

Dicionários / Tabelas de Símbolos II

Joaquim Madeira

19/05/2020

Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- O tipo abstrato **Hash Table** usando **Separate Chaining**
- **Versão “simples”**, que permite trabalho autónomo de desenvolvimento e teste

Sumário

- Recap
- Exemplo de aplicação – **contagem de ocorrências** usando uma Hash Table
- Hash Tables – Representação usando **Separate Chaining**
- Análise detalhada do TAD Hash Table – Separate Chaining
- Desempenho computacional

Let's
RECAP

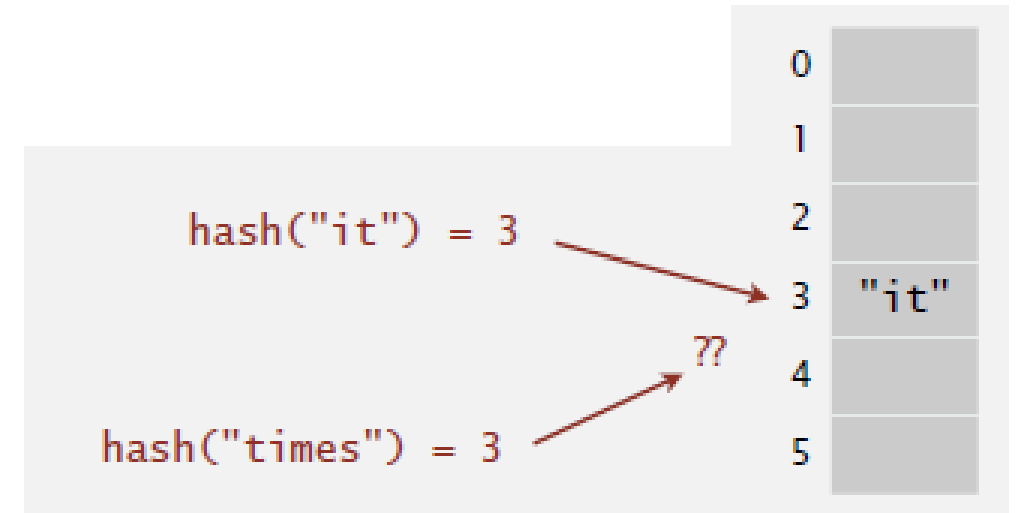
Recapitulação

TAD Dicionário / Tabela de Símbolos

- Usar **chaves** para aceder a **itens / valores**
- Chaves e itens / valores podem ser de **qualquer tipo**
- **Chaves** são **comparáveis**
- MAS, **não** há duas chaves **iguais** !!
- **Sem limite** de tamanho / do número de pares (chave, valor)
- Chaves não existentes são associadas a um **VALOR_NULO**
- API simples / Código cliente simples

Hash Tables – Tabelas de Dispersão

- Armazenar itens 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** da função de hashing



[Sedgewick & Wayne]

Linear Probing

- Aceder à **posição i**
- Se necessário, tentar em $(i + 1) \% M$, $(i + 2) \% M$, etc.

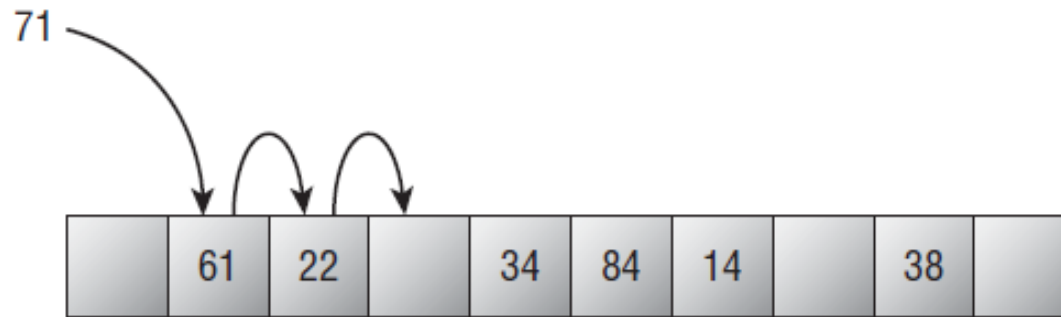


Figure 8-2: In linear probing, the algorithm adds a constant amount to locations to produce a probe sequence.

[Stephens]

Quadratic Probing

- Aceder à **posição i**
- Se necessário, tentar em $(i + 1) \% M$, $(i + 4) \% M$, $(i + 9) \% M$, etc.

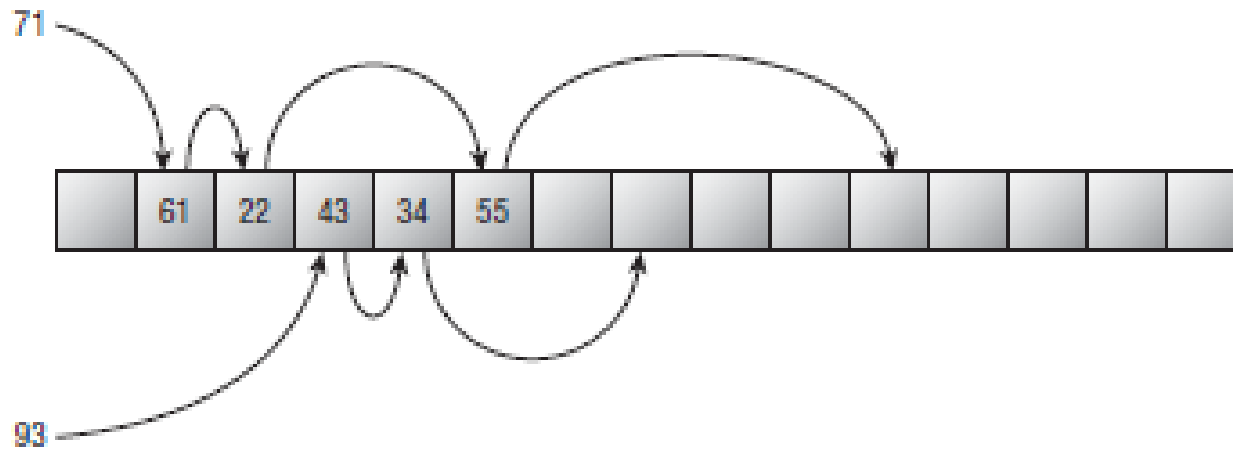


Figure 8-4: Quadratic probing reduces primary clustering.

[Stephens]

Análise – Linear Probing – Knuth, 1963

- Fator de carga – Load Factor $\lambda = N / M$
- Nº médio de tentativas para encontrar um item
 $1/2 \times (1 + 1/(1 - \lambda))$ $\rightarrow 3/2$, se $\lambda = 1/2$ $\rightarrow 3$, se $\lambda = 4/5$
- Nº médio de tentativas para inserir um item ou concluir que não existe
 $1/2 \times (1 + 1/(1 - \lambda)^2)$ $\rightarrow 5/2$, se $\lambda = 1/2$ $\rightarrow 13$, se $\lambda = 4/5$



Resizing + Rehashing

- **Objetivo** : fator de carga $\leq 1/2$
- **Duplicar o tamanho** do array quando fator de carga $\geq 1/2$
- **Reduzir para metade** o tamanho do array quando fator de carga $\leq 1/8$
- Criar a nova tabela e **adicionar**, um a um, todos os **itens**

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

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

[Sedgewick & Wayne]

Apagar um item (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

after deleting S ?

doesn't work, e.g., if $\text{hash}(H) = 4$

	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]

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**

TAD Hash Table

```
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);
```

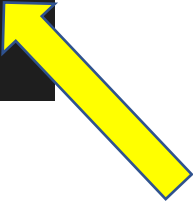


Estrutura de dados

```
struct _HashTableHeader {  
    unsigned int size;  
    unsigned int numActive;  
    unsigned int numUsed;  
    hashFunction hashF;  
    probeFunction probeF;  
    unsigned int resizeIsEnabled;  
    struct _HashTableBin* table;  
};
```



```
struct _HashTableBin {  
    char* key;  
    char* value;  
    unsigned int isDeleted;  
    unsigned int isFree;  
};
```



Funções auxiliares para testes

```
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];  
}
```

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

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

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;  
    }  
}
```




Exemplo – $M = 17 - N = 12$

```
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

Aplicação – Contagem

- Dado um ficheiro de **texto**
- Contar o **nº de ocorrências** de cada palavra
- Não se conhece, à partida, qual o **nº de palavras distintas !!**
- **Chave** : palavra
- **Valor** : nº de ocorrências 

Exemplo

```
Conan 2
Arthur 38
Doyle 2
Table 8
Scarlet 10
In 505
Four 14
Holmes 2913
Scandal 2
Sherlock 411
The 2777
Sign 6
Red 18
League 15
Boscombe 15
```

```
Life 6
Avenging 3
Angels 3
Continuation 2
Reminiscences 2
Watson 1028
Conclusion 2
Being 5
reprint 1
from 2780
reminiscences 3
late 156
Army 6
Medical 5
```

Tarefas

- Analisar as funções desenvolvidas
- E o programa de aplicação
- 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 ??

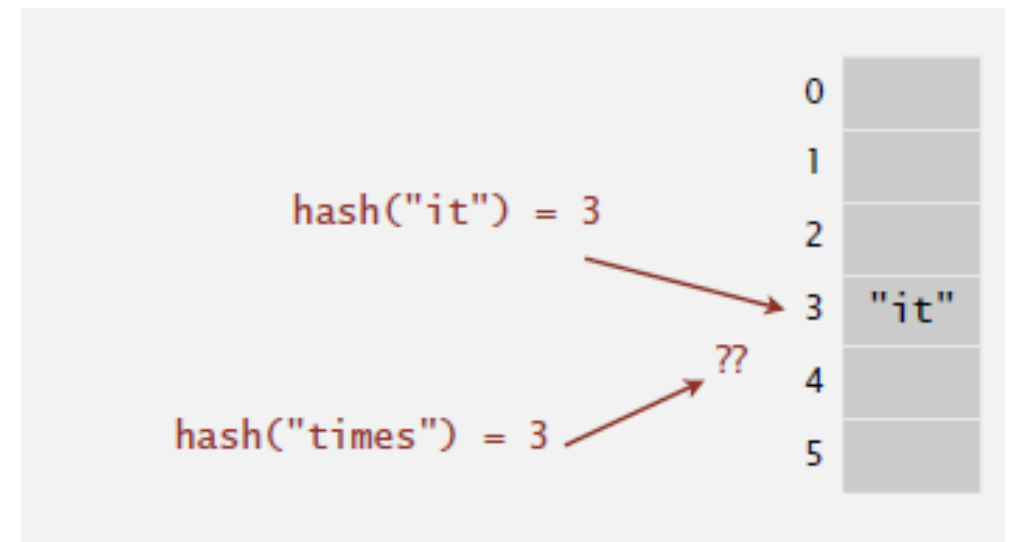


Hash Tables

- Separate Chaining

Colisões – Como proceder ?

- Duas **chaves distintas** são mapeadas no **mesmo índice** da tabela !!
- Como gerir de modo eficiente ?
- Sem usar “demasiada” memória !!
- **Alternativa** ao Open Addressing ?

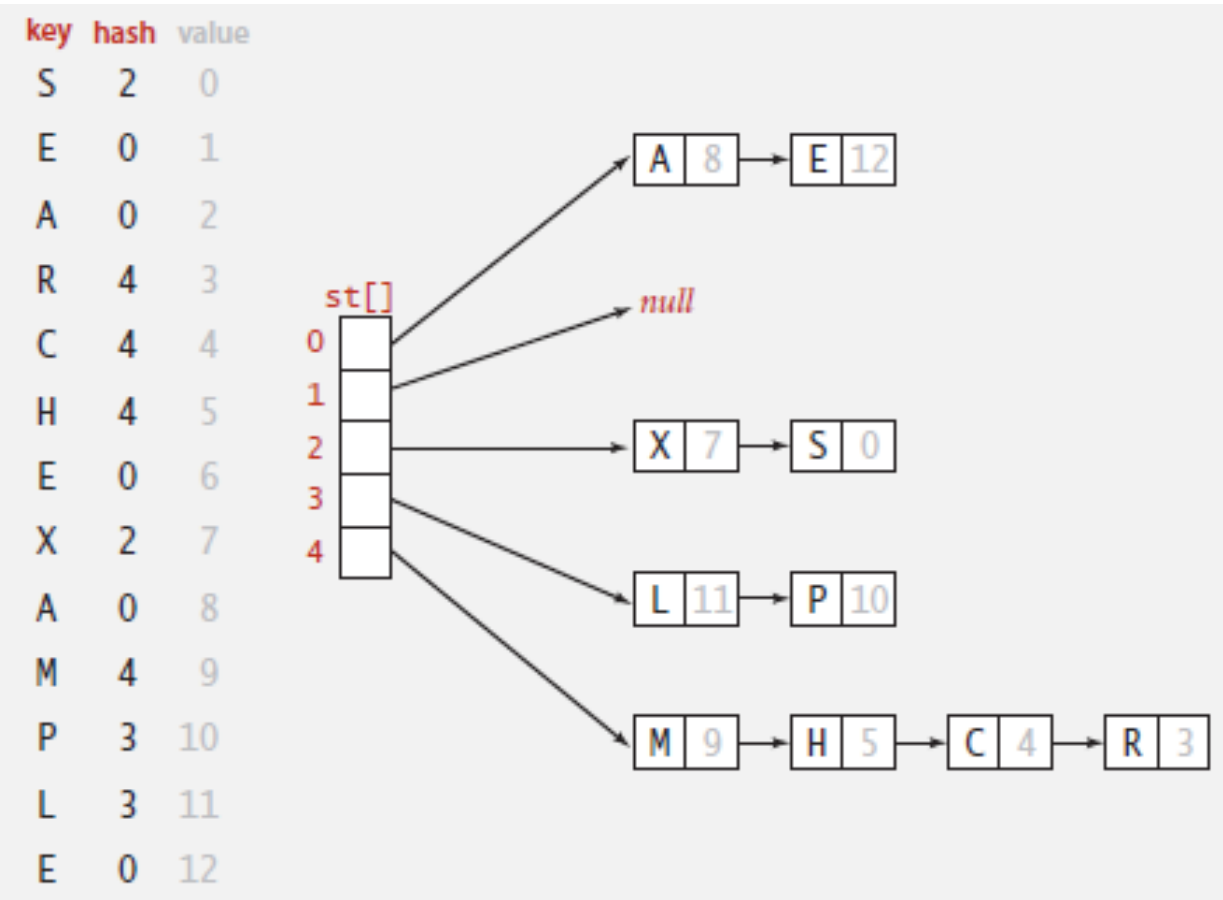


[Sedgewick & Wayne]

Separate Chaining (IBM, 1953)

- Array de **M < N itens**
- Mapear a chave em $[0..(M - 1)]$
- Inserir no **início** de uma cadeia, se não existir
- Procurar **só numa cadeia**

[Sedgewick & Wayne]



Separate Chaining

```
struct _HashTableHeader {  
    unsigned int size;  
    unsigned int numBins;  
    hashFunction hashF;  
    List** table;  
};
```

```
struct _HashTableBin {  
    char* key;  
    char* value;  
};
```

Procurar

```
// Search for the key
// If found, the list current node is updated
//
static int _searchKeyInList(List* l, char* key) {
    if (ListIsEmpty(l)) {
        return 0;
    }

    // Needed for the comparator
    // Shallow copy of the key: just the pointer
    struct _HashTableBin searched;
    searched.key = key;

    ListMoveToHead(l);
    return ListSearch(l, &searched) != -1;
}
```

Inserir

```
int HashTablePut(HashTable* hashT, char* key, char* value) {
    unsigned int index = hashT->hashF(key) % hashT->size;
    List* l = hashT->table[index];

    if (_searchKeyInList(l, key) == 1) {
        // FOUND, cannot be added to the table
        return 0;
    }

    // Does NOT BELONG to the table
    // Insert a new bin in the list

    struct _HashTableBin* bin = (struct _HashTableBin*)malloc(sizeof(*bin));
    bin->key = (char*)malloc(sizeof(char) * (1 + strlen(key)));
    strcpy(bin->key, key);
    bin->value = (char*)malloc(sizeof(char) * (1 + strlen(value)));
    strcpy(bin->value, value);

    ListInsert(l, bin);
    hashT->numBins++;

    return 1;
}
```

Substituir

```
int HashTableReplace(const HashTable* hashT, char* key, char* value) {
    unsigned int index = hashT->hashF(key) % hashT->size;
    List* l = hashT->table[index];

    // Search and update current, if found
    if (_searchKeyInList(l, key) == 0) {
        return 0;
    }

    struct _HashTableBin* bin = ListGetCurrentItem(l);

    free(bin->value);
    bin->value = (char*)malloc(sizeof(char) * (1 + strlen(value)));
    strcpy(bin->value, value);

    return 1;
}
```

Separate Chaining

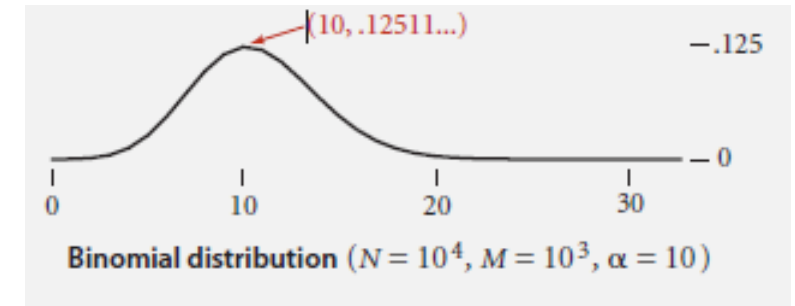
```
size = 17 | Active = 12
0 -
  | Hash = 68, (December, 12th month)
1 -
2 -
  | Hash = 70, (February, 2nd month of the year)
3 -
4 -
5 -
6 -
  | Hash = 74, (January, 1st month of the year)
  | Hash = 74, (July, 7th month)
  | Hash = 74, (June, 6th month)
7 -
8 -
9 -
  | Hash = 77, (March, 3rd month)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

Open Addressing + Linear Probing

```
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)
```

Análise

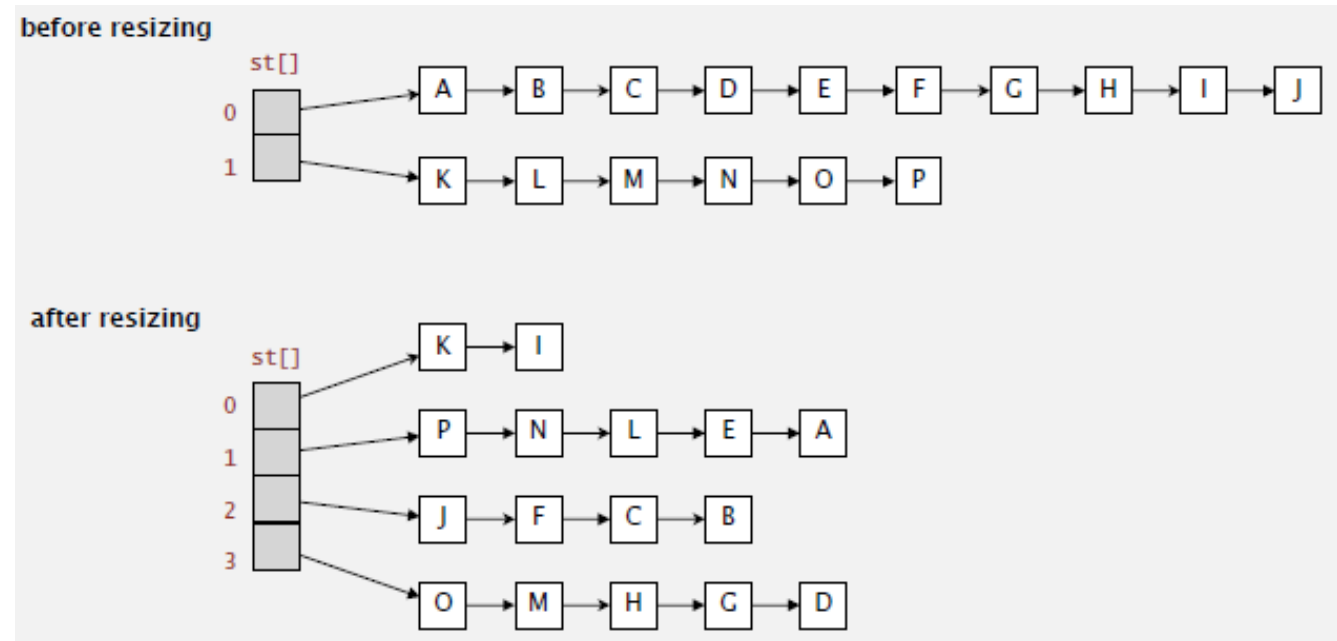
- Em média, N/M itens por cadeia – **Load Factor**
- Procurar / inserir \rightarrow nº de **comparações** é proporcional a N/M
 - M vezes mais rápido que na procura sequencial
- M muito **grande** \rightarrow demasiadas **cadeias vazias**
- M demasiado **pequeno** \rightarrow cadeias muito **longas**
- Escolha habitual : $M \approx N/4$ \rightarrow **$O(1)$**



[Sedgewick & Wayne]

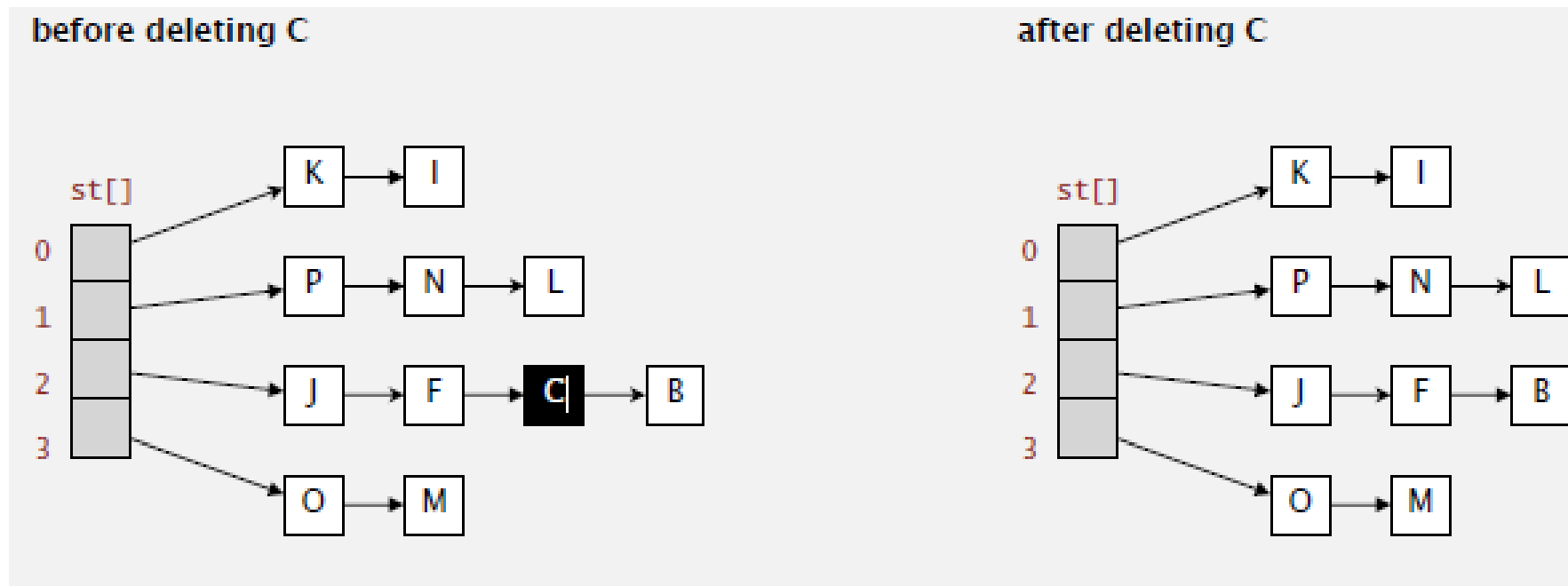
Resizing + Rehashing

- **Objetivo** : fator de carga aprox. constante
- **Duplicar o tamanho** do array quando $N/M \geq 8$
- **Reduzir para metade** o tamanho do array quando $N/M \leq 2$
- Criar a nova tabela e **adicionar**, um a um, todos os **itens**



[Sedgewick & Wayne]

Apagar é fácil



[Sedgewick & Wayne]

Apagar

```
int HashTableRemove(HashTable* hashT, char* key) {
    unsigned int index = hashT->hashF(key) % hashT->size;
    List* l = hashT->table[index];

    // Search and update current, if found
    if (_searchKeyInList(l, key) == 0) {
        return 0;
    }

    // Get rid of the bin
    struct _HashTableBin* bin = ListGetCurrentItem(l);
    free(bin->key);
    free(bin->value);
    free(bin);

    // Get rid of the list node
    ListRemoveCurrent(l);
    hashT->numBins--;

    return 1;
}
```

Tarefa

- Implementar uma função para fazer **Resizing + Rehashing**
- Adaptar o tamanho da tabela à evolução do **fator de carga**

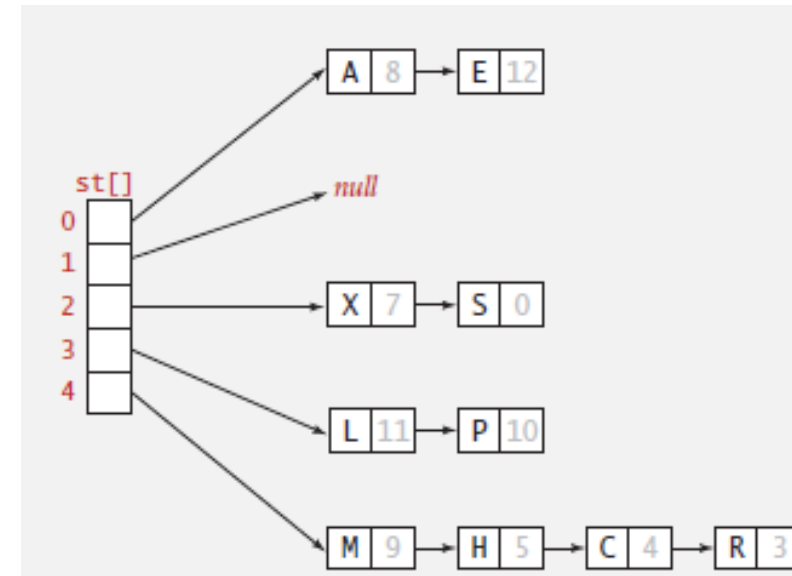
Eficiência

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
separate chaining	N	N	N	$3-5^*$	$3-5^*$	$3-5^*$		<code>equals()</code> <code>hashCode()</code>
linear probing	N	N	N	$3-5^*$	$3-5^*$	$3-5^*$		<code>equals()</code> <code>hashCode()</code>
* under uniform hashing assumption								

[Sedgewick & Wayne]

Separate Chaining **vs** Linear Probing

- **Separate Chaining**
 - Desempenho não se degrada abruptamente
 - Pouco sensível a funções de hashing menos boas
-
- **Linear Probing**
 - Menos espaço de memória desperdiçado



	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

[Sedgewick & Wayne]

Hash Tables vs Balanced Search Trees

- Tabelas de Dispersão
- Código mais simples
- Melhor alternativa se não pretendermos ordem
- Mais rápidas, para chaves simples
- Árvores Binárias Equilibradas
- Pior caso : $O(\log N)$ vs $O(N)$
- Suportam ordem
- `compareTo()` vs `equals()` + `hashCode()`