Algoritmos e Estrutura de Dados

Árvores B

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso
Faculdade de Engenharia

Roteiro

1. Objetivos
2. Introdução
3. Motivação
4. Definição
   - Ordem
   - Estrutura Nodo
   - Altura
5. Operações Básicas
   - Criar
   - Busca
   - Inserção
   - Exemplo
   - Remoção
6. Tipos de Árvores B
   - Árvore-2-3
   - Árvore-2-3-4
7. Referências bibliográficas

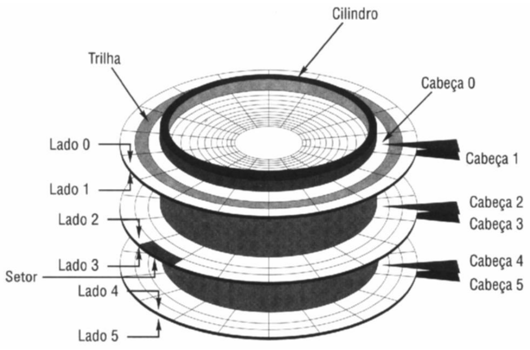
Objetivos

Esta aula tem como objetivos:

1. Apresentar os conceitos básicos sobre árvores B;
2. Exemplificar os algoritmos de manipulação de árvores B por meio de pseudo-códigos.

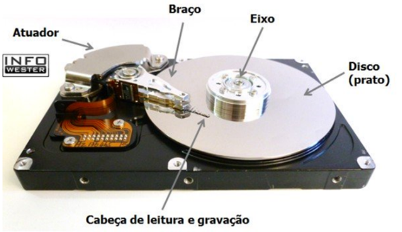
Introdução

- O acesso a disco envolve um posicionamento da cabeça do disco, além da transferência de dados propriamente ditos.
- O posicionamento depende do tempo de rotação do disco que é da ordem de 8 mili-segundos.



Introdução

- Um acesso a disco leva tipicamente 10 a 15 milisegundos, o que é considerável em comparação com o tempo de acesso à memória primária (RAM), da ordem de 100 nano-segundos.
- No tempo para acessar uma vez o disco, podemos fazer cerca de 100.000 acessos à memória.



Introdução

Motivação

- Árvores binárias de Busca (balanceadas ou não) não são adequadas para buscar dados na memória secundária.
  - Para acessar cada nodo da árvore é necessário fazer uma consulta ao disco rígido.
  - Portanto, ao realizar uma consulta, seria necessário aguardar alguns milisegundos.
- Mesmo em árvores AVL, uma grande quantidade de chaves pode requerer um número excessivo de acessos a disco rígido.
  - Em uma árvore AVL, com $n = 10^6$, o valor de $\log_2 n \approx 10$, ou seja, seriam necessários aproximadamente 20 acessos a disco rígido, o que é altamente custoso.

Introdução

Motivação

- Uma operação de acesso ao disco é cara, portanto é necessário consultar uma quantidade maior de dados.
  - Vetores são armazenados em posições contíguas no disco rígido, portanto em um único acesso todo o vetor é carregado para a memória.
- Diante disso, (BAYER; MCCREIGHT, 1970) criaram as Árvores B, em que cada nodo armazena um vetor de elementos.
  - O nome "Árvore B"é um mistério.
  - Conjectura-se que seja "B" de "B"ayer, ou de "B"alanceada ou ainda de "B"oeing, a companhia onde trabalhavam os dois autores.
- Nas árvores B, um nodo pode conter centenas de chaves, e é chamado de **página**.

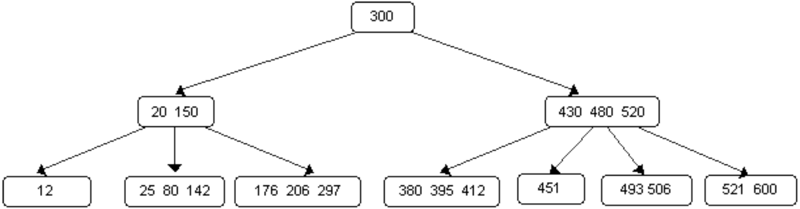
Árvores B
Definição

- Nas Árvores B existe uma quantidade máxima e mínima de chaves que podem ser armazenadas em um nodo.
    - Esses limites são definidos de acordo com uma **ordem**, explicada a seguir.
- Devido à sua estrutura, todos os nodos folhas permanecem no mesmo nível, que é a altura h da árvore.
    - Diferentemente das outras árvores estudadas, as Árvores B crescem para cima.
- Além disso, é garantido que qualquer nodo em uma Árvore B possui pelo menos 50% de quantidade máxima de chaves.

- Considere **ordem** um inteiro $t \geq 2$, de forma que representa o grau (quantidade de filhos) mínimo da árvore.
- A ordem de uma Árvore B define a quantidade mínima e máxima de chaves em uma página.
  - Qualquer página (exceto a raiz) deve ter no mínimo $t - 1$ chaves.
    - Ou seja, deve possuir no mínimo t filhos.
  - Toda página deve conter no máximo $2t - 1$ chaves.
    - Portanto, toda página interna tem no máximo $2t$ filhos.
- Quantidade n de chaves: $t - 1 \leq n \leq 2t - 1$.
- Quantidade m de filhos: $t \leq m \leq 2t$

Árvores B
Exemplo

Exemplo de uma Árvore B, de ordem $t = 2$, denominada Árvore 2-3-4:



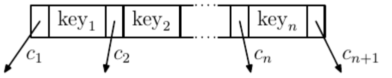
- Toda página interna possui $1 \leq n \leq 3$ chaves.
- Toda página interna possui $2 \leq m \leq 4$ filhos, daí o nome Árvore 2-3-4.
- Todos as páginas folhas estão na mesma altura na árvore.

Introdução

Estrutura Nodo

A estrutura de dados **nodo** de uma Árvore B possui quatro campos:

1. O total de chaves n atualmente armazenadas no nodo.
2. Um vetor, denominado *key*, de tamanho n, em que as chaves são armazenadas em ordem não-decrescente:
$$key_1 \leq key_2 \leq \ldots \leq key_n.$$
3. Um valor booleano, denominado *folha*, que indica se é folha ou um nodo interno.
4. Um vetor contendo $n+1$ ponteiros, $c_1, c_2, \ldots, c_{n+1}$, para os filhos.
    - Caso o nodo seja folha, seus ponteiros apontam para NULL.

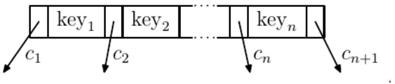


Algoritmo 1: Nodo

```
início
    registro {
        Vetor Inteiro: key[1..2t-1]; // Vetor de chaves.
        Vetor Ponteiro Nodo: c[1..2t]; // Vetor de ponteiros.
        Inteiro: n; // Quantidade de chaves armazenadas.
        Booleano: folha; // Indica se o nodo é folha.
    } Nodo;
```

Árvores B
Ordenação

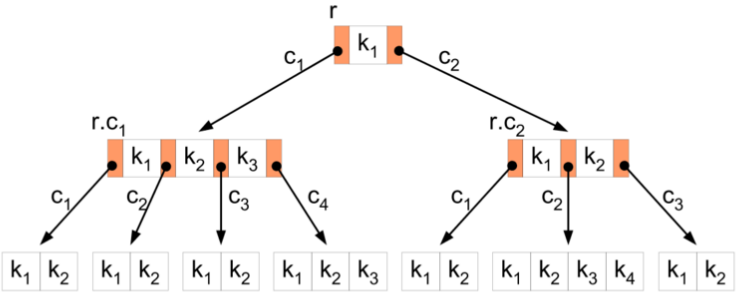
- Assim como nas árvores binárias, nas Árvores B cada chave key_i possui filhos à esquerda (localizados em c_i) e filhos à direita (localizados em c_{i+1}).



- Considere $c_i.k$, qualquer chave pertencente à c_i.
  - Toda chave $c_i.k$ armazenada no filho da esquerda c_i satisfaz $c_i.k \leq key_i$.
  - De forma análoga, toda chave $c_{i+1}.k$ armazenada no filho da direita c_{i+1} satisfaz $key_i \leq c_{i+1}.k \leq key_{i+1}$.
- Dessa forma, temos

$$c_1.k \leq key_1 \leq c_2.k \leq key_2 \leq c_3.k \leq \ldots \leq key_n \leq c_{n+1}.k.$$

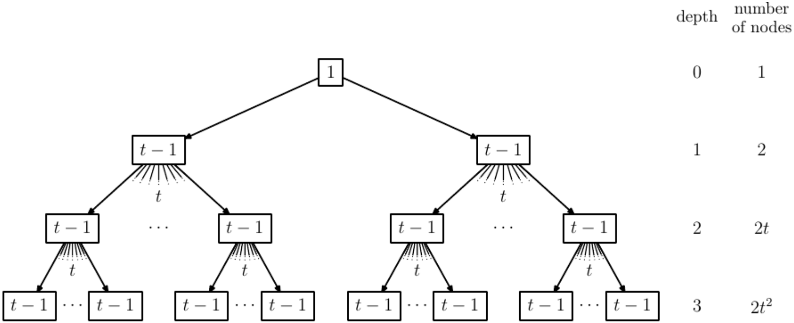
Árvores B
Ordenação



Introdução

Altura

Uma Árvore B de altura 3 contém um número **mínimo** possível de chaves.



Conforme mostrado na figura, cada nodo possui no mínimo $t-1$ chaves, exceto a raiz.

Introdução

Altura

- Analisando a quantidade de nodos:
  - No nível 1, tem-se $2t^0$ nodos.
  - No nível 2, tem-se $2t^1$ nodos.
  - No nível 3, tem-se $2t^2$ nodos.
  - De forma análoga, no nível i, tem-se $2t^{i-1}$ nodos.
- Portanto, a árvore possui um total de $1 + \sum_{i=1}^{h} 2t^{i-1}$ nodos.
  - O valor "+1" refere-se a raiz.

Introdução

Altura

- Verificamos que trata-se de uma PG (Progressão Geométrica).
  - $2t^0 + 2t^1 + 2t^2 + ... + 2t^{h-1}$.
- Essa PG possui $a_1 = 2$, a razão $q = t$, e a quantidade de elementos $n = h$, em que h é a altura da árvore.
- Com isso, podemos calcular o **total de nodos** S_n da seguinte forma:

$$
\begin{aligned}
S_n &= \frac{a_1(q^n - 1)}{q - 1} \\
&= \frac{2(t^h - 1)}{t - 1}.
\end{aligned}
$$

Introdução

Altura

Exceto a raiz, cada nodo armazena no mínimo $t-1$ chaves. Portanto, o **total de chaves** n da árvore é igual a $(t-1)$ vezes o total de nodos mais a chave na raiz:

$$\begin{aligned} n &\geq 1+(t-1)S_n \\ &\geq 1+(t-1)\frac{2(t^h-1)}{t-1} \geq 2t^h-1 \end{aligned}$$

A altura da árvore em função da quantidade de chaves pode ser obtida da seguinte forma:

$$\begin{aligned} 2t^h-1 &\leq n \\ t^h &\leq \frac{n+1}{2} \\ \log_t t^h &\leq \log_t(\frac{n+1}{2}) \\ h &\leq \log_t(\frac{n+1}{2}) \end{aligned}$$

Introdução

Altura

- O número de acessos ao disco é proporcional à altura da Árvore B.
- No pior caso, referente à altura da Árvore B, é

$$h \leq \log_t \frac{n+1}{2} \approx O(\log_t n).$$

Introdução

Altura

As principais vantagens da Árvore B em relação às outras árvores são:

- A base do logaritmo, t, deve ser um valor grande, o que reduz o custo das operações.
- Árvores B reduzem o número de nodos examinados nas operações (busca, inserção e remoção) em um fator $\log t$.
- Dessa forma, o número de acessos ao disco é reduzido substancialmente.

Operações Básicas

A seguir, as operações básicas a serem realizadas sobre Árvores B:

- ArvoreB-Criar
- ArvoreB-Busca
- ArvoreB-Inserir
- ArvoreB-Remover

Operações Básicas

Conveções:

- A raiz de uma Árvore B sempre está na memória principal (DISK-READ na raiz nunca será necessário).
- Qualquer nodo passado como parâmetro deve ter uma operação DISK-READ realizada sobre ele, a fim de ler os dados do disco.
- Sempre que um nodo for alterado, deve ter uma operação DISK-WRITE realizada sobre ele, a fim de salvar os dados no disco.
- Todos os procedimentos apresentados são algoritmos **top-down**, iniciando na raiz da árvore.

CriaÁrvore

Pseudocódigo

Algoritmo 2: ArvoreB-Criar

```
Entrada: Ponteiro T para a árvore.
início
    novo ← ALOCA_NODO()
    novo.folha ← TRUE
    novo.n ← 0
    DISK_WRITE(novo)
    T.raiz ← novo
```

Adaptado de (CORMEN et al., 2012).

- ALOCA_NODO() aloca uma página no disco de modo a ser usada como um novo nodo.
- Requer $O(1)$ operações no disco e tempo de processador $O(1)$.

Busca

- A busca de uma dada chave k numa árvore B é análoga à busca na árvore binária de busca.
- A busca começa pela página raiz r.
- É usual manter a raiz sempre na memória, evitando um acesso ao disco.

Busca

- Estando em uma página da Árvore B, procedemos assim:
  1. Busca-se k em r, usando um método de busca sequencial ou busca binária, dependendo do valor de t. Para pequenos valores de t, busca sequencial já basta.
  2. Se k estiver em r, então retorna o nodo r e a posição de k na página.
  3. Se $k < key[1]$, então realiza a busca recursivamente na página apontada por $c[1]$.
  4. Se $key[i] < k < key[i+1]$, então continua a busca na página apontada por $c[i+1]$.
  5. Se $k > key[n]$, então continua a busca na página apontada por $c[n+1]$.
- Pode-se ver que a busca leva tempo $O(\log_t n)$, onde t é a ordem da árvore B e n é o número total de chaves.

Busca

Pseudocódigo

Algoritmo 3: ArvoreB-Busca

Entrada: Ponteiro para a raiz r, chave k a ser buscada na árvore.
Saída: O nodo que contém k ou NULL caso não encontre.

início
    i$\leftarrow$ 1
    enquanto $(i \leq r.n)$ AND $(k > r.key[i])$ **faça**
        $i \leftarrow i + 1$
    se $(i \leq r.n)$ AND $(k = r.key[i])$ **então**
        retorna (r, i)
    se $(r.folha)$ **então**
        retorna NULL
    senão
        DISK_READ($r.c_i$)
        retorna ArvoreB-Busca($r.c[i]$, k)

Busca

- Número de páginas do disco acessadas por ArvoreB-Busca

$$\Theta(h) = \Theta(\log_t n)$$

- Tempo de execução do laço **enquanto** (linhas 3-4) dentro de cada nodo é $O(t)$.
- Portanto o tempo total de execução é:

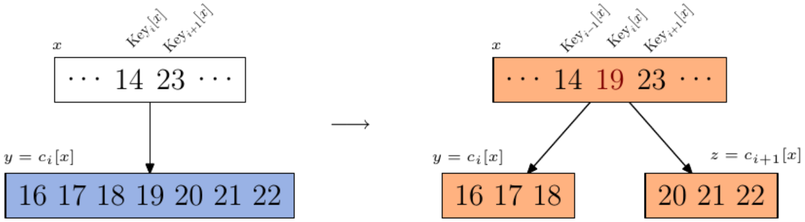
$$O(t \times h) = O(t \log_t n)$$

Inserção

- Para inserir uma nova chave x numa árvore B de ordem t, existem dois casos a serem considerados.
- A seguir os passos necessários:
  - Primeiro localizamos a página folha onde será feita a inserção.
  - Verificamos quantas chaves já estão na página antes de adicionar a chave k na mesma.
  - **Caso 1**: Se a página contém menos que $2t - 1$ chaves, então basta inserir a nova chave k na página.

Inserção

- **Caso 2**: Se a página contém exatamente $2t - 1$ chaves, então é necessário realizar a sua divisão:
    - Inclua a nova chave k, em ordem não decrescente, na página em questão. Dessa forma, a página terá $2t$ chaves
    - Em seguida, realiza-se a divisão (ou cisão) da página em duas.
    - O elemento do meio (posição t) é inserido, recursivamente, na página pai.
    - Alocamos as primeiras $t - 1$ chaves numa página e as últimas t chaves noutra página.

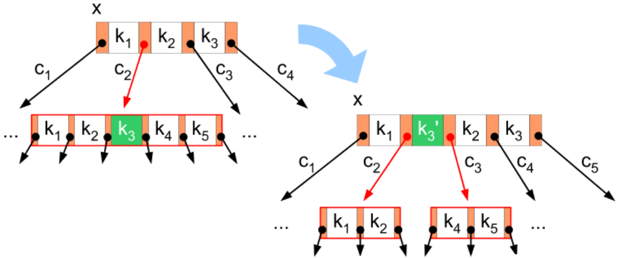


Inserção

Divisão

A seguir, os passos necessários para realizar a divisão de um nodo:

- Considere dividir o nodo y filho de x, localizado na posição i.
  - Ou seja, $y = x.c[i]$.
- Para isso, deve-se:
  - Alocar um novo nodo z.
  - Copiar as chaves de y para z.
  - Copiar os ponteiros de y para z.
  - Mover o elemento no meio de y para x.



Divisão

Pseudocódigo

Algoritmo 4: ArvoreB-Divide-Filho

Entrada: Um nodo interno não-cheio x, um índice i, um nodo cheio y filho de x, tal que $y = x.c_i$.

início
  z ← ALOCA_NODO()
  z.folha ← y.folha
  z.n ← t-1
  para *(j ← 1 até t-1)* **faça**
    $z.key[j] \leftarrow y.key[j+t]$
  se *(NOT(y.folha))* **então**
    para *(j ← 1 até t)* **faça**
      $z.c[j] \leftarrow y.c[j+t]$
  y.n ← t-1

Divisão

Pseudocódigo

Vamos analisar o algoritmo ArvoreB-Divide-Filho.

- Deseja-se dividir o nodo y, cujo pai é x.
  - O nodo y está cheio e possui $2t - 1$ chaves.
- Primeiramente cria-se um novo nodo, denominado z, linhas 2 a 4.
  - z será folha se y for folha.
  - z terá a quantidade mínima de chaves, que é $t - 1$.
- Em seguida, copias as chaves de y para z, linhas 5 a 6.
  - As primeiras $t - 1$ chaves permanecerão em y.
  - O elemento na posição t de y será movido para o pai x.
  - As $t - 1$ chaves restantes serão copiadas para z.
- Copia os ponteiros de y para z, linhas 7 a 9.
  - Copia t ponteiros de y para z a partir da posição t.
- Dessa forma, deve-se atualizar a quantidade de chaves de y (linha 10).

Divisão

Pseudocódigo

Algoritmo 5: ArvoreB-Divide-Filho (continuação)

```
início
    para (j ← x.n + 1  descendo até  i + 1) faça
        x.c[j + 1] ← x.c[j]
    x.c[i + 1] ← z
    para (j ← x.n  descendo até  i) faça
        x.key[j + 1] ← x.key[j]
    x.key[i] ← y.key[t]
    x.n ← x.n + 1
    DISK_WRITE(y)
    DISK_WRITE(z)
    DISK_WRITE(x)
```

Adaptado de (CORMEN et al., 2012).

Divisão

Pseudocódigo

Vamos analisar a continuação do algoritmo ArvoreB-Divide-Filho.

- Move os ponteiros de x para a direita, linhas 12 a 13.
- Adiciona z como filho de x na posição $i + 1$, linha 14.
- Move as chaves de x para a direita, linhas 15 a 16.
- Insere o elemento que encontra-se no meio de y em z, linha 17.
- Atualiza o total de nodos armazenados em x, linha 18.
- O tempo de execução usado por ArvoreB-Divide-Filho é $\Theta(t)$, devido aos loops **para**.

Inserção

A seguir, algums considerações sobre a inserção.

- A chave sempre é inserida em um nodo folha.
- A inserção é feita em um único passo descendo a partir da raiz.
- Requer $O(h) = O(\log_t n)$ acessos ao disco.
- Requer tempo de processamento $O(t \times h) = O(t \log_t n)$.
- Ao descer na árvore (da raiz em direção à folha), divide antecipadamente nodos cheios para garantir que a recursão não desce até um nodo cheio.

Inserção

Pseudocódigo

Algoritmo 6: ArvoreB-Inserir

Entrada: Ponteiro T para a árvore, chave k a ser inserida.

```
início
    r ← T.raiz
    se (r.n = 2t − 1) então
        s ← ALOCA_NODO()
        T.raiz ← s
        s.folha ← FALSE
        s.n ← 0
        s.c[i] ← r
        ArvoreB-Divide-Filho(s,1,r)
        ArvoreB-Inserir-NaoCheio(s,k)
    senão
        ArvoreB-Inserir-NaoCheio(r,k)
```

Adaptado de (CORMEN et al., 2012)

Inserção

Pseudocódigo

No algoritmo ArvoreB-Inserir, tem-se dois casos:

1. Caso seja encontrado um nodo cheio.
   - Mesmo que a chave não seja inserida nesse nodo, divide-o previamente.
   - Em seguida, chama a função ArvoreB-Inserir-NaoCheio, que recursivamente insere a chave na posição correta.
2. Caso seja encontrado um nodo não-cheio.
   - Insere a chave no nodo chamando a função ArvoreB-Inserir-NaoCheio.

Inserção

Pseudocódigo

Algoritmo 7: ArvoreB-Inserir-NaoCheio

Entrada: Nodo raiz x, chave k a ser inserida.

```
início
    i ← x.n
    se (x.folha) então
        enquanto (i ≥ 1) AND (k < x.key[i]) faça
            x.key[i + 1] ← x.key[i]
            i ← i - 1
        x.key[i + 1] ← k
        x.n ← x.n + 1
        DISK_WRITE(x)
    senão
        enquanto (i ≥ 1) AND (k < x.key[i]) faça
            i ← i - 1
        i ← i + 1
        DISK_READ(x.c[i])
        se (x.c[i].n = 2t − 1) então
            B-Tree-Split-Child(x,i,x.c[i])
            se (k > x.key[i]) então
                i ← i + 1
        ArvoreB-Inserir-NaoCheio(x.c[i], k)
```

Adaptado de (CORMEN et al., 2012).

Inserção

Pseudocódigo

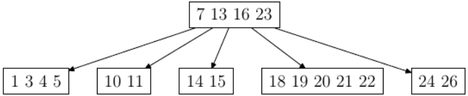
- A função auxiliar ArvoreB-Inserir-NaoCheio insere a chave k no nodo x, que se presume ser não-cheio.
- Essa função possui basicamente dois casos:
  1. Se x é um nodo folha,
     - As linhas 4 a 9 tratam esse caso, inserindo a chave k em x.
  2. Caso contrário, deve inserir k no nodo folha apropriado na subárvore com raiz em x.
     - Nesse caso, as linhas 10 a 13, determinam o filho de x para o qual deve-se inserir recursivamente.
     - A linha 15 verifica se a função será chamada recursivamente em um nodo cheio.
     - Se o nodo estiver cheio, divide-o em dois, linha 16, e as linhas 17 a 18 determinam qual dos dois filhos é o filho correto para inserir recursivamente.

Inserção

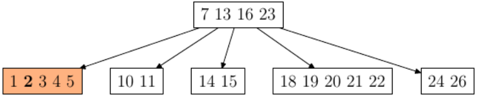
Exemplo

Initial tree:

$t = 3$

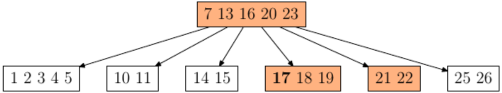


2 inserted:



17 inserted:

(to the previous one)

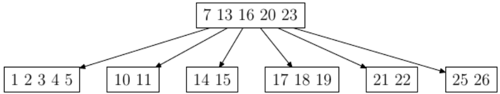


Inserção

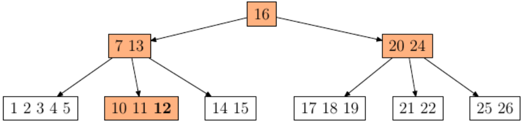
Exemplo

Initial tree:

$t = 3$

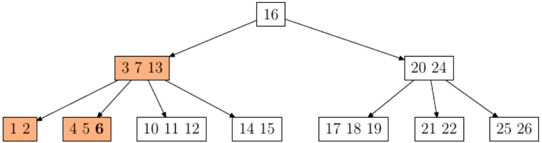


12 inserted:



6 inserted:

(to the previous one)



Remoção

- A remoção é similar à inserção, com adição de alguns casos especiais.
- Diferentemente da inserção, na remoção uma chave pode ser removida de qualquer nodo.
- É um procedimento mais complicado, mas com performance similar:
  - Acessos ao disco: $O(h)$.
  - Tempo de processamento: $O(t \times h) = O(t \log_t n)$.

Remoção

- Remover um nodo é feito em um único passo, partindo da raiz.
- Ao remover de um nodo folha:
  - Caso tenha pelo menos t chaves, pode-se simplesmente remover.
  - Caso contrário, pode-se tentar movimentar chaves de um dos irmãos.
- Ao remover uma chave de um nodo interno:
  - Primeiramente deve-se tentar movimentar chaves de um dos filhos ou dos irmãos.
- Em último caso, realiza-se a operação de **união** de dois nodos.

Remoção

Considere k a chave a ser removida e x o nodo que contém k. Existem três casos:

1. Se a chave k está em x, um nodo folha, e x contém pelo menos t chaves.
   - A remoção é trivial.
2. Se a chave k está em x, um nodo interno. Existem três casos a considerar:
   a. Verifique se o filho à esquerda y possui pelo menos t chaves. Nesse caso, a chave predecessora k' de k é movida para x, e em seguida remove-se k.
   b. Simetricamente, verifique se o filho à direita z possui pelo menos t chaves. Nesse caso, a chave sucessora k' de k é movida para x, e em seguida remove-se k.
   c. Caso contrário, se ambos y e z tem apenas $t - 1$ chaves, realize a união de k.
      - Todos os elementos em z e em y passam a pertencer a um único nodo, em conjunto com a chave k.
      - Com isso, k e o ponteiro para z são removidos de x. y agora contém $2t - 1$ chaves, e subsequentemente k é removida.

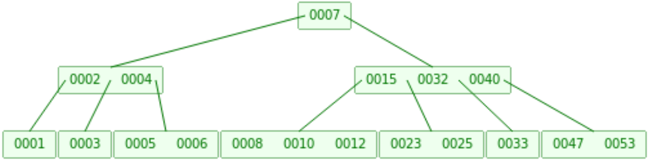
Remoção

3. Se a chave k não foi encontrada no nodo x.
   - É necessário determinar a subárvore $x.c[i]$ apropriada que deve conter k.

   a. Ao determinar o filho que contém k, verifica-se se este possui apenas $t-1$ chaves.
      - Verifique se um dos irmãos de $x.c[i]$ possui pelo menos t chaves. Se possível, movimente uma chave do irmão à esquerda ou à direita para x.
      - Se $x.c[i]$ tiver filhos, verifique também se um deles possui pelo menos t chaves. Nesse caso, movimente uma chave de um filho.
   b. Caso contrário, $x.c[i]$, todos os seus filhos e irmãos contém apenas $t-1$ chaves. Dessa forma, resta apenas fazer a intercalação de $x.c[i]$ com um de seus irmãos, o que envolve mover uma chave de x para baixo.

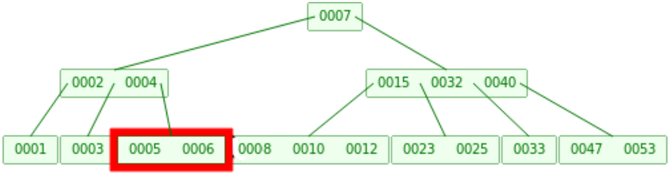
Exemplo

Árvore inicial (t=2):



Exemplo

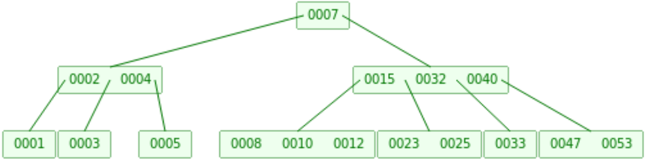
Caso 1: remover 6



Trivial, pois o nodo, que é folha, contém mais que $t - 1$ chaves.

Exemplo

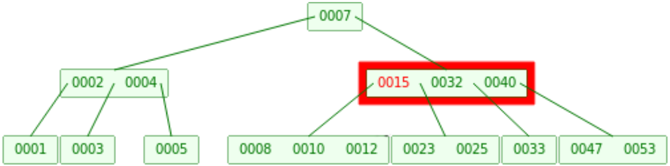
Caso 1: remover 6



Basta remover a chave.

Exemplo

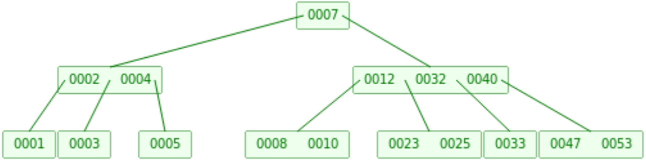
Caso 2 (a): remover 15



Verifique se o filho à esquerda possui pelo menos t chaves.

Exemplo

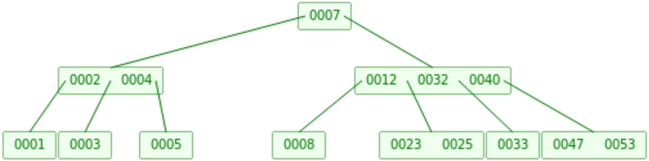
Caso 2 (a): remover 15



Remove o 15 e o substitui pelo seu antecessor, a chave 12.

Exemplo

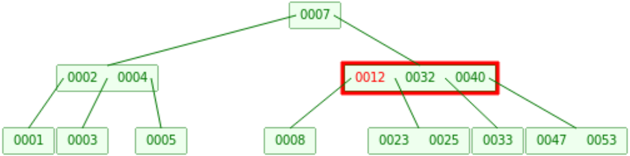
Caso 2 (b): remover 12



Filho à esquerda possui apenas $t - 1$ chaves.

Exemplo

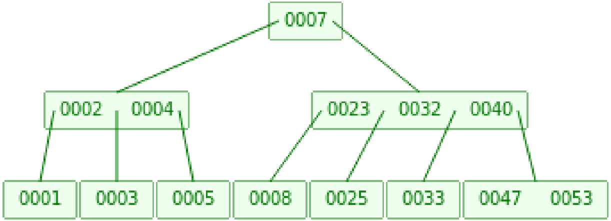
Caso 2 (b): remover 12



Verifique se o filho à direita possui pelo menos t chaves.

Exemplo

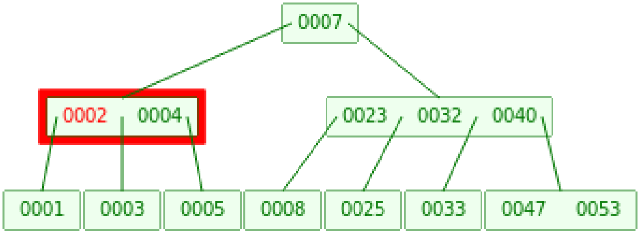
Caso 2 (b): remover 12



Remove o 12 e o substitui pelo seu sucessor, a chave 23.

Exemplo

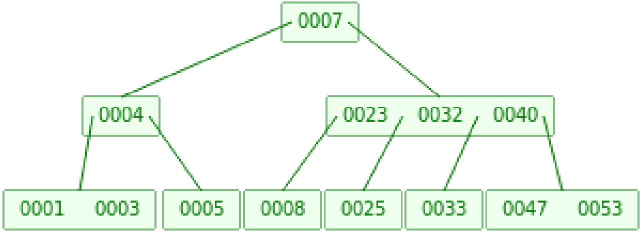
Caso 2 (c): remover 2



Nenhum dos filhos do nodo interno com chave 2 possui mais que $t - 1$ chaves

Exemplo

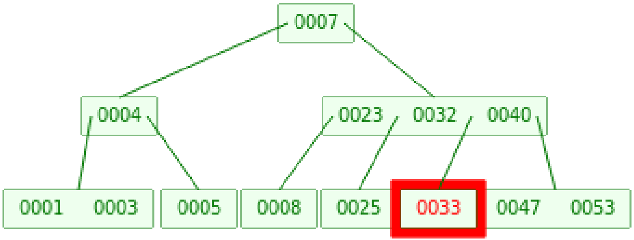
Caso 2 (c): remover 2



A chave 2 desce e realiza-se a unidão dos filhos. Em seguida remove a chave 2.

Exemplo

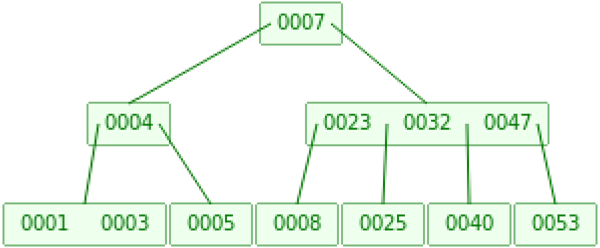
Caso 3 (a): remover 33



O nodo que contém a chave 33 possui apenas $t - 1$ chaves. O nodo não possui filhos. Verifique se um dos irmãos possui pelo menos t chaves.

Exemplo

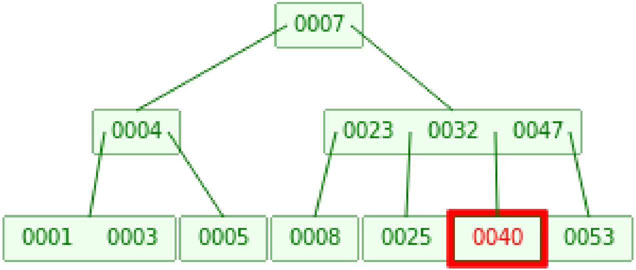
Caso 3 (a): remover 33



Uma chave do irmão sobre e uma chave do pai desce. Em seguida, remove a chave 33.

Exemplo

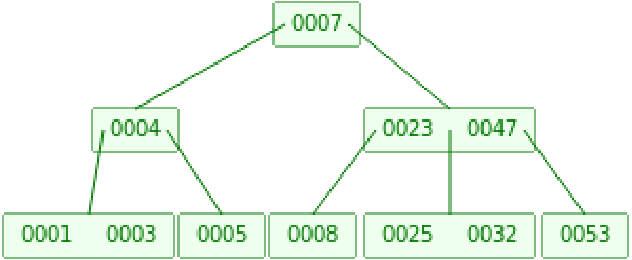
Caso 3 (b): remover 40



Nenhum dos filhos ou irmãos possui mais que $t - 1$ chaves.

Exemplo

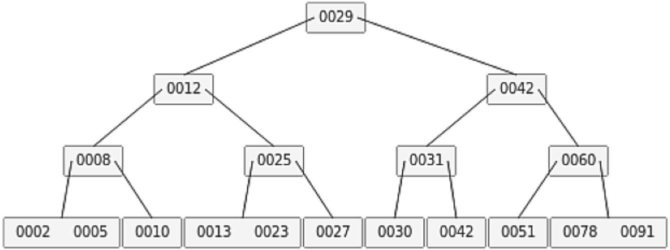
Caso 3 (b): remover 40



Uma chave do pai desce e realiza-se a união com um dos irmãos. Em seguida, remove a chave 40.

- Um caso particular de Árvore B é a chamada Árvore-2-3, que **não segue as regras** apresentadas em (CORMEN et al., 2012).
- Cada nó da árvore-2-3 tem 1 ou 2 chaves.
  - Consequentemente cada nodo possui 2 ou 3 filhos, daí o nome.
- A Árvore-2-3 é uma árvore usada fazer busca de dados armazenados na memória principal.
- Na prática, para armazenamento e busca em disco, uma árvore B usa uma ordem t grande, tipicamente de alguma centenas de chaves.

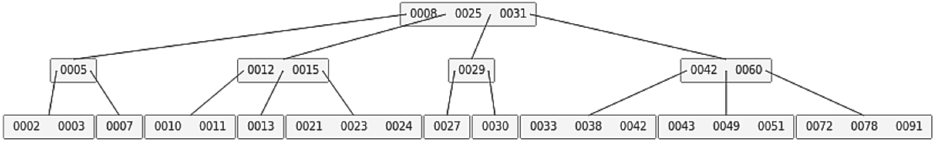
Exemplo



0029
0012
0042
0008
0025
0031
0060
0002 0005
0010
0013 0023
0027
0030
0042
0051
0078 0091

- Um caso particular de Árvore B é a chamada árvore-2-3-4.
- Segundo a definição de (CORMEN et al., 2012), é a menor árvore B possível.
- Uma árvore-2-3-4 é uma Árvore B de ordem $t = 2$.
- Cada nó da árvore-2-3-4 tem 1, 2 ou 3 chaves.
- Cada nó da árvore-2-3-4 tem 2, 3 ou 4 filhos, daí o nome.

Exemplo


0008 0025 0031
0005
0012 0015
0029
0042 0060
0002 0003
0007
0010 0011
0013
0021 0023 0024
0027
0030
0033 0038 0042
0043 0049 0051
0072 0078 0091

Conclusão


VIDADEPROGRAMADOR
.COM.BR
#273
CARA, VOCÊ VIU QUE HOJE É DIA DA ÁRVORE?
AHAN... ATÉ FIZ MINHA HOMENAGEM HOJE CEDO...
PLANTOU UMA ÁRVORE?
NÃO EXATAMENTE...
IMPLEMENTEI UMA ÁRVORE B!

Referências Bibliográficas

BAYER, R.; MCCREIGHT, E. Organization and maintenance of large ordered indices. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA: ACM, 1970. (SIGFIDET '70), p. 107–141. Disponível em: ⟨http://doi.acm.org/10.1145/1734663.1734671⟩.

CORMEN, T. H. et al. *Algoritmos: Teoria e Prática*. 3. ed. São Paulo: Campus, 2012. ISBN 978-0-262-03384-8.

Material Complementar

Animações

- https://www.cs.usfca.edu/ galles/visualization/BTree.html
  - Obs.: Escolha "Max Degree $= 4$" para árvore de ordem $t = 2$.
- http://cs.armstrong.edu/liang/animation/web/24Tree.html
  - Árvore 2-3-4