

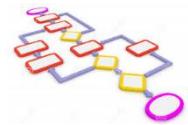


# Unidade 13 – Balanceamento de Árvores



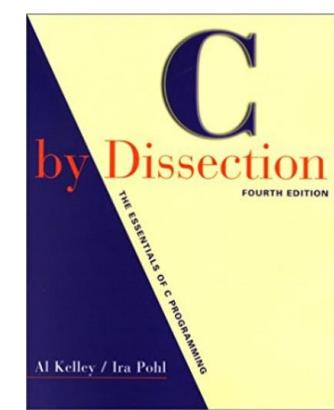
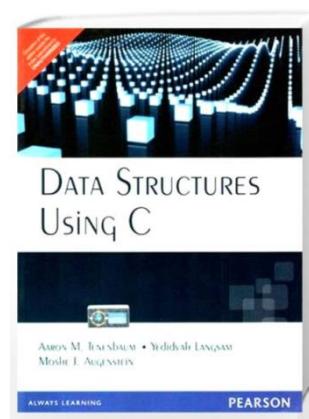
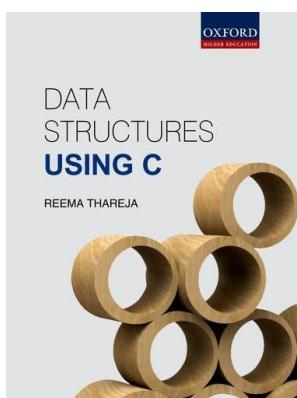
Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUSP  
[aparecidovfreitas@gmail.com](mailto:aparecidovfreitas@gmail.com)

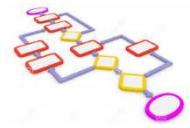




# Bibliografia

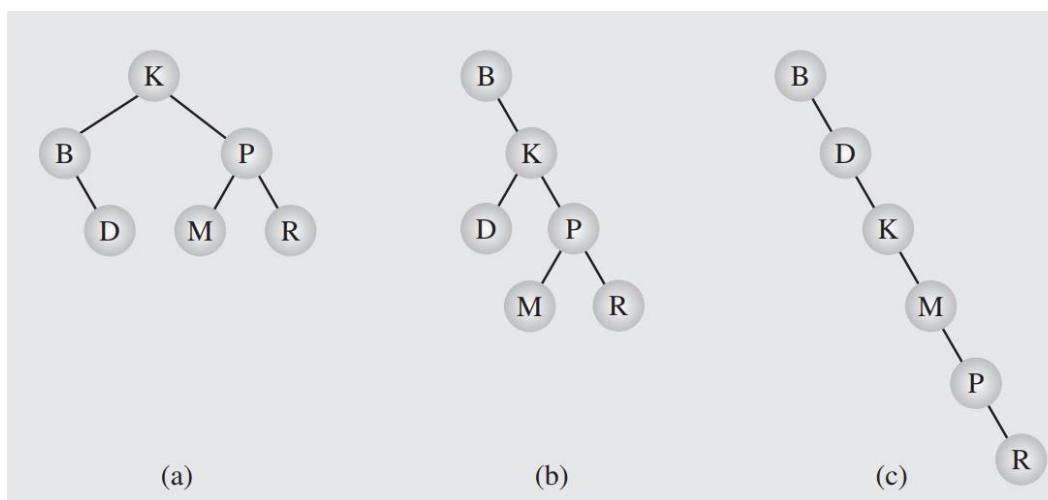
- Data Structures using C – Oxford University Press – 2014
- Data Structures Using C – A. Tenenbaum, M. Augensem, Y. Langsam, Pearson 1995
- C By Dissection – Kelley, Pohl – Third Edition – Addison Wesley





# Balanceando Árvores

- Árvores são estruturas muito apropriadas para representar objetos de forma hierárquica;
- Outra característica importante das árvores é que podem tornar o processo de busca muito rápido;
- No entanto, esta característica associada ao desempenho da busca depende do formato da árvore;
- Por exemplo, as árvores binárias de busca abaixo representam o mesmo conjunto de dados, porém qual delas representa o melhor desempenho em uma operação de busca?



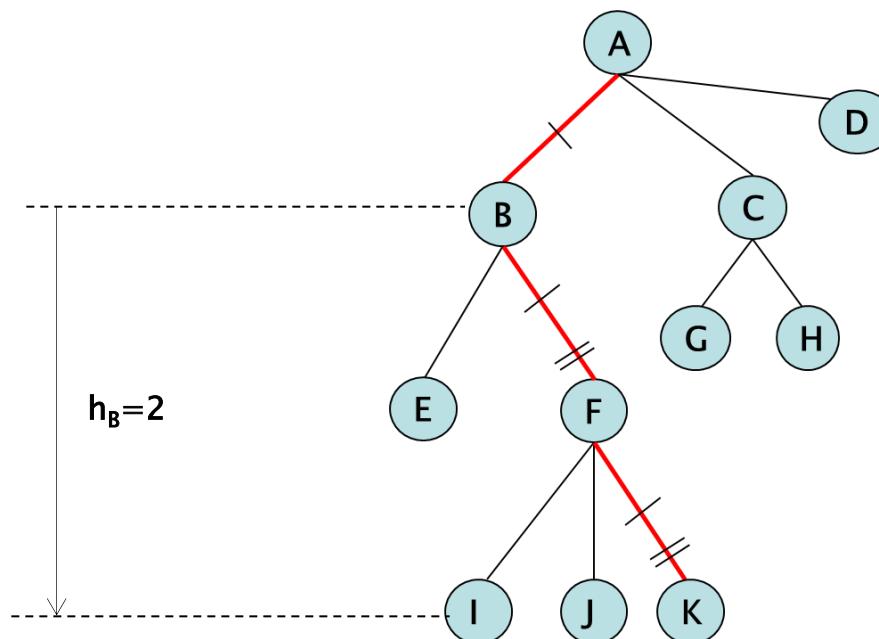
- A árvore da alternativa a) está balanceada e isso favorece o desempenho da busca.





# Árvore Balanceada

- Uma árvore binária em altura, ou simplesmente balanceada, quando a diferença na altura de ambas sub-árvores de qualquer nó é zero ou um;
- Lembrando que a altura de um nó **n** corresponde ao tamanho do caminho do nó **n** até o seu mais distante descendente. Por exemplo: A altura do nó **B** é 2.



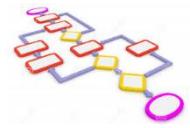


# Árvore Perfeitamente Balanceada

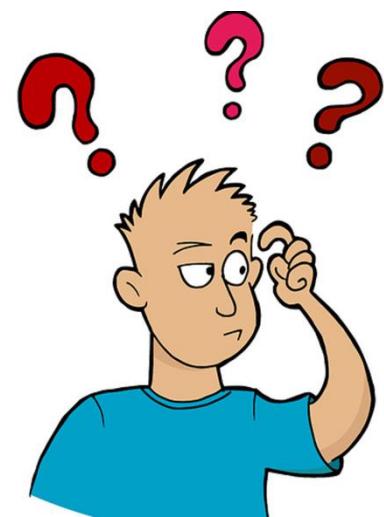
- Uma árvore binária é perfeitamente balanceada se ela é balanceada e todas as suas folhas estão no mesmo nível;

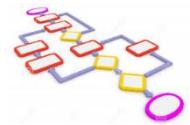
Nodes at One Level	Nodes at All Levels
$2^0 = 1$	$1 = 2^1 - 1$
$2^1 = 2$	$3 = 2^2 - 1$
$2^2 = 4$	$7 = 2^3 - 1$
$2^3 = 8$	$15 = 2^4 - 1$
$2^{10} = 1,024$	$2,047 = 2^{11} - 1$
$2^{13} = 8,192$	$16,383 = 2^{14} - 1$
$2^{h-1}$	$n = 2^h - 1$





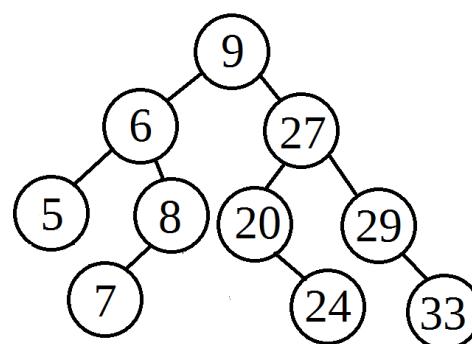
# Qual a vantagem de se criar árvores平衡adas?

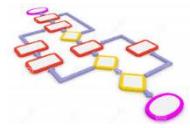




# Árvores Balanceadas

- Considere que uma árvore balanceada tem aproximadamente **10.000** nós;
- Então a árvore tem altura  $\log(10000) \approx 14$ ;
- Em termos práticos, isso significa que, se **10.000** elementos são armazenados em uma árvore balanceada, no máximo **14** nós terão de ser verificados para se localizar um elemento em particular;
- Essa é uma diferença substancial, comparada aos **10.000** testes necessários em uma lista ligada (no pior caso);
- Assim, é muito válido o esforço de se construir árvores平衡adas ou modificar-se uma árvore existente de modo que ela seja balanceada.





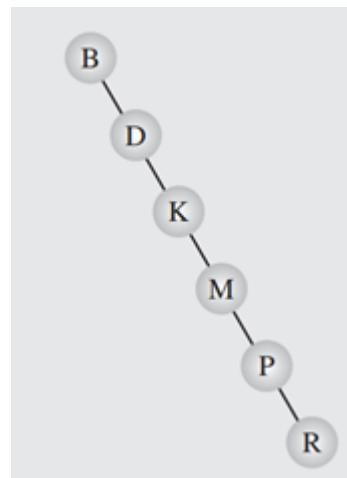
# Como balancear uma árvore binária





# Balanceamento de Árvores Binárias

- Há técnicas para se balancear apropriadamente uma árvore binária;
- Algumas baseiam-se em se reestruturar constantemente a árvore quando os elementos chegam;
- Outras consistem em reordenar os próprios dados e então construir a árvore;

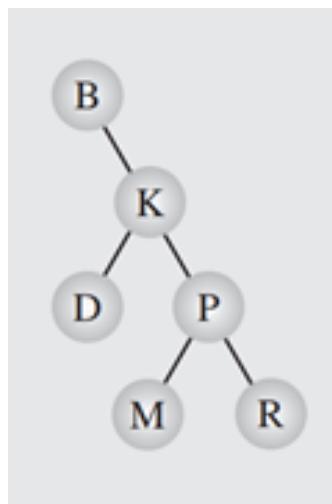


- Esta árvore é o resultado de uma corrente de dados particular;
- Quando os dados chegam em ordem ascendente ou descendente, a árvore se parece com uma lista ligada.



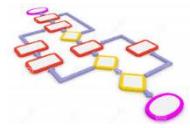


# Balanceamento de Árvores Binárias

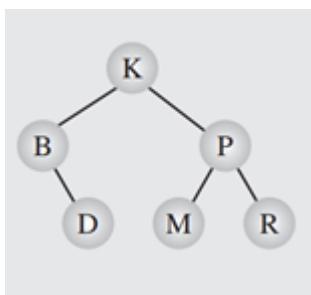


- Esta árvore é pendente lateralmente porque o primeiro elemento que chega é a letra B, que precede todas as outras, exceto A;
- Com isso, a sub-árvore da esquerda terá apenas um elemento.



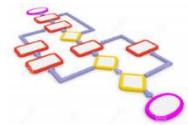


# Balanceamento de Árvores Binárias



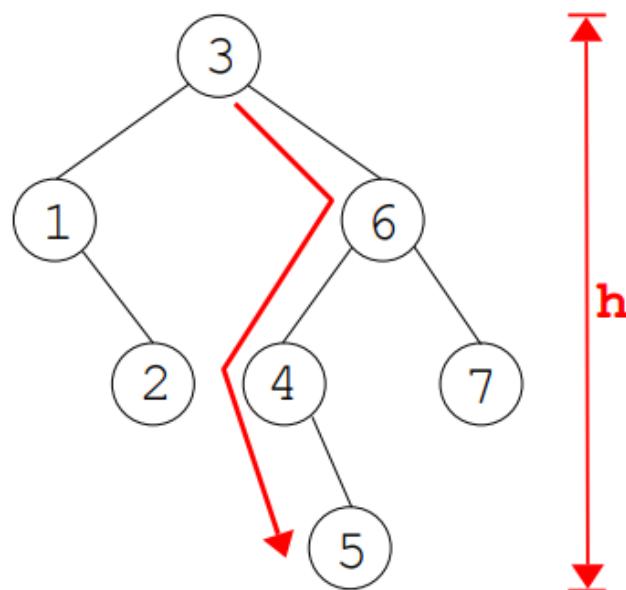
- Esta árvore se parece bem do ponto de vista de balanceamento;
- A raiz contém um elemento próximo do meio de todos os possíveis elementos e P está mais ou menos no meio de K e de Z;
- Isso nos leva a um algoritmo baseado na técnica binária de busca.





# Complexidade de Busca em Árvore Binária

- ▶ Busca em árvore binária = caminho da raiz até chave desejada (ou até uma folha, caso chave não exista).



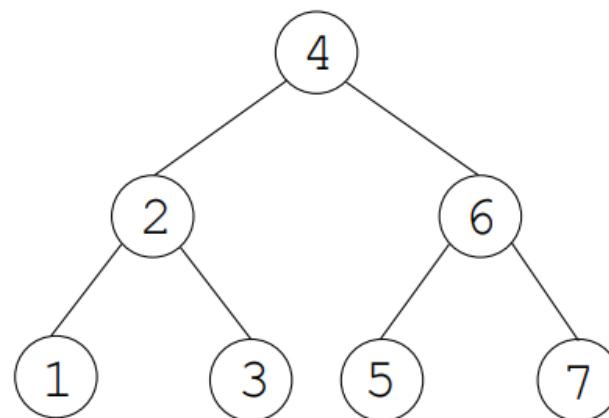
- ▶ Pior caso: maior caminho da raiz até folha = altura da árvore





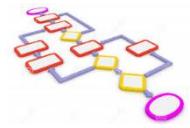
# Complexidade de Busca em Árvore Binária Ótima

- ▶ Árvore ótima: minimiza tempo de busca (no pior caso)
- ▶ Árvore completa, altura:  $h = \lfloor \log n \rfloor + 1$



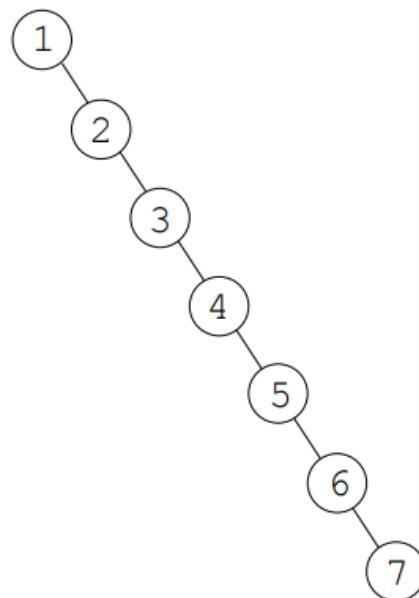
- ▶ complexidade temporal no pior caso:  $O(\log n)$





# Complexidade de Busca em Árvore Binária – Pior Caso

- ▶ após inserções, árvore binária de busca pode degenerar em uma lista



- ▶ tempo de busca pior caso:  $O(n)$



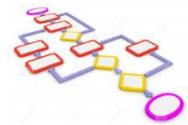


# Algoritmo de Balanceamento de Árvores Binárias – Técnica Binária de Busca

- Quando os dados chegarem, armazene-os em um array;
- Após o fluxo de dados, ordenar o array;
- Designe o para a raiz o elemento central do array;
- O array consiste agora de 2 sub-arrays: um entre o início do array e o elemento escolhido para raiz e outro entre a raiz e a extremidade do array;
- O algoritmo procede com a construção da árvore de forma recursiva.

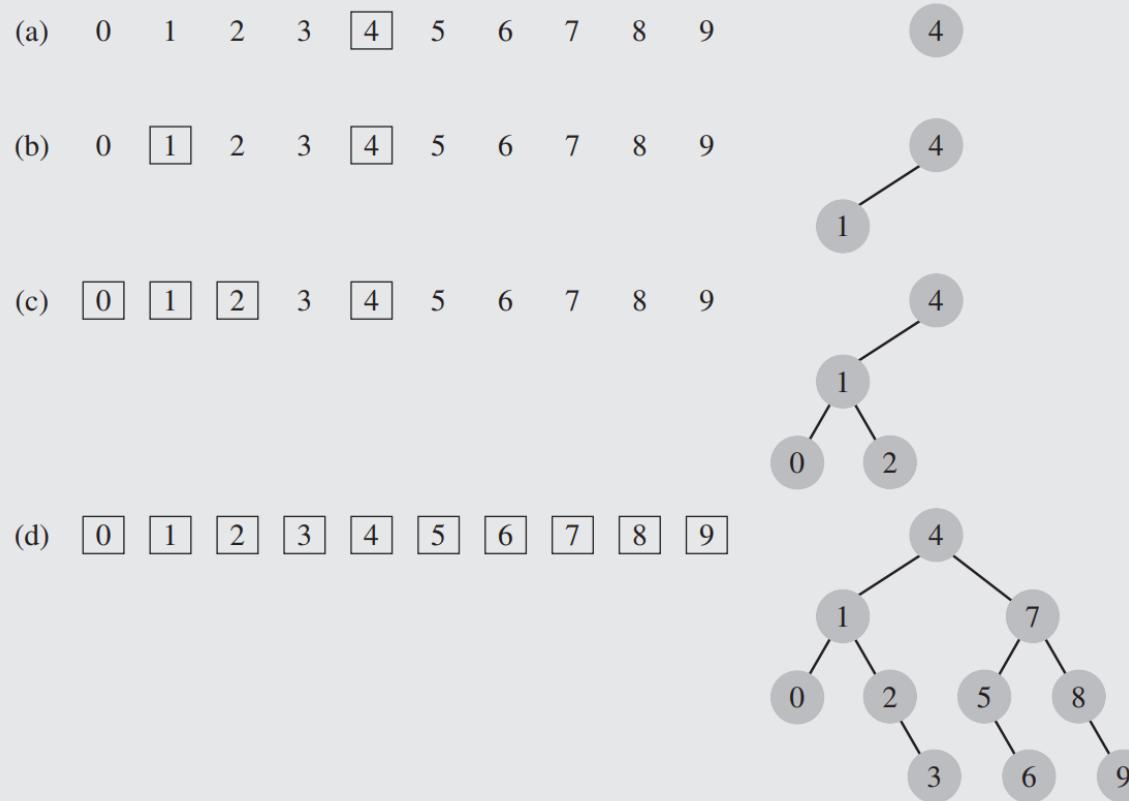
```
void balance(T data[], int first, int last) {  
  
    if (first <= last) {  
        int middle = (first + last) / 2;  
        insert(data[middle]);  
        balance(data, first, middle-1);  
        balance(data, middle+1, last);  
    }  
}
```

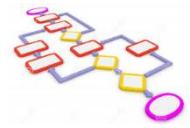




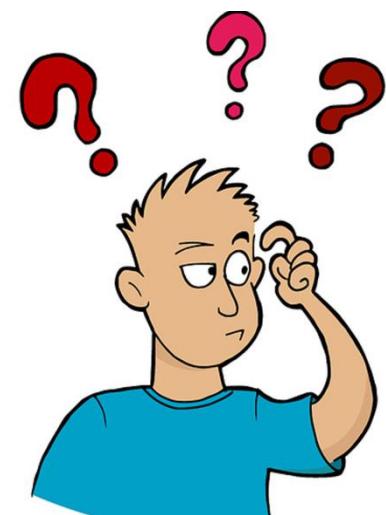
# Algoritmo de Balanceamento de Árvores Binárias – Técnica Binária de Busca

Stream of data: 5 1 9 8 7 0 2 3 4 6  
Array of sorted data: 0 1 2 3 4 5 6 7 8 9





# Há algum problema com esse algoritmo?



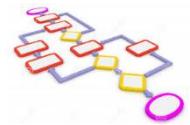


# Algoritmo de Balanceamento de Árvores Binárias – Técnica Binária de Busca

```
void balance(T data[], int first, int last) {  
    if (first <= last) {  
        int middle = (first + last) / 2;  
        insert(data[middle]);  
        balance(data, first, middle-1);  
        balance(data, middle+1, last);  
    }  
}
```

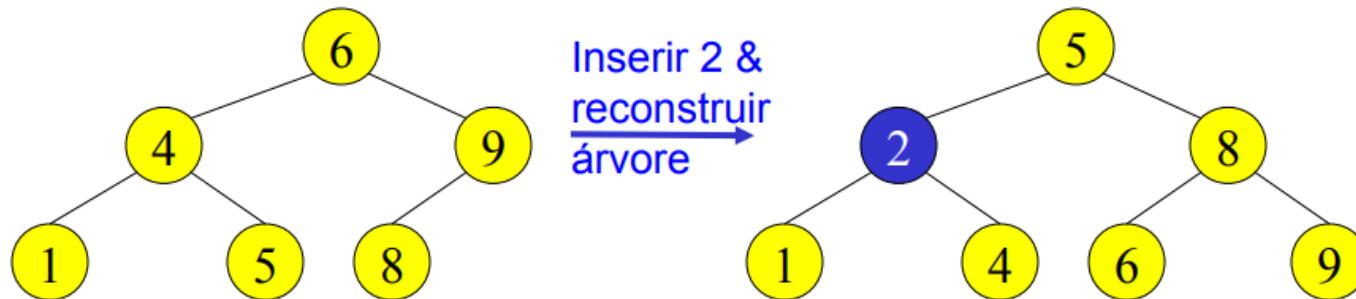
- Esse algoritmo tem um sério inconveniente;
- Todos os dados precisam ser colocados em um array antes da criação da árvore;
- Nesse caso, o algoritmo pode ser inadequado quando a árvore tem que ser usada enquanto os dados a serem incluídos na árvore ainda estiverem chegando.

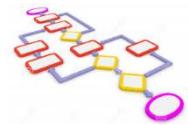




## Balanceamento in locus

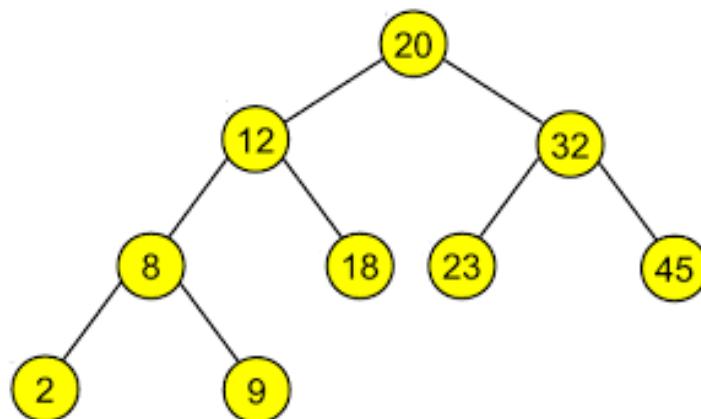
- Queremos uma **árvore completa** após cada operação
  - › a árvore está cheia exceto possivelmente na parte inferior direita.
- Isso é caro computacionalmente
  - › Por exemplo, inserir 2 na árvore da esquerda e então reconstruir toda a árvore.

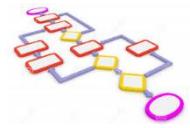




# Árvore AVL

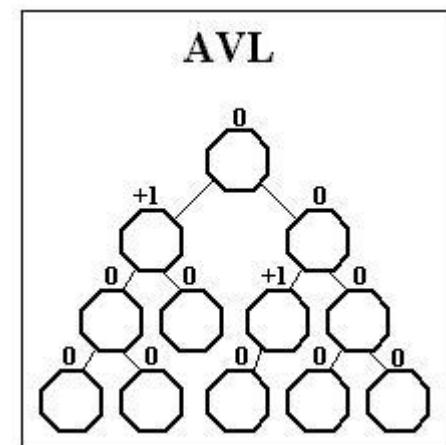
- ✓ É uma árvore de busca binária de altura autobalanceada ou de altura equilibrada;
- ✓ Em tal árvore, as **alturas** das suas sub-árvores a partir de cada nó diferem de no **máximo 1 unidade**;
- ✓ O nome **AVL** vem de seus criadores (Adelson Velsky e Landis)

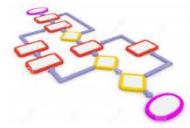




# Árvore AVL

- ✓ Em virtude do auto-balanceamento da árvore, as operações de busca, inserção e remoção em uma árvore com n elementos podem ser feitas, mesmo no pior caso, em **O(log n)**;
- ✓ Um teorema demonstrado por Adelson-Velskii e Landis, garante que a árvore balanceada nunca será **45%** mais alta que a correspondente árvore perfeitamente balanceada, independente do número de nós existentes;





# Árvore AVL

❑ Uma árvore AVL é definida como:

- Uma árvore vazia é uma árvore AVL
- Sendo **T** uma árvore binária de busca cujas subárvore esquerda e direita são **L** e **R**, respectivamente, **T** será uma árvore AVL contanto que:
  - ❖ **L** e **R** são árvores AVL
  - ❖  $|h_L - h_R| \leq 1$ , onde  $h_L$  e  $h_R$  são as alturas das subárvore **L** e **R**, respectivamente

❑ A definição de uma árvore binária de altura equilibrada (AVL) requer que cada subárvore seja também de altura equilibrada





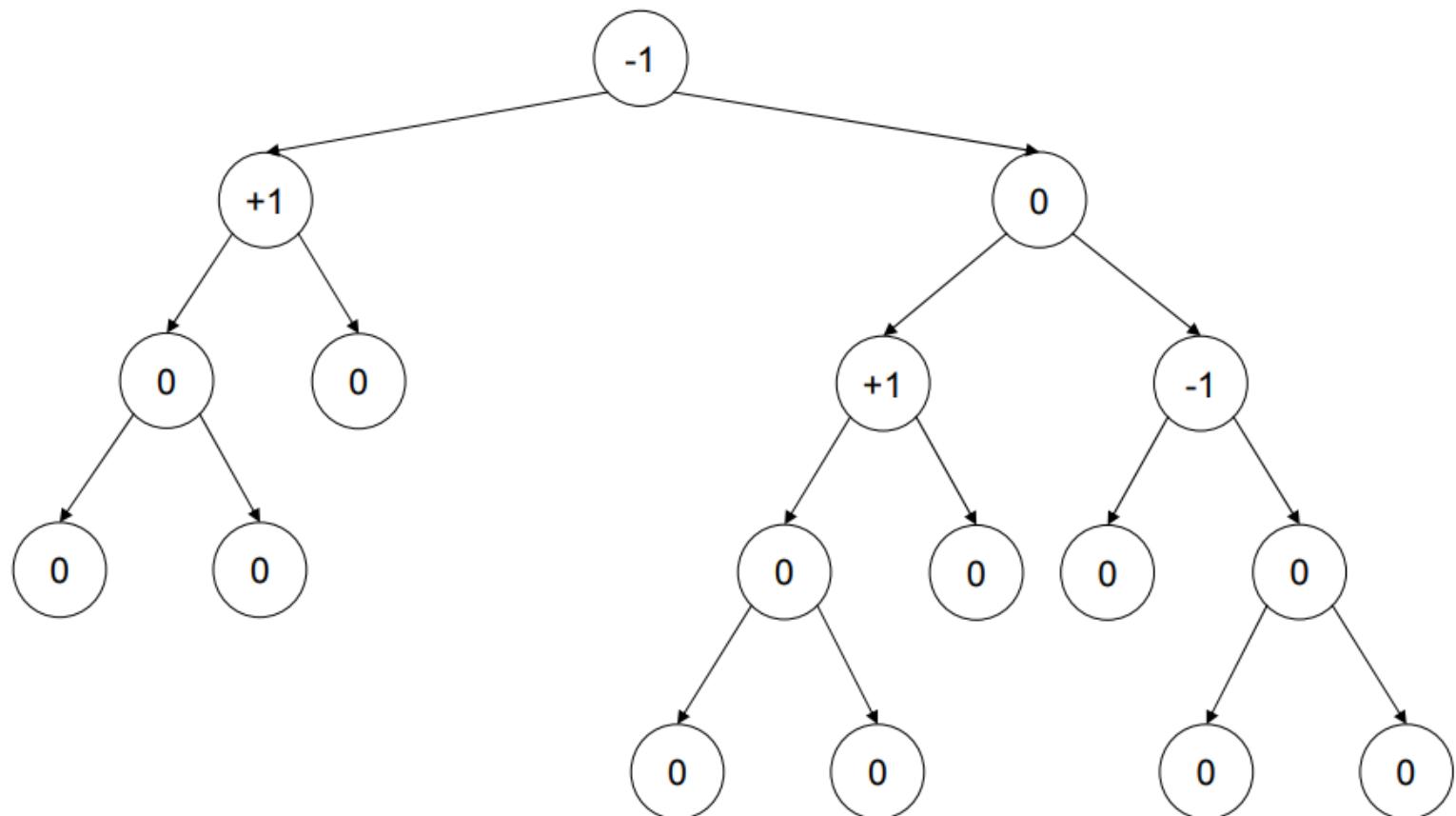
## Fator de Balanceamento

- ❑ O **fator de balanceamento** ou **fator de equilíbrio** de um nó  $T$  em uma árvore binária é definido como sendo  $h_L - h_R$  onde  $h_L$  e  $h_R$  são as alturas das subárvores esquerda e direita de  $T$ , respectivamente
- ❑ Para qualquer nó  $T$  numa árvore AVL, o fator de balanceamento assume o valor -1, 0 ou +1
  - O fator de balanceamento de uma folha é zero



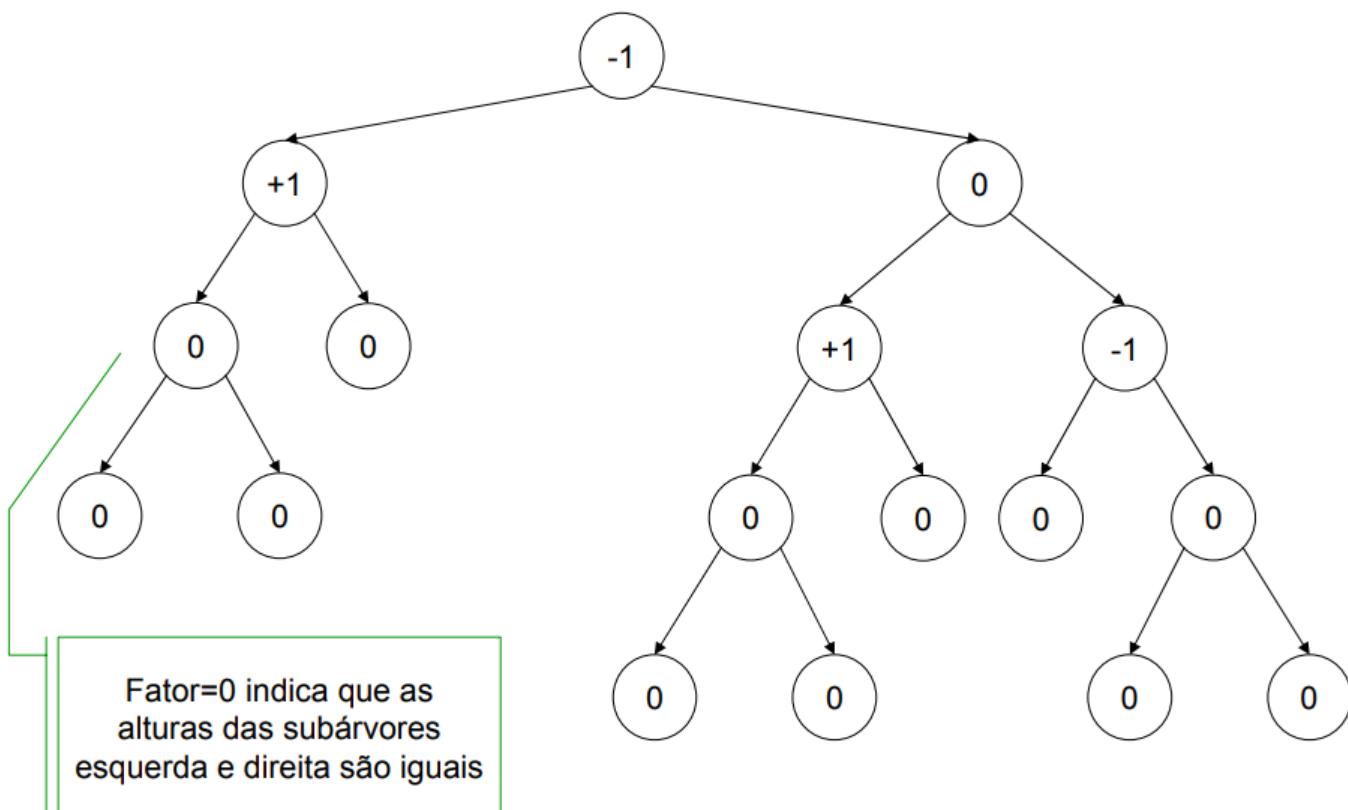


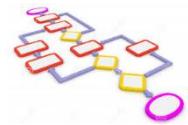
# Fator de Balanceamento



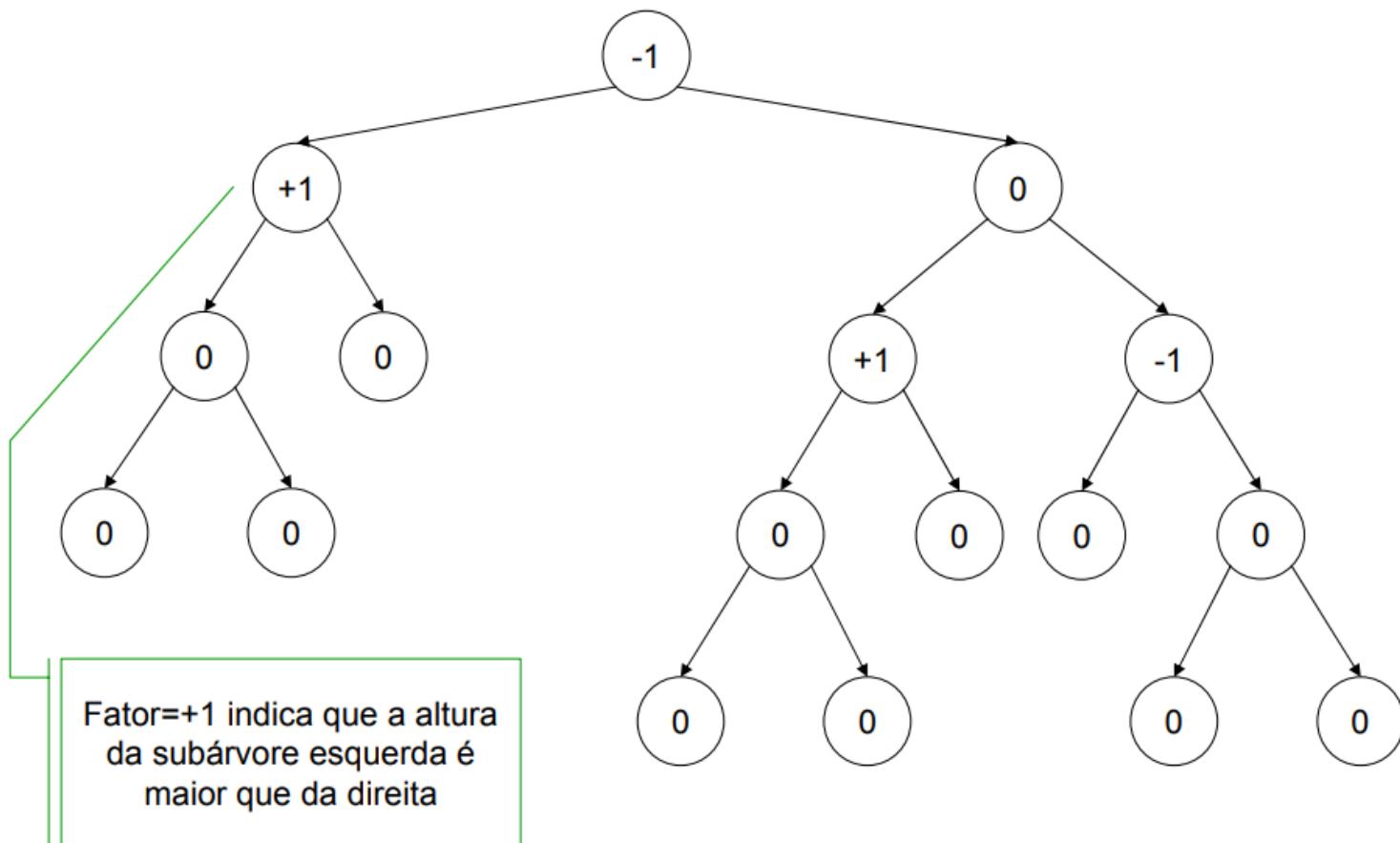


# Fator de Balanceamento



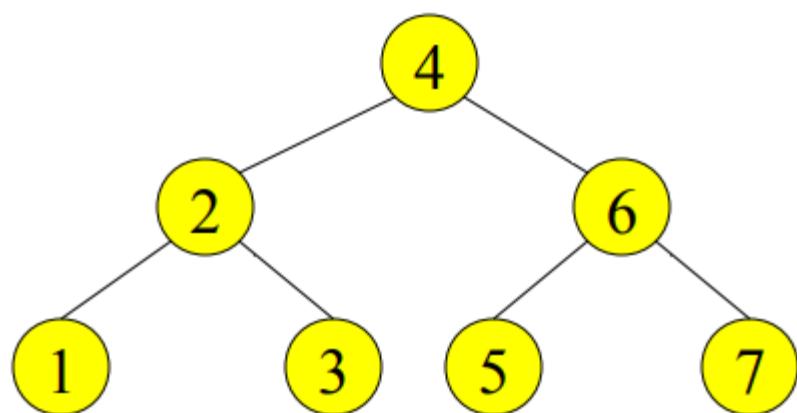


# Fator de Balanceamento



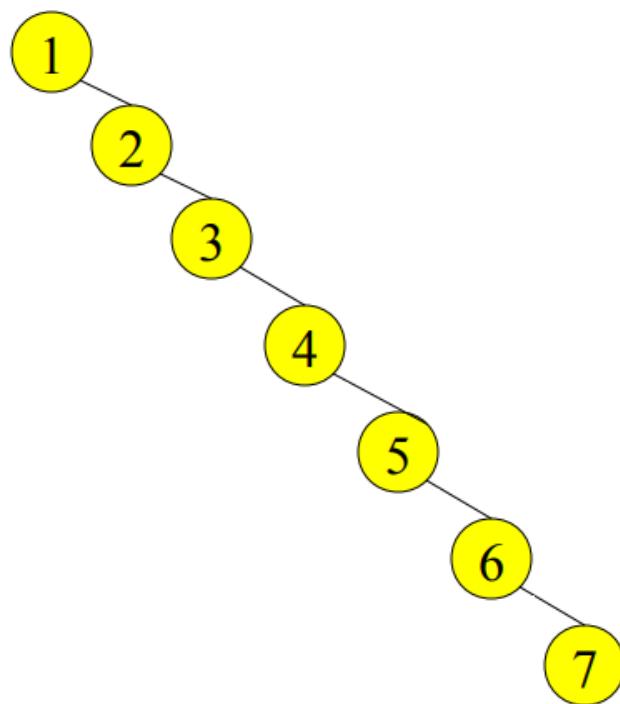


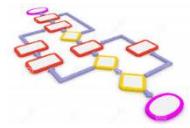
Esta árvore está balanceada?



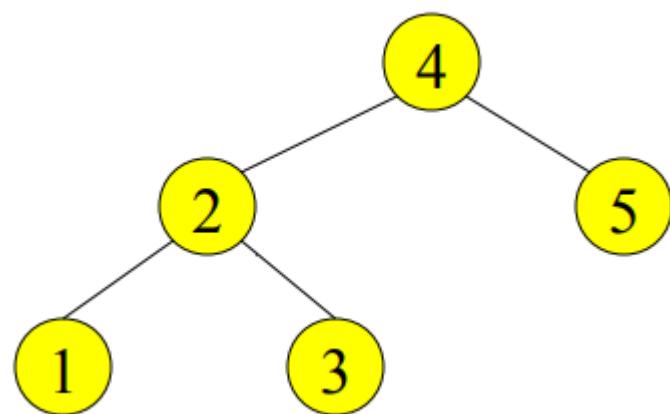


Esta árvore está balanceada?





Esta árvore está balanceada?





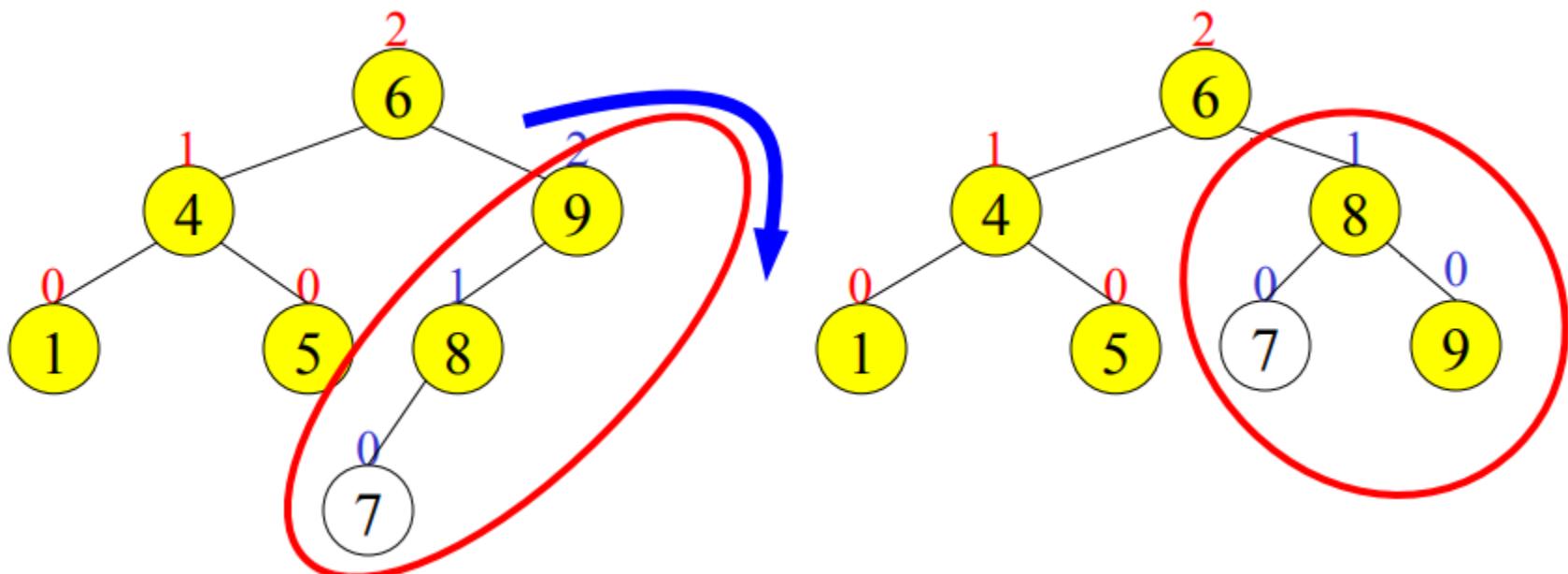
## Inserção e Rotação de nós em árvores AVL

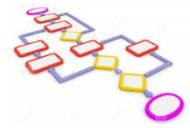
- A operação de inserção pode fazer com que o fator de balanceamento fique 2 ou -2 para algum nó.
  - › somente os nós no caminho do ponto de inserção até a raiz possivelmente mudaram a altura
  - › Após a inserção, deve-se voltar até a raiz, atualizando as alturas
  - › Se um novo fator de balanceamento é 2 or – 2, ajuste a árvore com *rotação* ao redor do nó





# Inserção e Rotação de nós em árvores AVL

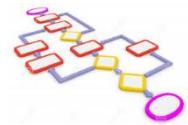




## Rotações em árvores AVL

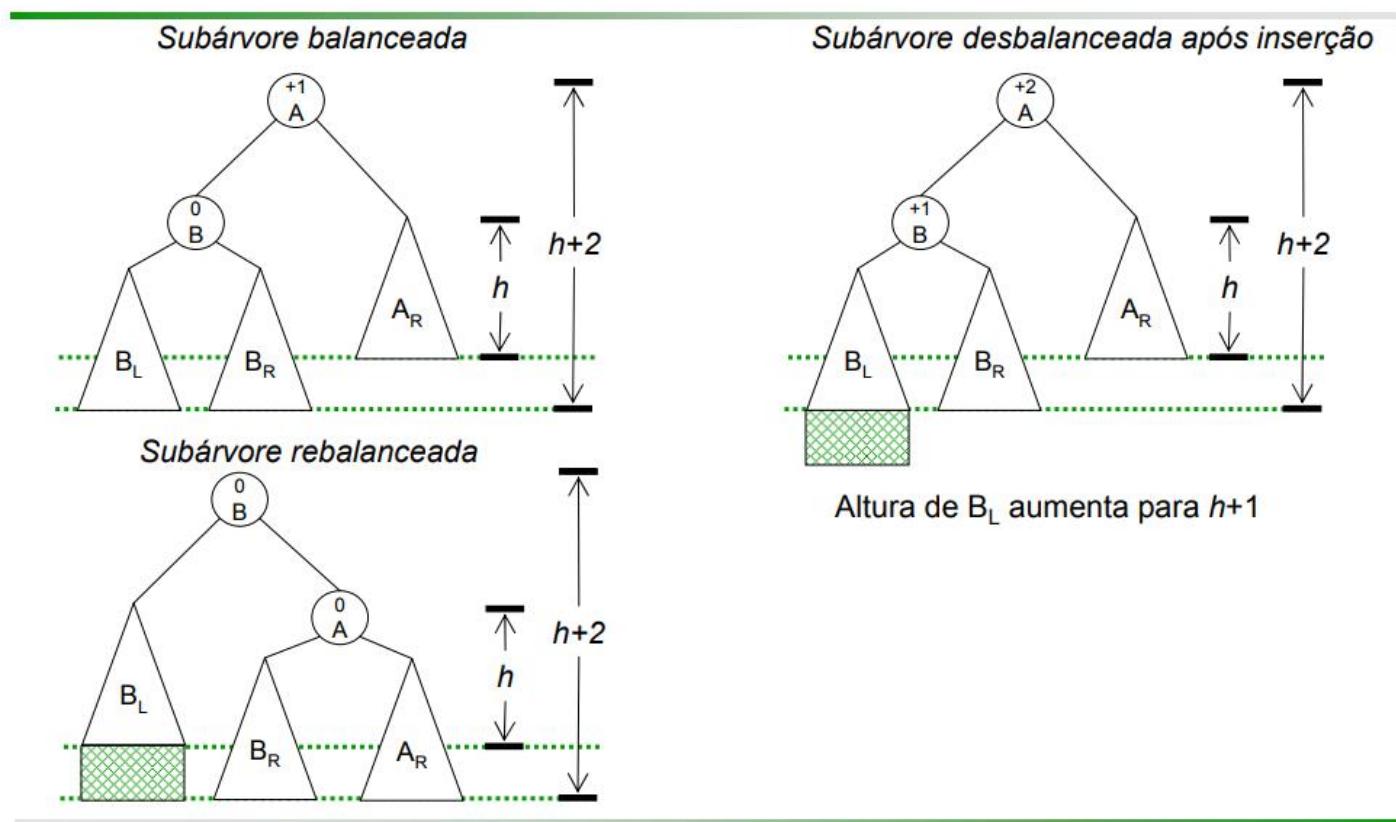
- ❑ O processo de rebalanceamento é conduzido utilizando 4 tipos de rotações: LL, RR, LR, RL
  - LL e RR são simétricas entre si assim como LR e RL
- ❑ As rotações são caracterizadas pelo ancestral mais próximo **A** do novo nó inserido **Y** cujo fator de balanceamento passa a ser +2 ou -2
  - LL: **Y** inserido na subárvore esquerda da subárvore esquerda de **A**
  - LR: **Y** inserido na subárvore direita da subárvore esquerda de **A**
  - RR: **Y** inserido na subárvore direita da subárvore direita de **A**
  - RL: **Y** inserido na subárvore esquerda da subárvore direita de **A**





## Rotação LL

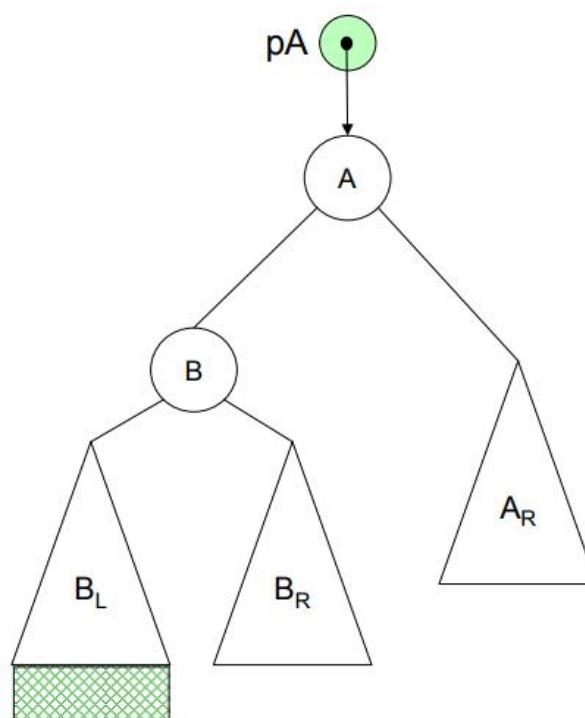
- LL: Y inserido na subárvore esquerda da subárvore esquerda de A





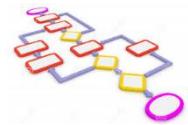
## Rotação LL

- LL: Y inserido na subárvore esquerda da subárvore esquerda de A
- 



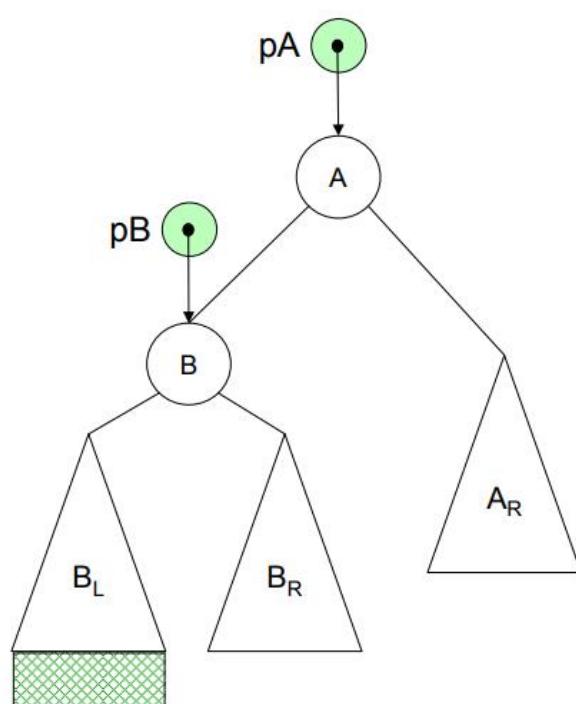
- Assumindo pA e pB ponteiros para as subárvore com raízes A e B:
  - $pB = pA->\text{LeftNode}$ ;
  - $pA->\text{LeftNode} = pB->\text{RightNode}$ ;
  - $pB->\text{RightNode} = pA$ ;
  - $pA = pB$ ;





## Rotação LL

- LL: Y inserido na subárvore esquerda da subárvore esquerda de A



- Assumindo pA e pB ponteiros para as subárvore com raízes A e B:
  - $pB = pA->LeftNode;$
  - $pA->LeftNode = pB->RightNode;$
  - $pB->RightNode = pA;$
  - $pA = pB;$

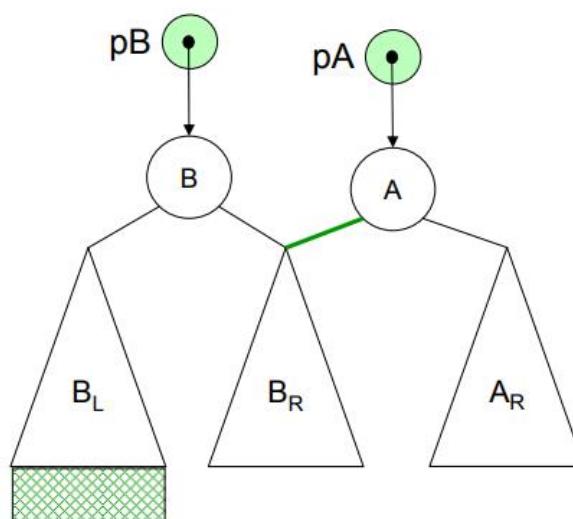




## Rotação LL

- LL: Y inserido na subárvore esquerda da subárvore esquerda de A

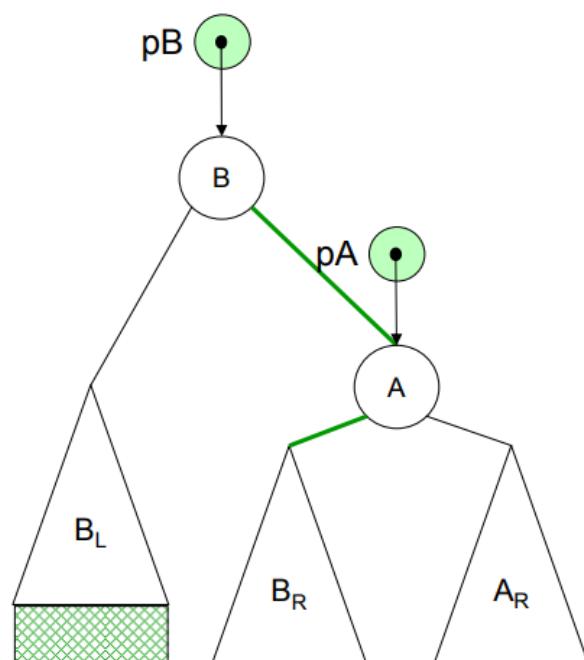
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - $pB = pA->LeftNode;$
  - $pA->LeftNode = pB->RightNode;$
  - $pB->RightNode = pA;$
  - $pA = pB;$





## Rotação LL

- LL: Y inserido na subárvore esquerda da subárvore esquerda de A



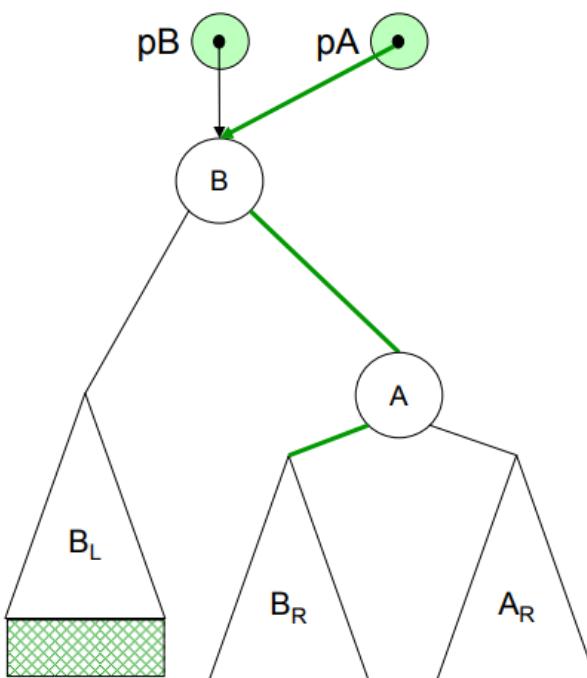
- Assumindo  $pA$  e  $pB$  ponteiros para as subárvores com raízes A e B:
  - $pB = pA->LeftNode;$
  - $pA->LeftNode = pB->RightNode;$
  - **$pB->RightNode = pA;$**
  - $pA = pB;$





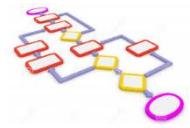
## Rotação LL

- LL: Y inserido na subárvore esquerda da subárvore esquerda de A



- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - $pB = pA->LeftNode;$
  - $pA->LeftNode = pB->RightNode;$
  - $pB->RightNode = pA;$
  - $pA = pB;$

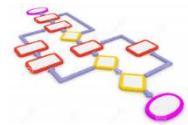




## Rotação LL

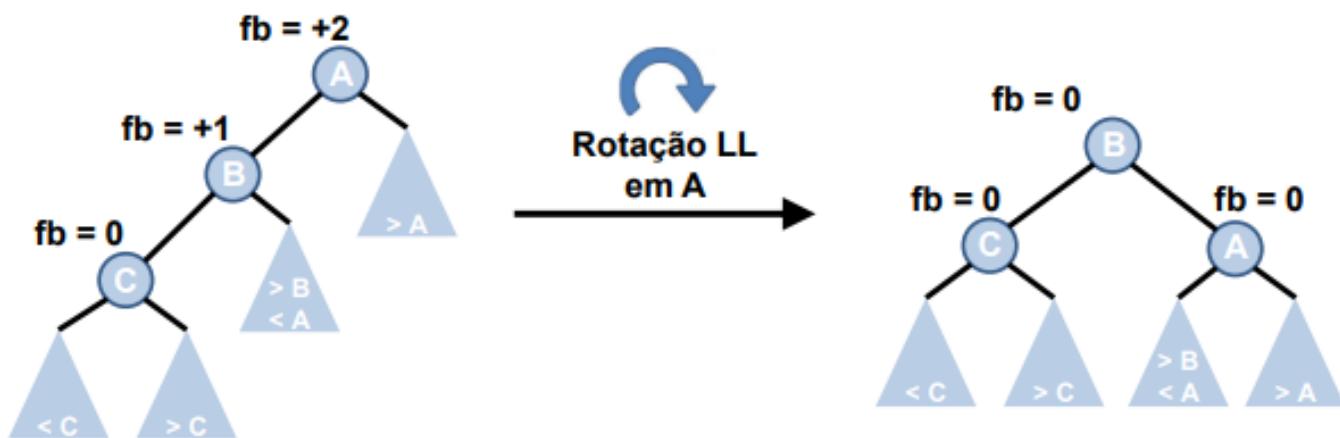
- Rotação LL ou rotação simples à direita
  - Um novo nó é inserido na **sub-árvore da esquerda do filho esquerdo** de A
  - A é o nó desbalanceado
  - É necessário fazer uma rotação à direita, de modo que o nó intermediário B ocupe o lugar de A, e A se torne a sub-árvore direita de B

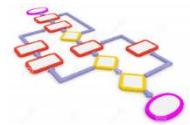




## Rotação LL

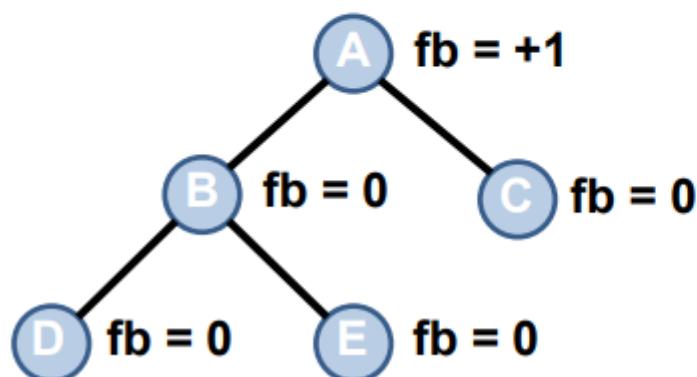
### □ Exemplo





## Rotação LL – Passo a Passo

### □ Passo a passo



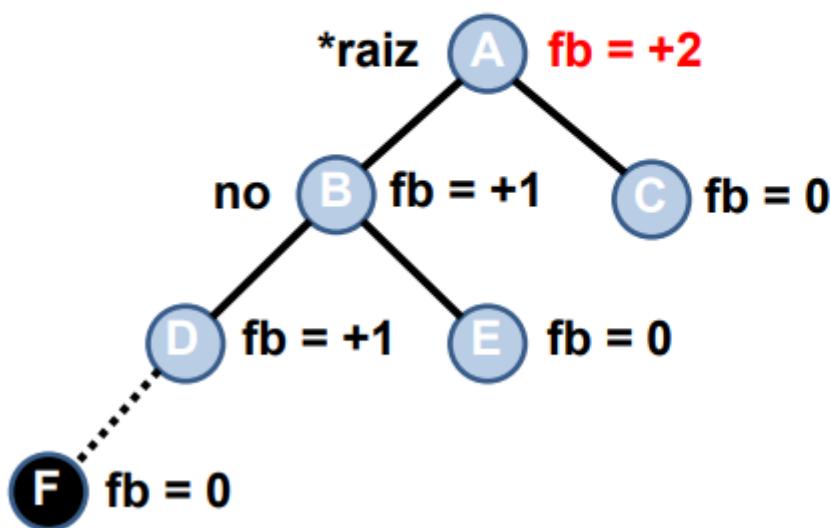
**Árvore AVL e fator de balanceamento de cada nó**





## Rotação LL – Passo a Passo

### □ Passo a passo



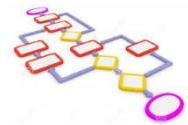
**Inserção do nó F na árvore**

**Árvore fica desbalanceada  
no nó A.**

**Aplicar Rotação LL no nó A**

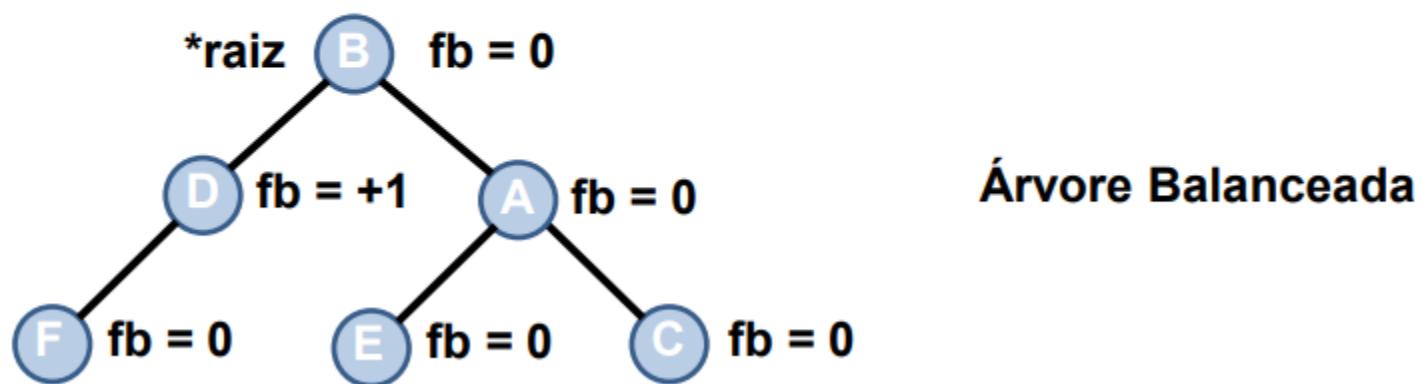
```
no = (*raiz)->esq;
(*raiz)->esq = no->dir;
no->dir = *raiz;
*raiz = no;
```

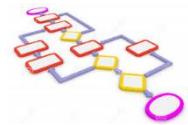




## Rotação LL – Passo a Passo

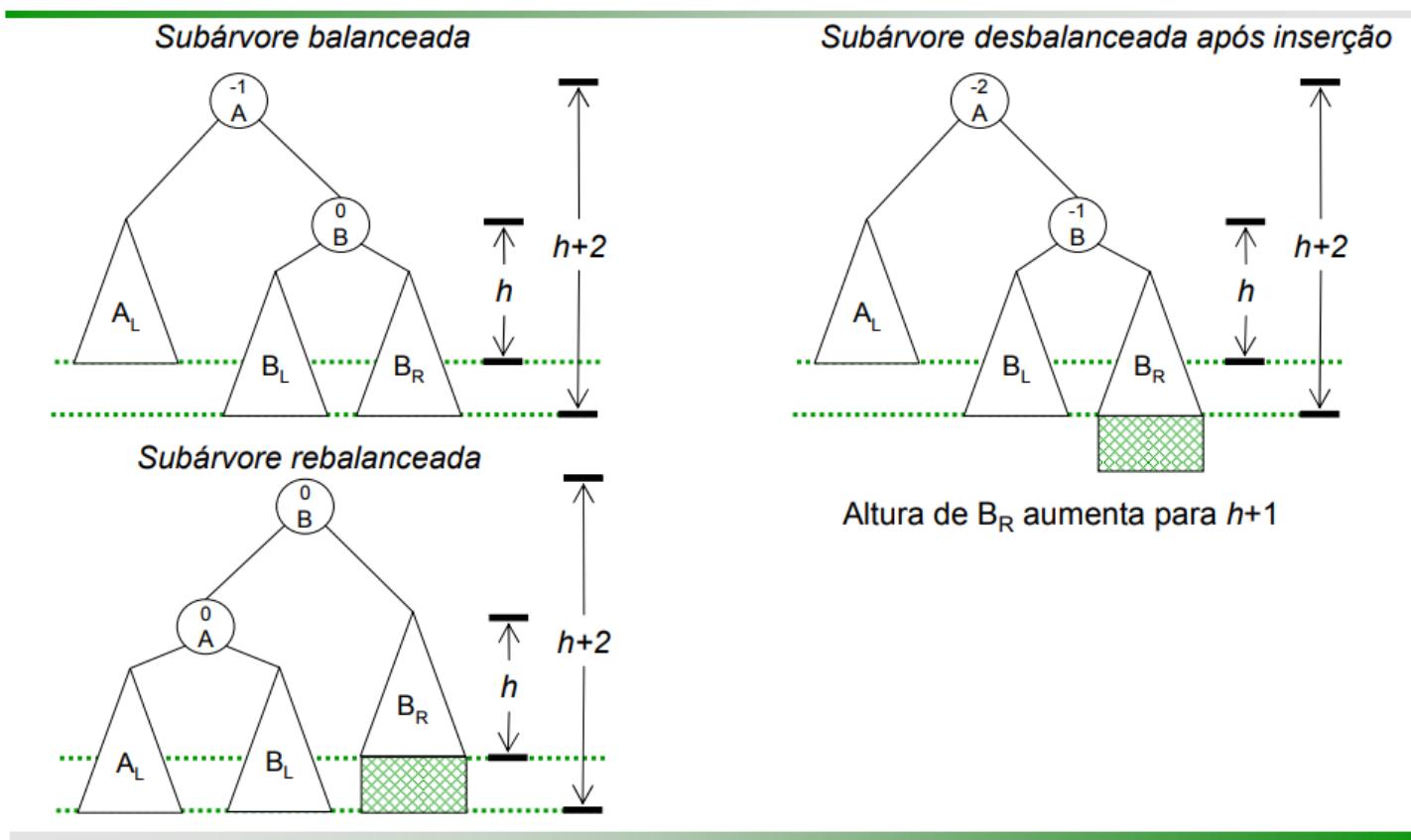
### □ Passo a passo





## Rotação RR

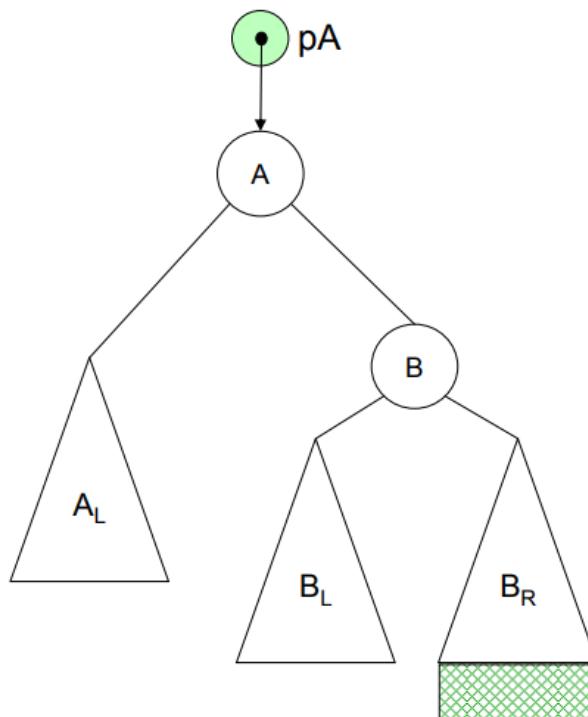
- RR: Y inserido na subárvore direita da subárvore direita de A





## Rotação RR

- RR: Y inserido na subárvore direita da subárvore direita de A
- 



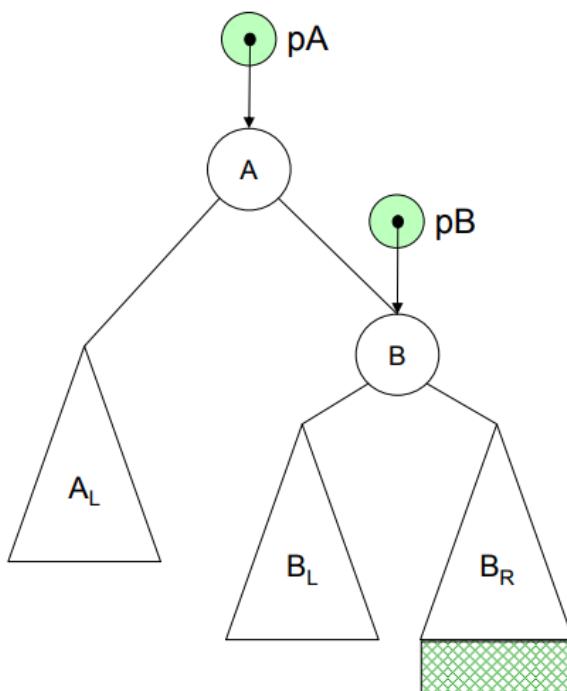
- Assumindo pA e pB ponteiros para as subárvore com raízes A e B:
  - $pB = pA->\text{RightNode};$
  - $pA->\text{RightNode} = pB->\text{LeftNode};$
  - $pB->\text{LeftNode} = pA;$
  - $pA = pB;$





## Rotação RR

- RR: Y inserido na subárvore direita da subárvore direita de A
- 



- Assumindo pA e pB ponteiros para as subárvore com raízes A e B:
  - $pB = pA->\text{RightNode};$
  - $pA->\text{RightNode} = pB->\text{LeftNode};$
  - $pB->\text{LeftNode} = pA;$
  - $pA = pB;$

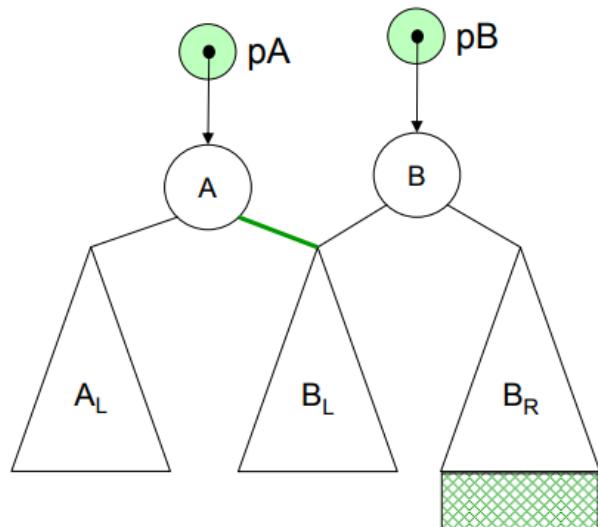




## Rotação RR

- RR: Y inserido na subárvore direita da subárvore direita de A
- 

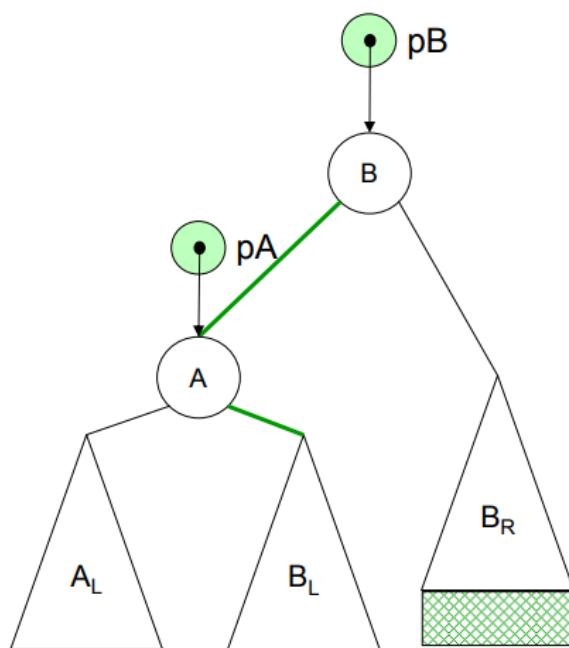
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - $pB = pA->RightNode;$
  - $\text{pA->RightNode} = \text{pB->LeftNode};$
  - $pB->LeftNode = pA;$
  - $pA = pB;$





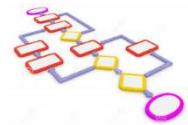
## Rotação RR

- RR: Y inserido na subárvore direita da subárvore direita de A



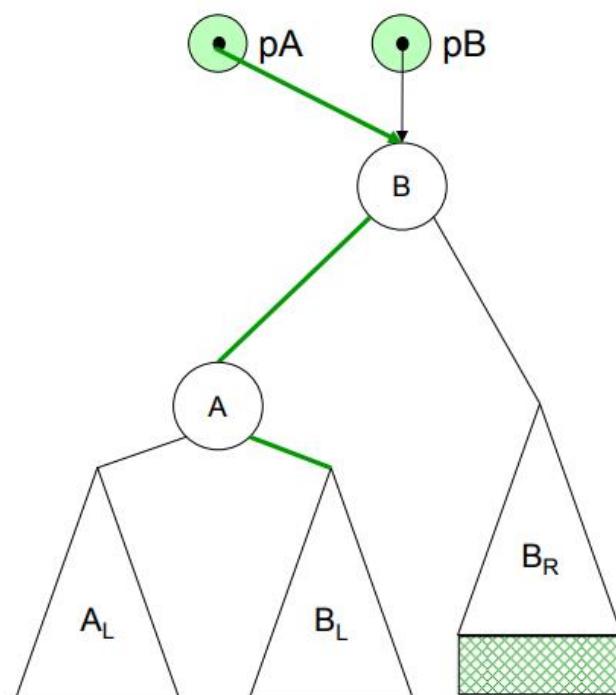
- Assumindo  $pA$  e  $pB$  ponteiros para as subárvore com raízes A e B:
  - $pB = pA->RightNode;$
  - $pA->RightNode = pB->LeftNode;$
  - $pB->LeftNode = pA;$
  - $pA = pB;$





## Rotação RR

- RR: Y inserido na subárvore direita da subárvore direita de A



- Assumindo pA e pB ponteiros para as subárvore com raízes A e B:
  - $pB = pA->RightNode;$
  - $pA->RightNode = pB->LeftNode;$
  - $pB->LeftNode = pA;$
  - $\textcolor{red}{pA = pB};$





## Rotação RR

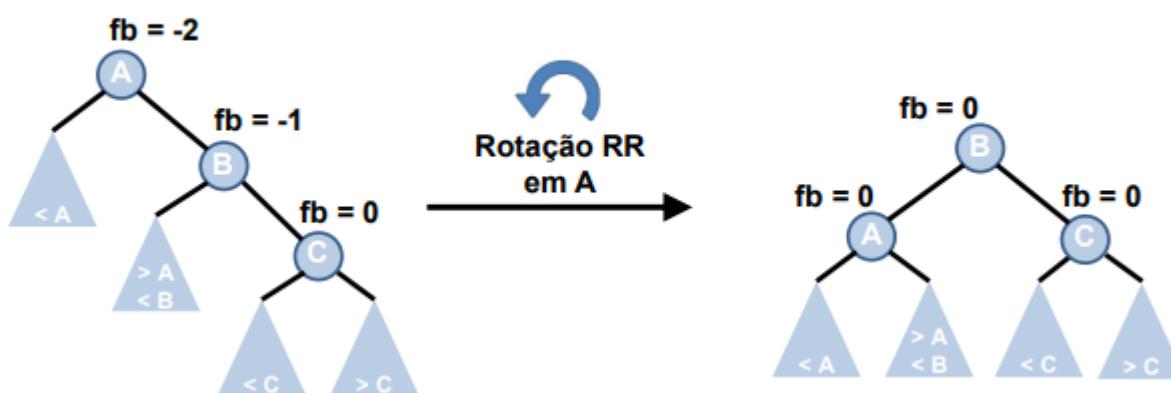
- Rotação RR ou rotação simples à esquerda
  - Um novo nó é inserido na **sub-árvore da direita do filho direito de A**
    - A é o nó desbalanceado
  - É necessário fazer uma rotação à esquerda, de modo que o nó intermediário **B** ocupe o lugar de **A**, e **A** se torne a sub-árvore esquerda de **B**

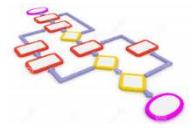




## Rotação RR

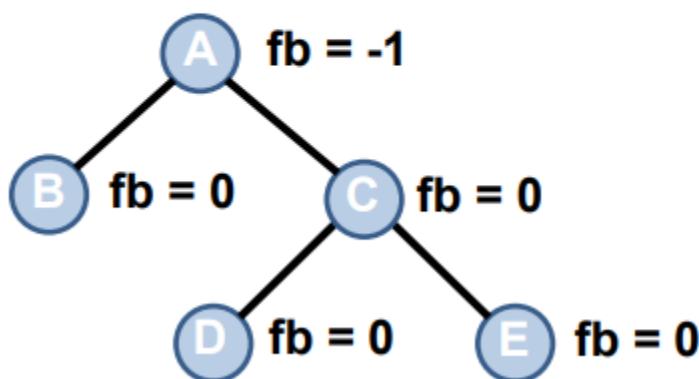
### □ Exemplo





## Rotação RR – Passo a Passo

### □ Passo a passo



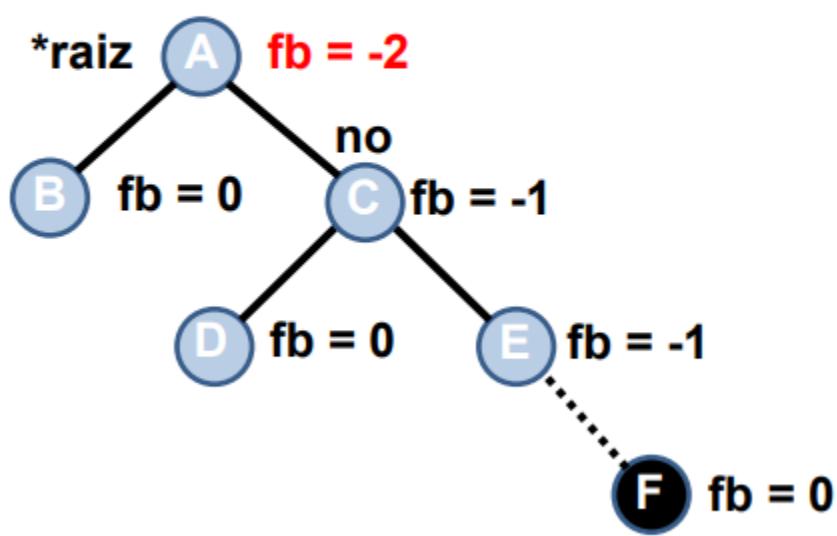
**Árvore AVL e fator de balanceamento de cada nó**





## Rotação RR – Passo a passo

### □ Passo a passo



Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação RR no nó A

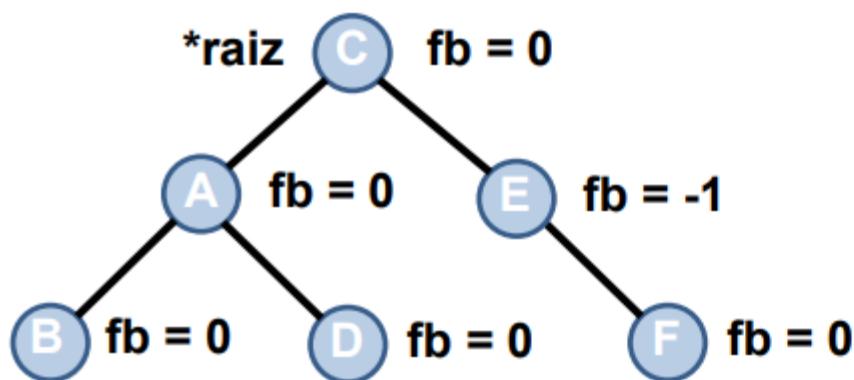
```
no = (*raiz)->dir;
(*raiz)->dir = no->esq;
no->esq = (*raiz);
(*raiz) = no;
```





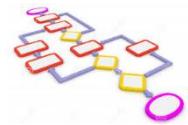
## Rotação RR – Passo a Passo

### □ Passo a passo



**Árvore Balanceada**

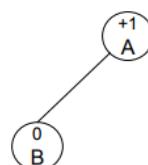




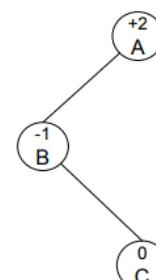
# Rotação LR

- LR: **Y** inserido na subárvore direita da subárvore esquerda de **A**

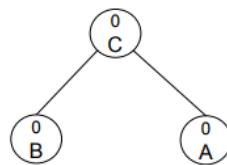
*Subárvore balanceada*

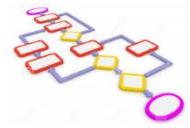


*Subárvore desbalanceada após inserção*



*Subárvore rebalanceada*





## Rotação LR

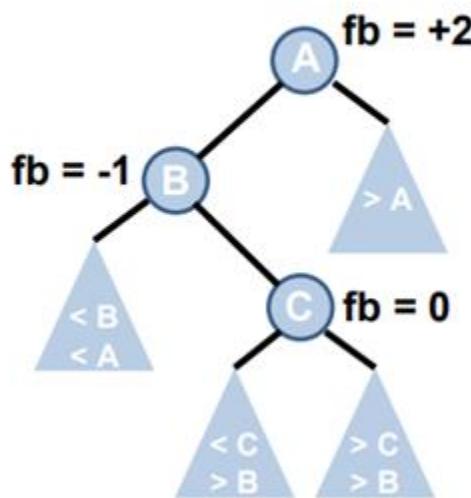
- Rotação LR ou rotação dupla à direita
  - Um novo nó é inserido na **sub-árvore da direita do filho esquerdo de A**
    - A é o nó desbalanceado
  - É necessário fazer uma rotação dupla, de modo que o nó **C** se torne o pai dos nós **A** (filho da direita) e **B** (filho da esquerda)
    - Rotação RR em **B**
    - Rotação LL em **A**



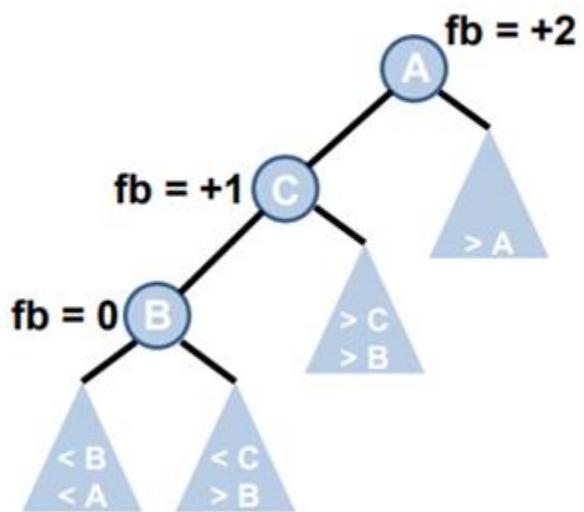


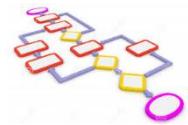
## Rotação LR

□ Exemplo: primeira rotação



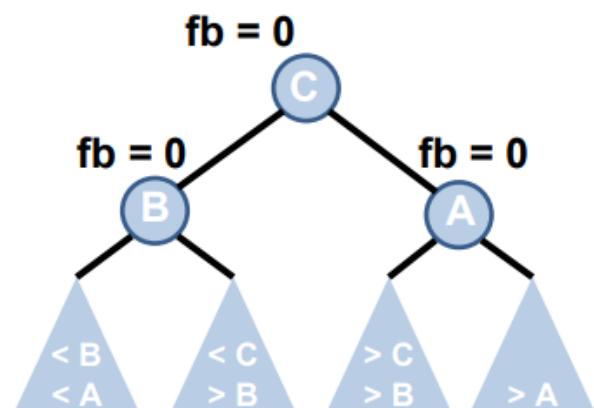
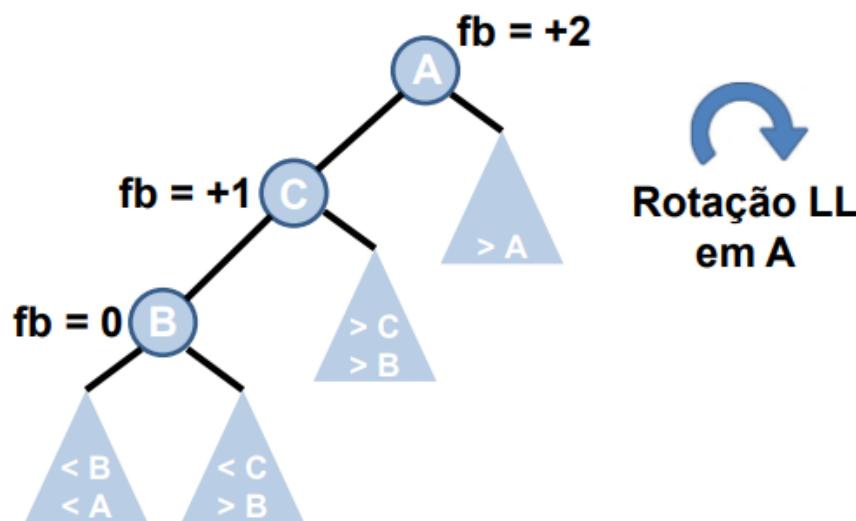
Rotação RR  
em B





## Rotação LR

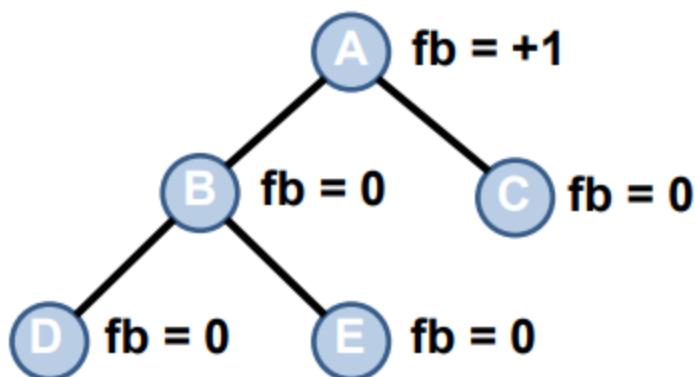
### □ Exemplo: segunda rotação





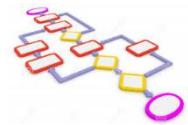
## Rotação LR – Passo a passo

### □ Passo a passo



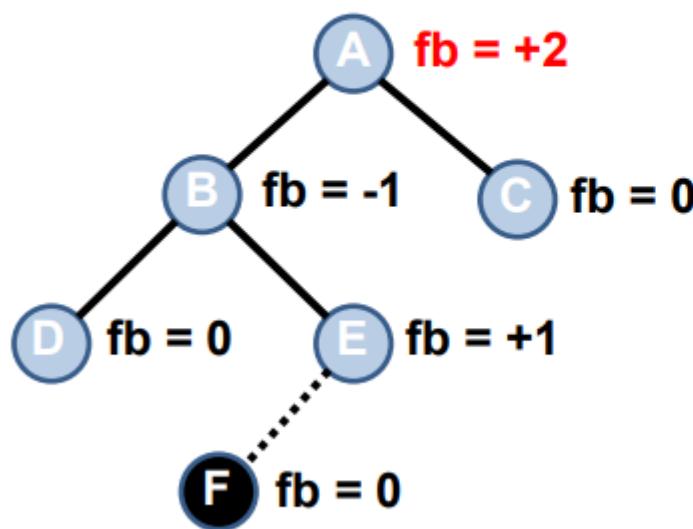
**Árvore AVL e fator de balanceamento de cada nó**





## Rotação LR – Passo a passo

### □ Passo a passo



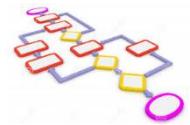
Inserção do nó F na árvore

Árvore fica desbalanceada no  
nó A.

Aplicar Rotação LR no nó A.  
Isso equivale a:

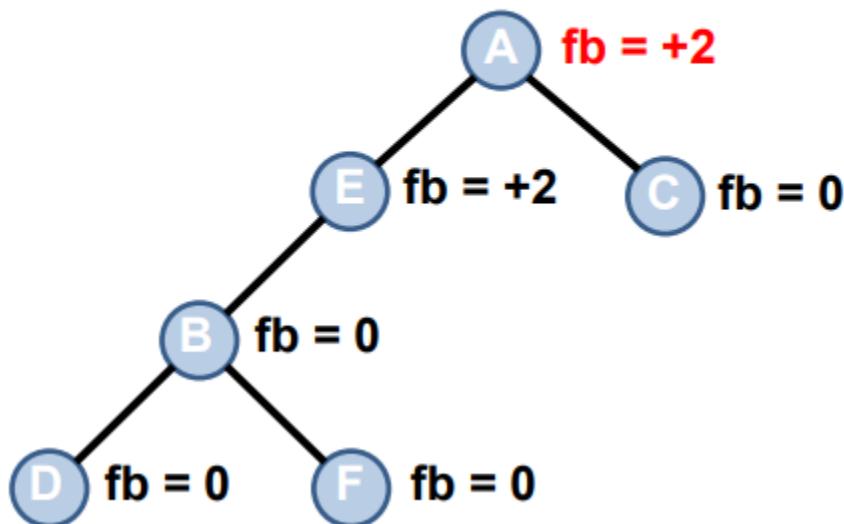
- Aplicar a Rotação RR no nó B
- Aplicar a Rotação LL no nó A





## Rotação LR – Passo a Passo

### □ Passo a passo



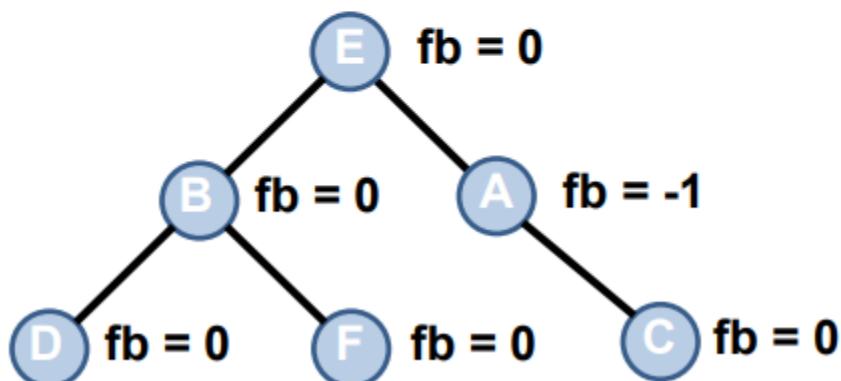
**Árvore após aplicar a  
Rotação RR no nó B**





## Rotação LR – Passo a Passo

### □ Passo a passo



**Árvore após aplicar a  
Rotação LL no nó A**

**Árvore Balanceada**

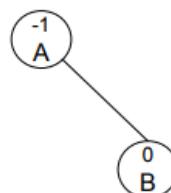




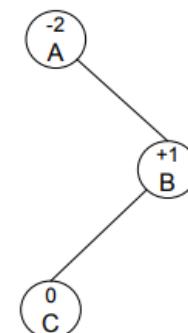
## Rotação RL

- RL: **Y** inserido na subárvore esquerda da subárvore direita de **A**

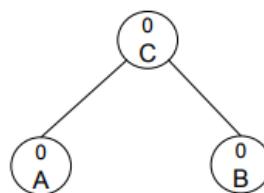
*Subárvore balanceada*

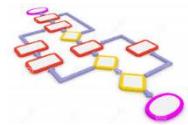


*Subárvore desbalanceada após inserção*



*Subárvore rebalanceada*

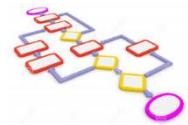




## Rotação RL

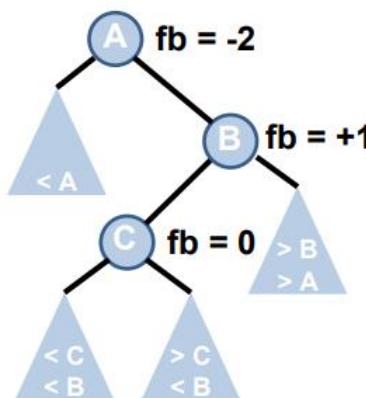
- Rotação RL ou rotação dupla à esquerda
  - um novo nó é inserido na **sub-árvore da esquerda do filho direito de A**
    - A é o nó desbalanceado
  - É necessário fazer uma rotação dupla, de modo que o nó **C** se torne o pai dos nós **A** (filho da esquerda) e **B** (filho da direita)
    - Rotação LL em **B**
    - Rotação RR em **A**



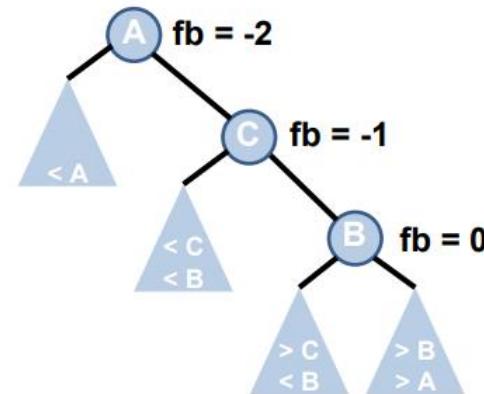


# Rotação RL

## Exemplo



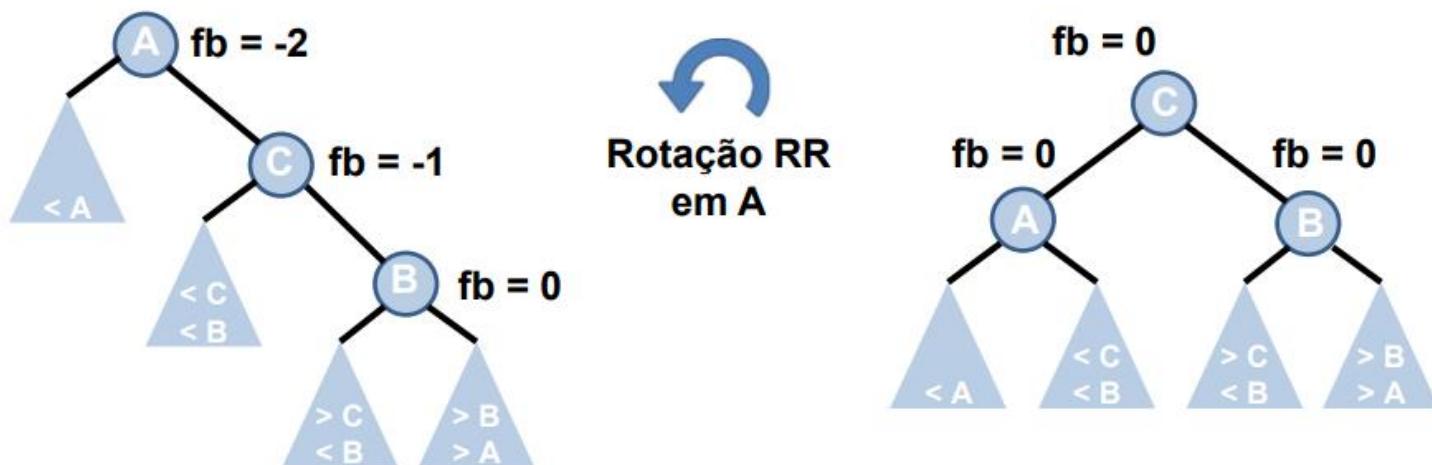
Rotação LL  
em B

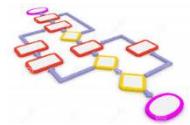




# Rotação RL

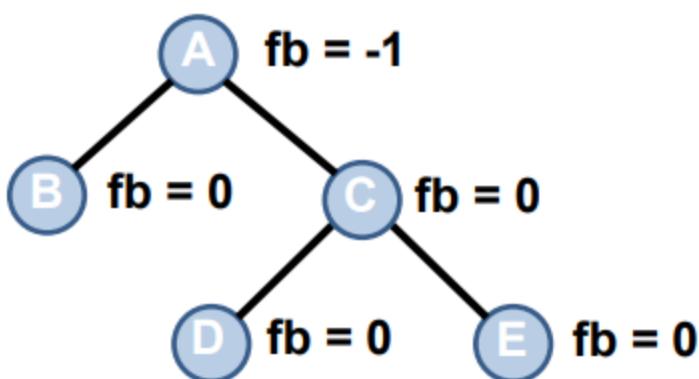
## Exemplo





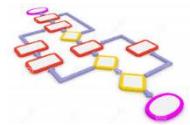
## Rotação RL – Passo a Passo

### □ Passo a passo



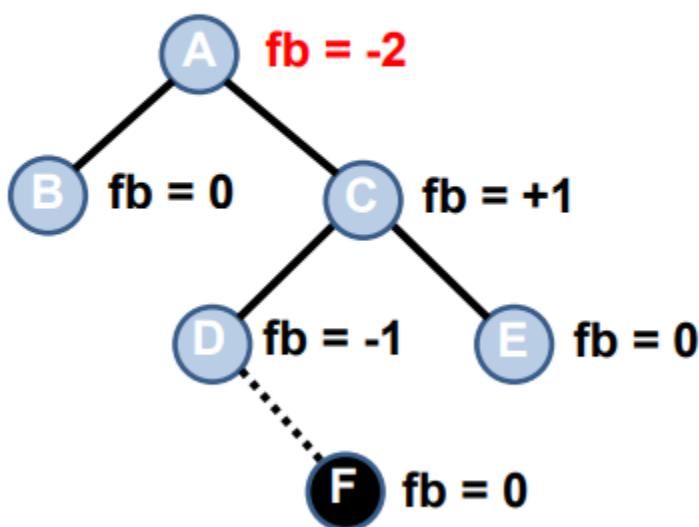
**Árvore AVL e fator de balanceamento de cada nó**





## Rotação RL – Passo a passo

### □ Passo a passo



Inserção do nó F na árvore

Árvore fica desbalanceada no  
nó A.

Aplicar Rotação RL no nó A.  
Isso equivale a:

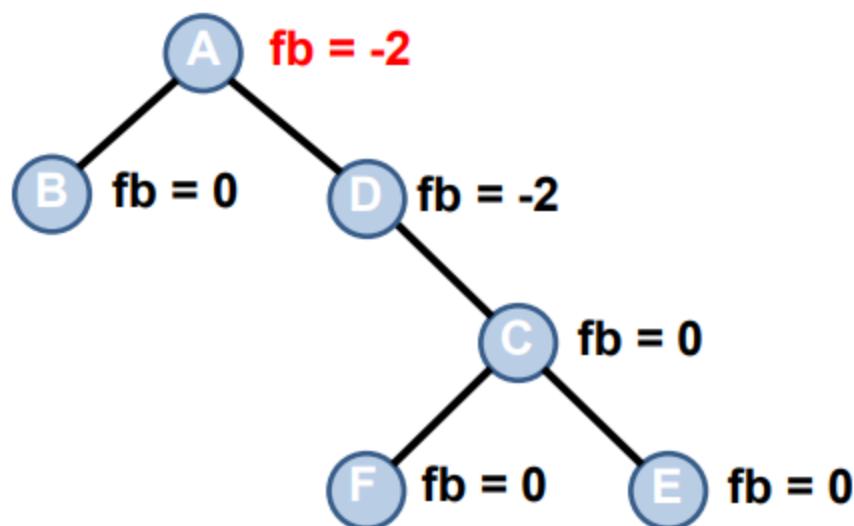
- Aplicar a Rotação LL no nó C
- Aplicar a Rotação RR no nó A





## Rotação RL – Passo a passo

### □ Passo a passo



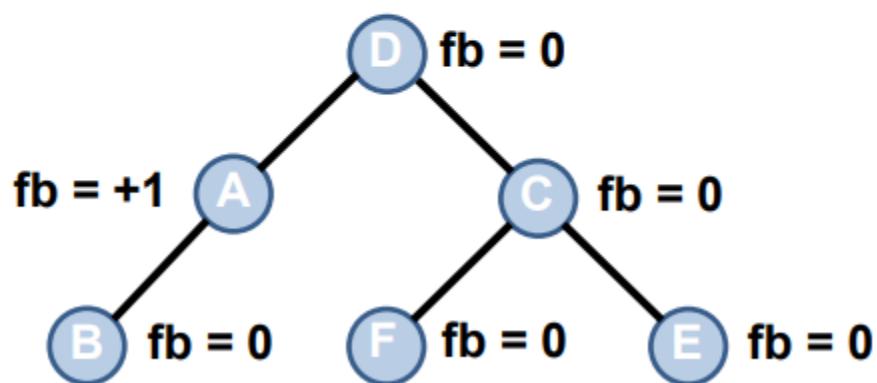
Árvore após aplicar a  
Rotação LL no nó C





## Rotação RL – Passo a Passo

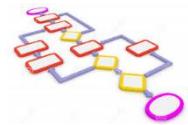
### □ Passo a passo



Árvore após aplicar a  
Rotação RR no nó A

Árvore Balanceada



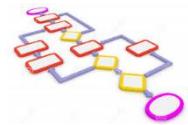


# Quando usar cada rotação?

- Uma dúvida muito comum é quando utilizar cada uma das quatro rotações

Fator de Balanceamento de A	Fator de Balanceamento de B	Posições dos nós B e C em relação ao nó A	Rotação
+2	+1	B é filho à esquerda de A C é filho à esquerda de B	LL
-2	-1	B é filho à direita de A C é filho à direita de B	RR
+2	-1	B é filho à esquerda de A C é filho à direita de B	LR
-2	+1	B é filho à de direita A C é filho à esquerda de B	RL



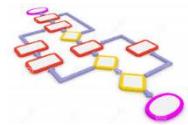


# Quando usar cada rotação?

- Sinais iguais: rotação simples
  - Sinal positivo: rotação à direita (LL)
  - Sinal negativo: rotação à esquerda (RR)

Fator de Balanceamento de A	Fator de Balanceamento de B	Posições dos nós B e C em relação ao nó A	Rotação
+2	+1	B é filho à esquerda de A C é filho à esquerda de B	LL
-2	-1	B é filho à direita de A C é filho à direita de B	RR
+2	-1	B é filho à esquerda de A C é filho à direita de B	LR
-2	+1	B é filho à de direita A C é filho à esquerda de B	RL



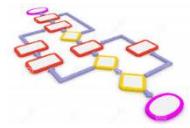


# Quando usar cada rotação?

- Sinais diferentes: rotação dupla
  - A positivo: rotação dupla a direita (LR)
  - A negativo: rotação dupla a esquerda (RL)

Fator de Balanceamento de A	Fator de Balanceamento de B	Posições dos nós B e C em relação ao nó A	Rotação
+2	+1	B é filho à esquerda de A C é filho à esquerda de B	LL
-2	-1	B é filho à direita de A C é filho à direita de B	RR
+2	-1	B é filho à esquerda de A C é filho à direita de B	LR
-2	+1	B é filho à de direita A C é filho à esquerda de B	RL

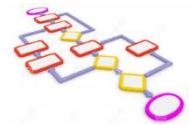




# Árvore AVL - Inserção

- Para inserir um valor **V** na árvore
  - Se a raiz é igual a **NULL**, insira o nó
  - Se **V** é menor do que a raiz: vá para a **sub-árvore esquerda**
  - Se **V** é maior do que a raiz: vá para a **sub-árvore direita**
  - Aplique o método **recursivamente**
- Dessa forma, percorremos um conjunto de nós da árvore até chegar ao nó folha que irá se tornar o pai do novo nó

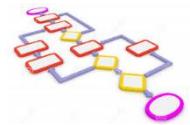




# Árvore AVL - Inserção

- Uma vez inserido o novo nó
  - Devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós visitados
  - Aplicar a rotação necessária para restabelecer o balanceamento da árvore se o fator de balanceamento for +2 ou -2



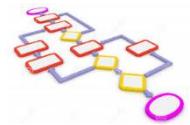


# Árvore AVL - Inserção

```
int insere_ArvAVL(ArvAVL *raiz, int valor) {
    int res;
    if(*raiz == NULL) { //árvore vazia ou nó folha
        struct NO *novo;
        novo = (struct NO*)malloc(sizeof(struct NO));
        if(novo == NULL)
            return 0;

        novo->info = valor;
        novo->altura = 0;
        novo->esq = NULL;
        novo->dir = NULL;
        *raiz = novo;
        return 1;
    }
    //continua...
}
```





# Árvore AVL - Inserção

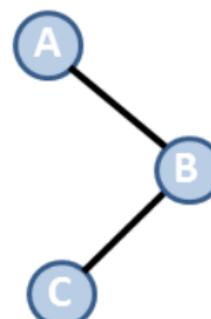
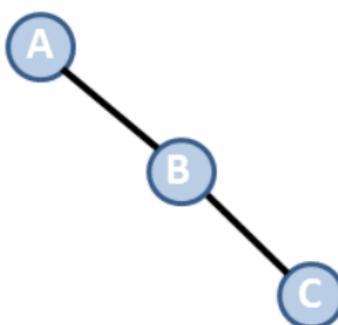
```
//continuação
struct NO *atual = *raiz;
if(valor < atual->info){
    if((res=insere_ArvAVL(&(atual->esq), valor))==1) {
        if(fatorBalanceamento_NO(atual) >= 2) {
            if(valor < (*raiz)->esq->info ) {
                RotacaoLL(raiz);
            } else {
                RotacaoLR(raiz);
            }
        }
    }
}
```

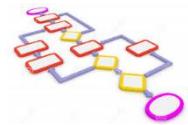




# Árvore AVL - Inserção

```
//continuação
else{
    if(valor > atual->info) {
        if((res=insere_ArvAVL(&(atual->dir), valor))==1) {
            if(fatorBalanceamento_NO(atual) >= 2) {
                if((*raiz)->dir->info < valor) {
                    RotacaoRR(raiz);
                } else{
                    RotacaoRL(raiz);
                }
            }
        }
    } else{
        printf("Valor duplicado!!\n");
        return 0;
    }
}
atual->altura = maior(altura_NO(atual->esq),
                      altura_NO(atual->dir)) + 1;
return res;
}
```





# Árvore AVL - Inserção

## □ Passo a passo

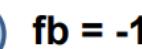
Insere valor: 1



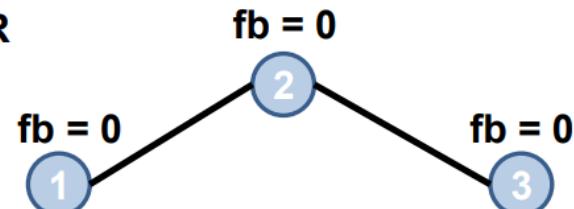
Insere valor: 2



Insere valor: 3



Nó “1” desbalanceado  
Aplicar Rotação RR

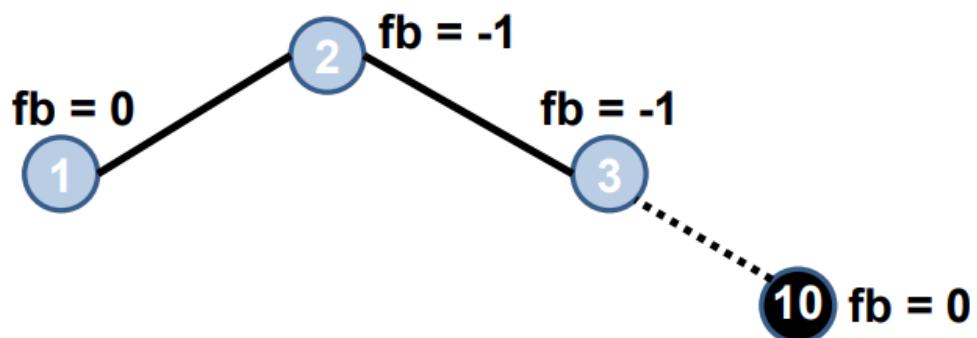


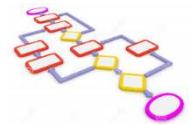


# Árvore AVL - Inserção

## □ Passo a passo

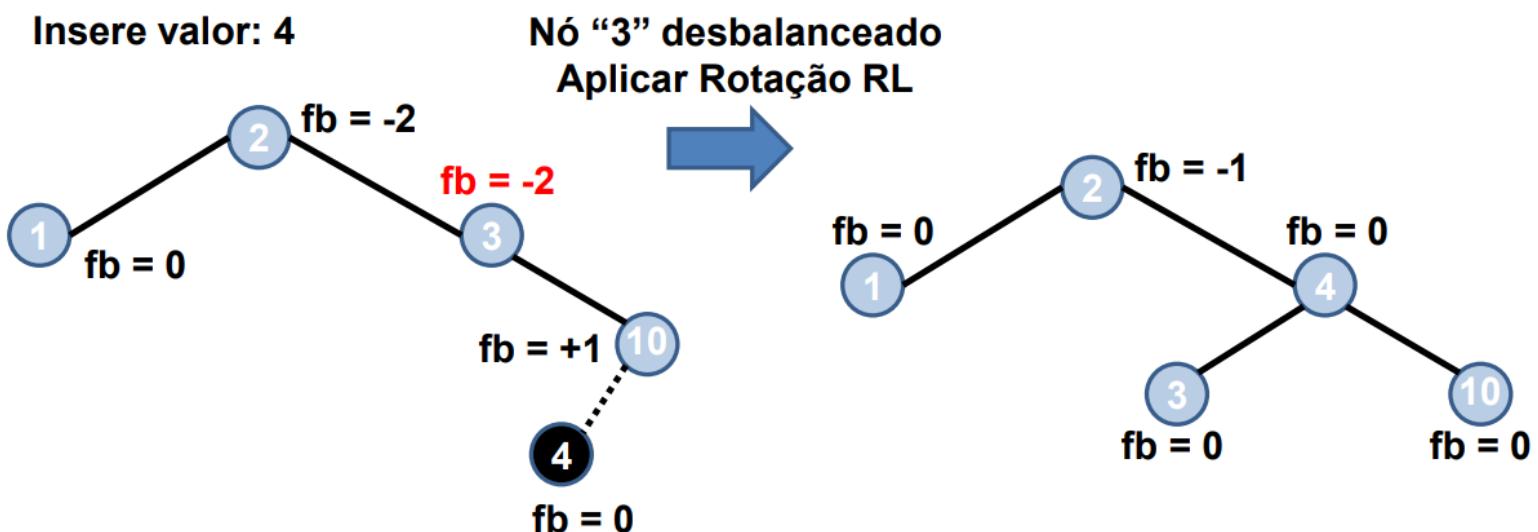
Insere valor: 10





# Árvore AVL - Inserção

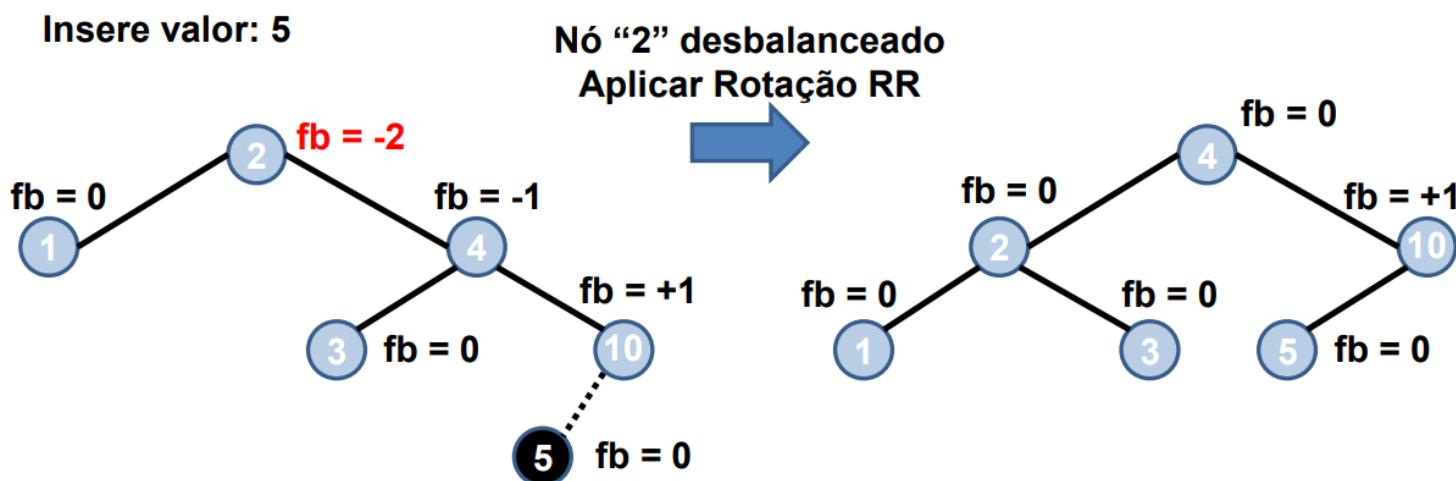
## □ Passo a passo





# Árvore AVL - Inserção

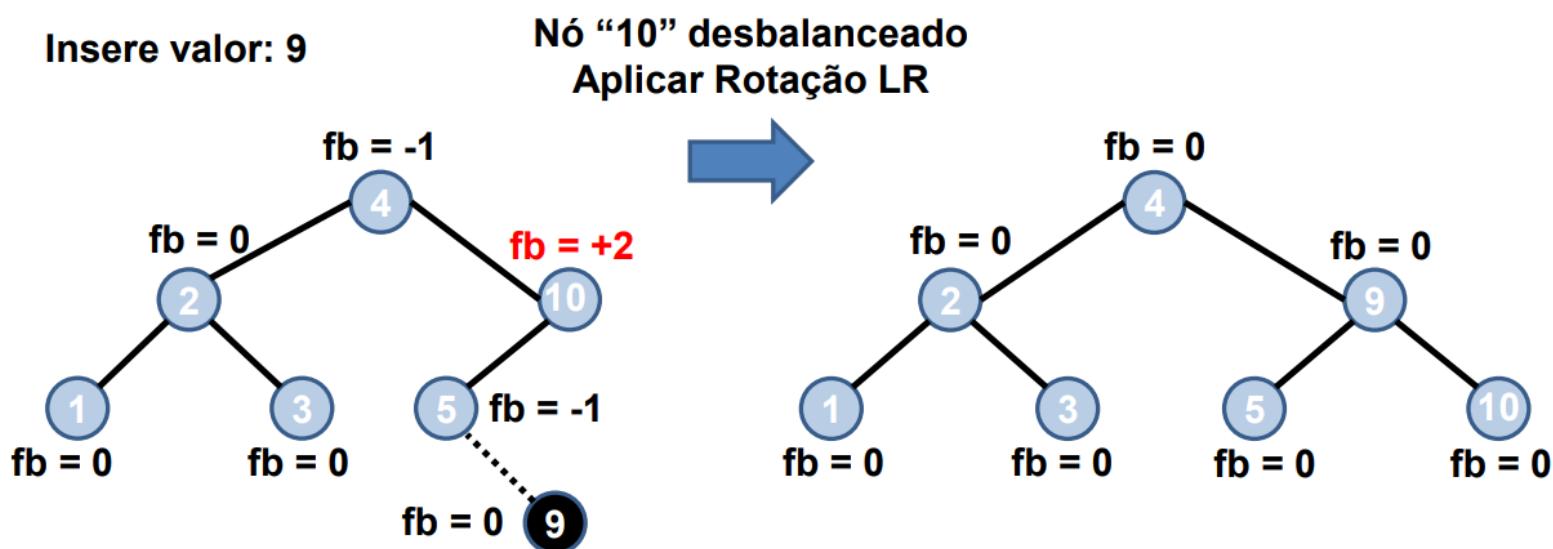
## □ Passo a passo

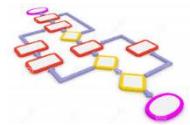




# Árvore AVL - Inserção

## □ Passo a passo





# Árvore AVL - Remoção

- Como na inserção, temos que percorremos um conjunto de nós da árvore até chegar ao nó que será removido
  - Existem 3 tipos de remoção
    - Nó folha (sem filhos)
    - Nó com 1 filho
    - Nó com 2 filhos

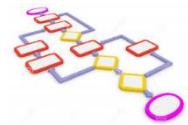




# Árvore AVL - Remoção

- Uma vez removido o nó
  - Devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós visitados
  - Aplicar a rotação necessária para restabelecer o balanceamento da árvore se o fator de balanceamento for **+2** ou **-2**
    - **Remover** um nó da sub-árvore **direita** equivale a **inserir** um nó na sub-árvore **esquerda**





# Árvore AVL - Remoção

## □ Remoção

### □ Trabalha com 2 funções

- Busca pelo nó
- Remoção do nó com 2 filhos

```
8 int remove_ArvAVL(ArvAVL *raiz, int valor) {
9     /*
10      FUNÇÃO RESPONSÁVEL PELA BUSCA
11      DO NÓ A SER REMOVIDO
12      */
13 }
14 struct NO* procuraMenor(struct NO* atual) {
15     /*
16      FUNÇÃO RESPONSÁVEL POR TRATAR OS
17      A REMOÇÃO DE UM NÓ COM 2 FILHOS
18      */
19 }
```

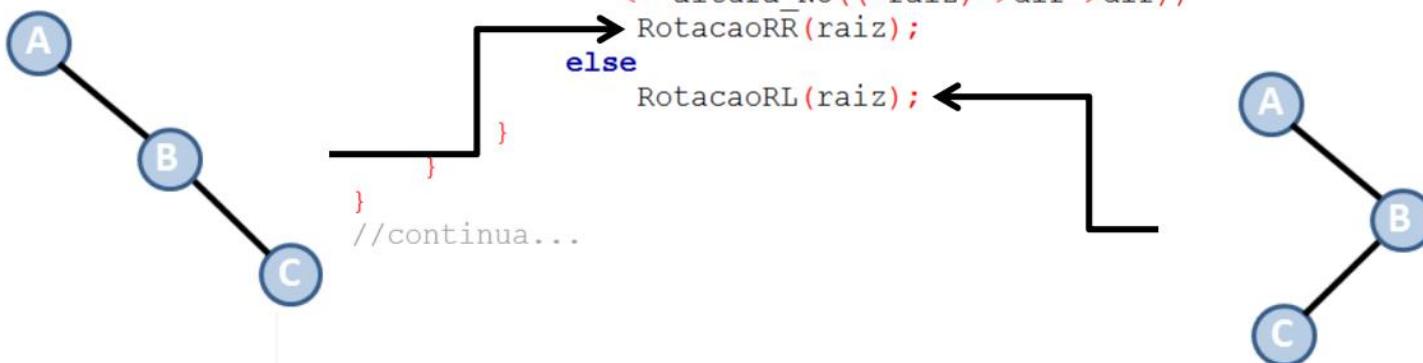


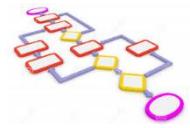


# Árvore AVL - Remoção

## □ Remoção

```
int remove_ArvAVL(ArvAVL *raiz, int valor) {
    if(*raiz == NULL) // valor não existe
        printf("valor não existe!!\n");
        return 0;
    }
    int res;
    if(valor < (*raiz)->info) {
        if((res=remove_ArvAVL(&(*raiz)->esq, valor)) == 1)
            if(fatorBalanceamento_NO(*raiz) >= 2) {
                if(altura_NO((*raiz)->dir->esq)
                    <= altura_NO((*raiz)->dir->dir))
                    RotacaoRR(raiz);
                else
                    RotacaoRL(raiz);
            }
        }
    }
    //continua...
}
```

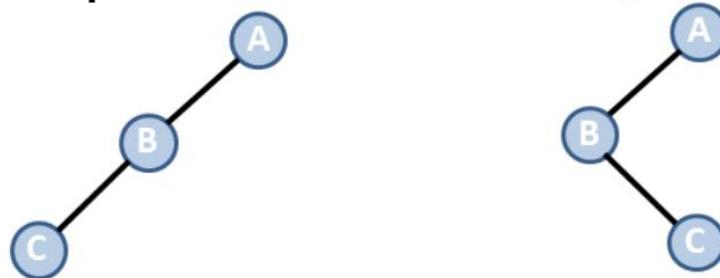


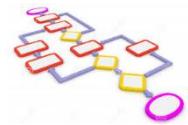


# Árvore AVL - Remoção

## □ Remoção

```
//continuação...
if((*raiz)->info < valor) {
    if(res=remove_ArvAVL(&(*raiz)->dir, valor))==1) {
        if(fatorBalanceamento_NO(*raiz) >= 2) {
            if(altura_NO((*raiz)->esq->dir)
                <= altura_NO((*raiz)->esq->esq) )
                RotacaoLL(raiz);
            else
                RotacaoLR(raiz);
        }
    }
}
//continua...
```





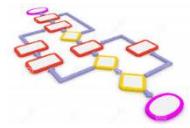
# Árvore AVL - Remoção

Pai tem 1 ou  
nenhum filho

```
if((*raiz)->info == valor){  
    if(((*raiz)->esq == NULL || (*raiz)->dir == NULL)) { // nó tem 1 filho ou nenhum  
        struct NO *oldNode = (*raiz);  
        if((*raiz)->esq != NULL)  
            *raiz = (*raiz)->esq;  
        else  
            *raiz = (*raiz)->dir;  
        free(oldNode);  
    }else { // nó tem 2 filhos  
        struct NO* temp = procuraMenor((*raiz)->dir);  
        (*raiz)->info = temp->info;  
        remove_ArvAVL(&(*raiz)->dir, (*raiz)->info);  
        if(fatorBalanceamento_NO(*raiz) >= 2){  
            if(altura_NO((*raiz)->esq->dir) <= altura_NO((*raiz)->esq->esq))  
                RotacaoLL(raiz);  
            else  
                RotacaoLR(raiz);  
        }  
    }  
    if (*raiz != NULL)  
        (*raiz)->altura = maior(altura_NO((*raiz)->esq),  
                                altura_NO((*raiz)->dir)) + 1;  
    return 1;  
}  
(*raiz)->altura = maior(altura_NO((*raiz)->esq),  
                        altura_NO((*raiz)->dir)) + 1;  
  
return res;  
}
```

Corrigir a  
altura

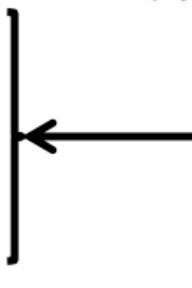




# Árvore AVL - Remoção

## □ Remoção

```
struct NO* procuraMenor(struct NO* atual) {
    struct NO *no1 = atual;
    struct NO *no2 = atual->esq;
    while(no2 != NULL) {
        no1 = no2;
        no2 = no2->esq;
    }
    return no1;
}
```



Procura pelo nó  
mais a esquerda

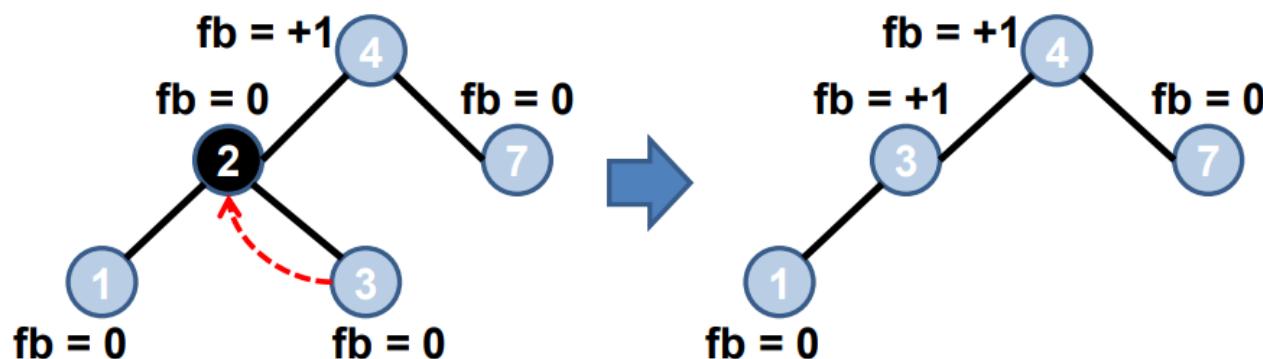




# Árvore AVL - Remoção

## □ Passo a passo

Remove valor: 2

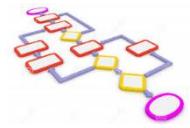




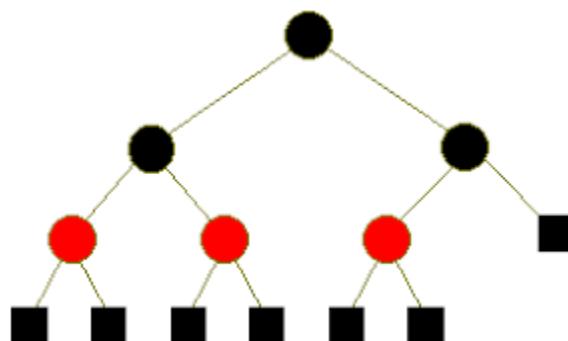
## Atividade

Escrever um programa em C que efetue operações de inserção em uma árvore AVL



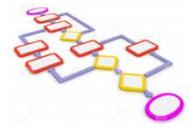


# Árvores Red-Black



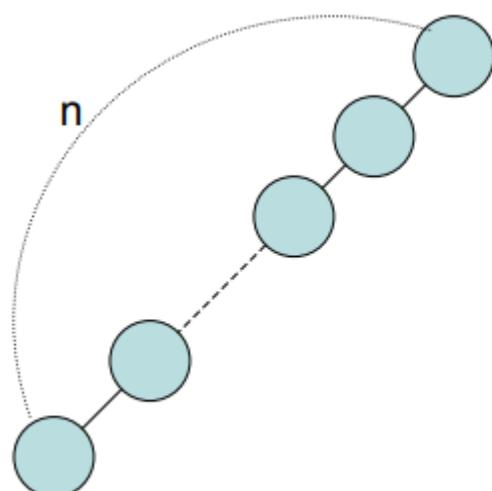
Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUSP  
[aparecidovfreitas@gmail.com](mailto:aparecidovfreitas@gmail.com)



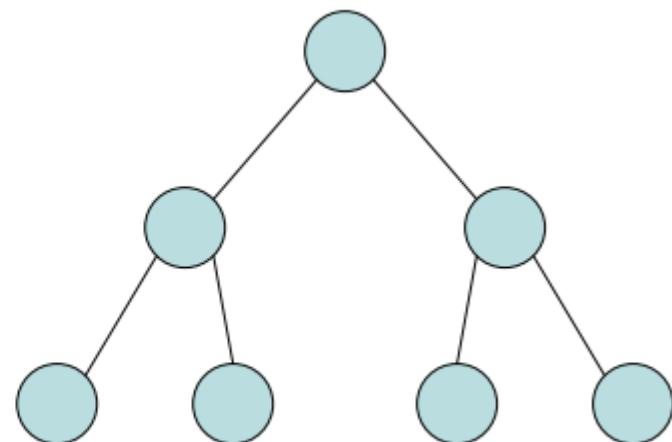


# Motivação

- Árvores Binárias de Busca são excelentes estruturas de dados para algoritmos de searching;
- O desempenho de uma Árvore Binária de Busca está relacionado a sua **altura h**;

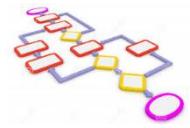


**Pior Caso:  $h = O(n)$**



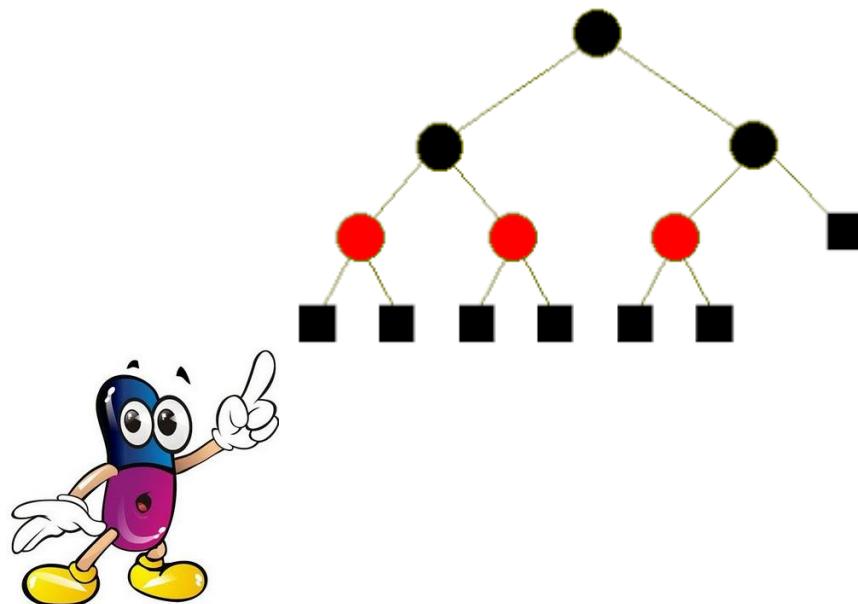
**Melhor Caso:  $h = O(\log n)$**





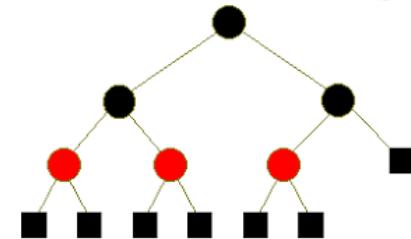
# Motivação

- Árvores Binárias de Busca balanceadas permitem operações de Searching em  $O(\log n)$ ;
- Árvores Binárias **Red-Black** (rubro-negra ou vermelho-preto) são árvores binárias de pesquisa balanceadas;





# Árvores Red-Black



- Foram inventadas por **Bayer** sob o nome Árvores Binárias Simétricas em **1972**, posteriormente às árvores AVL;

Acta Informatica 1, 290—306 (1972)  
© by Springer-Verlag 1972



Rudolf Bayer  
Computer scientist

## Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms\*

R. Bayer

Received January 24, 1972

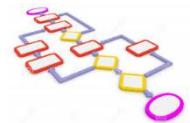
**Summary.** A class of binary trees is described for maintaining ordered sets of data. Random insertions, deletions, and retrievals of keys can be done in time proportional to  $\log N$  where  $N$  is the cardinality of the data-set. Symmetric B-Trees are a modification of B-trees described previously by Bayer and McCreight. This class of trees properly contains the balanced trees.

This paper will describe a further solution to the following well-known problem in information processing: Organize and maintain an index, i.e. an ordered set of keys or virtual addresses, used to access the elements in a set of data, in such a way that random and sequential insertions, deletions, and retrievals can be performed efficiently.

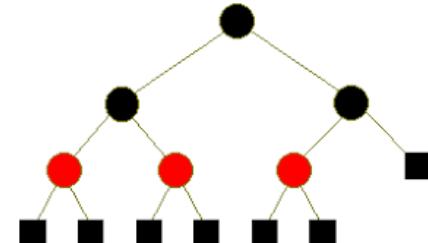
Other solutions to this problem have been described for a one-level store in [1, 3–5, 7] and for a two-level store with a pseudo-random access backup store in [2]. All these techniques use trees to represent the data sets. The class of trees to be described in this paper is a generalization of the trees described in [1, 3–5], but it is not comparable with the BB-trees described in [7]. The following technique is suitable for a one-level store.

<b>Rudolf Bayer</b>	
<b>Born</b>	May 7, 1939 (age 75)
<b>Nationality</b>	German
<b>Institutions</b>	Technical University Munich
<b>Alma mater</b>	University of Illinois at Urbana-Champaign
<b>Thesis</b>	Automorphism Groups and Quotients of Strongly Connected Automata and Monadic Algebras <a href="#">[1]</a> (1966)
<b>Doctoral advisor</b>	Franz Edward Hohn <a href="#">[1]</a>
<b>Known for</b>	B-tree UB-tree red-black tree
<b>Notable awards</b>	Cross of Merit, First class (1999), SIGMOD Edgar F. Codd Innovations Award (2001)





# Árvores Red-Black

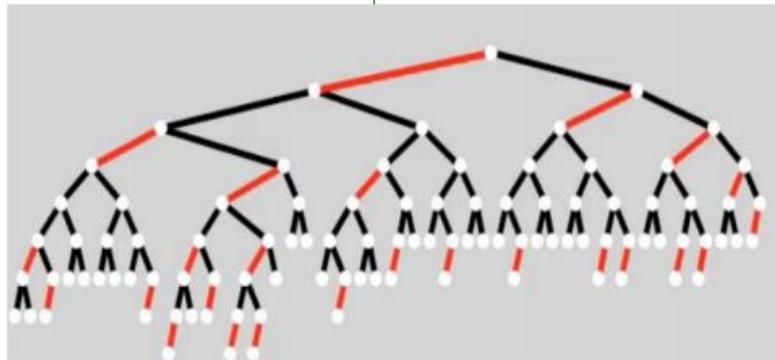


- Posteriormente, em **1978**, Guibas e Sedgewick, atribuiram a coloração na árvore;

**A dichromatic framework for balanced trees**

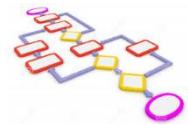
Authors: [Leo J. Guibas](#)  
[Robert Sedgewick](#)

Published in:  
• Proceeding  
SFCS '78 Proceedings of the 19th Annual Symposium on Foundations of Computer Science  
Pages 8-21  
IEEE Computer Society Washington, DC, USA ©1978  
[table of contents](#) [doi>10.1109/SFCS.1978.3](#)

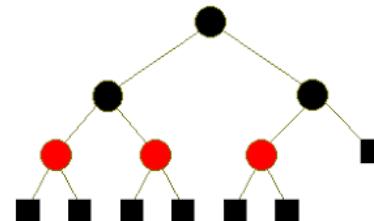


- Adquiriu esse nome pois cada nó possui um atributo de cor (além dos ponteiros para seus filhos), que pode ser **vermelho ou preto**;
- A cor **vermelho** foi escolhida em função da melhor qualidade das impressoras disponíveis na época;



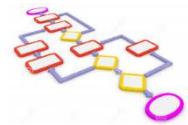


# Árvores Red-Black



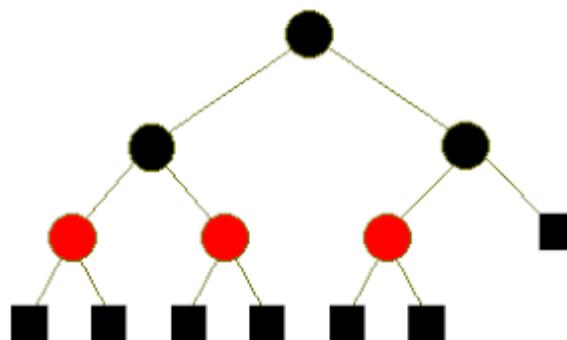
- As árvores vermelho-preto possuem um *flag* extra para armazenar a cor de cada nó, que pode ser **Vermelho** ou **Preto**
- Além deste, cada nó será composto ainda pelos seguintes campos
  - info (os “dados” do nó)
  - fesq (filho esquerdo)
  - fdir (filho direito)
  - pai

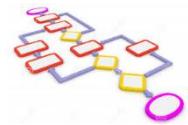




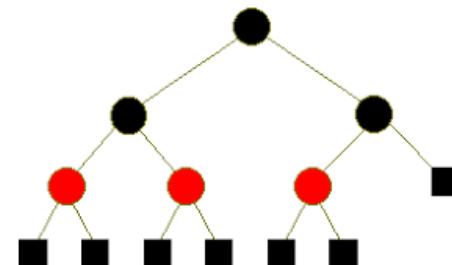
# Árvores Red-Black

- Restringindo o modo como os nós são coloridos desde a raiz até uma folha, assegura-se que **nenhum caminho será maior que duas vezes o comprimento de qualquer outro**, dessa forma, a árvore é aproximadamente balanceada





# Árvores Red-Black



**Prova-se que:**

- Uma árvore vermelho-preto com  $n$  nós tem altura máxima  $2 \log(n + 1)$
- Por serem “balanceadas” as árvores vermelho-preto possuem complexidade logarítmica em suas operações  $O(\log n)$





# Árvores Red-Black – Propriedades

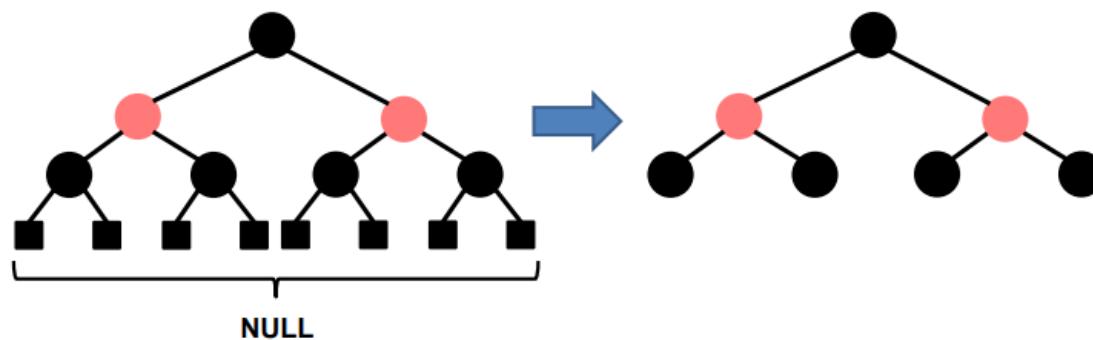
- Além da cor, a árvore deve satisfazer o seguinte conjunto de propriedades
  - Todo nó da árvore é **vermelho** ou **preto**
  - A raiz é sempre **preta**
  - Todo nó folha (**NULL**) é **preto**
  - Se um nó é **vermelho**, então os seus filhos são **pretos**
    - Não existem nós **vermelhos** consecutivos
  - Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém o mesmo número de nós **pretos**





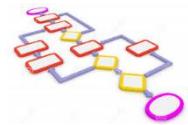
# Árvores Red-Black – Propriedades

- Como todo nó folha termina com dois ponteiros para **NULL**, eles podem ser ignorados na representação da árvore para fins de didática



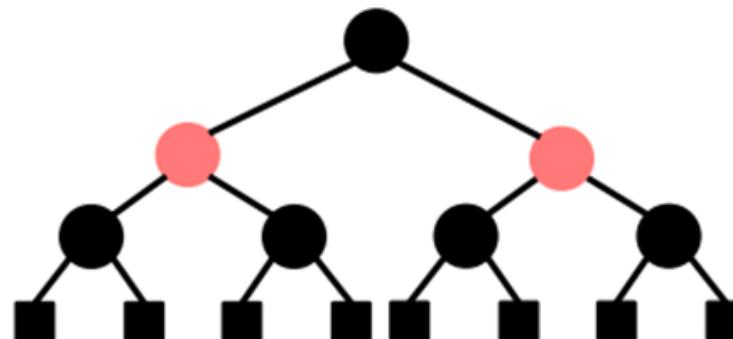
- Ponteiros nulos são considerados nós pretos.

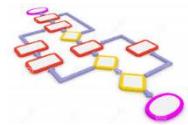




# Árvores Red-Black - Balanceamento

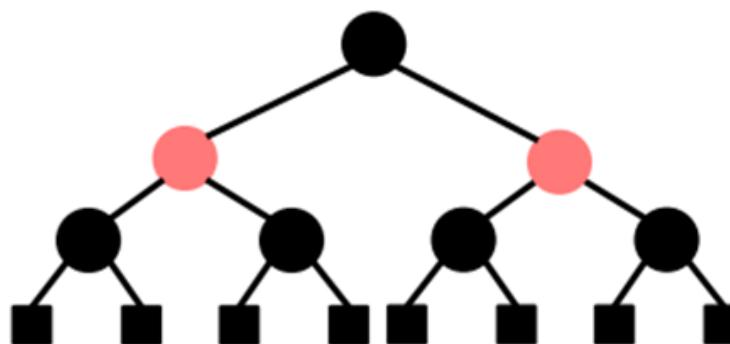
- É feito por meio de rotações e ajuste de cores a cada inserção ou remoção
  - ▣ Mantém o equilíbrio da árvore
  - ▣ Corrigem possíveis violações de suas propriedades
  - ▣ Custo máximo de qualquer algoritmo é  $O(\log N)$

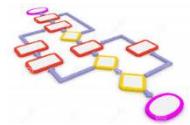




# Árvores Red-Black - Balanceamento

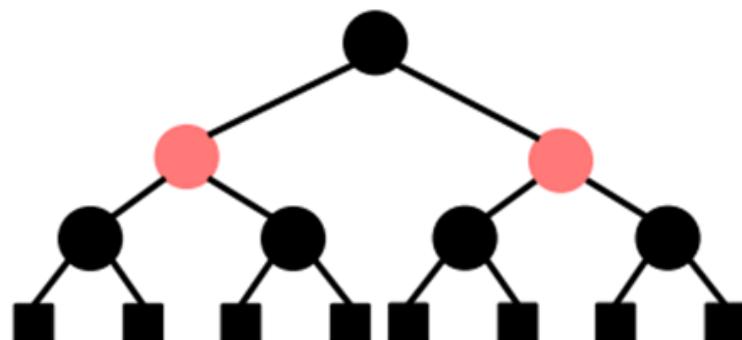
- Permite rebalanceamento **local**;
- Apenas a parte afetada pela **inserção** ou **remoção** é rebalanceada;
- Uso de “**rotações**” e “**ajuste de cores**” na etapa de rebalanceamento;
- Essas operações corrigem as propriedades que foram **violadas**;
- A árvore red-black busca manter-se como uma árvore binária binária quase completa;
- Custo de qualquer algoritmo é máximo  **$O(\log n)$** .

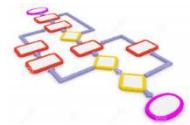




# Árvores Red-Black - Balanceamento

- O objetivo das operações de rotação e de ajuste de cores é garantir que as propriedades **não** sejam violadas durante a inserção e remoção de um nó, restituindo o equilíbrio da árvore;
- Assim, caso algumas das propriedades que definem a árvore red-black não seja atendida, serão realizadas **rotações** e/ou **ajustes de cores**, de forma que a árvore permaneça **balanceada**;





## Red-Black x AVL

- Na teoria, possuem a mesma complexidade computacional
  - Inserção, remoção e busca:  $O(\log N)$
- Na prática, a árvore AVL é mais rápida na operação de busca, e mais lenta nas operações de inserção e remoção
  - A árvore AVL é mais balanceada do que a árvore Rubro-Negra, o que acelera a operação de busca





## Red-Black x AVL

- AVL: balanceamento mais rígido
  - Maior custo na operação de inserção e remoção
    - No pior caso, uma operação de remoção pode exigir  $O(\log N)$  rotações na árvore AVL, mas apenas 3 rotações na árvore Rubro-Negra.
- Qual usar?
  - Operação de busca é a mais usada?
    - Melhor usar uma árvore AVL
  - Inserção ou remoção são mais usadas?
    - Melhor usar uma árvore Rubro-Negra

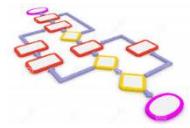




## Red-Black x AVL

- Árvores Rubro-Negra são de uso mais geral do que as árvores AVL
  - Ela é utilizada em diversas aplicações e bibliotecas de linguagens de programação
  - Exemplos
    - **Java:** java.util.TreeMap , java.util.TreeSet
    - **C++ STL:** map, multimap, multiset
    - **Linux kernel:** completely fair scheduler, linux/rbtree.h





# Red-Black - Implementação

<http://www.cs.usfca.edu/~galles/visualization/RedBlack.html>



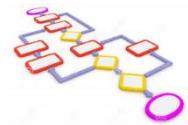


# Árvores Múltiplas



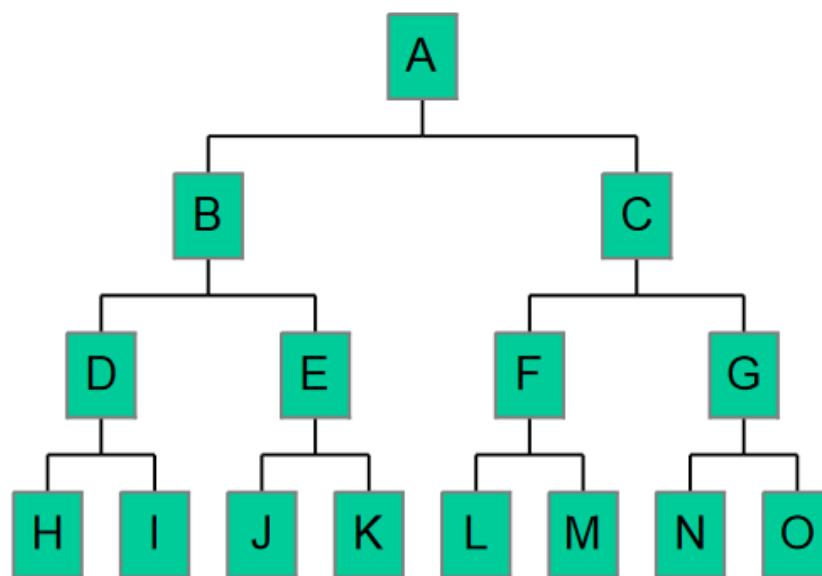
Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUSP  
[aparecidovfreitas@gmail.com](mailto:aparecidovfreitas@gmail.com)





## Buscas com árvores

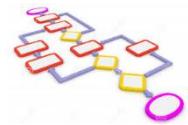
- ⊕ **Para grandes volumes de dados**, o emprego de árvores binárias de pesquisa trás o inconveniente de apresentarem grande altura.



- ⊕ Indexação em grandes volumes de dados tem maior eficiência com o emprego de Árvores n-árias de pesquisa.



Fonte: Navathe – Capítulo 18

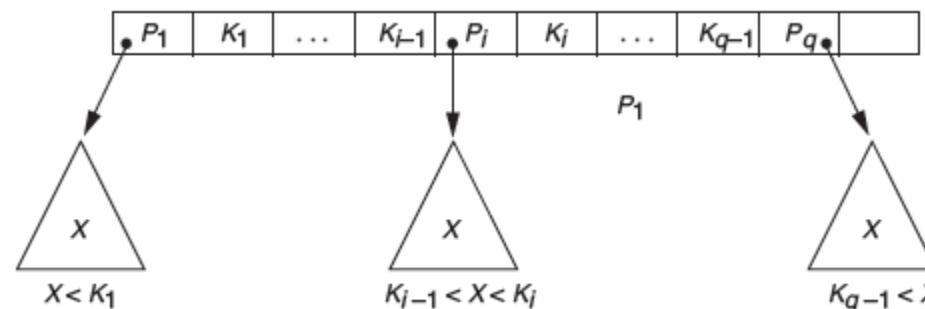


# Árvores n-ária de Pesquisa

- ⊕ Uma árvore n-ária de pesquisa de ordem  $p$  é uma árvore tal que cada nó contém no máximo  $p-1$  valores de pesquisa e  $p$  ponteiros na ordem:

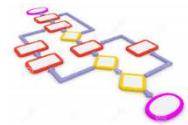
$P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q$ , onde  $q \leq p$ .

- ⊕ Cada  $P_i$  é um ponteiro para um nó filho (ou **null**), e cada  $K_i$  é um valor de pesquisa de algum conjunto ordenado de valores.

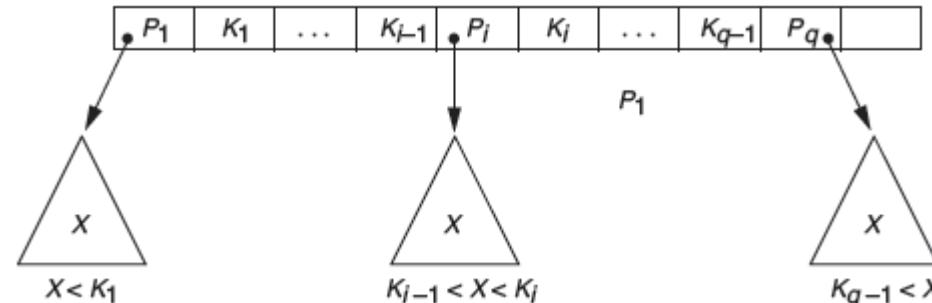


Fonte: Navathe – Capítulo 18





# Árvores n-ária de Pesquisa



- ⊕ Em cada nó ,  $K_1 < K_2 < \dots < K_{q-1}$
- ⊕ Para todos os valores **X** na subárvore apontada por  $P_i$ , temos:

$K_{i-1} < X < K_i$  para  $1 < i < q$  ;

**X** <  $K_i$  para  $i = 1$ ; e

$K_{i-1} < X$  para  $i = q$ .

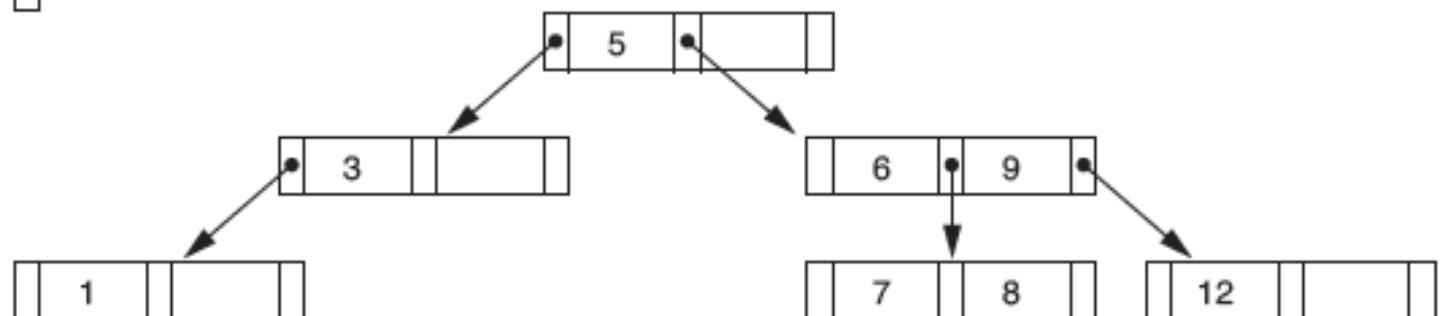
**Obs.** Sempre que se procura uma um valor **X**, deve-se seguir o ponteiro  $P_i$  apropriado, de acordo com as fórmulas acima.





# Árvores n-ária de Pesquisa – Exemplo

- Ponteiro de nó de árvore
- Ponteiro de árvore nulo

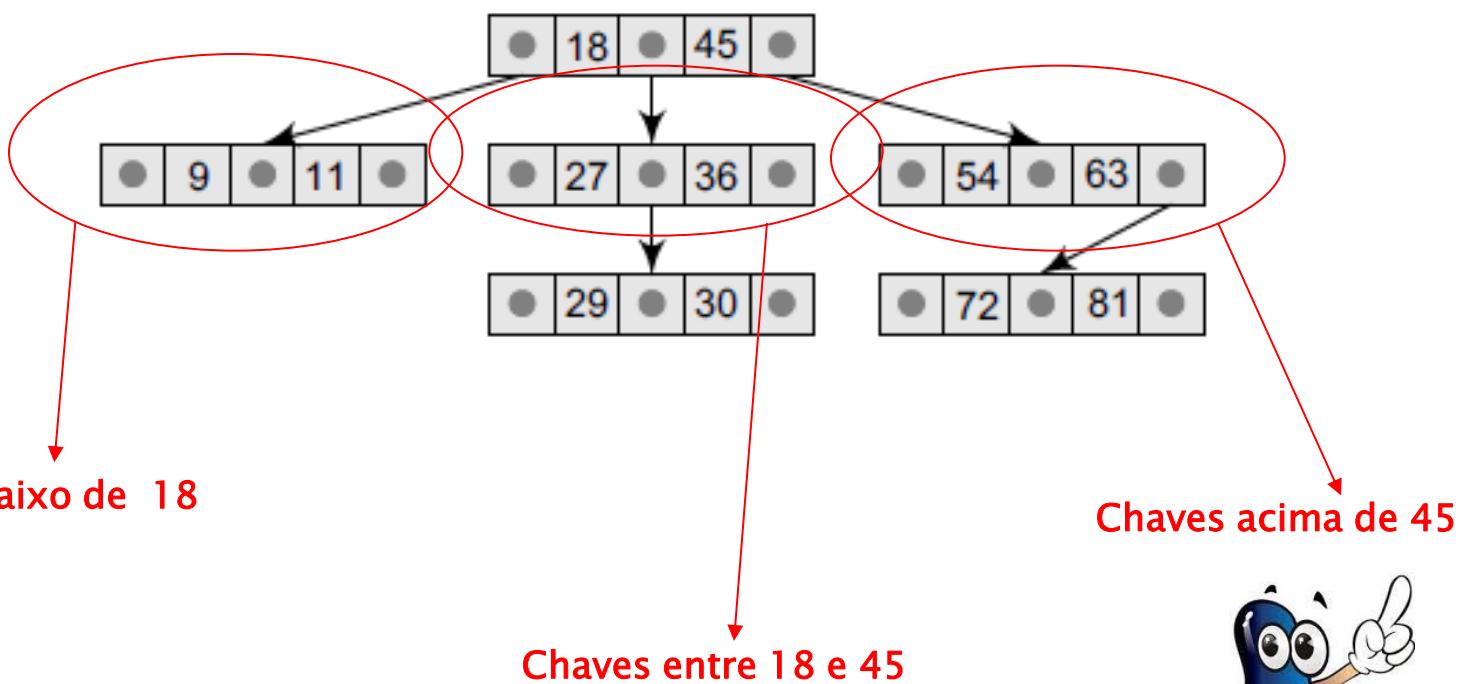


Uma árvore de pesquisa de ordem  $p = 3$ .





# Árvores n-ária de Pesquisa – Exemplo



Chaves abaixo de 18

Chaves acima de 45

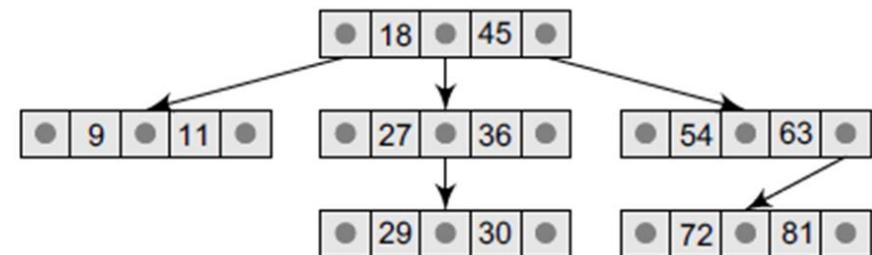
Chaves entre 18 e 45

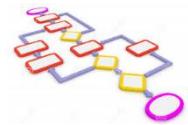




# Árvores n-ária de Pesquisa

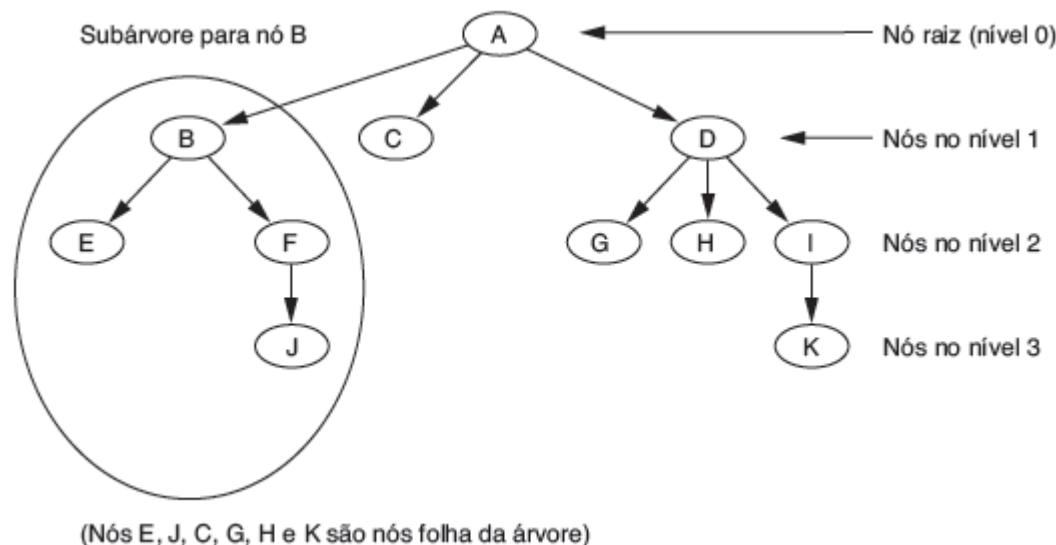
- ⊕ Uma **árvore n-ária** de pesquisa pode ser usada como um mecanismo para procurar um registro armazenado em disco.
- ⊕ Cada valor de chave na árvore é associado a um ponteiro para o registro no arquivo de dados que tem esse valor.
- ⊕ Quando um registro é inserido no arquivo, deve-se atualizar a árvore de pesquisa com os correspondentes ponteiros para o registro.
- ⊕ Em geral, os algoritmos de inserção e deleção de registros não garantem que a árvore de pesquisa seja balanceada.





# Árvores n-ária de Pesquisa Balanceada

- ⊕ Uma **árvore n-ária** de pesquisa é balanceada quando os nós folha estão no mesmo nível;



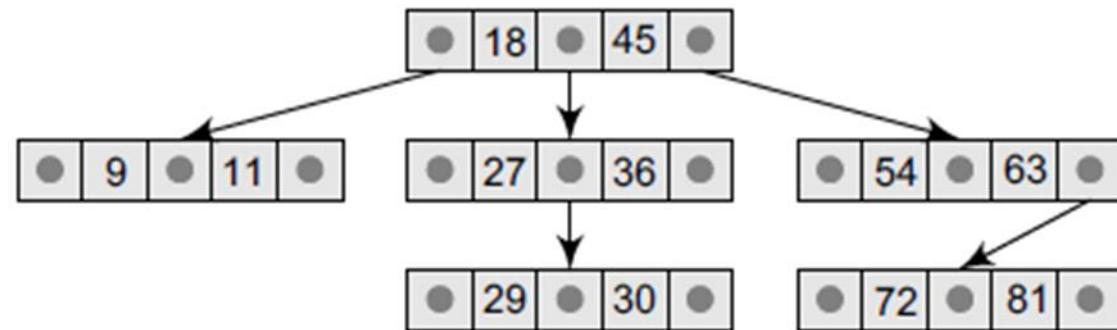
A árvore acima não é balanceada, pois há nós folha nos níveis 1, 2 e 3.





## Importância do Balanceamento da árvore

- ⊕ Garantir que os nós sejam igualmente distribuídos, de modo a **minimizar** a profundidade;
- ⊕ Tornar a velocidade de pesquisa **uniforme**, de modo que o tempo médio para a busca de qualquer chave aleatória seja aproximadamente o mesmo.





## Problemas com a Árvore n-ária de pesquisa

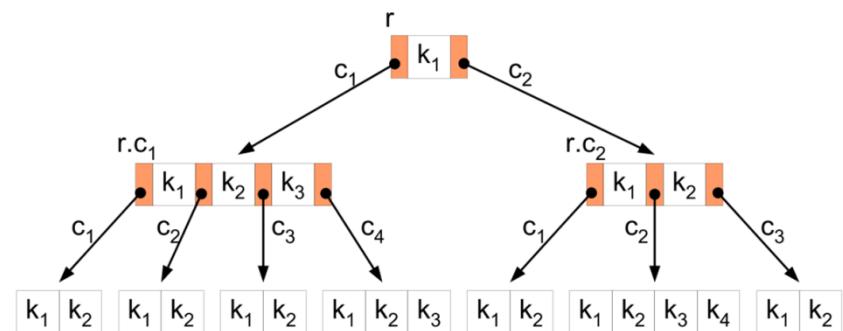
- ⊕ **Inclusões** e **exclusões** de registros podem causar muita **reestruturação** da árvore;
- ⊕ Em caso de muitas exclusões, nós podem ficar quase vazios, desperdiçando espaço de armazenamento e aumentando o número de níveis.
- ⊕ Árvores **B-tree** podem resolver esses problemas.





## Árvore B-tree

- ⊕ Estrutura de dados projetada para memória secundária;
- ⊕ Permite a inserção, remoção e busca de chaves com complexidade logarítmica;
- ⊕ **Por essa razão, é largamente empregada em sistemas de bancos de dados;**
- ⊕ Inventada por Rudolf Bayer e Edward Meyers McCreight em 1971;
- ⊕ Especula-se que o B venha da palavra balanceada;
- ⊕ São árvores de pesquisa **n-ária**, porém com restrições adicionais que garantem o balanceamento da árvore.

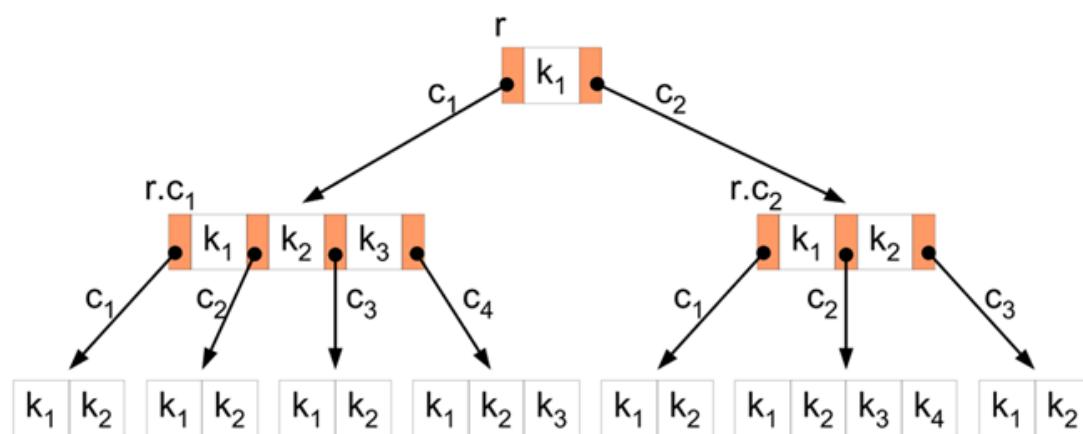


Fonte: Navathe – Capítulo 18



## Árvore B-tree - Algoritmo

- ⊕ Uma **B-tree** começa com um único nó raiz (que inicialmente também é folha) no nível 0;
- ⊕ Quando o nó raiz está cheio (com  $p-1$  valores de chave) e tentamos inserir outra entrada na árvore, o nó raiz se divide (**Split**) em dois nós no nível 1. Somente o valor do meio é mantido no nó raiz, e o restante dos valores é dividido por igual entre os dois nós;
- ⊕ Quando um nó não-raiz está cheio e uma nova entrada é inserida nele, esse nó é dividido em dois nós no mesmo nível, e a entrada do meio é movida para o nó pai. Se o nó pai estiver cheio, ele também é dividido.



Fonte: Navathe – Capítulo 18