Hashing

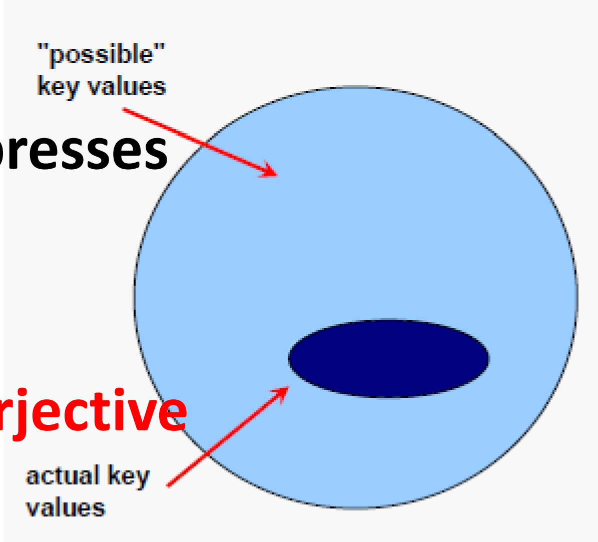
Alexander Koglin
Uni Hamburg
Seminar:
„Effiziente Programmierung in C“

10. Januar 2013

Structure

- Hash (function)
- Applications
- Implementations
- Summary
- References

Hash function (Streuwertfunktion)



courses.cs.vt.edu/~cs3114/ Summer11/Notes/T16.HashFunctions.pdf

- **a function (or algorithm) that usually compresses**

$$h: K \rightarrow S, |K| \leq |S|$$

- **K: set of keys, S: set of hashes**
- **mostly not injective and not necessarily surjective**
- **fixed output length**

- many different types of hash functions

- **Note: No randomness, but different grades of pseudo-randomness.**

Criterias for a good hash function

- hash function by definition
- easily and quickly computable
- ideally injective (perfect hash function)
  - uniform distribution of hashes → 2 different keys get different hashes
  - if not, you have a collision

- surjectivity
- chaos

Cryptographic hash function

- **popular**: MD5 (1992), SHA-1 (1995), but insecure
- **use instead**: SHA-2, WHIRLPOOL, RIPEMD-160 (1996)

- Change to the data will almost certainly change the hash.
- needs more CPU power than other hash functions

- the ideal cryptographic hash function’s properties:
  - it is infeasible to generate a message that has a given hash (one-way property)
  - it is infeasible to modify a message without changing the hash
  - it is infeasible to find two different messages with the same hash (collision resistance)

Example hash: DJBX33X (Daniel J. Bernstein)

„XOR“

„times“

```
uint32_t hash(string str)
{
        uint32_t hash = 5381;

        for (int i = 0; i < str.Length; i++)
        {
                hash = ((hash << 5) + hash) ^ (int)str[i];
        }
        return hash;
}
```

hash x 33

APPLICATIONS

Applications

  - identifying files/data
  - checking file/message integrity

checksums

- error correction
- (pseudo-)random number generators (for cryptography)

- cryptography
- password verfication
- in databases → hash table

Checksums (e.g.: crc32, ISBN)

- recognition of changes
  - by noise/interference
  - by sabotage → cryptographic hash function
- If hash values differ, then there was a change.→ not every hash function is suitable
- **Uses:**
  - Peer2Peer
  - Session ID in web applications (parameters: IP address, time, …)

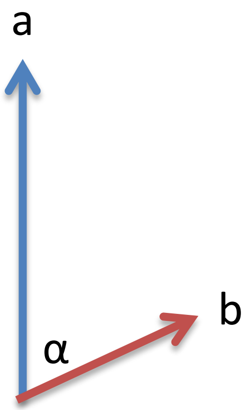
Efficient use cases for hashing

application/ purpose	key	type	in list
eliminate duplicates		dedup	duplicates
spell checker/ find misspelled words	word	exception	dictionary
browser/mark visited pages	URL	lookup	visited pages
chess/detect draw	board	lookup	positions
spam filter/eliminate spam	IP address	exception	spam
trusty filter	URL	lookup	good mail
credit cards	number	exception	stolen cards
data mining/ compare files	lines of file	comparison /lookup	mined documents

Data mining/statistics

- searching for similar files
- comparing files

- ansatz:
  - compute hashes of each line per document → vector
  - $\cos\alpha$ close to 0 → not similar
  - $\cos\alpha$ close to 1 → similar
- dimensions:
  penalty-factor if different



$$\cos\alpha = \frac{a * b}{||a|| * ||b||}$$

Hash chain

- apply successively a cryptographic-hash function (one-way) to a string
- application:
  - production of *one-time passwords* from a single key/password

→ in an insecure environment (e.g. server)

Server stores "$hashf^{1000}(password)$“, user authenticates by supplying "$hashf^{999}(password)$“, which is stored by the server.

→ You have 999 different passwords.

Hashing passwords

- saving passwords unencryptedly: high security risk
  - often: one username/password fits them all
- one-way cryptographic hash functions
- for a stronger password hash,
  - make the hash unique
    - randomly different hash functions
    - **salt** (min 64 bits): adding a random number at the end of the cleartext password
      - defeats rainbow tables, because the table can't be used to attack several passwords
  - make the algorithm slow
    - using hash chain
- **homemade is bad**

bcrypt and PBKDF2

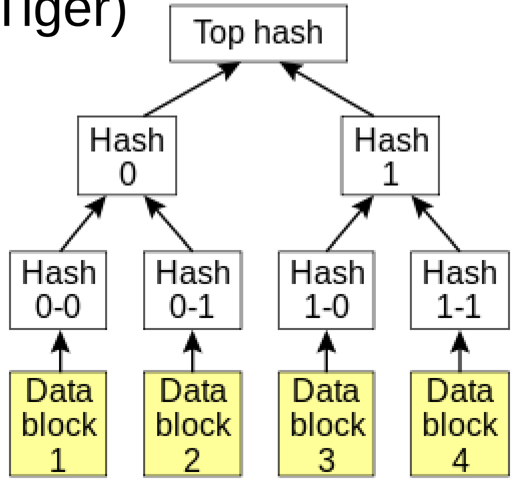
- cryptographic hash functions for password hashing (widely deployed standards)
- **difference to MD5 or SHA:**
  - MD5 and SHA are very efficient (especially they are used for checking file integrity) →Brute-Force and Rainbow-Tables are efficient
- **goal**: making hashing sufficiently inefficient
  - have a user-adjustable cost factor/number of hashings for hash calculation
    - cost factor can be adjusted in the future, when CPU/GPU power rises
- uses:
  - WPA and **WPA2**
  - **MacOS X** Mountain Lion (user passwords)
  - file system encryption in **Android and iOS**
  - **1Password** and **LastPass**
  - OpenOffice
  - WinZip

(Binary) hash tree

- data structure containing a tree of summary information about a larger piece of data
- used for verification
- Most hash tree implementations are binary.
- cryptographic hash function (SHA-1, Whirlpool, Tiger)

- function:…

- **uses**
  - Peer2Peer: check if received blocks are undamaged/unmanipulated
    - first: get top hash from trusted source
  - ZFS file system by Sun Microsystems
  - Git
  - Bitcoin
  - some NoSQL systems like Apache Cassandra



http://en.wikipedia.org/wiki/File:Hash_Tree.svg
David Göthberg

APPLICATIONS: HASH TABLE

„Dictionary Problem“

- **given:** set of objects, which can be identified by a unique key
- **wanted:** data structure for efficient execution of
  - *searching*
  - *pasting/adding*
  - *deleting*

Solving the problem

- **influenced by:**
  - **storage**: main memory or hard drive
  - rate of the operations (searching, pasting & deleting, uniform)
  - additional operations (specific order, union, section,…)
  - execution order of operations

Hash table/associative array

- key values are mapped to array index values
- fast
- relatively easy to program as compared to trees
- based on arrays, hence difficult to expand
- no convenient way to visit the items in a hash table in any kind of order→ trees are better
- no cmps
- constant complexity
- use: management of data

e.g:

  - DHT: distributed hash table: Big databases can be split up to networks.
  - Traffic and memory consumption are split up to different servers.

Collisions

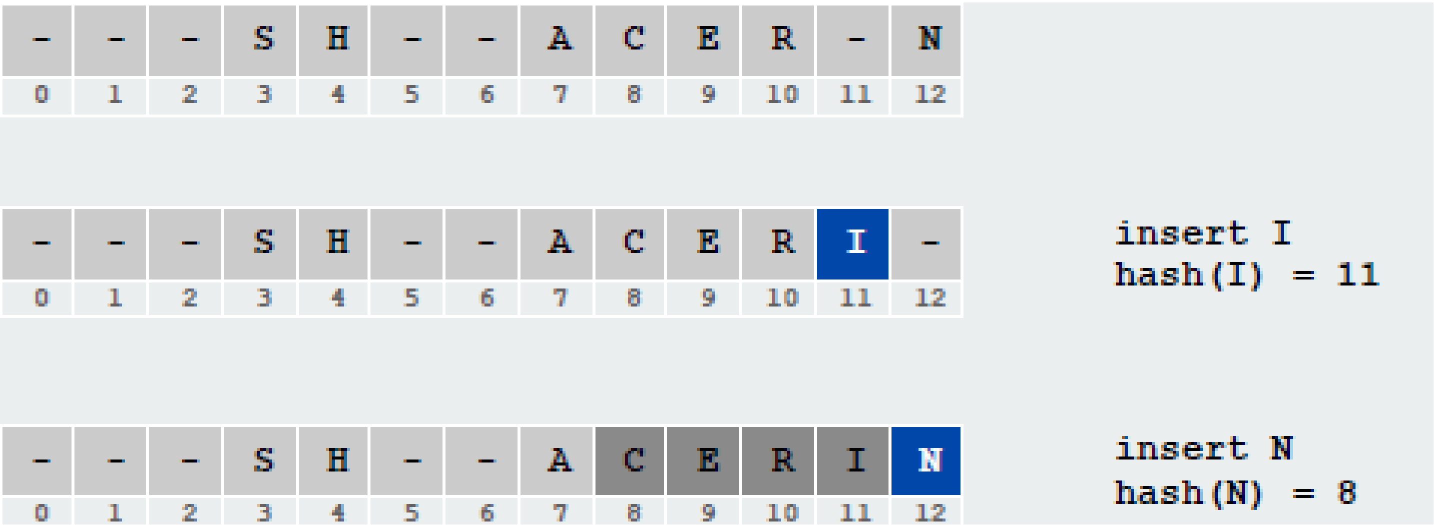
- 2 different keys are hashed to the same array index.

„Birthday Problem“ (link)

What’s the probability, that two people in the room have the same birthday?
→ for hashing: Collisions will certainly occur, when you have many hashes.
→ You need much memory, when you want to avoid as many collisions as possible.
→ deal as efficiently as possible with them

Collision resolution with “Open Addressing” (IBM, 1953)

- **“Linear Probing”:**
  - starts with hash address and searches sequentially an empty position
- **“Quadratic Probing”:**
  - If there is a collision at hash address h, quadratic probing goes to location h+i^2.
- As the array gets full, clusters grow larger, resulting in very long probe lengths.



0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	S	H	-	-	A	C	E	R	-	N

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	S	H	-	-	A	C	E	R	I	-

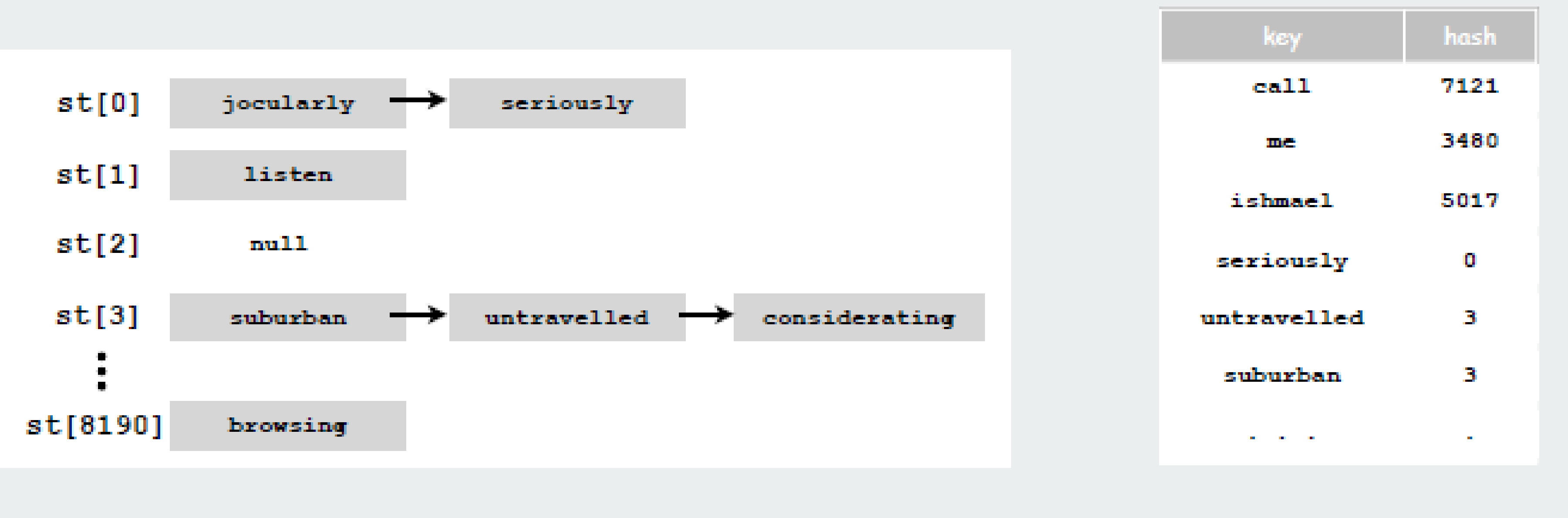
```
insert I
hash(I) = 11
```

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	-	S	H	-	-	A	C	E	R	I	N

```
insert N
hash(N) = 8
```

http://www.cs.princeton.edu/~rs/AlgsDS07/10Hashing.pdf

“Separate Chaining” (IBM, 1953)



key	hash
call	7121
me	3480
ishmael	5017
seriously	0
untravelled	3
suburban	3
. . .	.

http://www.cs.princeton.edu/~rs/AlgsDS07/10Hashing.pdf

- load factor can be ≥1
- deletion: no problem
- number of lists too small→chains too long
- number of lists too big→too many empty chains

Hash table - efficiency

	search	insert	delete
unordered array	O(n)	O(n)	O(n)
ordered array	O(log(n))	O(n)	O(n)
unordered list	O(n)	O(n)	O(n)
ordered list	O(n)	O(n)	O(n)
binary tree	O(log(n))	O(log(n))	O(log(n))
hashing	O(1)	O(1)	O(1)

- with collision: separate chaining: Access time depends on the resulting probe lengths.
- Time is proportional to the length of the probe in addition to a constant time for hash function.

“Open Addressing” vs. “Separate Chaining”

- If plenty of memory is available and the data won’t expand, then “Linear Probing” is simpler to implement.
- “Open Addressing” doesn’t support efficient deletion, but “Separate Chaining” needs more memory.
- If number of items to be inserted in hash table isn’t known, “Separate Chaining” is preferable to open addressing because of the dynamicly adjustable size.

Complexity attack

- „Denial of Service“ attack:
  - If the attacker knows your hash function, he can intentionally ask the server for a colliding key, which causes high CPU load.
  - Many languages are still vulnerable.
  - although it could be prevented by e.g. reducing time per hash

IMPLEMENTATIONS

LibTomCrypt: libtom.org

- among others:

- **one-way hash functions**
  - MD4
  - MD5
  - SHA-1
  - SHA-224/256/384/512
  - TIGER-192
  - RIPE-MD 128/160/256/320
  - WHIRLPOOL
- **pseudorandom number generators**
  - ...
- **public key algorithms**
  - ...
- **other standards**
  - PKCS #5 (password encryption)

Glib

- various hashing algorithms like MD5, SHA-1 and SHA-256.
  - http://developer.gnome.org/glib/2.28/glib-Data-Checksums.html

- GHashTable
  - developer.gnome.org/glib/2.29/glib-Hash-Tables.html
- Note: Neither keys nor values are copied when inserted.
  - → Temporary strings should be copied with g_strdup() before being inserted.
- Note: If keys or values are dynamically allocated: ensure that they are freed when they are removed/overwritten.

http://www.ibm.com/developerworks/linux/tutorials/l-glib/section5.html

```c
#include <glib.h>
int main(int argc, char** argv)
{
GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);

g_hash_table_insert(hash, "Virginia", "Richmond");
g_hash_table_insert(hash, "Texas", "Austin");
g_hash_table_insert(hash, "Ohio", "Columbus");

printf("There are %d keys in the hash\n", g_hash_table_size(hash));
printf("The capital of Texas is %s\n", g_hash_table_lookup(hash,
"Texas"));
gboolean found = g_hash_table_remove(hash, "Virginia");
printf("The value 'Virginia' was %sfound and removed\n", found ? "" :
"not ");
g_hash_table_destroy(hash);
return 0;
}
```

***** Output *****

```
There are 3 keys in the hash
The capital of Texas is Austin
The value 'Virginia' was found and removed
```

Inserting/replacing values

g_hash_table_insert: key already exists? Value will be replaced but not the key.

g_hash_table_replace: replacing both

```
#include <glib.h>
static char* texas_1, *texas_2;
void key_destroyed(gpointer data) {
printf("Got a key destroy call for %s\n", data == texas_1 ? "texas_1" : "texas_2");
}
int main(int argc, char** argv) {
GHashTable* hash = g_hash_table_new_full(g_str_hash, g_str_equal,
(GDestroyNotify)key_destroyed, NULL);
texas_1 = g_strdup("Texas");
texas_2 = g_strdup("Texas");
g_hash_table_insert(hash, texas_1, "Austin");
printf("Calling insert with the texas_2 key\n");
g_hash_table_insert(hash, texas_2, "Houston");
printf("Calling replace with the texas_2 key\n");
g_hash_table_replace(hash, texas_2, "Houston");
printf("Destroying hash, so goodbye texas_2\n");
g_hash_table_destroy(hash);
g_free(texas_1);
g_free(texas_2);
return 0; }
```

```
***** Output *****
Calling insert with the texas_2 key
Got a key destroy call for texas_2
Calling replace with the texas_2 key
Got a key destroy call for texas_1
Destroying hash, so goodbye texas_2
Got a key destroy call for texas_2
```

Summary

- Hashing helps to achieve many goals efficiently:
  - comparing/checking integrity
  - searching data
  - error correction
  - random numbers
  - storing passwords securely (but: hommade is bad)

- A hash table is an efficient data structure.
  - but beware of collisions
- Implementations (LibTomCrypt or Glib) are easy to use.

References

- http://www.ibm.com/developerworks/linux/tutorials/l-glib/section5.html
- http://de.wikipedia.org/wiki/Hashfunktion
- http://en.wikipedia.org/wiki/Hash_tree
- http://www.cs.princeton.edu/~rs/AlgsDS07/10Hashing.pdf
- http://www.cs.princeton.edu/courses/archive/fall08/cos521/hash.pdf
- http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-18.pdf
- http://electures.informatik.uni-freiburg.de/portal/download/56/7071/11_Hashverfahren_4up.pdf
- http://www.hit.bme.hu/~buttyan/courses/BSc_Coding_Tech/hash-mac-sig.pdf
- http://events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf
- www.latech.edu/~box/ds/chap11.ppt
- http://courses.cs.vt.edu/~cs3114/Summer11/Notes/T16.HashFunctions.pdf
- libtom.org
- http://developer.gnome.org/glib/2.28/glib-Data-Checksums.html