Hashing

Estrutura de Dados II

Jairo Francisco de Souza

Hashing para arquivos extensíveis

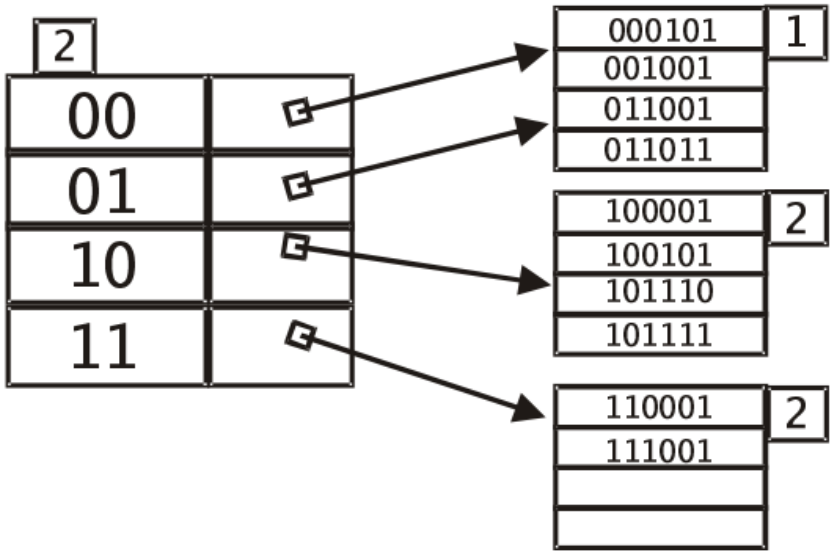
- Todos os métodos anteriores previam tamanho fixo para alocação das chaves.
- Existem várias técnicas propostas para implementar hashing em arquivos, os quais possuem tamanhos variáveis, com inserções e deleções constantes.
- Podem ser divididas em
  - Técnicas que usam diretórios
    - hashing expansível (Knott, 1971)
    - hashing dinâmico (Larson, 1978)
    - *hashing extensível* (Fagin et al, 1979)
  - Técnicas que não usam diretórios
    - hashing virtual (Litwin, 1978)
    - *hashing linear* (Litwin, 1980)

Hashing extensível

- Situação: balde (página primária) fica cheio.
- Porque não reorganizar o ficheiro duplicando o número de baldes?
- Ler e escrever todas as páginas é dispendioso!
- Idéia: usar um diretório de ponteiros para baldes, duplique o número de baldes através da duplicação do diretório, particionando justamente o balde que transbordou!
- Um diretório é muito menor que um ficheiro, de modo que duplicá-lo é muito menos dispendioso. Só uma página de verbetes de dados é particionada.
- O truque reside em como a função de hashing é ajustada!

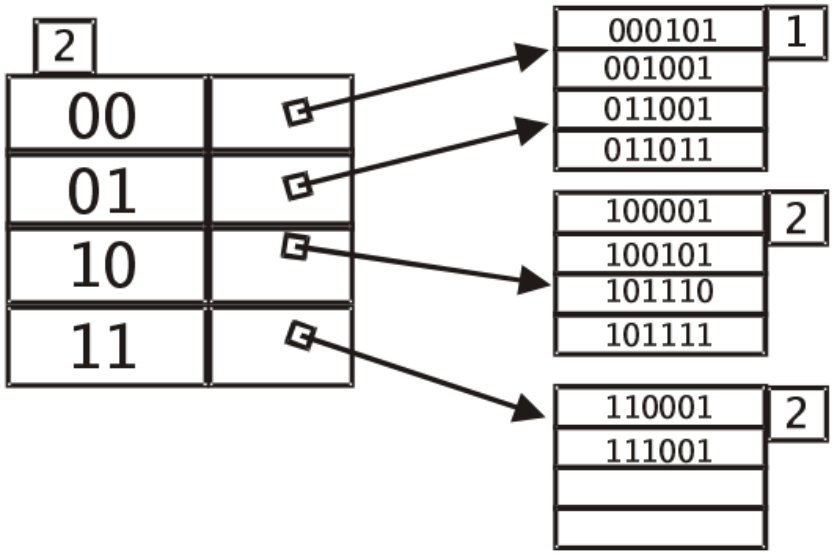
Hashing extensível

- A função do diretório é conter informações que indiquem a localização do balde de interesse.
  - Um diretório é um arranjo de 2^d endereços de baldes em que *d* é **profundidade global** (profundidade do diretório).
  - O tamanho de cada balde é fixo
- O valor inteiro correspondente aos primeiros *d* bits de uma chave são utilizados.
  - Supondo a profundidade do diretório igual a 2 e a pseudo-chave igual a 101110, procuraremos na posição 10 pelo endereço do balde que contém a informação desejada.



Hashing extensível

- É possível ter mais de uma entrada no diretório apontando para o mesmo balde.
- Cada balde armazena a informação de sua profundidade
  - A profundidade do balde é chamada de profundidade local.
  - A profundidade local indica quantos primeiros bits de cada pseudo-chave são comum a todos os elementos do balde.
  - A profundidade do balde pode ser menor ou igual à profundidade do diretório.

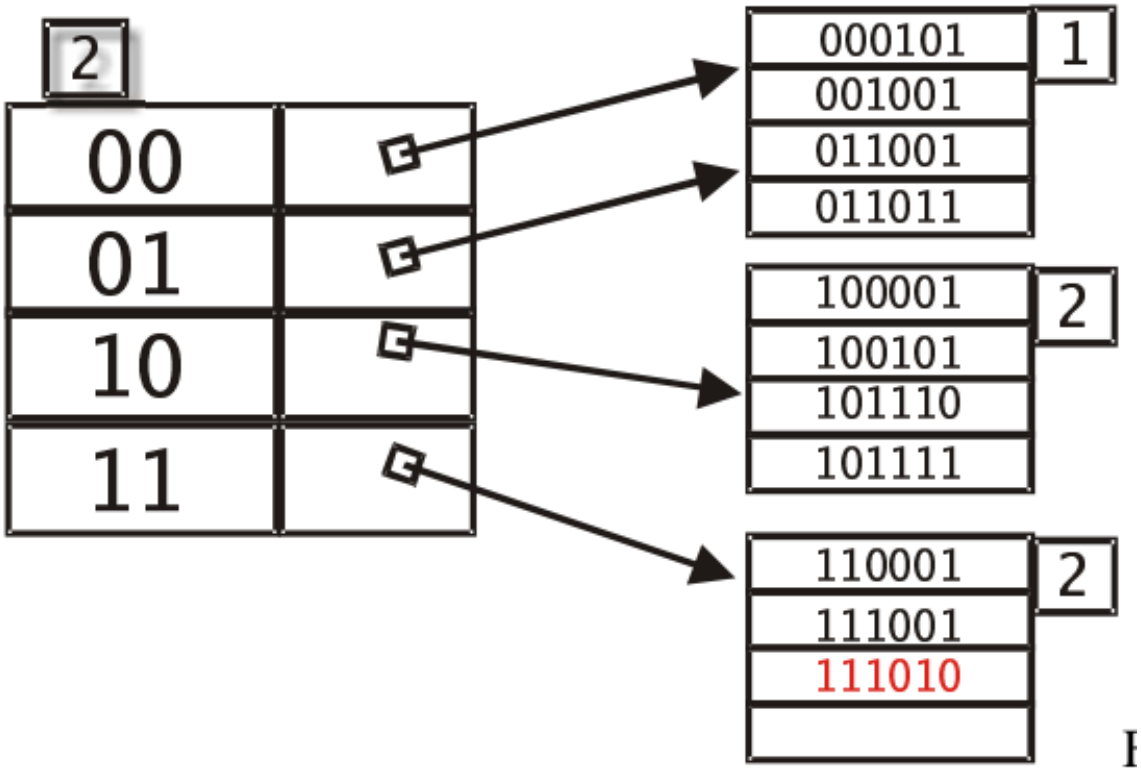


Hashing extensível

- Quando necessário, um diretório pode ser expandido
  - Isto ocorre quando “estoura” o espaço de endereços do diretório, ou seja, cada entrada aponta para um balde diferente de forma que não há como referenciar um novo balde.
  - Neste caso, o diretório dobra de tamanho
    - Ou seja, a profundidade é incrementada em 1.
- Inserção:
  - Após obtida a pseudo-chave, pega-se os *p* primeiros dígitos mais significativos da chave como índice para acessar uma posição do diretório.
    - Por exemplo, se a pseudo-chave é 10100110 e a profundidade é 3, acessaremos a posição 101, ou seja, 5 em decimal.
  - Daí, teremos 3 casos durante a inserção...

Hashing extensível: inserção

- Melhor caso
  - Há espaço no balde
  - A pseudo-chave é simplesmente inserida
  - Exemplo: inserção da pseudo-chave 111010

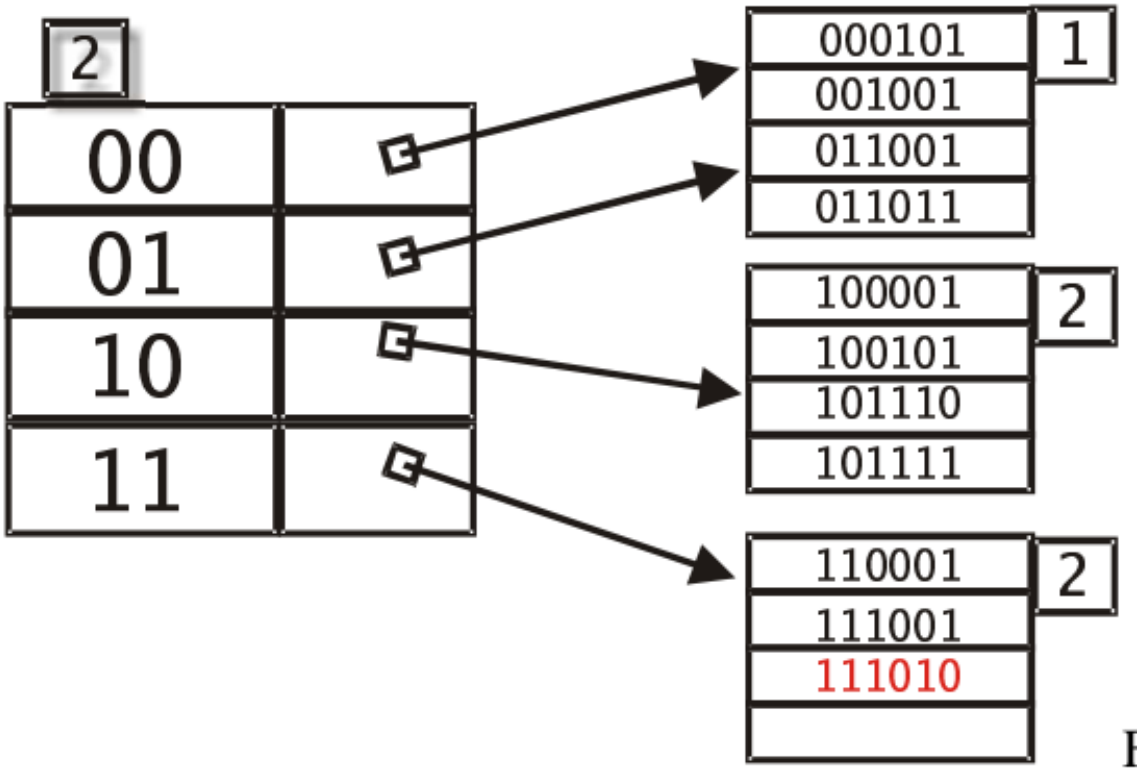


Hashing extensível: inserção

- Caso intermediário
  - Não há espaço no balde encontrado, mas ele pode ser dividido, visto que mais de uma entrada no diretório faz referência ao balde no qual estamos tentando inserir.
    - Equivalentemente, *p* do balde é menor que *p* do diretório
  - Um novo balde é criado.
  - As pseudo-chaves são redistribuídas entre o novo balde e o original
  - A profundidade local dos dois baldes serão iguais à do balde original incrementado de 1.

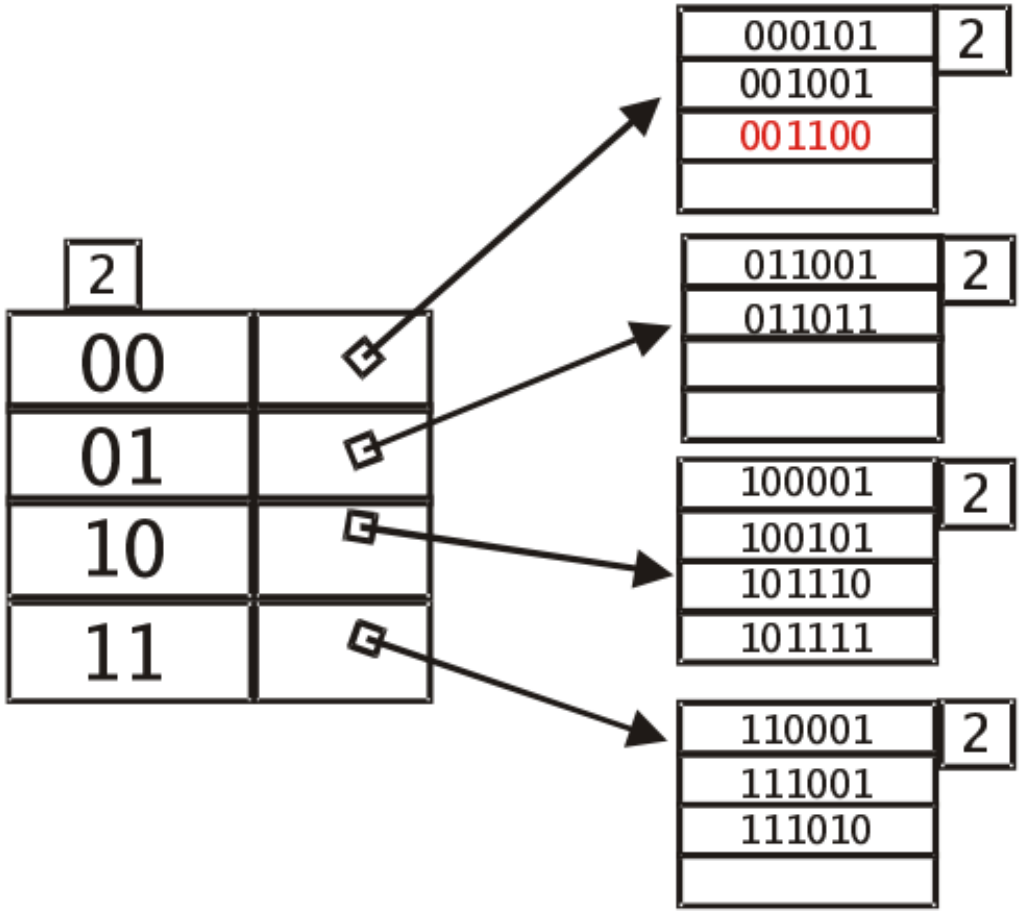
Hashing extensível: inserção

- Caso intermediário
  - Exemplo: inserção de 001100



Hashing extensível: inserção

- Caso intermediário
  - Exemplo: inserção de 001100

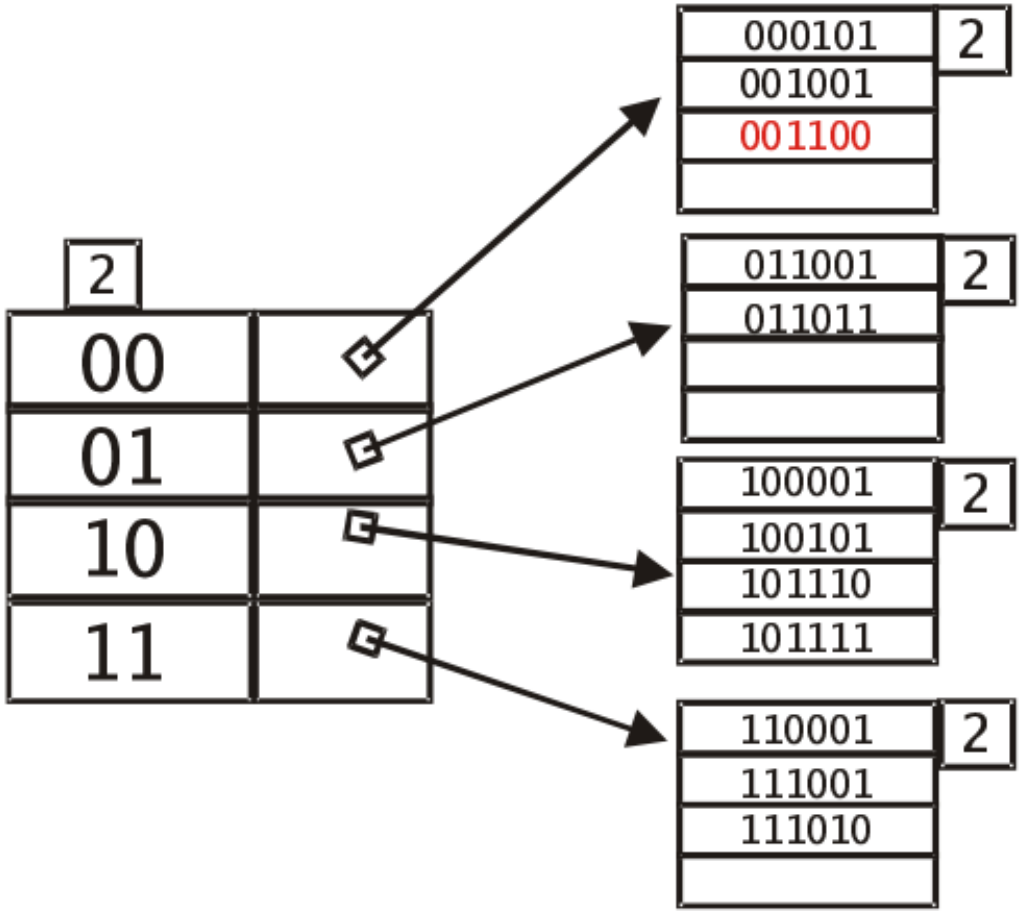


Hashing extensível: inserção

- Pior caso
  - Não há espaço no balde
  - Entrada no diretório faz referência única e exclusivamente a um balde
    - Ou seja, profundidade local é igual a profundidade global.
  - O diretório dobra de tamanho
    - Ou seja, profundidade global é incrementada de 1
  - As pseudo-chaves são distribuídas entre os baldes do novo diretório

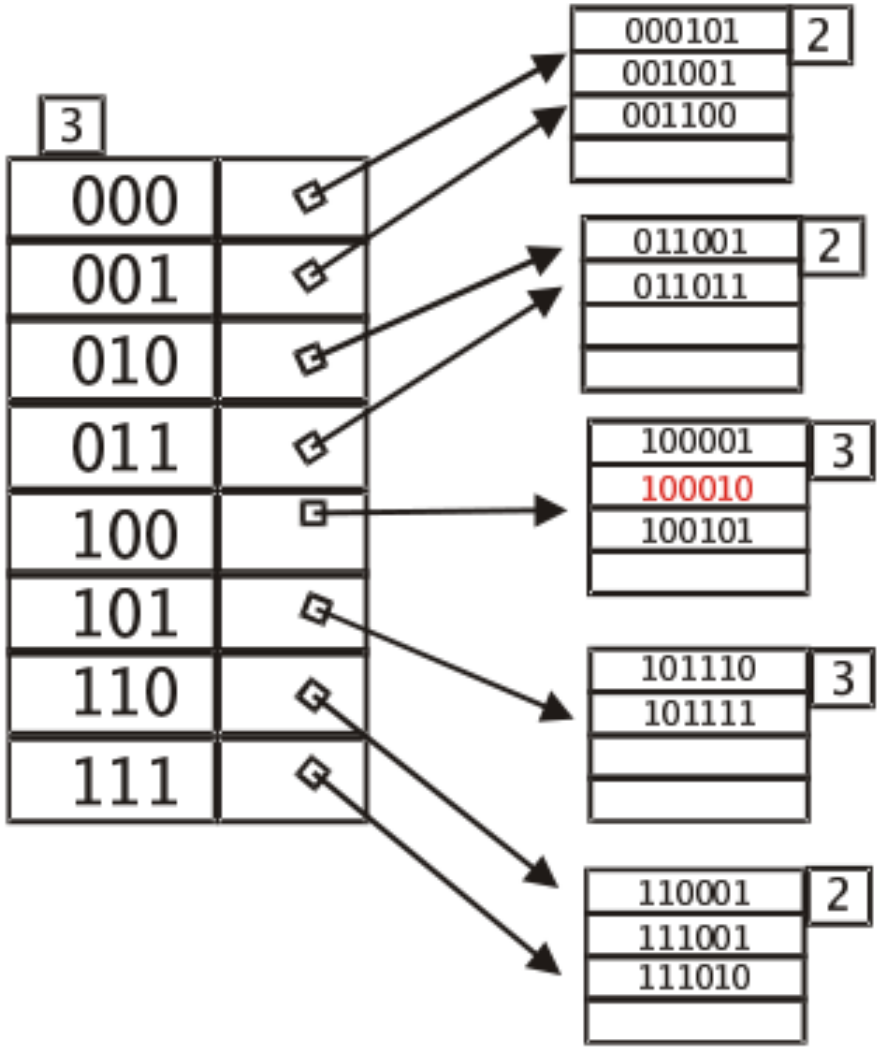
Hashing extensível: inserção

- Pior caso
  - Exemplo: Inserção de 100010



Hashing extensível: inserção

- Pior caso
  - Exemplo: Inserção de 100010

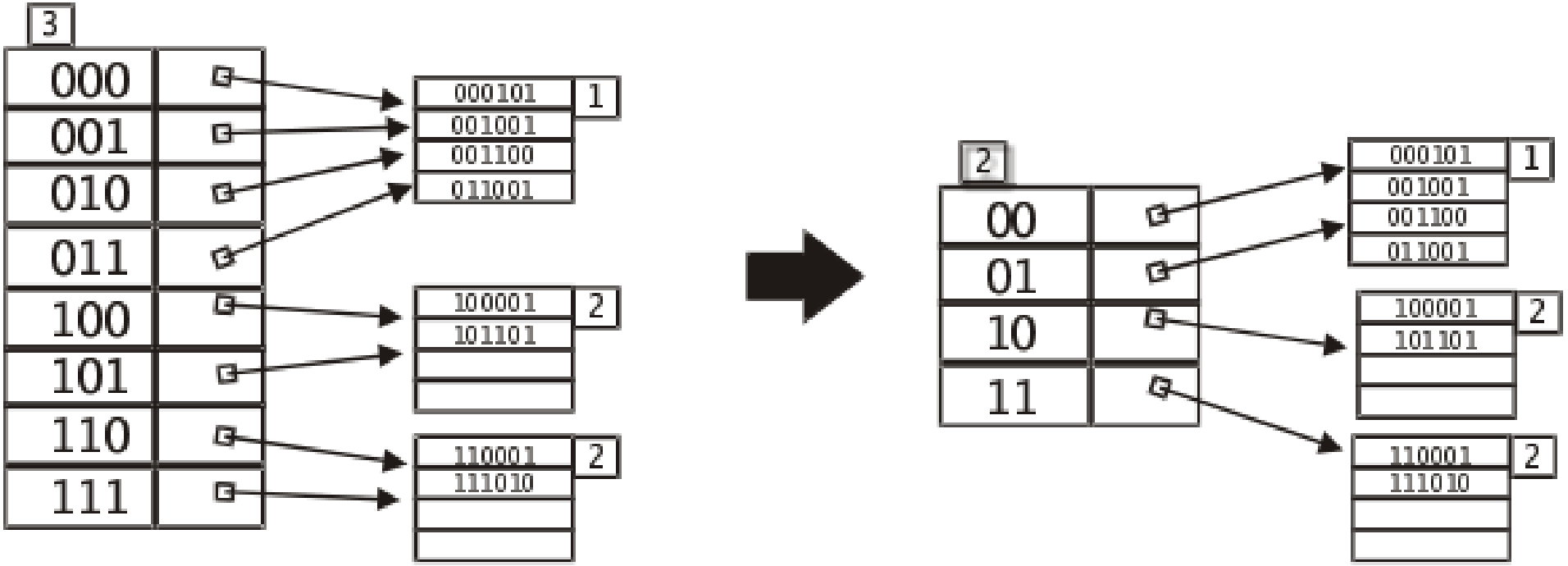


Hashing extensível: remoção

- Ao remover uma entrada, temos que fazer duas verificações
  - Se é possível fundir o balde com um balde amigo
  - Se o tamanho do diretório pode ser reduzido
- Encontrando balde amigo
  - Para haver um único balde amigo, a profundidade do balde tem que ser a mesma do diretório.
  - Dado um índice para o balde, o balde amigo é aquele que difere apenas no último bit do índice
  - Exemplo: Se um balde tem índice 000 e sua profundidade é 3 e igual a do diretório, o balde amigo tem índice 001.
  - Quando fundir?
    - Critério a cargo do programador
    - Pode-se sempre tentar fundir os baldes
    - Pode-se definir um número mínimo de informação em cada balde

Hashing extensível: remoção

- Reduzindo o diretório
  - Depois de fundir o balde, talvez o diretório possa ser reduzido
  - É possível quando cada balde é referenciado por **pelo menos duas** entradas do diretório
    - Isto é, todo *p* local **é menor** que o *p* global



Exercício

- Considere as seguintes pseudo-chaves já geradas por uma função de hash *h()*

  00011, 10011, 11011, 01100, 00110, 00100, 01011, 11100, 10100

- Pede-se
  - Mostrar a inserção, em ordem, de cada chave usando abordagem de hashing extensível
  - Mostrar a remoção das chaves 00100, 01100, 10100
- Considere
  - Diretório com profundidade 1 e balde com tamanho 2.
- Dica
  - No início somente balde é criado e todas as entradas para do diretório apontam para esse balde.
  - A profundidade local é inicialmente 0.

Hashing linear

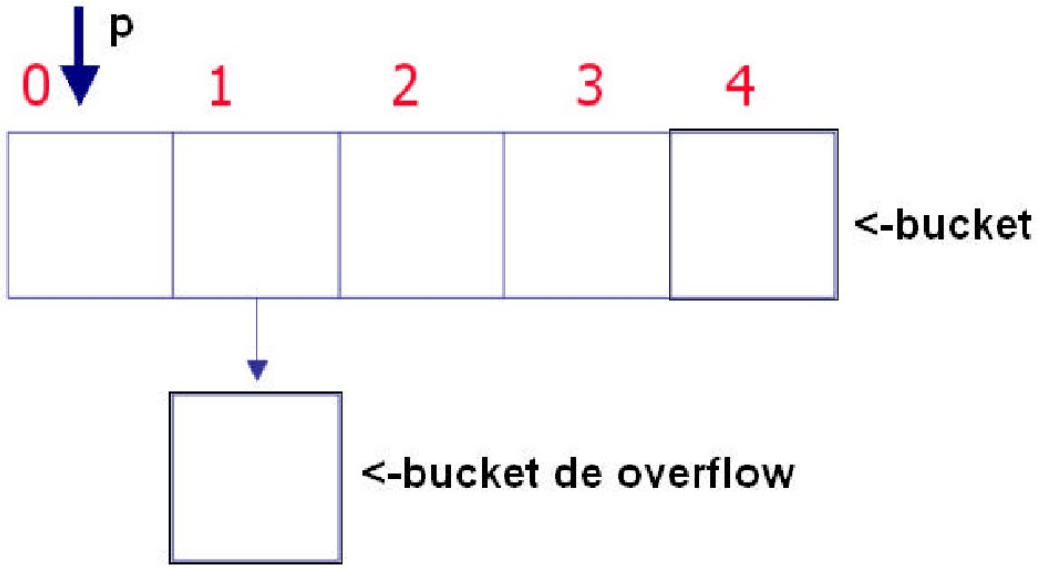
- O hashing extensível cresce de forma exponencial no momento que cada balde precisa de mais registros
- Além disso, é necessário um dicionário para armazenar o endereço de cada balde

- O hashing linear cresce de forma mais lenta
  - É criado um balde a cada momento que a tabela cresce
  - Não é necessária uma tabela de enderaçamento
    - Dessa forma, não é necessário também refazer todos os indíces a cada crescimento

- Funcionamento
  - Manipulação de espaços através de diferentes funções de hash aplicadas nos espaços disponíveis

Hashing linear

- Essa abordagem usa *buckets* que são estruturas com espaços para mais de um registro
  - Equivale a “baldes alocados em uma estrutura sequencial”
- Cada *bucket* possui um tamanho fixo *M*, ou seja, armazena *M* chaves. Inicia-se a estrutura com um número fixo de *buckets*.
- Caso o espaço do *bucket* estoure, é utilizado um *bucket de overflow* que pode ser implementado como uma lista encadeada
- Um ponteiro p determina qual o *bucket* a ser duplicado



Hashing linear

- A criação de novo *bucket* é determinado pelo fator de carga da estrutura
  - Ou seja, vários *buckets* podem conter mais registros do que o espaço determinado e fazer uso de *buckets de overflow*
  - A cada inserção, é verificado o fator de carga. Caso seja maior do que um limite estipulado, um novo espaço de alocação é determinado.

- O fator de carga é dado pela quantidade de registros inseridos divididos pela soma dos espaços possíveis nos *buckets*:
  FatorCarga = Total(registros) / M * Total(*buckets*)

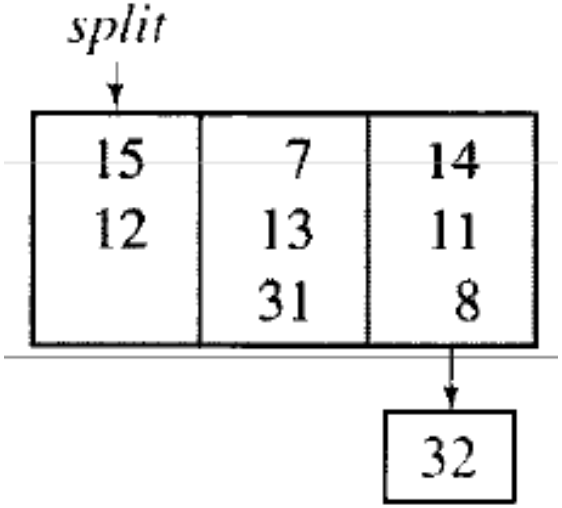
Hashing linear: inserção

- Sendo *N* o tamanho da tabela hash (número inicial de *buckets*), a inserção é dada pela fórmula h(k) = k mod 2^g * N, sendo *g* iniciado de 0.
- Ao inserir uma nova chave, aloca-a no *bucket* determinado pela fórmula acima
- Caso o *bucket* esteja cheio, a chave é alocada no *bucket de overflow.*
- Quando o fator de carga ultrapassa o máximo valor estabelecido, duplica-se o *bucket* apontado por *p*. O novo *bucket* é alocado no final da tabela e *p* é incrementado.
- Os *buckets* duplicados são reorganizados e todos os dados dos dois *buckets* passam a obedecer a fórmula h_1(k)=k mod 2^{g+1}* N.

Hashing linear: inserção

- Suponha M=3, N=3 e fator de carga = 80%. Suponha a tabela abaixo já preenchida com algumas chaves e *p* (ponteiro *split*) apontando para o índice 0 da tabela.

Fator de carga da tabela é 75% = 9 chaves / 3 *buckets* com 3 espaços e 1 *overflow* em cada *bucket*.



Pode-se também calcular o fator de carga em relação ao número de espaços em *buckets* somente, sem o *bucket de overflow*.

- Como a tabela ainda não sofreu expansão (N=3 é o valor inicial), todas as chaves foram alocadas com a função

  $h(k) = k \bmod 2^g * N$, ou seja

  $h(k) = k \bmod 2^0 * 3 = k \bmod 3$

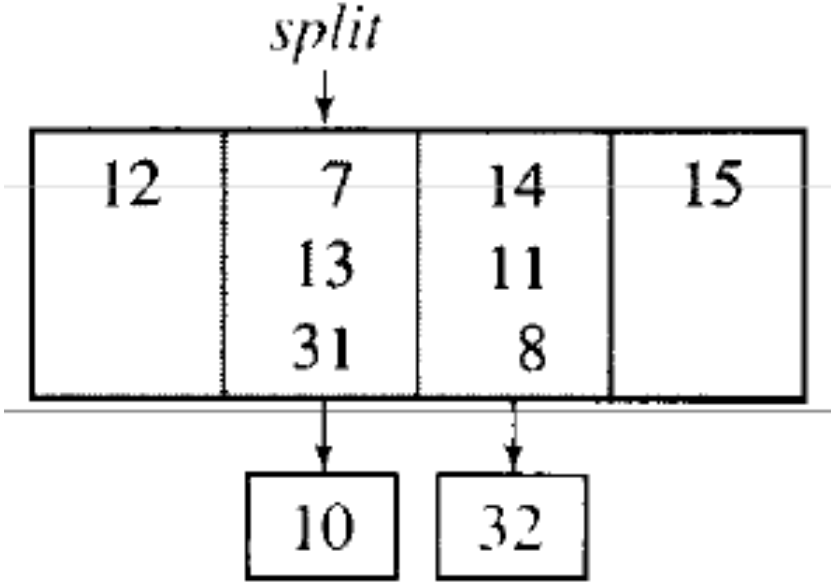
Hashing linear: inserção

- Inserindo 10, o fator de carga vai para 83% (10/12)
  - Torna-se necessário criar um novo *bucket*
  - O novo *bucket* é alocado no final da tabela.
  - As chaves dos dois *buckets* são redistribuídas com a fórmula

$h_1(k) = k \bmod 2^{g+1} * N$, ou seja

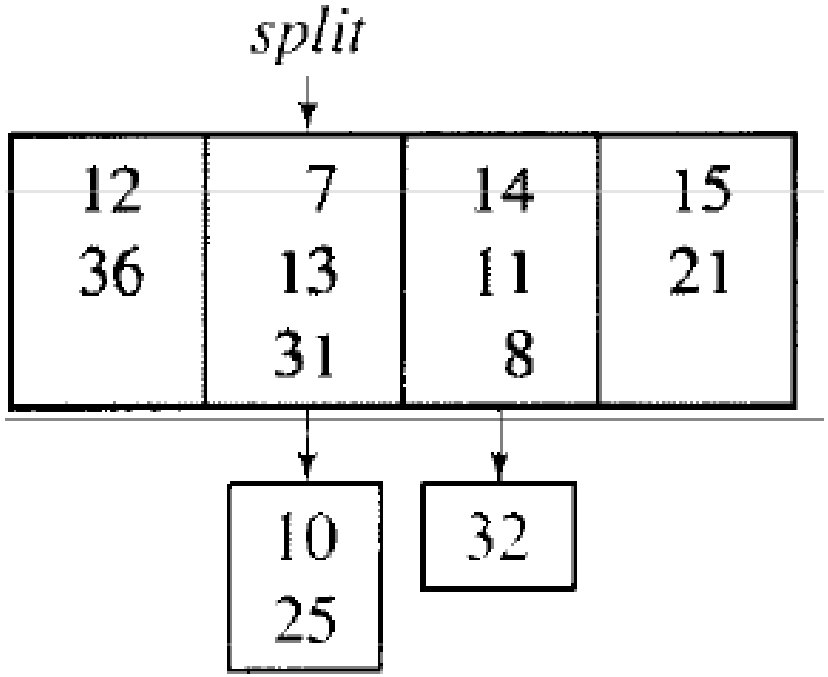
$h_1(k) = k \bmod 2^1 * 3 = k \bmod 6$

  - O ponteiro *p* (*split*) é incrementado.



Hashing linear: inserção

- Inserindo 21, 36 e 25.
  - O fator de carga vai para 87%



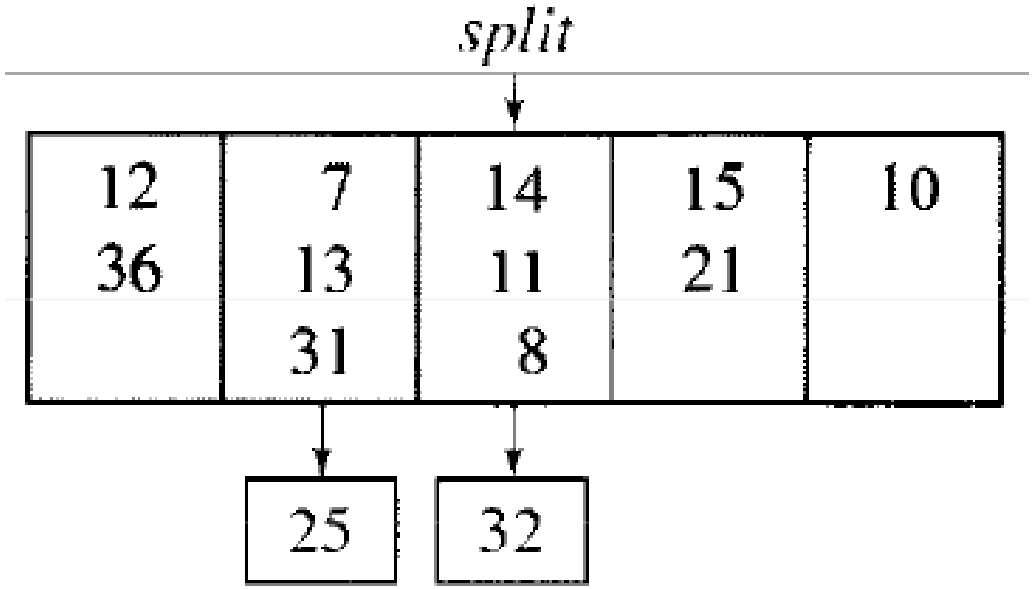
Hashing linear: inserção

- Inserindo 21, 36 e 25.
  - O novo *bucket* é alocado no final da tabela.
  - As chaves dos dois *buckets* são redistribuídas com a fórmula

$h_1(k) = k \bmod 2^{g+1} * N$, ou seja

$h_1(k) = k \bmod 2^1 * 3 = k \bmod 6$

  - O ponteiro *p* (*split*) é incrementado.



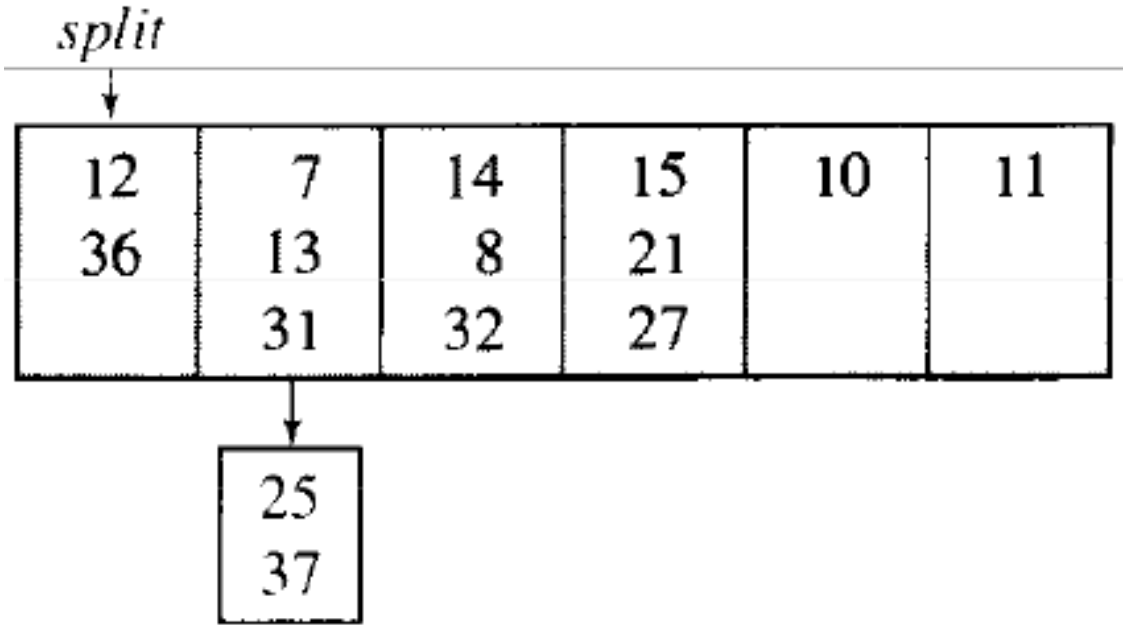
Hashing linear: inserção

- Inserindo 27 e 37.
  - O fator de carga vai para 83%.
  - As chaves dos dois *buckets* são redistribuídas com a fórmula

$h_1(k) = k \bmod 2^{g+1} * N$, ou seja

$h_1(k) = k \bmod 2^1 * 3 = k \bmod 6$

  - O ponteiro *p* (*split*) volta para a posição inicial.



Índice de *p* deve ser sempre menor que *N*!

Exercício

- Considere uma tabela com 4 *buckets* de tamanho 3 e fator de carga 70%.
- Insira as seguintes chaves:
  - 4, 8, 5, 9, 6, 7, 11, 13, 17, 15, 3, 18, 16, 36, 37

- Assuma o fator de carga como sendo a razão entre os registros inseridos pela soma dos espaços dos *buckets* (sem contar os espaços dos *buckets de overflow*)