Procura Externa: B-trees

Algoritmos e Estuturas de Dados

Cátia Vaz

Procura Externa: B-*trees*

- As estruturas de dados também são úteis quando lidamos com dispositivos de armazenamento externo, tais como discos rígidos.
- A troca de dados entre a memória externa e a memória principal é muito dispendiosa.
  - Esta troca é normalmente realizada através de blocos de tamanho igual designados por **páginas**.
- Para estruturas de dados grandes que não possam ser armazenadas na memória principal, pretende-se minimizar estas trocas.

Cátia Vaz

Procura Externa: B-*tree*

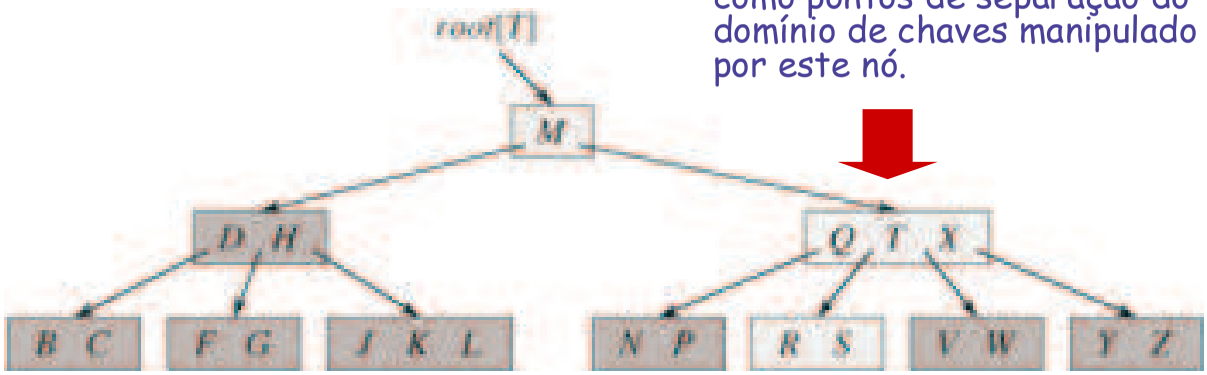
- As árvores até agora estudadas apenas contém um item por nó.
- Ler uma página para a memória do disco é dispendioso.
- O acesso à informação numa página em memória é feito em tempo constante.
  - Considerar que cada nó contém mais do que um item (ex: cada nó conter a informação de cada página que é trocada entre um dispositivo de armazenamento externo e a memória principal).
- **Objectivo: minimizar** o número de acessos a páginas.
  - O nó contém M items = tamanho da página

Procura Externa: B-*tree*

- As duas principais componentes do tempo de execução são:
  - **o número de acessos ao disco**
    - irá ser medido em termos do número de páginas de informação que necessitam de ser lidas ou escritas no disco
      - (nos algoritmos igoramos a questão o tempo de acesso ao disco, que na realidade não é constante)
  - **o tempo da CPU.**
- Os algoritmos das B-trees são desenhados de modo a que **apenas um número constante de páginas estejam na memória principal em cada momento**.
  - Assume-se que as páginas que já não estão a ser utilizadas são retiradas da memória principal pelo sistema. (nos algoritmos igoramos este tópico)

Procura Externa: B-*trees*

Este nó contém 3 chaves:
-> As chaves deste nó são usadas como pontos de separação do domínio de chaves manipulado por este nó.

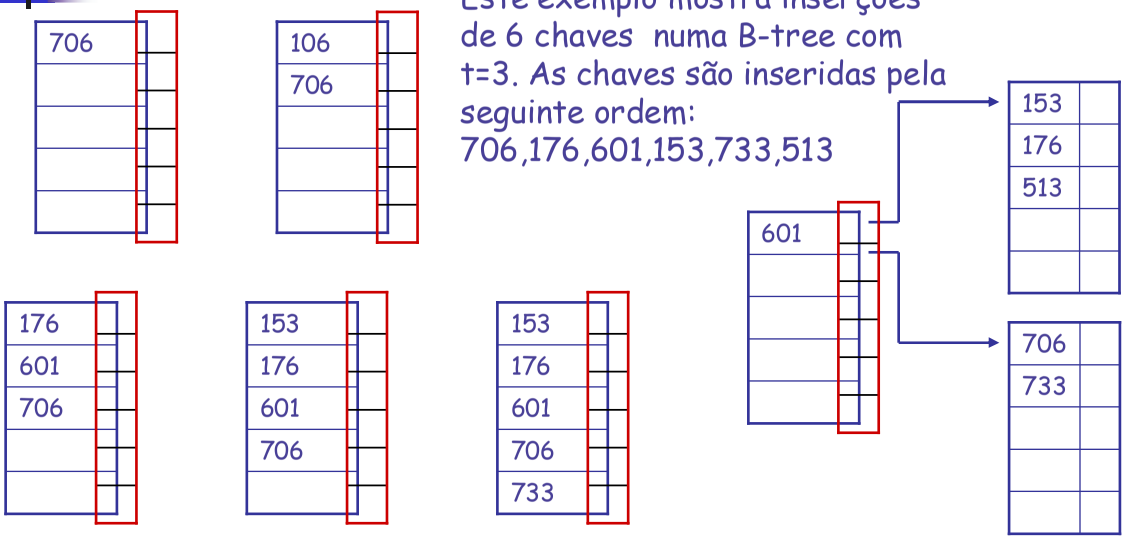


B-*tree*

- Definição: Uma **B-tree** de grau mínimo **t** (t>=2) é uma árvore com as seguintes propriedades:
  - Cada nó **x** contém:
    - O número de chaves armazenadas no nó (**n[x]**);
    - as chaves, armazenadas por ordem não decrescente, isto é $key_1[x] \leq key_2[x] \leq \cdots \leq key_n[x]$;
    - um booleano que indique se é uma folha
  - Cada nó interno **x** contém **n[x]+1** nó filhos.
  - As chaves separam o domínio das chaves armazenadas em cada subárvore
  - Todas as folhas têm altura **h**
  - Cada nó excepto a raíz tem pelo menos **t-1** chaves e no máximo **2*t-1** chaves .
  - Se a árvore é não vazia, **a raíz contém pelo menos uma chave**

Exemplo B-*tree*

Este exemplo mostra inserções de 6 chaves numa B-tree com t=3. As chaves são inseridas pela seguinte ordem:
706,176,601,153,733,513



Operações básicas em B-*tree*

- As operações básicas são: **B-TREE-SEARCH** e **B-TREE-INSERT**. Nestes procedimentos vamos adoptar duas convenções:
  - A raíz de uma B-tree está sempre na memória principal, logo uma operação **DISK-READ** à raíz nunca é necessário; caso o nó raíz seja alterado, é necessário uma operação **DISK-WRITE**.
  - Todos os nós passados como parâmetros têm de ter já tido uma operação **DISK-READ** realizada sobre eles.

B-tree: pesquisa

Em cada nó interno **x**, faz-se uma decisão **(n[x]+1)**-area.

```
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)
```

Chave **k** a ser procurada na subarvore de raíz **x**

retorna o nó **x** e um index **i** tal que **$key_i[x] = k$**.

$c_i[x]$ i-ésimo nó filho.

- O número de páginas de disco acedidas pelo algoritmo **B-TREE-SEARCH** é $\Theta(h) = \Theta(\log_t n)$, onde h é a altura da B-tree e n o número de chaves na B-tree.
- Como $n[x] < 2 * t$, o tempo tomado pela linhas do ciclo 2-3 em cada nó é $O(t)$, e o tempo total da CPU é $O(t * h) = O(t * \log_t n)$.

Cátia Vaz

B-tree – Inserir k

- A inserção de **k** é sempre numa página **folha y**.
- Se a folha **y** onde queremos inserir **k** estiver cheia, dividimos (*split*) a folha **y**, à volta da chave do meio (key_t), em duas páginas que contêm (t-1) chaves. A chave do meio é inserida na página *parent* de **y**.
  - Se o *parent* de **y** estiver cheio, também tem que ser dividido para que a chave seja inserida, e a propagação de divisões poderá ser até a raiz.
- Como inserir uma chave numa B-tree numa única passagem pela raíz da árvore até a uma folha?

B-tree – Inserir *k*

- Como inserir uma chave numa B-tree numa única passagem pela raíz da árvore até a uma folha?
  - à medida que vamos procurando na árvore pela posição onde a chave pertence, dividimos (*split*) cada nó cheio que encontrarmos (incluíndo a própria folha)
    - Assim sempre que quisermos dividir (*split*) um nó cheio **y**, é garantido que **parent[y]** não está cheio.

B-tree – Dividir (split) um nó

- O algoritmo **B-TREE-SPLIT-CHILD** recebe como parâmetros:
  - um nó interno **x** **não cheio** (assumido estar na memória principal);
  - um índex **i** ;
  - um nó **y** (também assumido estar na memória principal) tal que $y = c_i[x]$ é um nó cheio filho de x.
- O algoritmo divide o nó filho **y** em 2 e ajusta o nó **x** para conter um nó filho adicional
  - Para dividir um nó raíz cheio, primeiro faz-se o nó raíz filho de um novo nó raíz vazio

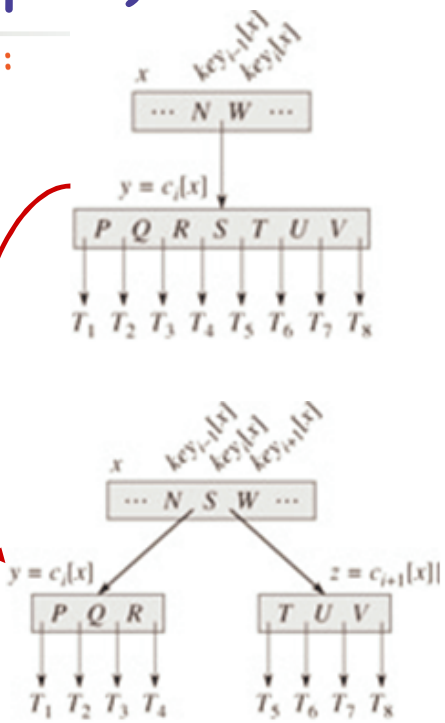
B-tree – Dividir (split) um nó

```
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)
```

insere novo filho

move a chave

Exemplo:



B-tree – Dividir (split) um nó

```
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)
```

insere novo filho

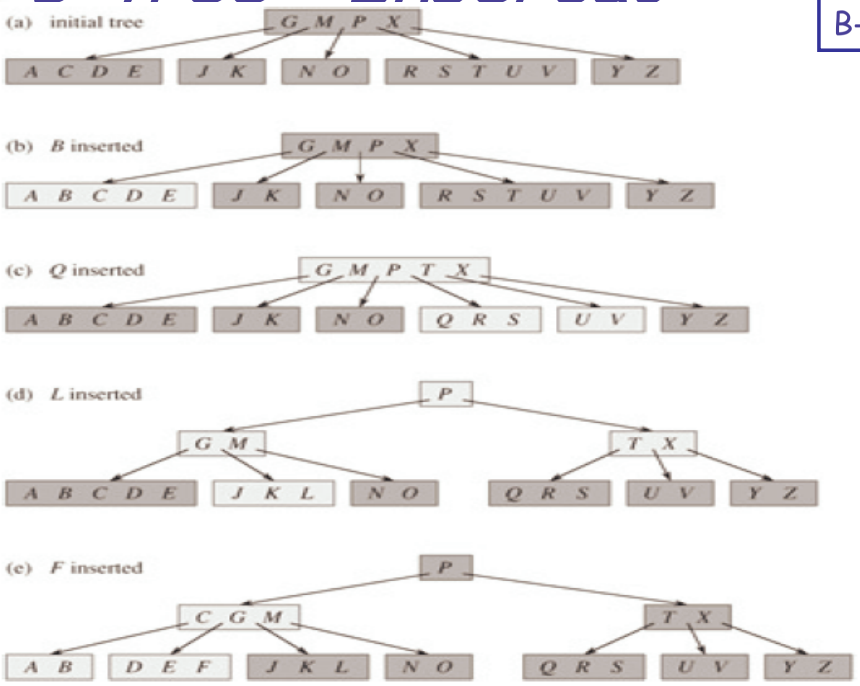
move a chave

O tempo da CPU utilizado pelo algoritmo **B-TREE-SPLIT-CHILD** é $\Theta(t)$, devido ao ciclo nas linhas 4-5 e 7-8. (Os outros ciclos são $O(t)$)

O algoritmo executa $O(1)$ operações no disco.

B-tree - Inserção

B-tree com t=3



B-tree - Inserção

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

- Para dividir o nó raíz **r** cheio, primeiro faz-se **r** filho de um novo nó raíz **s** vazio

- Inserimos uma chave k numa B-tree T de altura h numa única passagem pela árvore:
  - requere O(h) acessos ao disco
  - o tempo de CPU requerido é $O(t * h) = O(t * \log_t n)$.

B-tree - Inserção

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4             do key_{i+1}[x] ← key_i[x]
 5                i ← i - 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK-WRITE(x)
 9     else while i ≥ 1 and k < key_i[x]
10             do i ← i - 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t - 1
14            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                 if k> key_i[x]
16                   then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

se x for um nó folha

se x **não for** um nó folha, determinar o nó filho de x ao qual invocar este método

se o nó tiver cheio, dividir em dois (split)

após divisão, determinar o nó filho de x ao qual invocar este método

B-tree-Remoção

- Tem de se garantir que um nó tem pelo menos t-1 chaves (excepto o nó raíz) **(Propriedade de uma B-TREE)**
- **Algoritmo B-TREE-DELETE:**
  - **Caso 1:**

    Se a página x é uma **folha**, caso contenha a chave **k**, remove a chave **k** de **x** e retornar **true**, caso contrário retorna **false**.
  - **Caso 2:**

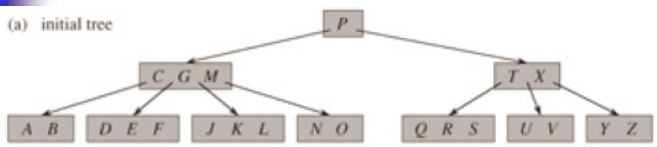
    Se a chave **k** estiver na página **x** e **x** é um página **interna**
  - **Caso 3:**

    Se a chave não estiver presente na página interna **x**

Algoritmo B-TREE-DELETE: Caso 2

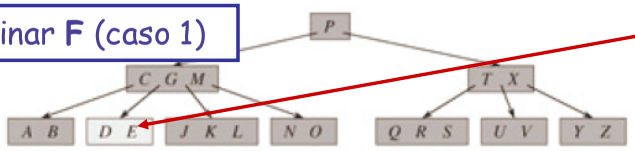
- Se a chave **k** estiver na página **x** e **x** é um página interna
  - a) **Se o filho y que precede k no nó x tem pelo menos t chaves:**
    - Procurar na sub-árvore **y** o predecessor **k'** de **k**.
    - Recursivamente remover **k'** e substituir **k** por **k' em x**.
    - (Procurar k' e eliminá-lo pode ser realizado numa única passagem)
  - b) Simetricamente., Se o filho z que sucede **k** em **x** tiver pelo menos **t** chaves:
    - Procurar o sucessor **k'** de **k** na sub-àrvore com raíz
    - Recursivamente remover **k'** e substituir **k** por **k'** .
    - (Procurar k' e eliminá-lo pode ser realizado numa única passagem)
  - c) Caso contrário, Se ambos **y** e **z** têm **(t-1)** chaves, juntar **k**, **z** em **y**:
    - **x** perde **k** e **z**;
    - **y** passa a conter **(2*t-1)** chaves.
    - Recursivamente remover **k** de **y**.

B-tree-Remoção



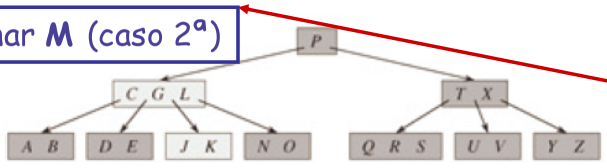
B-tree com t=3

Eliminar **F** (caso 1)



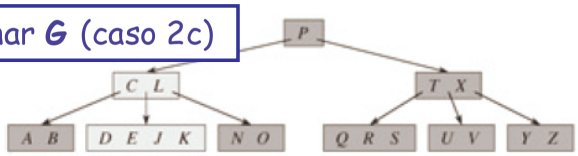
Simples eliminação num nó folha

Eliminar **M** (caso 2ª)



O predecessor de M (L) é movido para a posição de M

Eliminar **G** (caso 2c)



Cátia Vaz

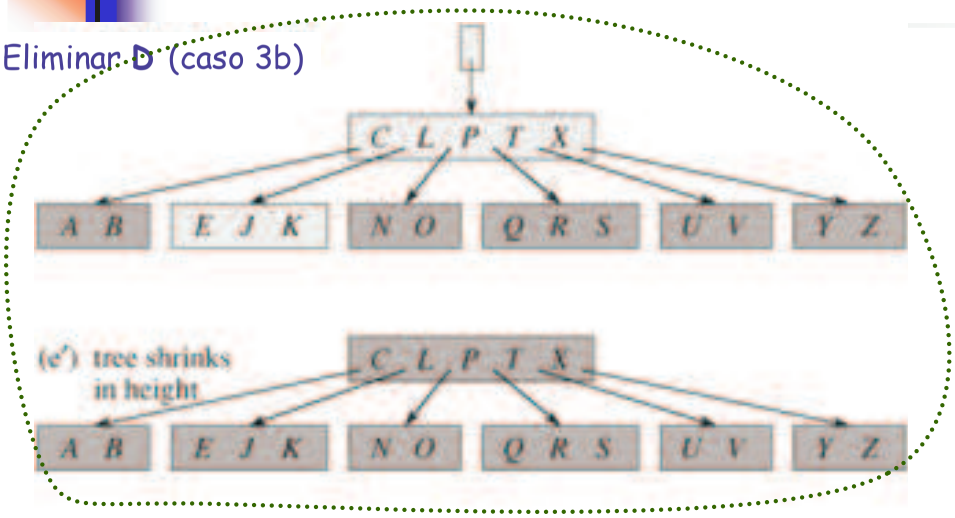
Algoritmo B-TREE-DELETE: Caso 3

- Se a **chave não estiver presente na página interna x**, determinar a sub-árvore c[i] de **x** que poderá conter **k**. Se **c[i]** de **x** só contém **(t-1)** chaves, executar 3a ou 3b de forma a garantir que se desce para uma página que contém pelo menos **t** chaves.
  a) Se **c[i]** de **x** tem (**t-1**) chaves mas um dos irmãos adjacentes têm **t** chaves, mover para **c[i]** de **x** um elemento do irmão.
  b) Se ambos os irmãos adjacentes têm **t-1** chaves, juntar **c[i]** de **x** com o irmão.

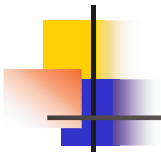
B-tree-Remoção

B-tree com t=3

Eliminar **D** (caso 3b)



Eliminar **B** (caso 3a)



B-tree-Remoção

- Este procedimento envolve **O(h)** operações no disco para uma B-tree de altura **h**.
  - (apenas são feitas **O(1)** invocações dos procedimentos **DISK-READ** e **DISK-WRITE** entre as invocações recursivas do procedimento **B-TREE-DELETE**.
- O tempo de CPU requerido é **O(t * h)** = =**O(t * log $_t$ n)**.