

# Aula 12

## Estruturas de Dados

### Tabelas de dispersão

Programação II, 2015-2016

v1.15, 25-05-2016

DETI, Universidade de Aveiro

12.1

#### Objectivos:

- Tabelas de dispersão (*Hash Tables*);

### Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Funções de Dispersão</b>	<b>4</b>
<b>3</b>	<b>Factor de Carga</b>	<b>5</b>
<b>4</b>	<b>Colisões</b>	<b>5</b>
4.1	Chaining Hash Table . . . . .	6
4.2	Open Addressing Hash Table . . . . .	7

12.2

## 1 Introdução

### Colecções de dados: o que vimos até agora

- LinkedList
  - `addFirst()`, `addLast()`, `removeFirst()`, `first()`, ...
- SortedList
  - `insert()`, `remove()`, `first()`, ...
- Stack
  - `push()`, `pop()`, `top()`, ...
- Queue
  - `in()`, `out()`, `peek()`, ...
- KeyValueList (implementa um **dicionário**)
  - `set()`, `get()`, `remove()`, ...

12.3

## Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
    - Espaço:  $O(n)$  (proporcional ao número de elementos).
    - Tempo (acesso por índice):  $O(1)$  (constante).
    - Tempo (procura por valor):  $O(n)$ .
    - Tempo (inserção com redimensionamento):  $O(n)$ .
  2. Listas Ligadas
    - Espaço:  $O(n)$ .
    - Tempo (acesso, procura):  $O(n)$ .
    - Tempo (inserção):  $O(1)$ .
  3. Dicionários
    - Eficiência depende da implementação.
    - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
    - Vamos agora ver implementações eficientes do conceito de dicionário.

12.4

## Dicionários: problema

- Uma empresa pretende aceder à informação de cada empregado usando como *chave* o respectivo *Número de Identificação de Segurança Social (NISS)*.
  - Há milhões de inscritos: o NISS tem 11 dígitos.
  - A empresa só está interessada nos seus empregados, na ordem das centenas.
  - Como garantir tempo de acesso  $O(1)$ ?
- Implementação em **lista de pares chave-valor**.
  - Não suporta a complexidade pretendida.
- Poderíamos usar o NISS como índice num **vector** de empregados.
  - Teria que ser um vector com dimensão  $10^{11}$  e índices de 0 a 99999999999.
  - Só iríamos utilizar uma pequeníssima percentagem das entradas do vector!!!
  - *Conclusão*: para termos tempo  $O(1)$ , estamos a desperdiçar muito espaço de memória.

12.5

## Dicionários: como otimizar?

- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós seguintes, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de  $O(n)$  para  $O(\log(n))$ .
  - Neste caso, as listas transformam-se em árvores binárias (aula 13).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - \* E não para o número total de chaves possíveis!
    - \* No exemplo dado: o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.
  - O problema neste caso é estabelecer a **correspondência** entre as **chaves** presentes no dicionário e os **índices** do vector.

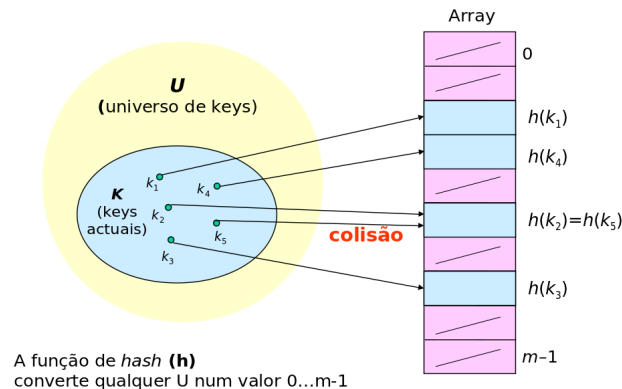
12.6

## Dicionários: implementação usando vector

- *Objectivo*: desempenho com o melhor dos “dois mundos”:
  - Tempo de acesso / procura por chave:  $O(1)$ , como nos vectores.
  - Tempo de inserção:  $O(1)$ , como nas listas não ordenadas.
  - Espaço:  $O(n)$ .
- Para cada chave a inserir ou procurar, calcula-se o índice correspondente no vector.
  - Convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas para o mesmo índice.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

12.7

## Tabelas de dispersão



12.8

## Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - HashTable
- Serviços:
  - HashTable(*n*): construtor;
  - get(*key*): devolve o elemento associado à chave dada
  - set(*key*, *elem*): actualiza o elemento associado à chave *k*, caso esta exista, ou insere o novo par (*k*, *e*)
  - remove(*key*): remove a chave dada bem como o elemento associado
  - contains(*key*): tabela contém a chave dada
  - isEmpty(): tabela vazia
  - size(): número de associações;
  - clear(): limpa a tabela;
  - keys(): devolve um vector com todas as chaves existentes.

12.9

## 2 Funções de Dispersão

### Tabelas de dispersão: Funções de Hash

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:
$$\text{chave} \longrightarrow \text{inteiro}$$
  - Função de Compressão ( $m$  é a dimensão do vector)
$$\text{inteiro} \longrightarrow \text{inteiro } [0, m-1]$$
- $h(k)$  é o valor de *hash* da chave  $k$ .
- *Problema*:
  - *Colisão*: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

12.10

### Tabelas de dispersão: Funções de Hash

- A escolha de uma “boa” função de *hash* deve reduzir o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma “boa” função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de  $h(k)$ ...

12.11

### Funções de hash: aproximações

#### 1. Método da divisão:

- Este método usa o resto da divisão inteira:

$$h(k) = k \% m$$

- Se  $m$  é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

- Outra má opção é  $m = 2^p$  ( $h(k)$  serão os  $p$  bits menos significativos).
- Para este método utilizar um valor primo para  $m$  é uma escolha razoável.

#### 2. Método da multiplicação:

- Pode fazer uso dos operadores de *bit shift*
- Exemplo:  $h(k) = (k \ll 3) + (k \gg 28) + 33$

12.12

## Funções de hash: Exemplo para chaves tipo String

```
private int hashstring(String str, int tablesiz)
{
    int len=str.length();
    long hash=0;
    char[] buffer=str.toCharArray();

    int c=0;
    for (int i=0; i < len; i++)
    {
        c = buffer[i]+33;
        hash = ((hash<<3) + (hash>>28) + c);
    }

    hash = hash % tablesiz;
    return (int) (hash>=0 ? hash : hash + tablesiz);
}
```

- Todos os objectos em Java têm a si associados uma função inteira de dispersão: `hashCode()`.
- Vamos utilizar esta função nas nossas tabelas de dispersão.

12.13

## 3 Factor de Carga

### Tabelas de dispersão: Factor de Carga

- O *factor de carga* (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ( $\alpha = \frac{n}{m}$ ).
- Dimensionamento de  $\alpha$ :
  - um valor alto de  $\alpha$  significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de  $\alpha$  significa que temos muito espaço desperdiçado;
  - valor recomendado para  $\alpha$ : entre 50% e 80%.

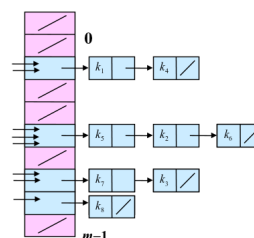
12.14

## 4 Colisões

### Resolução do Problema das Colisões

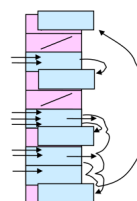
#### 1. Chaining Hash Table (Close Addressing or Open Bucket)

- Um conjunto de chaves (+elementos) associado a um mesmo índice (*bucket*);
- Cada entrada do vector contém uma lista ligada.



#### 2. Open Addressing Hash Table (Close Bucket)

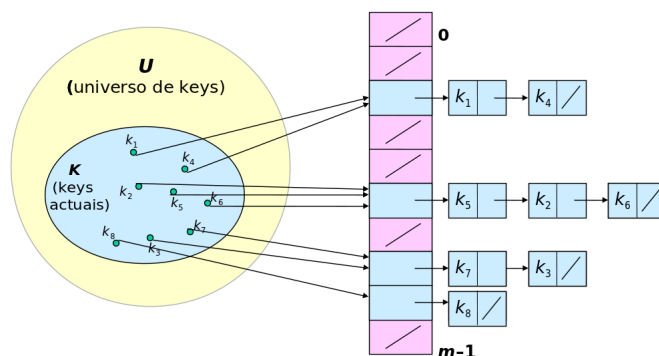
- Uma chave/elemento por *bucket*;
- No caso de colisão, faz-se uso de um procedimento consistente para armazenar o elemento numa entrada livre da tabela;
- O vector é tratado como circular.



12.15

## 4.1 Chaining Hash Table

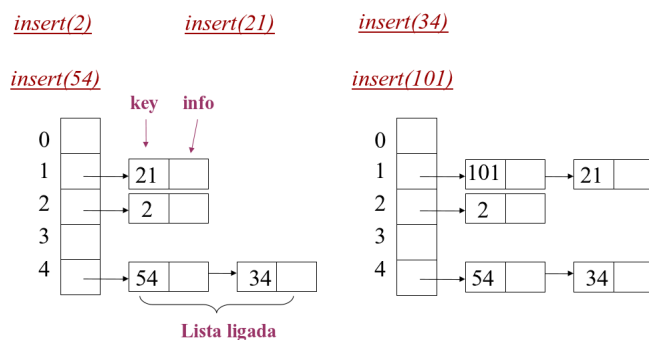
### Chaining Hash Table



12.16

### Chaining Hash Table: Exemplo

- $h(k) = k \% m$  com  $m = 5$  e  $k \in [0;999]$



12.17

### Chaining Hash Table

- Complexidade Temporal:
  - **Inserção:**  $O(1)$ 
    - \* tempo de cálculo da  $h(k)$  + tempo de inserção no início da lista ligada.
  - **Pesquisa:** proporcional ao comprimento máximo da lista ligada.
  - **Remoção:** o mesmo que a pesquisa.
- Não esquecendo que .....uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

12.18

### Chaining Hash Table: esqueleto

```

public class HashTable<E> {

    public HashTable(int n) {
        array = (KeyValueList<E>[])new KeyValueList[n];
        for(int i = 0; i < array.length; i++)
            array[i] = new KeyValueList<E>();
    }

    public E get(String k) {
        assert contains(k) : "Key does not exist";
        ... ..
    }

    public void set(String k, E e) {
        ... ..
        assert contains(k) && get(k).equals(e);
    }

    public void remove(String k) {
        assert contains(k) : "Key does not exist";
        ... ..
        assert !contains(k) : "Key still exists";
    }

    public boolean contains(String k) { ... }
    public String[] keys() { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }

    private KeyValueList<E>[] array;
    private int size = 0;
}

```

12.19

### Chaining Hash Table: set/get

```

public class HashTable<E>
...
    public E get(String key)
    {
        assert contains(key);

        int pos = hashFcn(key);
        return array[pos].get(key);
    }

    public void set(String key, E elem)
    {
        int pos = hashFcn(key);
        boolean newelem = array[pos].set(key, elem);
        if (newelem) size++;

        assert contains(key) && get(key).equals(elem);
    }
    ...
}

```

12.20

## 4.2 Open Addressing Hash Table

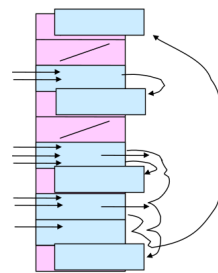
### Open Addressing Hash Table

- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- Usual dimensionar-se a tabela com tamanho 30% superior ao número máximo de elementos previsto ( $\alpha == 0.70$ ):
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

12.21

- Resolução de Colisões:

- $i_0 = h(k)$
- se índice/bucket ocupado, então:
- $i_{j+1} = (i_j + c) \% m$
- ... sucessivamente até encontrar um *bucket* livre.
- o valor  $c$  pode ser contante (pesquisa linear), ou seguindo outra estratégia (quadrática, ...).



### Open Addressing Hash Table: Exemplo

- $h(k) = k \% m$  com  $m = 5$  e  $k \in [0; 99]$

<u>insert(2)</u>	<u>insert(21)</u>	<u>insert(34)</u>	<u>insert(54)</u>
key data	key data	key data	key data
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Colisão:  
índice #4

$$(4 + 1) \% 5 = 0$$

12.22