

ESD 2

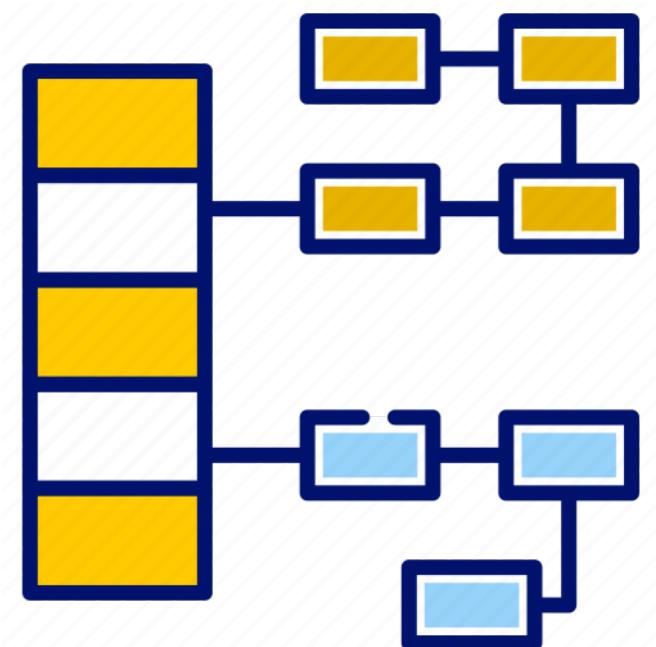
Estrutura de
Dados

Tabela Hash

Prof. Fernando Sambinelli

Tabela Hash

- Uma tabela hash (também conhecida por tabela de espalhamento ou tabela de dispersão) é uma estrutura de dados especial, que associa chaves de pesquisa a valores/objetos.
- Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado.
- A principal vantagem das tabelas hash é sua eficiência na busca, inserção e remoção de elementos, principalmente quando o número de elementos é grande. Em uma implementação bem-sucedida, o tempo de busca é geralmente $O(1)$, ou seja, constante, independentemente do tamanho da coleção de dados.

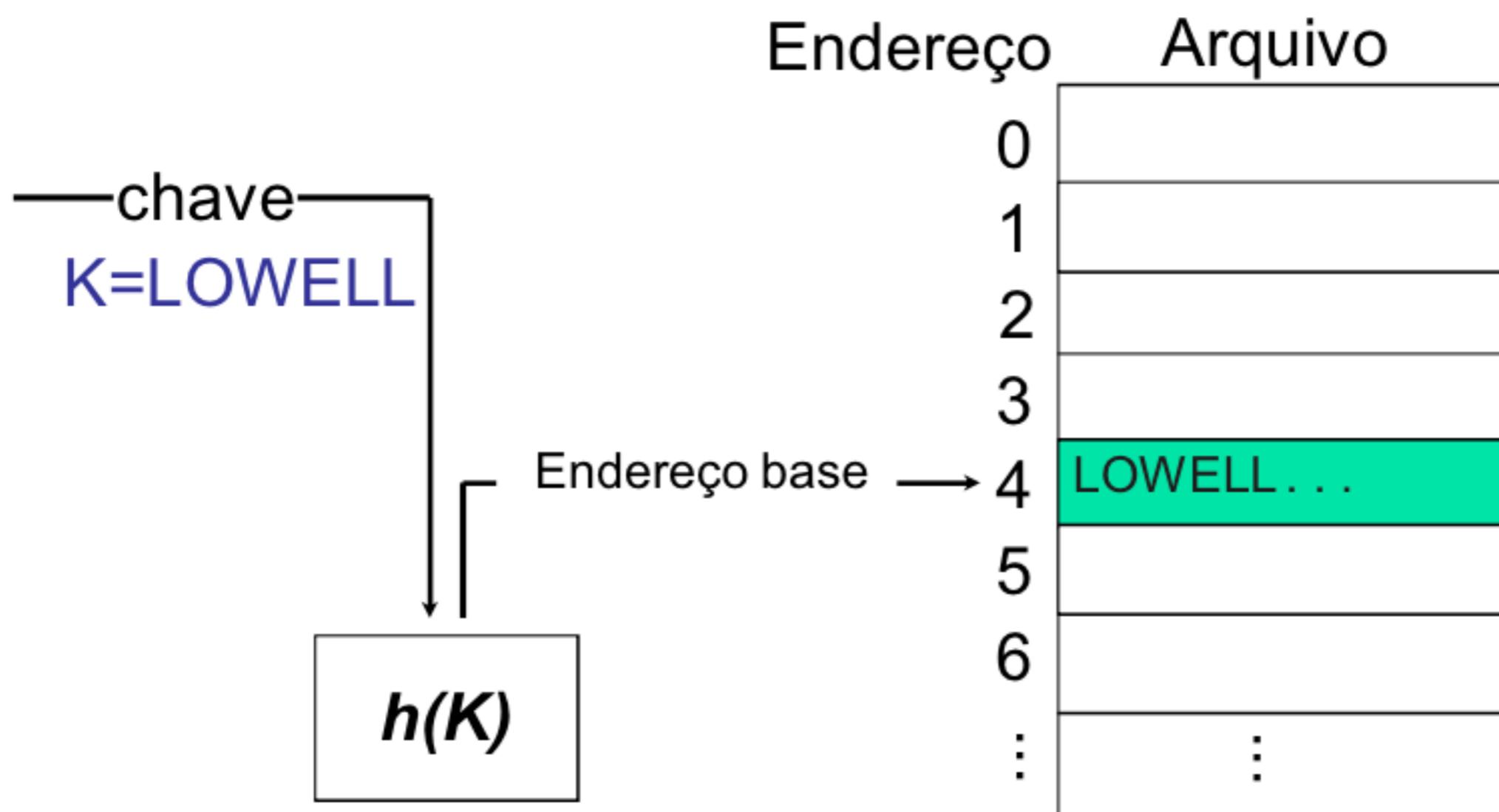


Função Hash

- Uma função de espalhamento (função hash) $h(k)$ transforma uma chave k em um endereço
 - *Este endereço é usado como a base para o armazenamento e recuperação de registros*
 - *É similar a uma indexação, pois associa a chave ao endereço relativo do registro*



Funcionamento do Hashing



Funcionamento do Hashing

- Suponha que foi reservado espaço para manter 1.000 registros e considere a seguinte $h(K)$:
 - Obter as representações ASCII dos dois primeiros caracteres do sobrenome
 - Multiplicar estes números e usar os três dígitos menos significativos do resultado para servir de endereço

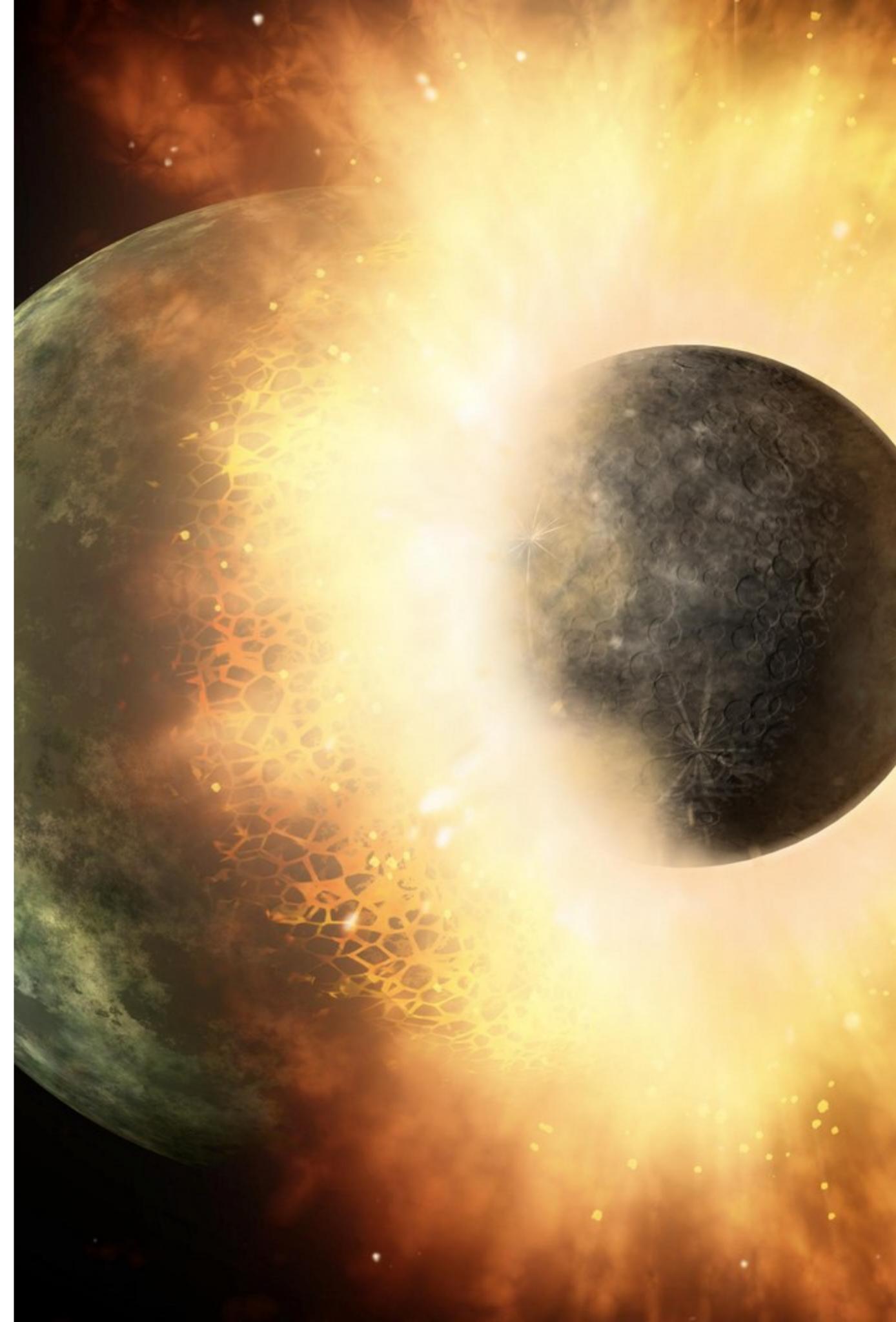
Name	Código ASCII para as 2 primeiras letras	Produto	Endereço
<u>BALL</u>	66 65	$66 \times 65 = 4.290$	290
<u>LOWELL</u>	76 96	$76 \times 96 = 6.004$	004
<u>TREE</u>	84 82	$84 \times 82 = 6.888$	888

ASCII

Decim al	letra	Decim al	letra
65	A	78	N
66	B	79	O
67	C	80	P
68	D	81	Q
69	E	82	R
70	F	83	S
71	G	84	T
72	H	85	U
73	I	86	V
74	J	87	W
75	K	88	X
76	L	89	Y
77	M	90	Z

Colisões

- Duas palavras diferentes podem produzir o mesmo endereço -> as chaves são sinônimas
- Temos, $h(\text{LOWELL}) = h(\text{LOCK}) = h(\text{OLIVER})$
- Maneiras de reduzir o número de colisões:
 - Utilizar um algoritmo que **distribua** os registros relativamente por igual entre os endereços disponíveis
 - Utilização de mais memória
 - Utilização de mais de um registro por endereço:
 - nesse caso, o endereço (chamado de cesto - **bucket**) tem espaço para armazenar vários registros.



Exemplo de Função Hashing

- A função apresentada a seguir é bastante eficiente na maioria dos casos e é facilmente modificável para acomodar ajustes que considerem a relevância das chaves
- **Exemplo 1: Representar a chave numericamente** (caso a chave não seja numérica). Por exemplo, usar os códigos ASCII dos caracteres (todos os caracteres da chave) para formar um número. Por exemplo:
 - $\text{LOWELL}$$$$ = 76 \ 79 \ 87 \ 69 \ 76 \ 76 \ 32 \ 32 \ 32 \ 32 \ 32$ (considera+ 6 brancos)



Exemplo de Função Hashing

- **Exemplo 2: Subdividir o número em partes e somar as partes:**
 - Exemplo de Subdivisão: 76 79 | 87 69 | 76 76 | 32
32 | 32 32 | 32 32 Soma das partes: $7679 + 8769 + 7676 + 3232 + 3232 + 3232 = 33820$
 - Suponha que utilizemos um endereço de 15 bits, o limite será 32767. Se aplicássemos a soma direta das partes $7679 + 8769 + 7676 + 3232 + 3232 + 3232 = 33820 > 32767$ (overflow!)
 - Considerando que o resultado permitido seja = 19937
 - Para assegurar que nenhum endereço excede 19937 fazemos: $33820 \bmod 19937 = 13883$



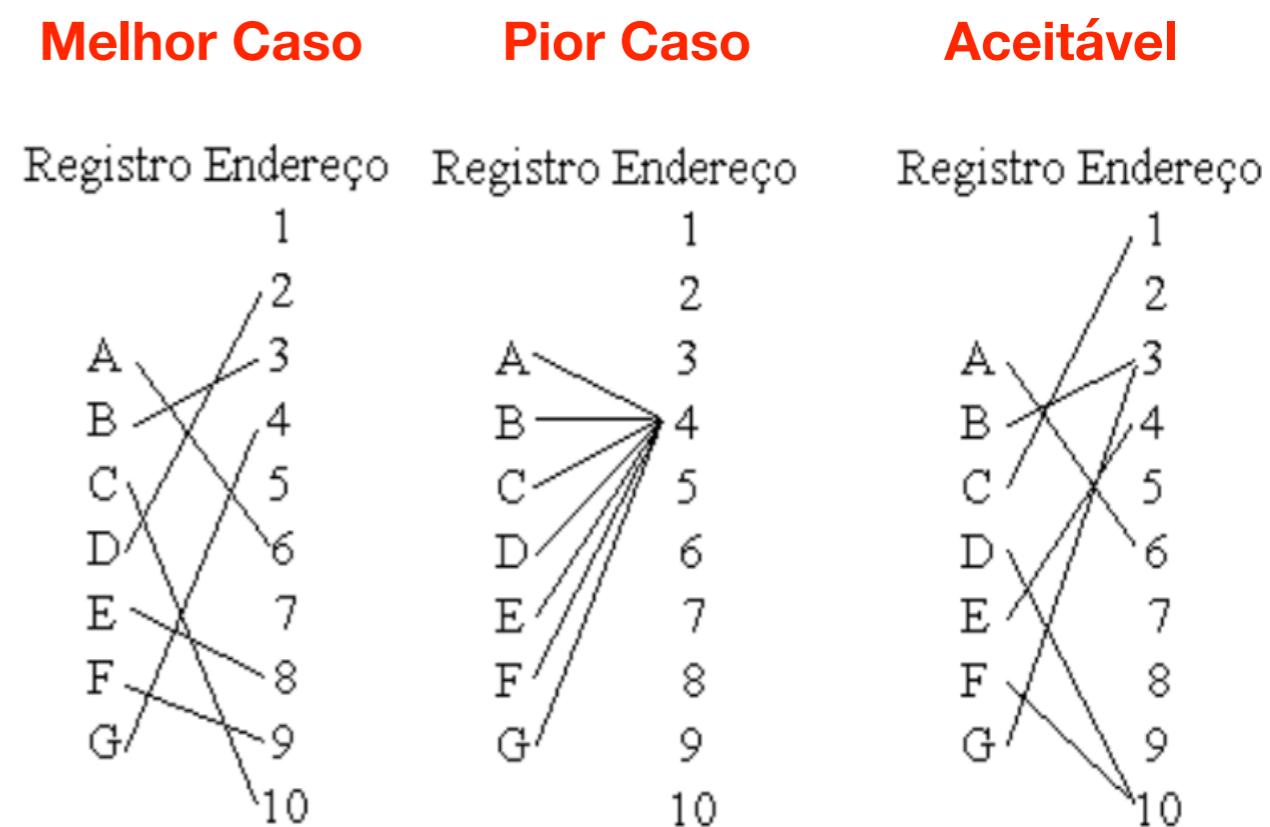
Exemplo de Função Hashing

- Exemplo 3: **Divisão do resultado da soma pelo espaço de endereçamento para obter o endereço final.** O objetivo é cortar o tamanho produzido no passo 2 de forma que ele caia no intervalo de endereços do arquivo.
- $a = s \text{ MOD } n$, onde:
 - a é o endereço base da chave
 - s é a soma produzida no passo 2
 - n é o número de endereços no arquivo
- O resto da divisão é um número entre 0 e $n-1$.
Recomenda-se que o valor escolhido para n seja um número primo.
- Exemplo: suponha que desejamos usar 101 endereços 0-100 para um arquivo $a= 13883 \text{ mod } 101 = 46$ (RRN do registro com a chave)



Distribuição de Registros entre Endereços

- A função de espalhamento perfeita (ou uniforme) para um dado conjunto de chaves seria uma que não produzisse nenhum **sinônimo** em um dado espaço de endereçamento
- A pior função seria aquela que, qualquer que fosse a chave, geraria sempre o mesmo endereço (todas as chaves seriam sinônimas)
- Uma função aceitável é uma que gera poucos sinônimos



Distribuição de Registros entre Endereços

Vejamos agora alguns métodos de espalhamento potencialmente melhores que os aleatórios.

1. Examinar as chaves buscando um certo padrão:

- Os números de identificação dos funcionários de uma empresa podem estar ordenados de acordo com a ordem de entrada dos funcionários na empresa. Se uma parte das chaves mostrar um padrão, pode-se usar um função que extrai esta parte

2. Separar partes da chave:

- Pode-se extrair dígitos de uma parte da chave e somar estes dígitos. Este método destrói eventuais padrões nas chaves originais, mas em algumas circunstâncias pode fazer a separação entre sub-conjuntos das chaves que se espalham naturalmente



Distribuição de Registros entre Endereços

3. Dividir a chave por um número primo

4. Elevar a chave ao quadrado e considerar os dígitos intermediários do resultado

5. Transformação Radix: Mudar de base e dividir pelo espaço de endereçamento

- Os 3 primeiros métodos tentam explorar a ordem natural que pode existir entre as chaves
- Os 2 últimos tentam explorar a "randomização" das chaves



Exemplo de Uso da Divisão por Números Primos

- Se as chaves são inteiros positivos, podemos usar a função modular (resto da divisão por M):

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

- Exemplos com $M = 100$ e com $M = 97$ (ao lado)

- No caso de strings, podemos iterar hashing modular sobre os caracteres da string:

```
int h = 0;  
for (int i = 0; i < s.length(); i++)  
    h = (31 * h + s.charAt(i)) % M;
```

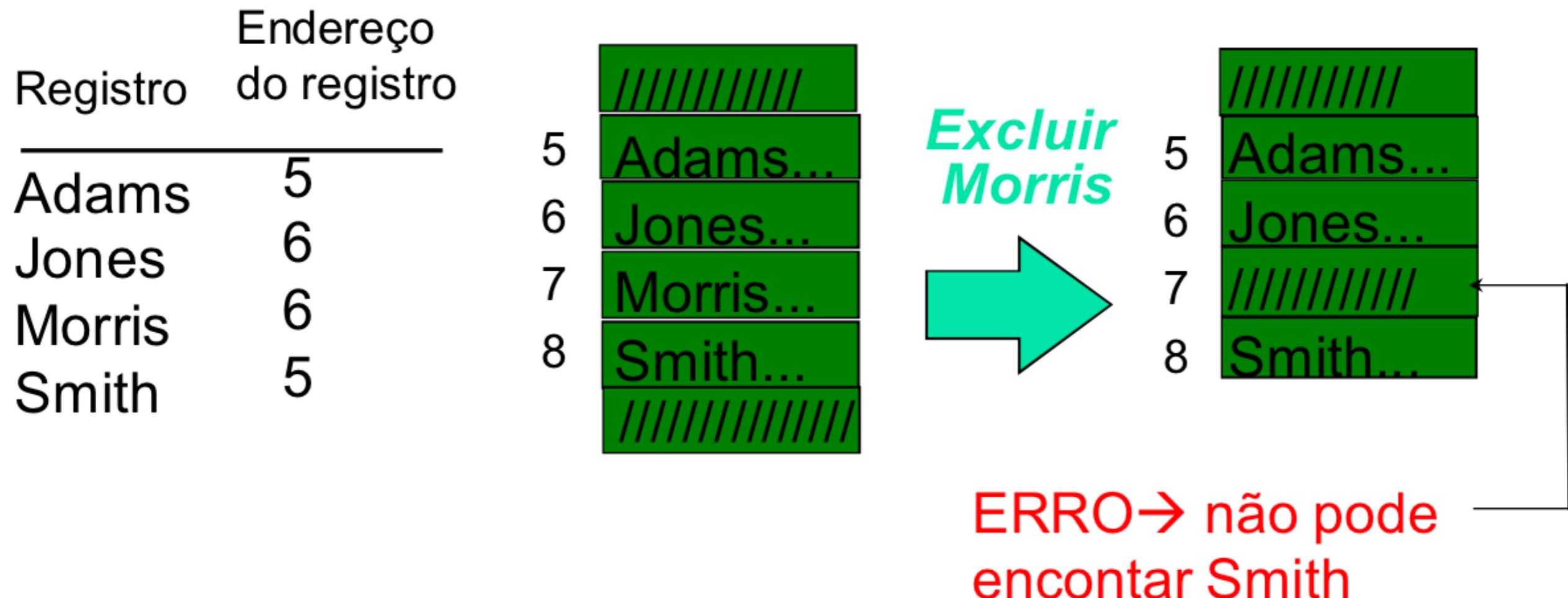
key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Remoção de Arquivo Hash

- **A eliminação de um registro requer alguns cuidados:**
 - A vaga liberada pela eliminação não deve impedir futuras pesquisas
 - Deve ser possível reutilizar o espaço liberado para futuras adições
 - a utilização de uma marca especial para identificar registros eliminados é um exemplo de estratégia de remoção

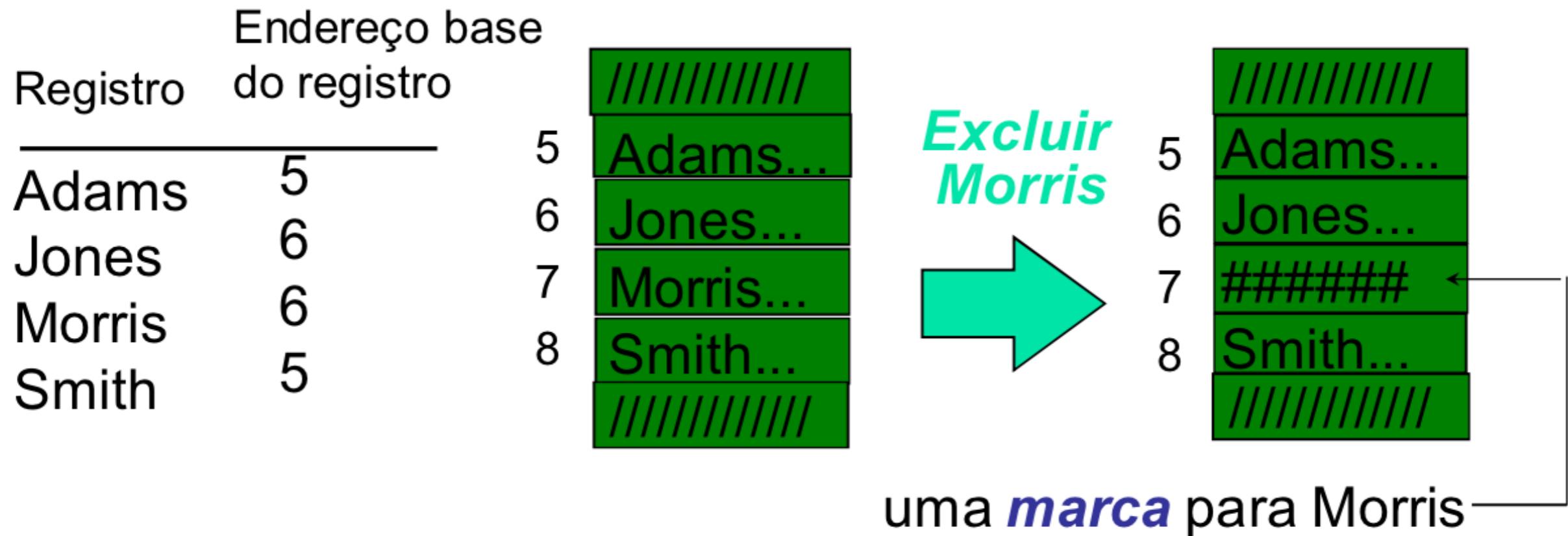


Exemplo de Remoção



Nesse exemplo, temos uma função hash com overflow progressivo, ou seja, a pesquisa termina quando um espaço vago é encontrado.

Exemplo de Remoção



1. A pesquisa deve continuar quando encontra a marca, mas deve parar quando encontra um endereço vazio
2. Inserir uma **marca** somente se o próximo registro está ocupado ou é uma marca => **? se o próx reg está vazio o reg removido pode ser feito igual ao vazio**

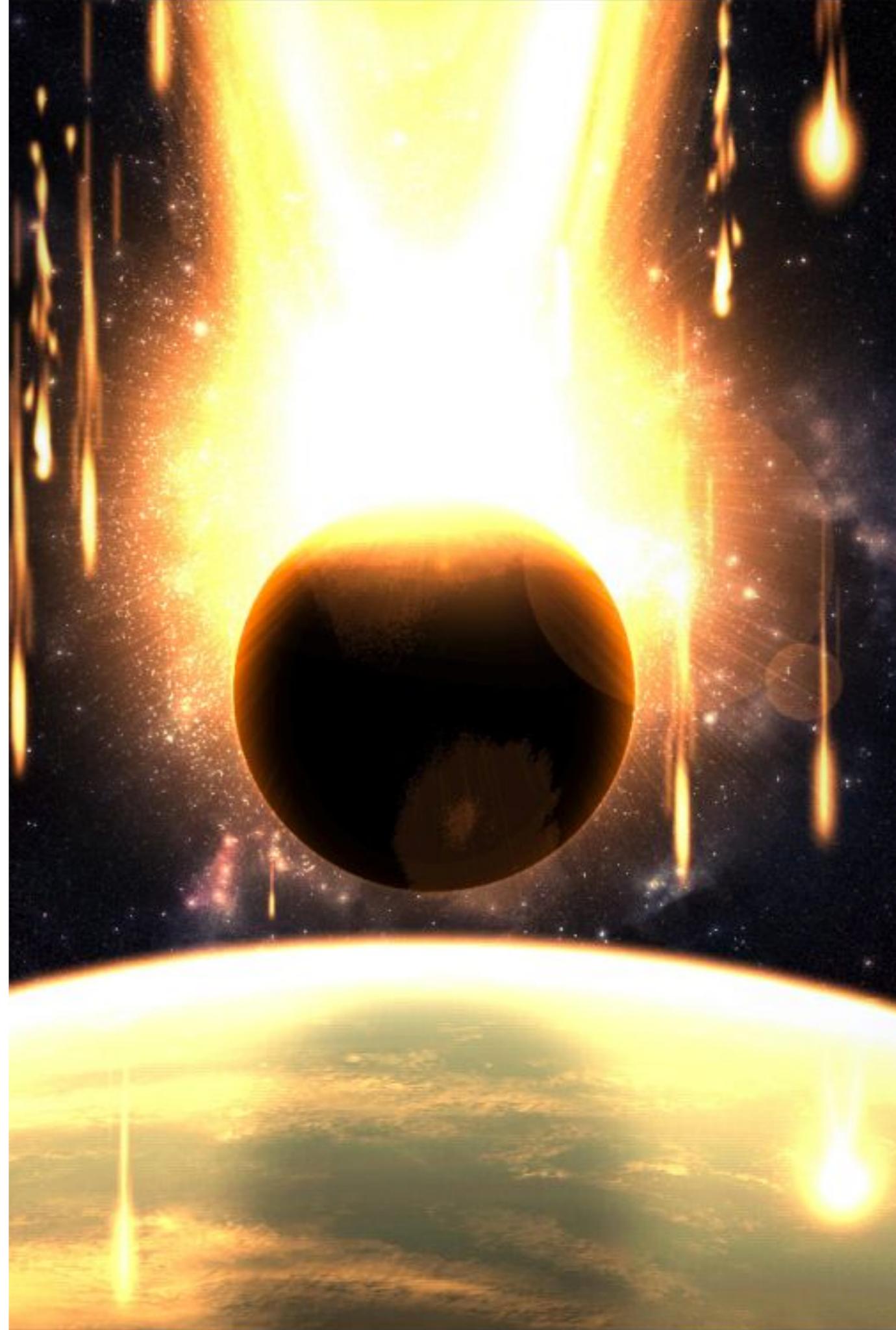
Remoção de Arquivo Hash

- Após um certo número de inserções e eliminações, o desempenho piora
- Após um certo limite, toda a organização está comprometida
- Soluções:
 - reorganizar parcial e localmente após cada eliminação
 - reorganizar completamente o arquivo
 - usar outra técnica para tratar as colisões



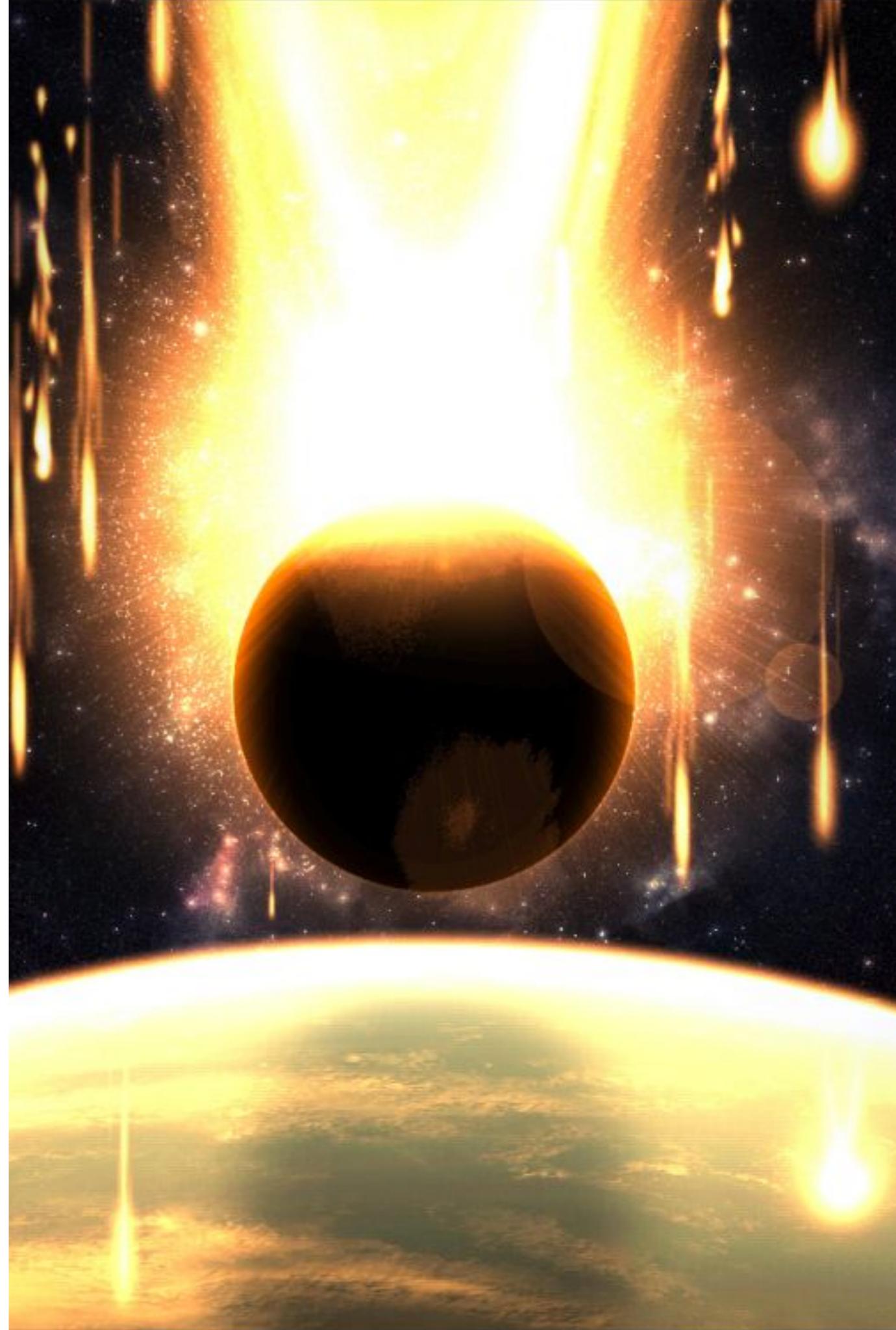
Tratamento de Colisões: abordagens

- **Endereçamento aberto ou linear:**
 - A partir da posição de colisão, procurar uma posição subsequente vaga
- **Encadeamento:**
 - Manter uma lista encadeada de registros de overflow para cada posição no espaço de endereços
- **Hashing múltiplo:**
 - Aplicar uma segunda função de hashing quando ocorrer uma colisão. Se ocorrer nova colisão, aplicar endereçamento aberto ou nova função de hashing



Tratamento de Colisões: ***Overflow Progressivo***

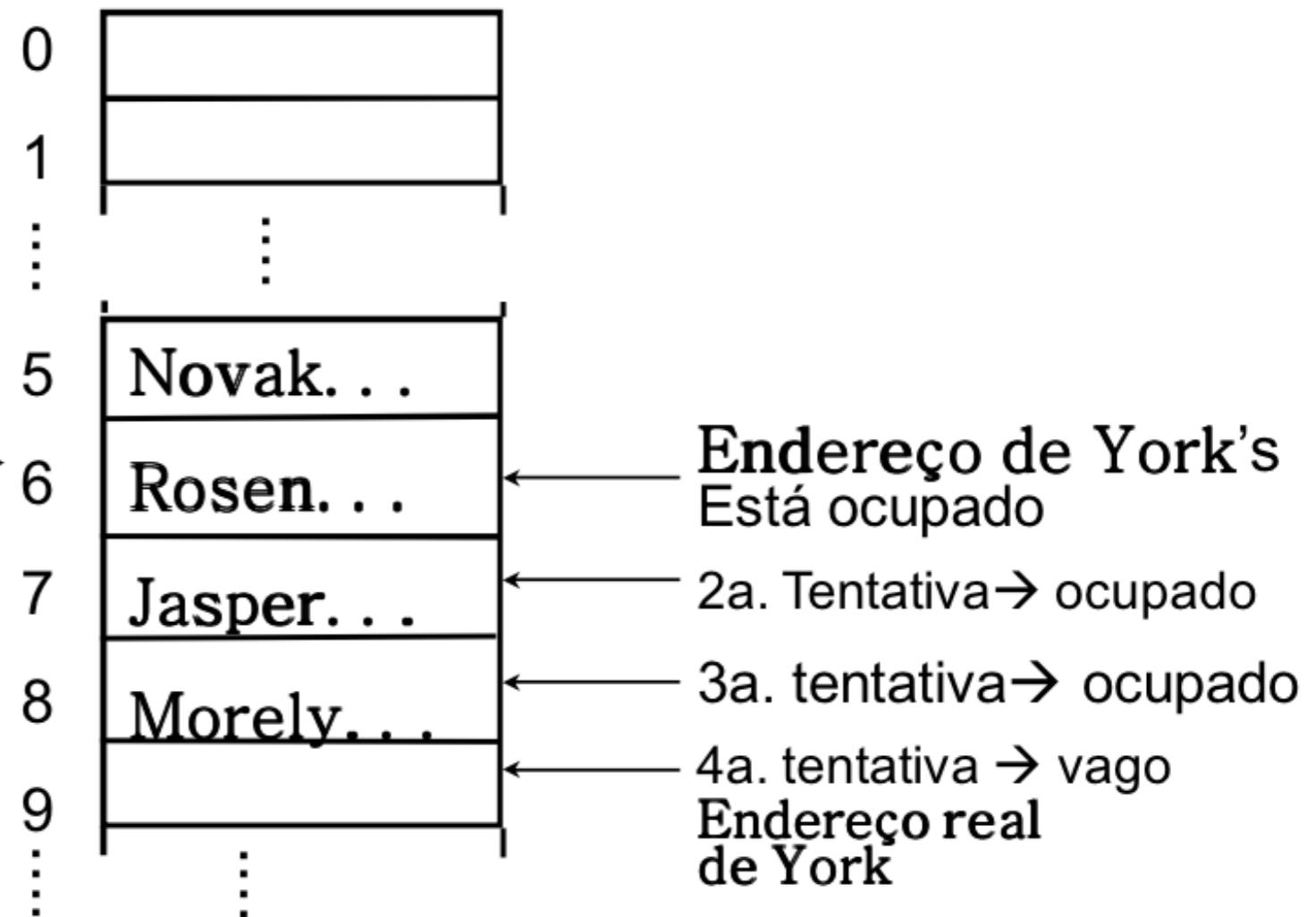
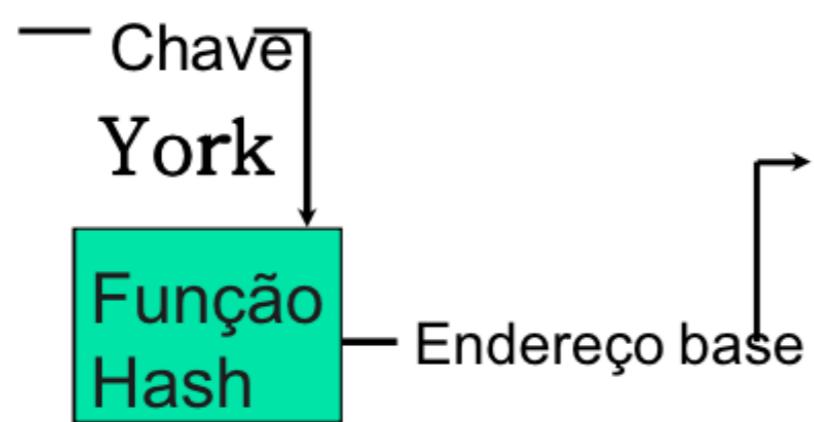
- Qualquer solução que use espalhamento precisa incorporar algum mecanismo para tratar registros que não possam ser colocados no seu endereço-base
- Existem várias maneiras de se tratar colisões. Uma técnica bastante simples mas que pode funcionar bem em muitas situações é conhecida como re-espalhamento linear ou overflow progressivo
- O tratamento de uma colisão é feito procurando-se a próxima posição vazia depois do endereço-base da chave



Tratamento de Colisões: *Overflow Progressivo*

Algoritmo de Inserção:

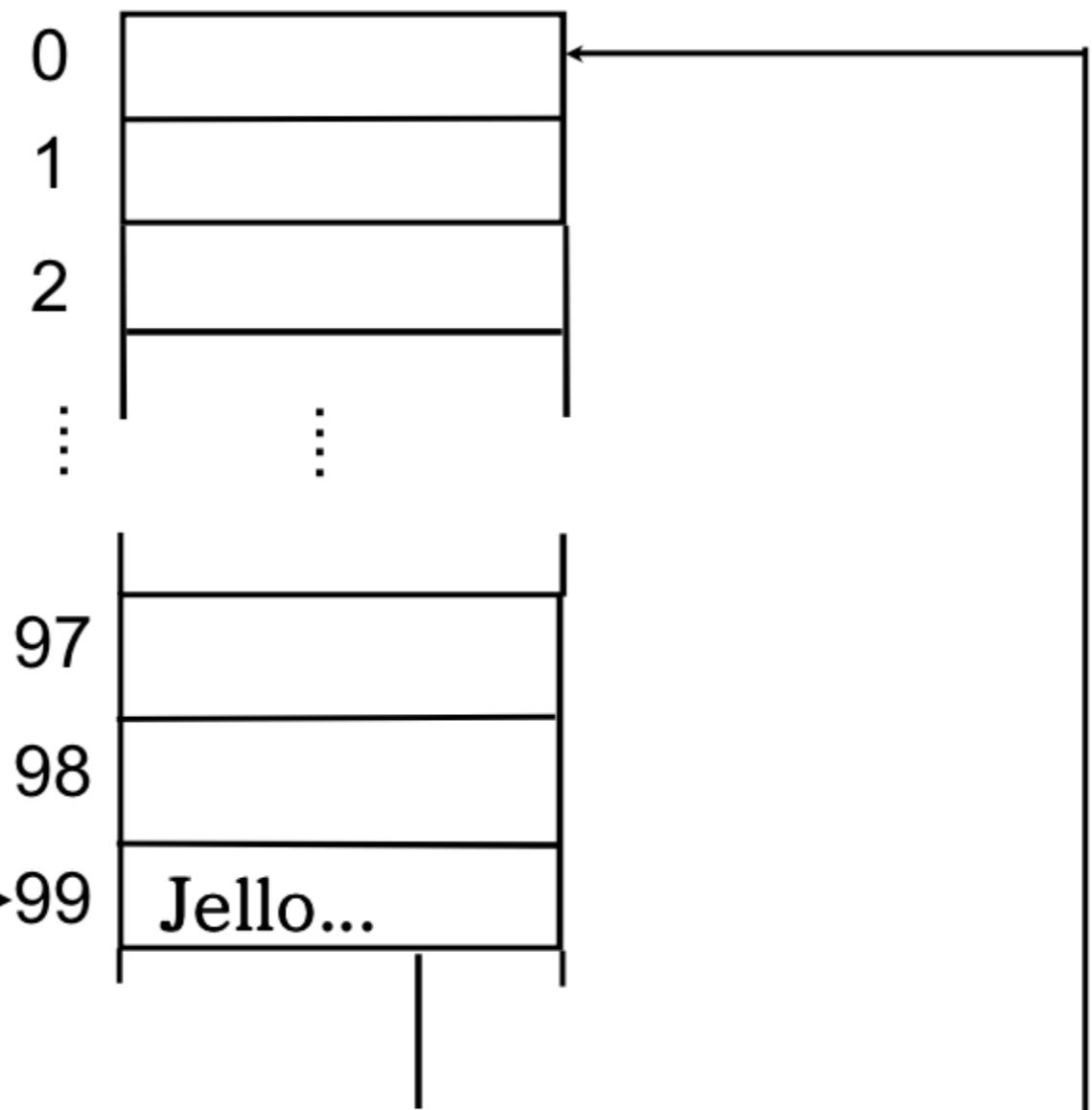
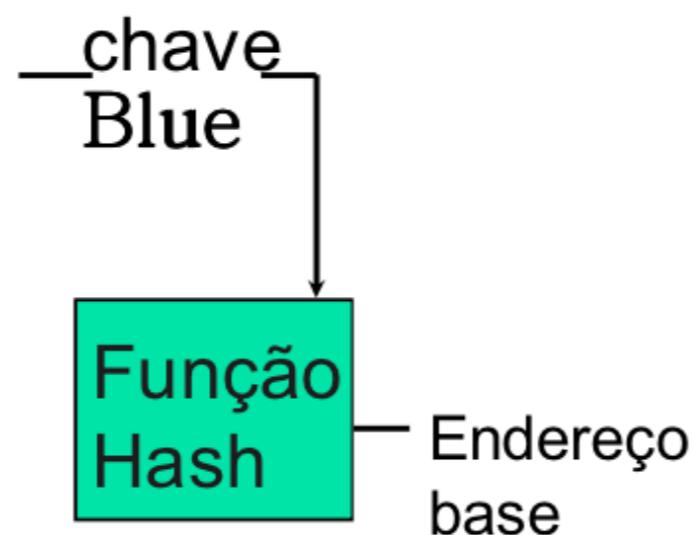
- vá ao endereço de $k=h(k)$
- se estiver livre coloque a chave ali
- se estiver ocupado, tente a próxima posição até que uma posição desocupada seja encontrada (a próx posição para a última é a posição 0)



Tratamento de Colisões: *Overflow Progressivo*

Algoritmo de Inserção:

- vá ao endereço de $k=h(k)$
- se estiver livre coloque a chave ali
- se estiver ocupado, tente a próxima posição até que uma posição desocupada seja encontrada (a próx posição para a última é a posição 0)



Tratamento de Colisões: *Overflow Progressivo*

Algoritmo de Busca:

- vá ao endereço de $k=h(k)$
 - se a chave estiver no endereço -> encontrou
 - senão tente a próxima posição até que seja encontrada: a chave ou uma posição vazia

	Endereço base da chave
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

0	DEAM
1	EVANS
2	
.	
.	
20	COLE
21	BATES
22	ADAMS



Tratamento de Colisões: *Overflow Progressivo*

PROBLEMA:

- Se ocorrerem muitas colisões, pode ser criado um agrupamento (*clustering*) de chaves em uma certa área
- Isso pode fazer com que sejam necessários muitos acessos para recuperar um certo registro

	Endereço base da chave	Tam. da pesquisa	
:			
COLE	20	1	0 DEAN
BATES	21	1	1 EVANS
ADAMS	21	2	2
DEAN	22	2	.
EVANS	20	5	.
			20 COLE
			21 BATES
			22 ADAMS

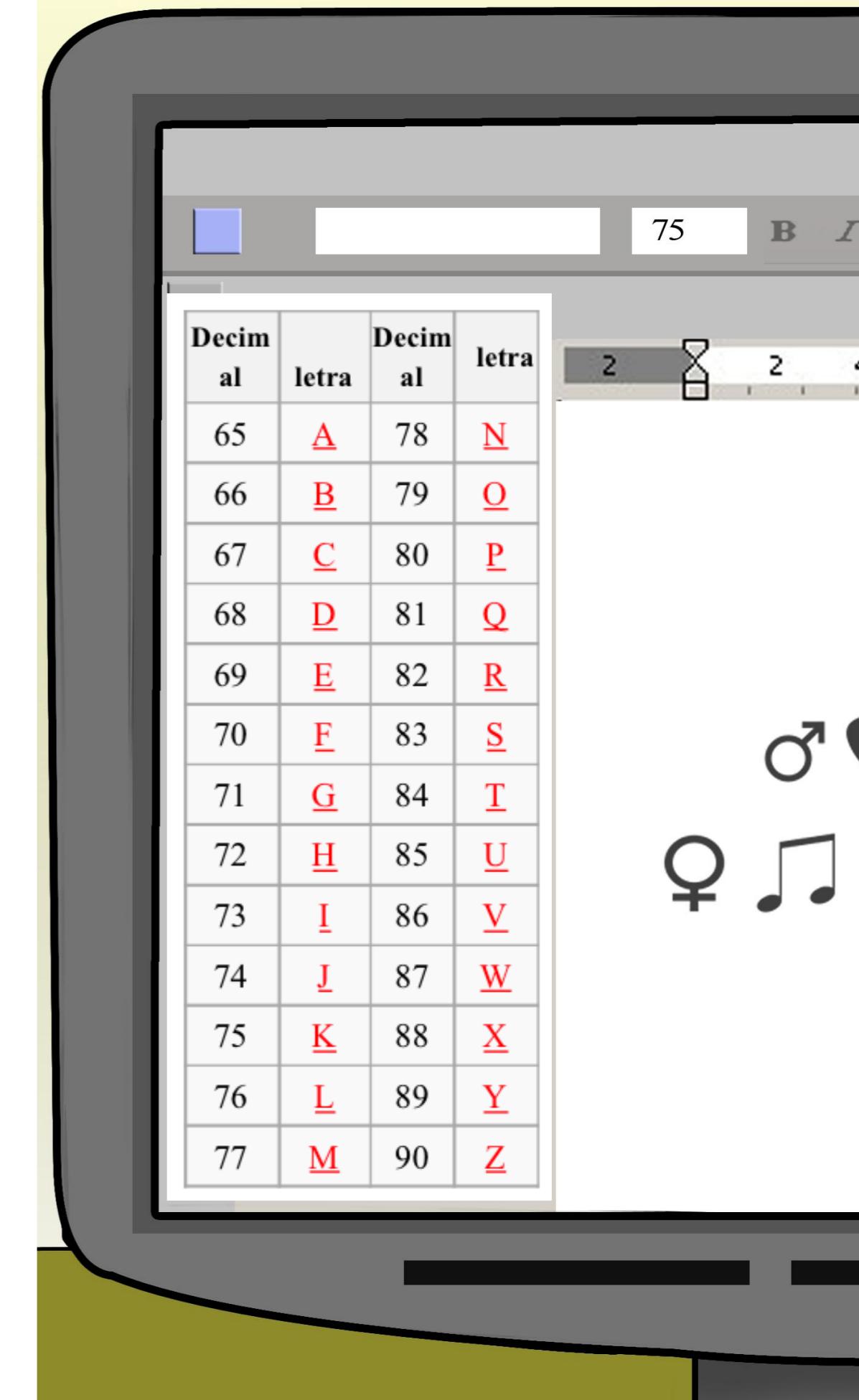
$$\text{Tamanho médio da pesquisa} = \frac{\text{Tamanho total da pesquisa}}{\text{Total de registros}} = 11/5 = 2.2$$

Atividade Prática

Considere a seguinte função *hashing* e resolva as questões utilizando *overflow* progressivo

```
public int Hashing (String Key)  
{  
    int h;  
    h = 2 * Key[0] + Key[1] + 3 * Key [2];  
  
    return h % 6;  
}
```

- A. Para quais endereços as chaves PAL e LAP serão mapeadas ? (use tabela ASCII)
- B. Dado o seguinte mapeamento (VAL,3), (LAV,1), (MAP,5), (PAT,3), (PET,1), mostre o conteúdo da tabela hash de tamanho 6, considerando que as chaves serão fornecidas nessa ordem
- C. Qual é o tamanho médio da pesquisa nesta tabela?



Tratamento de Colisões: **Espelhamento Duplo**

- Uma segunda função de espalhamento é aplicada novamente toda vez que uma colisão ocorrer. O processo é repetido até uma posição livre ser encontrada (para inserção ou busca)
 - A primeira função *hashing* determina o endereço base da chave
 - Se o endereço estiver ocupado, aplicar a segunda função *hashing* para obter um número
 - é **adicionado** ao endereço para produzir um endereço de *overflow*, até que uma vaga seja encontrada

Vantagem: tende a espalhar melhor as chaves pelos endereços

Desvantagem: os endereços podem estar muito distantes um do outro, provocando seekings adicionais.



Tratamento de Colisões: *Espelhamento Duplo*

Chave (k)	Adams	Jones	Morris	Smith
$h_1(k)$	5	6	6	5
$h_2(k)$	2	3	4	3

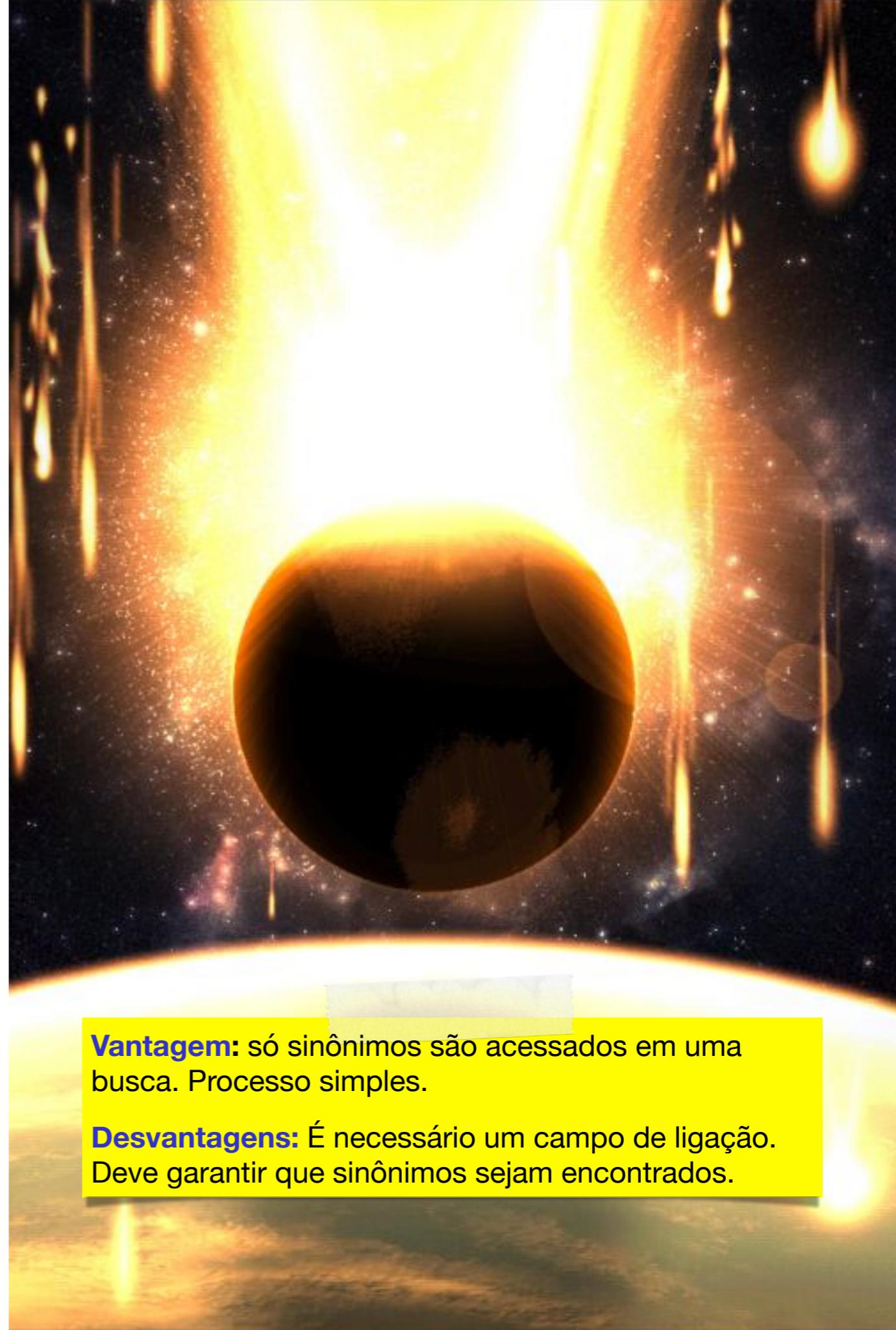
Nesse exemplo: se ocorrer a colisão, a função hash $h_2(k)$ é considerada e o novo endereço base será a soma de endereço atual ($h_1(k)$) + $h_2(k)$.

Entretanto, outras funções poderiam ser utilizadas na criação desse segundo endereço para chave que sofreu colisão

0	
1	
2	
3	
4	
5	Adams
6	Jones
7	
8	Smith
9	
10	Morris

Tratamento de Colisões: ***Overflow Progressivo Encadeado***

- Este processo é similar ao Overflow Progressivo, com a diferença de que os sinônimos são encadeados na própria área de endereçamento
- O objetivo é reduzir o comprimento da pesquisa de um registro dentro de um agrupamento de registros de mesmo endereço



Vantagem: só sinônimos são acessados em uma busca. Processo simples.

Desvantagens: É necessário um campo de ligação. Deve garantir que sinônimos sejam encontrados.

Tratamento de Colisões: *Overflow Progressivo Encadeado*

Considere o seguinte mapeamento de chaves: (ADAMS,20), (BATES,21), (COLES,20) (DEAN,21), (EVANS,24), (FLINT,20)

Overflow
progressivo

.	
.	
.	
.	
20	ADAMS
21	BATES
22	COLES
23	DEAN
24	EVANS
25	FLINT
.	
.	

Overflow progressivo
encadeado

.	
.	
.	
20	ADAMS
21	BATES
22	COLES
23	DEAN
24	EVANS
25	FLINT
.	
.	

Comprimento da
pesquisa

chave	Ende reço base	Overflow progres.	Overflow proges. encadeado
ADAMS	20	1	1
BATES	21	1	1
COLES	20	3	2
DEAN	21	3	2
EVANS	24	1	1
FLINT	20	6	3
Média		2,5	1,7

Tratamento de Colisões: *Overflow Progressivo Encadeado*

- **Problema:** tratamento especial de chaves que não podem ocupar seu endereço base pois este já está ocupado por um registro de outra lista
- Uma solução: two-pass loading:
 - Primeiro passo: Carga de todas as chaves no endereço-base;
 - Segundo passo: Carga das chaves de colisão.
- Exemplo: Suponha que o endereço de DEAN seja 22 em vez de 21. Como COLES já ocupa o endereço, nós não teremos um link começando neste endereço.

Cuidado com as remoções!!

chave	endereço
ADAMS	20
BATES	21
COLES	20
DEAN	22
EVANS	24
FLINT	20

Tabela após 1º. passo

20	ADAMS	-1
21	BATES	-1
22	DEAN	-1
23		
24	EVANS	-1
25		

Tabela após 2º. passo

20	ADAMS	23
21	BATES	-1
22	DEAN	-1
23	COLES	25
24	EVANS	-1
25	FLINT	-1

Tratamento de Colisões:

Encadeamento com Áreas de Overflow Separadas

- Aloca todos os sinônimos numa área especial de overflow (no final do arquivo, por exemplo)
- Endereços sem chaves na área primária nunca serão ocupados

Vantagem: Resolve o problema de manutenção. Processo mais rápido (requer menos processamento) do que, por exemplo, o reajuste depois de cada eliminação. Necessário, se a densidade de ocupação for maior que 1

Desvantagens: Overflow sempre requer outro acesso



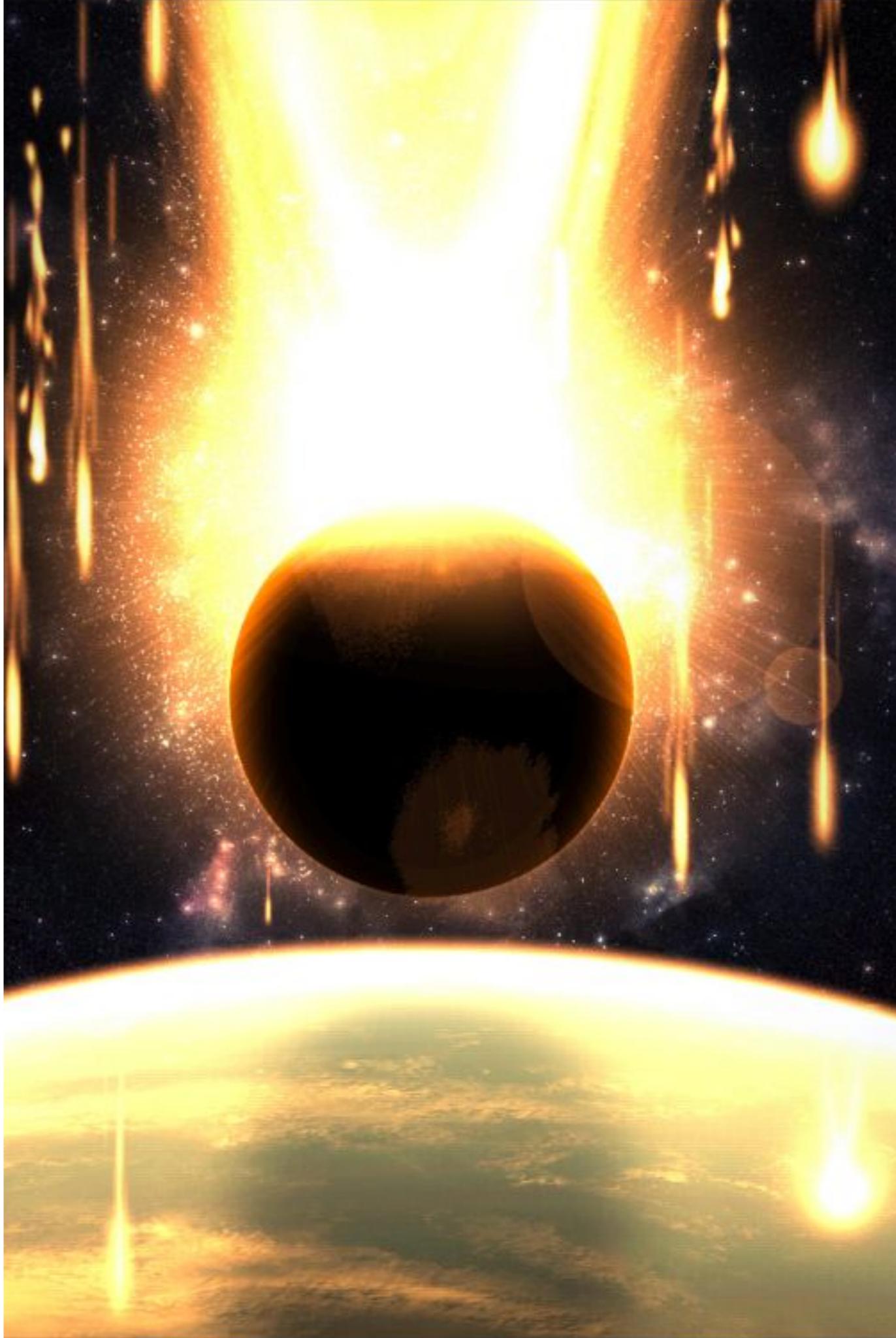
Tratamento de Colisões: *Encadeamento com Áreas de Overflow Separadas*

Considere o seguinte mapeamento de chaves: (ADAMS,20), (BATES,21), (COLES,20)
(DEAN,21), (EVANS,24), (FLINT,20)

		Área de overflow
20	ADAMS	0
21	BATES	1
22		
23		
24	EVANS	-1
25		
0	COLES	2
1	DEAN	-1
2	FLINT	-1
3		

Tratamento de Colisões: **Tabela de Índices de Endereços (Scatter Tables)**

- Utiliza uma área separada para overflow
- O arquivo hash guarda apenas as chaves e sua posição nos arquivos de dados
- O hashing gera um índice, e o arquivo relacionado é chamado scatter table



Tratamento de Colisões: *Tabela de Índices de Endereços (Scatter Tables)*

Considere o seguinte mapeamento de chaves: (ADAMS,20), (BATES,21), (COLES,20) (DEAN,21), (EVANS,24), (FLINT,20)

Índice (hash)

20	0
21	1
22	
23	
24	4
25	

Arquivo de dados

	dado	próximo
0	ADAMS	2
1	BATES	3
2	COLES	5
3	DEAN	-1
4	EVANS	-1
5	FLINT	-1

Observação: Note que o arquivo de dados pode ser organizado de várias formas diferentes: sequencial, ordenado, etc



Prof. Fernando Sambinelli
sambinelli@ifsp.edu.br



$$i = \frac{p}{4\pi r^2}$$

