PUCRS
Pontifícia Universidade Católica
do Rio Grande do Sul

ESCOLA
POLITÉCNICA

Hash e
Índices Bitmap

98H00-04 - Infraestrutura para Gestão de Dados

Prof. Msc. Eduardo Arruda
eduardo.arruda@pucrs.br

Índices *Hash*

- Utiliza técnica de *hashing* para distribuir registros em um espaço de distribuição
- Emprega uma função *hash* (de “espalhamento”) para organizar registros em um *data file* ou para organizar chaves de busca em um índice
  - A função *hash* é aplicada sobre o valor da chave de busca
  - Seu resultado indicará em que *bucket* (depósito) o registro ou entrada de índice deve ser armazenado (na inserção) ou está localizado (em buscas)
- Presta-se somente para buscas por igualdade
- Sensível à distribuição dos dados
  - Caso seja escolhida uma função *hash* inadequada, os registros podem se concentrar em poucos *buckets* e gerar muito *overflow*

Exemplo de função de *hash* para chaves numéricas

- Chave =‘x_1 x_2 … x_n’’, sequência de caracteres n bits
- Para endereçar 2^b *buckets*:

  h(k) = b bits menos significativos da chave k

- Exemplo: para endereçar 8 *buckets*
  - Chave k = 1971 = 11110110**011**
  - h(k) = 3 bits menos significativos de 1971 = 011 = 3
    - o registro será inserido ou está localizado no *bucket* 3

Exemplo de função de *hash* para sequências de caracteres

- Chave =‘x_1 x_2 … x_n’’, sequência de caracteres de n Bytes
- Para endereçar b *buckets*:

$$h(k) = \text{soma}\ (x_1\ x_2\ \ldots\ x_n) \bmod b$$

- Exemplo: para endereçar 9 *buckets*
  - Chave = ‘maria’ a=1, b=2, … , z=26
  - h(k) = (13+1+18+9+1) mod 9 = 6
    - o registro será inserido ou está localizado no *bucket* 6

Qualidade da função *hash*

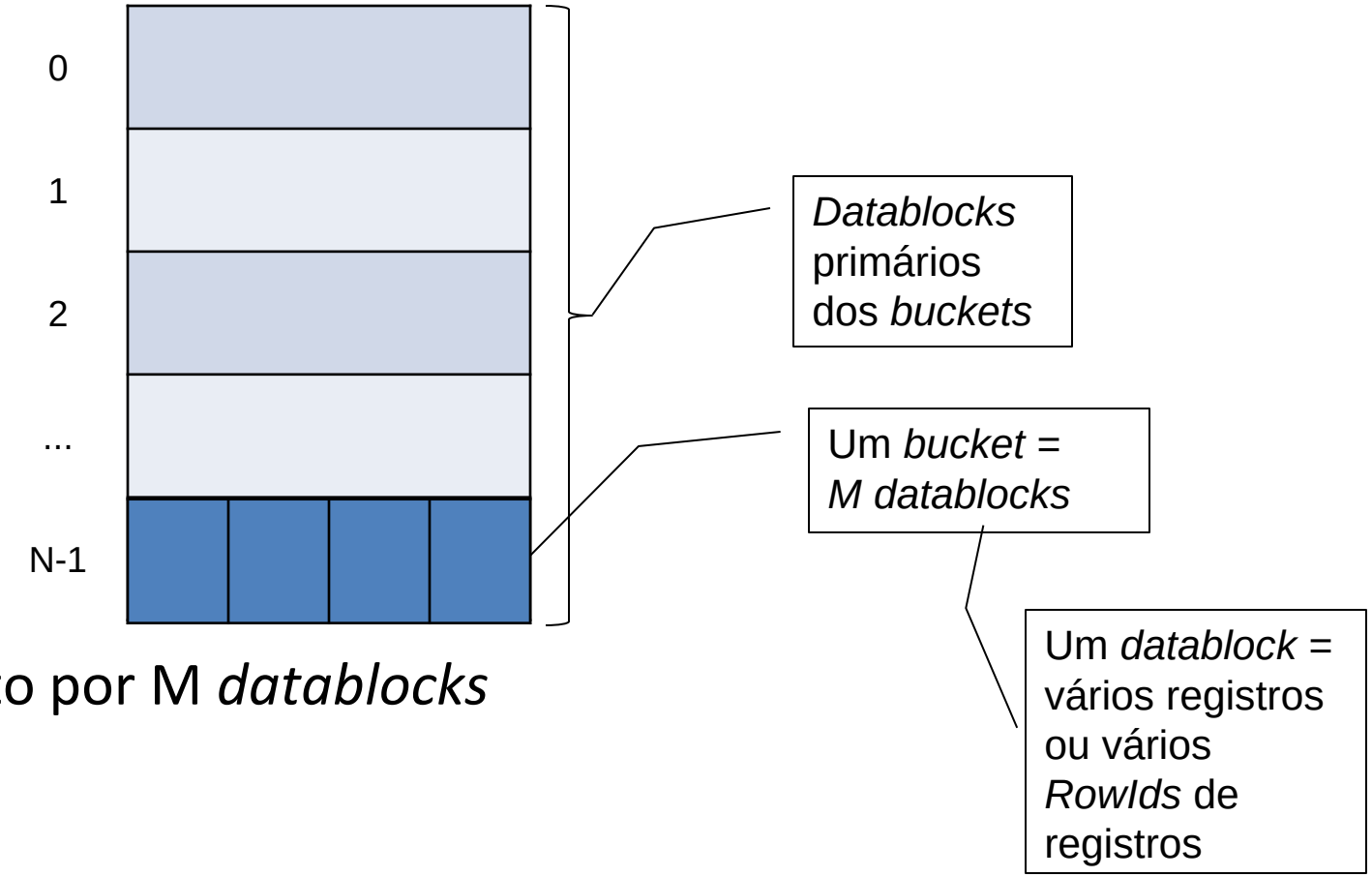
- Uma função *hash* é boa se o número esperado de chavespor *bucket* é o mesmo para todos os *buckets*, ou seja, a função gera uma distribuição homogênea
- Exemplo:
  - Se para sequências de caracteres escolhemos como função *hash* o número da primeira letra:

    A = 1, ... , Z = 26

    essa função não é boa. Por quê???

Estrutura de *hash*

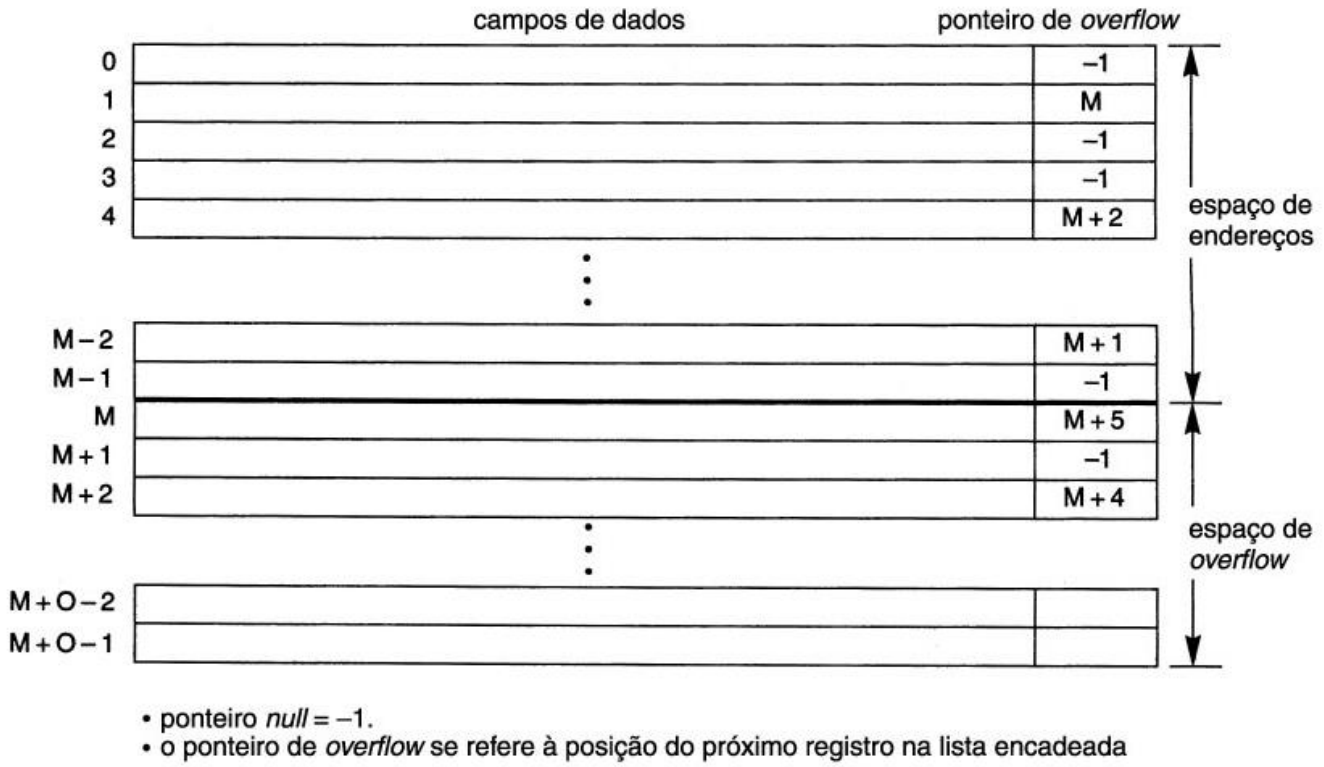
- Alocação *sequencial* de N *buckets*



- Cada *bucket* é composto por M *datablocks*

Overflow de *buckets*

- O espaço é dividido estaticamente em espaço de endereços e espaço de *overflow*
- Quando a função não é adequada, a multiplicação de *overflow* irá prejudicar o desempenho



Fonte: ELMASRI, R.; NAVATHE, S. B. Sistemas de Banco de Dados. 4ed. São Paulo: Pearson Education, 2005. 724p..

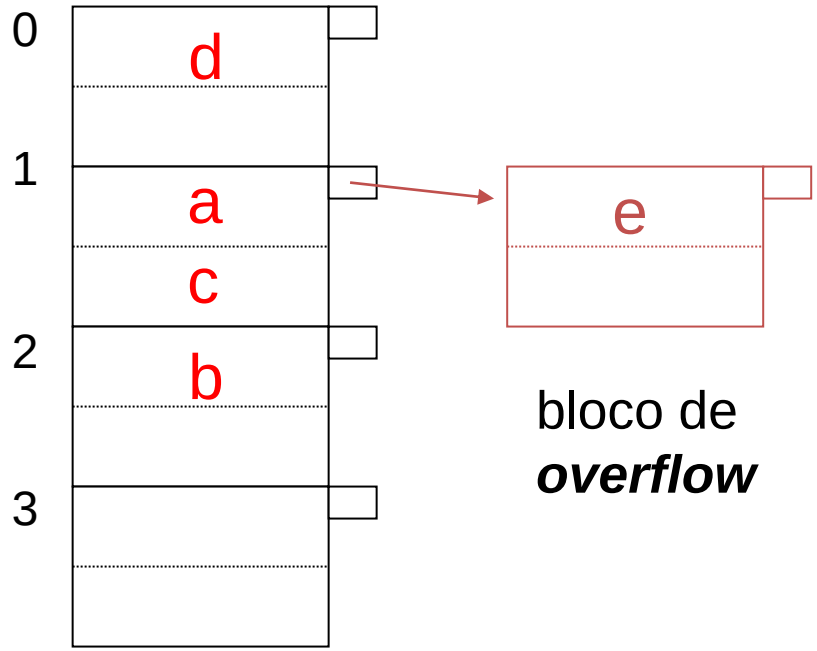
Hashing estático

- O número de *buckets* deve ser dimensionado de acordo com o número esperado de linhas na tabela
  - buckets = (linhas / linhas_por_*bucket*) * fator_ajuste (ex.: 1,25)
- Dentro de 1 bucket devemos manter as chaves ordenadas?
  Sim, se a frequência de inserção/deleção é baixa.

Exemplo

h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0

Inserção



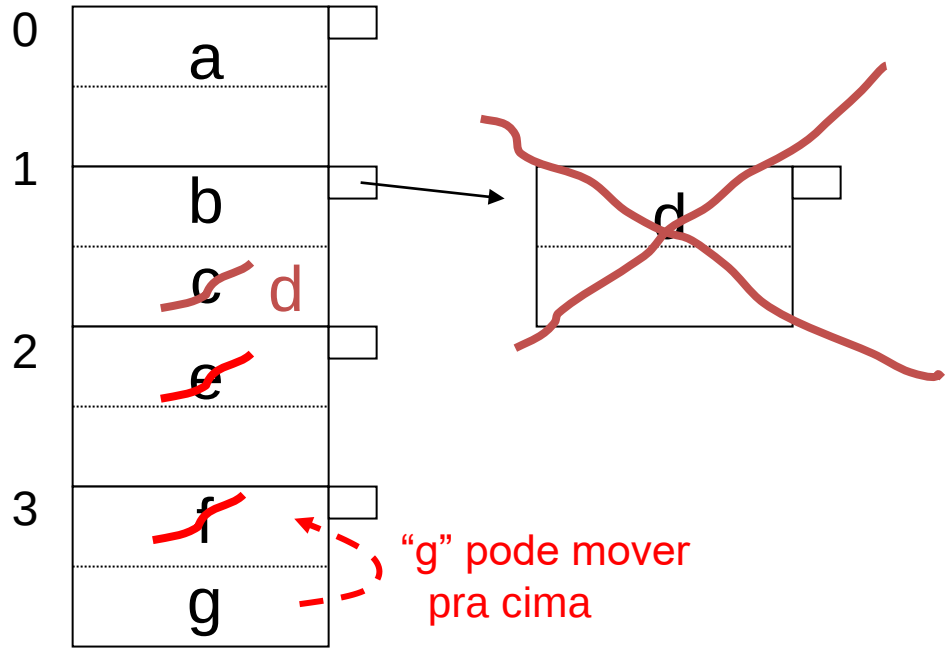
h(e) = 1

Exemplo

Exclusão

e
f

c



Overflow

Se o número de blocos de overflow fica significativo

↓

Busca vai ficando “sequencial”

↓

Perda de desempenho

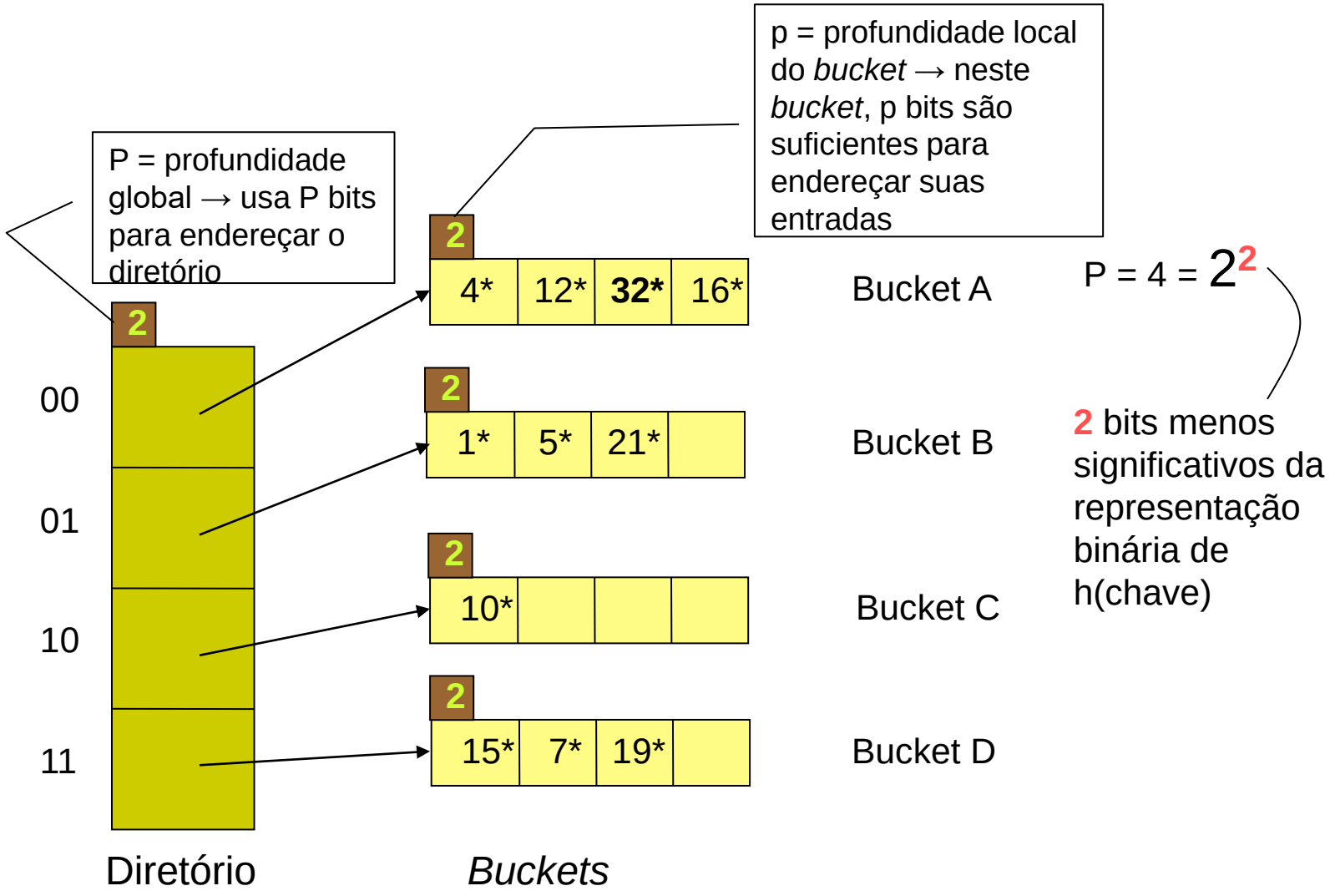
Regra geral

- Tentar manter o espaço utilizado entre 50% e 80%
  Utilização = # chaves usadas / # total espaços
  - Se < 50%, desperdiça espaço
  - Se > 80%, *overflow pode ser* significativo
- Deve-se ter:
  - Uma função *hash* boa
  - Número de entradas por *bucket* bem dimensionado

Hashing extensível

- É empregada função *hash* que gera inteiros binários com b bits
- i bits do início do valor do *hash* são usados como índice para uma tabela auxiliar de endereçamento dos *buckets*
- O valor de i cresce ou decresce com o aumento ou diminuição do número de registros
- O espaço de endereçamento possui um inteiro P, chamado “profundidade global”, que define o número de bits máximo utilizado para endereçamento de *buckets*
- Cada *bucket* j possui um inteiro p_j, chamado “profundidade local”, que define o número de bits utilizado para endereçamento daquele *bucket*

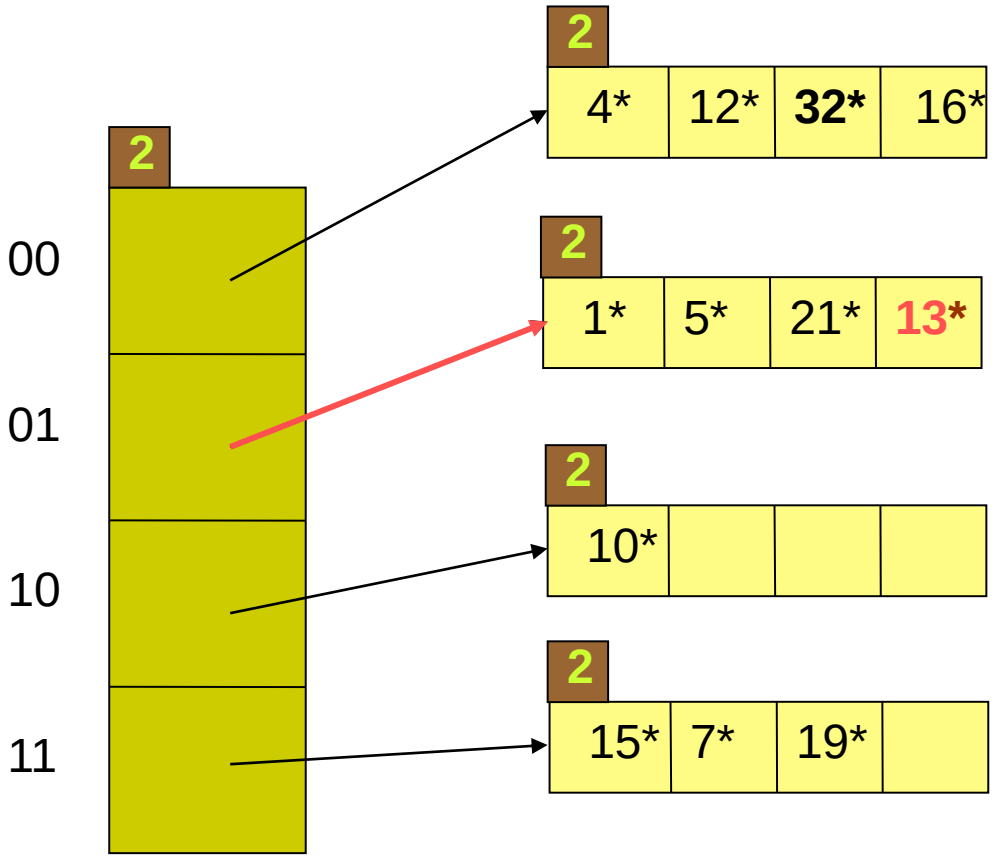
Exemplo


p = profundidade local do *bucket* → neste *bucket*, p bits são suficientes para endereçar suas entradas
P = profundidade global → usa P bits para endereçar o diretório
2
4* 12* 32* 16*
Bucket A
2
1* 5* 21*
Bucket B
2
10*
Bucket C
2
15* 7* 19*
Bucket D
00
01
10
11
Diretório
Buckets
P = 4 = 2^2
2 bits menos significativos da representação binária de h(chave)

Algoritmo de inserção em *hashing* extensível

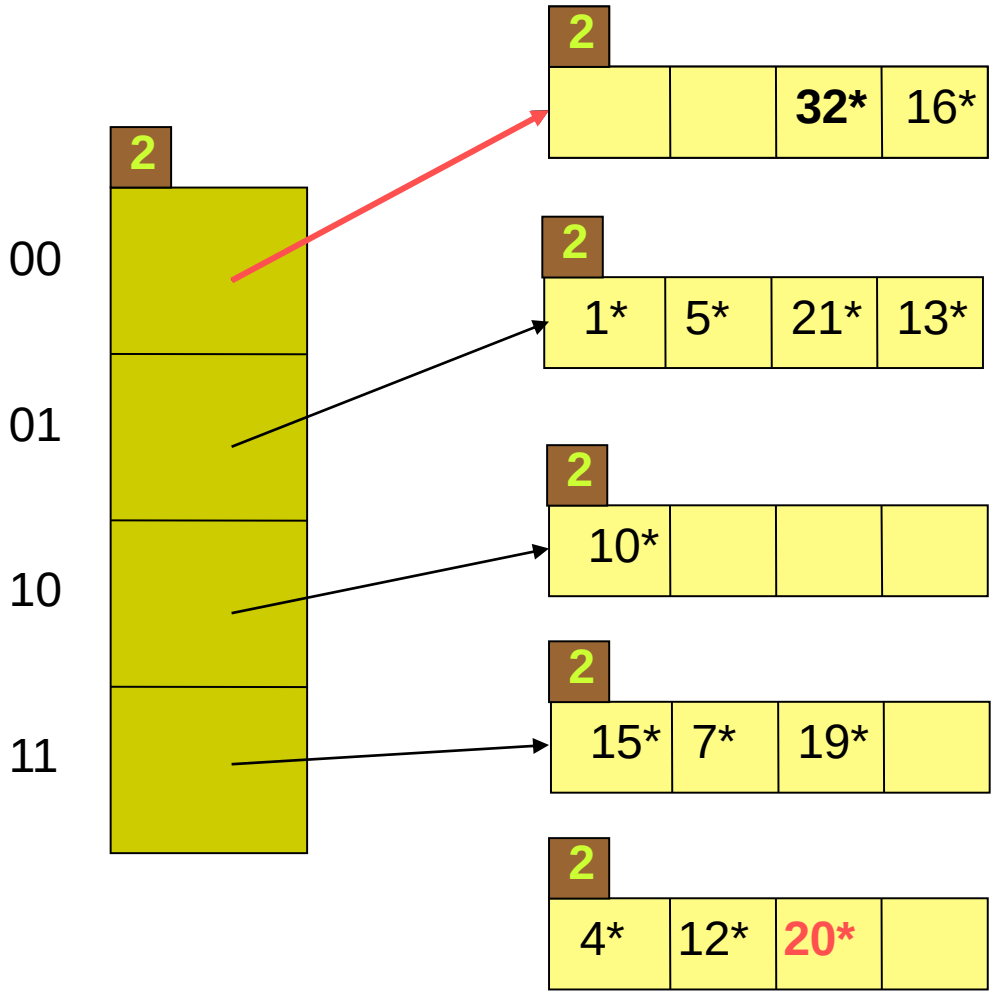
- Aplica-se a função *hash* e considera-se P bits, localizando a entrada no diretório
- Se não houver *overflow*, insere
- Se houver overflow, verifica:
  - Se p < P, então p = p + 1, aloca-se um novo *bucket* (“*bucket* ïrmão” – p bits são iguais) e redistribui-se as entradas entre estes dois *buckets*, considerando-se p bits
  - Se p = P, então duplica-se o diretório, fazendo P = P + 1, faz-se todas as demais novas entradas do diretório apontarem para os mesmos *buckets* de suas “entradas irmãs”, aloca-se um novo *bucket* e redistribui-se as entradas entre estes dois *buckets* considerando-se P bits
    - Os *buckets* que foram objeto do *overflow* serão os únicos com p = P

Inserção sem *overflow*



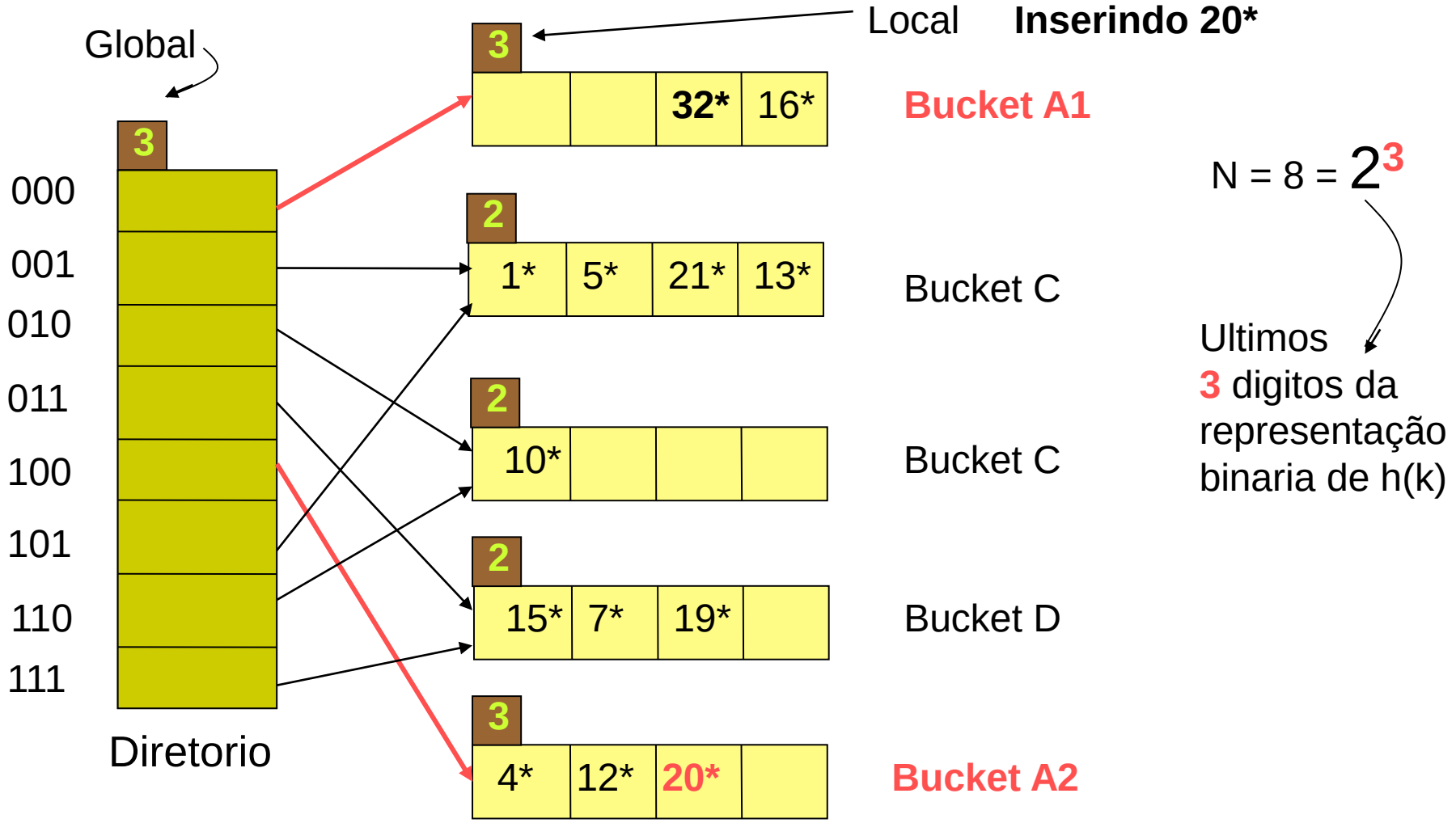
Inserindo 13*

Inserção com *overflow*



Inserindo 20*

Inserção com *overflow* e duplicação de diretório



Hashing linear

- Permite a expansão ou redução do número de *buckets* sem a necessidade de um diretório
- Trabalha com famílias de funções *hash*:
  - Supondo que se inicie com h(k) = k mod M, onde M é o número de *buckets*
  - Os *buckets* são inicialmente endereçados de 0 a M-1

Algoritmo de inserção em *hashing* linear

- Quando ocorrer um *overflow* em algum *bucket* (qualquer *bucket*), cria-se mais um *bucket* ao final do espaço de endereçamento
  - A primeira vez será o *bucket* M
- As entradas do *bucket* 0 são redistribuídas entre o *bucket* 0 e o *bucket* M utilizando a função
  - h(k) = k mod **2**M
  - Se o *overflow* tiver ocorrido em um *bucket* diferente do *bucket* 0, é criado um bloco de *overflow* para ele
  - No próximo *overflow*, será criado o *bucket* M+1 e as entradas do *bucket* 1 são redistribuídas entre o *bucket* 1 e o *bucket* M + 1
  - O algoritmo segue nesta rodada até ser criado o *bucket* 2M – 1
- No próximo *overflow*, passar-se-á a utilizar a função
  - h(k) = k mod **3**M

e o ciclo se repetirá até a criação do *bucket* 3M - 1

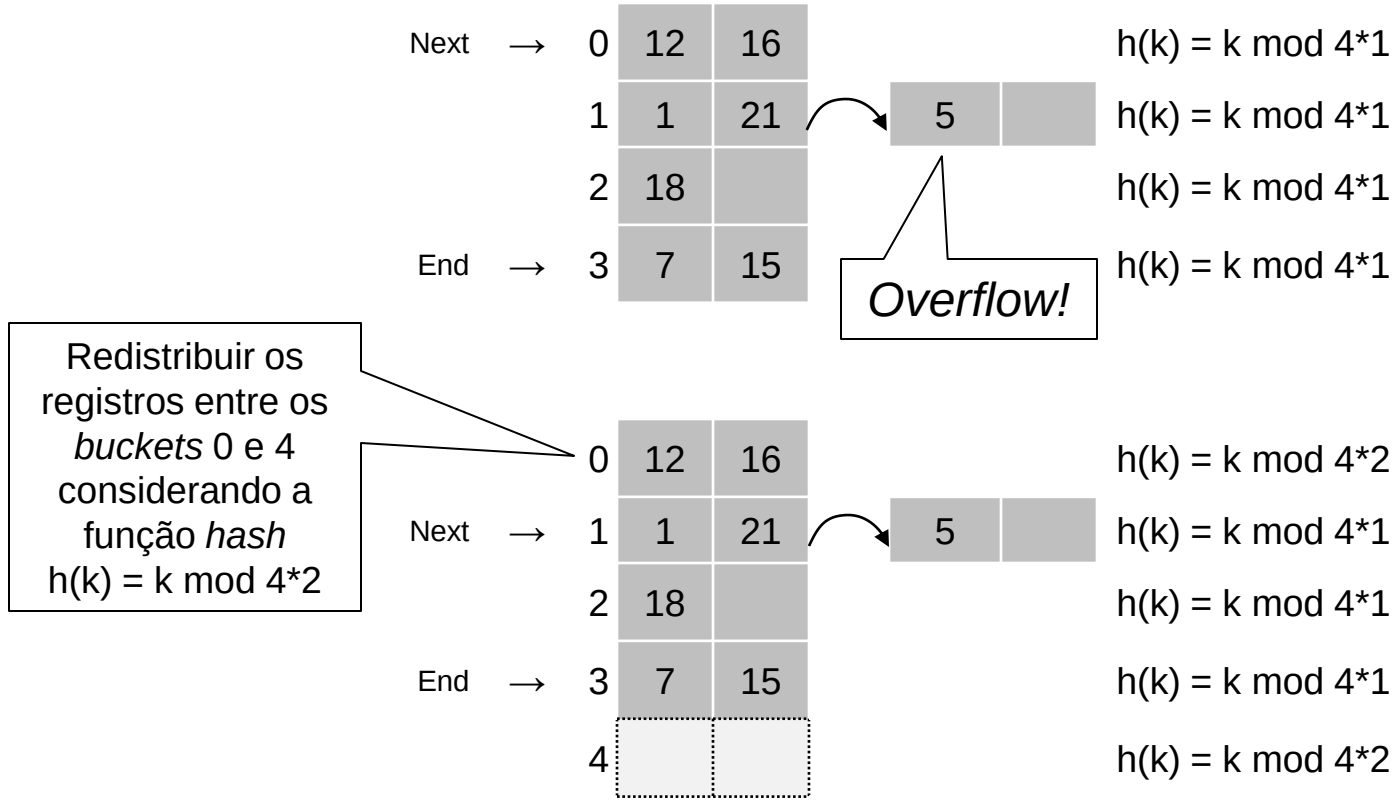
Exemplo de *hashing* linear

- Considere uma estrutura de acesso em *hash* linear com inicialmente 4 *buckets* com capacidade para 2 registros em cada *bucket*
- Considere a inserção dos registros com valores de chave de busca 1, 7, 12, 15, 16, 18 e 21
- Considere que esta estrutura em *hash* utiliza a seguinte família de funções *hash*
  - *h(k) = k mod 4*1 → h(k) = k mod 4*2 → h(k) = k mod 4*3 → ...*
- A estrutura *hash* estaria assim populada inicialmente

Next	→	0	12	16	h(k) = k mod 4*1
		1	1	21	h(k) = k mod 4*1
		2	18		h(k) = k mod 4*1
End	→	3	7	15	h(k) = k mod 4*1

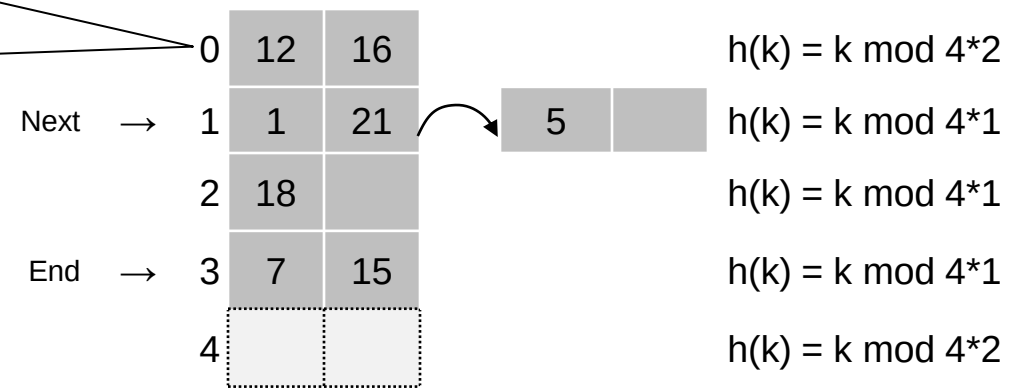
Exemplo de *hashing* linear

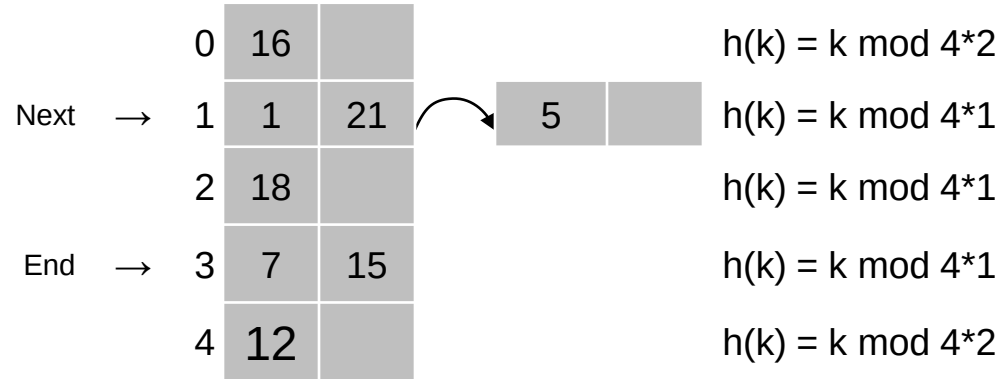
Após a inserção do registro com chave de busca 5 → 5 mod 4 = 1



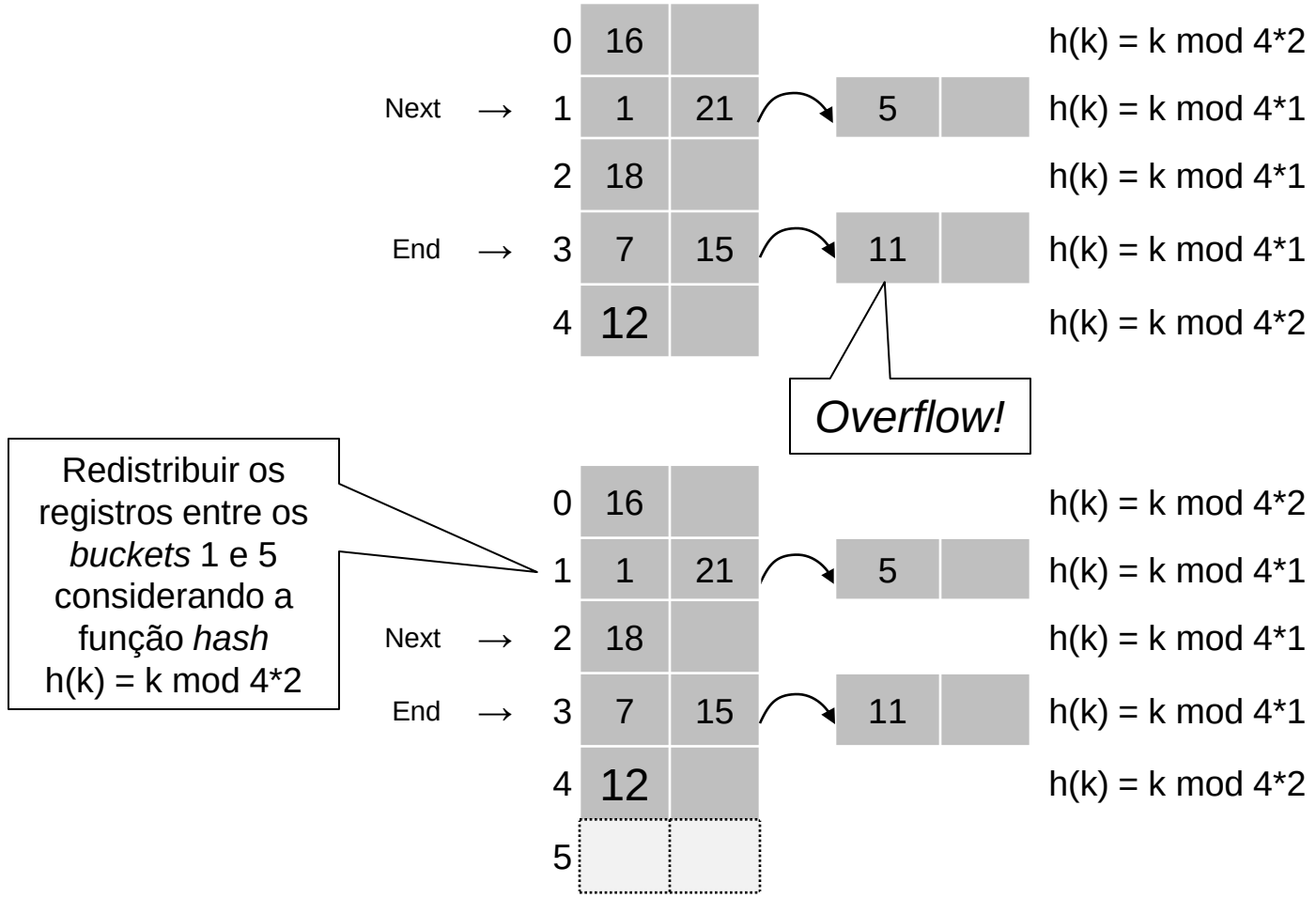
Exemplo de *hashing* linear

h(12) = 12 mod 4*2
h(12) = 4
h(16) = 16 mod 4*2
h(16) = 0



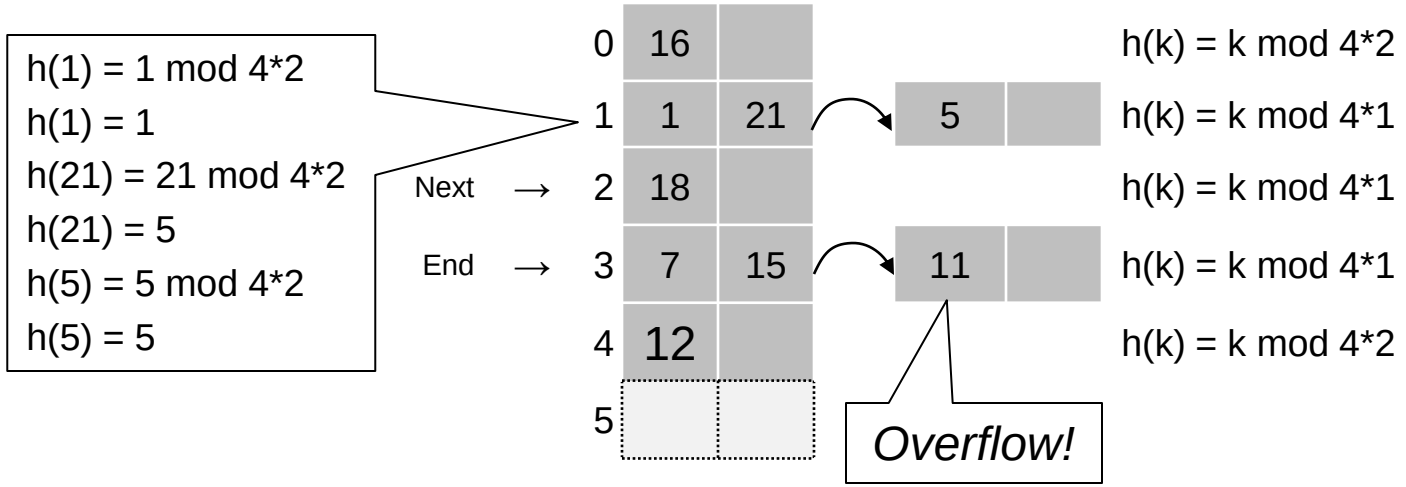


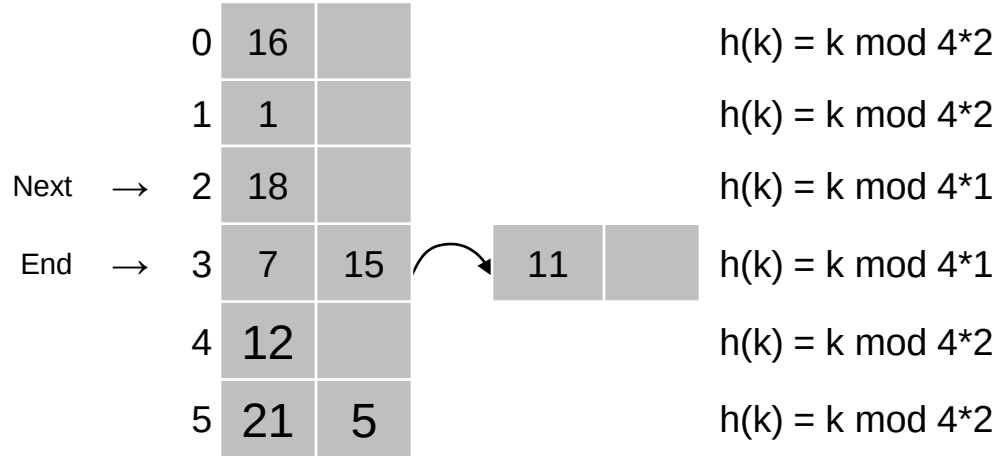
Exemplo de *hashing* linear



Inserir o registro com chave de busca 11
Aplicar
h(k) = k mod 4*1
h(11) = 11 mod 4*1 = 3
Como
Next < 3 < End, insere no *bucket* 3 sem aplicar a próxima função *hash* da família

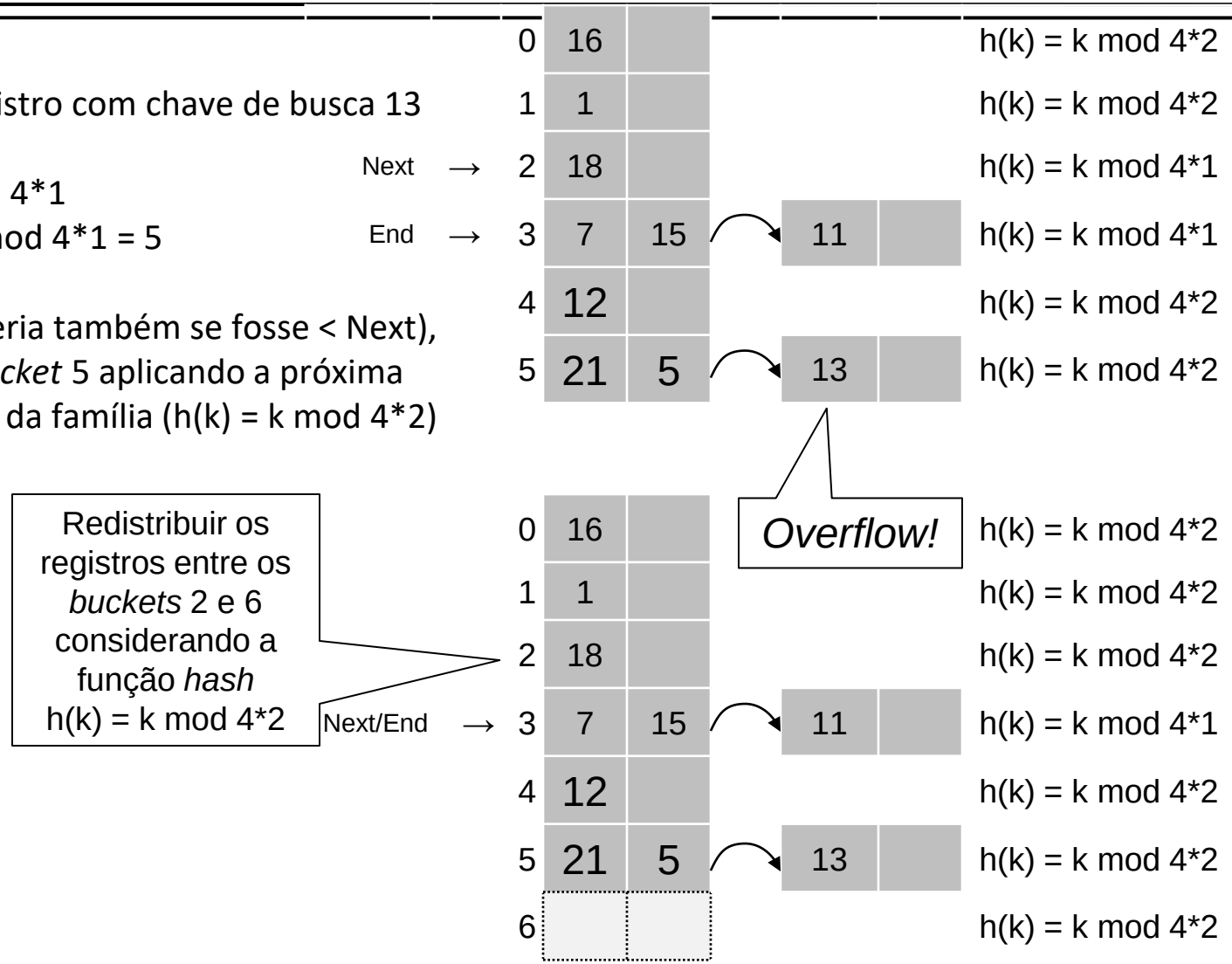
Exemplo de *hashing* linear



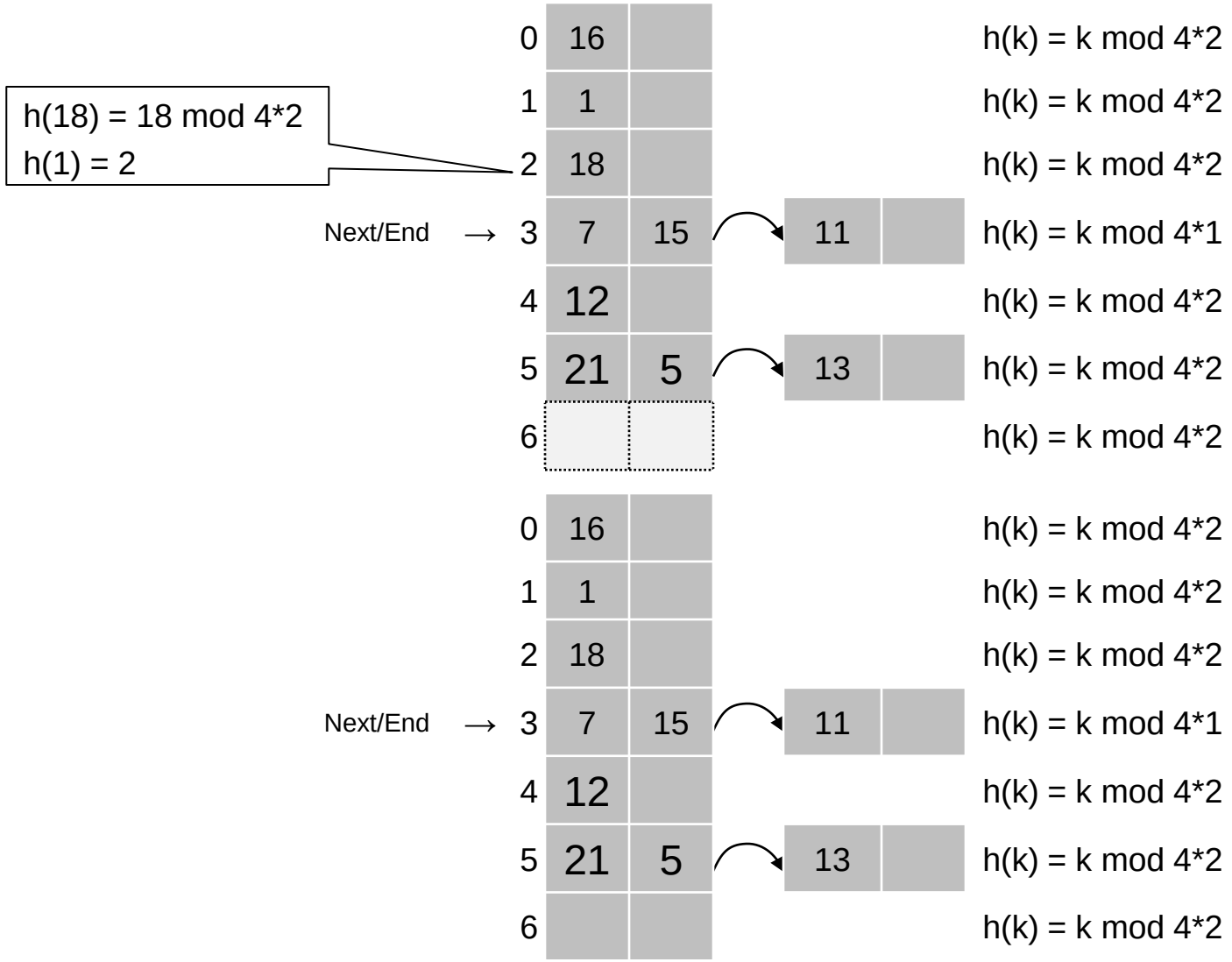


Exemplo de *hashing* linear

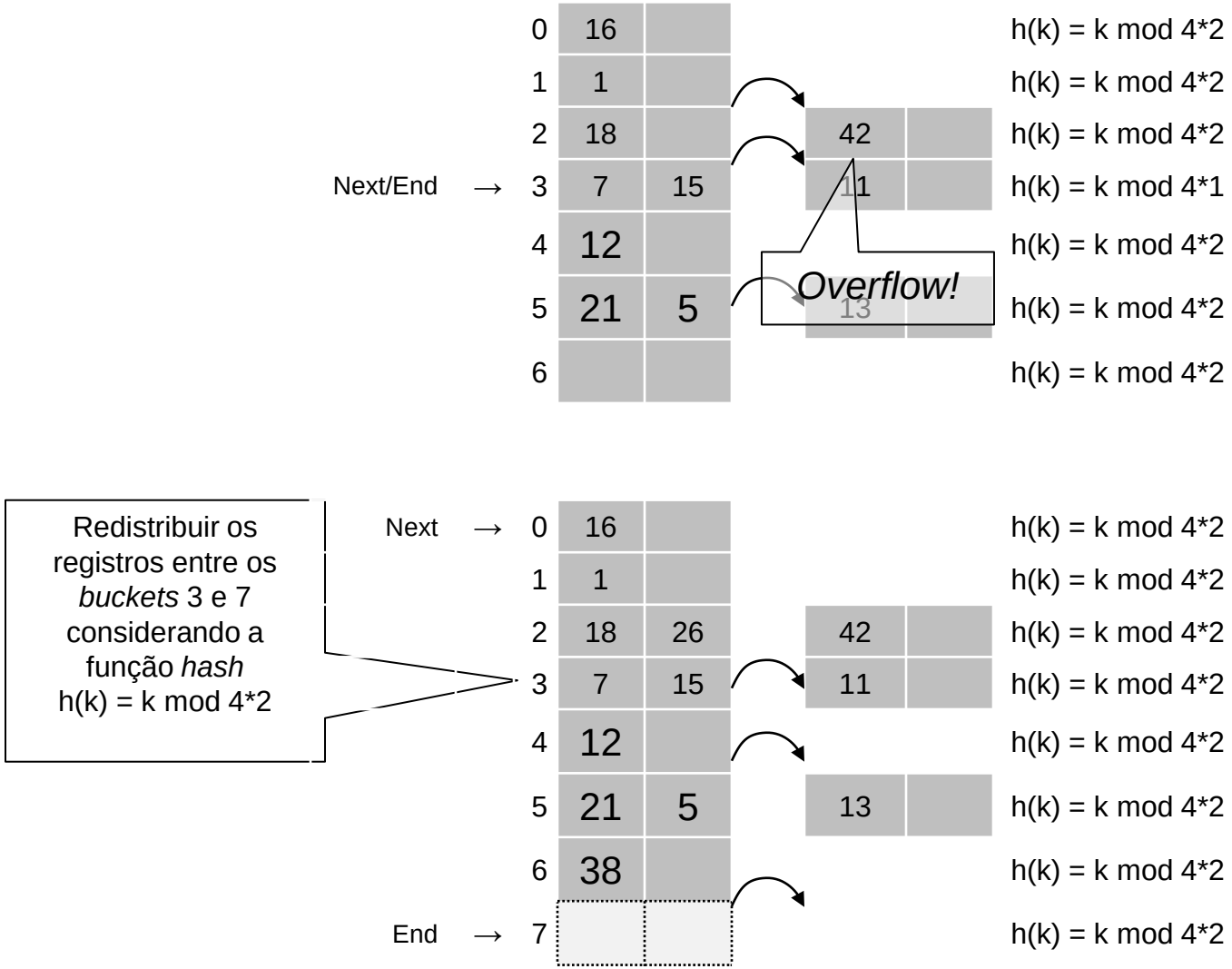
Inserir o registro com chave de busca 13
Aplicar
h(k) = k mod 4*1
h(13) = 13 mod 4*1 = 5
Como
5 > End (valeria também se fosse < Next),
insere no *bucket* 5 aplicando a próxima
função *hash* da família (h(k) = k mod 4*2)



Exemplo de *hashing* linear



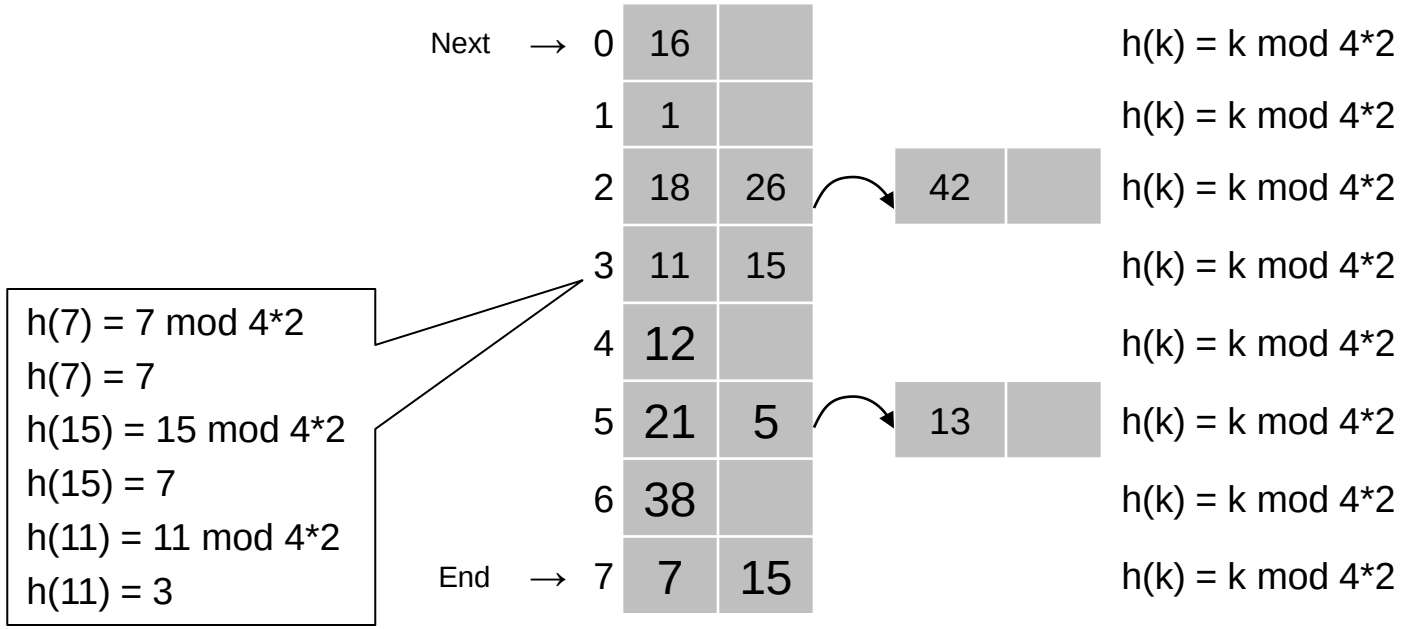
Exemplo de *hashing* linear



Inserir os registros com chaves de busca 26, 38 e 42
Aplicar
h(k) = k mod 4*1
h(26) = 2
h(38) = 6
h(42) = 2
Como
2 < Next, insere no *bucket* 2 aplicando a próxima função *hash* da família (h(k) = k mod 4*2)

Exemplo de *hashing* linear

- Um ciclo de crescimento do espaço de endereçamento do *hash* foi concluído
  - O número de *buckets* foi dobrado



Índices *bitmap*

- Um índice *bitmap* é um mapa de bits para cada um dos valores diferentes da chave, correspondendo às linhas da tabela (registros)
- Se a linha possui aquele valor de chave, no *bitmap* daquele valor o seu bit será 1
- Índices *bitmap* se prestam à execução eficiente de operações lógicas (*and* ou *or*) entre condições utilizadas no *where*
- Linhas que não satisfazem a expressão são rapidamente eliminadas o que pode aumentar muito o desempenho das consultas

Exemplo

Tabela de Clientes

Cliente	Estado Civil	REGIÃO	Sexo	Nivel Renda
101	solteiro	SUL	M	A
102	casado	SUDESTE	F	B
103	casado	SUDESTE	M	C
104	divorciado	NORDESTE	M	A
105	Solteiro	SUL	F	B
106	casado	SUL	F	A

REGIÃO='SUL'	REGIÃO=NORDESTE	REGIÃO='SUDESTE'
1	0	0
0	0	1
0	0	1
0	1	0
1	0	0
1	0	0

Índice Bitmap para Região

NIVEL_RENDA='A'	NIVEL_RENDA='B'	NIVEL_RENDA='C'
1	0	0
0	1	0
0	0	1
1	0	0
0	1	0
1	0	0

Índice Bitmap para Nível de Renda

Índice *bitmap*

- Benefícios
  - Redução de espaço para armazenamento do índice se valores não têm muita dispersão
  - Redução do tempo de resposta em consultas que utilizam operações complexas de seleção
- Quando Usar
  - Aplicações de *data warehousing*
  - Consultas com operadores lógicos (and, or, in especialmente) e operações de igualdade
  - Colunas com baixa cardinalidade (diversidade de valores)

Índice *bitmap*

- Quando não usar
  - Não é adequado para sistemas de OLTP (*Online Transaction Processing*)
    - Devido ao alto custo de atualização do índice
  - Não é adequado para consultas com operadores diferentes de "="
  - Colunas com grande cardinalidade (colunas com valores únicos)

Dúvidas?