ALGORITMOS E ESTRUTURAS DE DADOS II

Tabelas Hash 2

Karina Valdivia Delgado

Roteiro

- Revisão da aula anterior
- Hashing em memória secundária
  - Hashing estático em memória secundária
  - Hashing dinâmico em memória secundária
    - Hashing estensível
    - Hashing linear

Revisão da aula anterior

Construindo boas funções hash

Uma boa função de hash de boa qualidade satisfaz a **suposição de hash uniforme simples**:

Qualquer chave **k** tem igual probabilidade de ser colocada em qualquer das **m** posições (slots) da tabela

Funções hash

Divisão

$h(k)=k \bmod m$

- m não deve ser uma potência de 2 pois h(k) corresponderia aos p bits menos significativos de k.
- escolher um primo não muito próximo de uma potência de 2 para m

Multiplicação…

$h(k)=\lfloor m((kA) \bmod 1) \rfloor$

- $0<A<1$
- Funciona bem com $A=(\sqrt{5}-1)/2$

Funções hash

Hashing Universal

Seja $H=\{h_1,h_2,...,h_t\}$ uma coleção finita de funções hash que mapeiam um dado universo U de chaves no conjunto $\{0, 1, . . . , m - 1\}$.

Tal coleção é dita ser universal se para cada par de chaves distintas $k, l \in U$, o número de funções hash para as quais $h_i(k) = h_i(l)$ é no máximo t/m, ou seja escolhendo aleatoriamente uma função de H, a chance de colisão é 1/m.

Funções hash

Hashing Universal

$h_{a,b}(k)=[(ak+b) \bmod p] \bmod m$

- p é um primo grande o suficiente tal que toda chave k esteja no intervalo 0 a p-1, p>m

- a, b são constantes escolhidas cada vez que monto a minha tabela hash, tal que:
  $a \in Z^*_p=\{1,2, \ldots, p-1\}$
  $b \in Z_p=\{0,1, \ldots, p-1\}$

$H_{p,m}=\{h_{a,b}: a \in Z^*_p \text{ e } b \in Z_p\}$

Funções hash

Hashing Universal

$h_{a,b}(k)=[(ak+b) \bmod p] \bmod m$

Exemplo:

p=17 e m=6

$h_{3,4}(8) = [(3*8+4) \bmod 17] \bmod 6 = 5$

$h_{3,4}(4) = [(3*4+4) \bmod 17] \bmod 6 = 4$

Fator de carga

Data uma tabela T com m slots armazenando n chaves, o fator de carga α é definido por:

$$\alpha = n/m$$

(número médio de elementos armazenados em cada posição de T)

Resolução de colisões

encadeamento

- Todos os elementos com o mesmo valor de h são colocados em uma lista ligada.
- Tempo esperado para uma busca sem sucesso ou para uma busca bem sucedida é: $\Theta(1+\alpha)$

endereçamento aberto

- Todos os elementos são armazenados na própria tabela, não usa ponteiros.
- Busca sem sucesso: $O(1/(1-\alpha))$

Busca bem sucedida: $O(\alpha \ln [1/(1-\alpha)])$

Endereçamento aberto

Para inserir é feita uma **sondagem**, isto é, examinamos sucessivamente a tabela até encontrar um slot vazio no qual a chave será inserida.

Para determinar a seqüência de posições a serem examinadas, a função de hashing é estendida de maneira a incluir o número do teste (tentativa) como segunda entrada:

$$h : U \times \{0, 1 \ldots m - 1\} \rightarrow \{0, 1 \ldots m - 1\}$$

Endereçamento aberto

sondagem linear

$h(k,i)=(h'(k)+i) \bmod m$

- Número de sequências possíveis O(m) uma vez que a posição inicial determina toda a sequência posterior.
- Problema: agrupamento primário: longas sequências de slots ocupados.

sondagem quadrática ...

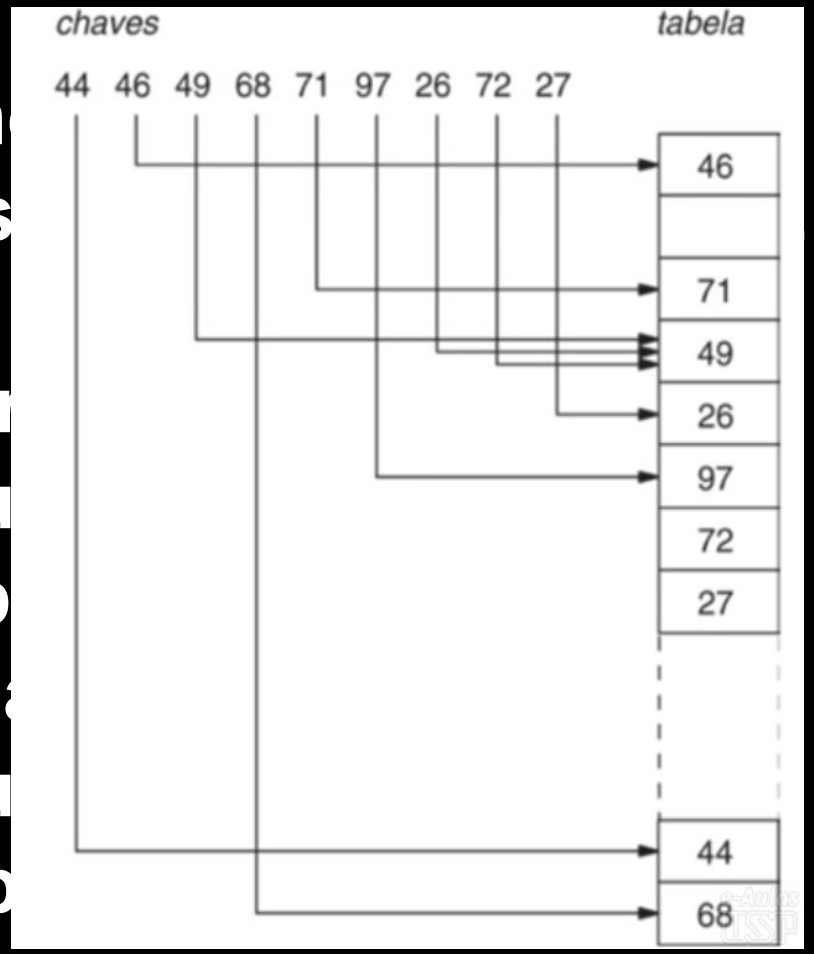
$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

- Número de sequências possíveis O(m) uma vez que a posição inicial determina toda a sequência posterior.
- Problema: agrupamento quadrático: se duas chaves têm a mesma posição inicial base, a sequência de sondagem será a mesma.

Endereçamento aberto

sondagem linear

$h(k,i)=(h'(k)+i) \bmod m$



- Núm
  poss
  que
  deter
  sequ
- Prob
  prim
  sequ
  ocup

sondagem quadrática ...

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

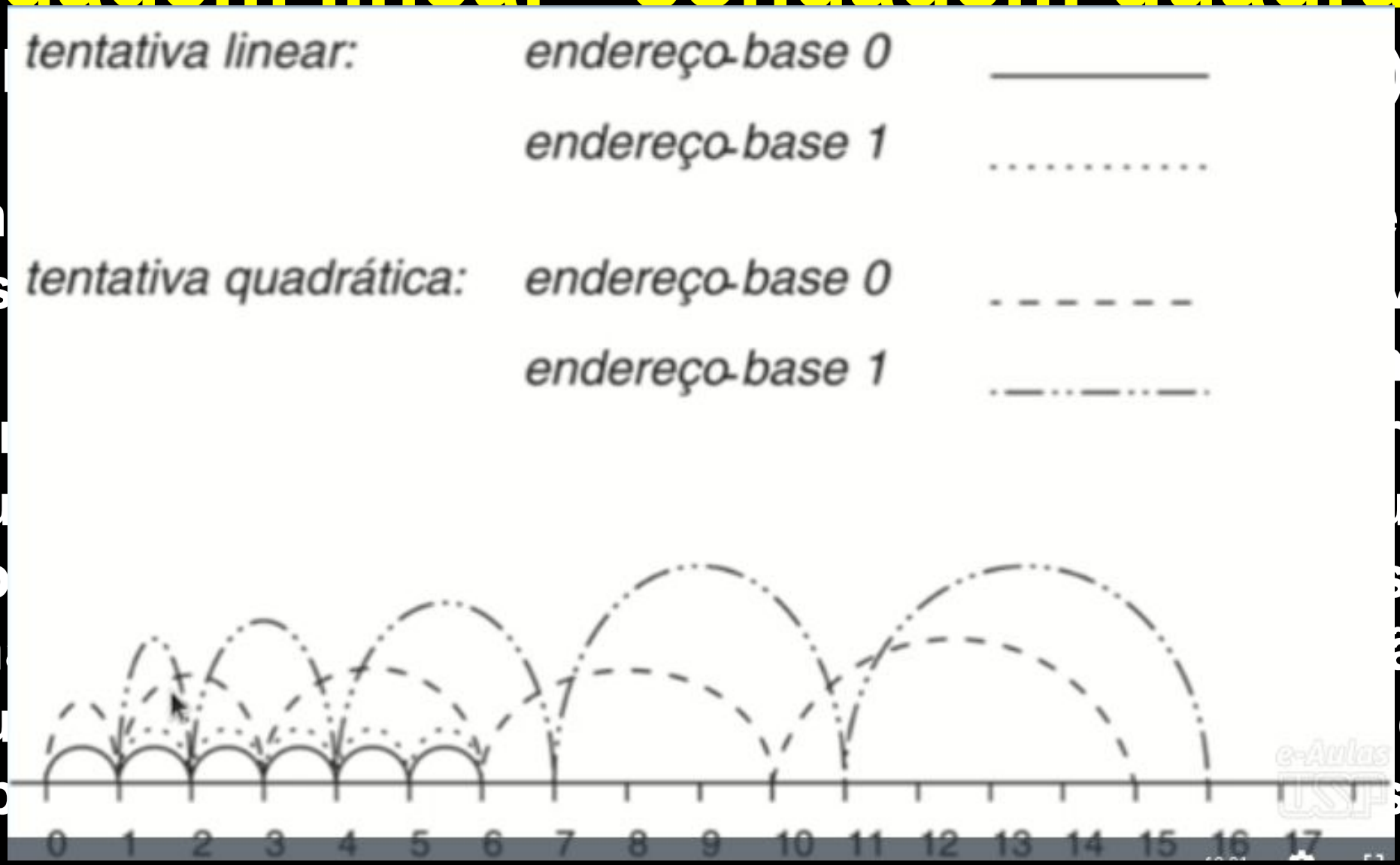
- Número de sequências possíveis O(m) uma vez que a posição inicial determina toda a sequência posterior.
- Problema: agrupamento quadrático: se duas chaves têm a mesma posição inicial base, a sequência de sondagem será a mesma.

Endereçamento aberto

sondagem linear sondagem quadrática ...

h(k,i)=( ) mod m

- Núm quências poss vez que a que nina toda deter . sequ upamento
- Prob s chaves prim ão inicial sequ cia de ocup sma.



Endereçamento aberto

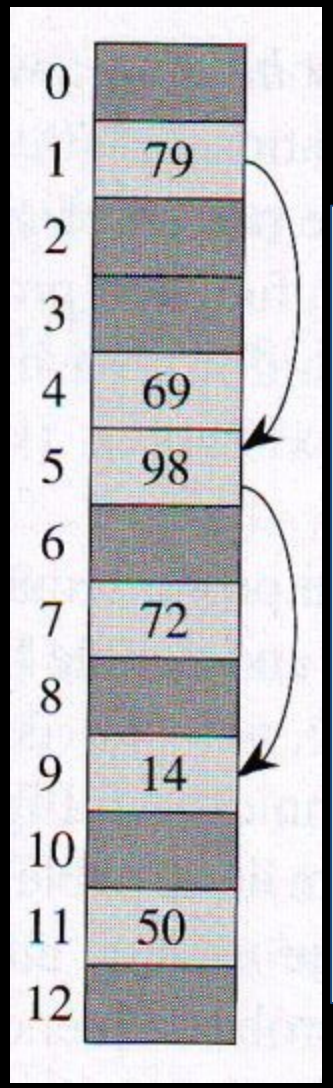
… hashing duplo

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

- Número de sequências possíveis $O(m^2)$, pois cada par $h_1(k)$ e $h_2(k)$ gera uma sequência distinta.
- Para que a tabela inteira seja pesquisada, o valor de $h_2(k)$ e o tamanho m da tabela devem ser primos entre si. Uma das formas de fazer isso é:
  - Fazer m igual a um número primo e projetar h_2 para retornar um inteiro positivo sempre menor que m.

Endereçamento aberto

… hashing duplo



$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

Ex: $m=13$, $h_1(k)=k \bmod m$, $h_2(k)=1+k \bmod (m-2)$

$h(14,0)=14 \bmod 13 +0\ (1+14 \bmod 11) =1$

$h(14,1)=14 \bmod 13 +1\ (1+14 \bmod 11) =5$

$h(14,2)=14 \bmod 13 +2\ (1+14 \bmod 11) =9$

$h(27,0)=27 \bmod 13 +0\ (1+27 \bmod 11) =1$

$h(27,1)=27 \bmod 13 +1\ (1+27 \bmod 11) =7$

$h(27,2)=(27 \bmod 13 +2\ (1+27 \bmod 11)) \bmod 13 =0$

Hashing estático e hashing dinâmico
Hashing interno e hashing externo

Slides baseados nas videoaulas da profa. Ariane Machado Lima
https://eaulas.usp.br/portal/video?idItem=28723

Hashing estático

- O tamanho da tabela é constante.

(Em todas as técnicas, revisadas até agora, o tamanho da tabela m não muda e foi feita a suposição que estavamos trabalhando em memória principal)

Hashing dinâmico

- A tabela pode ser expandida ou encolhida

Hashing interno

- Hashing em memória principal
- Cada slot da tabela de hash é um registro
- Colisões:
  - Encadeamento: em lista ligada
  - Endereçamento aberto: em outro slot da própria tabela.

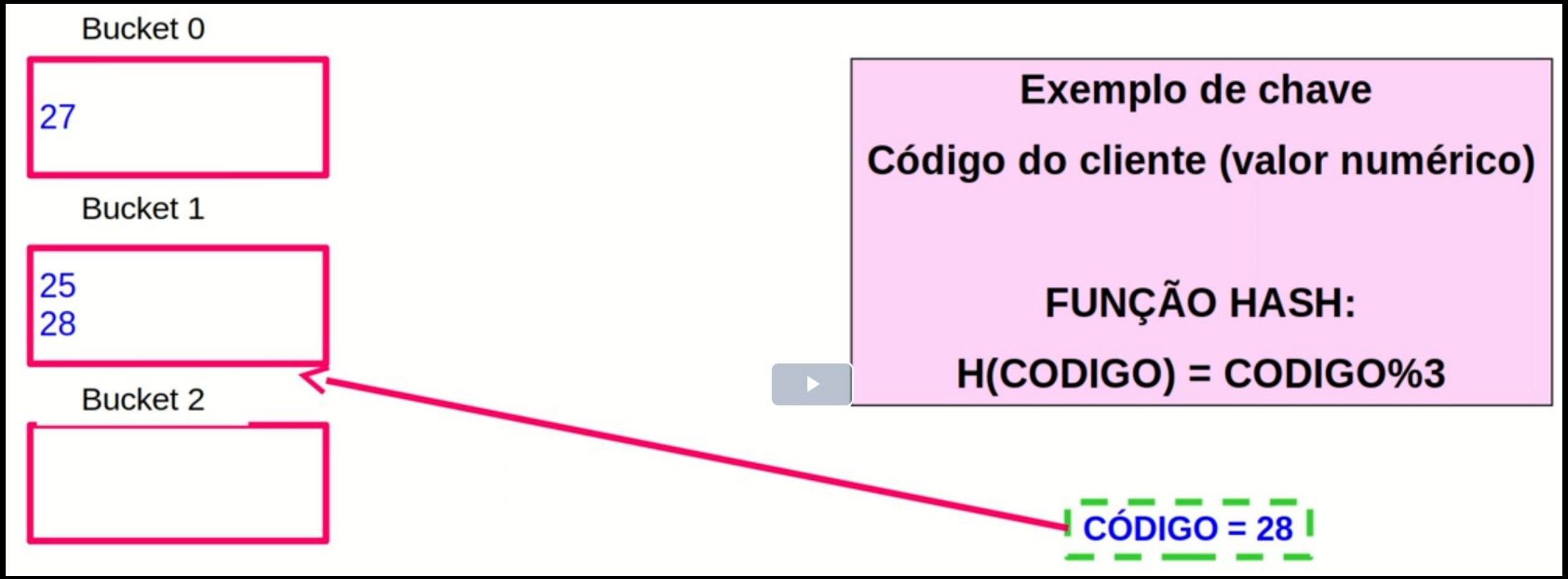
Hashing externo

- Hashing em memória secundária (armazenamento dos registros em disco)
- Cada slot da tabela de hash é um **bucket** (um bloco em disco)
- Colisões vão preenchendo o bucket
- Estamos preocupados com **overflow de buckets**
- Tabela hash fica no cabeçalho do arquivo
- Acessar um bucket significa realizar um **seek**

Hashing externo

- Tabelas hash podem ser usadas como uma alternativa a árvores B para fazer **busca em disco**.
  - Para obter o endereço do bloco de disco que contém o registro desejado
- Tabelas hash também podem ser usadas para construir **índices secundários**. Ex: indexar o campo Cargo da tabela Funcionário.

Hashing externo



Fonte: https://eaulas.usp.br/portal/video?idItem=28723

Fator de carga

- **Hashing interno:**

Data uma tabela T com m slots armazenando n chaves/registros, o fator de carga α é definido por:

$$\alpha = n/m$$

- **Hashing externo:**

$$\alpha = n/(m*r)$$

n = número de chaves/registros

m = número de slots (buckets)

r = número de registros que cabem em um bucket

Fator de carga

- **Hashing interno:**

Data uma tabela T com m slots armazenando n chaves/registros, o fator de carga α é definido por:

$$\alpha = n/m$$

- **Hashing externo:**

$$\alpha = n/(m*r)$$

n = número de chaves/registros
m = número de slots (buckets)
r = número de registros que cabem em um bucket

Cada bucket armazenar diversos registros → mais registros lidos/escritos de uma vez → menos seeks

Hashing estático em memória secundária

Slides baseados nas videoaulas da profa. Ariane Machado Lima
https://eaulas.usp.br/portal/video?idItem=28723

Overflow do bucket

Se h(x)=h(y)=i
Se o bucket i não estiver lotado, x e y vão para o bucket i
Caso contrário:

- **Encadeamento** - são usados buckets de overflow
  - Compartilhado
  - Exclusivo por endereço base
- **Endereçamento aberto** - vai para outro bucket usando alguma forma de sondagem.

Buckets de overflow compartilhado

- Buckets de overflow possuem uma **lista ligada de registros** que transbordaram de seus buckets
- No final dos buckets principais lotados existe um ponteiro para o próximo registro em um bucket de overflow.
- Existe uma lista ligada de registros desocupados nos buckets de overflow chamada de **lista livre**.
  - O início da lista livre pode ficar no cabeçalho do arquivo

Buckets de overflow compartilhados

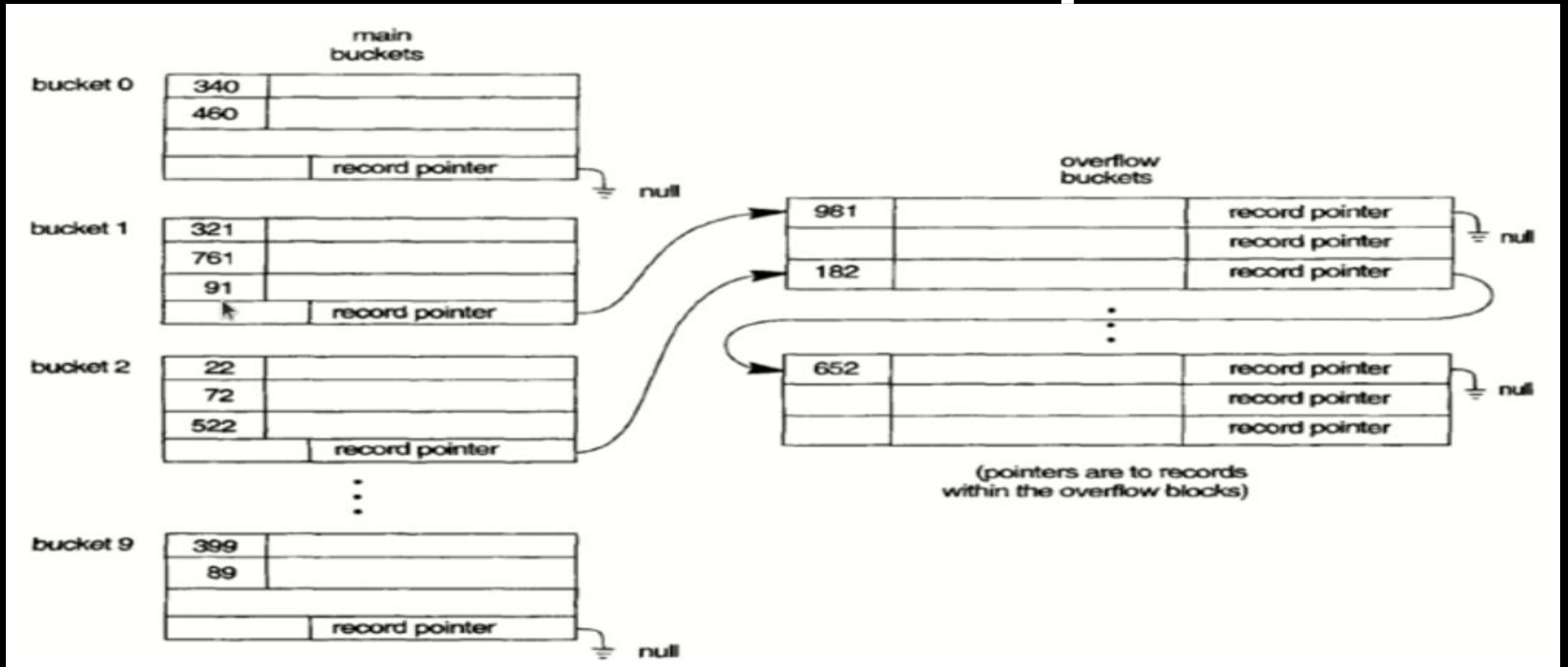


FIGURE 13.10 Handling overflow for buckets by chaining.

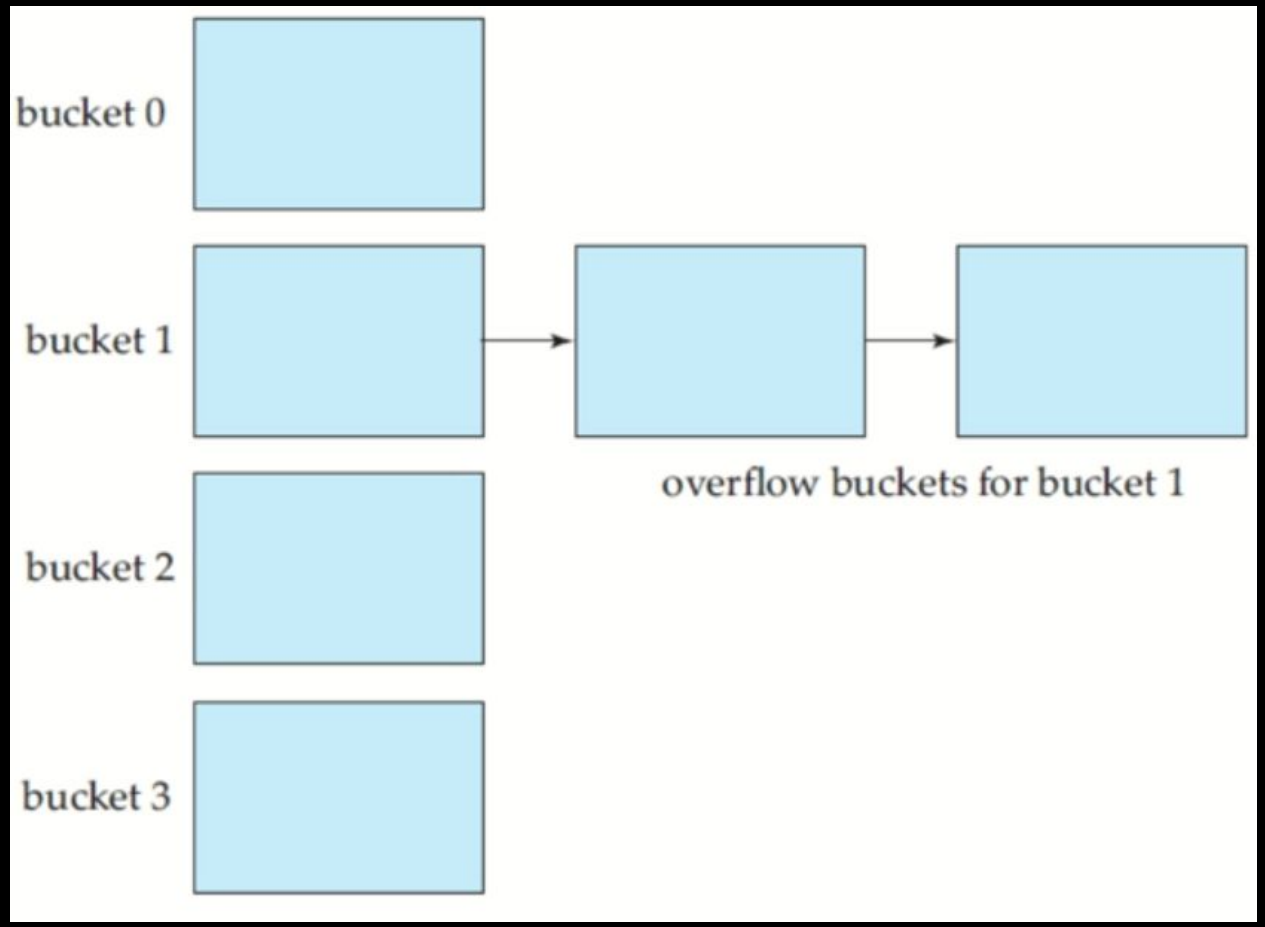
ELMARIS, NAVATHE, 2004

Buckets de overflow compartilhado

- **Busca:** calcular a função hash da chave e procurar no bucket principal correspondente, se não encontrar segue a **lista ligada de registros.**
- **Inserção:** em caso do bucket principal correspondente estar lotado, remove um espaço da lista livre e insere no início da lista ligada de registros (nos buckets de overflow)
- **Remoção:**
  - Se for em um bucket de overflow, adicionar o registro à lista livre
  - Se for no bucket principal, trazer algum registro de um bucket de overflow, se houver.

Buckets de overflow exclusivos

- **Lista ligada de buckets** de overflow para cada endereço base
- No final dos buckets lotados existe um ponteiro para o próximo bucket de overflow.



SILBERSCHATZ, 2011

Buckets de overflow exclusivos

- **Busca:** calcular a função hash da chave e procurar no bucket principal correspondente, se não encontrar segue a **lista ligada de buckets.**
- **Inserção:** insere no final do bucket principal ou no último bucket de overflow
- **Remoção:**
  - Remove e move para esse lugar o último registro do principal ou do último bucket de overflow se houver
  - Ou usa bit de validade e reorganiza depois, pois cada acesso a um bucket é um seek.

Buckets de overflow exclusivos

Quando comparado com a estratégia de buckets de overflow compartilhados que percorre listas ligadas de registros que podem estar espalhados por vários buckets de overflow compartilhados, a estratégia de usar buckets de overflow exclusivos:

- é mais simples, porém desperdiça mais espaço
- em média faz menos seeks

Endereçamento aberto

- **Busca:** segue a sequência de sondagens pelos buckets da tabela dada a função de hash h(k,i).
- **Inserção:** no primeiro bucket disponível identificado por endereçamento aberto.
- **Remoção:**
  - Se o bucket não ficar vazio, a sequência de sondagens não vai ser alterada, então tudo bem.
  - Se ficar, precisa ter os mesmos cuidados mencionados em endereçamento aberto para memória principal. Teremos o **bit de validade para o bucket**, para indicar que o bucket não tem registros, mas pode ter uma sequência depois.

Valor de m

$$m = n/r\ (1+d)$$

n = número de chaves/registros no arquivo
m = número de slots (buckets)
r = número de registros que cabem em um bucket
d = fator de fudge, tipicamente ao redor de 0.2
aproximadamente 20% do espaço dos buckets será perdido

Valor de m

- Em hashing estático o m é fixo
- O que fazer quando o arquivo é dinâmico, ou seja, arquivos que sofrem muitas inserções e remoções de registros?
  - Isso pode acontecer tanto em hashing em memória principal ou em memória secundária
  - Só que em memória secundária isso é mais crítico pois o overflow acarreta seeks.
  - O ideal seria que o tamanho da tabela de hash fosse dinâmico também, alterando-se com o tamanho do arquivo.

Hashing dinâmico em memória secundária

Hashing dinâmico

- Hashing extensível
- Hashing linear

Hashing extensível

Hashing extensível

- Primeiro é necessário definir uma função hash que transforme as chaves em números binários de tamanho fixo (uma sequência de b bits).
- É usado um diretório (tabela com ponteiros a buckets)
- Buckets podem ser divididos ou fundidos.

Hashing extensível

- Apenas os i primeiros bits da sequência são usados como endereço (i<=b).
  - Se i é o número de bits usados, a tabela de buckets terá 2^i entradas
  - Portanto, o tamanho da tabela de buckets cresce como potência de 2
- Seja r o número máximo de chaves/registros permitidos por bucket. O bucket está completo se tem r registros.

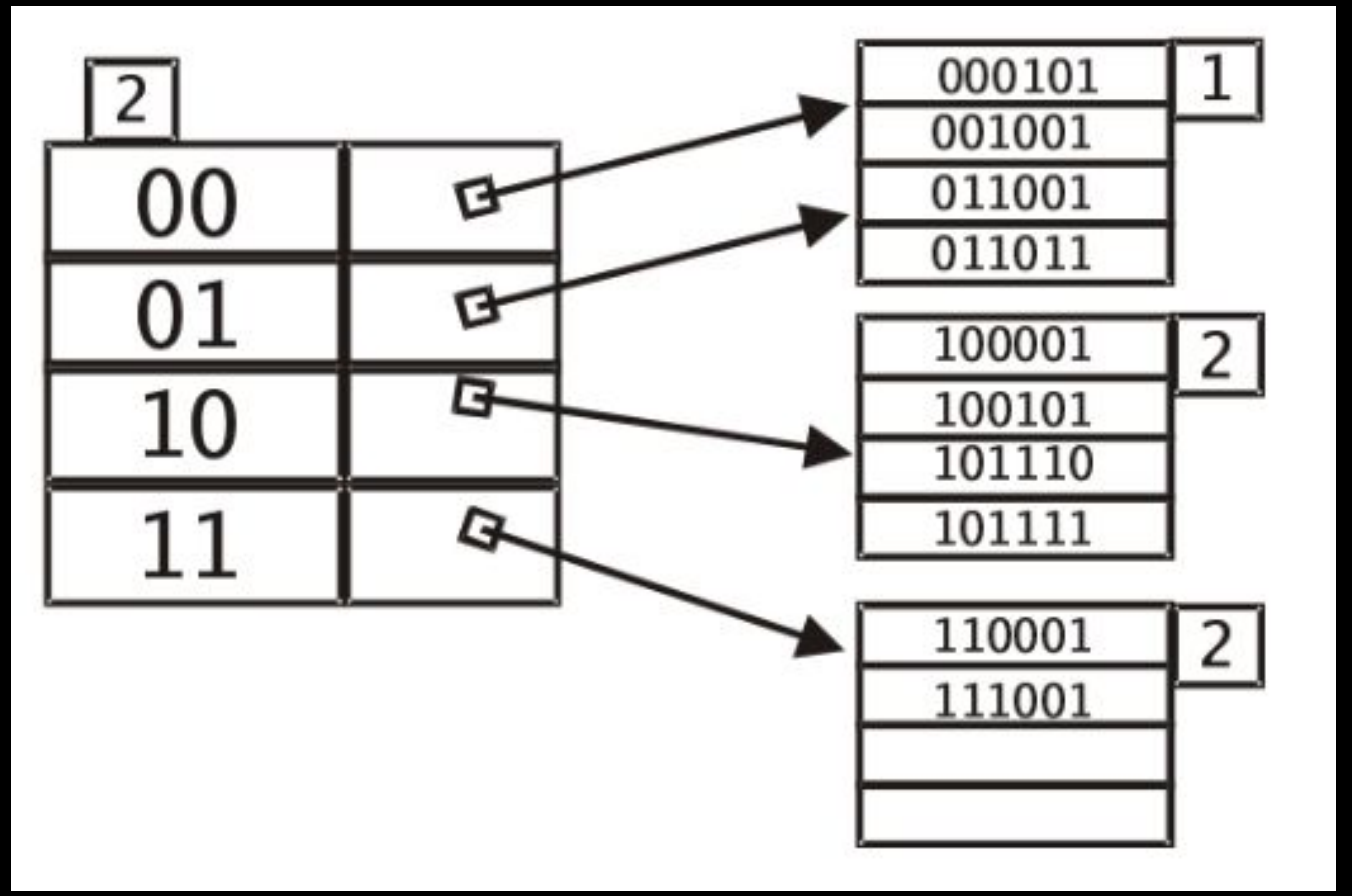
Hashing extensível

- O número i é chamado de profundidade global
- Cada bucket tem sua profundidade local j, j<=i, que indica que os primeiros j bits são iguais nesse bucket.
- É possível ter mais de um ponteiro apontando para o mesmo bucket.

Hashing extensível

Ex:

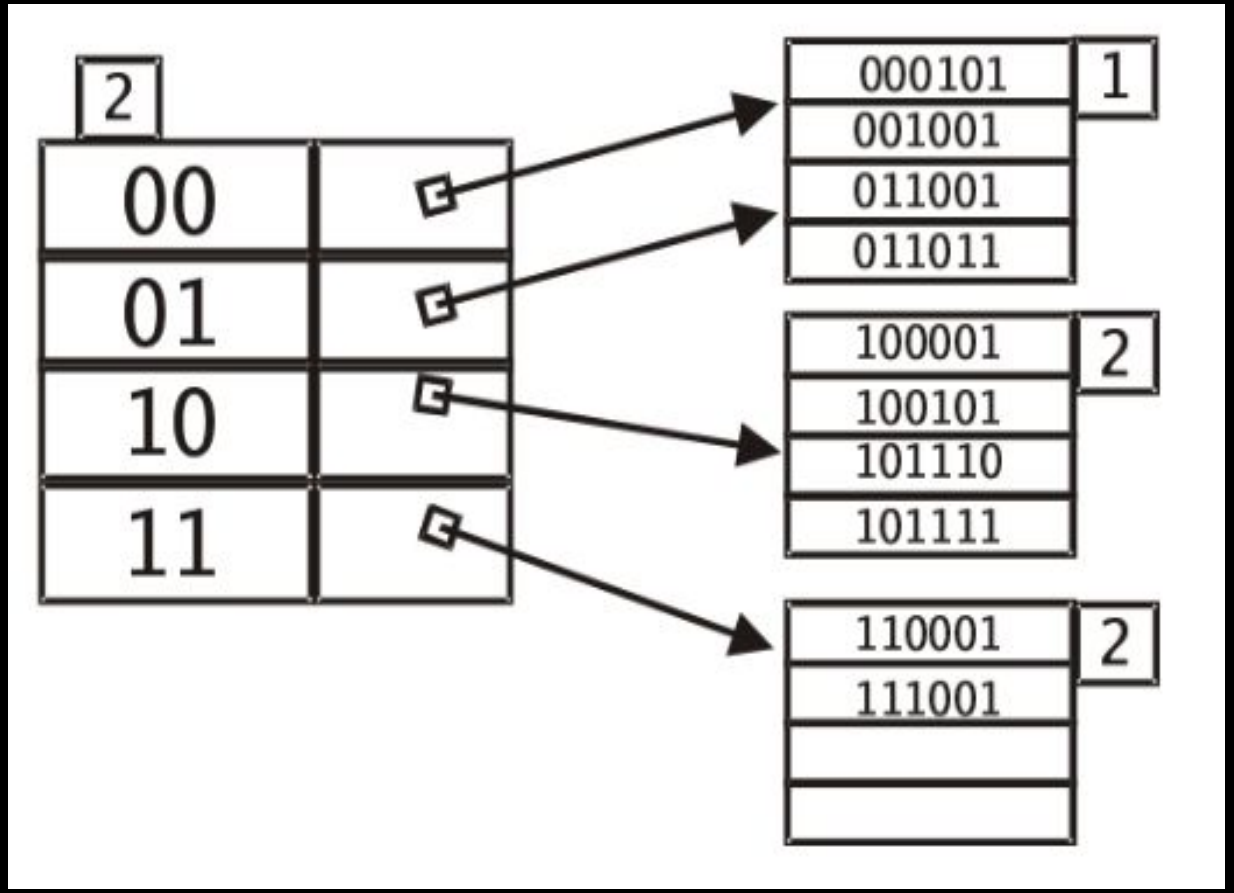
- **i: profundidade global = 2**
- **b = 6**
- **A tabela de buckets tem $2^i = 4$ entradas**
- **r: o número máximo de chaves/registros permitidos por bucket = 4**



Hashing extensível

Busca: Os i primeiros bits de h(k) são usados para acessar diretamente essa posição da tabela de buckets, a qual dá acesso ao bucket, logo uma busca sequencial nesse bucket é feita.

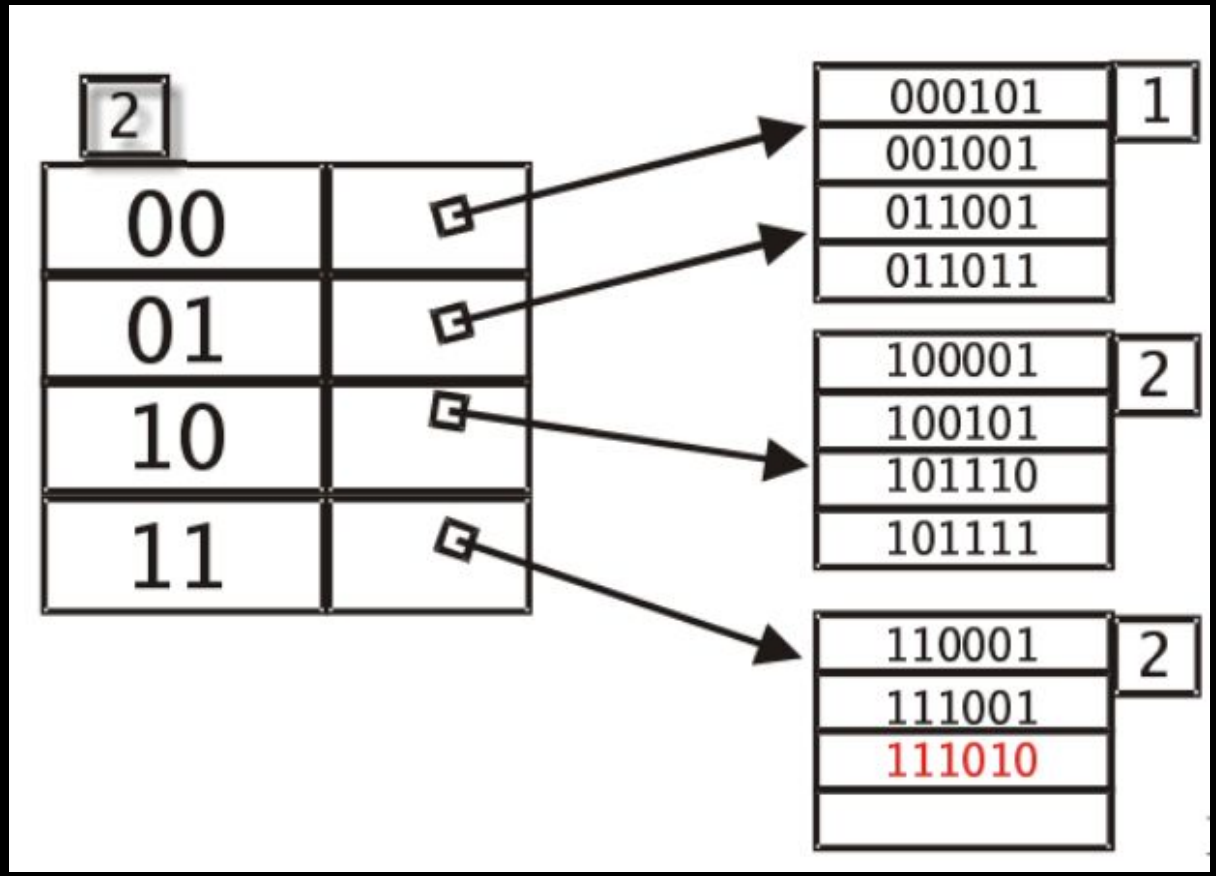
Ex: Seja h(k)=101110, acessamos a posição da tabela que tem o valor 10 e depois procuramos sequencialmente no bucket.



A busca requer 1 acesso a disco

Hashing extensível

Inserção: Os i primeiros bits de h(k) são usados para acessar diretamente essa posição da tabela de buckets, a qual dá acesso ao bucket.



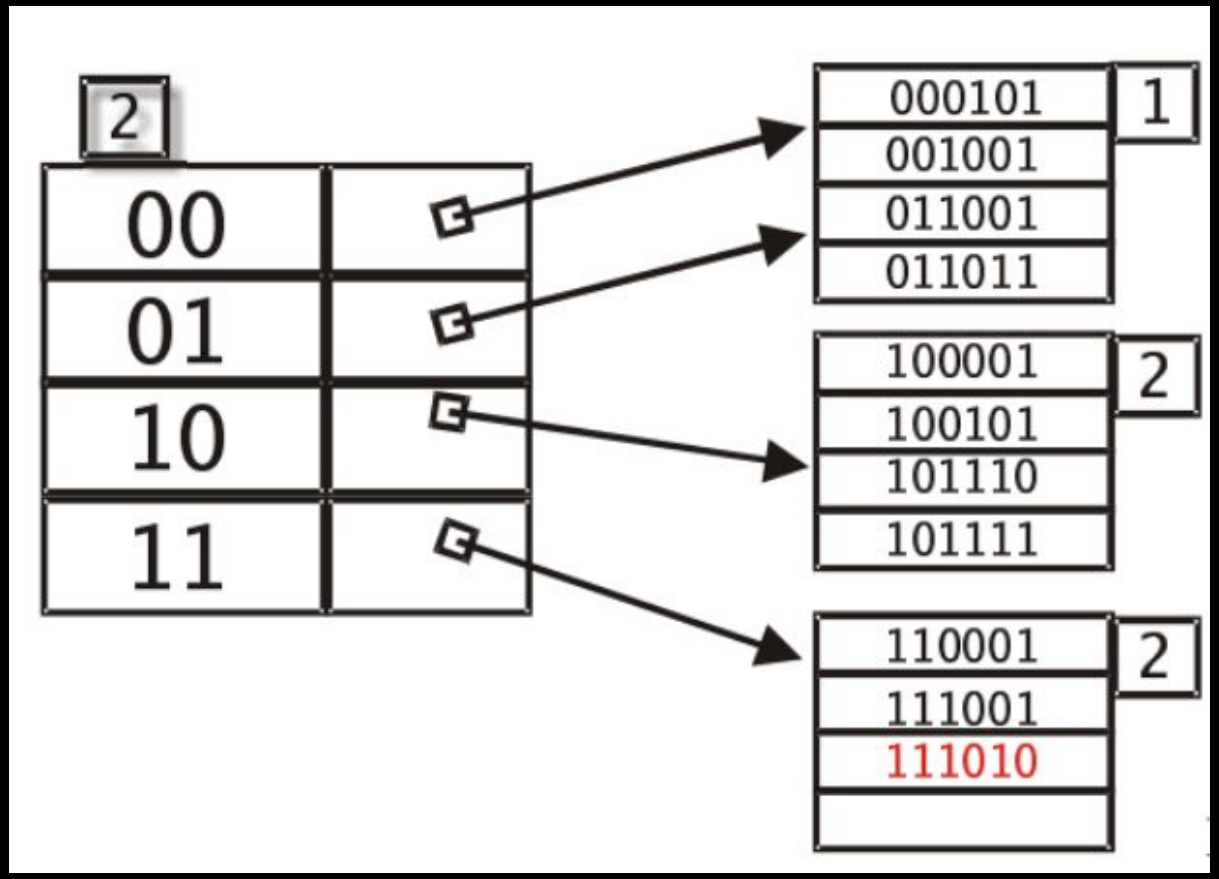
Caso 1: Há espaço no bucket

Simplesmente inserimos
Ex: inserção de 111010

Hashing extensível

Inserção:

Caso 2: O bucket está completo. A profundidade local do bucket onde será feita a inserção é menor do que a profundidade global, j<i.



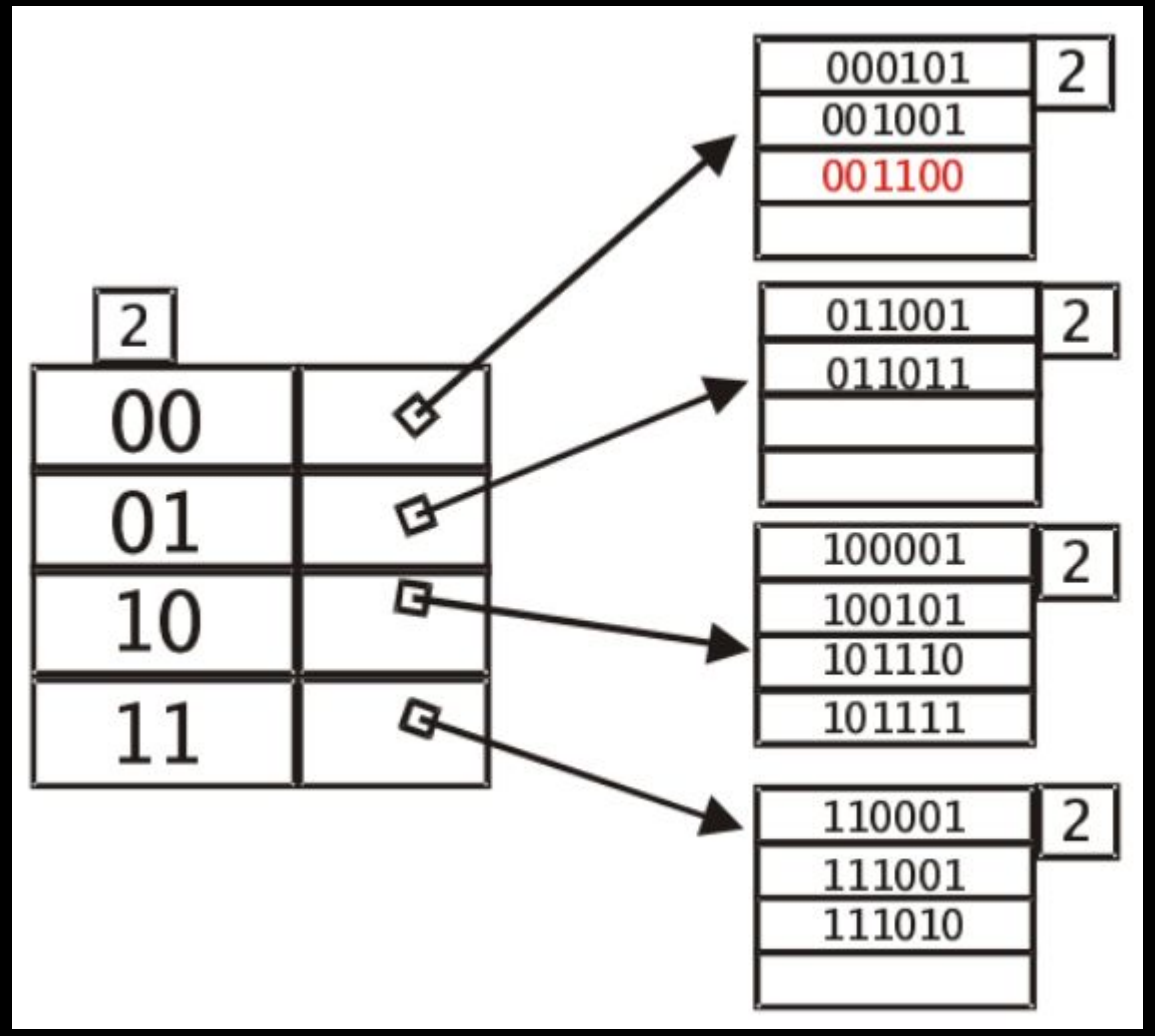
Criam-se buckets com uma profundidade um bit maior e dividem-se as chaves entre eles. Consertam-se os ponteiros. Ex: inserir 001100

Hashing extensível

Inserção:

Caso 2: O bucket está completo. A profundidade local do bucket onde será feita a inserção é menor do que a profundidade global, j<i.

Criam-se buckets com uma profundidade um bit maior e dividem-se as chaves entre eles. Consertam-se os ponteiros. Ex: inserir 001100

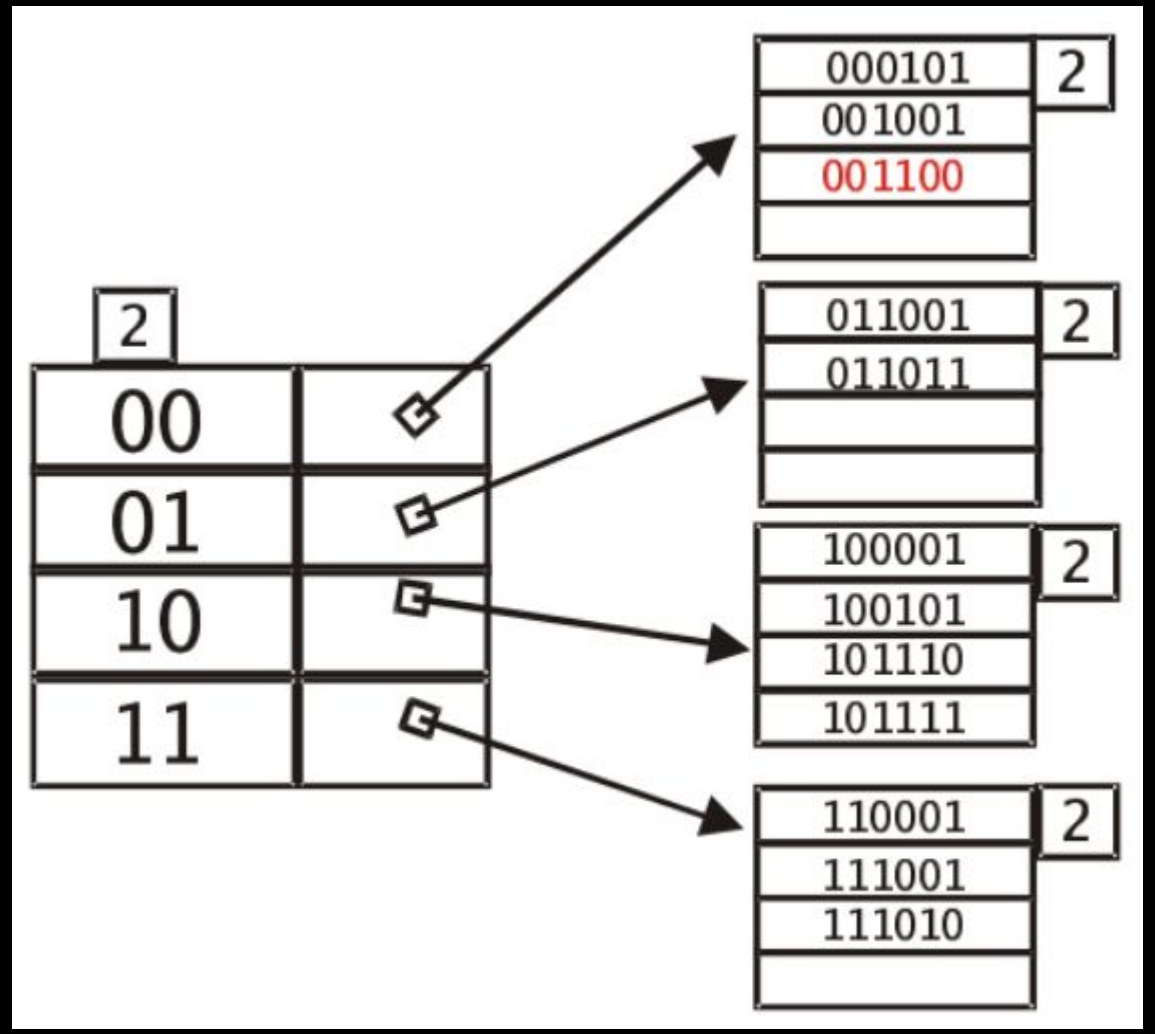


Hashing extensível

Inserção:

Caso 3: O bucket está completo. A profundidade local do bucket onde será feita a inserção é igual à profundidade global, i=j.

A tabela de buckets dobra de tamanho, ou seja a profundidade global é incrementada em 1. O bucket é dividido e dividem-se as chaves entre eles. Ex: inserir 100010

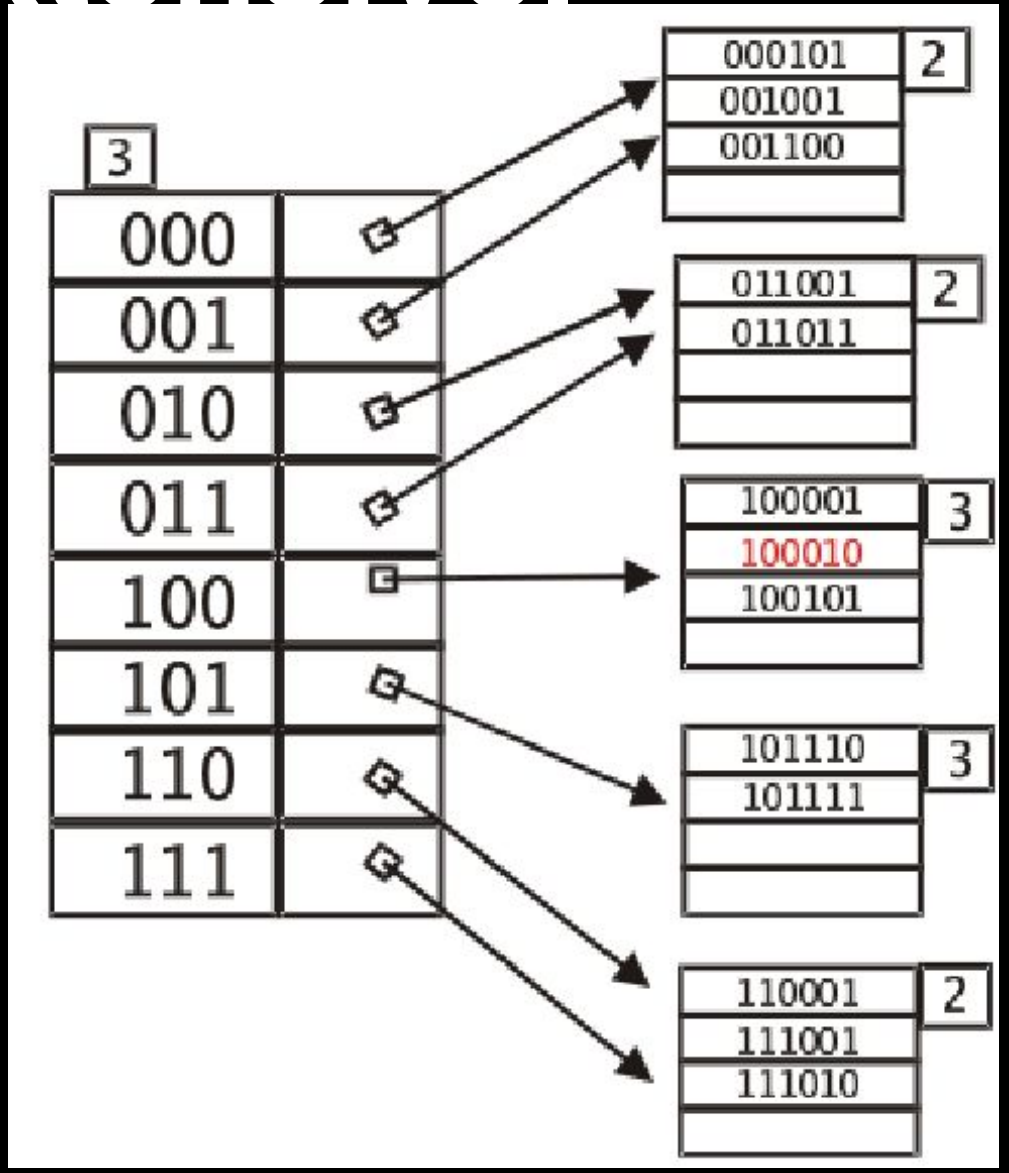


Hashing extensível

Inserção:

Caso 3: O bucket está completo. A profundidade local do bucket onde será feita a inserção é igual à profundidade global, i=j.

A tabela de buckets dobra de tamanho, ou seja a profundidade global é incrementada em 1. O bucket é dividido e dividem-se as chaves entre eles. Ex: inserir 100010



Hashing extensível

Inserção:

Supondo que a tabela de buckets está em memória principal. Além de fazer o READ-DISK

Caso 1: requer 1 operação WRITE-DISK
Caso 2: requer 2 operações WRITE-DISK
Caso 3: requer 2 operações WRITE-DISK

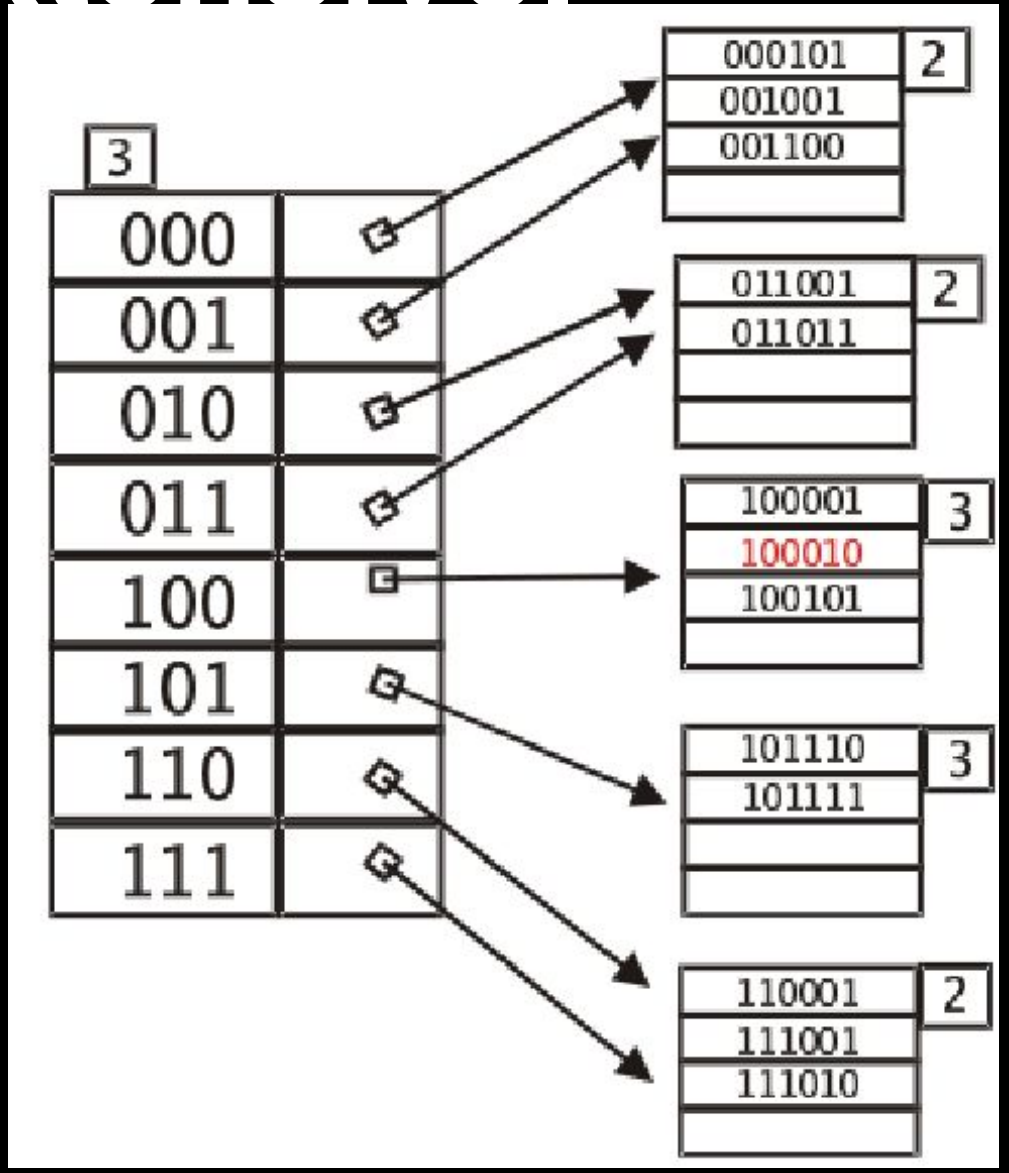
Assim, a inserção requer 2 WRITE-DISK no pior dos casos.

Hashing extensível

Remoção:

Ao remover uma chave, verificar:

- **É possível fundir o bucket com um bucket amigo?**
- **O tamanho da tabela de buckets pode ser reduzido?**



Hashing extensível

Remoção: fundindo buckets

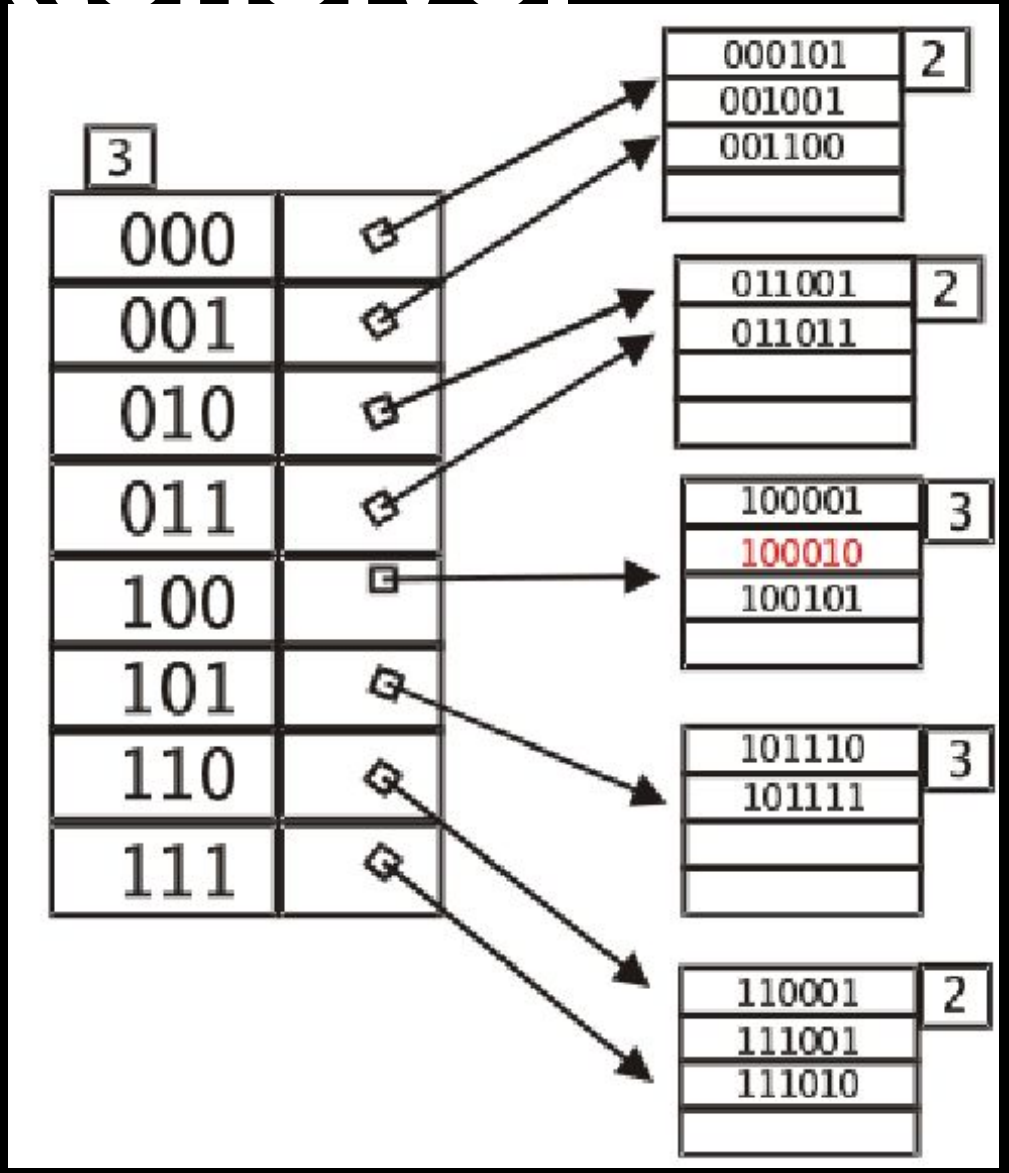
Bucket amigo:

- A profundidade do bucket tem que ser igual a profundidade global
- O bucket amigo é aquele que difere apenas no último bit do índice

Opções de quando fundir:

- Tentar sempre fundir os buckets
- Definir um número mínimo de elementos em cada bucket

Exemplo: remover 100010 e 100101



Hashing extensível

Remoção: fundindo buckets

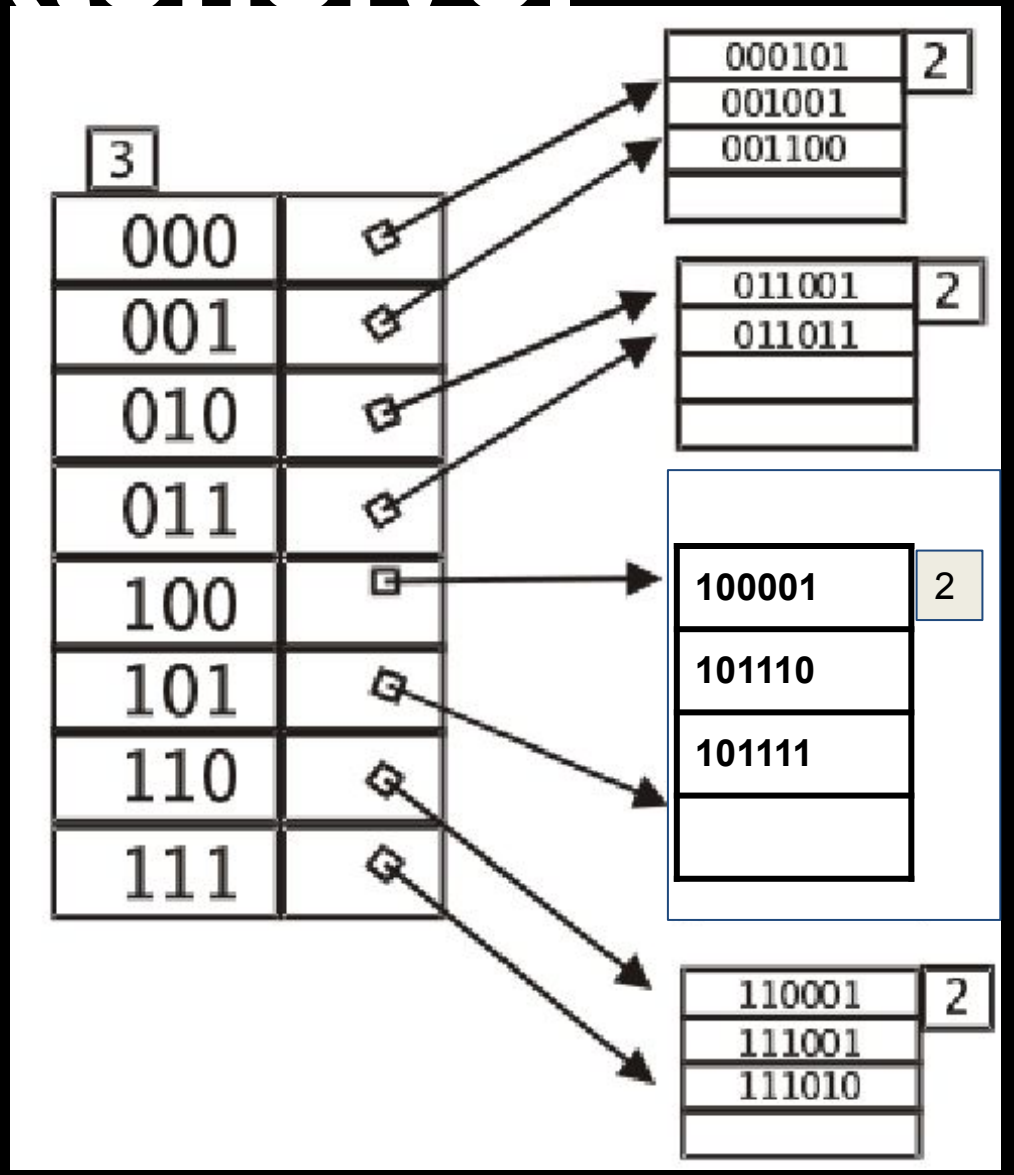
Bucket amigo:

- A profundidade do bucket tem que ser igual a profundidade global
- O bucket amigo é aquele que difere apenas no último bit do índice

Opções de quando fundir:

- Tentar sempre fundir os buckets
- Definir um número mínimo de elementos em cada bucket

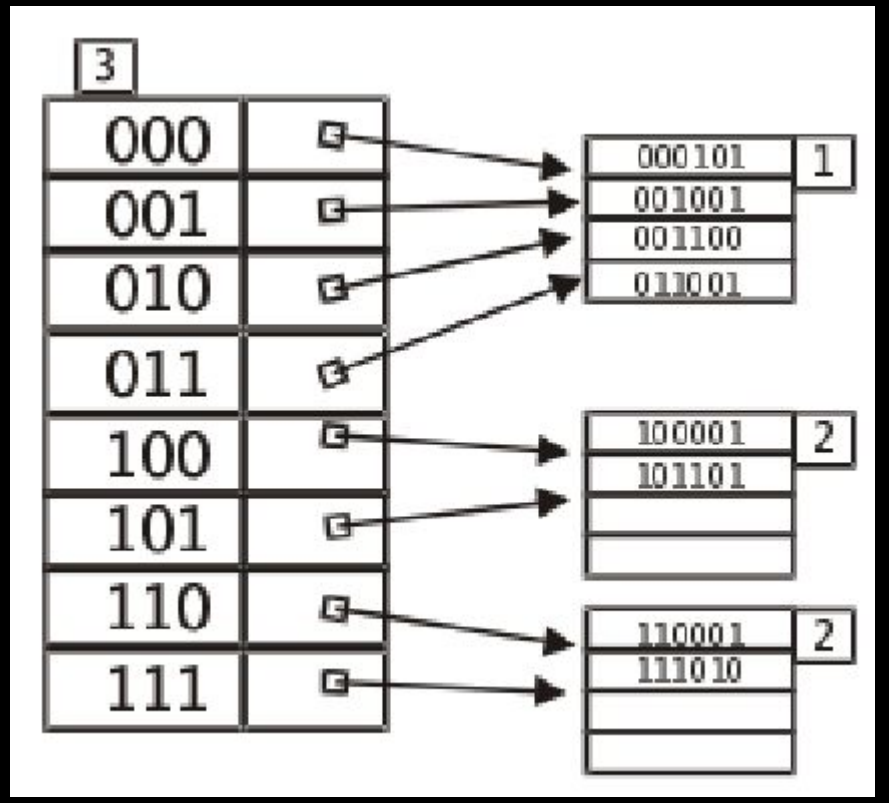
Exemplo: remover 100010 e 100101



Hashing extensível

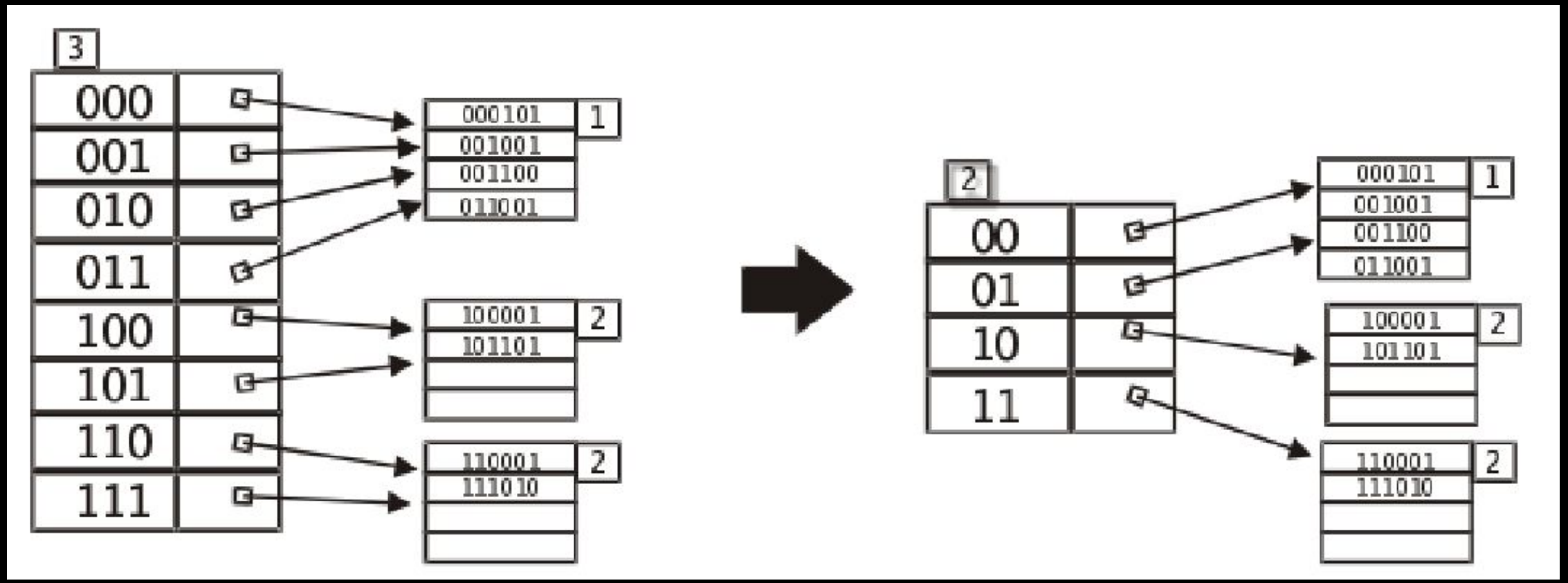
Remoção: reduzindo o tamanho da tabela de buckets

- Depois de fundir o bucket, verificar se é possível reduzir o tamanho da tabela de buckets.
- Se a profundidade local de todos os buckets é menor que a profundidade global, pode-se reduzir o tamanho da tabela de buckets.



Hashing extensível

Remoção: reduzindo o tamanho da tabela de buckets



Hashing linear

slides baseados nas videoaulas da profa. Ariane Machado Lima
https://eaulas.usp.br/portal/video?idItem=28810

Hashing linear

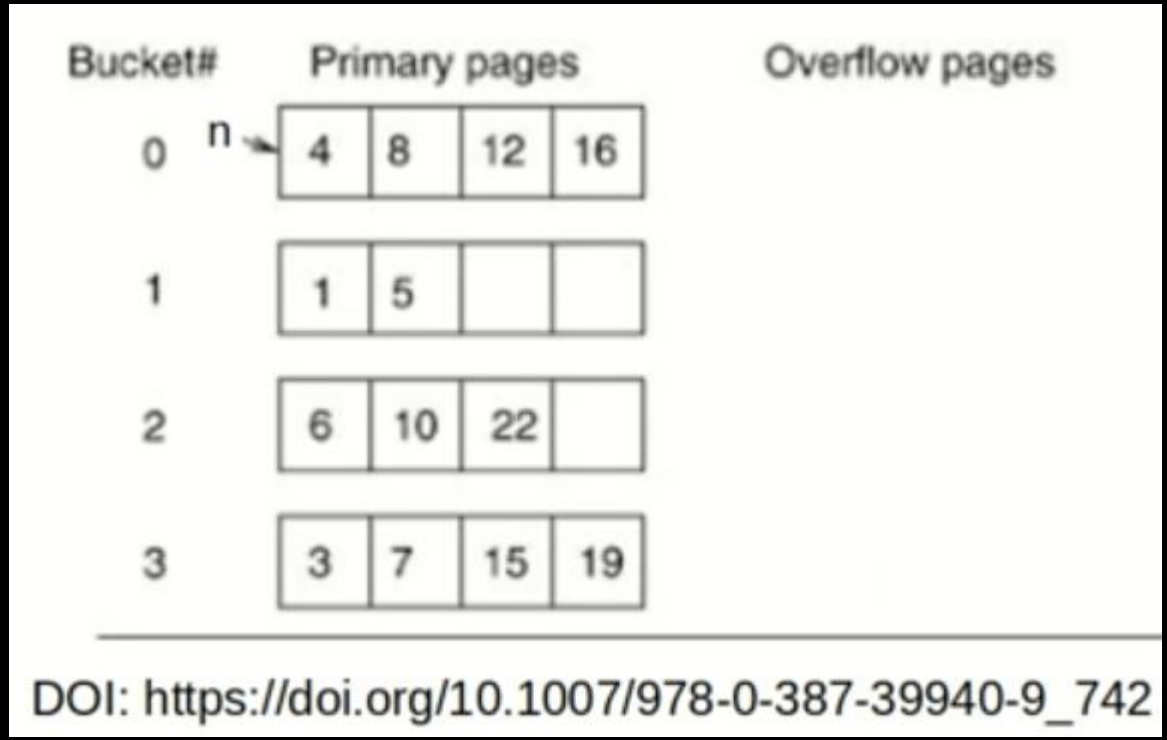
- É um tipo de hashing dinâmico
- Tem m buckets e função hash

  $h1 : U \rightarrow \{0, 1 \ldots m - 1\}$
- Colisões que causarem overflow vão para uma lista ligada de registros em buckets de overflow
- Diferente do hashing estático, usa um contador n de overflows (colisões em buckets lotados)
- À medida que overflows ocorrem (em quaisquer bucket), vou dividindo em dois os buckets 0,1,2,... linearmente.

Hashing linear

m=4, n=0 h1(k)=k mod 4
inserir 11
Primeiro overflow em qualquer bucket:

- Aloca bucket m
- Divide registros do bucket entre os buckets 0 e m com
  h2(k)=k mod 2M
- n=1

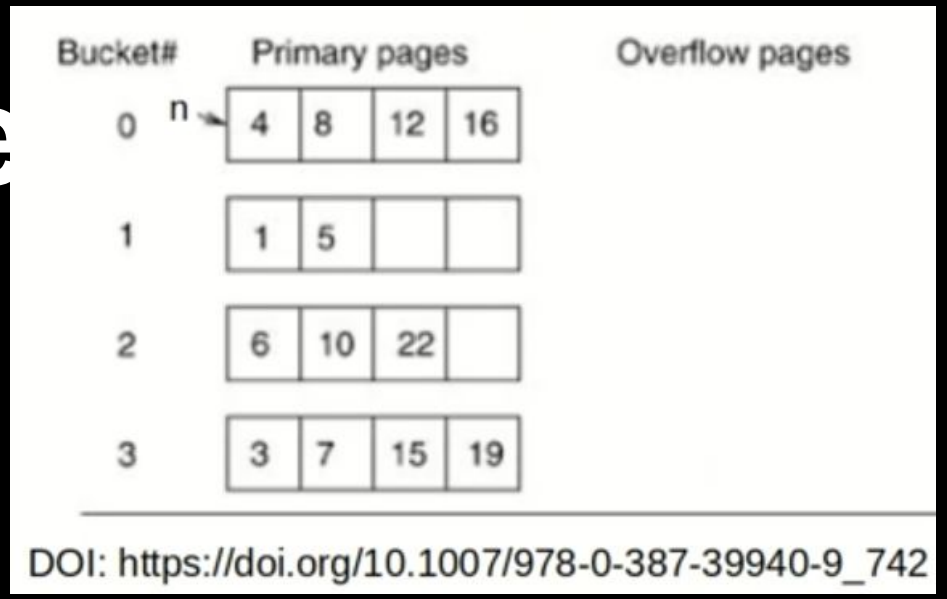


DOI: https://doi.org/10.1007/978-0-387-39940-9_742

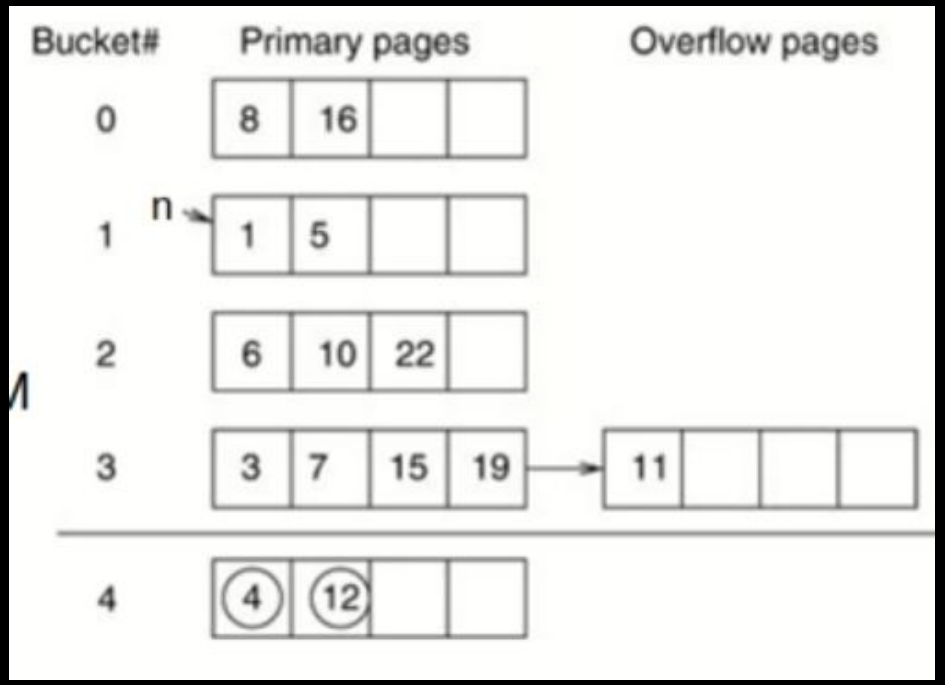
Hashing line

m=4, n=0 h1(k)=k mod 4
inserir 11
Primeiro overflow em qualquer bucket:

- Aloca bucket m
- Divide registros do bucket entre os buckets 0 e m com
  h2(k)=k mod 2m
- n=1



DOI: https://doi.org/10.1007/978-0-387-39940-9_742

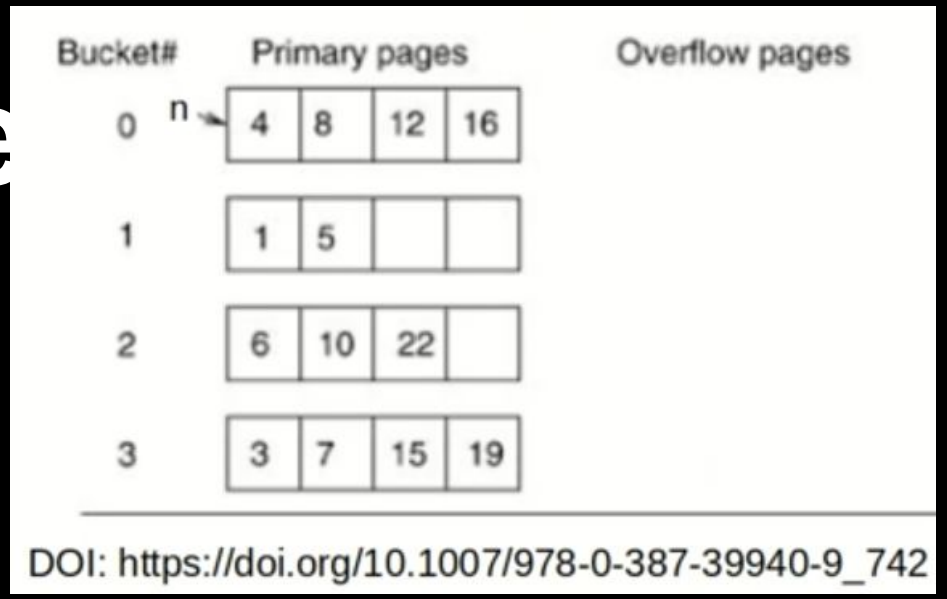


Hashing line

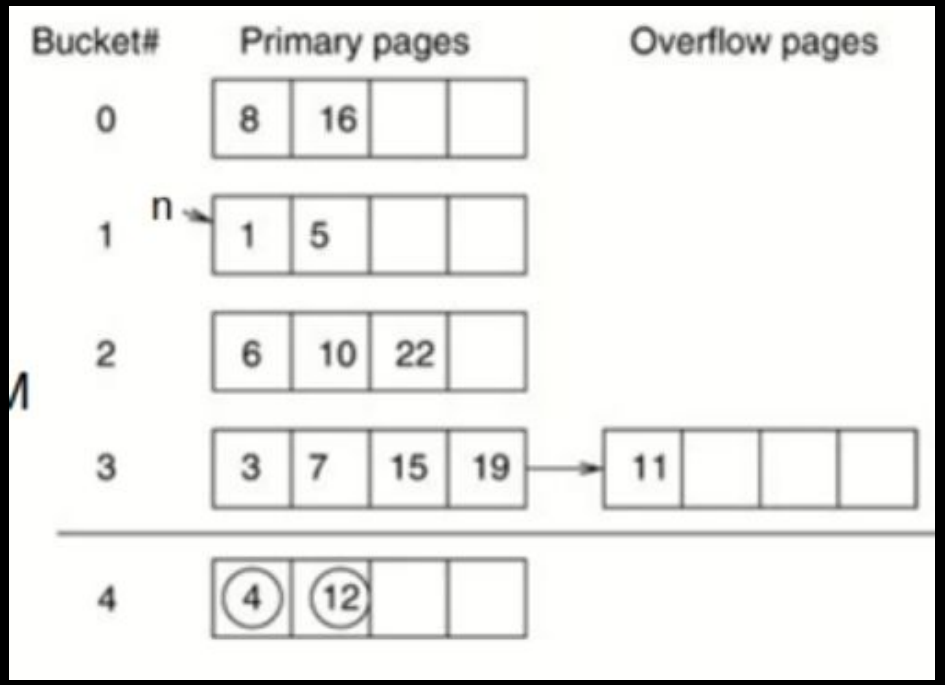
m=4, n=0 h1(k)=k mod 4
inserir 11
Primeiro overflow em
qua
- A
- [
  b
  (
  h2(k)=k mod 2m
- n=1

h1(k)=k mod 4

h2(k)=k mod 8 (vale para os buckets 0 e 4)



DOI: https://doi.org/10.1007/978-0-387-39940-9_742



Hashing line

m=4, n=0 **h1(k)=k mod 4**

ins

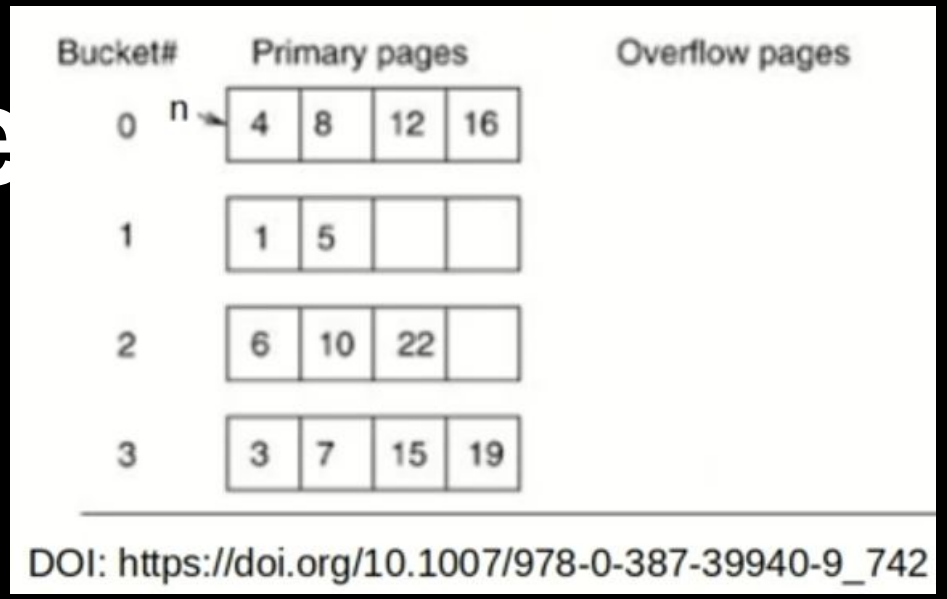
Pri

qua

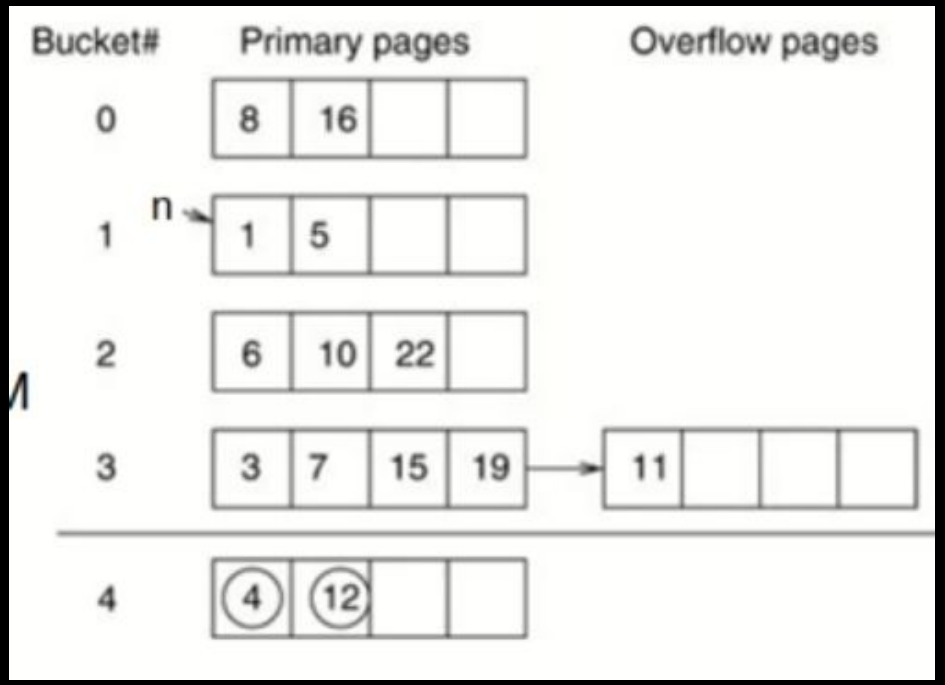
- A
- D
  k
  0
- n=1

O contador serve para saber qual função usar na busca da chave k:

h1(k)<n? Se sim, use h2(k), senão use h1(k) para saber em qual bucket está a chave ou deveria estar



DOI: https://doi.org/10.1007/978-0-387-39940-9_742

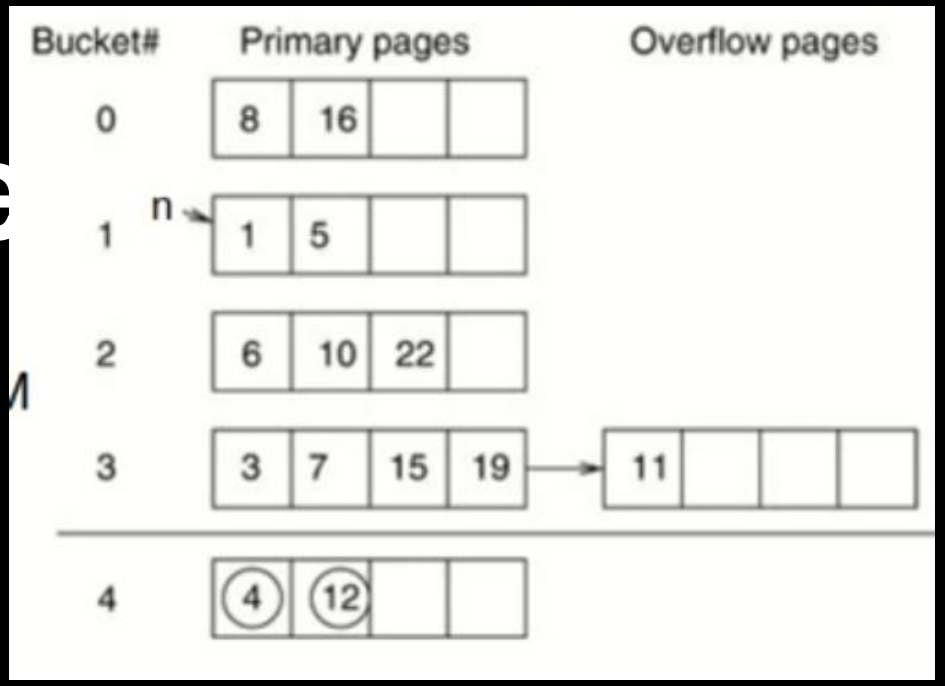


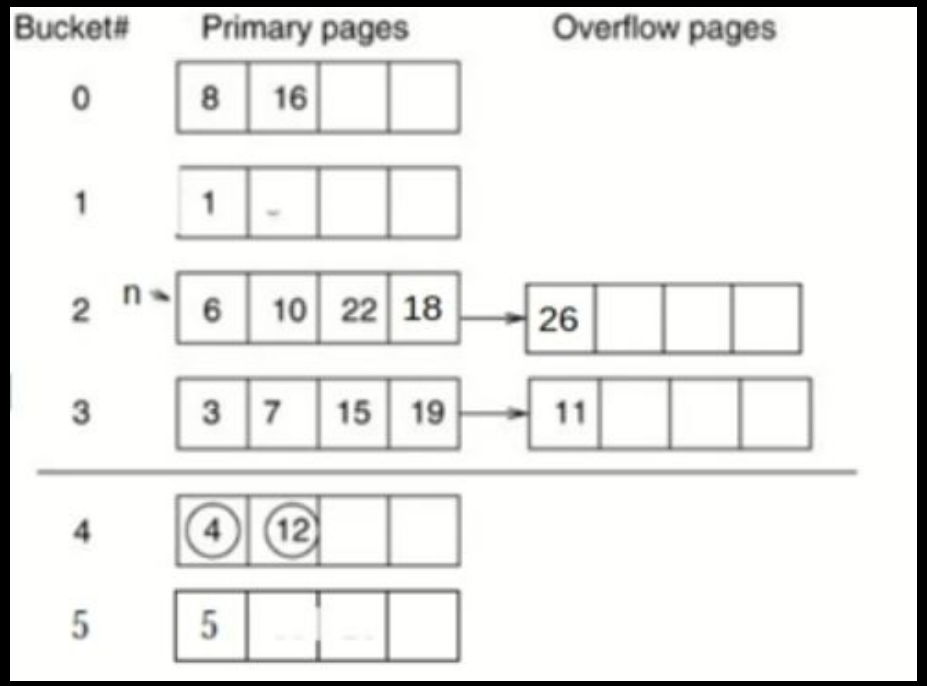
Hashing line

m=4, n=1 h1(k)=k mod 4
h2(k)=k mod 8

inserir 18, 26
Segundo overflow em qualquer bucket:

- Aloca bucket m+1
- Divide registros do bucket entre os buckets 1 e m+1 com h2(k)=k mod 2m
- n=2



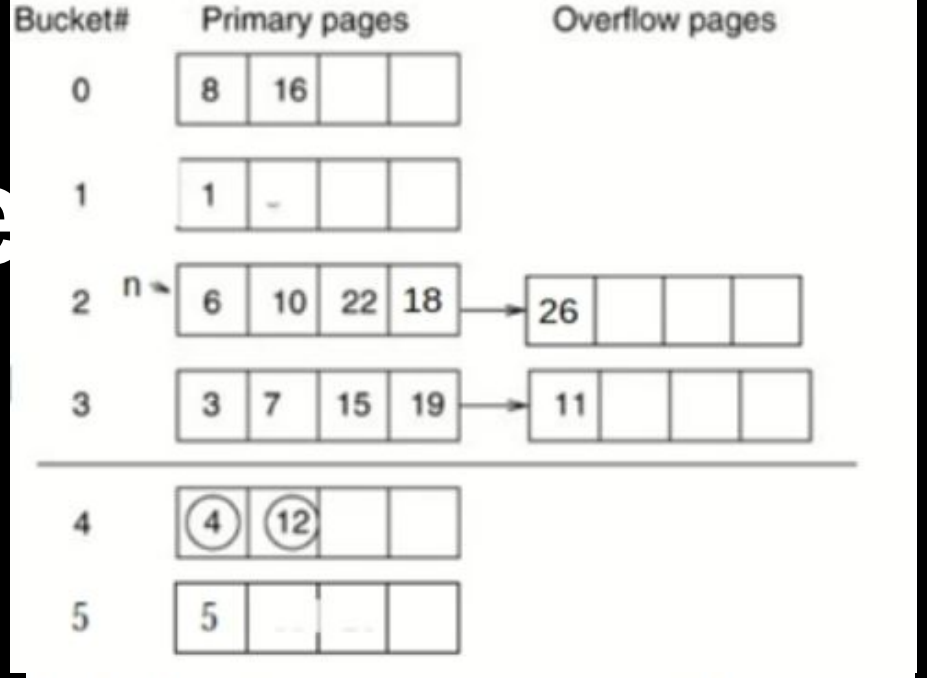


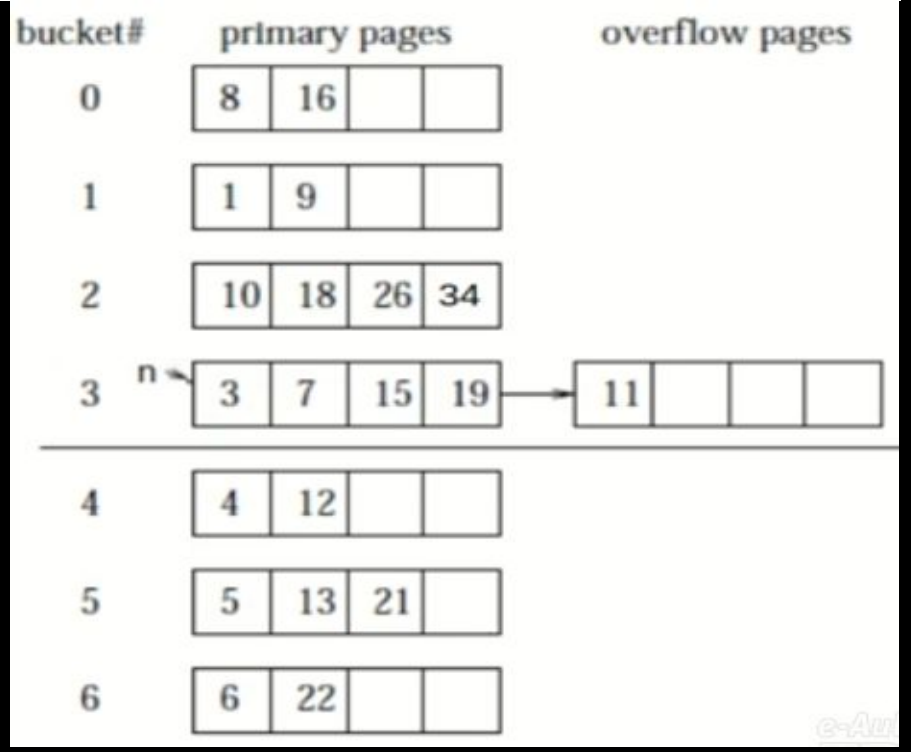
Hashing line

m=4, n=2 h1(k)=k mod 4
h2(k)=k mod 8

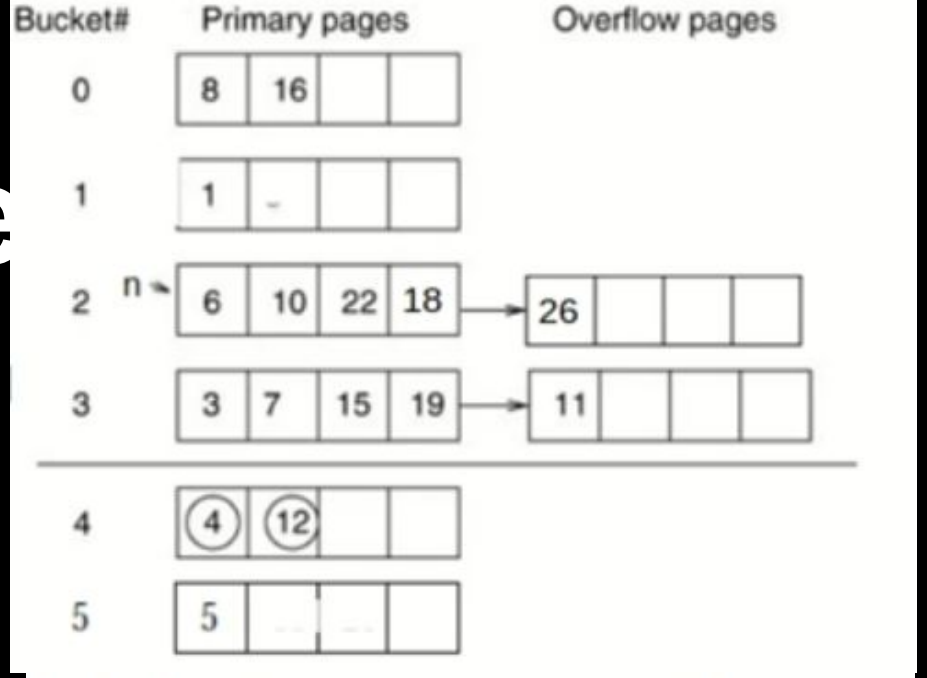
inserir 9,13,21,34
Terceiro overflow em qualquer bucket:

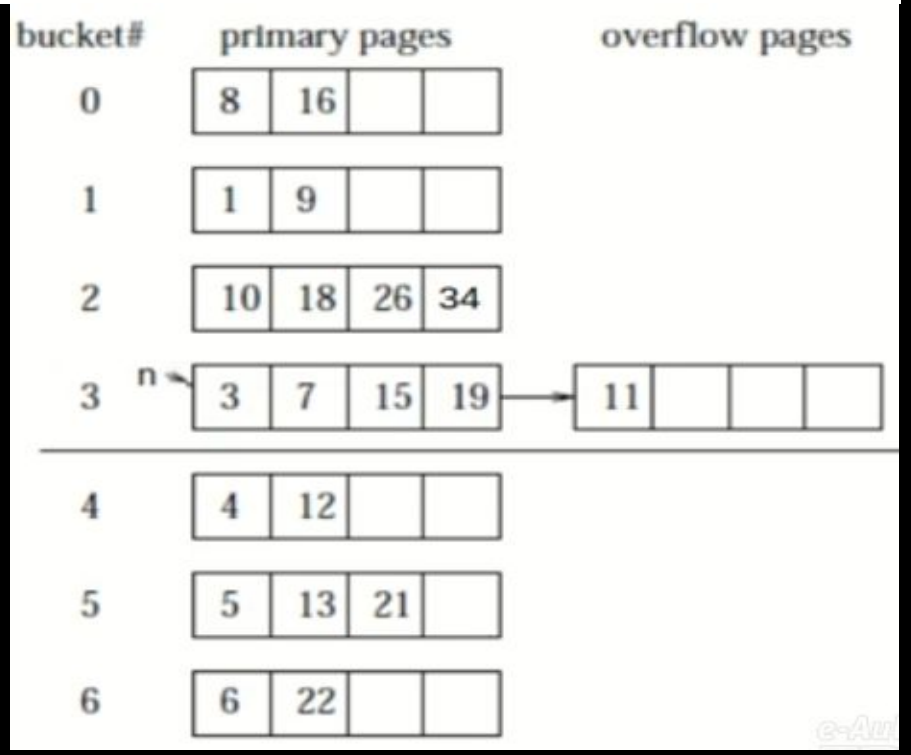
- Aloca bucket m+2
- Divide registros do bucket entre os buckets 2 e m+2 com h2(k)=k mod 2m
- n=3





Hashing line





n-ésimo overflow em qualquer bucket:

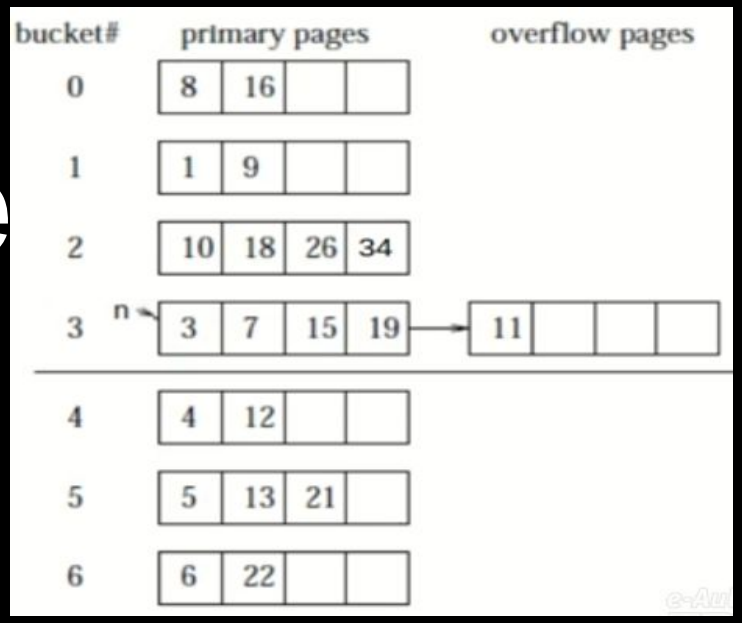
- Aloca bucket m+n-1
- Divide registros do bucket entre os buckets n-1 e m+n-1 com h2(k)=k mod 2m
- n++

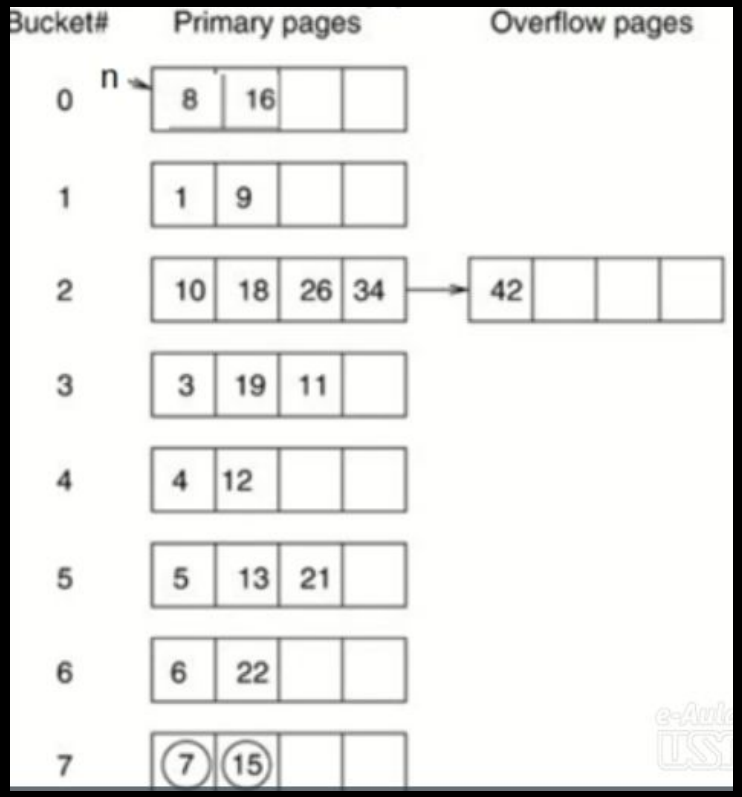
Hashing line

m=4, n=3 h1(k)=k mod 4
h2(k)=k mod 8

inserir 42
Quarto overflow em qualquer bucket:

- Aloca bucket m+3
- Divide registros do bucket entre os buckets 3 e m+3 com h2(k)=k mod 2m
- n=4

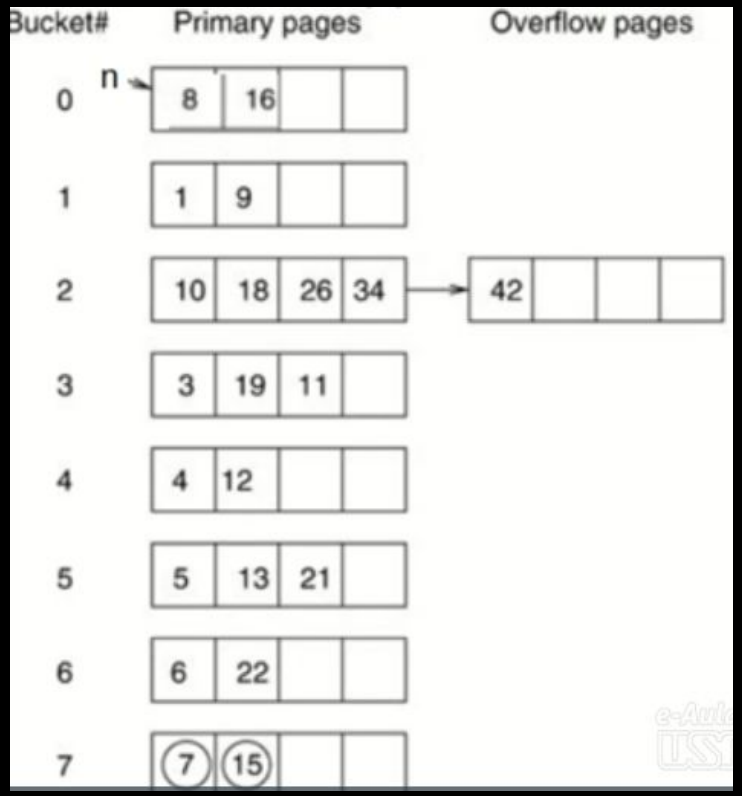




Hashing linear

Processo continua até que n=m:

- Todos os m buckets foram divididos
- Tabela de hash tem tamanho 2m
- h1 não é mais necessária
- n=0 e recomeça uma nova etapa de divisões.
- As funções de hash ativas:
  h2(k)=k mod 2m
  h3(k)=k mod 4m
- Processo continua até n=2m



Hashing linear

- Ao invés de dividir a cada colisão, dividir de acordo com o fator de carga:

$$\alpha = n/(m*r)$$

n = número de chaves/registros

m = número de slots (buckets)

r = número de registros que cabem em um bucket

Definir um intervalo aceitável de fator de carga

- Se α está acima do limite superior, divide buckets
- Se α está abaixo do limite inferior, fusiona buckets

Hashing dinâmico

extensível

- Usa uma tabela de ponteiros a buckets
- Usa uma função hash que transforma as chaves em números binários de tamanho fixo

linear

- Não usa tabela de ponteiros a buckets
- Existem 2 funções de hash ativas em cada passo

Hashing x árvores B

Hashing x árvores B

hashing

- Busca apenas por valores específicos (=)
  Ex: buscar a chave 543

árvores B e variações

- Buscar um conjunto de valores de acordo com uma relação entre os elementos (<, <=, >, >=)
  Buscar todas as chaves < 543