

[MAC0426] Sistemas de Bancos de Dados
[IBI5013] Bancos de Dados para Bioinformática

Aula 21

Índices

Acelerando o acesso
aos dados
(Parte 2)

Profa. Kelly Rosa Braghetto

12/06/2017

Estruturas de Dados Usadas como Índices

- ◆ Várias tipos de estruturas de dados podem ser usadas como índices
- ◆ Veremos neste curso:
 - ▶ Índices simples, sobre arquivos de dados ordenados (índices primários)
 - ▶ Índices secundários, sobre arquivos de dados não ordenados
 - ▶ Árvores B+
 - ▶ **Tabelas *Hash***

Tabelas *Hash*

Recordando *hashs* em memória principal:

- ◆ Função *hash* h que pega uma chave de busca como parâmetro e computa a partir dela um inteiro entre 0 e $B-1$, onde B é o número de *buckets*
- ◆ Um vetor de *buckets*, indexado de 0 a $B-1$, armazena a “cabeça” de B listas ligadas, uma para cada *bucket* do vetor
- ◆ Se um registro tem uma chave k , então ele é ligado à lista do *bucket* na posição $h(k)$
- ◆ Para uma boa introdução sobre o assunto:

<http://www.ime.usp.br/~pf/algoritmos/aulas/hash.html>

Tabelas *Hash*

- ◆ Uma tabela *hash* que armazena um grande número de registros precisa ser mantida em **memória secundária**
- ◆ Neste caso, o vetor de *buckets* consiste em um **vetor de blocos**
- ◆ A função de *hash* ***h*** determina o *bucket* (= bloco) onde um registro será armazenado
- ◆ Quando um *bucket* estoura (ou seja, não tem mais espaço para armazenar todos os registros direcionados para ele), uma cadeia de blocos de *overflow* pode ser adicionada ao *bucket*, para armazenar mais registros

Uma Tabela Hash

Chave do registro armazenado na posição

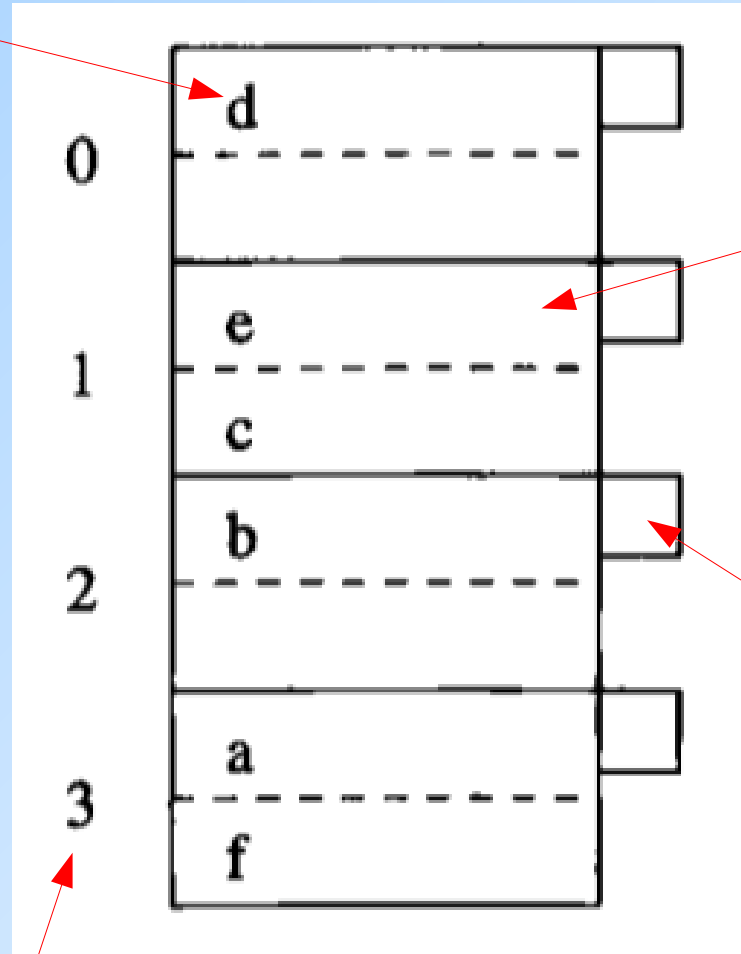
$$h(d) = 0$$

$$h(c) = h(e) = 1$$

$$h(b) = 2$$

$$h(a) = h(f) = 3$$

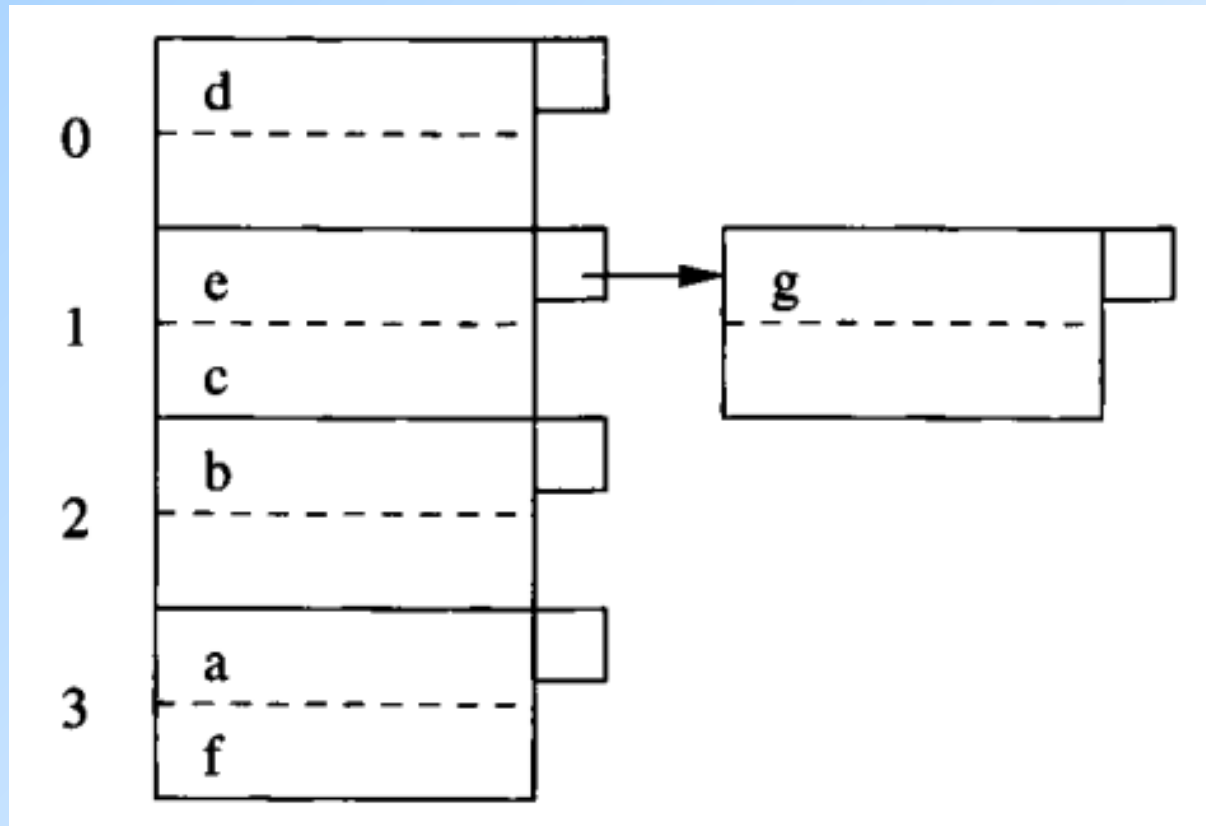
Número do bucket



Neste exemplo, cada bloco tem espaço para 2 registros

Nó que pode ser usado para encadear blocos de *overflow*

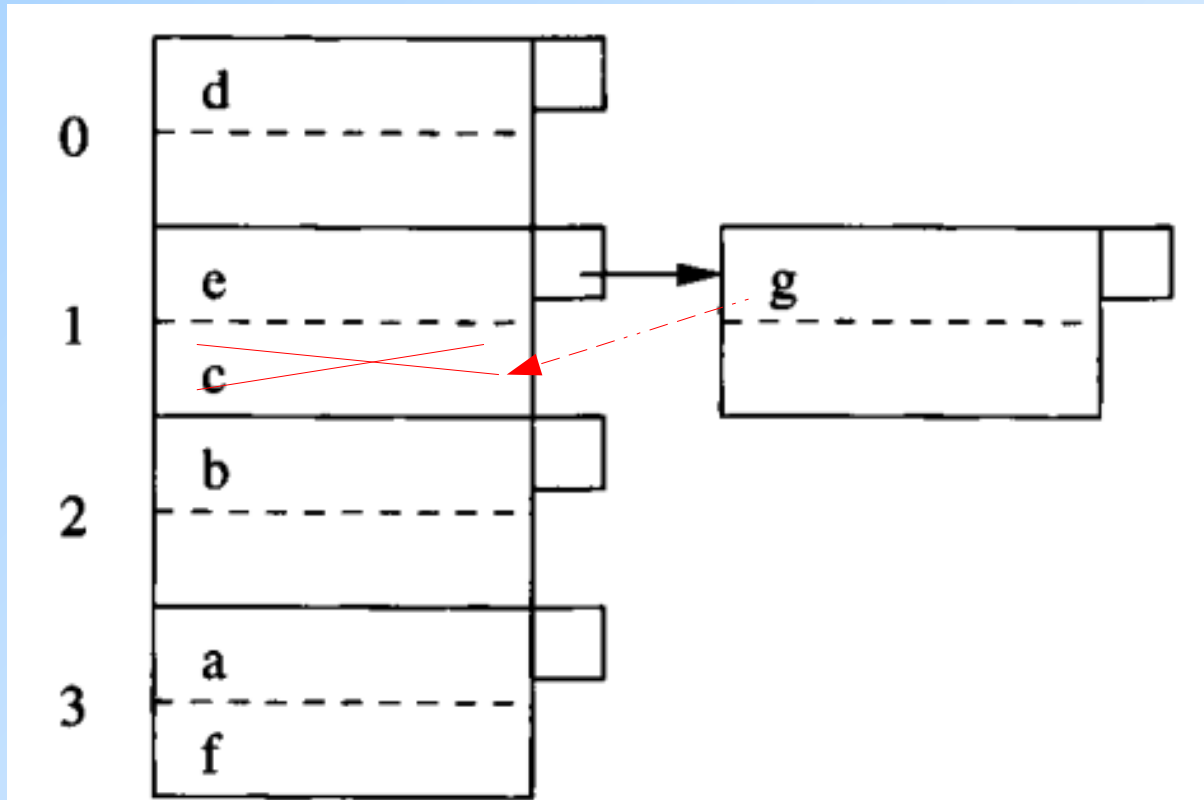
Inserção em uma Tabela Hash



Exemplo: inserção de um novo registro com chave g , sendo que $h(g) = 1$.

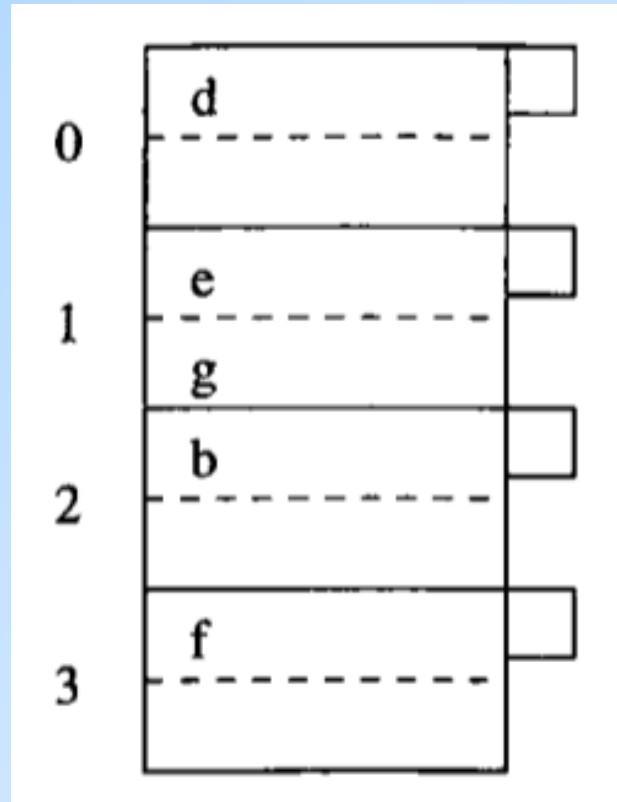
Como o bloco do *bucket* 1 já está cheio, é preciso incluir um novo bloco de dados para receber g e ligá-lo ao bloco original do *bucket* 1.

Remoção em uma Tabela Hash



Para remover o registro com chave **c**, nós vamos ao *bucket* de número $h(c)$ (lembrando que $h(c) = 1$) e procuramos por registros que tenham a chave **c**. Qualquer registro com essa chave deve ser excluído. Após uma remoção, pode ser possível remanejar registros de um bloco para outro, a fim de reduzir o número de blocos de um *bucket* (como é o caso nesse exemplo).

Remoção em uma Tabela Hash



Resultado após a remoção do registro com chave **c** e do remanejamento do registro com chave **g**. Também foi removido da tabela hash o registro de chave **a** ($h(a) = 3$).

Obs: nem sempre vale a pena remanejar registros com a intenção de reduzir o número de blocos depois de uma remoção. Isso pode causar um efeito “sanfona”, onde cada remoção/inserção de registro gera uma remoção/inserção de bloco.

Um Parênteses sobre a Escolha da Função de *Hash*

- ◆ Uma função de *hash* deve espalhar bem uma chave pelo conjunto de índices de buckets $0..B-1$
 - ▶ É desejável que os *buckets* tenham +/- a mesma quantidade de registros
- ◆ Uma função de *hash* precisa ser rápida de se computar
 - ▶ Ela será computada muitas vezes!
- ◆ Escolha comum para chaves que são números inteiros:
 - ▶ **$K \bmod B$** (resto da divisão inteira de K por B)
 - ▶ Frequentemente escolhe-se como valor de B um número primo grande → reduz o número de colisões

Eficiência de Índices em Tabelas *Hash*

- ◆ Idealmente, há *buckets* o suficiente para que cada *bucket* caiba em apenas um bloco
 - ▶ Neste caso, uma busca por um registro requer apenas um acesso ao disco, enquanto que uma operação de inserção ou remoção dos dados requer 2 acessos
 - ▶ Esse desempenho é melhor que o das árvores B+ e dos outros tipos de índices
 - ▶ Problema: diferentemente das árvores B+, tabelas *hash* não ajudam nas buscas por faixas de valores de chave.

Eficiência de Índices em Tabelas *Hash*

- ◆ Em índices de arquivos de dados grandes, pode haver muitos blocos associados a um mesmo *bucket*
 - ▶ Nesse caso, as buscas podem requerer o percorrimento de longas listas de blocos (com um acesso a disco para cada bloco)
 - ▶ Por isso, é muito importante manter o número de blocos por *bucket* pequeno

Tabelas de Hash Estáticas

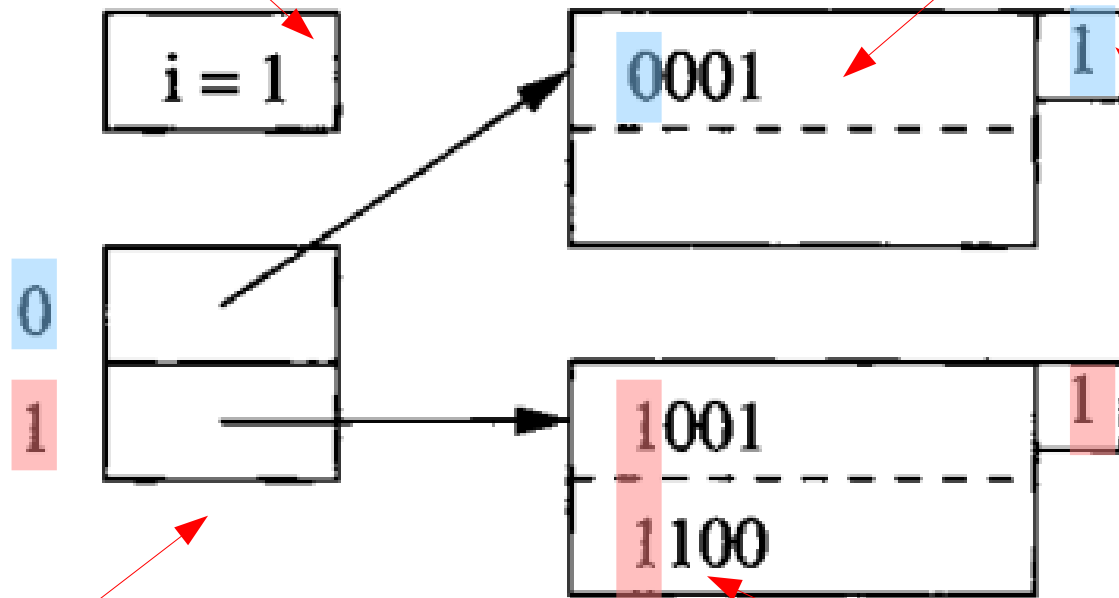
X

Tabelas de Hash Dinâmicas

- ◆ As tabelas de *hash* que vimos até agora são **estáticas** porque B , o número de *buckets*, nunca muda.
- ◆ Existem também tabelas **hash dinâmicas**, onde B varia de modo a ficar sempre próximo do número de registros dividido pelo número de registros que cabem em um bloco (~ 1 bloco por *bucket*)
- ◆ Veremos dois métodos para *hash* dinâmico:
 - ▶ **Hashing Extensível**
 - ▶ **Hashing Linear**

Tabela de *Hash* Extensível

$k = 4$, ou seja, a função de *hash* produz uma sequência de 4 bits. No momento, apenas 1 bit está sendo usado para definir a localização do registro, como indicado aqui.



j ; indica o número de bits usado para definir a pertinência no *bucket*

Vetor de *buckets* com 2^i posições (ponteiros para blocos)

Buckets

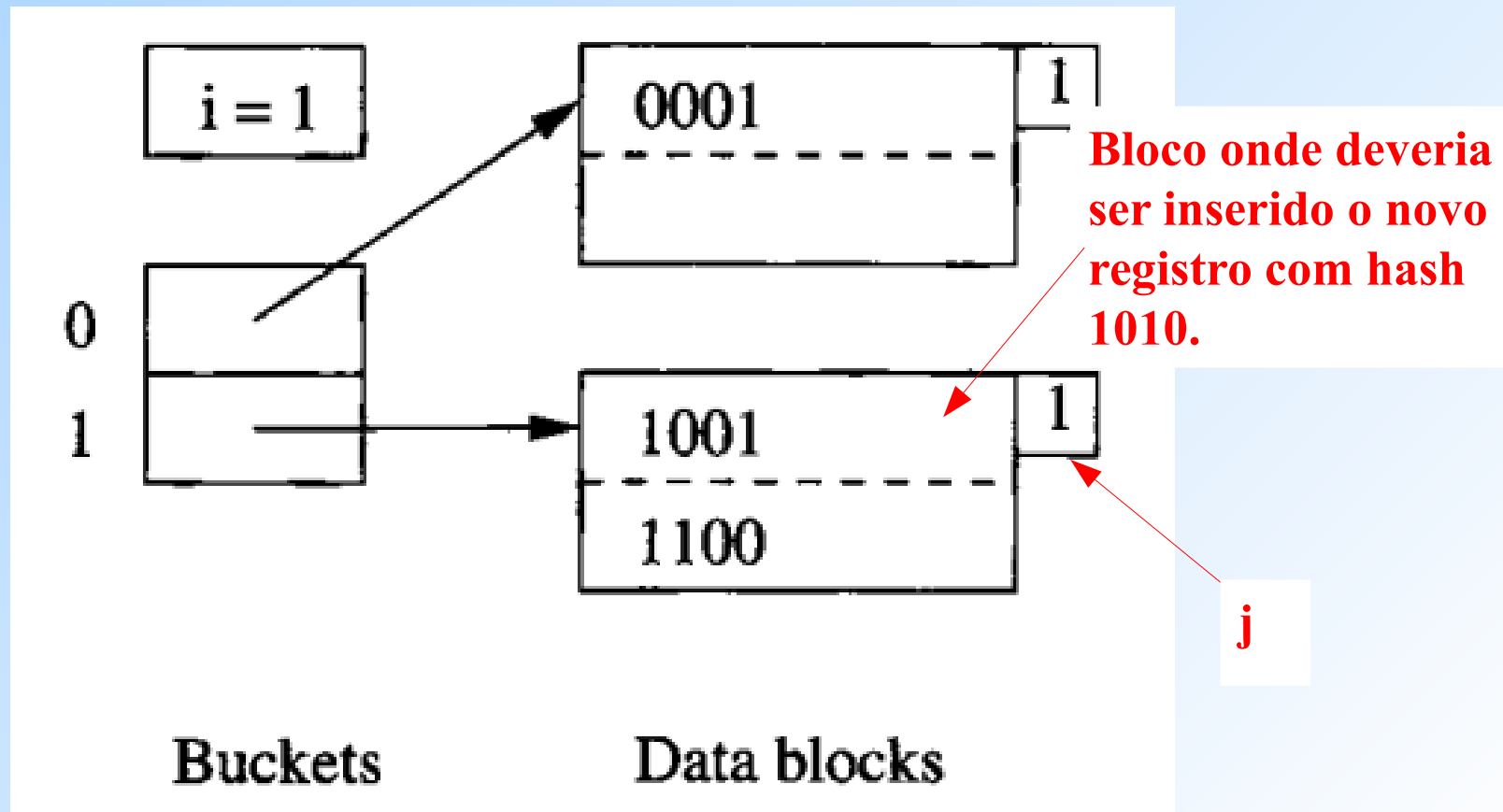
Data blocks

Armazena todos os registros com chave K em que $h(K)$ começa com o bit 1

Hash Extensível: Inserção

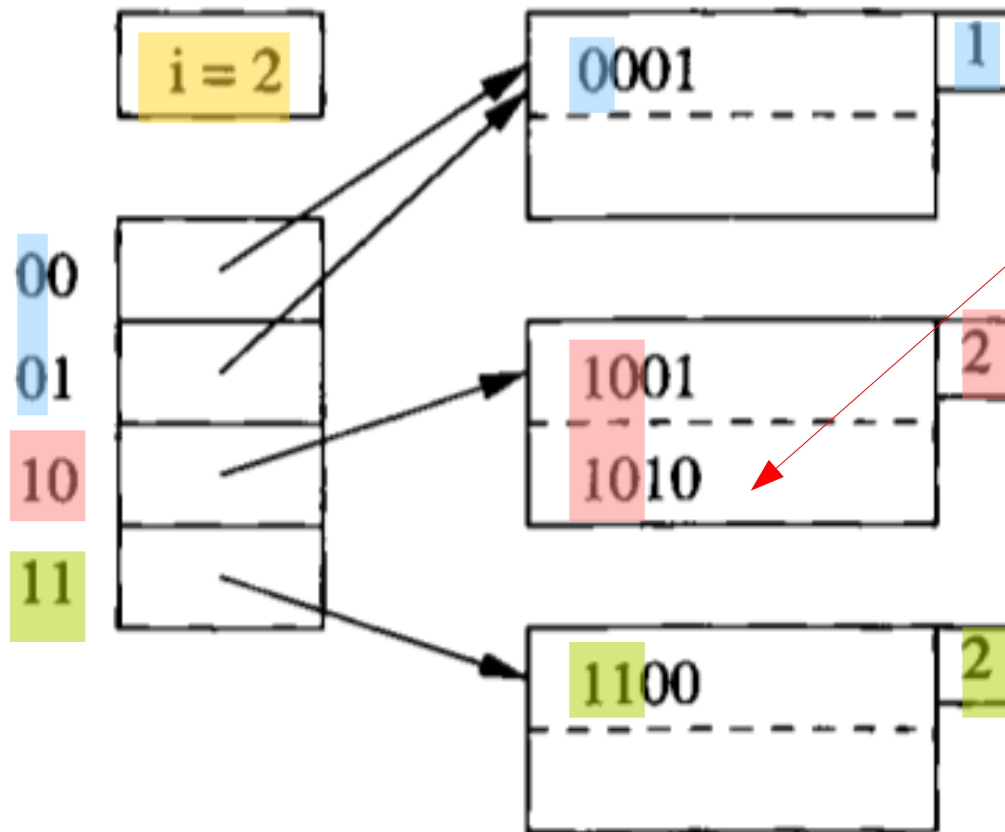
Exemplo: inserção de um registro com chave K tal que $h(K) = 1010$.

Como $i = 1$, o primeiro bit de $h(K)$ é 1, o novo registro pertence ao segundo bloco (*bucket* 1). Como o bloco já está cheio, ele precisa ser dividido em 2. Mas como $j = i = 1$, é preciso antes dobrar o tamanho do vetor de *buckets*.



Hash Extensível: Inserção

Exemplo (cont.): inserção de um registro com chave K tal que $h(K) = 1010$. O vetor de *buckets* foi dobrado e i foi atualizado para 2. O bloco 2 foi dividido em dois (e ambos tiveram o seu j atualizados para 2) e os registros foram redistribuídos

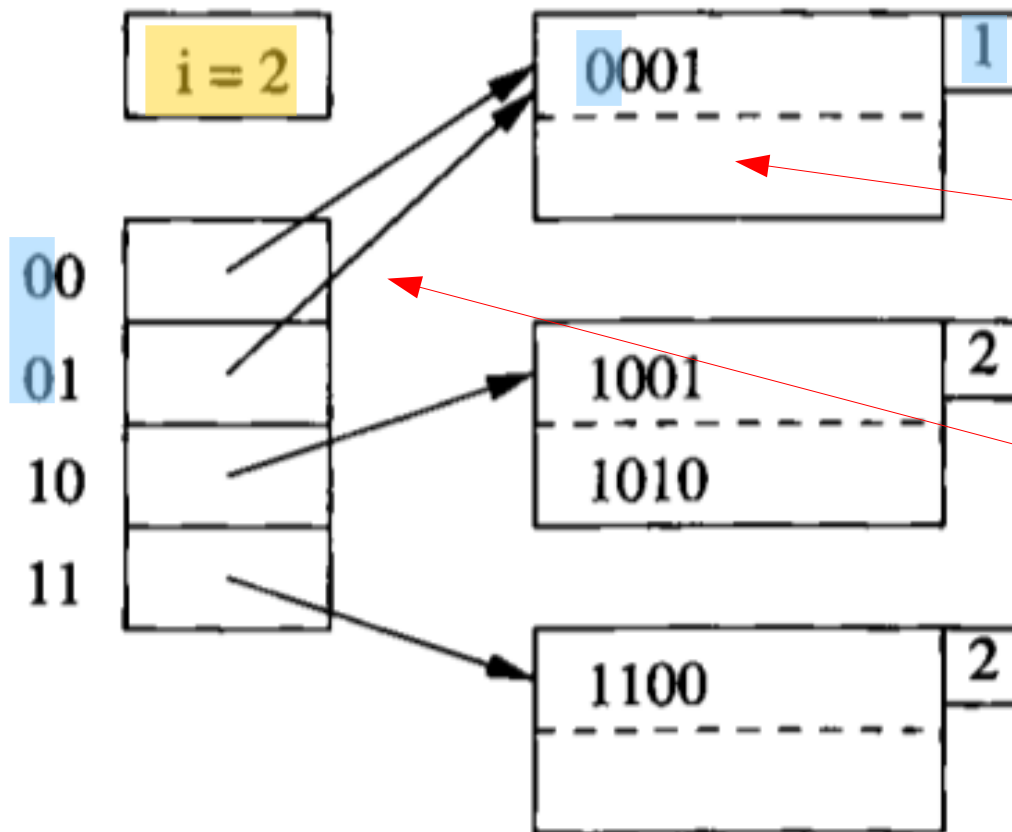


Novo registro inserido

Indicam que é preciso olhar para 2 bits de $h(K)$ para determinar se o registro pertence ao bloco

Hash Extensível: Inserção

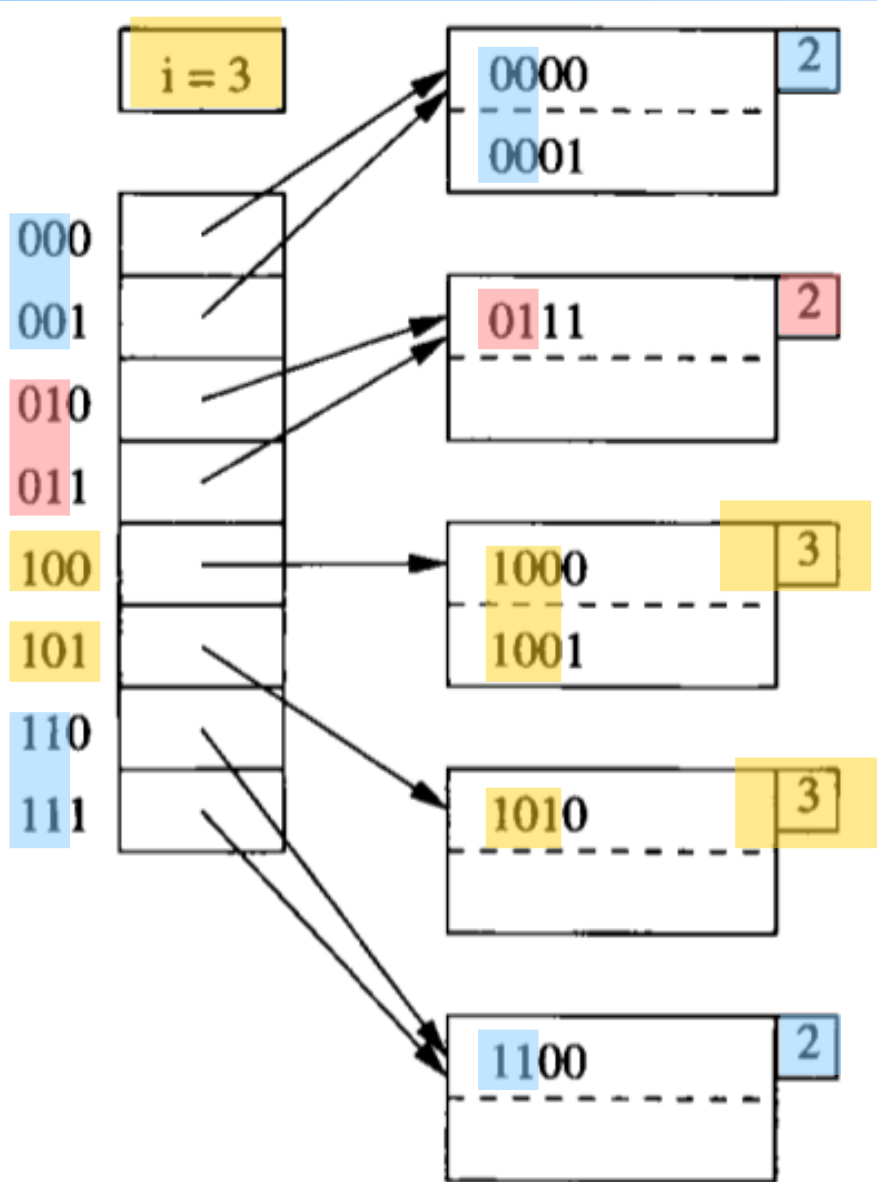
Exemplo 2: inserção de registros com chaves K1 e K2 tais que $h(K1) = 0000$ and $h(K2) = 0111$. Ambos os registros deveriam entrar no primeiro bloco da figura abaixo, o que causaria um estouro. Como o bloco usa apenas um 1 bit para determinar a pertinência no bloco (ou seja, $j = 1$) mas $i = 2$, então não é preciso mexer no tamanho do vetor de *buckets*.



Bucket que terá que ser dividido para acomodar os dois novos registros

Depois da divisão, cada um desses dois buckets passará a apontar para um bloco diferente.

Hash Extensível: Inserção



Exemplo 3: inserção de um registro com chave K tal que $h(K) = 1000$. Tal registro deveria entrar no bloco para 10 do slide anterior, que já estava cheio. Como esse bloco tem $j = 2$ e como $i = 2$, é preciso duplicar novamente o vetor de *buckets* para que a inserção possa ser feita. A figura ao lado mostra o resultado. Observe que i foi atualizado para 3, bem como o j dos blocos gerados pela divisão do bloco 10.

Tabelas de *Hash* Extensíveis

- ◆ **Vantagens:** na busca por um registro, nunca é necessário fazer mais do que um acesso ao disco
 - ▶ Geralmente, o vetor de *buckets* é pequeno o suficiente para ser mantido em memória principal, por isso ele não requer nenhum acesso a disco adicional
- ◆ **Desvantagens:**
 - ▶ Dobrar o vetor de *buckets* consome tempo
 - ▶ Dobrar o vetor de *buckets* pode fazer com que ele não caiba mais na memória principal meio repentinamente, ou causar um gasto de memória repentino que pode prejudicar o desempenho de outros programas

Tabelas de *Hash* Lineares

- ◆ Nessa técnica, o número n de *buckets* cresce mais lentamente
- ◆ O número n é sempre escolhido de forma que o número médio de registros por *bucket* seja uma fração fixa (por exemplo, 80%) do número de registros que cabem em um bloco
- ◆ Como os blocos não podem ser divididos a qualquer hora, blocos de *overflow* são permitidos
 - ▶ Mas o número médio de blocos de *overflow* por *bucket* será bem menor que 1
- ◆ O número de bits usado para numerar as entradas no vetor de *buckets* é $\lceil \log_2 n \rceil$, onde n é o número atual de *buckets*
 - ▶ Os bits usados são sempre os bits menos significativos no *hash* gerado

Tabelas de *Hash* Lineares

- ◆ Suponha que i bits estão sendo usados para numerar os *buckets* no vetor, e um registro com chave \mathbf{K} deve ser inserido no *bucket* $a_1 a_2 \dots a_i$, onde $a_1 a_2 \dots a_i$ são os i bits menos significativos de $\mathbf{h}(\mathbf{K})$.
- ◆ Seja $m = (a_1 a_2 \dots a_i)_{10}$
 - ◆ Se $m < n$, então o *bucket* de número m existe e o novo registro deve ir parar nele.
 - ◆ Se $n \leq m < 2^i$, então o *bucket* m ainda não existe e colocamos o registro no *bucket* $m - 2^{i-1}$, que é o *bucket* que obteríamos se mudássemos a_1 (que deve ser 1) para zero.

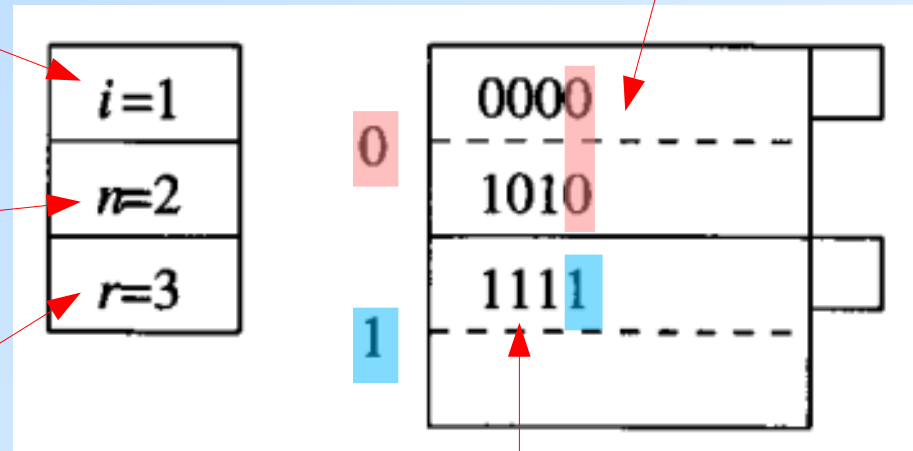
Tabela de *Hash* Linear

Indica o número de bits do *hash* que estão sendo usados atualmente para determinar o *bucket* de uma chave

n indica o número atual de *buckets* na tabela *hash*

r indica o número atual de registros na tabela *hash*

Registro com chave K tal que $h(K) = 0000$



Neste exemplo, supõe-se:

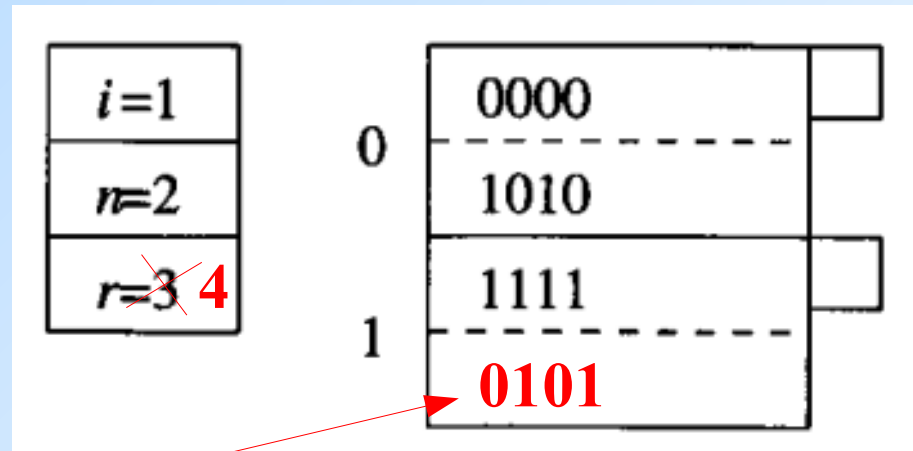
- que a função de *hash* h produz 4 bits
- $r \leq 1.7n$, ou seja, a ocupação média de um *bucket* não ultrapassa a 85% da capacidade de um bloco.

Inserção em uma Tabela de *Hash* Linear

**Exemplo 1: inserção de um registro com chave K tal que $h(K) = 0101$.
(Passo 1)**

Como $i = 1$, então usamos apenas o último dígito de $h(K)$ para determinar qual *bucket* deve conter o novo registro. Logo, o registro deve ficar no *bucket* de número 1.

Como ainda há uma posição vaga nesse *bucket*, basta inserir o registro com $h(K) = 0101$ nela.



Problema: Ao inserir o novo registro no *bucket* 1, ficamos com uma taxa de ocupação média (r/n) de 100%.

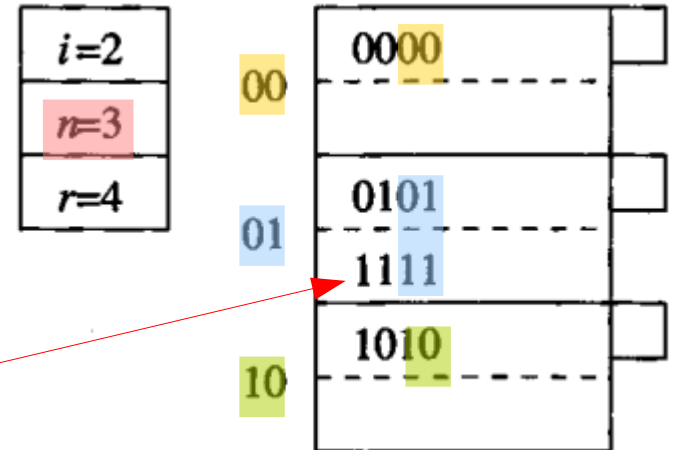
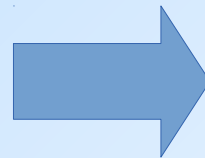
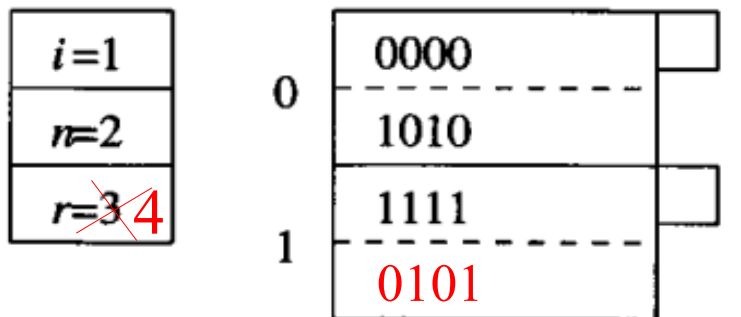
Solução: aumentar n para 3

Inserção em uma Tabela de *Hash* Linear

**Exemplo 1: inserção de um registro com chave K tal que $h(K) = 0101$.
(Passo 2 – aumentando n para 3)**

Como $\lceil \log_2 3 \rceil = 2$, i passará a 2 e então passaremos a numerar os *buckets* 0 e 1 por 00 e 01. E o novo *bucket* receberá o número 10.

Depois, redistribuímos os registros nos *buckets* de acordo com os 2 dígitos menos significativos de seus *hashs*.



Mas por quê o 1111 foi parar no bucket 01?

Busca em uma Tabela de *Hash* Linear

Exemplo 1 de busca: Busca por uma chave K tal que $h(K) = 1010$

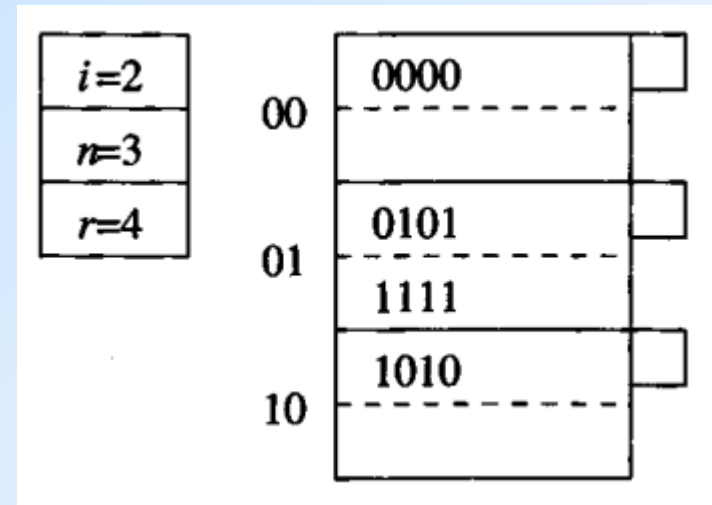
1) Como $i = 2$, devemos olhar para os dois últimos dígitos de $h(k)$, que são 10.

Como $m = (10)_{\text{dec}} = 2 < n = 3$, então existe um *bucket* com número 10 na tabela.

2) Para cada registro no *bucket* 10 que possua *hash* 1010 faça:

3) Se a chave do registro for igual à chave buscada K , então devolva o registro e encerre a busca

Importante: observe que encontrar um registro com *hash* 1010 na tabela não é garantia de que a chave K esteja na tabela! Diferentes chaves podem gerar um mesmo *hash*.



Busca em uma Tabela de *Hash* Linear

Exemplo 2 de busca: Busca por uma chave K tal que $h(K) = 1011$

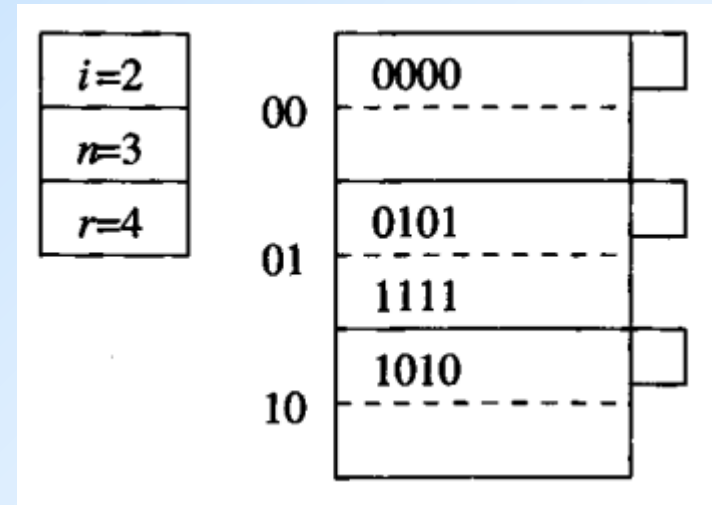
1) Como $i = 2$, devemos olhar para os dois últimos dígitos de $h(k)$, que são 11.

Como $m = (11)_{dec} = 3 \geq n$, então o *bucket* de número 11 não existe na tabela. Portanto, nós redirecionamos a busca para o *bucket* 01, trocando o primeiro dígito de 11 por 0.

2) Para cada registro no *bucket* 01 que possua *hash* 1011 faça:

3) Se a chave do registro for igual à chave buscada K , então devolva o registro e encerre a busca

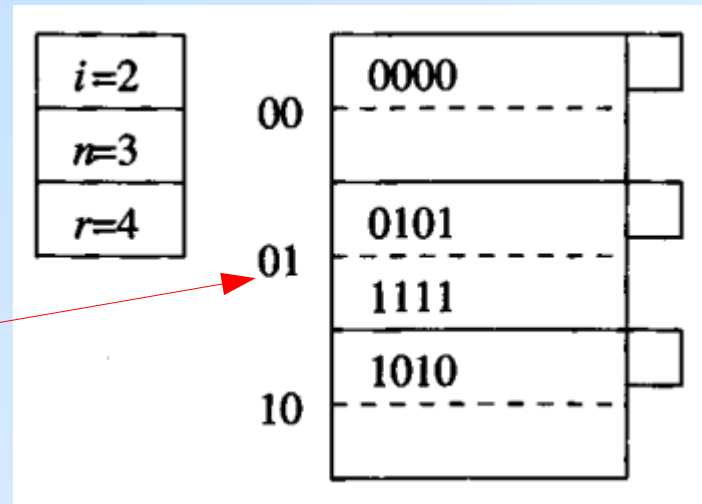
Obs.: O bucket 01 não possui nenhum registro com *hash* 1011, portanto o registro com chave K certamente não está na tabela.



Inserção em uma Tabela de *Hash* Linear

Exemplo 2: inserção de um registro com chave K tal que $h(K) = 0001$.

Como $i = 2$, então usamos os dois últimos dígitos de $h(K)$ para determinar qual *bucket* deve conter o novo registro. Logo, o novo registro deve ficar no *bucket* de número 01.

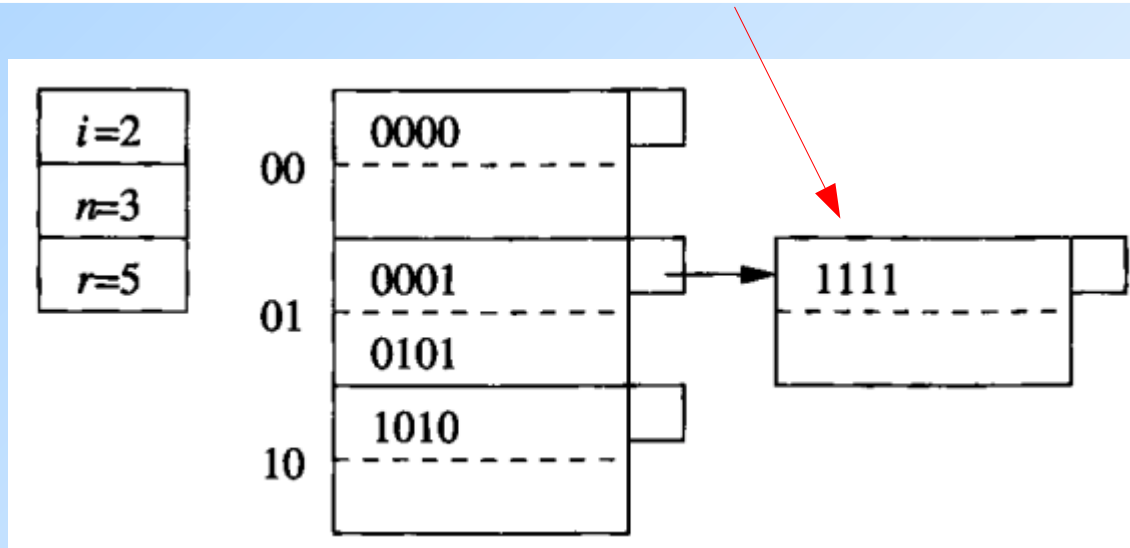


Problema: o *bucket* 01 já está cheio!

Solução: adicionar um bloco de *overflow* para o *bucket* 01 e distribuir os registros do *bucket*.

Inserção em uma Tabela de *Hash* Linear

Exemplo 2: inserção de um registro com chave K tal que $h(K) = 0001$.
(Passo 2 – após a inserção do bloco de *overflow* para o *bucket* 10)



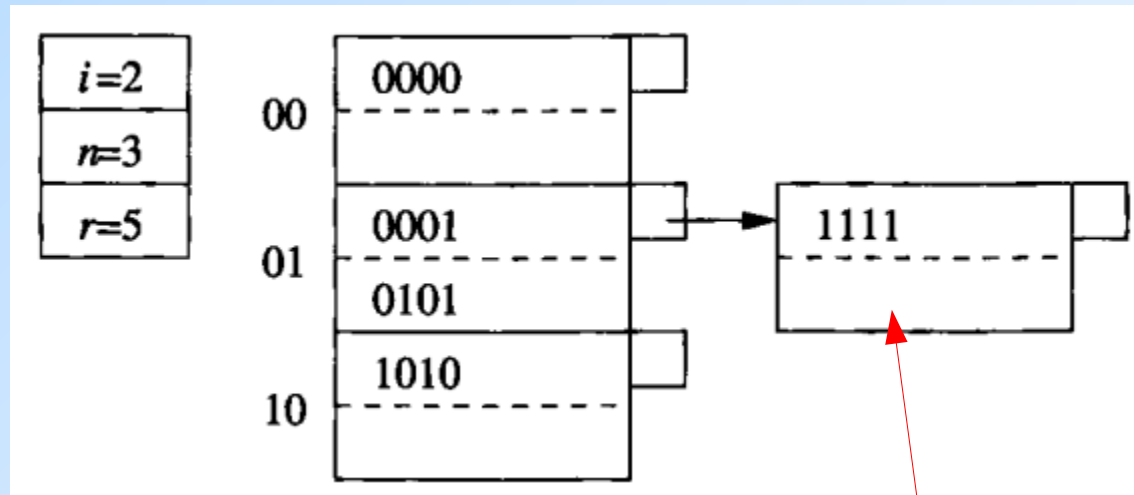
Obs 1: Na distribuição dos registros no *bucket* 01 após a criação do bloco de *overflow*, optou-se por manter os registros ordenados por $h(K)$, mas essa ordenação não é necessária.

Obs 2: Como $r/k = 5/3 < 1.7$, então **não** foi preciso incluir novos *buckets* na tabela de *hash*.

Inserção em uma Tabela de *Hash* Linear

Exemplo 3: inserção de um registro com chave K tal que $h(K) = 0111$.

Como $i = 2$, então usamos os dois últimos dígitos de $h(K)$ para determinar qual *bucket* deve conter o novo registro. Logo, o novo registro deve ficar no *bucket* de número 11.

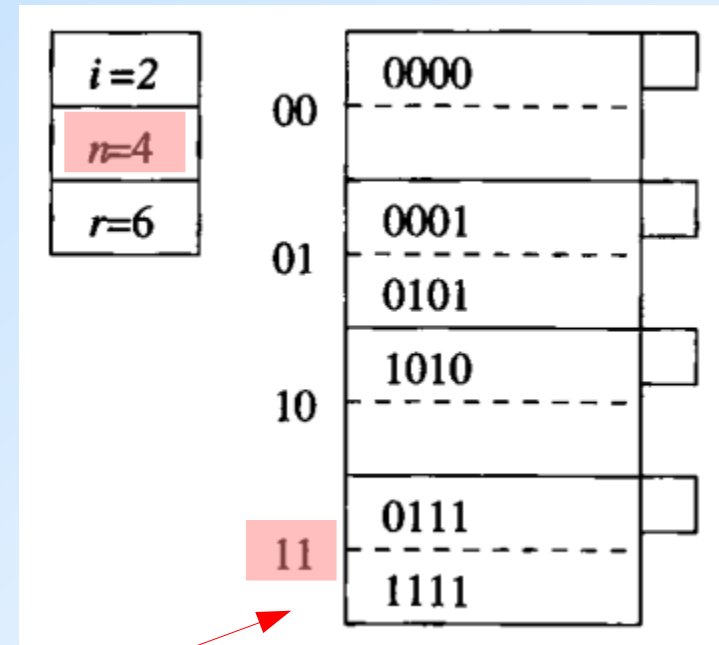
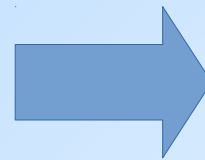
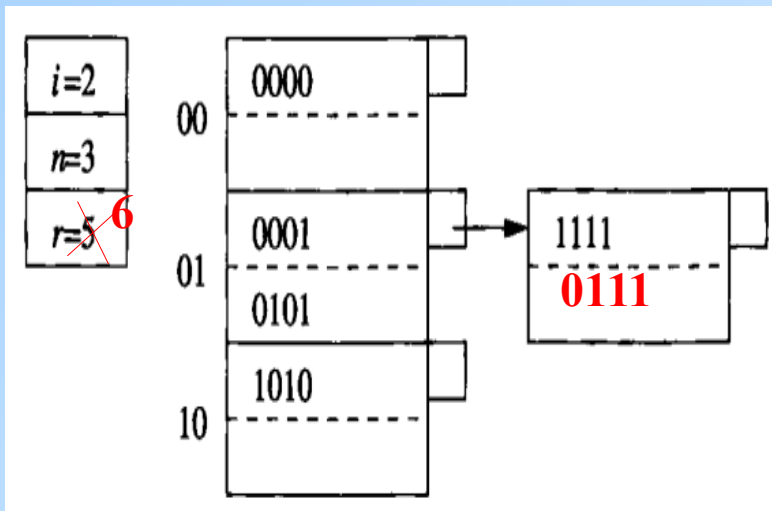


Problema: o *bucket* 11 não existe ainda!

Solução: redirecionar esse novo registro para o *bucket* 01, cujo número difere de 11 apenas por ter 0 no primeiro bit. O novo registro cabe no bloco de *overflow* do *bucket* 01.

Inserção em uma Tabela de *Hash* Linear

**Exemplo 3: inserção de um registro com chave K tal que $h(K) = 0111$.
(Passo 2, após a inserção do novo registro no *bucket* 01)**



Novo problema: $r/n = 6/3 > 1.7$

Solução: criar um novo *bucket* numerado com 11 e redistribuir os registros do *bucket* 01 entre os *buckets* 01 e 11.

Declaração de Índices na SQL

◆ Sintaxe típica:

```
CREATE INDEX <nome_indice> ON  
  <nome_tabela>(<lista de atributos>);
```

◆ Exemplos:

```
CREATE INDEX IdxFabRefri ON  
  Refrigerantes(fabricante);
```

```
CREATE UNIQUE INDEX IdxRefri ON  
  Refrigerantes(nome);
```

```
CREATE INDEX IdxVenda ON  
  Vendas(nome_lanch,nome_refri);
```

O Uso de Índices

- ◆ Dado um valor v , o índice nos leva apenas às tuplas que possuem v como valor para o(s) atributo(s) do índice
- ◆ **Exemplo:** use `IdxFabRefri` e `IdxVendas` para encontrar o preço dos refrigerantes produzidos pela 'Cola-Coca' e vendidos no 'Sujinhos'. (próximo slide)

O Uso de Índices (2)

```
SELECT price FROM
           Refrigerantes, Vendas
WHERE fabricante = 'Cola-Coca' AND
       Refrigerante.nome =
       Vendas.nome_refri AND
       nome_lanch = 'Sujinhos';
```

1. Usa IdxFabRefri para obter todos os refrigerantes feitos pela 'Cola-coca'.
2. Depois, usa IdxVendas para obter os preços desses refrigerantes no Sujinhos

Sintonia Fina (*Tuning*) de BDs

- ◆ Uma das principais dificuldades relacionadas a acelerar o acesso a um BD é decidir quais índices criar
- ◆ **Pró:** Um índice melhora o desempenho de consultas que podem usá-los
- ◆ **Contra:** Um índice piora o desempenho das modificações feitas sobre sua relação (porque uma modificação na relação pode fazer com que o índice precise ser atualizado também)

Efeitos “Colaterais” do Uso de Índices

- ◆ Mais consumo de espaço de armazenamento (problema pequeno)
- ◆ Custo para a criação do índice (problema médio)
- ◆ Custo para a manutenção do índice (problema grave)
 - ▶ Os benefícios do uso de um índice para melhorar o tempo de execução das consultas podem ser mitigados por esse custo de manutenção

Exemplo: Sintonia Fina

- ◆ Suponha que as únicas coisas que fazemos sobre o nosso BD de refri são:
 1. Inserir novos fatos em uma relação (10%).
 2. Encontrar o preço de um dado refri em uma dada lanchonete (90%).
- ◆ Nesse caso, **IdxVendas** em `Vendas(nome_lanch, nome_refri)` realmente ajudaria em um melhor desempenho, mas **IdxRefri** sobre `Refrigerantes(fabricante)` somente atrapalharia .

Benefícios de um Índice Dependem de:

- ◆ Tamanho da tabela (e possivelmente de seu *layout*)
- ◆ Distribuição dos valores das colunas indexadas
- ◆ Frequência de consultas X frequência de modificações

Softwares “Conselheiros” para Sintonia Fina (*Tuning Advisors*)

- ◆ Subárea de pesquisa muito importante
 - ▶ Fazer a sintonia de um BD de forma manual é uma tarefa muito árdua
- ◆ Um conselheiro obtém um conjunto de consultas que serão usadas como carga de trabalho para avaliar o desempenho do BD. Para constituir essa carga, o conselheiro:
 1. Escolhe consultas de forma aleatória, a partir de um histórico de consultas executadas sobre o BD ou
 2. Usa consultas de exemplo fornecidas por um projetista do BD

Software “Conselheiros” para Sintonia Fina (*Tuning Advisors*)

- ◆ O conselheiro gera índices candidatos e avalia cada um deles usando a carga de trabalho (= consultas) selecionada:
 - ▶ Cada consulta da carga é submetida ao otimizador de consulta, que assume que somente o índice em avaliação está disponível
 - ▶ A melhora/degradação no tempo médio de execução das consultas é medida

Plano de Execução de uma Consulta

- ◆ Para ver qual é o plano de execução que um SGBD usa para uma dada consulta, usamos o comando **EXPLAIN**.
- ◆ Exemplo:

```
EXPLAIN SELECT price FROM Refrigerantes, Vendas
WHERE fabricante = 'Cola-Coca' AND
  Refrigerante.nome = Vendas.nome_refri AND
  nome_lanch = 'Sujinhos';
```

Referências Bibliográficas

- ◆ Sobre estruturas de índices:
Capítulo 14 do livro “Database Systems – The Complete Book” (2ª edição), Garcia-Molina, Ullman e Widom
- ◆ Para mais detalhes sobre gerenciamento de armazenamento secundário, ler também:
Capítulo 13 do livro “Database Systems – The Complete Book” (2ª edição), Garcia-Molina, Ullman e Widom

Criação de Índices no PostgreSQL

- ◆ Estrutura geral (bem simplificada!):

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ]
```

```
<nome do índice> ON < nome da tabela >
```

```
[ USING <método> ] ( { coluna } [ ASC | DESC ] [, ...] );
```

- ◆ **UNIQUE** cria uma restrição de unicidade sobre a(s) coluna(s) do índice.
- ◆ **CONCURRENTLY** permite o PostgreSQL construir o índice sem bloquear a tabela para modificações concorrentes (inserts, updates ou deletes) → **isso pode ser perigoso!**

Criação de Índices no PostgreSQL

- ◆ O índice também pode ser construído para valores computados a partir de uma expressão envolvendo um ou mais atributos de uma tabela. Ex:

```
CREATE INDEX idx_maiuscula ON  
Lanchonete(upper(nome)) ;
```

- ◆ Além de índices para tabelas, podemos também criar índices para visões materializadas.

Métodos (Tipos) para Índices no PostgreSQL

Métodos existentes no PostgreSQL:

Btree (padrão), Hash, GiST, SP-GiST e GIN

- ◆ Somente o método **Btree** pode ser usado para índices do tipo **UNIQUE**.
- ◆ Somente os métodos **Btree, GiST e GIN** suportam índices com várias colunas.
- ◆ Índices com mais de um campo somente serão utilizados para agilizar consultas se as cláusulas envolvendo os campos indexados forem ligadas por AND.
- ◆ Quando indicamos atributos como chave primária em uma tabela, automaticamente é criado um índice Btree sobre eles.

Métodos para a Criação de Índices no PostgreSQL

- ◆ **Btree** - Árvores-B podem ser usadas em consultas com condições de seleção baseadas em igualdade de valores ou em intervalos, sobre dados que podem estar armazenados ordenadamente
- ◆ O planejador de consultas do PostgreSQL considerará o uso de um índice do tipo **Btree** sempre que uma coluna indexada estiver envolvida em uma comparação usando um ou mais dos seguintes operadores:
 - ▶ **<, <=, =, >= e >**
 - ▶ **like** (se o padrão for uma constante e estiver ancorado no início da string, como em coluna like 'MAC%')

Métodos para a Criação de Índices no PostgreSQL

- ◆ **Hash** - só pode ser usado em consultas envolvendo condições de seleção baseadas em simples comparações de igualdade
- ◆ O planejador de consultas do PostgreSQL considerará o uso de um índice do tipo **Hash** sempre que uma coluna indexada estiver envolvida em uma comparação usando o operador “=”
- ◆ Nota: por problemas na forma como ele é implementado no gerenciador, o uso desse tipo de índice no PostgreSQL atualmente é desencorajado

Referências Bibliográficas para o PostgreSQL

- ◆ <http://www.postgresql.org/docs/9.6/interactive/indexes.html>