Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen | Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

Wörterbücher: Mengen mit eingeschränkten Operationen

Für viele Anwendungen werden nicht alle Operationen auf Mengen benötigt.
Grundlegend wichtig scheinen oft die folgenden Operationen zu sein:
Nachschlagen (Lookup), also die Elementanfrage
Einfügen (Insert)
Löschen (Delete) eines einzelnen Elementes (musste gemäß bisheriger Spezifikation durch andere Mengenoperationen nachgebildet werden (wie genau?))
Initialisieren (init/clear)
Überlegen wir: Wie teuer sind die Operationen bei den drei vorgestellten Implementierungen?

Wörterbücher: Bessere Datenstrukturen

1. Hashing

2. binäre Suchbäume

3. AVL-Bäume

Hashing

Kriterien für eine gute Hash-Funktion:

1. Theoretisch: Surjektivität;
   praktisch: gute Speicherausnutzung, ausgedrückt durch Belegungsfaktor

2. Kollisionsarmut (im Widerstreit zu 1.)

3. Schnelle Berechenbarkeit

4. Die zu speichernden Urschlüssel sollten mit der Hash-Funktion möglichst gleichmäßig über alle Zielschlüssel verteilt werden.

Divisionsmethode

Wie immer: Urschlüssel: $0 \ldots N-1$, Hashwerte: $0 \ldots m-1$.
Hashfunktion: $h(k) = k \bmod m$.
— m sollte keine Zweierpotenz sein, da sonst $h(k)$ nicht von allen Bits (von k) abhängt.
— Ist k eine Kodierung eines Strings im 256er System, so bildet h bei $m = 2^p - 1$ zwei Zeichenketten, die sich nur durch eine Zeichenumstellung unterscheiden, auf denselben Wert ab.
— Eine gute Wahl für m ist eine Primzahl, die nicht nahe bei einer Zweierpotenz liegt.
—Bei professionellen Anwendungen empfiehlt sich ein Test mit “realen Daten”.
—Eine Implementierung finden Sie z.B. unter `http://guxx.de/2007/11/11/assoziative-arrays-als-hashmaps-selbst-verwalten/`

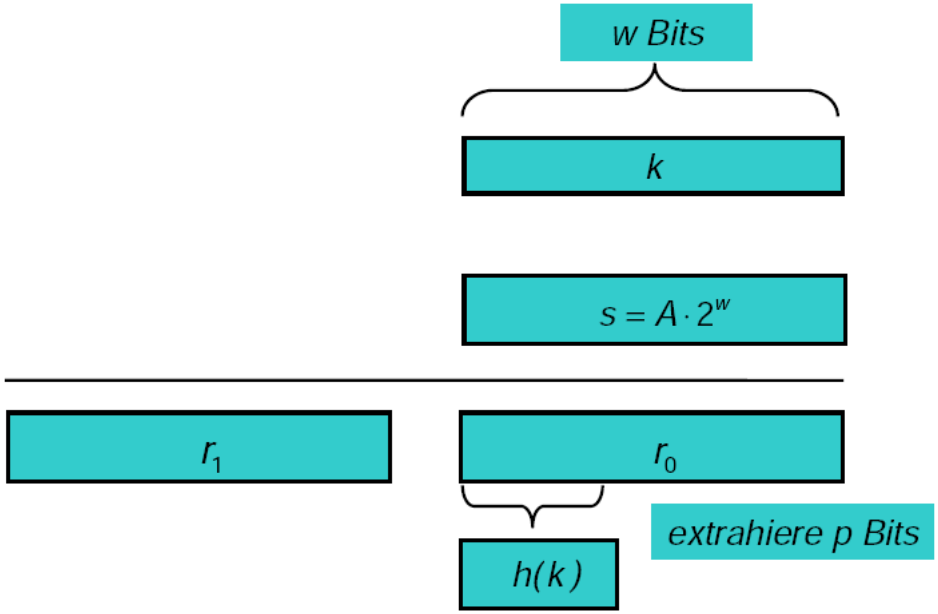
Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m(kA \bmod 1) \rfloor$ für $A \in (0, 1)$.
Hier: $x \bmod 1 =$ "gebrochener Teil von x", z.B.: $\pi \bmod 1 = 0,14159\ldots$.
Rationale Zahlen A mit kleinem Nenner führen zu Ungleichverteilungen, daher empfiehlt sich die Wahl $A = (\sqrt{5} - 1)/2$ *

Vorteile der Multiplikationsmethode:
— Arithmetische Progressionen von Schlüsseln $k = k_0; k_0 + d; k_0 + 2d; k_0 + 3d; \ldots$ werden ebenmäßig verstreut.
— Leicht zu implementieren, wenn $m = 2^p$ (hier unproblematisch) und $N < 2w$, wobei w die Wortlänge ist: Multipliziere k mit $\lfloor A \cdot 2^w \rfloor$. Dies ergibt zwei w-bit Wörter. Vom Niederwertigen der beiden Wörter bilden die p höchstwertigen Bits den Hashwert $h(k)$.

*Die Zahl des *Goldenen Schnitts* kommt in verschiedendsten Bereichen der Algorithmik vor. . . , siehe auch DSL

Multiplikationsmethode - Illustration



Zum Kollisionsproblem

Angenommen, unsere Hashfunktion verteilt die n tatsächlich zu verteilenden Urschlüssel völlig gleichmäßig auf die m Behälter (jedesmal, wenn so eine Zuweisung ansteht), mit $\mathrm{n} < \mathrm{m}$.

Sei $\mathrm{P}(\mathrm{i})$ die Wahrscheinlichkeit dafür, dass der i-te Urschlüssel auf einen freien Behälter abgebildet wird, wenn alle "vorigen" Urschlüssel ebenfalls kollisionsfrei abgebildet wurden.
$\rightsquigarrow$ Wahrscheinlichkeit, dass keine Kollision nach i Zuweisungen erfolgte, beträgt $\mathrm{P}(1) \cdot \mathrm{P}(2) \cdots \mathrm{P}(\mathrm{i})$.
Für $\mathrm{n} = 23$ und $\mathrm{m} = 365$ entspricht das dem *Geburtstagsparadoxon*: Lediglich 23 zufällig ausgewählte Personen genügen, um mit Wahrscheinlichkeit $> 0,5$ zu gewährleisten, dass zwei von ihnen am selben Tag Geburtstag haben.
Für uns bedeutet es: Kollisionsfreiheit ist nicht völlig zu umgehen.

Zum Umgang mit dem Kollisionsproblem

(1) *Hashing mit Verkettung* (auch *offenes Hashing* genannt):
Jeder Behälter wird durch eine beliebig erweiterbare Liste dargestellt.
Die Hashtabelle enthält daher Listenköpfe.

(2) *Hashing mit offener Adressierung*: Neben der Hashfunktion $\mathrm{h} = \mathrm{h}_0$ werden noch weitere Hashfunktionen h_i definiert, die eine Kollisionsbehandlung ermöglichen.

Heute. . .

Hashing mit offener Adressierung

(Perfektes Hashing)

Hashing mit offener Adressierung

Voraussetzung: # der Elemente, die in der Hash-Tabelle abgelegt werden sollen, kann im Voraus gut nach oben abgeschätzt werden.
$\rightsquigarrow$ Keine Listen zur Kollisionsvermeidung.
Wenn Anzahl eingefügter Objekte m erreicht, dann sind keine weiteren Einfügungen mehr möglich.

— Hashing mit Kollisionsvermeidung weist Objekt mit gegebenen Schlüssel feste Position in Hashtabelle zu.
— Bei Hashing durch offene Adressierung wird Objekt mit Schlüssel keine feste Position zugewiesen.
— Position der Objekte in der Tabelle ist abhängig von Schlüssel und bereits belegten Positionen in Hashtabelle.
— Für neues Objekt wird erste freie Position gesucht. Dazu wird Hashtabelle nach freier Position durchsucht.

Hashing mit offener Adressierung / **Zur Komplexität der Operationen**:

—Suchen von Objekten in der Regel schneller, da keine Listen linear durchsucht werden müssen.
— Laufzeit für Einfügen nur noch im Durchschnitt $\Theta(1)$.
— Entfernen von Objekten schwierig, deshalb Anwendung von offener Adressierung oft nur, wenn Entfernen nicht benötigt wird.
Probleme bei Entfernen
— Können Felder i mit gelöschten Schlüsseln nicht wieder mit NIL belegen, denn dann wird Suche nach Schlüsseln, bei deren Einfügung Position i getestet wird, fehlerhaft sein.
— Mögliche Lösung ist, Felder gelöschter Schlüssel mit DELETED zu markieren. Aber dann werden Laufzeiten für Einfügen und Löschen nicht mehr nur vom Belegungsfaktor $\beta = n/m$ abhängen. (Warum?)

Hashing mit offener Adressierung: Das allgemeine Schema

— Wir gehen der Einfachheit halber von $K = S$ aus.
— Es wird eine Folge von Hash-Funktionen $h_i : S \to S$ $i = 0, 1, \ldots$ betrachtet.
— Diese werden auch gerne zu einer Abbildung $h(k, i) = h_i(k)$ zusammengefasst.
— Oft ist $h(k, i)$ die Kombination zweier Hash-Funktionen h und g, z.B.:

$$h(k, i) = (h(k) + i \cdot g(k)) \bmod m$$

Hierbei darf g nicht den Wert 0 annehmen. (Warum?)

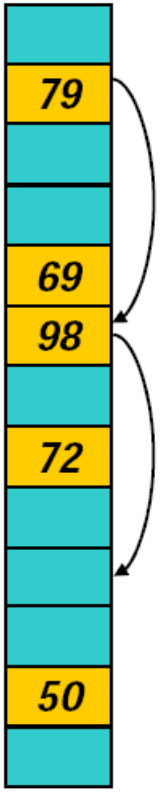
Hashing mit offener Adressierung: Was wir wollen

Wir gehen von einer Hashtabelle T als Feld $T[0..(m-1)]$ aus.
Bei leerer Tabelle gehen wir überall von einem Sonderwert NIL aus, der einen freien Behälter signalisiert.
$(h(k,0), h(k,1), \ldots, h(k,m-1))$ heißt *Testfolge* bei Schlüssel k.
Wir wollen gewährleisten, dass jeder Wert (aus $K = S$) schließlich in der Hashtabelle abgespeichert werden kann (ohne zusätzlichen Aufwand durch Listenverwaltung).
$\rightsquigarrow$ Forderung, dass für alle Schlüssel k die Testfolge bei k eine *Permutation* von S ist, also eine Bijektion definiert.
$\rightsquigarrow$ Wir brauchen $h(k,i)$ nur für $i \in S$ definieren.
Beispiel: Betrachte $h(k,i) = (h(k) + i \cdot g(k)) \bmod m$.

Einfügen bei offener Adressierung

```
Hash-Insert(T,k)
 1  i ← 0
 2  repeat j ← h(k,i)
 3          if T[j] = NIL
 4              then T[j] ← k
 5              else i ← i + 1
 6  until i = m
 7  error "Hashtabelle vollständig gefüllt"
```

Offene Adressierung - Illustration



Lineares Sondieren

Die einfachste Möglichkeit für das Hashing mit offener Adressierung besteht darin, so lange den jeweils (linear) nächsten Behälter zu prüfen, bis man auf einen freien Behälter trifft.

Die Definition der Folge von Hash-Funktionen sieht dann so aus:

$$h_i(x) = (h(x) + i) \bmod m$$

Die Anwendung des Modulo hat mit der begrenzten Zahl von Behältern zu tun: Wurde der letzte Behälter geprüft, so beginnt man wieder beim ersten Behälter.
Problem dieser Methode: schnell bilden sich *Ketten* $\rightsquigarrow$ die Zugriffszeiten im Bereich solcher Ketten steigen.
Das lineare Sondieren ist daher wenig effizient.

Hashing mit offener Adressierung: Der Idealfall

Satz: Gehen wir davon aus, dass jede der m Testfolgen selbst eine zufällige Permutation von $S = [0..m-1]$ darstellt und beträgt der Belegungsfaktor $\beta = n/m < 1$, so betragen die mittleren Kosten für eine Einfüge-Operation (bzw. der erfolglosen Suche)

$$\mathcal{O}\left(\frac{m+1}{m-n+1}\right) = \mathcal{O}\left(\frac{1}{1-\beta}\right).$$

Die mittleren Kosten für eine erfolgreiche Suche betragen

$$\mathcal{O}\left(\frac{1}{\beta} \cdot \log\left(\frac{1}{1-\beta}\right)\right).$$

Hashing mit offener Adressierung: Der Idealfall

$C(n, m)$: mittlere Kosten einer Einfüge-Operation in eine Tafel mit m Positionen, von denen n besetzt sind.
$q_j(n, m)$: Wahrsch., dass Tafelpositionen $h(k, 0), \ldots, h(k, j-1)$ besetzt sind (mit $q_0(n, m) = 1$).
Position $h(k, 0)$ ist in n von m Fällen besetzt.
Ist $h(k, 0)$ besetzt, so ist $h(k, 1)$ in $(n-1)$ von $(m-1)$ Fällen besetzt (denn: $h(k, 1) \neq h(k, 0)$ wegen der Permutationseigenschaft der Testfolgen).
Fortgeführt liefert dieses Argument:

Lemma: $q_j(n, m) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-(j-1)}{m-(j-1)} = \frac{\binom{n}{j}}{\binom{m}{j}}$.

Hashing mit offener Adressierung: Der Idealfall

$C(n, m)$: mittlere Kosten einer Einfüge-Operation in eine Tafel mit m Positionen, von denen n besetzt sind.
Betrachte ZV $Z(n, m)$: # Schritte, bis Einfügeposition in Tafel gefunden wurde
$\rightsquigarrow Z(n, m)(k) = \min\{i \mid h(k, i)$ ist unbesetzt $\}$.
$\rightsquigarrow C(n, m) = \sum j \cdot P[Z(n, m) = j]$.
Beobachte: $P[Z(n, m) > j] = q_j(n, m)$.
Der hilfreiche Rechentrick aus der vorigen Vorlesung liefert daher:
Lemma: $C(n, m) = \sum_{j=0}^{n} q_j(n, m)$.

Hashing mit offener Adressierung: Der Idealfall

Lemma: $C(n,m) = \frac{m+1}{m-n+1}$

Beweis: Dies folgt aus den beiden vorigen Lemmata durch *geschachtelte Induktion*: "außen" über m, und für festes m dann über n.

(A) $C(0,m) = q_0(0,m) = 1$. (B) $q_j(n,m) = \frac{n}{m} \cdot q_{j-1}(n-1,m-1)$.

$\rightsquigarrow C(n,m) = \sum_{j=0}^{n} q_j(n,m) = 1 + \frac{n}{m} \sum_{j=0}^{n-1} q_j(n-1,m-1) = 1 + \frac{n}{m} \cdot C(n-1,m-1)$

Aus dieser rekursiven Darstellung folgt mit den beschriebenen Induktionsvorgängen leicht die Behauptung:

$1 + \frac{n}{m} \cdot C(n-1,m-1) = 1 + \frac{n}{m} \cdot \frac{m}{m-n+1} = \frac{m-n+1}{m-n+1} + \frac{n}{m-n+1} = \frac{m+1}{m-n+1}$

Hashing mit offener Adressierung: Die Suche im Idealfall
Soll k in T gesucht werden, so durchlaufen wir die Folge $h(k,0),\ldots,h(k,i)$ bis so einem i, für das $T[h(k,i)] = k$ gilt.
Dieses i hat auch die Kosten bestimmt, um k einzufügen.
Die Kosten einer erfolgreichen Suche in einer Tabelle mit Belegungsfaktor n/m sind also:

$$C(n) := \frac{1}{n}\sum_{i=0}^{n-1} C(i,m) = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m+1}{m-i+1} = \frac{m+1}{n}\left[\sum_{j=1}^{m+1}\frac{1}{j} - \sum_{j=1}^{m-n+1}\frac{1}{j}\right] = \frac{m+1}{n}\left[H_{m+1} - H_{m-n+1}\right].$$

Wir wissen, das die Harmonische Zahl H_r das diskrete Gegenstück zum Logarithmus darstelle. Daher gilt:

$$\frac{1}{n}\sum_{i=0}^{n-1} C(i,m) = \frac{m+1}{n}\left[H_{m+1} - H_{m-n+1}\right] \approx \frac{m+1}{n}\left[\ln(m+1) - \ln(m-n+1)\right]$$

$$\frac{1}{n}\sum_{i=0}^{n-1} C(i,m) \approx \frac{1}{\beta}\ln\left(\frac{1}{\frac{m-n+1}{m+1}}\right) \approx \frac{1}{\beta}\ln\left(\frac{1}{1-\beta}\right).$$

Hashing mit offener Adressierung: Lineares Sondieren

Satz: Beträgt der Belegungsfaktor $\beta = n/m < 1$, so sind die mittleren Kosten für eine Einfüge-Operation (bzw. der erfolglosen Suche) beim linearen Sondieren

$$\mathrm{LIN}(n, m) = \frac{1}{2}\left(1 + \frac{1}{1-\beta}\right).$$

Die mittleren Kosten für eine erfolgreiche Suche betragen

$$\mathrm{LIN}(n) = \frac{1}{2}\left(1 + \frac{1}{(1-\beta)^2}\right).$$

Hashing mit offener Adressierung Die Faktoren mit und ohne Kettenbildung

Belastungsfaktor β	$C(n,m)$	$LIN(n,m)$	$C(n)$	$LIN(n)$
$0,20$	$1,25$	$1,28$	$1,12$	$1,125$
$0,50$	2	$2,5$	$1,38$	$1,5$
$0,80$	5	13	$2,01$	3
$0,90$	10	$50,5$	$2,55$	$5,5$
$0,95$	20	$200,5$	$3,15$	$10,5$

Für hohe Belegungsfaktoren ist lineares Sondieren daher ungeeignet.

Praktisches zu Hashing in JAVA

Welche Hash-Verfahren tatsächlich angewendet werden, bleibt dem Anwender / Programmierer oft verborgen, siehe auch die entsprechende Beschreibung der Hash-Klasse von JAVA unter `http://www.addison-wesley.de/Service/Krueger/kap12003.htm#E15E128`

Wer mehr zu Hashing in JAVA erfahren will, sei auf die schöne frei zugängliche JAVA-Einführung `http://www.rz.uni-hohenheim.de/anw/programme/prg/java/tutorials/javainsel4/javainsel_11_005.htm` verwiesen.