Universidade Federal de Uberlândia
Faculdade de Computação

Armazenamento secundário e árvores B

Prof. Renato Pimentel

2025/2

Sumário

1 Armazenamento secundário
- Conceitos básicos
- Índices ordenados
- Árvore B$^+$
- Exercícios

- **Mecanismos de indexação** usados para acelerar o acesso aos dados desejados em BD.
  - Ex.: **catálogo de autores** na biblioteca.
- **Chave de busca** – atributo ou conjunto de atributos usado para procurar registros em um arquivo.
- Um **arquivo de índice** consiste em registros (chamados **entradas de índice**) com estrutura:

chave de busca	ponteiro

- Um arquivo de índice é normalmente muito menor que o arquivo original.
- Dois tipos básicos de índices:
  - **Índices ordenados**: as chaves de pesquisa são armazenadas em ordem classificada;
  - **Índices de *hash***: as chaves de pesquisa são distribuídas uniformemente entre "buckets" (entradas) usando uma função de *hashing*.

Métricas de avaliação de índice

- Tipos de acesso suportados de forma eficiente. Exemplos:
  - Registros com um valor específico de um atributo.
  - Registros com um valor de atributo que se enquadra em um intervalo especificado de valores.
- Tempo de acesso;
- Tempo de inserção;
- Tempo de exclusão;
- Sobrecarga de espaço de armazenamento.

Sumário

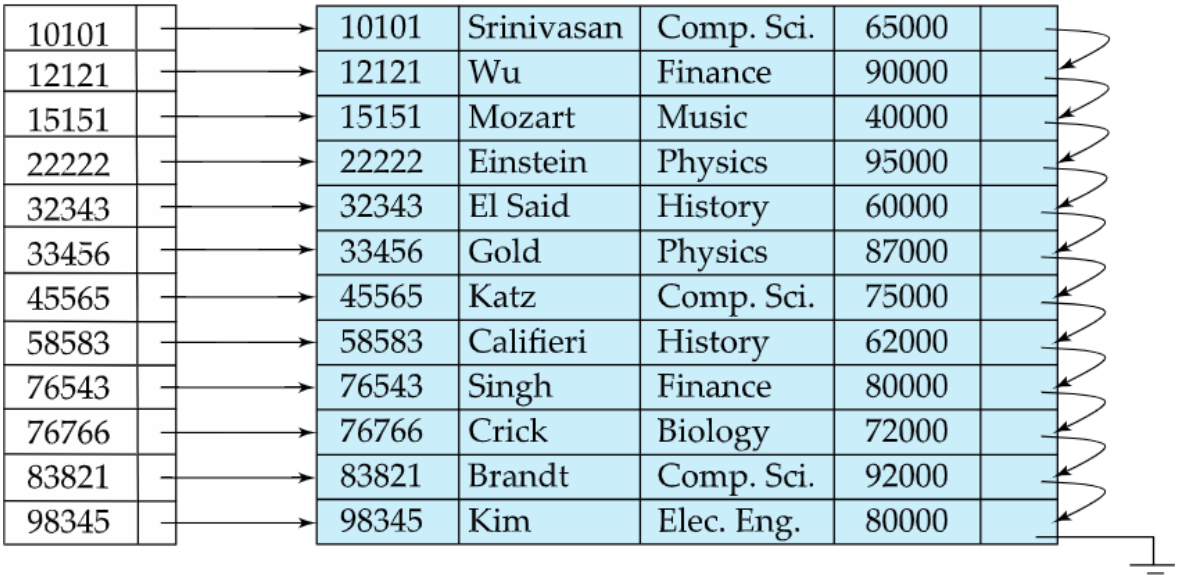
1 Armazenamento secundário
- Conceitos básicos
- Índices ordenados
- Árvore B^+
- Exercícios

Índice ordenado

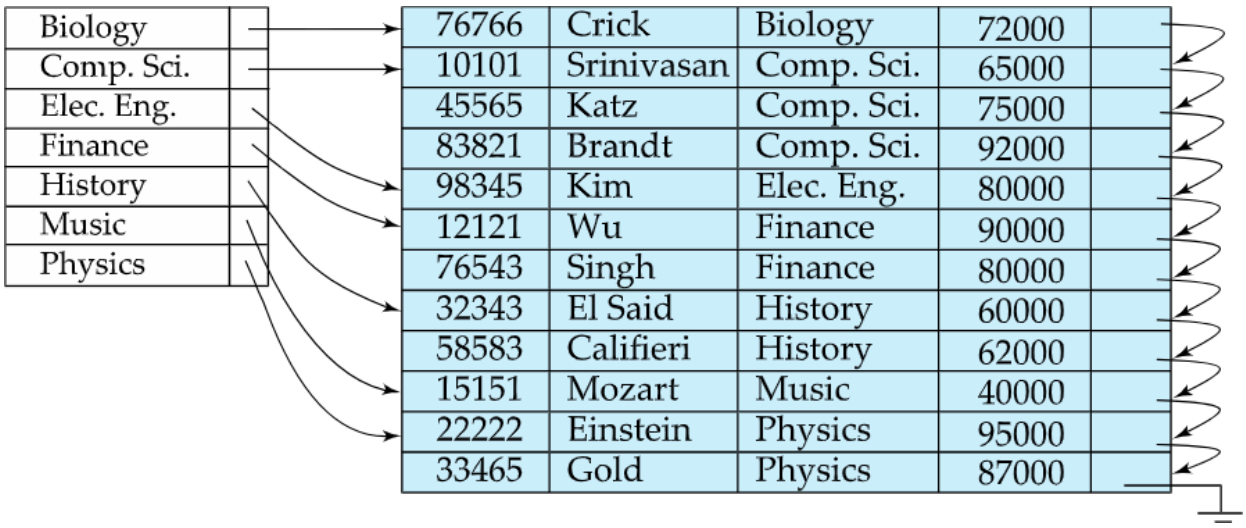
- Em um índice ordenado, as entradas são armazenadas classificadas pelo valor da **chave de busca**:
  - Números, **data/hora**, *strings*.
- **Índice agrupado** (*clustering index*): num arquivo ordenado sequencialmente, o índice cuja chave de busca especifica a ordem sequencial do arquivo.
  - Também chamado de **índice primário**.
  - A chave de busca de um índice primário geralmente é, mas não necessariamente, a chave primária.
- **Índice secundário**: um índice cuja chave de busca especifica uma ordem diferente da ordem sequencial do arquivo. Também chamado de **índice não agrupado** (*nonclustering index*).
- **Arquivo sequencial indexado**: arquivo sequencial ordenado em uma chave de busca, com um índice agrupado na mesma.

Arquivos de índice denso

- **Índice denso** – Há um registro de índice para cada valor de chave de busca no arquivo.
- Por exemplo, índice no atributo `ID` da relação `instrutor`:

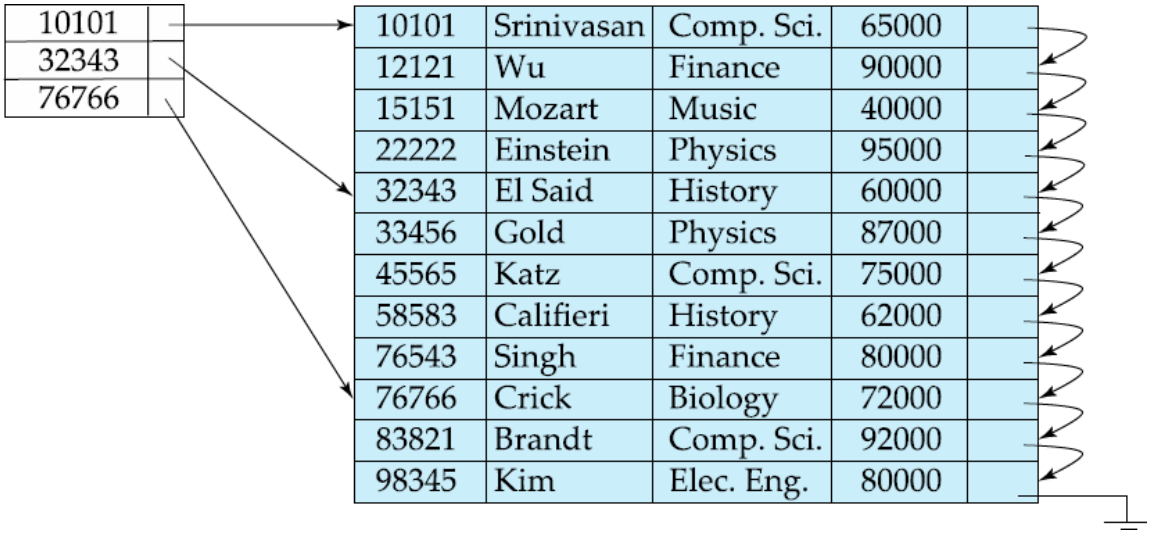


- Índice denso em dept_name, com arquivo de `instrutor` classificado em dept_name:



Arquivos de índice esparso

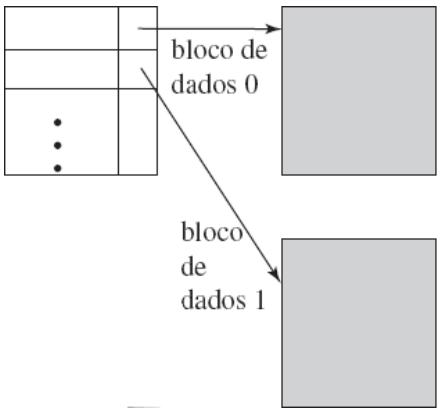
- **Índice esparso**: contém registros de índice apenas para alguns valores da chave de busca.
  - Aplicável quando os registros estão classificados pela chave de busca.
- Para localizar um registro com valor de chave de busca K deve-se:
  - Encontrar a entrada de índice com o maior valor de chave $< K$;
  - Pesquisar sequencialmente o arquivo a partir do registro apontado pela entrada de índice.



10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
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	

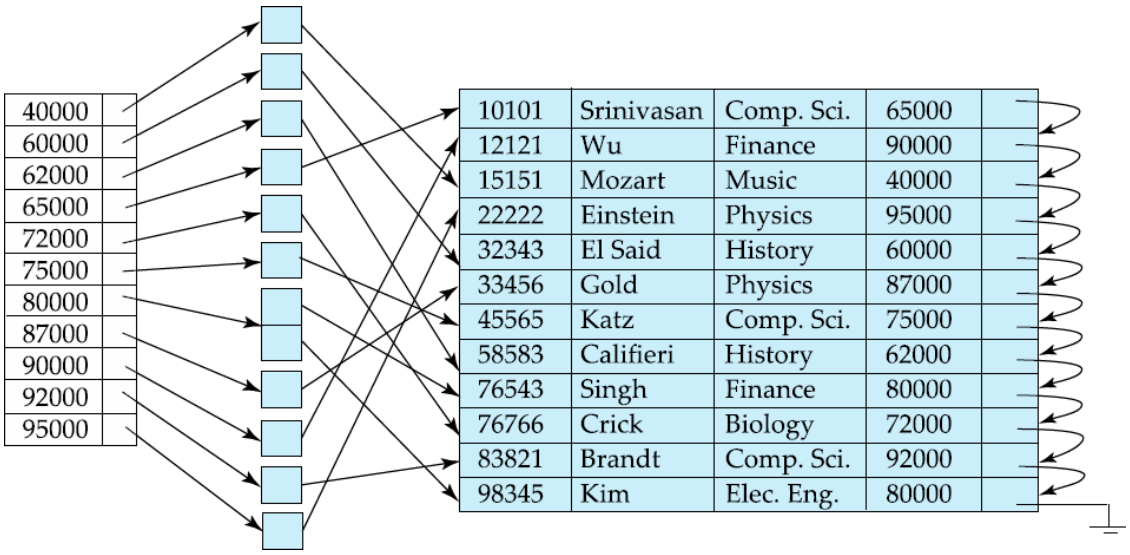
- Comparado com índices densos:
  - Menores espaço e sobrecarga de manutenção para inserções e exclusões.
  - Geralmente mais lento que o índice denso para localizar registros.
- Bom meio termo:
  - Para índice agrupado: índice esparso com uma entrada de índice por **bloco** do arquivo – menor valor da chave de busca no mesmo.



  - Para índice não agrupado: índice esparso sobre índice denso (**índice multinível**)

Exemplo de índice secundário

- Índice secundário no campo salarial de `instrutor`:



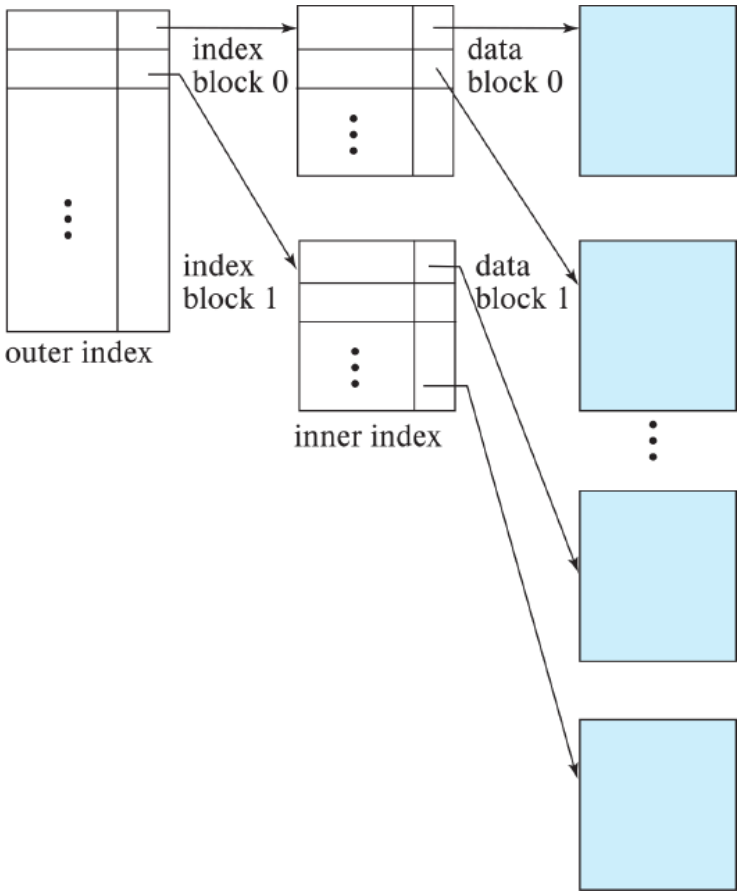
- O registro de índice aponta para um *bucket*, que contém ponteiros para todos os registros reais com aquele valor de chave de busca específico.
- Os índices secundários devem ser densos.

Índices agrupados × índices não agrupados

- Os índices oferecem benefícios substanciais na busca de registros.
- Entretanto, índices impõem sobrecarga na modificação do banco de dados:
  - Quando um registro é inserido ou excluído, cada índice na relação deve ser atualizado;
  - Quando um registro é atualizado, qualquer índice em um atributo atualizado deve ser atualizado.
- A varredura sequencial usando índice agrupado é eficiente, mas uma varredura sequencial usando um índice secundário (não agrupado) é cara em disco magnético:
  - Cada acesso ao registro pode buscar um novo bloco do disco;
  - Cada busca de bloco no disco magnético requer cerca de 5 a 10 milissegundos.

Índice multinível

- Se o índice não couber na memória, o acesso se torna caro.
- **Solução**: tratar o índice mantido no disco como um arquivo sequencial e construir um índice esparso nele.
  - **Índice externo** – um índice esparso do índice básico;
  - **Índice interno** – índice básico para o arquivo.
- Se mesmo o índice externo for grande demais para caber na memória principal, outro nível de índice poderá ser criado, e assim por diante.
- Índices em todos os níveis devem ser atualizados a cada inserção ou exclusão no arquivo.



Índices em várias chaves

- **Chave de busca composta**.
  - Por exemplo, índice na relação `instrutor` para atributos `(nome, ID)`
  - Os valores são classificados lexicograficamente.
    - ★ Ex.: `(João, 12121)` $<$ `(João, 13514)` e `(João, 13514)` $<$ `(Pedro, 11223)`.
  - Pode-se buscar apenas pelo nome, ou `(nome, ID)`.

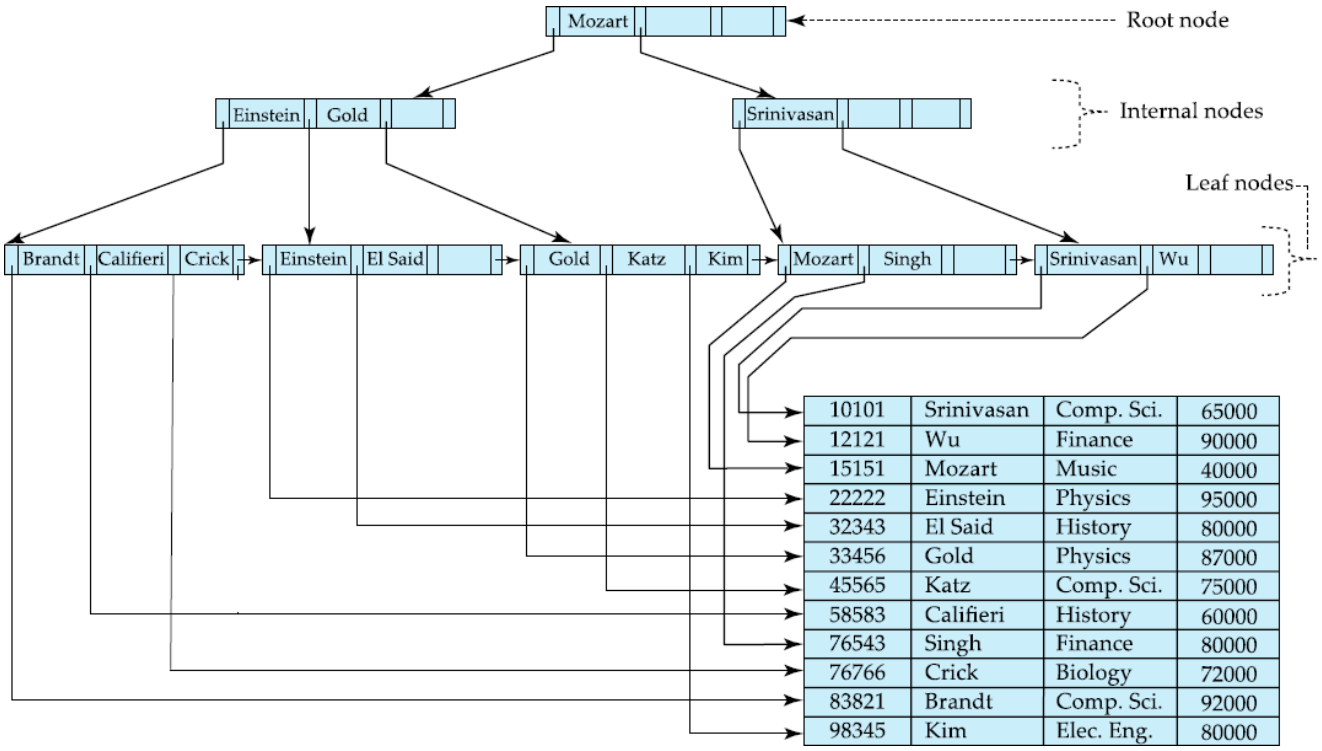
Sumário

1 Armazenamento secundário
- Conceitos básicos
- Índices ordenados
- Árvore B^+
- Exercícios

Árvore B$^+$

- Desvantagem dos arquivos sequenciais indexados:
  - O desempenho diminui à medida que o arquivo cresce, pois muitos **blocos de estouro** (*overflow blocks*) são criados.
  - É necessária a reorganização periódica de todo o arquivo – não desejável.
- Vantagem dos arquivos de índice de árvore B$^+$:
  - Reorganiza-se automaticamente com pequenas alterações locais, em face de inserções e exclusões.
  - Não é necessária a reorganização de todo o arquivo para manter o desempenho.
- (Menor) desvantagem das árvores B$^+$:
  - Sobrecarga extra na inserção e exclusão, sobrecarga de espaço.
- Vantagens das árvores B$^+$ superam as desvantagens:
  - Árvores B$^+$ são amplamente utilizadas.

Exemplo de árvore B$^+$



Definição

- A árvore B^+ é uma árvore enraizada que satisfaz as seguintes propriedades:
  - Todos os caminhos da raiz até a folha têm a mesma altura;
  - Cada nó que não é a raiz ou uma folha tem entre $\lceil n/2 \rceil$ e n filhos.
  - Um nó folha tem entre $\lceil (n-1)/2 \rceil$ e $n-1$ valores.
  - Casos especiais:
    - ★ Se a raiz não for uma folha, ela tem pelo menos 2 filhos.
    - ★ Se a raiz for uma folha (i.e. se não houver outros nós na árvore), ela pode ter entre 0 e $n-1$ valores.

Estrutura de um nó da árvore B^+

- Nó típico:

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

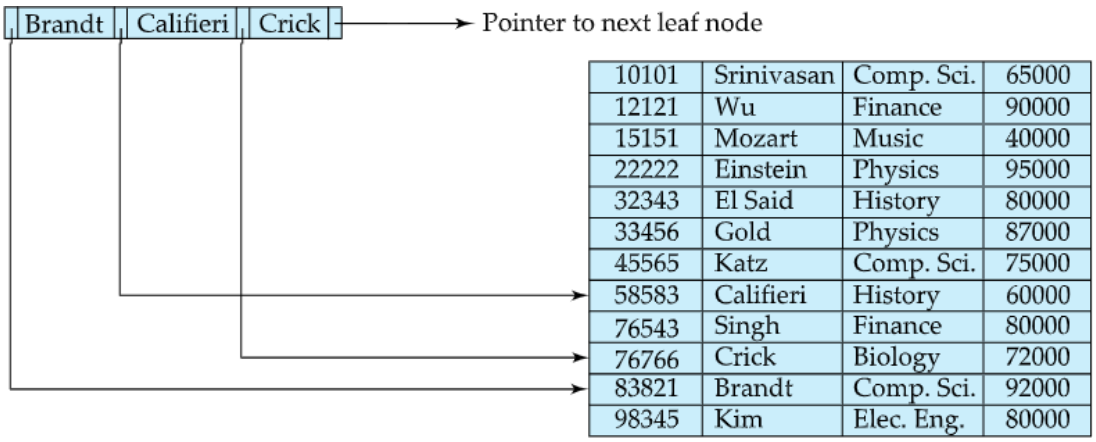
  - K_i são valores de chaves de busca;
  - P_i são ponteiros para filhos (nós internos), ou ponteiros para registros ou grupos de registros (nós folhas).
- As chaves de busca num nó são ordenadas:

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

(Inicialmente sem chaves duplicadas)

Nó folha de uma árvore B$^+$

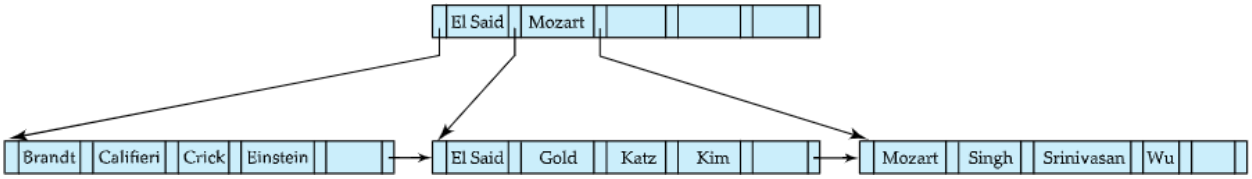
- Propriedades de um nó folha:
  - Para $i = 1, 2, \ldots, n-1$, o ponteiro P_i aponta para um registro de arquivo com valor de chave de busca K_i;
  - Se L_i e L_j são nós folhas e $i < j$, os valores das chaves de busca de L_i são menores ou iguais aos valores das chaves de busca de L_j;
  - P_n aponta para o próximo nó folha, seguindo a ordem das chaves de busca.



Nó interno de uma árvore B$^+$

- Nós internos formam um índice esparso multinível para os nós-folha. Para um nó interno com n ponteiros:
  - Todas as chaves de busca na subárvore para a qual P_1 aponta são menores que K_1;
  - Para $2 \leq i \leq n-1$, todas as chaves de busca na subárvore para a qual P_i aponta têm valores maiores ou iguais a K_{i-1} e menores que K_i;
  - Todas as chaves de busca na subárvore para a qual P_n aponta têm valores maiores ou iguais a K_{n-1}.

Exemplo de árvore B$^+$



Árvore B$^+$ para arquivo `instrutor` ($n = 6$).

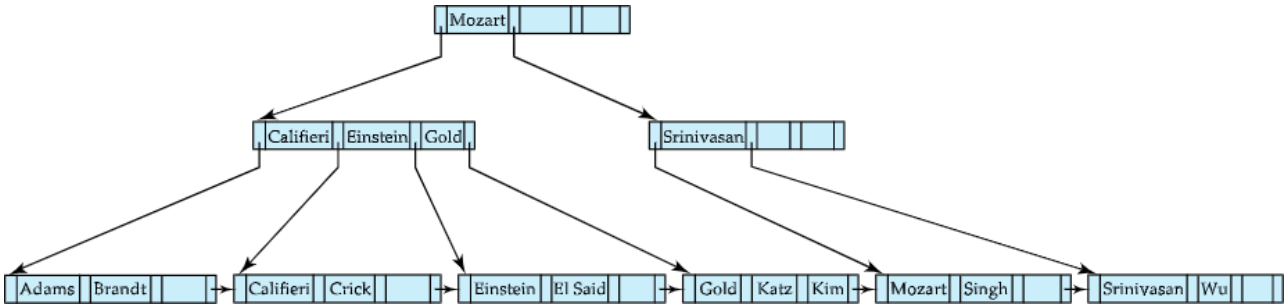
- Os nós folhas devem ter entre 3 e 5 valores ($\lceil (n-1)/2 \rceil$ e $n-1$, com $n = 6$).
- Nós internos (exceto a raiz) devem ter entre 3 e 6 filhos (entre $\lceil n/2 \rceil$ e n com $n = 6$).
- A raiz deve ter pelo menos 2 filhos.

Observações sobre árvores B$^+$

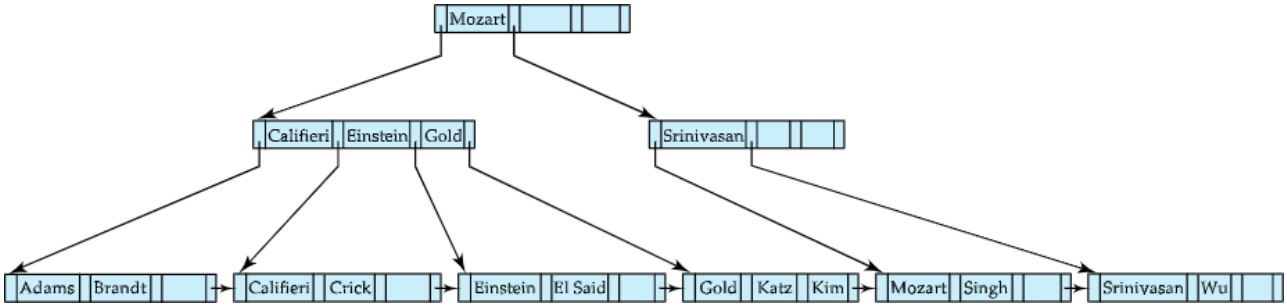
- Como as conexões entre nós são feitas por ponteiros, blocos "logicamente" próximos não precisam ser "fisicamente" próximos.
- Os níveis internos da árvore B$^+$ formam uma hierarquia de índices esparsos.
- A árvore B$^+$ contém um número relativamente pequeno de níveis:
  - ☐ O nível abaixo da raiz tem pelo menos $2 \cdot \lceil n/2 \rceil$ valores;
  - ☐ O próximo nível tem pelo menos $2 \cdot \lceil n/2 \rceil \cdot \lceil n/2 \rceil$ valores;
  - ☐ ... etc.
  - Se houver K valores de chave de busca no arquivo, a altura da árvore será no máximo $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$;
  - Assim as buscas podem ser conduzidas de forma eficiente.
- Inserções e exclusões no arquivo principal podem ser tratadas com eficiência, pois o índice pode ser reestruturado em tempo logarítmico.

Consultas em árvores B$^+$

```
function encontrar(v)
  C = nó raiz
  while (C não é um nó folha) begin
    seja i = menor número tal que v <= C.Ki
    if não houver tal número i then begin
      C = último ponteiro não nulo em C
    end
    else if ( v = C.Ki ) then C = Pi+1
    else C = C.Pi
  end
  if para algum i , Ki = v then
    retorne C.Pi
  else retorne null /* nenhum registro com valor v */
```



- **Consultas por intervalo** encontram todos os registros com valores de chave de busca num determinado intervalo.
  - Veja o livro do Silberschatz para detalhes da função `findRange(lb, ub)`, que retorna o conjunto de tais registros;
  - Implementações reais geralmente fornecem uma interface iteradora para buscar registros correspondentes um de cada vez, usando uma função `next()`.



Altura e desempenho da B^+

- Se houver K registros no arquivo, a altura da árvore não será maior que $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Um nó tem geralmente o mesmo tamanho de um **bloco de disco**, normalmente 4 kilobytes.
  - e um valor típico de n é em torno de 100 (40 bytes por entrada de índice)
- Com 1 milhão de valores de chave de busca e $n = 100$:
  - no máximo $\log_{50}(1.000.000) = 4$ nós são acessados numa travessia de consulta da raiz até uma folha.
- Compare isso com uma árvore binária balanceada com 1 milhão de valores da chave de busca – cerca de 20 nós serão acessados em uma consulta.
  - A diferença acima é significativa, pois cada acesso ao nó pode precisar de uma operação de E/S de disco, custando cerca de **10 milissegundos**!
  - Em memória *flash*, 10 a 100 microssegundos – ainda significativo!

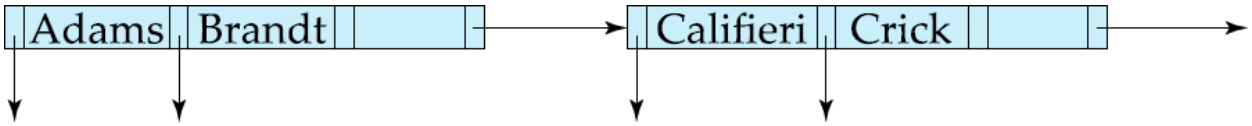
Chaves não-exclusivas

- Se uma chave de busca a_i não é chave candidata – i.e. pode ter duplicatas – cria-se em um índice em uma **chave composta** (a_i, A_p), que é única.
  - A_p pode ser uma chave primária, ID de registro, ou qualquer outro atributo que garanta exclusividade.
- A busca por $a_i = v$ pode ser implementada por uma **consulta por intervalo** na chave composta, com intervalo $(v, -\infty)$ a $(v, +\infty)$.
- Porém mais operações de E/S são necessárias para buscar os registros reais
  - Se o índice estiver agrupado, todos os acessos serão sequenciais.
  - Se o índice não for agrupado, cada acesso ao registro pode precisar de uma operação de E/S.

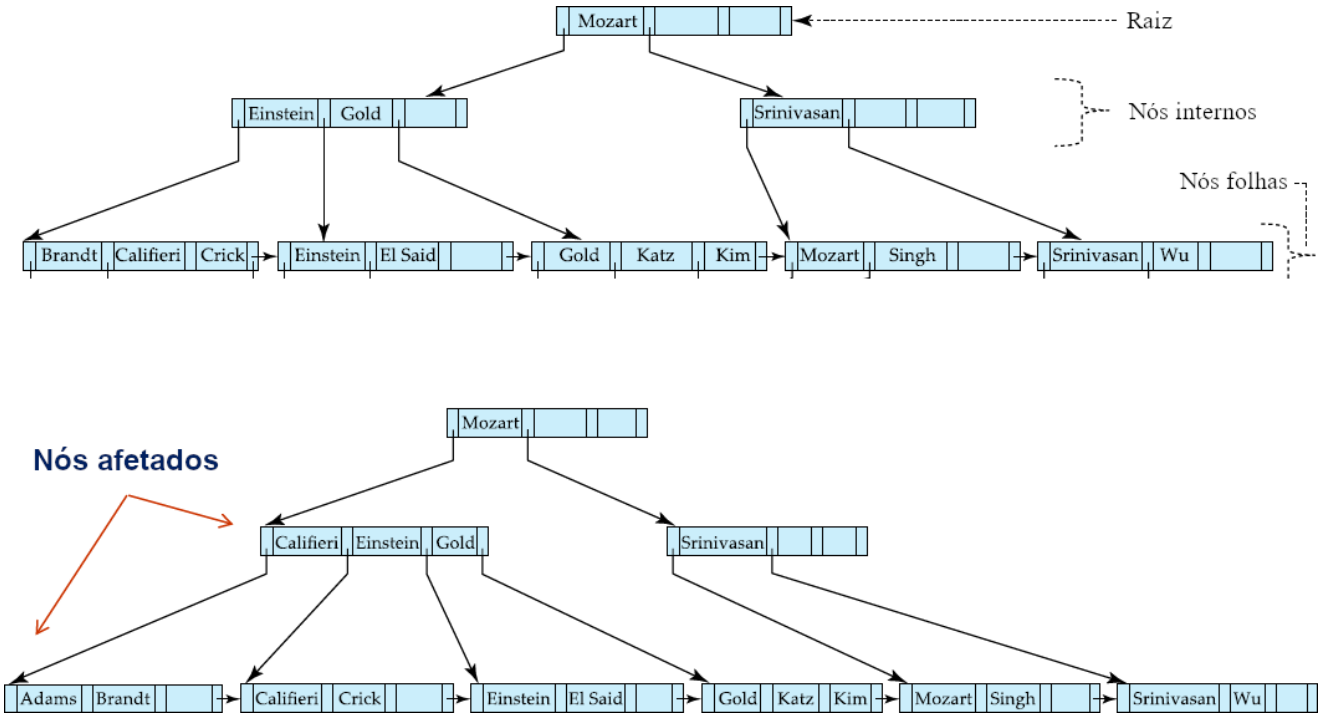
Inserção na árvore B^+

- Suponha que o registro já foi adicionado ao arquivo. Seja:
  - pr seja um ponteiro para o registro, e
  - v o valor da chave de busca do registro.
- Encontre o nó folha no qual o valor da chave de busca apareceria.
  1. Se houver espaço no nó folha, insira o par (v, pr) no nó folha;
  2. Caso contrário, divida o nó – junto com a nova entrada (v, pr) – conforme discutido a seguir e propague as atualizações para os nós pais.

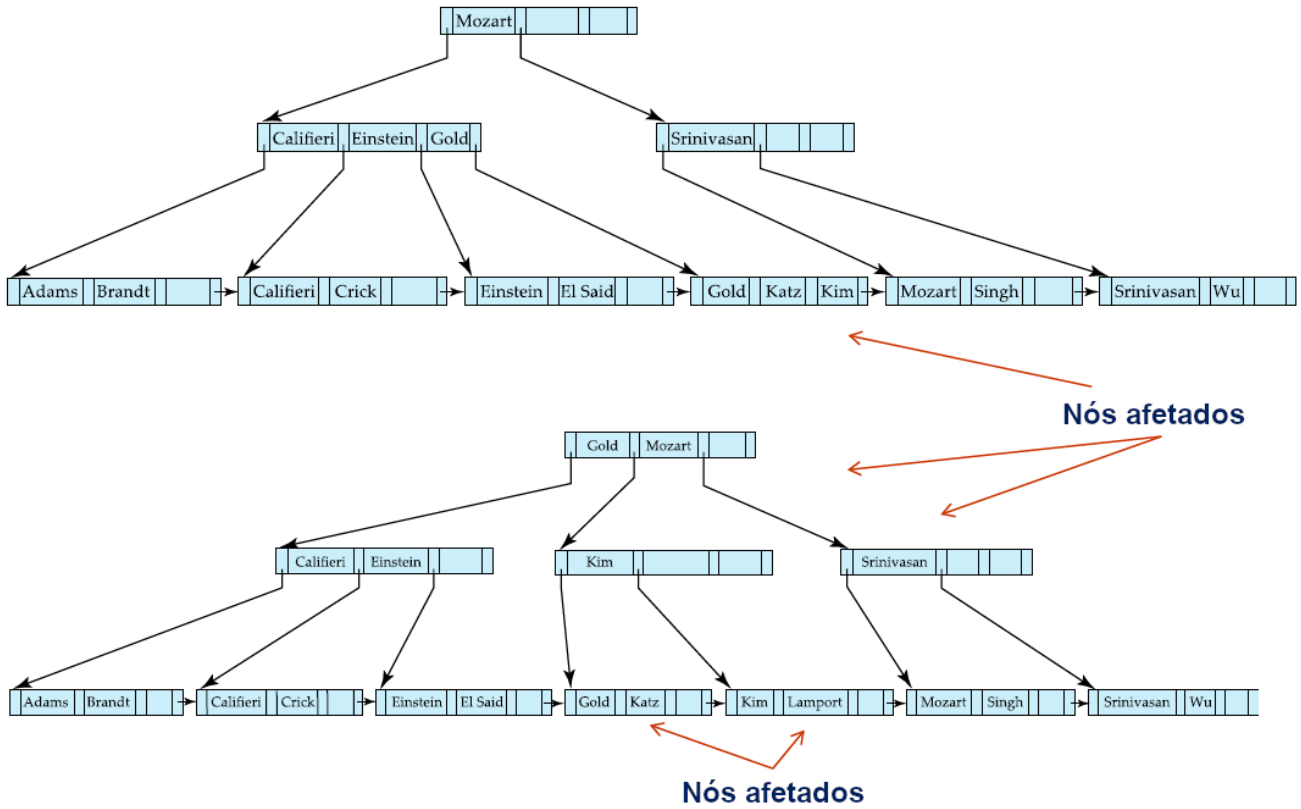
- Dividindo um nó folha:
  - Tomam-se os n pares (valor da chave de busca, ponteiro) (incluindo o que está sendo inserido) em ordem. Colocam-se os primeiros $\lceil n/2 \rceil$ no nó original, e o restante em um novo nó.
  - Seja p o novo nó, e k o menor valor da chave em p. Insira (k, p) no pai do nó que está sendo dividido.
  - Se o pai estiver completo, divida-o e **propague** a divisão para cima.
- A divisão dos nós prossegue para cima até que um nó que não esteja cheio seja encontrado.
  - No pior caso, o nó raiz pode ser dividido, aumentando a altura da árvore em 1.



Resultado da divisão do nó contendo Brandt, Califieri e Crick na inserção de Adams. Próximo passo: inserir entrada com (Califieri, ponteiro-para-novo-nó) no pai.

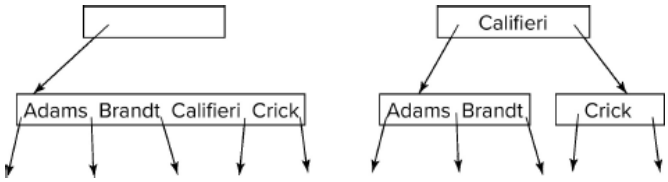


Árvore B^+ antes e após a inserção de Adams.



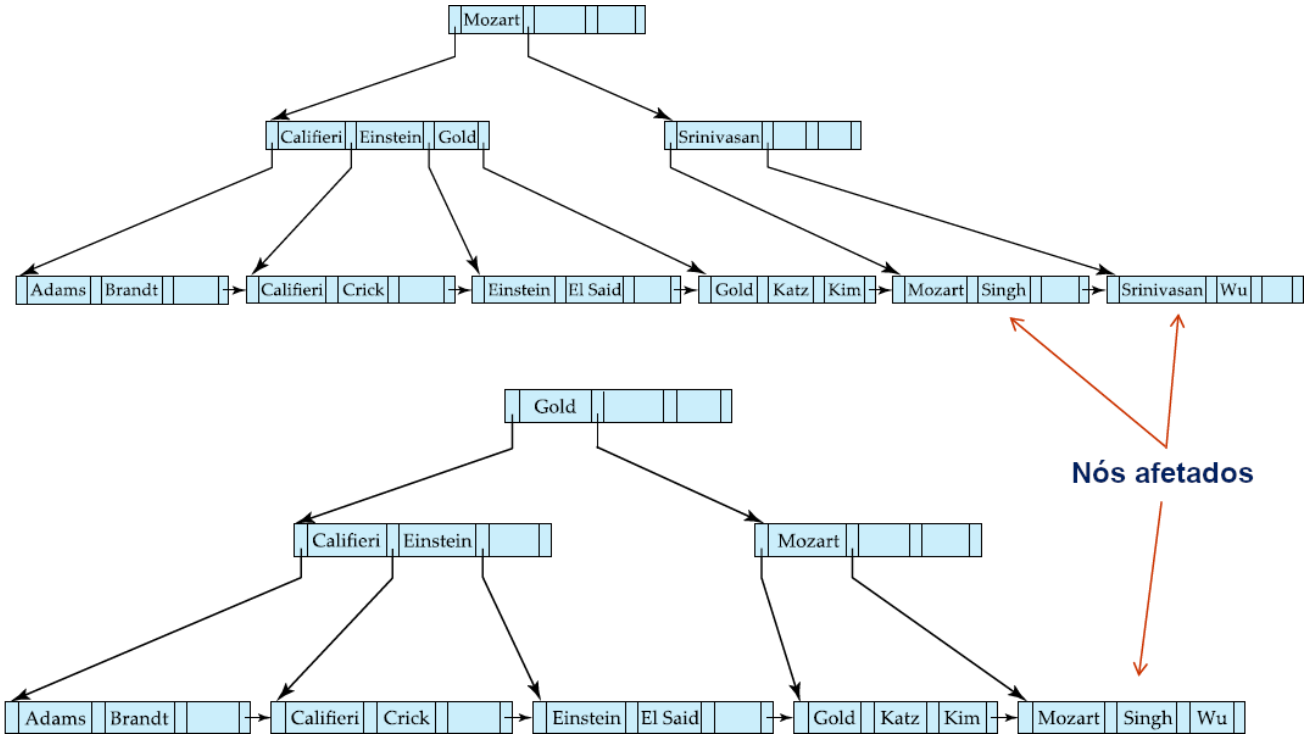
Árvore B^+ antes e após a inserção de Lamport.

- Dividindo um nó interno: ao inserir (k, p) em um nó interno N já completo:
  - Copie N para uma área de memória M com espaço para $n+1$ ponteiros e n chaves;
  - Insira (k, p) em M;
  - Copie $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ de M de volta para o nó N;
  - Copie $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ de M para um nó recém-alocado N'
  - Insira $(K_{\lceil n/2 \rceil}, N')$ no pai de N.
- Exemplo:



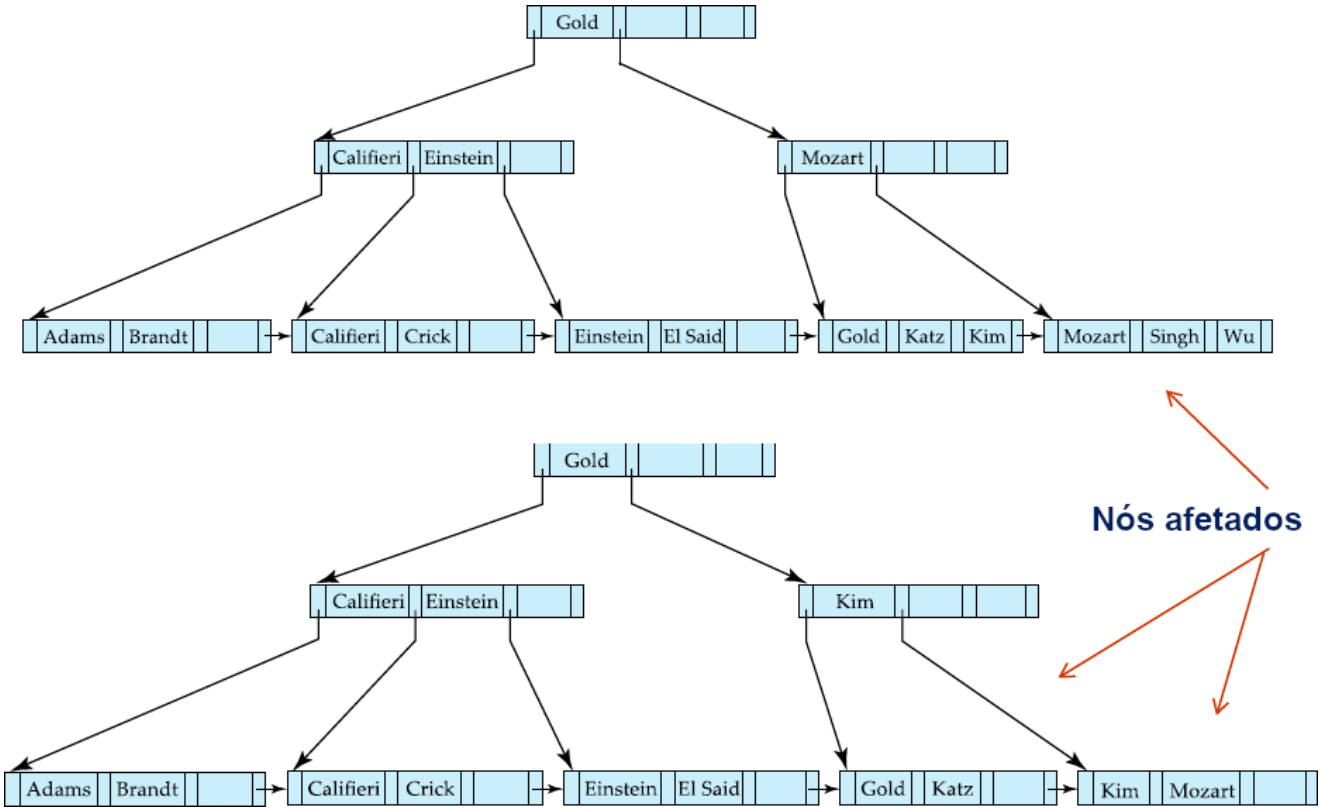
- Pseudocódigo: livro do Silberschatz.

Exemplos de exclusão na árvore B$^+$

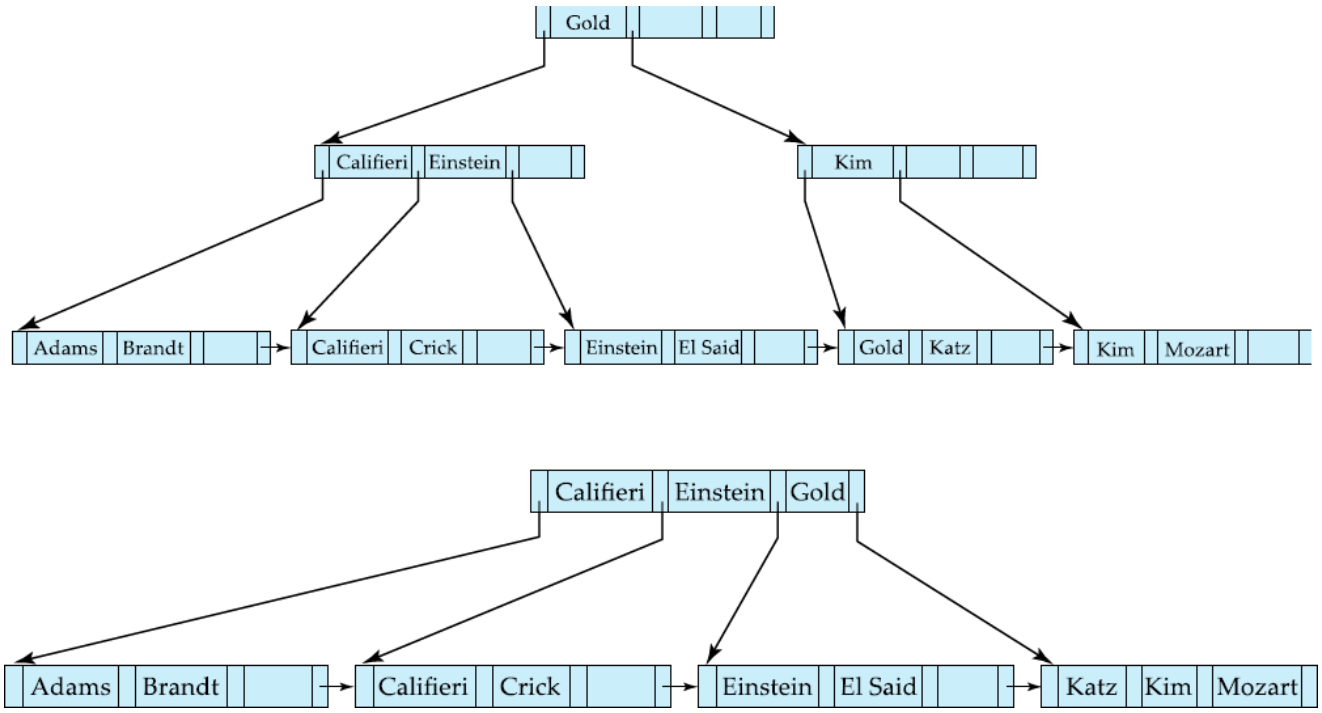


Antes e depois de excluir Srinivasan.

A exclusão causa a **fusão** das folhas não totalmente cheias abaixo



- Antes e depois de excluir Singh e Wu: a folha contendo Singh e Wu ficou com menos de $\lceil n/2 \rceil$ valores; **toma-se emprestado** Kim de seu irmão esq.;
- O valor da chave de busca no pai muda como resultado.



- Antes e depois de excluir Gold: nó com Gold e Katz ficou incompleto e foi mesclado com seu irmão; o mesmo ocorre com o nó pai.
  - O valor que separa os dois nós (no pai) **desce** na mesclagem.
- A raiz então tem apenas um filho e é excluída.

Exclusão na árvore B^+

- Suponha que o registro já foi excluído do arquivo. Seja v o valor de chave de busca do registro e pr o ponteiro para o registro.
- Remover (v, pr) do nó folha;
- Se o nó tiver poucas entradas devido à remoção, e se suas entradas e as de um irmão couberem em um único nó, então:
  - Insira todos os valores da chave de busca nos dois nós em um único nó (o da esquerda), e exclua o outro nó.
  - Exclua de seu pai o par (K_{i-1}, P_i), onde P_i é o ponteiro para o nó excluído, usando recursivamente o procedimento acima.

- Caso contrário, se o nó tiver poucas entradas devido à remoção, mas as entradas no nó e em um irmão não couberem em um único nó, **redistribua os ponteiros**:
  - Redistribua os ponteiros entre o nó e um irmão de modo que ambos tenham mais do que o número mínimo de entradas.
  - Atualize o valor da chave de busca correspondente no pai do nó.
- As exclusões de nós podem ocorrer em cascata, até que um nó que tenha $\lceil n/2 \rceil$ ou mais ponteiros seja encontrado;
- Se o nó raiz tiver apenas um ponteiro após a exclusão, ele será excluído e o único filho passa a ser a nova raiz.

Complexidade das atualizações

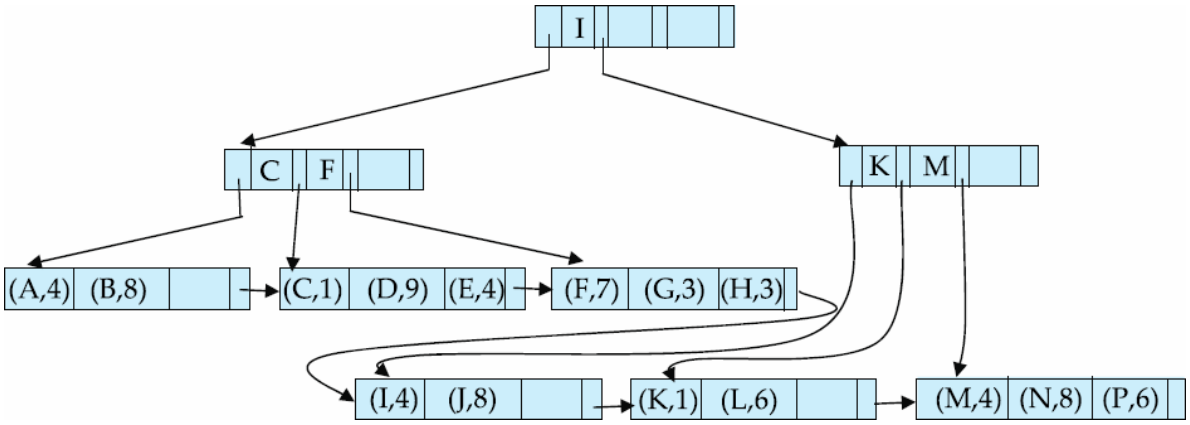
- Custo (em termos de número de operações de E/S) de inserção e exclusão de uma única entrada é proporcional à **altura** da árvore:
  - Com K entradas e um número máximo de n filhos, a pior complexidade de caso de inserção/exclusão de uma entrada é $O(\lceil \log_{\lceil n/2 \rceil}(K) \rceil)$
- Na prática, o número de operações de E/S é menor:
  - Os nós internos tendem a estar em *buffer* de memória;
  - Divisões/mesclas são raras: a maioria das operações de inserção/exclusão afetam apenas um nó folha.
- A ocupação média dos nós depende da ordem de inserção:
  - 2/3 ($\approx$ 67%) se aleatoriamente, 50% com inserção ordenada.

Chaves não-exclusivas

- Se lista de ponteiros de tupla com cada chave:
  - Código extra para lidar com listas longas;
  - A exclusão de uma tupla pode ser cara se houver muitas duplicatas na chave de busca
    - ⋆ A pior complexidade pode ser linear!
  - Baixo *overhead* de espaço, sem custo extra para consultas.
- Torne a chave de busca única adicionando um identificador de registro (ex.: adicionando a chave primária)
  - Espaço extra de armazenamento para chaves;
  - Código mais simples para inserção/exclusão;
  - Amplamente utilizado.

- Organização de arquivos por árvore B$^+$:
  - Os nós folhas em uma organização de arquivos por árvore B$^+$ armazenam registros, em vez de ponteiros.
  - Ajuda a manter os registros de dados agrupados mesmo quando há inserções/exclusões/atualizações.
- Os nós folhas ainda precisam estar meio cheios:
  - Como os registros são maiores que os ponteiros, o número máximo de registros que podem ser armazenados em um nó folha é menor que o número de ponteiros em um nó interno.
- A inserção e a exclusão são tratadas da mesma forma que a inserção e a exclusão de entradas em um índice por árvore B$^+$.

- Exemplo de organização de arquivos de árvore B$^+$ (*Index-Organized Tables*)



- A boa utilização do espaço é importante, pois os registros usam mais espaço do que os ponteiros.
- Para melhorar a utilização do espaço, envolva mais nós irmãos na redistribuição durante divisões e fusões.
  - Envolver dois irmãos na redistribuição (para evitar divisão/fusão sempre que possível) resulta em cada nó tendo pelo menos $\lfloor 2n/3 \rfloor$ entradas.

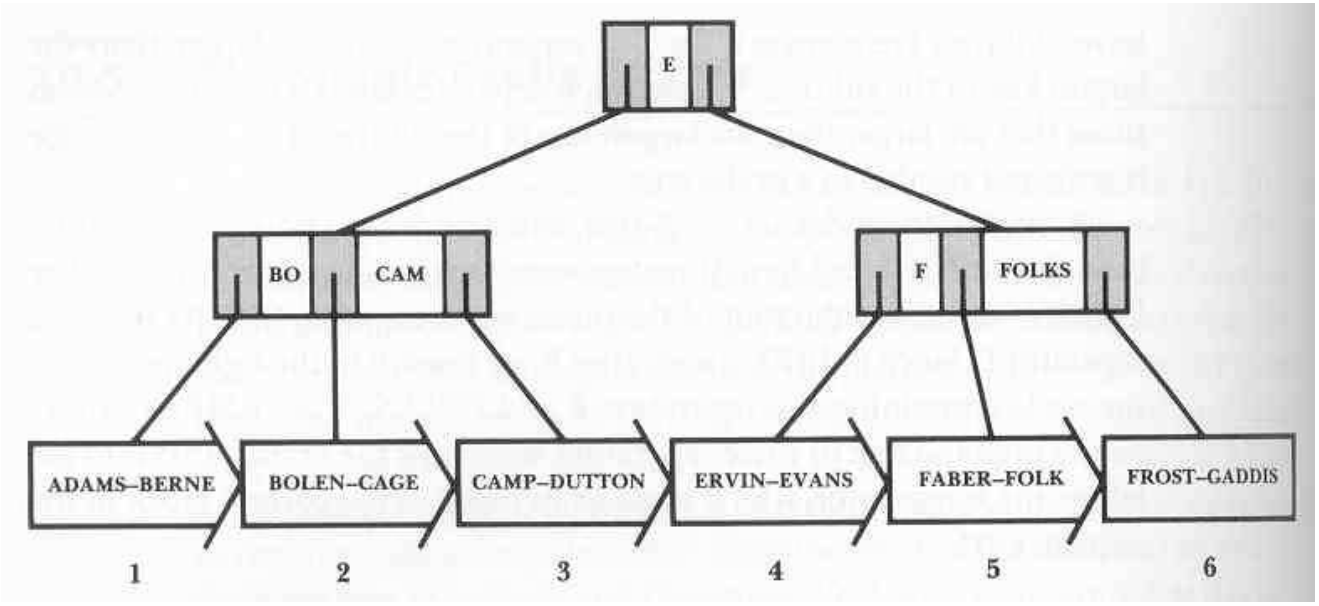
Índices secundários e realocação de registros

- Se um registro for movido (por exemplo, por uma atualização num registro com uma *string* armazenada em um VARCHAR), todos os índices secundários que armazenam ponteiros de registro devem ser atualizados.
- As divisões de nós em organizações de arquivos por B^+ tornam-se muito caras.
- **Solução**: use a **chave de busca** (chave primária) ao invés do ponteiro de registro no índice secundário:
  - Adicione uma id de registro se a chave de pesquisa da organização de arquivo B^+ não for exclusiva;
  - Travessia extra da organização do arquivo para localizar o registro;
  - Custo mais alto para consultas, mas as divisões dos nós são baratas.

Indexando *strings*

- Strings de comprimento variável como chaves
  - *Fanout* variável – *fanout* é o número de ponteiros do nó (quantidade de filhos);
  - Considerar a utilização do espaço como critério para divisão, e não o número de ponteiros.
- Compressão de prefixo:
  - Os valores-chave nos nós internos podem ser prefixos da chave completa.
    - ★ Mantenha caracteres suficientes para distinguir entradas nas subárvores separadas pelo valor da chave.
    - ★ Ex.: Silas e Silberschatz podem ser separados por Silb.
  - As chaves no nó folha podem ser compactadas compartilhando prefixos comuns.

Indexando *strings*: árvore B$^+$ de prefixo simples

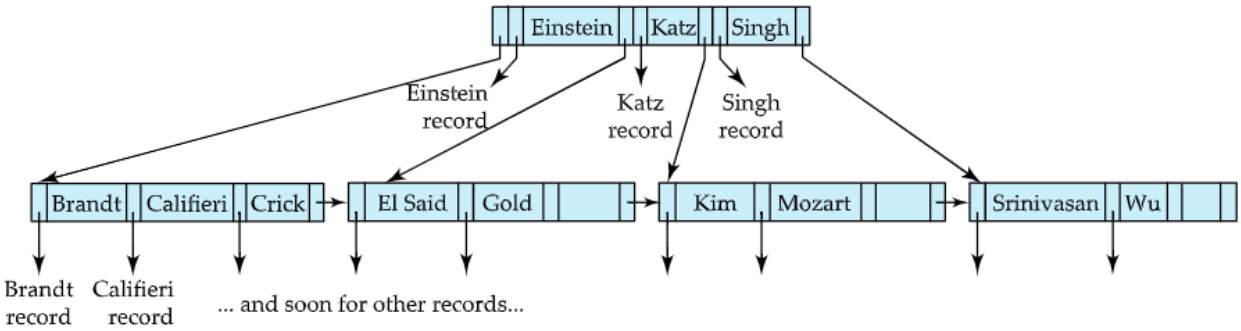


Carregamento em massa (*bulkloading*) e construção de baixo para cima (*bottom-up*)

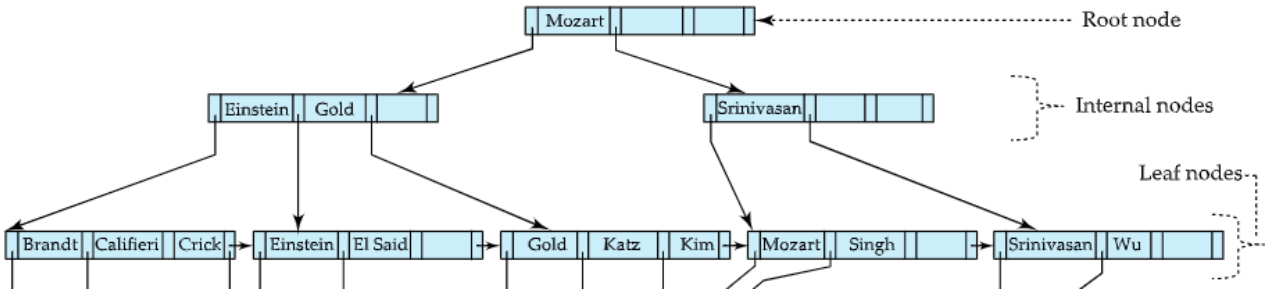
- Inserir entradas uma a uma numa árvore B$^+$ requer diversas operações de E/S (no melhor caso, 1; no pior, proporcional à altura da mesma).
  - Assumindo-se que o nível da folha não cabe na memória:
  - pode ser muito ineficiente inserir inúmeras entradas de uma vez (**carregamento em massa**)
- **Alternativa eficiente 1**:
  - ordenar as entradas primeiro (usando algoritmos eficientes para ordenação em memória externa – Sec. 15.4 livro-texto Silberschatz)
  - inserir em ordem:
    - ★ A inserção será na página existente (ou causará uma divisão)
    - ★ Desempenho de IO significativamente melhorado, porém com maioria dos nós folhas apenas meio cheios.

- **Alternativa eficiente 2: construção de árvore B^+ de baixo para cima**
  - Como antes, ordene as entradas;
  - E então, crie a árvore camada por camada, começando pelo nível das folhas;
  - Implementado como parte do utilitário de carregamento em massa pela maioria dos sistemas de banco de dados.

Exemplo de arquivo de índice de árvore B



Árvore B (acima) e B+ (abaixo) para os mesmos dados



Indexação em *flash*

- Custo de E/S aleatório muito menor no flash
  - 20 a 100 microssegundos para leitura/gravação;
- As reescritas não são feitas no mesmo local e (eventualmente) exigem uma exclusão mais cara;
- Tamanho de página ideal, portanto muito menor;
- O carregamento em massa ainda é útil, pois minimiza a eliminação de páginas;
- Estruturas de árvore otimizadas para gravação foram adaptadas para minimizar gravações de página para árvores de busca otimizadas para **flash**.

Indexação na memória principal

- Acesso aleatório na memória:
  - Muito mais barato do que em disco/*flash*;
  - Mas ainda caro comparado à leitura em cache.
    - ★ Estruturas de dados que fazem melhor uso de cache são preferíveis;
  - A busca binária por um valor de chave em um grande nó de árvore B^+ resulta em muitas falhas de cache.
- B^+ com pequenos nós que caibam em cache são preferíveis para reduzir tais falhas;
- **Ideia principal**: usar um tamanho de nó grande para otimizar o acesso ao disco, mas estruturar os dados em um nó usando uma árvore com tamanho de nó pequeno, em vez de usar um *array*.

Sumário

1. Armazenamento secundário
   - Conceitos básicos
   - Índices ordenados
   - Árvore B^+
   - Exercícios

Exercícios I

1. Qual é o número máximo de acessos a disco para encontrar uma chave em uma árvore B^+ de ordem n no pior caso? (considere a ordem como o número máximo de filhos de um nó).
2. Explique as árvores B^+ de prefixo simples.
3. Considere uma árvore B^+ com $n = 4$ filhos em que todos os elementos estão presentes em páginas folha. Considere o algoritmo de divisão de páginas padrão. Apresente graficamente as seguintes operações de inserção na árvore: 13, 28, 36, 45, 52, 3, 4, 8, 17, 20, 10, 14, 25, 40, 29, 37, 50, 43, 30, 7, 16, 6. Apresente graficamente a remoção dos elementos 28, 10, 40, 45, 3, 14.

Referências

1. SILBERSCHATZ, A. **Sistema de Banco de Dados**. 7. ed. Rio de Janeiro: GEN LTC, 2020. Disponível em: https://integrada.minhabiblioteca.com.br/reader/books/9788595157552.
   Recomendação de leitura: capítulo 14 do livro (até Sec. 14.4, inclusive)

Os materiais desta seção foram gentilmente cedidos por Humberto L. Razente (FACOM/UFU).
Adaptações: Renato Pimentel, FACOM/UFU