6. Estruturas de Índice

Considere que você deseje localizar um livro na biblioteca. A pesquisa sobre a posição do livro na estante pode ser feita usando o arquivo de fichas, ordenados por autor ou pelo titulo do livro. Observe que o fichário organizado por autor não pode ser utilizado para auxiliar a localização do livro pelo seu titulo e vice-versa.

A mesma idéia é aplicada quando queremos suportar recuperação de informações de forma eficiente de em um arquivo de dados.

Cada arquivo de índice tem uma chave de pesquisa associada, que é uma coleção de um ou mais campos do registro do arquivo de dados, sobre a qual é construído o índice. Qualquer subconjunto de campos pode ser uma chave de pesquisa.

Um arquivo de índice é construído para aumentar o desempenho em pesquisas por igualdade ou por intervalo de seleção sobre a chave de pesquisa.
Cada índice aumenta o desempenho de certos tipos de pesquisa mas não todos.

Exemplo 6.1: considere a seguinte relação:
cadastro-livros(*cod* char(10), titulo char (40), autor char(30) localização char(10))

Uma instancia da relação é mostrada na Figura 6.1.

	titulo	autor	estante
001	AB	ZE	E1
022	BA	AC	E8
003	CB	CK	E3
004	FF	MA	E2
015	HH	KK	E5
006	JJ	Joao	E4
007	MM	PP	E6
018	NN	WW	E7
009	Um dia	DD	E9

Figura 6.1 – instancia da relação cadastro-livros ordenado por livros

Um **arquivo de índice (ou índice)** sobre um arquivo de dados é uma estrutura auxiliar cujo objetivo é aumentar o desempenho na execução de operações que não são suportadas eficientemente pela organização básica do arquivo de dados.

Exemplo 6.2

SELECT localização
FROM cadastro-livros
WHERE autor = ‘João’

SELECT localização
FROM cadastro-livros
WHERE titulo = ‘Um dia’

Do ponto de vista de implementação um índice é um arquivo como qualquer outro, contendo registros (entradas) que direcionam a consulta para os registros de dados.

Os registros ou entradas de um índice podem apresentar os seguinte formato:
< *argumento de pesquisa, rid*>, onde *rid* identifica o registro no arquivo de dados que satisfaz a condição. A entrada de um arquivo de índice será denotada por k*. As entradas são organizadas de tal maneira a facilitar a recuperação de todos os registros do arquivo de dados cuja chave de pesquisa é k.

Exemplo 6.3 : arquivos de índice para os atributos titulo e autor da instancia da relação cadastro-livros dada na Figura 6.1.

titulo

AB	<0,0>
BA	<0,1>
CD	<0,2>
FF	<1,0>
HH	<1,1>

JJ	<1,2>
MM	<2,0>
NN	<2,1>
Um dia	<2,2>

arquivo de dados

001	AB	ZE	E1
022	BA	AC	E8
003	CB	CK	E3

004	FF	MA	E2
015	HH	KK	E5
006	JJ	Joao	E4

007	MM	PP	E6
018	NN	WW	E7
009	Um dia	DD	E9

autor

<0,1>	AC
<0,2>	CK
<2,2>	DD
<1,2>	Joao
<1,1>	KK

<1,0>	MA
<2,0>	PP
<2,1>	WW
<0,0>	ZE

Figura 6.1 – Indice para os atributos autor e titulo

6.1 Fatores que contribuem para tornar a pesquisa baseada em índice mais eficiente

- as entradas do índice são menores que o registro de dados (menor número de páginas), neste caso é esperado que os registros solicitados sejam recuperados com menos I/Os do que seria necessário se a pesquisa fosse feita diretamente no arquivo de dados;

- o índice pode ser pequeno o suficiente para caber permanentemente na memória o que significa que localizar o argumento de pesquisa ou a página onde ele se encontra envolve apenas acessos à memória principal.

6.2 Propriedades de um Arquivo de Índice

6.2.1. Agrupado X Não Agrupado (clustered X unclustered)

a) índice agrupado – um índice é dito agrupado quando o arquivo é organizado de tal forma que a ordem dos registros no arquivo de dados é a mesma ou bem próxima da ordem das entradas no arquivo de índice. Os índices agrupados são computacionalmente mais caros para se manter quando atualizados. Pode haver no máximo um índice agrupado por arquivo de dados.Veja índice associado à chave de pesquisa *titulo* no exemplo 6.3.

b) Índice não agrupado – as entradas no índice não coincidem com a ordem dos registros no arquivo de dados. Podem haver vários *índices não agrupados* sobre um arquivo de dados. Veja índice associado à chave de pesquisa *autor* no exemplo 6.3.

6.2.2. Denso X Esparso

a) Denso – há uma entrada no índice que aponta para cada registro no arquivo de dados. A localização do registro solicitado é mais rápida, no entanto ocupa mais espaço. O arquivo de dados só é consultado se o argumento de pesquisa aparece no índice. Um índice denso pode ser agrupado ou não agrupado. Os dois índices apresentados no exemplo 6.3 são densos.

b) Esparso – nem todos os registros de dados tem uma entrada correspondente no arquivo de índice. Em geral, há uma entrada no índice por página do arquivo de dados. Neste caso, durante uma consulta, o arquivo de dados sempre deve ser consultado. Para que um índice seja esparso ele deve ser agrupado, sendo assim, é possível construir apenas um índice esparso para cada arquivo de dados. Índices esparsos exigem menos espaço de armazenamento e menor *overhead* durante a modificação do arquivo de dados. Veja o arquivo de índice esparso associado à chave de pesquisa titulo na Figura 6.2.

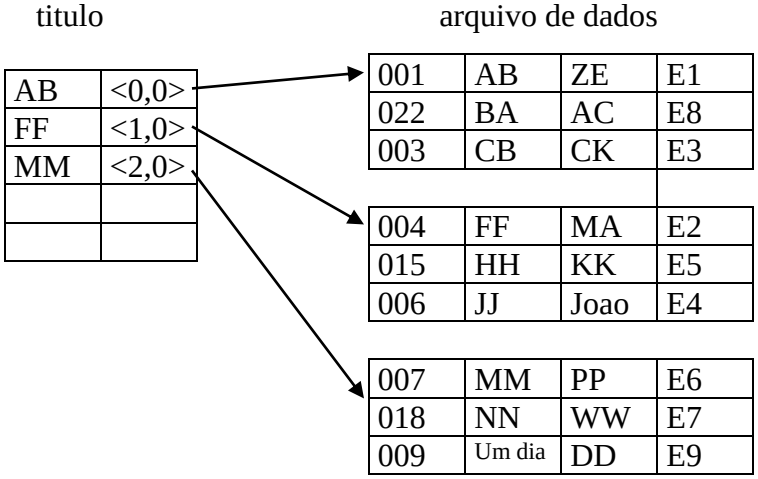


Figura 6.2 – índice esparso construído para o atributo título do arquivo de dados dado na Figura 6.1

6.2.3 Indice primário X Índice Secundário

a) índice primário – é um índice ordenado pela mesmo atributo que o arquivo de dados. O termo índice primário é muitas vezes utilizado para designar um índice associado à chave primaria, mas esta nomenclatura não é padrão. Os índices primários podem ser *densos ou esparsos*, e são sempre *agrupados*. O índice titulo do exemplo 6.3 é primário, agrupado e denso. O índice sobre o atributo titulo da Figura 6.2 é primário, agrupado e esparso.
b) índice secundário – é um índice cuja ordem das entradas não mantém qualquer correspondência com a ordem dos registros correspondentes no arquivo de dados. Um índice secundário é sempre *denso, não agrupad.*

Ambos os tipos de índice podem conter *entradas duplicadas*. Se o índice está associado a uma chave candidata, e portanto não possui entradas duplicadas, ele é dito *índice único*.

6.3 Métodos de Acesso

Técnicas de organização ou estruturas de dados para arquivos de índice são chamadas de *métodos de acesso*.

Os métodos de acesso são *unidimensionais* quando a chave de pesquisa associada ao arquivo de índice contém apenas um campo (ex: arquivos *hash*, árvores B+, arquivos ordenados)

Os métodos de acesso são ditos *multidimensionais* quando a chave de pesquisa associada ao arquivo de índice contém mais de um campo do registro de dados (ex: K-d-tree, grid, família R-tree, entre outros).

Neste capítulo trataremos apenas dos métodos de acesso unidimensionais. O enfoque será dado à construção de índices baseados em arquivos ordenados, baseados em árvore B+ e baseados em tabela hash.

6.3.1. Índice baseado em arquivos ordenados

Os argumentos de pesquisa do índice são mantidos ordenados pela chave de pesquisa. Os índices podem ser primários ou secundários.

a) índices primários

- índice está ordenado pela mesma chave que o arquivo de dados;
- podem ser:

i) denso

- para encontrar um argumento de pesquisa K em um índice denso é realizada uma pesquisa binária para encontrar o argumento de pesquisa. Um índice com N páginas resultaria numa pesquisa com custo da ordem de $\log_2 N$;

ii) *índices primários esparsos*

- requer menos espaço que o índice denso;
- o índice esparso é sempre agrupado (só pode haver um índice esparso por arquivo de dados);
- para encontrar um argumento de pesquisa K em um índice esparso procura-se pelo maior argumento que seja menor ou igual K (a pesquisa binária pode ser utilizada). Uma vez encontrada a entrada no índice, a página, no arquivo de dados, é acessada. O(s) registro(s) que conté(ê)m K é(são) recuperado(s) através de uma pesquisa na página de dados ⇨ acesso ao registro solicitado é mais lento, veja Figura 6.2.

iii) *índices primários com múltiplos níveis*

Mesmo usando índice esparso, o próprio índice poderia tronar-se muito grande para ser processado de forma eficiente. Neste caso o próprio índice é tratado como um arquivo de dados, criando-se índice para o arquivo de índice. Veja exemplo 6.4. Observe que o nível interno pode ser denso ou esparso e o nível externo sempre é esparso para garantir uma redução no tamanho do índice.

Exemplo 6.4: a seguir será construído um índice interno denso e externo esparso e outro com os dois níveis com índices esparsos.

AB	<0,0>
JJ	<1,0>

AB	<0,0>
BA	<0,1>
CD	<0,2>
FF	<1,0>
HH	<1,1>

JJ	<1,2>
MM	<2,0>
NN	<2,1>
Um dia	<2,2>

001	AB	ZE	E1
002	BA	AC	E8
003	CB	CK	E3

004	FF	MA	E2
005	HH	KK	E5
006	JJ	Joao	E4

007	MM	PP	E6
008	NN	WW	E7
009	Um dia	DD	E9

2^o. nível 1^o. nível arquivo de dados

Figura 6.3 – arquivo de índice multiníveis (denso /esparso)

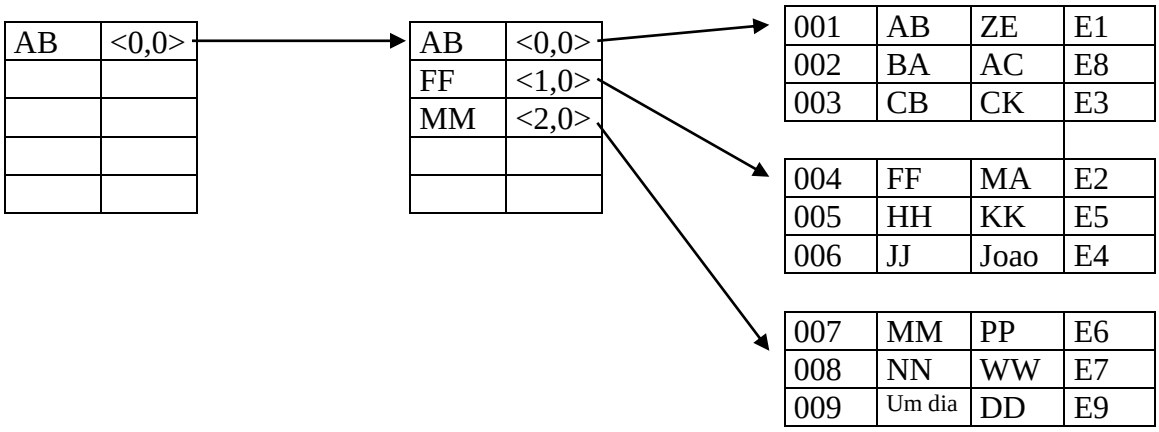


Figura 6.4 – arquivo de índice multiníveis (esparso/esparso)

iv) índices com argumentos de pesquisa duplicados

índices são frequentemente utilizados para atributos que não são chaves primárias, sendo assim é possível que mais de um registro tenha o mesmo argumento de pesquisa.

Uma solução simples é criar um índice primário denso e duplicar os argumentos de pesquisa, veja exemplo 6.5. Um indice esparso também pode ser utilizado, veja exemplo 6.6.

Exemplo 6.5 considere o arquivo de dados dado na Figura 6.5, o índice para este arquivo apresenta uma maneira de tratar entradas duplicadas.

AB	<0,0>
AB	<0,1>
CB	<0,2>
CB	<1,0>
CB	<1,1>

JJ	<1,2>
JJ	<2,0>
NN	<2,1>
PF	<2,2>

001	AB	ZE	E1
002	AB	AC	E8
003	CB	CK	E3

004	CB	MA	E2
005	CB	KK	E5
006	JJ	Joao	E4

007	JJ	PP	E6
008	NN	WW	E7
009	PF	DD	E9

Figura 6.5- índice denso com duplicação de entradas

Uma solução mais eficiente é ter apenas uma entrada para cada argumento de pesquisa K no índice. Esta entrada está associada com um ponteiro que aponta para a primeira ocorrência de K no arquivo de dados, as outras ocorrências são recuperadas pesquisando o próprio arquivo de dados, veja Figura 6.6.

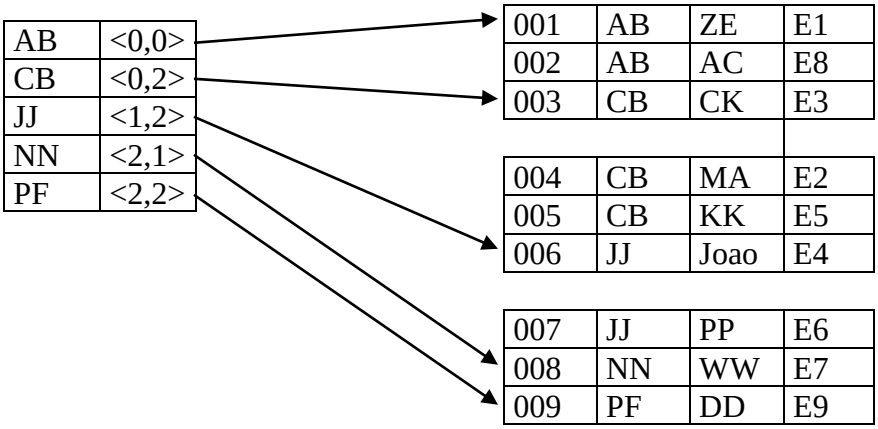


Figura 6.6 – índice denso sem duplicação de entradas

Exemplo 6.6 considere o arquivo de dados dado na Figura 6.5, podemos construir um índice esparso que trata entradas duplicadas, veja Figura 6.7.

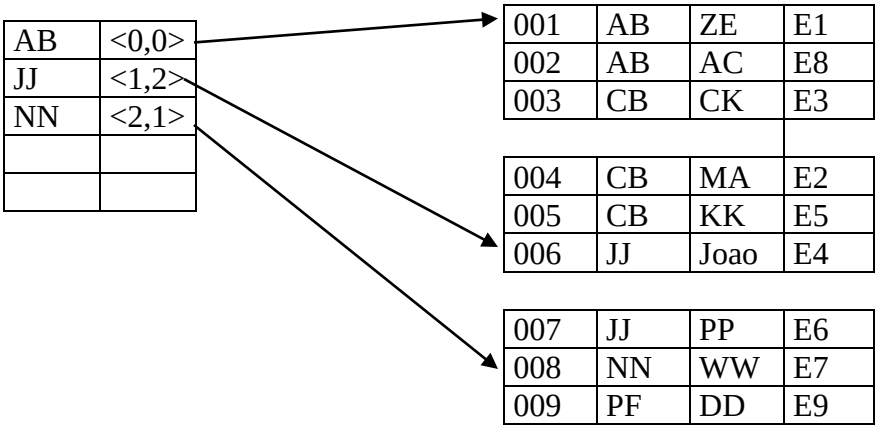


Figura 6.7 – índice esparso com duplicação de entradas

v) gerenciamento do índice primário durante a modificação do arquivo de dados

Quando ocorrem alterações no arquivo de dados, estas alterações devem se refletir no arquivo de índice.
A forma como o arquivo de índice será afetado depende se ele é denso ou esparso.
Um índice ordenado pode ser tratado como um arquivo ordenado pela chave de pesquisa. Sendo assim, as mesmas estratégias usadas para manter um arquivo de dados ordenado podem ser empregadas para manter arquivos de índice:

- quando não há mais espaço em um bloco para suportar uma nova inserção, os registros devem ser movimentados;
- criar blocos de *overflow* se espaço extra é necessário ou removê-los se os registros removidos daquele bloco não são mais necessários;
- inserir novos blocos no arquivo de índice.
-

Exemplo 6.7: realizaremos inserções e remoções sobre o arquivo de dados mostrado na Figura 6.8 e seus reflexos em arquivos de índice. Considere que: i) os registros do arquivo de dados sejam de tamanho fixo; ii) registros possam se movimentar dentro da página; iii) a cada remoção de um registro seja feita a compactação dos dados; iv) ao inserir um registro no arquivo de dados, e não houver lugar na página, páginas de *overflow* serão criadas (o diretório de *slots* é mantido na página principal), *v)* marcação de registros removidos (lápide) não serão considerados.

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Figura 6.8 arquivo de dados

A Figura 6.9 mostra um índice denso construído para o arquivo de dados da Figura 6.8:(lembre-se que o *rid* é dado em termos do diretório de slots)

10	<0,0>
20	<0,1>
30	<1,0>
40	<1,1>

50	<2,0>
60	<2,1>
70	<3,0>
80	<3,1>

90	<4,0>
100	<4,1>

Figura 6.9 – índice denso, agrupado e primário

A remoção do registro cuja chave de pesquisa é K=30 provoca a modificação apresentada na Figura 6.10.

10	<0,0>
20	<0,1>
40	<1,1>

10	
20	

40	

Figura 6.10 – reflexos que uma remoção causa em um arquivo de índice denso

A inserção de um registro cuja chave de pesquisa é K=15 provoca os reflexos em um indice denso de acordo com a Figura 6.11

10	<0,0>
15	<0,2>
20	<0,1>
40	<1,1>

10	
15	

→

20	

40	

Figura 6.11 – inserção de página adicional no arquivo de dados

Agora ao inserir o registro com chave de pesquisa K=25 no arquivo de dados, serão necessárias páginas adicionais no índice, como mostrado na Figura 6.11, veja Figura 6.12.

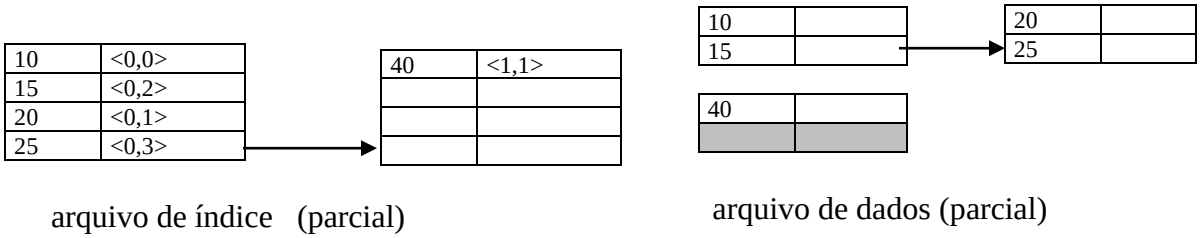


Figura 6.12 – inserção de uma página adicional no arquivo de índice

Agora vamos avaliar como as alterações no arquivo de dados se refletem em um índice esparso. Considere o índice esparso mostrado na Figura 6.13 construído para arquivo de dados da Figura 6.8

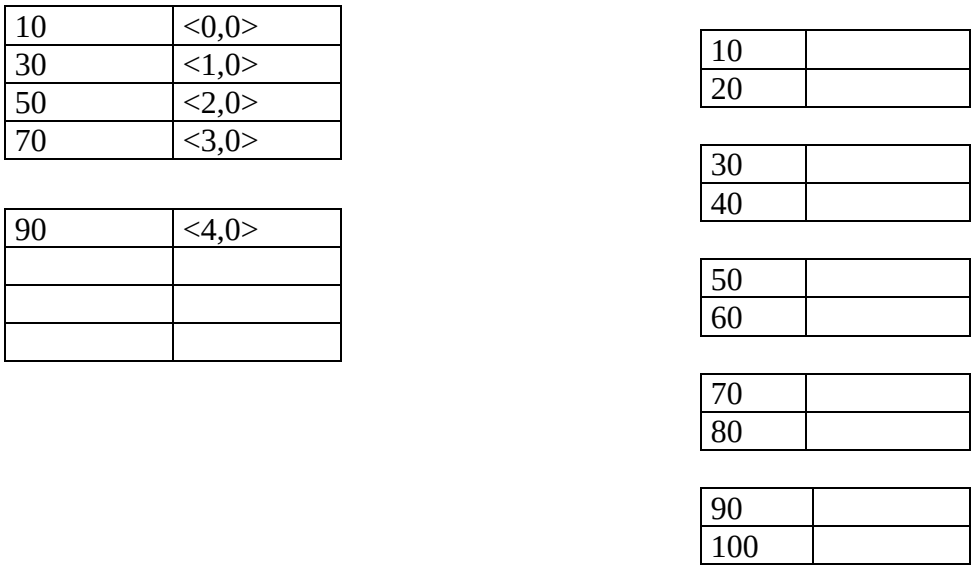


Figura 6.13 – índice esparso para o arquivo de dados

A remoção do registro cuja chave é k=30 modifica o índice pois o registro a ser removido é o primeiro registro da página, veja Figura 6.14.

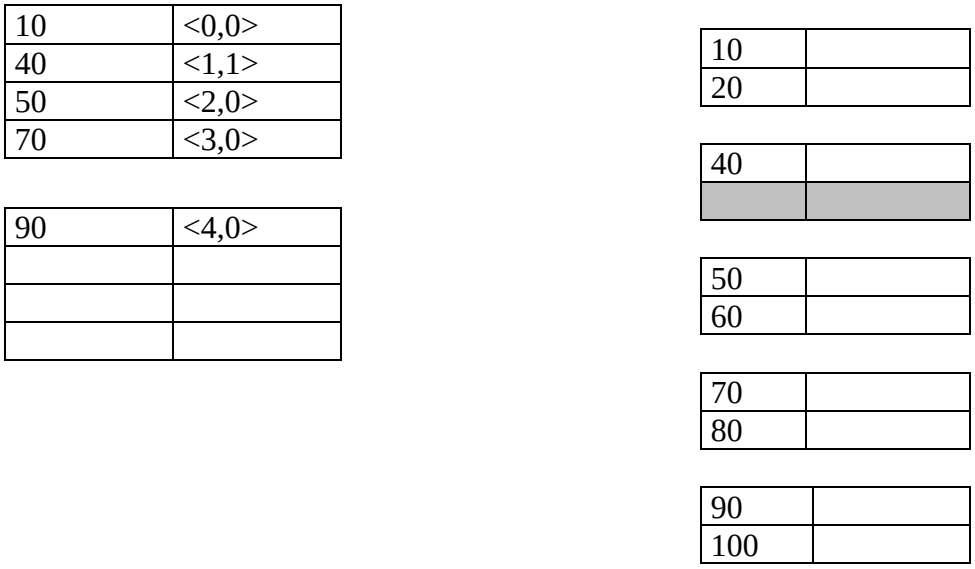


Figura 6.14 – modificação do arquivo de dados e o seu reflexo no índice esparso

O resultado da remoção do registro cuja chave é k=40 é mostrado na Figura 6.15.

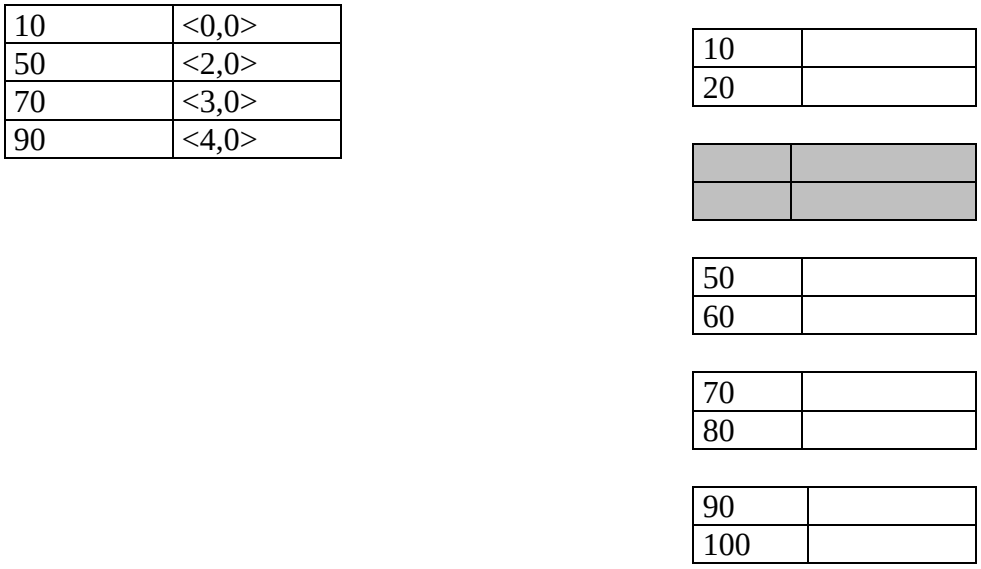


Figura 6.15- a modificação causada no arquivo de dados permite a liberação da página 1.

A inserção do registro cuja chave é K=15 não causa modificação no índice esparso porque páginas de overflow não são endereçadas diretamente pelo índice, veja Figura 6.16

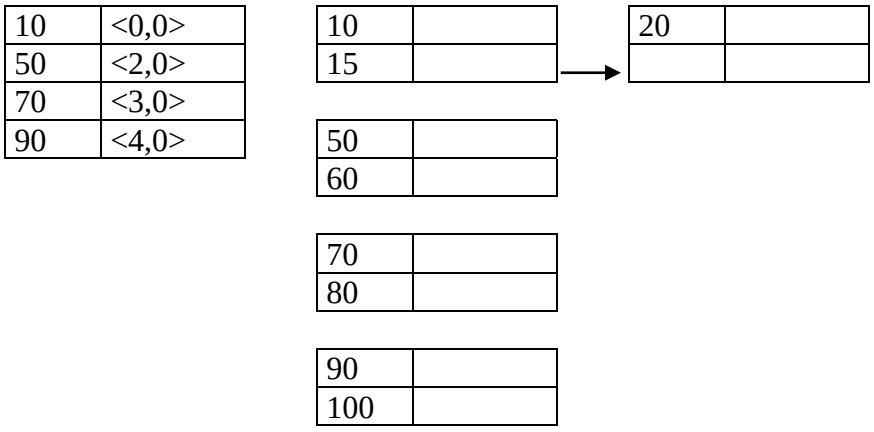


Figura 6.16- a inserção de um registro em pagina de *overflow* não modifica um índice esparso

A tabela 6.1 resume as ações que devem ser tomadas para atualizar o arquivo de índice ordenado, dependendo de ser denso ou esparso.

Operação	**Índice denso**	**Índice esparso**
Cria um bloco de overflow vazio	nenhuma	nenhuma
Remove um bloco de overflow vazio	nenhuma	nenhuma
Cria um bloco no arquivo de dados	nenhuma	Insere
Remove um bloco do arquivo de dados	nenhuma	Remove
Insere registro	Insere	Atualiza(?)
Remove registro	Remove	Atualiza(?)
Movimenta registro	atualiza	Atualiza(?)

Tabela 6.1 – resumo dos reflexos que as alterações em um arqivo de dados causa no índice

- criar e destruir um bloco de *overflow* vazio não tem efeito em qualquer um dos tipos de índices
- criar e remover blocos do arquivo de dados ordenado (arquivo seqüencial) não tem qualquer efeito sobre índices densos, porque este índice refere-se a registros e não blocos. Afeta o índice esparso uma vez que uma entrada no índice deve ser inserida ou removida se um bloco é inserido ou removido do arquivo seqüencial
- inserir ou remover registros no arquivo seqüencial afeta o índice denso (entradas devem ser inseridas ou removidas). Só afeta o índice esparso se o registro for inserido ou removido na/da primeira posição no bloco. Neste caso, o valor da entrada no índice esparso deve ser modificada.

w) índices secundários

Frequentemente são necessários diversos índices sobre um arquivo de dados para facilitar uma variedade de consultas;

Índices secundários são *não agrupados* e são sempre *densos*. Em geral possuem argumentos de pesquisa duplicados.

i) *projeto de um índice secundário*
Um índice secundário é sempre denso e não agrupado, veja a Figura 6.17

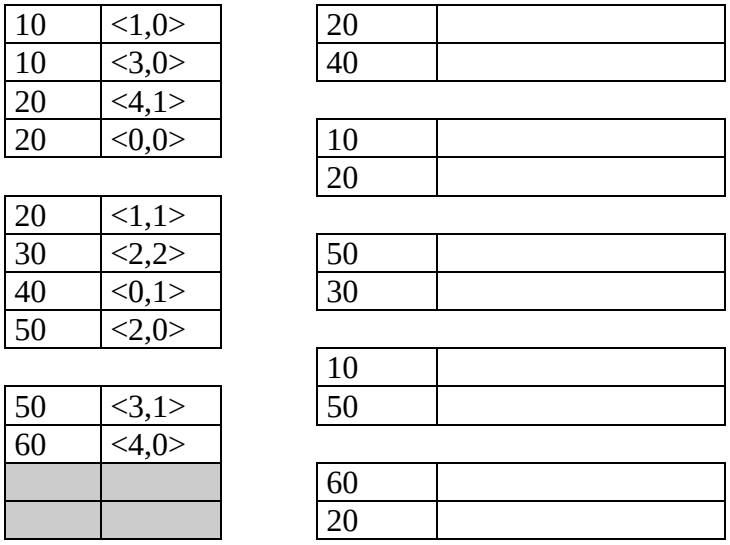


Figura 6.17 – índice secundário

Os ponteiros em um bloco no índice podem endereçar diferentes blocos no arquivo de dados, neste caso um maior número de I/Os são necessários para recuperar informações se comparado com um índice primário

ii) *índices secundários com argumentos de pesquisa duplicados*
uma maneira conveniente de evitar duplicação de argumentos de pesquisa no índice é usar um nível de indireção, chamados buckets, entre o índice e o arquivo de dados. Veja Figura 6.18:

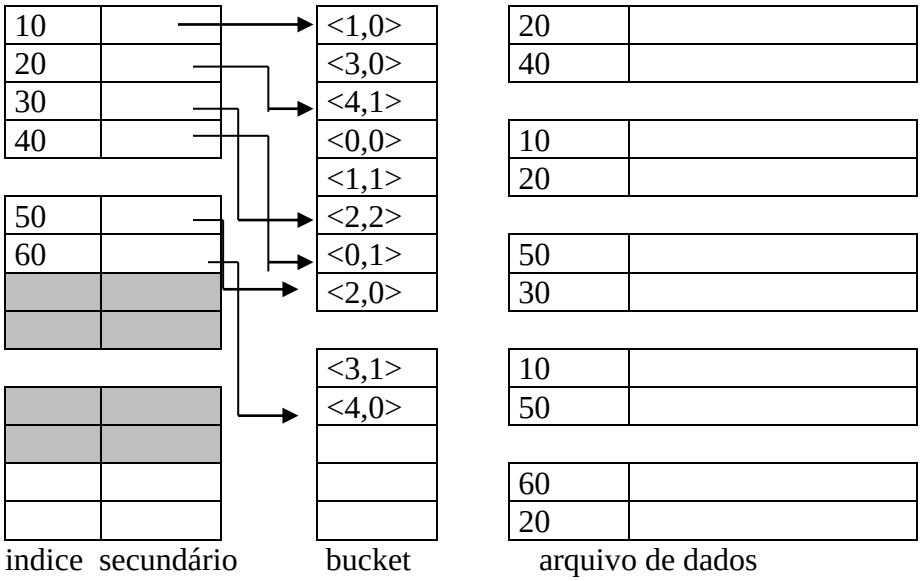


Figura 6.18 – uso de indireção para tratar duplicação de chaves de pesquisa

vantagens no uso de indireção do índice:

a) argumentos são maiores que ponteiros ⇨ economiza espaço no arquivo de índice ;
b) auxilia na consulta em situações em que há várias condições, e cada condição possui um indice secundário

Exemplo 6.8 considere a relação e a consulta dadas abaixo :

cinema(*cod*,título, ano, duração, nome-estúdio)

```
SELECT titulo
FROM cinema
 WHERE nome-estúdio="Disney" and ano=1995
```

Se há indice secundário com indireção para os atributos nome-estúdio e ano, então:

1. encontra os ponteiros no bucket do indice nome-estudio que satisfaça a condição *nome-estúdio="Disney"* (por exemplo as tuplas <0,2>,<1,4> e <3,5>);
2. encontra os ponteiros no bucket do índice ano que satisfaça a condição *ano=1985* (por exemplo <3,2>, <1,4>, <3,4>,<3,5>);
3. o resultado da consulta são as tuplas da intersecção <1,4> e <3,5>.

iii) índices secundários com múltiplos níveis

neste caso, somente índices esparsos podem ser usados para um segundo nível de índices.

6.3.2. Índices baseados em árvore B+

- Índices baseados em árvore B+, são os mais utilizados em SGBD comerciais;
- Suportam pesquisas com seleção de igualdade e de intervalo são realizadas de forma eficiente;
- Garantem ocupação mínima de 50% para cada nó da árvore;
- Mantém, de forma automática, tantos níveis de índice quanto necessários, de acordo com o tamanho do arquivo sendo indexado. Cada nível interno forma um índice esparso. O nível de nós folha pode formar um índice denso ou esparso.

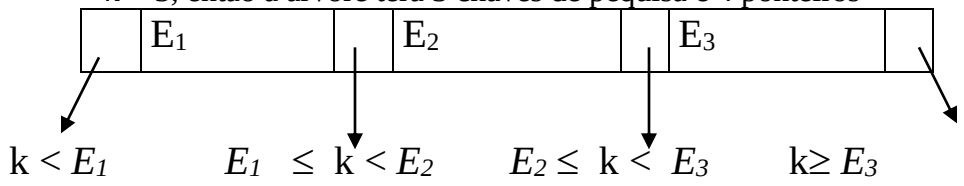
6.3.2.1. A estrutura da árvore B+

- Organiza seus blocos (ou páginas) em uma árvore (cada nó é uma página);
- A árvore é balanceada (todos os caminhos da raiz até uma folha tem o mesmo comprimento)
- A árvore possui *nós não folha* (ou nó índice) e *nós folha*. O nó *raiz* é um nó não folha especial;
- A cada índice baseado em árvore B+ é associado um valor *n*. O valor *n* é dito ser a ordem a da árvore e determina que cada nó não folha da árvore B+ terá no máximo, **n** chaves de pesquisa e **n+1** ponteiros. O número de ponteiros em uma árvore B+ é chamado *fanout* (espalhar-se) do nó. (autores diferentes têm definições diferentes para a ordem da árvore, estamos nos baseando no livro de Garcia-Molina):

    i. Cada chave de pesquisa E_i no nó não folha funciona como uma chave divisora: todas as chaves *K* da sub-árvore à esquerda de E_i terão valores menores que E_i e todas as chaves *K* à direita de E_i terão valores maiores ou iguais a E_i e menores que E_{i+1};

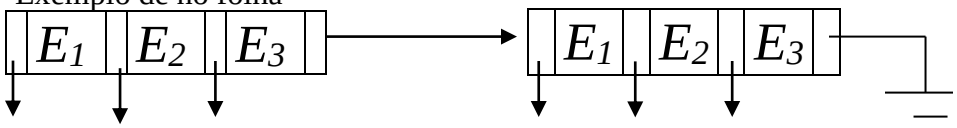
Exemplo de nó não folha

n = 3, então a árvore terá 3 chaves de pequisa e 4 ponteiros



ii. O nó folha possui *n* pares <E_i, *rid*>, onde cada chave de pesquisa E_i no nó folha representa um argumento de pesquisa de um registro armazenado no arquivo de dados e o *rid* o seu endereço. O *(n+1)ésimo* ponteiro é utilizado para ligar os nós folhas da árvore.

Exemplo de nó folha



iii. a ocupação mínima do nó folha é de $\lfloor (n+1)/2 \rfloor$ ponteiros, e dos nós não folha é de $\lceil (n+1)/2 \rceil$ ponteiros , exceto a raiz que pode conter apenas dois ponteiros.

Dado o arquivo de dados:

10	
20	

30	
40	

50	
60	

70	
80	

90	

Figura 6.19 – arquivo de dados

A árvore B+ de ordem n=2 para o arquivo de dados mostrado na Figura 6.19 é exibida na Figura 6.20.

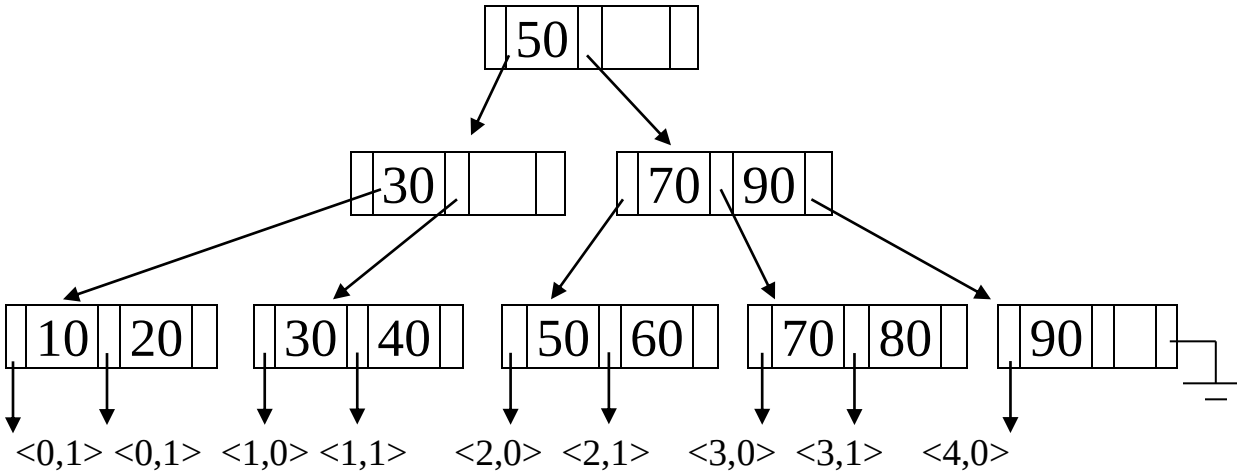


Figura 6.20 – Um árvore B+ de ordem n=2

6.3.2.2 Operações sobre arvore B+

a) Consulta

Toda consulta por um argumento de pesquisa deve começar pelo nó raiz N.

i. pesquisa por igualdade

seja K o argumento de pesquisa;
cada chave de pesquisa E_i do nó N é avaliada até encontrar a sub-árvore de N que satisfaz a condição : $E_i <= K < E_i + 1$;
o nó pai da sub-árvore encontrada passa a ser o nó N;
este procedimento continua recursivamente até que um nó folha seja alcançado. Neste caso as chaves de pesquisa do nó folha são pesquisadas até encontrar uma que seja igual a K ou até que todas as chaves de pesquisa do nó folha tenham sido pesquisadas.

ii. pesquisa por seleção de intervalo

seja [K1,K2] o intervalo de seleção;
é realizada uma busca por igualdade usando o argumento K1;
uma vez encontrado o nó folha que deve conter K1, procurar pela primeira ocorrência de K1 ou de um argumento que possua o menor valor maior que K1;
uma vez localizado o primeiro argumento do intervalo de seleção, os outros são obtidos percorrendo os nós folha, até que seja encontrada uma entrada cujo valor seja maior que K2.

b) Inserção

Considere que desejamos inserir uma entrada <K,rid> em uma árvore B+;
Uma pesquisa por igualdade deve ser realizada para encontrar o nó apropriado para o argumento de pesquisa K;

Uma vez encontrado o nó folha N onde a entrada <K,rid> será inserida, duas situações que podem ocorrer :

b1) o nó N tem lugar para suportar a inserção. Neste caso a entrada é inserida em N, de forma ordenada. Por exemplo vamos inserir na árvore dada na Figura 6.18 a entrada com o valor de K=100. Neste caso a entrada <100,rid> será inserida na quinta folha.

b2) o nó N não tem lugar para suportar a inserção.
neste caso o nó folha precisa ser dividido quando ocorre uma divisão no nó folha, esta divisão pode se propagar para os níveis mais altos da árvore.
Para ilustrar, vamos inserir na árvore da Figura 6.20 uma chave de pesquisa com o valor de K=15. Fazendo a busca, descobrimos que a entrada <K,rid> deve ser inserida é o primeiro nó folha da árvore, que denotamos por N. Não há lugar neste nó e ele precisa ser dividido. Podemos utilizar a seguinte estratégia para dividir um nó folha:

- criar um nó M a direita de N;
- ordenar as n+1 chaves de pesquisa de N, ou seja (10,15,20);
- colocar os $\lceil \frac{n+1}{2} \rceil$ primeiros pares <chave, rid> para o nó N e os $\lfloor \frac{n+1}{2} \rfloor$ pares <chave,rid> restantes no novo nó M (10,15) (20);
- uma chave divisora deve ser incluída no nó pai de N para orientar corretamente a pesquisa para os nós N e M. Para isso, uma *cópia* do argumento da primeira entrada do nó M (20) deve ser inserida no nó pai de N e os seus ponteiros atualizados, para refletir as alterações que ocorreram no nível abaixo dele;
- se há lugar no nó pai de N, basta inserir a chave divisora neste na ordem apropriada e atualizar o ponteiros (veja Figura 6.21);

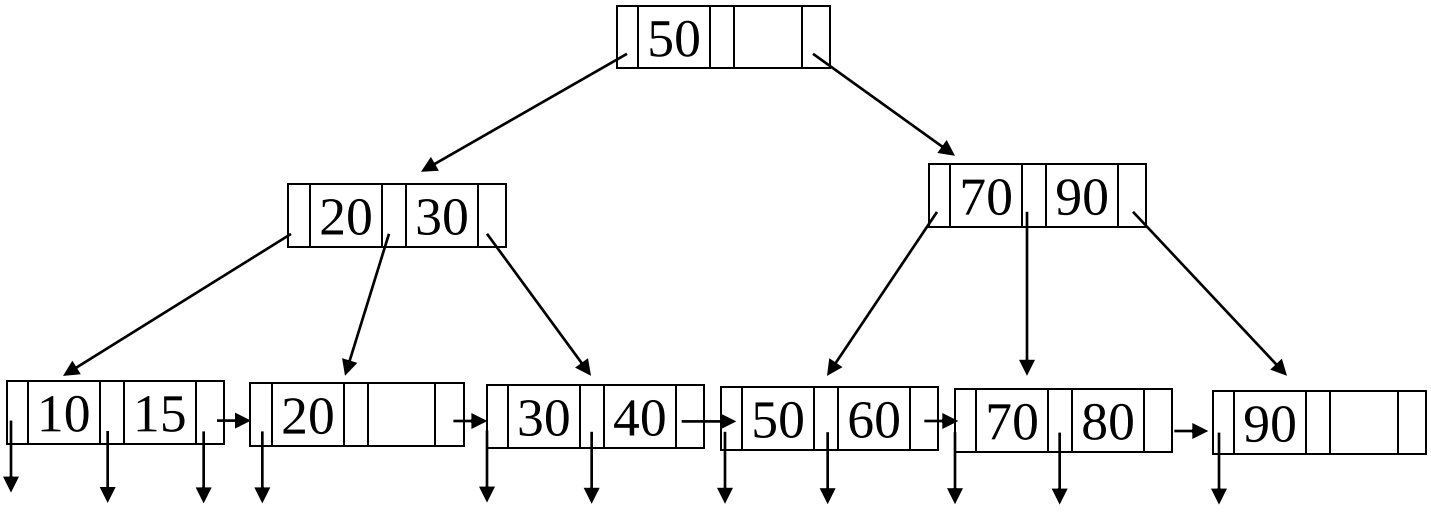


Figura 6.21 – árvore resultante insersão da chave k=15

- se não há lugar no nó pai para uma nova chave divisora, então este nó deve ser dividido.
  Por exemplo, vamos inserir na árvore da Figura 6.21 uma entrada com argumento de pesquisa igual a 35 (K=35). Neste caso o nó folha N onde a entrada deverá ser inserida (3ª. Folha) não tem lugar e portanto precisa ser dividido: (30,35),(40). Uma chave divisora (40), para o novo nó, deverá ser inserida no nó pai de N. No entanto não há lugar. A divisão de um nó não folha é diferente da divisão de um nó folha, veja a seguir:

- seja T o nó não folha a ser dividido, agora contendo n+1 entradas (20,30,40);
- estas n+1 são ordenadas (20,30,40);
- um nó P é criado à direita de T;
- as $\frac{n}{2}\rceil$ primeiras chaves são colocadas em T (20) e as $\frac{n}{2}\rfloor$ últimas chaves restantes são colocadas em P (40). Uma chave vai ficar sobrando (30). Esta chave deve ser *movida* para o nó pai de T. Se houve lugar no nó pai, então a chave divisora é inserida e os ponteiros atualizados, senão o processo de divisão de nó não folha se propaga, podendo chegar até o nó raiz;
- a divisão do nó raiz e´ uma exceção: um novo nó é criado para que a chave divisora seja inserida, uma vez que o nó raiz da árvore foi dividido. Veja o resultado na Figura 6.22.

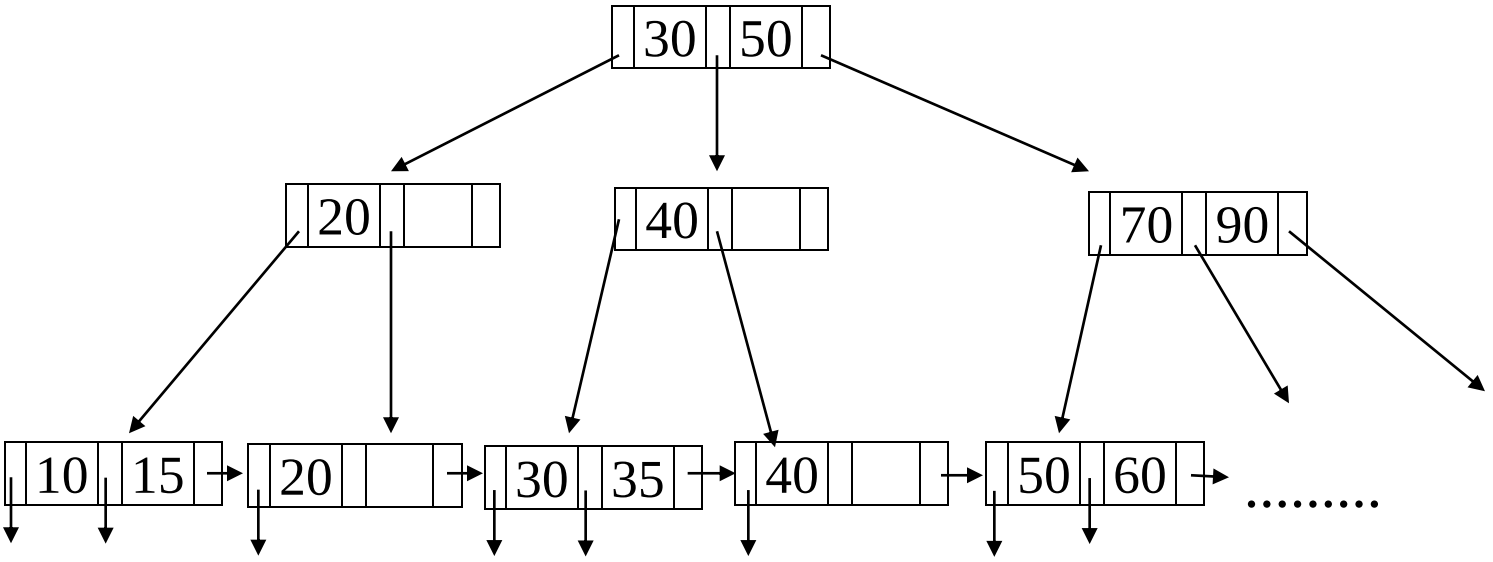


Figura 6.22 – árvore B+ após a inserção da chave k=35

c) *Remoção*

Considere que desejamos remover uma entrada <k,rid> de uma árvore B+;
Uma pesquisa por igualdade é realizada para encontrar o nó folha onde a chave de pesquisa está;
Uma vez encontrado o nó folha N de onde o par <k,rid> será removido, podem ocorrer as seguintes situações:

c1. *a remoção do par <k,rid> não afeta a ocupação mímina do folha nó N*. Neste caso basta remover a entrada de N, ordenar as entradas. Para ilustrar, considere a árvore mostrada na Figura 6.23.

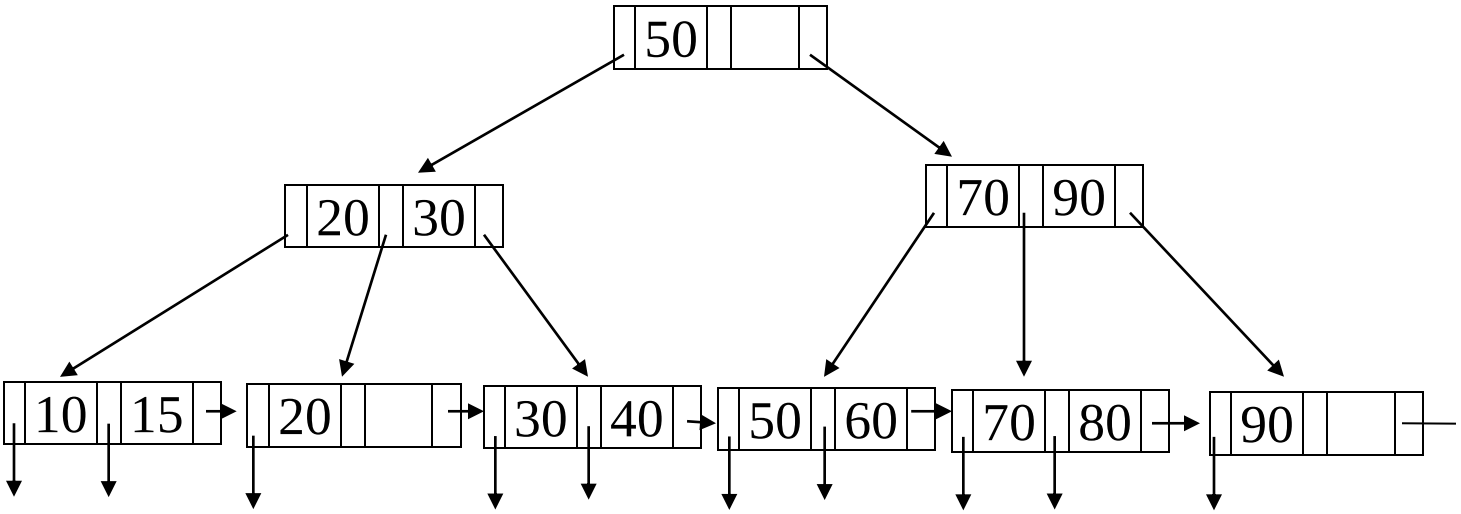


Figura 6.23 – uma árvore B+

A remoção da entrada <60,rid> do quarto nó folha (denotado por nó N) pode ser realizada sem problemas uma vez que o nó manterá sua ocupação mínima garantida.

c2. *a remoção do par <k,rid> afeta a ocupação mímina do nó folha N*

se removermos o par <50,rid> deste mesmo nó N, a sua ocupação ficará abaixo da mínima (⌊(n+1)/2⌋ ponteiros).

Uma de duas estratégias pode ser adotada para resolver este problema:

i. *distribuir as entradas entre N e seu nó vizinho e irmão M:*
neste caso, verificamos se um nó vizinho e irmão M do nó N tem uma entrada para "emprestar" sem comprometer sua ocupação mínima. Se tiver, uma chave de M é movida para N e a chave divisora dos nós N e M deve ser atualizada. No nosso exemplo, o quinto nó folha é vizinho e irmão de N e pode emprestar uma entrada (<70,rid>). A árvore resultante ficará como mostrada na Figura 6.24.

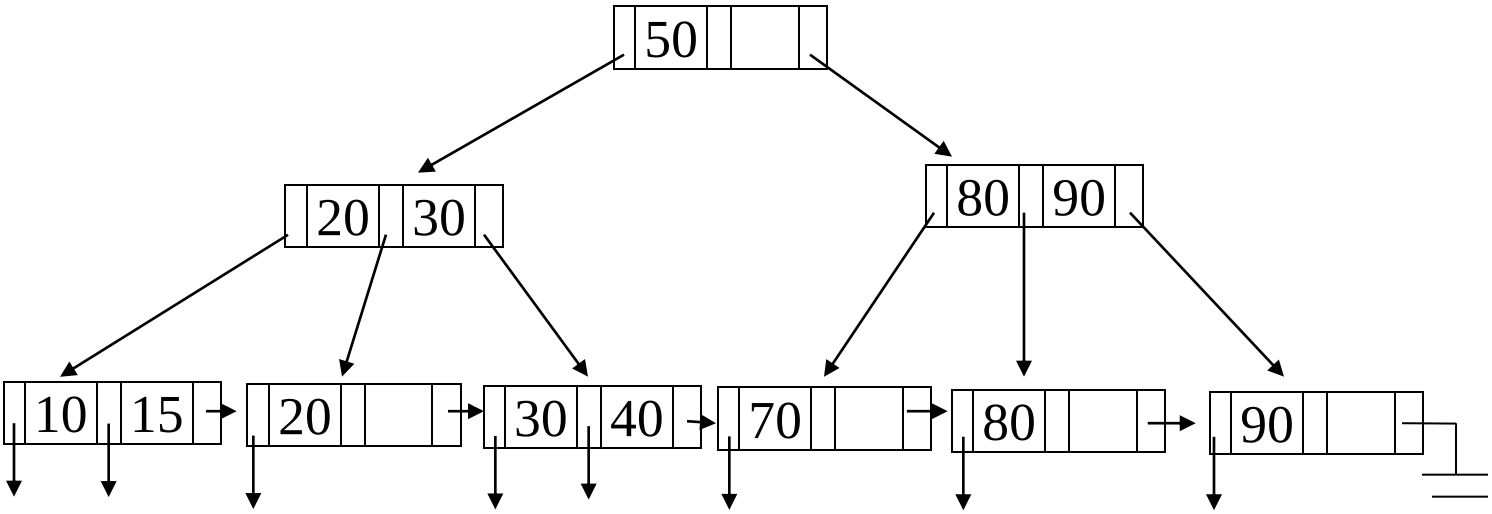


Figura 6.24 – árvore após a remoção da chave k = 50 com distribuição de chaves

ii. *unir as entradas de um nó vizinho e irmão de N:*

esta estratégia é utilizada quando não é possível distribuir as entradas entre N e M. Para ilustrar, vamos remover a entrada <90,rid> do sexto nó não folha, que chamaremos N, da árvore da Figura 6.24. O seu irmão e vizinho M (quinto nó não folha) não pode "emprestar" uma entrada. Neste caso os nós N e M devem ser unidos: todas as entradas restantes de N passarão para M e o nó N poderá ser removido. A chave divisora (90) para os nós N e M precisa ser removida do seu nó pai. Se a remoção desta chave não comprometer a ocupação mínima do nó pai, esta pode ser removida sem problemas e os ponteiros atualizados, veja Figura 6.25.

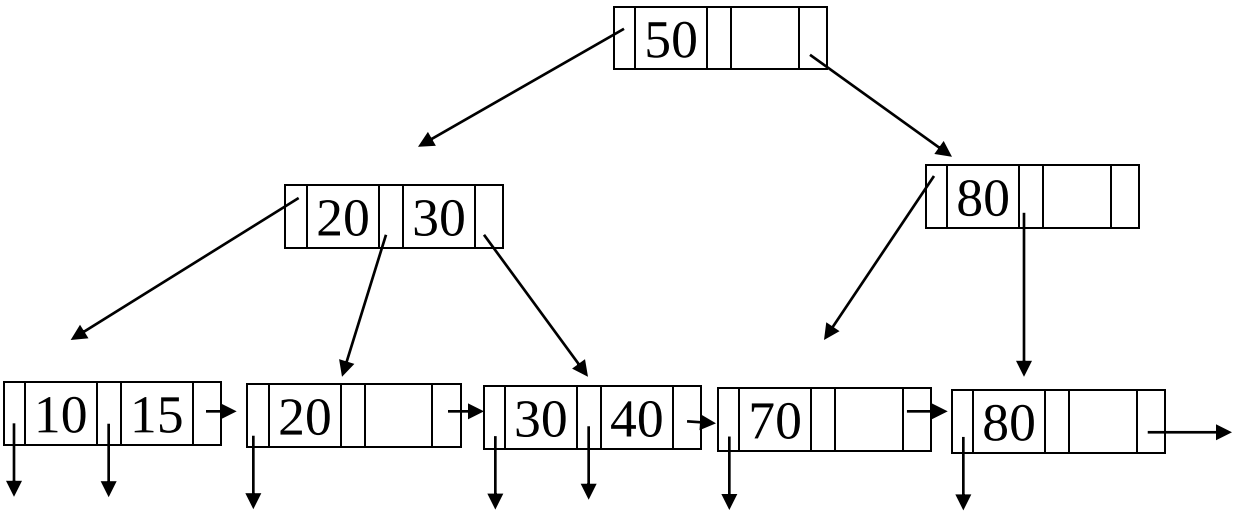


Figura 6.25 – árvore após a remoção da chave k=90 com união de nós

Agora vamos remover o par <80,rid> que está no quinto nó folha (N) da árvore da Figura 6.25. Como esta remoção comprometerá a ocupação mínima de N, o seu vizinho e irmão M (quarto nó folha) devem ser unidos o que resulta em remover a chave divisora 80 do seu pai P. A remoção da chave divisora 80 de P compromete a ocupação mínima do nó. Aqui começa outra etapa da operação de remoção em árvore B+. As estratégias de distribuir e unir nós não folha é um pouco diferente das estratégias para nós folhas:

i. *distribuir as entradas entre P e seu nó vizinho e irmão Q*

Da mesma forma que para nós folha, um nó vizinho e irmão de P deve ter uma chave de pesquisa para emprestar. No nosso exemplo o nó Q (vizinho de P) pode "emprestar". Neste caso a chave de pesquisa de Q a ser emprestada é movida para o lugar da chave divisora dos nós P e Q, e esta chave divisora é movida para P e os ponteiros são atualizados. A remoção da entrada <80,rid> resultará na árvore, veja Figura 6.26.

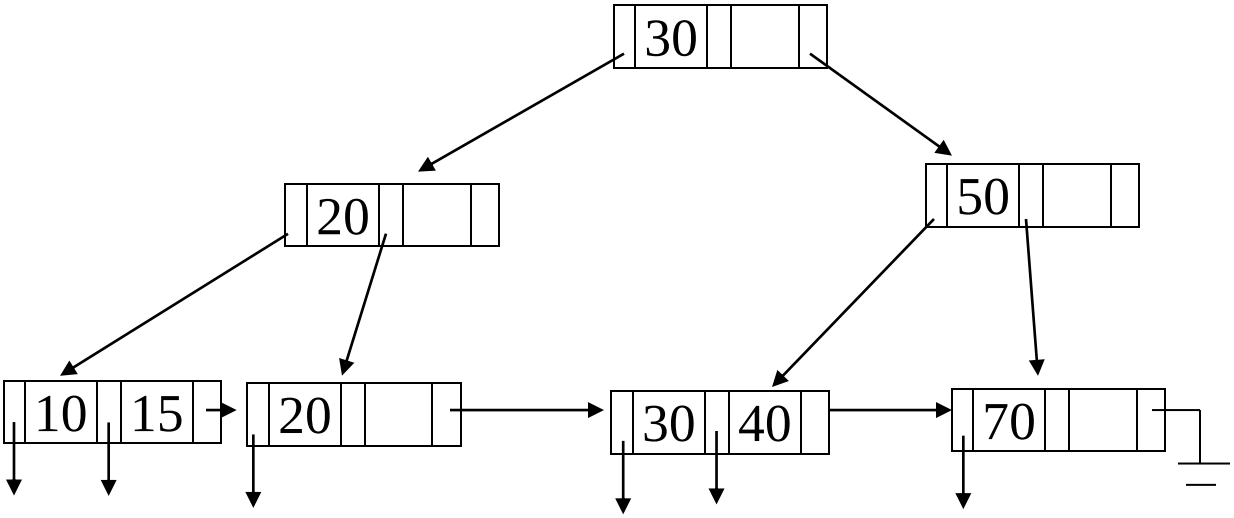


Figura 6.26 – remoção da chave k=80 com redistribuição de entradas

ii. *unir nós não folha:*

Vamos remover o par <15,rid> da árvore da Figura 6.26 e em seguida remover o par <20,rid>. Na segunda remoção o segundo nó folha ficará com sua ocupação abaixo da mínima, neste caso o primeiro e segundo nós folhas serão unidos e a chave divisora (20) de seu pai P deverá ser removida. A remoção da chave 20 comprometerá a ocupação de P. Seu vizinho Q não pode "emprestar". Neste caso P e Q deverão ser unidos. A união de dois nós não folha se dá da seguinte forma:

- une todas as chaves de P e Q em um único nó deixando uma posição vazia entre eles;
- move para a posição vazia a chave divisora de P e Q

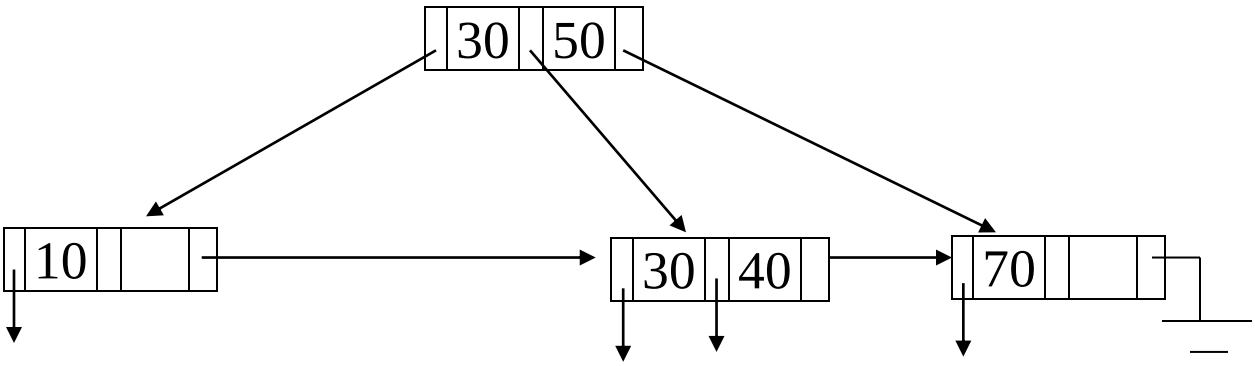


Figura 6.27 – remoção da chave k=15 seguida da chave k=20 com união de nós

6.3.2.3. Eficiência de uma B+

As operações de consulta, inserção e remoção em árvores B+ requerem poucos acessos a disco. Qualquer nó folha é alcançado com no máximo:
$\text{Log}_F N$, onde N é o número de registros do arquivo de dados e $F=\lceil (n+1)/2 \rceil$ é o número mínimo de ponteiros por nó não folha.

6.3.2.4. Árvore B+ com argumentos de pesquisa duplicados

Os algoritmos de pesquisa, inserção e remoção apresentados ignoram a duplicação de chaves de pesquisa. Algumas possibilidades para gerenciar esta questão são tratadas a seguir:

1. os algoritmos propostos assumem que todas as entradas de um dado valor de pesquisa reside em uma única folha. Uma maneira de satisfazer esta asserção é usar páginas de *overflow* para trabalhar com duplicatas;

2. as entradas em duplicata podem ser manipuladas como qualquer outra entrada e diversas páginas folha podem conter entradas com um mesmo valor de chave. Neste caso é necessária uma leve alteração da definição do que significa um nó não folha. Suponha que existam as chaves k1,k2,...,kn em um nó não folha. Então a k_i entrada deverá ser a menor nova chave que aparece na parte da subárvore acessível a partir do (i+1)-ésimo ponteiro. Por nova chave entendemos que não há ocorrência de k_i na subárvore à esquerda da i-ésima entrada mas há pelo menos uma ocorrência de k_i na subárvore à direita da i-ésima entrada. Note que podem ocorrer situações em que não exista uma nova chave para uma entrada i. Neste caso k_i será inválido. Os ponteiros associados à entrada i ainda serão necessários uma vez que ele aponta para uma parte da árvore que contém apenas um valor de chave nela. A Figura 6.28 mostra uma árvore que permite a ocorrência de valores duplicados.

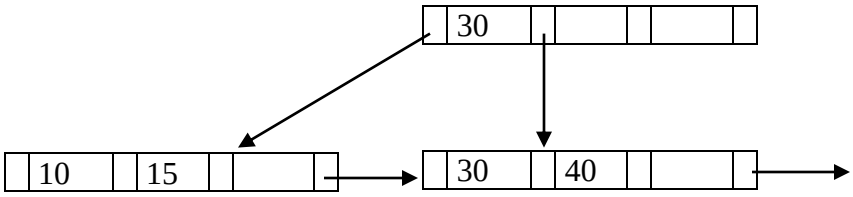


Figura 6.28 – uma árvore B+ que suporta valores de chaves duplicados.

A inserção da chave k=30, seguida da inserção da chave k=30 causa a seguinte modificação na árvore, veja Figura 6.29. Note que a chave k=40 é a primeira nova entrada no 3º nó por isso aparece como entrada no nó raiz.

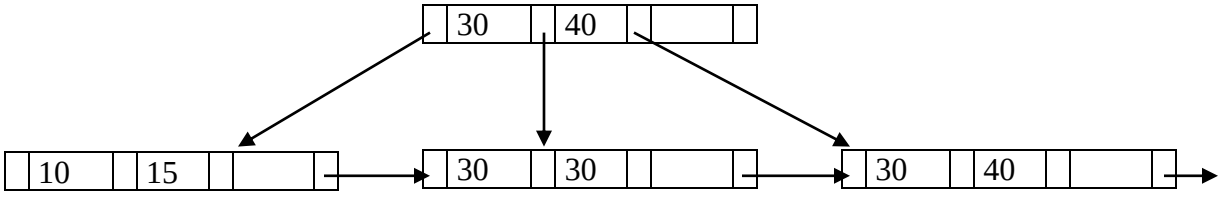


Figura 6.29 – árvore resultante após a inserção das chaves k=30 seguida da chave k=30.

Após inserir duas chaves de valor igual a 30, o resultado pode ser visto na Figura 6.30. Note que a segunda entrada do nó raiz não tem valor válido porque a sua subárvore da direita não possui nenhuma entrada nova. O nó apontado pelo ponteiro à direita da entrada com valor 40 não está representado na Figura 6.30.

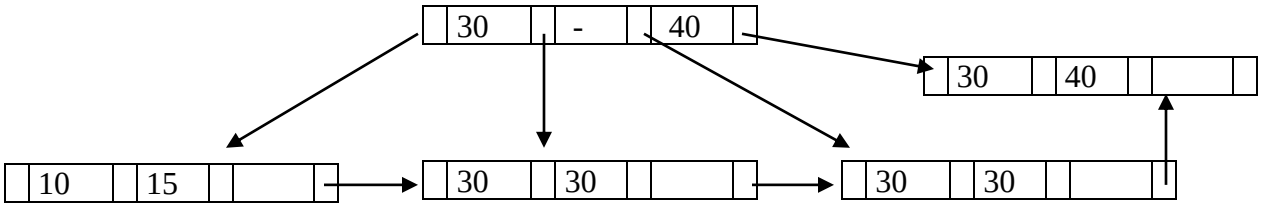


Figura 6.30 – árvore resultante após a inserção de duas chaves k=30

A inserção da chave k=35 causa a modificação mostrada na Figura 6.31. Note que o valor 35 é uma chave nova e passa a ser a chave divisora da segunda entrada do nó raiz.

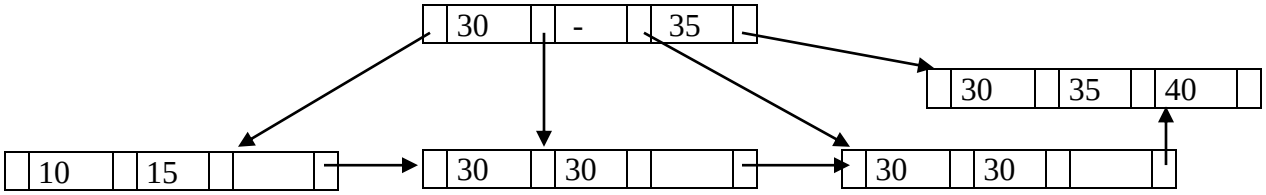


Figura 6.31 – árvore resultante após a inserção da chave k=35

3. Um problema que surge nas duas técnicas previamente apresentadas é que quando um registro é removido, encontrar a entrada correta na árvore a ser removida pode ser ineficiente porque temos que verificar diversos pares (chave, rids). Este problema pode ser resolvido considerando o rid de um registro como parte da chave de pesquisa para o propósito de posicionamento das entradas de dados na árvore . Esta solução transforma o índice em um índice único.

6.3.2 O conceito de ordem

A árvore B+ apresenta um mínimo de ocupação dada em termos do número de entradas do nó. No entanto na prática esta maneira de calcular a ocupação mínima de um nó é relaxada e substituída por um critério baseado em espaço, por exemplo o nó deve ser mantido com no mínimo a metade do espaço de uma página.

Uma razão para isto é que os nós folha e os nós não folha podem ter diferentes números de entradas. Lembre-se que os nós não folha mantém apenas as chaves e os rids enquanto os nós folha podem manter o próprio registro de dados e o tamanho do último será maior que o primeiro, o que levamos a uma situação em que cabem mais entradas nos nós não folha do que nos nós folha.

Outra motivação para o relaxamento da ocupação mínima é que a chave de pesquisa pode ser formada por string de caracteres, cujo tamanho varia de registro para registro. Neste caso o número de entradas nos nós não podem variar, de acordo com as entradas de cada nó.

6 3.3 índice baseado em tabelas hash

As tabelas hash dispõem de uma organização de arquivos excelente para seleção por igualdade. A idéia básica é usar uma função hashing que mapeia valores de uma chave de pesquisa para um número de grupo (bucket) para encontrar a página onde está o registro desejado. No capítulo 5 introduzimos três esquemas de tabelas hash: estática, dinâmica extensível e linear.

Técnicas de indexação baseadas em tabelas hash não suportam pesquisas por intervalo de forma eficiente. A maioria dos SGBD comerciais usa a indexação baseada em B+ embora

tabelas hash sejam muito úteis para implementar o operador de junção uma vez que este gera muitas consultas por igualdade.