Aula 12

Estruturas de Dados

Tabelas de dispersão

Programação II, 2015-2016

v1.1, 01-05-2016

DETI, Universidade de Aveiro

12.1

Objectivos:
- Tabelas de dispersão (*Hash Tables*);

Conteúdo

12.2

1 Introdução

Colecções de dados: o que vimos até agora

- Analisamos 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

12.3

Dicionários: problema

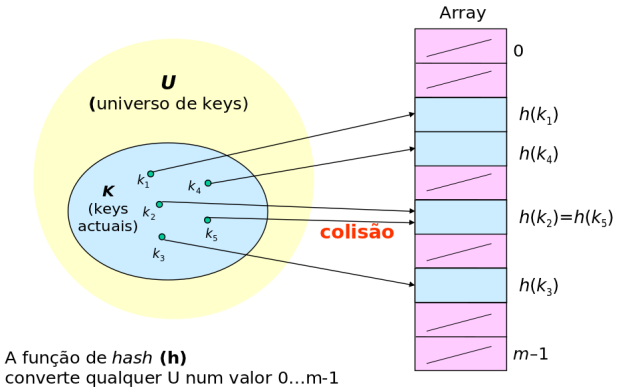
- Uma empresa pretende aceder à informação de cada empregado usando como *chave* o respectivo *Número de Identificação de Segurança Social* (*NISS*)
  - Há milhões de inscritos: o NISS tem 11 dígitos
  - A empresa só está interessada nos seus empregados, na ordem das centenas
  - Como garantir tempo 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

Dicionários: como optimizar?

- 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

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*)



2 Funções de Dispersão

Tabelas de dispersão: Funções de *Hash*

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

  `chave` $\longrightarrow$ `inteiro`

- Função de Compressão (m é a dimensão do vector)

  `inteiro` $\longrightarrow$ `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!

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 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 $h(k)\ldots$

Funções de *hash*: Aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

- 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: $h(k) = (k << 3) + (k >> 28) + 33$

12.10

Funções de *hash*: Exemplo para chaves tipo `String`

```
private int hashstring(String str, int tablesize)
{
   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 % tablesize);
}
```

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

12.11

3 Factor de Carga

Tabelas de dispersão: Factor de Carga

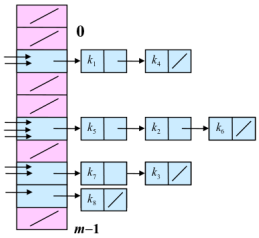
- O *factor de carga* (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$)
- Dimensionamento de α:
  - um elevado valor de α significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de α significa que temos um elevado consumo de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

12.12

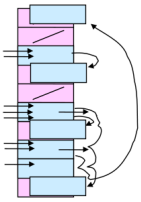
4 Colisões

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.



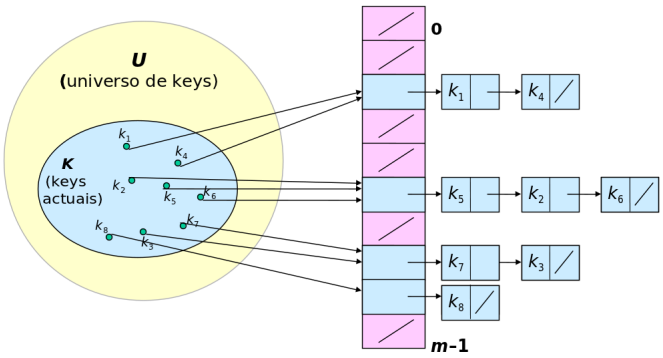
2. *Open Addressing Hash Table (Close Bucket)*
   - Uma chave/elemento por *bucket*;
   - No caso de colisão, faz-se uso de um procedimento consistente para armazenar o elemento numa entrada livre da tabela;
   - O vector é tratado como circular.



12.13

4.1 *Chaining Hash Table*

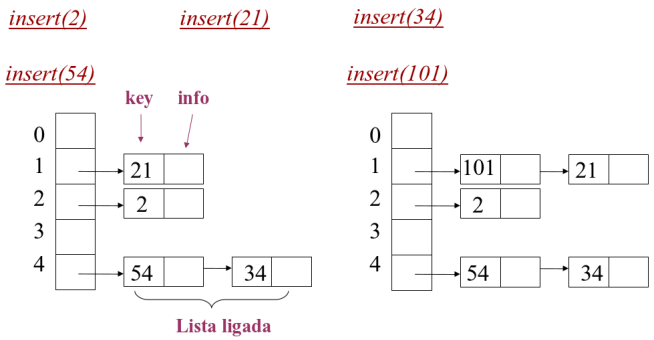
Chaining Hash Table



12.14

Chaining Hash Table: Exemplo

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



12.15

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 que ..... uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

12.16

4.2 *HashTable*

Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `set(key,elem)`: definir uma associação;
  - `get(key) -> elem`: devolve valor associado a uma chave;
  - `delete(key)`: apaga uma associação;
  - `contains(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;
  - `keys() -> key[]`: devolve um vector com todas as chaves existentes.

12.17

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
```

12.18

Chaining Hash Table: `get` & `contains`

```
get(key)
   assert contains(key)

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

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

12.19

Chaining Hash Table: `delete`

```
delete(key)
   assert contains(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 contains then
      lastn.next = n.next
   else
      array[pos] = n.next
```

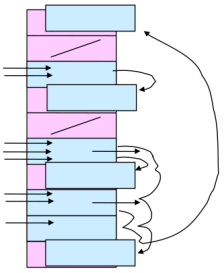
12.20

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.

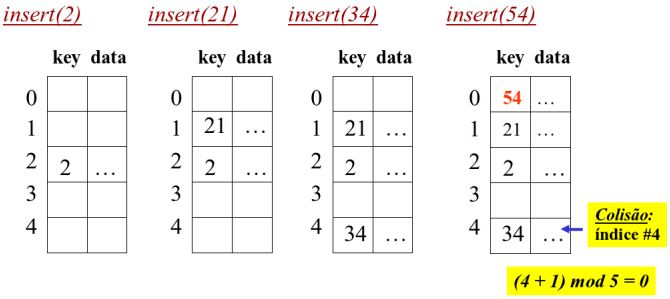
- Resolução de Colisões:
  - $i_0 = h(k)$
  - se índice/*bucket* ocupado, então:
  - $i_{j+1} = (i_j + c)\,\%\,m$
  - ... sucessivamente até encontrar um *bucket* livre.
  - o valor c pode ser contante (pesquisa linear), ou seguindo outra estratégia (quadrática, ...).



12.21

Open Addressing Hash Table: Exemplo

- $\boxed{h(k) = k\,\%\,m}$ com $\boxed{m = 5 \text{ e } k \in [0;99]}$



12.22