Aula 19
Tabelas de Símbolos

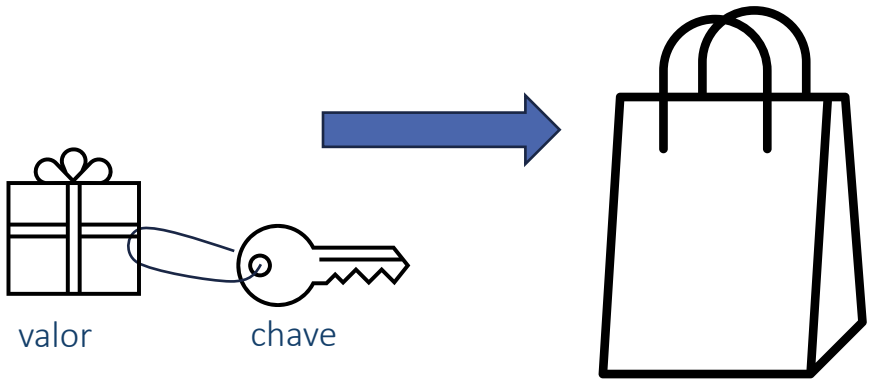
Tabelas de Dispersão (Hash Tables)

Algoritmos e Estruturas de Dados

Tabelas de Dispersão

Hash Tables

Objetivo de uma tabela de dispersão



Dada uma chave e um valor, quero guardar esse valor num "saco" com base na chave, que pode ter milhões de valores guardados…

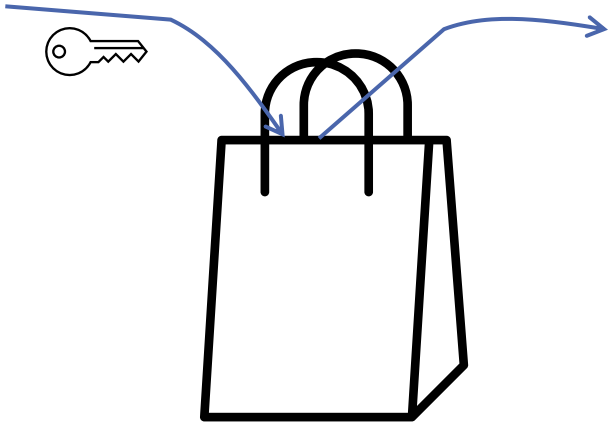
… e mais tarde, usar a chave para procurar pelo valor no saco de forma muito eficiente

Tabela de dispersão

Sabemos que se usarmos uma árvore de pesquisa balanceada conseguimos uma complexidade temporal de O(log n).

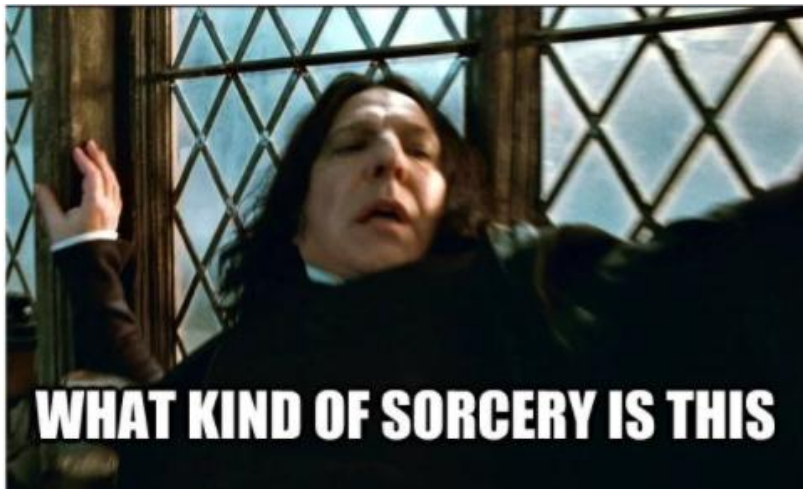
Mas na realidade conseguimos bem melhor se não precisarmos que as chaves estejam ordenadas entre si!

Tabela de dispersão



- É possível encontrar o valor em O(1)!

Tabela de dispersão

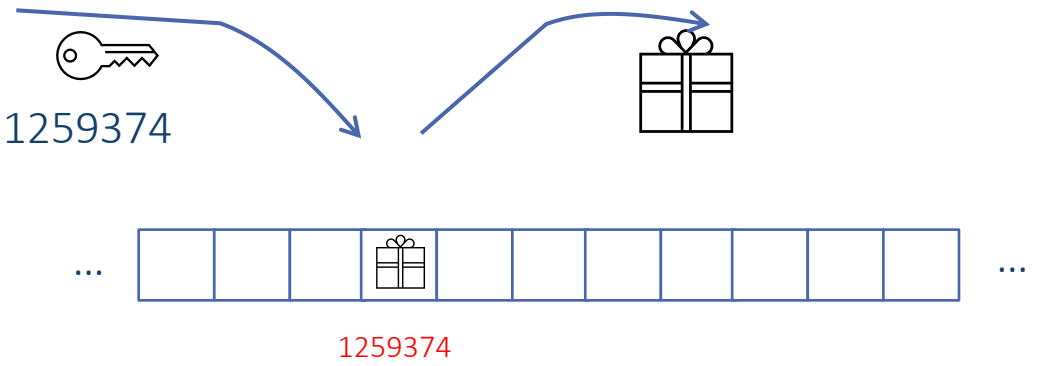


- Parece magia, mas não é.
  - Na realidade o que estamos a fazer é trocar memória por tempo
  - Em computação isto é algo que é sempre possível fazer

Tabela de dispersão com memória infinita

- Se considerarmos memória infinita, e tivermos controle sobre as chaves
  - a criação de uma tabela de dispersão com O(1) é trivial…

Tabela de dispersão com memória infinita



- Se considerarmos memória infinita, e tivermos controle sobre as chaves
  - Atribuir a cada novo valor a ser guardado **um identificador numérico único**
  - Guardar o objeto num array de tamanho infinito na posição indicada pelo **identificador númerico único**

Técnica do problema relaxado

- Um problema relaxado é um problema derivado a partir de um problema original, relaxando (ou ignorando) algumas das restrições
  - Em computação, a solução para um problema relaxado dá-nos boas ideias para resolver o problema original

- Ideias boas:
  - Transformar uma chave num identificador numérico único (quase único)
  - Guardar o valor num array de tamanho finito, numa posição calculada a partir do seu identificador numérico

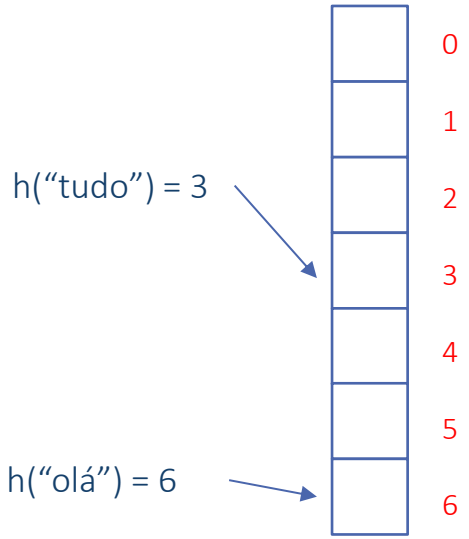
Tabelas de dispersão

- Ideia para implementação de tabela de símbolos
  - Array indexado por chave

    Se conseguirmos mapear uma chave k num índice para um array

    Conseguimos aceder a uma chave usando tempo ***constante***

    - **h** – função de dispersão (*hash* function)

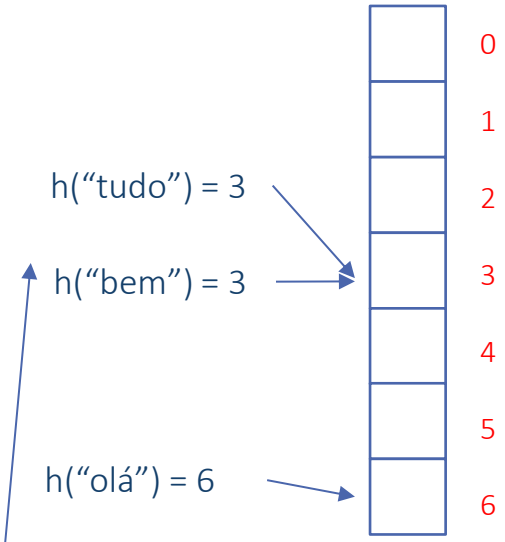


Propriedades função dispersão

- **Consistente**
  - Para duas chaves k_1 e k_2

    Se $k_1 = k_2$, então $h(k_1) = h(k_2)$
- **Eficiente**
  - Deve ser rápida a calcular
- **Dispersar as chaves**
  - A função h deve distribuir uniformemente as chaves pelos índices do array
- Deve minimizar colisões (é impossivel evitar)
  - Idealmente

    Se $K_1 \neq k_2$, então $h(k_1) \neq h(k_2)$



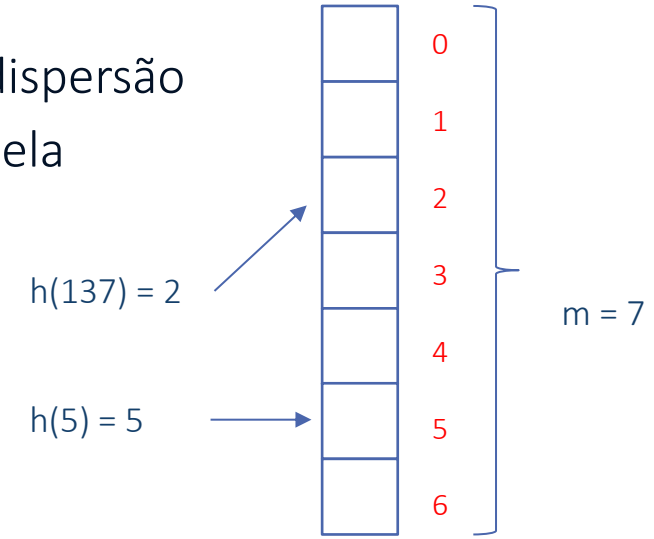
Quando 2 chaves diferentes dão origem ao mesmo valor de h, diz-se que existe uma colisão

Função de dispersão para inteiros

- Suponha que as chaves podem ser transformadas em valores inteiros
- Podemos determinar a posição na tabela de dispersão
  - resto da divisão inteira pelo tamanho da tabela
  - m

    tamanho da tabela de dispersão

    Ou número de baldes




```
public int hash(Key k)
{
    return (k.hashCode() & 0x7fffffff) % m;
}
```

Função que existe em java para todos os objectos, e que retorna um número inteiro que pode ser usado para uma função de hash

Esta operação serve para transformar números negativos em positivos

32 bit

1000 0000 0000 0000 0000 0000 0001 0001 → -17

AND

0111 1111 1111 1111 1111 1111 1111 1111

7 f f f f f f f

=

0000 0000 0000 0000 0000 0000 0001 0001 → 17

Tamanho da tabela de dispersão

- Tamanho da tabela de dispersão é muito importante para a função de hash
  - A função h deve distribuir uniformemente as chaves pelos índices do array

    1) Se as chaves forem aleatórias, podemos escolher qualquer tamanho m

    2) problema: se existir um padrão (divisor) comum entre o tamanho da tabela m e um conjunto de chaves, esse conjunto não será uniformemente distribuido pelas posições

    Este tipo de padrões ocorre com alguma frequência em problemas reais

Tamanho da tabela de dispersão

- problema: se existir um padrão (divisor) comum entre o tamanho da tabela m e um conjunto de chaves, esse conjunto não será uniformemente distribuido pelas posições
  - Ex:

    chaves são uma sequência de números pares (múltiplos de 2)

    12, 14, 18, …, 64, .., 128, …, 164

    tamanho da tabela é 12 (múltiplo de 2)

    h(12) = 0; h(14) = 2; h(18) = 6; h(64) = 4; h(128) = 8; h(164) = 8

    Os índices são sempre pares

    Estamos apenas a usar metade da tabela!

Tamanho da tabela de dispersão

- Solução
  - Escolher um tamanho m que

    minimize a probabilidade de existir um divisor comum entre m e um conjunto de chaves

    Escolher como ***m um número primo***

    Só tem 2 divisores: 1 e m

    Ex: m=11, chaves = 12, 14, 18, 20, 22, 24,26,..

    h(12)=1; h(14) = 3; h(18) = 7; h(20) = 9; h(22) = 0; h(24) = 2; h(26) = 4

Função HashCode em Java

- Qualquer chave pode ser transformada num número inteiro a ser usado numa função de hash

- Na línguagem Java todas as classes herdam o método hashCode()
  - retorna um inteiro que representa um valor que pode ser usado por uma função de hash
  - Implementações boas

    Tipos já existentes: String, Integer, Boolean, Float, File, Date, …
  - Implementação má

    Implementação por omissão para novos tipos criados pelo programador

    Devemos redefinir o método hashCode quando criamos um novo tipo

Implementação por omissão do método *hashCode*

- Em java, quando criamos uma nova classe, ela vem logo com uma implementação por omissão do método hashCode
  - x.hashCode() – retorna o endereço de memória do objecto x

  - Funciona como hashCode enquanto só usarmos o objecto x
  - Se tentarmos usar uma cópia da chave x

    y= x.clone();

    y não funciona como chave para x, pois y.hashCode() != x.hashCode()

    Falha a propriedade de consistência

    Para duas chaves k_1 e k_2

    Se $K_1 = k_2$, então $h(k_1) = h(k_2)$

Implementação do método *hashCode*

- Tipos Inteiro, Boolean

```
public final class Integer
{
    private final int value;
    ...
    public int hashCode(){return value;}
}
```

```
public final class Boolean
{
    private final boolean value;
    ...
    public int hashCode()
    {
        if(value) return 1231;
        else return 1237;
    }
}
```

Implementação do método *hashCode*

- Tipo String
  - Método de Horner

Para uma string s de tamanho L

$h(s) = s[0]R^{L-1} + s[1]R^{L-2} + \ldots + s[L-2]R^{1} + s[L-1]R^{0}$

Para obter uma melhor dispersão, escolhe-se um R primo

Na implementação java R = 31

Ex:

$h(\text{“call”}) = 99x31^3 + 97x31^2 + 108x31^1 + 108x31^0 = 3045982$

$(=)\ 108 + 31.(108 + 31.(97+31.(99)))$

Multiplica-se o valor anterior por 31, e soma-se o próximo caracter

Implementação do método *hashCode*

- Método de Horner

```
public final class String
{
    private static final int R = 31;
    private final char[] s;
    ...
    public int hashCode()
    {
        int hash = 0;
        for(int i = 0; i < s.length; i++)
        {
            hash = s[i] + (R*hash);
        }
        return hash;
    }
}
```

Multiplica-se o valor anterior por 31, e soma-se o próximo caracter

método hashCode para novas classes

```
public class myClass
{
    int x;
    Date y;
    String z;
    Long w;
}
```

3 Propriedades para uma boa função de dispersão (hash)

- Consistente
- Eficiente
- Dispersa de forma uniforme as chaves

Receita

Combinar cada parâmetro relevante x da classe usando o método de Horner

- *h = Rh + x*
- *Se parâmetro for um tipo primitivo, usar classe invólucro*
- *Se parâmetro for null retornar 0*
- *Se parâmetro for referência para objecto, usar hashCode()*
- *Se parâmetro for array, aplicar a regra para cada elemento do array*

método hashCode para novas classes

```
public class myClass
{
    int x;
    Date y;
    String z;
    Long w;

    private static final int R = 31;

    @Override
    public int hashCode()
    {
        int hash = 17;
        hash = R*hash + ((Integer) x).hashCode();
        hash = R*hash + (y == null ? 0 : y.hashCode());
        hash = R*hash + (z == null ? 0 : z.hashCode());
        hash = R*hash + (w == null ? 0 : w.hashCode());

        return hash;
    }
}
```

3 Propriedades para uma boa função de dispersão (hash)

- Consistente
- Eficiente
- Dispersa de forma uniforme as chaves

Escolher número primo não muito elevado

Inicialização com constante diferente de 0

método hashCode para novas classes

```
public class myClass
{
    int x;
    Date y;
    String z;
    Long w;

    private static final int R = 31;

    private int hash = 0;

    @Override
    public int hashCode()
    {
        if(hash == 0)
        {
            hash = 17;
            hash = R*hash + ((Integer) x).hashCode();
            hash = R*hash + (y == null ? 0 : y.hashCode());
            hash = R*hash + (z == null ? 0 : z.hashCode());
            hash = R*hash + (w == null ? 0 : w.hashCode());
        }

        return hash;
    }
}
```

Caso o cálculo do código de hash seja pesado, podemos guardar o valor para chamadas futuras

parâmetro usado para guardar o código de hash

apenas calculamos a primeira vez o código de hash

Se algum dos parâmetros for alterado, devemos colocar hash = 0 para forçar o recálculo

se hash != 0 retonar o valor guardado

Tratamento de colisões

- O que fazer quando duas chaves diferentes têm o mesmo valor de hash?
- Tratamento de colisões
  - Várias soluções
  - Tabelas de encadeamento separado

    Separate chaining
  - Tabelas de endereçamento aberto

    Linear probing