Hashing Extensível

Organização e Recuperação de Dados

Profa. Valéria

UEM – CTC – DIN

Slides preparados com base no Cap. 11 do livro FOLK, M.J. & ZOELLICK, B. *File Structures*. 2nd Edition, Addison-Wesley Publishing Company, 1992

Hashing extensível

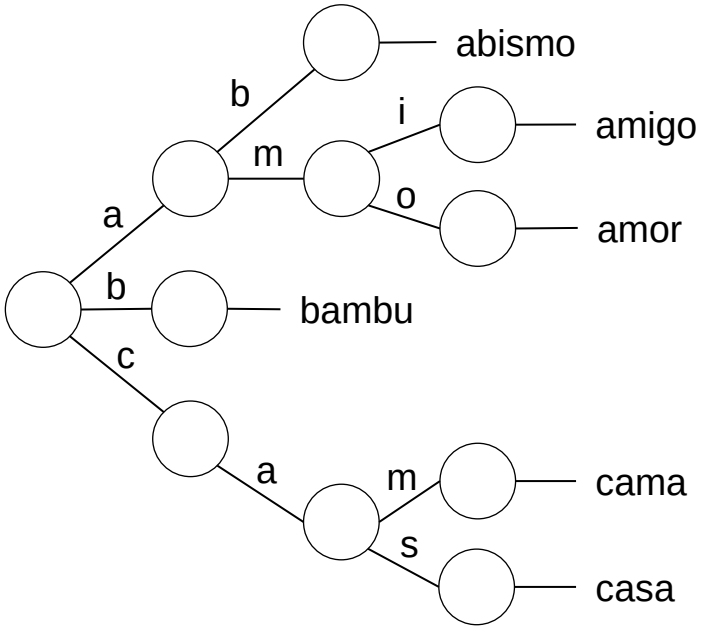
- *Hashing* extensível é uma técnica que permite que o **espaço de endereços** seja **dinâmico**
  - O espaço de endereços aumenta e diminui conforme as inserções/remoções acontecem
  - **Não é mais necessário pré-definir a quantidade de endereços disponíveis, nem gerenciar colisões**
  - O *hashing* manterá o seu desempenho mesmo que o arquivo cresça muitas vezes em tamanho

Hashing extensível

- Combinação de duas estruturas:
  - **Arquivo de *buckets***
  - ***Diretório*** com os endereços dos *buckets*
- O diretório é inspirado em uma estrutura de dados chamada ***trie***
  - Uma ***trie*** é uma árvore de busca chamada de árvore digital, na qual o fator de divisão de cada nó é igual ao número de símbolos do alfabeto
  - As chaves são indexadas símbolo a símbolo (ou dígito a dígito)
  - Os *n* primeiros símbolos de uma chave definem o **prefixo de tamanho *n*** da chave

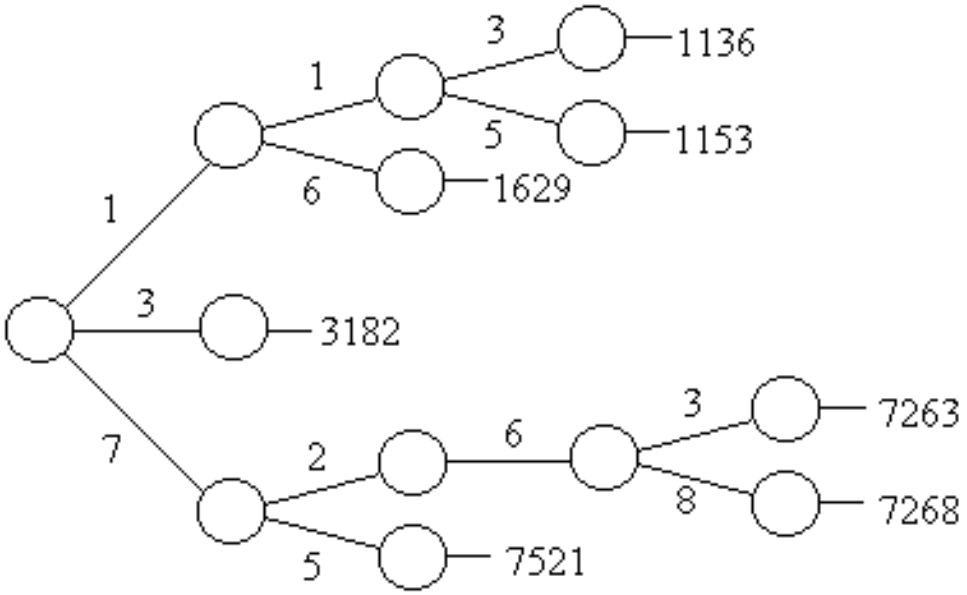
Trie

- **Exemplo 1**: Chaves alfabéticas (*trie de 26 prefixos*)
  - Supondo um alfabeto de a – z, o fator de divisão é 26
  - Um prefixo da chave é utilizado como endereço
  - Cada caminho da raiz até as folhas compõe o menor prefixo necessário para identificar uma chave de forma única
  - Conforme novas chaves vão sendo inseridas, o espaço de endereços (caminhos possíveis) vai aumentando



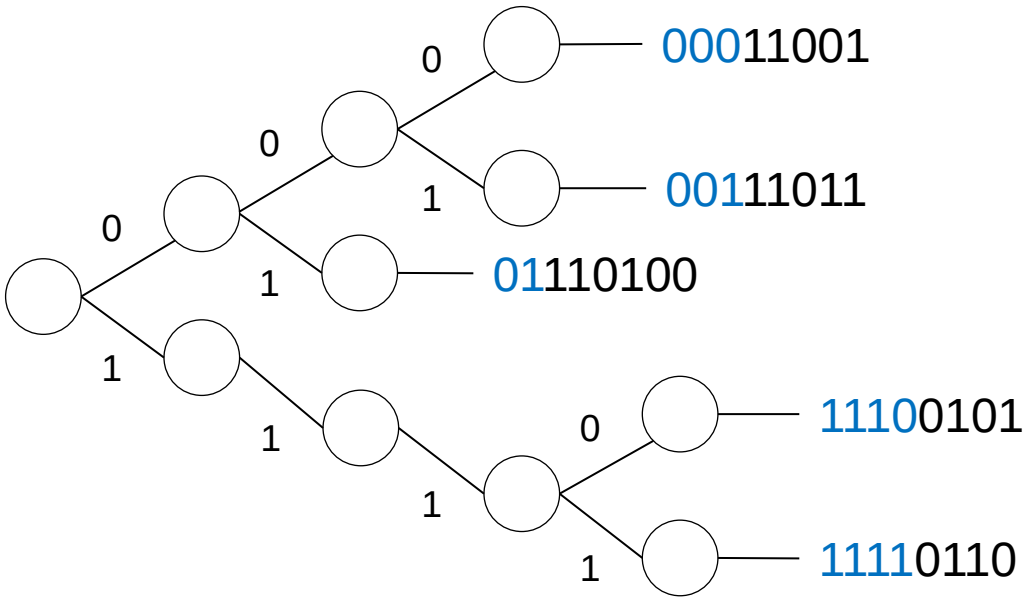
Trie

- **Exemplo 2**: Chaves numéricas (*trie de 10 prefixos*)
  - Alfabeto de 0 a 9, então o fator de divisão é 10, uma vez que existem 10 símbolos possíveis



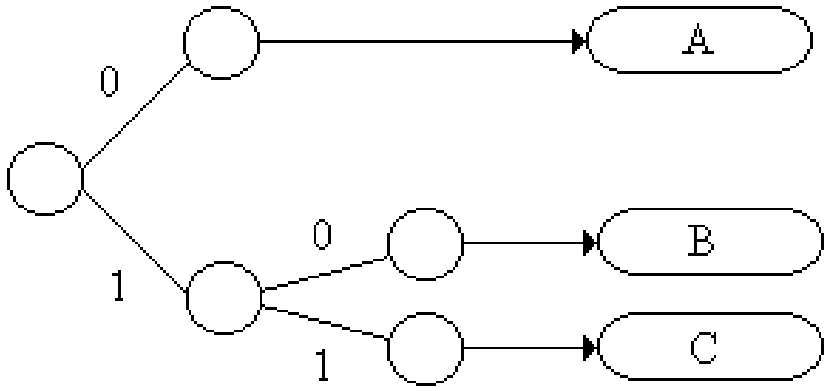
Trie

- **Exemplo 3**: Chaves binárias (*trie de* 2 *prefixos – trie binária*)
  - Alfabeto 0 e 1, então o fator de divisão é 2



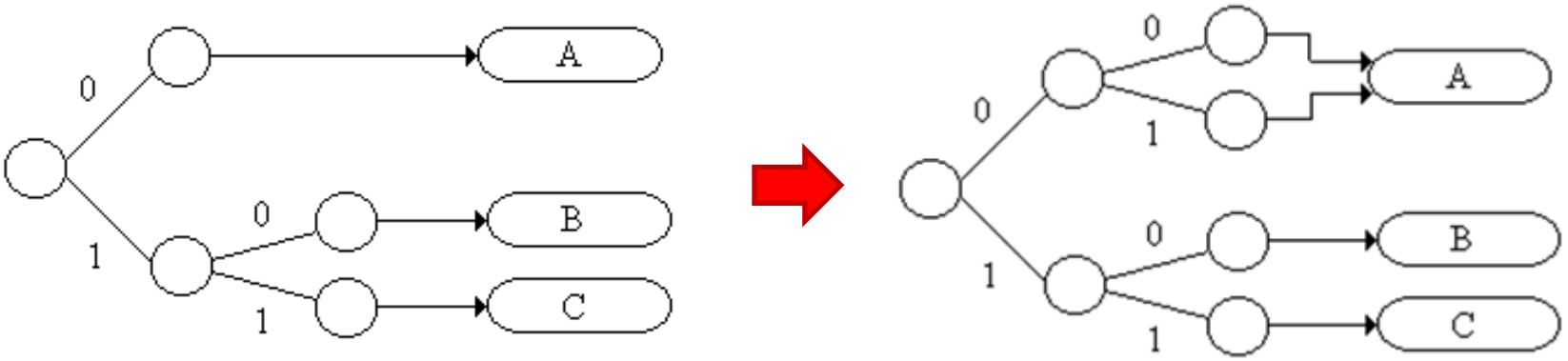
Transformando a *trie* em diretório

- *O hashing* extensível utiliza um **diretório** que é inspirado em uma ***trie binária***
- A *trie* endereça ***buckets***
  - O endereço gerado pela função *hash* é considerado bit-a-bit
- A figura ilustra uma *trie* endereçando os *buckets* A, B e C
  - O *bucket* A contém chaves cujos endereços começam com o bit **0**
  - As chaves do *bucket* B produzem endereços que começam com os bits **10**
  - As chaves no *bucket* C produzem endereços que começam com os bits **11**



Transformando a *trie* em diretório

- O **diretório** do *hashing* extensível é a representação de uma ***trie* binária completa**
  - Uma *trie* completa possui todas as folhas no mesmo nível, i.e., **todos os caminhos na *trie* têm o mesmo comprimento**



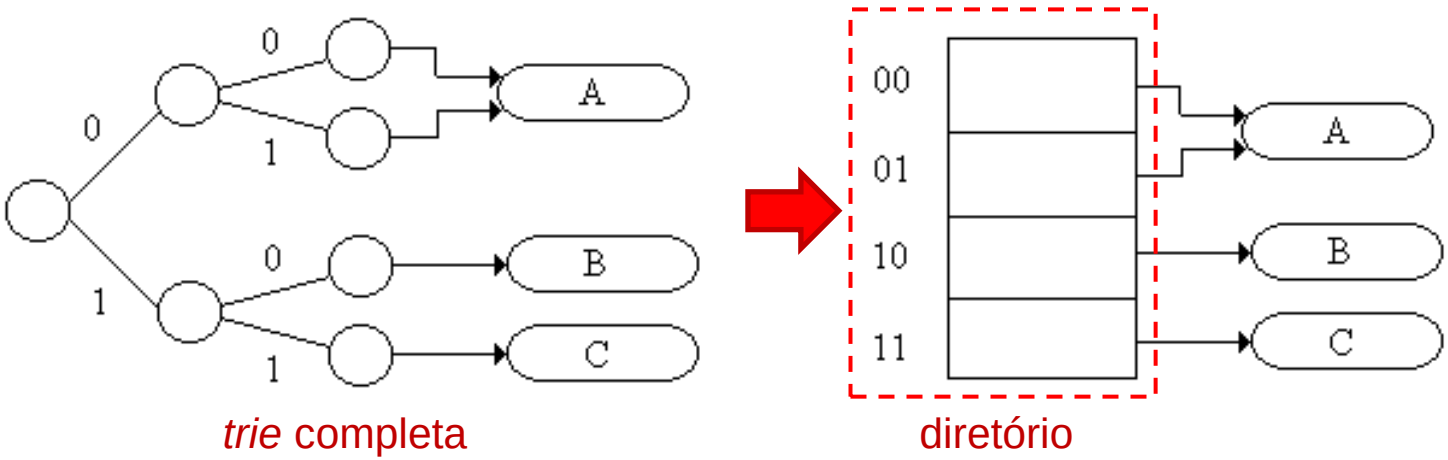
trie original

trie completa

Note que embora o bit **0** seja suficiente para endereçar o *bucket* A, a *trie* completa utiliza um segundo bit nesse caminho. Por isso, ambos endereços iniciados por **0** (**00** e **01**) apontam para o *bucket* A.

Transformando a *trie* em diretório

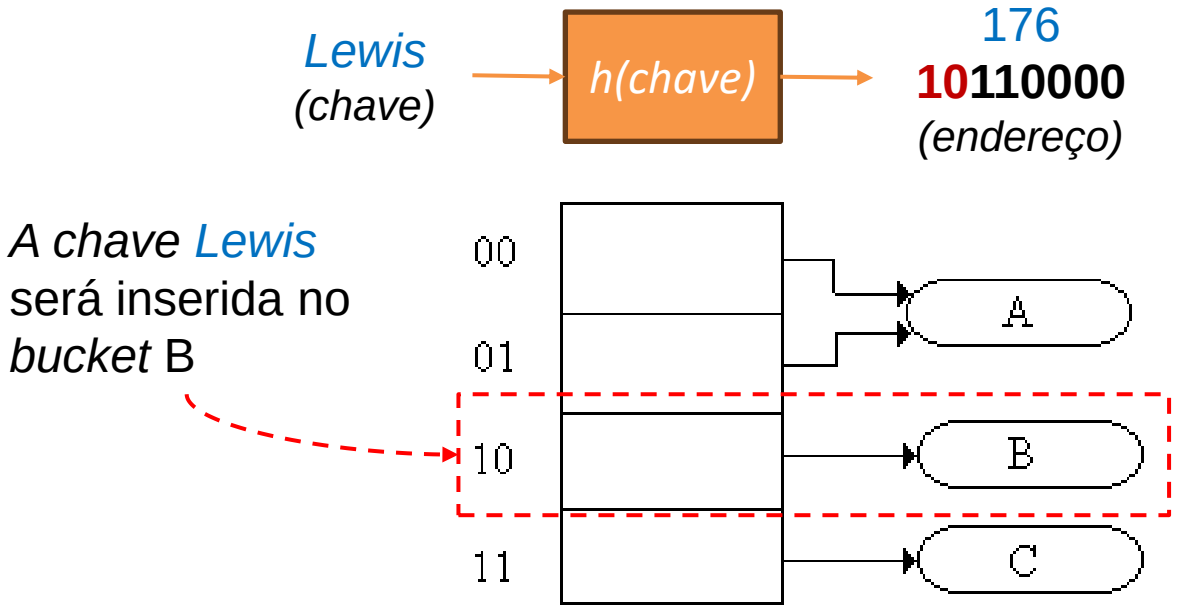
- Dado que em uma ***trie* completa** todos os caminhos têm o mesmo comprimento, podemos representa-la por meio de uma lista → **diretório**
  - Cada entrada do diretório armazena uma referência (RRN) para um *bucket* associado



trie completa diretório

Acessando o diretório

- Dada uma chave e uma função *hash*, o endereço gerado pela função é usado para acessar uma entrada do **diretório**
  - Por ex., dada uma chave cujo endereço comece com os bits **10**, a entrada **10** do diretório fornece a referência do *bucket* onde essa chave deve ser armazenada



Inserção e tratamento de *overflow*

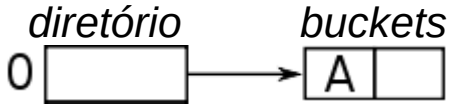
- Veremos o funcionamento do *hashing* extensível por meio de um exemplo:
  - Adaptado de Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R. Extendible Hashing - A Fast Access Method for Dynamic Files, **ACM Transactions on Database Systems**, v. 4, n. 3, p. 315–344, 1979.

- Suponha uma função *h*(*k*) que retorna um **número inteiro** como endereço

- Para simplificar o exemplo, vamos assumir que o tamanho do *bucket* é 1 (pode armazenar apenas uma chave), mas **na prática os buckets serão maiores (blocos de registros)**

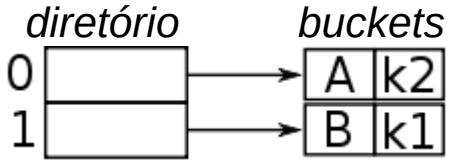
Inserção e tratamento de *overflow*

- Inicialmente, o *hashing* está vazio



- Endereços retornados por *h(k)* para as chaves k_1, e k_2:
  - $h(k_1)$ = 172 = 10101100
    $h(k_2)$ = 090 = 01011010

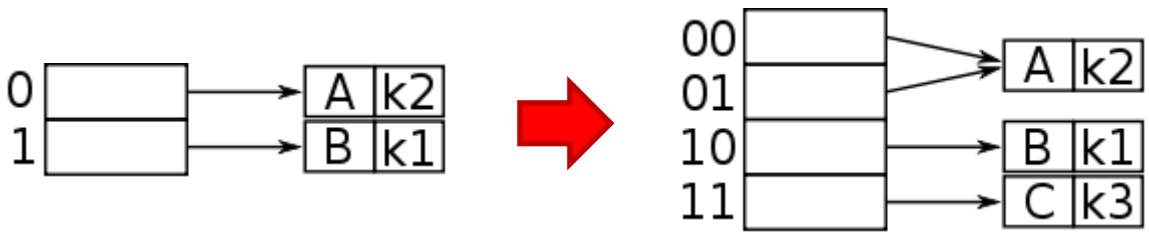
- k_1 e k_2 podem ser distinguidas pelo bit mais significativo e seriam inseridas nos *buckets* A e B:



Inserção e tratamento de *overflow*

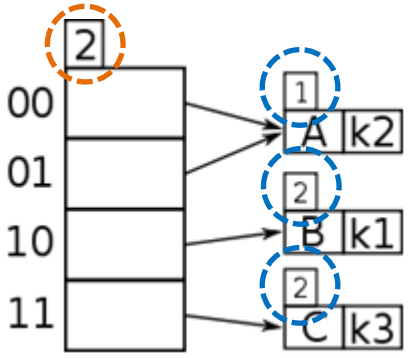
- Quando for feito o *hashing* de k_3, um bit não será mais suficiente para distinguir entre as três chaves
  - Ocorrerá *overflow* no *bucket* B (as chaves K_1 e k_3 colidem, pois o *bucket* está cheio)
  - Se usarmos os 2 bits mais significativos de cada chave, resolvemos o problema da colisão
  - Para isso, dobramos o tamanho do diretório e dividimos o *bucket* B

$h(k_1)$ = 10101100
$h(k_2)$ = 01011010
$h(k_3)$ = 11011000



Inserção e tratamento de *overflow*

- Toda vez que o diretório aumenta, ele dobra de tamanho devido ao fato de estarmos usando números binários no endereçamento (2^n)
- Chamamos de **profundidade** a **quantidade de bits que está sendo utilizada nos endereços**
- Inicialmente, o diretório tem profundidade 0 e tamanho 1 ($2^0 = 1$)
  - Se incluímos um bit nos endereços, o diretório passa a ter profundidade 1 e tamanho 2 ($2^1 = 2$)
  - E assim sucessivamente: $2^2 = 4$; $2^3 = 8$; $2^4 = 16$...
- Um *overflow* pode provocar o aumento do diretório, mas nem sempre!
- Para decidir se aumentamos o diretório, comparamos:
  - A **quantidade de bits dos endereços do diretório**, chamada de **profundidade global**
  - A **quantidade de bits em comum dos endereços do *bucket***, chamada de **profundidade local**

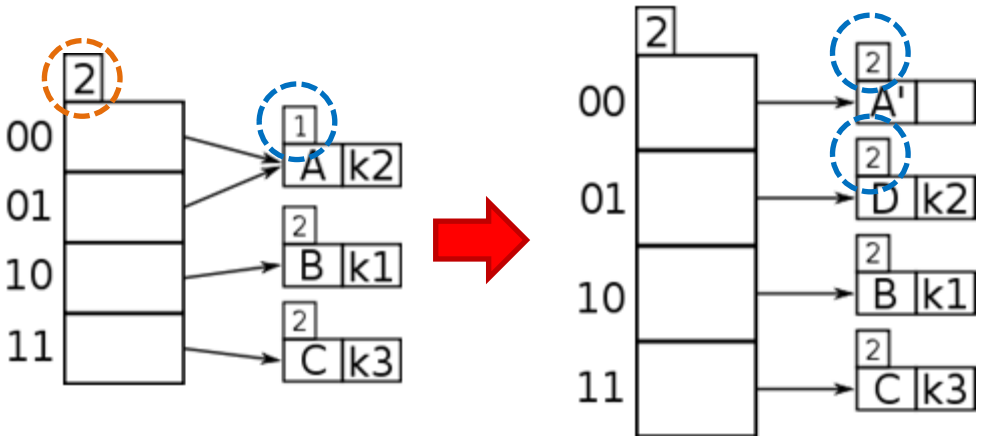


Inserção e tratamento de *overflow*

- O *hashing* da chave k_4 diz que ela deve ser inserida no *bucket* A
- O *bucket* A está cheio, então ocorre *overflow*
- Como a profundidade local < profundidade global, o *bucket* A é dividido e o diretório atual acomoda o novo *bucket*
  - É criado um novo *bucket* (D) e a chave do *bucket* A é remapeada utilizando agora 2 bits do seu endereço base
    - As profundidades dos *buckets* A e D são incrementadas para **2**

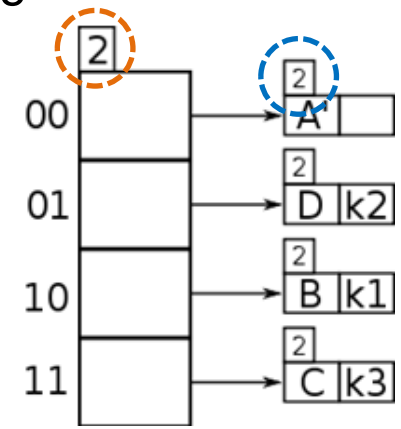
$h(k_1)$ = 10101100
$h(k_2)$ = 01011010
$h(k_3)$ = 11011000
$h(k_4)$ = 01111000

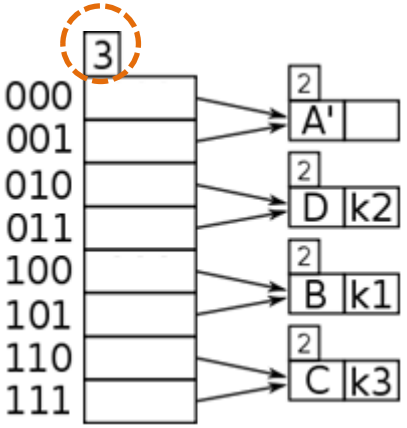
O problema é que a chave que estava em A é remapeada para D e com a inserção da chave k_4 **o *overflow* continua no *bucket* D** (k_2 e k_4 colidem)



Inserção e tratamento de *overflow*

- O *overflow* continua no *bucket* D, então haverá nova divisão
- O diretório aumentará?
  - **Agora sim**, pois a profundidade *bucket* D = profundidade do diretório
  - Cada endereço de 2 bits será mapeado em dois endereços de 3 bits
  - Cada *bucket* passa a ter duas referências no diretório
  - A **profundidade global** passa a ser igual a **3**



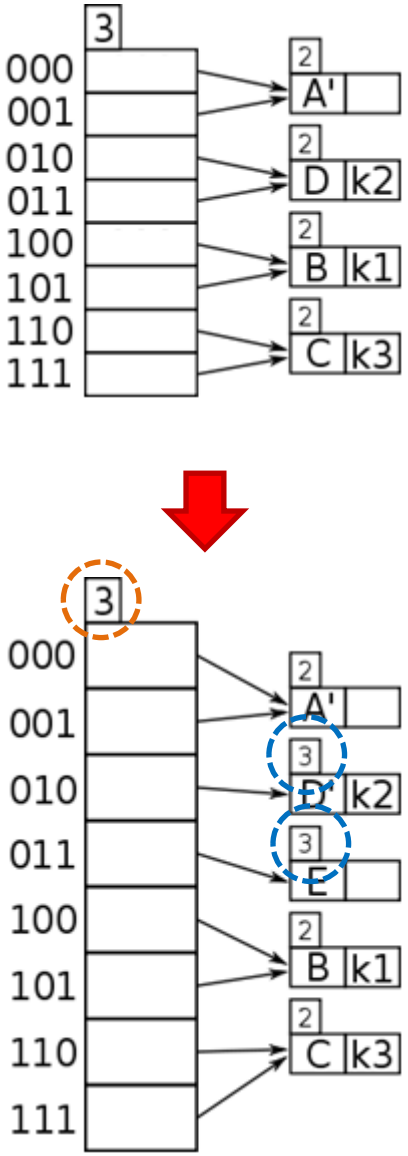


$h(k_1)$ = 10101100
$h(k_2)$ = 01011010
$h(k_3)$ = 11011000
$h(k_4)$ = 01111000

Inserção e tratamento de *overflow*

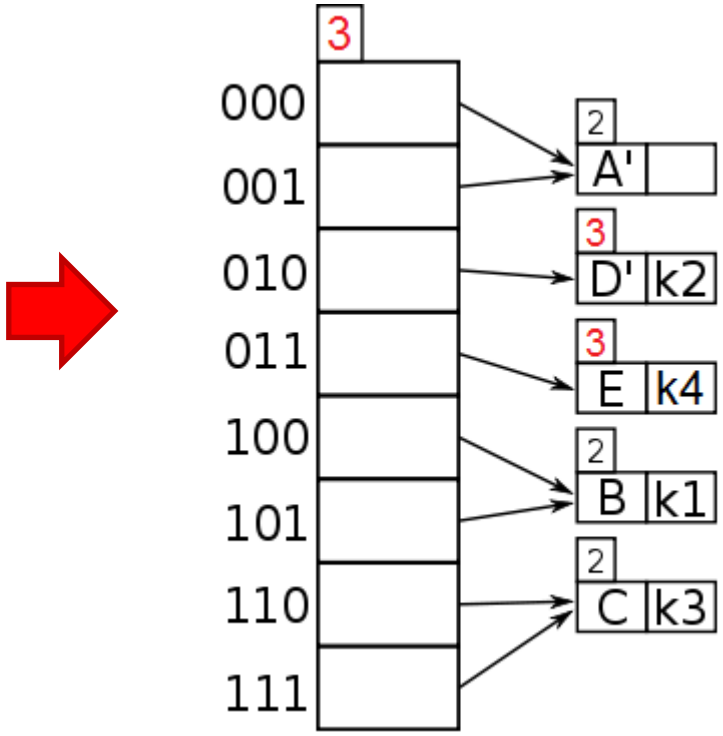
- O diretório aumentou e a profundidade global agora é 3
  - O *bucket* D (profundidade local = 2) é dividido, gerando o *bucket* E, ambos com profundidade local = 3
  - A chave do *bucket* D é remapeada usando 3 bits do seu endereço e fica em D mesmo ($h(k_2)$ = **010**11010)
  - A chave k_4 também é remapeada usando 3 bits do seu endereço ($h(k_4)$ = **011**11000) e vai para o *bucket* E

$h(k_1)$ = 10101100
$h(k_2)$ = 01011010
$h(k_3)$ = 11011000
$h(k_4)$ = 01111000



Inserção e tratamento de *overflow*

- Situação final:
  - Endereços
    - $h(k_1)$ = 10101100
    - $h(k_2)$ = 01011010
    - $h(k_3)$ = 11011000
    - $h(k_4)$ = 01111000
  - Profundidade global = 3

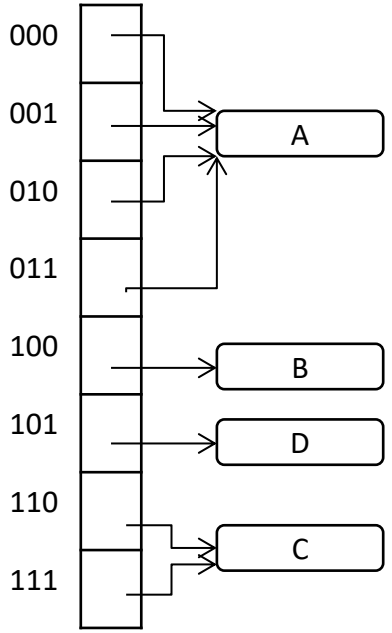


Resumindo

- Quando o hashing está vazio, a profundidade global é zero, assim como a profundidade local do *bucket* ligado ao diretório
- Sempre que o diretório dobrar o tamanho, a profundidade global é incrementada em 1
- No momento da divisão de um *bucket* (*overflow*):
  - Se a profundidade do *bucket* a ser dividido for menor que profundidade global, um novo *bucket* é criado e mapeado no diretório existente
  - Se a profundidade do *bucket* a ser dividido for igual a profundidade global, o diretório deverá ser aumentado antes da divisão, causando incremento na profundidade global
  - Sempre que um *bucket* é dividido, sua profundidade é incrementada em 1 e o novo *bucket* passa a ter a mesma profundidade daquele que o originou

Hashing extensível

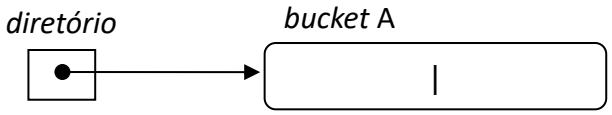
- Na figura, temos um diretório “desbalanceado”
  - Ele tem o dobro do tamanho necessário
- O bom funcionamento do *hashing* extensível continua dependendo da função *hash*
  - Se a função *hash* produz uma distribuição ruim, a tendência é que o diretório cresça além do necessário



- A função *hash* para um *hashing* extensível pode ser a mesma usada no *hashing* estático, mas a modulação pelo tamanho do espaço de endereços não é mais necessária
  - Em contrapartida, precisamos de uma função para analisar o endereço *hash* gerado bit a bit

Exercício 1

- Suponha um *hashing* extensível em que cada *bucket* pode armazenar dois registros. Inicialmente o sistema está vazio (as profundidades global e local são iguais a zero)

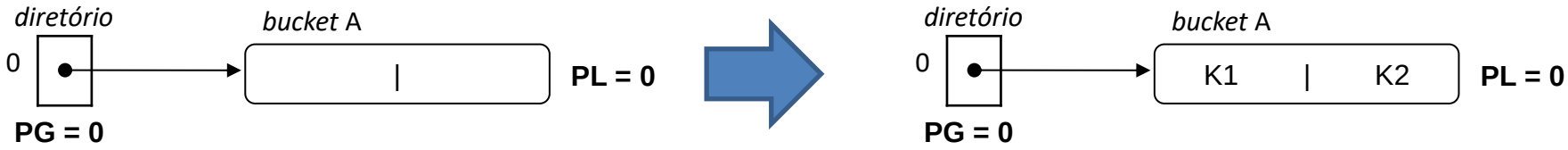


  - Mostre como ficam o diretório e os *buckets* após a inserção das chaves k1, k2, k3, k4, k5 e k6, nesta ordem, sabendo que os respectivos endereços bases são: 1111, 0000, 1100, 1010, 0101, 1110. Indique as profundidades global e locais nas representações geradas

h(K1) = 1111
h(K2) = 0000

Inserção

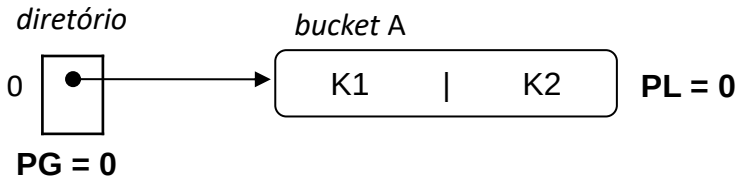
Insira K1 e K2



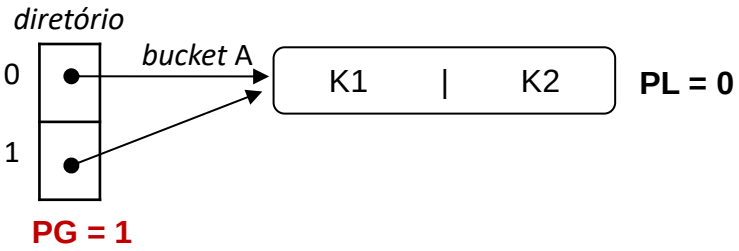
h(K1) = 1111
h(K2) = 0000
h(K3) = 1100

Inserção

Insira K3



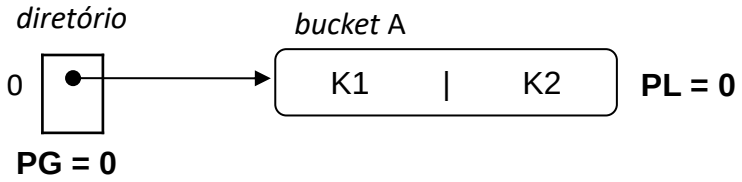
Overflow no *bucket* A
→ aumente o diretório



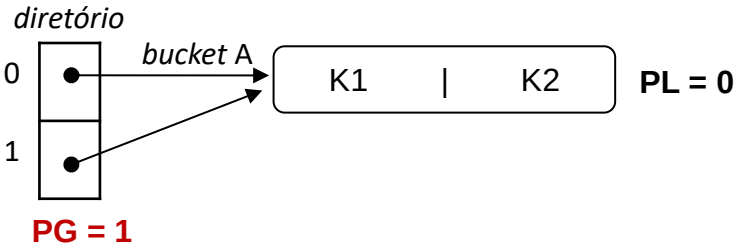
h(K1) = 1111
h(K2) = 0000
h(K3) = 1100

Inserção

Insira K3

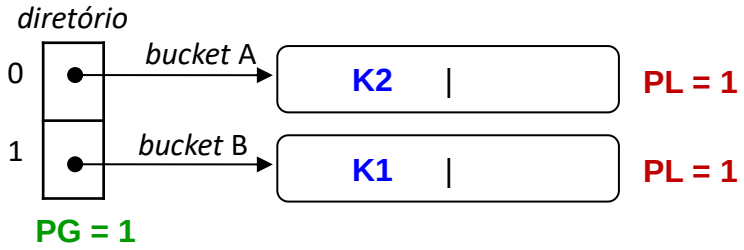


Overflow no *bucket* A
→ aumente o diretório



Divida o *bucket* A e
remapeie as chaves
usando um bit

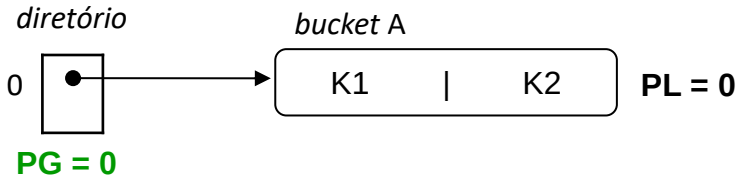
K1 = **1**111
K2 = **0**000



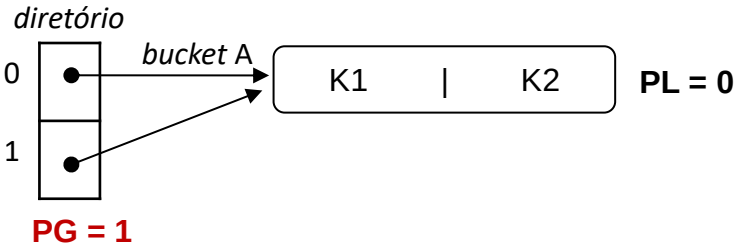
h(K1) = 1111
h(K2) = 0000
h(K3) = 1100

Inserção

Insira K3

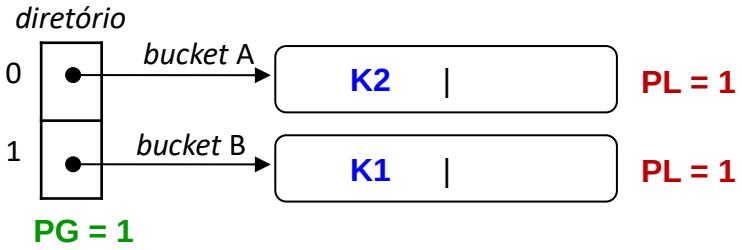


Overflow no *bucket* A
→ aumente o diretório



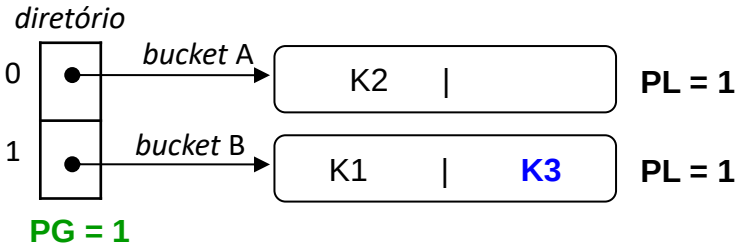
Divida o *bucket* A e remapeie as chaves usando um bit

K1 = **1**111
K2 = **0**000



Insira K3

K3 = **1**100

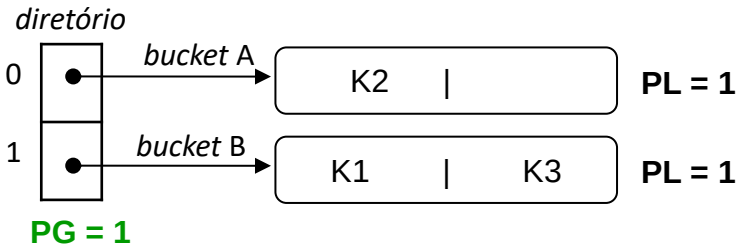


h(K1) = 1111
h(K2) = 0000
h(K3) = 1100
h(K4) = 1010

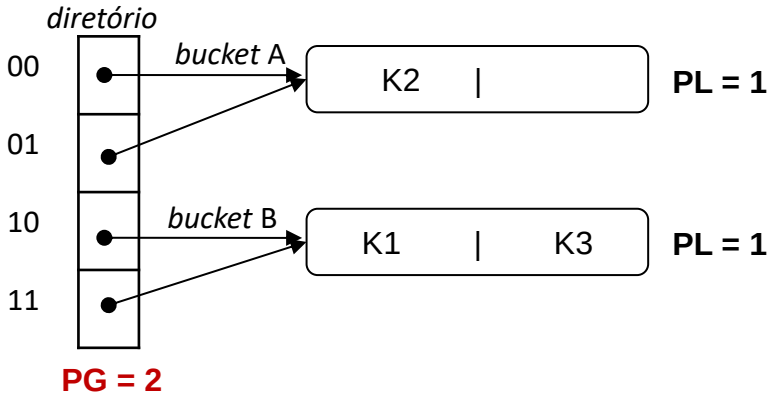
Inserção

Insira K4

K4 = 1010



Overflow no *bucket* B → aumente o diretório

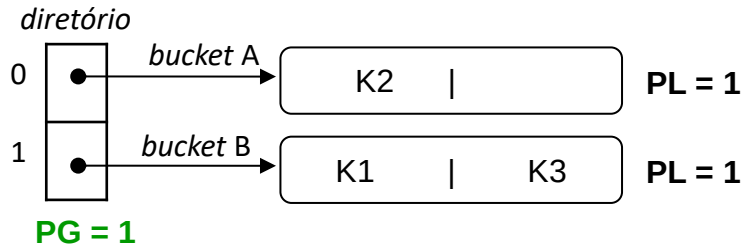


h(K1) = 1111
h(K2) = 0000
h(K3) = 1100
h(K4) = 1010

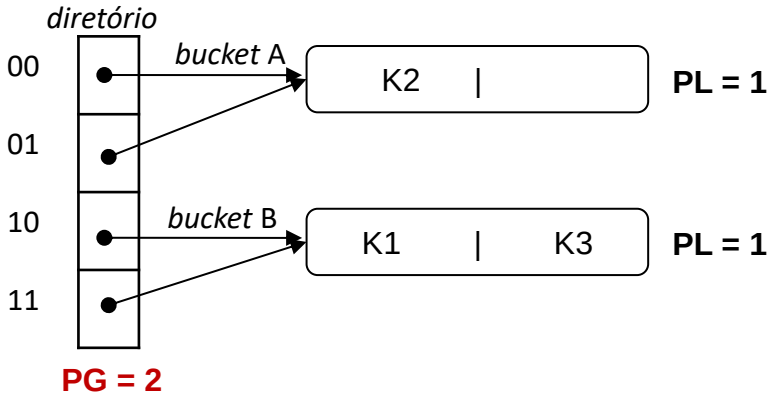
Inserção

Insira K4

K4 = **1**010

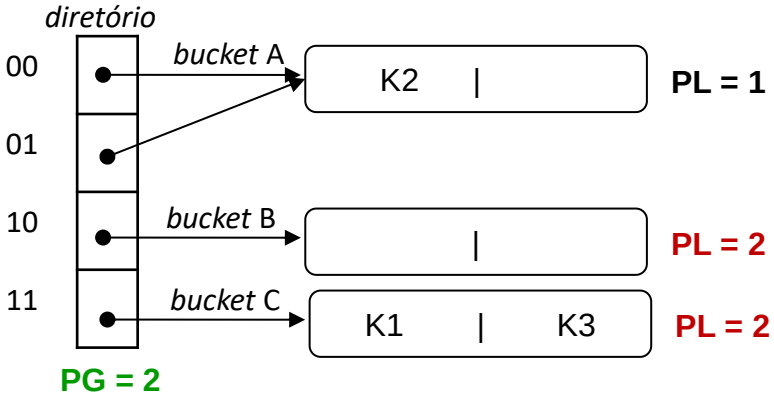


Overflow no *bucket* B → aumente o diretório



Divida o *bucket* B e remapeie as chaves usando dois bits

K1 = **11**11
K3 = **11**00

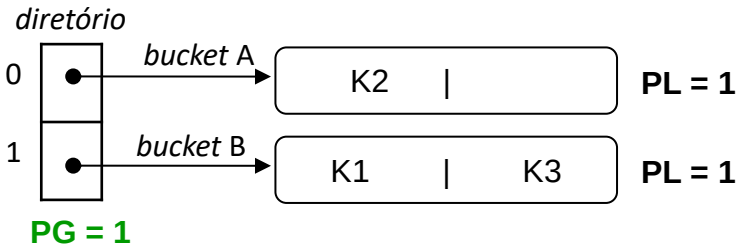


h(K1) = 1111
h(K2) = 0000
h(K3) = 1100
h(K4) = 1010

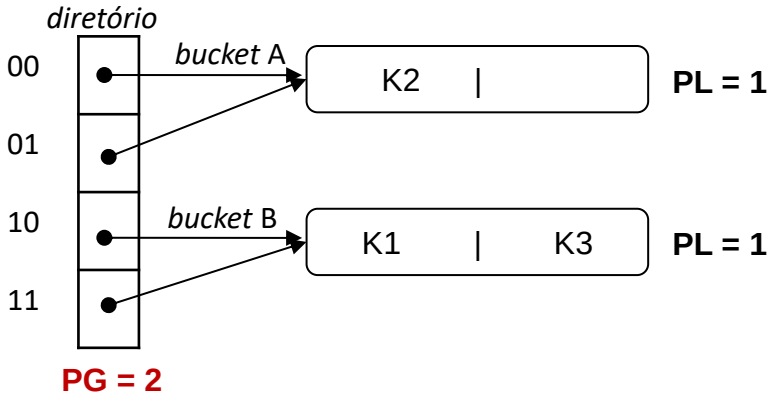
Inserção

Insira K4

K4 = 1010

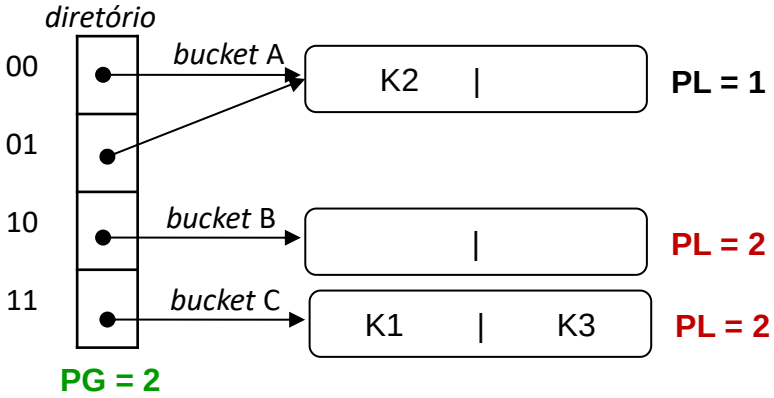


Overflow no *bucket* B → aumente o diretório



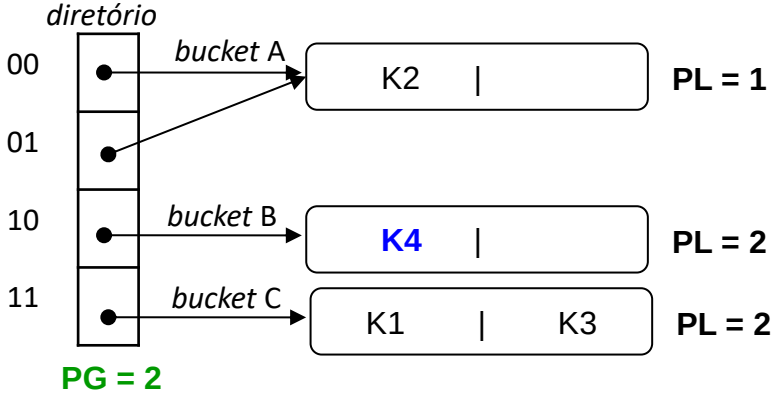
Divida o *bucket* B e remapeie as chaves usando dois bits

K1 = 1111
K3 = 1100



Insira K4

K4 = 1010

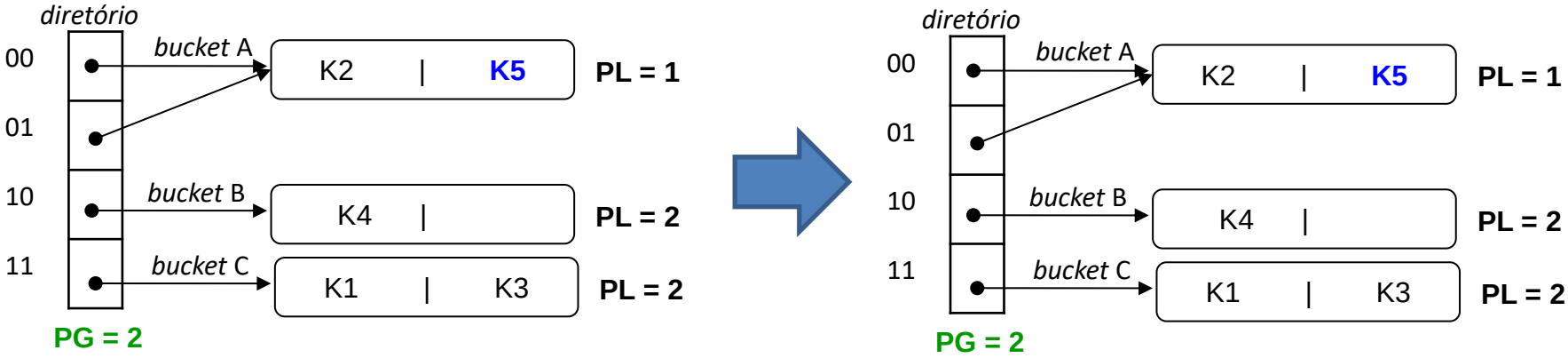


h(K1) = 1111
h(K2) = 0000
h(K3) = 1100
h(K4) = 1010
h(K5) = 0101

Inserção

Insira K5

K5 = **01**01

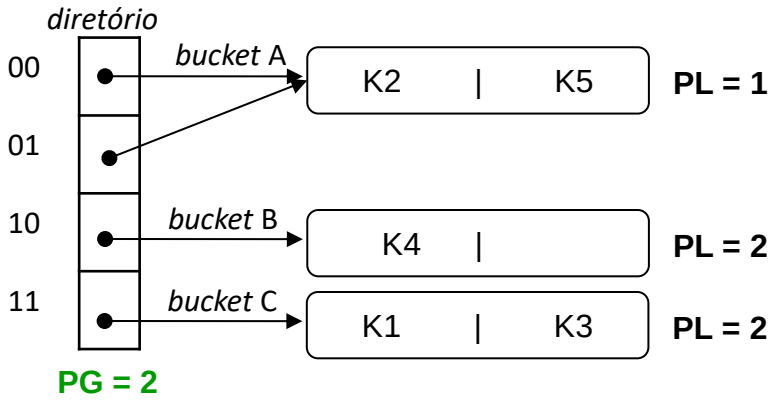


h(K1) = 1111
h(K2) = 0000
h(K3) = 1100
h(K4) = 1010
h(K5) = 0101
h(K6) = 1110

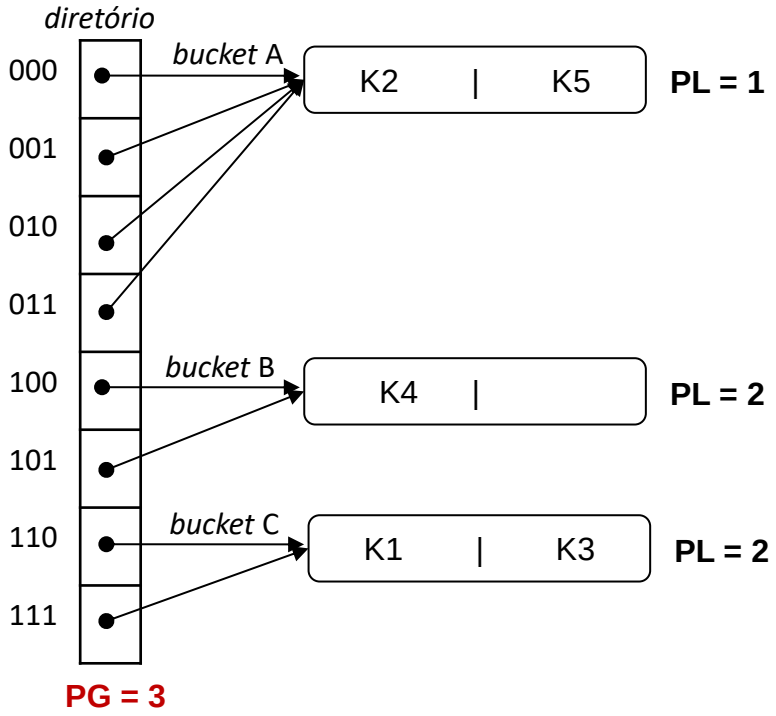
Inserção

Insira K6

K6 = **11**10



Overflow no *bucket* C → aumente o diretório



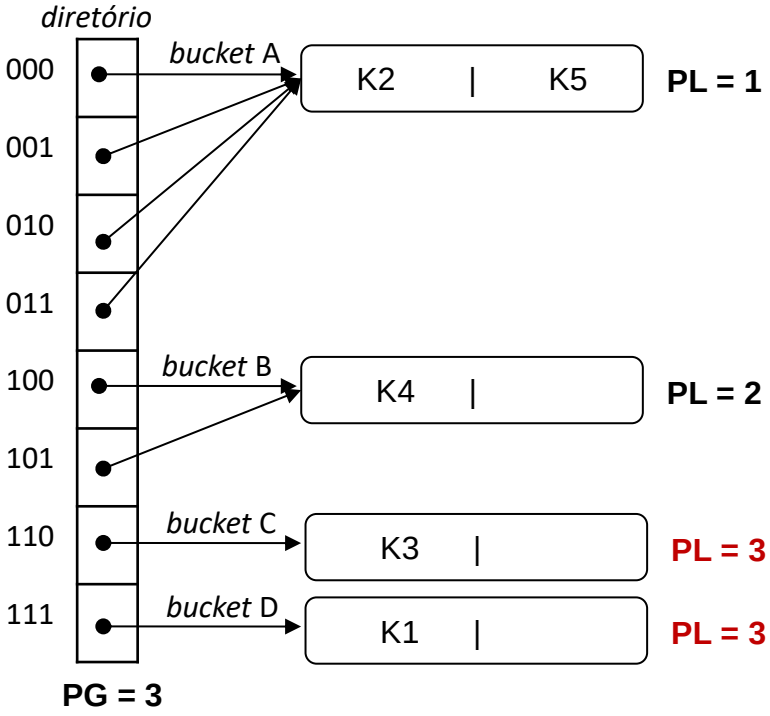
h(K1) = 1111
h(K2) = 0000
h(K3) = 1100

h(K4) = 1010
h(K5) = 0101
h(K6) = 1110

Inserção

Divida o *bucket* C e remapeie as chaves usando três bits

K1 = **111**1
K3 = **110**0

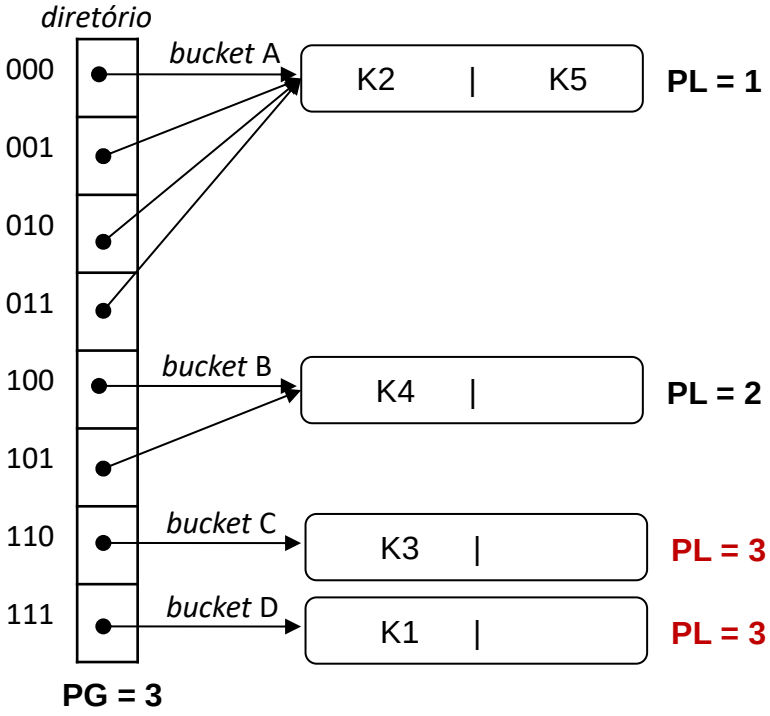


h(K1) = 1111
h(K2) = 0000
h(K3) = 1100
h(K4) = 1010
h(K5) = 0101
h(K6) = 1110

Inserção

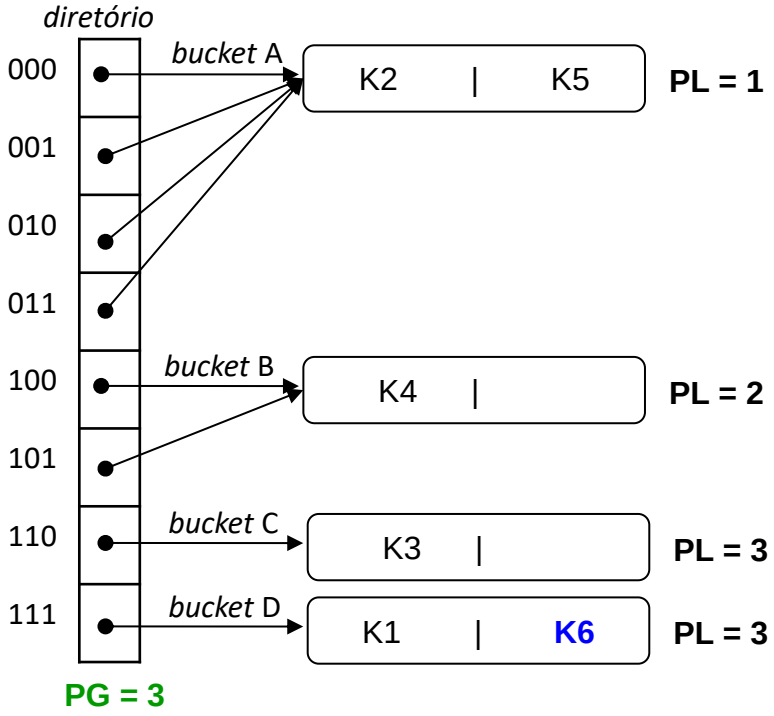
Divida o *bucket* C e remapeie as chaves usando três bits

K1 = **111**1
K3 = **110**0



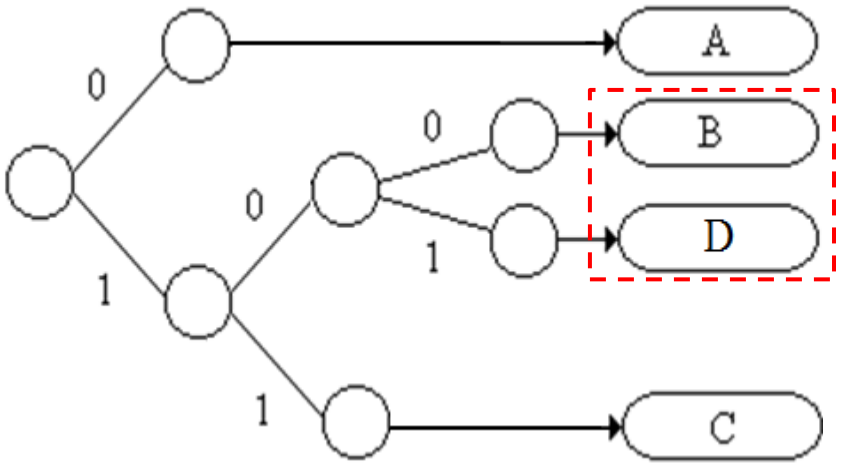
Insira K6

K6 = **111**0



Remoção

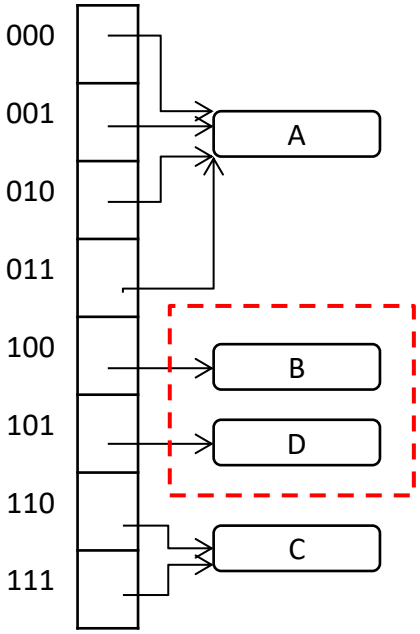
- Para ser uma estrutura dinâmica, não basta que o *hashing* extensível seja capaz de aumentar de tamanho, ele também deve ser capaz de diminuir
- Relembrando a concatenação nas árvores-B
  - Apenas páginas irmãs imediatas podiam ser concatenadas
  - A concatenação sempre se iniciava com duas páginas folhas
- No *hashing* extensível
  - Dois *buckets* podem ser concatenados apenas quando são **filhos imediatos de um mesmo nó na *trie***
  - Além disso, eles **devem estar no nível mais baixo da *trie***



Remoção

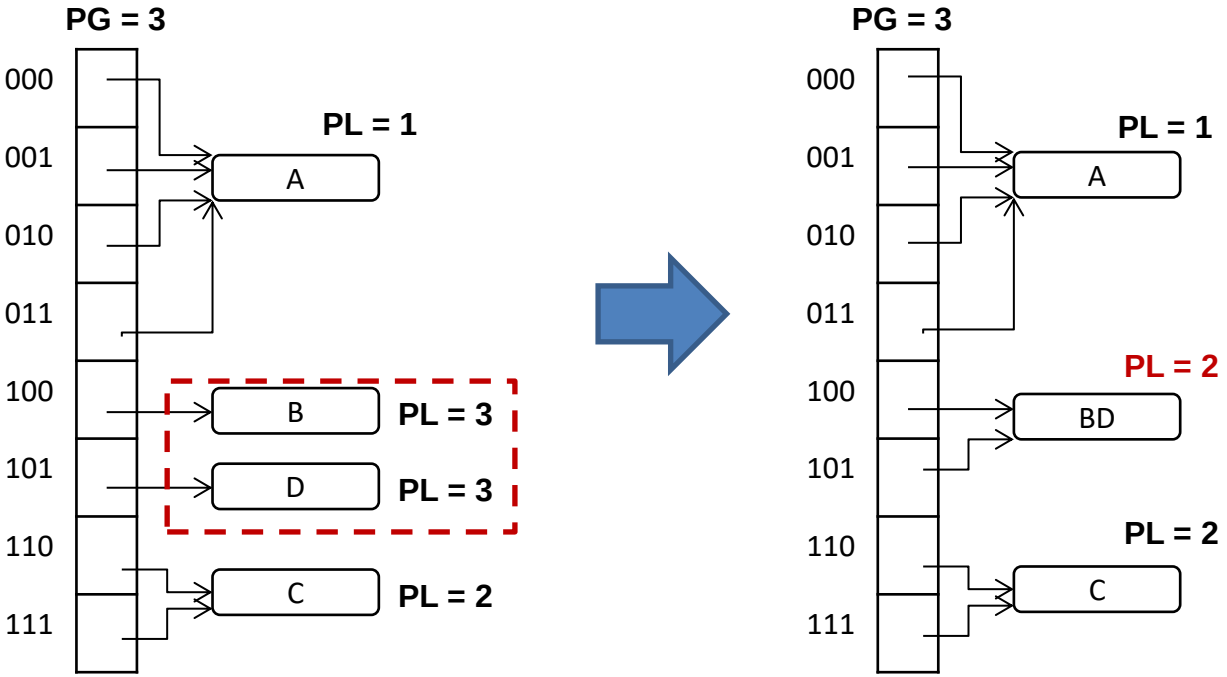
- Como saber se o *bucket* está no nível mais baixo da *trie*?
  - Comparando a profundidade local com a global
  - Se elas forem iguais, o *bucket* está no último nível
- Dado um determinado *bucket,* como encontrar o seu "par"?
  - Todo *bucket* tem um único par
  - O par pode ser encontrado invertendo-se o bit menos significativo do endereço do *bucket*
- Chamaremos dois *buckets* que podem ser concatenados de "amigos" (*buddy buckets*)

B e D são *buckets* amigos



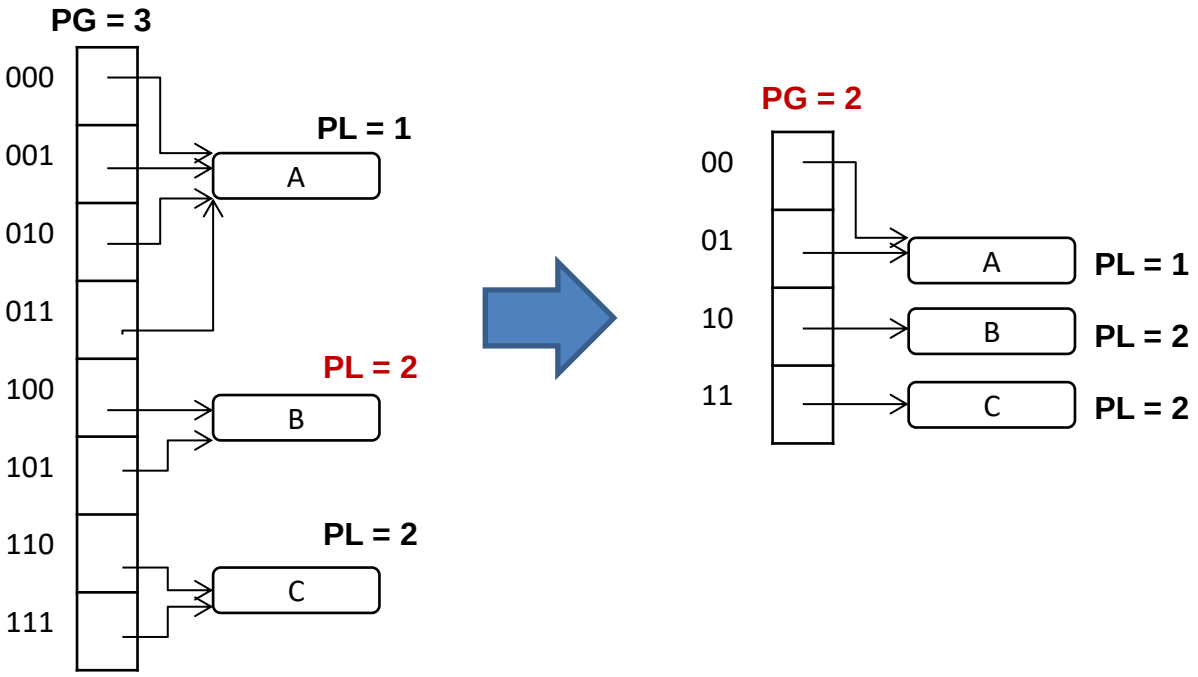
Remoção

- Quando os *buckets* amigos são concatenados, a profundidade do *bucket* resultante diminui em 1



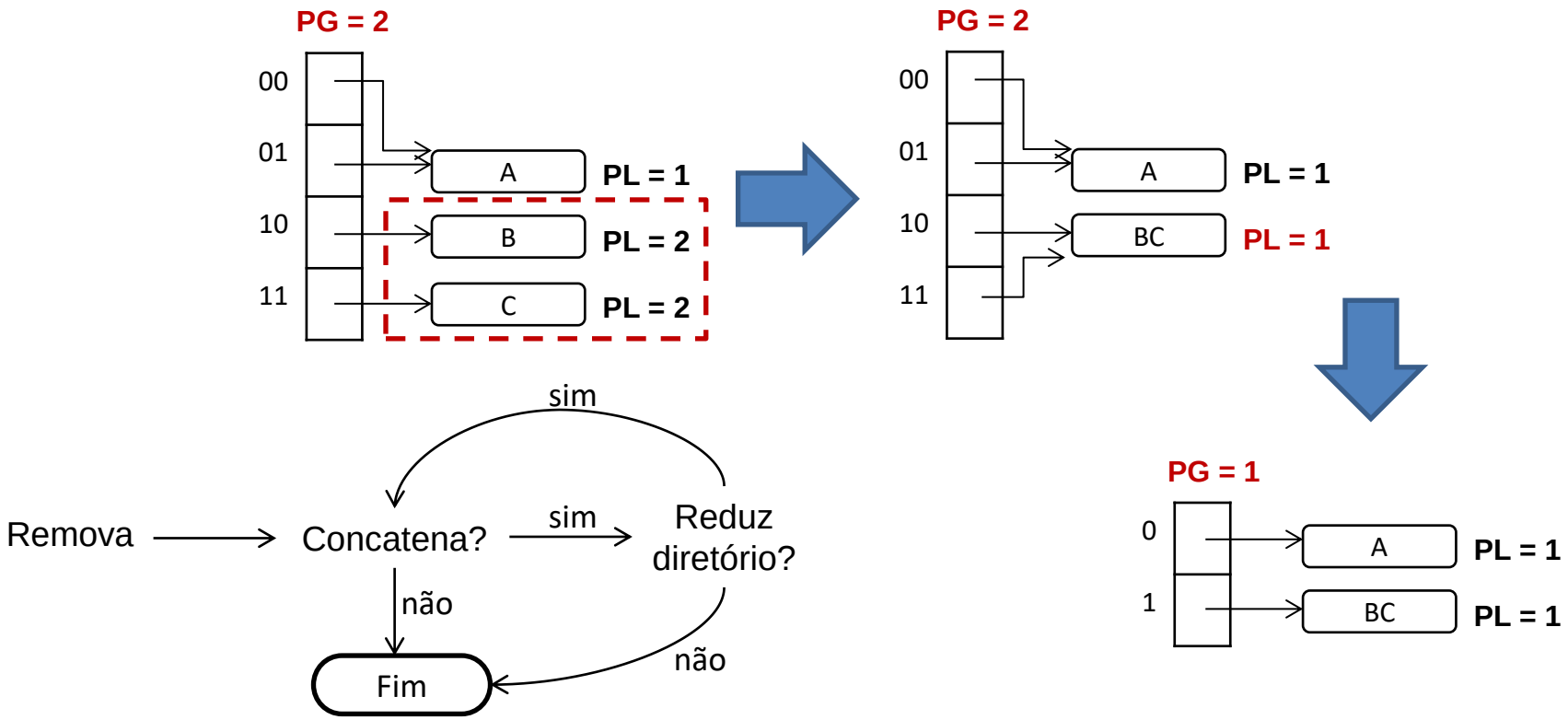
Remoção

- Após uma concatenação, é possível que o diretório possa reduzir o seu tamanho
  - Se todos os *buckets* possuírem pelo menos duas referências para eles, o diretório pode ser reduzido à metade do tamanho



Remoção

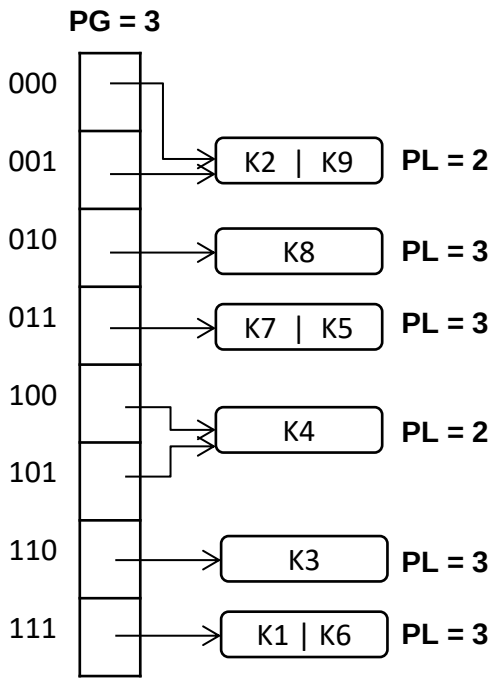
- Quando o diretório reduz de tamanho, é possível que um novo *bucket* amigo surja e uma nova concatenação possa ser realizada



Exercício 2

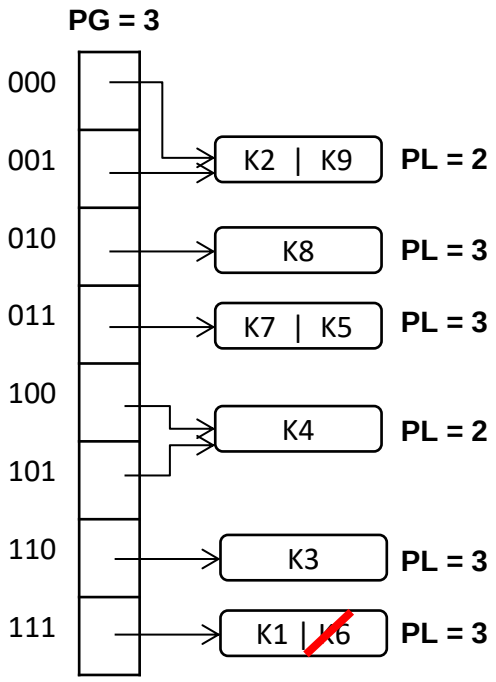
- Considere o *hashing* extensível mostrado na figura e simule as <u>remoções</u> das seguintes chaves, nesta ordem:

K6, K9, K7, K3, K1, K5, K8

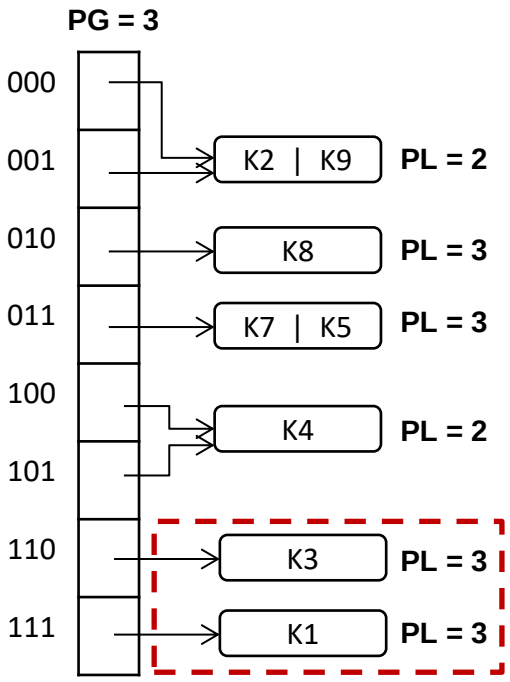


Remoção

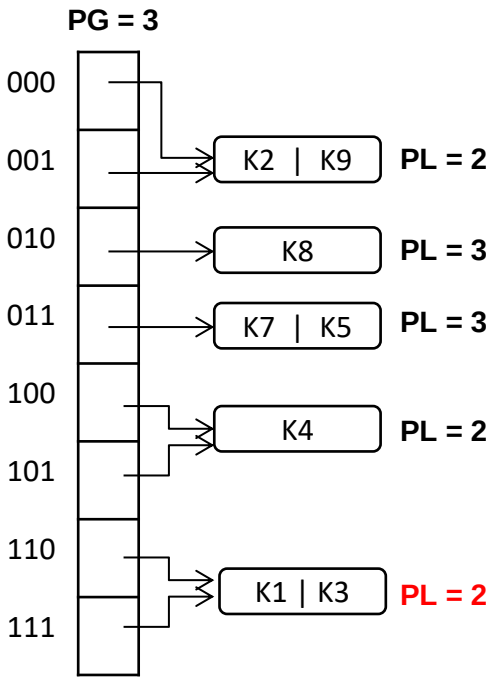
K6, K9, K7, K3, K1, K5, K8



Existe um *bucket* amigo



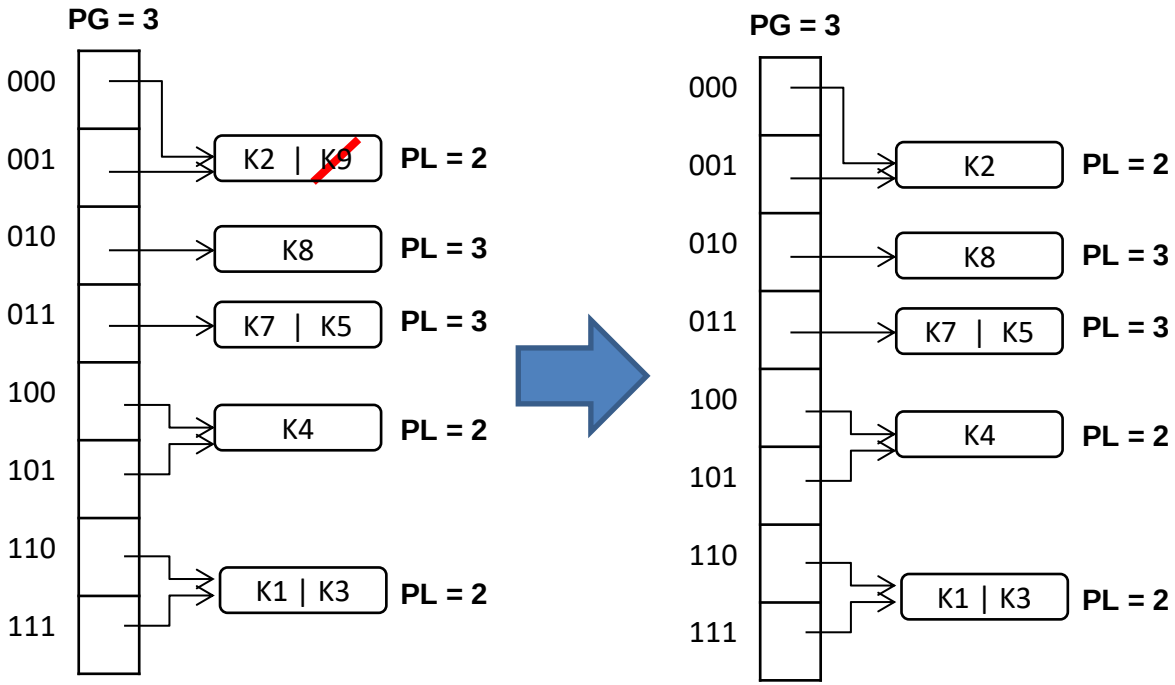
É possível concatenar com o amigo



Não é possível diminuir o diretório

Remoção

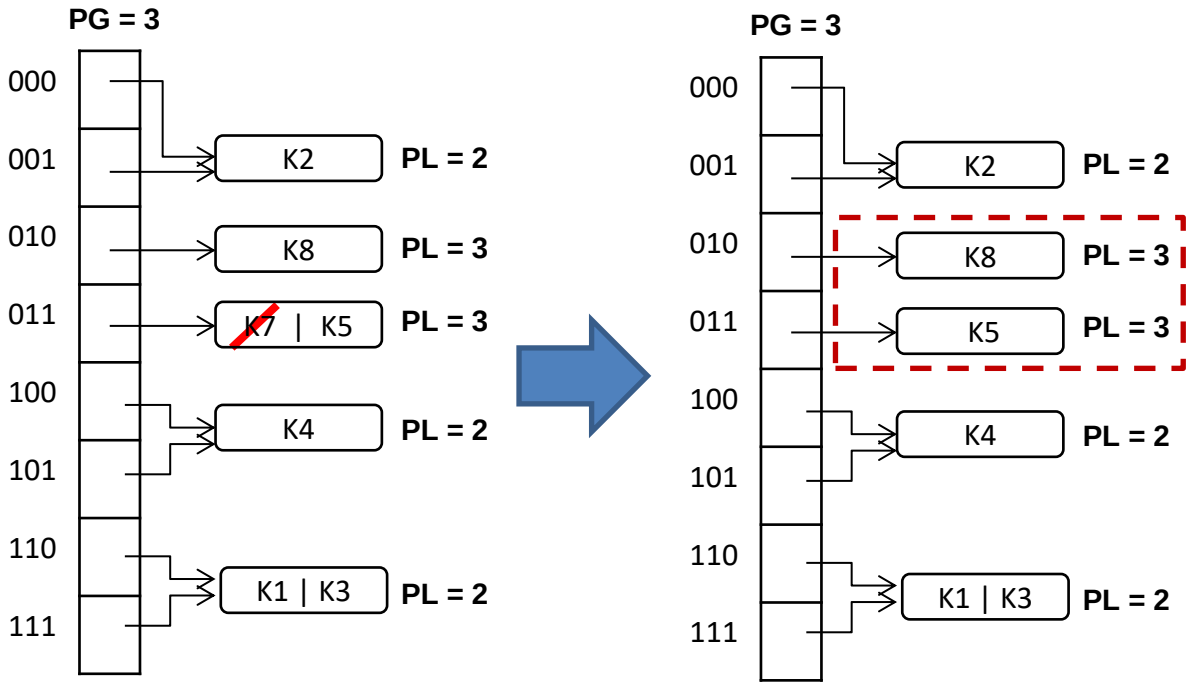
K6, K9, K7, K3, K1, K5, K8



Não existe um *bucket* amigo

Remoção

K6, K9, K7, K3, K1, K5, K8

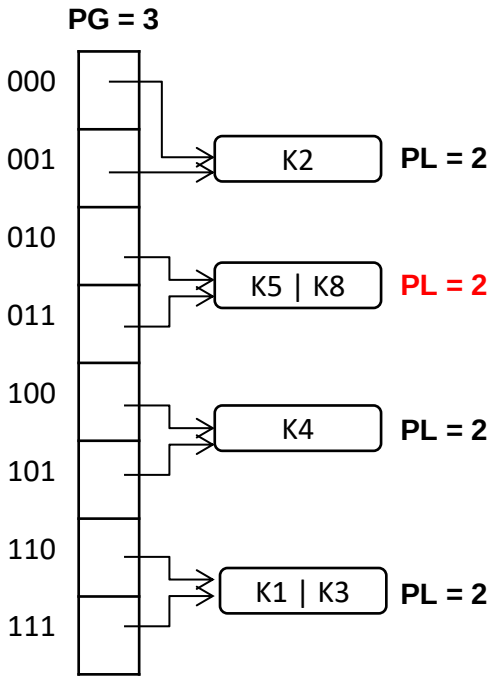


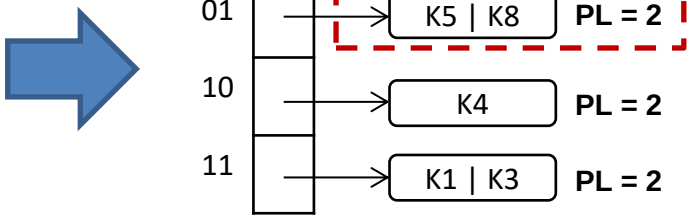
Existe um *bucket* amigo

É possível concatenar com o amigo

Remoção

K6, K9, K7, K3, K1, K5, K8



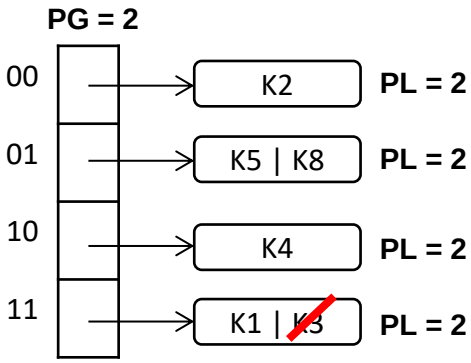


É possível
diminuir o diretório

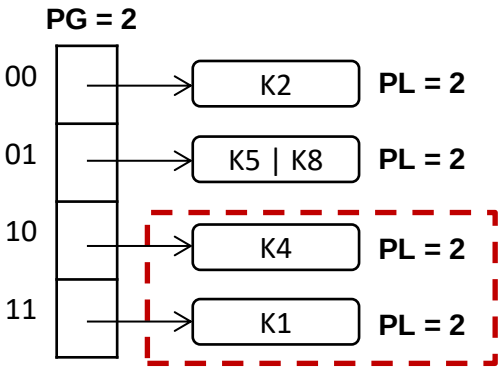
Não é possível concatenar
com o novo amigo

Remoção

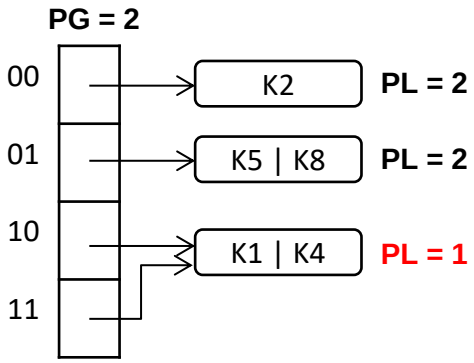
K6, K9, K7, K3, K1, K5, K8



Existe um *bucket* amigo



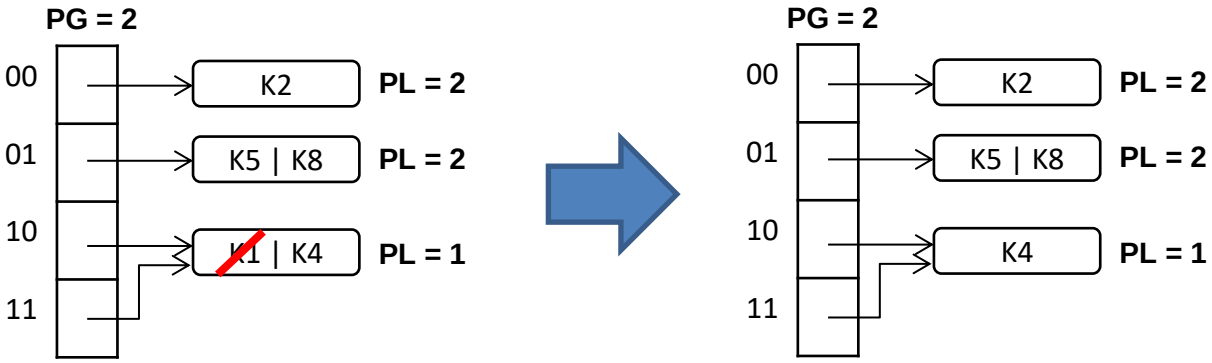
É possível concatenar com o amigo



Não é possível diminuir o diretório

Remoção

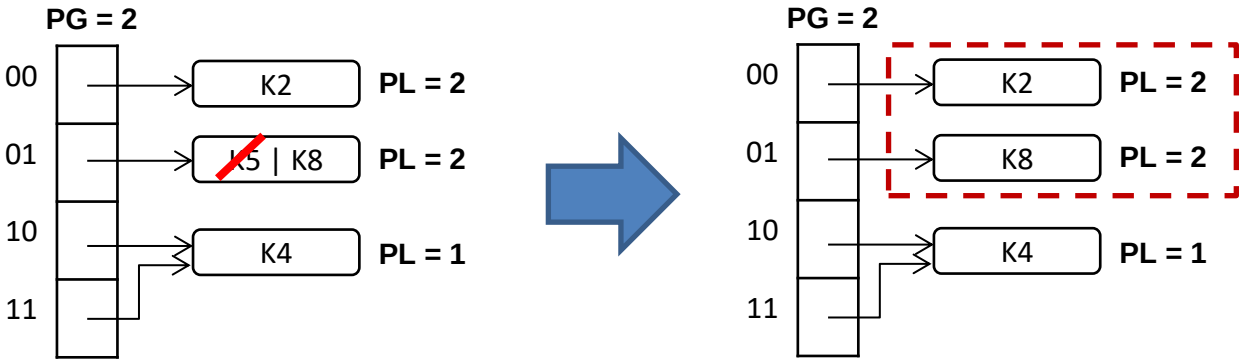
K6, K9, K7, K3, K1, K5, K8



Não existe um *bucket* amigo

Remoção

K6, K9, K7, K3, K1, K5, K8

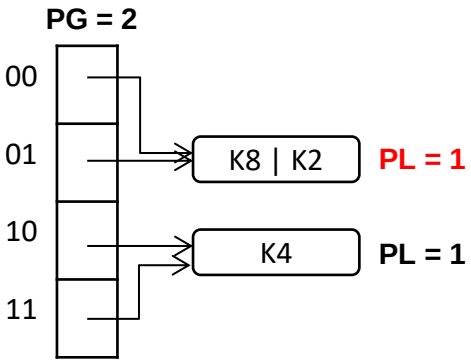


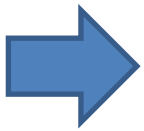
Existe um *bucket* amigo

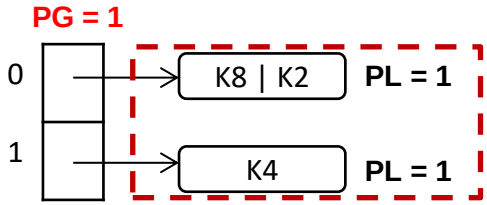
É possível concatenar com o amigo

Remoção

K6, K9, K7, K3, K1, K5, K8





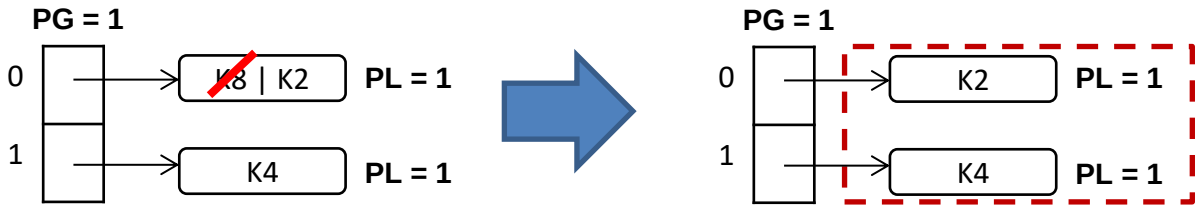


É possível
diminuir o diretório

Não é possível concatenar
com o novo amigo

Remoção

K6, K9, K7, K3, K1, K5, K8

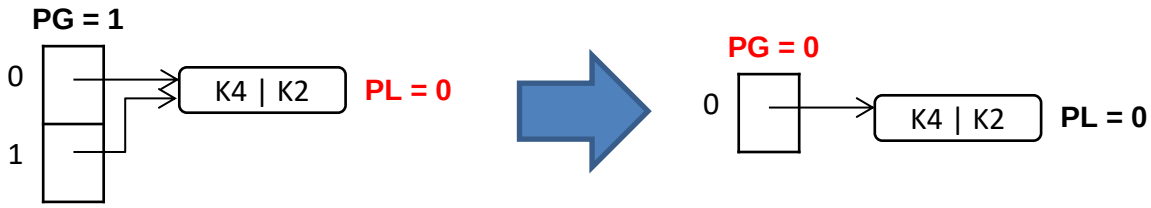


Existe um *bucket* amigo

É possível concatenar com o amigo

Remoção

K6, K9, K7, K3, K1, K5, K8



É possível diminuir o diretório

Não existe um novo amigo