ACH2024

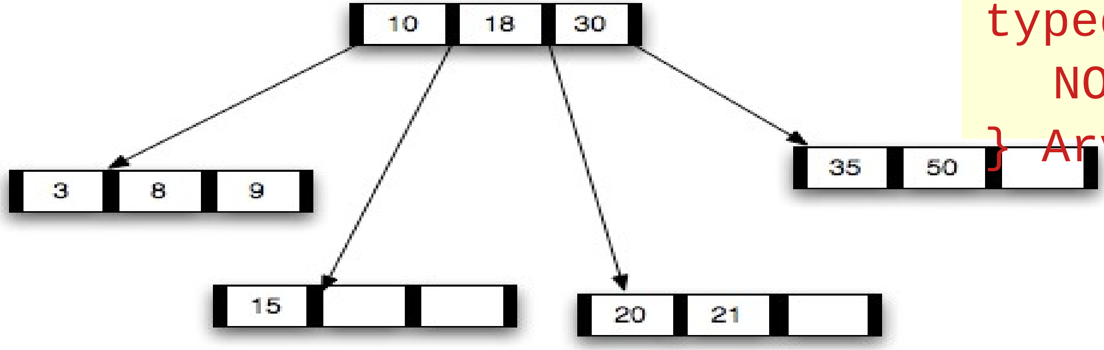
Aula 18

Organização de arquivos
Árvores B – parte 2 (inserção e remoção)

Profa. Ariane Machado Lima

Aula passada

Estrutura de uma árvore B




```c
#define t 2  /* grau min da árvore*/
typedef  int  TipoChave;
typedef struct str_no {
    TipoChave chave[2*t-1];
    struct str_no* filhos[2*t];
    int numChaves;
    bool folha;
} NO;
typedef struct {
  NO* raiz;
} ArvB;
```

Complexidade: O(1) seeks (1 especificamente)

Criação de uma árvore B vazia

```
#define t 2
typedef  int  TipoChave;
typedef struct str_no {
    TipoChave chave[2*t-1];
    struct str_no* filhos[2*t];
    int numChaves;
    bool folha;
} NO;
typedef struct {
   NO* raiz;
} ArvB;
```

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

```
bool criaArvoreB(ArvB* T){
  NO* x;
  if(!(x = (NO*) malloc(sizeof(NO)))) {
      /* msg erro e retorna false */
  }
  x->folha = true;
  x->numChaves = 0;
  escreveNoDisco(x); /*vamos abstrair isso*/
  T->raiz = x;
  retorna true;
}
```

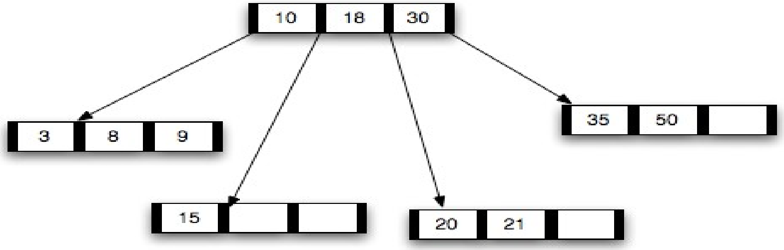
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

Busca na 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$(root[T], k)$.

Ex: B-Tree-Search(T→raiz, 22)

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

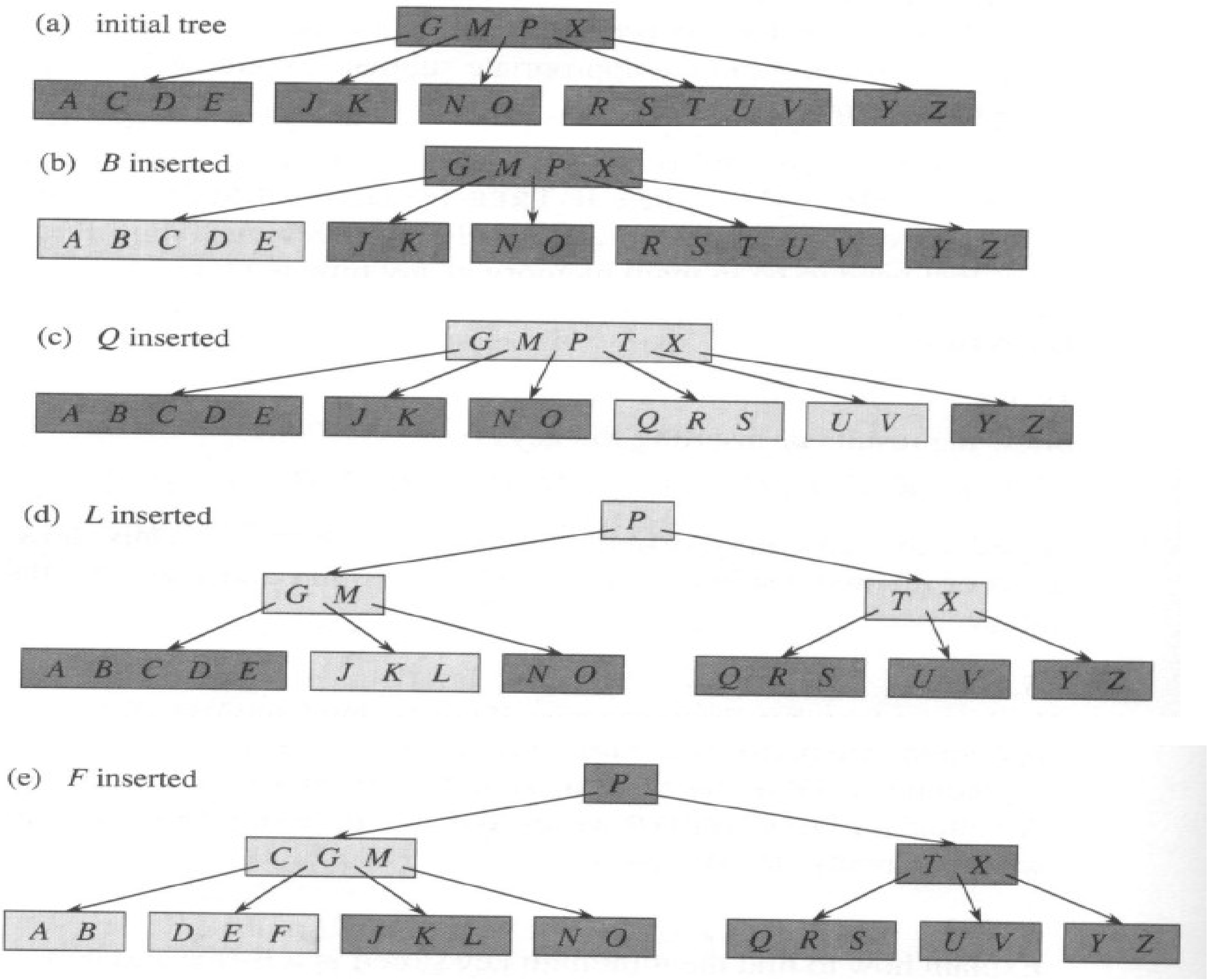


Complexidade: O(h) seeks (O($\log_t$ b))

Inserção em árvore B

As inserções ocorrem sempre nas folhas

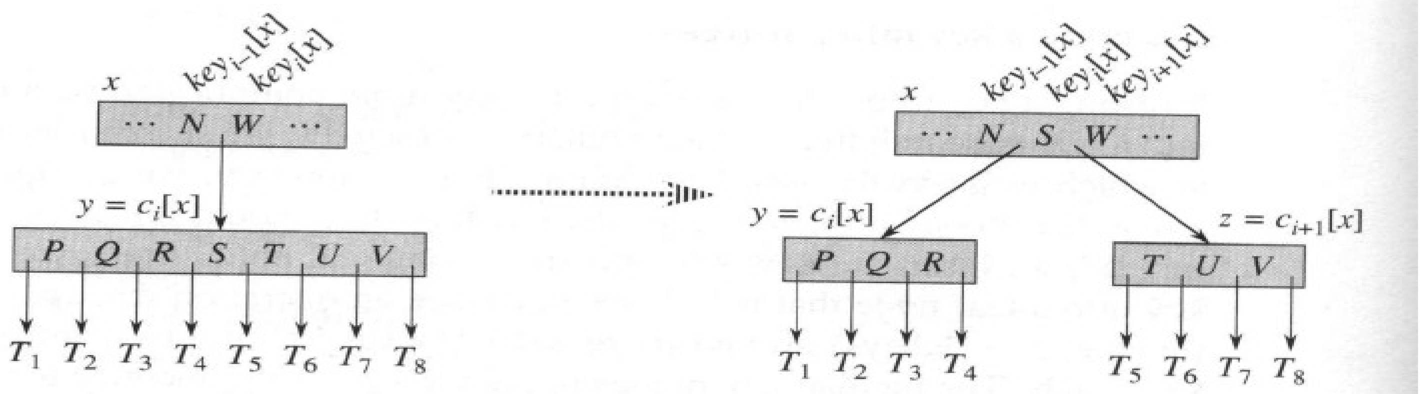
Exemplo Inserção t = 3



Porque se ele estava cheio já foi dividido

- 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.



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

```
B-TREE-SPLIT-CHILD(x, i, y)
z ← ALLOCATE-NODE()
leaf[z] ← leaf[y]
n[z] ← t − 1
for j ← 1 to t − 1
        do key_j[z] ← key_{j+t}[y]
if not leaf[y]
    then for j ← 1 to t
                do c_j[z] ← c_{j+t}[y]
n[y] ← t − 1
for j ← n[x] + 1 downto i + 1
        do c_{j+1}[x] ← c_j[x]
c_{i+1}[x] ← z
for j ← n[x] downto i
        do key_{j+1}[x] ← key_j[x]
key_i[x] ← key_t[y]
n[x] ← n[x] + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

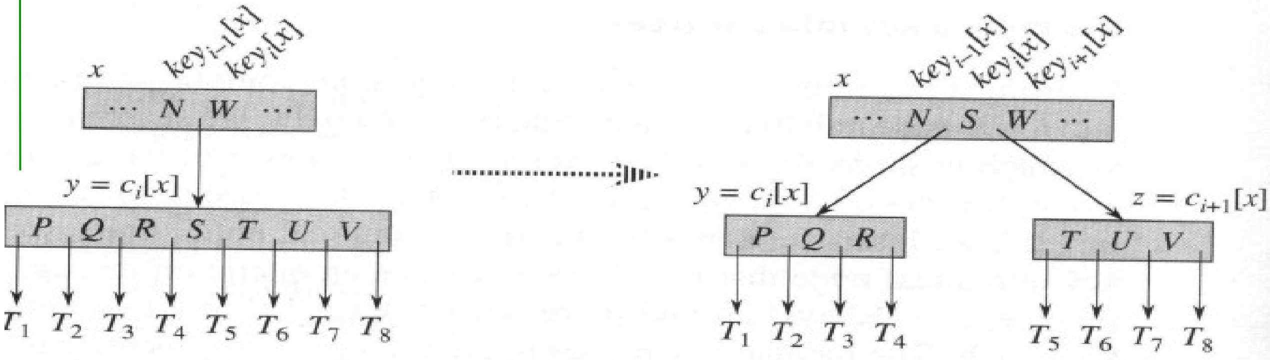
Aloca e inicializa z

COMPLEXIDADE:

O(1) seeks (3 mais precisamente)

Ajusta y

Ajusta x



Aula de hoje

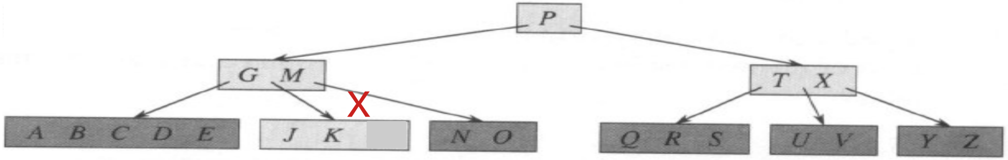
- Continuação da inserção

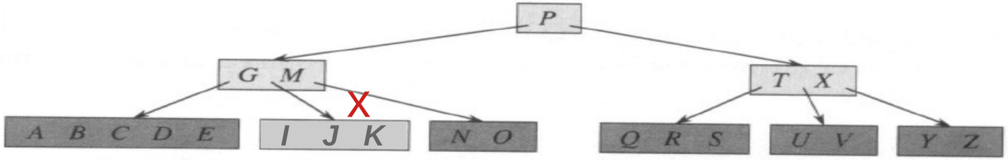
- Deleção

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

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
            do key_{i+1}[x] ← key_i[x]
               i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
            do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
          then B-TREE-SPLIT-CHILD(x, i, c_i[x])
               if k > key_i[x]
                 then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

Ex: B-Tree-Insert-Nonfull (x,I)

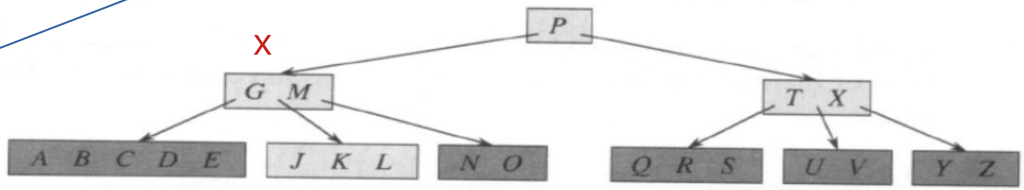




- Inserção de uma chave em uma subárvore cuja raiz x não está cheia:

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
              do key_{i+1}[x] ← key_i[x]
                 i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
              do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
           then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                if k > key_i[x]
                   then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

Ex: B-Tree-Insert-Nonfull (x,F)



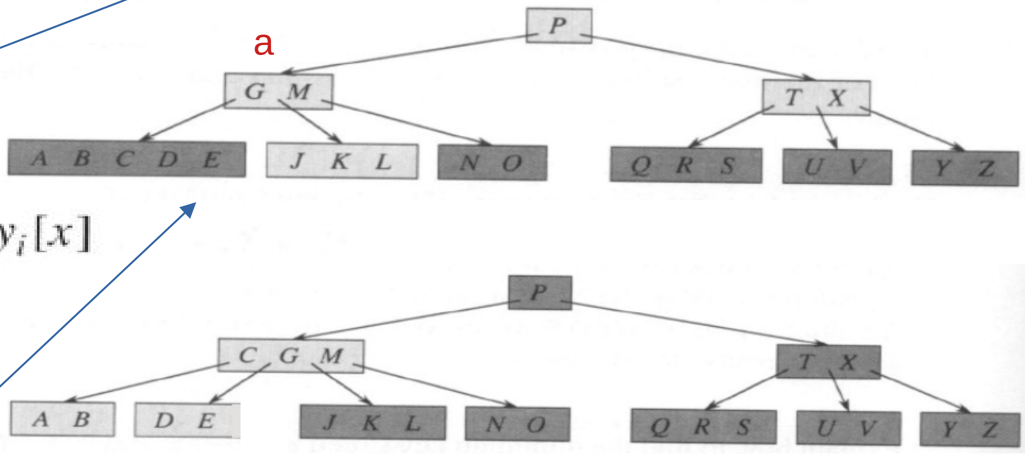
Procuro por qual filho tenho que descer

Nó filho cheio (verifico antes de descer recursivamente para ele)

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

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
              do key_{i+1}[x] ← key_i[x]
                 i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
              do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
           then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                if k > key_i[x]
                   then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

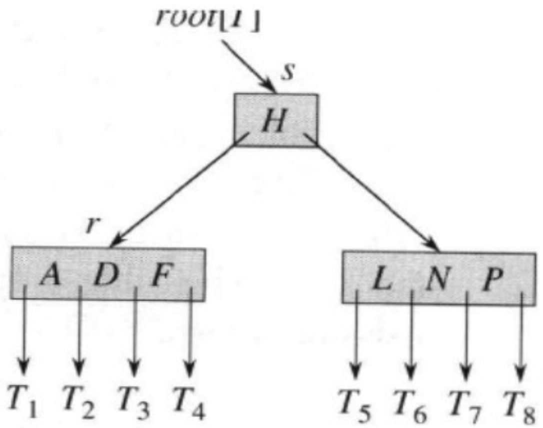
Ex: B-Tree-Insert-Nonfull (a,F)



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

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
              do key_{i+1}[x] ← key_i[x]
                 i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
              do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
           then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                if k > key_i[x]
                   then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

Ex: B-Tree-Insert-Nonfull (s,C)



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

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
              do key_{i+1}[x] ← key_i[x]
                 i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
              do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
           then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                if k > key_i[x]
                   then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

COMPLEXIDADE:

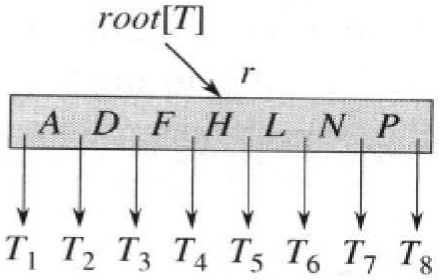
- Inserção de uma chave em uma subárvore cuja raiz x não está cheia:

```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
              do key_{i+1}[x] ← key_i[x]
                 i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < key_i[x]
              do i ← i − 1
        i ← i + 1
        DISK-READ(c_i[x])
        if n[c_i[x]] = 2t − 1
           then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                if k > key_i[x]
                   then i ← i + 1
        B-TREE-INSERT-NONFULL(c_i[x], k)
```

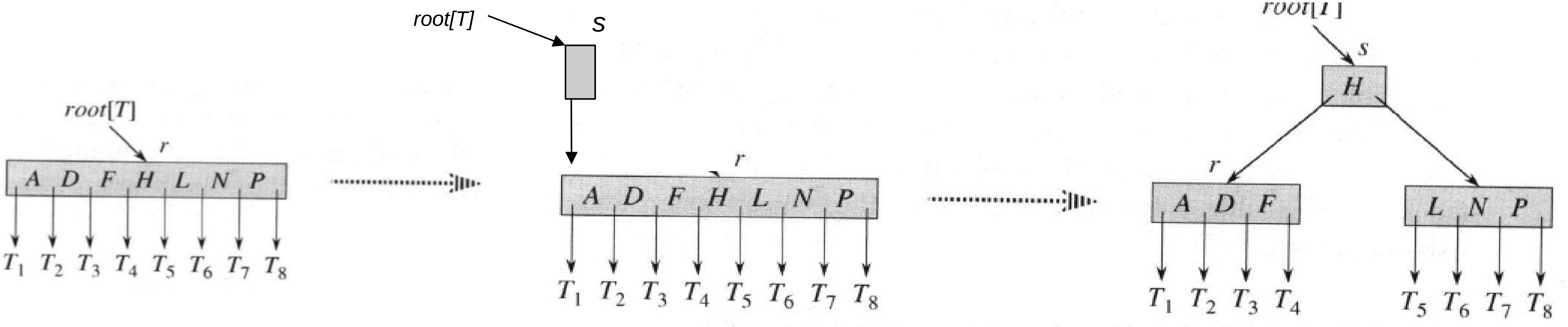
COMPLEXIDADE:

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

- Inserção de uma chave na árvore com raiz T:



- Inserção de uma chave na árvore com raiz T:




```
B-TREE-INSERT(T, k)
r ← root[T]
if n[r] = 2t − 1
    then s ← ALLOCATE-NODE()
         root[T] ← s
         leaf[s] ← FALSE
         n[s] ← 0
         c1[s] ← r
         B-TREE-SPLIT-CHILD(s, 1, r)
         B-TREE-INSERT-NONFULL(s, k)
    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 na árvore com raiz T:

COMPLEXIDADE TOTAL DA INSERÇÃO:

```
B-TREE-INSERT(T, k)
r ← root[T]
if n[r] = 2t − 1
    then s ← ALLOCATE-NODE()
         root[T] ← s
         leaf[s] ← FALSE
         n[s] ← 0
         c1[s] ← r
         B-TREE-SPLIT-CHILD(s, 1, r)
         B-TREE-INSERT-NONFULL(s, k)
    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 na árvore com raiz T:

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

COMPLEXIDADE TOTAL DA INSERÇÃO:

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

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

Remoção em árvores B

B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

Que propriedades gostaríamos que nossa remoção tivesse?

Remoção em árvores B

B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

Que propriedades gostaríamos que nossa remoção tivesse?

1. Ela precisa manter as propriedades da árvore B
2. Já que vou remover, será que consigo diminuir a altura da árvore?
   Para isso, a cada remoção quero liberar espaço em uma folha

Remoção em árvores B

B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

Que propriedades gostaríamos que nossa remoção tivesse?

1. Ela precisa manter as propriedades da árvore B
2. Já que vou remover, será que consigo diminuir a altura da árvore? Para isso, a cada remoção quero liberar espaço em uma folha

3 casos a serem tratados (com seus subcasos):

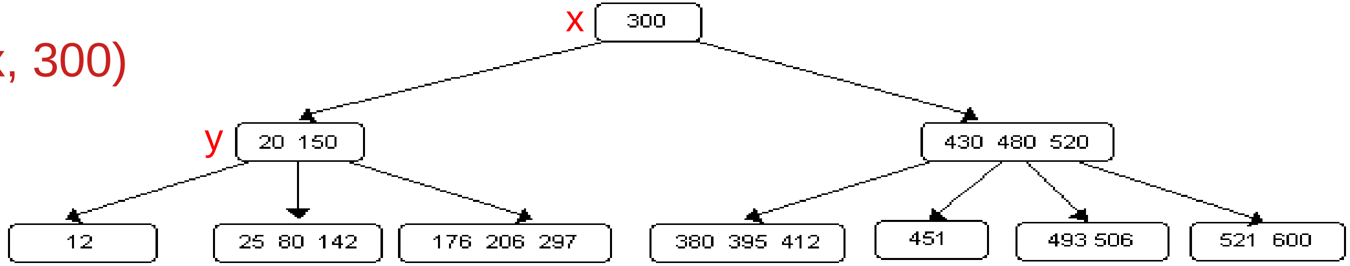
- x é nó interno e a chave k está lá
- x é nó interno mas a chave k não está lá
- x é folha (e a chave k está ou não lá)

Remoção em árvores B

- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

2. Se a chave k está no nó x e x é um nó interno, faça:

B-Tree-Delete(x, 300)
(t = 2)



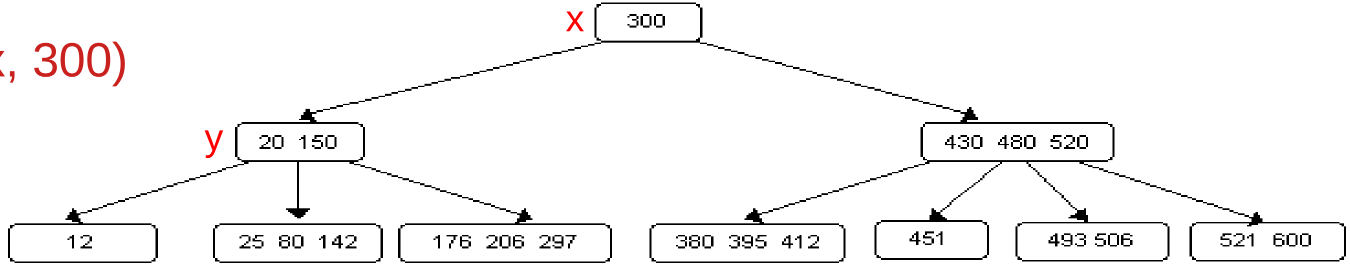
Remoção em árvores B

- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

  2. Se a chave k está no nó x e x é um nó interno, faça:

     a) Se o filho y que precede k no nó x tem pelo menos t chaves, então

B-Tree-Delete(x, 300)
(t = 2)



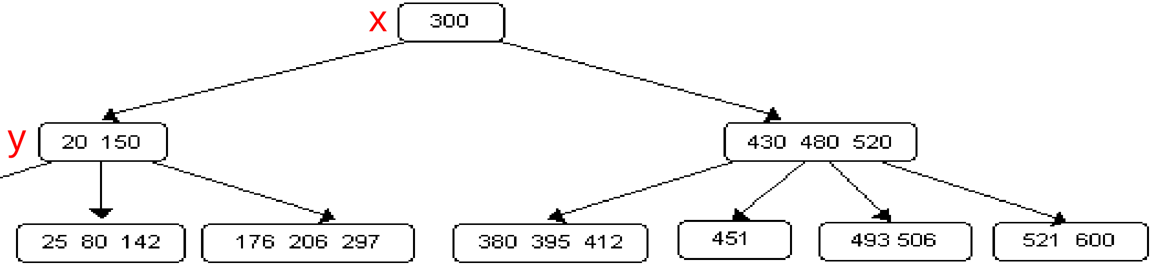
Remoção em árvores B

- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

  2. Se a chave k está no nó x e x é um nó interno, faça:

     a) Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz y. Delete recursivamente k', e substitua k por k' em x.

B-Tree-Delete(x, 300)
(t = 2)



Remoção em árvores B

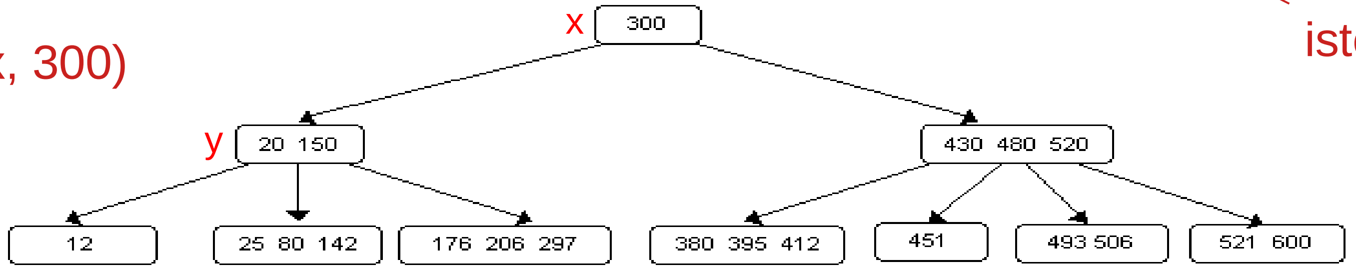
- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

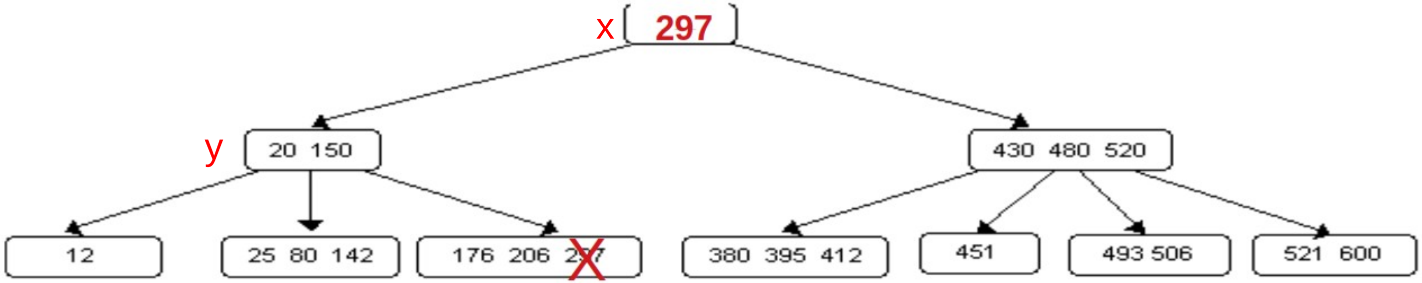
  2. Se a chave k está no nó x e x é um nó interno, faça:

     a) Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz y. Delete recursivamente k', e substitua k por k' em x.

isto é,

B-Tree-Delete(x, 300)
(t = 2)





Remoção em árvores B

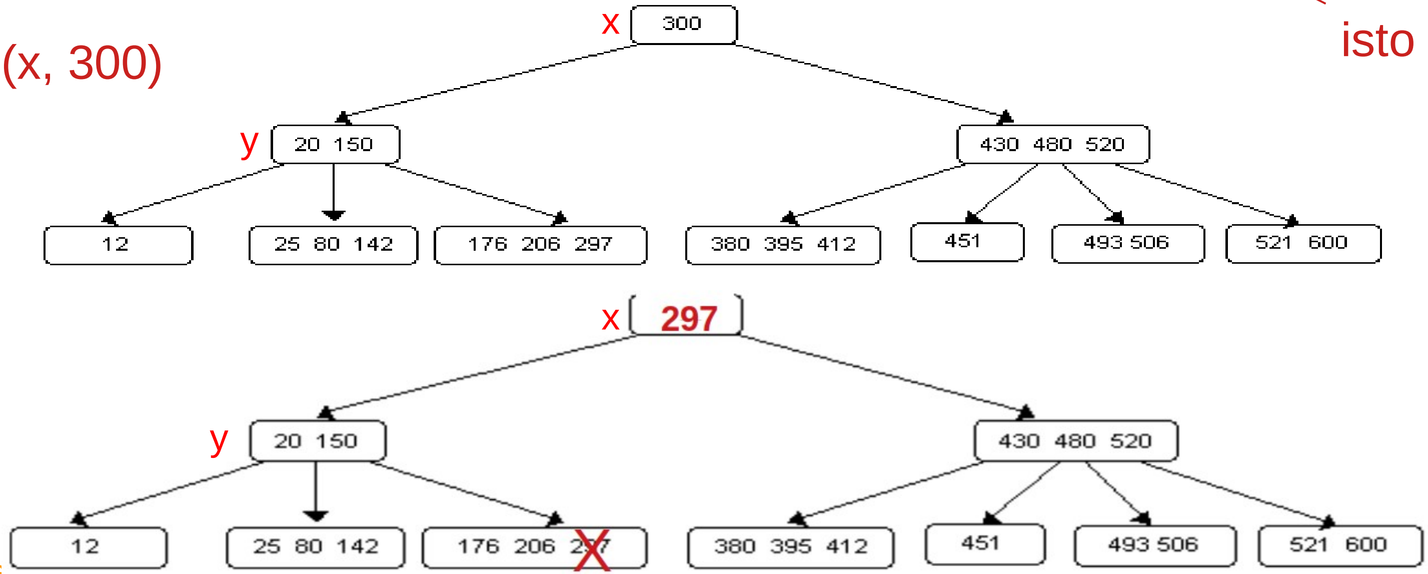
- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

  2. Se a chave k está no nó x e x é um nó interno, faça:

     a) Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz y. Delete recursivamente k', e substitua k por k' em x.

isto é, max(y)

B-Tree-Delete(x, 300)
(t = 2)



Remoção em árvores B

- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

2. Se a chave k está no nó x e x é um nó interno, faça:

a) Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz y. Delete recursivamente k', e substitua k por k' em x.

OU

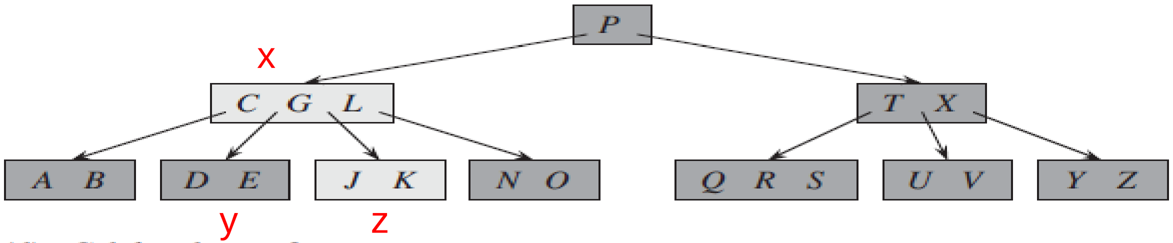
b) Simetricamente, se o filho z imediatamente após k no nó x tem pelo menos t chaves, então encontre o sucessor k' de k na subárvore com raiz z. Delete recursivamente k', e substitua k por k' em x.

Remoção em árvores B

- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

  2. Se a chave k está no nó x e x é um nó interno, faça:

     c) Caso contrário, se ambos y e z possuem apenas $t - 1$ chaves, faça a



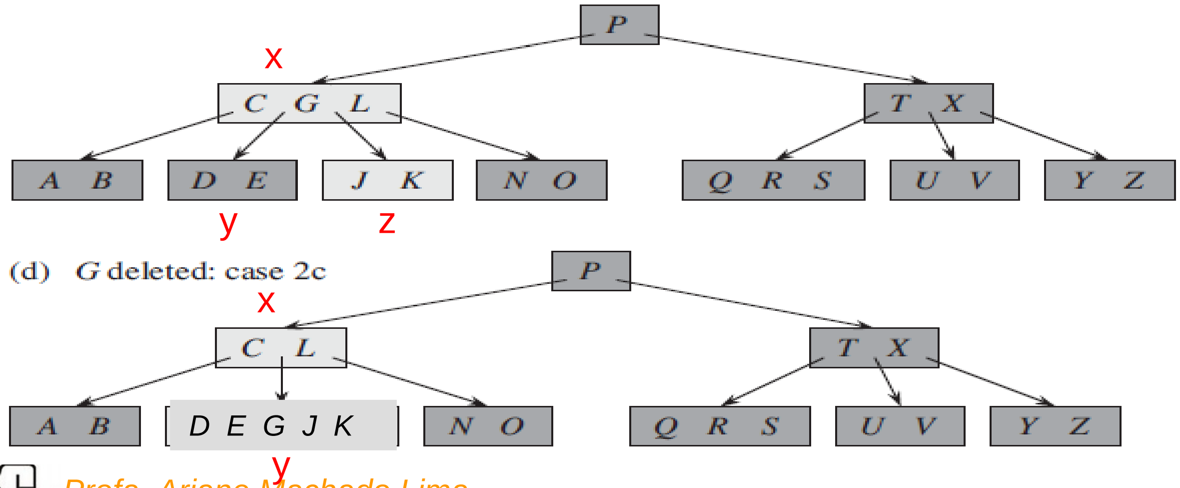
(d) G deleted: case 2c

Remoção em árvores B

- B-Tree-Delete(x,k): remoção da chave k da subárvore com raiz x.

2. Se a chave k está no nó x e x é um nó interno, faça:

   c) Caso contrário, se ambos y e z possuem apenas $t - 1$ chaves, faça a junção de k e todas as chaves de z em y, de forma que x perde tanto a chave k como o ponteiro para z, e y agora contém $2t - 1$ chaves. Então, libere z e delete recursivamente k de y.

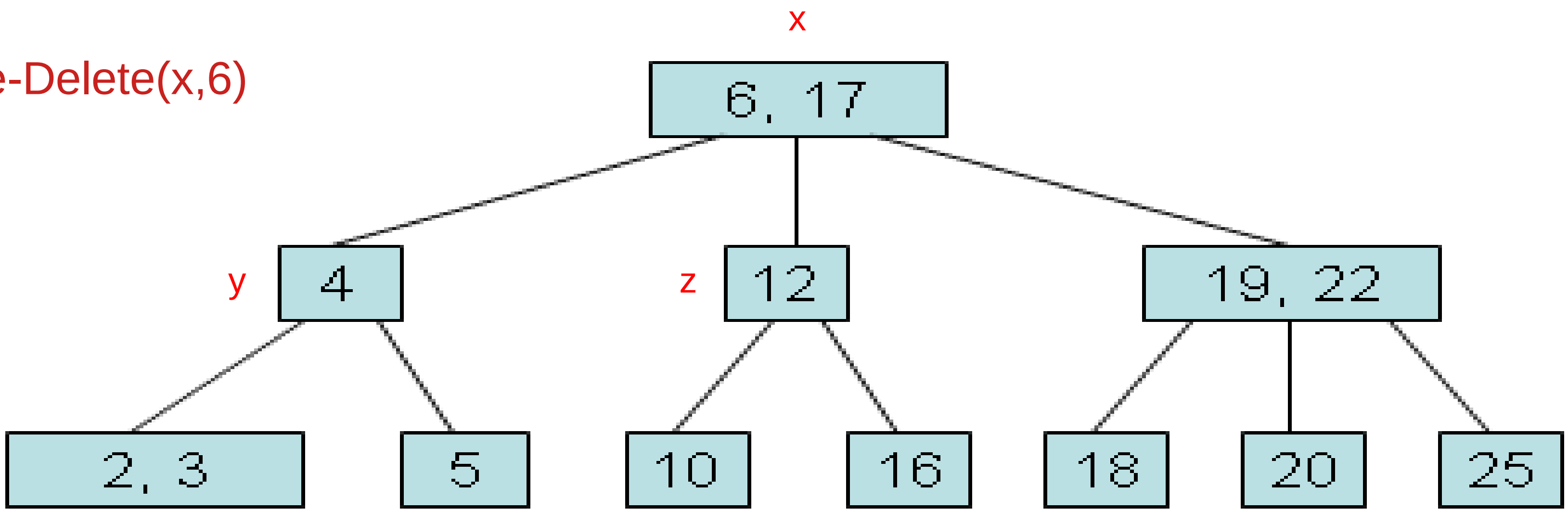


B-Tree-Delete(x,G)

Outro exemplo deste último caso: remoção do 6 (t = 2) : exercício

c) Caso contrário, se ambos y e z possuem apenas $t - 1$ chaves, faça a junção de k e todas as chaves de z em y, de forma que x perde tanto a chave k como o ponteiro para z, e y agora contém $2t - 1$ chaves. Então, libere z e delete recursivamente k de y.

B-Tree-Delete(x,6)



Continua na próxima aula

Referências

Livro do Cormen: (3ª ed.) cap 18

Livro do Drozdek (4ª ed) cap 7