

# **Tabelas de Dispersão** (*Hash Tables*)

Algoritmos e Estruturas de Dados

---

Material gentilmente cedido pelo Prof. Bruno Machiavello

Departamento de Engenharia Elétrica (ENE), Faculdade de Tecnologia (FT)

Tabela Hash

Colisão

Algoritmos de Hashing

Algoritmo de Dispersão Simples

Implementação de Tabelas Hash

Leitura sugerida: Seções 10.1 a 10.3 do livro-texto (Drozdek).

## Tabela Hash

---

## Algoritmos de Busca

- Sequencial: Pior caso de acesso =  $O(n)$ .
- Índices Simples (lista ordenada): Busca binária =  $O(\log_2 n)$ .
- Indexação usando Árvores Binárias de Busca: Pior caso de  $O(\log_2 n)$ .

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

- A ordem de inserção dos elementos não têm influência.



Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

- A ordem de inserção dos elementos não têm influência.
- O número de elementos não tem influência.

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

- A ordem de inserção dos elementos não têm influência.
- O número de elementos não tem influência.
- Não há comparações desnecessárias.

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

- A ordem de inserção dos elementos não têm influência.
- O número de elementos não tem influência.
- Não há comparações desnecessárias.
- Sempre vamos achar o que estamos buscando na primeira tentativa

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

- A ordem de inserção dos elementos não têm influência.
- O número de elementos não tem influência.
- Não há comparações desnecessárias.
- Sempre vamos achar o que estamos buscando na primeira tentativa

Mas como?

- O acesso direto implica a utilização de um vetor (tabela).

Uma estrutura que trabalhe na ordem de  $\log_m(n)$  é muito eficiente, mas o que seria melhor?

$$O(1)$$

Isto significa que:

- A ordem de inserção dos elementos não têm influência.
- O número de elementos não tem influência.
- Não há comparações desnecessárias.
- Sempre vamos achar o que estamos buscando na primeira tentativa

Mas como?

- O acesso direto implica a utilização de um vetor (tabela).
- Se cada chave precisa ser recuperada em um único acesso, a posição do elemento depende da somente da chave.

## Chave - Chave é uma ferramenta conceitual

- Uma chave deve ter uma forma padrão
- Uma chave na forma padrão é dita chave **canônica**.
- O ideal é que exista uma relação um a um entre chave e dado (registro) que representa, ou seja cada chave corresponde um único dado (registro).
  - Se a regra define chaves com letras maiúsculas, qualquer entrada dada pelo usuário é convertida para a forma canônica antes da pesquisa.

## Chave Primária

- Identifica unicamente cada registro
- Em geral, não pode ser modificada. Exemplo: matrícula, nome, CPF, ...

## Chave Secundária

- Chave que pode ser compartilhada por dois ou mais registros
- Não há garantias de unicidade. Exemplo: nome, cidade, UF, ...
- Pode ser utilizada para buscas simultâneas de várias chaves (todos os “Silva” que moram em Curitiba, por exemplo).

Função Hash também é conhecida como Função de Dispersão ou Função de Espalhamento.

- A idéia é descobrir a localização de uma chave simplesmente examinando o conteúdo da chave; para isso precisamos de uma função hash.



Função Hash também é conhecida como Função de Dispersão ou Função de Espalhamento.

- A idéia é descobrir a localização de uma chave simplesmente examinando o conteúdo da chave; para isso precisamos de uma função hash.
- Uma função Hash é uma função matematicamente definida que converte uma grande quantidade de dados (possivelmente de tamanho variável) em uma menor quantidade de dados, normalmente um número que sirva como índice para um array ou qualquer outra estrutura de dados.

Função Hash também é conhecida como Função de Dispersão ou Função de Espalhamento.

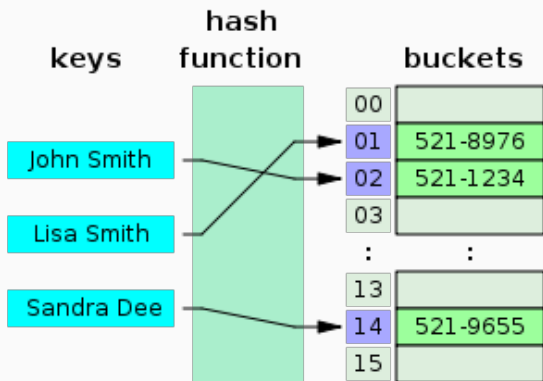
- A idéia é descobrir a localização de uma chave simplesmente examinando o conteúdo da chave; para isso precisamos de uma função hash.
- Uma função Hash é uma função matematicamente definida que converte uma grande quantidade de dados (possivelmente de tamanho variável) em uma menor quantidade de dados, normalmente um número que sirva como índice para um array ou qualquer outra estrutura de dados.
  - O endereço resultante é usado como base para armazenar e recuperar registros.

Função Hash também é conhecida como Função de Dispersão ou Função de Espalhamento.

- A idéia é descobrir a localização de uma chave simplesmente examinando o conteúdo da chave; para isso precisamos de uma função hash.
- Uma função Hash é uma função matematicamente definida que converte uma grande quantidade de dados (possivelmente de tamanho variável) em uma menor quantidade de dados, normalmente um número que sirva como índice para um array ou qualquer outra estrutura de dados.
  - O endereço resultante é usado como base para armazenar e recuperar registros.
  - O espaço do endereço é escolhido antecipadamente. Por exemplo, podemos decidir que o array terá 1000 endereços disponíveis.

Vantagens:

- Utilidade
- Velocidade

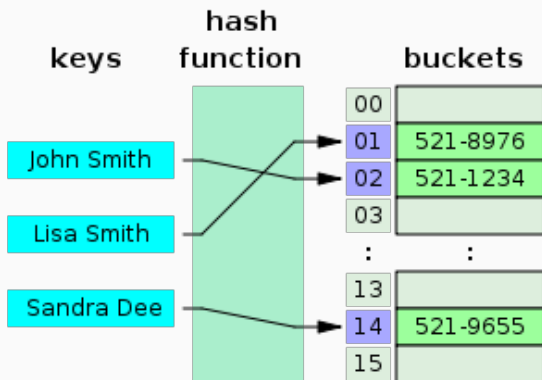


### Vantagens:

- Utilidade
- Velocidade

### Desvantagens:

- Colisões
- Não permite:
  - ordenação
  - recuperar o elemento antecessor/sucessor
  - chaves duplicadas
- Pode necessitar de redimensionamento



## Função Hash

- É uma caixa preta que produz um endereço todas as vezes que você introduz uma chave.
  - Semelhante a indexação.
- Difere de indexação em dois aspectos:
  1. O endereço gerado é aleatório: Não há uma conexão óbvia entre a chave e o endereço.
  2. Chaves diferentes podem gerar o mesmo endereço e ocasionar colisões. O ideal é gerar endereços com distribuição uniforme.

## Exemplo

Suponha que:

- Desejamos armazenar 75 registros num arquivo.
- A chave é o nome da pessoa.
- Determinamos que o array terá 1000 endereços disponíveis. Se  $U$  é o conjunto de todas as chaves possíveis, a função hash:

$$h : U \rightarrow \{0, 1, \dots, 999\}$$

k	ASCII (2 primeiras letras)	Produto	$h(k) = \text{produto} \bmod 1000$
BALL	66, 65	4290	290
LOWELL	76, 79	6004	4
TREE	84, 82	6888	888

k	ASCII (2 primeiras letras)	Produto	$h(k) = \text{produto} \bmod 1000$
BALL	66, 65	4290	290
LOWELL	76, 79	6004	4
TREE	84, 82	6888	888

RRN | FILE

000

001

⋮

004

LOWELL

⋮

290

BALL

⋮

888

TREE

⋮

999

Não existe uma relação óbvia entre a chave e o endereço;  
duas chaves diferentes podem ser enviadas para o mesmo  
endereço.



- Exemplo para o caso em questão: LOWELL, LOCK, OLIVER, e qualquer outra palavra com as duas primeiras letras sendo L ou O serão mapeadas para o mesmo endereço.
  - $h(\text{LOWELL}) = h(\text{LOCK}) = h(\text{OLIVER}) = 4$
- Estas chaves são chamadas de sinônimos.
- Evitar colisões é extremamente difícil, por isso precisamos de técnicas para isso.
- **Hashing só é eficiente se provocar poucas colisões (logo, requer grandes áreas)**

# Colisão

---

**Definição**

Dois ou mais registros geram o mesmo endereço com a aplicação da função de hash.

## Solução 1: Hashing Perfeito

- Um algoritmo de transformação que evita colisões
- A função não gera endereços repetidos.
- Suponhamos que queremos armazenar 4.000 registros em 5.000 endereços. E a chave não é diretamente uma função hash perfeita (por exemplo número em ordem).
- Hanson em 1982<sup>1</sup> mostrou que apenas 1 entre  $10^{120000}$  algoritmos evita completamente a colisão.
- Assim, não vale a pena buscar o algoritmo perfeito.

---

<sup>1</sup>Hanson, O. Design of Computer Data Files. Rockville, Computer Science Press, 1982.

## **Solução 2: Um algoritmo aceitável**

- Gera um número de colisões aceitáveis.
- Gera poucos sinônimos.

A colisão é um problema sério?

A colisão é um problema sério?

### **Problema do Aniversário**

Dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%.

A colisão é um problema sério?

### **Problema do Aniversário**

Dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%. Para 57 ou mais pessoas, a probabilidade é maior do que 99%.



A colisão é um problema sério?

### **Problema do Aniversário**

Dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%. Para 57 ou mais pessoas, a probabilidade é maior do que 99%.

Uma aplicação com 2500 chaves uniformemente distribuídas, a serem armazenadas em um vetor de tamanho  $10^6$  tem 95% de chance de colisão.

A colisão é um problema sério?

### **Problema do Aniversário**

Dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%. Para 57 ou mais pessoas, a probabilidade é maior do que 99%.

Uma aplicação com 2500 chaves uniformemente distribuídas, a serem armazenadas em um vetor de tamanho  $10^6$  tem 95% de chance de colisão.

A colisão pode ser evitada conhecendo-se todas as chaves

A colisão é um problema sério?

### **Problema do Aniversário**

Dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%. Para 57 ou mais pessoas, a probabilidade é maior do que 99%.

Uma aplicação com 2500 chaves uniformemente distribuídas, a serem armazenadas em um vetor de tamanho  $10^6$  tem 95% de chance de colisão.

A colisão pode ser evitada conhecendo-se todas as chaves → escolhe-se a função de dispersão ideal para obter a indexação perfeita.

A colisão é um problema sério?

### **Problema do Aniversário**

Dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%. Para 57 ou mais pessoas, a probabilidade é maior do que 99%.

Uma aplicação com 2500 chaves uniformemente distribuídas, a serem armazenadas em um vetor de tamanho  $10^6$  tem 95% de chance de colisão.

A colisão pode ser evitada conhecendo-se todas as chaves → escolhe-se a função de dispersão ideal para obter a indexação perfeita.

Como isto não é prático, outras técnicas são utilizadas.

*Load Factor*: razão entre o número de chaves e o tamanho da tabela

- Geralmente  $< 0.7$  (com uma boa função de dispersão)

*Load Factor*: razão entre o número de chaves e o tamanho da tabela

- Geralmente  $< 0.7$  (com uma boa função de dispersão)
- Fator de carga alto indica que a memória está sendo bem utilizada, mas implica em maior probabilidade de colisão.

*Load Factor*: razão entre o número de chaves e o tamanho da tabela

- Geralmente  $< 0.7$  (com uma boa função de dispersão)
- Fator de carga alto indica que a memória está sendo bem utilizada, mas implica em maior probabilidade de colisão.
- Fator de carga baixo implica em menor probabilidade de colisão mas indica que a memória está sendo desperdiçada.

Redução de Colisões: Há vários meios de reduzir colisões, entre eles:

- **Espalhamento de registros:** Consistem em achar uma outra função de hash que mapeie chaves em endereços de modo mais uniforme, em um dado espaço de endereços, com baixo grau de colisões.
- **Uso de memória adicional:** consistem em aumentar o espaço de endereços em relação a quantidade de registros a ser incluídos na área.
  - Por exemplo, reservar 5.000 registros ao invés de 1.000.
  - Isto leva ao desperdício de área memória.
- **Colocar mais de um registro em um único endereço:** utilização de baldes ou de listas encadeadas.

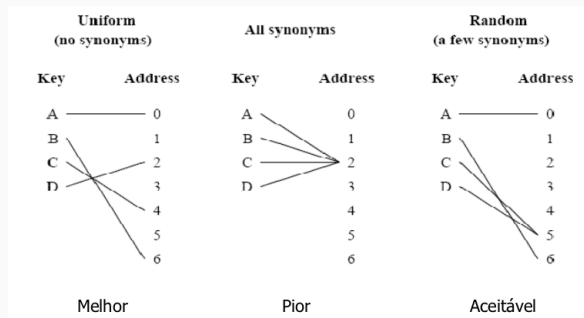


# Algoritmos de Hashing

---

# Algoritmos de Hashing

Existem 3 possibilidades:



- Distribuições uniformes são extremamente raras.
- Distribuições aleatórias são aceitáveis e mais fáceis de serem obtidas.

- Uma distribuição uniforme é praticamente impossível e está fora de questão.
- Ideal:
  - Existência de mapeamento que garantisse distribuição melhor que a aleatória para todos os casos.
  - De um modo geral, não há! (depende das chaves)
  - A escolha de um mapeamento adequado envolve:
    - Consideração inteligente das chaves.
    - Alguma experimentação.
- Geração aleatória de endereços:
  - Usado quando a estrutura natural das chaves não gera bom espalhamento.
  - Faz transformação que produz espalhamento.

- Tentando uma distribuição melhor do que a aleatória, tirando vantagem da ordem natural entre as chaves.
- Buscar padrões nas chaves. Tais padrões podem levar a uma distribuição natural:
  - Por exemplo: chaves são anos entre 1970 e 2000.
  - $h(ano) = (ano - 1970) \bmod (2000 - 1970 + 1)$ .
  - $h(1970) = 0, h(1971) = 1, \dots, h(2000) = 30$

- Juntar parte das chaves:
  - Envolve extrair dígitos de partes da chave e juntá-los (como no exemplo anterior).
  - Destrói o padrão original da chave, mas em alguns casos pode preservar componentes que naturalmente espalham.
- Usar um número primo para modular (dividir) a chave:
  - A modulação preserva sequências naturais.
  - Pesquisas tem mostrado que a utilização de um número primo diminui as chances de colisão.

## **Geração Aleatória de Endereços Meio-Quadrado:**

- Considere a chave como um grande número.
- Eleve a chave ao quadrado.
- Extraia os bits do meio do número obtido: a quantidade necessária para gerar o endereço no espaço disponível.
- Exemplo:
  - Desejamos gerar endereços entre 0 e 99.
  - Chave = 453.
  - $453^2 = 205209$ .
  - Extraíndo o meio = 52.

## Transformação de Base

- Converte o número para outra base.
- Toma os dígitos nessa base e modula pelo endereço máximo.

$$e = \textit{Numero} - b \mod \textit{max\_espaco}$$

- Exemplo:
  - Endereços de 0 a 99.
  - 453 na base 10 = 382 na base 11

$$e = 382 \mod 99 = 85$$

- Qualidade desejável numa função de hash: espalhamento uniforme das chaves no espaço de endereços.
- Um hashing com os passos abaixo “randomiza” bem os endereços:
  - 1) Representar a chave na forma numérica.
  - 2) Dividir a chave em partes e somá-las (se necessário limitar o somatório).
  - 3) Dividir por um número primo e usar o resto da divisão como endereço.



# Algoritmo de Hashing Simples: Operacionalização

## 1) Representar a chave na forma numérica

- Se a chave é um número, então ok.
- Se é uma string, deixe-a como está: considere a string como um vetor ASCII (código) ou UNICODE

```
LOWELL = | L | O | W | E | L | L | | | | | | |  
ASCII code: 76 79 87 69 76 76 32 32 32 32 32 32
```

- Neste algoritmo usamos a chave toda, ao invés de apenas os dois primeiros caracteres.
- Usando mais partes da chaves aumentamos a probabilidade de produzirmos endereços diferentes.
- O processamento extra necessário para isso é insignificante quando comparado ao potencial de melhora na performance.

2) Partir a chave e somar as partes.

- Ex: tome a string de dois em dois bytes e faça a soma deles, gerando s.

```
7679|8769|7676|3232|3232|3232|
```

```
7679+8769+7676+3232+3232+3232 = 33,820
```

```
33,820 mod 19937 = 13,883
```

- A divisão por um número primo geralmente produz uma distribuição mais aleatória do que a transformação por um número não primo (logo 19937 é escolhido).

- 3) Dividir  $s$  pelo tamanho do espaço de endereços (preferencialmente um número primo) e usar o resto como o endereço “é”:

$$e = s \mod end\_max$$

- O espaço de endereços vai de 0 a  $end\_max - 1$ .

Se  $end\_max = 101$  (0 a 100) (Para um array com 75 registros,  $X = 101$  é uma boa escolha, que preencheria 74.2% do arquivo ( $\frac{75}{101} = 0.742$ )).

$$e = 13883 \mod 101 = 46$$

- A chave LOWELL ficaria no reg 46.
- O objetivo deste passo é reduzir o tamanho do número produzido no passo 2, de modo que fique entre o intervalo de endereços dos registros do arquivo.

# Implementação de Tabelas Hash

---

## Onde Armazenar os Sinônimos (colisões) ?

### **Na mesma área:**

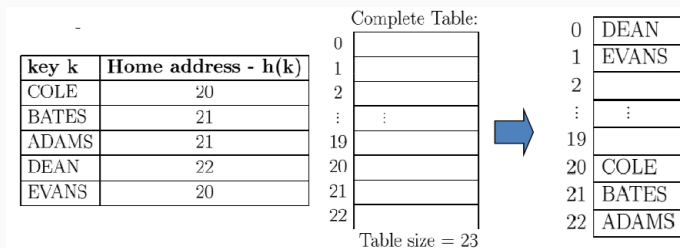
- 1) Inspeção linear.
- 2) Overflow (balde).

### **Em área separada:**

- 1) Encadeamento

# Resolução de Colisão por Inspeção Linear

- 1) Ocorrendo colisão, a tabela é pesquisada até encontrar um endereço vazio, onde o registro é colocado.
- 2) Se atingir o fim, continua a inspeção no início da tabela.
- 3) A vantagem é que todos os registros ficam na mesma tabela.
- 4) Inspeção linear é um dos métodos ditos de endereçamento aberto.



**Vantagem:** Simplicidade.

**Desvantagem:** Se há muitas colisões, clusters de registros podem se formar.

Pesquisando por uma chave:

- Vá para o endereço da chave  $k : h(k)$ .
- Se encontrou, fim.
- Senão continue a inspeção na próxima posição até que a chave seja encontrada ou um espaço vazio seja encontrado.
- Exemplo:
  - Uma busca por “EVANS” testa 20, 21, 22, 0, e 1, encontrando o registro na posição 1.
  - Uma busca por “MOURA”, se  $h(MOURA) = 22$ , testa 22, 0, 1, e 2, onde conclui que “MOURA” não está na tabela.
  - Uma busca por “SMITH”, se  $h(SMITH) = 19$ , testa 19, e conclui que “SMITH” não está na tabela.

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

Média do Comprimento de Busca, na Inspeção Linear:

$$\text{média} = \frac{c_1 + c_2 + \dots + c_n}{r}$$

Onde:


- $c_i$  = número de acessos para achar o  $i$ -ésimo registro.
- $r$  = número de registro armazenados.



- Calcularemos a média do comprimento de busca do exemplo anterior:

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

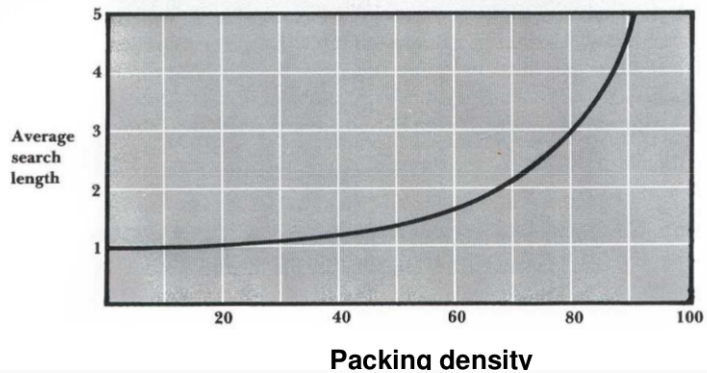
Número de seeks  
para acessar o registro.



key	Search Length
COLE	1
BATES	1
ADAMS	2
DEAN	2
EVANS	5

Média do comprimento de busca =  $\frac{1+1+2+2+5}{5} = 2.2$

- Se não existir colisões a média do comprimento de busca é 1.
- Por outro lado se tiver muitas colisões o comprimento de busca aumenta significativamente.
- Estudos mostram que, à medida que a densidade de carga aumenta, o comprimento de busca da inspeção linear aumenta exponencialmente.



### Hashing com Baldes (Buckets)

- É uma variação das tabelas hash onde mais de um registro/chave é armazenado por endereço.
- **Balde:** bloco de registros correspondendo a um endereço na tabela hash.
- A função hash gera um endereço do balde.

## Balde

- Bloco de registros associado a um endereço.
- Se Balde = 3 registros, temos que até 3 registros podem ser armazenados no mesmo endereço.

.			
.			
.			
30	Verde ... 30	Salão ... 30	
31			
32	Janio ... 32		
33	Reis ... 33	Terra ... 33	Marco ... 33
34			
35			
.	.		
.	.		
.	.		

## Balde com Inspeção Linear

- Pode-se usar para armazenar os registros sinônimos que excedam a capacidade do balde.
- Também na busca de um dado registro:
  - Se o registro não está no balde, procure-o nos baldes subsequentes.

.			
.			
.			
30	Verde ... 30	Salão ... 30	
31			
32	Janio ... 32		
33	Reis ... 33	Terra ... 33	Marco ... 33
34	Torres ... 33		
35			
.	.		
.	.		
.	.		



## Área de Overflow (Baldes): Efeito no Desempenho

	Tamanho do Balde				
Packing density (%)	1	2	5	10	100
10	4,8	0,6	0,0	0,0	0,0
20	9,4	2,2	0,1	0,0	0,0
30	13,6	4,5	0,4	0,0	0,0
40	17,6	7,3	1,1	0,1	0,0
50	21,3	10,4	2,5	0,4	0,0
60	24,8	13,7	4,5	1,3	0,0
70	28,1	17,0	7,1	2,9	0,0
75	29,6	18,7	8,6	4,0	0,0
80	31,2	20,4	10,3	5,3	0,1
90	34,1	23,8	13,8	8,6	0,8
100	36,8	27,1	17,6	12,5	4,0

**Tabela 1:** Quantidade de sinônimos (porcentagem de registros)

## **Qual o Tamanho dos Baldes?**

- Não existe uma resposta simples para isso, porque depende de várias características do sistema, incluindo o tamanho dos buffers que o sistema operacional pode gerenciar.
- Como hashing quase sempre envolve recuperar um registro por busca, qualquer tempo de transmissão extra, fruto do uso de baldes extremamente grandes, é essencialmente perdido.



- O tamanho lógico da tabela hash deve ser definida antes que ela seja preenchida com os registros/chaves, e deve permanecer fixa enquanto a mesma função hash for utilizada.

## Estrutura do Balde

- Deve ter um contador indicando o número de registros armazenados no balde.
- Um balde de tamanho 3 armazenando 2 registros:

2	JONES	////////.../	ARNSWORTH
---	-------	--------------	-----------

## Inicializando a Tabela Hash

- Decidir o tamanho lógico (número de endereços disponíveis) e o número de baldes por endereço.
- Criar uma tabela de baldes vazio antes de armazenar os registros.

0	////////.../	////////.../	////////.../
---	--------------	--------------	--------------

# Eliminação de Registros

Eliminações em tabelas de dispersão devem ser feitas com cuidado:

Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Tabela Hash usando  
overflow progressivo:

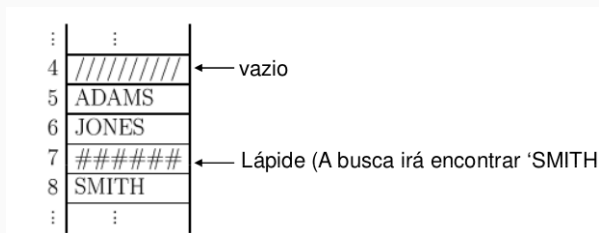
⋮	⋮
4	////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

Exclusão de 'MORRIS':

⋮	⋮	
4	////	← vazio
5	ADAMS	
6	JONES	
7	////	← vazio (Erro! Não pôde encontrar 'SMITH')
8	SMITH	
⋮	⋮	

A busca por 'SMITH' iria para a posição 5, e quando chegasse em 7 concluiria que 'SMITH' não está no arquivo.

**Solução:** usar lápides (tombstones), ou seja, substituir registros eliminados por uma marca indicando que havia um registro naquela posição.



- A busca deve continuar quando encontra uma lápide, mas deve parar quando uma posição vazia é encontrada.
- Somente insira uma lápide quando o próximo registro está ocupado ou é uma lápide. Se o próximo registro está vazio, marque o registro eliminado como vazio

### Observação

Inserções devem ser modificadas para lidar com lápides. Se uma posição vazia ou uma lápide for encontrada, o novo registro deve ser inserido naquela posição.

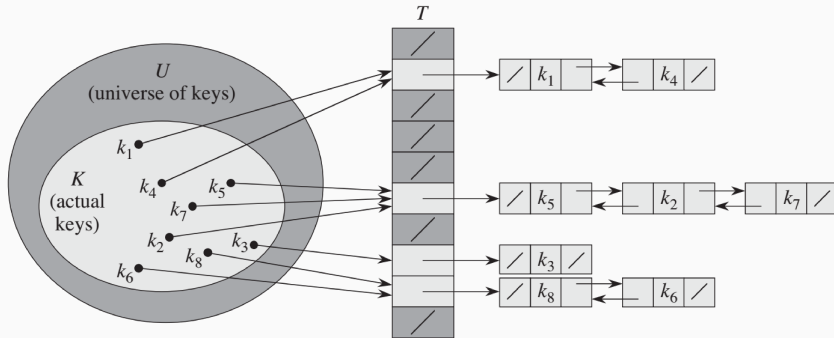
**Problema:** A presença de muitas lápides aumenta o tamanho da busca.

- Soluções para o problema de deteriorização do tamanho médio da busca:
  - Reorganização local: o algoritmo de exclusão pode tentar mover (puxar) o(s) registro(s) a frente da lápide para seu endereço original.
  - Fazer uma reorganização completa do arquivo.
  - Usar outras técnicas de resolução de colisão.

# Resolução de Colisão por Encadeamento em Área Separada

- Temos duas áreas:
  - Área de endereçamento/primária (N endereços).
  - Área de overflow (sinônimos).
- Uma lista encadeada dos sinônimos inicia nos endereços naturais da área primária, e continua na área de overflow (separada).

Key	Home	primary data area		overflow area	
ADAMS	20	20	ADAMS 0	0	COLES 2
BATES	21	21	BATES 1	1	1 DEAN -1
COLES	20	22			2 FLINT -1
DEAN	21	23			3
EVANS	24	24	EVANS -1		
FLINT	20	25			⋮



### No Caso com Baldes

Quando um balde lota, seus registros sinônimos são solocados em baldes encadeados na área de overflow.

Implemente, em C++ uma função hash como a descrita a partir do slide 25.