das Ziel ist nun

$$\text{Finde } h : U \to \{0, 1, \ldots, m-1\}$$
$$\text{sodass } \forall\ 0 \leq i < m : \left|\ \{x \in S \mid h(x) = i\}\ \right| \leq \left\lceil\frac{n}{2}\right\rceil$$

Man sucht also eine Funktion h, sodass keine zwei Elemente aus S auf dieselbe Zahl zeigen.

Beispiel

Seien $U = \mathbb{N}, S = \{1, 7, 23, 99\}, n = 4$ und $m = 5$ gegeben. Dann ist die Funktion

$$h(x) = x \bmod 5$$

eine sehr gute Hashfunktion für S.

- $\rightarrow h(1) = 1$
- $\rightarrow h(7) = 2$
- $\rightarrow h(23) = 3$
- $\rightarrow h(99) = 4$

Für die Menge $S' = \{2, 17, 22, 32\}$ ist $h(x)$ aber eine sehr schlechte Hashfunktion.

- $\rightarrow h(2) = 2$
- $\rightarrow h(17) = 2$
- $\rightarrow h(22) = 2$
- $\rightarrow h(32) = 2$

Satz: Seien U, m und h gegeben, und sei $k = |U|$ sowie $n = |S| \Rightarrow S \in \binom{U}{n}$. Dann gibt es für jedes n mit $1 \leq n \leq \frac{k}{m}$ ein S, sodass alle Elemente aus S von h auf denselben Wert in $\{0, 1, \cdots, m-1\}$ abgebildet werden.

Beweis: Nach Schubfachsystem existiert ein i mit $0 \leq i < m$, sodass $\underbrace{|k^{-1}(i)|}_{x \in U \ \mid\ h(x) = i} \supseteq \frac{|U|}{m} = \frac{k}{m}$.

Nach Annahme ist $\frac{k}{m} \geq n$. Man kann nun $S \subseteq n^{-1}(i)$ mit $|S| = n$ wählen für ein beliebiges i mit $k^{-1}(i) \geq \frac{U}{m}$.
Um eine gute Hashfunktion zu finden, müssen also immer auch die Daten betrachtet werden, die mit dieser Funktion gehasht werden.

Datenstruktur

Man betrachte die Datenstruktur *Wörterbuch*, die folgende Operationen unterstützt.

`makeset()`	Erzeugt ein leeres Wörterbuch
`insert(x, S)`	Fügt $\underbrace{\text{Item } x}_{\substack{\text{Key +}\\ \text{Information}}}$ in Wörterbuch ein, überschreibt falls vorhanden
`delete(x, S)`	Löscht Item x aus S
`lookup(x, S)`	Gibt Item $x = \underbrace{(x, \text{Info})}_{\text{Paar}}$ aus, falls vorhanden

Man sucht eine Datenstruktur, die Zugriffe in $\mathcal{O}(1)$ erlaubt und dabei nicht mehr als $\mathcal{O}(n)$ Platz verbraucht.

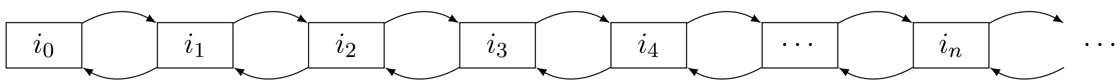
Implementierungsansatz 1: Array für Items, Zähler für $|S|$

i_0	i_1	i_2	i_3	i_4	$\cdots$	i_n

- `makeset()` Diese Operation gelingt in $\mathcal{O}(1)$.
- `insert(x,S)` kostet $\mathcal{O}(1)$, falls bekannt ist, dass noch kein Item mit demselben Schlüssel in S existiert. Ansonsten muss zuerst in $\mathcal{O}(n)$ geprüft werden, ob der Schlüssel bereits enthalten ist, bevor das Item in $\mathcal{O}(1)$ eingefügt werden kann.
- `delete(x,S)` Da nicht mithilfe des Index gelöscht wird, müssen die Einträge von S durchlaufen werden, um das Item zu löschen. Das braucht $\mathcal{O}(n)$.
- `lookup(x,S)` Es müssen wie bei `delete(x,S)` die Einträge durchlaufen werden, um das gesuchte Element zu finden. Das braucht ebenfalls $\mathcal{O}(n)$.

Vorteile	**Nachteile**
• Einfach	• Performance bei `delete(x, S)`
• Platzsparend	• Performance bei `lookup(x, S)`

Implementierungsansatz 2: Einfach/doppelt verkettete Listen



Im Allgemeinen verhält sich diese Datenstruktur für den Wörterbuchansatz recht ähnlich wie das Array aus dem ersten Ansatz.

Implementierungsansatz 3: Direkte Adressierung

Für diese Datenstruktur wird angenommen, dass das Schlüsseluniversum endlich und nicht zu groß ist:

$$U := \{0, 1, 2, \cdots, n-1\}$$

Man legt nun wie im ersten Ansatz ein Array an, diesmal mit der Größe k. An Position i steht die Information für Schlüssel i, falls eine Information für diesen Schlüssel abgelegt wurde. Ansonsten erhält man `NIL`.

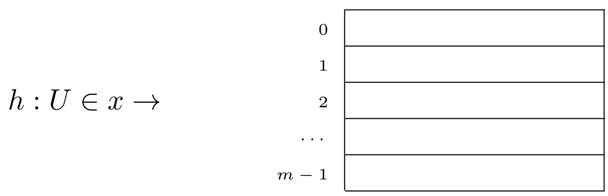
Vorteil	**Nachteil**
• Alle Operationen bis auf `makeset()` in $\mathcal{O}(1)$	• Platz und Größe des Schlüsseluniversums (nicht der Menge S)

Implementierungsansatz 4: Suchstrukturen wie (2,3,4)-, AvL- oder RS-Bäume

Vorteil	**Nachteil**
• Platzverbrauch tatsächlich $\mathcal{O}(n)$	• Zugriffszeit nur in $\mathcal{O}(\log(n))$

Hashing mit Verkettung

Angenommen, ein gewähltes h ist nicht injektiv für S, das bedeutet, es gibt mehrere Elemente in $x, y \in S$ für die gilt $h(x) = h(y)$. Man kann damit dennoch eine Hashdatenstruktur bauen:



Jeder Hasheintrag ist Kopf einer einfach verketteten Liste, $x \in S$ wird in der $h(x)-$ten verketteten Liste gespeichert.

- **Platzbedarf**:
$$\mathcal{O}(m+n) = \mathcal{O}\left(n \cdot \left(1 + \frac{1}{B}\right)\right)$$
$B = \frac{n}{m}$ ist der *Belegungsfaktor*. Je kleiner B, desto ineffizienter ist die Datenstruktur, aber möglicherweise ist es dann einfacher, eine *gute* Hashfunktion zu finden.

- **Zugriffszeit**: Man nimmt an, $h(x)$ kann in $\mathcal{O}(n)$ ausgewertet werden. Dann ist der Zugriff $x \in S$ in
$$\mathcal{O}(1 + \text{Position von } x \text{ in Liste } L_{h(x)})$$
und der Zugriff auf $x \in U \setminus S$ in
$$\mathcal{O}(1 + \text{Länge von } L_{h(x)})$$

Erwartete Suchzeit

Man nimmt an, h verteilt U gleichmäßig über $\{0, 1, 2, \cdots, m-1\}$, dass bedeutet

$$\forall i \in \{0, 1, 2, \cdots, m-1\} : |\{k \in U | h(x) = i\}| \leq \left\lceil \frac{|U|}{m} \right\rceil$$

Zum Beispiel $h(x) = x \bmod m$

Satz: Sei x ein zufälliges (gleichverteiltes) Element aus $U \backslash S$ und $h \leq \frac{|U|}{2}$. Die erwartete Suchzeit nach Element x ist dann $\mathcal{O}(1+B)$

Beweis (erster Teil): Sei l_i die Anzahl der Elemente aus S, die $i - L_i$ gespeichert wurde $= |L_i|$. Es gilt $\sum_{i=0}^{m-1} l_i = n = |S|$. Die erwartete Suchzeit ist damit

$$E := \left(\sum_{i=0}^{m-1} \underbrace{\Pr(h(i) = i)}_{\text{Beweis-Einschub}} \cdot l_i \right) + 1$$

Beweis (Einschub):

$$\begin{aligned}
\Pr(h(x) = i) &= \frac{|U_i \setminus S|}{|U \setminus S|} \\
&\leq \frac{|U_i|}{\underbrace{|U \setminus S|}_{\geq \frac{U}{2}}} \\
&\leq \frac{\left\lceil \frac{|U|}{m} \right\rceil}{\frac{|U|}{2}} \\
&\leq \frac{\frac{|U|}{m} + 1}{\frac{|U|}{2}} \\
&= \frac{2}{m} + \underbrace{\frac{2}{|U|}}_{n \leq \frac{|U|}{2}} \leq \underbrace{\frac{2}{m} + \frac{1}{n}}_{\text{Im zweiten Teil einsetzen}}
\end{aligned}$$

Beweis (zweiter Teil):

$$\begin{aligned}
E &:= \left(\sum_{i=0}^{m-1} \Pr(h(i) = i) \cdot l_i \right) + 1 \\
&\leq \left(\sum_{i=0}^{m-1} \left(\frac{2}{m} + \frac{1}{n} \right) \cdot l_i \right) + 1 \\
&= 1 + \frac{2}{m} \sum_{i=0}^{m-1} l_i + \frac{1}{n} \sum_{i=0}^{m-1} l_i \\
&= 1 + \frac{2n}{m} + 1 \\
&= 2 + \frac{2n}{m} \\
&= \mathcal{O}(1+B)
\end{aligned}$$