

ACH2024

Aula 17

Organização de arquivos Árvores B

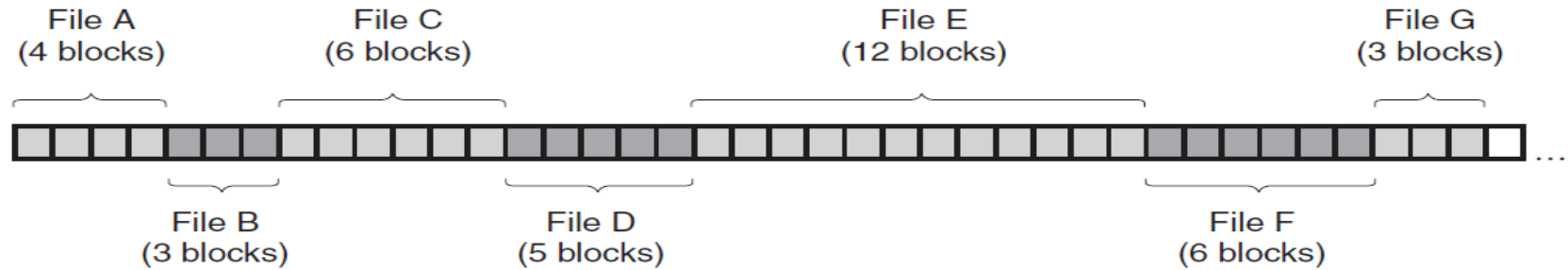
Profa. Arianne Machado Lima

Aulas passadas

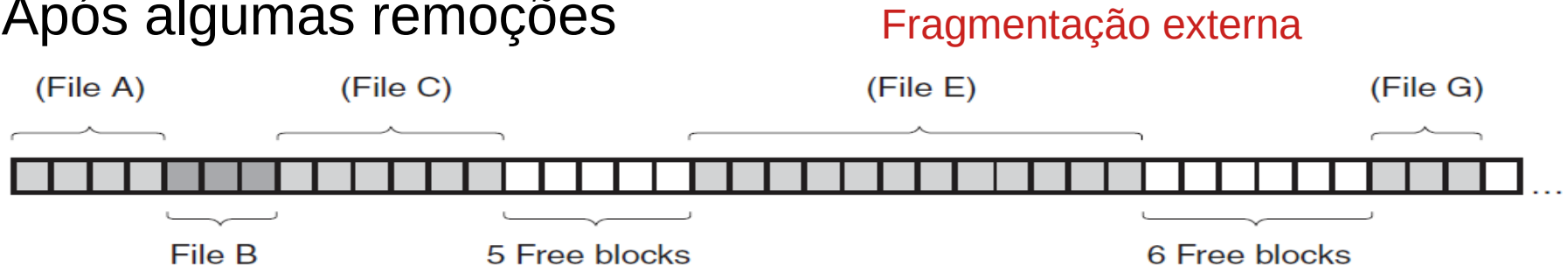
- Leitura, escrita, buscas, etc., são realizadas por blocos.
- Os arquivos não são estáticos, eles crescem e diminuem
- Estratégias de alocação de blocos no disco e organização de registros pelos blocos devem considerar esse fato
 - **Sequencial** não ordenado (heap files)
 - **Sequencial** ordenado (sorted files)
 - Por listas ligadas
 - Indexado
 - Árvores B / B+
 - Hashing
- Para cada estratégia analisaremos complexidade de leitura sequencial, leitura aleatória (busca), inserção e remoção de registros
- Complexidade em termos de número de seeks (estimado no pior caso pelo número de blocos a serem lidos); **assume-se que o arquivo já foi aberto e que o cabeçalho do arquivo está em memória**

Alocação sequencial

- Blocos alocados sequencialmente no disco (pelos cilindros)

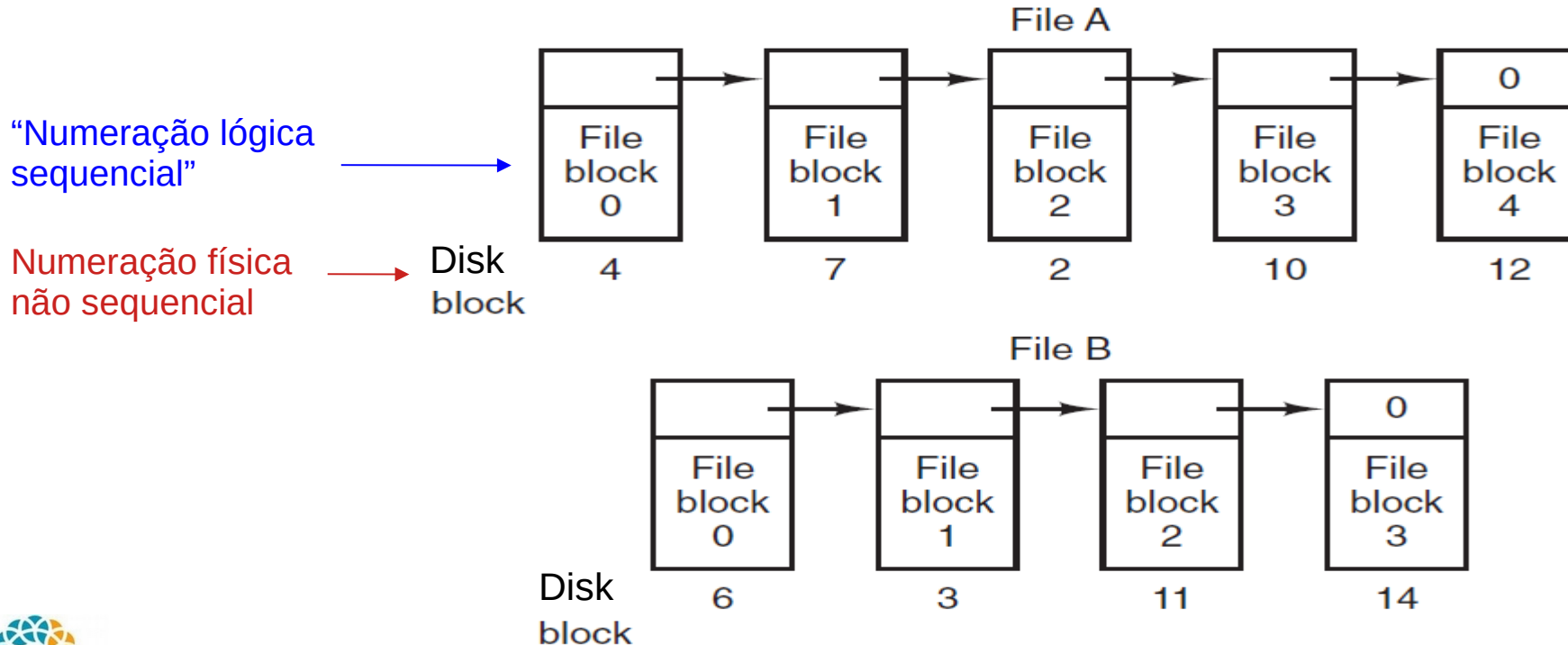


- Após algumas remoções



Alocação por listas ligadas

Cada arquivo é uma lista ligada de blocos



Fonte: (TANEMBAUM, 2015)

Organização indexada

Busca: $O(\lg bi)$
Inserção/Remoção: $O(b+bi)$
(ou $O(bi)$ se deixar ter blocos esparsos)

Index file
($\langle K(i), P(i) \rangle$ entries)

Block anchor primary key value	Block pointer
Aaron, Ed	•
Adams, John	•
Alexander, Ed	•
Allen, Troy	•
Anderson, Zach	•
Arnold, Mack	•
⋮	

⋮

(Primary
key field)

Name	Ssn	Birth_date	Job	Salary	Sex
Aaron, Ed					
Abbot, Diane					
⋮					
Acosta, Marc					
⋮					
Adams, John					
Adams, Robin					
⋮					
Akers, Jan					
⋮					
Alexander, Ed					
Alfred, Bob					
⋮					
Allen, Sam					
⋮					
Allen, Troy					
Anders, Keith					
⋮					
Anderson, Rob					
⋮					
Anderson, Zach					
Angel, Joe					
⋮					
Archer, Sue					
⋮					
Arnold, Mack					
Arnold, Steven					
⋮					
Atkins, Timothy					
⋮					

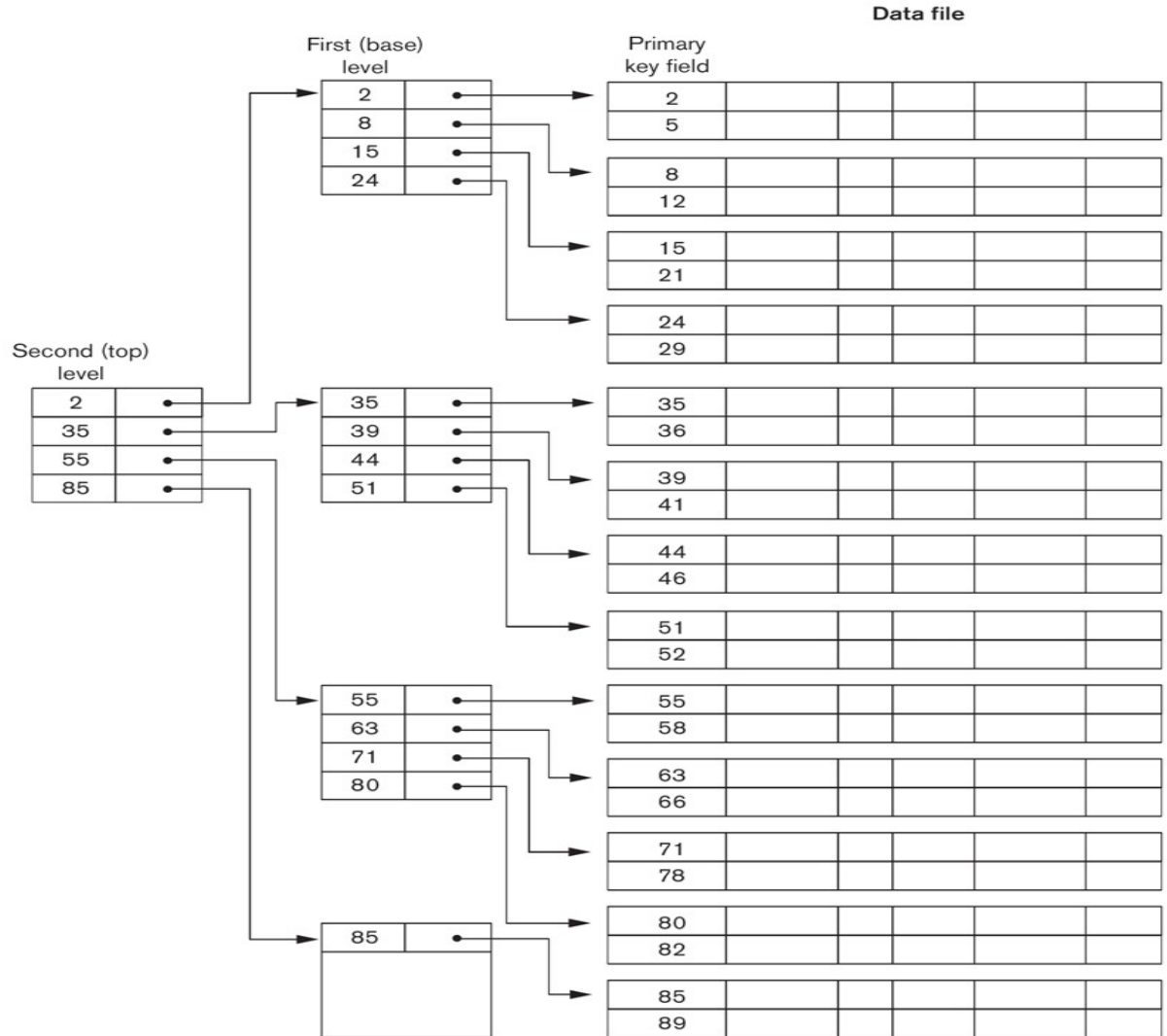
Organização indexada multiníveis

Busca: $O(\log_{fbi} r^1)$
Inserção/Remoção: $O(bi)$
(para todos os blocos de índices...)



O que fazer???

Como ter uma busca eficiente mas
permitir uma inserção e remoção
razoável?



Lembrando de AED 1...

- Busca eficiente em dados ordenados sem gastar memória:

Lembrando de AED 1...

- Busca eficiente em dados ordenados sem gastar memória:
 - Busca binária (em um vetor)
 - Mas o problema era justamente inserção / deleção
 - Qual era a alternativa de **continuar fazendo busca binária** mas permitir dinamismo de inserção / remoção?

Lembrando de AED 1...

- Busca eficiente em dados ordenados sem gastar memória:
 - Busca binária (em um vetor)
 - Mas o problema era justamente inserção / deleção
 - Qual era a alternativa de **continuar fazendo busca binária** mas permitir dinamismo de inserção / remoção?
- Árvores Binárias de Busca!!!!

Lembrando de AED 1...

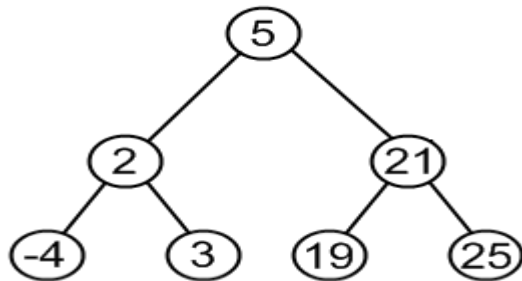
- Busca binária (em um vetor):
-4, 2, 3, 5, 19, 21, 25

Lembrando de AED 1...

- Busca binária (em um vetor):
-4, 2, 3, 5, 19, 21, 25
- Árvores Binárias de Busca:

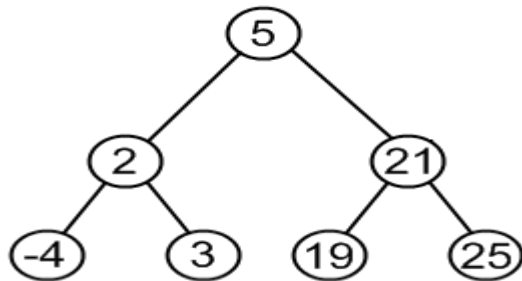
Lembrando de AED 1...

- Busca binária (em um vetor):
-4, 2, 3, 5, 19, 21, 25
- Árvores Binárias de Busca:



Lembrando de AED 1...

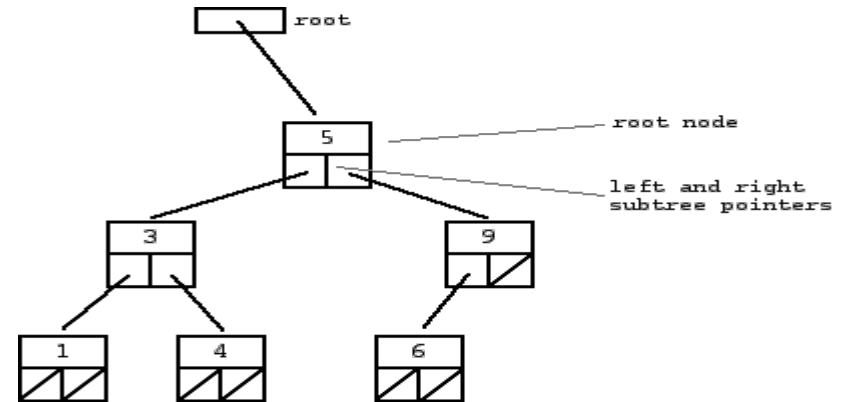
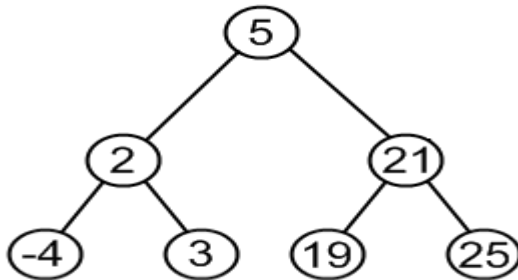
- Busca binária (em um vetor):
-4, 2, 3, 5, 19, 21, 25
- Árvores Binárias de Busca:



IMPLEMENTAÇÃO ?

Lembrando de AED 1...

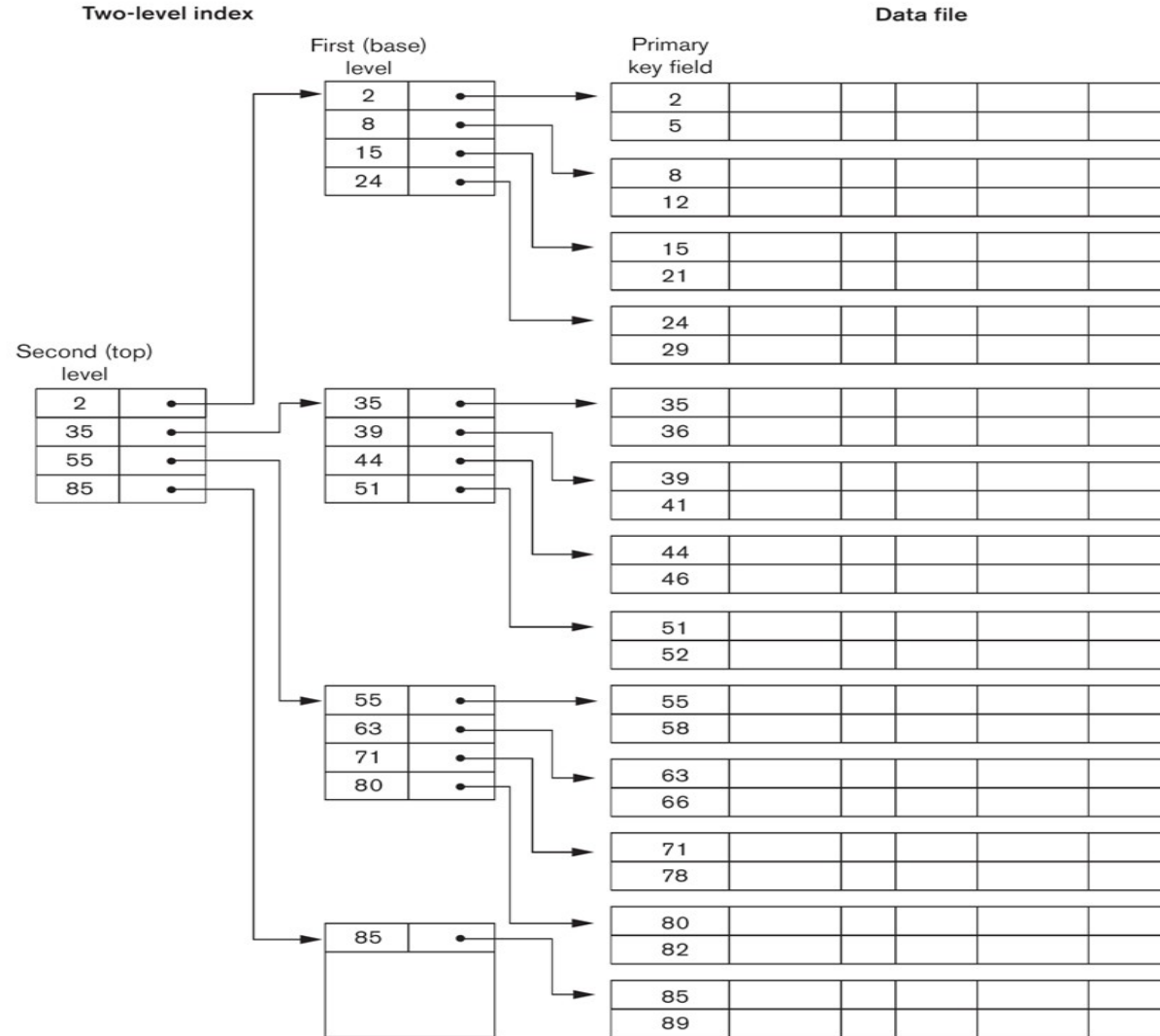
- Busca binária (em um vetor):
-4, 2, 3, 5, 19, 21, 25
- Árvores Binárias de Busca:



Podemos
pensar em algo
semelhante para
melhorar o
dinamismo dos
índices
múltiníveis?

Figure 18.6

A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

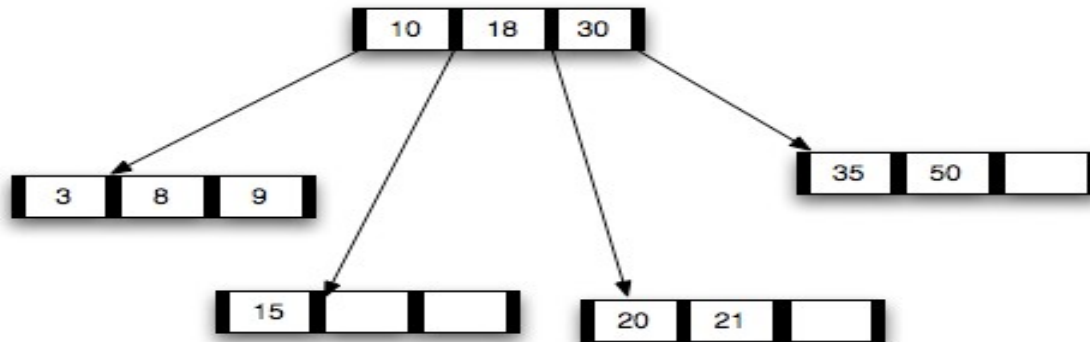


Podemos pensar em algo semelhante para melhorar o dinamismo dos índices múltiplos?

- Árvores de busca $n+1$ -árias!
- N = nr de registros representados em um nó da árvore (bloco), cada registro com uma chave k_i
- $N+1$ ponteiros para nós filhos contendo registros com chaves em cada intervalo
O segredo será mantê-las balanceadas!

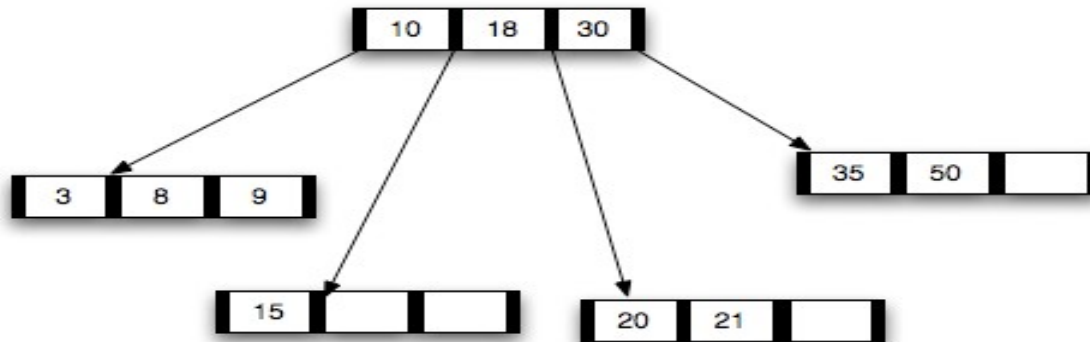
ÁRVORES B !!!

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



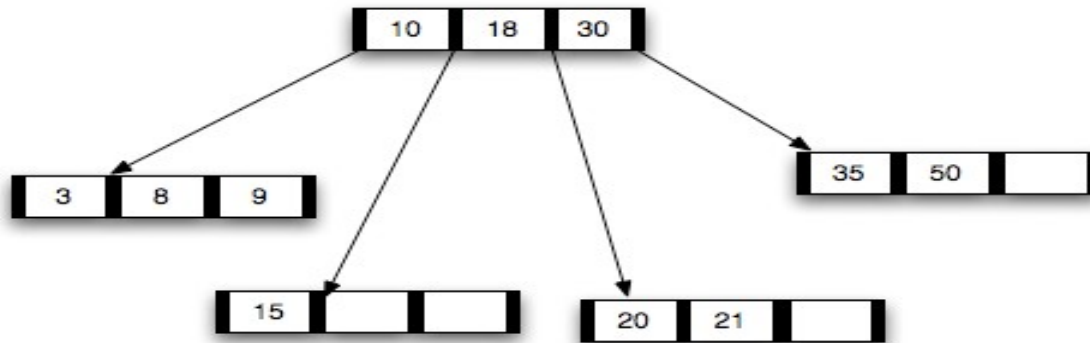
Árvore B Clássica

- Vamos começar estudando a árvore B clássica, depois fica fácil adaptar para a árvore B+



Árvore B Clássica

- Como deve ser a estrutura de dados para essa árvore?

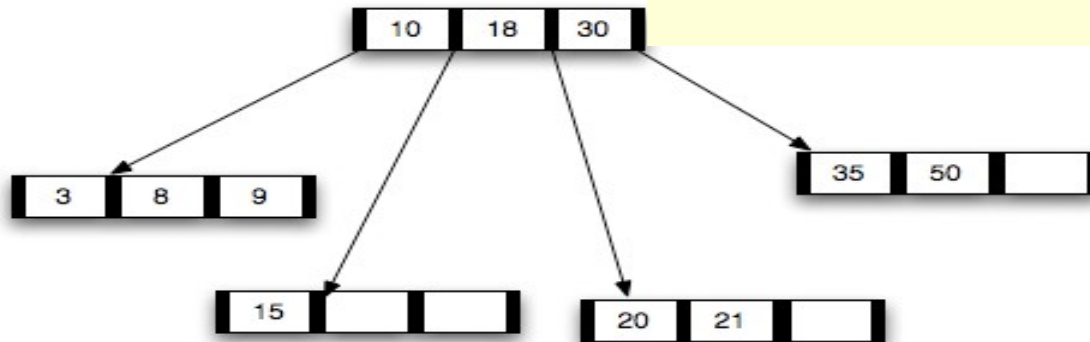


Árvore B Clássica

- Como deve ser a estrutura de dados para essa árvore?

Lembrando que, para simplificar, estamos colocando só a chave do registro

```
typedef int TipoChave;  
typedef struct str_no {  
    TipoChave chave[MAX_CHAVES];  
    struct str_no* filho[MAX_CHAVES+1];  
    int numChaves;  
    bool folha;  
} NO;
```

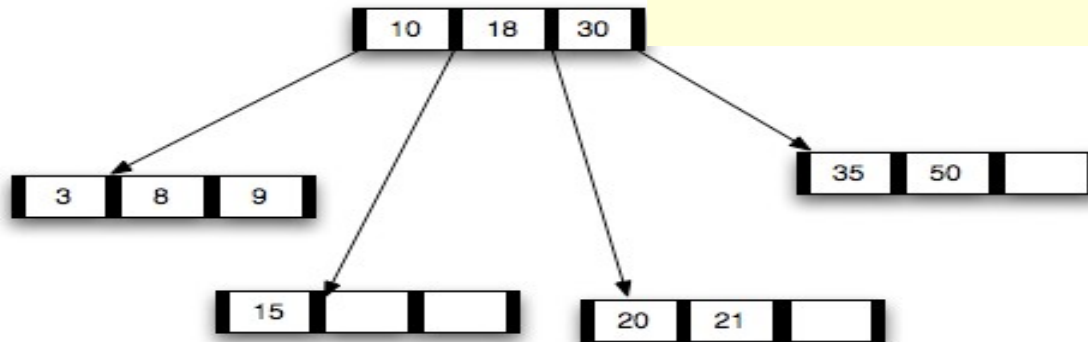


Árvore B Clássica

- Como deve ser a estrutura de dados para essa árvore?

PROVISÓRIO!!! →

```
typedef int TipoChave;  
typedef struct str_no {  
    TipoChave chave[MAX_CHAVES];  
    struct str_no* filho[MAX_CHAVES+1];  
    int numChaves;  
    bool folha;  
} NO;
```



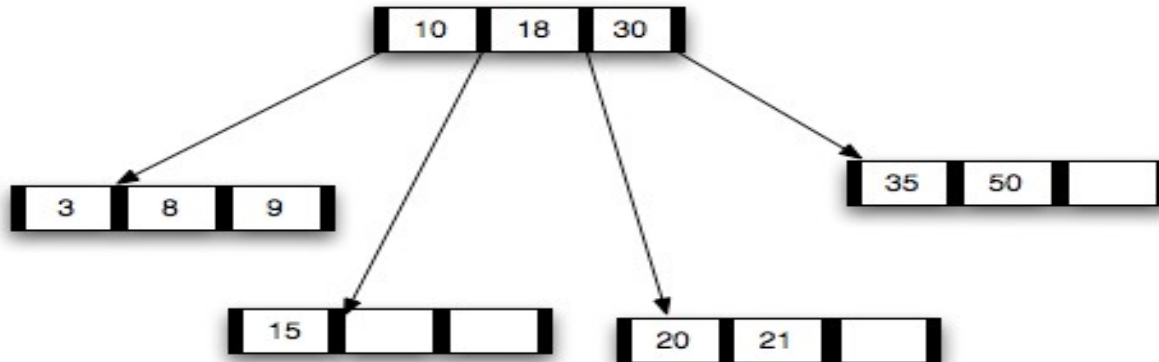
Árvore B – Definição (notação usada pelo Cormen)

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

Vamos usar essa notação nas aulas!!!

1. Cada nó x contém os seguintes campos:

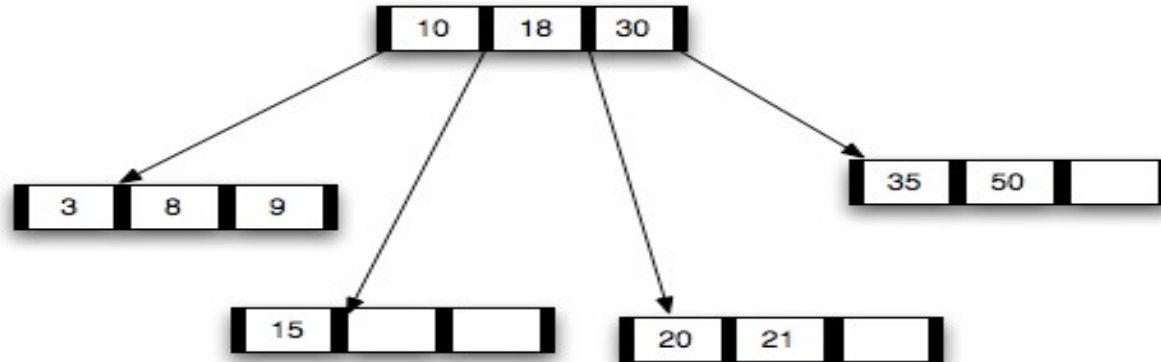
- $n[x]$, o número de chaves atualmente armazenadas no nó x ;
- as $n[x]$ chaves, armazenadas em ordem não decrescente, de modo que $key_1[x] \leq key_2[x] \leq \dots \leq key_{n[x]}[x]$;
- $leaf[x]$, um valor booleano indicando se x é uma folha (TRUE) ou um nó interno (FALSE).
- se x é um nó interno,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



Árvore B - Definição

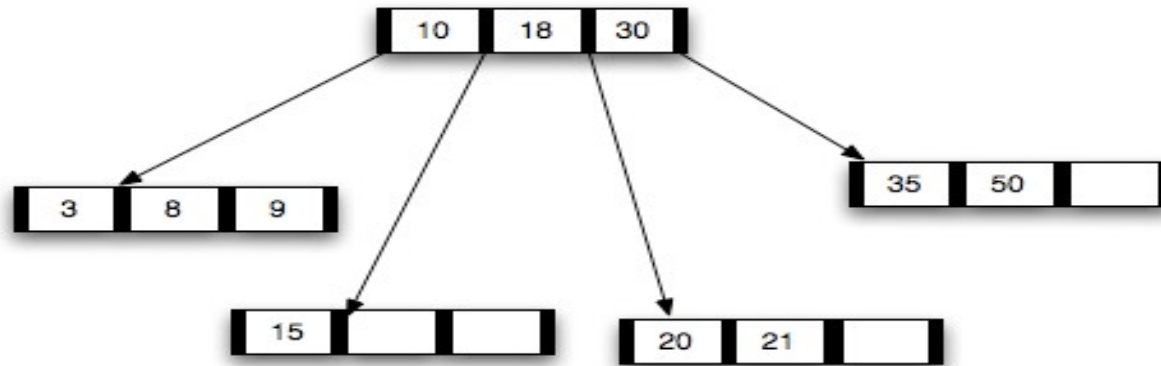
2. As chaves $key_i[x]$ separam as faixas de valores armazenados em cada subárvore: denotando por k_i uma chave qualquer armazenada na subárvore com nó $c_i[x]$, tem-se

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



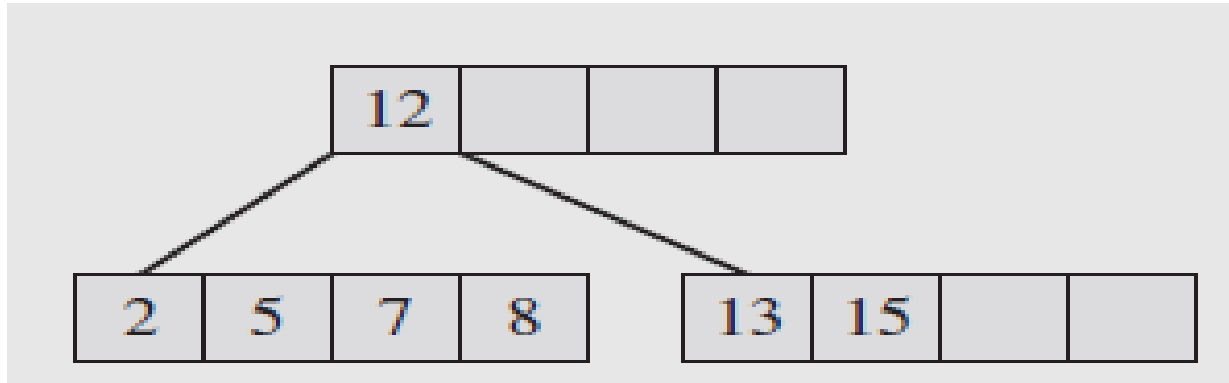
Árvore B - Definição

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



Árvore B - Definição

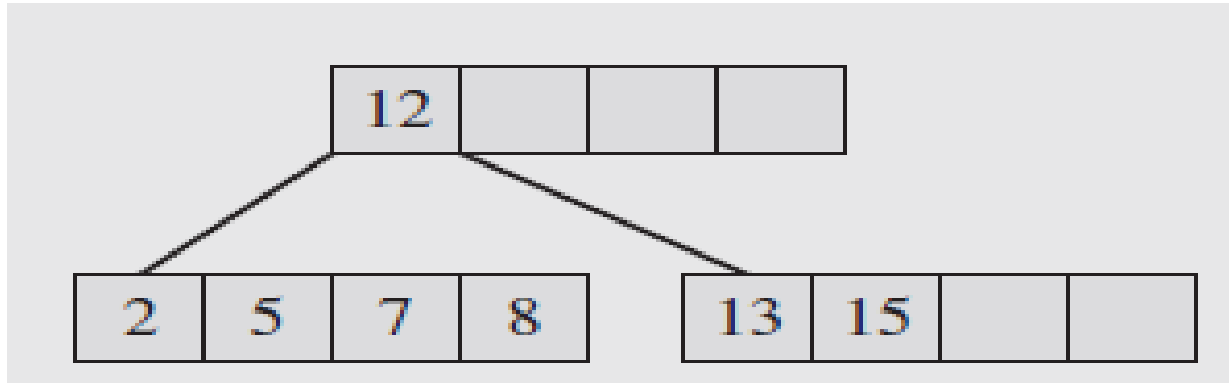
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h .



OBS: E se eu precisasse inserir o valor 1 ?

Árvore B - Definição

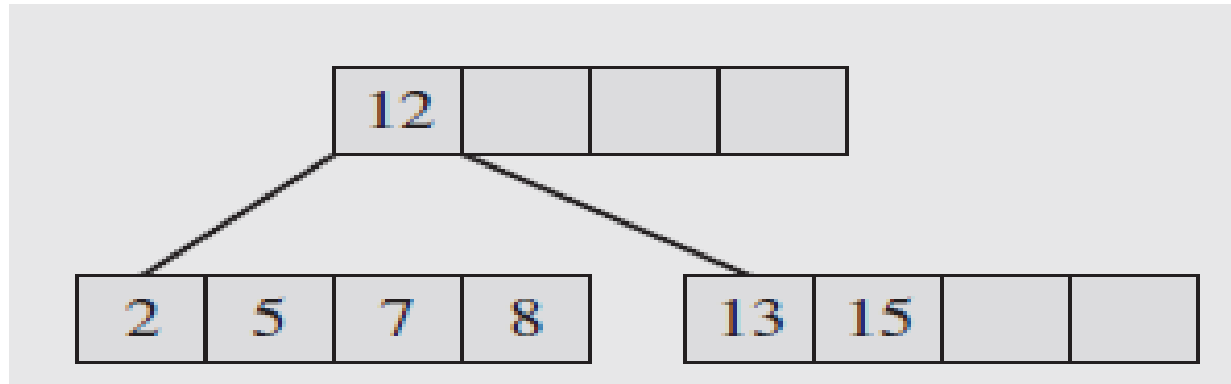
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h .



OBS: E se eu precisasse inserir o valor 1 ? Criaria um filho à esquerda de 2?

Árvore B - Definição

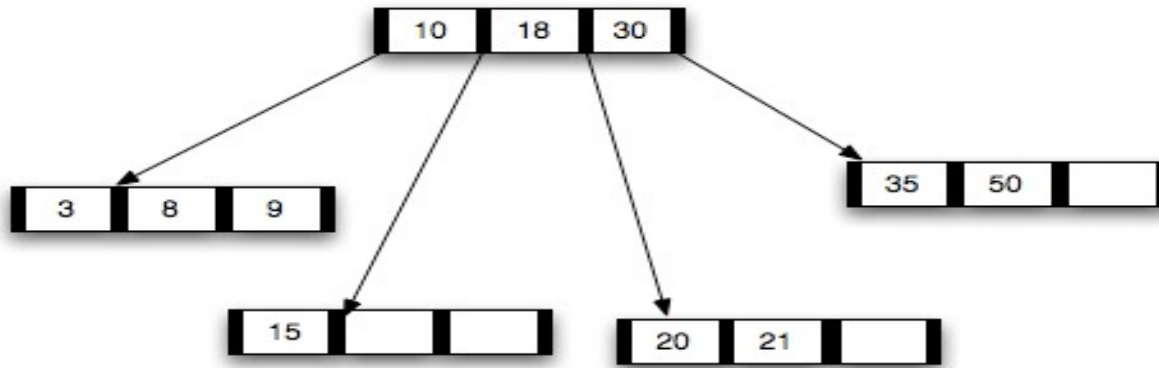
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h .



OBS: E se eu precisasse inserir o valor 1 ? Criaria um filho à esquerda de 2?
1-) Aumentaria a altura da árvore mesmo ela tendo espaço
2-) Feriria a definição 3...

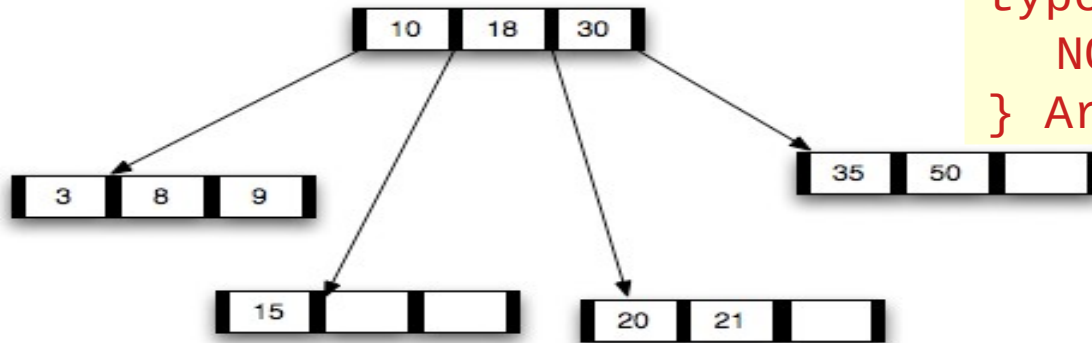
Árvore B - Definição

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



Estrutura de uma árvore B

Final

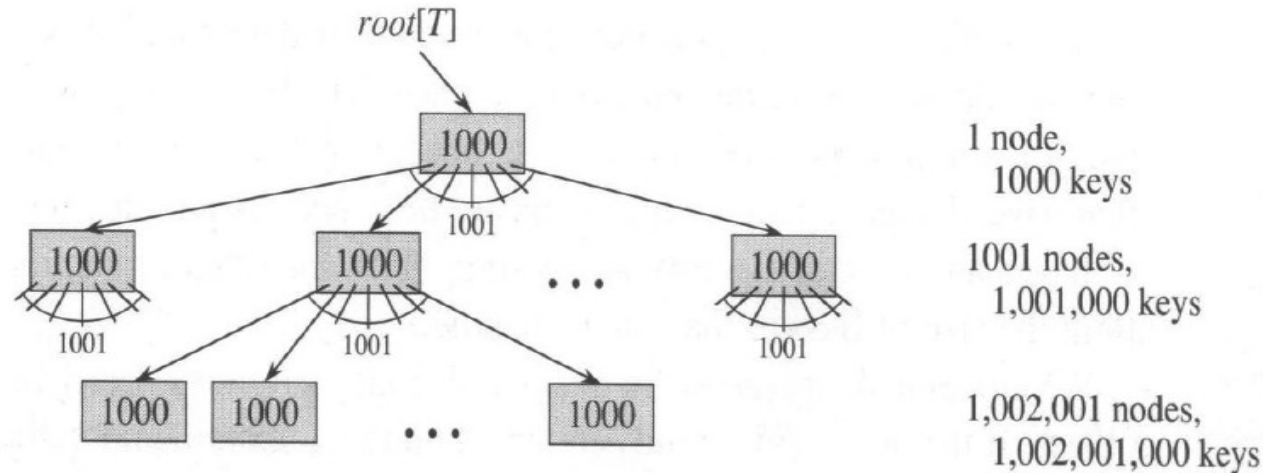


```
#define t 2 /* grau min da árvore */
typedef int TipoChave;
typedef struct str_no {
    TipoChave chave[2*t-1];
    struct str_no* filhos[2*t];
    int numChaves;
    bool folha;
} NO;
typedef struct {
    NO* raiz;
} ArvB;
```

Árvore B - 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.



Árvore B – altura máxima

- **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 contém 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,

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

Operações Básicas em Árvores B

Criação de uma árvore B vazia

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;
```

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)

```
1  $x \leftarrow \text{ALLOCATE-NODE}()$ 
2  $leaf[x] \leftarrow \text{TRUE}$ 
3  $n[x] \leftarrow 0$ 
4  $\text{DISK-WRITE}(x)$ 
5  $root[T] \leftarrow x$ 
```

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)

```
1  $x \leftarrow \text{ALLOCATE-NODE}()$ 
2  $leaf[x] \leftarrow \text{TRUE}$ 
3  $n[x] \leftarrow 0$ 
4  $\text{DISK-WRITE}(x)$ 
5  $root[T] \leftarrow x$ 
```

```
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;
}
```

OBS 1: sobre DISK-WRITE e DISK-READ

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;
```

```
void escreveNoDisco(NO* x){
    /* código específico */
}
```

NO* é na verdade
o número do bloco
do disco

B-TREE-CREATE(T)

```
1   $x \leftarrow \text{ALLOCATE-NODE}()$ 
2   $leaf[x] \leftarrow \text{TRUE}$ 
3   $n[x] \leftarrow 0$ 
4   $\text{DISK-WRITE}(x)$ 
5   $root[T] \leftarrow x$ 
```

```
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;
}
```

Na verdade precisa saber onde escrever. Aqui tem que alocar um novo bloco no disco (ex: 350) e efetuar um `seek(350*4Kb) + write(x)` (assumindo tam do bloco = 4Kb)

OBS 2: quando na prática, para economizar espaço (evitar vetor filhos nas folhas)

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;
```

```
bool alocaNo(bool folha){
    ...
}
```

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

OBS 2: quando na prática, para economizar espaço (evitar vetor filhos nas folhas)

Criação de uma árvore B vazia

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

B-TREE-CREATE(T)

```
1  $x \leftarrow \text{ALLOCATE-NODE}()$ 
2  $leaf[x] \leftarrow \text{TRUE}$ 
3  $n[x] \leftarrow 0$ 
4  $\text{DISK-WRITE}(x)$ 
5  $root[T] \leftarrow x$ 
```

```
bool alocaNo(bool folha){
    NO* x;
    if(!(x = (NO*) malloc(sizeof(NO)))) {
        /* msg erro e retorna false */
    }
    if(!(x->chave = (TipoChave*) calloc(2*t-1, sizeof(TipoChave))))
    { /* msg erro e retorna false */
    if(!folha){
        if(!(x->filhos = (NO**) calloc(2*t, sizeof(NO*))))
        { /* msg erro e retorna false */
        } else x->filhos = NULL;
    }
}
```

OBS 2: mas vamos considerar essa estrutura na disciplina

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;
```

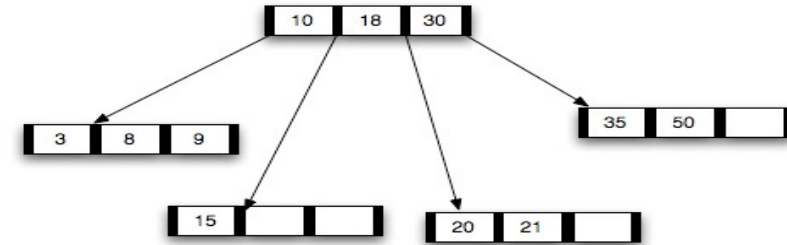
```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

```
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;
}
```

Busca na árvore B

- $B\text{-Tree-Search}(x, k)$: 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 .

Ex: $B\text{-Tree-Search}(T \rightarrow \text{raiz}, 22)$



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

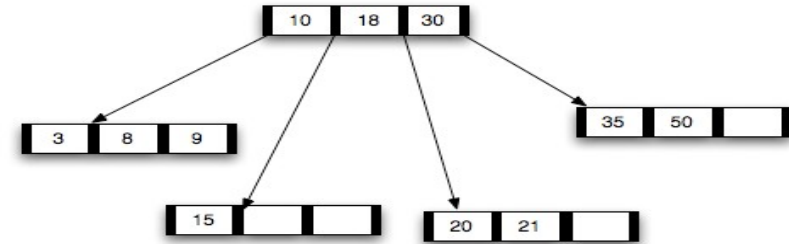
Busca na árvore B

- 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($T \rightarrow$ raiz, 22)

B-TREE-SEARCH(x, k)

```
1   $i \leftarrow 1$   
2  while  $i \leq n[x]$  and  $k > key_i[x]$   
3      do  $i \leftarrow i + 1$ 
```



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

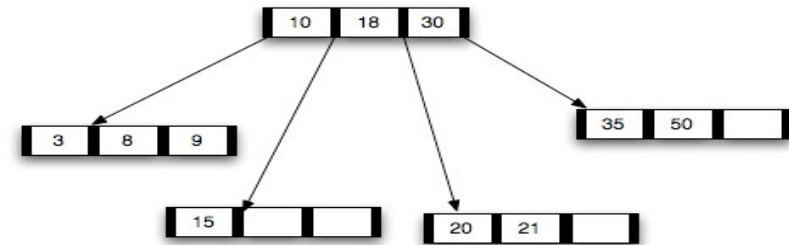
Busca na árvore B

- 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($T \rightarrow$ raiz, 22)

B-TREE-SEARCH(x, k)

```
1   $i \leftarrow 1$   
2  while  $i \leq n[x]$  and  $k > key_i[x]$   
3      do  $i \leftarrow i + 1$   
4  if  $i \leq n[x]$  and  $k = key_i[x]$ 
```



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

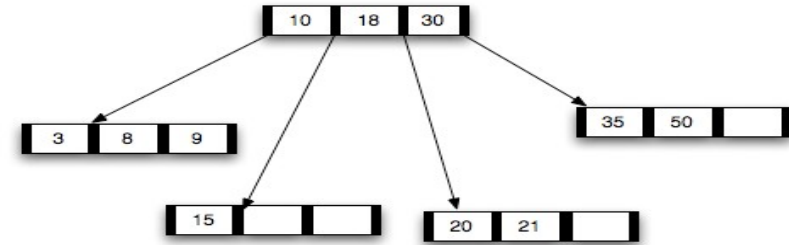
Busca na árvore B

- 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($T \rightarrow$ raiz, 22)

B-TREE-SEARCH(x, k)

```
1   $i \leftarrow 1$ 
2  while  $i \leq n[x]$  and  $k > key_i[x]$ 
3      do  $i \leftarrow i + 1$ 
4  if  $i \leq n[x]$  and  $k = key_i[x]$ 
5      then return  $(x, i)$ 
```



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

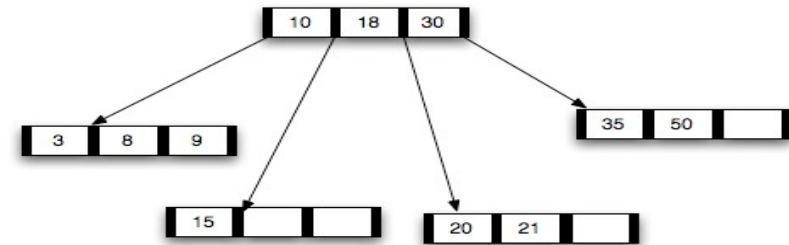
Busca na árvore B

- $B\text{-Tree-Search}(x, k)$: 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: $B\text{-Tree-Search}(root[T], k)$.

Ex: $B\text{-Tree-Search}(T \rightarrow \text{raiz}, 22)$

B-TREE-SEARCH(x, k)

```
1   $i \leftarrow 1$ 
2  while  $i \leq n[x]$  and  $k > key_i[x]$ 
3      do  $i \leftarrow i + 1$ 
4  if  $i \leq n[x]$  and  $k = key_i[x]$ 
5      then return  $(x, i)$ 
6  if  $leaf[x]$ 
```



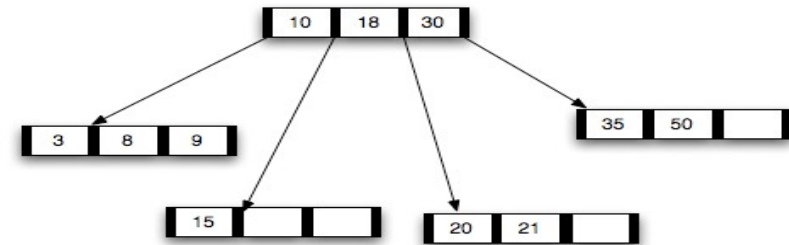
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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($T \rightarrow$ raiz, 22)

```
B-TREE-SEARCH( $x, k$ )
1   $i \leftarrow 1$ 
2  while  $i \leq n[x]$  and  $k > key_i[x]$ 
3      do  $i \leftarrow i + 1$ 
4  if  $i \leq n[x]$  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( $c_i[x], k$ )
```



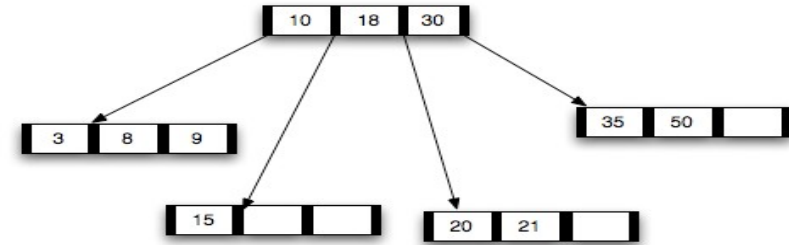
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

Busca na árvore B

- $B\text{-Tree-Search}(x, k)$: 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: $B\text{-Tree-Search}(root[T], k)$.

Ex: $B\text{-Tree-Search}(T \rightarrow \text{raiz}, 22)$

```
B-TREE-SEARCH( $x, k$ )
1   $i \leftarrow 1$ 
2  while  $i \leq n[x]$  and  $k > key_i[x]$ 
3    do  $i \leftarrow i + 1$ 
4  if  $i \leq n[x]$  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( $c_i[x], k$ )
```



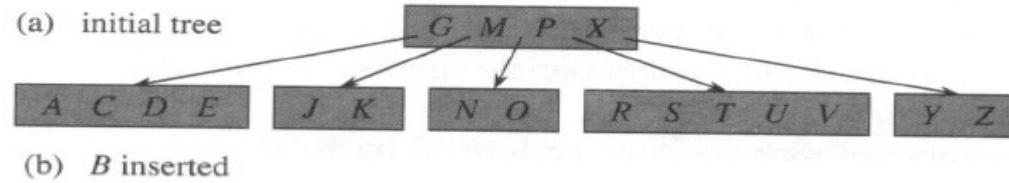
Na verdade precisa saber de onde ler (ex bloco 981) e efetuar um $seek(981 * 4Kb) + read(4kb)$ (assumindo tam do bloco = 4Kb)

Na disciplina vamos abstrair isso também.

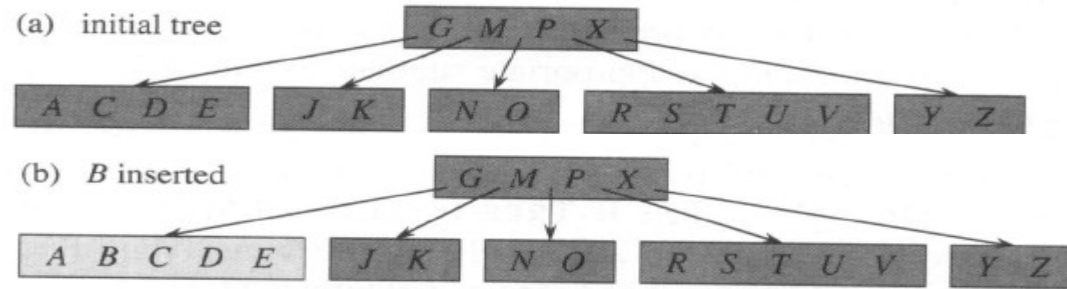
Inserção em árvore B

As inserções ocorrem sempre nas folhas

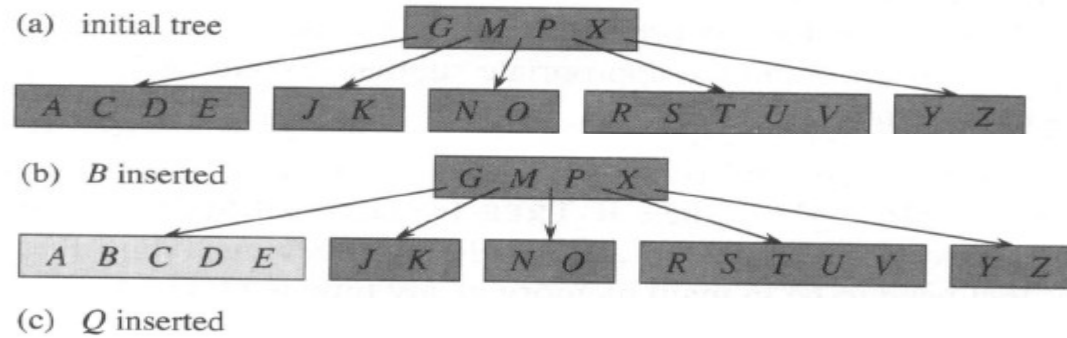
Exemplo Inserção $t = 3$



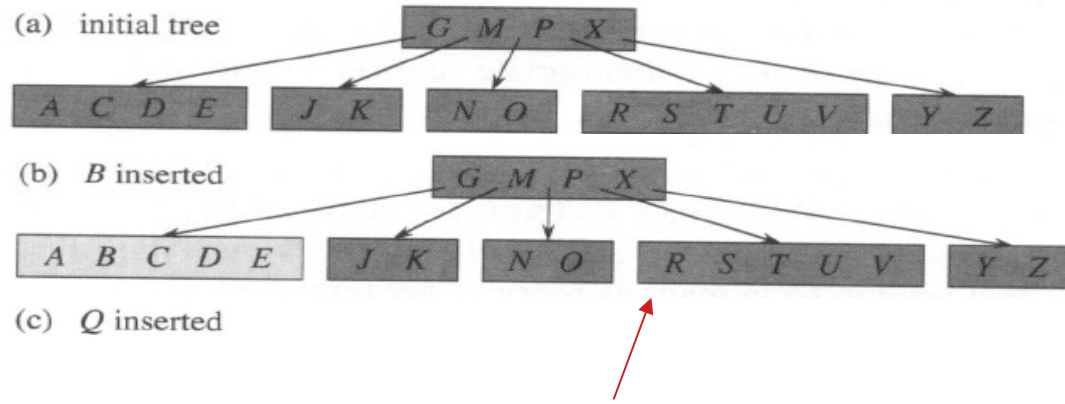
Exemplo Inserção $t = 3$



Exemplo Inserção $t = 3$

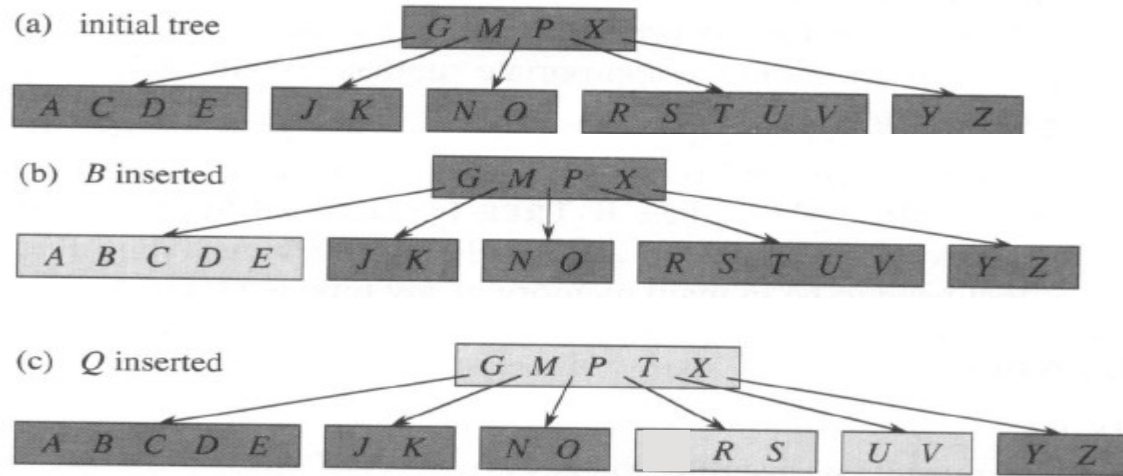


Exemplo Inserção $t = 3$



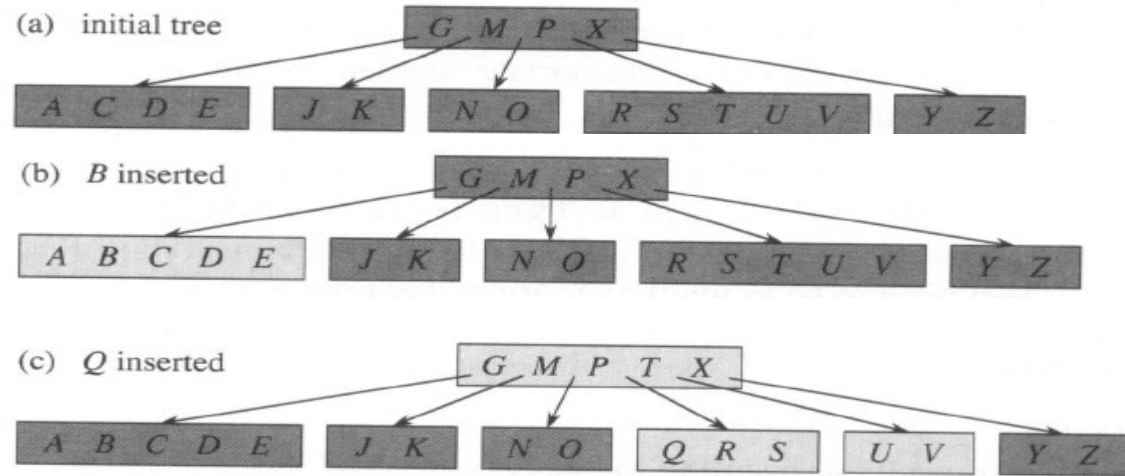
Seria aqui, mas este nó está cheio...
O que fazer?

Exemplo Inserção $t = 3$



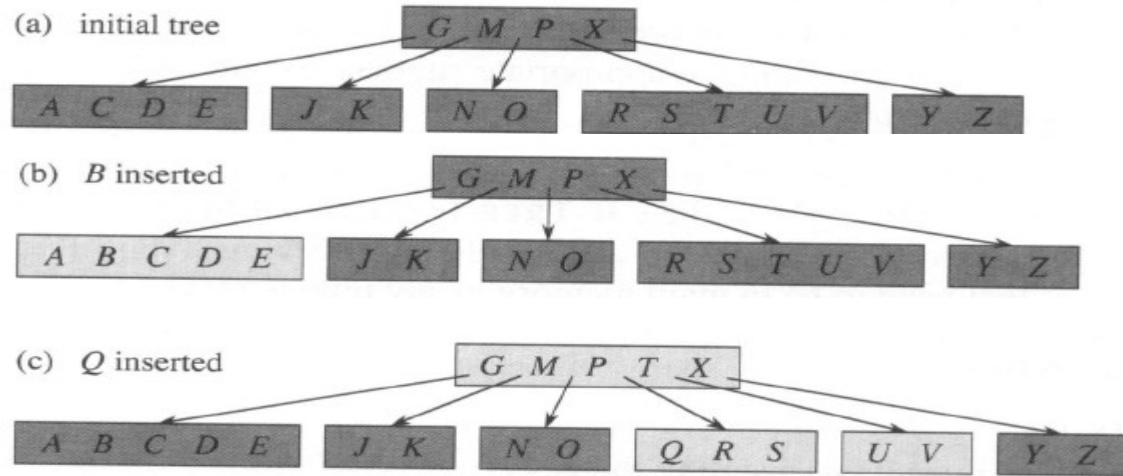
Primeiro quebro o nó em dois
(envolve a subida da chave do
meio)

Exemplo Inserção $t = 3$



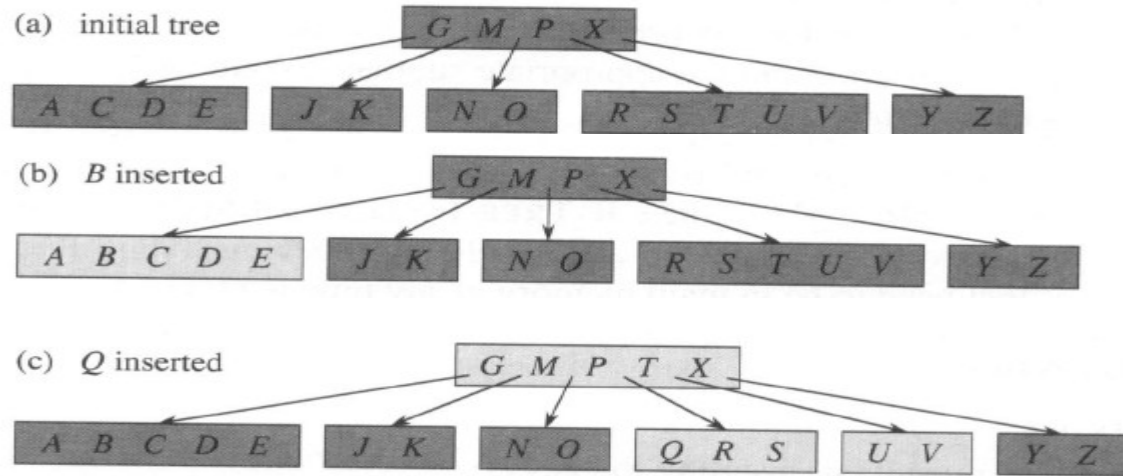
Insiro a chave

Exemplo Inserção $t = 3$



E se quisesse inserir F ?

Exemplo Inserção $t = 3$

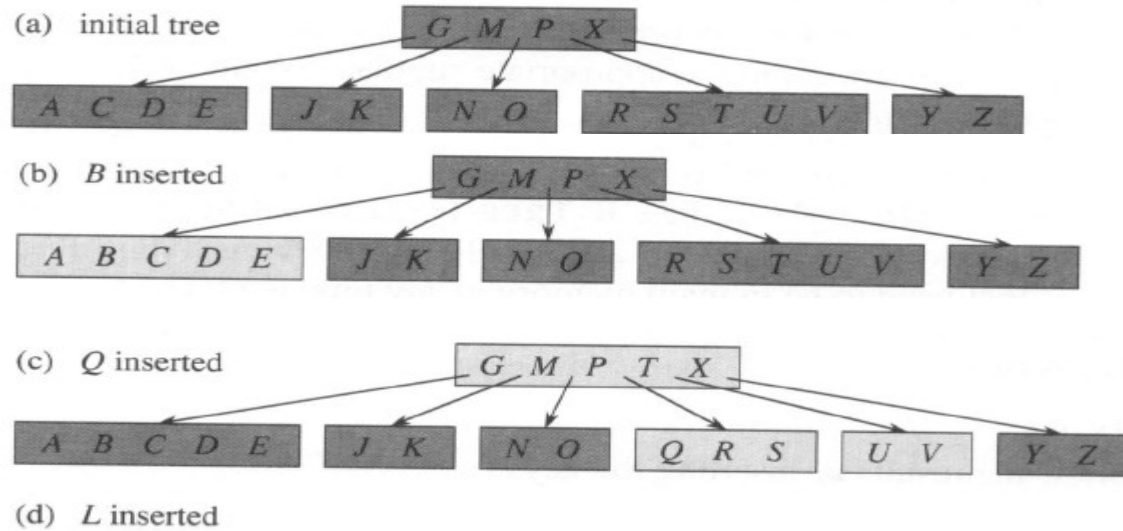


E se quisesse inserir F ?

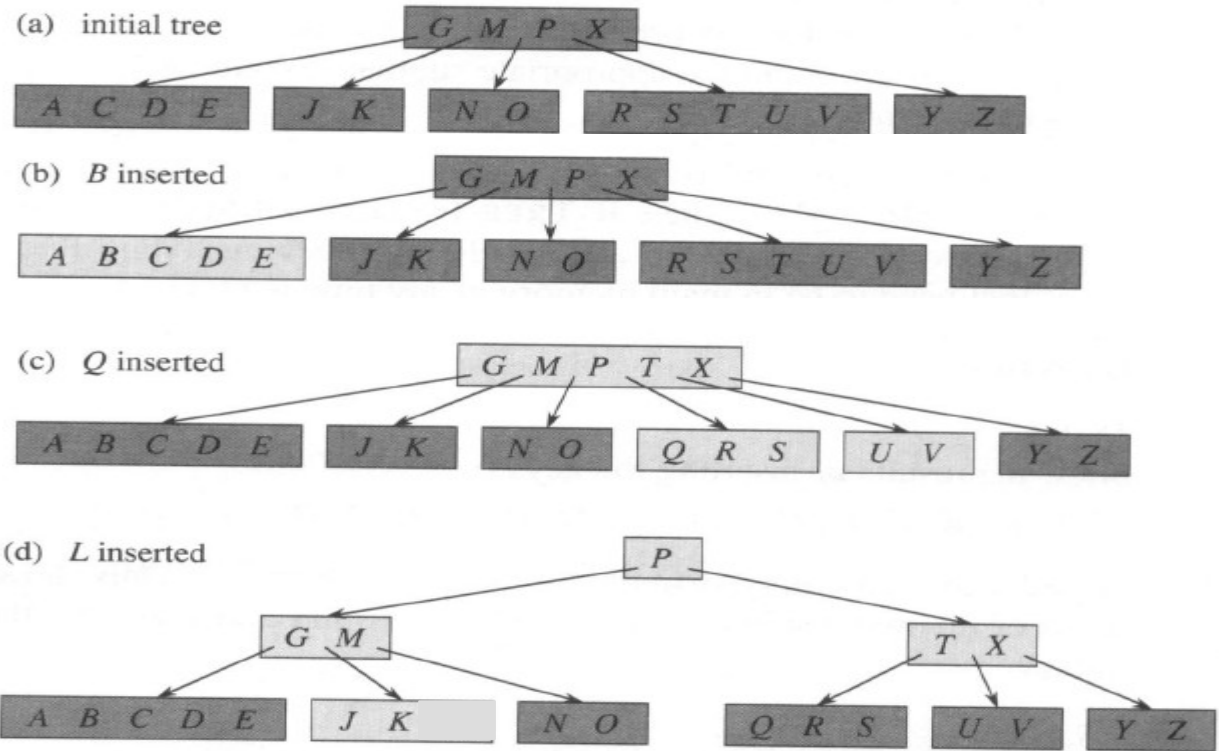
O nó pai estaria cheio, mas não tenho ponteiro para subir para ele para quebrá-lo em dois...

Então, sempre que descemos na árvore para achar onde inserir, se encontro um nó cheio já quebro

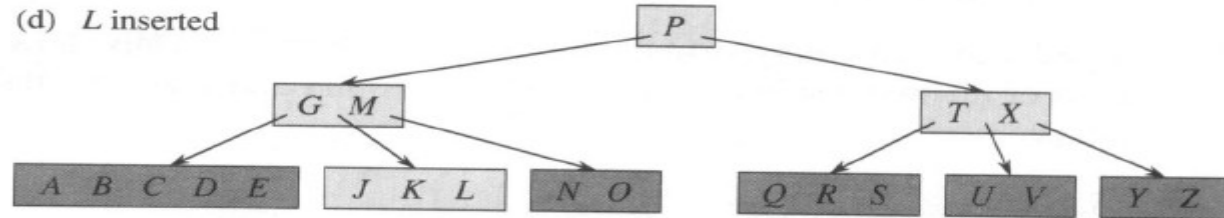
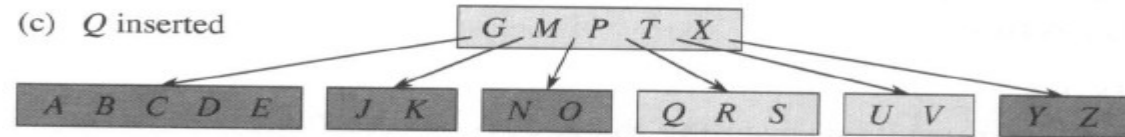
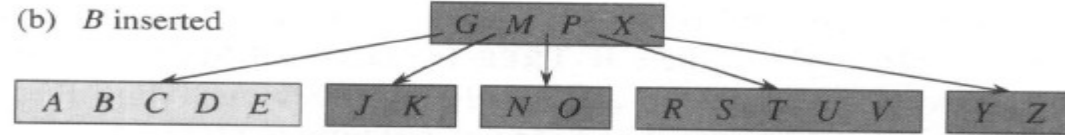
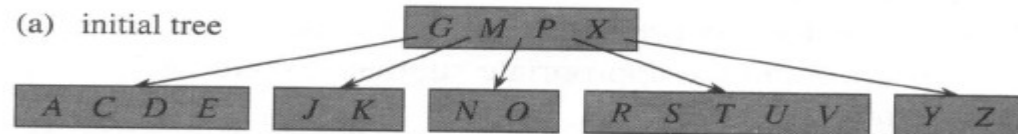
Exemplo Inserção $t = 3$



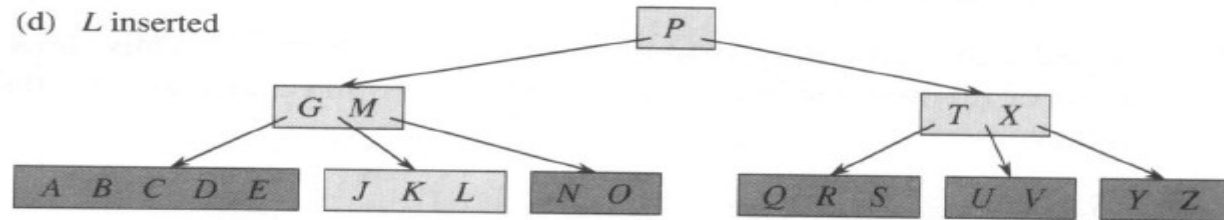
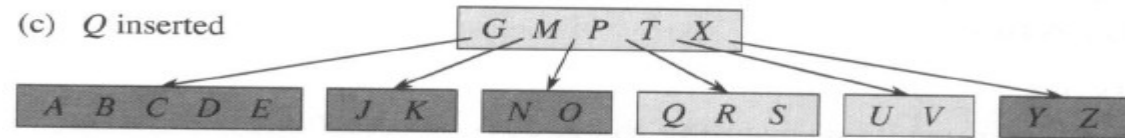
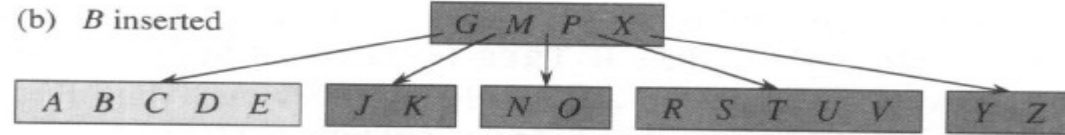
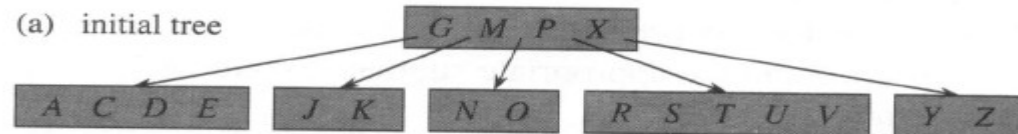
Exemplo Inserção $t = 3$



Exemplo Inserção $t = 3$

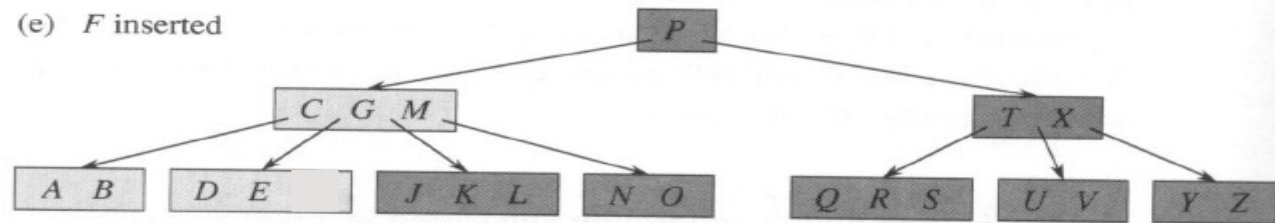
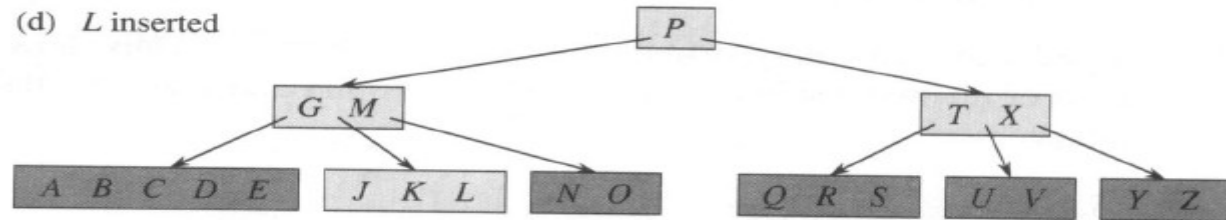
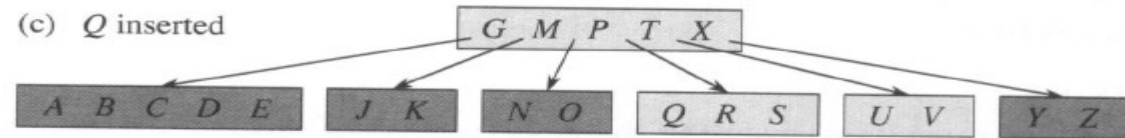
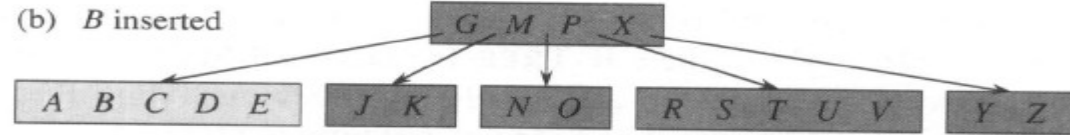
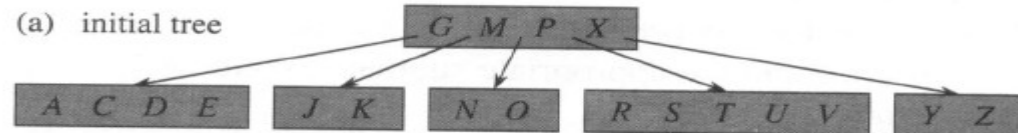


Exemplo Inserção $t = 3$

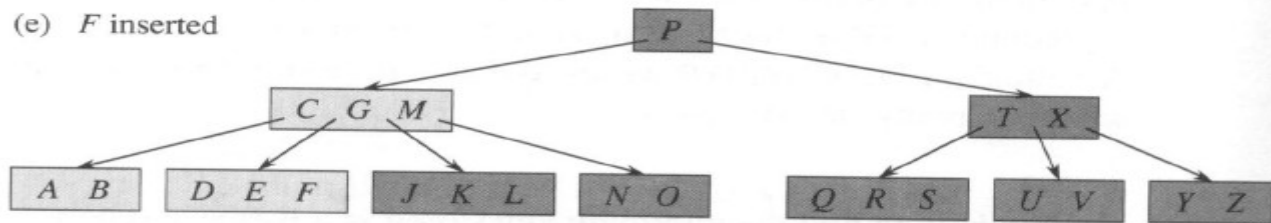
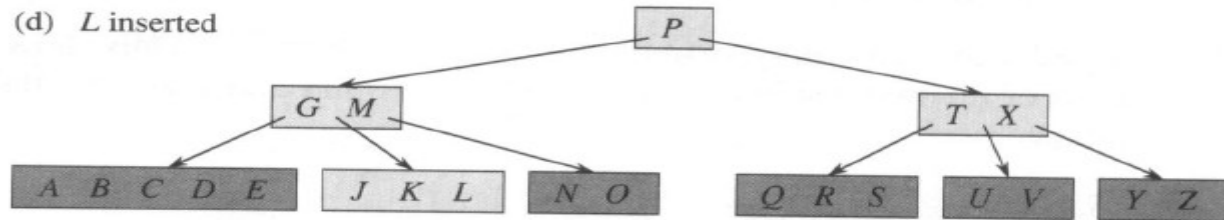
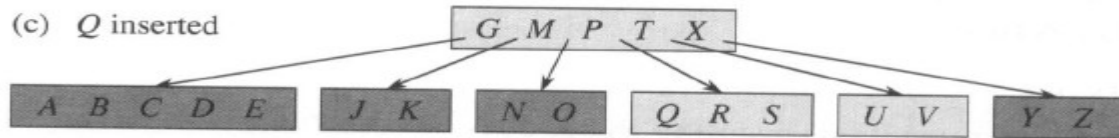
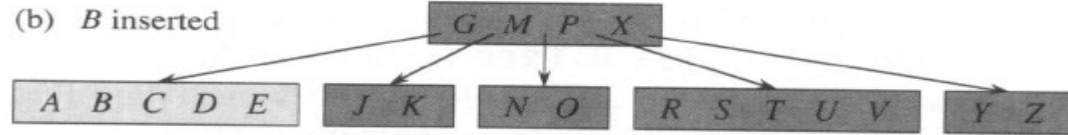
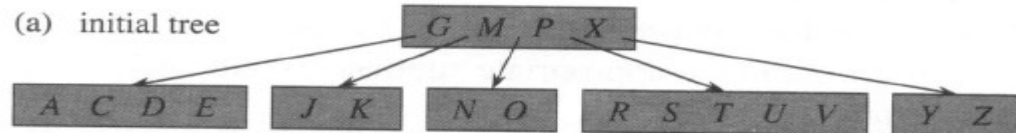


(e) F inserted

Exemplo Inserção $t = 3$

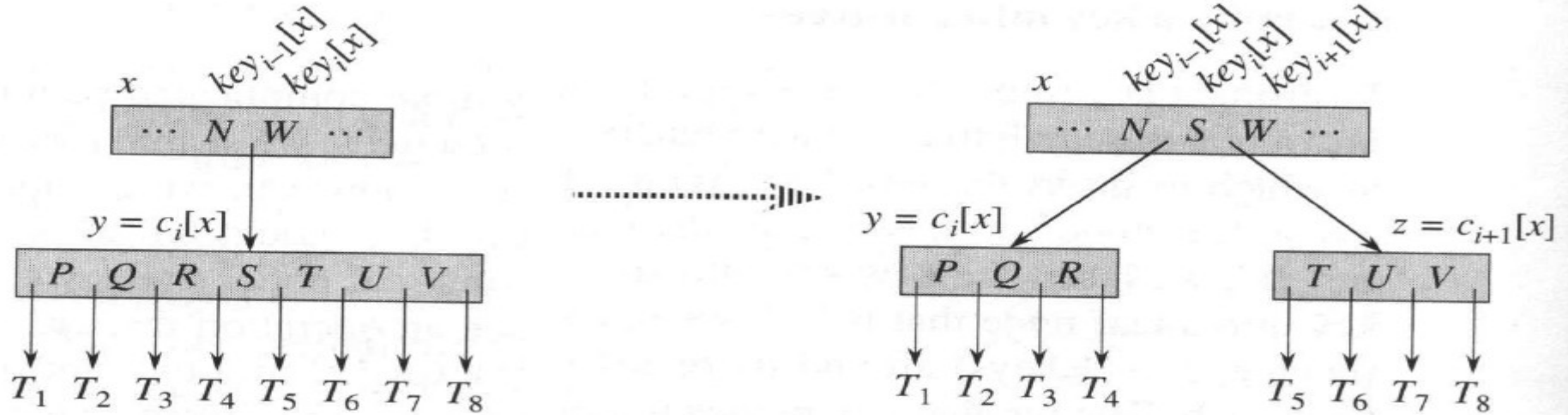


Exemplo Inserção $t = 3$



As inserções ocorrem sempre nas folhas

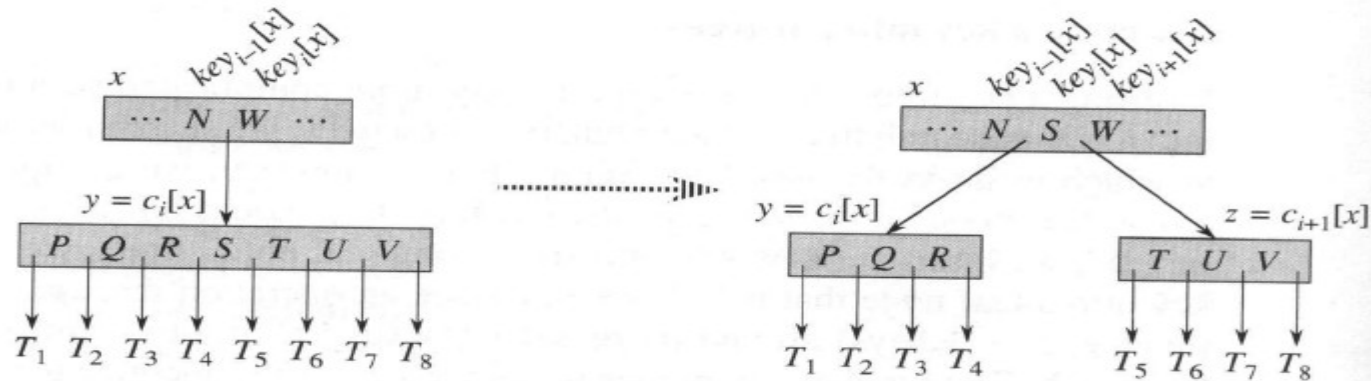
Resolvendo o problema do nó cheio...



- Divisão de um nó na árvore:

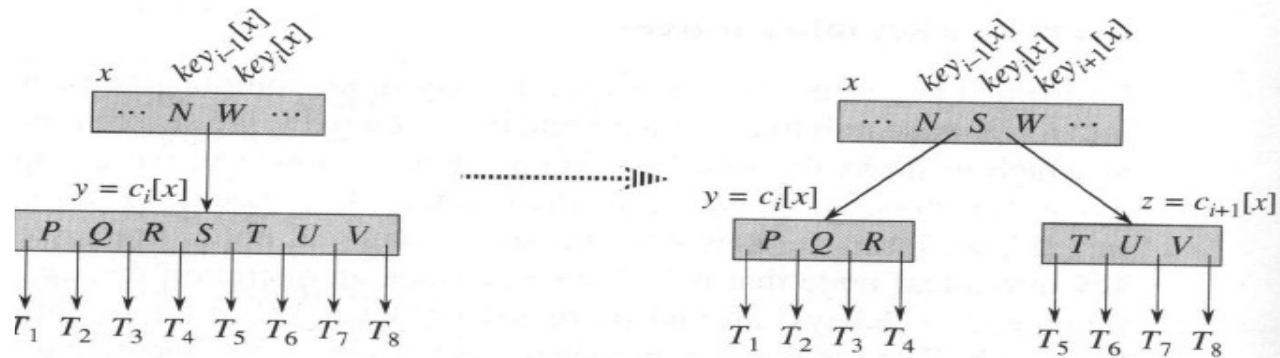
Porque se ele estava cheio já foi dividido

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?



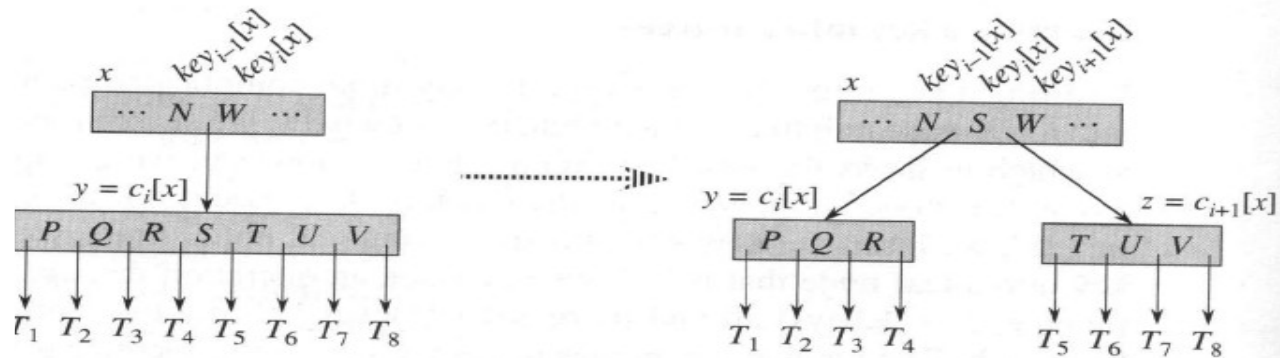
B-TREE-SPLIT-CHILD(x, i, y)

O que precisa fazer?

Aloca e inicializa z

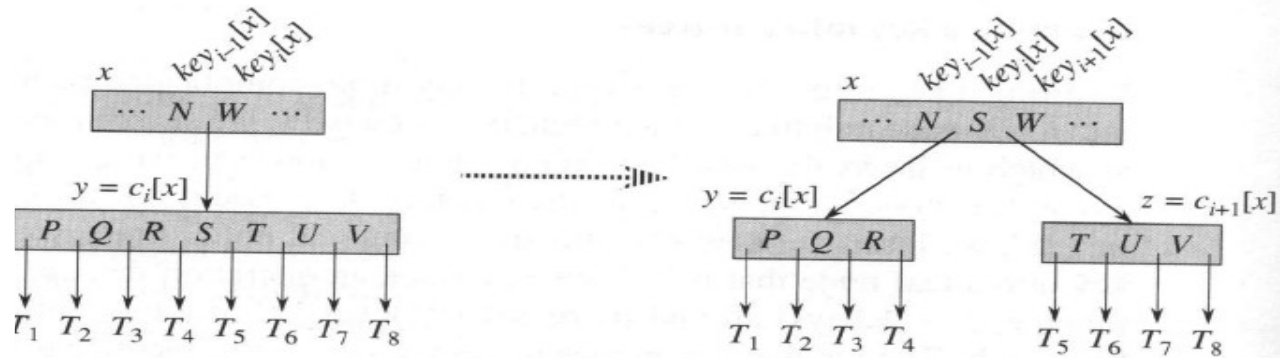
Ajusta y

Ajusta x



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

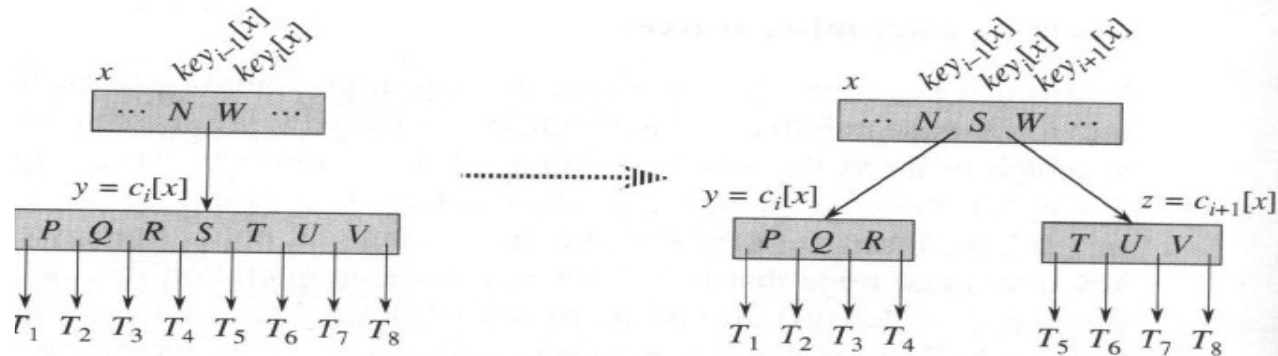
Aloca e inicializa z



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

```
1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
```

Aloca e inicializa z

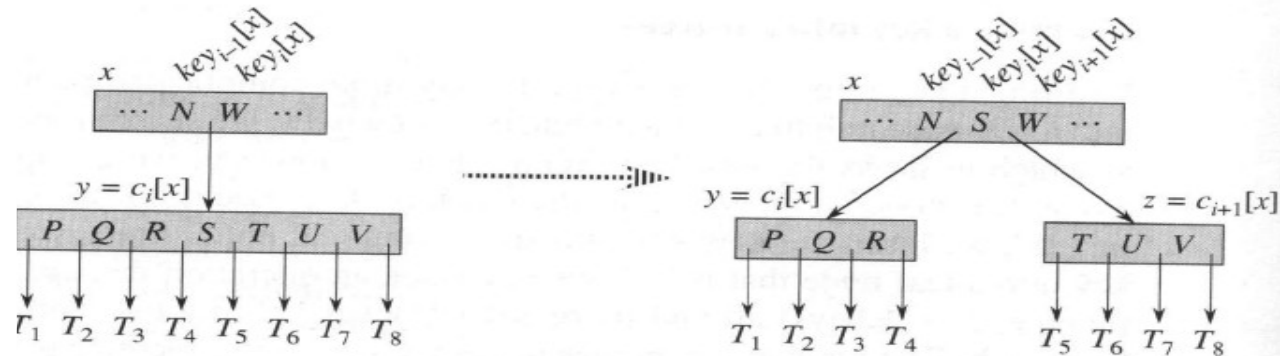


B-TREE-SPLIT-CHILD(x, i, y)

```
1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
```

Aloca e inicializa z

Ajusta y



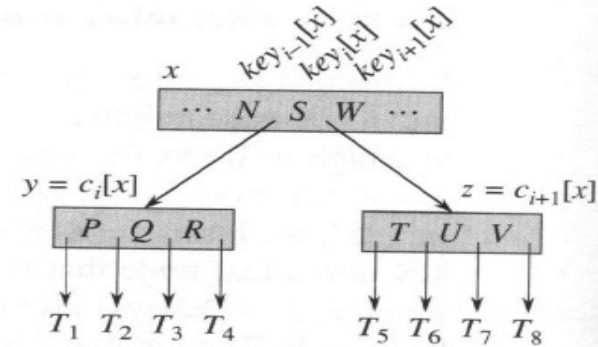
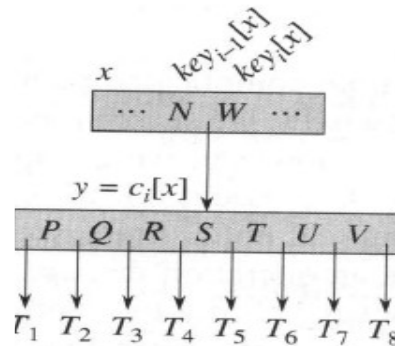
B-TREE-SPLIT-CHILD(x, i, y)

```

1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9   $n[y] \leftarrow t - 1$ 
    
```

Aloca e inicializa z

Ajusta y



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

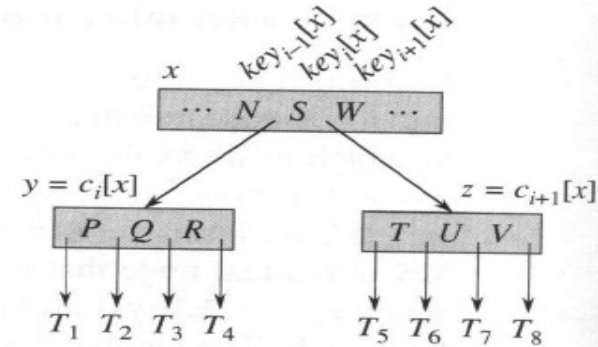
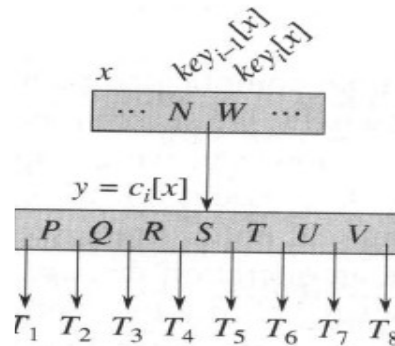
```

1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9   $n[y] \leftarrow t - 1$ 
    
```

Aloca e inicializa z

Ajusta y

Ajusta x



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

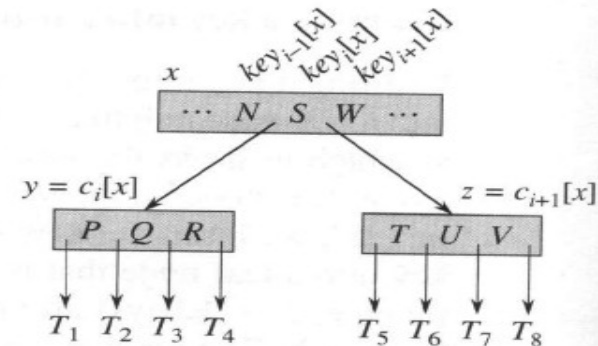
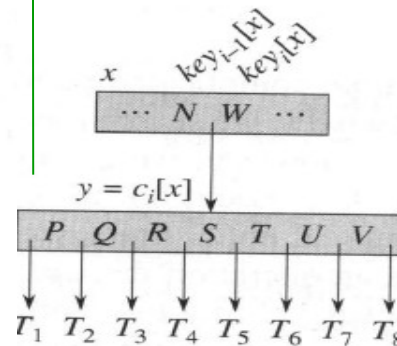
```

1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9   $n[y] \leftarrow t - 1$ 
10 for  $j \leftarrow n[x] + 1$  downto  $i + 1$ 
11     do  $c_{j+1}[x] \leftarrow c_j[x]$ 
12  $c_{i+1}[x] \leftarrow z$ 
13 for  $j \leftarrow n[x]$  downto  $i$ 
14     do  $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$ 
15  $\text{key}_i[x] \leftarrow \text{key}_i[y]$ 
16  $n[x] \leftarrow n[x] + 1$ 
17 DISK-WRITE( $y$ )
18 DISK-WRITE( $z$ )
19 DISK-WRITE( $x$ )
    
```

Aloca e inicializa z

Ajusta y

Ajusta x



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

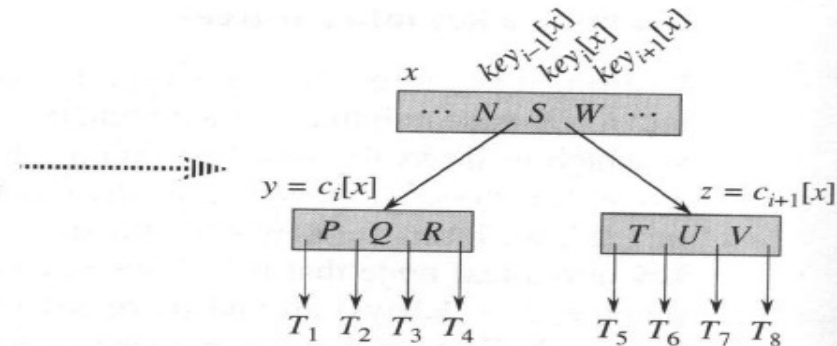
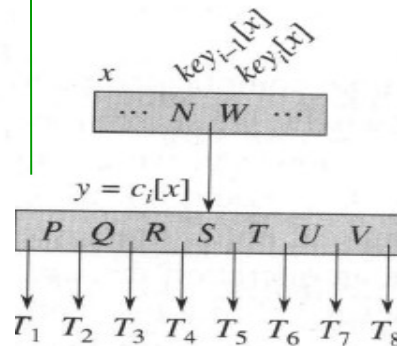
B-TREE-SPLIT-CHILD(x, i, y)

```
1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9   $n[y] \leftarrow t - 1$ 
10 for  $j \leftarrow n[x] + 1$  downto  $i + 1$ 
11     do  $c_{j+1}[x] \leftarrow c_j[x]$ 
12  $c_{i+1}[x] \leftarrow z$ 
13 for  $j \leftarrow n[x]$  downto  $i$ 
14     do  $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$ 
15  $\text{key}_i[x] \leftarrow \text{key}_i[y]$ 
16  $n[x] \leftarrow n[x] + 1$ 
17 DISK-WRITE( $y$ )
18 DISK-WRITE( $z$ )
19 DISK-WRITE( $x$ )
```

Aloca e inicializa z

Ajusta y

Ajusta x

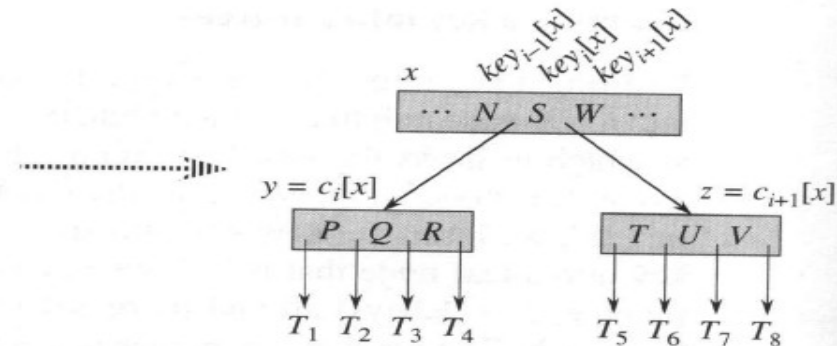
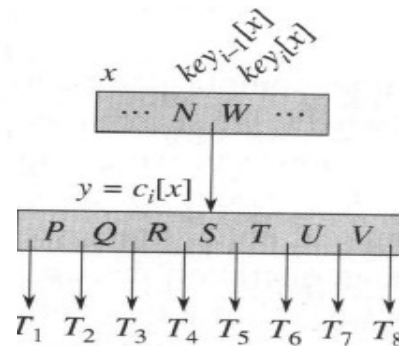


B-TREE-SPLIT-CHILD(x, i, y)

```

1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9   $n[y] \leftarrow t - 1$ 
10 for  $j \leftarrow n[x] + 1$  downto  $i + 1$ 
11     do  $c_{j+1}[x] \leftarrow c_j[x]$ 
12  $c_{i+1}[x] \leftarrow z$ 
13 for  $j \leftarrow n[x]$  downto  $i$ 
14     do  $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$ 
15  $\text{key}_i[x] \leftarrow \text{key}_i[y]$ 
16  $n[x] \leftarrow n[x] + 1$ 
17 DISK-WRITE( $y$ )
18 DISK-WRITE( $z$ )
19 DISK-WRITE( $x$ )
    
```

COMPLEXIDADE:



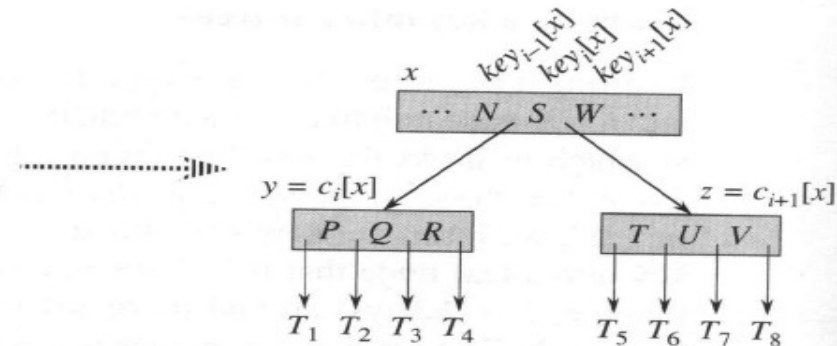
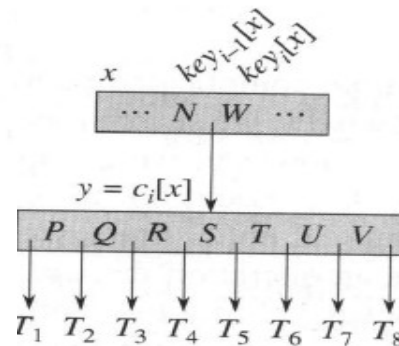
B-TREE-SPLIT-CHILD(x, i, y)

```
1   $z \leftarrow \text{ALLOCATE-NODE}()$ 
2   $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
3   $n[z] \leftarrow t - 1$ 
4  for  $j \leftarrow 1$  to  $t - 1$ 
5      do  $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
6  if not  $\text{leaf}[y]$ 
7      then for  $j \leftarrow 1$  to  $t$ 
8          do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9   $n[y] \leftarrow t - 1$ 
10 for  $j \leftarrow n[x] + 1$  downto  $i + 1$ 
11     do  $c_{j+1}[x] \leftarrow c_j[x]$ 
12  $c_{i+1}[x] \leftarrow z$ 
13 for  $j \leftarrow n[x]$  downto  $i$ 
14     do  $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$ 
15  $\text{key}_i[x] \leftarrow \text{key}_i[y]$ 
16  $n[x] \leftarrow n[x] + 1$ 
17 DISK-WRITE( $y$ )
18 DISK-WRITE( $z$ )
19 DISK-WRITE( $x$ )
```

COMPLEXIDADE:

Acessos ao disco: $O(1)$ – 3 mais precisamente

CPU: $O(t)$



Continua na próxima aula

Referências

Livro do Cormen: cap 18 (3ª ed.)

Livro do Drozdek (4ª ed) cap 7