



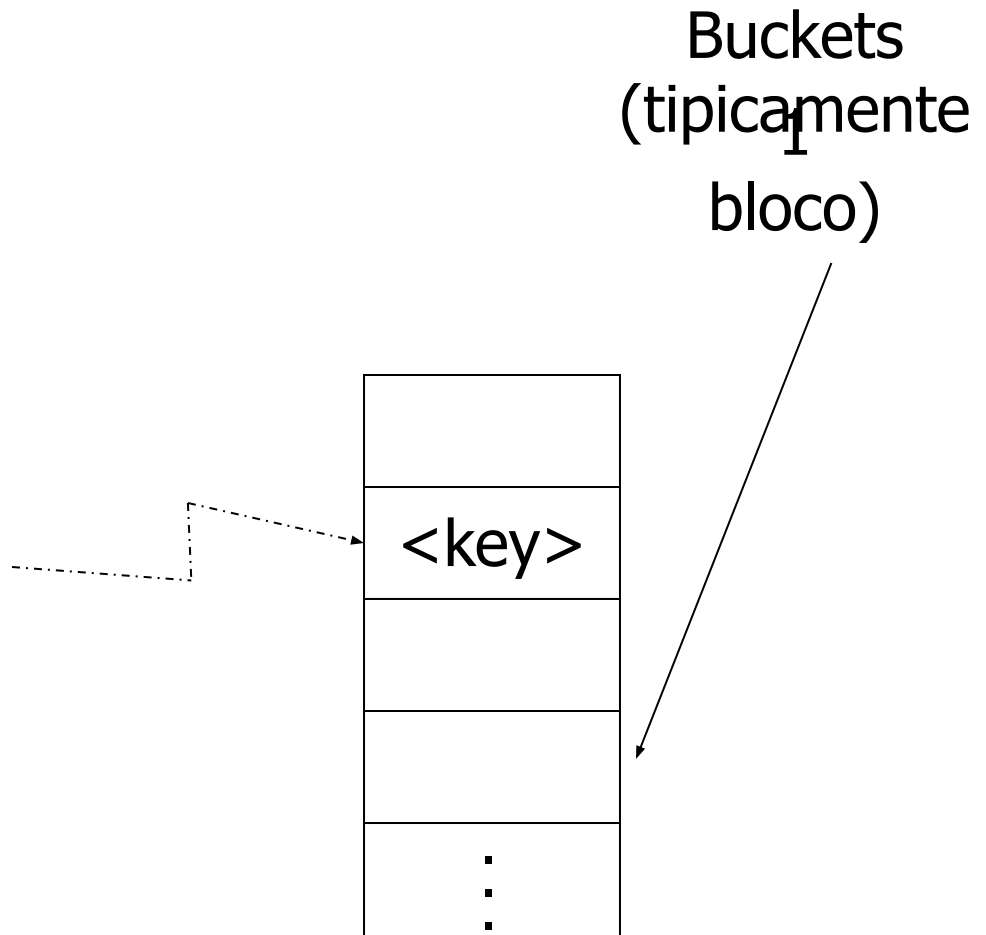
Banco de Dados II

Hashing e mais

Traduzido e adaptado de Hector Garcia-Molina
(Database Systems – The Complete Book 2a Ed)

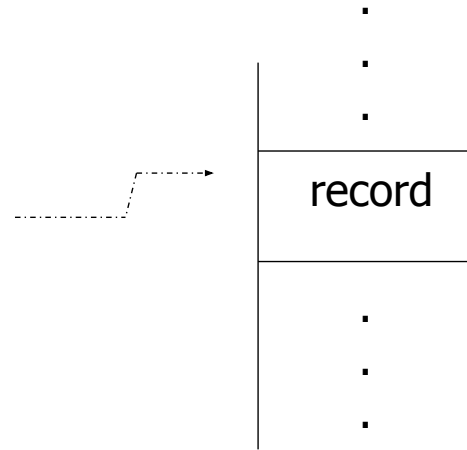
Hashing

key \rightarrow h(key)

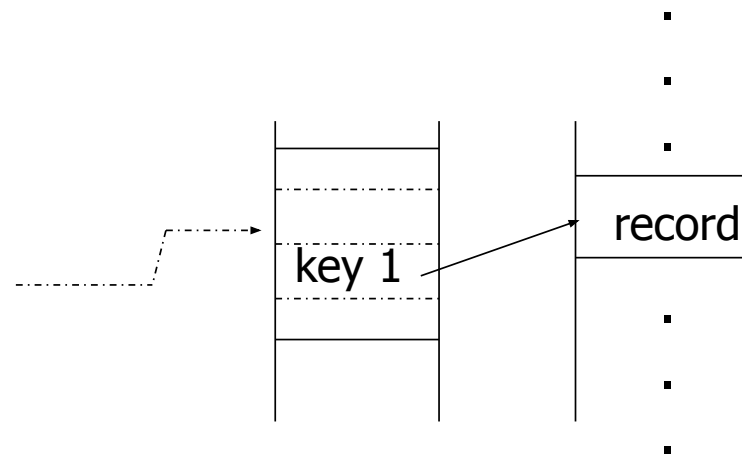


Duas alternativas

(1) $\text{key} \rightarrow \text{h}(\text{key})$



(2) $\text{key} \rightarrow \text{h}(\text{key})$



Índice

Exemplo de função de hashing

- Key = " $x_1 x_2 \dots x_n$ " onde n é o número de bytes da cadeia
- Se temos k buckets
- $h: (x_1 + x_2 + \dots + x_n) \text{ módulo } k$
 - Todas as chaves serão alocadas em um bucket b_i ($0 \leq i < k$).

- Esta pode não ser a melhor função...
- Leiam Knuth (Art of Computer Programming, Volume 3: Sorting and Searching) se vocês quiserem realmente uma boa função de *hashing*

Boa função
de hashing:

✓ O número **esperado** de
chaves/bucket é o mesmo
para os buckets

Dentro de um bucket:

- As chaves devem estar ordenadas?

Sim, Se o tempo de CPU é crucial &
Inserts/Deletes não são frequentes

Hashing Estático

(número de buckets não muda)

Exemplo: 4 buckets
2 registros por bucket

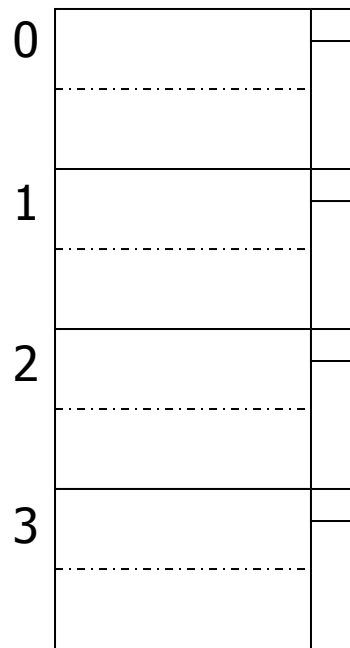
INSERT:

$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$



Exemplo: 4 buckets
2 registros por bucket

INSERT:

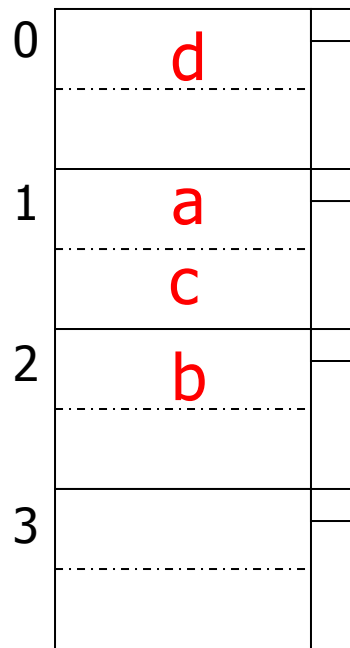
$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$

$h(e) = 1$



Exemplo: 4 buckets
2 records por bucket

INSERT:

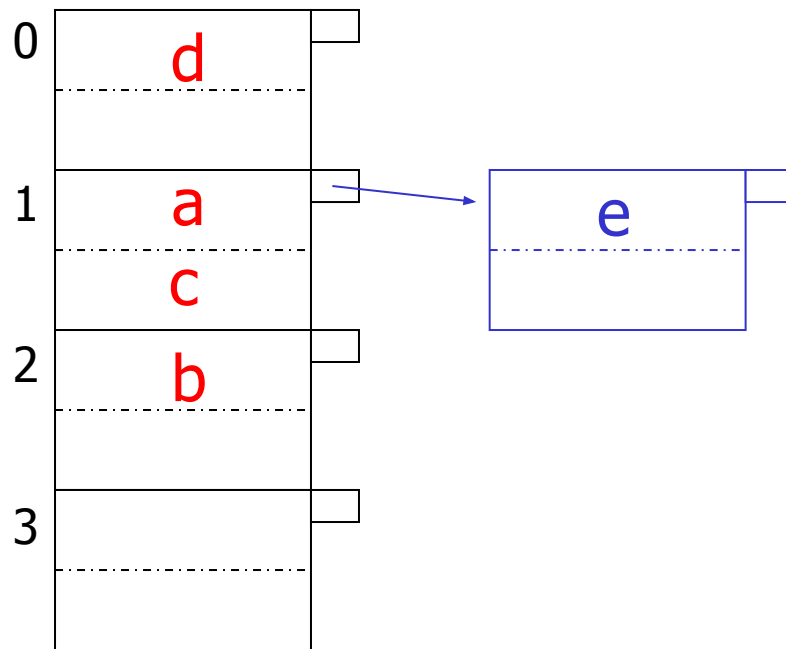
$h(a) = 1$

$h(b) = 2$

$h(c) = 1$

$h(d) = 0$

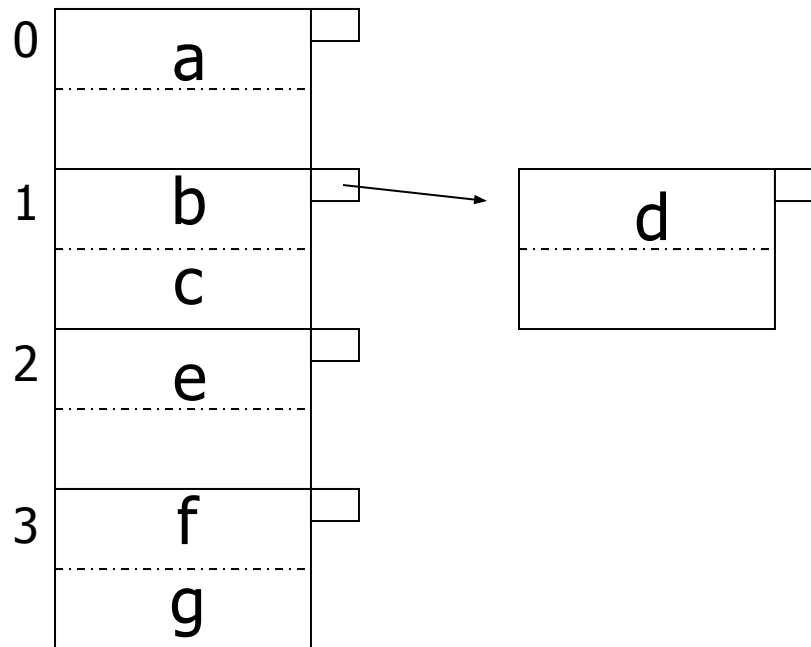
$h(e) = 1$



Exemplo: exclusão

Delete:

e
f



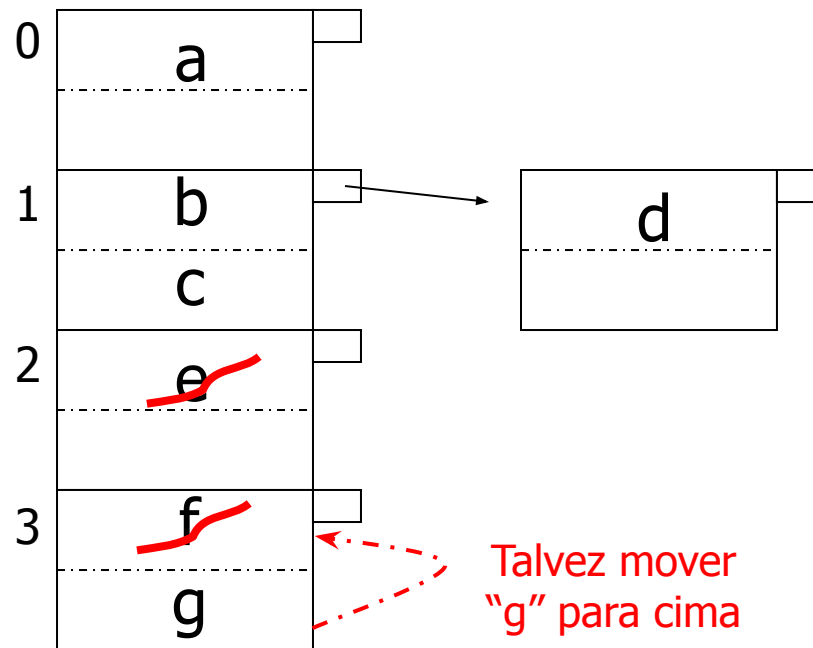
Exemplo: exclusão

Delete:

e

f

c



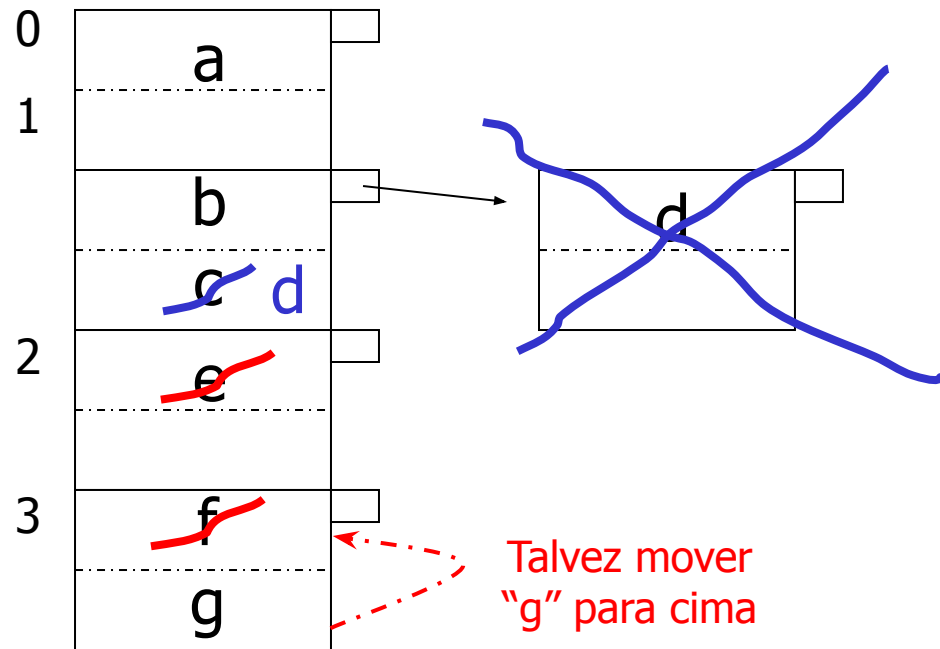
Exemplo: exclusão

Delete:

e

f

c



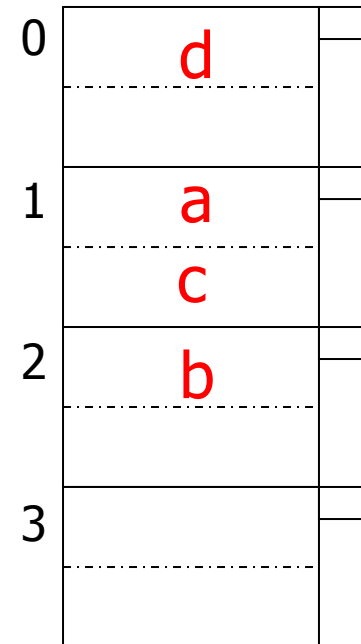
Regra geral

- Tentar manter o espaço de utilização entre 50% e 80%

$$\text{Utilização} = \frac{\text{\# keys}}{\text{buckets capacity}}$$

- Exemplo

$$\text{Utilização} = \frac{4}{8} \Rightarrow 50\%$$



Regra geral

- Tentar manter o espaço de utilização entre 50% e 80%

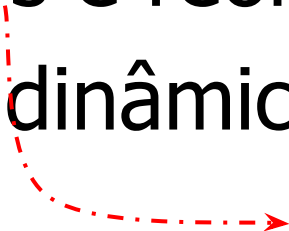
$$\text{Utilização} = \frac{\text{\# keys}}{\text{buckets capacity}}$$

- Se $< 50\%$, espaço desperdiçado
- Se $> 80\%$, signficante **overflow**
depende de quão boa é
a função de hashing &
 $\#$ keys por bucket

Como tratar o crescimento?

- Overflows e reorganização
- Hashing dinâmico

Como tratar o crescimento?

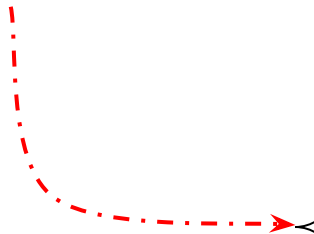
- Overflows e reorganização
 - Hashing dinâmico
- 
- Linear Probe Hashing

Linear Probe Hashing

- Uma tabela hashing enorme para tratar colisões
- Usa uma busca linear para tratar colisões
- Exemplo no quadro 🧐

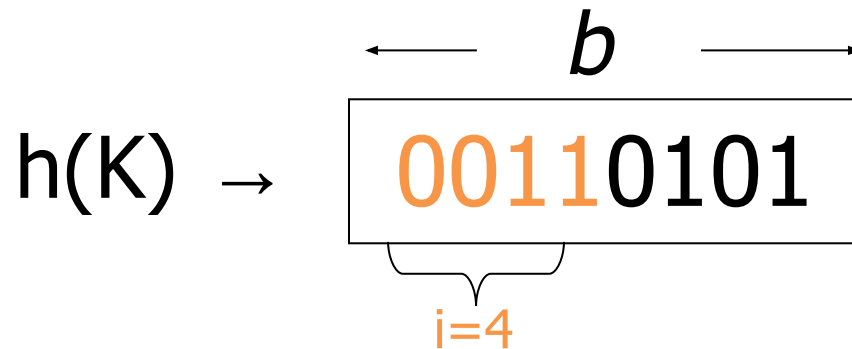
Como tratar o crescimento?

- Overflows e reorganização
- Hashing dinâmico

- 
- Extensível
 - Linear

Hashing extensível: duas ideias

(a) Use i bits b de saída pela função de hashing

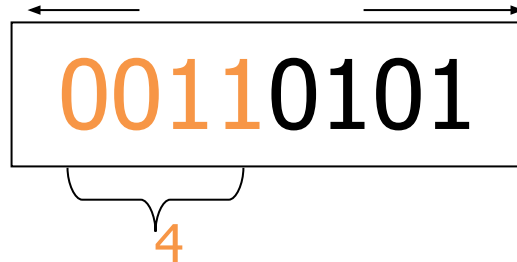


i começa com 1 e cresce com o tempo....

Hashing extensível:

(a) Use i bits b de saída pela função de hashing

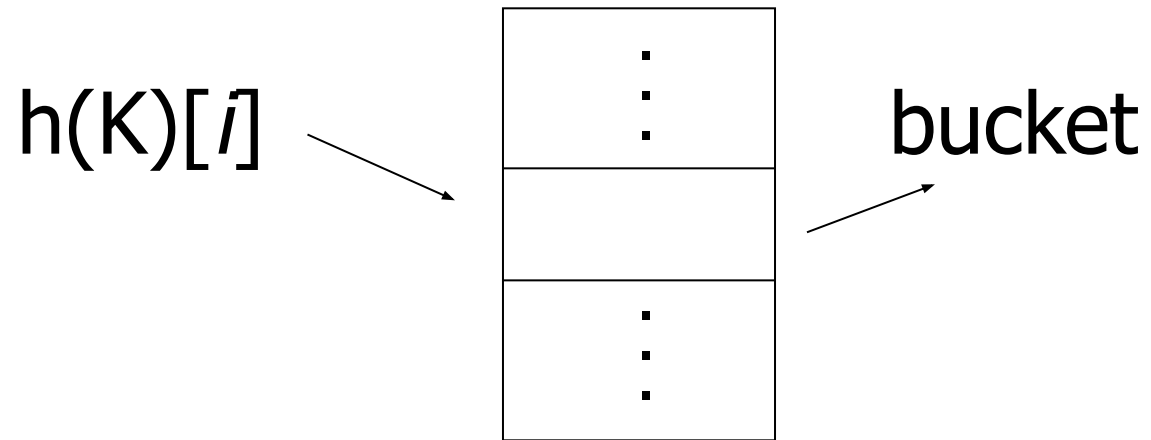
$h(K) \rightarrow$



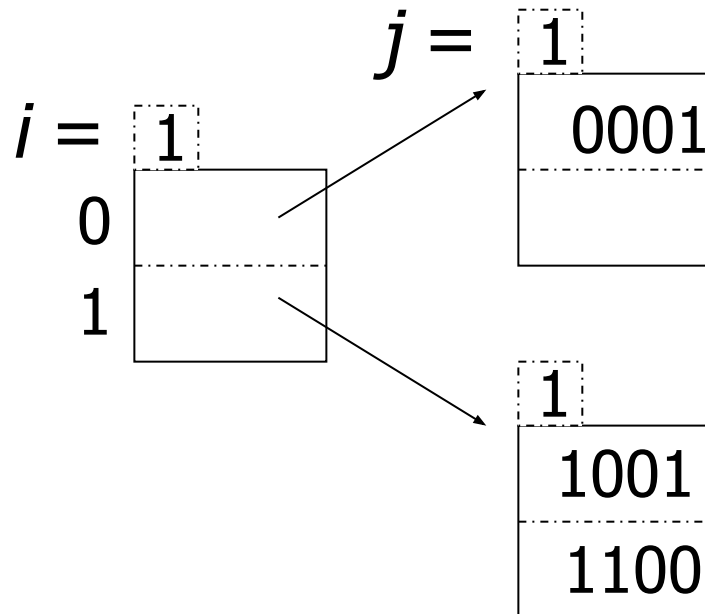
2 variáveis de controle:

- Global indicando o máximo de bits a considerar iniciando com 1
- Local indicando quantos bits o bucket considera

(b) Baseia-se em um diretório



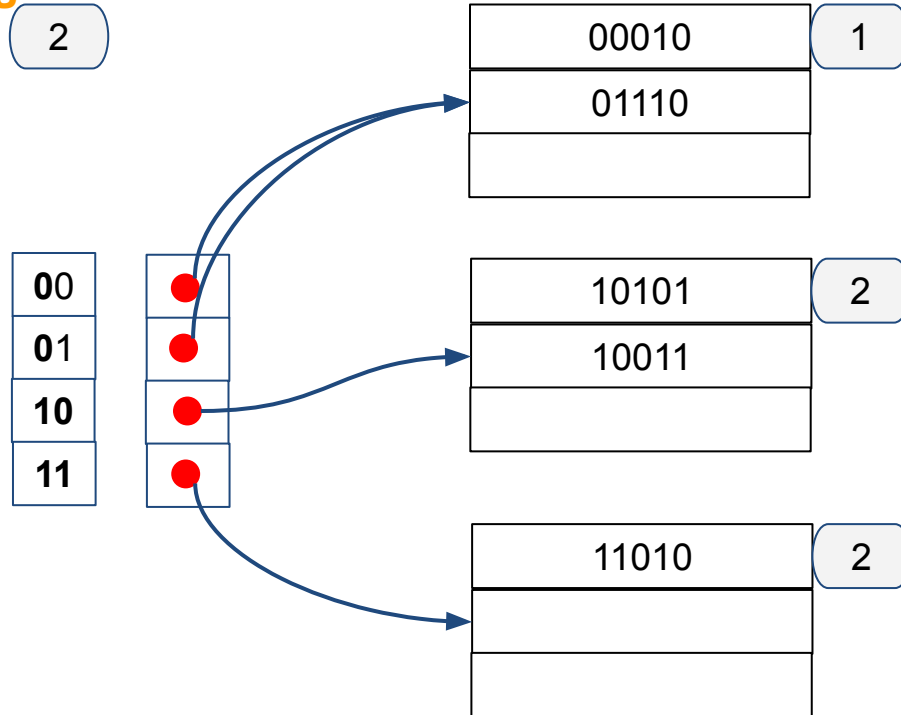
Exemplo: $h(k)=4$ bits (máx); 2 keys por bucket



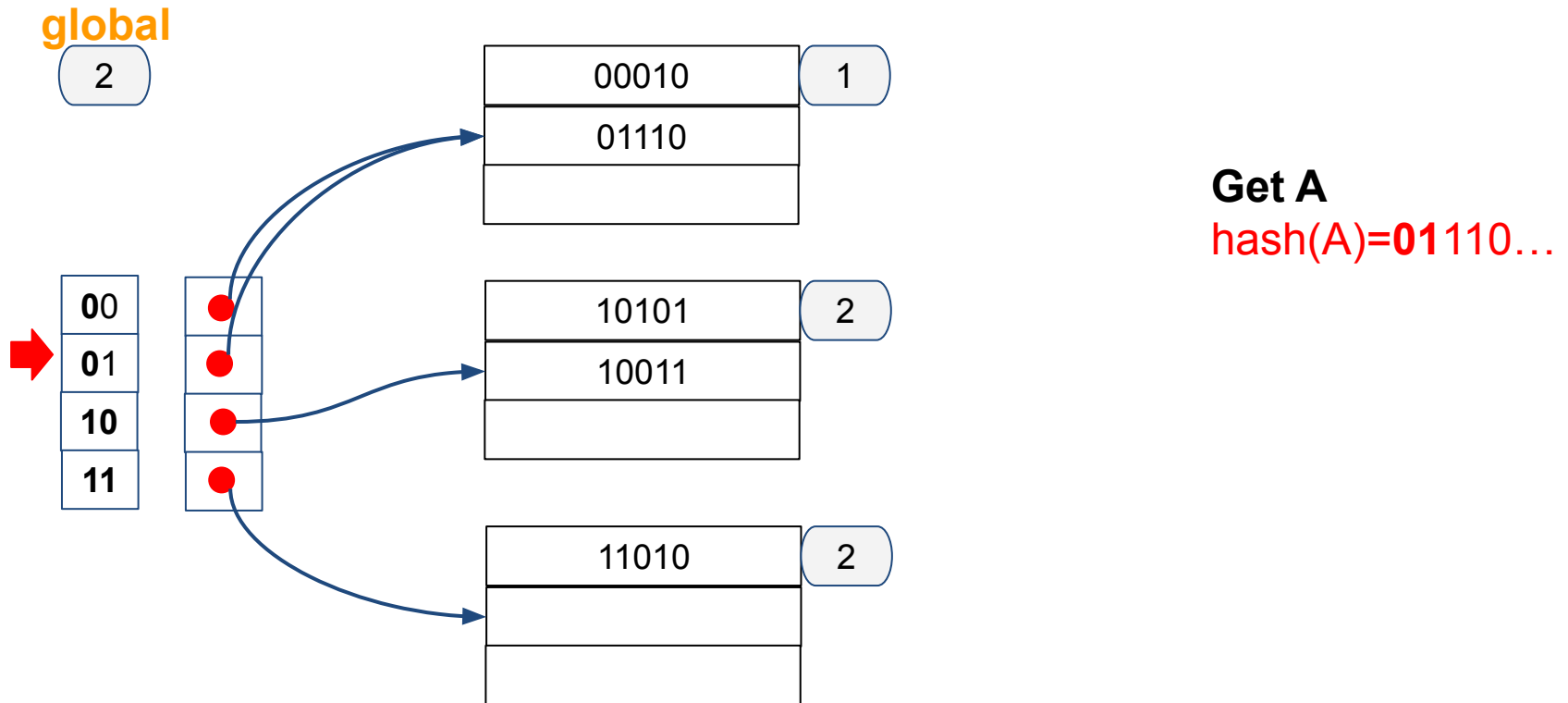
Insert 1010

Exemplo: $h(k)=4$ bits (máx); 3 keys por bucket

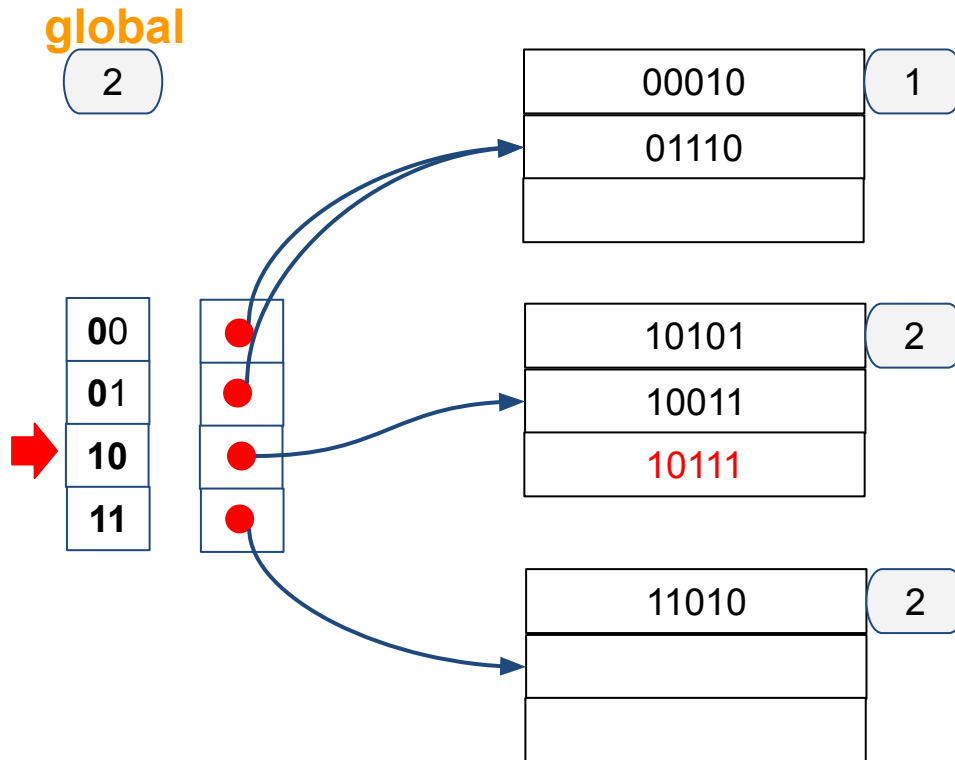
global



Exemplo: $h(k)=4$ bits (máx); 3 keys por bucket



Exemplo: $h(k)=4$ bits (máx); 3 keys por bucket



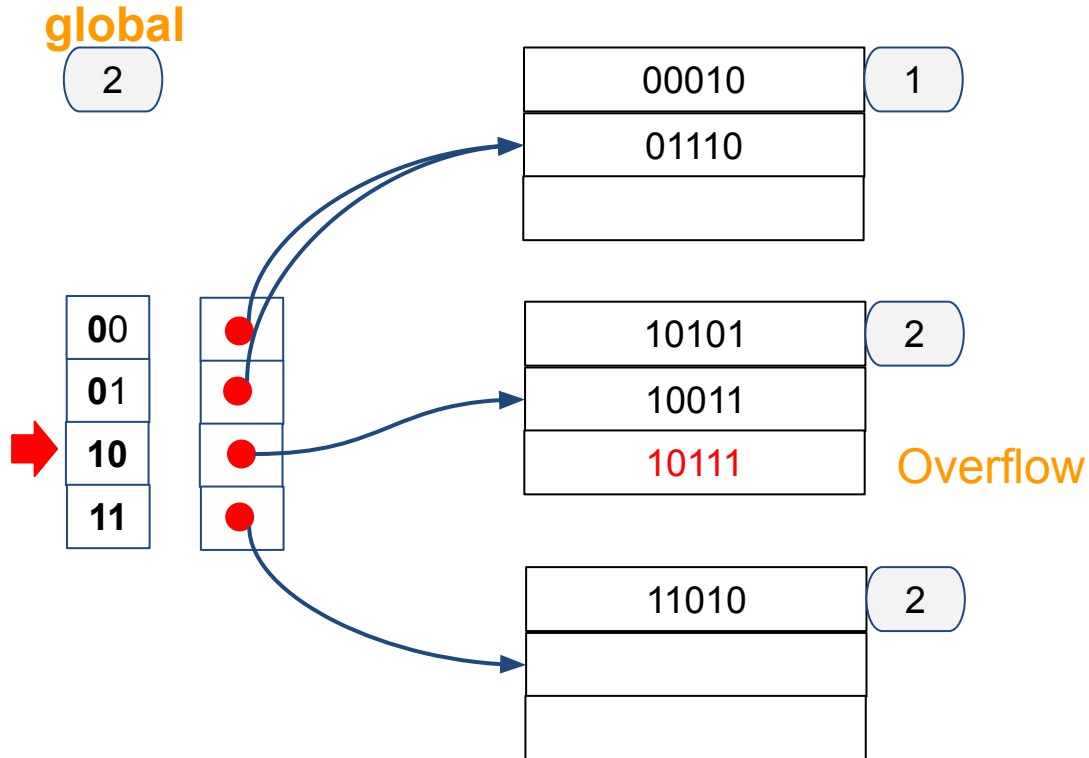
Get A

hash(A)=**01110**...

Put B

hash(B)=**10111**...

Exemplo: $h(k)=4$ bits (máx); 3 keys por bucket



Get A

hash(A)=**0**1110...

Put B

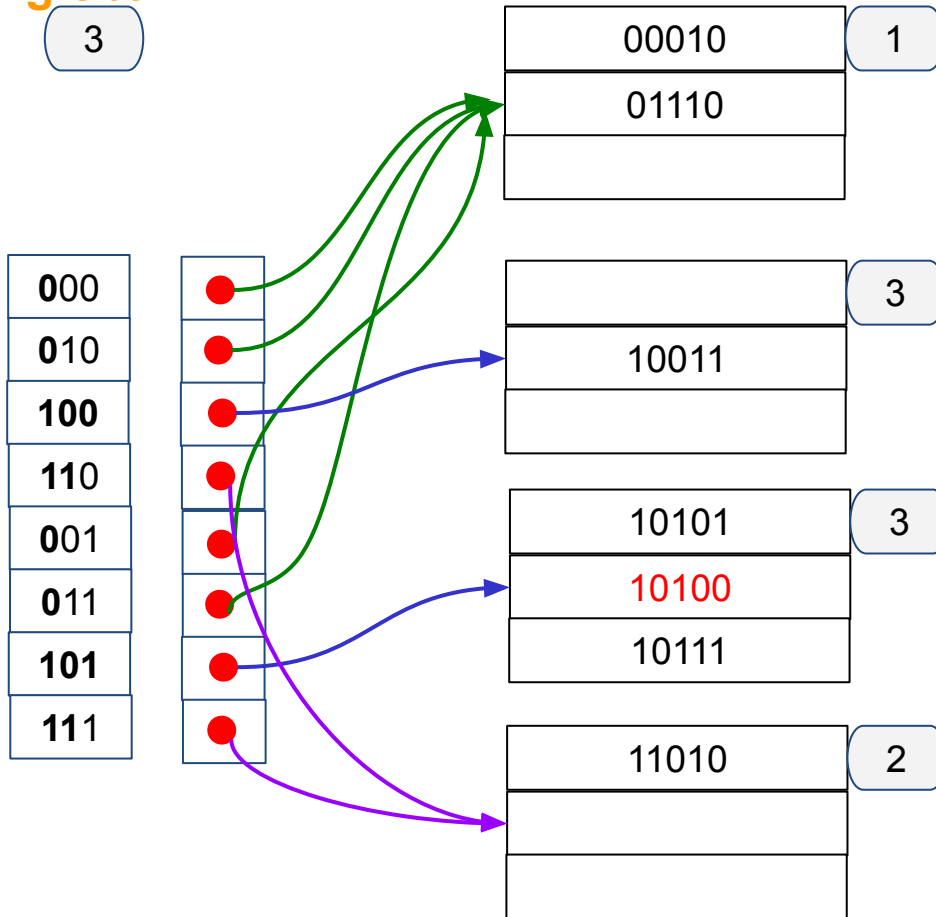
hash(B)=**1**0111...

Put C

hash(C)=**1**0100...

Exemplo: $h(k)=4$ bits (máx); 3 keys por bucket

global



Get A

hash(A)=**01110**...

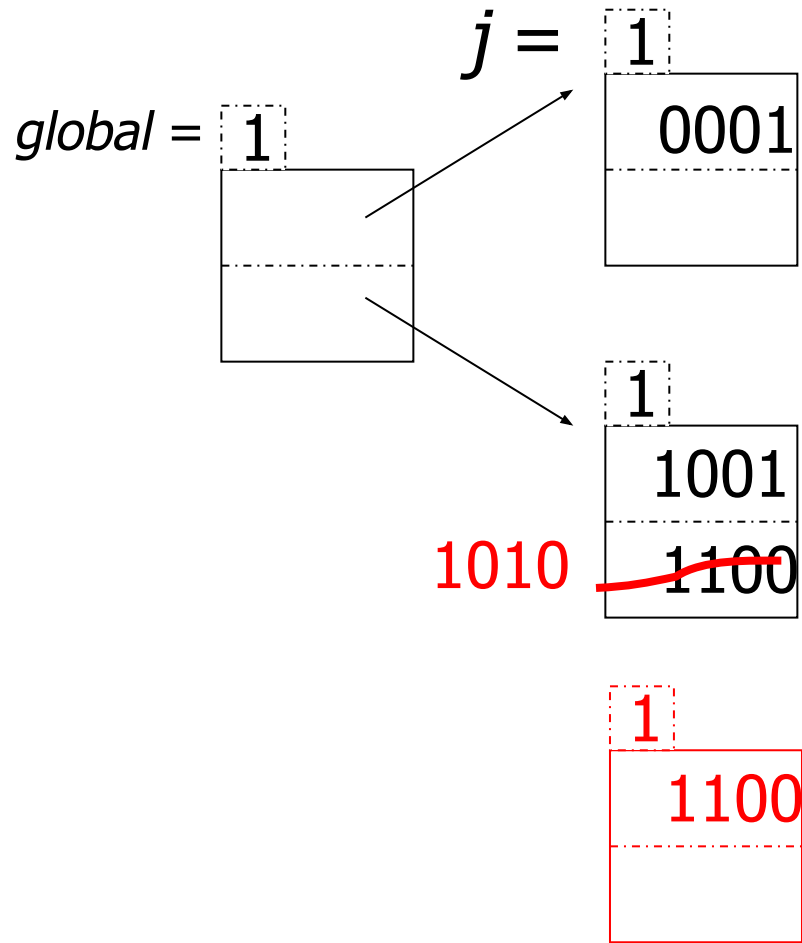
Put B

hash(B)=**10111**...

Put C

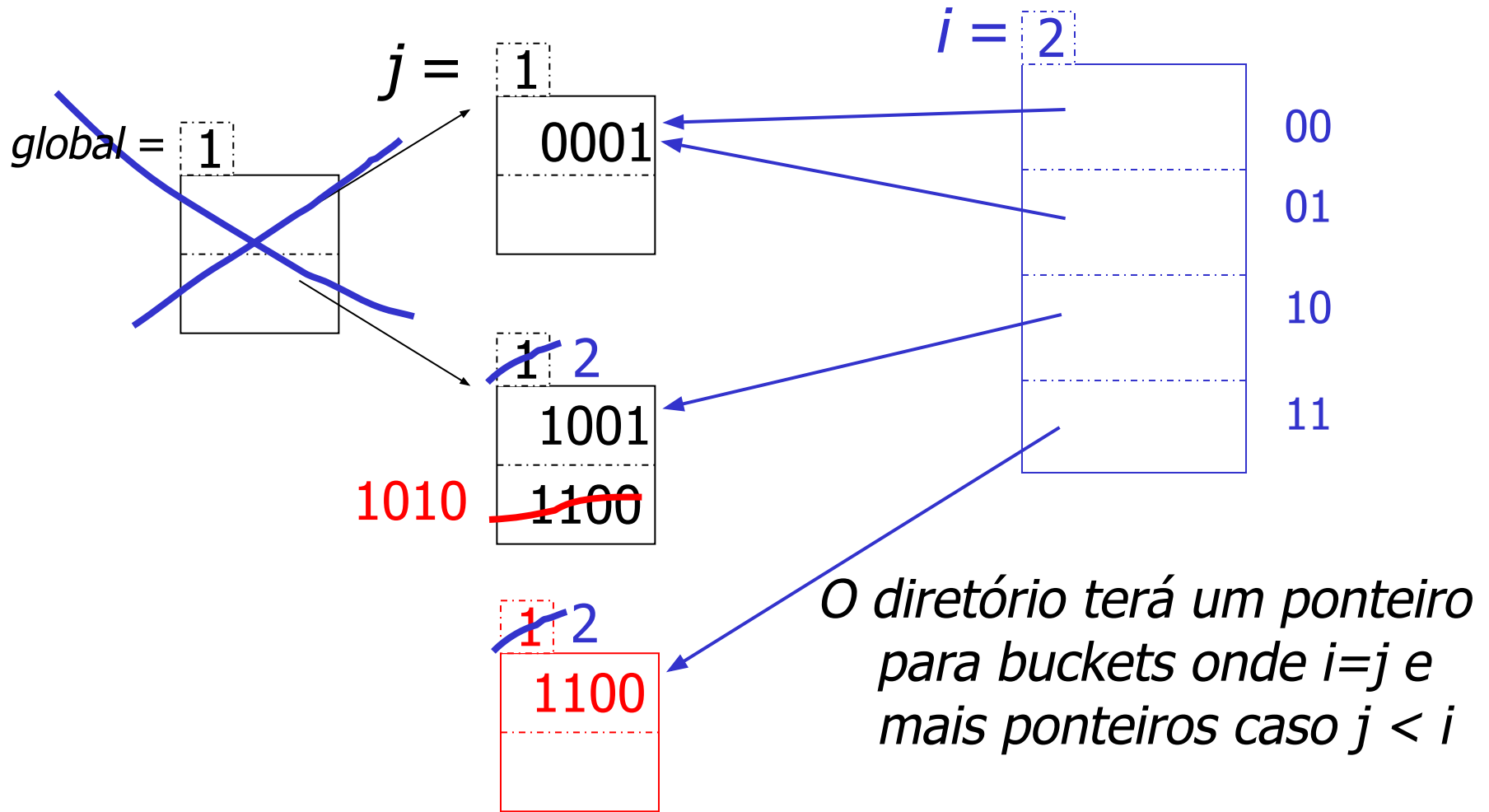
hash(C)=**10100**...

Exemplo: $h(k)=4$ bits (máx); 2 keys por bucket

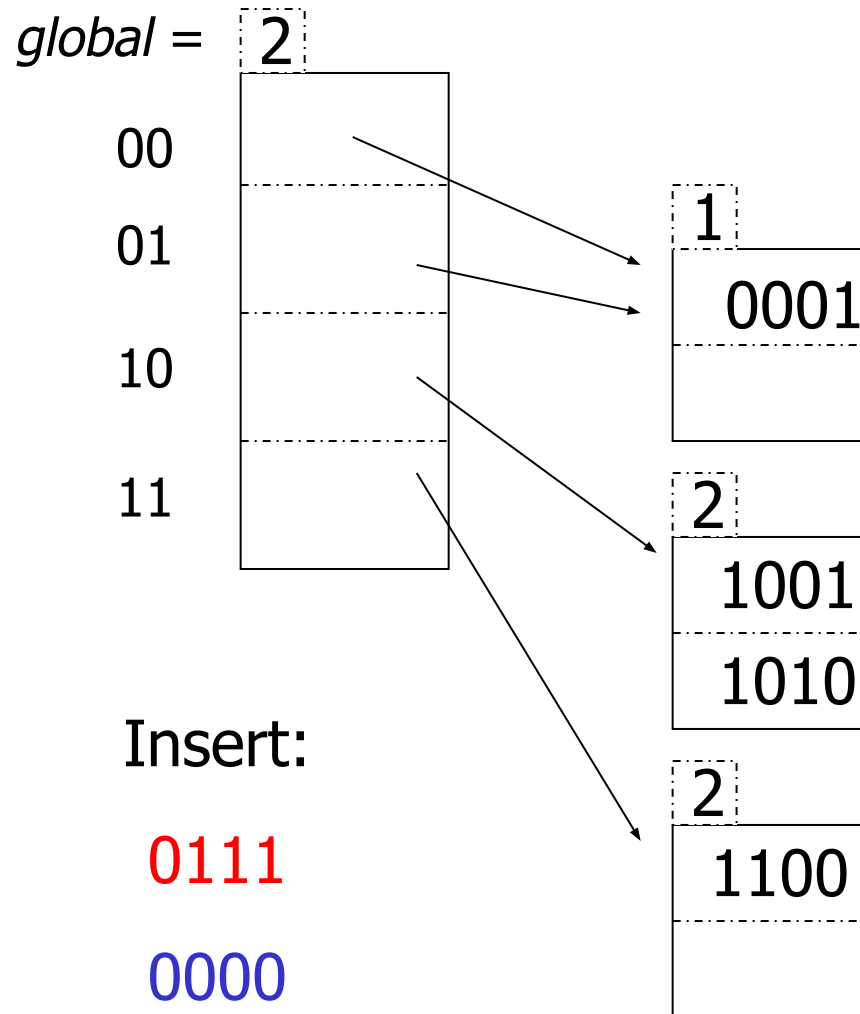


Como 1010 não cabe, cria-se um novo bucket e como $j=i$, deve-se expandir o diretório também.

Exemplo: $h(k)=4$ bits (máx); 2 keys por bucket



continuação



continuação

global =

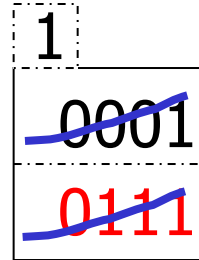
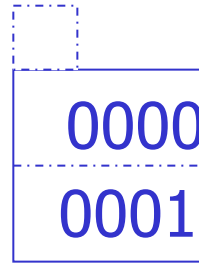
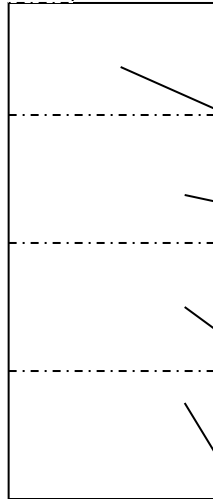
2

00

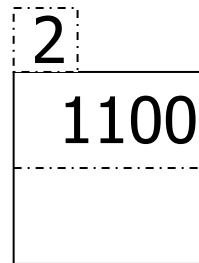
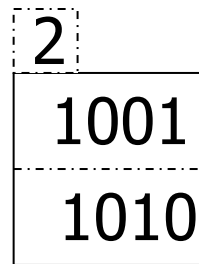
01

10

11



0111

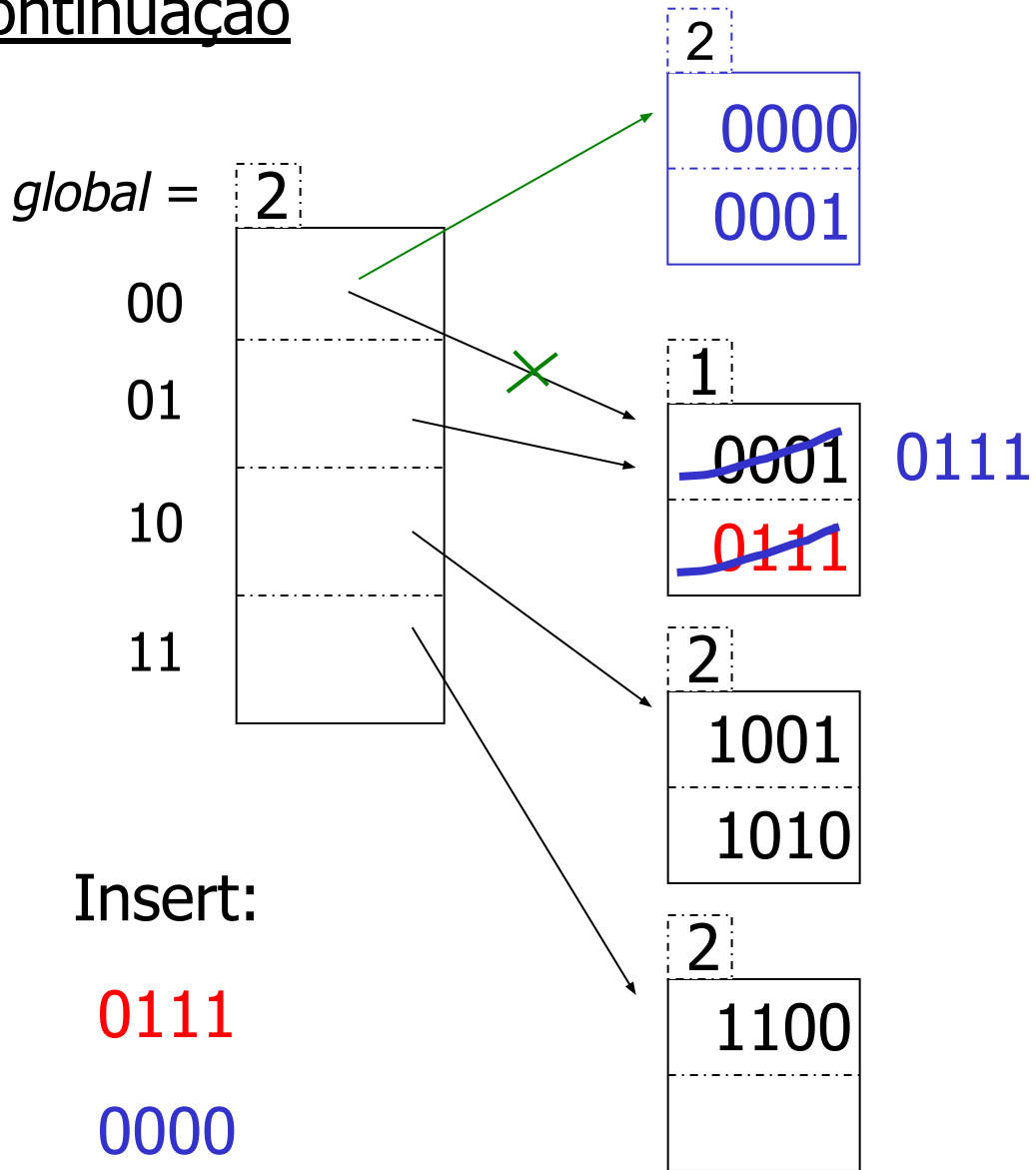


Insert:

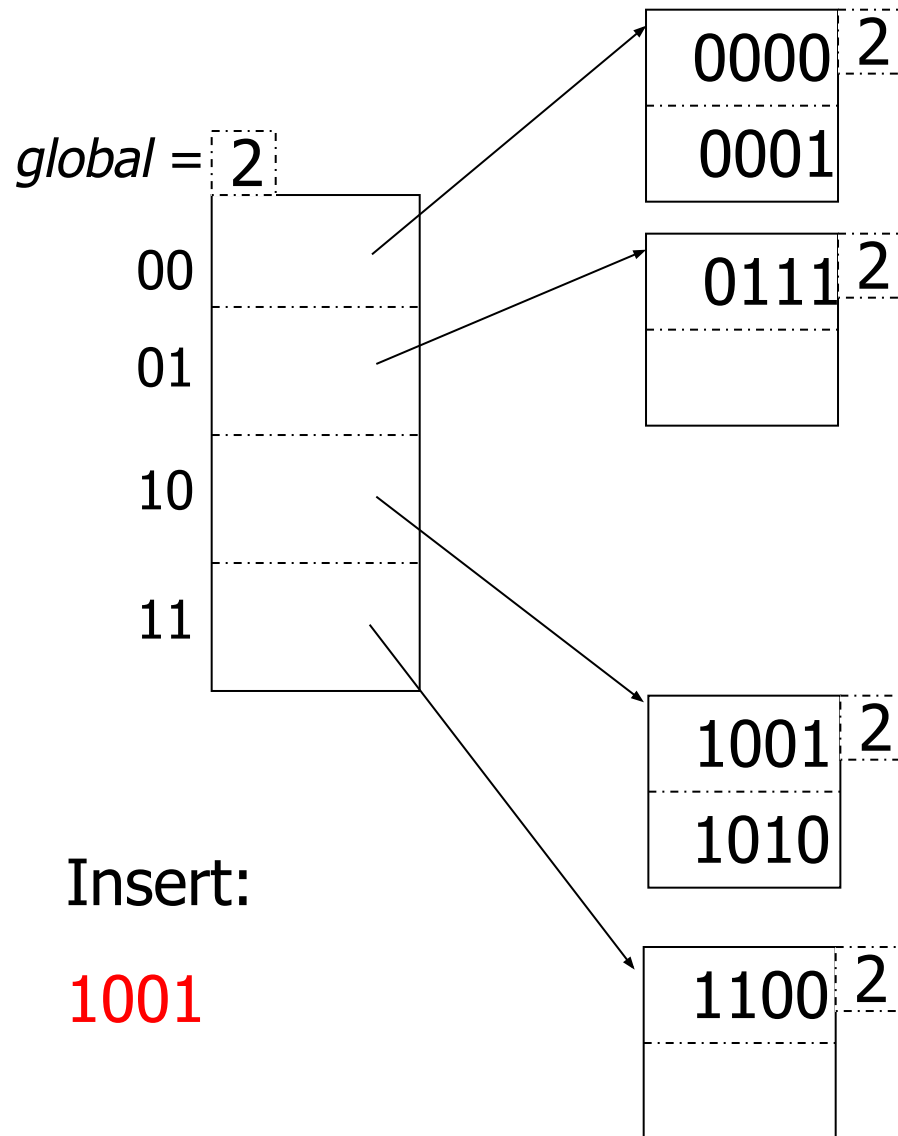
0111

0000

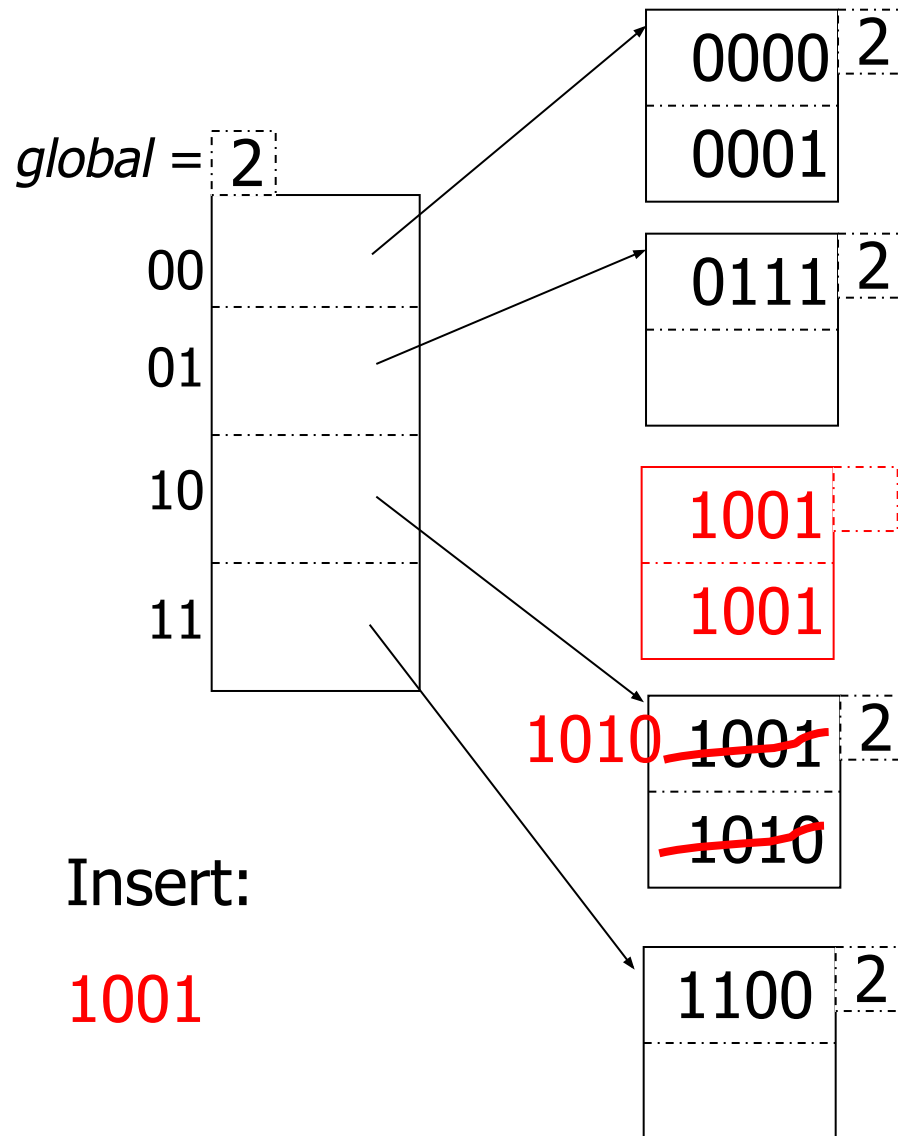
continuação



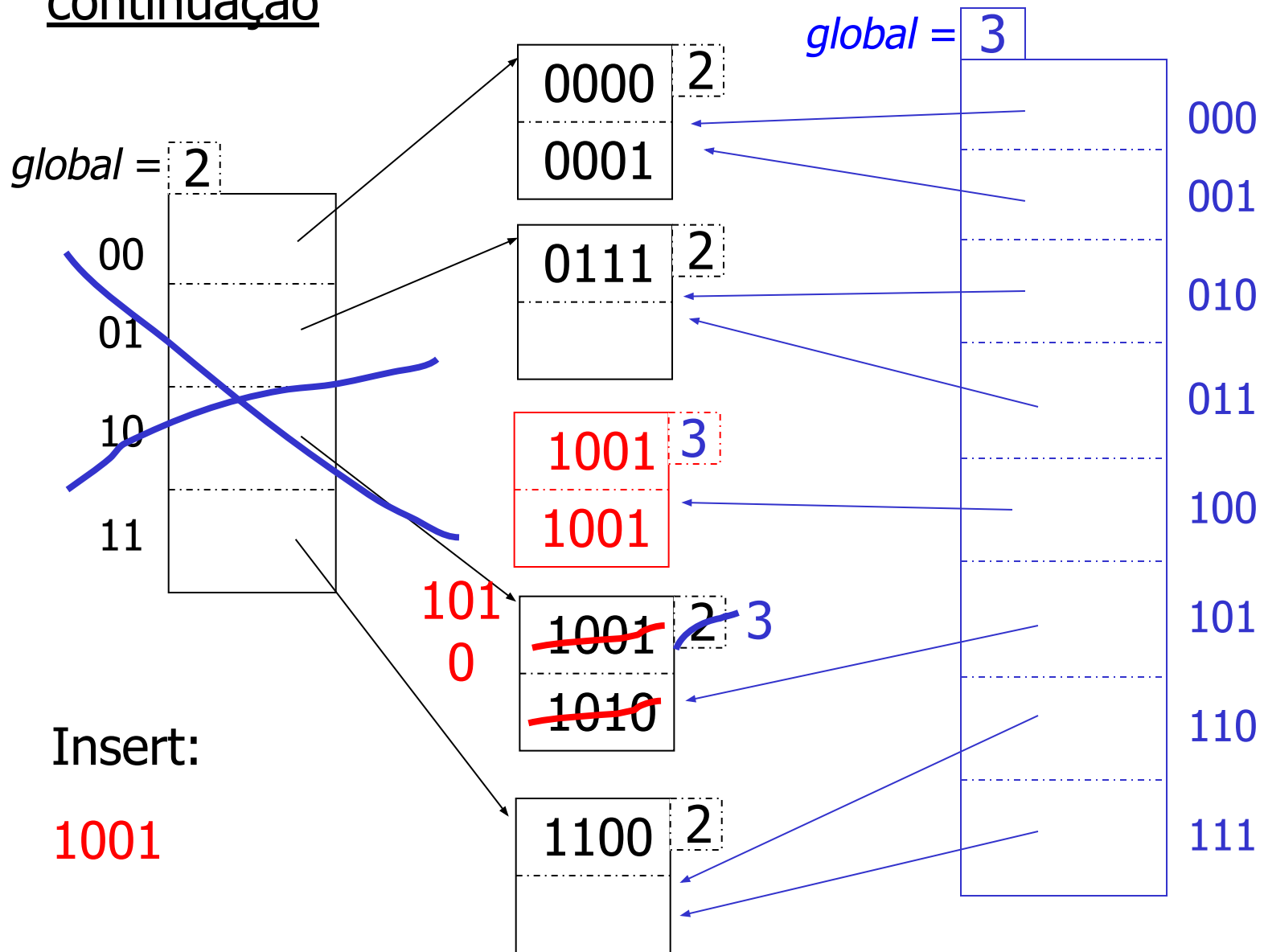
continuação



continuação



continuação



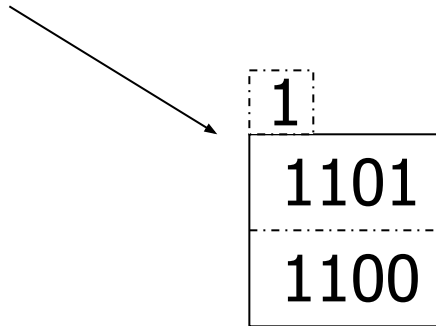
Hashing extensível: exclusão

- Sem fusão de blocos
- Fundir blocos e reduzir o diretório se possível
(operação reversa ao insert)

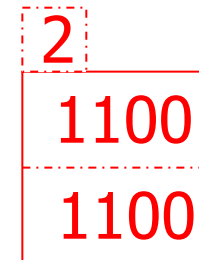
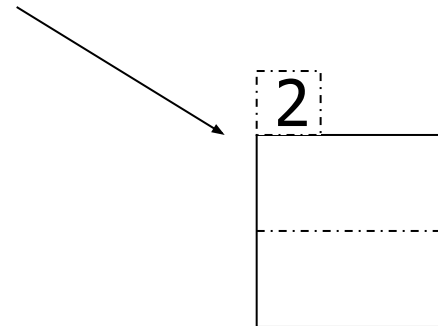
Nota: Ainda é necessário cadeias de overflow

- Exemplo: muitos registros com chaves duplicadas

insert
1100

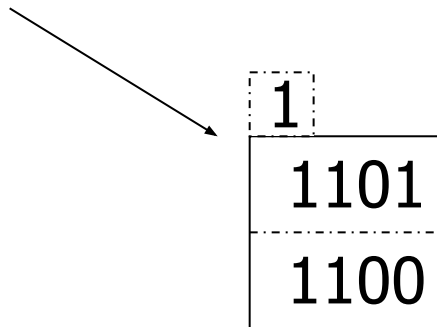


Se dividirmos:

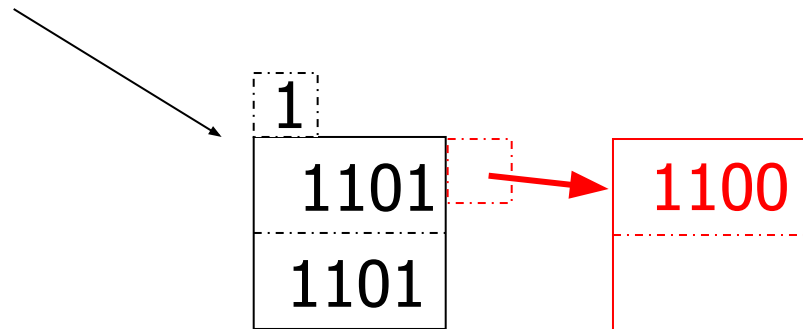


Solução: cadeias de overflow

insert
1100



Adiciona um bloco de overflow:



- ⊕ Pode manipular arquivos em crescimento
 - com menos espaços desperdiçados
 - sem reorganização completa

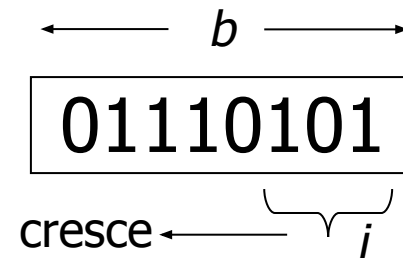
- ⊕ Pode manipular arquivos em crescimento
 - com menos espaços desperdiçados
 - sem reorganização completa
- ⊖ Indireção
 - (Menos mal se diretório na RAM)
- ⊖ Diretório duplica de tamanho
 - (Agora ele cabe, depois não cabe mais)

Linear hashing

- Outro esquema de hashing dinâmico

2 Estratégias:

- (a) Inicia com n buckets e função de hashing $key \% n$
- (b) Use i bits de menor ordem do hash



Linear hashing

- Estratégia (a)
 - Layout inicial: n buckets e um ponteiro apontando para o bucket que vai ser dividido
 - Bucket a ser dividido: quando o primeiro overflow ocorre, o bucket 0 é dividido e uma nova h é disponibilizada para os buckets divididos
 - O ponteiro é incrementado para o próximo bucket

Linear hashing

- Estratégia (a) $h_0 = \text{key} \% 4$

p → 0

4	8	12	16
---	---	----	----

1

1	5		
---	---	--	--

2

6	10	22	
---	----	----	--

3

3	7	15	19
---	---	----	----

Linear hashing

- Estratégia (a) $h_0 = \text{key} \% 4$

$p \rightarrow 0$

4	8	12	16
1	5		
6	10	22	
3	7	15	19

Insere 11

$$h_0 = \text{key} \% 4$$

$$h_1 = \text{key} \% 8$$

$p \rightarrow 1$

0	8	16		
1	1	5		
2	6	10	22	
3	3	7	15	19
4	4	12		

→

11			
----	--	--	--

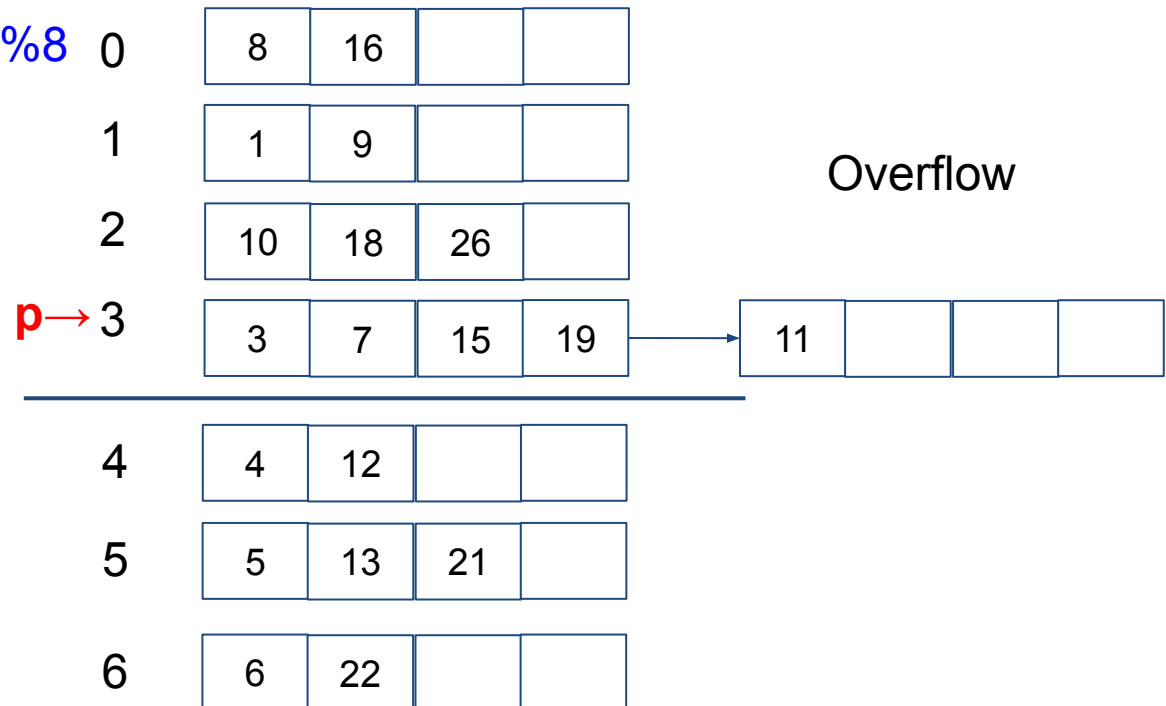
Overflow

Linear hashing

- Estratégia (a) - fim round 0

$$h_0 = \text{key} \% 4$$

$$h_1 = \text{key} \% 8$$



Linear hashing

- Estratégia (a) - início round 1

$$h_0 = \text{key} \% 4$$

$$h_1 = \text{key} \% 8$$

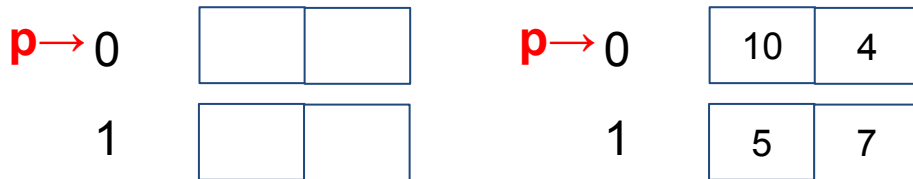
$p \rightarrow$

0	8	16			
1	1	9			
2	10	18	26	34	42
3	3	19	11		
4	4	12			
5	5	13	21		
6	6	22			
7	7	15			

Overflow

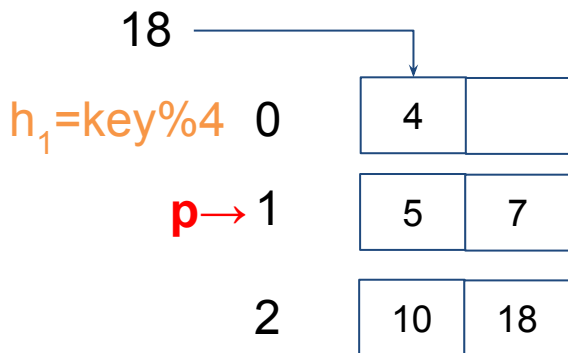
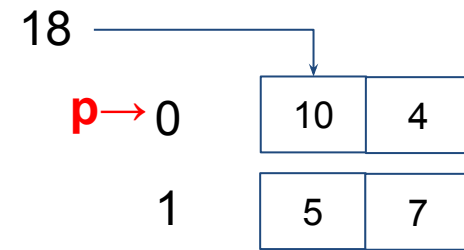
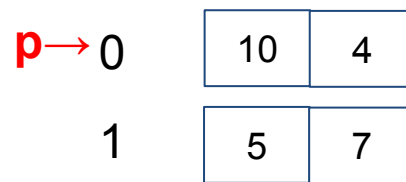
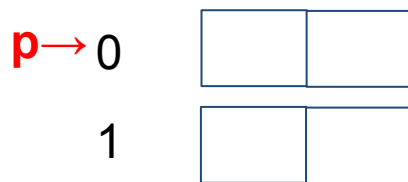
Linear hashing

- Estratégia (a) - Exemplo (**round 0**)
 - ~~10, 5, 4, 7~~, 18, 14, 22, 9, 13, 8, 11 ($h_0(k)=k\%2$)



Linear hashing

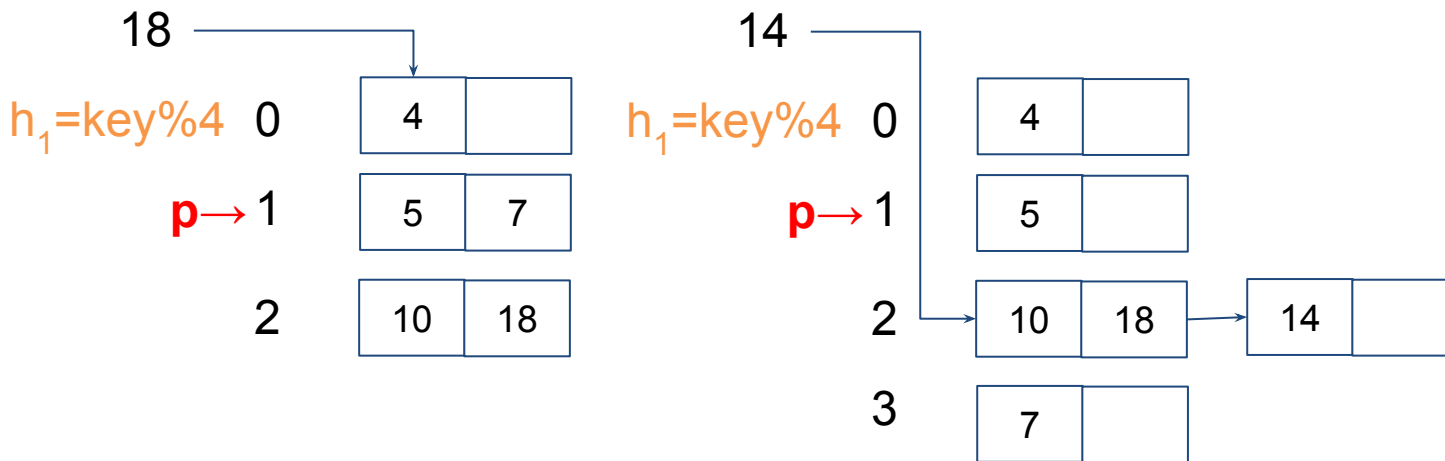
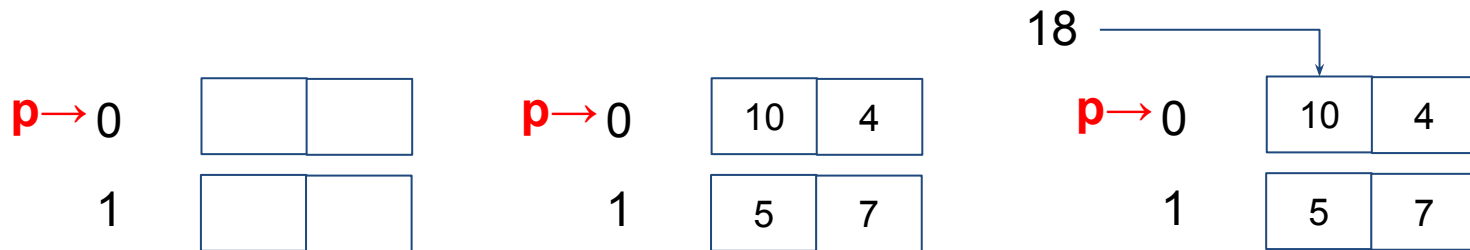
- Estratégia (a) - Exemplo (**round 0**)
 - ~~10, 5, 4, 7~~, **18**, 14, 22, 9, 13, 8, 11 ($h_0(k)=k\%2$)



Linear hashing

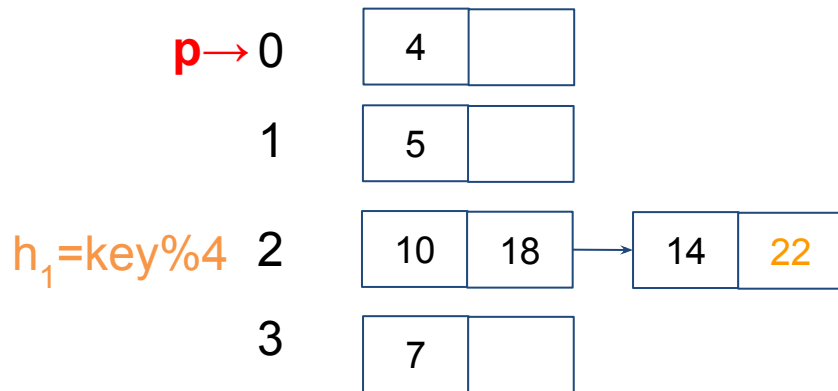
- Estratégia (a) - Exemplo (**round 0**)

- ~~10, 5, 4, 7, 18~~, **14**, 22, 9, 13, 8, 11 ($h_0(k)=k\%2$)



Linear hashing

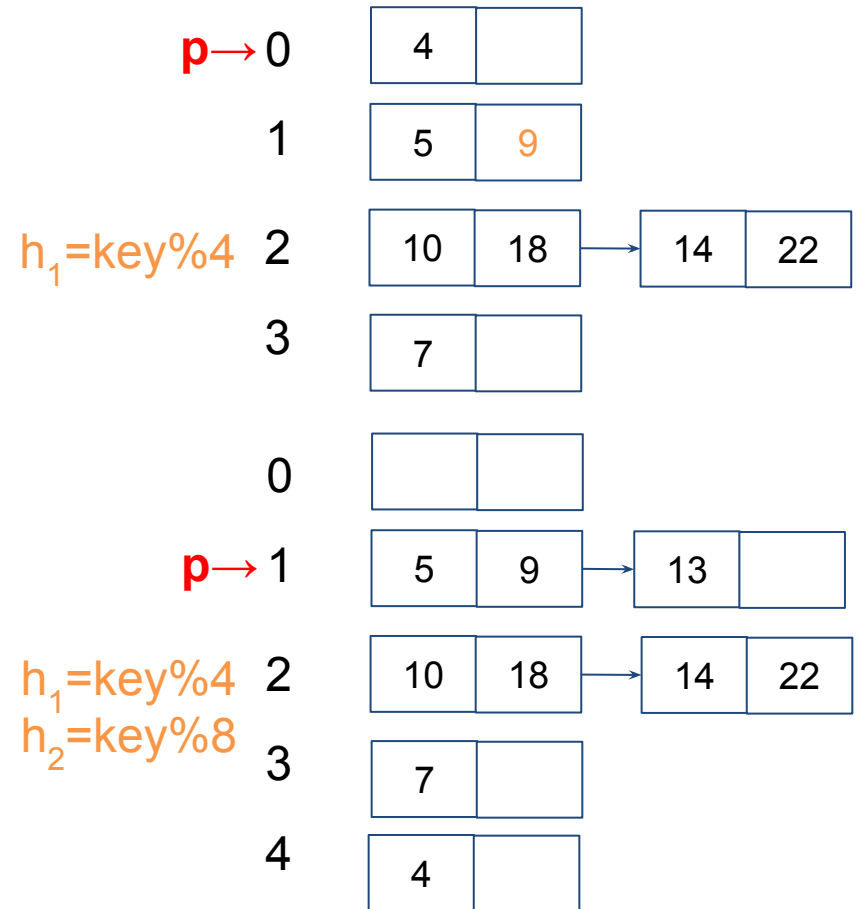
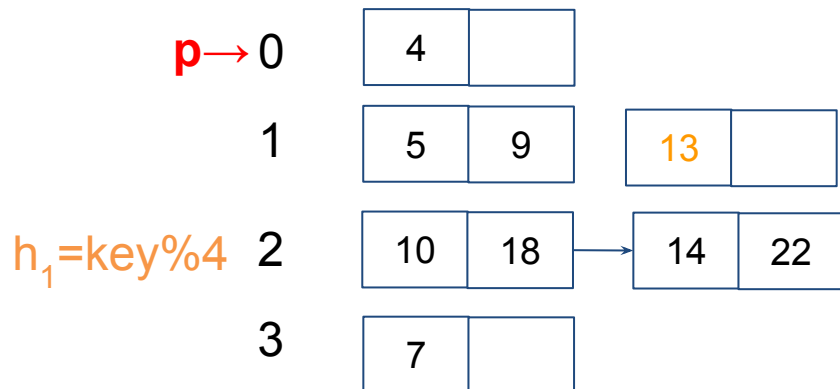
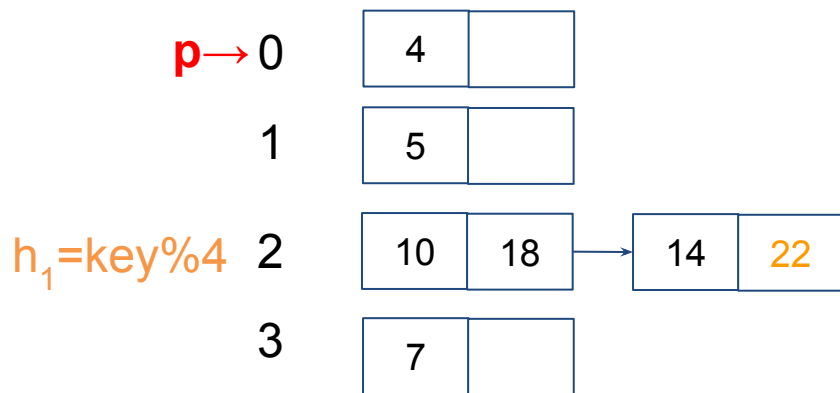
- Estratégia (a) - Exemplo (**round 2**)
 - ~~10, 5, 4, 7, 18, 14, 22~~, 9, 13, 8, 11 ($h_0(k)=k\%2$)



Linear hashing

- Estratégia (a) - Exemplo (**round 2**)

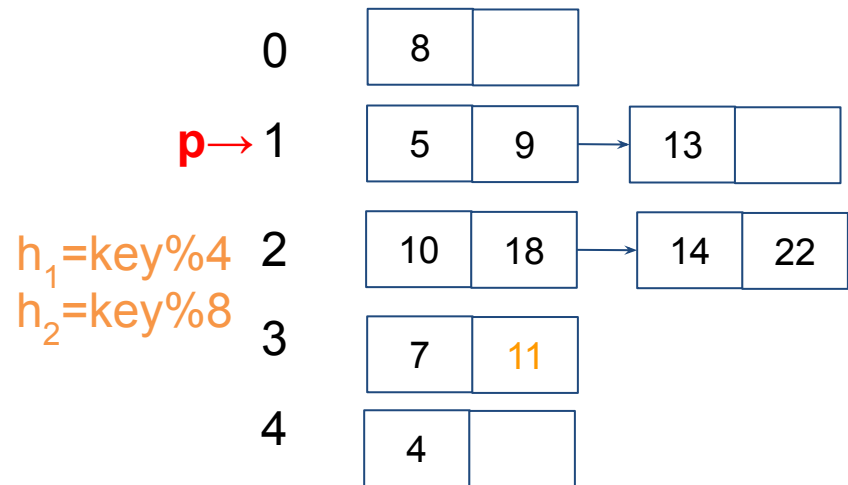
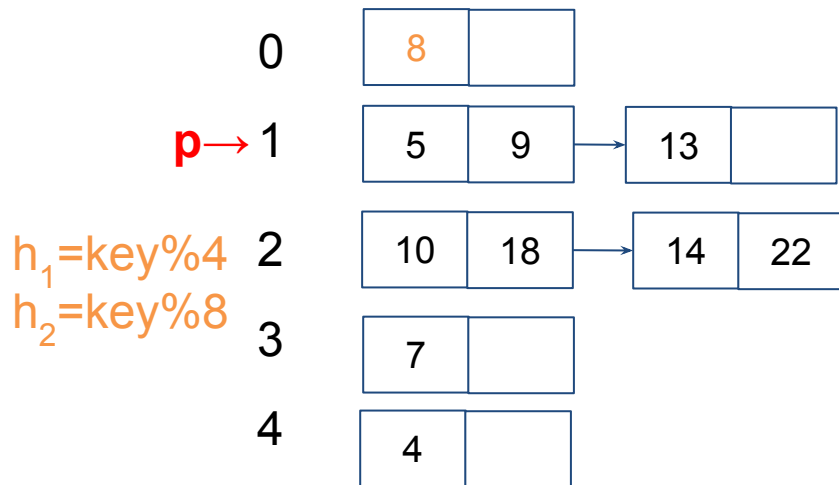
- ~~10, 5, 4, 7, 18, 14, 22~~, 9, 13, 8, 11 ($h_0(k) = k \% 2$)



Linear hashing

- Estratégia (a) - Exemplo

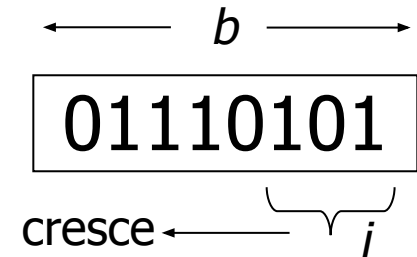
- ~~10, 5, 4, 7, 18, 14, 22, 9, 13, 8, 11~~ ($h_0(k) = k \% 2$)



Linear hashing

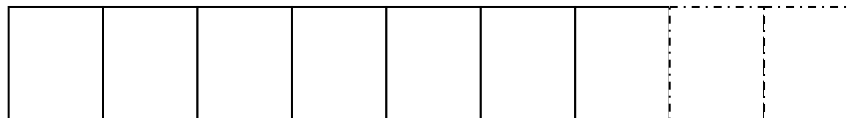
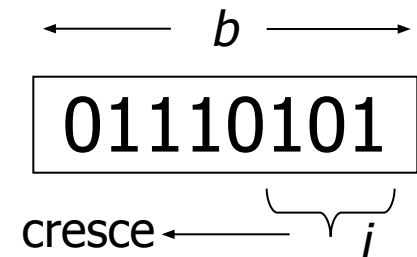
- Estratégia (b)

(b) Use i bits de menor ordem do hash



Linear hashing

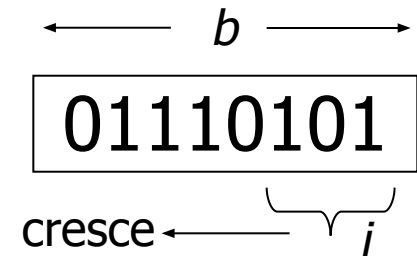
- Estratégia (b)
 - Use i bits de menor ordem do hash
 - Arquivo cresce linearmente



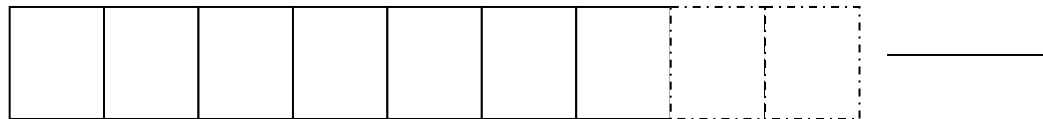
Linear hashing

- Estratégia (b)

- Use i bits de menor ordem do hash



- Arquivo cresce linearmente (+ lento que o extensível)



- Mantém-se um percentual de ocupação dos buckets (e.g., 85%)

Exemplo $n=2$ (dois buckets); 2 chaves por bucket;
 $i=1$ (1 bit utilizado); $r=3$ (número de registro na
tabela hashing)

0000	1111
1010	

0 1

← chaves terminadas em 0 e 1

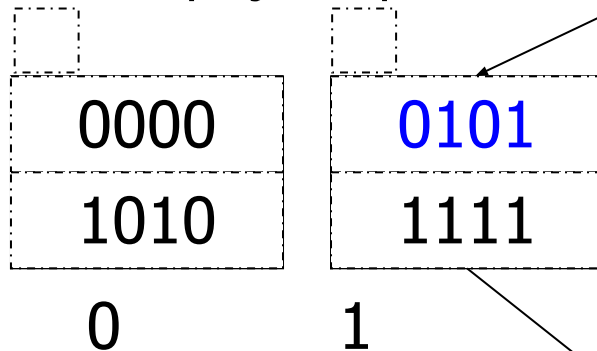
Política de ocupação:

- Para n buckets, podemos fazer r/n como a razão de ocupação. Pode-se definir que $r \leq 1.7n$ ($1.7 \times 3 = 3.4$) de ocupação de registros. Ou seja, uma ocupação dos buckets em $\cong 80\%$.

Exemplo $n=2$ (dois buckets); 2 chaves por bucket;
 $i=1$ (1 bit utilizado); $r=3$ (número de registro na
tabela hashing)

•insere 0101

Existe espaço disponível



← chaves terminadas em 0 e 1

estoura a razão 80%
(ocupação 100%). Solução $n=3$

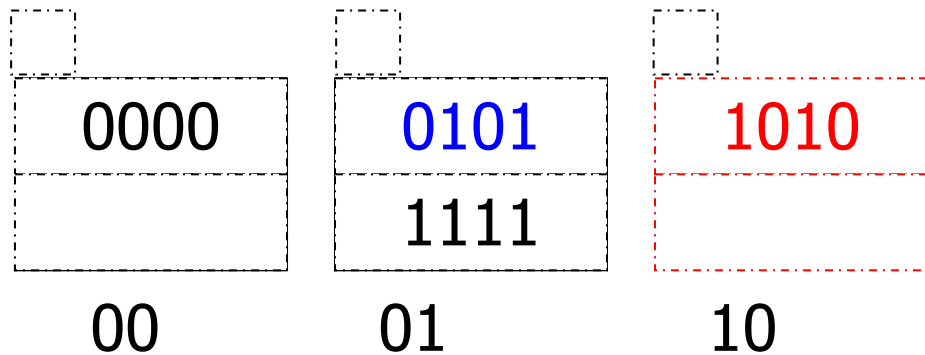
Regra

Se $h(k)[i] \leq n$

então procure no bucket $h(k)[i]$

senão procure bucket $h(k)[i] - 2^{i-1}$

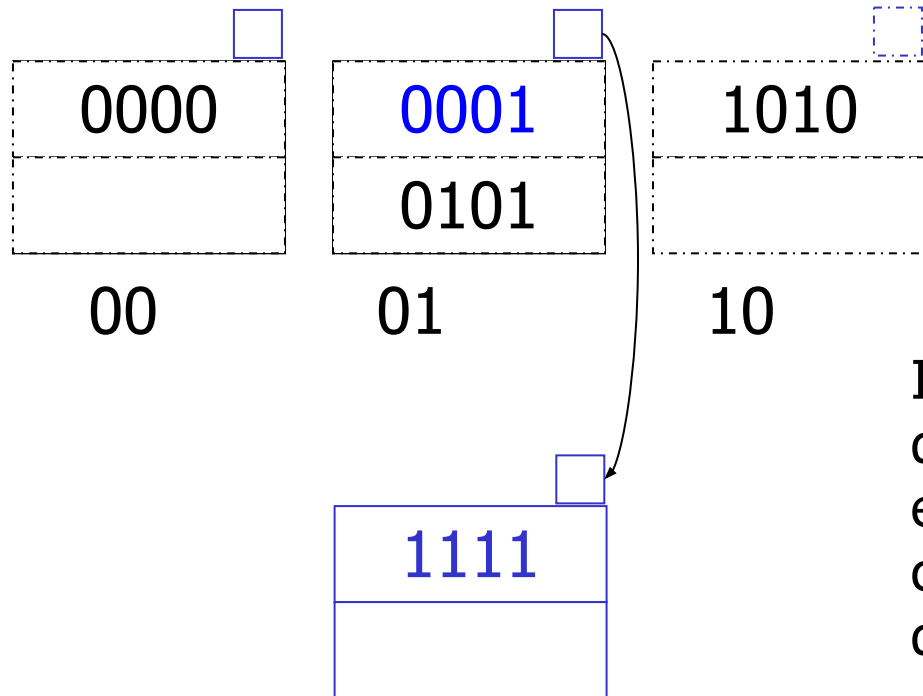
Exemplo $n=3$; 2 chaves por bucket; $i=2$; $r=4$



Cria o novo bucket e separa os valores que terminam com 0, incrementa i em 1.

E se inserir o valor 0001? Vai para o bucket 01 que está cheio (overflow)

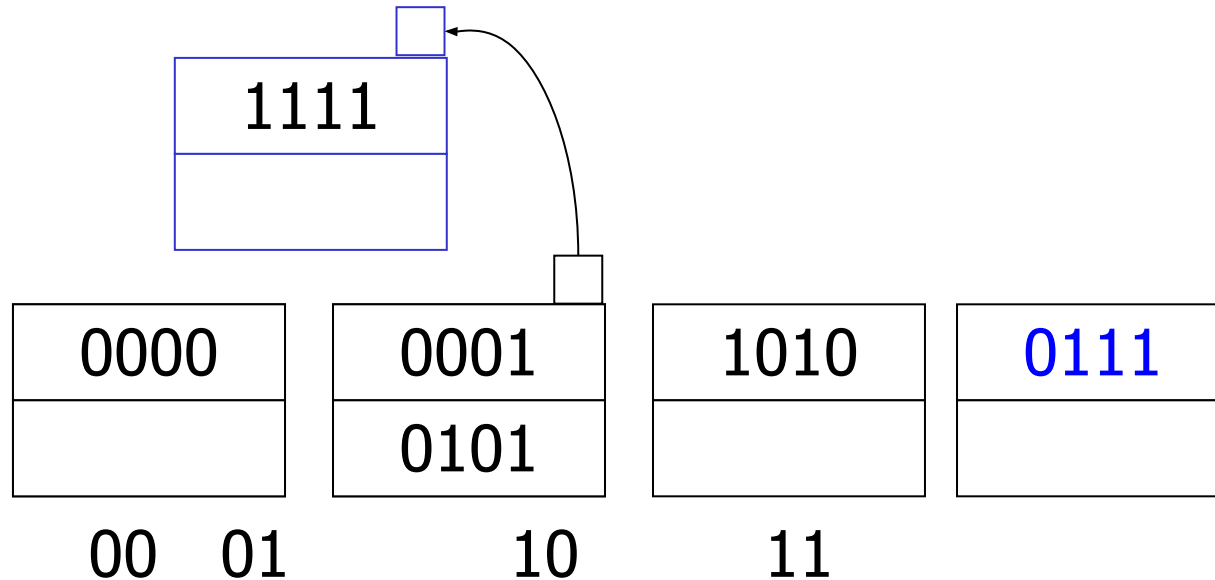
Exemplo $n=3$; 2 chaves por bloco; $i=2$; **$r=5$**



Inserir 0111. Dois últimos bits 11 deveria ir para o bucket 11 (não existe), vai para o 01 que está cheio. Pode ser colocado no bucket de overflow ...

Continuação: Como crescer além?

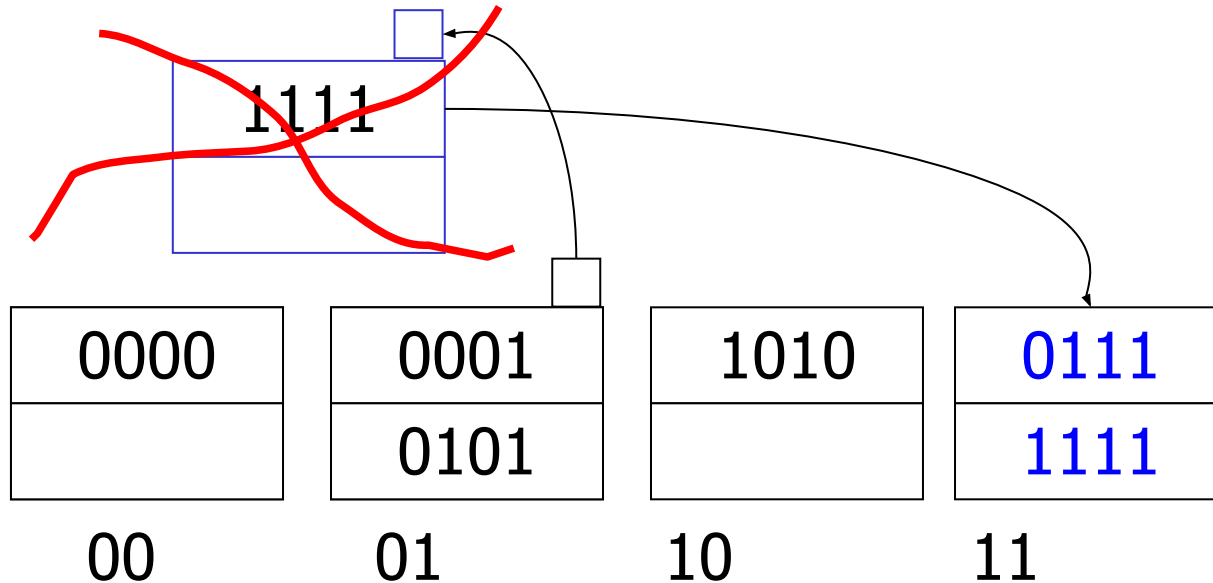
$n=4$; 2 chaves por bloco; $i=2$; **$r=6$**



Não pode ser colocado no bucket de overflow porque estoura a razão de 80% ($r > 1.7n$). Cria-se um novo bucket para finais 11 ($i=2$)

Continuação: Como crescer além?

$n=4$; 2 chaves por bloco; $i=2$; **$r=6$**



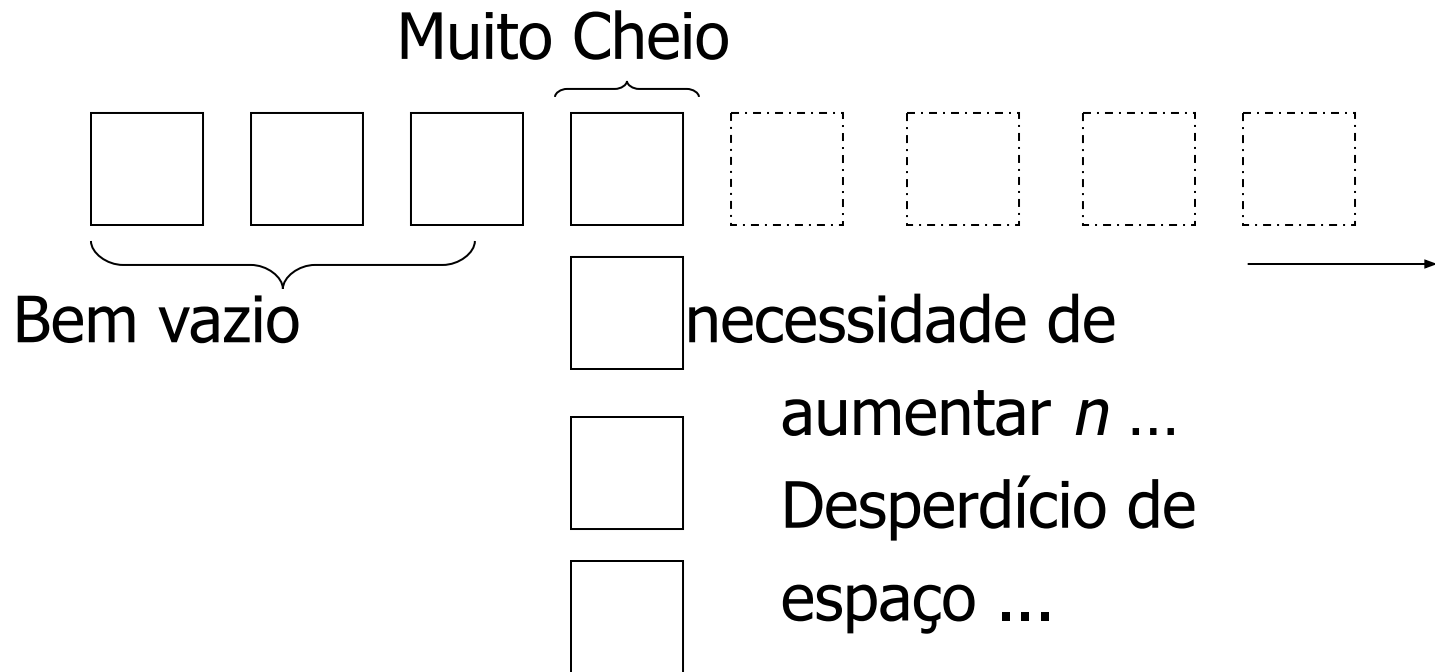
Como temos um bucket 11, a chave 1111 vai para este novo bucket

Resumo

Hashing linear

- ⊕ Pode trabalhar com arquivos que crescem
 - com menos perda de espaço
 - sem reorganização total
- ⊕ Sem indireção (hashing extensível)
- Pode ainda ter cadeias de overflow

Exemplo: Caso Ruim



Índice_{B+} vs Hashing

- Hashing bom para procurar uma dada chave

e.g., **SELECT ...**
 FROM R
 WHERE R.A = 5

Indexing_{B+} vs Hashing

- INDEXING (incluindo B Trees) bom para Procura por faixa de valores:

e.g., **SELECT** ...

FROM R

WHERE R.A > 5

Definição (básica) de índice em SQL

- `create index name on` rel (attr)
- `create unique name on` rel (attr)

pode-se adicionar a opção:

`include (att1, ..., attn)`

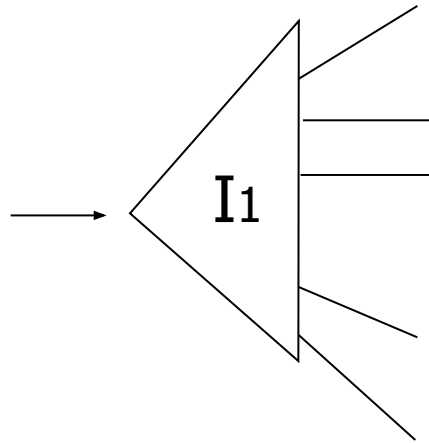
- `drop index name`

Índice multichave

Motivação: Encontre registros os quais
DEPT = "Toy" AND SAL > 50k

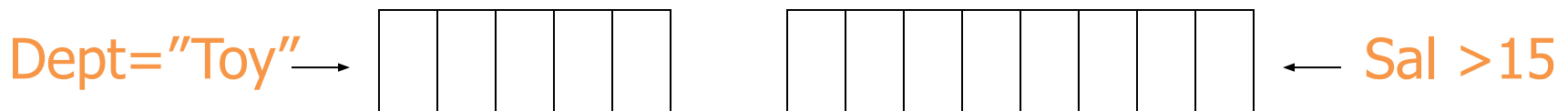
Estratégia I:

- Usar um índice, digamos Dept.
- Retorne todos os registros Dept = "Toy" e verifique os salários



Estratégia II:

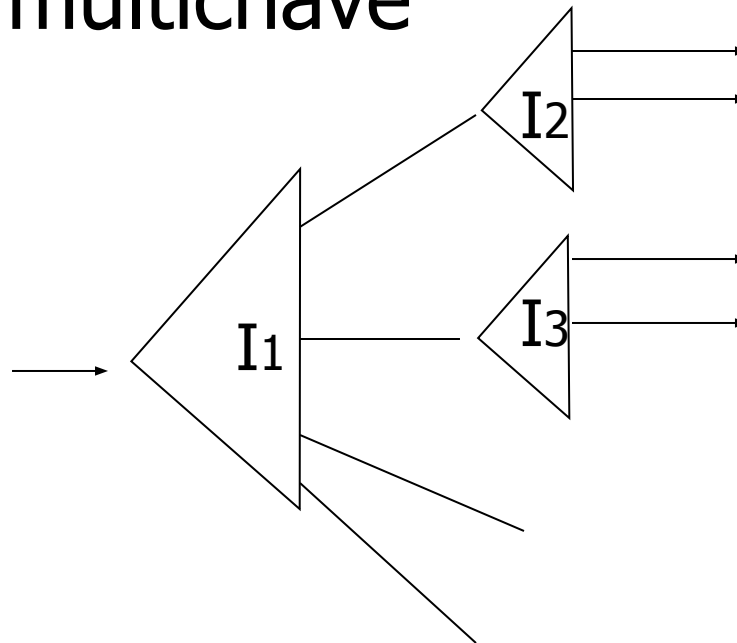
- Usar 2 Índices; Manipular os ponteiros



Estratégia III:

- Índice multichave

Ideia 1:



Exemplo

Art	
Sales	
Toy	

Dept
Index

10k	
15k	
17k	
21k	

12k	
15k	
15k	
19k	

Salary
Index

Registro
Exemplo

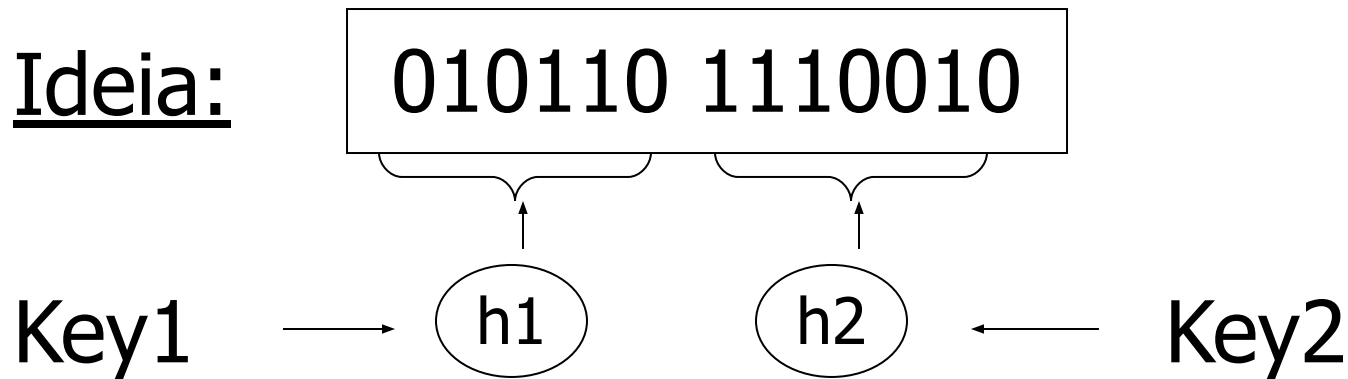
Name=Joe
DEPT=Sales
SAL=15k

Para quais consultas este índice é bom?

- ✓ RECs Dept = "Sales" \wedge SAL=20k
- ✓ RECs Dept = "Sales" \wedge SAL \geq 20k
- ✓ RECs Dept = "Sales"
- ✓ RECs SAL = 20k

Função de hashing particionada

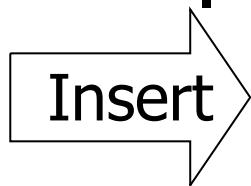
Ideia:



EX:

h1(toy)	=0	000	
h1(sales)	=1	001	
h1(art)	=1	010	
.		011	
h2(10k)	=01	100	
h2(20k)	=11	101	
h2(30k)	=01	110	
h2(40k)	=00	111	

⋮

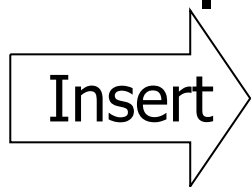


<Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

EX:

h1(toy)	=0	000	
h1(sales)	=1	001	<Fred>
h1(art)	=1	010	
.		011	
h2(10k)	=01	100	
h2(20k)	=11	101	<Joe> <Sally>
h2(30k)	=01	110	
h2(40k)	=00	111	

⋮



<Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe> <Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom> <Bill>
h2(40k)	=00	111	<Andy>
:			
:			

- $\sigma_{\text{Dept} = \text{"Sales"} \wedge \text{Sal}=40\text{k}} (\text{Employee})$

h1(toy) = 0

h1(sales) = 1

h1(art) = 1

.

h2(10k) = 01

h2(20k) = 11

h2(30k) = 01

h2(40k) = 00

:

.

000

<Fred>

001

<Joe> <Jan>

010

<Mary>

011

100

<Sally>

101

110

<Tom> <Bill>

111

<Andy>

- $\sigma_{\text{Dept} = \text{"Sales"} \wedge \text{Sal}=40\text{k}} (\text{Employee})$

h1(toy) =0

h1(sales) =1

h1(art) =1

.

h2(10k) =01

h2(20k) =11

h2(30k) =01

h2(40k) =00

:

.

- $\sigma_{\text{Sal}=30\text{k}}$ (Employee)

000

<Fred>

001

<Joe> <Jan>

010

<Mary>

011

<Sally>

100

101

110

<Tom> <Bill>

111

<Andy>

h1(toy) = 0

h1(sales) = 1

h1(art) = 1

.

h2(10k) = 01

h2(20k) = 11

h2(30k) = 01

h2(40k) = 00

:

- $\sigma_{\text{Sal}=30k}$ (Employee)

000	<Fred>
001	<Joe> <Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom> <Bill>
111	<Andy>

procure aqui

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe> <Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom> <Bill>
h2(40k)	=00	111	<Andy>
:			
.			

- $\sigma_{\text{Dept} = \text{"Sales"}} (\text{Employee})$

h1(toy) = 0

h1(sales) = 1

h1(art) = 1

.

h2(10k) = 01

h2(20k) = 11

h2(30k) = 01

h2(40k) = 00

:

.

- $\sigma_{\text{Dept} = \text{"Sales"}}$ (Employee)

000

<Fred>

001

<Joe> <Jan>

010

<Mary>

011

100

<Sally>

101

110

<Tom> <Bill>

111

<Andy>

Procure aqui

BitMap Indexes

Ideia

- Um índice bitmap para um atributo F é uma coleção de vetores de bits de comprimento n (número de valores distintos de F)
- Cada bit representa o valor que deve aparecer em F
- O vetor para o valor v tem 1 na posição i para o i -ésimo registro com v no atributo F , caso contrário será 0

Exemplo

- Suponhamos um arquivo com dois atributos **age** e **name** com os seguintes registros:
- $\langle 30, \text{joão} \rangle, \langle 30, \text{josé} \rangle, \langle 40, \text{maria} \rangle, \langle 50, \text{joão} \rangle, \langle 40, \text{josé} \rangle, \langle 30, \text{maria} \rangle$
- O bitmap para **age** terá três vetores de seis bits
- Primeiro **110001** para **30**
- Segundo **001010** para **40**
- Terceiro **000100** para **50**
- E para o atributo **name**?

Exemplo

- $\langle 30, \text{joão} \rangle, \langle 30, \text{josé} \rangle, \langle 40, \text{maria} \rangle, \langle 50, \text{joão} \rangle, \langle 40, \text{josé} \rangle, \langle 30, \text{maria} \rangle$

Valor	Vetor
joão	100100
josé	010010
maria	001001

Comentários

- Parece que os índices bitmap exigem muito espaço de armazenamento especialmente quando existem muitos valores distintos para o atributo utilizado
- Se o bitmap é feito sobre uma chave teremos n^2 bits, *i.e.*, n bits para localização de valor v vezes n bits para cada valor v' distinto
- Podem ser utilizadas técnicas de compressão para chegar a um valor próximo de n (Seção 14.7.2)

Exemplo

- Suponhamos uma tabela com os campos **age** e **salary** com 12 tuplas:
- <25,60> <45,60> <50,75> <50,100>
- <50,120> <70,110> <85,140> <30,260>
- <25,400> <45,350> <50,275> <60,260>
- Como seria o índice bitmap para **age**? (7 valores diferentes)

Exemplo

- $\langle 25, 60 \rangle$ $\langle 45, 60 \rangle$ $\langle 50, 75 \rangle$ $\langle 50, 100 \rangle$
- $\langle 50, 120 \rangle$ $\langle 70, 110 \rangle$ $\langle 85, 140 \rangle$ $\langle 30, 260 \rangle$
- $\langle 25, 400 \rangle$ $\langle 45, 350 \rangle$ $\langle 50, 275 \rangle$ $\langle 60, 260 \rangle$

25: 100000001000 30: 000000010000

50: 001110000010 60: 000000000001

85: 000000100000 45: 010000000100

70: 000001000000

Exemplo

- $\langle 25, 60 \rangle$ $\langle 45, 60 \rangle$ $\langle 50, 75 \rangle$ $\langle 50, 100 \rangle$
- $\langle 50, 120 \rangle$ $\langle 70, 110 \rangle$ $\langle 85, 140 \rangle$ $\langle 30, 260 \rangle$
- $\langle 25, 400 \rangle$ $\langle 45, 350 \rangle$ $\langle 50, 275 \rangle$ $\langle 60, 260 \rangle$

Salary (10 valores distintos)

60:	1100000000000	75:	0010000000000
100:	0001000000000	110:	0000010000000
120:	0000100000000	140:	0000001000000
260:	000000010001	275:	0000000000010
350:	0000000000100	400:	0000000001000

Exemplo

- Consultas:

$\sigma_{\text{age}=50 \wedge \text{salary}=100}(\text{Employee})$

$\sigma_{(\text{age} \geq 45 \wedge \text{age} \leq 55) \wedge (\text{salary} \geq 100 \wedge \text{salary} \leq 200)}(\text{Employee})$

Considerações

- Índice bitmap é útil quando a chave utilizada possui muitas repetições
- O armazenamento é compactado pois utiliza apenas bits.
- Geralmente, o SGBD utiliza índices bitmaps em tempo de execução para otimizar consultas.