ÁRVORES BALANCEADAS

Prof. André Backes | @progdescomplicada

ÁRVORE AVL

Problema do balanceamento

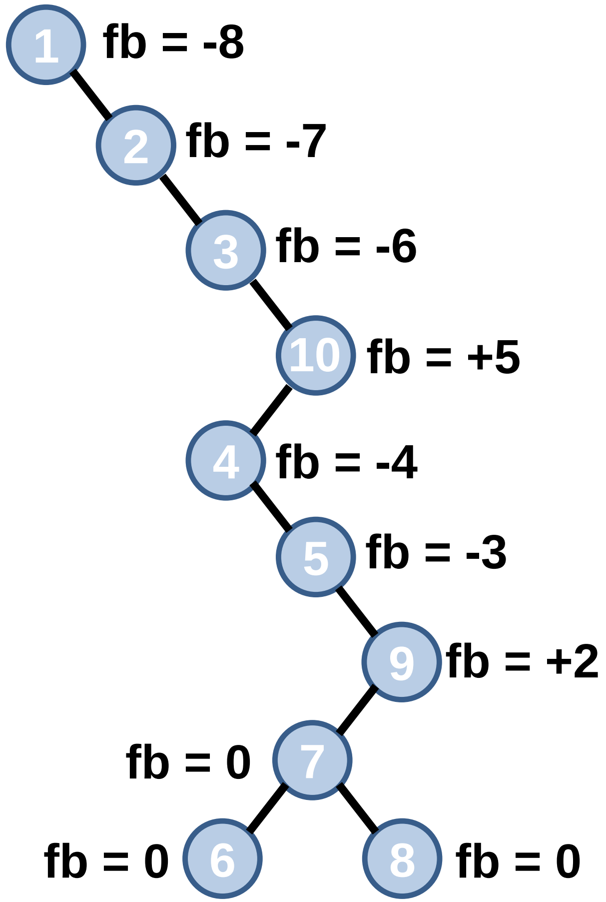
- A eficiência da busca em uma árvore binária depende do seu balanceamento.
  - ***O(log N)***, se a árvore está balanceada
  - ***O(N)***, se a árvore não está balanceada
    - **N** corresponde ao número de nós na árvore

Problema do balanceamento

- 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

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



Problema do balanceamento

- 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 | 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

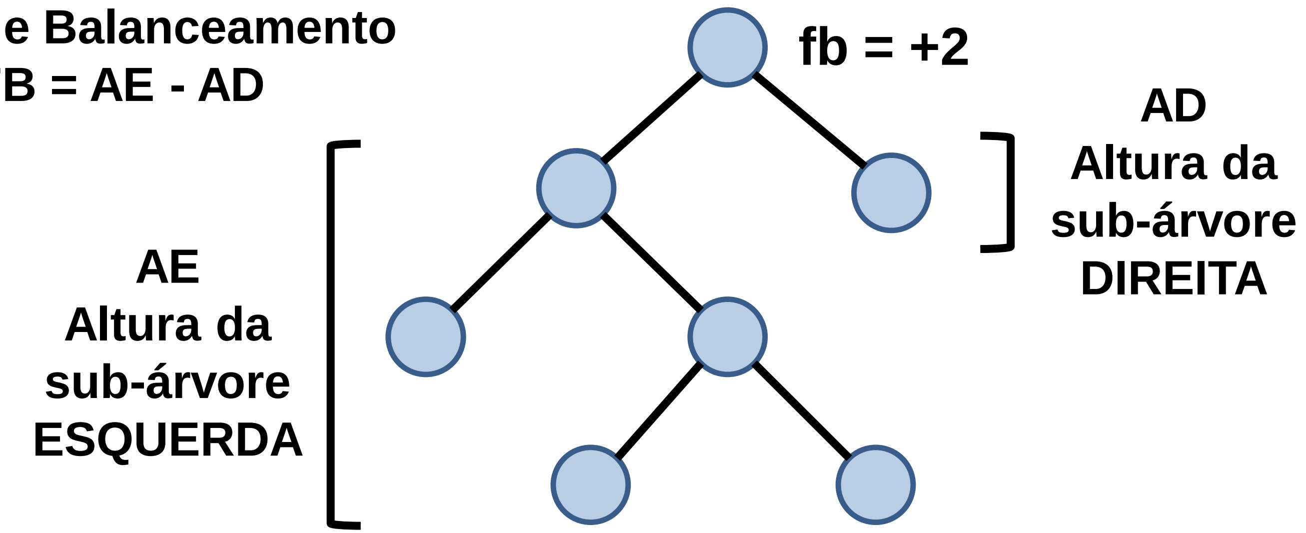
Árvore AVL | Definição

- 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 | Definição

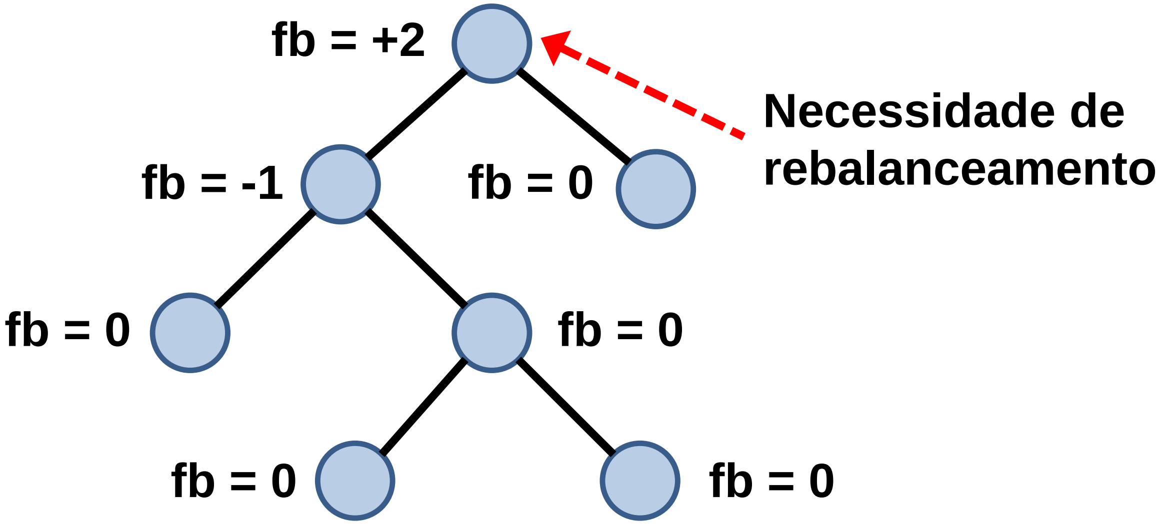
- Objetivo das rotações:
  - Corrigir o **fator de balanceamento** (ou **fb**)
    - Diferença entre as alturas das sub-árvore de um nó
  - Caso uma das sub-árvores de um nó não existir, então a altura dessa sub-arvore será igual a -1.

Fator de Balanceamento
FB = AE - AD



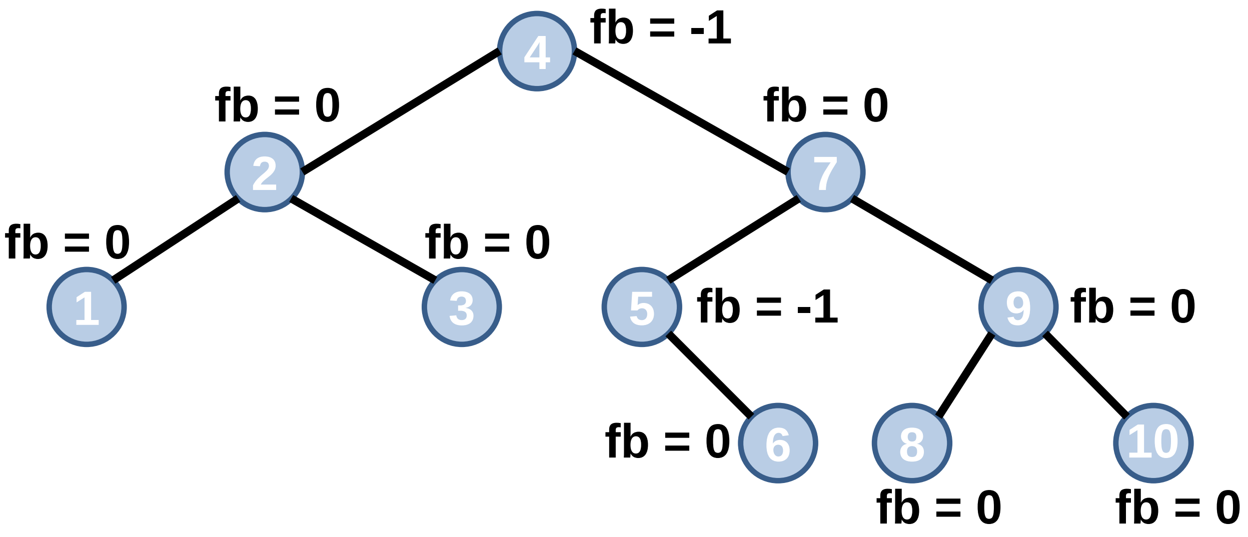
Árvore AVL | Definição

- 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

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



TAD Árvore AVL

- 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 altura;
    struct NO *esq;
    struct NO *dir;
};

// programa principal
ArvAVL* avl; // ponteiro para ponteiro
```

TAD Árvore AVL

- Calculando o fator de balanceamento

```c
// Funções auxiliares
// Calcula a altura de um nó
int altura_NO(struct NO* no){
    if(no == NULL)
        return -1;
    else
    return no->altura;
}

// Calcula o fator de balanceamento de um nó
int fatorBalanceamento_NO(struct NO* no){
    return labs(altura_NO(no->esq) - altura_NO(no->dir));
}

// Calcula o maior valor
int maior(int x, int y){
    if(x > y) return x;
    else return y;
}
```

Rotações

- 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

- 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

- 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**

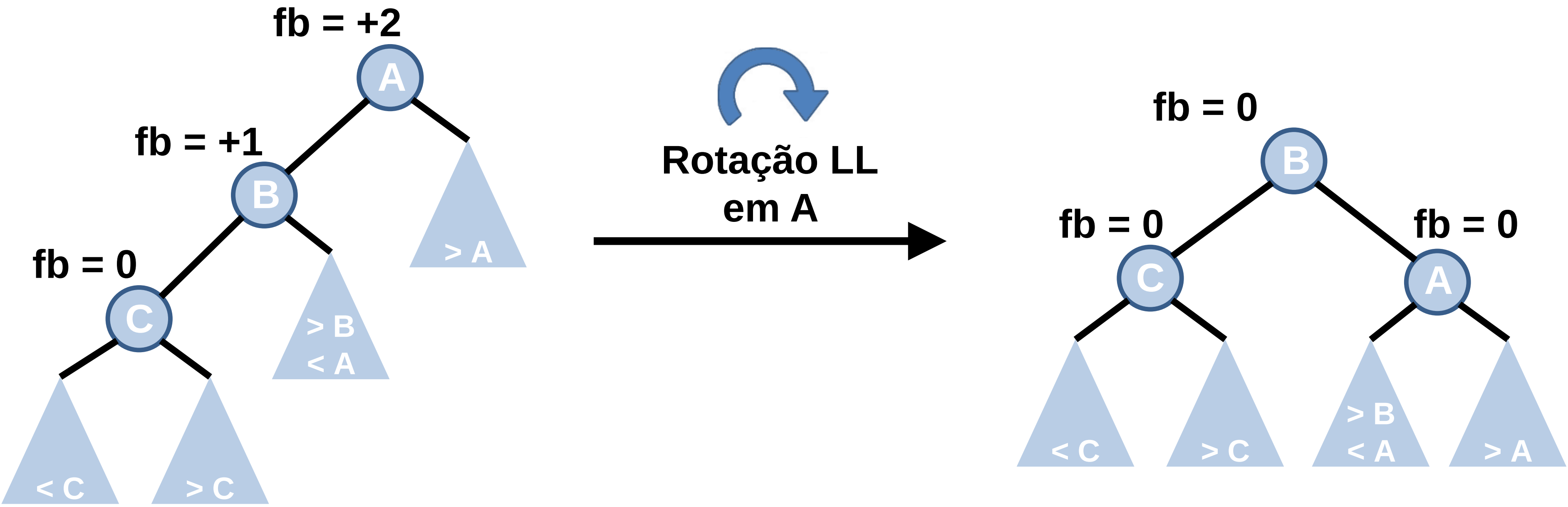
Rotações

- 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

- 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**

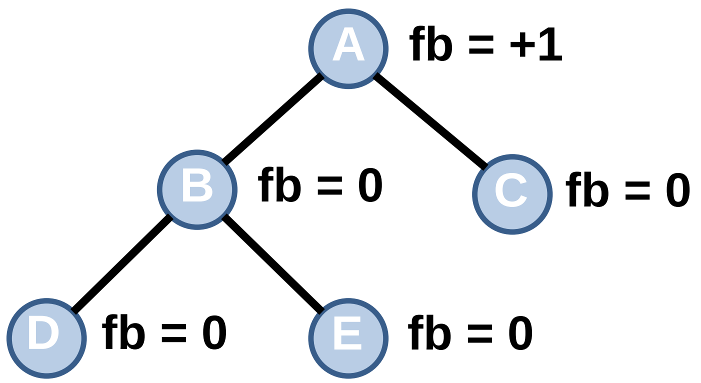
Rotação LL | Exemplo



Rotação LL | Implementação

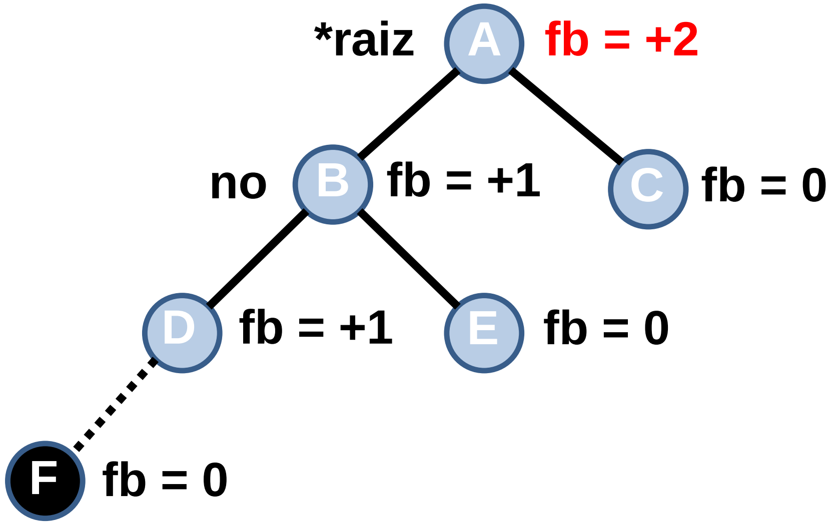
```
void RotacaoLL(ArvAVL *A){
    struct NO *B;
    B = (*A)->esq;
    (*A)->esq = B->dir;
    B->dir = *A;
    (*A)->altura = maior(altura_NO((*A)->esq),altura_NO((*A)->dir)) + 1;
    B->altura = maior(altura_NO(B->esq),(*A)->altura) + 1;
    *A = B;
}
```

Rotação LL | Passo a passo



Árvore AVL e fator de balanceamento de cada nó

Rotação LL | Passo a passo



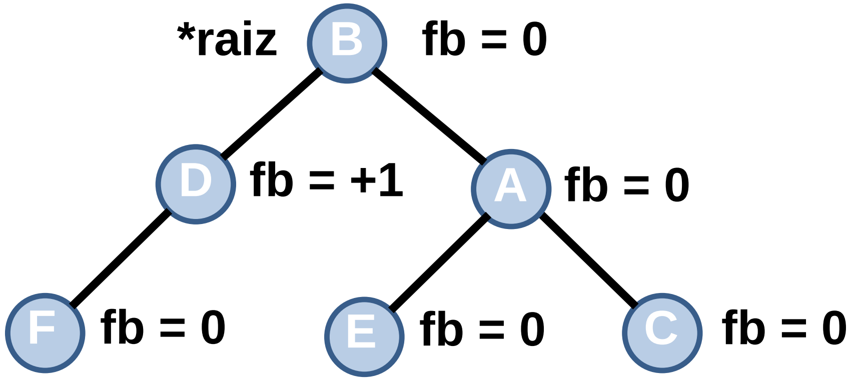
Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação LL no nó A

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

Rotação LL | Passo a passo

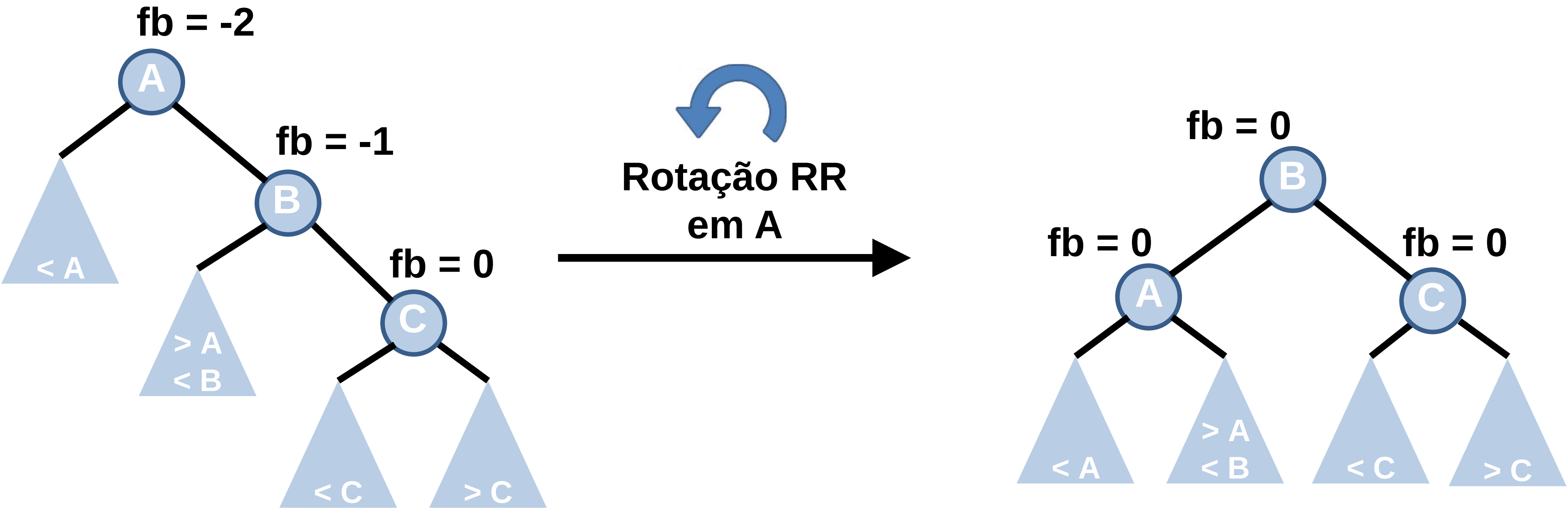


Árvore Balanceada

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
    - 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**

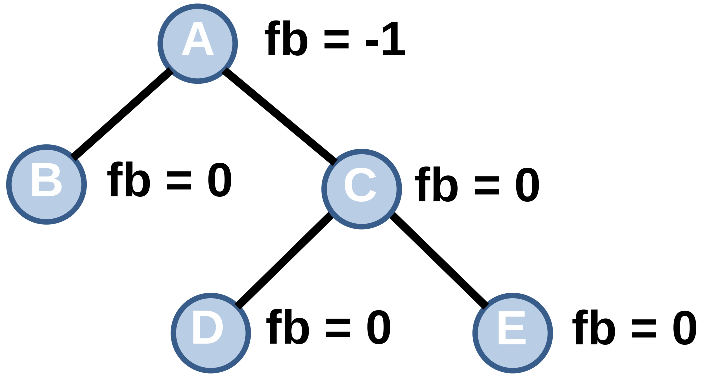
Rotação RR | Exemplo



TAD Árvore AVL | Implementação

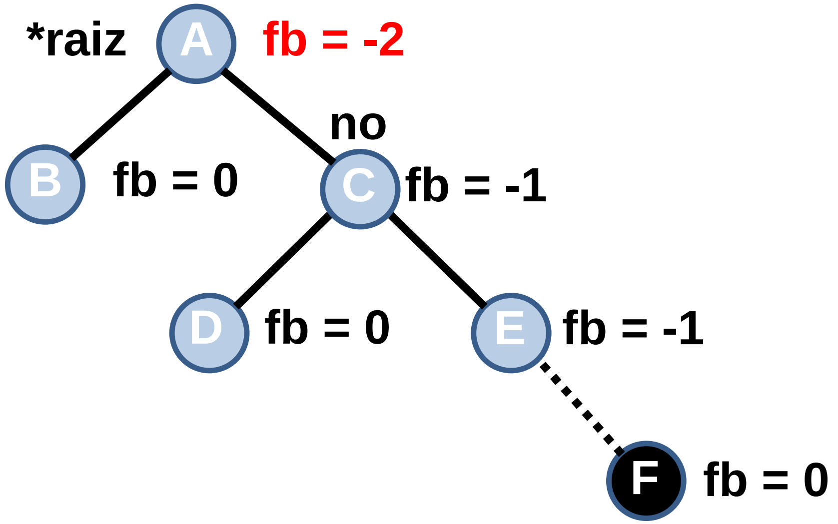
```c
void RotacaoRR(ArvAVL *A){
    struct NO *B;
    B = (*A)->dir;
    (*A)->dir = B->esq;
    B->esq = (*A);
    (*A)->altura = maior(altura_NO((*A)->esq),altura_NO((*A)->dir)) + 1;
    B->altura = maior(altura_NO(B->dir),(*A)->altura) + 1;
    (*A) = B;
}
```

Rotação RR | Passo a passo



Árvore AVL e fator de balanceamento de cada nó

Rotação RR | Passo a passo



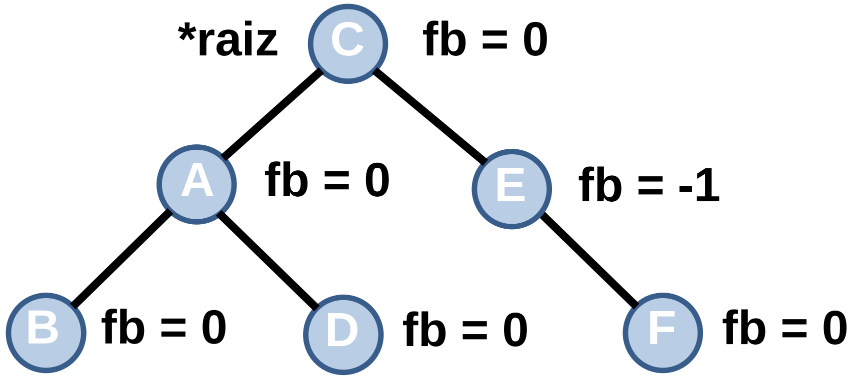
Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação RR no nó A

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

Rotação RR | Passo a passo



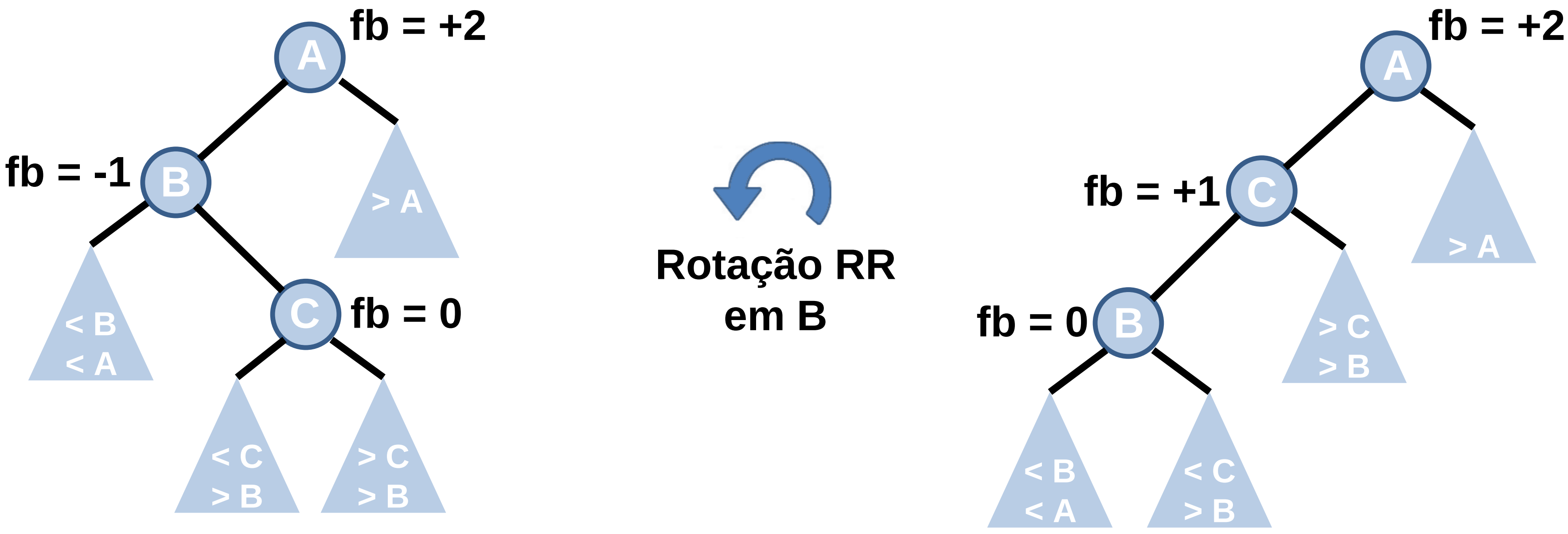
Árvore Balanceada

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
    - 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 | Exemplo

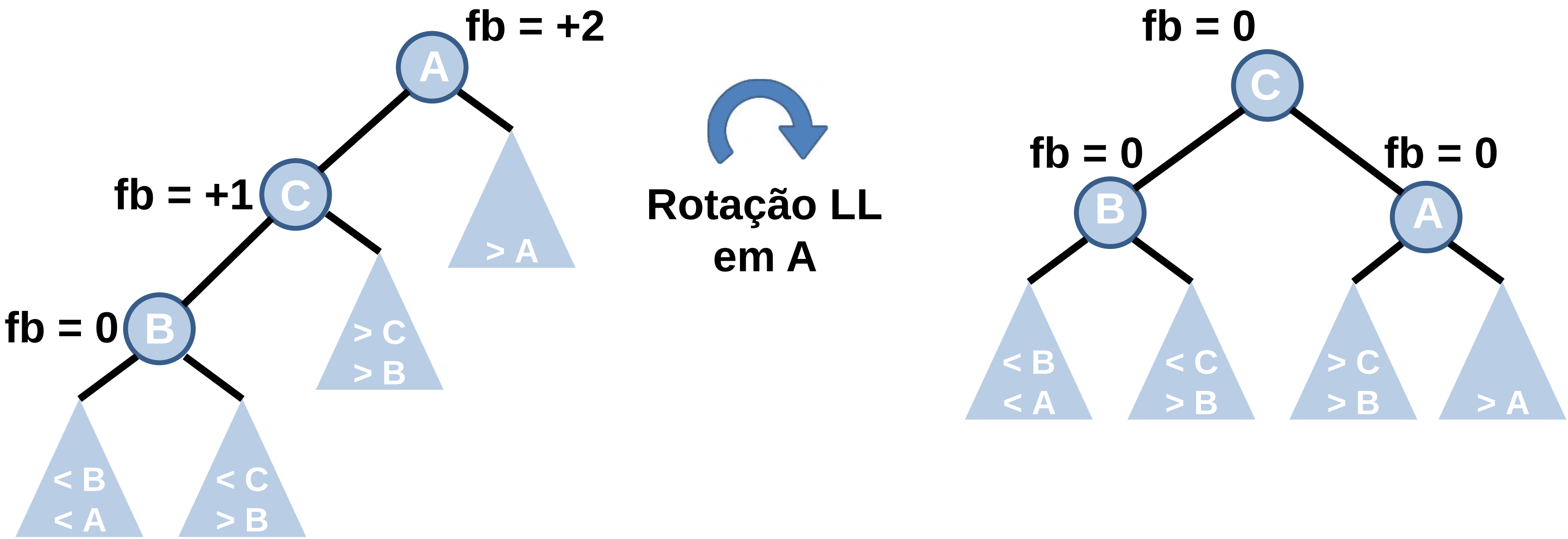
- Primeira rotação




```c
void RotacaoLR(ArvAVL *A){
    RotacaoRR(&(*A)->esq);
    RotacaoLL(A);
}
```

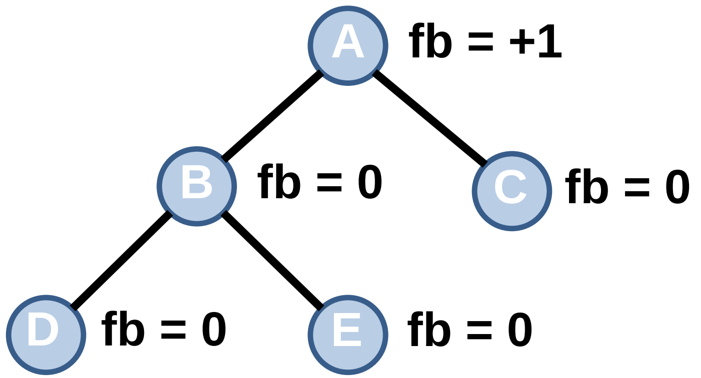
Rotação LR | Exemplo

- Segunda rotação



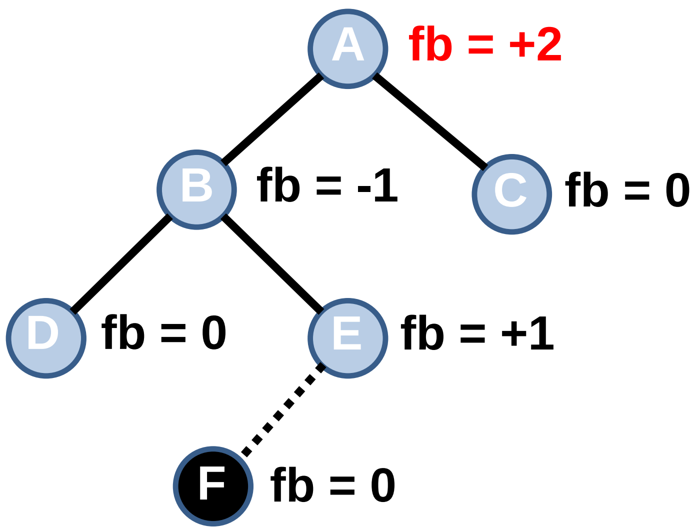

```c
void RotacaoLR(ArvAVL *A){
    RotacaoRR(&(*A)->esq);
    RotacaoLL(A);
}
```

Rotação LR | Passo a passo



Árvore AVL e fator de balanceamento de cada nó

Rotação LR | Passo a passo



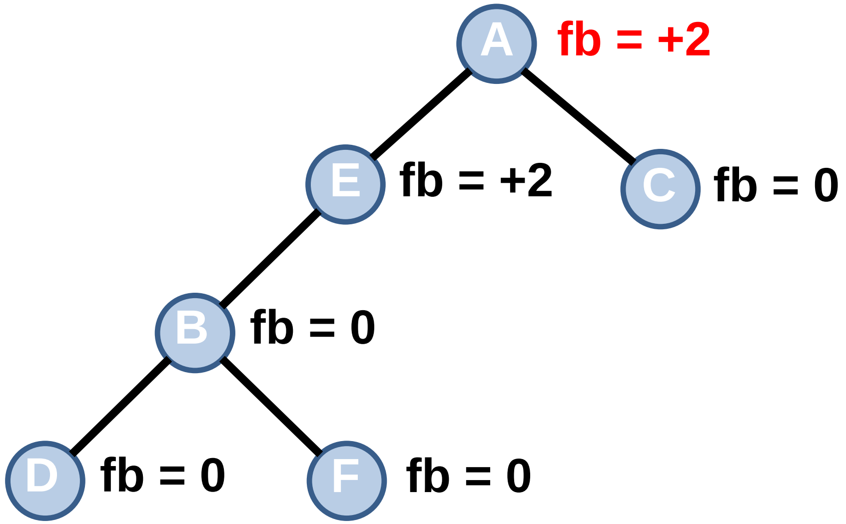
Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação LR no nó A.
Isso equivale a:

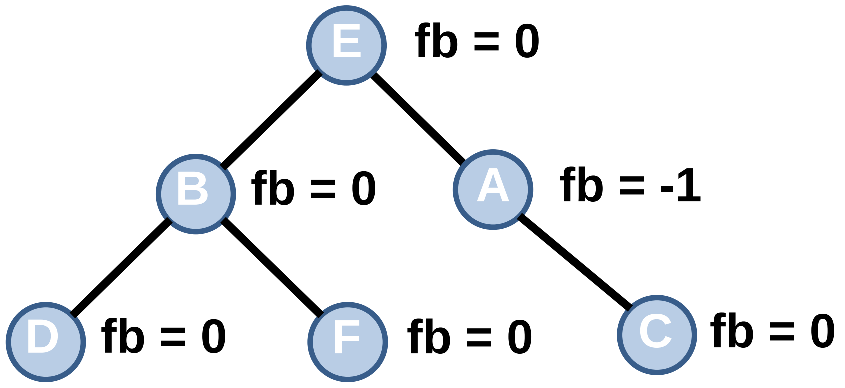
- Aplicar a Rotação RR no nó B
- Aplicar a Rotação LL no nó A

Rotação LR | Passo a passo



Árvore após aplicar a Rotação RR no nó B

Rotação LR | Passo a passo



Árvore após aplicar a Rotação LL no nó A

Árvore Balanceada

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
    - 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**

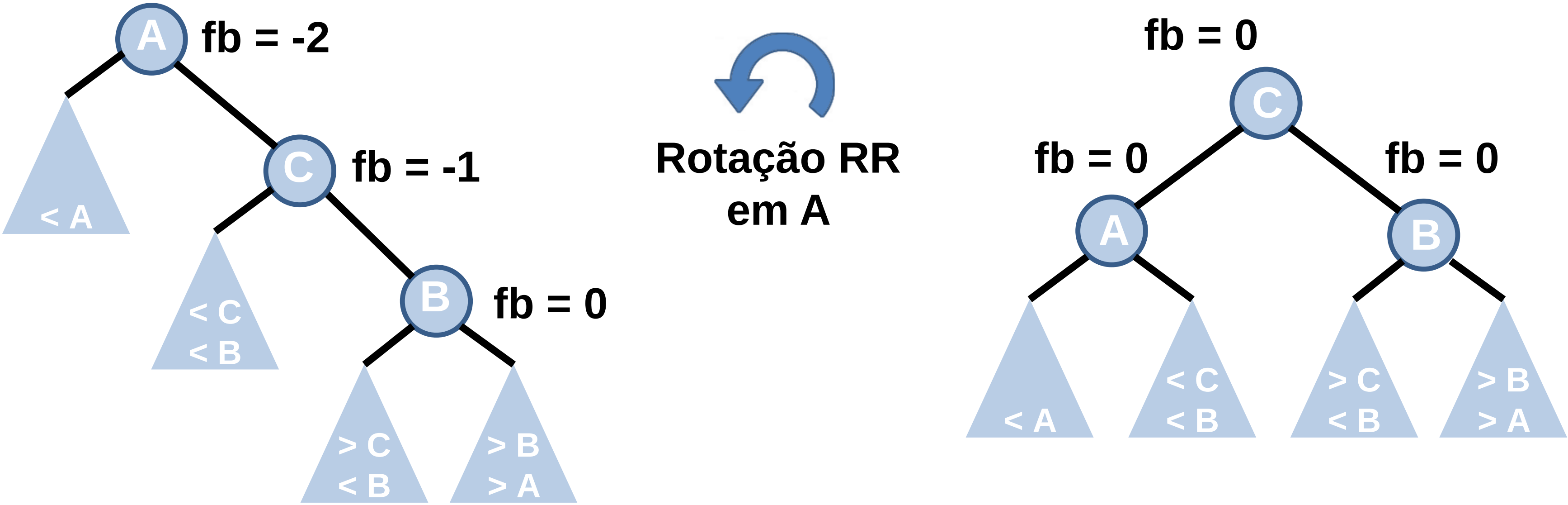
Rotação RL | Exemplo

- Primeira rotação


```c
void RotacaoRL(ArvAVL *A){
    RotacaoLL(&(*A)->dir);
    RotacaoRR(A);
}
```

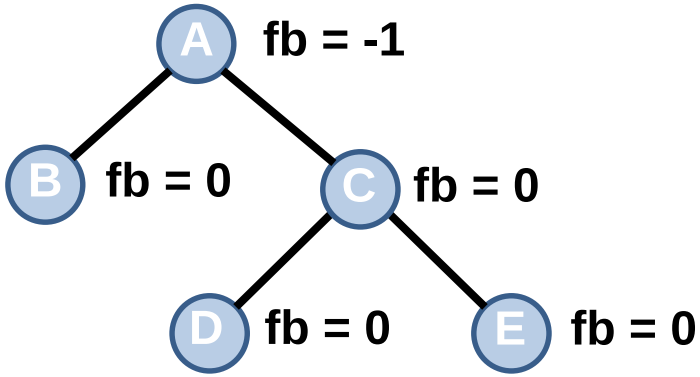
Rotação RL | Exemplo

- Segunda rotação



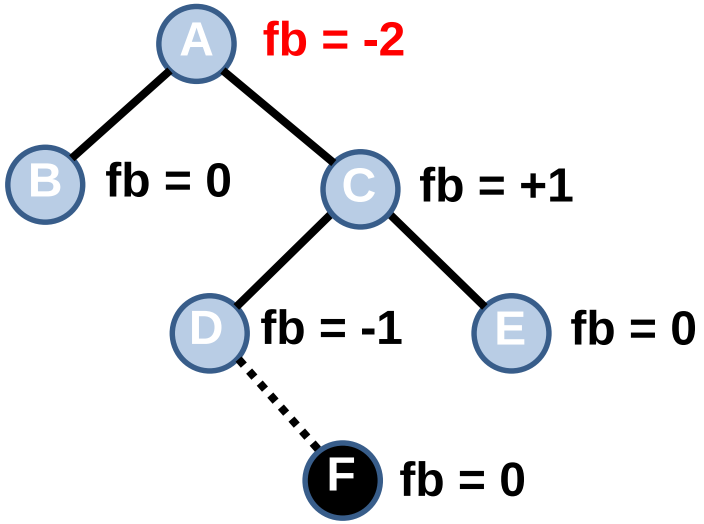

```
void RotacaoRL(ArvAVL *A){
    RotacaoLL(&(*A)->dir);
    RotacaoRR(A);
}
```

Rotação RL | Passo a passo



Árvore AVL e fator de balanceamento de cada nó

Rotação RL | Passo a passo



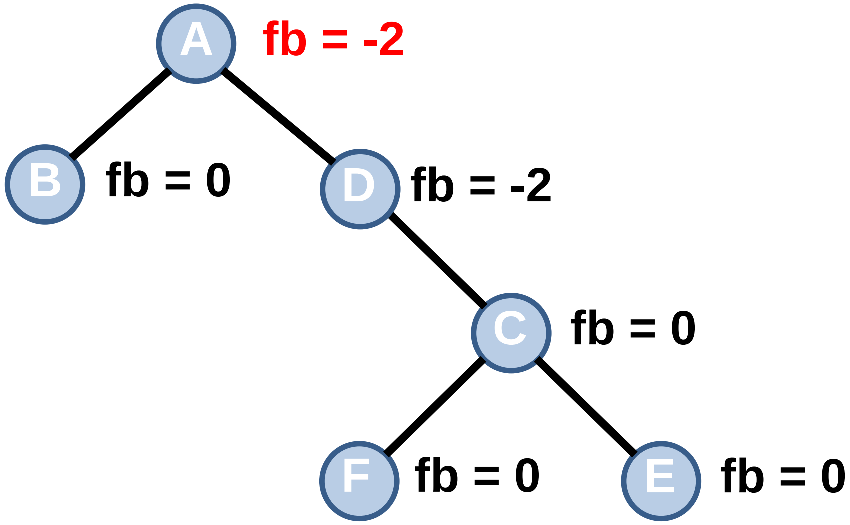
Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação RL no nó A.
Isso equivale a:

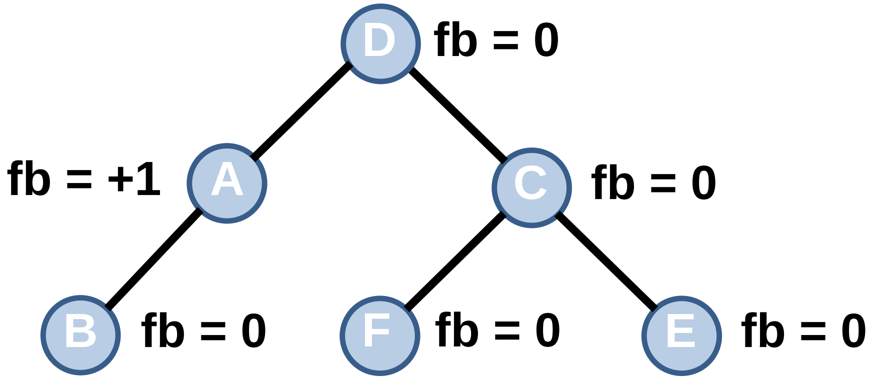
- Aplicar a Rotação LL no nó C
- Aplicar a Rotação RR no nó A

Rotação RL | Passo a passo



Árvore após aplicar a Rotação LL no nó C

Rotação RL | 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

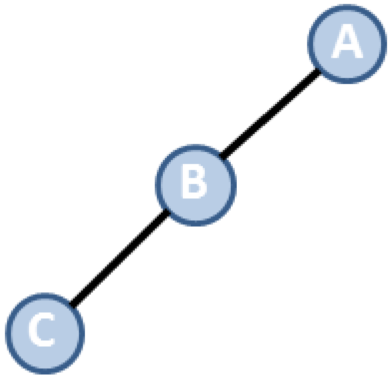
- Uma vez inserido o novo nó
  - Devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós visitados
  - Aplicar a rotação necessária para restabelecer o balanceamento da árvore se o fator de balanceamento for **+2** ou **-2**

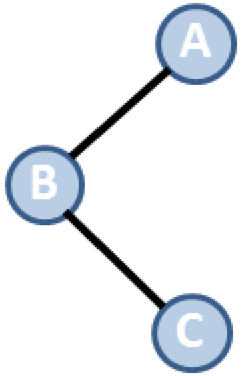
Árvore AVL | Inserção

```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...
```

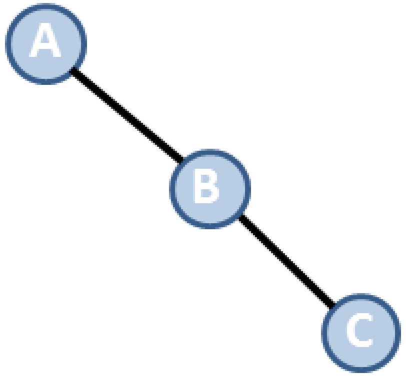
Árvore AVL | Inserção

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

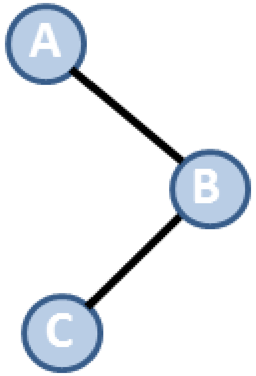




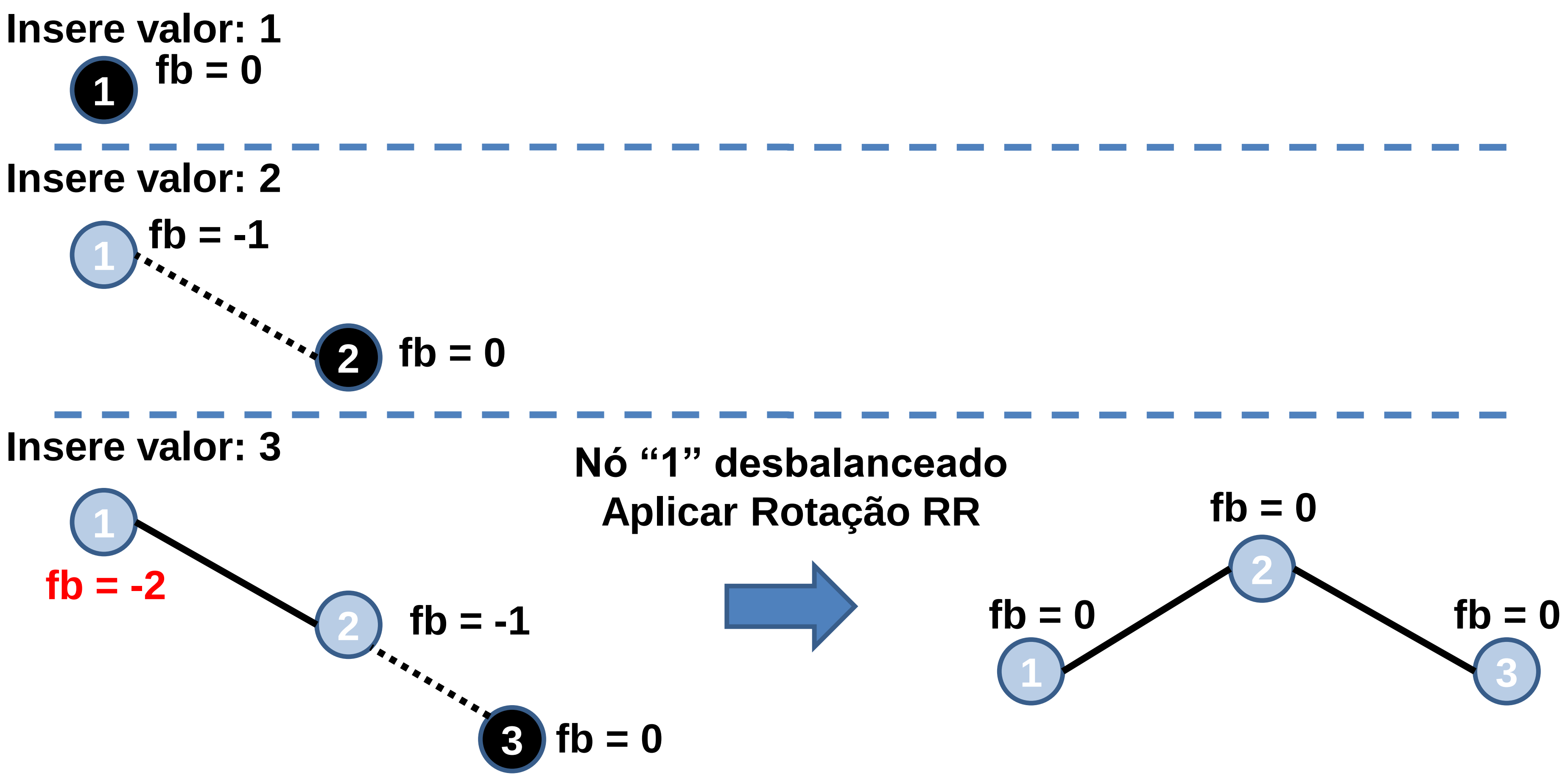
Árvore AVL | Inserção




```c
//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;
}
```

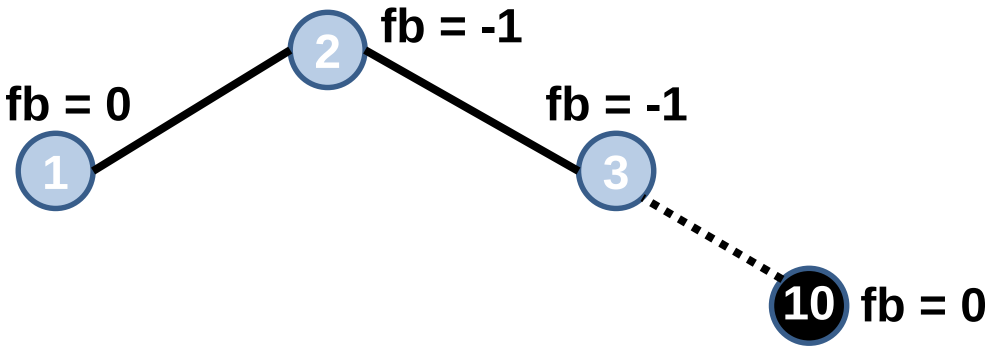


Árvore AVL | Inserção passo a passo

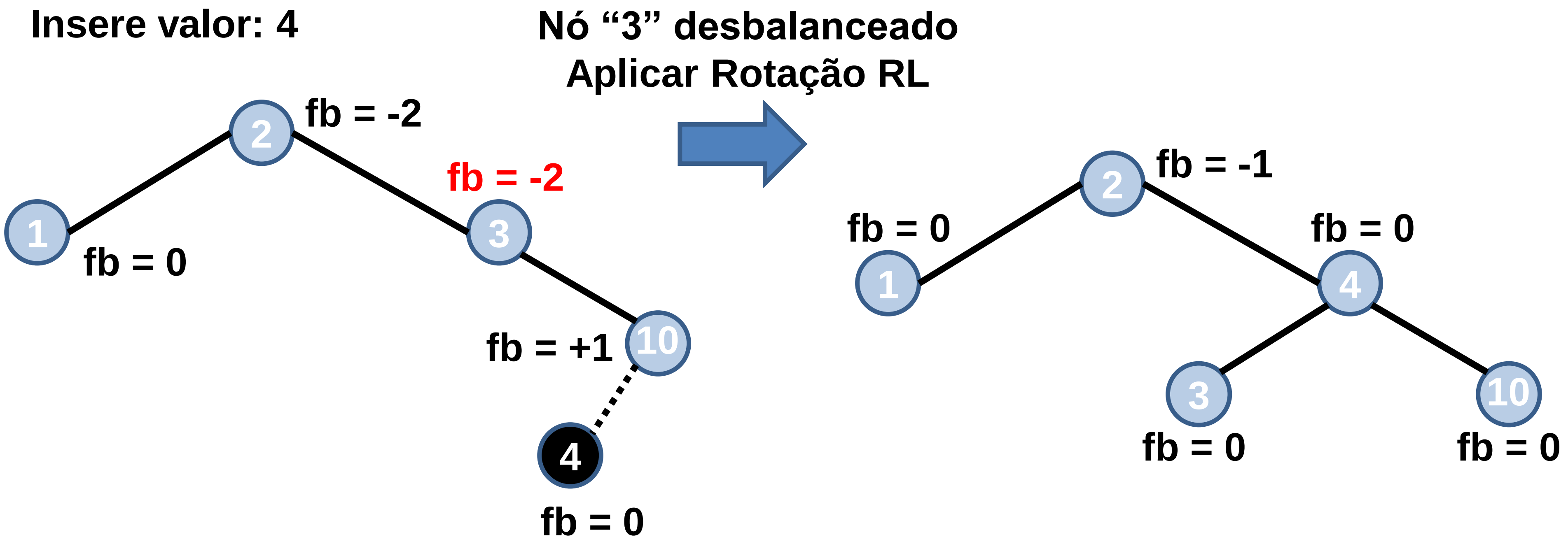


Árvore AVL | Inserção passo a passo

Insere valor: 10



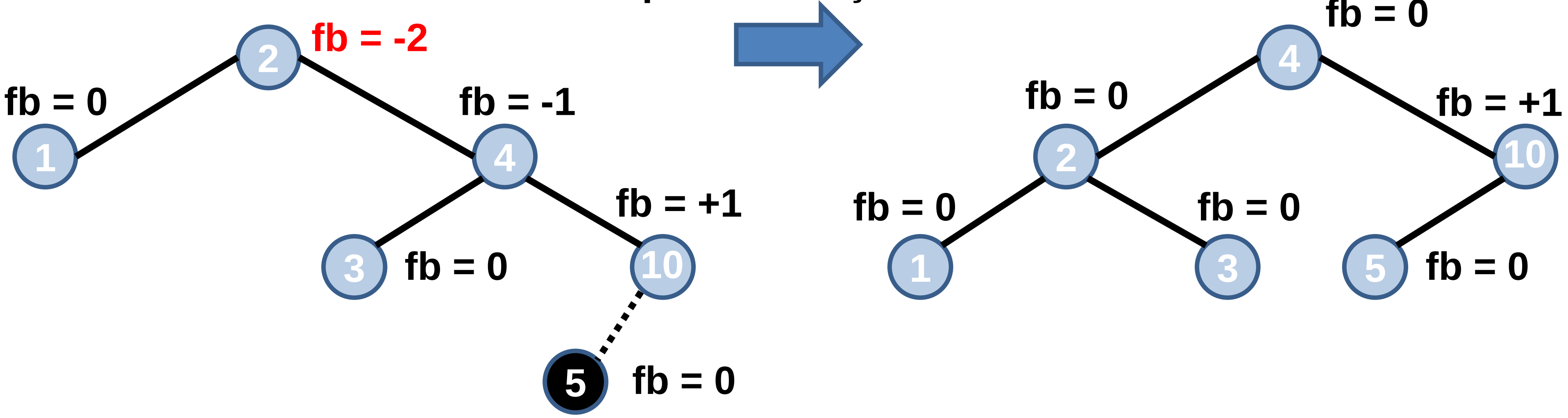
Árvore AVL | Inserção passo a passo



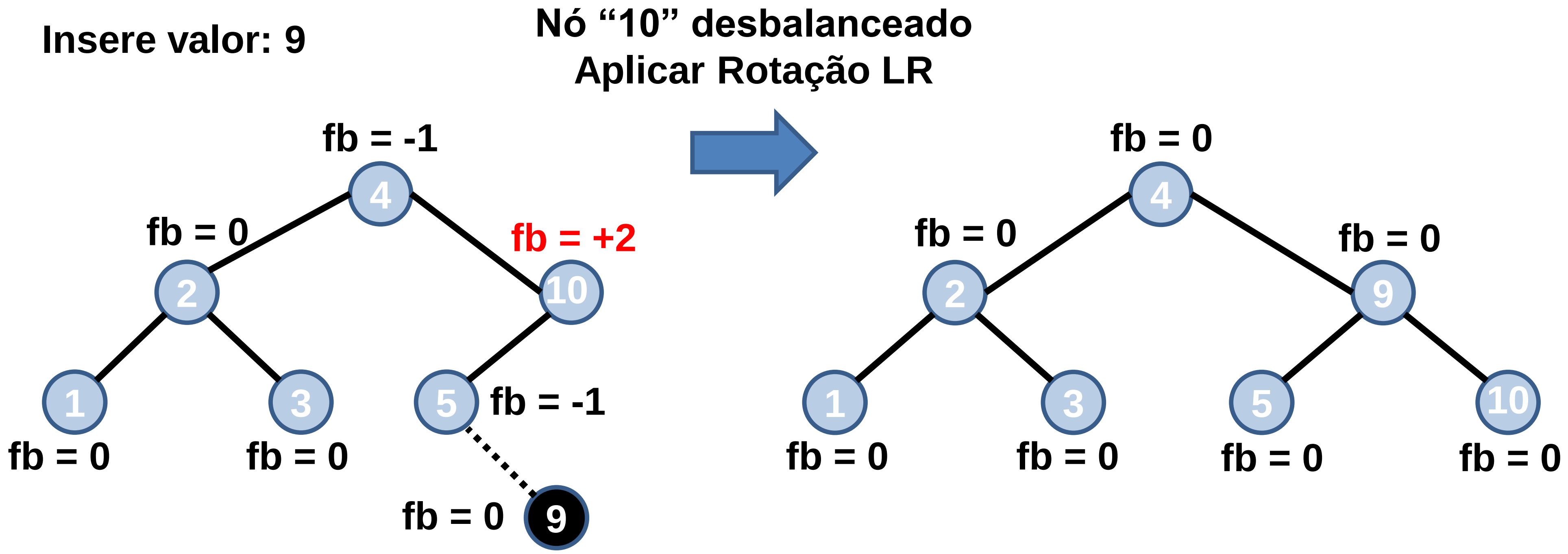
Árvore AVL | Inserção passo a passo

Insere valor: 5

Nó “2” desbalanceado
Aplicar Rotação RR



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**

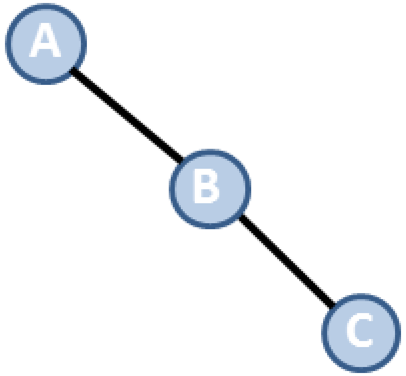
Árvore AVL | Remoção

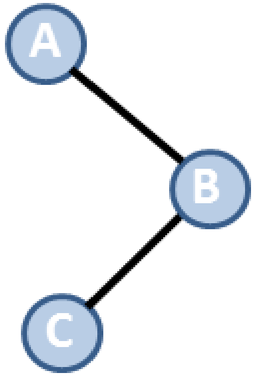
- Trabalha com 2 funções
  - Busca pelo nó
  - Remoção do nó com 2 filhos

```
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 OS
    A REMOÇÃO DE UM NÓ COM 2 FILHOS
    */
}
```

Árvore AVL | Remoção

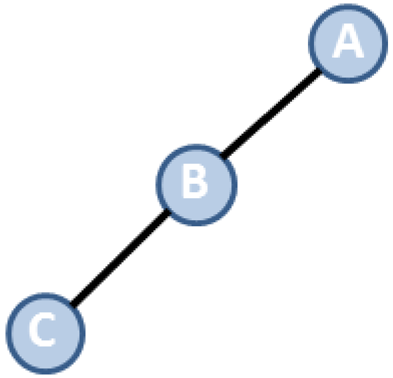
```
int remove_ArvAVL(ArvAVL *raiz, int valor){
    if(*raiz == NULL){// valor não existe
        printf("valor não existe!!\n");
        return 0;
    }
    int res;
    if(valor < (*raiz)->info){
        if((res=remove_ArvAVL(&(*raiz)->esq,valor))==1)
            if(fatorBalanceamento_NO(*raiz) >=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

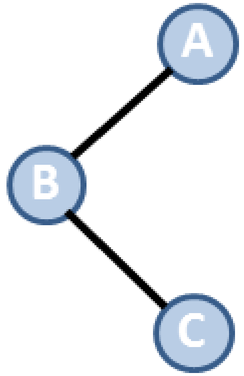




Árvore AVL | Remoção

```
//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

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

Corrige a altura

```
    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;
}
```

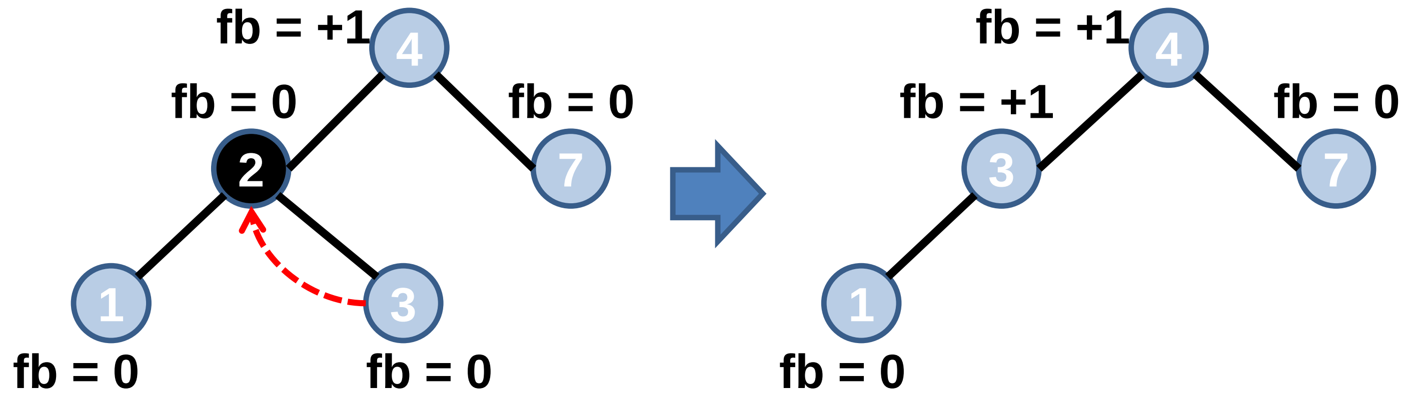
Árvore AVL | Remoção

```
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;
}
```

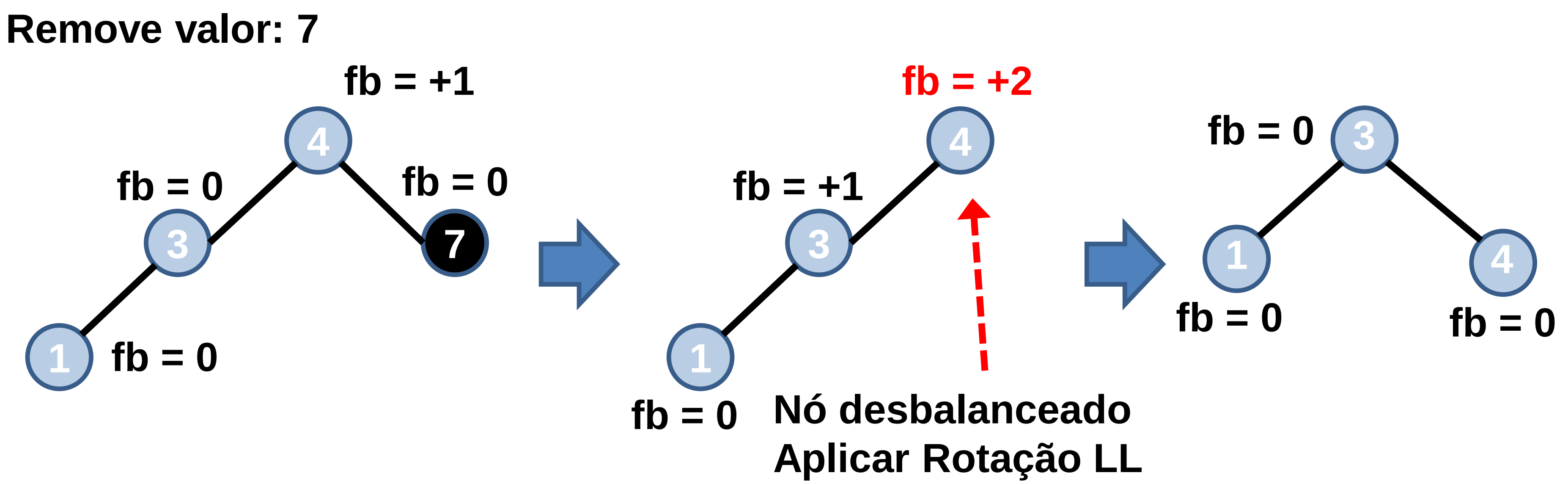
Procura pelo nó
mais a esquerda

Árvore AVL | Remoção passo a passo

Remove valor: 2



Árvore AVL | Remoção passo a passo



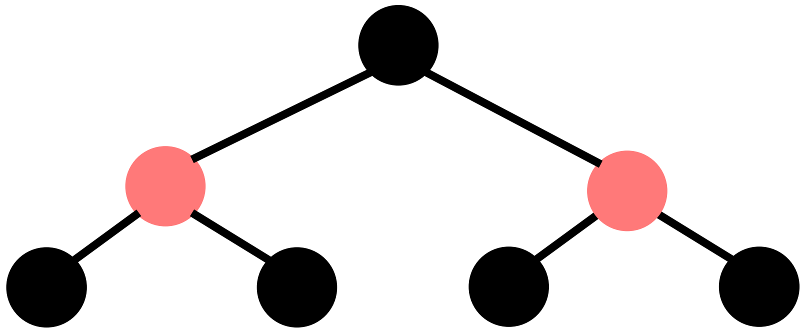
ÁRVORE RUBRO NEGRA

Árvore rubro-negra

- Também conhecida como árvore vermelho-preto ou *red-black*
  - Tipo de árvore binária balanceada
  - Originalmente criada por Rudolf Bayer em 1972
    - Chamadas de Árvores Binárias Simétricas
  - Adquiriu o seu nome atual em um trabalho de Leonidas J. Guibas e Robert Sedgewick de 1978

Árvore rubro-negra

- Utiliza um esquema de coloração dos nós para manter o balanceamento da árvore
  - Árvore AVL usa a altura das sub-árvores
- Cada nó da árvore possui um atributo de cor, que pode ser **vermelho** ou **preto**
  - Além dos dois ponteiros para seus filhos

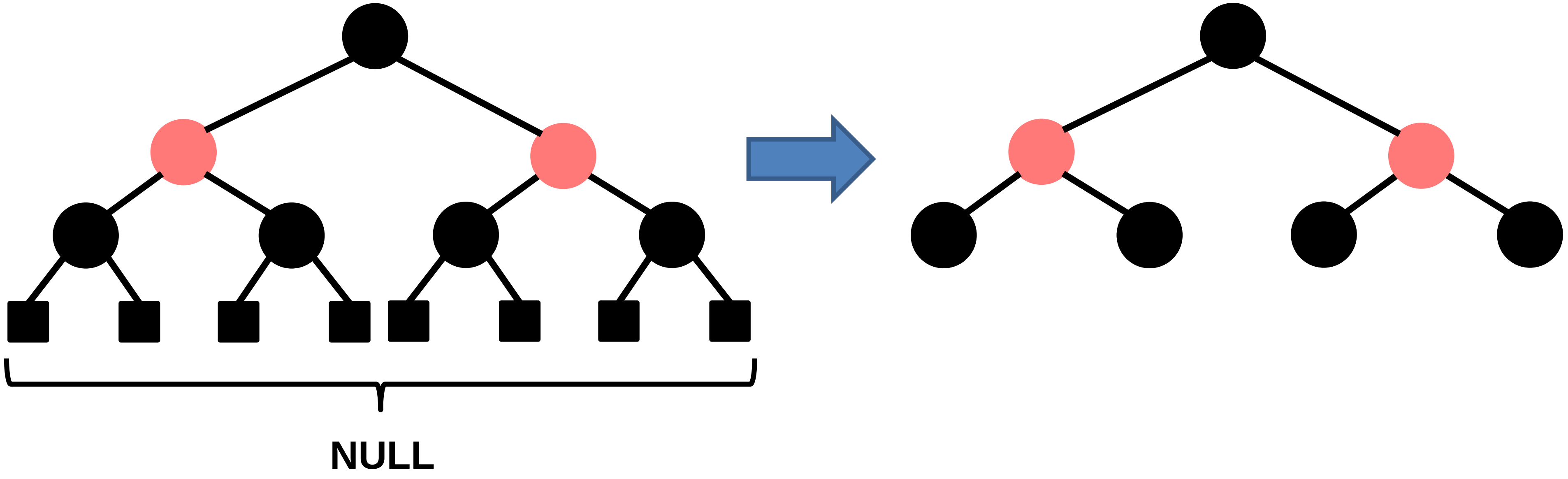


Árvore rubro-negra

- 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**

Árvore rubro-negra

- 3º propriedade
  - Como todo nó folha termina com dois ponteiros para **NULL**, eles podem ser ignorados na representação da árvore para fins de didática



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)***

AVL vs Rubro-Negra

- 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

AVL vs Rubro-Negra

- 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

AVL vs Rubro-Negra

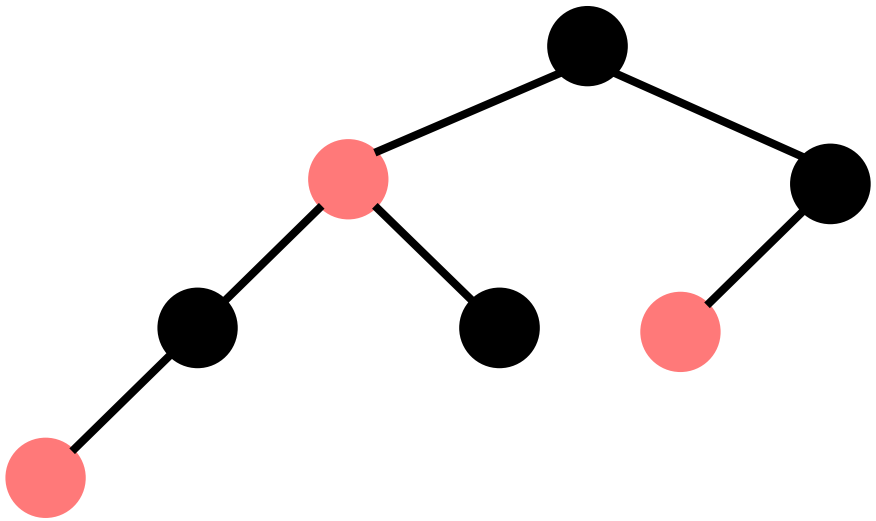
- Árvores Rubro-Negra são de uso mais geral do que as árvores AVL
  - Ela é utilizada em diversas aplicações e bibliotecas de linguagens de programação
  - Exemplos
    - **Java**: java.util.TreeMap , java.util.TreeSet
    - **C++ STL**: map, multimap, multiset
    - **Linux kernel**: completely fair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

Árvore Rubro-Negra caída para a Esquerda

- Desenvolvida por Robert Sedgewick em 2008
  - Do inglês, *left leaning red black tree*
  - Variante da árvore rubro-negra
  - Garante a mesma complexidade de operações, mas possui um implementação mais simples da inserção e remoção

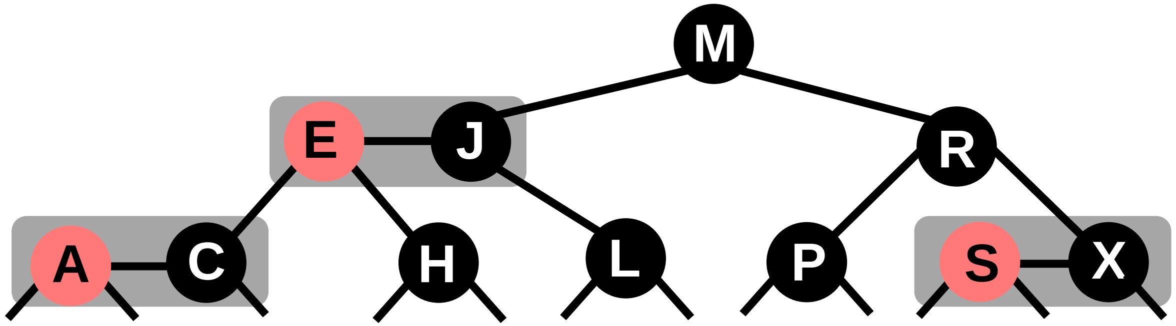
Árvore Rubro-Negra caída para a Esquerda

- Possui uma propriedade extra além das propriedades da árvore convencional,
  - Se um nó é **vermelho**, então ele é o filho esquerdo do seu pai
  - Aspecto de caída para a esquerda



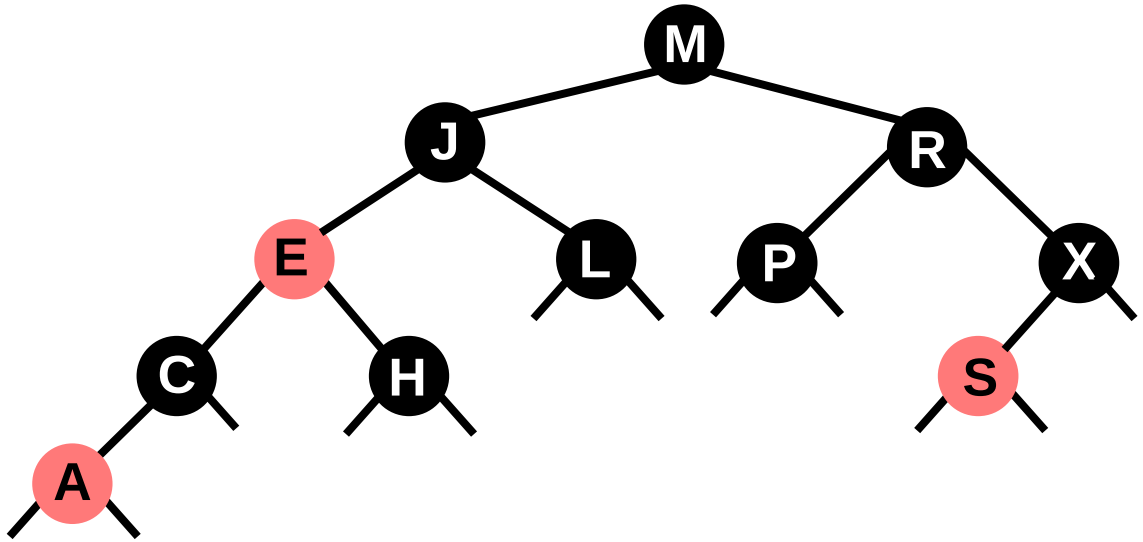
Árvore Rubro-Negra caída para a Esquerda

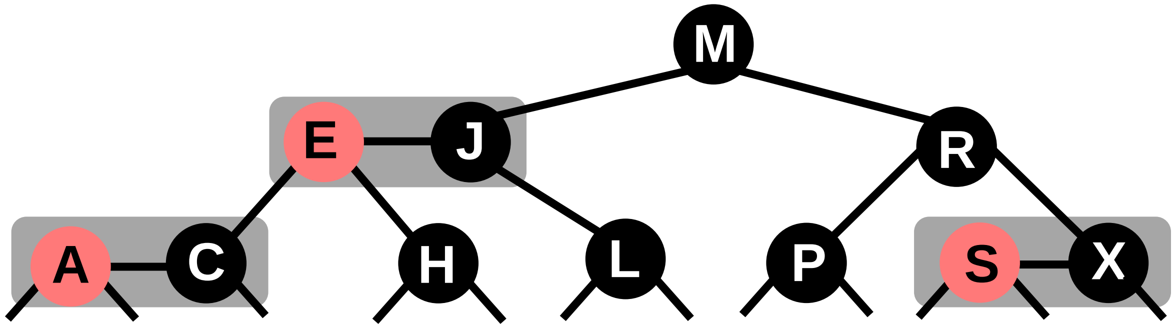
- Sua implementação corresponde a de uma **árvore 2-3**
  - A árvore 2-3 não é uma árvore binária
    - Cada nó interno pode armazenar um ou dois valores
    - Pode ter dois (um valor) ou três (dois valores) filhos
    - Seu funcionamento é o mesmo da árvore binária de busca



Árvore Rubro-Negra caída para a Esquerda

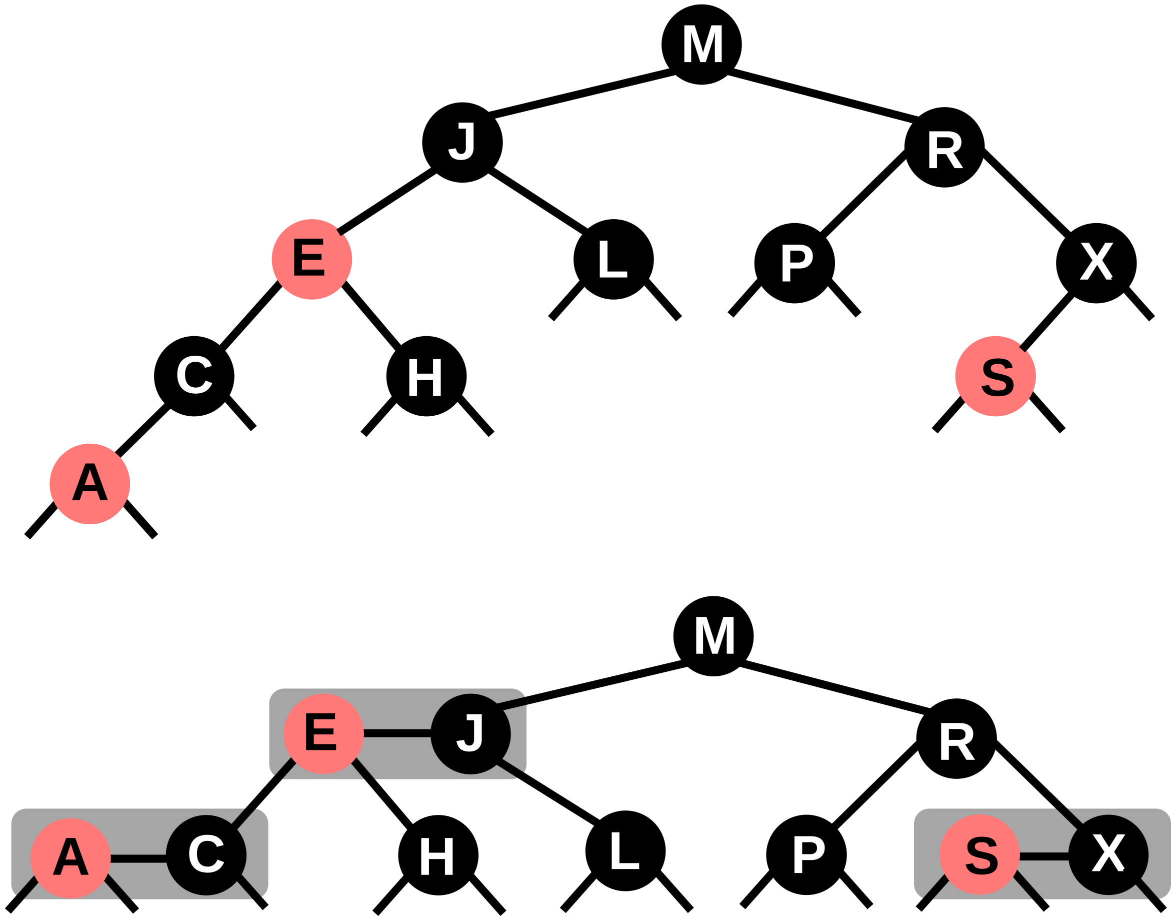
- Neste caso, o nó vermelho será sempre o valor menor de um nó contendo dois valores e três sub-árvores





Árvore Rubro-Negra caída para a Esquerda

- Balancear a árvore rubro-negra equivale a manipular uma árvore 2-3, uma tarefa muito mais simples



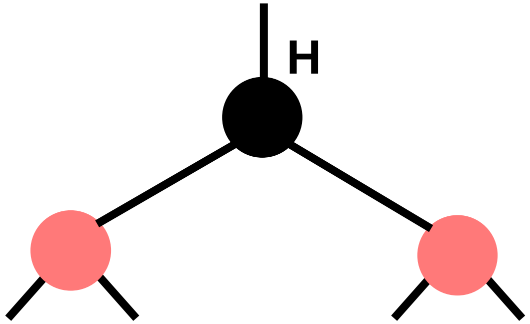
Árvore Rubro Negra | TAD

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

```c
//Arquivo ArvoreLLRB.h
typedef struct NO* ArvLLRB;

//Arquivo ArvoreLLRB.c
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreLLRB.h" //inclui os Protótipos

#define RED 1 //define as cores
#define BLACK 0

struct NO{
    int info;
    struct NO *esq;
    struct NO *dir;
    int cor;
};
//programa principal
ArvLLRB* raiz; //ponteiro para ponteiro
```

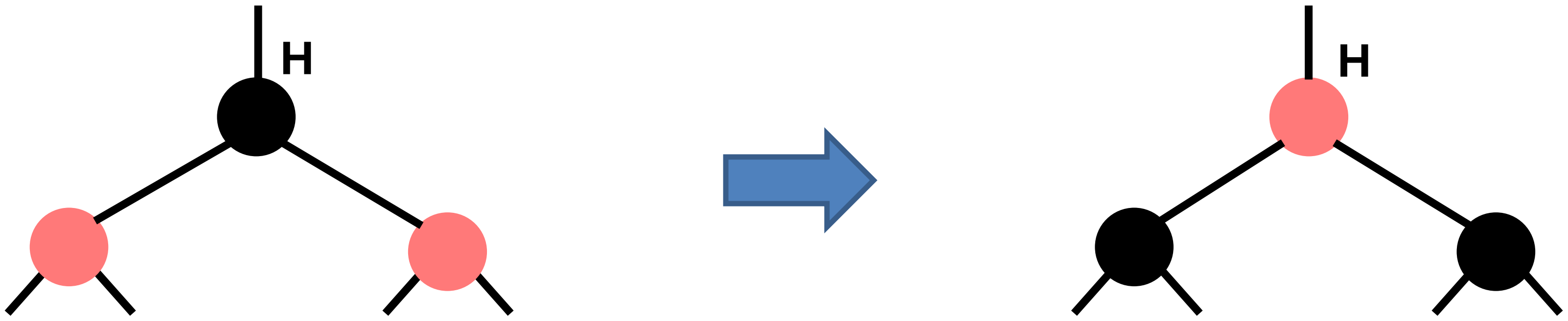
Troca das cores dos nós

- Durante o balanceamento da árvore
  - Necessidade de mudar a cor de um nó e de seus filhos de **vermelho** para **preto** ou vice-versa
  - Exemplo: um nó possui dois filhos **vermelhos**
    - Violação de uma das propriedades da árvore



Troca das cores dos nós

- Operação de mudança de cor
  - Não altera o número de nós pretos da raiz até os nós folhas.
  - Problema: pode introduzir dois nós consecutivos **vermelhos** na árvore
    - Deve ser corrigido com outras operações



Árvore Rubro Negra | TAD

- Acessando a cor e trocando as cores

```c
//Funções auxiliares
//Acessando a cor de um nó
int cor(struct NO* H){
    if(H == NULL)
        return BLACK;
    else
        return H->cor;
}
//Inverte a cor do pai e de seus filhos
//É uma operação "administrativa": não altera
//a estrutura ou conteúdo da árvore
void trocaCor(struct NO* H){
    H->cor = !H->cor;
    if(H->esq != NULL)
        H->esq->cor = !H->esq->cor;
    if(H->dir != NULL)
        H->dir->cor = !H->dir->cor;
}
```

Rotações

- Árvore AVL
  - Utiliza quatro funções de rotação para rebalancear a árvore
- Árvore rubro-negra
  - Possui apenas duas funções de rotação
    - Rotação à Esquerda
    - Rotação à Direita

Rotações

- Funcionamento
  - Dado um conjunto de três nós, visa deslocar um nó **vermelho** que esteja à **esquerda** para à **direita** e vice-versa.
  - Mais simples de implementar e de depurar em comparação com as rotações da árvore AVL
    - As operações de rotação apenas atualizam ponteiros
    - Complexidade é ***O(1)***

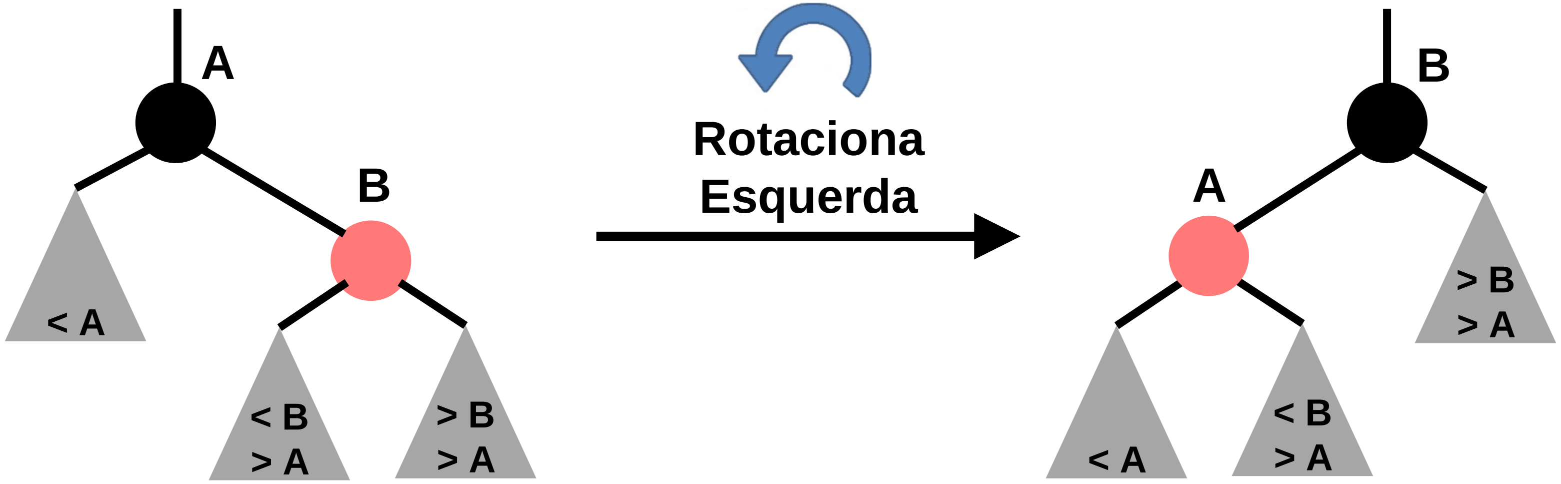
Rotações

- Rotação à Esquerda
  - Recebe um nó **A** com **B** como filho **direito**
  - Move **B** para o lugar de **A**, **A** se torna o filho **esquerdo** de **B**
  - **B** recebe a cor de **A**, **A** fica **vermelho**

```c
struct NO* rotacionaEsquerda(struct NO* A){
    struct NO* B = A->dir;
    A->dir = B->esq;
    B->esq = A;
    B->cor = A->cor;
    A->cor = RED;
    return B;
}
```

Rotações

- Rotação à Esquerda
  - Recebe um nó **A** com **B** como filho **direito**
  - Move **B** para o lugar de **A**, **A** se torna o filho **esquerdo** de **B**
  - **B** recebe a cor de **A**, **A** fica **vermelho**



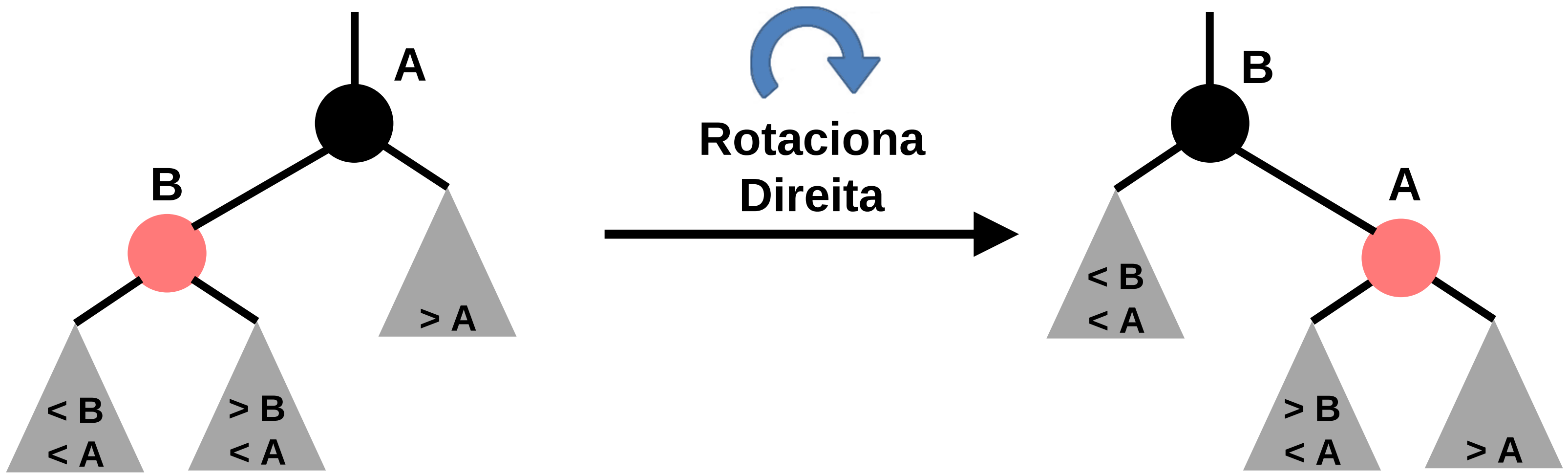
Rotações

- Rotação à Direita
  - Recebe um nó **A** com **B** como filho **esquerdo**
  - Move **B** para o lugar de **A**, **A** se torna o filho **direito** de **B**
  - **B** recebe a cor de **A**, **A** fica **vermelho**

```c
struct NO* rotacionaDireita(struct NO* A){
    struct NO* B = A->esq;
    A->esq = B->dir;
    B->dir = A;
    B->cor = A->cor;
    A->cor = RED;
    return B;
}
```

Rotações

- Rotação à Direita
  - Recebe um nó **A** com **B** como filho **esquerdo**
  - Move **B** para o lugar de **A**, **A** se torna o filho **direito** de **B**
  - **B** recebe a cor de **A**, **A** fica vermelho



Árvore rubro-negra | Inserção

- Similar a inserção na árvore AVL
- 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 rubro-negra | Inserção

- **Importante**
  - Todo nó inserido é inicialmente **vermelho**
- Uma vez inserido o novo nó
  - Devemos voltar pelo caminho percorrido e verificar se ocorreu a violação de alguma das propriedades da árvore para cada um dos nós visitados
  - Aplicar uma das rotações ou mudança de cores para restabelecer o balanceamento da árvore

Árvore rubro-negra | Inserção

- Função que gerencia o nó raiz após a inserção

```c
//arquivo ArvoreLLRB.c
int insere_ArvLLRB(ArvLLRB* raiz, int valor){
    int resp;
    //FUNÇÃO RESPONSÁVEL PELA BUSCA DO LOCAL
    //DE INSERÇÃO DO NÓ
    *raiz = insereNO(*raiz,valor,&resp);
    if((*raiz) != NULL)
        (*raiz)->cor = BLACK;

    return resp;
}
```

Árvore rubro-negra | Inserção

```c
struct NO* insereNO(struct NO* H,int valor,int *resp){
    if(H == NULL){
      struct NO *novo
      novo = (struct NO*)malloc(sizeof(struct NO));
      if(novo == NULL){
        *resp = 0;
        return NULL;
      }
      novo->info = valor;
      novo->cor = RED;
      novo->dir = NULL;
      novo->esq = NULL;
      *resp = 1;
      return novo;
    }
     //continua...
```

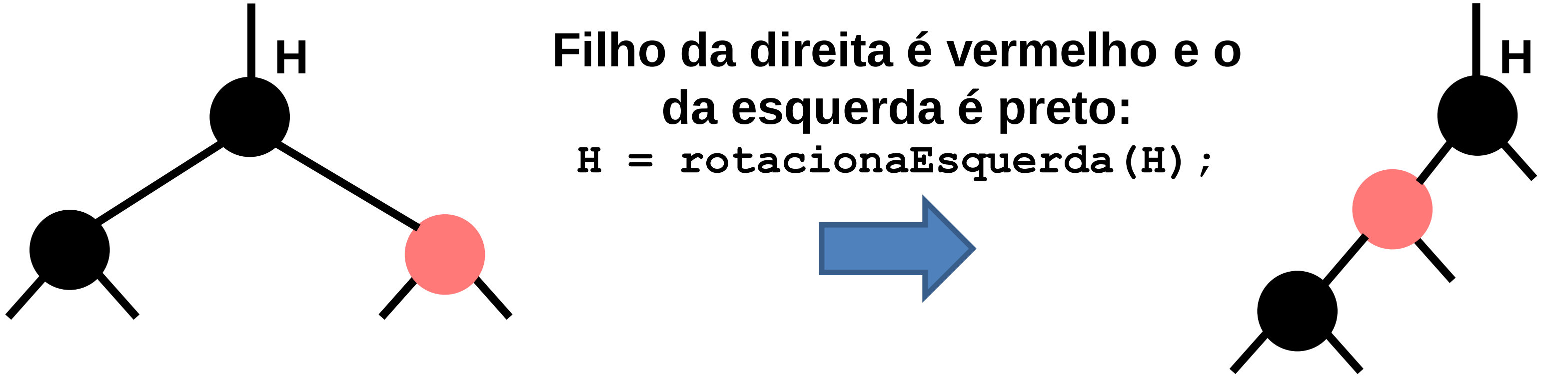
Árvore rubro-negra | Inserção

Corrige violações de propriedades

```
//continuação
if(valor == H->info)
  *resp = 0;// Valor duplicado
else{
  if(valor < H->info)
      H->esq = insereNO(H->esq,valor,resp);
  else
      H->dir = insereNO(H->dir,valor,resp);
}

if(cor(H->dir) == RED && cor(H->esq) == BLACK)
  H = rotacionaEsquerda(H);

if(cor(H->esq) == RED && cor(H->esq->esq) == RED)
  H = rotacionaDireita(H);

if(cor(H->esq) == RED && cor(H->dir) == RED)
  trocaCor(H);

return H;
}
```

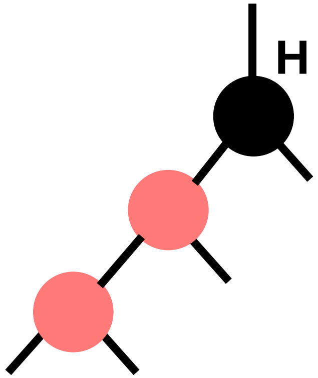
Árvore rubro-negra | Inserção

- Violações das propriedades na inserção
  - Filho da direita é **vermelho** e o filho da esquerda é **preto**
    - Solução: Rotação à esquerda



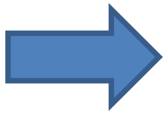
Árvore rubro-negra | Inserção

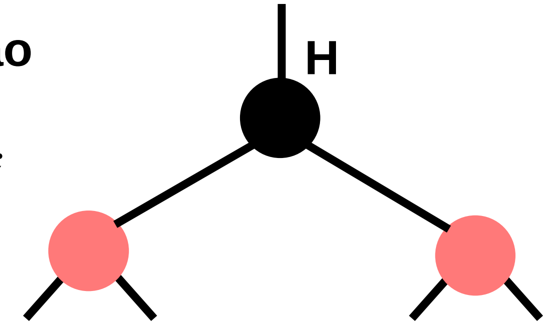
- Violações das propriedades na inserção
  - Filho da esquerda é **vermelho** e o filho à esquerda do filho da esquerda também é **vermelho**
    - Solução: Rotação à direita



Filho e neto da esquerda são vermelhos:

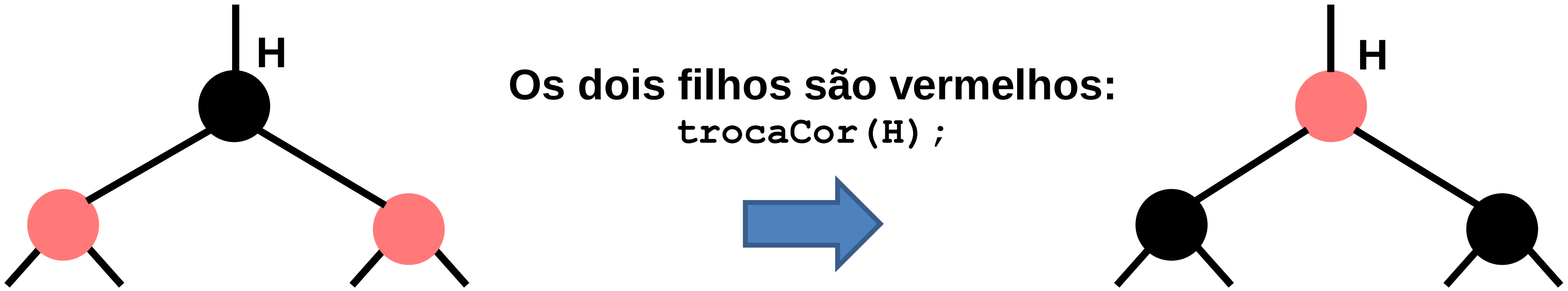
```
H = rotacionaDireita(H);
```





Árvore rubro-negra | Inserção

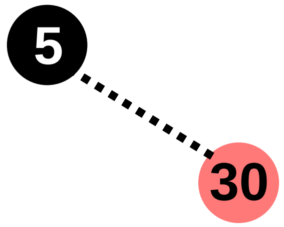
- Violações das propriedades na inserção
  - Ambos os filhos são **vermelhos**
    - Solução: troca de cores



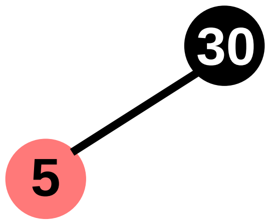
Árvore rubro-negra | Inserção passo a passo

Insere valor: 1

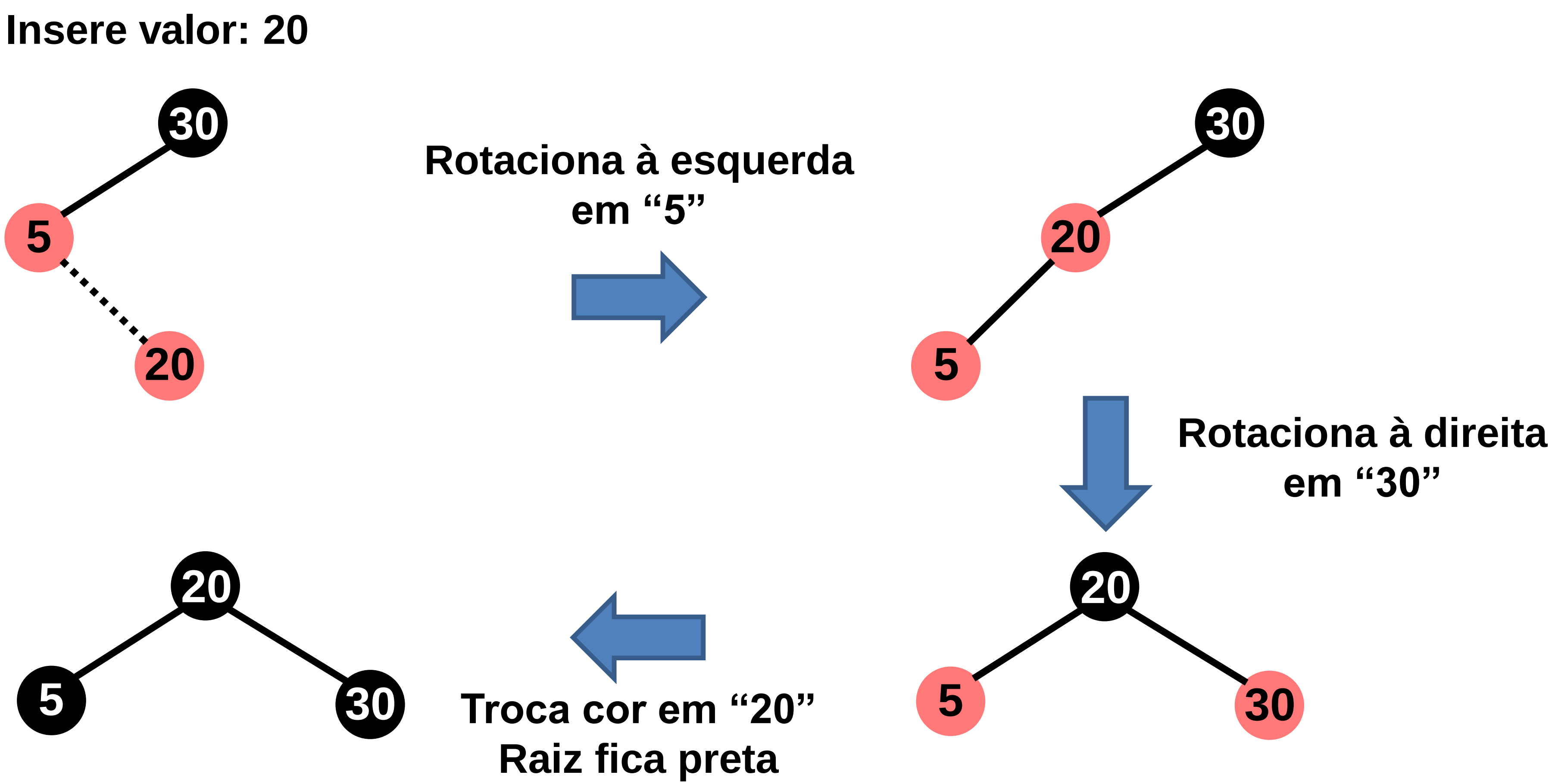
Insere valor: 30



Rotaciona à esquerda em “5”

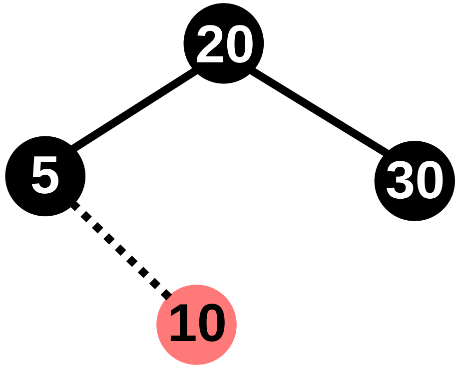


Árvore rubro-negra | Inserção passo a passo

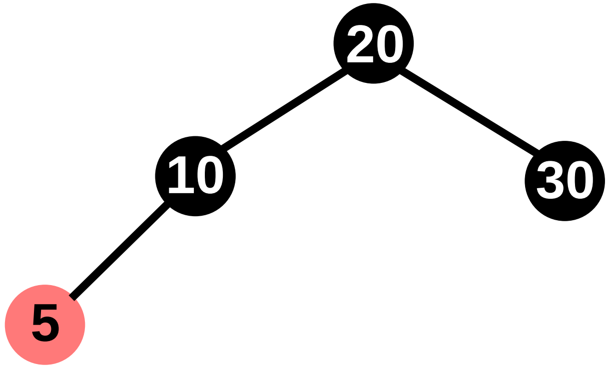


Árvore rubro-negra | Inserção passo a passo

Insere valor: 10



Rotaciona à esquerda em “5”



Árvore rubro-negra | 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 rubro-negra | Remoção

- Uma vez removido o nó
  - Devemos voltar pelo caminho percorrido e verificar se ocorreu a violação de alguma das propriedades da árvore para cada um dos nós visitados
  - Aplicar uma das rotações ou mudança de cores para restabelecer o balanceamento da árvore

Árvore rubro-negra | Remoção

- Diferença com relação a árvore AVL
  - A remoção na árvore rubro-negra corrige o balanceamento da árvore tanto na ida quanto na volta da recursão
    - O processo de busca pelo nó a ser removido já prevê possíveis violações das propriedades da árvore
    - Somente devemos executar a remoção se o nó a ser removido realmente existe na árvore

Árvore rubro-negra | Remoção

- Verificar se é possível antes de remover
- Também gerencia o nó raiz após a remoção

```c
//arquivo ArvoreLLRB.c
int remove_ArvLLRB(ArvLLRB *raiz, int valor){
    if(consulta_ArvLLRB(raiz,valor)){
        struct NO* h = *raiz;
        //FUNÇÃO RESPONSÁVEL PELA BUSCA
        //DO NÓ A SER REMOVIDO
        *raiz = remove_NO(h,valor);
        if(*raiz != NULL)
            (*raiz)->cor = BLACK;
        return 1;
    }else
        return 0;
}
```

Árvore rubro-negra | Remocão

- Uso de várias funções auxiliares

```
struct NO* remove_NO(struct NO* H, int valor){
    if(valor < H->info){
        if(cor(H->esq) == BLACK && cor(H->esq->esq)==BI
            H = move2EsqRED(H);

        H->esq = remove_NO(H->esq, valor);
    }else{
        if(cor(H->esq) == RED)
            H = rotacionaDireita(H);

        if(valor == H->info && (H->dir == NULL)){
            free(H);
            return NULL;
        }

        if(cor(H->dir) == BLACK && cor(H->dir->esq)==BI
            H = move2DirRED(H);

        if(valor == H->info){
            struct NO* x = procuraMenor(H->dir);
            H->info = x->info;
            H->dir = removerMenor(H->dir);
        }else
            H->dir = remove_NO(H->dir, valor);
    }
    return balancear(H);
}
```

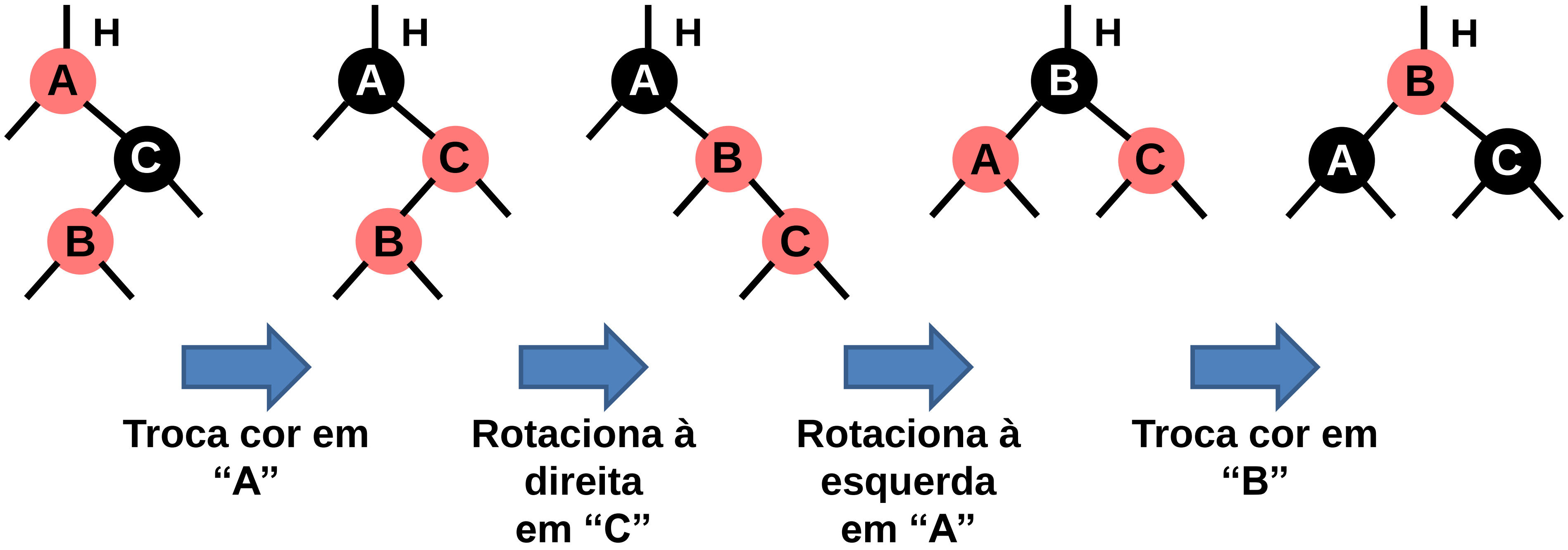
Função *move2EsqRED*

- Move um nó **vermelho** para a esquerda
  - Troca as cores do nó **H** e seus filhos
  - Filho a **esquerda** do filho **direito** é **vermelho**
    - Rotação à **direita** no **filho direito** e à **esquerda** no **pai**
  - Troca as cores do nó pai e de seus filhos

```c
struct NO* move2EsqRED(struct NO* H){
    trocaCor(H);
    if(cor(H->dir->esq) == RED){
        H->dir = rotacionaDireita(H->dir);
        H = rotacionaEsquerda(H);
        trocaCor(H);
    }
    return H;
}
```

Função *move2EsqRED*

- Move um nó **vermelho** para a esquerda



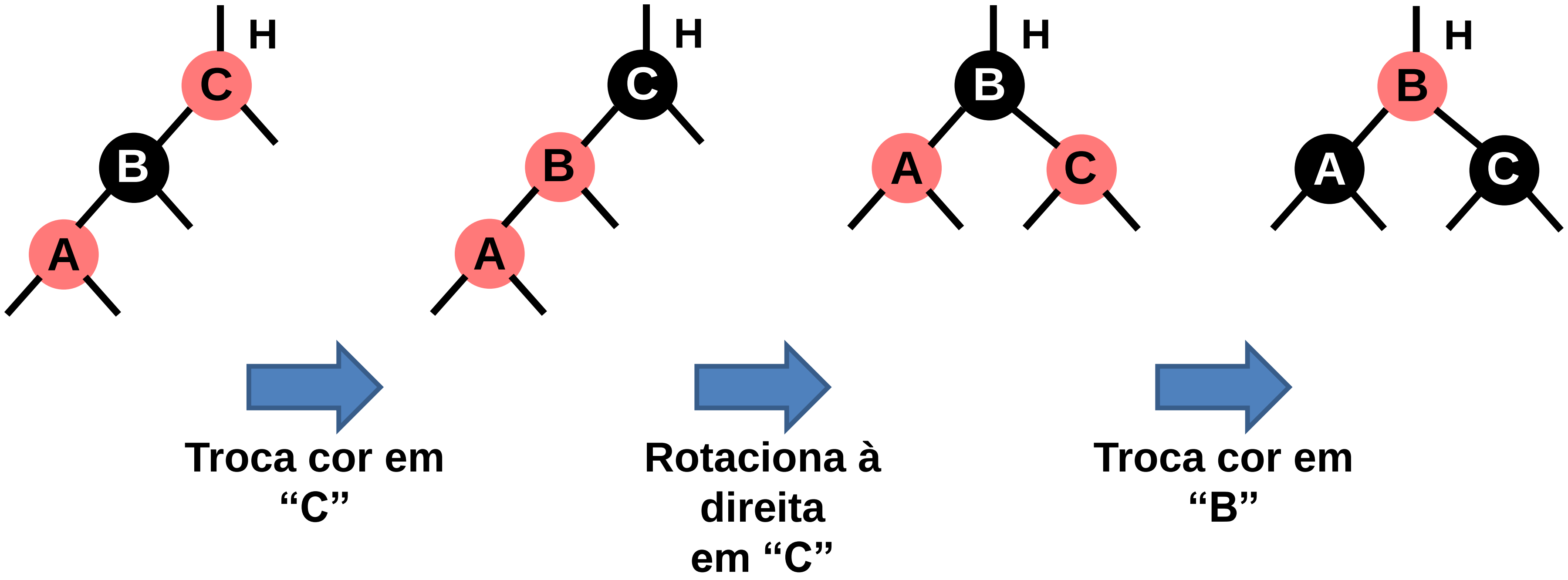
Função *move2DirRED*

- Move um nó **vermelho** para a direita
  - Troca as cores do nó **H** e seus filhos
  - Filho a **esquerda** do filho **esquerdo** é **vermelho**
    - Rotação à **direita** no **pai**
  - Troca as cores do nó pai e de seus filhos

```
struct NO* move2DirRED(struct NO* H){
    trocaCor(H);
    if(cor(H->esq->esq) == RED){
        H = rotacionaDireita(H);
        trocaCor(H);
    }
    return H;
}
```

Função *move2DirRED*

- Move um nó **vermelho** para a direita



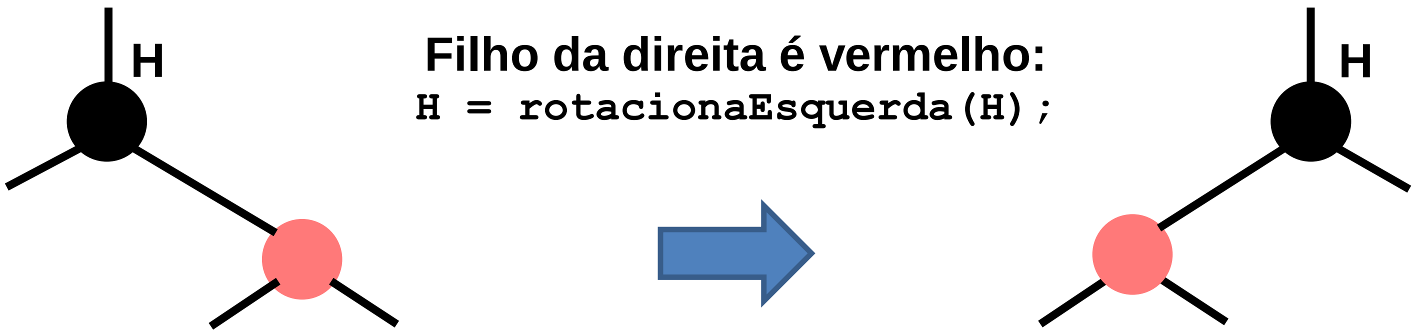
Função *balancear*

- Trata várias violações de propriedades

```
struct NO* balancear(struct NO* H){
    //nó Vermelho é sempre filho à esquerda
    if(cor(H->dir) == RED)
        H = rotacionaEsquerda(H);

    //Filho da esquerda e neto da esquerda são vermelhos
    if(H->esq != NULL && cor(H->esq) == RED && cor(H->esq->esq) == RED)
        H = rotacionaDireita(H);

    //2 filhos Vermelhos: troca cor!
    if(cor(H->esq) == RED && cor(H->dir) == RED)
        trocaCor(H);

    return H;
}
```

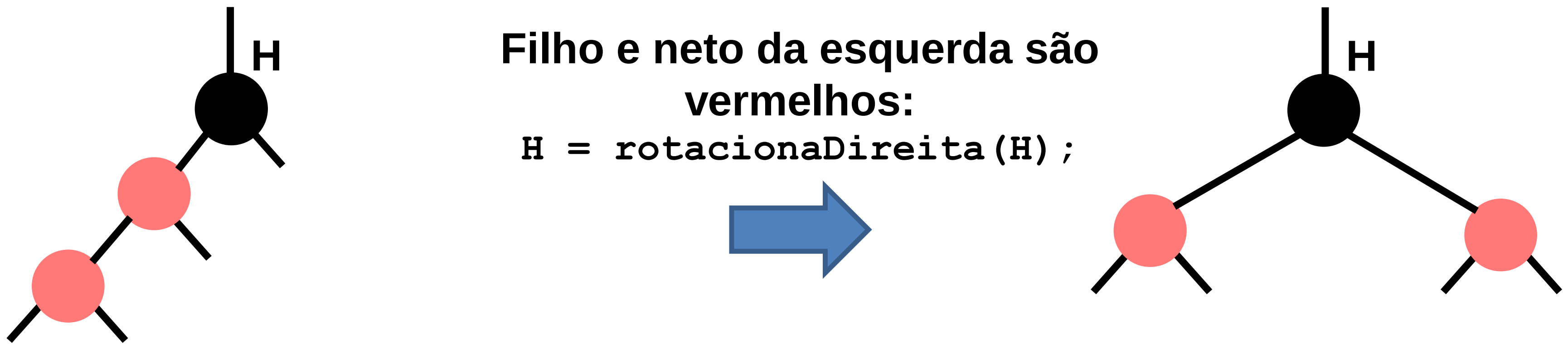
Função *balancear*

- Violações das propriedades na remoção
  - Nó da direita é **vermelho**
    - Solução: rotação à esquerda



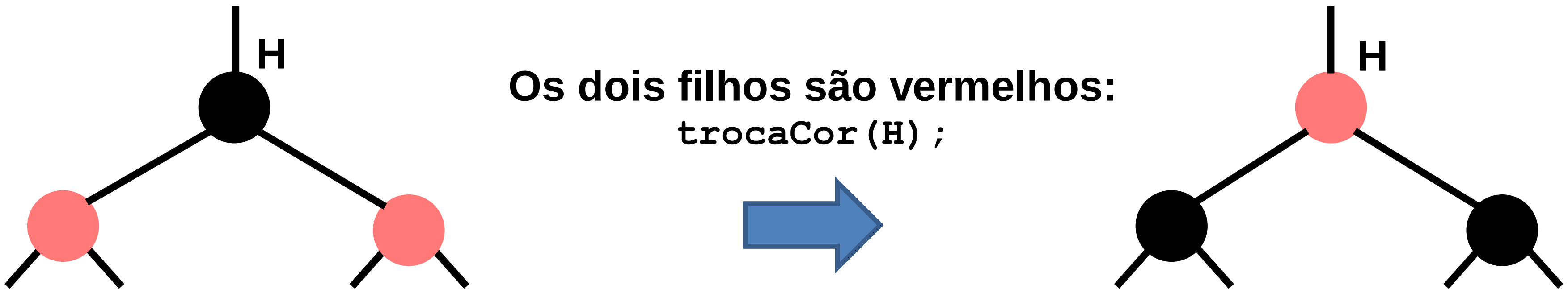
Função *balancear*

- Violações das propriedades na remoção
  - Filho da esquerda e neto da esquerda são **vermelhos**
    - Solução: rotação à direita



Função *balancear*

- Violações das propriedades na remoção
  - Ambos os filhos são **vermelhos**
    - Solução: troca de cores



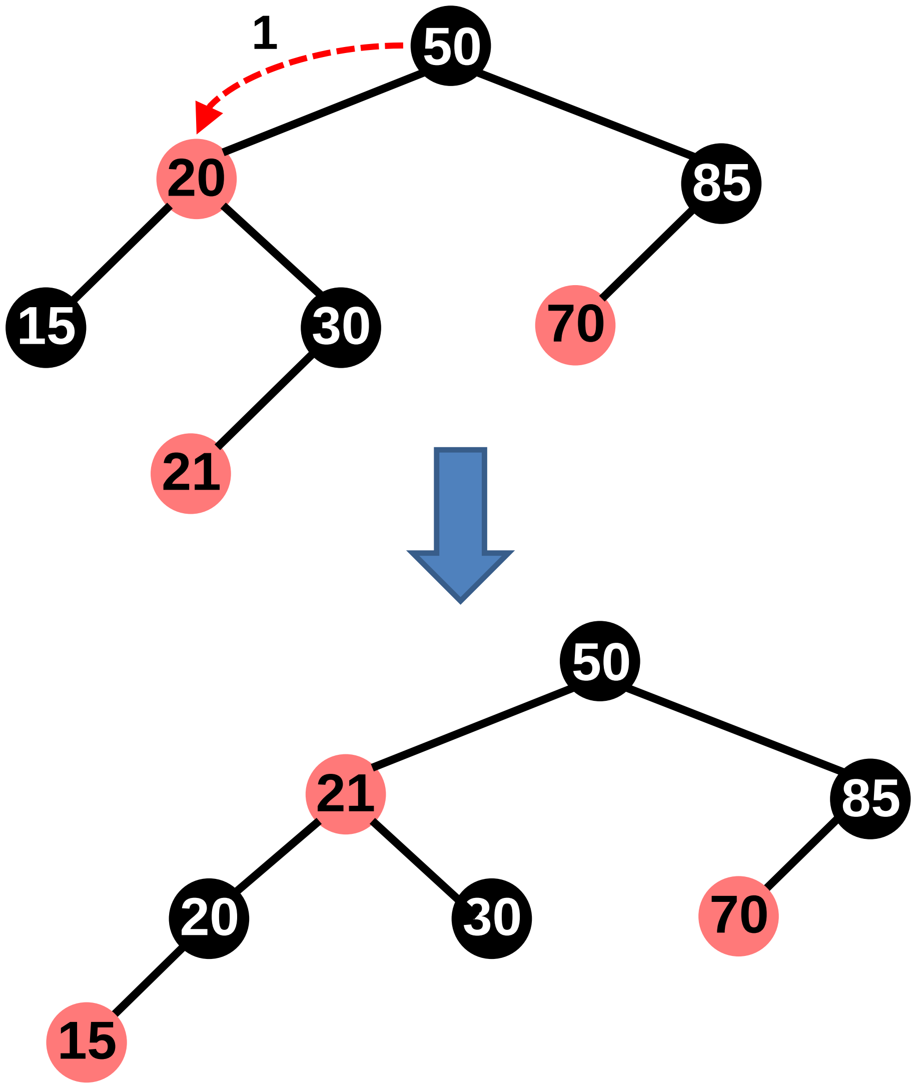
Funções *procuraMenor* e *removerMenor*

- Nó removido possui filhos

```
struct NO* removerMenor(struct NO* H){
    if(H->esq == NULL){
        free(H);
        return NULL;
    }
    if(cor(H->esq) == BLACK && cor(H->esq->esq)==BLACK)
        H = move2EsqRED(H);

    H->esq = removerMenor(H->esq);
    return balancear(H);
}
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

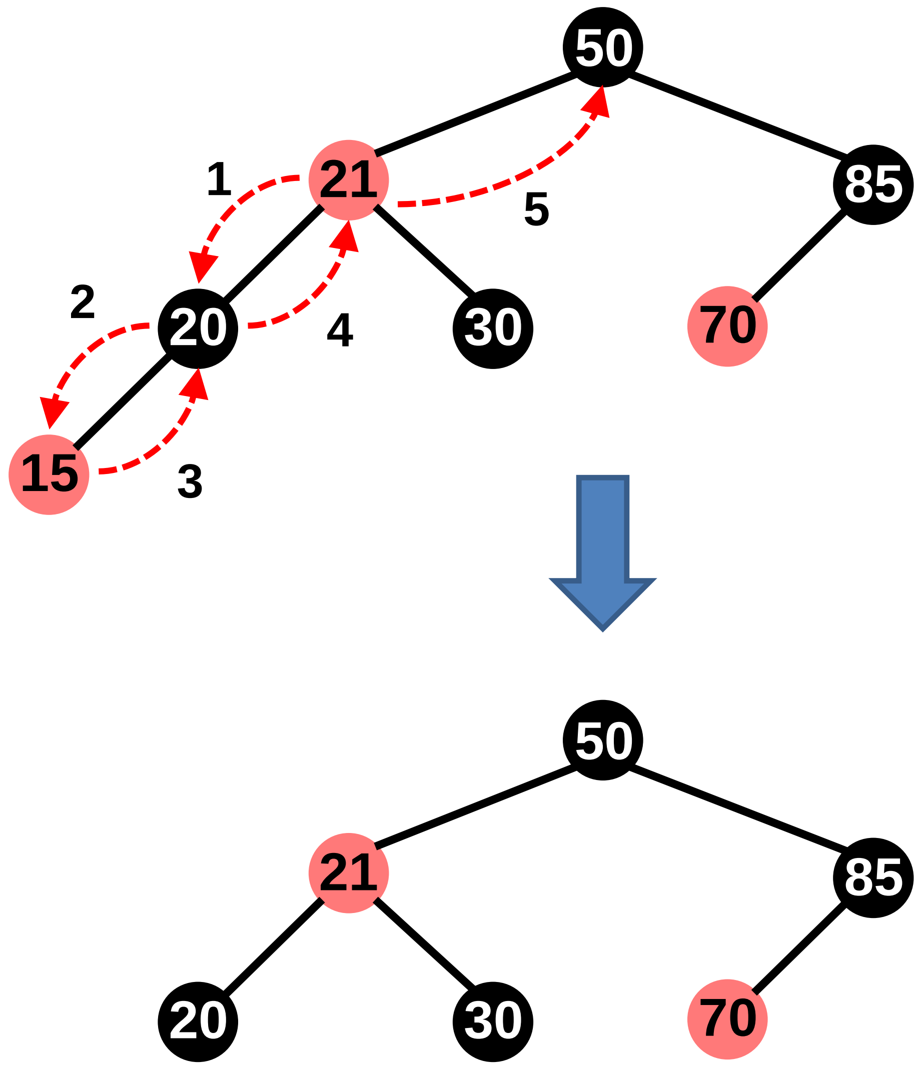
Árvore rubro-negra | Remoção passo a passo



Remove valor: 15

	Inicia a busca pelo nó a ser removido a partir do nó "50"
1	Nó procurado é menor do que 50. Visita nó "20"
	Nó "20" tem filho e neto (NULL) da cor preta à ESQUERDA. Chama a função move2EsqRED()

Árvore rubro-negra | Remoção passo a passo



	Continua a busca a partir do nó “21”
1	Nó procurado é menor do que 21. Visita nó “20”
2	Nó procurado é menor do que 20. Visita nó “15”
3	Nó a ser removido foi encontrado. Libera o nó e volta para o nó “20”
4	Balanceamento no “20” está OK. Volta para o nó “21”
5	Balanceamento no “21” está OK. Volta para o nó “50”
	Balanceamento no “50” está OK. Processo de remoção termina

Material Complementar | Vídeo Aulas

- Aula 78 – Árvores Balanceadas:
  - youtu.be/Au-6c55J90c
- Aula 79: Árvore AVL: Definição:
  - youtu.be/4eO3UbTiRyo
- Aula 80: Árvore AVL: Implementação:
  - youtu.be/l5cl39jdnow
- Aula 81: Árvore AVL: Tipos de Rotação:
  - youtu.be/1HkWqH7L2rU
- Aula 82: Árvore AVL: Implementando as Rotações:
  - youtu.be/6OJ8stXwdq0
- Aula 83: Árvore AVL: Inserção:
  - youtu.be/lQsVUxa3Auk
- Aula 84: Árvore AVL: Remoção:
  - youtu.be/F7_Daymw-WM

Material Complementar | Vídeo Aulas

- Aula 105: Árvore Rubro Negra – Definição:
  - youtu.be/DaWNuijRRFY
- Aula 106: Árvore Rubro Negra Caída para a Esquerda (LLRB):
  - youtu.be/TYBTOay_i3g
- Aula 107: Implementando uma Árvore Rubro Negra:
  - youtu.be/_lTz-ePzWjk
- Aula 108: Rotação da Árvore Rubro Negra LLRB:
  - youtu.be/Pa8PI6o09Ic
- Aula 109: Movendo os nós vermelhos:
  - youtu.be/lo6Zk7zXOww
- Aula 110: Inserção na Árvore Rubro-Negra – LLRB:
  - youtu.be/L4gWuqpvk4E
- Aula 111: Remoção na Árvore Rubro-Negra – LLRB:
  - youtu.be/p5aukRcjdqc

Material Complementar | GitHub

- https://github.com/arbackes

Popular repositories

Livro_Python — Public

118 · 12

Estrutura-de-Dados-em-C — Public

Códigos fontes de diversas estrutura de dados implementadas em linguagem C

C · 49 · 4

Estrutura-de-Dados-em-Python — Public

Códigos fontes de diversas estrutura de dados implementadas em linguagem Python

Python · 9 · 2

Codigos-Fontes-Diversos-em-C — Public

Códigos fontes de diferentes projetos implementados em linguagem C

C · 7 · 1