Hashing e Árvore Trie

Prof. Dr. Lucas C. Ribas

Disciplina: Estrutura de Dados II

Departamento de Ciências de Computação e Estatística

unesp
UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

IBILCE / UNESP - CÂMPUS DE SÃO JOSÉ DO RIO PRETO

Estruturas de indexação

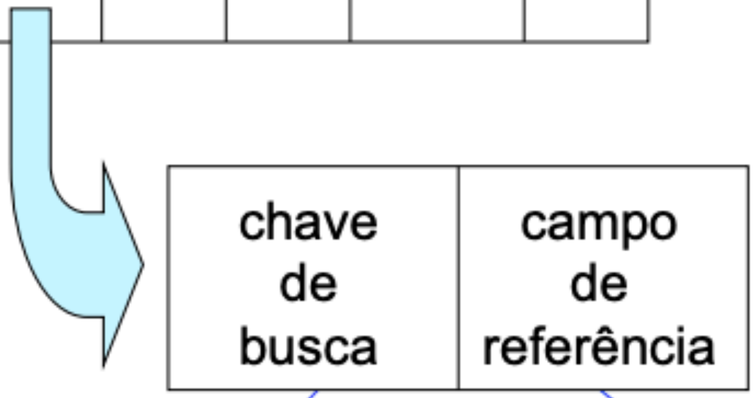
- **Índice:** estrutura de acesso auxiliar usada para melhorar o desempenho na recuperação de registros

- **Pesquisa:**
  - restringida a um subconjunto dos registros, em contrapartida à análise do conjunto completo
  - realizada em resposta a certas condições

- **Observações:**
  - existe uma variedade de índices, cada qual com uma estrutura de dados particular
  - qualquer campo em um arquivo pode ser usado para criar um índice
  - vários índices podem ser definidos para um mesmo arquivo

unesp

Índice Simples ou Linear



apesar de simples, índices proporcionam ferramentas poderosas para a recuperação de registros

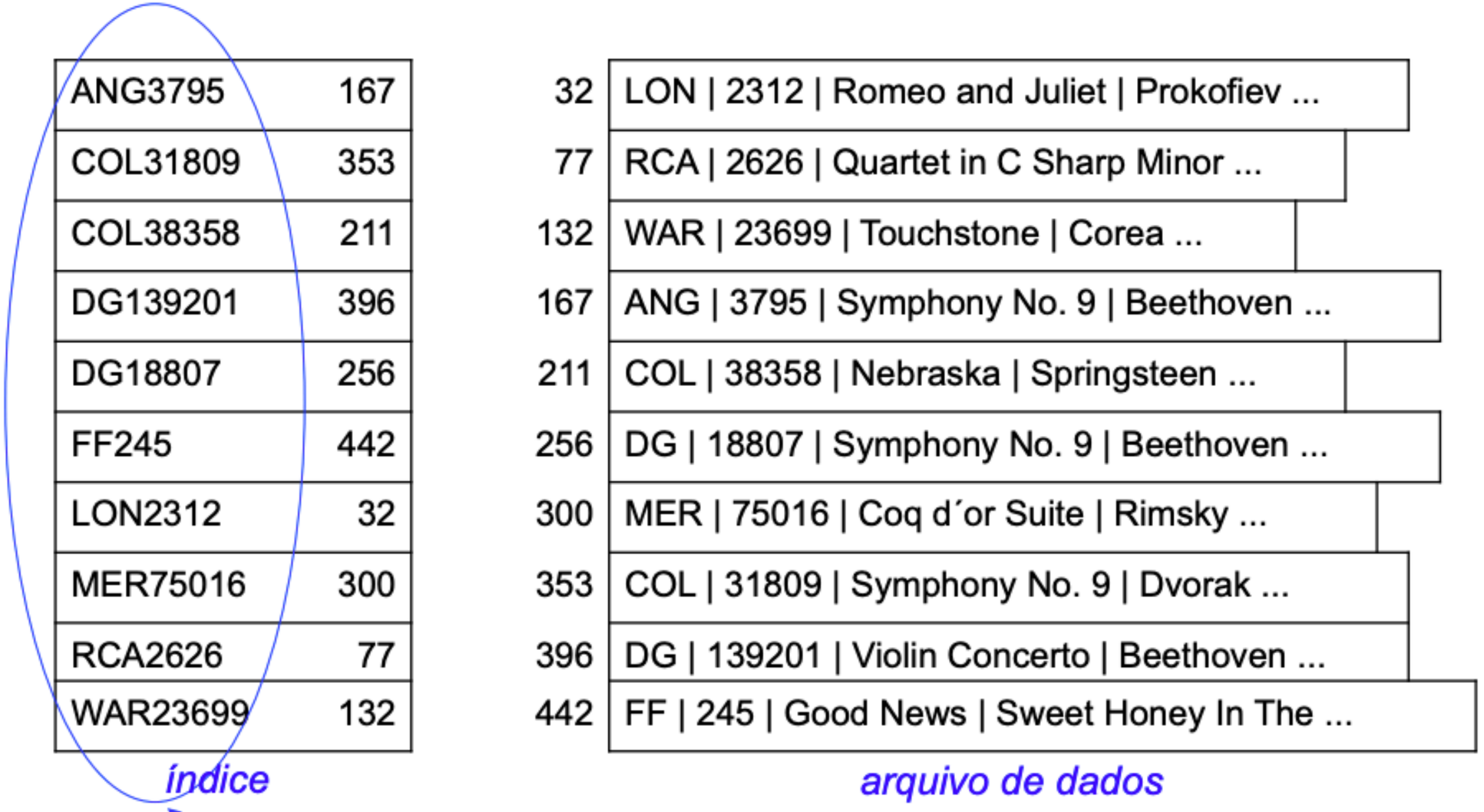
valores ordenados

número de um bloco de disco (RNN) ou endereço do registro

byte-offset = endereço do primeiro *byte* do bloco (ou registro) correspondente no arquivo de dados

Índice Simples ou Linear

Índice Simples ou Linear



Índice Simples ou Linear

ANG3795	167
COL31809	353
COL38358	211
DG139201	396
DG18807	256
FF245	442
LON2312	32
MER75016	300
RCA2626	77
WAR23699	132

índice
campos e
registros de tamanho fixo

32	LON | 2312 | Romeo and Juliet | Prokofiev ...
77	RCA | 2626 | Quartet in C Sharp Minor ...
132	WAR | 23699 | Touchstone | Corea ...
167	ANG | 3795 | Symphony No. 9 | Beethoven ...
211	COL | 38358 | Nebraska | Springsteen ...
256	DG | 18807 | Symphony No. 9 | Beethoven ...
300	MER | 75016 | Coq d´or Suite | Rimsky ...
353	COL | 31809 | Symphony No. 9 | Dvorak ...
396	DG | 139201 | Violin Concerto | Beethoven ...
442	FF | 245 | Good News | Sweet Honey In The ...

arquivo de dados
campos e
registros de tamanho fixo ou variável

unesp

Operações em um Índice

- **Pesquisa**: baseada na chave de busca. Encontra a posição da chave no índice, obtém o byte-offset ou RRN, move para o registro no arquivo de dados e recupera o registro solicitado no arquivo de dados

- **Criação:** cria um índice juntamente com a criação do arquivo de dados (apenas registro de cabeçalho) **ou** cria o índice baseado em um arquivo de dados já existente (cabeçalho e demais registros (chave de busca + campo de referência)), obtidos a partir de uma varredura no arquivo de dados

- **Inserção**: adiciona registros no índice devido às inserções no arquivo de dados. No *arquivo de dados* não ordenado é adicionado no final do arquivo, já no *arquivo de índices* é preciso manter a ordem baseado na chave

- **Remoção:** remove registros no índice devido às remoções no arquivo de dados. No arquivo de dados é **lógica** (estratégia para reaproveitamento de espaço) no arquivo de índice é **lógica** e **física** (deslocamento dos registros).

unesp

Operações em um Índice

- **Atualização:** modifica registros no índice devido às modificações no arquivo de dados. Em geral, utiliza-se remoção seguida de inserção para atualizar.

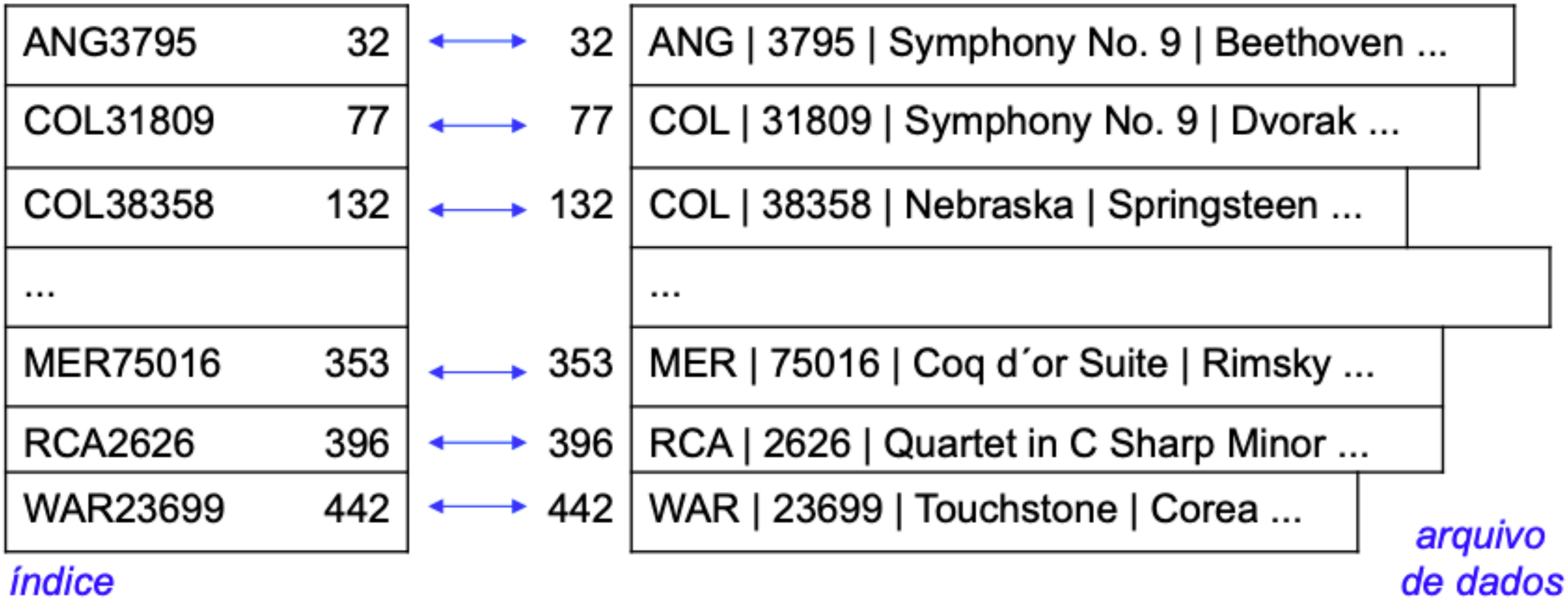
- **Carregamento:** carrega o arquivo de índice na memória principal antes de usá-lo. *Passos*: 1. aponta para o primeiro registro do arquivo de índice em disco; 2. varre o arquivo de índices sequencialmente; 3. cria o índice em memória principal, em geral implementado como um vetor.

- **Reescrita**: atualiza o arquivo de índice em disco com base no arquivo de índice em memória principal, quando necessário. Status no registro de cabeçalho (verdadeiro/falso) para inconsistência nos índices, devido à queda de energia, travamento do programa de atualização, etc.

unesp

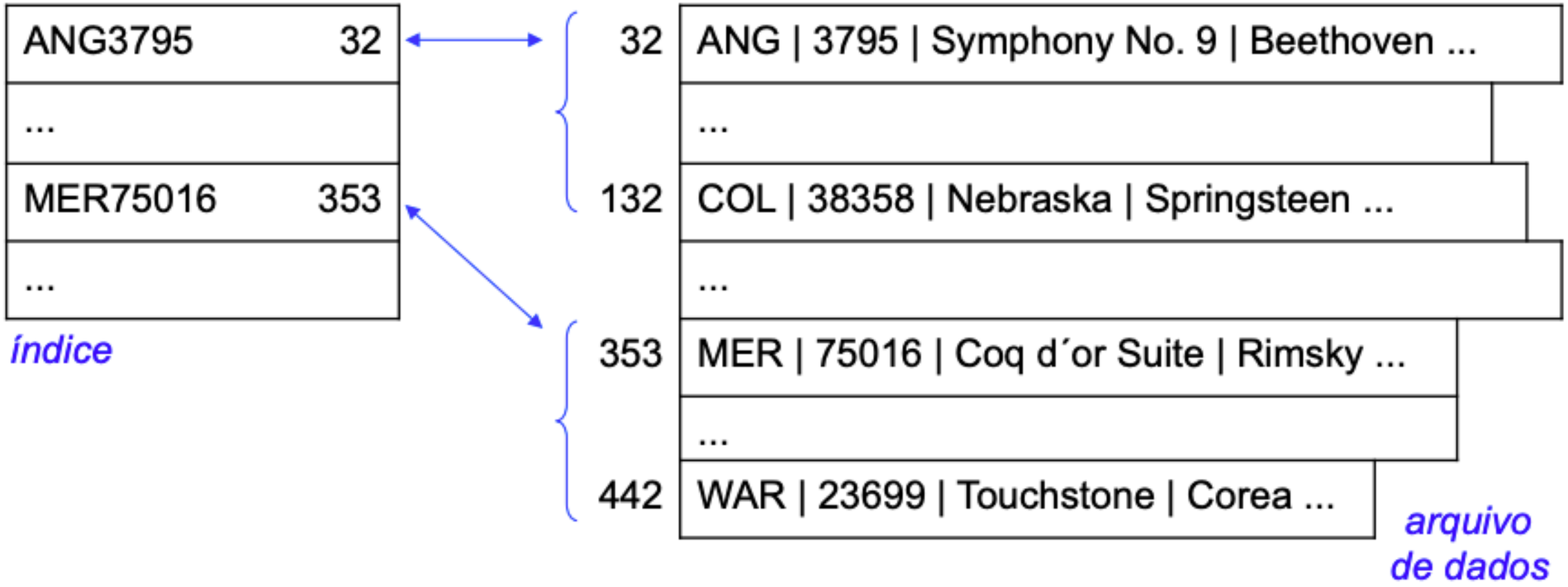
Índice denso

- Possui uma entrada no índice para cada valor de chave (i.e., cada registro) no arquivo de dados



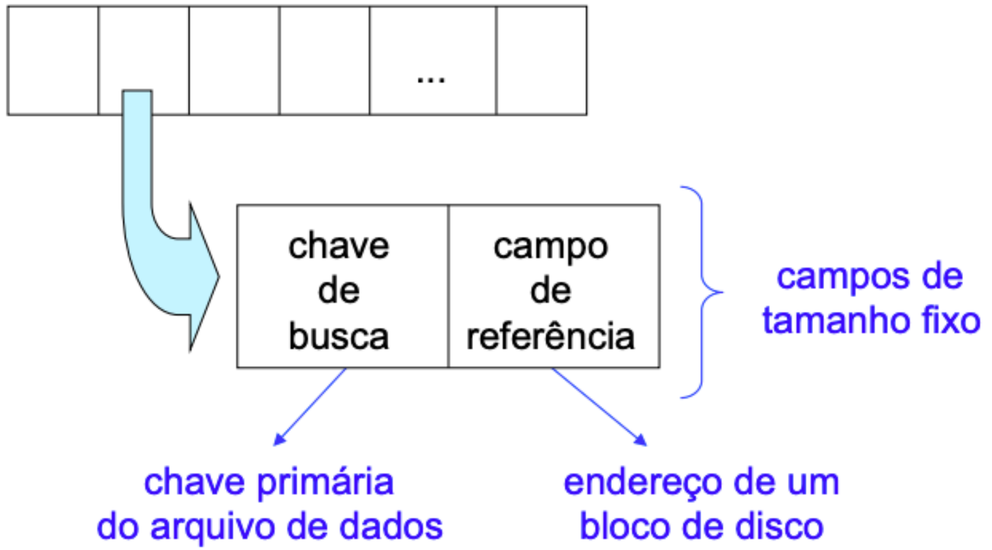
Índice esparso

- Possui uma entrada no índice para cada bloco do arquivo de dados



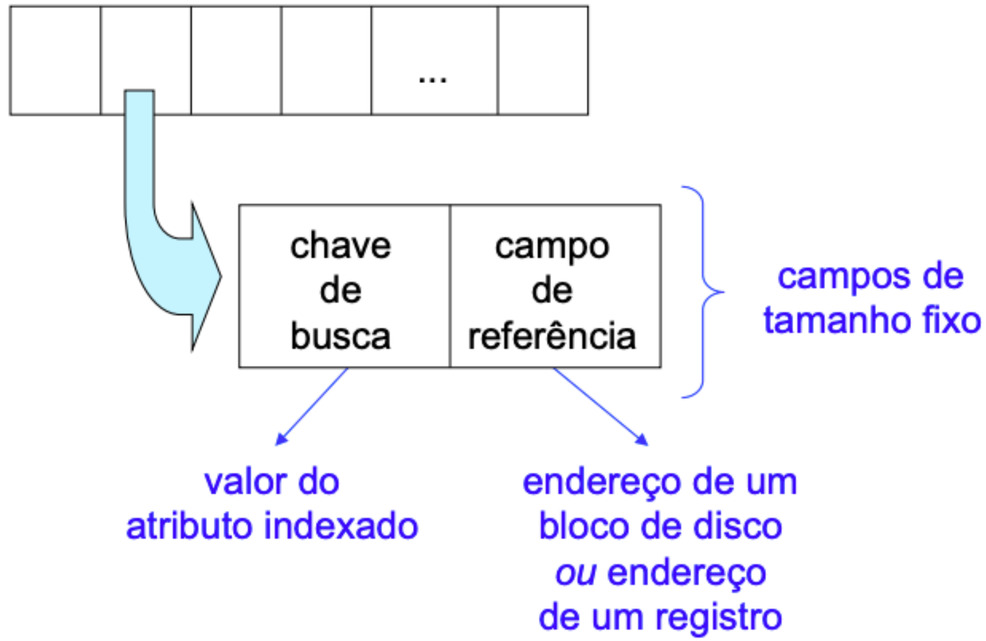
Indice primário

- **Características**
  - ordenado
  - definido com base em um arquivo de dados ordenado pela chave primária
  - possui um único nível
- **Estrutura do registro (entrada)**



Índice secundário

- **Características**
  - ordenado
  - definido sobre um atributo não ordenado do arquivo de dados
  - possui um único nível
- **Arquivo de dados**
  - em geral, desordenado
  - porém, pode estar ordenado por outro atributo que não o indexado com índice secundário

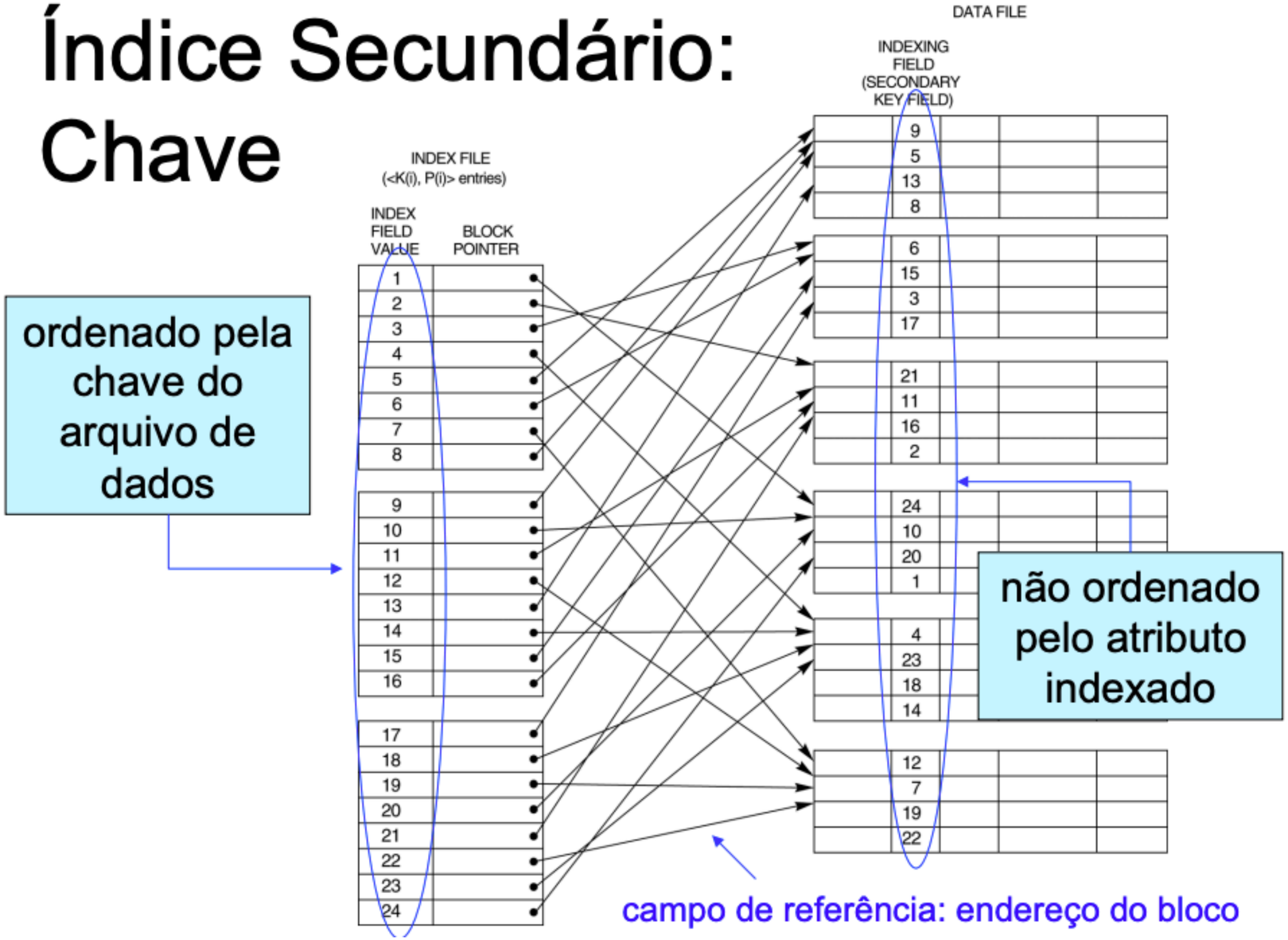


Índice secundário

- **Vantagens**
  - propicia uma ordenação lógica do arquivo de dados
  - facilita as operações de inserção e remoção em arquivos de dados desordenados
- **Pode ser definido sobre atributo**
  - chave (UNIQUE)
  - não chave

Índice secundário

Índice Secundário: Chave



Índice secundário

Tipo de Índice	Arquivo de Índice	Arquivo de Dados	Melhora no Desempenho
primário *chave primária*	busca binária $O(\log_2 b)$	busca binária $O(\log_2 b)$	discreta
secundário *chave primária*	busca binária $O(\log_2 b)$	busca linear $O(b)$	significativa

- Índice secundário
  - deve ser utilizado para pesquisas freqüentes

Hashing

Fundamentos de hashing

- **Tabela Hash**
  - Uma *tabela hash* é uma estrutura de dados que permite o acesso rápido e eficiente aos elementos com base em uma chave
  - Ela é composta por um array de "baldes" onde os dados são armazenados
  - A ideia é usar uma **função de hash** que transforma a chave em um índice no array, determinando assim onde o valor associado àquela chave será armazenado ou encontrado

unesp

Fundamentos de hashing

- **Função *Hashing***
  - Uma função de espalhamento (função *hash*) ***h(k)*** transforma uma chave **k** em um endereço
  - Este endereço é usado como a base para o armazenamento e recuperação de registros
  - É similar a uma indexação, pois **associa a chave** ao **endereço relativo do registro**
  - Uma boa função de *hashing* é aquela que distribui as chaves uniformemente pelos baldes da tabela hash, minimizando assim a probabilidade de colisões (duas chaves distintas que resultam no mesmo índice)

Fundamentos de hashing

- **Função *x* Tabela *Hashing***
  - A **Tabela Hash** é a estrutura de dados completa que permite mapear chaves a valores e realiza operações de inserção, busca e remoção de elementos baseado em chaves
  - As **Funções de Hashing** são o mecanismo que as tabelas hash usam para determinar onde cada valor deve ser armazenado ou recuperado, com base em sua chave
  - Em resumo, a **função de hash** é um componente da **tabela hash**, responsável por determinar rapidamente onde um valor específico é armazenado na estrutura

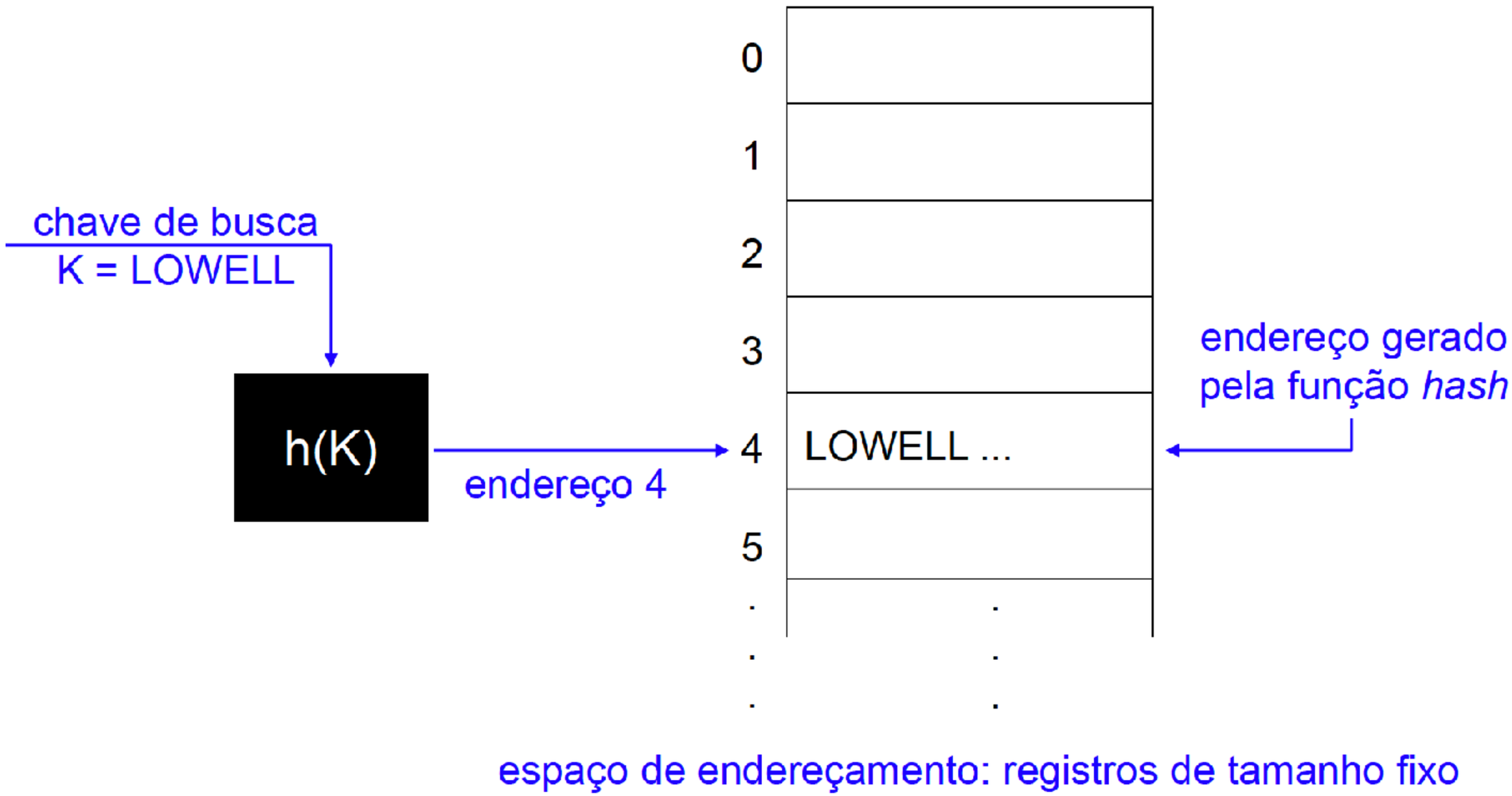
Hashing interno x externo

- **Hashing interno:**
  - *Hashing* em memória principal
  - Cada slot da tabela hash é um registro
  - Colisões:
    - em lista ligada (endereçamento fechado = *hashing* aberto): quando uma colisão ocorre os elementos colididos são simplesmente adicionados à lista ligada no respectivo slot
    - em outro slot (endereçamento aberto = *hashing* fechado): quando uma colisão ocorre, a estratégia é procurar outro slot aberto na tabela para armazenar o elemento colidido. Várias estratégias podem ser usadas para encontrar um slot aberto, incluindo sondagem linear, sondagem quadrática e duplo hash.

Hashing interno x externo

- **Hashing externo:**
  - *hashing* em memória secundária (armazenamento e recuperação em disco)
  - Cada slot da tabela de *hash* é um *bucket* (um bloco ou *cluster* de blocos em disco)
  - Colisões vão preenchendo o *bucket*
  - Tabela de *hash* fica no cabeçalho do arquivo

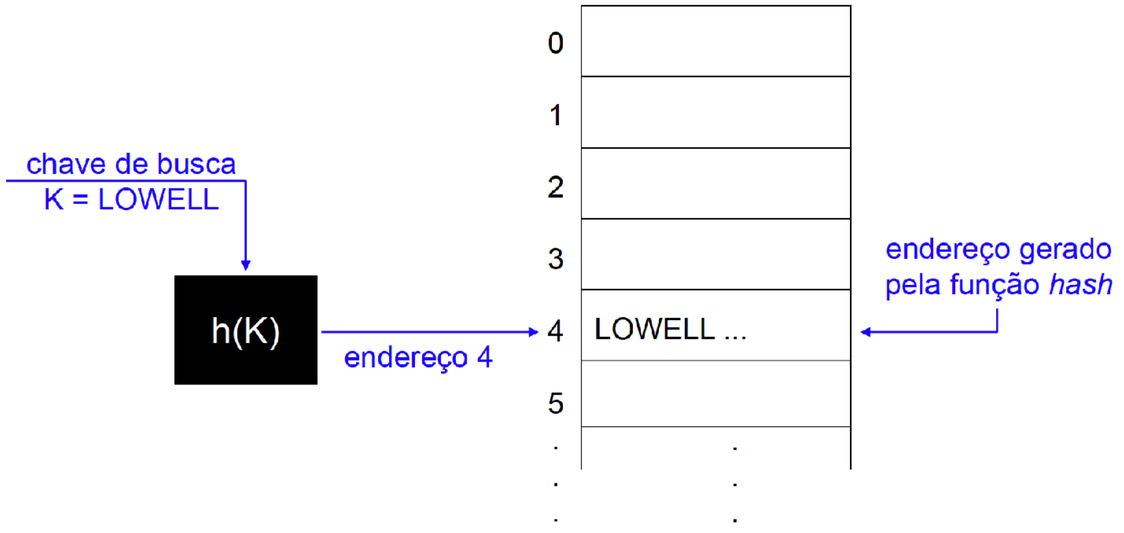
Hashing


chave de busca
K = LOWELL
h(K)
endereço 4
0
1
2
3
4
LOWELL ...
5
endereço gerado
pela função *hash*
espaço de endereçamento: registros de tamanho fixo

Hashing: Exemplo

- Suponha que foi reservado espaço para manter 1.000 registros e considere a seguinte h(K):
  - Obter as representações ASCII dos dois primeiros caracteres do sobrenome;
  - multiplicar estes números e usar os três dígitos menos significativos do resultado para servir de endereço

Name	Código ASCII para as 2 primeiras letras	Produto	Endereço
BALL	66 65	66 X 65 = 4.290	290
LOWELL	76 96	76 X 96 = 6.004	004
TREE	84 82	84 X 82 = 6.888	888



Hashing

Função hash

- caixa preta que produz um endereço toda vez que uma chave de busca é passada como parâmetro

Endereço resultante

- usado para armazenamento e recuperação de registros no arquivo de dados

Nomenclatura

- *h(K) -> endereço*
  - K: chave de busca

unesp

Hashing & Indexação

- **Semelhança**
  - ambos envolvem associação de uma chave de busca a um endereço de registro
- **Diferenças (*hashing*)**
  - Endereço gerado é (teoricamente) aleatório - não existe conexão óbvia entre a chave e o endereço, apesar da chave ser utilizada no cálculo do endereço
  - no espalhamento duas chaves diferentes podem levar ao mesmo endereço (**colisão**) – portanto as colisões devem ser tratadas.

unesp

Exemplo de Colisão

nome	código ASC II 1ª e 2ª letras	produto	endereço gerado
BALL	66 65	66 x 65 = 4.290	290
LOWELL	76 79	76 x 79 = 6.004	004
TREE	84 82	84 x 82 = 6.888	888
OLIVER			

Exemplo de Colisão

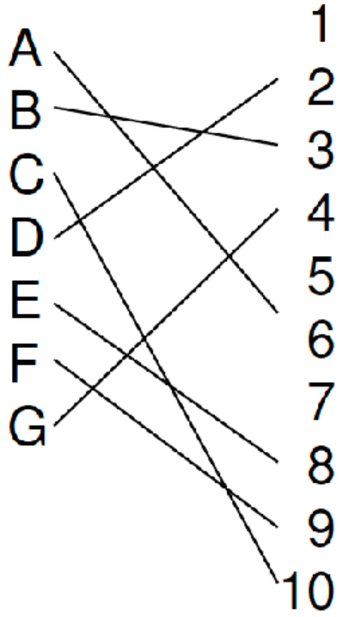
nome	código ASC II 1ª e 2ª letras	produto	endereço gerado
BALL	66 65	66 x 65 = 4.290	290
LOWELL	76 79	76 x 79 = 6.004	004
TREE	84 82	84 x 82 = 6.888	888
OLIVER	79 76	79 x 76 = 6.004	004

chaves sinônimas: LOWELL e OLIVER

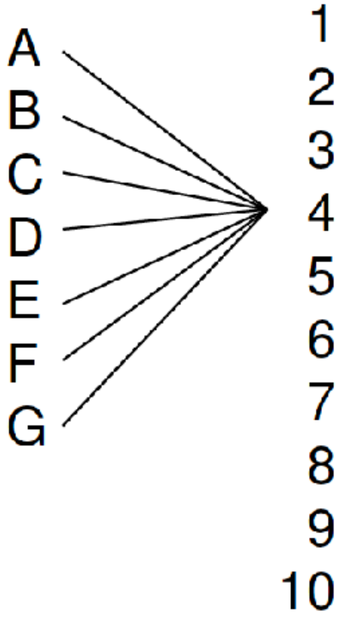
Distribuição de Registros

- Como uma função *hash* distribui (espalha) os registros no espaço de endereços?

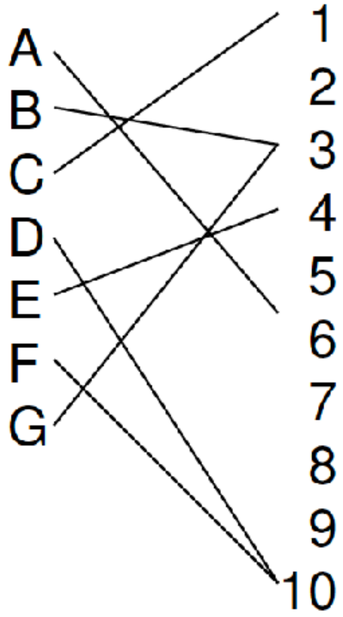
(a) melhor caso



(b) pior caso



(c) caso aceitável



7 registros x 10 endereços

Distribuição Uniforme x Aleatória

- Uniforme: registros espalhados uniformemente entre os endereços
  - Características
    - pouca ou nenhuma colisão
    - muito difícil de ser obtida
- Aleatória: os registros são espalhados no espaço de endereços com algumas colisões
  - para uma certa chave, todos os endereços possuem a mesma probabilidade de serem escolhidos

unesp

Alguns Métodos de *Hash*

- **Pegar o resto da divisão da chave pelo tamanho do espaço disponível:**
  - Este é um dos métodos mais comuns e simples. Dado um número inteiro *k* (a chave) e uma tabela de hash de tamanho *m*, o índice é calculado como *k* ***mod*** *m*. Isso garante que o índice sempre esteja no intervalo de *0* a *m*-1.
  - Exemplo: Se temos uma chave *k = 105* e um tamanho de tabela *m = 10*, o índice será 105 ***mod*** 10 = 5.
- **Examinar as chaves em busca de um padrão:**
  - Este método procura por padrões específicos nas chaves, como certos dígitos ou caracteres, e usa esses padrões para calcular o índice. Por exemplo, se as chaves são números de telefone, podemos usar os últimos 4 dígitos como índice.

Alguns Métodos de *Hash*

- **Segmentar a chave em diversos pedaços e depois fundir os pedaços:**
  - Este método divide a chave em diferentes segmentos, realiza operações independentes em cada segmento e depois combina os resultados para obter o índice final.
  - Por exemplo, se a chave é 123456, podemos dividir em 123 e 456, somar os dois 123 + 456 = 579 e então aplicar a operação de módulo com o tamanho da tabela.
- **Dividir a chave por um número:**
  - Este método divide a chave por um número selecionado (geralmente um número primo para obter uma distribuição mais uniforme) e usa a parte fracionária da divisão como índice.
  - Por exemplo, se a chave é $k = 100$ e o número escolhido é $p = 31$, o índice pode ser calculado como (k/p) ***mod*** m.

Alguns Métodos de *Hash*

- **Elevar a chave ao quadrado e pegar o meio:**
  - Este método envolve elevar a chave ao quadrado e, em seguida, extrair um conjunto de dígitos do meio do resultado. Esses dígitos do meio são então usados como o índice na tabela de *hash*. Este método é baseado na ideia de que os dígitos do meio de quadrados de números têm uma boa chance de serem aleatórios.

- **Transformar a base:**
  - Este método envolve a interpretação da chave como um número em uma base *b* diferente da base 10. A chave transformada é então usada para calcular o índice. Por exemplo, se temos uma chave $k = 19$ na base 10, podemos transformá-la para a base 5 (19 ***mod*** 5 = 3 e depois 3 ***mod*** 5), obtendo 34. Então, aplicamos a operação de módulo com o tamanho da tabela.

unesp

Colisão: Solução 1

- Encontrar um algoritmo de *hashing* perfeito que não produza colisões
- Cenário de uso
  - conjunto de dados pequenos e estáveis
- Limitação
  - abordagem não indicada para determinadas configurações de número de chaves e de dinâmica dos dados

unesp

Colisão: Solução 2

- Encontrar um algoritmo de *hashing* que produza poucas colisões
- Objetivo
  - evitar o agrupamento de registros em certos endereços
- Funcionalidade
  - espalhar os registros aleatoriamente no espaço disponível para armazenamento
  - distribuir o mais uniformemente possível

Colisão: Solução 3

- Ajustar a forma de armazenamento dos registros
- Possibilidade 1: usar memória extra
  - aumentar o espaço de endereçamento, para um mesmo conjunto de registros
  - cenário de uso
    - poucos registros para serem distribuídos entre muitos endereços
- É muito fácil encontrar um algoritmo hash que evita colisões se existem poucos registros para serem distribuídos entre muitos endereços
- É muito mais difícil encontrar um algoritmo hash que evita colisões quando o número de registros e de endereços é aproximadamente o mesmo

unesp

Colisão: Solução 3

- Possibilidade 1: uso de memória extra
  - complexidade de espaço
    - perda de espaço de armazenamento
- **Exemplo**
  - registros: 75
  - espaço de endereçamento: 1.000
  - alocado= 7,5%
  - não usado = 92,5%

unesp

Colisão: Solução 3

Possibilidade 2: armazenar mais de um registro em um único endereço

- uso de buckets (cestos) -> técnica de blocagem
  - cada endereço é suficientemente grande para armazenar diversos registros -> mais registros de uma vez, menos seeks
- exemplo
  - registros de 80 bytes
  - bucket de 512 bytes

cada endereço pode armazenar até 6 registros!

- complexidade de espaço
  - perda de espaço para registros sem sinônimos

unesp

Colisão: Solução 3

- Para as duas possibilidades, soluções convencionais
  - *Overflow* progressivo
  - *Hashing* duplo
  - Encadeamento

unesp

Categorias de hashing

- Hashing estático: garante acesso O(1), para arquivos estáticos
  - Organização do arquivo pode deteriorar se houver muitas inserções e remoções
- Hashing “dinâmico”/não estático: extensão do hashing estático para tratar arquivos dinâmicos, ou seja, que sofrem muitas inserções e remoções de registros
  - Muitos tipos semelhantes
    - Extensível, dinâmico, linear, etc.

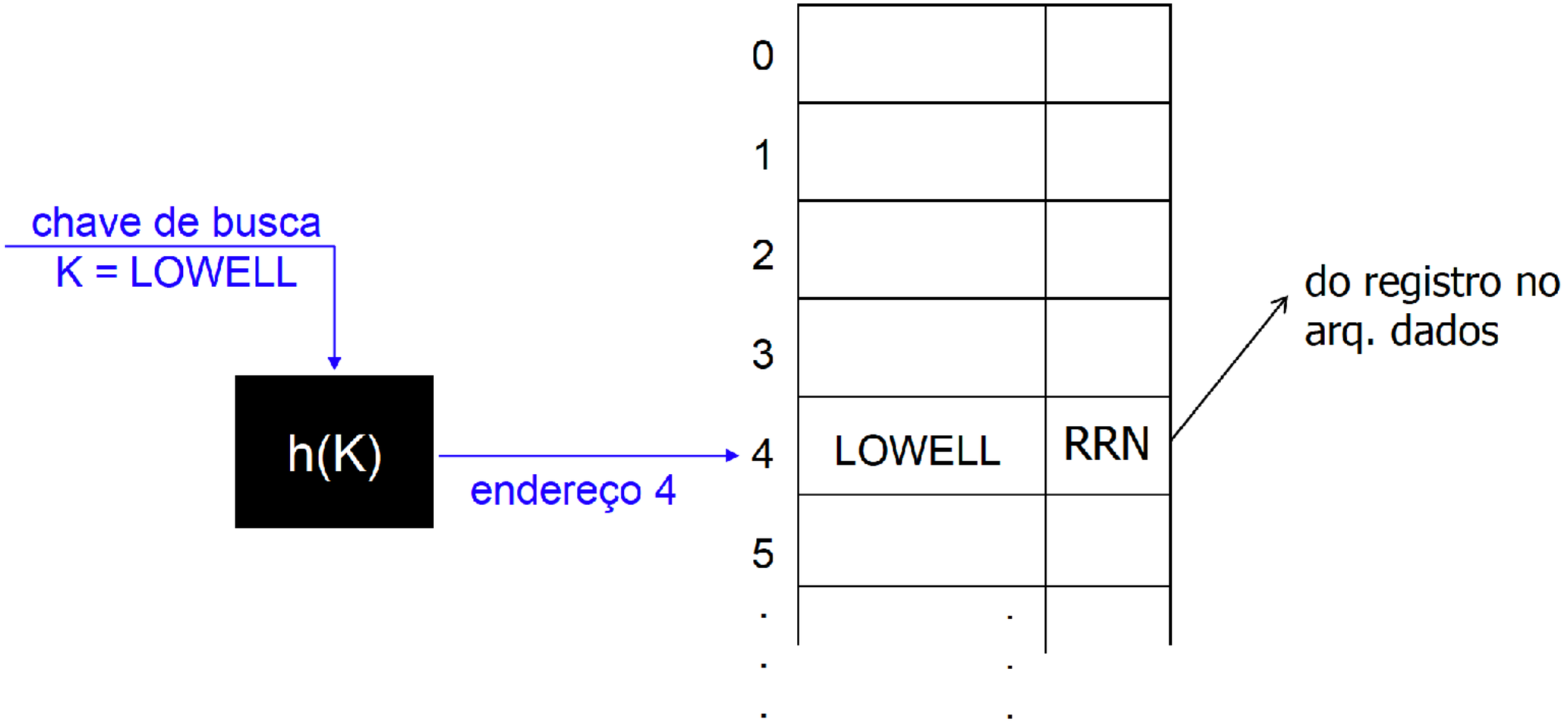
unesp

Organização de índices hashing

- único arquivo
  - Os **dados** e o **índice (tabela)** *hashing* ficam no mesmo arquivo
- dois arquivos
  - Os dados ficam em um arquivo e o índice (tabela) hashing das chaves fica em outro

unesp

Hashing Externo



Possibilidades

- Índice cabe em RAM
  - Compreendendo tabela + espaço de tratamento de overflow (*buckets*, p.ex.)
  - ***Seek*** ocorre após obtenção do RRN em RAM;
  - **O(1)**
- Índice não cabe na RAM
  - Como então gerar endereços fixos de tabela???

Problemas

- Se arquivo de dados é estático (não muda após as inserções):
  - Basta encontrar a melhor função hash e o espaço de endereçamento que mais espalha as chaves
- Se o arquivo é dinâmico
  - O espaço de endereçamento inicialmente adequado pode ser insuficiente após algumas operações de inserção

Árvore Trie

Tries

- Uma **trie** (também conhecida como *radix searching tree*) é uma árvore de busca na qual o fator de sub-divisão, ou número máximo de filhos por nó, é igual ao número de símbolos do alfabeto.
  - Chaves são colocadas em cestos, que são partes independentes de um arquivo em disco
  - Chaves tendo um endereço hashing com o **mesmo prefixo** compartilham o **mesmo cesto**
- Tries são utilizadas para acesso rápido aos cestos. Utiliza-se um prefixo do endereço hashing para localizar o cesto desejado

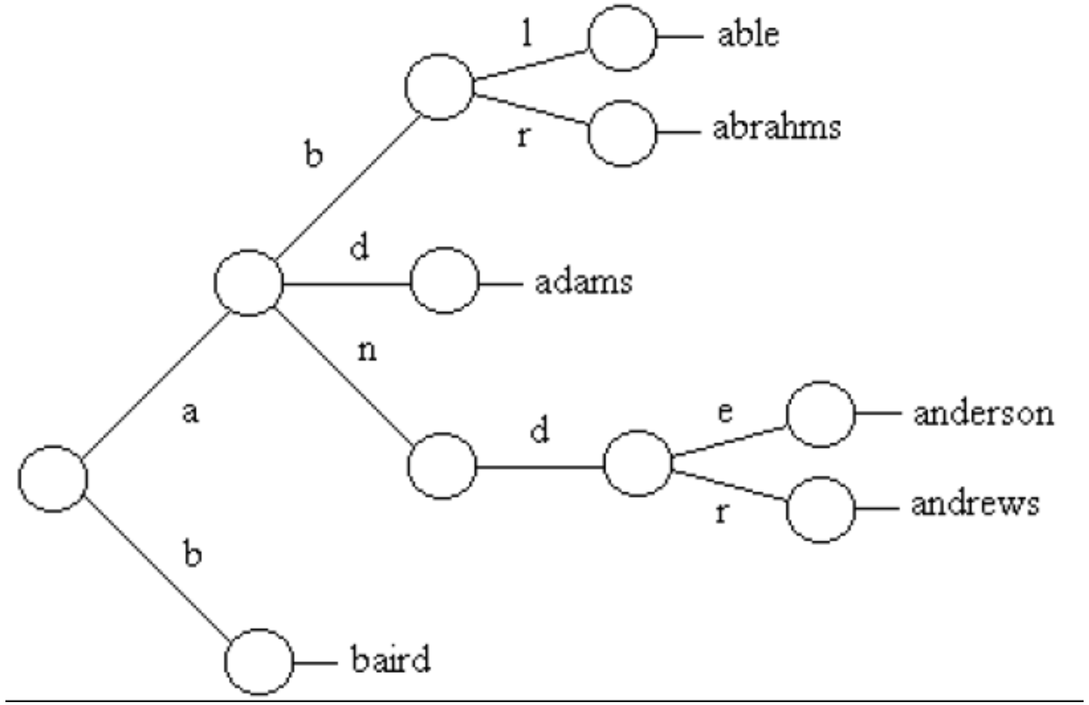
Tries

- Tempo de busca proporcional ao tamanho das chaves
- Chaves com **sufixos comuns compartilham caminho até a raiz**
- Propícias para compactação no caso de letras do alfabeto
- boa opção para manter chaves grandes e de tamanho variável

Tries Exemplo

- Suponha que desejamos construir uma trie para armazenar as chaves: *able, abrahms, adams, anderson, andrews, baird*

- A idéia é que a busca prossegue letra por letra ao longo da chave. Como existem 26 letras no alfabeto, tem-se uma árvore com ordem 26 - o número potencial de subdivisões a partir de cada nó é 26



Observe que, na busca em uma trie, é possível que precisemos usar apenas parte da chave.

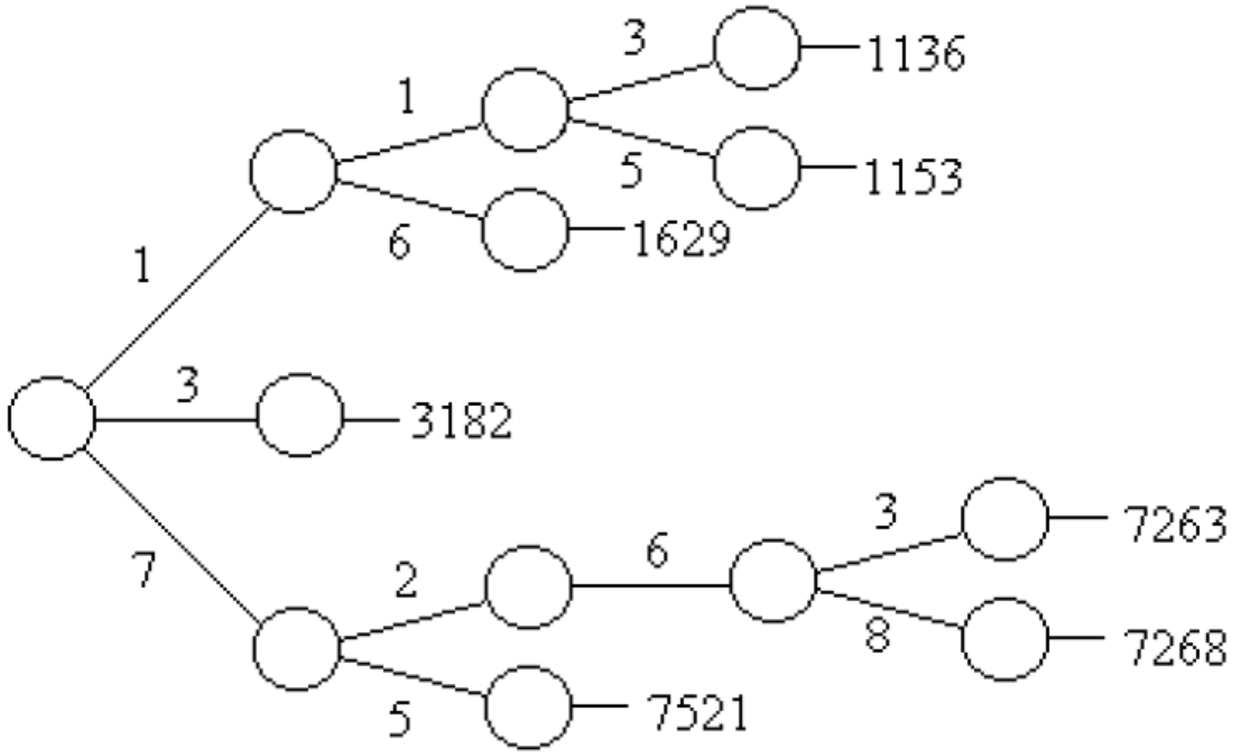
Na verdade, usamos apenas a informação necessária para identificá-la.

Tries Exemplo

- Exercício: apresente uma estrutura Trie para o seguinte conjunto de chaves:1136; 1153; 3182; 7263; 7268; 1629; 7521

Tries Exemplo

- Exercício: apresente uma estrutura Trie para o seguinte conjunto de chaves:1136; 1153; 3182; 7263; 7268; 1629; 7521



Hashing Extensível

Hashing Extensível

- Espalhamento convencional: pouco adequado a arquivos dinâmicos, que crescem e diminuem com o tempo
- **Espalhamento Extensível** (*Extendible Hashing*): permite um auto-ajuste do espaço de endereçamento do espalhamento
  - Maior o número de chaves, maior o número de endereços
- Ideia chave é combinar o espalhamento convencional com uma técnica de recuperação de informações denominada **trie**

unesp

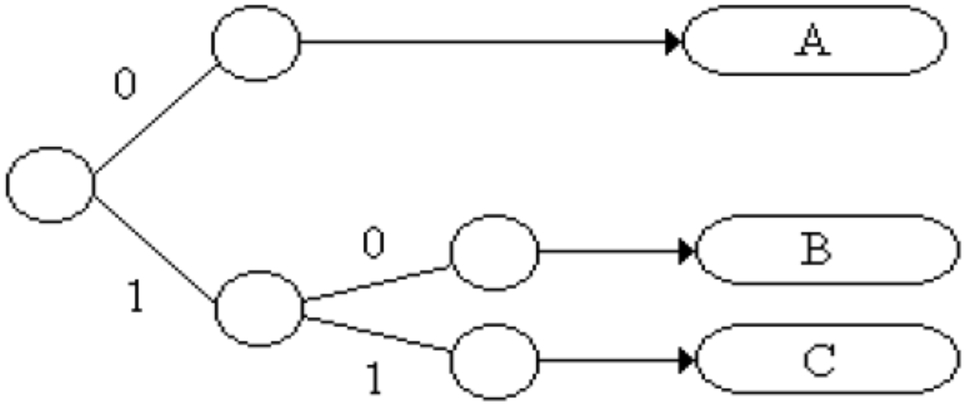
Hashing Extensível e Tries

- O espalhamento extensível usa tries de **ordem 2** como índices
- Tabela de espalhamento indexa um conjunto de cestos (buckets = cestos)
- Função *Hash* gera um endereço binário
- Conjuntos de chaves (ou registros) são armazenados em cestos
- **Busca** por chave: análise bit-a-bit do valor de *h(key)* permite localizar o seu cesto
  - como estamos recuperando informações da memória secundária, trabalha-se com cestos contendo várias chaves, e não com chaves individuais.

unesp

Hashing Extensível e Tries

- **Níveis internos:** rotulados com bits
- **Nível dos nós folha:** *buckets* contendo várias chaves ou registros
  - Ex.: no bucket A, há chaves de endereço começando com 0; em B, endereços começando com 10; em C, com 11 (não foi possível colocar todas chaves de endereços começando com 1 num único bucket).



Bucket (Cesto)

- Segmento físico útil de armazenamento externo
  - página, trilha ou segmento de trilha
- Análogo aos blocos da Árvore-B+, porém não são ordenados.

Transformando a trie em um diretório

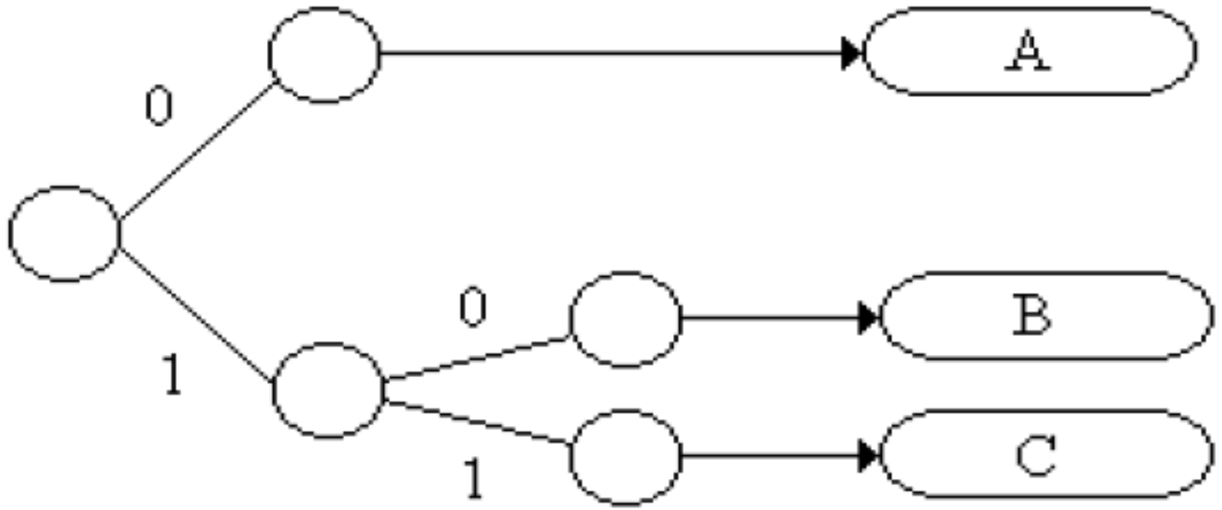
Como representar a trie?

- Se for mantida como uma árvore, será necessário fazer várias comparações para descer ao longo de sua estrutura.
- Assim, o que se faz é transformar a trie em um vetor de registros consecutivos, formando um diretório de endereços de espalhamento e ponteiros para os cestos associados.

unesp

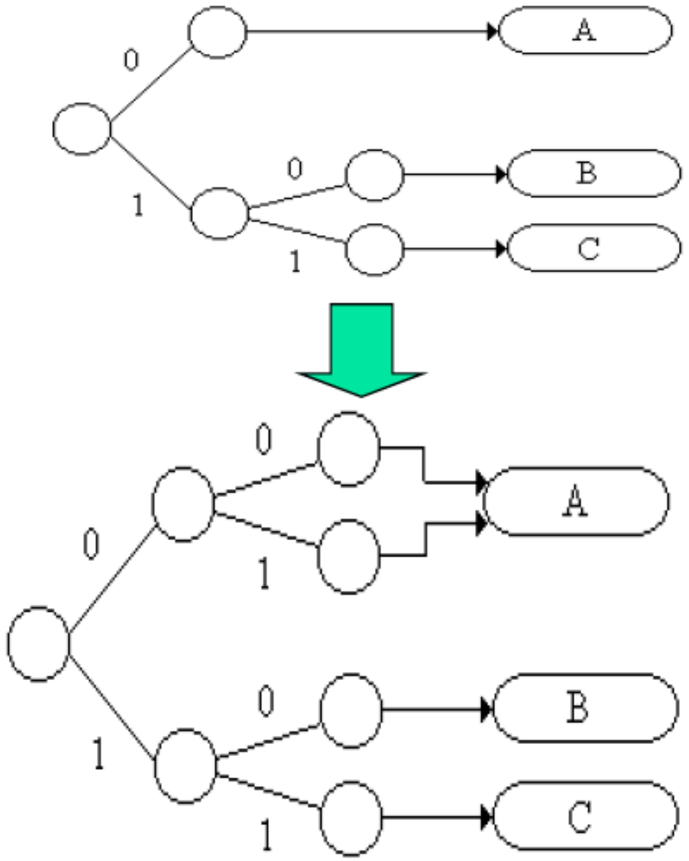
Transformando a trie em um diretório

- Exemplo: Considere cestos contendo chaves com endereços hash com prefixos:
  - cesto A com chaves que tem endereço hashing que começam com o bit 0
  - cesto B com chaves que tem endereço hashing que começam com os bits 10
  - cesto C com chaves que tem endereço hashing que começam com os bits 11



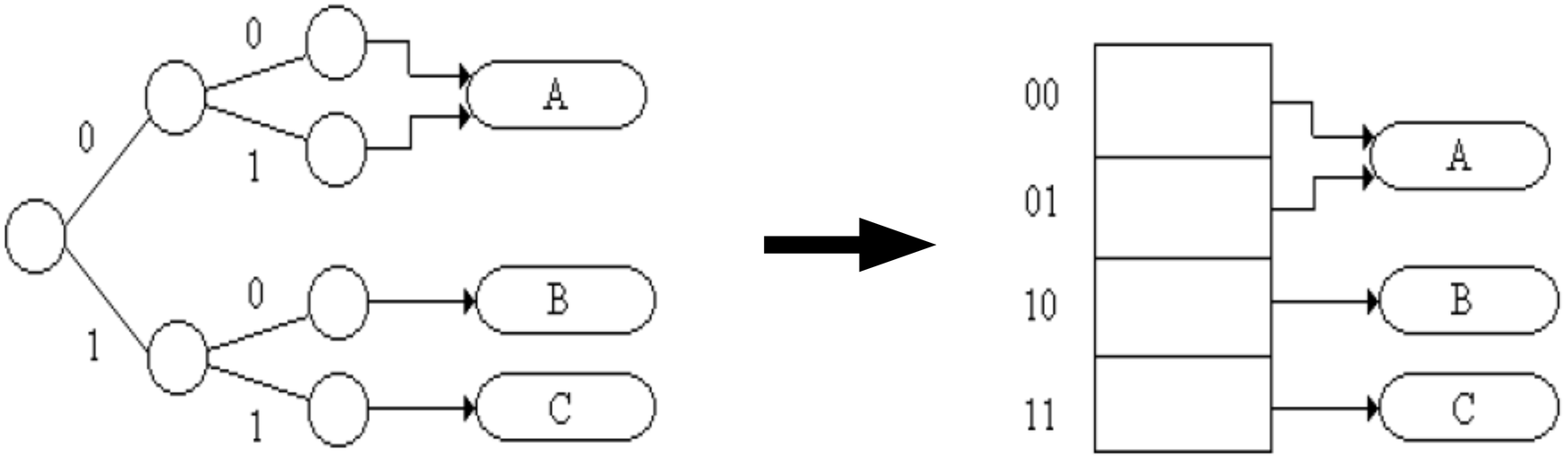
Transformando a trie em um diretório

- O primeiro passo para **transformar a trie** em um diretório é estendê-la de modo a que se torne uma **árvore binária completa** - na qual todas as folhas estão no mesmo nível.



Transformando a trie em um diretório

- Segundo passo: uma vez estando "completa", a trie pode ser representada pelo vetor mostrado na figura abaixo.
  - Agora temos uma **estrutura (vetor)** que fornece o **acesso direto (aos endereços) associado** ao processo de espalhamento:
    - **Exemplo**: dado um endereço começando com os bits 10, o diretório na posição 10 (base 2) = 2 do vetor nos dá um ponteiro para o cesto associado.



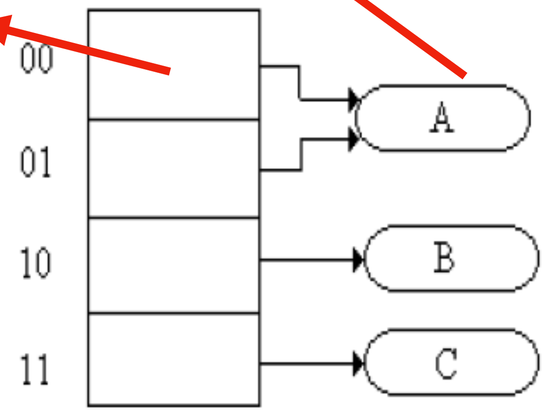
Estrutura de dados do Diretório e Buckets

Record Type: Bucket

Depth	integer count of the number of bits usede “in common” by the keys in this bucket
Count	integer count of the number of keys in the bucket
Key [ ]	array [1..Max_Bucket_Size] of strings to hold keys

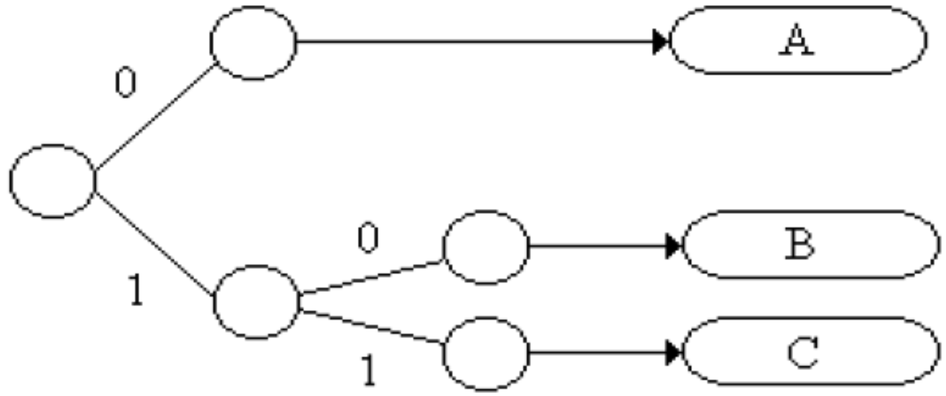
Record Type: Directory_Cell

Bucket_Ref	relative record number (RRN) or other reference to a specific **Bucket** record on disk



Profundidade do Diretório

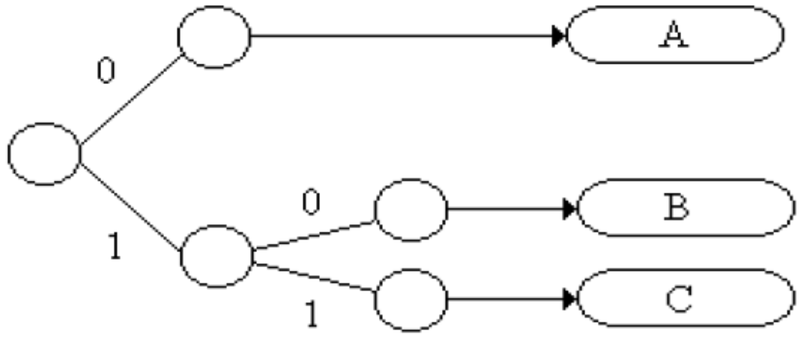
- **Profundidade do cesto**: indicação do número de bits da chave necessários para determinar quais registros ele contém
  - Quanto mais ponteiros do diretório para o cesto, menor sua profundidade
  - Essa informação pode ser mantida junto ao cesto
- No exemplo:
  - Cesto A: profundidade 1
  - Cestos B e C: profundidade 2



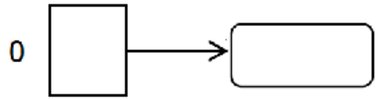
Profundidade do Diretório

- Maior profundidade dos cestos, ou número de bits necessários para distinguir os cestos, ou número de bits dos endereços (binários) do diretório, ou $\mathbf{log_2 E}$, onde E é o tamanho do diretório

- No exemplo: Profundidade do diretório = 2



- Inicialmente, a profundidade do diretório é 0 e há um único bucket

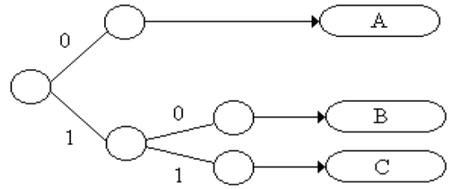


Hashing Extensível - Busca por uma chave

- Calcular o endereço *hash* para a chave => o tamanho da tabela não é especificado, portanto não utilizar *mod*
- Verificar quantos bits são utilizados no diretório. Seja **i** o número de bits
- Pegar os **i** bits menos significativos do endereço hash (em ordem inversa) => esse é o índice do diretório
- Utilizar esse índice para ir ao diretório e encontrar o endereço do cesto onde o registro deve estar
- Como essa abordagem pode ser dinâmica expandindo e encolhendo conforme os registros vão sendo incluídos ou removidos? Os aspectos dinâmicos são gerenciados por dois mecanismos:
  - **Inserção e divisão de cestos**
  - **Remoção e combinação de cestos**

unesp

Hashing Extensível - Inserção



Divisão de cestos para tratar *overflow*

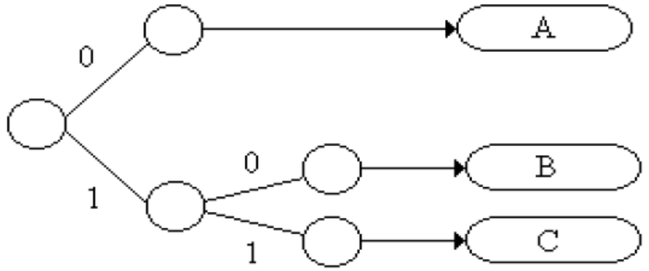
- Se um registro precisa ser inserido e não existe espaço no cesto que é o seu endereço base, o cesto é sub-dividido (*spliting*). Utiliza-se um **bit adicional dos valores de espalhamento das chaves** para dividir os registros entre dois cestos.
  - Distribui-se o conjunto de chaves do bucket cheio de modo que o próximo bit do endereço seja distinto nos 2 buckets
  - Ver exemplo anterior, buckets B (10) e C (11)
- se o novo espaço de endereçamento resultante da consideração desse novo bit já estava previsto no diretório, nenhuma alteração adicional se faz necessária.
- senão, é necessário dobrar o espaço de endereçamento do diretório para acomodar o novo bit.

Hashing Extensível - Inserção

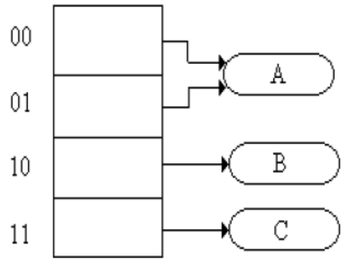
- Seja ***d*** a profundidade do índice (diretório), dada pela maior profundidade dos cestos
- Localiza chave no diretório: seja ***i*** a profundidade do seu cesto
- Se a inserção da chave provoca a sub-divisão do cesto, existem 2 casos a considerar: ***i < d*** e ***i = d***

Hashing Extensível - Inserção

- **Se i < d** (Ex. Bucket A: há endereço disponível no diretório para um novo bucket)
  - Remaneja os registros entre os 2 cestos
  - Insere nova chave no cesto adequado
  - Altera a profundidade de ambos os cestos



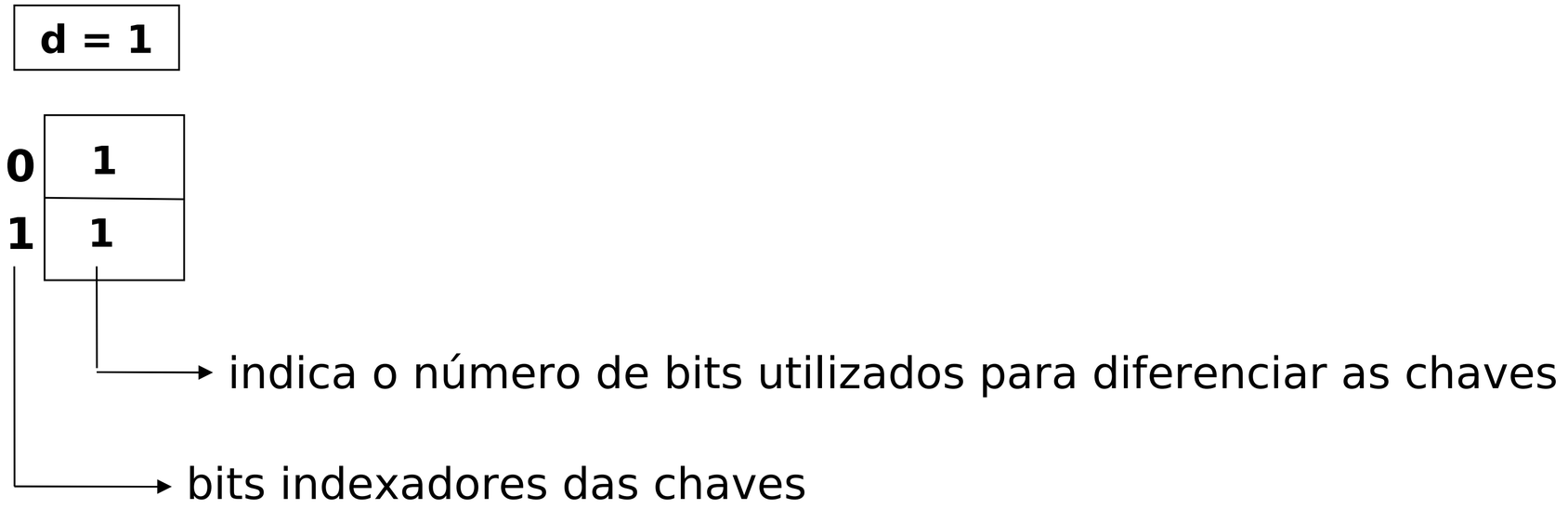
- **Se i = d** (Ex. Bucket B: já usa todos os bits possíveis)
  - É necessário dobrar o tamanho do diretório
  - Profundidade do índice passa a ser **d + 1**, assim como a dos cestos envolvidos na inserção
  - Distribui-se o conjunto de chaves do cesto cheio de modo que o próximo bit do endereço seja distinto nos 2 cestos
  - Antigo conteúdo de todas as demais posições do índice copiado para o novo índice



Hashing Extensível - Inserção

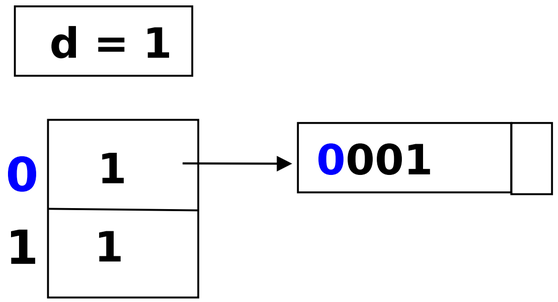
- *Hashing* extensível: inicialmente, tabela vazia
  - m = 4 (bits), N = 2 (número de chaves por cesto)

(4 bits menos significativos)



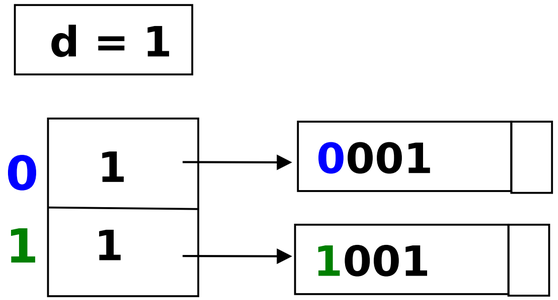
Hashing Extensível - Inserção

- *Hashing* extensível: inserção do elemento 0001
  - m = 4 (bits), N = 2



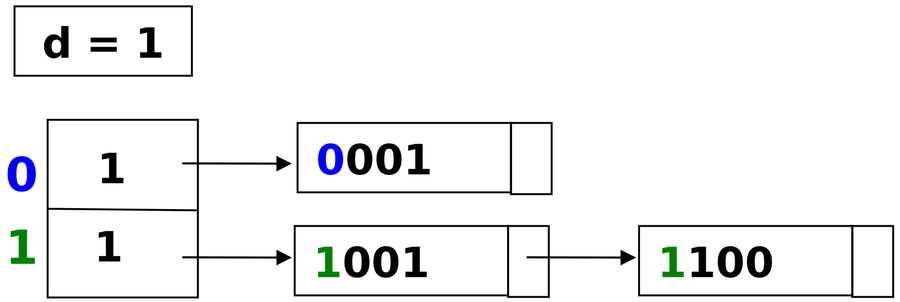
Hashing Extensível - Inserção

- *Hashing* extensível: inserção do elemento 1001
  - m = 4 (bits), N = 2



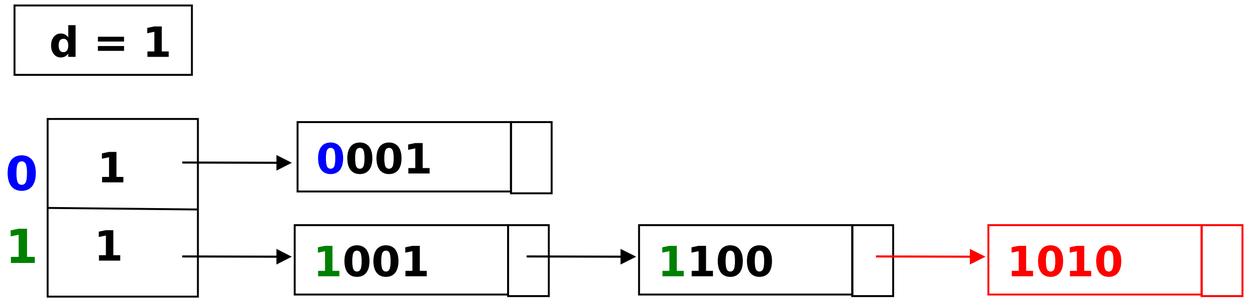
Hashing Extensível - Inserção

- *Hashing* extensível: inserção do elemento 1100
  - m = 4 (bits), N = 2



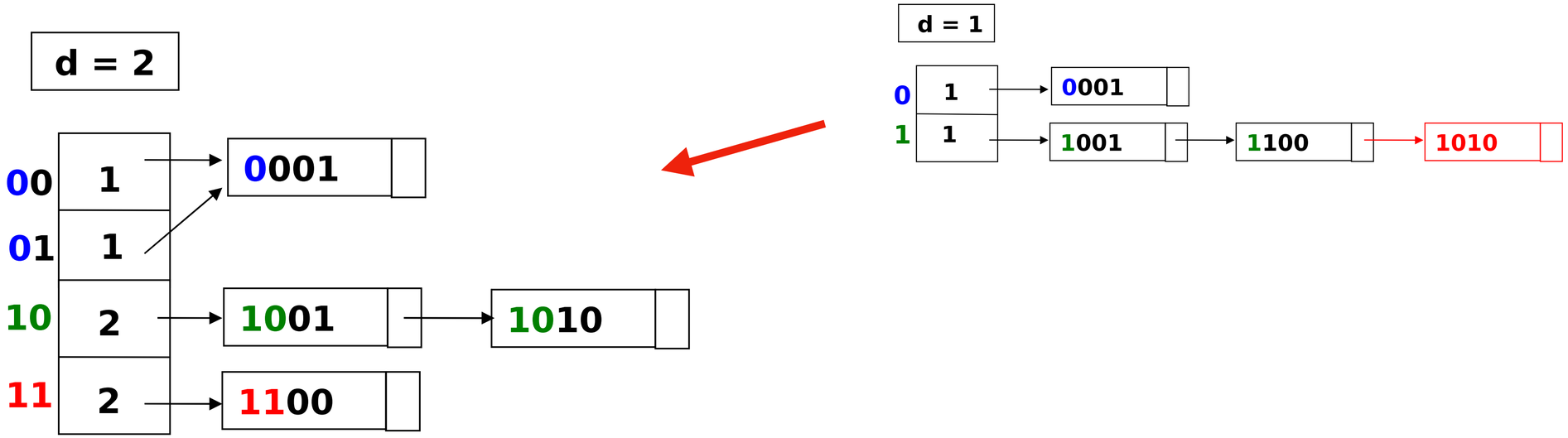
Hashing Extensível - Inserção

- *Hashing* extensível: inserção do elemento 1010
  - m = 4 (bits), N = 2
    - N é ultrapassado e a tabela precisa ser **rearranjada**, pois um único bit não é suficiente para diferenciar os elementos, sendo que o índice em que houve problema tem seu bit incrementado



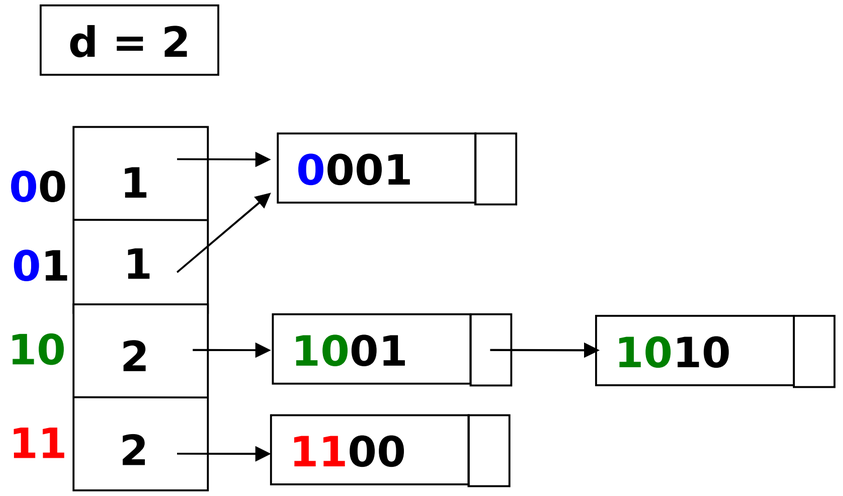
Hashing Extensível - Inserção

- *Hashing* extensível: rearranjando tabela
  - m = 4 (bits), N = 2
    - Número de posições aumenta para observar a restrição de N e chaves são rearranjadas



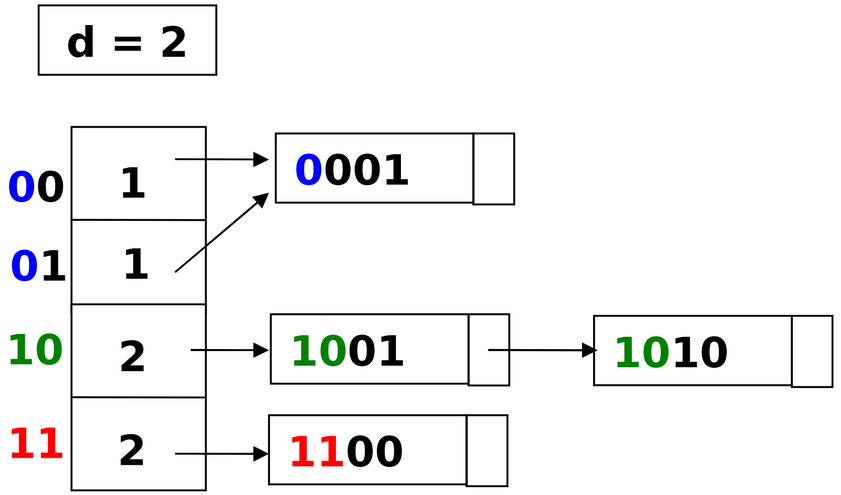
Exercício

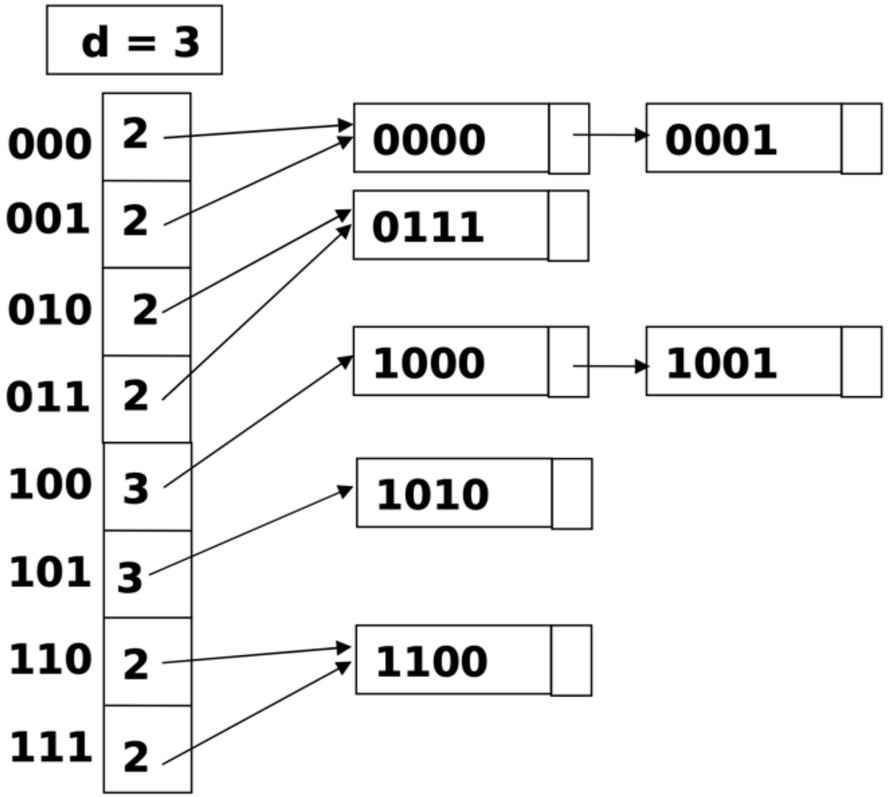
- Insira os elementos 0000, 0111 e 1000, nesta ordem



Exercício

- Insira os elementos 0000, 0111 e 1000, nesta ordem





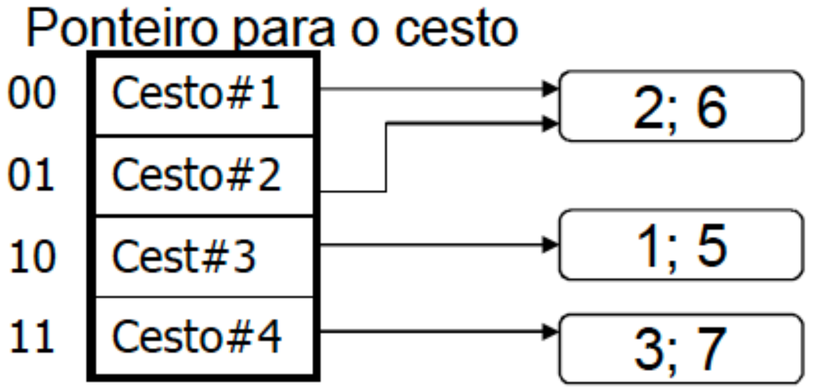
Exercício

- Represente a estrutura de *hash* extensível para as seguintes chaves: **2; 6; 1; 5; 3; e 7** considerando
  - Que a função *hash* retorna como índice do diretório os dois bits menos significativos do número em ordem inversa Ex. h(2) -> 0010 => 01

Exercício

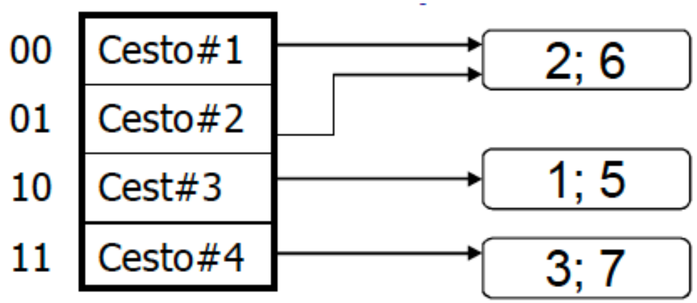
- Represente a estrutura de *hash* extensível para as seguintes chaves: **2; 6; 1; 5; 3; e 7** considerando
  - Que a função *hash* retorna como índice do diretório os dois bits menos significativos do número em ordem inversa Ex. h(2) -> 0010 => 01

- Solução: Temos as seguintes representações binárias para as chaves:
  - 2 (0010) e 6 (0110) 2 bits menos significativos em ordem inversa= 01
  - 1 (0001) e 5 (0101) 2 bits menos significativos em ordem inversa=10
  - 3 (0011) e 7 (0111) 2 bits menos significativos em ordem inversa = 11



Exemplo - Um caso simples de inserção

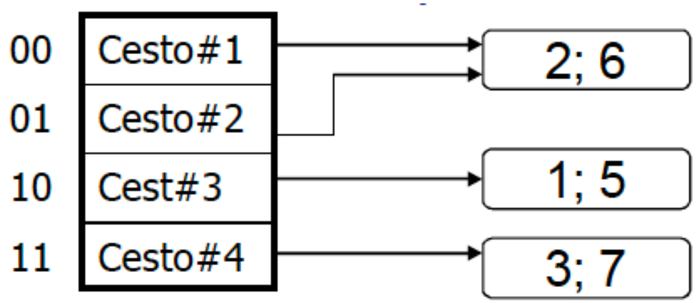
Insira a chave 4 na estrutura dada abaixo considerando que cada cesto comporta 2 registros



Considere que a função hash retorna como índice do diretório os dois bits menos significativos do número em ordem inversa Ex. h(2) -> 0010 => 01

Exemplo - Um caso simples de inserção

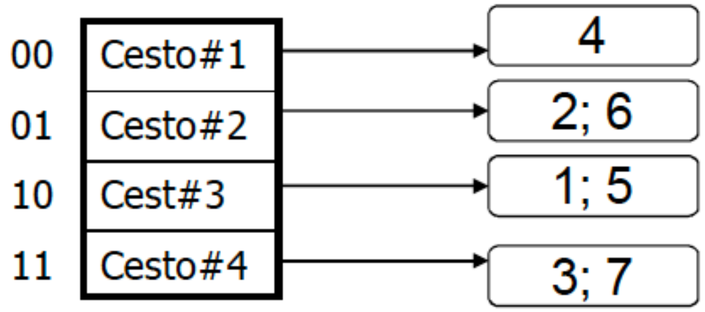
Insira a chave 4 na estrutura dada abaixo considerando que cada cesto comporta 2 registros



Considere que a função hash retorna como índice do diretório os dois bits menos significativos do número em ordem inversa Ex. h(2) -> 0010 => 01

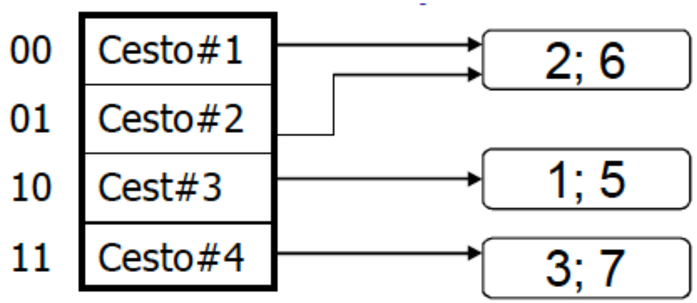
Solução:

- A representação binária da chave **4** é **0100**
- O bit menos significativo de 0100 em ordem inversa é **00** -> cesto #1
- As chaves **2** e **6** (0010 - 0110) tem bit menos significativo em ordem inversa = **01**
- O bit **00** já foi previsto no diretório



Exemplo - Um caso mais complexo de inserção

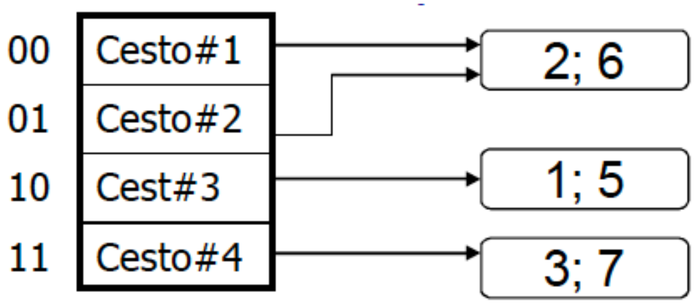
Insira a chave 9 na estrutura dada abaixo considerando que cada cesto comporta 2 registros

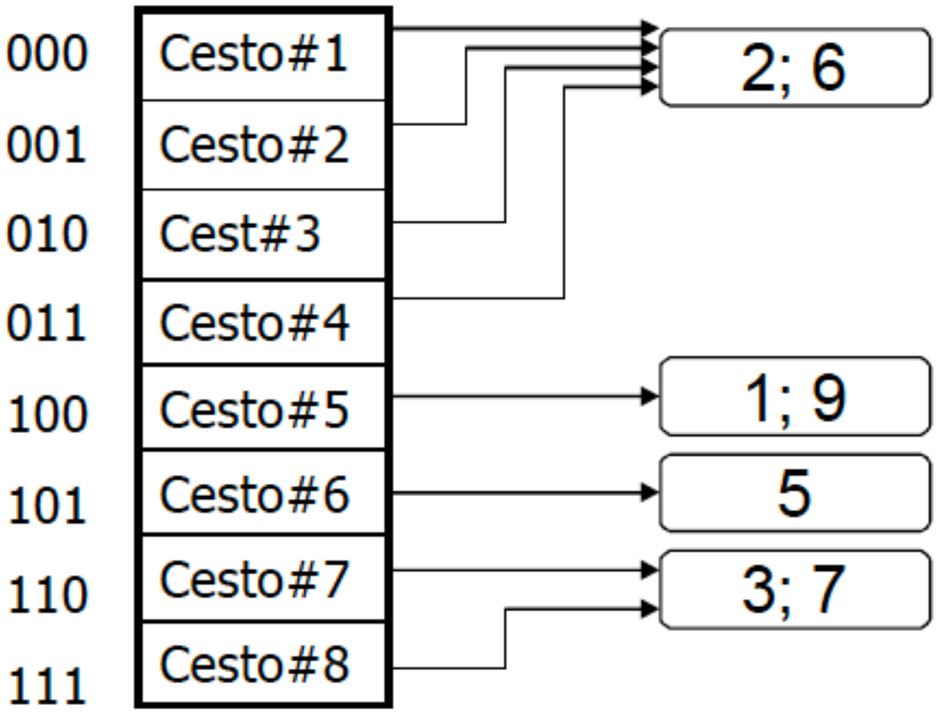


Exemplo - Um caso mais complexo de inserção

Solução:

- A representação binária da chave 9 é 1001
- O bit menos significativo de 1001 em ordem inversa é **10** -> **cesto #3**
- A subdivisão para separar a chave 9 das chaves 1 (0001) e 5 (0101):
  - O diretório não está preparado para acomodar a sub-divisão **->** adicionar um bit extra no índice **->** dobrar o tamanho do diretório
  - 3 bits menos significativos em ordem inversa das chaves
    - para a chave **9 (1001) -> 100**
    - para as chaves **1 e 5 (0001- 0101) -> 100 e 101**





Exercício

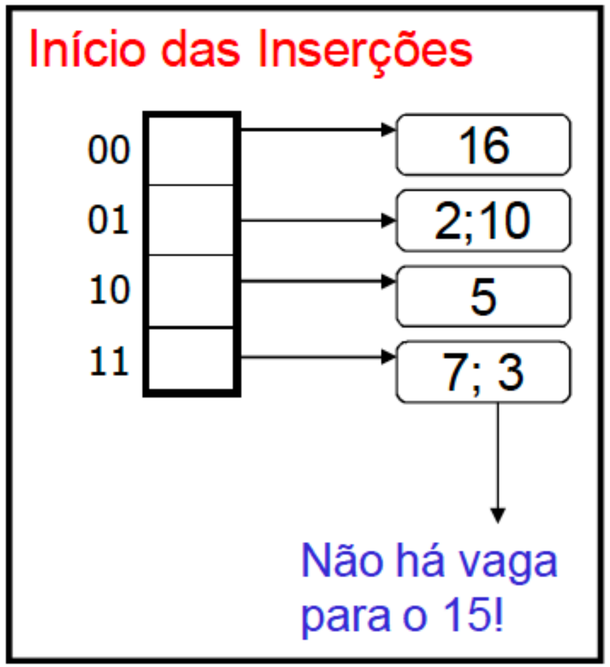
- Insira as seguintes chaves em uma estrutura de *hash* extensível vazia. Considere que cada cesto pode conter até 2 registros.
  - a) Chaves: 2; 10; 7; 3; 5; 16;
  - b) Acrescentar 15; 9 às chaves já incluídas

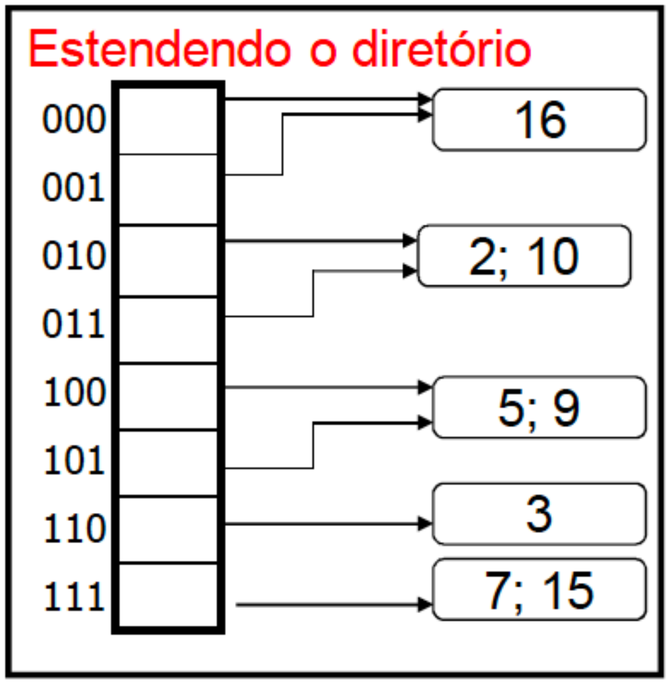
2 → 00010 →01
10→ 01010 →01
7 → 00111 →11
3 → 00011 →11
5 → 00101 →10
16→ 10000 →00
15→ 01111 →11
9 → 01001 →10

Exercício

- Insira as seguintes chaves em uma estrutura de *hash* extensível vazia. Considere que cada cesto pode conter até 2 registros.
  - a) Chaves: 2; 10; 7; 3; 5; 16;
  - b) Acrescentar 15; 9 às chaves já incluídas

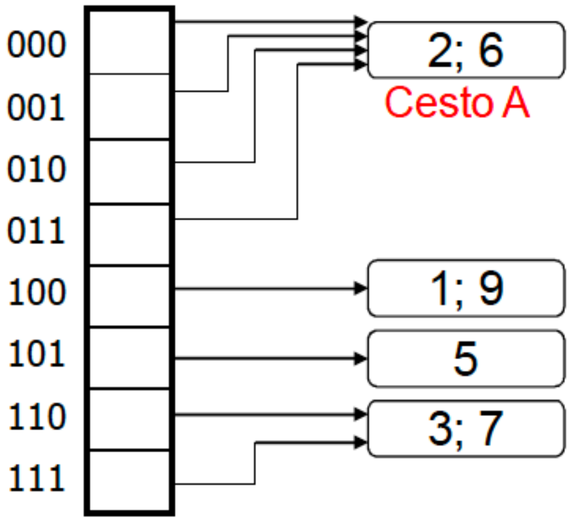
2 → 00010 →01
10→ 01010 →01
7 → 00111 →11
3 → 00011 →11
5 → 00101 →10
16→ 10000 →00
15→ 01111 →11
9 → 01001 →10

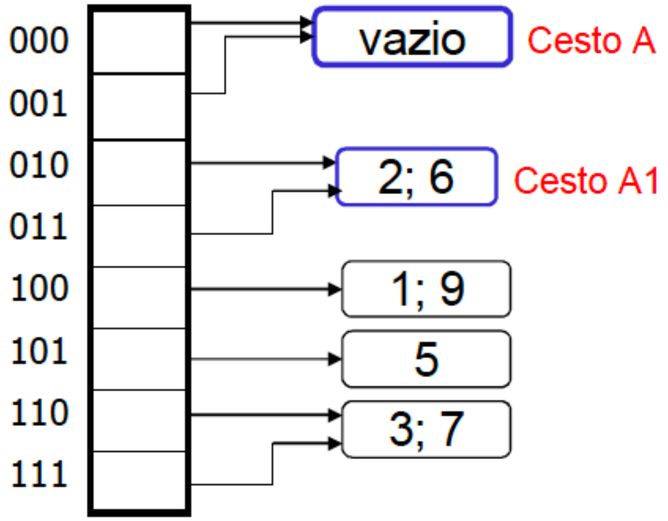




Inserção: com várias divisões de cesto

- Em alguns casos várias subdivisões são necessárias para acomodar uma nova chave.
- O exemplo a seguir aplica o algoritmo para implementar tais casos.
  - Exemplo: inserir a chave 10 (1010 -> 010) na estrutura dada abaixo



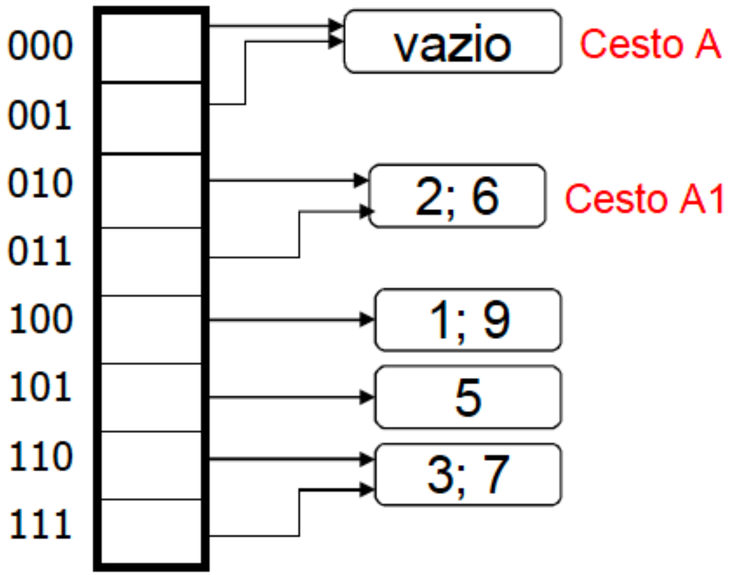


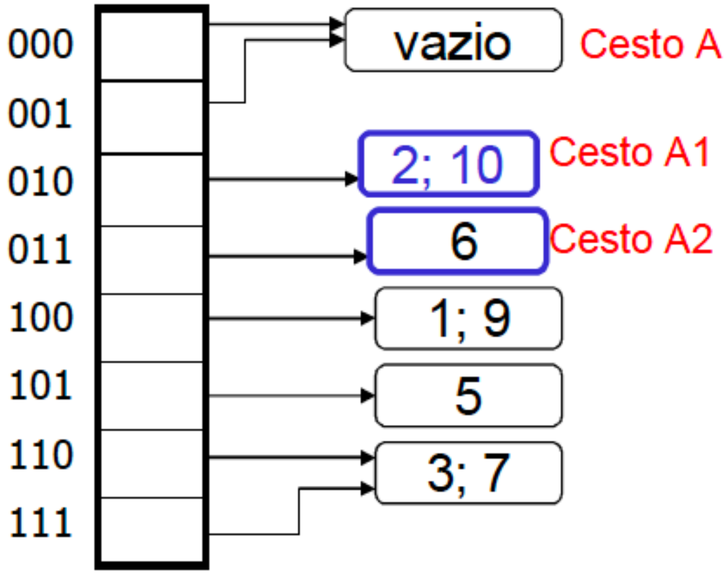
- Note que o **cesto A** deve ser subdividido
- A tem profundidade 1, visto que um **dígito (0)** determina se uma chave deve ser **colocada em A**
- A 1a. subdivisão deve criar um novo cesto A1. A e A1 têm profundidade 2 **->** 2 dígitos serão examinados para determinar se uma chave vai para A(00*) ou A1(01*)

Inserção: com várias divisões de cesto

- Será necessário subdividir o cesto A1 para acomodar a chave 10 (1010 -> 010) na estrutura dada abaixo
- O cesto apontado por 010, que é A1, está cheio -> subdividir A1
- A1 tem profundidade 2
  - ele será subdividido em A1 e A2 com profundidade 3

6 (0110) -> 011
2 (0010) -> 010





Hashing Extensível - Algoritmo Inserção

```
Insert (key) {
// indexkey: indice no diretório onde a chave deve ser colocada
// A: cesto apontado pelo indexkey
Se (cesto A não está cheio)
      então coloque a chave no cesto A
      senão { // cesto está cheio e deve ser dividido
                se (a profundidade do cesto = profundidade do
          diretório) então
                      { dobre o tamanho do diretório
                       reajuste os ponteiros do cesto
                      }
                senão divida o cesto A
                Insert(key);  // chamada recursiva da função
                }
```

Hashing Extensível - Remoção

- Localiza chave no diretório
- Se encontrada, elimina a chave do seu cesto
- Verifica-se se o cesto possui “*cestos amigos*”
  - Um par de cestos “*amigos*” é formado por dois cestos que são descendentes imediatos do mesmo nó na trie
- Se o “*cesto amigo*” existe, então verifica se é possível unir os cestos “*amigos*”
- Verifica se é possível diminuir (colapsar) o tamanho do diretório

Hashing Extensível - Remoção (par de cestos amigos)

- Se a profundidade do cesto for menor que a profundidade do diretório, tal cesto não tem “amigo”
- Caso o “amigo” exista, pode-se determinar o endereço do cesto “amigo” usando o do cesto atual
- No exemplo:
  - Endereço de B: 10**0**; Endereço de C: 10**1**

unesp

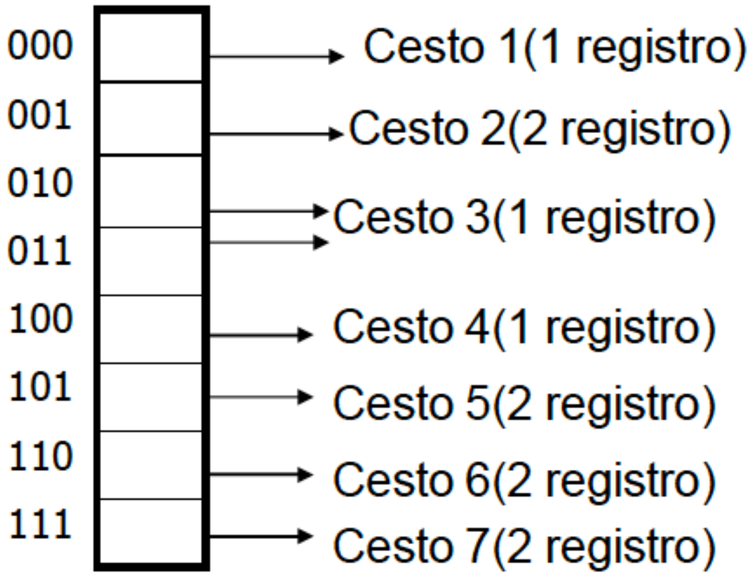
Hashing Extensível - Remoção (colapso de diretórios)

- Se um par de cestos “amigo” é unido, pode acontecer que todo cesto tenha, no mínimo, um par de endereços referenciando-o
  - Verifica-se se todas as profundidades dos cestos são menores que a do diretório
- Neste caso, o diretório pode ser colapsado, e seu tamanho reduzido pela metade

unesp

Hashing Extensível - Remoção (Combinação de Cestos)

- Quando fazemos remoção de registros pode ser necessário combinar cestos, ou seja verificar se o cesto que agora está menor pode ser combinado com um companheiro.
- Um **par de cestos amigos** é formado por **dois cestos que são descendentes imediatos do mesmo nó** em uma trie: eles são de fato, filhos consecutivos resultantes de uma sub-divisão.
- Após a combinação de cestos o diretório poderá ou não ser alterado (dividido pelo meio).
- Qual dos seguintes cestos podem ser combinados se for removida uma chave, considerando cestos de tamanho 2?



➢ **Podem ser combinados:**

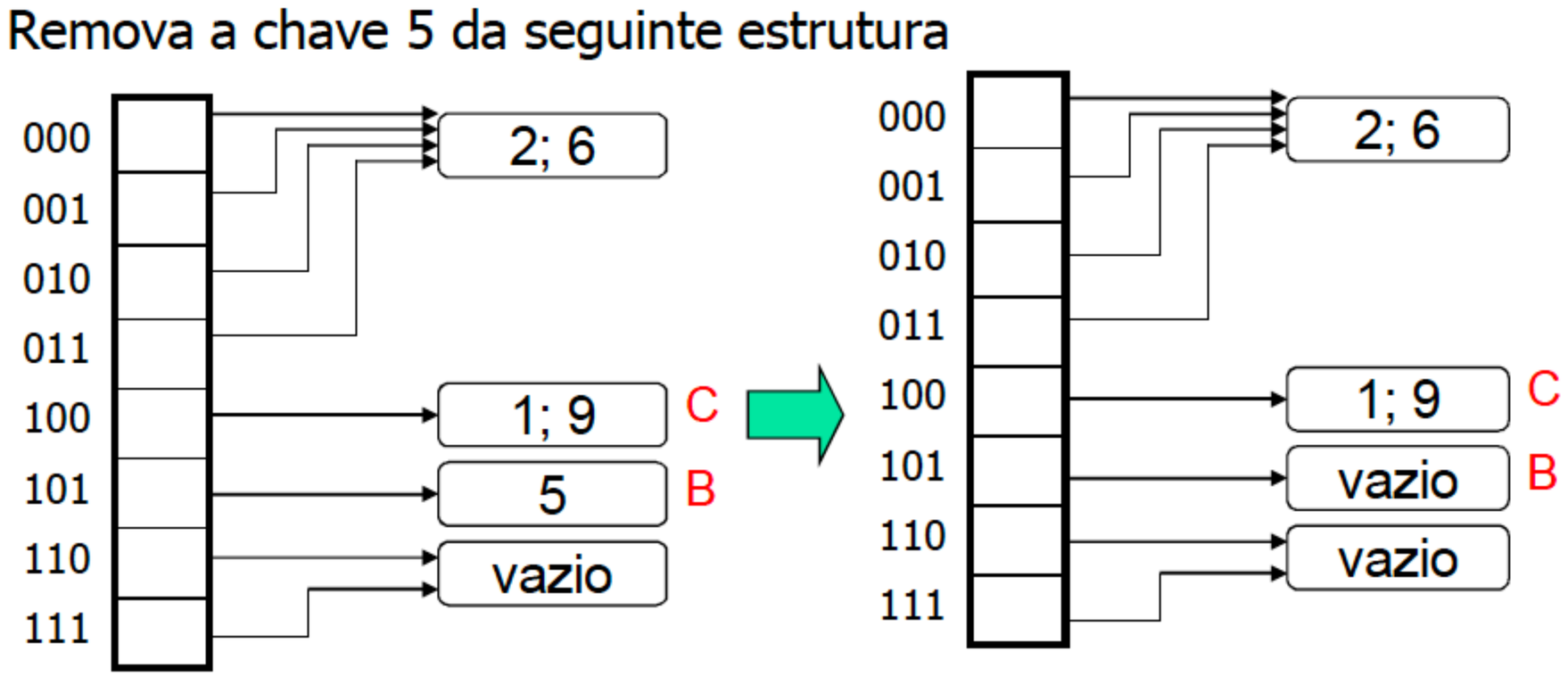
- Cestos 1&2
- Cestos 4&5

➢ **não podem ser combinados:**

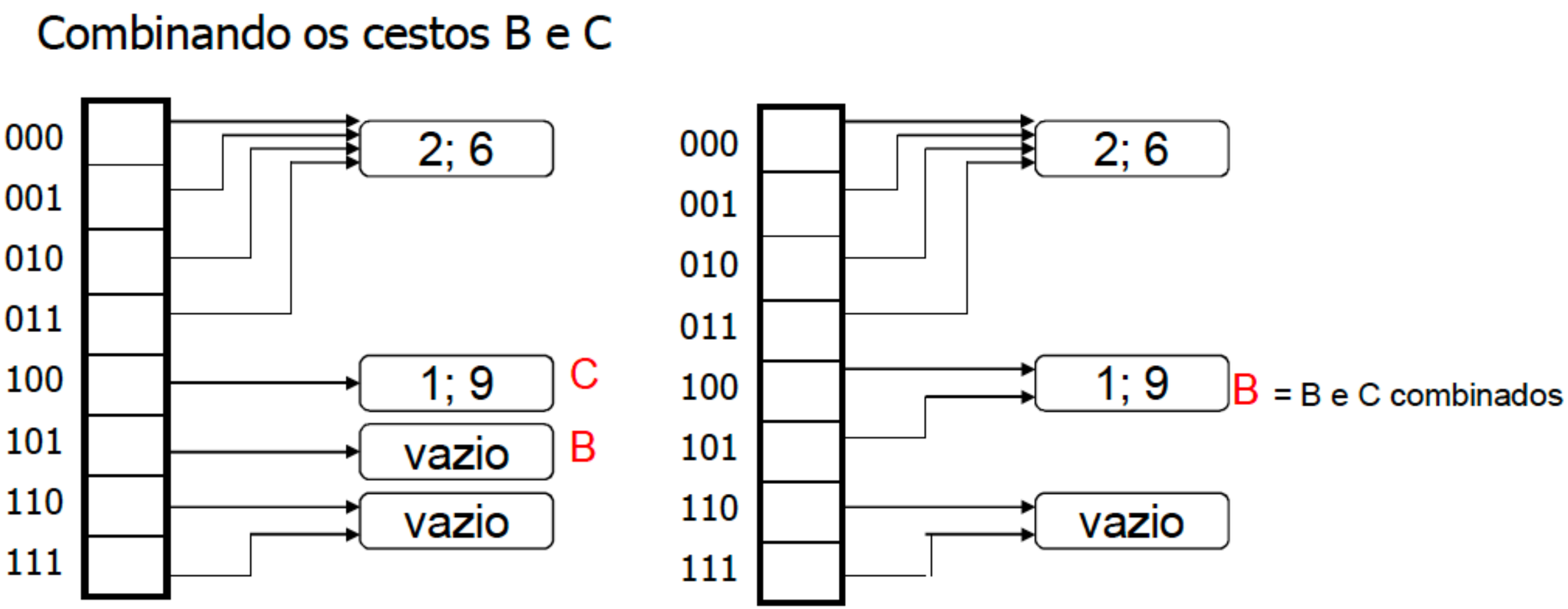
➢ O cesto 3 pois ele armazena todas as chaves iniciadas com os bits 01*

➢ Os cestos 6 e 7 pois não poderão armazenar uma terceira chave

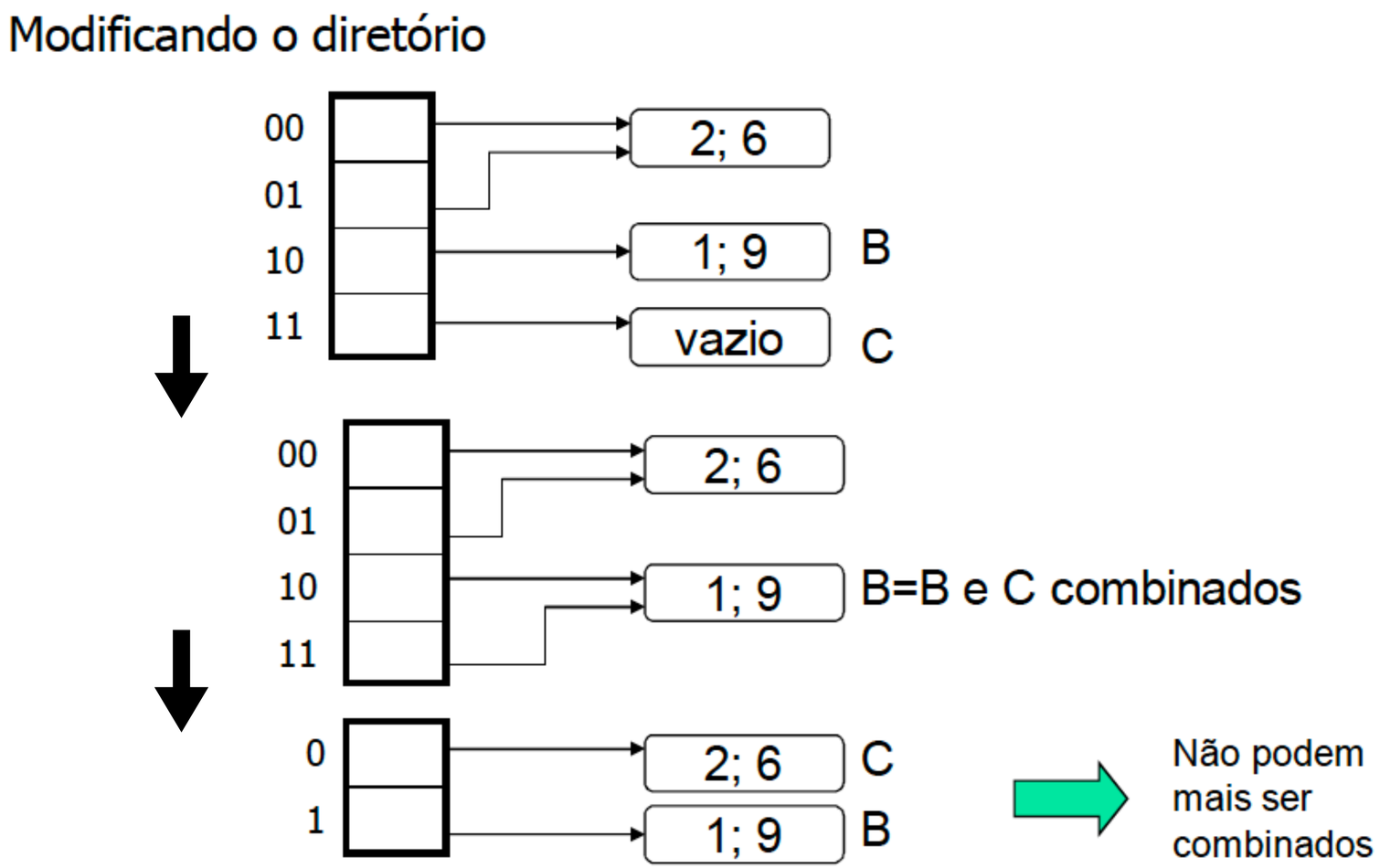
Hashing Extensível - Remoção (Combinação de Cestos)



Hashing Extensível - Remoção (Combinação de Cestos)



Hashing Extensível - Remoção (Combinação de Cestos)



Utilização de espaço

- Utilização de memória para o diretório
  - Frajolet em 1983: tamanho estimado do diretório = 3.92 r(1+1/b)/b. Sendo r o número total de registros e b o tamanho do cesto
  - EX: um diretório para mil registros e cesto de tamanho 5 tem um tamanho estimado de 1.5 Kb.
- Utilização de memória para os cestos
  - Espera-se que os cestos estejam entre 0.53 e 0.94 cheios.
  - Fagin *at al* sugere que para um número r de registros e um bloco de memória de tamanho b, o número médio de blocos N é aproximadamente: N = r / (b *log* 2).
  - A utilização é dada por r / (b N). Substituindo N nesta fórmula temos que a ocupação dos cestos é igual a log 2 = 0.69 -> esperamos uma utilização de 69%.

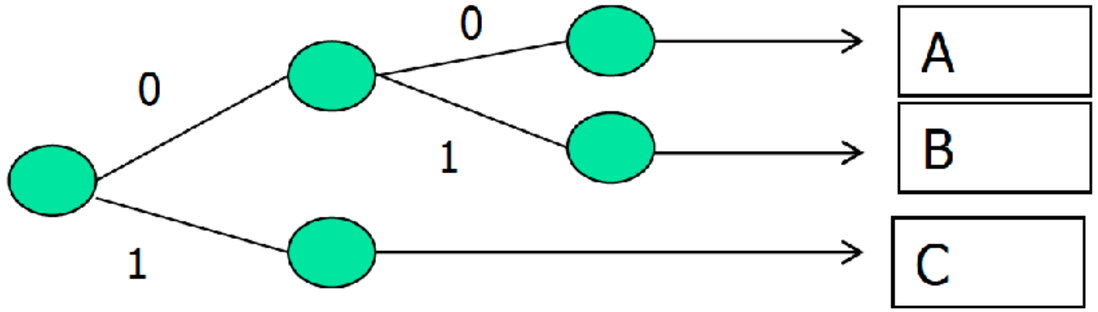
Desempenho de espalhamento extensível

- Se o diretório puder ser mantido em RAM, é necessário **apenas um seek**.
- Se o diretório precisa ser alocado **em páginas do disco**, tem-se **2 seeks no pior caso.**
- A utilização do espaço alocado aos cestos é de **aproximadamente 69%.**
- Hash extensível é uma solução elegante para o problema de estender ou contrair espaço de endereços para um arquivo de hash conforme ele cresce ou diminui.
- Hash extensível é usado quando temos um grande número de informações que têm de ser acessadas rapidamente.
  - Exemplos: sistemas de arquivos de um sistema operacional e sistemas de bancos de dados.

unesp

Exercícios

(1) Considere a seguinte *trie* de ordem (raio) 2, com ponteiros para *buckets* com capacidade para abrigar 100 chaves (ou registros):



a. Desenhe a *trie* estendida e o diretório de endereços hash correspondente.

b. Considerando que os buckets A, B e C contém, respectivamente, 100, 50 e 03 registros, dê a configuração do diretório, e a condição de cada bucket após a inserção de uma nova chave cujo valor da função hash é 00.

c. Ainda na configuração inicial, considere agora que todas as chaves de B são eliminadas. O que acontece com o diretório?

Exercícios

(2) Considere um máximo de 3 elementos por *bucket*, e que a função hash gera 4 bits para uma chave. Simule a inserção de chaves que geram os seguintes endereços: 0000, 1000, 1001, 1010, 1100, 0001, 0100, 1111, 1011

(3) Considere um máximo de **2 elementos por *bucket***, e que a função hash pega o **3 bits menos significante do resultado de k *mod* 8 (**profundidade **máxima igual a 3)**. Exemplo: k=3 **->** 3 mod 8 = 3 **->** 011
-Simule a inserção de chaves que geram os seguintes endereços: 1,2,3,4,5,6,7,8 e 9.

Exercícios

Para exercício (3) e outros pratique em https://devimam.github.io/exhash/
Lá você pode escolher a função hash e outros parâmetros

Referências

- FOLK, M.J. File Structures, Addison-Wesley, 1992.
- File Structures: Theory and Pratice”, P. E. Livadas, Prentice-Hall, 1990;
- Contém material extraído e adaptado das notas de aula dos professores Moacir Ponti, Thiago Pardo, Leandro Cintra, Thelma Cecília Chiossi e Maria Cristina de Oliveira.