Árvores B

MO637 – Complexidade de Algoritmos II

14 de setembro de 2007

Overview

Overview

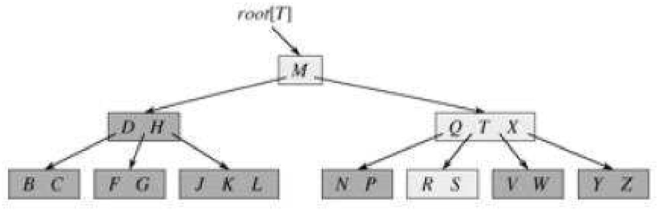
- São árvores balanceadas, desenvolvidas para otimizar o acesso a armazenamento secundário

Overview

- São árvores balanceadas, desenvolvidas para otimizar o acesso a armazenamento secundário
- Os nós da árvore B podem ter muitos filhos. Esse fator de ramificação é determinante para reduzir o número de acessos a disco. Árvores B são balanceadas, ou seja, sua altura é $O(lg(n))$

Overview

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



Armazenamento Secundário

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)

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 aproveita-lo da melhor maneira possível, trazendo o máximo de informação relevante

Armazenamento Secundário

Armazenamento Secundário

- A quantidade de dados utilizados numa árvore B óbviamente não cabem na memória de uma só vez, por isso é necessário paginá-la

Armazenamento Secundário

- A quantidade de dados utilizados numa árvore B óbviamente não cabem na memória de uma só vez, por isso é necessário paginá-la
- Especializações são feitas de acordo com as necessidades da aplicação. O fator de ramificação, por exemplo, pode variar de 3 a 2048 por exemplo (dependendo do buffer dos discos e do tamanho das páginas de memória alocados pelo SO)

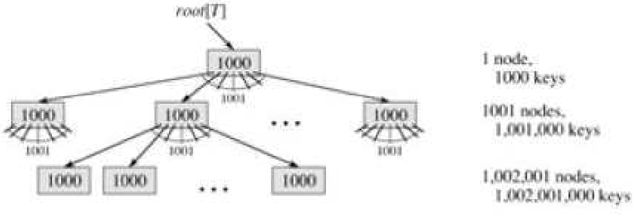
Armazenamento Secundário

Armazenamento Secundário

- Na grande maioria dos sistemas, o tempo de execução de um algoritmo de árvore-B é determinado pelas leituras e escritas no disco

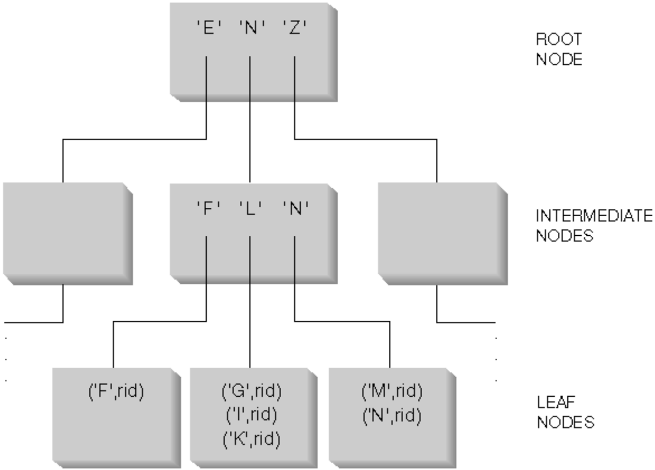
Armazenamento Secundário

- Na grande maioria dos sistemas, o tempo de execução de um algoritmo de árvore-B é determinado pelas leituras e escritas no disco
- Um fator de ramificação alto reduz drasticamente a altura da arvore. Tomemos o exemplo:



Definição da árvore-B

Consideraremos que o os dados dos registros sejam guardados junto com a chave da árvore. Se estivéssemos usando uma árvore B+, os registros ficariam todos nas folhas:



Definição 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 seguintes campos:

Definição 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 seguintes campos:
   - a. $n[x]$, o número de chaves atualmente guardadas no nó x,

Definição 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 seguintes campos:
   - a. $n[x]$, o número de chaves atualmente guardadas no nó x,
   - b. As $n[x]$ chaves, guardadas em ordem crescente, tal que $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$,

Definição 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 seguintes campos:
   - a. $n[x]$, o número de chaves atualmente guardadas no nó x,
   - b. As $n[x]$ chaves, guardadas em ordem crescente, tal que $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$,
   - c. $leaf[x]$, Um valor booleano, TRUE se x é uma folha e FALSE se x é um nó interno

Definição da árvore-B

Definição da árvore-B

- 2. Cada nó interno x também contém $n[x]+1$ apontadores $c_1[x], c_2[x]$, ...,$c_{n[x]+1}[x]$ para os filhos. As folhas tem seu apontador nulo

Definição da árvore-B

- 2. Cada nó interno x também contém $n[x]+1$ apontadores $c_1[x], c_2[x]$, ...,$c_{n[x]+1}[x]$ para os filhos. As folhas tem seu apontador nulo
- 3. As chaves $key_i[x]$ separam os intervalos de chaves guardadas em cada sub-árvore: se k_i é uma chave guardada numa sub-árvore com raiz $c_i[x]$, então: $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq ... \leq key_{n[x]}[x] \leq k_{n[x]+1}$

Definição da árvore-B

- 2. Cada nó interno x também contém $n[x] + 1$ apontadores $c_1[x], c_2[x]$, ...,$c_{n[x]+1}[x]$ para os filhos. As folhas tem seu apontador nulo
- 3. As chaves $key_i[x]$ separam os intervalos de chaves guardadas em cada sub-árvore: se k_i é uma chave guardada numa sub-árvore com raiz $c_i[x]$, então: $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq ... \leq key_{n[x]}[x] \leq k_{n[x]+1}$
- 4. Todas as folhas têm a mesma profundidade, que é a altura da árvore: h

Definição da árvore-B

5. Existem limites superiores e inferiores para o número de chaves num nó. Estes limites podem ser expressados em termos de um inteiro fixo $t \geq 2$ chamado grau mínimo:

5. Existem limites superiores e inferiores para o número de chaves num nó. Estes limites podem ser expressados em termos de um inteiro fixo $t \geq 2$ chamado grau mínimo:

- a. Todo nó que não seja raiz deve ter pelo menos $t - 1$ chaves. Todo nó interno que não a raiz tem portanto t filhos. Se a árvore for não vazia, a raiz deve ter pelo menos uma chave

Definição da árvore-B

5. Existem limites superiores e inferiores para o número de chaves num nó. Estes limites podem ser expressados em termos de um inteiro fixo $t \geq 2$ chamado grau mínimo:

- a. Todo nó que não seja raiz deve ter pelo menos $t - 1$ chaves. Todo nó interno que não a raiz tem portanto t 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 considerado cheio quando ele contém exatamente $2t - 1$ chaves

Definição da árvore-B

Definição da árvore-B

- Podemos ver o poder da árvore-B quando comparada a outros tipos de árvores balanceadas com altura $O(log_2(n))$. No caso da árvore-B a base do logaritmo é proporcional ao fator de ramificação

Definição da árvore-B

- Podemos ver o poder da árvore-B quando comparada a outros tipos de árvores balanceadas com altura $O(log_2(n))$. No caso da árvore-B a base do logaritmo é proporcional ao fator de ramificação
- Por exemplo, se tivermos um fator de ramificação 1000 e aproximadamente 1 milhão de registros, precisaremos de apenas $log_{1000}(10^6) \cong 3$ idas ao disco

Busca por Elemento

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

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
- O algoritmo de busca na árvore é uma generalização da busca em uma árvore binária

Busca por Elemento

Busca por Elemento

- A função B-TREE-SEARCH recebe o apontador para o nó raiz (x) e a chave k sendo procurada

Busca por Elemento

- 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

B-TREE-SEARCH(x, k)

$i \leftarrow 1$
while $i \leq n[x]$ and $k > key_i[x]$ do
    $i \leftarrow i + 1$
if $i \leq n[x]$ 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($c_i[x], k$)

Busca por Elemento

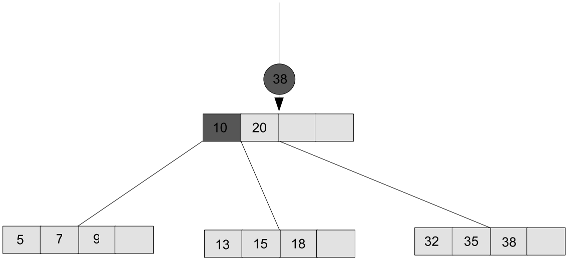
Busca por Elemento

- Como dito anteriormente, o número de acessos a disco é O($log_t(n)$) onde n é o número de chaves na árvore

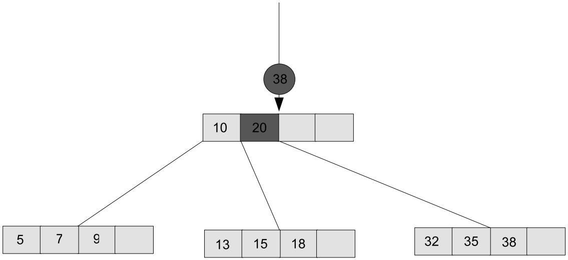
Busca por Elemento

- 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 $O(t * log_t(n))$

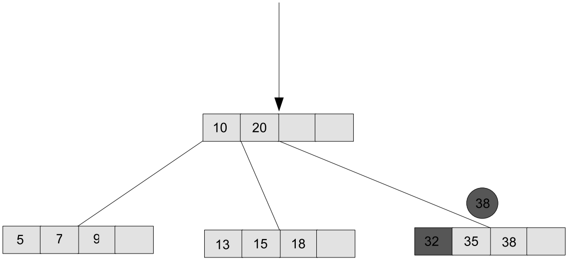
Busca por Elemento: Exemplo



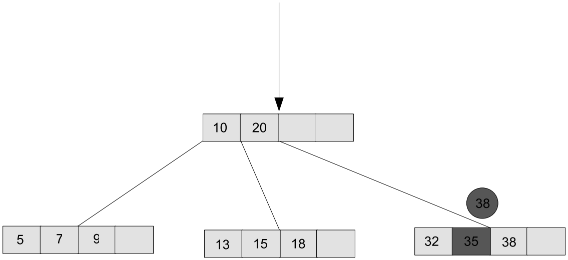
Busca por Elemento: Exemplo



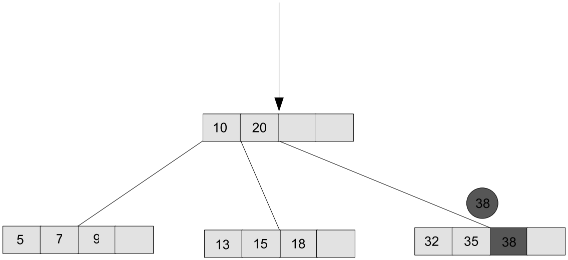
Busca por Elemento: Exemplo



Busca por Elemento: Exemplo



Busca por Elemento: Exemplo



Inserção de Elemento

Inserção de Elemento

- A inserção nas árvores-B são relativamente mais complicadas, pois, precisamos inserir a nova chave no nó correto da árvore, sem violar suas propriedades

Inserção de Elemento

- A inserção nas árvores-B são relativamente mais complicadas, pois, precisamos inserir a nova chave no nó correto da árvore, sem violar suas propriedades
- Como proceder se o nó estiver cheio?

Inserção de Elemento

- A inserção nas árvores-B são relativamente mais complicadas, 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

Inserção de Elemento

Inserção de Elemento

- O elemento mediano é promovido, passando a fazer parte do nó pai daquele nó

Inserção de Elemento

- O elemento mediano é promovido, passando a fazer parte do nó pai daquele nó
- A inserção é feita em um único percurso na árvore, a partir da raiz até uma das folhas

Separação de Nó (Split)

Separação de Nó (Split)

- 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

Separação de Nó (Split)

- 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
- O procedimento então, separa o nó 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

B-TREE-SPLIT-CHILD

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

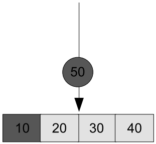
$z \leftarrow$ ALLOCATE-NODE()
$leaf[z] \leftarrow leaf[y]$
$n[z] \leftarrow t - 1$
for $j \leftarrow 1$ to $t - 1$ do
    $key_j[z] \leftarrow key_{j+t}[y]$
if not $leaf[y]$ then
    for $j \leftarrow 1$ to t do
        $c_j[z] \leftarrow c_{j+t}[y]$
$n[y] \leftarrow t - 1$
for $j \leftarrow n[x] + 1$ downto $i + 1$ do
    $c_{j+1}[x] \leftarrow c_j[x]$
(continua)

B-TREE-SPLIT-CHILD

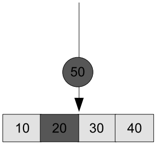
B-TREE-SPLIT-CHILD(cont.)

```
c_{i+1}[x] ← z
for j ← n[x] downto i do
    key_{j+1}[x] ← key_j[x]
key_i[x] ← key_t[y]
n[x] ← n[x] + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

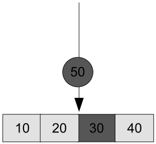
Separação de Nó (Split): Exemplo



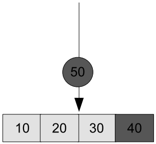
Separação de Nó (Split): Exemplo



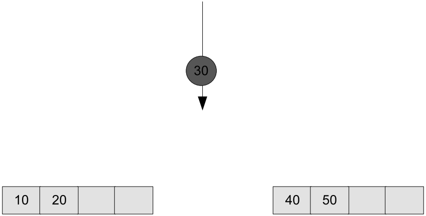
Separação de Nó (Split): Exemplo



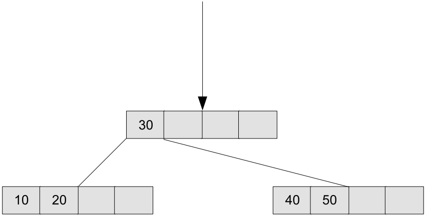
Separação de Nó (Split): Exemplo



Separação de Nó (Split): Exemplo



Separação de Nó (Split): Exemplo



Inserção com Split

Inserção com Split

- Dessa maneira, em uma única passagem pela árvore, da raiz às folhas, inserimos uma determinada chave, dividindo (splits) cada nó da árvore que encontrarmos no caminho, caso o nó esteja cheio

Inserção com Split

- Dessa maneira, em uma única passagem pela árvore, da raiz às folhas, inserimos uma determinada chave, dividindo (splits) cada nó da árvore que encontrarmos no caminho, caso o nó esteja cheio
- O código a seguir faz uso de B-TREE-INSERT-NONFULL:

B-TREE-INSERT

```
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
    c1[s] ← r
    B-TREE-SPLIT-CHILD(s, 1, r)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

Inserção com Split

B-TREE-INSERT-NONFULL insere a chave k no nó x, caso este seja uma folha, caso contrário, procura o filho adequado e desce à ele recursivamente até encontrar a folha onde deve inserir k

B-TREE-INSERT-NONFULL

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

$i \leftarrow n[x]$
if $leaf[x]$ then
    while $i \geq 1$ and $k < key_i[x]$ do
        $key_{i+1}[x] \leftarrow key_i[x]$
        $i \leftarrow i - 1$
        $key_{i+1}[x] \leftarrow k$
        $n[x] \leftarrow n[x] + 1$
        DISK-WRITE(x)
    else while $i \geq 1$ and $k < key_i[x]$ do
            $i \leftarrow i - 1$
    $i \leftarrow i + 1$
        DISK-READ($c_i[x]$)
(continua)

B-TREE-INSERT-NONFULL

B-TREE-INSERT-NONFULL(cont.)

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$(c_i[x], k)$

Inserção com Split

Inserção com Split

- Logo, o procedimento de inserção leva $O(t * log_t(n))$, onde t é o tamanho da página da árvore e n é o número total de elementos

Remoção de Chaves

Remoção de Chaves

- A remoção de uma chave é análoga à inserção, porém com alguns complicadores, já que uma chave pode ser removida de qualquer nó, seja ele raiz ou não

Remoção de Chaves

- A remoção de uma chave é análoga à inserção, porém com alguns complicadores, já que uma chave pode ser removida de qualquer nó, seja ele raiz ou não
- Assim como na inserção, precisamos garantir que, ao removermos a chave as propriedades da árvore-B não sejam violadas

Remoção de Chaves

- A remoção de uma chave é análoga à inserção, porém com alguns complicadores, já que uma chave pode ser removida de qualquer nó, seja ele raiz ou não
- Assim como na inserção, precisamos garantir que, ao removermos a chave as propriedades da árvore-B não sejam violadas
- Da mesma maneira que tivemos de garantir que um nó não se tornasse grande demais na inserção, devemos garantir que ele não torne-se pequeno demais, ou seja, deve sempre ter pelo menos $t - 1$ elementos

Remoção de Chaves

Remoção de Chaves

- Sendo assim, seguiremos para os casos de remoção de chaves

Remoção de Chaves

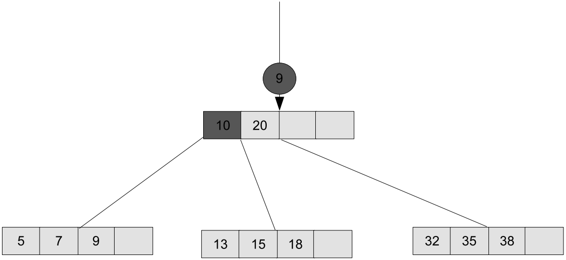
- Sendo assim, seguiremos para os casos de remoção de chaves
- Existem 6 casos possíveis para a remoção de uma chave de uma árvore-B:

Remoção de Chaves

- Sendo assim, seguiremos para os casos de remoção de chaves
- Existem 6 casos possíveis para a remoção de uma chave de uma árvore-B:
- Caso 1. Se a chave k estiver numa folha da árvore e a folha possui pelo menos t chaves,remove-se a chave da árvore

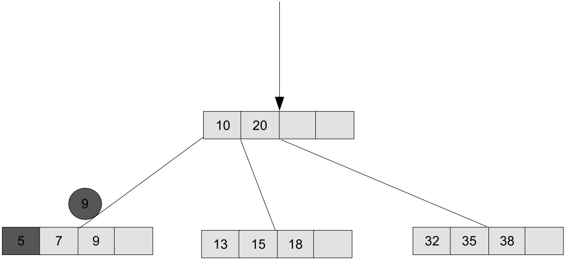
Remoção de Chaves

Caso1:

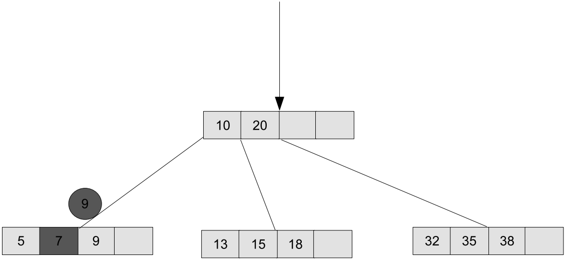


Remoção de Chaves

Caso1:

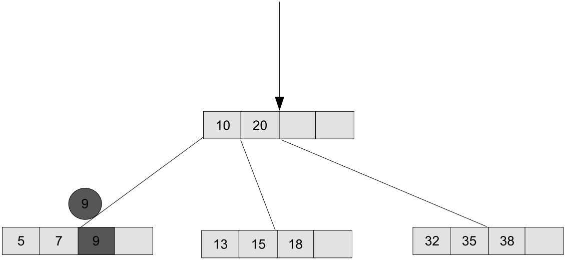


Caso1:

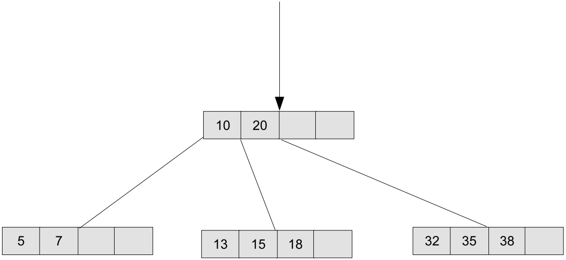


Remoção de Chaves

Caso1:



Caso1:



Remoção de Chaves

Caso 2. Se a chave k está num nó interno x o seguinte deve ser feito:

Remoção de Chaves

Caso 2. Se a chave k está num nó interno x o seguinte deve ser feito:

- 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. Remova k' do nó filho e substitua k por k' no nó atual

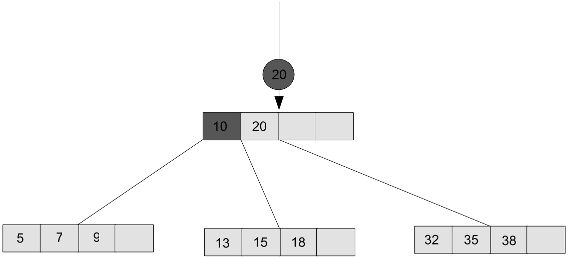
Remoção de Chaves

Caso 2. Se a chave k está num nó interno x o seguinte deve ser feito:

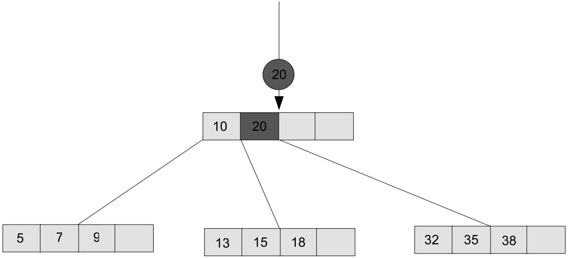
- a. Se o filho y que precede k no nó x possui pelo menos t chaves, encontre o predecessor k' de k na sub-árvore com raiz em y. Remova k' do nó filho e substitua k por k' no nó atual
- b. Simetricamente, se o filho z que sucede k no nó x possui pelo menos t chaves, encontre o sucessor k' de k na sub-árvore com raiz em z. Remova k' do nó filho e substitua k por k' no nó atual

Remoção de Chaves

Caso2 A/B:

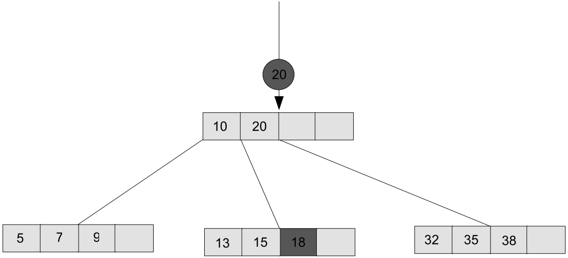


Caso2 A/B:



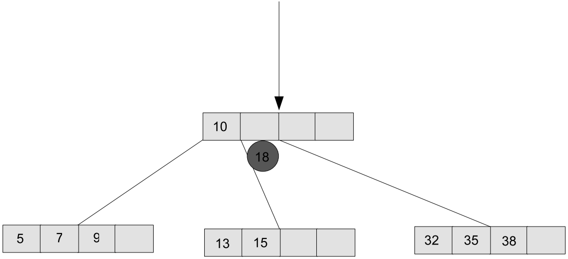
Remoção de Chaves

Caso2 A/B:



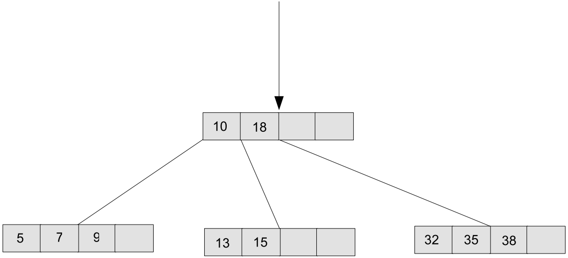
Remoção de Chaves

Caso2 A/B:



Remoção de Chaves

Caso2 A/B:

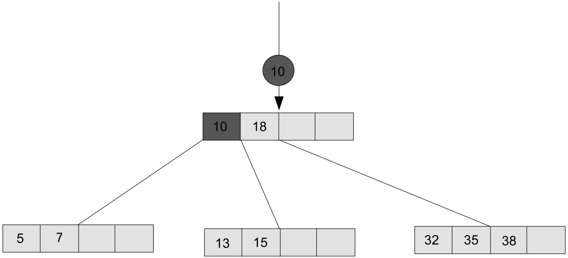


Caso 2(cont.) Se a chave k está num nó interno x o seguinte deve ser feito:

Caso 2(cont.) Se a chave k está num nó interno x o seguinte deve ser feito:

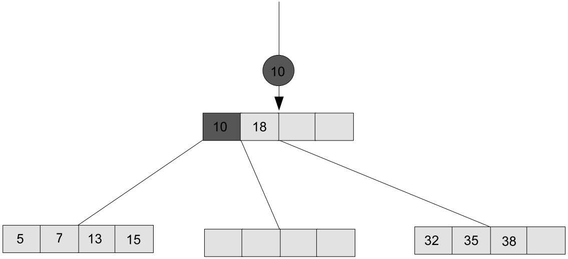
- c. Caso ambos y e z possuem somente $t-1$ chaves, copie todos os elementos de z em y , libere a memória ocupada por z e remova o apontador em x e remova k de x.

Caso2 C:



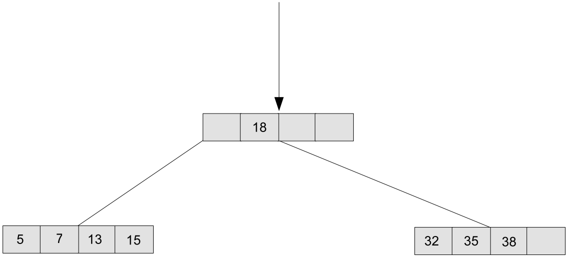
Remoção de Chaves

Caso2 C:



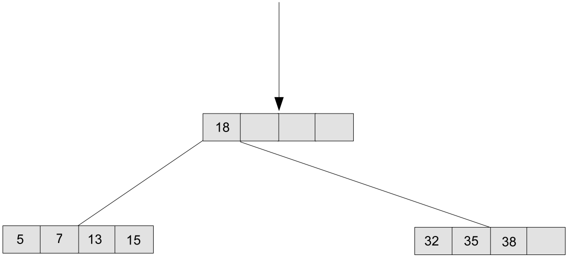
Remoção de Chaves

Caso2 C:



Remoção de Chaves

Caso2 C:



Remoção de Chaves

Caso 3 Se a chave k não está presente no nó interno x, determine a sub-árvore $c_i[x]$ apropriada que deve conter k. Caso $c_i[x]$ possuir somente $t - 1$ chaves, proceder da seguinte maneira:

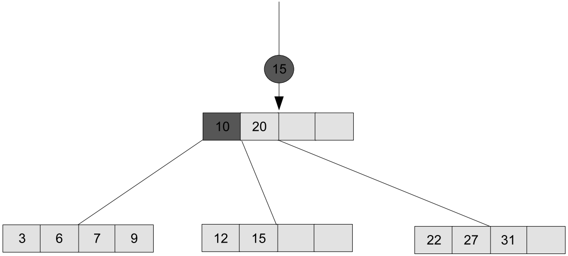
Remoção de Chaves

Caso 3 Se a chave k não está presente no nó interno x, determine a sub-árvore $c_i[x]$ apropriada que deve conter k. Caso $c_i[x]$ possuir somente $t - 1$ chaves, proceder da seguinte maneira:

- a. Se $c_i[x]$ possui pelo menos $t - 1$ chaves mas possui um irmão adjascente com pelo menos t chaves copie para $c_i[x]$ uma chave extra, movendo uma chave de x para $c_i[x]$ em seguida movendo uma chave de um dos irmãos adjascentes a $c_i[x]$ de volta para x e ajustando o apontador para o nó correspondente

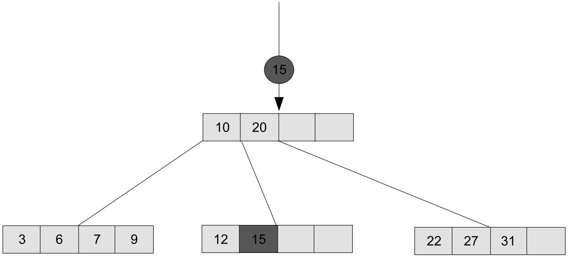
Remoção de Chaves

Caso3 A:



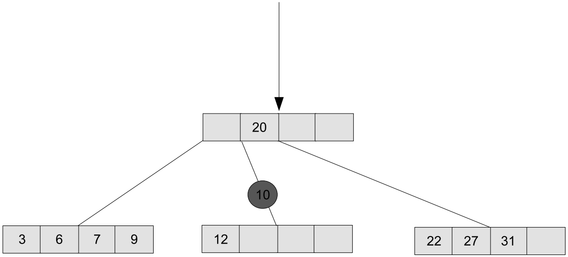
Remoção de Chaves

Caso3 A:



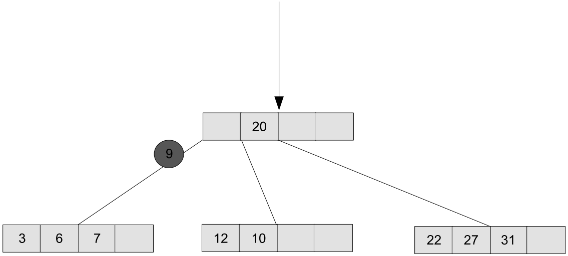
Remoção de Chaves

Caso3 A:

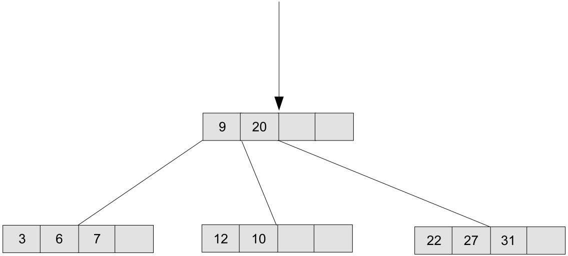


Remoção de Chaves

Caso3 A:



Caso3 A:



Remoção de Chaves

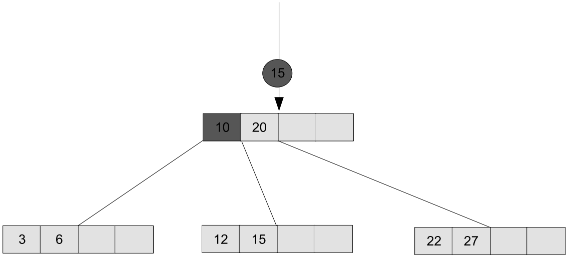
Caso 3(cont.) Se a chave k não está presente no nó interno x, determine a sub-árvore $c_i[x]$ apropriada que deve conter k. Caso $c_i[x]$ possuir somente $t - 1$ chaves, proceder da seguinte maneira:

Remoção de Chaves

Caso 3(cont.) Se a chave k não está presente no nó interno x, determine a sub-árvore $c_i[x]$ apropriada que deve conter k. Caso $c_i[x]$ possuir somente $t-1$ chaves, proceder da seguinte maneira:

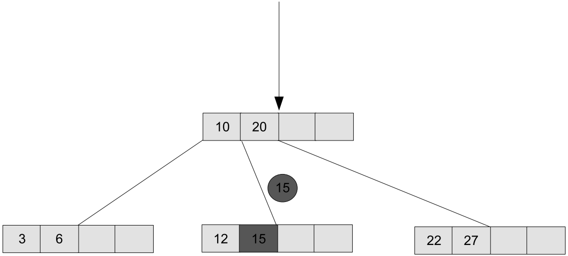
- b. Se $c_i[x]$ e ambos os seus irmãos esquerdo e direito possuem $t-1$ chaves, una $c_i[x]$ com um dos irmãos o que envolve mover uma chave de x para o novo nó que acabou de ser formado, sendo que x é o elemento mediano daquele nó

Caso3 B:

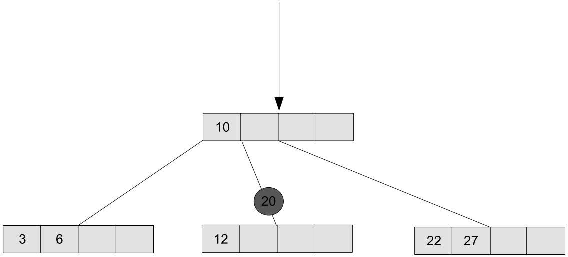


Remoção de Chaves

Caso3 B:

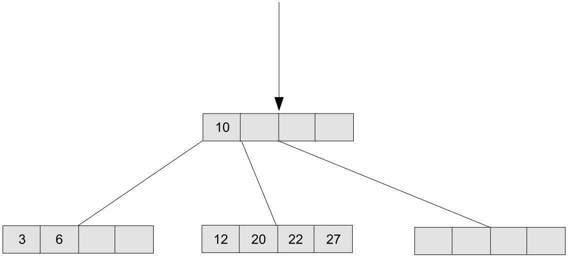


Caso3 B:

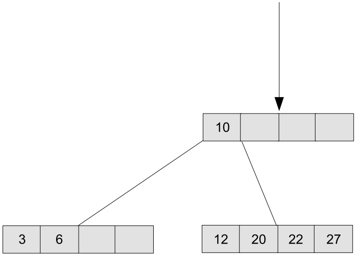


Remoção de Chaves

Caso3 B:



Caso3 B:



Complexidade da Remoção

Complexidade da Remoção

- Sabemos que antes da remoção, é feita uma busca por elemento na árvore, o que gasta $O(t * log_t(n))$, onde t é o tamanho da página da árvore e n é o número total de elementos

Complexidade da Remoção

- Sabemos que antes da remoção, é feita uma busca por elemento na árvore, o que gasta $O(t * log_t(n))$, onde t é o tamanho da página da árvore e n é o número total de elementos
- No pior caso, teremos todas as páginas da árvore com t-1 elementos pois esse é o limite inferior para cada página. Daí teremos o seguinte:

Complexidade da Remoção

Somente os casos 2c e 3b poderão ocorrer e para qualquer um deles teremos o seguinte cenário:

Complexidade da Remoção

Somente os casos 2c e 3b poderão ocorrer e para qualquer um deles teremos o seguinte cenário:

- Se o caso 2c ocorrer, teremos o nó que perdeu a chave com $t-2$ chaves e $t-1$ filhos pois já ocorreu um merge. Se for o caso 3c o pai terá $t-2$ chaves e o elemento que perdeu o nó terá t chaves por causa do merge

Complexidade da Remoção

Somente os casos 2c e 3b poderão ocorrer e para qualquer um deles teremos o seguinte cenário:

- Se o caso 2c ocorrer, teremos o nó que perdeu a chave com $t - 2$ chaves e $t - 1$ filhos pois já ocorreu um merge. Se for o caso 3c o pai terá $t - 2$ chaves e o elemento que perdeu o nó terá t chaves por causa do merge
- Em qualquer um dos dois a reação disparada para corrigir a árvore será a mesma pois isso encaixa o nó com $t - 2$ chaves na situação do caso 3b. Ou seja, uma chave será rebaixada da página pai para ele, e um merge dele com um dos irmãos será feito

Complexidade da Remoção

Complexidade da Remoção

- Agora, o nó pai é que possui $t-2$ chaves repetindo a situação anterior. Ou seja a operação propaga-se num determinado subconjunto de páginas até chagar à raiz

Complexidade da Remoção

- Agora, o nó pai é que possui $t-2$ chaves repetindo a situação anterior. Ou seja a operação propaga-se num determinado subconjunto de páginas até chagar à raiz
- Como os merge copiam $t-1$ elementos a cada nível da árvore teremos uma complexidade de $O((t-1) * log_t(n))$ para os merge onde $log_t(n)$ é a altura da árvore

Complexidade da Remoção

- Agora, o nó pai é que possui $t - 2$ chaves repetindo a situação anterior. Ou seja a operação propaga-se num determinado subconjunto de páginas até chagar à raiz
- Como os merge copiam $t - 1$ elementos a cada nível da árvore teremos uma complexidade de $O((t-1) * log_t(n))$ para os merge onde $log_t(n)$ é a altura da árvore
- Dessa maneira a complexidade da remoção é dada por: $O(t * log_t(n)) + O((t-1) * log_t(n)) = O(t * log_t(n))$

Endereço das Animações

Endereço das Animações

- Vivio B-Tree (necessita do plug-in vivio):
  https://www.cs.tcd.ie/Jeremy.Jones/vivio/trees/B-tree.htm
  Busca no Google: vivio b tree animation

Endereço das Animações

- Vivio B-Tree (necessita do plug-in vivio):
  https://www.cs.tcd.ie/Jeremy.Jones/vivio/trees/B-tree.htm
  Busca no Google: vivio b tree animation

- slady: Java B-Tree applet animation
  http://slady.net/java/bt/view.php
  Busca no Google: b tree animation java