



<http://algs4.cs.princeton.edu>

## 3.4 TABELAS HASH

---

- ▶ *Funções de hash*
- ▶ *Encadeamento separado*
- ▶ *Exploração linear*
- ▶ *Contexto*

# Sumário: implementações de dicionário

---

Implementação	Garantia			Caso médio			Operações ordenadas ?	Interface Principal
	search	insert	delete	search hit	insert	delete		
Busca sequencial (array aleatório)	$N$	$N$	$N$	$\frac{1}{2} N$	$N$	$\frac{1}{2} N$		equals()
Busca binária (array ordenado)	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	compareTo()
ABP	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	compareTo()
ABP balanceada (ex: rubro-negra)	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	compareTo()

P. Podemos fazer melhor?

R. Sim, mas com uma forma diferente de acesso aos dados.

# Hashing: plano básico

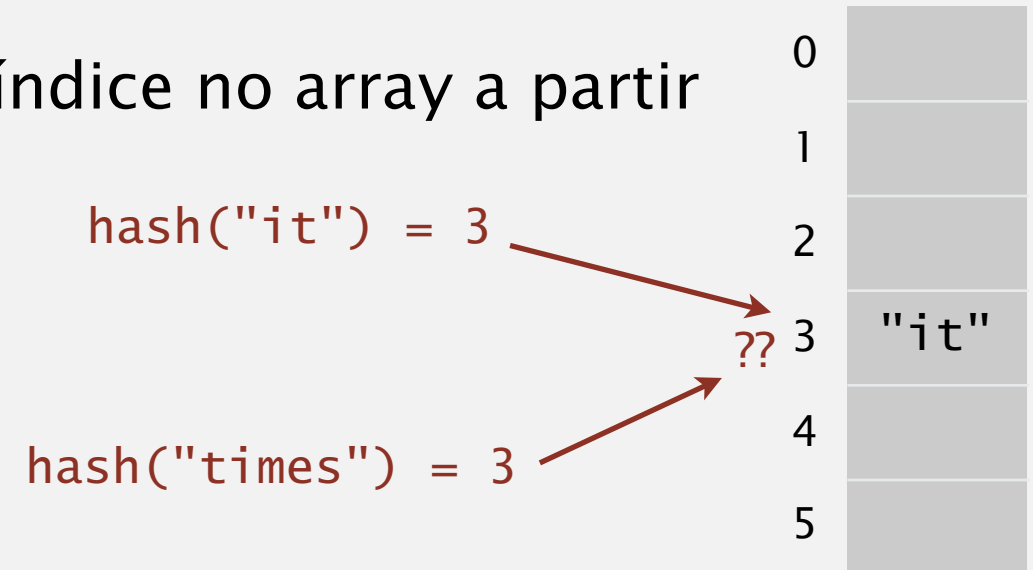
---

Armazenar itens em uma **tabela indexada por chave** (índice é uma função sobre a chave).

**Função de hash:** Método para calcular um índice no array a partir de uma chave.

## Questões:

- Como calcular a função de hash?
- Teste de igualdade: como testar se duas chaves são iguais?
- O que fazer em caso de colisões - chaves que “caem” na mesma posição do array.



## Clássico dilema de espaço x tempo:

- Sem limitação de espaço: função de hash é a própria chave!
- Sem limitação de tempo: solução trivial de colisões com busca sequencial
- Limitações de espaço e tempo: hashing (o mundo real).



<http://algs4.cs.princeton.edu>

## 3.4 TABELAS HASH

---

- ▶ *Funções de hash*
- ▶ *Encadeamento separado*
- ▶ *Busca linear*
- ▶ *Contexto*

# Calculando a função de hash

---

**Objetivo ideal:** Espalhar as chaves de forma uniforme.

- Eficientemente computável
- Cada índice igualmente provável para cada chave.

Problema amplamente pesquisado,  
ainda problemático para aplicações  
práticas

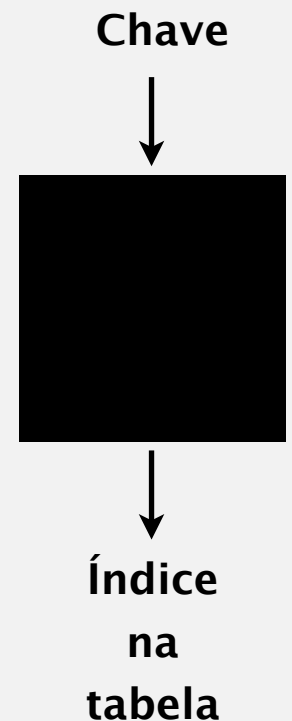
**Ex 1. Números de telefone.**

- Ruim: primeiros três dígitos.
- Melhor: últimos três dígitos.

**Ex 2. Números de matrícula.**

- Ruim: primeiros três dígitos.
- Melhor: últimos três dígitos.

201 = 2020/1, 182 = 2018/2  
(ano e semestre de entrada de cada aluno)



**Desafio prático:** Exige abordagens diferentes para cada tipo de chave.

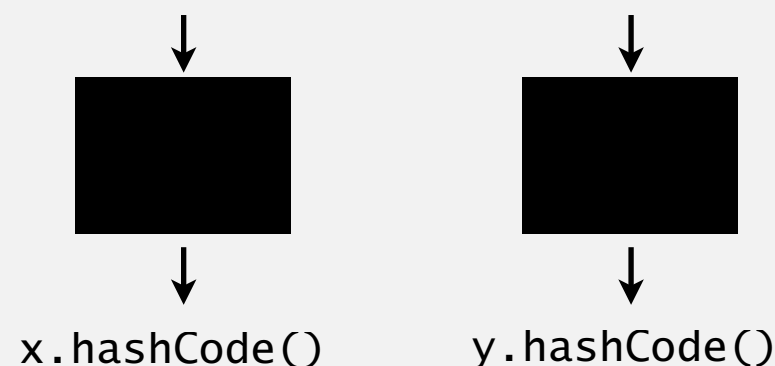
# Convenções de *hashCode* em Java

---

Todas as classes Java herdam um método `hashCode()`, que retorna um número inteiro de 32 bits.

**Exigência:** Se `x.equals(y)`, então `(x.hashCode() == y.hashCode())`.

**Desejável (muito):** Se `!x.equals(y)`, então `(x.hashCode() != y.hashCode())`.



**Implementação padrão:** Endereço de memória de `x`.

**Implementação legal (mas ruim):** Sempre retornar, por ex., 17.

**Implementações customizadas:** `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

**Tipos criados pelo usuário:** Usuários são responsáveis por criar *hashCode*

# Implementando *hashCode*: inteiros, *booleans* e *doubles*

---

## Implementações da biblioteca Java

```
public final class Integer
{
    private final int value;
    ...

    public int hashCode()
    { return value; }
}
```

```
public final class Boolean
{
    private final boolean value;
    ...

    public int hashCode()
    {
        if (value) return 1231;
        else      return 1237;
    }
}
```

```
public final class Double
{
    private final double value;
    ...

    public int hashCode()
    {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

↑  
converte para representação IEEE 64-bit;  
xor 32 bits mais significativos  
com os 32 menos significativos

Cuidado: -0.0 e +0.0 têm *hashCode* diferente!

# Implementando *hashCode*: strings

## Implementação da biblioteca Java

```
public final class String
{
    private final char[] s;
    ...

    public int hashCode()
    {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

Caractere na posição  $i$

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

- Método de Horner para string de tamanho  $L$ :  $L$  multiplicações/somas.
- Equivalente a  $h = s[0] \cdot 31^{L-1} + \dots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$ .

Ex: `String s = "call";`  
`int code = s.hashCode();`

←  $3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$   
 $= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$   
(Método de Horner)



# Hashing modular

**Código de hash:** Um int entre  $-2^{31}$  e  $2^{31} - 1$ .

**Função de hash:** Um int entre 0 e  $M - 1$  (usado com índice num array).

Tipicamente um primo ou potência de 2

```
private int hash(Key key)
{ return key.hashCode() % M; }
```

**bug**

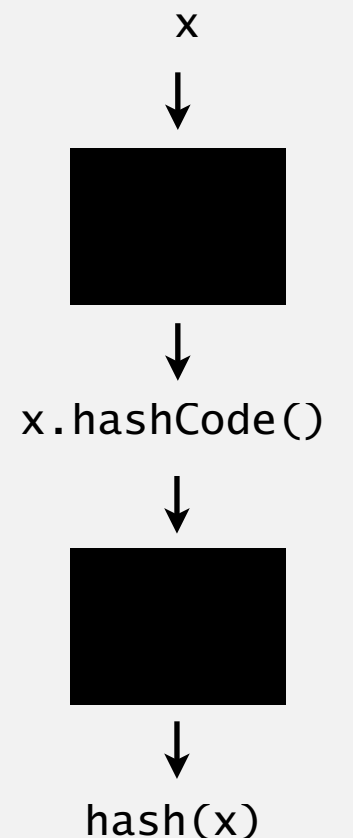
```
private int hash(Key key)
{ return Math.abs(key.hashCode()) % M; }
```

**Bug 1-em-um-bilhão**

$\text{hashCode() de "polygenelubricants" é } -2^{31}$

```
private int hash(Key key)
{ return (key.hashCode() & 0x7fffffff) % M; }
```

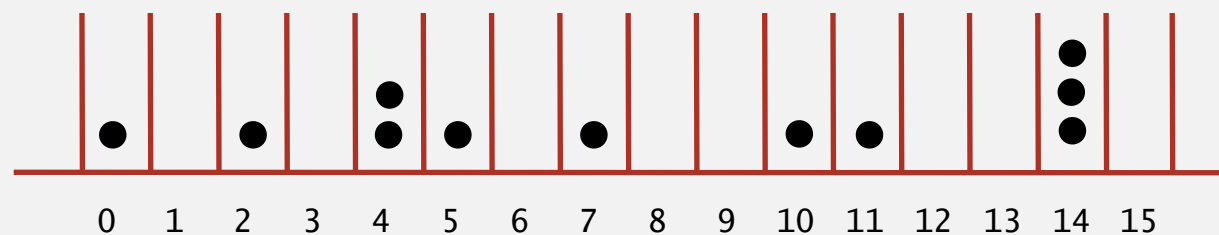
**Correto (ignora o bit de sinal)**



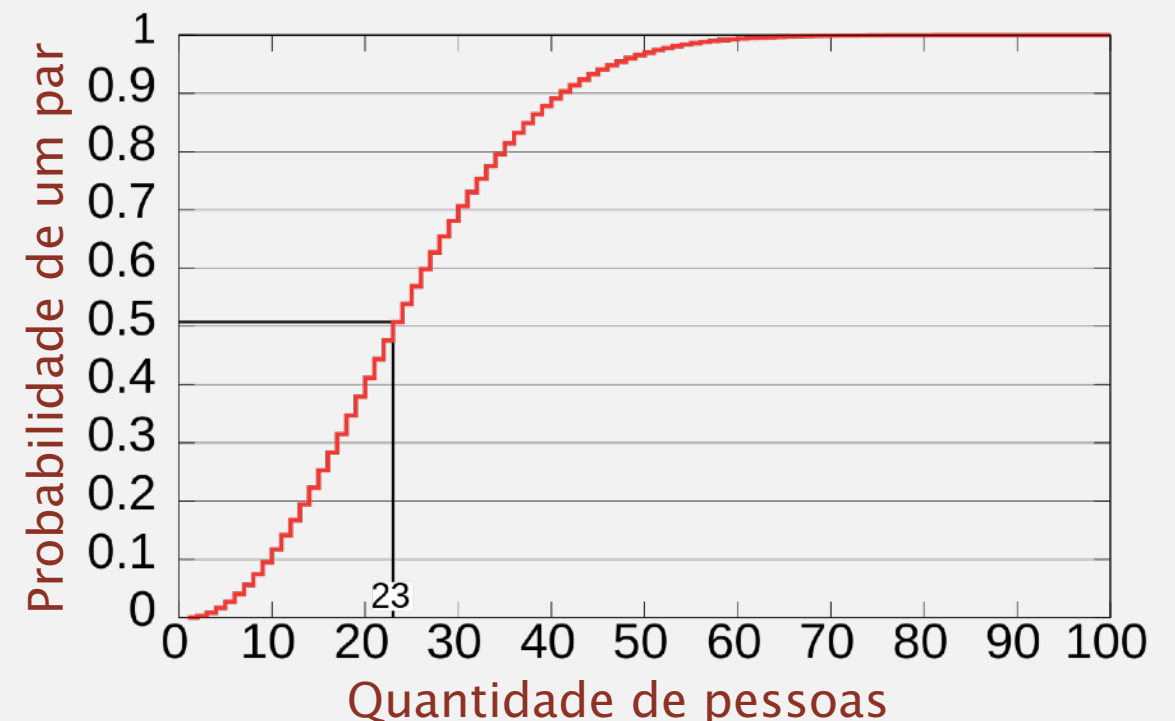
# Condição de hashing uniforme

**Hashing uniforme:** A probabilidade de cada chave gerar um inteiro entre 0 e  $M - 1$  é sempre a mesma para todas as chaves.

**Cestos e bolas:** Atire bolas aleatoriamente em  $M$  cestos.



**Problema do aniversário:** Espere duas bolas no mesmo cesto depois de  $\sim \sqrt{\pi M / 2}$  lançamentos.





<http://algs4.cs.princeton.edu>

## 3.4 TABELAS HASH

---

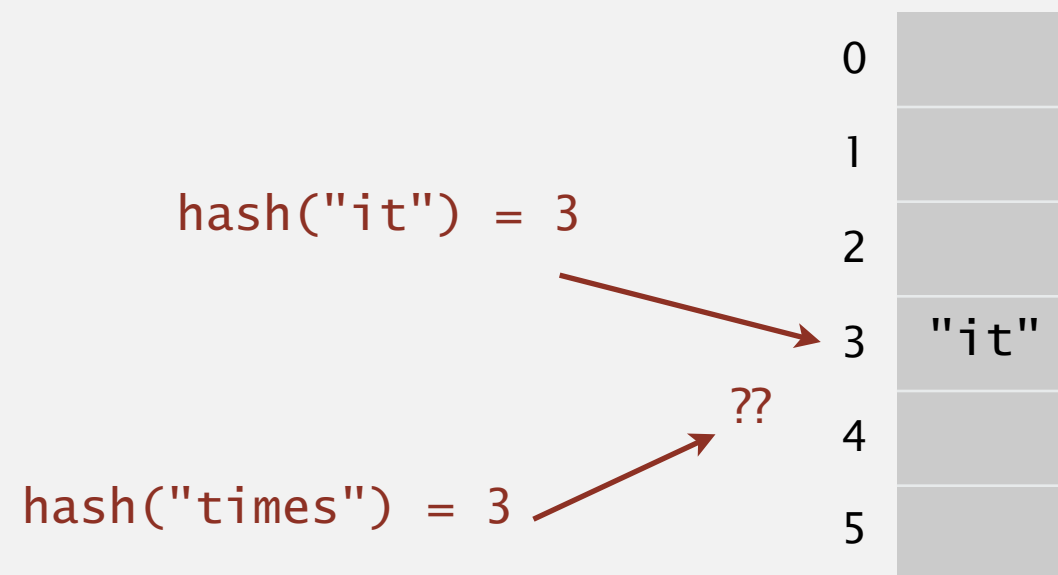
- ▶ *Funções de hash*
- ▶ **Encadeamento separado**
- ▶ *Exploração linear*
- ▶ *Contexto*

# Colisões

---

**Colisão:** Duas chaves que geram *hash* para o mesmo índice.

- Problema do aniversário  $\Rightarrow$  impossível evitar colisões a não ser que tenhamos uma quantidade enorme de memória (quadrática).
- Colisões estão igualmente distribuídas.

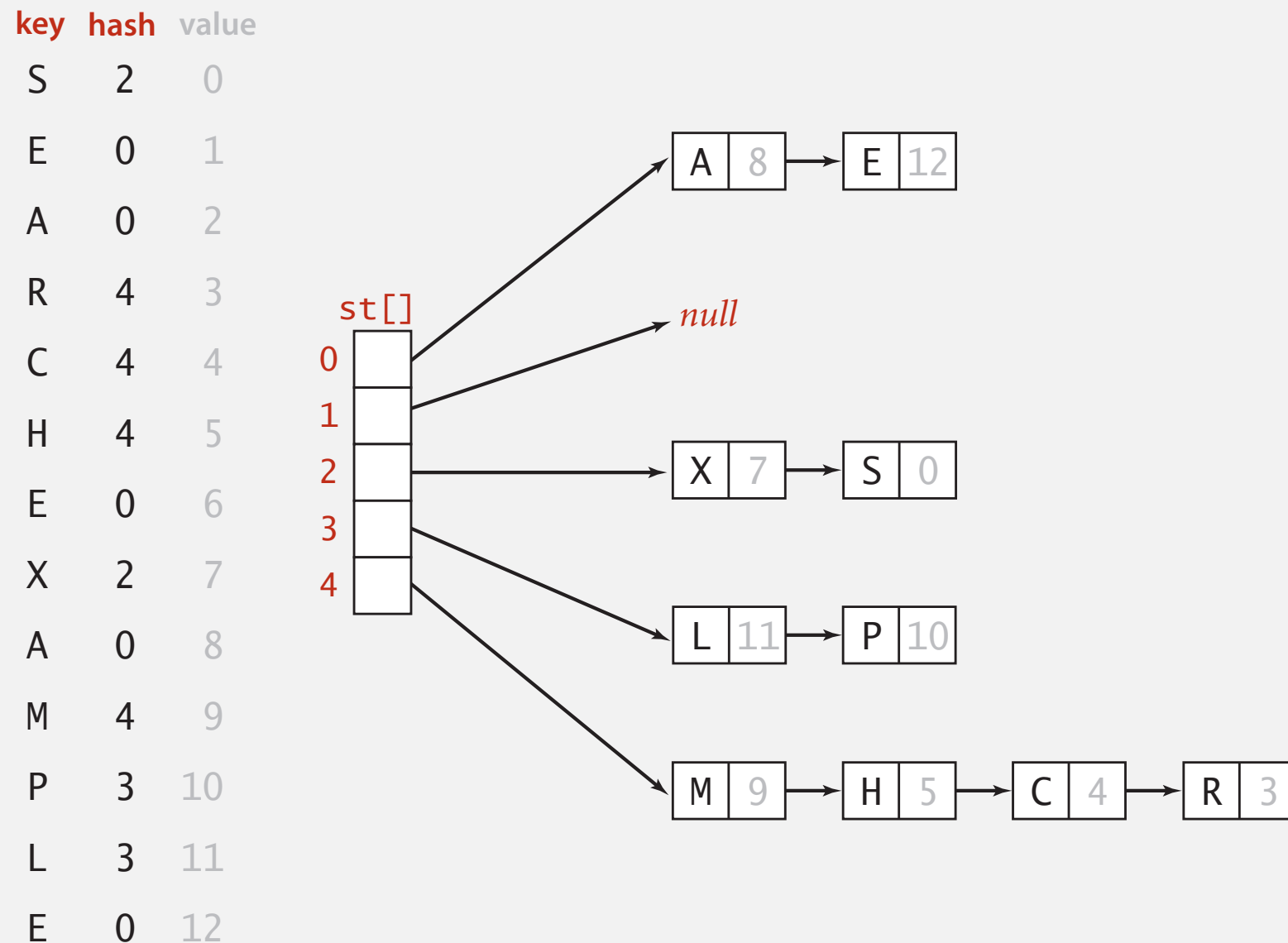


**Desafio:** Como lidar eficientemente com colisões.

# Encadeamento separado

Usar um array de  $M < N$  listas encadeadas. [H. P. Luhn, IBM 1953]

- Hash: mapeia chave para inteiro  $i$  entre 0 e  $M - 1$ .
- Inserção: inserir no início da lista na posição  $i$  (se já não estiver lá).
- Busca: precisa procurar apenas na lista  $i$ .



# Encadeamento separado: implementação Java

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;           // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }
}
```

← Código para dobrar e reduzir à metade o tamanho do array foi omitido

← Java não permite criação de array genérico (declara-se key e value do tipo Object)

# Encadeamento separado: implementação Java

---

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;           // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

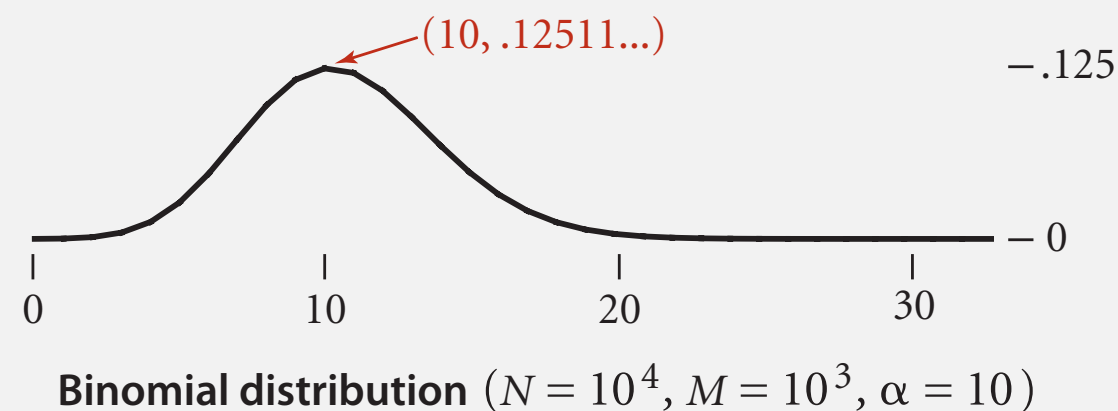
    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }
}
```

# Análise do encadeamento separado

**Hipótese:** Considerando hashing uniforme, a probabilidade que a quantidade de chaves em uma lista esteja dentro de um fator constante de  $N/M$  é extremamente próxima de 1.

**Provando:** Distribuição do tamanho de uma lista obedece uma distribuição binomial.



`equals()` e `hashCode()`

**Resultado:** Número de acessos para busca/inserção é proporcional a  $N/M$ .


- $M$  muito grande  $\Rightarrow$  muitas listas vazias.
- $M$  muito pequeno  $\Rightarrow$  listas muito longas.
- Escolha usual:  $M \sim N/4 \Rightarrow$  operações de tempo constante.

↑  
M vezes mais rápido  
que busca sequencial

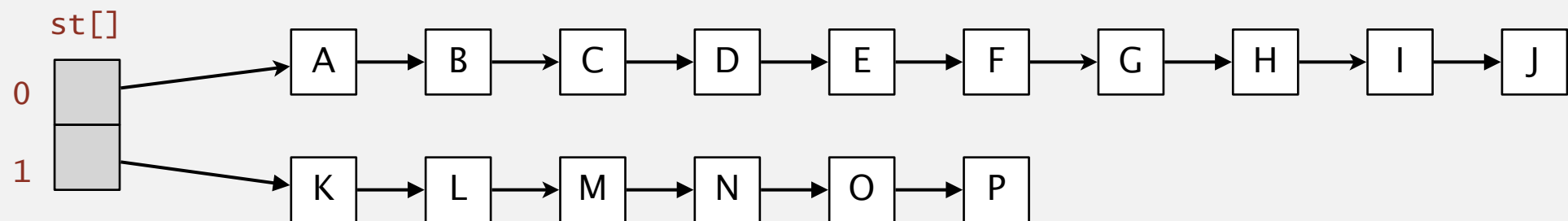


# Redimensionamento no encadeamento separado

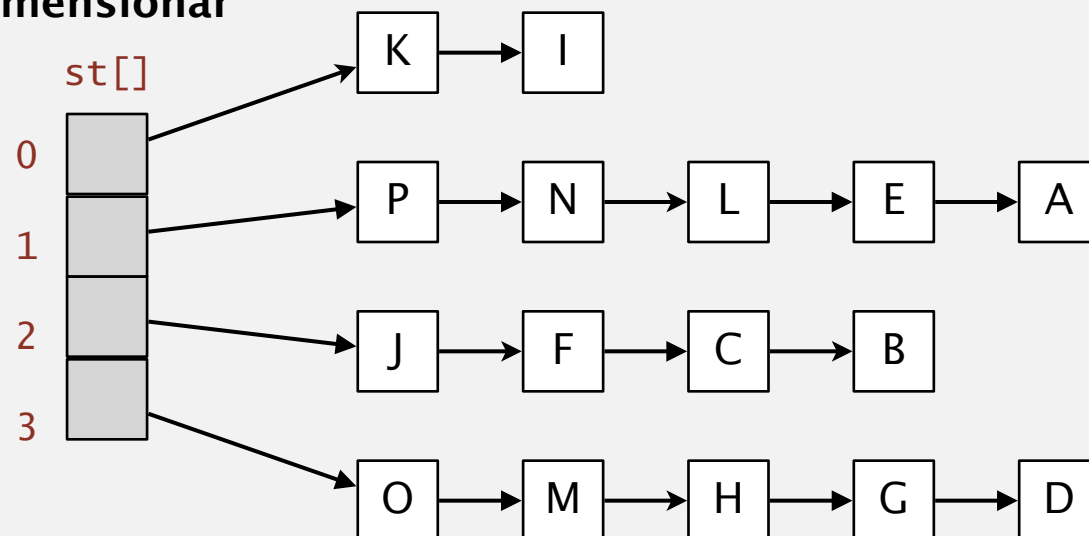
**Objetivo:** Manter o tamanho médio da lista =  $N / M = \text{constante}$ .

- Dobrar tamanho do array  $M$  quando  $N / M \geq 8$ .
- Reduzir pela metade o tamanho de  $M$  quando  $N / M \leq 2$ .
- Precisa fazer hash em todas as chaves!   $x.\text{hashCode}()$  não muda, mas  $\text{hash}(x)$  pode mudar

Antes de redimensionar



Depois de redimensionar

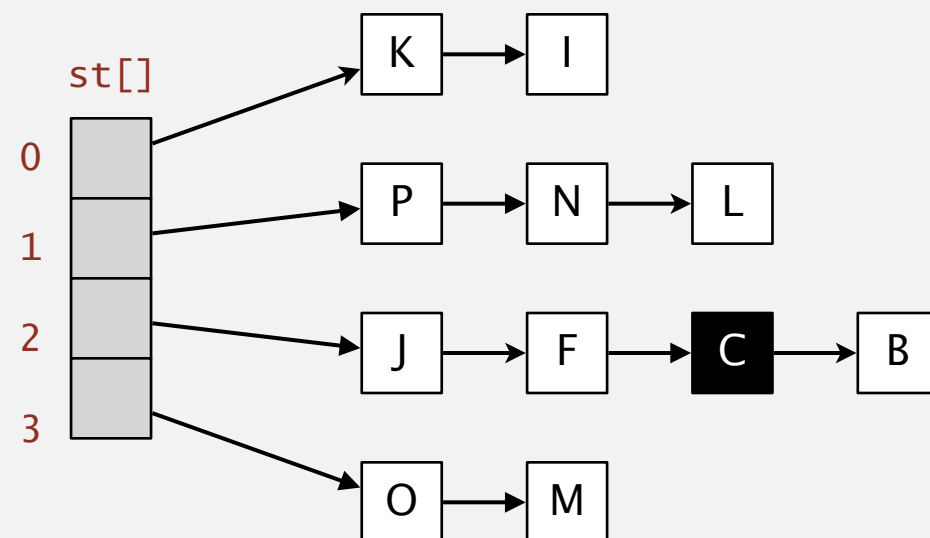


# Remoção em encadeamento separado

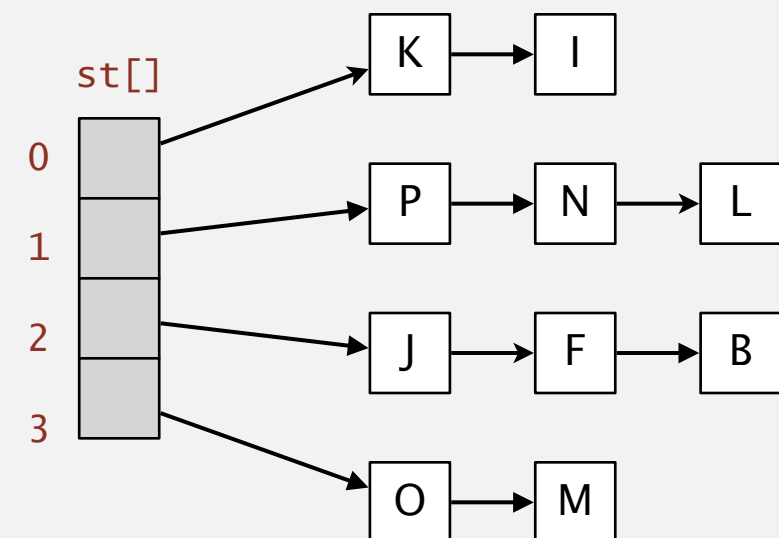
**P:** Como remover uma chave (e seu valor associado?)

**R:** Fácil: precisa apenas considerar a lista contendo a chave.

Antes de remover C



Depois de remover C



# Sumário: implementações de dicionário

Implementação	Garantia			Caso médio			Operações ordenadas ?	Interface principal
	search	insert	delete	search hit	insert	delete		
Busca sequencial (array aleatório)	$N$	$N$	$N$	$\frac{1}{2} N$	$N$	$\frac{1}{2} N$		<code>equals()</code>
Busca binária (array ordenado)	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
ABP	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	<code>compareTo()</code>
ABP balanceada (ex: rubro-negra)	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>
Hashing por encadeamento separado	$N$	$N$	$N$	$3-5 *$	$3-5 *$	$3-5 *$		<code>equals()</code> <code>hashCode()</code>

\* considerando hashing uniforme



<http://algs4.cs.princeton.edu>

## 3.4 TABELAS HASH

---

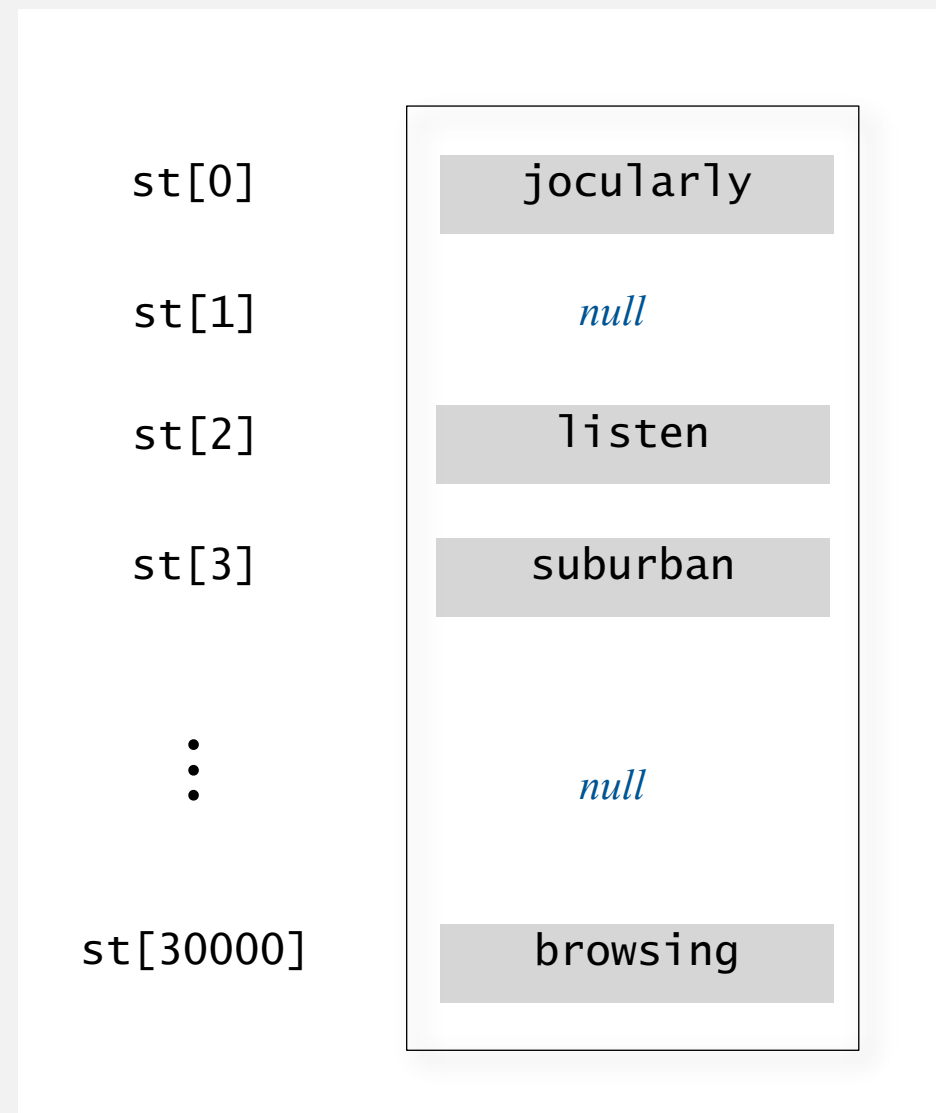
- ▶ *Funções de hash*
- ▶ *Encadeamento separado*
- ▶ ***Exploração linear***
- ▶ *Contexto*

# Resolução de colisões: endereçamento aberto

---

**Endereçamento aberto:** [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

Quando uma nova chave colide, encontra um espaço livre e armazena ela nele.



Exploração linear ( $M = 30001$ ,  $N = 15000$ )

# Exploração linear: demonstração

---

**Hash:** Mapeia chave para inteiro  $i$  entre  $0$  e  $M-1$ .

**Inserção:** Armazena no índice  $i$  se estiver livre, senão tenta  $i+1$ ,  $i+2$ , etc.

**Tabela hash por exploração linear**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

$M = 16$



# Exploração linear: demonstração

---

**Hash:** Mapeia chave para inteiro  $i$  entre  $0$  e  $M-1$ .

**Inserção:** Armazena no índice  $i$  se estiver livre, senão tenta  $i+1$ ,  $i+2$ , etc.

search  $K$   
 $\text{hash}(K) = 5$

	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

$M = 16$

$K$

Não encontra  
(retorna null)

# Exploração linear: sumário

---

**Hash:** Mapeia chave para inteiro  $i$  entre  $0$  e  $M-1$ .

**Inserção:** Armazena no índice  $i$  se estiver livre, senão tenta  $i+1$ ,  $i+2$ , etc.

**Busca:** Busca na posição  $i$ , se estiver ocupada mas não for a chave, tenta  $i+1$ ,  $i+2$ , etc.

**Nota:** Tamanho do array  $M$  **precisa ser maior** que a quantidade  $N$  de pares chave-valor.

	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

$M = 16$



# Exploração linear: implementação Java


```
public class LinearProbingHashST<Key, Value>
{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];

    private int hash(Key key) { /* como antes */ }

    private void put(Key key, Value val) { /* próximo slide */ }

    public Value get(Key key)
    {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M)
            if (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
```

Código para dobrar e  
reduzir à metade o  
tamanho do array foi  
omitido



# Exploração linear: implementação Java

---

```
public class LinearProbingHashST<Key, Value>
{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];

    private int hash(Key key) { /* como antes */ }

    private Value get(Key key) { /* slide anterior */ }

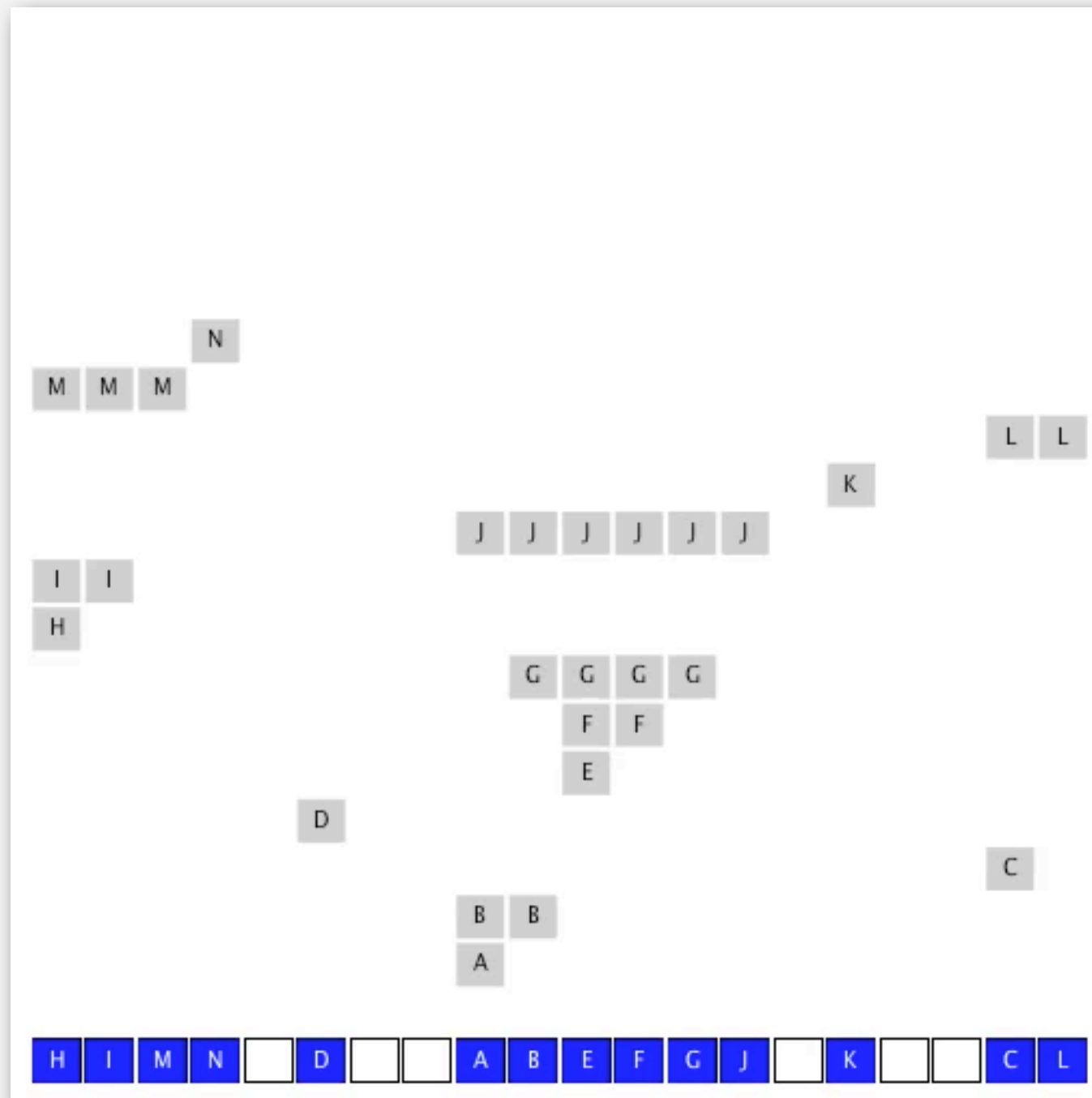
    public void put(Key key, Value val)
    {
        int i;
        for (i = hash(key); keys[i] != null; i = (i+1) % M)
            if (keys[i].equals(key))
                break;
        keys[i] = key;
        vals[i] = val;
    }
}
```

# Clustering

---

**Cluster:** Um bloco contíguo de itens.

**Observação:** Novas chaves têm muitas chances de caírem no meio de clusters grandes.



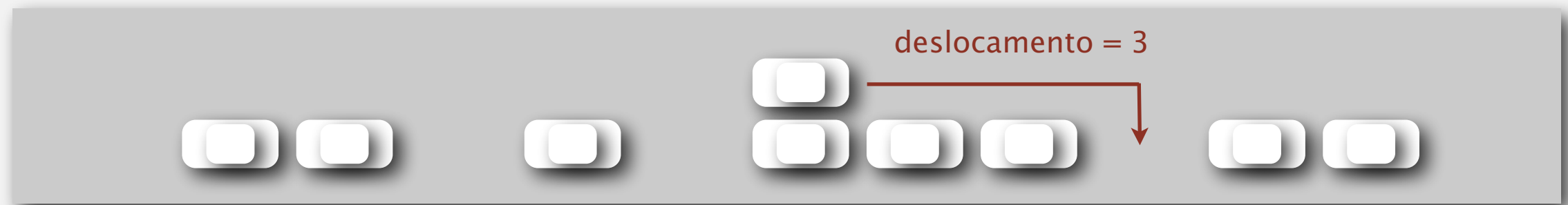
# Problema do estacionamento de Knuth

---

**Modelo:** Carros numa rua de mão única com  $M$  locais para estacionar.

Cada um quer um local aleatório  $i$ : se estiver ocupado, tenta  $i + 1, i + 2$ , etc.

**P:** Qual é o deslocamento médio de um carro?



**Metade cheio:** Com  $M / 2$  carros, deslocamento médio é  $\sim 3 / 2$ .

**Cheio:** Com  $M$  carros, deslocamento é  $\sim \sqrt{\pi M / 8}$ .

# Análise da exploração linear

**Proposição:** Considerando hashing uniforme, a quantidade de consultas numa tabela hash de tamanho  $M$  que contém  $N = \alpha M$  chaves é:

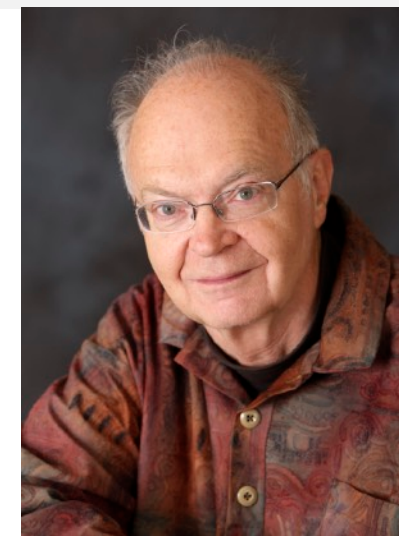
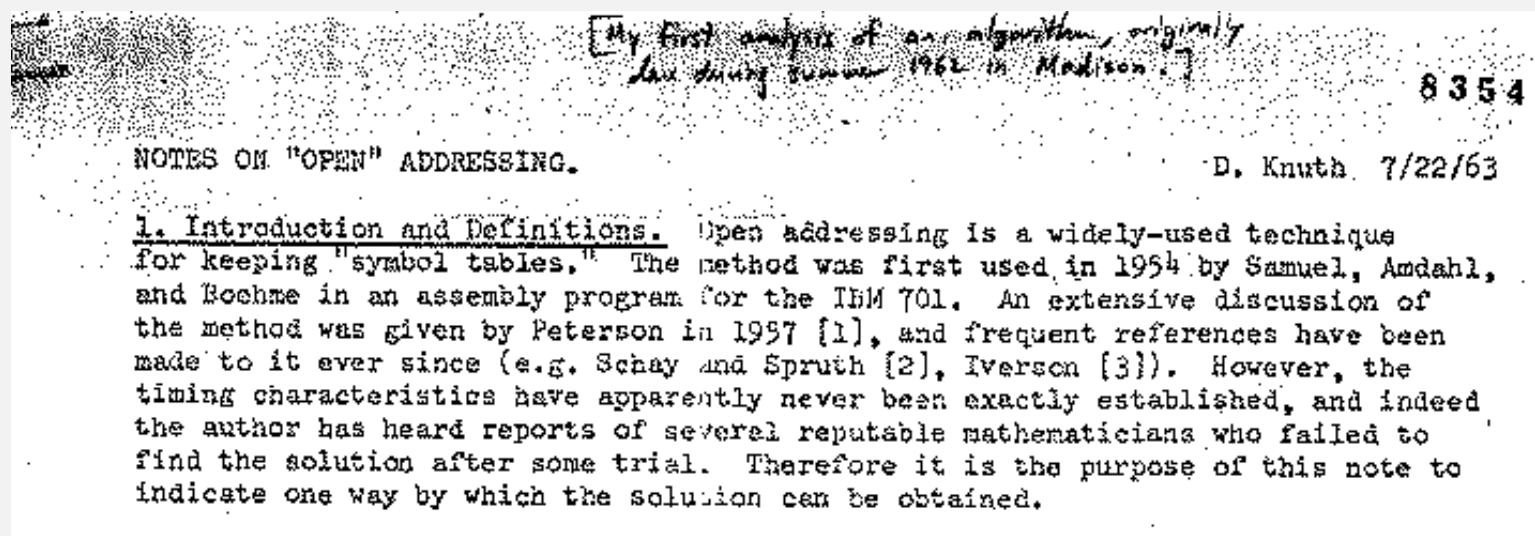
$$\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \sim \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$

Acerto

Erro / inserção

$\alpha \Rightarrow$  percentual de ocupação

**Prova:**



**Parâmetros:**

- $M$  muito grande  $\Rightarrow$  muitos espaços vazios no array.
- $M$  muito pequeno  $\Rightarrow$  tempo de busca "explode".
- Escolha típica:  $\alpha = N / M \sim 1/2$ . ← # buscas para acerto: aprox. 3/2  
# buscas para erro: aprox. 5/2

# Redimensionamento em exploração linear

---

**Objetivo:** Manter o tamanho médio da busca  $N / M \leq \frac{1}{2}$ .

- Dobrar tamanho do array  $M$  quando  $N / M \geq \frac{1}{2}$ .
- Reduzir à metade o tamanho do array  $M$  quando  $N / M \leq \frac{1}{8}$ .
- Necessário fazer hash de todas as chaves novamente.

Antes de redimensionar

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

Depois de redimensionar

	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	

# Remoção em exploração linear

**P:** Como remover uma chave (e seu valor associado)?

**R:** Requer algum cuidado: não podemos simplesmente apagar entradas!

Antes de remover 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

Não funciona, por ex. se  $\text{hash}(H) = 4$

Depois de remover 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

# Remoção em exploração linear

**P:** Como remover uma chave (e seu valor associado)?

**R:** Requer algum cuidado: não podemos simplesmente apagar entradas!

Solução: marcar que o espaço já havia sido ocupado alguma vez

Antes de remover 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
used[]	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1

Agora funciona, pois se  $\text{hash}(H) = 4$   
vai passar pelo 6 pois  $\text{used}[6]$  contém 1

Depois de remover 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
used[]	1	1	0	0	1	1	1	1	1	0	1	0	0	0	1	1



# Sumário: implementações de dicionário

Implementação	Garantia			Caso médio			Operações ordenadas ?	Interface principal
	search	insert	delete	search hit	insert	delete		
Busca sequencial (array aleatório)	$N$	$N$	$N$	$\frac{1}{2} N$	$N$	$\frac{1}{2} N$		<code>equals()</code>
Busca binária (array ordenado)	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
ABP	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	<code>compareTo()</code>
ABP balanceada (ex: rubro-negra)	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>
Hashing por encadeamento separado	$N$	$N$	$N$	$3-5 *$	$3-5 *$	$3-5 *$		<code>equals()</code> <code>hashCode()</code>
Hashing por exploração linear	$N$	$N$	$N$	$3-5 *$	$3-5 *$	$3-5 *$		<code>equals()</code> <code>hashCode()</code>

\* considerando hashing uniforme



<http://algs4.cs.princeton.edu>

## 3.4 TABELAS HASH

---

- ▶ *Funções de hash*
- ▶ *Encadeamento separado*
- ▶ *Exploração linear*
- ▶ **Contexto**

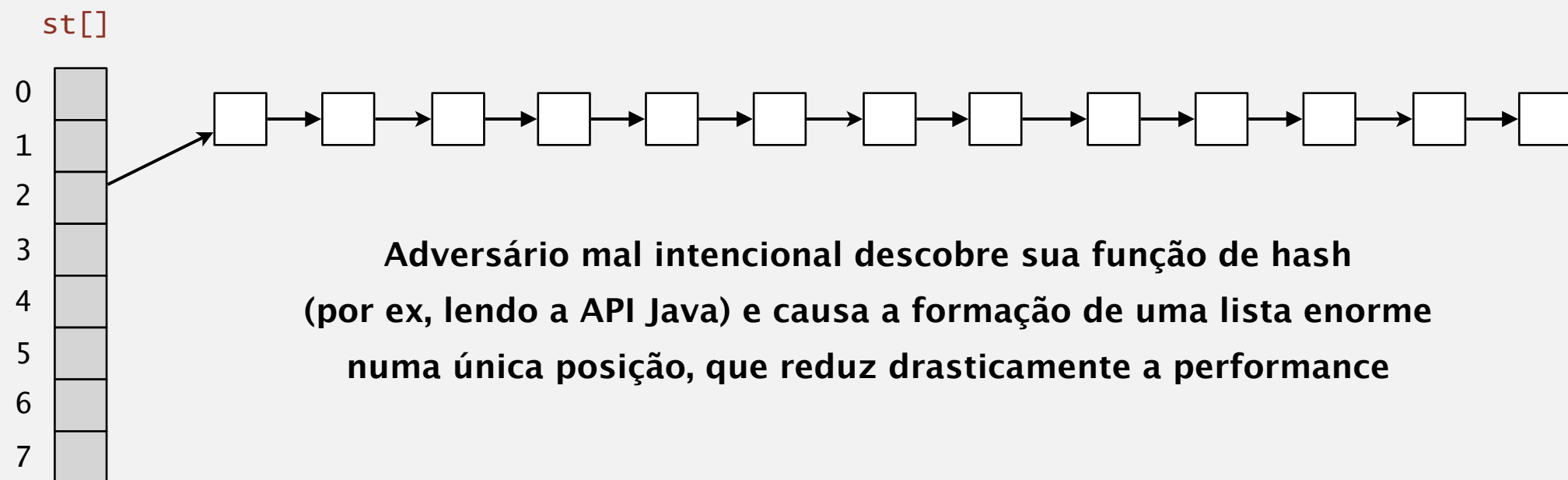
# História de guerra: ataques de complexidade algorítmica

---

**P:** Por que a ideia de hashing uniforme é importante na prática?

**R:** Situações óbvias: controle de aeronave, reator nuclear, marca-passo.

**R:** Situações surpreendentes: ataques de **negação de serviço**.



**Ataques reais:** [Crosby-Wallach 2003]

- Servidor Bro: enviar pacotes bem escolhidos para criar negação de serviço, usando menos largura de banda que uma linha discada
- Perl 5.8.0: inserir strings bem escolhidas no array associativo.
- Kernel Linux 2.4.20: gravar arquivos com nomes bem escolhidos.

# História de guerra: ataques de complexidade algorítmica

---

## Um relatório de bugs no Java:

Jan Lieskovsky 2011-11-01 10:13:47 EDT

Description

Julian Wälde and Alexander Klink reported that the `String.hashCode()` hash function is not sufficiently collision resistant. `hashCode()` value is used in the implementations of `HashMap` and `Hashtable` classes:

<http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html>  
<http://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.html>

A specially-crafted set of keys could trigger hash function collisions, which can degrade performance of `HashMap` or `Hashtable` by changing hash table operations complexity from an expected/average  $O(1)$  to the worst case  $O(n)$ . Reporters were able to find colliding strings efficiently using equivalent substrings and meet in the middle techniques.

This problem can be used to start a denial of service attack against Java applications that use untrusted inputs as `HashMap` or `Hashtable` keys. An example of such application is web application server (such as tomcat, see [bug #750521](#)) that may fill hash tables with data from HTTP request (such as GET or POST parameters). A remote attack could use that to make JVM use excessive amount of CPU time by sending a POST request with large amount of parameters which hash to the same value.

This problem is similar to the issue that was previously reported for and fixed in e.g. perl:

[http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach\\_UsenixSec2003.pdf](http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf)

# Ataque de complexidade algorítmica no Java

---

**Objetivo:** Encontrar família de strings com mesmo *hashCode*.

**Solução:** O código de *hashCode* que é parte da API de strings.

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBaA"	-540425984
"AaAaBBBB"	-540425984
"AaBBaAaA"	-540425984
"AaBBaABB"	-540425984
"AaBBBBaA"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBaAaAaA"	-540425984
"BBaAaABB"	-540425984
"BBaABBAa"	-540425984
"BBaBBBBB"	-540425984
"BBBBaAaA"	-540425984
"BBBBaABB"	-540425984
"BBBBBBaA"	-540425984
"BBBBBBBB"	-540425984

**2<sup>N</sup> strings de tamanho 2N que geram o mesmo hash!**

## Distração: funções de hash *one-way*

---

**Função de hash *one-way*:** “Difícil” encontrar uma chave que irá gerar um hash para um valor desejado (ou duas chaves que gerem o mesmo hash).

**Ex.** MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160, ....

*Sabe-se que são inseguras*

```
String password = args[0];  
MessageDigest sha1 = MessageDigest.getInstance("SHA1");  
byte[] bytes = sha1.digest(password);  
  
/* Escrever os bytes como string hexadecimal */
```

**Aplicações:** Assinatura digital, resumos de mensagens, senhas.

**Ressalva:** Muito custosa para usar em aplicações de dicionário.

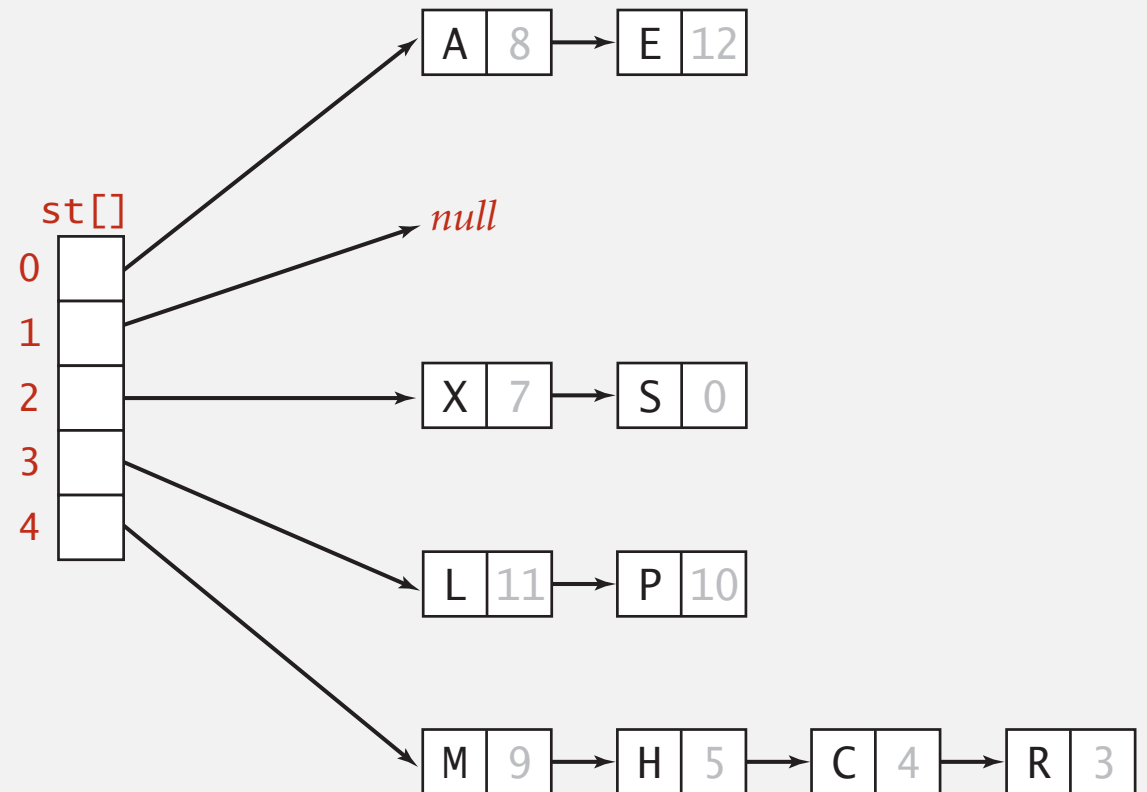
# Encadeamento separado x exploração linear

## Encadeamento separado:

- Performance degrada "graciosamente".
- Clustering menos sensível a funções de hash ruins.

## Exploração linear:

- Menos espaço perdido.
- Melhor performance de cache.



	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

# Tabelas hash x árvores de pesquisa balanceadas

---

## Tabelas hash:

- Código mais simples.
- Não há outra alternativa efetiva para chaves sem critério de ordem.
- Mais rápidas para chaves simples (poucas operações ao invés de  $\log N$  comparações).
- Melhor suportadas em Java para strings (faz cache dos hash codes).

## Árvores de pesquisa balanceadas:

- Performance garantida.
- Suporte para operações de dicionário ordenado.
- Mais fácil implementar `compareTo()` do que `equals()` e `hashCode()`.

## Sistema Java inclui ambas:

- ABPs rubro-negras: `java.util.TreeMap`, `java.util.TreeSet`.
- Tabelas hash: `java.util.HashMap`, `java.util.IdentityHashMap`.