Dicionários / Tabelas de Dispersão II

22/11/2023

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

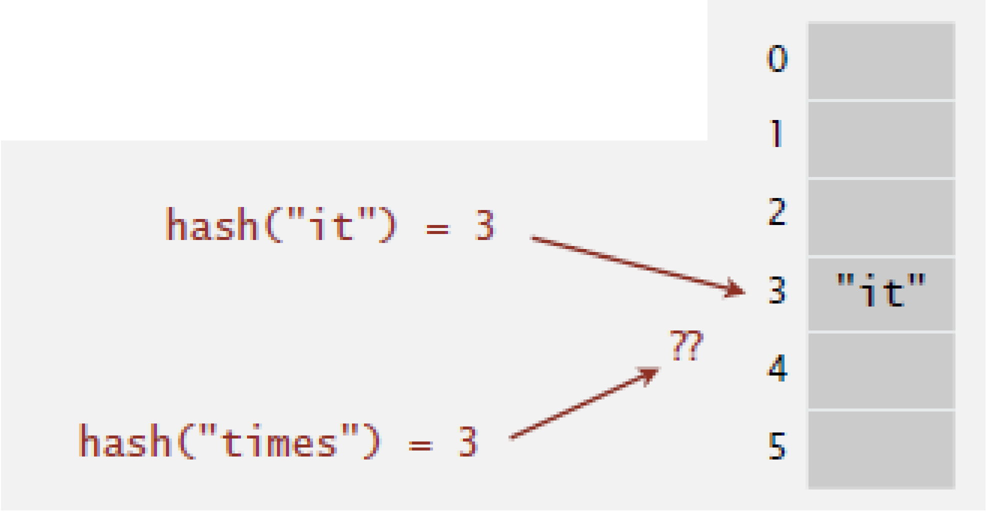
Recapitulação

Tabelas de Dispersão – Hash Tables

- Estrutura de dados para armazenar pares (chave, valor)
- Sem chaves duplicadas
- Sem limite de tamanho
- Sem uma ordem implícita
- MAS, com acesso rápido !!

Hash Tables – Open Addressing

- 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 índice da tabela

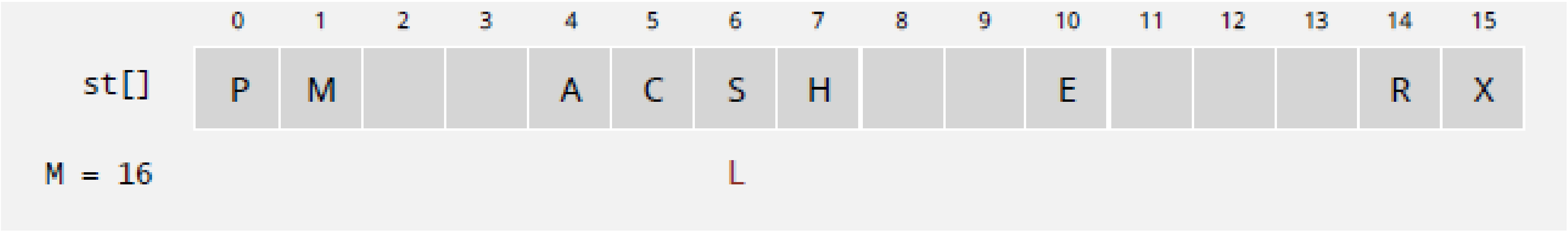


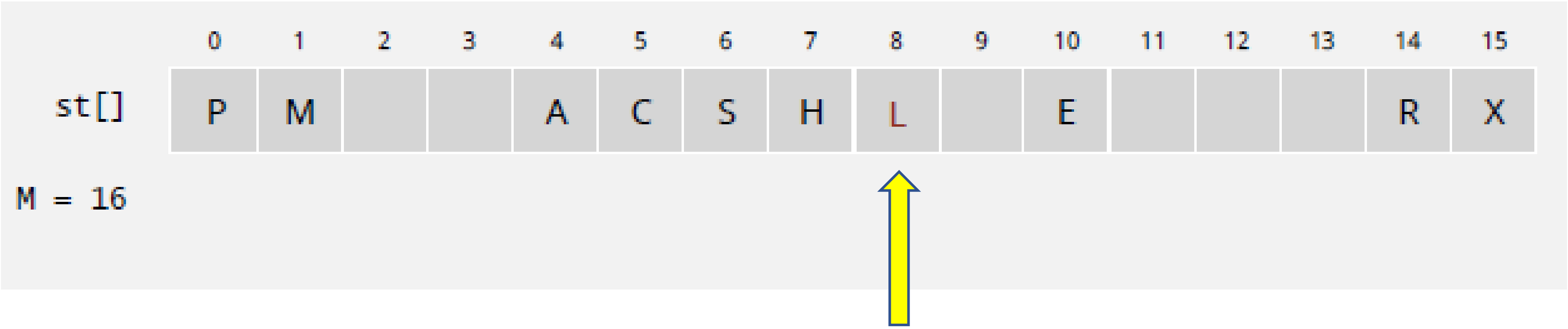
[Sedgewick & Wayne]

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 !!
- ...





[Sedgewick & Wayne]

Resolução de colisões – Linear Probing

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

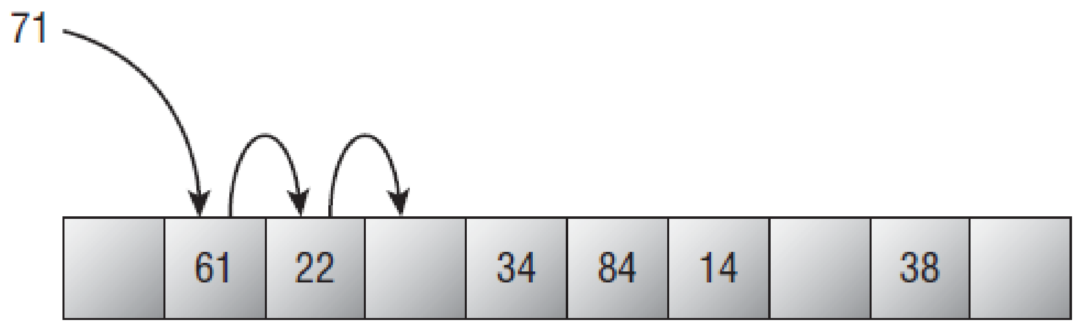


Figure 8-2: In linear probing, the algorithm adds a constant amount to locations to produce a probe sequence.

[Stephens]

Resolução de colisões – Quadratic Probing

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

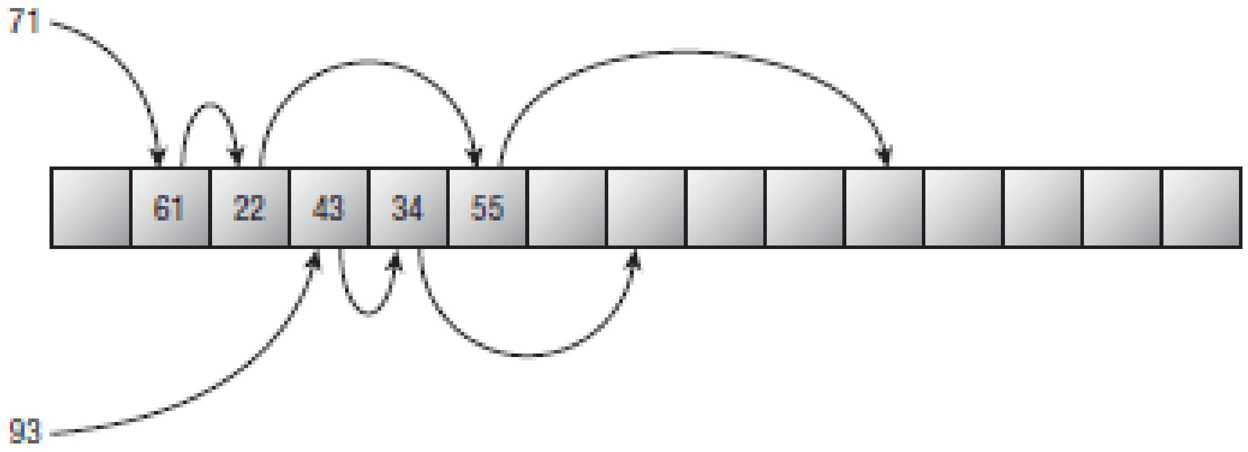


Figure 8-4: Quadratic probing reduces primary clustering.

[Stephens]

Análise – Linear Probing – Knuth, 1963

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

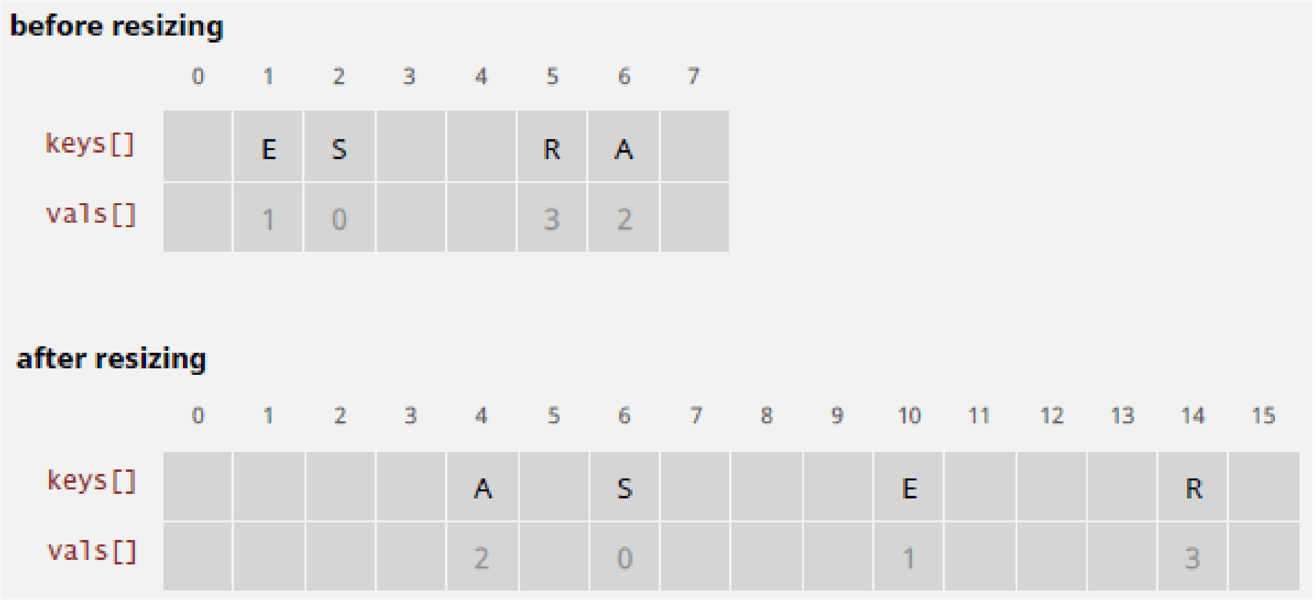
  $1/2 \times (1 + 1/(1 - \lambda))$ -> 1.5, se λ = 50% -> 3, se λ = 80%

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

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

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]

Exemplo
– Hash Table (String, String)

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)
```

TAD Hash Table

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

void HashTableDestroy(HashTable** p);
```

```c
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 Header + Hash Table Bin

```
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;
};
```

Procura de uma chave

```c
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;
  }
}
```

HashTableGet

```c
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;
}
```

Tarefas

- Analisar as funções desenvolvidas
- E o simples programa de teste

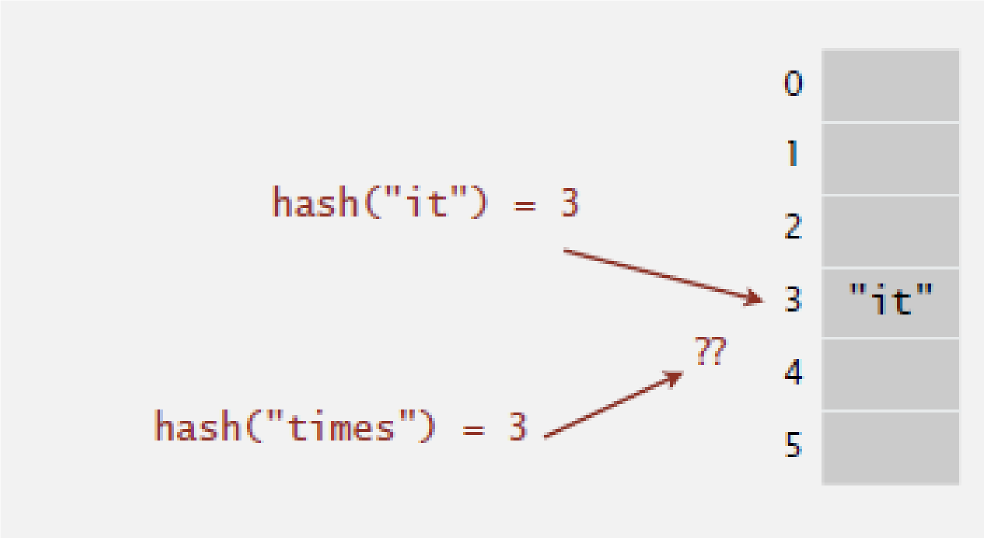
Hash Tables

– Separate Chaining

Colisões – Como proceder ?

- Duas chaves distintas são mapeadas no mesmo índice da tabela !!

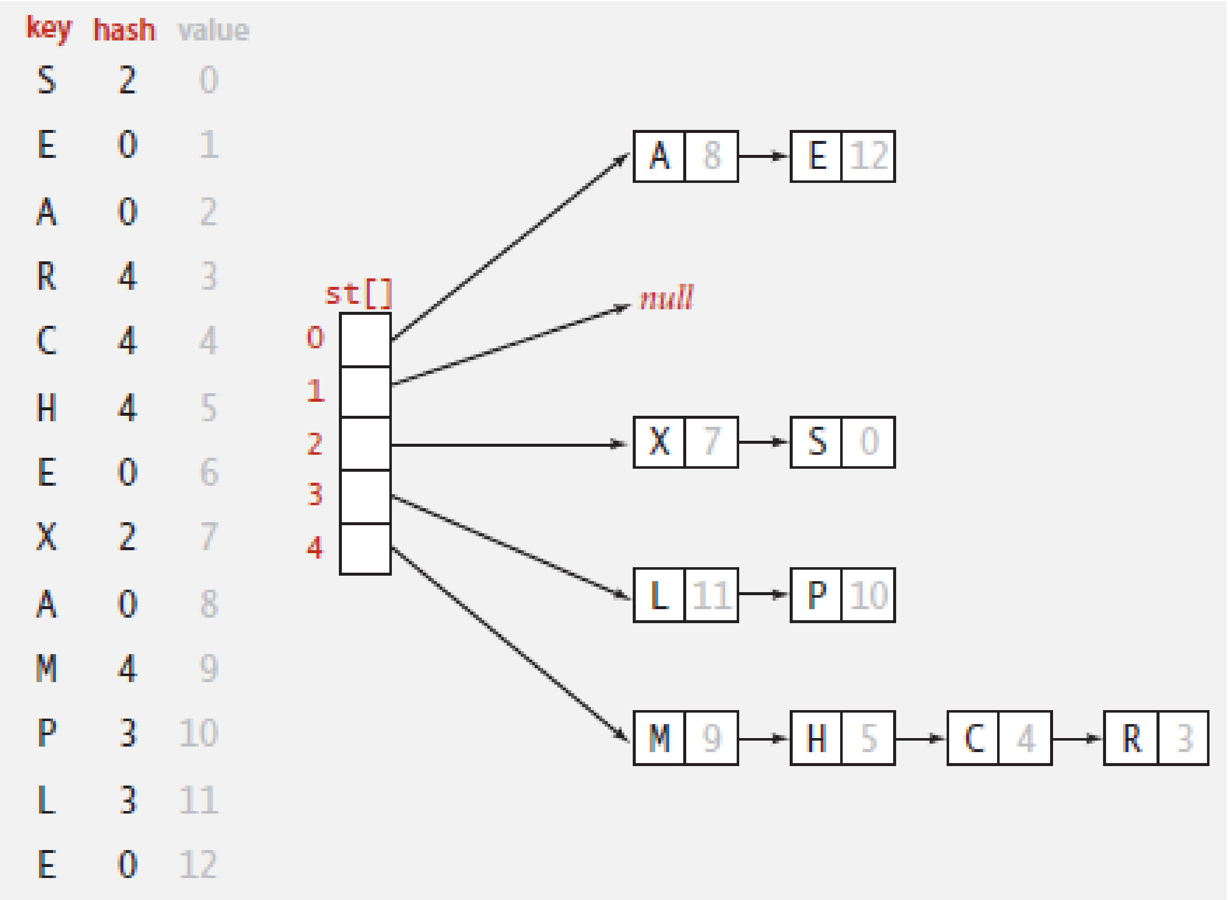
- Como gerir de modo eficiente ?
- Sem usar "demasiada" memória !!
- Alternativa ao Open Addressing ?



[Sedgewick & Wayne]

Separate Chaining (IBM, 1953)

- Array de M 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]

Hash Table Header + Hash Table Bin

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

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

HashTableCreate

```
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);
}
```

HashTableDestroy

```
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);
```

Procurar

```c
// 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;
}
```

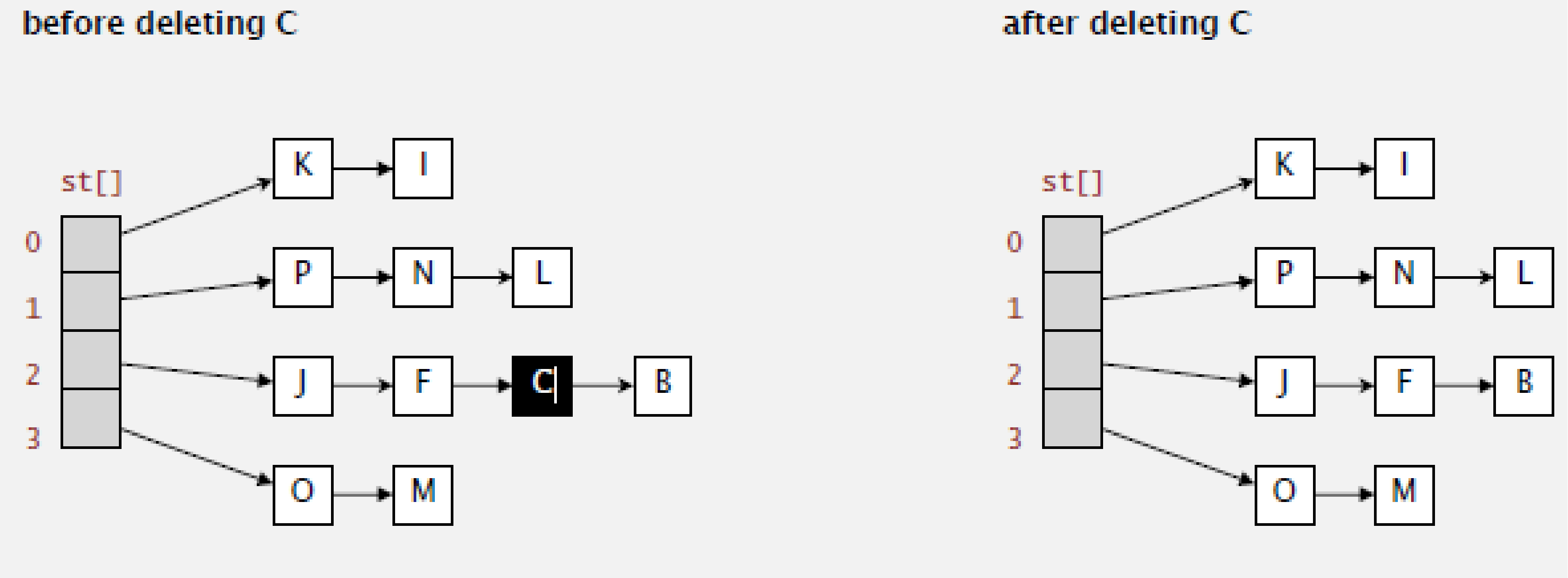
Inserir

```c
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;
}
```

Apagar é fácil !



[Sedgewick & Wayne]

Apagar

```c
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;
}
```

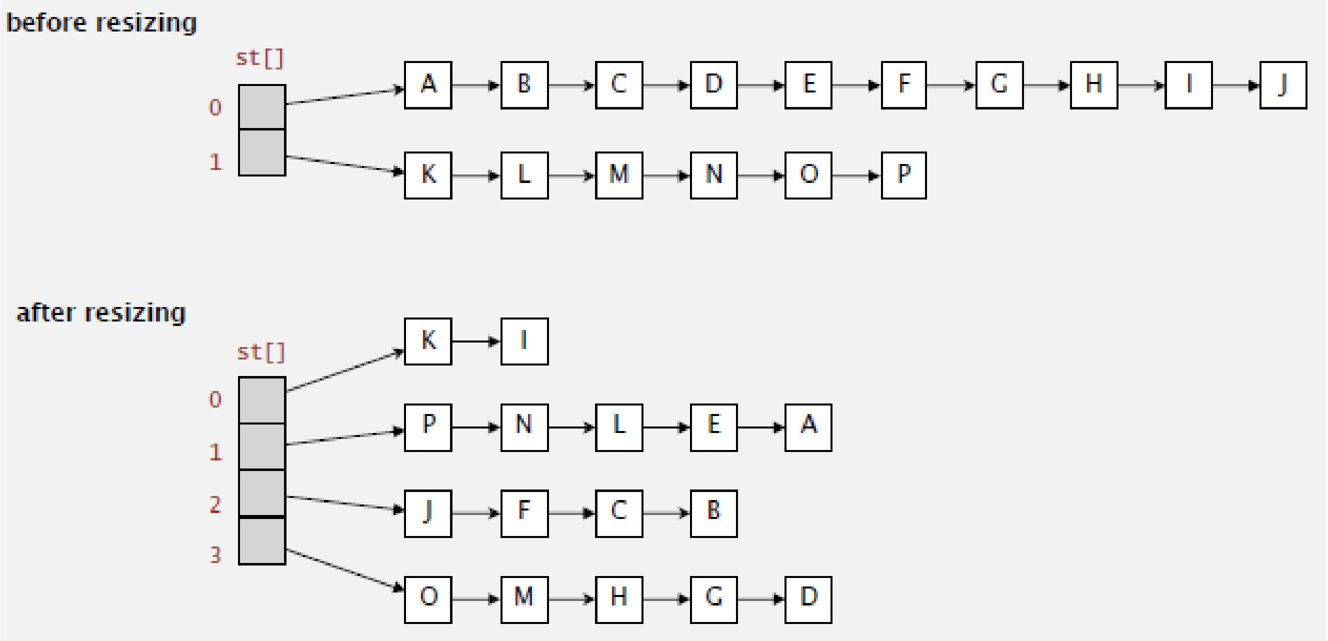
Eficiência Computacional

- Em média, N/M elementos em cada lista – Load Factor
- Procurar / inserir -> nº de comparações é proporcional a N/M
  - M vezes mais rápido que na procura sequencial

- M muito grande -> demasiadas listas vazias
- M demasiado pequeno -> listas muito longas
- Escolha habitual : $M \approx N/4$ **-> 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]

Tarefa

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

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)
```

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 *		`equals()` `hashCode()`
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

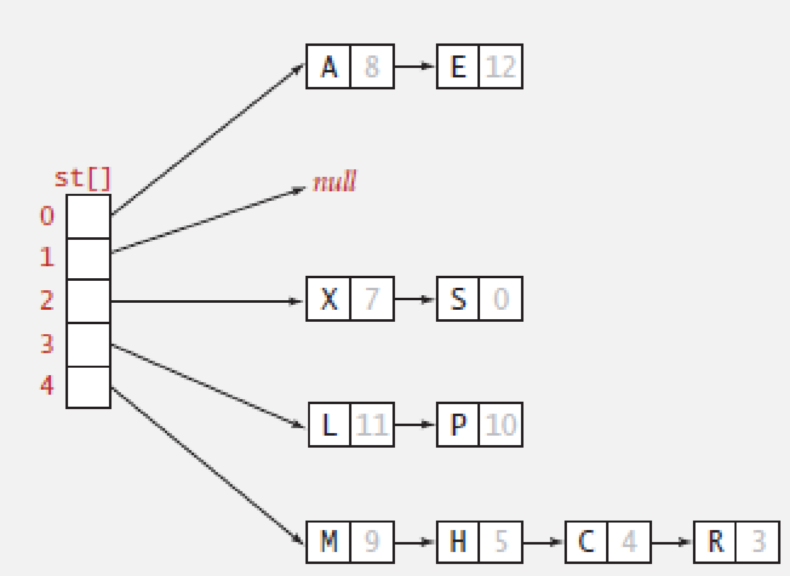
* under uniform hashing assumption

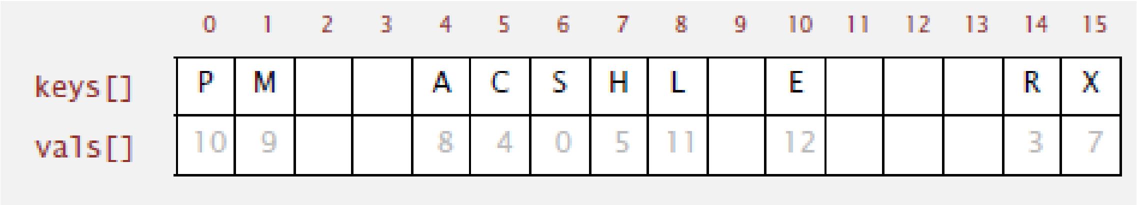
[Sedgewick & Wayne]

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





[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()