

Árvores B (part 1)

Prof. Dr. Lucas C. Ribas

Disciplina: Estrutura de Dados II

Departamento de Ciências de Computação e Estatística



UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"



IBILCE / UNESP - CÂMPUS DE SÃO JOSÉ DO RIO PRETO



● Cenário até então

- Acesso a disco é **caro** (lento)
- **Pesquisa binária** é útil em índices ordenados...
- mas com índice grande que não cabe em memória principal, pesquisa binária exige **muitos acessos a disco**
- Exemplo: 15 itens podem requerer 4 acessos, enquanto 1.000 itens podem requerer até 11 acessos
- Se as chaves estão em memória secundária, qualquer procedimento que exija **mais do que 5 ou 6 acessos** para localizar uma chave é altamente indesejável



● Cenário até então

- Manter em disco um índice ordenado para busca binária tem custo proibitivo
 - Inserir ou eliminar, mantendo o arquivo ordenado custa muito caro.
- Necessidade de método com inserção e eliminação com apenas efeitos locais, isto é, que não exija a reorganização total do índice



- Podemos utilizar uma árvore como um mecanismo para pesquisar registros armazenados em um arquivo em disco. A estrutura de dados em modo "árvore" seria utilizada como um índice para acessar o arquivo.
- Armazenamos na árvore os valores de um dos campos dos registros do arquivo de dados (campo **chave** ou **campo de pesquisa**).
- Cada valor chave na árvore é associado a um ponteiro para o bloco em disco que contém aquele registro no arquivo de dados.
- Para manter os dados da árvore (**índice**) em disco, podem ser utilizados arquivos seqüenciais, onde cada nó da árvore seria um registro.

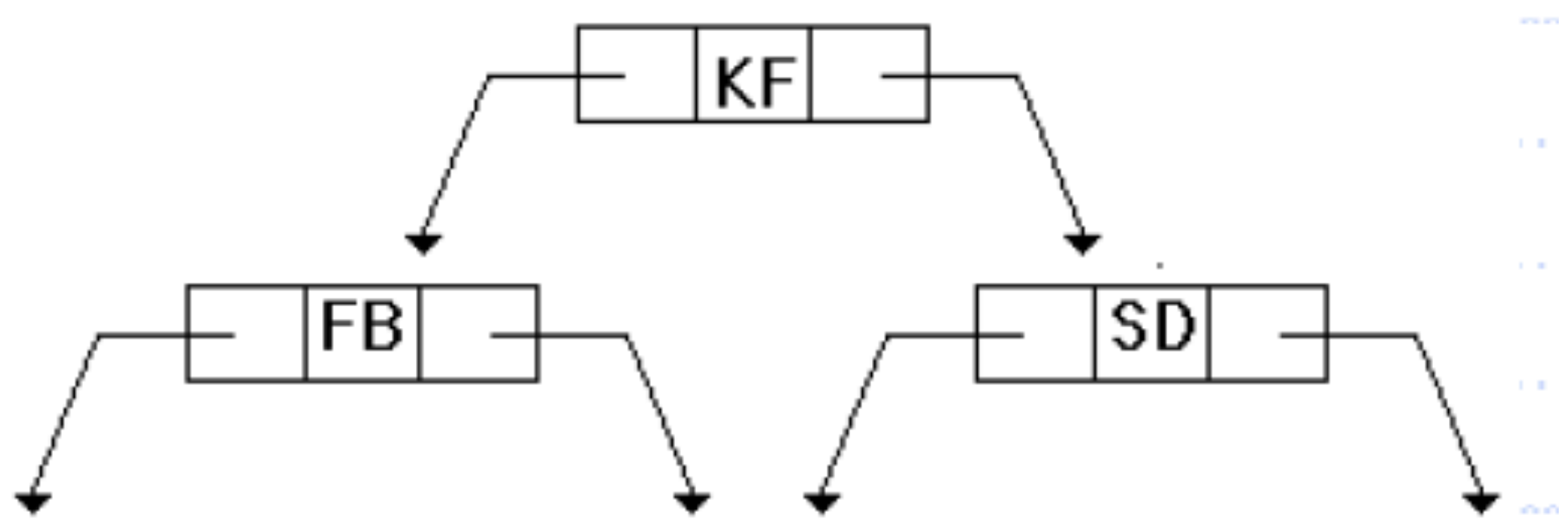
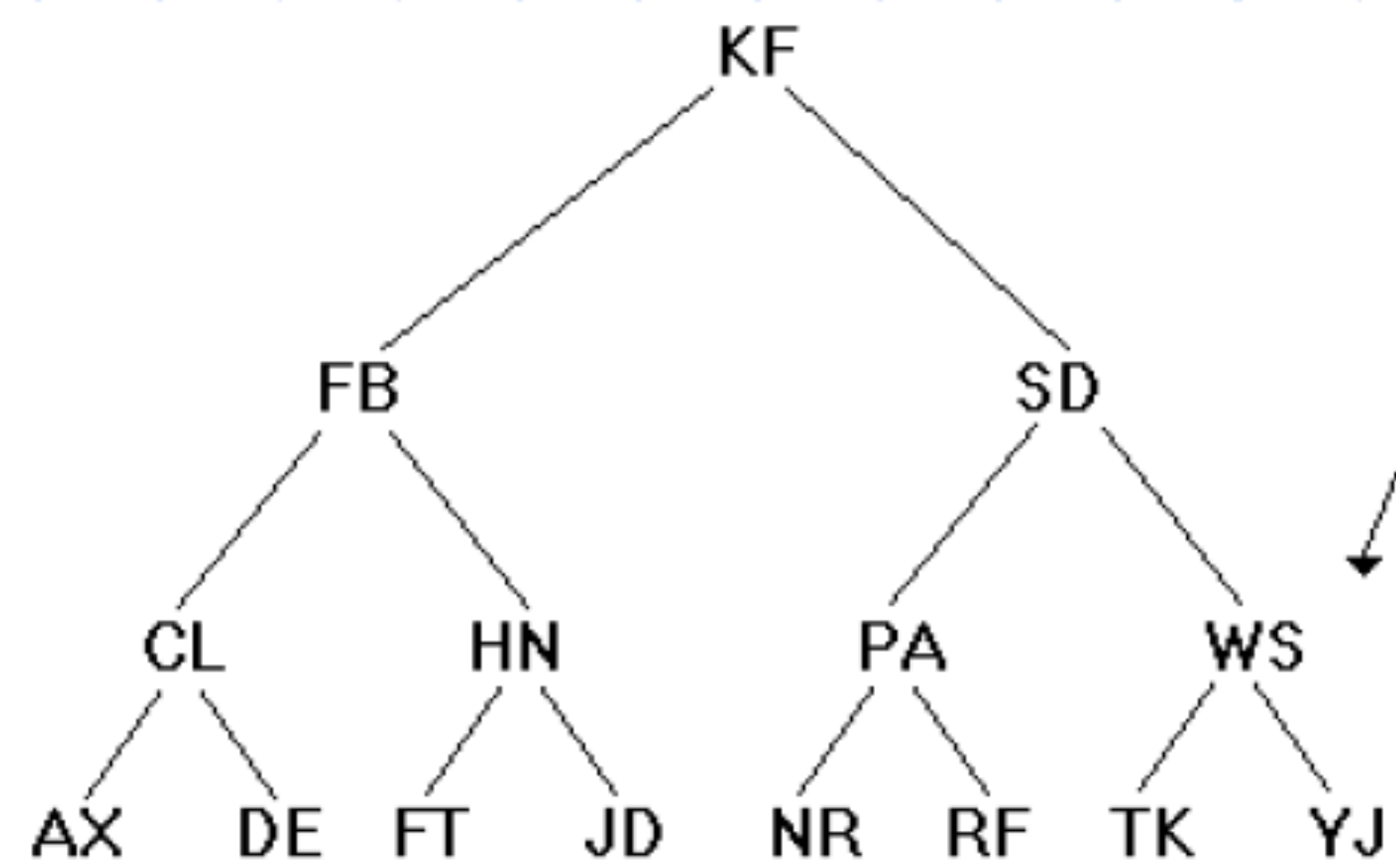
Árvores Binárias de Busca



- Os registros (de índice) são mantidos num arquivo, e ponteiros (esq e dir) indicam onde estão os registros filhos.
- Este arquivo de índice pode ser mantido em memória secundária: os ponteiros para os filhos dariam o RRN das entradas correspondentes aos filhos.
- Formato dos nós a serem empregados na organização dos dados seria:
 - um elemento (que é o valor do campo chave);
 - o endereço do registro, referente a esta chave, no arquivo de dados no disco
 - e a referencia das duas sub-árvores.

endereço filho da esquerda	valor do campo chave	endereço do registro de dados desta chave	endereço do filho da direita
-----------------------------------	-----------------------------	--	-------------------------------------

Solução: Árvore de Busca Binária (ABB)?



Raiz = 9

	key	filho esq.	filho dir.
0	FB	10	8
1	JD		
2	RF		
3	SD	6	15
4	AX		
5	YJ		
6	PA	11	2
7	FT		

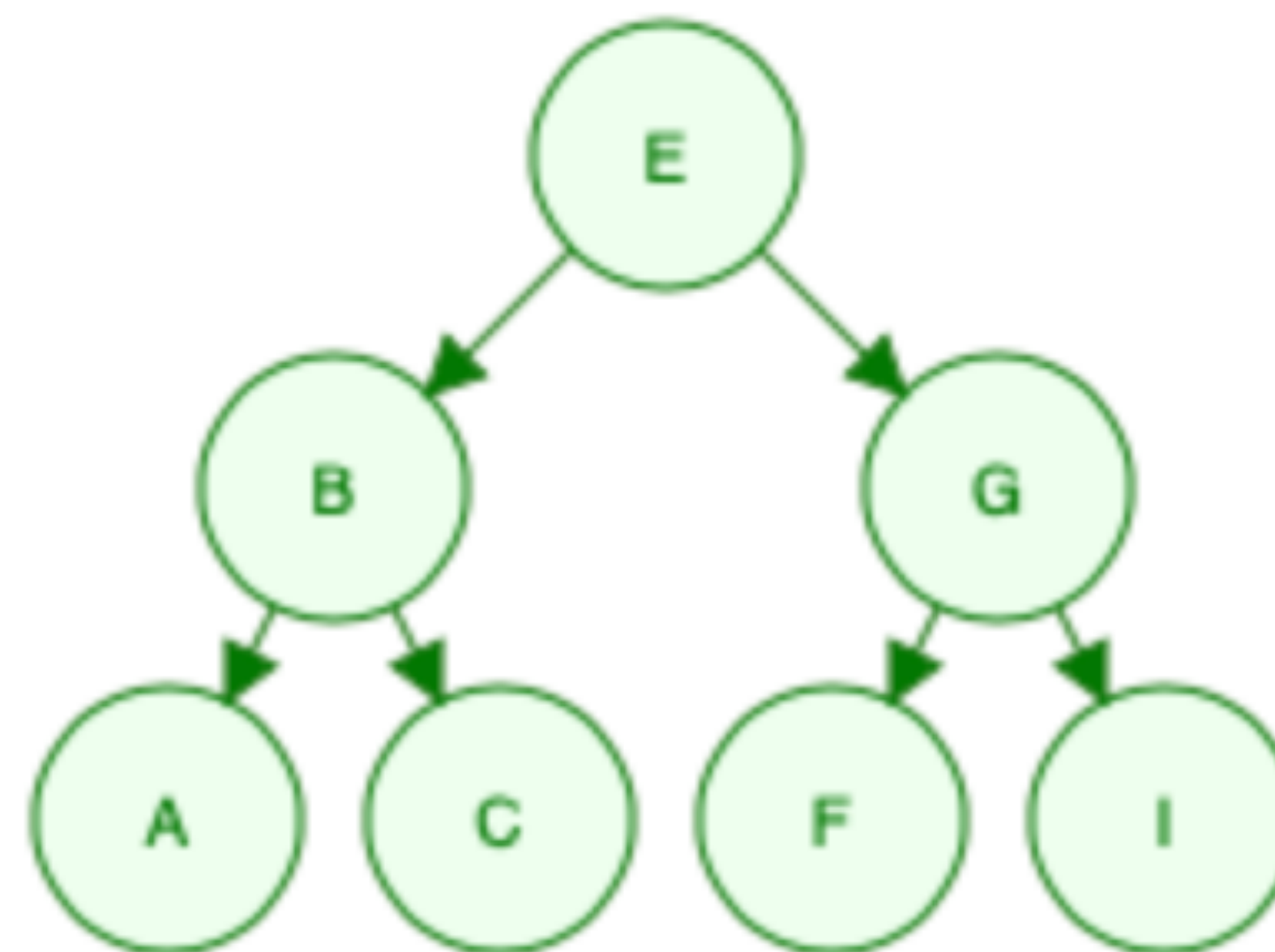
	key	filho esq.	filho dir.
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	NR		
12	DE		
13	WS	14	5
14	TK		

- Registros (tam. fixo) são mantidos em arquivo, e ponteiros (esq e dir) indicam onde (RRN) estão os registros filhos.
- Nessa estrutura foi omitido o RRN associado à chave, para facilitar o desenho
- O ponteiro para o nó raiz pode ser mantido no cabeçalho do arquivo.
- A ordem lógica dos registros não está associada à ordem física no arquivo.
- O arquivo físico do índice não precisa ser mantido ordenado -> Para recuperar utiliza-se dos campos esq e dir.
- Para acrescentarmos uma nova chave ao arquivo -> mantê-la uma ABB

● Construa a árvore binária de busca em arquivo usando registros de tamanho fixo.

• E B G A C F I

RRN	Key	Esq	Dir
0	E	1	2
1	B	3	4
2	G	5	6
3	A		
4	C		
5	F		
6	I		





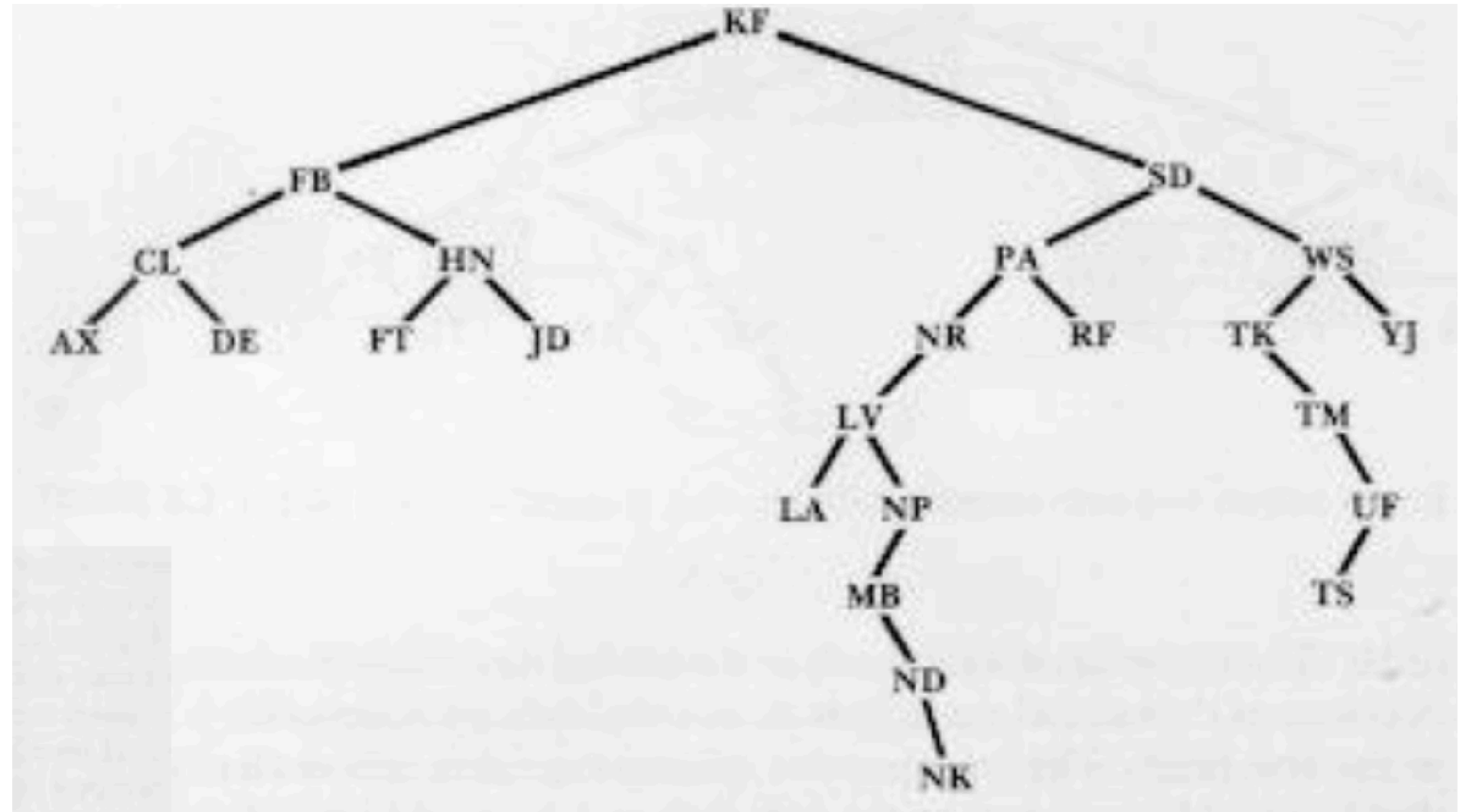
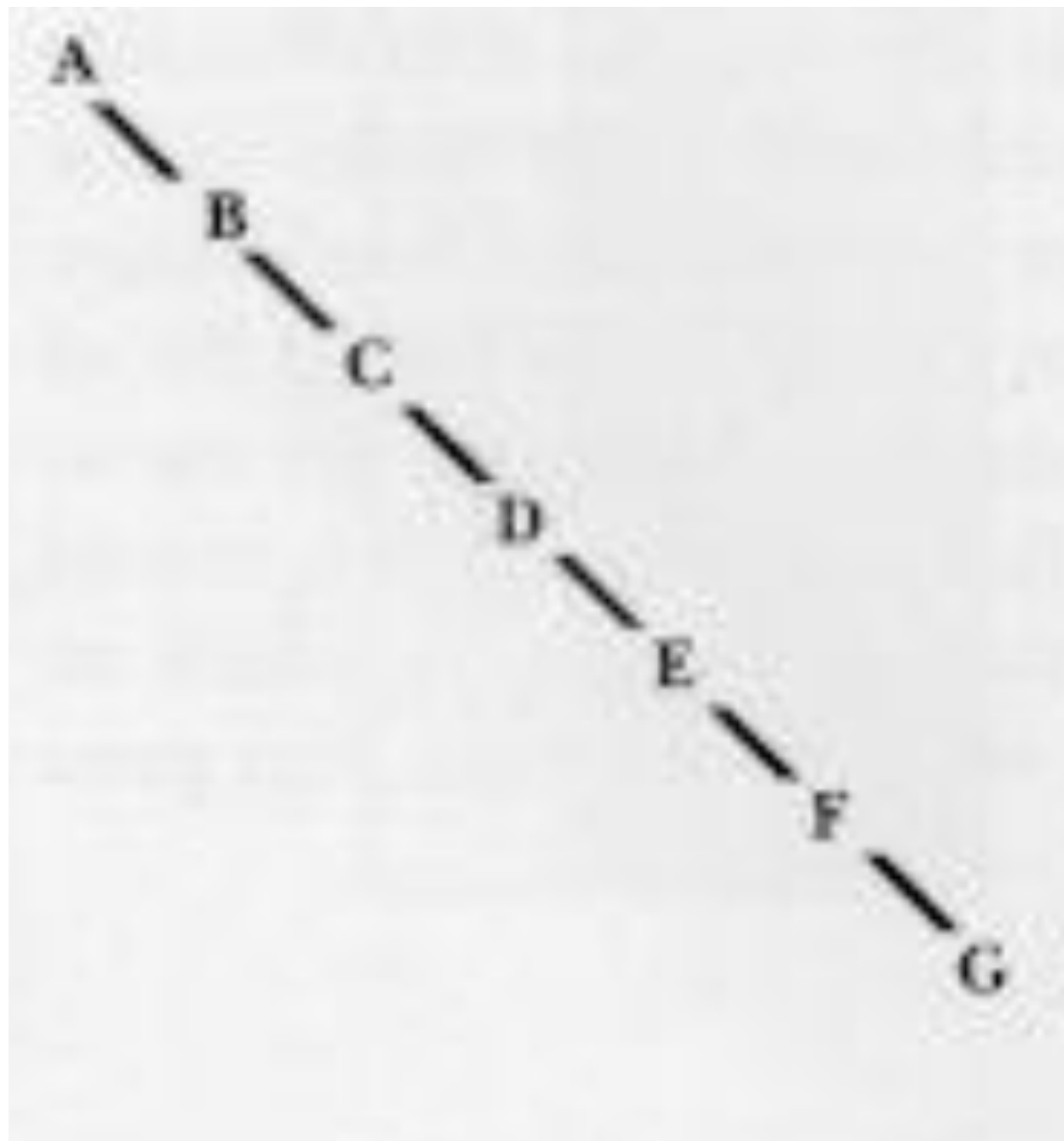
- ◎ **Ordem lógica** dos registros != **ordem física** no arquivo
 - Ordem lógica: dada por ponteiros **esq** e **dir**
 - Registros não precisam estar fisicamente ordenados
- ◎ Inserção de uma nova chave no arquivo
 - É necessário **saber onde inserir**
 - Busca pelo registro é necessária, mas **reorganização do arquivo não**

Problema: Desbalanceamento



● Inserção das chaves NP MB TM LA UF ND TS NK

Situação indesejável: inserção em ordem alfabética



AVL



- A eficiência do uso de árvores binárias de busca exige que estas sejam mantidas balanceadas:
 - Isso implica no uso de árvores AVL,
 - e algoritmos associados para inserção e eliminação de registros.
- **Árvore binária perfeitamente balanceada:** altura da árvore, ou seja, $\log_2(N+1)$
- Numa **árvore AVL**, o número máximo de comparações para localizar uma chave em uma árvore com N chaves é
 - igual à $1.44 \cdot \log_2(N+2)$ (pior caso)
 - Portanto, dadas 1.000.000 de chaves a busca poderia percorrer até 28 níveis ($1.44 \log_2(1.000.000+2)$)



● Problema

- Se chaves em memória secundária, ainda há muitos acessos!
 - **28 seeks são inaceitáveis!**

● Árvores balanceadas

- 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, não resolvem o problema no número excessivo de acessos a disco.

● Até agora...

- Árvores binárias de busca **dispensam ordenação dos registros** 😊
- Mas **número excessivo de acessos** ☹️

Árvores Binárias Paginadas



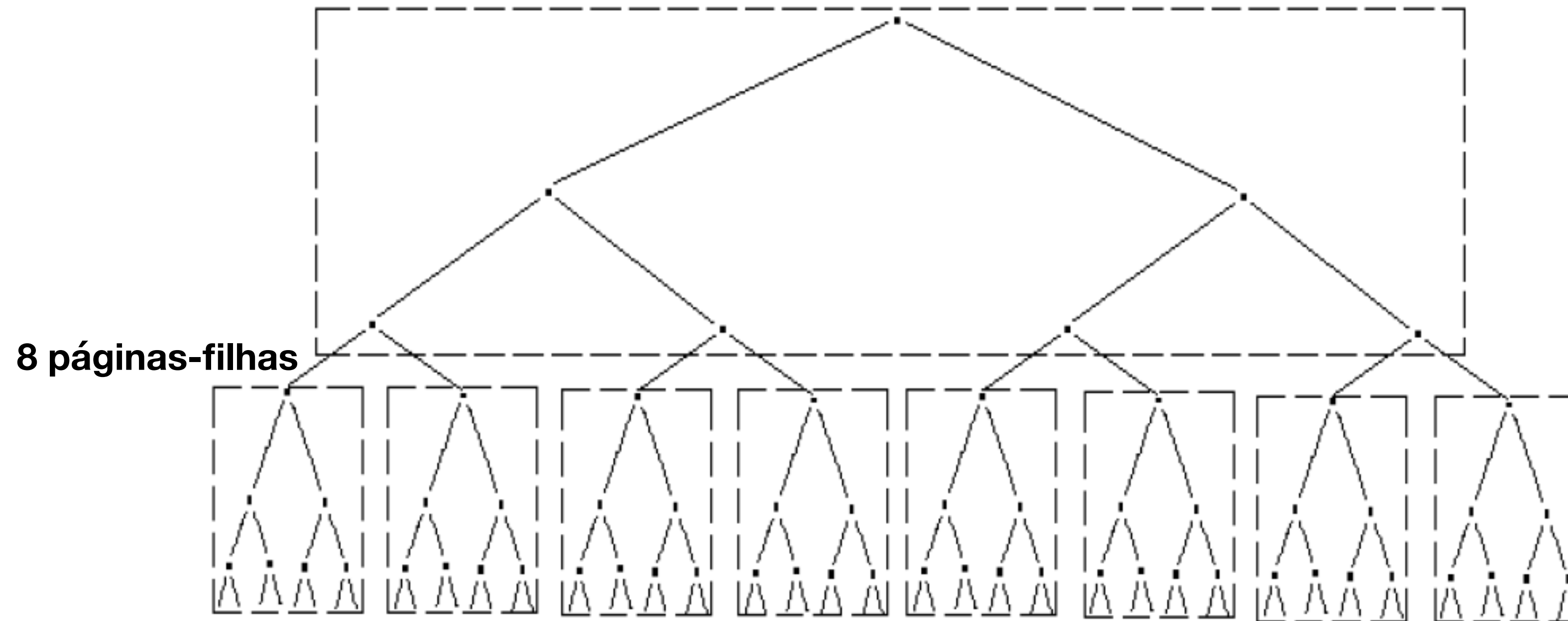
© *Paged Binary Trees*

- © A **busca** (*seek*) por uma posição específica do disco é **muito lenta**, mas uma vez encontrada a posição, pode-se ler uma grande quantidade registros seqüencialmente a um custo relativamente pequeno.
- © Esta combinação de busca (*seek*) lenta e transferência rápida sugere a noção de **página**.
 - Cada página é armazenada em um bloco -> uma vez realizado um seek, que consome um tempo considerável, toda os registros em uma mesma "página" do arquivo são lidos.
 - A página pode conter um número bastante grande de registros, e se o próximo registro a ser recuperado estiver na mesma página, será economizado um acesso ao disco.

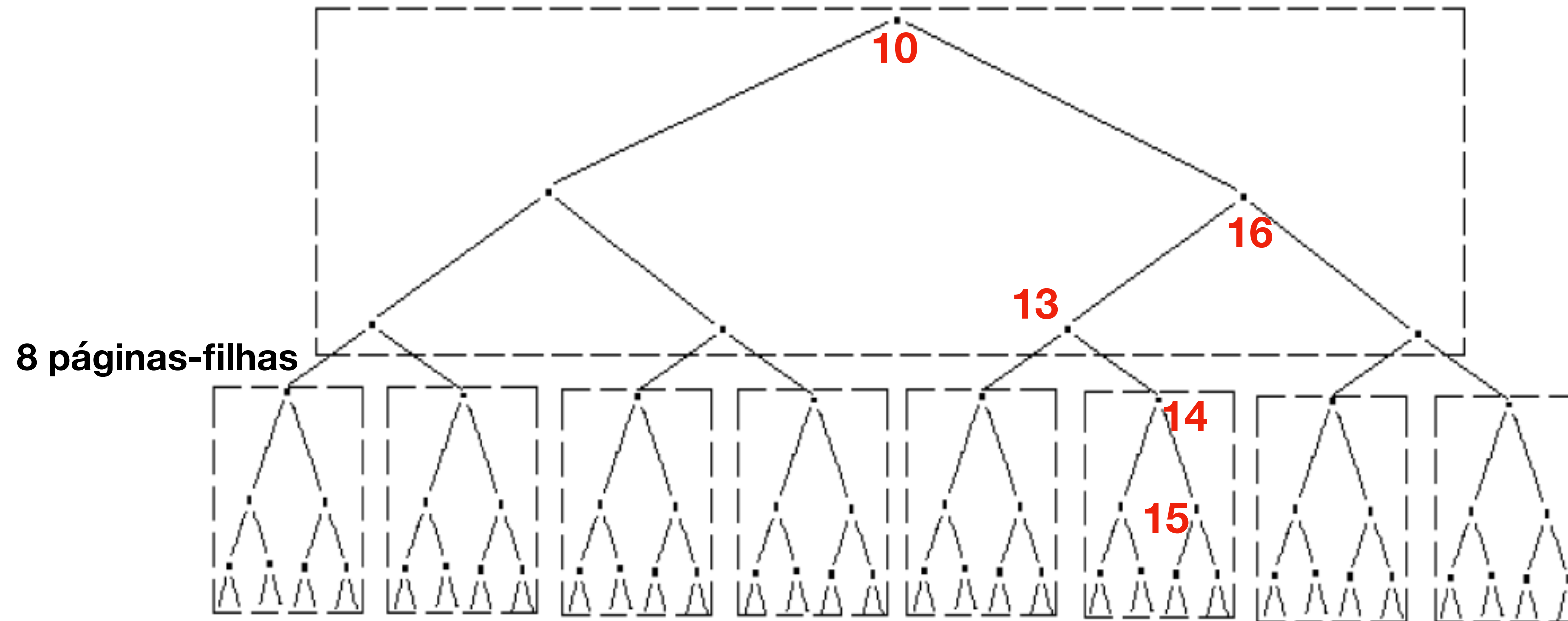


● Noção de **página** em sistemas paginados

- Conjunto de setores logicamente contíguos no disco
- Um arquivo é visto pelo S.O. como um grupo de páginas de disco
 - arquivos são alocados em um ou mais páginas de disco
- **1 seeking** para localizar **1 página de disco**
- Feito o seek, todos os registros de uma mesma "**página**" do arquivo são lidos
- Esta página pode conter um número grande de registros
 - Se o próximo registro a ser recuperado estiver na mesma página já lida, evita-se um novo acesso ao disco



- Na figura acima cada página contém **7 nós** e pode ter ramos para **8 novas páginas**.
- 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, teremos 64 novas páginas, e poderemos encontrar qualquer um das 511 chaves armazenadas com **apenas 3 seeks**



- Na figura acima cada página contém **7 nós** e pode ter ramos para **8 novas páginas**.
- 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, teremos 64 novas páginas, e poderemos encontrar qualquer um das 511 chaves armazenadas com **apenas 3 seeks**



- unesp 



◎ Pior caso para o número de seeks:

- Árvore binária completa, perfeitamente balanceada: **$\log_2 (N+1)$**
- Versão em páginas: **$\log_{k+1}(N+1)$** onde k é o no. de chaves por página

◎ Se N é o número total de chaves e $k = 511$

- **árvore binária:** $\log_2(134.217.727) = 27$ acessos
- **versão em páginas:** $\log_{511+1}(134.217.727) = 3$ acessos

◎ Preço a pagar:

- maior tempo na transmissão de grandes quantidades de dados, e
- a necessidade de manutenção da organização da árvore. Como seria a construção 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.
- Sabemos que **temos de começar com a chave do meio** para garantir que o conjunto de chaves seja dividido de forma balanceada.
- Entretanto, a situação se complica se estamos recebendo as chaves em uma seqüência aleatória e construindo a árvore a medida em que as chaves chegam.



Problema da construção *Top-Down* de árvores paginadas

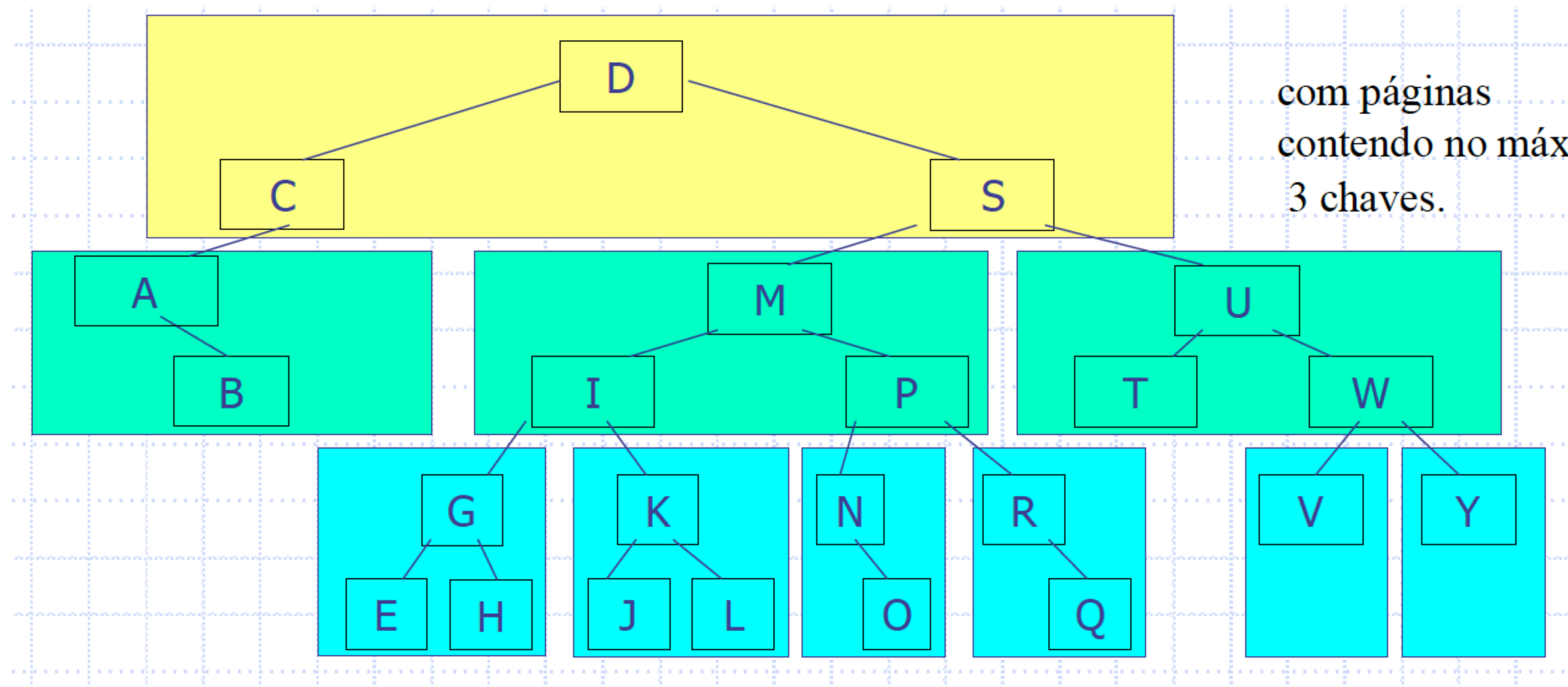
Considere o problema de construir uma árvore binária paginada, com páginas contendo no máximo 3 chaves.

- 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**

- Considere que a construção é feita com a inserção das chaves nas folhas, rodando a árvore sempre que possível para manter o balanceamento.
- A construção a partir do topo implica que as chaves iniciais estarão necessariamente na raiz.

Conjunto de dados: **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**



- A construção top-down faz com que as primeiras chaves fiquem na página-raiz.
- Nesse exemplo C e D não deveriam estar no topo
- A árvore construída dessa forma não está balanceada.



Problema da construção *Top-Down* de árvores paginadas

- E se rotacionássemos as páginas (como fazemos como os nós)?
 - Tente formular um algoritmo para isso
 - Muito complexo!



Problema da construção *Top-Down* de árvores paginadas

● Questões:

- Como garantir que as chaves da raiz são boas separadoras?
- Como impedir o agrupamento de chaves que não deveriam estar na mesma página (C, D e S)?
- Como garantir que cada página contenha um número mínimo de chaves?

● Solução: utilização de árvores-B

- árvore de busca balanceada, onde ao se inserir uma chave ela é colocada sempre numa folha
- por meio de sub-divisão (split) e promoção (promote), a árvore fica sempre balanceada

Árvore-B



- **Árvores-B** são uma **generalização** da idéia de ABB paginada
 - Não são binárias
 - Conteúdo de uma página não é mantido como uma árvore
 - A construção é **bottom-up**
- Um pouco de história
 - 1972: Bayer and McGreight publicam o artigo [Organization and Maintenance of Large Ordered Indexes](#)
 - **1979**: **árvores-B** viram praticamente **padrão** em sistemas de arquivos de propósito geral



● Características

- Completamente balanceadas
- criação bottom-up (em disco)
 - nós folhas -> nó raiz

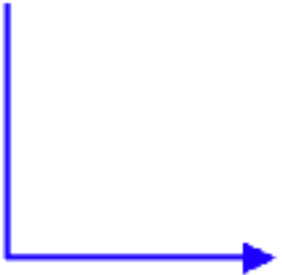
● Inovação

- Não é necessário construir a árvore a partir do nó raiz, como é feito para árvores em memória principal e para as árvores anteriores



● Consequências

- Chaves “erradas” não são mais alocadas no nó raiz
 - Elimina as questões em aberto de *chaves separadoras* e de *chaves extremas*
- Não é necessário tratar o problema de desbalanceamento

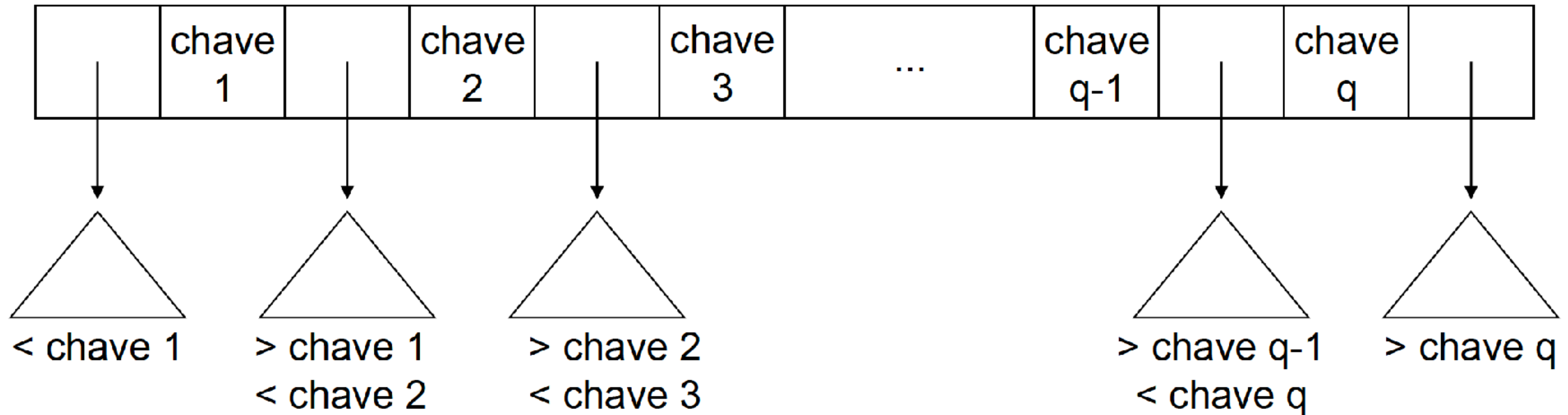


na árvore-B, as chaves na raiz da árvore
emergem naturalmente

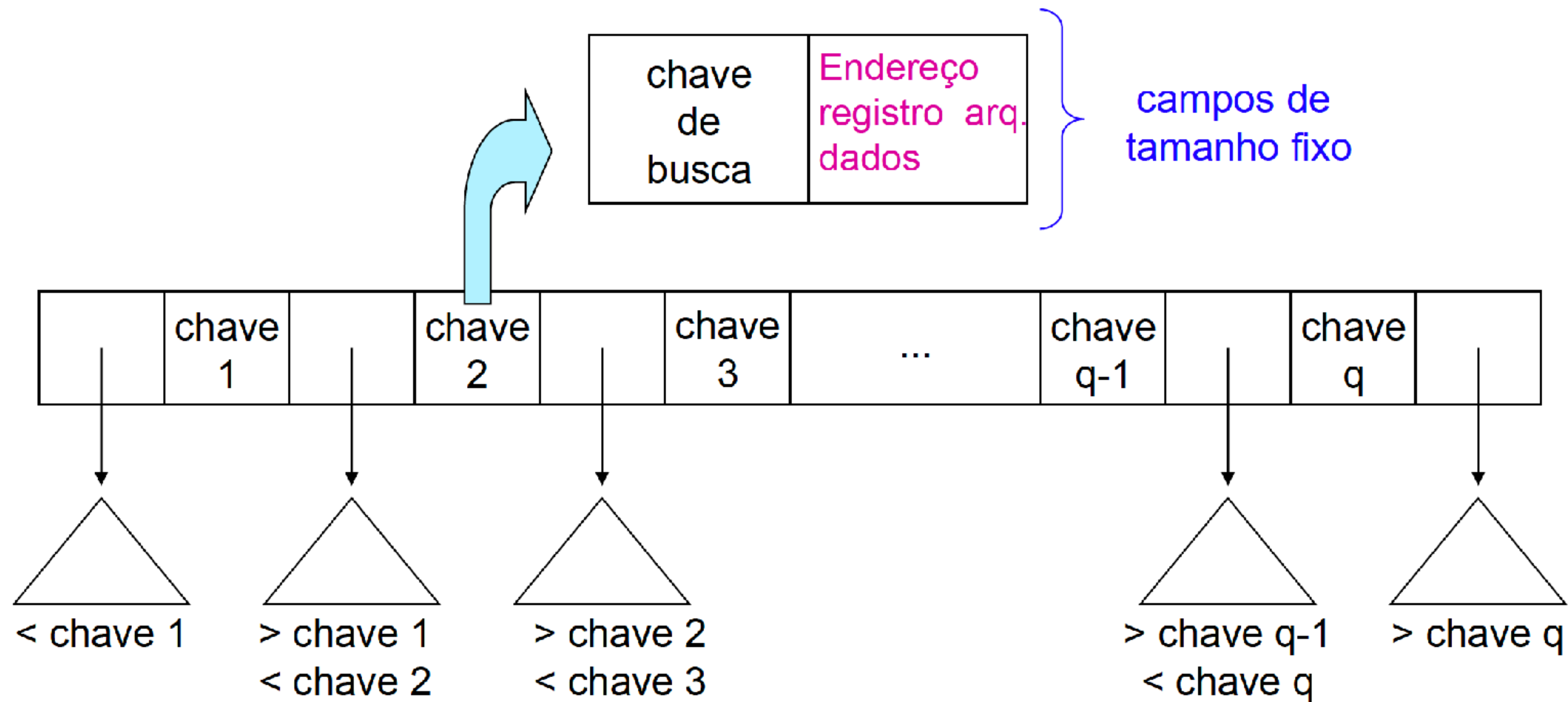


- Cada página (ou nó) é formada por uma seqüência ordenada de chaves e um conjunto de ponteiros.
- Não existe uma árvore explícita dentro de uma página (diferente da árvore página mostrada anteriormente).
- O número de ponteiros em um **nó = número de chaves + 1**.
- **Ordem**
 - O **número máximo de ponteiros** que podem ser armazenados em um nó é a **ordem da árvore**. EX: uma árvore-B de **ordem 8** possui nós com, no máximo, **7 chaves e 8 filhos**
- **Observações**
 - O número máximo de ponteiros é igual ao número máximo de descendentes de um nó.
 - Os nós folha não possuem filhos, e seus ponteiros são nulos.
 - Registros de tamanho fixo para armazenar um nó

- Estrutura lógica de um nó
 - Registro de tamanho fixo -> RRN

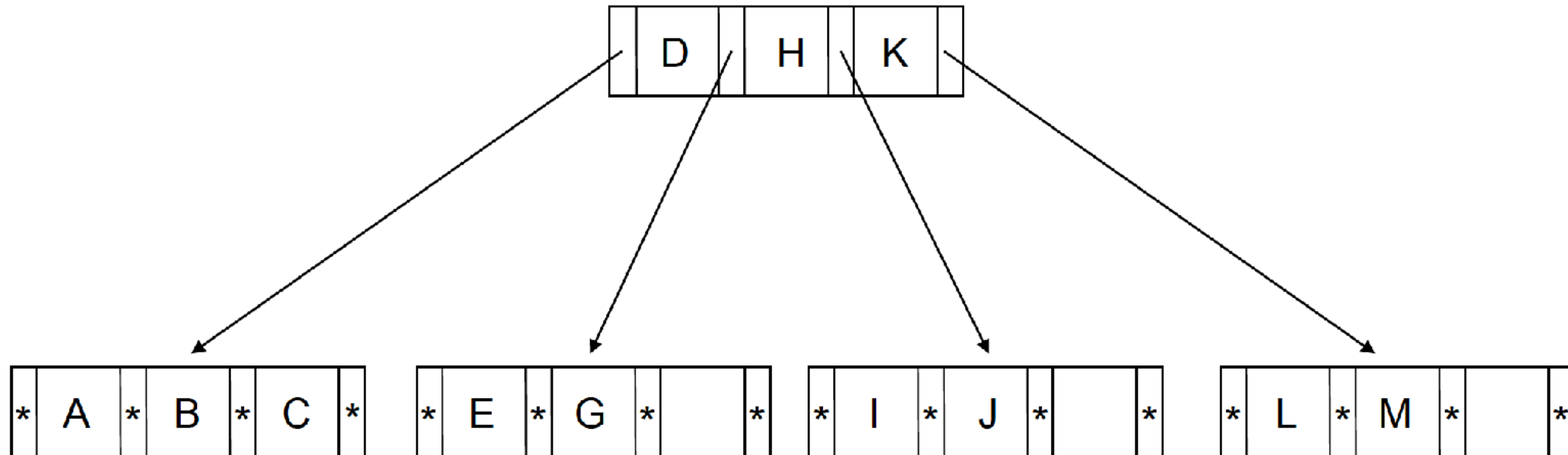


● Estrutura lógica de um nó





- Ordem 4: até 3 chaves por nó; até 4 filhos





● Característica

- Sempre realizada nos **nós folha** (a busca binária por uma chave inexistente termina sempre no nó folha)

● Situações a serem analisadas

1. árvore vazia

2. overflow no nó raiz

3. inserção em nós folha

Inserção em Árvore-B Vazia



- Criação e preenchimento do nó
 - primeira chave: criação do nó raiz
 - demais chaves: inserção até a capacidade limite do nó
- Exemplo
 - nó com capacidade para 7 chaves → ordem 8
 - chaves: letras do alfabeto
 - situação inicial: árvore vazia



- Chaves B C G E F D A
 - inseridas desordenadamente
 - mantidas ordenadas no nó
- Ponteiros (*)
 - nós folhas: -1 ou fim de lista (NIL)
 - nós internos: RRN do nó filho ou -1
- Nó raiz (= nó folha)

Raiz=0

RRN 0

*	<u>A</u>	*	<u>B</u>	*	C	*	D	*	E	*	F	*	G	*
-		-				-								

Overflow no nó raiz

● Passo 1 – particionamento do nó (split)

- nó original -> nó original + novo nó
 - split 1-to-2
- as chaves são distribuídas uniformemente nos dois nós
 - chaves do nó original + nova chave

RRN 0

*	<u>A</u>	*	<u>B</u>	*	C	*	D	*	E	*	F	*	G	*
---	----------	---	----------	---	---	---	---	---	---	---	---	---	---	---

● Exemplo: inserção de J

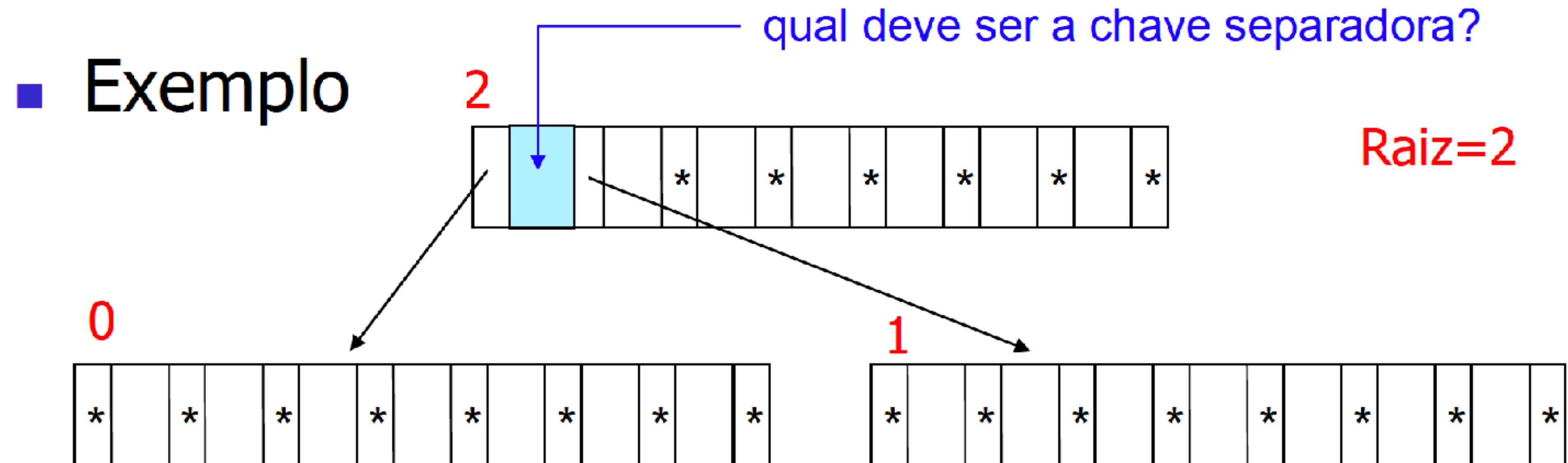
0

*	A	*	B	*	C	*	D	*		*		*		*
---	---	---	---	---	---	---	---	---	--	---	--	---	--	---

1

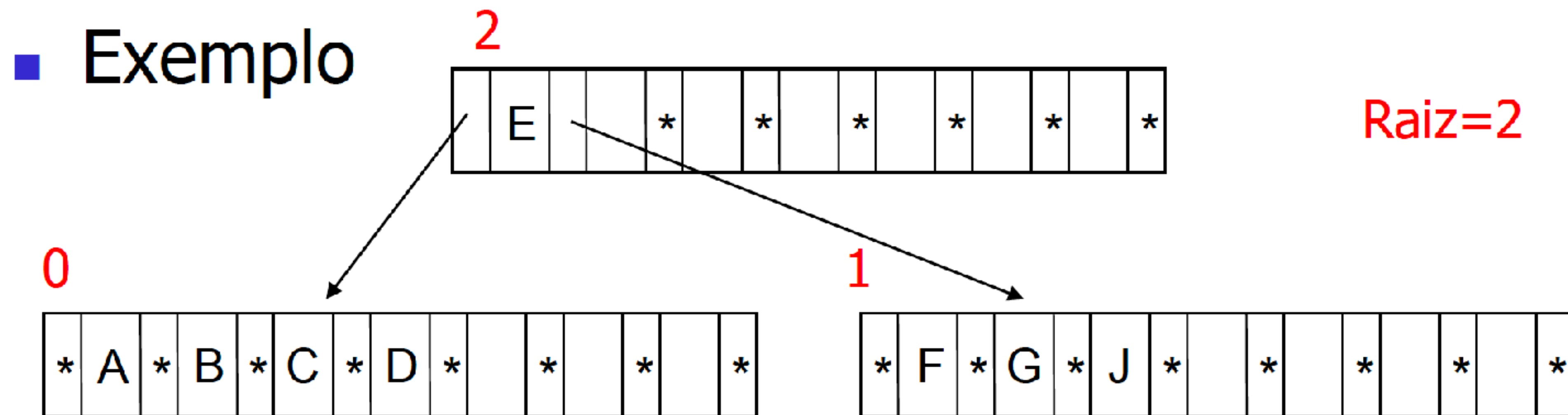
*	E	*	F	*	G	*	J	*		*		*		*
---	---	---	---	---	---	---	---	---	--	---	--	---	--	---

- **Passo 2** – criação de uma **nova raiz** (efeito *bottom-up*); aumenta altura
 - a existência de um nível mais alto na árvore permite a escolha das folhas durante a pesquisa



● **Passo 3** – promoção de chave (*promotion*)

- a primeira chave do novo nó resultante do particionamento é **promovida** para o nó raiz → é a mediana do conjunto dos dois nós



Inserção: overflow nó raiz



Raiz=0

RRN 0

*	A	*	B	*	C	*	D	*	E	*	F	*	G	*
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Inserir J

2

	E		*		*		*		*		*		*	
--	---	--	---	--	---	--	---	--	---	--	---	--	---	--

Raiz=2

0

*	A	*	B	*	C	*	D	*		*	*		*
---	---	---	---	---	---	---	---	---	--	---	---	--	---

1

*	F	*	G	*	J	*		*	*		*	*		*
---	---	---	---	---	---	---	--	---	---	--	---	---	--	---

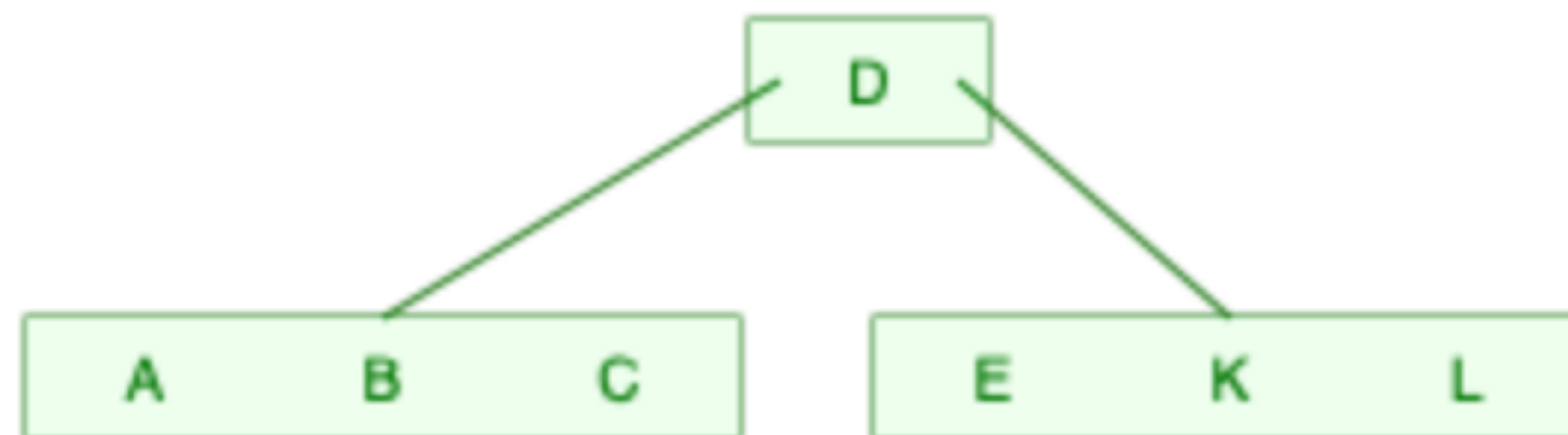


- Crie uma árvore B de ordem 5
- D C E A L K B



● Crie uma árvore B de ordem 5

● D C E A L K B



Inserção em nós folhas



- **Passo 1** – pesquisa binária (inserção sempre nas folhas)
 - a árvore é percorrida até encontrar o nó folha no qual a nova chave será inserida
 - **nó folha em memória principal**
- **Passo 2(a)** – inserção em nó com lugar disponível
 - inserção ordenada da chave no nó (sequencial)
 - alteração dos valores dos campos de referência



- **Passo 2(b)** – inserção em nó cheio (**overflow**)
 - Particionamento (**split**)
 - criação de um novo nó (nó original -> nó original + novo nó)
 - distribuição uniforme das chaves nos dois nós
 - Promoção (**promotion**)
 - escolha da primeira chave do novo nó como chave separadora no nó pai
 - ajuste do nó pai para apontar para o novo nó
 - **propagação recursiva de overflow**



● Insira as seguintes chaves em um índice árvore-B

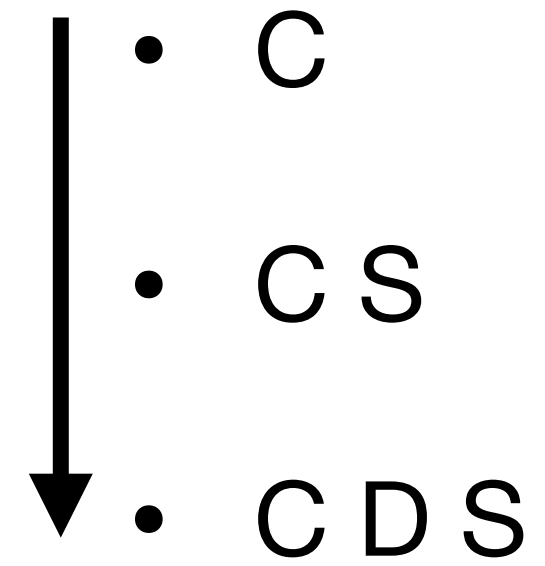
• **C S D T A M P I B W N G U R K E H O L J Y Q Z F X V**

● Ordem da árvore-B: 4

- em cada nó (página de disco)
 - número de chaves: 3
 - número de ponteiros: 4

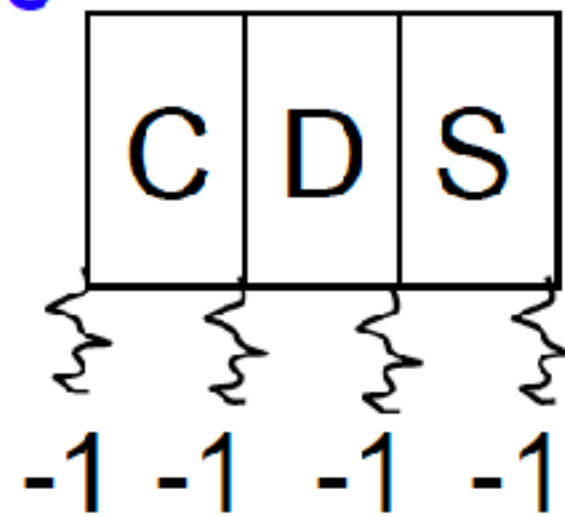
● Passo 1 - inserção de C, S, D

- Criação do nó raiz



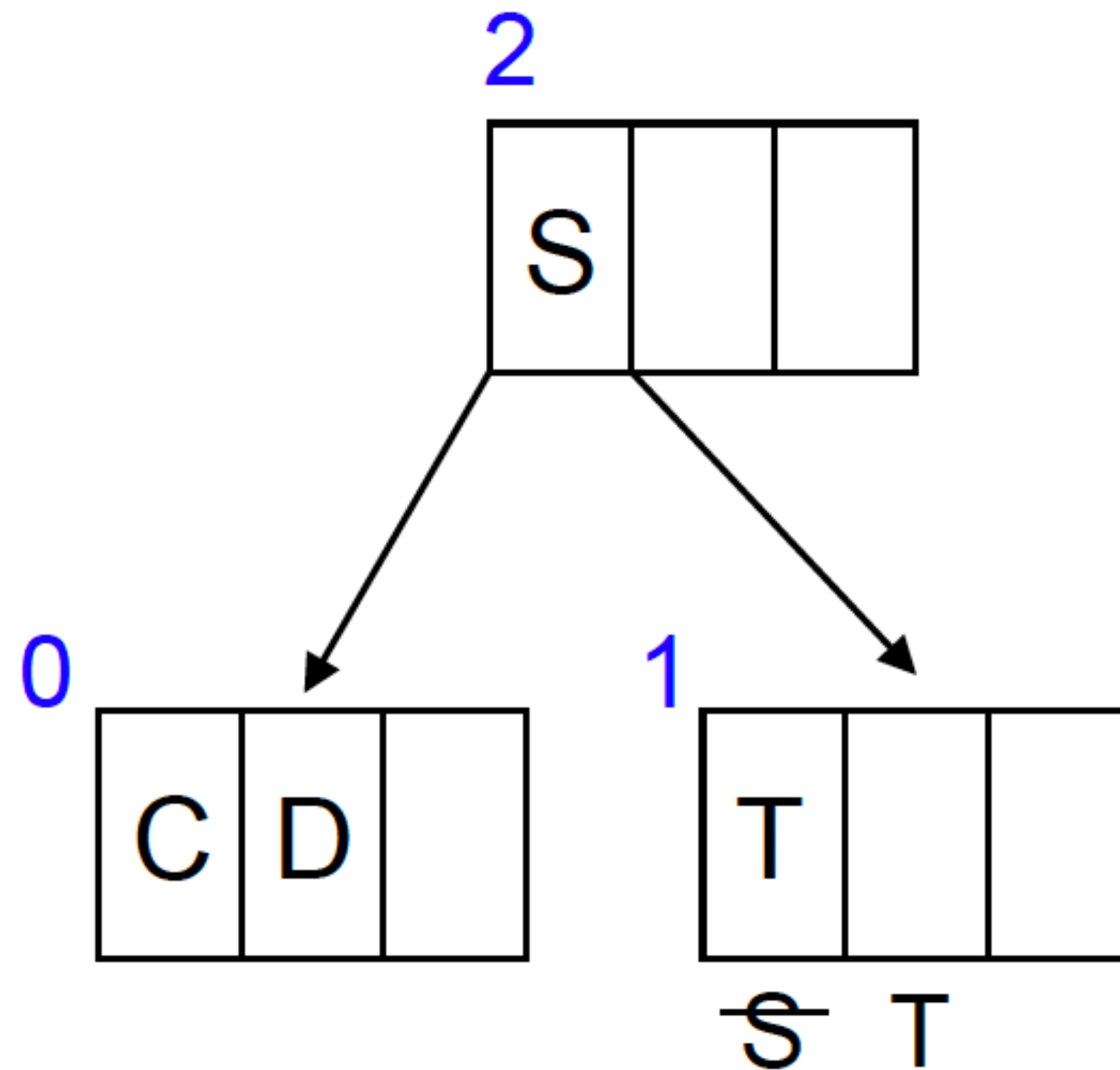
RRN da
página/
registro

→ 0



● Passo 2 - inserção de T

- nó raiz cheio

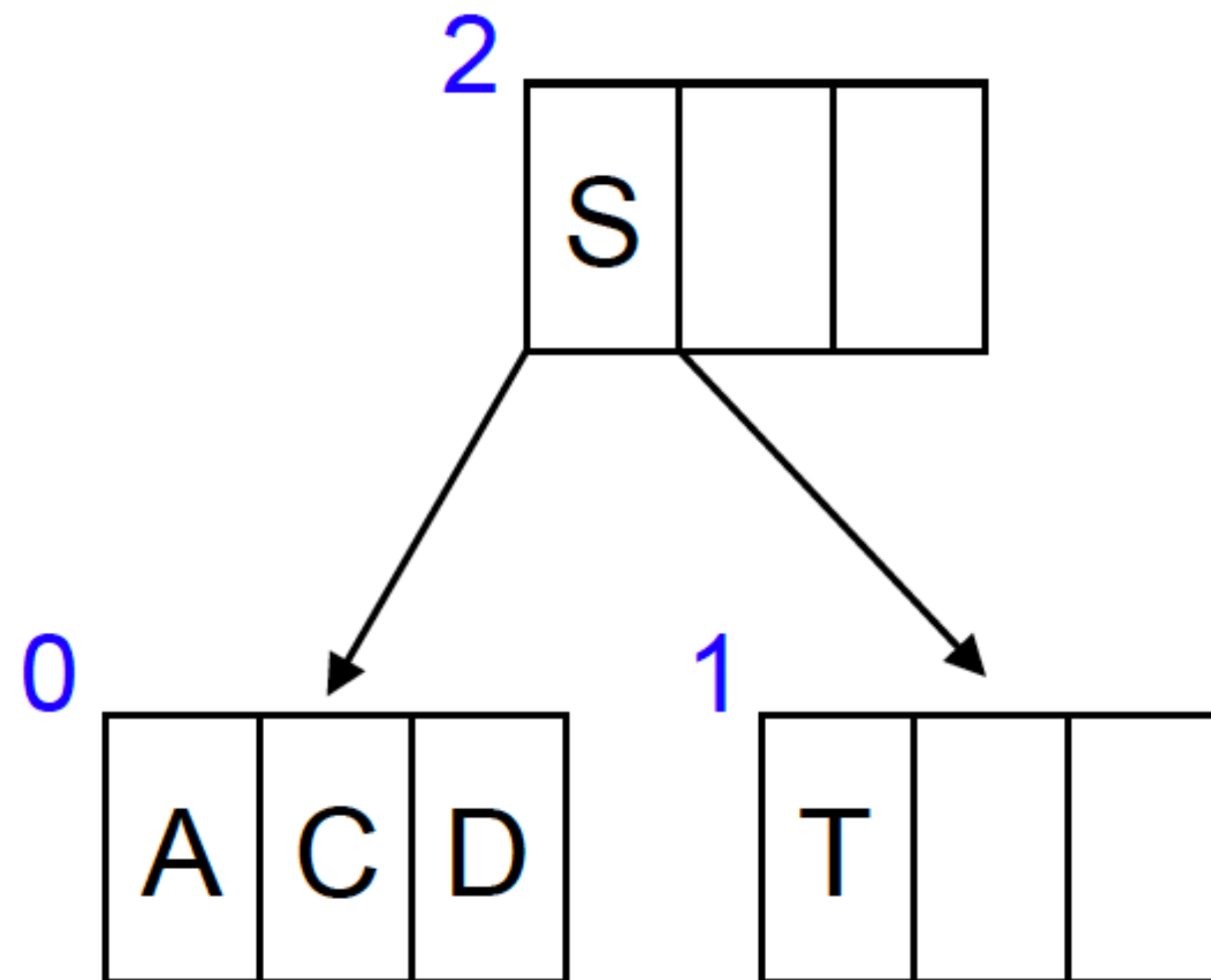


- particionamento do nó
- criação de uma nova raiz
- promoção de S



● Passo 3 - inserção de A

- nó folha com espaço

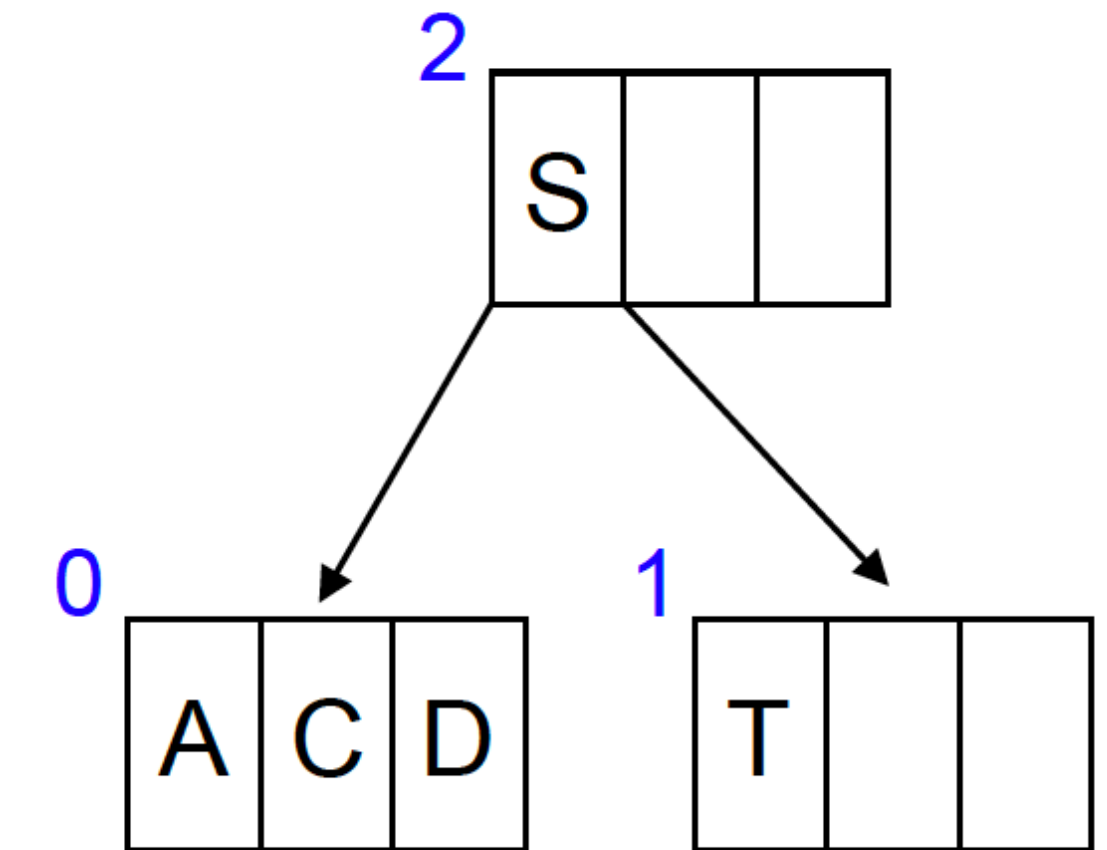
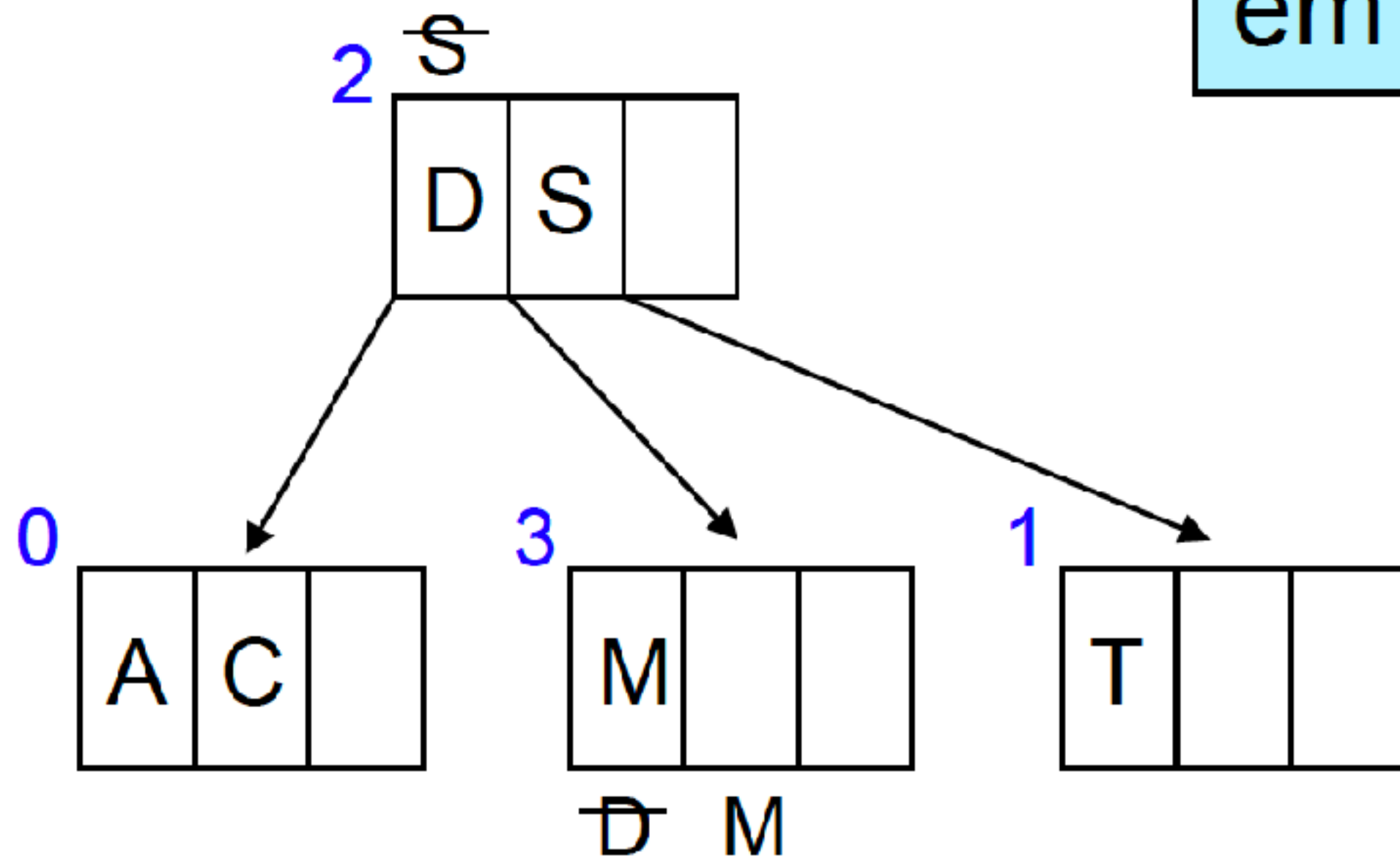




● Passo 4 - inserção de M

- nó folha 0 cheio

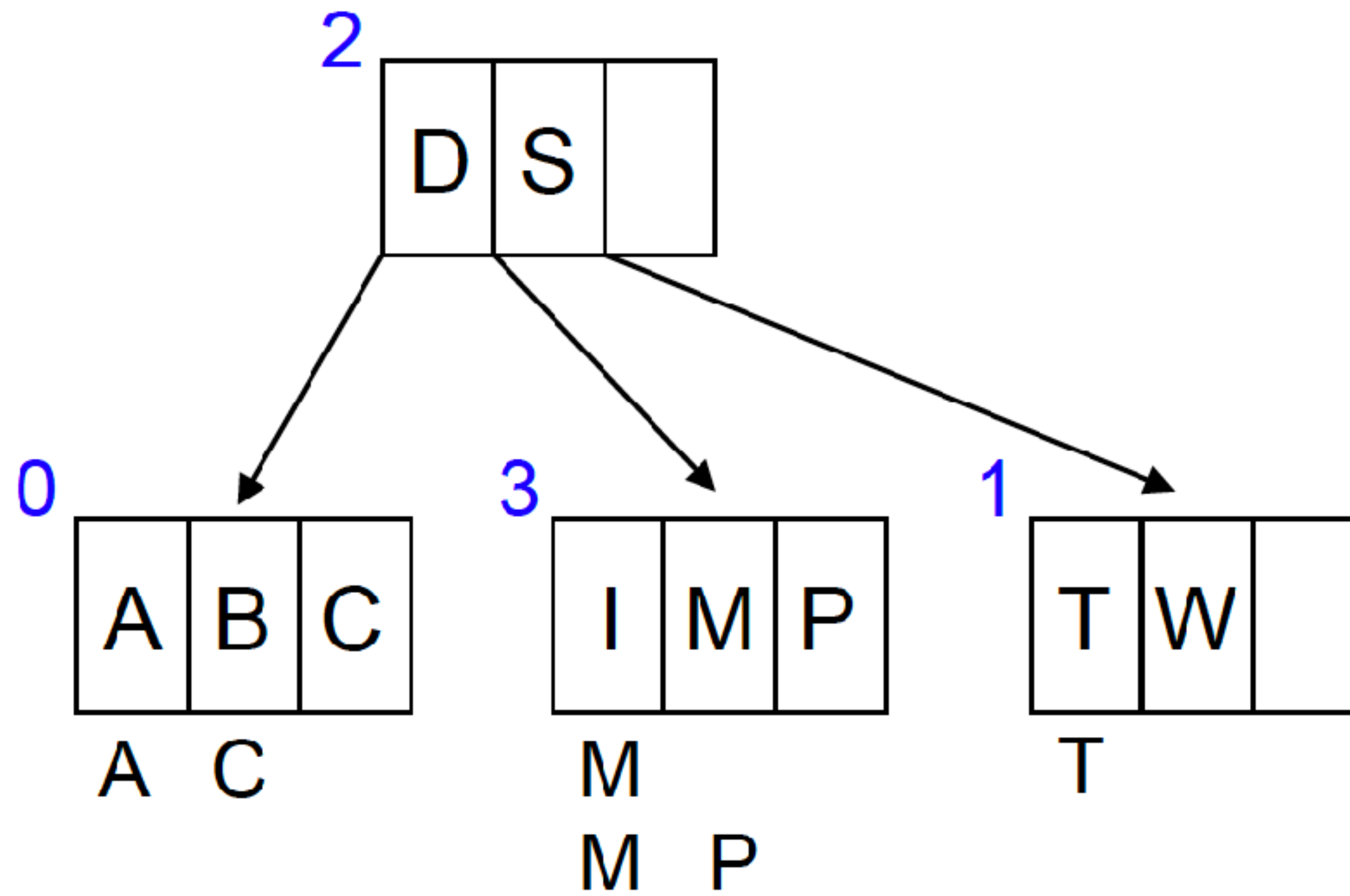
- particionamento do nó
- promoção de D; inserção em nó (raiz) com espaço





● Passo 5 - inserção de P, I, B, W

- nós folhas com espaço

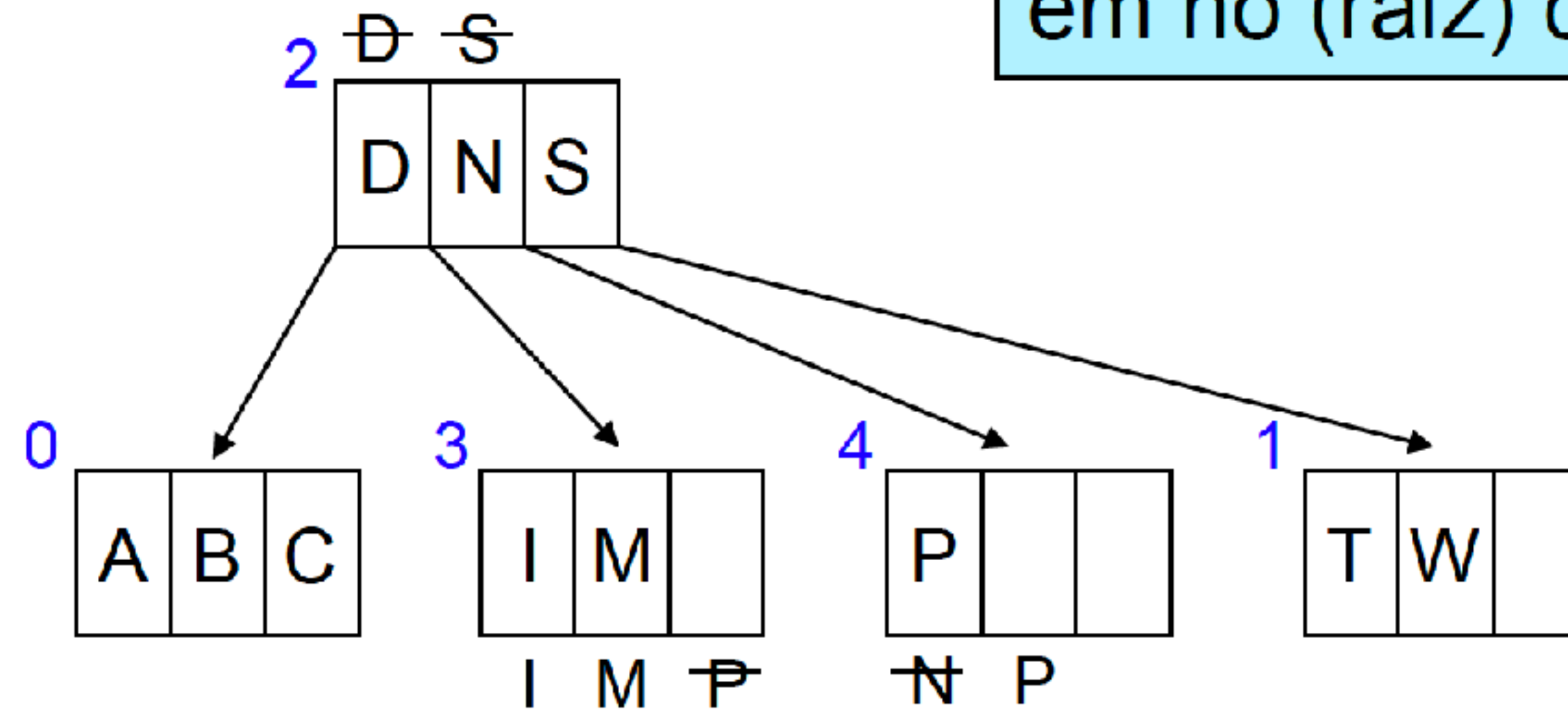




● Passo 6 - inserção de N

- nó folha 3 cheio

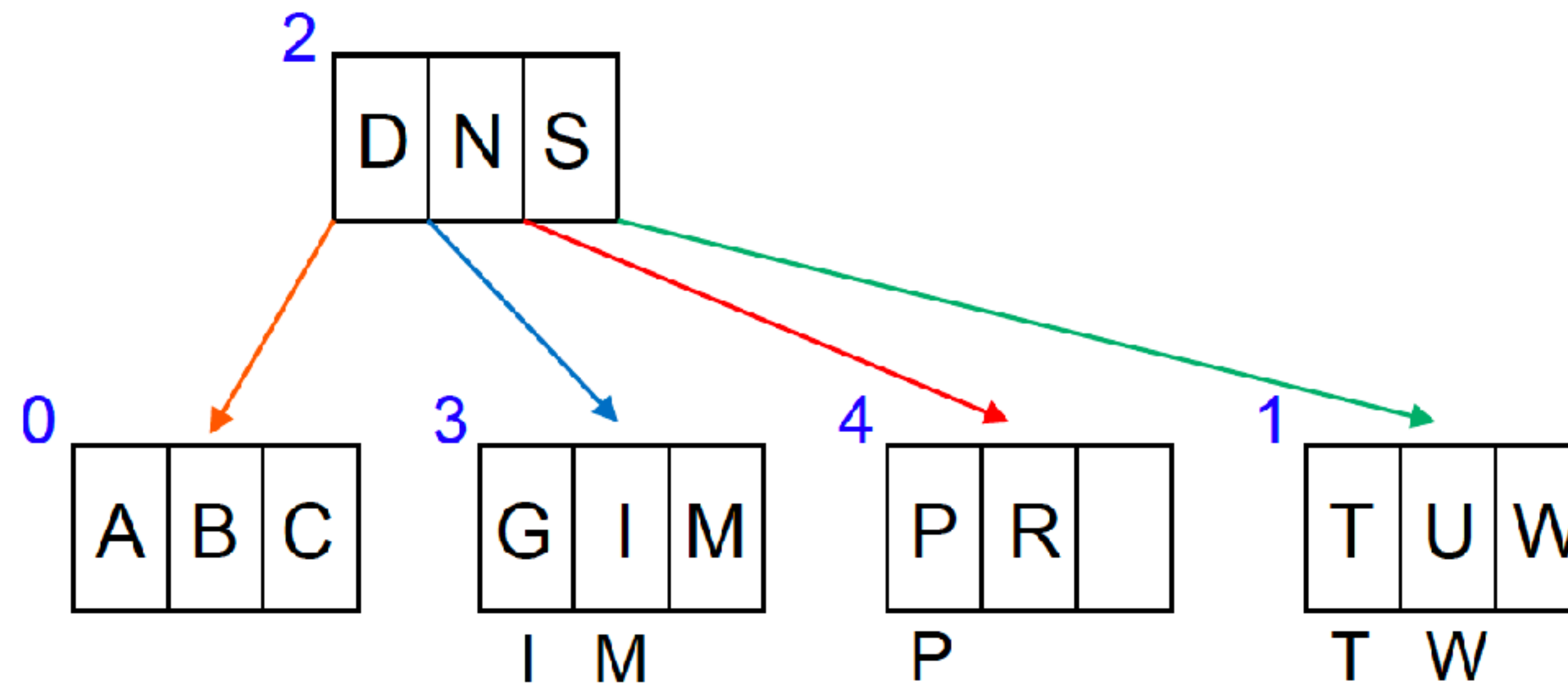
- particionamento do nó
- promoção de N; inserção em nó (raiz) com espaço





● Passo 7 - inserção de G, U, R

- nós folhas com espaço

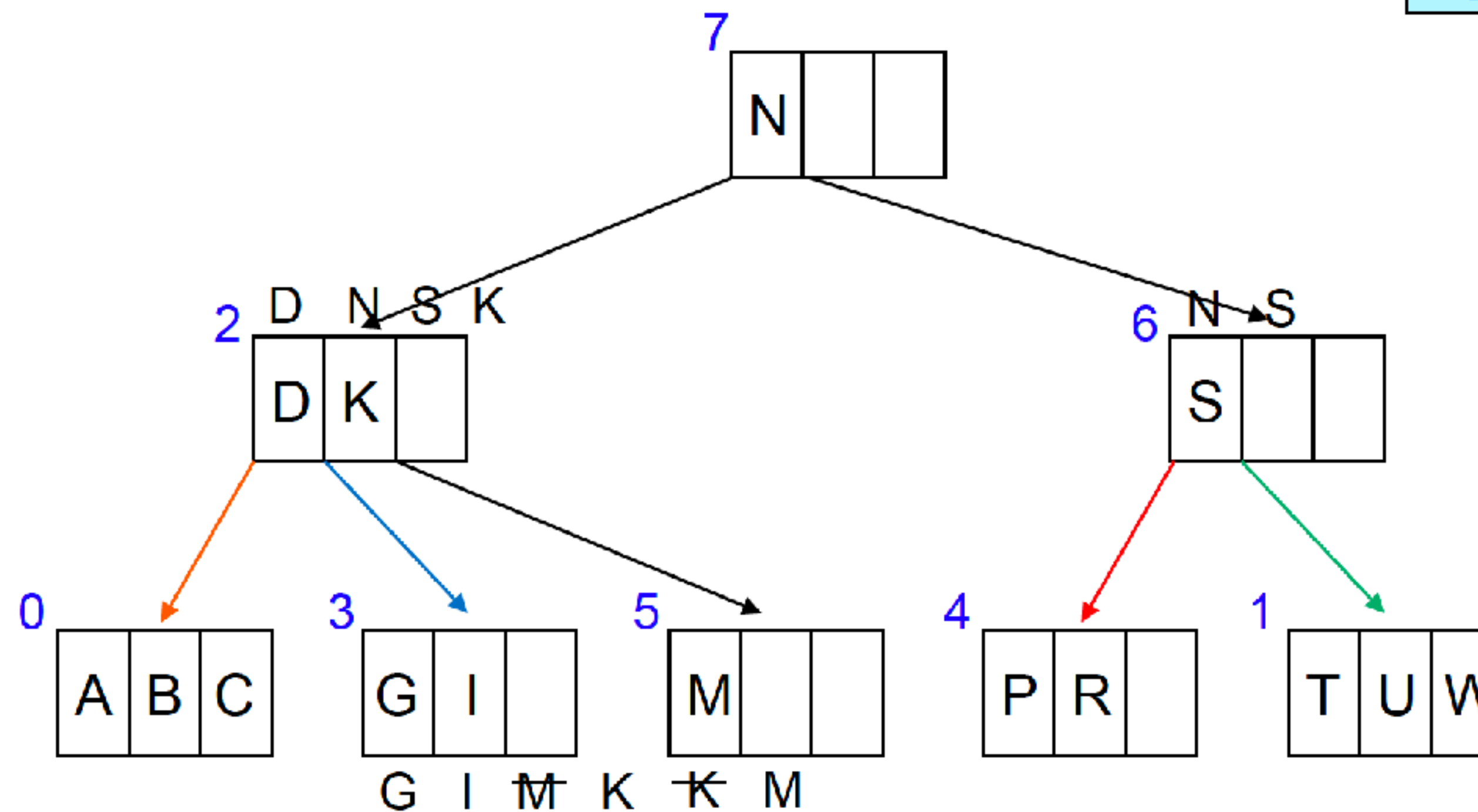




● Passo 8 - inserção de K

- nó folha 3 cheio

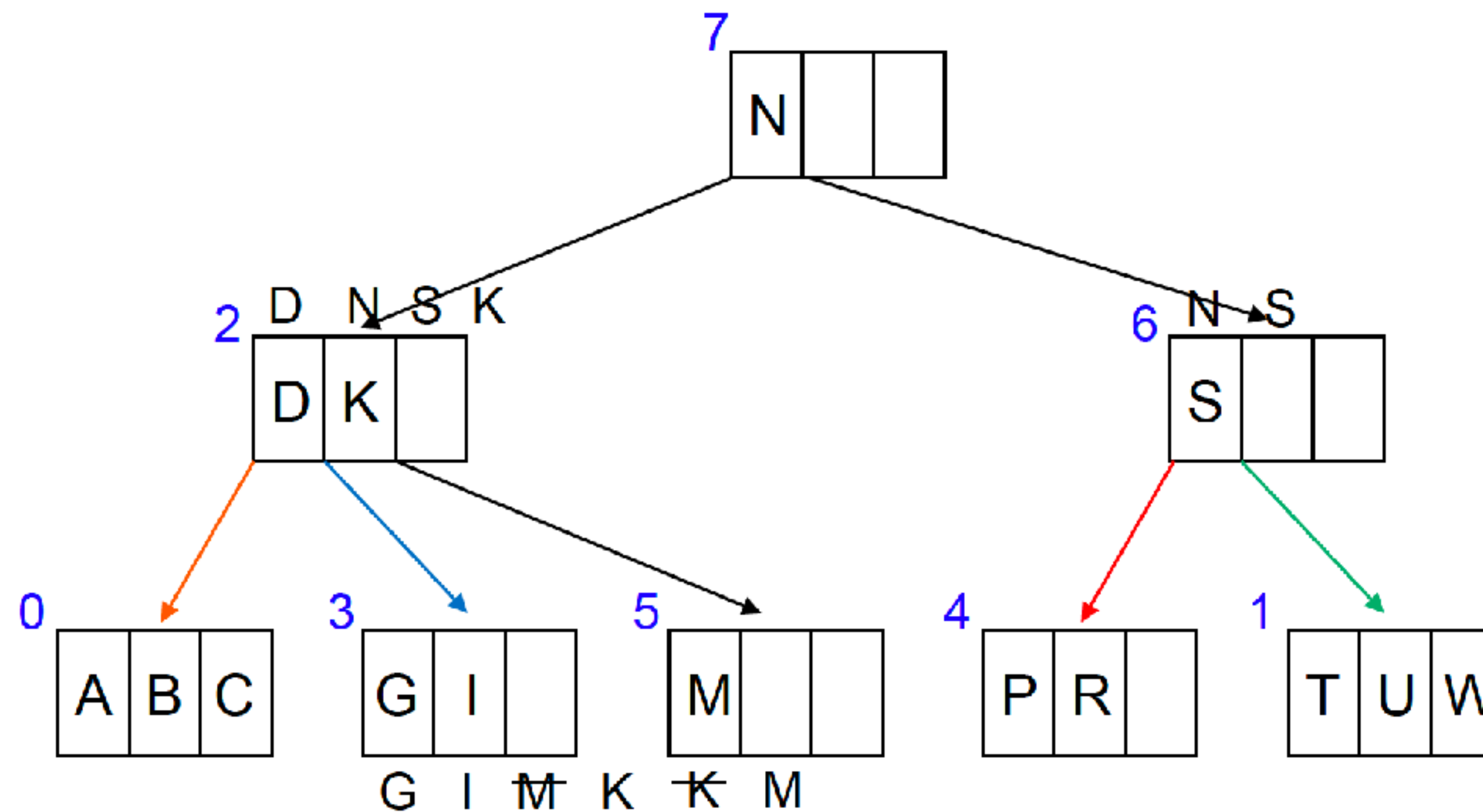
- particionamento do nó 3
- promoção de K
- particionamento do nó 2
- promoção de N





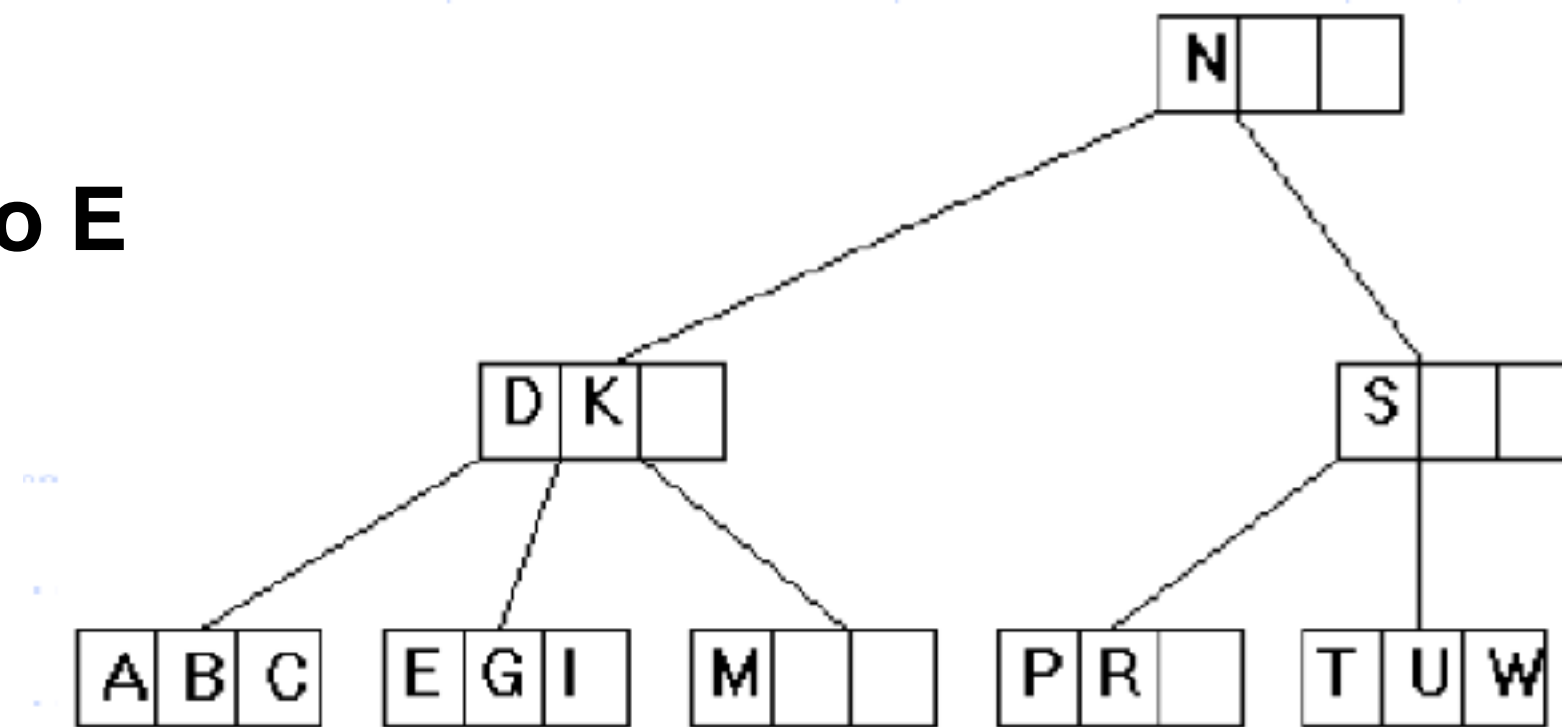
● Finalizar a construção da árvore

● ... **E H O L J Y Q Z F X V**

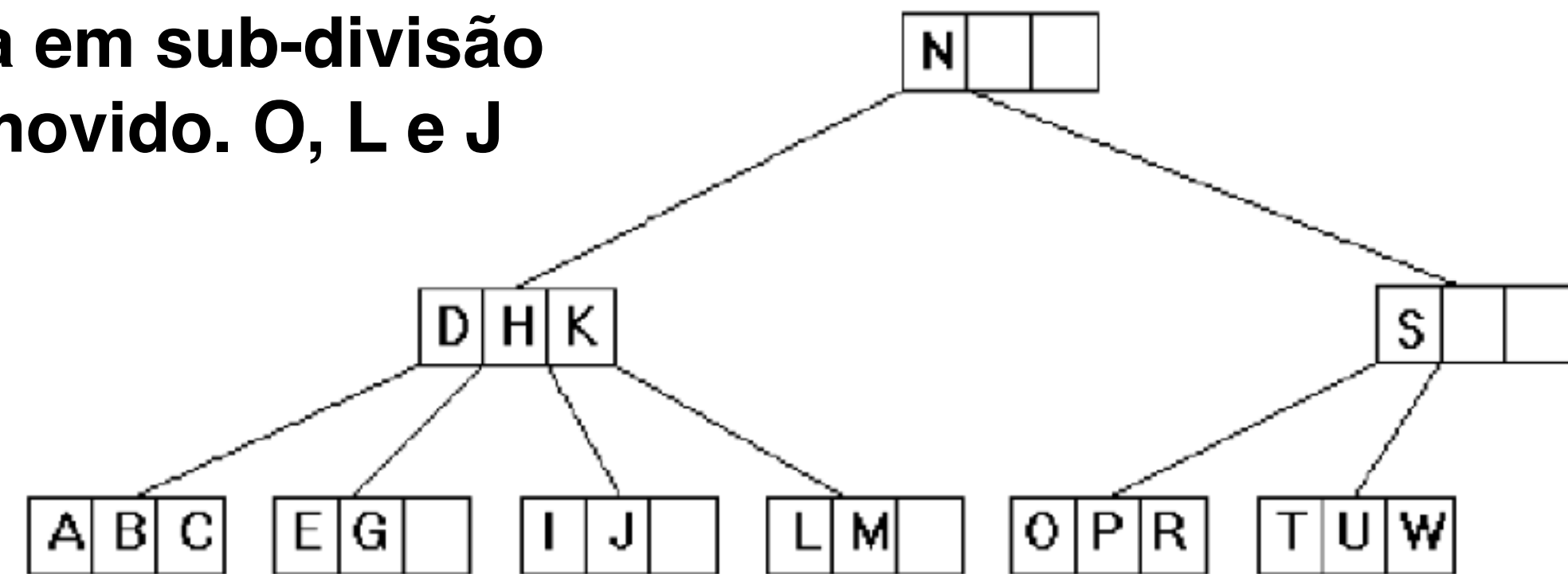




Inserção E

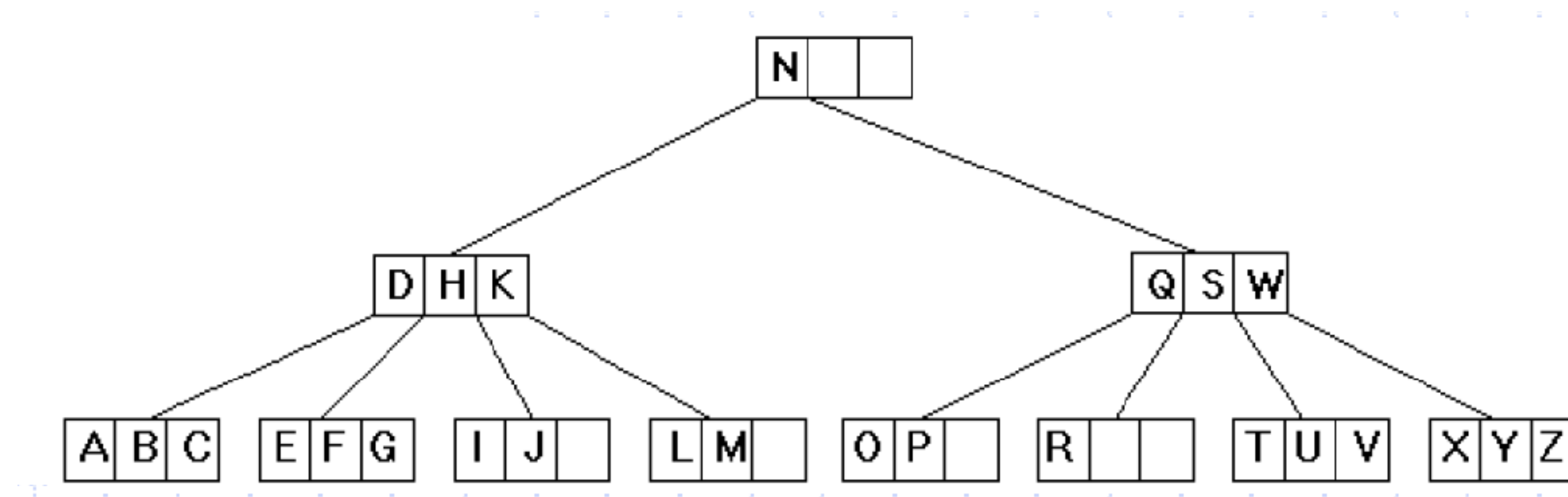


Inserção de H resulta em sub-divisão no nó folha. H é promovido. O, L e J são adicionados.



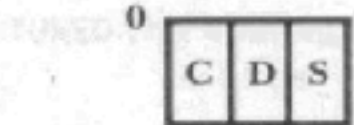


Inserção de Y e Q força mais dois splits nos nós folhas. O restante das letras são adicionados.

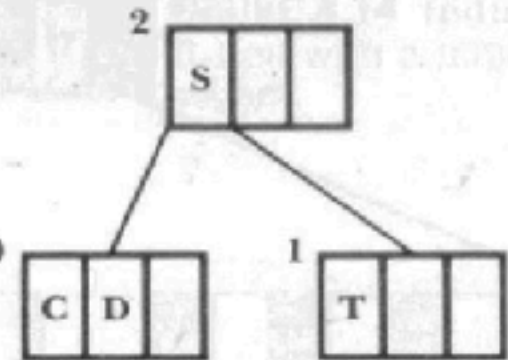




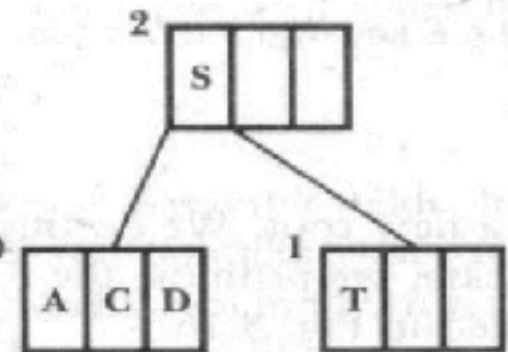
Insertion of C, S, and D into the initial page:



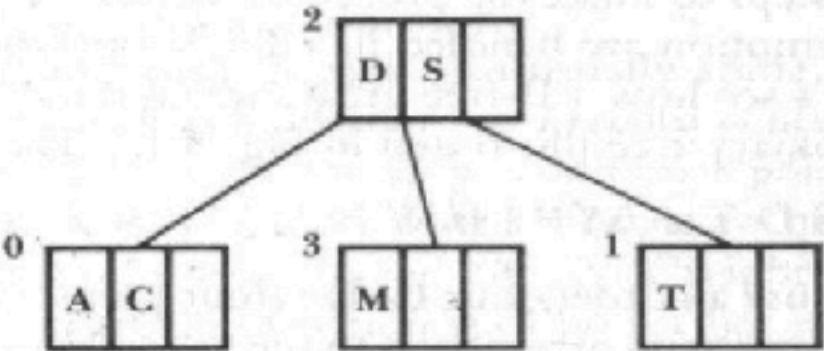
Insertion of T forces the split and the promotion of S:



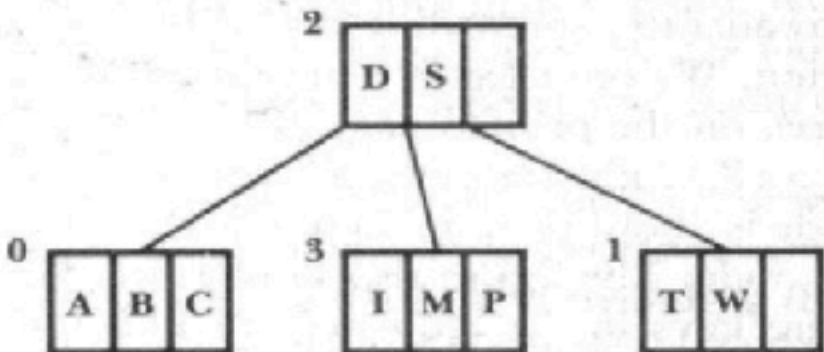
A added without incident:



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



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



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

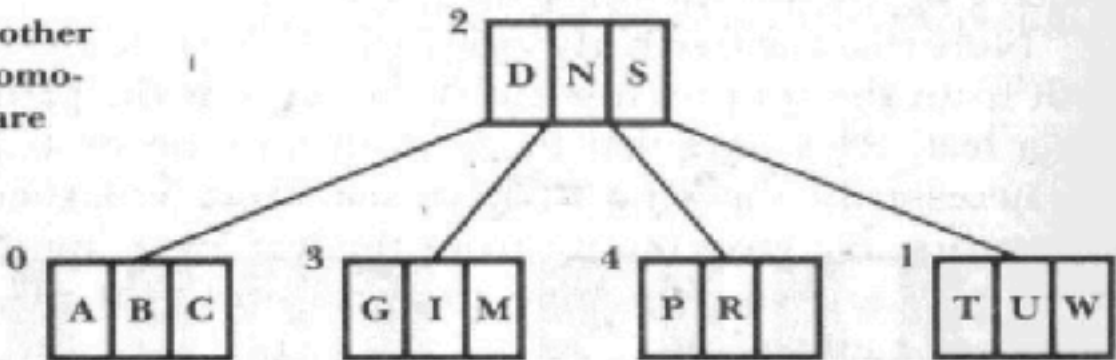
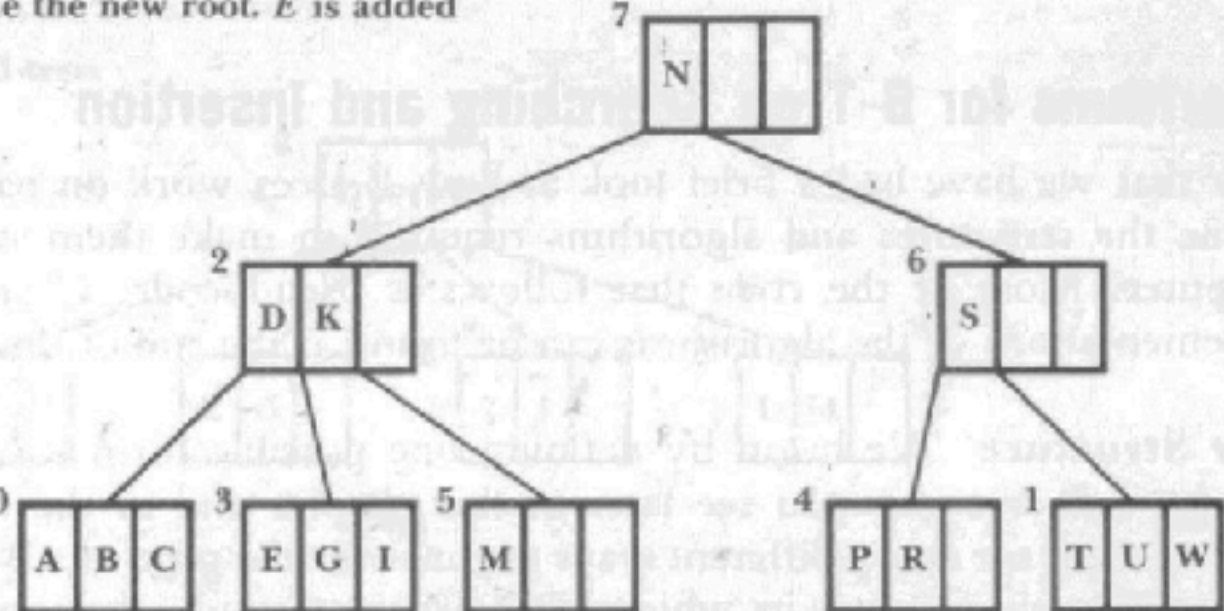
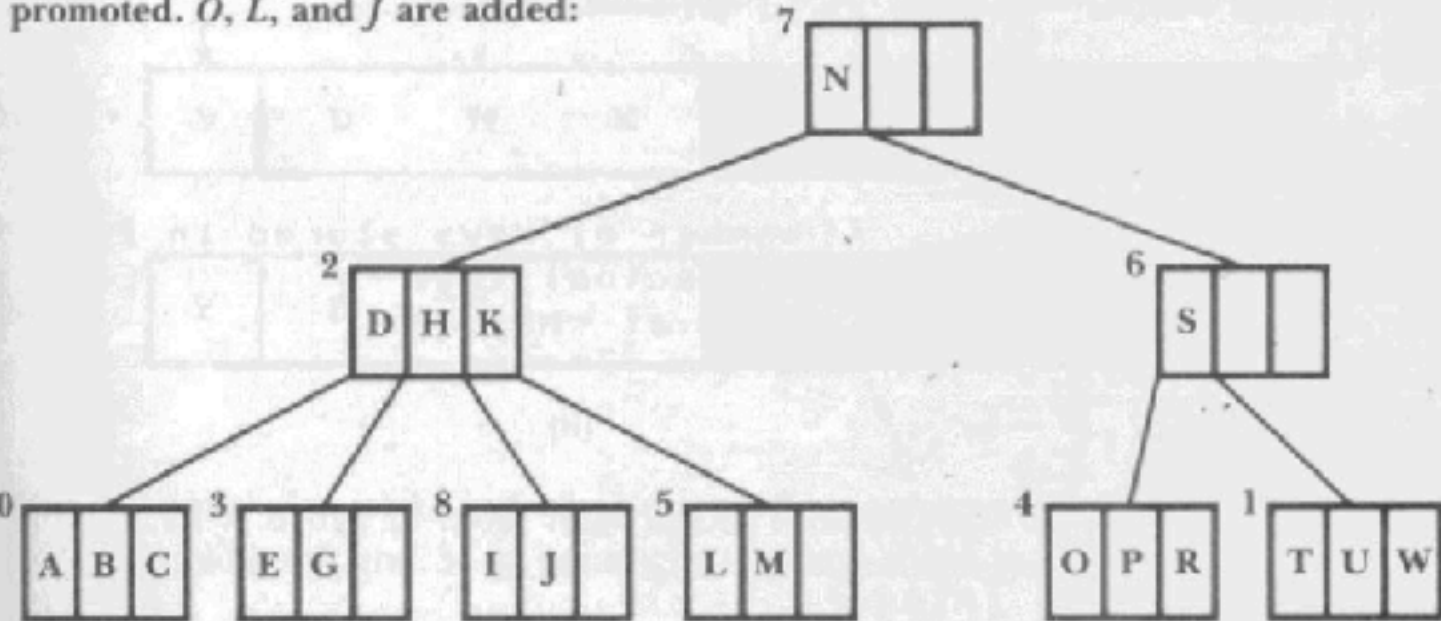


FIGURE 8.17 Growth of a B-tree, part I. The tree grows to a point at which splitting of the root is imminent.

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:



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



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

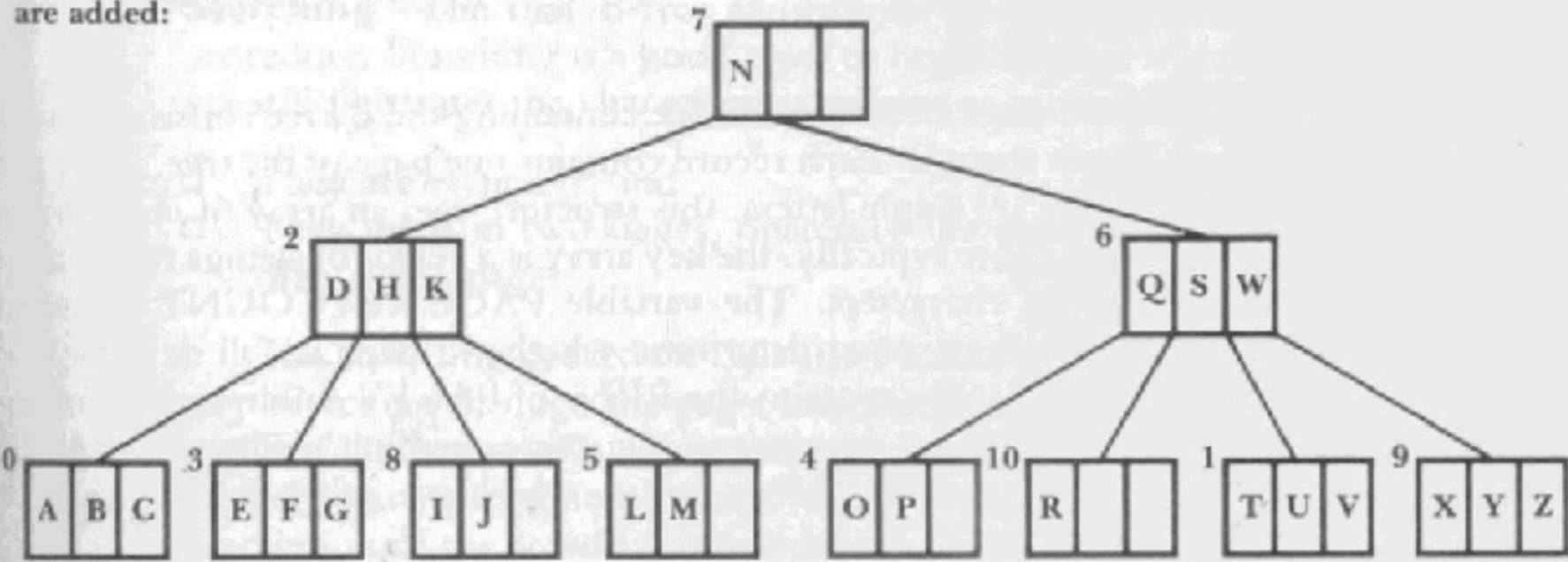
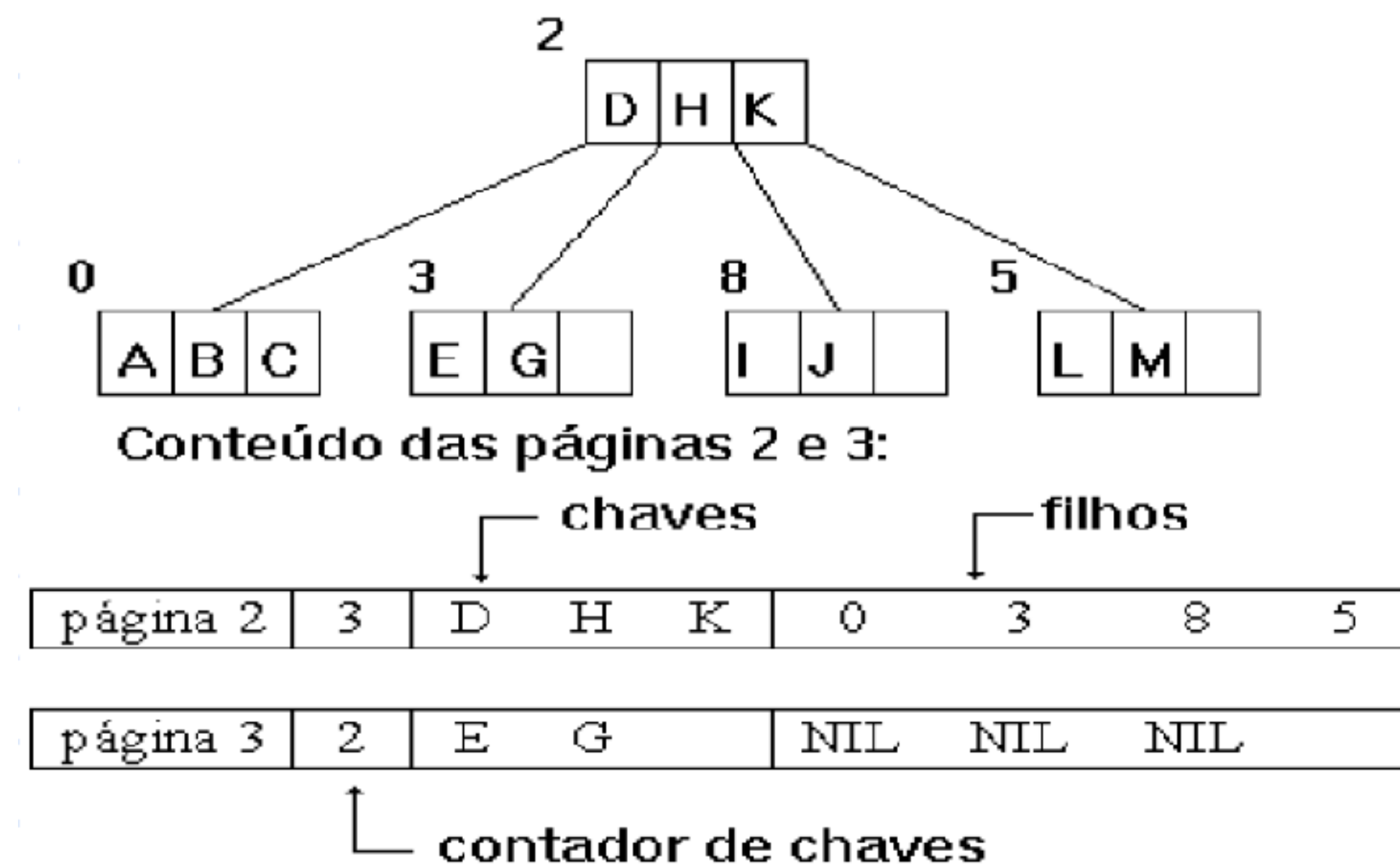


FIGURE 8.18 Growth of a B-tree, part II. The root splits to add a new level; remaining keys are inserted.



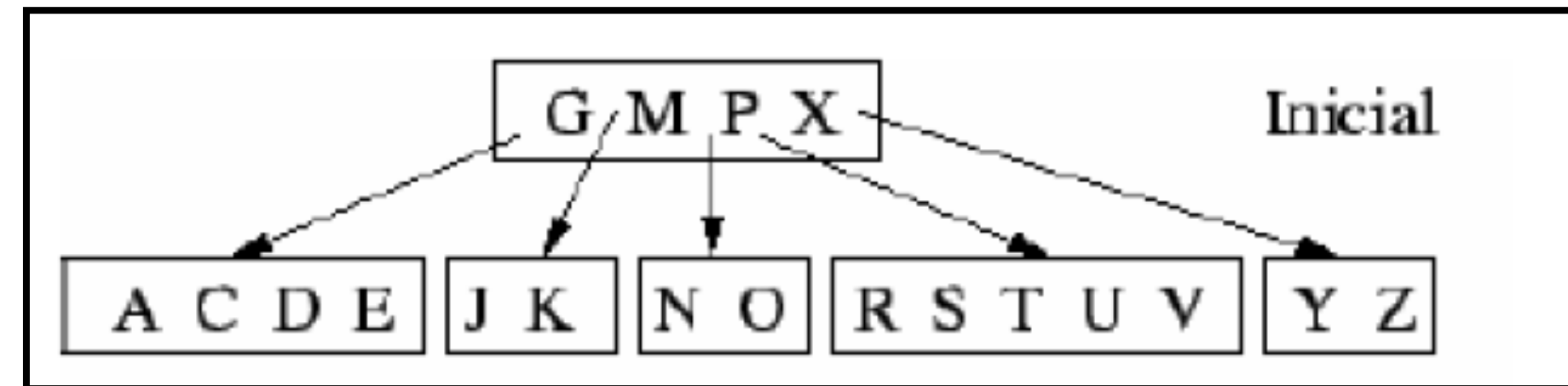


- Um exemplo de parte de uma árvore-B de ordem 4 é dado na figura abaixo. Um nó interno e 4 nós folha, são explicitados os RRN de cada página (o RRN é um número de página válido, e os ponteiros das folhas apontam para nil (que pode ser -1)).
- O arquivo que contém a árvore-B é um arquivo com registros de tamanho fixo, sendo que cada registro contém uma página da árvore.

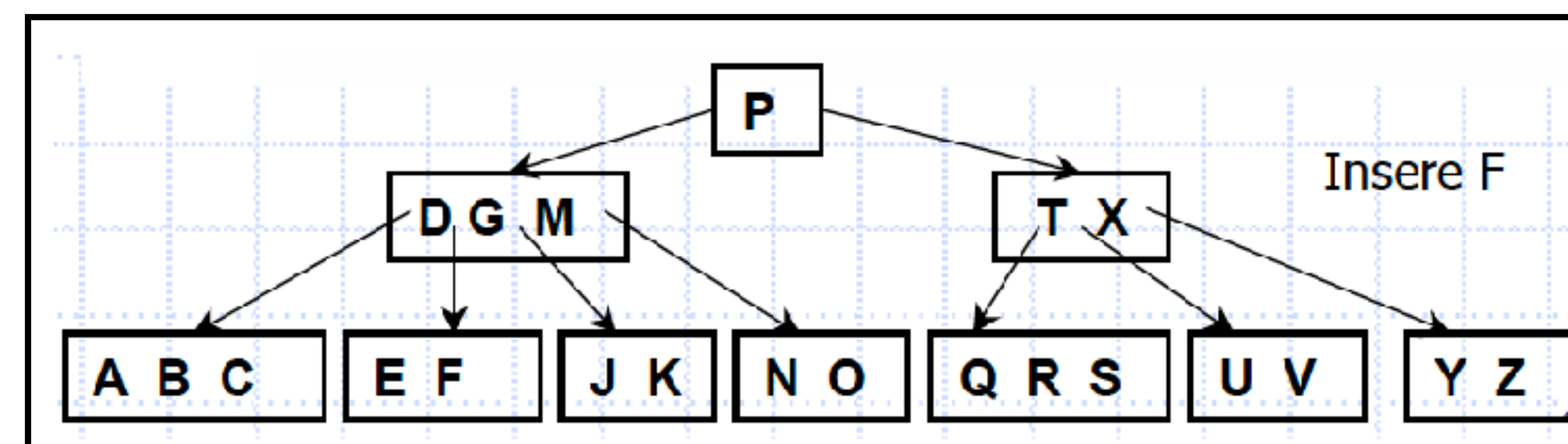
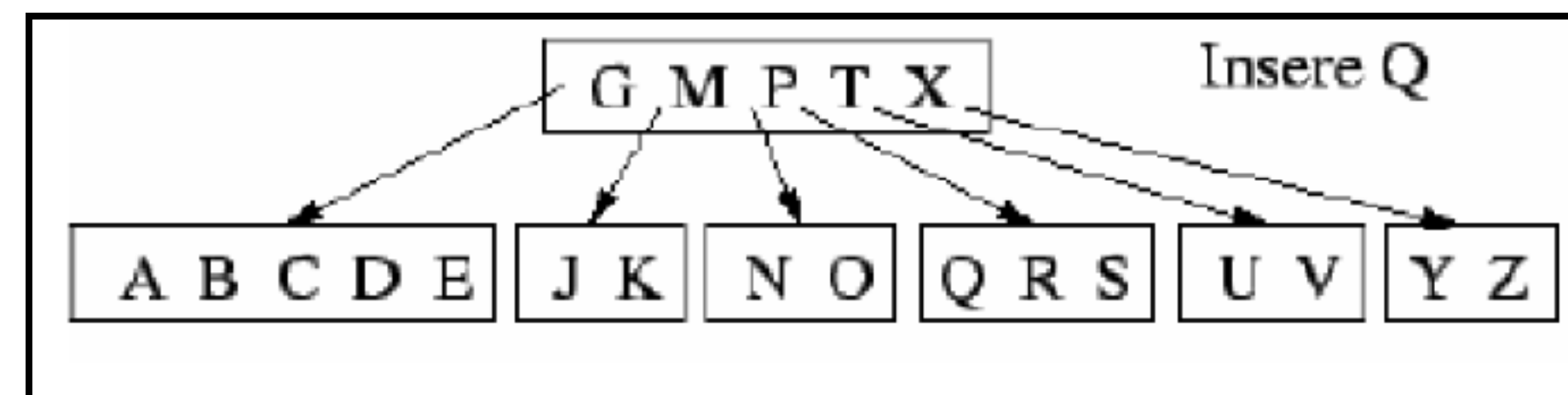
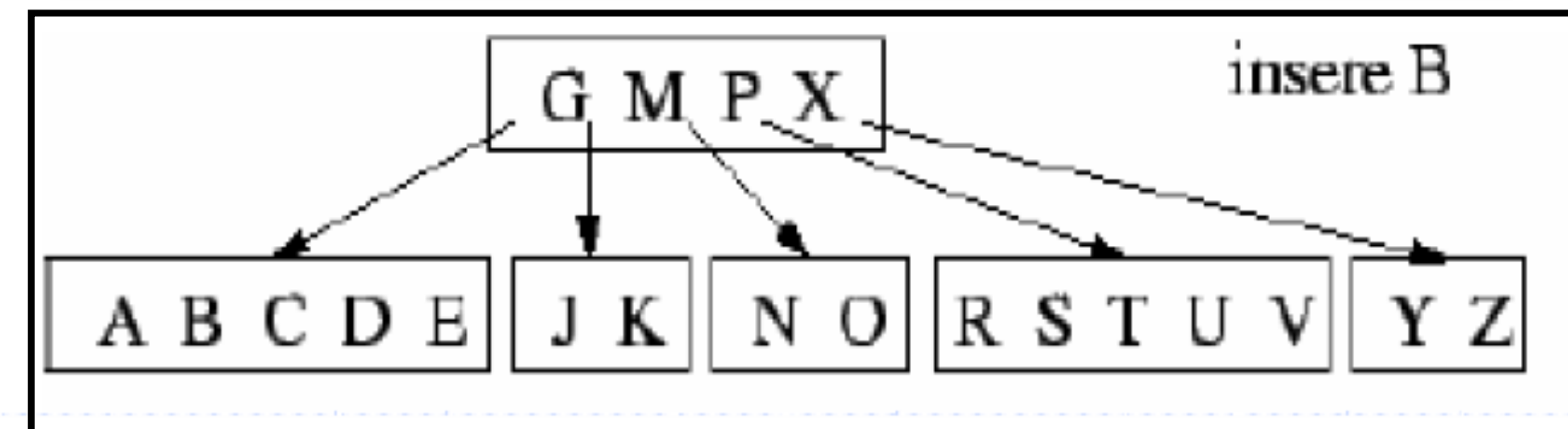
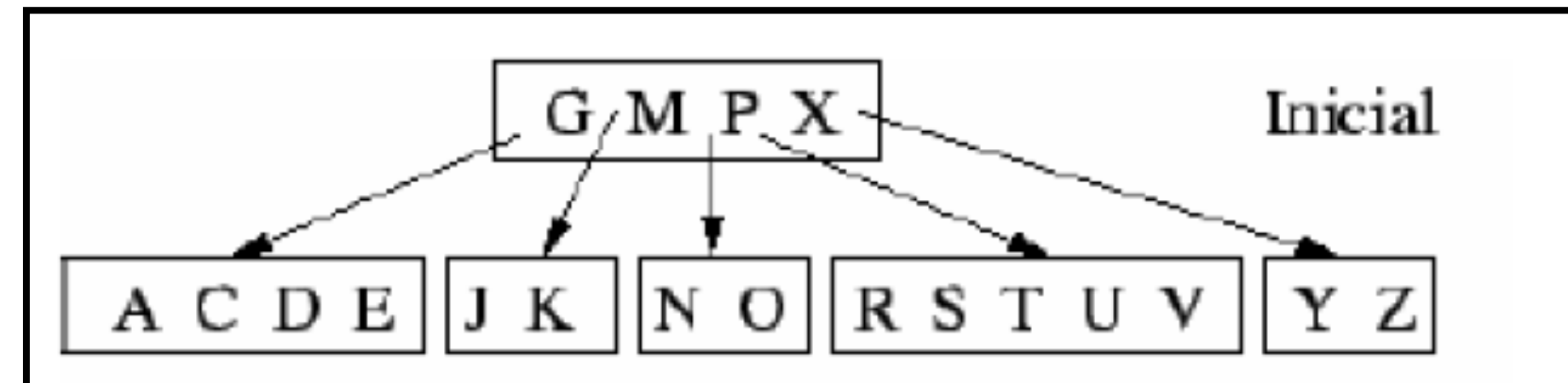




- Exercício: incluir novos (**B**, **Q**, **F**) elementos em uma árvore-B de ordem 6:



● Exercício: incluir novos (**B**, **Q**, **F**) elementos em uma árvore-B de ordem 6:





- Na árvore-B do exemplo anterior (ordem 4), insira a chave \$, sendo que $\$ < A$



© Insira as seguintes chaves em um índice árvore-B

- **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**

- diferentemente do exemplo anterior, escolha o último elemento do primeiro nó para promoção durante o particionamento do nó.



- Esboce um algoritmo recursivo de busca em uma árvore-B
- Esboce um algoritmo de inserção de chaves em uma árvore-B



- FOLK, M.J. File Structures, Addison-Wesley, 1992.
- File Structures: Theory and Practice”, P. E. Livadas, Prentice-Hall, 1990;
- Contém material extraído e adaptado das notas de aula dos professores Moacir Ponti, Thiago Pardo, Leandro Cintra, Thelma Cecília Chiossi e Maria Cristina de Oliveira.