



# Estruturas de Dados 2

## aula 05 – árvore AVL – parte 2

Antonio Angelo de Souza Tartaglia  
angelot@ifsp.edu.br

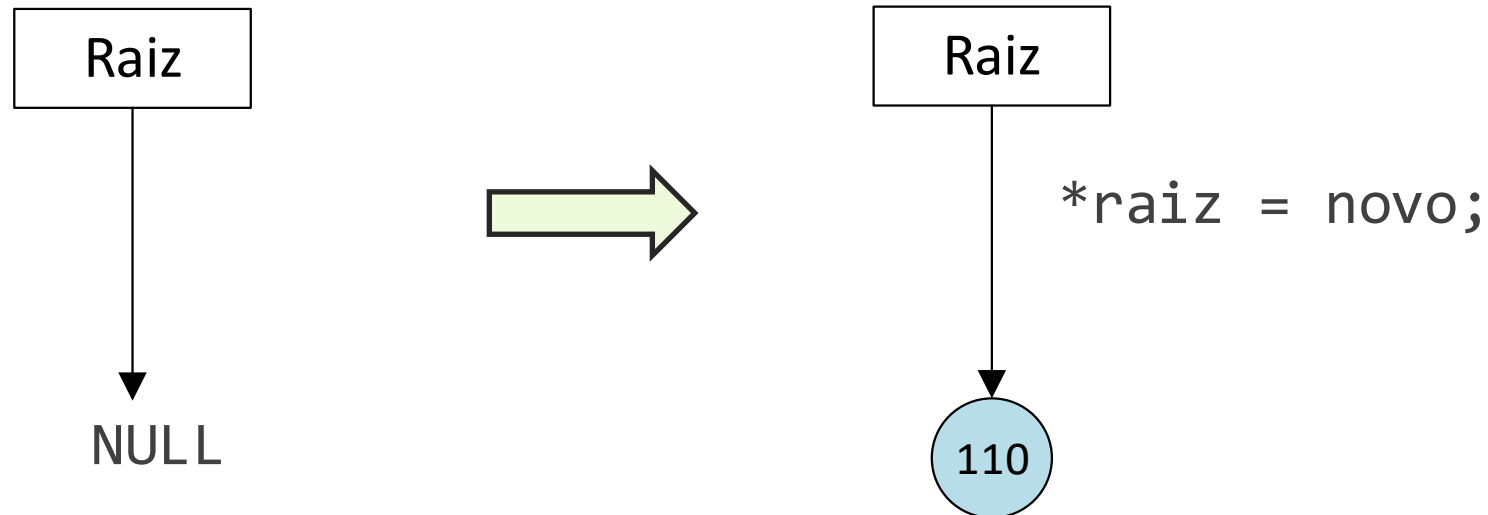
## Árvore AVL – Inserção em Árvore AVL



- Para inserir um novo nó em uma árvore AVL, basicamente o que tem que ser feito é alocar espaço para um novo nó, e procurar sua posição na árvore usando os seguintes passos:
  - Se a árvore está vazia e a raiz aponta para NULL : insira o nó como raiz da árvore;
  - Se a chave do nó é menor do que a da raiz: vá para sua sub-árvore esquerda;
  - Se a chave do nó é maior do que a da Raiz: vá para sua sub-árvore direita;
  - Aplique o método de inserção recursivamente, buscando a posição correta para o novo elemento.
- **Ao voltar da recursão, recalcule as alturas de cada sub-árvore;**
- Aplique a rotação necessária se o fator de balanceamento passar a ser +2 ou -2.

## Árvore AVL – Inserção em Árvore AVL

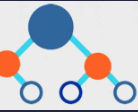
- Para o caso onde a inserção é feita em uma árvore AVL que está vazia, simplesmente se insere o novo elemento como a raiz da árvore.



# Estrutura de Dados 2

## Árvore AVL – Inserção em Árvore AVL

A inserção de um novo nó na Árvore AVL é exatamente igual à inserção na Árvore Binária. Porém uma vez inserido, começam a surgir as diferenças entre uma simples Árvore Binária de Busca e uma Árvore AVL.



```
//Arquivo arvoreAVL.h
int insere_arvAVL(arvAVL *raiz, int valor);

//arquivo main()
confirmaInsercao(insere_arvAVL(raiz, 160));
confirmaInsercao(insere_arvAVL(raiz, 150));
confirmaInsercao(insere_arvAVL(raiz, 100));
confirmaInsercao(insere_arvAVL(raiz, 110));
confirmaInsercao(insere_arvAVL(raiz, 130));
confirmaInsercao(insere_arvAVL(raiz, 140));
confirmaInsercao(insere_arvAVL(raiz, 120));
confirmaInsercao(insere_arvAVL(raiz, 170));
confirmaInsercao(insere_arvAVL(raiz, 180));
confirmaInsercao(insere_arvAVL(raiz, 190));
confirmaInsercao(insere_arvAVL(raiz, 200));
confirmaInsercao(insere_arvAVL(raiz, 200));

//arquivo arvoreAVL.c
void confirmaInsercao(int x) {
    if(x) {
        printf("Elemento inserido com sucesso.\n");
    } else {
        printf("Erro! Elemento nao inserido.\n");
    }
}
```

# Estrutura de Dados 2

## Árvore AVL – Inserção em Árvore AVL



Tentou-se inserir o elemento 200 duas vezes. Na primeira houve a inserção normalmente, porém, na segunda, a função de inserção detectou a duplicidade

Mensagem da função de inserção

```
"C:\Users\angelot\Documents\Aulas\EDA2\Aulas\05 - Árvore AVL\material apoio\arvoreA1"

Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento inserido com sucesso.
Elemento 200 ja existe. Insercao duplicada!
Erro! Elemento nao inserido.
```

Mensagens da função `confirmaInserção()`



## Árvore AVL – Inserção em Árvore AVL

A inserção do elemento na árvore é feita através de chamadas recursivas que descem pela árvore, procurando a posição correta, até chegar ao nó folha, quando então, é encontrada a posição a se inserir o elemento.

Este “if”, é o caso base da recursão: só será executado quando a chamada recursiva encontrar a posição correta onde inserir o novo nó.

```
//arquivo arvoreAVL.c
int insere_arvAVL(arvAVL *raiz, int valor){
    int res; // armazena resp. sucesso retorno das funções
    if(*raiz == NULL){ //arvore vazia ou chegou no nó folha
        struct NO *novo;
        novo = (struct NO*) malloc(sizeof(struct NO));
        if(novo == NULL){
            return 0;
        }
        novo->info = valor;
        novo->alt = 0;
        novo->esq = NULL;
        novo->dir = NULL;
        *raiz = novo;
        return 1;
    }
}
```

Encontrada a posição, preenche-se um novo nó com sua altura como 0, e seus dois ponteiros como NULL, porque um novo nó, sempre será inserido como uma folha.



## Árvore AVL – Inserção em Árvore AVL

Se inserção realizada foi com sucesso, e a resposta da chamada recursiva (que ficou armazenada em res), for igual a 1, é necessário testar o balanceamento

Se valor a ser inserido for menor do que campo "info" do Nó atual, a inserção tem que ser feita na esquerda.

Aqui é o passo recursivo.

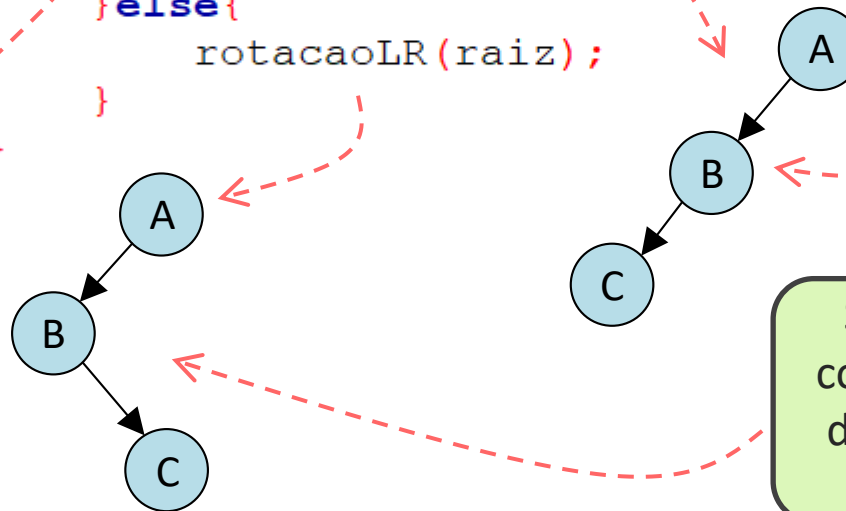
```
struct NO *atual = *raiz;  
if(valor < atual->info){  
    if((res = insere_arvAVL(&(atual->esq), valor)) == 1){  
        if(fatorBalanceamento_NO(atual) >= 2){  
            if(valor < (*raiz)->esq->info){  
                rotacaoLL(raiz);  
            }else{  
                rotacaoLR(raiz);  
            }  
        }  
    }  
}else{
```

Testado o fator de balanceamento, se este for maior ou igual a 2, a árvore precisa ser balanceada. Falta apenas saber para que lado o balanceamento terá que ocorrer

Se o valor é menor do que o conteúdo do filho da esquerda da raiz, que é o atual, então é uma inserção deste tipo.

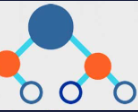
Se o valor é maior, do que o conteúdo do filho da esquerda da raiz, que é o atual, então é uma inserção deste tipo.

Trata valores maiores ou iguais



# Estrutura de Dados 2

## Árvore AVL – Inserção em Árvore AVL



Se inserção OK, chama o Balanceamento e calcula para o nó atual

Se o valor a inserir for maior do que o atual

Aqui é o passo recursivo.

Igual a etapa anterior, este if detecta em que direção a inserção ocorreu.

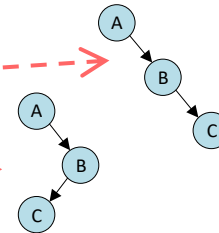
Recalcula a altura do nó atual: maior valor entre a altura dos filhos esquerdo e direito, somando de +1.

Se o valor a inserir não for menor nem maior, ele é igual. Não teremos valores repetidos...

Retorna se inserção OK.

```
}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("Elemento %d ja existe. Insercao duplicada!\n", valor);
        return 0;
    }
}

atual->alt = maior(alt_no(atual->esq), alt_no(atual->dir)) + 1;
return res;
```





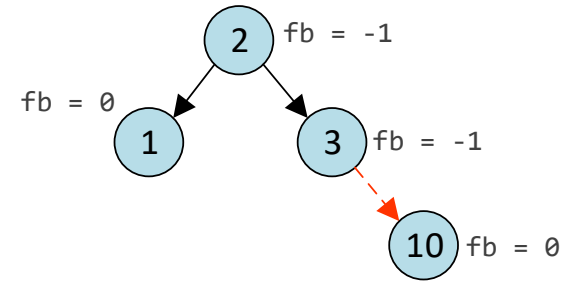
# Estrutura de Dados 2



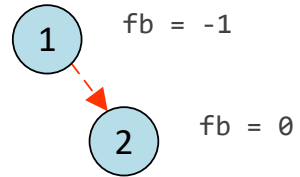
Insere valor 1



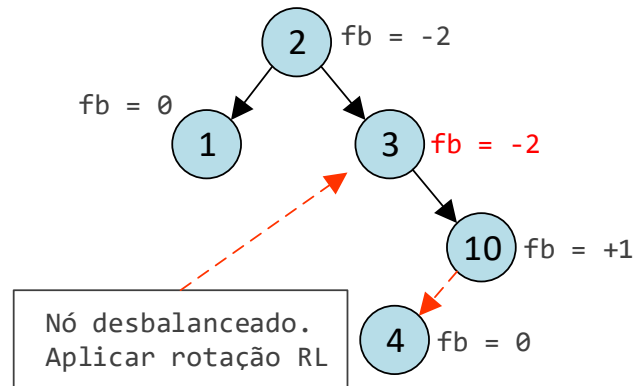
Insere valor 10



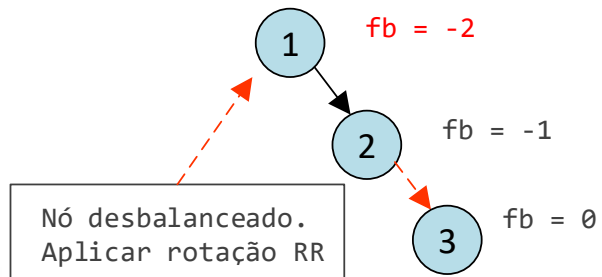
Insere valor 2



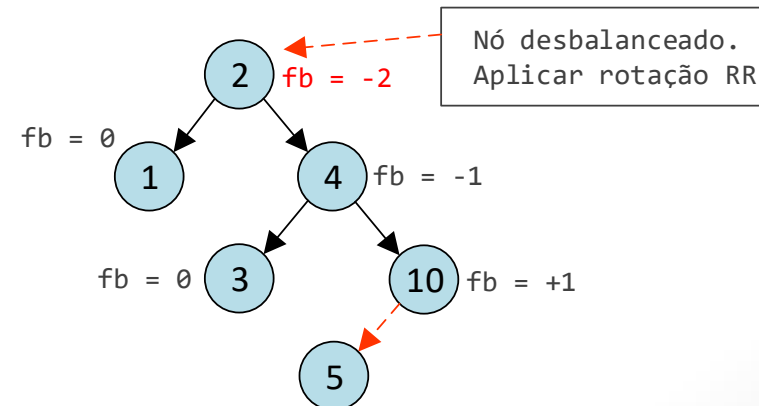
Insere valor 4



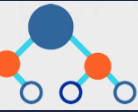
Insere valor 3



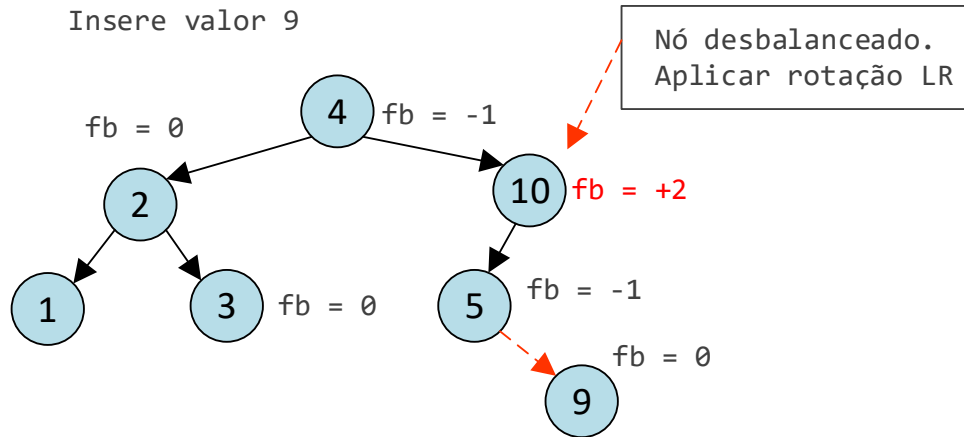
Insere valor 5



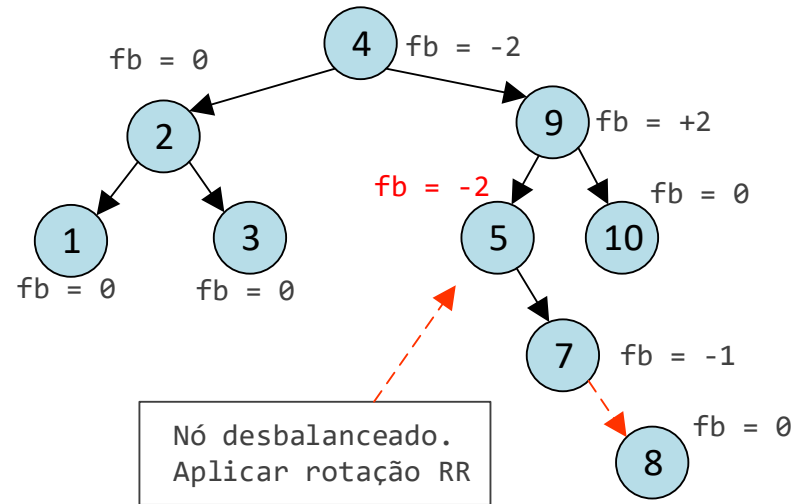
# Estrutura de Dados 2



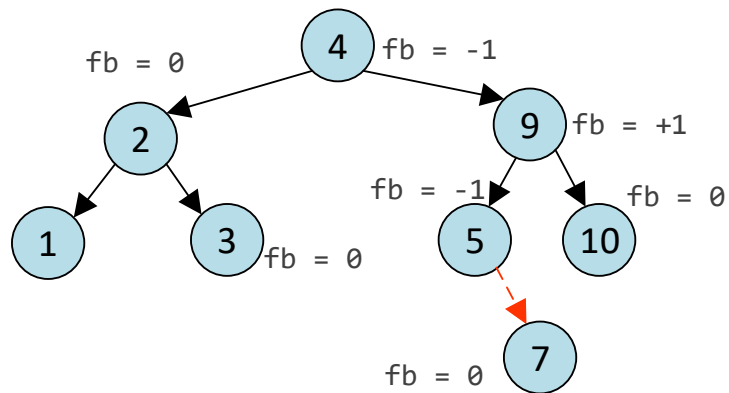
Inserir valor 9



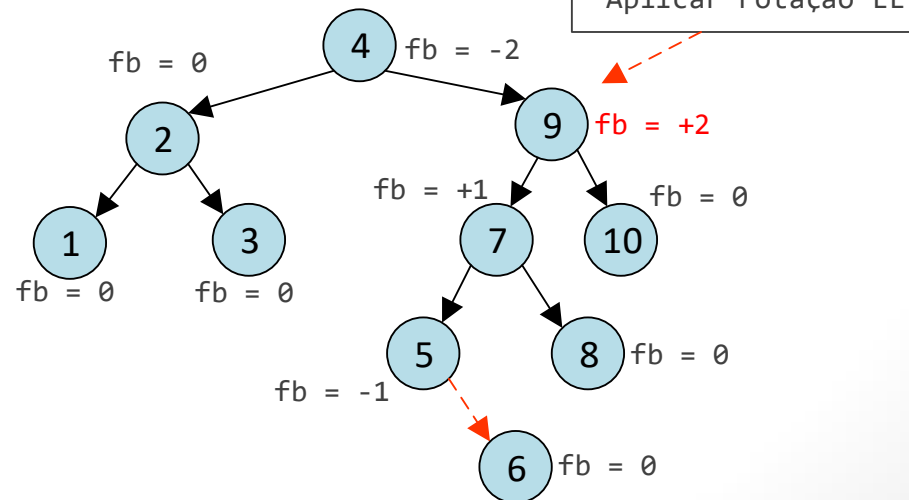
Inserir valor 8



Inserir valor 7



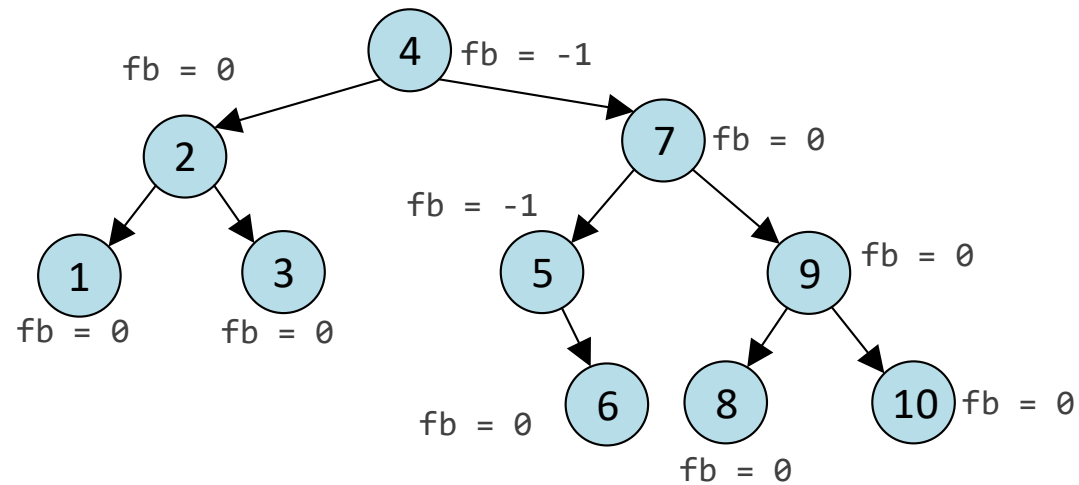
Inserir valor 8



## Árvore AVL – Inserção em Árvore AVL



Árvore Balanceada



## Árvore AVL – Remoção em Árvore AVL

- Existem 3 tipos de remoção:
  - Nó folha, sem filhos;
  - Nó com 1 filho;
  - Nó com 2 filhos.
- Os 3 tipos de remoção trabalham juntos. A remoção sempre remove um elemento específico da Árvore, o qual pode ser Nó folha, ter um ou dois filhos. Somente é possível ter essa informação no momento da remoção.



## Árvore AVL – Remoção em Árvore AVL




- Cuidado:

- Não se pode remover de uma Árvore vazia;
- Removendo o último Nó, a Árvore fica vazia.

- Balanceamento:

- Valem as mesmas regras da inserção;
- Remover um Nó da Sub-Árvore da direita equivale a inserir um Nó na Sub-Árvore da esquerda.



Removido um nó da árvore da direita, balanceia-se a árvore da esquerda, e vice e versa.

## Árvore AVL – Remoção em Árvore AVL

```
//Arquivo arvoreAVL.h
int remove_arvAVL(arvAVL *raiz, int valor);

//programa principal
x = remove_arvAVL(raiz, valor);
if(x){
    printf("Elemento removido com sucesso!.");
}else{
    printf("Erro, não foi possível remover o elemento.");
}
```

```
//Arquivo arvAVL.c
int remove_arvAVL(arvAVL *raiz, int valor){
    //função responsável pela busca do Nó a ser removido

    struct NO *procuramenor(struct NO *atual){
        //função responsável por tratar a remoção de um Nó com 2 filhos
```

Não é possível simplesmente remover o nó, é necessário substituí-lo por outro.



# Estrutura de Dados 2

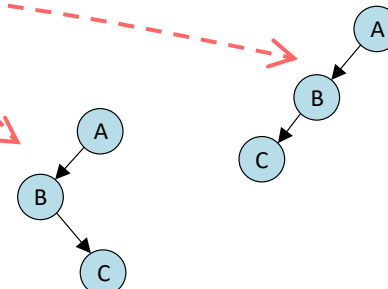
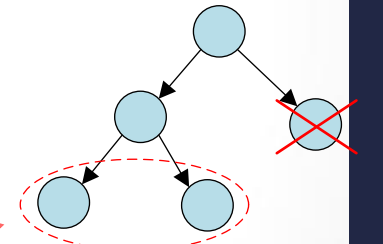
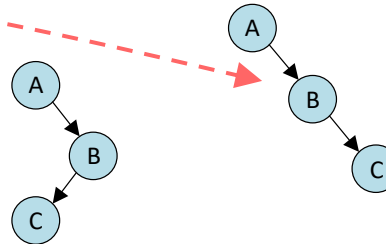
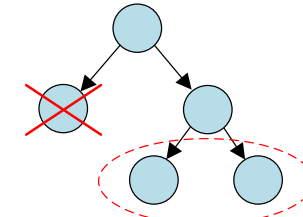


```
//Arquivo arvoreAVL.c
```

```
int remove_arvAVL(arvAVL *raiz, int valor){
    if(*raiz == NULL){
        return 0;
    }
    int res;
    if(valor < (*raiz)->info){
        if((res = remove_arvAVL(&(*raiz)->esq, valor)) == 1){
            if(fatorBalanceamento_NO(*raiz) >= 2){
                if(alt_no((*raiz)->dir->esq) <= alt_no((*raiz)->dir->dir)){
                    rotacaoRR(raiz);
                }else{
                    rotacaoRL(raiz);
                }
            }
        }
    }
    if((*raiz)->info < valor){
        if((res = remove_arvAVL(&(*raiz)->dir, valor)) == 1){
            if(fatorBalanceamento_NO(*raiz) >= 2){
                if(alt_no((*raiz)->esq->dir) <= alt_no((*raiz)->esq->esq)){
                    rotacaoLL(raiz);
                }else{
                    rotacaoLR(raiz);
                }
            }
        }
    }
}
```

Se a resposta for positiva, removeu um nó, então verifica balanceamento

Como foi removido da esquerda, é necessário verificar a Árvore da direita



# Estrutura de Dados 2



Pai tem um filho ou nenhum

Pai tem 2 filhos, substituir pelo Nó mais a esquerda (menor) da sub-árvore da direita

Remove da sub-árvore da direita o valor recuperado no passo anterior que está armazenado em "temp"

Trata realmente a remoção

Determina quantos filhos tem

Verifica qual é o filho que existe

Remove o nó que foi retornado pela função `procuramenor()`, e balanceia na esquerda

Terminada a remoção, atualiza a altura do nó e retorna 1 indicando o sucesso na remoção

```
if((*raiz)->info == valor){
    if((( *raiz)->esq == NULL) || ( *raiz)->dir == NULL){
        struct NO *no_velho = ( *raiz);
        if(( *raiz)->esq != NULL){
            *raiz = ( *raiz)->esq;
        }else{
            *raiz = ( *raiz)->dir;
        }
        free(no_velho);
    }else{
        struct NO *temp = procuramenor(( *raiz)->dir);
        ( *raiz)->info = temp->info;
        remove_arvAVL( ( *raiz)->dir, ( *raiz)->info);
        if(fatorBalanceamento_NO( *raiz) >= 2){
            if(alt_no(( *raiz)->esq->dir) <= alt_no(( *raiz)->esq->esq)){
                rotacaoLL(raiz);
            }else{
                rotacaoLR(raiz);
            }
        }
    }
    if( *raiz != NULL){
        ( *raiz)->alt = maior(alt_no(( *raiz)->esq), alt_no(( *raiz)->dir)) + 1;
    }
    return 1;
}
if( *raiz != NULL){
    ( *raiz)->alt = maior(alt_no(( *raiz)->esq), alt_no(( *raiz)->dir)) + 1;
}
return res;
}
```



## Árvore AVL – Remoção em Árvore AVL



```
//função auxiliar - procura nó mais a esquerda
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;
}
```

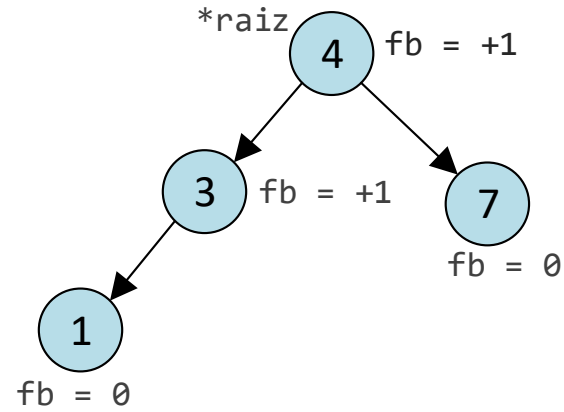
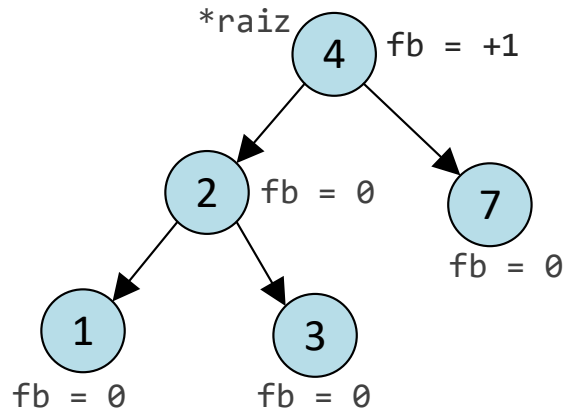
Procura o nó que está mais a esquerda

Enquanto no2 for diferente de NULL, ponteiro se desloca cada vez mais a esquerda. Sempre guardando o último nó visitado no ponteiro no1

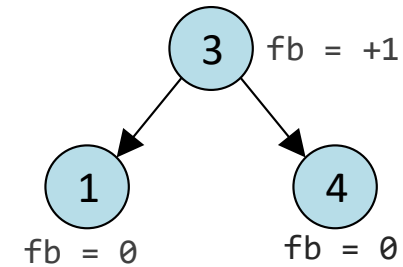
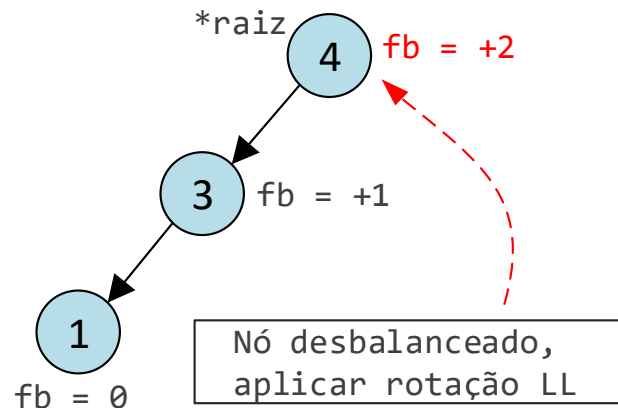
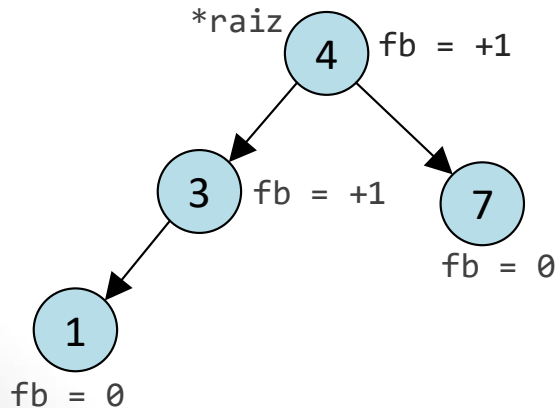
Por fim, retorna o ultimo nó válido visitado à esquerda, que é o menor dos maiores elementos que estão a direita do nó a ser removido, que então tomará a sua posição

## Árvore AVL – Remoção em Árvore AVL

Remove valor 2



Remove valor 7



Árvore Balanceada

# Estrutura de Dados 2

## Atividade Árvore AVL

- Entregue no Moodle o projeto Árvore AVL final.

