Hash funtions

Motivação

Em muitos programas de computador torna-se necessário aceder a informação através de uma chave como por exemplo obter o nome associado a um numero de telefone.

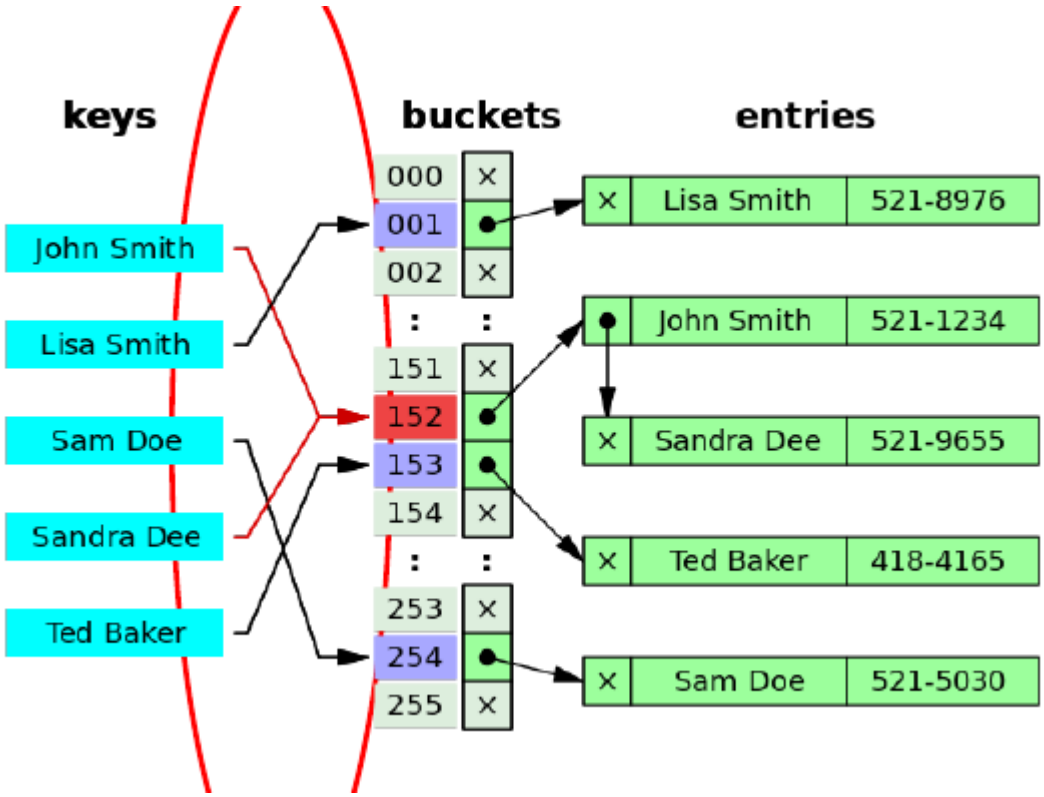
Hashtable

Uma hashtable é um dicionário simples em cada chave está associada a uma valor.

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

String valor = '962344631';
String chave = 'Tiago';


table.put(chave,valor);

System.out.println(table.get(chave));

// 92344631
```

Implementação com linked lists separadas

As chave são transformadas em posições de um array usando uma **função de disperção** onde cada posição do array é uma lista ligada



O que é uma hash function

Em termos gerais uma **Hash function** é uma função que mapeia um conjunto de variáveis de tamanho irregular para um outro conjunto de menor dimensão

Esta recebe um elemento de U como entrada e devolve um inteiro no intervalo de **[0,M-1]** ao que chamamos de **Hash Code**

Hash Code

O conjunto dos símbolos efectivamente 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

A **hash function** só é eficiente se distribuir as diferentes chaves de de forma relativamente **uniforme** ou seja **aleatoriamente**

A execução de uma **hash funtion** pode ser dividido em dois passos

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

Notação

- **h()** - hashfuntion
- **k** - chave

Colisões

Dado um conjunto de dados **U** mapeado em **M**, como **U** é inevitavelmente maior que **M** é inevitável que **h()** mapeie vários elementos para o mesmo valor.

Nestas situações dizemos que houve uma **colisão**.

Exemplo

Sendo k um elemento de U onde

h(k,M) = k mod M

teremos colisões para k,k+M,2M+k

Propriedades

- **Determinísticas**
- **Uniformes**
    - 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 **hash functions** devem ser gerados com aproximadamente a mesma probabilidade

Hash Functions para Inteiros

Existem vários tipos:

- baseadas em divisão
- baseadas em multiplicação
- membros de famílias universais.

Método da Divisão

Utiliza o resto da divisão

$$h(k) = \mathrm{k} \bmod \mathrm{M}$$

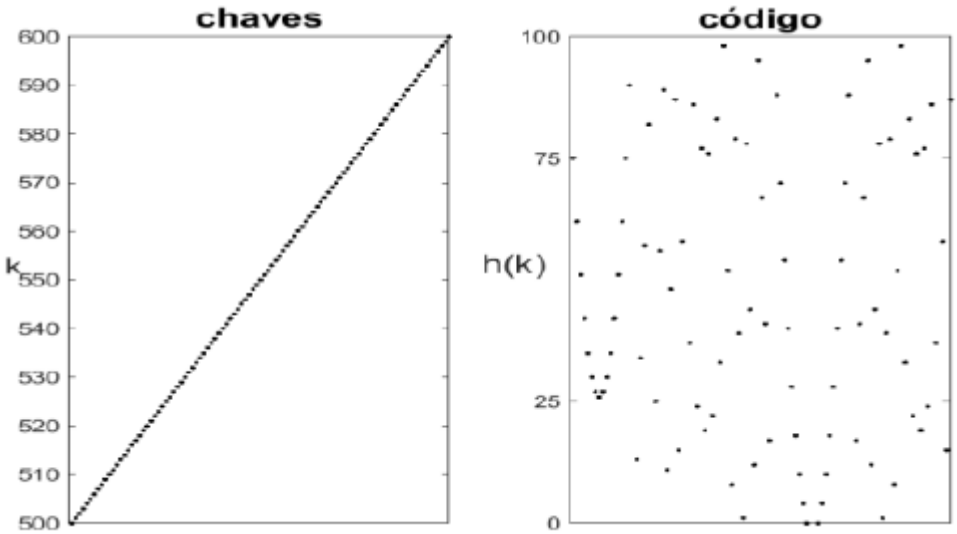
Em que **M** é o numero de posições e k a chave

- Método bastante rápido
- Funciona muito mal devido a colisões conforme o padrão das chaves

Foram desenvolvidas variações como a de **Knuth** que funcionam melhor

Variante de Knuth

$$h(k) = k(k+3) \bmod \mathrm{M}$$



(códico == hashcode)

Método da multiplicação

Este método opera em duas etapas

- Multiplica-se a chave por uma constante **A** tal que 0<**A**<1 e extrai-se a parte fraccionaria de kA
- Depois multiplicamos por **M** e arredonda-se ao menor ou igual inteiro

Hash Functions para sequencias de caracteres

Uma **hash function** para uma cadeia de caracteres calcula a partir da string, independente do tamanho, um inteiro.

Como sabemos uma string, portanto uma sequencia de caracteres é representada por inteiros, o código ASCII, pelo que a chave **k** será do tipo

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

Sendo que **h(k)** será um inteiro pequeno

O código ASCII é composto por 8 bits sendo usados apenas 7 e desses 7 para os caracteres comuns são usados 6 sendo o bit mais significativo o que indica se se trata de uma letra maiúscula ou minúscula o que, em muitas aplicações acaba por ser irrelevante.

Portanto em consequência a maioria dos algoritmos concentram -se 5 bits menos significativos.

Os algoritmos funcionam da seguinte maneira

- Inicializar uma variável **h** com valor igual a 0
- Percorrer a sequência de inteiros combinado-os com **h**
    - Esta parte depende do algoritmo
- Obtenção do hashcode fazedo **h mod M**

Para evitar problemas com overflows os inteiros **ki são representados por unsigned ints**

Ex :

```
int hash (const String &key,int tablesize){
    int HashVal = 0;
    for (int i = 0; i< key.length()){
        HashVal = 37 * hasVal + key[i];
        HashVal %= tablesize;
    }
    if(HashVal < 0)
        HashVal =+ tablesize;
    return HashVal
}
```

Variante CRC

- Fazer um shift circular de 5 bits para esquerda do h (shift circular quer dizer que em vez de substituir os elementos menos significantes com 0 substitui com aqueles que foram movidos)
- Depois fazer um XOR com ki

Variante PJW hash

- Shift para esquerda de 4 bits no h
- Adicionar ki
- Move os 4 bits mais significativos de h para o direita se não forem 0

Exemplos 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
```

Problemas

- As funções terão que lidar com conjuntos **S ⊆ U com |S| = n** chaves não conhecidos de antemão
- O objectivo é obter o mínimo de colisões o que pode não acontecer
- Uma **hash function deterministica** não pode garantir nenhuma garantia consoante a amostra de **U** que não ira ocorrer o pior caso.
- Não é facil alterar uma **hash function** "on the fly' se estiverem a ocorrer muitas colisões

Solução

Escolher uma função aleatoriamente dentro de uma família de **hash function**

Hash functions universais

Uma família de hash functions é universal se:

$$\forall x, y \subset U, \mathrm{x} \mathrel{!}= \mathrm{y}\colon P_{h \in H}[h(x) == h(y) \leq \frac{1}{M}]$$

Portanto quaisquer duas chaves do universo com probabilidade máxima igual 1/M quando a **hash function** é extraida aleatóriamente

- 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

Metodos de construção de famílias de Hash functions

Carter Wegman

Esta proposta consiste em escolher um numero primo **p >= M** e defenir cada função da familia da seguinte maneira:

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

Sendo **a e b** gerados aleatoriamente **mod p** em que **a!=0**

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

Método da matriz

1. Considerar as chaves na sua representação binária
2. Construir uma matriz de bits aleatoriamente
3. Multiplicar a chave pela matriz

Ex:

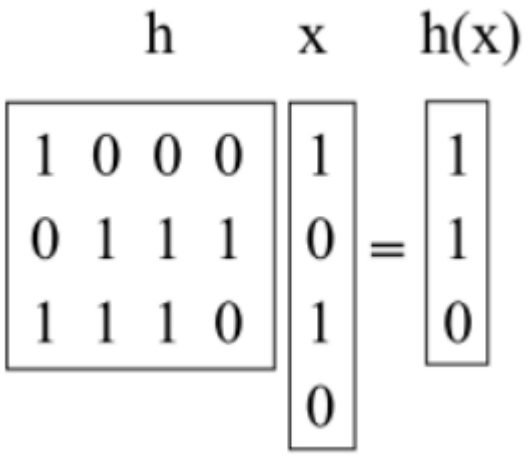
Consideremos chaves representadas por **u** bits onde **M é uma potencia de 2 -> 2^b**

- Criamos uma matriz h **b x u**

- Definimos **h(x) = h*x**

Exemplo:

u = 4, b = 3



A função será definida da seguinte forma:

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

Demonstração

• Um par de chaves diferentes x e y difere em algum dos bits. Consideremos que diferem no bit na posição i e que xi = 0 e yi = 1. • Se selecionarmos toda a matriz h exceto a coluna i obteremos um valor fixo para h x ; • No entanto, cada uma das 2b 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 que o da Matriz
- A chave é um vetor de inteiros

$$[x_1, x_2, ..., x_k] \ , \ x_i \in [0, M-1]$$

Onde **M** é um numero primo

Escolhe se **k** números aleatórios pertencentes a [0,M-1]

E define-se h(x):

$$h_x = (r_1 x_1 + r_2 x_2 + ... + r_k x_k) \; mod \; M$$

Como temos n hash functions

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