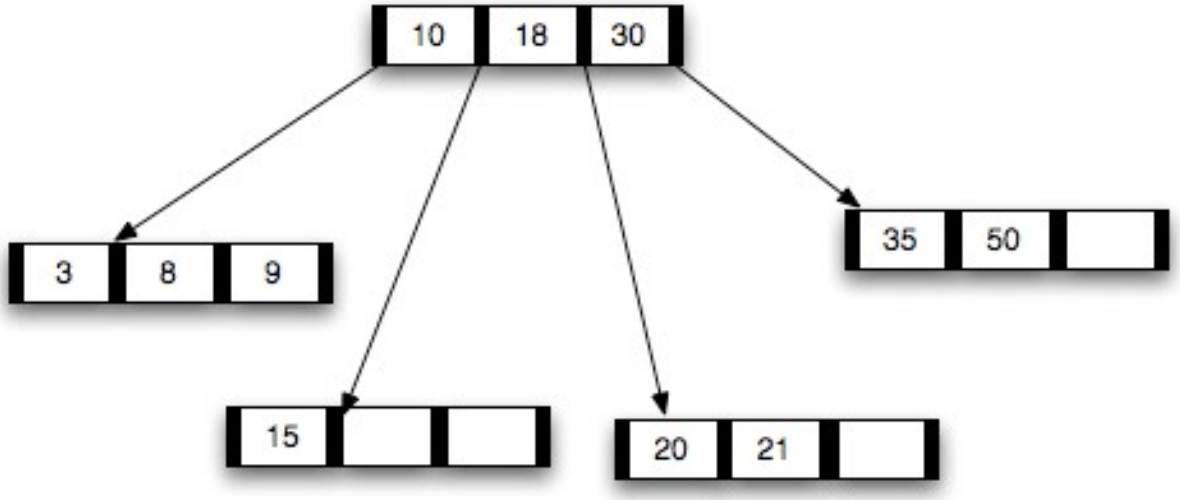
# ACH2024

# Árvores B
# (continuação - inserção)

# Nas aulas passadas...

# ÁRVORES B !!!

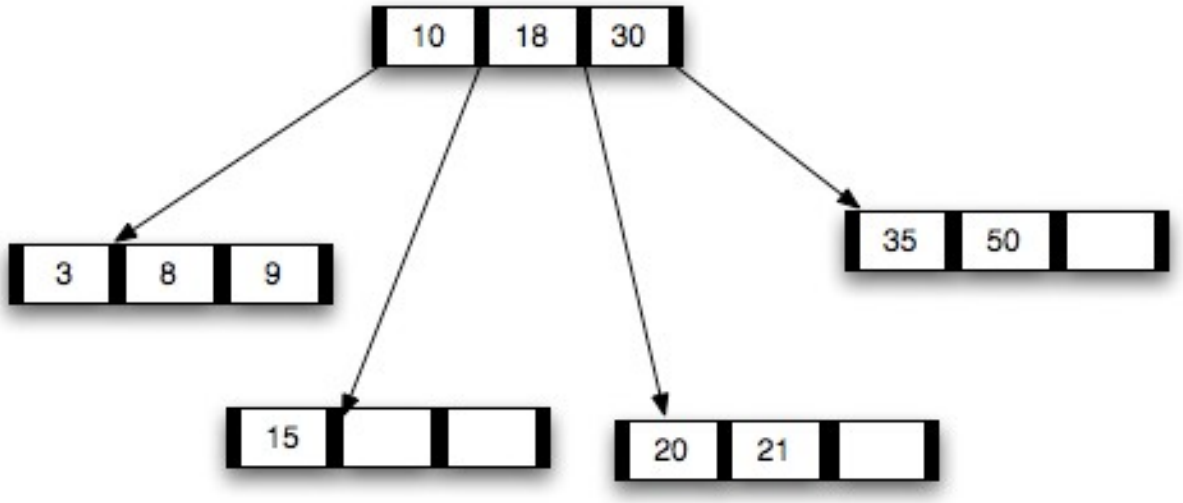
- Registros organizados pela árvore, assim como na árvore binária de busca
- Logo, se a chave é única, não há repetição de valores
- Abaixo é representada só a chave para  simplificar a figura, mas na verdade deve conter, para cada chave, o resto do registro (demais dados daquele item) ou um ponteiro para o registro  (ki, pi)



# Árvore B - 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.

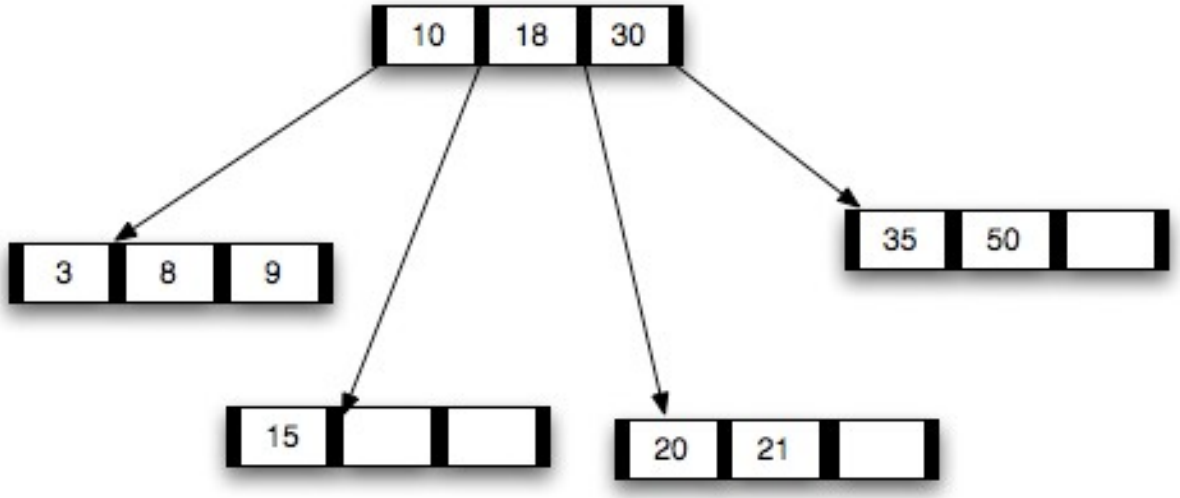


# Árvore B - Definição

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

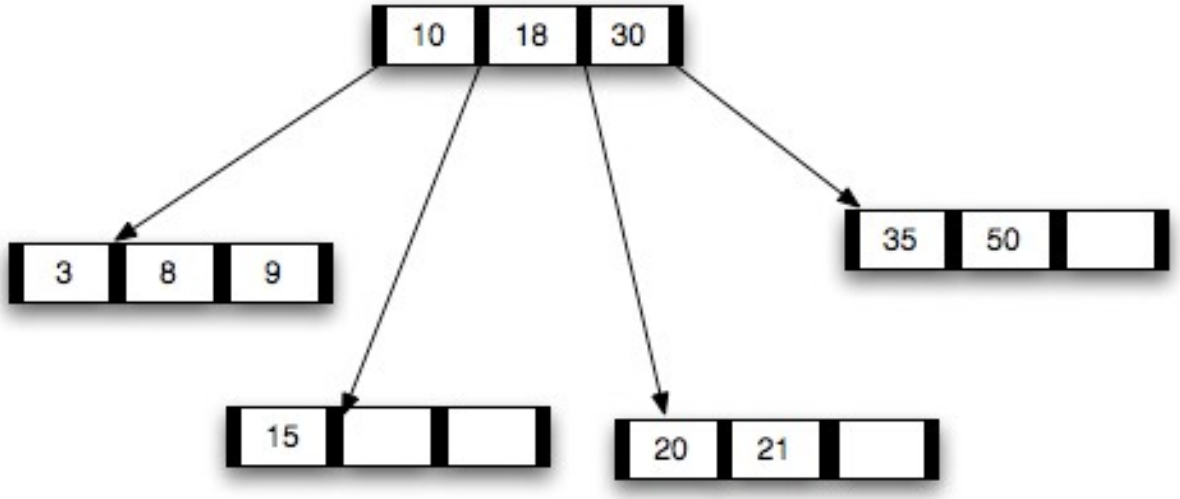
$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}$$

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



# Árvore B - 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



## Busca na árvore B

- B-Tree-Search$(x, k)$: tem como parâmetros um ponteiro para a raiz do nó $x$ 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$(root[T], k)$.

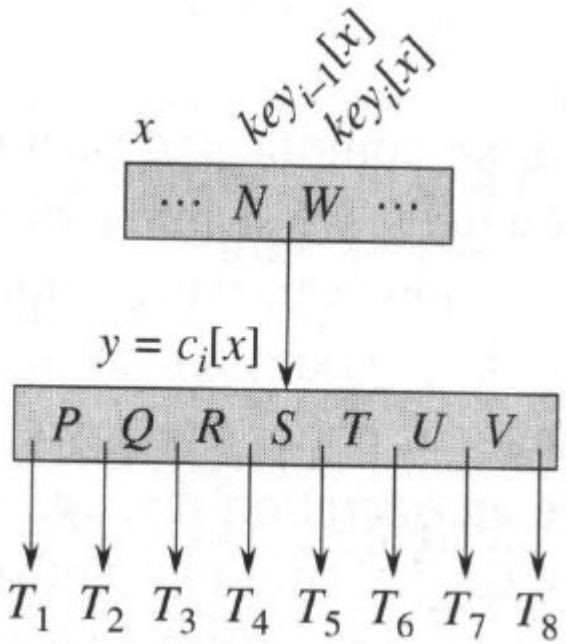
```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

# Aula de hoje

# Inserção em árvore B

As inserções ocorrem sempre nas folhas

Como inserir o registro com chave “O” nesta árvore?

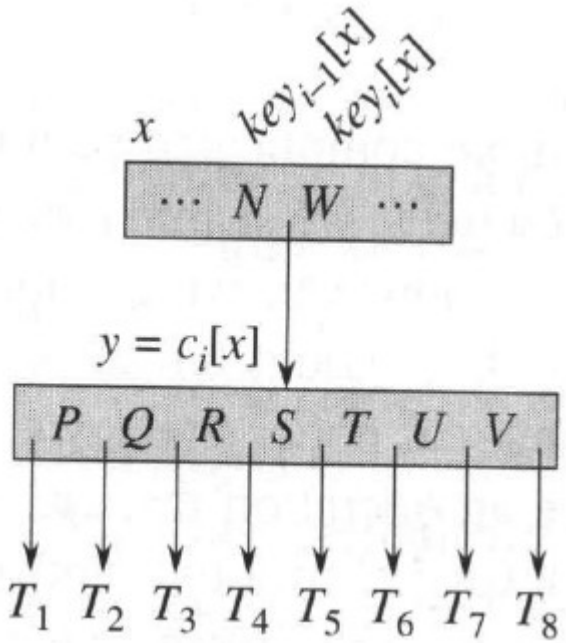


# Inserção em árvore B

As inserções ocorrem sempre nas folhas

Como inserir o registro com chave “O” nesta árvore?

Antes, precisa resolver o problema do nó cheio...

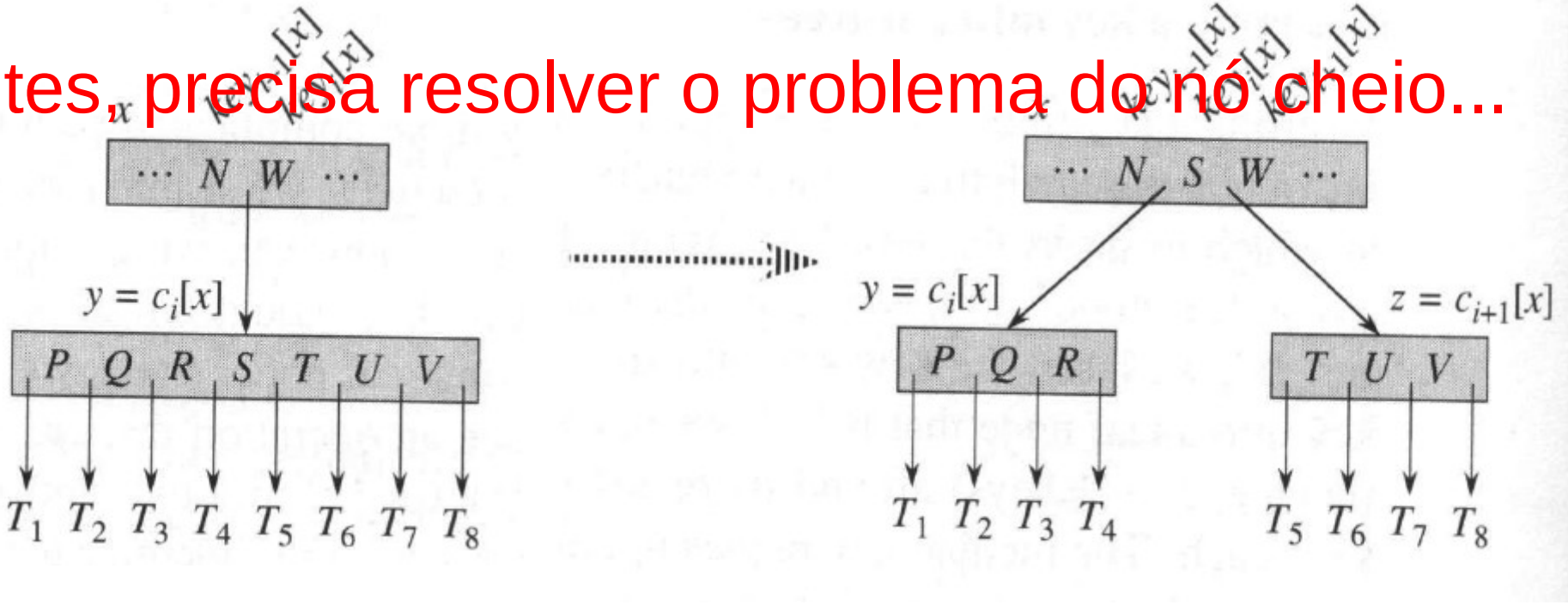


# Inserção em árvore B

As inserções ocorrem sempre nas folhas

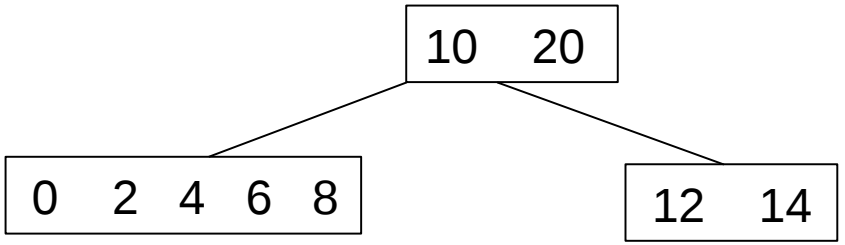
Como inserir o registro com chave “O” nesta árvore?

Antes, precisa resolver o problema do nó cheio...



# Inserção em árvore B

- O que aconteceria se o nó pai também estiver cheio?
  - Teria que dividir também o pai, o que poderia causar subdivisões em cascata
  - Todo nó teria que guardar ponteiros para o nó pai
    - Aumentar o conteúdo de um nó → diminui o fator de blocagem → diminui o fator de ramificação → aumenta a altura da árvore
  - Além disso (e principalmente), as subdivisões podem alterar em qual nó o registro deve ser armazenado



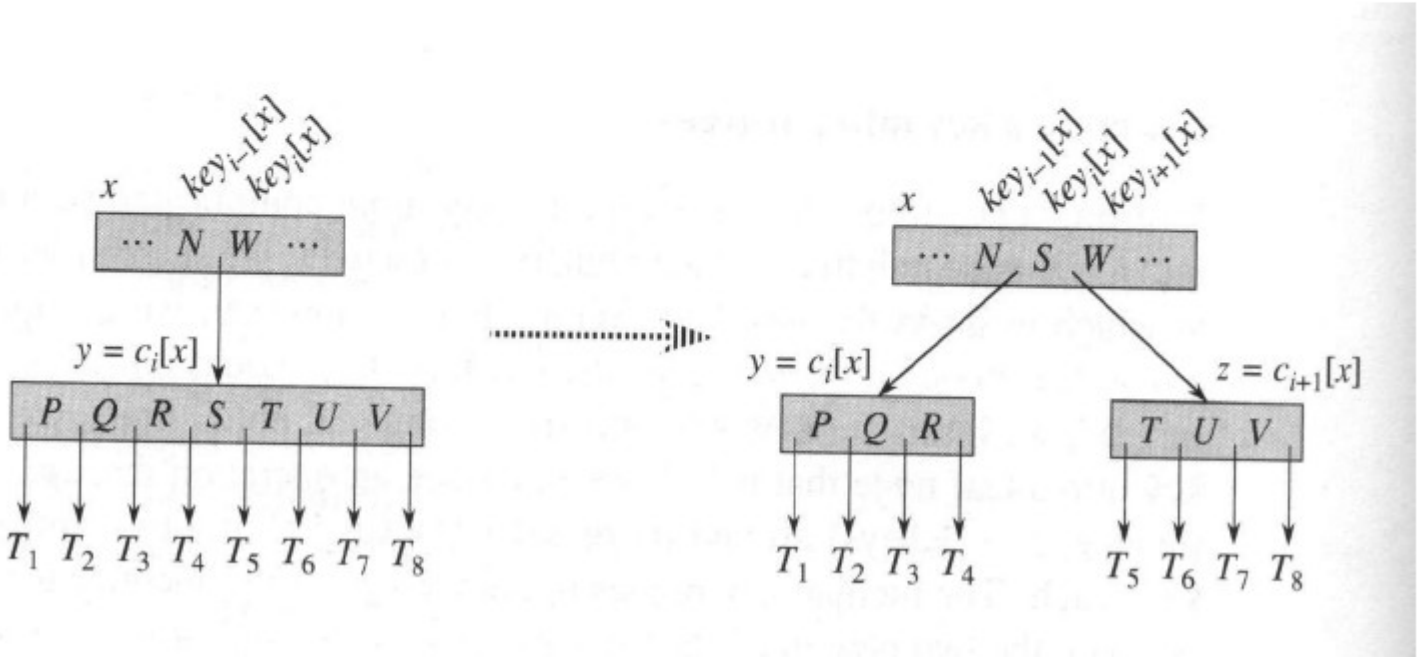

Ex: inserção do “5”

# Inserção em árvore B

- O que aconteceria se o nó pai também estiver cheio?
  - Teria que dividir também o pai, o que poderia causar subdivisões em cascata
  - Todo nó teria que guardar ponteiros para o nó pai
    - Aumentar o conteúdo de um nó → diminui o fator de blocagem → diminui o fator de ramificação → aumenta a altura da árvore
  - Além disso (e principalmente), as subdivisões podem alterar em qual nó o registro deve ser armazenado
- Então uma solução é, durante a busca da localização de inserção do nó, no caminho da raiz até uma folha, se achar um nó filho (a ser seguido) cheio, já o subdivide
- Vamos assumir que o nó atual (pai do nó cheio) não é cheio, a menos da raiz que deve ser tratada separadamente

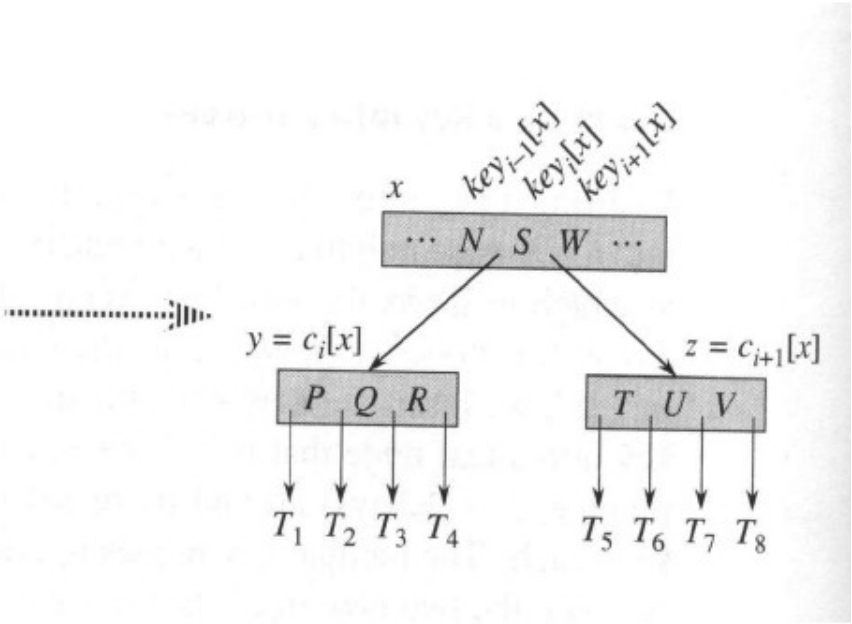
- Divisão de um nó na árvore:

  B-Tree-Split-Child$(x, i, y)$: tem como entrada um nó interno $x$ *não cheio*, um índice $i$ e um nó $y$ tal que $y = c_i[x]$ é um filho *cheio* de $x$. O procedimento divide $y$ em 2 e ajusta $x$ de forma que este terá um filho adicional.



B-TREE-SPLIT-CHILD$(x, i, y)$

O que precisa fazer?



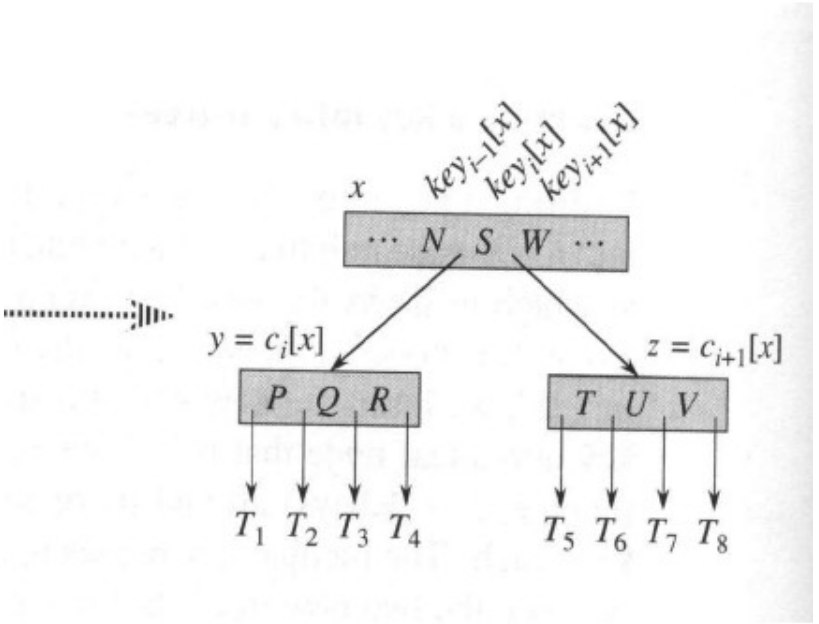
B-TREE-SPLIT-CHILD$(x, i, y)$

O que precisa fazer?

Aloca e inicializa z

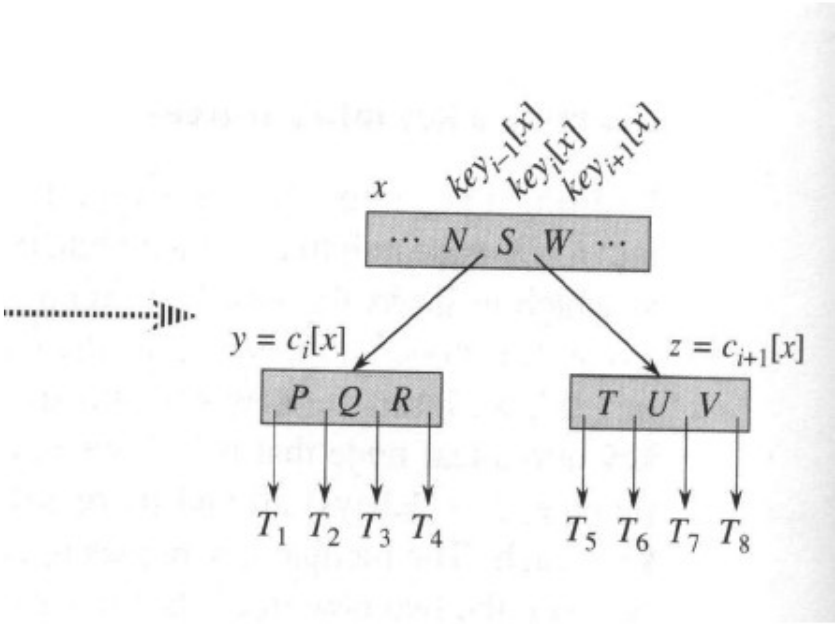
Ajusta y

Ajusta x



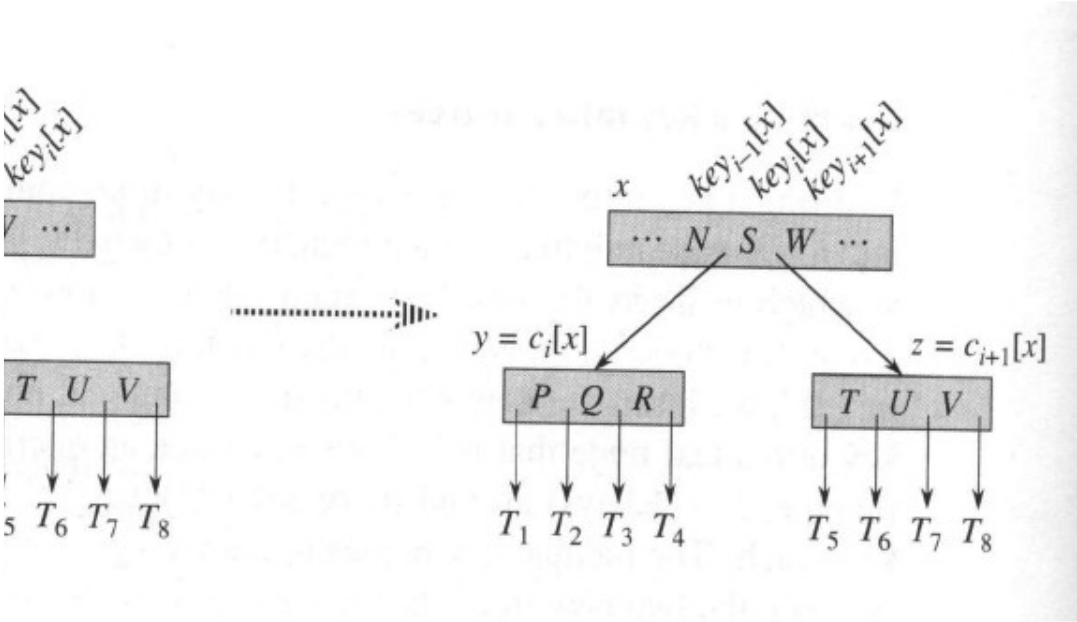
B-TREE-SPLIT-CHILD$(x, i, y)$

Aloca e inicializa z



```
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]
```

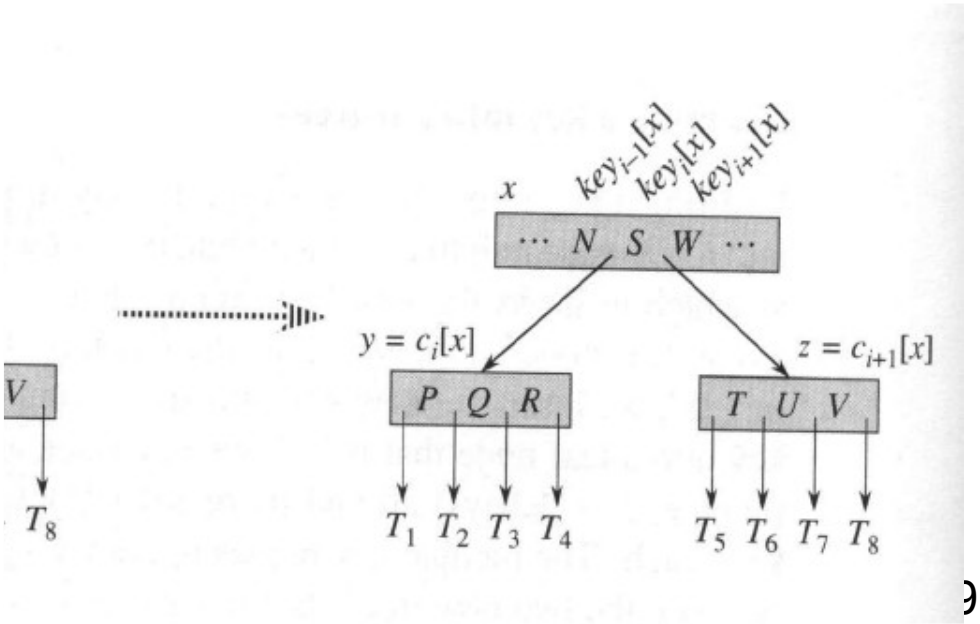
Aloca e inicializa z



```
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]
```

Aloca e inicializa z

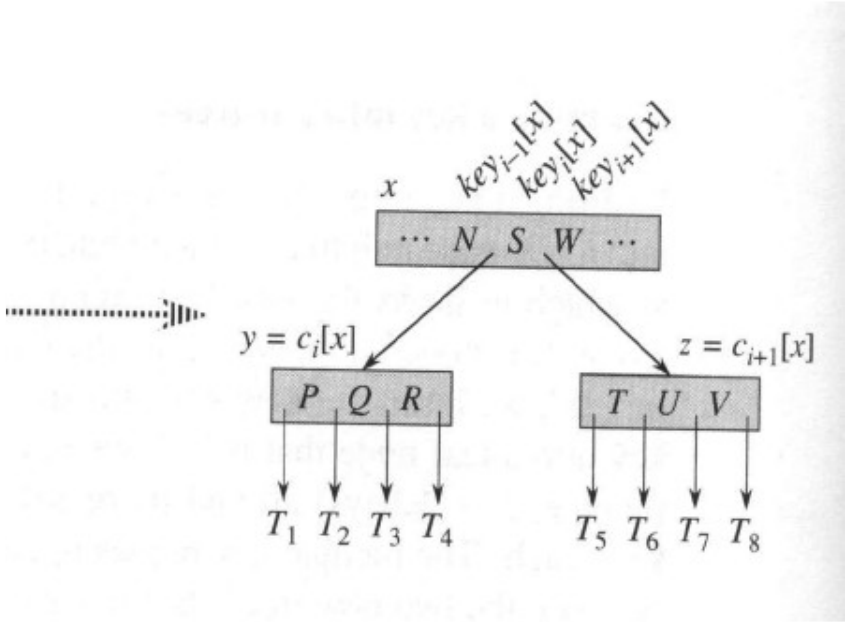
Ajusta y



```
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
```

Aloca e inicializa z

Ajusta y

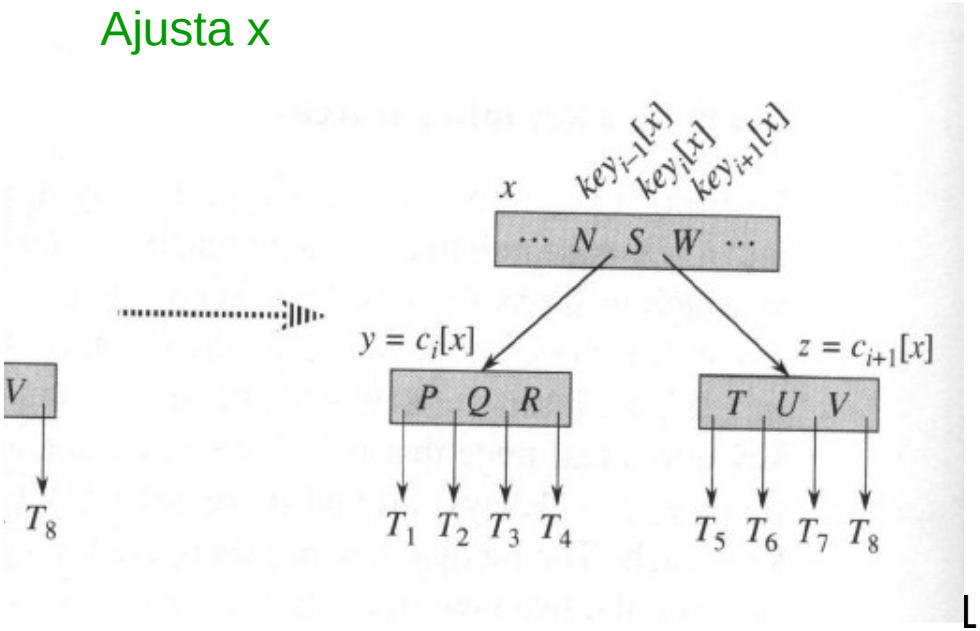


```
B-TREE-SPLIT-CHILD(x, i, y)
1  z ← ALLOCATE-NODE()
2  leaf[z] ← leaf[y]
3  n[z] ← t − 1
4  for j ← 1 to t − 1
5       do key_j[z] ← key_{j+t}[y]
6  if not leaf[y]
7     then for j ← 1 to t
8               do c_j[z] ← c_{j+t}[y]
9  n[y] ← t − 1
```

Aloca e inicializa z

Ajusta y

Ajusta x



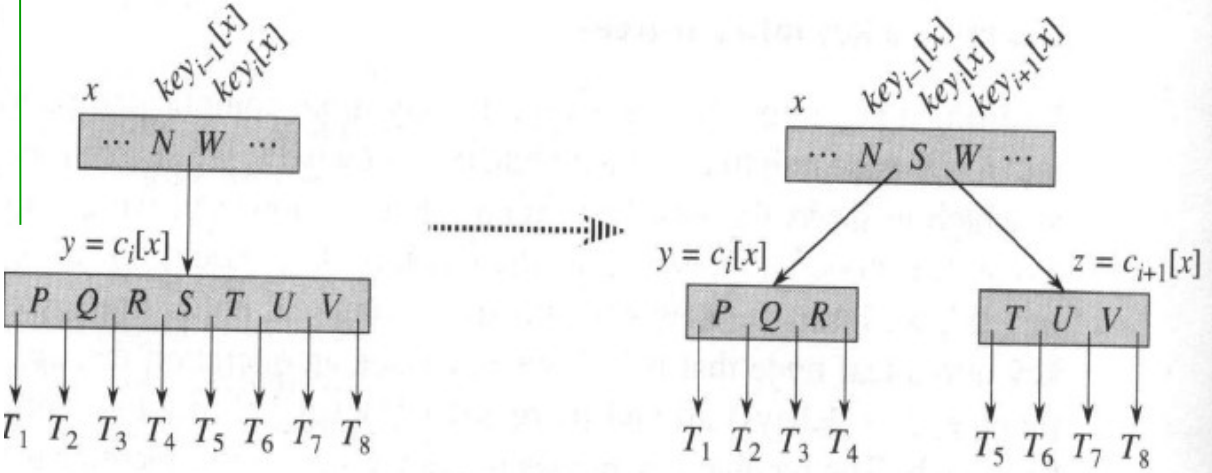

L

```
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)
```

Aloca e inicializa z

Ajusta y

Ajusta x



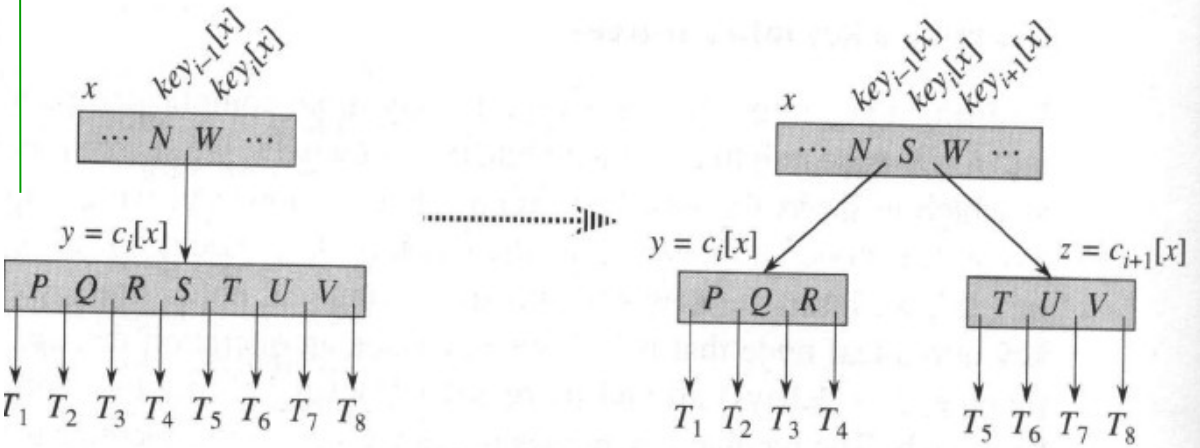
**Note que assume-se que as chaves e filhos começam na posição 1 !!!**

```
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)
```

Aloca e inicializa z

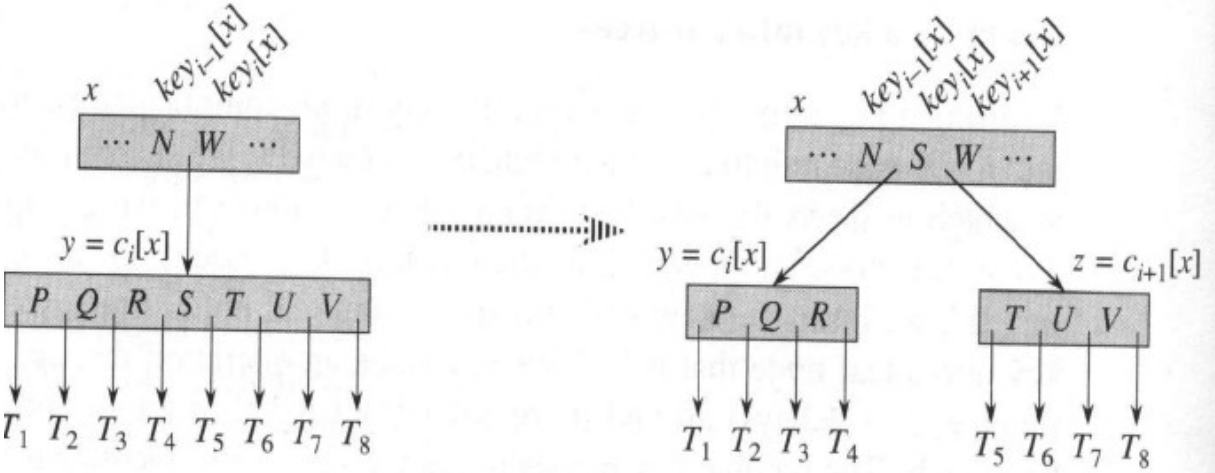
Ajusta y

Ajusta x



```
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**:

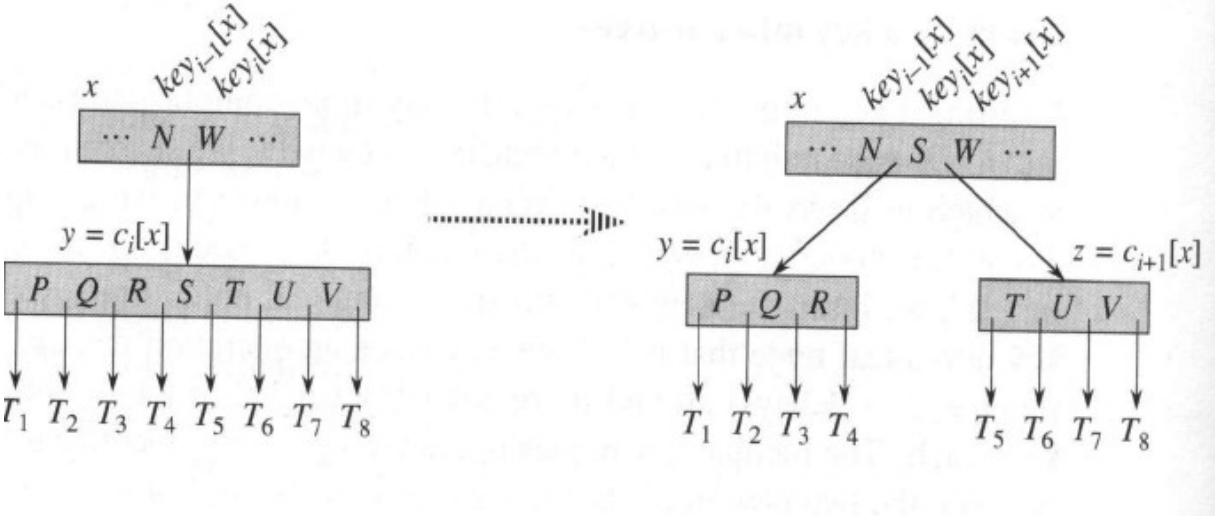


```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do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

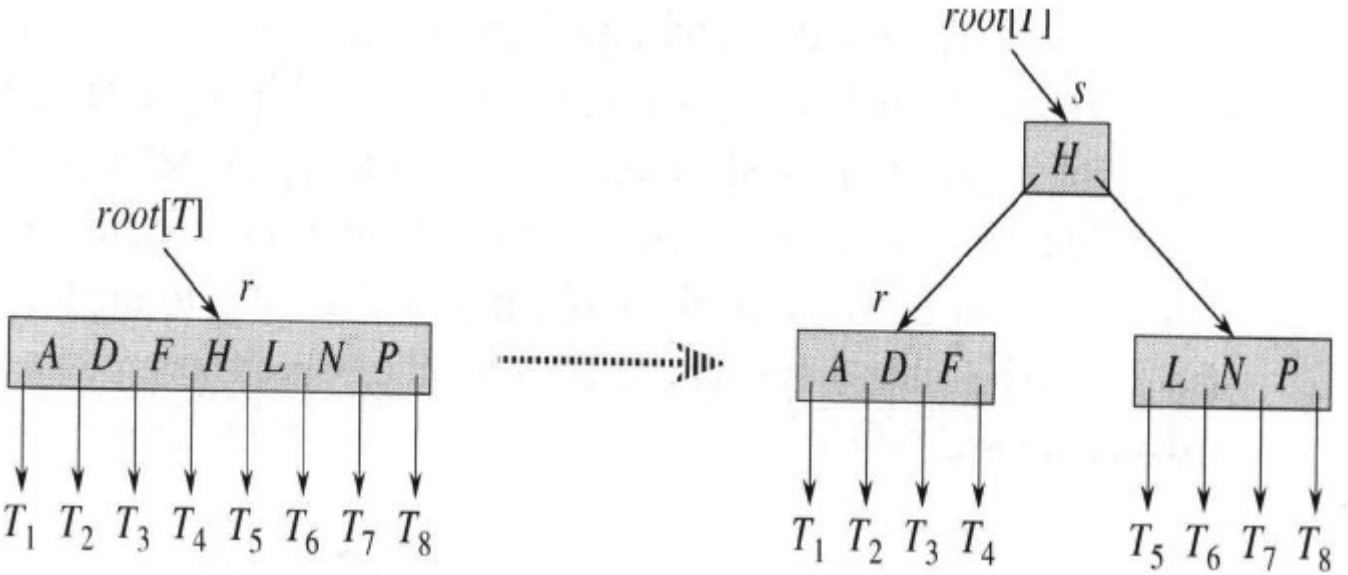
**COMPLEXIDADE**:

Acessos ao disco: O(1)

CPU: O(t)



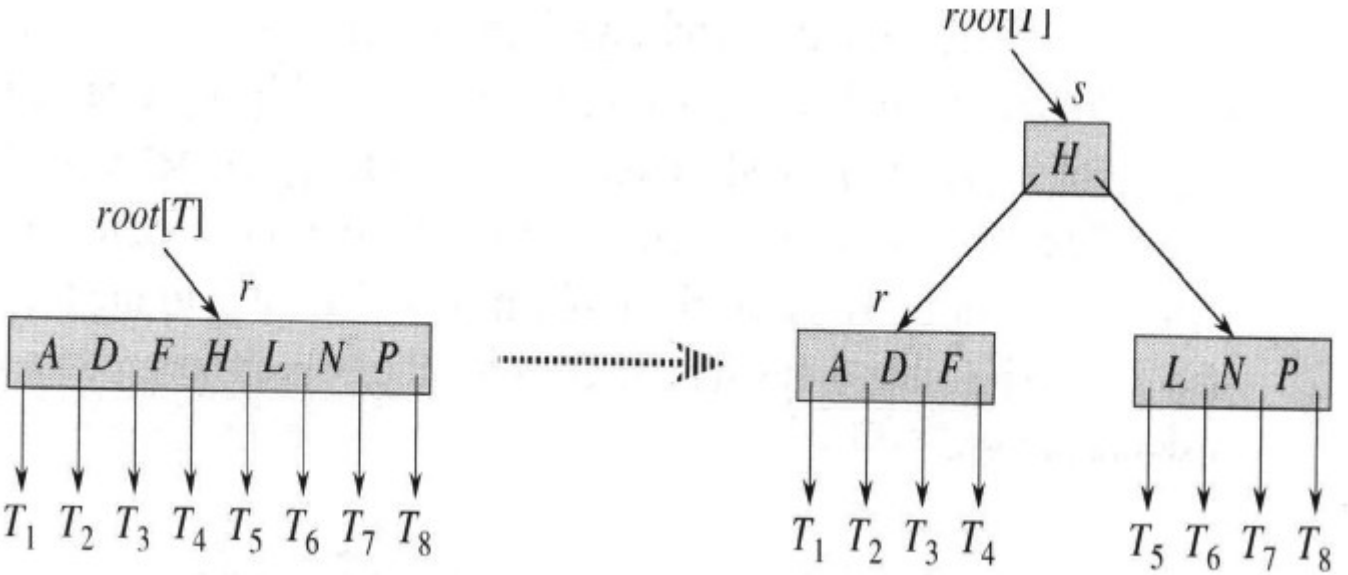
- Inserção de uma chave na árvore com raiz $T$:




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)
```

- Inserção de uma chave na árvore com raiz $T$:




```
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)
```

Não preciso escrever *s* no disco pois isso já será feito no B-Tree-Split-Child

- Inserção de uma chave em uma subárvore cuja raiz $x$ não está cheia:

```
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)
```

- Inserção de uma chave em uma subárvore cuja raiz $x$ não está cheia:

```
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: $O(h) = O(\lg_t b)$

CPU: $O(t*h) = O(t \lg_t b)$

- Inserção de uma chave na árvore com raiz $T$:

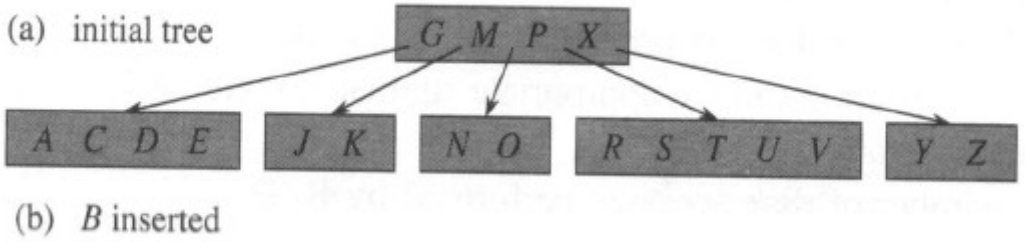
```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then s ← ALLOCATE-NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c1[s] ← r
 8          B-TREE-SPLIT-CHILD(s, 1, r)
 9          B-TREE-INSERT-NONFULL(s, k)
10     else B-TREE-INSERT-NONFULL(r,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

**COMPLEXIDADE**:

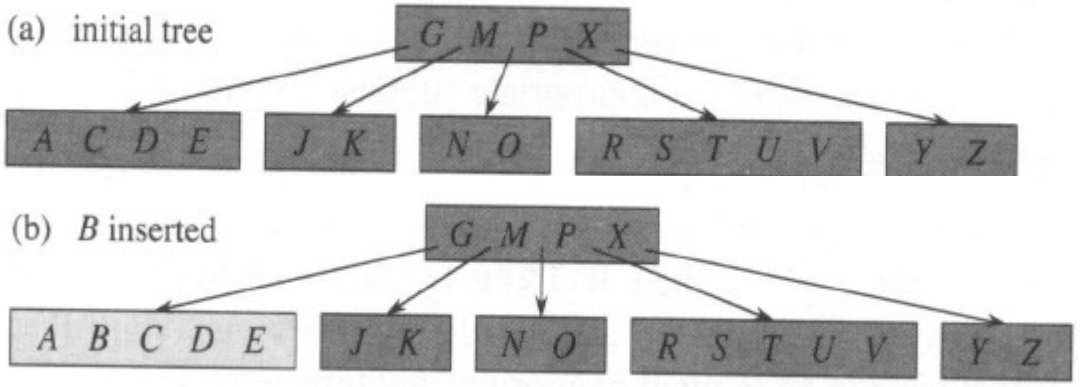
Acessos ao disco: O(h) = $O(lg_t\ b)$

CPU: O(t*h) = $O(t\ lg_t\ b)$

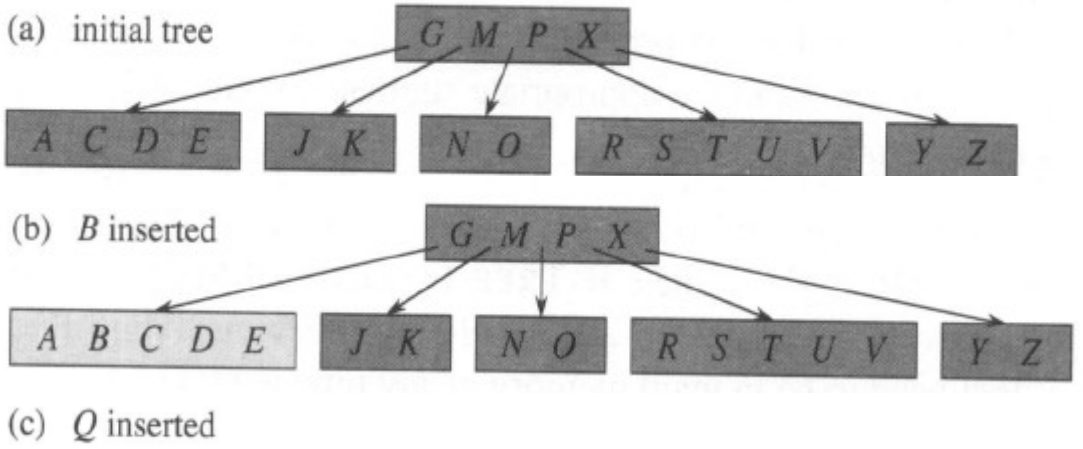
Não preciso escrever *s* no disco pois isso já será feito no B-Tree-Split-Child




# Exemplo Inserção t = 3

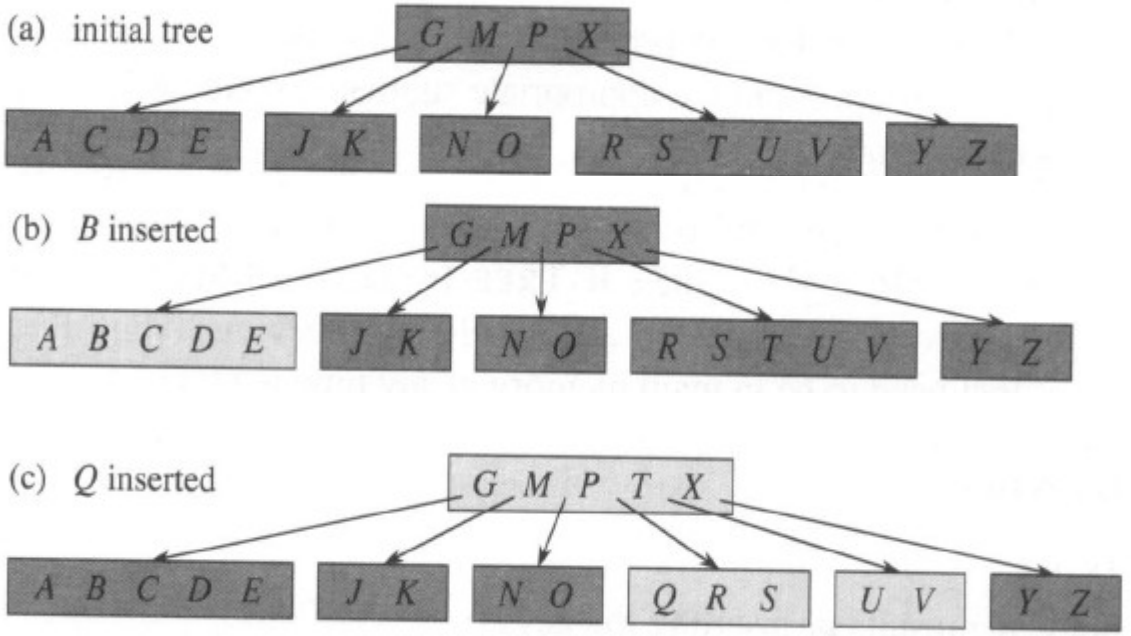



# Exemplo Inserção t = 3

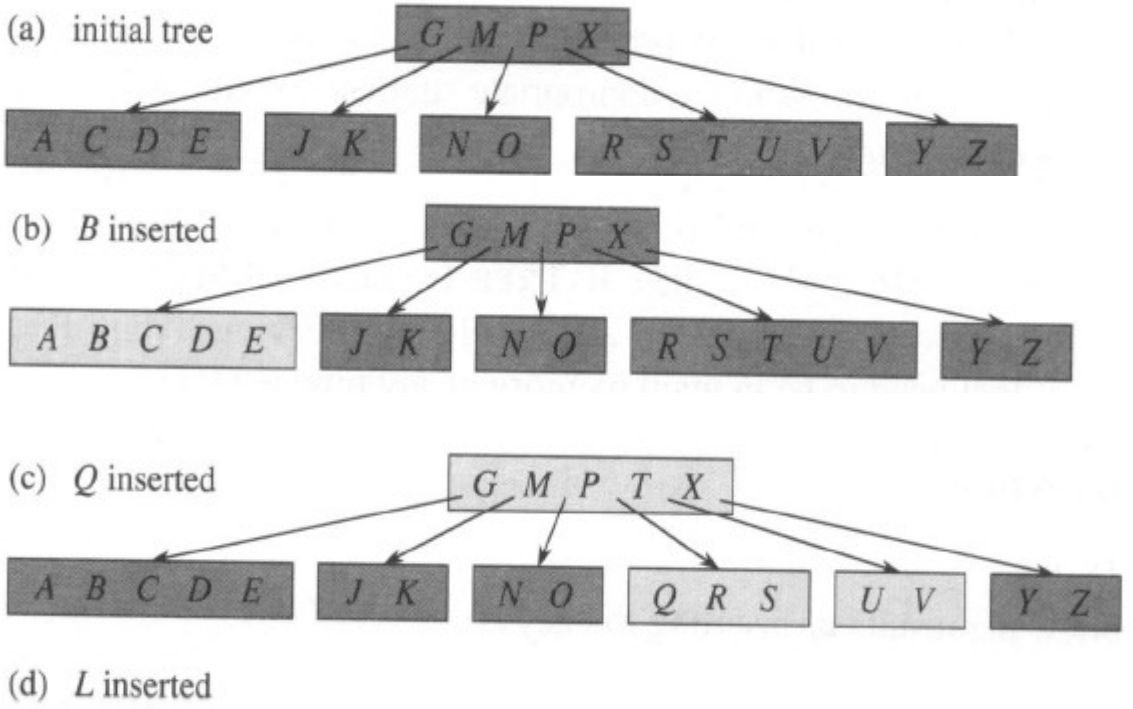



# Exemplo Inserção t = 3

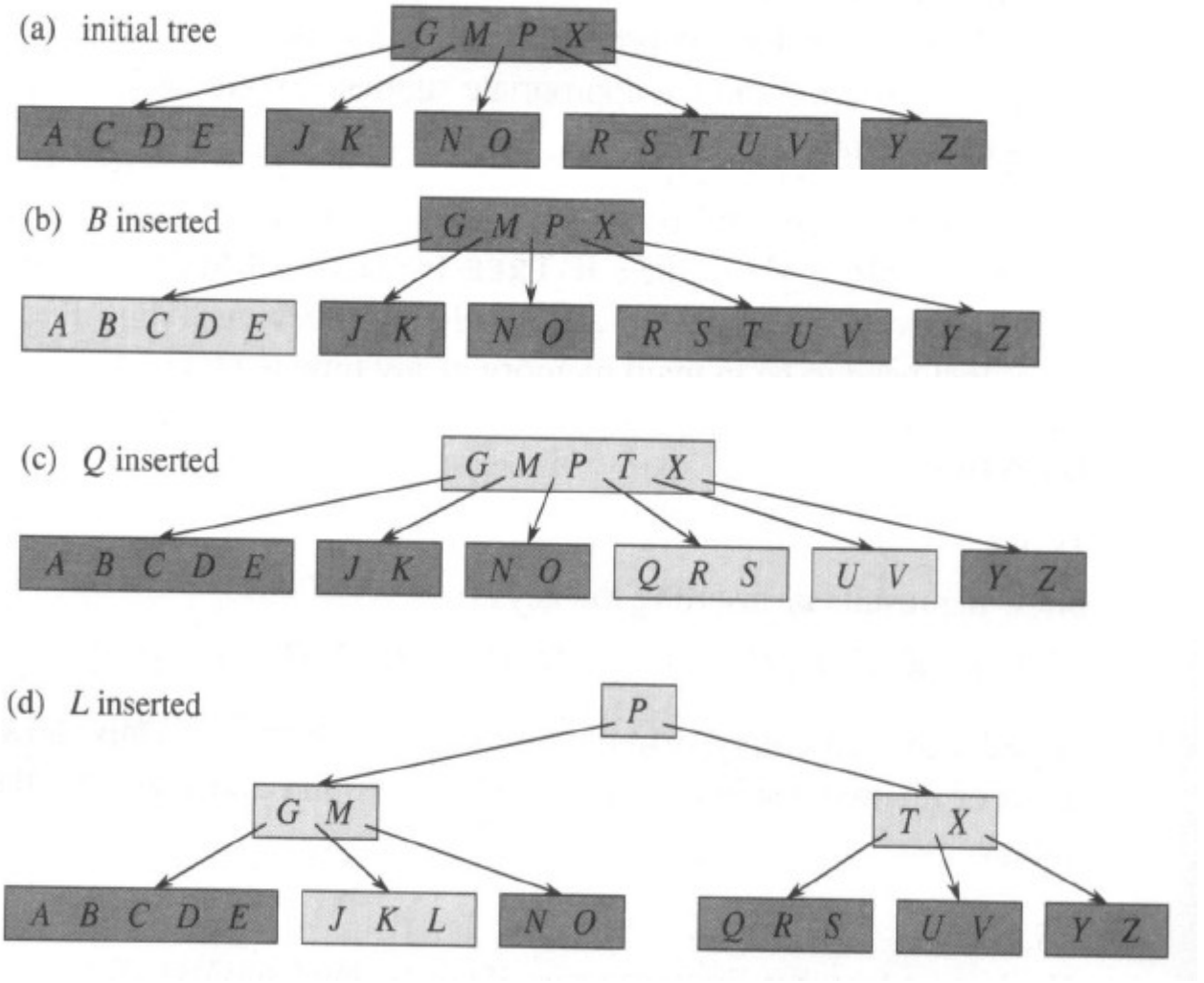
# Exemplo Inserção t = 3



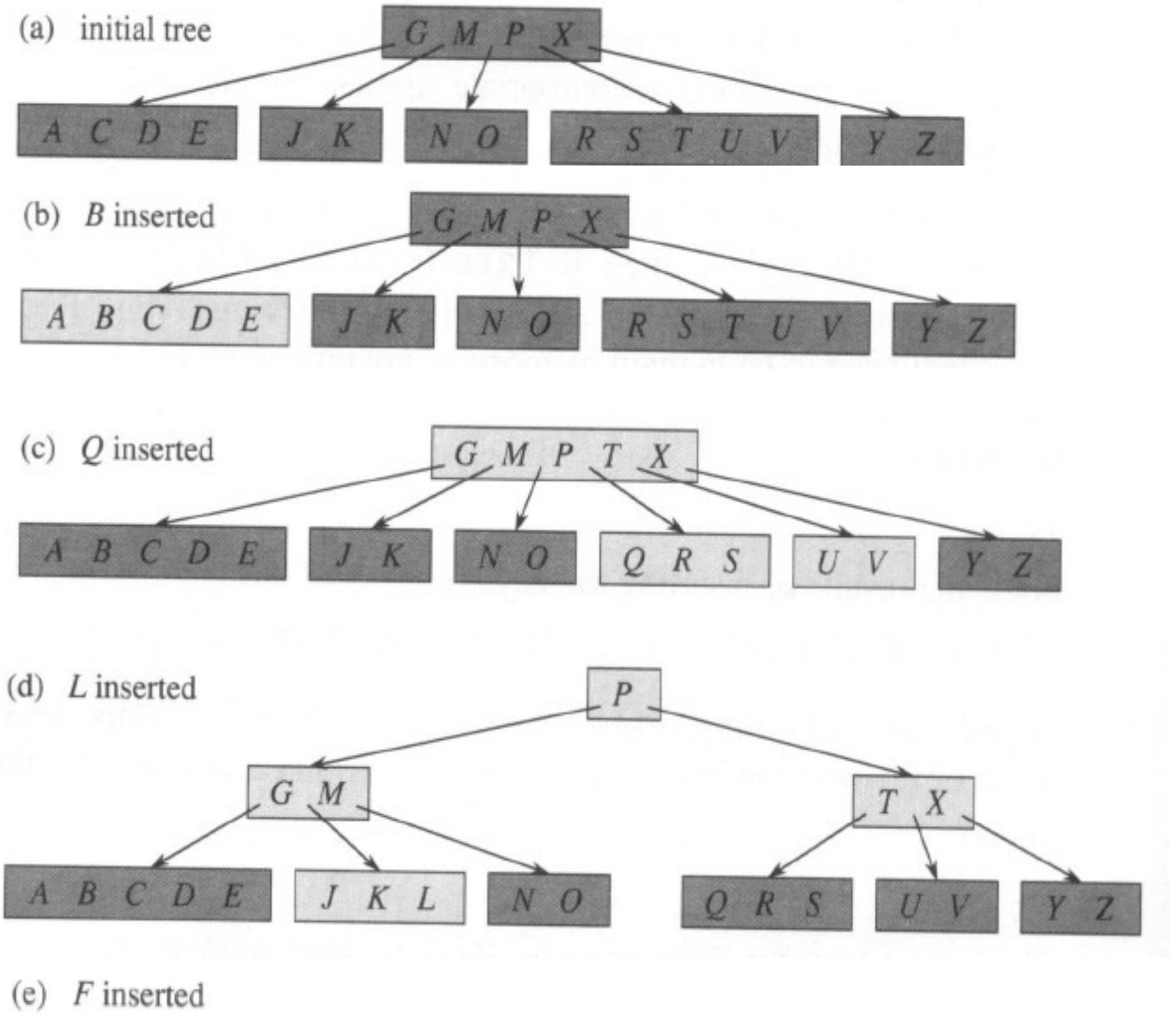
# Exemplo Inserção t = 3



# Exemplo Inserção t = 3



# Exemplo Inserção t = 3



# Exemplo Inserção t = 3