Árvores B

Luiz E. Buzato

29 de setembro de 2010

Instituto de Computação - UNICAMP
buzato@ic.unicamp.br

Ler observação na próxima transparência !

Nota importante

originalmente preparadas pelo Prof. Cid C. de Souza e pelo pós-graduando Alison Cruz sob supervisão do Prof. Zanoni Dias em setembro de 2007 como parte das atividades da disciplina MO637 do Instituto de Computação da UNICAMP.

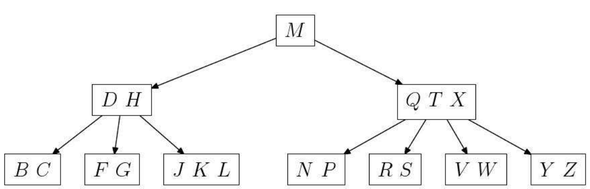
A referência básica usada nesta apresentação é o livro *"Introduction to Algorithms"* de autoria de T. Cormen, C. Leiserson, R. Rivest e C. Stein, editado pela *McGraw-Hill* em 2001.

Algumas figuras utilizadas neste documento foram extraídas do conjunto de transparências preparadas pelo Prof. Tomasz Kowaltowski para a disciplina MC202 do Instituto de Computação da UNICAMP.

Finalmente, parte do material foi aproveitado da apresentação que se encontra em www.iua.upf.es/~rramirez/TA/btrees.pdf.

Visão Geral

- As árvores *B* são generalizações de árvores binárias de busca
- Elas são balanceadas, ou seja, sua altura é $O(lg(n))$
- As árvores *B* foram desenvolvidas para otimizar o acesso a dispositivos de armazenamento secundário (ex., discos)
- Os nós da árvore *B* podem ter muitos filhos. Esse fator de ramificação elevado é determinante para reduzir o número de acessos a disco.

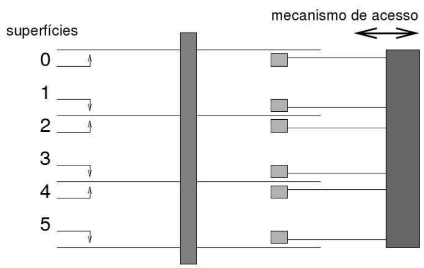


Armazenamento Secundário

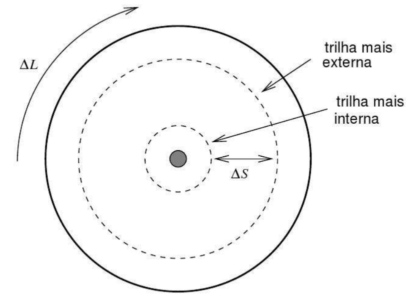
- Atualmente o armazenamento estável é feito em discos magnéticos, e o custo de cada acesso (da ordem de mili segundos) é muito alto quando comparado ao acesso à memória RAM (ordem de nano segundos)
- Toda vez que um acesso é feito, deve-se aproveitá-lo da melhor maneira possível, trazendo o máximo de informação relevante
- Tipicamente, a quantidade de dados armazenados numa árvore B é muito grande e não pode ser armazenada na memória principal de uma só vez. Por isso, os dados da árvore são paginados

Armazenamento Secundário

- Especializações são feitas de acordo com as necessidades da aplicação. O fator de ramificação, chegar à ordem de milhares (p.ex., 2048) dependendo do *buffer* dos discos e do tamanho das páginas de memória alocados pelo sistema operacional.



Armazenamento Secundário



Tempos de acesso a disco:

- busca (*seek*): ΔS
- latência: ΔL
- transferência de dados: ΔT

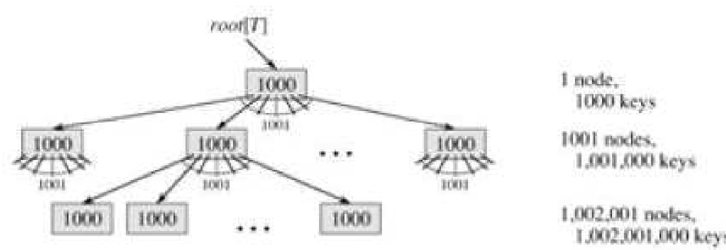
Armazenamento Secundário

- O tempo de execução de um algoritmo de árvore *B* é determinado pelas leituras e escritas no disco
- Análise de complexidade possui **duas** componentes principais: o número de acessos a disco e o tempo de CPU
- Manipulando dados em memória secundária:

```
int main() {
 T *x; /* apontador para objeto em disco */
 ...
 x = ...;        /* x recebe endereço */
 /* lê dados do objeto apontado por x para memória princ. */
 DISK-READ(x);
 /* comandos que acessam/modificam campos de x */
 ...
 /* grava informações de volta no disco */
 DISK-WRITE(x);
 ...
 return 0;
} /* main */
```


Armazenamento Secundário

- Um fator de ramificação alto reduz drasticamente a altura da árvore ($\equiv$ # acessos a disco).
  Por exemplo, se tivermos um fator de ramificação 1000 e cerca de um **bilhão** de chaves, precisaremos de apenas $log_{1000}(10^6) \approx 3$ acessos a disco (contra ≈ 32 para uma ávore binária)



Propriedades da árvore B

Seja T uma árvore B com raiz ($root[T]$). Ela possuirá então as seguintes propriedades:

1. Todo o nó x tem os campos:
   a. $n[x]$: o número de chaves atualmente armazenadas em x,
   b. as $n[x]$ chaves armazenadas em ordem crescente, i.e.,
   $$key_0[x] \leq key_1[x] \leq \ldots \leq key_{n[x]-1}[x]$$
   c. $leaf[x]$: um valor *booleano* que vale TRUE se x é uma folha e FALSE se x é um nó interno
2. Cada nó interno x também contém $n[x]+1$ apontadores $c_0[x], c_1[x], \ldots, c_{n[x]}[x]$ para os filhos. As folhas têm todos seus apontadores nulos
3. Todas as folhas têm a mesma profundidade, que é a altura da árvore: h

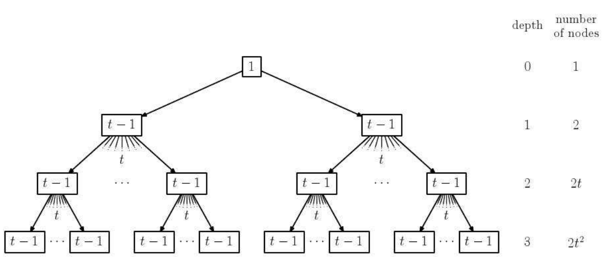
Propriedades da árvore B

1. As chaves $key_i[x]$ separam os intervalos de chaves armazenadas em cada sub-árvore. Assim, se k_i é uma chave armazenada na sub-árvore com raiz $c_i[x]$, então:
   $$k_0 \leq key_0[x] \leq k_1 \leq key_1[x] \leq \ldots \leq key_{n[x]-1}[x] \leq k_{n[x]}$$
2. Existem limites superiores e inferiores para o número de chaves num nó. Eles são expressos em termos de um inteiro fixo $t \geq 2$ chamado **grau mínimo** da árvore:
   a. Todo nó que não seja raiz deve ter pelo menos $t-1$ chaves. Portanto, todo nó interno que não seja a raiz tem pelo t ou mais filhos. Se a árvore for não vazia, a raiz deve ter pelo menos uma chave
   b. Cada nó pode conter no máximo $2t-1$ chaves. Portanto, um nó interno, pode ter no máximo $2t$ filhos. O nó é dito estar **cheio** quando ele contém exatamente $2t-1$ chaves

Propriedades da árvore B

Teorema:
Seja T uma árvore B de altura h e grau mínimo $t \geq 2$ contendo $n \geq 1$ chaves. Então (considerando a raiz no nível zero),
$$h \leq \log_t \frac{n+1}{2}.$$

Prova: ...



Operações básicas em árvores B

1. Veremos inicialmente três operações básicas em árvores B: B-TREE-CREATE, B-TREE-SEARCH e B-TREE-INSERT.
2. As convenções adotadas nestes procedimentos são:
   - A raiz está sempre na memória principal, portanto, não há necessidade de fazer um DISK-READ. Por outro lado, se o nó raiz mudar, será necessário fazer um DISK-WRITE
   - Quaisquer nós passados como parâmetros já devem ter sofrido um DISK-READ.

B-TREE-CREATE(T)

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

$O(1)$ acessos a disco e $O(1)$ de tempo de CPU

Busca por Elemento

- A busca em uma árvore B é similar à busca em uma árvore binária, só que ao invés de uma bifurcação em cada nó, temos vários caminhos a seguir de acordo com o número de filhos do nó e a chave procurada
- A função B-TREE-SEARCH recebe o apontador para o nó raiz (x) e a chave k sendo procurada
- Se a chave k pertencer à árvore o algoritmo retorna o nó ao qual ela pertence e o índice dentro do nó correspondente à chave procurada, caso contrário, retorna NIL

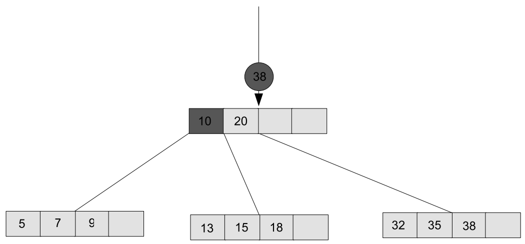
B-TREE-SEARCH: pseudo-código

B-TREE-SEARCH(x, k)

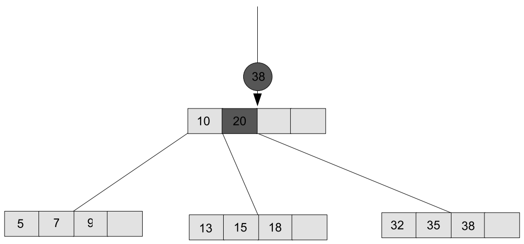
```
i ← 0
while i < n[x] and k > key_i[x] do i ← i + 1
if i < n[x] and k = key_i[x] then return (x, i)
if leaf[x] then return NIL
else DISK-READ(c_i[x])
    return B-TREE-SEARCH(c_i[x], k)
```

- Como dito anteriormente, o número de acessos a disco é $O(\log_t(n))$, onde n é o número de chaves na árvore
- Como em cada nó, é feita uma busca linear, temos um gasto de $O(t)$ em cada nó. Sendo assim, o tempo total é de $O(t \log_t(n))$

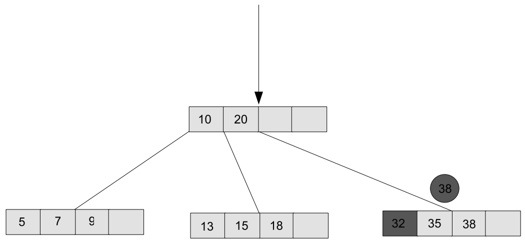
Busca por Elemento: Exemplo



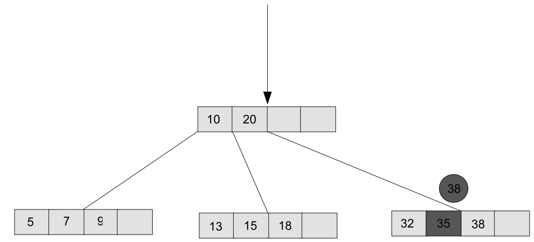
Busca por Elemento: Exemplo



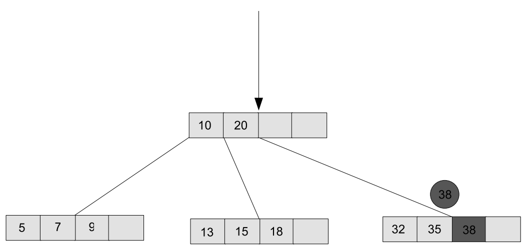
Busca por Elemento: Exemplo



Busca por Elemento: Exemplo



Busca por Elemento: Exemplo

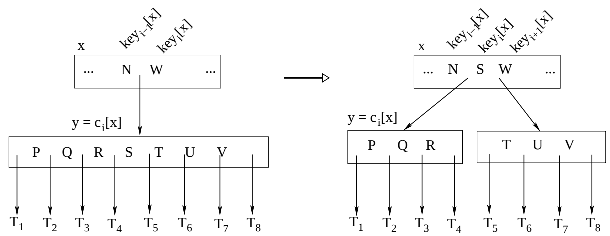


Inserção de elemento: a operação *split*

- A inserção nas árvores *B* é relativamente mais complicada, pois, precisamos inserir a nova chave no nó correto da árvore, sem violar suas propriedades
- Como proceder se o nó estiver cheio ?
- Caso o nó esteja cheio, devemos separar (split) o nó ao redor do **elemento mediano**, criando 2 novos nós que não violam as definições da árvore
- O elemento mediano é promovido, passando a fazer parte do nó pai daquele nó

Inserção de elemento: a operação *split*

Exemplo: $t = 4$



- O procedimento B-TREE-SPLIT-CHILD recebe como parâmetros um nó interno (não cheio) x, um índice i e um nó y tal que $y = c_i[x]$ é um filho de x que está cheio
- Ele cria um novo nó z, separa o nó y ao redor do elemento mediano, copiando os elementos maiores que ele em z, deixando os menores em y, ajusta o contador de elementos de z e y para $t-1$, e promove o elemento mediano para o nó x

B-TREE-SPLIT-CHILD: pseudo-código e 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 ← 0 to t − 2 do
5      key_j[z] ← key_{j+t}[y]
6  if not leaf[y] then
7      for j ← 0 to t − 1 do
8          c_j[z] ← c_{j+t}[y]
9  n[y] ← t − 1
10 for j ← n[x] downto i + 1 do
11     c_{j+1}[x] ← c_j[x]
12 c_{i+1}[x] ← z
13 for j ← n[x] − 1 downto i do
14     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)
```

O tempo de CPU é $O(t)$ por causa dos laços 4–5, 7–8, 10–11 e 13–14. O número de acessos a disco é constante, ou seja, $O(1)$

Inserção em árvores *B*

- A nova chave **<u>sempre</u>** é inserida em uma folha
- A inserção é feita em um único percurso na árvore, a partir da raiz até uma das folhas
- O procedimento B-TREE-SPLIT-CHILD é usado para garantir que a recursão **<u>nunca</u>** desce em um nó cheio
- O código a seguir faz uso do procedimento B-TREE-INSERT-NONFULL, que é responsável pela inserção da chave em um nó **<u>não</u>** cheio
- B-TREE-INSERT-NONFULL insere a chave k no nó x, caso este seja uma folha, caso contrário, procura o filho adequado e desce a ele recursivamente até encontrar a folha onde deve inserir k

B-TREE-INSERT: pseudo-código

B-TREE-INSERT(T, k)

```
r ← root[T]
if n[r] = 2t − 1 then
    s ← ALLOCATE-NODE()
    root[T] ← s
    leaf[s] ← FALSE
    n[s] ← 0
    c_0[s] ← r
    B-TREE-SPLIT-CHILD(s, 0, r)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

Observação: o *split* na raiz é o único jeito de aumentar a altura da árvore *B*. Ao contrário das árvores binárias, o crescimento se dá na raiz em vez das folhas.

B-TREE-INSERT-NONFULL: pseudo-código

B-TREE-INSERT-NONFULL(x, k)

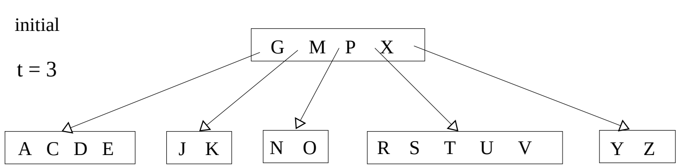
$i \leftarrow n[x] - 1$
if $leaf[x]$ then
  while $i \geq 0$ and $k < key_i[x]$ do
    $key_{i+1}[x] \leftarrow key_i[x]$
    $i = i - 1$
  $key_{i+1}[x] \leftarrow k$; $n[x] \leftarrow n[x] + 1$; DISK-WRITE(x);
else
  while $i \geq 0$ and $k < key_i[x]$ do $i \leftarrow i - 1$;
  $i \leftarrow i + 1$; DISK-READ$(c_i[x])$
  if $n[c_i[x]] = 2t - 1$ then
    B-TREE-SPLIT-CHILD$(x, i, c_i[x])$
    if $k > key_i[x]$ then $i \leftarrow i + 1$
  B-TREE-INSERT-NONFULL$(c_i[x], k)$

Complexidade de inserção

- O # de acessos a disco de B-TREE-INSERT é $O(h)$ pois apenas $O(1)$ operações DISK-READ/WRITE são feitas entre duas chamadas consecutivas de B-TREE-INSERT-NONFULL
- O tempo total de CPU é $O(th) = O(t \log_t n)$.
- Note que o procedimento B-TREE-INSERT-NONFULL apresenta uma **recursão caudal**. Esta recursão pode ser removida usando um laço **while**, com o qual fica mais claro perceber que o número de páginas que devem estar em memória principal a qualquer instante é $O(1)$.

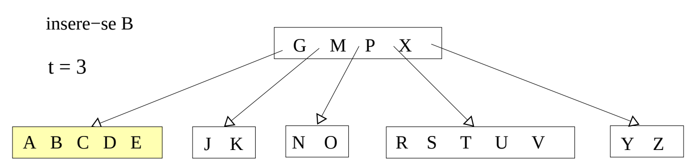
Inserção em árvores B

Início



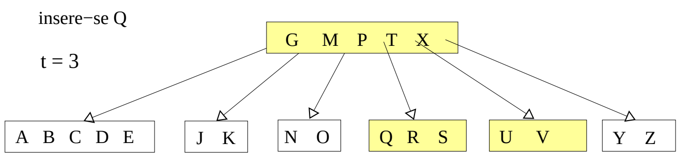
Inserção em árvores B

Insere B



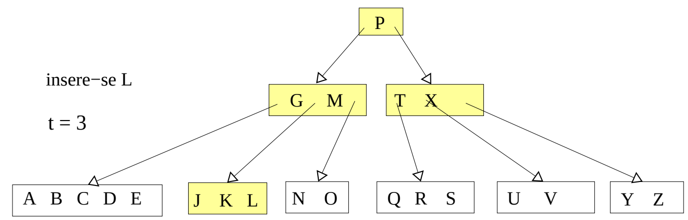
Inserção em árvores B

Insere Q



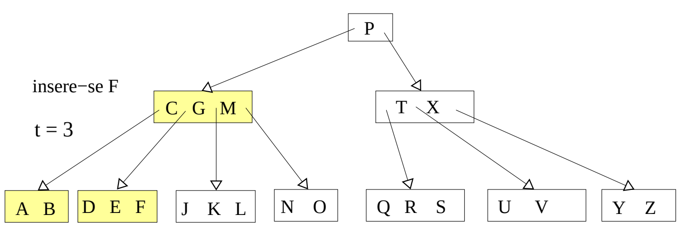
Inserção em árvores B

Insere L



Inserção em árvores B

Insere F



Remoção de Chaves

- Contrariamente ao que ocorre na inserção, a remoção de uma chave pode ser feita em qualquer nó
- Assim como na inserção, precisamos garantir que, ao removermos uma chave as propriedades da árvore B serão preservadas
- Da mesma maneira que tivemos de garantir que a inserção não ocorresse em um nó cheio, no caso da remoção, devemos assegurar que ela não aconteça em um nó *vazio demais*, ou seja, com $t-1$ chaves

Remoção de Chaves

Estratégia do procedimento de remoção:

- sempre que o procedimento B-TREE-DELETE for chamado recursivamente em um nó x, devemos ter $n[x] \geq t$, sendo t o grau mínimo da árvore
- note que esta condição obriga que o número de chaves no nó x seja pelo menos uma unidade maior do que o mínimo exigido pelas propriedades das árvores B.
- assim, em algumas situações, uma chave poderá ter de ser movida de x para um de seus filhos y antes que a recursão desça para y
- com isto, podemos remover uma chave fazendo uma única descida na árvore B sem precisar executar um *backtracking* (com uma única exceção a ser explicada adiante)

Remoção de Chaves

Considerações sobre o procedimento de remoção:

- a especificação da remoção dada a seguir subentende que, se o nó raiz x se tornar um nó interno vazio (i.e., sem chaves), então x será removido da árvore B e seu único filho $c_0[x]$ tornar-se-á a nova raiz da árvore
- na situação descrita acima, a árvore B decresce em altura e preserva-se a propriedade de que a raiz tem pelo menos uma chave, exceto, é claro, se a árvore ficar vazia
- a seguir são discutidos os seis casos a serem considerados para a remoção de uma chave em uma árvore B

Remoção de Chaves

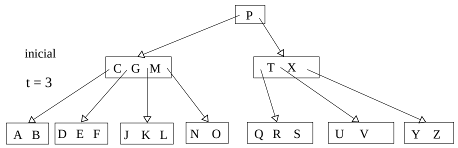
- **Caso 1.** Se a chave k estiver numa folha da árvore que possui pelo menos t chaves, remove-se a chave daquele nó
- **Caso 2.** Se a chave k está num nó interno x, faz-se o seguinte:
  - **a.** Se o filho y que precede k no nó x possui pelo menos t chaves, encontre o predecessor k' de k na sub-árvore com raiz em y. Recursivamente, remova k' de y e substitua k por k' no nó x
  - **b.** Simetricamente, se o filho z que sucede k no nó x possui pelo menos t chaves, encontre o sucessor k' de k na sub-árvore com raiz em z. Recursivamente, remova k' de z e substitua k por k' no nó x
  - **c.** Caso ambos y e z possuam somente $t-1$ chaves, intercale a chave k e todas as chaves de z no nó y, de modo que x perde tanto a chave k quanto o ponteiro para z e $n[y]$ passe a valer $2t-1$ (y fica cheio). Libere a memória ocupada por z e, recursivamente, remova k do nó y.

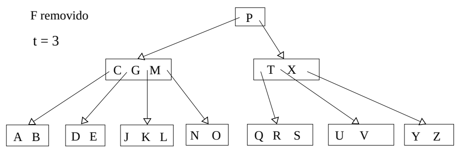
Remoção de Chaves

- **Caso 3.** Se a chave k não pertence ao nó interno x, determine a sub-árvore $c_i[x]$ que pode conter k. Caso $c_i[x]$ possua só $t-1$ chaves, execute os subcasos **3a** ou **3b** abaixo, conforme a necessidade, de modo a garantir que o procedimento descerá para um nó com pelo menos t chaves:
  - **a.** [EMPRÉSTIMO DE CHAVES] Se $c_i[x]$ possui $t-1$ chaves mas tem um irmão adjacente y com pelo menos t chaves, mova para $c_i[x]$ a chave de x cujo valor encontra-se entre aqueles das chaves de $c_i[x]$ e y. Em seguida mova uma chave de y (a menor se y for irmão direito de $c_i[x]$, a maior se for irmão esquerdo) para x e mova o apontador de filho apropriado de y para $c_i[x]$
  - **b.** Se $c_i[x]$ e ambos os seus irmãos à esquerda e à direita possuem $t-1$ chaves, una (intercale) $c_i[x]$ com um dos irmãos, o que envolve mover (para baixo) uma chave de x para o novo nó que acabou de ser formado, chave esta que ocupará o elemento mediano daquele nó

Remoção de Chaves

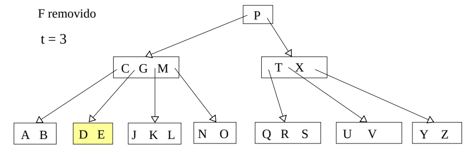
Exemplo 1: $t = 3$

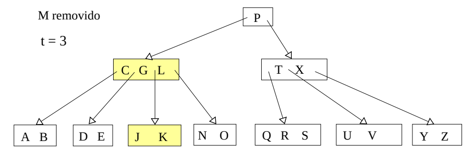




Remoção de Chaves

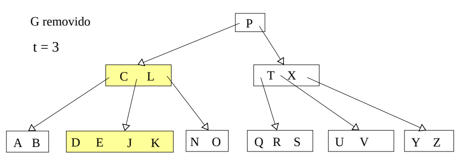
Exemplo 1: $t = 3$

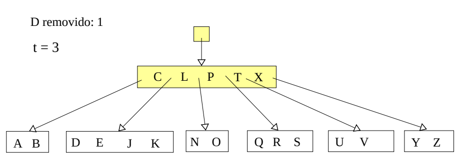




Remoção de Chaves

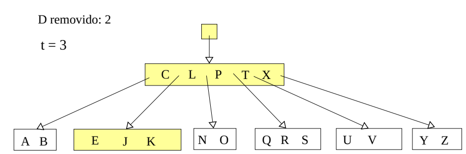
Exemplo 1: $t = 3$

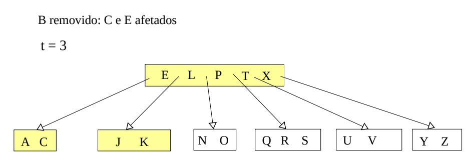




Remoção de Chaves

Exemplo 1: $t = 3$





Complexidade da Remoção

- Suponhamos que antes da remoção é feita uma busca para garantir que a chave k pode de fato ser removida da árvore B. Como vimos, o tempo consumido nesta operação é $O(t \log_t n)$
- No pior caso, teremos todos os nós da árvore com $t - 1$ elementos, exceto possivelmente a raiz, forçando a intercalação de nós e/ou o empréstimo de chaves entre nós toda vez que a recursão for descer um nível na árvore.
- Nesta situação, somente os casos 2c e 3b poderão ocorrer. Vamos analisá-los então.

Complexidade da Remoção

- Se o caso 2c ocorrer uma vez, então a chave k foi encontrada no nó x corrente e será movida para um nó y, filho de x. A recursão irá remover k de y e, é claro, recai-se novamente no caso 2c. Isto irá se propagar até atingirmos uma folha, recaindo-se no caso 1.

  Cada intercalação realizado no caso 2c envolve não mais que dois acessos a disco e tempo de CPU $O(t)$ (busca da chave mais intercalação propriamente dita).
- Se for o caso 3b, as intercalações vão ocorrendo de modo semelhante ao que foi descrito acima, só que o caso 3b poderá ir se repetindo até que se chegue numa folha ou até que a chave seja encontrada e o caso 2c ocorra, voltando-se então à situação do item anterior.

  A análise do número de acessos a disco e do tempo de CPU consumido é análgo àquela feita acima.

Complexidade da Remoção

Como a altura da árvore é $O(\log_t n)$, conclui-se que a complexidade da remoção é dada por $O(t \log_t n)$.