ÁRVORE RUBRO-NEGRA

Prof. André Backes

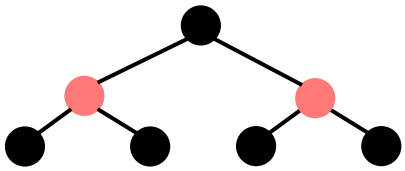
Árvore rubro-negra

- 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

3

- 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

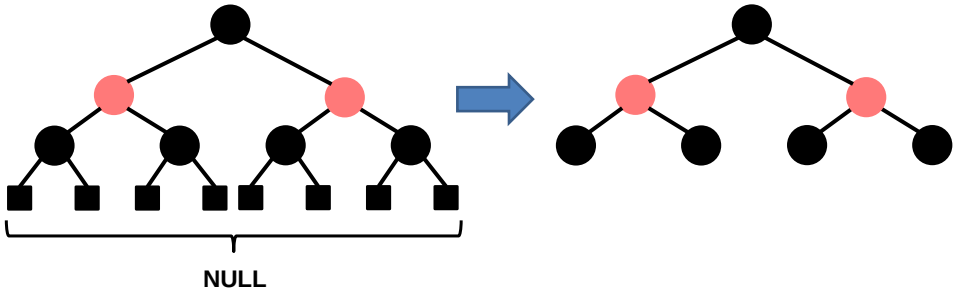
4

- 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

5

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



Balanceamento

6

- 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

7

- 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

8

- 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

9

- 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

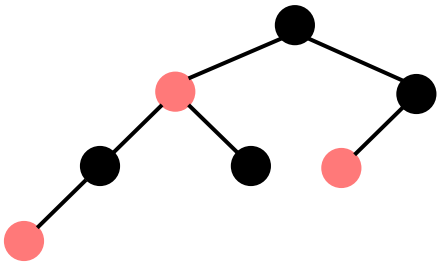
10

- 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

11

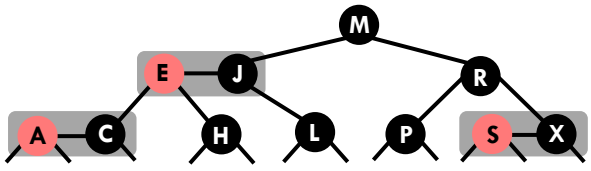
- Possui uma propriedade extra além das propriedades da árvore convencional,
  - Se um nó é **vermelho**, então ele é o filho esquerdo do seu pai
  - Aspecto de caída para a esquerda



Árvore Rubro-Negra caída para a Esquerda

12

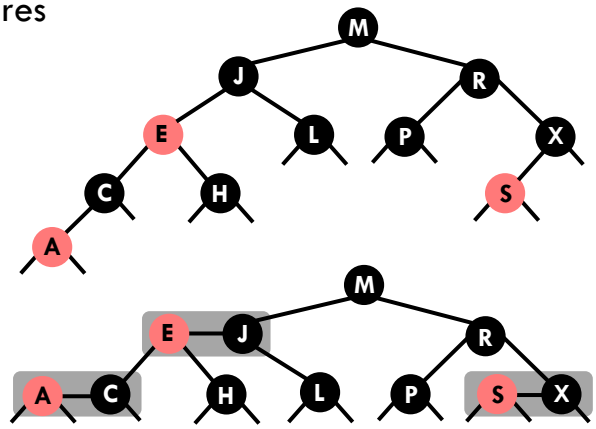
- Sua implementação corresponde a de uma **árvore 2-3**
  - A árvore 2-3 não é uma árvore binária
    - Cada nó interno pode armazenar um ou dois valores
    - Pode ter dois (um valor) ou três (dois valores) filhos
    - Seu funcionamento é o mesmo da árvore binária de busca



Árvore Rubro-Negra caída para a Esquerda

13

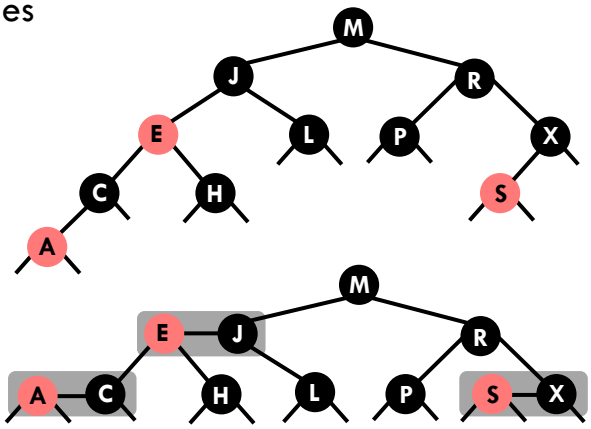
- Neste caso, o nó vermelho será sempre o valor menor de um nó contendo dois valores e três sub-árvores



Árvore Rubro-Negra caída para a Esquerda

14

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



TAD Árvore Rubro Negra

15

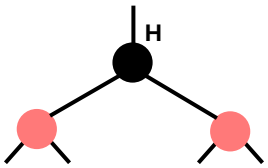
- Definindo a árvore
  - Criação e destruição: igual a da árvore binária

```
1  //Arquivo ArvoreLLRB.h
2  typedef struct NO* ArvLLRB;
3
4  //Arquivo ArvoreLLRB.c
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "ArvoreLLRB.h" //inclui os Protótipos
8
9  #define RED 1 //define as cores
10 #define BLACK 0
11
12 struct NO{
13     int info;
14     struct NO *esq;
15     struct NO *dir;
16     int cor;
17 };
18 //programa principal
19 ArvLLRB* raiz; //ponteiro para ponteiro
```

Troca das cores dos nós

16

- 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

17

- 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

TAD Árvore Rubro Negra

18

- Acessando a cor e trocando as cores

```
1  //Funções auxiliares
2  //Acessando a cor de um nó
3  int cor(struct NO* H){
4      if(H == NULL)
5          return BLACK;
6      else
7          return H->cor;
8  }
9  //Inverte a cor do pai e de seus filhos
10 //É uma operação "administrativa": não altera
11 //a estrutura ou conteúdo da árvore
12 void trocaCor(struct NO* H){
13     H->cor = !H->cor;
14     if(H->esq != NULL)
15         H->esq->cor = !H->esq->cor;
16     if(H->dir != NULL)
17         H->dir->cor = !H->dir->cor;
18 }
```

Rotações

19

- 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

20

- 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

21

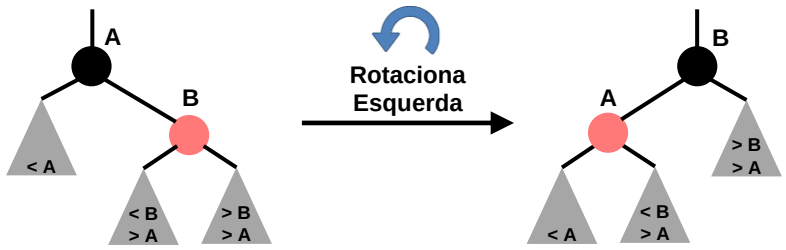
- Rotação à Esquerda
  - Recebe um nó **A** com **B** como filho **direito**
  - Move **B** para o lugar de **A, A** se torna o filho **esquerdo** de **B**
  - **B** recebe a cor de **A, A** fica **vermelho**

```
8  struct NO* rotacionaEsquerda(struct NO* A){
9      struct NO* B = A->dir;
10     A->dir = B->esq;
11     B->esq = A;
12     B->cor = A->cor;
13     A->cor = RED;
14     return B;
15 }
```

Rotações

22

- 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

23

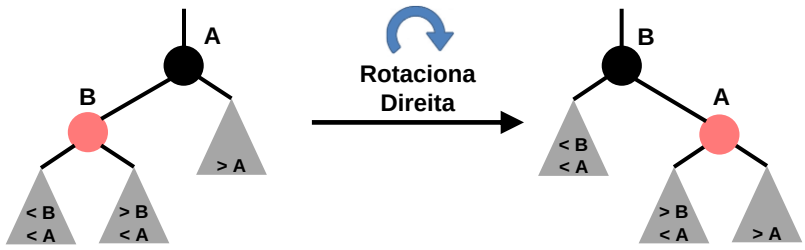
- Rotação à Direita
  - Recebe um nó **A** com **B** como filho **esquerdo**
  - Move **B** para o lugar de **A, A** se torna o filho **direito** de **B**
  - **B** recebe a cor de **A, A** fica **vermelho**

```
8  struct NO* rotacionaDireita(struct NO* A){
9      struct NO* B = A->esq;
10     A->esq = B->dir;
11     B->dir = A;
12     B->cor = A->cor;
13     A->cor = RED;
14     return B;
15 }
16
```

Rotações

24

- 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

25

- 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

26

- **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

TAD Árvore Rubro Negra

27

- Inserção
  - Função que gerencia o nó raiz após a inserção

```
7  //arquivo ArvoreLLRB.c
8  int insere_ArvLLRB(ArvLLRB* raiz, int valor){
9      int resp;
10     //FUNÇÃO RESPONSÁVEL PELA BUSCA DO LOCAL
11     //DE INSERÇÃO DO NÓ
12     *raiz = insereNO(*raiz,valor,&resp);
13     if((*raiz) != NULL)
14         (*raiz)->cor = BLACK;
15
16     return resp;
17 }
```

TAD Árvore Rubro Negra

28

- Inserção

```
1  struct NO* insereNO(struct NO* H,int valor,int *resp){
2     if(H == NULL){
3       struct NO *novo
4       novo = (struct NO*)malloc(sizeof(struct NO));
5       if(novo == NULL){
6         *resp = 0;
7         return NULL;
8       }
9       novo->info = valor;
10      novo->cor = RED;
11      novo->dir = NULL;
12      novo->esq = NULL;
13      *resp = 1;
14      return novo;
15    }
16    //continua...
```

TAD Árvore Rubro Negra

29

- Inserção

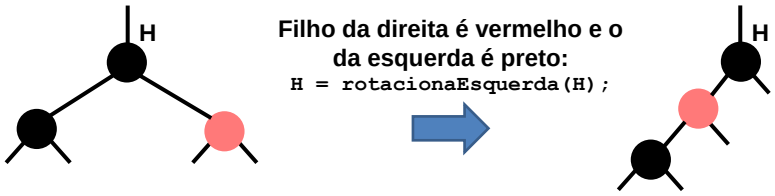
Corrige violações de propriedades

```
1   //continuação
2   if(valor == H->info)
3     *resp = 0;// Valor duplicado
4   else{
5     if(valor < H->info)
6         H->esq = insereNO(H->esq,valor,resp);
7     else
8         H->dir = insereNO(H->dir,valor,resp);
9   }
10
11  if(cor(H->dir) == RED && cor(H->esq) == BLACK)
12    H = rotacionaEsquerda(H);
13
14  if(cor(H->esq) == RED && cor(H->esq->esq) == RED)
15    H = rotacionaDireita(H);
16
17  if(cor(H->esq) == RED && cor(H->dir) == RED)
18    trocaCor(H);
19
20  return H;
21  }
```

Árvore rubro-negra: Inserção

30

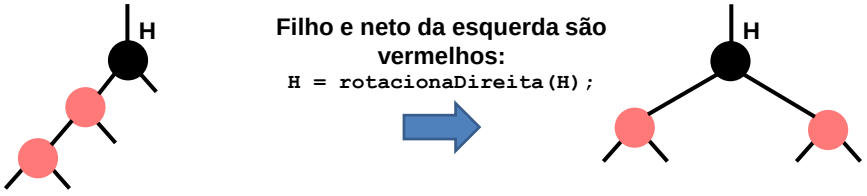
- Violações das propriedades na inserção
  - Filho da direita é **vermelho** e o filho da esquerda é **preto**
    - Solução: Rotação à esquerda



Árvore rubro-negra: Inserção

31

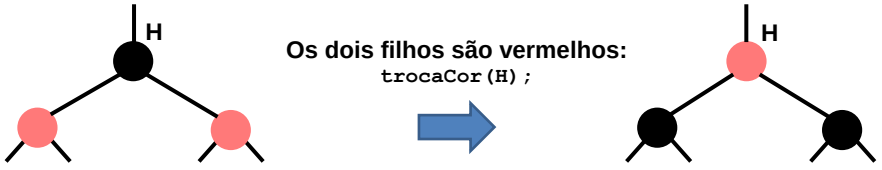
- Violações das propriedades na inserção
  - Filho da esquerda é **vermelho** e o filho à esquerda do filho da esquerda também é **vermelho**
    - Solução: Rotação à direita



Árvore rubro-negra: Inserção

32

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



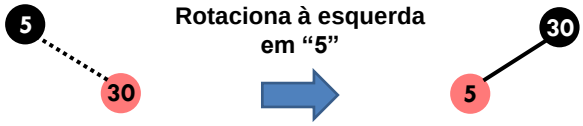
Árvore rubro-negra: Inserção

33

- Passo a passo:

Insere valor: 1

Insere valor: 30

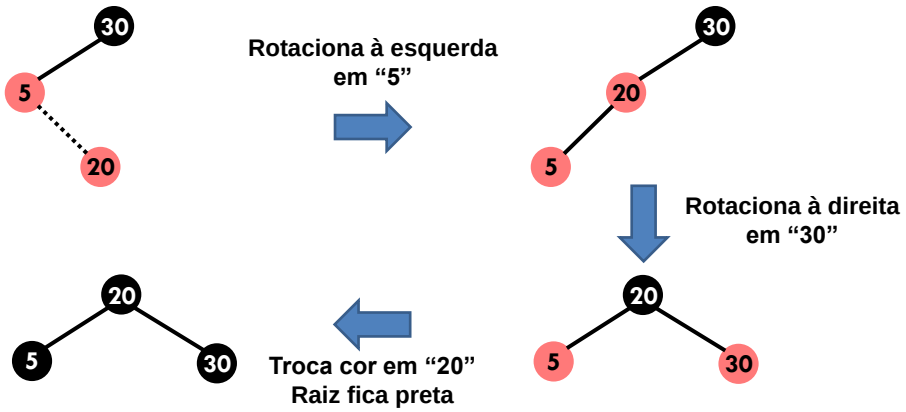


Árvore rubro-negra: Inserção

34

- Passo a passo:

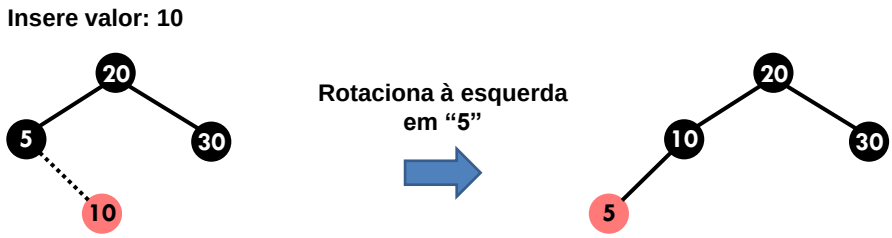
Insere valor: 20



Árvore rubro-negra: Inserção

35

- Passo a passo:



Árvore rubro-negra: Remoção

36

- 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

37

- 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

38

- 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

TAD Árvore Rubro Negra

39

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

```
7   //arquivo ArvoreLLRB.c
8   int remove_ArvLLRB(ArvLLRB *raiz, int valor){
9       if(consulta_ArvLLRB(raiz,valor)){
10          struct NO* h = *raiz;
11          //FUNÇÃO RESPONSÁVEL PELA BUSCA
12          //DO NÓ A SER REMOVIDO
13          *raiz = remove_NO(h,valor);
14          if(*raiz != NULL)
15              (*raiz)->cor = BLACK;
16          return 1;
17      }else
18          return 0;
19  }
```

TAD Árvore Rubro Negra

40

- Remoção
  - 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

41

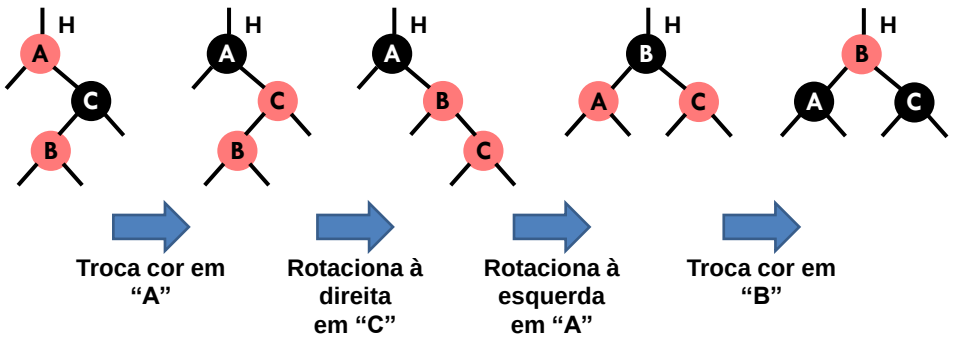
- Move um nó **vermelho** para a esquerda
  - Troca as cores do nó **H** e seus filhos
  - Filho a **esquerda** do filho **direito** é **vermelho**
    - Rotação à **direita** no **filho direito** e à **esquerda** no **pai**
  - Troca as cores do nó pai e de seus filhos

```
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

42

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



Função move2DirRED

43

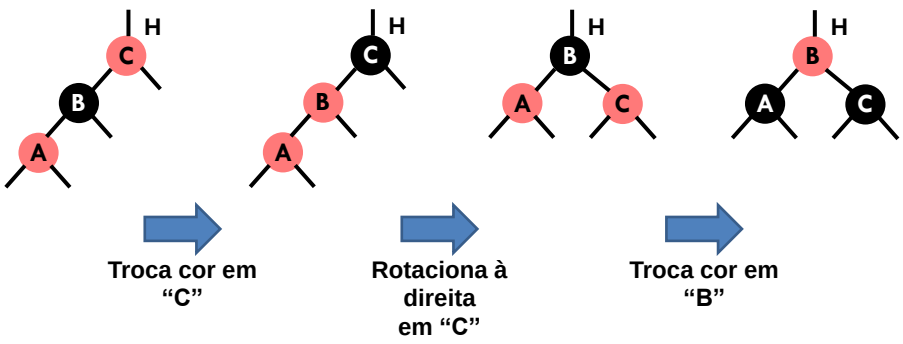
- Move um nó **vermelho** para a direita
  - Troca as cores do nó **H** e seus filhos
  - Filho a **esquerda** do filho **esquerdo** é **vermelho**
    - Rotação à **direita** no **pai**
  - Troca as cores do nó pai e de seus filhos

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

Função move2DirRED

44

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



Função balancear

45

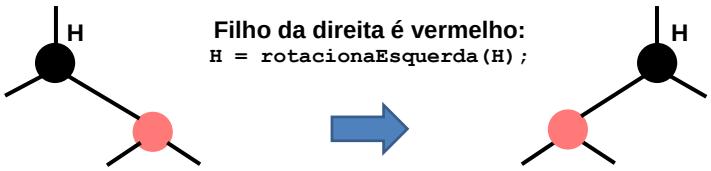
- Trata várias violações

```
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

46

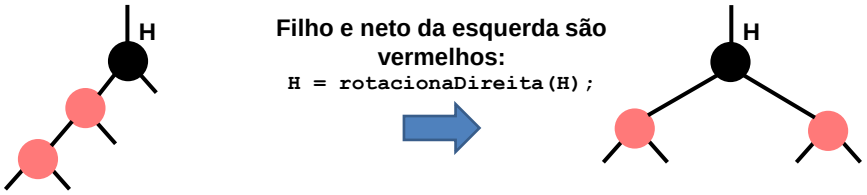
- Violações das propriedades na remoção
  - Nó da direita é **vermelho**
    - Solução: rotação à esquerda



Função balancear

47

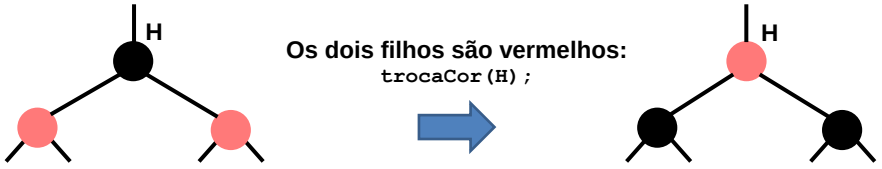
- Violações das propriedades na remoção
  - Filho da esquerda e neto da esquerda são **vermelhos**
    - Solução: rotação à direita



Função balancear

48

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



Funções procuraMenor e removerMenor

49

- Nó removido possui filhos

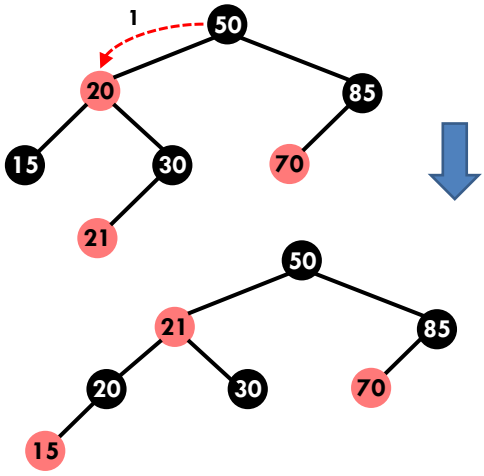
```
1  struct NO* removerMenor(struct NO* H){
2      if(H->esq == NULL){
3          free(H);
4          return NULL;
5      }
6      if(cor(H->esq) == BLACK && cor(H->esq->esq)==BLACK)
7          H = move2EsqRED(H);
8
9      H->esq = removerMenor(H->esq);
10     return balancear(H);
11 }
12 struct NO* procuraMenor(struct NO* atual){
13     struct NO *no1 = atual;
14     struct NO *no2 = atual->esq;
15     while(no2 != NULL){
16         no1 = no2;
17         no2 = no2->esq;
18     }
19     return no1;
20 }
```

Procura pelo nó mais a esquerda

Árvore rubro-negra: Remoção

50

- Passo a passo:



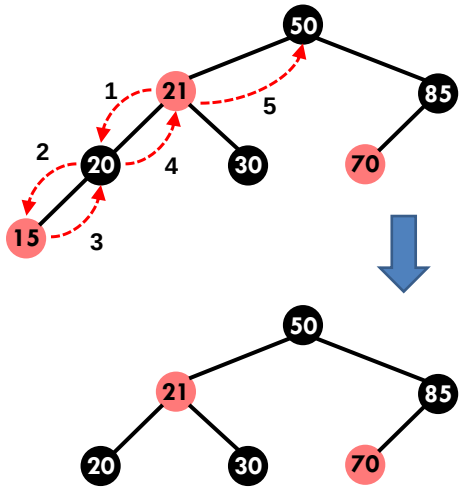
Remove valor: 15

	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

51

- 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

52

- 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/_ITz-ePzWjk
  - Aula 108: Rotação da Árvore Rubro Negra LLRB:
  - youtu.be/Pa8PI6o09Ic
  - Aula 109: Movendo os nós vermelhos:
  - youtu.be/Io6Zk7zXOww
  - Aula 110: Inserção na Árvore Rubro-Negra – LLRB:
  - youtu.be/L4gWuqpvk4E
  - Aula 111: Remoção na Árvore Rubro-Negra – LLRB:
  - youtu.be/p5aukRcjdqc