



# Unidade 8 – Hashing



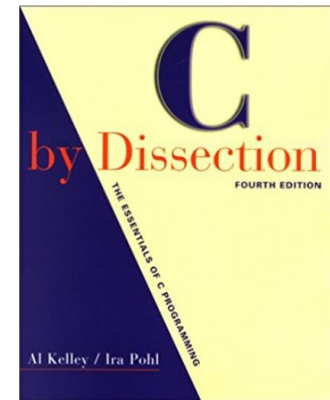
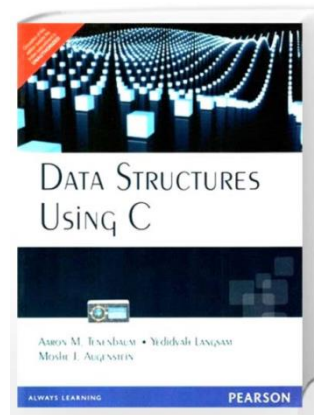
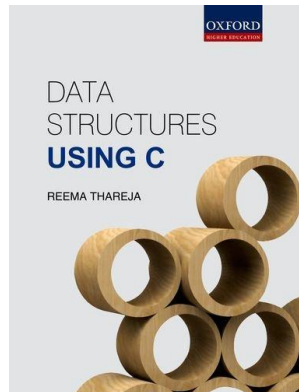
Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUVSP  
[aparecidovfreitas@gmail.com](mailto:aparecidovfreitas@gmail.com)





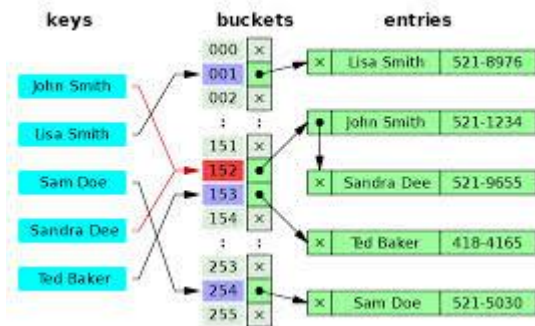
# Bibliografia

- Data Structures using C – Oxford University Press – 2014
- Data Structures Using C – A. Tenenbaum, M. Augensem, Y. Langsam, Pearson 1995
- C By Dissection – Kelley, Pohh – Third Edition – Addison Wesley





## Tabelas de Dispersão (Tabelas HASH) e Funções Hash





# Tabelas de Dispersão

- Também chamadas de Tabelas Hash.
- Quando bem projetadas, podem ser usadas para se buscar um dado em uma tabela em tempo constante:  **$O(1)$**
- O preço que se paga por isso é o uso de um pouco mais memória.
- Implementam os arrays associativos ou dicionários (mapeamentos).





# Visão Geral

- Considere uma pequena escola com cerca de **80** estudantes, onde cada estudante é identificado por uma código de matrícula de **2** dígitos.
- Por exemplo, o estudante Paulo de Souza Alves tem o código de matrícula **55**.





# Visão Geral

▣ Suponha que para cada estudante são armazenados registros com os seguintes dados:

- ✓ Código de matrícula: 2 bytes
- ✓ Nome do estudante: 50 bytes
- ✓ Endereço do estudante: 70 bytes
- ✓ Fone: 10 bytes

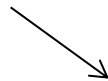


**Tamanho do Registro: 132 bytes**



Como implementar uma estrutura de dados para armazenar os estudantes da escola que estão matriculados na disciplina “História” ?

Observação:



A escola tem 80 estudantes, mas somente **40** alunos estão matriculados na disciplina “História”.





Considerando que se conhece previamente o tamanho da tabela, a escolha natural é um array.

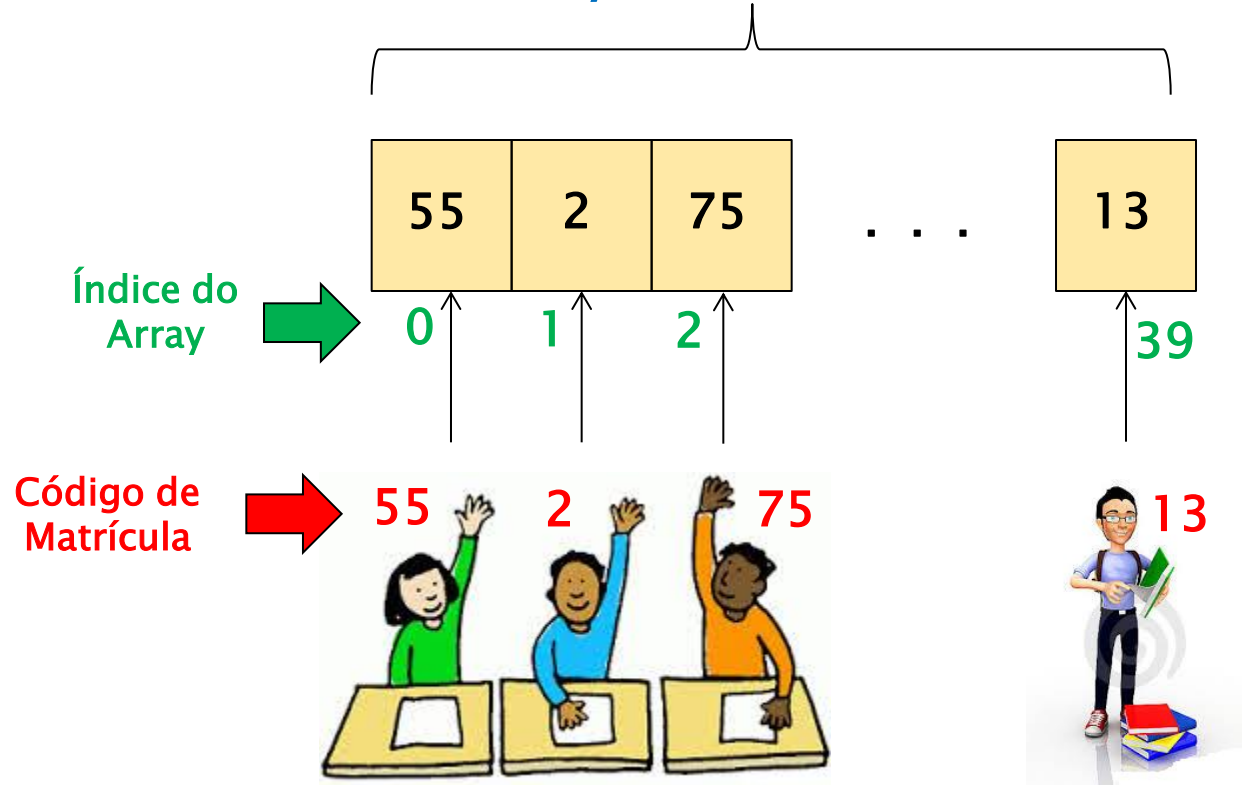






# Implementação

Array com 40 estudantes





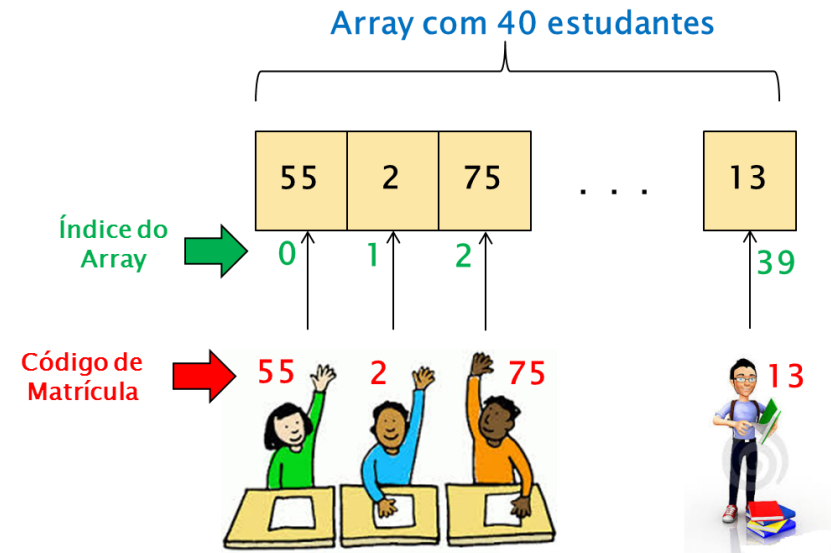
Quais os comentários em  
relação à essa implementação ?





# Implementação pura de Arrays

- ❑ O array tem tamanho exato para alocar 40 estudantes.
- ❑ Considerando que cada registro ocupa 132 bytes, o consumo de memória será de  $40 * 132 = 5280$  bytes = **~5Kb**





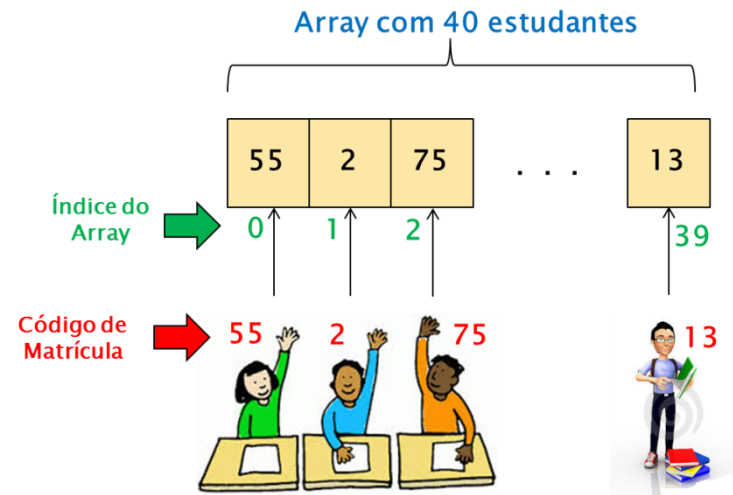
O array ocupa pouca memória, mas como é a eficiência para se efetuar a busca (searching) de um estudante ?





# Operação de busca na implementação pura de Arrays

- ❑ Não há relacionamento entre o código de matrícula (chave) e o índice do array.
- ❑ Além disso, os dados estão desordenados.
- ❑ Assim, a pesquisa será sequencial e o tempo é proporcional ao tamanho do array  $O(n)$ .
- ❑ Será necessário percorrer-se todo o array...





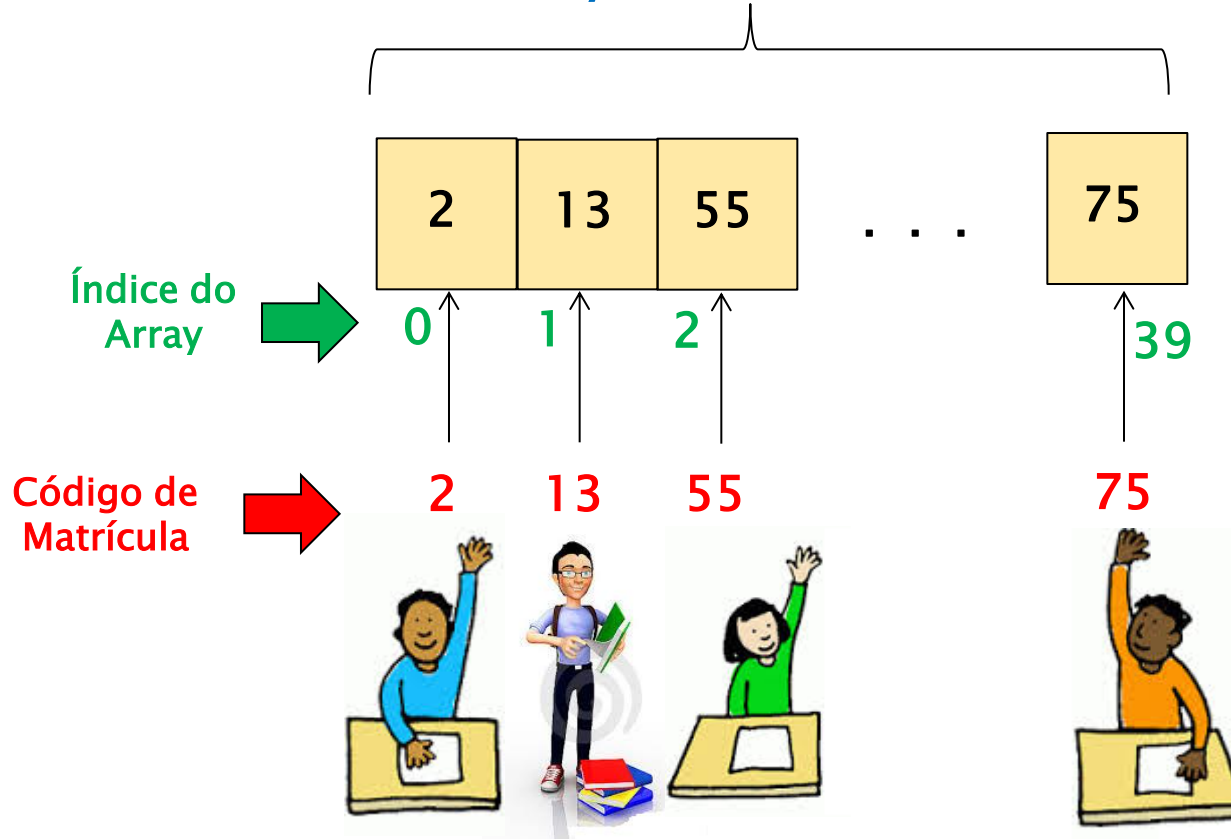
# Como melhorar a eficiência da busca ?





# Melhorando a eficiência da busca

## Array com 40 estudantes

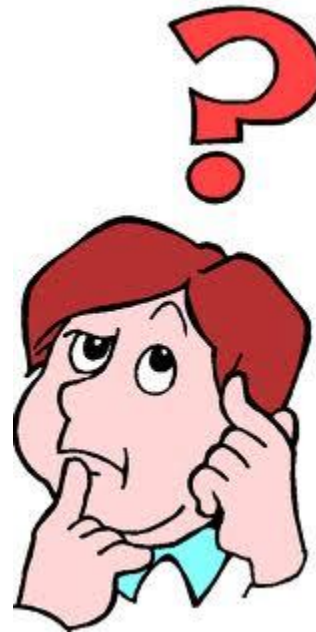


- ✓ Pode-se ordenar o array e efetuar-se uma **BUSCA BINÁRIA**
- ✓ Mas a complexidade ainda será  **$O(\log n)$**





Existe algum meio de se fazer uma busca  
com tempo constante  $O(1)$  ?



**Acesso Direto ao dado ?**

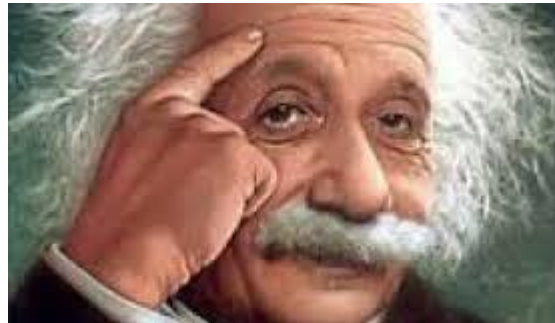


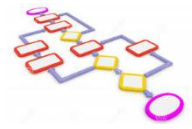




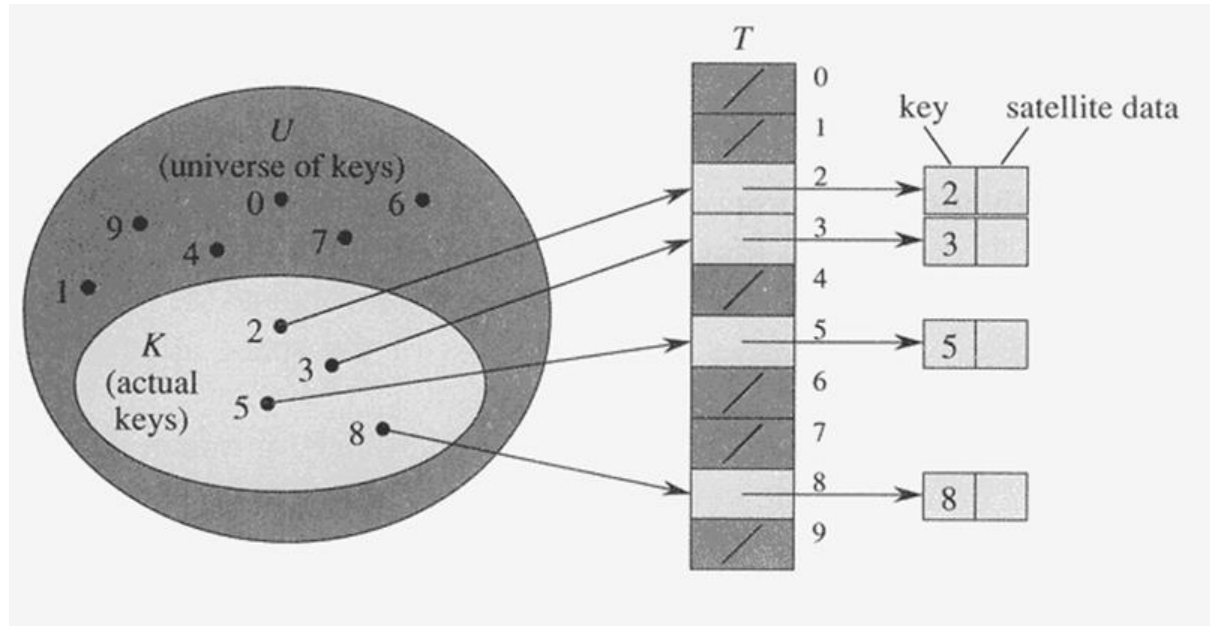
## Redesenhando o array

- ▣ Cada estudante tem código de matrícula com dois dígitos, que daria um conjunto universo  $U = (1, 2, \dots, 99)$ .
- ▣ Poderíamos criar um array com 100 elementos e associar cada código do estudante ao índice do array.
- ▣ Esse array é chamado ARRAY ASSOCIATIVO.



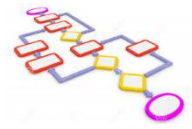


# Empregando array associativo



- A tabela  $T$  é acessada de modo direto.
- Cada chave no Universo  $U$  corresponde a um índice na tabela.
- O conjunto  $K$  de chaves reais estão associados na tabela  $T$  à pointers aos dados.
- Inviável do ponto de vista de alocação de memória quando  $|K| \ll |U|$

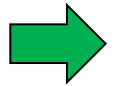




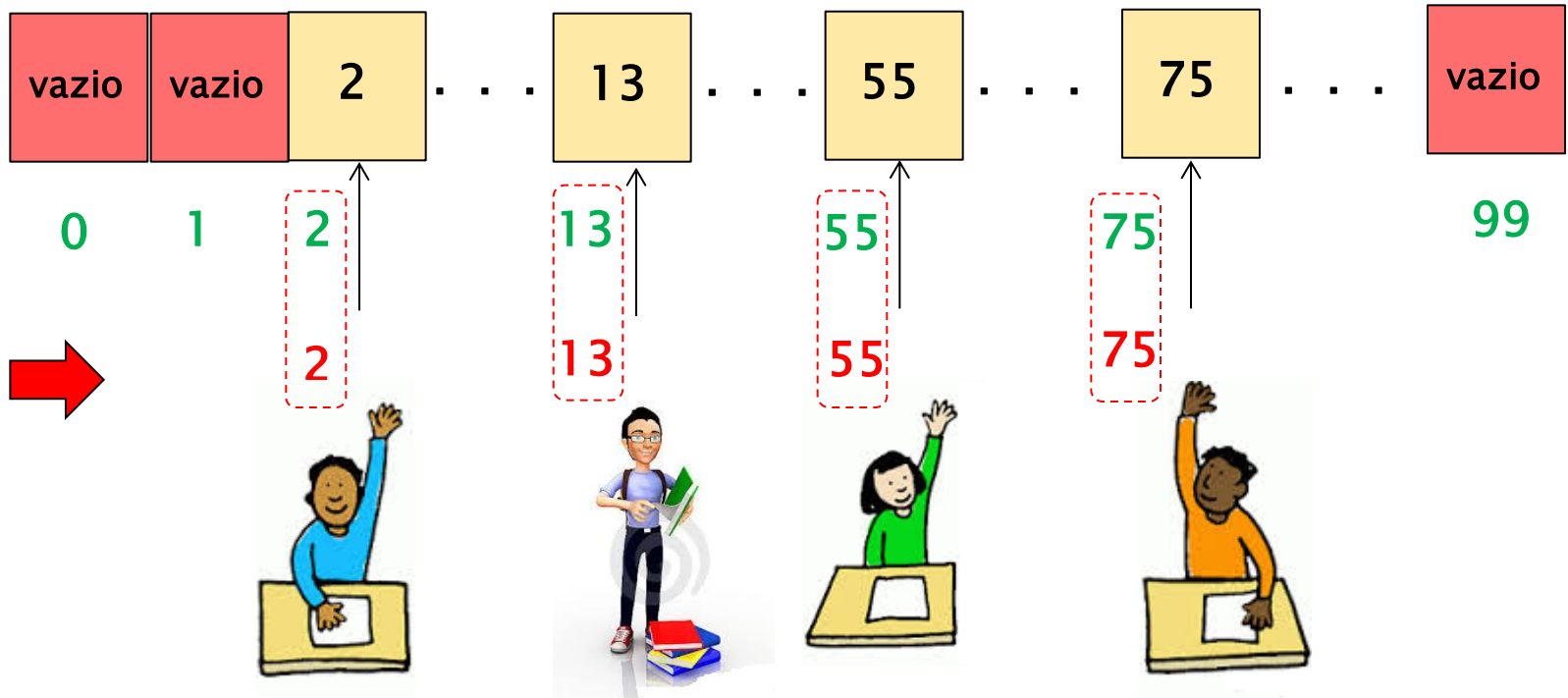
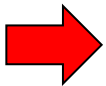
# Array Associativo

Array alocado para 100 estudantes

Índice do  
Array



Código de  
Matrícula





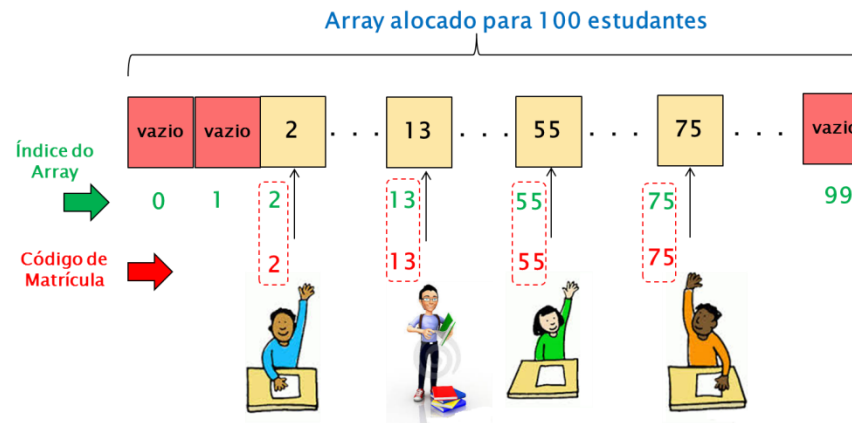
Quais as vantagens desse modelo ?





# Array Associativo – Vantagens

- O acesso aos dados do estudante é feito de forma **direta**, uma vez que o **índice do array coincide com a chave do estudante**.
- O estudante de chave 5 está na posição 55 do array.
- **O tempo para acessar o estudante não depende do tamanho do array**. Esse tempo é constante e a complexidade é  **$O(1)$** .



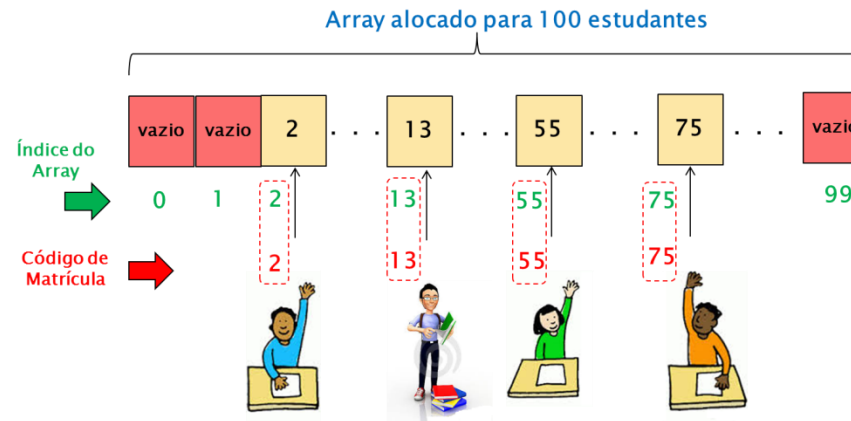


Quais as desvantagens desse modelo ?



## Array Associativo – Desvantagens

- Conforme premissa do problema, somente **40** alunos estão matriculados na disciplina “História”.
- No entanto, foi alocada memória para **100** estudantes.
- A melhoria da eficiência da busca teve o preço de maior alocação de memória.





Qual o novo valor da memória alocada ?







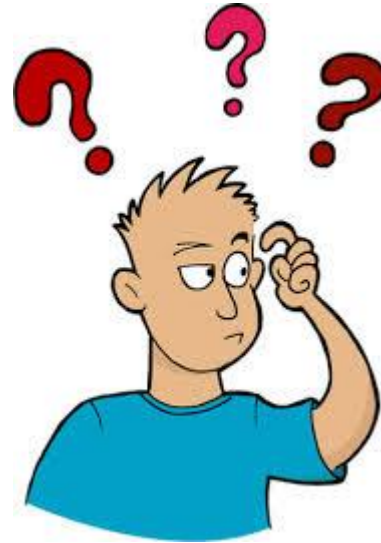
## Nova memória alocada

- O array tem tamanho exato para alocar 100 estudantes, embora somente 40 efetivamente serão usados.
- Considerando que cada registro ocupa 132 bytes, o consumo de memória será de  $100 * 132 = 13200$  bytes = **~13 Kb**
- Assim, nesse novo modelo, o consumo de memória aumento de 5Kb para 13kb.
- Houve um aumento de memória de **160 %**.



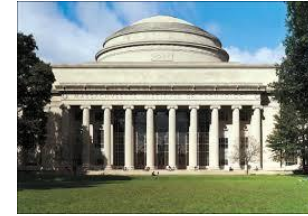


Mas, será que ainda há alguns inconvenientes com essa solução ?





## Nova situação



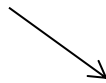
- Consideremos o mesmo problema...
- Porém, o estudante é de uma grande universidade com cerca de 10.000 alunos.
- Nessa Instituição, cada estudante é identificado por um código de matrícula correspondente a seu CPF.
- Por exemplo, o estudante André de Andrade Silva tem o código de matrícula associado a seu CPF: **640.348.123-15**.
- Assim, a chave do estudante é um valor de 11 dígitos.



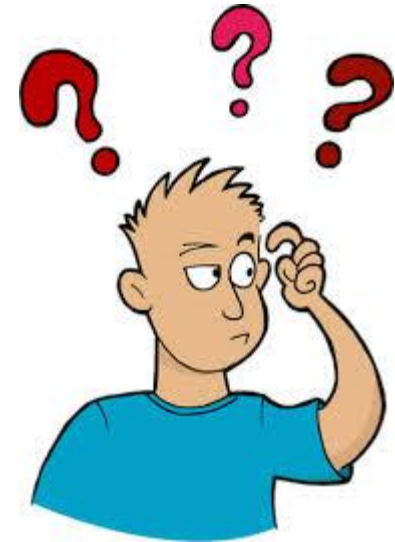


Considerando uma Universidade, como implementar uma estrutura de dados para armazenar os estudantes que estão matriculados na disciplina “História”

Observação:

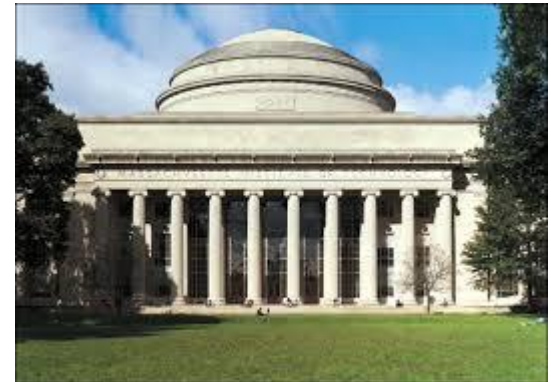


A Universidade tem 10.000 estudantes, mas somente **40** alunos estão matriculados na disciplina “História”.



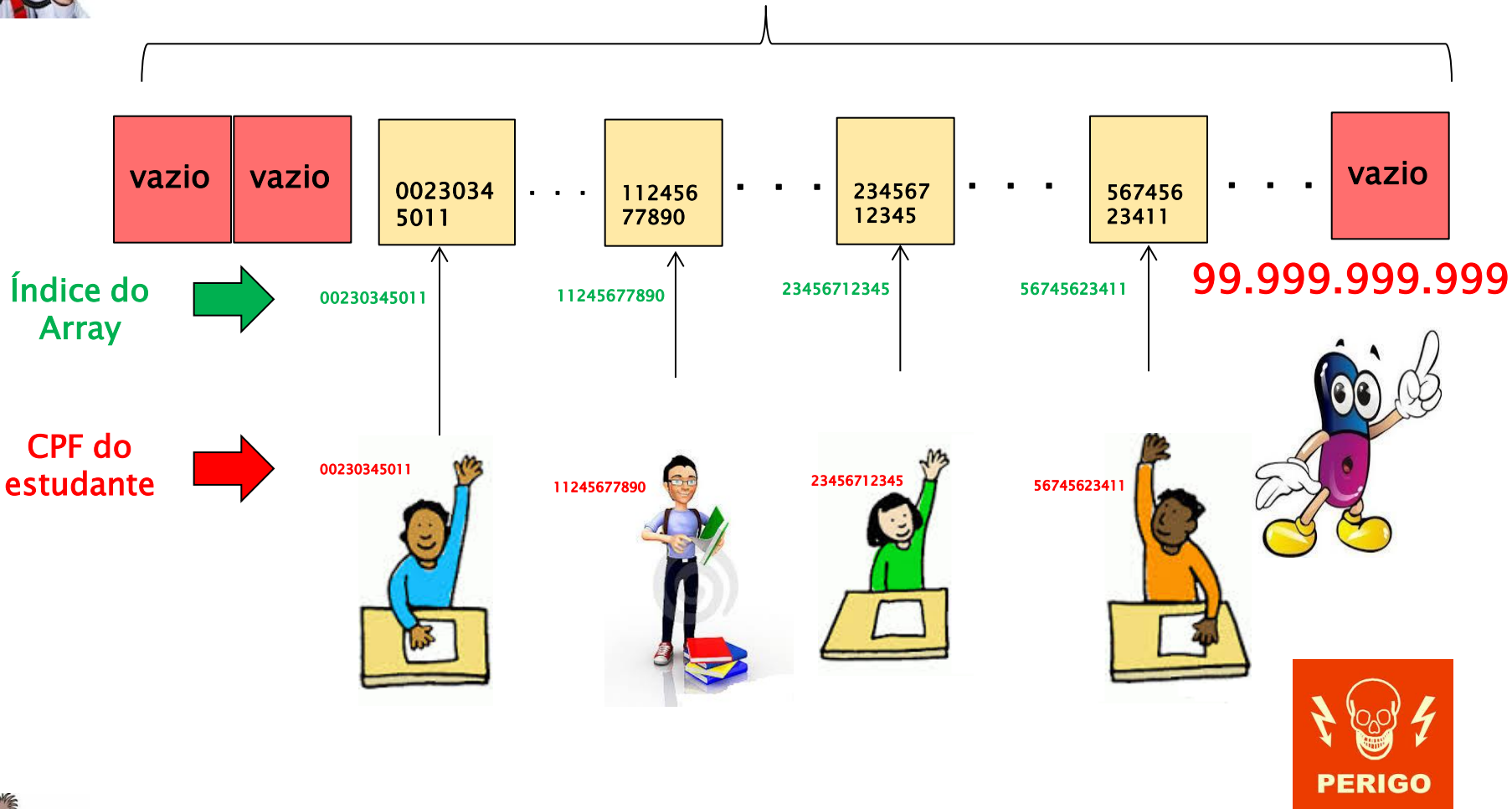


Será que para esse caso também poderemos modelar um array associativo ?



# Empregando array associativo

Array alocado para 100.000.000.000 estudantes





Qual o novo valor da memória alocada ?





## Nova memória alocada

- O array tem tamanho exato para alocar **100.000.000.000** estudantes, embora somente 40 efetivamente serão usados.
- Considerando que cada registro ocupa 132 bytes, o consumo de memória será de  $100.000.000.000 * 132 = 13.200.000.000.000 = \mathbf{13\ TB}$
- Assim, nesse novo modelo, o consumo de memória aumento de 5Kb para **13 Tb**.
- Dos **100.000.000.000** slots de memória somente **40** estarão sendo efetivamente alocados, o restante serão espaços vazios.







# Quando então usarmos Arrays Associativos ?





# Arrays Associativos

- São implementados por tabelas de acesso direto.
- Aplicáveis quando o conjunto universal de chaves U for pequeno.
- Operações de dicionários podem ser efetuadas em tempo constante -  **$O(1)$** .



Index Key	Element Value
1	100
2	200
3	300
4	400
5	500
6	600
7	700





Existe alguma forma de contornar o problema  
da excessiva alocação de memória ?



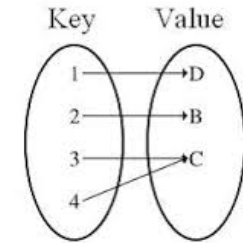


Como obter eficiência na busca do dado sem comprometer a alocação de memória ?





## Estrutura de Dados MAP



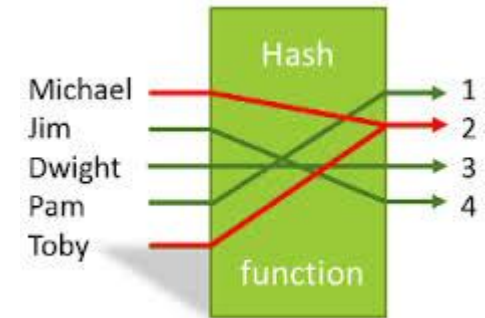
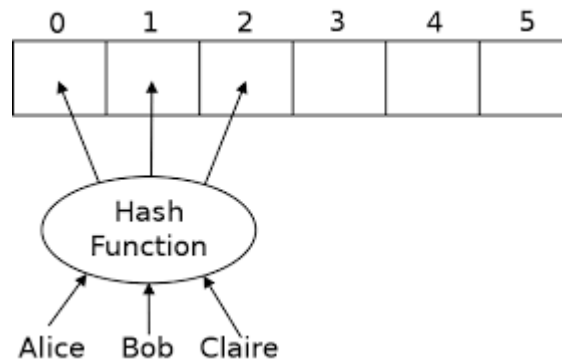
- Em termos matemáticos, um mapa (**map**) é uma estrutura de dados que estabelece uma relação de **mapeamento** entre dois conjuntos.
- Podemos definir um **mapa** como sendo um conjunto de **pares** na forma **(chave,valor)** no qual cada **chave** está associada a um determinado **valor**.
- Mapas são também chamados de Estruturas de Dados **Dicionário**.
- A implementação pode ser feita por **arrays associativos** ou por **tabelas hash**.





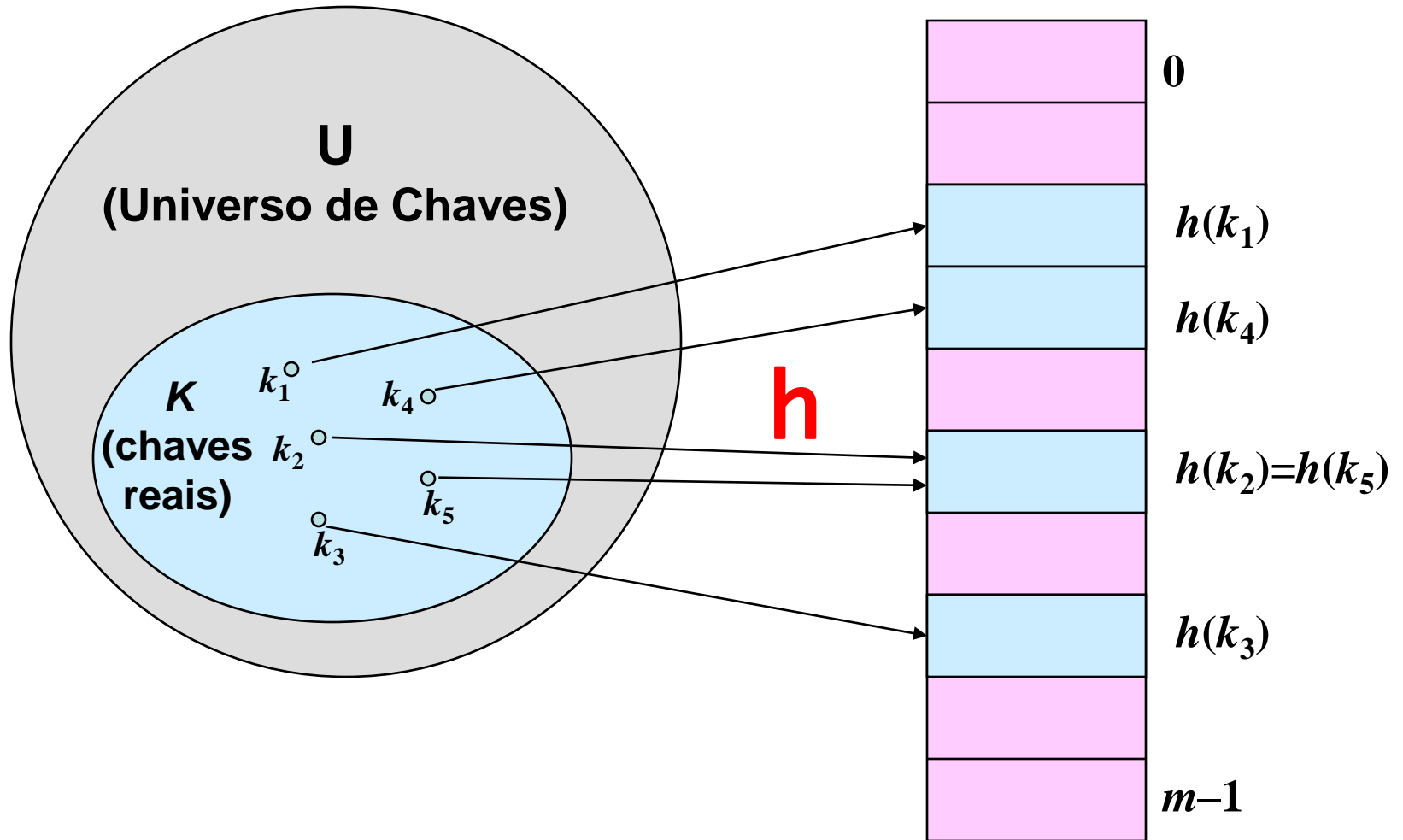
# Hash Tables

- Correspondem a tipos de arrays associativos que são implementados quando  $|k| \ll |U|$ , sendo  $k$  o conjunto de chaves válidas e  $U$  o conjunto universo de chaves.
- O mapeamento entre  $k$  e  $U$  é feito por meio de uma função.
- Essa função denomina-se Função de HASH.





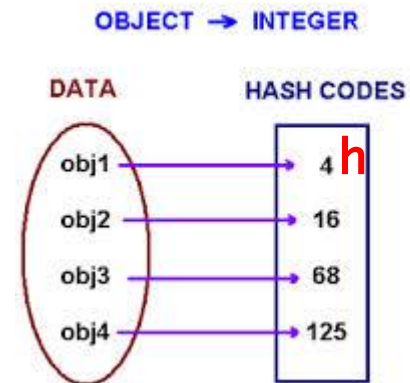
# Hashing





# Hashing

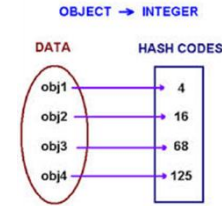
- Função Hash **h**: Efetua o mapeamento de U para os slots da hash table T[0..m-1].  
 $h : U \rightarrow \{0, 1, \dots, m-1\}$
- Com arrays, a chave K é mapeada para o slot A[k].
- Com hash tables, a chave K é mapeada para o slot T[h(k)].
- h(k)** é o valor hash da chave k.







# Hashing

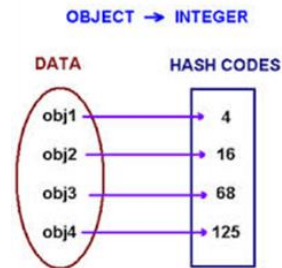


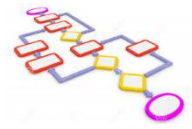
- ⊕ Aplica uma fórmula para calcular um endereço, determinando assim o posicionamento de um dado.
- ⊕ O cálculo é feito através de uma função que mapeia as chaves dentro do conjunto (**Hashing**).
- ⊕ Permite "acesso direto" aos registros como um índice (endereço) dentro da tabela.
- ⊕ Útil quando a busca é feita sobre um número muito grande de dados que possuam faixas de valores muito variável. Exemplo: CPF em um arquivo de habitantes de uma cidade.





# Gerando Funções Hashing Método da Divisão

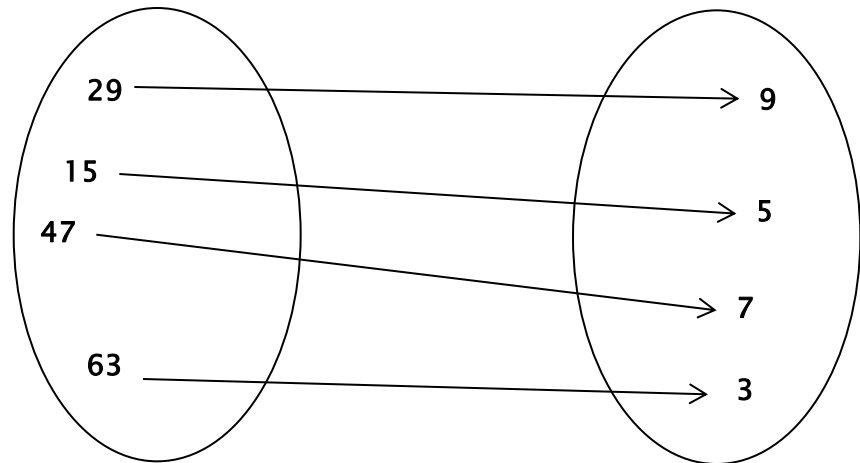
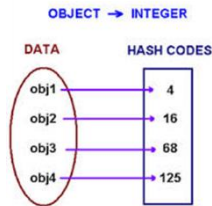


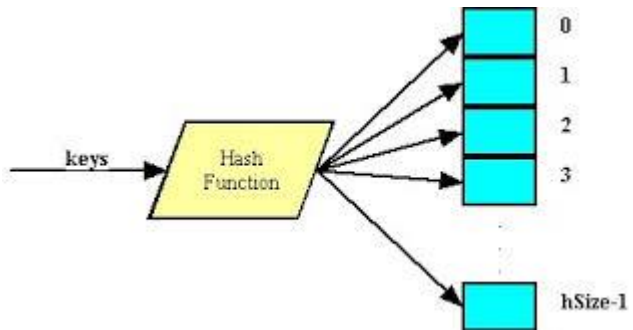


# Gerando Funções Hashing Método da Divisão

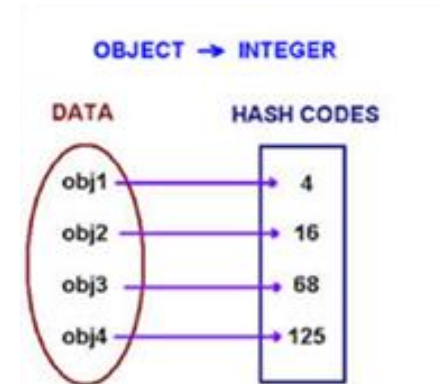
**Princípio Básico:** O endereço do elemento na tabela é dado pelo resto da divisão da sua chave por  $m$  ( $h(k) = k \bmod m$ ), onde  $m$  é o tamanho da tabela  
 $x$  é um inteiro correspondendo à chave

$$h(k) = k \bmod 10$$





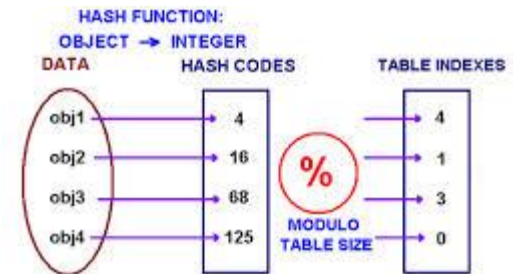
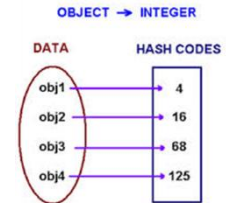
## Exemplo - Hashing



# Hashing – Exemplo

⊕ Consideremos uma tabela hash com as seguintes características:

- ◆ A tabela terá um máximo de **80** entradas (**0-79**).
- ◆ Cada entrada da tabela irá armazenar o nome de um estudante.
- ◆ Os valores dos campos chave terão valores contidos no intervalo [**0..1000**]
- ◆ A função **HASH** é definida por  **$h(k) = k \% 80$** .
- ◆ Ou seja, a função  **$h(k)$**  recebe uma chave  **$k$**  (entre 0 e 1000) e retorna um valor (índice) entre 0 e 79.



HASH TABLE

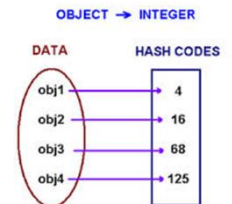
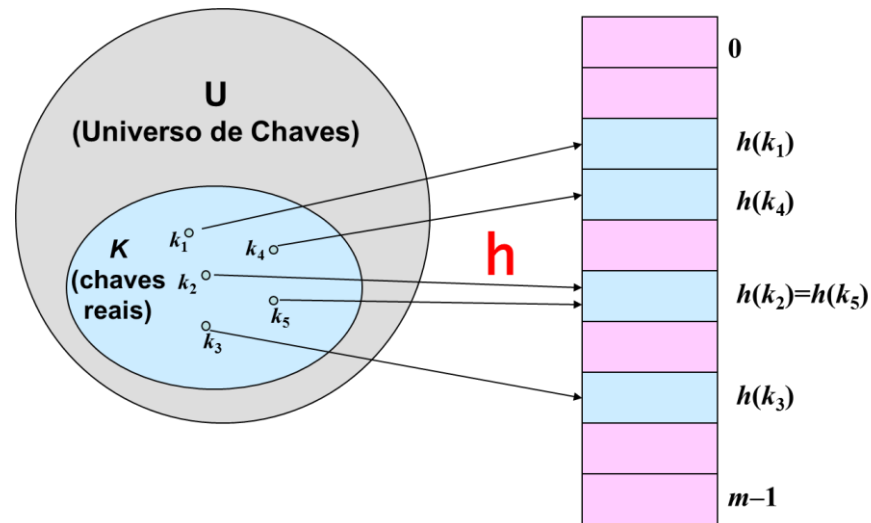
OBJ4	OBJ2		OBJ3	OBJ1
0	1	2	3	4





# Simulações da Função Hash

Key	383	487	235	527	510	564	103	66	14
Índice	63	7	75	47	30	4	23	66	14





Puxa ! Então com o uso da Função Hash resolvi todos os problemas ?

Pouco uso de memória e acesso direto aos dados –  $O(1)$  !!!





Pois é ! Tudo na vida tem um preço ! ! !







# Hashing – Observação



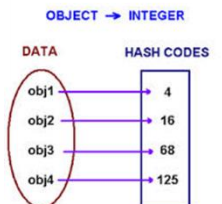
⊕ No exemplo anterior, vamos considerar as seguintes chaves:

Key	100	180	260
Índice	20	20	20

Função **HASH**:  $h(k) = k \% 80$ .



As chaves 100, 180 e 260 resultaram no mesmo índice !!!





Esse problema é conhecido por





Executando novamente a aplicação



```

package maua;

public class Hash_02 {

    public static void main(String[] args) {

        Integer[] tabKeys = { 100, 180, 260} ;

        String[] tabNomes = { "Ana", "Ivo", "Ari"} ;

        String[] tabHash = new String[80];

        for (int i=0; i<tabKeys.length; i++ ) {

            System.out.println("Chave: " + tabKeys[i] + "
                               hashCode = " + hash(tabKeys[i]));
            tabHash[hash(tabKeys[i]) ] = tabNomes[i];

        }

        for (int i = 0; i < tabHash.length ; i++)
            if (tabHash[i] != null )
                System.out.println("Indice = " + i + " ==> Valor
                                   armazenado na tabela hash: " + tabHash[i] );

        }

    public static Integer hash(Integer key) {
        return (key % 80);
    }

}

```





## Executando novamente a aplicação

Chave: 100    hashCode = 20

Chave: 180    hashCode = 20

Chave: 260    hashCode = 20

Índice = 20    ==> Valor armazenado na tabela hash: Ari



Somente uma chave  
foi armazenada na  
tabela HASH !!!





# Como resolver o problema da colisão na tabela HASH ?





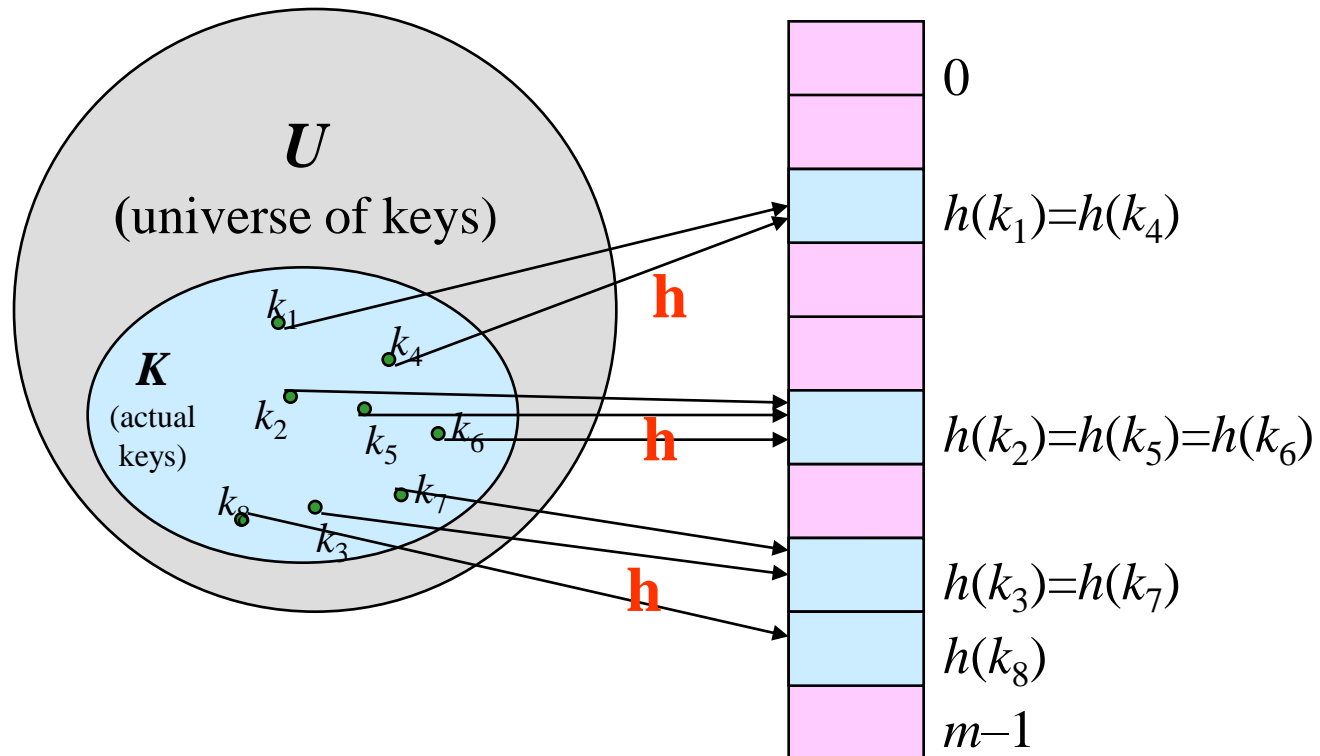
# Tratamento da colisão na tabela HASH

- ⊕ Encadeamento
- ⊕ Endereçamento aberto (Rehashing)





# Resolução da Colisão por Encadeamento

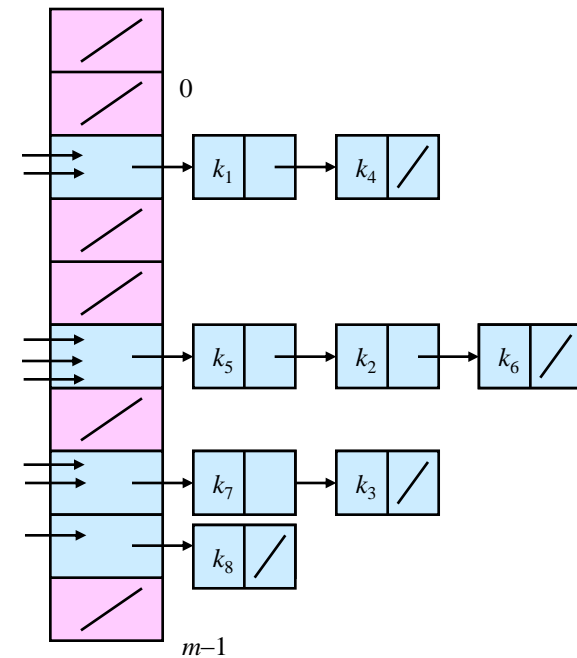






# Hashing com Encadeamento

- ⊕ Todos os elementos que têm mesmo hashcode são armazenados num slot que referencia uma lista encadeada (ligada).
- ⊕ O pointer para o head da lista é armazenado no slot da tabela hash.



# Hashing com Encadeamento

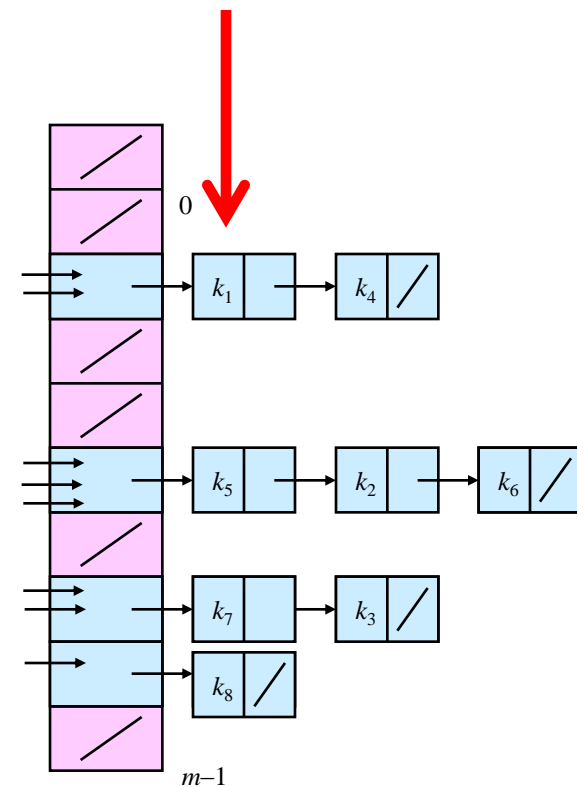
## Operações de Dicionário

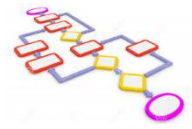
- ⊕ Operação de **Insert** no head da lista:

$T[h(\text{key}[x])]$

- ⊕ Complexidade no pior caso =>  **$O(1)$**

**Insert – head da lista**





# Hashing com Encadeamento

## Operações de Dicionário

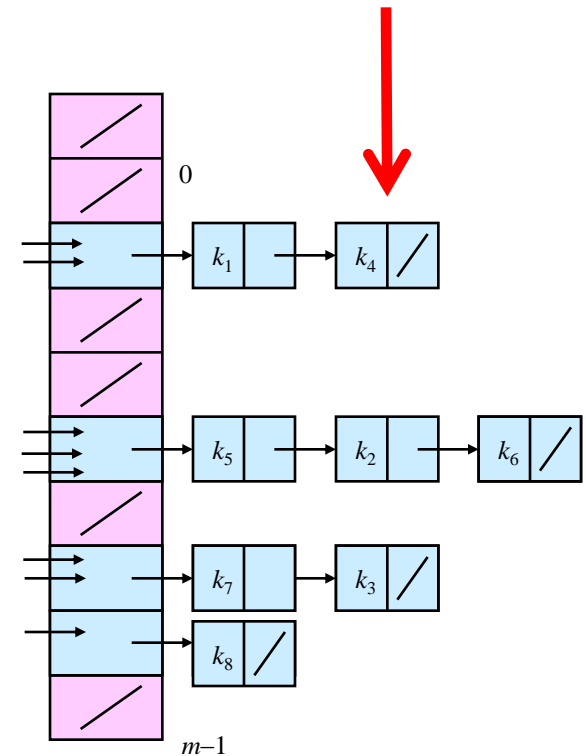
- ⊕ Operação de **Delete** numa posição qualquer da lista encadeada:

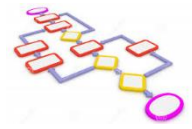
$$T[h(\text{key}[x])]$$

- ⊕ Complexidade no pior caso => **Proporcional ao tamanho da lista:  $O(n)$**



Delete numa posição x





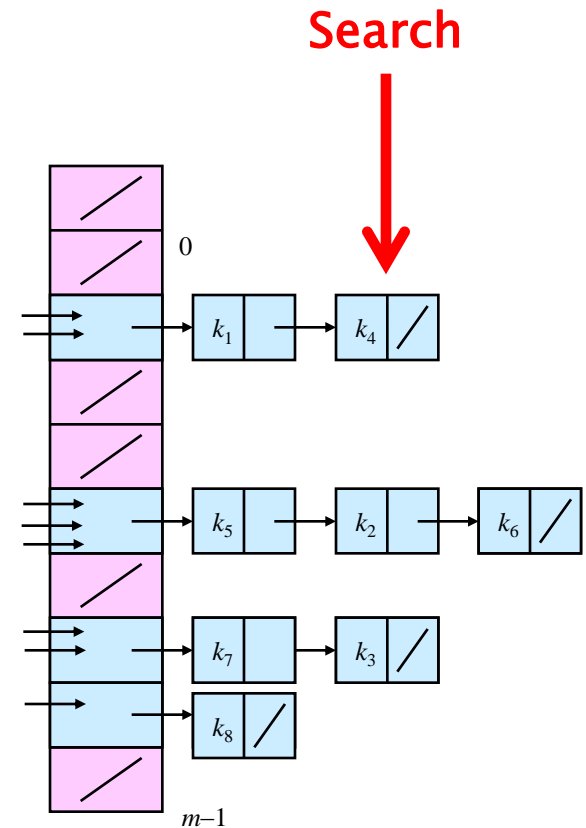
# Hashing com Encadeamento

## Operações de Dicionário

- ⊕ Operação de **Search** de uma chave K na lista encadeada:

$T[h(k)]$

- ⊕ Complexidade no pior caso => **Proporcional ao tamanho da lista:  $O(n)$**



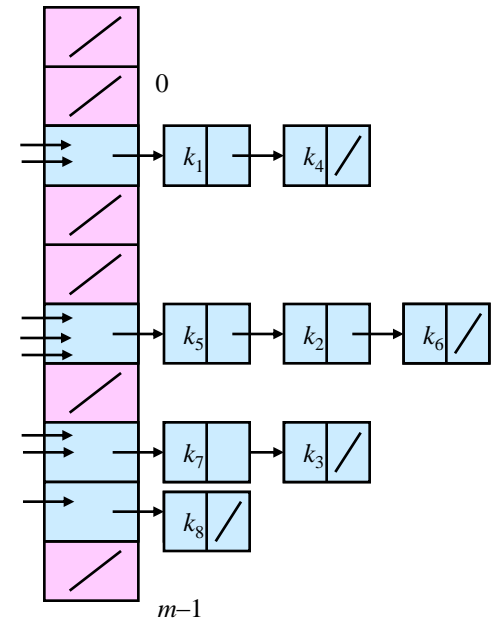
# Hashing com Encadeamento

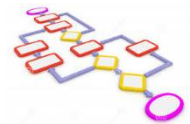
## Exercício 1

- ✓ Insira os seguintes elementos em uma tabela Hash.
- ✓ A função de hashing é “resto da divisão por 10”.
- ✓ Adotar a técnica de encadeamento para tratar as colisões.

Valores:

**23, 45, 77, 11, 33, 49, 10, 4, 89, 14**



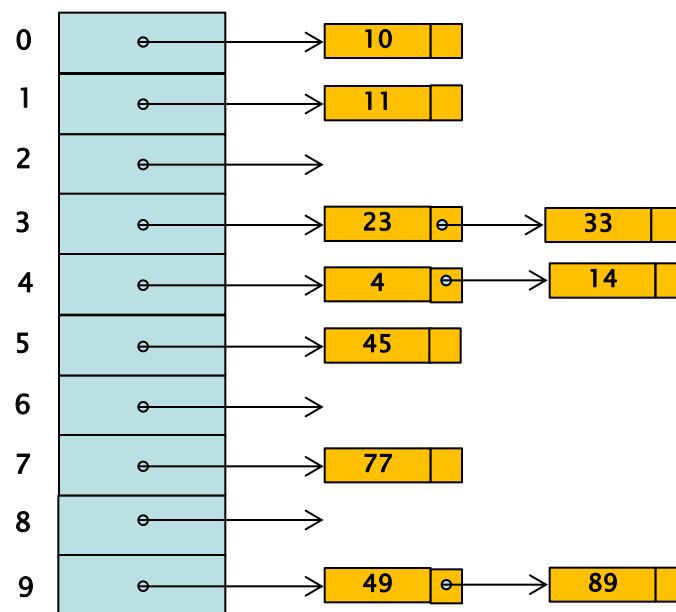


# Hashing com Encadeamento

## Exercício 1 – Solução

Chave	23	45	77	11	33	49	10	4	89	14
Endereço Calculado	3	5	7	1	3	9	0	4	9	4
Colisão ?	N	N	N	N	S	N	N	N	S	S

Função de Hashing:  $E(c) = c \% 10$

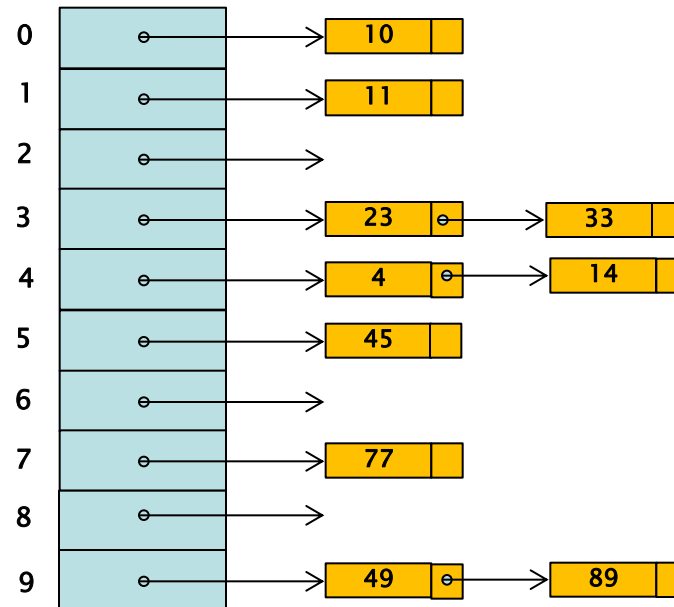




# Hashing com Encadeamento

## Exercício 2

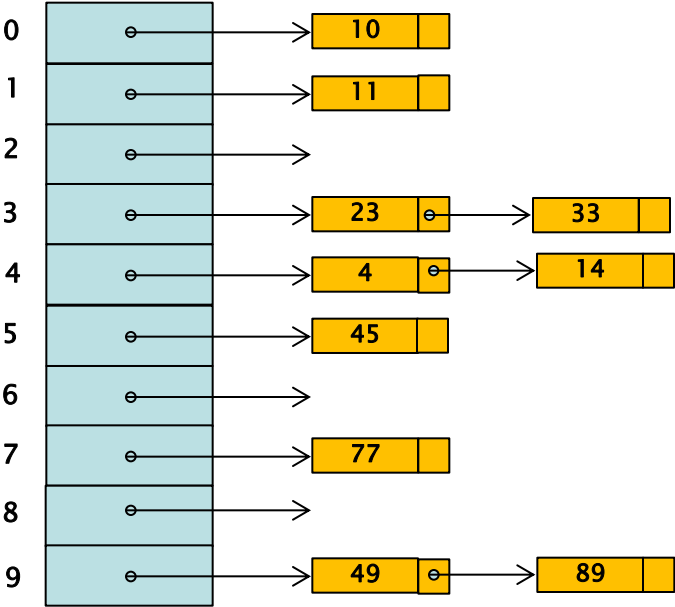
- ✓ Qual o número de acessos necessários para buscar cada registro inserido no exercício anterior ?



# Hashing com Encadeamento

## Exercício 2 – Solução

✓ Qual o número de acessos necessários para buscar cada registro inserido no exercício anterior ?



Chave	Nº de acessos
23	1
45	1
77	1
11	1
33	2
49	1
10	1
4	1
89	2
14	2
total	13
média	1,3



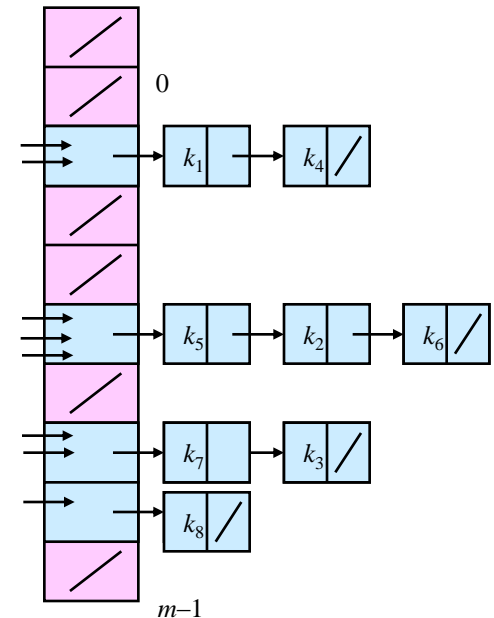


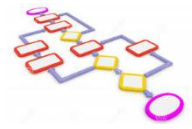
# Hashing com Encadeamento

## Exercício 3

- ✓ Insira os seguintes elementos em uma tabela Hash.
- ✓ Função Hashing:  $F(k) = k \bmod 10$ .
- ✓ Usar hashing encadeado.

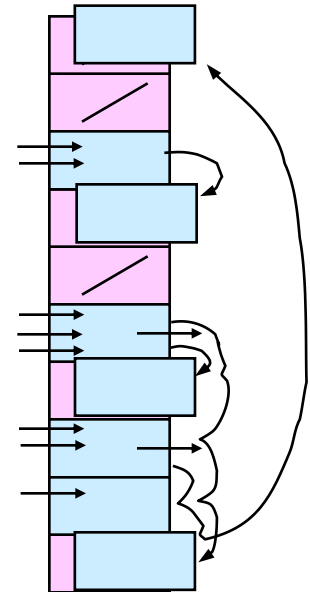
*Valores:* **23, 45, 77, 11, 33, 49, 10, 4, 89, 14**

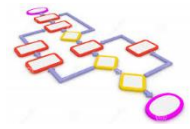




# Hashing com Endereçamento Aberto (Rehashing)

- ⊕ Todos os elementos são armazenados na própria tabela hash.
- ⊕ Quando ocorrer colisão, usa-se um procedimento sistemático (consistente) para armazenar os elementos em slots livres da tabela.
- ⊕ Filosofias para tratamento de colisão:  
Busca Linear (utilização do primeiro espaço vazio)





# Hashing com Endereçamento Aberto (Rehashing)

Chave	23	45	77	11	33	49	10	4	89	14
Endereço Calculado	3	5	7	1	3	9	0	4	9	4
Colisão ?	N	N	N	N	S	N	N	S	S	S
Endereço Efetivo					4			6	2	8

	Chave	Situação
0	10	<u>1</u>
1	11	<u>1</u>
2	89	<u>1</u>
3	23	<u>1</u>
4	33	<u>1</u>
5	45	<u>1</u>
6	4	<u>1</u>
7	77	<u>1</u>
8	14	<u>1</u>
9	49	<u>1</u>

(0=livre 1=ocupado)

**Função de Hashing:  $E(c) = c \% 10$**

10	11	89	23	33	45	4	77	14	49
0	1	2	3	4	5	6	7	8	9

**A inclusão da chave 33 gera colisão na posição 3.  
Insere na posição 4, que é a primeira livre após 3.**



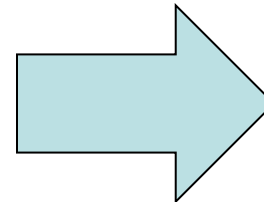


# Hashing Aberto - Exercício 4

**Insira as chaves em uma tabela Hash: 73, 15, 44, 37, 30, 59, 49, 99**

*Função Hashing:  $h(k) = k \bmod 11$*

k	h(k)
73	7
15	4
44	0
37	4
30	8
59	4
49	5
99	0



0	44
1	99
2	
3	
4	15
5	37
6	59
7	73
8	30
9	49
10	

