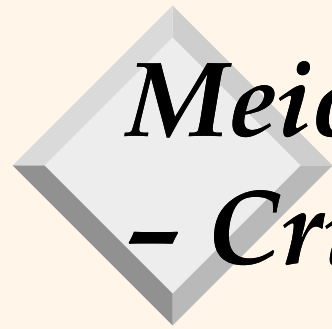


Projeto Físico

Prof. Heloise Manica P. Teixeira

Capítulo 10 e 11

KORTH, H. F.; SILBERSCHATZ, A.; SUDARSHAN, S.
Sistema de Banco de Dados,
Tradução da 5ª Edição, Campus, 2006.



Meios físicos de armazenamento

– Critérios de Classificação

- ❖ Velocidade na qual os dados podem ser acessados
- ❖ Custo por unidade de dados
- ❖ Confiabilidade com relação à
 - Perda de dados por falta de energia ou falhas do sistema
 - Falha física do dispositivo de armazenamento, que podem ser:

Em termos de confiabilidade, mídias podem ser:

- Voláteis: perdem conteúdo quando não alimentadas
- Não voláteis: inclui armazenamento secundário e terciário, bem como memória principal alimentada por bateria.



Meios Físicos de Armazenamento

- ❖ **Cache** – A mais rápida e cara forma de armazenamento;
 - É volátil, de acesso aleatório;
 - É gerenciada pelo hardware / sistema operacional.
- ❖ **Memória Principal**
 - Acesso rápido, mas geralmente muito pequena para conter todo o banco de dados;
 - Custos têm diminuído de forma constante e rápida;
 - Volátil: seu conteúdo é normalmente perdido em caso de falta de energia ou falha do sistema;

Meios Físicos de Armazenamento

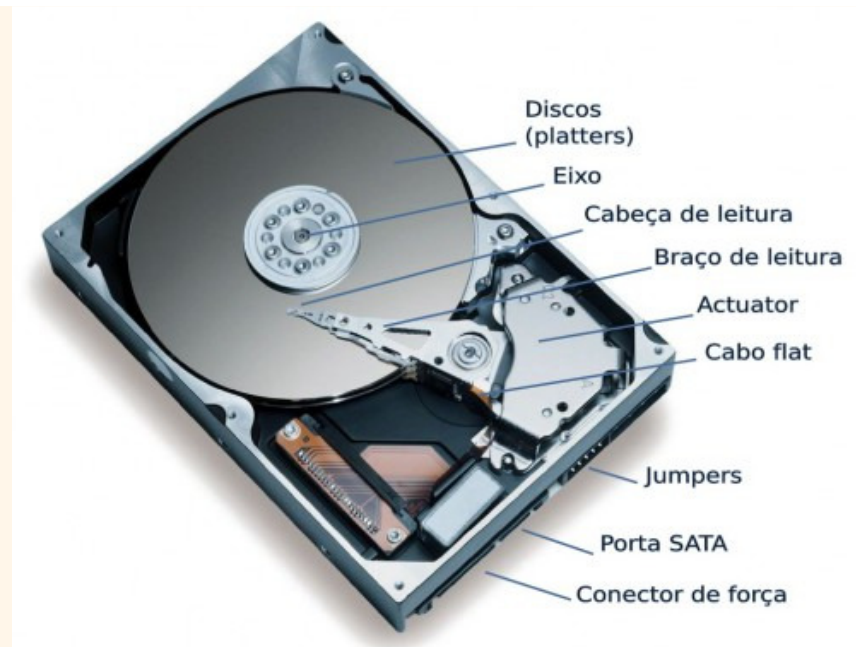
Memória Flash (EEPROM)

- Não volátil;
- Executa leituras quase tão rápido quanto a memória principal;
- Os dados podem ser escritos uma vez, mas para se poder escrever novamente deve-se apagar todo o conteúdo primeiro.
- Como desvantagem, suporta apenas um número limitado de ciclos para apagá-la.
 - * Escrever e apagar é lento



❖ Disco Magnético

- ❖ Meio primário para o armazenamento de dados por longos períodos;



- Em geral armazena o banco de dados inteiro.
- Os dados devem ser movidos do disco para a memória principal para serem acessados; ao final das operações, devem ser escritos de volta no disco
- **Acesso direto** – é possível ler dados no disco em qualquer ordem
- Maior capacidade e custo que memória flash e principal Cresce constantemente e rapidamente com melhorias tecnológicas (fator de 2 a 3 a cada 2 anos)
- Sobrevive a falhas de energia e falhas no sistema falha de disco pode destruir dados, mas é raro

Meios Físicos de Armazenamento

❖ **Armazenamento Ótico**

- não volátil (populares CD-ROM, DVD e Blu-Ray)
- Os dados são lidos oticamente por laser;
- Write-one, read-many (WORM) (CD-R, DVD-R, DVD+R)
- Multiple write versions disponiveis (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Leitura e escrita mais lenta que nos disco magnético



Meios Físicos de Armazenamento

❖ Armazenamento em Fita

- Não volátil, usado primordialmente para backup (recuperação de falhas de disco) e para arquivos de dados
- Apresentam acesso sequencial – muito mais lentos que discos
- Grande capacidade (fitas de 185 terabytes podem ser encontradas)
- Pode ser removida da unidade de fita \Rightarrow são um meio barato e fácil de armazenamento (drives são caros).

Fitas magnéticas

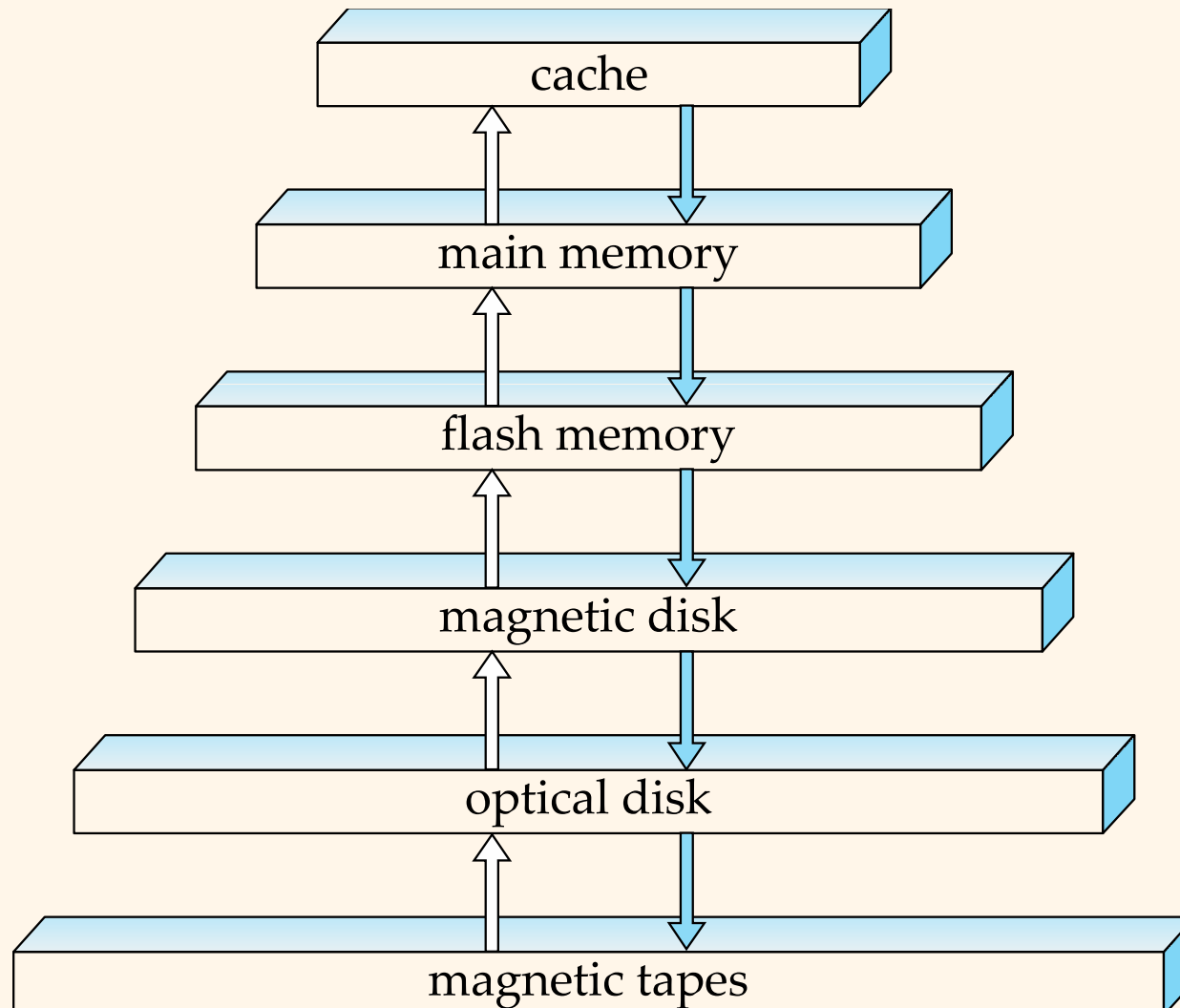


(Foto: Licença creative commons)

Sony cria fita magnética de 185 terabytes

Hierarquia de Dispositivos:

Velocidade x Custo





Hierarquia de Dispositivos(Cont.)

- ❖ **Armazenamento Primário:** mídias mais rápidas, porém voláteis (**cache, memória principal**)
- ❖ **Armazenamento Secundário:** formado por mídias não voláteis com tempos de acesso relativamente rápidos; também chamado de armazenamento **on-line** (**memória flash, discos magnéticos**)
- ❖ **Armazenamento Terciário:** nível mais baixo da hierarquia, composto por mídias não voláteis com tempo de acesso lento; também chamado de armazenamento **off-line** (**fita magnética, armazenamento ótico**)

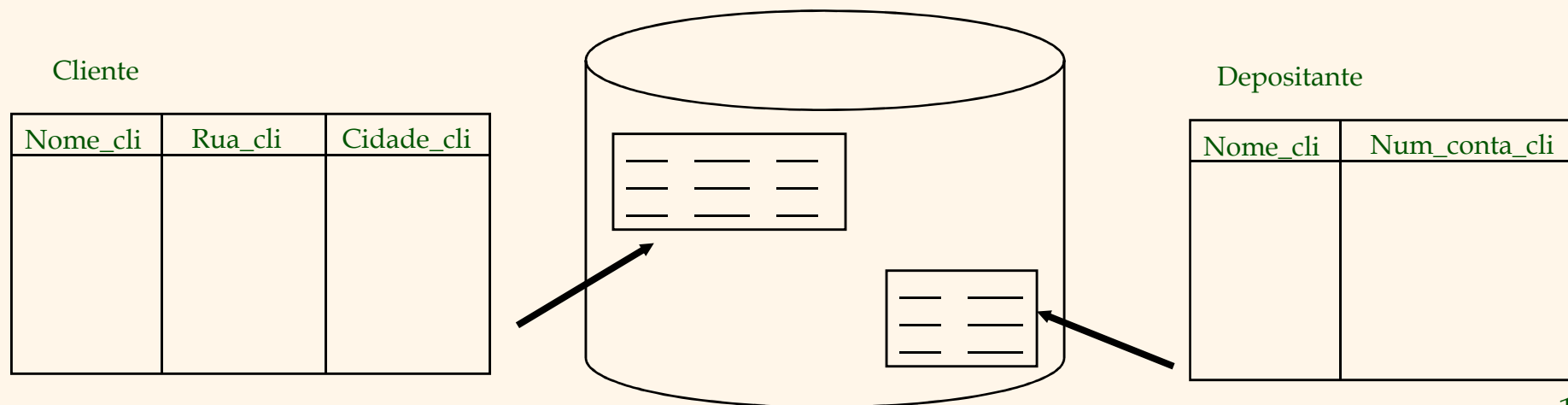


Organização de Arquivo

- ❖ O banco de dados é armazenado como um conjunto de **arquivos**. Cada arquivo é uma **seqüência de registros**. Um registro é uma **seqüência de campos**.
- ❖ Em um BD relacional, as tuplas de relações distintas têm geralmente tamanhos diferentes.
- ❖ Apesar dos blocos de disco terem tamanho fixo, (propriedades físicas do disco e SO), o tamanho dos registros pode variar.
- ❖ Abordagens para mapeamento de BD em arquivos:
 - Registros de **tamanho fixo**
 - Registros de **tamanho variável**

Registros de Tamanho Fixo

- ❖ Cada arquivo armazena apenas um tipo de registro
- ❖ Diferentes arquivos são usados para diferentes relações
- ❖ Implementação mais fácil



Registros de Tamanho Fixo (Cont)

- ❖ Exemplo: arquivo de registro de *contas*
type *deposito* = record

nome_agencia: char (22);

numero_conta: char (10);

saldo: real;

end

- ❖ Se cada caractere ocupa um byte e um real ocupa 8 bytes, o registro teria 40 bytes.
- ❖ Usar os primeiros 40 bytes para o 1º registro, os 40 bytes seguintes para o 2º e assim por diante

Registros de Tamanho Fixo (Cont)

- ❖ Armazena registro i iniciando do byte $n^* (i - 1)$, indo até $n^* (i) - 1$

Onde: n é o tamanho de cada registro

0	1	2	3	4					38	39	40	41					47	48	49
X	X	X	X	X	...				X	X	Y	Y	...				Y	Y	Y
Y	Y	Y	Y	Y	...	Y						
50	51	52	53	54					79					90	91				

1º reg. (X): $n^* (i - 1) \Rightarrow 40 (1 - 1) \Rightarrow 0$ até $n^*(i) - 1 \Rightarrow 40 - (1) \Rightarrow 39$

2º reg.(Y): $n (i - 1) \Rightarrow 40 (2 - 1) \Rightarrow 40$ até $n(i) - 1 \Rightarrow 80 - (1) \Rightarrow 79$

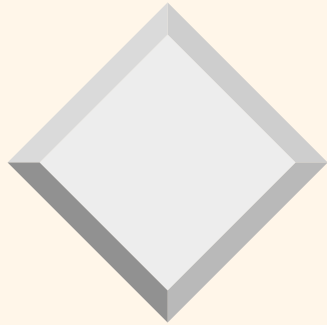
Registros de Tamanho Fixo (Cont)

❖ Problemas:

- O acesso aos registros é simples, mas os registros podem cruzar a fronteira dos blocos se o tamanho do bloco não for múltiplo do tamanho do registro.
- É difícil apagar um registro

❖ As alternativas para deleção do registro i são:

- mover registro $i + 1, \dots, n$ para $i, \dots, n - 1$
- mover registro n para i (n é o último registro)
- Encadear todos os registros livres em uma *lista livre*

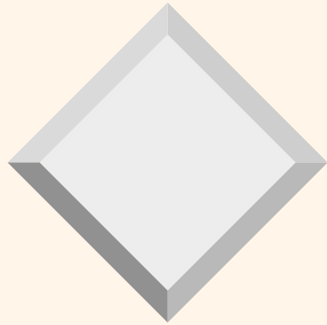


- ❖ Exemplo, o registro 2 ($i=2$) foi deletado,
- ❖ mover registro $i+1, \dots, n$ para $i, \dots, n-1$

0	1	2	3					38	39	40	41					47	48	49		
X	X	X	...					X	X			...								
			...					Z	...	Z	Z	Z	Z	...			Z	Z	Z	
Z	Z	Z		Z	A	A	...													
100	101					119	120													

- ❖ Então, mover registro $i+1, \dots, n$ para $i, \dots, n-1$
- ❖ Mover registro 3 ($i+1$), até 4 (n) para 2 (i), ..., 3 ($n-1$)

0	1	2	3					38	39	40	41					47	48	49		
X	X	X	...					X	X	Z	Z	...			Z	Z	Z			
Z	Z	Z	...			Z	A	...	A	A	A	A	...			A	A	A		
A	A	A		A			...													
100	101					119	120													



- ❖ Exemplo, o registro 2 ($i=2$) foi deletado,
- ❖ mover registro n para i (n é o último registro)

0	1	2	3					38	39	40	41					47	48	49	
X	X	X	...					X	X			...							
			...				Z	...	Z	Z	Z	Z	...			Z	Z	Z	
Z	Z	Z		Z	A	A	...												
100	101					119	120												

- ❖ Mover registro 4 para posição do 2

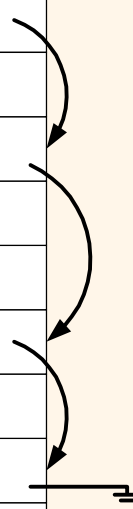
0	1	2	3					38	39	40	41					47	48	49
X	X	X	...					X	X	A	A	...			A	A	A	
A	A	A	...			A	Z	...	Z	Z	Z	Z	...			Z	Z	Z
Z	Z	Z		Z			...											
100	101					119	120											

Registros de Tamanho Fixo – Lista Livre

- ❖ Armazenar o endereço do primeiro registro cujo conteúdo foi deletado no cabeçalho do arquivo
- ❖ Usar este primeiro registro para armazenar o endereço do segundo registro disponível, e assim por diante
- ❖ Esses endereços armazenado podem ser considerados como ponteiros, já que “apontam” para a localização de um registro.

- Representação mais eficiente de espaço: reusa espaço para atributos normais de registros livres para armazenar ponteiros. (Nenhum ponteiro armazenado em registros *em uso*)

header				
record 0	Perryridge	A-102	400	
record 1				
record 2	Mianus	A-215	700	
record 3	Downtown	A-101	500	
record 4				
record 5	Perryridge	A-201	900	
record 6				
record 7	Downtown	A-110	600	
record 8	Perryridge	A-218	700	



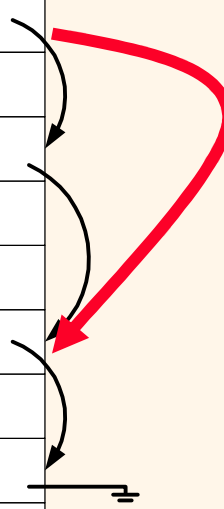
Registros de Tamanho Fixo

Lista Livre (Cont)

❖ Inserção de um novo registro

- ❖ utiliza-se o registro apontado pelo cabeçalho.
- ❖ O ponteiro do cabeçalho é trocado para apontar para o próximo registro disponível.
- ❖ Se não há, adiciona-se um novo registro no fim do arquivo

header				
record 0	Perryridge	A-102	400	
record 1	Novo	A-199	300	
record 2	Mianus	A-215	700	
record 3	Downtown	A-101	500	
record 4				
record 5	Perryridge	A-201	900	
record 6				
record 7	Downtown	A-110	600	
record 8	Perryridge	A-218	700	





Registros de Tamanho Fixo

Lista Livre (Cont)

- ❖ Inserção e remoção em arquivos de registros de tamanho fixo é simples de implementar, pois o espaço disponível deixado por um registro apagado é exatamente o espaço necessário para inserir um novo registro.



Registros de Tamanho Variável

- ❖ Aparecem em sistemas de BD de várias formas:
 - Armazenamento de múltiplos tipos de registros em um arquivo.
 - Tipos de registros que permitem tamanhos variáveis para um ou mais campos.
 - Tipos de registros que permitem campos repetidos (usado em alguns modelos de dados mais antigos).



Registros de Tamanho Variável

❖ Exemplo: **type** *lista_conta* = **record**

nome_agência: char(22);

informação_conta: **array**[1..oo] **of**

record;

número_conta: char(10);

saldo: real;

end

end

❖ Representação de cadeias de bytes

– Anexar um símbolo especial de *fim de registro* (\perp) no final de cada registro

– Como alternativa: armazenar o tamanho do registro no início de cada registro

Registros de Tamanho Variável

Representação de Cadeia de Bytes

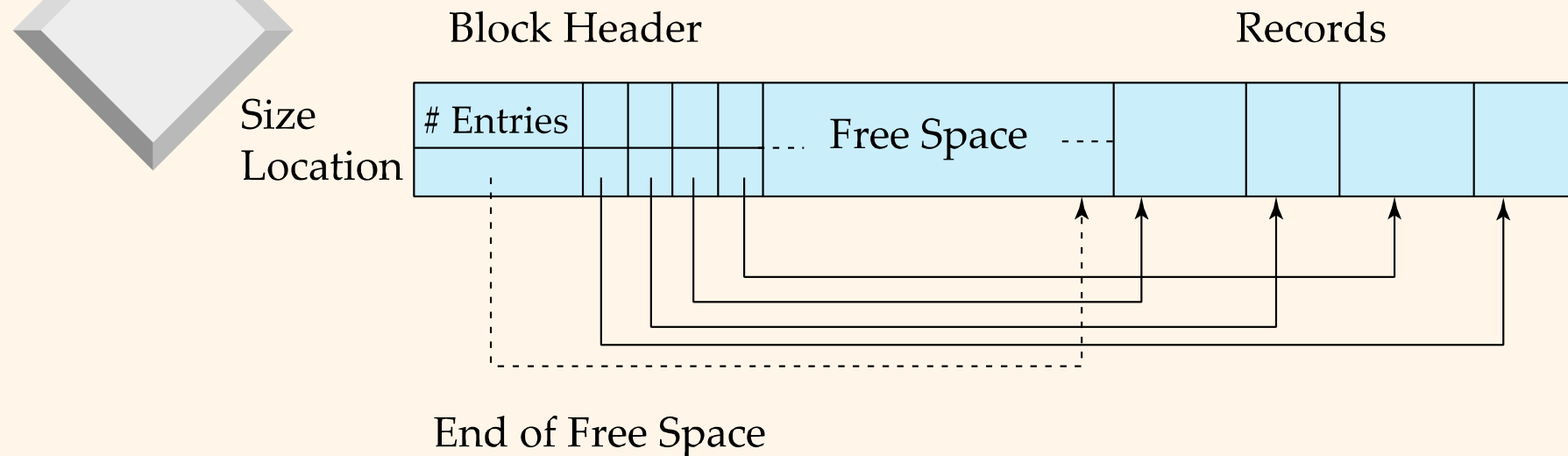
0	Perryridge	A-102	400	A-201	900	A-218	700	⊥			
1	Round Hill	A-305	350	⊥							
2	Mianus	A-215	700	⊥							
3	Downtown	A-101	500	A-110	600	⊥					
4											
5	Redwood	A-222	700	⊥							
6	Brighton	A-217	750	⊥							



Registro Tamanho Variável: Desvantagens

- ❖ Cadeia de bytes dificulta a reutilização do espaço ocupado por um registro apagado anteriormente
- ❖ Assim, esta representação não costuma ser utilizada para registros de tamanho variável
- ❖ *Slotted-page* (página vaga):
 - forma modificada da cadeia de bytes, organizando registros *dentro* de um único bloco

Registro Tamanho Variável: Slotted Page



- ❖ O cabeçalho, no início de cada bloco, contém:
 - Número de entradas de registro
 - Final de espaço livre no bloco
 - localização e o tamanho de cada registro
- ❖ Os registros são alocados contiguamente, iniciando a partir do fim do bloco.
- ❖ Se um registro é inserido, é alocado no fim do espaço livre, e uma entrada é adicionada ao cabeçalho.
- ❖ Se um registro é apagado, o espaço é liberado e sua entrada é configurada como apagada

Organização de Registros em Arquivos

- ❖ Dado um conjunto de registros, como ORGANIZÁ-LOS em um arquivo?
 - **Sequencial** – registros são armazenados em ordem seqüencial, baseada no valor da chave primária de cada registro
 - **Clustering** – registros de diferentes relações podem ser armazenados no mesmo arquivo; registros relacionados de diferentes relações são armazenados no mesmo bloco
 - **Heap** – um registro pode ser colocado em qualquer lugar do arquivo onde haja espaço; não há uma ordem; normalmente há um único arquivo para cada relação
 - **Hashing** – uma função hash é calculada sobre algum atributo de cada registro; o resultado especifica em qual bloco do arquivo o registro deve ser colocado

Organização *Sequencial* de Arquivo

- ❖ Os registros no arquivo são ordenados por uma chave primária e encadeados através de ponteiros
- ❖ Para minimizar acessos a blocos os registros são armazenados fisicamente na ordem da chave primária, ou o mais próximo possível dessa ordem.

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round Hill	A-305	350	

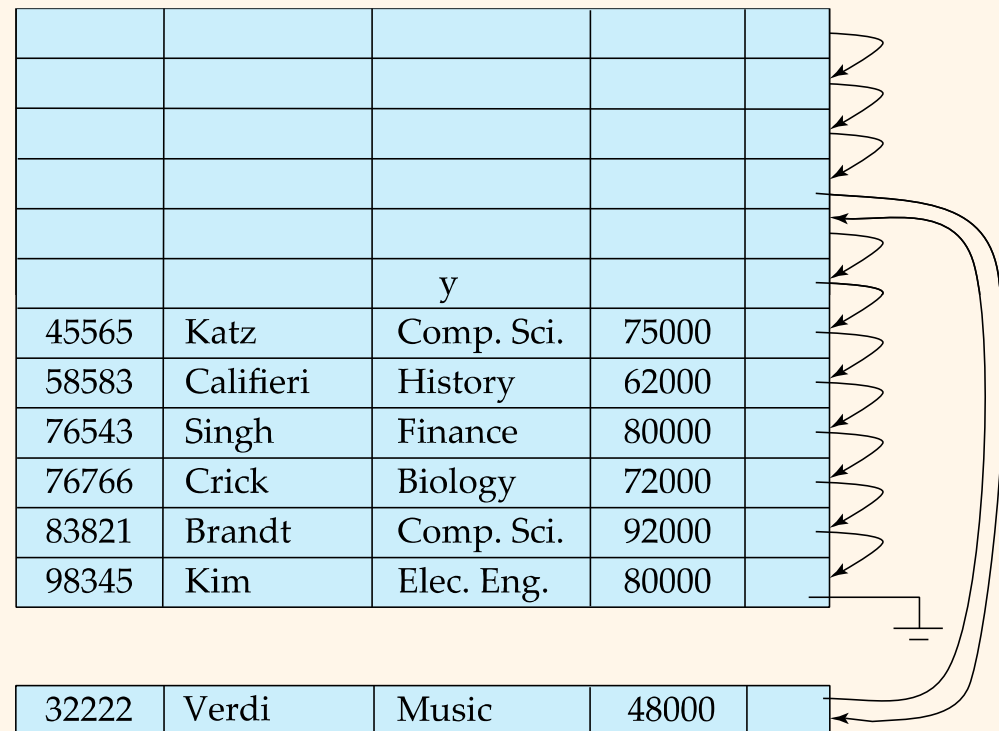


Organização Sequencial de Arquivo (Cont.)

❖ Inserção – localizar posição no arquivo onde o registro deve ser inserido

- se houver espaço livre, inserir o novo registro neste espaço
- se não, inserir o registro em um *bloco de overflow*
- Em ambos os casos, atualizar os ponteiros

❖ Necessidade de reorganizar o arquivo de tempos em tempos para restaurar a ordem sequencial



❖ Eliminação – usar a cadeia de ponteiros



Organização de Arquivo Clustering

- ❖ Muitos sistemas de BD relacional armazenam cada relação em um arquivo;
- ❖ Essa abordagem simples para a implementação de BD relacional torna-se menos satisfatória conforme o tamanho do BD aumenta.
- ❖ Muitos sistemas de BD de grande escala não se apoiam diretamente no SO para o gerenciamento de arquivos.
- ❖ Em vez disso, um grande arquivo de operação do sistema é alocado para o sistema de BD.
- ❖ Todas as relações são armazenadas nesse arquivo e o gerenciamento desse arquivo é deixado a cargo do SO.

Organização de Arquivo Clustering

- ❖ Armazena várias relações em um arquivo

department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

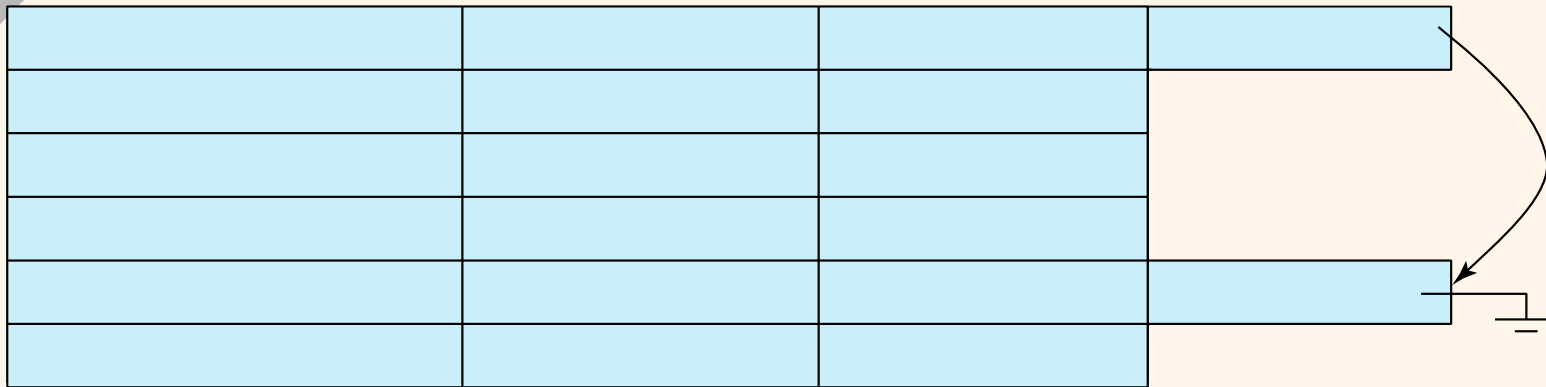
instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

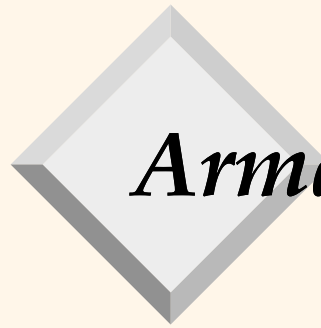
multitable clustering
of *department* and
instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Organização de Arquivo Clustering



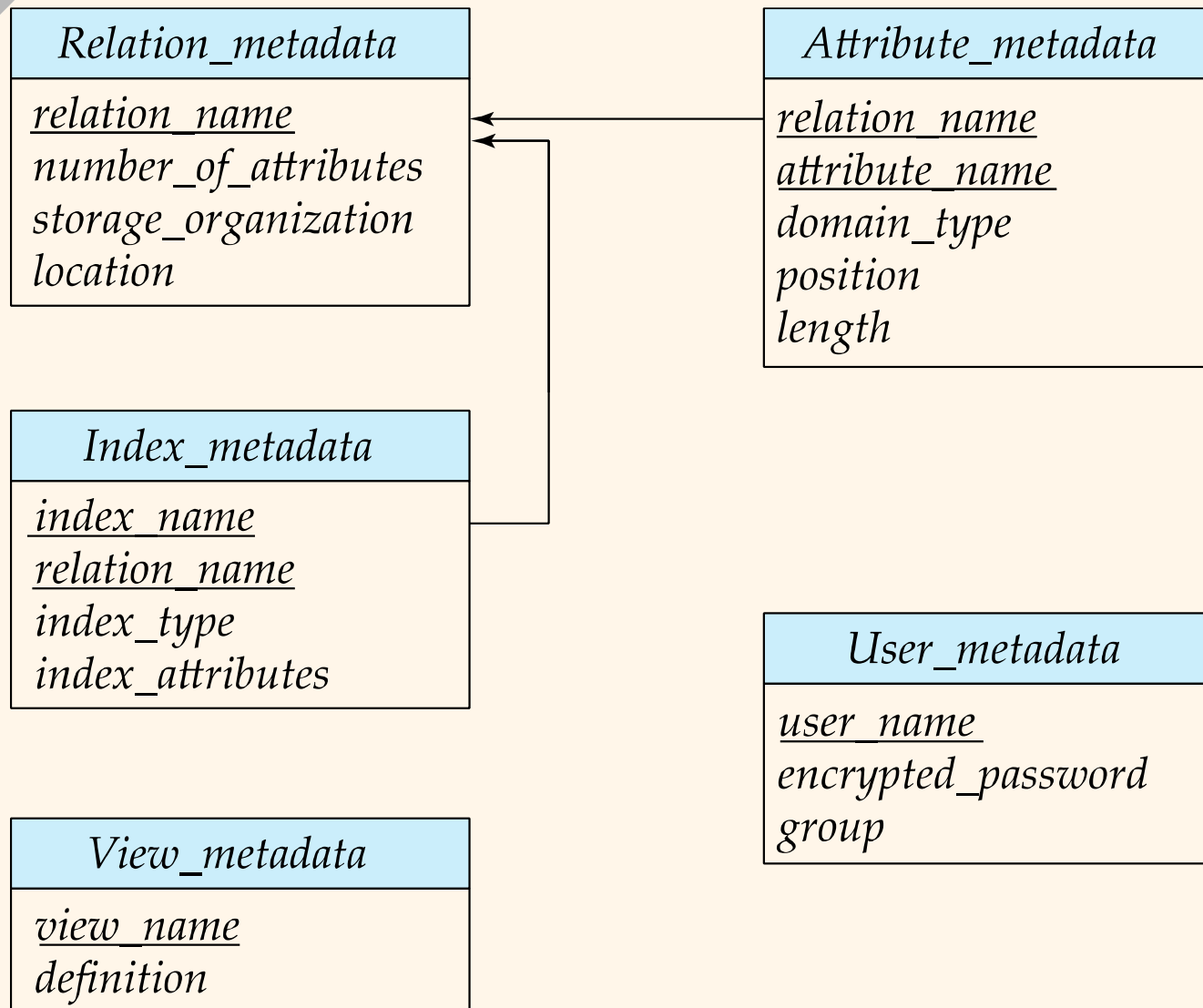
- Bom para consultas envolvendo departamento \bowtie instrutor
- Bom para consultas envolvendo um único departamento e seus instrutores
- Ruim para consultas envolvendo somente departamento
- Resulta em registros de tamanho variável
- Pode ser usado ponteiros para ligar registros de uma relação em particular (figura)
- A utilização de clustering depende dos tipos de consulta serão as mais frequentes

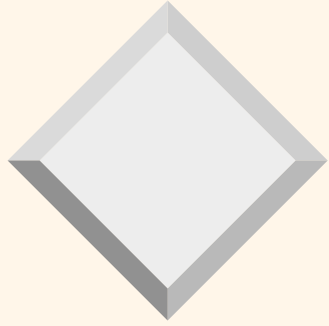


Armazenamento de Dicionário de Dados

- ❖ Dicionário de dados, ou catálogo do sistema, armazena metadados (dados sobre dados), como:
 - informação sobre relações, como:
 - ◆ nomes das relações
 - ◆ nomes e tipos dos atributos
 - ◆ informação sobre a organização física dos arquivos
 - ◆ dados estatísticos, como número de tuplas em cada relação
- ❖ Requisitos de integridade
- ❖ Definição de visões
- ❖ Dados sobre usuários do sistema (contas, senhas)
- ❖ Informações sobre índices

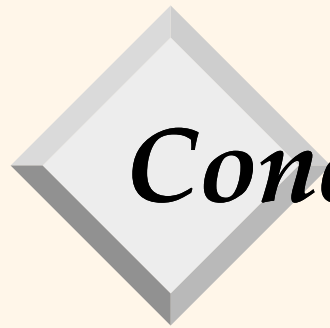
Armazenamento de Dicionário de Dados





Indexação e Hashing

Cap 11



Conceitos Básicos

- ❖ Mecanismos de **Indexação** são usados para aumentar a velocidade de acesso aos dados. Ex: catálogo de autores em uma biblioteca.
- ❖ **Chave de Procura**— atributo ou conjunto de atributos usados para localizar registros em um arquivo.
- ❖ Um **arquivo de índices** consiste de registros (chamados **entradas de índices**) da forma

search-key	pointer
------------	---------
- ❖ Arquivos de Índices são tipicamente menores que o arquivo original. Dois **tipos básicos de índices**:
 - **Índices Ordenados**: valores das chaves de procura são armazenadas ordenadamente.
 - **Índices Hash**: valores das chaves de procura são distribuídos uniformemente através de uma faixa de “buckets” utilizando uma “função hash”.



Métricas para Avaliação de Indexação

- ❖ Não existe uma **técnica de indexação** melhor, mas sim a **mais adequada** para aplicações **específicas** de BD.
- ❖ Técnicas de **Indexação** são **avaliadas** com base em:
 - **Tipos de acessos aceitos de forma eficiente.**
 - ◆ **Encontrar** registros com um **atributo de determinado valor** ou que se encontram dentro de uma **faixa de valores.**
 - **Tempo de Acesso**
 - ◆ tempo gasto para encontrar os dados desejados
 - **Tempo de Inserção**
 - ◆ tempo gasto para encontrar o local correto, incluir e atualizar a **estrutura de índice**
 - **Tempo de Exclusão**
 - ◆ tempo gasto para **encontrar o item, excluí-lo e atualizar a estrutura de índice**
 - **Sobrecarga de Espaço Adicional**

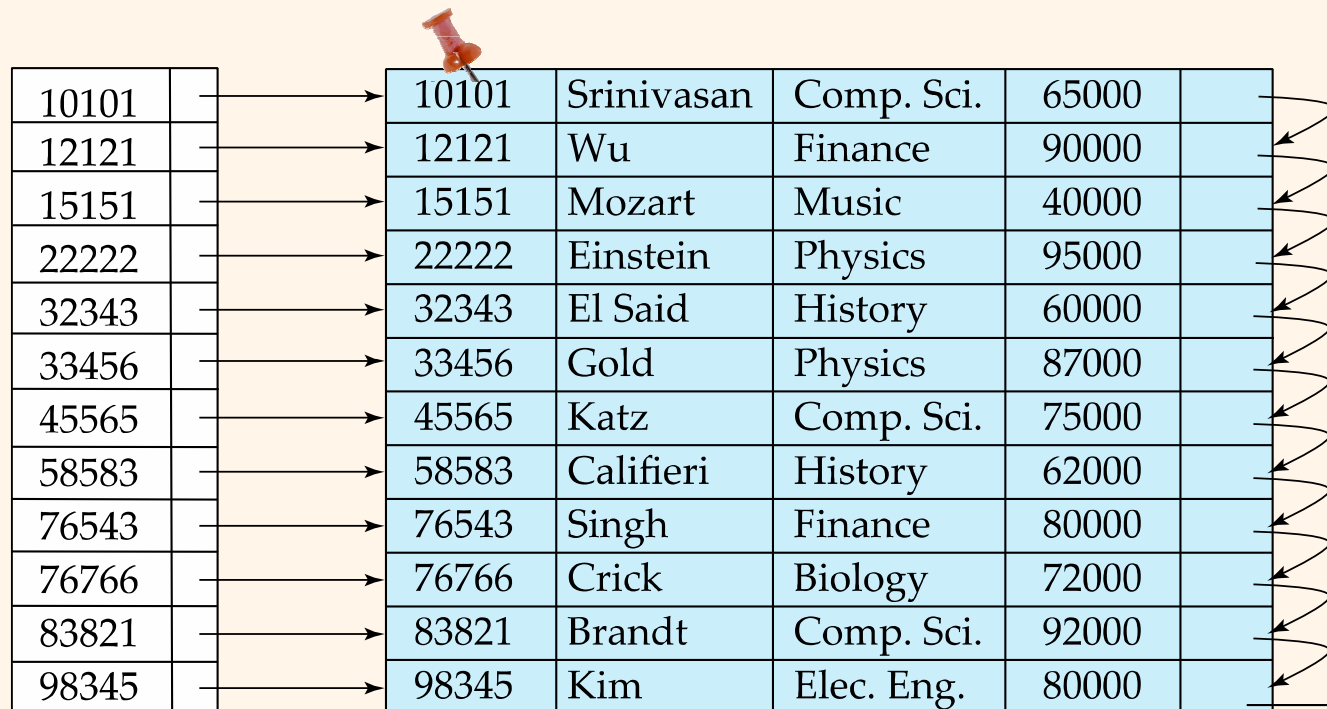


Índices Ordenados

- ❖ Em um **índice ordenado**, valores das chaves de procura são armazenados ordenadamente. Ex: catálogo de autor.
- ❖ **Índice Primário**: índice cuja chave de procura especifica a ordem sequencial do arquivo.
 - Também chamados **índices clustering**.
 - chave de procura é usualmente, mas **não** necessariamente, a **chave primária**.
- ❖ **Índice Secundário**: um índice o qual a chave de procura especifica uma ordem diferente da ordem sequencial do arquivo.
 - Também chamado de **índice não-clustering**.

Índices Ordenados Densos

- ❖ há uma entrada no índice para cada valor de chave que ocorre em um registro de dados
- ❖ Exemplo: Índice para o atributo ID com arquivo ordenado pelo identificador



10101	→	10101	Srinivasan	Comp. Sci.	65000
12121	→	12121	Wu	Finance	90000
15151	→	15151	Mozart	Music	40000
22222	→	22222	Einstein	Physics	95000
32343	→	32343	El Said	History	60000
33456	→	33456	Gold	Physics	87000
45565	→	45565	Katz	Comp. Sci.	75000
58583	→	58583	Califieri	History	62000
76543	→	76543	Singh	Finance	80000
76766	→	76766	Crick	Biology	72000
83821	→	83821	Brandt	Comp. Sci.	92000
98345	→	98345	Kim	Elec. Eng.	80000

Arquivos de Índices Densos

- ❖ **Exemplo:** Índice para o atributo dept_name, com arquivo ordenado pelo nome do departamento
- ❖ a entrada aponta para o primeiro registro que contém aquele valor de chave

Biology		76766	Crick	Biology	72000	
Comp. Sci.		10101	Srinivasan	Comp. Sci.	65000	
Elec. Eng.		45565	Katz	Comp. Sci.	75000	
Finance		83821	Brandt	Comp. Sci.	92000	
History		98345	Kim	Elec. Eng.	80000	
Music		12121	Wu	Finance	90000	
Physics		76543	Singh	Finance	80000	
		32343	El Said	History	60000	
		58583	Califieri	History	62000	
		15151	Mozart	Music	40000	
		22222	Einstein	Physics	95000	
		33465	Gold	Physics	87000	

Arquivos de Índices Esparsos

- ❖ Contém registros de índices para apenas alguns valores de chave de busca.

10101	
32343	
76766	

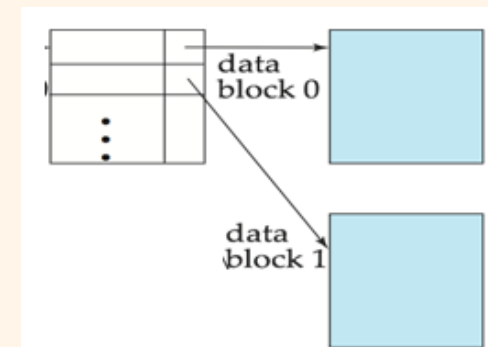
Para localizar um registro com valor de chave de busca K:

- Encontrar o registro de índice com o maior valor de chave, que seja $\leq K$.
- Varrer o arquivo sequencialmente, iniciando no registro apontado pela entrada de índice, até encontrarmos o registro desejado.

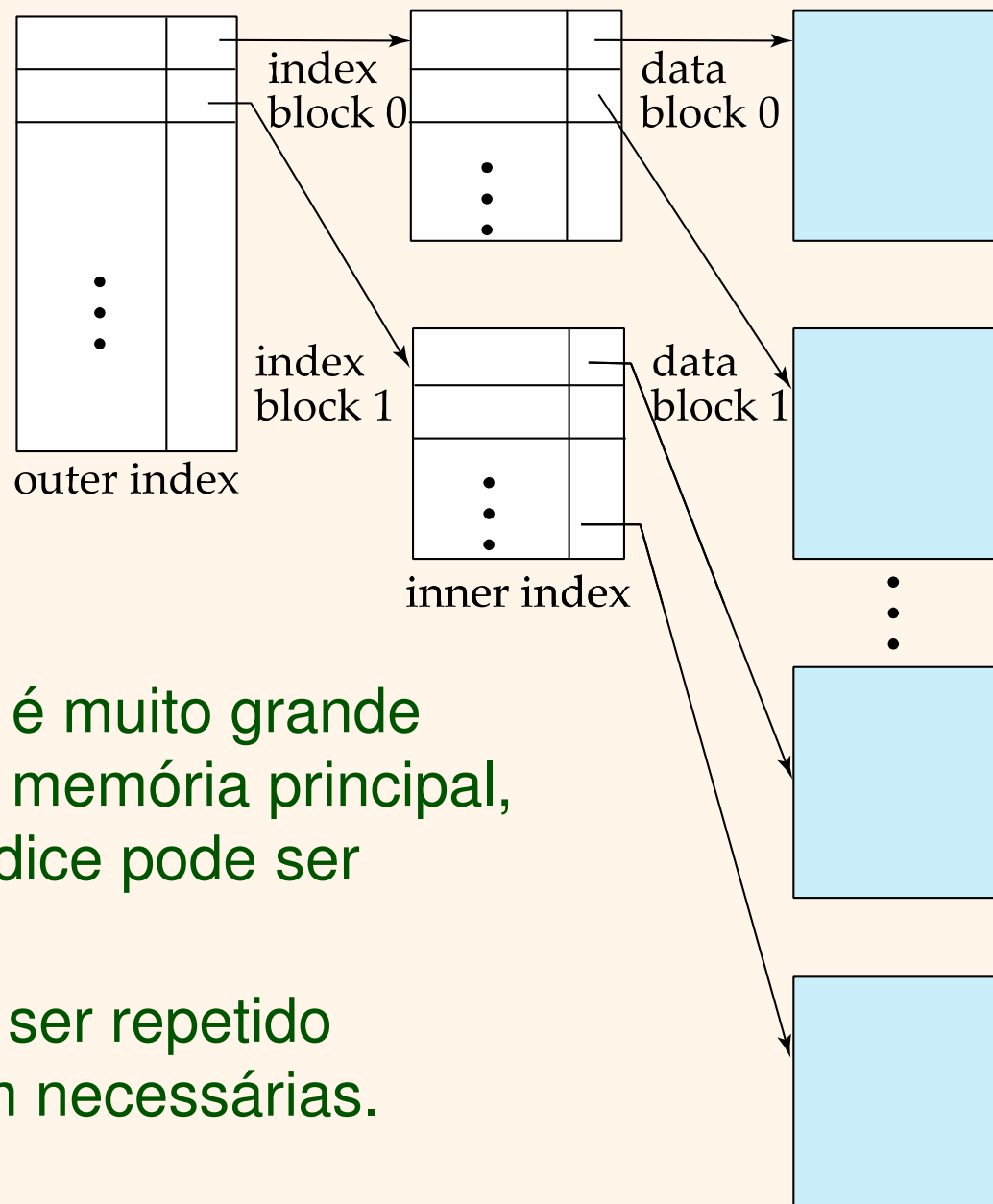
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Arquivos de Índices Esparsos

- ❖ Comparando índice esparso com denso
 - Esparsos geralmente são mais lentos para localizar registros e ocupam menos espaço e menos sobrecarga de manutenção para inserções e exclusões.
- ❖ O projetista deve escolher entre o tempo de acesso e a sobrecarga de espaço
 - Em BD o custo predominante é o tempo para trazer um bloco para a memória, uma vez na memória o tempo para varrer o bloco inteiro é desprezível
- ❖ Uma boa solução: índice esparso com uma entrada de índice para cada bloco em um arquivo.



Índices de Níveis Múltiplos



- ❖ Se o índice primário é muito grande para ser mantido na memória principal, um outro nível de índice pode ser criado.
- ❖ Este processo pode ser repetido quantas vezes forem necessárias.



Índices de Níveis Múltiplos

- ❖ Problema dos índices multiníveis:
 - índices são arquivos fisicamente ordenados, portanto, ineficientes na inserção e remoção
- Solução:
 - Deixar algum espaço em cada um dos blocos para inserção de novas entradas
 - Estruturas de dados: Árvores B e suas variações.

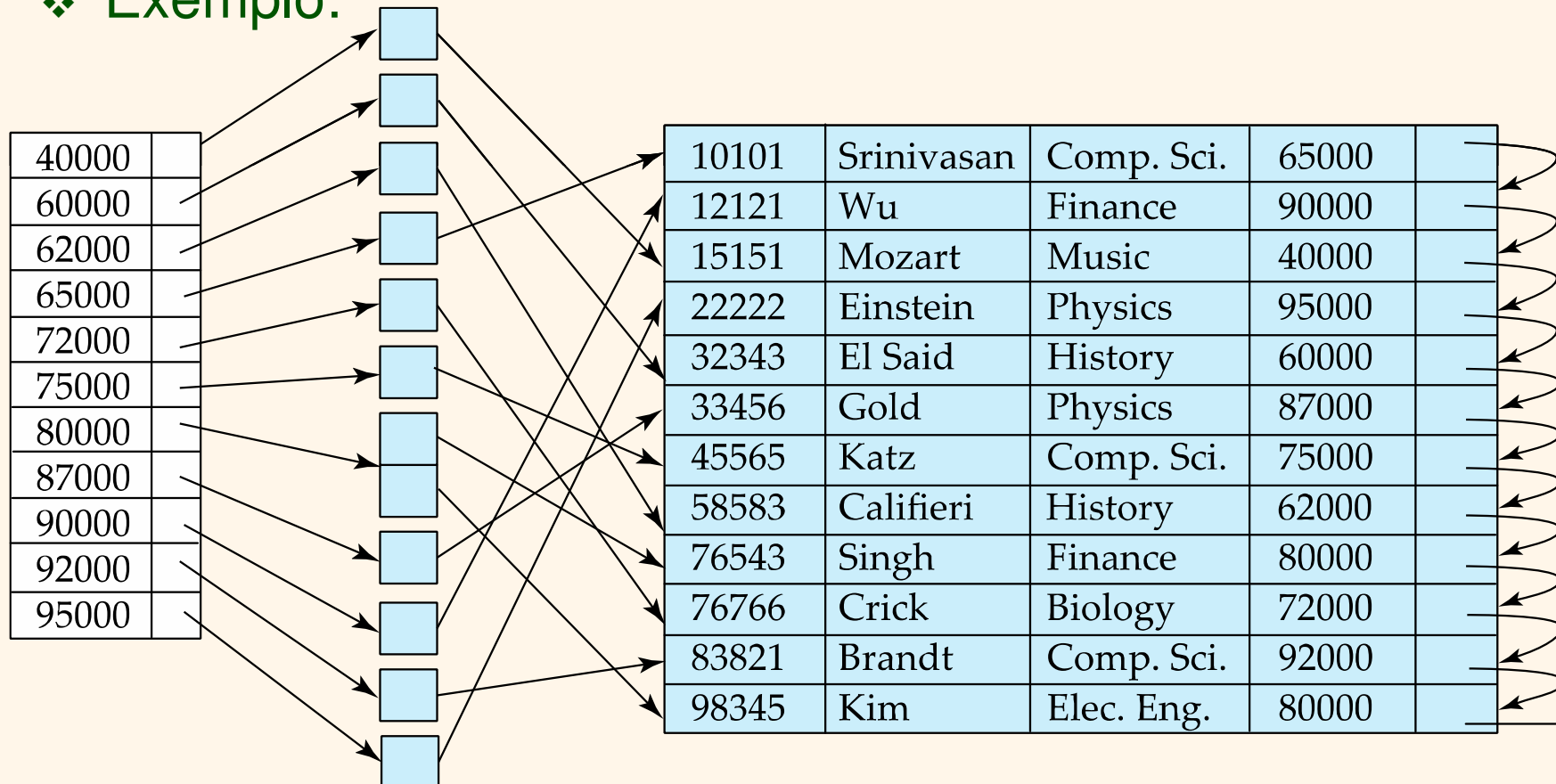


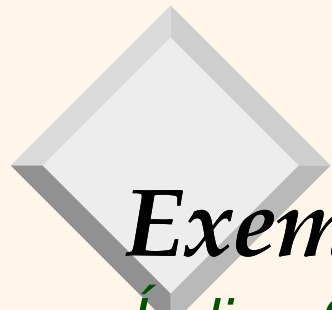
Índices Secundários

- ❖ Índice cuja chave não é aquela da ordem sequencial do arquivo
- ❖ Organização:
 - há uma entrada no índice para cada valor de chave que ocorre em um registro de dados
 - a entrada aponta para **todos** os registros que contêm aquele valor de chave x
 - pode-se usar um nível indireto adicional, como na figura do slide seguinte

Índices Secundários

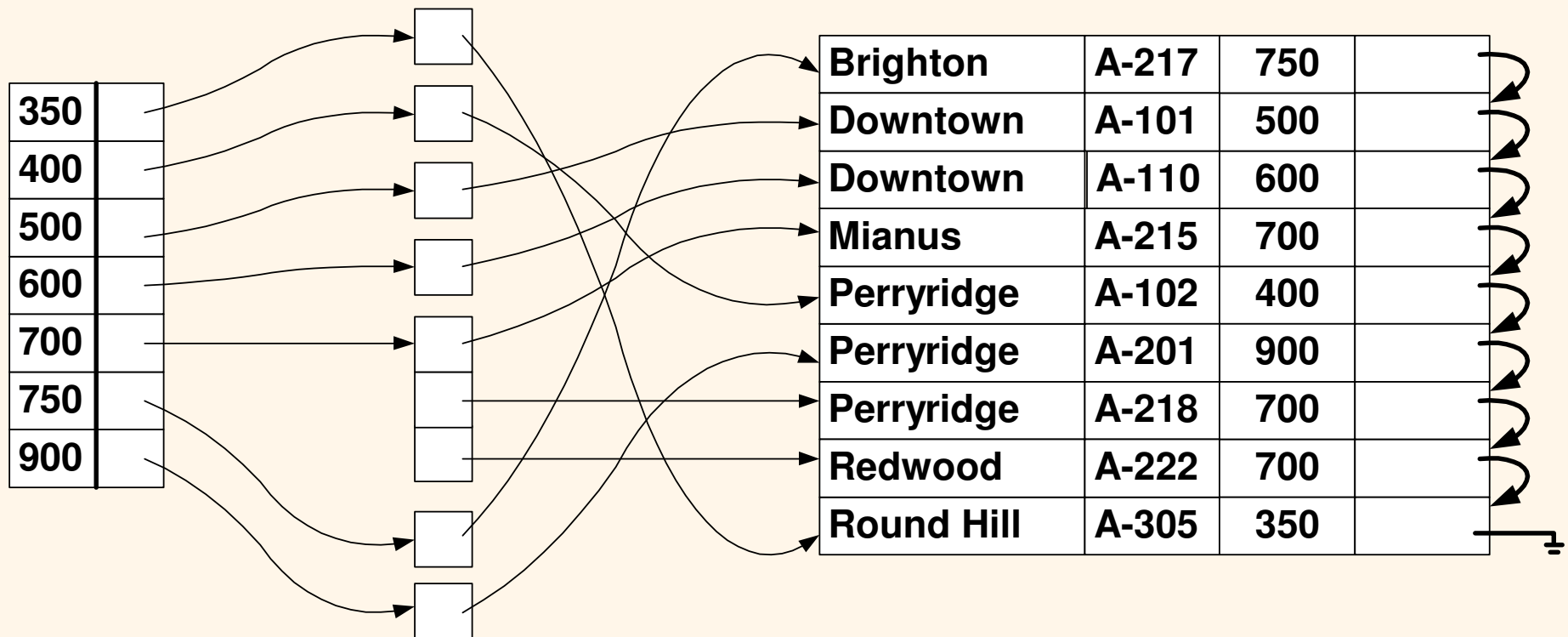
- ❖ Cada registro do índice aponta para um *bucket* que contém ponteiros para todos os registros do arquivos com o valor de chave de procura em particular.
- ❖ Exemplo:

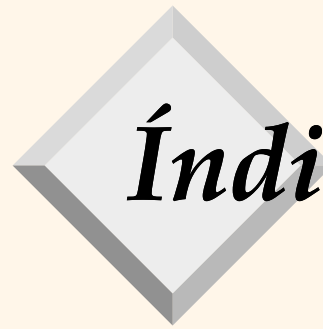




Exemplo:

Índice Secundário para o Campo Saldo no Arquivo Conta





Índices Primário e Secundário

- ❖ A utilização de índices oferecem **benefícios** para busca de registros.
- ❖ Quando um arquivo é modificado, cada índice para o arquivo deve ser atualizado, o que impõem uma certa **sobrecarga** na atualização do banco de dados.
- ❖ Busca sequencial utilizando índice primário é eficiente, mas utilizando índice secundário não é (a leitura de cada um dos registros provavelmente irá requerer a leitura de um novo bloco do disco).



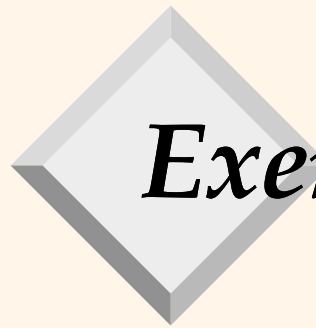
Exercícios

1. Quando é preferível usar um índice denso em lugar de um índice esparsos? Justifique sua resposta.
2. Como os índices aceleram o processamento de consultas, por que eles não devem ser mantidos sobre diversas chaves de procura? Liste o máximo de razões possíveis.
3. Qual a diferença entre um índice primário e um secundário?
4. De forma geral, é possível ter dois índices primários na mesma relação para chaves de procura diferentes? Justifique sua resposta.

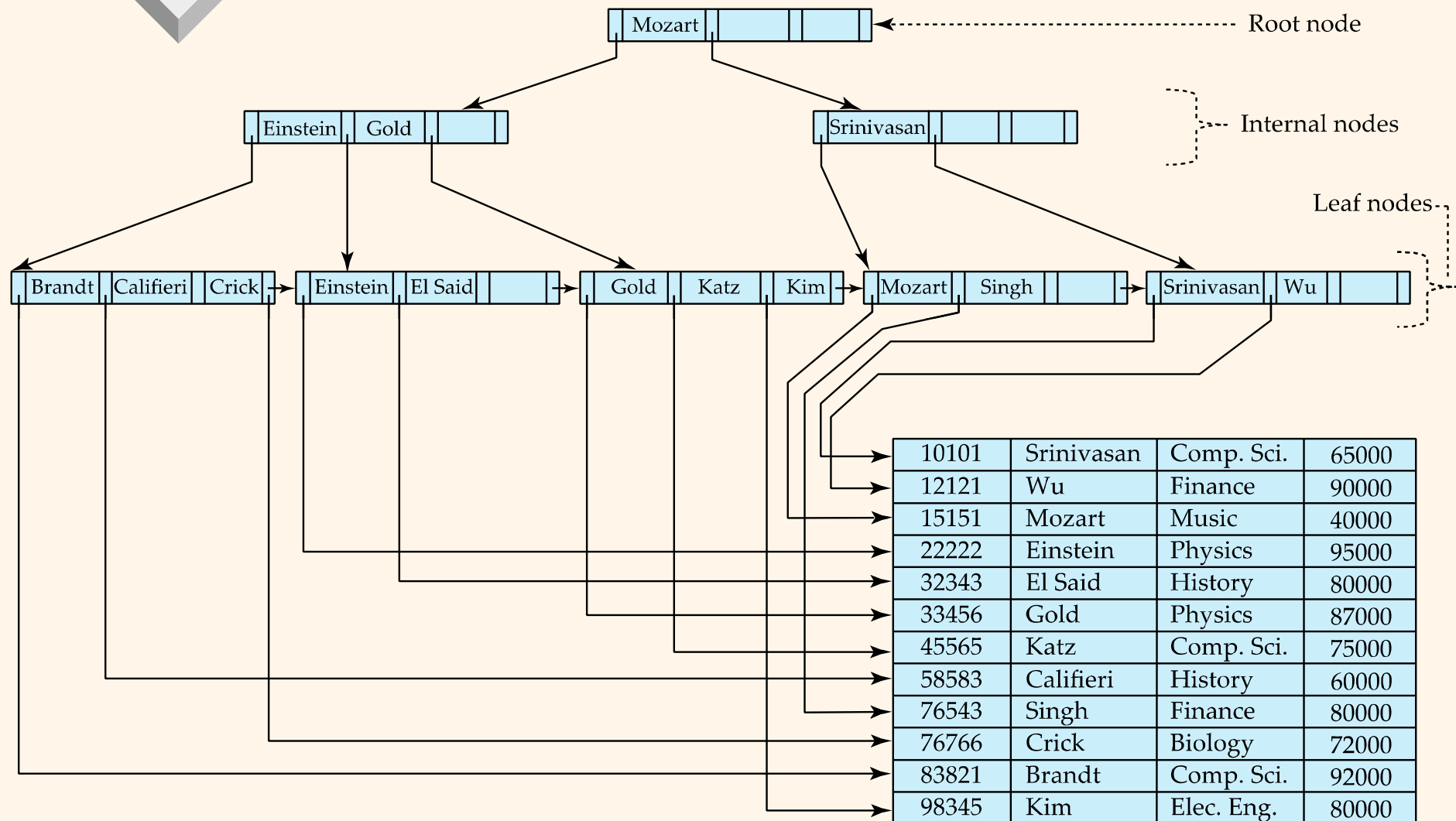


Arquivos de Índices Árvore-B+

- ❖ Nos índices sequenciais o **desempenho** se degrada conforme o arquivo cresce (necessárias reorganizações periódicas do arquivo).
- ❖ Índices **árvore-B+** são uma alternativa para manter eficiência a despeito da inserção e remoção de dados.
 - reorganizações no arquivo não são necessárias para manter o desempenho.
- ❖ Exige espaço adicional, porém **aceitáveis** devido aos **benefícios** de desempenho da estrutura árvore-B+, fazendo desta estrutura uma **das mais utilizadas**.



Exemplo: Árvore-B+





Estrutura de uma Árvore-B+

❖ Um Nó Típico



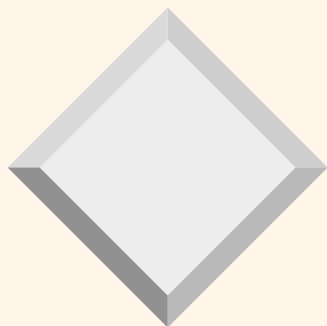
- K_i são os valores de chave de procura.
 - P_i são ponteiros para os **nó-filhos** (se nós **não folhas**) ou ponteiros para **registros** ou para **bucket** de ponteiros (se nós **folhas**).
- ❖ Os valores da **chave de procura** dentro de um nó são mantidos **ordenados**.

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

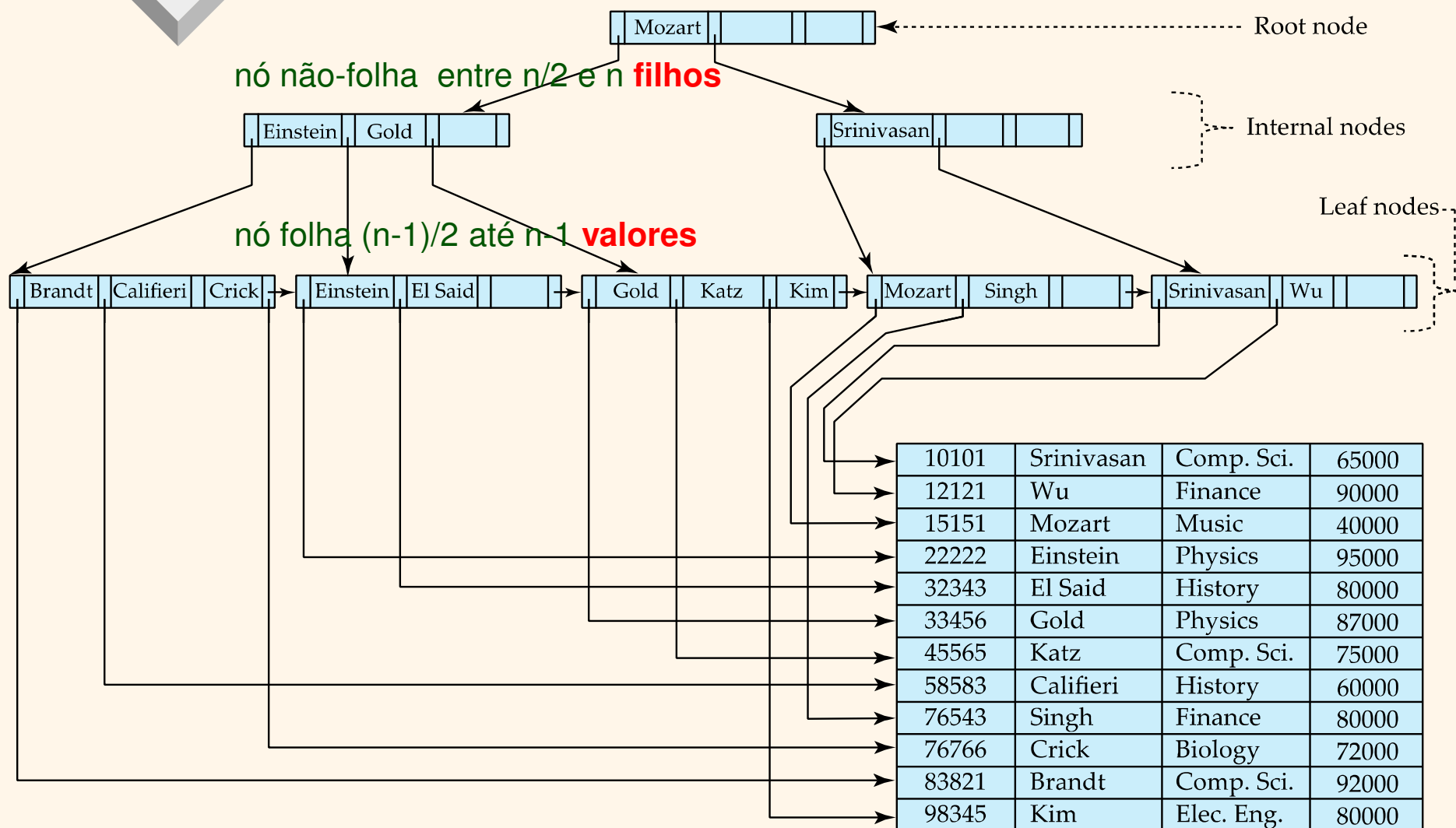


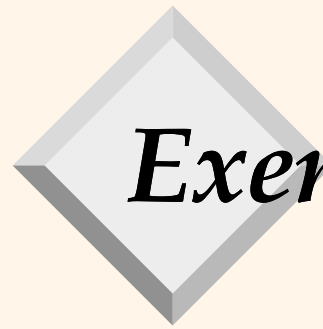
Arquivos de Índices Árvore-B+

- ❖ Um índice árvore-B+ tem a forma de uma árvore balanceada, com as seguintes propriedades:
 - Todos os caminhos a partir da raiz até uma de suas folhas são do mesmo comprimento.
 - Cada nó não-folha da árvore possui entre $n/2$ e n filhos, em que n é fixo para uma árvore em particular.
 - um nó folha pode conter entre $(n-1)/2$ até $n-1$ valores.
- ❖ Casos Especiais:
 - se a raiz **não é** um nó folha, **deve ter** no mínimo dois filhos.
 - se a raiz **é** um nó folha (não há outros nós na árvore), então pode conter entre 0 e $n-1$ valores.

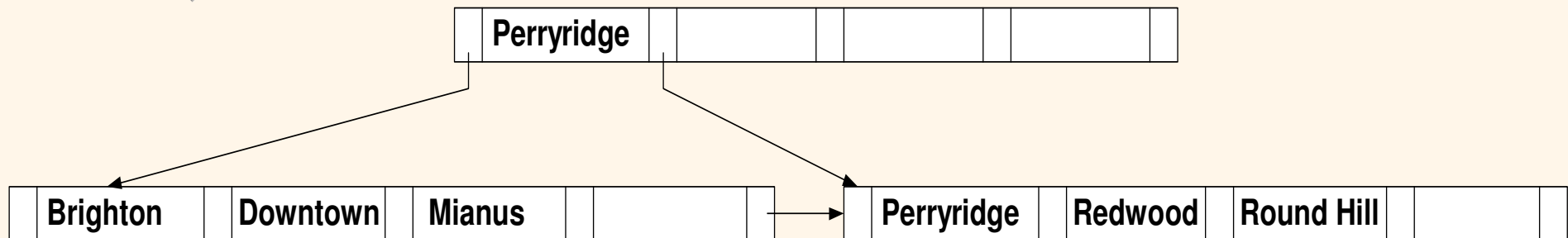


se a raiz **não é** um nó folha, deve ter no mínimo dois filhos





Exemplo Árvore-B+ com $n=5$

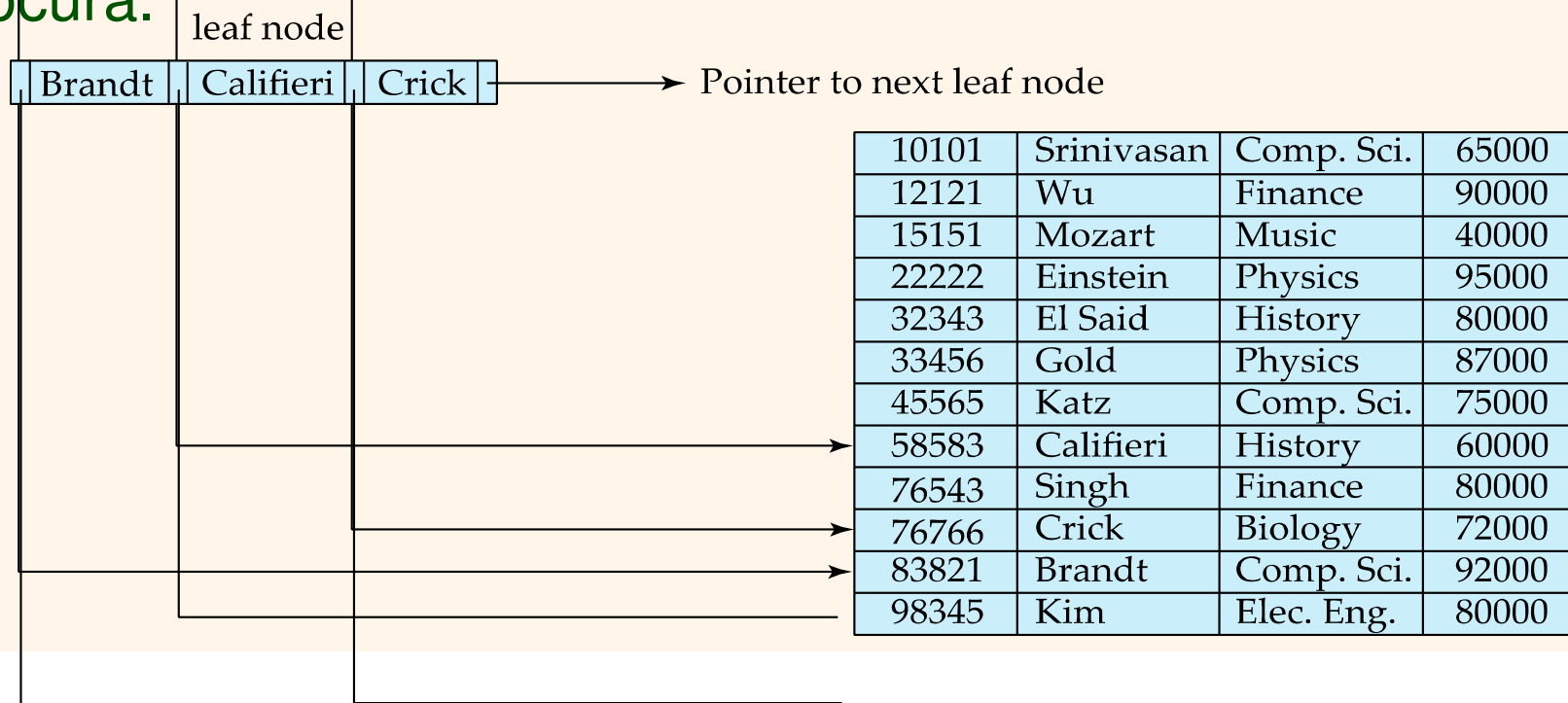


B⁺-tree for account file ($n = 5$)

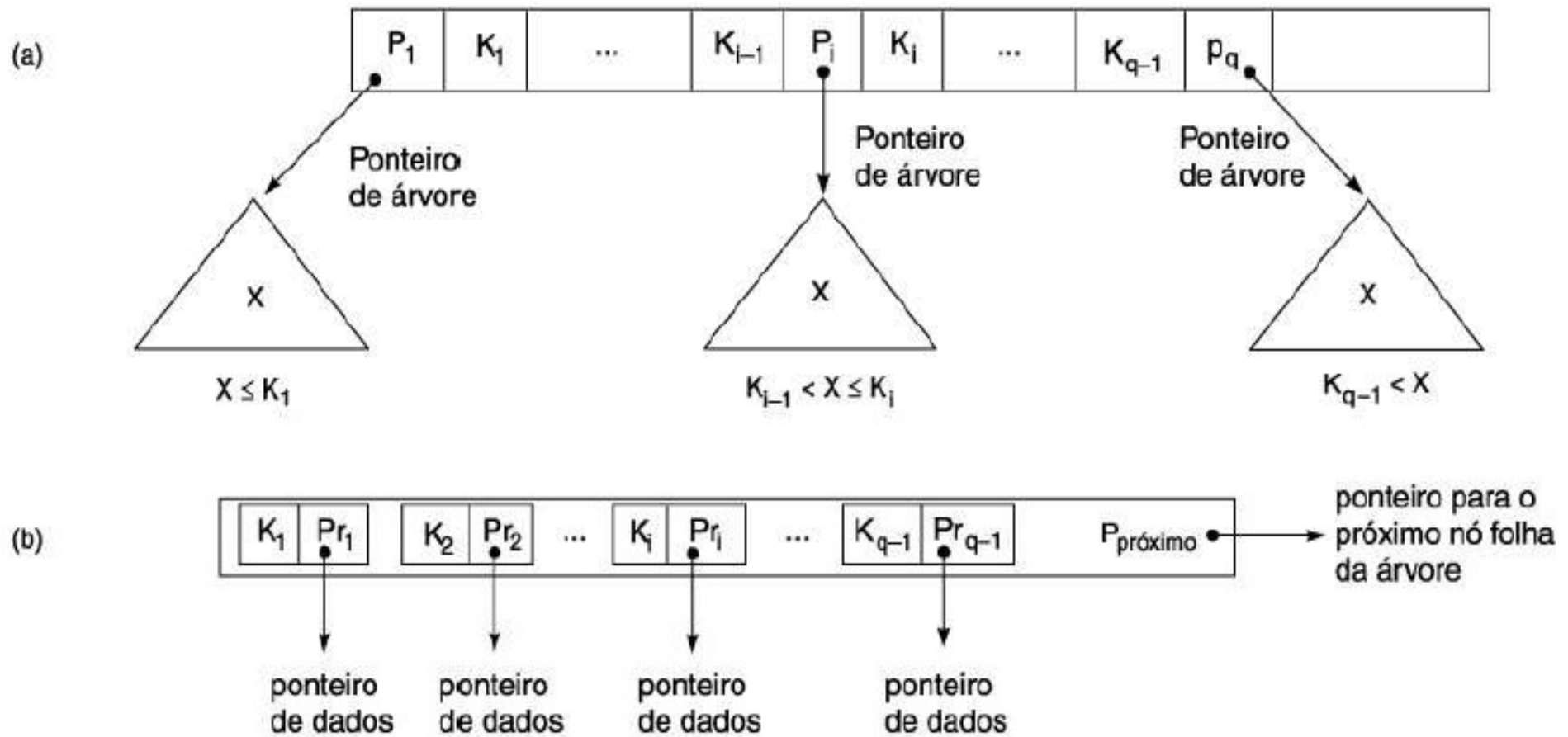
- ❖ O nó raiz deve ter ao menos 2 filhos.
- ❖ Nós não-folhas, **exceto a raiz**, devem ter entre 3 e 5 **filhos** ($n/2$ e n).
- ❖ Os nós folhas devem ter entre 2 e 4 **valores** ($(n-1)/2$ e $n-1$).

Nós Folhas - Propriedades:

- ❖ Para $i=1, 2, \dots, n-1$, o **ponteiro P_i** aponta para um **registro do arquivo** com chave de procura de valor K_i
- ❖ Se L_i, L_j são nós **folhas** e $i < j$, então todos os valores de chave de procura em L_i são menores do que todos os valores de chave de procura em L_j
- ❖ O ponteiro **P_n** encadeia os nós **folhas** na ordem da chave de procura.



Árvores-B+ - Propriedades



Os nós de uma árvore-B+. (a) Nó interno de uma árvore-B+ com $q - 1$ valores de busca. (b) Nó folha de uma árvore-B+ com $q - 1$ valores de busca e $q - 1$ ponteiros de dados.



Observações sobre Árvores-B+

- ❖ Em uma árvore-B+, um **nó** é **grande**, tipicamente do tamanho de um **bloco de disco**,
 - mas **não necessariamente** um bloco lógico de dados corresponde à um bloco físico.
- ❖ Árvores-B+ contêm um número **relativamente pequeno de níveis**, portanto as consultas apresentam eficiência.
- ❖ **Inserções e remoções** no arquivo principal são executadas **eficientemente**, bem como o índice pode ser reestruturado em um intervalo de tempo logarítmico.


Consultas em Árvores-B+

- ❖ Encontrar registros com um valor de chave de procura k :
 - Inicie no nó raiz.
 - ◆ Examine procurando o menor valor de chave de procura que seja maior k .
 - ◆ Caso esse valor exista, suponhamos que seja K_i . Então seguimos o ponteiro P_i até o nó filho.
 - ◆ Se tivermos m ponteiros no nó e $k \geq K_{m-1}$, então seguimos P_m até o nó filho.
 - Se nó atual não é folha, repita o procedimento acima e siga o ponteiro correspondente.
 - Eventualmente, alcançamos um nó folha.
 - ◆ Se $K_i = k$, então o ponteiro P_i levará ao registro desejado ou para um bucket.
 - ◆ Caso contrário, não há um valor de registro correspondente ao valor da chave de procura k .



Atualizações em Árvores-B+

- ❖ A inserção e remoção são mais complicadas do que a busca:
 - É necessário **dividir o nó** que se torne muito **grande** como resultado de uma inserção
 - **Combinar nós** se um tornar-se muito **pequeno** (menor que $n/2$ ponteiros)
 - Além disso, temos que garantir que o **balanceamento** seja preservado

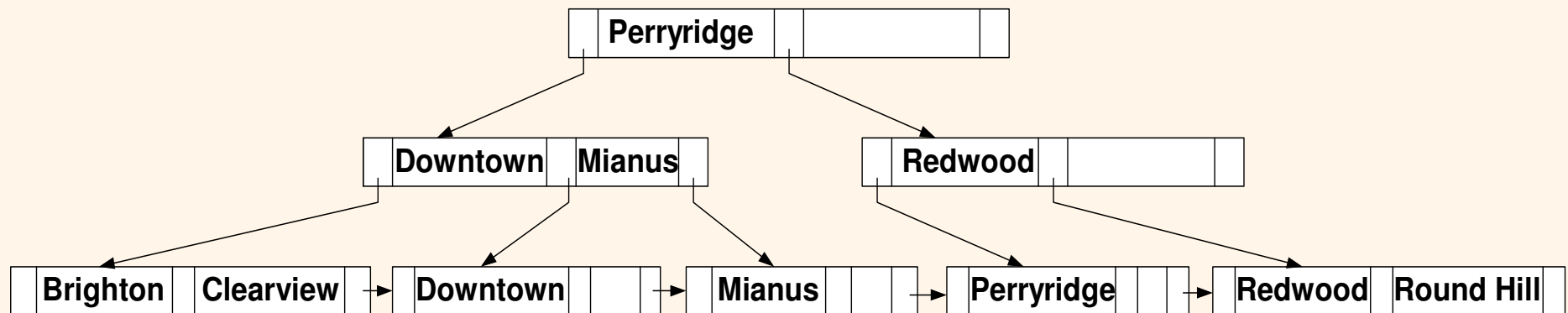
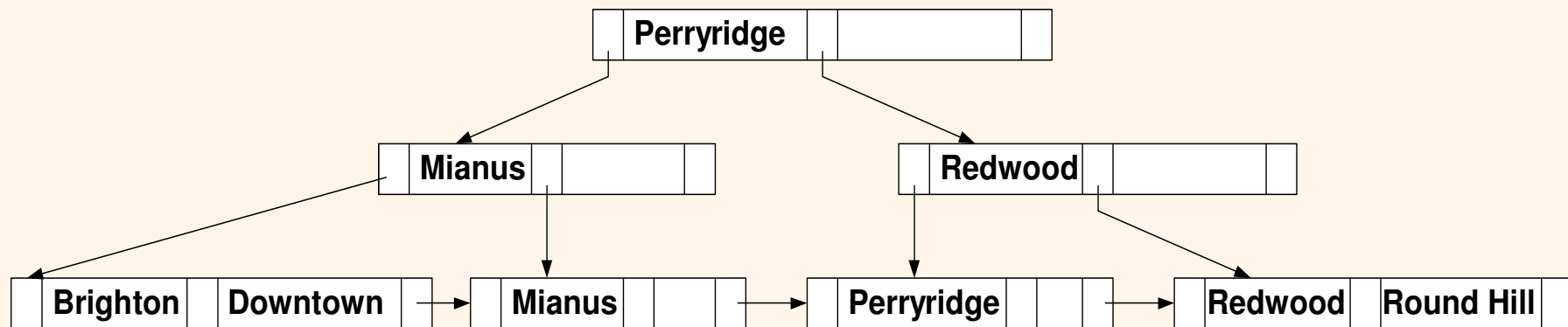


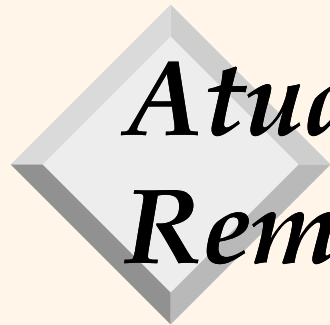
Árvores-B+: Inserção e Remoção

- ❖ Vamos considerar que os nós nunca ficam muito grandes ou pequenos.
- ❖ **Inserção**
 - Utilizando a técnica de busca encontramos o nó folha o qual o valor da chave de procura deveria aparecer;
 - Se o valor da chave de procura já aparece no nó folha, adicionamos um novo registro ao arquivo e, se necessário, um ponteiro ao bucket.
 - Se o valor da chave de procura não aparece, então adicionamos um novo registro ao arquivo e, se necessário, criamos um novo bucket com o ponteiro apropriado. Então:
 - ◆ Se há espaço no nó folha, inserimos o par (valor de chave de procura, ponteiro para o registro/bucket), na posição adequada.
 - ◆ Se não há espaço no nó folha, dividimos este nó em dois e inserimos o par (valor de chave de procura, ponteiro para o registro/bucket) como ilustra o exemplo a seguir.

Árvores- B^+ : Inserção

- ❖ Árvore- B^+ antes e depois de uma inserção (Clearview) para $n=3$

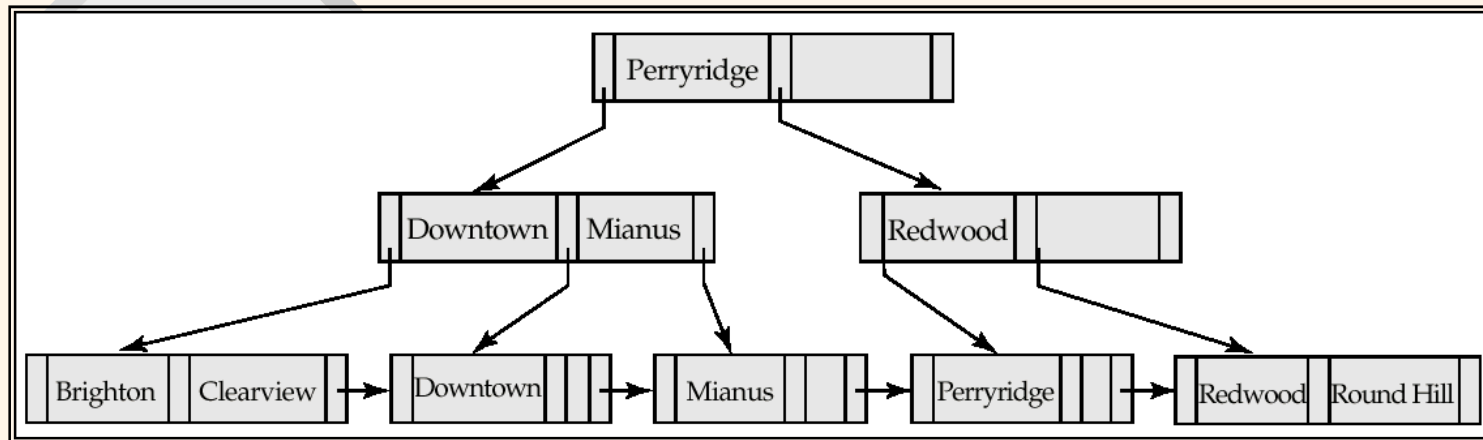




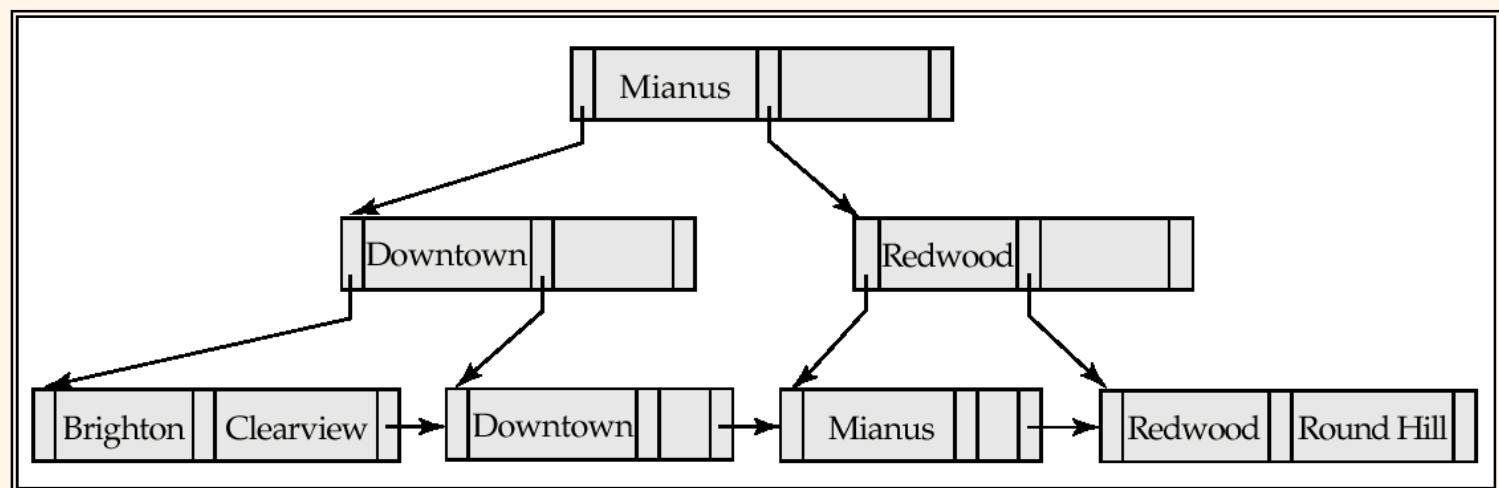
Atualizações em Árvores-B+: *Remoção*

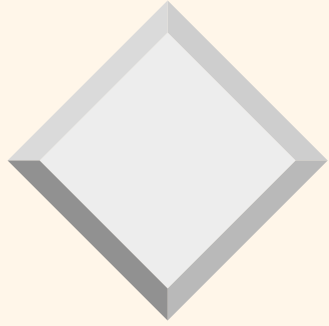
- ❖ Utilizando a técnica de busca, encontramos o registro a ser removido e o **excluímos do arquivo e do bucket** (caso esteja presente).
- ❖ Removemos (valor da chave de procura, ponteiro) do nó folha se não houver bucket associado àquele valor de chave de procura ou se o bucket tornar-se vazio como resultado da remoção.

Exemplo de Remoção - Árvores-B+



Resultado da remoção de
“Perryidge” de contas



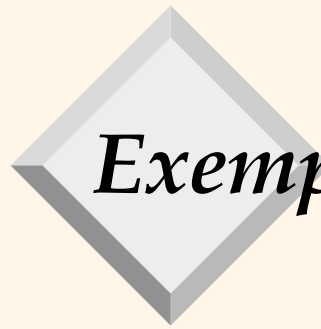


Hashing



Hashing Estático

- ❖ Em uma organização de arquivos **hashing**, obtemos diretamente o **endereço do bloco** de disco que contém um registro desejado por meio da **aplicação de uma função** sobre o **valor da chave** de procura do registro.
- ❖ Um bucket (balde) é uma unidade de armazenamento que contém **um ou mais registros** (geralmente, é um bloco de disco).
- ❖ Seja K o conjunto de todos os valores de chave e B todos os endereços de buckets.
 - Uma função hash h é uma função de K para B , isto é, uma função do conjunto de todos os valores de chaves de procura K para o conjunto de todos os endereços de buckets B .
- ❖ A função Hash é utilizada na localização, manipulação, inserção e também na deleção.



Exemplo de organização de arquivo de hash

- Organização de arquivo de hash do arquivo conta, usando nome-agência como chave
 - → ver figura no slide seguinte
- Existem 10 buckets
- A representação binária do i -ésimo caractere é considerada como o inteiro i
- A maioria das funções Hash realizam cálculos com a representação binária das chaves de pesquisa.
 - Ex. Uma função de hash retorna a soma das representações binárias dos caracteres módulo 10

Exemplo de Organização de Arquivo com Hash

Por exemplo:

$$h(\text{Perryridge}) = 5$$

$$h(\text{Round Hill}) = 3$$

$$h(\text{Brighton}) = 3$$

Bucket 0

--	--	--

Bucket 1

--	--	--

Bucket 2

--	--	--

Bucket 3

Brighton	A-217	750
Round Hill	A-305	350

Bucket 4

Redwood	A-222	700

Bucket 5

Perryridge	A-102	400
Perryridge	A-201	900
Perryridge	A-218	700

Bucket 6

--	--	--

Bucket 7

Mianus	A-215	700

Bucket 8

Downtown	A-101	500
Downtown	A-110	600

Bucket 9

--	--	--

Funções Hash

- ❖ Para **localizar** um valor de chave de procura k , computamos $h(k)$, que dá o endereço do bucket para aquele registro;
- ❖ Para **inserir** um registro com chave de procura k , computamos $h(k)$, que dá o endereço do bucket para aquele registro, se há espaço o registro é armazenado no bucket informado.
- ❖ Para **remoção**, calculamos $h(k)$, procuramos o bucket e removemos o registro.
- ❖ Registros com **valores de chaves** de pesquisa **diferentes** podem apontar para o **mesmo bucket**; nesse caso, o bucket inteiro deve ser pesquisado **seqüencialmente** para localizar o registro.

Funções Hash

- ❖ A **piores** funções Hash mapeiam todos os valores das chaves de procura para o mesmo bucket; isso torna o tempo de acesso proporcional à quantidade de valores das chaves de procura no arquivo.
- ❖ Uma função Hash **ideal** é **uniforme**, i.e, a cada bucket é atribuído o mesmo número de valores de chaves de procura do conjunto total de valores possíveis.
- ❖ Uma função Hash **ideal** é **aleatória**, logo, cada bucket terá o mesmo número de registros atribuídos a ele, independente da distribuição real dos valores de chave de busca no arquivo.



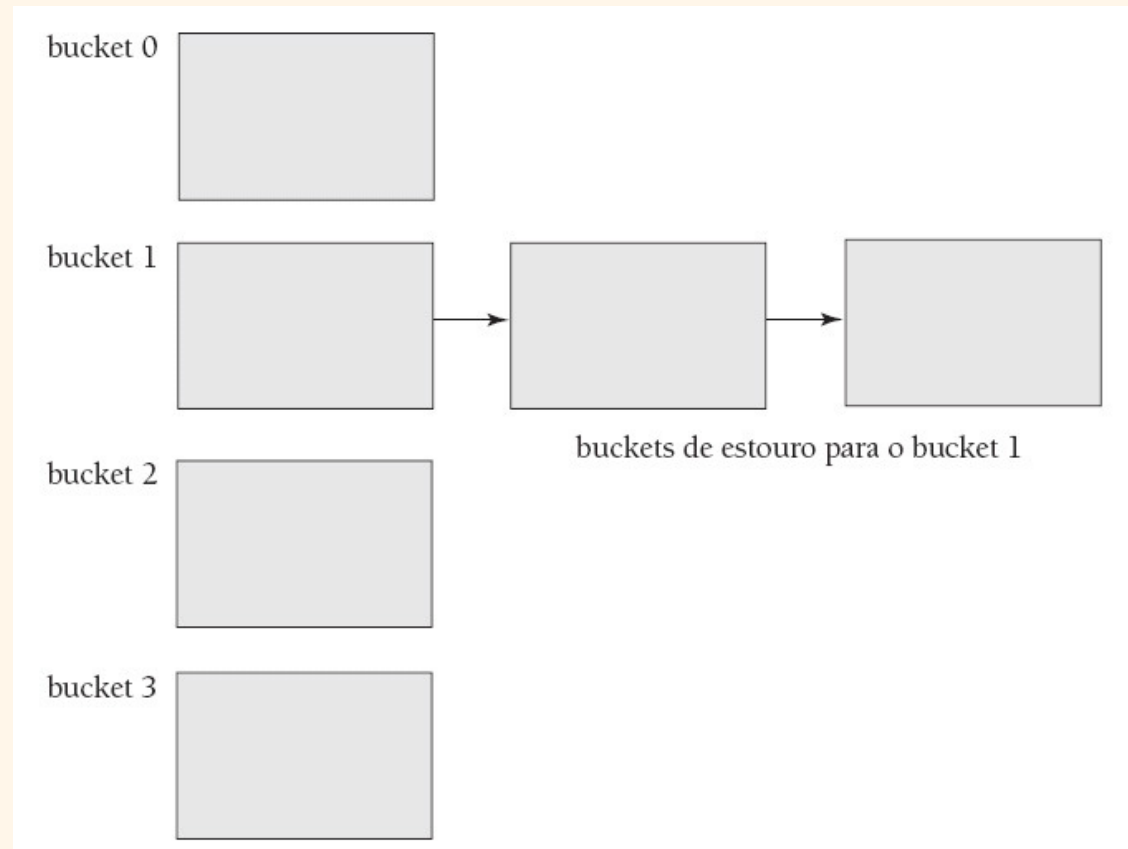
Tratamento de estouros de bucket

- ❖ O estouro de balde pode ocorrer devido a
 - Buckets insuficientes
 - Distorção na distribuição de registros, devido:
 - ▶ a vários registros possuírem o mesmo valor de chave de busca
 - ▶ a função de hash escolhida produz a distribuição não uniforme de valores de chave
- ❖ Embora a probabilidade de estouro de bucket possa ser reduzida, ela não pode ser eliminada; ele é tratado pelo uso de bucket de estouro



Tratamento de estouros de bucket

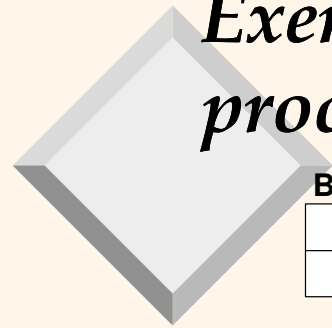
- ❖ Encadeamento de estouro – os baldes de estouro são encadeados em uma lista interligada.
- ❖ O esquema é chamado de hashing fechado.
 - Uma alternativa, chamada *hashing* aberto, que não usa baldes de estouro, não é apropriada para aplicações de banco de dados.



Índices Hash

- ❖ Além de ser utilizado na organização de arquivos, a técnica de Hashing pode ser utilizada também na criação de **estruturas de índices**.
- ❖ Um índice Hash organiza as chaves de procura, com seus respectivos ponteiros associados, numa estrutura de arquivo com hash.
- ❖ Índices Hash **são sempre índices secundários**.

Exemplo de Índice Hash com chave de procura número_conta



Bucket 0

Bucket 1

A-215	
A-305	

Bucket 2

A-101	
A-110	

Bucket 3

A-217	
A-102	

A-201	

Bucket 4

A-218	

Bucket 5

Bucket 6

A-222	

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round Hill	A-305	350	



Deficiências do Hashing Estático

- ❖ No Hashing Estático, a função h mapeia valores de chaves de procura num valor fixo de endereços por bucket. Porém:
 - BD crescem com o passar do tempo. Se o número inicial de buckets for muito pequeno, o desempenho degradará devido a muitos estouros.
 - Se um grande número de buckets forem alocados (previsão de crescimento do BD), uma quantidade de espaço significativa será desperdiçada inicialmente. Além disso, se a base não crescer como esperado, ou diminuir, tem-se espaço perdido.
 - Outra alternativa é periodicamente reorganizar a estrutura hash em reposta ao crescimento da base, mas isso é muito dispendioso.
- ❖ Esses problemas podem ser evitados usando-se técnicas que permitem modificar dinamicamente o número de buckets.

Hashing Dinâmico

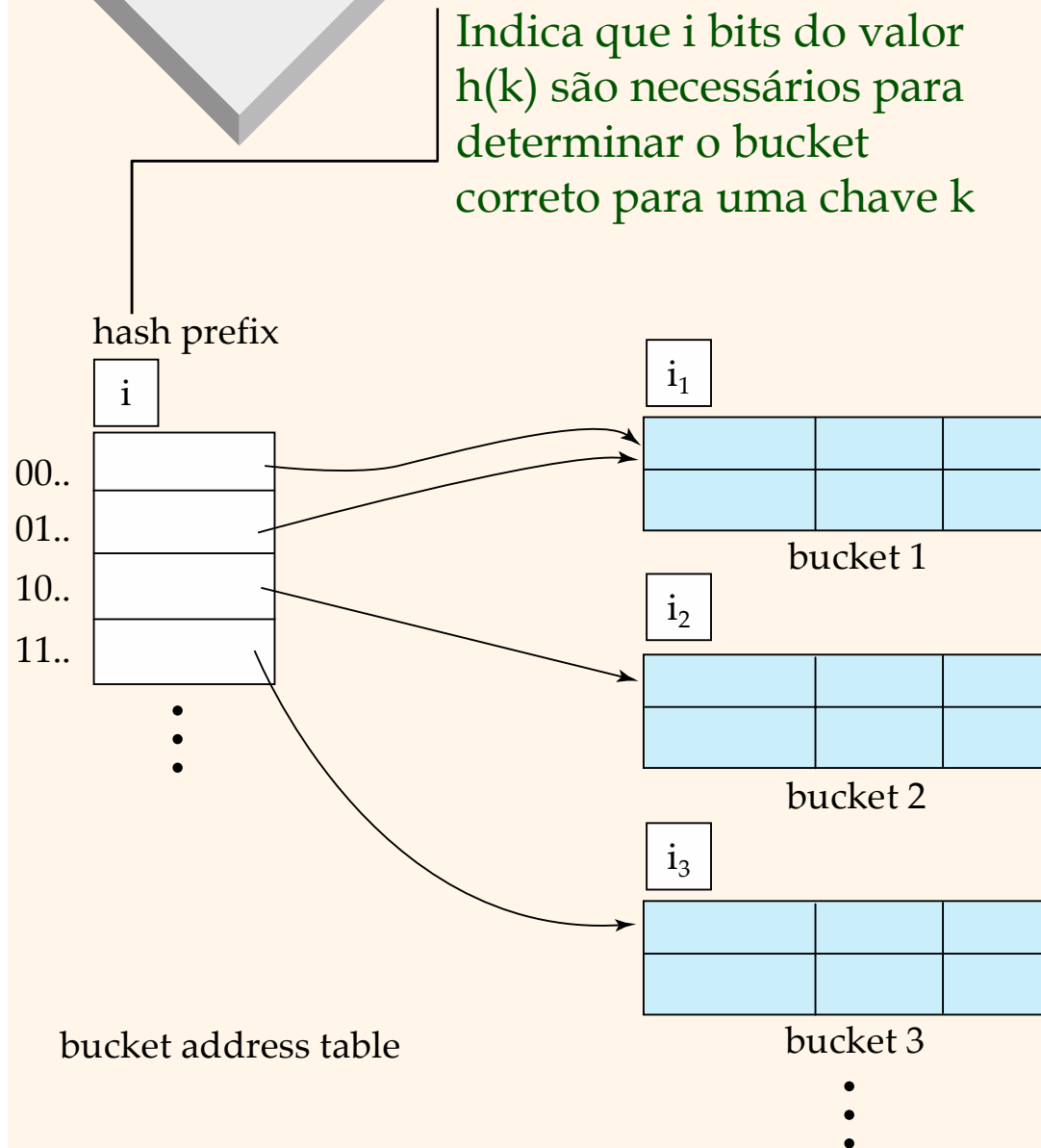
- ❖ Bom para BD que possuem tamanho variável
- ❖ Permitem que a função hash seja modificada dinamicamente
- ❖ **Hash Expansível** – uma forma de hash dinâmico
 - Trata as mudanças no tamanho do BD por meio da divisão e fusão de buckets;
 - A reorganização é feita em **um bucket por vez**, assim o overhead sobre o **desempenho é aceitável**;
 - Escolhemos uma **função hash h** com as propriedades desejadas e que gere valores dentro de uma **faixa** relativamente **grande** - inteiros binários de b bits, sendo b tipicamente 32.

Hashing Dinâmico

❖ **Hash Expansível** (continuação)

- Os buckets são **criados por demanda**, conforme registros são inseridos nos arquivos – inicialmente usa-se i bits, sendo $0 \leq i \leq 32$
- Inicia-se **com $i=0$** , seu valor aumenta e diminui conforme o tamanho do BD.
- O número efetivo de buckets é $< 2^i$, e sempre muda dinamicamente devido à junção ou divisão de buckets.

Estrutura de Hash Expansível




- Localizar um bucket com chave K_j :
 - Calcule $h(K_j) = X$
 - Use os primeiros i bits de X e siga o ponteiro para bucket apropriado
- Para inserir um registro com valor de chave de busca K_j
 - siga o mesmo procedimento da pesquisa e localize o bucket, digamos j .
 - se houver espaço no balde j , insira o registro no balde.
 - senão, o balde precisa ser dividido e a inserção tentada novamente (próximo slide.)



Atualizações na Estrutura de Hash Expansível

- ❖ Se $i > i_j$ (mais do que um ponteiro para o bucket j)
 - aloca-se um novo bucket z , e atribui-se i_{j+1} a i_j e i_z .
 - fazer com que as entradas da segunda metade que apontavam para j apontem para z
 - remova e insira novamente cada registro do j .
 - recalcule o novo bucket K_j e insira o registro no bucket (outras divisões serão necessárias de o bucket continuar cheio)
- ❖ Se $i = i_j$ (somente um ponteiro no bucket j)
 - incrementar i e duplicar o tamanho da tabela de endereços do bucket.
 - Atualizar cada entrada na tabela por duas entradas que apontam para o mesmo bucket.
 - Recalcular uma nova entrada para K_j . Agora $i > i_j$, então use o algoritmo visto anteriormente.



Atualizações na Estrutura de Hash Expansível (Cont.)

- ❖ Ao inserir um valor, se o bucket continua cheio após várias divisões (ou seja, i alcança o limite b), crie um bucket de overflow ao invés de dividir a tabela de endereços mais vezes.
- ❖ Para deletar um valor de chave, localize-o e no respectivo bucket e remova-o.
- ❖ O próprio bucket pode ser apagado caso se torne vazio (com as devidas atualizações na tabela de endereços de buckets).
- ❖ Junção de buckets e decréscimo do tamanho da tabela de endereços também é possível.



Uso da Estrutura de Hash Expansível: Exemplo

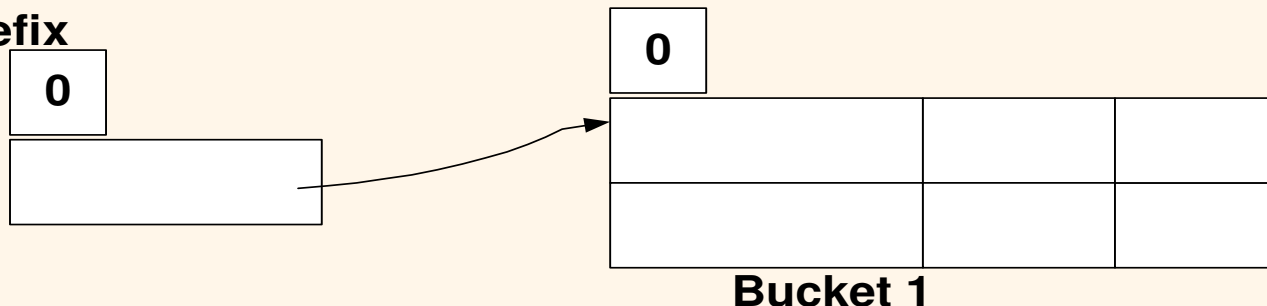
nome_agência

$h(\text{nome_agência})$ - valor hash 32 bits

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

Vamos supor que o arquivo esta inicialmente vazio:

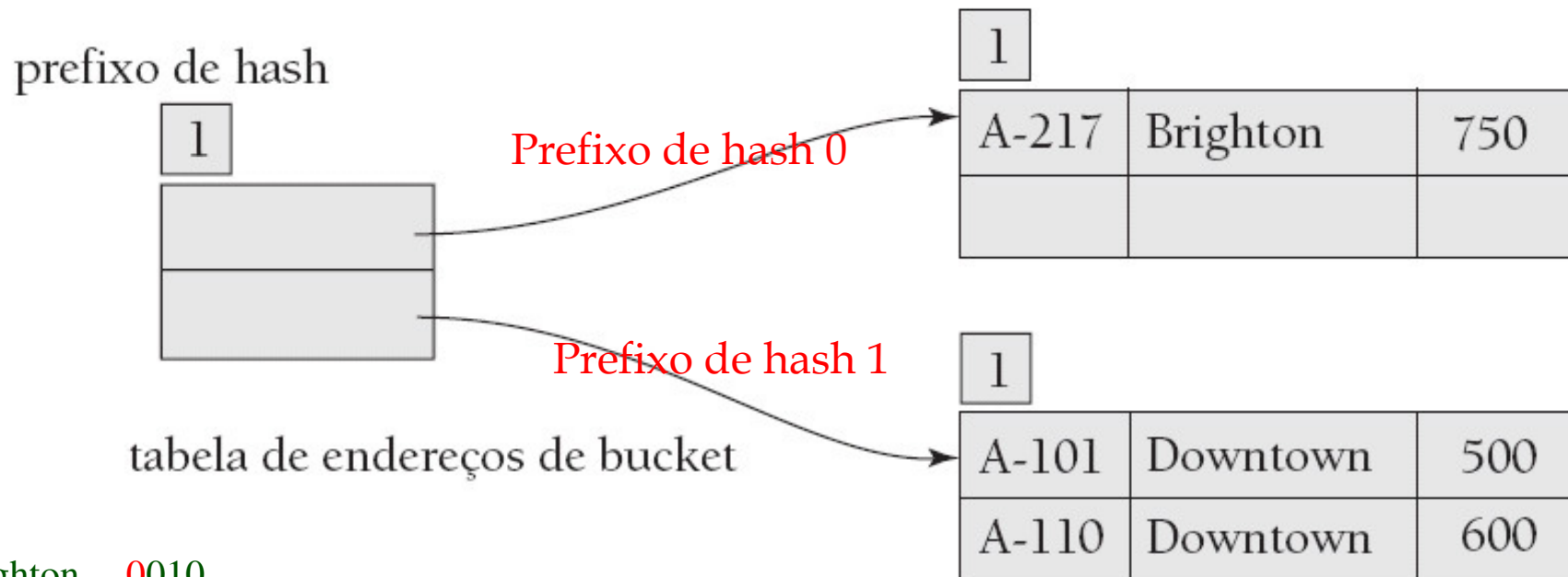
Hash prefix



bucket adress table

Para simplificar, vamos supor que um bucket pode manter apenas 2 registros

- Estrutura de hash após inserção de um registro de **Brighton** e dois de **Downtown**

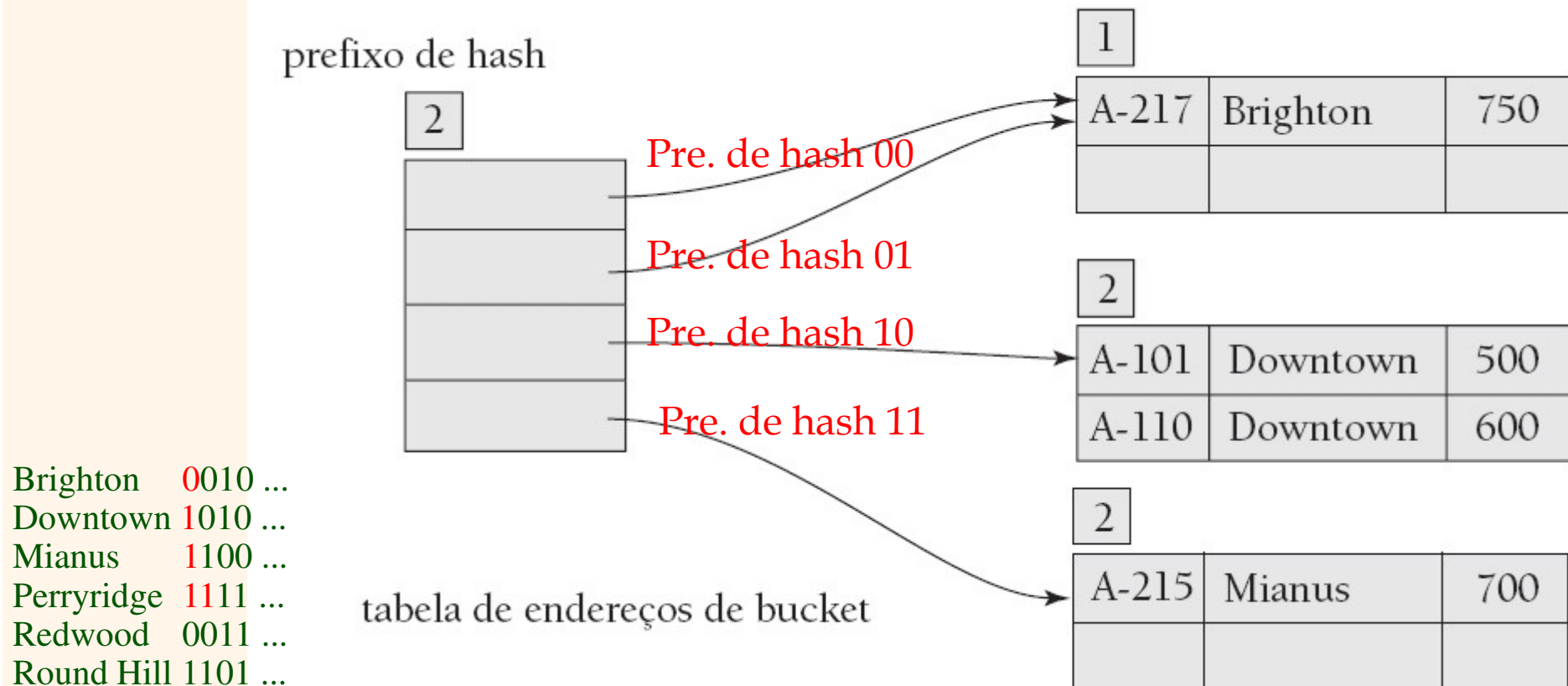


Brighton 0010 ...
Downtown 1010 ...
Mianus 1100 ...
Perryridge 1111 ...
Redwood 0011 ...
Round Hill 1101 ...

Para simplificar, vamos supor que um bucket pode manter apenas 2 registros

Exemplo (Cont.)

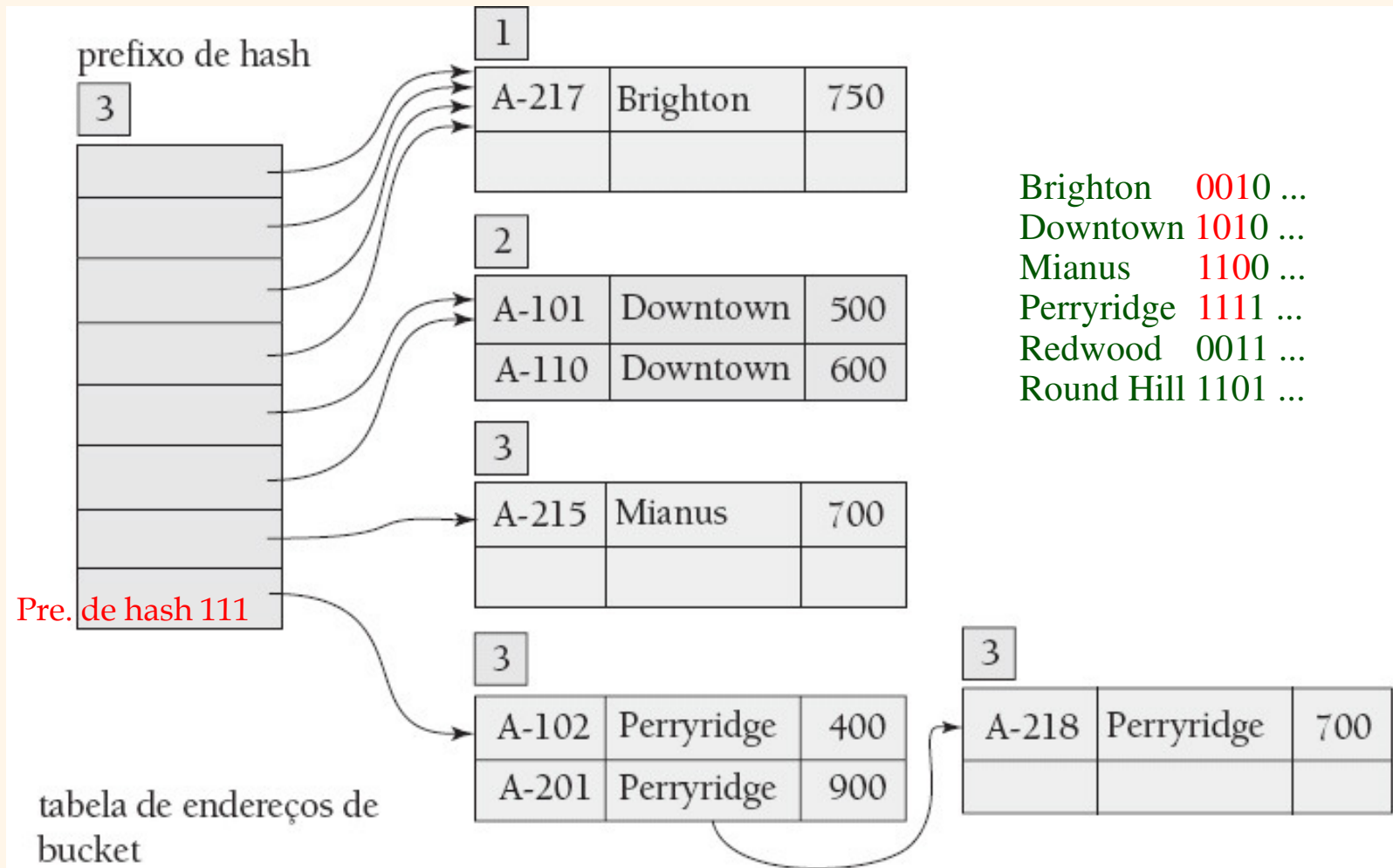
- Estrutura de hash após inserção do registro de **Mianus** – exige aumentarmos hash para 2 bits, a tabela de endereços passa a ter 4 entradas.

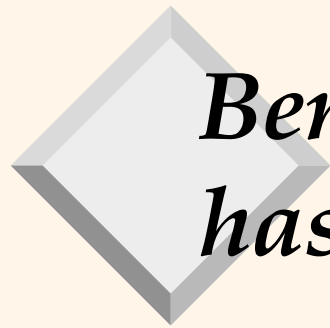


Para simplificar, vamos supor que um bucket pode manter apenas 2 registros

Exemplo (Cont.)

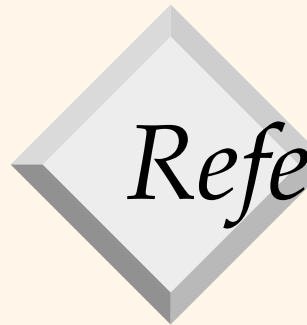
Estrutura de hash após inserção de três registros de **Perryridge**





Benefícios e desvantagens - hashing extensível

- ❖ Benefícios
 - ❖ Desempenho do hash não diminui com o crescimento do arquivo
 - ❖ Sobrecarga de espaço mínima
- ❖ Desvantagens do hashing extensível:
 - ❖ Nível extra de indireção para encontrar registro desejado
 - ❖ A própria tabela de endereços de balde pode se tornar muito grande (maior que a memória)
 - ❖ Precisa de uma estrutura de árvore para localizar o registro desejado na estrutura!
 - ❖ Mudar o tamanho da tabela de endereços de balde é uma operação dispendiosa
- ❖ Hashing linear é um mecanismo alternativo, que evita essas desvantagens ao custo possível de mais estouros de balde



Referência

- ❖ KORTH, H. F.; SILBERSCHATZ, A.; SUDARSHAN, S. Sistema de Banco de Dados, Tradução da 5ª Edição, Campus, 2006.