MPEI

Funções de dispersão

(*Hash functions*)

Motivação

- Em muitos programas de computador torna-se necessário aceder a informação através de uma chave
  - Exemplo:
    - Obter nome associado a um número de telefone

- Em Java, por exemplo, temos estruturas de dados como HashMap e Hashtable

Um dicionário simples: `Hashtable`

- Para criar uma *Hashtable*:

```
import java.util.*;
Hashtable table = new Hashtable();
```

- Para colocar elementos (par chave-valor) na Hashtable, usa-se:

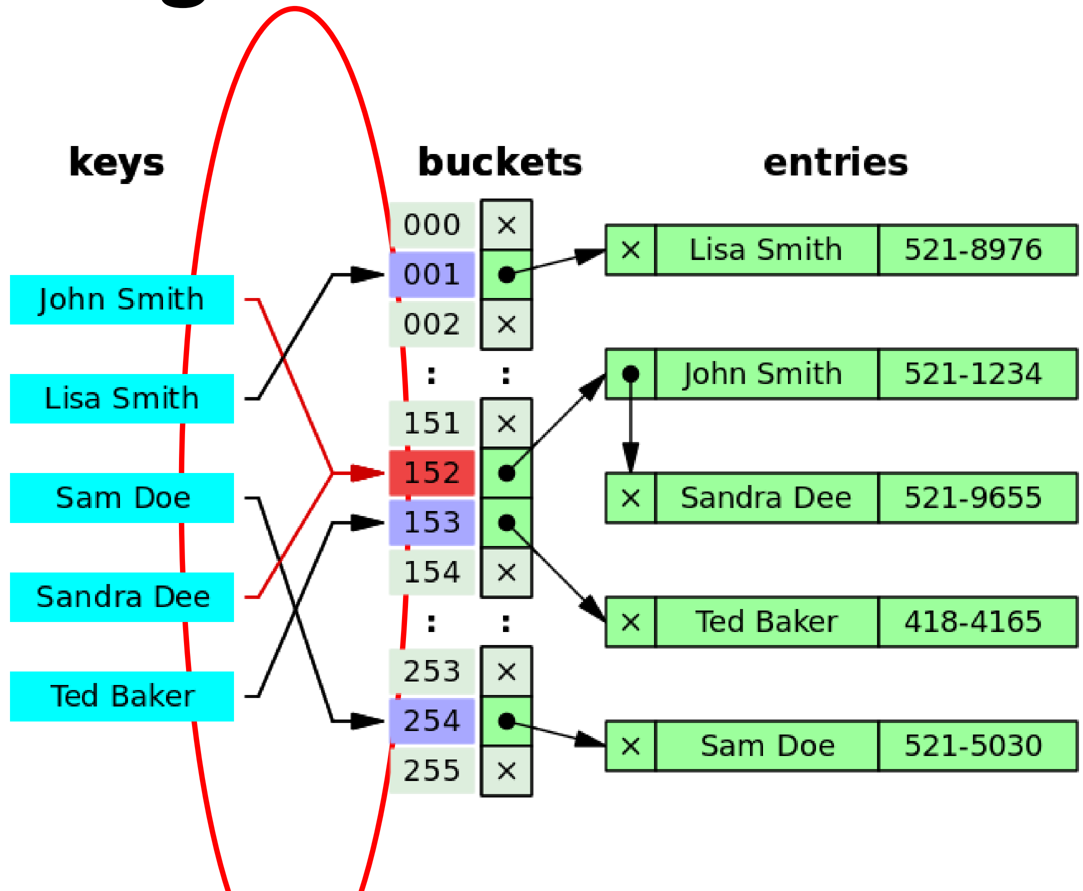
```
table.put(chave, valor);
```

- Para obter um valor:

```
valor = table.get(chave);
```

Implementação comum
Separate chaining with linked lists

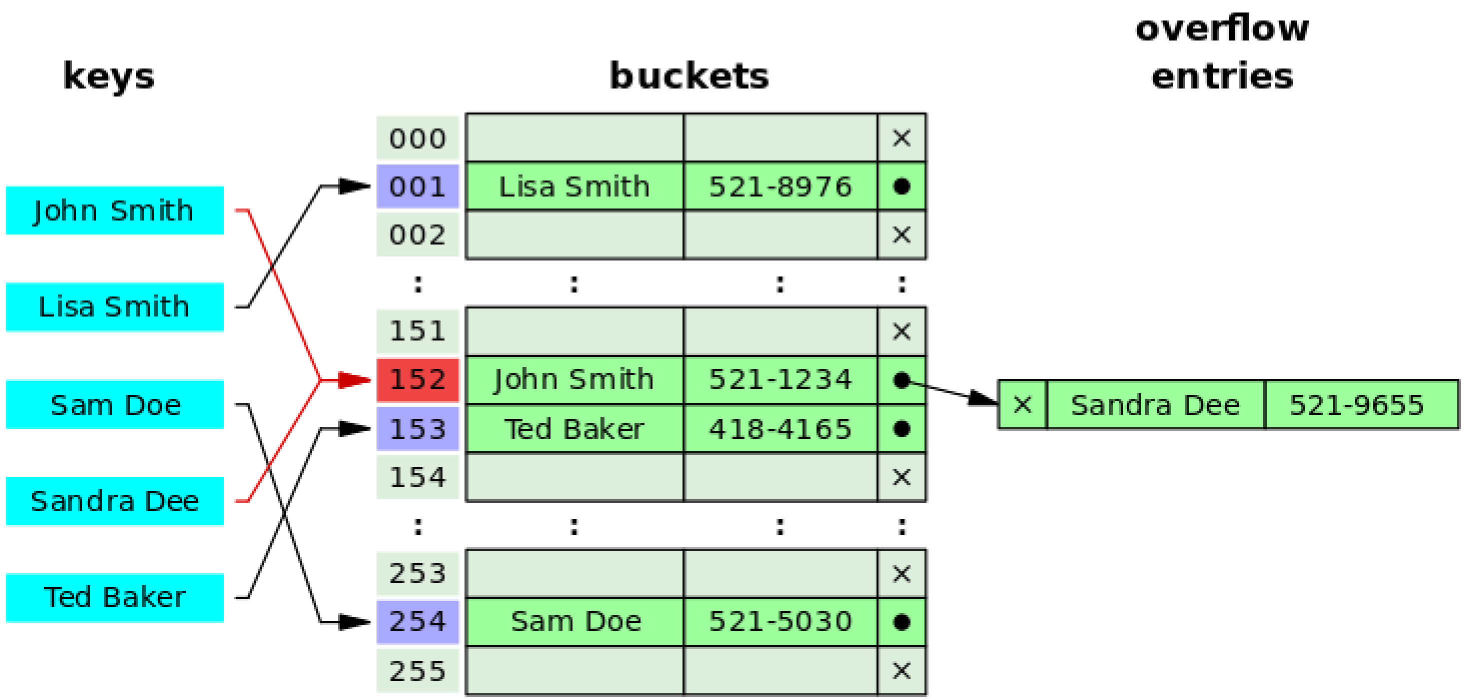
- As chaves são transformadas em posições num array
  - usando uma função
- Cada posição do array é o início de uma lista ligada



Função de Dispersão / *Hash Function*

Outra implementação

Separate chaining with list head cells



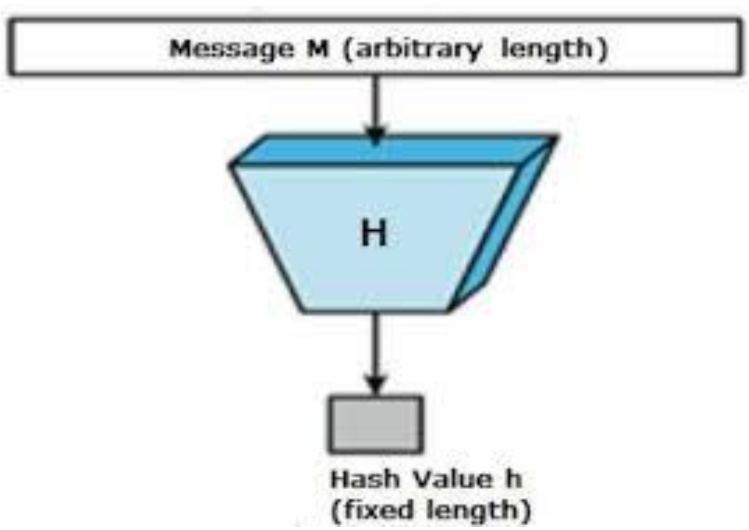
Função de dispersão

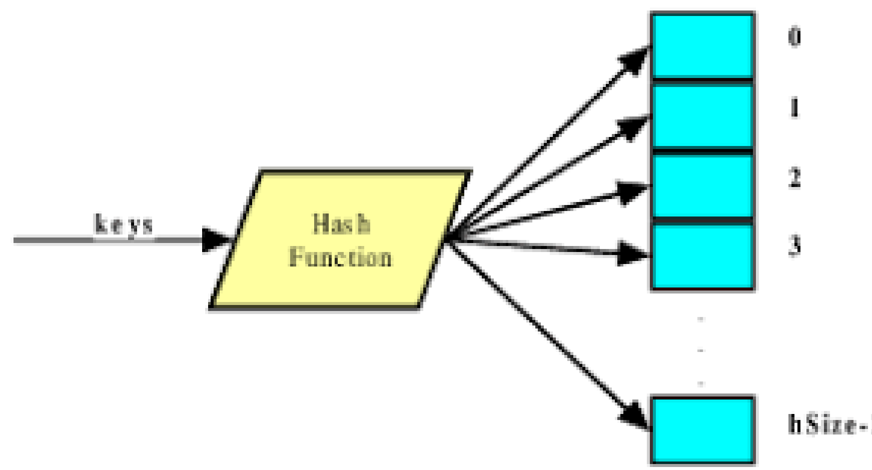
- Em termos gerais, uma função de dispersão -- em Inglês *hash function* - é qualquer algoritmo que **mapeia um conjunto grande e de tamanho variável para um conjunto de tamanho fixo de menor dimensão**

- É, como veremos, essencial para muitas aplicações

Função de dispersão / Hash function

- Uma função de dispersão (hash function) **mapeia** símbolos de um **universo U** num conjunto de M valores, em geral inteiros





- Processo pode ser visto como a atribuição de uma posição num vetor de M posições, entre 0 e M-1, a cada símbolo.
  – As posições designam-se nuitas vezes por *buckets*

Hash Code

- O conjunto dos símbolos efetivamente usados numa determinada aplicação é, em geral, apenas uma parte do universo de valores (U) pelo que faz todo o sentido usar um **valor de M muito menor do que a dimensão de U**
  - Muitas vezes os valores designam-se por chaves

- Uma função de dispersão recebe um elemento de U como entrada e devolve um número inteiro h no intervalo $0, \dots, M-1$
  - h é o Código de dispersão (em Inglês **hash code**)

Funções de dispersão / Hash functions

- Qualquer função que mapeie uma chave do universo U no intervalo $0..M-1$ é uma função de dispersão em potencial.
- No entanto, uma tal função só é eficiente se distribuir as chaves pelo intervalo de uma forma razoavelmente uniforme
  - mesmo quando existem regularidades nas chaves.

- Uma função de dispersão ideal mapeia as chaves em inteiros de uma forma aleatória
  - De forma a que as *keys* sejam igualmente distribuídos pelos *buckets.*

- É fundamental que a função de dispersão seja uma função no sentido matemático do termo,
  - Isto é, que para cada chave a função devolva sempre o mesmo código

Funções de dispersão / Hash functions

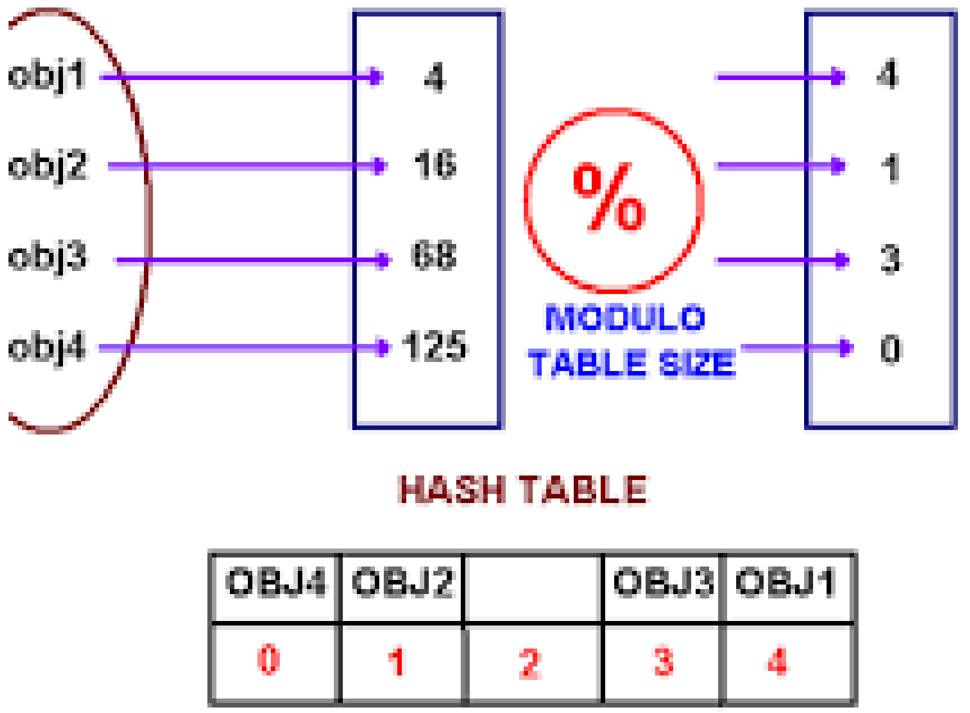
- Uma função de dispersão ideal mapeia as chaves em inteiros de uma forma aleatória,
  - De forma a que os valores sejam igualmente distribuídos, mesmo quando existem regularidades nas chaves

- As funções de dispersão são tipicamente transformações matemáticas pseudo-aleatórias, existindo uma grande variedade, com diferentes graus de complexidade e diferentes desempenhos
  - Em geral o desempenho depende da aplicação pelo que é recomendável testar várias.

Funções de dispersão

- O processo pode ser dividido em dois passos:

1. Mapeamento do elemento para um inteiro
2. Mapeamento do inteiro para um conjunto limitado (de inteiros).



Notação

- Adopta-se para a representação das funções de dispersão
  $h()$
  - do Inglês hash function

- e k para uma chave
  - do Inglês key

Funções de dispersão - colisões

- Como o número de elementos de U é em geral maior que M, é inevitável que a função de dispersão mapeie vários elementos diferentes no mesmo valor de h, situação em que dizemos ter havido uma **colisão**

- Por exemplo, sendo k um elemento de U e a função de dispersão:

$$h(k, M) = k \; mod \; M$$

- teremos colisões para $k, M + k, 2M + k, \ldots.$

Exemplo muito simples

Considere o universo U é o conjunto dos números inteiros que vai de 100001 a 9999999. Suponha que $M = 100$ e se adota os dois últimos dígitos da chave como código de dispersão (em outras palavras, o código é o resto da divisão por 100). Calcule os códigos (*hash codes*) para 123456, 7531 e3677756.

Resultado:

chave	código
123456	56
7531	31
3677756	56

Propriedades

- Requer-se, em geral, que as funções de dispersão satisfaçam algumas propriedades, como:

- Serem determinísticas

- **Uniformidade**:
  - Uma boa função de dispersão deve mapear as entradas esperadas de forma igual por toda a gama de valores possíveis para a sua saída

  - Todos os valores possíveis para a função de dispersão devem ser gerados com aproximadamente a mesma probabilidade

Funções de dispersão para inteiros

- Estas funções mapeiam uma única chave inteira k num número inteiro $h(k)$ entre M possíveis

- Existem vários tipos:
  - baseadas em divisão
  - baseadas em multiplicação
  - membros de famílias universais.

Método da Divisão

- Utiliza o resto da divisão por M
- A fução de de dispersão é

$$h(k) = k \; mod \; M$$

  - M é o número de posições (igual ao tamanho da tabela), que deve ser um número primo
- Exemplo: se $M = 12$ e a chave $k = 100$ temos $h(k) = 4$

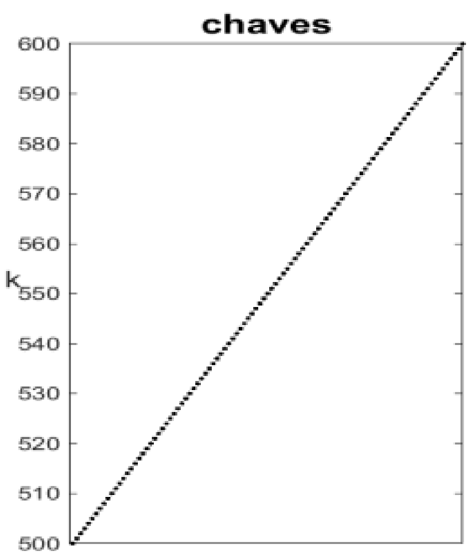
- Método bastante rápido
  - Requer apenas uma operação de divisão

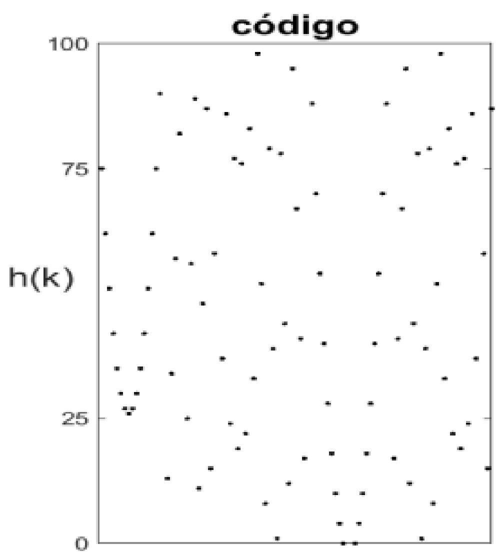
- Funciona muito mal para muitos tipos de padrões nas chaves
- Foram desenvolvidas variantes como a de Knuth:

$$h(k) = k(k+3) \; mod \; M$$

Exemplo: Variante de Knuth

- $h(k) = k(k + 3) \mod M$
- $M = 113$
- Aplicação a todos os inteiros de 500 a 600.





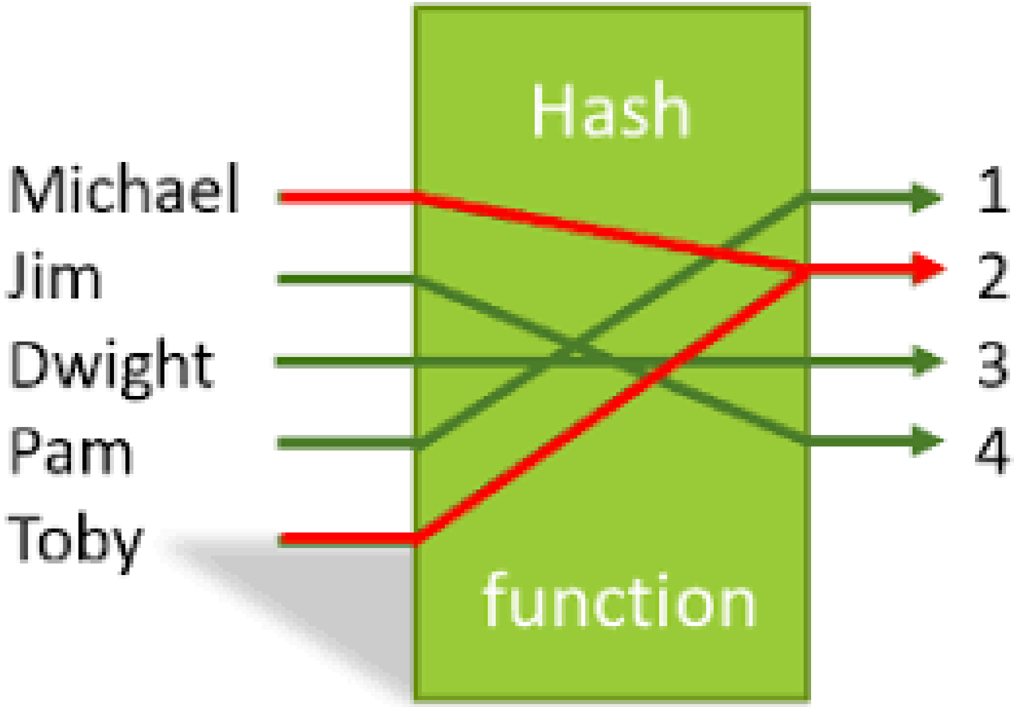
- A sequência igualmente espaçada de números (à esquerda) é dispersada sem regularidade aparente
  – que é o que se pretende de uma boa função de dispersão

Método da multiplicação

- Este método opera em duas etapas:
  - primeiro, multiplica-se a chave por uma constante A, $0 < A < 1$, e extrai-se a parte fraccionária de kA;
  - de seguida,multiplica-se por M e arredonda-se para o maior inteiro menor ou igual ao valor obtido
- Matlab:

```
function h = hmultiplic(k,M)
% Função de dispersão para  baseada na multiplicação
% Entradas:   k - chave;
%             M  - núm. de valores possíveis [0,M-1]

A= 0.5*(sqrt(5) - 1);

h=floor(M*(mod(k*A,1)));
```

Função de dispersão de uma sequência de caracteres



Função de dispersão de uma sequência de caracteres

- Uma função de dispersão para cadeias de caracteres (strings) calcula a partir desta, qualquer que seja o seu tamanho, um inteiro
- Como sabemos uma sequência de caracteres (String) é em geral representada como uma sequência de inteiros
- Em consequência, a função de dispersão para Strings tem por entrada uma sequência de inteiros

$$k = k1, \dots, ki, \dots., kn$$

- e produz um número inteiro pequeno $h(k)$

- Os algoritmos para este tipo de entrada assumem que os inteiros são de facto códigos de caracteres

Função de dispersão de uma sequência de caracteres

- Os algoritmos para este tipo de entrada fazem em geral o uso do seguinte:
- Em muitas linguagens um caracter é representado em 8 bits
- O código ASCII apenas usa 7 desses 8 bits
- Desses 7, os caracteres comuns apenas usam os 6 menos significativos
  - E o mais significativo desses 6 indica essencialmente se é maiúscula ou minúscula, muitas vezes pouco relevante

- Em consequência os algoritmos concentram-se na preservação do máximo de informação dos 5 bits menos significativos, fazendo muito menos uso dos 3 bits mais significativos

Função de dispersão de uma sequência de caracteres (String)

- Em geral, o processamento efetuado consiste em:
  - inicializar h com 0 ou outro valor inicial
  - Percorrer a sequência de inteiros (representando os caracteres) combinando os inteiros ki, um por um, com h
    - Os algoritmos diferem na forma como combinam ki com h
  - Obtenção do resultado final através de h mod M (método da divisão).

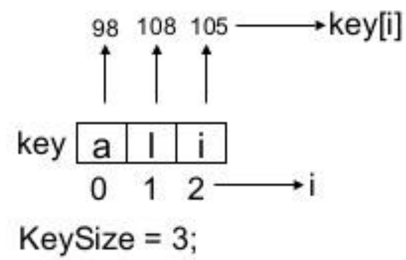
- Para evitar ao máximo problemas com overflow, em geral os inteiros ki são representados por números inteiros sem sinal (unsigned int)
  - A utilização de representações de inteiros com sinal pode resultar em comportamentos estranhos

Exemplo simples

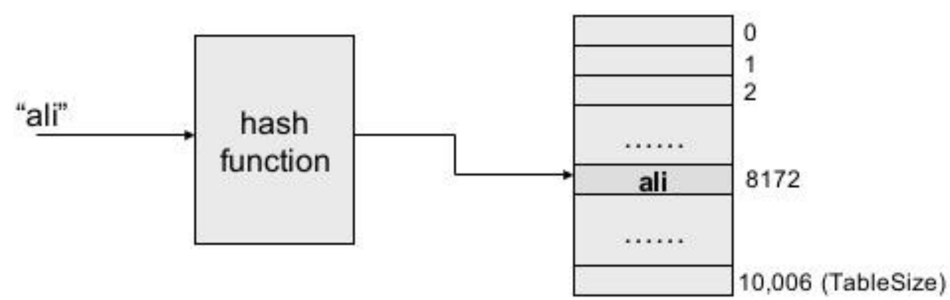
$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

```
int hash (const string &key, int tableSize)
{
   int hashVal = 0;

   for (int i = 0; i < key.length(); i++)
      hashVal = 37 * hashVal + key[i];

   hashVal %=tableSize;
   if (hashVal < 0)   /* in case overflows occurs */
      hashVal += tableSize;

   return hashVal;
};
```

Hash function for strings:



hash("ali") = (105 * 1 + 108*37 + 98*37²) % 10,007 = 8172



CENG 213 Data Structures 12

Exemplo – hashCode() do Java

- A classe java.lang.String implementa desde o Java 1.2 a função hashCode() usando um somatório de produtos envolvendo todos os caracteres
- Uma instância s da classe java.lang.String tem o seu código h(*s*) definido por:

$$h(s) = \sum_{i=0}^{n-1} s\,[\,i\,] \cdot 31^{n-1-i}$$

- com *s*[*i*] representando o código UTF-16 do caracter *i* da cadeia de comprimento *n*
- A adição é efectuada usando 32 bits

Alguns algoritmos – variante CRC

- Fazer um shift circular de 5 bits para a esquerda ao h.
- De seguida fazer XOR de h com ki.

CRC variant: Do a 5-bit left circular shift of h. Then XOR in ki. Specifically:

```
highorder = h & 0xf8000000    // extract high-order 5 bits from h
                              // 0xf8000000 is the hexadecimal representation
                              //   for the 32-bit number with the first five
                              //   bits = 1 and the other bits = 0
h = h << 5                    // shift h left by 5 bits
h = h ^ (highorder >> 27)     // move the highorder 5 bits to the low-order
                              //   end and XOR into h
h = h ^ ki                    // XOR h and ki
```

Alguns Algoritmos– PJW hash

- Deslocar (shift) h 4 bits para a esquerda
- Adicionar ki
- Mover 4 bits mais significativos

PJW hash (Aho, Sethi, and Ullman pp. 434-438): Left shift h by 4 bits. Add in ki. Move the top 4 bits of h to the bottom. Specifically:

```
// The top 4 bits of h are all zero
h = (h << 4) + ki                // shift h 4 bits left, add in ki
g = h & 0xf0000000               // get the top 4 bits of h
if (g != 0)                      // if the top 4 bits aren't zero,
   h = h ^ (g >> 24)             //   move them to the low end of h
   h = h ^ g
// The top 4 bits of h are again all zero
```

Exemplo de uma função completa

- Mapeia uma string de comprimento arbitrário num inteiro (>=0)
- DJB31MA

```
uint hash(const uchar* s, int len, uint seed)
{
   uint h = seed;
   for (int i=0; i < len; ++i)
       h = 31 * h + s[i];
   return h;
}
```

  – Fonte: Paulo Jorge Ferreira “MPEI – summary” 2014

Exemplo Matlab

function hash=string2hash(str,type)

```
% This function generates a hash value from a text string
%
% hash=string2hash(str,type);
%
% inputs,
%   str : The text string, or array with text strings.
% outputs,
%   hash : The hash value, integer value between 0 and 2^32-1
%   type : Type of has 'djb2' (default) or 'sdbm'
%
% From c-code on : http://www.cse.yorku.ca/~oz/hash.html
.....
```

- From: http://www.mathworks.com/matlabcentral/fileexchange/27940-string2hash/content/string2hash.m

Exemplo Matlab

```
str=double(str);

hash = 5381*ones(size(str,1),1);

for i=1:size(str,2),
    hash = mod(hash * 33 + str(:,i), 2^32-1);
end
```

Exemplos de uso ($M = 11$):

```
k = António        ->  h(k) = 4
k = Antónia        ->  h(k) = 1
k = Manuel         ->  h(k) = 6
k = Manu           ->  h(k) = 4
k = Manuela        ->  h(k) = 0
k = Vitor          ->  h(k) = 0
```

Problemas

- As funções de dispersão terão que lidar com conjuntos $S \subseteq U$ com $|S| = n$ chaves não conhecidos de antemão
- Normalmente, o objetivo destas funções é obter um número baixo de colisões (chaves de S que mapeiam na mesma posição)

- Uma função de dispersão determinística (fixa) não pode oferecer qualquer garantia de que não ocorrerá o pior caso:
  - um conjunto S com todos os elementos a serem mapeados na mesma posição, tornando a função de dispersão inútil em muitas situações.
- Além disso, uma função determinística não pode ser alterada facilmente em situações em que ocorram muitas colisões.

Solução

- A solução para estes problemas consiste em **escolher uma função aleatoriamente de uma família de funções**,

- Têm particular interesse as famílias de funções universais.

Funções de dispersão universais

- Uma família *H* de funções de dispersão *h* é universal se:

$$\forall x, y \in U,\ x \neq y: \quad P_{h \in H}[h(x) = h(y)] \leq \frac{1}{M}$$

- Por palavras…
- **quaisquer duas chaves do universo colidem com probabilidade máxima igual a $1/M$** quando a função de dispersão h é extraída aleatoriamente de H
  - exatamente a probabilidade de colisão esperada caso a função de dispersão gerasse códigos realmente aleatórios para cada chave.

Funções de dispersão universais

- Esta solução garante um baixo número de colisões em média, mesmo no caso de os dados serem escolhidos por alguém interessado na ocorrência do pior cenário (ex: *hacker*).
- Este tipo de funções pode utilizar mais operações do que as funções que vimos anteriormente

- Existe uma diversidade de famílias universais e métodos para as construir
  – Veremos a seguir alguns

Método de Carter Wegman

- A proposta original, de Carter e Wegman, consiste em escolher um primo $p \geq M$ e definir

$$h_{a,b}(x) = ((ax + b) \, mod \, p) \, mod \, M$$

- sendo *a* e *b* inteiros aleatórios módulo *p*

- Trata-se de uma iteração de um gerador de números aleatórios de congruência linear.

Método da Matriz

- Este método baseia-se em:

1. considerar as **chaves na sua representação binária**
2. construir uma matriz de bits aleatoriamente
3. multiplicar a chave e matriz

Método da Matriz (continuação)

- Consideremos que as chaves são representáveis por u bits
- Sendo M uma potência de 2 : $M = 2^b$
- Criar uma matriz h de 0s e 1s de forma aleatória
  - a matriz terá dimensões b x u

- Definir $h(x) = hx$
  - usando adição mod 2

- Examplo:
  - u = 4
  - b= 3

$$\begin{array}{cccc} h & x & & h(x) \\ \begin{bmatrix}1&0&0&0\\0&1&1&1\\1&1&1&0\end{bmatrix} & \begin{bmatrix}1\\0\\1\\0\end{bmatrix} & = & \begin{bmatrix}1\\1\\0\end{bmatrix}\end{array}$$

O que significa hx ?

- Pode ser interpretada como a adição de algumas das colunas de h, aquelas em que *x* tem o valor 1 nas linhas

$$\begin{array}{cccc} h & x & & h(x) \\ \begin{bmatrix}1&0&0&0\\0&1&1&1\\1&1&1&0\end{bmatrix} & \begin{bmatrix}1\\0\\1\\0\end{bmatrix} & = & \begin{bmatrix}1\\1\\0\end{bmatrix}\end{array}$$

- No exemplo:
- A 1ª e 3ª coluna são somadas
    - 1 + 0 = 1
    - 0 + 1 = 1
    - 1 + 1 = 0

Método da matriz

Propriedade. *A função de dispersão h*(*x*) *definida desta forma terá:*

$$\forall\, x \neq y, \quad P_{h \in H}[h(x) = h(y)]) = \frac{1}{M} = \frac{1}{2^b}$$

Demonstração:

- Um par de chaves diferentes $x\ e\ y$ difere em algum dos bits. Consideremos que diferem no bit na posição *i* e que $x_i = 0$ e $y_i = 1$.
- Se selecionarmos toda a matriz *h* exceto a coluna *i* obteremos um valor fixo para $h(x)$;
- No entanto, cada uma das 2^b diferentes possibilidades da coluna *i* implica um valor diferente para *h*(*y*), pois sempre que se muda um valor nessa coluna muda o bit correspondente em *h*(*y*);
- Em consequência temos exatamente a probabilidade $1/2^b$ de $h(x) = h(y)$.

Outro método

- Mais eficiente do que o da matriz
- A chave é representa por um **vetor de inteiros**
    - Em vez do vetor de bits do método da matriz

$[x_1, x_2, \ldots, x_k]$

    - x_i pertencendo a $\{0,1,\ldots,M-1\}$
    - k é o tamanho do vetor
    - M um número primo

- Exemplo:
  - Em Strings, x_i pode representar o código do caracter i

Outro método (continuação)

- Para seleccionar uma função de dispersão h escolhem-se k números aleatórios

$$r_1, r_2, \dots, r_k \quad \text{de } \{0,1,\dots,M-1\}$$

- E define-se :

$$h(x) = (r_1 x_1 + r_2 x_2 + \ \dots + r_k\, x_k)\ mod\ M$$

Exemplo Matlab

```
s='Métodos Probabilísticos'
M=  113;

% converter para vetor
x=double(s)

% gerar vetor r
r=randi(M-1,1,length(x))

% h(x) =  r * x  mod M
h=mod( r* x', M)
```

Demonstração da universalidade

- A demonstração segue a mesma linha da apresentada anteriormente para o método da matriz

- Considere-se duas chaves distintas x e y
- Pretendemos demonstrar que

$$P[h(x) = h(y)] \leq 1/M$$

Demonstração da universalidade

Como $x \neq y$, existe pelo menos um indíce i tal que $x_i \neq y_i$. Selecionando todos os números aleatórios r_j com $j \neq i$ podemos definir $h'(x) = \sum_{j \neq i} r_j x_j$.

Desta forma, ao escolher um valor para r_i teremos $h(x) = h'(x) + r_i x_i$.

Teremos uma colisão entre x e y exatamente quando

$$h'(x) + r_i x_i = h'(y) + r_i y_i \ mod \ M$$

ou, de forma equivalente, quando

$$r_i(x_i - y_i) = h'(y) - h'(x) \ mod \ M \ .$$

Demonstração da universalidade

Como M é primo, a divisão por um valor não nulo módulo M é possível e existe apenas um único valor $r_i \ mod \ M$ que constitui a solução, mais exactamente

$$r_i = \frac{h^{'}(y) - h^{'}(x)}{x_i - y_i} \ mod \ M$$

Temos assim apenas uma possibilidade de igualdade entre 1 e $M - 1$ e, em consequência, a probabilidade de colisão é exatamente $1/M$, como pretendíamos demonstrar.

$\square$

Exemplo em Matlab

```
%
function InitHashFunction(this)
    % Set prime parameter
    ff = 1000; % fudge factor
    pp = ff * max(this.m + 1,76);
    pp = pp + ~mod(pp,2); % make odd
    while (isprime(pp) == false)
        pp = pp + 2;
    end
    this.p = pp; % sufficiently large prime number

    % Randomized parameters
    this.a = randi([1,(pp - 1)]);
    this.b = randi([0,(pp - 1)]);
    this.c = randi([1,(pp - 1)]);
end
```

Exemplo em Matlab - HashCode()

```
    function hk = HashCode(this,key)
        % Convert character array to integer array
        ll = length(key);
        if (ischar(key) == false)
            % Non-character key
            HashTable.KeySyntaxError();
        end
        key = double(key) - 47; % key(i) = [1,...,75]

        %
        % Compute hash of integer vector
        %
        % Reference: http://en.wikipedia.org/wiki/Universal_hashing
        %            Sections: Hashing integers
        %                      Hashing strings
        %
        hk = key(1);
        for i = 2:ll
            % Could be implemented more efficiently in practice via bit
            % shifts (see reference)
            hk = mod(this.c * hk + key(i),this.p);
        end
        hk = mod(mod(this.a * hk + this.b,this.p),this.m) + 1;
    end
end
```

Como ter *n* funções de dispersão ?

Possíveis soluções:

1. Ter mesmo *n* funções diferentes
2. Usar funções customizáveis (definindo uma família de funções) e usando parâmetros diferentes
3. Usar a mesma função de dispersão e processar a chave por forma a ter *n* chaves diferentes baseadas na chave original

Exemplo (Matlab):

```
for i=1:n
    str= [str num2str(i)];
    h=HashCode(hash,m,str);
end
```

Propriedades (continuação)

- As *n* funções de dispersão devem cumprir um requisito adicional:
- Produzir resultados **não-correlacionados**
- Esta propriedade é muito importante e é aconselhável verificá-la/avaliá-la em trabalhos envolvendo várias funções

“Teste” de funções de dispersão

- Um teste simples e básico consiste em:

1. Gerar um conjunto grande de chaves (pseudo)aleatórias

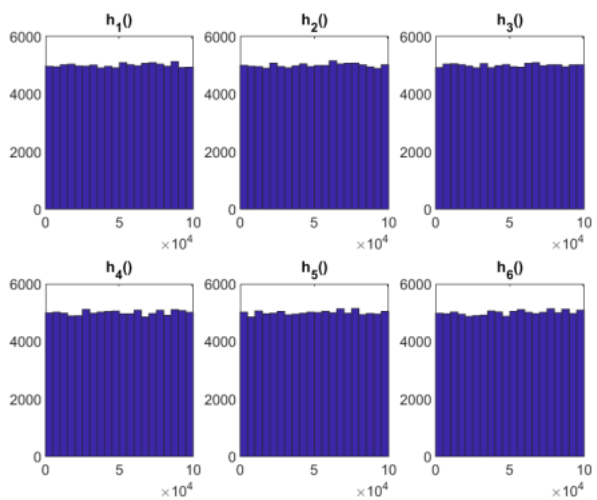
2. Processar todas essas chaves com as *n* funções de dispersão
   - Guardando os resultados produzidos (*hash codes*)

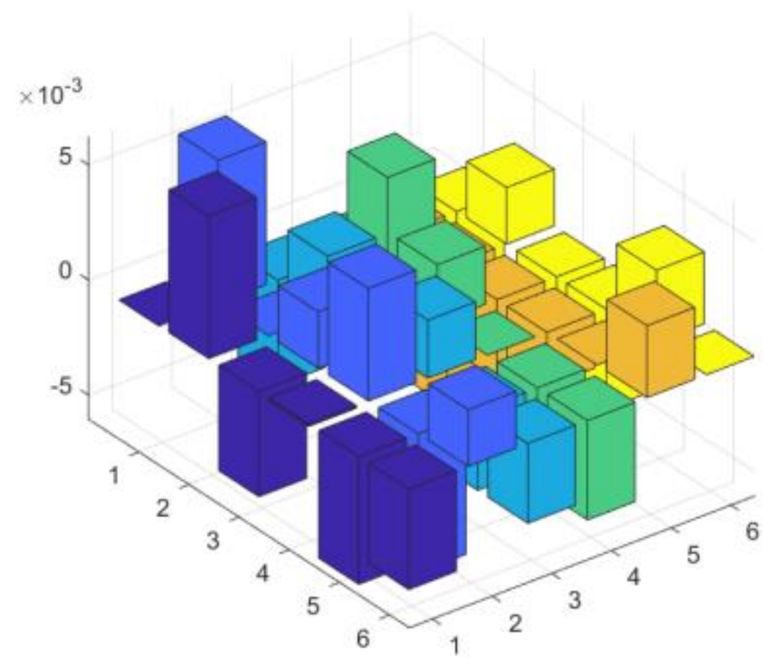
3. Analisar o histograma de cada função de dispersão
   - Para verificar a uniformidade da distribuição dos *hash codes*

4. Calcular, visualizar e analisar as correlações entre os resultados produzidos pelas várias funções de dispersão

Exemplo

- Teste com 100 mil números de 6 funções (h1, . . . , h7)





Funções de dispersão criptográficas

- Para este tipo de funções (cryptographic hash functions) M é um número exponencialmente grande
  - Como 2^{256}
- A tabela seria maior que o número de eletrões no universo!
- Mas não temos a tabela... $h(k)$ é apenas uma impressão digital (fingerprint) de k.

- Mesmo para *M* com valores muito grandes os indíces de h(k) são pequenos
  - Ex: apenas 256 bits (32 bytes) para $M = 2^{256}$

- **A principal propriedade** requerida para uma função de dispersão criptográfica é de que seja computacionalmente intratável para alguém descobrir $y \neq x$ tal que $h(y) = h(x)$

cryptographic hash functions

- For cryptographic hash functions M is as an exponentially large number, like 2^256.
- The table would be bigger than the # electrons in the universe!
- But we don’t have the table. Instead, h(x) is a "fingerprint" of x.

- Note that even for M very big (like 2^256) the indices h(x) are fairly small
  - e.g., only 256 bits (32 bytes)

- The main property you want for a cryptographic hash function is that given x, it should be computationally intractable for anyone to find $y \neq x$ such that $h(y) = h(x)$

Alguns links

- http://www.mathworks.com/matlabcentral/fileexchange/27940-string2hash/content/string2hash.m
- http://www.mathworks.com/matlabcentral/fileexchange/45123-data-structures/content/Data%20Structures/Hash%20Tables/HashTable.m
- http://www.cse.yorku.ca/~oz/hash.html
- http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed
- https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html
- https://en.wikipedia.org/wiki/Hash_function#Properties
- https://en.wikipedia.org/wiki/Universal_hashing
- http://www.i-programmer.info/programming/theory/2664-universal-hashing.html

Funções de Dispersão Universais

- https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1004.pdf
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-8-universal-hashing-perfect-hashing/lec8.pdf
- http://cs-www.bu.edu/faculty/homer/537/talks/SarahAdelBargal_UniversalHashingnotes.pdf