ACH2025

Laboratório de Bases de Dados

Aula 9

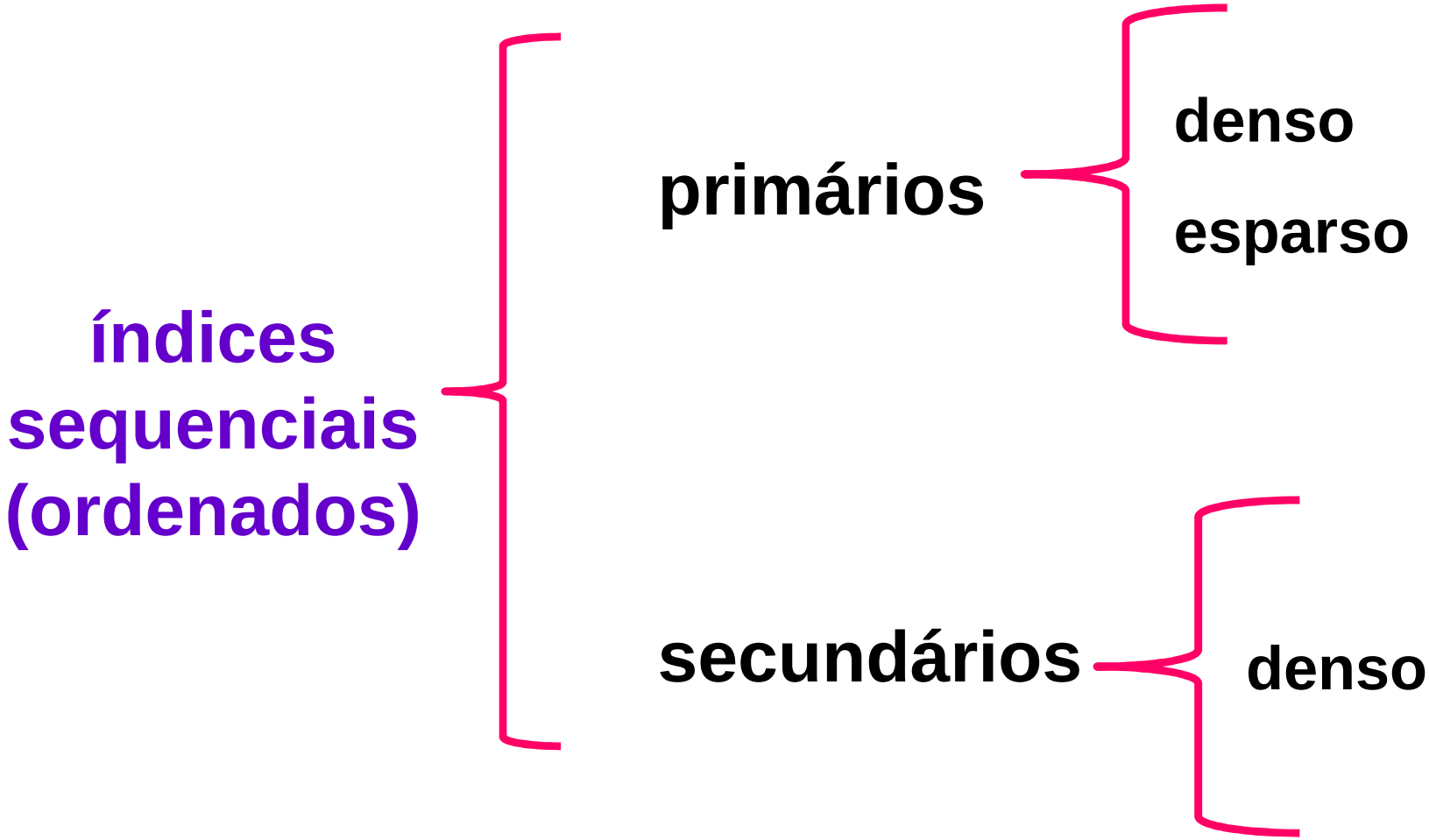
Indexação e Hashing – Parte 2

Professora:

- **Fátima L. S. Nunes**

Índices:

- ✓ Até agora:



Índices:

- ✓ Grande desvantagem de índices sequenciais ???

Índices:

- ✓ Grande desvantagem de índices sequenciais ➔ desempenho cai conforme arquivo cresce
- ✓ Degradação remediada reorganizando arquivos
  - mas isso não é desejável que se faça frequentemente
- ✓ Solução?

Índices:

- ✓ Grande desvantagem de índices sequenciais ➔ desempenho cai conforme arquivo cresce
- ✓ Degradação remediada reorganizando arquivos
  - mas isso não é desejável que se faça frequentemente
- ✓ Solução?
  - Outras estruturas de organização, que não sejam sequenciais.

Conceitos iniciais (simplificados)

- ✓ Árvore-B:
  - Árvore de busca balanceada
  - Árvore de busca de ordem *n* ➔ todos os nós (exceto a raiz) têm no mínimo ⌈(n-1/)2⌉ chaves e no máximo *n-1* chaves
  - Todos os nós internos com *k* chaves possuem *k+1* filhos
  - Chaves não se repetem nos nós
- ✓ Árvore-B*:
  - Semelhante à Árvore-B
  - Árvore de busca de ordem *n* ➔ todos os nós (exceto a raiz) no mínimo ⌈*(2/3)n-1* ⌉ chaves e no máximo *n-1* chaves
  - Todos os nós internos com *k* chaves possuem *k+1* filhos
- ✓ Árvore-B+:
  - Semelhante à Árvore-B
  - Nós internos só possuem chaves - dados (registros) estão somente nos nós-folha
  - Chaves podem se repetir

Índices Árvore-B$^+$

- ✓ Características da estrutura de índice árvore-B$^+$:
  - mantém eficiência, mesmo com inserção e remoção de dados
  - forma de uma árvore balanceada – todos os caminhos a partir da raiz da árvore até uma das folhas são do mesmo comprimento
  - cada nó não-folha possui entre $\lceil (n/2) \rceil$ e n filhos ➔ n é fixo para uma árvore em particular
  - impõe uma sobrecarga de desempenho na inclusão e remoção – aceitável porque evita o custo de reorganização
  - desperdício de espaço aceitável (existe se houver nós com mínimo de filhos) – compensado pelo desempenho.

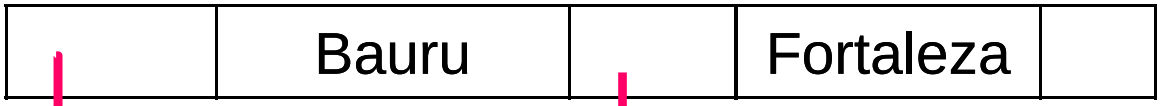
Índices Árvore-B$^+$ ➔ Estrutura da árvore

- ✓ Índice árvore-B$^+$:
  - índice de níveis múltiplos
  - estrutura de um nó típico:

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

- K_i = valores de chave de procura (*n-1* valores em cada nó)
- P_i = ponteiros
- Valores das chaves de procura dentro de um nó são mantidos ordenados (se $i < j$ ➔ $K_i < K_j$)
- Ponteiro P_i aponta para um registro do arquivo com chave de procura de valor K_i (ou para um ***bucket*** de ponteiros, cada um dos quais apontando para um registro do arquivo com chave de procura de valor K_i).
- Ponteiro P_n ➔ usado para encadear os nós folhas na ordem da chave de procura

Índices Árvore-B⁺ ➔ Estrutura da árvore



Nó folha

Cidade	Nome	Endereço
Bauru	Júlia	Rua dos Anjos, 123
Fortaleza	Ana	Av. Antonio, 865
Fortaleza	Bruno	Av. Antonio, 865

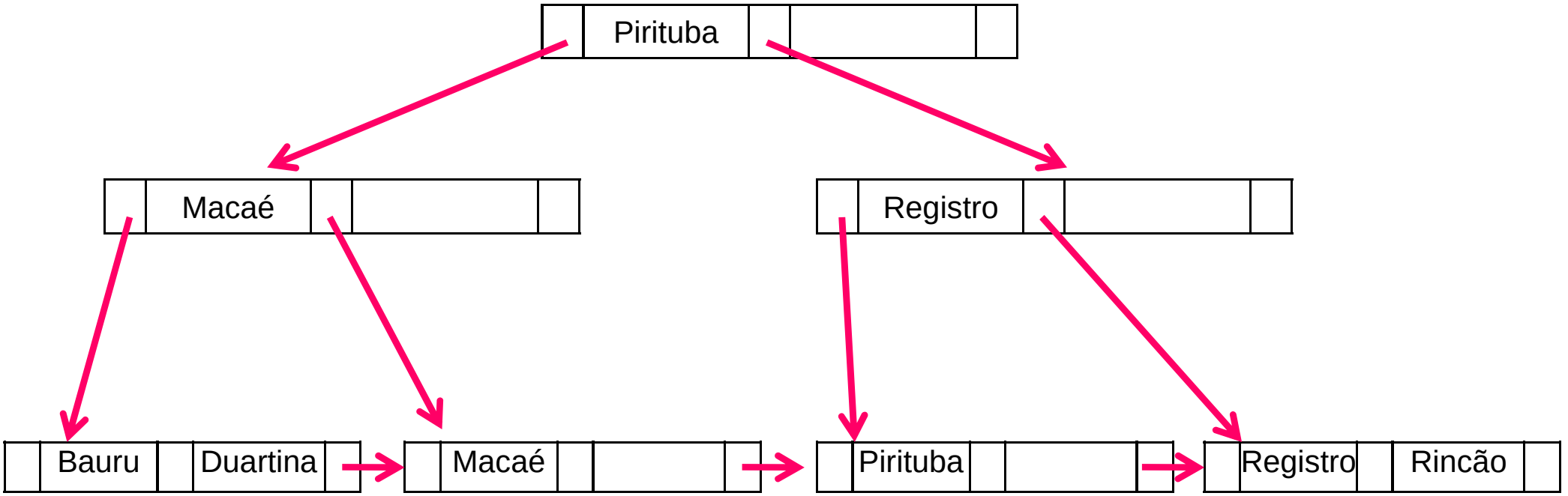
Índices Árvore-B$^+$ ➔ Estrutura da árvore

- ✓ Como os valores de chave são atribuídos a um nó em particular:
  - Cada folha pode manter até n-1 valores de chave
  - Cada nó folha contém no mínimo $\lceil (n-1)/2 \rceil$ valores de chave
  - Faixas de valores em cada folha não se sobrepõem:
    - Se L_i e L_j são nós folhas e $i < j$ ➔ todos valores de chave de procura de L_i são menores que todos os valores de chave de procura de L_j
    - Se índice for denso ➔ todos valores de chave de procura devem aparecer em algum nó folha
  - Ponteiro P_n ➔ usado para encadear os nós folhas na ordem da chave de procura

Índices Árvore-B$^+$ ➔ Estrutura da árvore

- ✓ Nós não folhas
  - formam índice esparso de níveis múltiplos dos nós folhas – mesma estrutura dos nós folhas
  - todos ponteiros apontam para nós da árvore
  - podem manter até ***n*** ponteiros e deve manter pelos menos ($\lceil n/2 \rceil$) ponteiros (exceto nó raiz)
  - número de ponteiros em um nó: *fanout (leque)* do nó

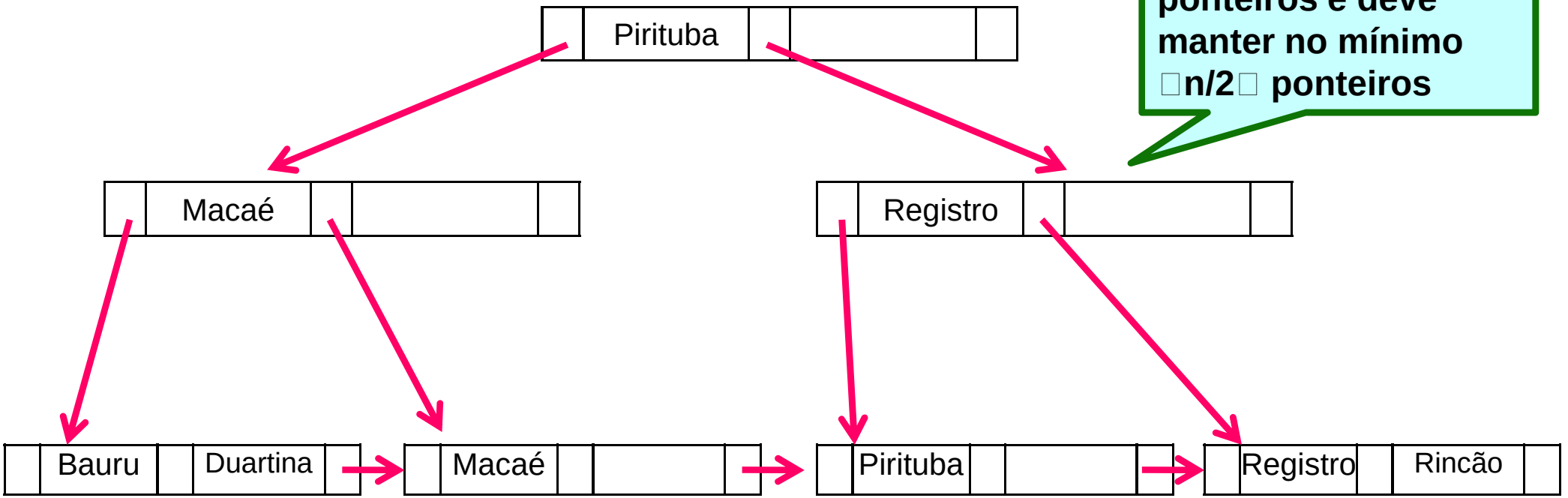
Índices Árvore-B+ ➔ Estrutura da árvore



Exemplo de Árvore-B+ para arquivo de contas de Clientes (n=3)

Índices Árvore-B+ ➔ Estrutura da árvore

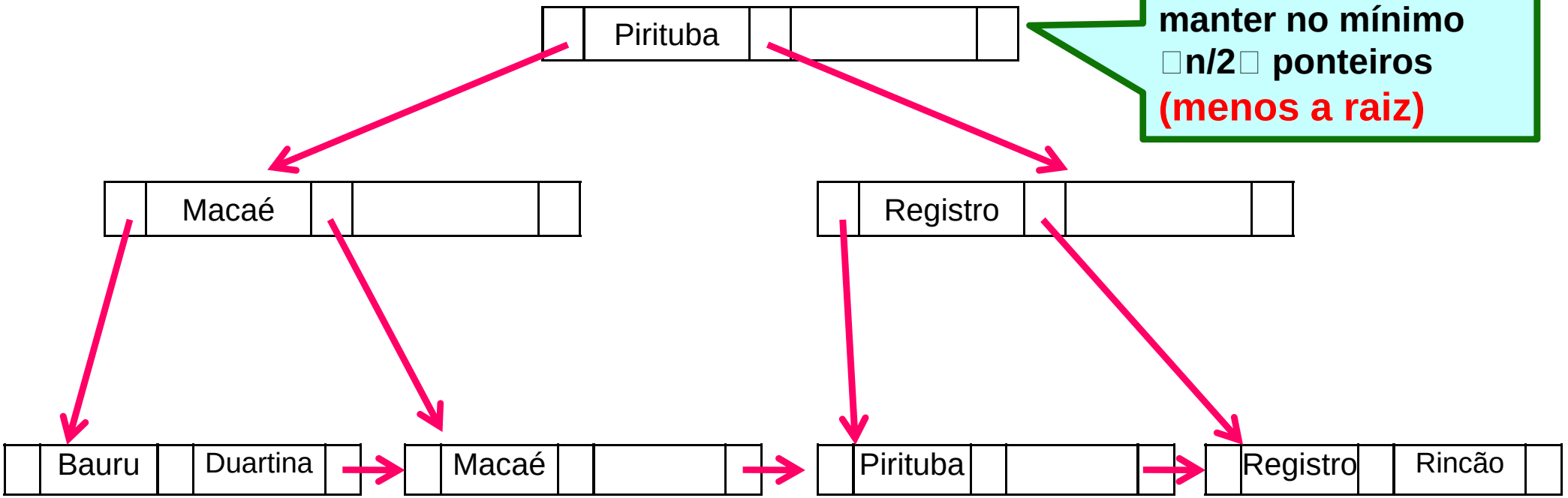
Nó não folha pode manter até *n* ponteiros e deve manter no mínimo ⌈n/2⌉ ponteiros



Exemplo de Árvore-B+ para arquivo de contas de Clientes (n=3)

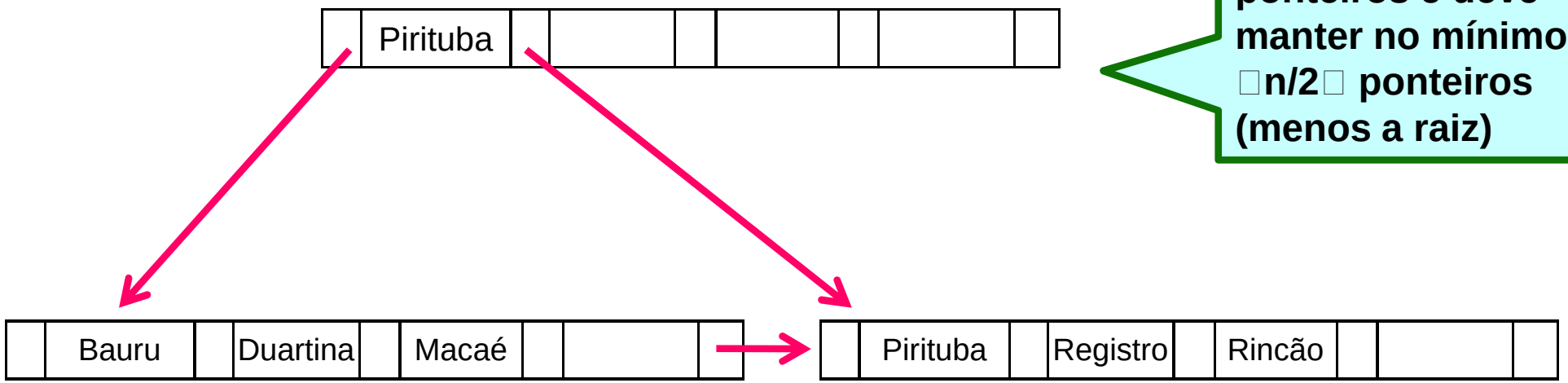
Índices Árvore-B+ ➔ Estrutura da árvore

Nó não folha pode manter até *n* ponteiros e deve manter no mínimo ⌈n/2⌉ ponteiros (menos a raiz)



Exemplo de Árvore-B+ para arquivo de contas de Clientes (n=3)

Índices Árvore-B+ ➔ Estrutura da árvore



Nó não folha pode manter até *n* ponteiros e deve manter no mínimo ⌈n/2⌉ ponteiros (menos a raiz)

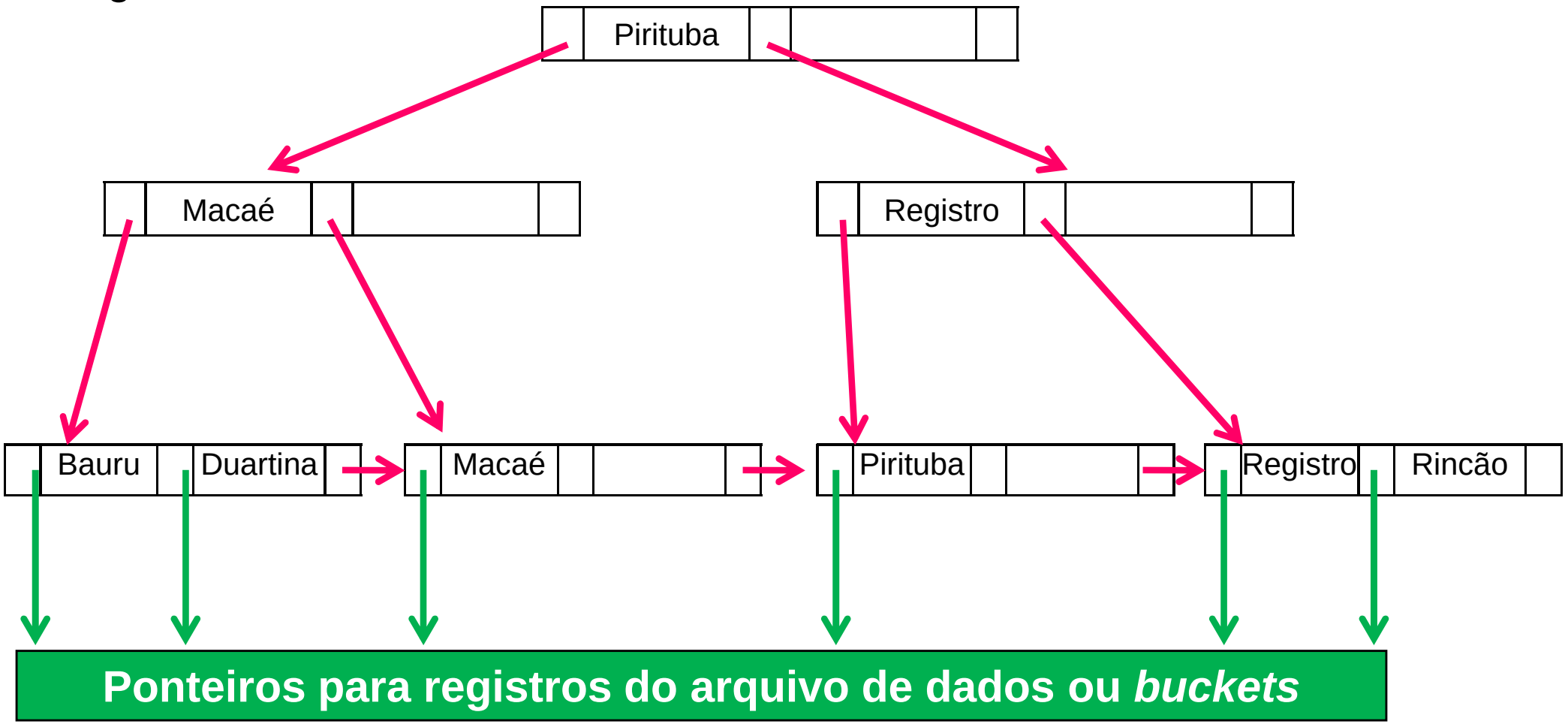
Exemplo de Árvore-B+ para arquivo de contas de Clientes (n=5)

Propriedade da Árvore-B+: "Balanceadas" – comprimento de todos os caminhos a partir da raiz para qualquer um dos nós folha é o mesmo.

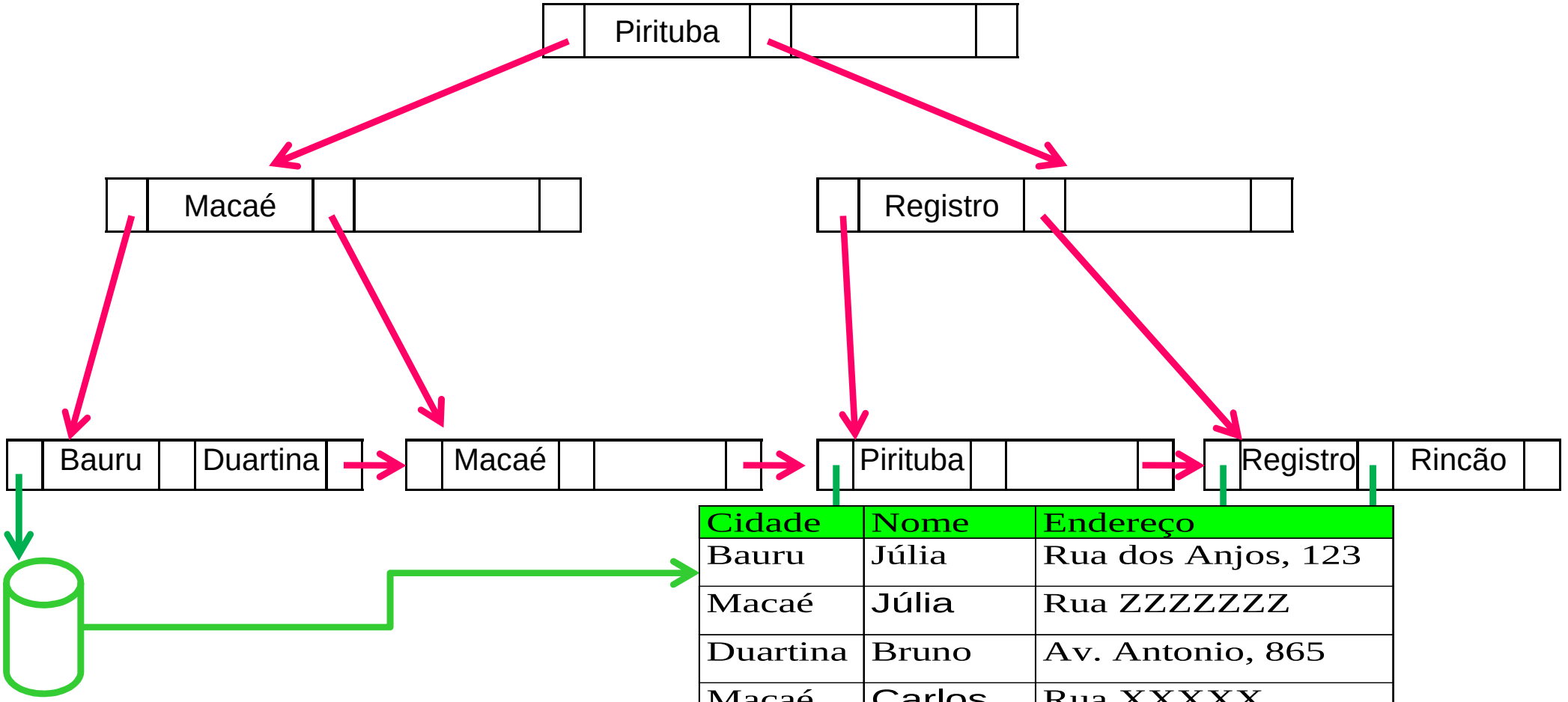
Isso assegura bom desempenho na recuperação, inserção e remoção.

Índices Árvore-B$^+$ ➔ Consultas

- ✓ O que queremos: todos os registros com valor de chave de procura = *k*
- ✓ Algoritmo:

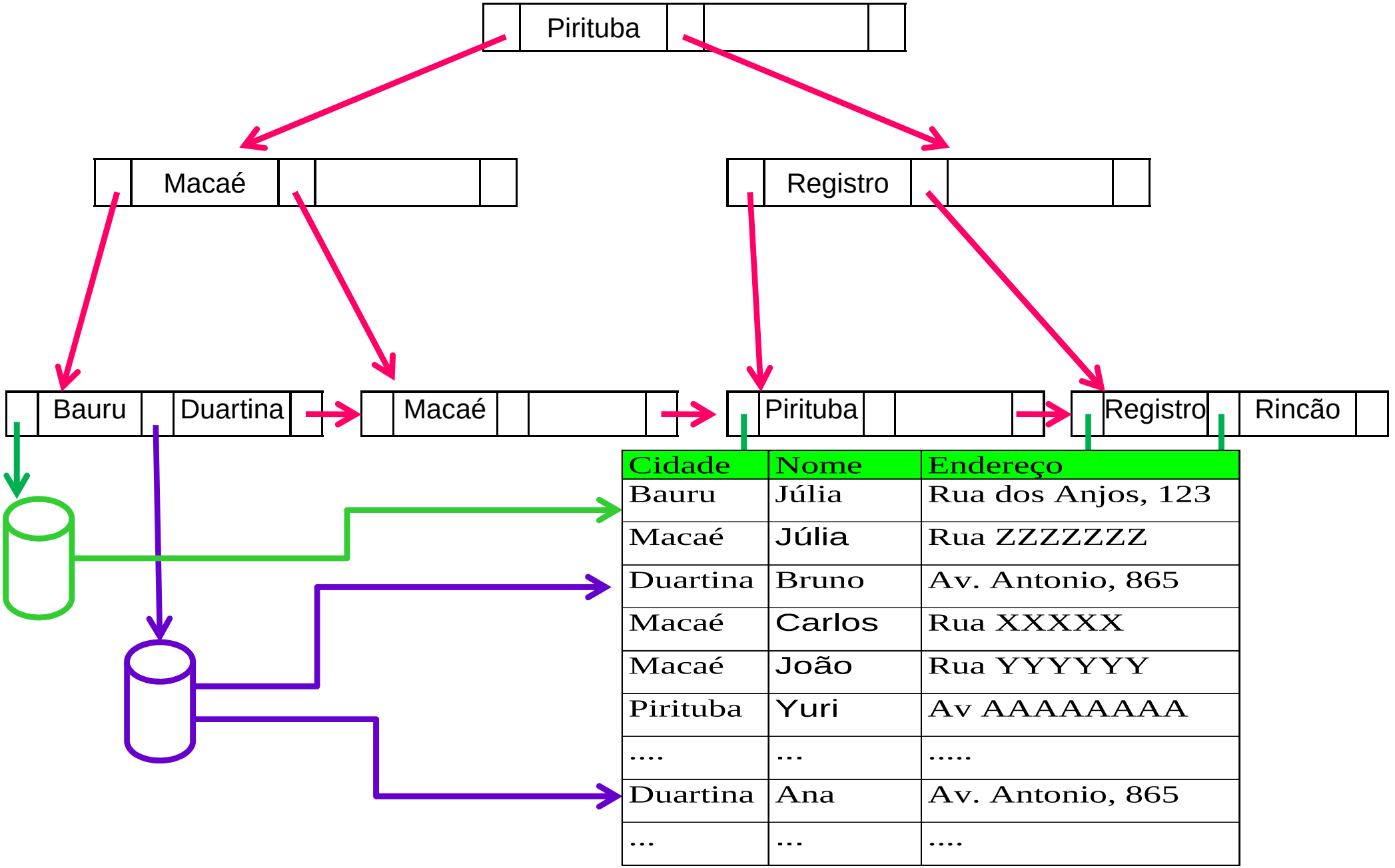


Índices Árvore-B+ ➔ Consultas



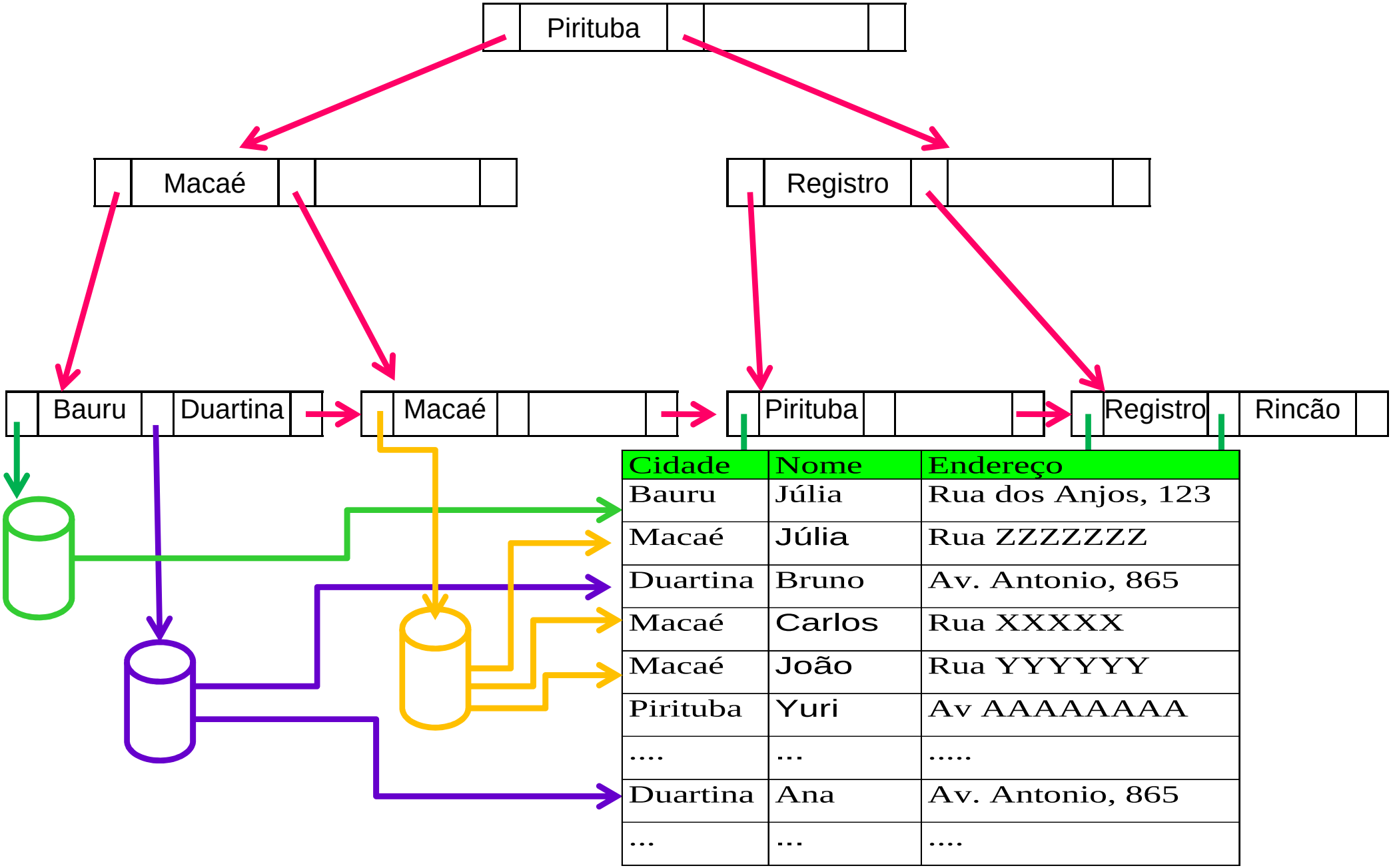
Cidade	Nome	Endereço
Bauru	Júlia	Rua dos Anjos, 123
Macaé	Júlia	Rua ZZZZZZZ
Duartina	Bruno	Av. Antonio, 865
Macaé	Carlos	Rua XXXXX
Macaé	João	Rua YYYYYY
Pirituba	Yuri	Av AAAAAAAA
....	...	.....
Duartina	Ana	Av. Antonio, 865
...	...	....

Índices Árvore-B+ ➔ Consultas



Cidade	Nome	Endereço
Bauru	Júlia	Rua dos Anjos, 123
Macaé	Júlia	Rua ZZZZZZZ
Duartina	Bruno	Av. Antonio, 865
Macaé	Carlos	Rua XXXXX
Macaé	João	Rua YYYYYY
Pirituba	Yuri	Av AAAAAAAA
....	...	.....
Duartina	Ana	Av. Antonio, 865
...	...	....

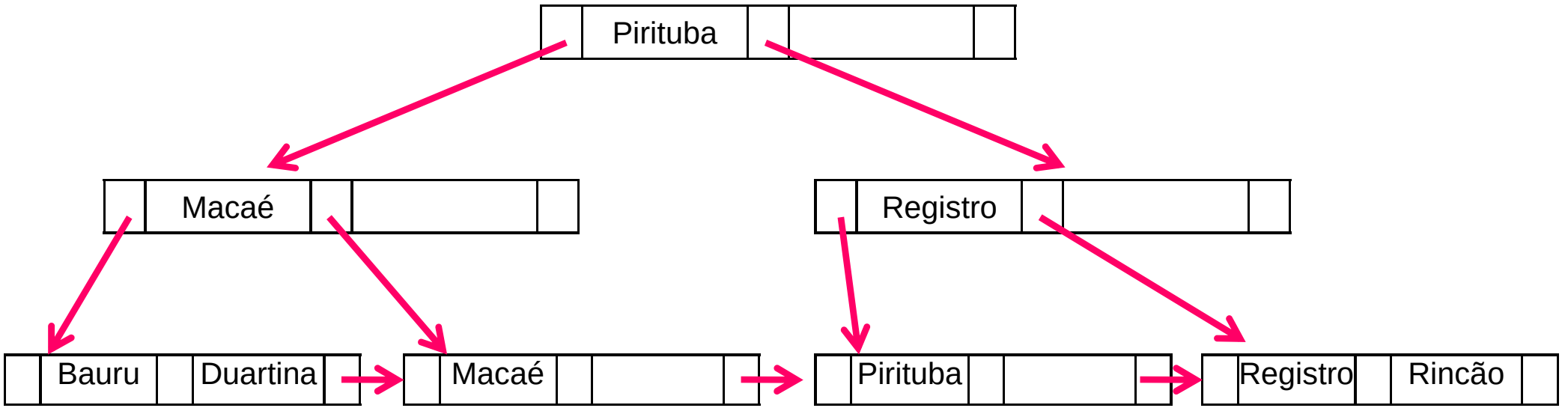
Índices Árvore-B+ ➔ Consultas



Cidade	Nome	Endereço
Bauru	Júlia	Rua dos Anjos, 123
Macaé	Júlia	Rua ZZZZZZZ
Duartina	Bruno	Av. Antonio, 865
Macaé	Carlos	Rua XXXXX
Macaé	João	Rua YYYYYY
Pirituba	Yuri	Av AAAAAAAA
....	...	.....
Duartina	Ana	Av. Antonio, 865
...	...	....

Índices Árvore-B$^+$ ➔ Consultas

- ✓ O que queremos: todos os registros com valor de chave de procura = *k*
- ✓ Algoritmo:
  - Examinar nó-raiz – procurar menor valor que seja maior ou igual a *k*. Chamamos o valor encontrado de K_i
  - Seguir o ponteiro P_i até próximo nó. Se $K < K_1$, seguimos P_1 *até outro nó*
  - Se temos *m* ponteiros nos nós e $K \geq K_{m-1}$ ➔ seguir P_m até o outro nó
  - Repetir até alcançar o nó folha ou o *bucket*



Índices Árvore-B$^+$ ➔ Consultas

- ✓ O que queremos: todos os registros com valor de chave de procura = *k*
- ✓ Algoritmo:
  - Examinar nó-raiz – procurar menor valor que seja maior ou igual a *k.* Chamamos o valor encontrado de K_i
  - Seguir o ponteiro P_i até próximo nó. Se $K < K_1$, seguimos P_1 *até outro nó*
  - Se temos *m* ponteiros nos nós e $K \geq K_{m-1}$ ➔ seguir P_m até o outro nó
  - Repetir até alcançar o nó folha ou o *bucket*
- ✓ Quantos nós são percorridos para K valores de chave de procura existentes em um arquivo?

Índices Árvore-B$^+$ ➔ Consultas

- ✓ O que queremos: todos os registros com valor de chave de procura = *k*
- ✓ Algoritmo:
  - Examinar nó-raiz – procurar menor valor que seja maior ou igual a *k.* Chamamos o valor encontrado de K_i
  - Seguir o ponteiro P_i até próximo nó. Se $K < K_1$, seguimos P_1 *até outro nó*
  - Se temos *m* ponteiros nos nós e $K \geq K_{m-1}$ ➔ seguir P_m até o outro nó
  - Repetir até alcançar o nó folha ou o *bucket*
- ✓ Quantos nós são percorridos para K valores de chave de procura existentes em um arquivo:

$$\lceil \log_{\lceil n/2 \rceil} K \rceil$$

Índices Árvore-B+ ➔ Consultas

- ✓ Tipicamente o tamanho de um nó é determinado como sendo o tamanho de um bloco de disco (em geral, 4Kbytes)
- ✓ Com chave de procura com 32 bytes e ponteiro com 8 bytes, *n* é aproximadamente 100
- ✓ Com n= 100 e 1 milhão de chaves de procura em um arquivo ➔ quantidade de nós acessados = $\log_{50}(1.000.000) = 4$ nós
  - 4 blocos do disco precisam ser lidos para uma procura
  - nó raiz geralmente está no buffer: geralmente 3 ou menos blocos lidos

Índices Árvore-B$^+$ ➔ Consultas

- ✓ Diferença entre estruturas de árvore-B$^+$ e estruturas de árvores binárias: tamanho do nó (consequência ➔ altura da árvore)
  - Árvore binária: nó pequeno e com no máximo 2 ponteiros
  - Árvore-B$^+$: nó grande (tamanho bloco em geral) e nó com grande número de ponteiros
  - tamanho do caminho de busca em árvore binária balanceada: $\lceil \log_2 K \rceil$ (K=quantidade de valores da chave de procura)

    ➔ 20 acessos para K= 1.000.000

Índices Árvore-B$^+$ ➔ Atualizações

- ✓ Inserção e remoção mais complexas que busca porque:
  - divisão de nó muito grande em inserção
  - combinação de nós pequenos (< que $\lceil n/2 \rceil$ ponteiros) na remoção
  - garantia de balanceamento

Índices Árvore-B$^+$ ➔ Atualizações

- ✓ Inserção (sem divisão de nós)
  - encontrar nó folha, conforme definido na busca
  - se valor procurado aparece no nó folha:
    - adicionar novo registro ao arquivo e ponteiro no *bucket* (se necessário)
  - se valor procurado não aparece no nó folha:
    - inserir valor no nó folha, mantendo a ordem das chaves de procura
    - adicionar novo registro ao arquivo e ponteiro no *bucket* (se necessário)

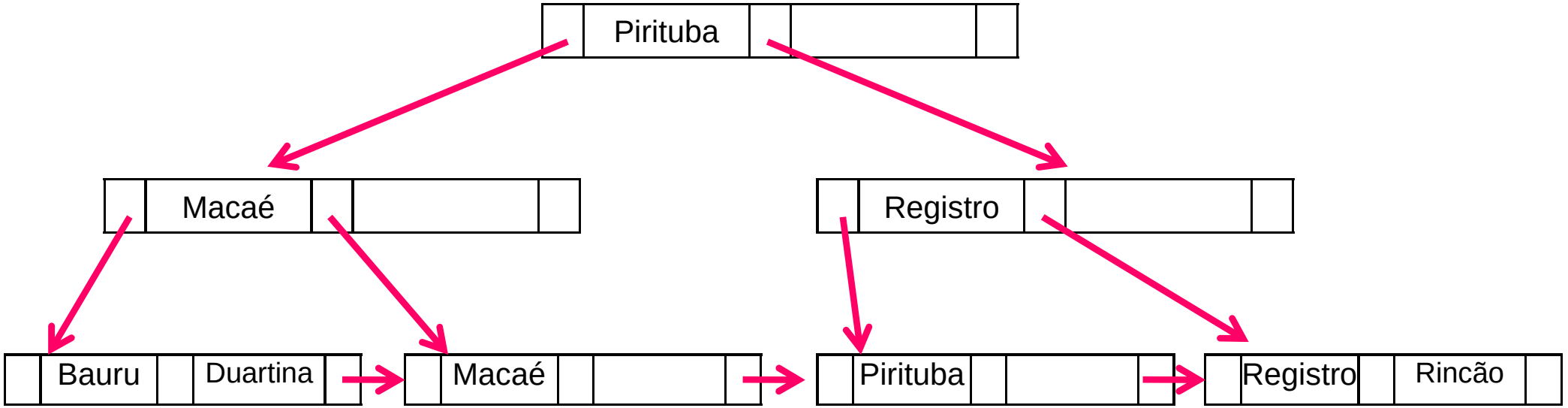
Índices Árvore-B$^+$ ➔ Atualizações

✓ Inserção (com divisão de nós)

- encontrar nó folha, conforme definido na busca
- se não há espaço para inserir o valor de chave procura
  - dividir o nó em dois (considerando n-1 valores existentes + valor inserido)
    - primeiro nó (já existente): n/2 primeiros
    - segundo nó (novo): valores restantes
  - inserir o novo nó folha na estrutura da árvore
    - se não houver espaço ➔ dividir o pai
- Pior caso: todos os nós divididos, até a própria raiz (árvore se torna mais profunda se raiz for dividida)

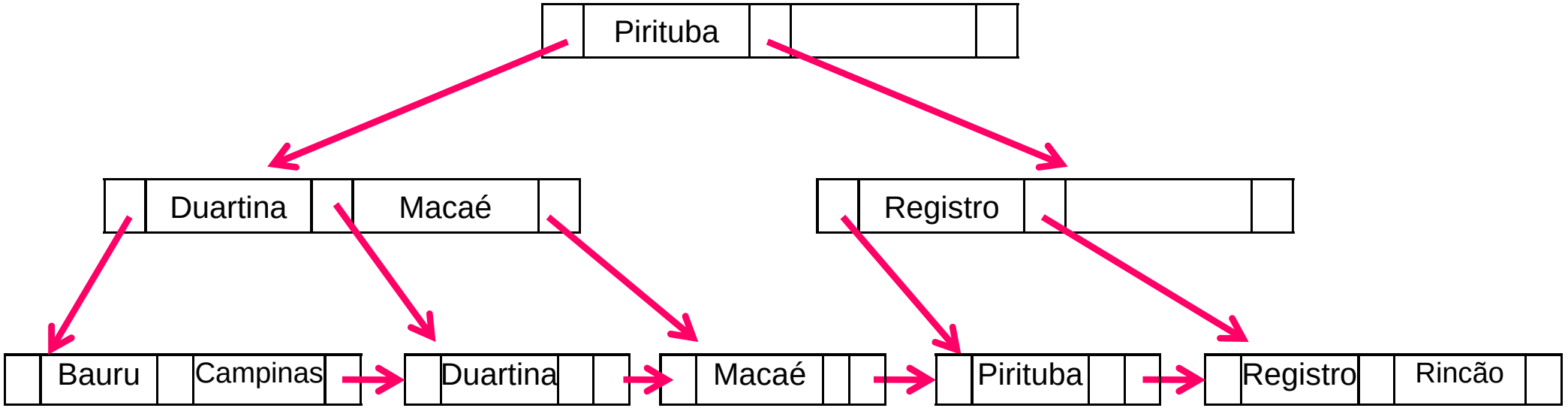
Índices Árvore-B⁺ ➔ Atualizações

✓ Exemplo: inserir Campinas na árvore dada:



Índices Árvore-B⁺ ➔ Atualizações

✓ Exemplo: inserir Campinas na árvore dada:



Índices Árvore-B$^+$ ➔ Atualizações

✓ Remoção (sem junção de nós)

- encontrar nó folha, conforme definido na busca
- excluir registro do arquivo
- remover valor da chave de procura do nó-folha se não houver *bucket* associado àquele valor ou se o *bucket* tornar-se vazio como resultado da remoção.

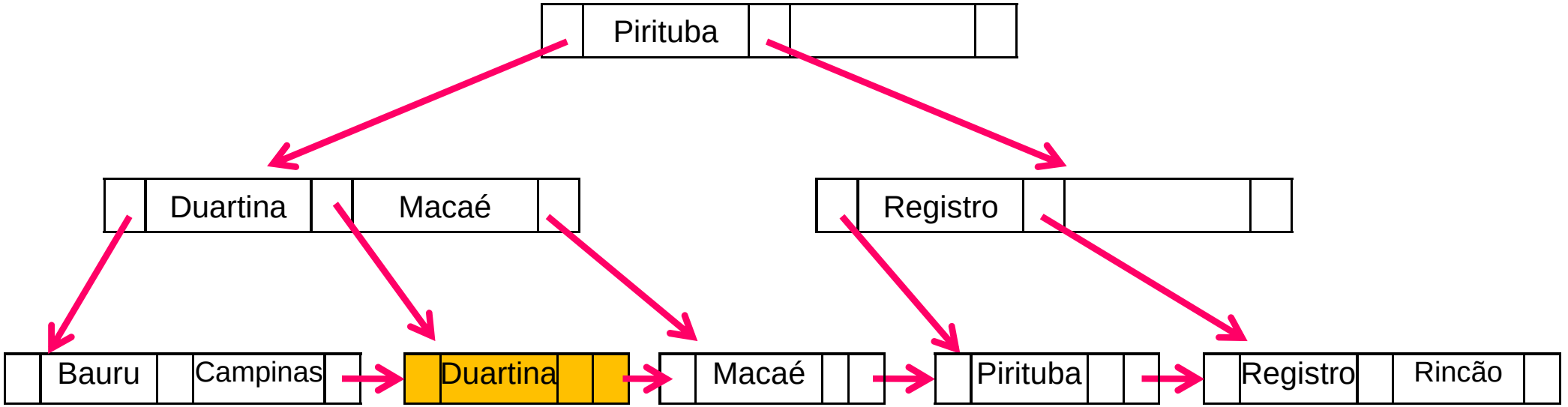
Índices Árvore-B$^+$ ➔ Atualizações

✓ Remoção (com junção de nós)

- encontrar nó folha, conforme definido na busca
- se folha fica vazia ou muito pequena, é necessário eliminar nó (lembrar regra da quantidade de nós)
  - remover do nó pai o ponteiro para o nó a ser eliminado
  - se o pai também ficar muito pequeno ou vazio
    - olhar o nó irmão (nó não folha que contém a chave de procura)
      - » se nó irmão tiver espaço ➔ fundir os nós
      - » se nó irmão não tiver espaço ➔ redistribuir os nós

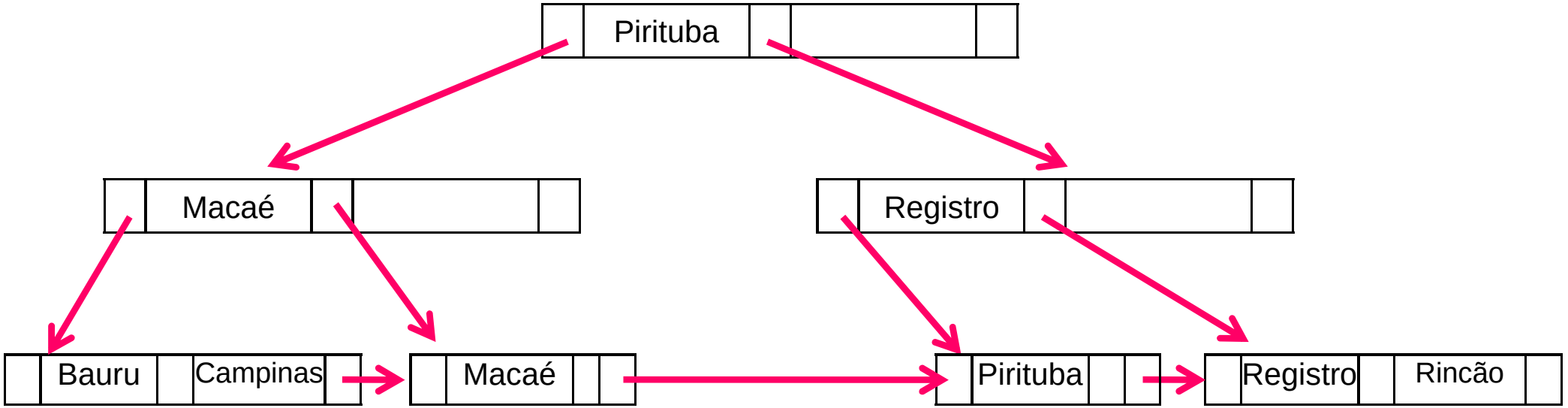
Índices Árvore-B⁺ ➔ Atualizações

✓ Exemplo: remover Duartina da árvore dada:



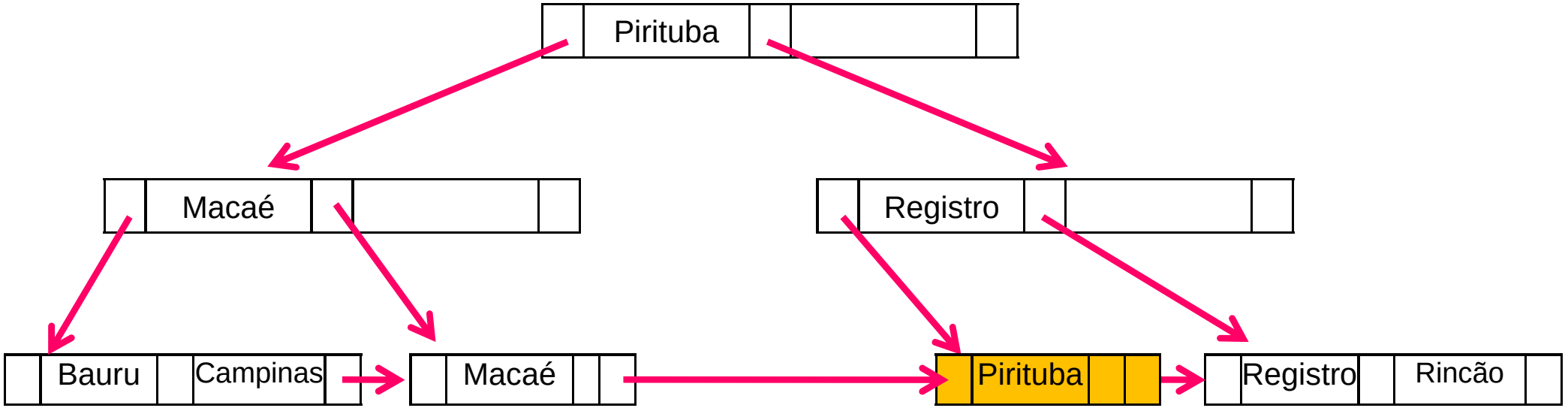
Índices Árvore-B⁺ ➔ Atualizações

✓ Exemplo: remover Duartina da árvore dada:



Índices Árvore-B⁺ ➔ Atualizações

✓ Exercício: remover Pirituba da árvore dada:



Árvore-B+ ➔ Organização de arquivo

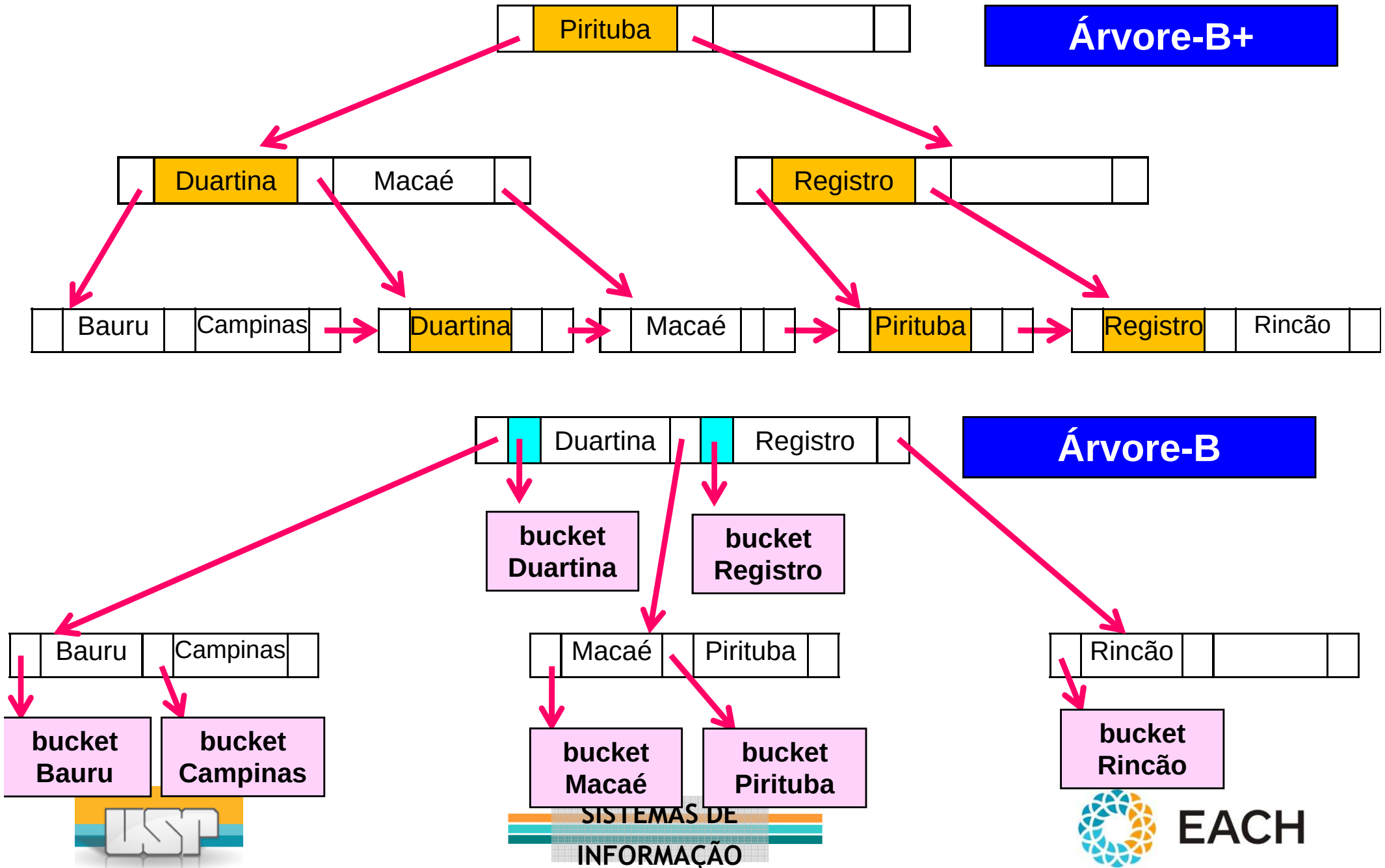
✓ Árvore-B+

- resolve o problema da degradação dos índices sequenciais
- não é usada somente como índice, mas como organizador dos registros do arquivo.
- nós folhas armazenam os próprios registros – e não ponteiros para eles
- registros normalmente são maiores que ponteiros:
  - número máximo de registros armazenados em um nó folha é menor que número de ponteiros em nó não folha
  - lembrar que nós folhas precisam estar completos pelo menos até metade
- inserção e remoção em uma organização de arquivo com Árvore-B+ são manipuladas da mesma forma que índices de Árvore-B+

Árvore-B ➔ Índices

- ✓ Índices Árvore-B:
  - semelhante índices Árvore-B$^+$
  - Diferença básica: Árvore-B elimina armazenamento redundante de valores de chaves de procura
    - podemos armazenar índice usando número menor de nós;
    - chaves de procura dos nós-folhas não aparecem em nenhuma outra parte da árvore-B:
      - temos que incluir um ponteiro adicional para cada chave de procura em um nó não-folha
      - ponteiros adicionais apontam para registros de arquivos ou para o *bucket* da chave de procura associada.

Árvore-B ➔ Índices



Árvore-B ➔ Índices

– estrutura de um nó folha típico (igual árvore B+):

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

– estrutura de um nó não folha típico

P_1	B_1	K_1	P_2	B_2	K_2	...	...	...	P_{m-1}	B_{m-1}	K_{m-1}	P_m

- Estrutura de nó folha é diferente de nó não folha

- Bi: ponteiro para *bucket* (que apontam para registros)

- **m < n** (cabe quantidade menor de chaves nos nós não folhas do que nos nós folha)

Árvore-B ➔ Índices

- ✓ Quantidade de nós acessados: depende de onde a chave de procura está localizada.
  - Árvore-B^{+}: sempre percorre da raiz até nó folha
  - Árvore-B: às vezes é possível achar chave de procura antes de atingir nó folha:
    - benefício pode ser pequeno porque podem ser armazenados muito mais chaves no nível de folha do que nos níveis não-folha.
  - Árvore-B tem *fanout* menor (profundidade maior que Árvore-B^{+})

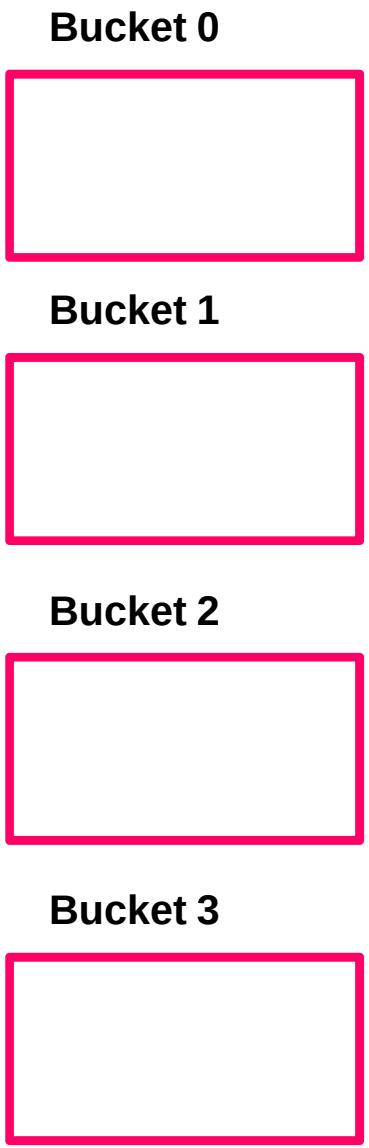
Árvore-B ➔ Índices

- ✓ Remoção:
  - mais complexa que Árvore-B$^+$
  - entrada removida pode aparecer em um nó não-folha.
  - valor correto para sua substituição deve ser escolhido da sub-árvore do nó que contém a chave removida.
  - podem ser necessárias ações posteriores para rearranjar a sub-árvore se o nó folha tiver poucas entradas
- ✓ Inserção:
  - ligeiramente mais complexa que Árvore-B$^+$
- ✓ Árvore-B versus Árvore-B$^+$:
  - vantagens de espaço das Árvore-B são poucas para grandes índices e normalmente não superam desvantagens citadas
  - Árvore-B$^+$ é mais simples e, por isso, preferida por muitos projetistas de BD.

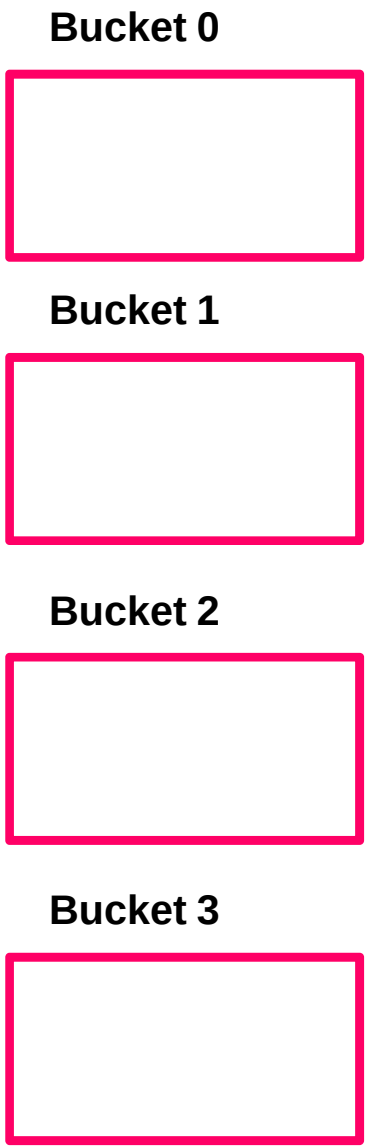
Hashing ➔ Organização de arquivos

- ✓ Organização sequencial de um arquivo ➔ necessária estrutura de índice ou busca binária ➔ mais operações de I/O
- ✓ *Hashing* ➔ permite evitar acesso a estruturas de índices
- ✓ *Hashing* também permite meio para construir índices
- ✓ Organização de arquivos em *Hashing*
  - obtém diretamente o endereço do bloco de disco que contém um registro desejado usando uma função sobre o valor da chave de procura;
  - *bucket* ➔ tipicamente um bloco de disco, mas pode ser menor ou maior que o bloco.

Hashing ➔ Organização de arquivos



Hashing ➔ Organização de arquivos



Exemplo de chave

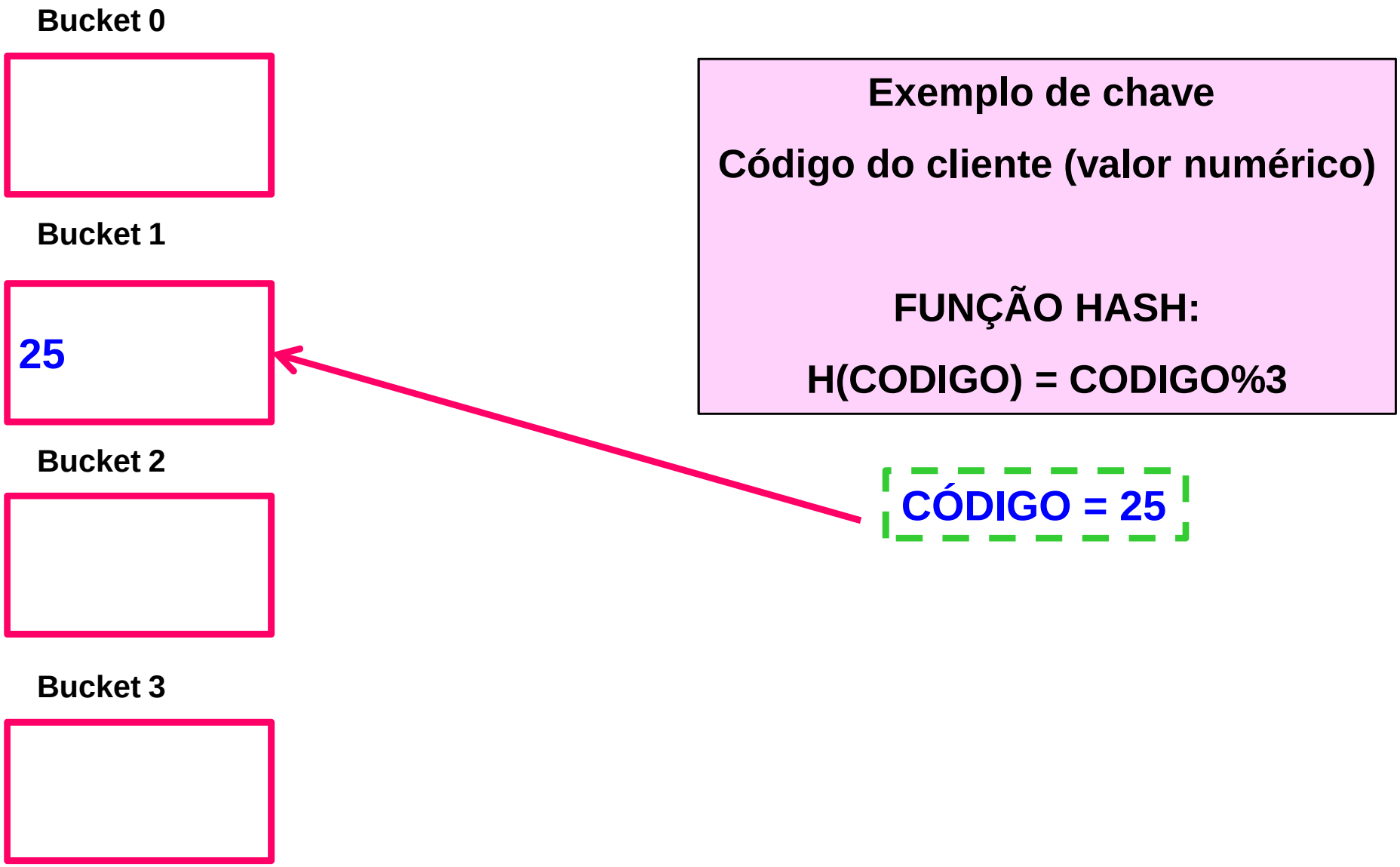
Código do cliente (valor numérico)

FUNÇÃO HASH:

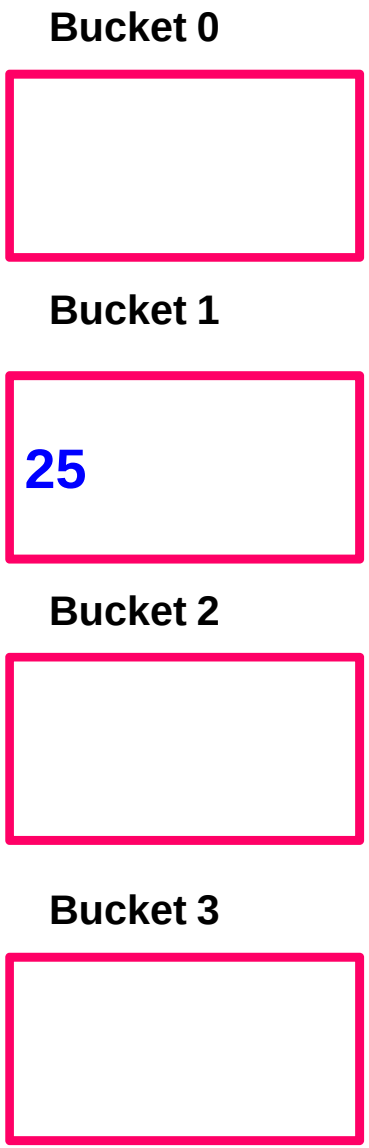
H(CODIGO) = CODIGO%3

CÓDIGO = 25

Hashing ➔ Organização de arquivos



Hashing ➔ Organização de arquivos



Exemplo de chave

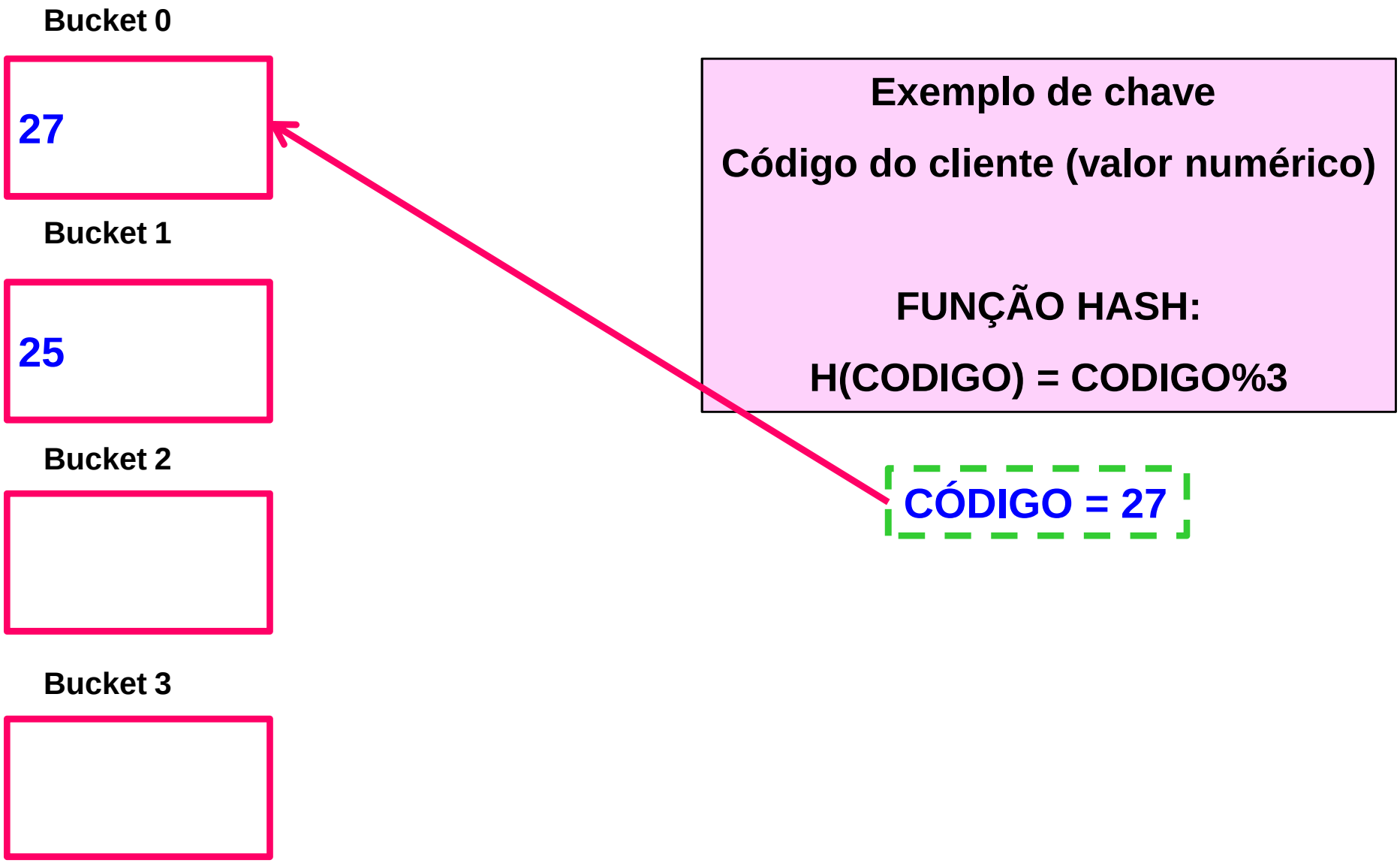
Código do cliente (valor numérico)

FUNÇÃO HASH:

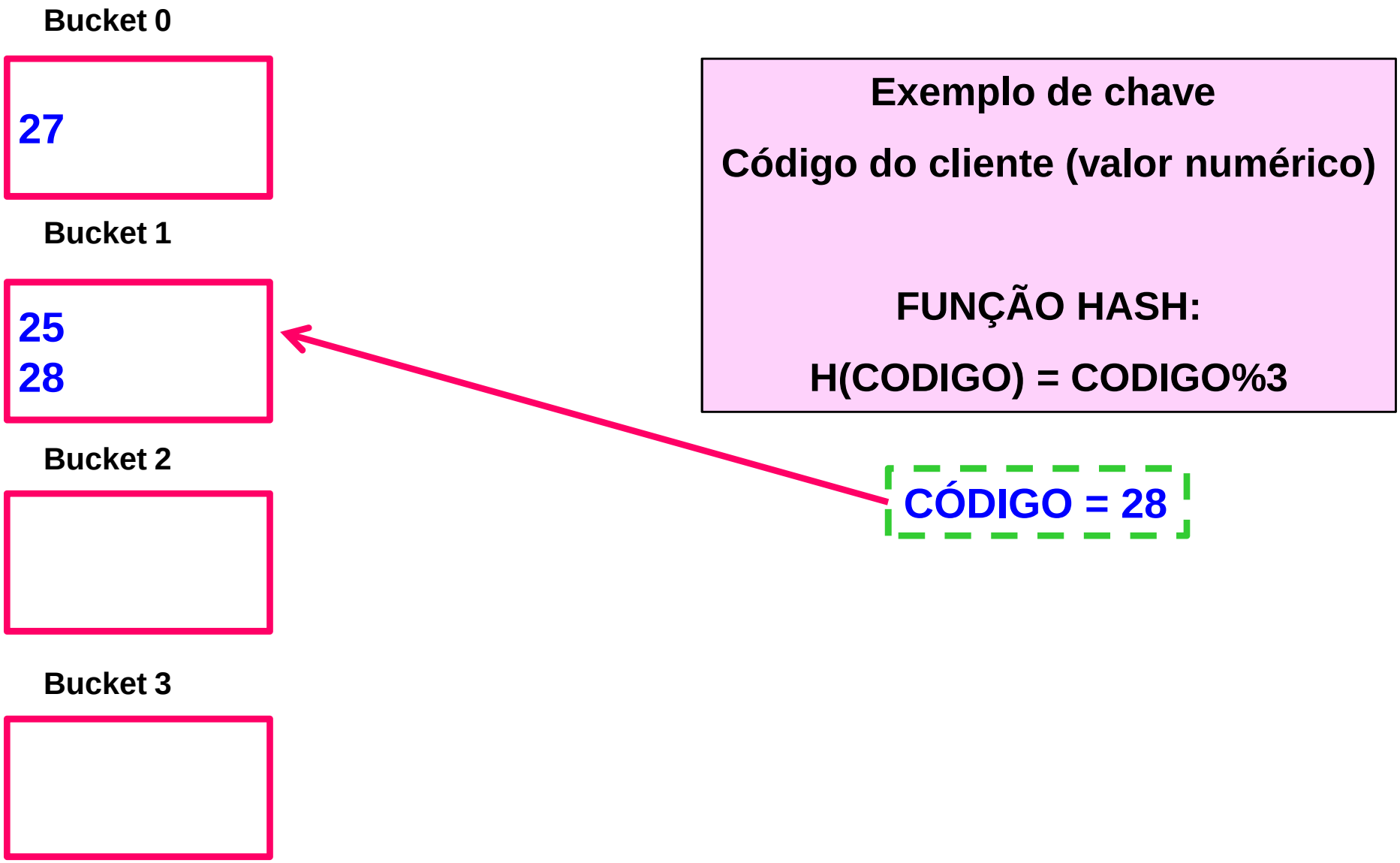
H(CODIGO) = CODIGO%3

CÓDIGO = 27

Hashing ➔ Organização de arquivos



Hashing ➔ Organização de arquivos



Hashing ➔ Organização de arquivos

- ✓ Definimos:
  - K = conjunto de todos os valores de chaves de procura
  - B = todos os endereços de *bucket*
  - *Função* ***hash h*** = função de K para B
- ✓ Inserção de um registro com chave de procura K_i
  - calcula $h(K_i)$ ➔ fornece endereço do *bucket* para aquele registro
  - registro é armazenado no *bucket,* se houver espaço
- ✓ Procura de um registro com chave de procura K_i
  - calcula $h(K_i)$ ➔ fornece endereço do *bucket* para aquele registro
  - conferir o valor da chave de procura de todos os registros no *bucket* para verificar se o registro é um dos desejados **(Por quê ???)**

Hashing ➔ Organização de arquivos

- ✓ Definimos:
  - K = conjunto de todos os valores de chaves de procura
  - B = todos os endereços de *bucket*
  - *Função* ***hash h*** = função de K para B
- ✓ Inserção de um registro com chave de procura K_i
  - calcula $h(K_i)$ ➔ fornece endereço do *bucket* para aquele registro
  - registro é armazenado no *bucket,* se houver espaço
- ✓ Procura de um registro com chave de procura K_i
  - calcula $h(K_i)$ ➔ fornece endereço do *bucket* para aquele registro
  - conferir o valor da chave de procura de todos os registros no *bucket* para verificar se o registro é um dos desejados (porque duas chaves de procura podem ter o mesmo valor de *hash*)

Hashing ➔ Organização de arquivos

- ✓ Remoção de um registro com chave de procura K_i
  - calcula $h(K_i)$ ➔ fornece endereço do *bucket* para aquele registro
  - registro é removido do *bucket*

Hashing ➔ Organização de arquivos

- ✓ Funções *hash* requerem projeto cuidadoso
- ✓ piores funções ➔ mapeiam todos os valores de chave de procura para o mesmo *bucket*
- ✓ função ideal ➔ distribui chaves uniformemente entre os *buckets*
- ✓ O que se deseja:
  - distribuição uniforme
  - distribuição aleatória ➔ valor de *hash* não relacionado a qualquer ordem visível externamente de valores de chaves de procura

Hashing ➔ Organização de arquivos

- ✓ Exemplo:
  - vamos definir função *hash* para mapear o arquivo de contas dos clientes por cidade
  - Quantos *buckets?* Alguma sugestão?

Hashing ➔ Organização de arquivos

- ✓ Exemplo:
  - vamos definir função *hash* para mapear o arquivo de contas dos clientes por cidade
  - *26 buckets* (um para cada letra do alfabeto): será uniforme?

Hashing ➔ Organização de arquivos

- ✓ Exemplo:
  - vamos definir função *hash* para mapear o arquivo de contas dos clientes por cidade
  - *26 buckets* (um para cada letra do alfabeto): será uniforme?

NÃO!

Hashing ➔ Organização de arquivos

- ✓ Funções *hash* típicas
  - executam cálculos sobre a representação binária interna à máquina de caracteres da chave de procura
  - Exemplo:
    - 1) calcula a soma das representações binárias dos caracteres de uma chave
    - 2) retorna o resto da soma dividido pelo número de *buckets*

Considerando 10 buckets, como ficariam distribuídos os registros abaixo supondo que a iésima letra do alfabeto é representada pelo inteiro i?

Cidade	Nome	Endereço
Avaí	Júlia	Rua dos Anjos, 123
Bauru	Ana	Av. Antonio, 865
Brasília	Bruno	Av. Antonio, 865
Brasília	Maria	Rua Estreita, 89
Petrópolis	Carlos	Alameda das Rosas, 634
Petrópolis	Maria	Rua São Paulo, 432
Rondonópolis	Luiza	Avenida Dom Pedro, 800
Santos	Bruno	Rua Aparecida, 7600
Santos	Maria	Rua Santa Rita, 632

EACH

Hashing ➔ Organização de arquivos

- ✓ Funções *hash* típicas
  - executam cálculos sobre a representação binária interna à máquina de caracteres da chave de procura
  - Exemplo:
    - 1) calcula a soma das representações binárias dos caracteres e uma chave
    - 2) retorna o resto da soma dividido pelo número de *buckets*

Considerando 10 buckets, como ficariam distribuídos os registros abaixo supondo que a iésima letra do alfabeto é representada pelo inteiro i?

1 + 22 + 1 + 9 =
33 % 10 = 3

Cidade	Nome	Endereço
Avaí	Júlia	Rua dos Anjos, 123
Bauru	Ana	Av. Antonio, 865
Brasília	Bruno	Av. Antonio, 865
Brasília	Maria	Rua Estreita, 89
Petrópolis	Carlos	Alameda das Rosas, 634
Petrópolis	Maria	Rua São Paulo, 432
Rondonópolis	Luiza	Avenida Dom Pedro, 800
Santos	Bruno	Rua Aparecida, 7600
Santos	Maria	Rua Santa Rita, 632

Hashing ➔ Organização de arquivos

- ✓ Se *bucket* não tem espaço para armazenar registro ➔ overflow de *bucket*
- ✓ Razões:
  - *buckets* insuficientes ➔ número de *buckets* (n_B) deve ser > n_r / f_r
    - n_r = número total de registros armazenados
    - f_r = número de registros que cabem no *bucket*
  - desequilíbrio (*skew*) ➔ mais registros para alguns *buckets*
    - registros múltiplos podem ter a mesma chave de procura
    - função *hash* pode resultar em distribuição não uniforme de chaves de procura

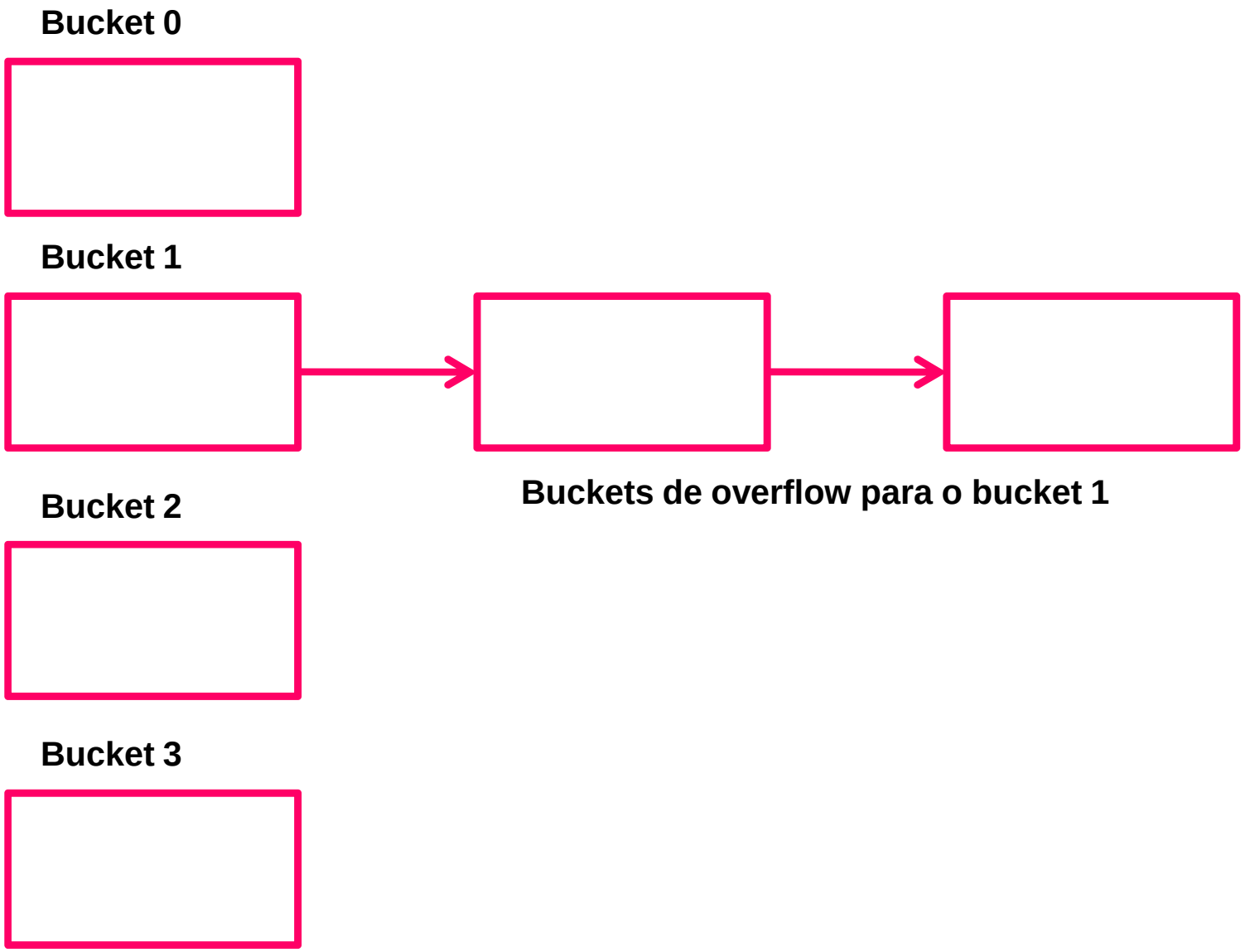
Hashing ➔ Organização de arquivos

- ✓ Para reduzir probabilidade de overflow:
  - escolha do número de ➔ $(n_r / f_r)*(1+d)$
    - d = fator de *fudge* – tipicamente ao redor 0,2
    - aproximadamente 20% do espaço dos *buckets* será perdido

Hashing ➔ Organização de arquivos

- ✓ Mesmo com cálculo anterior, overflow ainda pode acontecer.
- ✓ Solução ➔ *bucket* de overflow
  - se *bucket* ***b*** está cheio ao inserir um registro ➔ este é armazenado no *bucket* de overflow
  - se *bucket* de overflow cheio ➔ novo *bucket* de overflow
  - *buckets* de overflow de um *bucket* são encadeados em uma lista ligada
  - manipulação desse tipo de lista ➔ encadeamento de overflow

Hashing ➔ Organização de arquivos



Hashing ➔ Organização de arquivos

- ✓ Procura de um registro com o conceito de overflow
  - calcula $h(K_i)$ ➔ fornece endereço do *bucket* para aquele registro
  - conferir o valor da chave de procura de todos os registros no *bucket* para verificar se o registro é um dos desejados
  - se *bucket* ***b*** possuir *buckets* de overflow ➔ todos registros de todos os *buckets* de overflow de ***b*** devem ser examinados

- ✓ Estrutura vista até agora ➔ hashing fechado

Hashing ➔ Organização de arquivos

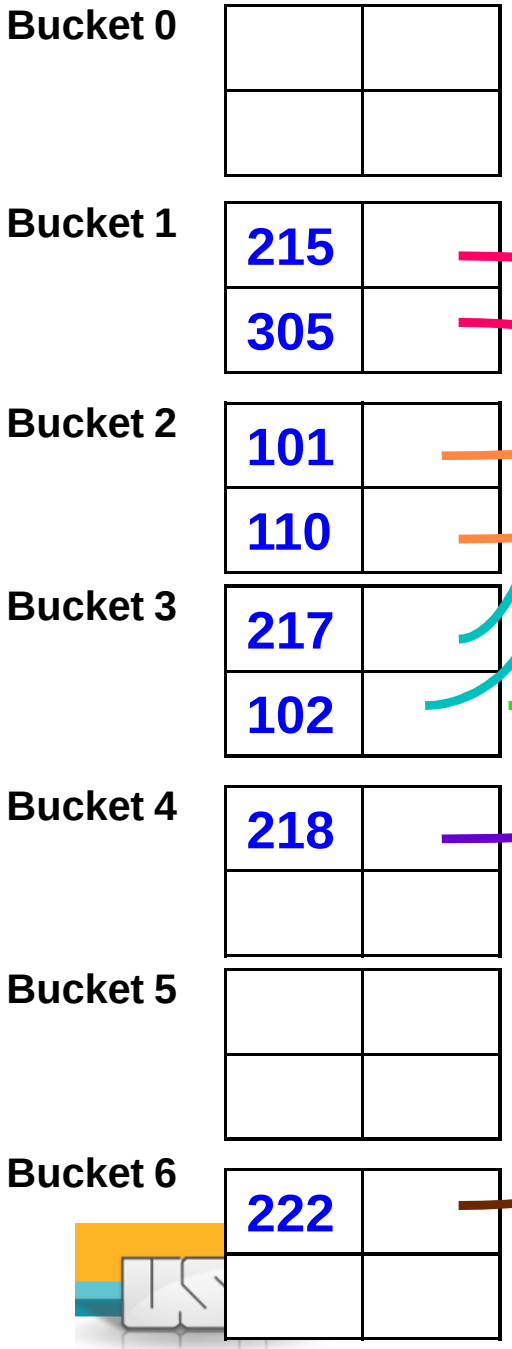
- ✓ hashing aberto ➔ conjunto fixo de *buckets* e não há cadeia de overflow
  - são estabelecidas políticas para escolher novo *bucket* quando um *bucket* está cheio
  - Exemplo de política ➔ registros inseridos em outro *bucket* que tem espaço (ordem cíclica) – *linear probing*
  - outras políticas: calcular funções *hash* adicionais
  - **SGBD preferem *hashing* fechado. Por quê?**

Hashing ➔ Organização de arquivos

- ✓ hashing aberto ➔ conjunto fixo de *buckets* e não há cadeia de overflow
  - são estabelecidas políticas para escolher novo *bucket* quando um *bucket* está cheio
  - Exemplo de política ➔ registros inseridos em outro *bucket* que tem espaço (ordem cíclica) – *linear probing*
  - outras políticas: calcular funções *hash* adicionais
  - **SGBD preferem *hashing* fechado. Por quê?**
    - **Remoção no *hashing* aberto é complicada.**

Hashing ➔ Índices

- ✓ Organiza as chaves de procura, com seus ponteiros associados, em uma estrutura de arquivo *hash* **(exemplo considerando K=número da conta)**



h(K)=(soma dígitos de K) % 7

Cidade	Conta	Endereço
Avai	217	Rua dos Anjos, 123
Bauru	101	Av. Antonio, 865
Brasília	110	Av. Antonio, 865
Brasília	215	Rua Estreita, 89
Petrópolis	102	Alameda das Rosas, 634
Petrópolis	201	Rua São Paulo, 432
Rondonópolis	218	Avenida Dom Pedro, 800
Santos	222	Rua Aparecida, 7600
Santos	305	Rua Santa Rita, 632

Hashing ➔ Índices

- ✓ Termo índice *hash* ➔ denota estruturas de arquivo *hash* e também os índices *hash* secundários.

- ✓ Índice *hash* nunca é necessário como uma estrutura de índice primária ➔ se um arquivo é organizado usando *hashing* não é necessária estrutura de índice *hash* para ele.

Hashing ➔ Hashing dinâmico

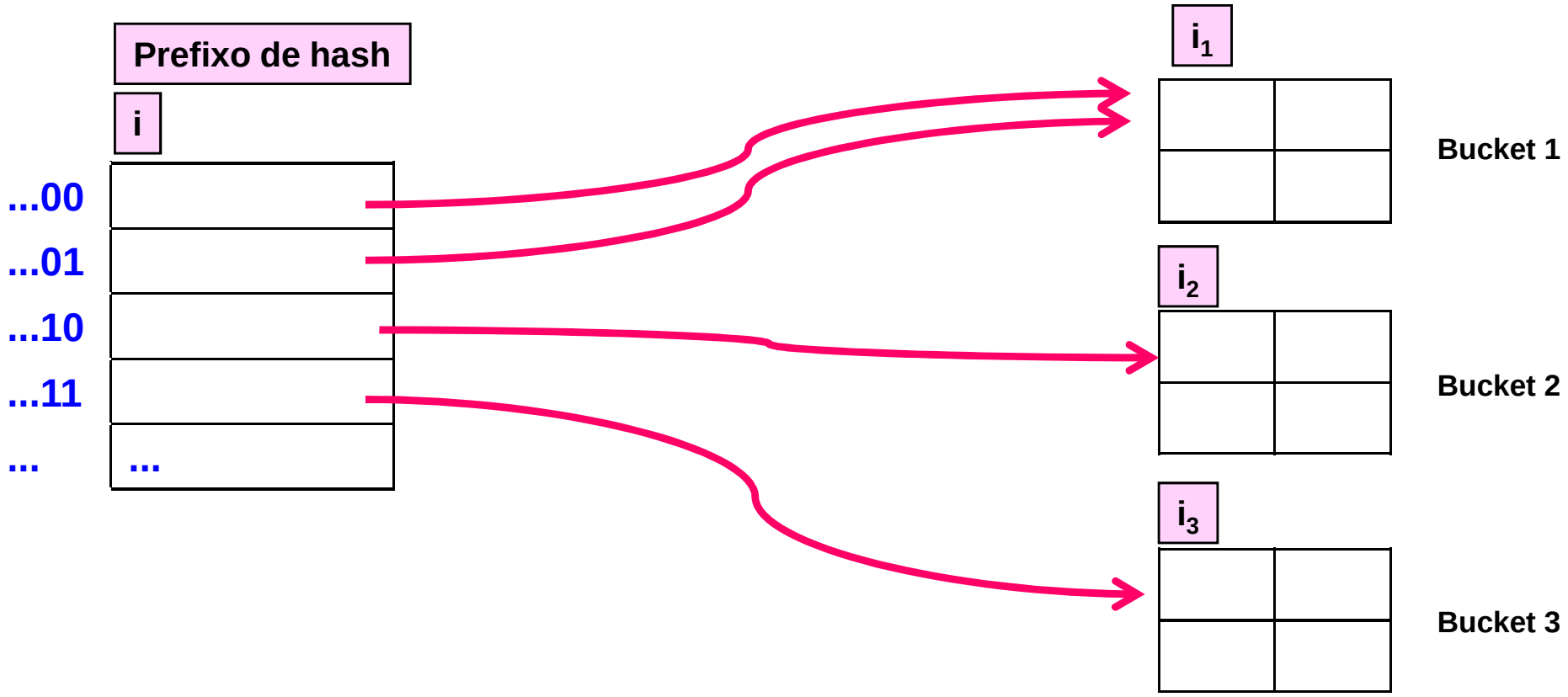
- ✓ Problema do Hashing estático ➔ definição do número de *buckets*
- ✓ Opções:
  - escolher função *hash* com base no tamanho atual do arquivo ➔ degradação de desempenho à medida que BD cresce
  - escolher função *hash* com base no tamanho previsto do arquivo ➔ desperdício de espaço
  - reorganizar periodicamente estrutura de *hash*, escolher função *hash* nova e gerar novas atribuições de *bucket* ➔ operação demorada (proibição de acesso durante a operação)
- ✓ Técnicas de hashing dinâmico ➔ permite modificar função *hash* dinamicamente para acomodar crescimento ou diminuição do BD.
- ✓ Uma das formas: hashing expansível (ou extensível)

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Trata mudança no tamanho do BD por meio de divisão e fusão de *buckets*
  - eficiência espacial mantida
  - reorganização realizada apenas em um *bucket* por vez ➔ overhead de desempenho aceitável e baixo
- ✓ Como funciona:
  - escolhe função *hash* h com propriedades desejadas de uniformidade e aleatoriedade
  - função gera valores dentro de grande faixa – inteiros binários de ***b*** bits (valor típico para b = 32)
  - 2^{32} ➔ mais de 4 bilhões de *buckets* – impossível!
  - o que se faz: criação de *buckets* **por demanda**.Não usa os ***b*** bits, mas ***i*** bits, sendo $0 \leq i \leq b$
  - valor de ***i*** cresce ou diminui de acordo com o tamanho do BD

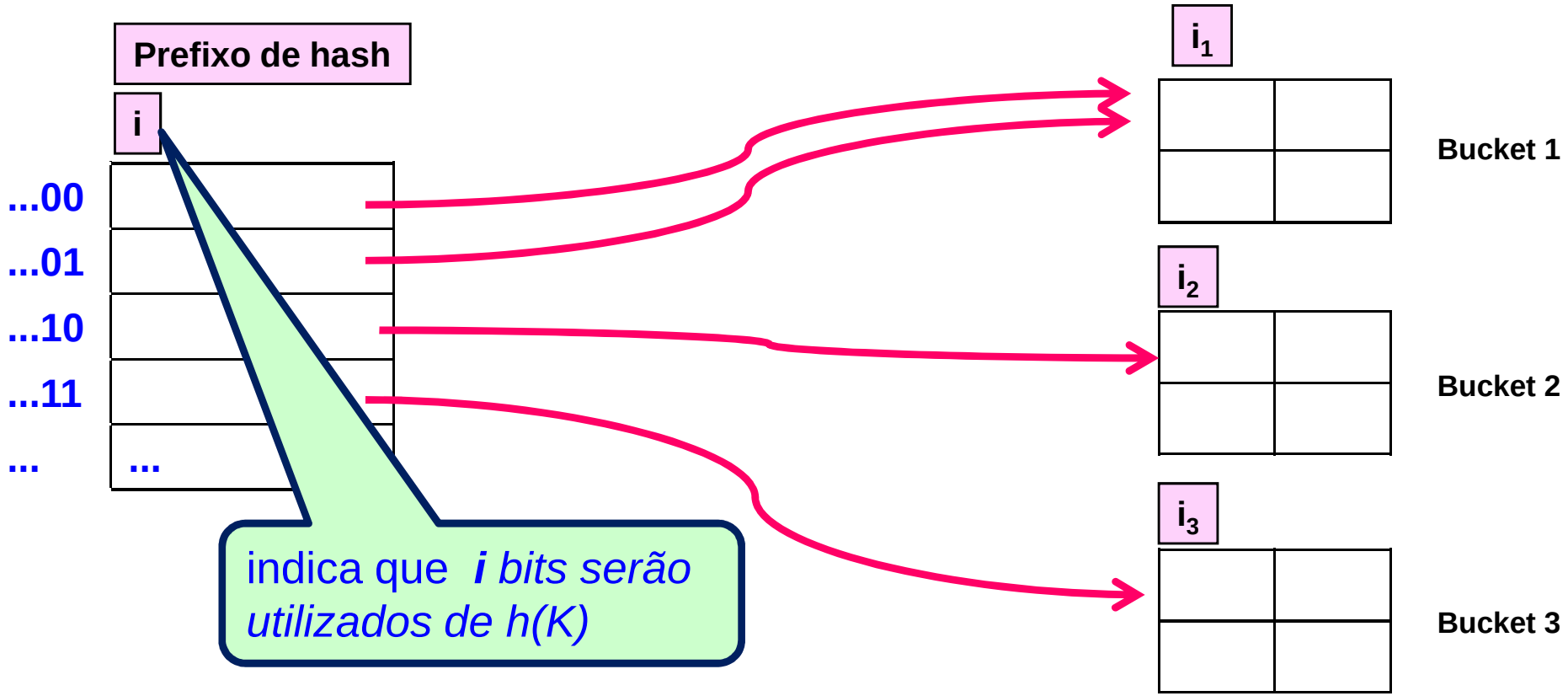
Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Como funciona:
  - o que se faz: criação de *buckets* por demanda.Não usa os ***b*** bits, mas ***i*** bits, sendo $0 \leq i \leq b$
  - valor de ***i*** cresce ou diminui de acordo com o tamanho do BD



Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Como funciona:
  - o que se faz: criação de *buckets* por demanda.Não usa os ***b*** bits, mas ***i*** bits, sendo $0 \leq i \leq b$
  - valor de ***i*** cresce ou diminui de acordo com o tamanho do BD



Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Procura de um registro com chave de procura K_1
  - pegar primeiros *i* bits de maior ordem de $h(K_1)$
  - verifica-se entrada da tabela correspondente para a sequência de bits
  - segue ponteiro de *bucket* na entrada da tabela
- ✓ Inserção de um registro com chave de procura K_1
  - mesmo procedimento da procura, parando no *bucket j*
  - se há espaço em j, insere registro no *bucket*
  - se *j* está cheio ➔ dividir *j* e redistribuir registros atuais, mais novo registro
  - Para dividir *bucket* ➔ determinar se precisa aumentar o número de bits utilizado

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Inserção de um registro com chave de procura K_1
  - Para dividir *bucket* ➔ determinar se precisa aumentar o número de bits utilizado
    - Incrementa o valor de i em 1 ➔ dobra tamanho da tabela de endereços de *bucket*
    - Cada entrada é substituída por duas, ambas com mesmo ponteiro da entrada original (*bucket j*)
    - Aloca novo *bucket* (*bucket z*)
    - Configura segunda entrada para apontar para novo *bucket*
    - Atribui *i* a i_j e i_z
    - Recalcula valor *hash* de cada registro no *bucket j* a fim de verificar se fica no *bucket j* ou *z*

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

Valor de *hash* de 32 bits para a chave nome_cidade

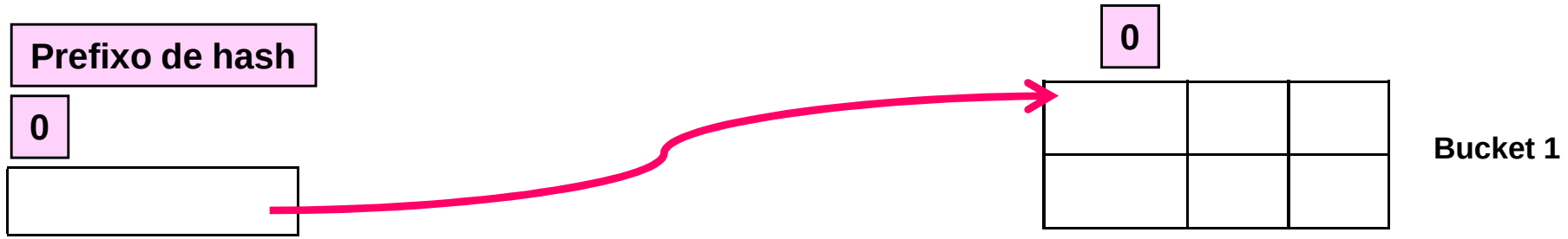
Considerando que cabem 2 registros de dados em cada bucket

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



Estrutura inicial (vazia)

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010					
Rincão	1101	1000	0011					

Quantidade de bits (*i*) usados.Várias entradas consecutivas da tabela podem apontar para o mesmo endereço.

A cada **bucket** é associado um prefixo, que pode ser menor que ***i.***

Estrutura inicial (vazia)

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

Estrutura após inserir 2 registros

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

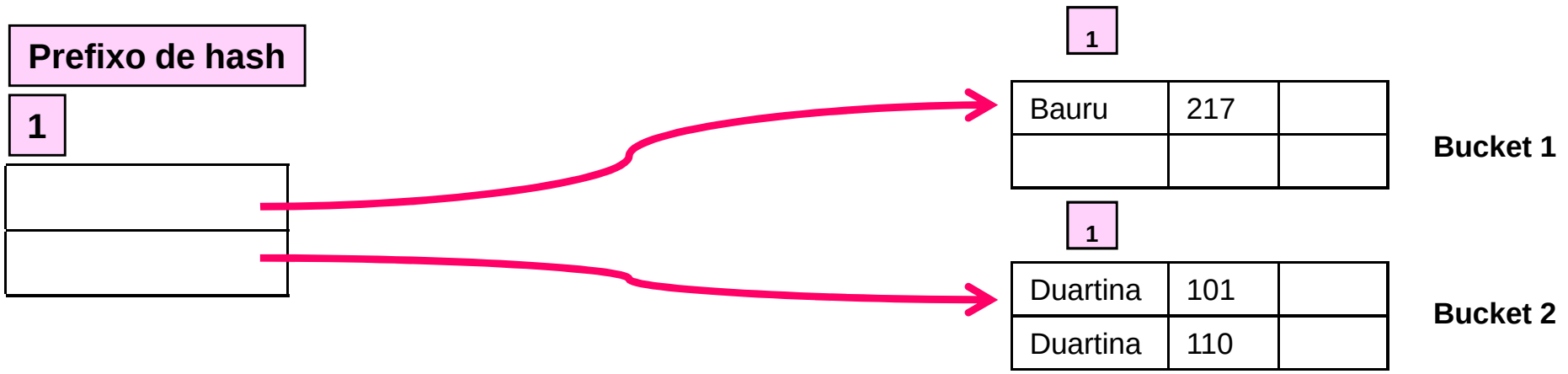
Terceiro registro não cabe no *bucket.* Tem que aumentar número de bits, dividir o *bucket. Dobra quantidade de entradas na tabela de endereços.* Recalcular *hash* dos registros

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



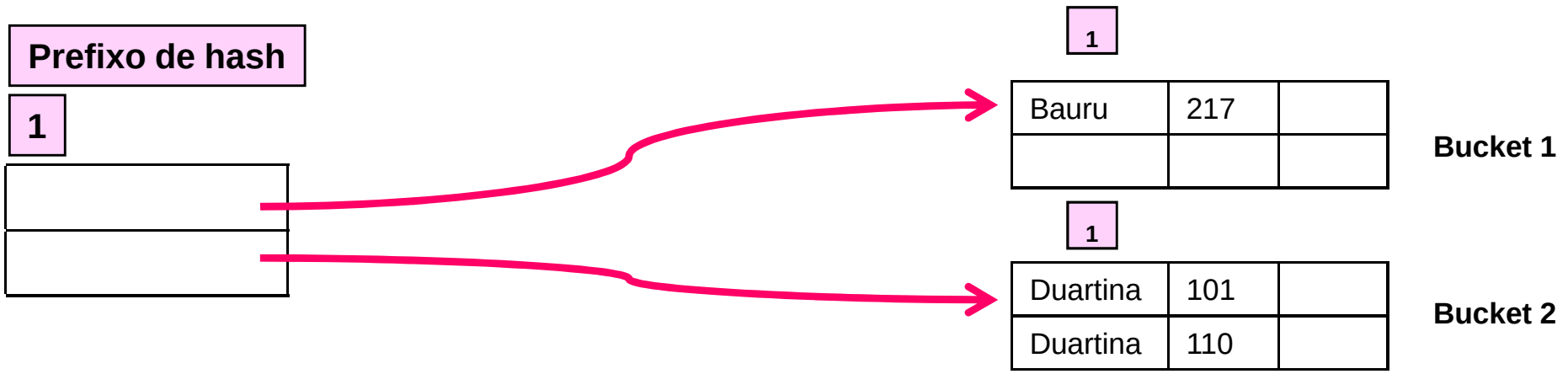
Estrutura após inserir 3 registros

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



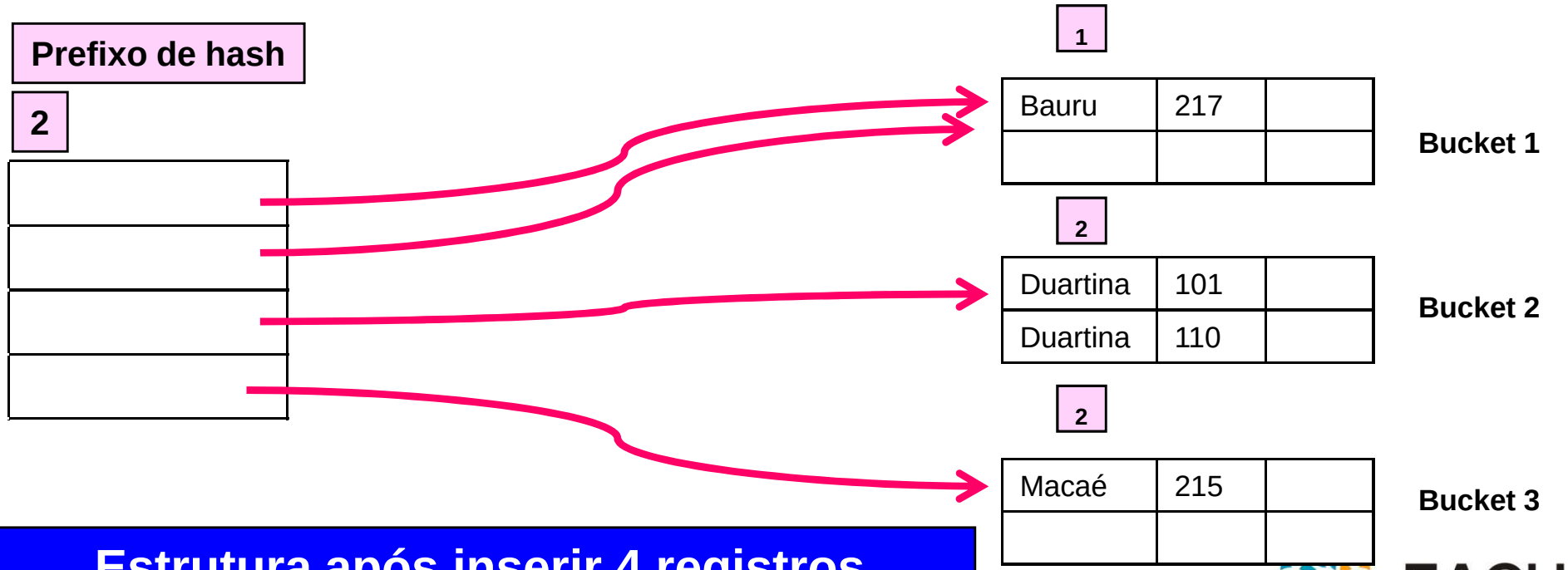
Não consigo inserir Macaé = Buffer cheio (1º bit = 1). Tem que aumentar número de bits e repetir o processo (dividir o último *bucket*).

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



Estrutura após inserir 4 registros

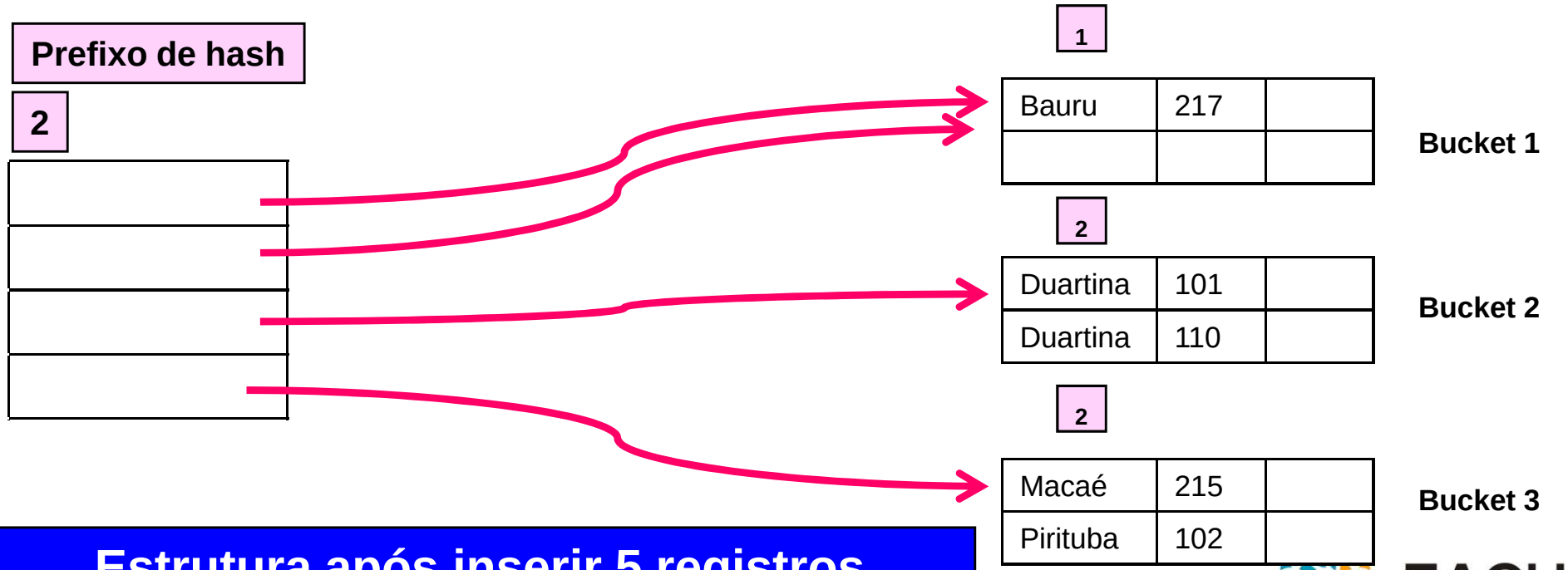
EACH

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



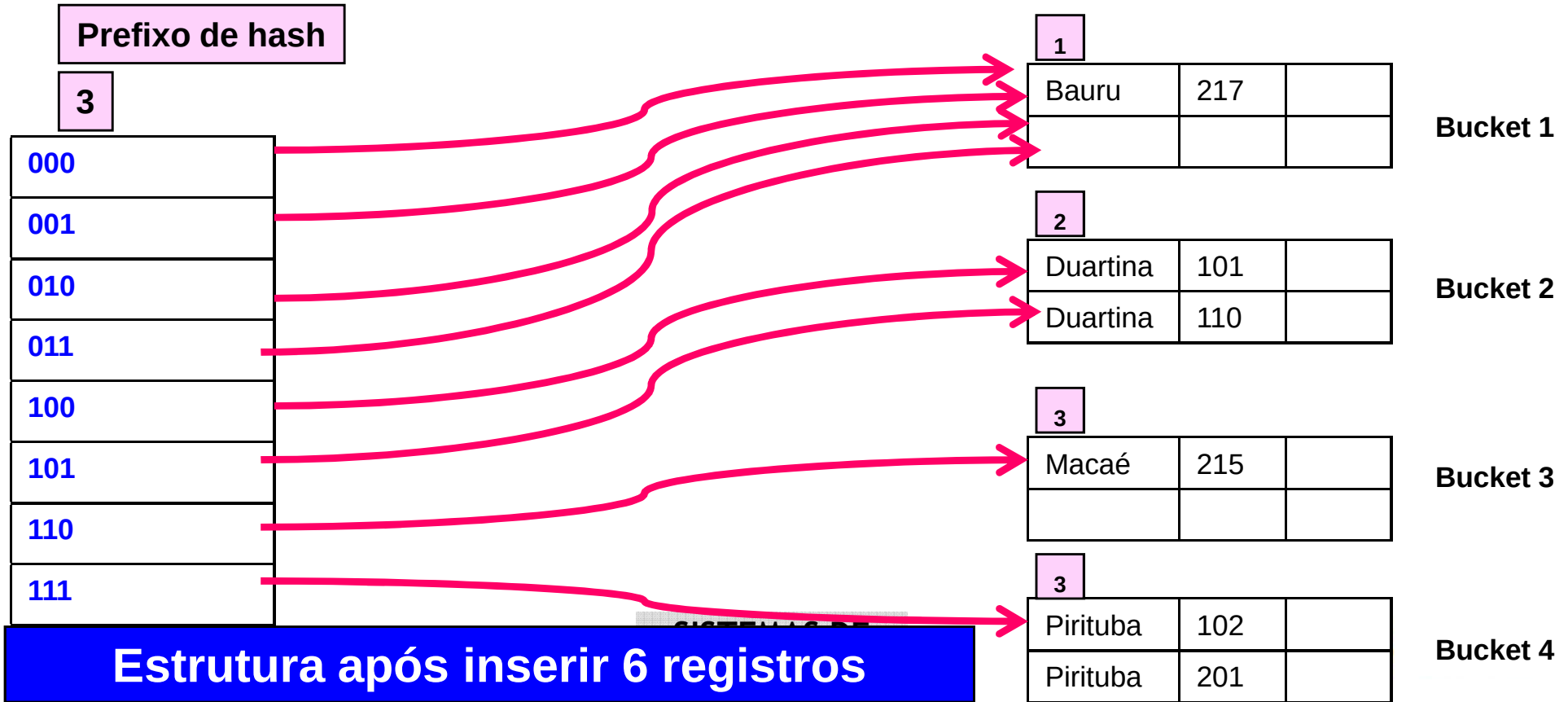
Estrutura após inserir 5 registros

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



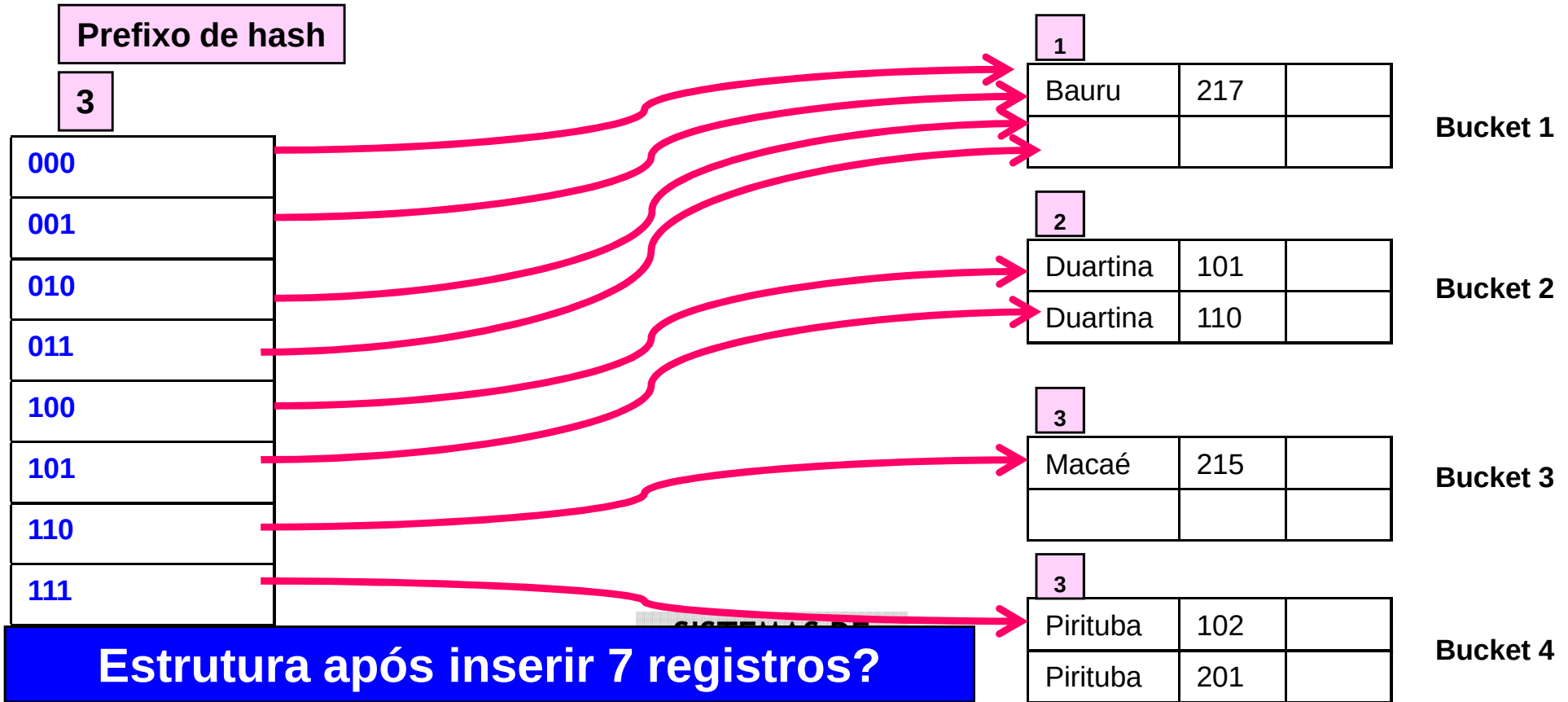
Estrutura após inserir 6 registros

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001

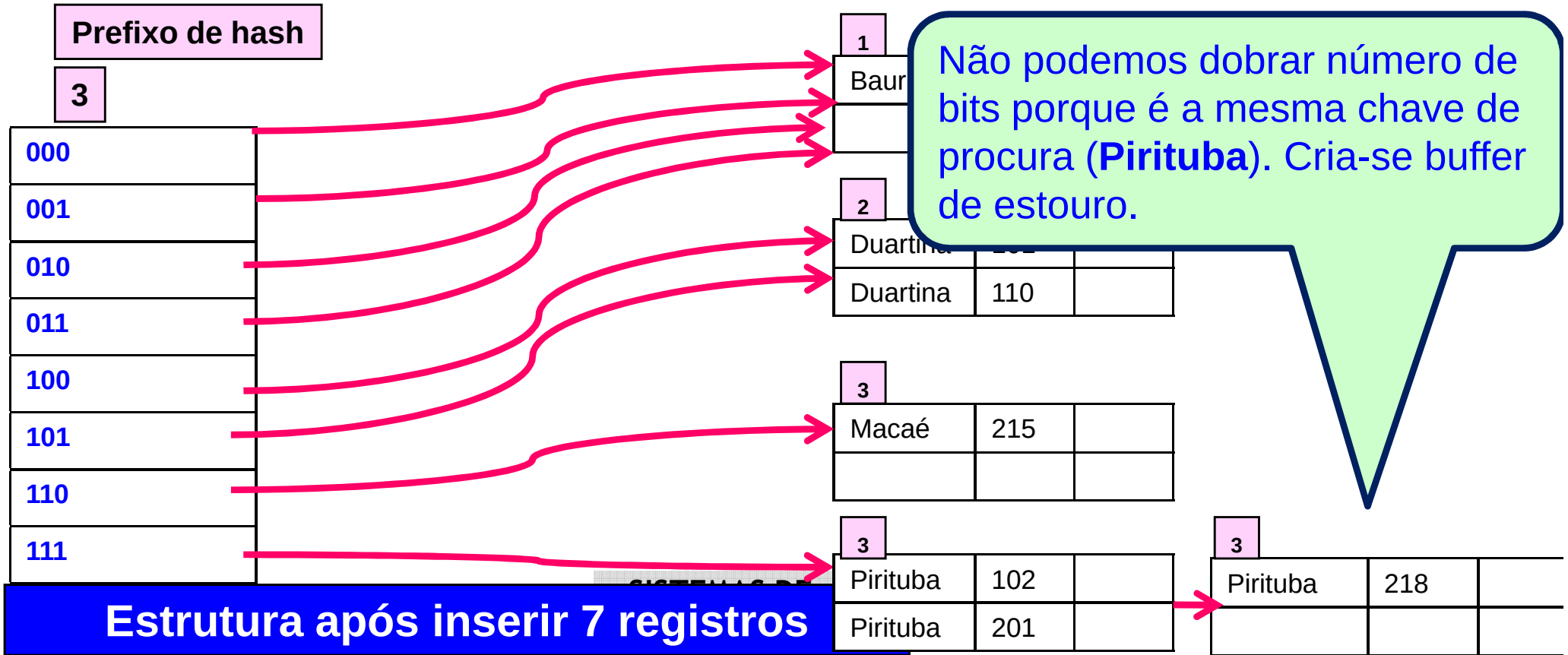


Hashing ➔ Hashing dinâmico ➔ Hashing expansível

✓ Inserção de um registro com chave de procura K_1

Cidade	Conta	Endereço
Bauru	217	Rua dos Anjos, 123
Duartina	101	Av. Antonio, 865
Duartina	110	Av. Antonio, 865
Macaé	215	Rua Estreita, 89
Pirituba	102	Alameda das Rosas, 634
Pirituba	201	Rua São Paulo, 432
Pirituba	218	Avenida Dom Pedro, 800
Registro	222	Rua Aparecida, 7600
Rincão	305	Rua Santa Rita, 632

Cidade	h(cidade)							
Bauru	0010	1101	1111	1011	0010	1100	0011	0000
Duartina	1010	0011	1010	0000	1100	0110	1001	1111
Macaé	1100	0111	1110	1101	1011	1111	0011	1010
Pirituba	1111	0001	0010	0100	1001	0011	0110	1101
Registro	0011	0101	1010	0110	1100	1001	1110	1011
Rincão	1101	1000	0011	1111	1001	1100	0000	0001



Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Remoção de um registro com chave de procura K_1
  - mesmo procedimento da procura, parando no *bucket j*
  - remove chave de procura de *j* e o registro do arquivo
  - remover *j* se ficar vazio
  - se *buckets* forem fundidos ➔ tamanho da tabela de endereço de *bucket* pode ser cortado pela metade

Hashing ➔ Hashing dinâmico ➔ Hashing expansível

- ✓ Vantagens:
  - desempenho não se degrada com crescimento do arquivo
  - overhead mínimo
  - tabela de endereços contém só um endereço para cada valor *hash*
  - nenhum *bucket* reservado para crescimento futuro: economia de espaço
- ✓ Desvantagens:
  - procura envolve nível de acesso indireto adicional
  - maior complexidade na implementação

Indexação Ordenada X Hashing

- ✓ Maioria dos sistemas de BD usa somente algumas ou uma única forma de indexação ordenada ou *hashing*
- ✓ Aspectos para que o desenvolvedor de BD escolha:
  - custo da reorganização periódica do índice ou da organização *hash*
  - frequência de inserções e remoções
  - se compensa otimizar tempo médio de acesso às custas do aumento do tempo de acesso no pior caso
  - tipos de consultas com maior probabilidade de ocorrência

Indexação Ordenada X Hashing

- ✓ Maioria dos sistemas de BD usa somente algumas ou uma única forma de indexação ordenada ou *hashing*
- ✓ Aspectos para que o desenvolvedor de BD escolha:
  - custo da reorganização periódica do índice ou da organização *hash*
  - frequência de inserções e remoções
  - se compensa otimizar tempo médio de acesso às custas do aumento do tempo de acesso no pior caso
  - tipos de consultas com maior probabilidade de ocorrência

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai = c
```

Qual a diferença ???

```
select A1, A2, ..., An
from r
where Ai <= c2 and Ai >= c1
```

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai = c
```

Sistema procura pelo valor *c* no atributo *Ai*

Qual indexação é preferível? Ordenada ou Hashing?

Por quê???

Indexação Ordenada X Hashing

✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai = c
```

Sistema procura pelo valor *c* no atributo *Ai*

Qual indexação é preferível? Ordenada ou Hashing?

Hashing preferível (tempo constante)

Busca ordenada: tempo=proporcional log do número de valores de *c* no atributo *Ai*

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai <= c2 and Ai >= c1
```

Como seria a procura para esta consulta usando índice ordenado?

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai <= c2 and Ai >= c1
```

Como seria a procura para esta consulta usando índice ordenado?

Primeiro busca c1. Quando encontrar *bucket* com valor = c1, segue cadeia de ponteiros até achar c2.

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai <= c2 and Ai >= c1
```

Como seria a procura para esta consulta usando índice hash?

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai <= c2 and Ai >= c1
```

Como seria a procura para esta consulta usando índice hash?

Acha bucket com valor c1.

E fica difícil seguir ponteiros até *bucket* para achar c2.

Então, qual o melhor neste caso?

Indexação Ordenada X Hashing

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select A1, A2, ..., An
from r
where Ai <= c2 and Ai >= c1
```

Como seria a procura para esta consulta usando índice hash?

Acha bucket com valor c1.

E fica difícil seguir ponteiros até *bucket* para achar c2.

Então, qual o melhor neste caso?

Índice ordenado !!!

SQL e índices

- ✓ **SQL padrão não provê que usuário ou DBA controle formas de construir índices**
- ✓ **SGBD pode decidir automaticamente quais índices criar. Como isso não é fácil, é permitido que usuário crie e remova índices:**
- ✓ **Criar um índice:**

  create [unique] index <nome_indice> on <nome_relação> (<lista de atributos>)

  Exemplo:

  create index iconta on conta (num_conta)

- ✓ **Remover um índice:**

  drop index <nome_indice>

  Exemplo:

  drop index iconta

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```

Como processar ?

Quantos índices ?

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```

Como processar ?

1) Encontra todos os registros de Florianópolis usando índice por **cidade_agencia** e examina cada registro para atender saldo
2) Índice por **saldo** e examina cada registro para atender cidade_agencia
3) Usa 2 índices e faz intersecção

PROBLEMAS ????

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```

Como processar ?

1) Encontra todos os registros de Florianópolis usando índice por cidade_agencia e examina cada registro para atender saldo
2) Índice por saldo e examina cada registro para atender cidade_agencia
3) Usa 2 índices e faz intersecção

PROBLEMAS ???? Muitos registros para examinar em 1) e 2). Pouca intersecção em 3).

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```

Como processar ?

Outra estratégia: índice por (cidade_agencia, saldo)

PROBLEMAS?

Arquivos grid

✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```

Como processar ?

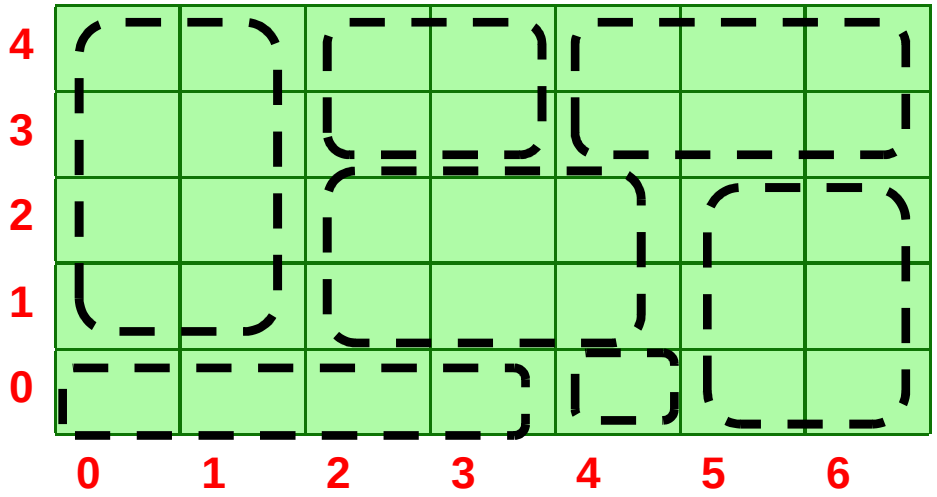
Outra estratégia: índice por (cidade_agencia, saldo)

PROBLEMAS? Registros ordenados sequencialmente por um dos atributos ➔ acesso a blocos diferentes no disco

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```



LINHAS

4 São Paulo

3 Niterói

2 Manaus

1 Curitiba

Escala linear para cidade_agencia

COLUNAS

1000	2000	5000	10000	500000	1000000
1	2	3	4	5	6

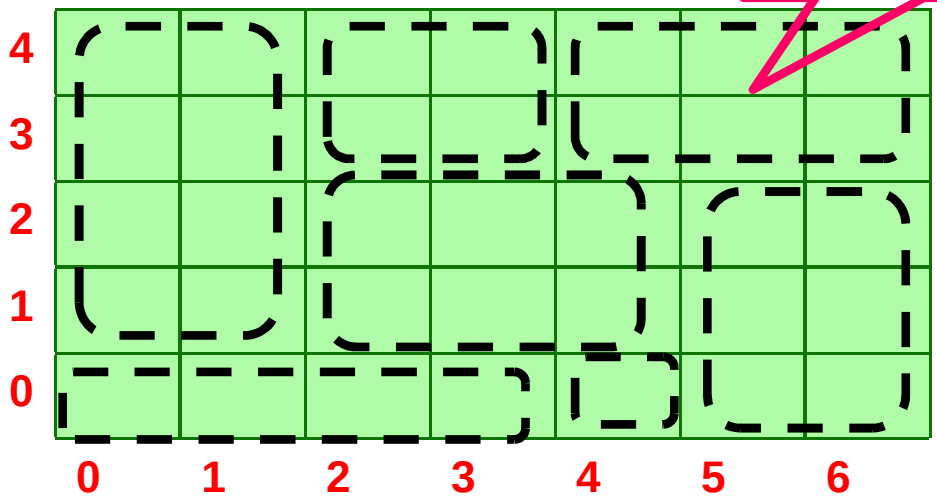
Escala linear para saldo

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```

cada célula tem ponteiro para um *bucket*



LINHAS

4 São Paulo

3 Niterói

2 Manaus

1 Curitiba

Escala linear para cidade_agencia

COLUNAS

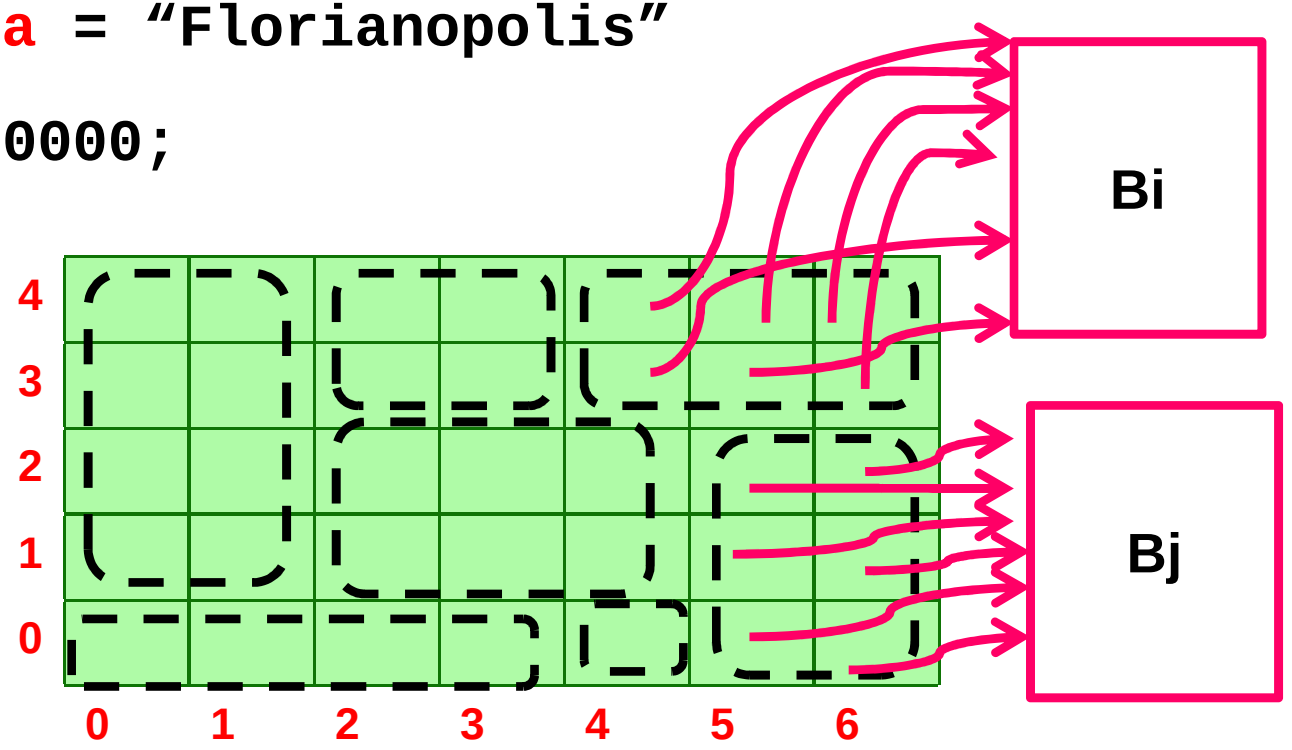
1000	2000	5000	10000	500000	1000000
1	2	3	4	5	6

Escala linear para saldo

Arquivos grid

- ✓ tipos de consultas com maior probabilidade de ocorrência

```
select numero_conta
from conta
where cidade_agencia = “Florianopolis”
      AND saldo = 60000;
```



LINHAS

4 São Paulo

3 Niterói

2 Manaus

1 Curitiba

Escala linear para cidade_agencia

COLUNAS

1000	2000	5000	10000	500000	1000000
1	2	3	4	5	6

Escala linear para saldo

Referências e Exercícios

- ✓ Silberschatz, A.; Korth, H.F.; Sudarshan, S. “Database system concepts” 4th edition. McGraw-Hill, 2001. (capítulos 11 e 12)
- ✓ Elmasri, R.; Navathe, S. “Fundamentals of Database Systems” third edition. Addison-Wesley Pubs. 2001.**(capítulos 13 e 14)**

ACH2025

Laboratório de Bases de Dados

Aula 9

Indexação e Hashing – Parte 2

Professora:

- **Fátima L. S. Nunes**