6 Hashing

Summer Term 2011

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

The dictionary problem

Different approaches to the dictionary problem:

- Previously: Structuring the set of currently stored keys: trees (lists, graphs)
- Structuring the complete universe of all possible keys: hashing

Hashing describes a special way of storing the elements of a set by partitioning the universe of possible keys.

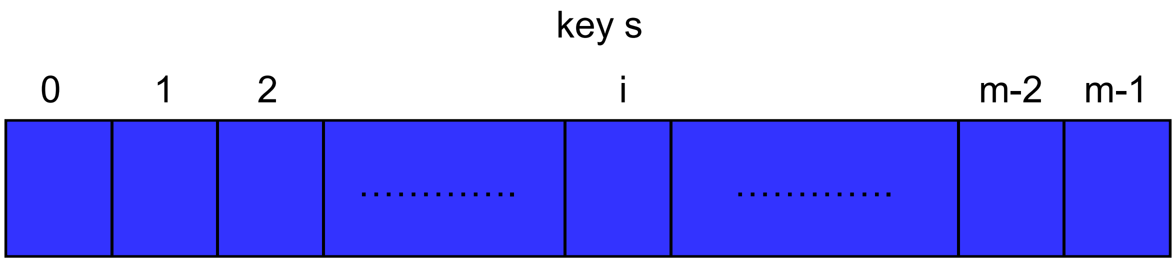
The position of the data element in the memory is given by computing a so-called hash value directly from the key.

Hashing

Dictionary problem:
    Lookup, insertion, deletion of data sets (keys)

Place of data set *d*: computed from the key *s* of *d*
    → no comparisons
    → constant time

Data structure: linear field (array) of size *m*
    Hash table



The memory is divided into *m* containers (buckets) of the same size.

Example

- Suppose the keys are numbers between 1, ..., n, and $m \ll n$. Then we might use "mod m" as a hash function. Each key i is stored in bucket i mod m.

Implementation in Java

```
class TableEntry {
      private Object key,value;
   }
   abstract class HashTable {
      private TableEntry[] tableEntry;
      private int capacity;
      // Constructor
      HashTable (int capacity) {
          this.capacity = capacity;
          tableEntry = new TableEntry [capacity];
          for (int i = 0; i <= capacity-1; i++)
              tableEntry[i] = null;
      }
      // the hash function
      protected abstract int h (Object key);
      // insert element with given key and value (if not there already)
      public abstract void insert (Object key Object value);
      // delete element with given key (if there)
      public abstract void delete (Object key);
      // locate element with given key
      public abstract Object search (Object key);
   } // class hashTable
```

Hashing - problems

1. Size of the hash table
   Only a small subset *S* of all possible keys (the universe) *U* actually occurs. The size of the hash table should be not too much bigger than the number of occurring keys.

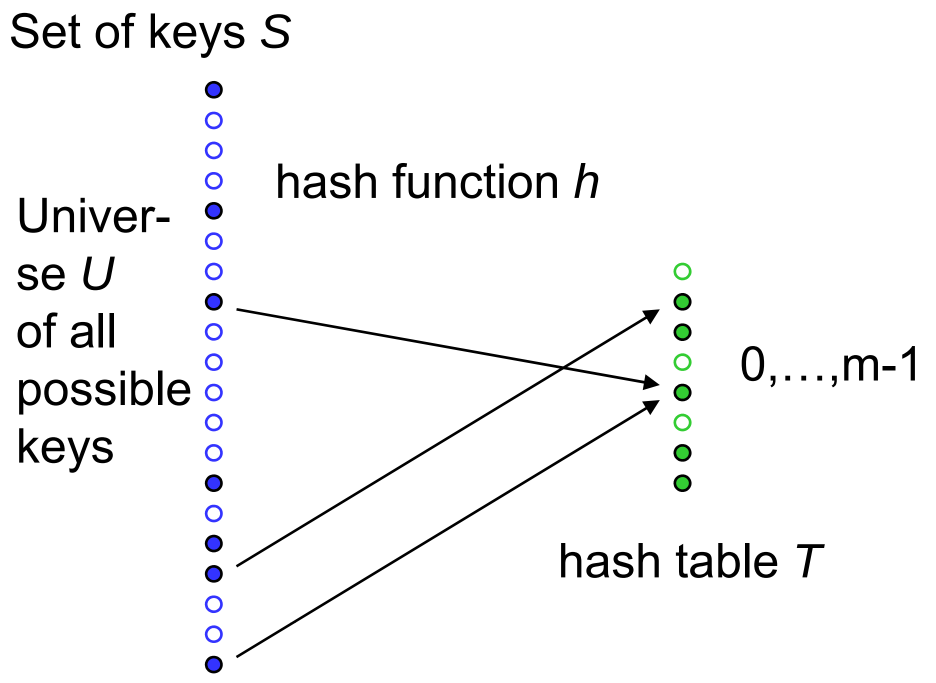
2. Calculation of the adress of a data set
   - keys are not necessarily integers
   - index depends on the size of hash table

In Java:

```
public class Object {
    ...
    public int hashCode()  {…}
    ...
}
```

The universe *U* should be distributed as evenly as possibly to the numbers 0,…,*m-1.*

Hash function (1)



$h(s)$ = hash address

$h(s) = h(s')$ $\Leftrightarrow$ s and s' are synonyms with respect to h

address collision

Hash function (2)

Definition: Let U be a universe of possible keys and $\{B_0, . . . ,B_{m-1}\}$ a set of m buckets for storing elements from U. Then a hash function

$$h : U \rightarrow \{0, \dots , m - 1\}$$

maps each key $s \in U$ to a *value* $h(s)$
(and the corresponding element to the bucket $B_{h(s)}$).

- The indices of the buckets also called hash addresses, the complete set of buckets is called hash table.

B_0
B_1
…
…
B_{m-1}

Address collisions

- A hash function h calculates for each key s the index of the associated bucket.
- It would be ideal if the mapping of a data set with key s to a bucket $h(s)$ was unique (one-to-one): insertion and lookup could be carried out in constant time ($O(1)$).
- In reality, there will be collisions: several elements can be mapped to the same hash address. Collisions have to be addressed (in one way or another).

Hashing methods

Example for U: all names in Java with length ≤ 40 → $|U| = 62^{40}$

If $|U| > m$: address collisions are inevitable

Hashing methods:

1. Choice of a hash function that is as "good" as possible
2. Strategy for resolving address collisions

Load factor α: $\alpha = \frac{\text{\# of stored keys}}{\text{size of hash table}} = \frac{|S|}{m} = \frac{n}{m}$

Assumption: table size m is fixed

Requirements for good hash functions

Requirements

- A collision occurs if the bucket $B_{h(s)}$ for a newly inserted element with key s is not empty.

- A hash function h is called perfect for a set S of keys if no collisions occur for S.

- If h is perfect and $|S| = n$, then $n \leq m$.
  The load factor of the hash table is $n/m \leq 1$.

- A hash function is well chosen if
  – the load factor is as high as possible,
  – for many sets of keys the number of collisions is as small as possible,
  – it can be computed efficiently.

Example of a hash function

Example: hash function for strings

```
public static int h (String s){
    int k = 0, m = 13;
    for (int i=0; i < s.length(); i++)
        k += (int)s.charAt (i);
    return ( k%m );
}
```

The following hash addresses are generated for m = 13.

key s	$h(s)$
Test	0
Hallo	2
SE	9
Algo	10

The greater the choice of m, the closer to perfect h becomes.

Probability of collision (1)

Choice of the hash function

- The requirements high load factor and small number of collisions are in conflict with each other. We need to find a suitable compromise.
- For the set S of keys with $|S| = n$ and buckets $B_0, \ldots, B_{m-1}$:
  – for $n > m$ conflicts are inevitable
  – for $n < m$ there is a probability $P_K(n,m)$ for the occurrence of at least one collision.

How can we find an estimate for $P_K(n,m)$?

- For any key s the probability that $h(s) = j$ with $j \in \{0, \ldots, m - 1\}$ is:
  $P_K\,[h(s) = j] = 1/m$, provided that there is an equal distribution.
- We have $P_K(n,m) = 1 - P_{\neg K}(n,m)$,
  if $P_{\neg K}(n,m)$ is the probability that storing of n elements in m buckets leads to no collision.

Probability of collision (2)

On the probability of collisions

- If n keys are distributed sequentially to the buckets B_0, ..., B_{m-1} (with equal distribution), each time we have $P\,[h(s) = j\,] = 1/m$.
- The probability $P(i)$ for no collision in step i is $P(i) = (m - (i - 1))/m$
- Hence, we have

$$P_K(n,m) = 1 - P(1) * P(2) ... * P(n) = 1 - \frac{m(m-1)...(m-n+1)}{m^n}$$

For example, if $m = 365$, $P(23) > 50\%$ and $P(50) \approx 97\%$ (“birthday paradox”)

Common hash functions

Hash fuctions used in practice:

- see: D.E. Knuth: *The Art of Computer Programming*
- For U = integer the divisions-residue method is used:

  $h(s) = (a \times s) \bmod m$ ($a \neq 0$, $a \neq m$, m prime)

- A more complicated example: For strings of characters of the form $s = s_0 s_1 \ldots s_{k-1}$ one can use

$$h(s) = \left(\left(\sum_{i=0}^{k-1} B^i \ s_i \right) mod \ 2^w \right) mod \ m$$

e.g. B = 131 and w = word width (bits) of the computer (w = 32 or w = 64 is common).

Simple hash functions

Choice of the hash function
- simple and quick computation
- even distribution of the data (example: compiler)

(Simple) division-residue method

$$h(k) = k \bmod m$$

How to choose of m?

Examples:

a) m even $\rightarrow h(k)$ even $\Longleftrightarrow$ k even

Problematic if the last bit has a meaning (e.g. 0 = female, 1 = male)

b) $m = 2^p$ yields the p lowest dual digits of k

Rule: Choose m prime, and m is not a factor of any r^i +/- j ,
where i and j are small, non-negative numbers and r is the radix of the number representation.

Multiplicative method (1)

- Choose constant $\theta, 0 < \theta < 1$
- 1. Compute $k\theta \ mod \ 1 = k\theta - \lfloor k\theta \rfloor$
- 2. $h(k) = \lfloor m(k\theta) \ mod \ 1 \rfloor$

Multiplicative method (2)

- Example:

$$\begin{aligned}
\theta &= \frac{\sqrt{5}-1}{2} \approx 0.6180339887 \\
k &= 123456 \\
m &= 10000
\end{aligned}$$

$$\begin{aligned}
h(k) &= \lfloor 10000(123456 \cdot 0.6180339887\ldots \mod 1) \rfloor \\
&= \lfloor 10000(76300.0041089472\ldots \mod 1) \rfloor \\
&= \lfloor 41.089472\ldots \rfloor \\
&= 41
\end{aligned}$$

- Of all numbers $0 \leq \theta \leq 1, \frac{\sqrt{5}-1}{2}$ leads to the most even distribution.

Universal hashing

- Problem: if *h* is fixed, then there are $S \subseteq M$ with many collisions
- Idea of universal hashing:
  Choose hash function *h* randomly

- *H* finite set of hash functions

$$h \in H : U \rightarrow \{0, \ldots, m-1\}$$

- Definition: *H* is universal, if for arbitrary *x*,*y* ∈ *U*:

$$\frac{|\{h \in H : h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

- Hence: if *x*, *y* ∈ *U*, *H* universal, *h* ∈ *H* picked randomly

$$Pr_H\,(h(x) = h(y)) \leq \frac{1}{m}$$

A universal class of hash functions

Assumptions:

- $|U| = p$ (p prime) and $U = \{0, \ldots, p-1\}$
- Let $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$ and $\underline{h_{a,b}} : \underline{U} \rightarrow \{0,\ldots,m-1\}$ be defined as follows

$$h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$$

Then:

The set

$$H = \{h_{a,b} \mid 1 \leq a < p, 0 \leq b < p\}$$

is a universal class of hash functions.

Universal hashing – example

Hash table T of size 3, $|U| = 5$

Consider the 20 functions (set H):

$x+0$	$2x+0$	$3x+0$	$4x+0$
$x+1$	$2x+1$	$3x+1$	$4x+1$
$x+2$	$2x+2$	$3x+2$	$4x+2$
$x+3$	$2x+3$	$3x+3$	$4x+3$
$x+4$	$2x+4$	$3x+4$	$4x+4$

each (mod 5) (mod 3)

and the keys 1 und 4

For 4 our of 20 functions, we get a collision:

(1*1+0) mod 5 mod 3 = 1 = (1*4+0) mod 5 mod 3
(1*1+4) mod 5 mod 3 = 0 = (1*4+4) mod 5 mod 3
(4*1+0) mod 5 mod 3 = 1 = (4*4+0) mod 5 mod 3
(4*1+4) mod 5 mod 3 = 0 = (4*4+4) mod 5 mod 3

Possible ways of treating collisions

Treatment of collisions:

- Collisions are treated differently in different methods.
- A data set with key *s* is called a colliding element if bucket $B_{h(s)}$ is already taken by another data set.
- What can we do with colliding elements?
  1. Chaining: Implement the buckets as linked lists. Colliding elements are stored in these lists.
  2. Open Addressing: Colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called probing.