Hash function

• the universe of keys is a subset of
  \[ N = \{0,1,2, \ldots \} \]

• if the keys are not natural numbers - interpret them as natural numbers

Example:

a character string
  consider successive ASCII codes
Method for Creating Hash Function

maps the universe $U$ of keys into the slots of a hash table $T$

1. The division method.
2. The multiplication method.
3. Universal hashing.
Building hash function: division method

\[ h(k) = k \mod m \] 0-based arrays

experiments =>
good values for \( m \) are
prime not too close to exact powers of 2
Building hash function: multiplication method

The multiplication method

\[ h(k) = \text{floor}( m \times \text{frac}(k \times A)) \]

where

- \( m \) - hash table size
- \( A \) - constant in the range \( 0 < A < 1 \)

Remark:
- the value of \( m \) is not critical

Easy implementation:
- restrict \( A = s/2^w \),
- \( w \) = machine word size

\[ m = 2^p \text{ for some integer } p \]
Building hash function: multiplication method

The multiplication method

good value for $A$
(experimental)

$$A \approx \frac{\sqrt{5} - 1}{2} \approx 0.6180339887$$

Donald Knuth, *The Art of Computer Programming*, 1968

Numeric example:

$k = 123456$
$m = 10000$
$A = 0.6180339887$

$h(k) = \text{floor}(41.151…) = 41$

$k = 50$

$h(k) = 9016$
## Hash function

<table>
<thead>
<tr>
<th>key</th>
<th>Multiplication Method</th>
<th>Division Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456</td>
<td>4</td>
<td>456</td>
</tr>
<tr>
<td>123459</td>
<td>858</td>
<td>459</td>
</tr>
<tr>
<td>123496</td>
<td>725</td>
<td>496</td>
</tr>
<tr>
<td>123956</td>
<td>21</td>
<td>956</td>
</tr>
<tr>
<td>129456</td>
<td>208</td>
<td>456</td>
</tr>
<tr>
<td>193456</td>
<td>383</td>
<td>456</td>
</tr>
<tr>
<td>923456</td>
<td>195</td>
<td>456</td>
</tr>
</tbody>
</table>

The value of \( m \) is not critical.
Building hash function: universal hashing

**Universal hashing**: refers to selecting a hash function at random from a family of hash functions with a certain property.

**Universal class of hash functions**

Let $H$ be a finite collection of hash functions that map: $U \rightarrow \{0,1,\ldots,m-1\}$

Such a collection is said to be universal if for each pair of distinct keys $x, y \in U$, the number of hash functions for which $h(x) = h(y)$ is at most $|H|/m$.

With a function $h$ chosen uniformly at random from $H$, the chance of a collision between $x$ and $y$, where $x \neq y$, is less than $1/m$.

$$P(h(x) = h(y)) \leq \frac{1}{m}$$
Building hash function: universal hashing

Example:

m – the size of hash table, prime
key x: decompose a key x into r bytes

\[ x = <x_1, x_2, \ldots, x_r> \]

with: \( x_i \leq m \)

hash function:

\[ h_a(x) = \sum_{i=1}^{r} a_i \cdot x_i \mod m \]

<a_1, a_2, \ldots, a_r> is a fixed sequence of random numbers
\( a_i \in \{0, \ldots, m-1\} \)

universal class of hash functions

\[ H = \bigcup_{a} h_a \]

union taken over all possible \( a \)-s

- \( m^r \) members
- can be shown to be universal
Building hash function: universal hashing

**Universal hashing:** refers to selecting a hash function at random from a family of hash functions with a certain property.

Useful for algorithms that need multiple hash functions. Ex.: rehashing

the data structure needs to be rebuilt

if too many collisions occur.
Hash function

• perfect hash function
  – injective: maps distinct elements with no collisions
  – it is too expensive to compute it for every input

→ build a hash function to minimize collisions

good hash function

In practice:
  – use heuristic information to create a hash function that is likely to perform well

Choose between:
  – simple and fast, but have a high number of collisions;
  – more complex functions, with better quality, but take more time to calculate
Good hash function

A good hash function satisfies the assumption of simple uniform hashing

- a key \( x \) is equally likely to hash to any of the \( m \) slots
  \[
P(h(x)=j) = \frac{1}{m}, \text{ for any } j=0,\ldots,m-1
  \]
- each bucket is equally likely to be occupied
- probability that two keys map to the same slot is \( \frac{1}{m} \)

\[
P(h(x) = h(y)) = \frac{1}{m} \quad \text{x, y - independent random variable}
\]
Good hash function

**Need**: qualitative information about $P$

**uniform distributed keys**

**Example**:

- keys are random real numbers independently and uniformly distributed in the range $[0,1)$.
  \[ h(k) = \lfloor k \times m \rfloor \]
  satisfies the simple uniform hashing property
- keys are random integers independently and uniformly distributed in the range $0$ to $N-1$
  where $N$ much larger than $m$
  \[ h(k) = k \mod m \]
  satisfies the simple uniform hashing property
Hash function

Need: qualitative information about $P$

Special-purpose hash function

• exceptionally good for a specific kind of data
  no performance on data with different distribution

Example (1)

input data: file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc., with mostly sequential numbers.

• extracts the numeric part $k$ of the file name $fn$

\[ h(fn) = \text{numeric\_part}(fn) \mod m \]
Hash function

Example (2)
input data: text in any natural language

has highly non-uniform distributions of characters, and character pairs, very
characteristic of the language

• string
• variable length data

it is prudent to use a hash function that depends on all characters of the
string—and depends on each character in a different way

Example of hash function:

Function HashMultiplicative(strKey) {
    hash = \texttt{INITIAL\_VALUE};
    for i = 1, length(strKey) do
        hash = \texttt{M} \times hash + strKey [i]
    endfor
    return hash \mod \texttt{TABLE\_SIZE};
}

D. Bernstein, \texttt{comp.lang.c}, (1991 ?) \quad \texttt{INITIAL\_VALUE} = 5381 \quad \texttt{M} = 33

B. Kernighan, D. Ritchie, \textit{The C Programming Language}, 1978 \quad \texttt{INITIAL\_VALUE} = 0 \quad \texttt{M} = 31
Hash function

Example (3)
input data: an unchanging dictionary
  (text in a natural language)

If the dictionary is unchanging, you might want to consider perfect hashing;
• for a given dataset you can guarantee that there will be no collisions
Hash function

Example (4)
assume
input data: three-letter words
formed with any of a set of char extended ASCII code

perfect hashing
• $h(\text{str}) = \text{ASCIIcode}([\text{str}[0]]) \times 256^2$
  $+ \text{ASCIIcode}([\text{str}[1]]) \times 256^1$
  $+ \text{ASCIIcode}([\text{str}[2]])$

• ASCIIcode([str[i]]): values from range 0..255
• hash table of size $3^{256}$
Java

HashMap
- Hash table based implementation of the Map interface

HashSet
- implements the Set interface, backed by a hash table
Hash in programming languages

Java Object

• public int hashCode()

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

• public boolean equals(Object obj)

  if two objects are equal then they must return same hash code
  - that is compared by equal() of that class
Hash in programming languages

The `java.lang.String` hash function

Given: `s` of `java.lang.String`

\[ h(s) = s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1] \]

- uses arithmetic `int`

where `s[i]` is the `i`th character of the string,

`n` is the length of the string

\(^\text{^ indicates exponentiation.}\)

(The hash value of the empty string is zero.)
Hash tables in programming languages

• STL map: Associative key-value pair held in balanced binary tree structure
  – usually a red-black tree

New in C++ 11

• unordered_map

Some implementations

• hash_map was a common extension provided by many library implementations