### 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: computed from the key of the data set
  - 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.

---

### Implementation in Java

```java
class TableEntry {
    private Object key, value;
}

abstract class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;

    HashTable(int capacity) {
        this.capacity = capacity;
        this.tableEntry = new TableEntry[capacity];
        for (int i = 0; i < capacity; i++)
            tableEntry[i] = null;
    }

    protected abstract int h(Object key);

    public abstract void insert(Object key, Object value);
    public abstract void delete(Object key);
    public abstract Object search(Object key);
}
```

---

### Hashing - Problems

1. Size of 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 address of a data set
   - Keys are not necessarily integers
   - Index depends on the size of hash table

In Java:

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

The universe $U$ should be distributed as evenly as possible to the numbers $0, ..., m - 1$. 

---
Hash function (1)

- Set of keys $S$
- Universe $U$ of all possible keys
- Hash function $h$
- Hash table $T$

Hash function (2)

Definition: Let $U$ be a universe of possible keys and $(B_0, \ldots, B_m)$ 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 $B_{h(s)}$).

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

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

Load factor $\alpha = \frac{|S|}{m}$

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.
- 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).

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</th>
<th>h(s)</th>
</tr>
</thead>
<tbody>
<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 closer to perfect $h$ becomes.
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 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:
  - $P_k[h(s) = j] = \frac{1}{m}$, provided that there is an equal distribution.
- We have $P_k(n,m) = 1 - P_k(n,m)$, if $P_k(n,m)$ 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 $P_k[h(s) = j] = \frac{1}{m}$.
- The probability $P_i$ for no collision in step $i$ is $P_i = \left(\frac{m - (i-1)}{m}\right)$.
- Hence, we have

$$P_k(n,m) = 1 - P(1) \cdot P(2) \cdot ... \cdot P(n) = 1 - \frac{(n-1)!}{(m-1)!}$$

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

Common hash functions
Hash functions used in practice:
- see: D.E. Knuth: The Art of Computer Programming
- For $U$ = integer the divisions-residue method is used:
  - $h(k) = (a \times k) \mod m$ ($a \neq 0, a \neq m$, $m$ prime)
- A more complicated example: For strings of characters of the form $s = s_0s_1...s_{k-1}$ one can use
  - $h(s) = \left(\sum_{i=0}^{k-1} B^i s_i\right) \mod 2^w \mod m$

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 $\iff$ $h(k)$ even $\iff$ $k$ even
- b) $m = 2^p$ yields the $p$ lowest dual digits of $k$

Rule: Choose $m$ prime, and $m$ is not a factor of any $r_i \pm j_i$, where $i$ and $j$ are small, non-negative numbers and $r$ is the radix of the number representation.

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

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

Examples:
- $\theta = 0.6180339887$
- $k$ = 123456
- $m$ = 100000

$h(k) = \left[100000 \times 123456 \times (0.6180339887 \times \mod 1)\right]$
  $= [100000 \times 76300 + 0.04166667 \times \mod 1]$
  $= 41,089,172 \ldots$

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

- Problem: If h is fixed, then there are many collisions.
- Idea of universal hashing:
  - Choose hash function h randomly.
  - H finite set of hash functions
  - Definition: H is universal, if for arbitrary x, y ∈ U:
    \[\Pr_H(h(x) = h(y)) \leq \frac{1}{m}\]

A universal class of hash functions

Assumptions:
- |U| = p (p prime) and \(U = \{0, \ldots, p-1\}\)
- Let a ∈ {1, ..., p-1}, b ∈ {0, ..., p-1} and \(h_{a,b} : U \rightarrow \{0, \ldots, m-1\}\) be defined as follows
  \[h_{a,b}(x) = ((ax+b) \mod p) \mod m\]

Then:
The set
\[H = \{h_{a,b} | 1 \leq a < p, 0 \leq b < p\}\]
is a universal class of hash functions.

Universal hashing – example

Consider the 20 functions (set H):
\[
\begin{align*}
&x=0 & 2x+0 & 3x+0 & 4x+0 \\
&x=1 & 2x+1 & 3x+1 & 4x+1 \\
&x=2 & 2x+2 & 3x+2 & 4x+2 \\
&x=3 & 2x+3 & 3x+3 & 4x+3 \\
&x=4 & 2x+4 & 3x+4 & 4x+4 \\
\end{align*}
\]
each (mod 5) (mod 3)
and the keys 1 and 4

For 4 out of 20 functions, we get a collision:
\[
\begin{align*}
&(1^1+0) \mod 5 \mod 3 = 1 = (1^4+0) \mod 5 \mod 3 \\
&(1^1+4) \mod 5 \mod 3 = 0 = (1^4+4) \mod 5 \mod 3 \\
&(4^1+0) \mod 5 \mod 3 = 1 = (4^4+0) \mod 5 \mod 3 \\
&(4^1+4) \mod 5 \mod 3 = 0 = (4^4+4) \mod 5 \mod 3 \\
\end{align*}
\]

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.