

Estruturas de Dados

Árvores Balanceadas e Árvore AVL

Prof. André Luiz Moura
<andreluiz@inf.ufg.br>

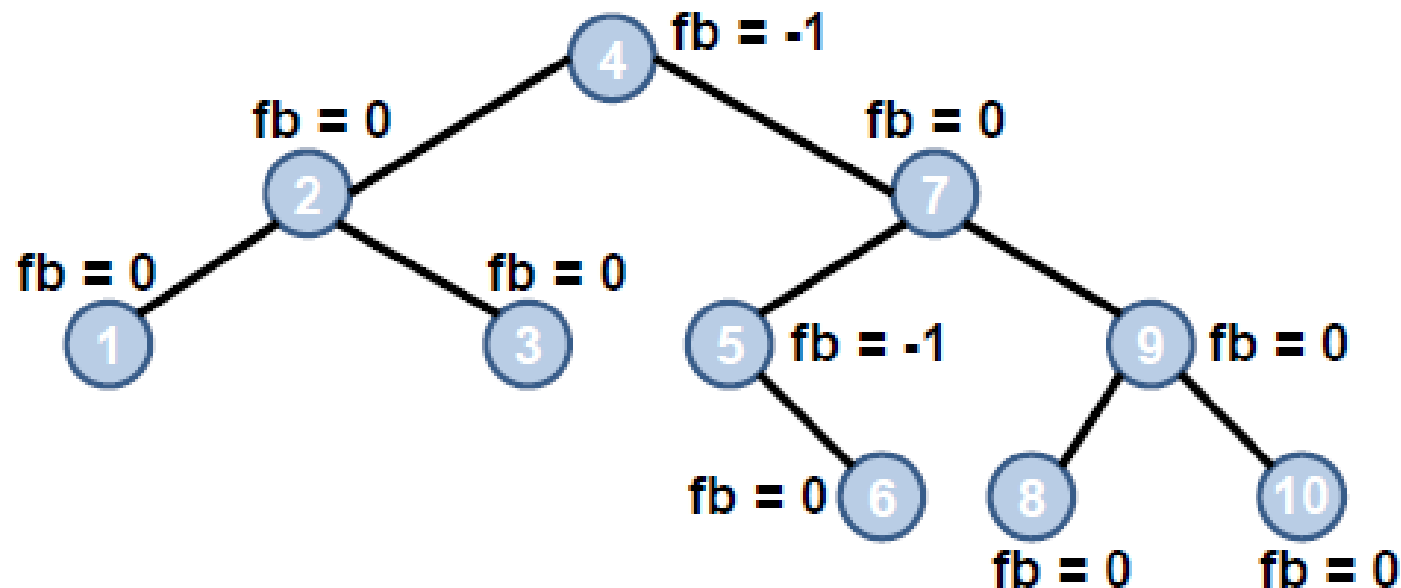


INSTITUTO DE
INFORMÁTICA
UFG

Árvores Balanceadas: definição

É uma árvore binária onde as alturas das subárvores “esquerda” e “direita” de cada nó diferem de no máximo uma unidade.

Essa diferença é chamada de “fator de balanceamento” do nó.



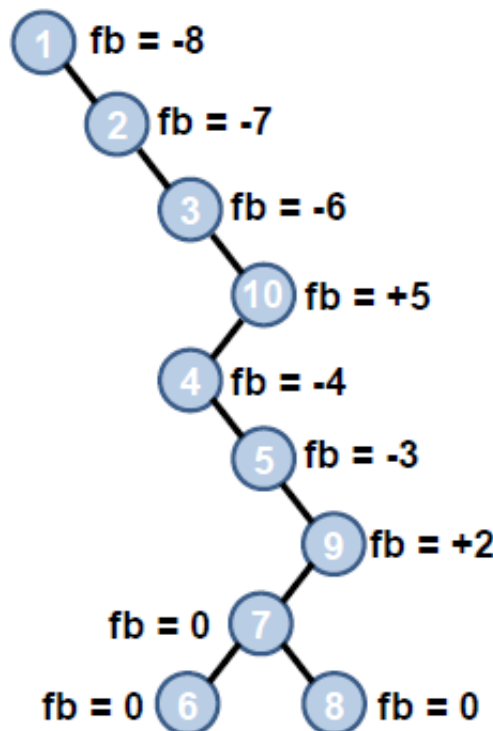
Árvores Balanceadas

A eficiência da busca em uma árvore binária depende do seu balanceamento.

Problema:

Algoritmos de inserção e remoção não garantem que a árvore gerada a cada passo seja balanceada

Sequência de inserções em ordem de “escada”



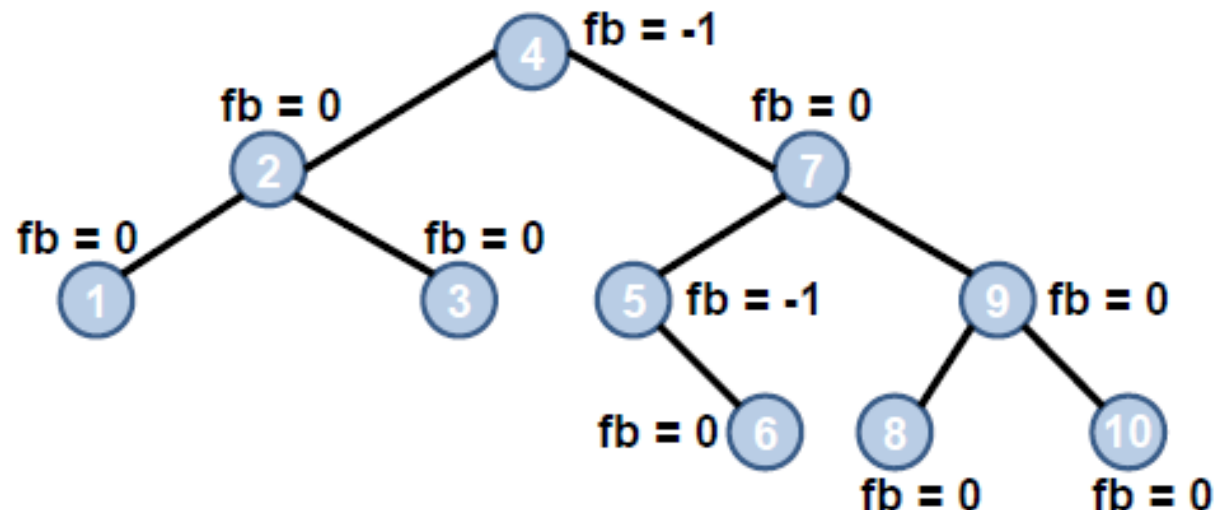
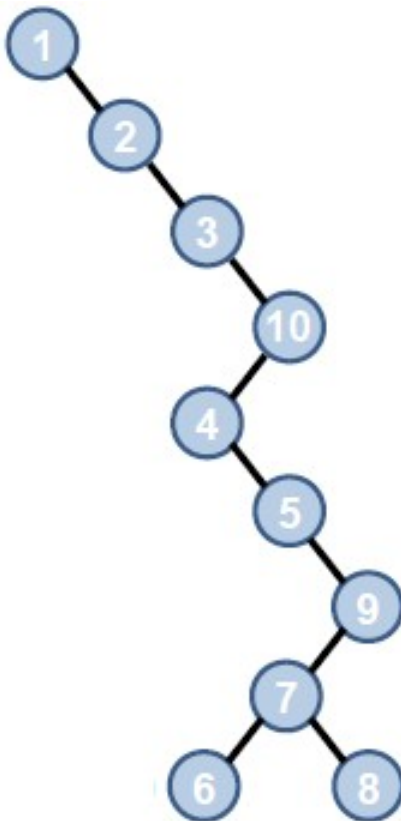
Árvores Balanceadas

Custo da inserção, busca e remoção em uma árvore binária:

Balanceada: $O(\log N)$

Não balanceada: $O(N)$

N corresponde ao número de nós na árvore



Árvores Balanceadas

Solução para o problema do balanceamento?

Modificar as operações de “inserção” e “remoção” da árvore

Exemplos de árvores balanceadas:

Árvore AVL

Árvore 2-3-4

Árvore Red-Black (também conhecida por vermelho-preto ou rubro-negra)

Árvore AVL

É um tipo de árvore binária balanceada

Criada pelos soviéticos* Adelson-Velskii e Landis em 1962

Permite rebalanceamento local

Apenas a parte afetada pela inserção ou remoção é rebalanceada

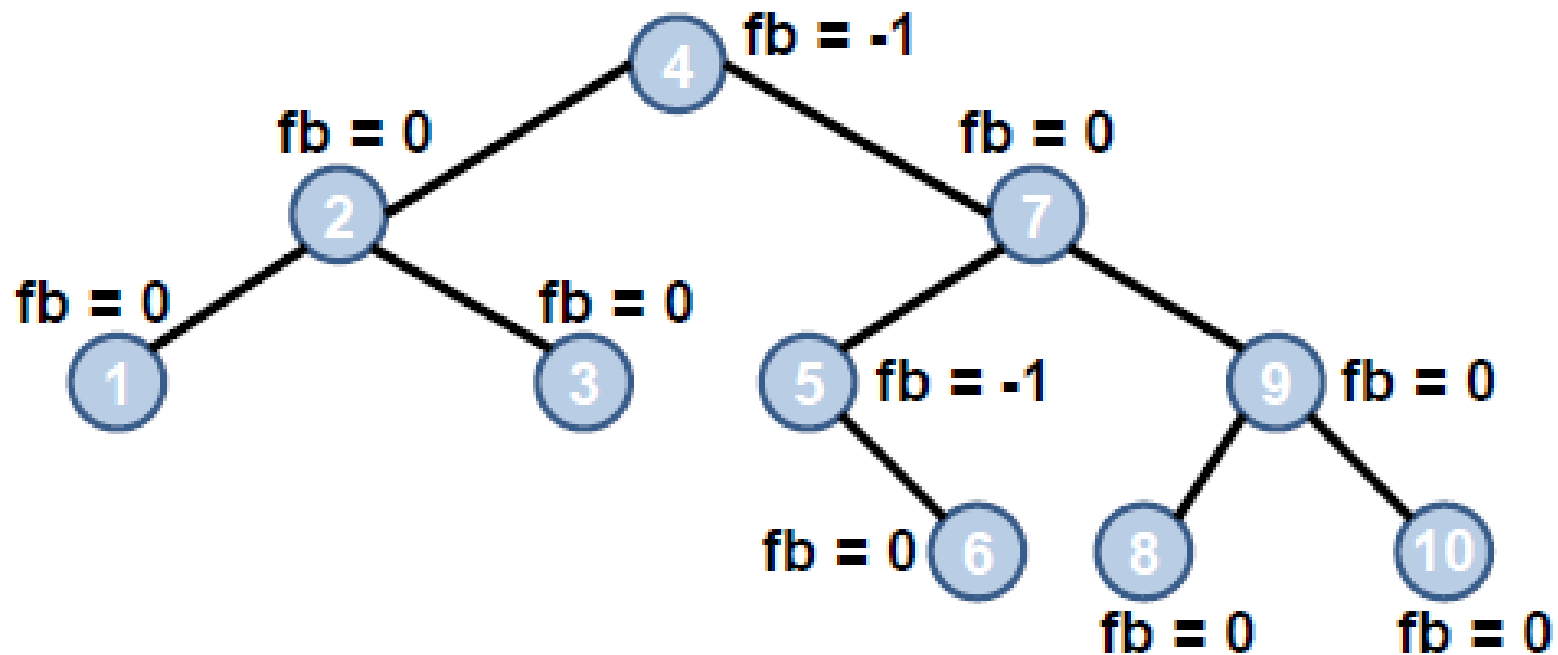
Uso de rotações simples ou duplas

*Georgy Adelson-Velsky e Yevgeniy Landis

Árvore AVL

A árvore AVL busca manter-se como uma **árvore binária quase completa**

Custo de qualquer algoritmo é no máximo $O(\log N)$



Árvore AVL: fator de balanceamento

Fator de balanceamento ou fb

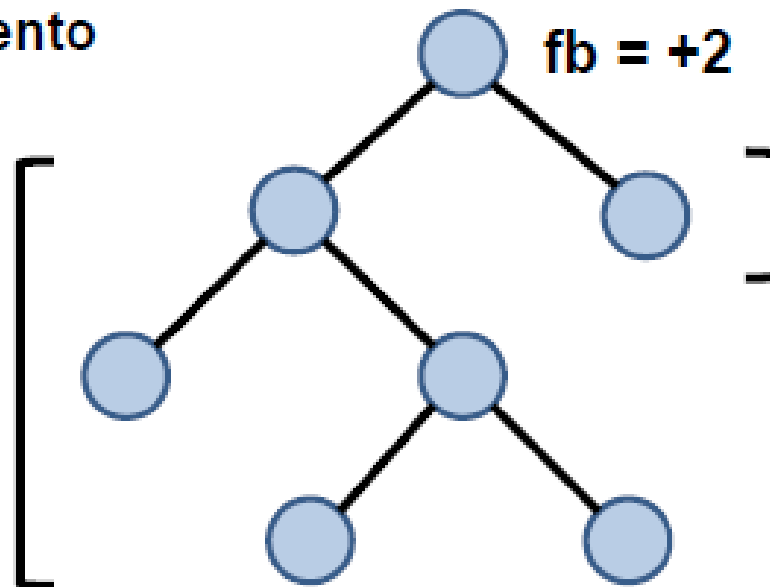
Diferença nas alturas das subárvores esquerda (sae) e direita (sad)

Se uma das subárvores não existir sua altura será -1

Fator de Balanceamento

$$FB = AE - AD$$

AE
Altura da
sub-árvore
ESQUERDA



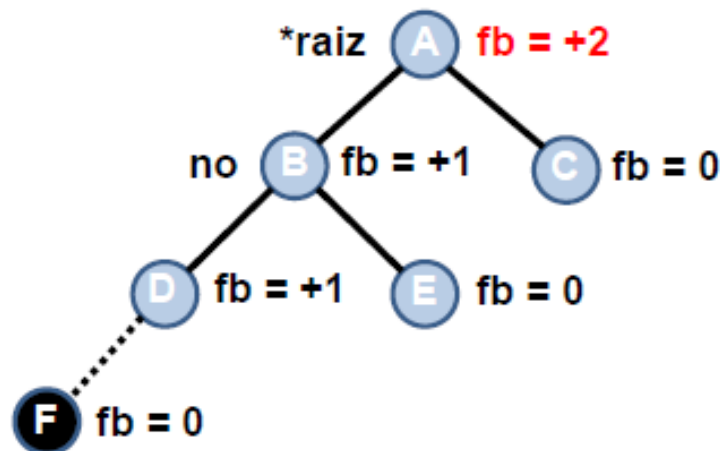
AD
Altura da
sub-árvore
DIREITA

Árvore AVL: fator de balanceamento

Fator de balanceamento é usado no balanceamento da árvore

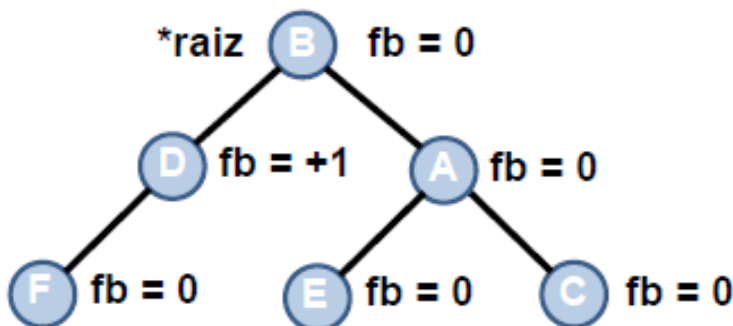
Numa AVL, **fb** deve ser **+1**, **0** ou **-1**.

Se **fb** < -1 ou **fb** > +1, então a árvore deve ser balanceada



Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.



Árvore Balanceada

Árvore AVL: fator de balanceamento

Seja um nó n qualquer da árvore:

$$FB(n) = altura(sae) - altura(sad)$$

se $FB(n) = 0$, as duas subárvores têm a mesma altura

se $FB(n) = +1$, a subárvore esquerda é mais alta que a direita em 1 unidade

se $FB(n) = -1$, a subárvore direita é mais alta que a esquerda em 1 unidade

Árvore AVL: implementação

Implementação é idêntica à da **Árvore Binária**

Para guardar o primeiro nó da árvore, utilizamos um **ponteiro para ponteiro**

Um **ponteiro para ponteiro** pode guardar o endereço de um **ponteiro**

Assim, fica fácil mudar a **raiz** da árvore (se necessário)

Árvore AVL: implementação

“ArvoreAVL.h”: define:

- Os protótipos das funções.

- O tipo de dado armazenado na árvore.

- O ponteiro “*árvore*”.

“ArvoreAVL.c”:

- Define o tipo de dados “*árvore*”.

- Implementa as suas funções.

Com exceção da *inserção* e da *remoção* as demais funções da *Árvore AVL* são idênticas às da *Árvore Binária*

Árvore AVL: implementação

```
// Arquivo ArvoreAVL.h
typedef struct NO *ArvAVL;

// Arquivo ArvoreAVL.c
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreAVL.h" // Esse arquivo contém os protótipos
struct NO {
    int info;
    int alt; // altura daquela subárvore
    struct NO *esq;
    struct NO *dir;
};

// Programa principal
ArvAVL *raiz; // Ponteiro para ponteiro
```

Árvore AVL: implementação

// Funções auxiliares

// Calcula a altura de um nó

```
int alt_NO (struct NO *no)
{
    if (no == NULL)
        return -1;
    else
        return no->alt;
}
```

// Calcula o fator de balanceamento de um nó

```
int fatorBalanceamento_NO(struct NO *no)
{
    return labs(alt_NO(no->esq) - alt_NO(no->dir));
}
```

// Calcula o maior valor

```
int maior(int x, int y)
{
    return (x > y ? x : y);
}
```

Árvore AVL: Operação de Rotação

Operação básica para balanceamento da AVL

Existem dois tipos de rotações:

Simples:

O nó desbalanceado e seu filho estão no mesmo sentido da inclinação

Dupla:

O nó está desbalanceado e seu filho estão inclinado no sentido inverso ao pai.

Equivale a duas rotações simples

Existem duas rotações simples e duas rotações duplas:

Rotação à direita e rotação à esquerda

Árvore AVL: Rotação RR

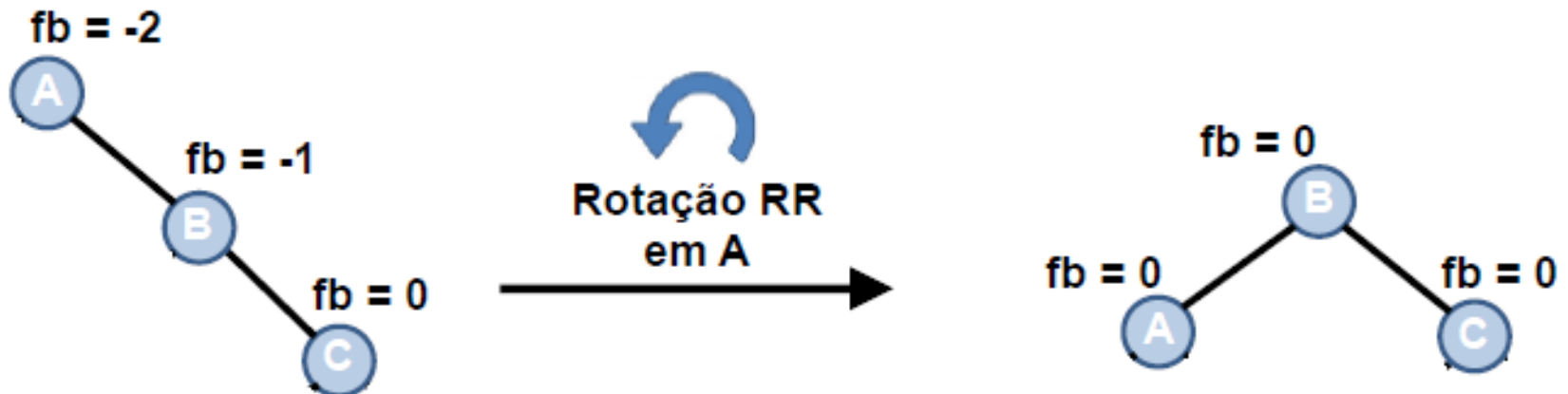
Ocorre quando um nó é inserido à direita (**R**ight) da subárvore direita (**R**ight).

Requer rotação simples à esquerda

Exemplo:

O nó **C** é inserido à direita da subárvore direita do nó **A**

O nó intermediário **B** deve ser escolhido para ser a raiz da árvore resultante



Árvore AVL: Rotação LL

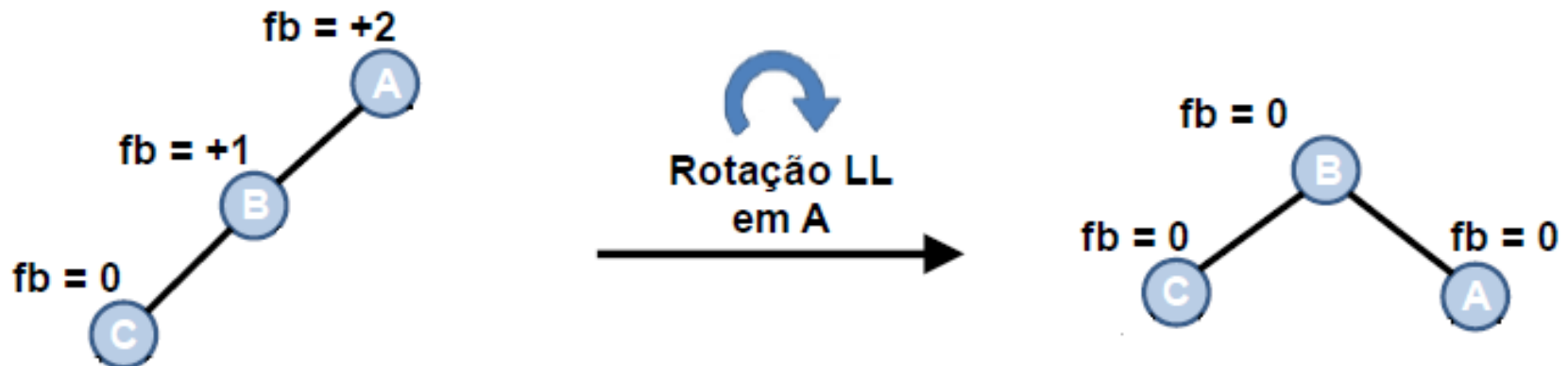
Ocorre quando um nó é inserido à esquerda (Left) da subárvore esquerda (Left).

Requer rotação simples à direita

Exemplo:

O nó **C** é inserido à esquerda da subárvore esquerda do nó **A**

O nó intermediário **B** deve ser escolhido para ser a raiz da árvore resultante



Árvore AVL: Rotação RL

Ocorre quando um nó é inserido à esquerda (Left) da subárvore direita (Right).

Requer rotação dupla à esquerda

Exemplo:

O nó C é inserido à esquerda da subárvore direita do nó A

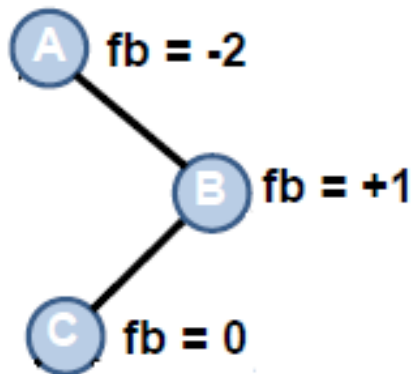
É necessário fazer uma rotação dupla, de modo que o nó C se torne pai dos nós A (filho da esquerda) e B (filho da direita)


Rotação LL em B

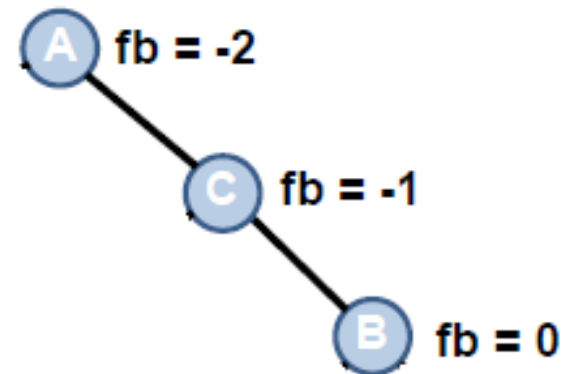
Rotação RR em A

Árvore AVL: Rotação RL

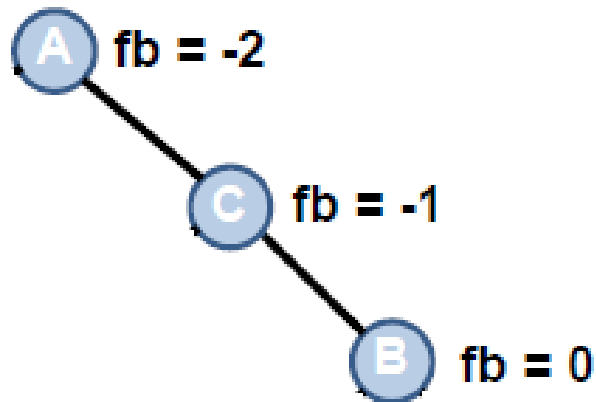
1ª) Rotação LL em B



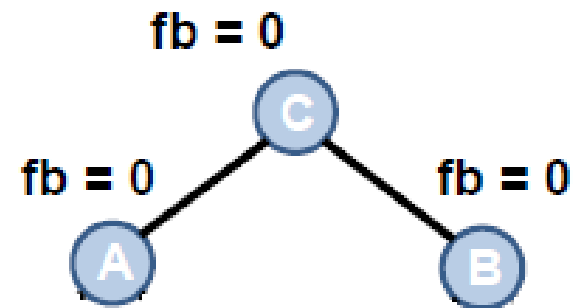

Rotação LL
em B



2ª) Rotação RR em A




Rotação RR
em A



Árvore AVL: Rotação LR

Ocorre quando um nó é inserido à direita (**R**ight) da subárvore esquerda (**L**eft).

Requer rotação dupla à direita

Exemplo:

O nó **C** é inserido à direita da subárvore esquerda do nó **A**

É necessário fazer uma rotação dupla, de modo que o nó **C** se torne pai dos nós **B** (filho da direita) e **A** (filho da esquerda)

Rotação **RR** em **B**

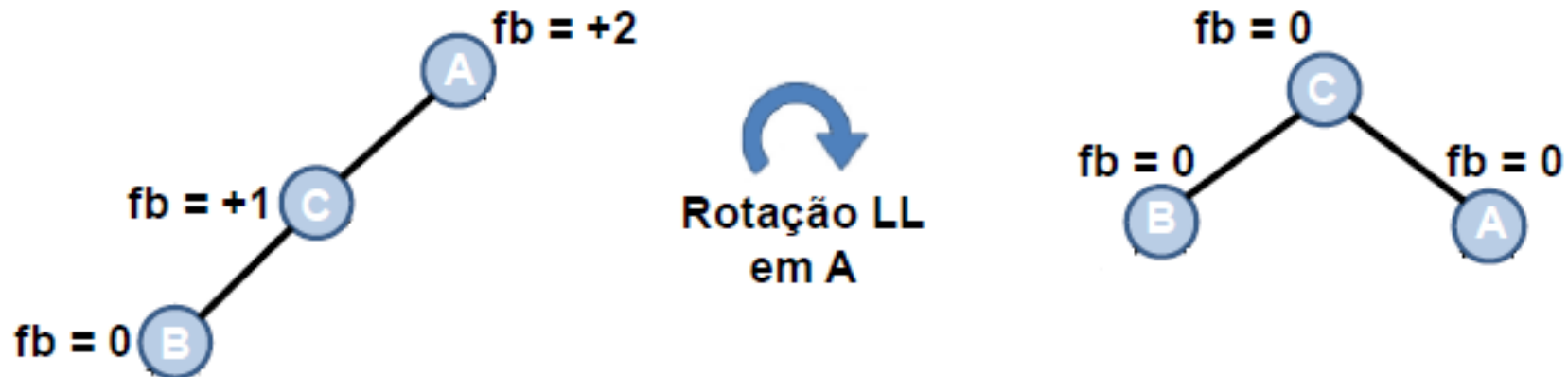
Rotação **LL** em **A**

Árvore AVL: Rotação LR

1ª) Rotação RR em B



2ª) Rotação LL em A



Árvore AVL: Quando usar cada Rotação

Considerando que o nó **C** foi inserido como filho do nó **B** e que **B** é filho do nó **A**, se o fator de balanceamento for

$A = +2$ e $B = +1$: **Rotação LL**

$A = -2$ e $B = -1$: **Rotação RR**

$A = +2$ e $B = -1$: **Rotação LR**

$A = -2$ e $B = +1$: **Rotação RL**

As rotações LL e RR são simétricas entre si assim como o são as rotações LR e RL.

Sinais iguais, rotação simples; sinais diferentes, rotação dupla

Sinal negativo, rotaciona-se para a direita; caso contrário, para a esquerda

Árvore AVL: Implementando as Rotações

As rotações são aplicadas no ancestral mais próximo do nó inserido cujo fator de balanceamento passa a ser +2 ou -2

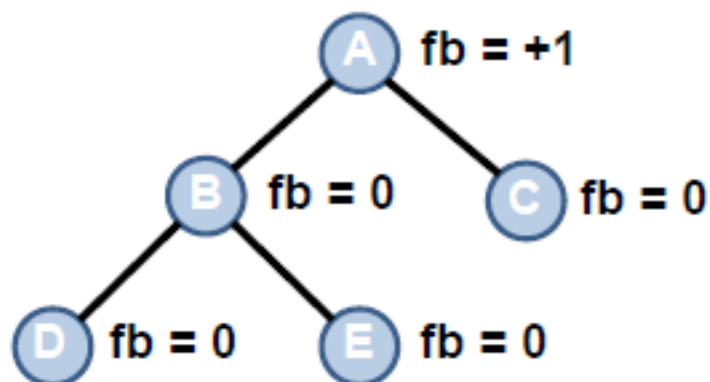
Esse é o parâmetro das funções implementadas

As rotações **simples** (LL e RR) atualizam as novas alturas das subárvores

As rotações **duplas** (LR e RL) podem ser implementadas com 2 (duas) rotações simples

Árvore AVL: Implementação da Rotação LL

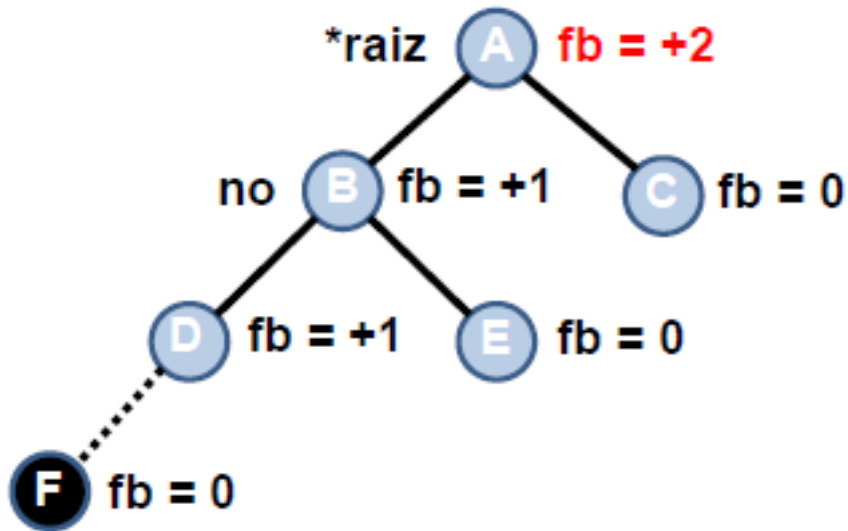
```
1
2 void RotacaoLL(ArvAVL *raiz) {
3     struct NO *no;
4     no = (*raiz)->esq;
5     (*raiz)->esq = no->dir;
6     no->dir = *raiz;
7     (*raiz)->altura = maior(altura_NO((*raiz)->esq),
8                             altura_NO((*raiz)->dir))
9                             + 1;
10    no->altura = maior(altura_NO(no->esq),
11                      (*raiz)->altura) + 1;
12    *raiz = no;
13 }
```



Árvore AVL e fator de balanceamento de cada nó

Árvore AVL: Implementação da Rotação LL

Passo a passo

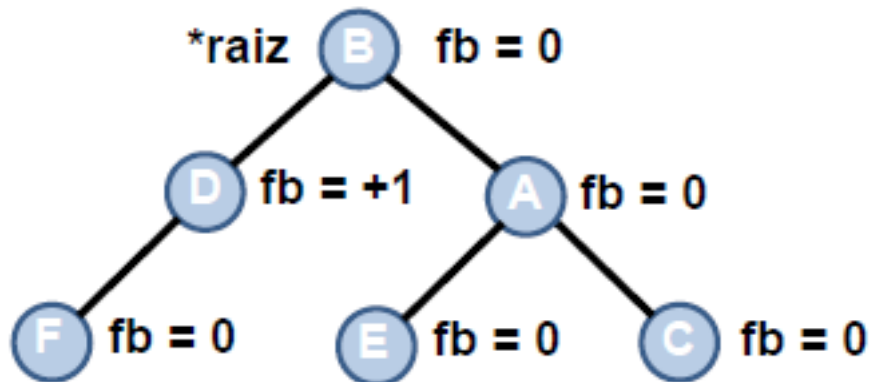


Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação LL no nó A

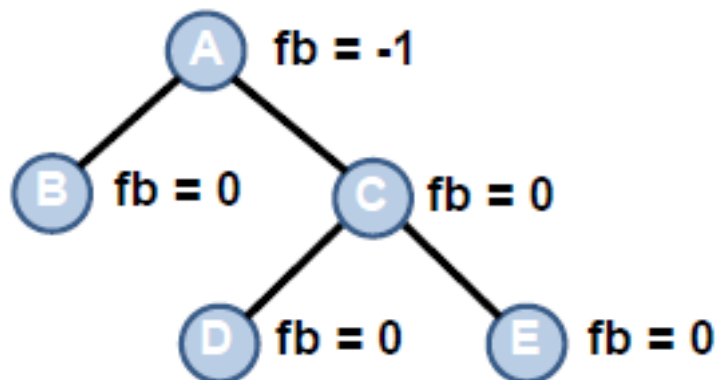
```
no = (*raiz)->esq;  
(*raiz)->esq = no->dir;  
no->dir = *raiz;  
*raiz = no;
```



Árvore Balanceada

Árvore AVL: Implementação da Rotação RR

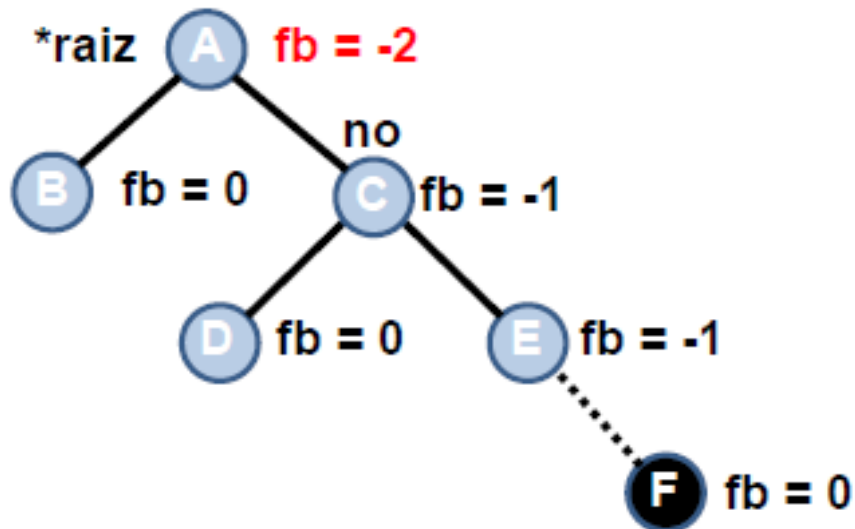
```
1
2 void RotacaoRR(ArvAVL *raiz) {
3     struct NO *no;
4     no = (*raiz)->dir;
5     (*raiz)->dir = no->esq;
6     no->esq = (*raiz);
7     (*raiz)->altura = maior(altura_NO((*raiz)->esq),
8                             altura_NO((*raiz)->dir))
9                             + 1;
10    no->altura = maior(altura_NO(no->dir),
11                      (*raiz)->altura) + 1;
12    (*raiz) = no;
13 }
```



Árvore AVL e fator de
balanceamento de cada nó

Árvore AVL: Implementação da Rotação RR

Passo a passo

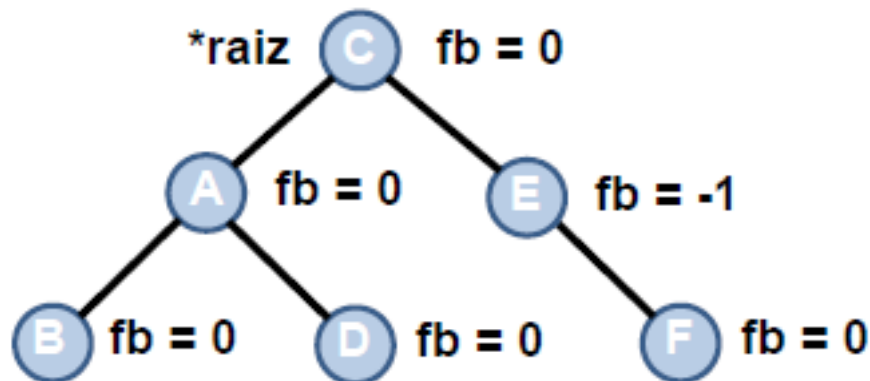


Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação RR no nó A

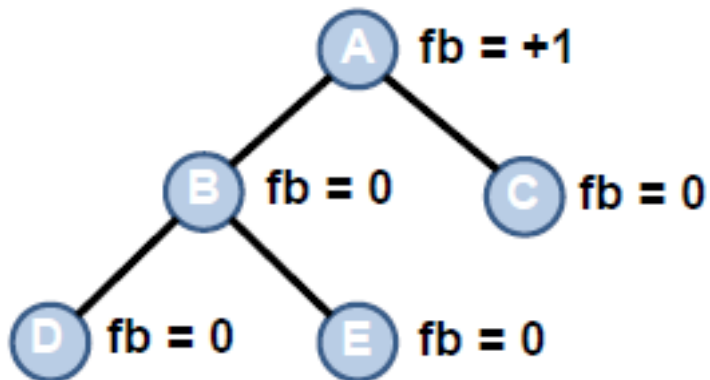
```
no = (*raiz)->dir;
(*raiz)->dir = no->esq;
no->esq = (*raiz);
(*raiz) = no;
```



Árvore Balanceada

Árvore AVL: Implementação da Rotação LR

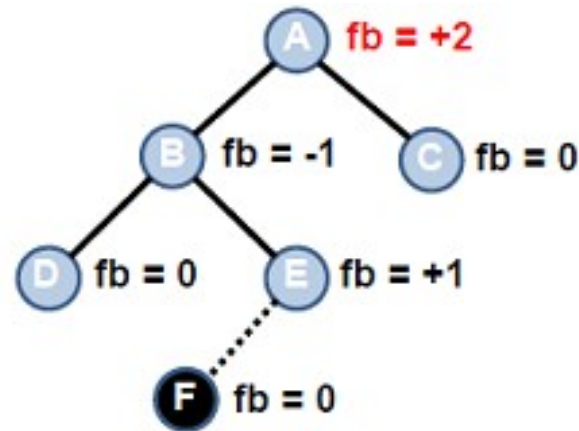
```
1  
2 void RotacaoLR (ArvAVL *raiz) {  
3     RotacaoRR (&(*raiz)->esq);  
4     RotacaoLL (raiz);  
5 }
```



Árvore AVL e fator de balanceamento de cada nó

Árvore AVL: Implementação da Rotação LR

Passo a passo

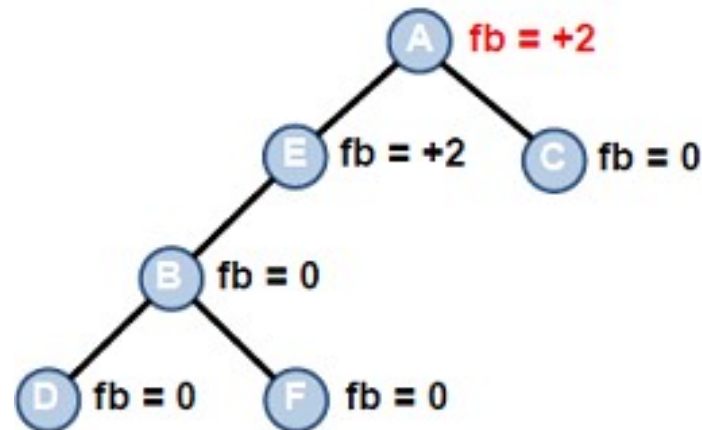


Inserção do nó F na árvore

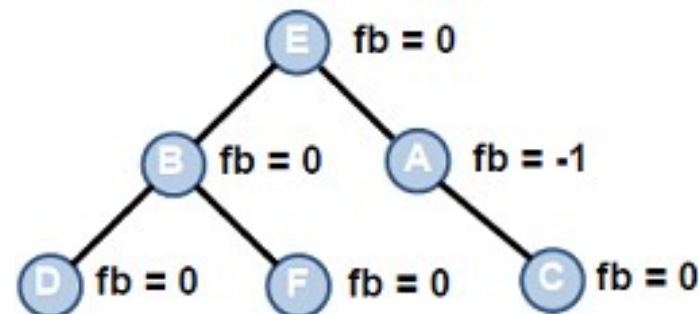
Árvore fica desbalanceada no nó A.

Aplicar Rotação LR no nó A. Isso equivale a:

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



Árvore após aplicar a Rotação RR no nó B

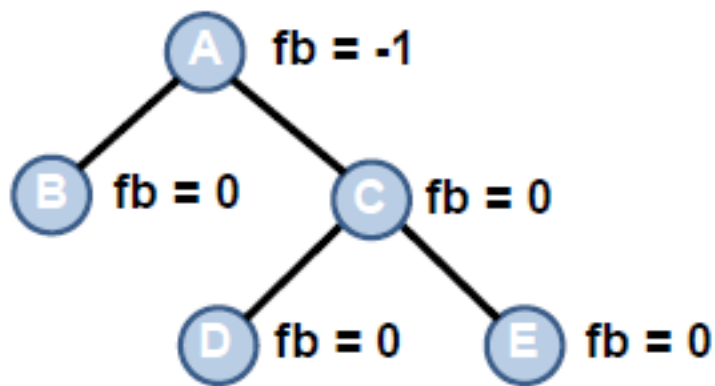


Árvore após aplicar a Rotação LL no nó A

Árvore Balanceada

Árvore AVL: Implementação da Rotação RL

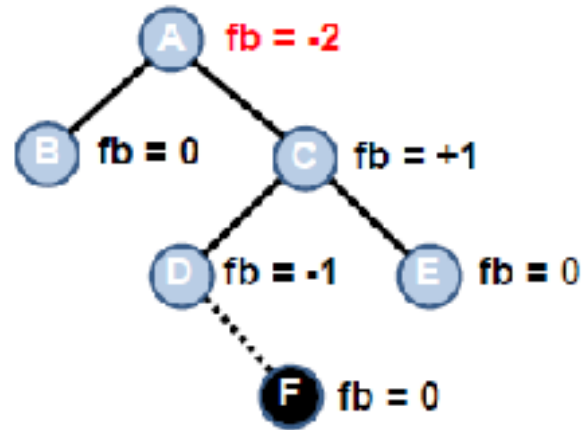
```
1  
2 void RotacaoRL(ArvAVL *raiz) {  
3     RotacaoLL(&(*raiz)->dir);  
4     RotacaoRR(raiz);  
5 }
```



Árvore AVL e fator de balanceamento de cada nó

Árvore AVL: Implementação da Rotação RL

Passo a passo

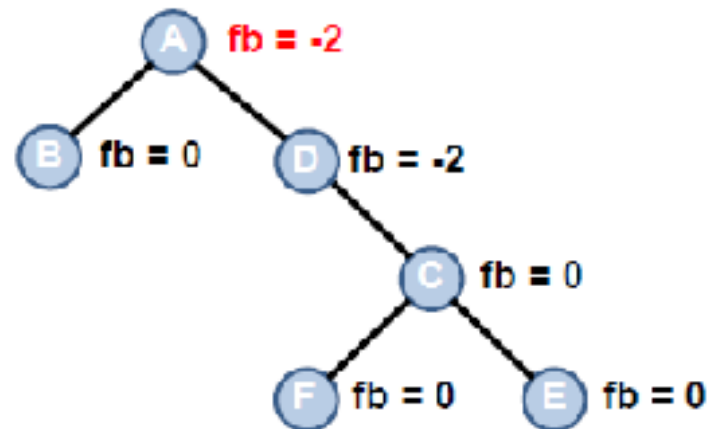


Inserção do nó F na árvore

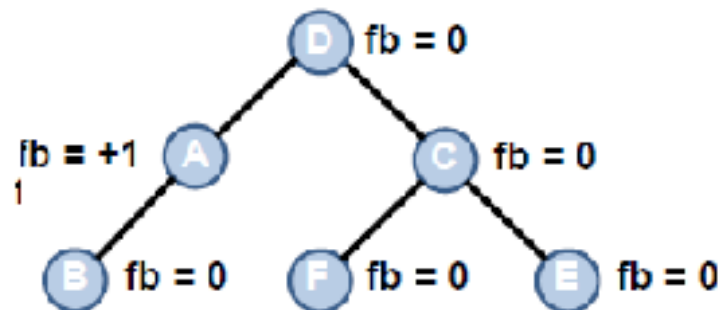
Árvore fica desbalanceada no nó A.

Aplicar Rotação RL no nó A. Isso equivale a:

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



Árvore após aplicar a Rotação LL no nó C



Árvore após aplicar a Rotação RR no nó A

Árvore Balanceada

Árvore AVL: Operação de Inserção

Para inserir um valor **V** na árvore:

Se raiz é igual a **NULL**, insira o nó

Se **V** é menor do que a raiz, vá para a subárvore **esquerda**

se **V** é maior do que a raiz, vá para a subárvore **direita**

Aplique o método **recursivamente**

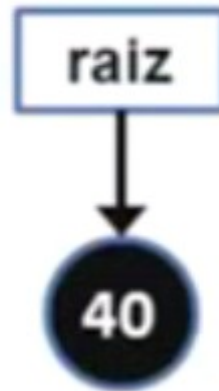
Ao voltar na recursão, recalcule as alturas de cada subárvore

Aplique a rotação necessária se o fator de balanceamento for **+2**
ou **-2**

Árvore AVL: Operação de Inserção

Também existe o caso onde a inserção é feita em uma **árvore AVL** que está vazia

Inserir em uma
árvore vazia



*raiz = novo;

Árvore AVL: Operação de Inserção

// Programa principal

```
int x = insere_ArvAVL(raiz, valor);
```

// Arquivo ArvoreAVL.h

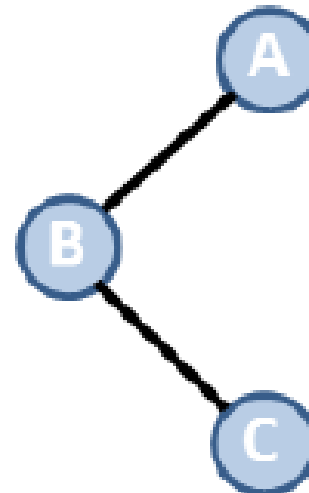
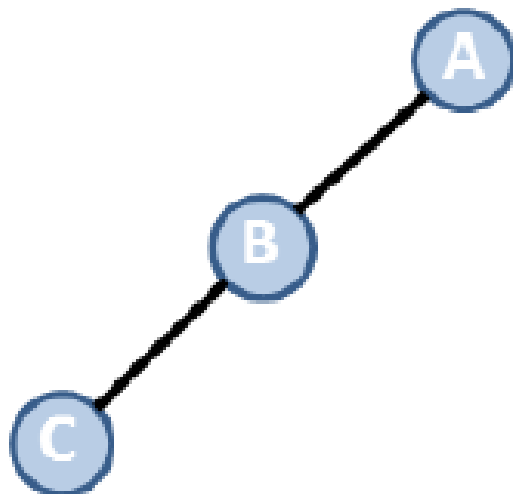
```
int insere_ArvAVL(ArvAVL *raiz, int valor);
```

// Arquivo ArvoreAVL.c

```
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...
```

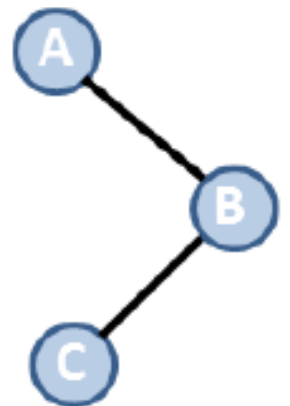
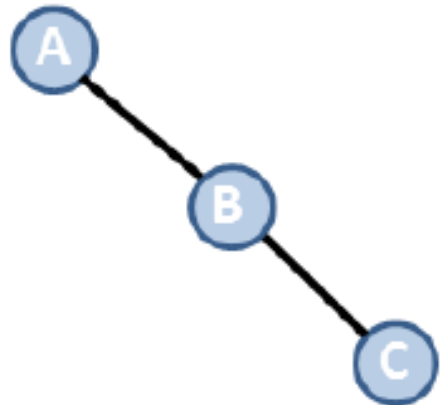
Árvore AVL: Operação de Inserção

```
//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: Operação de Inserção

```
//continuação
else{
    if(valor > atual->info){
        if((res=inserere_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;
}
```



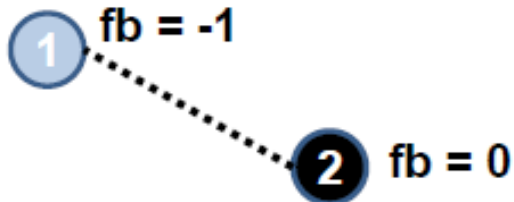
Árvore AVL: Operação de Inserção

Passo a passo

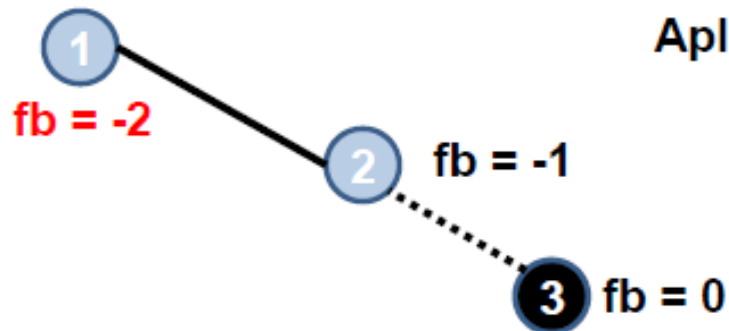
Inserir valor: 1



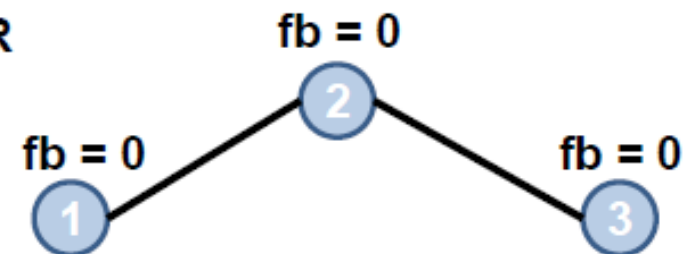
Inserir valor: 2



Inserir valor: 3



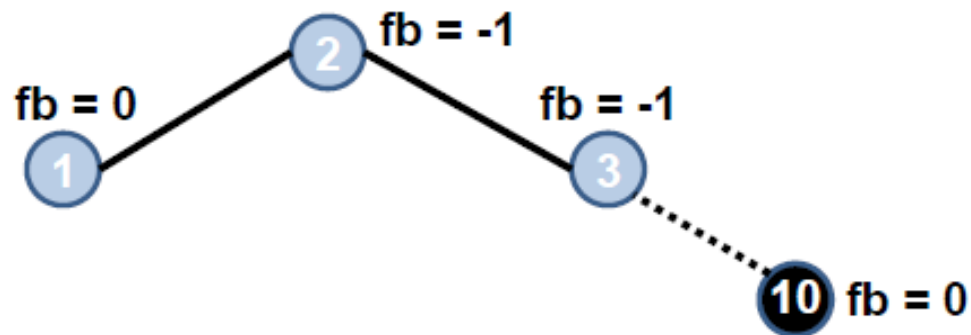
Nó "1" desbalanceado
Aplicar Rotação RR



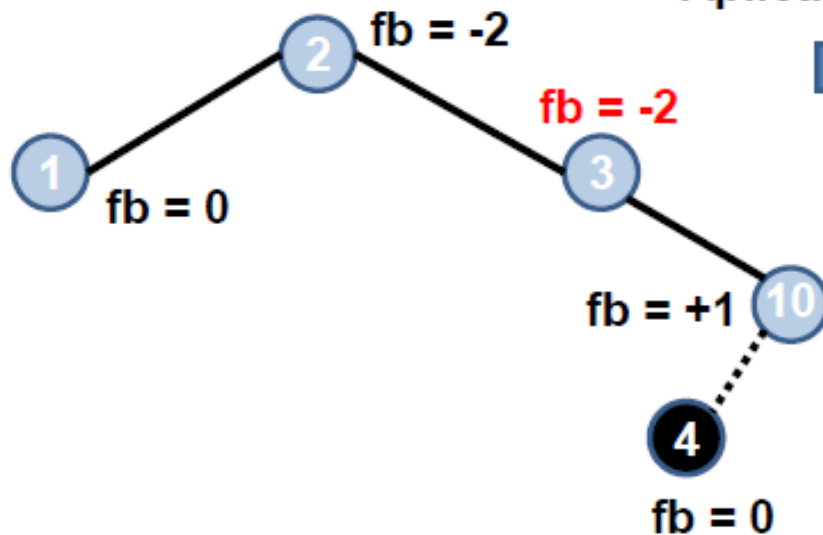
Árvore AVL: Operação de Inserção

Passo a passo

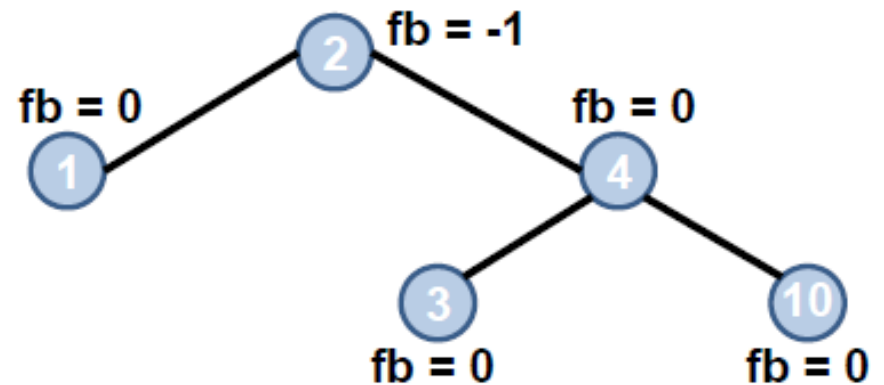
Inserir valor: 10



Inserir valor: 4



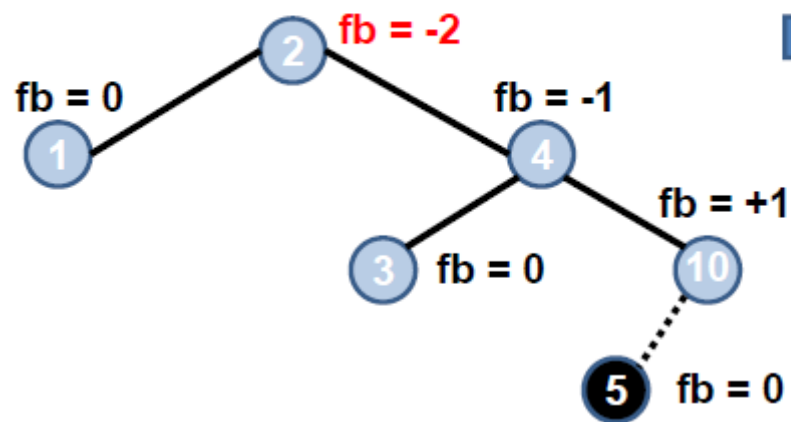
Nó "3" desbalanceado
Aplicar Rotação RL



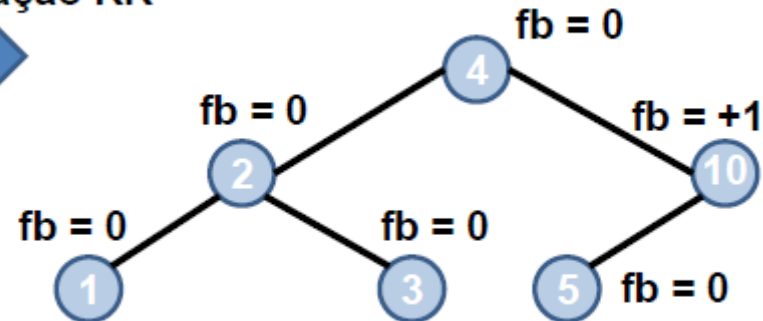
Árvore AVL: Operação de Inserção

Passo a passo

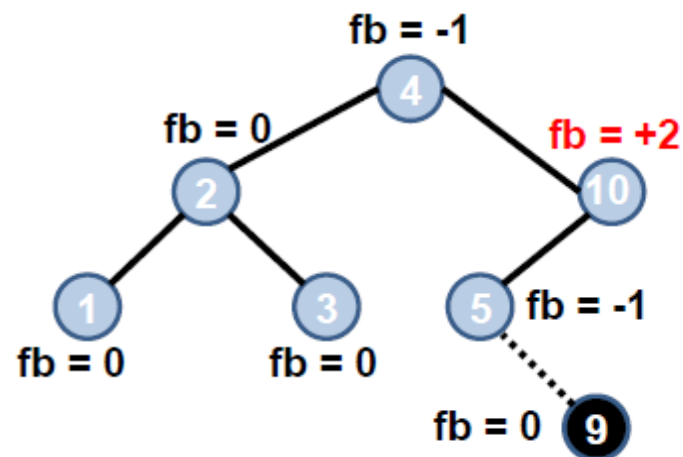
Inserir valor: 5



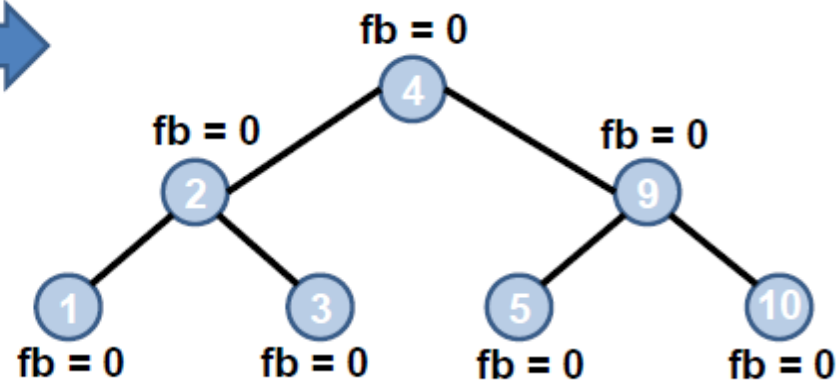
Nó "2" desbalanceado
Aplicar Rotação RR



Inserir valor: 9



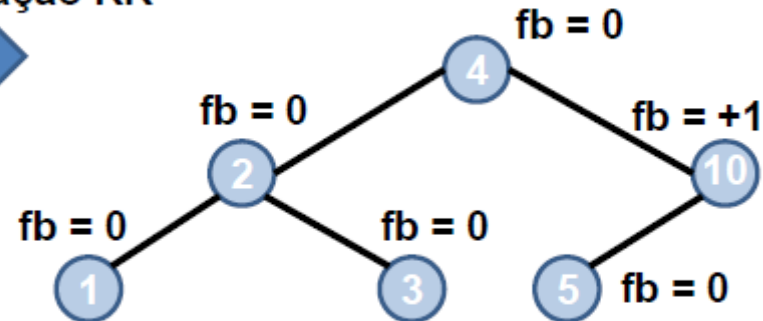
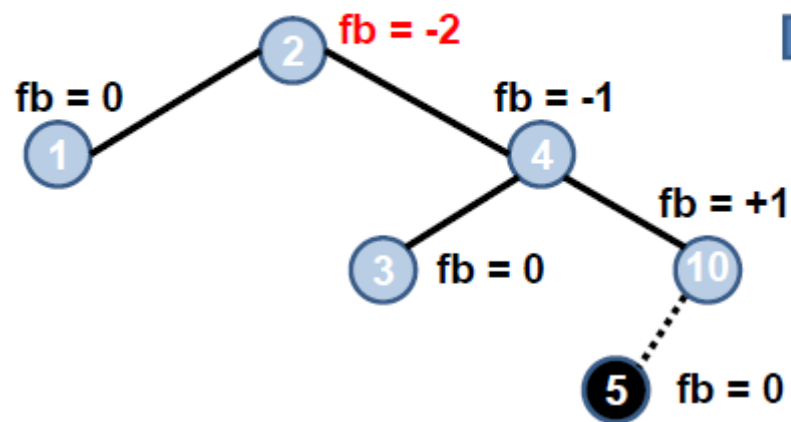
Nó "10" desbalanceado
Aplicar Rotação LR



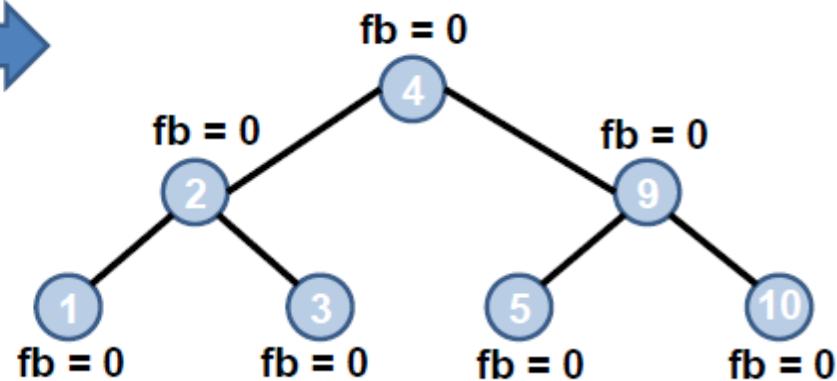
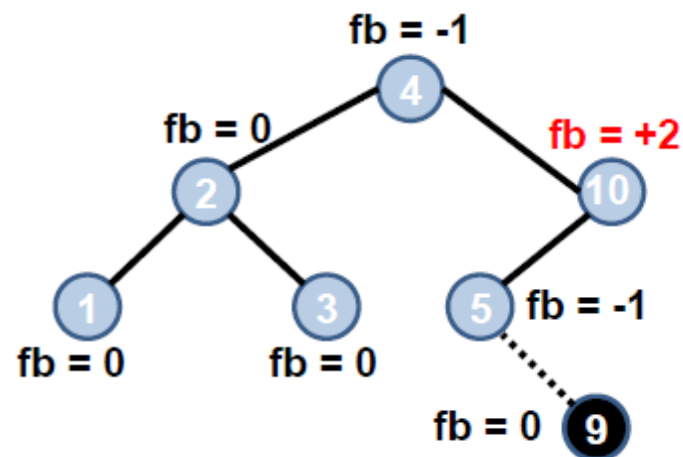
Árvore AVL: Operação de Inserção

Passo a passo

Inserir valor: 5



Inserir valor: 9



Árvore AVL: Operação de Remoção

Como na inserção, tem-se que percorrer 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

Os 3 (três) tipos de remoção trabalham juntos. A remoção sempre remove um elemento específico da árvore, o qual pode ser um nó folha, ter um ou dois filhos.

Cuidado:

- Não se pode remover nó de uma árvore vazia

- Removendo-se o último nó, a árvore fica vazia

Balanceamento:

- Valem as mesmas regras da inserção

- Remover um nó da subárvore da **direita** equivale a inserir um nó na subárvore da **esquerda**

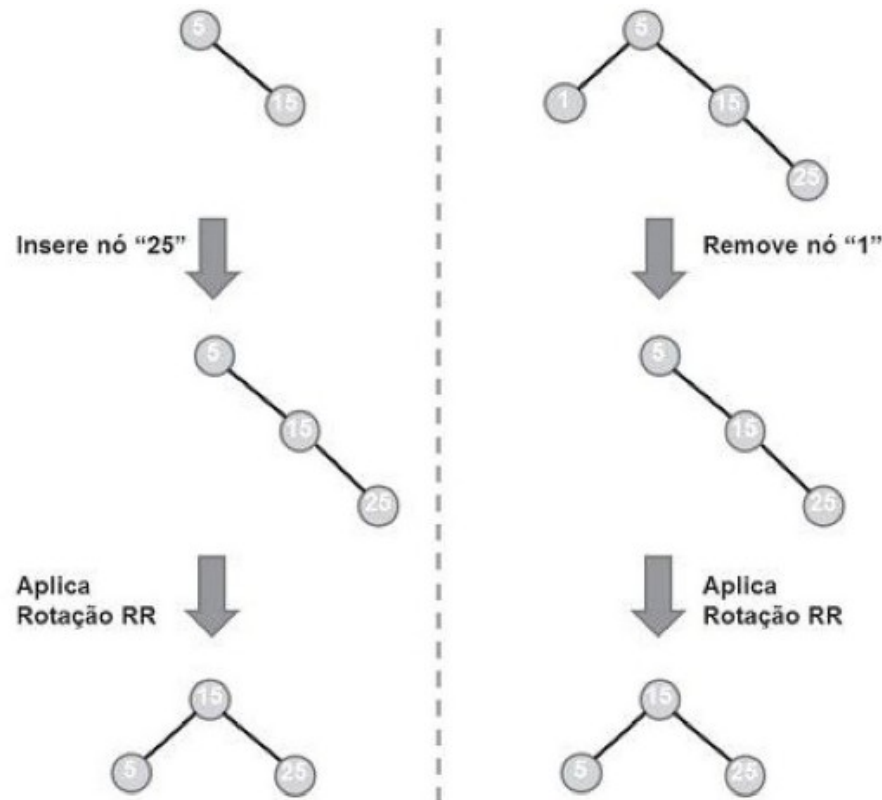
Árvore AVL: Operação de Remoção

Uma vez removido o nó

Deve-se 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 subárvore **direita** equivale a inserir um nó na subárvore da **esquerda**.



Árvore AVL: Operação de Remoção

A operação de remoção trabalha com duas funções:

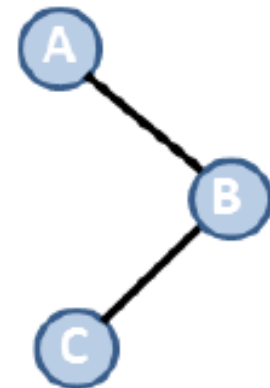
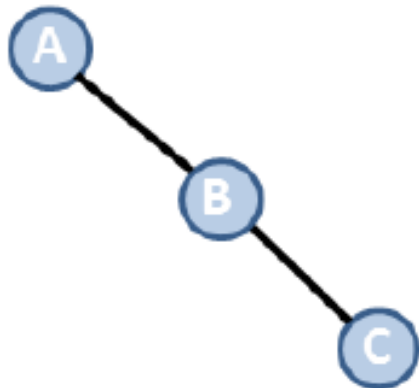
Busca pelo nó

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 }
```

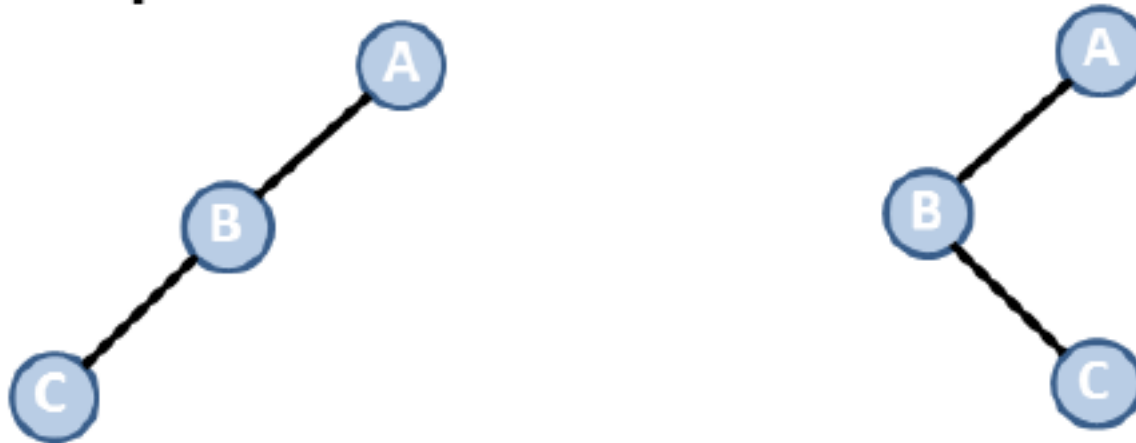
Árvore AVL: Operação de Remoção

```
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: Operação de 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: Operação de 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
```

Pai tem 2 filhos:
Substituir pelo nó
mais a esquerda
da subárvore da
direita

```
        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);  
        }  
    }  
}
```

Corrige a
altura

```
    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;  
}
```

Árvore AVL: Operação de 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;  
}
```

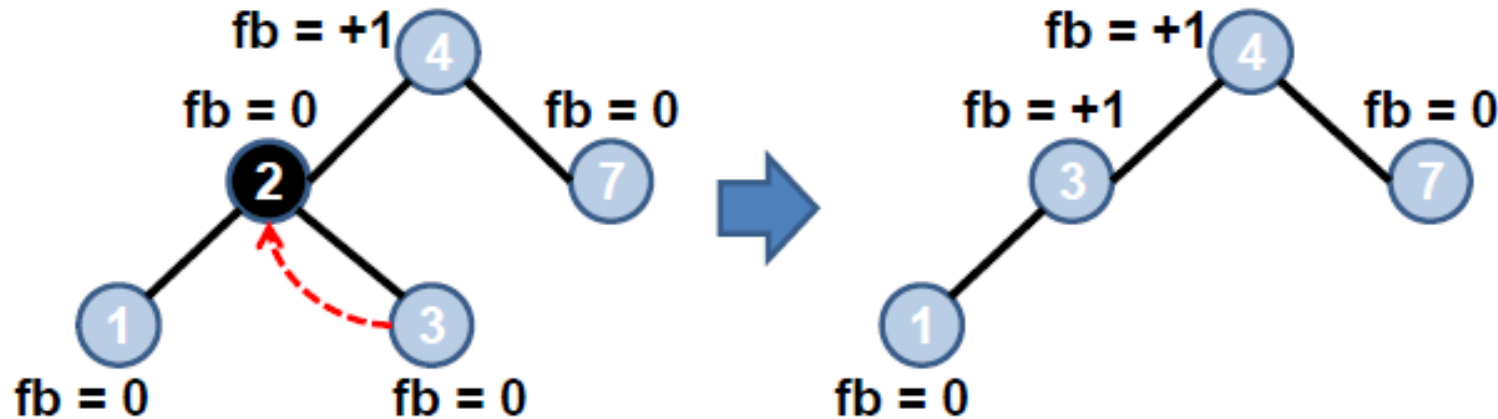


Procura pelo nó
mais a esquerda

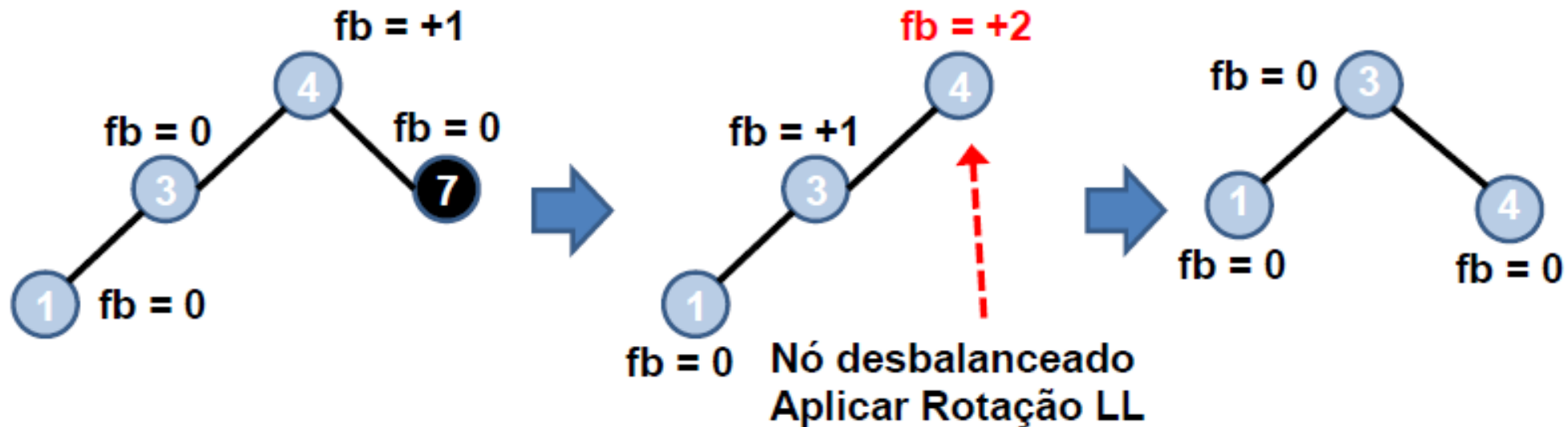
Árvore AVL: Operação de Remoção

Passo a passo

Remove valor: 2



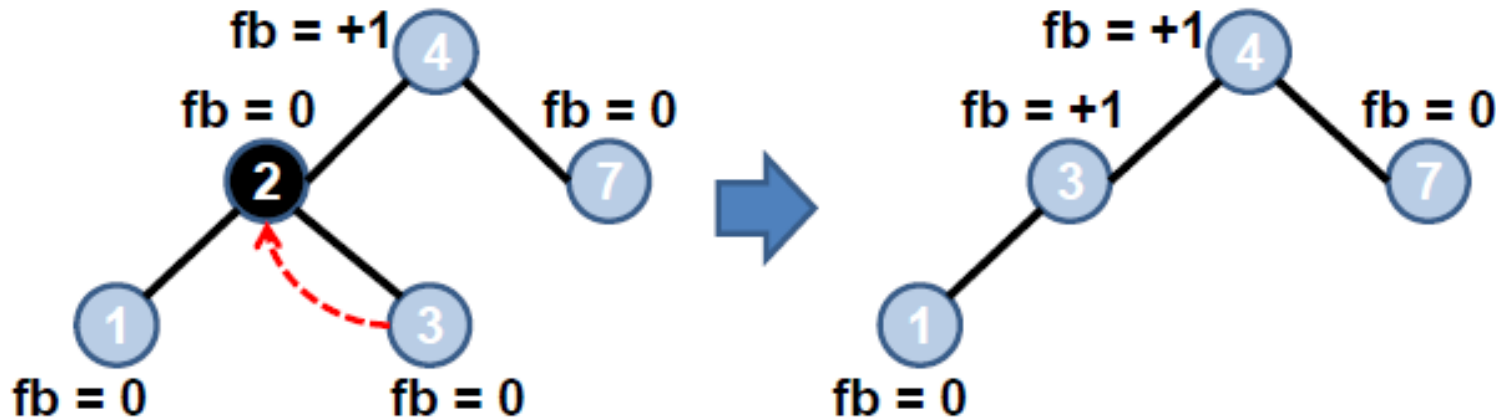
Remove valor: 7



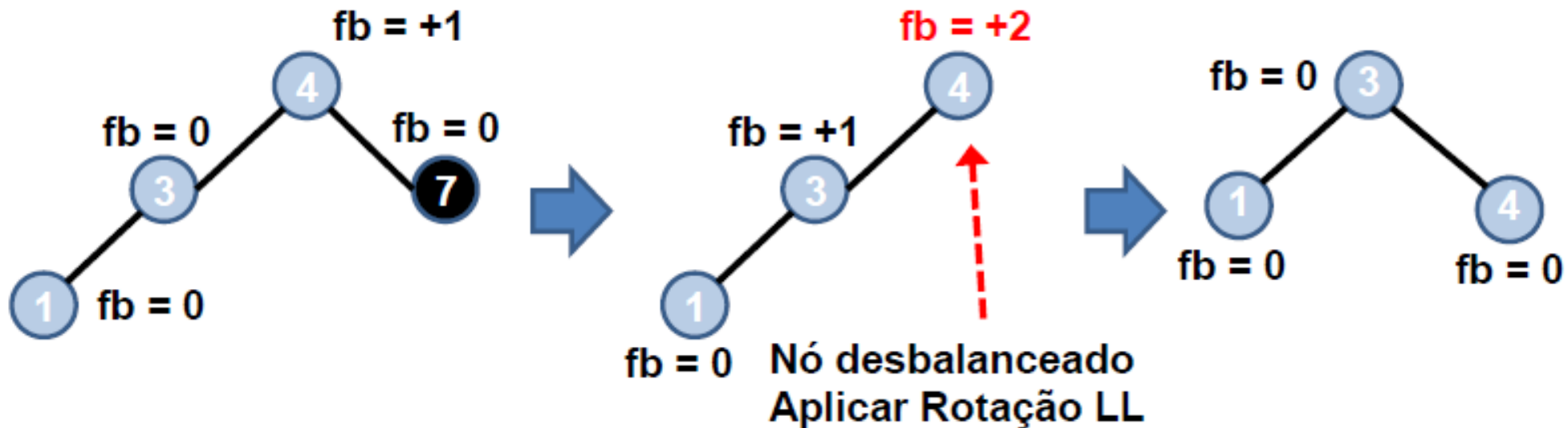
Árvore AVL: Operação de Remoção

Passo a passo

Remove valor: 2



Remove valor: 7



Referências

Backes, André. Slides de aulas: Árvore AVL