Indexação e Hashing

- Conceitos básicos
- Índices Ordenados
- Arquivos de Índice de árvore B+
- Arquivos de Índice de árvore B
- Hashing Estático
- Hashing Dinâmico
- Comparação de Indexação Ordenada e Hashing
- Definição de Índices em SQL
- Acessos Multi-Chave

Arquivos de índice de árvore B^+

Índices de árvore B^+ são uma alternativa para arquivos seqüências indexados

- Desvantagem dos arquivos seqüenciais indexados: o desempenho degrada na medida que o arquivo cresce, já que vários blocos de overflow são criados. Reorganização periódica de todo o arquivo é necessária.
- Vantagem dos arquivos de árvore B^+: Se reorganizam automaticamente com mudanças pequenas e locais, devido a inserções e remoções. A reorganização de todo o arquivo não é necessário para manter o desempenho.
- Desvantagem das árvores B^+: “overhead” extra de inserção e remoção, “overhead” de espaço.
- Vantagens das árvores B^+ são maiores que as desvantagens, e eles são usados extensivamente.

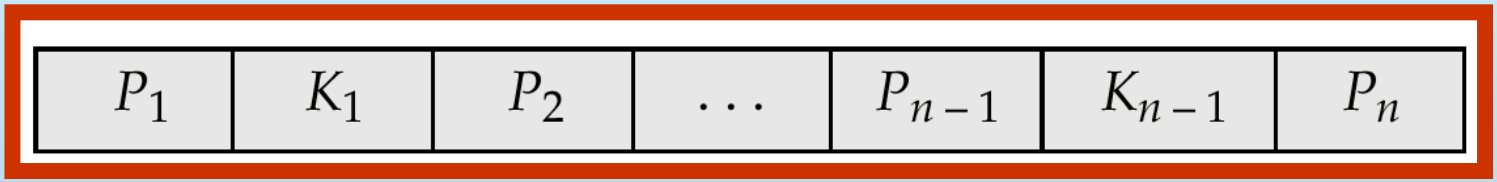
Arquivos de índice de árvore B^+ (Cont.)

Uma árvore B^+ é balanceada e satisfaz as seguintes propriedades:

- Todos os caminhos da raiz até as folhas são do mesmo tamanho
- Cada nodo que não é uma raiz ou uma folha tem entre $[n/2]$ e n filhos.
- Um nodo folha tem entre $[(n–1)/2]$ e $n–1$ valores
- Casos especiais:
  - Se a raiz não é uma folha, ela tem no mínimo 2 filhos.
  - Se a raiz é uma folha (isto é, não existem outros nodos na árvore), ela pode ter entre 0 e $(n–1)$ valores.

Estrutura de um nó da árvore B+

- Nó típico



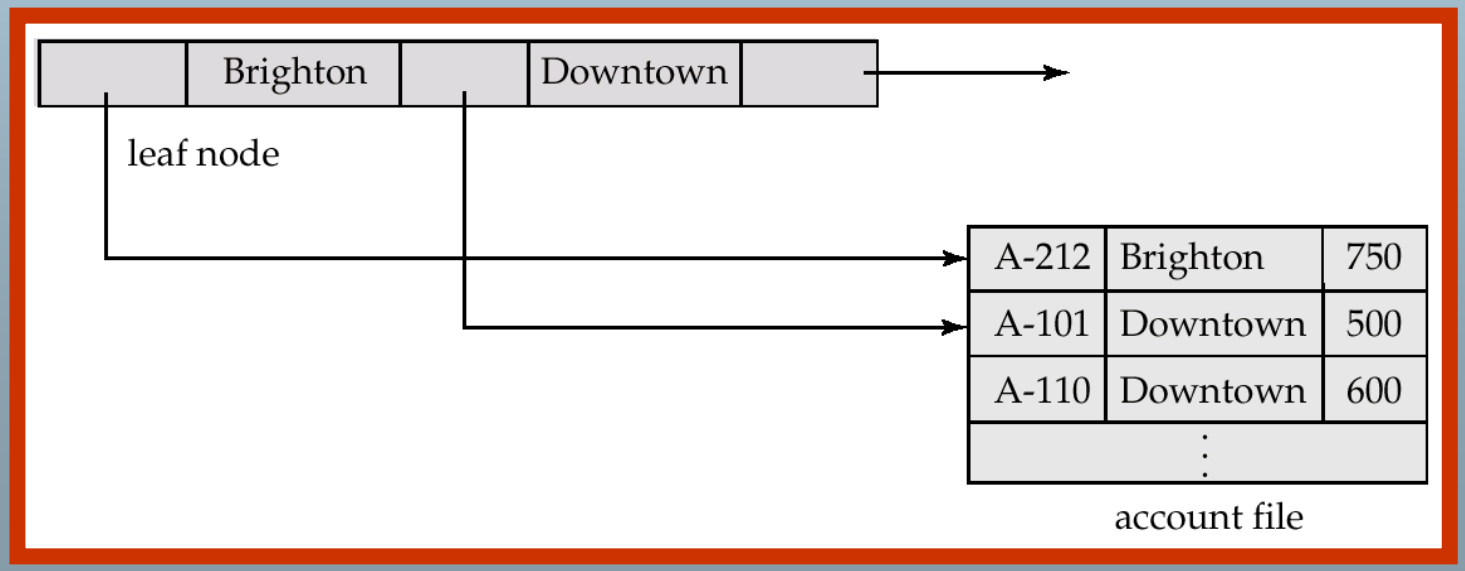
  - K_i são os valores das chaves de busca
  - P_i são ponteiros a filhos (para nós não folhas) ou ponteiros a registros ou grupos de registros (para nodos folhas).
- As chaves de busca em um nó são ordenadas

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Nós Folhas nas árvores B+

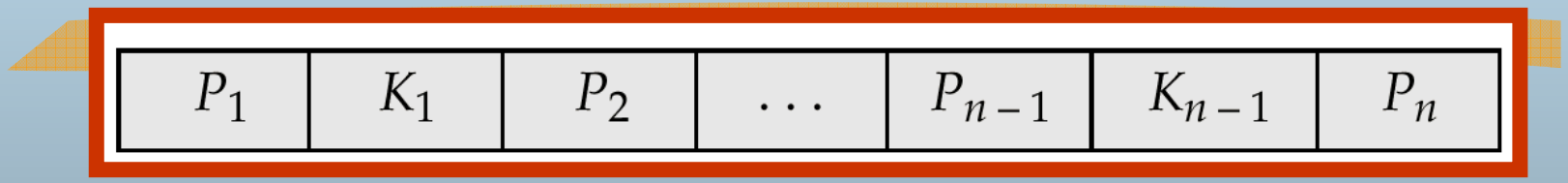
Propriedades de um nó folha:

- Para $i = 1, 2, \ldots, n-1$, ponteiro P_i ou aponta para um registro do arquivo com valor da chave de busca K_i, ou a um bucket de ponteiros para registros de arquivo, cada registro tendo como valor da chave K_i. Somente é necessário uma estrutura de bucket se a chave de busca não forma uma chave primária.
- Se L_i, L_j são nós folhas e $i < j$, *os valores das chaves de busca de* L_i são menores que os de L_j
- P_n aponta para o próximo nó folha na ordem da chave de busca

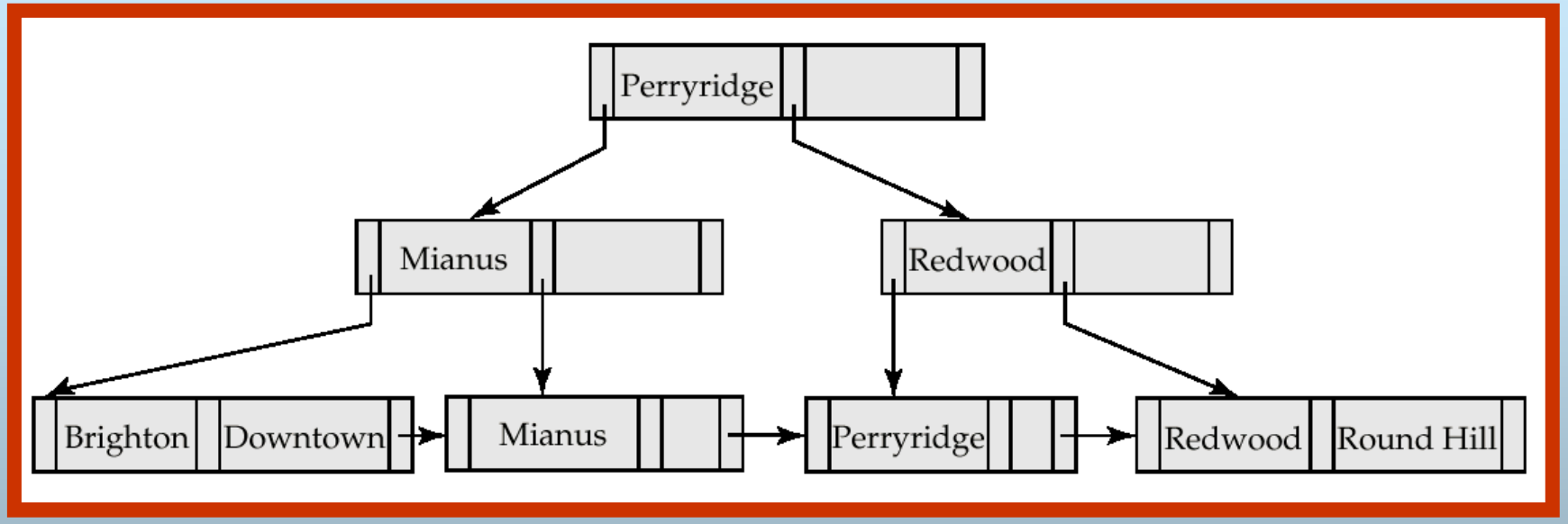


Nós não-folhas nas árvores B+

- Nós não-folhas formam um índice esparso de multi-nível sobre os nós folhas. Para um nó não-folha com *n* ponteiros:
  - Todas as chaves de busca na sub-árvore para a qual P_1 aponta são menores que K_1
  - Para $2 \leq i \leq n - 1$, todas as chaves de busca na sub-árvore para a qual P_i aponta têm valores maiores que ou iguais a K_{i-1} e menores que K_i

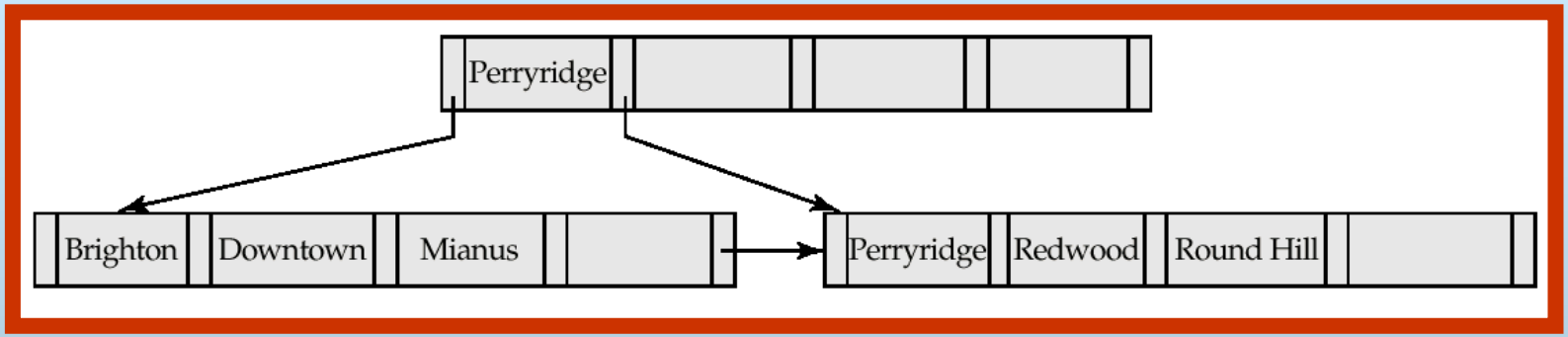


Exemplo de uma árvore B+



Árvore B+ para o arquivo de contas ($n = 3$)

Exemplo de uma árvore B^+



Árvore B^+ para o arquivo de contas ($n = 5$)

- Nós folhas devem ter entre 2 e 4 valores ($\lceil (n–1)/2 \rceil$ e $n – 1$, com $n = 5$).
- Nós não-folhas diferentes da raiz devem ter entre 3 e 5 filhos ($\lceil (n/2 \rceil$ e n com $n = 5$).
- Raiz deve ter no mínimo 2 filhos.

Observações sobre árvores B^+

- Já que as conexões entre nós são feitas por ponteiros, blocos fechados “logicamente” não precisam ser “fisicamente” fechados.
- Os níveis dos nós não-folhas da árvore B^+ formam uma hierarquia de índices dispersos.
- A árvore B^+ contem um número relativamente pequeno de níveis (logarítmico no tamanho do arquivo principal), assim buscas podem ser feitas eficientemente.
- Inserções e remoções ao arquivo principal podem ser feitas eficientemente, já que o índice pode ser reestruturado em tempo logarítmico (como será visto depois).

Consultas em árvores B+

- Achar todos os registros com um valor de chave de busca *k.*
  1. Iniciar com o nó raiz
     1. Examinar o nó pela menor chave de busca cujo valor > *k.*
     2. Se tal valor existe, seja este K_j. Então siga por P_i até o nó filho
     3. Senão, $k \geq K_{m-1}$, onde existem *m* ponteiros no nó. Então, siga P_m até o nó filho.
  2. Se o nó alcançado pelo ponteiro acima não é um nó folha, repita o procedimento acima, e siga o ponteiro correspondente.
  3. Se eventualmente alcança um nó folha. Se para algum *i*, a chave $K_i = k$ siga o ponteiro P_i *ao registro desejado ou* bucket. Senão nenhum registro com o valor da chave *k* existe.

Consultas em árvores B+ (Cont.)

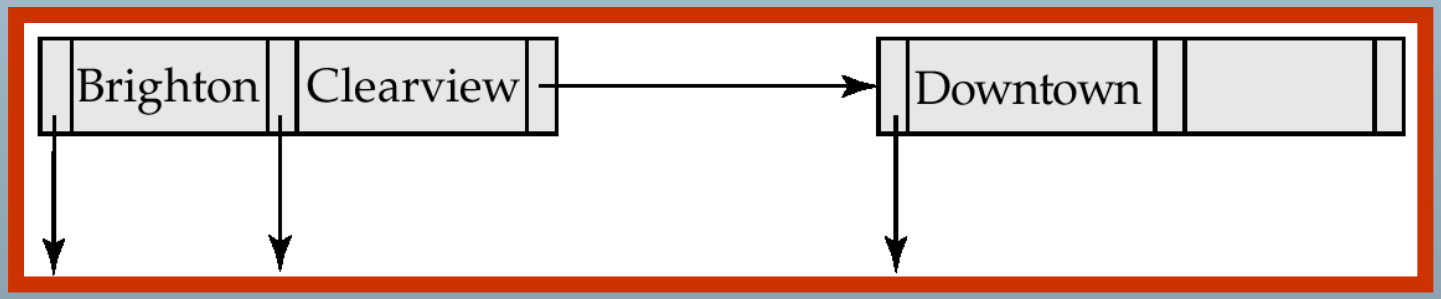
- No processamento de uma consulta, um caminho é percorrido na árvore desde a raiz até algum nó folha.
- Se existem *K* valores de chaves de busca no arquivo, o caminho não é maior de $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Um nó é geralmente do mesmo tamanho que um bloco de disco, tipicamente 4 kilobytes, e n *é* tipicamente aoredor de 100 (40 bytes por entrada de índice).
- Com 1 milhão de valores de chaves e $n = 100$, no máximo $log_{50}(1{,}000{,}000) = 4$ nós são acessados numa busca.
- Ao contrário, com uma árvore binária balanceada com1 milhão de valores de chaves — aprox. 20 nós são acessados numa busca
  - a diferença acima é significativa já que todo acesso a um nó pode precisar um acesso a disco, custando aoredor de 20 millisegundos!

Atualização em árvores B^+: Inserção

- Ache o nó folha no qual o valor da chave de busca pode aparecer
- Se o valor da chave de busca está já no nó folha, o registro é adicionado ao arquivo e se for necessário um ponteiro é inserido no bucket.
- Se o valor da chave de busca não está lá, então adicione o registro ao arquivo principal e crie um bucket se for necessário. Então:
  - Se existir espaço no nó folha, insira par (valor-chave, ponteiro) no nó folha
  - Senão, divida o nó (com a nova entrada (valor-chave, ponteiro)) da maneira descrita a seguir.

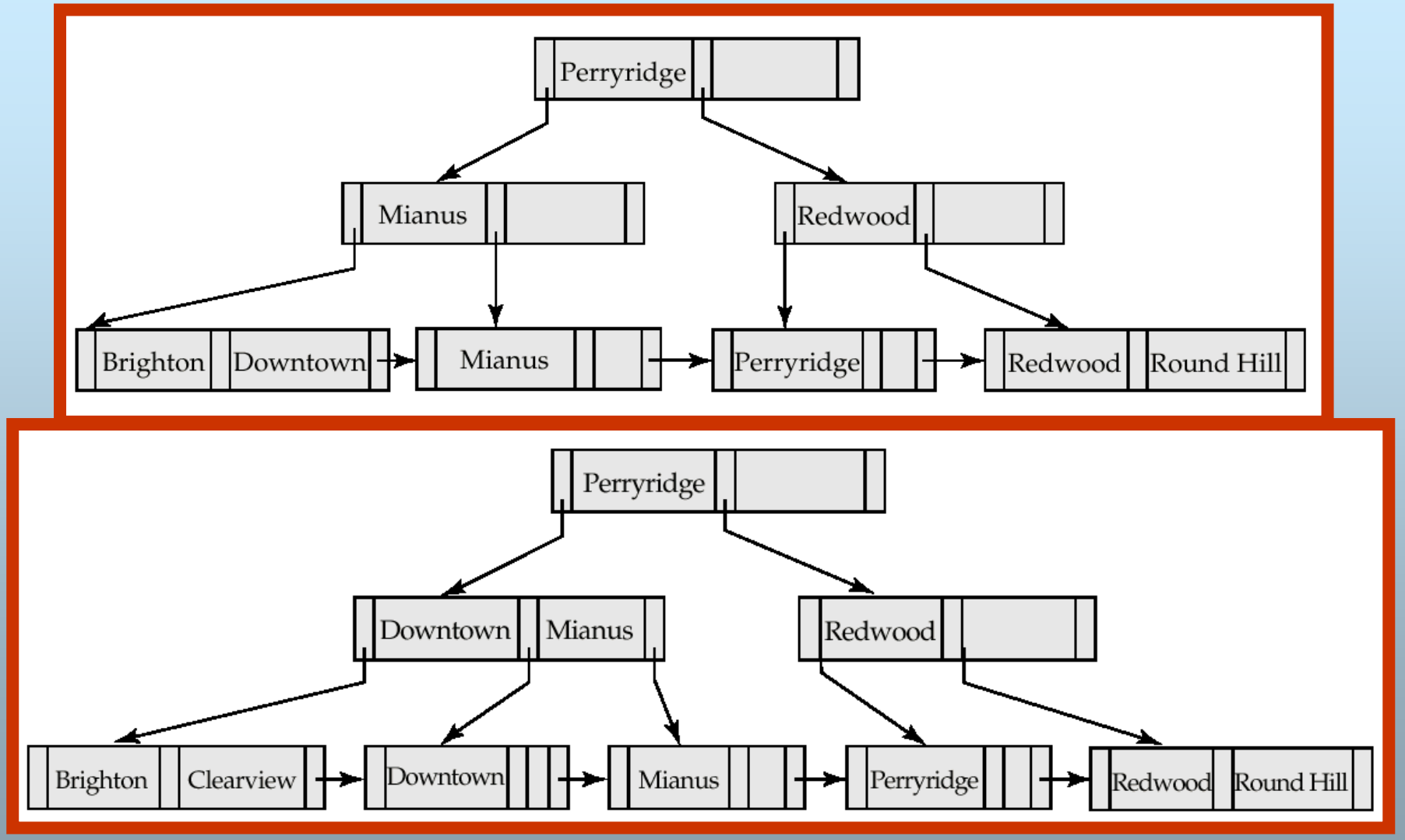
Atualização em árvores B+: Inserção (Cont.)

- Dividindo um nó:
  - Pegue os *n pares* (valor da chave busca, ponteiro) (incluindo o que está sendo inserido) na ordem. Coloque os primeiros $\lceil n/2 \rceil$ no nó original, e o resto num novo nó.
  - Seja o novo nó *p,* e seja *k* o mínimo valor em *p.* Insira (*k,p*) no nó pai do nó sendo dividido. Se o pai está cheiol, divida este e propague a divisão para cima.
- A divisão de nós é propagada até um nó que não esteja cheio seja achado. No pior caso o nó raiz pode ser dividido incrementando a altura da árvore em 1.



Resultado de dividir o nó contendo Brighton e Downtown devido à inserção de Clearview

Atualização em árvores B+: Inserção (Cont.)



Árvore B+ antes e depois da inserção de “Clearview”

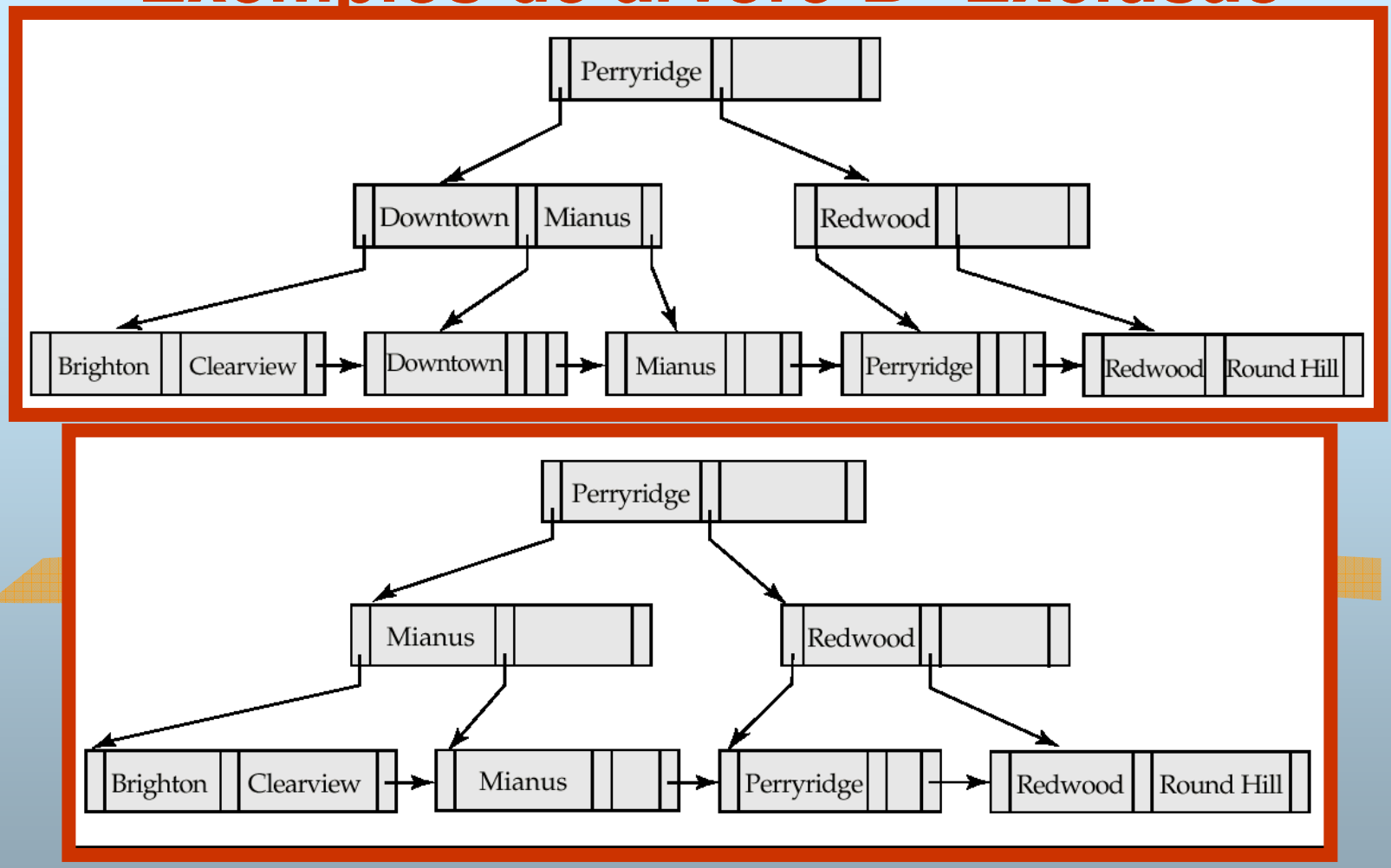
Atualização em árvores B+: Remoção

- Achar o registro a ser removido, e remova este do arquivo principal e o bucket (se for necessário)
- Remove o par (valor chave busca, ponteiro) do nó folha se não existir bucket ou se o bucket fica vazio
- Se o nó tem muito poucas entradas devido à remoção, e as entradas do nó e seu irmão se ajustam num só nó, então
  - Insira todos os valores das chaves dos dois nós num só nó (aquele à esquerda), e remova o outro nó.
  - Remova o par (K_{i-1}, P_i), onde P_i é o ponteiro ao nó removido, do seu pai, recursivamente usando o procedimento acima.

Atualização em árvores B+: Remoção

- De outro modo, se o nó tem muito poucas entradas devido à remoção, e as entradas no nó e um irmão não se ajustam em um só nó, então
  - Redistribua os apontadores entre o nó e o irmão tal que os dois tenham mais do que o mínimo número de entradas.
  - Atualize o valor da chave de busca correspondente no pai do nó.
- As remoções de nós podem ser em cascata até alcançar um nó que tenha $\lceil n/2 \rceil$ ou mais ponteiros. Se a raiz tem somente um ponteiro após a remoção, esta é removida e o filho único vira raiz.

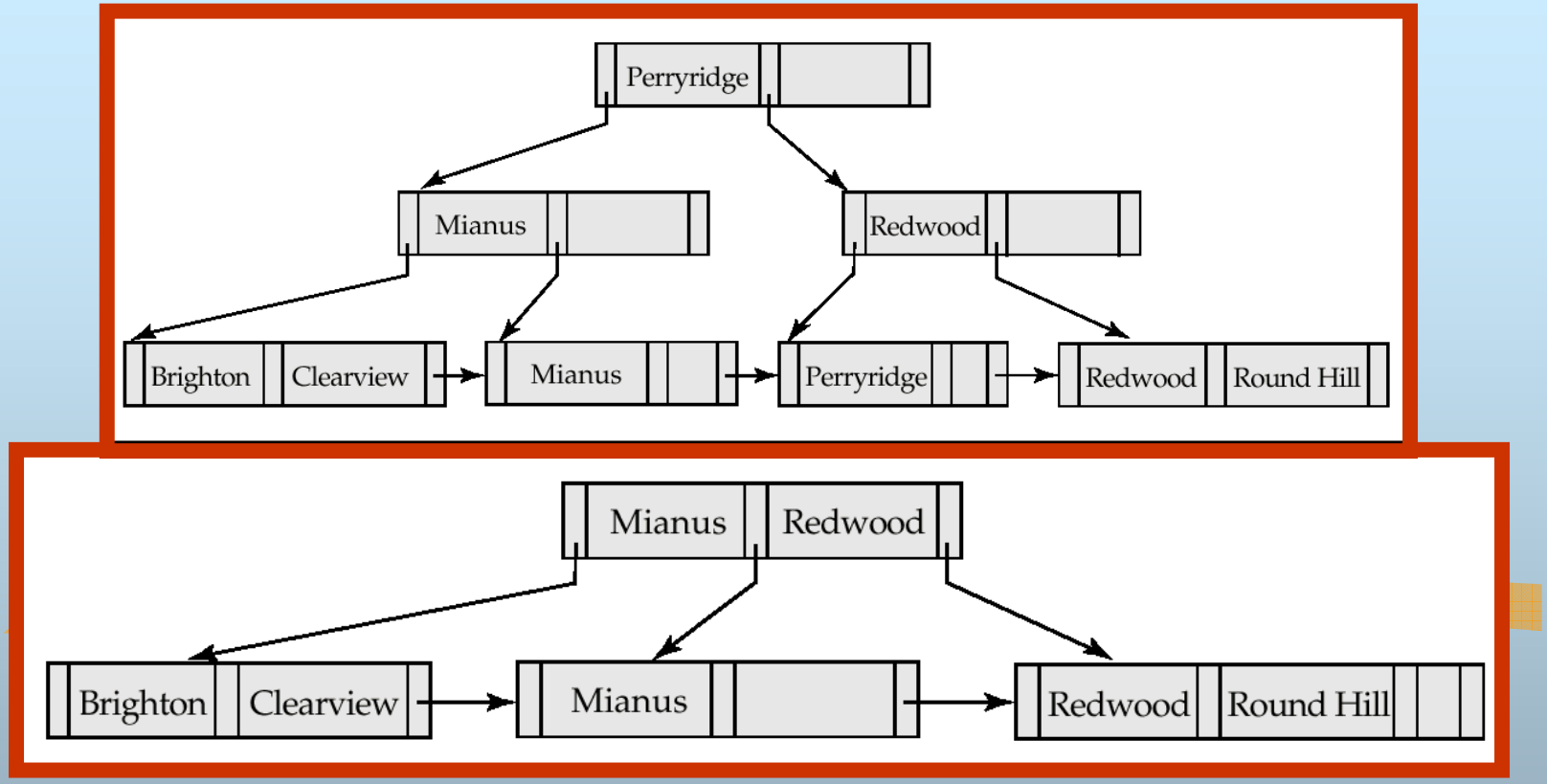
Exemplos de árvore B+ Exclusão



Antes e depois da remoção de “Downtown”

- A remoção do nó folha contendo “Downtown” não gerou poucos ponteiros no nó pai. Assim as remoções em cascata param com a folha removida e o ponteiro do nó pai.

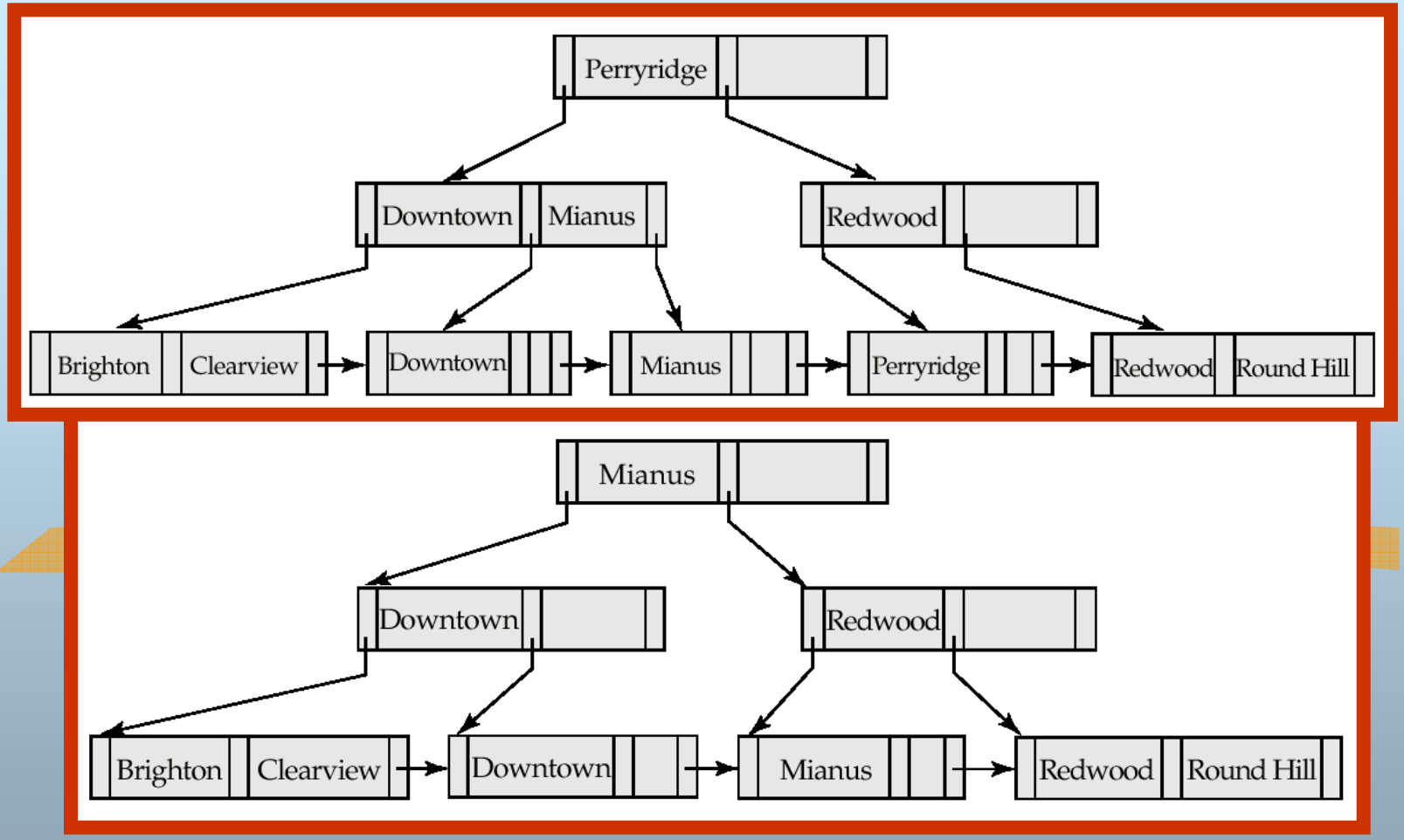
Exemplos de árvore B+ Exclusão (Cont.)



Remoção de “Perryridge” do resultado do exemplo anterior

- Nó com “Perryridge” vira vazio e é eliminado (misturado com seu irmão).
- Como um resultado o nó pai de “Perryridge” vira quase vaziol, e é misturado com seu irmão (e uma entrada foi removida de seu pai)
- O nó raiz então tinha um único filho, e foi removido e seu filho vira um novo nó raiz

Exemplo de árvore B+ Exclusão (Cont.)



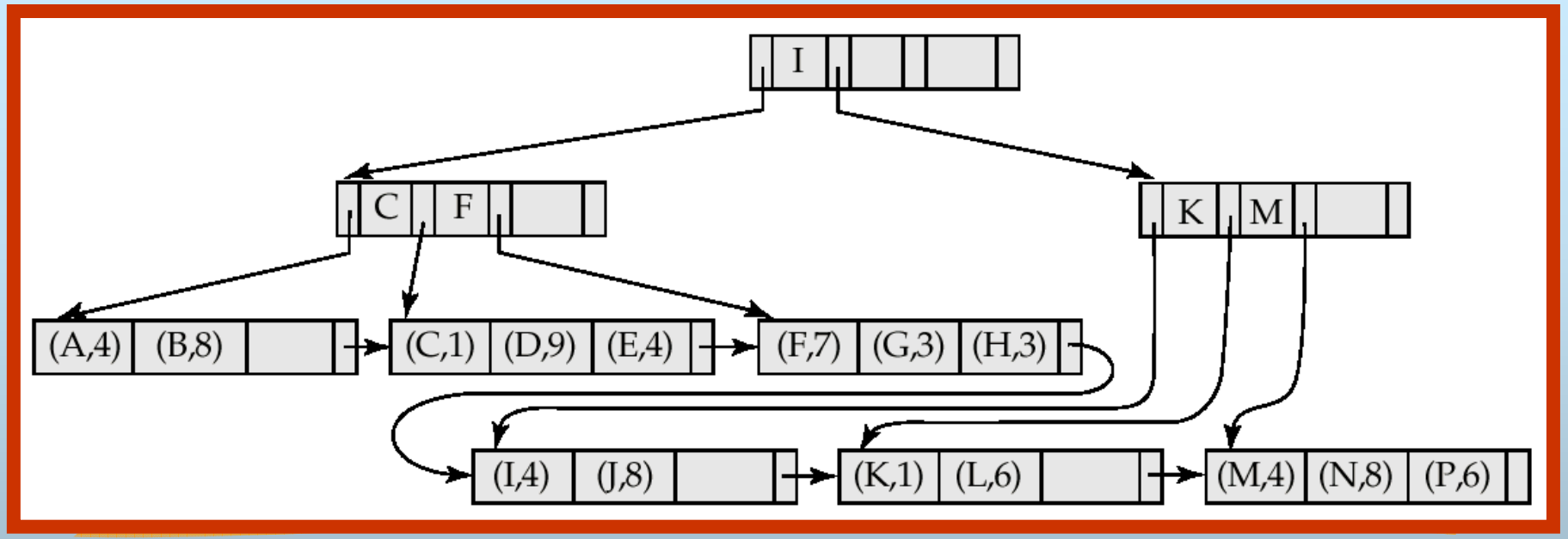
Antes e depois da remoção de “Perryridge” do exemplo inicial

- Pai da folha que contem Perryridge vira quase vazio, e pede emprestado um ponteiro a seu irmão esquerdo
- Valor da chave de busca no pai do pai muda como um resultado

Organização de arquivos de árvore B^+

- O problema de degradação dos arquivos de índices é resolvido usando índices de árvore B^+. O problema de degradação dos arquivos de dados é resolvido usando uma organização de arquivos de árvores B^+.
- Os nós folhas numa organização de arquivos de árvore B^+ armazena registro, no lugar de ponteiros.
- Já que os registros são maiores que os ponteiros, o máximo número de registros que podem ser armazenados em um nó folha é menor que o número de ponteiros em um nó não folha.
- Os nós folhas ainda precisam ter a metade cheia.
- A Inserção e remoção são tratadas na mesma forma que a inserção e remoção de entradas num índice de árvore B^+.

Organização de arquivos de árvore B+ (Cont.)

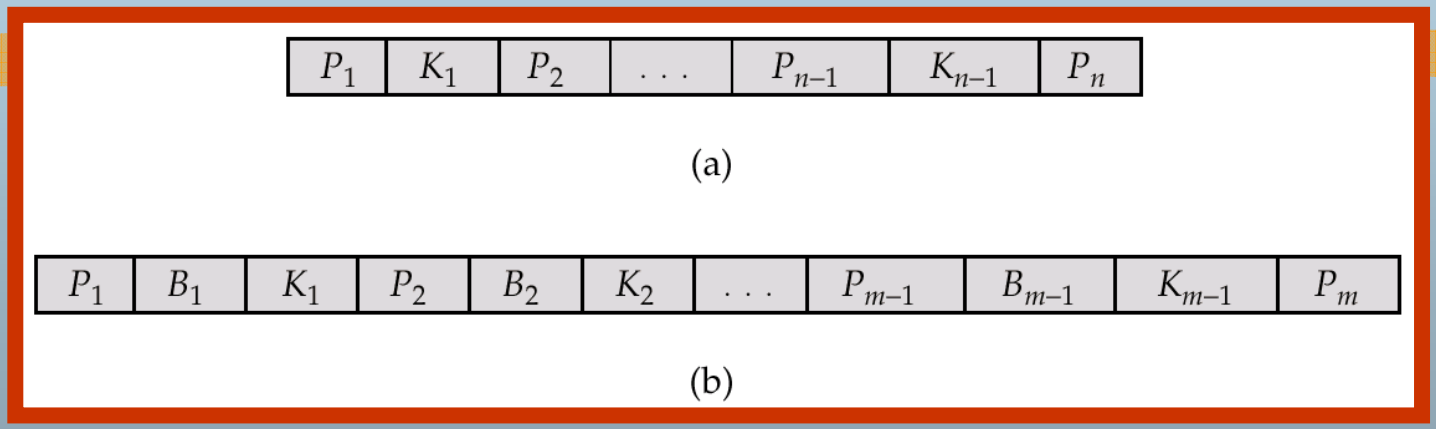


Exemplo de Organização de arquivo de árvore B+

- Boa utilizaçãodo espaço é importante já que os registro usam mais espaço que os ponteiros.
- Para melhorar a utilização do espaço, tem que ser envolvidos mais nós irmão na redistribuição durante as divisões e junções

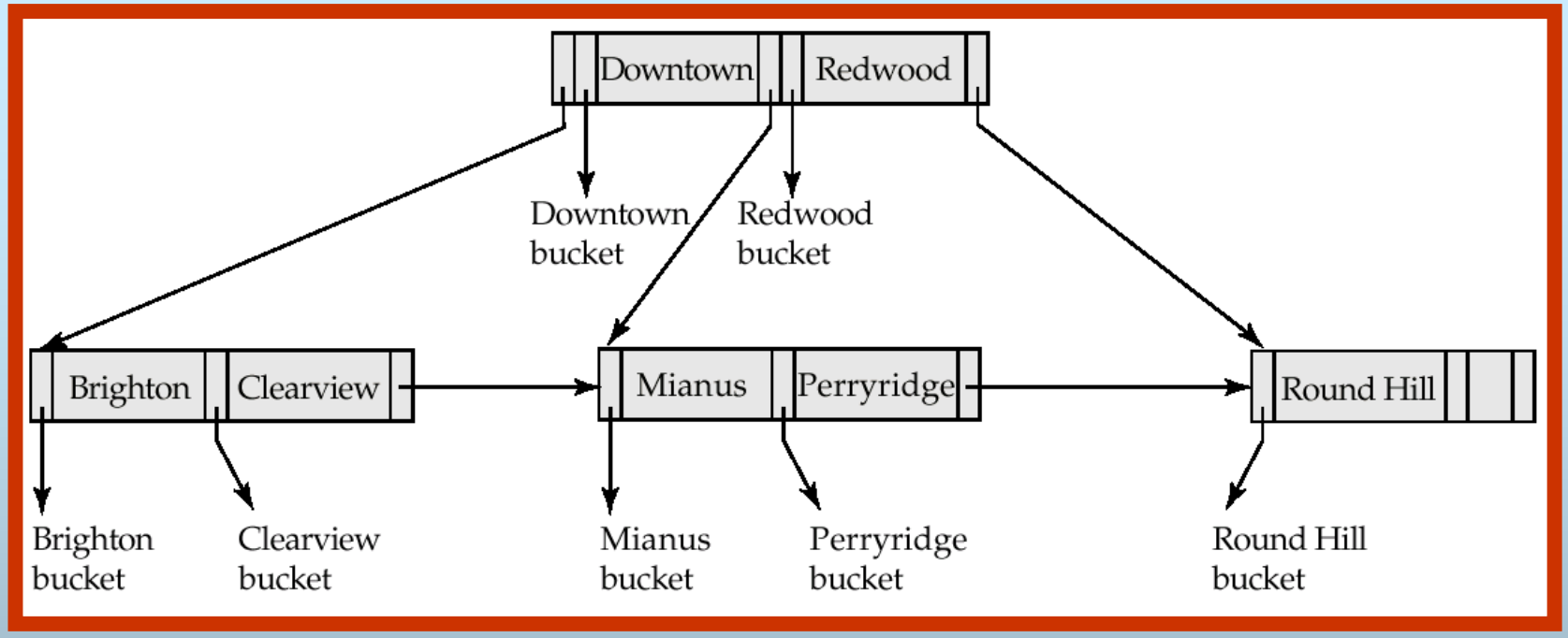
Arquivos de índice de árvore B

- Similar a árvore B+, porém árvore B permite que os valores das chaves de busca apareçam uma única vez; elimina armazenamento redundante das chaves de busca.
- As chaves de busca nos nós não folhas aparecem uma só vez na árvore B; um campo de ponteiro adicional para cada chave de busca em um nó não folha deve ser incluído.
- Nós típico de uma árvore B

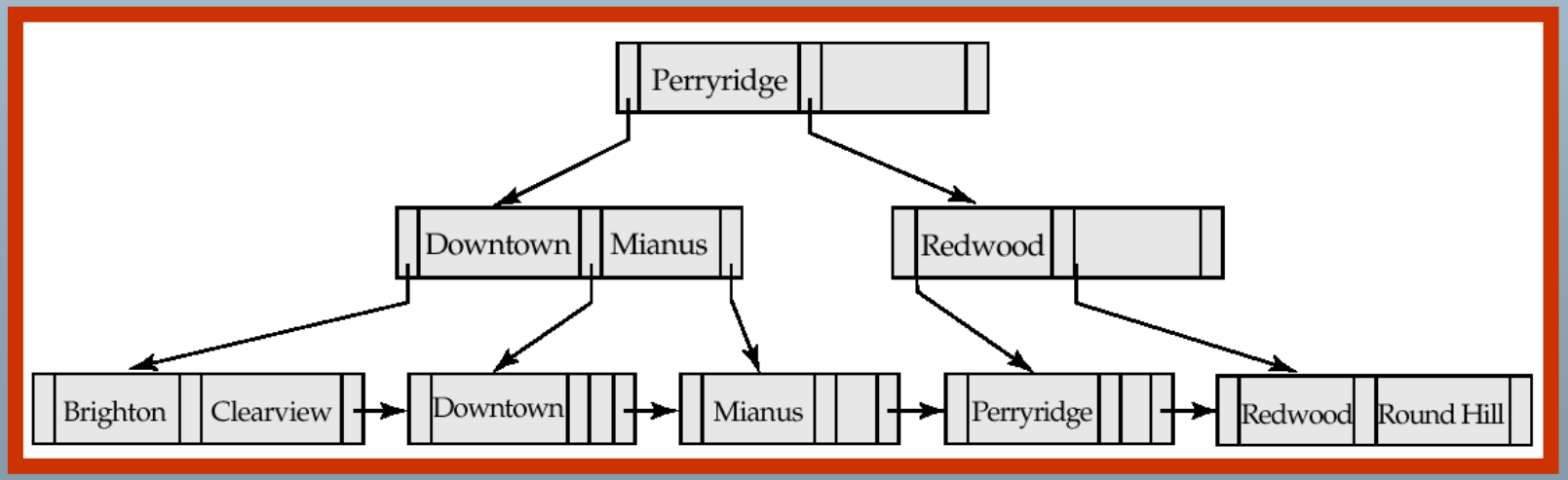


- Nós não folhas – ponteiros B_i são aos buckets ou arquivo de registros.

B-Tree Index File Example



Árvore B (acima) e árvore B+ (embaixo) com os mesmos dados



Hashing Estático

- Um **bucket** é uma unidade de armazenamento que contem um ou mais registros (um bucket é tipicamente um bloco de disco).
- Numa **organização de arquivo hash** nos obtemos o bucket de um registro diretamente do valor de sua chave de busca usando uma **função de hash.**
- Uma função de Hash é uma função cujo domínio é o conjunto de todos os valores da chave de busca *K* e o resultado é o conjunto de todos endereços de bucket *B.*
- Uma função de Hash é usada para localizar registros para acesso, inserção o mesmo que remoção.
- Registros com diferentes valores da chave de busca podem ser mapeados ao mesmo bucket; assim todo o bucket tem que ser pesquisado seqüencialmente para localizar um registro.

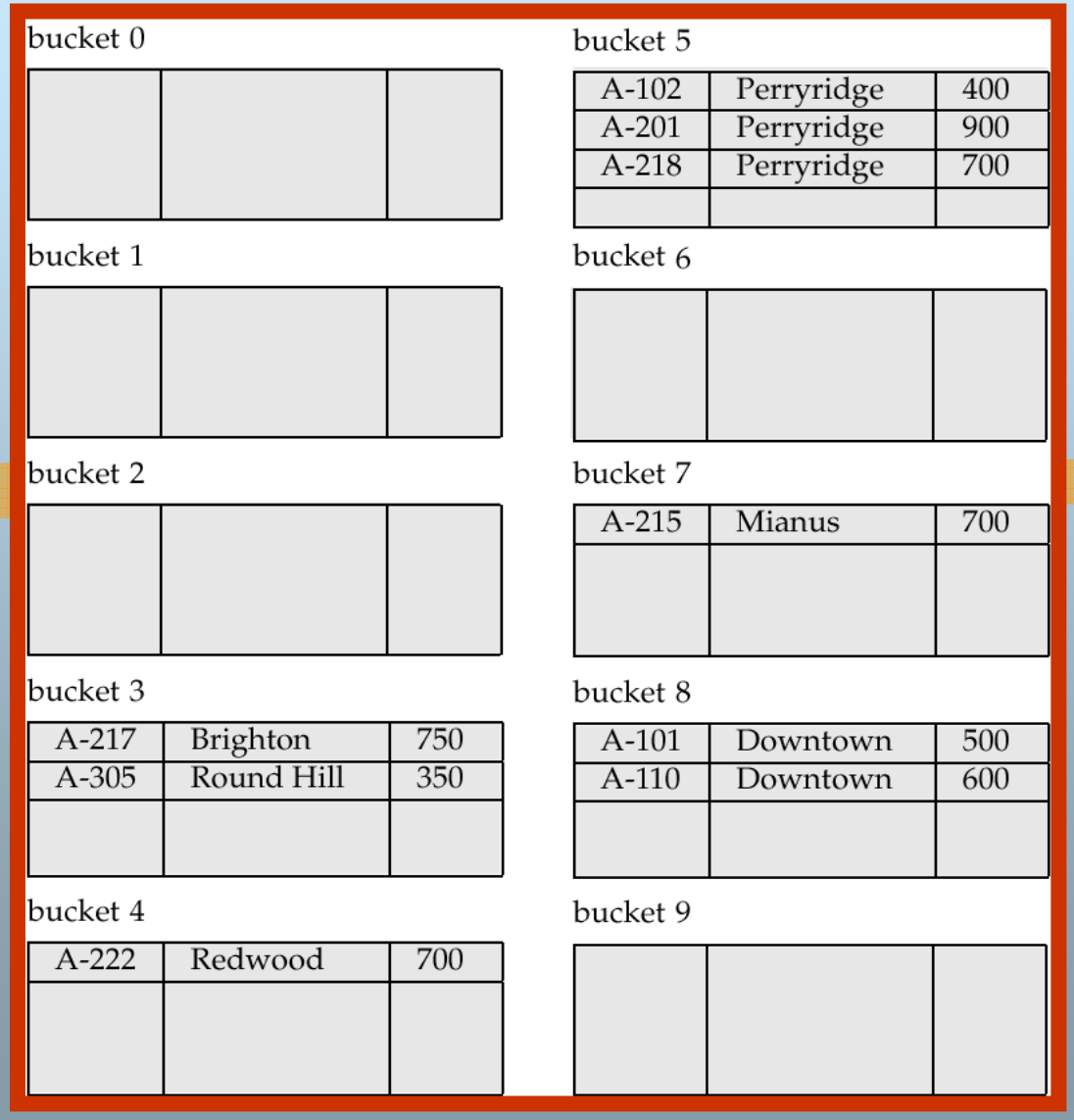
Exemplo de uma organização de arquivo de Hash (Cont.)

Uma organização de arquivo de Hash de Contas, usando o nome_agência como chave (Veja figura no próximo slide.)

- Existem 10 buckets,
- A representação binária do *l*-ésimo caracter do alfabeto é assumida como o inteiro *i.*
- A função de hash retorna a suma das representações binárias dos caracteres módulo 10
  - Ex. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

Exemplo de uma organização de arquivo de Hash

Uma organização de arquivo de Hash de Contas, usando o nome_agência como chave (veja slide anterior para detalhes).



Funções de Hash

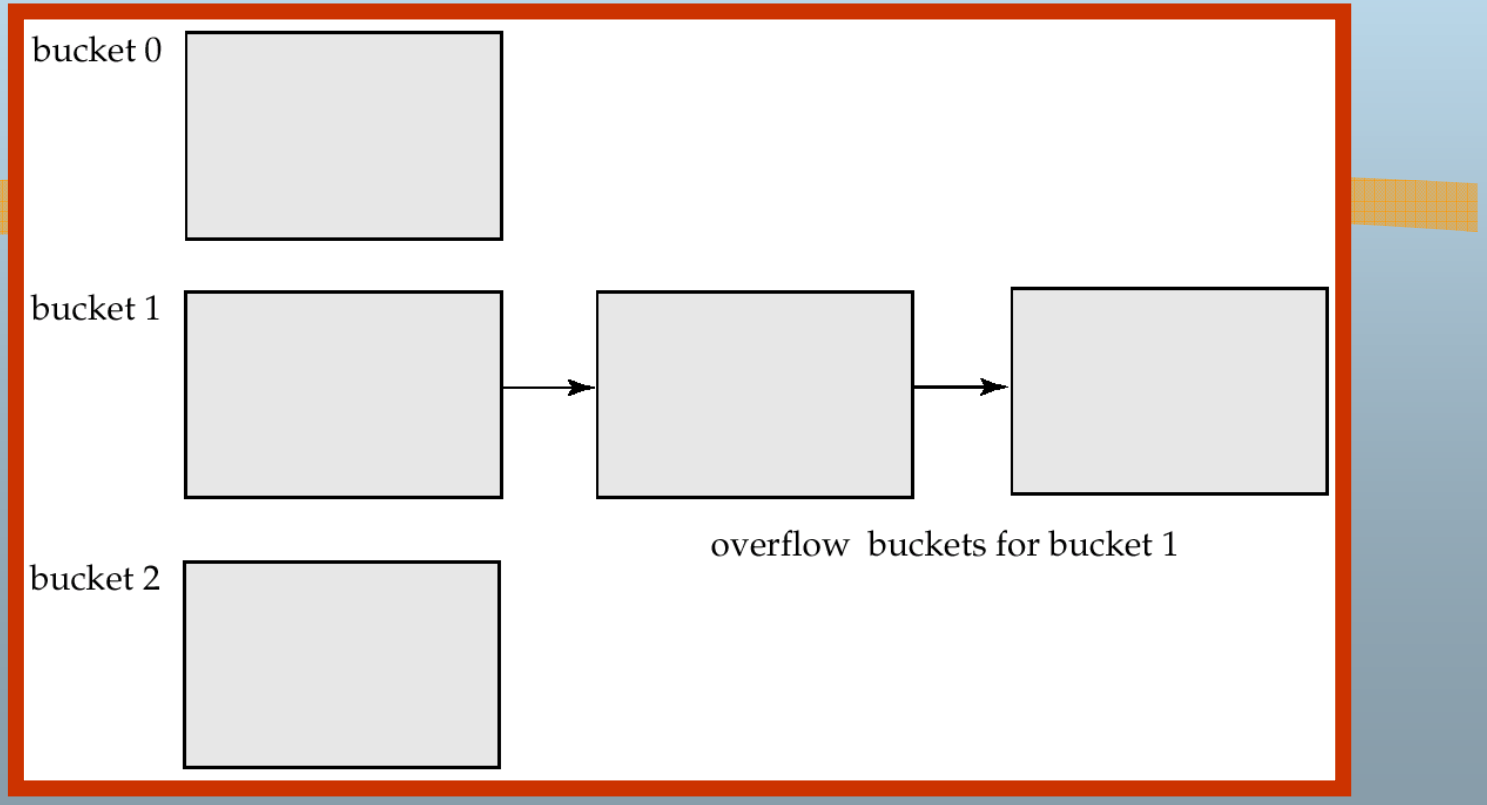
- A pior função mapeia todos os valores das chaves de busca para o mesmo bucket; isto faz o tempo de acesso proporcional ao número de valores das chaves de busca no arquivo.
- Uma função de hash ideal é **uniforme**, isto é, a cada bucket é atribuído o mesmo número de valores das chaves de busca do conjunto de todos os possíveis valores.
- Uma função de hash ideal é **aleatória**, assim cada bucket terá quase o mesmo número de valores atribuídos a ele, independente da distribuição real dos valores da chave de busca no arquivo.
- Funções de hash típicas fazem cálculos baseados na representação interna da chave de busca.
  - Por exemplo, para uma chave de busca cadeia de caracteres, a representação binária de todos os caracteres na cadeia poderiam ser somados e a suma módulo o número de buckets poderia ser retornado.

Tratamento de Overflows de Buckets

- Um Overflow de Buckets pode ocorrer devido a
  - Insuficiente número de buckets
  - Distorção na distribuição de registros. Isto pode ocorrer por dois motivos:
    - vários registros podem ter a mesma chave de busca
    - A função de hash escolhida produz uma distribuição não uniforme das chaves de busca
- Embora a probabilidade do overflow de buckets pode ser reduzido, ele não pode ser eliminado; ele é tratado usando *buckets de overflow*.

Tratamento de Overflows de Buckets (Cont.)

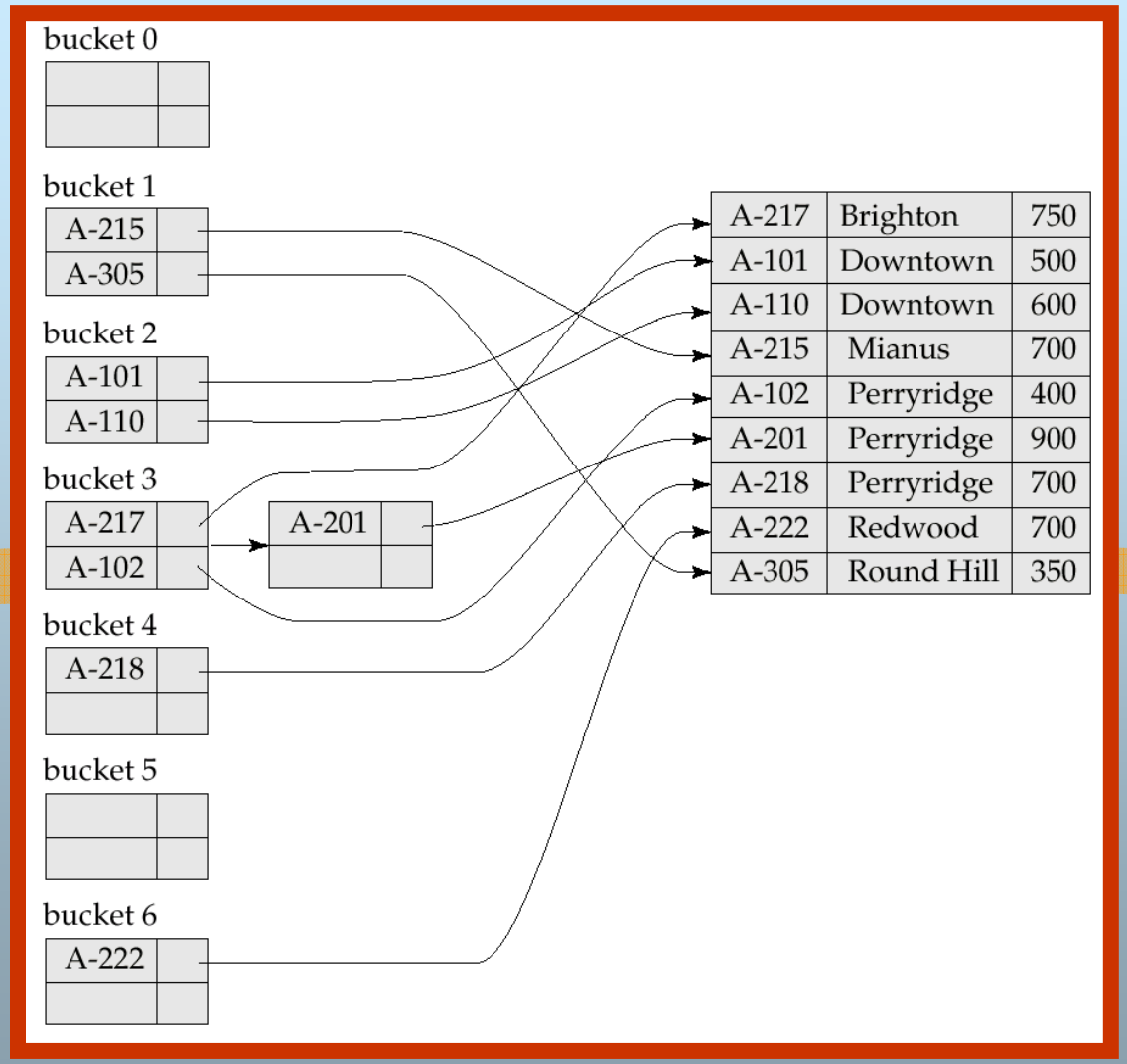
- Encadeamento de Overflow – os buckets de overflow de um dado bucket são encadeados juntos numa lista encadeada.
- O esquema acima é chamado de hashing fechado.
  - Uma alternativa, chamada de hashing aberto, que não usa buckets de overflow, não é adequada para aplicações de bancos de dados.



Índices de Hash

- Hashing pode ser usado não só para a organização de arquivos, mas também para a criação de estruturas de índices.
- Um **índice de hash** organiza as chaves de busca, com seus ponteiros de registros associados, numa estrutura de arquivo hash.
- Falando estritamente, os índices de hash são sempre índices secundários
  - Um índice de hash não é necessário com uma estrutura de índice agrupado, pois, se um arquivo já estiver organizado por hashing, não será necessária uma estrutura de índice de hash separada.
  - Entretanto, como a organização de arquivo de hash oferece o mesmo acesso direto que a indexação, fazemos de conta que um arquivo organizado por hashing também possui um índice de hash agrupado sobre ele.

Exemplo de índice de Hash



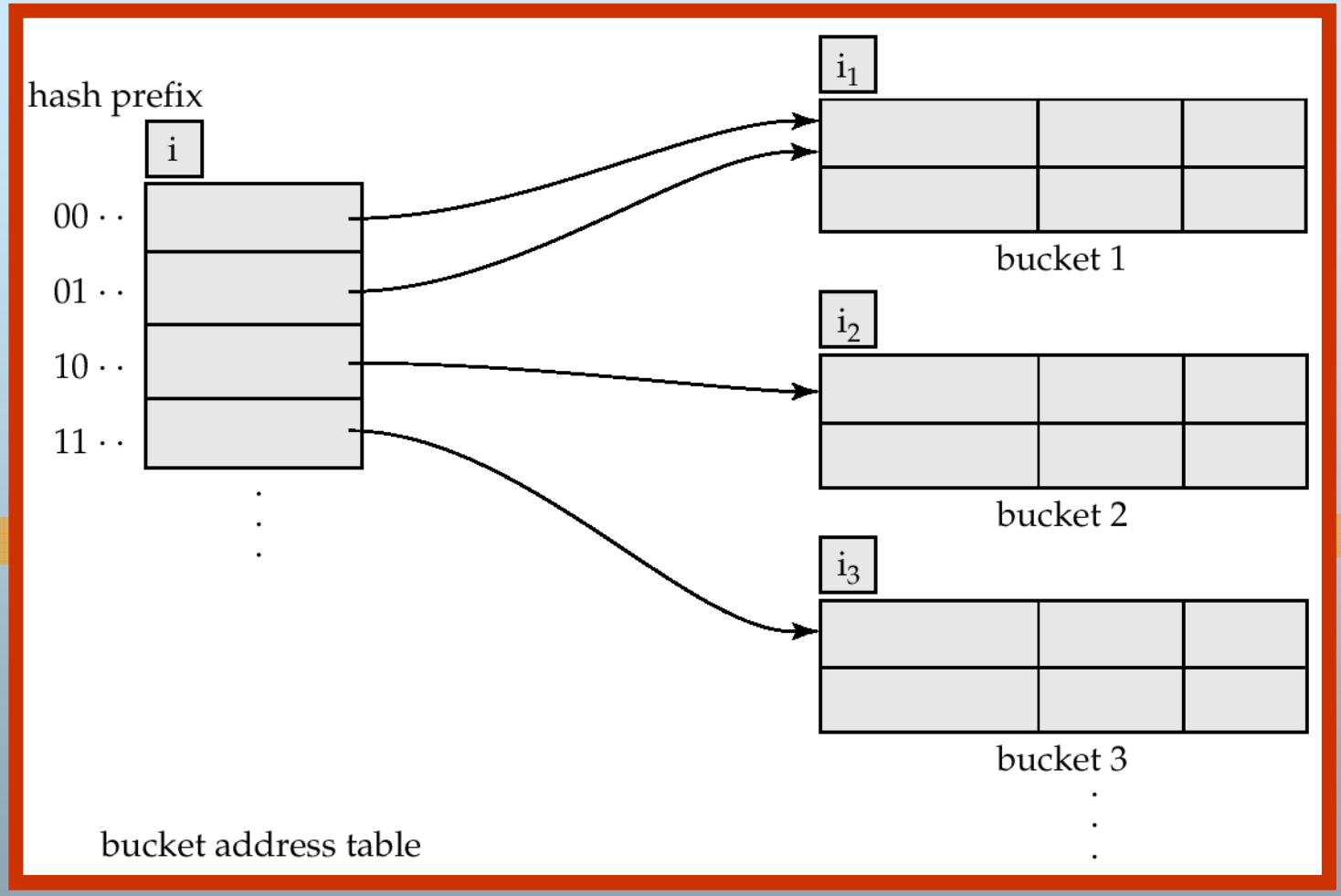
Deficiências do Hashing Estático

- No hashing estático, a função *h* mapeia valores da chave de busca a um conjunto fixo *B* de endereços de bucket.
  - Bancos de dados crescem com o tempo. Se o número inicial de buckets é muito pequeno, o desempenho degradará devido a muitos overflows.
  - Se o tamanho do arquivo em algum ponto no futuro é antecipado e o número de buckets atribuído de acordo com isso, uma quantidade significante de espaço será desperdiçado inicialmente.
  - Se o banco de dados diminui, de novo o espaço será desperdiçado.
  - Uma opção é a re-organização periódica do arquivo com uma nova função de hash, mas isto é muito custoso.
- Estes problemas podem ser evitados usando técnicas que permitem o número de buckets ser modificado dinamicamente.

Hashing Dinâmico

- Bom para bancos de dados que crescem e diminuem no tamanho
- Permite que função de hash seja modificada dinamicamente
- **Hashing Extensível** – uma forma de hashing dinâmico
  - Função de Hash gera valores por um intervalo relativamente grande — a saber, inteiros binários de *b*-bit, com $b = 32$.
  - Em qualquer instante é usado somente um prefixo da hash função de hash para indexar na tabela de endereços de buckets
  - Seja o tamanho do prefixo *i* bits, $0 \leq i \leq 32$.
  - O tamanho da tabela de endereços de Bucket = 2^{i}. Inicialmente $i = 0$
  - O valor de *i* cresce e diminui na medida que o tamanho do banco de dados cresce e diminui.
  - Entradas múltiplas na tabela de endereços de bucket podem apontar a um bucket.
  - Assim, o número real de buckets é $< 2^{i}$
    - O número de buckets também muda dinamicamente devido à junção e divisão de buckets.

Estrutura Geral do Hash Extensível



Nesta estrutura, $i_2 = i_3 = i$, enquanto $i_1 = i - 1$ (veja próximo slide para detalhes)

Uso da Estrutura de Hash Extensível

- Cada bucket *j* armazena um valor i_j; todas as entradas que apontam ao mesmo bucket têm os mesmos valores nos primeiros i_j bits.
- Para localizar o bucket contendo a chave de busca K_j:
  1. Calcular $h(K_j) = X$
  2. Use os primeiros *i* bits de ordem maior de *X* como um desplazamento na tabela de endereços de bucket, e segue o ponteiro de bucket na entrada da tabela
- Para inserir um registro com valor de chave de busca K_j
  - Segue o mesmo procedimento para buscar e localizar o bucket, digamos *j*.
  - Se houver espaço no bucket *j* insere o registro nele.
  - Senão o bucket deve ser dividido e os registros atuais e o novo devem ser redistribuídos (próximo slide.)
    - Buckets de Overflow usados em alguns casos (veremos brevemente)

Atualizações na Estrutura de Hash Extensível

Para dividir um bucket *j* quando inserimos um registro com valor da chave de K_j:

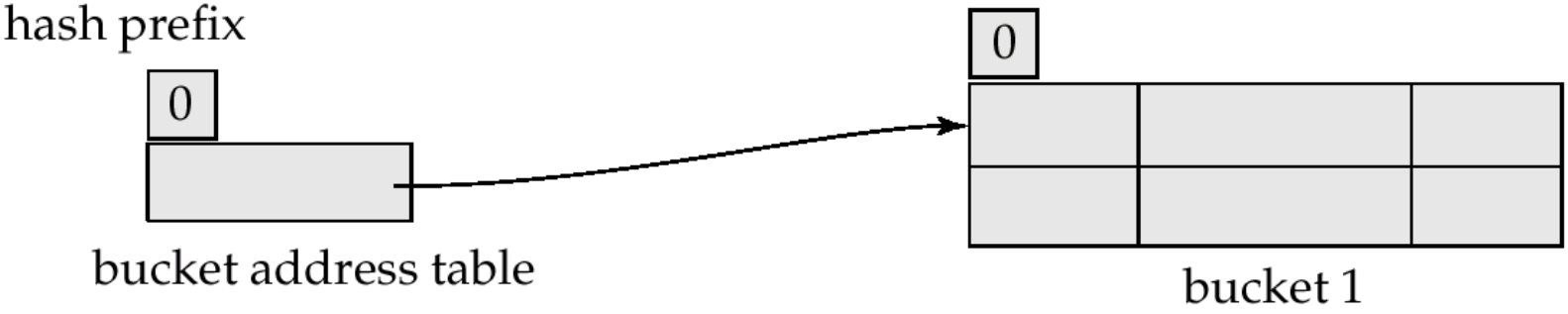
- Se $i > i_j$ (mais de um ponteiro para o bucket *j*)
  - Atribuir um novo bucket *z*, e colocar i_j e i_z para ser o velho $i_j + 1$.
  - Fazer a segunda metade dos endereços de bucket das entradas da tabela que apontavam a *j* para apontar a *z*
  - Remover e inserir de novo cada registro no bucket *j.*
  - Recalcular o novo bucket para K_j e inserir o registro no bucket (divisões adicionais são necessárias se o bucket ainda está cheio)
- Se $i = i_j$ (somente um ponteiro para o bucket *j*)
  - incrementar *i* e dobrar o tamanho da tabela de endereço de buckets.
  - Trocar cada entrada da tabela por duas entradas que apontem ao mesmo bucket.
  - Recalcular nova entrada na tabela de endereços de buckets para K_j Agora $i > i_j$ assim usar o primeiro caso acima.

Atualizações na Estrutura de Hash Extensível (Cont.)

- Quando inserimos um valor, se o bucket continua cheio após várias divisões (isto é, *i* alcança algum limite *b*) criar um bucket de overflow no lugar de dividir a tabela de entradas de buckets ainda mais.
- Para remover uma chave,
  - Achar ela no bucket e remover ela.
  - O bucket pode ser removido se ele ficar vazio (com atualizações apropriadas na tabela de endereços de bucket).
  - A união de buckets pode ser feita (pode se unir somente com um bucket que tenha o mesmo valor de i_j e igual prefixo i_j –1, se ele existir)
  - A diminuição do tamanho da tabela de endereços de bucket também é possível
    - PS: esta diminuição é uma operação custosa e deveria ser feita somente se o número de buckets ficar muito menor que o tamanho fa tabela

Uso da Estrutura de Hash Extensível: Exemplo

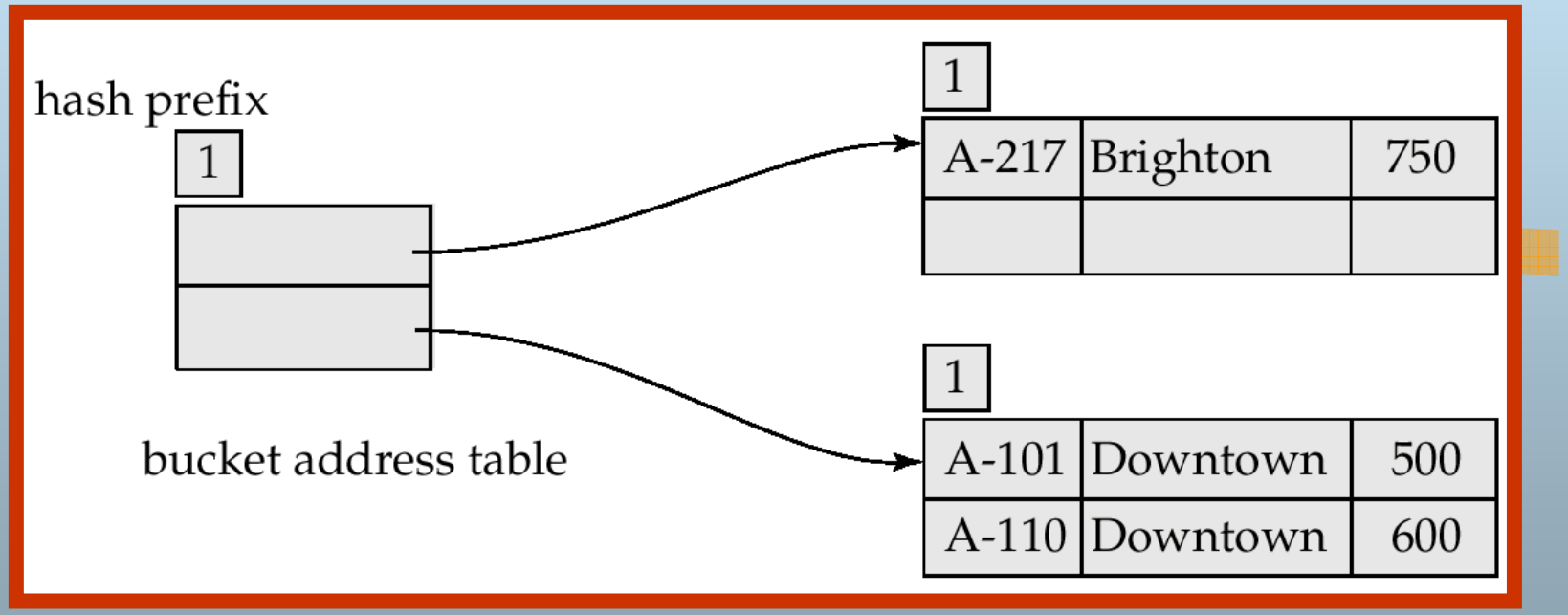
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Estrututra de Hash inicial, tamanho do bucket = 2

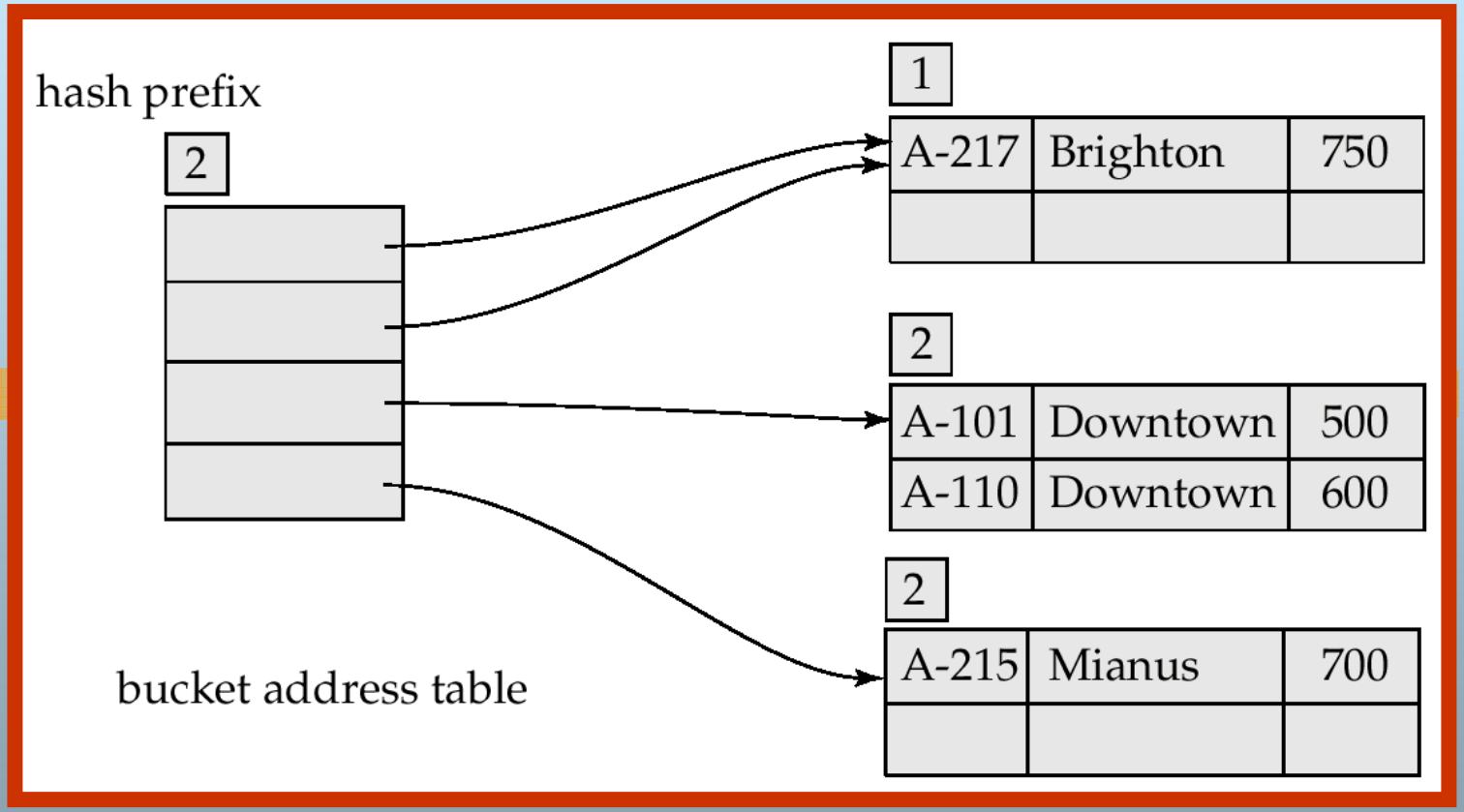
Exemplo (Cont.)

- Estrutura de Hash depois da inserção de um registro Brighton e dois Downtown

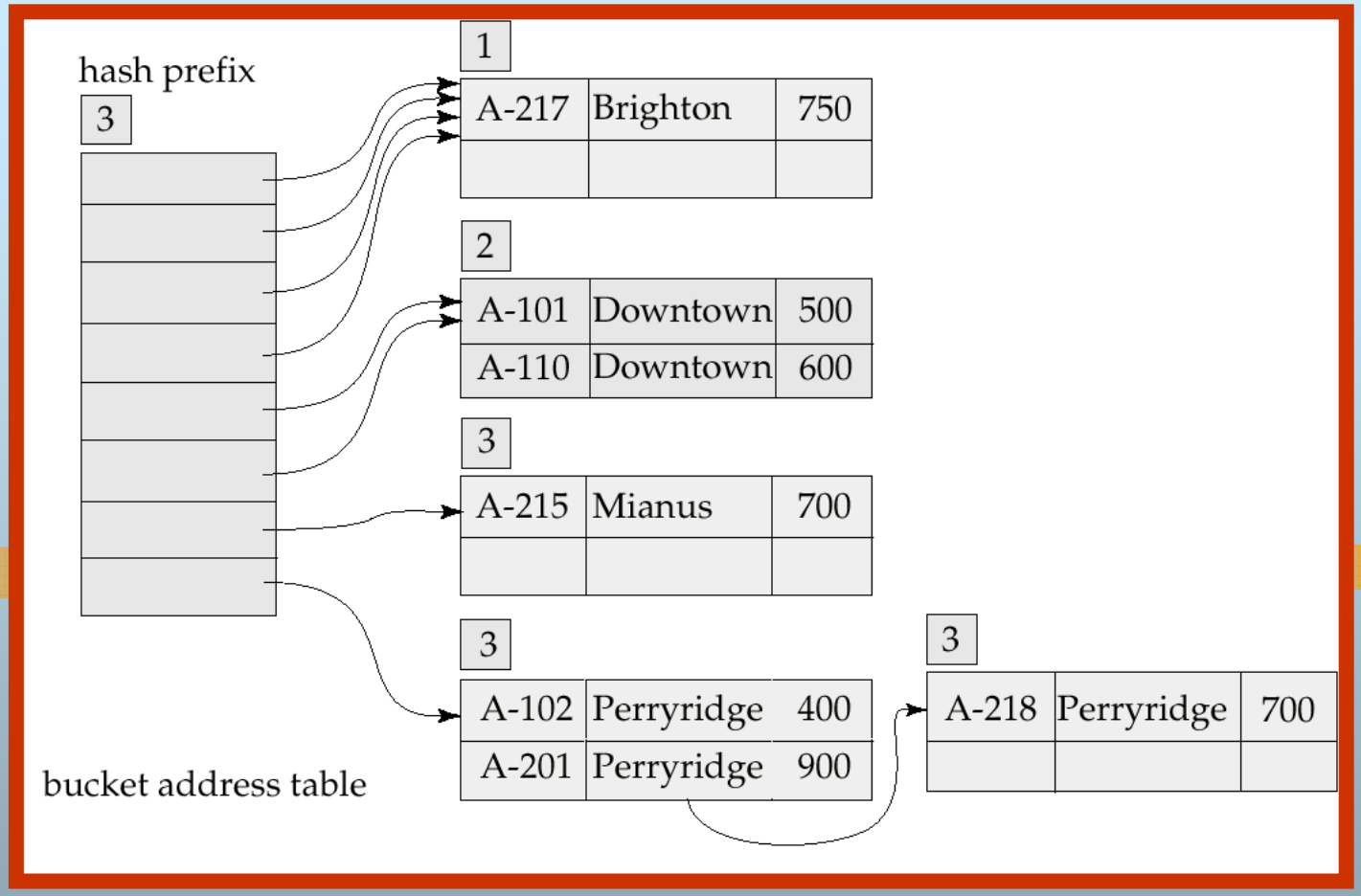


Exemplo (Cont.)

Estrutura de Hash depois da inserção do registro Mianus



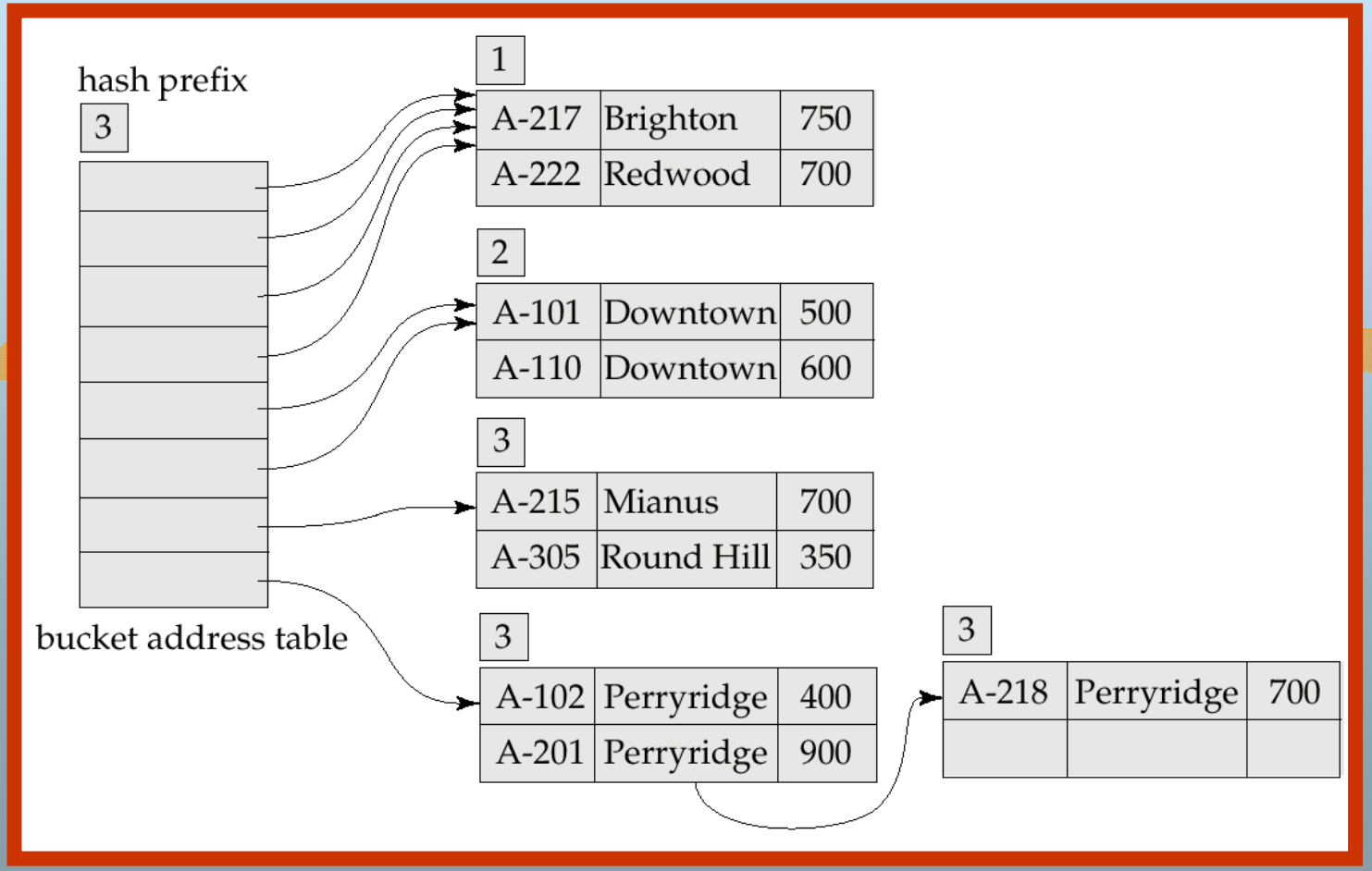
Exemplo (Cont.)



Estrutura de Hash após inserção de três registros Perryridge

Exemplo (Cont.)

- Estrutura de Hash structure após inserção dos registros Redwood e Round Hill



Hashing extensível vs. Outros Esquemas

- Benefícios do hashing extensível :
  - Desempenho do Hash não degrada com o crescimento do arquivo
  - Mínimo overhead de espaço
- Desvantagens do hashing extensível
  - Nível extra de indireção para achar os registros desejados
  - A tabela de endereços de bucket pode ficar muito grande (maior que a memória)
    - Precisa de uma estrutura em árvore para achar o registro desejado na estrutura!
  - Mudar o tamanho da tabela de endereços de buckets é uma operação custosa
- Hashing Linear é um mecanismo alternativo que evita estas desvantagens ao custo possível de mais buckets de overflows

Comparação de Indexação Ordenada e Hashing

- Custo da re-organização periódica
- Freqüência Relativa de inserções e remoções
- É desejável otimizar o tempo médio de acesso às custas de aumentar o tempo de acesso ao pior caso?
- Tipos esperados de consultas:
  - Hashing é geralmente melhor na recuperação de registros tendo um valor específico da chave.
  - Se as consultas que especificam intervalos de valores são comuns, índices ordenados são preferidos

Definição de Índices em SQL

- Criar um índice

  create index < nome-índice> **on** < nome-relação>
  (<lista-atributos>)

  Ex: **create index** *b-índice* **on** *agência(nome-agência)*

- Usar **create unique index** para indiretamente especificar e reforçar a condição de que a chave de busca é uma chave candidata.
  - Realmente não é preciso se a restrição de integridade **unique** de SQL
- Muitos sistemas de BDs oferecem meios de especificar o tipo de índice (árvore B+, hashing). Também índice agrupado.
- Eliminar um índice

  drop index <nome-índice>

Acesso por Chaves Múltiplas

- Uso de índices múltiplos por certo tipo de consultas.
- Exemplo:

  select *numero-conta*

  from *conta*

  where *nome-agência* = “Perryridge” **and** *saldo* = 1000
- Estratégias possíveis de processamento de índices usando consultas sobre um único atributo:
  1. Use o índice sobre *nome-agência* para achar contas com saldos de $1000; verifique *nome-agência* = “Perryridge”.
  2. Use o índice sobre *saldo* para achar contas com saldos de $1000; verifique *nome-agência* = “Perryridge”.
  3. Use o índice sobre *nome-agência* para achar ponteiros a todos os registros pertencendo à agência Perryridge. Igualmente, use o índice sobre *saldo*. Faça a interseção dos dois conjuntos de ponteiros obtidos.

Índices sobre Atributos Múltiplos

Suponhamos que temos um índice sobre a chave de busca combinada (*nome-agência, saldo*).

- Com a clausula **where**
  where *nome-agência* = “Perryridge” **and** *saldo* = 1000
  o índice sobre a chave combinada pegará somente os registros que satisfazem as duas condições.
  Usando índices separados é menos eficiente — podemos pegar vários registros (ou apontadores) que satisfazem somente uma das condições.
- Pode tratar também eficientemente
  where *nome-agência* = “Perryridge” **and** *saldo* < 1000
- Mas não pode tratar eficientemente
  where *nome-agência* < “Perryridge” **and** *saldo* = 1000
  Pode pegar muitos registros que satisfazem a primeira mas não a segunda condição.

Índices de mapa de Bits

- Índices de mapas de bits são um tipo especial de índice projetado para a consulta eficiente com chaves múltiplas
- Registros em uma relação são assumidos para ser numerados seqüencialmente desde, digamos, 0
  - Dado um número *n* deve ser fácil recuperar o registro *n*
    - Particularmente fácil se os registros são de tamanho fixo
- Aplicáveis sobre atributos que têm um número relativamente pequeno de valores diferentes
  - Ex. gênero, país, estado, …
  - Ex. nível-ingresso (ingresso dividido em um pequeno número de níveis tais como: 0-9999, 10000-19999, 20000-50000, 50000-infinito)
- Um mapa de bits é simplesmente um array de bits

Índices de mapa de Bits (Cont.)

- Na sua forma mais simples um índice de mapa de bits sobre um atributo tem um mapa de bits para cada valor do atributo
  - Um mapa de bits tem tantos bits quanto registros
  - Em um mapa de bits o valor para um registro é 1 se o registro tem o valor v para o atributo, e é 0 no outro caso.

record number	*name*	*gender*	*address*	*income -level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	1 0 0 1 0
f	0 1 1 0 1

Bitmaps for *income-level*

L1	1 0 1 0 0
L2	0 1 0 0 0
L3	0 0 0 0 1
L4	0 0 0 1 0
L5	0 0 0 0 0

Índices de mapa de Bits (Cont.)

- Índices de mapas de bits são úteis para consultas sobre atributos múltiplos
  - Não é particularmente útil para consultas de um só atributo
- Consultas são respondidas usando operações de mapas bits
  - Interseção (and)
  - União (or)
  - Complementação (not)
- Cada operação pega dois mapas de bits do mesmo tamanho e aplica a operação correspondente sobre os bits para obter o mapa de bits resultante
  - Ex. 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Homens com nível de ingresso L1: 10010 AND 10100 = 10000
    - Pode então recuperar as tuplas requeridas.
    - A contagem do número de casamentos é ainda mais rápido

Índices de Mapas de Bits (Cont.)

- Os índices de mapas de bits geralmente são muito pequenos comparados com o tamanho da relação
- A remoção precisa ser executada apropriadamente
  - Mapas de bits de Existência para declarar se existe um registro válido na localização do registro