


# Dicionários / Tabelas de Dispersão II

19/11/2025

# 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
- **Hash Tables** – Representação usando **Separate Chaining**
- Análise detalhada do TAD Hash Table – Separate Chaining
- Desempenho computacional
- Exercícios / Tarefas 

Let's  
RECAP

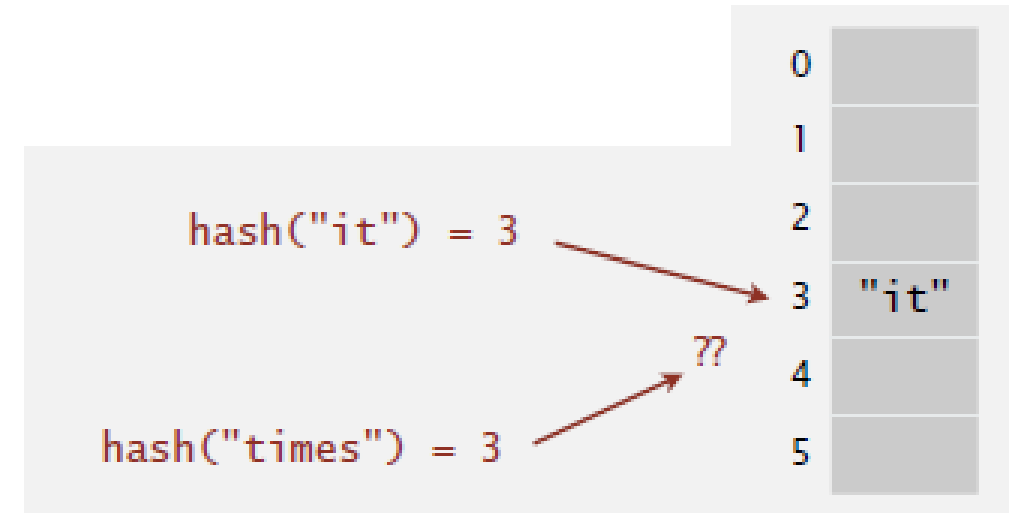
# Recapitulação

# 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 :  $O(1)$**

# Hash Table – Open Addressing

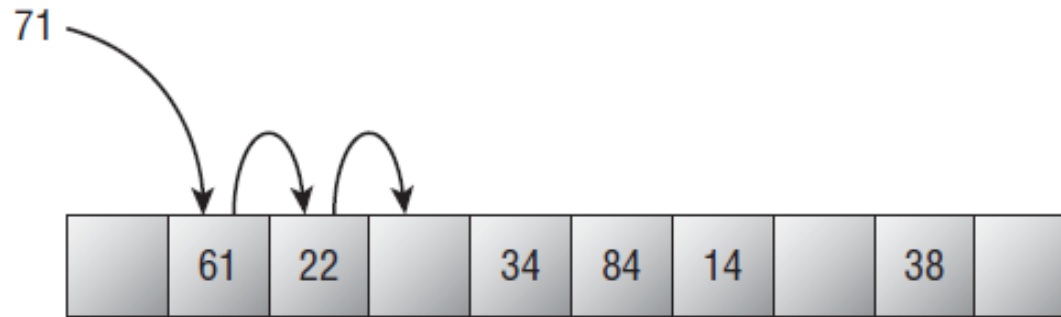
- Armazenar um **item** numa **tabela/array indexada** pela **chave**
  - Índice é função da chave !!
- **Função de Hashing** : para calcular o **índice** a partir da **chave**
  - Rapidez !!
- **Colisão** : 2 **chaves diferentes** originam o **mesmo resultado / índice da tabela**



[Sedgewick & Wayne]

# 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** ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
M = 16	L															

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16																



[Sedgewick & Wayne]



# 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)

# 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  $\lambda = 80\%$



# Resizing + Rehashing

- **Objetivo** : fator de carga  $< 1/2$
- **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]

# 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);
```

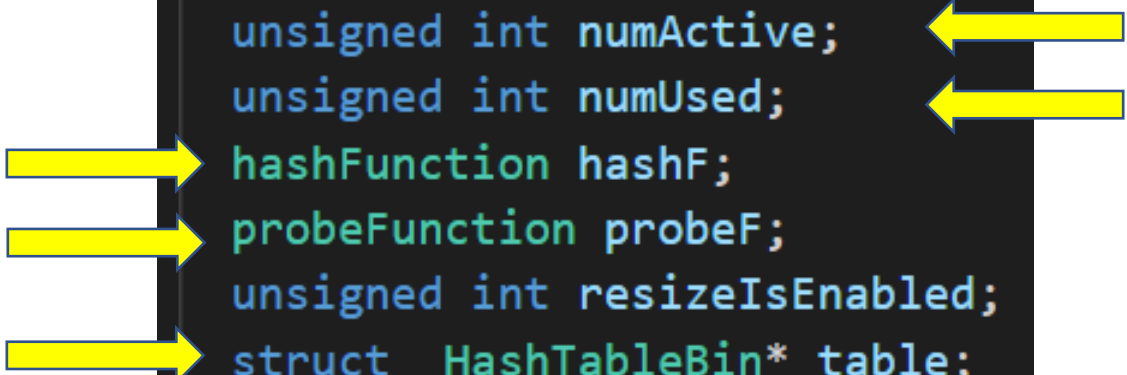
```
int HashTableReplace(const HashTable* hashT, const char* key,  
                    const char* value);
```

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

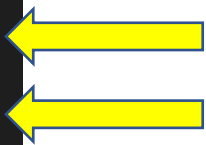


# Hash Table – Cabeçalho e Elemento da Tabela

```
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;  
};
```



# 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 ?

# Hash Table – Consulta dada uma chave

```
char* HashTableGet(HashTable* hashT, const char* key) {  
    int index = _searchHashTable(hashT, key);  
    if (index == -1 || hashT->table[index].isFree == 1) {  
        // NOT FOUND  
        return NULL;  
    }  
  
    struct _HashTableBin* bin = &(hashT->table[index]);  
    char* result = (char*)malloc(sizeof(char) * (1 + strlen(bin->value)));  
    strcpy(result, bin->value);  
  
    return result;  
}
```



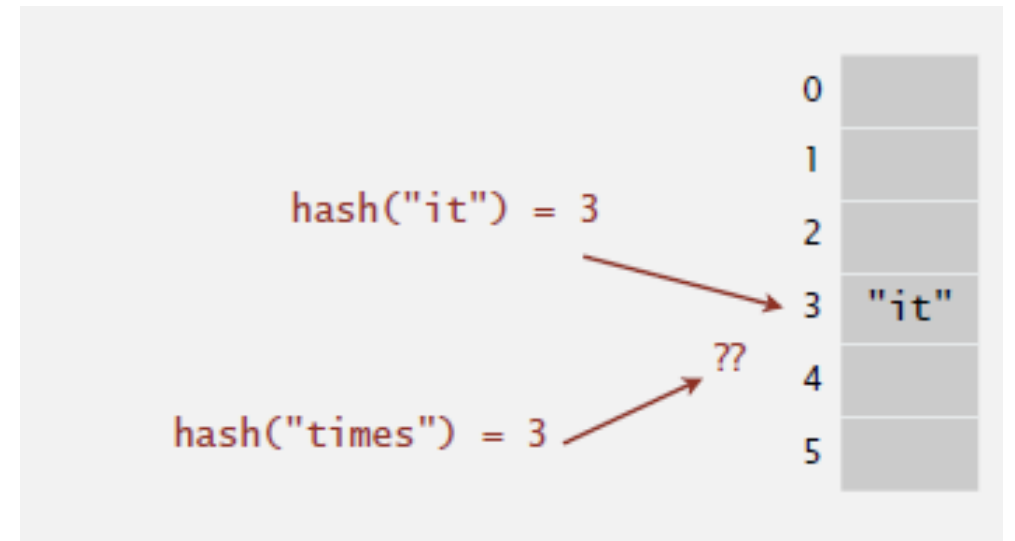
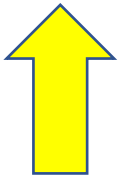


# Hash Tables

## – Separate Chaining

# Hash Table – 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 ?

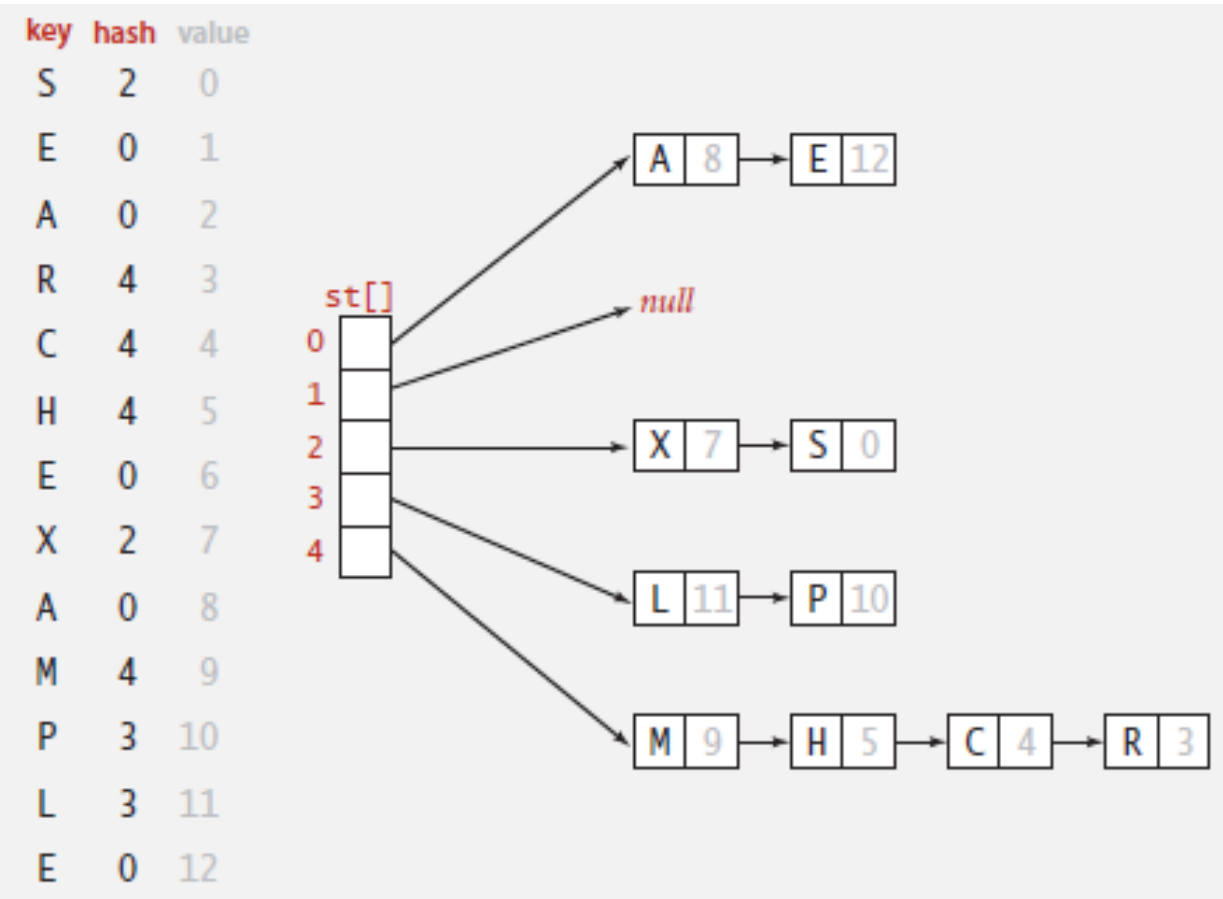


[Sedgewick & Wayne]

# Hash Table – Separate Chaining (IBM, 1953)

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

[Sedgewick & Wayne]

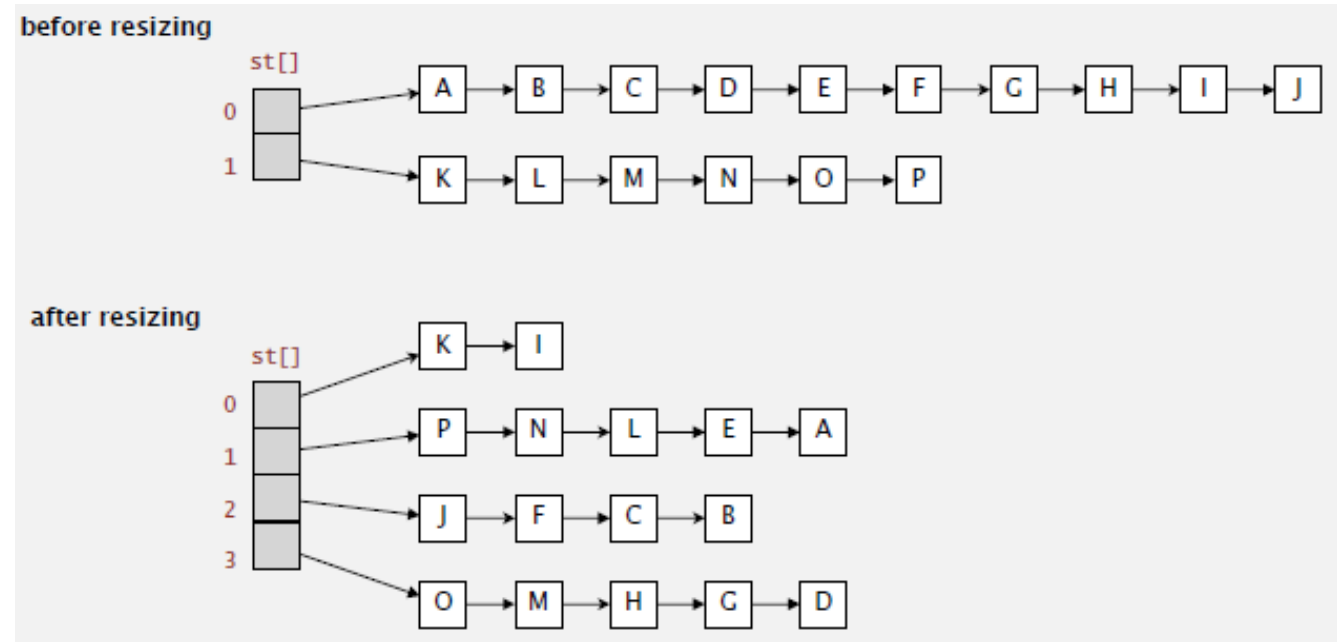


# Eficiência Computacional

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

# 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]

# Exemplo


- Hash Table (String, String)

# Hash Table – Cabeçalho e Par (chave, valor)

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



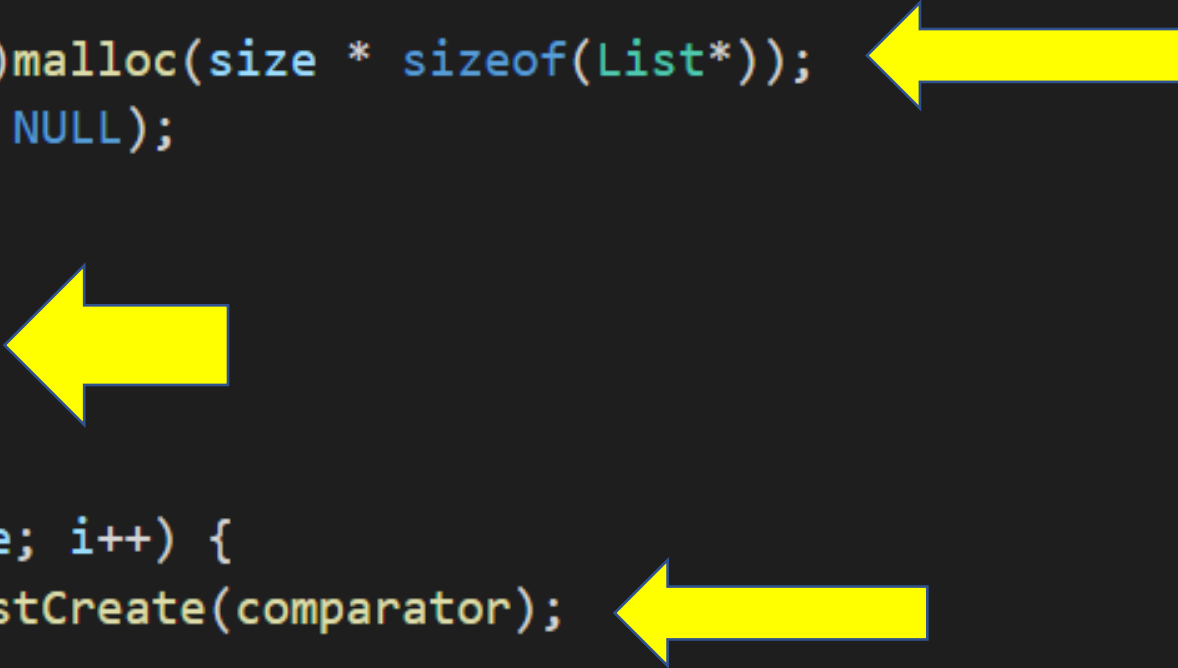
```
struct _HashTableBin {  
    char* key;  
    char* value;  
};
```



- Usar o **TAD Sorted List** para instanciar cada uma das listas da tabela
  - Reutilização – Para facilitar o desenvolvimento
- Cada **nó de uma lista** referencia um **par (chave, valor) – HT bin**

# HashTable – Construtor – Criar listas vazias



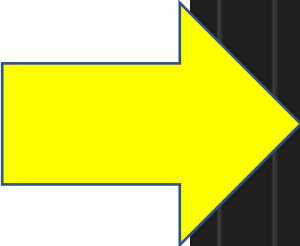
```
HashTable* hTable = (HashTable*)malloc(sizeof(struct _HashTableHeader));  
assert(hTable != NULL);  
hTable->table = (List**)malloc(size * sizeof(List*));  
assert(hTable->table != NULL);  
  
hTable->size = size;  
hTable->numBins = 0;  
hTable->hashF = hashF;  
  
for (int i = 0; i < size; i++) {  
    hTable->table[i] = ListCreate(comparator);  
}
```





# HashTable – Destrutor

```
for (int i = 0; i < t->size; i++) {  
    List* l = t->table[i];  
    // Free the HT bins of each list  
    while (ListIsEmpty(l) == 0) {  
        struct _HashTableBin* bin = ListRemoveHead(l);  
        free(bin->key);  
        free(bin->value);  
        free(bin);  
    }  
    // Destroy the list header  
    ListDestroy(&(t->table[i]));  
}  
free(t->table);  
free(t);
```



- Percorrer cada **lista**
- **Libertar** o espaço de memória atribuído a cada **HT bin** e **nó**
- **Libertar** o espaço de memória atribuído ao **cabeçalho da lista**
- **Libertar** o espaço de memória atribuído à **tabela**

# HashTable – Procurar

- Procurar a chave na correspondente lista de HT bins
- A lista está vazia ?
- Procurar a partir do início da lista

```
// 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

- Calcular o índice da tabela
- Procurar na lista usando a chave
- Ainda não existe ?
- Criar novo HT bin
- Copiar a chave e o valor
- Inserir na lista

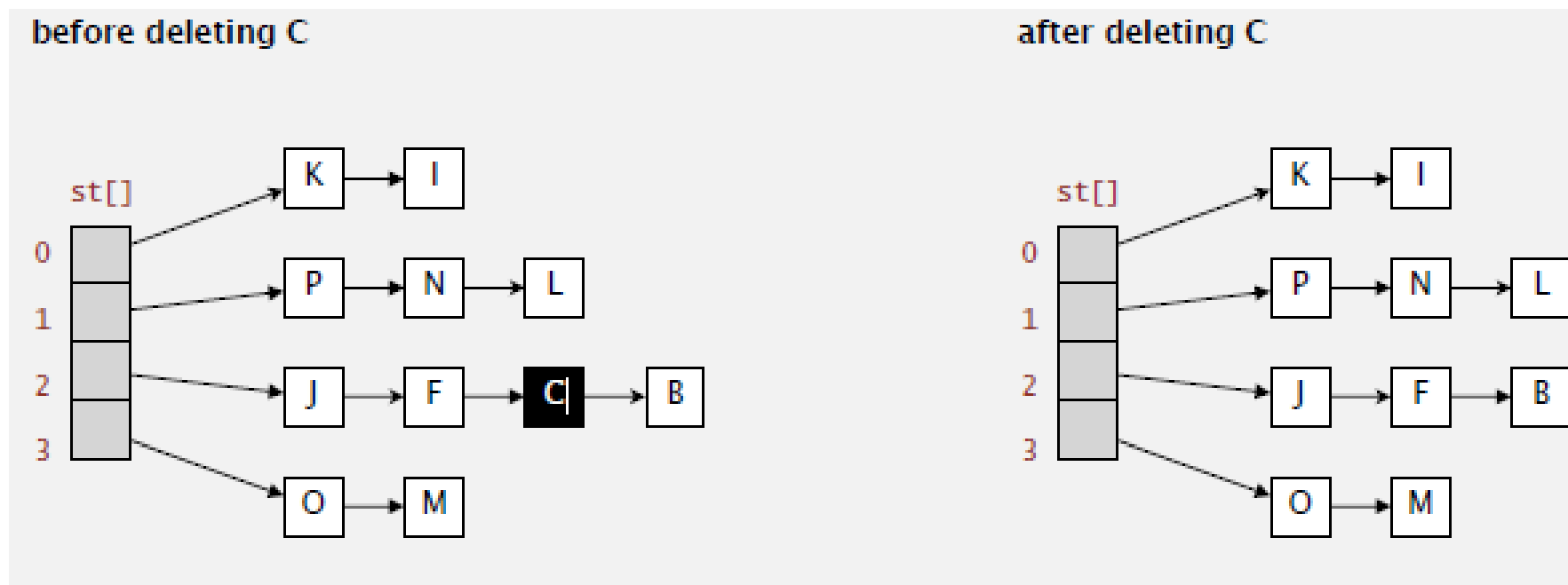
```
int HashTablePut(HashTable* hashT, char* key, char* value) {  
    unsigned int index = hashT->hashF(key) % hashT->size;  
    List* l = hashT->table[index];  
  
    if (_searchKeyInList(l, key) == 1) {  
        // FOUND, cannot be added to the table  
        return 0;  
    }  
  
    // Does 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

- Calcular o índice da tabela
- Procurar na lista usando a chave
- Existe ?
- Aceder ao HT bin
- Atualizar o valor

```
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;  
}
```

# Hash Table – Apagar é fácil !

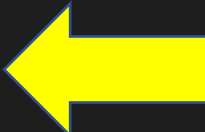


[Sedgewick & Wayne]

# Apagar

- Calcular o índice da tabela
- Procurar na lista usando a chave
- Existe ?
- Libertar a memória atribuída à chave e ao valor
- E ao HT bin
- Remover o nó da lista

```
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;  
}
```



# Separate Chaining *vs* Open Addressing

# Separate Chaining

- Tamanho da tabela :  $M = 17$
- Número de items :  $N = 12$
- 4 colisões !
- 9 listas vazias
- 8 listas com elementos
- Lista mais longa tem 3 nós

size = 17   Active = 12	
0 -	Hash = 68, (December, 12th month)
1 -	
2 -	Hash = 70, (February, 2nd month of the year)
3 -	
4 -	
5 -	
6 -	
7 -	Hash = 74, (January, 1st month of the year)
8 -	Hash = 74, (July, 7th month)
9 -	Hash = 74, (June, 6th month)
10 -	
11 -	
12 -	Hash = 77, (March, 3rd month)
13 -	Hash = 77, (May, 5th month)
14 -	
15 -	Hash = 78, (November, 11th month)
16 -	Hash = 79, (October, 10th month)
17 -	
18 -	
19 -	
20 -	
21 -	
22 -	
23 -	
24 -	
25 -	
26 -	
27 -	
28 -	
29 -	
30 -	
31 -	
32 -	
33 -	
34 -	
35 -	
36 -	
37 -	
38 -	
39 -	
40 -	
41 -	
42 -	
43 -	
44 -	
45 -	
46 -	
47 -	
48 -	
49 -	
50 -	
51 -	
52 -	
53 -	
54 -	
55 -	
56 -	
57 -	
58 -	
59 -	
60 -	
61 -	
62 -	
63 -	
64 -	
65 -	
66 -	
67 -	
68 -	
69 -	
70 -	
71 -	
72 -	
73 -	
74 -	
75 -	
76 -	
77 -	
78 -	
79 -	
80 -	
81 -	
82 -	
83 -	
84 -	
85 -	
86 -	
87 -	
88 -	
89 -	
90 -	
91 -	
92 -	
93 -	
94 -	
95 -	
96 -	
97 -	
98 -	
99 -	
100 -	
101 -	
102 -	
103 -	
104 -	
105 -	
106 -	
107 -	
108 -	
109 -	
110 -	
111 -	
112 -	
113 -	
114 -	
115 -	
116 -	
117 -	
118 -	
119 -	
120 -	
121 -	
122 -	
123 -	
124 -	
125 -	
126 -	
127 -	
128 -	
129 -	
130 -	
131 -	
132 -	
133 -	
134 -	
135 -	
136 -	
137 -	
138 -	
139 -	
140 -	
141 -	
142 -	
143 -	
144 -	
145 -	
146 -	
147 -	
148 -	
149 -	
150 -	
151 -	
152 -	
153 -	
154 -	
155 -	
156 -	
157 -	
158 -	
159 -	
160 -	
161 -	
162 -	
163 -	
164 -	
165 -	
166 -	
167 -	
168 -	
169 -	
170 -	
171 -	
172 -	
173 -	
174 -	
175 -	
176 -	
177 -	
178 -	
179 -	
180 -	
181 -	
182 -	
183 -	
184 -	
185 -	
186 -	
187 -	
188 -	
189 -	
190 -	
191 -	
192 -	
193 -	
194 -	
195 -	
196 -	
197 -	
198 -	
199 -	
200 -	
201 -	
202 -	
203 -	
204 -	
205 -	
206 -	
207 -	
208 -	
209 -	
210 -	
211 -	
212 -	
213 -	
214 -	
215 -	
216 -	
217 -	
218 -	
219 -	
220 -	
221 -	
222 -	
223 -	
224 -	
225 -	
226 -	
227 -	
228 -	
229 -	
230 -	
231 -	
232 -	
233 -	
234 -	
235 -	
236 -	
237 -	
238 -	
239 -	
240 -	
241 -	
242 -	
243 -	
244 -	
245 -	
246 -	
247 -	
248 -	
249 -	
250 -	
251 -	
252 -	
253 -	
254 -	
255 -	
256 -	
257 -	
258 -	
259 -	
260 -	
261 -	
262 -	
263 -	
264 -	
265 -	
266 -	
267 -	
268 -	
269 -	
270 -	
271 -	
272 -	
273 -	
274 -	
275 -	
276 -	
277 -	
278 -	
279 -	
280 -	
281 -	
282 -	
283 -	
284 -	
285 -	
286 -	
287 -	
288 -	
289 -	
290 -	
291 -	
292 -	
293 -	
294 -	
295 -	
296 -	
297 -	
298 -	
299 -	
300 -	
301 -	
302 -	
303 -	
304 -	
305 -	
306 -	
307 -	
308 -	
309 -	
310 -	
311 -	
312 -	
313 -	
314 -	
315 -	
316 -	
317 -	
318 -	
319 -	
320 -	
321 -	
322 -	
323 -	
324 -	
325 -	
326 -	
327 -	
328 -	
329 -	
330 -	
331 -	
332 -	
333 -	
334 -	
335 -	
336 -	
337 -	
338 -	
339 -	
340 -	
341 -	
342 -	
343 -	
344 -	
345 -	
346 -	
347 -	
348 -	
349 -	
350 -	
351 -	
352 -	
353 -	
354 -	
355 -	
356 -	
357 -	
358 -	
359 -	
360 -	
361 -	
362 -	
363 -	
364 -	
365 -	
366 -	
367 -	
368 -	
369 -	
370 -	
371 -	
372 -	
373 -	
374 -	
375 -	
376 -	
377 -	
378 -	
379 -	
380 -	
381 -	
382 -	
383 -	
384 -	
385 -	
386 -	
387 -	
388 -	
389 -	
390 -	
391 -	
392 -	
393 -	
394 -	
395 -	
396 -	
397 -	
398 -	
399 -	
400 -	
401 -	
402 -	
403 -	
404 -	
405 -	
406 -	
407 -	
408 -	
409 -	
410 -	
411 -	
412 -	
413 -	
414 -	
415 -	
416 -	
417 -	
418 -	
419 -	
420 -	
421 -	
422 -	
423 -	
424 -	
425 -	
426 -	
427 -	
428 -	
429 -	
430 -	
431 -	
432 -	
433 -	
434 -	
435 -	
436 -	
437 -	
438 -	
439 -	
440 -	
441 -	
442 -	
443 -	
444 -	
445 -	
446 -	
447 -	
448 -	
449 -	
450 -	
451 -	
452 -	
453 -	
454 -	
455 -	
456 -	
457 -	
458 -	
459 -	
460 -	
461 -	
462 -	
463 -	
464 -	
465 -	
466 -	
467 -	
468 -	
469 -	
470 -	
471 -	
472 -	
473 -	
474 -	
475 -	
476 -	
477 -	
478 -	
479 -	
480 -	
481 -	
482 -	
483 -	
484 -	
485 -	
486 -	
487 -	
488 -	
489 -	
490 -	
491 -	
492 -	
493 -	
494 -	
495 -	
496 -	
497 -	
498 -	
499 -	
500 -	
501 -	
502 -	
503 -	
504 -	
505 -	
506 -	
507 -	
508 -	
509 -	
510 -	
511 -	
512 -	
513 -	
514 -	
515 -	
516 -	
517 -	
518 -	
519 -	
520 -	
521 -	
522 -	
523 -	
524 -	
525 -	
526 -	
527 -	
528 -	
529 -	
530 -	
531 -	
532 -	
533 -	
534 -	
535 -	
536 -	
537 -	
538 -	
539 -	
540 -	
541 -	
542 -	
543 -	
544 -	
545 -	
546 -	
547 -	
548 -	
549 -	
550 -	
551 -	
552 -	
553 -	
554 -	
555 -	
556 -	
557 -	
558 -	
559 -	
560 -	
561 -	
562 -	
563 -	
564 -	
565 -	
566 -	
567 -	
568 -	
569 -	
570 -	
571 -	
572 -	
573 -	
574 -	
575 -	
576 -	
577 -	
578 -	
579 -	
580 -	
581 -	
582 -	
583 -	
584 -	
585 -	
586 -	
587 -	
588 -	
589 -	
590 -	
591 -	
592 -	
593 -	
594 -	
595 -	
596 -	
597 -	
598 -	
599 -	
600 -	
601 -	
602 -	
603 -	
604 -	
605 -	
606 -	
607 -	
608 -	
609 -	
610 -	
611 -	
612 -	
613 -	
614 -	
615 -	
616 -	
617 -	
618 -	
619 -	
620 -	
621 -	
622 -	
623 -	
624 -	
625 -	
626 -	
627 -	
628 -	
629 -	
630 -	
631 -	
632 -	
633 -	
634 -	
635 -	
636 -	
637 -	
638 -	
639 -	
640 -	
641 -	
642 -	
643 -	
644 -	
645 -	
646 -	
647 -	
648 -	
649 -	
650 -	
651 -	
652 -	
653 -	
654 -	
655 -	
656 -	
657 -	
658 -	
659 -	
660 -	
661 -	
662 -	
663 -	
664 -	
665 -	
666 -	
667 -	
668 -	
669 -	
670 -	
671 -	
672 -	
673 -	
674 -	
675 -	
676 -	
677 -	
678 -	
679 -	
680 -	
681 -	
682 -	
683 -	
684 -	
685 -	
686 -	
687 -	
688 -	
689 -	
690 -	
691 -	
692 -	
693 -	
694 -	
695 -	
696 -	
697 -	
698 -	
699 -	
700 -	
701 -	
702 -	
703 -	
704 -	
705 -	
706 -	
707 -	
708 -	
709 -	
710 -	
711 -	
712 -	
713 -	



# Open Addressing + Linear Probing – Clusters

```
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)
```

# Hash Tables

## – Eficiência Computacional

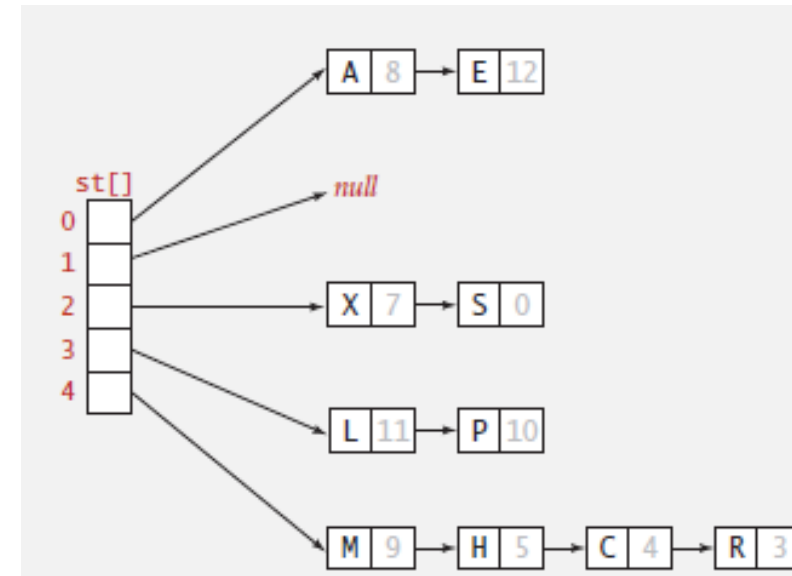
# Hash Table – Eficiência Computacional

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
- 
- **Open Addressing + 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()`



# Exercícios / Tarefas

# Exercício 1 – Escolha múltipla

Numa tabela de dispersão (“*Hash Table*”) implementada usando endereçamento direto (“*Separate Chaining*”),

- a) se ocorrer uma colisão, durante a inserção de um novo elemento, i.e., par (chave, valor), este é inserido na lista de elementos que partilham o mesmo endereço (“*hash-address*”).
- b) após a inserção de um grande número de pares (chave, valor), o fator de carga (“*Load Factor*”) poderá tornar-se **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, String)

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

# Tarefa 3 – HashTable – Nova Versão

- A implementação do TAD Hash Table usa o TAD Sorted List
- Essa decisão tornou rápido o desenvolvimento da solução
- MAS, torna-a demasiado “pesada”, sem necessidade, pois as listas deverão ser curtas
- Desenvolva uma solução “mais leve”, em que as funções do TAD Hash Table operam diretamente sobre os nós das listas, que são apenas constituídos pelos campos key, value, e next