Dicionários / Tabelas de Dispersão I

17/11/2025

Ficheiro ZIP

- Está disponível no Moodle um ficheiro ZIP de suporte aos tópicos de hoje
- O tipo abstrato Hash Table usando Open Addressing
- Versões “simples” e exemplos de aplicação, que permitem trabalho autónomo de desenvolvimento e teste

Sumário

- Dicionários – Motivação
- Hash Tables – Tabelas de Dispersão
- Funções de Hashing
- Representação usando um array e Endereçamento Calculado – Open Addressing
- O TAD Hash Table (String, String)
- Exemplo : Contagem de ocorrências – O TAD HashTable(String, Int)
- Exercícios / Tarefas 

Dicionários – Motivação

Dicionário / Tabela de pares (chave,valor)

- Usar uma chave para aceder a um item / valor
- Chaves e itens / valores podem ser de qualquer tipo
- Chaves são comparáveis
- MAS, não há duas chaves iguais !!
- Idealmente, sem limite de tamanho / do número de pares (chave, valor)
- Chaves não existentes são associadas a um VALOR_NULO

- API simples / Código cliente simples

Dicionário – Aplicações

application	*key*	*value*
contacts	name	phone number, address
credit card	account number	transaction details
file share	name of song	computer ID
dictionary	word	definition
web search	keyword	list of web pages
book index	word	list of page numbers
cloud storage	file name	file contents
domain name service	domain name	IP address
reverse DNS	IP address	domain name
compiler	variable name	value and type
internet routing	destination	best route
...	...	...

[Sedgewick & Wayne]

Dicionário – Operações básicas

- Criar um dicionário vazio
- Registar um par (chave, valor) – **put**
  - Se chave ainda não existe, adicionar (chave, valor)
  - Se já existe, alterar o valor
- Consultar o valor associado a uma chave – **get**
- Verificar se uma chave pertence ao dicionário – **contains**
- Limpar / destruir

- EXTRA : iterar sobre todas as chaves (em ordem ?, como ?)

Java – Dicionários

- HashMap<>
- TreeMap<>
- LinkedHashMap<>

- Diferenças ?
- System.out.println(myMap); // O que acontece ?

Possíveis estruturas de dados – Fazer melhor?

	implementation	guarantee			average case			ordered ops?	key interface	
		search	insert	delete	search hit	insert	delete			
LISTA NÃO ORDENADA	sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		equals()	O(n)
ARRAY ORDENADO	binary search (ordered array)	$1 \lg N$	N	N	$1 \lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	compareTo()	
SEARCH TREE	BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	compareTo()	
BALANCED TREE	red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	compareTo()	O(log n)

[Sedgewick & Wayne]

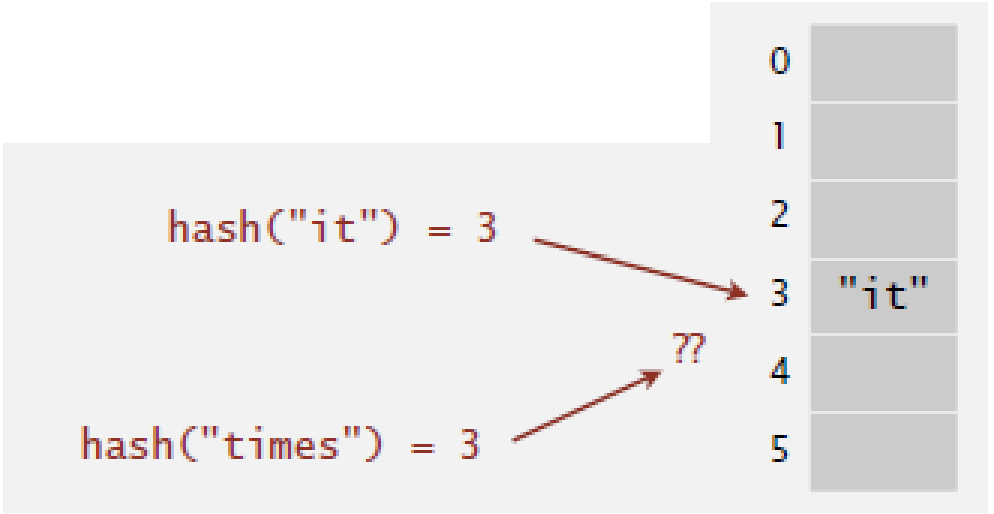
Hash Tables
– Tabelas de Dispersão

Hash Table – Tabela de Dispersão

- Estrutura de dados para armazenar pares (chave, valor)
- **Sem** chaves duplicadas
- **Sem** uma **ordem** implícita !!
- **MAS, com operações (muito) rápidas !!** 
- **Objetivo : O(1)**

Tabelas de Dispersão – Ideia

- Armazenar um item numa tabela/array indexada pela chave
  - Índice é função da chave !!
- Função de Hashing : para calcular o índice a partir da chave
  - Rapidez !!
- **Colisão :** 2 chaves diferentes originam o mesmo resultado / índice da tabela



[Sedgewick & Wayne]

Tabelas de Dispersão – Questões em aberto

- Como escolher a **função de hashing** ?
  - Rapidez + simplicidade
- Como calcular o ***hash value*** / **índice** ?
- Como verificar se dois **índices são iguais** ?
- Como resolver **colisões** ?
  - Método / estrutura de dados para armazenar itens com o mesmo valor de hashing
  - Rapidez !!
  - Memória adicional ?

Espaço de memória vs Tempo de execução

- Não há limitações memória : usar a chave diretamente como índice !!
- Não há restrições temporais : colisões resolvidas com procura sequencial

- MAS, o **espaço de memória é limitado** !!
- E pretendemos **operações em tempo quase-constante**, qualquer que seja a chave !!

- **Como fazer ?**

Funções de Hashing
– Funções de Dispersão

Funções de Hashing

- **Requisito :** se x == y, então hash(x) = hash(y) – Mesma chave
- **Desejável :** se x ≠ y, então hash(x) ≠ hash(y) – Chaves diferentes

- Exemplos simples
- int **hash(int x)** { return x; }
- int **hash(double x)** { long bits = doubleToLongBits(x); // 32 to 64 bits
  return (int) (bits ^ logicalShiftRight(bits, 32)); }

Outra Função de Hashing

- int hash(char* s) {

```
    int hash = 0;
    for (int i = 0; i < strlen(s); i++)          // Ponderar
        hash = s[i] + (31 * hash);               // todos os
    return hash;                                 // caracteres
}
```

- hash(“call”) = ?

Outra Função de Hashing

- int hash(char* s) {

```
int hash = 0;
for (int i = 0; i < strlen(s); i++)          // Ponderar
        hash = s[i] + (31 * hash);           // todos os
return hash;                                 // caracteres
}
```

- hash("call") = 3045982

  = 108 + 31 x (108 + 31 x (97 + 31 x (99))) Mét. de Horner

  = $99 \times 31^3 + 97 \times 31^2 + 108 \times 31^1 + 108 \times 31^0$

Funções de Hashing

- Há muitas funções de hashing para diferentes aplicações
  - Que outras aplicações conhecem ?
- Diferentes graus de complexidade
- Diferenças no desempenho computacional

- Tabelas de Hashing : privilegiar a **rapidez** e o **reduzido nº de colisões**

Conversão para índices da tabela

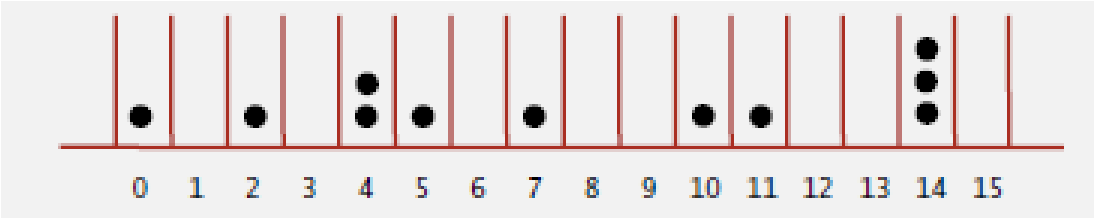
- Tabela de tamanho M
- Índices da tabela entre 0 e M – 1
- M é habitualmente um número primo ou uma potência de 2

- Como fazer ?

abs(hash(x)) % M

Distribuição equiprovável dos índices

- Assume-se a equiprobabilidade !!
- Cada chave tem a mesma probabilidade de ser mapeada num dos índices (0 a M – 1)
- O que acontece na prática ?



- Conhecem o **Paradoxo do Aniversário** ?

[Sedgewick & Wayne]

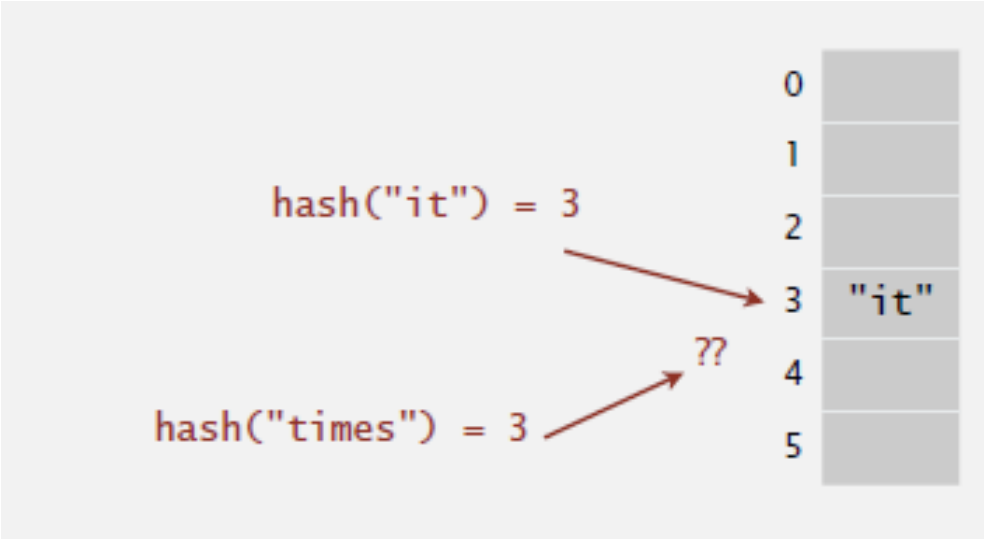
Hash Tables

- Endereçamento Calculado
- Open Addressing

Colisões – Como proceder ?

- Duas chaves distintas são mapeadas no mesmo índice da tabela !!
- Colisões são “evitadas” usando tabelas de muito grande dimensão !!

- Como gerir de modo eficiente ?
- Sem usar “demasiada” memória ?

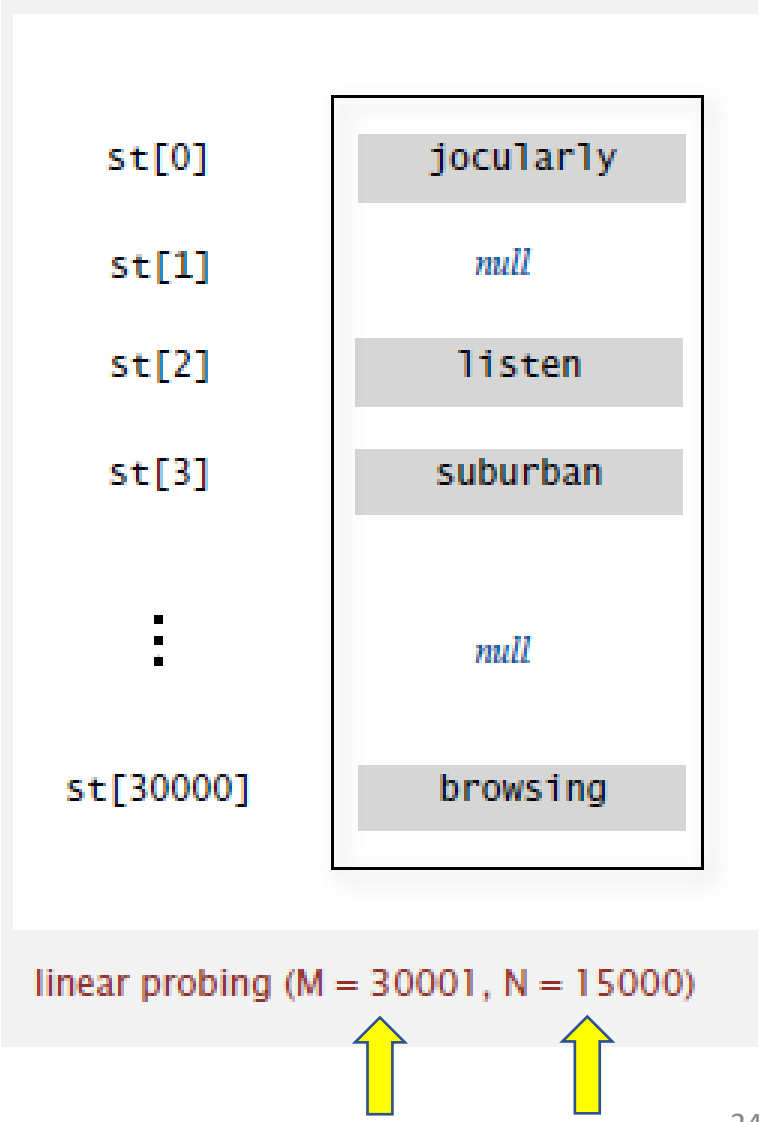


[Sedgewick & Wayne]

Open Addressing (IBM, 1953)

[Sedgewick & Wayne]

- Quando há uma colisão, procurar o espaço vago seguinte e armazenar o item – (chave, valor)
- Linear Probing – Sondagem Linear
- O tamanho da tabela (M) tem de ser maior do que o número de itens (N) !!
- Quantas vezes maior ??



Linear Probing – Sondagem Linear

- Aceder ao elemento de índice i
- Se necessário, tentar em (i + 1) % M, (i + 2) % M, etc.

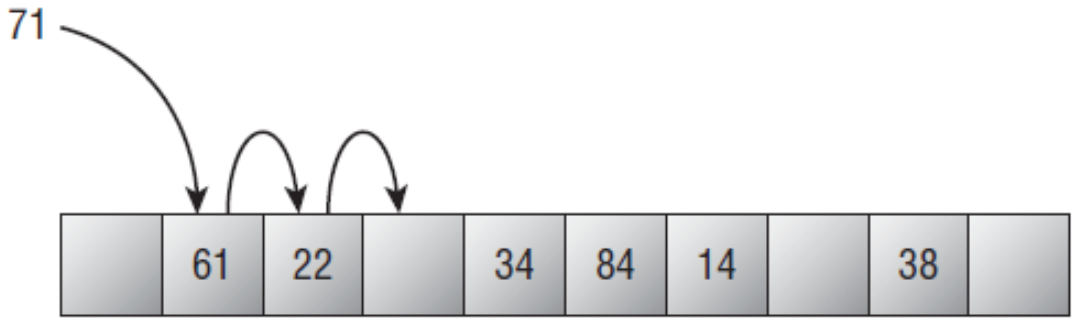


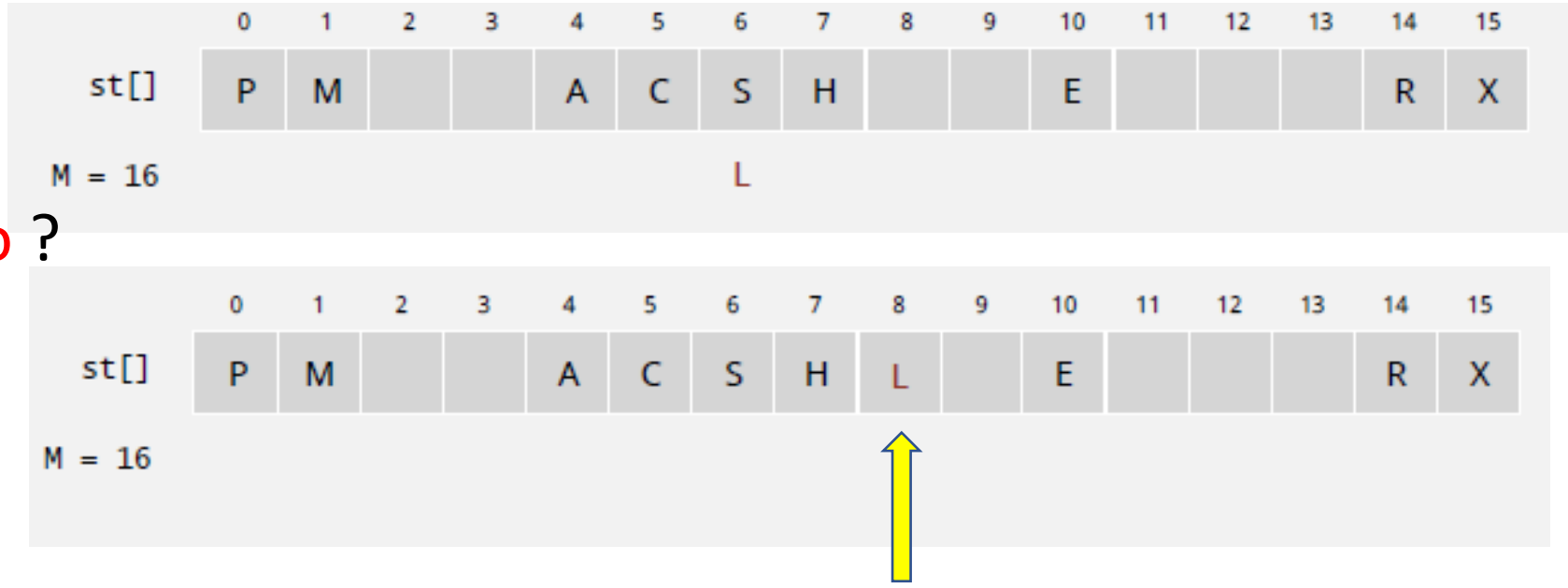
Figure 8-2: In linear probing, the algorithm adds a constant amount to locations to produce a probe sequence.

[Stephens]

Inserir na tabela – Linear Probing

- Guardar na posição i, se estiver disponível
- Caso contrário, tentar (i + 1) % M, (i + 2) % M, etc.

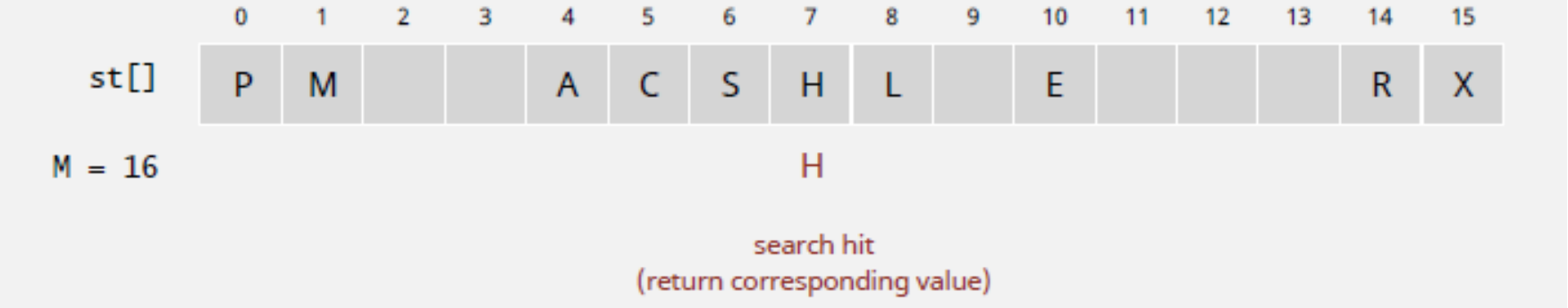
- Inserir L -> índice = 6
- Colisão !!
- Próximo espaço vago ?



[Sedgewick & Wayne]

Procurar na tabela – Linear Probing

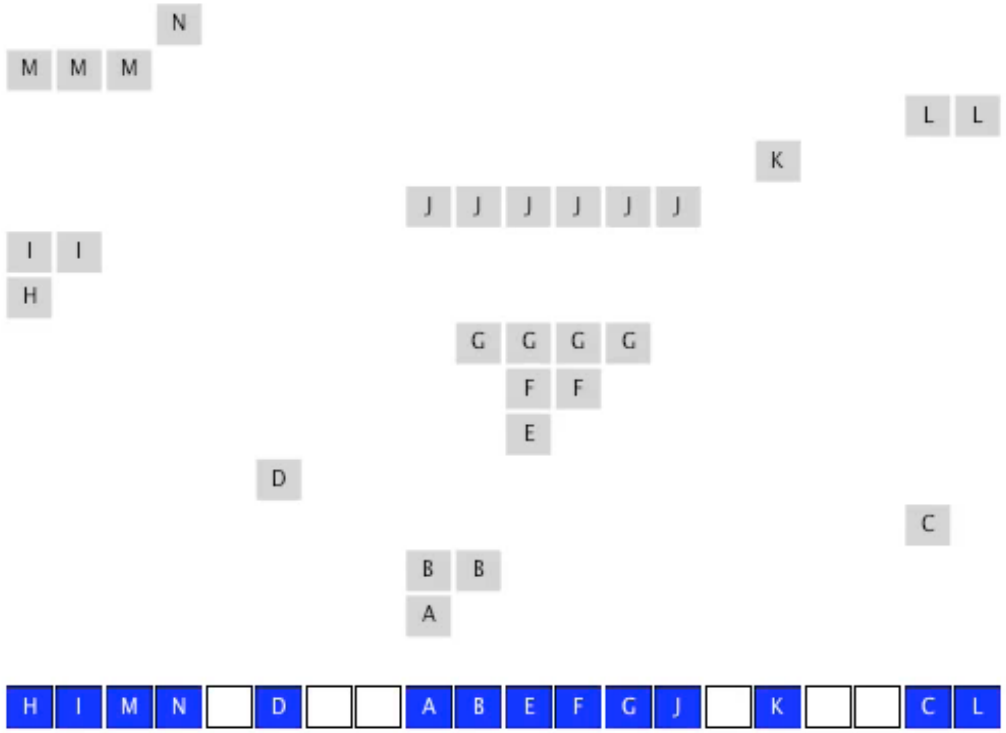
- Procurar na posição i
- Se estiver ocupada, verificar se as chaves são iguais
- Se forem diferentes, tentar em (i + 1) % M, (i + 2) % M, etc.
- Até encontrar a chave procurada ou chegar a um espaço vago
- Procurar H -> índice = 4 **-> 4 comparações**



[Sedgewick & Wayne]

Problema – Clustering

- Cluster : bloco de itens contíguos
- Novas chaves são indexadas no meio de "grandes" clusters
- E os itens colocados no final dos clusters
- O que não é "bom" para as operações de inserção e procura



[Sedgewick & Wayne]

Alternativa – Quadratic Probing

- Inserir na posição de índice i, se estiver disponível
- Caso contrário, tentar (i + 1) % M, (i + 4) % M, (i + 9) % M, etc.
- Redução do efeito de clustering

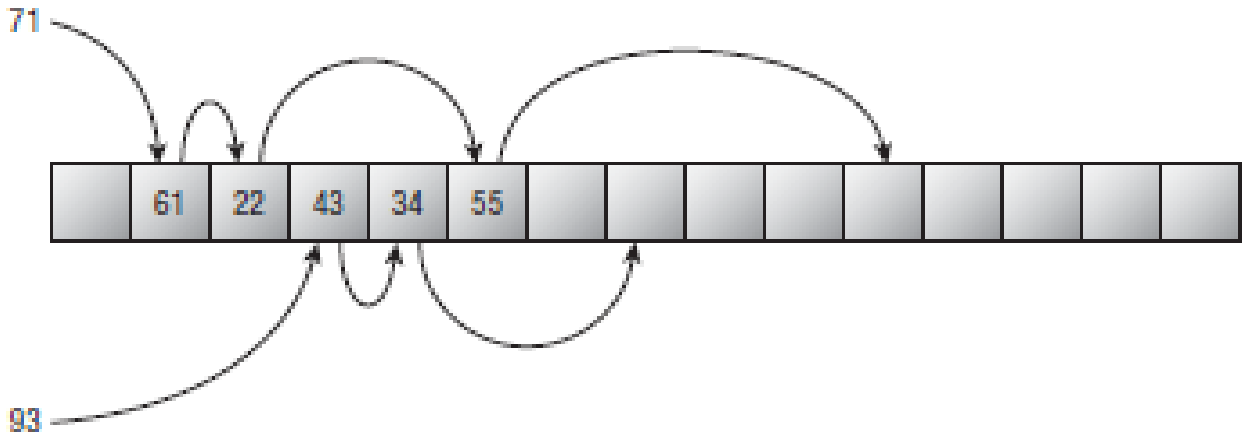


Figure 8-4: Quadratic probing reduces primary clustering.

[Stephens]

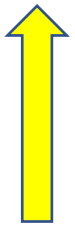
Análise – Linear Probing – Knuth, 1963

- Fator de carga – Load Factor $\lambda = N / M$
- Nº médio de tentativas para encontrar um item

  $1/2 \times (1 + 1/(1 - \lambda))$ -> 1.5, se $\lambda = 50\%$ -> 3, se $\lambda = 80\%$

- Nº médio de tentativas para inserir um item ou concluir que não existe

  $1/2 \times (1 + 1/(1 - \lambda)^2)$ -> 2.5, se $\lambda = 50\%$ -> 13, se $\lambda = 80\%$



Análise – Linear Probing

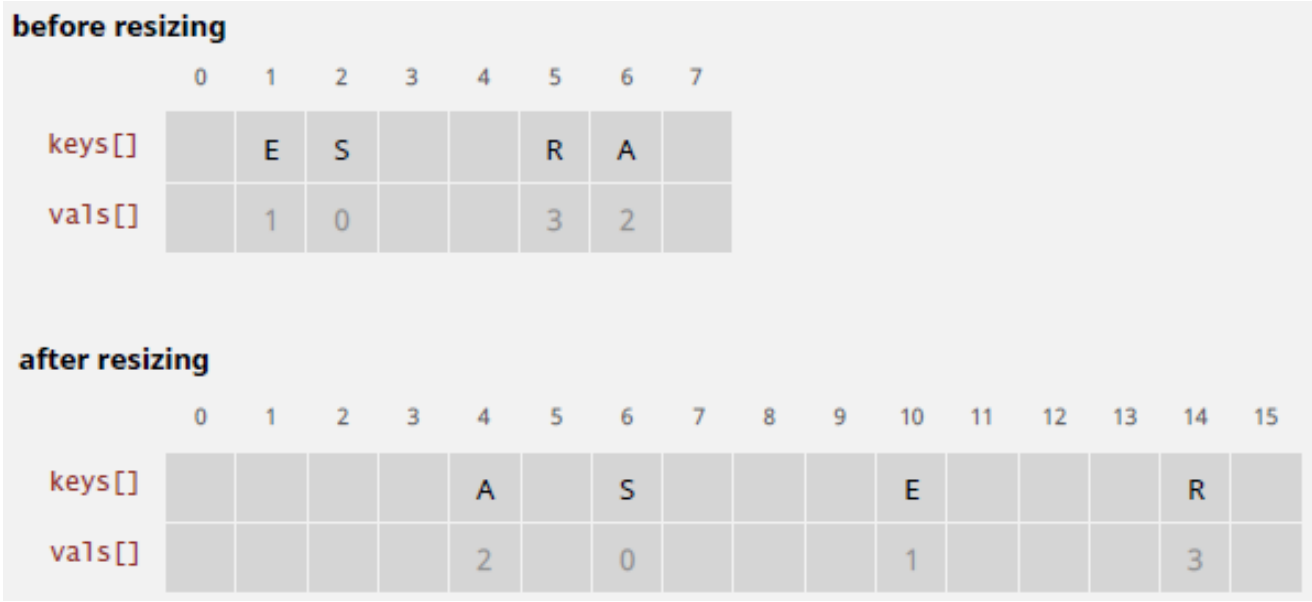
- M muito grande -> demasiados espaços vagos !!
- M "pequeno" -> tempo de procura aumenta muito !!

- Limiar habitual para o fator de carga : 50%
- Nº médio de tentativas para encontrar um item : 1,5 **hit**
- Nº médio de tentativas para inserir um item : 2,5 **miss**

- Como controlar ? RESIZING + REHASHING !!

Resizing + Rehashing

- Objetivo : fator de carga < 1/2
- Duplicar o tamanho do array quando fator de carga ≥ 1/2
- Reduzir para metade o tamanho do array quando fator de carga ≤ 1/8
- Criar a nova tabela e adicionar, um a um, todos os itens



[Sedgewick & Wayne]

Apagar um par (chave,valor) ?

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if hash(H) = 4

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

[Sedgewick & Wayne]

Ideia – Lazy Deletion

- Marcar inicialmente todos elementos da tabela como **livres**
- Ao inserir um item, o correspondente elemento fica **ocupado**
- Ao apagar um item, marcar esse elemento da tabela como **apagado**
- Para que qualquer **cadeia** que o use **não** seja **quebrada** !!

- E se possa continuar a procurar uma chave usando probing

- Quando **termina** uma procura ?
- Ao encontrar a chave procurada ou um elemento marcado como livre

Hash Table – Eficiência Computacional

- A complexidade temporal de uma procura é limitada inferiormente por **O(1)** e superiormente por **O(N)**

- **Pior Caso ?**
- Sequência de colisões
- Toda a tabela tem de ser percorrida e cada elemento consultado para encontrar a chave procurada !!
- Ou concluir que não existe na tabela !!
- **Não deve nunca ocorrer !!**

Hash Table – Eficiência Computacional

LISTA NÃO ORDENADA

ARRAY ORDENADO

SEARCH TREE

HASH TABLE

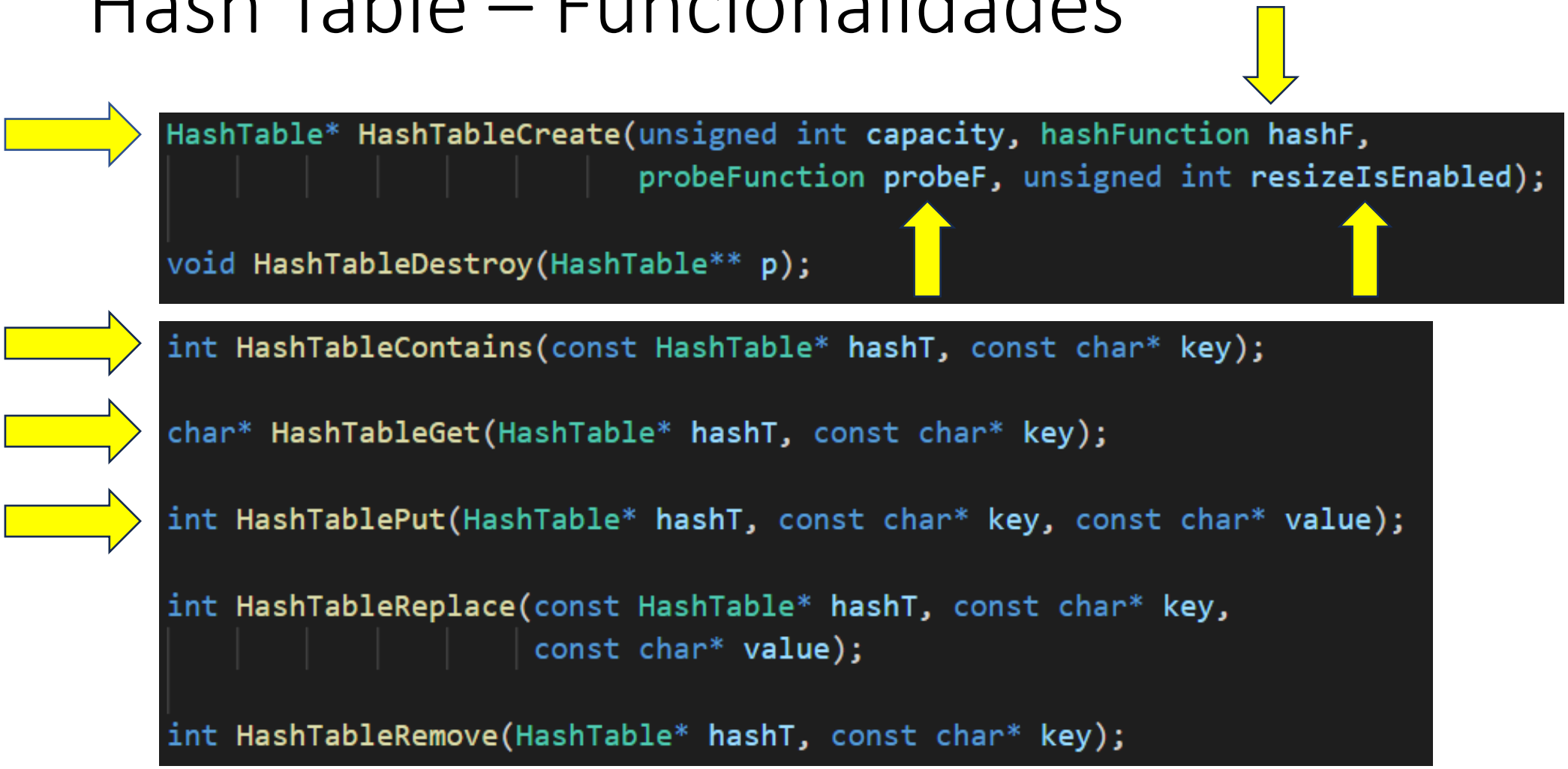
implementation	worst-case cost			average case cost (after N random inserts)		key interface
	search	insert	delete	search hit	insert	
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	`compareTo()`
BST	N	N	N	$1.4 \lg N$	$1.4 \lg N$	`compareTo()`
linear probing	N	N	N	3–5 *	3–5 *	`equals()` `hashCode()`

* under the uniform hashing assumption

[Sedgewick & Wayne]

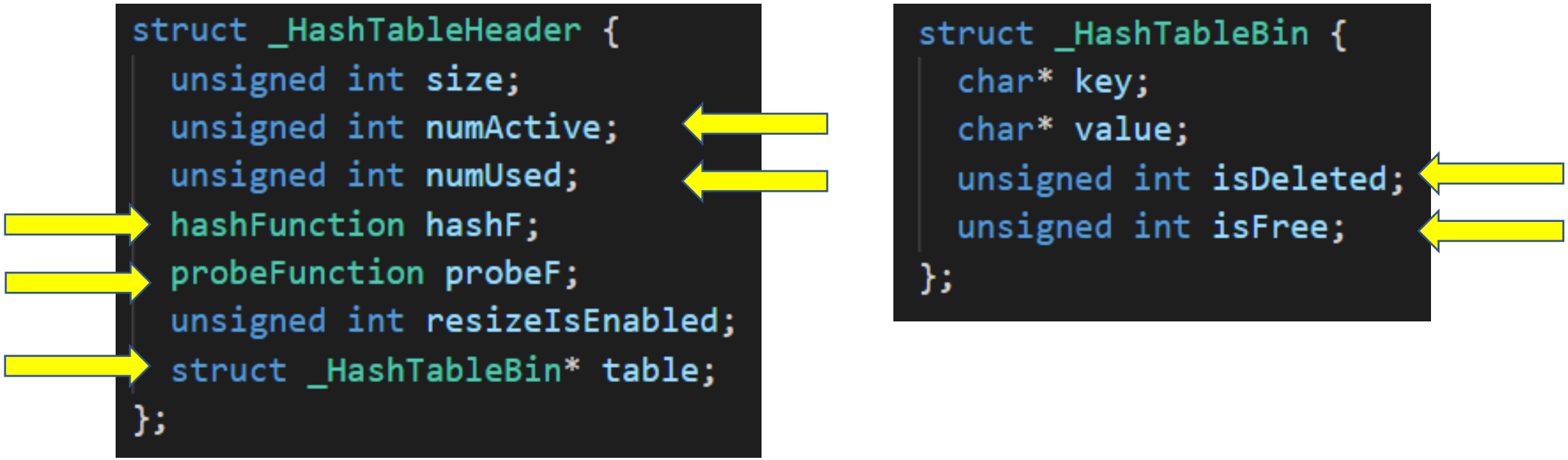
Exemplo
– Hash Table (String, String)

Hash Table – Funcionalidades



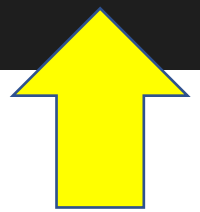
```
HashTable* HashTableCreate(unsigned int capacity, hashFunction hashF,
                           probeFunction probeF, unsigned int resizeIsEnabled);

void HashTableDestroy(HashTable** p);

int HashTableContains(const HashTable* hashT, const char* key);

char* HashTableGet(HashTable* hashT, const char* key);

int HashTablePut(HashTable* hashT, const char* key, const char* value);

int HashTableReplace(const HashTable* hashT, const char* key,
                     const char* value);

int HashTableRemove(HashTable* hashT, const char* key);
```

Hash Table – Cabeçalho e Elemento da Tabela



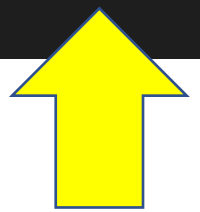
Exemplos – Funções de Hashing e Probing

```c
unsigned int hash1(const char* key) {
  assert(strlen(key) > 0);
  return key[0];
}

unsigned int hash2(const char* key) {
  assert(strlen(key) > 0);
  if (strlen(key) == 1) return key[0];
  return key[0] + key[1];
}
```



```c
unsigned int linearProbing(unsigned int index, unsigned int i,
                           unsigned int size) {
  return (index + i) % size;
}
```

```c
unsigned int quadraticProbing(unsigned int index, unsigned int i,
                              unsigned int size) {
  return (index + i * i) % size;
}
```



Hash Table – Procura de uma chave

```
for (unsigned int i = 0; i < hashT->size; i++) {
    index = hashT->probeF(hashKey, i, hashT->size);

    bin = &(hashT->table[index]);

    if (bin->isFree) {
        // Not in the table !
        return index;
    }

    if ((bin->isDeleted == 0) && (strcmp(bin->key, key) == 0)) {
        // Found it !
        return index;
    }
}
```

- Obter índice
- Consultar posição
- Elemento existe ?
- É a chave procurada ?

Exemplo – Meses de Ano – M = 17 – N = 12

```
size = 17 | Used = 12 | Active = 12
  0 - Free = 0 - Deleted = 0 - Hash =   68, 1st index =   0, (December, The last month of the year)
  1 - Free = 1 - Deleted = 0 -
  2 - Free = 0 - Deleted = 0 - Hash =   70, 1st index =   2, (February, The second month of the year)
  3 - Free = 1 - Deleted = 0 -
  4 - Free = 1 - Deleted = 0 -
  5 - Free = 1 - Deleted = 0 -
  6 - Free = 0 - Deleted = 0 - Hash =   74, 1st index =   6, (January, 1st month of the year)
  7 - Free = 0 - Deleted = 0 - Hash =   74, 1st index =   6, (June, 6th month)
  8 - Free = 0 - Deleted = 0 - Hash =   74, 1st index =   6, (July, 7th month)
  9 - Free = 0 - Deleted = 0 - Hash =   77, 1st index =   9, (March, 3rd month)
 10 - Free = 0 - Deleted = 0 - Hash =   77, 1st index =   9, (May, 5th month)
 11 - Free = 0 - Deleted = 0 - Hash =   79, 1st index =  11, (October, 10th month)
 12 - Free = 0 - Deleted = 0 - Hash =   78, 1st index =  10, (November, Almost at the end of the year)
 13 - Free = 1 - Deleted = 0 -
 14 - Free = 0 - Deleted = 0 - Hash =   65, 1st index =  14, (April, 4th month)
 15 - Free = 0 - Deleted = 0 - Hash =   65, 1st index =  14, (August, 8th month)
 16 - Free = 0 - Deleted = 0 - Hash =   83, 1st index =  15, (September, 9th month)
```

Exemplo

- Contagem de Ocorrências
- Hash Table (String, Int)

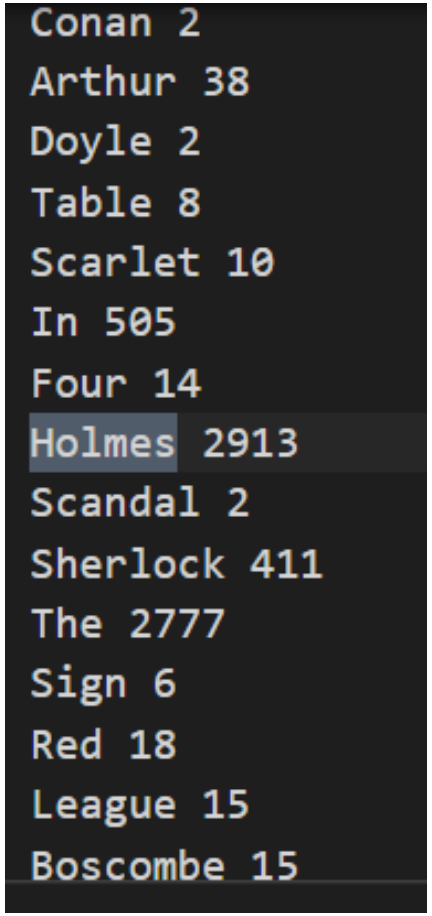
Objetivo – Contar o número de ocorrências

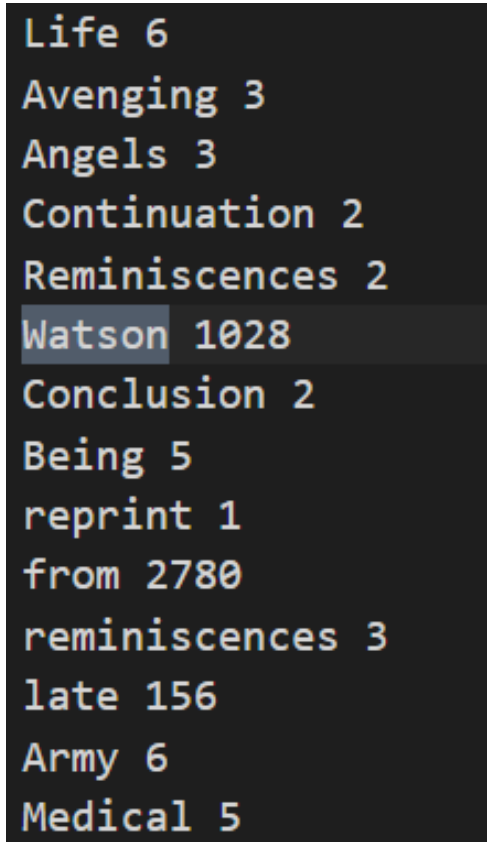
- Dado um ficheiro de texto
- Contar o nº de ocorrências de cada palavra

- Não se conhece, à partida, qual o número de palavras distintas !!

- Chave : palavra
- Valor : nº de ocorrências 

Exemplo – Palavras de uma coleção de livros





Exercícios / Tarefas

Exercício 1 – Escolha múltipla

Numa tabela de dispersão (***"Hash Table"***) implementada usando endereçamento calculado **(*"Open Addressing"***),

a) eventuais colisões, durante a inserção de um novo elemento, são resolvidas efetuando deslocamentos e determinando a posição do *array* em que o novo elemento vai ser inserido.

b) o factor de carga (***"Load Factor"***) poderá ser superior à unidade.

c) Ambas estão corretas.

d) Nenhuma está correta.

Tarefa 1 – HashTable(String, String)

- Analisar o simples programa de teste
- Executá-lo e analisar o output
- Analisar as funções do TAD HashTable(String, String)

Tarefa 2 – HashTable(String, Int)

- Analisar o programa de aplicação
- Executá-lo e analisar o output
- Analisar as funções do TAD HashTable(String, Int)

- Escolher vários textos e contar as suas palavras distintas
- Melhorar o processamento das palavras lidas
  - Por exemplo, converter maiúsculas em minúsculas
- Não contar "stop words"
- Obter uma listagem ordenada – Como fazer ??