

Dicionários / Tabelas de Símbolos I

Joaquim Madeira

14/05/2020

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”**, que permitem trabalho autónomo de desenvolvimento e teste

Sumário

- Recap
- Motivação
- Digital Search Trees – Breve referência
- Prefix Trees – Breve referência
- Hash Tables – Tabelas de Dispersão
- Funções de Hashing
- Representação usando um array – Open Addressing
- O TAD Hash Table (String, String)

Let's
RECAP


Recapitulação

TAD **Árvore Binária de Procura**

- Conjunto de **elementos** do **mesmo tipo**
- Armazenados **em-ordem**
- Procura / inserção / remoção / substituição
- Pertença
- **search() / insert() / remove() / replace()**
- **size() / isEmpty() / contains()**
- **create() / destroy()**

Lista ligada / Array ordenado / ABP

search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N



h = height of BST
(proportional to $\log N$
if keys inserted in random order)

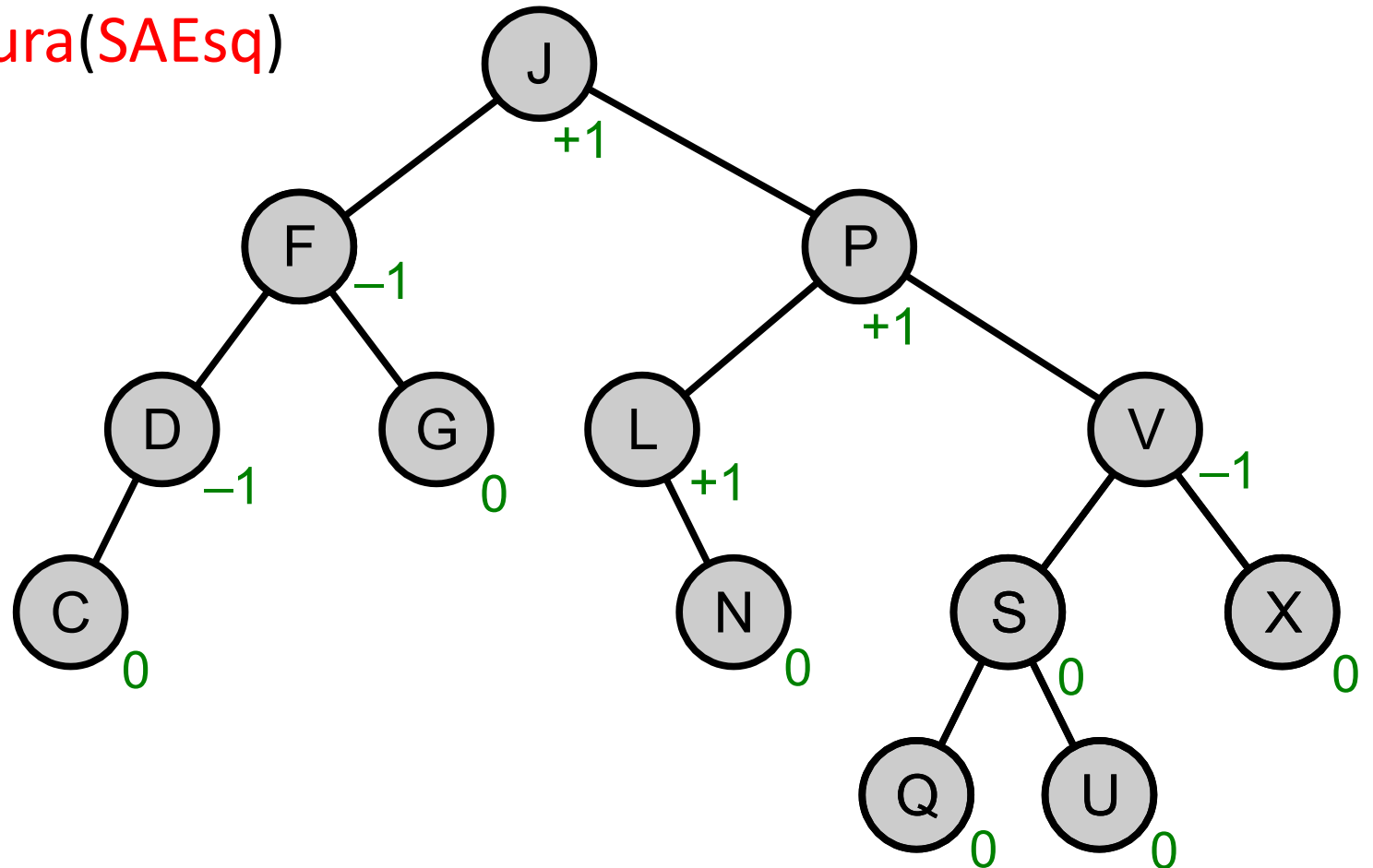
[Sedgewick & Wayne]

Árvores equilibradas em altura

- **Esforço** computacional das operações habituais sobre ABPs depende do **comprimento do caminho** a partir da raiz da árvore
- **Evitar** que uma ABP tenha uma **altura “exagerada”**, para assegurar um bom “comportamento” – **Altura $\in O(\log n)$**
- **O que fazer?**
- Assegurar que, para cada nó, a **altura** das suas duas **subárvores** não é “muito diferente”

Fator de equilíbrio de um nó

- $F = \text{altura}(\text{SADir}) - \text{altura}(\text{SAEsq})$



[Wikipedia]

Quando fazer? / Como fazer?

- Assegurar o critério de equilíbrio sempre que se adiciona ou remove um nó : $F = -1, 0, +1$
- Reposicionar nós / subárvores quando falha : $F = -2, +2$
- MAS, manter o critério de ordem da ABP !!
- 4 tipos de operações de rotação
- Basta fazer a verificação / rotações ao longo do caminho entre a raiz e o nó – **traceback**

Árvore AVL – Animação simples



[Wikipedia]

1ª experiência computacional

- **Criar** uma árvore vazia
- **Inserir** ordenadamente sucessivos números pares: 2, 4, 6, ...
- **Procurar** cada um desses números pares na árvore
- **Contar** o número de **comparações** efetuadas em cada nó
 - 1 ou 2 comparações por nó visitado

Procurar os sucessivos números pares

nós	Altura ABP	Nº médio comps	Altura AVL	Nº médio comps
5000	4999	5001	12	17,69
10000	9999	10001	13	19,19
20000	19999	20001	14	20,69
40000	39999	40001	15	22,19

2ª experiência computacional

- **Criar** uma árvore vazia
- **Inserir** uma sequência de números aleatórios
- **Procurar** cada um desses números na árvore
- **Contar** o número de **comparações** efetuadas em cada nó
 - 1 ou 2 comparações por nó visitado

Procurar os sucessivos números aleatórios

nós	Altura ABP	Nº médio comps	Altura AVL	Nº médio comps
2500	27	19,64	12	16,18
5000	25	22,10	14	17,66
10000	30	25,72	15	18,85
20000	28	25,83	16	19,83
40000	32	26,73	16	20,91

Dicionários

– Motivação

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

Aplicações

<i>application</i>	<i>key</i>	<i>value</i>
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]

Operações básicas

- Criar uma tabela vazia
- Registrar 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 à tabela – **contains**
- Limpar / destruir
- **EXTRA** : **iterar** sobre todas as chaves (**em ordem**)

Java

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

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

Algumas estruturas de dados – Fazer melhor?

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>

[Sedgwick & Wayne]

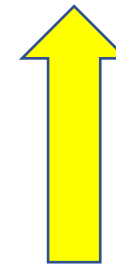
Digital Search Trees

Ideias principais


- Árvore binária
- Armazenar um par (**chave**, **valor**) em cada nó, sem repetições
- Cada elemento é inserido como folha da árvore
- Diferença para as ABPs :
- Se a **chave procurada** for a **chave registada** num nó -> procura bem sucedida
- Caso contrário, examinar um **bit da chave** e procurar recursivamente ou na **SAEsq** ou na **SADir**

Ideias principais


- O **caminho**, a partir da raiz, para um dado nó é definido por **sucessivos bits** da sua **chave** -> procura + inserção
- **bit = 0** -> avançar para a **SAEsquerda**
- **bit = 1** -> avançar para a **SADireita**
- A **organização** de uma árvore depende da **sequência de inserção** dos seus elementos



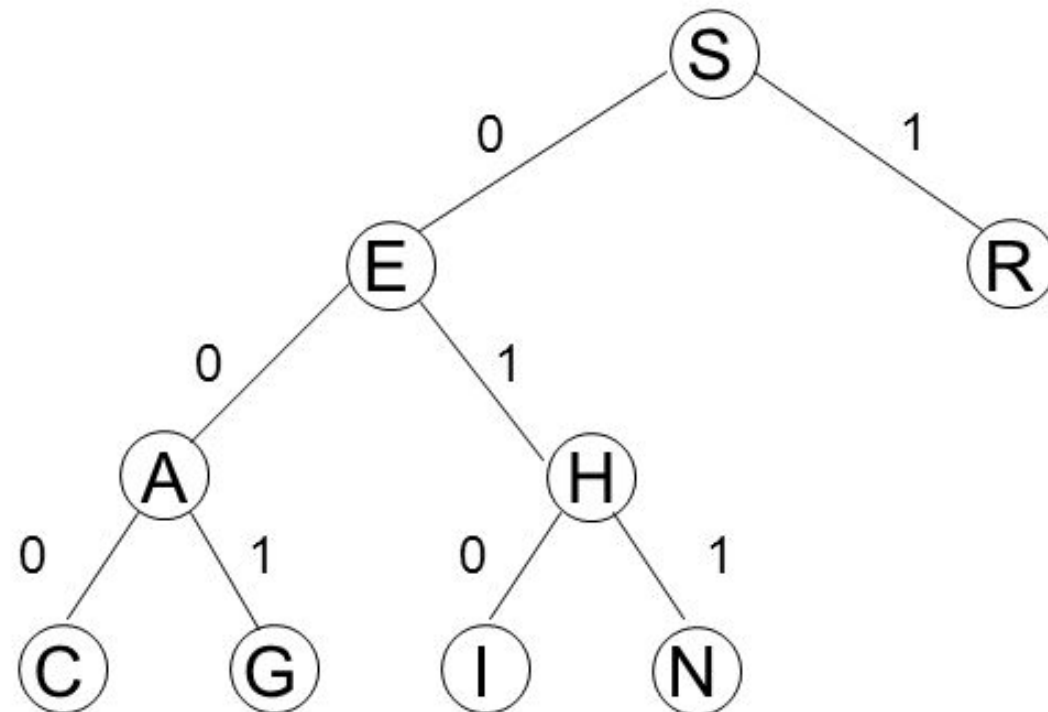
Exemplo – Fazer a sequência de inserção



S	1	0	0	1	1
E	0	0	1	0	1
A	0	0	0	0	1
R	1	0	0	1	0
C	0	0	0	1	1
H	0	1	0	0	0
I	0	1	0	0	1
N	0	1	1	1	0
G	0	0	1	1	1



Digital Search Tree



Procura recursiva

```
Item* search(Node* p, Key key, int i) {  
    if (p == NULL) return NULL;  
    if (equals(v, p->item->key)) return p->item;  
    if (bit(key, i) == 0)  
        return search(p->left, key, i+1);  
    return search(p->right, v, i+1);  
}
```

Procura recursiva

- O nº de nós visitados é, quando muito, igual ao nº de bits mais significativos necessários para distinguir a chave de procura das outras chaves
- MAS, a visita de cada nó implica a comparação usando toda a chave
- Pior Caso : $O(n)$, n é o nº de bits da chave
- Caso Médio : $O(\log n)$, para chaves aleatórias

Tries – Prefix Trees

Ideias principais


- Árvore binária
- Armazenar um par (**chave**, **valor**) em cada **FOLHA**, sem repetições
- Diferenças para as ABPs :
- Os **nós não-terminais** não contêm itens !!
- Em cada nó não-terminal, examinar um **bit da chave** e procurar recursivamente ou na **SAEsq** ou na **SADir**
- Se a **chave procurada** for a **chave registada** numa folha -> procura bem sucedida

Ideias principais

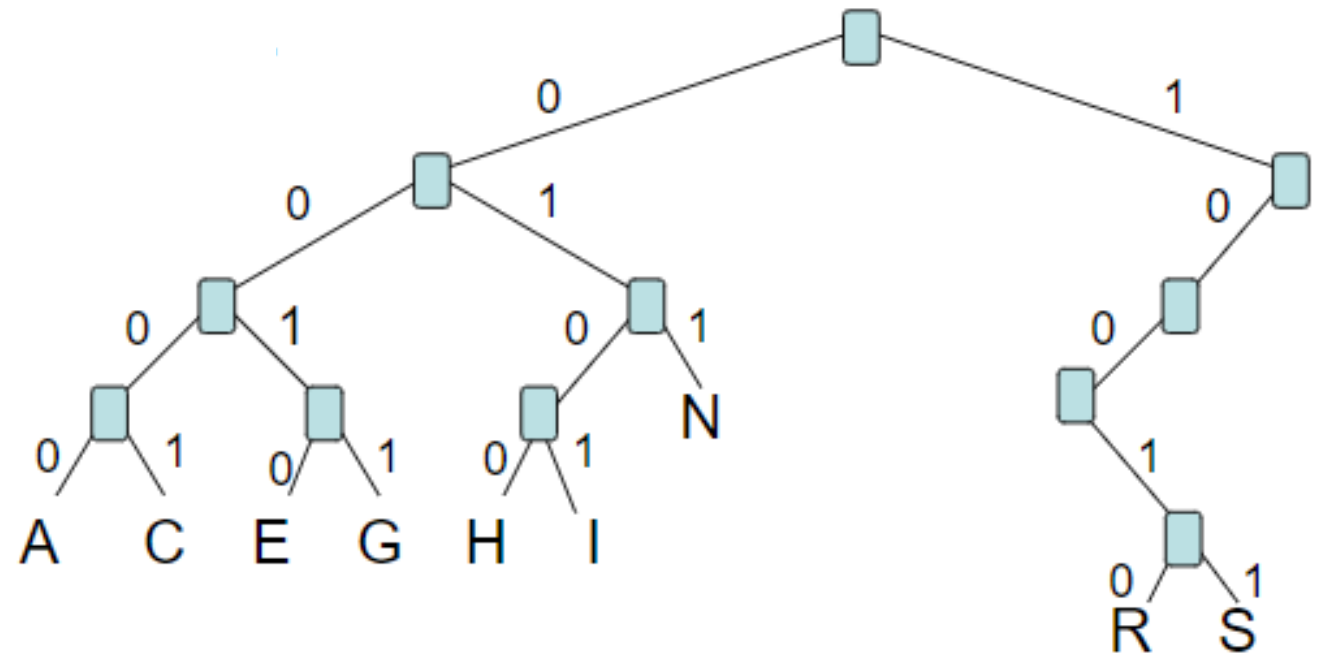
- Chaves de tamanho fixo; nenhuma chave é prefixo de outra
- O **caminho**, a partir da raiz, para cada **folha** é definido por **sucessivos bits** da sua **chave** -> procura + inserção
- **bit = 0** -> avançar para a **SAEsquerda**
- **bit = 1** -> avançar para a **SADireita**
- A **organização** de uma árvore **NÃO** depende da **sequência de inserção** dos seus elementos



Exemplo – Fazer a sequência de inserção



S	1	0	0	1	1
E	0	0	1	0	1
A	0	0	0	0	1
R	1	0	0	1	0
C	0	0	0	1	1
H	0	1	0	0	0
I	0	1	0	0	1
N	0	1	1	1	0
G	0	0	1	1	1



Procura recursiva

```
Item* search(Node* p, Key key, int i) {  
    if (p == NULL) return NULL;  
    if (p->left == NULL && p->right == NULL )  
        if (equals(v, p->item->key)) return p->item;  
        else return NULL;  
    if (bit(key, i) == 0)  
        return search(p->left, key, i+1);  
    return search(p->right, v, i+1);  
}
```

Procura recursiva

- O nº de nós visitados é, quando muito, igual ao nº de bits mais significativos necessários para distinguir a chave de procura das outras chaves
- MAS, a visita de uma folha implica a comparação usando toda a chave
- Pior Caso : $O(n)$, n é o nº de bits da chave
- Caso Médio : $O(\log n)$, para chaves aleatórias

Hash Tables

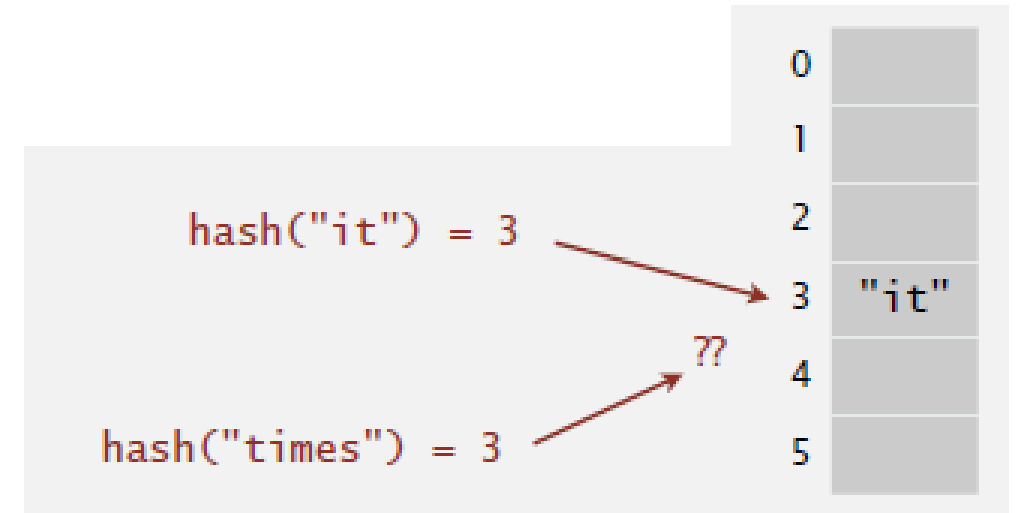
– Tabelas de Dispersão

Tabelas de Dispersão

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

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]

Tabelas de Dispersão - Problemas

- Como calcular a **função de hashing** ?
 - Rapidez + simplicidade
- Como verificar se duas **chaves 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

- 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 !!

Funções de Hashing

- Requisito : se $x = y$, então $\text{hash}(x) = \text{hash}(y)$
- Desejável : se $x \neq y$, então $\text{hash}(x) \neq \text{hash}(y)$
- 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)); }`

Funções de Hashing

- `int hash(char* s) {`
 `int hash = 0;`
 `for (int i = 0; i < strlen(s); i++)`
 `hash = s[i] + (31 * hash);`
 `return hash;`
 `}`
- `hash("call") = ?`

Funções de Hashing

- ```
int hash(char* s) {
 int hash = 0;
 for (int i = 0; i < strlen(s); i++)
 hash = s[i] + (31 * hash);
 return hash;
}
```
- $\text{hash}(\text{"call"}) = 3045982$   
 $= 108 + 31 \times (108 + 31 \times (97 + 31 \times (99)))$  M. 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 **nº reduzido de colisões**

# Conversão para índices da tabela

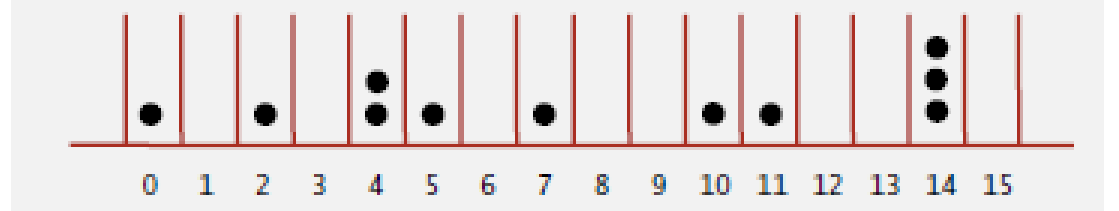
- Índices da tabela entre 0 e  $M - 1$
- $M$  é um número primo ou uma potência de 2
- Como fazer ?

$$\text{abs}(\text{hash}(x)) \% M$$

# Equiprobabilidade

- 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 ?



- Lembram-se do **Paradoxo do Aniversário** ?

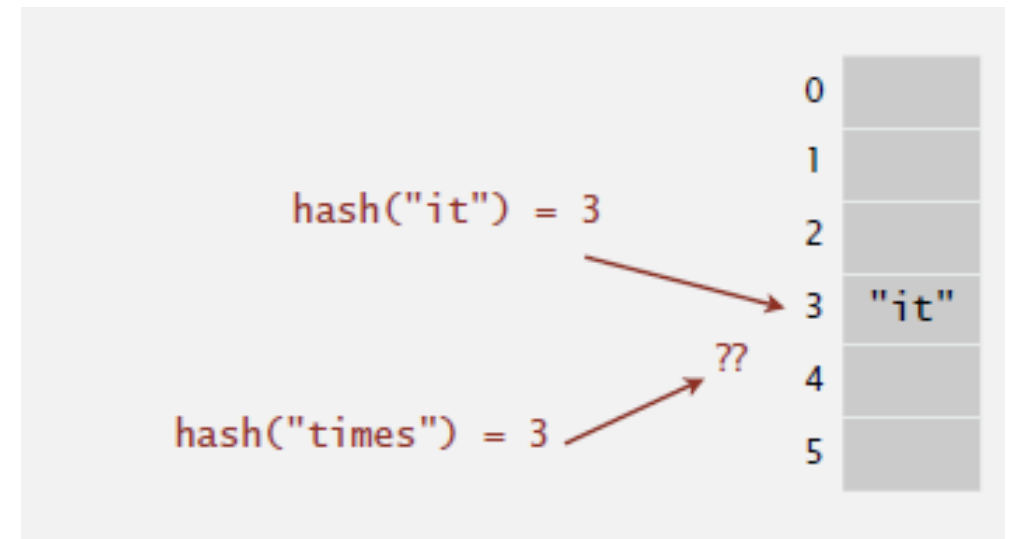
[Sedgewick & Wayne]

# Hash Tables

- 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 !!

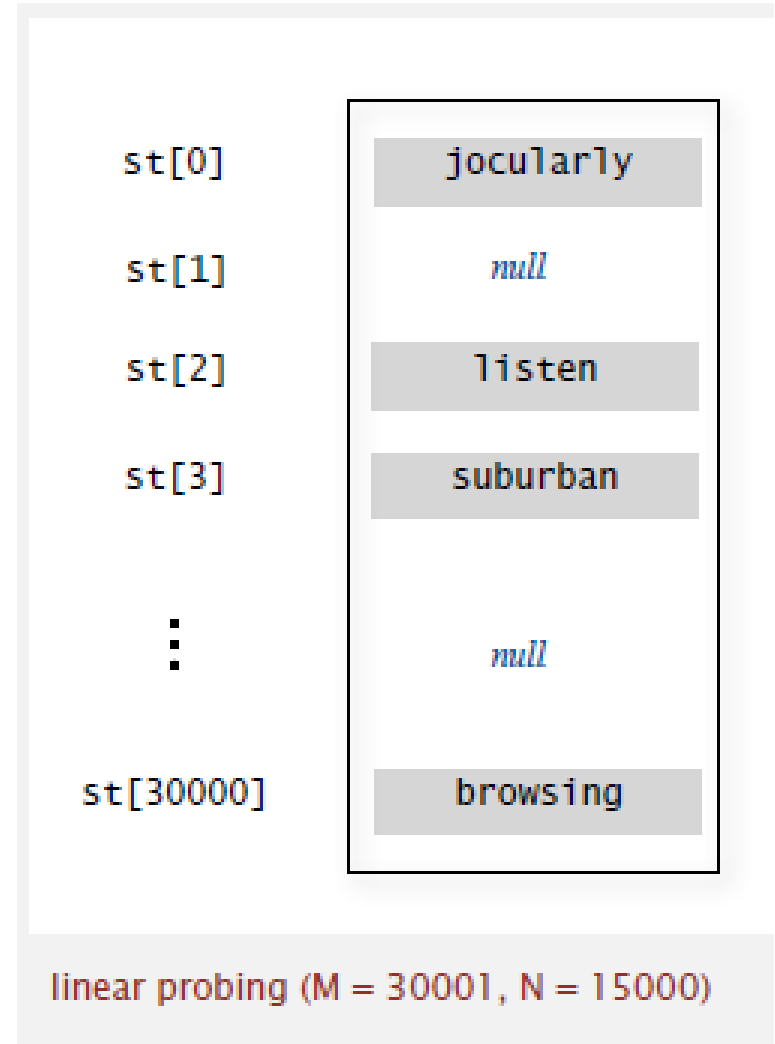


[Sedgewick & Wayne]

# 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** tem de ser **maior** do que o número de itens !!
- Quantas vezes maior ??




# 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 !!**
- ...

|        |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|--------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|        | 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]

# 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**

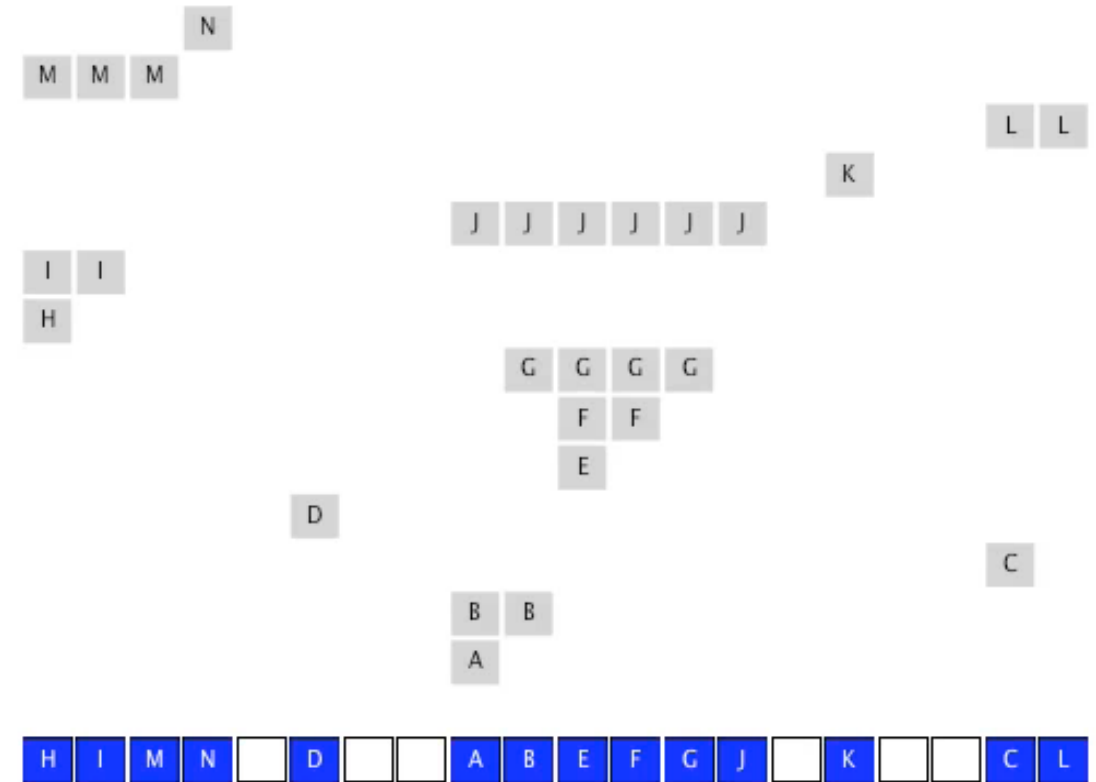
|        |                                            |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|--------|--------------------------------------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|        | 0                                          | 1 | 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 | H                                          |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|        | search hit<br>(return corresponding value) |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |

[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**



[Sedgewick & Wayne]

# 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  $\lambda = 4/5$



# Análise – Linear Probing

- M muito **grande** -> demasiados espaços vagos !!
- M “**pequeno**” -> tempo de procura aumenta muito !!
- **Limiar** habitual para o fator de carga : **50%**
- N<sup>o</sup> médio de tentativas para encontrar um item : 1,5 **hit**
- N<sup>o</sup> médio de tentativas para inserir um item : 2,5 **miss**
- Como controlar ? **RESIZING + REHASHING !!**

# Inserir na tabela – Quadratic Probing

- Guardar na posição  $i$ , se estiver disponível
- Caso contrário, tentar  $(i + 1) \% M$ ,  $(i + 4) \% M$ ,  $(i + 9) \% M$ , etc.

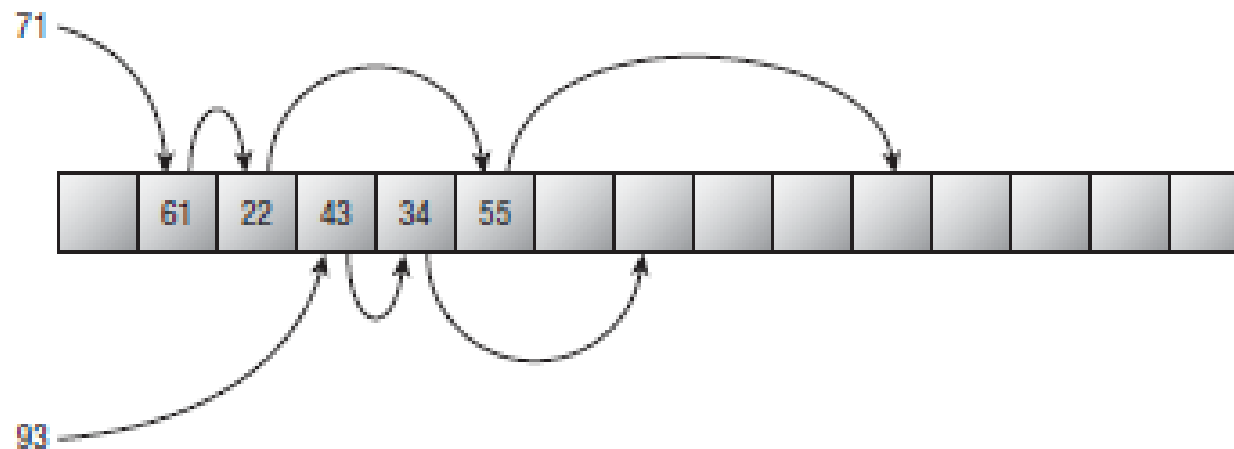


Figure 8-4: Quadratic probing reduces primary clustering.

[Stephens]

# 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**

# Eficiência

- 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



# Eficiência

| implementation                               | worst-case cost |        |        | average case cost<br>(after $N$ random inserts) |                 | key<br>interface                                 |
|----------------------------------------------|-----------------|--------|--------|-------------------------------------------------|-----------------|--------------------------------------------------|
|                                              | search          | insert | delete | search hit                                      | insert          |                                                  |
| <b>sequential search</b><br>(unordered list) | $N$             | $N$    | $N$    | $\frac{1}{2} N$                                 | $N$             | <code>equals()</code>                            |
| <b>binary search</b><br>(ordered array)      | $\lg N$         | $N$    | $N$    | $\lg N$                                         | $\frac{1}{2} N$ | <code>compareTo()</code>                         |
| <b>BST</b>                                   | $N$             | $N$    | $N$    | $1.4 \lg N$                                     | $1.4 \lg N$     | <code>compareTo()</code>                         |
| <b>linear probing</b>                        | $N$             | $N$    | $N$    | $3-5^*$                                         | $3-5^*$         | <code>equals()</code><br><code>hashCode()</code> |
| * under the uniform hashing assumption       |                 |        |        |                                                 |                 |                                                  |

[Sedgewick & Wayne]

# Exemplo

- Hash Table (String, String)

# 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)
```

# Tarefas

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