INF / UFG

Disciplina

Banco de Dados

Conteúdo

Armazenamento, estruturas básicas de arquivo e *hashing*.

(3)

Técnicas de *hashing*

A organização baseada em *hashing* é chamada de **arquivo *hash*** (*hash file*).

A busca deve ser uma condição de igualdade em um único campo, chamado de **campo de *hash*** (na maioria dos casos, o campo de *hash* é também um campo de chave: **chave de *hash***):

- exemplo: NOME = “JOSE DA SILVA”

É aplicada uma **função *hashing* h**, chamada de uma **função *hash*** ou **função de randomização**):

>> ao campo de *hash* é aplicada à função hash, que produz o endereço do bloco do disco em que o registro está armazenado;

>> o valor do campo de *hash* é **transformado** no endereço em que está o registro procurado.

Hashing interno

Para arquivos internos (cabem na memória principal), *hashing* é tipicamente implementado como uma tabela de *hash* através do uso de um *array* de registos.

Suponha-se que o intervalo de índice de matriz é de 0 a M - 1, como se mostra na Figura 17.8 (a):

>> temos M **slots** cujos endereços correspondem ao índices de *array*.

Uma função *hash* é o comum **h (K) = K mod M**, que retorna o resto da divisão inteira por M:

>> o valor resultante é usado para o endereço de registro.

O problema com a maioria das funções *hash* é a não garantia que valores distintos resultarão endereços distintos:

>> inclusive, o espaço do número de valores possíveis de um campo de *hash* pode tomar-se geralmente muito maior do que o espaço de endereços.

Hashing interno

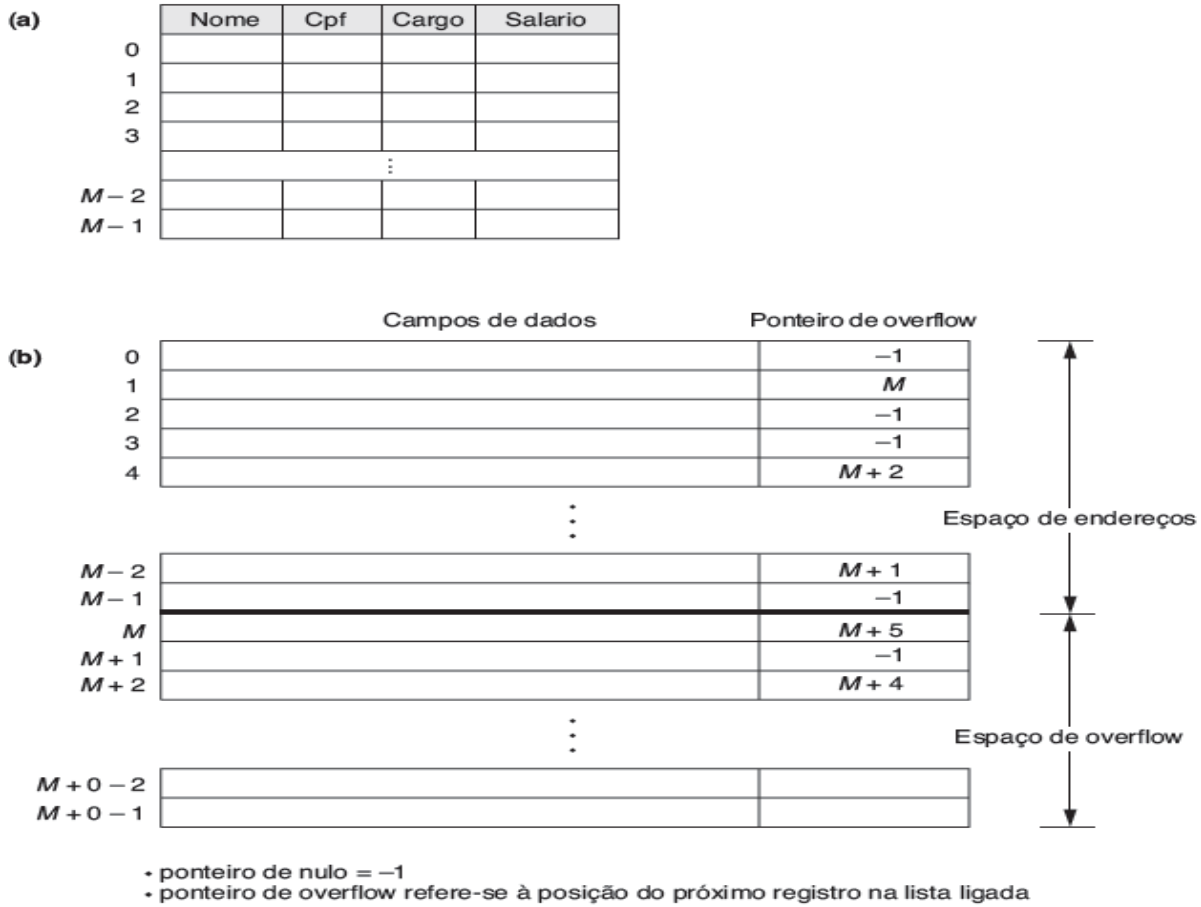


Figura 17.8
Estruturas de dados de hashing interno. (a) Array de *M* posições para uso no hashing interno. (b) Resolução de colisão ao encadear registros.

Hashing interno

O problema com a maioria das funções *hash* é a não garantia de que valores distintos resultarão em endereços distintos:

>> inclusive, o espaço do número de valores possíveis de um campo de *hash* pode tomar-se geralmente muito maior do que o espaço de endereços.

Uma **colisão** ocorre quando o valor do campo *hash* de um registro que está sendo inserido resultar em um endereço que já contém um registro diferente.

>> a função *hash* é aplicada às duas chaves distintas, em ambos os casos resultando no mesmo endereço.

É preciso solucionar as colisões, pois elas são inevitáveis.

Hashing interno

Existem vários métodos para resolver colisões.

Endereçamento aberto.
Partindo da posição ocupada especificado pelo endereço *hash*, o programa verifica as posições seguintes na ordem, até que uma posição não utilizada seja encontrada.

Encadeamento.
Vários locais de *overflow* são disponibilizados, aumentando-se o tamanho do o *array*, para se ter algumas posições de *overflow*. Também, um ponteiro é adicionado para indicar o endereço do próximo registro de *overflow*.

>> uma colisão é resolvida ao colocar o novo registro em um local de *overflow* não usado (em adição, ao ponteiro é atribuído o endereço do local de *overflow*);

>> portanto é mantida uma lista ligada de registros de *overflow*; ver Figura 17.8(b).

Hashing interno

***Hashing* múltiplos**. O programa aplica uma segunda função *hash* se a primeira resultar numa colisão. Se houver outra colisão, é usado o mecanismo de *endereçamento aberto* ou se aplica uma terceira função *hash*.

Cada método de resolução de colisão exige que seus próprios algoritmos para inserção, recuperação e exclusão de registros:

>> os algoritmos para encadeamento são os mais simples;

>> os algoritmos de exclusão de registros em *endereçamento aberto* são os mais complicados.

O objetivo de uma boa função *hash* é distribuir uniformemente os registros sobre o espaço de endereços, de modo a minimizar as colisões não deixando muitos locais não utilizados.

Hashing Externo

Hashing para arquivos em disco é chamado ***hash* externo**.

Relembrando …

>> bloco (ou página) – unidade de transferência (512...4096 Bytes).

O espaço de endereço de destino é feito de segmentos, cada um dos quais contém múltiplos registros.

Um ***bucket*** (balde) é um bloco de disco ou um conjunto de blocos de disco contíguos (*cluster*).

A função *hash* mapeia uma chave em um **número de *bucket* relativo**:

>> uma tabela mantida no cabeçalho do arquivo converte o número de *bucket* para o endereço correspondente do bloco no disco de bloco; ver Figura 17.9.

Hashing Externo

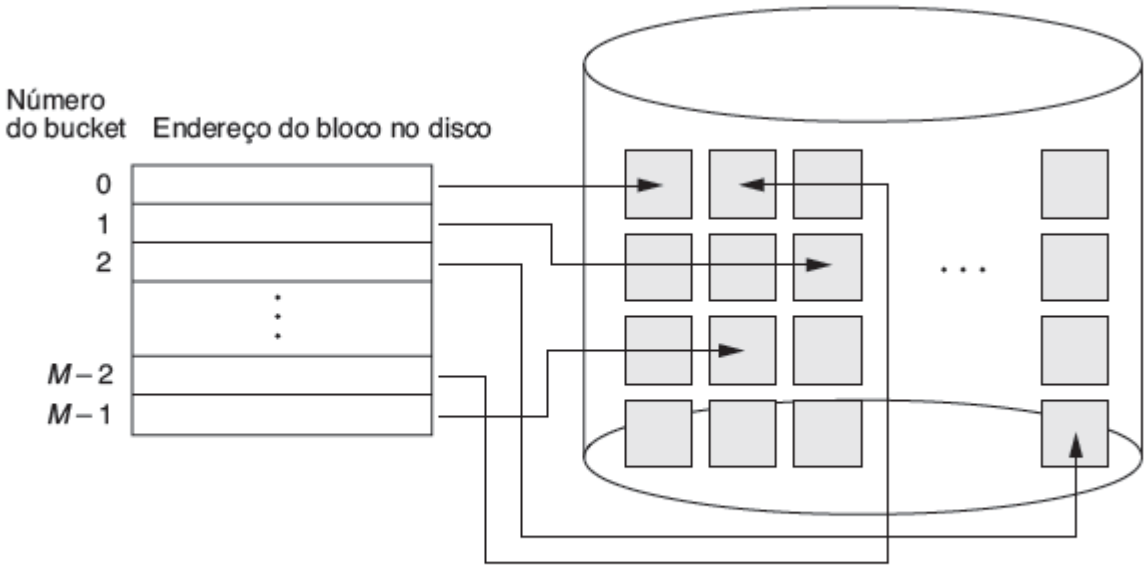
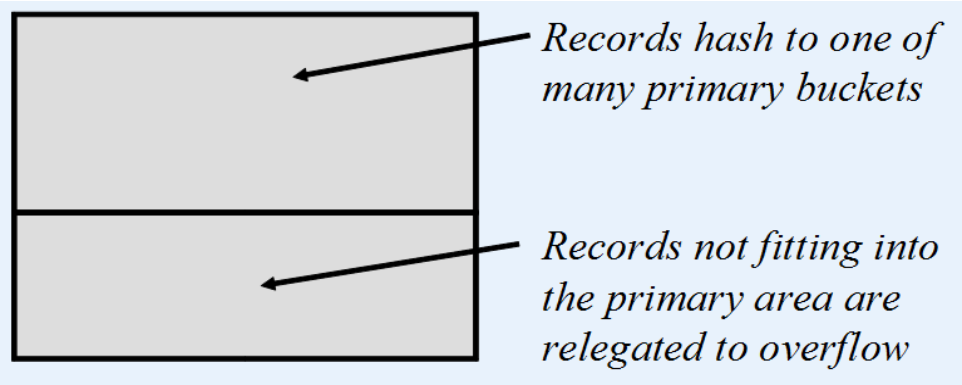


Figura 17.9
Correspondendo números de bucket a endereços de bloco de disco.

Hashing Externo

O problema de colisão é menos grave, porque vários registros cabem em um mesmo *bucket*.

No entanto, se o *bucket* estiver cheio, um ***bucket* de *overflow*** deve ser alocado.



Pode-se usar uma variação do mecanismo de *encadeamento*, em que um ponteiro é mantido em cada *bucket*, para uma lista de registros de *overflow*, como mostrado na Figura 17.10.

Hashing Externo

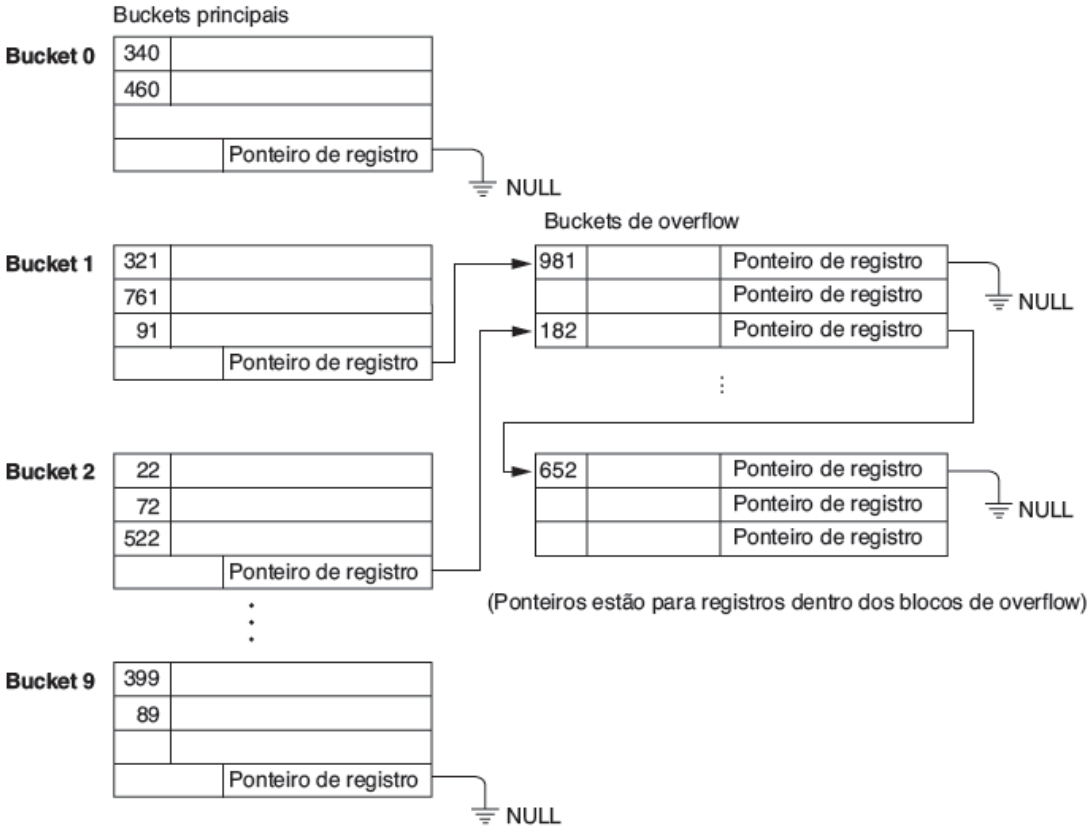
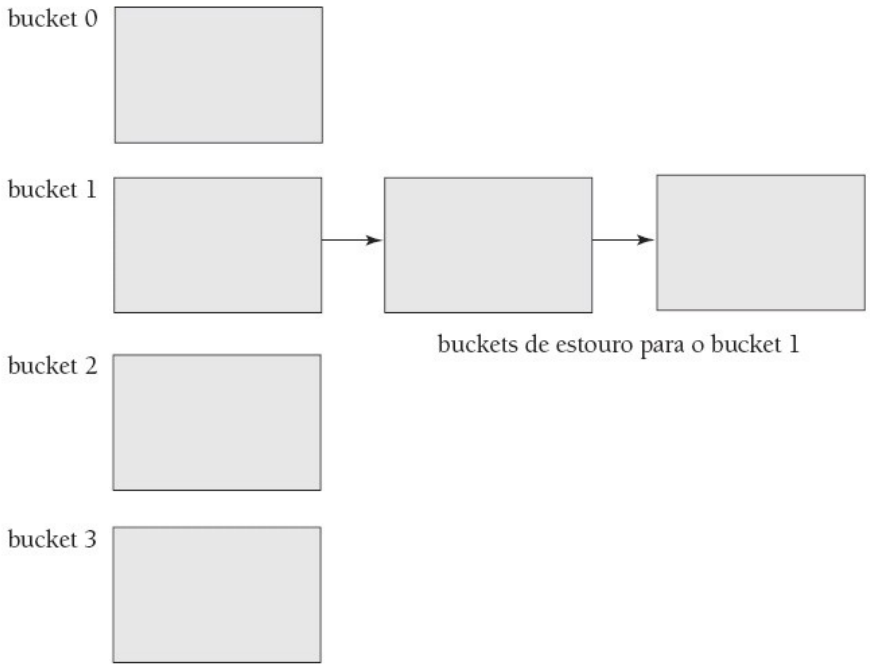


Figura 17.10
Tratamento de overflow para buckets por encadeamento.

Hashing Externo

Os buckets de *overflow* são encadeados em uma lista interligada (*hashing* fechado).
Outra alternativa, chamada *hashing* aberto, que não usa *buckets* de *overflow*, não é apropriada para aplicações de banco de dados [Silberschatz].



Hashing Externo

Quando:

(1) o número de registros tornar-se reduzido, teremos a ocorrência de espaço alocado e sem uso;
(2) o número de registro aumentar, resultará em mais colisões, tornando o processo de recuperação (busca) mais lento, devido ao aumento de listas de registros de *overflow*;

>> em ambos os casos, pode-se redimensionar M (número de blocos alocados) e aplicar uma nova função de *hash* (com o novo valor de M).

Hashing Externo

A **pesquisa** cujo predicado não explora o campo *hash* possui custo similar à pesquisa em arquivos de registros não ordenados.

Exclusão:

(1) se registro estiver no *bucket*:
>> remove o registro do *bucket* associado:
>> se for o caso, move um registro da área de *overflow* para o *bucket*, substituindo o registro excluído;

(2) se registro estiver na área de *overflow*:
>> remove o registro da lista ligada;
>> mantém uma lista ligada de espaços não utilizados na área de *overflow*.

Hashing Externo

Modificação:

(1) condição de busca:

>> busca eficiente (quando explora campo *hash*) ou busca linear;

(2) campo modificado:

>> simples se não for modificado o campo *hash*;

>> se for modificado o campo *hash*, envolve remover e inserir o registro em outro *bucket*.

Técnicas de *Hashing* para a Expansão Dinâmica do Arquivo

O maior problema com o esquema de *hashing* estático é que o espaço de endereçamento é fixo (o valor de M é fixo).

Para amenizar tal problema:

Hashing extensível
Hashing linear
Hashing dinâmico

Nesses casos, a estrutura de acesso é baseada na representação binária da função *hash* resultante (*string* de bits).

Registros são distribuídos nos *buckets* baseando-se no bits do seu valor *hash*.

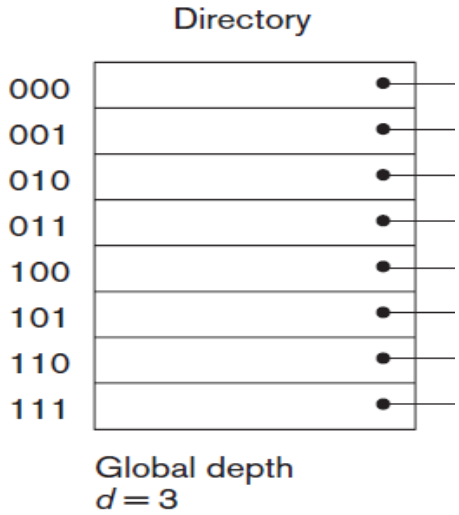
Hashing extensível

Grava uma estrutura adicional ao arquivo.
Um tipo de diretório é mantido (um *array* de $\mathbf{2^d}$ endereços de *buckets*), onde **d** é denominado profundidade **global** do diretório.

O inteiro correspondente ao **primeiros d bits** do valor *hash* é usado para:

>> determinar uma entrada no diretório, e

>> o valor (endereço) na referida entrada determina o *bucket* em que os registros correspondentes estão gravados.



Hashing extensível

Algumas entradas podem referenciar o mesmo endereço de *bucket*:
>> uma profundidade local **d'** determina o número de bits para tal.

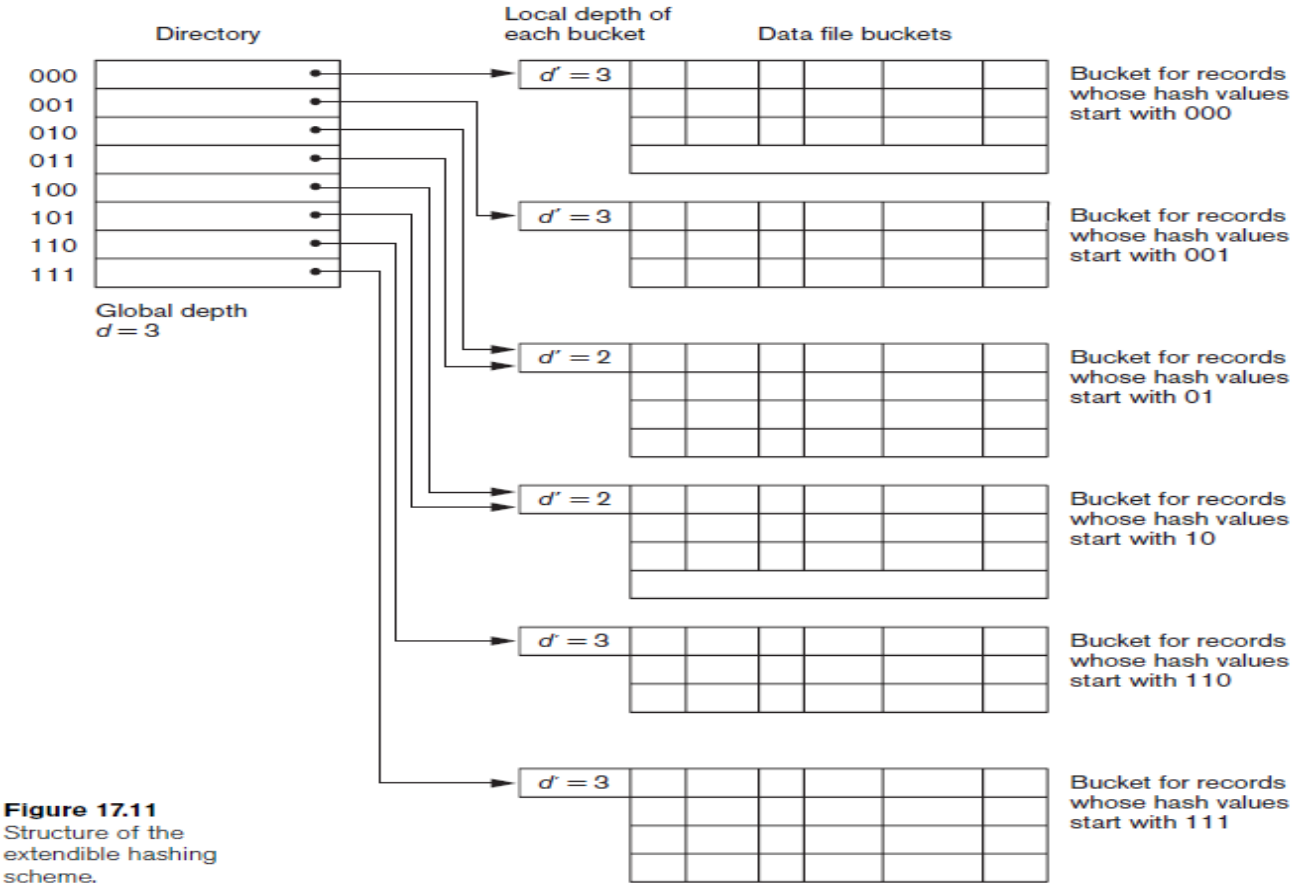


Figure 17.11
Structure of the extendible hashing scheme.

Hashing extensível

O valor de d pode aumentar ou diminuir; por exemplo:

>> aumentar (dobrar) **d**, se um *bucket*, em que d = d', estiver cheio;

>> reduzir (metade) **d**, se **d > d'** em todos os *buckets*.

Na Figura 17.1:

>> ao inserir de um novo registro cujo valor *hash* inicia com 01 (**d>d'**), cursa *overflow* no *bucket*; os registro serão, então, distribuídos em 2 *buckets*:

* um *bucket* para valor *hash* iniciando com 010
* um *bucket* para valor *hash* iniciando com 011

>> ao inserir de um novo registro cujo valor *hash* inicia com 111 (**d=d'**):

* ocorre *overflow* no *bucket*
* os valor são distribuídos em 2 *buckets*: inciando com 1110 e 1111
* o tamanho do diretório é dobrado (d=4)
* cada entrada no diretório é transformada em um par de entradas
* cada par de entradas terá o mesmo valor de ponteiro para *bucket*

Hashing extensível

Observações:

>> a performance não degrada quando o arquivo cresce;
>> não é necessário alocar espaço para crescimento futuro, pois *buckets* adicionais são adicionados quando necessário;
>> o espaço alocado na tabela de diretórios é mínimo;
>> a ocorrência de *overflow*, em muitos casos, resulta em pequena reorganização, quando os registros de um *bucket* são redistribuídos em 2 *buckets*;
>> um reorganização mais cara ocorre quando é necessário dobrar ou reduzir pela metade a estrutura de diretório;
>> o diretório é sempre acessado antes visando a determinar o endereço do *bucket* correspondente ao registro pesquisado (acesso a 2 blocos; no *hashing* estático é possível acessar apenas um único bloco):
    * penalidade pequena quando o arquivo é dinâmico.

Hashing Dinâmico

Precursor do *hashing* extensível.
O endereçamento de *buckets* considera os **n** (ou n-1) bits mais significativos.

Mantém um estrutura de diretório em árvore (diferente do *hashing* extensível):

>> nós internos possuem 2 ponteiros:
    * esquerda corresponde ao bit 0
    * direita corresponde ao bit 1
>> nós folha possuem um ponteiro para o *bucket* com registros.

Ver Figura 17.2

Hashing Dinâmico

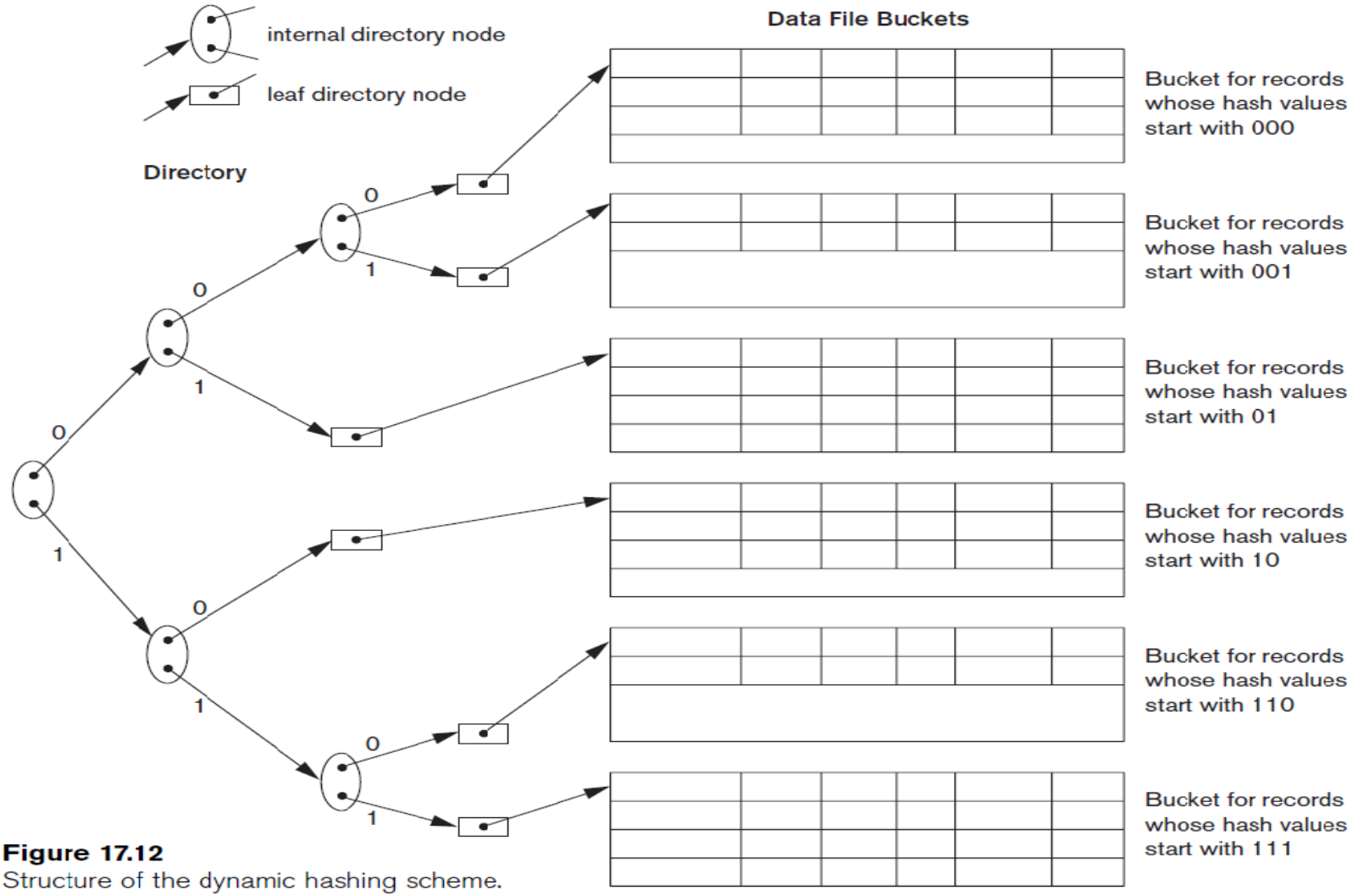


Figure 17.12
Structure of the dynamic hashing scheme.

Hashing Linear

A ideia é permitir que o arquivo possa expandir ou encolher, sem necessitar de uma estrutura de diretório.

Inicialmente, o arquivo possui **M** *buckets* primários, numerados: **0, 1, …, M-1**

O processo é dividido em várias fases: 0, 1, 2, …

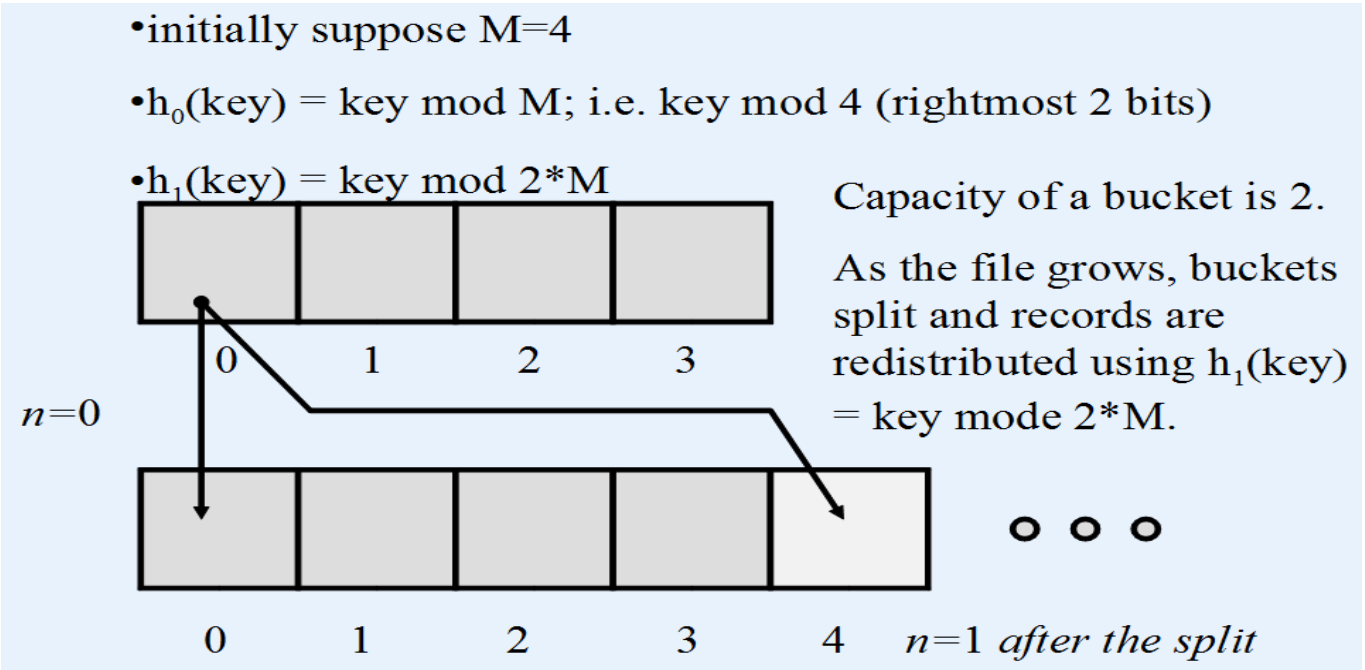
Na Fase **j**, a localização de registros em *buckets* é determinada por duas funções: $\mathbf{h_j(K)}$ e $\mathbf{h_{j+1}(K)}$, onde $\mathbf{h_j(K) = K \bmod (2^j * M)}$

Fase 0: $\mathbf{h_0(K) = K \bmod (2^0 * M)}$, $\mathbf{h_1(K) = K \bmod (2^1 * M)}$
Fase 1: $\mathbf{h_1(K) = K \bmod (2^1 * M)}$, $\mathbf{h_2(K) = K \bmod (2^2 * M)}$
Fase 2: $\mathbf{h_2(K) = K \bmod (2^2 * M)}$, $\mathbf{h_3(K) = K \bmod (2^3 * M)}$
...

Hashing Linear

Dividir um *bucket* significa redistribuir seus registros em dois *buckets*:
>> *bucket* original e novo *bucket*

A divisão de *bucket* ocorre de acordo com regras específicas:
>> ocorrência de *overflow*, ou
>> o fator de carga (*load factor*) alcança certo valor, etc.



Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.

>> regra para divisão de *buckets*: fator de carga (*load factor*) > 0.7
>> inicialmente M = 4 (M: tamanho da área primária)
>> funções *hash*: $h_i(K) = K \bmod 2^i \times M$ (i = 0, 1, 2, …)
>> capacidade de *bucket* = 2

Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.

The first phase – phase$_0$

•when inserting the sixth record we would have

4 8	1	2 14	3
0	1	2	3

n=0 before the split

(n is the point to the bucket to be split.)

•but the load factor 6/8= 0.75 > 0.70 and so bucket 0 must be split (using h_1 = Key mod 2M):

8	1	2 14	3	4
0	1	2	3	4

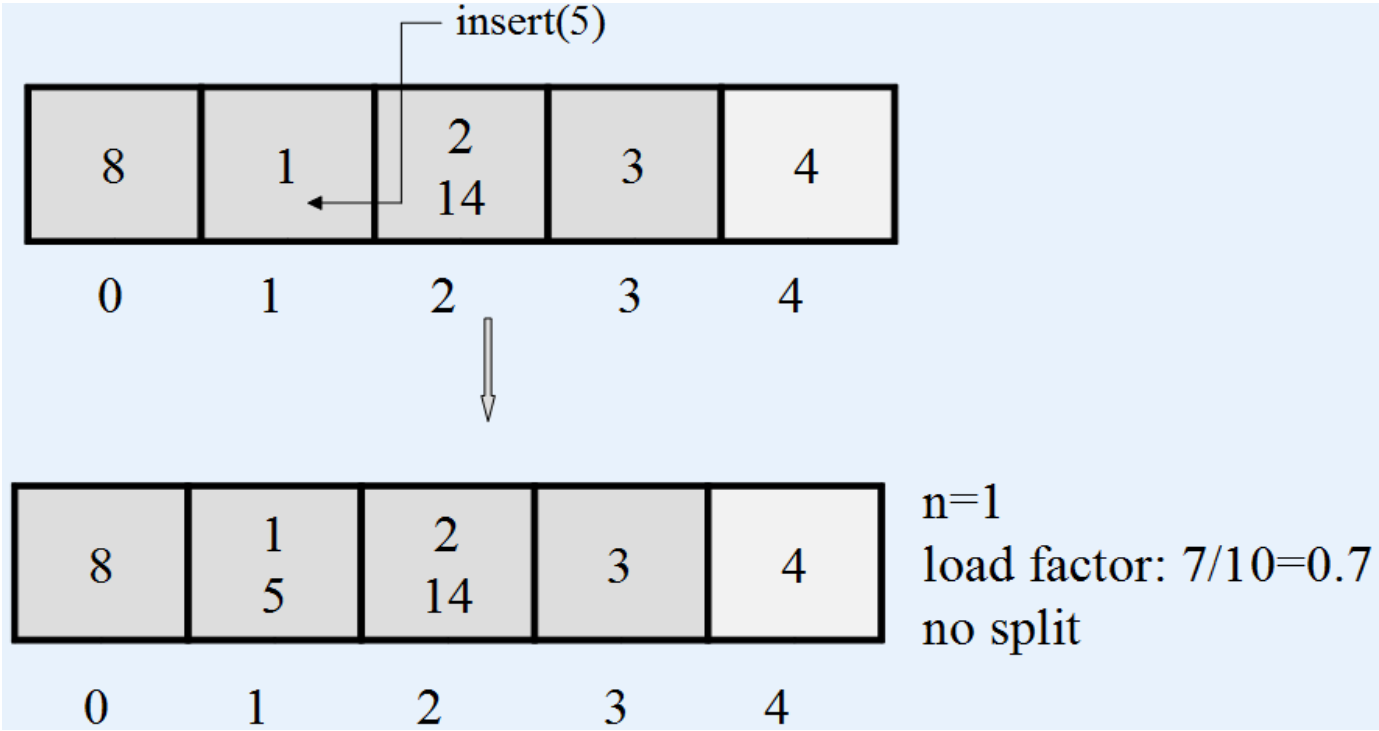
n=1 after the split

load factor: 6/10=0.6

no split

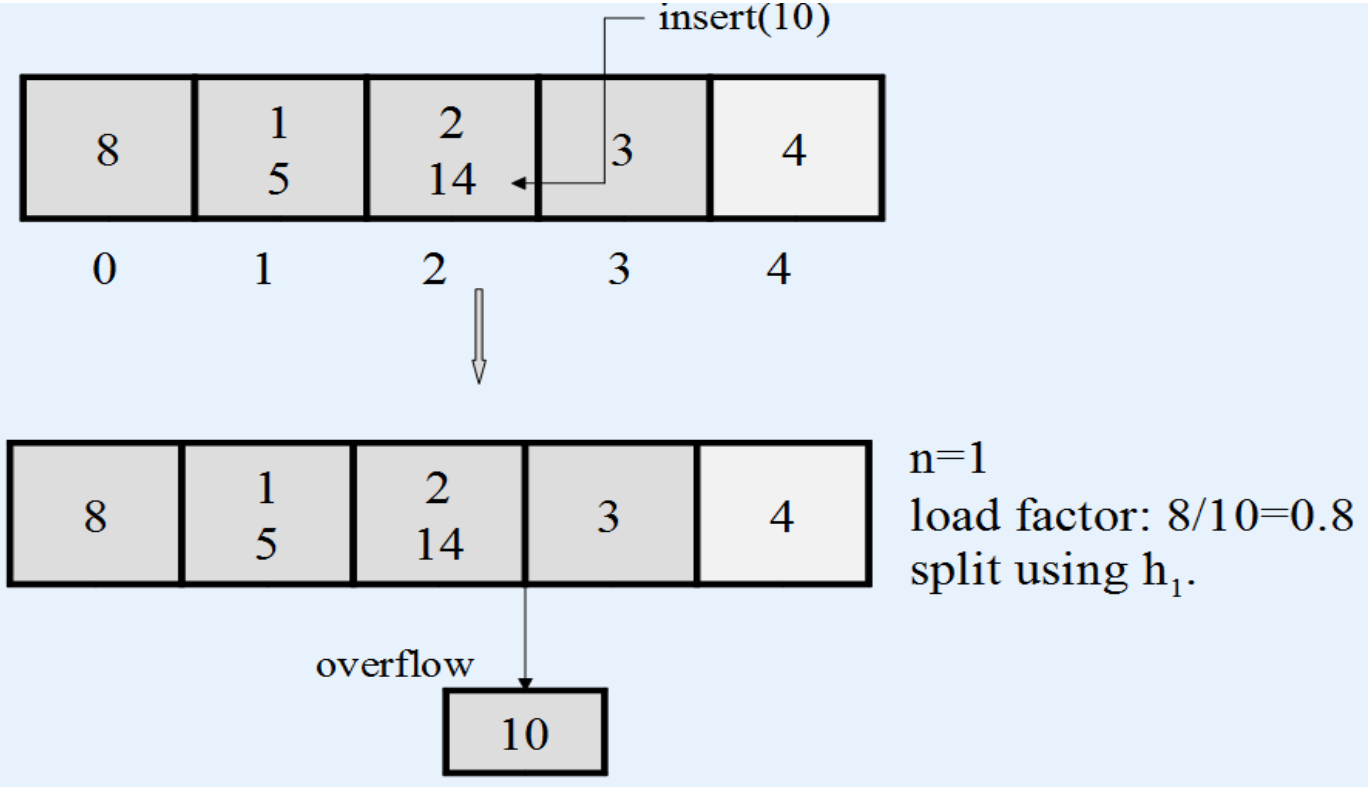
Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



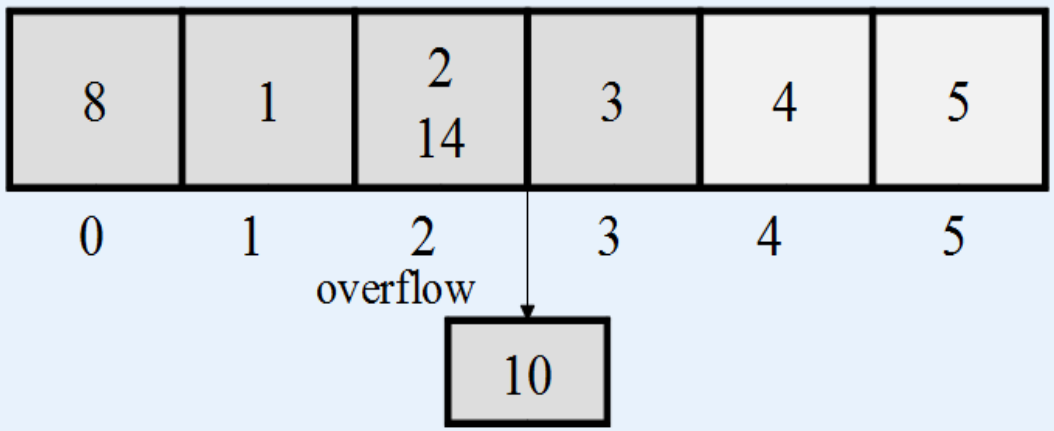
Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



Hashing Linear

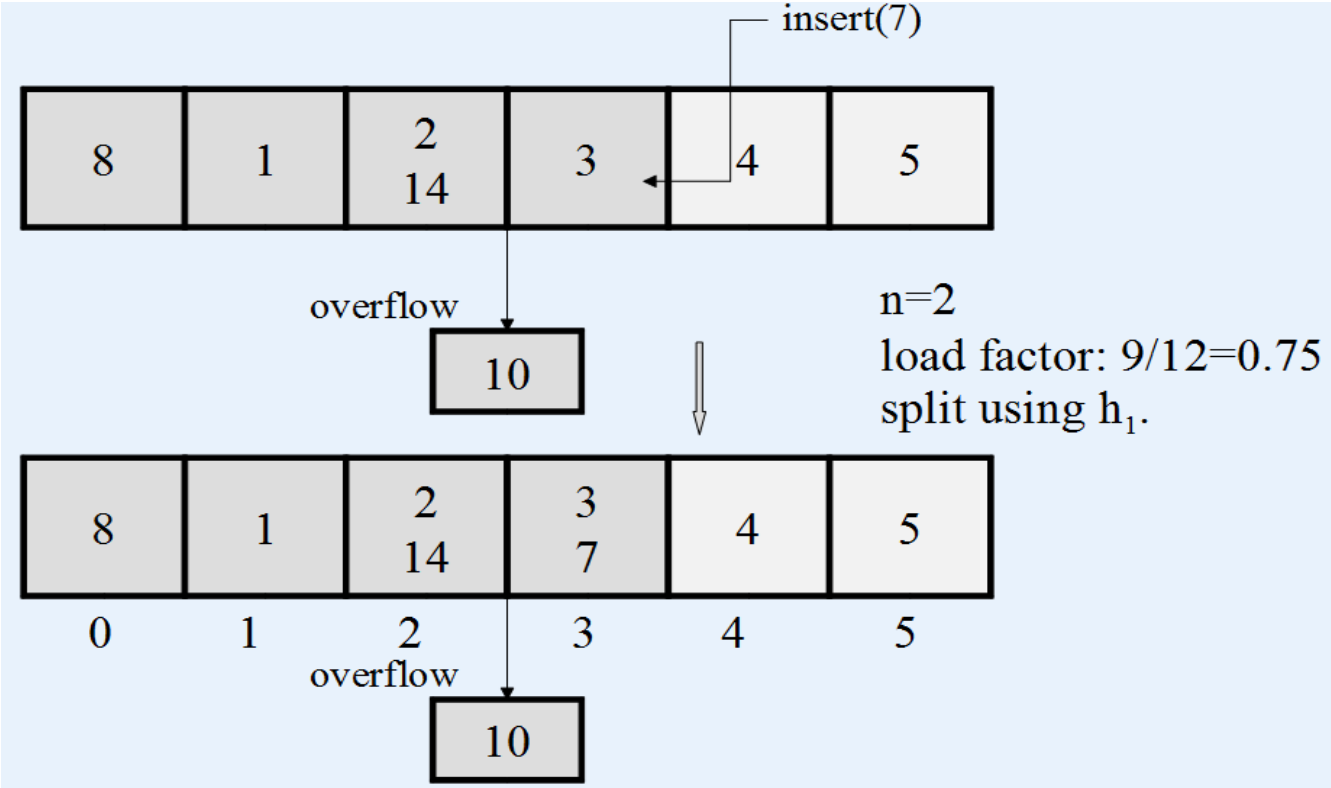
Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



n=2
load factor: 8/12=0.66
no split

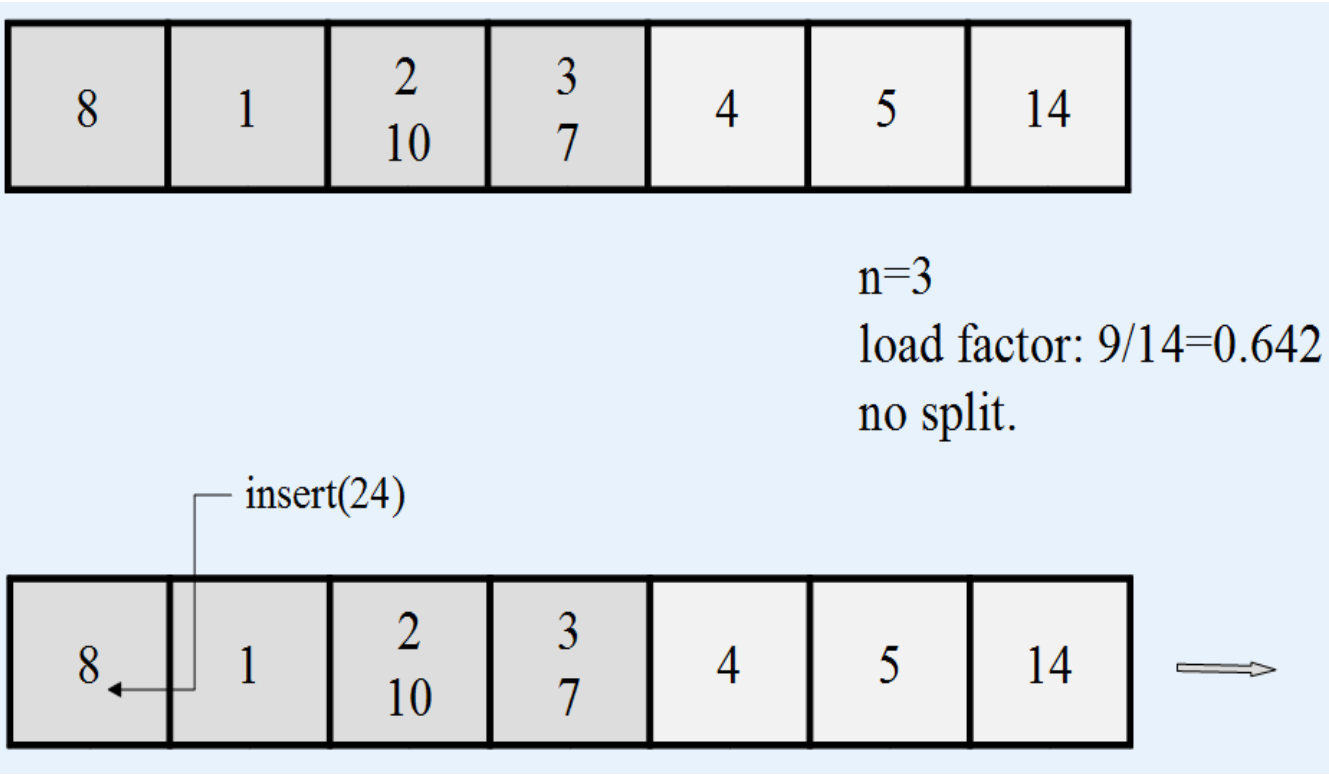
Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.

8 24	1	2 10	3 7	4	5	14

n=3
load factor: 10/14=0.71
split using h_1.

8 24	1	2 10	3	4	5	14	7

Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.

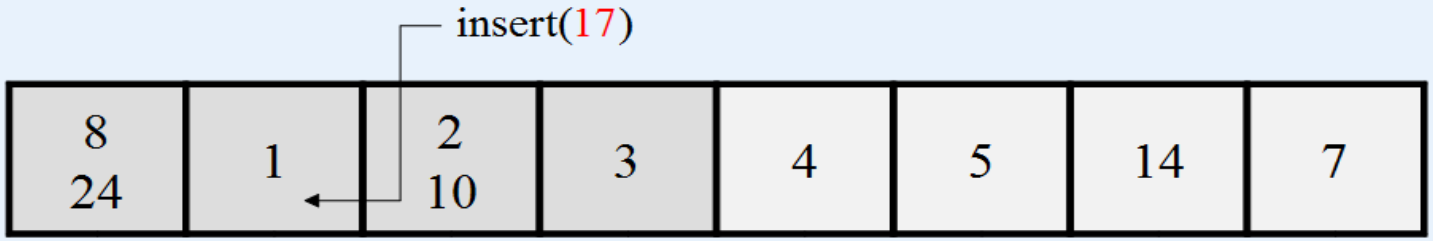
8 24	1	2 10	3	4	5	14	7

n=4

The second phase – phase$_1$

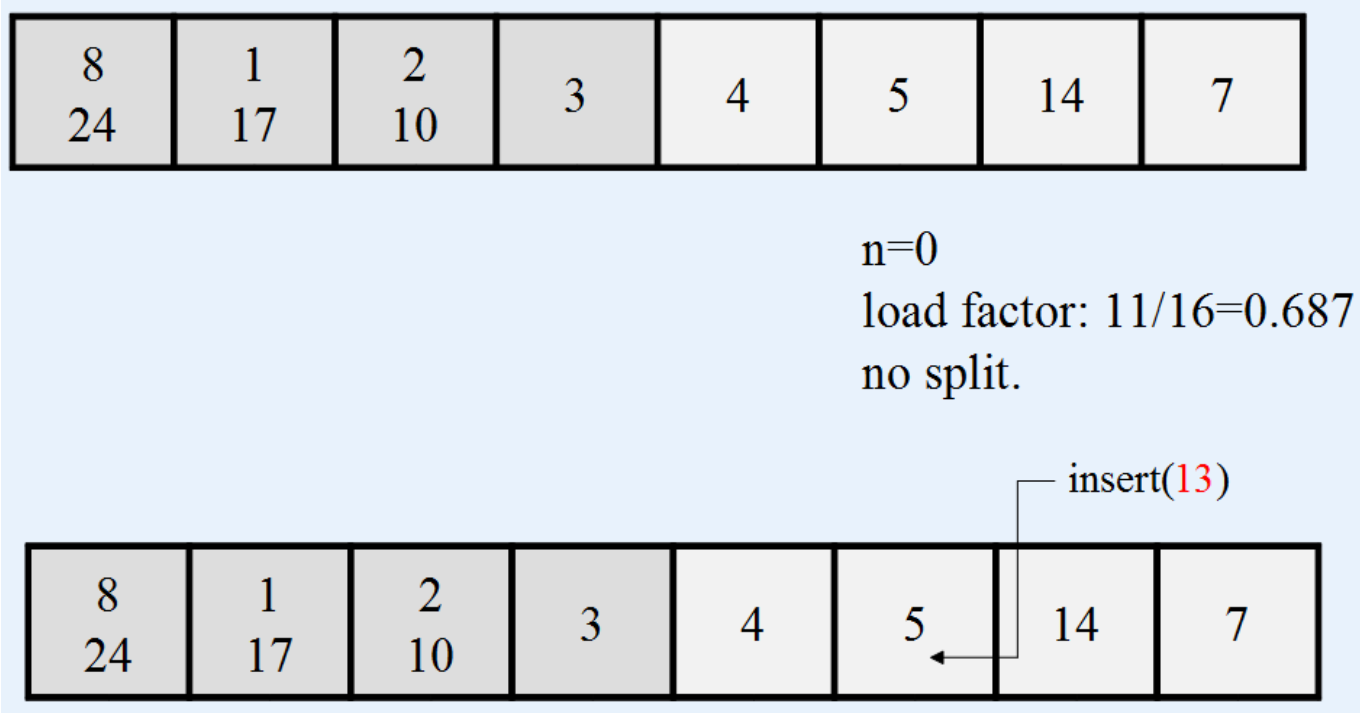
n = 0; using h_1 = Key mod 2M to insert and

h_2 = Key mod 4M to split.



Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.

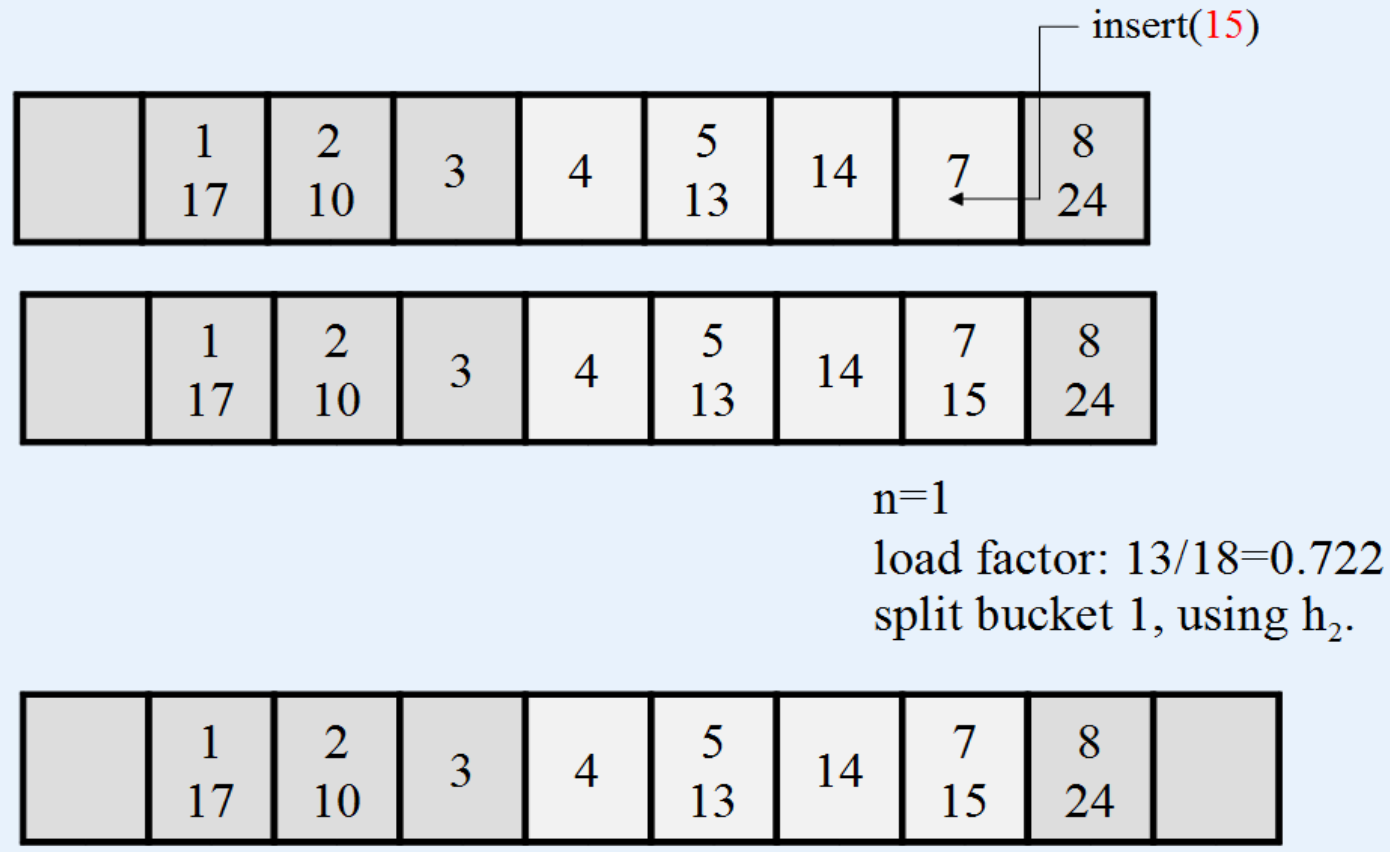
8 24	1 17	2 10	3	4	5 13	14	7

n=0
load factor: 12/16=0.75
split bucket 0, using h_2:
h_2 = Key mod 4M

	1 17	2 10	3	4	5 13	14	7	8 24

Hashing Linear

Exercício – apresente o processo de inserção da seguinte sequência de chaves: 3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15.



Hashing Linear

O fator de carga do arquivo pode ser usado para disparar divisões e combinações:

>> *buckets* que foram divididos também podem ser recombinados.

Vantagens:

>> não requer diretório;

>> mantém o fator de carga razoavelmente constante, enquanto o arquivo aumenta e diminui.

Procedimento de busca:

Algorithm 17.3. The Search Procedure for Linear Hashing

```
if n = 0
    then m ← h_j (K) (* m is the hash value of record with hash key K *)
    else begin
        m ← h_j (K);
        if m < n then m ← h_{j+1} (K)
        end;
```