

Aula 11

Estruturas de Dados

Tabelas de dispersão

Programação 2, 2º semestre 2013-2014

v0.96, 23-05-2015

DETI, Universidade de Aveiro

11.1

Objectivos:

- Tabelas de dispersão (*Hash Tables*);
- *Arrays* associativos.

Conteúdo

1	Introdução	1
2	Funções de Dispersão	3
3	Factor de Carga	4
4	Colisões	4
4.1	<i>Chaining Hash Table</i>	5
4.2	<i>Array Associativo</i>	5
4.3	<i>Open Addressing Hash Table</i>	7

11.2

1 Introdução

Tabelas de dispersão: Introdução

- Quando analisamos a complexidade de uma estrutura de dados composta temos duas componentes:
 1. Espaço: quantidade de memória necessária;
 2. Tempo: de Pesquisa, Inserção, Remoção, . . . ;
- Análise das duas implementações conhecidas:
 1. Listas Ligadas:
 - Espaço: $O(n)$
 - Tempo: $O(n)$
 2. *Array*:
 - Espaço: Muito Grande. Proporcional ao número máximo elementos que este suporta.
 - Tempo (procura por índice): $O(1)$ (Constante)
 - Tempo (procura por valor): $O(n)$ (Se ordenado pode baixar para $O(\log(n))$)

11.3

Tabelas de dispersão: Introdução

- Exemplo: armazenar registos de pessoas, requisitos:
 - utilizando como *chave* o seu número de segurança social (11 dígitos);
 - e *complexidade algoritmica* temporal $O(1)$.
- Para obtermos essa complexidade algoritmica, utilizarmos a chave como índice do *array*:
 - implica um *array* com dimensão $10^{11}[0 \dots 99999999999]$
 - em Portugal temos uma população de 10 milhões, pelo que só iríamos utilizar uma pequenís-sima percentagem das entradas do *array*!!!
- *Conclusão*: para termos $O(1)$ estamos a desperdiçar muito espaço.

11.4

Tabelas de dispersão: Objectivo

- *Objectivo*: desempenho com o melhor dos “dois mundos”:
 1. Tempo: $O(1)$
 2. Espaço: $O(n)$
- *Solução*:
 1. Fazer uso de um *array*, em que o índice é calculado por uma função inteira;
 2. Encolher o espaço de estados das chaves por forma a serem indexáveis nesse *array*;
- A compressão e mapeamento das chaves para os índices válidos do *array* é feita pela função de dispersão (*hash function*).

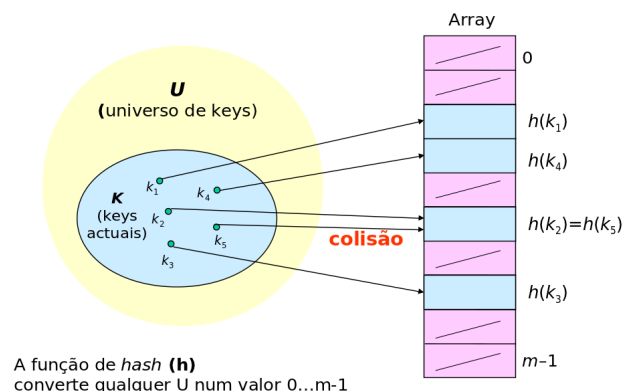
11.5

Tabelas de dispersão: Conteúdo

- Conteúdo
 - Conjunto de pares (*chave*, *valor*)
 - Cada valor está assim associado a uma “chave”
- A “chave” pode ser qualquer coisa, desde que dela possamos calcular um índice para um *array*:
 - um inteiro 32/64 bits;
 - um *String*;
 - ...
- Por outro lado, o *array* será da dimensão que desejarmos.

11.6

Tabelas de dispersão



11.7

2 Funções de Dispersão

Tabelas de dispersão: Funções de Hash

- Funções de Hash (duas partes):
- Cálculo do *hash code*:

chave \rightarrow inteiro

- Função de Compressão (m é a dimensão do *array*):

inteiro \rightarrow 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 *array*)!

11.8

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 *array*.
- A escolha de uma “boa” função de *hash* pode ter em consideração o tipo de 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 $h(k)$...

11.9

Funções de hash: Aproximações

1. Método da divisão:

- Este método faz uso da operação 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: $h(k) = (k \ll 3) + (k \gg 28) + 33$

11.10

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);
    }

    return (int) (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.

11.11

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 ($\alpha = \frac{n}{m}$)
- Regra de Dimensionamento de α :
 - um elevado valor de α significa que vamos ter maior probabilidade de colisões;
 - um valor baixo de α significa que temos um elevado consumo de espaço;
 - valor recomendado para α : entre 50% e 80%.

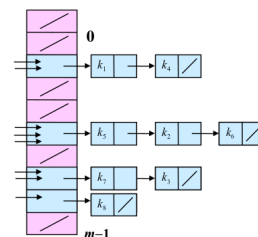
11.12

4 Colisões

Resolução do Problema das Colisões

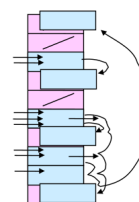
1. Chaining Hash Table (Close Addressing or Open Bucket)

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



2. Open Addressing Hash Table (Close Bucket)

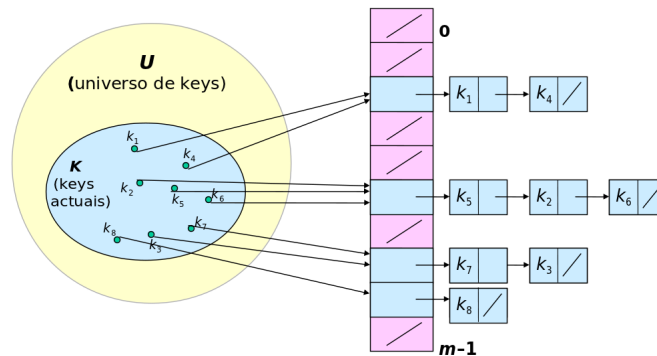
- Uma chave/elemento por *bucket*;
- No caso de colisão, faz-se uso de um procedimento consistente para armazenar o elemento numa entrada livre da tabela;
- O *array* é tratado como circular.



11.13

4.1 Chaining Hash Table

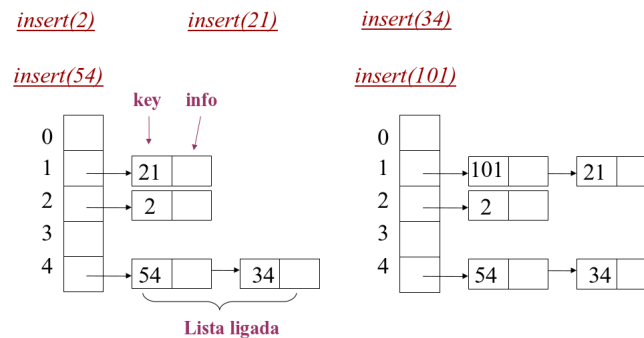
Chaining Hash Table



11.14

Chaining Hash Table: Exemplo

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



11.15

Chaining Hash Table

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

11.16

4.2 Array Associativo

Módulo Array Associativo

- Nome do módulo:
 - AssociativeArray
- Serviços:
 - AssociativeArray(n): construtor;

- `set(key, elem)`: definir uma associação;
- `get(key) -> elem`: devolve valor associado a uma chave;
- `delete(key)`: apaga uma associação;
- `exists(key) -> boolean`: indica se existe associação a uma chave;
- `isEmpty()` -> boolean: tabela vazia;
- `isFull()` -> boolean: tabela cheia;
- `size()` -> int: número de associações;
- `clear()`: limpa a tabela;
- `keysToArray()` -> `key[]`: devolve um *array* com todas as chaves existentes.

11.17

Chaining Hash Table: set

```
set(key, elem)
    pos = hashCode(key)
    n = searchNode in array[pos] with key
    if n null then
        n = new Node
        n.key = key
        n.next = array[pos]
        array[pos] = n
    n.elem = elem
```

11.18

Chaining Hash Table: get & exists

```
get(key)
    assert exists(key)

    pos = hashCode(key)
    n = searchNode in array[pos] with key
    result = n.elem
```

```
exists(key)
    pos = hashCode(key)
    n = searchNode in array[pos] with key
    result = n not null
```

11.19

Chaining Hash Table: delete

```
delete(key)
    assert exists(key)

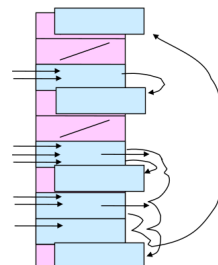
    pos = hashCode(key)
    lastn = null
    n = array[pos]
    while (n not null) and (n.key not equal to key)
        lastn = n
        n = n.next
    if lastn exists then
        lastn.next = n.next
    else
        array[pos] = n.next
```

11.20

4.3 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.
- 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, ...).



11.21

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	54 ...
1		1	21 ...
2	2 ...	2	2 ...
3		3	
4		4	34 ...

Colisão: índice #4

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

11.22

