# ÁRVORES AVL

## Problema do balanceamento



- A eficiência da busca em uma árvore binária depende do seu balanceamento.
  - ***O(log N)***, se a árvore está balanceada
  - ***O(N)***, se a árvore não está balanceada
    - **N** corresponde ao número de nós na árvore

# Problema do balanceamento

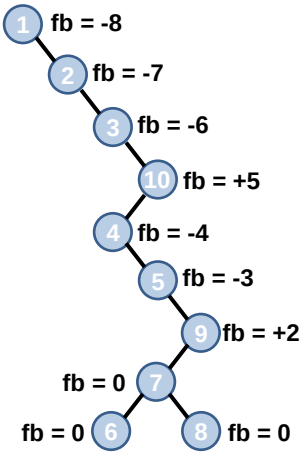
3

- Infelizmente, os algoritmos de inserção e remoção em árvores binárias não garantem que a árvore gerada a cada passo esteja balanceada.
- Dependendo da ordem em que os dados são inseridos na árvore, podemos criar uma árvore na forma de uma escada

# Problema do balanceamento

4

- Inserção dos valores {1,2,3,10,4,5,9,7,8,6}



# Problema do balanceamento

5

- Solução para o problema de balanceamento
  - Modificar as operações de inserção e remoção de modo a balancear a árvore a cada nova inserção ou remoção.
    - Garantir que a diferença de alturas das sub-árvores esquerda e direita de cada nó seja de no máximo uma unidade
  - Exemplos de árvores balanceadas
    - Árvore AVL
    - Árvore 2-3-4
    - Árvore Rubro-Negra

# Árvore AVL

6

- Definição
  - Tipo de árvore binária balanceada com relação a altura das suas sub-árvores
  - Criada por **A**delson-**V**elskii e **L**andis, de onde recebeu a sua nomenclatura, em 1962

# Árvore AVL

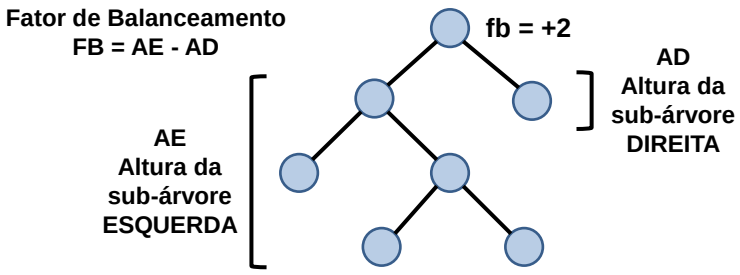
7

- Definição
  - Permite o rebalanceamento local da árvore
    - Apenas a parte afetada pela inserção ou remoção é rebalanceada
  - Usa **rotações simples** ou **duplas** na etapa de rebalanceamento
    - Executadas a cada inserção ou remoção
    - As rotações buscam manter a árvore binária como uma árvore quase completa
    - Custo máximo de qualquer algoritmo é ***O(log N)***

# Árvore AVL

8

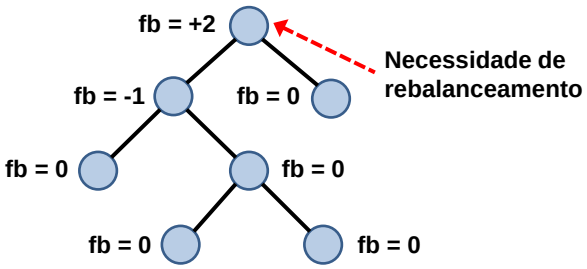
- Objetivo das rotações:
  - Corrigir o **fator de balanceamento** (ou **fb**)
    - Diferença entre as alturas das sub-árvore de um nó
  - Caso uma das sub-árvores de um nó não existir, então a altura dessa sub-arvore será igual a -1.



# Árvore AVL

9

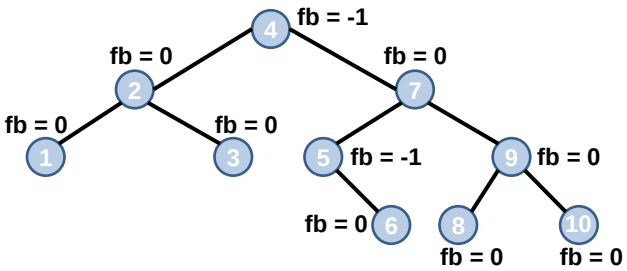
- As alturas das sub-árvores de cada nó diferem de no máximo uma unidade
  - O fator de balanceamento deve ser +1, 0 ou -1
  - Se **fb > +1** ou **fb < -1**: a árvore deve ser balanceada naquele nó




# Árvore AVL

10

- Voltando ao problema anterior
- Inserção dos valores {1,2,3,10,4,5,9,7,8,6}



# TAD Árvore AVL

11

- Definindo a árvore
  - Criação e destruição: igual a da árvore binária

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

//Arquivo ArvoreAVL.c
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreAVL.h" //inclui os Protótipos
struct NO{
    int info;
    int alt;//altura daquela sub-árvore
    struct NO *esq;
    struct NO *dir;
};
//programa principal
ArvAVL* raiz; //ponteiro para ponteiro
```

# TAD Árvore AVL

12

- Calculando o fator de balanceamento

```c
//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));
}
```

# Rotações

13

- Objetivo: corrigir o **fator de balanceamento** (ou **fb**) de cada nó
  - Operação básica para balancear uma árvore AVL
- Ao todo, existem dois tipos de rotação
  - Rotação simples
  - Rotação dupla

# Rotações

14

- As rotações diferem entre si pelo sentido da inclinação entre o nó pai e filho
  - Rotação simples
    - O nó desbalanceado (pai), seu filho e o seu neto estão todos no mesmo sentido de inclinação
  - Rotação dupla
    - O nó desbalanceado (pai) e seu filho estão inclinados no sentido inverso ao neto
    - **Equivale a duas rotações simples.**

## Rotações

15

- Ao todo, existem duas rotações simples e duas duplas:
  - Rotação **simples a direita** ou **Rotação LL**
  - Rotação **simples a esquerda** ou **Rotação RR**
  - Rotação **dupla a direita** ou **Rotação LR**
  - Rotação **dupla a esquerda** ou **Rotação RL**

## Rotações

16

- Rotações são aplicadas no ancestral mais próximo do nó inserido cujo fator de balanceamento passa a ser +2 ou -2
  - Após uma inserção ou remoção, devemos voltar pelo mesmo caminho da árvore e recalcular o fator de balanceamento, **fb**, de cada nó
  - Se o **fb** desse nó for +2 ou -2, uma rotação deverá ser aplicada

# Rotação LL

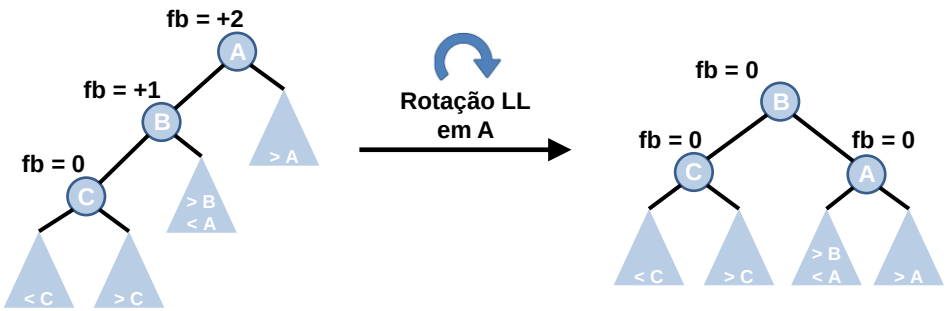
17

- 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
    - Dois movimentos para a esquerda: **LEFT LEFT**
  - 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

18

- Exemplo



# TAD Árvore AVL

19

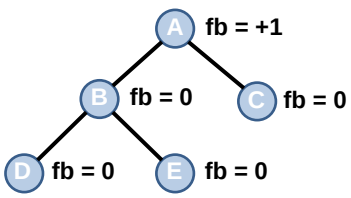
- Rotação LL

```
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 }
```

# Rotação LL

20

- Passo a passo

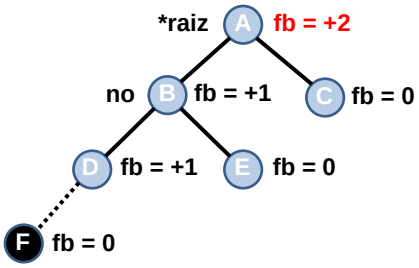



**Árvore AVL e fator de balanceamento de cada nó**

# Rotação LL

21

- Passo a passo




**Inserção do nó F na árvore**

**Árvore fica desbalanceada no nó A.**

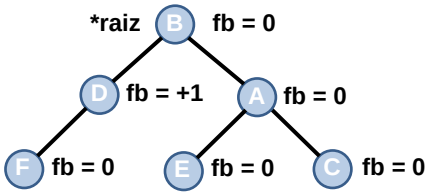
**Aplicar Rotação LL no nó A**

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

# Rotação LL

22

- Passo a passo




**Árvore Balanceada**

# Rotação RR

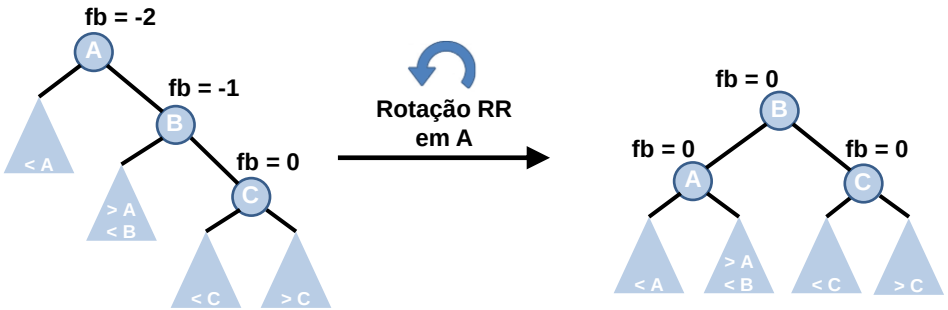
23

- 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
    - Dois movimentos para a direita: **RIGHT RIGHT**
  - 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

24

- Exemplo



# TAD Árvore AVL

25

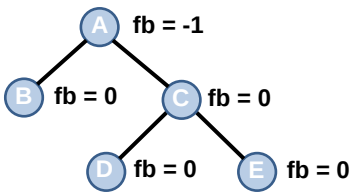
- Rotação RR

```
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 }
```

# Rotação RR

26

- Passo a passo

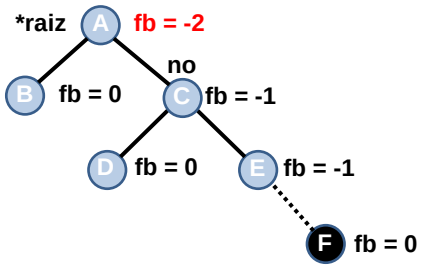



**Árvore AVL e fator de balanceamento de cada nó**

# Rotação RR

27

- Passo a passo




**Inserção do nó F na árvore**

**Árvore fica desbalanceada no nó A.**

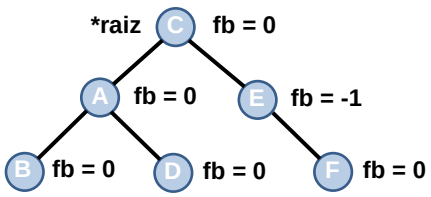
**Aplicar Rotação RR no nó A**

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

# Rotação RR

28

- Passo a passo




**Árvore Balanceada**

# Rotação LR

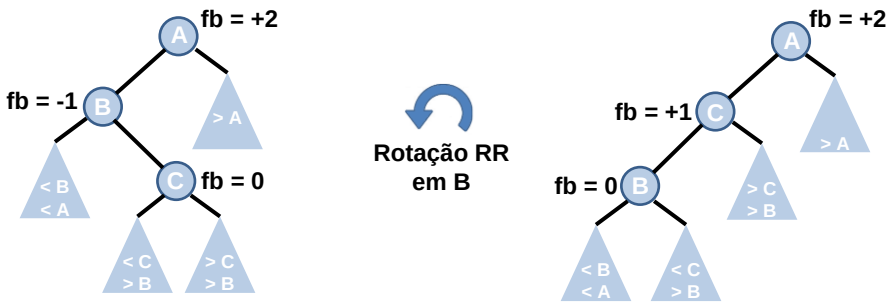
29

- 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
    - Um movimento para a esquerda e outro para a direita: **LEFT RIGHT**
  - 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

30

- Exemplo: primeira rotação

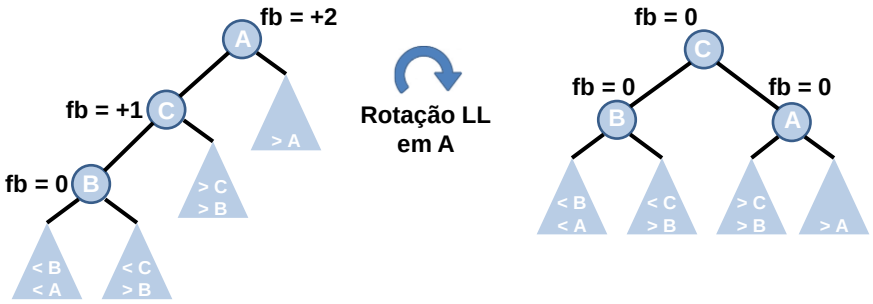



```
1
2  void RotacaoLR(ArvAVL *raiz){
3      RotacaoRR(&(*raiz)->esq);
4      RotacaoLL(raiz);
5  }
```

# Rotação LR

31

- Exemplo: segunda rotação

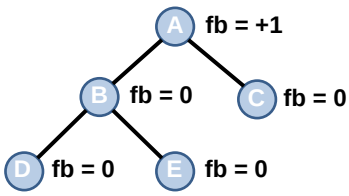



```
1
2  void RotacaoLR(ArvAVL *raiz){
3      RotacaoRR(&(*raiz)->esq);
4      RotacaoLL(raiz);
5  }
```

# Rotação LR

32

- Passo a passo

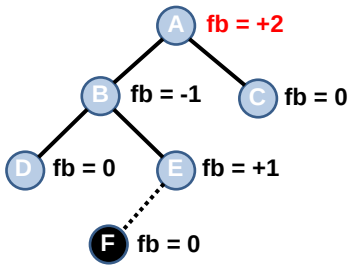



**Árvore AVL e fator de balanceamento de cada nó**

# Rotação LR

33

- Passo a passo




**Inserção do nó F na árvore**

**Árvore fica desbalanceada no nó A.**

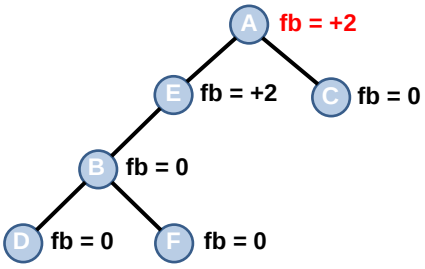
**Aplicar Rotação LR no nó A. Isso equivale a:**

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

# Rotação LR

34

- Passo a passo

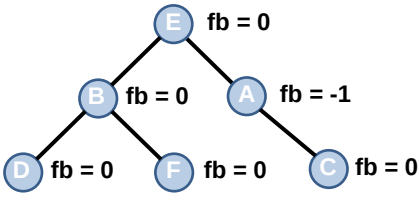



**Árvore após aplicar a Rotação RR no nó B**

# Rotação LR

35

- Passo a passo




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

**Árvore Balanceada**

# Rotação RL

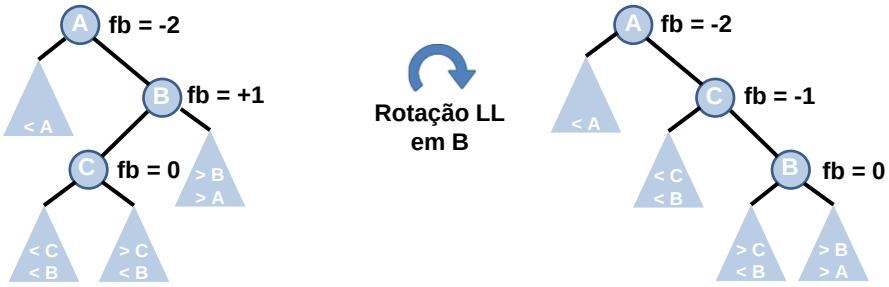
36

- 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
    - Um movimento para a direita e outro para a esquerda: **RIGHT LEFT**
  - 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

37

- Exemplo

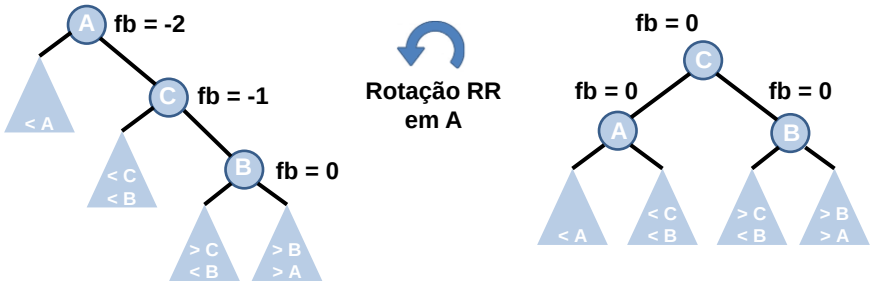



```
1
2  void RotacaoRL(ArvAVL *raiz){
3      RotacaoLL(&(*raiz)->dir);
4      RotacaoRR(raiz);
5  }
```

# Rotação RL

38

- Exemplo

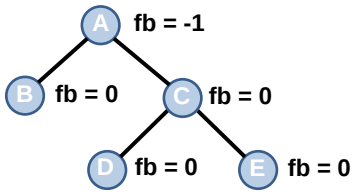



```
1
2  void RotacaoRL(ArvAVL *raiz){
3      RotacaoLL(&(*raiz)->dir);
4      RotacaoRR(raiz);
5  }
```

# Rotação RL

39

- Passo a passo

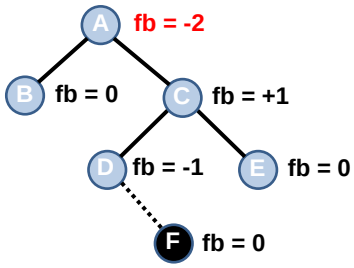



**Árvore AVL e fator de balanceamento de cada nó**

# Rotação RL

40

- Passo a passo




**Inserção do nó F na árvore**

**Árvore fica desbalanceada no nó A.**

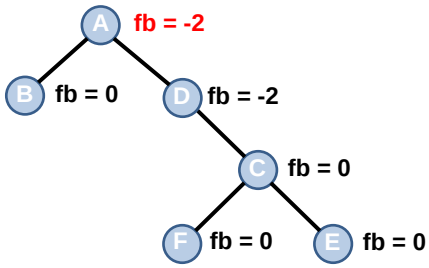
**Aplicar Rotação RL no nó A. Isso equivale a:**

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

# Rotação RL

41

- Passo a passo

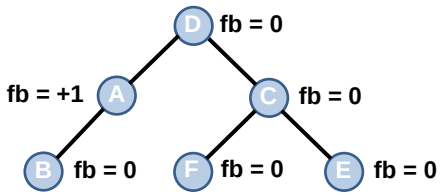



**Árvore após aplicar a Rotação LL no nó C**

# Rotação RL

42

- Passo a passo




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

**Árvore Balanceada**

# Quando usar cada rotação?

43

- 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<br>C é filho à esquerda de B | LL |
| -2 | -1 | B é filho à direita de A<br>C é filho à direita de B | RR |
| +2 | -1 | B é filho à esquerda de A<br>C é filho à direita de B | LR |
| -2 | +1 | B é filho à de direita A<br>C é filho à esquerda de B | RL |

# Quando usar cada rotação?

44

- 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<br>C é filho à esquerda de B | LL |
| -2 | -1 | B é filho à direita de A<br>C é filho à direita de B | RR |
| +2 | -1 | B é filho à esquerda de A<br>C é filho à direita de B | LR |
| -2 | +1 | B é filho à de direita A<br>C é filho à esquerda de B | RL |

# Quando usar cada rotação?

45

- 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<br>C é filho à esquerda de B | LL |
| -2 | -1 | B é filho à direita de A<br>C é filho à direita de B | RR |
| +2 | -1 | B é filho à esquerda de A<br>C é filho à direita de B | LR |
| -2 | +1 | B é filho à de direita A<br>C é filho à esquerda de B | RL |

# Árvore AVL: Inserção

46

- 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

47

- 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**

# TAD Árvore AVL

48

- Inserção

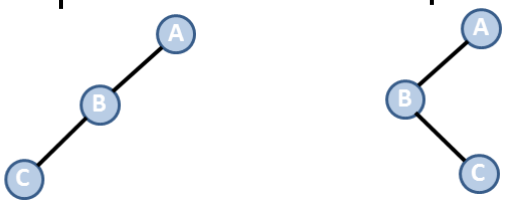
```
int insere_ArvAVL(ArvAVL *raiz, int valor){
    int res;
    if(*raiz == NULL){//árvore vazia ou nó folha
        struct NO *novo;
        novo = (struct NO*)malloc(sizeof(struct NO));
        if(novo == NULL)
            return 0;

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

# TAD Árvore AVL

49

- 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);
          }
      }
  }
}
```

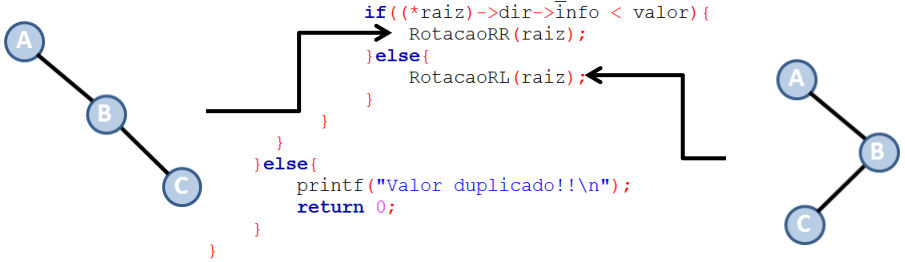



# TAD Árvore AVL

50

- Inserção

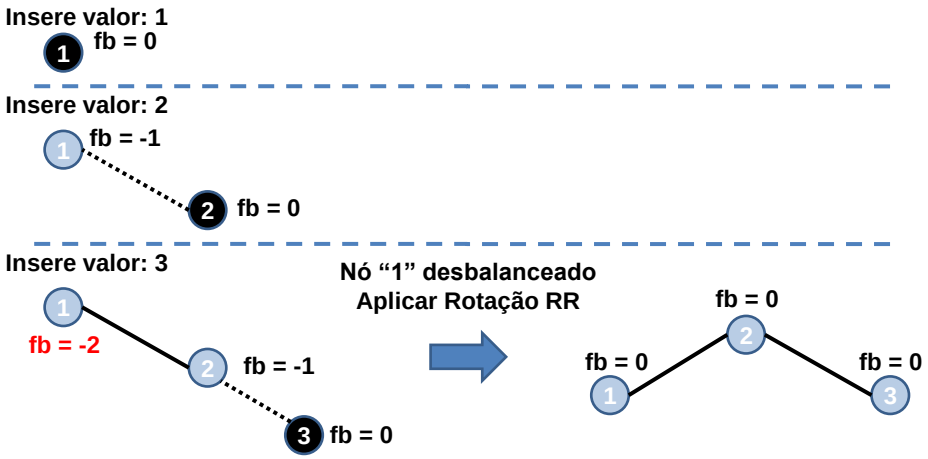
```
//continuação
else{
    if(valor > atual->info){
      if((res=insere_ArvAVL(&(atual->dir), valor))==1){
          if(fatorBalanceamento_NO(atual) >= 2){
              if((*raiz)->dir->info < valor){
                  RotacaoRR(raiz);
              }else{
                  RotacaoRL(raiz);
     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



# Árvore AVL: Inserção

51

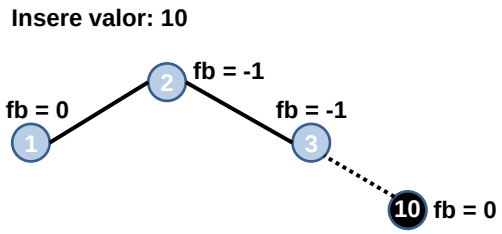
- Passo a passo




# Árvore AVL: Inserção

52

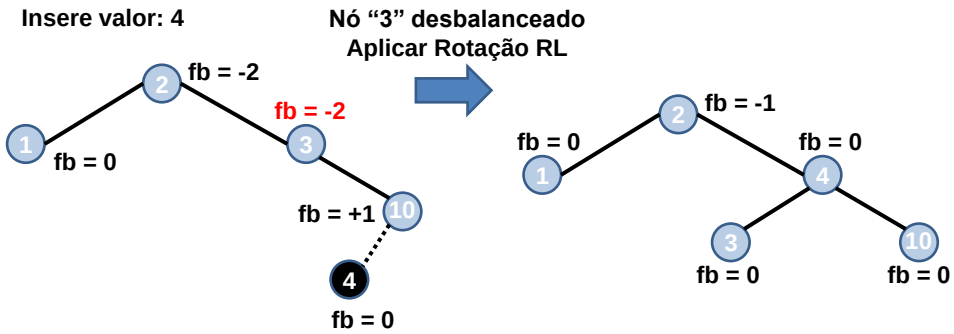
- Passo a passo



# Árvore AVL: Inserção

53

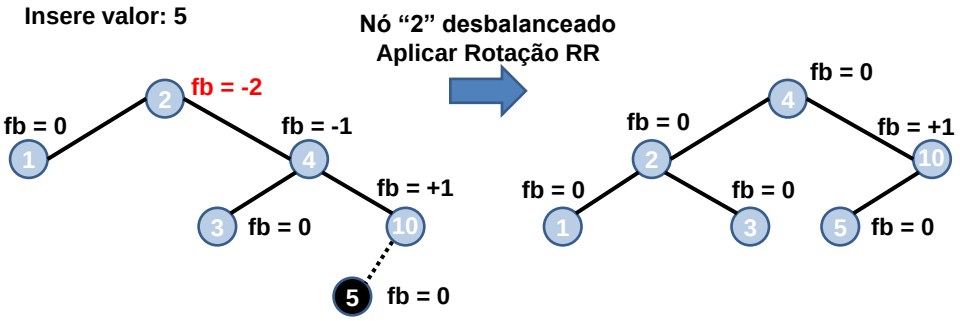
- Passo a passo




# Árvore AVL: Inserção

54

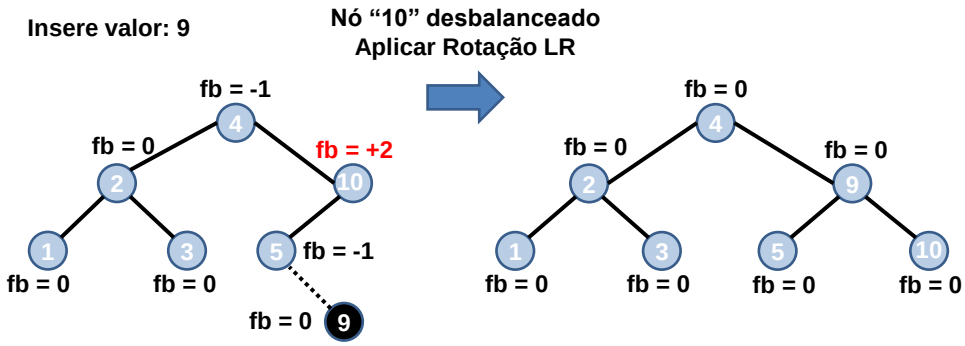
- Passo a passo



# Árvore AVL: Inserção

55

- Passo a passo




# Árvore AVL: Remoção

56

- 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

57

- 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**

# TAD Árvore AVL

58

- Remoção
  - Trabalha com 2 funções
    - Busca pelo nó
    - Remoção do nó com 2 filhos

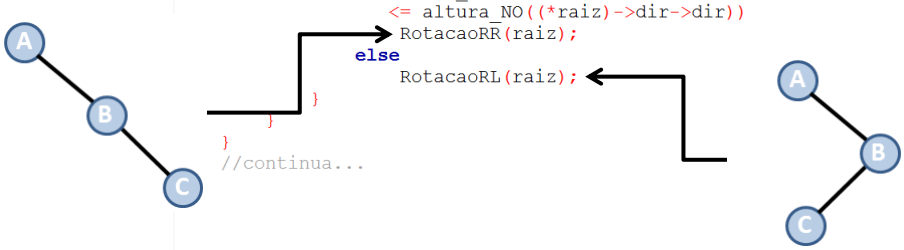
```
8  int remove_ArvAVL(ArvAVL *raiz, int valor){
9      /*
10     FUNÇÃO RESPONSÁVEL PELA BUSCA
11     DO NÓ A SER REMOVIDO
12     */
13 }
14 struct NO* procuraMenor(struct NO* atual){
15     /*
16     FUNÇÃO RESPONSÁVEL POR TRATAR OS
17     A REMOÇÃO DE UM NÓ COM 2 FILHOS
18     */
19 }
```

# TAD Árvore AVL

59

- 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...
```

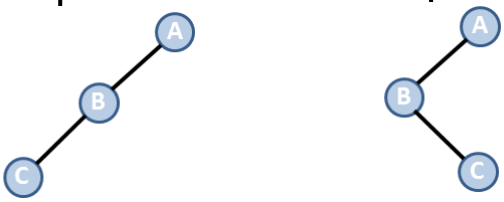



# TAD Árvore AVL

60

- 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...
```



# TAD Árvore AVL

61

## Remoção

```
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;
}
```

Pai tem 1 ou nenhum filho

Pai tem 2 filhos: Substituir pelo nó mais a esquerda da sub-árvore da direita

Corrige a altura

# TAD Árvore AVL

62

## Remoção

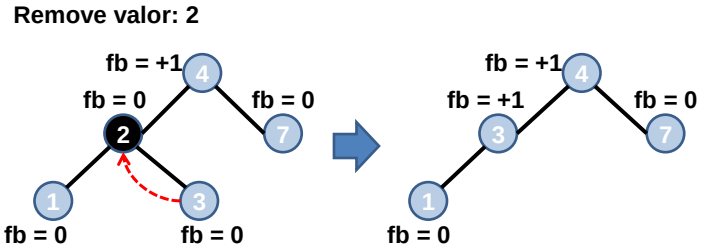
```
struct NO* procuraMenor(struct NO* atual){
    struct NO *no1 = atual;
    struct NO *no2 = atual->esq;
    while(no2 != NULL){
        no1 = no2;
        no2 = no2->esq;
    }
    return no1;
}
```

Procura pelo nó mais a esquerda

# Árvore AVL: Remoção

63

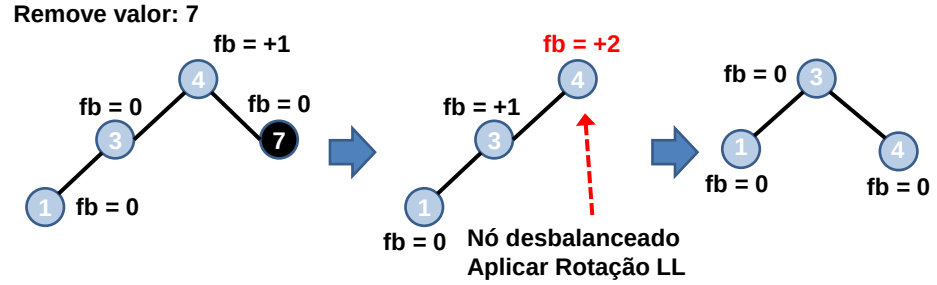
- Passo a passo




# Árvore AVL: Remoção

64

- Passo a passo




FIM