Hashing

Paulus Böhme

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

February 13, 2014

Gliederung

Was ist Hashing

- Abbildung von beliebig langen Werten auf einen Wert mit fester Länge
- Abbildung von Werten auf einen Wert aus einer begrenzten Teilmenge
- Hashfunktion
  - Speichern und Verwalten von Datensätzen mit Hilfe einer Tabelle
  - ein Schlüssel wird zur Tabellenadresse umgewandelt bzw. komprimiert

Hashfunktion

- Abbildung: $h = K \rightarrow S$, $|K| \geq |S|$
- h liefert eine Hashtabelle mit der Größe $|S|$ [1]
- K: Werte die gehasht werden sollen (Schlüssel)
- S: Menge der möglichen Hashwerte
  - $S \in \mathbb{N}$
  - Adressraum: $S \subseteq \{0,...,m\text{-}1\}$
- Quersumme: $25 = 2 + 5 = 7$

[1] Offene Adressierung

Eigenschaften

- k $\neq$ k', h(k) = h(k') $\rightarrow$ Kollision
- wenig Kollisionen (Perfekte Hashfunktion wäre injektiv)
- Chaos
- Surjektivität
- Speicherbedarf des Hashwertes $<$ Eingabewert, Effizienz

Einsatz von Hashing

- Datenbanken/Datenbankindex
  - Suche von großen Daten per Hashtabelle
- Prüfsumme
  - Erkennung von Veränderung an Datensätzen (Übertragungen)
  - Quersumme $\rightarrow$ ISBN10
  - Hashtree
- Kryptographie
  - kollisionsfreie Einwegfunktionen
  - MD5, SHA, etc.

Hashfunktionen

- Divisionsmethode
  - h(k) = k mod m
  - H(19876) = 19876 mod 11 = 10
- Mitquadratmethode
  - Führende und endende Ziffern werden abgeschnitten
  - h(10278436) = 1027 [8] 436 = 8 [1]
- Zerlegungsmethode
  - Zerlegung des Schlüssel, bis gültige Adresse vorhanden ist
  - h(135612) = [13]+[56]+[12]= 81 = [8]+[1] = 9 [1]

[1] Sehr vereinfachtes Beispiel: http://openbook.galileocomputing.de/c_von_a_bis_z/022_c_algorithmen_005.htm

Einleitung Hashtabellen

- Offene Adressierung (geschlossenes Hashing)
  - begrenzte Anzahl an Adressen
  - jede Adresse kann nur an einen Wert vergeben werden

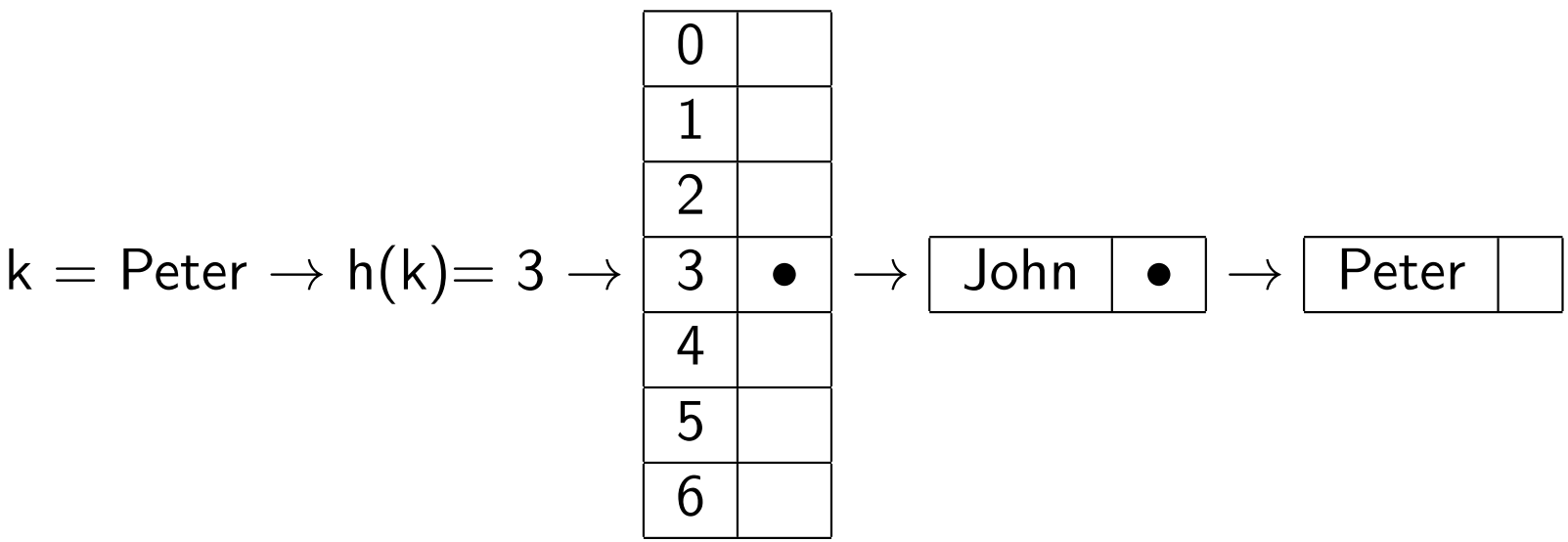
$k = \text{John} \longrightarrow h(k) = 3 \longrightarrow$

0	
1	
2	
3	•
4	
5	
6	

- Kollisionsauflösungsstrategien

Einleitung Hashtabellen (fort.)

- Adressierung mit verketteten Listen
  - jeder Behälter der Tabelle kann eine Verkettete Liste aufnehmen
  - Praxis: Array[i]=Verkettete Liste



Hashfunktion Beispiel

```c
#define MAX_HASH 100

int hashfunction(char *string)
{
      unsigned int hash_add;
      unsigned char *pointer;

      hash_add = 0;
      pointer = (unsigned char *) string;
      while(*pointer != '\0')
      {
          hash_add = 33 * hash_add + *pointer;
          pointer++;
      }
      return hash_add % MAX_HASH;
}
```

Hashtabelle offene Adressierung

```
char hash_table[MAX_HASH][MAX_STRING];

void insertOpAd(char *string)
{
     int hashad = hashFunction(string);
     while(hash_table_open[hashad][0] != 0)
      {
           hashad = (hashad + 1) % MAX_HASH;
      }
      strncpy(hash_tabl[hashad],string,MAX_STRING);
}
```

Hashtabelle mit verketteten Listen

```
struct perso
{
    char vname[255];
    char nname[255];
    int age;
    struct perso *next;
};

typedef struct {
    struct perso *start;
    struct perso *current;
    struct perso *last;
} perso_list;

perso_list *hash_table[MAX_HASH];
```

Hashtabelle: Einfügen

```
void insert(char *nname, char *vname, int age)
{
      int hash_ad = hashFunctionVarTwo(nname);
      perso_list *list = &hash_table[hash_ad];

      if(list->start == NULL){
          list->start = malloc(sizeof(struct perso));
          list->last = list->start;
      } else {
          list->last->next = malloc(sizeof(struct perso
              ));
          list->last = list->last->next;
      }
      if(!list->last) assert("Memmory");
      strcpy(list->last->nname, nname);
      strcpy(list->last->vname, vname);
      list->last->age = age;
}
```

glib

- Hashtabelle:

```
GHashtable hash_table = *g_hash_table_new(g_str_hash, g_str_equal);
```

- Hashfunktionen:
  - `g_direct_hash()`
  - `g_int_hash()` und `g_int64_hash()`
  - `g_double_hash()`
  - `g_str_hash()` : `h(k) = k + 33 + c`
- `g_hash_table_insert(*hash_table, key, value)`
- `g_hash_table_replace(*hash_table, key, value)`
- `g_hash_table_lookup(*hash_table, key)`

glib Beispiel

```c
#include <glib.h>

int main()
{
GHashTable *hash_table = g_hash_table_new(g_str_hash,
    g_str_equal);

g_hash_table_insert(hash_table,"Boehme","Paulus");
g_hash_table_insert(hash_table,"Simpson","Homer");
g_hash_table_insert(hash_table,"Muster","Max");

int size = g_hash_table_size(hash_table));
printf("%s",g_hash_table_lookup(hash_table,"Boehme"));

return 0;
}
```

Prüfsummen

- Erkennung von Veränderungen an Daten
- Hashfunktionen
  - simpel
  - Rechenaufwand = gering
  - Schutz vor Lese- und Schreibfehlern
  - Adler-32
- kryptografische Hashfunktionen
  - komplex
  - Rechenaufwand = hoch
  - Schutz vor gezielter Manipulation
  - SHA, MD5 und co.

Adler-32

```
uint32_t adler(unsigned char *data, size_t datalength)
{
      uint32_t s1 = 1;
      uint32_t s2 = 0;
      size_t n;

      for(n = 0; n < datalength; n++)
      {
           s1 = (s1 + data[n]) % 65521;
           s2 = (s2 + s1) % 65521;
      }

      return(s2 << 16) | s1;
}
```

glib - krypto. Hashfunktionen - Prüfsummen

```
sha256obj = g_compute_checksum_for_data(G_CHECKSUM_SHA256,data,-1);
sha1obj = g_compute_checksum_for_data(G_CHECKSUM_SHA1,data,-1);
md5obj = g_compute_checksum_for_data(G_CHECKSUM_MD5,data,-1);

printf("Datei: 500mb \n");
printf("SHA-256-Hashwert: %s \n",sha256obj);
printf("SHA-1-Hashwert: %s \n",sha1obj);
printf("MD5-Hashwert: %s \n",md5obj);

Output:

Datei: 500mb
SHA-256-Hashwert: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
SHA-1-Hashwert:   da39a3ee5e6b4b0d3255bfef95601890afd80709
MD5-Hashwert:     d41d8cd98f00b204e9800998ecf8427e
```

Effizienz

- Hashtabellen vs. Array vs. BST[1] vs. Linked List

	search()	insert()	delete()
Hashtabelle	O(1)	O(1)	O(1)
Array	O(1)/O(n)	O(n)/O(log n)	O(n)
BST	O(log n)	O(log n)	O(log n)
Linked List	O(n)	O(1)	O(1)

- Average-Case (Mittelwert)
- Worst-Case: O(n)

[1] Binary Search Tree

Test I

- Was wurde getestet?
  - Hashtabelle vs. Array
  - Offene Adressierung vs. Hashtabellen mit LL[1]
  - Selfmade vs. glib
- Was wurde verwendet?
  - Liste: 3902 Männer Vornamen als .txt
  - Home PC [AMD Phenom(tm)II X6 1090T 3.20 GHz]
  - hashing, insert, search

[1] Linked List

Test: Hashtabelle vs. Array

- Größe: 3.902
- Iterationen: 100

	Hashtabelle	Array
insert	0,45s	0,06s
search	0,47s	6,15s

Test: Offene Ad. vs. HT mit LL

- Größe LL: 100
- Größe OA: 3.902
- Iterationen: 1.000

	Offene Ad.	HT mit LL
insert	4,46s	1,36s
search	5,44s	1,56s

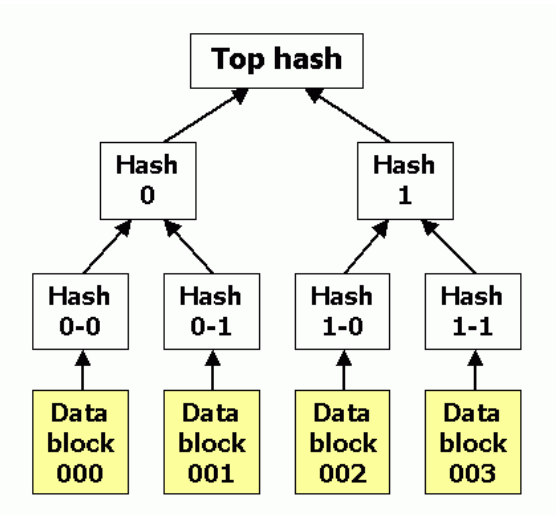
Test: glib vs. Selfmade

- Größe HT: 100
- Iterationen i/s: 1.000
- Iterationen h: 100.000.000

	Standard	glib
hashing	3,66s	2,63s
insert	1,33s	1,36s
search	1,56s	0,81s

HashTrees

- Binär-Baum
- Kinder sind Hash-Werte von Datenblöcken
- Nutzen: Integrität
- zwecks Übertragung: mehrere kleine Datenblöcke
- Integrität trotz Unvollständigkeit des Baumes
- Fehlerhafte Blöcke leicht und schnell austauschbar



Hashbaum, Wikipedia, http://upload.wikimedia.org/wikipedia/commons/6/6d/Hash_tree.png

Prüfsumme - ISBN

- ISBN10
  - $z_{10} = \left(\sum_{i=1}^{9} i * z_i\right) \bmod 11$
- ISBN13
  - $z_{13} = \left(10 - \left(\sum_{i=1}^{12} z_i * 3^{(i+2) mod 2}\right) mod 10\right) \bmod 10$
  - Erkennung von Lese- und Schreibfehler
  - letzte Zahl ist Puffer-Zahl
  - Alles gut, wenn ISBN10/ISBN13 = 0
- Bitcoin
  - Integrität von Client-SW

Hardware

- Trusted Platform Modul
  - Versiegelung
    - Bindung des Systems an einen TPM
    - Hashing der System-Konfiguration
  - beinah überall verbaut
- Skylake (Mikroarchitektur)
  - Intel, Nachfolger für Broadwell
  - voraussichtlich 2015
  - Intel-SHA-Erweiterung
    - SHA-1
    - SHA-256

Hardware (fort.)

CPU Architecture	Frequency	Algorithm	Word size (bits)	Cycles/Byte x86	MiB/s x86	Cycles/Byte x86-64	MiB/s x86-64
Intel Ivy Bridge	3.5 GHz	SHA-256	32-bit	16.80	199	13.05	256
		SHA-512	64-bit	43.66	76	8.48	394
AMD Piledriver	3.8 GHz	SHA-256	32-bit	22.87	158	18.47	196
		SHA-512	64-bit	88.36	41	12.43	292

SHA-256, Wikipedia, en.wikipedia.org/wiki/SHA-256, 19.01.2014

Was solltet ihr mitgenommen haben

- Hashtabellen
  - offene Adressierung u. Tabellen mit verketteten Listen
  - Effizienz
- Hashfunktionen
  - Datenbankverwaltung, Prüfsummen, krypt. Funktionen
- Implementation
  - \+ Standard
  - ++ Nützlich: glib, andere libs
- Sonstiges
  - Hashbäume
  - Hardware

Quellen

- Hashing, http://openbook.galileocomputing.de/c_von_a_bis_z/022_c_algorithmen_005.htm, 10.01.2014
- C-Programmierung, http://de.wikibooks.org/wiki/C-Programmierung/* , 10.01.2014
- Hashtabels, https://developer.gnome.org/glib/unstable/glib-Hash-Tables.html, 10.01.2014
- Hashfunktion, http://de.wikipedia.org/wiki/Hashfunktion, 10.01.2014
- Hashtabelle, http://de.wikipedia.org/wiki/Hashtabelle, 10.01.2014

Quellen fort.

- Alexander Koglin, Hashing, http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2012_2013/epc-1213-koglin-hashing-praesentation.pdf, 10.01.2014
- Big O-Notation, http://bigocheatsheet.com/, 19.01.2014
- Hashbaum, http://de.wikipedia.org/wiki/Hash-Baum, 19.01.2014
- Intel-SHA-Extension, http://software.intel.com/en-us/articles/intel-sha-extensions, 19.01.2014

Software, Libraries

- wxDev-Cpp (IDE)
- MinGw32, GNU C Compiler für Windows
- glib