ACH2024

Aula 17

Organização de arquivos: Árvores B (parte 1)

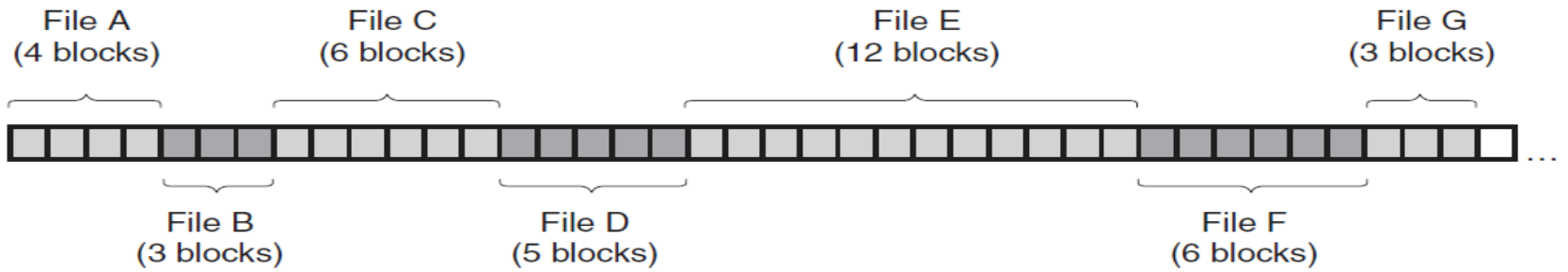
Profa. Ariane Machado Lima

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 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**

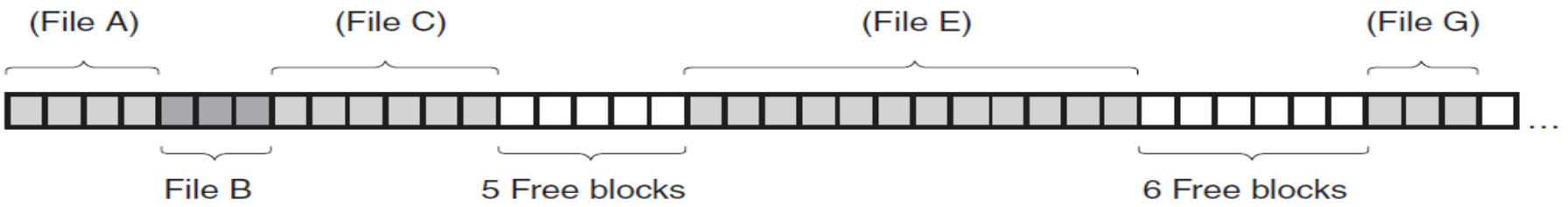
Alocação sequencial

- Blocos alocados sequencialmente no disco (pelos cilindros)



- Após algumas remoções

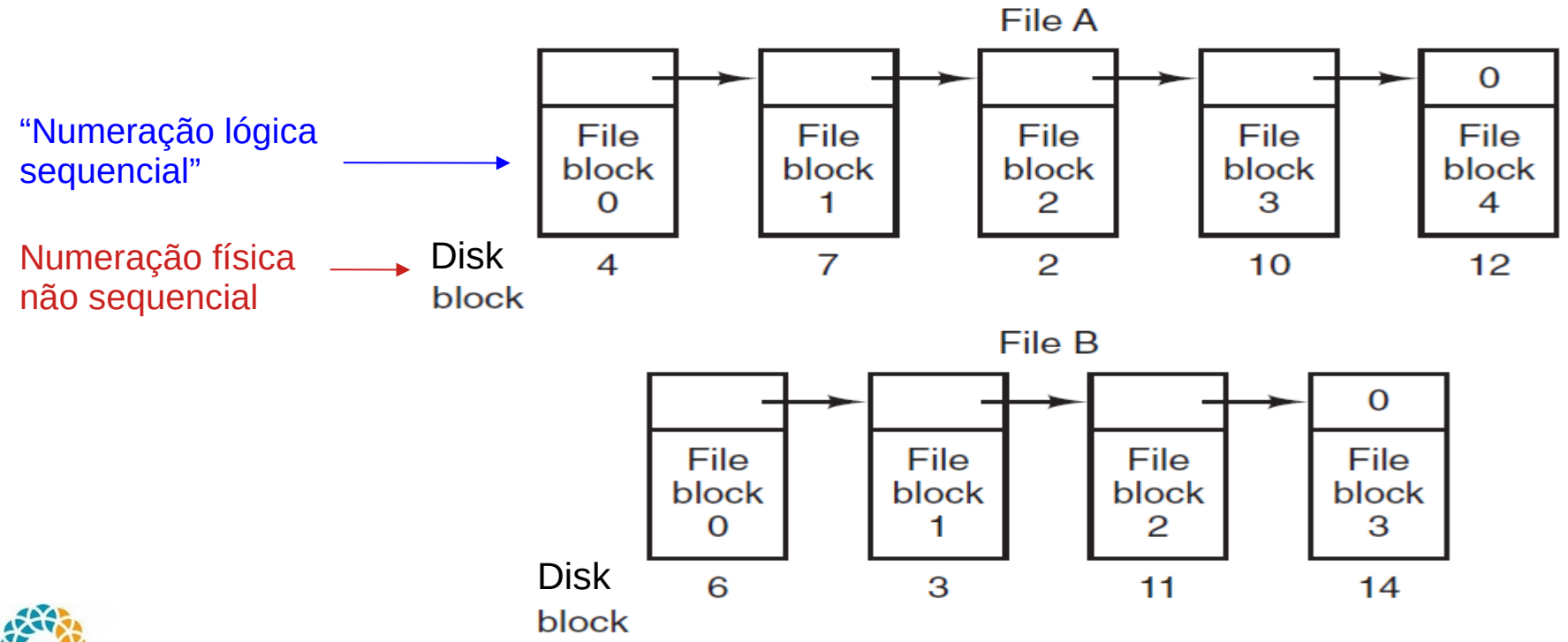
Fragmentação externa



Fonte: (TANEMBAUM, 2015)

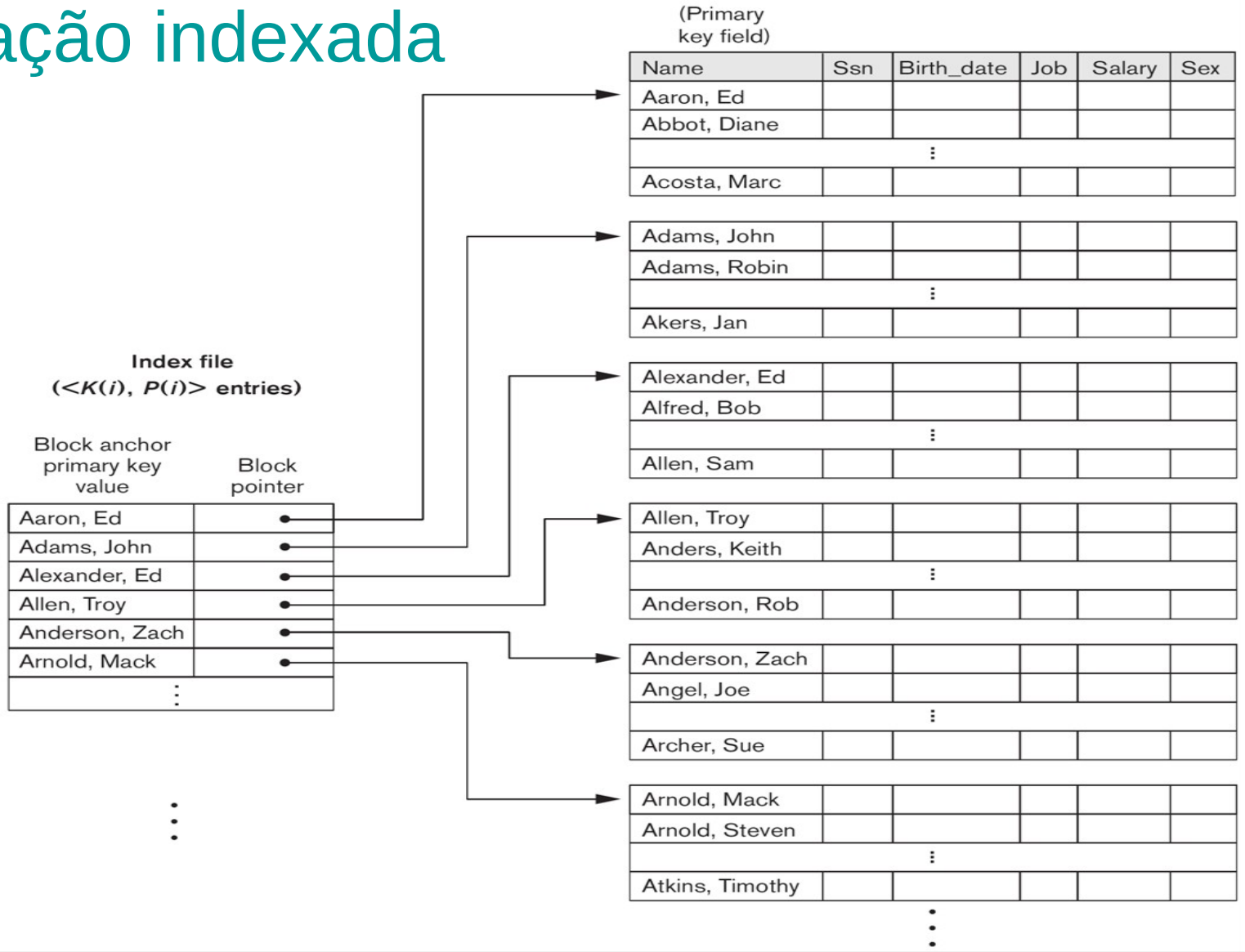
Alocação por listas ligadas

Cada arquivo é uma lista ligada de blocos

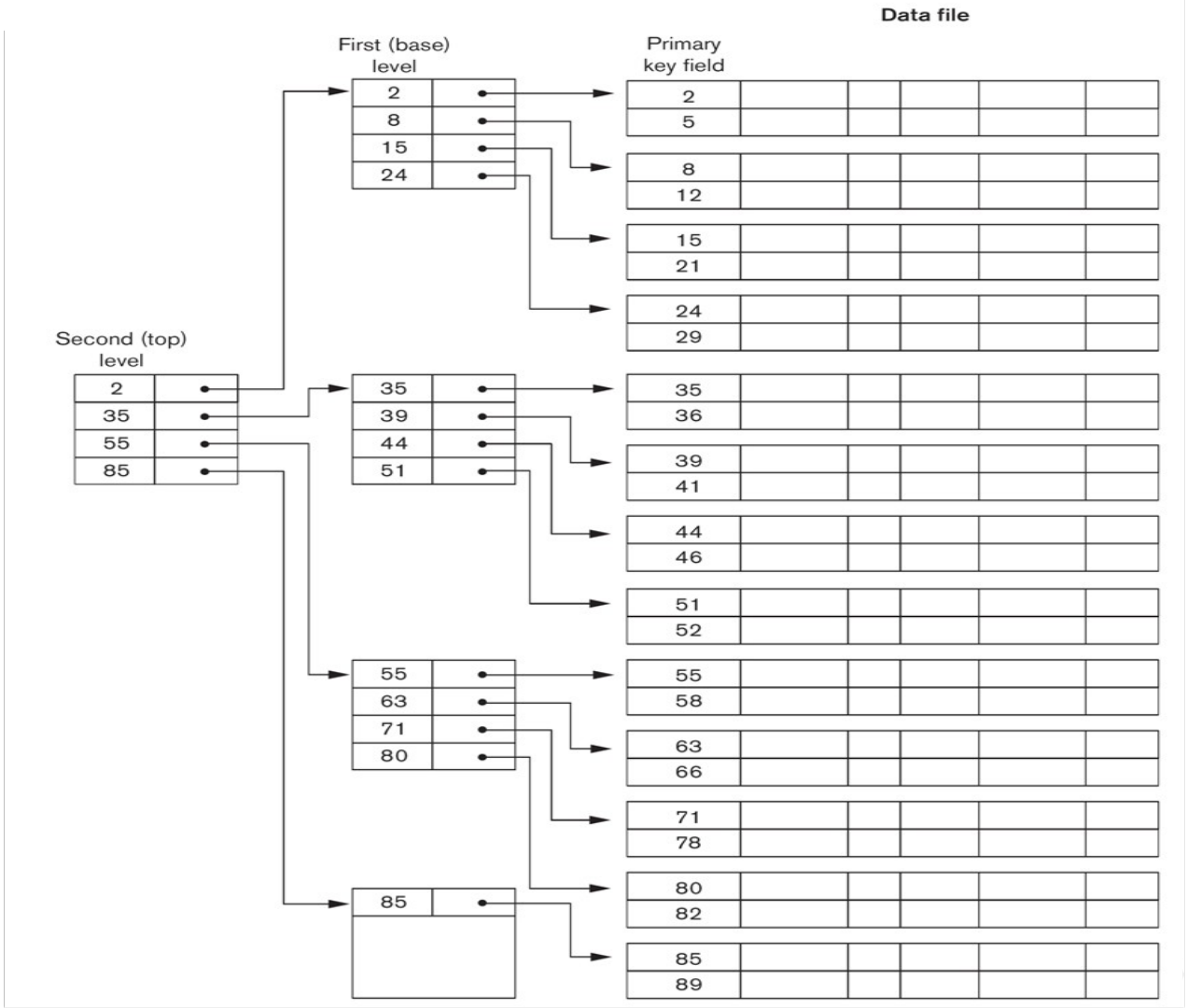


Fonte: (TANEMBAUM, 2015)

Organização indexada



Organização indexada multiníveis



	Sequencial		Ligada	Indexada	Indexada Multinível
	Não-Ordenado	Ordenado	Ordenado	Ordenado	Ordenado
Busca	O(b)	O(lg b)	O(b), O(lg b) só se usar FAT (discos pequenos)	O(log bi)	O(log $_{fbi}$ r[1])
Inserção**	O(1) se tiver espaço no final, O(b) c.c.	O(1) se tiver espaço no bloco, O(b) c.c.	O(1)	O(b+bi) = O(b)	O(b+bi) = O(b)
Remoção* , **	O(1)	O(1)	O(1)	O(1)	O(1)
Leitura ordenada	ω(b) (depende do alg de ord. externa)	O(b), O(1) se fizer a leitura toda de uma vez	O(b)	O(b+bi) = O(b)	O(b+bi) = O(b)
Mínimo/máximo	O(b)	O(1)	O(1)	O(1)	O(1)
Modificação**	O(1)	O(b) se no campo chave, O(1) c.c.	O(1)	O(b) se no campo chave, O(1) c.c.	O(b) se no campo chave, O(1) c.c.
* considerando uso de bit de validade					
** considerando que já se sabe a localização do registro (busca já realizada)					

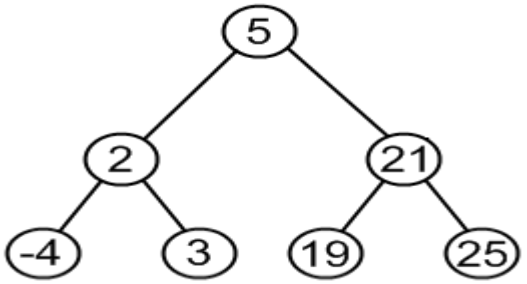
VELHO DILEMA ENTRE TEMPO DE BUSCA E DINAMISMO!

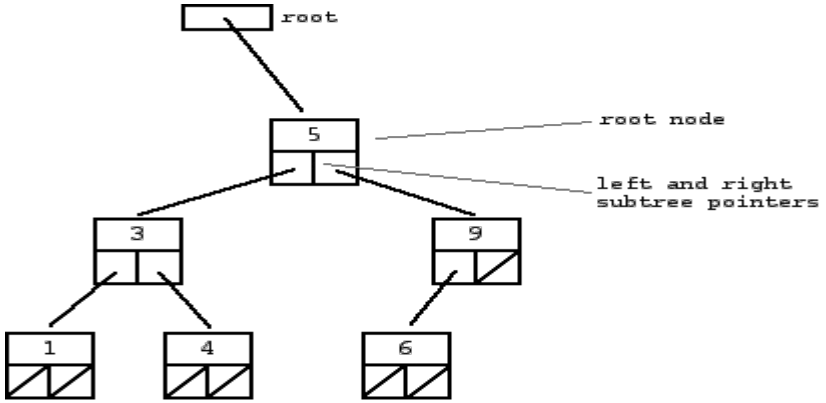
Lembrando de AED 1...

- Busca binária (em um vetor):

  -4, 2, 3, 5, 19, 21, 25

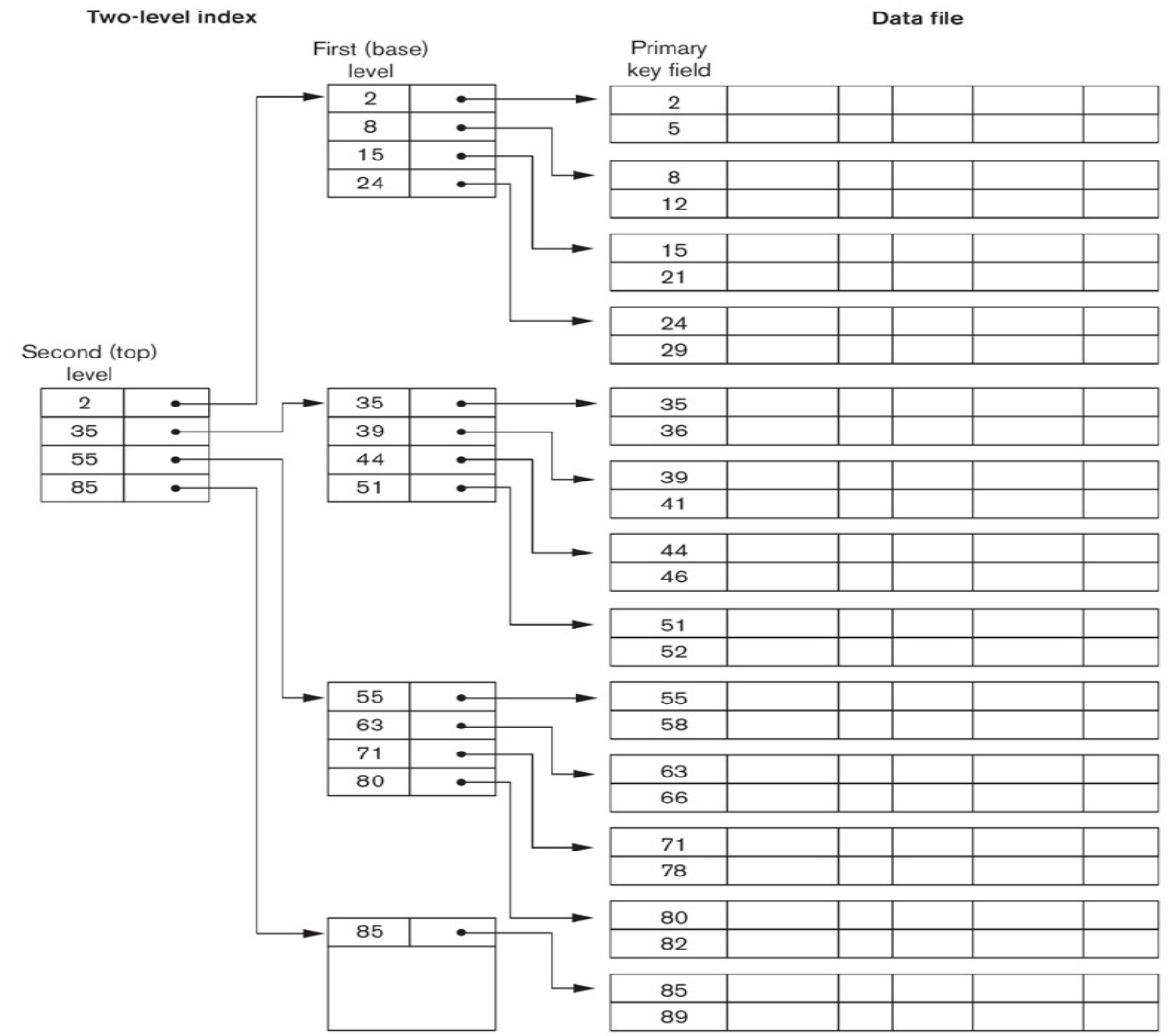
- Árvores Binárias de Busca:





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.



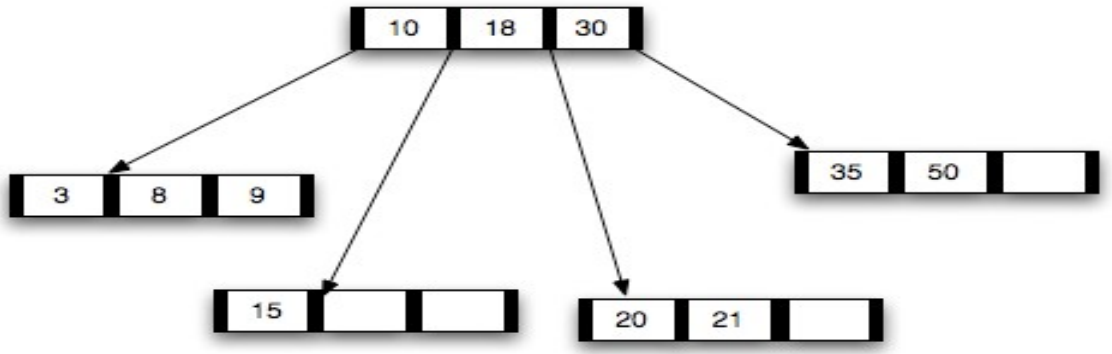
Podemos pensar em algo semelhante para melhorar o dinamismo dos índices múltiníveis?

- 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 na árvore
- 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)

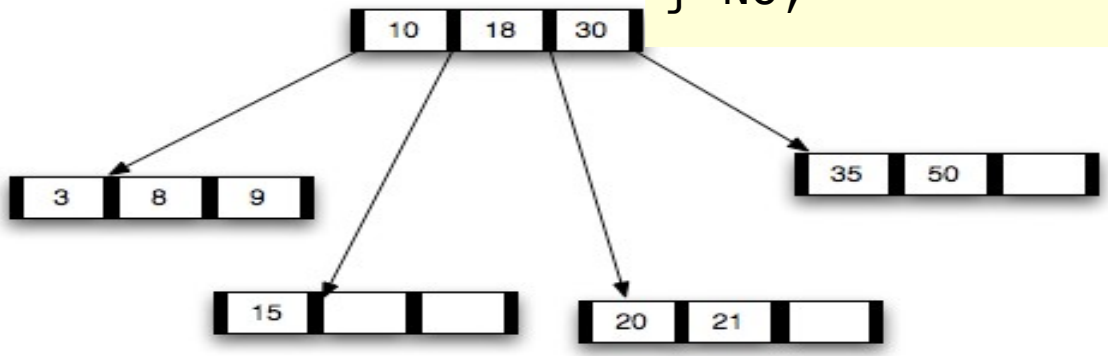


Árvore B Clássica

- 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;
```

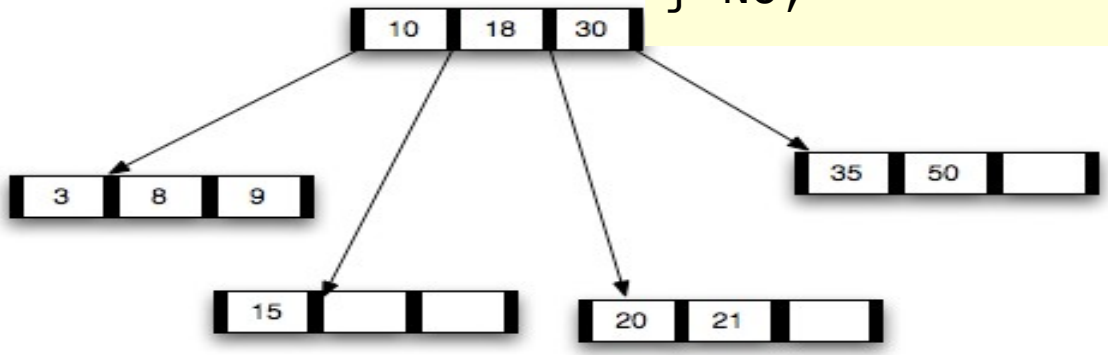


Árvore B Clássica

- 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;
```



Aula de hoje

- Árvores-B:

→ definição

→ busca

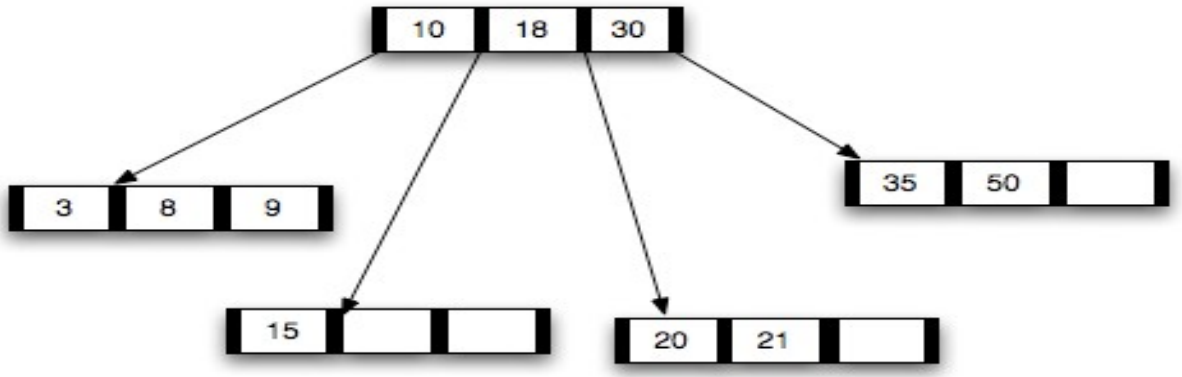
→ inserção

Árvore B – Definição (notação usada pelo Cormen)

Vamos usar essa notação nas aulas!!!

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

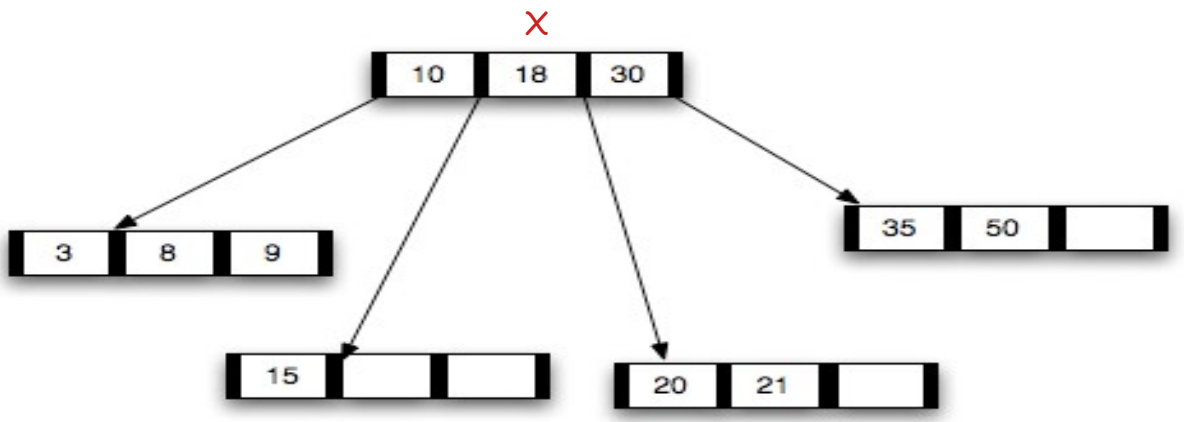
  1. Cada nó x contém os seguintes campos:
     - $n[x]$, o número de chaves atualmente armazenadas no nó x;
     - as $n[x]$ chaves, armazenadas em ordem não decrescente, de modo que $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$;
     - $leaf[x]$, um valor booleano indicando se x é uma folha (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

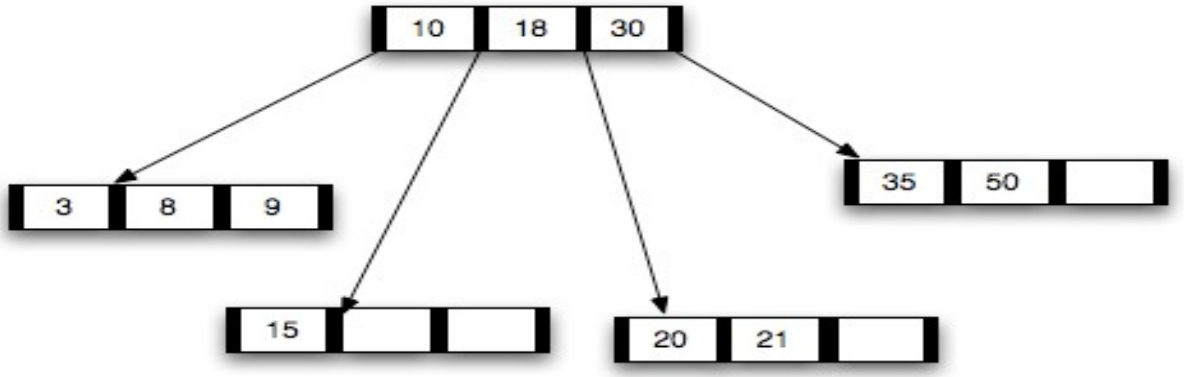
2. As chaves $key_i[x]$ separam as faixas de valores armazenados em cada subárvore: denotando por k_i uma chave qualquer armazenada na subárvore com nó $c_i[x]$, tem-se

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



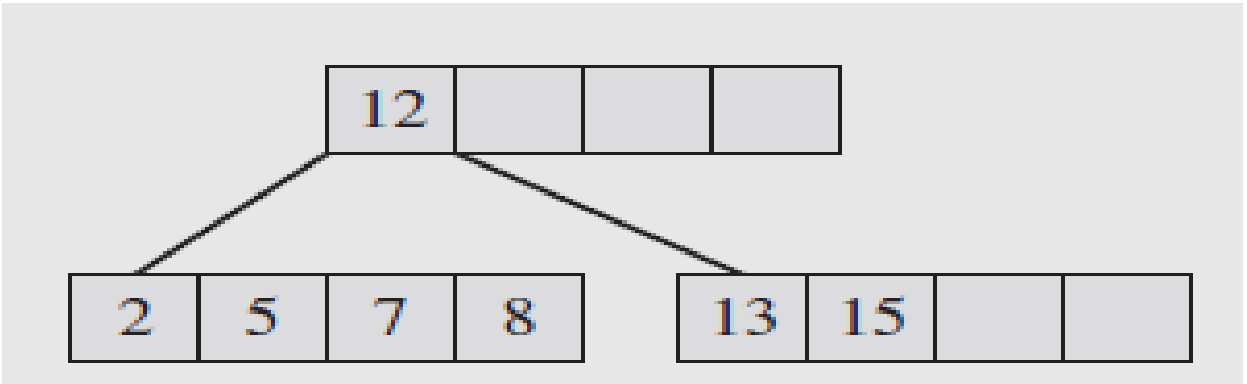
Árvore B - Definição

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



Árvore B - Definição

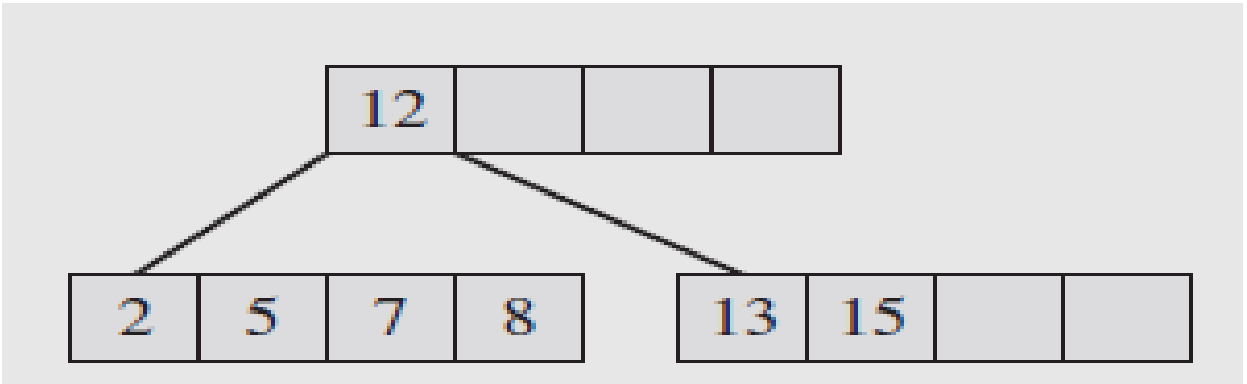
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h.



OBS: E se eu precisasse inserir o valor 1 ?

Árvore B - Definição

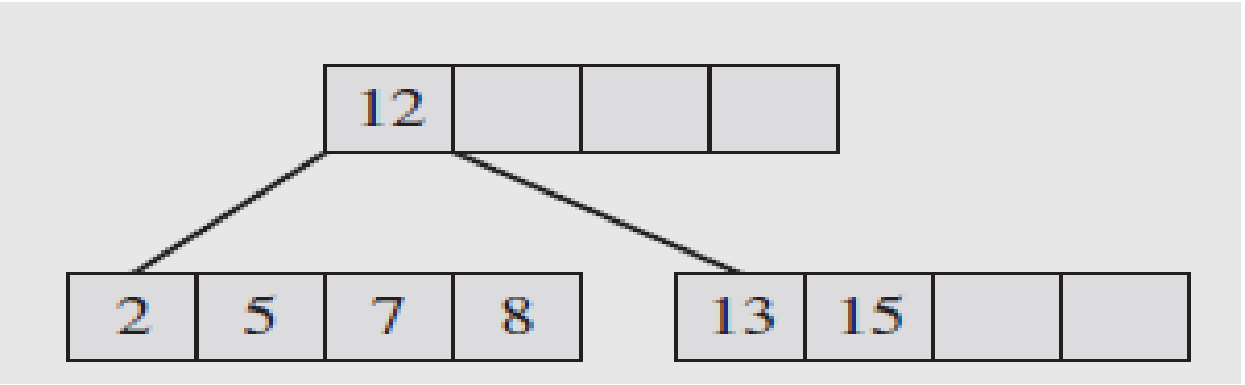
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h.



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.



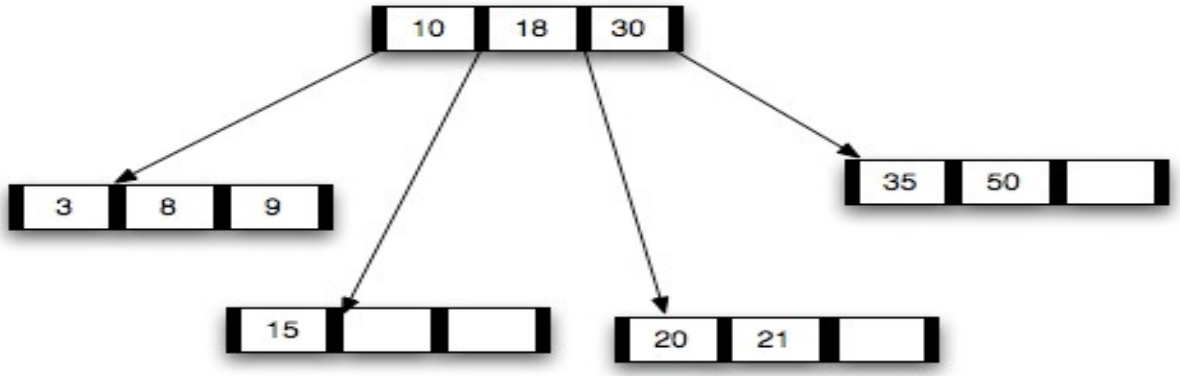
OBS: E se eu precisasse inserir o valor 1 ? Criaria um filho à esquerda de 2?

1-) Aumentaria a altura da árvore mesmo ela tendo espaço

2-) Feriria a definição 3...

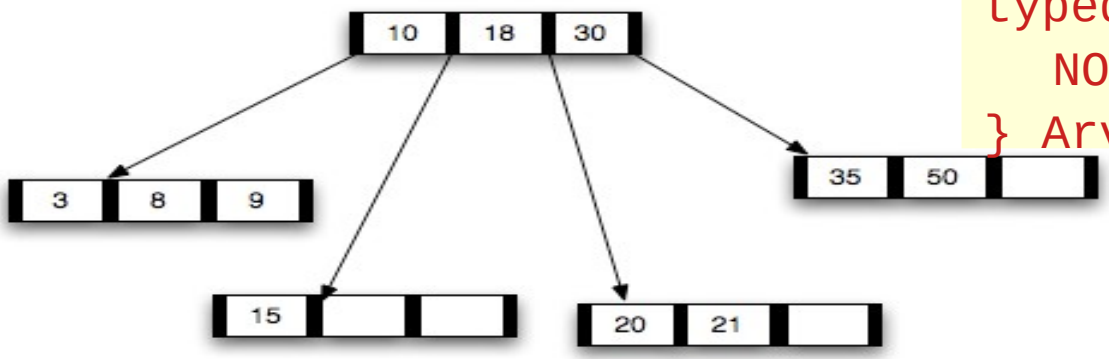
Árvore B - Definição

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

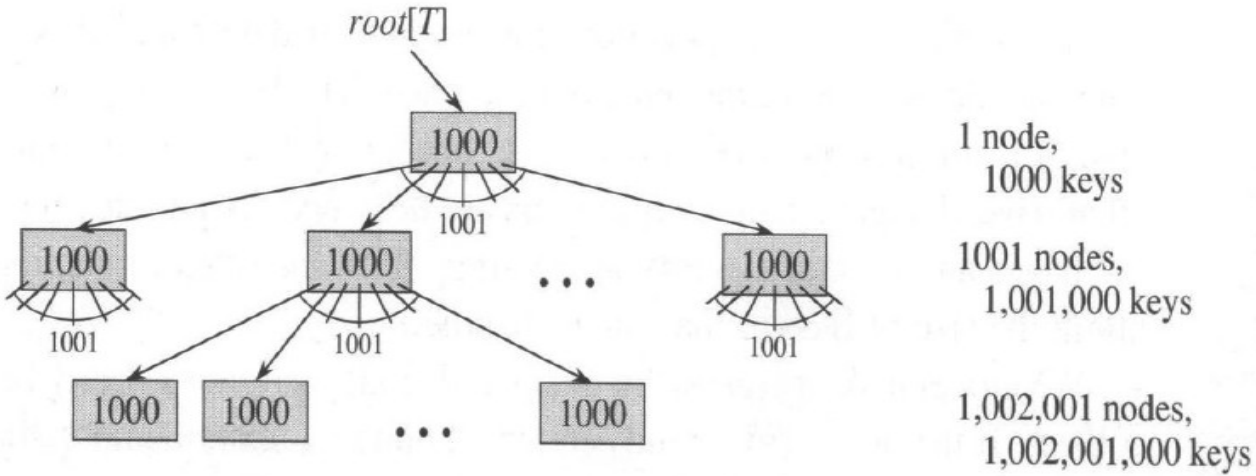



```
#define t 2  /* grau min da árvore*/
typedef  int  TipoChave;
typedef struct str_no {
    TipoChave chave[2*t-1];
    struct str_no* filhos[2*t];
    int numChaves;
    bool folha;
} NO;
typedef struct {
   NO* raiz;
} ArvB;
```

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 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,

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

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 $x \leftarrow$ ALLOCATE-NODE()
2 $leaf[x] \leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 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 $x \leftarrow$ ALLOCATE-NODE()
2 $leaf[x] \leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 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: 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;
```

NO* é na verdade o número do bloco do disco

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

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

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)

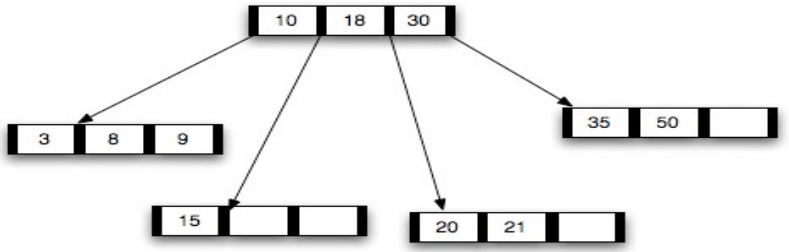
```
bool criaArvoreB(ArvB* T){
  NO* x;
  if(!(x = (NO*) malloc(sizeof(NO)))) {
      /* msg erro e retorna false */
  }
  x->folha = true;
  x->numChaves = 0;
  escreveNoDisco(x); /*vamos abstrair isso*/
  T->raiz = x;
  retorna true;
```

Busca na árvore B

- B-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-Tree-Search(T→raiz, 22)

Lembrando que nos pseudocódigos do Cormen os vetores começam em 1



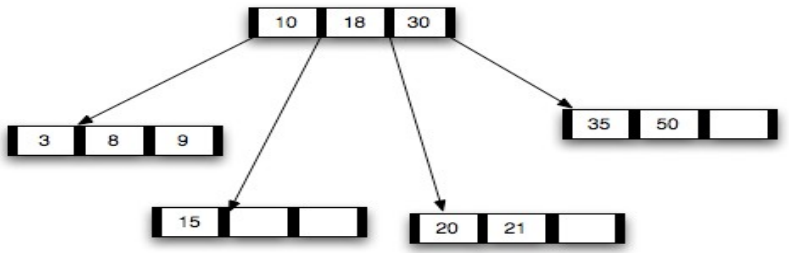
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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→raiz, 22)

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
```



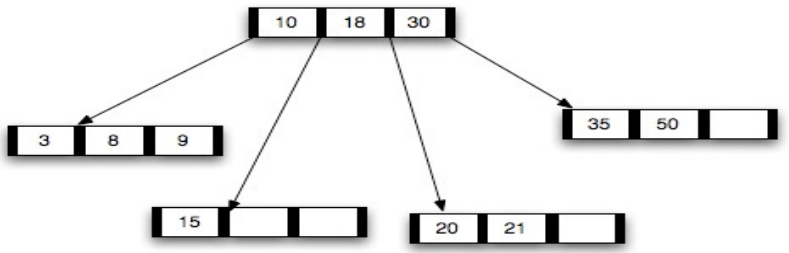
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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→raiz, 22)

```
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]
```



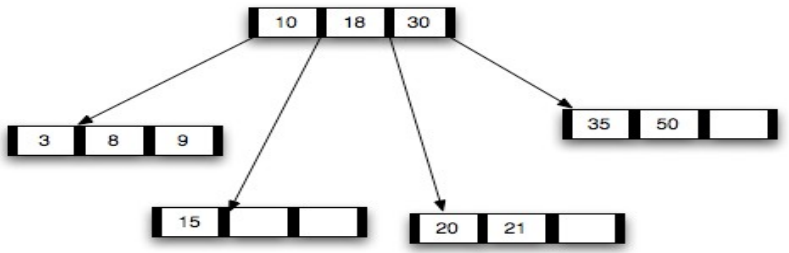
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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)
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)
```



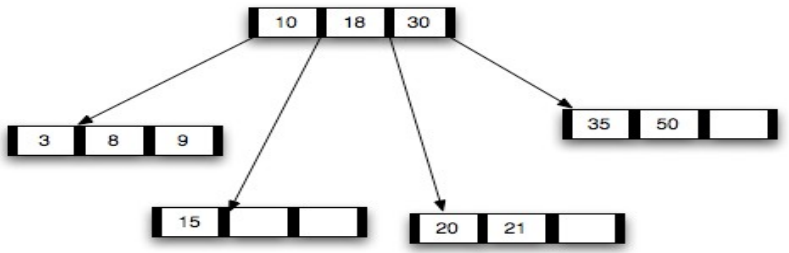
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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)
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]
```



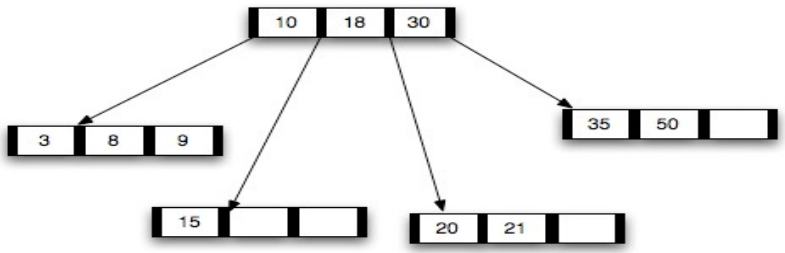
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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)
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)
```



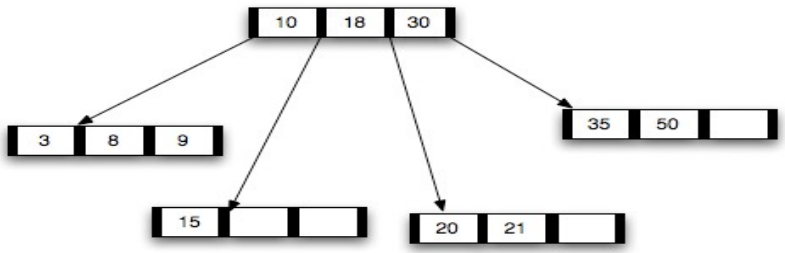
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

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)
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)
```



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.

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

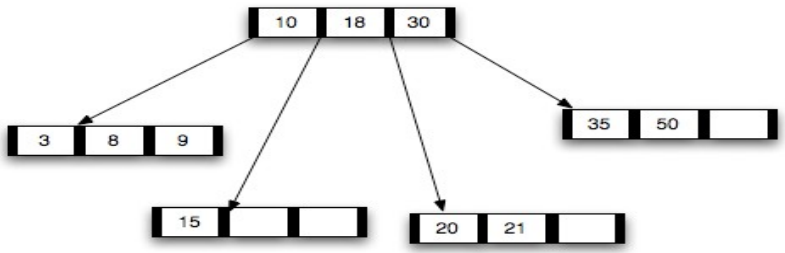
Busca na árvore B

- 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)

Complexidade:

```
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)
```



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

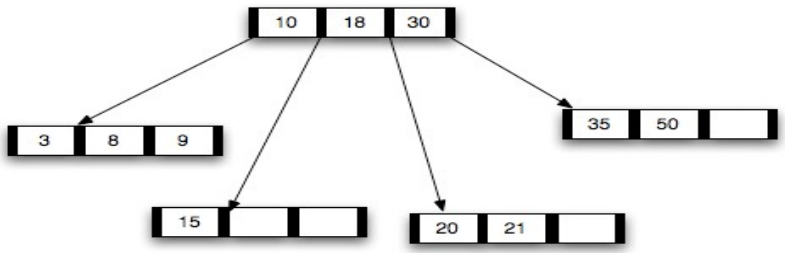
Busca na árvore B

- 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)

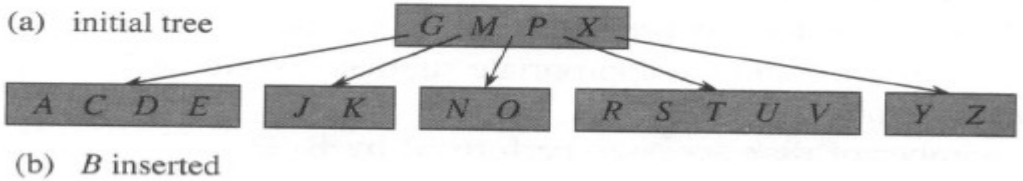
Complexidade: $O(\log_t n)$

```
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)
```

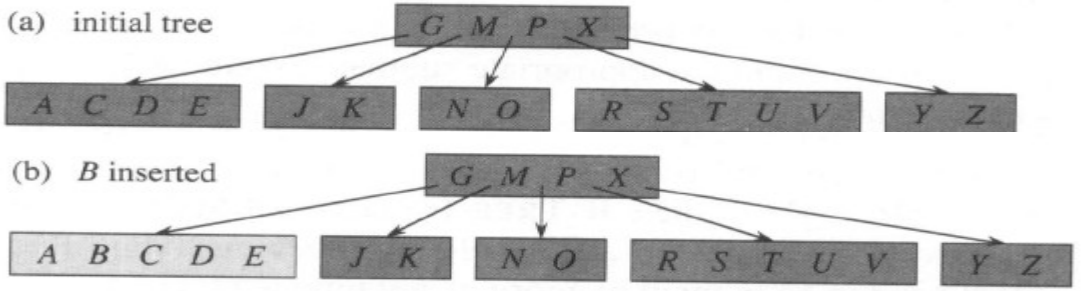


Inserção em árvore B

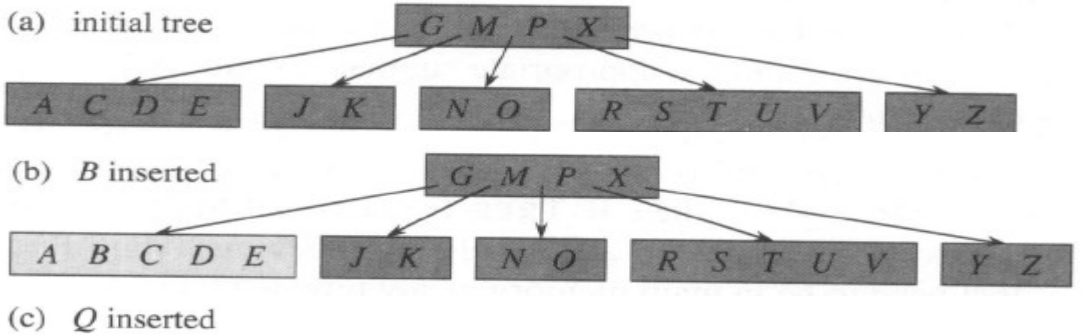
As inserções ocorrem sempre nas folhas



Exemplo Inserção t = 3

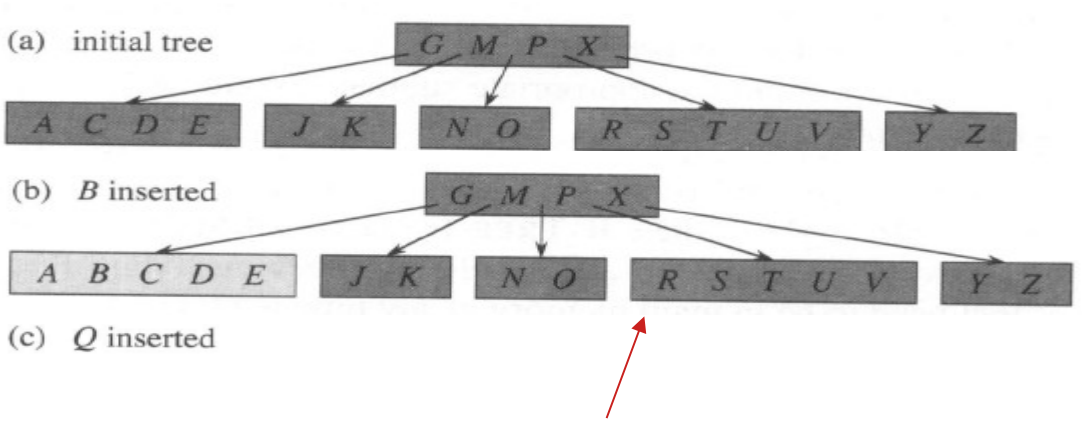


Exemplo Inserção t = 3



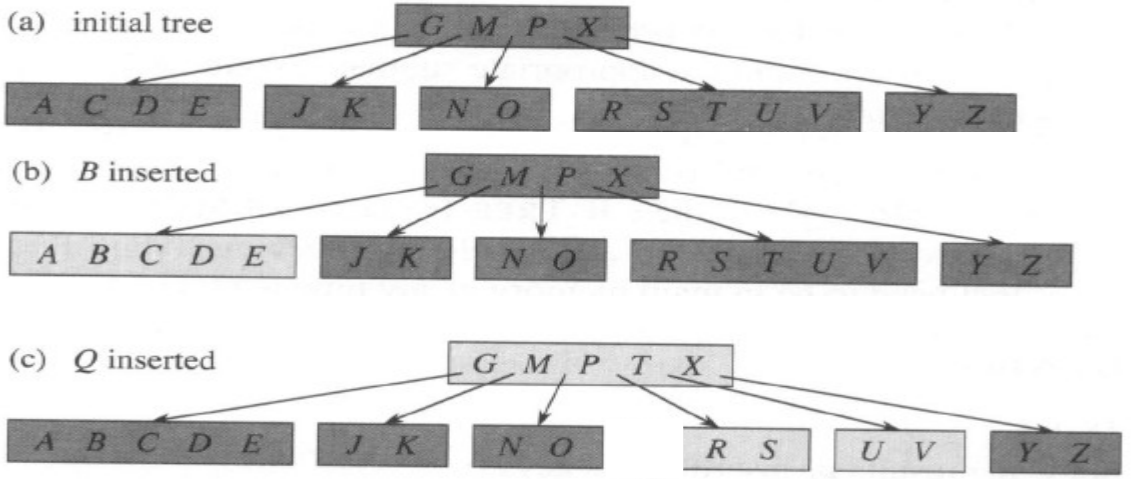
Exemplo Inserção t = 3

Exemplo Inserção t = 3



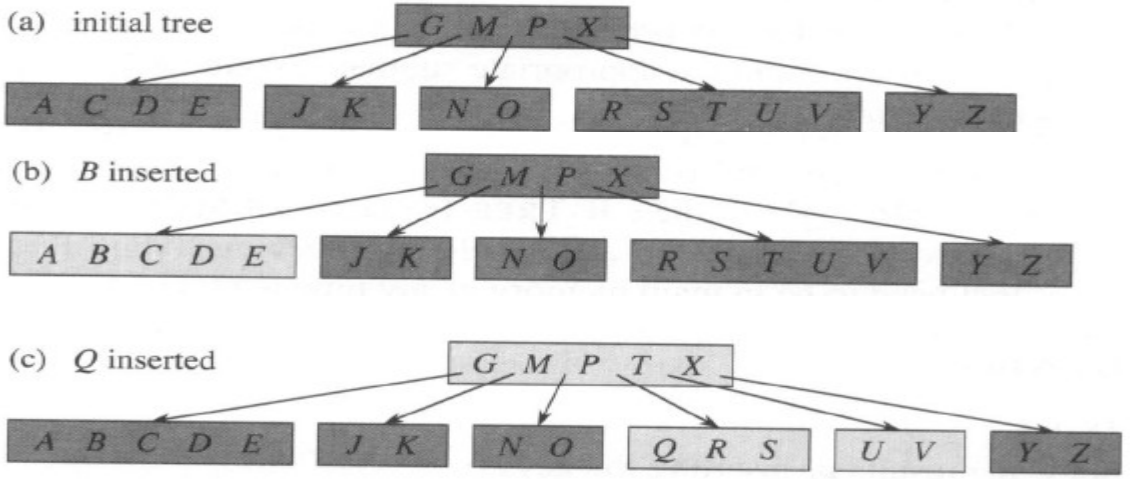
Seria aqui, mas este nó está cheio…
O que fazer?

Exemplo Inserção t = 3



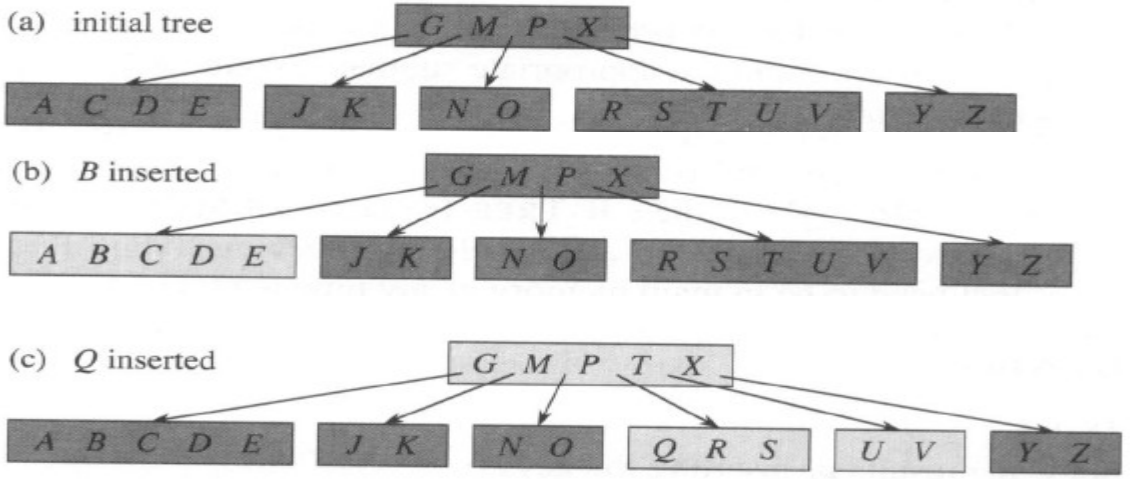
Primeiro quebro o nó em dois (envolve a subida da chave do meio)

Exemplo Inserção t = 3



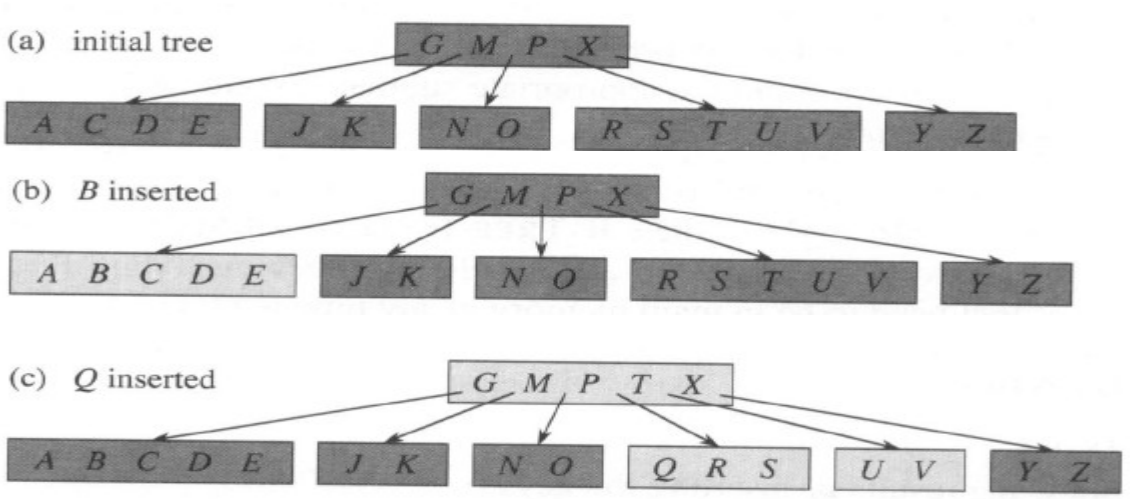
Insiro a chave

Exemplo Inserção t = 3



E se quisesse inserir F?

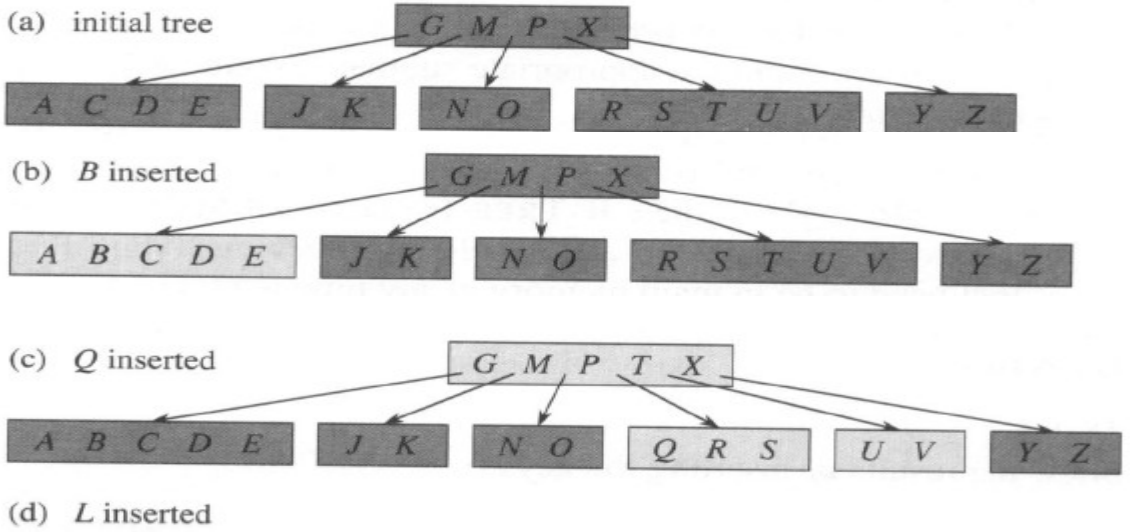
Exemplo Inserção t = 3



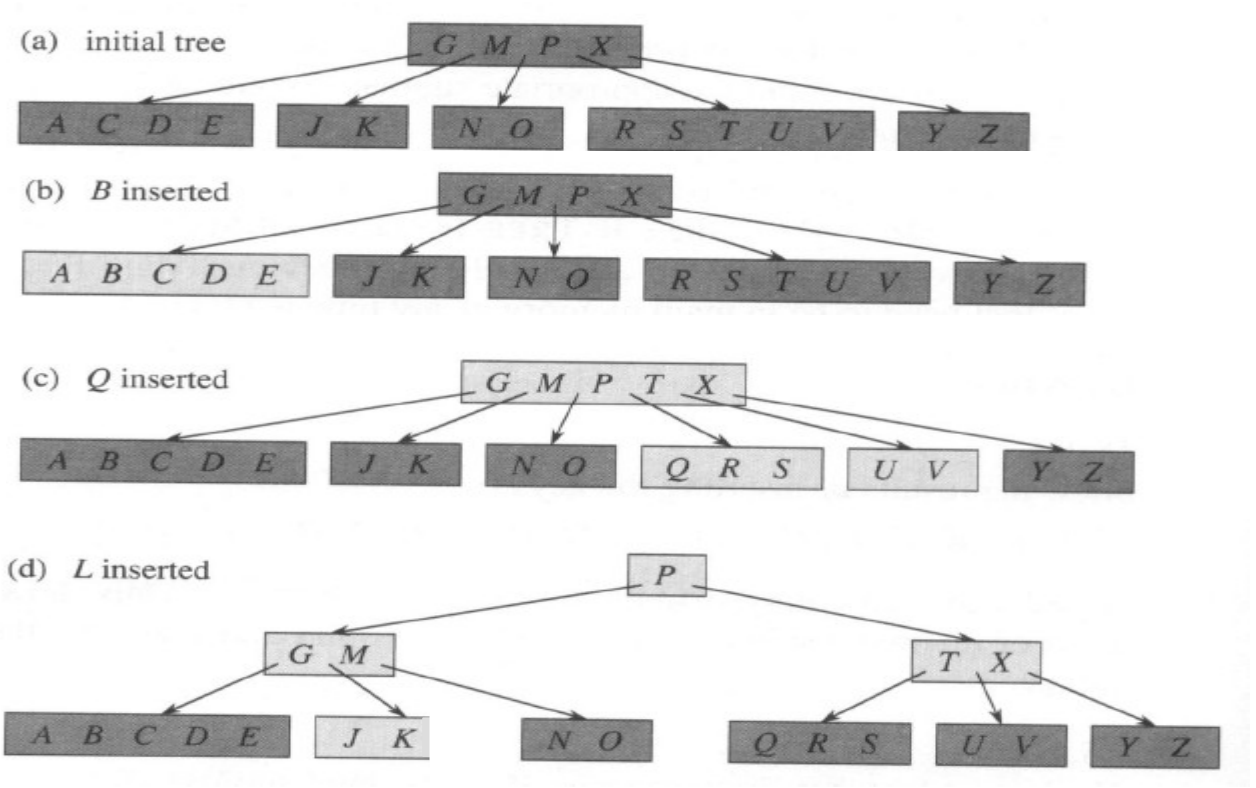
E se quisesse inserir F?
O nó pai estaria cheio, mas não tenho ponteiro para subir para ele para quebrá-lo em dois…

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

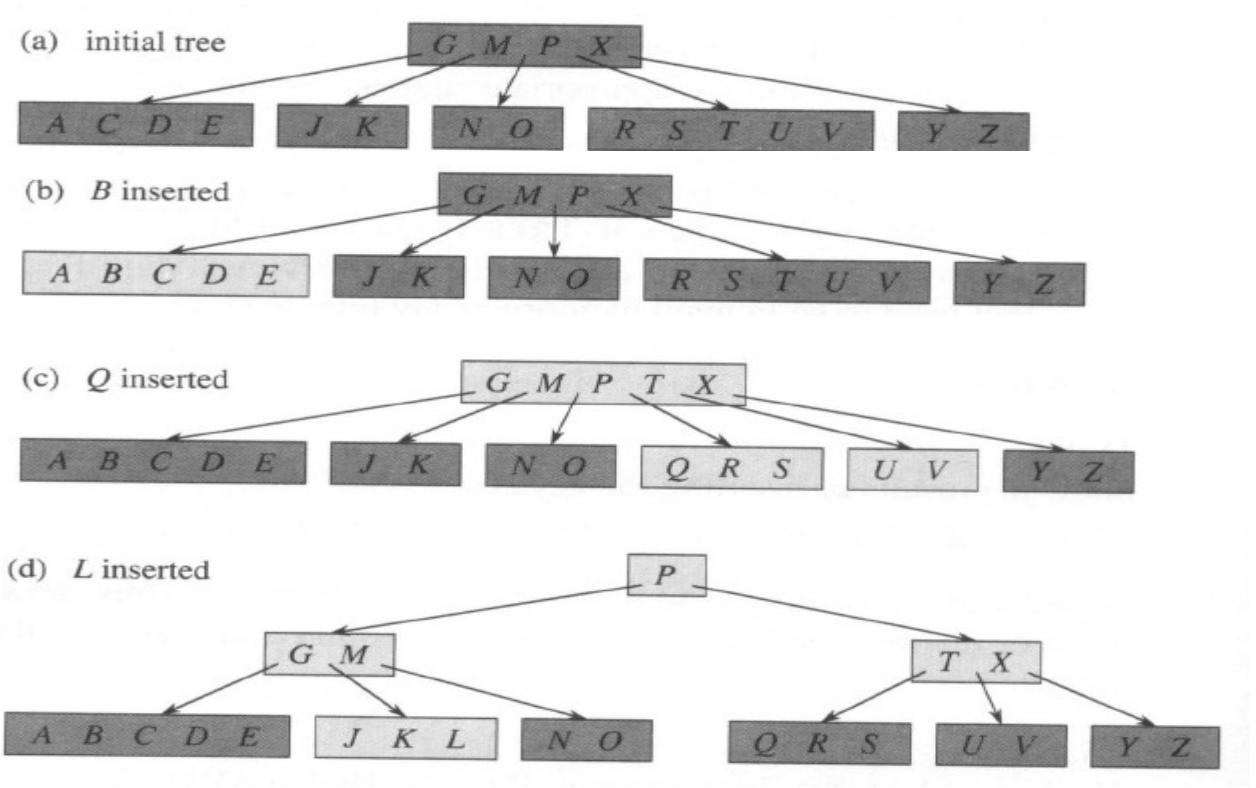
Exemplo Inserção t = 3



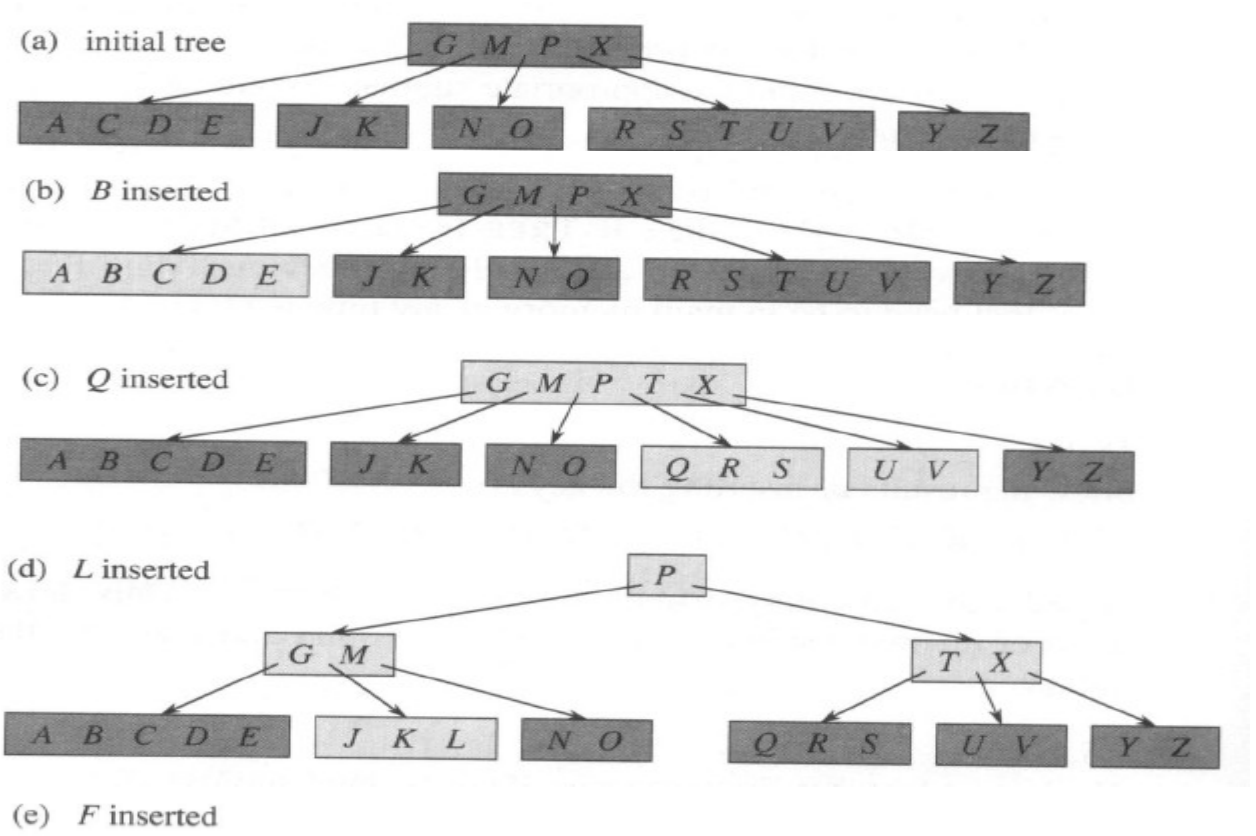
Exemplo Inserção t = 3



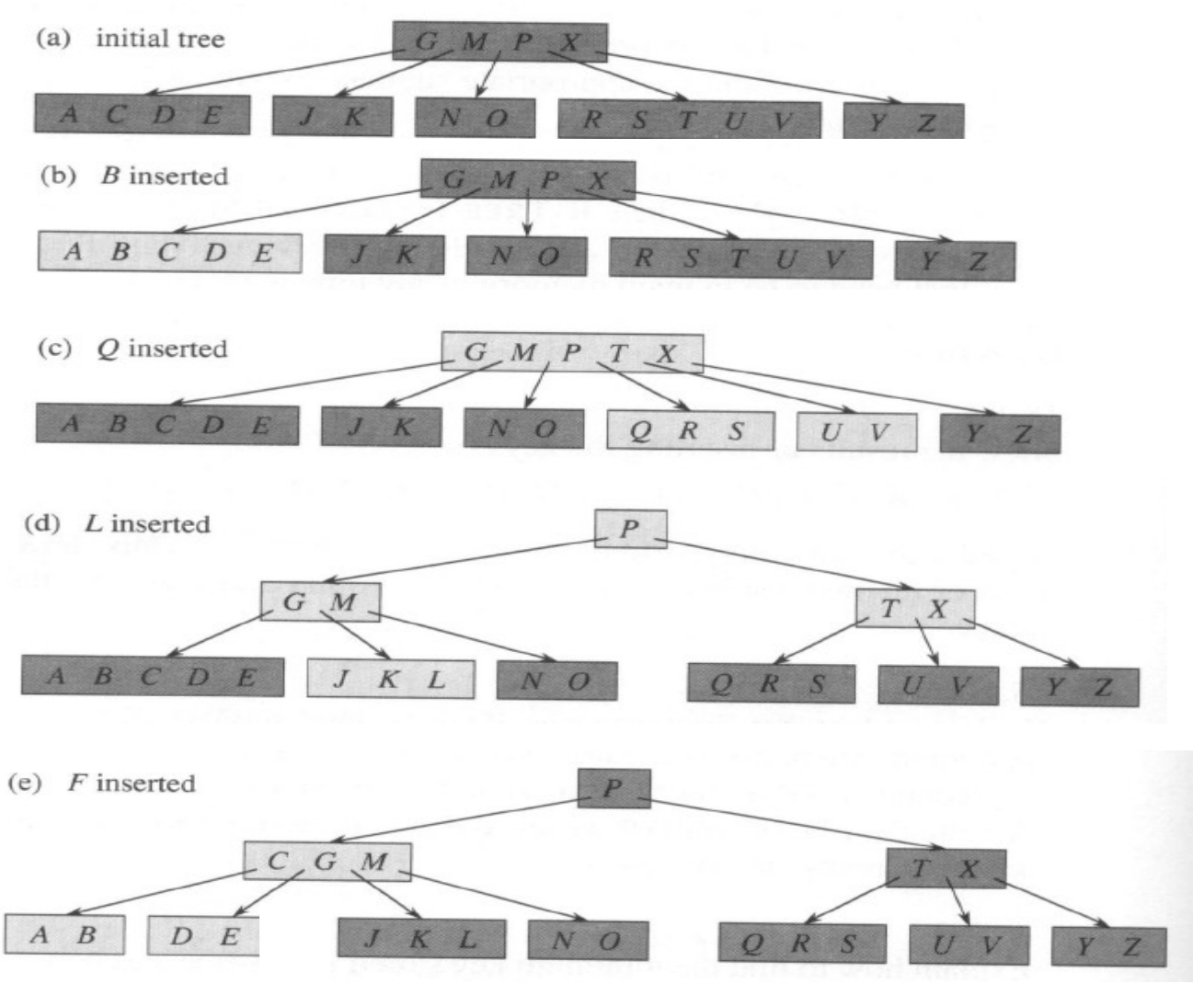
Exemplo Inserção t = 3



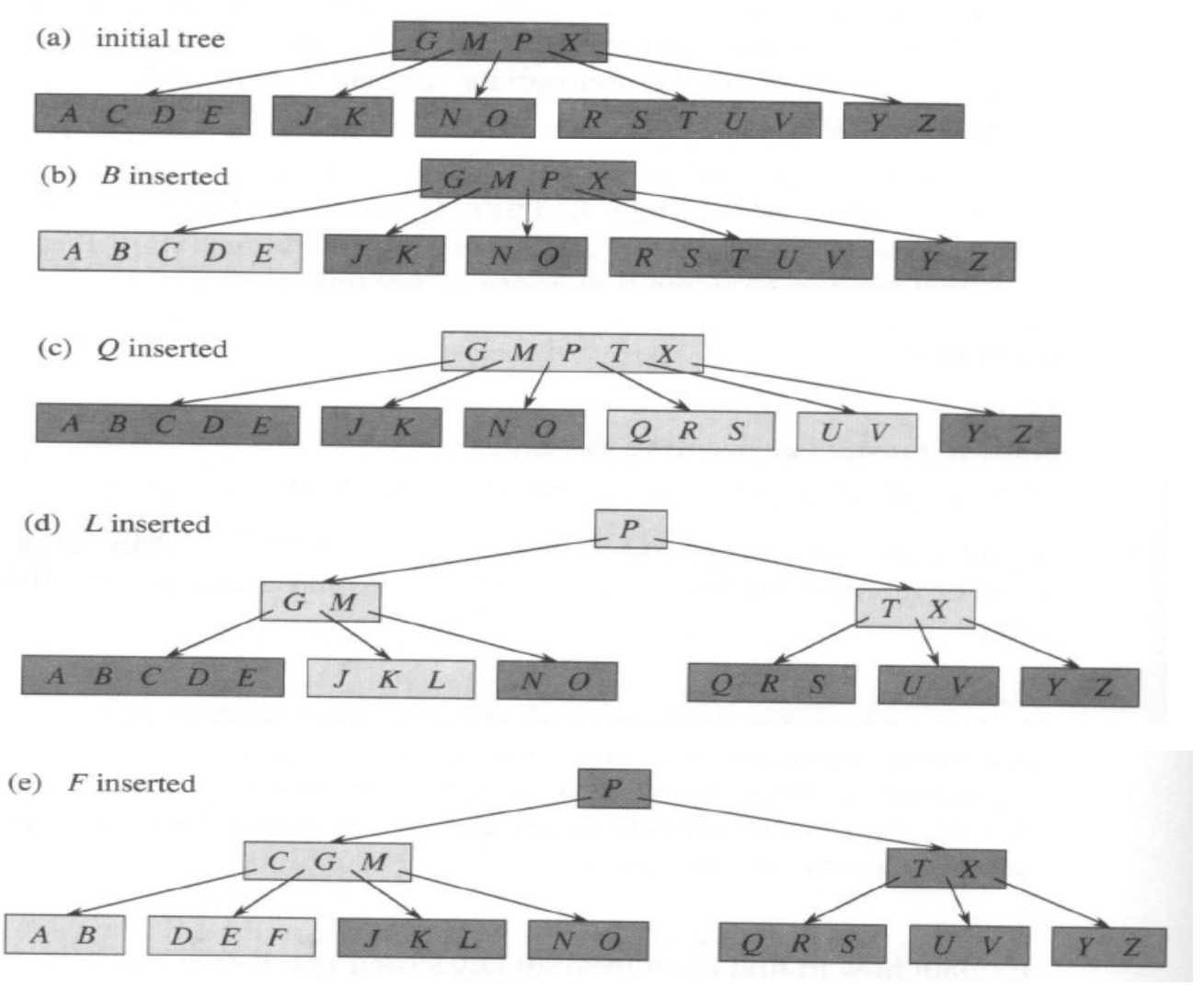
Exemplo Inserção t = 3



Exemplo Inserção t = 3

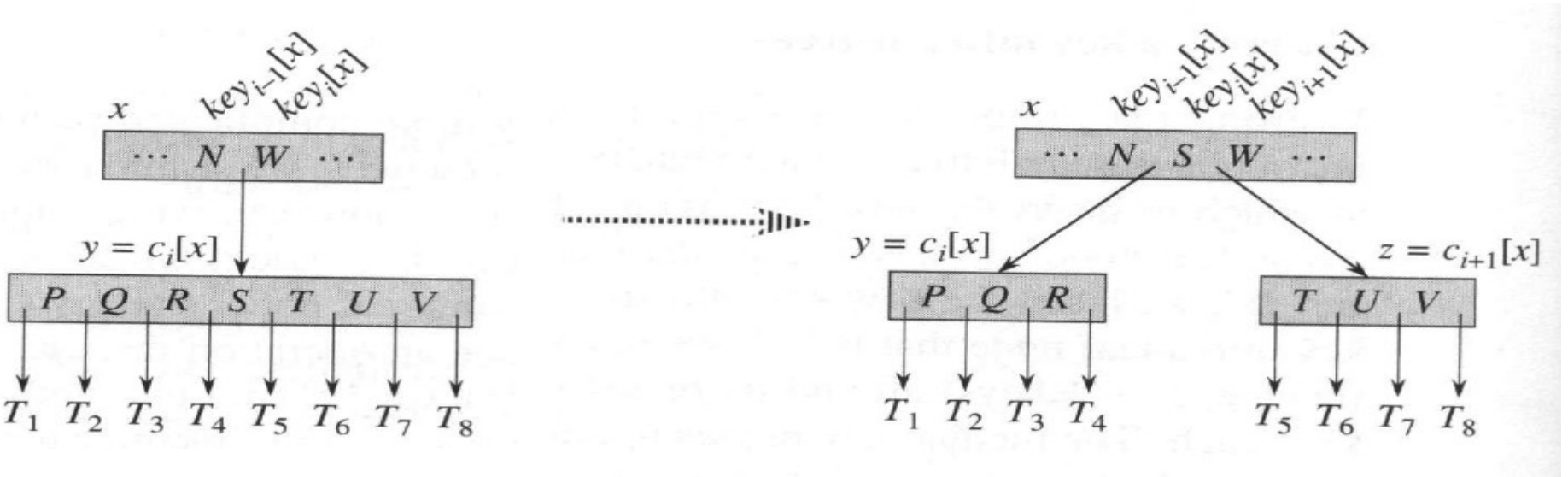


Exemplo Inserção t = 3



As inserções ocorrem sempre nas folhas

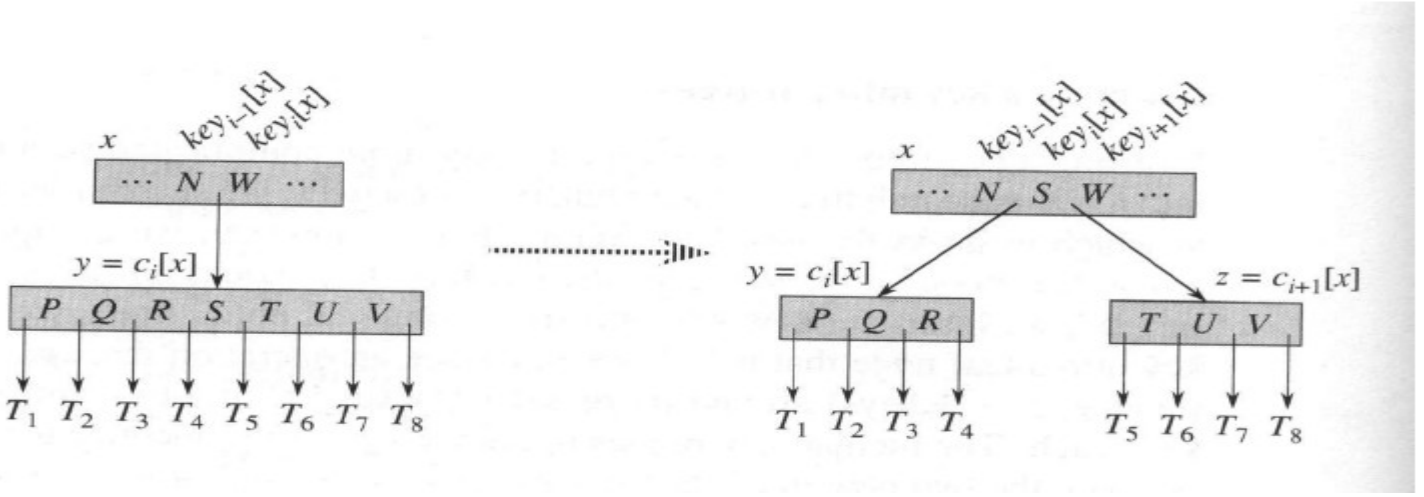
Resolvendo o problema do nó cheio...



- Divisão de um nó na árvore:

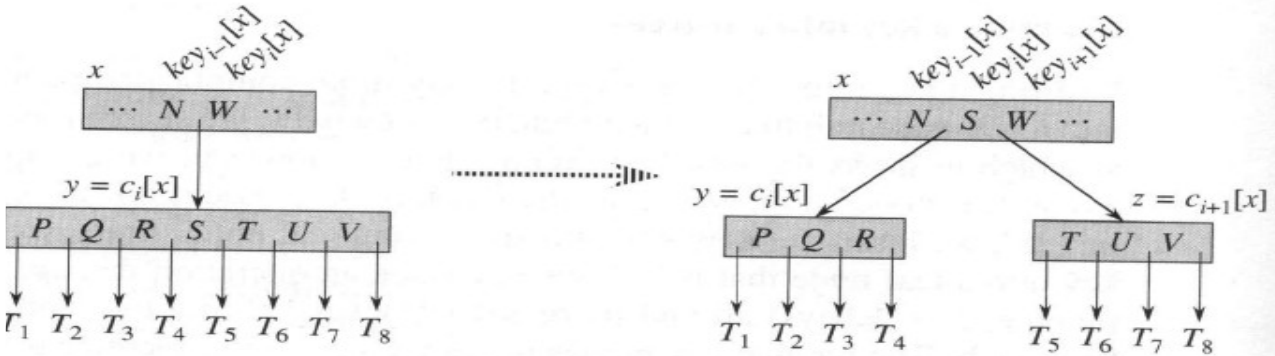
  B-Tree-Split-Child(x, i, y): tem como entrada um nó interno *x não cheio*, um índice i e um nó y tal que $y = c_i[x]$ é um filho *cheio* de x. O procedimento divide y em 2 e ajusta x de forma que este terá um filho adicional.

Porque se ele estava cheio já foi dividido



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

O que precisa fazer?



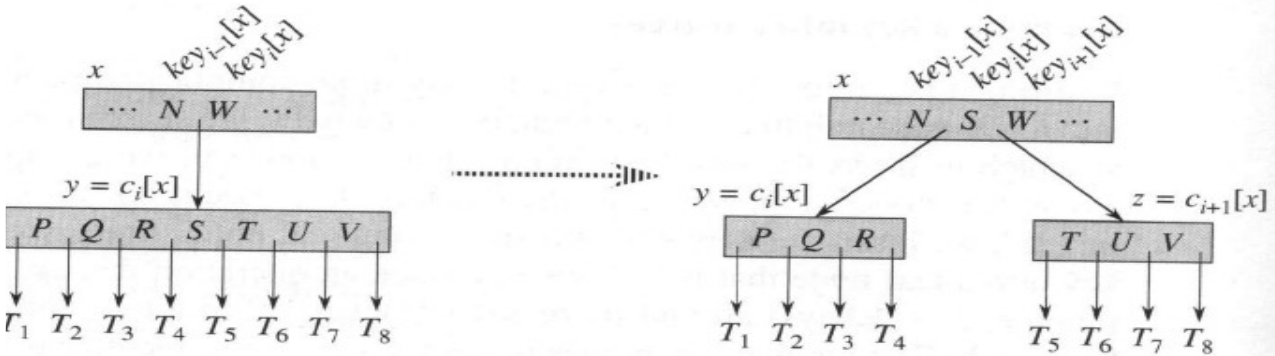
B-TREE-SPLIT-CHILD (x, i, y)

O que precisa fazer?

Aloca e inicializa z

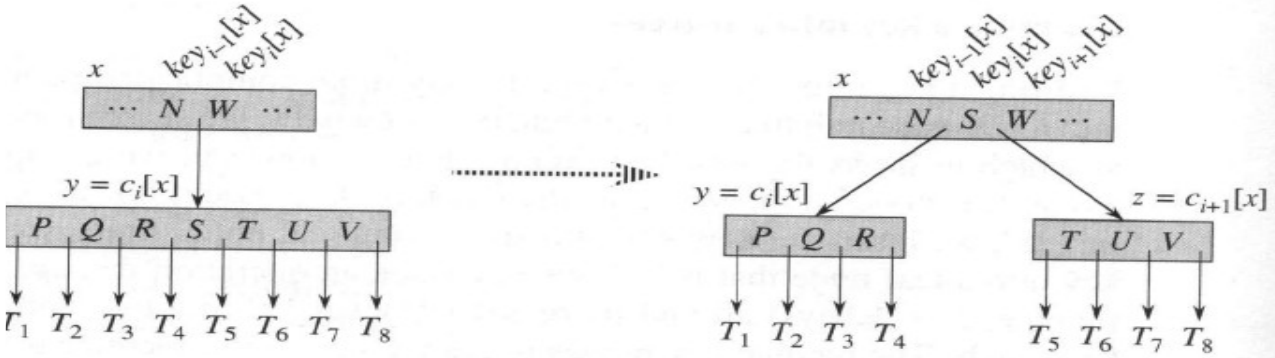
Ajusta y

Ajusta x



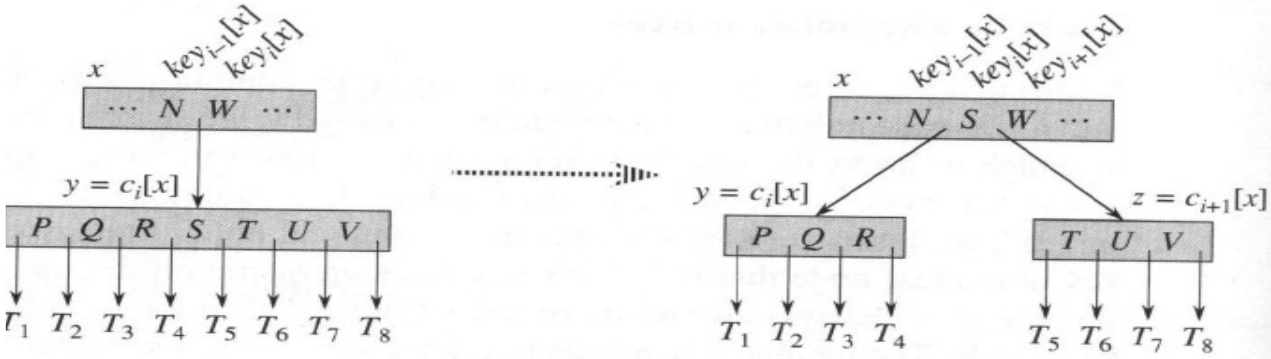
B-TREE-SPLIT-CHILD (x, i, y)

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]
```

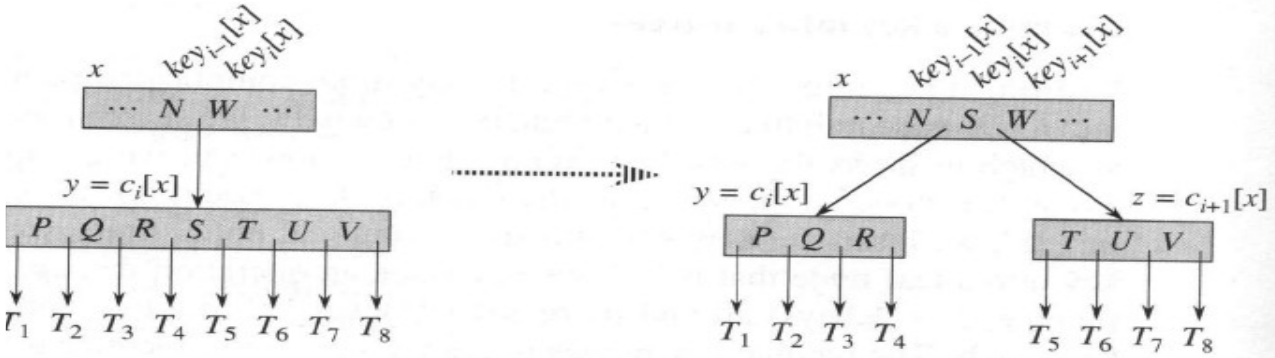
Aloca e inicializa z



```
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

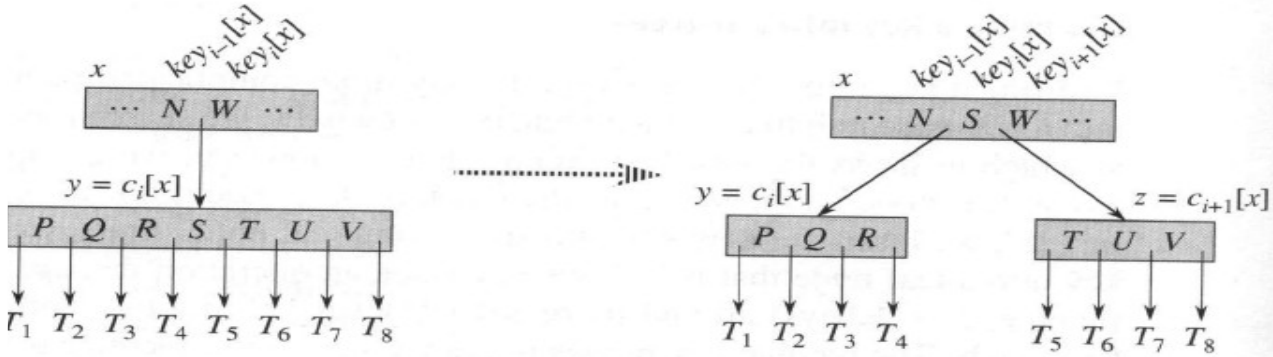
Ajusta y



```
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

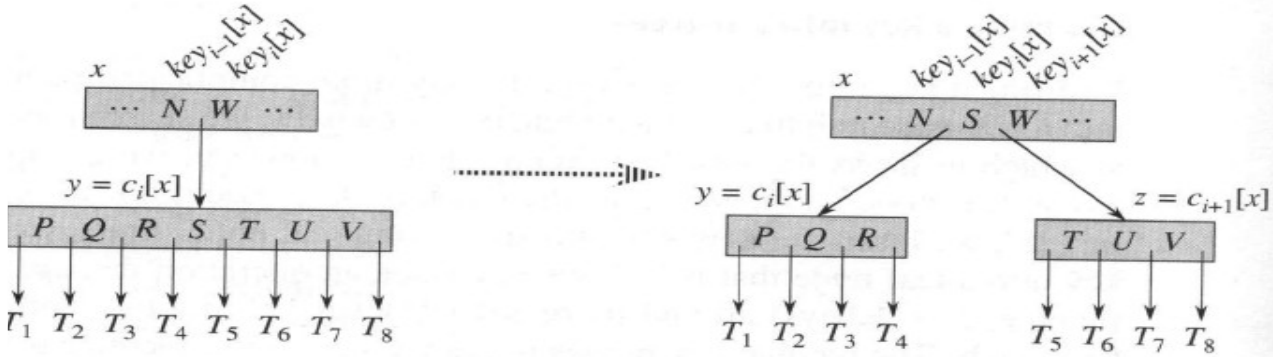


```
B-TREE-SPLIT-CHILD(x, i, y)
1  z ← ALLOCATE-NODE()
2  leaf[z] ← leaf[y]
3  n[z] ← t − 1
4  for j ← 1 to t − 1
5       do key_j[z] ← key_{j+t}[y]
6  if not leaf[y]
7     then for j ← 1 to t
8               do c_j[z] ← c_{j+t}[y]
9  n[y] ← t − 1
```

Aloca e inicializa z

Ajusta y

Ajusta x

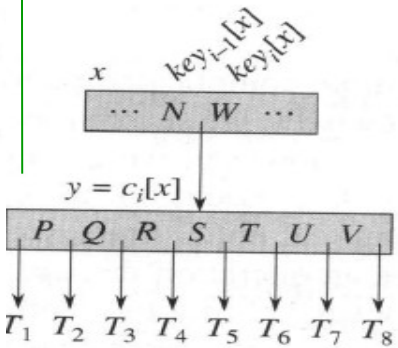


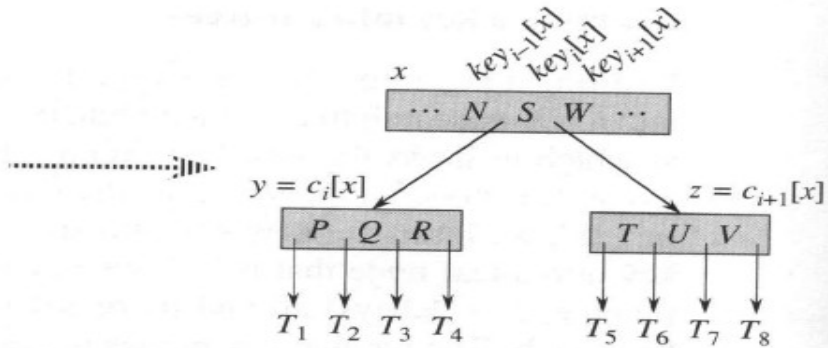
```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do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





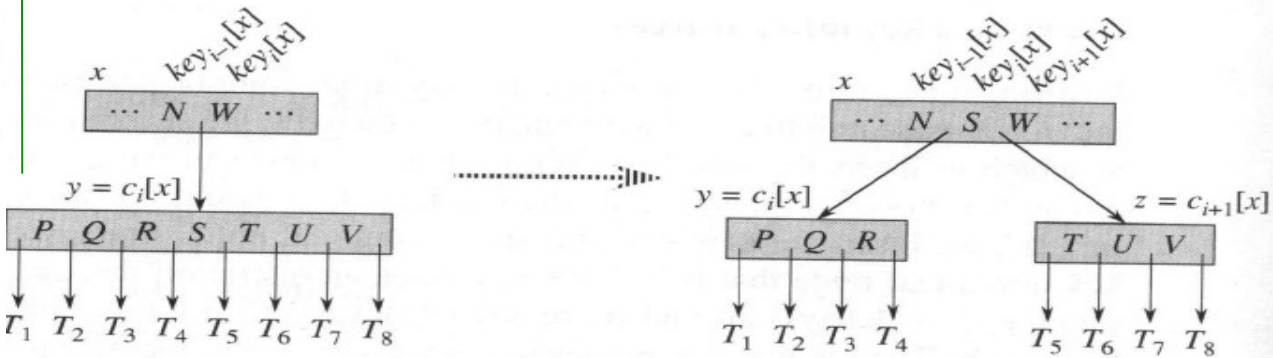
Note que nesses pseudocódigos assume-se que as chaves e filhos começam na posição 1 !!!

```
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

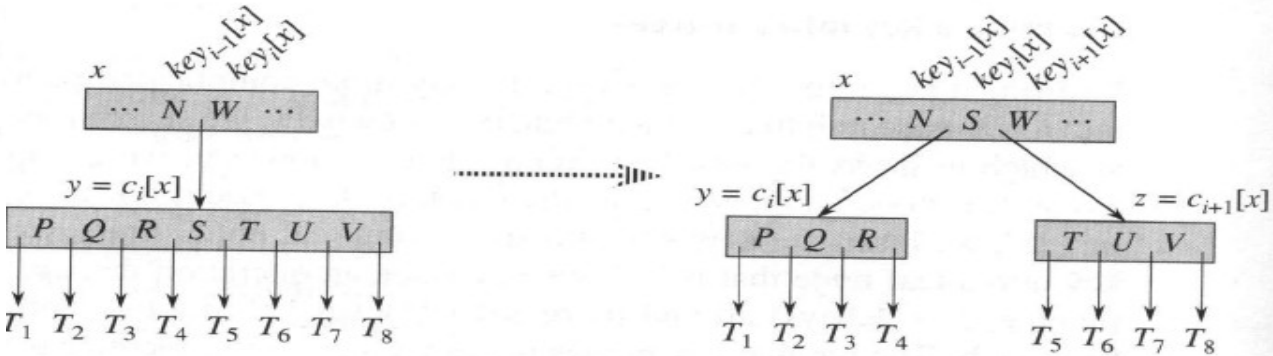
Ajusta y

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)
```

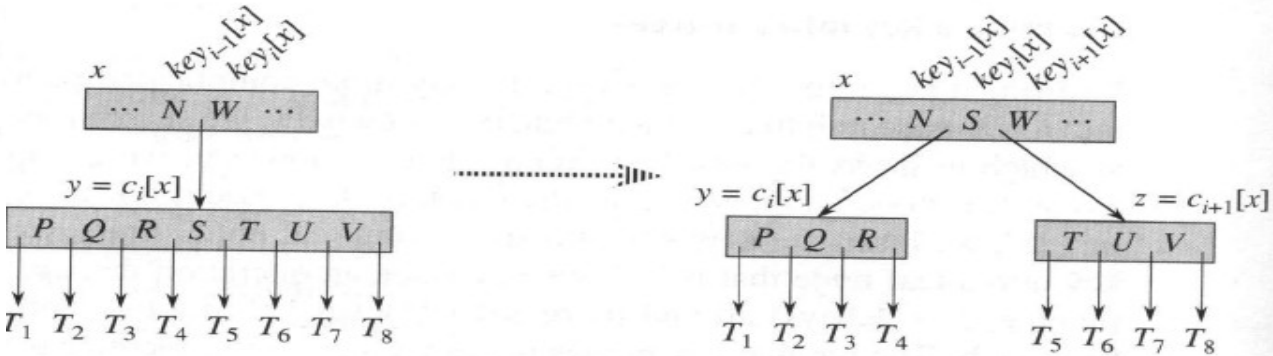
COMPLEXIDADE:



```
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:

SEEKS: O(1) – 3 mais precisamente



Continua na próxima aula

Referências

Livro do Cormen: cap 18 (3ª ed.)

Livro do Drozdek (4ª ed) cap 7