

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

# Aula 20

# **Índices**

Acelerando o acesso  
aos dados  
(Parte 1)

Profa. Kelly Rosa Braghetto

17/05/2016

# Representando Relações em Armazenamento Secundário

- ◆ Atributos são representados por **campos** → sequências de bytes (de tamanho fixo ou variável).
- ◆ Campos são agrupados em coleções (de tamanho fixo ou variável) chamadas de **registros**.
- ◆ Registros são armazenados em **blocos** físicos
  - ▶ Diferentes tipos de estruturas de dados podem ser usadas para os blocos → às vezes, os blocos precisam ser reorganizados quando o banco de dados é modificado.

# O que são índices?

- ◆ Uma coleção de registros que forma uma relação é armazenada em disco como uma coleção de blocos chamada de **arquivo de dados**.
- ◆ Um arquivo de dados pode ter um ou mais **arquivos de índice** associados a ele.
- ◆ Cada arquivo de índice associa valores da chave de busca a ponteiros para registros nos arquivos de dados que contêm esses valores para o(s) atributo(s) da chave de busca.

# Um Parênteses sobre Tipos de Chaves...

- ◆ **Chave primária** - atributo(s) que identifica(m) de forma unívoca as tuplas de uma relação
- ◆ **Chave de ordenação** - atributo(s) usado(s) para ordenar os registros no arquivo de dados
- ◆ **Chave de busca** - atributo(s) para usados para a busca de tuplas por meio de um índice

Às vezes, uma só chave faz o papel das três.

# Índices

- ◆ **Índice** → estrutura de dados usada para acelerar o acesso às tuplas de uma relação, dados valores para um ou mais atributos (= chave de busca)
- ◆ Índices evitam varreduras nas relações (“table scans”)
  - ▶ Tuplas são localizadas imediatamente
- ◆ Índices são mantidos pelos SGBDs e ficam armazenados nos seus respectivos bancos de dados

# Tipos de Índices

## ◆ Densos X Esparsos

- ▶ Índice denso: possui uma entrada no arquivo para cada registro no arquivo de dados
- ▶ Índice esparsos: possui entradas somente para alguns dos registros no arquivo de dados (geralmente, uma entrada no índice para cada bloco do arquivo de dados)
  - Só pode ser usado se o arquivo de dados estiver ordenado pela chave de busca

# Tipos de Índices

## ◆ Primários X Secundários

- ▶ **Índice primário:** é especificado sobre o(s) atributo(s) da chave de ordenação do arquivo de dados
  - ▶ **Índice secundário:** não determina a localização dos registros.
- ◆ Uma relação pode ter no máximo um índice primário (geralmente criado sobre a chave primária), mas pode ter vários índices secundários sobre outros atributos.

# Estruturas de Dados Usadas como Índices

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

# Arquivo Sequencial

- ◆ É um arquivo de dados com registros ordenados segundo uma chave de ordenação
- ◆ Geralmente, usa-se a chave primária da relação como chave de ordenação
  - ◆ Mas pode ser usado com outros atributos também

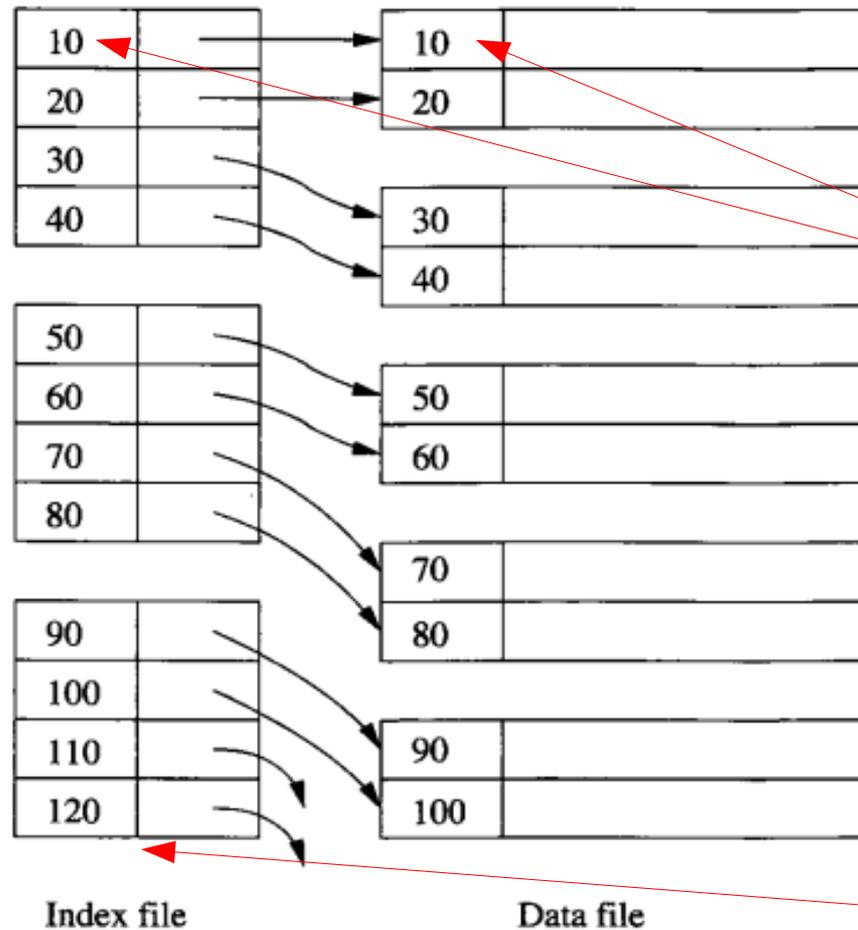
# Arquivo Sequencial

- ◆ Vantagens:
  - ▶ Leitura dos registros pela ordem da chave de ordenação é feita de forma eficiente
  - ▶ Encontrar o próximo registro a partir do registro atual (quase sempre) não requer a leitura de um novo bloco do disco
  - ▶ Buscar um registro pelo valor da chave de ordenação pode ser feito de forma eficiente (com busca binária, por exemplo)

# Arquivo Sequencial

- ◆ Desvantagens:
  - ▶ A ordenação não facilita a busca de registros por outros atributos diferentes da chave de ordenação
  - ▶ Inserção e exclusão de registros são operações dispendiosas → registros devem permanecer fisicamente ordenados
    - Para diminuir o impacto desse problema, costuma-se manter algum espaço livre em cada bloco ou um bloco especial (de *overflow*) para acomodar a inserção de novos registros

# Índice Primário Denso



Cada bloco do arquivo de dados acomoda só 2 registros

Valor da chave de ordenação do registro

Cada bloco do arquivo de índice acomoda 4 pares chave-ponteiro

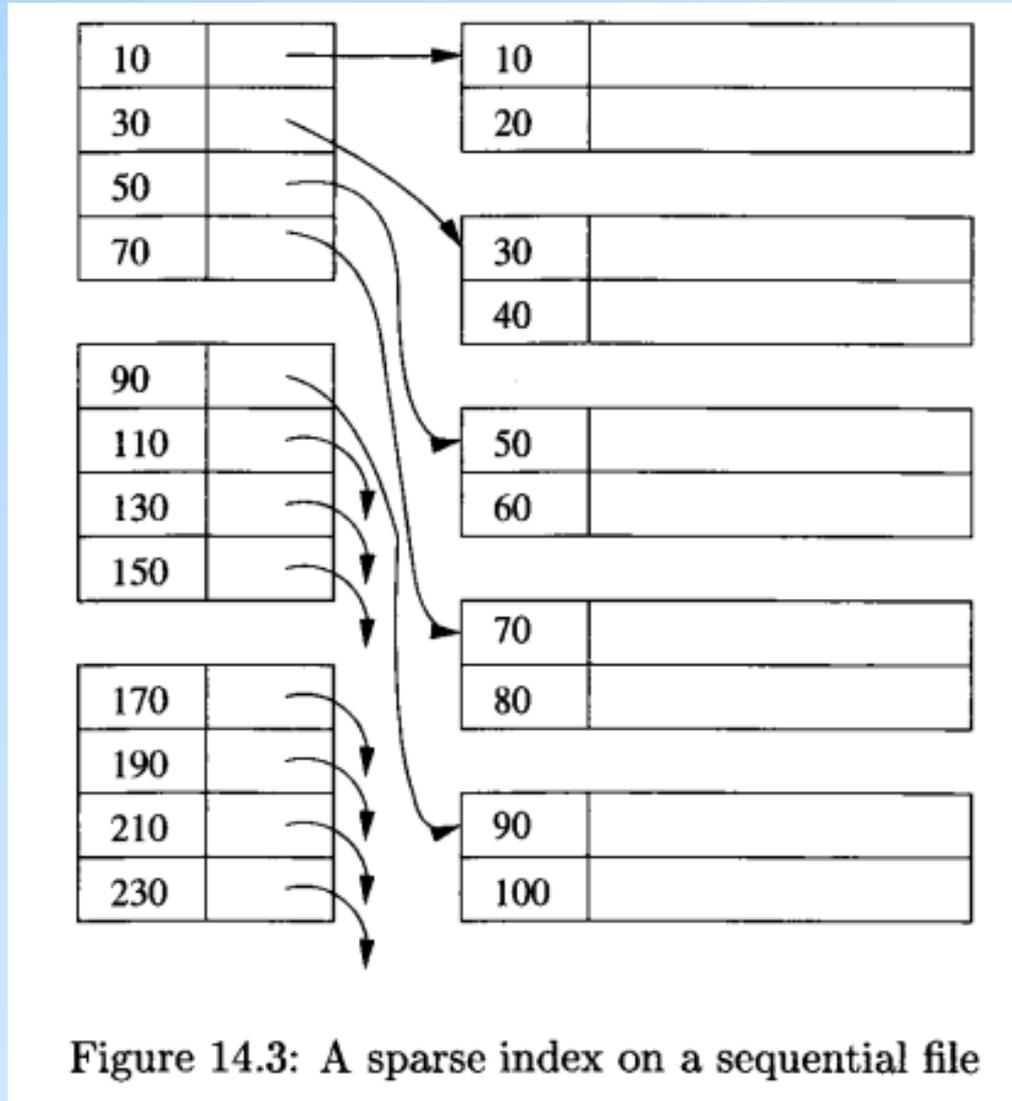
Figure 14.2: A dense index (left) on a sequential data file (right)

Neste exemplo, chave de ordenação = chave primária da relação<sup>12</sup>

# Índice Primário Denso

- ◆ **Índice denso** é uma sequência de blocos que armazenam apenas as chaves dos registros e ponteiros (endereços) para eles
- ◆ É denso porque toda chave no arquivo sequencial é representada no índice
- ◆ Se o par (chave,ponteiro) ocupa menos espaço que o registro inteiro, o índice ocupa menos blocos do que o arquivo de dados em si
  - ◆ Particularmente vantajoso quando o índice cabe na memória principal (e o arquivo de dados não!) → pelo índice, podemos encontrar qualquer registro dado sua chave de busca com apenas um acesso ao disco.

# Índice Primário Esparso



# Índice Primário Esparso

- ◆ Geralmente mantém apenas uma chave para cada bloco do arquivo de dados
  - ▶ A chave mantida é a do primeiro registro do bloco
- ◆ Ocupa menos espaço que um índice denso, ao custo de se demorar um pouco mais para encontrar um registro dada a sua chave
- ◆ Mais fácil “de caber” na memória principal
- ◆ Assim como um índice esparso, possibilita o uso de busca binária para localizar uma chave
- ◆ Diferentemente de um índice denso, não nos permite responder a consultas do tipo “existe algum registro com o valor  $k$  para a chave?” olhando apenas para as entradas no índice (ou seja, sem envolver o acesso aos blocos do arquivo de dados)

# Índice Primário Esparsos Multinível

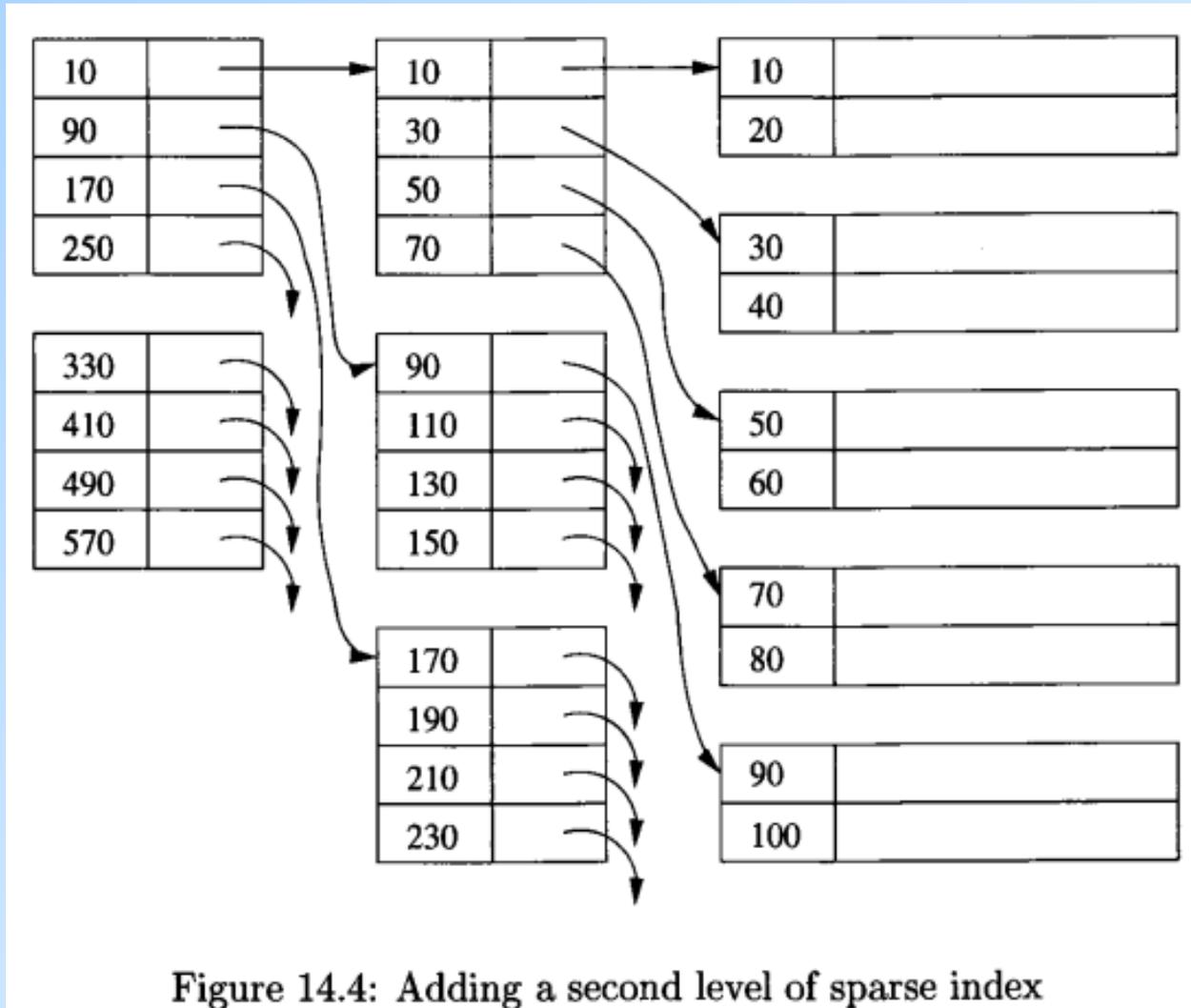
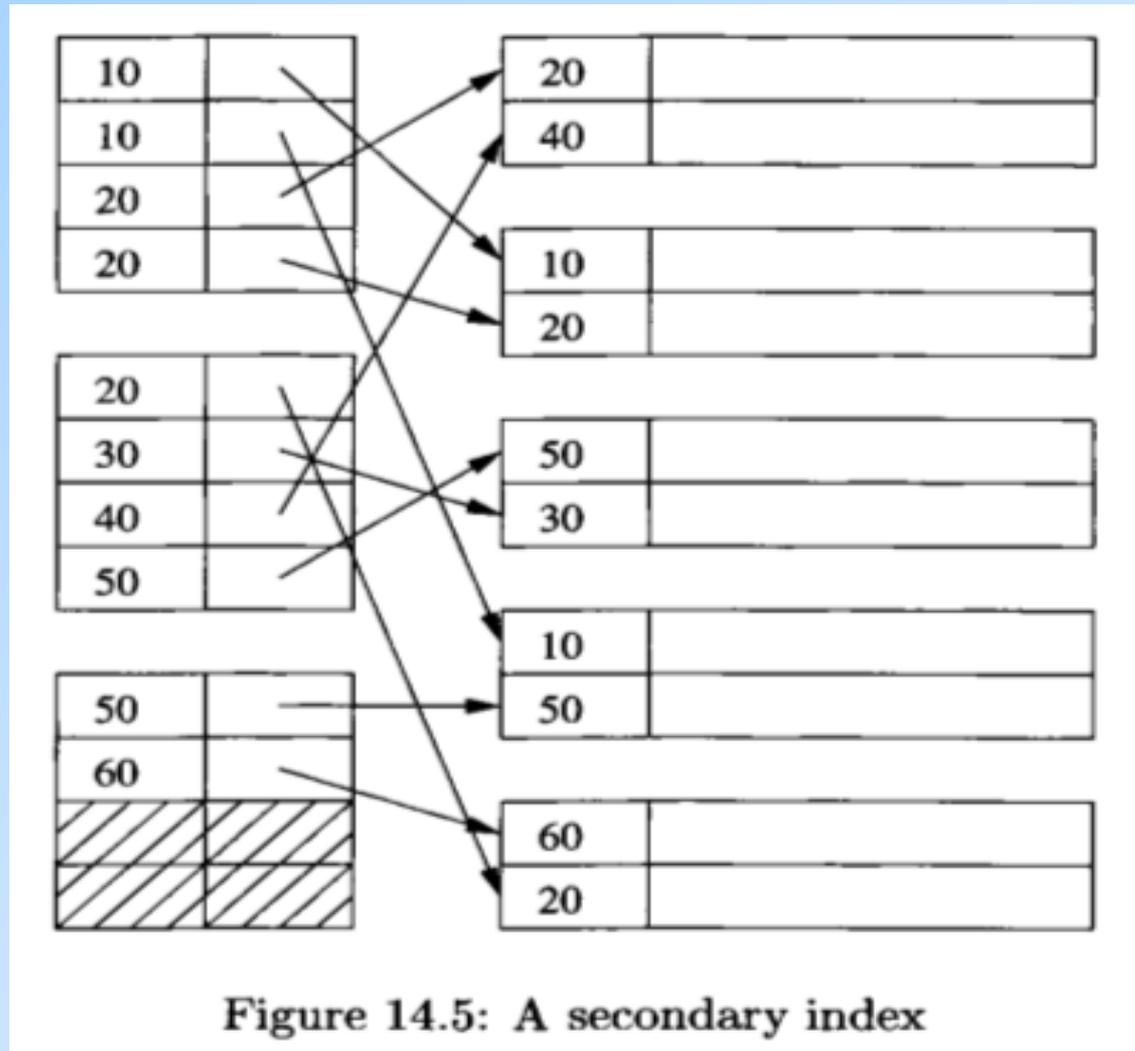


Figure 14.4: Adding a second level of sparse index

# Índice Primário Esparsos Multinível

- ◆ Um índice pode cobrir muitos blocos
- ◆ A busca binária por uma chave no índice pode demandar a leitura (do disco) de vários blocos do índice
- ◆ Colocando um índice sobre o índice, reduzimos o número de blocos de índice que precisam ser lidos do disco para se encontrar um dado registro por meio de sua chave

# Índice Secundário



**Índices secundários são sempre densos! Por quê?**

# Índice Secundário

- ◆ Com frequência, precisamos realizar buscas sobre atributos que não são a chave primária (ou a chave de ordenação) da relação
  - ▶ Índices secundários facilitam esse tipo de buscas
- ◆ Um índice secundário não “decide” qual deve ser a localização dos registros no arquivo de dados
  - ▶ Essa localização é definida pelo índice primário
  - ▶ Um índice secundário apenas nos informa sobre essa localização

# Índice Secundário

- ◆ Como um índice secundário não influencia a localização, ele não pode ser usado para prever a localização de um registro cuja chave não aparece no arquivo do índice explicitamente
  - ▶ **Por isso, índices secundários são sempre densos**
- ◆ Para facilitar as buscas, as entradas num índice secundário aparecem ordenadas pela chave de busca
  - ▶ Mas os registros no arquivo de dados não!
- ◆ A chave de busca usada em um índice secundário não precisa ser única
  - ▶ Podem aparecer chaves repetidas no arquivo de índice

# Índice Secundário Multinível

- ◆ Assim como ocorre num índice primário, é possível adicionar um segundo nível (esparso) num índice secundário
  - ▶ Esse nível pode conter pares (chave, ponteiro) correspondentes à primeira chave de cada bloco no índice denso ou à primeira ocorrência de cada chave no índice denso
  - ▶ Objetivo: reduzir o número de blocos de índice que precisam ser lidos do disco na busca por uma chave

# Desperdício de Espaço em Índices Secundários

Espaço gasto com chaves repetidas

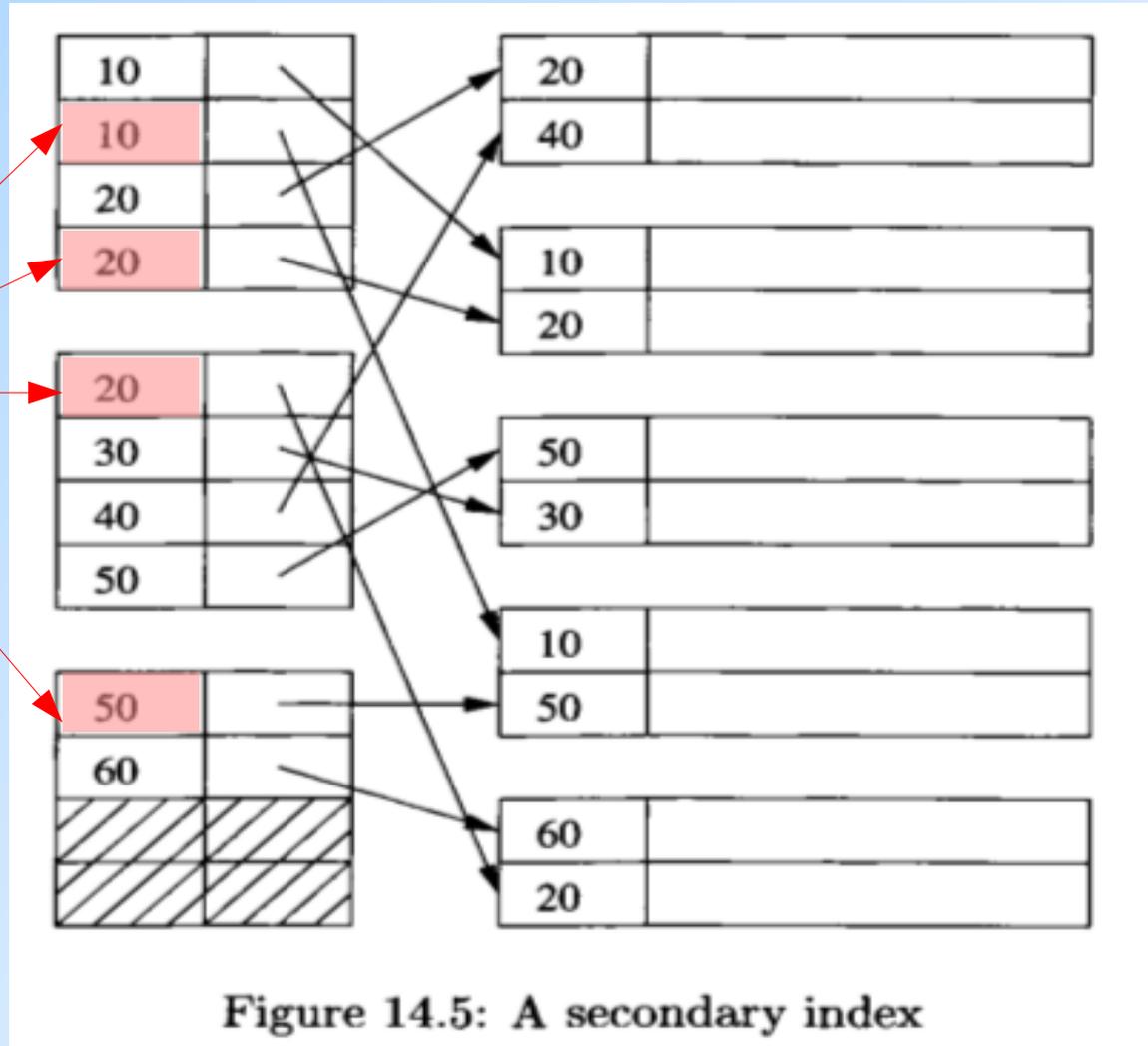


Figure 14.5: A secondary index

# Índice Secundário com Indireção

- ◆ Pode haver um desperdício significativo de espaço na estrutura de índice do slide anterior
- ◆ Se uma dada chave de busca aparece  $n$  vezes no arquivo de dados, então ela é escrita  $n$  vezes no arquivo de índice
  - ◆ Seria melhor escrevê-la uma única vez no índice
- ◆ Essa repetição de valores pode ser evitada usando um nível de indireção (*buckets*) entre o arquivo de índice secundário e o arquivo de dados (ver exemplo no próximo slide)

# Índice Secundário com Indireção

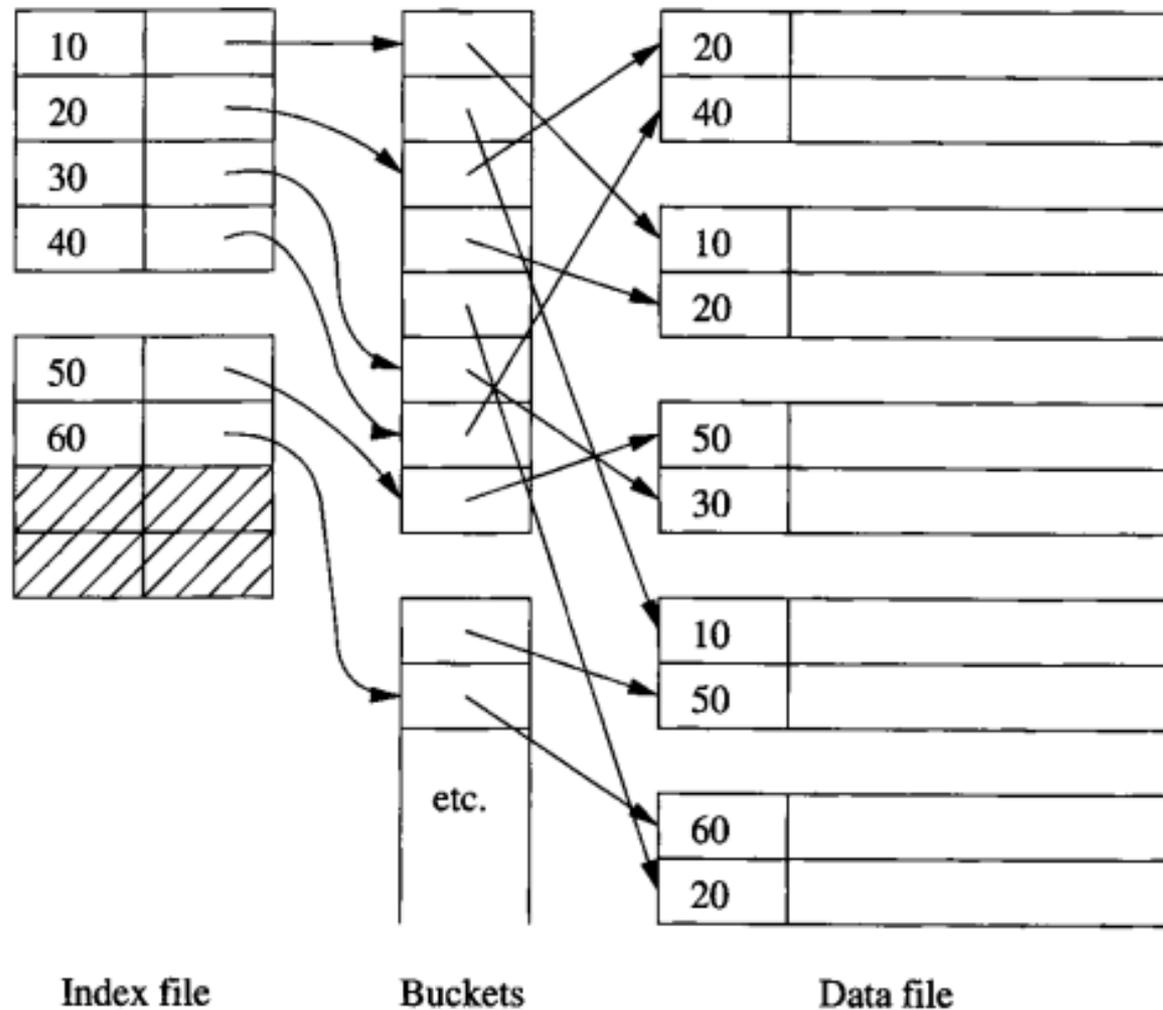


Figure 14.7: Saving space by using indirection in a secondary index

# Índice Secundário com Indireção

- ◆ O uso da indireção tem duas vantagens:
  - ▶ Economia espaço se os valores das chaves de busca forem maiores que os ponteiros e se cada chave aparecer (em média) pelo menos 2x
  - ▶ Os ponteiros nos *buckets* ajudam a responder consultas sem ter que “olhar” para a maior parte dos registros no arquivo de dados (\*)

(\*) Quando uma consulta tem várias condições e existe um índice secundário que a ajude, pode-se encontrar os ponteiros nos *buckets* que satisfazem todas as condições calculando (em memória) a intersecção dos conjuntos de ponteiros e recuperando apenas os registros apontados pelos ponteiros na intersecção. Isso pode ser feito mesmo em índices sem indireção; mas o uso dos *buckets* nesse caso economiza acessos ao disco, já que os ponteiros ocupam menos espaço que pares (chave, ponteiro).

# Recuperação de Documentos

- ◆ A comunidade de recuperação de informação lida com o problema de armazenar documentos e recuperá-los de forma eficiente dado um conjunto de palavras-chaves
- ◆ Com o advento da WWW e a possibilidade de se manter todos os documentos online, a recuperação de documentos por palavras-chaves passou a ser um dos problemas mais relevantes na área de BD
- ◆ Existem diferentes tipos de consultas que podem ser usadas nesse contexto

# Recuperação de Informação

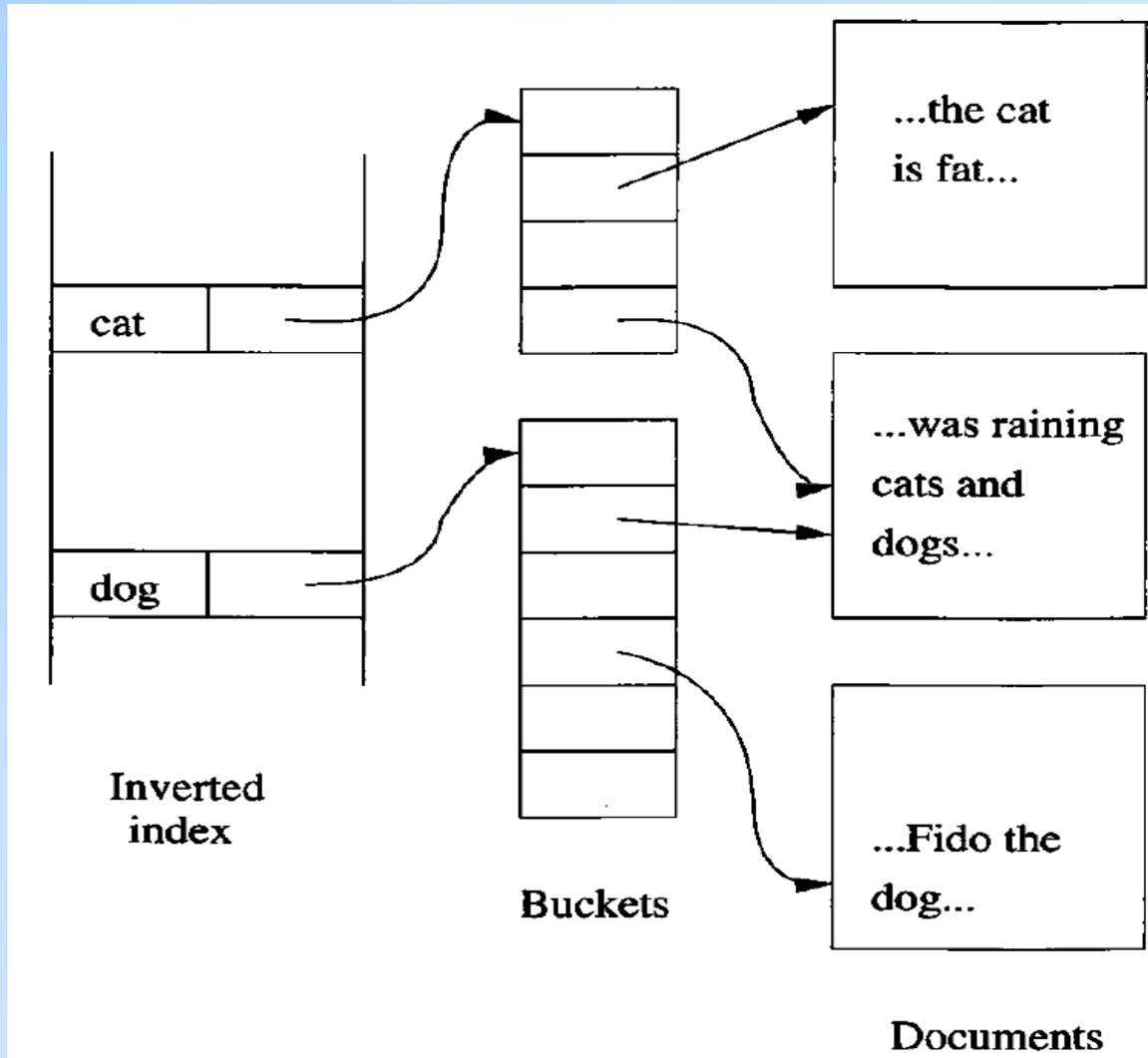
Em termos relacionais, as consultas mais simples e comuns podem ser vistas da seguinte forma:

- ◆ documento = tupla numa relação **Doc**
- ◆ **Doc** tem um atributo *booleano* para cada possível palavra que um documento nela possa conter
  - ▶ Ex: **Doc(hasCat, hasDog, ...)**, onde **hasCat** é verdadeiro sse o documento tem a palavra “cat” pelo menos 1 vez
- ◆ Existe um índice secundário para cada atributo de Doc
  - ▶ O índice pode manter apenas as entradas relacionadas aos documentos que contêm a palavra, ou seja, quando o valor da chave de busca é verdadeiro
- ◆ Ao invés de criar um índice separado para cada atributo (= palavra), os índices são combinados em um só - o **índice invertido**

## Índices Invertidos

- ▶ Esse índice usa *buckets* para indireção, para economia de espaço

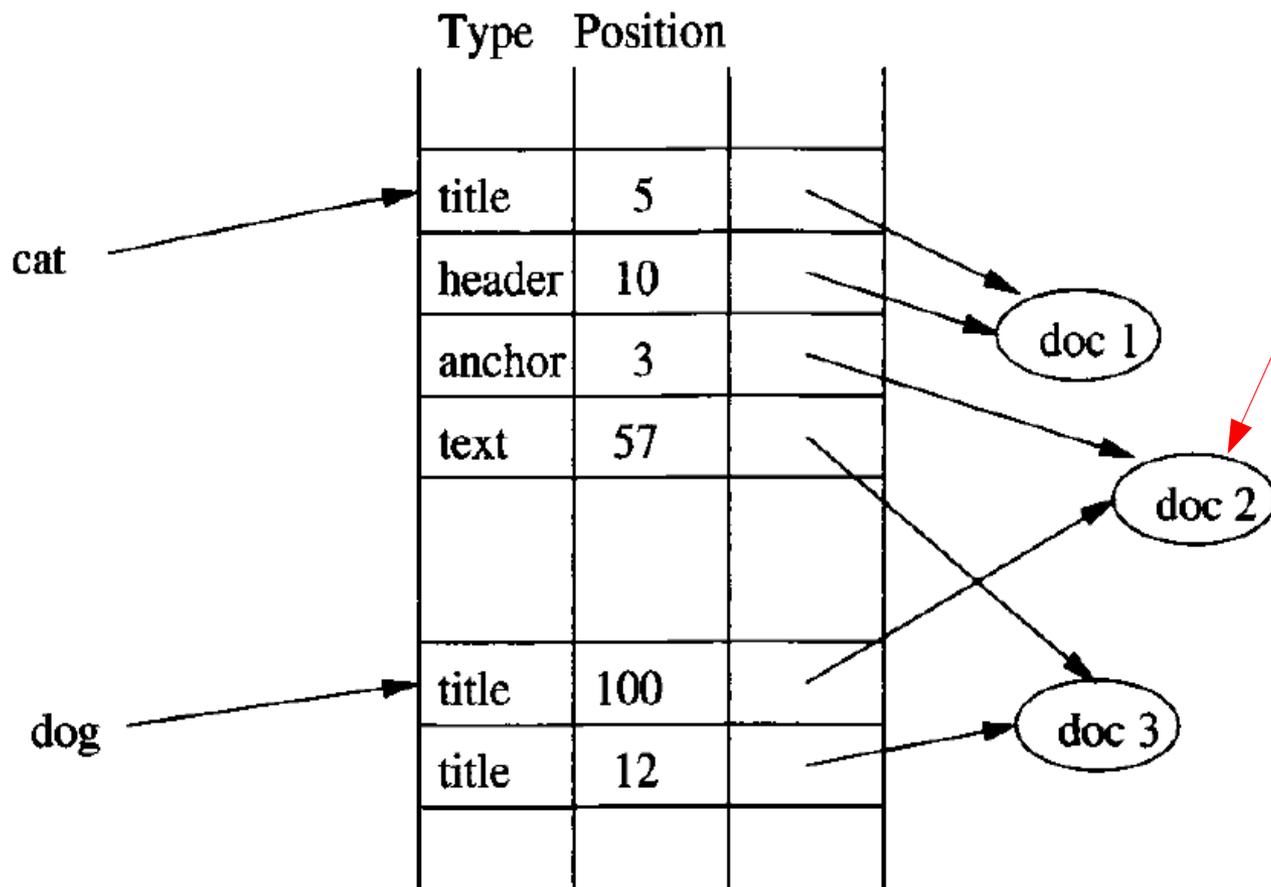
# Índice Invertido



# Índice Invertido

- ◆ O índice invertido é composto por um conjunto de pares (palavra, ponteiro)
  - ▶ As palavras são a chave de busca para o índice
  - ▶ Os ponteiros se referem a posições no arquivo de *buckets*.
- ◆ Ponteiros no arquivo de *buckets* podem ser:
  - ▶ Ponteiros para os documentos propriamente ditos
  - ▶ Ponteiros para uma ocorrência da palavra
    - Nesse caso, o ponteiro pode ser um par com o endereço do início do documento e um inteiro indicando o número da palavra ou a posição dela no documento.
    - Também pode-se distinguir ocorrências que aparecem no título, no resumo ou no corpo do texto. Em documentos em HTML, XML e etc., é possível associar marcações a trechos e palavras do texto.

# Índice Invertido com Mais Informações Armazenadas



Possível resposta a uma busca por docs sobre cachorros e que os comparam com gatos. (Docs que mencionam cachorro no título e mencionam gatos nas âncoras (links) são bons candidatos para a resposta.)

# Estruturas de Dados Usadas para Implementar Índices

- ◆ Tabela *hash*
  - ▶ Tempo de acesso: constante
  - ▶ Úteis para condições envolvendo comparações de igualdade
- ◆ Árvore balanceada de busca, com nós “gigantes” (uma página inteira do disco)
  - ▶ *Árvore B+ (B+ tree)*
  - ▶ Tempo de acesso: logarítmico
  - ▶ Úteis para condições envolvendo comparações feitas com os operadores =, >, >=, <, <=

# Árvores B+

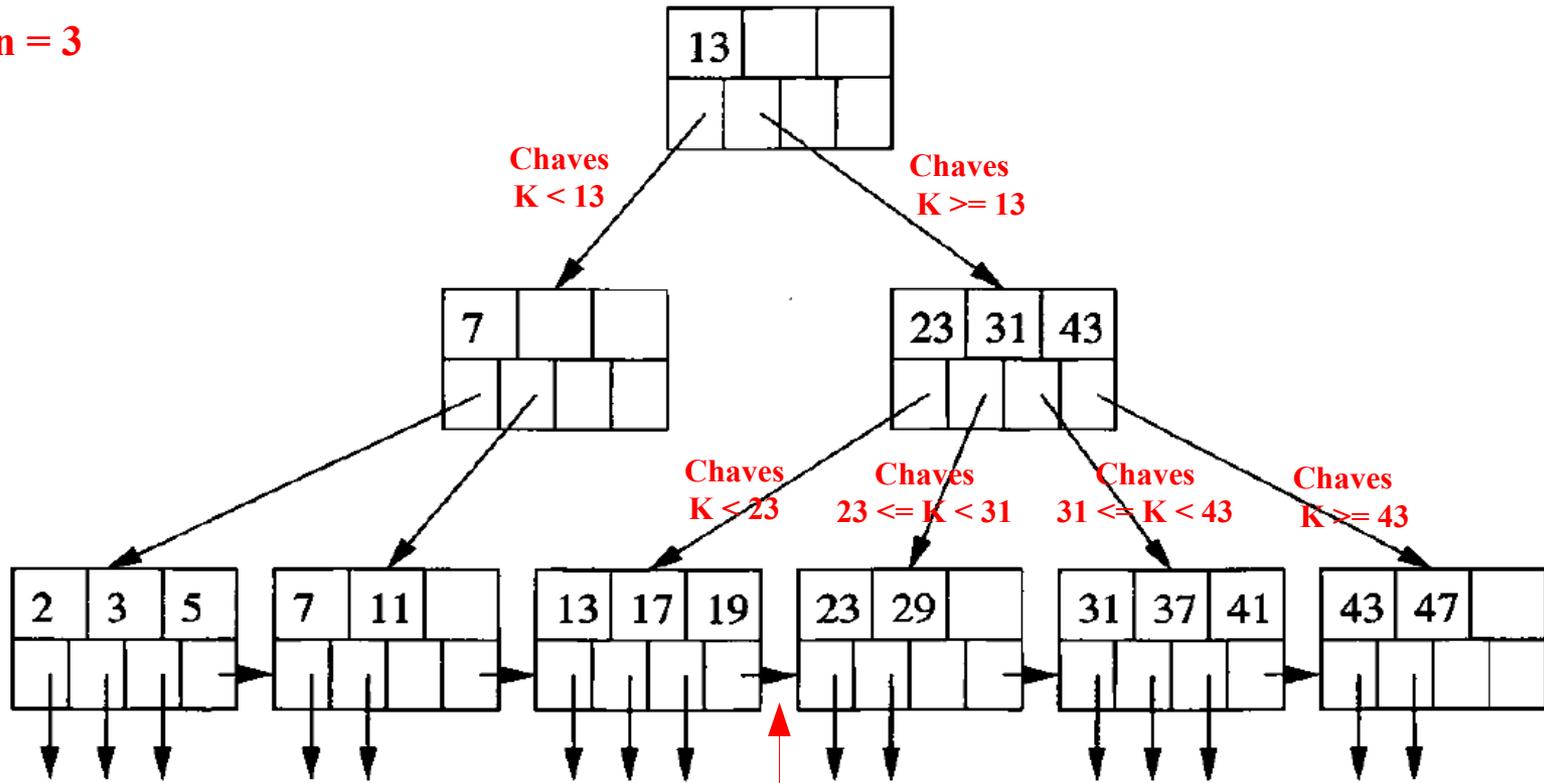
- ◆ Vimos que um ou dois níveis adicionais no índice ajudam a melhorar o desempenho de consultas
- ◆ Para se beneficiar disso de forma mais “plena”, os SGBDs exploram essa ideia por meio de uma estrutura mais geral – as árvores B+
  - ▶ As árvores B+ automaticamente mantêm tantos níveis de índice quanto for apropriado para o tamanho do arquivo de dados sendo indexado
  - ▶ As árvores B+ gerenciam o espaço usado nos blocos de modo a não precisar de blocos de *overflow*. Todo bloco fica com uma ocupação entre a metade de sua capacidade ou completamente cheio.

# Estrutura das Árvores B+

- ◆ Uma árvore B organiza seus blocos em um árvore balanceada
  - ▶ Todos os caminhos do nó raiz até um nó folha tem o mesmo comprimento
- ◆ Há um parâmetro  $n$  associado a toda árvore B+, que determina o leiaute de todos os blocos da árvore. Cada bloco tem:
  - ▶  $n$  valores de chaves de busca
  - ▶  $n+1$  ponteiros
- ◆ Escolhe-se o maior valor de  $n$  possível tal que  $n+1$  ponteiros e  $n$  chaves caibam em um só bloco

# Árvore B+

$n = 3$



Ponteiro para o registro com chave 2

Ponteiro para o próximo nó folha na sequência

# Estrutura das Árvore B+

- ◆ As chaves nos nós folhas são cópias das chaves do arquivo de dados
  - ▶ As chaves são distribuídas nas folhas ordenadamente, da esquerda para a direita
- ◆ No nó raiz, há sempre pelo menos 2 ponteiros em uso
  - ▶ Exceção: índice para um arquivo de dados que possui apenas 1 registro
- ◆ Numa folha, pelo menos  $\lfloor (n+1)/2 \rfloor$  ponteiros devem estar “ocupados” apontando para registros
  - ▶ No máximo  $n$  ponteiros são usados para registros, já que 1 ponteiro é usado para apontar para o nó folha seguinte
- ◆ Num nó interno, todos os  $n+1$  ponteiros podem ser usados para apontar para blocos da árvore que estão num nível inferior. Pelo menos  $\lfloor (n+1)/2 \rfloor$  ponteiros devem estar “ocupados” (se o nó não for a raiz)

# Aplicações das Árvores B+

As árvores B+ podem ser usadas para implementar qualquer um dos índices que vimos nesta aula.

Exemplos:

- ◆ **Índice denso** → chave de busca da árvore B+ = chave primária do arquivo de dados + 1 par chave-ponteiro nas folhas para cada registro no arquivo de dados
  - ◆ Arquivo de dados pode ou não estar ordenado pela chave primária
- ◆ **Índice primário esparsos** → arquivo de dados ordenado por sua chave primária + árvore B+ como índice esparsos, com 1 par chave-ponteiro na folhas para cada bloco do arquivo de dados

# Aplicações das Árvores B+

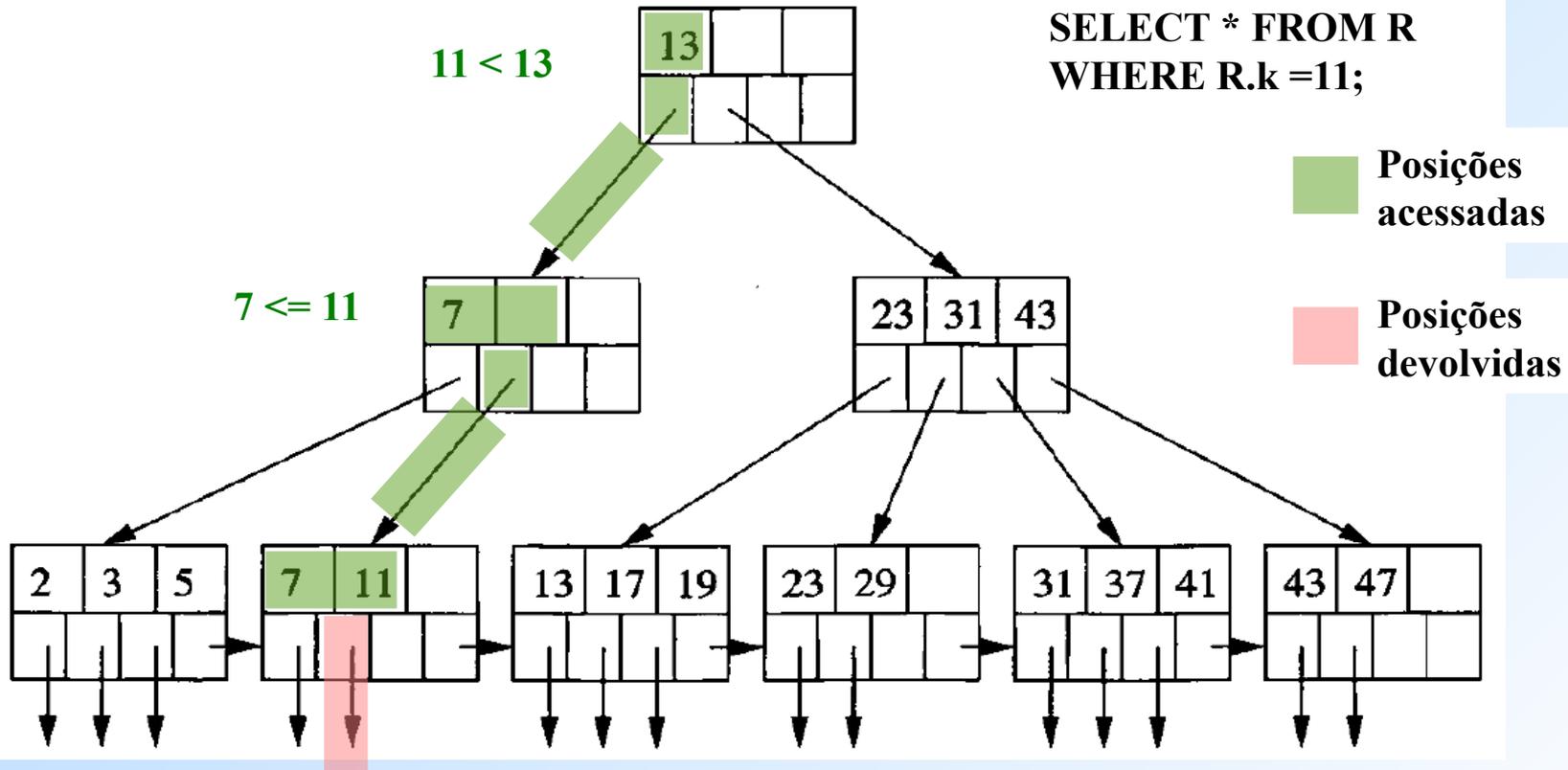
As árvores B+ podem ser usadas para implementar qualquer um dos índices que vimos nesta aula. Exemplos:

- ◆ **Índice para chaves duplicadas** → arquivo de dados ordenado por um atributo que não é chave primária da relação e esse atributo é a chave de busca para a árvore B+. Para cada chave  $k$  que aparece no arquivo dos dados, há 1 par chave-ponteiro numa folha. Esse ponteiro aponta para o **primeiro** registro que possui  $k$  como valor para a chave de ordenação
  - ▶ Note que, embora possa haver chaves de ordenação repetidas no arquivo de dados, não há chaves de busca repetidas nos nós folhas da árvore

# Buscando uma Chave em uma Árvore B+

Suposição: o índice é denso

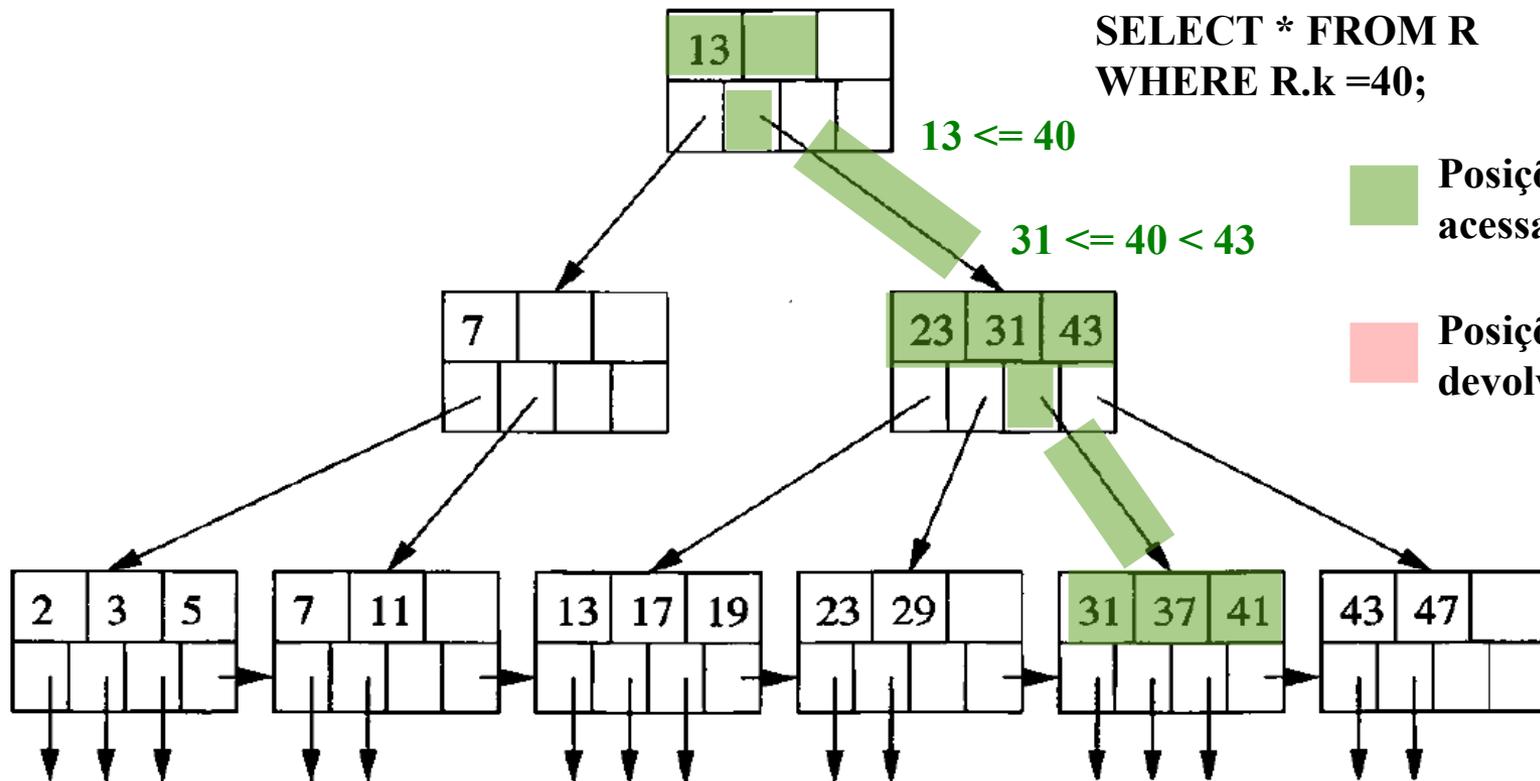
SELECT \* FROM R  
WHERE R.k = 11;



# Buscando uma Chave em uma Árvore B+

Suposição: o índice é denso

SELECT \* FROM R  
WHERE R.k = 40;

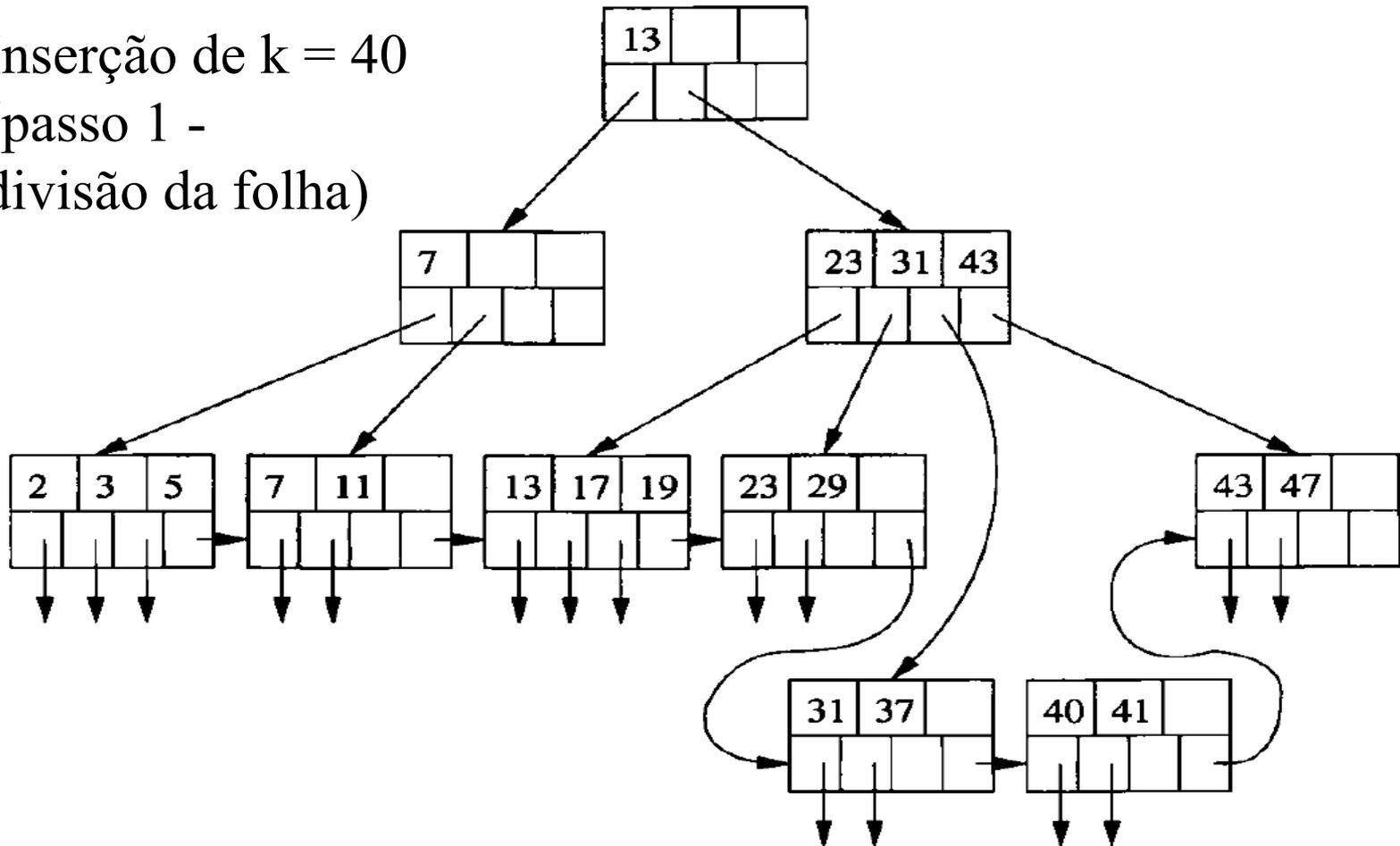


40 não está na folha,  
então ele não existe no  
arquivo de dados

# Inserção em Árvores B+

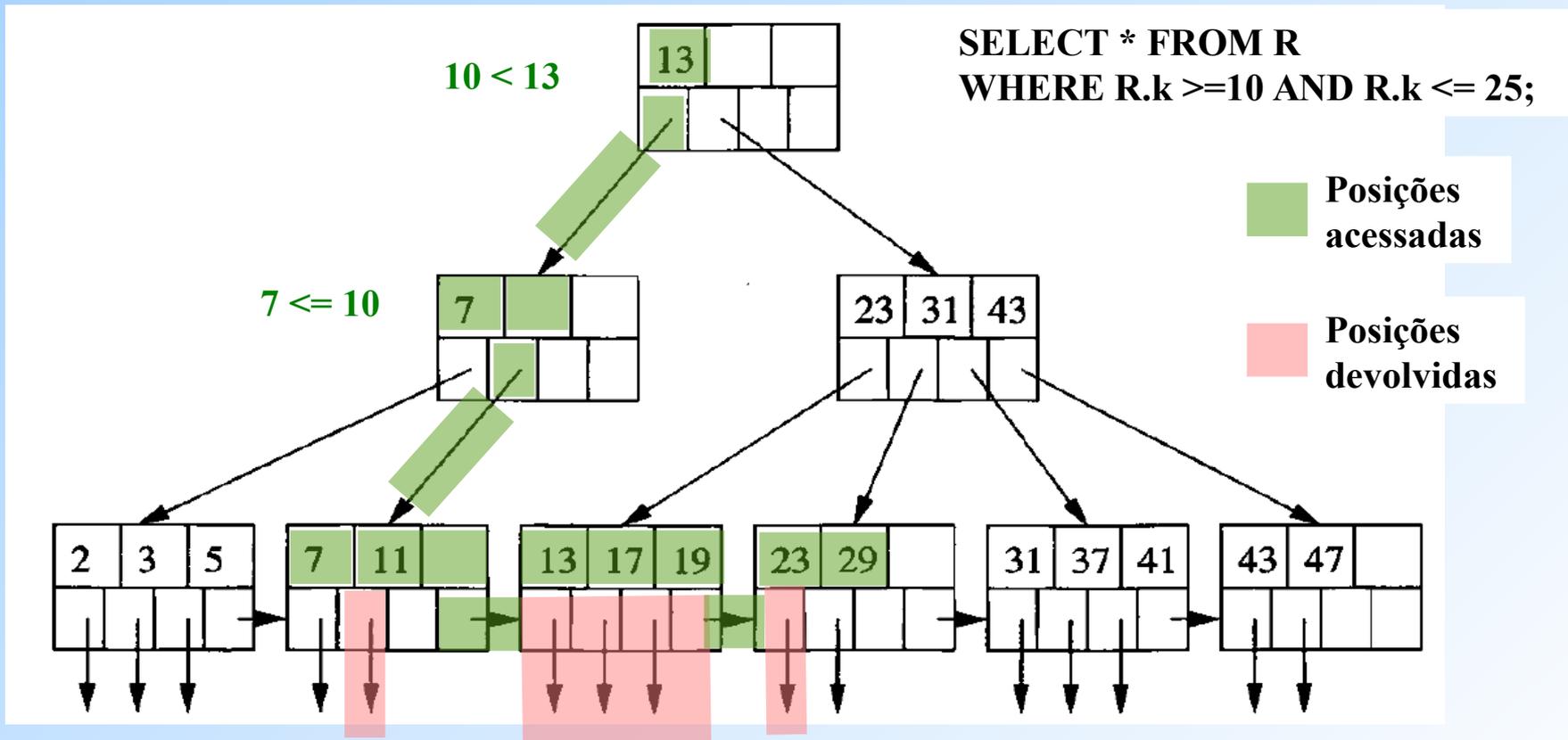
## Exemplo 2

Inserção de  $k = 40$   
(passo 1 -  
divisão da folha)



# Buscando uma Faixa de Valores em uma Árvore B+

Suposição: o índice é denso

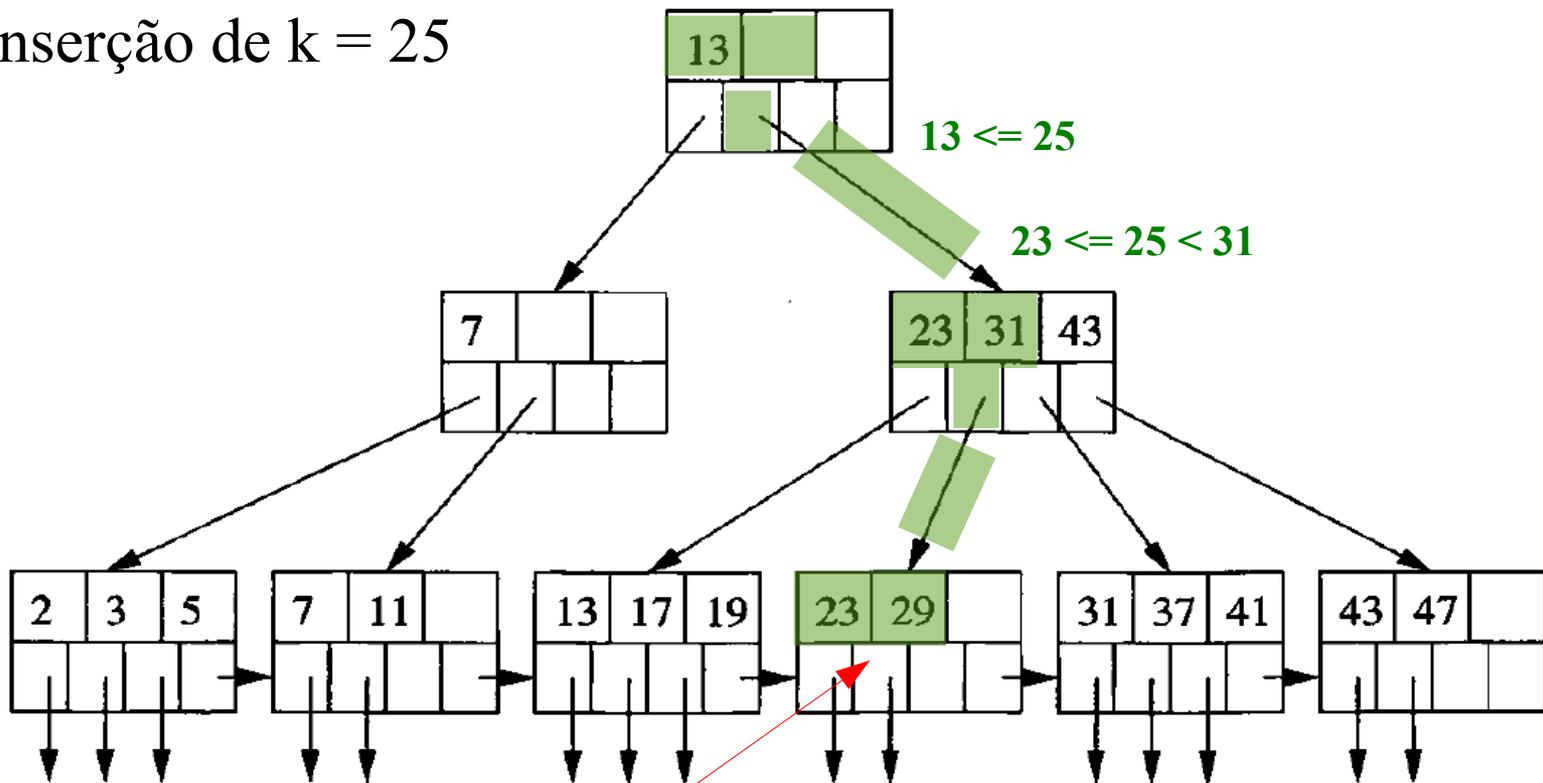


Como 10 não está na folha, então encontramos o primeiro valor maior que ele (11). Depois, segue-se navegando sequencialmente pelas folhas até encontrar o primeiro valor > 25 (no caso, o 29) ou o fim da seq.

# Inserção em Árvores B+

## Exemplo 1

Inserção de  $k = 25$

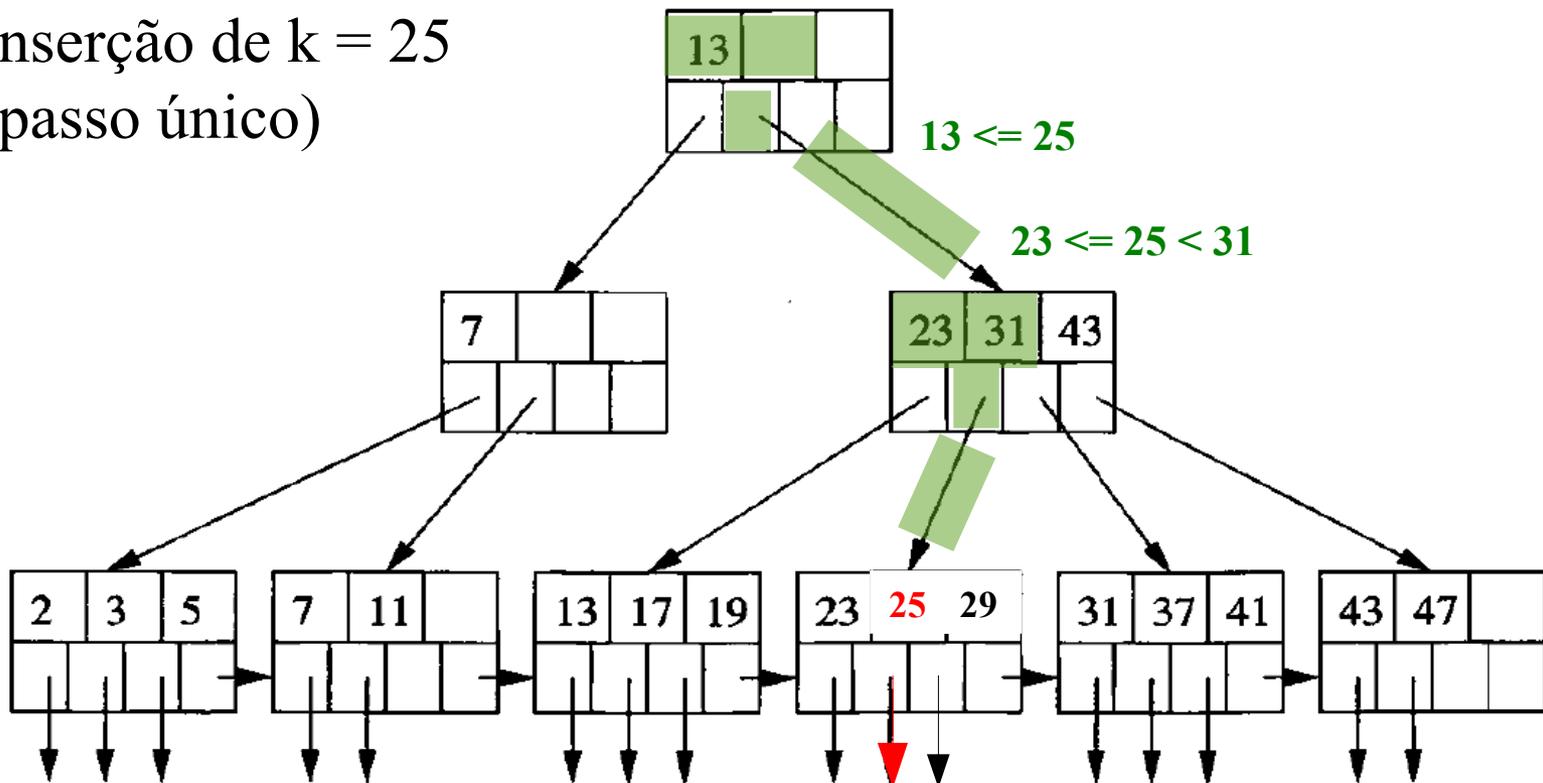


O valor 25 deve ficar entre os valores 23 e 29. Como ainda há espaço livre no nó, basta inserir a nova chave nele, mas mantendo a ordenação.

# Inserção em Árvores B+

## Exemplo 1

Inserção de  $k = 25$   
(passo único)

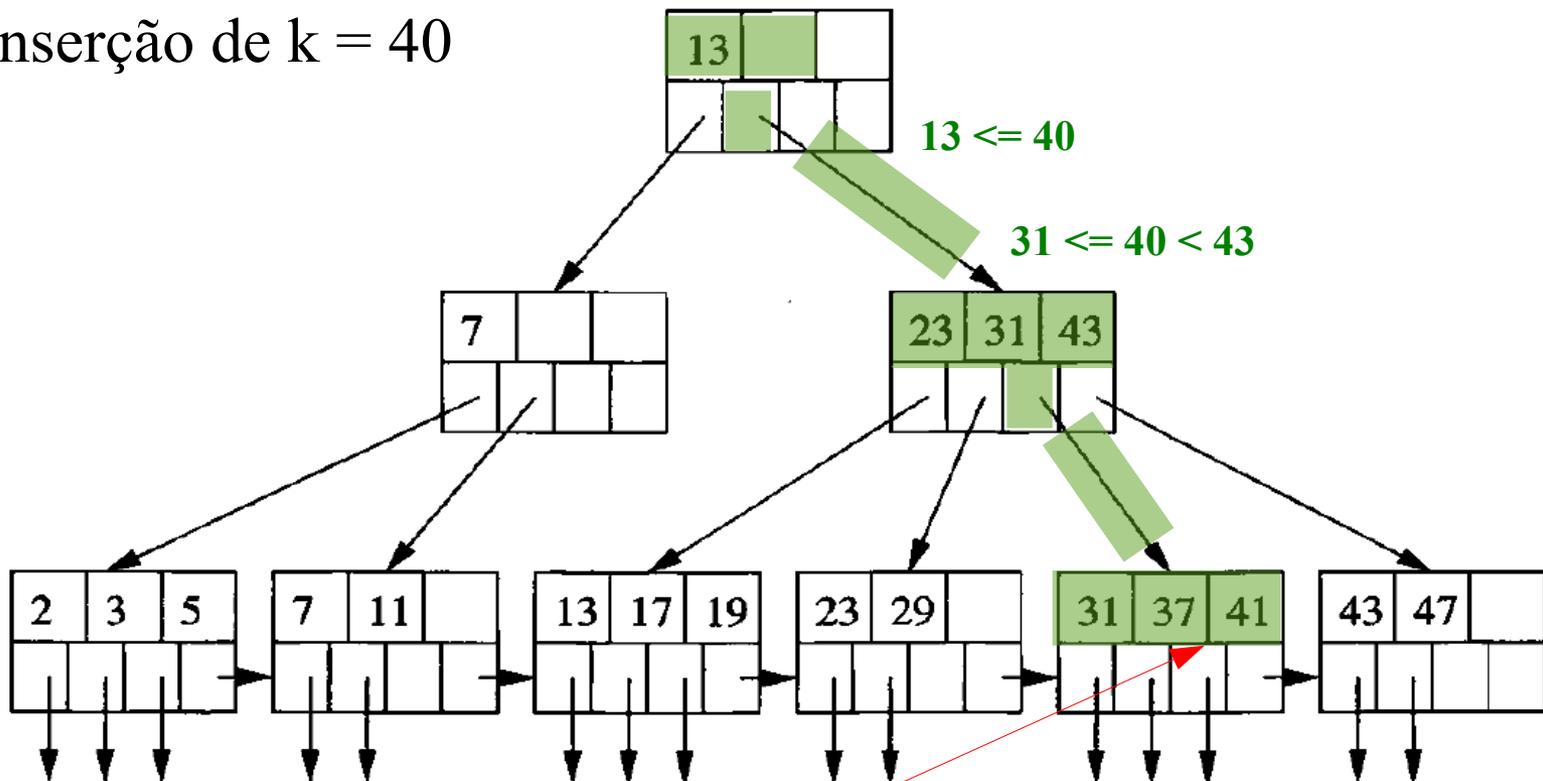


O valor 25 deve ficar entre os valores 23 e 29. Como ainda há espaço livre no nó, basta inserir a nova chave nele, mas mantendo a ordenação.

# Inserção em Árvores B+

## Exemplo 2

Inserção de  $k = 40$

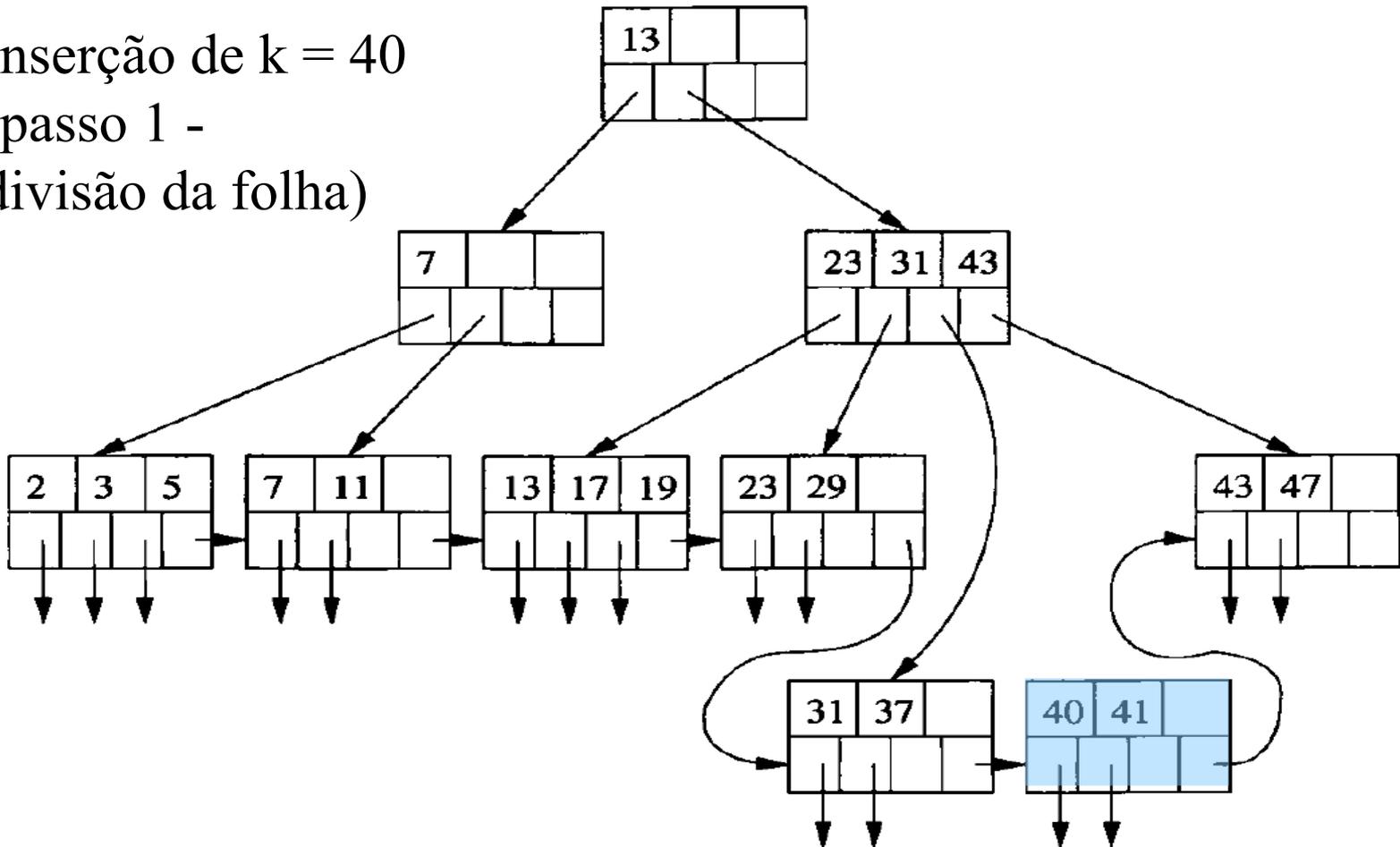


O valor 40 deve ficar entre os valores 37 e 41. Como não há mais espaço livre no nó, é preciso “quebrar” o nó em dois e dividir as chaves (inclusive a nova) entre os nós, de modo que cada um fique metade cheio ou com uma chave a mais que isso.

# Inserção em Árvores B+

## Exemplo 2

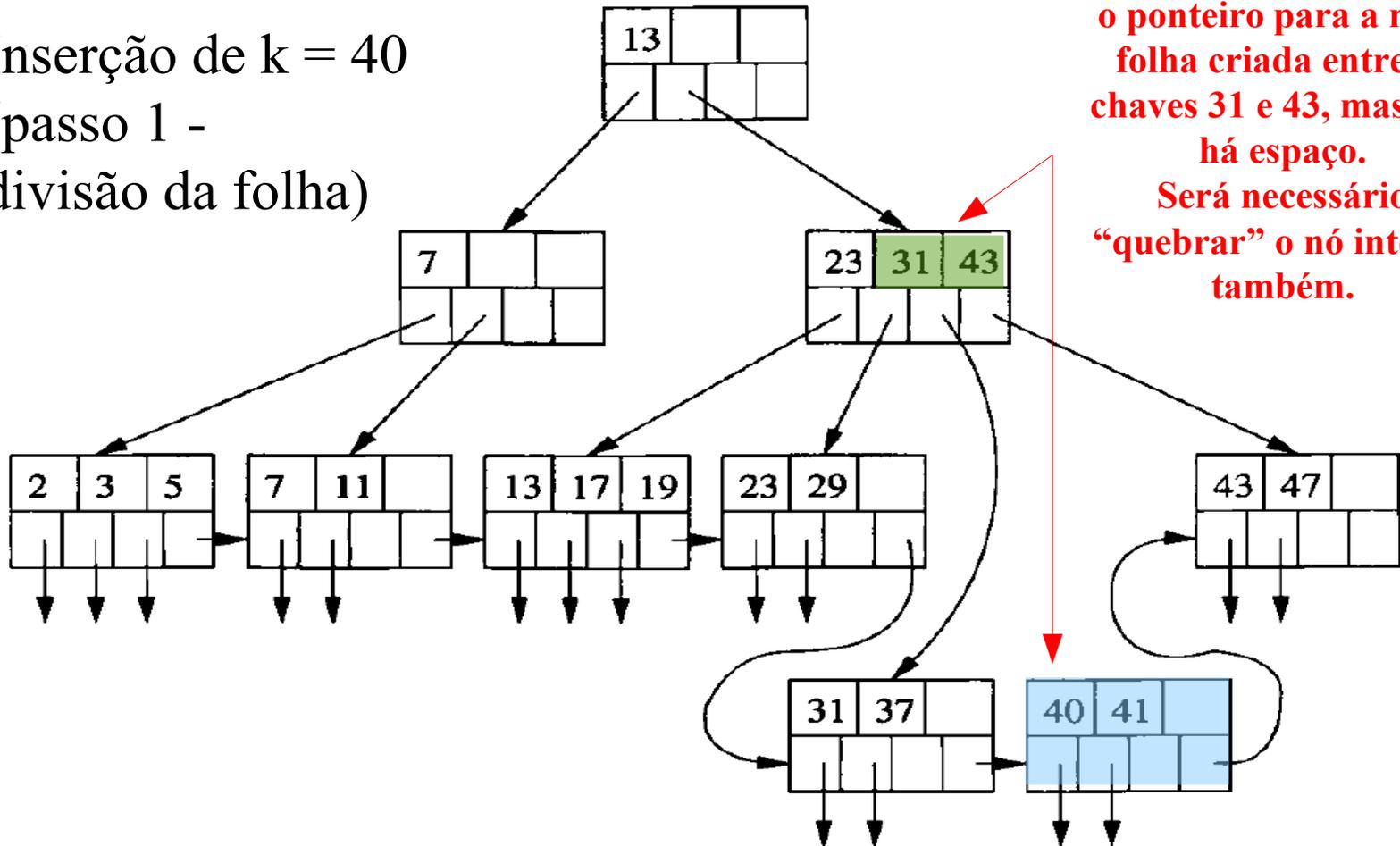
Inserção de  $k = 40$   
(passo 1 -  
divisão da folha)



# Inserção em Árvores B+

## Exemplo 2

Inserção de  $k = 40$   
(passo 1 -  
divisão da folha)

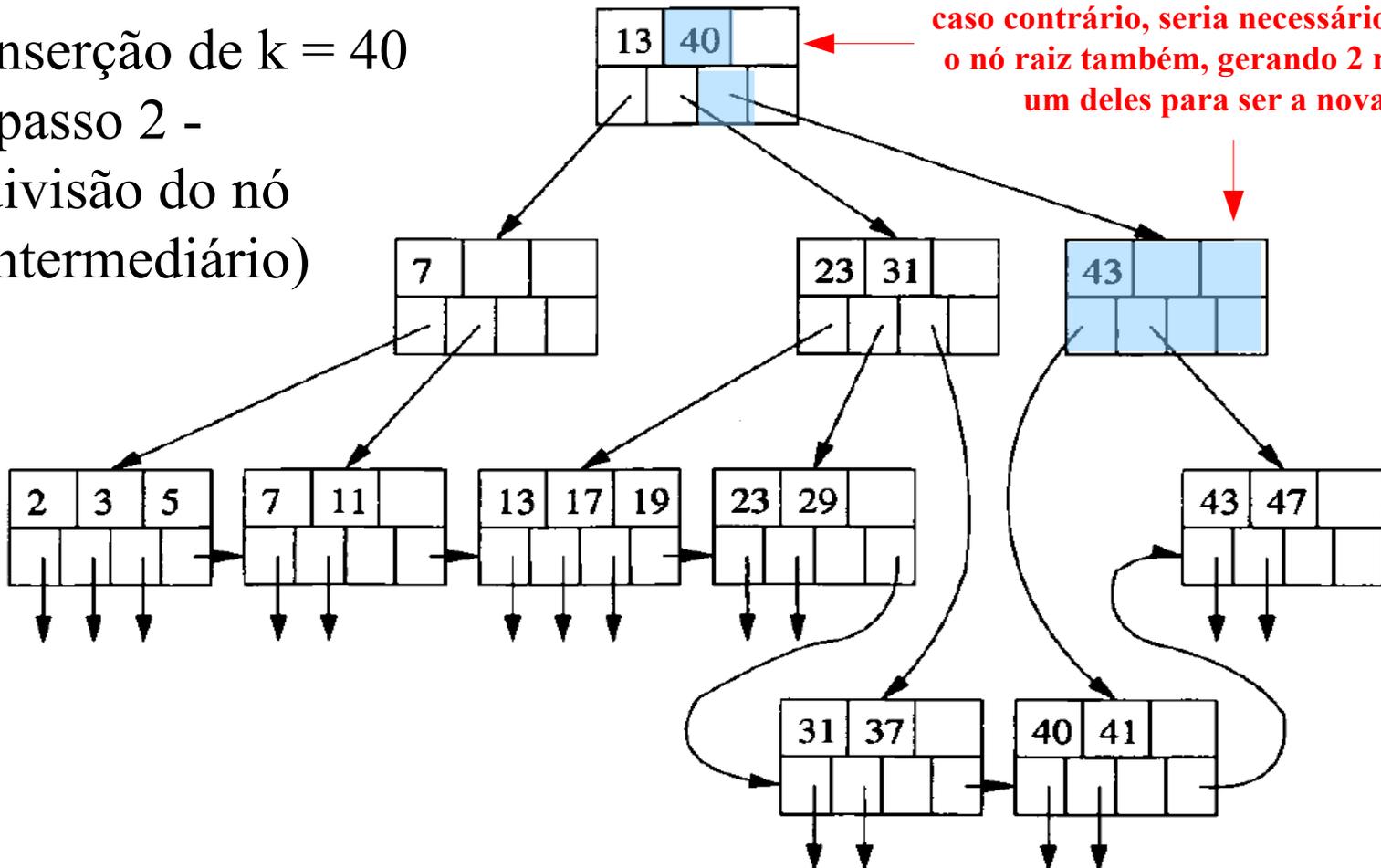


# Inserção em Árvores B+

## Exemplo 2

Por conta do novo nó interno criado neste passo, é preciso incluir na raiz um novo ponteiro. Como há espaço livre na raiz, a inclusão pode ser feita diretamente. No caso contrário, seria necessário “quebrar” o nó raiz também, gerando 2 novos nós – um deles para ser a nova raiz.

Inserção de  $k = 40$   
(passo 2 -  
divisão do nó  
intermediário)



# Inserção em Árvores B+

## (Resumo)

- 1) Insira a nova chave em um nó folha
  - a) Se houver espaço na folha apropriada para a nova chave, insira-a ordenadamente na folha
  - b) Senão, separe a folha em duas, dividindo as chaves (inclusive a nova) entre elas
- 2) A separação de nós em um nível sempre deve ser refletida no nível acima, já que um novo par chave-ponteiro precisa ser inserido num nível superior. Aplique recursivamente a seguinte estratégia para inserir no nível acima:
  - a) Se houver espaço no nó interno “pai”, insira o par nele
  - b) Senão, separe o nó interno “pai” em dois e continue a inserção árvore acima

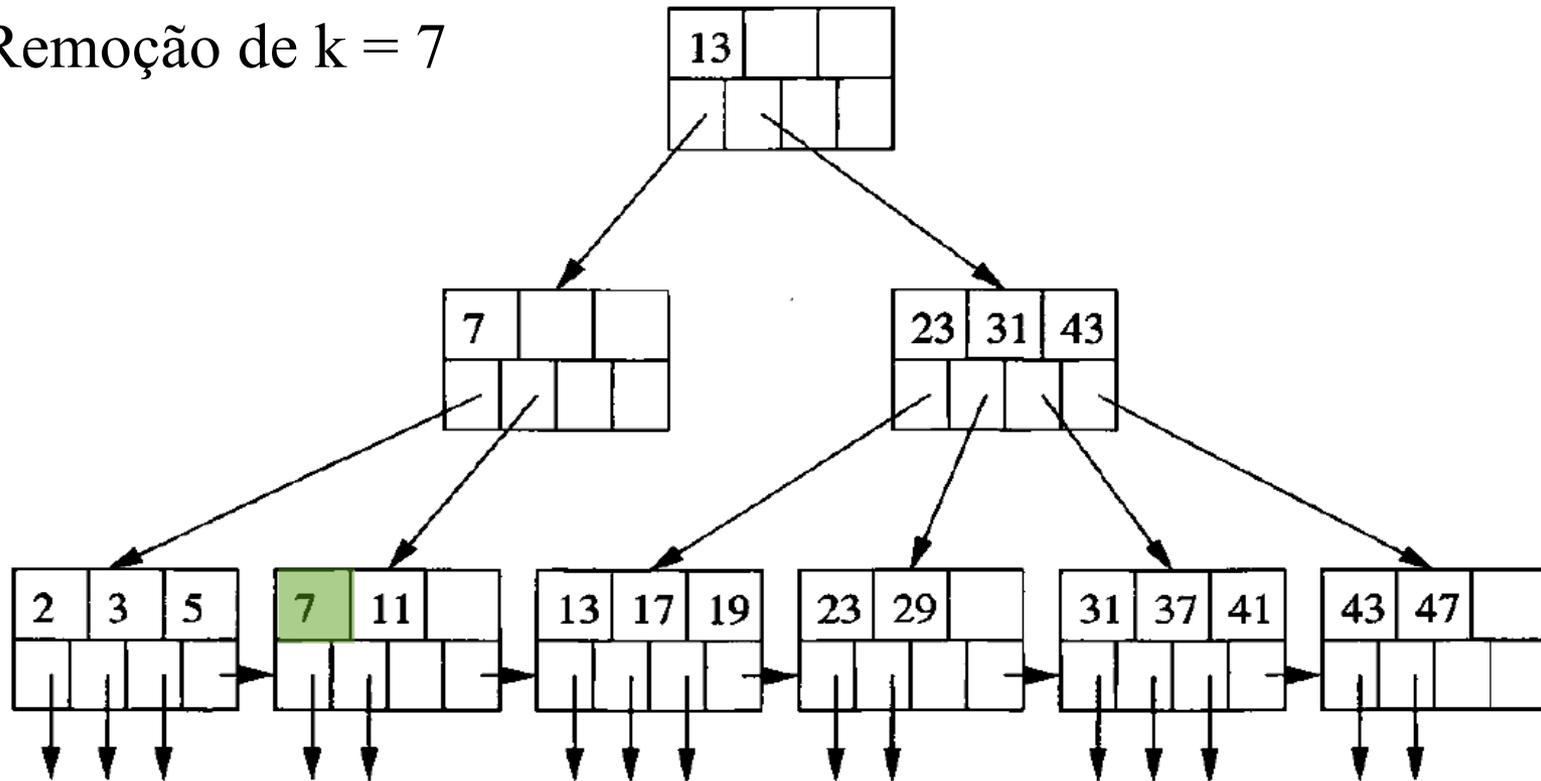
# Inserção em Árvores B+ (Resumo)

- 3) Um caso excepcional ocorre quando tentamos inserir um novo par chave-ponteiro na raiz e não há mais espaço livre
- ◆ Nesse caso, é necessário separar a raiz em dois nós e criar uma nova raiz no próximo nível superior. A nova raiz terá como filhos os dois nós resultantes da separação
  - ◆ Lembre-se de que, não importa o tamanho  $n$ , sempre é permitido que a raiz tenha uma só chave e dois ponteiros.

# Remoção em Árvores B+

## Exemplo 1

Remoção de  $k = 7$

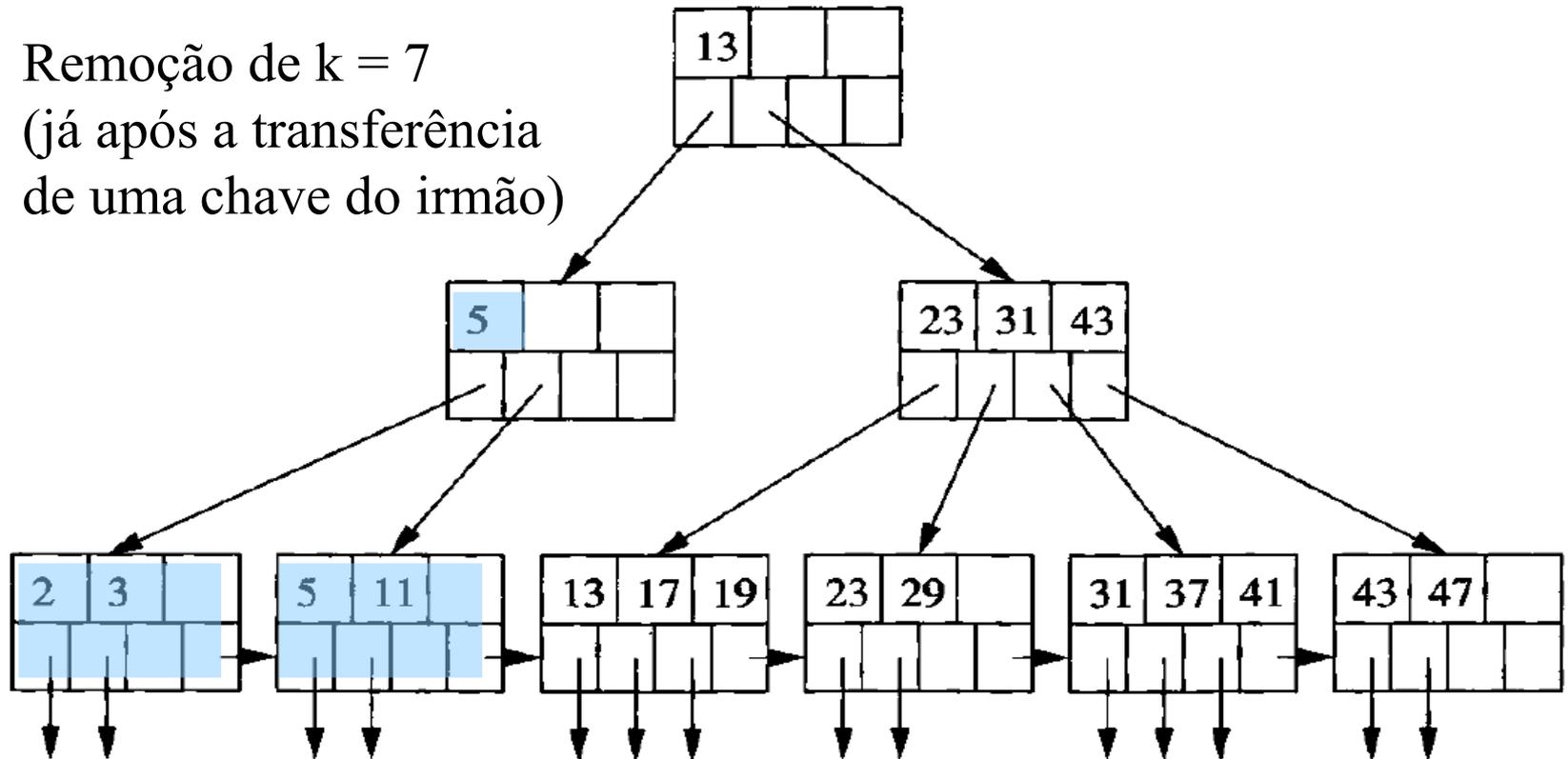


A remoção da chave 7 do nó folha onde se encontra deixará o nó com uma quantidade de ponteiros sub-mínimo ( $< \lfloor (n+1)/2 \rfloor$ ). Mas como um nó irmão seu tem ponteiros “sobrando” ( $> \lfloor (n+1)/2 \rfloor$ ), podemos passar uma das chaves desse nó irmão para o nó de onde o 7 será removido, mantendo a ordenação das chaves intacta nos nós folhas.

# Remoção em Árvores B+

## Exemplo 1

Remoção de  $k = 7$   
(já após a transferência  
de uma chave do irmão)

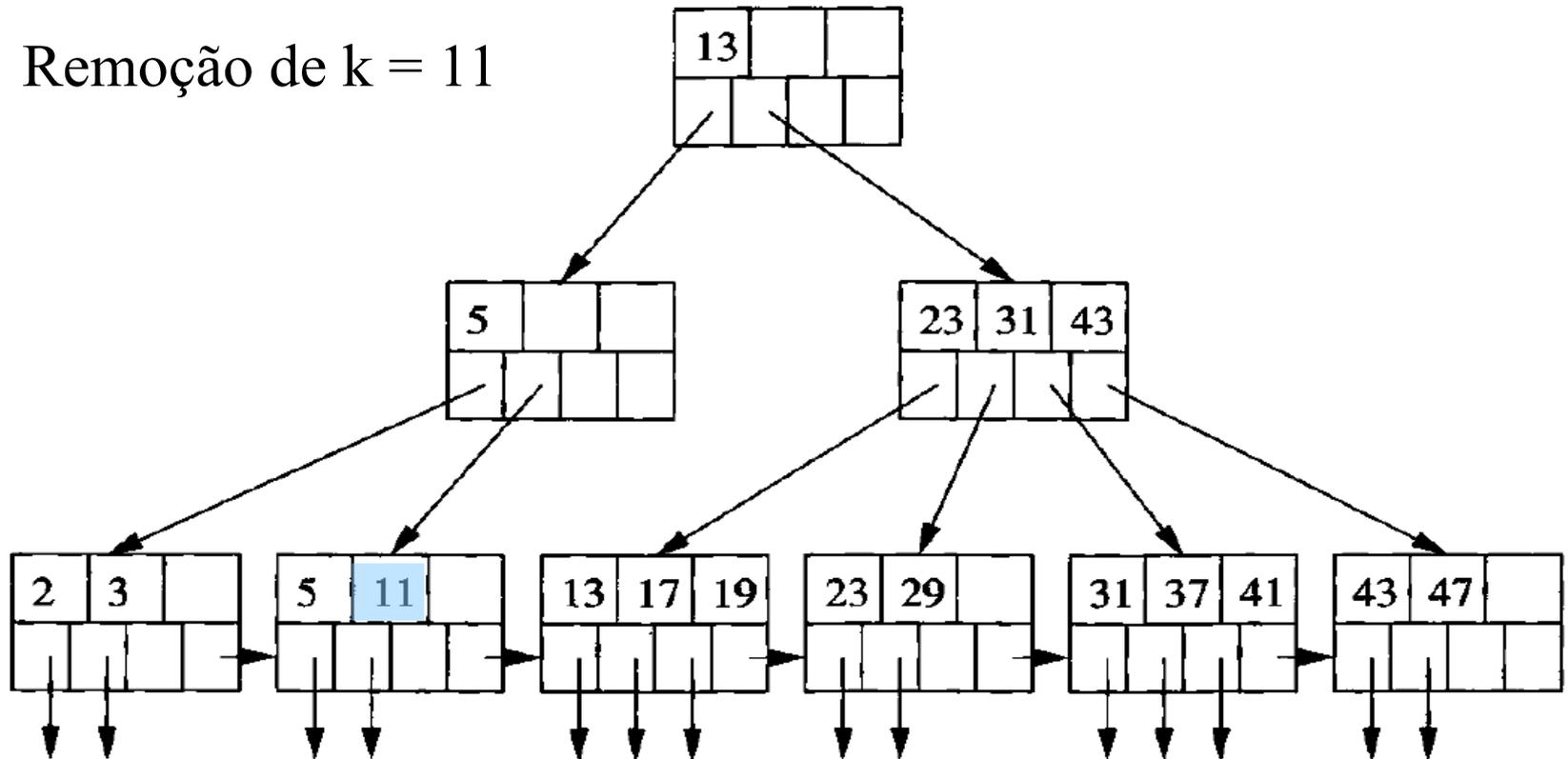


A chave 5 foi transferida do irmão (à esquerda) do nó folha onde se encontrava 7. Essa transferência só foi possível porque mesmo sem essa chave a quantidade de ponteiros no nó irmão continua  $\geq \lfloor (n+1)/2 \rfloor$ . Observe que (como ocorreu no exemplo), às vezes é preciso ajustar as chaves do nó pai dos blocos modificados por conta do remoção.

# Remoção em Árvores B+

## Exemplo 2

Remoção de  $k = 11$



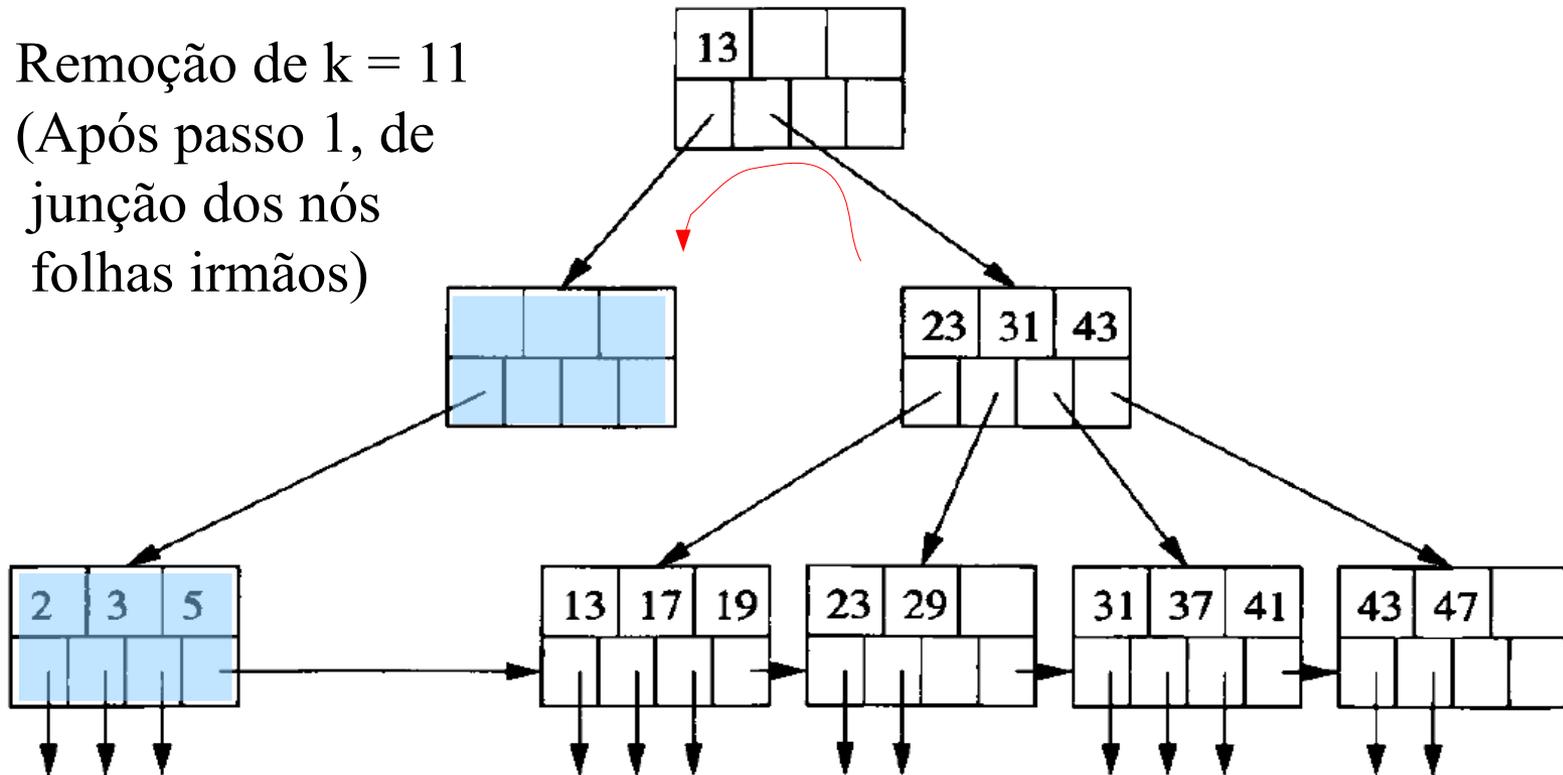
**Problema:** o nó folha onde 11 está não possui um irmão que possa transferir uma chave para ele após a remoção do 11. O irmão à esquerda possui a quantidade mínima de ponteiros em uso. E o nó de 11 não possui irmão à direita (o nó folha à direita dele não é irmão porque tem um pai diferente!).

**Solução:** as três chaves que sobram nos dois primeiros nós folhas após a remoção de 11 caberiam em um só nó. Então podemos mover a chave 5 para o primeiro nó e remover o segundo nó.

# Remoção em Árvores B+

## Exemplo 2

Remoção de  $k = 11$   
(Após passo 1, de  
junção dos nós  
folhas irmãos)



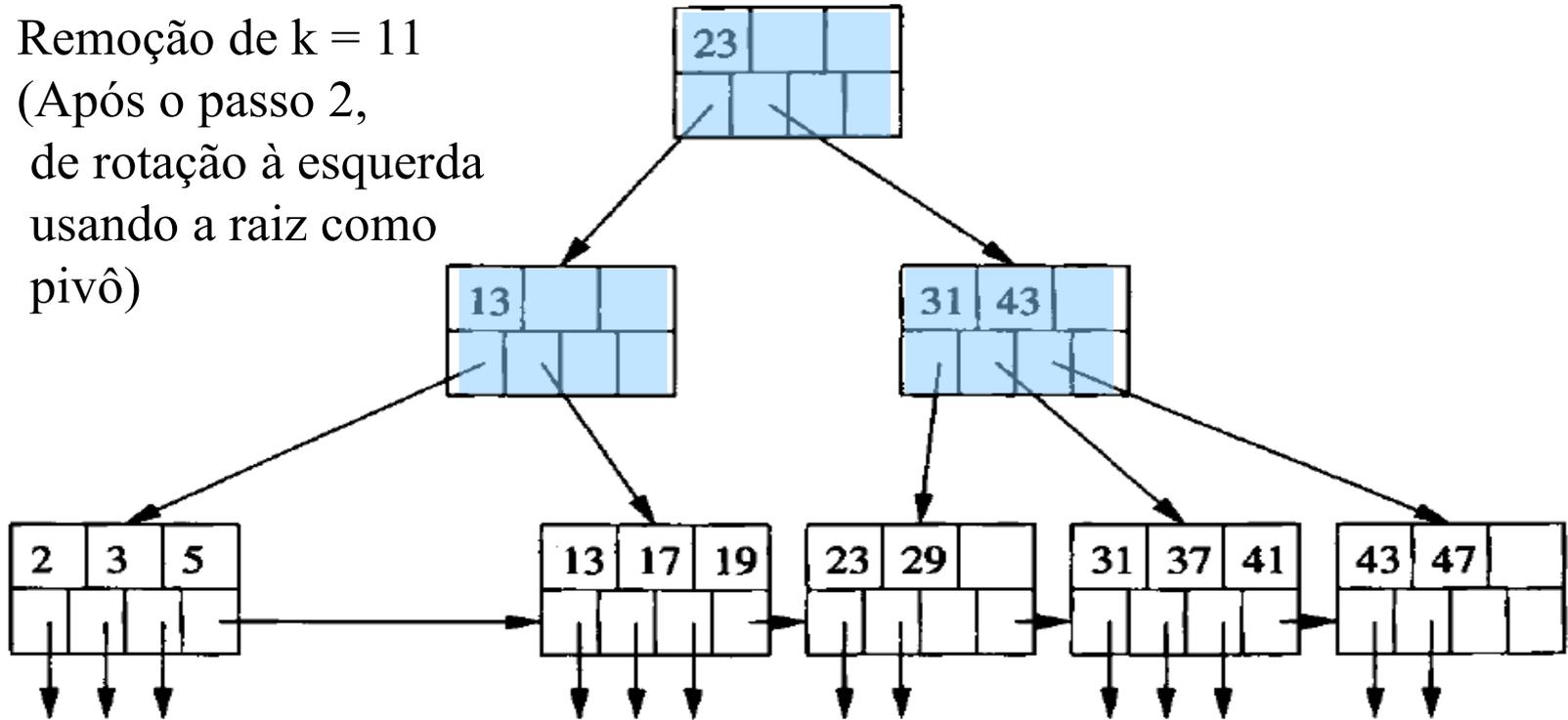
**Novo problema:** a junção dos nós folhas irmãos fez com que o nó pai deles ficasse sem nenhuma chave e 1 só ponteiro (pois antes o pai tinha somente dois filhos).

**Solução:** nesse caso do exemplo, é possível obter uma chave e um ponteiro do nó irmão à direita (que tem ponteiros “sobrando”) → **balanceamento por rotação à esquerda**

# Remoção em Árvores B+

## Exemplo 2

Remoção de  $k = 11$   
(Após o passo 2,  
de rotação à esquerda  
usando a raiz como  
pivô)



# Eficiência das Árvores B+

- ◆ As árvores B+ possibilitam a realização de buscas, inserção e remoção de registros usando **bem poucos acessos ao disco por cada operação sobre o arquivo de dados**
- ◆ Se  $n$  é grande (= o número de chaves por bloco é grande), o número de eventos que causarão separações ou junções de blocos serão raros
  - ▶ Além disso, quando operações dessas são necessárias, elas se limitam quase sempre aos nós folhas, de modo que apenas 2 nós folhas irmãos e o seu nó pai sejam afetados.
  - ▶ Por isso, **o custo (em termos de acesso a disco) da reorganização de um índice árvore B+ pode ser desconsiderado**

# Eficiência das Árvores B+

- ◆ Cada busca por registro(s) com uma dada chave requer que naveguemos do nó raiz até um nó folha da árvore
- ◆ O número de acessos ao disco (= leitura de blocos) em uma operação sobre um registro é dado por:  
***número de níveis da árvore + número de acessos necessários para a manipulação do registro***, que é
  - ▶ 1, no caso de uma leitura de registro
  - ▶ 2, no caso de uma modificação (inserção, alteração ou remoção)
- ◆ **Quantos níveis uma árvore B costuma ter?**
  - ▶ Para tamanhos de chaves, ponteiros e blocos típicos, **3 níveis são suficientes até mesmo para grandes bancos de dados!**

# Uma Ilustração para Justificar os 3 Níveis

- ◆ Suponha que:
  - ▶ Um bloco tem 4096 bytes
  - ▶ Uma chave é um inteiro de 4 bytes
  - ▶ Um ponteiro tem 8 bytes
- ◆ Para encontrar o melhor valor de  $n$  para os nós da árvore B+, fazemos:  
$$4n + 8(n+1) \leq 4096 \rightarrow n=340$$
- ◆ Suponha que um bloco médio da árvore tenha uma ocupação que fica entre o mínimo e o máximo, ou seja, um bloco típico tem 255 ponteiros.
- ◆ De uma raiz, temos 255 filhos e  $255 * 255 = 65025$  folhas. E essas folhas têm um total de  $255 * 255 * 255$  ponteiros para registros
  - ▶ **Portanto, arquivos de dados com cerca de 16,6 milhões registros podem ser indexados por uma árvore B+ de apenas 3 níveis.**

# Referências Bibliográficas

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