



Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – ALGORITMOS E ESTRUTURAS DE DADOS I

TAD – ÁRVORES VERMELHO-PRETO

Prof. Mateus Ferreira Satler

Índice

1

• Introdução

2

• Árvore Rubro-Negra Esquerdista

3

• Inserção

4

• Remoção

5

• Análise

6

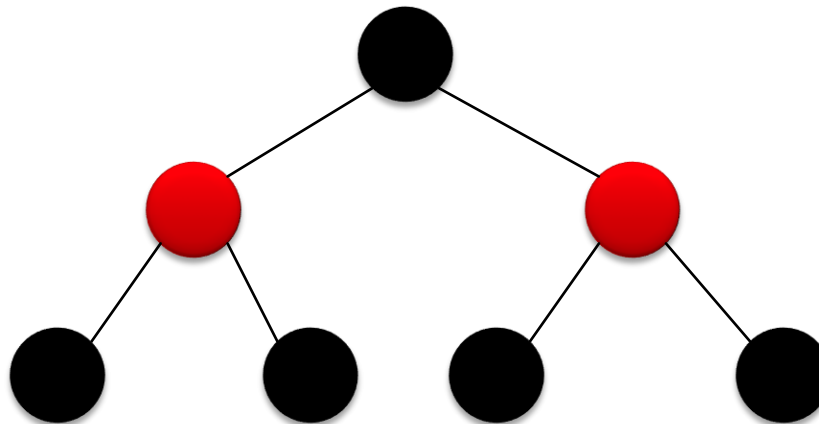
• Referências

1. Introdução

- ▶ Também conhecidas como árvores rubro-negras ou red-black trees.
 - 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 Sedgwick de 1978.

1. Introdução

- ▶ 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 e o registro.



1. Introdução

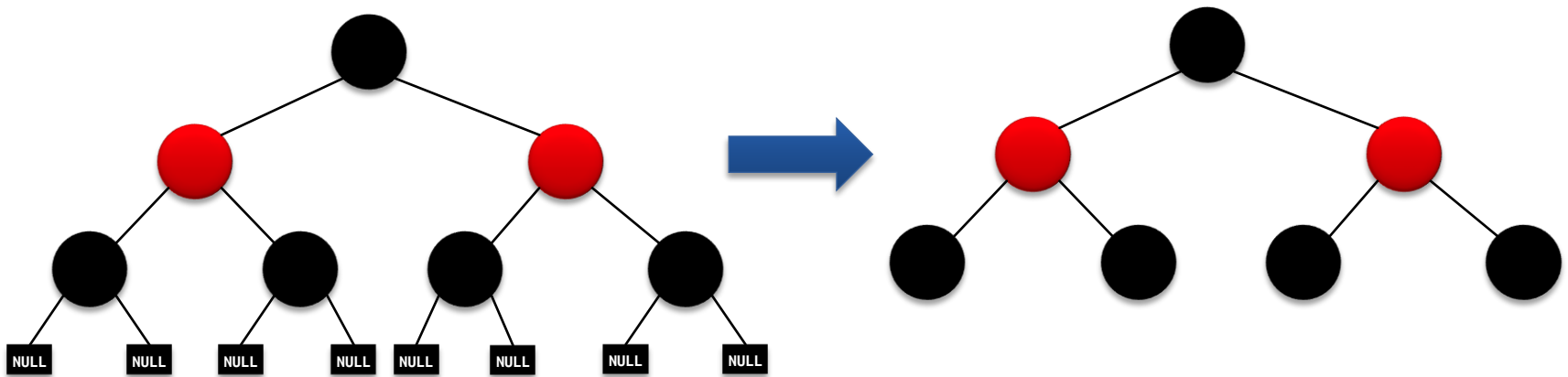
- ▶ Além da cor, a árvore deve satisfazer o seguinte conjunto de propriedades:
 - Todo nó da árvore é **vermelho** ou **preto**.
 - A raiz é sempre **preta**
 - Se um nó é **vermelho**, então os seus filhos são **pretos**.
 - Não existem nós **vermelhos** consecutivos.
 - Para cada nó, todos os caminhos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

1. Introdução

- ▶ Quando um nó não possui um filho (esquerdo ou direito) então supõe-se que ele aponta para um nó fictício (**NULL**), que será uma folha da árvore.
 - Assim, todos os nós internos contêm chaves e todas as folhas são nós fictícios.
 - Além disso, todo nó folha (**NULL**) é **preto**.

1. Introdução

- ▶ Como todo nó folha fictício tem valor **NULL**, eles podem ser ignorados na representação da árvore para fins didáticos.



1. Introdução

► 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 $O(\log n)$.

1.1. AVL vs Vermelho-Preto

- ▶ 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 Vermelho-Preto, o que acelera a operação de **busca**.

1.1. AVL vs Vermelho-Preto

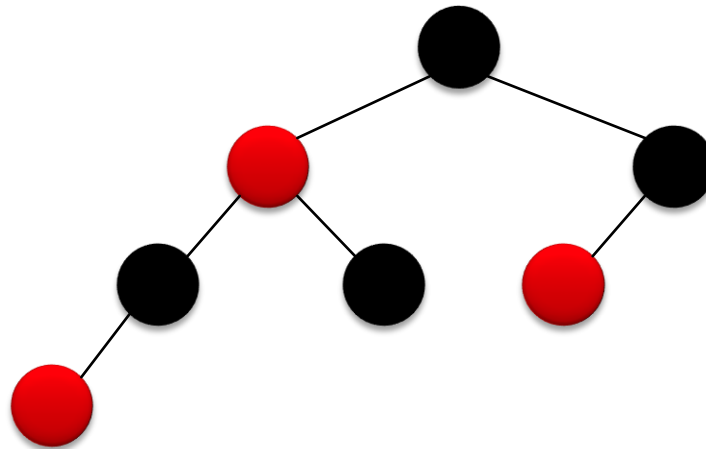
- ▶ AVL: balanceamento mais rígido.
 - Maior custo na operação de inserção e remoção.
 - No pior caso, uma operação de remoção pode exigir $O(\log n)$ rotações na árvore AVL, mas apenas 3 rotações na árvore Vermelho-Preto.
 - Qual usar?
 - Operação de busca é a mais usada?
 - Melhor usar uma árvore AVL.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

2. Árvore Rubro–Negra Esquerdista

- ▶ Desenvolvida por Robert Sedgewick em 2008.
 - Variante da árvore rubro–negra.
 - Garante a mesma complexidade de operações, mas possui uma implementação mais simples da inserção e remoção.

2. Árvore Rubro-Negra Esquerdista

- ▶ 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.



2.1. Implementação

```
enum Cor {VERMELHO , PRETO};
```

```
typedef struct {  
    long chave;  
    /* outros componentes */  
}TRegistro;
```

```
typedef struct TNo_Est {  
    TRegistro reg;  
    struct TNo_Est *pEsq, *pDir;  
    enum Cor cor;  
}TNo;
```

```
int ehVermelho (TNo* x) {  
    if (x == NULL)  
        return 0;  
    return x->cor == VERMELHO;  
}
```

```
int ehPreto (TNo* x) {  
    if (x == NULL)  
        return 1;  
    return x->cor == PRETO;  
}
```

2.2. Rotação

- ▶ **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.

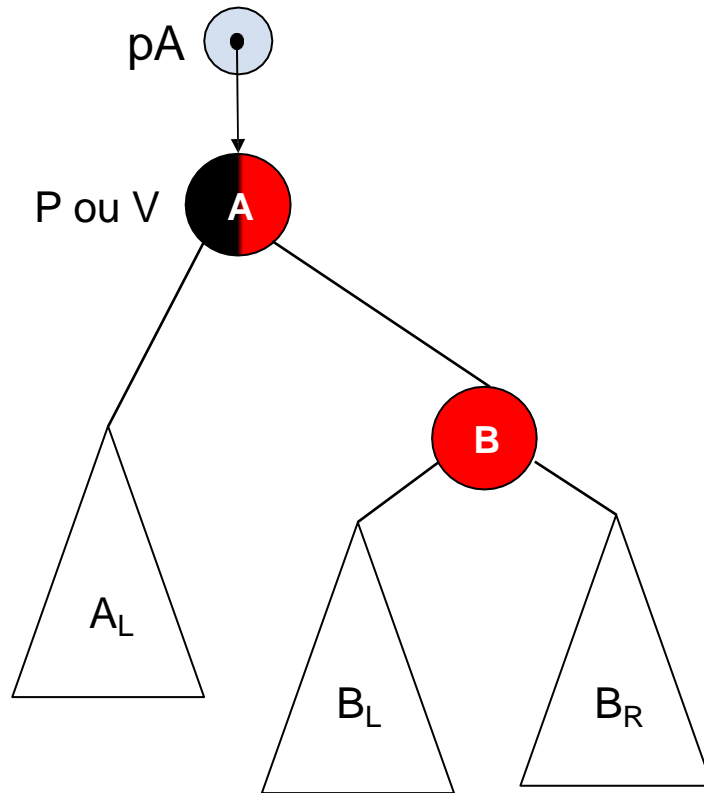
2.2. Rotação

► 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 $O(1)$.

2.2.1. Rotação à Esquerda

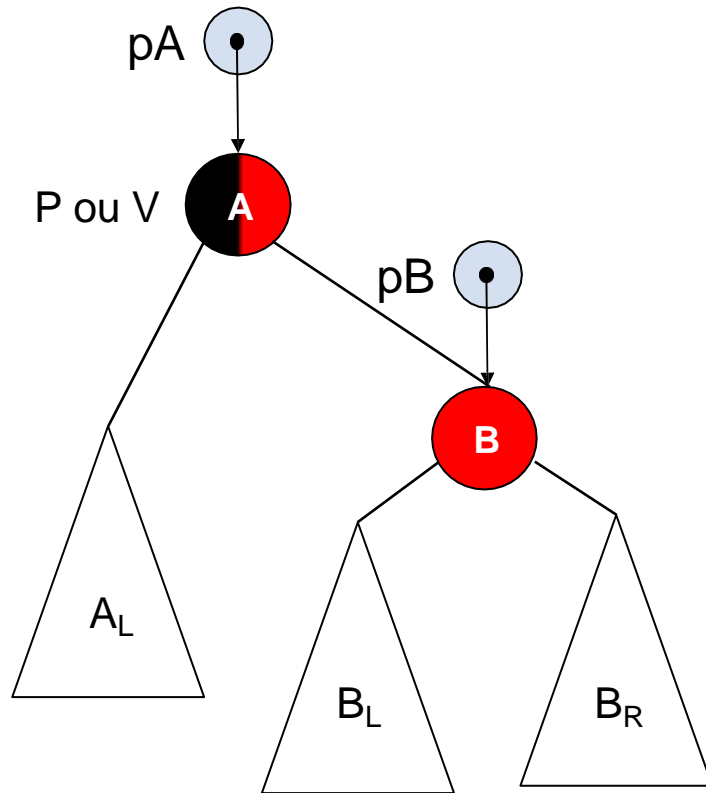
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```


2.2.1. Rotação à Esquerda

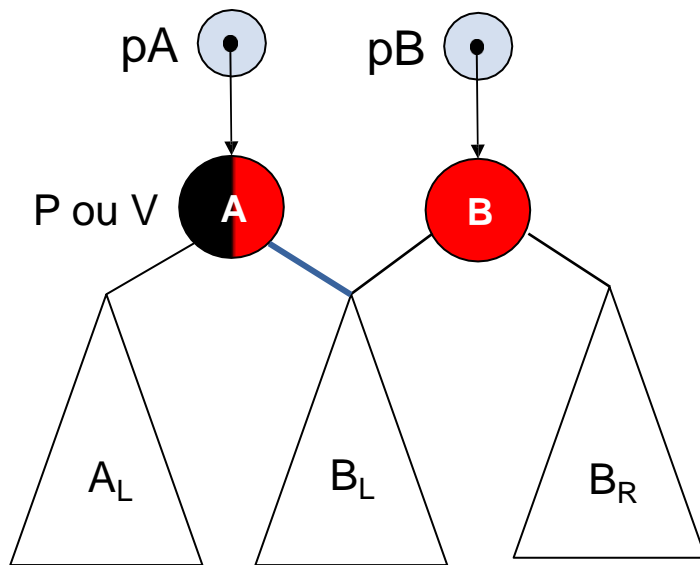
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.1. Rotação à Esquerda

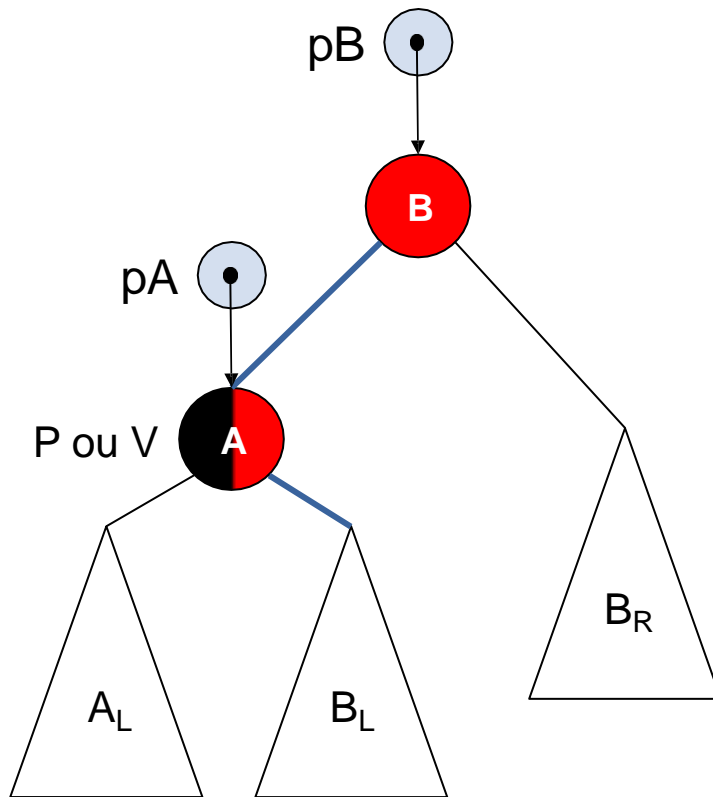
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.1. Rotação à Esquerda

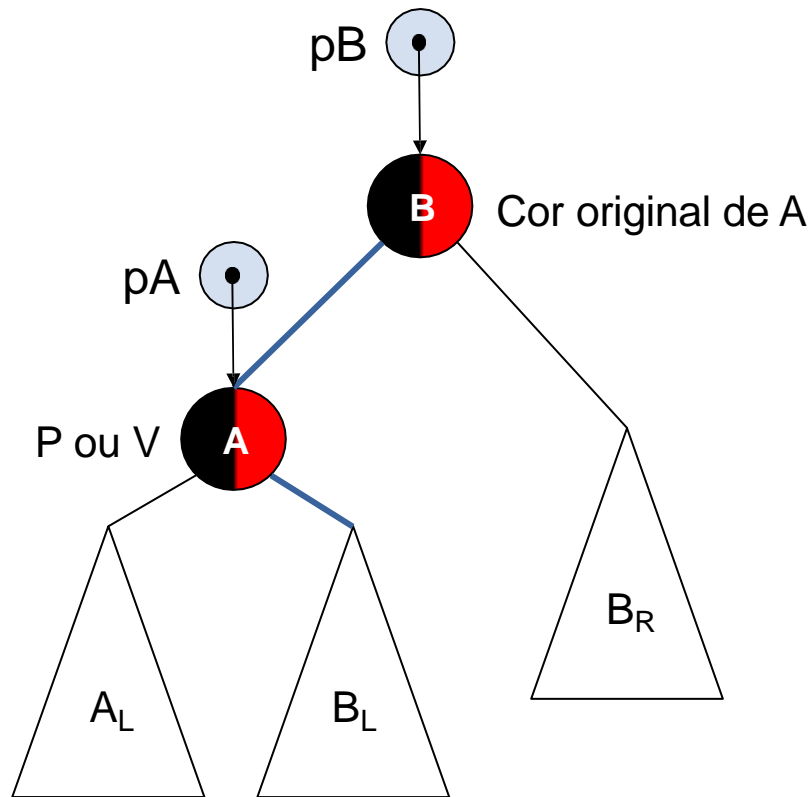
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.1. Rotação à Esquerda

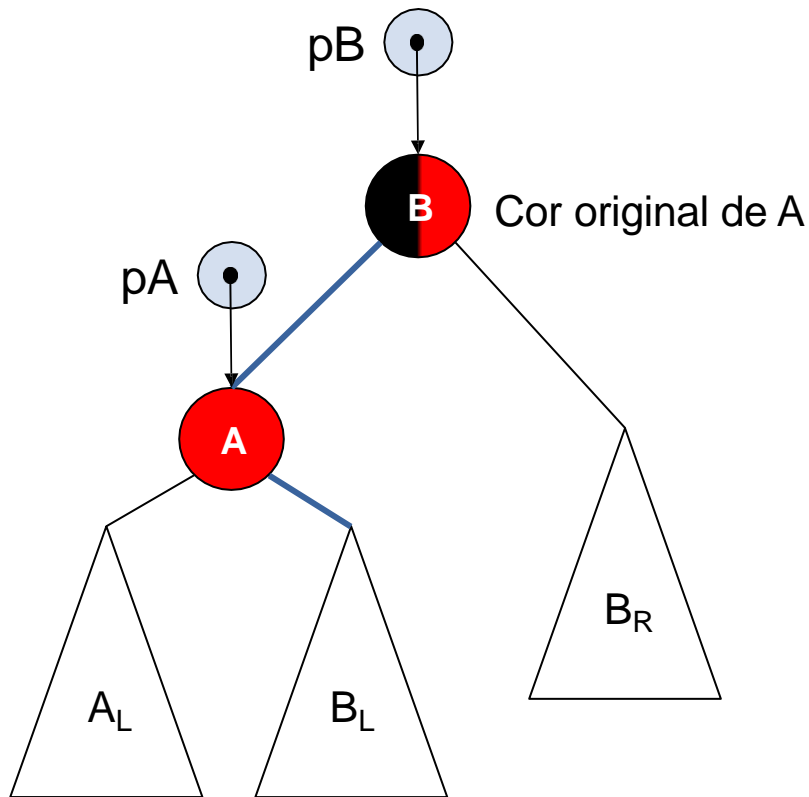
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.1. Rotação à Esquerda

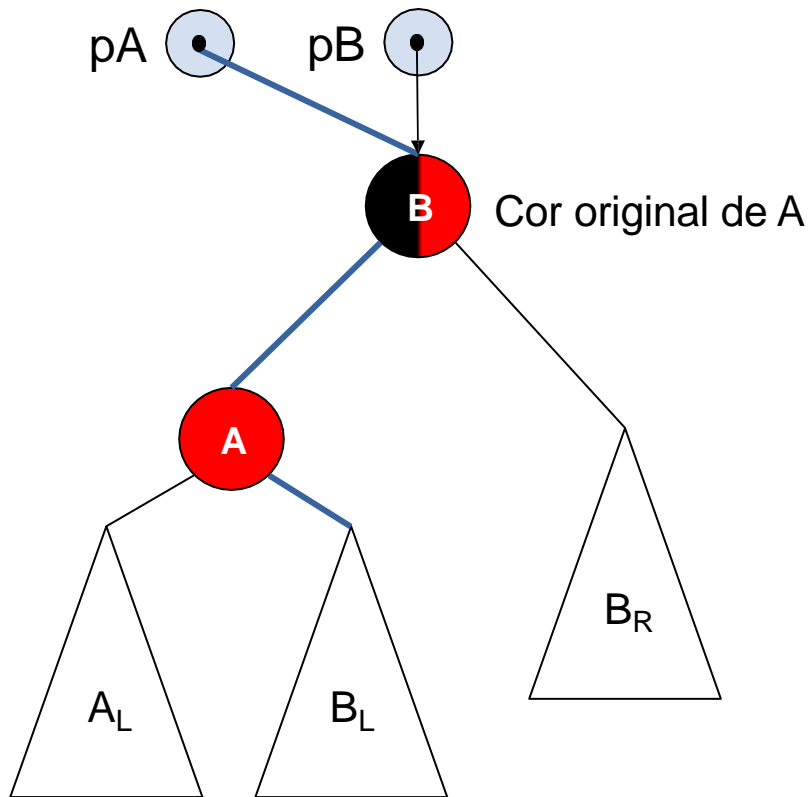
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.1. Rotação à Esquerda

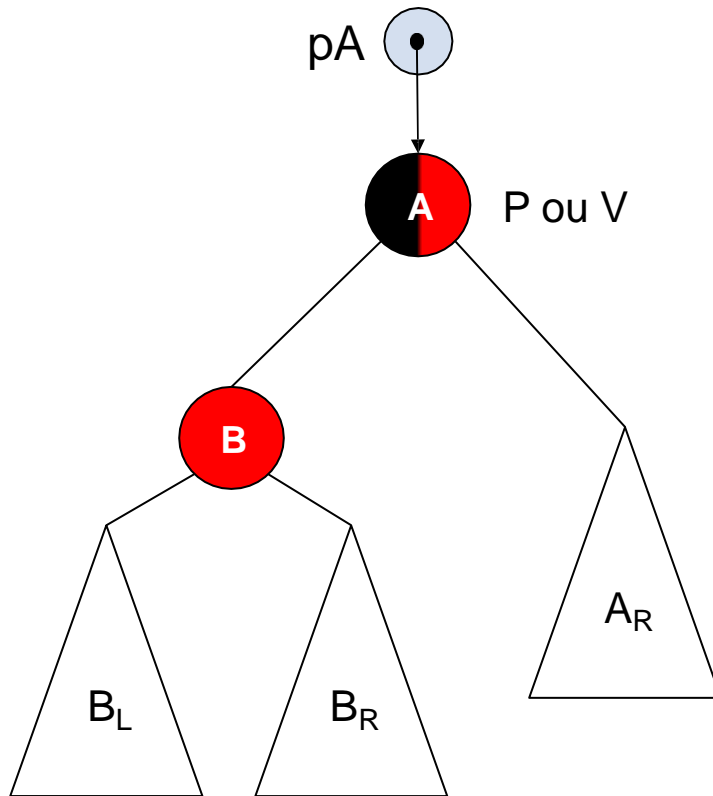
► Implementação: passo a passo



```
void RE (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pDir;  
    (*pA)->pDir = pB->pEsq;  
    pB->pEsq = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.2. Rotação à Direita

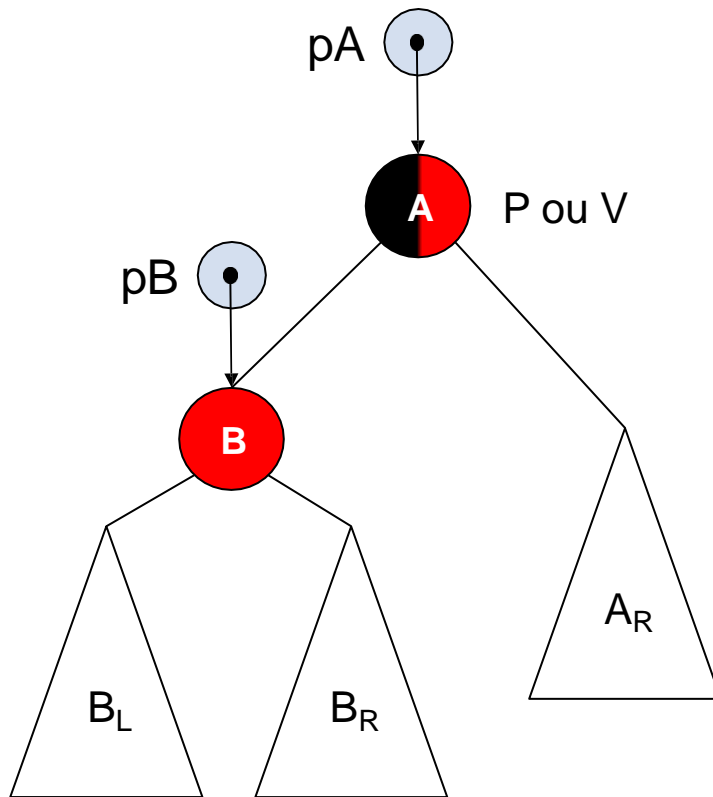
► Implementação: passo a passo



```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.2. Rotação à Direita

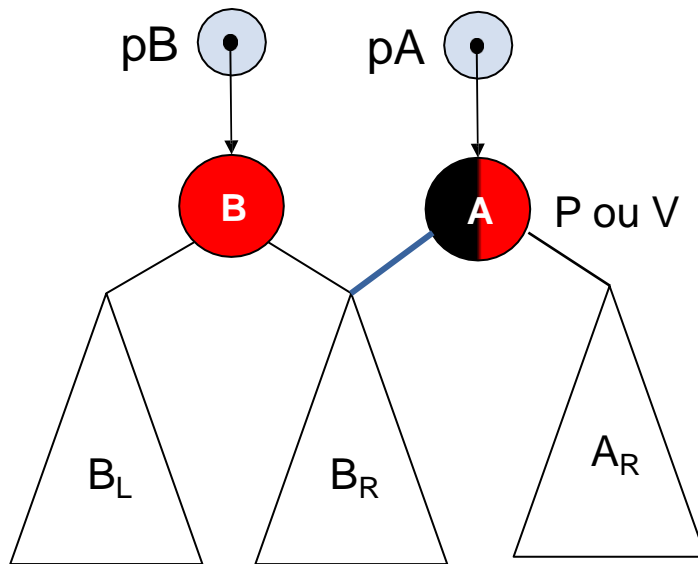
► Implementação: passo a passo



```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```


2.2.2. Rotação à Direita

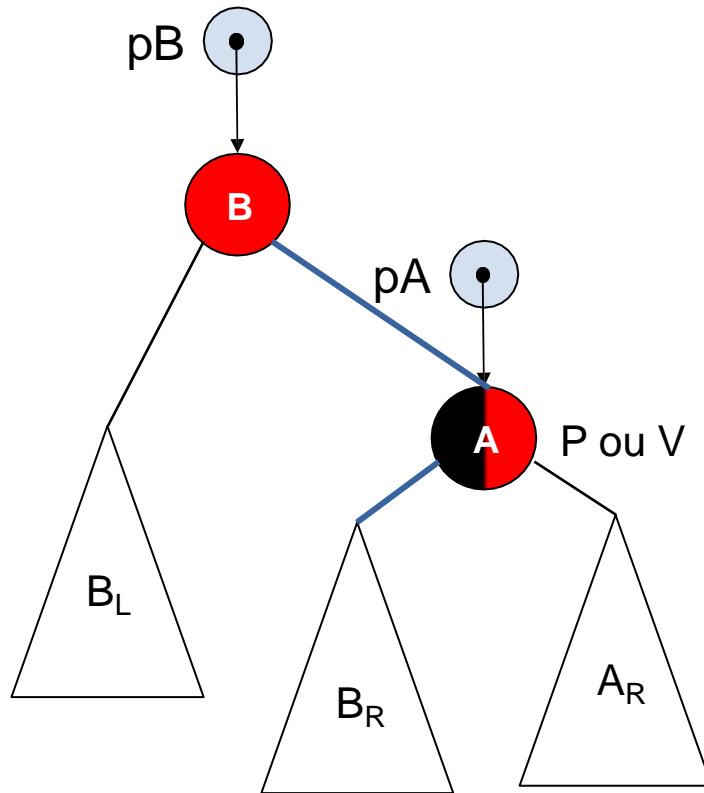
► Implementação: passo a passo



```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.2. Rotação à Direita

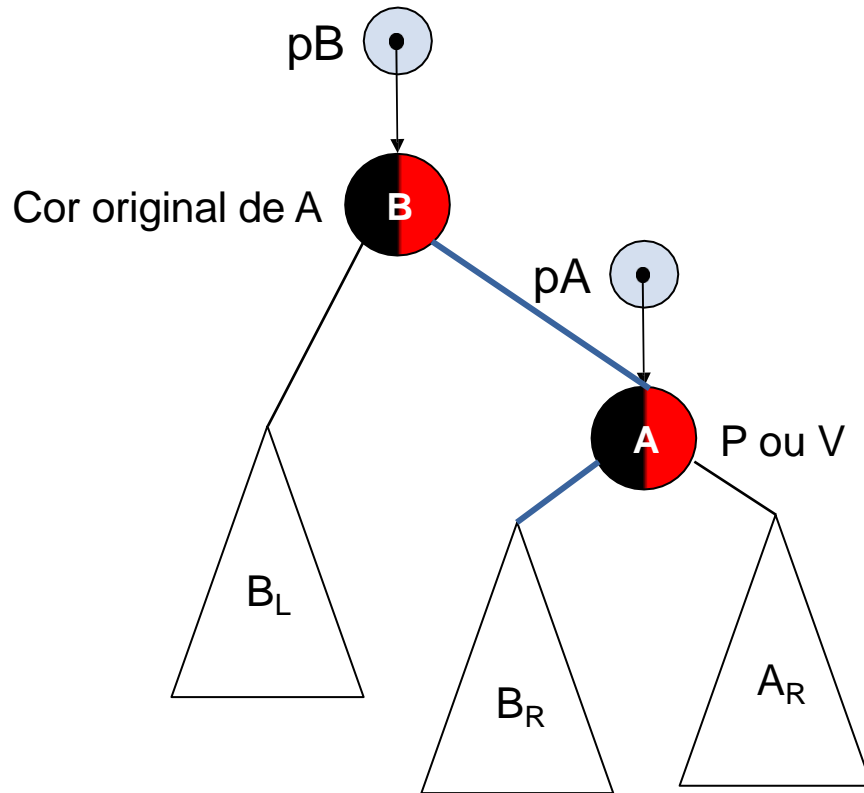
► Implementação: passo a passo



```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.2. Rotação à Direita

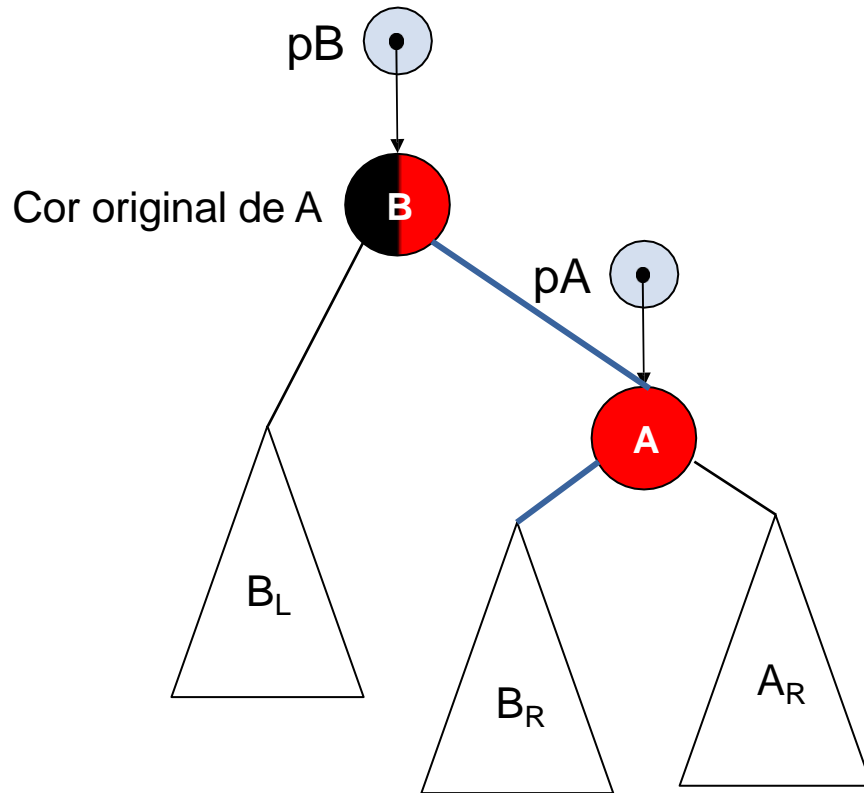
► Implementação: passo a passo



```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.2. Rotação à Direita

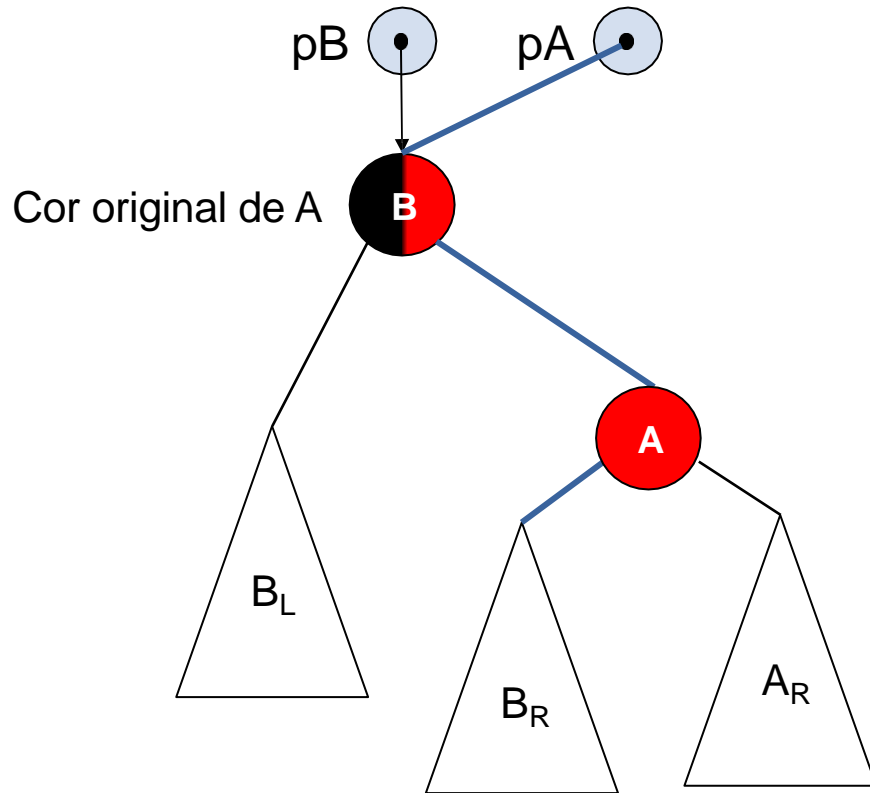
► Implementação: passo a passo



```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.2.2. Rotação à Direita

► Implementação: passo a passo



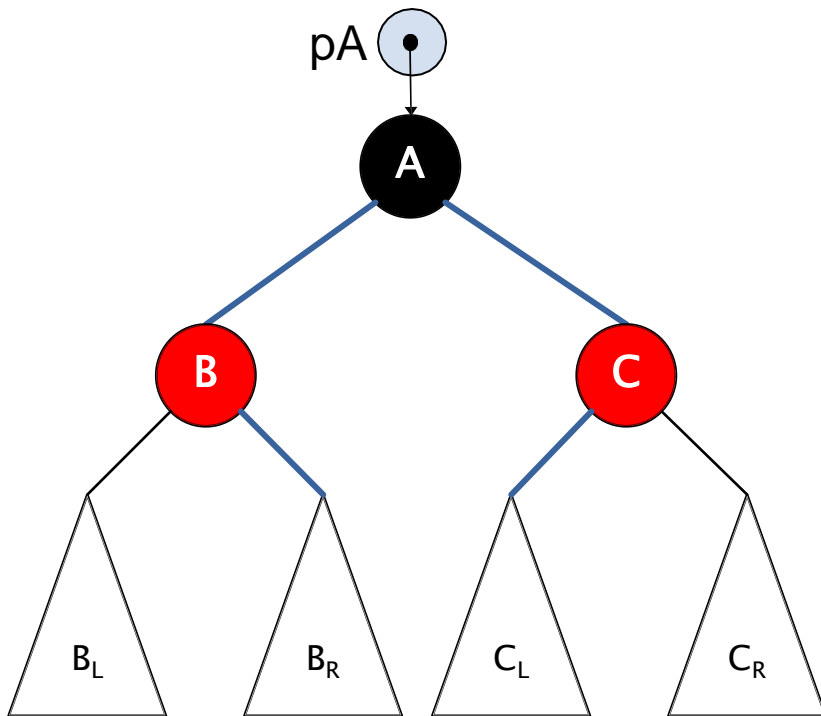
```
void RD (TNo** pA) {  
    TNo *pB;  
    pB = (*pA)->pEsq;  
    (*pA)->pEsq = pB->pDir;  
    pB->pDir = (*pA);  
    pB->cor = (*pA)->cor;  
    (*pA)->cor = VERMELHO;  
    (*pA) = pB;  
}
```

2.3. 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**.
 - Não altera o número de nós **pretos** da raiz até os nós folhas.
 - Mas pode pintar a raiz de **vermelho**.

2.3. Troca das cores dos nós

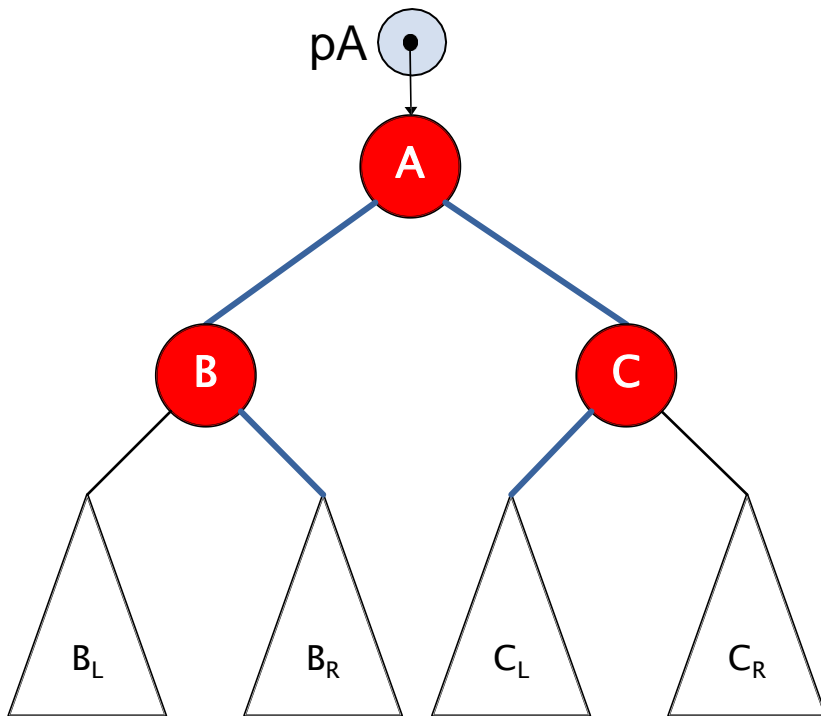
- Implementação: passo a passo



```
void sobe_vermelho (TNo* pA)
{
    pA->cor = VERMELHO;
    pA->pEsq->cor = PRETO;
    pA->pDir->cor = PRETO;
}
```

2.3. Troca das cores dos nós

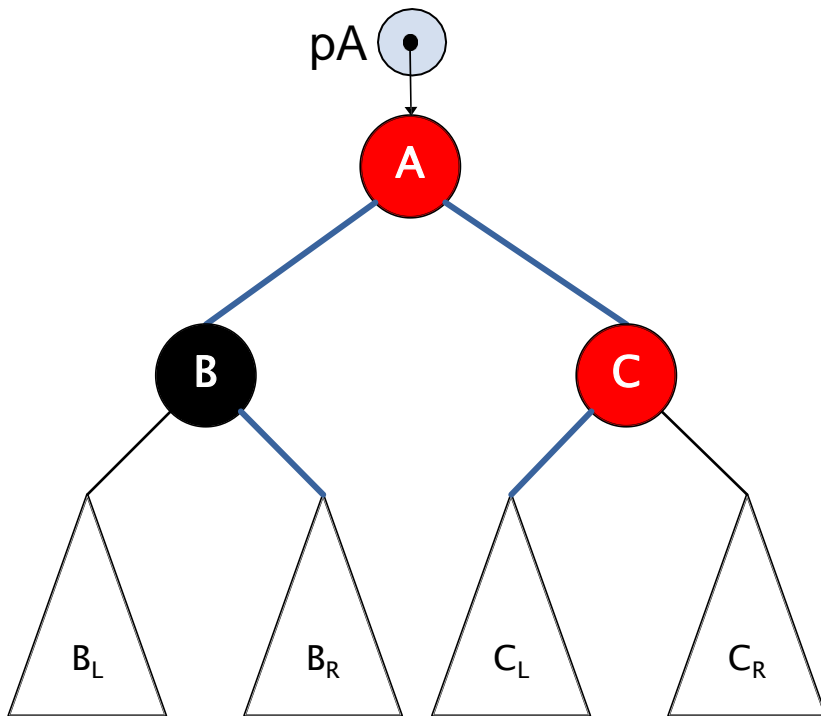
- Implementação: passo a passo



```
void sobe_vermelho (TNo* pA)
{
    pA->cor = VERMELHO;
    pA->pEsq->cor = PRETO;
    pA->pDir->cor = PRETO;
}
```


2.3. Troca das cores dos nós

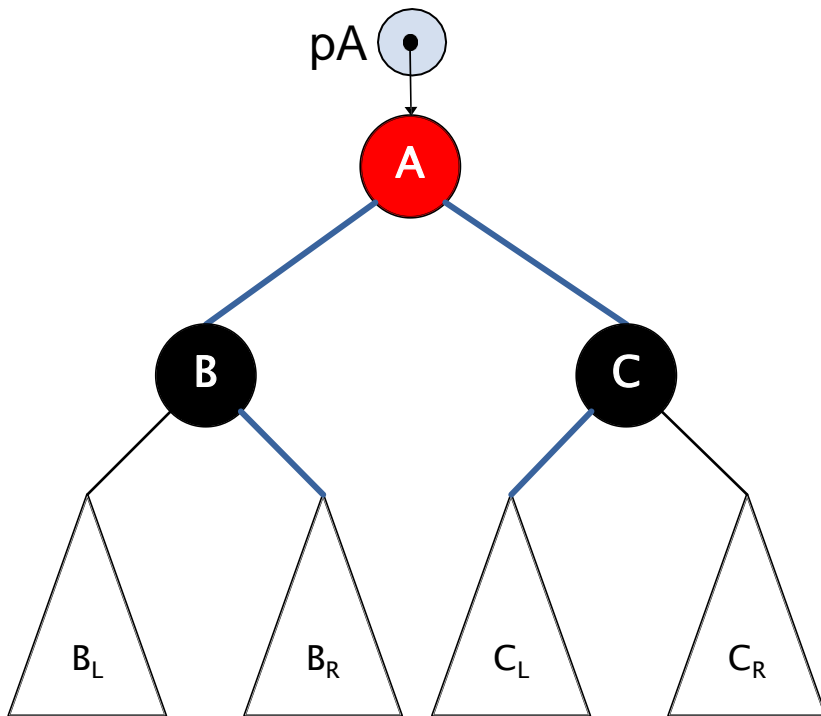
- Implementação: passo a passo



```
void sobe_vermelho (TNo* pA)
{
    pA->cor = VERMELHO;
    pA->pEsq->cor = PRETO;
    pA->pDir->cor = PRETO;
}
```

2.3. Troca das cores dos nós

- Implementação: passo a passo



```
void sobe_vermelho (TNo* pA)
{
    pA->cor = VERMELHO;
    pA->pEsq->cor = PRETO;
    pA->pDir->cor = PRETO;
}
```

3. 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.

3. Inserção

- ▶ Todo nó inserido é inicialmente **vermelho**.
- ▶ Uma vez inserido o novo nó:
 - É necessário 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.

3.1. Implementação

```
int insere_aux (TNo** ppRaiz, TRegistro* x) {  
  
    if (*ppRaiz == NULL) {  
        *ppRaiz = (TNo*) malloc (sizeof(TNo));  
        (*ppRaiz)->reg = *x;  
        (*ppRaiz)->pEsq = NULL;  
        (*ppRaiz)->pDir = NULL;  
        (*ppRaiz)->cor = VERMELHO;  
        return 1; }  
  
    if ((*ppRaiz)->reg.chave == x->chave)  
        return 0; /* valor já presente */  
}
```


3.1. Implementação

```
int resp;
```

```
if ((*ppRaiz)->reg.chave > x->chave)  
    resp = insere_aux (&(*ppRaiz)->pEsq, x);
```

```
else if ((*ppRaiz)->reg.chave < x->chave)  
    resp = insere_aux (&(*ppRaiz)->pDir, x);
```

```
/* Corrigir a árvore*/
```



Vamos inserir o código para corrigir a árvore aqui!

```
return resp;
```

```
}
```

3.1. Implementação

```
int insere (TNo** ppRaiz, TRegistro* x) {
```

```
    if (insere_aux (ppRaiz, x)) {  
        (*ppRaiz)->cor = PRETO;  
        return 1;  
    }
```

```
    else  
        return 0;  
}
```



Mantém a raiz preta

3.2. Corrigindo a Árvore

- ▶ É necessário corrigir cada propriedade,
 - Vamos supor que as sub-árvores esquerda e direita já satisfazem todas propriedades, com exceção da raiz **preta**.
 - Vamos corrigir as propriedades de raiz, uma por vez.

3.2. Corrigindo a Árvore

- ▶ Não queremos que um nó tenha apenas o filho direito **vermelho**:

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);
```

- ▶ Nem que um nó **vermelho** seja filho esquerdo de nó **vermelho**:

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);
```

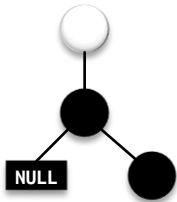
- ▶ Também não queremos que ambos filhos sejam **vermelhos**:

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 1

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 1

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 1

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq)) ← F
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 1

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```



3.2. Corrigindo a Árvore

► Inserção – Caso 1

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



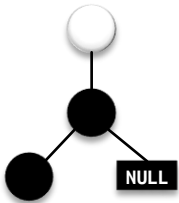
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir)) ← F
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 2

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **preto**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 2

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **preto**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```


3.2. Corrigindo a Árvore

► Inserção – Caso 2

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **preto**.
- Inserimos no filho direito.



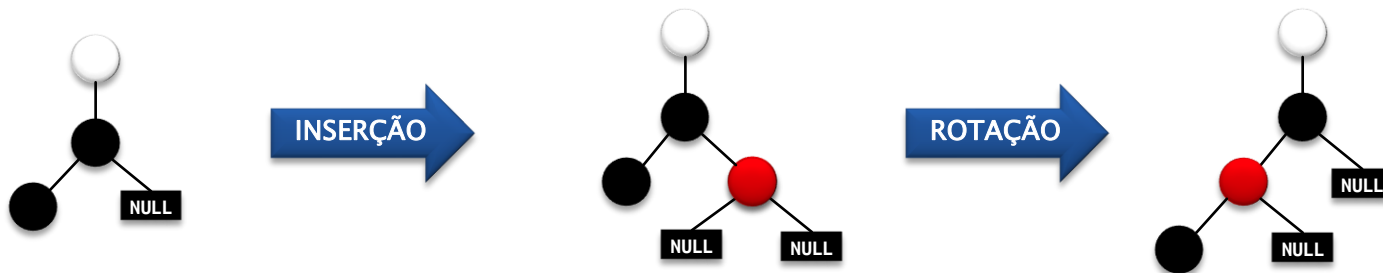
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq)) ← V
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 2

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **preto**.
- Inserimos no filho direito.



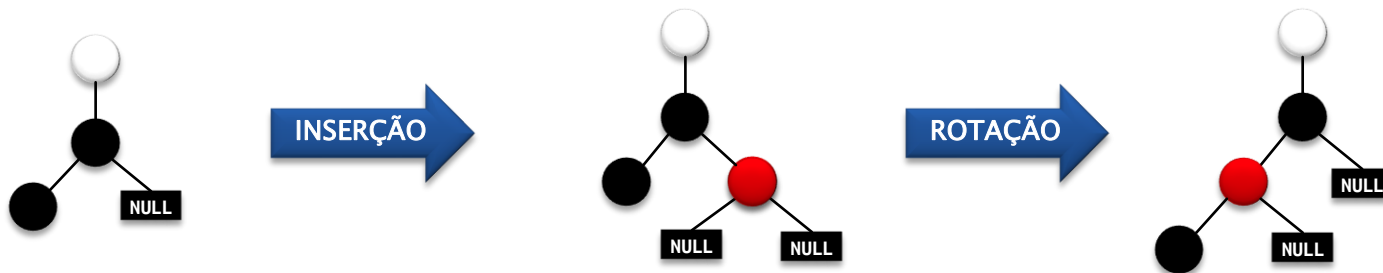
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 2

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **preto**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

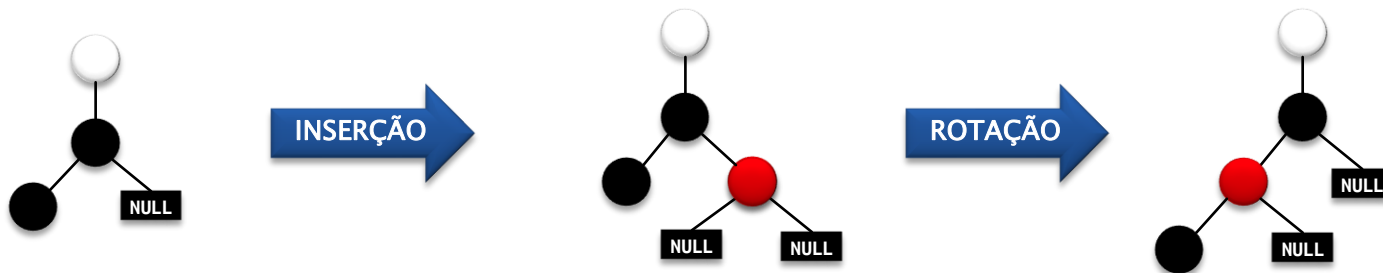
```
    sobe_vermelho((*ppRaiz));
```

← F

3.2. Corrigindo a Árvore

► Inserção – Caso 2

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **preto**.
- Inserimos no filho direito.



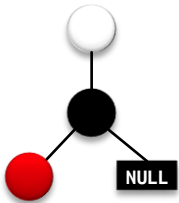
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir)) ← F
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 3

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **vermelho**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 3

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **vermelho**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 3

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **vermelho**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq)) ← F
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 3

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **vermelho**.
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

← F

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```


3.2. Corrigindo a Árvore

► Inserção – Caso 3

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **vermelho**.
- Inserimos no filho direito.



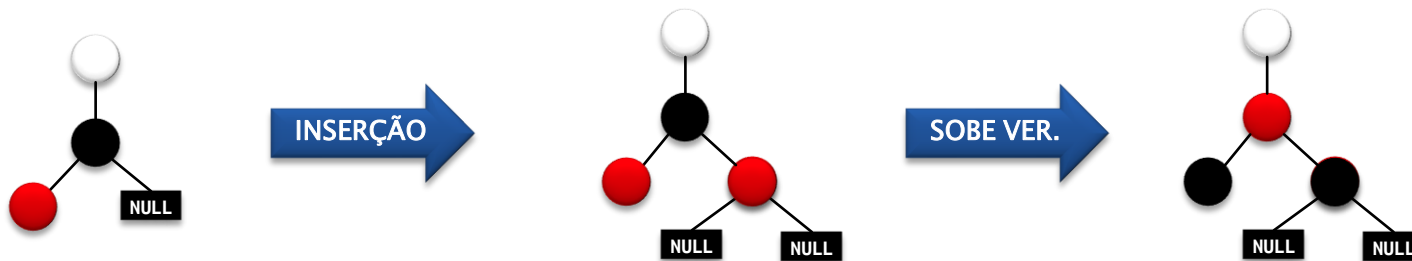
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir)) ← V
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 3

- Nó atual é **preto**.
 - Não sabemos a cor do seu pai.
 - Nem se ele é o filho esquerdo ou direito.
- Filho esquerdo é **vermelho**.
- Inserimos no filho direito.



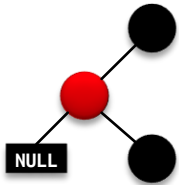
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz)); ←
```

3.2. Corrigindo a Árvore

► Inserção – Caso 4

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 4

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 4

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq)) ← F
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 4

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

← F

3.2. Corrigindo a Árvore

► Inserção – Caso 4

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho direito é **preto** (tem que ser).
- Inserimos no filho esquerdo.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

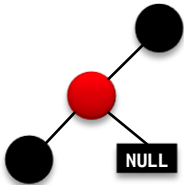
```
    sobe_vermelho((*ppRaiz));
```



3.2. Corrigindo a Árvore

► Inserção – Caso 5

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho esquerdo é **preto** (tem que ser).
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```


3.2. Corrigindo a Árvore

► Inserção – Caso 5

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho esquerdo é **preto** (tem que ser).
- Inserimos no filho direito.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 5

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho esquerdo é **preto** (tem que ser).
- Inserimos no filho direito.



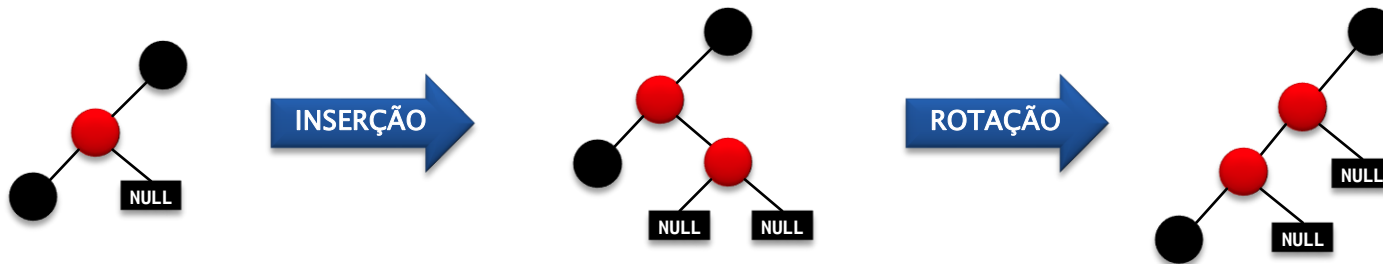
/* Corrigir a árvore*/

```
if (ehVermelho(*ppRaiz->pDir) && ehPreto(*ppRaiz->pEsq)) ← V
    RE(ppRaiz);
if (ehVermelho(*ppRaiz->pEsq) && ehVermelho(*ppRaiz->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho(*ppRaiz->pEsq) && ehVermelho(*ppRaiz->pDir))
    sobe_vermelho(*ppRaiz);
```

3.2. Corrigindo a Árvore

► Inserção – Caso 5

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho esquerdo é **preto** (tem que ser).
- Inserimos no filho direito.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz); ←
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

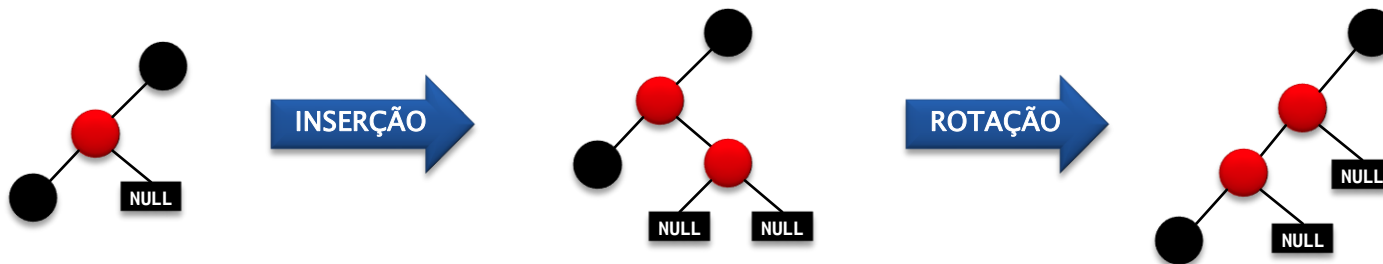
```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 5

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho esquerdo é **preto** (tem que ser).
- Inserimos no filho direito.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

← F

```
    RD(ppRaiz);
```

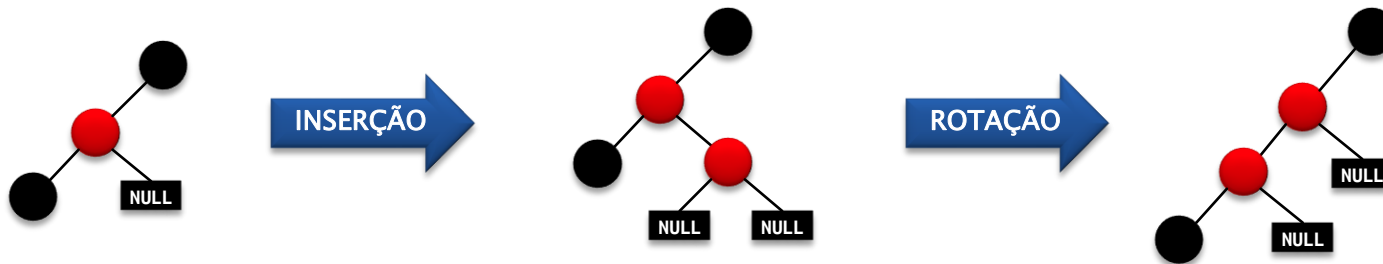
```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Inserção – Caso 5

- Nó atual é **vermelho**.
 - Seu pai é **preto** (ele não é a raiz).
 - É o filho esquerdo (pois é **vermelho**).
- Filho esquerdo é **preto** (tem que ser).
- Inserimos no filho direito.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```



3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

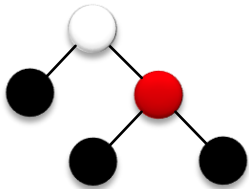
- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.

```
/* Corrigir a árvore */
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **preto**, basta rotacionar para a esquerda.

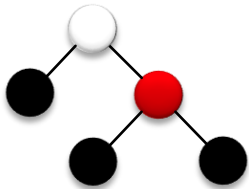


```
/* Corrigir a árvore */
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```


3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **preto**, basta rotacionar para a esquerda.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  V  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz));
```


3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **preto**, basta rotacionar para a esquerda.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz); ←
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **preto**, basta rotacionar para a esquerda.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

← F

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **preto**, basta rotacionar para a esquerda.



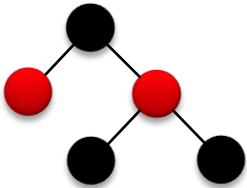
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir)) ← F
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **vermelho**, basta subir o **vermelho**.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```

```
    RD(ppRaiz);
```

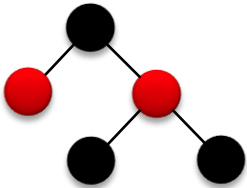
```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **vermelho**, basta subir o **vermelho**.



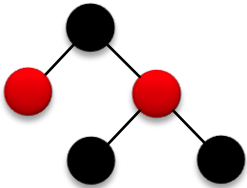
```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq)) ← F
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **vermelho**, basta subir o **vermelho**.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```



```
    RD(ppRaiz);
```

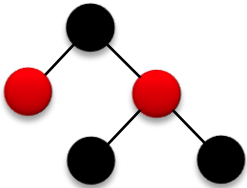
```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **vermelho**, basta subir o **vermelho**.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir)) ◀ V  
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho direito seja **vermelho** (não é esquerdista).
 - Só pode ter acontecido porque a cor **vermelha** subiu.
- Se o filho esquerdo for **vermelho**, basta subir o **vermelho**.



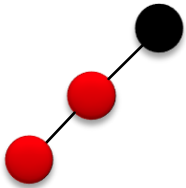
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz)); ←
```


3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho esquerdo seja **vermelho**.
 - E o neto mais a esquerda seja **vermelho**.

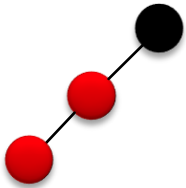


```
/* Corrigir a árvore */
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho esquerdo seja **vermelho**.
 - E o neto mais a esquerda seja **vermelho**.



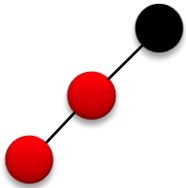
```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq)) ← F
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho esquerdo seja **vermelho**.
 - E o neto mais a esquerda seja **vermelho**.



```
/* Corrigir a árvore*/
```

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
```

```
    RE(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
```



```
    RD(ppRaiz);
```

```
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
```

```
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho esquerdo seja **vermelho**.
 - E o neto mais a esquerda seja **vermelho**.



/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz); ←
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho esquerdo seja **vermelho**.
 - E o neto mais a esquerda seja **vermelho**.



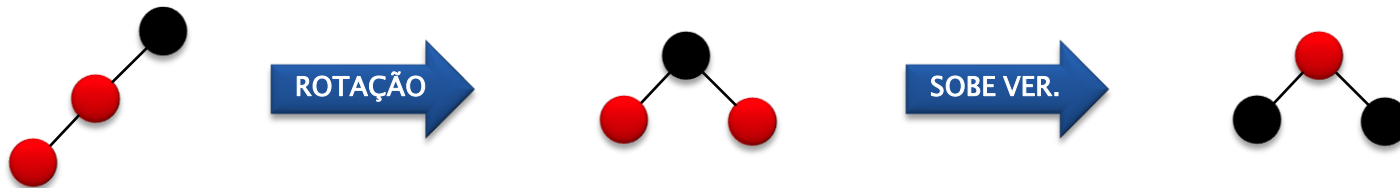
/* Corrigir a árvore*/

```
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir)) ← V
    sobe_vermelho((*ppRaiz));
```

3.2. Corrigindo a Árvore

► Resolvendo problemas no pai

- Quais problemas sobraram para o pai resolver?
 - Talvez o filho esquerdo seja **vermelho**.
 - E o neto mais a esquerda seja **vermelho**.



```
/* Corrigir a árvore */  
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))  
    RE(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))  
    RD(ppRaiz);  
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))  
    sobe_vermelho((*ppRaiz)); ←
```

3.3. Implementação (com correção)

```
int insere_aux (TNo** ppRaiz, TRegistro* x) {  
  
    if (*ppRaiz == NULL) {  
        *ppRaiz = (TNo*) malloc (sizeof(TNo));  
        (*ppRaiz)->reg = *x;  
        (*ppRaiz)->pEsq = NULL;  
        (*ppRaiz)->pDir = NULL;  
        (*ppRaiz)->cor = VERMELHO;  
        return 1; }  
  
    if ((*ppRaiz)->reg.chave == x->chave)  
        return 0; /* valor já presente */  
}
```

3.3. Implementação (com correção)

```
int resp;

if ((*ppRaiz)->reg.chave > x->chave)
    resp = insere_aux (&(*ppRaiz)->pEsq, x);

else if ((*ppRaiz)->reg.chave < x->chave)
    resp = insere_aux (&(*ppRaiz)->pDir, x);

/* Corrigir a árvore */
if (ehVermelho((*ppRaiz)->pDir) && ehPreto((*ppRaiz)->pEsq))
    RE(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pEsq->pEsq))
    RD(ppRaiz);
if (ehVermelho((*ppRaiz)->pEsq) && ehVermelho((*ppRaiz)->pDir))
    sobe_vermelho((*ppRaiz));

return resp; }
```


4. Remoção

- ▶ É possível fazer remoções em árvores vermelho-preto.
 - Mas não veremos aqui...
- ▶ A ideia é basicamente a mesma:
 - Encontrar operações que corrijam a árvore.
 - Operações locais que mantêm as propriedades globais.
 - O algoritmo é um pouco mais complexo que a inserção e há 4 casos a considerar.

4. Remoção

► Sugestão de leitura:

- Sedgewick e Wayne, Algorithms, 4th Edition, Addison–Wesley Professional, 2011.
- Cormen, Leiserson, Rivest e Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.

5. Análise

- ▶ As árvores rubro-negras esquerdistas suportam as seguintes operações:
 - Busca
 - Inserção
 - Remoção
- ▶ Todas em tempo $O(\log n)$.
- ▶ É uma variante da árvore rubro-negra com menos operações para corrigir a árvore na inserção e na remoção.
- ▶ Árvores rubro-negras são usadas como a árvore padrão no C++, no JAVA e no kernel do Linux.

6. Referências

- ▶ Material de aula dos Profs. Luiz Chaimowicz e Raquel O. Prates, da UFMG:
<https://homepages.dcc.ufmg.br/~glpappa/aeds2/AEDS2.1%20Conceitos%20Basicos%20TAD.pdf>
- ▶ Horowitz, E. & Sahni, S.; Fundamentos de Estruturas de Dados, Editora Campus, 1984.
- ▶ Wirth, N.; Algoritmos e Estruturas de Dados, Prentice/Hall do Brasil, 1989.
- ▶ Material de aula do Prof. José Augusto Baranauskas, da USP:
<https://dcm.ffclrp.usp.br/~augusto/teaching.htm>
- ▶ Material de aula do Prof. Rafael C. S. Schouery, da Unicamp:
<https://www.ic.unicamp.br/~rafael/cursos/2s2019/mc202/index.html>