



Árvores-B (B-Trees)

Estrutura de Dados II



Problema

- O problema fundamental associado à manutenção de um índice em disco é que o acesso é muito lento, o que coloca problemas para uma manutenção eficiente.
- O melhor acesso a um índice ordenado, até agora, foi dado pela Pesquisa Binária, porém:
 - Pesquisa binária requer muitos acessos. 15 itens podem requerer 4 acessos, 1.000 itens podem requerer até 10 acessos. Esses números são muito altos.
 - Pode ficar muito caro manter um índice ordenado de forma a permitir busca binária. É necessário um método no qual a inserção e a eliminação de registros tenha apenas efeitos locais, isto é, não exija a reorganização total do índice.



Solução através de Árvores Binárias de Busca

- Os registros são mantidos num arquivo, e ponteiros (**esq** e **dir**) indicam aonde estão os registros filhos.
- Esta estrutura pode ser mantida em memória secundária: os ponteiros para os filhos dariam o RRN das entradas correspondentes aos filhos, i.e., a sua localização no arquivo.

Solução através de Árvores Binárias de Busca

AX CL DE FB FT HN JD KF NR PA RF SD TK WS

FIGURE 8.1 Sorted list of keys.

FIGURE 8.2 Binary search tree representation of the list of keys.

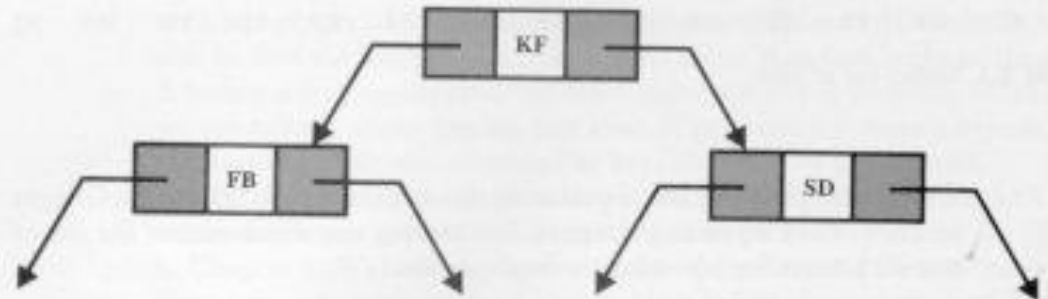
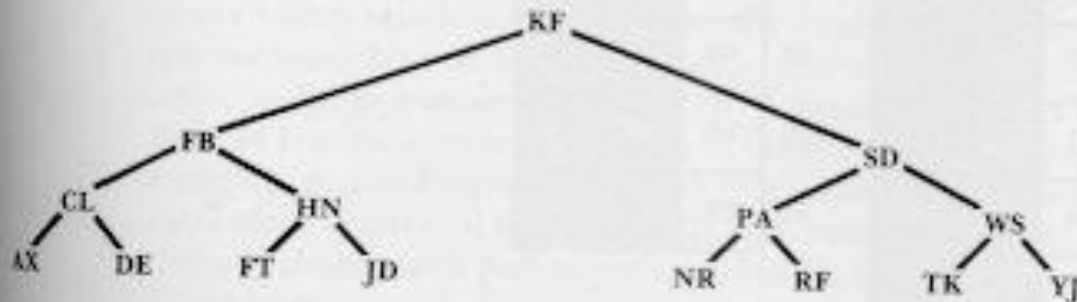
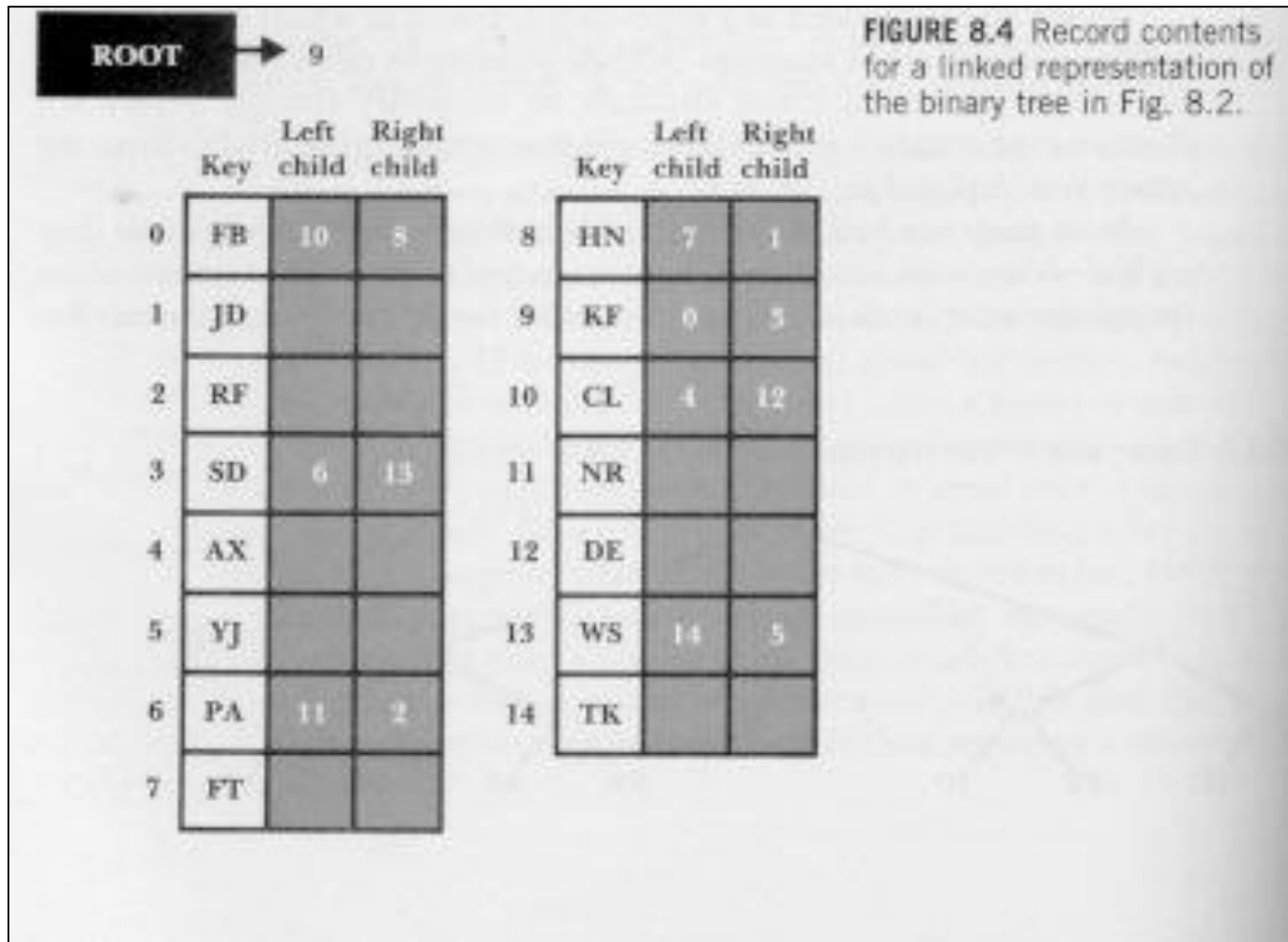


FIGURE 8.3 Linked representation of part of a binary search tree.

Solução através de Árvores Binárias de Busca





Solução através de Árvores Binárias de Busca

- **Vantagem:**

- A ordem lógica dos registros não está associada à ordem física no arquivo.
- O arquivo físico do índice não precisa mais ser mantido ordenado, uma vez que a sequência física dos registros no arquivo é irrelevante: o que interessa é poder recuperar a estrutura lógica da árvore, o que é feito através dos campos **esq** e **dir**.
- Se acrescentarmos uma nova chave ao arquivo, por exemplo LV, é necessário apenas saber aonde inserir esta chave na árvore, de modo a mantê-la como ABB. A busca pelo registro é necessária, mas a reorganização do arquivo não é.

Solução através de Árvores Binárias de Busca

Desempenho da busca ainda continua bom, uma vez que a árvore se encontra em um estado balanceado

Mas o que acontece se inserirmos as chaves
NP MB TM LA UF ND TS NK

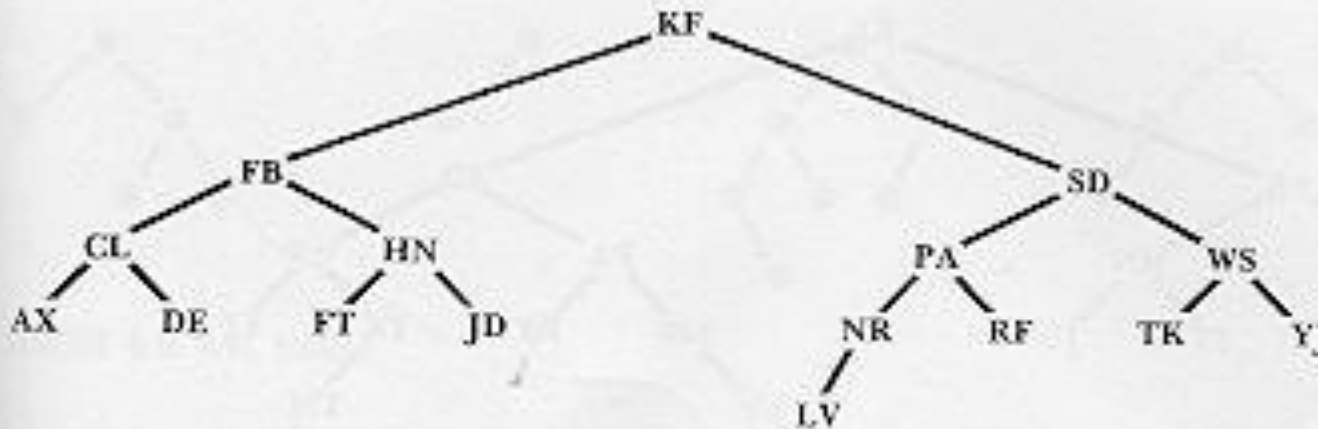


FIGURE 8.5 Binary search tree with LV added.

Solução através de Árvores Binárias de Busca

Árvore Desbalanceada

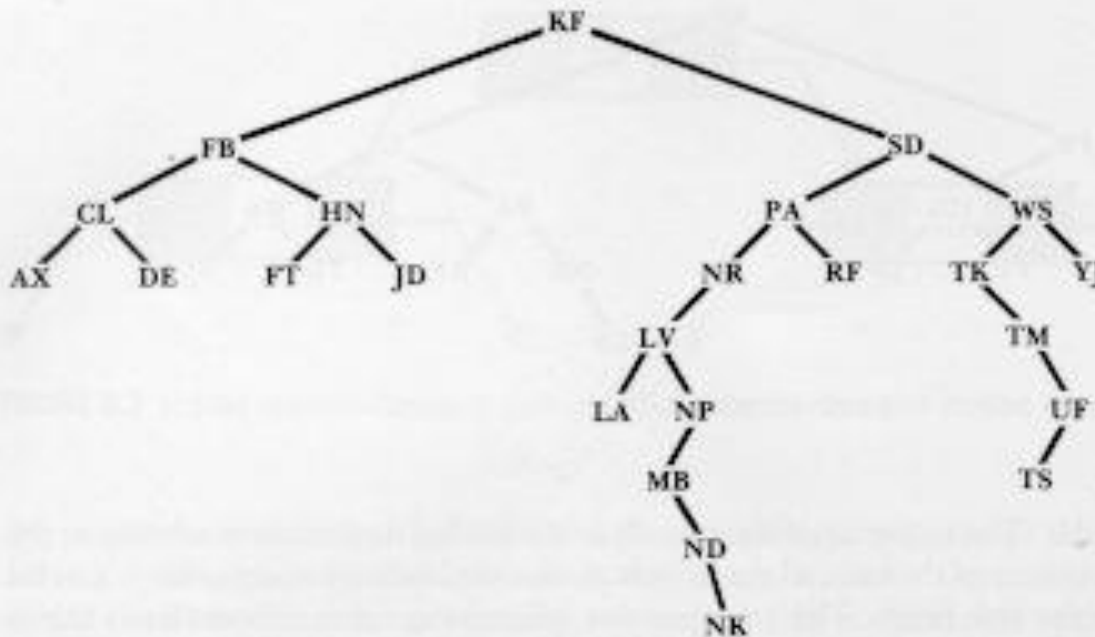


FIGURE 8.6 Binary search tree showing the effect of added keys.

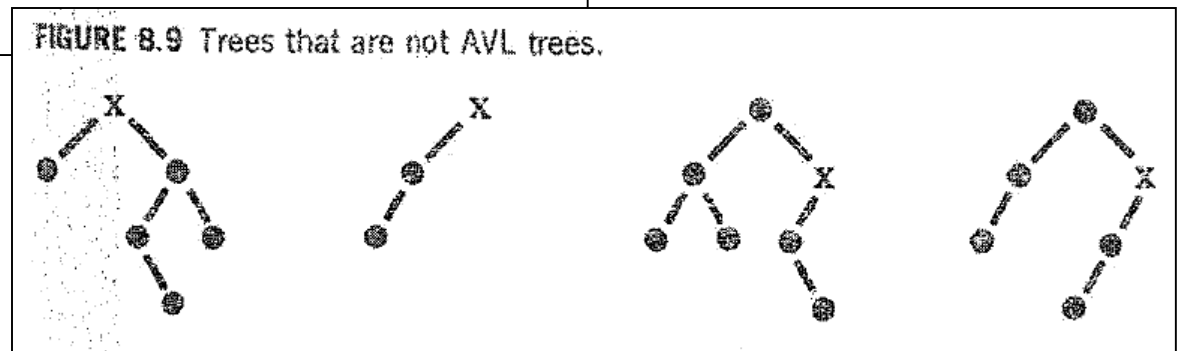
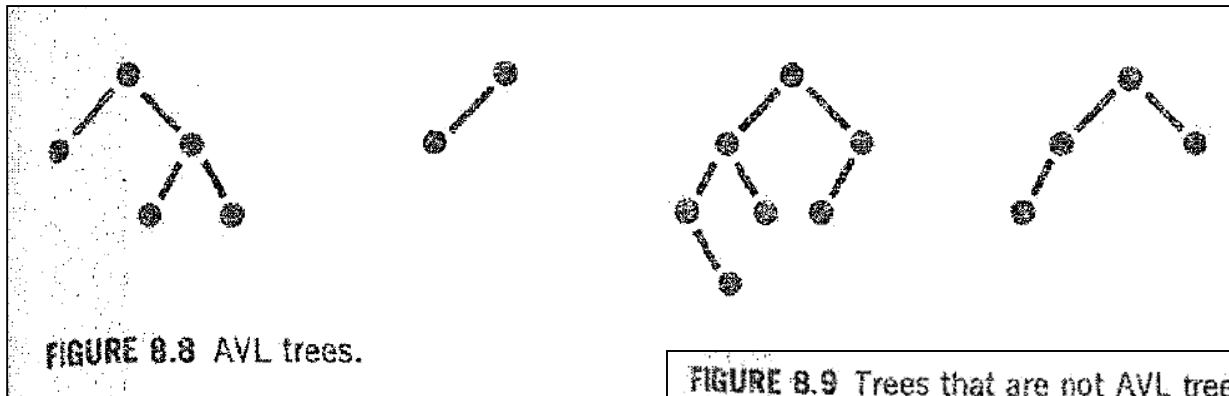
Alguns nós requerem agora 9 acessos para recuperação!!!

Embora não seja mais necessário ordenar o arquivo, estamos realizando acessos adicionais

É necessário reorganizar a árvore de forma que a mesma fique balanceada: Árvores AVL

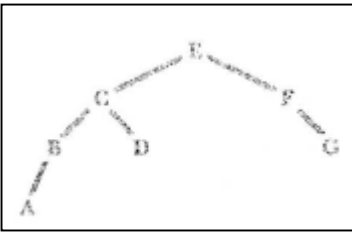
Solução por Árvores AVL

- Árvore de altura balanceada
 - A diferença de altura entre duas sub-árvores compartilhando a mesma raiz é no máximo 1



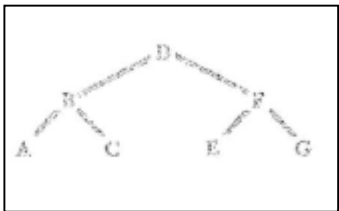
Solução por Árvores AVL

- Assim, garante-se um desempenho mínimo na busca*



$$1.44 * \log_2(N+2)$$

$$1.44 * \log_2(1.000.000+2) = 28 \text{ acessos}$$



(árvore binária completa perfeitamente balanceada: $\log_2(N+1)$
= $\log_2(1.000.000+1) = 20$ acessos)

- Entretanto, manter uma árvore AVL envolve o uso de uma das 4 possíveis rotações (inserções/remoções)

*Pior Caso



Solução por Árvores AVL

- Árvores AVL são uma boa alternativa se considerarmos o problema da ordenação, pois não requerem a ordenação do índice e sua reorganização sempre que houver nova inserção
- Por outro lado, as soluções vistas até agora não resolvem o problema do número excessivo de acessos a disco



Solução por Árvores Binárias Paginadas (Paged Binary Trees)

- A busca por uma posição específica do disco é muito lenta
- Por outro lado, uma vez encontrada a posição, pode-se ler uma grande quantidade de registros sequencialmente a um custo relativamente pequeno



Solução por Árvores Binárias Paginadas (Paged Binary Trees)

- Esta combinação de busca (seek) lenta e transferência rápida sugere a noção de página:
 - em um sistema "paginado" você não incorre no custo de fazer um "seek" para recuperar apenas alguns bytes;
 - ao invés disso, uma vez realizado um seek, que consome um tempo considerável, todos os registros em uma mesma "página" do arquivo são lidos;
 - esta página pode conter um número bastante grande de registros, e se por acaso o próximo registro a ser recuperado estiver na mesma página, você economizou um acesso a disco.

Solução por Árvores Binárias Paginadas (Paged Binary Trees)

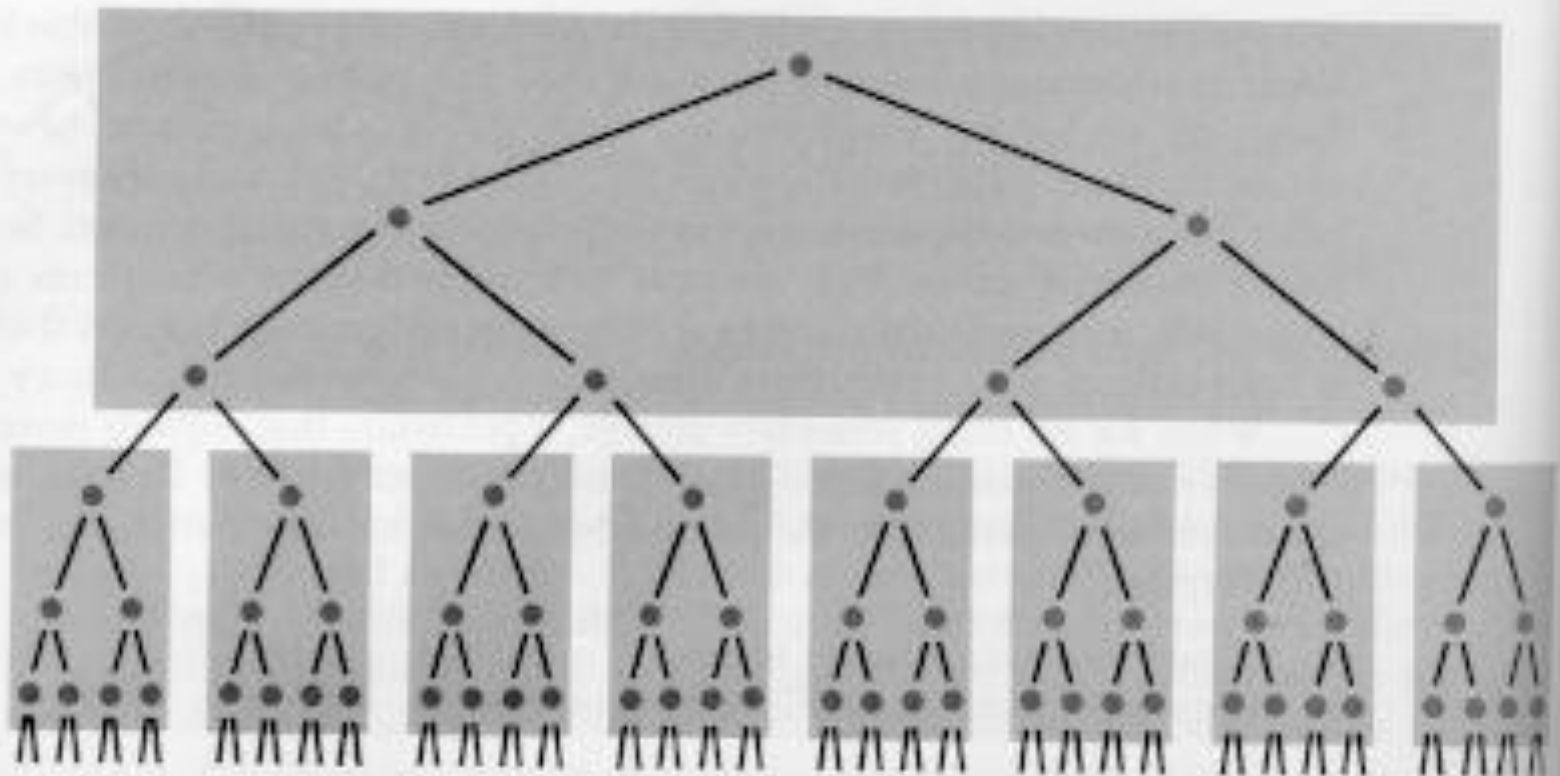


FIGURE 8.12 Paged binary tree.



Solução por Árvores Binárias Paginadas (Paged Binary Trees)

- Nessa árvore de 9 páginas, quaisquer dos 63 registros podem ser acessados em, no máximo, 2 acessos
- Se a árvore é estendida com um nível de paginação adicional, adicionamos 64 novas páginas, e poderemos encontrar qualquer uma das 511 chaves armazenadas com apenas 3 seeks (quantos seeks seriam necessários em uma busca binária?)



Solução por Árvores Binárias Paginadas (Paged Binary Trees)

- Se cada página dessa árvore ocupar 8KB, permitindo a armazenagem de 511 pares chave-referência,
- e cada página contiver uma árvore completa perfeitamente balanceada,
- então a árvore toda pode armazenar um total de 134.217.727 chaves, sendo que qualquer delas pode ser acessada em, no máximo, 3 acessos ao disco.



Solução por Árvores Binárias Paginadas (Paged Binary Trees)

- Pior caso para:

- árvore binária completa perfeitamente balanceada:

$$\log_2(N+1)$$

$$\log_2 (134.217.727 + 1) = 27 \text{ acessos}$$

- versão em páginas:

$$\log_{k+1}(N+1)$$

$$\log_{511+1} (134.217.727 + 1) = 3 \text{ acessos}$$

N é o número total de chaves; k é o número de chaves armazenadas em uma página



Solução por Árvores Binárias Paginadas (Paged Binary Trees)

- Preços a pagar:
 - maior tempo na transmissão de grandes quantidades de dados, e, mais sério,
 - a necessidade de manutenção da organização da árvore.



Problema da Construção Top-Down de Árvores Paginadas

- Construir uma árvore paginada é relativamente simples se temos todo o conjunto de chaves antes de iniciar a construção, pois sabemos que temos de começar com a chave do meio (considerando que as mesmas estão ordenadas) para garantir que o conjunto de chaves seja dividido de forma balanceada (sabe-se que chave deve ser colocada na raiz).



Problema da Construção Top-Down de Árvores Paginadas

- Entretanto, a situação se complica se estamos recebendo as chaves em uma sequência aleatória e construindo a árvore a medida em que as chaves chegam.
- Considere o problema de construir uma árvore binária paginada, com páginas contendo no máximo 3 chaves.

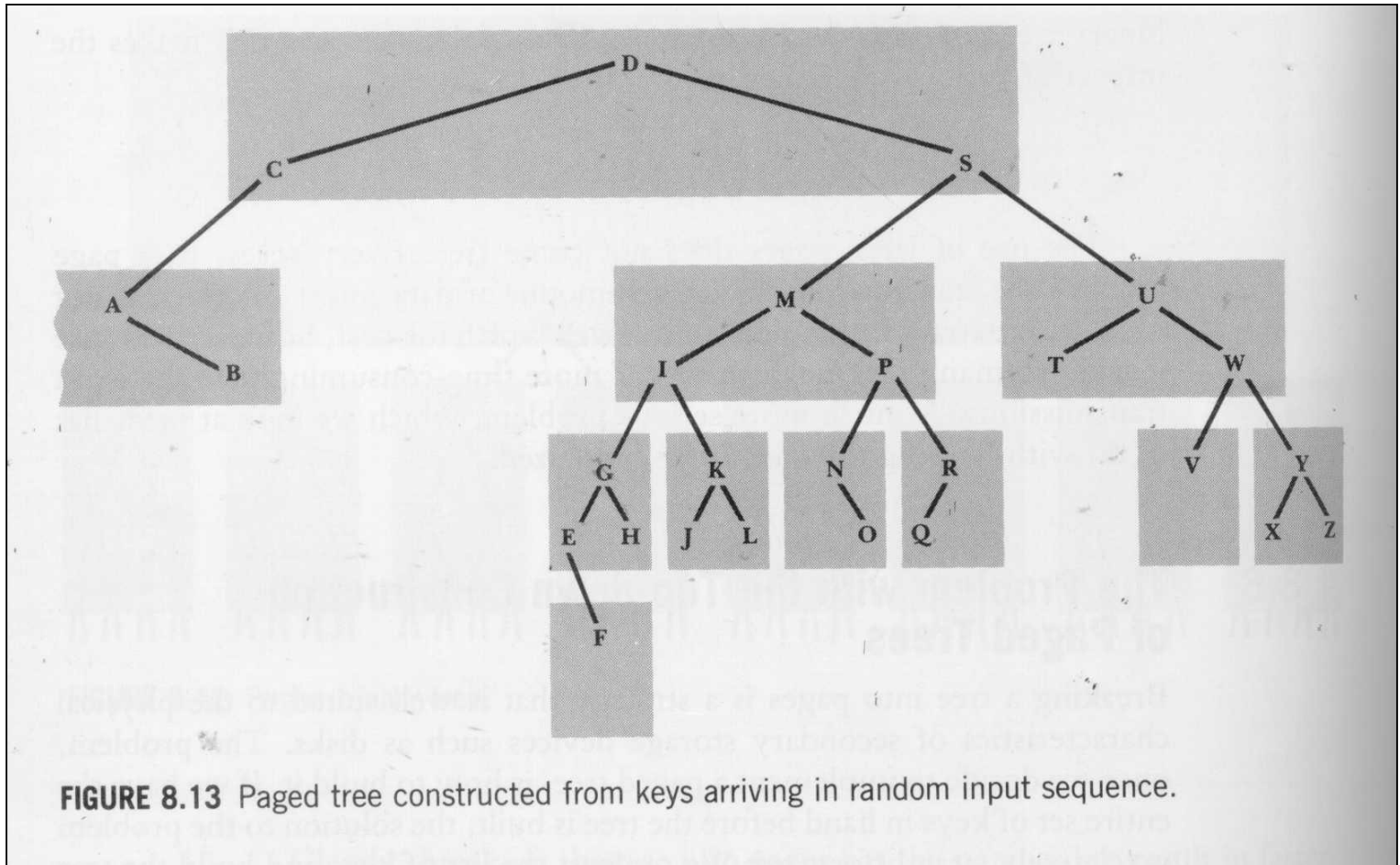


Problema da Construção Top-Down de Árvores Paginadas

- Suponha que o conjunto de dados consiste em letras do alfabeto, que serão fornecidas na seguinte ordem:

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Problema da Construção Top-Down de Árvores Paginadas





Problema da Construção Top-Down de Árvores Paginadas

- A construção foi feita top-down, a partir da raiz, sendo que, cada vez que uma chave é inserida a árvore dentro da página é rotacionada, sempre que necessário, para manter o balanceamento.
- A construção a partir da raiz implica em que as chaves iniciais estarão necessariamente na raiz.



Problema da Construção Top-Down de Árvores Paginadas

- Nesse exemplo, C e D não deveriam estar no topo, e acabaram desbalanceando a árvore.
- Esta árvore não está muito ruim, mas o que aconteceria se as chaves fossem fornecidas em ordem alfabética?



Problema da Construção Top-Down de Árvores Paginadas

- Uma vez que as chaves erradas foram colocadas na raiz de uma árvore ou sub-árvore, o que fazer?
- Não existem resposta simples...; entretanto, não dá para rotacionar páginas como rotacionamos chaves individuais!



Problema da Construção Top-Down de Árvores Paginadas

- Questões:

- Como garantir que as chaves na página raiz sejam boas separadoras, i.e., dividam o conjunto de chaves de maneira balanceada?
- Como impedir o agrupamento de chaves que não deveriam estar na mesma página (como C, D e S, por exemplo)
- Como garantir que cada página contenha um número mínimo de chaves?



Construção Bottom-Up: Árvores-B

- Propôs-se que as árvores fossem construídas de baixo para cima.
- Desta forma, as chaves na raiz da árvore emergem naturalmente. Uma ideia elegante e poderosa.
 - Não é necessário setar a raiz e depois encontrar formas de alterá-la



Árvores-B

- Em uma árvore-B, uma página, ou um nó, consiste de uma sequência ordenada de chaves e um conjunto de ponteiros.
- Não existe nenhuma árvore explícita dentro de um nó, como no caso das árvores binárias paginadas.



Árvores-B

- Árvores-B permitem a recuperação em tempo proporcional a $O(\log_m n)$, na qual m é o tamanho do nó (página).
- Portanto, se $m=64$ e $n=1.000.000$, é possível encontrar a chave em não mais do que 4 acessos.
 - Busca binária = 20 acessos
- Além disso, as operações de inserção e remoção são tão rápidas quanto a busca (a manutenção também é eficiente).



Árvores-B

- O número de ponteiros em um nó excede o número de chaves em 1.
- O número máximo de ponteiros que podem ser armazenados em um nó representa a **ordem** da árvore-B.
- O número máximo de ponteiros é igual ao número máximo de descendentes de um nó.

Árvores-B

- Árvores-B possuem um crescimento de baixo para cima (bottom-up).
- Para isso, quando um nó se torna cheio ele é dividido (splitting) e uma chave é promovida (promoting).
- Seja a seguinte página inicial de uma árvore-B de ordem 8, que armazena 7 chaves.



FIGURE 8.14 Initial leaf of a B-tree with a page size of seven.

Árvores-B

- Todos os ponteiros dos nós folhas indicam fim de lista (-1), ou seja, não levam a outras páginas da árvore.
- Em aplicações reais existem outras informações associadas aos nós; assim outros ponteiros podem existir dentro da página.

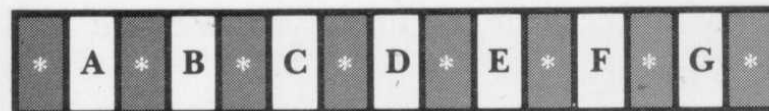


FIGURE 8.14 Initial leaf of a B-tree with a page size of seven.



Árvores-B: Splitting e Promoting

- Esta folha que, coincidentemente, é também a raiz da árvore, está cheia. Como inserir uma nova chave, digamos J?

Árvores-B: Splitting

- Dividimos (split) o nó folha em dois nós folhas, distribuindo as chaves igualmente entre os nós.

FIGURE 8.15 Splitting the leaf to accommodate the new *J* key.





Árvores-B: Promoting

- Temos agora duas folhas: precisamos criar uma nova raiz. Fazemos isso "promovendo", ou "subindo", uma das chaves que estão nos limites de separação das folhas.
- Nesse caso, "promovemos" a chave E para a raiz.

Árvores-B: Promoting

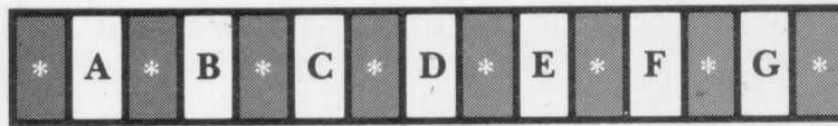


FIGURE 8.14 Initial leaf of a B-tree with a page size of seven.

FIGURE 8.15 Splitting the leaf to accommodate the new *J* key.

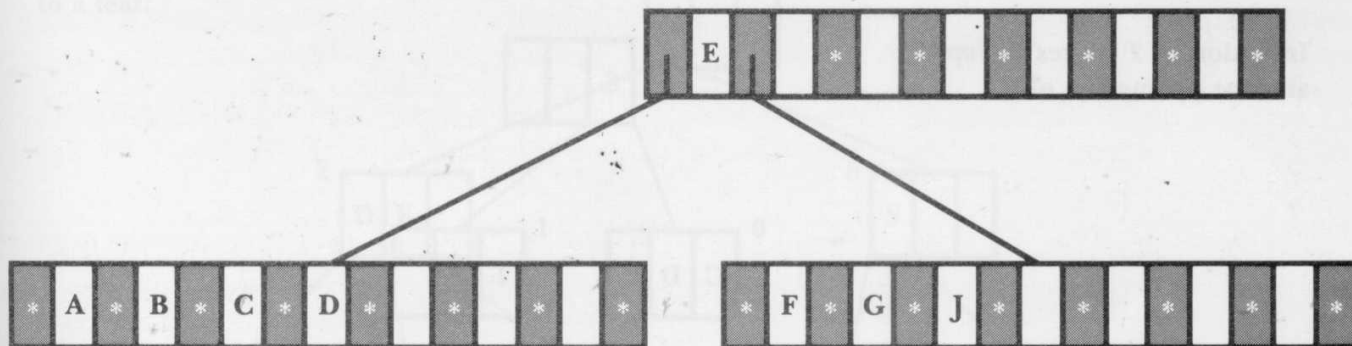


FIGURE 8.16 Promotion of the *E* key into a root node.



Árvores-B: Exemplo

- Vamos fazer um exemplo. Consiste na inserção das seguintes chaves:

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

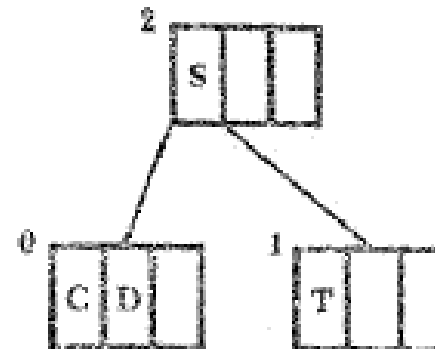
- Considere uma árvore-B de ordem 4.

Árvores-B: Exemplo

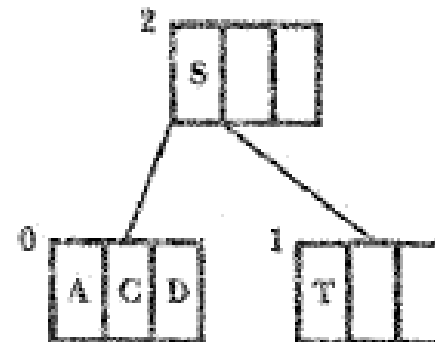
Insertion of *C*, *S*, and *D*
into the initial page:



Insertion of *T* forces the split
and the promotion of *S*:

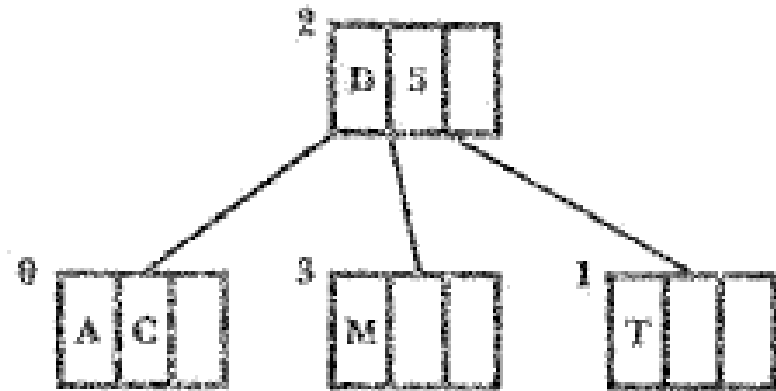


A added without incident:

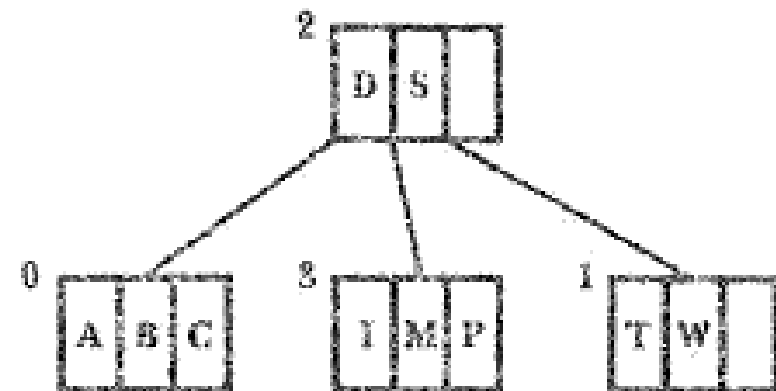


Árvores-B: Exemplo

Insertion of *M* forces another split and the promotion of *D*:

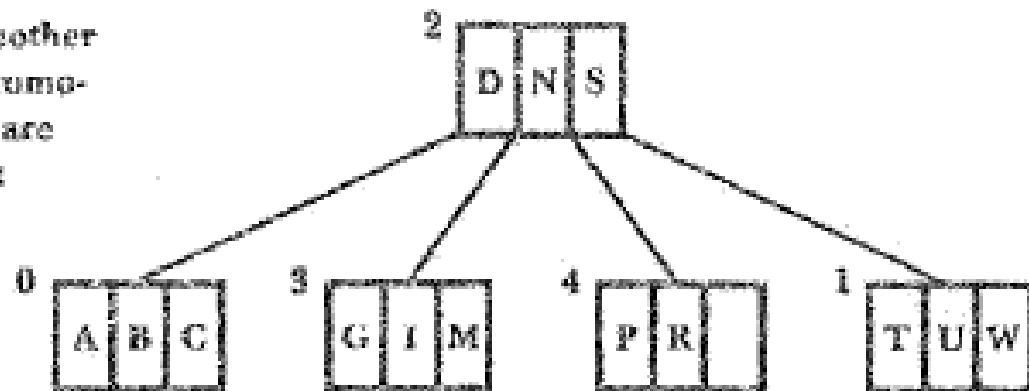


P, *I*, *B*, and *W* inserted into existing pages:



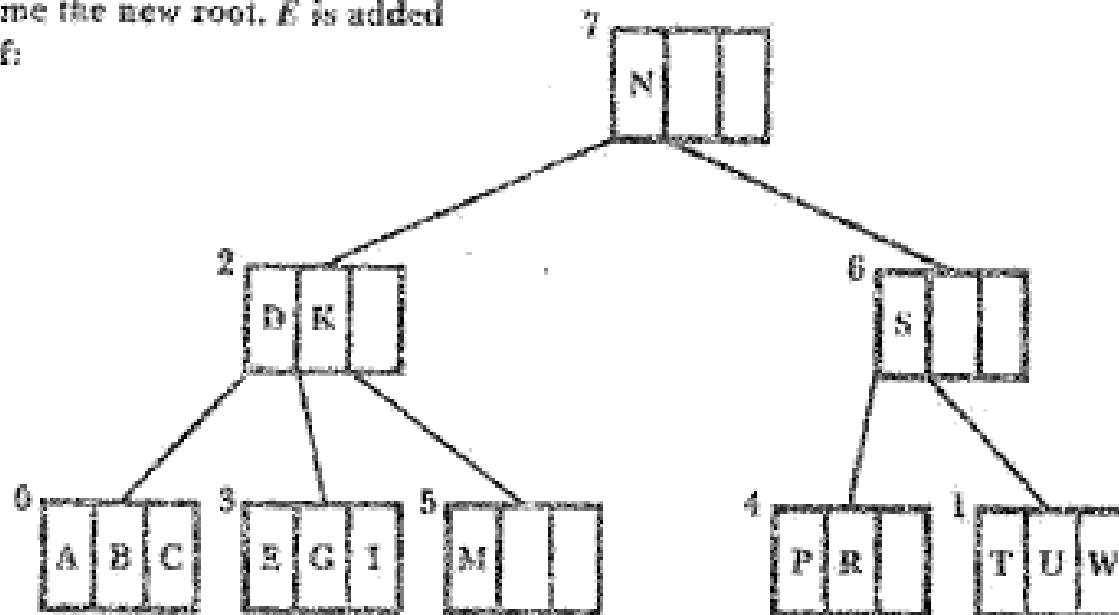
Árvores-B: Exemplo

Insertion of *N* causes another split, followed by the promotion of *N*. *G*, *U*, and *R* are added to existing pages:



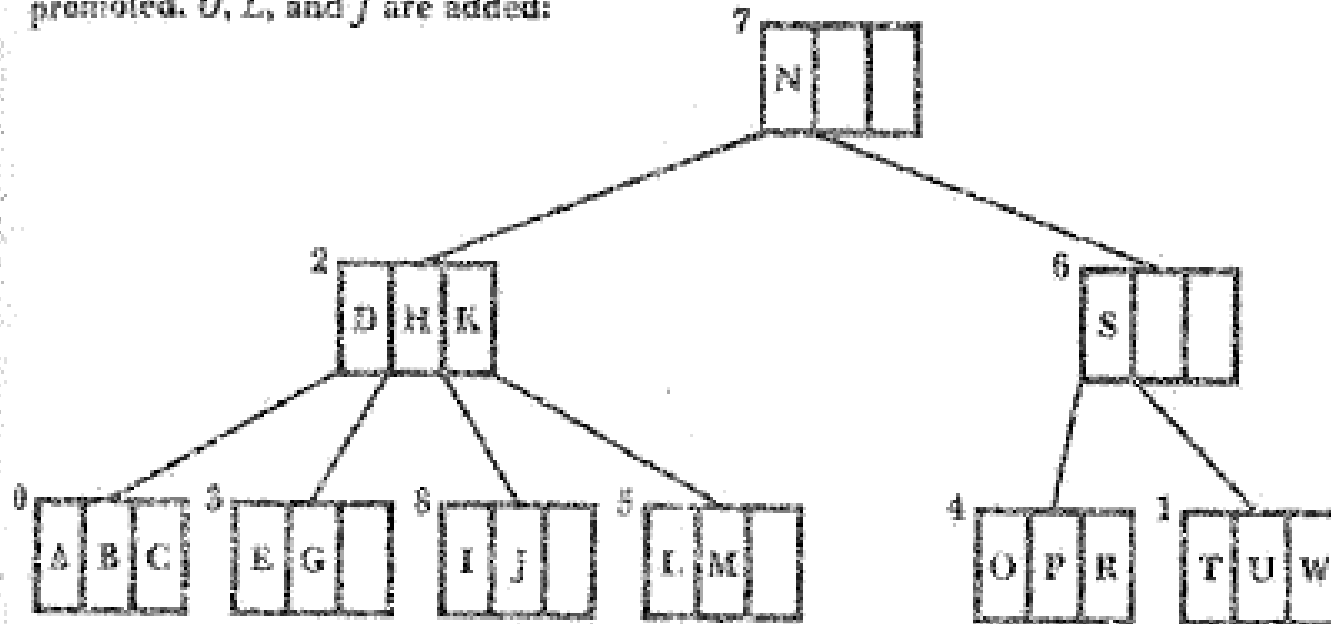
Árvores-B: Exemplo

Insertion of *K* causes a split at leaf level, followed by the promotion of *K*. This causes a split of the root. *N* is promoted to become the new root. *E* is added to a leaf:



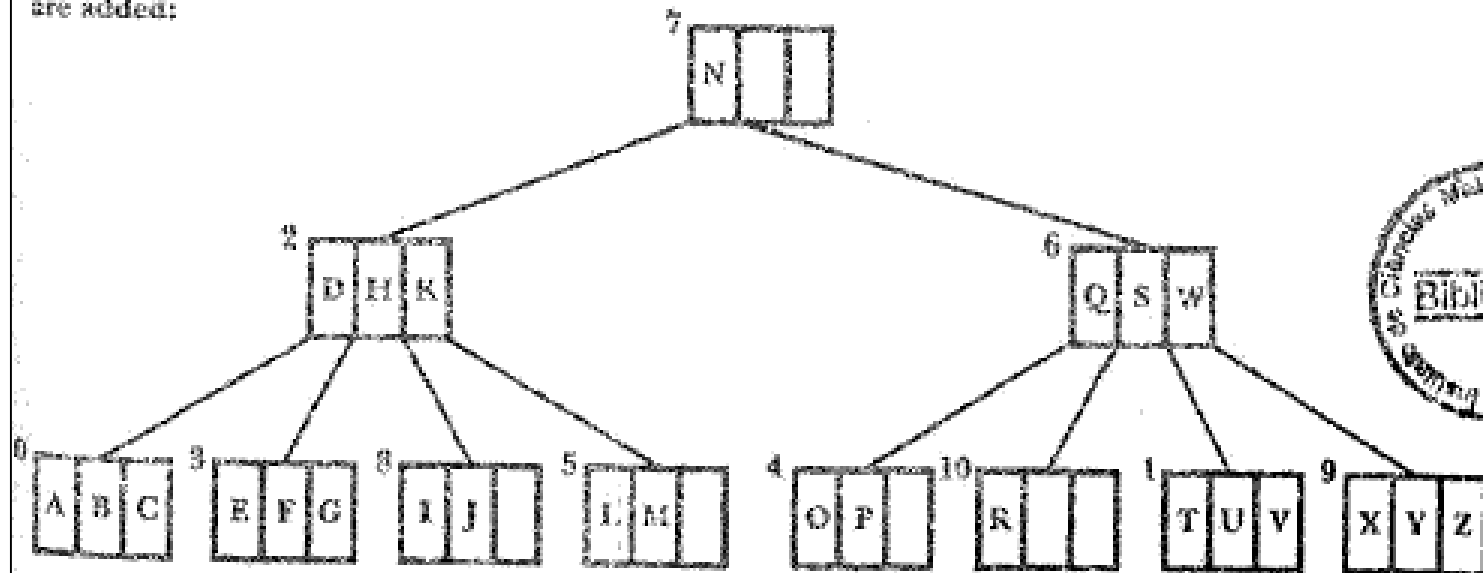
Árvores-B: Exemplo

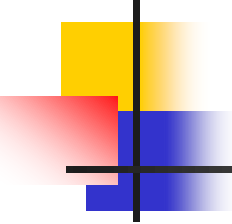
Insertion of *H* causes a leaf to split. *H* is promoted. *O*, *L*, and *J* are added:



Árvores-B: Exemplo

Insertion of Y and Q force two more leaf splits and promotions. Remaining letters are added:





Árvores-B: Exemplo

- Observe que a árvore é perfeitamente balanceada com relação à altura
 - O comprimento do caminho entre a raiz e qualquer nó folha é sempre o mesmo
- Observe também que as chaves "promovidas" sempre são boas separadoras



Árvores-B: Implementação

- Existem quatro operações principais sobre árvores-B:
 - Pesquisa;
 - Percurso;
 - Inserção e,
 - Remoção.

→ <http://www.csanimated.com/animation.php?t=B-tree>

→ <https://www.cs.usfca.edu/~galles/visualization/BTree.html>

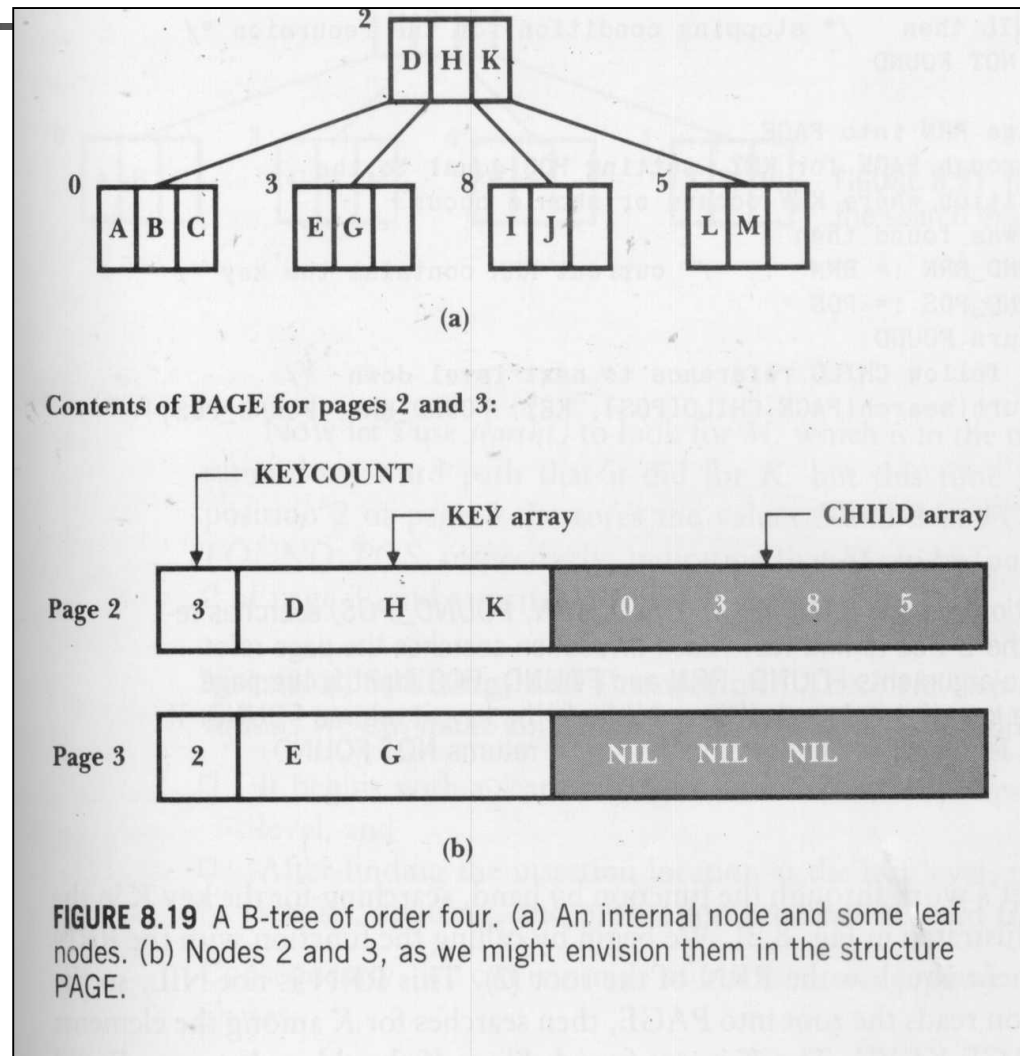


Árvores-B: Implementação

- Estrutura da página
 - Conjunto de registros de tamanho fixo
 - Cada registro contém uma página da árvore

```
In C:
struct BTPAGE {
    short    KEYCOUNT;           /* number of keys stored in PAGE */
    char     KEY[MAXKEYS];        /* the actual keys                */
    short    CHILD[MAXKEYS+1];    /* RRNs of children              */
} PAGE;
```

Árvores-B: Implementação





Árvores-B: Pesquisa

- A maioria dos algoritmos referentes á árvores-B são:
 - Recursivos;
 - Trabalham em duas etapas, operando intercaladamente nas páginas inteiras e dentro das páginas.



Árvores-B: Pesquisa

- A operação de pesquisa pode ser organizada em dois passos:
 - Pesquisa externa ao nó:
 - Desce pela árvore à procura da chave de interesse.
 - Pesquisa interna ao nó:
 - Realiza uma busca sequencial ou binária entre as chaves dentro de um nó.



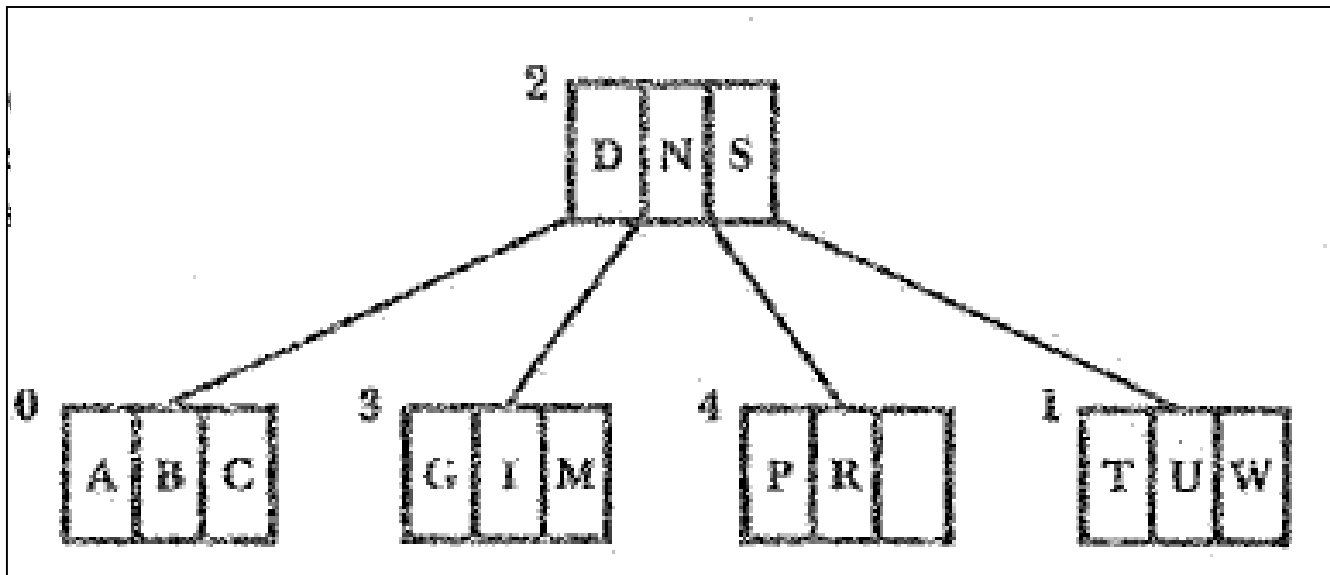
Árvores-B: Pesquisa

```
FUNCTION: search (RRN, KEY, FOUND_RRN , FOUND_POS)

    if RRN == NIL then    /* stopping condition for the recursion */
        return NOT FOUND
    else
        read page RRN into PAGE
        look through PAGE for KEY, setting POS equal to the
            position where KEY occurs or should occur.
        if KEY was found then
            FOUND_RRN := RRN      /* current RRN contains the key */
            FOUND_POS := POS
            return FOUND
        else /* follow CHILD reference to next level down */
            return(search(PAGE.CHILD[POS], KEY, FOUND_RRN, FOUND_POS))
        endif
    endif
end FUNCTION
```

FIGURE 8.20 Function *search (RRN, KEY, FOUND_RRN, FOUND_POS)* searches recursively through the B-tree to find KEY. Each invocation searches the page referenced by RRN. The arguments FOUND_RRN and FOUND_POS identify the page and position of the key, if it is found. If *search()* finds the key, it returns FOUND. If it goes beyond the leaf level without finding the key, it returns NOT FOUND.

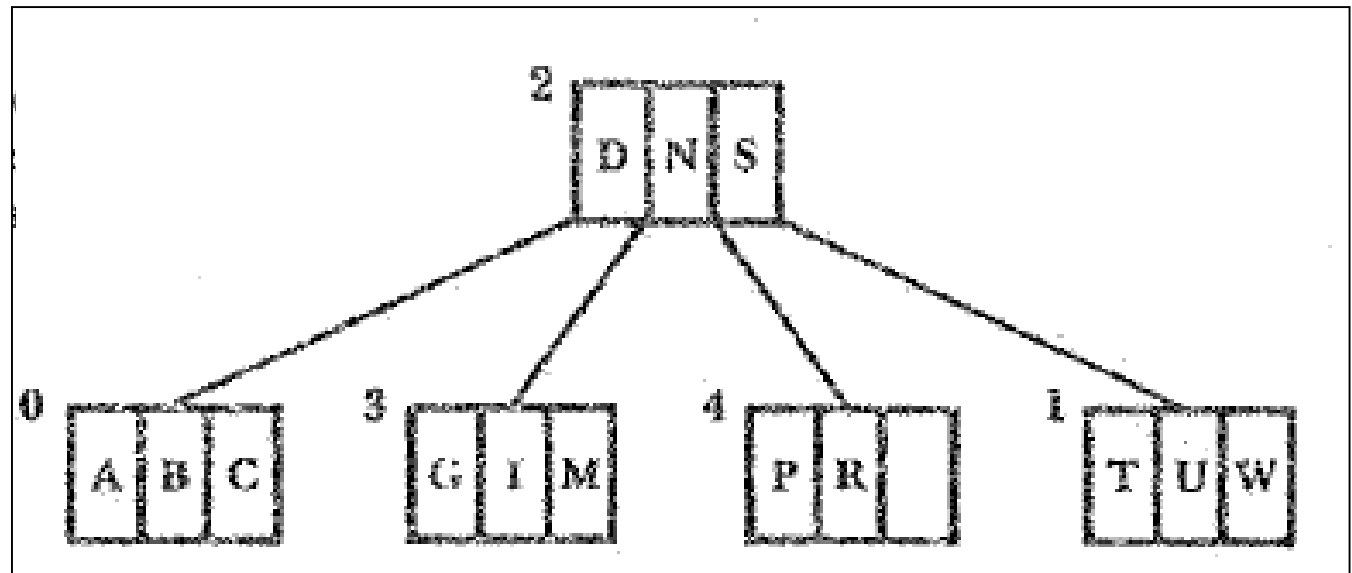
Árvores-B: Pesquisa



Fazer exemplo com K e M

Árvores-B: Percurso

- Útil para imprimir as chaves em ordem crescente.



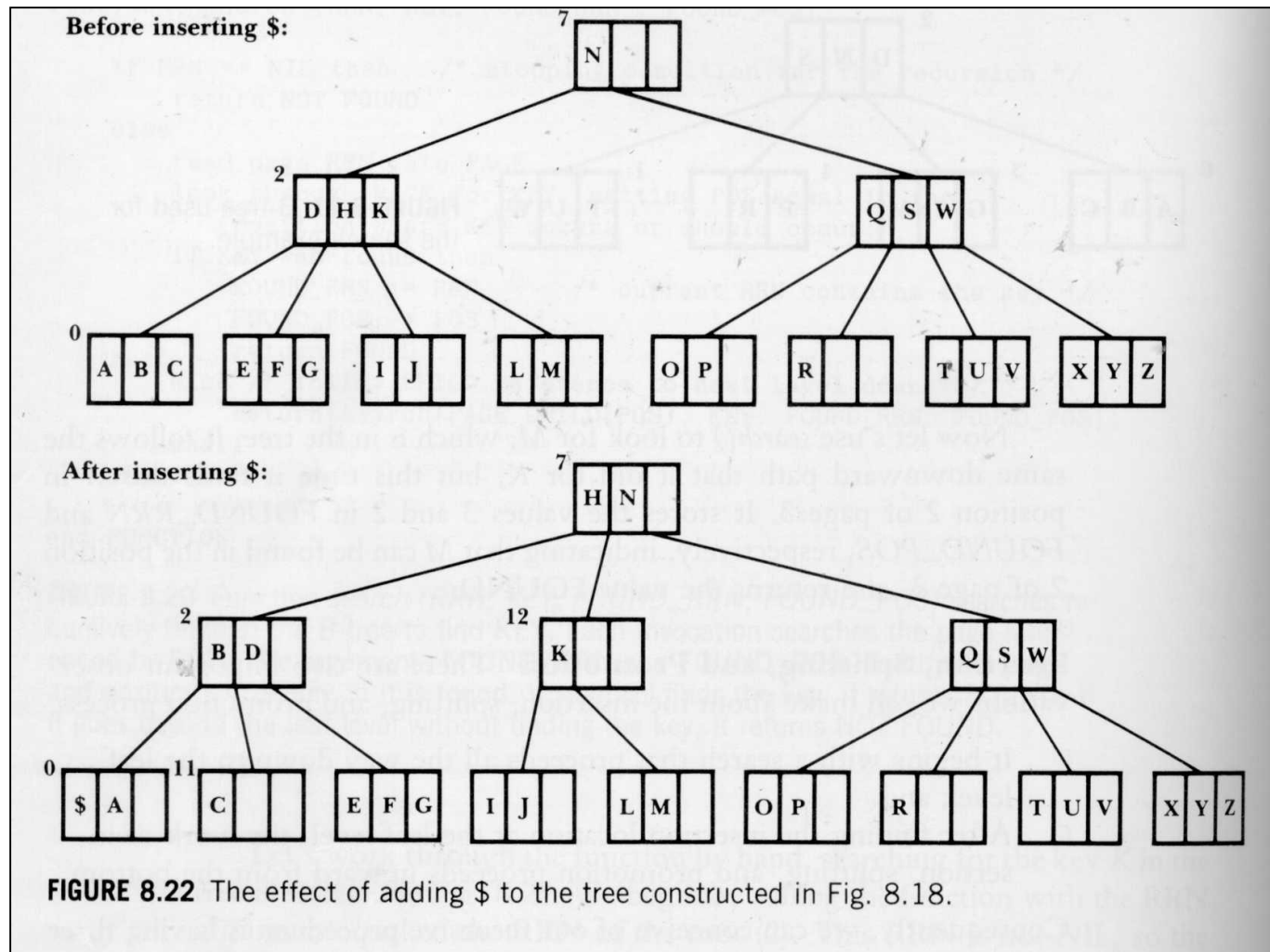
Como seria o algoritmo?



Árvores-B: Inserção

- O processo de inserção:
 - começa com uma busca, a partir do nó raiz, que continua até atingir um nó folha; e
 - uma vez localizada a posição de inserção (em uma folha), pode ser necessário realizar divisões e promoções, de baixo para cima.

Árvores-B: Inserção





Á

```

FUNCTION: insert (CURRENT_RRN, KEY, PROMO_R_CHILD, PROMO_KEY)

    if CURRENT_RRN = NIL then          /* past bottom of tree */
        PROMO_KEY := KEY
        PROMO_R_CHILD := NIL
        return PROMOTION              /* promote original key and NIL */
    else
        read page at CURRENT_RRN into PAGE
        search for KEY in PAGE.
        let POS := the position where KEY occurs or should occur.

        if KEY found then
            issue error message indicating duplicate key
            return ERROR

        RETURN_VALUE := insert(PAGE.CHILD[POS], KEY, P_B_RRN, P_B_KEY)


        if RETURN_VALUE == NO PROMOTION or ERROR then
            return RETURN_VALUE

        elseif there is space in PAGE for P_B_KEY then
            insert P_B_KEY and P_B_RRN (promoted from below) in PAGE
            return NO PROMOTION
        else
            split(P_B_KEY, P_B_RRN, PAGE, PROMO_KEY, PROMO_R_CHILD, NEWPAGE)
            write PAGE to file at CURRENT_RRN
            write NEWPAGE to file at rrn PROMO_R_CHILD
            return PROMOTION          /* promoting PROMO_KEY and PROMO_R_CHILD */
        endif
    endif

end FUNCTION

```

FIGURE 8.24 Function *insert* (*CURRENT_RRN*, *KEY*, *PROMO_R_CHILD*, *PROMO_KEY*) inserts a *KEY* in a B-tree. The insertion attempt starts at the page with relative record number *CURRENT_RRN*. If this page is not a leaf page, the function calls itself recursively until it finds *KEY* in a page or reaches a leaf. If it finds *KEY*, it issues an error message and quits, returning *ERROR*. If there is space for *KEY* in *PAGE*, *KEY* is inserted. Otherwise, *PAGE* is split. A split assigns the value of the middle key to *PROMO_KEY* and the relative record number of the newly created page to *PROMO_R_CHILD* so insertion can continue on the recursive ascent back up the tree. If a promotion does occur, *insert()* indicates this by returning *PROMOTION*. Otherwise, it returns *NO PROMOTION*.



```
PROCEDURE: split (I_KEY, I_RRN, PAGE, PROMO_KEY, PROMO_R_CHILD, NEWPAGE)
```

```
    copy all keys and pointers from PAGE into a working page that  
        can hold one extra key and child.
```

```
    insert I_KEY and I_RRN into their proper places in the working page.
```

```
    allocate and initialize a new page in the B-tree file to hold NEWPAGE.
```

```
    set PROMO_KEY to value of middle key, which will be promoted after  
        the split.
```

```
    set PROMO_R_CHILD to RRN of NEWPAGE.
```

```
    copy keys and child pointers preceding PROMO_KEY from the working  
        page to PAGE.
```

```
    copy keys and child pointers following PROMO_KEY from the working  
        page to NEWPAGE.
```

```
end PROCEDURE
```

FIGURE 8.25 *Split (I_KEY, I_RRN, PAGE, PROMO_KEY, PROMO_R_CHILD, NEWPAGE)*, a procedure that inserts I_KEY and I_RRN, causing overflow, creates a new page called NEWPAGE, distributes the keys between the original PAGE and NEWPAGE, and determines which key and RRN to promote. The promoted key and RRN are returned via the arguments PROMO_KEY and PROMO_R_CHILD.



Árvores-B: Inserção

```
MAIN PROCEDURE: driver
```

```
    if the B-tree file exists, then
```

```
        open B-tree file
```

```
    else
```

```
        create a B-tree file and place the first key in the root
```

```
    get RRN of root page from file and store it in ROOT
```

```
    get a key and store it in KEY
```

```
    while keys exist
```

```
        if (insert(ROOT, KEY, PROMO_R_CHILD, PROMO_KEY) == PROMOTION) then
```

```
            create a new root page with key := PROMO_KEY, left
```

```
                child := ROOT, and right child := PROMO_R_CHILD
```

```
            set ROOT to RRN of new root page
```

```
            get next key and store it in KEY
```

```
        endwhile
```

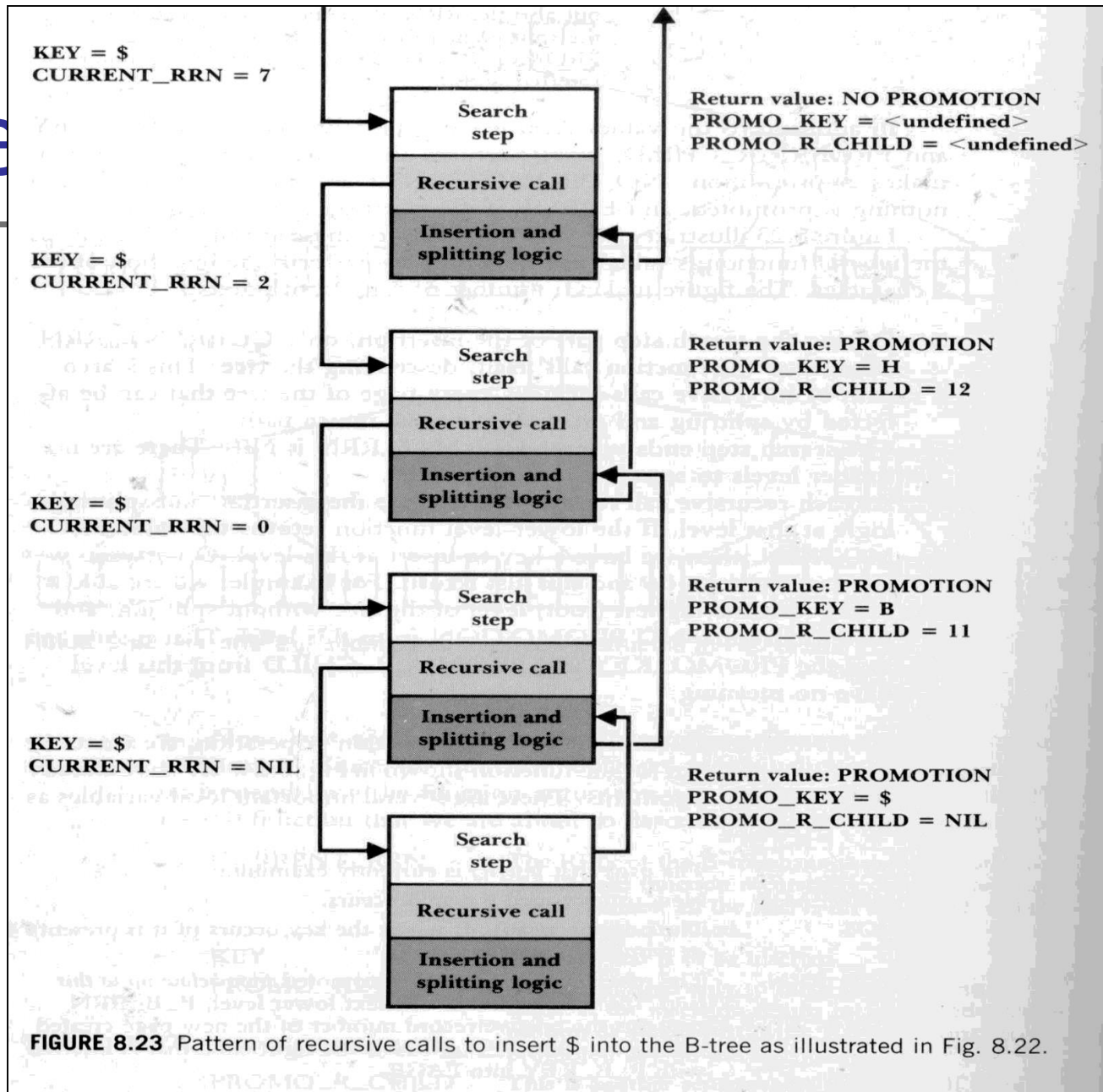
```
    write RRN stored in ROOT back to B-tree file
```

```
    close B-tree file
```

```
end MAIN PROCEDURE
```

FIGURE 8.27 *Driver for building a B-tree.*

Árvore





Definições

- A **ordem** de uma árvore-B é dada pelo número máximo de descendentes que um nó pode possuir.
- Em uma árvore-B de ordem m , o número máximo de chaves em uma página é $m-1$
- **Exemplo:**
 - Uma árvore-B de ordem 8 tem, no máximo, 7 chaves por página.



Definições

- Número mínimo de chaves por nó
 - Quando um nó é dividido na inserção, as chaves são divididas igualmente entre os nós velho e novo. Deste modo, o número mínimo de chaves em um nó é dado por $\lceil m/2 \rceil - 1$ (exceto para a raiz).
 - **Exemplo:** Uma árvore-B de ordem 8, que armazena no máximo 7 chaves por página, tem, no mínimo, 3 chaves por página.



Definições

- Nó folha
 - Os nós folhas são aqueles alocados no nível mais baixo da árvore.



Definição Formal das Propriedades de Árvores-B

- Para uma árvore-B de ordem m :
 - **(1)** Cada nó tem, no máximo, m descendentes;
 - **(2)** Cada nó, exceto a raiz e as folhas, tem no mínimo $\lceil m/2 \rceil$ descendentes;
 - **(3)** A raiz tem, no mínimo, dois descendentes (a menos que seja uma folha também);
 - **(4)** Todas as folhas aparecem em um mesmo nível;
 - **(5)** Um nó que não é folha e possui k descendentes, contém $k-1$ chaves;
 - **(6)** Um nó folha contém, no mínimo $\lceil m/2 \rceil - 1$ e, no máximo, $m-1$ chaves, e nenhum descendente.



Profundidade da Busca no Pior Caso

- É importante entender o relacionamento quantitativo entre o número de chaves armazenadas em uma página e o número de níveis que uma árvore-B pode ter (i.e., a altura da árvore, d).
- **Exemplo:** "Você precisa armazenar 1.000.000 de chaves, e considera utilizar uma árvore-B de ordem 512. Qual o número máximo de acessos necessários para localizar uma chave nesta árvore?"
 - **Ou seja: qual a altura máxima que a árvore pode atingir?**



Profundidade da Busca no Pior Caso

- O pior caso ocorre quando cada página tem apenas o número mínimo de descendentes, e a árvore possui, portanto, altura máxima e largura mínima.
- O número mínimo de descendentes para o nó raiz é 2, sendo que cada um destes nós, por sua vez, possui no mínimo $\lceil m/2 \rceil$ descendentes; ou seja, o segundo nível possui $2 * \lceil m/2 \rceil$ descendentes.
- O terceiro nível, por sua vez, possui no mínimo $2 * \lceil m/2 \rceil * \lceil m/2 \rceil$ descendentes.
- Em geral, para um nível d da árvore, o número mínimo de descendentes é dado por $2 * \lceil m/2 \rceil^{(d-1)}$.

Profundidade da Busca no Pior Caso

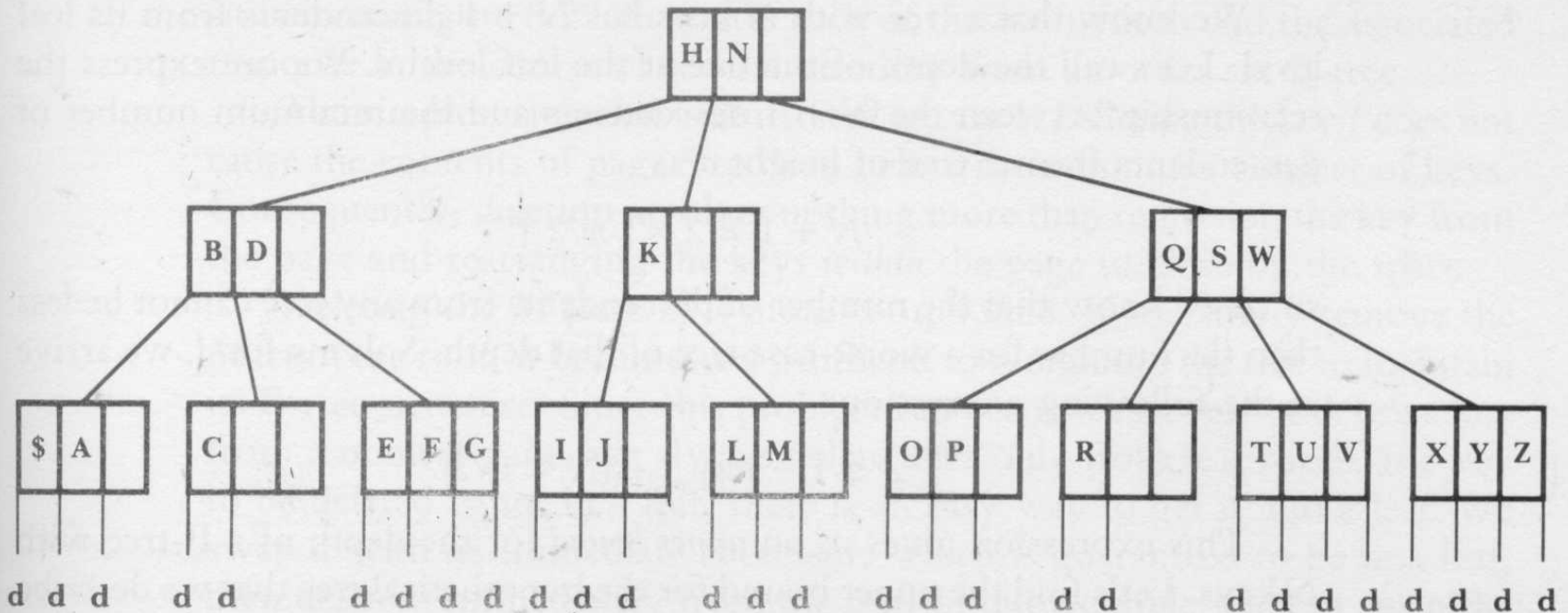


FIGURE 8.28 A B-tree with N keys can have $(N + 1)$ descendents from the leaf level.

O número de descendentes a partir de qualquer nível é uma unidade maior do que o número de chaves contidas no nível mais todas as chaves do nível acima



Profundidade da Busca no Pior Caso

- Uma árvore de n chaves tem $n+1$ descendentes a partir de seu nível mais inferior; as folhas.
- Seja d a profundidade da árvore no nível dos nós folhas. Podemos expressar a relação entre os $n+1$ descendentes e o número mínimo de descendentes de uma árvore de altura d como:

$$n+1 \geq 2^{\lceil m/2 \rceil (d-1)} \equiv$$
$$d \leq 1 + \log_{\lceil m/2 \rceil} ((n+1)/2)$$



Profundidade da Busca no Pior Caso

$$d \leq 1 + \log_{\lceil m/2 \rceil} ((n+1)/2)$$

- **Exemplo:** Para a árvore-B de ordem 512 com 1.000.000 de chaves, tem-se:

$$d \leq 1 + \log_{256}(500.000,5) \leq 3.37$$

- A nossa árvore terá altura 3, no máximo, e no máximo 3 acessos serão necessários para localizar uma chave, o que é um desempenho muito bom.



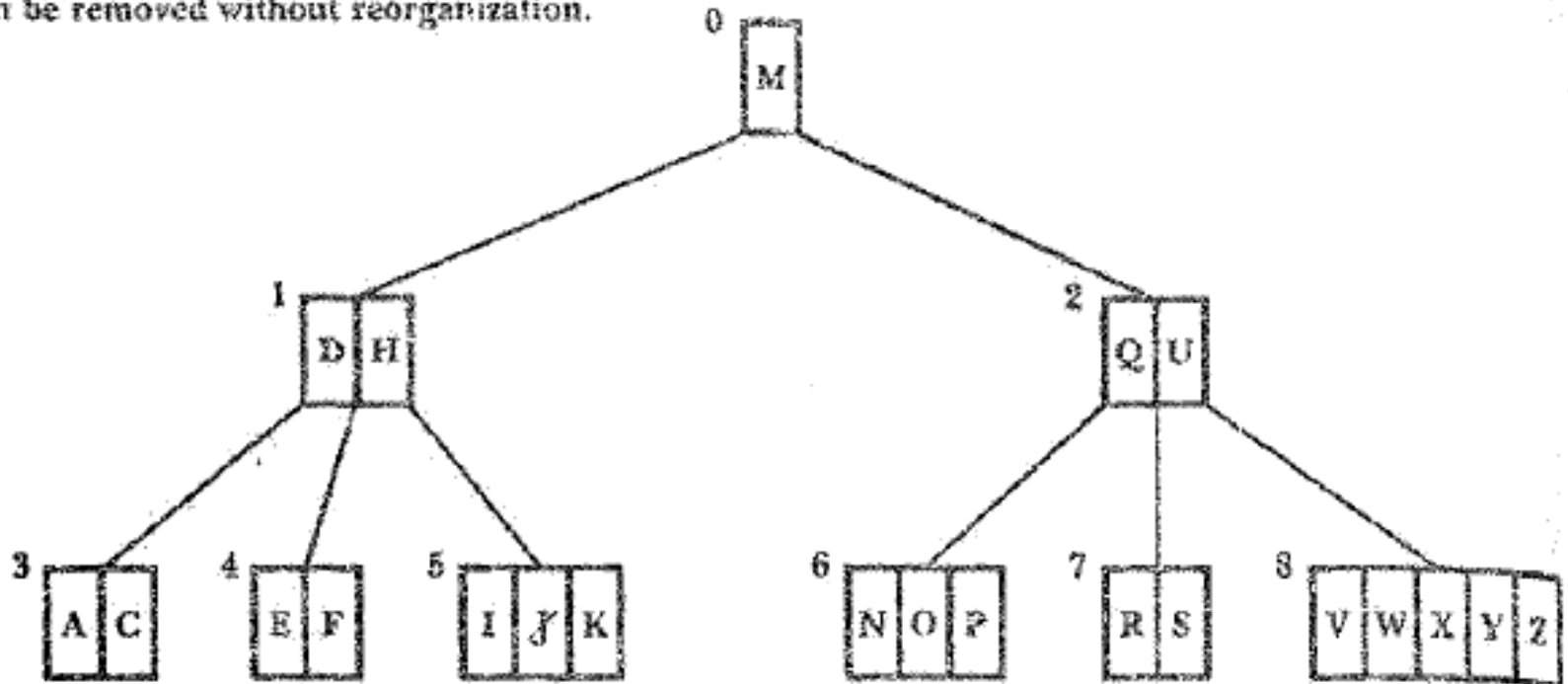
Árvores-B: Remoção

- Para realizar uma remoção em uma árvore-B deve-se preservar a exigência de $\lceil m/2 \rceil - 1$ chaves por nó.
- Como regra geral, todas as chaves são removidas de nós folha:
 - Caso a remoção ocorra em um nó não-folha, a chave sucessora em nó folha deve ser movida para a posição vaga.
- Operações: redistribuição e concatenação

Árvores-B: Remoção

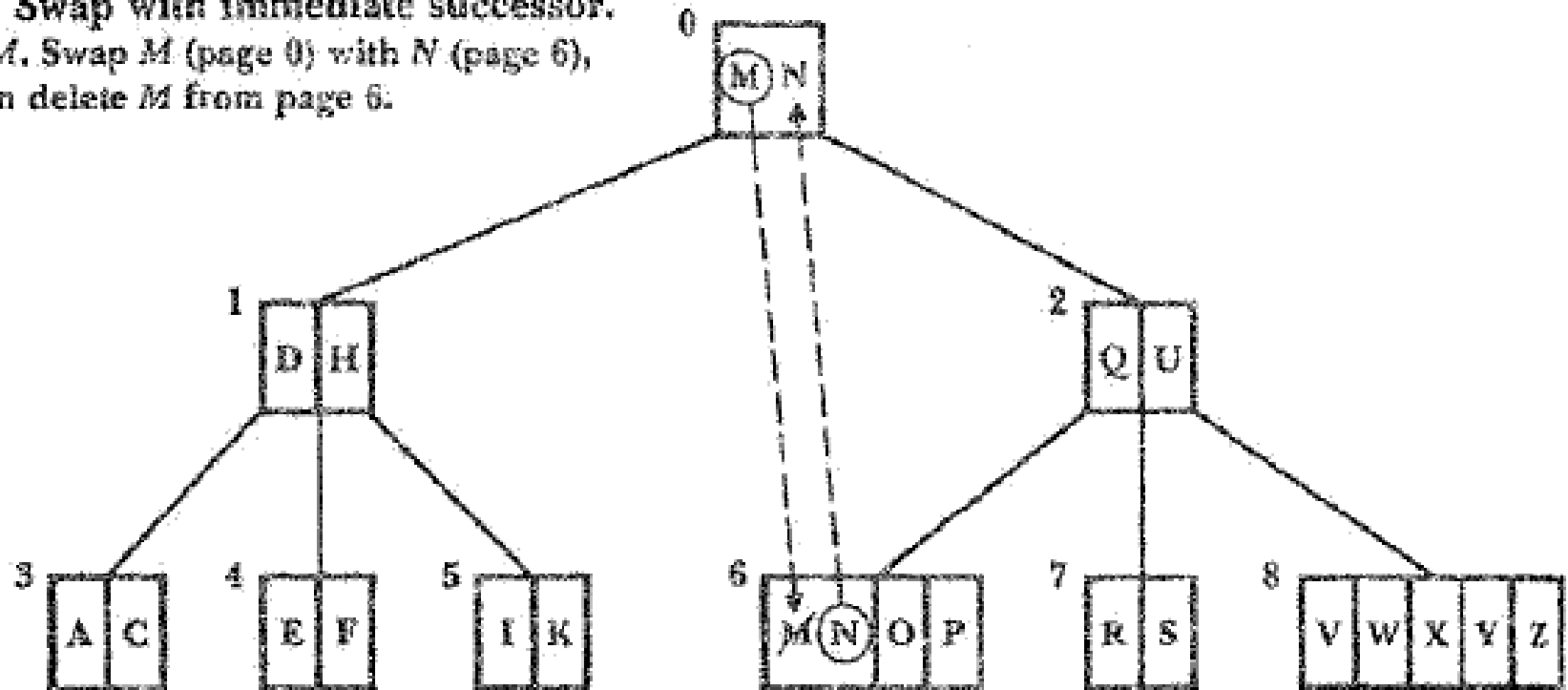
Case 1: No action.

Delete *J* from page 5. Since page 5 has more than the minimum number of keys, *J* can be removed without reorganization.



Árvores-B: Remoção

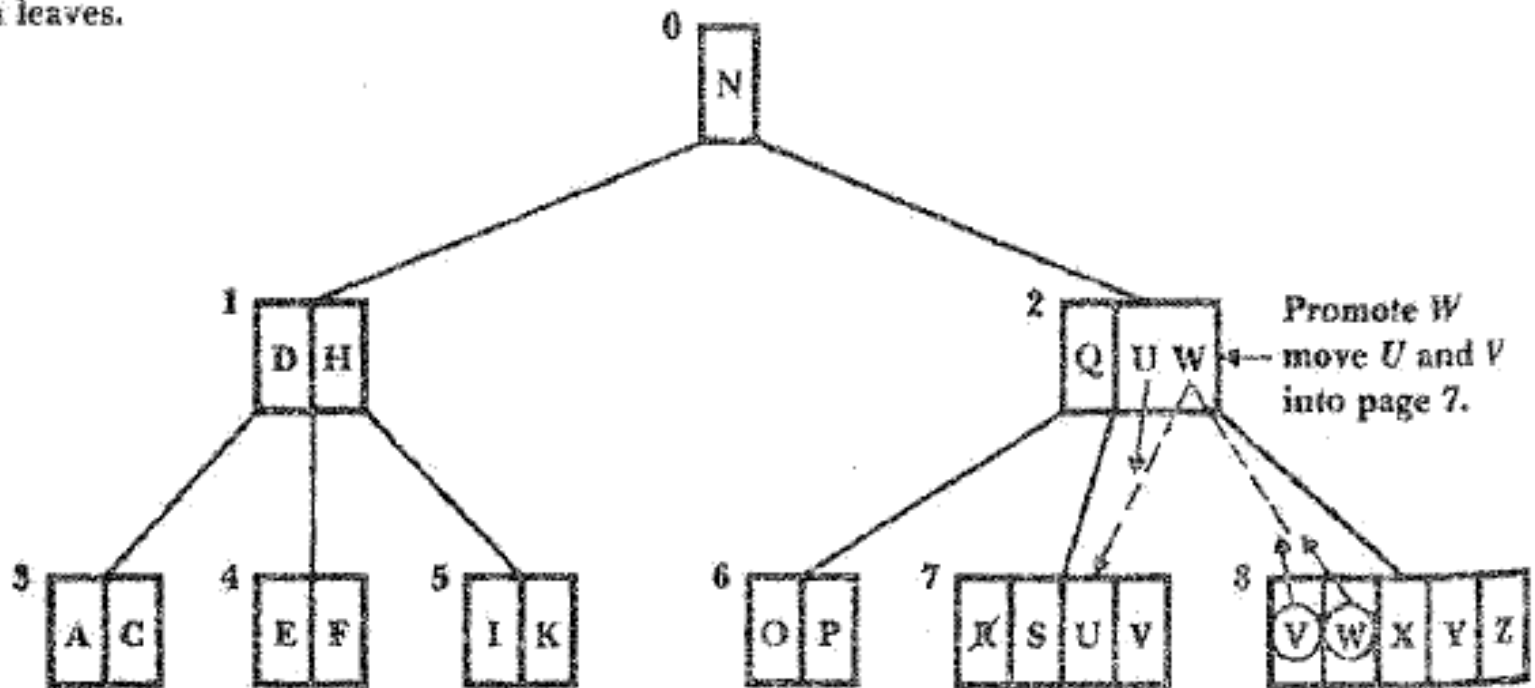
Case 2: Swap with immediate successor.
Delete *M*. Swap *M* (page 0) with *N* (page 6),
and then delete *M* from page 6.



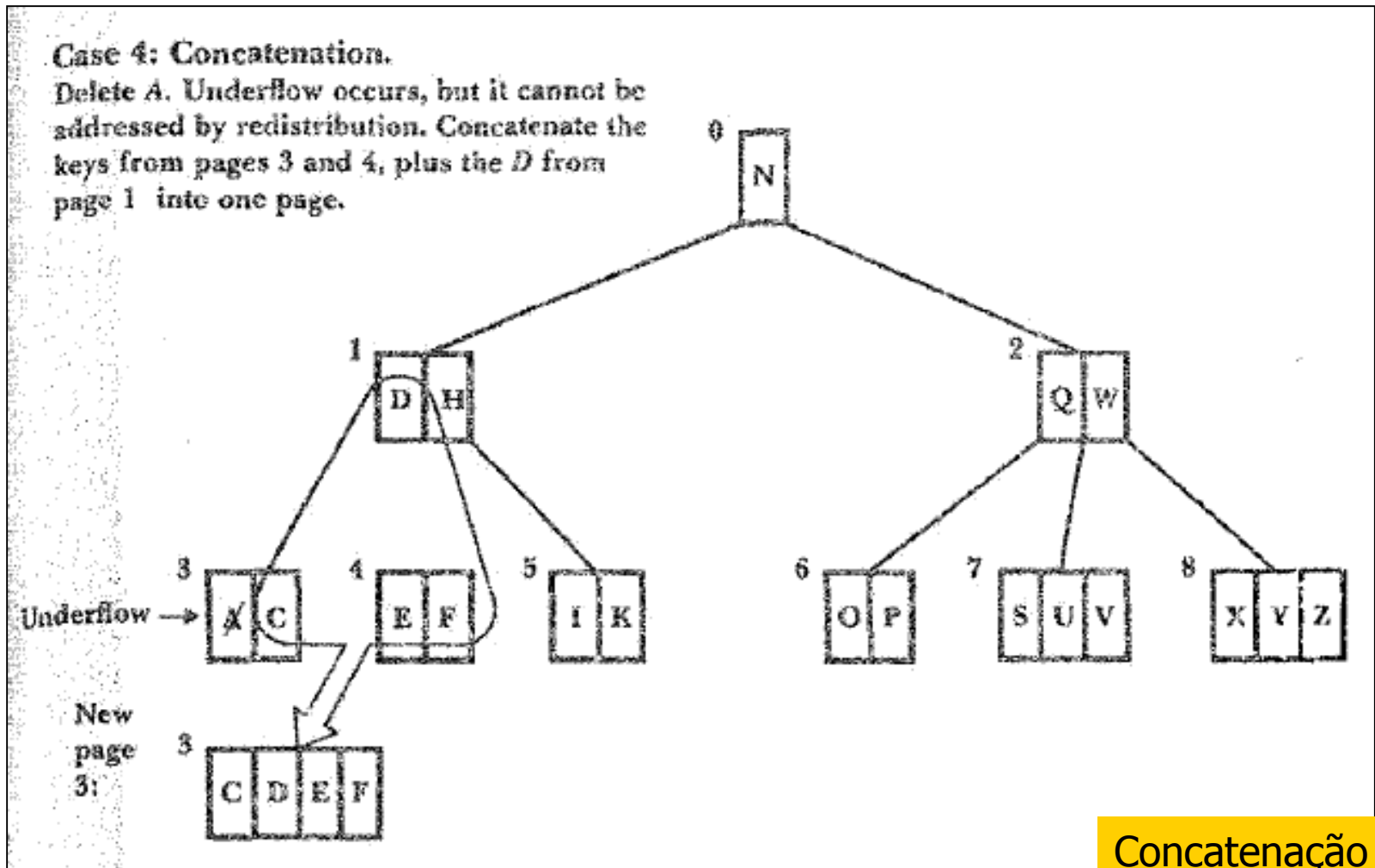
Árvores-B: Remoção

Case 3: Redistribution.

Delete *R*. Underflow occurs. Redistribute keys among pages 2, 7, and 8 to restore balance between leaves.

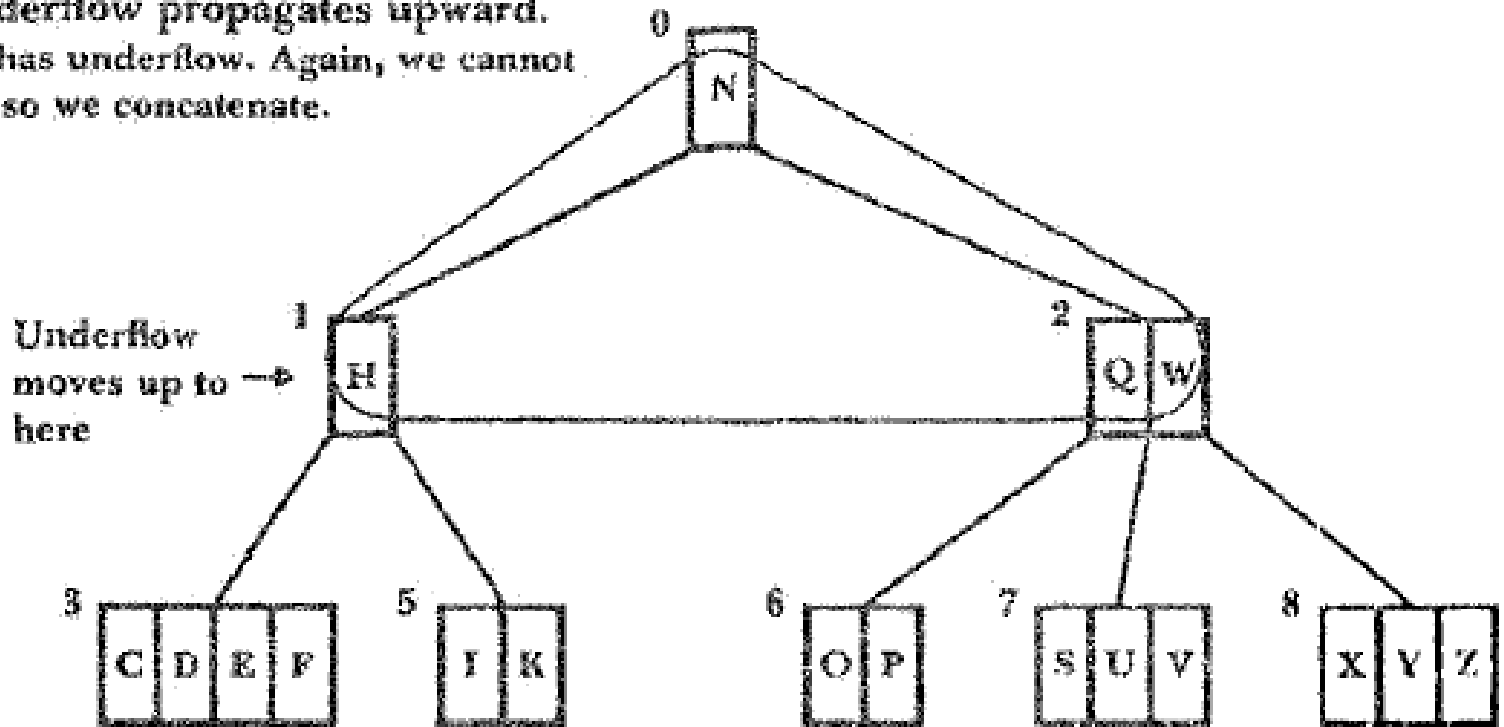


Árvores-B: Remoção



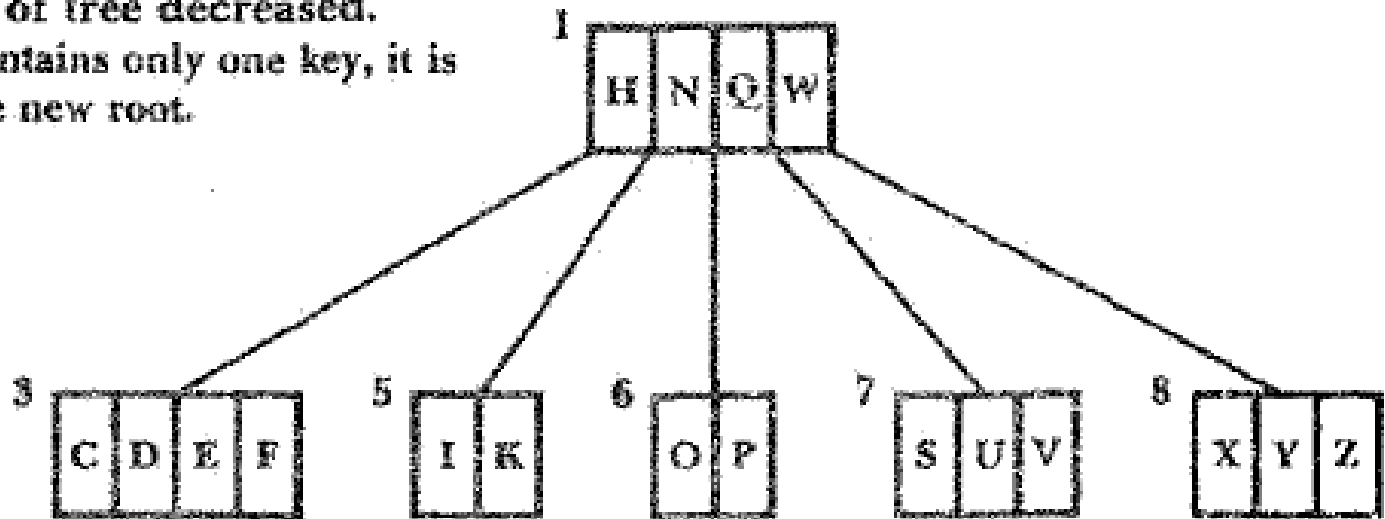
Árvores-B: Remoção

Case 5: Underflow propagates upward.
Now page 1 has underflow. Again, we cannot redistribute, so we concatenate.



Árvores-B: Remoção

Case 6: Height of tree decreased.
Since the root contains only one key, it is absorbed into the new root.



Este caso mostra o que ocorre quando a concatenação é propagada até a raiz. Note que esse nem sempre é o caso: se a página 2 (Q e W) tivesse mais uma chave, seria possível aplicar a redistribuição ao invés da concatenação.



Árvores-B: Remoção

- Eliminação de chave em árvores-B
 - **(1)** Se a chave não estiver em uma folha, troque-a com sua sucessora imediata.
 - **(2)** Elimine a chave da folha.
 - **(3)** Se a folha continuar com o número mínimo de chaves, fim.
 - **(4)** A folha tem uma chave a menos que o mínimo. Verifique as páginas irmãs a esquerda e a direita:
 - **(4.1)** se uma delas tiver mais do que o número mínimo de chaves, aplique redistribuição;
 - **(4.2)** senão concatene a página com uma das irmãs e a chave separadora do pai.
 - **(5)** Se ocorreu concatenação, aplique os passos de 3 a 6 para a página pai.
 - **(6)** Se a última chave da raiz for removida, a altura da árvore diminui.



Redistribuição

- Diferentemente da divisão e da concatenação, o efeito da redistribuição é local (somente entre páginas irmãs). Não existe propagação.
- Outra diferença é que não existe uma regra fixa para o rearranjo das chaves.



Redistribuição

- A redistribuição é útil uma vez que restabelece as propriedades da árvore-B movendo chaves de uma página irmã para a página com problema, ainda que a distribuição de chaves entre as páginas fique muito desigual.
- A estratégia usual é redistribuir as chaves igualmente entre as páginas.



Redistribuição

- **Exemplo:** dada uma árvore-B de ordem 101, os números mínimo e máximo de chaves são, respectivamente, 50 e 100.
- Se ocorre underflow de uma página, e a página irmã tem 100 chaves, qualquer número de chaves entre 1 e 50 pode ser transferido.
- Normalmente transfere-se 25, deixando as páginas equilibradas.



Redistribuição

- A redistribuição é uma idéia nova, que não foi explorada no algoritmo de inserção.
- Entretanto, seria uma opção desejável também na inserção, uma vez que é um modo de evitar, ou pelo menos, adiar, a criação de novas páginas.
- Ao invés de dividir uma página cheia em duas páginas novas semi-vazias, pode-se optar por colocar a chave que sobra (ou mais que uma!) em outra página.
- Essa estratégia resulta em uma melhor utilização do espaço alocado para a árvore.



Redistribuição

- Depois da divisão de uma página, cada página fica 50% vazia.
- Portanto, a utilização do espaço, no pior caso, em uma árvore-B que utiliza splitting, é de cerca de 50%. Em média, para árvores grandes, foi provado que o índice é de 69%.
- Estudos empíricos indicam que a utilização de redistribuição pode elevar o índice para 85%.
 - Splitting ocorre somente quando as páginas irmãs estão cheias
- Esses resultados sugerem que qualquer aplicação séria de árvore-B deve utilizar, de fato, redistribuição durante a inserção.



Variações de Árvores-B

- Árvores-B*
- Árvores-B Virtuais
- Árvores-B⁺

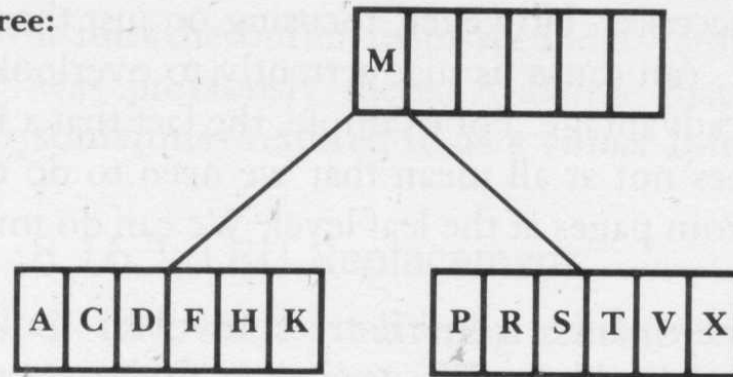
Árvores-B*

(B*-trees)

- Essa nova organização tenta redistribuir as chaves durante a inserção antes de dividir o nó
 - É uma variação de árvore-B na qual cada nó tem, no mínimo, $2/3$ do número máximo de chaves
- A geração destas árvores utiliza uma variação do processo de divisão
 - A divisão é adiada até que duas páginas irmãs estejam cheias
 - realiza-se, então, a divisão do conteúdo das duas páginas em 3 páginas (two-to-three split)

Árvores-B*

Original tree:



Two-to-three-split:
After the insertion of the
key *B*.

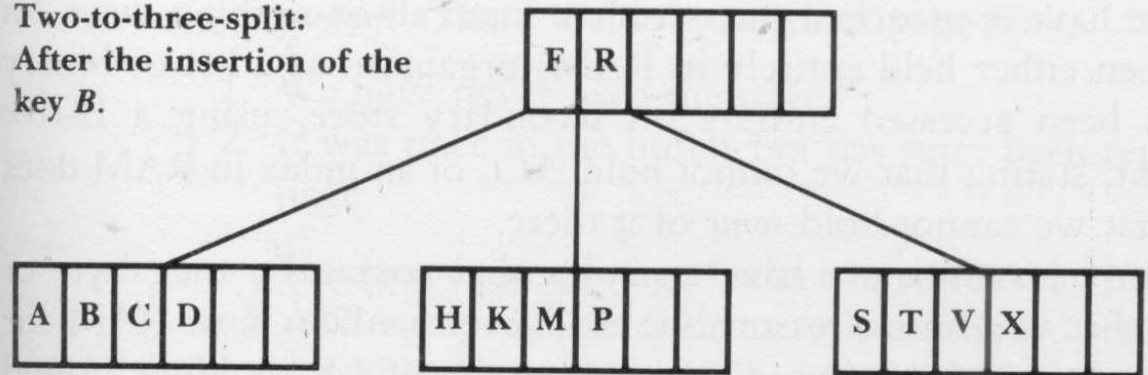


FIGURE 8.30 A two-to-three split.



Árvores-B*

■ Propriedades:

- **(1)** Cada página tem no máximo m descendentes;
- **(2)** *Toda página, exceto a raiz e as folhas, tem no mínimo $\lceil 2m/3 \rceil$ descendentes;*
- **(3)** A raiz tem pelo menos 2 descendentes, a menos que seja uma folha;
- **(4)** Todas as folhas estão no mesmo nível;
- **(5)** Uma página não-folha com k descendentes contém $k-1$ chaves;
- **(6)** *Uma página folha contém no mínimo $\lceil 2m/3 \rceil - 1$ e no máximo $m-1$ chaves.*



Árvores-B*

- As principais alterações estão na segunda e na última regra.
- Estas propriedades afetam as regras para remoção e redistribuição.
- Deve-se tomar cuidado na implementação, uma vez que a raiz nunca tem irmã e, portanto, requer tratamento especial.
 - Uma solução é dividir a raiz usando a divisão convencional (one-to-two split), outra é permitir que a raiz seja maior.
- Esse tipo de variação de árvore-B garante uma utilização mínima de 67% do espaço (versus 50%).



Árvores-B Virtuais (Virtual B-trees)

- Árvores-B são muito eficientes, mas podem ficar ainda melhores.
- Observe, por exemplo, que o fato da árvore ter profundidade 3 não implica em fazer 3 acessos para recuperar as páginas folha.
- O fato de não podermos manter todo o índice na RAM não significa que não se possa manter pelo menos parte dele.



Árvores-B Virtuais

- **Exemplo:**

- Suponha que temos um índice que ocupa 1 MB, e que temos disponíveis 256KB de RAM.
- Supondo que uma página usa 4 KB, e armazena em torno de 64 chaves por página, nossa árvore-B pode estar totalmente contida em 3 níveis.
- Assim, podemos atingir qualquer página com, no máximo, 3 acessos a disco.
- Mas se a raiz for mantida todo o tempo na memória, ainda sobraria muito espaço em RAM:
 - com essa solução simples, o pior caso do número de acessos diminui em um.



Árvores-B Virtuais

- Podemos generalizar esta idéia e ocupar toda a memória disponível com quantas páginas pudermos, sendo que, quando precisarmos da página, ela pode já estar em memória.
- Se não estiver, ela é carregada para a memória, substituindo uma página que estava em memória.
- Quando uma árvore-B utiliza um RAM buffer dessa forma ela é chamada de árvore-B virtual.



Árvores-B Virtuais

- **Política de gerenciamento de substituição: LRU**
 - O processo de acessar o disco para trazer uma página que não está no buffer é denominado page fault e ocorre quando:
 - a página nunca foi utilizada;
 - já esteve no buffer, mas foi substituída por uma nova página (nesse caso, utilizar uma política para gerenciar o problema).
 - Se a página não estiver em RAM, e esta estiver cheia, precisamos escolher uma página para ser substituída.
 - Uma opção: **LRU (Last Recently Used)**
 - Substitui-se a página que foi acessada menos recentemente.



Árvores-B Virtuais

- **Substituição baseada na altura da árvore**
 - Podemos optar por colocar todos os níveis mais altos da árvore em RAM. No exemplo de 256KB de RAM e páginas de 4KB, podemos manter 64 páginas em memória.
 - Isso comporta a raiz e mais, digamos, as 8 ou 10 que compõem o segundo nível. Ainda sobra espaço para as páginas folhas (utiliza-se LRU), e o número de acessos diminui em mais uma unidade.
- **Importante!!!**
 - Bufferização deve ser incluída em qualquer situação real de utilização de árvore-B.



Alocação de Informação Associada à Chave

- E a informação associada às chaves (os demais campos dos registros), onde fica?
- Se a informação for mantida junto com a chave, ganha-se um acesso a disco, mas perde-se no número de chaves que pode ser colocado em uma página. Isso reduz a ordem da árvore, e aumenta a sua altura.
- Se ficar em um arquivo separado, a árvore é realmente usada como índice, e cada chave tem o RRN, ou byte offset, que dá a posição do registro associado no arquivo de dados.



Alocação de Informação Associada à Chave

- **Exemplo:**

- Suponha que temos que indexar 1.000 chaves e informações associadas aos registros:
 - Tamanho requerido para armazenar a chave e suas informações: 128 bytes.
 - Tamanho requerido para armazenar a chave e um ponteiro para suas informações: 16 bytes.
- Suponha que se tenha 512 bytes disponíveis para as chaves e informações associadas.



Alocação de Informação Associada à Chave

- **Exemplo:**

- Temos duas alternativas:

- Informação armazenada com a chave: quatro chaves por página – ordem 5

$$d \leq 1 + \log_3(500.5) = 6.66$$

- Ponteiro armazenado com a chave: 32 chaves por página – ordem 33

$$d \leq 1 + \log_{17}(500.5) = 3.19$$

É necessário avaliar cada situação!!!