

UFPI - CCN - DC  
Ciência da Computação  
Estrutura de Dados

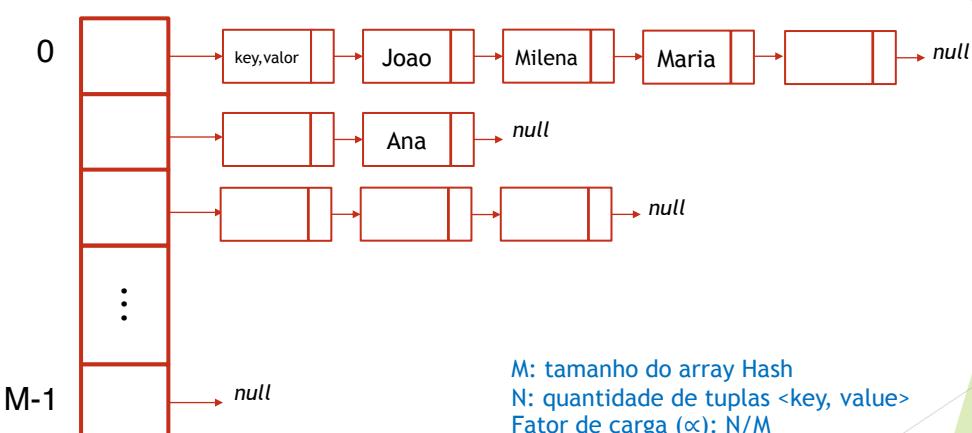
# Hashing (Implementação)



Prof. Raimundo Moura  
rsm@ufpi.edu.br

1

## Tabela Hash



M: tamanho do array Hash  
N: quantidade de tuplas <key, value>  
Fator de carga ( $\alpha$ ):  $N/M$

2

1

UFPI - CCN - DC  
Ciência da Computação  
Estrutura de Dados

## Hashing



Prof. Raimundo Moura  
rsm@ufpi.edu.br

3

### Ideias preliminares: exemplo 1

- Imagine um pequeno país (bem menos que 100 mil cidadãos) onde os números de CPF têm apenas 5 dígitos decimais. Considere a tabela que leva CPFs em nomes:

Chave	Valor associado
01555	Ronaldo
01567	Pelé
...	...
80114	Maradona
80320	Zico
95222	Romário

4

2

## Ideias preliminares: exemplo 1

► Como armazenar a tabela?

- Vetor de 100 mil posições, usando a própria chave como índice do vetor!
- O vetor é conhecido como “*tabela hash*”
- Terá muitas posições vagas (desperdício de espaço)
- A busca (*get*) e a inserção (*put*) serão extremamente rápidas

5

## Ideias preliminares: exemplo 2

► Imagine uma lista ligada cujas chaves são nomes de pessoas. Suponha que a lista está em ordem alfabética:

Chave	Valor associado
Antonio Silva	8536152
Arthur Costa	7210629
Bruno Carvalho	8536339
...	...
Vitor Sales	8535922
Wellington Lima	5992240
Yan Ferreira	8536023

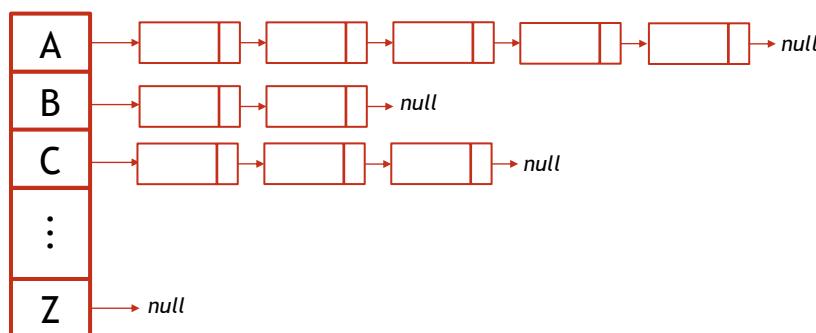
6

## Ideias preliminares: exemplo 2

- ▶ Para acelerar as buscas, divida a lista em 26 pedaços: os nomes que começam com "A", os que começam com "B", etc.

7

## Ideias preliminares: exemplo 2



- O vetor de 26 posições é a “*tabela hash*”
- Cada posição do vetor aponta para o começo de uma das listas

8

## Exercícios 1

- ▶ Qual a diferença entre o exemplo dos CPFs e a implementação [BinarySearchST](#) já discutida?
- ▶ Qual a diferença entre o exemplo das listas ligadas de nomes e a implementação [SequentialSearchST](#) já discutida?

9

## Funções de *Hashing*

- ▶ Uma *tabela de dispersão* ou *tabela hash (hash table)* é um vetor onde cada uma de suas posições armazena zero, uma, ou mais chaves (e valores associados)
  - Parâmetros importantes:
    - M: número de posições na tabela *hash*
    - N: número de chaves da tabela de símbolos
    - $\alpha = N/M$ : fator de carga (*load factor*)

10

## Funções de Hashing

► **Função de espalhamento** ou **função de hashing (hash function)**: transforma cada chave em um índice da tabela de hash.

- A **função de hashing** responde a pergunta “Em qual posição da *tabela de hash* devo colocar esta chave?”
- A **função de hashing** *espalha* as chaves pela *tabela de hash*.

11

## Funções de Hashing

➤ A **função de hashing** associa um *valor hash (hash value)*, entre 0 e  $M-1$ , a cada chave

► No exemplo dos CPFs temos  $\alpha < 1$  e a **função de hashing** é a identidade.

► No exemplo dos nomes de pessoas temos  $\alpha > 1$  e a **função de hashing** é *nome.charAt(0)*

12

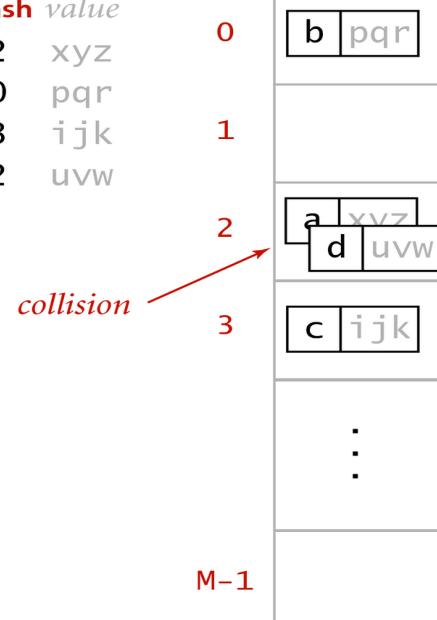
## Funções de Hashing

► A função de hashing produz uma **colisão** quando duas chaves diferentes têm o mesmo **valor hash** e, portanto, são levadas à mesma posição da *tabela de hash*.

13

## Funções de Hashing: Problema de Colisões

key	hash	value
a	2	xyz
b	0	pqr
c	3	ijk
d	2	uvw



Hashing: the crux of the problem

14

## Funções de *Hashing*: Exemplo

- ▶ Chaves são números de identificação (7 dígitos) de estudantes da universidade e  $M$  vale 100.
- Uma possível função de hashing: 2 primeiros dígitos da chave.
- Outra possibilidade: 2 dígitos do meio.
- Outra possibilidade: 2 últimos dígitos.
- ▶ Qual dessas espalha melhor as chaves pelo intervalo 0 . . . 99?

Chave
9049143
9026483
9016346
9040184
...
9000736
9035841
9129130
9041965

15

## Outro Exemplo

- ▶ Função de hashing que leva qualquer número de CPF brasileiro (que tem nove dígitos) no correspondente dígito verificador (um número entre 0 e 99). (Por exemplo, leva 111444777 em 35).
- O dígito verificador de um CPF depende de todos os dígitos do CPF.

16

## Considerações

► **Ideal:** a *função de hashing* deveria usar *todos* os dígitos da chave; assim, chaves ligeiramente diferentes serão levadas em números muito diferentes.

17

## Considerações

► Escolha  $M$  e a *função de hashing* de modo a:

- diminuir o número de colisões;
- *espalhar bem* as chaves pelo intervalo  $0 \dots M-1$ .

18

UFPI - CCN - DC  
Ciência da Computação  
Estrutura de Dados

## Hashing: uso real



Prof. Raimundo Moura  
rsm@ufpi.edu.br

19

### *Função de Hashing*

- ▶ Que *funções de hashing* são usadas na prática?
- ▶ Se as chaves são inteiros positivos, podemos usar a **função modular** (resto da divisão por  $M$ ):

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

20

## Função de Hashing Modular

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

OBS: Em *hashing modular* é bom que M seja primo (por algum motivo não óbvio)

21

## Função de Hashing

- 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;
```

22

## *Função de Hashing*

- ▶ No lugar do multiplicador 31, poderia usar qualquer outro inteiro R, de **preferência primo**, mas suficientemente pequeno para que os cálculos não produzam *overflow*.

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

23

## *Os Métodos hashCode de Java*

- ▶ Em Java, todo tipo-de-dados tem uma método padrão **hashCode()** que produz um inteiro entre  $-2^{31}$  e  $2^{31}-1$  (aproximadamente 2 bilhões negativos a 2 bilhões positivos).

```
String s = StdIn.readString();
int h = s.hashCode();
```

24

## *Os Métodos hashCode de Java*

- ▶ Para converter o *hashCode()* em um número entre 0 e  $M-1$ , tome o resto da divisão por  $M$ . Antes, é melhor desprezar o bit mais significativo para evitar que % lide com números negativos e produza um resultado negativo:

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

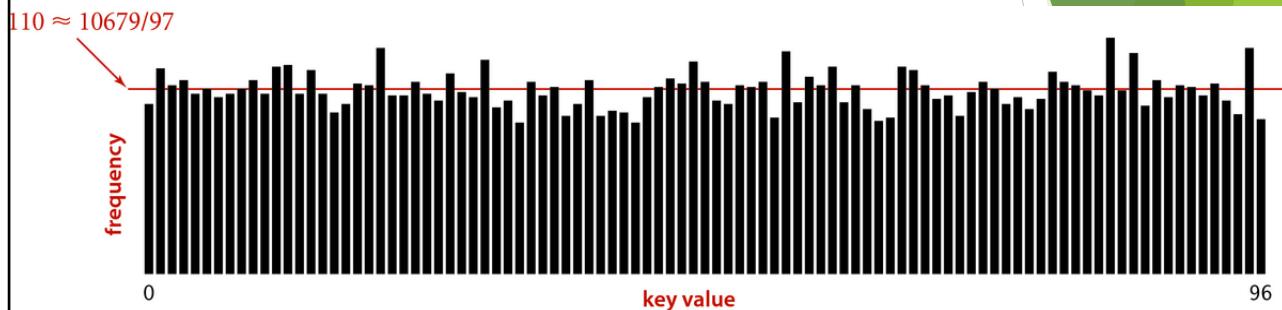
25

## *Os Métodos hashCode de Java*

- ▶ Valores *hash* calculados a partir do *hashCode()* padrão para o conjunto de palavras (excluídas as repetidas) em tale.txt, com  $M = 97$ .

26

**Histograma:** cada barra dá o número de palavras que têm o *valor hash* indicado na ordenada.



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys,  $M = 97$ )

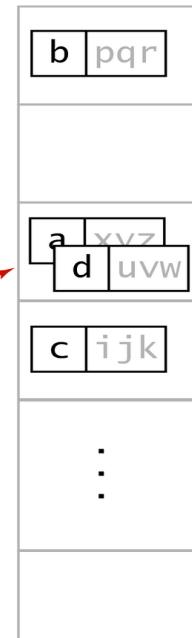
OBS: O histograma sugere que a função espalha bem as palavras.

27

Como resolver o problema de colisões?

key	hash	value
a	2	xyz
b	0	pqr
c	3	ijk
d	2	uvw

collision



Hashing: the crux of the problem

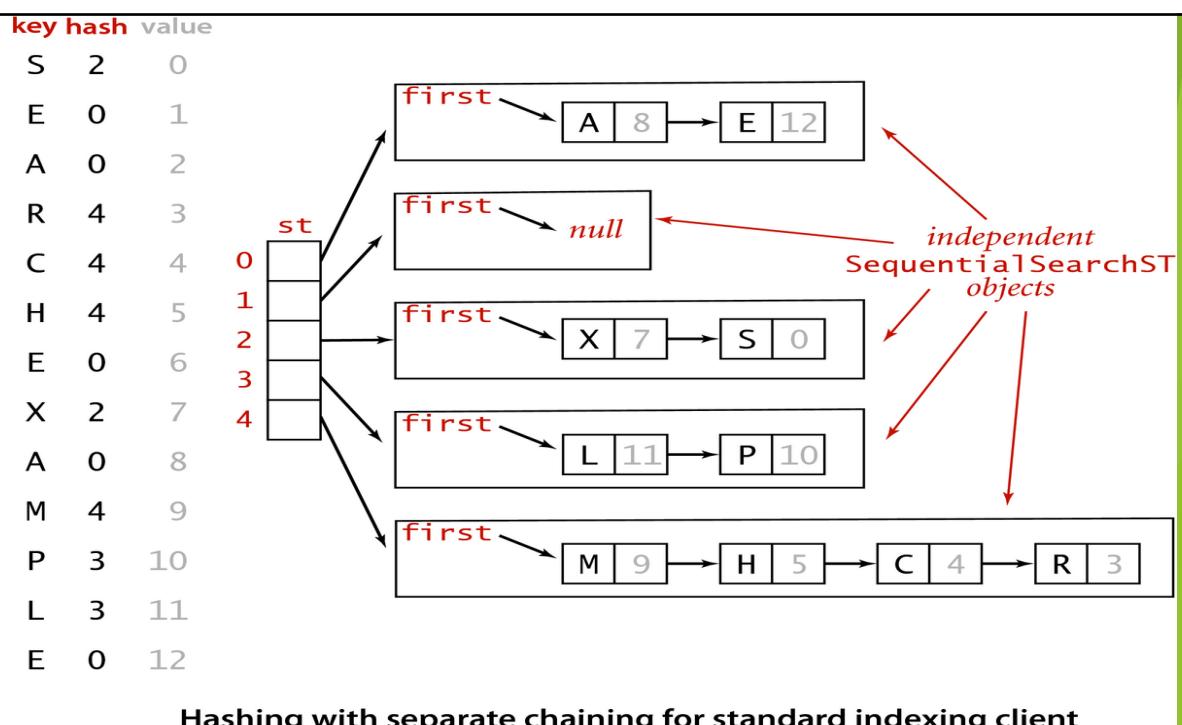
28

## Como resolver o problema de colisões?

### ► Resolução de colisões por encadeamento (*separate chaining*):

- $M$  listas ligadas, cada uma implementa uma tabela de símbolos.

29



30

## Hashing com encadeamento

- ▶ Em geral,  $N > M$  e portanto  $\alpha > 1$
- ▶ Classe **SeparateChainingHashST** (usa SequentialSearchST)

31

UFPI - CCN - DC  
Ciência da Computação  
Estrutura de Dados

## Hashing: Exercícios



Prof. Raimundo Moura  
[rsm@ufpi.edu.br](mailto:rsm@ufpi.edu.br)

32

## Exercícios

1. Elimine as palavras repetidas de `tale.txt`. Depois, faça um histograma como o mostrado em sala, mas usando  $M = 100$ .
2. Faça um histograma para as palavras de `tale.txt` (sem repetições), com  $M = 97$ , mas alguma *função de hashing* diferente da usada acima.

33

## Exercícios

3. Repita os experimentos com o livro `tale.txt` usando valores de  $M$  diferentes de 97 (tente valores que são potência de 2, por exemplo).
4. Acrescente um método `delete()` à classe **SeparateChainingHashST**. O seu método deve ser ansioso: ele deve remover o chave solicitada imediatamente (e não simplesmente marcá-la com `null` para remoção posterior).

34

## Exercícios

5. Podemos ter  $\alpha < 1$  numa *tabela de hash com encadeamento*?
6. Insira as chaves E A S Y Q U T I O N, nessa ordem, usando *hashing* com encadeamento, em uma tabela com  $M = 5$  listas. Use a função de *hashing*  $11^k \% M$  para transformar a  $k$ -ésima letra do alfabeto em um índice da *tabela de hash*.

35

## Exercícios

7. É fácil implementar operações como *min()*, *max()*, *floor()* e *ceiling()* em tabelas de símbolos implementadas com *tabelas de hash*?

36