Hashing

Hashing

Problem:
Eine (kleine) Teilmenge eines (großen) Universums U
soll in einem (kleinen) Array T mit Indizes $\{0, \ldots, m-1\}$
abgespeichert werden.

Typischerweise ist $U = \{0, \ldots, N-1\}$ oder
$U =$ Menge von Strings (über einer Menge von Zeichen).

Idee:

> Nimm eine Funktion h, die das Universum U irgendwie
> auf die Indizes $\{0, \ldots, m-1\}$ abbildet.
>
> Trage jedes Element $x \in U$ in $T[h(x)]$ ein.

Hashing: Kollisionen

Problem:

> Die „Hashfunktion“ h ist notwendigerweise nicht-injektiv:
> es existieren Elemente $x, y \in U$, so daß $x \neq y$, $h(x) = h(y)$.
>
> Was tun, wenn x an einer Stelle eingetragen werden soll,
> die schon von y besetzt ist („Kollision“)?

Hashing: Kollisionen

Ein möglicher Ausweg: Verkettung.

> $T[i]$ enthält eine verkettete Liste von Elementen aus U;
> in der Liste werden alle Elemente x abgespeichert,
> für die $h(x) = i$ ist.

Hashing: Kollisionen

Ein anderer Ausweg: Offene Adressierung.

Falls das Element $x \in U$ eingetragen werden soll
und $T[h(x)]$ schon von einem anderen Element besetzt ist,
probiere es an einer anderen Stelle:

z.B.: linear weitersuchen.
Probiere $h(x)$, $(h(x) + 1) \bmod m$, $(h(x) + 2) \bmod m$, ...
($\rightsquigarrow$ Cluster $\rightsquigarrow$ Ineffizienz)

z.B.: doppeltes Hashing.
Sei h' eine weitere Hashfunktion, dann probiere
$h(x)$, $(h(x) + h'(x)) \bmod m$, $(h(x) + 2h'(x)) \bmod m$, ...
($\rightsquigarrow$ Gefahr von Clustern verringert)

Hashing: Kollisionen

Vergleich: Verkettung/Offene Adressierung

Offene Adressierung hat einfachere Datenstruktur, aber:

Clusterbildung ist möglich.

Löschen ist schwierig.

Tabelle darf nicht zu voll werden.
(Anzahl der Einträge muß *deutlich* kleiner als m sein.)

Inkrement und Tabellengröße m *müssen* teilerfremd sein,
z.B. m prim.
(Sonst besteht die Gefahr, daß kein freier Platz in T
gefunden wird, obwohl noch freie Plätze existieren.)

Gute Hashfunktionen

Forderung an Hashfunktion:

gleichmäßig streuend:
für jedes $i \in \{0, \ldots, m-1\}$ existieren ungefähr gleich viele Elemente des Universums U, die auf i abgebildet werden.
(für Universum $U = \{0, \ldots, N-1\}$ ideal: auf jedes i werden zwischen $\lfloor N/m \rfloor$ und $\lceil N/m \rceil$ Elemente aus U abgebildet.)

Gegenbeispiel:

Die Funktion $\lfloor \sqrt(x) \rfloor$ bildet $\{0, \ldots, 99\}$ auf $\{0, \ldots, 9\}$ ab.
Dabei werden 19 Zahlen auf 9, aber nur eine Zahl auf 0
und nur drei Zahlen auf 1 abgebildet.

Gute Hashfunktionen

Übliche Beispiele:

Für Zahlen:

$h(x) = ((a\,x + b) \bmod N) \bmod m$,
$a, b \in \{1, \ldots, N-1\}$, N prim.

$h(x) = \lfloor m \cdot (c\,x - \lfloor c\,x \rfloor) \rfloor$, c reell, $0 < c < 1$
(häufig: $c = (\sqrt{5} - 1)/2 = 0{,}6180339887$)

Für Strings:

Betrachte x als Tupel $(x_r, \ldots, x_0)$,
$x_i \in \{0, \ldots, m-1\}$, m prim.
$h(x) = \sum a_i x_i \bmod m$.

Effizienz

Auslastung einer Hashtabelle:

$\beta = n/m$
(n = Anzahl der bisherigen Einträge, m = Größe der Tabelle)

Effizienz der Zugriffsoperationen (Hashing mit Verkettung):

Suchen, Einfügen, und Löschen eines Elements brauchen durchschnittlich $O(1 + n/m) = O(1 + \beta)$ Zeit (unter der Veraussetzung, daß alle Elemente aus U gleiche Wahrscheinlichkeit haben).

aber: worst-case-Zeitbedarf = $O(n)$.

Effizienz

Folgerung:

β beschränken, z.B. $\beta < 0{,}9$ oder $\beta < 2$.
(bei offener Adressierung: Schranke muß kleiner als 1 sein!)

Dann gilt: durchschnittliche Zugriffszeit ist $O(1)$.

Falls β zu groß wird:
Tabelle vergrößern,
neue Hashfunktion nehmen (für die größere Tabelle),
alte Werte umkopieren
(wie bei erweiterbaren Arrays: Größe ungefähr verdoppeln)

Theorie und Praxis

In realen Anwendungen haben nicht alle Eingabewerte die gleiche Wahrscheinlichkeit.

Beispiel:

Universum: zulässige Bezeichner in einer Programmiersprache.

$h(x) = \sum x_i \bmod m$.

$\rightsquigarrow$ die Namen `i2`, `j1`, `k0` werden auf denselben Index abgebildet.

Beispiel:

Universum: Personennamen (`"Nachname, Vorname"`).

h benutzt nur die ersten acht Zeichen zur Berechnung.

$\rightsquigarrow$ alle `"Schmidt, ..."` werden auf denselben Index abgebildet.

Perfektes Hashing

In manchen Anwendungen ist genau bekannt, welche Elemente aus dem Universum tatsächlich vorkommen
(z.B.: reservierte Schlüsselwörter einer Programmiersprache).

Dann kann eine optimale Hashfunktion h berechnet werden, die genau für diese Elemente garantiert keine Kollision produziert.

Datenstrukturen im Vergleich

Datenstrukturen im Vergleich

Bisher behandelte Datenstrukturen (Auswahl):

- doppeltverkettete unsortierte Liste
- unsortiertes erweiterbares Array
- sortiertes Array
- balancierter Suchbaum
- Hashtabelle mit Verkettung

Datenstrukturen im Vergleich

Kriterium: Reihenfolge der Einträge.

- durch Anwender kontrollierbar
- durch Datenstruktur vorgegeben (sortiert)
- durch Datenstruktur vorgegeben (unsystematisch)

Datenstrukturen im Vergleich

Kriterium: Zeitbedarf.

- Suchen
- Einfügen
  - am Anfang
  - am Ende
  - an der aktuellen Position
    (während die Datenstruktur durchlaufen wird)
  - an der durch die Datenstruktur vorgegebenen Position
- Löschen (eines bereits gefundenen Elements)
- Zugriff auf erstes/letztes Element
- Zugriff auf i-tes Element

Datenstrukturen im Vergleich

Doppeltverkettete unsortierte Liste:

- Reihenfolge der Einträge: durch Anwender kontrollierbar.
- Suchen: $O(n)$
- Einfügen am Anfang: $O(1)$
- Einfügen am Ende: $O(1)$
- Einfügen an der aktuellen Position: $O(1)$
- Löschen: $O(1)$
- Zugriff auf erstes/letztes Element: $O(1)$
- Zugriff auf i-tes Element: $O(n)$

Datenstrukturen im Vergleich

Unsortiertes erweiterbares Array:

- Reihenfolge der Einträge: durch Anwender kontrollierbar.
- Suchen: $O(n)$
- Einfügen am Anfang: $O(n)$
- Einfügen am Ende: amortisiert $O(1)$, worst-case $O(n)$
- Einfügen an der aktuellen Position: $O(n)$
- Löschen: $O(n)$, am Ende: $O(1)$
- Zugriff auf erstes/letztes Element: $O(1)$
- Zugriff auf i-tes Element: $O(1)$

Datenstrukturen im Vergleich

Sortiertes Array:

- Reihenfolge der Einträge:
  - durch Datenstruktur vorgegeben (sortiert).
- Suchen: $O(\log n)$
- Einfügen: $O(n)$
- Löschen: $O(n)$
- Zugriff auf erstes/letztes Element: $O(1)$
- Zugriff auf i-tes Element: $O(1)$

Datenstrukturen im Vergleich

Balancierter Suchbaum:

- Reihenfolge der Einträge:
  - durch Datenstruktur vorgegeben (sortiert).
- Suchen: $O(\log n)$
- Einfügen: $O(\log n)$
- Löschen: $O(\log n)$
- Zugriff auf erstes/letztes Element: $O(\log n)$
- Zugriff auf i-tes Element: $O(\log n)$ (wie geht das?)

Datenstrukturen im Vergleich

Hashtabelle:

Reihenfolge der Einträge:
durch Datenstruktur vorgegeben (unsystematisch).
Suchen: im Durchschnitt $O(1)$, worst-case $O(n)$
Einfügen: im Durchschnitt $O(1)$, worst-case $O(n)$
Löschen: im Durchschnitt $O(1)$, worst-case $O(n)$
(bei Verkettung)

Datenstrukturen kombiniert

Beispiel:

Liste der E-Mail-Adressen in einem Mailprogramm.

gewünscht:
Suchen, Einfügen, Löschen: schnell.
Reihenfolge der Einträge: durch Anwender kontrollierbar.

Lösung: kombiniere Hashing und doppeltverkettete Liste.