Datenstrukturen – Hashing

by

Dr. Günter Kolousek

Hashing – Überblick

- Hashverfahren: Basis für Dictionarys und Mengen!
- Beobachtung: nur Teilmenge der möglichen Schlüssel gespeichert!
- Idee: Finden durch **Berechnung** anstatt durch Schlüsselvergleiche
- Verwendet wird: Array mit Index $0, \ldots, m-1$
- Hashfunktion h
  $h : K \to \{0, \ldots, m-1\}$
  ordnet jedem Schlüssel $k \in K$ einen Index $h(k)$ mit $0 \leq h(k) \leq m-1$ zu.
- $\leadsto$ Hashtabelle, Dictionary, Assoziatives Array, Streuspeicherung

Anforderungen an Hashfunktion

- gleichmäßige Verteilung, um Adresskollisionen zu vermeiden
  - hashing …'zerhacken'
  - auch wenn Schlüssel nicht gleichmäßig verteilt sind. z.B. Variablenamen: i1, i2, i3,…
- Surjektivität: Alle möglichen Hashwerte sollen auch durch Hashfunktion auch errechnet werden können
  - d.h. $|K| \geq m$
- schnell und einfach berechenbar

Beispiele für Hashfunktionen

- Annahme: Schlüssel k ist binär
- XOR-Methode (einfach)
  - bei Rechnern mit langsamer Division
  - k in Stücken zu je n Bits zerschneiden und diese XOR verknüpfen
    - Das Resultat – als Dualzahl interpretiert – liegt in $[0, 2^n - 1]$
- Multiplikationsmethode
  - Multiplikation des Schlüssels mit Konstante $A \in (0, 1)$
  - Multiplikation des gebrochenen Rests der vorhergehenden Multiplikation mit m und abrunden
  - d.h.: $h(k) = \lfloor m((k \cdot A) \bmod 1) \rfloor = \lfloor m(k \cdot A) - \lfloor k \cdot A \rfloor \rfloor$
    - mit z.B.: $A = \frac{\sqrt{5}-1}{2}$ (lt. Knuth)
    - $\lfloor x \rfloor$ ... `floor(x)` (größte ganze Zahl nicht größer als x)
  - Vorteil: Wahl von m beliebig möglich
    - z.B. $m = 2^p$, dann Multiplikation schnell!
- Kongruenzmethode (Divisions-Rest-Methode)
  - siehe folgende Folie

Kongruenzmethode

- auch: Divisions-Rest-Methode
- Interpretation als nicht negative Dualzahl im Intervall $[0, 2^n - 1]$
- $h(k) = k \bmod m$
- Wie sieht m aus?
  - (vorzugsweise) Primzahl
    - aber nicht wenn Primzahl gleich $2^n - 1$ (für beliebiges n) ist
  - Vorsicht bei bestimmten Werten: Bei $m = 2^j$ wäre Index die Dualzahl aus den letzten j Bits der Schlüssel!
- Spezialfall der Multiplikationsmethode mit $A = \frac{1}{m}$

Hashfunktion für Strings

- Schlüssel k besteht aus n Zeichen $z_{n-1}...z_0$
  - z.B. ASCII oder UTF-8,...
- basierend auf Kongruenzmethode
- Jedes Zeichen (Annahme: Byte) interpretiert man als eine Stelle einer Zahl $z_{n-1}...z_0$ im Zahlensystem zur Basis 256 und definiert

$$h(k) = (\sum_{i=0}^{n-1} \text{ord}(z_i) \cdot 256^i) \bmod m$$

  mit einer geeigneten Primzahl m.
- String abgespeichert als Bytes $b_0...b_{n-1}$ (d.h. 1 Zeichen $\equiv$ 1 oder mehrere Bytes). Damit erfolgt die Berechnung als:

$$h(k) = (\sum_{i=0}^{n-1} \text{ord}(b_{n-1-i}) \cdot 256^i) \bmod m$$

Hashfunktion für Strings – 2

- Multiplizieren? potenzieren? ($\rightsquigarrow \cdot 256^i$)

Hashfunktion für Strings – 2

- Multiplizieren? potenzieren? ($\rightsquigarrow \cdot 256^i$)
  - $\rightarrow$ verschieben

$$h(k) = (\sum_{i=0}^{n-1} \text{ord}(b_{n-1-i}) << (8 \cdot i)) \bmod m$$

- Multiplizieren mit 8?

Hashfunktion für Strings – 2

- Multiplizieren? potenzieren? ($\rightsquigarrow \cdot 256^i$)
  - $\rightarrow$ verschieben

$$h(k) = (\sum_{i=0}^{n-1} \text{ord}(b_{n-1-i}) << (8 \cdot i)) \bmod m$$

- Multiplizieren mit 8?
  - $\rightarrow$ Hornerschema!

$$h(k) = (\sum_{i=0}^{n-1} \text{ord}(b_{n-1-i}) << (8 \cdot i)) \bmod m =$$
$$= (\text{ord}(b_{n-1}) + (\text{ord}(b_{n-2}) + (\text{ord}(b_{n-3}) +$$
$$+ (...) << 8) << 8) << 8) \bmod m$$

Hashfunktion für Strings – 3

- Als Algorithmus?

Hashfunktion für Strings – 3

- Als Algorithmus?

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = (val << 8) + ord(s[i])
    return val % m
```

  - nur mehr Verschiebeoperation und Addition!

Hashfunktion für Strings – 4

- Gibt es hier ein Problem?

Hashfunktion für Strings – 4

- Gibt es hier ein Problem?

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = (val << 8) + ord(s[i])
    print(val)
    return val % m

hash_str_horner("value_or_not_value?", 163)
# liefert 52 als Rückgabe ...aber Ausgabe?
```

Hashfunktion für Strings – 4

- Gibt es hier ein Problem?

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = (val << 8) + ord(s[i])
    print(val)
    return val % m

hash_str_horner(”value_or_not_value?”, 163)
# liefert 52 als Rückgabe ...aber Ausgabe?
```

26399747317036548841622126199245956521481598 07

Hashfunktion für Strings – 4

- Gibt es hier ein Problem?

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = (val << 8) + ord(s[i])
    print(val)
    return val % m

hash_str_horner("value_or_not_value?", 163)
# liefert 52 als Rückgabe ...aber Ausgabe?
```

263997473170365488416221261992459565214815980 7

  - zu groß für eine 32 Bit unsigned Zahl!

Hashfunktion für Strings – 4

- Gibt es hier ein Problem?

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = (val << 8) + ord(s[i])
    print(val)
    return val % m

hash_str_horner("value_or_not_value?", 163)
# liefert 52 als Rückgabe ...aber Ausgabe?
```

26399747317036548841622126199245956521481598 07

  - zu groß für eine 32 Bit unsigned Zahl!
    - 4294967295

Hashfunktion für Strings – 4

- Gibt es hier ein Problem?

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = (val << 8) + ord(s[i])
    print(val)
    return val % m

hash_str_horner("value_or_not_value?", 163)
# liefert 52 als Rückgabe ...aber Ausgabe?
```

26399747317036548841622126199245956521481598 07

  - zu groß für eine 32 Bit unsigned Zahl!
    - 4294967295
  - **auch** zu groß für eine 64 Bit unsigned Zahl!!!
    - 18446744073709551615

Hashfunktion für Strings – 5

- Ein bisschen Mathematik gefällig?

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$$

$$(a \cdot b) \bmod m = (a \bmod m \cdot b \bmod m) \bmod m$$

- In Algorithmus einbauen:

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = ((val << 8) % m + ord(s[i]) % m) % m
        print(val, end=”,”)
    return val

hash_str_horner(”value_or_not_value?”, 163)
# wieder 52 als Rückgabe ...aber Ausgabe?
```

Hashfunktion für Strings – 5

- Ein bisschen Mathematik gefällig?

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$$

$$(a \cdot b) \bmod m = (a \bmod m \cdot b \bmod m) \bmod m$$

- In Algorithmus einbauen:

```
def hash_str_horner(s, m):
    val = 0
    for i in range(len(s)):
        val = ((val << 8) % m + ord(s[i]) % m) % m
        print(val, end=”,”)
    return val

hash_str_horner(”value_or_not_value?”, 163)
# wieder 52 als Rückgabe ...aber Ausgabe?
```

118,150,40,88,135,99,27,17,46,150,43,40,66,62,158,132,5,77,52,52

Hashfunktion für Strings – 6

- Alternativen zur Kongruenzmethode?
  - liefert gute Ergebnisse, aber… oft werden theoretisch weniger abgesicherte, aber performantere und bewährte Funktionen verwendet!

Hashfunktion für Strings – 6

- Alternativen zur Kongruenzmethode?
  - liefert gute Ergebnisse, aber… oft werden theoretisch weniger abgesicherte, aber performantere und bewährte Funktionen verwendet!
- Algorithmus ”djb2” von Dan Bernstein:

```
unsigned long hash(unsigned char* str) {
    unsigned long hash{5381};
    int c;

    while (c = *str++) {
        /*            hash * 33       + c */
        hash = ((hash << 5) + hash) + c;
    }

    return hash;
}
```

Einfügen und Enfernen

- Datensätze … DS
- Hasharray … t
  - hat fixe Größe, speichert DS
- Einfügen
  1. $i = h(k)$ berechnen
  2. Wenn Platz i frei, eintragen in t
  3. Anderenfalls: Kollisionsbehandlung!
     - Offene Hashverfahren
       Eintragen der Überläufer in freien Plätzen
     - Verkettung der Überläufer
       Eintragen der Überläufer in verketteter Liste
- Entfernen: abhängig von gewählter Strategie der Kollisionsbehandlung

Offene Hashverfahren – Allgemeines

- Eintragen der Überläufer in freie Plätze
  - wenn voll, dann neues Array anlegen und im neuem Array neu eintragen; dann altes löschen (Heap!)
- Wenn Platz belegt, dann einen freien Platz (*offene* Stelle) finden und eintragen
- Wie findet man einen neuen freien Platz?
  - *Sondierungsfolge*: Reihenfolge der zu betrachtenden Speicherplätze (d.h. eine Permutation aller Hashadressen).
- Sondierungsfunktion: $s(j, k)$ eine Funktion von j und k, dass
  - $(h(k) - s(j, k))$ mod m

  für $j = 0, 1, \ldots, m - 1$ eine *Sondierungsfolge* bildet.

Offene Hashverfahren – Operationen

- Suchen
  - Beginne mit $i = h(k)$
  - Solange k nicht in $t[i]$ gespeichert ist und $t[i]$ nicht frei ist, suche weiter bei $i = (h(k) - s(j,k)) \bmod m$.
  - Falls $t[i]$ belegt, wurde k gefunden, anderenfalls Suche erfolglos
- Einfügen
  - Beginne mit $i = h(k)$
  - Solange $t[i]$ belegt ist, mache weiter bei $i = (h(k) - s(j,k)) \bmod m$.
  - Trage k in $t[i]$ ein.
- Entfernen
  - problematisch, da nicht aus Sondierierungfolge entfernt werden darf. D.h. wird nur als entfernt markiert:
    - Beim Suchen: wie belegt
    - Beim Einfügen: wie frei

Offene Hashverfahren – Sondieren

- Lineares Sondieren
  - Sondierungsreihenfolge:
    $h(k), h(k)-1, h(k)-2, \ldots, 0, m-1, \ldots, h(k)+1$
  - Sondierungsfunktion: $s(j,k) = j$
- Quadratisches Sondieren (m Primzahl, $m = 4i+3$)
  - Sondierungsreihenfolge:
    $h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \ldots$ jeweils modulo m.
  - Sondierungsfunktion: $s(j,k) = (\text{ceil}(j/2))^2(-1)^j$
- Double Hashing (zweite Hashfunktion)
  - Sondierungsreihenfolge:
    $h(k), h(k)-h'(k), h(k)-2\,h'(k), \ldots, h(k)-(m-1)\,h'(k)$
    jeweils modulo m.
  - Sondierungsfunktion: $s(j,k) = j * h'(k)$

Verkettung der Überläufer

- Separate Verkettung der Überläufer
  - zusätzlich zum DS wird ein Zeiger auf eine verkettete Liste gespeichert, die alle den selben Hashwert aufweisen ($\rightsquigarrow$ Überläufer).
  - Suchen
    1. Beginne bei $i = h(k)$
    2. Wenn DS nicht gefunden, dann verkettete Liste der Überläufer absuchen bis gefunden, oder nicht gefunden.
  - Einfügen (analog zu Suchen und Einfügen in Liste)
  - Entfernen (u.U. erstes Element der Überläuferliste in Hashtabelle eintragen)
- Direkte Verkettung der Überläufer
  - wie separate Verkettung, jedoch werden keine DS direkt in der Hashtabelle gespeichert.