6 Hashing
The dictionary problem

Different approaches to the dictionary problem:

- previously: Structuring the set of currently stored keys: lists, trees, graphs, ...
- structuring the complete universe of all possible keys: hashing

**Hashing** describes a special way of storing the elements of a set by breaking down 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 set (keys)

Place of data set $d$: computed from the key $k$ of $d$
$\rightarrow$ no comparisons
$\rightarrow$ constant time

Data structure: linear field (array) of size $m$
hash table

The memory is divided in $m$ containers (buckets) of the same size.
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

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 $-2^{31}, \ldots, 2^{31}-1$. 
Hash function (1)

Set of keys $S \subseteq U$

Universe $U$ of all possible keys

hash function $h$

$0,\ldots,m-1$

hash table $T$

$h(s) = \text{hash address}$

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

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

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

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.
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 $\leq 40 \rightarrow |U| = 62^{40}$

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

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 # of collisions is as small as possible,
  - it can be computed efficiently.
Example of a hash function

Example: hash function for strings

```java
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 $m = 13$

<table>
<thead>
<tr>
<th>key $s$</th>
<th>$h(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>0</td>
</tr>
<tr>
<td>Hallo</td>
<td>2</td>
</tr>
<tr>
<td>SE</td>
<td>9</td>
</tr>
<tr>
<td>Algo</td>
<td>10</td>
</tr>
</tbody>
</table>

The greater the choice of $m$, the more 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 $B_0, ..., B_{m-1}$:
  - for $n > m$ conflicts are inevitable
  - for $n < m$ there is a (residual) 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 $j \in \{0, ..., m - 1\}$ 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.
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

$$P_K(n, m) = 1 - P(1) \cdot P(2) \cdots P(n) = 1 - \frac{m(m-1)\ldots(m-n+1)}{m^n}$$

For example, if $m = 365$, $P(23) > 50\%$ and $P(50) \approx 97\%$ (“birthday paradox”)
Probability of collision (3)

Birthday paradox

![Graph showing the probability of a pair occurring as a function of the number of people. The x-axis represents the number of people, ranging from 0 to 100, with a significant jump at 2^{23}. The y-axis represents the probability of a pair, ranging from 0 to 1. The graph approaches 1 as the number of people increases.]
Common hash functions

Hash functions used in practice:

- see: D.E. Knuth: *The Art of Computer Programming*
- for $U = \text{integer}$ the [divisions-residue method] is used:
  $$h(s) = (a \times s) \mod m \ (a \neq 0, \ a \neq m, \ m \text{ prime})$$
- for strings of characters of the form $s = s_0s_1 \ldots s_{k-1}$ 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 $w = \text{word width (bits) of the computer}$ ($w = 32$ 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

\[ h(k) = k \mod m \]

How to choose of \( m \)?

Examples:

a) \( m \) even \( \rightarrow h(k) \) even \( \iff k \) 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 representation.
Multiplicative method (1)

- Choose constant \( \theta, 0 < \theta < 1 \)
- 1. Compute \( k\theta \mod 1 = k\theta - \lfloor k\theta \rfloor \)
- 2. \( h(k) = \lfloor m(k\theta) \mod 1 \rfloor \)
- Choice of \( m \) is uncritical
Multiplicative method (2)

- Example:

\[
\begin{align*}
\theta &= \frac{\sqrt{5} - 1}{2} \approx 0.6180339887 \\
k &= 123456 \\
m &= 10000 \\
\end{align*}
\]

\[
\begin{align*}
h(k) &= \lfloor 10000(123456 \cdot 0.6180339887 \ldots \mod 1) \rfloor \\
&= \lfloor 10000(76300.0041089472\ldots \mod 1) \rfloor \\
&= \lfloor 41.089472\ldots \rfloor \\
&= 41
\end{align*}
\]

- 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 $\rightarrow$ there are $S \subseteq U$ with many collisions

- **Idea of universal hashing:**
  Choose hash function $h$ randomly ($h$ is independent of the keys that are going to be stored)

- $H$ finite set of hash functions
  \[
  h \in H : U \rightarrow \{0, \ldots, m - 1\}
  \]

- **Definition:** $H$ is universal, if for arbitrary distinct keys $x, y \in U$:
  \[
  \frac{|\{h \in H : h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}
  \]

- Hence: if $x$, $y \in U$, $H$ universal, $h \in H$ 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 $h_{a,b} : U \to \{0, \ldots, m-1\}$ be defined as follows

\[ h_{a,b} = ((ax+b) \mod p) \mod m \]

Then:

The set

\[ H = \{h_{a,b} \mid 1 \leq a < p-1, 0 \leq b < p-1\} \]

is a universal class of hash functions.
Universal hashing – example

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

Consider the 20 functions (set $H$):

\[
\begin{align*}
x+0 & \quad 2x+0 & \quad 3x+0 & \quad 4x+0 \\
x+1 & \quad 2x+1 & \quad 3x+1 & \quad 4x+1 \\
x+2 & \quad 2x+2 & \quad 3x+2 & \quad 4x+2 \\
x+3 & \quad 2x+3 & \quad 3x+3 & \quad 4x+3 \\
x+4 & \quad 2x+4 & \quad 3x+4 & \quad 4x+4
\end{align*}
\]

each (mod 5) (mod 3)

and the keys 1 and 4

We get:

\[
\begin{align*}
(1\cdot1+0) \mod 5 \mod 3 &= 1 = (1\cdot4+0) \mod 5 \mod 3 \\
(1\cdot1+4) \mod 5 \mod 3 &= 0 = (1\cdot4+4) \mod 5 \mod 3 \\
(4\cdot1+0) \mod 5 \mod 3 &= 1 = (4\cdot4+0) \mod 5 \mod 3 \\
(4\cdot1+4) \mod 5 \mod 3 &= 0 = (4\cdot4+4) \mod 5 \mod 3
\end{align*}
\]
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.