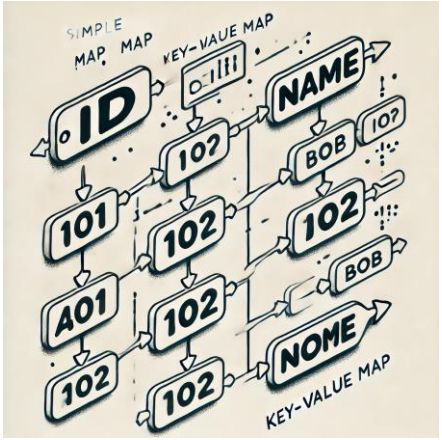
MPEI 2024-2025

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

Um dicionário simples: `Hashtable`

- Em Java, por exemplo, temos estruturas de dados como HashMap e 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);
```

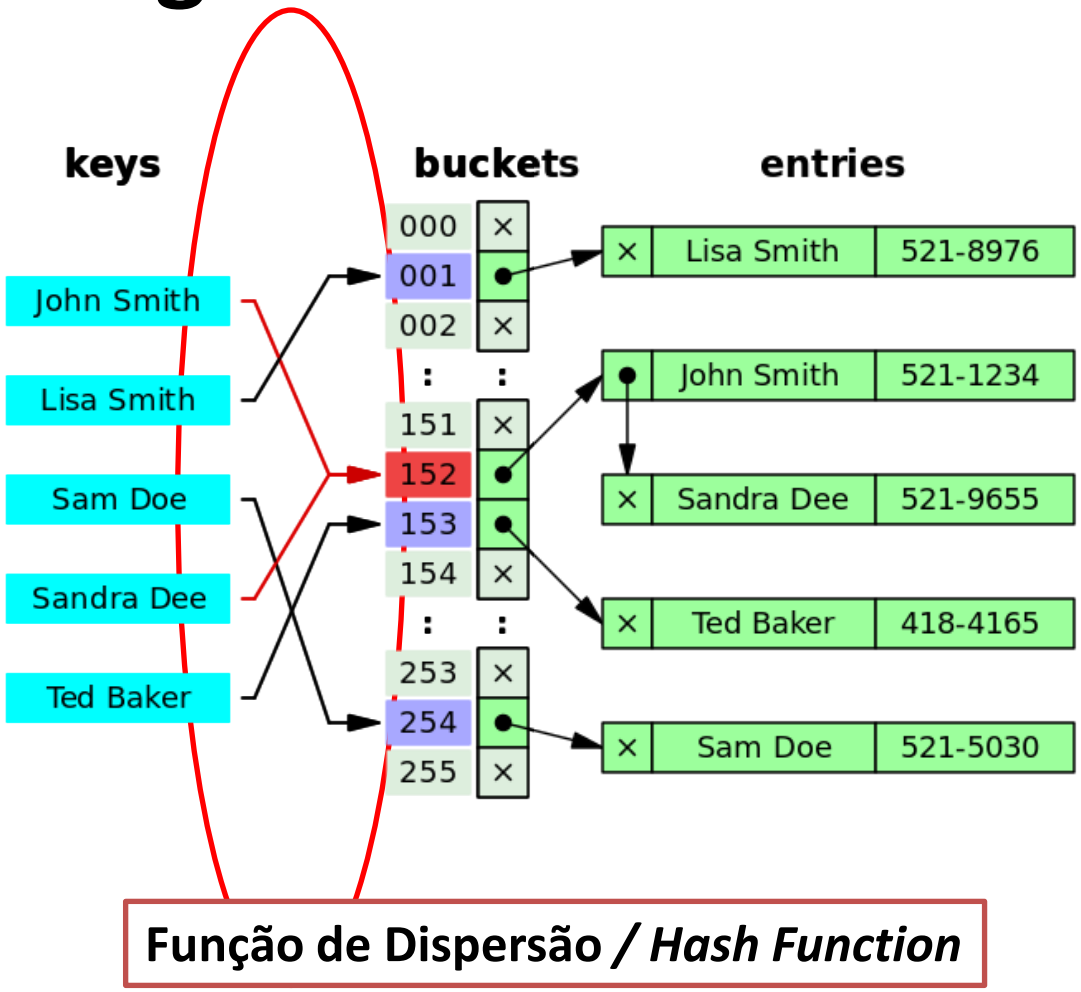
- Para obter um valor:

```
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



Implementação da Hashtable Java

- Fragmento do Código fonte:

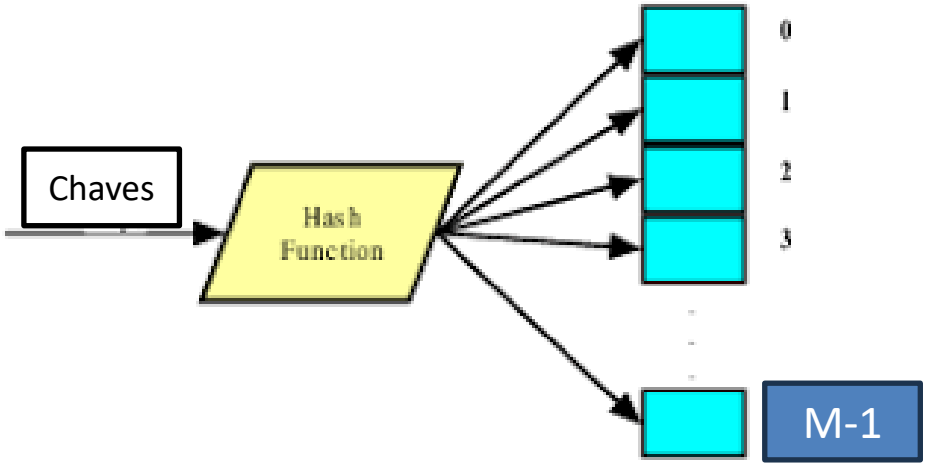
```
public synchronized V put(K key, V value) {
      …
      Entry<?,?> tab[] = table;
      int hash = key.hashCode();
      int index = (hash & 0x7FFFFFFF) % tab.length;

      Entry<K,V> entry = (Entry<K,V>)tab[index];
      for(; entry != null ; entry = entry.next) {
               if ((entry.hash == hash) && entry.key.equals(key)) {
                        V old = entry.value;
                        entry.value = value;
                        return old;
               }
       }
       addEntry(hash, key, value, index);
       return null
}
```

Função de dispersão

- Em Inglês *hash function*
- Qualquer algoritmo que **mapeia um conjunto grande e de tamanho variável para um conjunto de tamanho fixo de menor dimensão**
- Muitas vezes os elementos do conjunto grande designam-se por chaves
- É, como veremos, essencial para muitas aplicações

Função de dispersão / Hash function

- **Mapeia** símbolos de um **universo U** num conjunto de valores
  - em geral inteiros



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

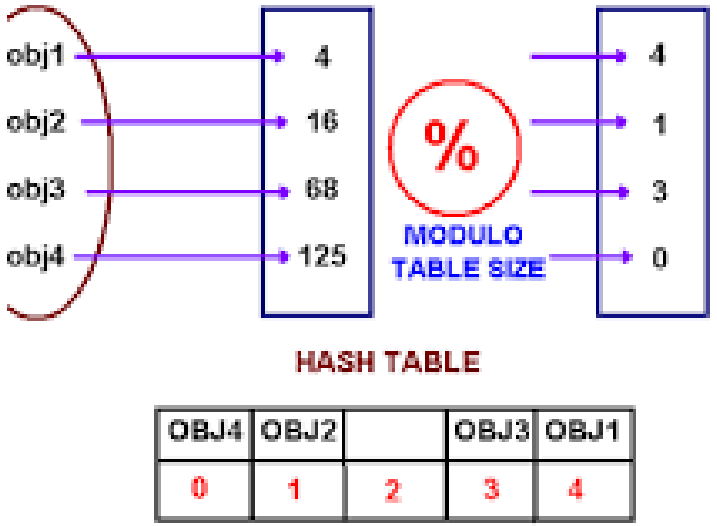
Hash Code

- Usa-se um **valor de *M* muito menor do que a dimensão de U**
  - Porque o conjunto dos símbolos efetivamente usados é, em geral, apenas uma parte do universo de valores (U)

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

Funções de dispersão / Hash functions

- 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 $h()$
  - do Inglês hash function

- e k para uma chave
  - do Inglês key

Propriedades

- As funções de dispersão devem ter as seguintes propriedades:

- Serem determinísticas
  - para cada chave a função deve devolver sempre o mesmo código

- **Uniformidade**:
  - Uma função de dispersão ideal mapeia as chaves em inteiros de uma forma "aleatória"
    - De forma a que as *keys* sejam igualmente distribuídas pelos *buckets / hash codes.*

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**

- Exemplo:
  - sendo k um elemento de U e a função de dispersão:

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

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

Colisão - 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

Funções de dispersão / Hash functions

- Existe uma grande variedade
  - com diferentes graus de complexidade e diferentes desempenhos
  - para diferentes tipos de dados
    - inteiros, strings …

- Em geral o desempenho depende da aplicação
  - pelo que é recomendável testar várias

Funções de dispersão **para inteiros**

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

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

Método da Divisão

- Utiliza o resto da divisão por M
- A função 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 $M = 11$ e a chave $k = 100$ temos $h(k) = 1$

- Método bastante rápido
  - Requer apenas uma operação de divisão

Método da Divisão

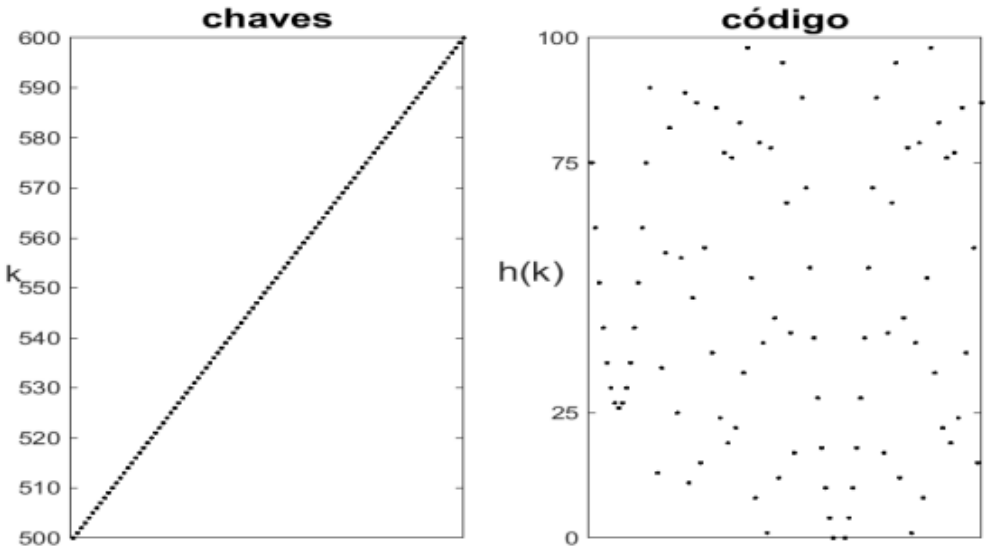
- Funciona muito mal para muitos tipos de padrões nas chaves

- Foram desenvolvidas variantes como a de Knuth:

$$h(k) = k(\boldsymbol{k} + \mathbf{3}) \, mod \, M$$

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

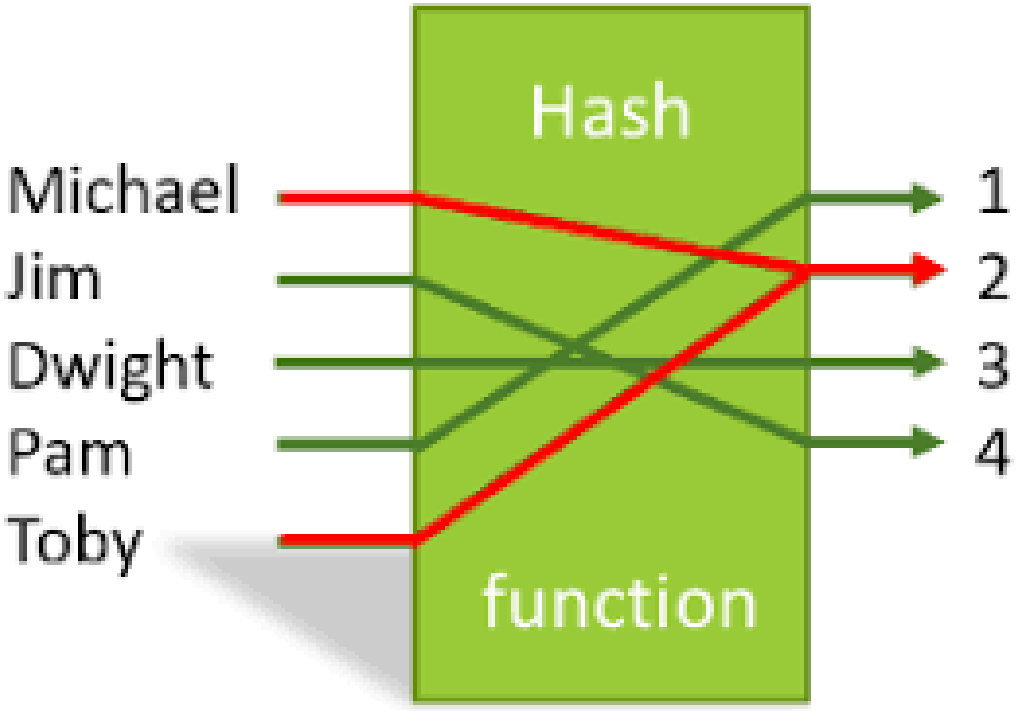
- Este método opera em duas etapas:
  1. multiplica-se a chave por uma constante A e extrai-se a parte fraccionária de kA

     Com: $0 < A < 1$

  2. 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 para uma sequência de caracteres (string)



Função de dispersão de uma sequência de caracteres (String)

- **Calcula qualquer que seja o tamanho da string um inteiro**

- Uma sequência de caracteres é em geral representada como uma sequência de inteiros (ex: códigos ASCII)
- Em consequência, a função de dispersão para Strings tem por entrada uma sequência de inteiros

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

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 (String)

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

- Porque:
  - Muitas vezes 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

Função de dispersão de uma sequência de caracteres (String)

- Em geral, o processamento consiste em:
  1. inicializar h (0 ou outro valor inicial)
  2. 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
  3. Obtenção do resultado final através de h mod M (método da divisão).

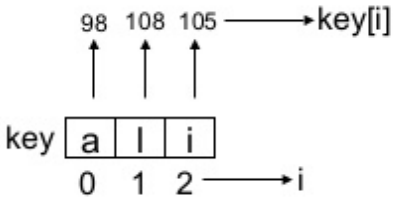
- Para evitar problemas de overflow, em geral os inteiros ki são representados por números inteiros sem sinal (unsigned int)
  - A utilização de representações de inteiros com sinal pode resultar em comportamentos estranhos

Exemplo simples

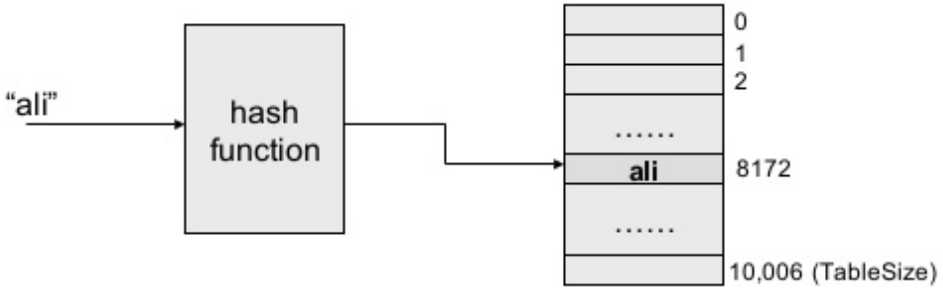
$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

```
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*37^2) % 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

Exemplo Matlab - string2hash()

```
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 de dispersão universais

Funções de Dispersão Universais

Funções de dispersão universais

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

  quaisquer duas chaves do universo colidem com probabilidade máxima igual a $1/M$ quando a função de dispersão h é extraída aleatoriamente de H

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

  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

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

  – sendo *a* e *b* inteiros aleatórios módulo *p (a ≠0)*

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

Outro método

- A chave é representa por um **vetor de inteiros**

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

  - 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

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

- E define-se :

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

Exemplo em 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

- 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 índice i tal que $x_i \neq y_i$
- Selecionando números aleatórios r_i com $j \neq i$
  Podemos definir $h'(x) = \sum_{j \neq i} r_j x_j$
- Desta forma $h(x) = h'(x) + r_j x_j$

- Exemplo:

  r=[1 2 3] x=[7 8 9] y= [7 1 9]

  h'(x)=h'(y) = 1 x 7 + 3 x 9 = 34

  h(x) = (h' () + 2 x 8) mod M

  h(y) = (h' () + 2 x 1) mod M

Demonstração da universalidade

- 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 uma possibilidade de igualdade em M
- A probabilidade de colisão é $1/N$
  - Como pretendíamos demonstrar

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

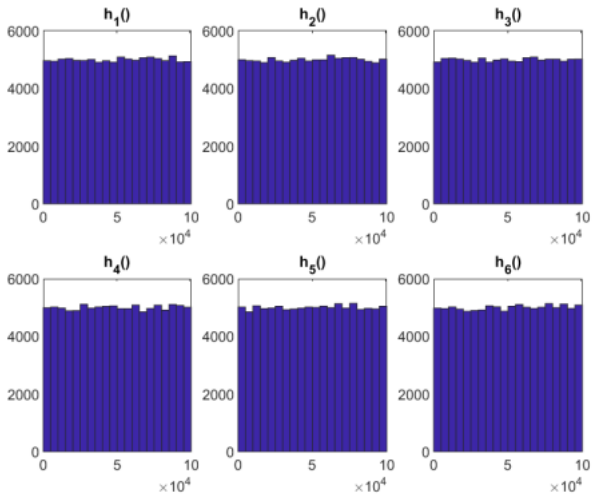
2. Processar todas essas chaves com as *n* funções de dispersão
   - Guardando os resultados produzidos (*hash codes*)

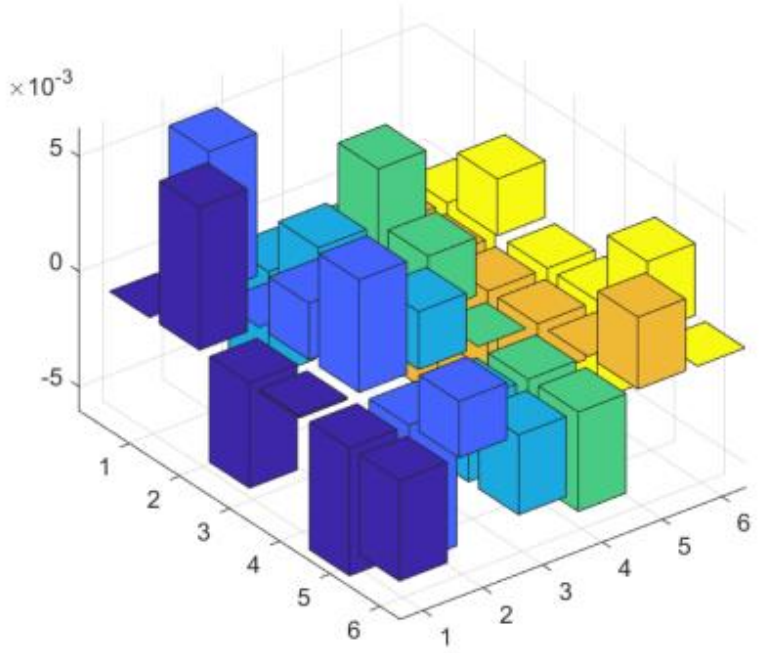
3. Analisar o histograma de cada função de dispersão
   - Para verificar a uniformidade da distribuição dos *hash codes*

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 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