# Árvores B

- Nesta aula será apresentado o ADT árvore B que são árvores de m-vias completas
- As árvores B são projetadas para funcionar bem em dispositivos memória secundária

**Algoritmos e Estruturas de Dados II**

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
augusto@ffclrp.usp.br
http://dfm.ffclrp.usp.br/~augusto

# Introdução

- Na pesquisa binária, as operações de busca em um conjunto com **n** elementos podem ser efetuadas em $O(\log_2 n)$, na realidade, $\lceil \log_2 n \rceil$ mesmo no pior caso
- No caso em que o conjunto de elementos se encontre em um arquivo em disco, sendo que os elementos (registros) estão ordenados por um campo chave então seria possível aplicar uma pesquisa binária para encontrar um elemento chave qualquer
  - Se os registros são recuperados do disco, um de cada vez, então a pesquisa binária exigiria um total de 20 acessos para um arquivo com um milhão ($10^6$) de entradas
- Entretanto, a inserção e remoção de elementos exigiria que o arquivo inteiro fosse reordenado
- Veremos uma técnica no qual inserções e remoções de registros tem apenas efeitos locais, sem exigir a reorganização total do arquivo, conhecida **indexação**



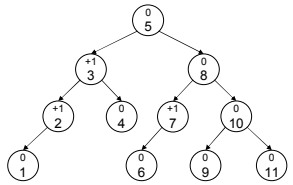
# Introdução

- Um índice é uma estrutura de dados que permite rápido acesso aos dados com base em uma chave
- É importante que um índice, além de agilizar a busca, permita inserções e remoções de dados igualmente rápidas
- Por exemplo, nas árvores AVL as operações de busca, inserção e remoção em um conjunto com **n** elementos podem ser efetuadas em $O(\log_2 n)$, mesmo no pior caso
- Como as mesmas funções devem ser executadas para um índice, seria possível utilizar árvores AVL também para essa aplicação, ou seja, a árvore AVL residiria em disco
- Se os nós são recuperados do disco, um de cada vez, então a pesquisa de um índice com **n** entradas precisaria, no máximo, de $1.4*\log_2 n$ acessos (a profundidade máxima de uma árvore AVL é $1.4*\log_2 n$)



# AVL como Índice em Disco

- No exemplo, se cada nó da AVL reside em disco (um registro representando cada nó), o acesso às chaves 1, 6, 9 e 11 requer a leitura de 4 registros






# Representação

- Assim, um nó AVL em memória principal:

```
struct TreeNode
{ int key;         // chave
  int bal;         // -1, 0, +1
  TreeNode *LeftNode, *RightNode; //subárvores
};
typedef TreeNode *TreePointer;
```

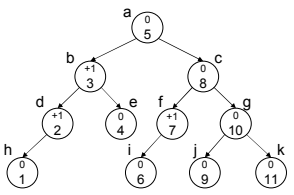
- Poderia ser alterado para ser representado em disco:

```
struct TreeNode
{ int key;         // chave
  int bal;         // -1, 0, +1
  int LeftNode, RightNode; //subárvores
};
```

- Uma vez que os campos **LeftNode** e **RightNode** correspondem, nesta última representação, a número de registros (1, 2, ...)



# AVL como Índice em Disco




| | key | bal | LeftNode | RightNode |
|---|---|---|---|---|
| a | 5 | 0 | b | c |
| b | 3 | +1 | d | e |
| c | 8 | 0 | f | g |
| d | 2 | +1 | h | 0 |
| e | 4 | 0 | 0 | 0 |
| f | 7 | +1 | i | 0 |
| g | 10 | 0 | j | k |
| h | 1 | 0 | 0 | 0 |
| i | 6 | 0 | 0 | 0 |
| j | 9 | 0 | 0 | 0 |
| k | 11 | 0 | 0 | 0 |

# AVL como Índice em Disco

- No pior caso, isso significa cerca 23 acessos para um índice com um milhão ($10^6$) de entradas
- É possível reduzir consideravelmente o número de 23 acessos, usando uma árvore equilibrada, baseada numa árvore de busca em **m**-vias, ao invés daquela baseada em uma árvore de busca binária balanceada



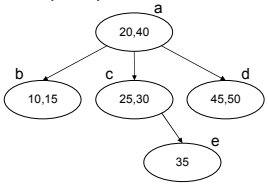
# Histórico

- Em 1972 Bayer & McGreight publicaram o artigo *Organization and Maintenance of Large Ordered Indexes*
- Em 1979, o uso de árvores B era um padrão adotado em sistemas de arquivos de propósito geral para a manutenção de índices para bases de dados



# Árvore de Busca em m-Vias

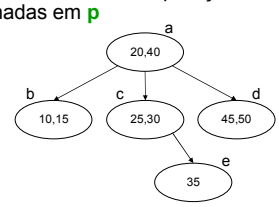
- As árvores de busca em **m**-vias (**m**-árias) generalizam as árvores de busca binária de maneira natural
- Note que se um nó **p** da árvore tem **n** chaves então **p** tem (**n**+1) filhos






# Árvore de Busca em m-Vias

- As chaves no nó **p** são usadas como pontos que separam o intervalo de chaves gerenciadas por **p** em (**n**+1) intervalos, cada um gerenciado por um filho de **p**
- Em uma busca por uma valor, a decisão é feita de (**n**+1) modos, com base em comparações com as **n** chaves armazenadas em **p**



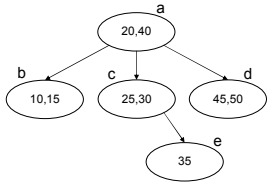



# Árvore de Busca em m-Vias

- Uma árvore de busca em **m**-vias é definida como:
  - Uma árvore vazia é uma árvore de busca em **m**-vias
  - Uma árvore de busca em **m**-vias T é uma árvore na qual todos os nós são de um grau menor ou igual a **m**
- Quando T for não vazia, ela tem as seguintes propriedades:
  - (a) T é um nó do tipo $n,A_0,(K_1,A_1),(K_2,A_2),\ \ldots\ ,(K_n,A_n)$ onde
    $A_i$, $0 \le i \le n$ são ponteiros para as subárvores de T;
    $K_i$, $1 \le i \le n$ são os valores de chave e $1 \le n < m$
  - (b) $K_i < K_{i+1}$, $1 \le i < n$
  - (c) Todos os valores de chave na subárvore $A_i$ são menores do que o valor da chave $K_{i+1}$, $0 \le i < n$
  - (d) Todos os valores de chave na subárvore $A_n$ são maiores do que $K_n$
  - (e) As subárvores $A_i$, $0 \le i \le n$ são também árvores de busca em **m**-vias



# Exemplo AB em 3-vias




$n,A_0,(K_1,A_1),(K_2,A_2),\ \ldots\ ,(K_n,A_n)$

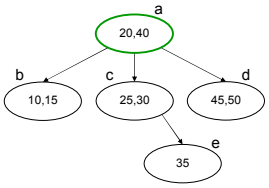
| Nó | Formato esquemático |
|---|---|
| a | 2,b,(20,c),(40,d) |
| b | 2,0,(10,0),(15,0) |
| c | 2,0,(25,0),(30,e) |
| d | 2,0,(45,0),(50,0) |
| e | 1,0,(35,0) |

# Busca de um elemento

- Para buscar um valor **x** na árvore T inicia-se pela raiz, determinando o valor de **i** para o qual $K_i \le x < K_{i+1}$
  - por conveniência define-se $K_0 = [-\infty]$ e $K_{n+1} = [+\infty]$, onde $[-\infty]$ é menor do que todos os valores de chave permitidas e $[+\infty]$ é maior do que todos os valores de chave permitidas
- Se T é vazia então a busca falhou (**x** não se encontra em T)
- Se $x = K_i$ então a busca teve sucesso
- Se $x \neq K_i$, pela definição de uma AB em **m**-vias, então **x** só pode estar na subárvore $A_i$ (quando **x** se encontra na árvore), ou seja, a busca deve continuar a partir do nó $A_i$
- Quando **n** (o número de chaves em um nó) for "grande", a pesquisa do valor apropriado de **i** deve ser conduzida usando busca binária; para **n** "pequeno" uma busca seqüencial é mais adequada



# Busca de x=35




A busca inicia pela raiz (nó **a**), determinando i tal que $K_i \le x < K_{i+1}$. Como i=1 (e $x \neq K_1$) então a busca deve continuar na subárvore $A_1$ (nó **c**)

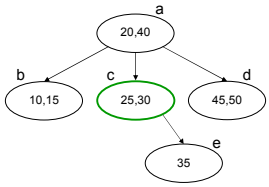
$n,A_0,(K_1,A_1),(K_2,A_2), \ldots ,(K_n,A_n)$

| Nó | Formato esquemático |
|---|---|
| a | 2,b,(20,c),(40,d) |
| b | 2,0,(10,0),(15,0) |
| c | 2,0,(25,0),(30,e) |
| d | 2,0,(45,0),(50,0) |
| e | 1,0,(35,0) |

| | $K_0$ | $K_1$ | $K_2$ | $K_3$ |
|---|---|---|---|---|
| a | [-∞] | 20 | 40 | [+∞] |



# Busca de x=35




A busca continua pelo nó **c**, determinando i tal que $K_i \le x < K_{i+1}$. Como i=2 (e $x \neq K_2$) então a busca deve continuar na subárvore $A_2$ (nó **e**)

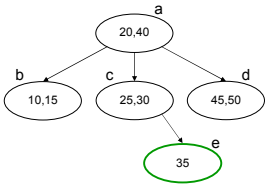
$n,A_0,(K_1,A_1),(K_2,A_2), \ldots ,(K_n,A_n)$

| Nó | Formato esquemático |
|---|---|
| a | 2,b,(20,c),(40,d) |
| b | 2,0,(10,0),(15,0) |
| c | 2,0,(25,0),(30,e) |
| d | 2,0,(45,0),(50,0) |
| e | 1,0,(35,0) |

| | $K_0$ | $K_1$ | $K_2$ | $K_3$ |
|---|---|---|---|---|
| c | [-∞] | 25 | 30 | [+∞] |



# Busca de x=35




A busca continua pelo nó **e**, determinando i tal que $K_i \le x < K_{i+1}$. Como i=1 (e $x = K_1$) então a busca termina com sucesso

$n,A_0,(K_1,A_1),(K_2,A_2), \ldots ,(K_n,A_n)$

| Nó | Formato esquemático |
|---|---|
| a | 2,b,(20,c),(40,d) |
| b | 2,0,(10,0),(15,0) |
| c | 2,0,(25,0),(30,e) |
| d | 2,0,(45,0),(50,0) |
| e | 1,0,(35,0) |

| | $K_0$ | $K_1$ | $K_2$ |
|---|---|---|---|
| e | [-∞] | 35 | [+∞] |



# Algoritmo de Busca

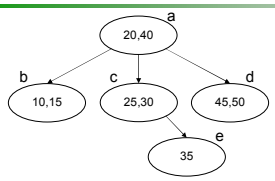
```
// pré: procure na árvore de busca de m-vias T residente em disco
   pela chave x. O formato do nó é n,A0,(K1,A1),...,(Kn,An), n < m.
// pós: retorna triplo (p,i,j); j = true significa que x foi
   encontrado no nó p, chave Ki. Caso contrário j = false e p é o
   nó onde x pode ser inserido

function MSearch(T, x)
 1  p ← T; q ← 0;              // q é o pai de p
 2  while p ≠ 0 do
 3    leia nó p do disco;
 4    seja p da forma: n,A0,(K1,A1),...,(Kn,An)
 5    K0 ← -∞; Kn+1 ← +∞;
 6    Encontre i tal que Ki ≤ x < Ki+1;
 7    if x = Ki then return (p,i,true); endif // x foi encontrado
 8    q ← p; p ← Ai;
 9  endwhile
    // x não está em T; retorne nó onde pode ser feita a inserção
10  return (q, i, false);
end MSearch
```



# Pontos Importantes

- Na prática, quando a árvore de busca representa um índice, os duplos ($K_i$, $A_i$) em cada nó devem ser acrescidos na forma de triplos ($K_i$, $A_i$, $B_i$), onde $B_i$ é o endereço do registro com a chave $K_i$ no arquivo
- No caso de discos
  - Esse endereço será formado pelos números de cilindro, superfície, trilha e setor que deve ser acessado para recuperar o registro
  - Os $A_i$'s, $0 \le i \le n$ são os números de endereços dos nós raiz das subárvores; Como esses nós encontram-se também no disco, os $A_i$'s são números de cilindro, superfície, trilha e setor
- No caso de arquivos de acesso aleatório
  - Esse endereço será formado pelos números de registros, cujos valores permitidos dependem da linguagem de programação utilizada. No caso de C++, os registros são numerados a partir de zero
  - Os $A_i$'s, $0 \le i \le n$ são números de registros

# Exemplo




$n,A_0,(K_1,A_1,B_1),(K_2,A_2,B_2),\ \ldots\ ,(K_n,A_n,B_n)$

Arquivo Índice

| Registro | Formato esquemático |
|---|---|
| a | 2,b,(20,c,#2),(40,d,#1) |
| b | 2,0,(10,0,#3),(15,0,#5) |
| c | 2,0,(25,0,#9),(30,e,#8) |
| d | 2,0,(45,0,#7),(50,0,#6) |
| e | 1,0,(35,0,#4) |

Arquivo Principal

| Registro | Chave | Demais campos |
|---|---|---|
| #1 | 40 | |
| #2 | 20 | |
| #3 | 10 | |
| #4 | 35 | |
| #5 | 15 | |
| #6 | 50 | |
| #7 | 45 | |
| #8 | 30 | |
| #9 | 25 | |



# Análise de MSearch

- O número máximo de acessos ao disco é igual à altura da árvore T
- A meta é minimizar a quantidade de acessos, necessários para conduzir uma busca, porque os acessos individuais ao disco são bastante dispendiosos em relação ao tempo necessário para processar um nó (para determinar o próximo nó a ser acessado, linhas 4-8); isso equivale a minimização da altura de árvore de busca



# Análise de MSearch

- Em uma árvore de grau **m** e altura **h** ≥ 1, a quantidade máxima de nós é (considerando a raiz no nível 1)

$$\sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1}$$

- Como cada nó tem, no máximo, (**m**−1) chaves, então o número máximo de entradas no índice de árvore em **m**-vias e de altura **h** é $\mathbf{m^h}-1$
- Para uma árvore binária com **h** = 3 este valor é 7, enquanto que para uma árvore com 200-vias e **h** = 3 tem-se $m^h-1 = 8 * 10^6 - 1$



# Análise de MSearch

- Dessa forma, o potencial das árvores de busca de ordem alta é muito maior do que daquelas de ordem baixa
- Para atingir um desempenho próximo das melhores árvores de busca em **m**-vias, para um determinado número de entradas **N**, é necessário que a árvore de busca seja equilibrada
- A variedade particular das árvores de busca em **m**-vias, equilibradas, que vamos considerar aqui, é conhecida como **árvores B**



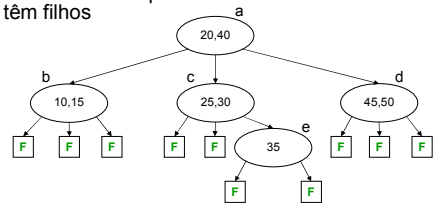
# Nós de Falha

- Para definir uma árvore B será conveniente introduzir o conceito de nós de falha
- Um **nó de falha** representa o nó que pode apenas ser alcançado, durante a pesquisa, se o valor X não se encontra na árvore
- Cada subárvore vazia (raiz=0) é o ponto que pode ser alcançado durante a pesquisa, apenas se X não está na árvore
- Essas subárvores vazias serão substituídas, por conveniência, por **nós hipotéticos** chamados de **nós de falha**
- Esses nós serão desenhados como quadrados e marcados com um '**F**'



# Nós de Falha

- Ressalta-se que a estrutura da árvore não contém efetivamente nenhum nó de falha, mas apenas o valor zero (ou outro valor indicando subárvore vazia) quando ele ocorrer
- É fácil verificar que os nós de falha são os únicos que não têm filhos



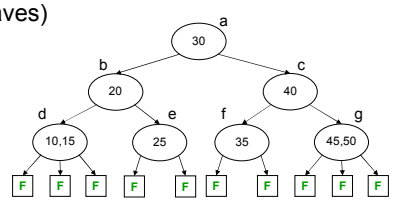
# Árvores B

- Uma árvore T do tipo B e ordem **m** é uma árvore de busca em **m**-vias, que é ou vazia ou de altura **h** ≥ 1 e satisfaz as seguintes propriedades:
  - (a) a raiz tem pelo menos 2 filhos
  - (b) todos os nós, exceto a raiz e nós de falha têm, no mínimo, $\lceil m/2 \rceil$ filhos (ou seja, $\lceil m/2 \rceil$-1 chaves)
  - (c) todos os nós de falha estão no mesmo nível
- Lembrar que $\lceil x \rceil$ é o teto de **x**, ou seja, é o menor inteiro ≥ **x**
- A raiz da árvore encontra-se no nível 1



# Árvores B

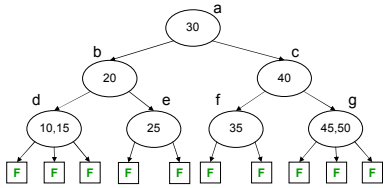
- A árvore seguinte é uma árvore B de ordem 3
- Observe que todos os nós, exceto a raiz e nós de falha, têm pelo menos $\lceil 3/2 \rceil = 2$ filhos ($\equiv \lceil 3/2 \rceil$-1=1 chaves)






# Árvores B

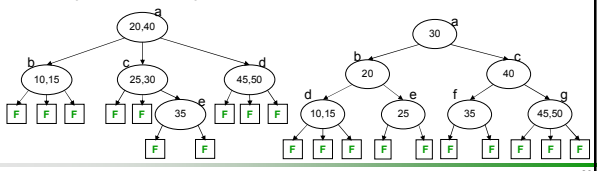
- Note que todos os nós, exceto os de falha, são de grau 2 ou 3
- De fato, a definição de uma árvore B de ordem 3 implica que todos os nós, exceto os de falha, devem ter grau 2 ou 3
- Por esse motivo, as árvores B de ordem 3 são também conhecidas como árvores 2-3






# Árvores B

- Enquanto o número total de nós não-de-falha pode ser maior em uma árvore B, do que o número de tais nós na melhor árvore de busca em **m**-vias de mesma ordem (a árvore da esquerda tem apenas 5 nós, enquanto a árvore da direita 7) constataremos que é mais fácil inserir e eliminar nós na árvore B do que manter em qualquer tempo, a melhor possível árvore de busca em **m**-vias






# Árvores B

- As razões de usarmos para índices as árvores B, ao invés de árvores de busca em **m**-vias otimizadas, são as mesmas daqueles que determinaram o uso de árvores AVL para manter as tabelas internas dinâmicas e não as árvores de busca binária otimizadas



# Número de Chaves

- Se T é uma árvore B de ordem **m**, na qual todos os nós de falha estão no nível **h**+1 então a quantidade máxima de entradas de índice de T é $m^h$-1
- A quantidade mínima **N** de chaves em T pode ser determinada usando a definição de uma árvore do tipo B, que no caso de **h** > 1 o nó raiz tem pelo menos 2 filhos
- Portanto, há pelo menos dois nós no nível 2
- Cada um desses nós deve ter pelo menos $\lceil m/2 \rceil$ filhos
- Por isso há, pelo menos, $2*\lceil m/2 \rceil$ nós no nível 3
- No nível 4 deve ter, pelo menos, $2*\lceil m/2 \rceil^2$ e continuando este raciocínio, observamos que haverá, pelo menos, $2*\lceil m/2 \rceil^{h-2}$ nós no nível **h**, quando **h** > 1
- Todos esses nós são não-de-falha

## Número de Chaves

- Se os valores de chave na árvore são $K_1$, $K_2$, ..., $K_N$ e $K_i<K_{i+1}$, $1 \le i < N$ então existem $N+1$ nós de falha
- Isso é assim, porque os nós de falhas acontecem para $K_i < X < K_{i+1}$ $0 \le i \le N$ e $K_0 = [-\infty]$, $K_{N+1} = [+\infty]$
- Ou seja, pode-se alcançar $N+1$ nós de falha distintos, buscando um valor de $X$ que não está na T
- Por isso, temos
  - $N+1$ = quantidade de nós de falha em T
  - $N+1$ = quantidade de nós no nível $h+1$
  - $N+1 \ge 2*\lceil m/2 \rceil^{h-1}$
  - e, desse modo, $N \ge 2*\lceil m/2 \rceil^{h-1}-1$, $h \ge 1$



## Número de Chaves

- Por sua vez, isso implica que havendo $N$ chaves em uma árvore B de ordem $m$, então todos os nós não-de-falha encontram-se nos níveis menores do que ou igual à $h$, $h \le \log_{\lceil m/2 \rceil}((N+1)/2) + 1$
- O número máximo de acessos que precisam ser feitos para uma busca é $h$
- Usando uma árvore B de ordem $m = 200$ e o índice com $N \le 2*10^6-6$ temos $h \le \log_{100}((N+1)/2) + 1$, ou seja, $h \le 3$ (já que $h$ é um inteiro)
- Para $N \le 2*10^7$ temos $h \le 4$
- Por essa razão, a utilização de árvores B de ordem alta, resulta em um índice de árvore que pode ser pesquisado, minimizando o número de acessos ao disco, mesmo para uma quantidade muito grande de entradas



## Escolha do m

- Dessa forma, sabemos que Árvores B de alta ordem são desejáveis, pois resultam em uma redução de número de acessos ao disco, necessários para pesquisar um índice
- Para um índice de $N$ entradas, uma árvore B de ordem $m=N+1$ teria apenas um nível
- Como supomos que o índice é grande demais para ser acomodado na memória interna, essa escolha de $m$ claramente não seria razoável
- Conseqüentemente, um único nó representando o índice não caberia na memória principal para ser processado
- Para encontrar uma escolha razoável de $m$, precisamos nos lembrar que estamos interessados na minimização do tempo total necessário para pesquisar o valor X na árvore B



## Escolha do m

- Suponhamos que cada nó da árvore B de ordem $m$ é de tamanho fixo e suficientemente grande para acomodar $n$, $A_0$ e $m$-1 valores para ($K_i$, $A_i$, $B_i$), $1 \le i < m$
- Se os $K_i$'s ocupam, no máximo, $\alpha$ bytes e ambos $A_i$ e $B_i$ cada um $\beta$ bytes então o tamanho de um nó será de aproximadamente $m*(\alpha+2*\beta)$ bytes
- O tempo $t_i$ para ler um nó será então:
  - $t = t_p + t_i + m*(\alpha+2*\beta)*t_b = a + b*m$, onde
  - $a = t_p+t_i$ = tempo de posicionamento + tempo de latência
  - $b = (\alpha+2*\beta)*t_b$, e $t_b$ = tempo de transmissão por byte
- Se a busca binária for usada para conduzir a pesquisa da linha 6 do algoritmo MSearch então o tempo de processamento interno por nó será $c*\log_2(m)+d$ para algumas constantes $c$ e $d$
- Desse modo, o tempo total de processamento por nó é
  - $\tau = a + b*m + c*\log_2 m + d$



## Escolha do m

- Desse modo, o tempo total de processamento por nó é
  - $\tau = a + b*m + c*\log_2 m + d$
- O número de níveis $h$, para um índice de $N$ entradas, é limitado por:
  - $h \le \log_{\lceil m/2 \rceil}((N+1)/2) + 1$
  - $h \le f*(\log_2((N+1)/2) + 1)/\log_2 m$, para alguma constante $f$
- Logo, o tempo máximo de pesquisa é $h*\tau$:

$$g\left\{\frac{a+d}{\log_2 m}+\frac{b\times m}{\log_2 m}+c\right\}$$

  - onde $g = f*\log_2((N+1)/2) + 1$

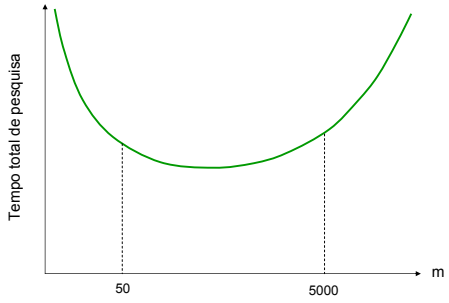


## Escolha do m

- Procuramos um valor de $m$ que minimize o tempo máximo de pesquisa
- Assumindo que a unidade de disco disponível tem $t_p$=10 ms e $t_i$ = 8 ms, teremos $a$=18 ms
- Como $d$ será tipicamente apenas alguns nano segundos, podemos desprezá-lo em comparação com $a$, ou seja, $a+d \approx a$ = 18ms
- Assumindo que cada valor de chave tem, no máximo, 6 bytes e que cada $A_i$ e $B_i$ ocupam, cada um, 3 bytes, temos $\alpha$=6, $\beta$=3
- Se a velocidade de transmissão for $t_b=5*10^{-3}$ ms/byte então $b = (\alpha+2*\beta)*t_b$ = 0.06 ms
- A equação anterior fica

$$g\left\{\frac{18}{\log_2 m}+\frac{0.06\times m}{\log_2 m}+c\right\}$$

# Escolha do m





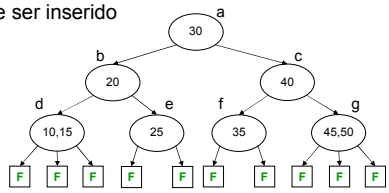

# Inserção

- Além de pesquisar o índice para encontrar um valor de chave específico X, queremos também inserir e eliminar as entradas
- A inserção deve ser conduzida de tal maneira, que a árvore conseguida depois da inserção seja também uma árvore do tipo B
- O algoritmo para fazer isso é conceitualmente mais simples do que o algoritmo de inserção correspondente para uma árvore AVL



# Inserção X = 38

- Com o intuito de descobrir as operações necessárias para conduzir a inserção, considere a inserção de X = 38 na árvore
- Primeiro pesquisa-se a árvore para determinar onde 38 deve ser inserido

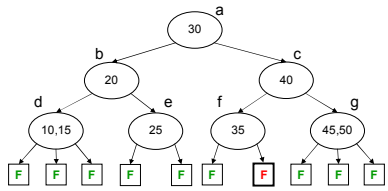





# Inserção X = 38

| R | W |
|---|---|
| 3 | |

- O nó de falha que se alcança durante a pesquisa de X = 38 é o quarto da direita; o nó pai deste nó de falha é **f**
- O nó **f** contém apenas uma chave, e assim, tem espaço para outra

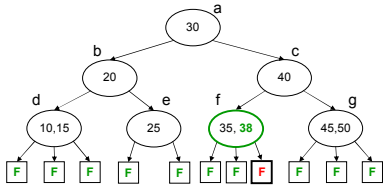





# Inserção X = 38

| R | W |
|---|---|
| 3 | 1 |

- Logo, X = 38 pode ser inserido no nó **f**, obtendo-se a árvore seguinte

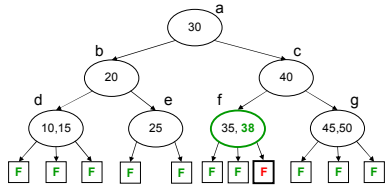





# Inserção X = 38

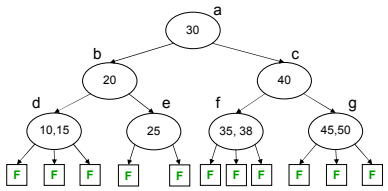
| R | W |
|---|---|
| 3 | 1 |

- Fazendo esta inserção, três nós **a**, **c** e **f** foram acessados no disco durante a pesquisa
- Além disso, o novo nó **f** precisa ser gravado de volta no disco
- A quantidade total de acessos é, portanto, quatro



# Inserção X = 55

- A seguir, vamos inserir X = 55

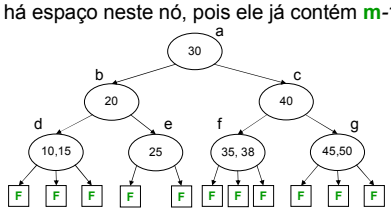





# Inserção X = 55

| R | W |
|---|---|
| 3 | |

- A seguir, vamos inserir X = 55
- Uma pesquisa na árvore B indica que esta chave deve ser encaminhada para o nó **g**
- Não há espaço neste nó, pois ele já contém **m**-1 chaves






# Inserção (Eq.1)

- Inserindo simbolicamente o novo valor de chave em sua posição apropriada, em um nó que anteriormente tinha **m**-1 chaves, produzirá um nó P, com o seguinte formato
  - P: **m**, $A_0$, ($K_1$,$A_1$), ($K_2$,$A_2$),..., ($K_m$,$A_m$)
  - Com $K_i$<$K_{i+1}$,1 ≤ **i** < **m**
- Esse nó pode ser fracionado em dois nós P e Q com os seguintes formatos (Equação 1)
  - P: $\lceil m/2 \rceil$-1, $A_0$, ($K_1$,$A_1$), ..., ($K_{\lceil m/2 \rceil-1}$,$A_{\lceil m/2 \rceil-1}$)
  - Q: **m**-$\lceil m/2 \rceil$, $A_{\lceil m/2 \rceil}$, ($K_{\lceil m/2 \rceil+1}$, $A_{\lceil m/2 \rceil+1}$), ..., ($K_m$,$A_m$)
- O valor restante $K_{\lceil m/2 \rceil}$ e o novo nó Q formam o par ($K_{\lceil m/2 \rceil}$,Q) e é feita uma tentativa de inserir este par no pai de P



# Inserção (Eq.1)

- Assumindo **m**=3, temos
  - P: 3, $A_0$, ($K_1$,$A_1$), ($K_2$,$A_2$),($K_3$,$A_3$)
  - Com $K_i$<$K_{i+1}$,1 ≤ **i** < 3
- Esse nó pode ser fracionado em dois nós P e Q com os seguintes formatos (Equação 1)
  - P: 1, $A_0$, ($K_1$,$A_1$)
  - Q: 1, $A_2$, ($K_3$,$A_3$)
- O valor restante $K_2$ e o novo nó Q formam o par ($K_2$,Q) e é feita uma tentativa de inserir este par no pai de P



# Inserção (Eq.1)

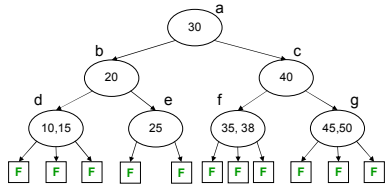
- Assumindo **m**=4, temos
  - P: 4, $A_0$, ($K_1$,$A_1$), ($K_2$,$A_2$),($K_3$,$A_3$),($K_4$,$A_4$)
  - Com $K_i$<$K_{i+1}$,1 ≤ **i** < 4
- Esse nó pode ser fracionado em dois nós P e Q com os seguintes formatos (Equação 1)
  - P: 1, $A_0$, ($K_1$,$A_1$)
  - Q: 2, $A_2$, ($K_3$,$A_3$),($K_4$,$A_4$)
- O valor restante $K_2$ e o novo nó Q formam o par ($K_2$,Q) e é feita uma tentativa de inserir este par no pai de P



# Inserção X = 55

| R | W |
|---|---|
| 3 | |

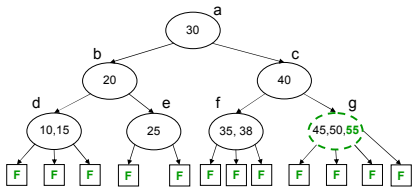
- Considerando o nó **g**:
  - 2, 0, (45,0), (50,0)



# Inserção X = 55

| R | W |
|---|---|
| 3 | |

- Considerando o nó **g**:
  - 2, 0, (45,0), (50,0)
- Após a inserção simbólica de X=55, o nó **g** fica:
  - 3, 0, (45,0), (50,0), (55, 0)

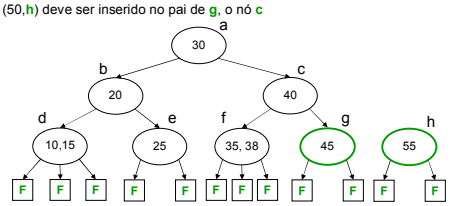





# Inserção X = 55

| R | W |
|---|---|
| 3 | 2 |

- Após a inserção simbólica de X=55, o nó **g** fica como:
  - 3, **0**, (**45**,**0**), (50,**0**), (**55**, **0**)
- Portanto, o nó **g** precisa ser fracionado em:
  - **g**: 1, **0**, (**45**,**0**)
  - **h**: 1, **0**, (**55**,**0**)
- O par (50,**h**) deve ser inserido no pai de **g**, o nó **c**

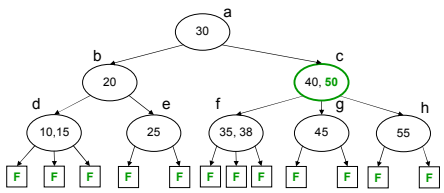





# Inserção X = 55

| R | W |
|---|---|
| 3 | 3 |

- Como o nó **c** tem apenas uma chave, a inserção do par (50,**h**) pode ser feita facilmente






# Inserção X = 55

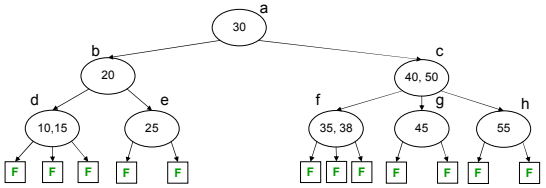
| R | W |
|---|---|
| 3 | 3 |

- Desta vez foram feitos 3 acessos para determinar que X = 55 deve ser inserido em **g**
- Como o nó **g** foi alterado, precisa ser gravado no disco
- Assumindo que o nó **c** encontra-se ainda na memória interna, será necessário um outro acesso ao disco, para gravar o novo nó **c**
- Logo, o número total de acessos é 6



# Exercício

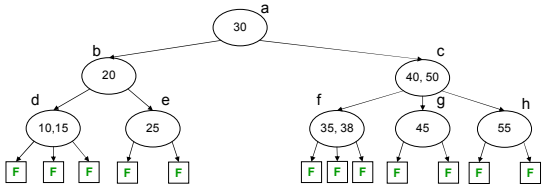
- Inserir os valores 37, 5, 18 e 12 na árvore B, anotando quantos acessos a disco são necessários






# Inserção X = 37

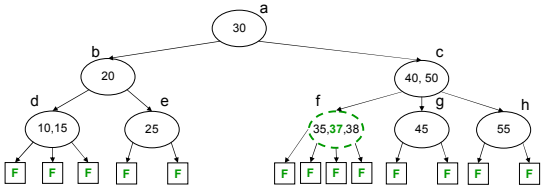
- Ao inserir X = 37...



# Inserção X = 37

| R | W |
|---|---|
| 3 | |

- Ao inserir X = 37 no nó **f**, ele fica grande demais e precisa ser particionado (em **f** e **i**)

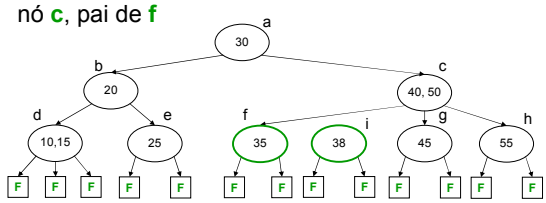





# Inserção X = 37

| R | W |
|---|---|
| 3 | 2 |

- Ao inserir X = 37 no nó **f**, ele fica grande demais e precisa ser particionado (em **f** e **i**)
- O par (37,**i**) deve ser inserido simbolicamente no nó **c**, pai de **f**

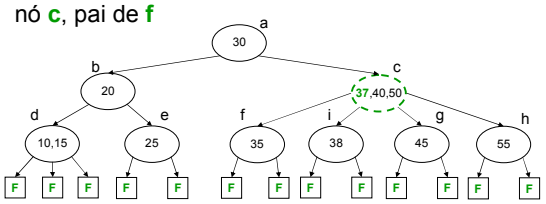





# Inserção X = 37

| R | W |
|---|---|
| 3 | 2 |

- Ao inserir X = 37 no nó **f**, ele fica grande demais e precisa ser particionado (em **f** e **i**)
- O par (37,**i**) deve ser inserido simbolicamente no nó **c**, pai de **f**

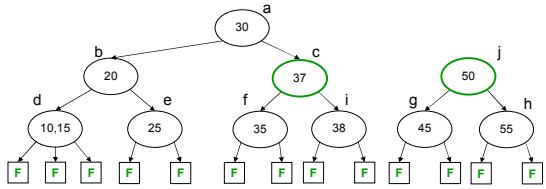





# Inserção X = 37

| R | W |
|---|---|
| 3 | 4 |

- Ao inserir o par (37,**i**) no nó **c**, ele fica grande demais e precisa ser particionado (em **c** e **j**)
- O par (40,**j**) deve ser inserido no nó **a**, pai de **c**

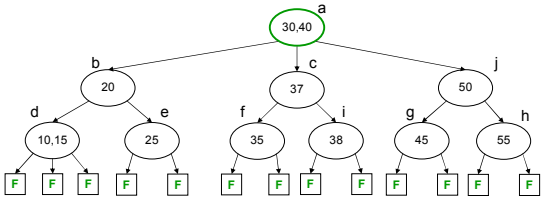





# Inserção X = 37

| R | W |
|---|---|
| 3 | 5 |

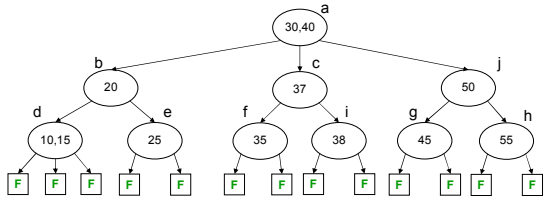
- Como o nó **a** tem apenas uma chave, a inserção do par (40,**j**) é simples
- Foram feitos 8 acessos a disco






# Inserção X = 5

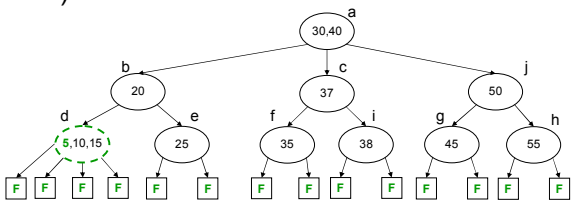
- Ao inserir X = 5...



# Inserção X = 5

| R | W |
|---|---|
| 3 | |

- Ao inserir X = 5 no nó **d**, ele fica grande demais e precisa ser particionado (em **d** e **k**)

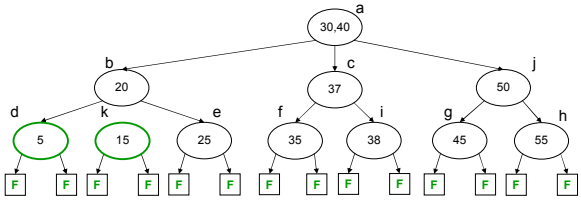





# Inserção X = 5

| R | W |
|---|---|
| 3 | 2 |

- Ao inserir X = 5 no nó **d**, ele fica grande demais e precisa ser particionado (em **d** e **k**)
- O par (10,**k**) deve ser inserido no nó **b**, pai de **d**

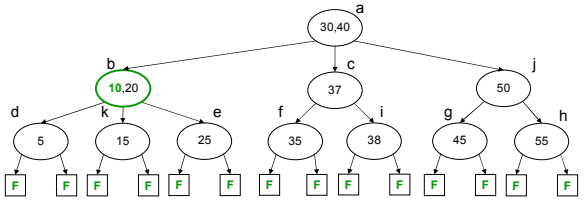





# Inserção X = 5

| R | W |
|---|---|
| 3 | 3 |

- Como o nó **b** tem apenas uma chave, a inserção do par (10,**k**) é simples
- Foram feitos 6 acessos a disco

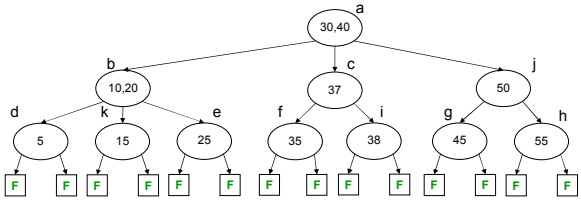





# Inserção X = 18

| R | W |
|---|---|
| 3 | |

- A inserção de X = 18...

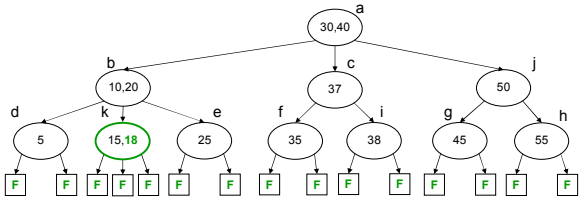





# Inserção X = 18

| R | W |
|---|---|
| 3 | 1 |

- A inserção de X = 18 é trivial, com 4 acessos a disco

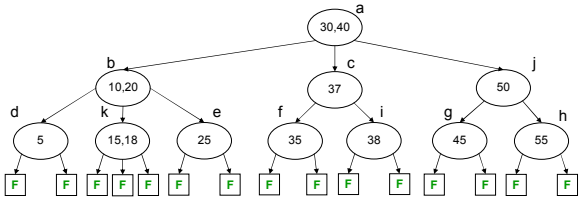





# Inserção X = 12

| R | W |
|---|---|
| 3 | |

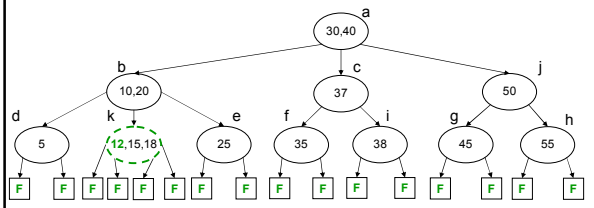
- Ao inserir X = 12...



# Inserção X = 12

| R | W |
|---|---|
| 3 | |

- Ao inserir X = 12 no nó **k**, ele fica grande demais e precisa ser particionado (em **k** e **l**)

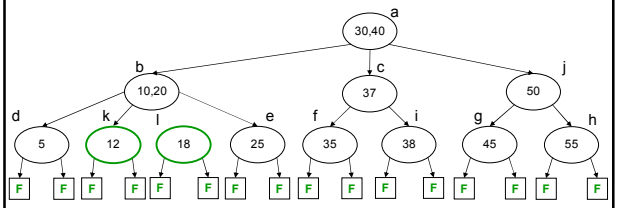





# Inserção X = 12

| R | W |
|---|---|
| 3 | 2 |

- Ao inserir X = 12 no nó **k**, ele fica grande demais e precisa ser particionado (em **k** e **l**)
- O par (15,**l**) deve ser inserido no nó **b**, pai de **k**

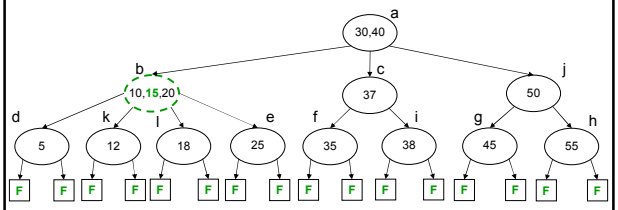





# Inserção X = 12

| R | W |
|---|---|
| 3 | 2 |

- Ao inserir o par (15,**l**) no nó **b**, ele fica grande demais e precisa ser particionado (em **b** e **q**)

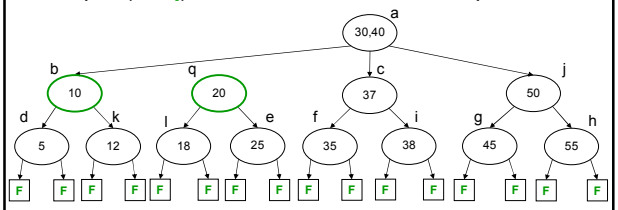





# Inserção X = 12

| R | W |
|---|---|
| 3 | 4 |

- Ao inserir o par (15,**l**) no nó **b**, ele fica grande demais e precisa ser particionado (em **b** e **q**)
- O par (15,**q**) deve ser inserido no nó **a**, pai de **b**

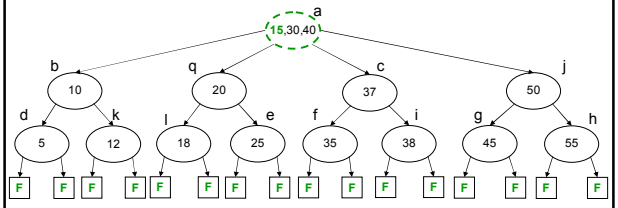





# Inserção X = 12

| R | W |
|---|---|
| 3 | 4 |

- Ao inserir o par (15,**q**) no nó **a**, ele fica grande demais e precisa ser particionado (em **a** e **r**)

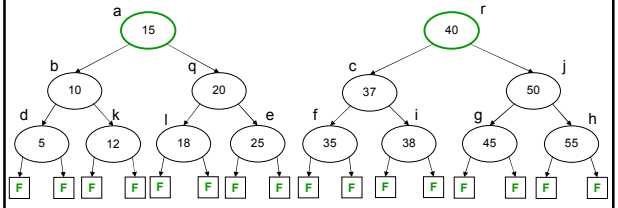





# Inserção X = 12

| R | W |
|---|---|
| 3 | 6 |

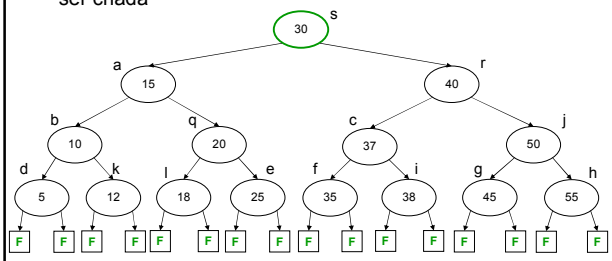
- Ao inserir o par (15,**q**) no nó **a**, ele fica grande demais e precisa ser particionado (em **a** e **r**)
- O par (30,**r**) deve ser inserido no pai de **a**



# Inserção X = 12

| R | W |
|---|---|
| 3 | 7 |

- Como **a** é a raiz, uma nova raiz da forma 1,**a**,(30,**r**) deve ser criada






# Inserção X = 12

- Para inserir X=12 foram acessados no disco os nós **a**, **b** e **k**
- Admitimos que existe uma memória interna disponível suficiente para manter estes três nós na memória
- A inserção dentro do nó **k** requer que ele seja fracionado em dois nós **k** e **l**
- O par (15,**l**) precisa ser inserido no nó pai **b**
- Os novos nós **k** e **l** são gravados em disco
- A inserção em **b** resulta em fracionamento deste nó em dois nós **b** e **q**, sendo ambos gravados em disco
- O par (15,**q**) deve ser inserido no nó **a**, que também se fraciona
- Os novos nós **a** e **r** são gravados, deixando o espaço para inserir o par (30,**r**) no nó pai de **a**
- O nó **a** é o nó raiz e por isso não tem pai
- Assim, cria-se um novo nó raiz **s** com as subárvores **a** e **r**
- A nova raiz **s** é gravada em disco e a altura da árvore B aumenta 1 unidade
- É de 10 o número total de acessos necessários para esta inserção



# Algoritmo de Inserção

- Pode-se verificar que as transformações de inserção, descritas anteriormente, preservam o índice como uma árvore do tipo B e cobrem todas as possibilidades
- O algoritmo de inserção resultante, **InsertB**, assume que todos os nós, no caminho da raiz até o ponto de inserção, são empilhados
- Não tendo memória disponível bastará empilhar apenas os endereços e **pai(p)** poderá ser usado
- No algoritmo **InsertB** o valor da chave X é inserido na árvore T de tipo B de ordem **m**
- Assume-me que T reside em disco e que a raiz da árvore B é o nó T



# Algoritmo de Inserção

```
procedure InsertB(T, X)
 1 A ← 0; K ← X;                  // (K,A) é o par a ser inserido
 2 (p,i,achou) ← MSearch(T,X);  // p é o nó para inserção
 3 if achou then return; endif  // X já está em T
 4 while p ≠ 0 do
 5   insira (K,A) nas posições apropriadas de p
     Seja o nó p resultante da forma: n, A0, (K1, A1), ..., (Kn, An)
 6   if n ≤ m-1 then // nó resultante não é grande demais
       Escreva p para o disco; return;
     endif
     // p precisa ser fracionado
 7   Sejam p e q definidos conforme Equação 1 (slide Inserção (Eq.1));
 8   Escreva p e q para o disco;
 9   K ← K⌈m/2⌉; A ← q; p ← pai(p);
10 endwhile
   // a nova raiz deve ser criada
11 crie um novo nó R com formato 1,T,(K,A)
12 T ← R;
13 Escreva T para o disco
end InsertB
```



# Análise InsertB

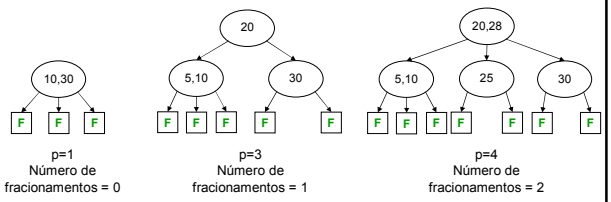
- Se a árvore B tem **h** níveis então a pesquisa da linha 2 vai necessitar de **h** acessos, pois X não está em T
- Cada vez que um nó fraciona-se (linha 7) são feitos dois acessos ao disco (linha 8)
- Depois do fracionamento final é feito ainda um acesso, seja da linha 6 ou linha 13
- Sendo o número total de nós que se fracionam igual a **k** então o número total de acessos ao disco é **h**+2***k**+1
- Este valor presume que exista a memória interna disponível, suficiente para manter todos os nós acessados durante a chamada de **MSearch** da linha 2
- Como no máximo um nó pode ser fracionado em cada um dos **h** níveis, **k** ≤ 1 sempre (em cada nível)
- Portanto, o número máximo de acessos será 3***h**+1
- No pior caso, como em situações práticas, **h** não excederá a 4, o que significaria em torno de treze acessos necessários para fazer uma inserção



# Análise InsertB

- No caso de algoritmo **InsertB**, o valor de pior caso de 3***h**+1 acessos não reflete a história completa
- O número médio de acessos é consideravelmente inferior a isso
- Uma estimativa do número de acessos médio, necessário para uma inserção pode ser obtido se começarmos com uma árvore vazia e inserirmos nela **N** valores; então o número total de fracionamentos de nós é, no máximo, (**p**-2), onde **p** é o número de nós não-de-falha na árvore B final, com **N** chaves
- Esta limitação superior de (**p**-2) origina-se da observação de que cada vez que um nó é fracionado, cria-se pelo menos um nó adicional
- Quando a raiz é fracionada, criam-se dois nós adicionais
- O primeiro nó criado não tem origem de fracionamento e se uma árvore B tiver mais do que um único nó, então a sua raiz deve ter se fracionado pelo menos uma vez
- O valor (**p**-2) é a melhor limitação superior possível de fracionamentos de nós, na criação de uma árvore B com **p** nós
  - **p** > 2 (repare que não existe uma árvore B com **p** = 2)

# Análise InsertB






# Análise InsertB

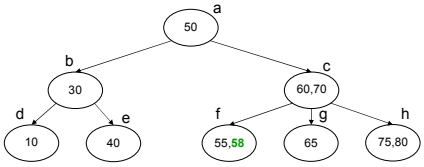
- Uma árvore B de ordem **m** com **p** nós, tem pelo menos
  - $1 + (\lceil m/2 \rceil - 1)*(p-1)$
- chaves, pois a raiz tem pelo menos um valor de chave e os nós restantes pelo menos $\lceil m/2 \rceil - 1$ chaves
- A quantidade média de fracionamentos **s**, pode ser determinada agora como
  - **s** = (quantidade total de fracionamentos)/**N**
  - $s \leq (p-2)/ (1 + (\lceil m/2 \rceil - 1)*(p-1))$
  - $s < 1/(\lceil m/2 \rceil - 1)$
- Para **m**=200 a quantidade de fracionamentos é inferior a 1/99 por chave inserida
  - Logo, o número médio de acessos ao disco é de apenas $h+2*s+1 < h+101/99 \approx h+1$



# Remoção X = 58

| R | W |
|---|---|
| 3 | |

- Vamos analisar o problema de eliminar os valores dos nós folha
- A remoção do valor da chave X = 58...

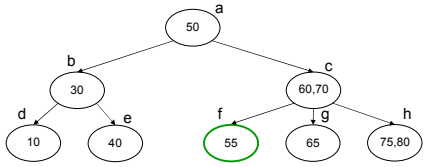





# Remoção X = 58

| R | W |
|---|---|
| 3 | 1 |

- Vamos analisar o problema de eliminar os valores dos nós folha
- A remoção do valor da chave X = 58 do nó **f** é fácil, já que a retirada deste valor de chave deixa o nó ainda com um outro valor de chave, que é o mínimo exigido para o nó diferente da raiz
- Uma vez que foi determinado que X = 58 está no nó **f**, o único acesso adicional será para regravar no disco o novo nó **f**

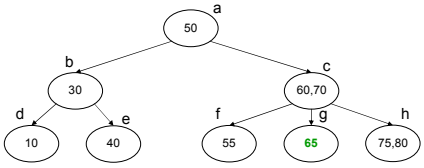





# Remoção X = 65

| R | W |
|---|---|
| 3 | |

- A remoção de X = 65, do nó **g**, resulta em que o número de chaves restantes fique abaixo do mínimo exigido de $\lceil m/2 \rceil - 1$

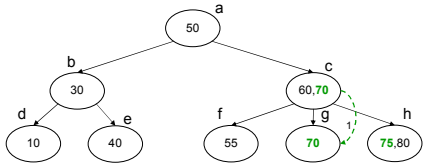





# Remoção X = 65

| R | W |
|---|---|
| 4 | |

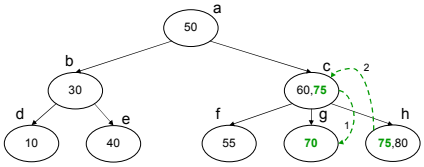
- A remoção de X = 65, do nó **g**, resulta em que o número de chaves restantes fique abaixo do mínimo exigido de $\lceil m/2 \rceil - 1$
- Um exame do irmão direito mais próximo **h**, do nó **g**, indica que ele tem $\geq \lceil m/2 \rceil$ chaves e então a chave de menor valor de **h**, isto é, 75 é movida para cima, para o nó pai **c**; antes disso, o valor 70 é movido do pai **c** para o filho **g**



# Remoção X = 65

| R | W |
|---|---|
| 4 | |

- A remoção de X = 65, do nó **g**, resulta em que o número de chaves restantes fique abaixo do mínimo exigido de $\lceil m/2 \rceil - 1$
- Um exame do irmão direito mais próximo **h**, do nó **g**, indica que ele tem $\geq \lceil m/2 \rceil$ chaves e então a chave de menor valor de **h**, isto é, 75 é movida para cima, para o nó pai **c**; antes disso, o valor 70 é movido do pai **c** para o filho **g**

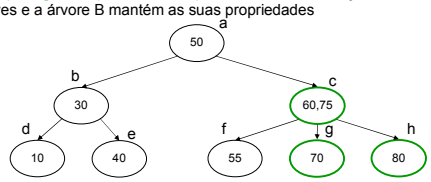





# Remoção X = 65

| R | W |
|---|---|
| 4 | 3 |

- A remoção de X = 65, do nó **g**, resulta em que o número de chaves restantes fique abaixo do mínimo exigido de $\lceil m/2 \rceil - 1$
- Um exame do irmão direito mais próximo **h**, do nó **g**, indica que ele tem $\geq \lceil m/2 \rceil$ chaves e então a chave de menor valor de **h**, isto é, 75 é movida para cima, para o nó pai **c**; antes disso, o valor 70 é movido do pai **c** para o filho **g**
- Como resultado dessas transformações, o número de chaves é $> \lceil m/2 \rceil - 1$ tanto para **g** como para **h**; o nó **c** permanece sem alteração em número de chaves e a árvore B mantém as suas propriedades






# Remoção X = 65

| R | W |
|---|---|
| 4 | 3 |

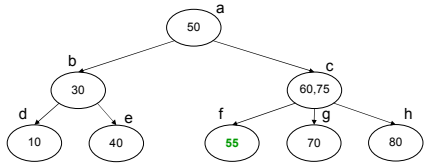
- Durante a pesquisa para encontrar X = 65 foram acessados os nós **a**, **c** e **g**
- Se todos esses três nós podem ser mantidos em memória, então pode ser determinado a partir do nó **c** o endereço de **h** como o irmão mais próximo de **g**; o nó **h** então é acessado
- Desde que três nós foram alterados (**g**, **c** e **h**), devem ser feitos mais três acessos para gravar novamente esses nós no disco
- Assim, precisam ser feitos mais quatro acessos, além dos acessos feitos para pesquisar a localização do X = 65, para efetuar a eliminação de X = 65
- Se **g** não tivesse um irmão direito próximo, ou se esse irmão tiver apenas a quantidade mínima de valores de chave, isto é, $\lceil m/2 \rceil - 1$ poder-se-ia fazer a mesma coisa com o irmão esquerdo mais próximo



# Remoção X = 55

| R | W |
|---|---|
| 3 | |

- Removendo X = 55 observamos que o irmão direito **g**, mais próximo de **f**, tem apenas $\lceil m/2 \rceil - 1$ chaves e que **f** não tem um irmão esquerdo próximo
- Conseqüentemente, é necessária uma transformação diferente

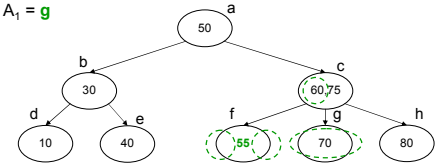





# Remoção X = 55

| R | W |
|---|---|
| 4 | |

- Se o nó **c** (pai de **f**) estiver na forma
  - **c**: $n, A_0, (K_1, A_1), \ldots, (K_n, A_n)$ e
  - $A_i$ = **g**
- Então podemos combinar as chaves restantes de **f**, com as chaves de **g** e a chave $K_i$ para formar um novo **g**
- No Exemplo
  - **c**: 2, **f**, (60,**g**), (75,h)
  - $A_1$ = **g**

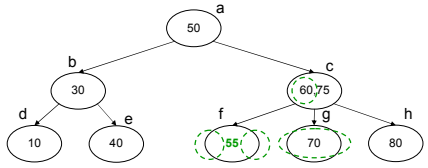





# Remoção X = 55

| R | W |
|---|---|
| 4 | |

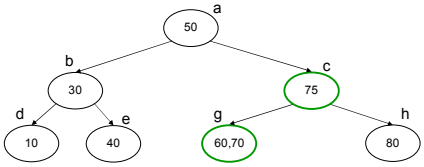
- O novo nó g terá $(\lceil m/2 \rceil - 2) + (\lceil m/2 \rceil - 1) + 1 = 2 * \lceil m/2 \rceil - 2 \leq m - 1$ chaves presentes
  - $\lceil m/2 \rceil$ - 2 chaves restantes em **f**
  - $\lceil m/2 \rceil$ - 1 chaves em **g**
  - 1 chave emprestada do pai **c**



# Remoção X = 55

| R | W |
|---|---|
| 4 | 2 |

- São necessários três acessos adicionais (além daqueles necessários à busca)
  - um para constatar que **g** está pequeno demais e
  - um de cada para regravar **g** e **c**

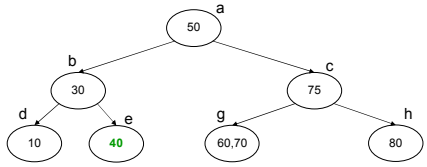





# Remoção X = 40

| R | W |
|---|---|
| 3 | |

- A remoção de X = 40 deixa o nó **e** com ⌈**m**/2⌉-2 chaves
- O seu irmão esquerdo **d** não tem nenhuma chave para emprestar

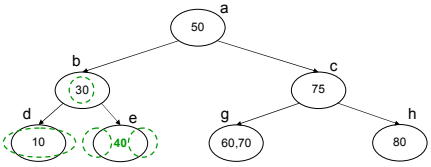





# Remoção X = 40

| R | W |
|---|---|
| 4 | |

- Nesse caso, as chaves do nó **d** são combinadas com aquelas restantes do nó **e** e a chave 30 do nó pai (**b**) para obter o novo nó **d** cheio, com os valores 10 e 30

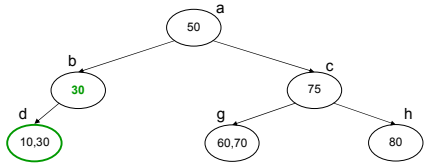





# Remoção X = 40

| R | W |
|---|---|
| 4 | 1 |

- Nesse caso, as chaves do nó **d** são combinadas com aquelas restantes do nó **e** e a chave 30 do nó pai (**b**) para obter o novo nó **d** cheio, com os valores 10 e 30
- Isso, porém, deixa **b** com uma chave a menos do permitido

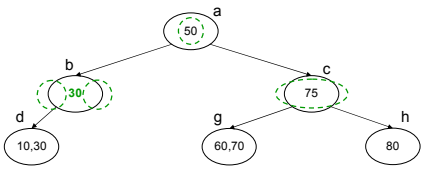





# Remoção X = 40

| R | W |
|---|---|
| 5 | 1 |

- Isso, porém, deixa **b** com uma chave a menos do permitido
- O seu irmão **c** não tem chaves extras e por isso, as chaves restantes de **b** são combinadas, com o valor 50 de **a** e ⌈**m**/2⌉-1 chaves de **c**, para conseguir um novo nó **c**

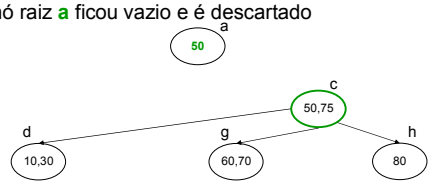





# Remoção X = 40

| R | W |
|---|---|
| 5 | 2 |

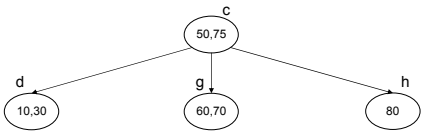
- Isso, porém, deixa **b** com uma chave a menos do permitido
- O seu irmão **c** não tem chaves extras e por isso, as chaves restantes de **b** são combinadas, com o valor 50 de **a** e ⌈**m**/2⌉-1 chaves de **c**, para conseguir um novo nó **c**
- O nó raiz **a** ficou vazio e é descartado



# Remoção X = 40

| R | W |
|---|---|
| 5 | 2 |

- Isto toma um total de quatro acessos adicionais (1 para buscar cada um dos nós **d** e **c**, 1 para regravar cada um dos nós **c** e **d** alterados), totalizando 7 acessos





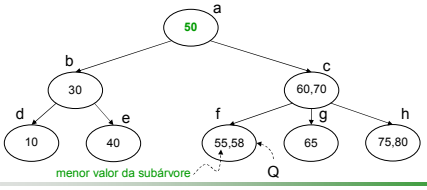

# Remoção

- Quando a chave X a ser eliminada não se encontra em uma folha então pode ser feita uma simples transformação que permitirá efetuar a remoção como no caso de um nó folha
- Suponha que $K_i$=X no nó P que tem a forma
  - **n**, $A_0$, ($K_1$,$A_1$), ..., ($K_n$, $A_n$) com $1 \le i \le$ **n**
- Como P não é uma folha então $A_i \neq 0$
- Podemos determinar Y como o menor valor de chave na subárvore $A_i$, que se encontrará em um nó folha Q
- Substitua $K_i$ por Y no P e grave o novo P
- Isso nos deixa com o problema de eliminar Y do Q
- Note que estão sendo mantidas as propriedades de pesquisa da árvore



# Remoção

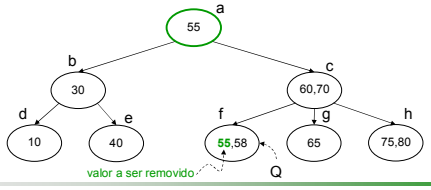
- Por exemplo, a remoção de X = 50 pode ser conseguida substituindo primeiro 50 do nó **a** por 55 do nó **f** e então procedendo com a remoção de 55 do nó **f**






# Remoção

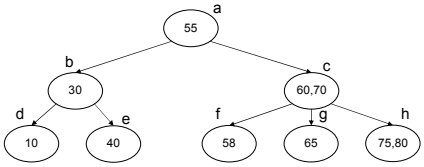
- Por exemplo, a remoção de X = 50 pode ser conseguida substituindo primeiro 50 do nó **a** por 55 do nó **f** e então procedendo com a remoção de 55 do nó **f**






# Exercício

- Remova a chave 55 da árvore B, anotando quantos acessos a disco são necessários

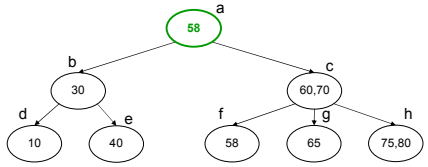





# Remoção X = 55

| R | W |
|---|---|
| 3 | 1 |

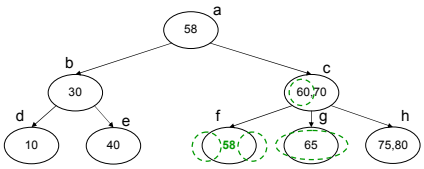
- A chave 55 (nó **a**) é substituída pela chave 58 (nó **f**), bastando remover a chave 58 do nó **f**



# Remoção X = 55

| R | W |
|---|---|
| 4 | 1 |

- A chave 55 (nó **a**) é substituída pela chave 58 (nó **f**), bastando remover a chave 58 do nó **f**
- O nó **f** ficou pequeno demais e precisa ser reunido

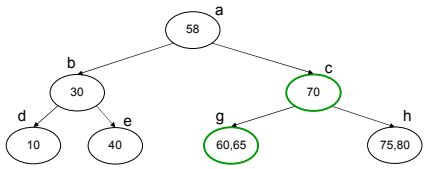





# Remoção X = 55

| R | W |
|---|---|
| 4 | 3 |

- A chave 55 (nó **a**) é substituída pela chave 58 (nó **f**), bastando remover a chave 58 do nó **f**
- O nó **f** ficou pequeno demais e precisa ser reunido, totalizando 7 acessos a disco






# Algoritmo de Remoção

- No algoritmo de remoção, **DeleteB**, assume-se que a chave X deve ser removida da árvore T tipo B da ordem **m** que reside em disco



# Algoritmo de Remoção

```
procedure DeleteB(T, X)
 1 (p,i,achou) ← MSearch(T,X);
 2 if not achou then return; endif // X não está em T
 3 seja p da forma: n,A0,(K1,A1),...,(Kn,An) com Ki=X
 4 if A0 ≠ 0 then
 5   // remoção a partir de não-folha, encontre chave p/mover
 6   q ← Ai;  // ir para subárvore direita
 7   seja q da forma: nq,Aq0, (Kq1,Aq1), ... , (Kqnq,Aqnq)
 8   while Aq0 ≠ 0 do // q não é uma folha
 9     q ← Aq0;       // procurar uma folha
10   endwhile
11   substitua Ki por Kq1 no nó p e grave em disco o nó p alterado
12   p ← q; i ← 1;
13   seja n, A0,(K1,A1), ..., (Kn,An) definido pelo novo p
   endif
   // suprima Ki do nó p, uma folha
14 p=n,A0,(K1,A1),...,(Kn,An) remova (Ki,Ai) de p e substitua n por
   n-1;
```



# Algoritmo de Remoção

```
15 while n < ⌈m/2⌉-1 and p ≠ T do
16   if p tem um irmão direito y mais próximo then
17     seja y: ny,Ay0,(Ky1,Ay1), ..., (Kyny,Ayny);
18     seja z: nz,Az0,(Kz1,Az1), ..., (Kznz,Aznz) o pai de p e y
19     seja j tal que p=Azj-1 e y=Azj
20     if ny ≥ ⌈m/2⌉ then // y pode emprestar, redistribua as chaves
21       atualize p: (Kn+1,An+1)←(Kzj,Ay0); n ← n+1;
22       atualize z: Kzj ← Ky1;
23       atualize y: (ny,Ay0,(Ky1,Ay1),... ) ← (ny-1,Ay1,(Ky2,Ay2),...);
24       Escreva nós p,z,y para disco
25       return;
       endif
       // combinar p, Kzj e y
26     r ← 2*⌈m/2⌉ - 2;
27     Escreva r,A0,(K1,A1),...,(Kn,An),(Kzj,Ay0),(Ky1,Ay1),...,(Kyny,Ayny)
         como novo nó p para disco
28     (nz,Az0,...)←(nz-1,Az0,...,(Kzj-1,Azj-1),(Kzj+1,Azj+1), ...);
29     p ← z; // continuar processo no nó pai
     else
30     // p deve ter um irmão esquerdo, o que é simétrico às linhas 17-29
     endif
31 endwhile
```



# Algoritmo de Remoção

```
32   if n ≠ 0 then
33     Escreva nó p:n,A0,(K1,A1)...,(Kn,An) para disco;
34   else
35     T ← A0;  // nova raiz
36   endif
end DeleteB
```

# Algoritmo de Remoção

- Nas linhas 4-13 é feito o tratamento para o caso de uma remoção em um nó não folha, procurando um elemento para trocar e proceder a remoção em uma folha
- Nas linhas 21-25 são efetuadas as trocas de chaves (“empréstimo” de chaves)
- Nas linhas 26-28 é efetuada a reunião de chaves
- As linhas 32-36 somente são executadas se o processo chegou até a raiz



# Análise DeleteB

- A busca de X (linha 1), junto com a pesquisa da folha Q nas linhas 5-10 precisam de **h** acessos, sendo **h** o número de níveis em T
- No caso que **p** não é um nó folha (linha 3) então ele é modificado e gravado conforme a linha 11
- Assim, a quantidade máxima de acessos nas linhas 1-13 é **h**+1
- Começando de um nó folha, cada iteração do laço **while** das linhas 15-31, move **p** um nível para cima na árvore
  - Por isso, podem ser feitas no máximo **h**-1 iterações desse laço
  - Admitindo-se, que todos os nós desde a raiz da T até a folha **p**, encontram-se na memória principal, os únicos acessos adicionais necessários são para os nós irmãos e para regravar os nós que foram alterados
  - O pior caso acontece quando são feitas **h**-1 iterações e a última resulta em terminação na linha 25 (ou local correspondente da linha 30)
  - Portanto, a quantidade máxima de acessos para esse laço é (**h**-1) para os irmãos, mais (**h**-2) atualizações na linha 27 (ou o local correspondente na linha 30), mais as 3 atualizações da linha 24 (ou 30) para término, totalizando, (**h**-1)+(**h**-2)+3 = 2***h**
- Portanto, a quantidade máxima de acessos necessários para uma remoção é de (**h**+1)+(2***h**) = 3***h**+1



# Análise DeleteB

- O tempo da remoção pode ser reduzido em troca de dispêndio de espaço no disco e um ligeiro aumento do tamanho do nó, incluindo um bit de remoção $F_i$, para cada valor de chave $K_i$ em um nó
- Com isso, podemos posicionar $F_i$=1 se $K_i$ não foi suprimido e $F_i$=0 quando foi
- Portanto, não ocorre uma remoção física e este tipo de remoção exige no máximo **h**+1 acessos (**h** para localizar o nó contendo X e 1 para gravar esse nó, com bit de remoção posicionado em 0)
- Com esta estratégia, nunca decresce a quantidade de nós da árvore, porém o espaço usado pelas entradas removidas pode ser usado novamente, durante novas inserções
- Como resultado, a estratégia teria pouca influência sobre os tempos de busca e inserção (quando **m** é grande, o número de níveis aumenta muito devagar com o aumento do número de chaves)
- O tempo de inserção pode até mesmo diminuir ligeiramente, por causa da habilidade para reusar o espaço de chaves removidas, o que pode evitar a necessidade de fracionar nós



# Variações de Árvores B

- Árvores B virtuais
  - Parte da árvore é mantida em memória principal
- Árvores B*
  - Há uma exigência que 2/3 de cada nó esteja preenchido, diferente de uma árvore B que exige apenas 1/2 de cada nó ocupado
- Árvores B+
  - Mantém todas as chaves (registros) nas folhas



# Árvores B Virtuais

- Apesar de Árvores B serem muito eficientes, seu desempenho pode ainda ser melhorado
- O fato de não ser possível manter todo o índice na memória principal não significa que não se possa manter pelo menos parte dele



# Árvores B Virtuais

- Assuma um índice que:
  - ocupa 1 Mb
  - existem disponíveis apenas 256 Kb de RAM
  - um nó (página) ocupa 4 Kb com m=64 chaves por nó
- Este índice pode ser armazenado em uma árvore B totalmente contida em 3 níveis
- Dessa forma, é possível chegar a qualquer nó com, no máximo, 3 acessos a disco
- Entretanto, se a raiz for mantida todo o tempo em memória principal, ainda sobraria muito espaço em RAM
- Com essa solução, número de acessos no pior caso diminui em uma unidade

## Árvores B Virtuais

- É possível generalizar esta idéia e ocupar toda a memória disponível com quantos nós forem possíveis
- Assim, quando um nó for necessário ele pode já estar em memória principal
- Se não estiver, ele é carregado para a memória, substituindo uma página que estava em memória
- Tem-se um RAM *buffer* também conhecido como **árvore B virtual**



## Árvores B Virtuais

- Política de gerenciamento de substituição de nós (LRU)
  - Se o nó não estiver em RAM, e esta estiver cheia, um nó em RAM precisa ser substituído
  - LRU (*Last Recently Used*)
    - substitui-se o nó que foi acessado menos recentemente
- O processo de acessar o disco para trazer um nó que não está no RAM *buffer* é denominado *page fault*



## Árvores B Virtuais

- Uma opção seria colocar todos os níveis mais altos da árvore em RAM
- No exemplo de 256 Kb de RAM e nós de 4Kb, podemos manter 64 nós (páginas) em memória
- Isso comporta a raiz e cerca de 8-10 nós do segundo nível, restando ainda espaço (utiliza-se LRU) sendo que o número de acessos diminui em mais uma unidade
- Na verdade, a criação de *buffers* é incluída em qualquer aplicação real de utilização de árvores B



## Árvores B Virtuais

- Com relação à informação associada às chaves, sou seja, os demais campos dos registros:
  - Se a informação for mantida junto com a chave, ganha-se um acesso a disco, mas perde-se no número de chaves que pode ser colocado em uma página; isso reduz a ordem da árvore, e aumenta a sua altura
  - Se ficar em um arquivo separado, a árvore é realmente usada como índice, e cada chave tem o ponteiro da posição do registro associado no arquivo principal de dados



## Árvores B*

- Uma árvore T do tipo B* e ordem **m** é uma árvore de busca em **m**-vias, que é ou vazia ou de altura **h** ≥ 1 e satisfaz as seguintes propriedades:
  - (a) a raiz tem pelo menos 2 filhos e no máximo $2*\lfloor(2*\mathbf{m}-2)/3\rfloor+1$
  - (b) todos os nós, exceto a raiz e nós de falha, têm pelo menos $2*\lceil(2*\mathbf{m}-2)/3\rceil$ e no máximo **m** filhos
  - (c) todos os nós de falha estão no mesmo nível
- $\lfloor \mathbf{x} \rfloor$ é o piso de **x**, ou seja, é o maior inteiro ≤ **x**
- $\lceil \mathbf{x} \rceil$ é o teto de **x**, ou seja, é o menor inteiro ≥ **x**



## Árvores B+

- Outra variação para a implementação de árvores B é conhecida como B+
- Em uma árvore B+ todas as chaves (registros) são armazenadas no último nível (nas folhas)
- Os níveis acima do último nível constituem um índice organizado como árvore B

## Acesso Seqüencial Indexado (ISAM)

- Em uma árvore B não é possível fazer acesso seqüencial eficiente
- Como arquivos devem suportar tanto acesso indexado eficiente como acesso seqüencial o conceito de árvores B foi estendido para árvores B+
- ISAM: *Indexed Sequential Access Method*
- ***Sequence set***: um conjunto de registros ordenados (segundo alguma chave), que podem ser recuperados com um acesso
  - Inviável manter todo o arquivo ordenado, mas é possível manter blocos



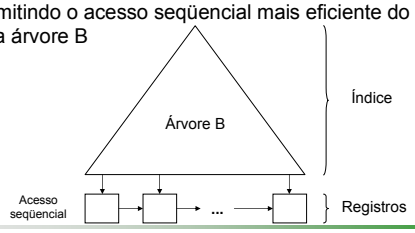
## Acesso Seqüencial Indexado (ISAM)

- Como visto, um bloco é a menor unidade básica de I/O
  - O tamanho do bloco é determinado pelo tamanho dos *buffers* de I/O
- Assim, um arquivo pode ser mantido como uma seqüência lógica de blocos, ou seja, um *sequence set*
  - O conteúdo de um bloco está ordenado, e pode ser recuperado em um único acesso
  - Cada bloco mantém um ponteiro para a posição do próximo bloco (na seqüência) no disco
  - Blocos logicamente adjacentes não estão (necessariamente) fisicamente adjacentes
  - Garantia de acesso seqüencial ao arquivo



## Árvores B+

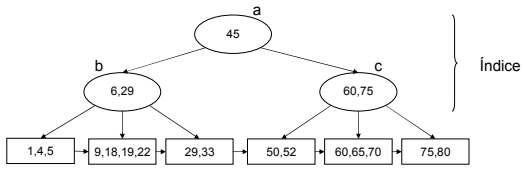
- No índice aparecem somente as chaves
- Nas folhas estão todas chaves (ou todos os registros) do arquivo
- As folhas são conectadas da esquerda para a direita, permitindo o acesso seqüencial mais eficiente do que em uma árvore B






## Árvores B+

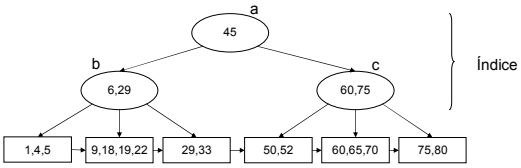
- Na busca por uma chave X, o processo tem início pela raiz e continua até uma folha
- Como todos os registros residem nas folhas, a pesquisa não pára se X for encontrado em um nó do índice
  - Neste caso, o filho à direita é seguido até que uma folha seja encontrada






## Árvores B+

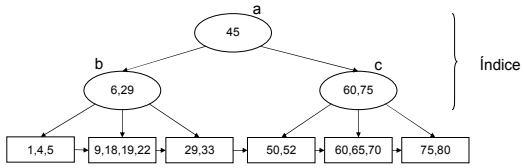
- Por exemplo, na busca das chaves 29, 60 e 75 (que aparecem tanto no índice como nas folhas) os valores encontrados no índice são usados apenas para conduzir até a folha correta





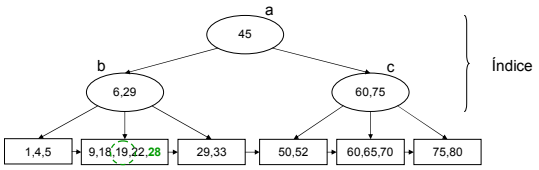

## Árvores B+

- A inserção em uma árvore B+ é similar àquela em uma árvore B
- A diferença reside em, quando uma folha é dividida em duas, o algoritmo efetua uma cópia da chave que pertence ao registro do meio para o nó pai no nível anterior, ficando o registro do meio na folha da direita



# Árvores B+

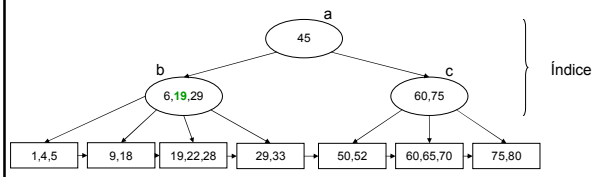
- No exemplo, ao inserir o registro 28 a folha fica grande demais
- Assim, a folha é dividida e registro 19 vai para a folha da direita
- A chave 19 é copiada para o nó pai






# Árvores B+

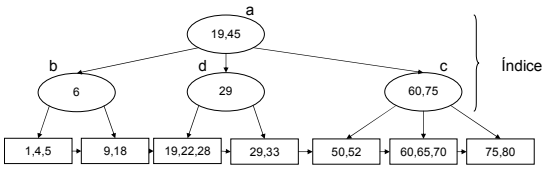
- O demais passos são similares à inserção em árvore B






# Árvores B+

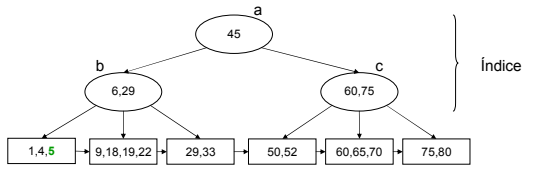
- O demais passos são similares à inserção em árvore B






# Árvores B+

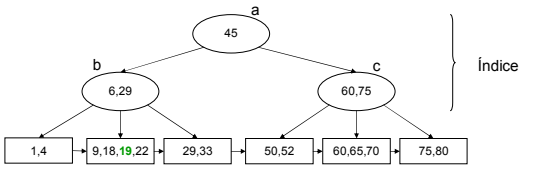
- A remoção é mais simples do que em uma árvore B
- O registro a ser retirado encontra-se sempre em uma folha, não havendo necessidade procurar um elemento para substituí-lo
- Desde que a folha fique pelo menos com a metade dos registros necessários, o índice não precisa ser modificado, mesmo se uma cópia da chave pertencendo ao registro sendo removido esteja no índice






# Árvores B+

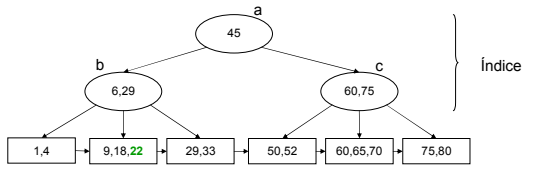
- No exemplo, removendo-se os registros 5





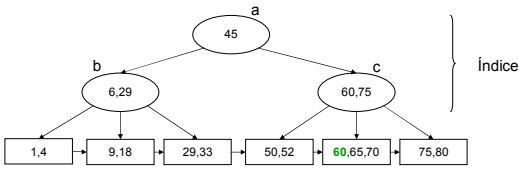

# Árvores B+

- No exemplo, removendo-se os registros 5, 19



# Árvores B+

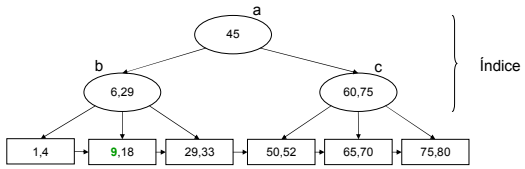
- No exemplo, removendo-se os registros 5, 19, 22






# Árvores B+

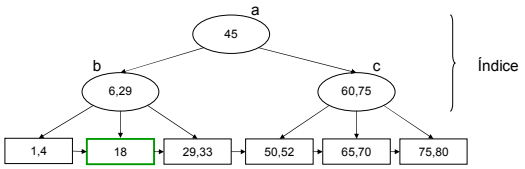
- No exemplo, removendo-se os registros 5, 19, 22 e 60 não há alteração no índice
- Removendo o registro 9...






# Árvores B+

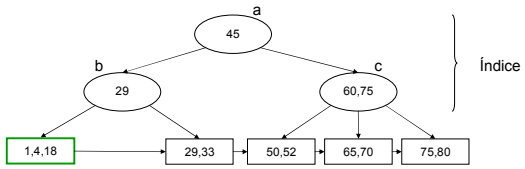
- No exemplo, removendo-se os registros 5, 19, 22 não há alteração no índice
- Removendo o registro 9 a folha fica com poucos registros e precisa ser agrupada com outra folha






# Árvores B+

- No exemplo, removendo-se os registros 5, 19, 22 não há alteração no índice
- Removendo o registro 9 a folha fica com poucos registros e precisa ser agrupada com outra folha






# Resumo

- Árvores B são simples, de fácil manutenção e eficientes
- O custo computacional para buscar, inserir e remover é O(**h**)=O($\log_{\lceil m/2 \rceil}$ **n**) onde **n** é o número de elementos na árvore e **h** é o número de níveis
- O espaço utilizado pelos dados é, no mínimo, 50% do espaço reservado
- As árvores B crescem e diminuem automaticamente e nunca requerem uma reorganização completa