Estruturas de Dados 2

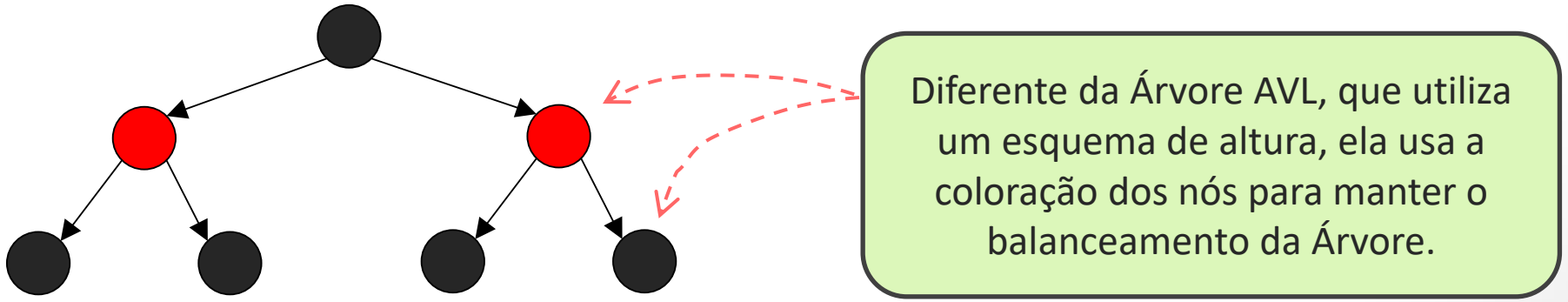
aula 05 – Árvore Rubro-Negra – parte 1

Antonio Angelo de Souza Tartaglia

angelot@ifsp.edu.br

Árvore Rubro-Negra

- É um tipo de Árvore Binária Balanceada;
- Originalmente criada por **Rudolf Bayer** em 1972, e recebeu o nome de **Árvore Binária Simétrica**;
- Posteriormente, em um trabalho de **Leonidas J. Guibas** e **Robert Sedgewick** de 1978, recebeu seu nome atual;
- Utiliza um esquema de coloração dos nós para manter o balanceamento da Árvore:

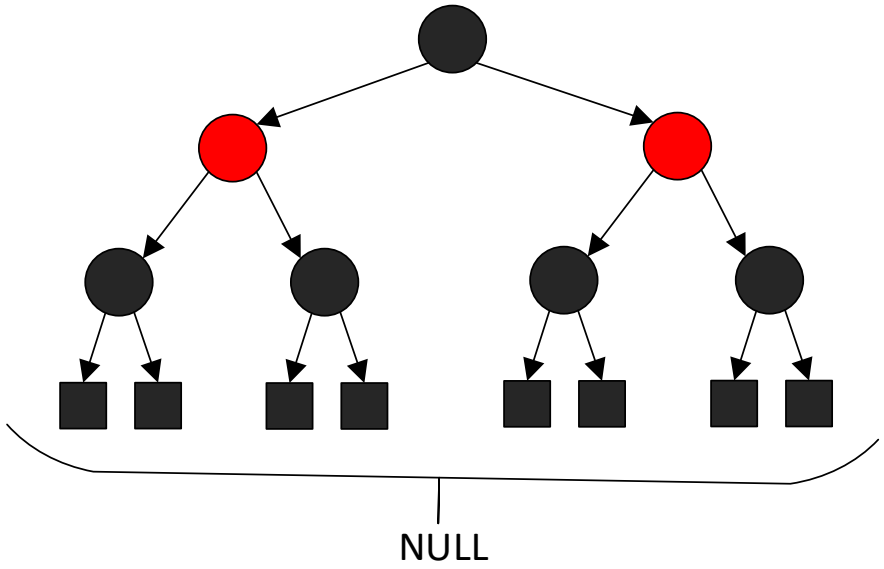


Árvore Rubro-Negra

- Assim como a Árvore AVL a Rubro-Negra é complexa, mas tem um bom **pior caso** de tempo de execução para suas operações, é eficiente e prática;
- Pode-se efetuar as operações de busca, inserção e remoção com o tempo de execução de $\boldsymbol{O(log\ n)}$;
- De uma maneira simplificada, uma Árvore Rubro-Negra é uma Árvore de Busca Binária que insere e remove de forma inteligente, para assegurar que a Árvore permaneça balanceada.

Árvore Rubro-Negra

- Os nós da Árvore possuem um **atributo** de cor que pode ser **vermelho** ou **preto**;



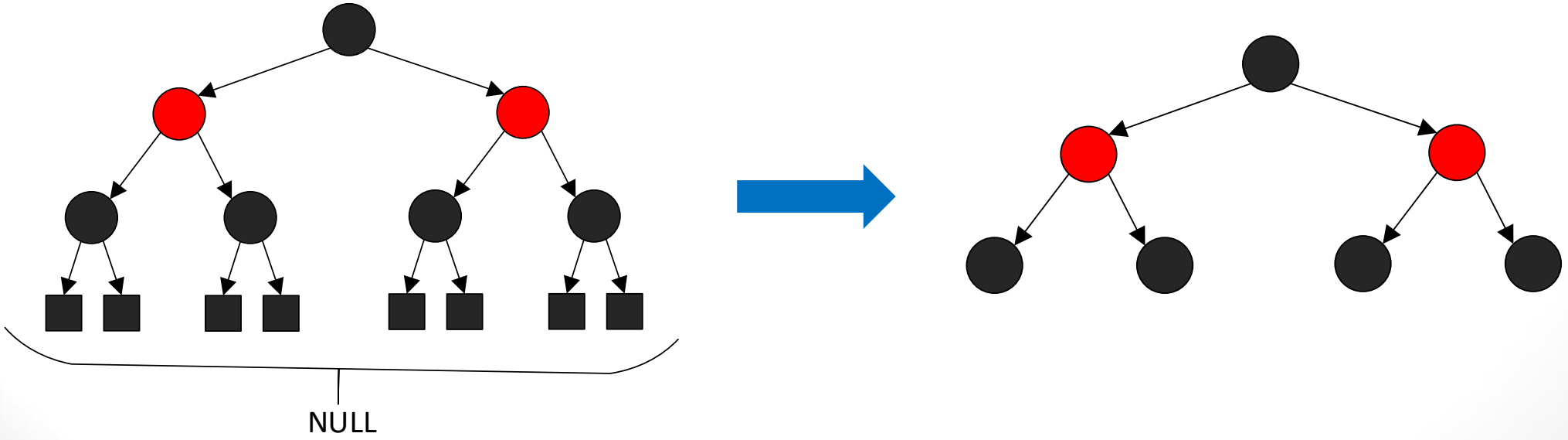
- Propriedades da Árvore:
  - Todo nó da Árvore é **vermelho** ou **preto**;
  - A raiz é sempre **preta**;
  - Todo nó folha (**NULL**) é **preto**;
  - Se um nó é **vermelho**, então seus filhos são **pretos**, ou seja, não existem nós **vermelhos** consecutivos;
  - Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém o mesmo número de nós **pretos**.

Árvore Rubro-Negra

- Árvore Rubro-Negra permite balanceamento local:
  - Apenas a parte afetada pela inserção ou remoção é rebalanceada;
  - Uso de rotação e ajuste de cores na etapa de rebalanceamento;
  - Essas operações corrigem as propriedades da Árvore que foram violadas.
- A Árvore Rubro-Negra busca manter-se como uma Árvore Binária quase completa:
  - O custo de qualquer algoritmo é no máximo $\boldsymbol{O(log\ n)}$.

Árvore Rubro-Negra

- Todos os ponteiros do nó folhas (**NULL**) são pretos;
- Como todo nó folha termina com dois ponteiros para NULL, eles podem ser ignorados na representação da Árvore para fins didáticos:

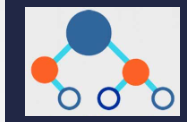


Árvore Rubro-Negra

- AVL x Rubro-Negra
  - Na teoria possuem a mesma complexidade computacional:

    Inserção, remoção e busca = $\boldsymbol{O(log\ n)}$

- 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 Rubro-Negra, o que acelera a operação de busca;
- Por possuir um balanceamento mais rígido, a AVL tem maior custo na operação de inserção e remoção: no pior caso, uma operação de remoção pode exigir $\boldsymbol{O(log\ n)}$ rotações na AVL, mas apenas três rotações na Rubro-Negra.

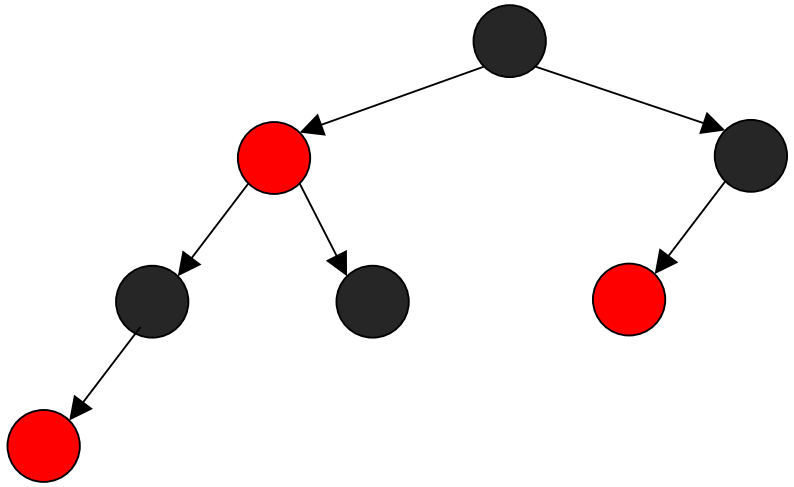


Árvore Rubro-Negra

- AVL x Rubro-Negra
  - Se a aplicação realiza de forma intensa a operação de busca, é melhor usar uma Árvore AVL;
  - Se a operação mais usada é a inserção ou remoção, use uma Árvore Rubro-Negra, ela trabalha melhor com essas operações;
  - Árvores Rubro-Negras são de uso mais geral do que as Árvores AVL. Isso faz com que elas sejam utilizadas em diversas aplicações de bibliotecas de linguagens de programação:
    - Java: `java.util.TreeMap, java.util.TreeSet;`
    - C++: `map, multmap, multset;`
    - Linux Kernel: `completely fair sheduler, linux/rbTree.h`

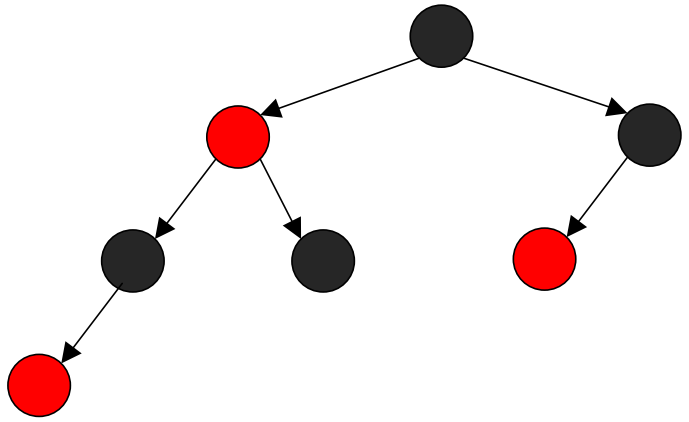
Árvore Rubro-Negra caída para a esquerda - LLRB

- Do inglês, **left-leaning red-black tree**;
- Desenvolvida por Robert Sedgewick em 2008;
- É uma variante da Árvore Rubro-Negra que garante a mesma complexidade de operação, mas possui uma implementação mais simples na inserção e remoção de nós:



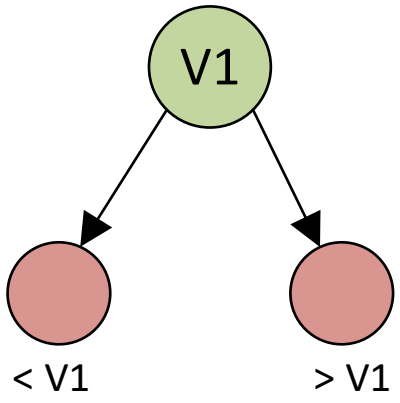
Árvore Rubro-Negra caída para a esquerda - LLRB

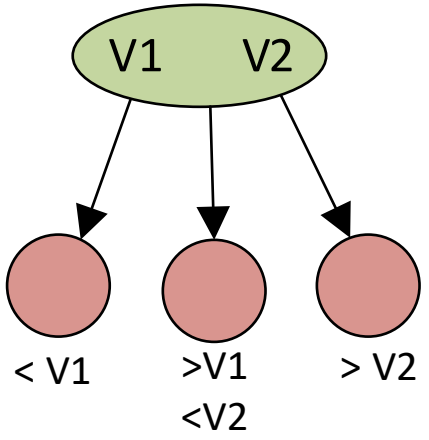
- Este tipo de Rubro-Negra, satisfaz todas as propriedades da Árvore Rubro-Negra convencional;
- Possui uma propriedade extra: se um nó é **vermelho**, então ele é o **filho esquerdo** do seu **pai**;
- Essa propriedade confere o seu aspecto de “caída para a esquerda”: os nós **vermelhos** sempre são **filhos** à **esquerda**.



Árvore Rubro-Negra caída para a esquerda - LLRB

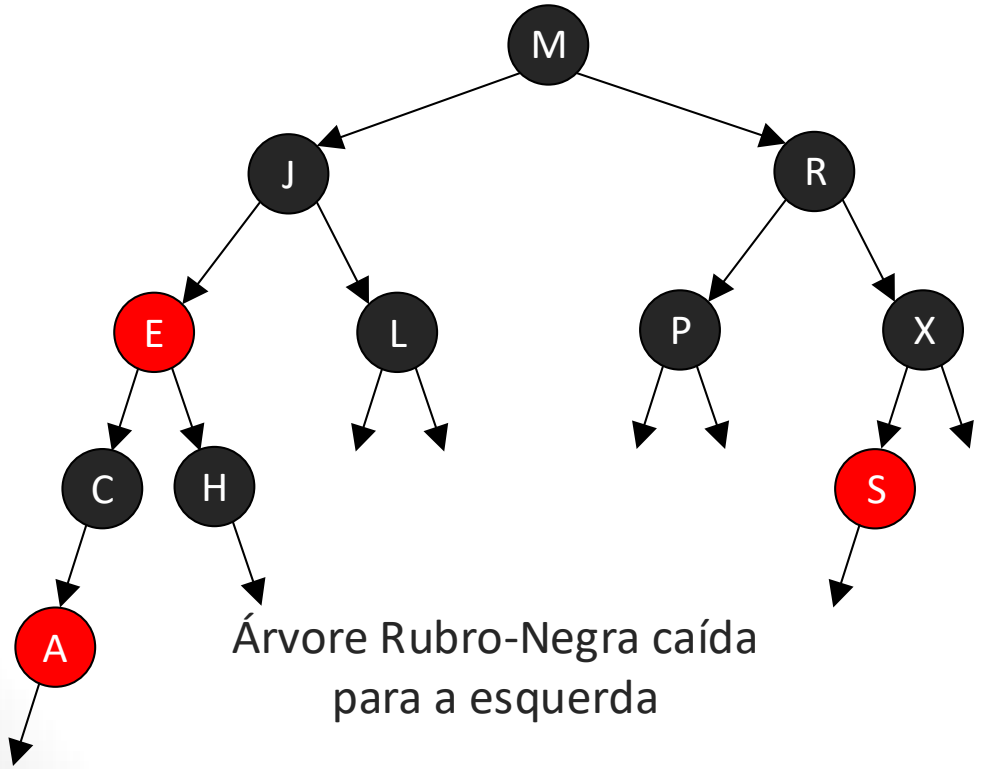
- Sua implementação corresponde a implementação de uma Árvore 2-3, que não é uma Árvore Binária;
- Em uma Árvore 2-3, cada nó interno pode armazenar um ou dois valores e, dependendo da quantidade de valores armazenados, ter dois (um valor) ou três (dois valores) filhos;
- Tem o funcionamento igual ao da Árvore Binária de Busca. No caso de 3 Sub-Árvores, na Sub-Árvore do meio se encontram os elementos que são maiores do que o primeiro, mas menores do que o segundo.



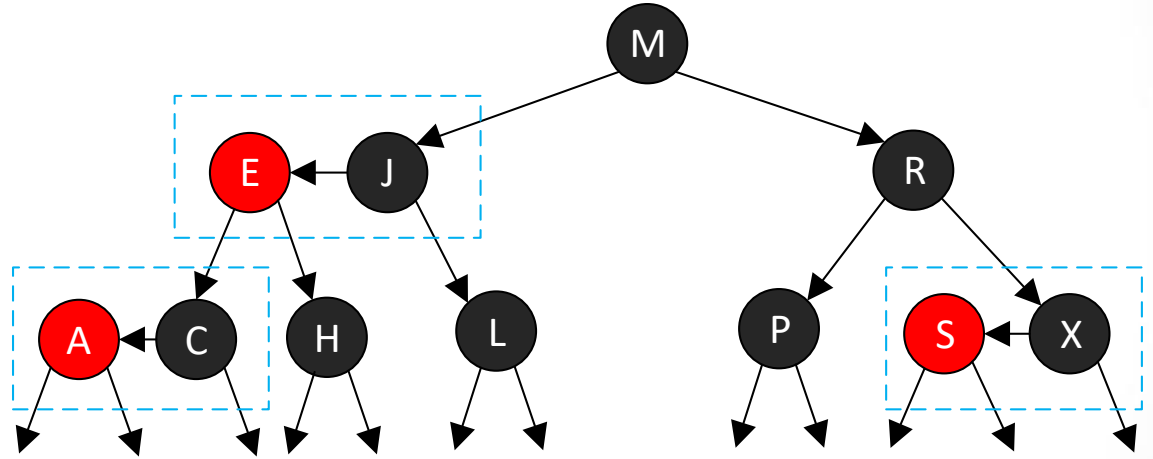


Árvore Rubro-Negra caída para a esquerda - LLRB

- Sua implementação corresponde a implementação de uma Árvore 2-3, que não é Binária:



Árvore Rubro-Negra caída para a esquerda



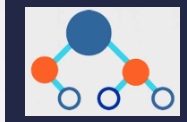
Árvore 2-3

Sempre que existir um nó com filho a esquerda vermelho, é como se estivéssemos em um nó de uma Árvore 2-3 com 2 valores armazenados.

Árvore Rubro-Negra caída para a esquerda - LLRB

- Sua implementação corresponde a implementação de uma Árvore 2-3 se considerarmos que o nó vermelho sempre será o valor menor de um nó contendo dois valores e três Sub-Árvores;

- Assim balancear a Árvore Rubro-Negra equivale a manipular uma Árvore 2-3, uma tarefa muito mais simples do que manipular uma Árvore AVL ou uma Rubro-Negra convencional.



Implementando uma Árvore Rubro-Negra caída para a esquerda - LLRB

- A implementação é idêntica a da Árvore Binária e da AVL;
- Para guardar o primeiro nó da Árvore utilizamos um ponteiro para ponteiro;
- Um ponteiro para ponteiro pode guardar o endereço de um ponteiro, facilitando a mudança da raiz da Árvore se for necessário.

Implementando uma Árvore Rubro-Negra caída para a esquerda - LLRB

- arvoreLLRB.h
  - Os protótipos das funções disponíveis para trabalhar com esse tipo de Árvore que serão implementadas no arquivo **arvoreLLRB.c**;
  - O tipo de dado armazenado na Árvore;
  - O ponteiro Árvore.
- arvoreLLRB.c
  - As chamadas às bibliotecas necessárias a implementação da Árvore Rubro-Negra;
  - A definição de duas constantes para representar as cores dos nós da Árvore;
  - A definição do tipo que descreve cada nó a árvore – **struct NO**;
  - Implementação de suas funções definidas no arquivo **arvoreLLRB.h**.
- Com exceção da inserção e remoção, as demais funções da Árvore Rubro-Negra são idênticas as da Árvore Binária

Implementando uma Árvore Rubro-Negra caída para a esquerda - LLRB

```c
//Arquivo arvoreLLRB.h
typedef struct NO *arvoreLLRB;
```

```c
//programa principal
arvoreLLRB *raiz;
```

```c
//Arquivo arvoreLLRB.c
#include <stdio.h>
#include <stdlib.h>
#include "arvoreLLRB.h"

#define RED 1
#define BLACK 0

struct NO{
    int info;
    struct NO *esq;
    struct NO *dir;
    int cor;
};
```

Implementando uma Árvore Rubro-Negra caída para a esquerda – LLRB – Funções Auxiliares

Acessando a cor de um nó:

```
//funções auxiliares
//Arquivo arvoreLLRB.c
int cor(struct NO *H){
    if(H == NULL){
        return BLACK;
    }else{
        return H->cor;
    }
}
```

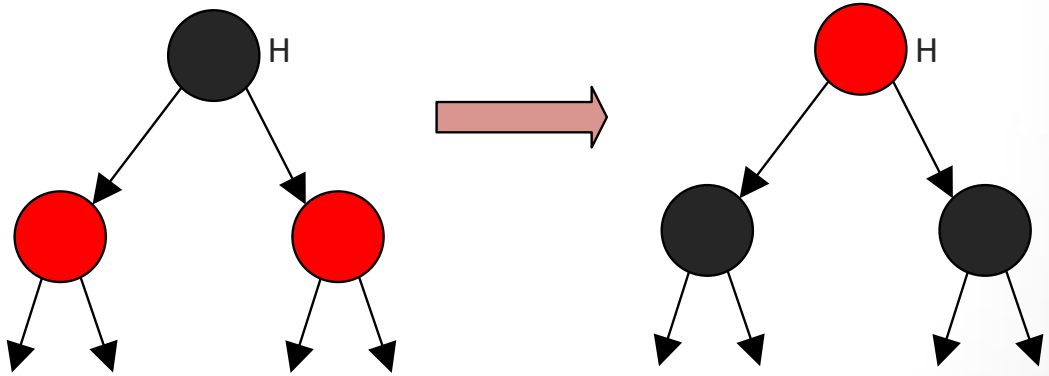
Inverte a cor de um nó Pai e de seus Filhos.
Esta é uma operação administrativa: não altera
A estrutura ou o conteúdo da Árvore.

```
//funções auxiliares
//Arquivo arvoreLLRB.c
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;
    }
}
```

Implementando uma Árvore Rubro-Negra caída para a esquerda – LLRB – Funções Auxiliares

- Inverte a cor de um nó Pai e de seus Filhos. Esta é uma operação administrativa: não altera a estrutura ou o conteúdo da Árvore. Basicamente, ela inverte a cor do nó pai com uma operação de negação e verifica se cada um dos seus filhos (dir e esq) existem. Se existirem a função também inverte sua cor.

```
//funções auxiliares
//Arquivo arvoreLLRB.c
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;
    }
}
```



Esta é uma operação “administrativa”, já que não altera o conteúdo da árvore

Rotação da Árvore Rubro-Negra caída para a esquerda – LLRB

- Operação básica para balanceamento
  - Temos apenas dois tipos de rotação:
    - Rotação à esquerda;
    - Rotação à direita.

A Árvore Rubro-Negra possui outras funções que auxiliam em seu balanceamento.

- A Árvore Rubro-Negra tem uma depuração mais simples de se implementar se comparada a Árvore AVL.
  - Dado um conjunto de três nós, a rotação visa deslocar um nó vermelho que esteja à esquerda, para a direita, e vice e versa.
  - A rotação apenas atualiza ponteiros, de modo que sua complexidade é $O(1)$.

Rotação da Árvore Rubro-Negra caída para a esquerda – LLRB

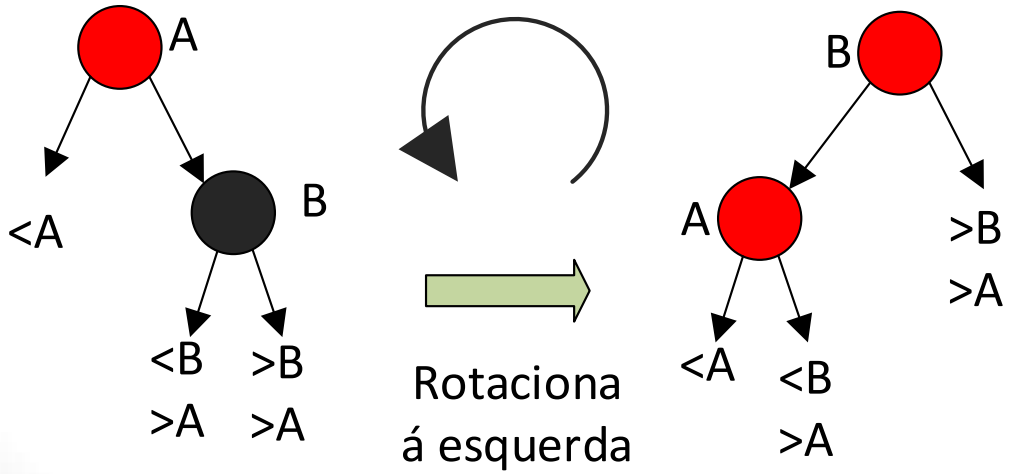
- Rotação à esquerda:
  - Recebe o nó “A” com “B” como filho direito;
  - Move “B” para o lugar de “A”, “A” se torna filho esquerdo de “B”;
  - “B” recebe a cor de “A”, e o nó “A” fica vermelho.

```
//funções auxiliares
//Arquivo arvoreLLRB.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;
}
```

Rotação da Árvore Rubro-Negra caída para a esquerda – LLRB

- Rotação à esquerda:
  - Recebe o nó “A” com “B” como filho direito;
  - Move “B” para o lugar de “A”, “A” se torna filho esquerdo de “B”;
  - “B” recebe a cor de “A”, “A” fica vermelho.

```c
//funções auxiliares
//Arquivo arvoreLLRB.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;
}
```



A rotação se preocupa apenas em rotacionar um conjunto de nós. Uma rotação, dependendo das cores dos nós antes da rotação, pode criar uma nova violação de propriedades.

A rotação não corrige todos os problemas, e ainda pode gerar um outro problema que pode ser corrigido por uma rotação, ou uma outra tarefa como por exemplo uma troca de cores.

Rotação é apenas uma parte do balanceamento.

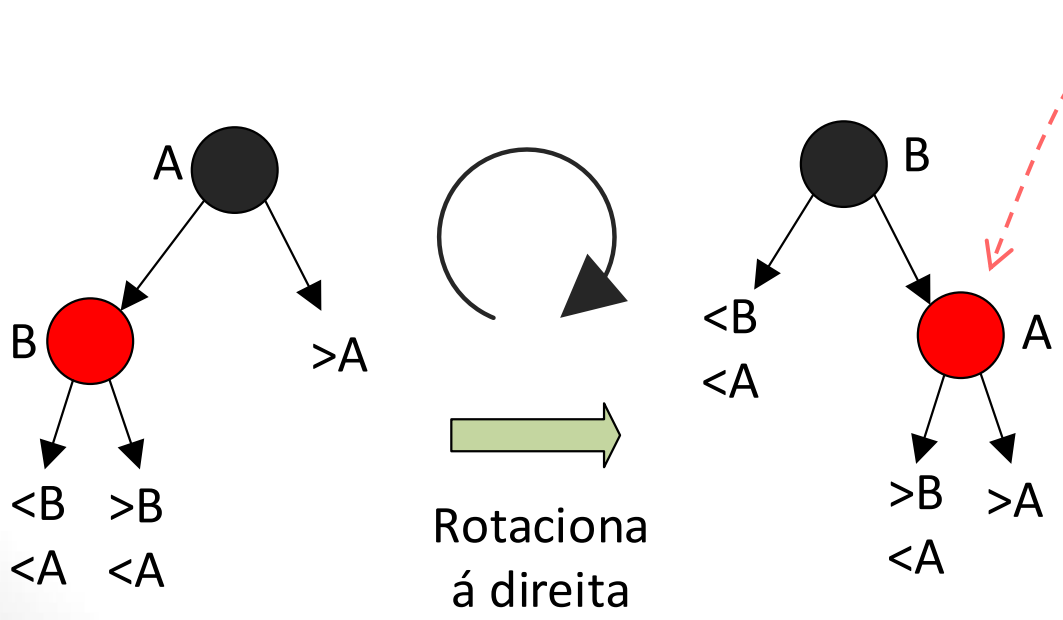
Rotação da Árvore Rubro-Negra caída para a esquerda – LLRB

- Rotação à direita:
  - Recebe o nó “A” com “B” como filho esquerdo;
  - Move “B” para o lugar de “A”, “A” se torna filho direito de “B”;
  - “B” recebe a cor de “A”, “A” fica vermelho.

```
//funções auxiliares
//Arquivo arvoreLLRB.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;
}
```

Rotação da Árvore Rubro-Negra caída para a esquerda – LLRB

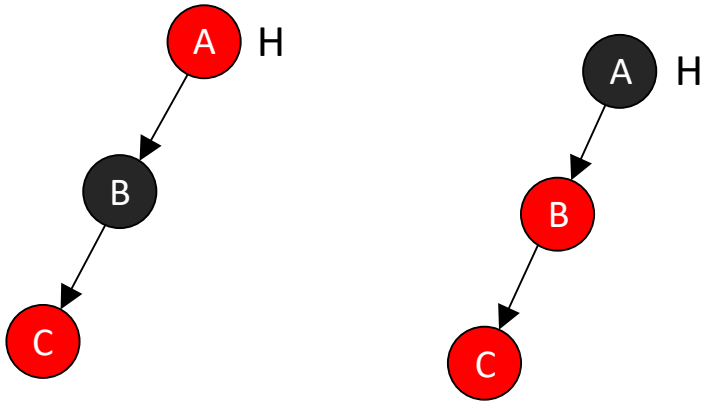
- Rotação à direita:
  - Recebe o nó “A” com “B” como filho esquerdo;
  - Move “B” para o lugar de “A”, “A” se torna filho direito de “B”;
  - “B” recebe a cor de “A”, “A” fica vermelho.




```c
//funções auxiliares
//Arquivo arvoreLLRB.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;
}
```

Árvore Rubro-Negra caída para a esquerda – LLRB

- Movendo os nós Vermelhos:
  - Rotações e troca de cores podem causar uma violação das propriedades da Árvore;
  - Por exemplo, a função `trocaCor()` pode introduzir sucessivos nós vermelhos à esquerda, o que viola uma das propriedades da Árvore.
  - Para resolver este problema, a árvore Rubro-Negra possui outras funções, além das funções de rotação, que ajudam a reestabelecer o balanceamento da Árvore e garantem que suas propriedades sejam respeitadas.



Árvore Rubro-Negra caída para a esquerda – LLRB

- Movendo os nós vermelhos
- Temos a necessidade de outras **três funções**, além da rotação, para reestabelecer o balanceamento da Árvore e garantir que as suas propriedades sejam respeitadas:
  - Mover o nó vermelho para a esquerda;
  - Mover o nó vermelho para a direita;
  - Acertar o balanceamento geral de uma Sub-Árvore ou uma Árvore inteira.

- Essas funções movimentam um nó vermelho para a Sub-Árvore esquerda ou direita, dependendo da situação em que ele se encontre.

Árvore Rubro-Negra caída para a esquerda – LLRB

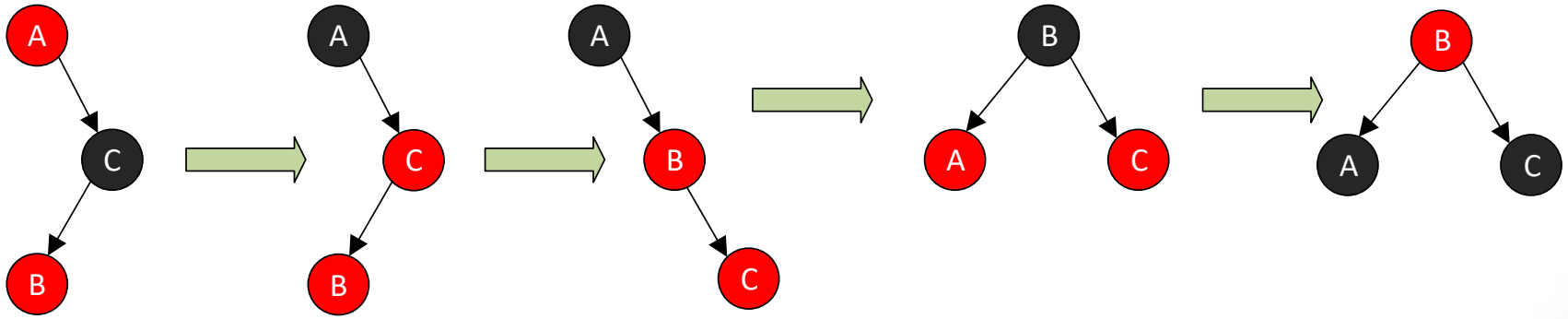
- Mover um nó vermelho para a esquerda:
  - Recebe um nó “H” e troca as cores dele e de seus filhos;
  - Se o filho à esquerda do filho direito é vermelho, aplicar uma rotação à direita no filho direito e uma rotação esquerda no Pai;
  - Trocar as cores do nó Pai e de seus filhos.

```
//Arquivo arvoreLLRB.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;
}
```

Árvore Rubro-Negra caída para a esquerda – LLRB

- Mover um nó vermelho para a esquerda:
  - Recebe um nó “A”, representado pelo ponteiro “H”, e troca as cores dele e de seus filhos;
  - Se o filho à esquerda (B) do filho direito (C) de A é vermelho, aplicar uma rotação à direita no filho direito (C) e uma rotação esquerda no Pai (A). Assim o nó B se torna pai de A e C, sendo A filho esquerdo;
  - Por fim, trocar as cores do nó Pai e de seus filhos.

```c
//Arquivo arvoreLLRB.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;
}
```



Troca a cor em A

Rotaciona a direita em C

Rotaciona a esquerda em A

Troca a cor em B

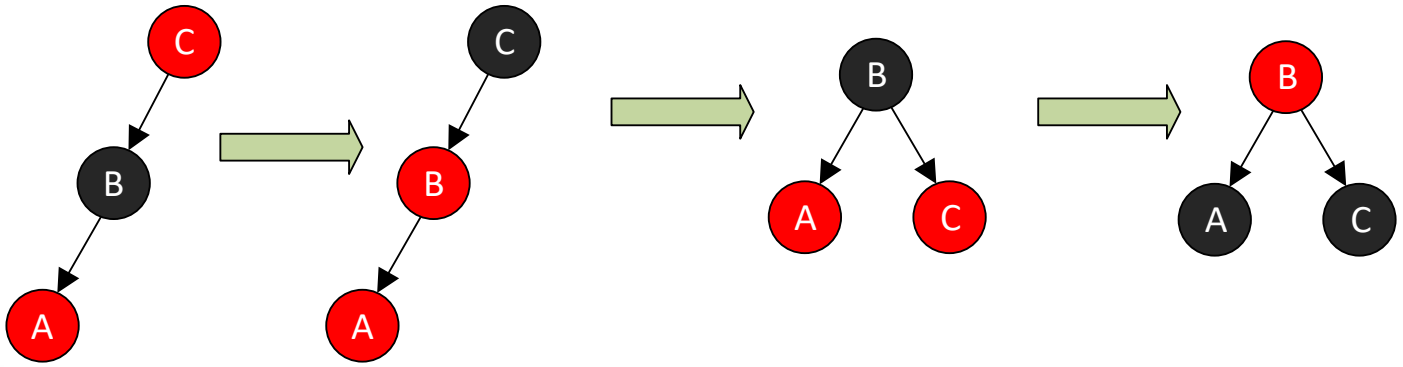
Árvore Rubro-Negra caída para a esquerda – LLRB

- Mover um nó vermelho para a direita:
  - Recebe um nó “H” e troca as cores dele e de seus filhos;
  - Se o filho à esquerda do filho esquerdo é vermelho, aplicar uma rotação à direita do Pai;
  - Trocar as cores do nó Pai e de seus filhos.

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

Árvore Rubro-Negra caída para a esquerda – LLRB

- Mover um nó vermelho para a direita:
  - Recebe um nó “C”, representado pelo ponteiro “H”, e troca as cores dele e de seus filhos;
  - Verifica se o filho à esquerda (A) do filho esquerdo (C) é vermelho, se for aplicar uma rotação à direita do Pai (C). Desse modo, o nó B se torna pai dos nós A e C, sendo o nó C seu filho à direita;
  - Trocar as cores do nó Pai e de seus filhos.



Troca a cor em C

Rotaciona a direita em C

Troca a cor em B

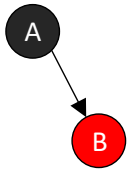
```
//Arquivo arvoreLLRB.c
struct NO *move2DirRED(struct NO *H){
    trocaCor(H);
    if(cor(H->esq->esq) == RED){
        H = rotacionaDireita(H);
        trocaCor(H);
    }
    return H;
}
```

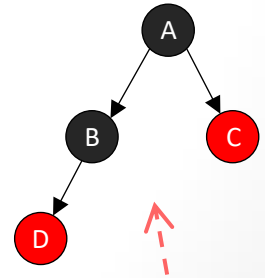
Árvore Rubro-Negra caída para a esquerda – LLRB

- Acertar o Balanceamento
  - Se o filho direito é vermelho: rotação a esquerda;
  - Se o filho direito e o neto da esquerda são vermelhos: rotação à direita – se existir filho à esquerda;
  - Se ambos os filhos são vermelhos: trocar a cor do pai e dos filhos.

```
//Arquivo arvoreLLRB.c
struct NO *balancear(struct NO *H){
    if(cor(H->dir) == RED){
        H = rotacionaEsquerda(H);
    }
    if(H->esq != NULL && cor(H->dir) == RED && cor(H->esq->esq) == RED){
        H = rotacionaDireita(H);
    }
    if(cor(H->esq) == RED && cor(H->dir) == RED){
        trocaCor(H);
    }
    return H;
}
```

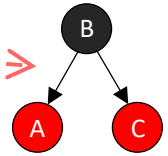
A Nó vermelho é sempre filho à esquerda





Filho da direita e neto da esquerda são vermelhos

Os dois filhos são vermelhos: trocar a cor!



Árvore Rubro-Negra caída para a esquerda – LLRB

- Entregue a atividade parcial no Moodle.