

Aula 12

Dicionários

Tabelas de dispersão

Programação II, 2020-2021

v1.5, 27-05-2018

Introdução

Tabelas de Dispersão

Funções de Dispersão

Factor de Carga

Colisões

Tabela de dispersão
com encadeamento
externo

Tabela de dispersão
com encadeamento
interno

Encadeamento
externo versus interno

- 1 Introdução
- 2 Tabelas de Dispersão
- 3 Funções de Dispersão
- 4 Factor de Carga
- 5 Colisões
- 6 Tabela de dispersão com encadeamento externo
- 7 Tabela de dispersão com encadeamento interno
- 8 Encadeamento externo versus interno

Introdução

Tabelas de Dispersão

Funções de Dispersão

Factor de Carga

Colisões

Tabela de dispersão
com encadeamento
externo

Tabela de dispersão
com encadeamento
interno

Encadeamento
externo versus interno

Coleções 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()`, ...

- 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 no fim): $O(1)$.
- Tempo (procura em vector ordenado): $O(\log n)$.
- Tempo (inserção por ordem): $O(n)$.

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 de dicionários.

Introdução

Tabelas de Dispersão

Funções de Dispersão

Factor de Carga

Colisões

Tabela de dispersão
com encadeamento
externo

Tabela de dispersão
com encadeamento
interno

Encadeamento
externo versus interno

- Uma empresa pretende aceder à informação de cada empregado usando como **chave** o respectivo *Número de Identificação de Segurança Social (NISS)*.
 - O NISS tem 11 dígitos.
 - A empresa só tem algumas centenas ou milhares de empregados.
 - 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 entre 0 e 99 999 999 999.
 - Só iríamos utilizar uma pequeníssima percentagem das entradas do vector!
 - **Conclusão**: para termos tempo $O(1)$, teríamos de desperdiçar muito espaço de memória.

Introdução

Tabelas de Dispersão

Funções de Dispersão

Factor de Carga

Colisões

Tabela de dispersão
com encadeamento
externo

Tabela de dispersão
com encadeamento
interno

Encadeamento
externo versus interno

- Lista de pares chave-valor.
 - Se cada nó passar a apontar para dois nós, 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.
- do 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: $O(n)$, onde n é o número de pares armazenados.
- **Solução:** Para cada chave a inserir ou procurar, **calculamos** o índice correspondente no 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 no mesmo índice.
 - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
- Esta estrutura de dados — um vector indexado através de uma função de dispersão — é conhecida como **tabela de dispersão** (*hash table*).

Introdução

Tabelas de Dispersão

Funções de Dispersão

Factor de Carga

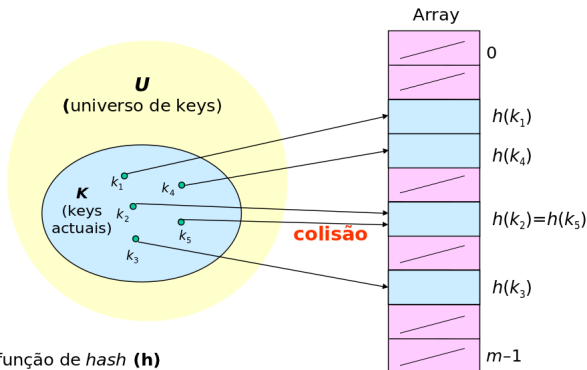
Colisões

Tabela de dispersão
com encadeamento
externo

Tabela de dispersão
com encadeamento
interno

Encadeamento
externo versus interno

Tabelas de dispersão



A função de hash (**h**)
converte qualquer U num valor $0 \dots m-1$

- 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.

- Funções de *Hash* (duas partes):
 - Cálculo do *hash code*:
 $\text{chave} \longrightarrow \text{inteiro}$
 - Função de Compressão (m é a dimensão do vector)
 $\text{inteiro} \longrightarrow \text{inteiro} \in [0, m - 1]$
- $h(k)$ 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)!

- Uma “boa” função de *hash*:
 - Deve minimizar o número de colisões.
 - Deve produzir índices *distribuídos uniformemente* por todo o vector.
 - Deve “quebrar” padrões que possam ocorrer nas chaves.
 - Deve ser fácil de calcular (rápida).
- É possível criar funções de *hash perfeitas*, se todas as chaves forem conhecidas previamente.
- Vamos ver vários exemplos de $h(k)$...

① Método da divisão:

- Este método usa o resto da divisão inteira:

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

- Mas, 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).
- Por isso, é recomendável utilizar um valor primo para m .

② Método da multiplicação:

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

Funções de *hash*: Exemplo para chaves tipo String

```
private int hashstring(String str, int tablesiz) {  
    // Hashcode:  
    long hash = 0;  
    for (int i=0; i < str.length(); i++) {  
        char c = str.charAt(i);  
        hash = (hash<<5) - hash + c; // = 31*hash + c  
    }  
  
    // Compression:  
    hash = Math.abs(hash % tablesiz);  
    return hash;  
}
```

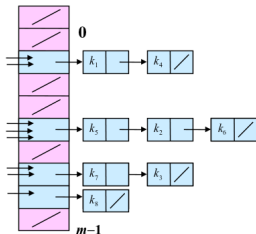
- Todos os objectos em Java têm uma função de dispersão, `hashCode()`, que devolve um inteiro.
- Vamos utilizar essa função nas nossas tabelas de dispersão.

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

Resolução do Problema das Colisões

1 Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)

- Múltiplos pares chaves-valor associados a um mesmo índice;
- Cada entrada do vector contém uma lista ligada de pares chave-valor.



2 Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)

- No máximo, um par chave-valor em cada posição do vector;
- No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
- O vector é tratado como circular.

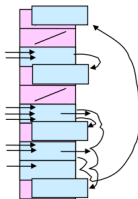


Tabela de dispersão com encadeamento externo

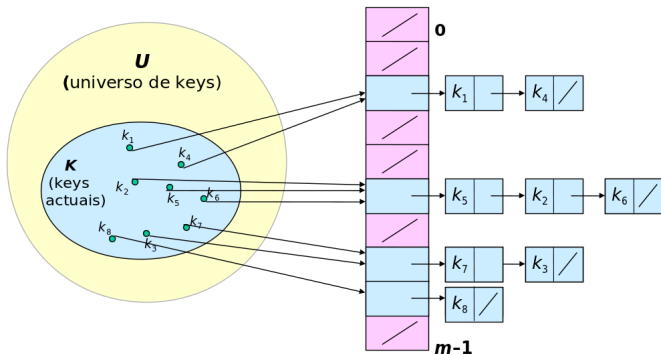


Tabela de dispersão com encadeamento externo: exemplo

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

insert(2)

insert(21)

insert(34)

insert(54)

insert(101)

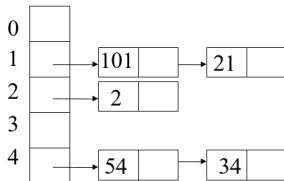
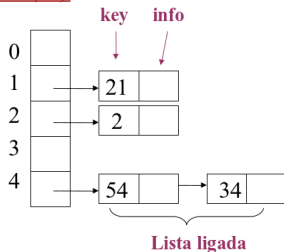


Tabela de dispersão com encadeamento externo: desempenho

- As operações fundamentais (`contains`, `get`, `set`, `remove`) requerem:
 - 1 Cálculo da função $h(k)$, que se considera demorar um tempo fixo.
 - 2 Pesquisa na lista ligada, que demora um tempo proporcional ao comprimento da lista.
- Numa tabela de tamanho m com n elementos, o comprimento médio das listas é $n/m = \alpha$.
- Logo, a complexidade temporal (média) é $O(1 + \alpha)$.
- Ou, aproximadamente $O(1)$, quando a tabela é bem dimensionada ($\alpha < 1$).
- Mas... uma má função de *hash* \implies demasiadas colisões \implies desempenho pior até $O(n)$.
- As operações de travessia (`keys`, `toString`) são $O(m + n)$.

Tabela de dispersão com encadeamento externo: esqueleto

```
public class HashTable<E> {  
    private KeyValueType<E>[] array;  
    private int size = 0;  
  
    public HashTable(int n) {  
        array = (KeyValueType<E>[])new KeyValueType[n];  
        for(int i = 0; i < array.length; i++)  
            array[i] = new KeyValueType<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() { ... }  
}
```

Tabela de dispersão com encadeamento externo: 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);  
    }  
    ...  
}
```

Tabela de dispersão com encadeamento interno

- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
 - O objectivo é minimizar o tempo despendido com a resolução das colisões.
- Resolução de Colisões:
 - $i_0 = h(k)$
 - se posição i_j ocupada, então tentar:
 - $i_{j+1} = (i_j + c) \% m$
 - e repetir até encontrar uma posição livre.
 - o valor c pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, ...).

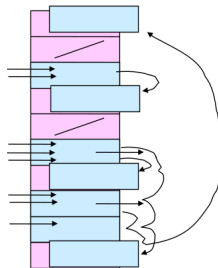


Tabela de dispersão com encadeamento interno: exemplo

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

insert(2)

	key	data
0		
1		
2	2	...
3		
4		

insert(21)

	key	data
0		
1	21	...
2	2	...
3		
4		

insert(34)

	key	data
0		
1	21	...
2	2	...
3		
4	34	...

insert(54)

	key	data
0	54	...
1	21	...
2	2	...
3		
4	34	...

Colisão:
índice #4

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

[Introdução](#)

[Tabelas de Dispersão](#)

[Funções de Dispersão](#)

[Factor de Carga](#)

[Colisões](#)

[Tabela de dispersão
com encadeamento
externo](#)

[Tabela de dispersão
com encadeamento
interno](#)

[Encadeamento
externo versus interno](#)

- Tabela de dispersão com encadeamento externo:
 - Não tem limite rígido do número de elementos.
 - Desempenho degrada suavemente à medida que o factor de carga aumenta.
 - Não desperdiça memória com dados que ainda não existem.
- Tabela de dispersão com encadeamento interno:
 - Não precisa de guardar apontadores de uns elementos para os outros.
 - Não perde tempo a alocar nós sempre que chega um novo elemento.
 - Toda a memória é alocada no início. Não requer alocação dinâmica.
 - Especialmente adequado quando os elementos são de pequena dimensão.
- Na prática, e para a maior parte das situações, estas diferenças são marginais.