

Algoritmos e Estruturas de Dados II

Aula 22 – Árvores B

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

2025

Alocação de Blocos na Memória Secundária

- 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
 - Indexada (busca rápida, problemas na inserção e exclusão)
 - Árvores B / B+
 - *Hashing*

Alocação de Blocos na Memória Secundária

- 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
 - Indexada (busca rápida, problemas na inserção e exclusão)
 - Árvores B / B+
 - *Hashing*

Alocação de Blocos na Memória Secundária

- 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
 - Indexada (busca rápida, problemas na inserção e exclusão)
 - Árvores B / B+
 - Hashing

Lembrando de AED I

Lembrando de AED I

- Busca eficiente em dados ordenados sem gastar memória

Lembrando de AED I

- Busca eficiente em dados ordenados sem gastar memória
 - Busca Binária (em um arranjo)

Lembrando de AED I

- Busca eficiente em dados ordenados sem gastar memória
 - Busca Binária (em um arranjo)
 - Mas o problema era justamente inserção / exclusão

Lembrando de AED I

- Busca eficiente em dados ordenados sem gastar memória
 - Busca Binária (em um arranjo)
 - Mas o problema era justamente inserção / exclusão
 - Qual a solução em AED I para continuar fazendo busca binária mas permitir dinamismo de inserção / remoção?

Lembrando de AED I

- Busca eficiente em dados ordenados sem gastar memória
 - Busca Binária (em um arranjo)
 - Mas o problema era justamente inserção / exclusão
 - Qual a solução em AED I para continuar fazendo busca binária mas permitir dinamismo de inserção / remoção?
- Árvores Binárias de Busca!

Lembrando de AED I

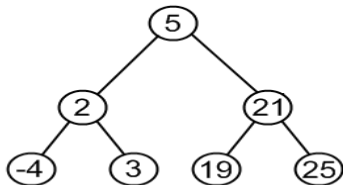
- Busca Binária (em um arranjo):
-4, 2, 3, 5, 19, 21, 25

Lembrando de AED I

- Busca Binária (em um arranjo):
-4, 2, 3, 5, 19, 21, 25
- Árvores de Busca Binária:

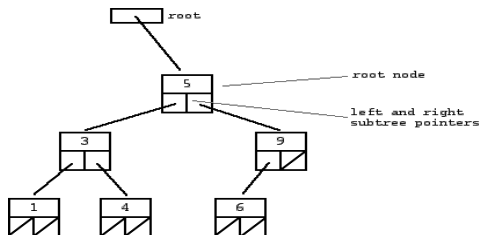
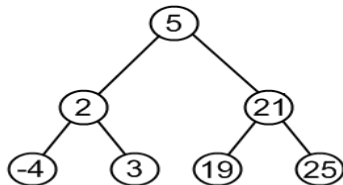
Lembrando de AED I

- Busca Binária (em um arranjo):
-4, 2, 3, 5, 19, 21, 25
- Árvores de Busca Binária:



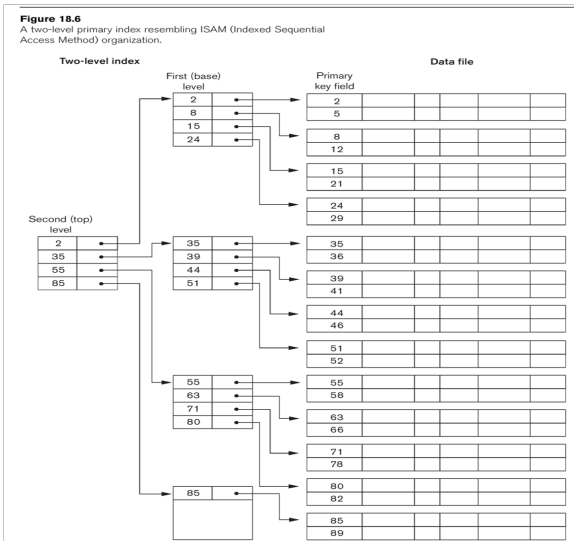
Lembrando de AED I

- Busca Binária (em um arranjo):
-4, 2, 3, 5, 19, 21, 25
- Árvores de Busca Binária:



Árvores B

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



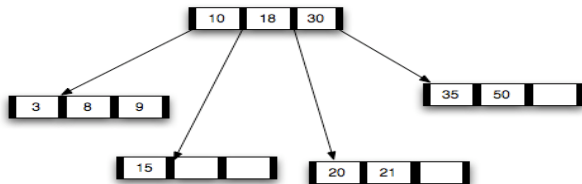
Árvores B

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

- Árvores de busca $n+1$ -árias
- N = número 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

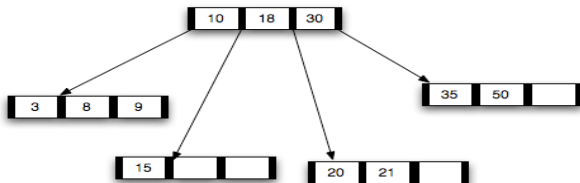
Árvores B

- 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)



Árvores 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 $key_1[x] \leq key_2[x] \leq \dots \leq key_{n[x]}[x]$;
 - $leaf[x]$, 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.



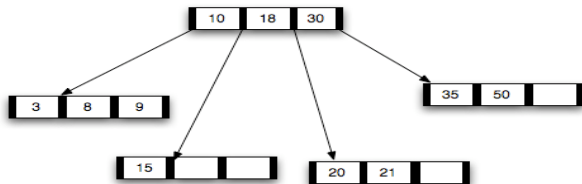
(CORMEN, LEISERSON & RIVEST, 2009)

Árvores 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

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

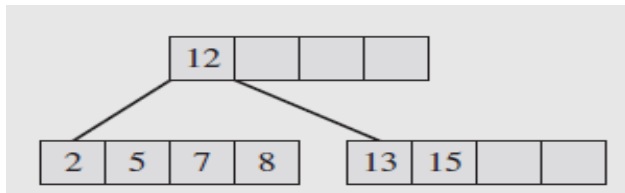
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h .



(CORMEN, LEISERSON & RIVEST, 2009)

Árvores B - Para Pensar

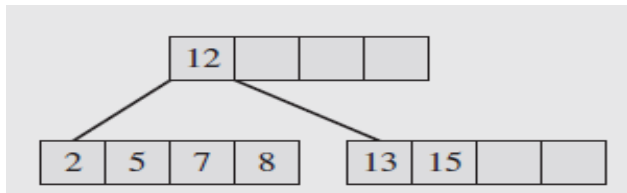
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore h



- Onde seria inserido um registro com chave igual a 1?

Árvores B - Para Pensar

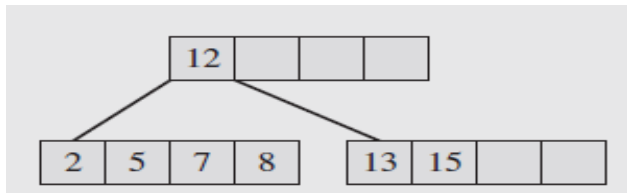
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore h



- Onde seria inserido um registro com chave igual a 1?
- Poderíamos criar um filho à esquerda de 2?

Árvores B - Para Pensar

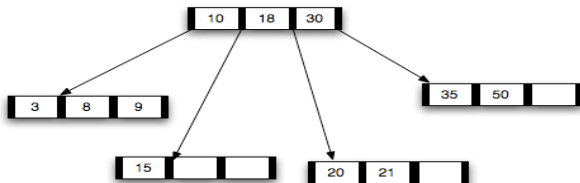
3. Todas as folhas aparecem no mesmo nível, que é a altura da árvore h



- Onde seria inserido um registro com chave igual a 1?
- Poderíamos criar um filho à esquerda de 2?
 - Isso aumentaria a altura da árvore mesmo ela tendo espaço
 - E feriria a definição 3.

Árvores 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 $2t - 1$ filhos.

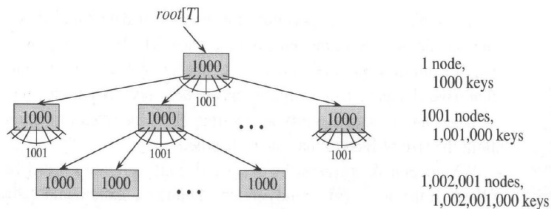


(CORMEN, LEISERSON & RIVEST, 2009)

Árvores 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.



(CORMEN, LEISERSON & RIVEST, 2009)

Árvores 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,

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

(CORMEN, LEISERSON & RIVEST, 2009)

Árvores B - Estrutura Clássica

```
#define T 2 // Grau Mínimo
```

```
typedef int bool;
```

```
typedef int TipoChave;
```

```
typedef struct {  
    TipoChave chave;  
    // OUTROS CAMPOS  
} Registro;
```

```
typedef struct auxNo{  
    int numChaves;  
    bool folha;  
    Registro regs[2*T-1];  
    struct auxNo* filhos[2*T];  
} No;
```

```
typedef struct{  
    No* raiz;  
} ArvB;
```

Árvores B - Inicialização

B-TREE-CREATE(T)

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

Árvores B - Inicialização

B-TREE-CREATE(T)

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

```
void inicializa(ArvB* a){
    No* novo = (No*) malloc(sizeof(No));
    novo->numChaves = 0;
    novo->folha = true;
    a->raiz = novo;
    salvarNoDisco(novo);
}
```

Árvores B - Busca

Busca na árvore B

- $\text{B-Tree-Search}(x, k)$: 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 $\text{key}_i[y] = k$. Caso contrário, retorna NIL .
- Chamada inicial: $\text{B-Tree-Search}(\text{root}[T], k)$.

B-TREE-SEARCH(x, k)

```
1   $i \leftarrow 1$ 
2  while  $i \leq n[x]$  and  $k > \text{key}_i[x]$ 
3      do  $i \leftarrow i + 1$ 
4  if  $i \leq n[x]$  and  $k = \text{key}_i[x]$ 
5      then return  $(x, i)$ 
6  if  $\text{leaf}[x]$ 
7      then return  $\text{NIL}$ 
8  else  $\text{DISK-READ}(c_i[x])$ 
9      return  $\text{B-TREE-SEARCH}(c_i[x], k)$ 
```

Árvores B - Busca

```
bool buscaRegistro(ArvB* a, TipoChave ch, Registro* reg){  
    return buscaRegistroAux(a->raiz, ch, reg);  
}
```

```
bool buscaRegistroAux(No* atual, TipoChave ch, Registro* reg){  
    int i = 0;  
    while (i < atual->numChaves && ch > atual->regs[i].chave) i++;  
    if (i < atual->numChaves && ch == atual->regs[i].chave){  
        *reg = atual->regs[i];  
        return true;  
    }else{  
        if(!atual->folha){  
            return buscaRegistroAux(atual->filhos[i], ch, reg);  
        }  
    }  
    return false;  
}
```

$O(\log_t(b))$ seeks

Árvores B - Inserção

Árvores B - Inserção

- As inserções ocorrem **sempre** nas folhas.

Árvores B - Inserção

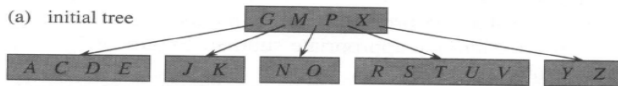
- As inserções ocorrem **sempre** nas folhas.
- Para evitar chegar em uma folha cheia (e precisar, de alguma forma, reorganizar a árvore), sempre que a inserção passar por um nó cheio ela irá **subdividi-lo**.

Árvores B - Inserção

- As inserções ocorrem **sempre** nas folhas.
- Para evitar chegar em uma folha cheia (e precisar, de alguma forma, reorganizar a árvore), sempre que a inserção passar por um nó cheio ela irá **subdividi-lo**.
 - Veremos a seguir, graficamente, como isso funciona.

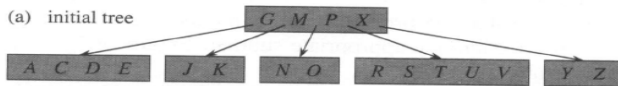
Árvores B - Inserção - $T=3$

(a) initial tree



Árvores B - Inserção - $T=3$

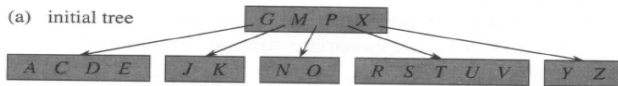
(a) initial tree



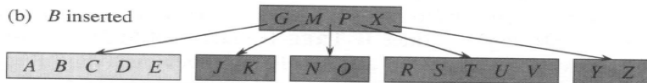
Queremos
inserir B

Árvores B - Inserção - $T=3$

(a) initial tree



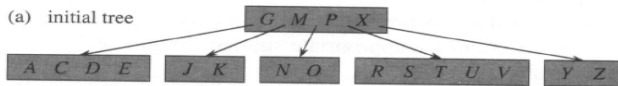
(b) *B* inserted



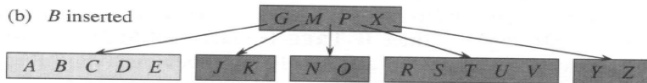
Queremos
inserir B

Árvores B - Inserção - $T=3$

(a) initial tree

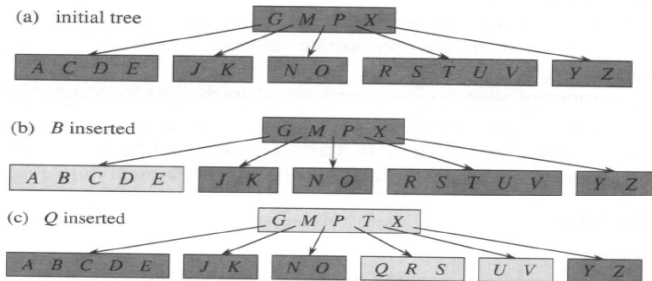


(b) *B* inserted



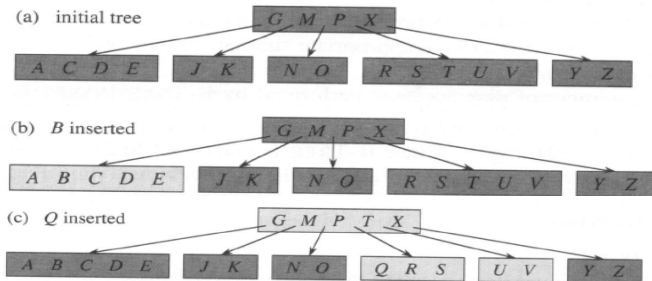
Queremos
inserir Q

Árvores B - Inserção - T=3



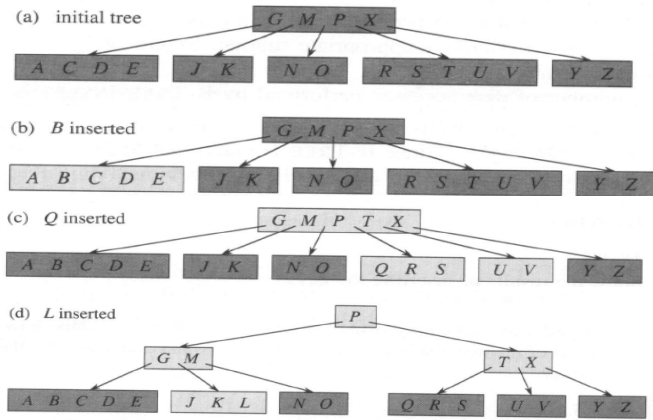
Queremos
inserir Q

Árvores B - Inserção - T=3



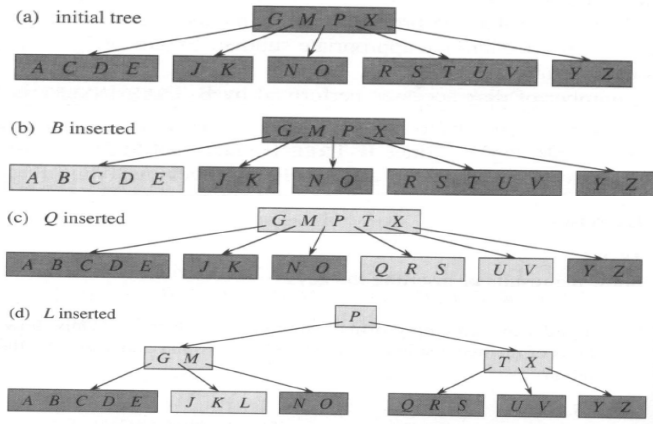
Queremos
inserir L

Árvores B - Inserção - T=3



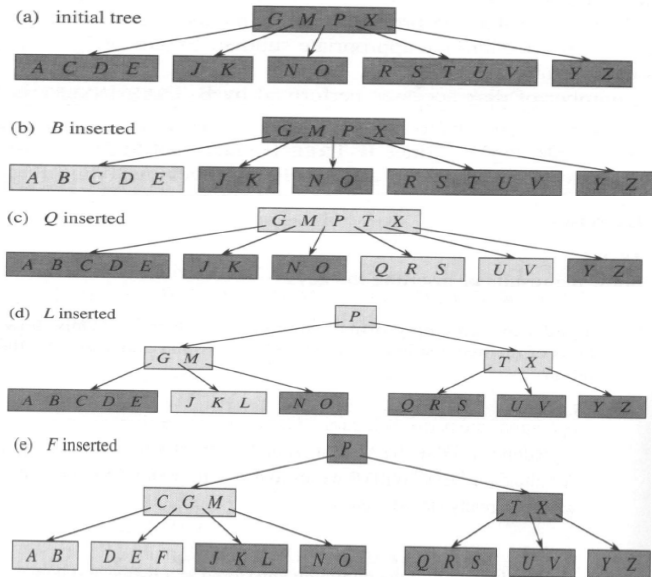
Queremos
inserir L

Árvores B - Inserção - T=3



Queremos
inserir F

Árvores B - Inserção - T=3

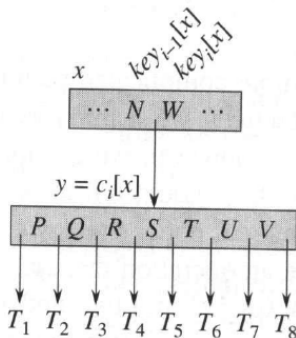


Queremos
inserir F

Árvores B - Inserção - Divisão de Nós

Como dividir um nó que está cheio?

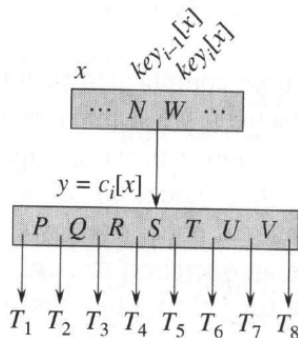
Antes:



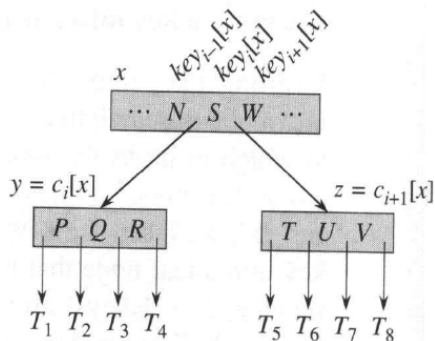
Árvores B - Inserção - Divisão de Nós

Como dividir um nó que está cheio?

Antes:



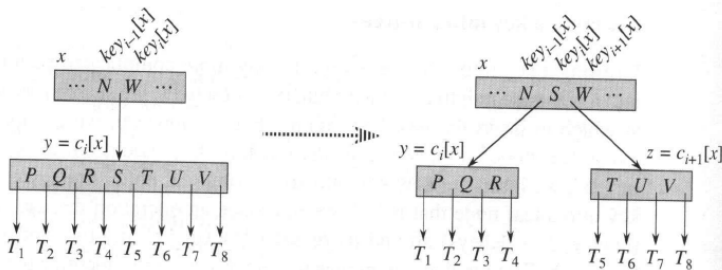
Depois:



Árvores B - Inserção - Divisão de Nós

- Divisão de um nó na árvore:

$\text{B-Tree-Split-Child}(x, i, y)$: 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.



Árvores B - Inserção - Divisão de Nós

Dividir(x , i , y):

- x : nó pai
- i : índice de y no arranjo de filhos de x
- y : nó que será dividido (i -ésimo filho de x)

Árvores B - Inserção - Divisão de Nós

Dividir(x, i, y):

- x : nó pai
- i : índice de y no arranjo de filhos de x
- y : nó que será dividido (i -ésimo filho de x)

1. Alocamos memória para o novo nó z

Árvores B - Inserção - Divisão de Nós

Dividir(x, i, y):

- x : nó pai
 - i : índice de y no arranjo de filhos de x
 - y : nó que será dividido (i -ésimo filho de x)
1. Alocamos memória para o novo nó z
 2. Ajustamos z e y (com $T - 1$ elementos em cada)

Árvores B - Inserção - Divisão de Nós

Dividir(x, i, y):

- x : nó pai
 - i : índice de y no arranjo de filhos de x
 - y : nó que será dividido (i -ésimo filho de x)
1. Alocamos memória para o novo nó z
 2. Ajustamos z e y (com $T - 1$ elementos em cada)
 3. Ajustamos x que receberá a mediana de y (antes dos ajustes)

Árvores B - Inserção - Divisão de Nós

Dividir(x, i, y):

- x : nó pai
 - i : índice de y no arranjo de filhos de x
 - y : nó que será dividido (i -ésimo filho de x)
1. Alocamos memória para o novo nó z
 2. Ajustamos z e y (com $T - 1$ elementos em cada)
 3. Ajustamos x que receberá a mediana de y (antes dos ajustes)

```
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( $x$ )
```

(CORMEN, LEISERSON &
RIVEST, 2009)

Árvores B - Inserção - Divisão de Nós

```
void divideNoFilho(No* x, int i, No* y){
    No* novo = (No*) malloc(sizeof(No));
    novo->folha = y->folha;
    novo->numChaves = T - 1;
    int j;
    for (j=0; j<T-1; j++) novo->regs[j] = y->regs[j+T];
    if (!y->folha)
        for (j=0; j<T; j++) novo->filhos[j] = y->filhos[j+T];
    y->numChaves = T - 1;
    for (j=x->numChaves; j>i; j--) x->filhos[j+1] = x->filhos[j];
    x->filhos[i+1] = novo;
    for (j=x->numChaves-1; j>=i; j--) x->regs[j+1] = x->regs[j];
    x->regs[i] = y->regs[T-1];
    x->numChaves++;
    salvarNoDisco(y);  salvarNoDisco(novo);  salvarNoDisco(x);
}
```

$O(1)$ (3 seeks)

Árvores B - Inserção

Continua na próxima aula!

Referência

- Slides baseados no material da profa. Arianne Machado Lima - ACH2024
- CORMEN, H. T.; LEISERSON, C.E.; RIVEST, R.L.
Introduction to Algorithms, MIT Press, McGraw-Hill, 2009.

Algoritmos e Estruturas de Dados II

Aula 22 – Árvores B

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

2025