Algoritmos e Estruturas de Dados

Árvore B

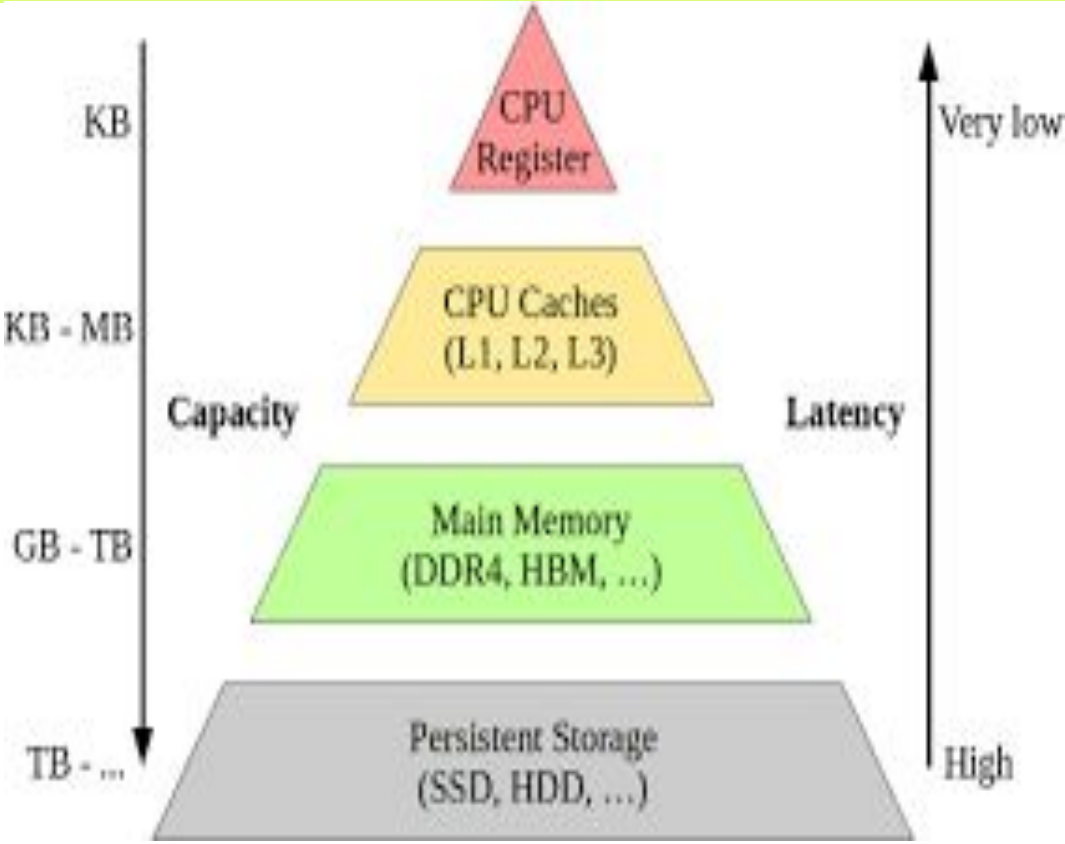
Slides baseados em:

•**CORMEN, H.T.; LEISERSON, C.E.; RIVEST, R.L. Introduction to Algorithms, MIT Press, McGraw-Hill, 1999.**

Profa. Karina Valdivia Delgado
EACH-USP

Memória

- Memória Cache
  - Fica próxima do processador para ter acesso rápido.
- Memória RAM (Random-Access Memory)
  - Memória volátil, é perdida quando o computador é desligado.
- Memória secundária
  - HDD (Hard Disk Drive) ou SDD (Solid-State Drive)
  - Memória permanente onde gravamos arquivos



Memória secundária

- Desejamos armazenar registros na memória secundária porque ela não cabe na memória principal ou porque desejo que seja armazenada de maneira permanente.
- A memória secundária é dividida em páginas usualmente de 2MB a 16MB.
- Se a página está na memória principal, podemos acessá-la. Senão, precisamos lê-la (DISK-READ)
- O acesso a memória secundária é muito mais lento, assim desejamos ler o menor número de páginas possível.

Árvores B

- Árvores binárias de busca, balanceadas ou não, não são adequadas para o armazenamento e busca de dados em memória secundária.
- Uma vez que o acesso a disco é uma operação cara, então ao invés de buscar um dado de cada vez, procura-se transferir em cada acesso uma quantidade maior de dados.
- Uma árvore B é uma generalização de uma árvore binária de busca balanceada criada por R. Bayer, E. McCreight.
- Uma árvore B é uma estrutura de dados muito bem sucedida e tem diversas variantes que são usadas na implementação de bases de dados de uso comercial.

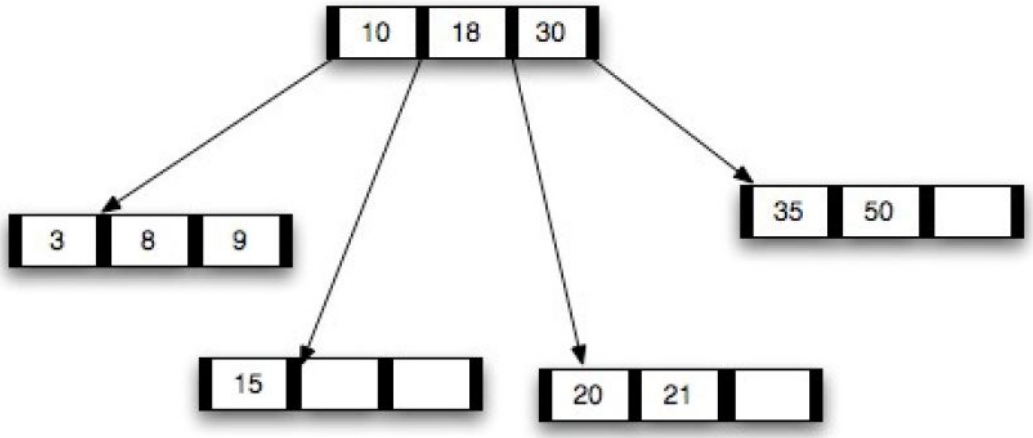
Árvore B

- Cada nó da árvore B é também conhecido pelo nome de página.
- Cada página pode conter uma grande quantidade de chaves.
- A chave tem um papel importante na busca pois ela identifica unicamente um elemento de informação. Naturalmente, além da chave, podemos ter outras informações associadas. Ex: A chave dos funcionários de uma empresa poderia ser o CPF.
- Nos exemplos são mostradas apenas as chaves, mas subentende-se que pode haver outras informações associadas, as quais podem estar armazenadas junto com a chave.

Definição

- Uma *árvore* B é uma árvore com as seguintes propriedades:

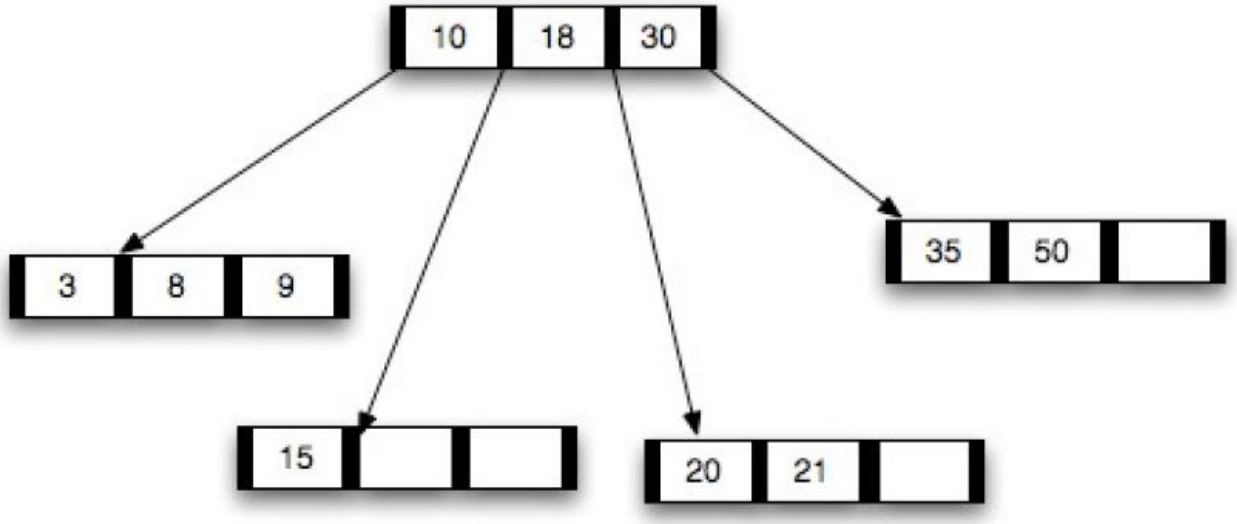
  1. Cada nó x contém os seguintes campos:
     - $n[x]$, o número de chaves atualmente armazenadas no nó x;
     - as $n[x]$ chaves, armazenadas em ordem não decrescente, de modo que $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$;
     - $leaf[x]$, um valor booleano indicando se x é uma folha (TRUE) ou um nó interno (FALSE).
     - se x é um nó interno, x contém $n[x] + 1$ ponteiros $c_1[x], c_2[x], \ldots c_{n[x]+1}[x]$ para seus filhos.



Definição

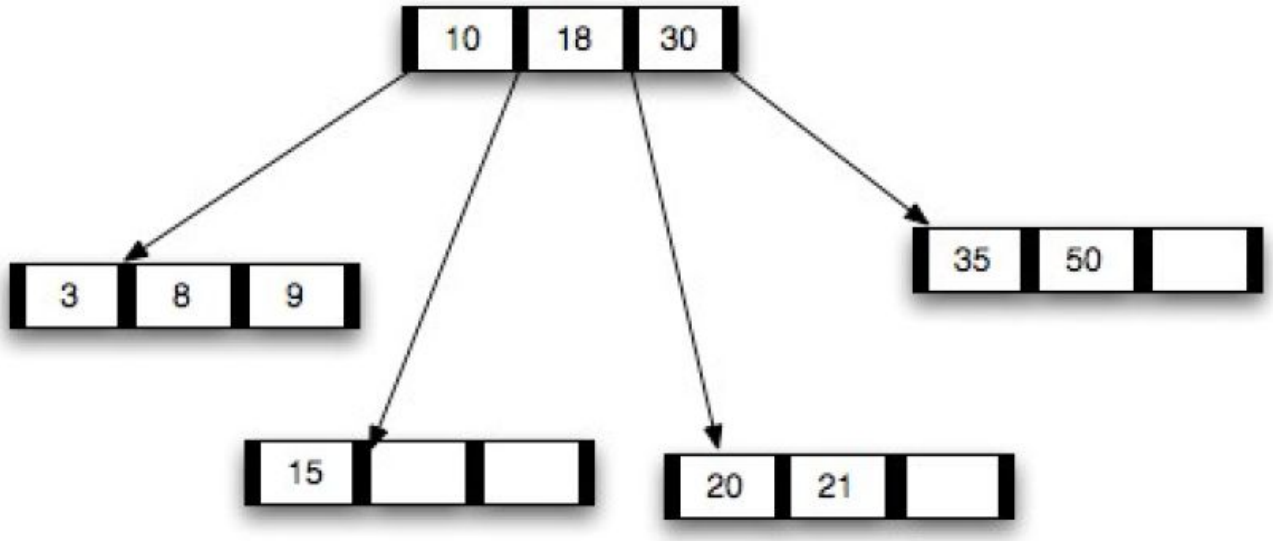
2. As chaves $key_i[x]$ separam as faixas de valores armazenados em cada subárvore: denotando por k_i uma chave qualquer armazenada na subárvore com nó $c_i[x]$, tem-se

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}$$



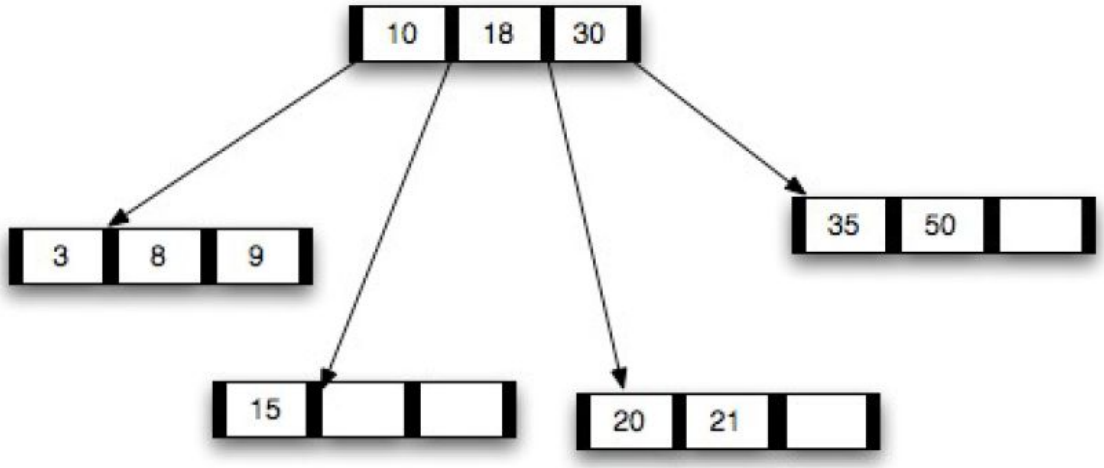
Definição

3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h.



Definição

4. Há um limite inferior e superior no número de chaves que um nó pode conter, expressos em termos de um inteiro fixo $t \geq 2$ chamado o *grau mínimo* (ou *ordem*) da árvore.
   - Todo nó que não seja a raiz deve conter pelo menos $t - 1$ chaves. Todo nó interno que não seja a raiz deve conter pelo menos t filhos.
   - Todo nó deve conter no máximo $2t - 1$ chaves (e portanto todo nó interno deve ter no máximo $2t$ filhos). Dizemos que um nó está *cheio* se ele contiver exatamente $2t - 1$ chaves

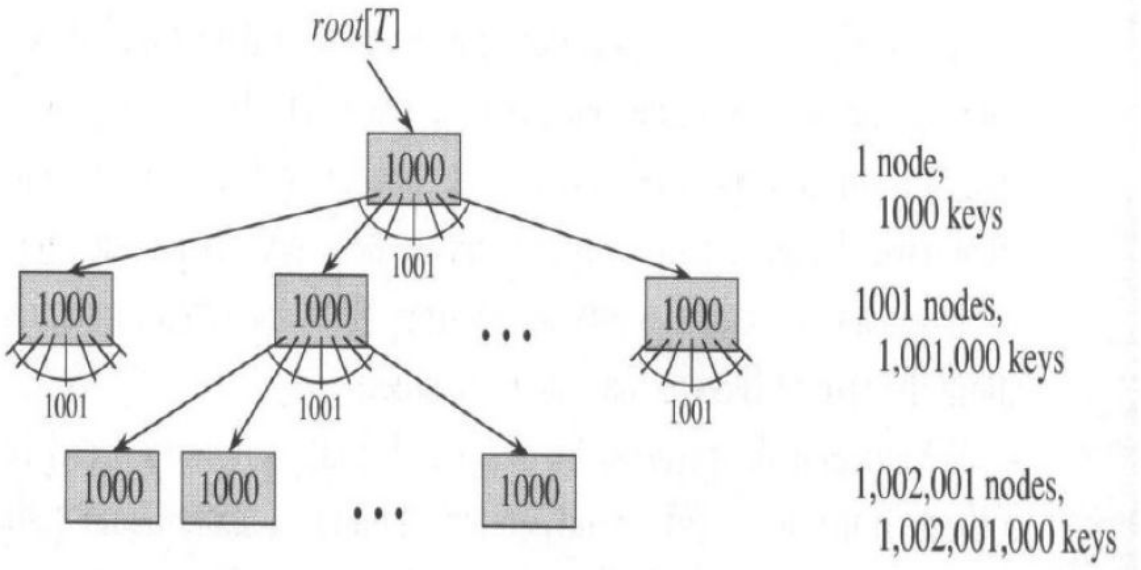


t=2

Observação

- Número máximo de chaves (e filhos) por nó deve ser proporcional ao tamanho da página. Valores usuais de 50 a 2000. Fatores de ramificação altos reduzem drasticamente o número de acessos ao disco.

  Por exemplo, uma árvore B com fator de ramificação 1001 e altura 2 pode armazenar $\geq 10^9$ chaves. Uma vez que a raiz pode ser mantida permanentemente na memória primária, bastam *dois* acessos ao disco para encontrar qualquer chave na árvore.



t=1001

Altura da árvore B

- **Teorema:** Para toda árvore B de grau mínimo $t \geq 2$ contendo n chaves, sua altura h máxima será:

$$h \leq log_t \frac{n+1}{2}$$

Altura da árvore B

- **Teorema:** Para toda árvore B de grau mínimo $t \geq 2$ contendo n chaves, sua altura h máxima será:

$$h \leq log_t \frac{n+1}{2}$$

Demonstração: Se uma árvore B tem altura h:

- Sua raiz contem pelo menos uma chave e todos os demais nós contêm pelo menos $t-1$ chaves.
- Logo, há pelo menos 2 nós no nível 1, pelo menos $2t$ nós no nível 2, etc, até o nível h, onde haverá pelo menos $2t^{h-1}$ nós.

- Assim, o número n de chaves satisfaz a desigualdade:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\frac{t^h - 1}{t-1} = 2t^h - 1$$

Obs: Usamos acima a igualdade: $\sum_{i=1}^{h} t^{i-1} = \frac{t^h - 1}{t-1}$.

- Logo,

$$t^h \leq (n+1)/2 \Rightarrow h \leq log_t(n+1)/2.$$

Convenção

- A árvore B não é armazenada na memória principal e precisa ser lida do disco
- A raiz da árvore está sempre na memória principal
- Duas operações serão usadas:
  - DISK-READ(x): para ler o objeto x que reside no disco e inseri-lo na memória principal
  - DISK-WRITE(x): para gravar em disco o objeto x

Criação de uma árvore B vazia

```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

Busca em uma árvore B

Busca em uma árvore B é similar a busca em uma árvore binária de busca (ABB):

- Verificar se a chave está no nó x
- Se não estiver, buscar no filho correto

Busca em uma árvore B

- B-Tree-Search(x, k): tem como parâmetros um ponteiro para o nó x raiz de uma subárvore e uma chave k a ser procurada na subárvore. Se k está na subárvore, retorna o par ordenado (y, i) composto pelo ponteiro do nó y e o índice i tal que $key_i[y] = k$. Caso contrário, retorna NIL.
- Chamada inicial: B-Tree-Search$(root[T], k)$.

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

Verificar se a chave k está no nó x

Busca em uma árvore B

- B-Tree-Search(x, k): tem como parâmetros um ponteiro para o nó x raiz de uma subárvore e uma chave k a ser procurada na subárvore. Se k está na subárvore, retorna o par ordenado (y, i) composto pelo ponteiro do nó y e o índice i tal que $key_i[y] = k$. Caso contrário, retorna NIL.
- Chamada inicial: B-Tree-Search$(root[T], k)$.

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

Se a chave k não está no nó x e x é uma folha, não encontrei a chave

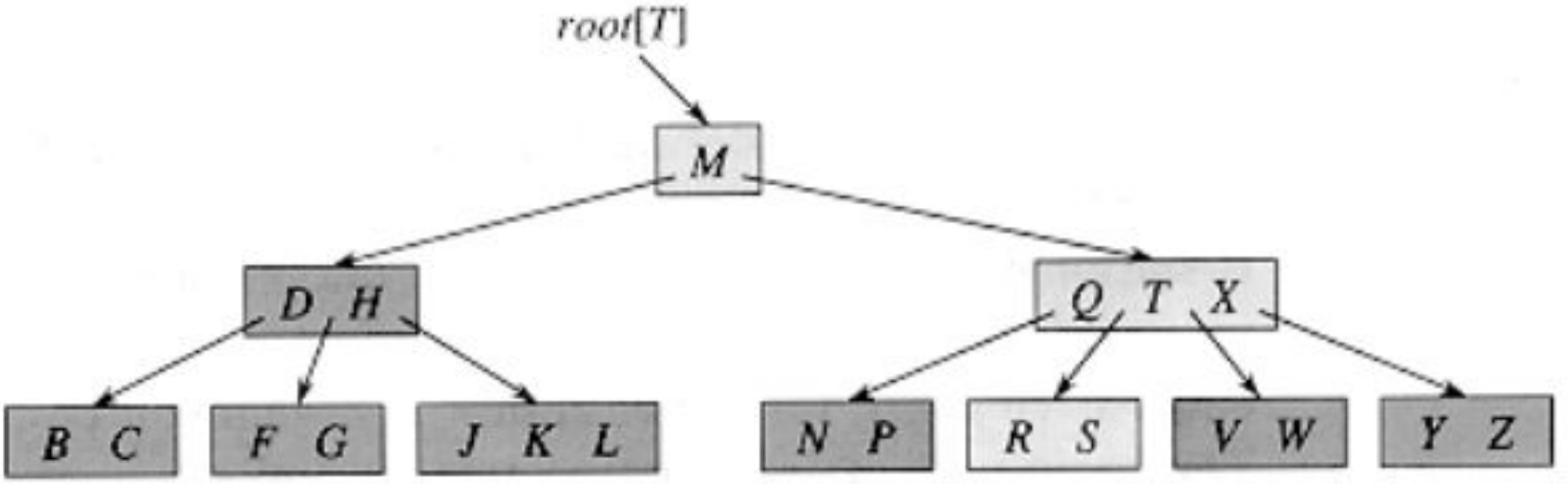
Busca em uma árvore B

- B-Tree-Search(x, k): tem como parâmetros um ponteiro para o nó x raiz de uma subárvore e uma chave k a ser procurada na subárvore. Se k está na subárvore, retorna o par ordenado (y, i) composto pelo ponteiro do nó y e o índice i tal que $key_i[y] = k$. Caso contrário, retorna NIL.
- Chamada inicial: B-Tree-Search$(root[T], k)$.

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

Caso contrário, trazemos o filho correto para a memória RAM e procuramos a chave k nesse filho.

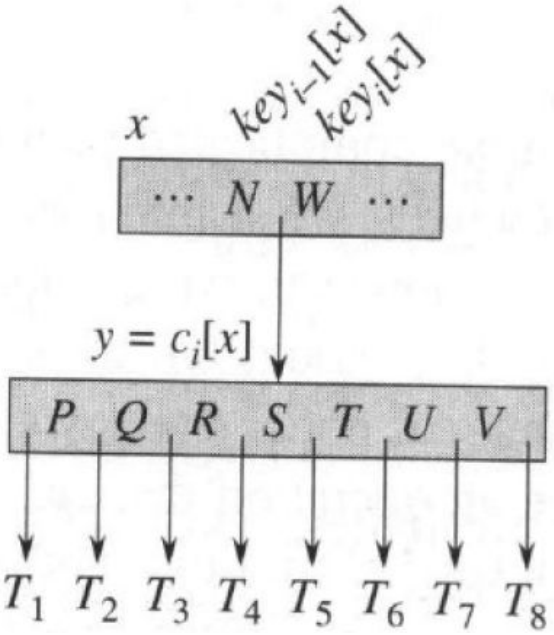
Busca em uma árvore B



Inserção de uma chave em uma árvore B

As inserções ocorrem sempre nas folhas.

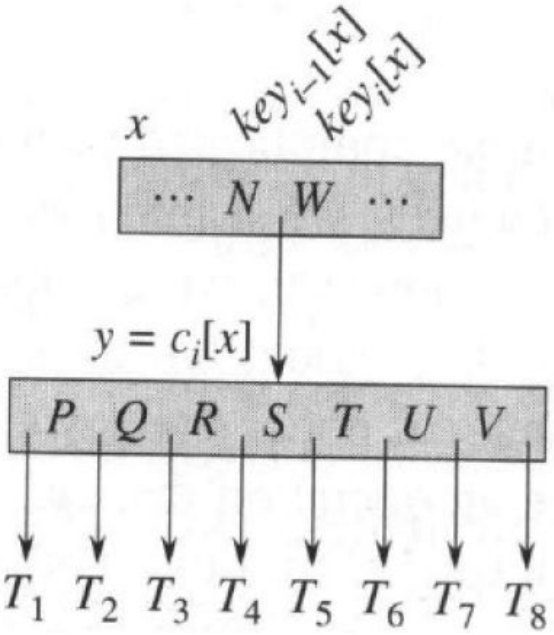
Ex: considere a seguinte árvore B com t=4, inserir “O” nesta árvore.



Inserção de uma chave em uma árvore B

As inserções ocorrem sempre nas folhas.

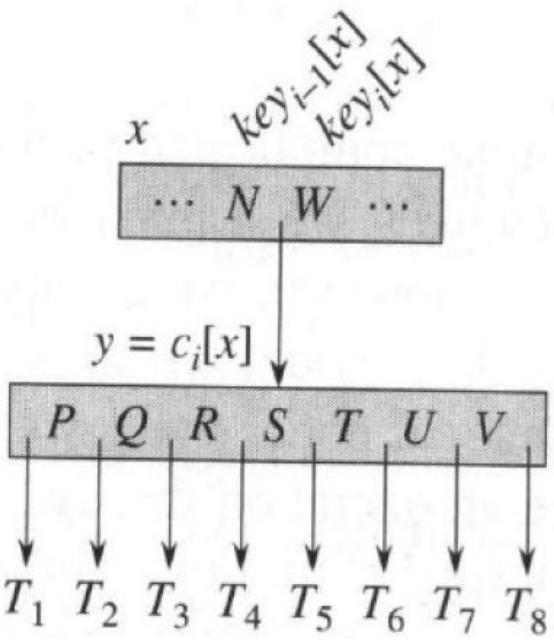
Ex: considere a seguinte árvore B com t=4, inserir “O” nesta árvore.



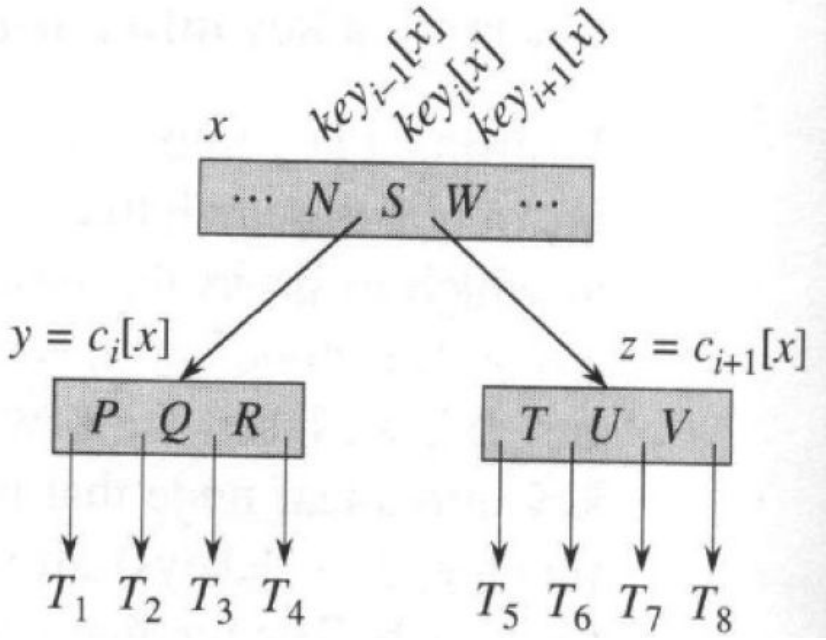
resolvendo o problema do nó cheio

Inserção de uma chave em uma árvore B

Como não podemos inserir uma chave em um nó folha completo, precisamos de um método que divide o nó y, que está completo, ao redor de sua chave mediana ($key_t[y]$) em dois nós que têm t-1 chaves cada. A chave mediana se desloca para o nó pai de y, supondo que o pai não é completo.



resolvendo o problema do nó cheio

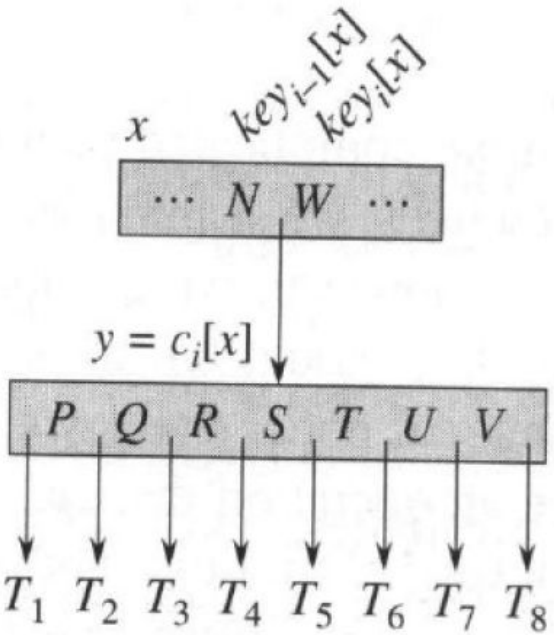


Inserção de uma chave em uma árvore B

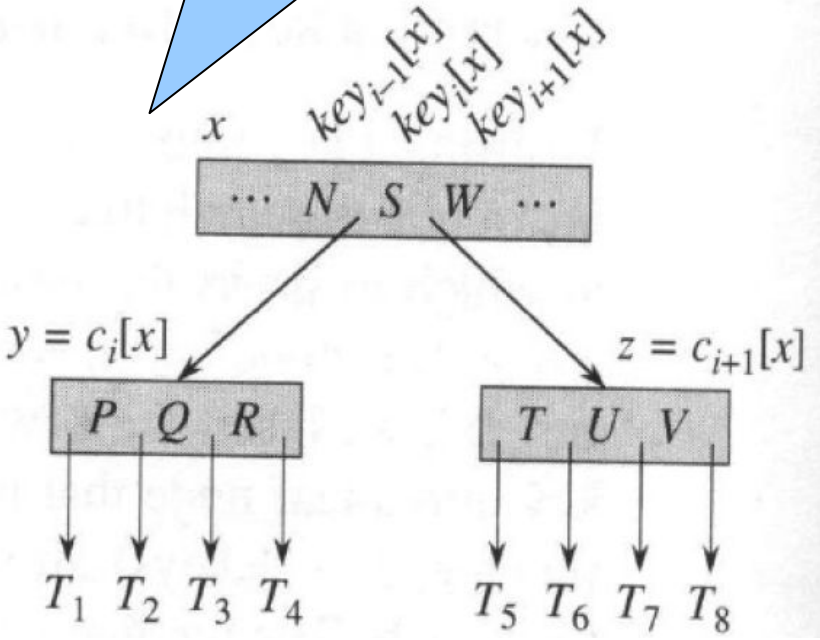
Como não pode
completo, preci
que está compl
($key_t[y]$) em dois nós que têm t-1 chaves
mediana se desloca para o nó pai de y, s
não é completo.

Se o nó pai estiver cheio teria que dividir também o pai, o que poderia causar subdivisões em cascata. Essas subdivisões podem alterar em qual nó a chave deve ser armazenada,isto é teria que buscar novamente o local da inserção.

que o pai



resolvendo o problema do nó cheio



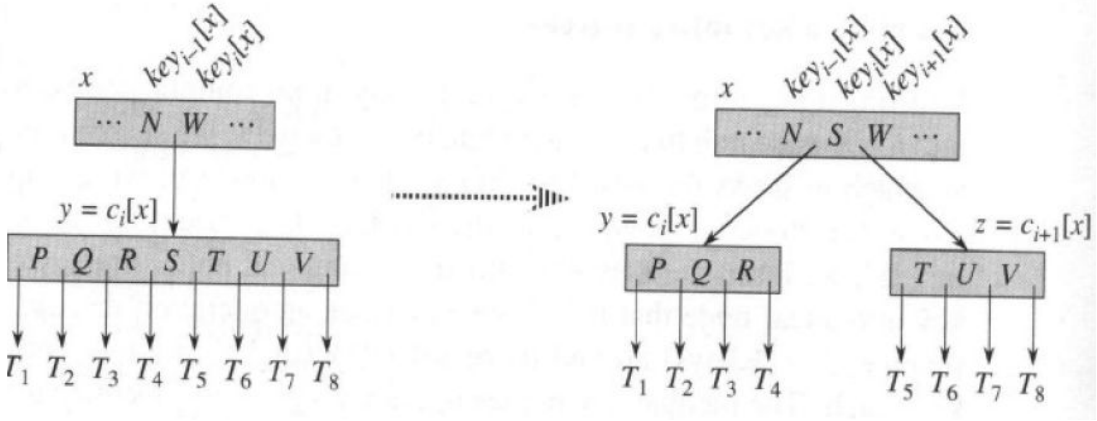
Inserção de uma chave em uma árvore B

Solução: a medida que descemos a árvore procurando pela posição ao qual a nova chave pertence, dividimos cada nó completo que encontramos pelo caminho, inclusive a própria folha.

Inserção de uma chave em uma árvore B

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

x: pai do nó y, não cheio
i: índice i do nó pai, onde será inserida a chave mediana.
y: nó completo a ser dividido, tal que $y=c_i[x]$



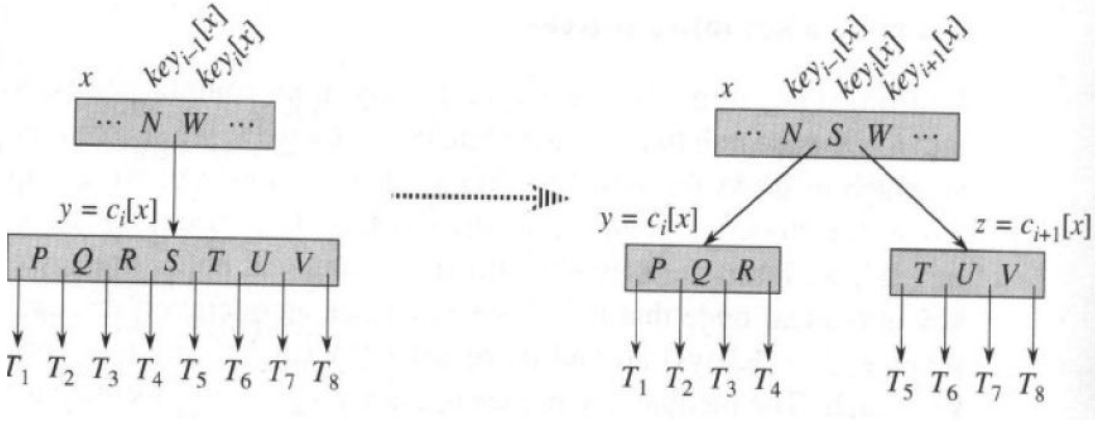
Inserção de uma chave em uma árvore B

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

z é o novo nó com t-1 chaves

copiamos as chaves

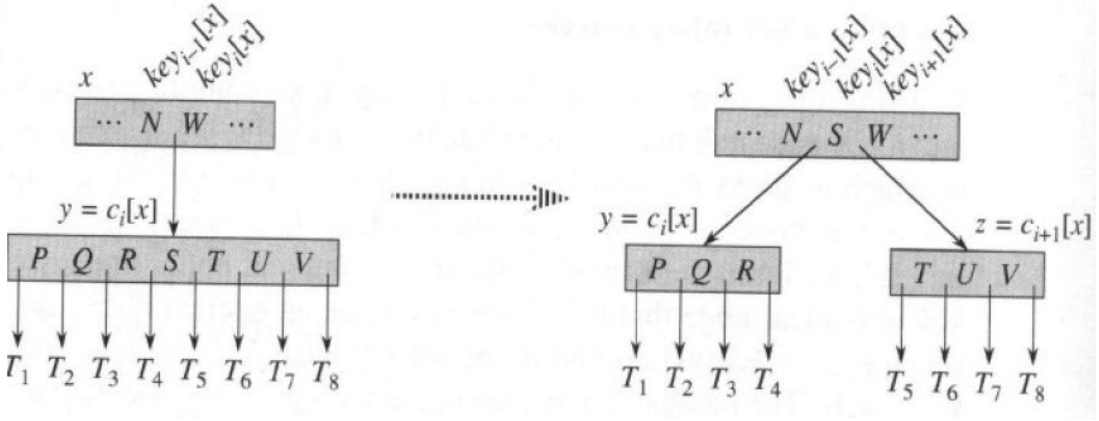
copiamos os ponteiros se y não for folha



Inserção de uma chave em uma árvore B

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

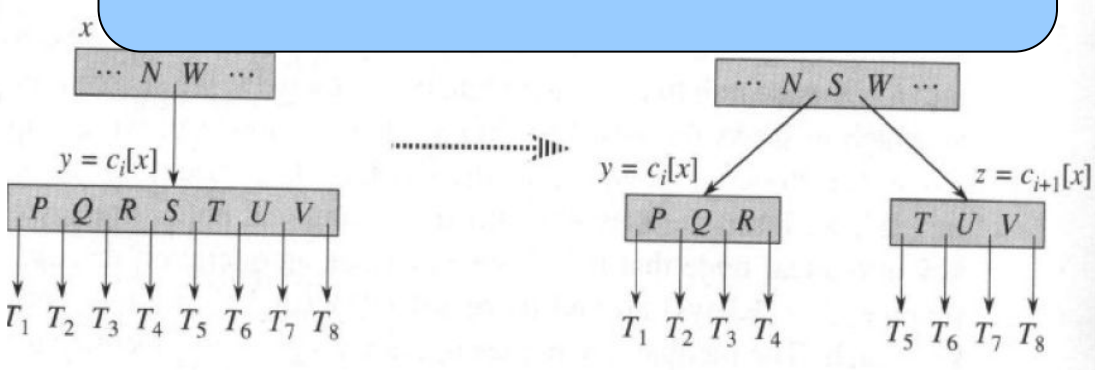
y agora tem t-1 chaves



Inserção de uma chave em uma árvore B

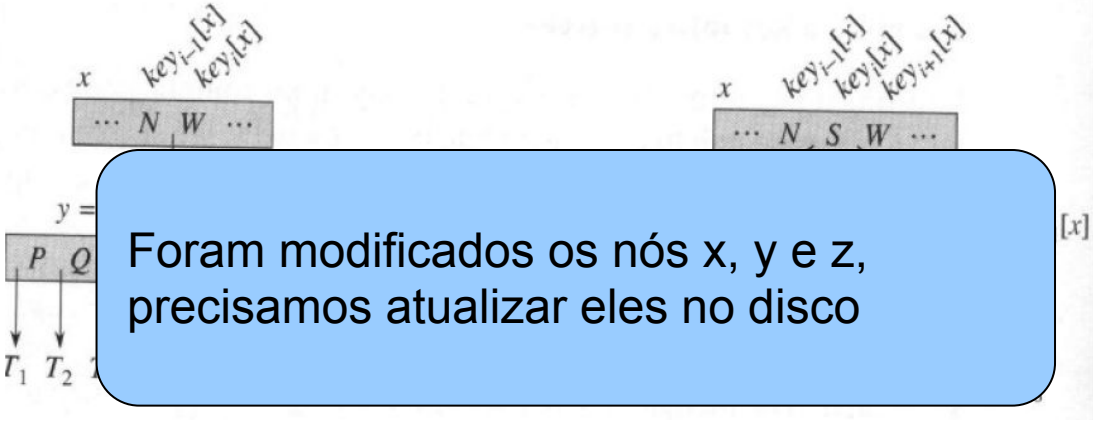
```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

empurramos os ponteiros e as chaves em x para poder inserir a chave mediana.



Inserção de uma chave em uma árvore B

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```



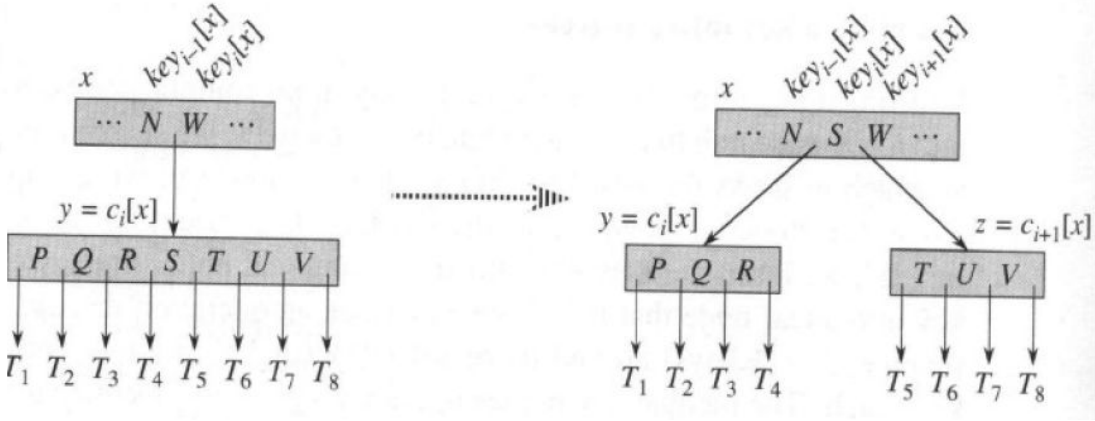
Foram modificados os nós x, y e z, precisamos atualizar eles no disco

Inserção de uma chave em uma árvore B

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

Complexidade:

Acessos ao disco: O(1)

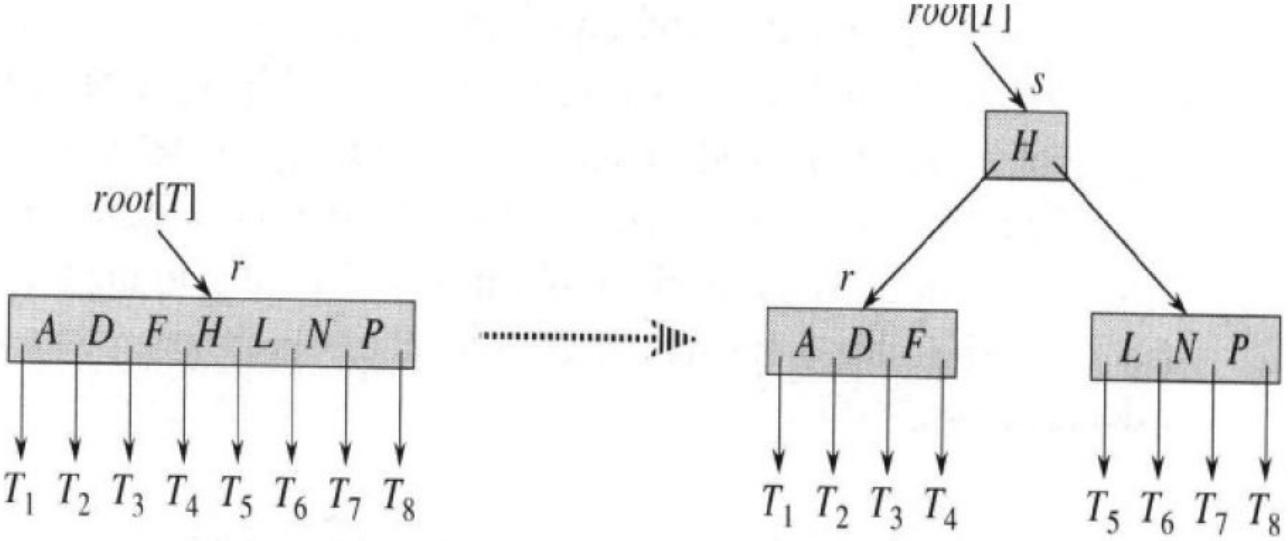


Inserção de uma chave em uma árvore B

```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then s ← ALLOCATE-NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c1[s] ← r
 8          B-TREE-SPLIT-CHILD(s, 1, r)
 9          B-TREE-INSERT-NONFULL(s, k)
10     else B-TREE-INSERT-NONFULL(r, k)
```

A raiz da árvore está completa? Se sim, precisamos criar um novo nó s e ele será a nova raiz.

Inserção de B

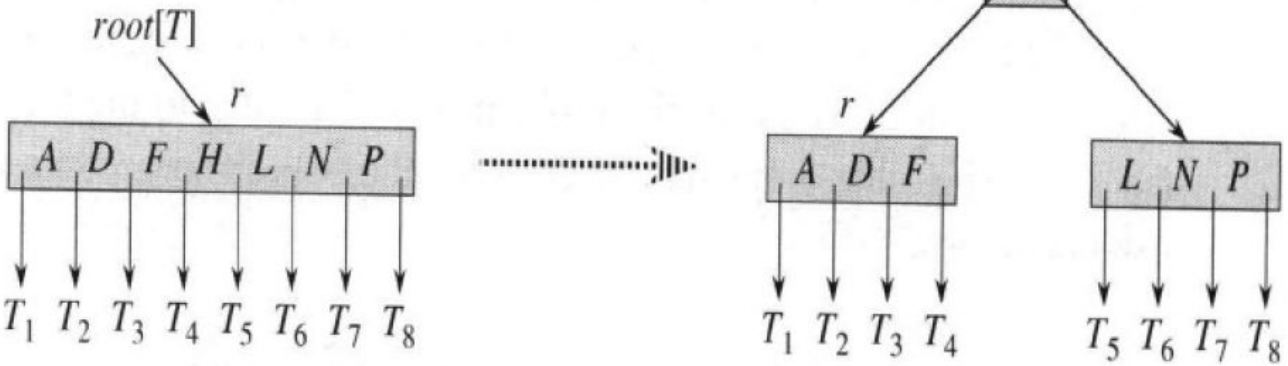


B-TREE-IN

```
1   r ← rc
2   if n[r] = 2t − 1
3       then s ← ALLOCATE-NODE()
4            root[T] ← s
5            leaf[s] ← FALSE
6            n[s] ← 0
7            c1[s] ← r
8            B-TREE-SPLIT-CHILD(s, 1, r)
9            B-TREE-INSERT-NONFULL(s, k)
10      else B-TREE-INSERT-NONFULL(r, k)
```

Note que a árvore aumenta em altura na parte superior

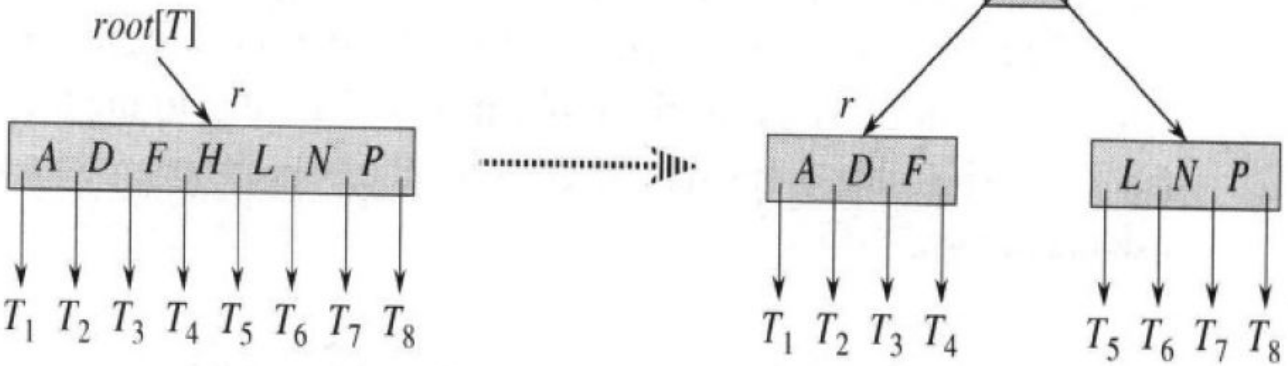
Inserção de B




```
B-TREE-IN
 1   r ← rc
 2   if n[r] = 2t − 1
 3       then s ← ALLOCATE-NODE()
 4            root[T] ← s
 5            leaf[s] ← FALSE
 6            n[s] ← 0
 7            c1[s] ← r
 8            B-TREE-SPLIT-CHILD(s, 1, r)
 9            B-TREE-INSERT-NONFULL(s, k)
10       else B-TREE-INSERT-NONFULL(r, k)
```

Dividimos o nó r e tentamos inserir a chave no nó s

Inserção de B




```
B-TREE-IN
 1   r ← rc
 2   if n[r] = 2t − 1
 3      then s ← ALLOCATE-NODE()
 4           root[T] ← s
 5           leaf[s] ← FALSE
 6           n[s] ← 0
 7           c1[s] ← r
 8           B-TREE-SPLIT-CHILD(s, 1, r)
 9           B-TREE-INSERT-NONFULL(s, k)
10      else B-TREE-INSERT-NONFULL(r, k)
```

Se o nó r não está cheio tentamos inserir a chave no nó r

Inserção de uma chave em uma árvore B

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4             do key_{i+1}[x] ← key_i[x]
 5                i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK-WRITE(x)
 9     else while i ≥ 1 and k < key_i[x]
10             do i ← i − 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                 if k > key_i[x]
16                   then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

Se x é uma folha empurramos as chaves maiores para a direita e inserimos a chave k em x

Inserção de uma chave em uma árvore B

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4                do key_{i+1}[x] ← key_i[x]
 5                   i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK-WRITE(x)
 9     else while i ≥ 1 and k < key_i[x]
10                do i ← i − 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14             then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                  if k > key_i[x]
16                     then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

Se x não é uma folha, determinamos o filho correto para fazer a recursão, mas antes verificamos se o filho está completo (se sim, dividimos ele')

Inserção de uma chave em uma árvore B

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4             do key_{i+1}[x] ← key_i[x]
 5                i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK-WRITE(x)
 9     else while i ≥ 1 and k < key_i[x]
10             do i ← i − 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                 if k > key_i[x]
16                   then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

Comparamos com a chave mediana para saber em que nó devo inserir k.

Inserção de uma chave em uma árvore B

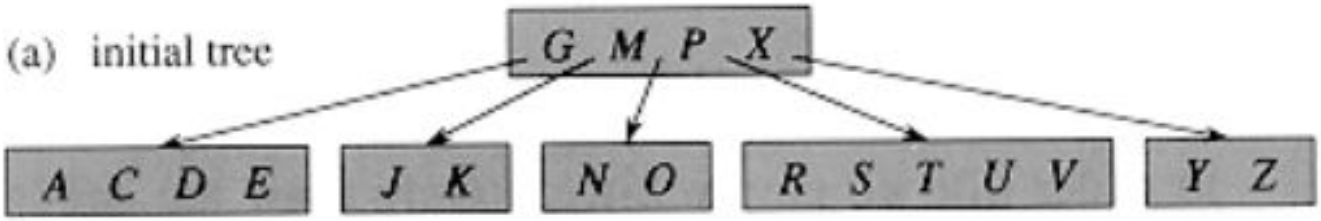
```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4             do key_{i+1}[x] ← key_i[x]
 5                i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK-WRITE(x)
 9     else while i ≥ 1 and k < key_i[x]
10             do i ← i − 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                 if k > key_i[x]
16                   then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

Complexidade:

Acessos ao disco:
$O(h)=O(\log_t n)$

Inserção de uma chave em uma árvore B

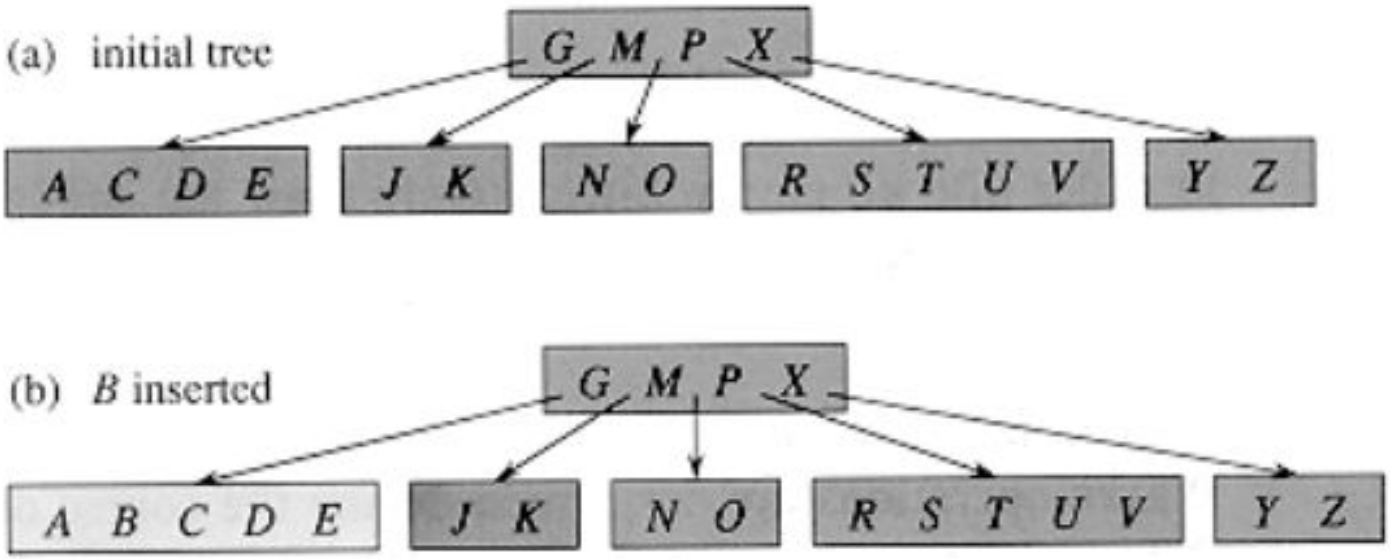
t=3



inserir B

Inserção de uma chave em uma árvore B

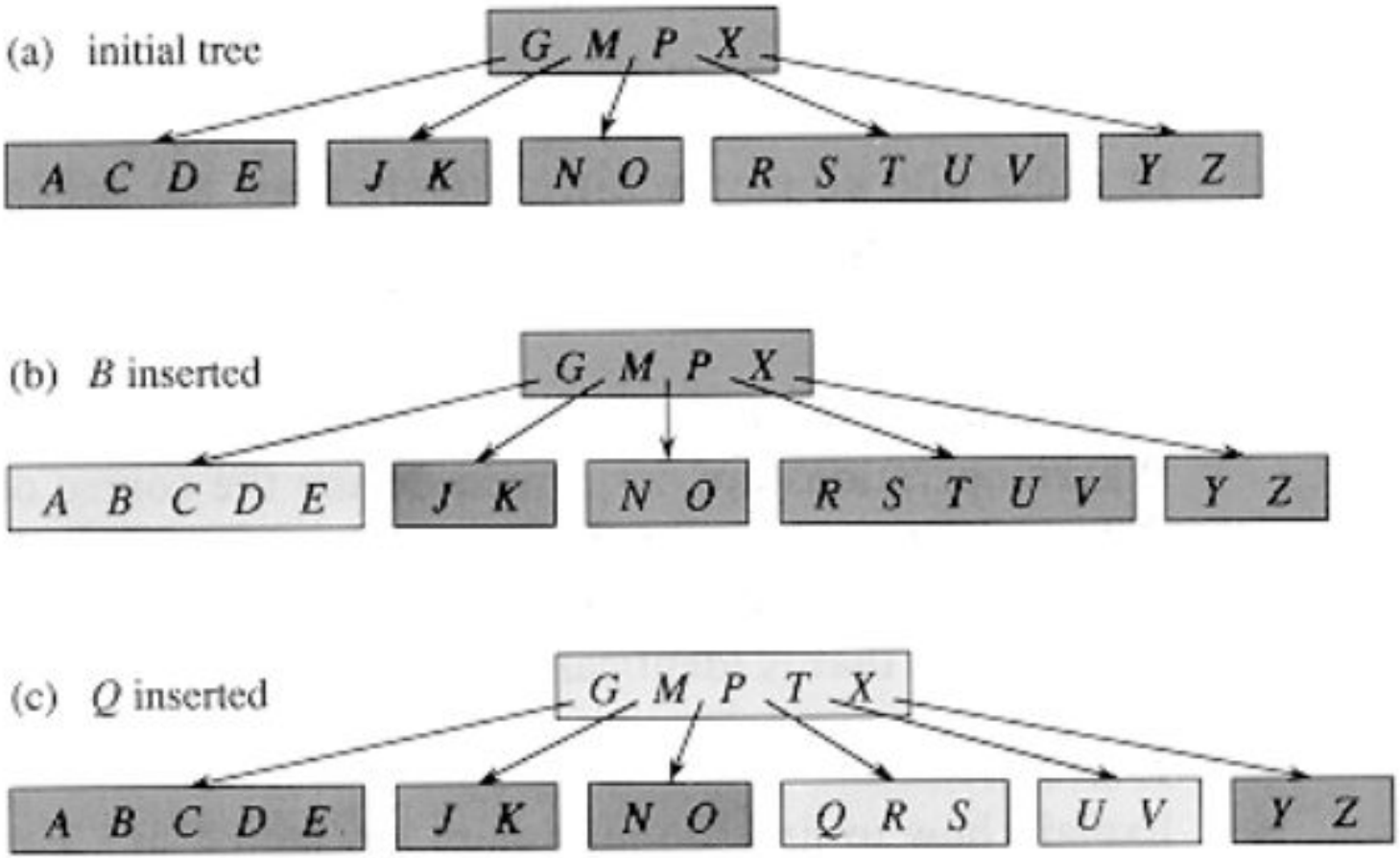
t=3



inserir Q

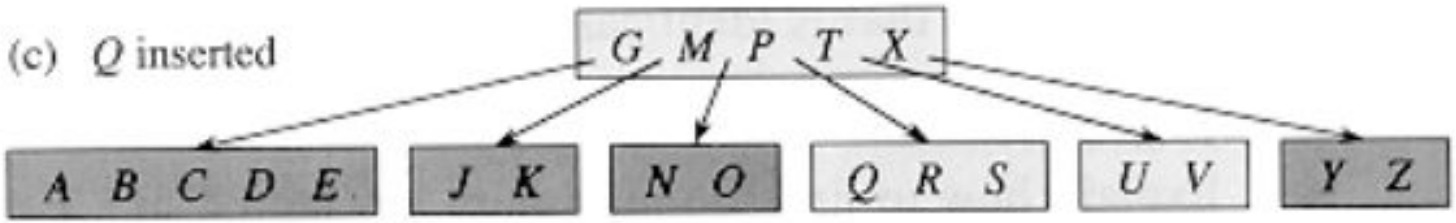
Inserção de uma chave em uma árvore B

t=3



Inserção de uma chave em uma árvore B

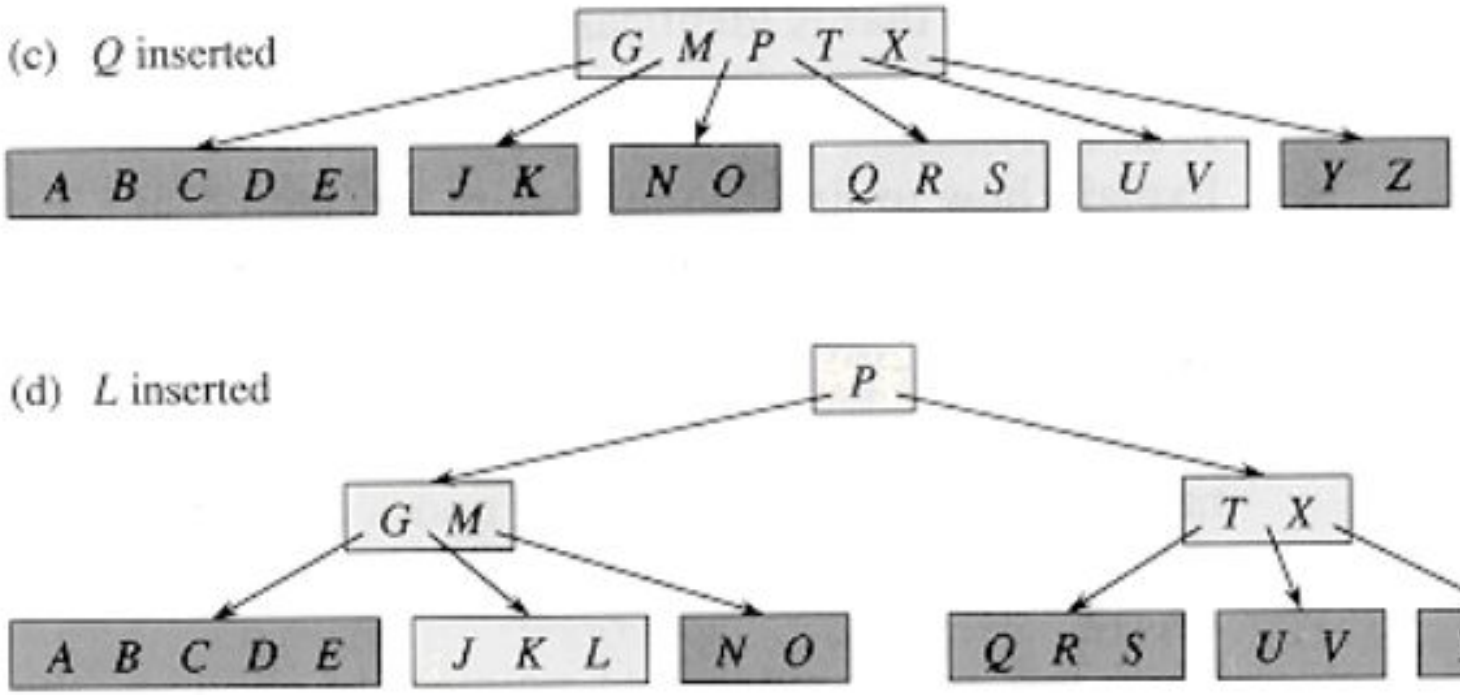
t=3



inserir L

Inserção de uma chave em uma árvore B

t=3

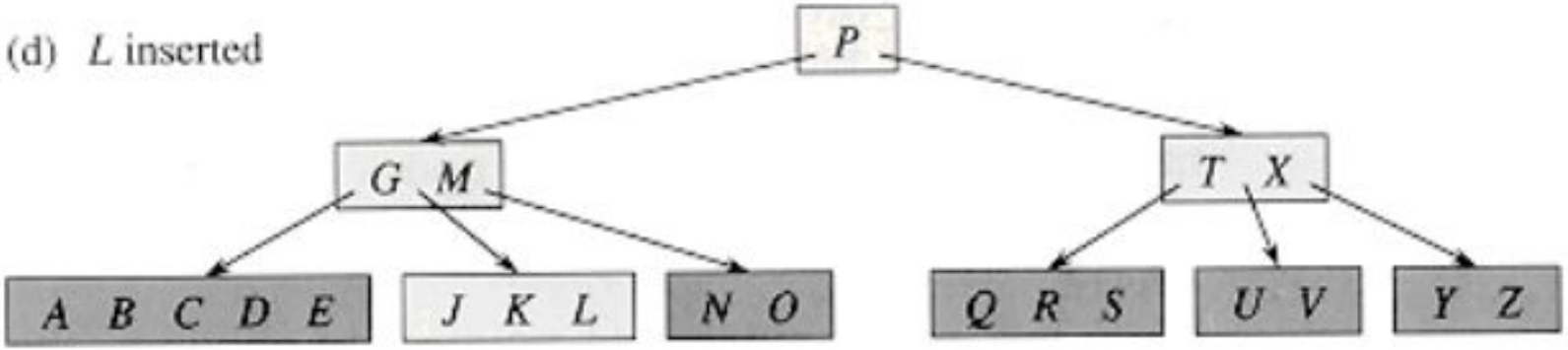


inserir F

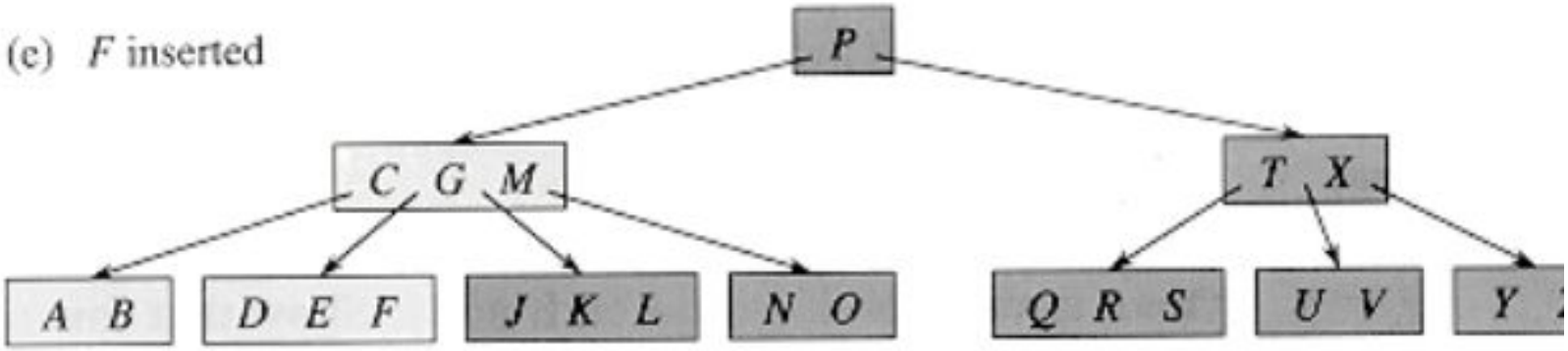
Inserção de uma chave em uma árvore B

t=3

(d) *L* inserted



(e) *F* inserted



Exercícios

- Mostre os resultados da inserção das chaves

200, 25, 60, 100, 10, 30, 40, 70, 80, 90, 95, 96, 5, 7, 13, 15, 22, 24, 26, 27, 32, 33, 41, 45, 230, 340, 720, 66, 67, 68, 71, 73, 85, 88, 110, 120, 130, 118 e 138

nessa ordem em uma árvore B vazia com t=2. Desenhe apenas as configurações da árvore imediatamente antes de ter de dividir algum nó e desenhe a configuração final.

Remoção na árvore B

- A remoção é mais complicada que a inserção pois ela pode ocorrer em qualquer lugar da árvore.
- Devemos ter certeza de que um nó não fique pequeno demais durante a eliminação.

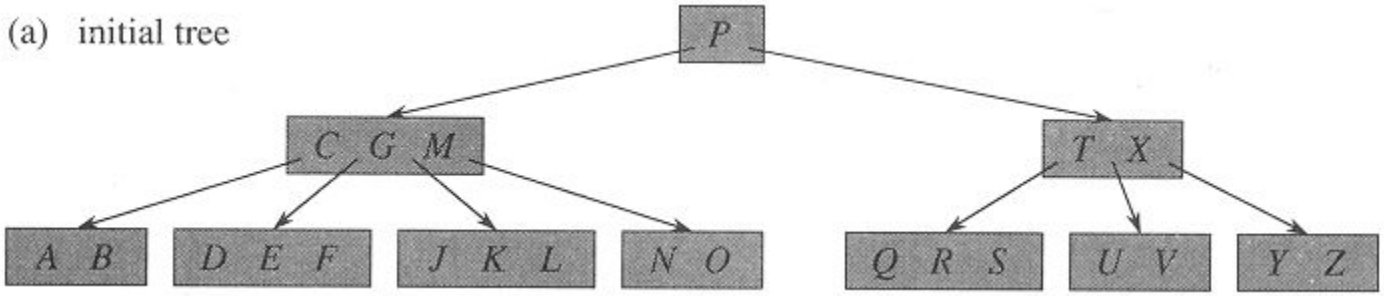
Remoção na árvore B

- O algoritmo BTREE-DELETE(x,k) permite remover a chave k da subárvore com raiz x.
- O algoritmo BTREE-DELETE(x,k) **sempre é chamado com um nó x que tenha pelo menos t chaves**. Essa condição permite eliminar uma chave em uma única passagem descendente (na maioria dos casos).
- O algoritmo eventualmente junta dois nós vizinhos com t-1 chaves formando um nó com 2t-1 chaves.

Remoção na árvore B

- Caso 1: Se a chave k está no nó x e x é uma folha, exclua a chave k de x.

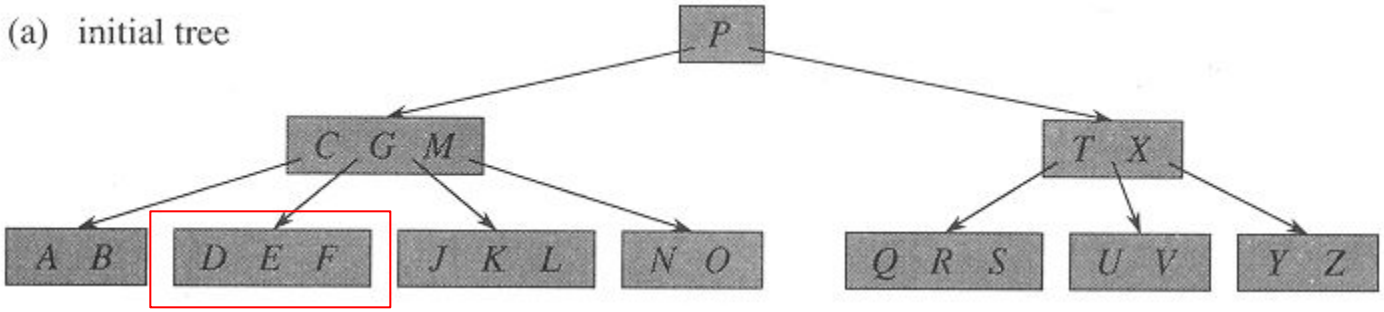
Remoção na árvore B



Em que t=3

Eliminar F

Remoção na árvore B



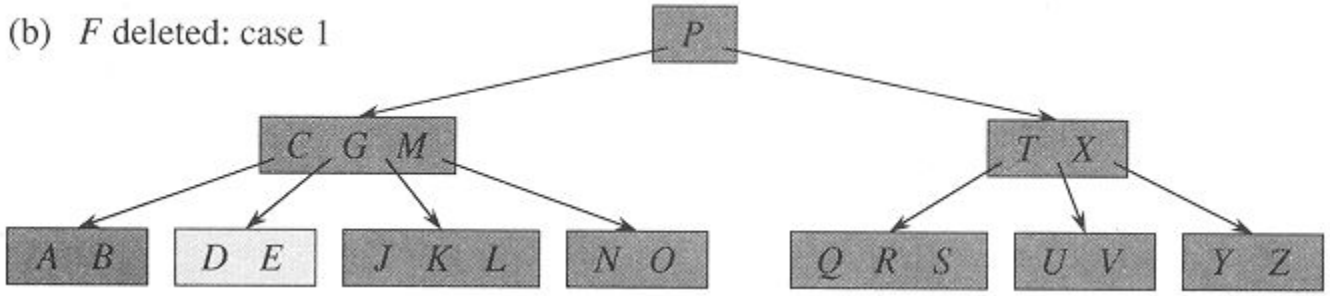
Eliminar F

k = F

x =

– Caso 1: Se a chave k está no nó x e x é uma folha, exclua a chave k de x.

Remoção na árvore B



Remoção na árvore B

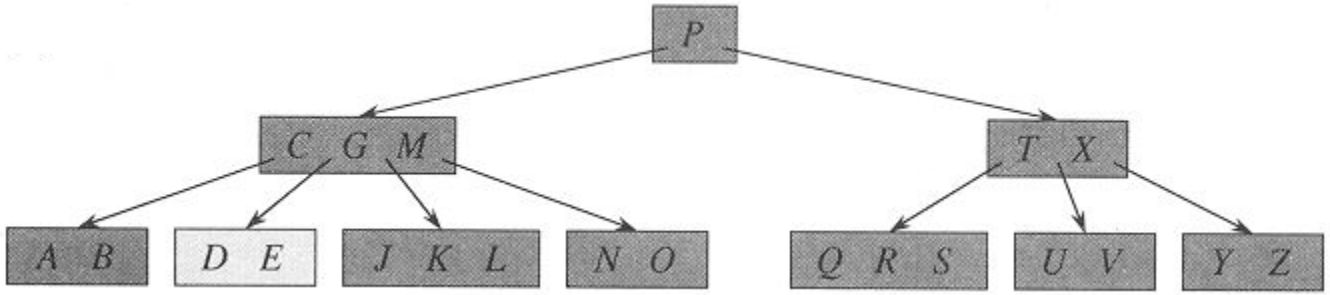
- Caso 2: Se a chave k está no nó x e x é um nó interno:

2.1 Se o filho y que precede k do nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz y. Elimine recursivamente k', e substitua k por k' em x.

2.2 Simetricamente, se o filho z que segue k do nó x tem pelo menos t chaves, então encontre o sucessor k' de k na subárvore com raiz z. Elimine recursivamente k', e substitua k por k' em x.

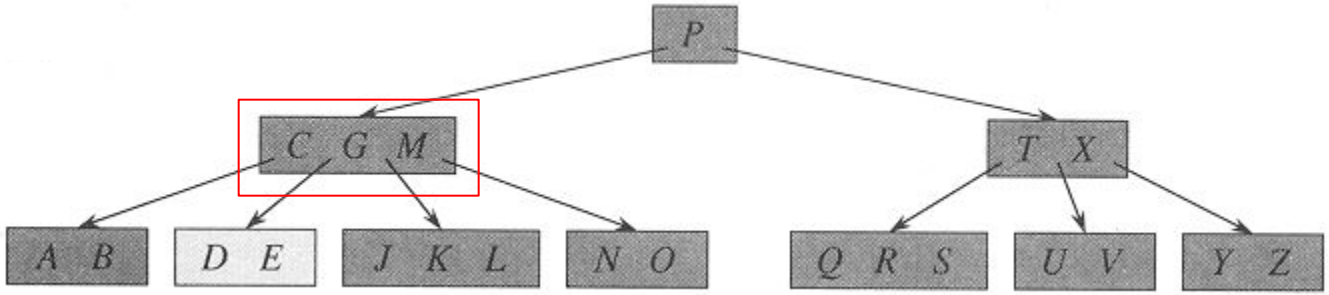
2.3 Caso contrario, se ambos y e z possuem apenas t-1 chaves, faça a juncão de k e todas as chaves de z em y. O nó x perde tanto a chave k como o ponteiro para z, e y agora contem 2t - 1 chaves. Em seguida, libere z e elimine recursivamente k de y.

Remoção na árvore B



Eliminar M

Remoção na árvore B

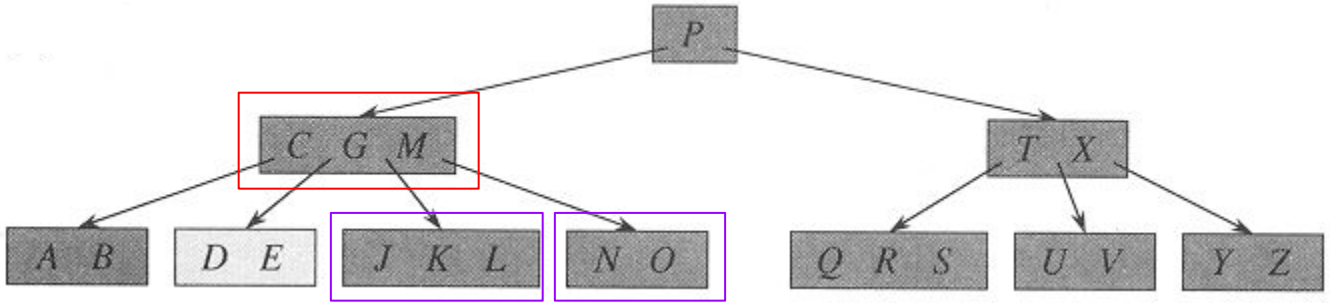


Eliminar M

k = M

x =

Remoção na árvore B



Eliminar M

k = M

x =

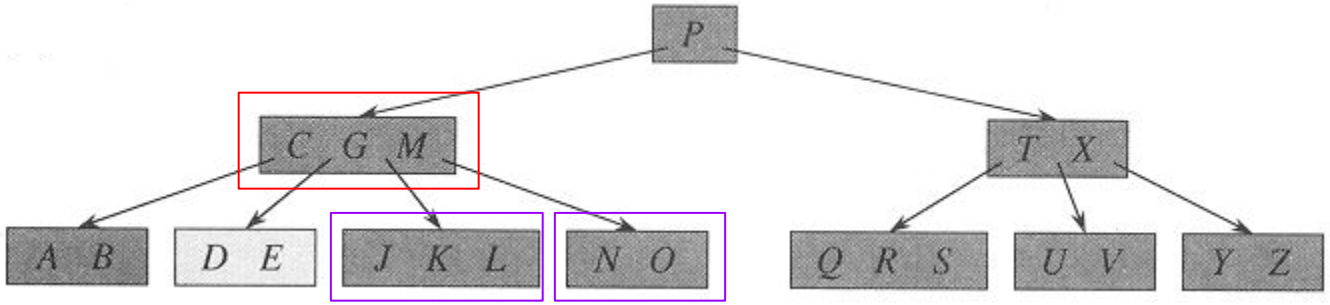
Caso 2: Se a chave k está no nó x e x é um nó interno:

2.1 Se o filho y que precede k do nó x tem pelo menos t chaves

2.2 Se o filho z que segue k do nó x tem pelo menos t chaves.

2.3 Se ambos y e z possuem apenas t-1 chaves.

Remoção na árvore B



Eliminar M

k = M y=JKL z=NO

x =

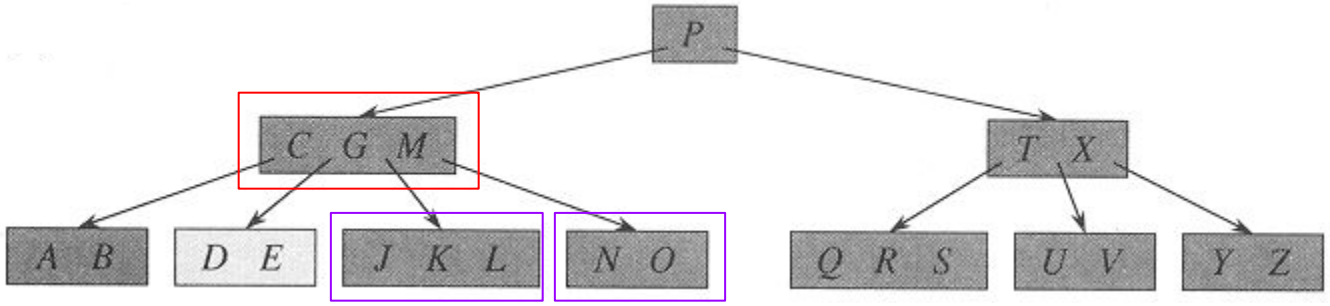
Caso 2: Se a chave k está no nó x e x é um nó interno:

2.1 Se o filho y que precede k do nó x tem pelo menos t chaves…

2.2 Se o filho z que segue k do nó x tem pelo menos t chaves…

2.3 Se ambos y e z possuem apenas t-1 chaves.

Remoção na árvore B



Eliminar M

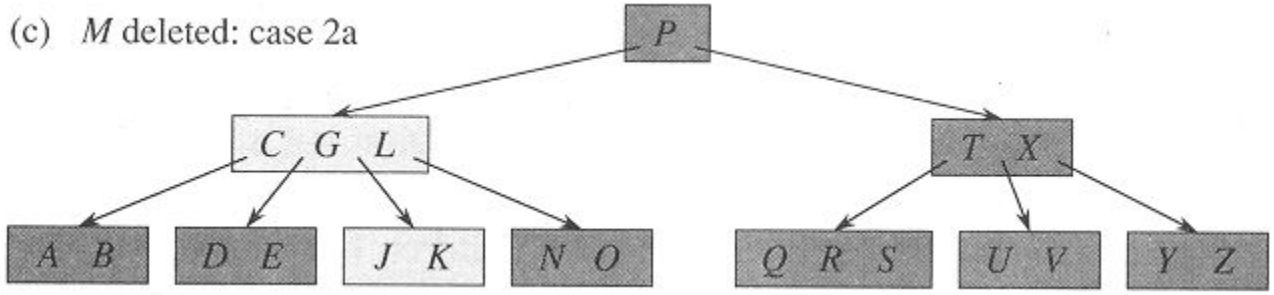
k = M y=JKL z=NO

x =

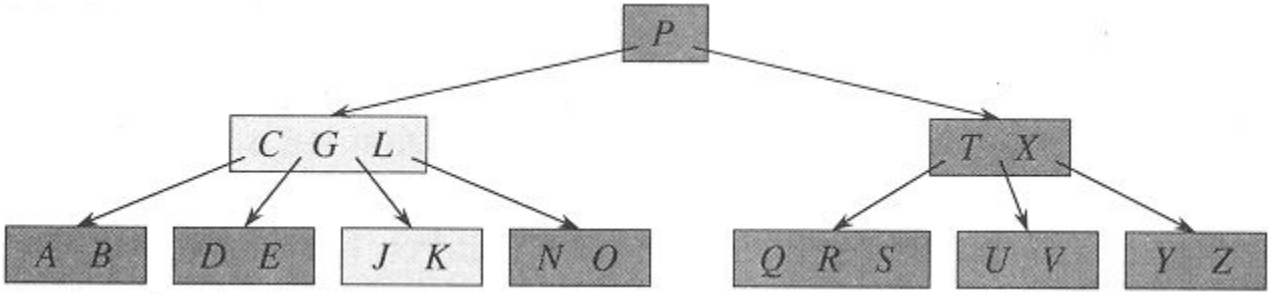
2.1 Se o filho y que precede k do nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz y. Elimine recursivamente k', e substitua k por k' em x.

k'=L, chamamos BTREE-DELETE com o nó JKL e a chave L, estamos no caso 1.

Remoção na árvore B

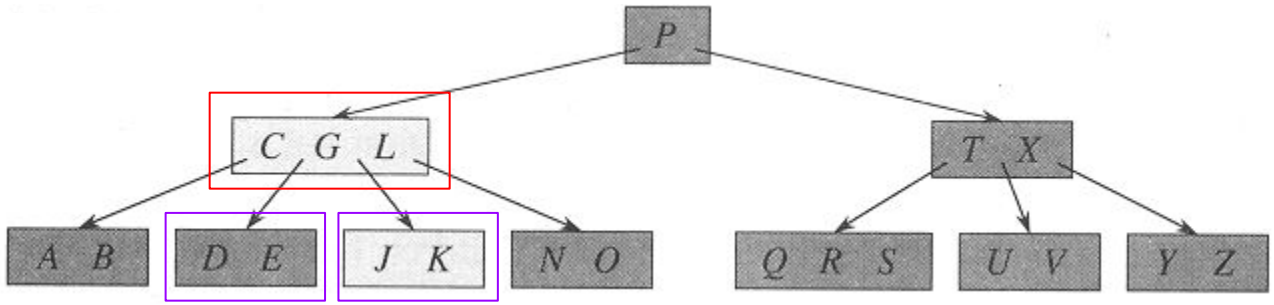


Remoção na árvore B



Eliminar G

Remoção na árvore B



Eliminar G

k = G y=DE z=JK

x =

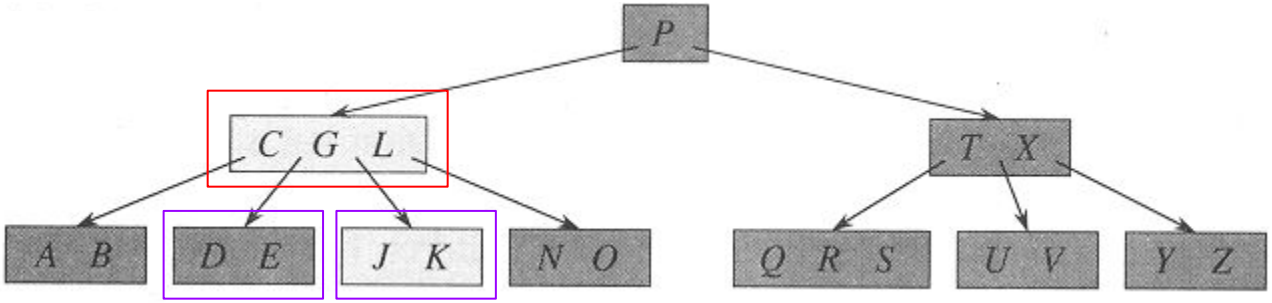
Caso 2: Se a chave k está no nó x e x é um nó interno:

2.1 Se o filho y que precede k no nó x tem pelo menos t chaves…

2.2 Se o filho z que segue k no nó x tem pelo menos t chaves...

2.3 Se ambos y e z possuem apenas t-1 chaves...

Remoção na árvore B



Eliminar G

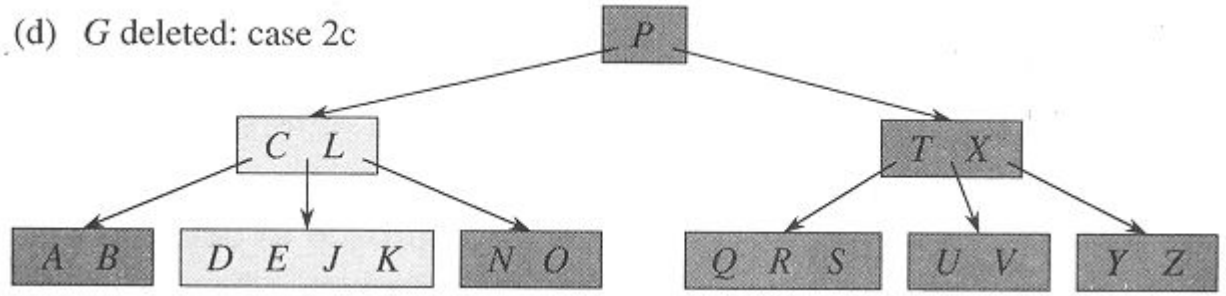
k = G y=DE z=JK

x =

2.3 Se ambos y e z possuem apenas t-1 chaves, faça a juncão de k e todas as chaves de z em y. O nó x perde tanto a chave k como o ponteiro para z, e y agora contem 2t - 1 chaves. Em seguida, libere z e elimine recursivamente k de y.

y= DEGJK e x=CL, chamamos BTREE-DELETE com o nó DEGJK e chave G, estamos no caso 1.

Remoção na árvore B



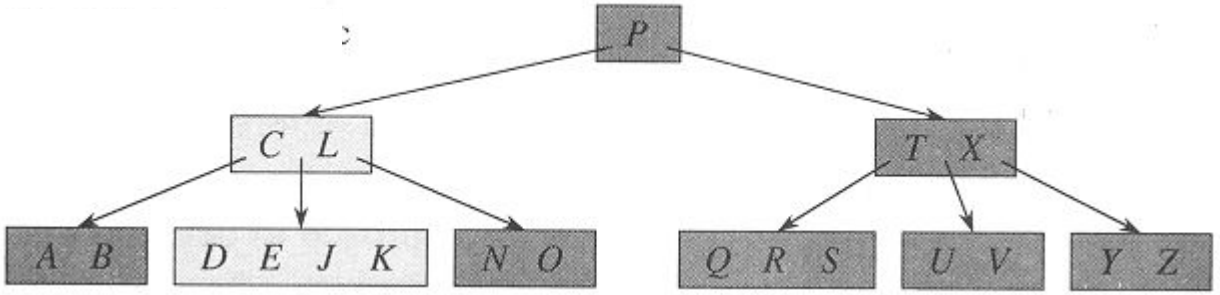
Remoção na árvore B

- Caso 3: Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k. Se $c_i[x]$ tem apenas t-1 chaves, execute o passo 3.1 ou 3.2 conforme necessario para garantir que o algoritmo desça para um nó contendo pelo menos t chaves. Em seguida, faça uma recursão com o filho apropriado de x.

3.1 Se $c_i[x]$ tem um irmão imediato com pelo menos t chaves, dê para $c_i[x]$ uma chave extra movendo uma chave de x para $c_i[x]$, movendo uma chave do irmão imediato de $c_i[x]$ a esquerda ou a direita para dentro de x, e movendo o ponteiro do filho apropriado do irmão para o nó $c_i[x]$.

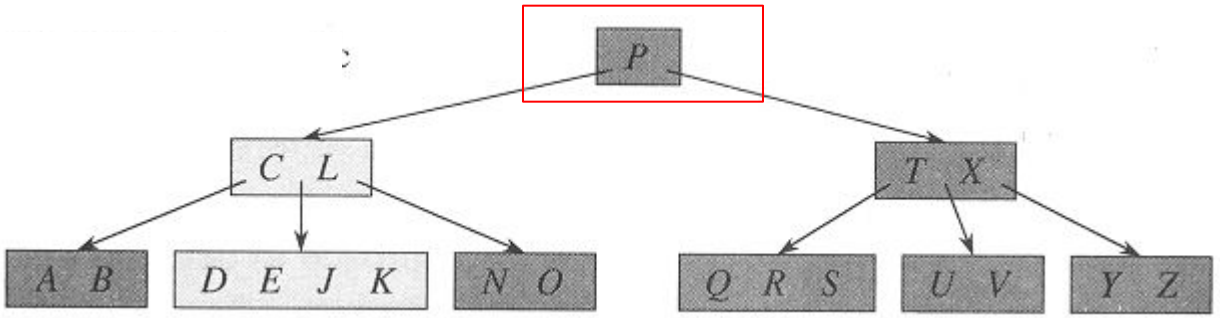
3.2 Se ambos os irmãos imediatos de $c_i[x]$ contêm t- 1 chaves, faça a junção de $c_i[x]$ com um de seus irmãos e mova uma chave de x para o novo nó fundido (que se tornara a chave mediana para aquele nó).

Remoção na árvore B



Eliminar D

Remoção na árvore B



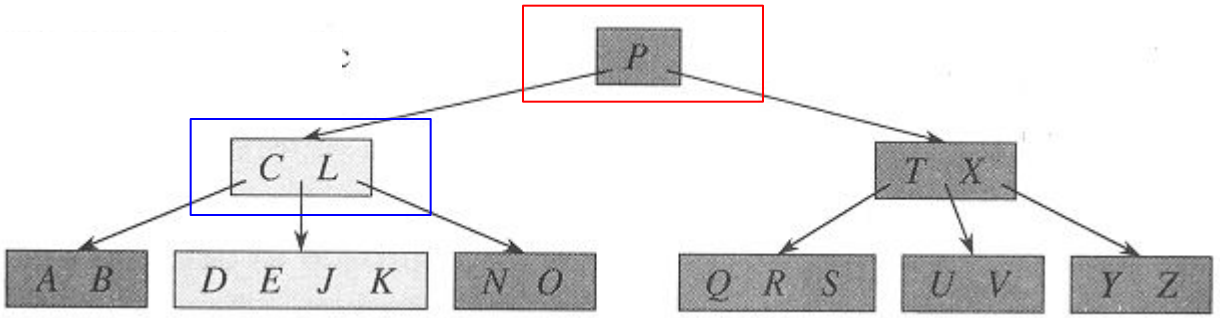
Eliminar D, a recursão não pode descer até CL porque ele apenas tem t-1 chaves

k = D

x =

Caso 3: Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k

Remoção na árvore B



Eliminar D

k = D

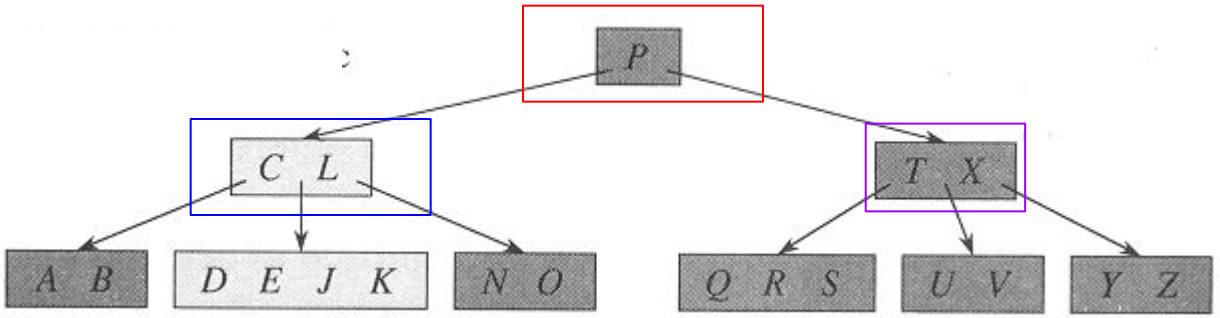
x = $c_i[x]$

Caso 3: Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k...

3.1 Se $c_i[x]$ tem um irmão imediato com pelo menos t chaves...

3.2 Se ambos os irmãos imediatos de $c_i[x]$ contêm t- 1 chaves,

Remoção na árvore B



Eliminar D

k = D

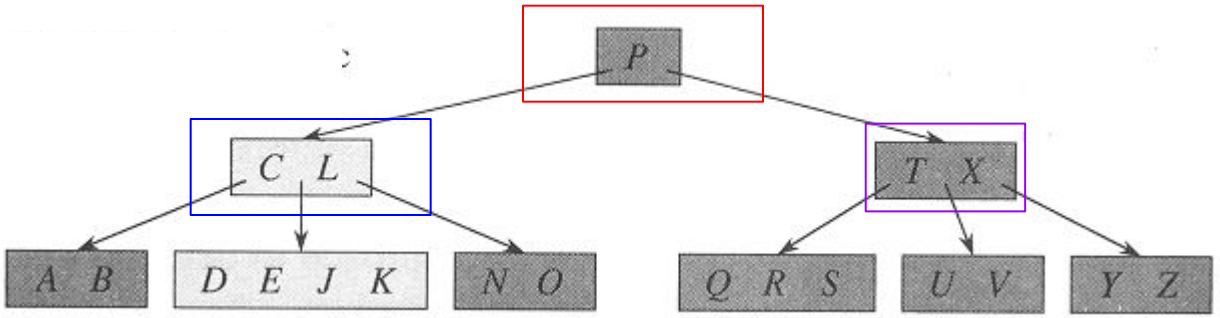
x = $c_i[x]$

Caso 3: Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k...

3.1 Se $c_i[x]$ tem um irmão imediato com pelo menos t chaves...

3.2 Se ambos os irmãos imediatos de $c_i[x]$ contêm t- 1 chaves,...

Remoção na árvore B



Eliminar D

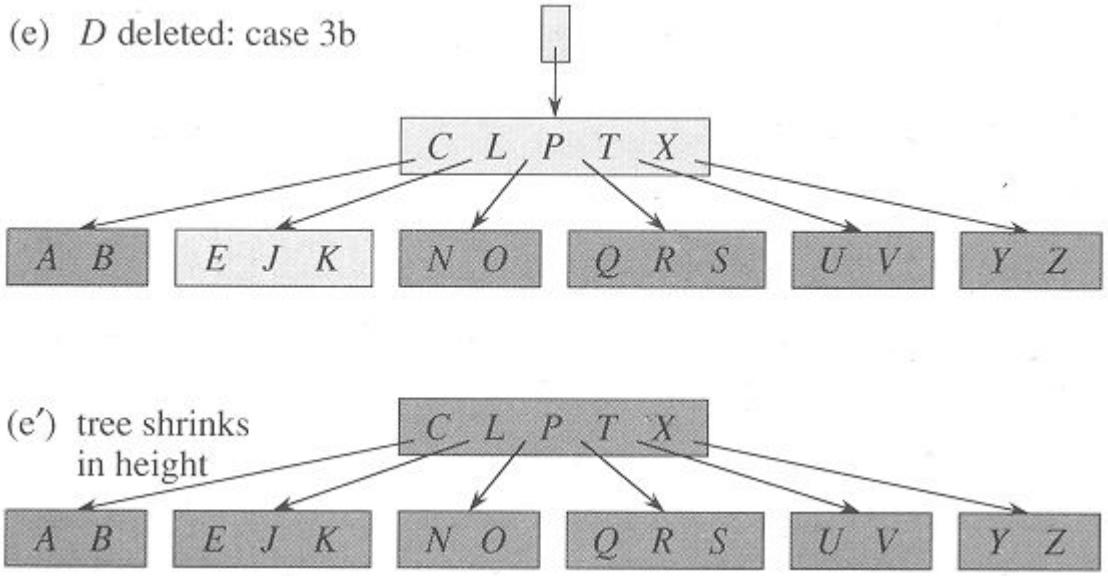
k = D

x = $c_i[x]$

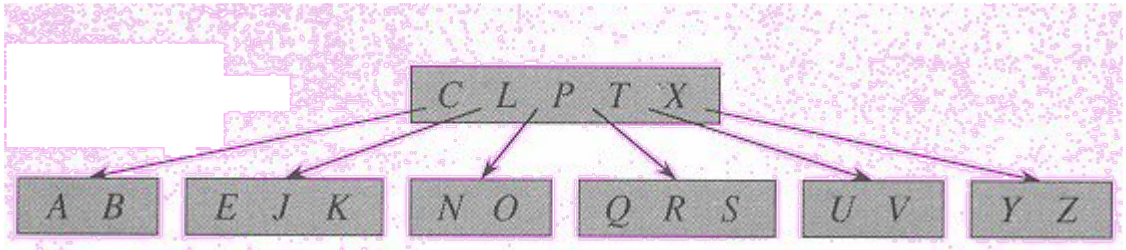
3.2 Se ambos os irmãos imediatos de $c_i[x]$ contêm t- 1 chaves, faça a junção de $c_i[x]$ com um de seus irmãos e mova uma chave de x para o novo nó fundido (que se tornara a chave mediana para aquele nó).

Obtemos um novo nó CLPTX. Chamamos BTREE-DELETE com o nó CLPTX e a chave D, descemos recursivamente na árvore e chegamos novamente no caso 1 eliminando D do nó DEJK.

Remoção na árvore B

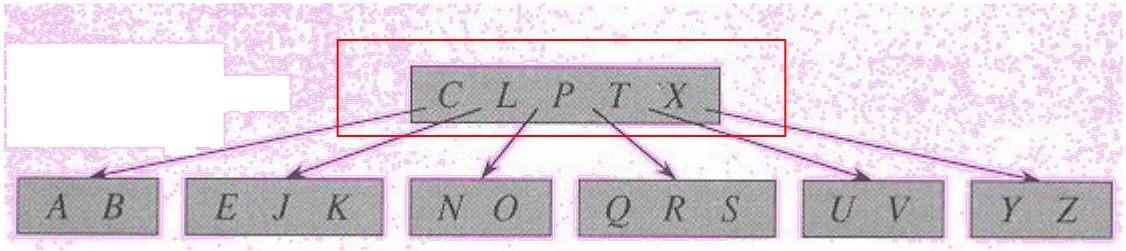


Remoção na árvore B



Eliminar B

Remoção na árvore B



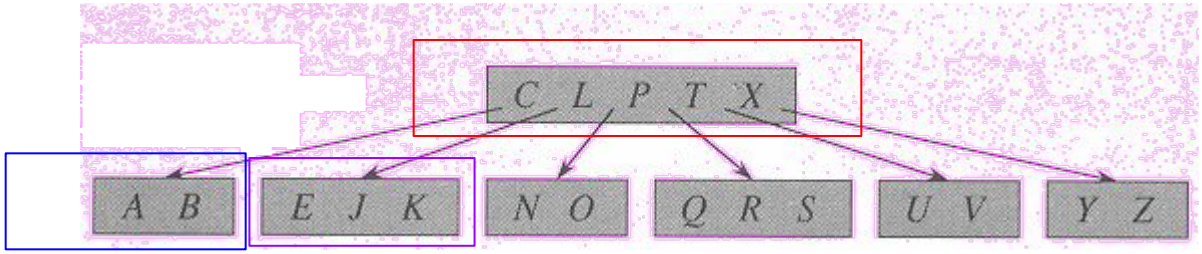
Eliminar B, a recursão não pode descer até AB porque ele apenas tem t-1 chaves

k = B

x =

Caso 3: Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k

Remoção na árvore B



Eliminar B, a recursão não pode descer até AB porque ele apenas tem t-1 chaves

k = B

x =

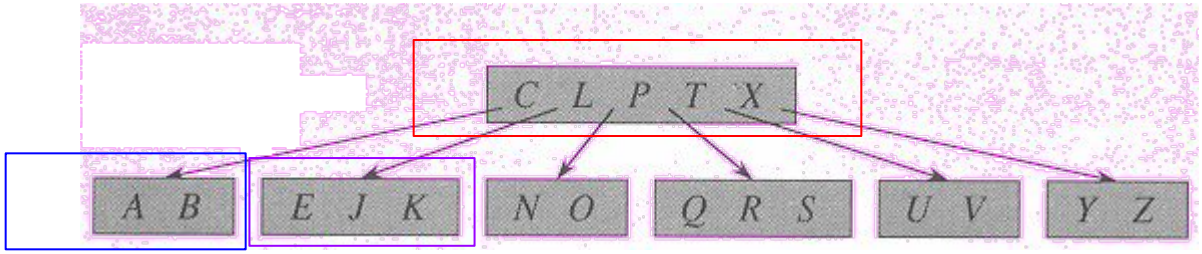
$c_i[x]$=

Caso 3: Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k ...

3.1 Se $c_i[x]$ tem um irmão imediato com pelo menos t chaves...

3.2 Se ambos os irmãos imediatos de $c_i[x]$ contêm t- 1 chaves...

Remoção na árvore B



Eliminar B, a recursão não pode descer até AB porque ele apenas tem t-1 chaves

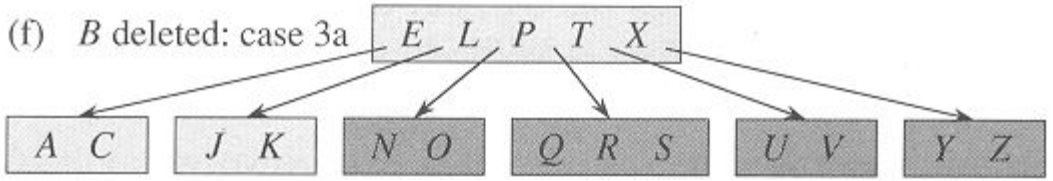
k = B

x = $c_i[x]$=

3.1 Se $c_i[x]$ tem um irmão imediato com pelo menos t chaves, dê para $c_i[x]$ uma chave extra movendo uma chave de x para $c_i[x]$, movendo uma chave do irmão imediato de $c_i[x]$ à esquerda ou à direita para dentro de x, e movendo o ponteiro do filho apropriado do irmão para o nó $c_i[x]$.

C é movido para o nó $c_i[x]$ e E é movido para preencher a posição de C. Agora o nó $c_i[x]$ =ABC, seu irmão JK e x=ELPTX. O antigo ponteiro da esquerda de E agora é o ponteiro à direita de C. Chamamos BTREE-DELETE com o nó ABC e a chave B, chegamos novamente no caso 1 eliminando B.

Remoção na árvore B



Remoção na árvore B

- A raiz x pode se tornar um nó sem nenhuma chave (quando aplicarmos os casos 2.3 ou 3.2). Nesse caso x será eliminado e o único filho de x, $c_i[x]$, se tornará a nova raiz da árvore, diminuindo a altura da árvore em uma unidade.

Exercícios

- Desenhe a nova árvore B depois de remover cada uma das seguintes chaves

200, 80, 90, 26, 27, 32, 33 ,120, 130, 138, 10.

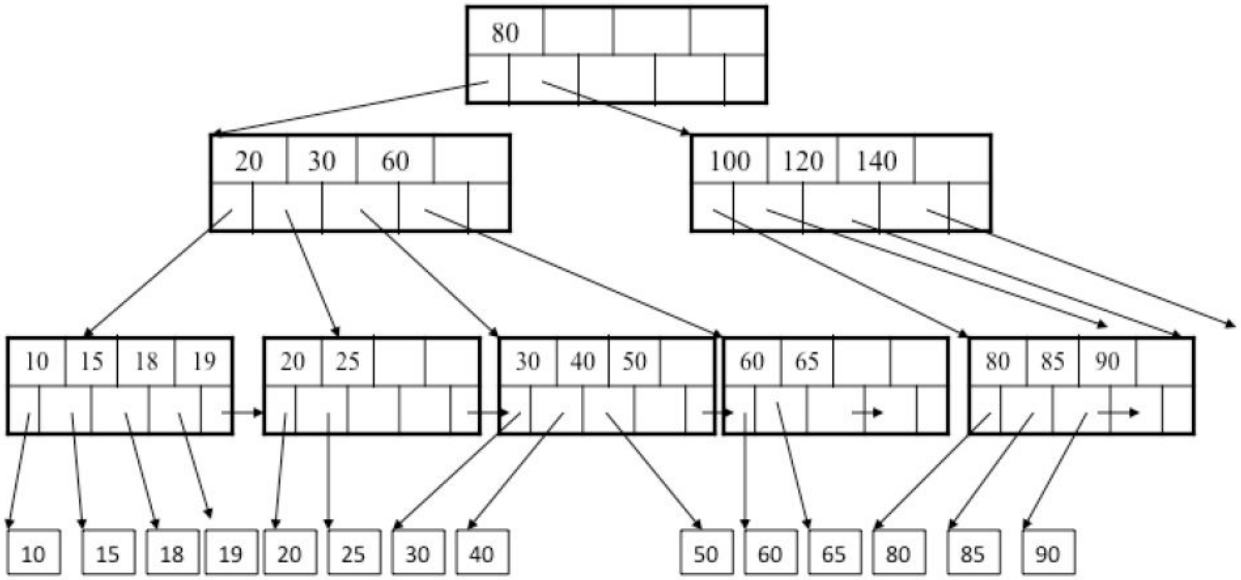
Indique para cada remoção o caso aplicado.

Variantes

- Árvores B*
  - Nos internos (sem incluir a raiz) precisam ficar ⅔ cheios ao invés de ½ cheios.
- Árvores B+
  - Mantém cópias das chaves nos nós internos e nas folhas armazena as chaves e os registros.

Árvore B+

- Mantém cópias das chaves nos nós internos e nas folhas armazena as chaves e os registros (conectadas da esquerda para a direita, permitindo acesso sequencial)
- Cabem mais chaves nos nós internos o que implica em uma altura menor.



Nós internos (de índices)

Nós folhas (de dados)