Aula 16
Coleções Informadas
Árvores Red-Black

Algoritmos e Estruturas de Dados

Árvores de Pesquisa

Red-Black

Árvores de Pesquisa *Red-Black*

- Ideia:
  - Melhor a eficiência das árvores de pesquisa binária para o pior caso

  - Garantido que:

    Independentemente da ordem de inserção

    A árvore encontra-se sempre balanceada

Propriedades Árvores Red-Black

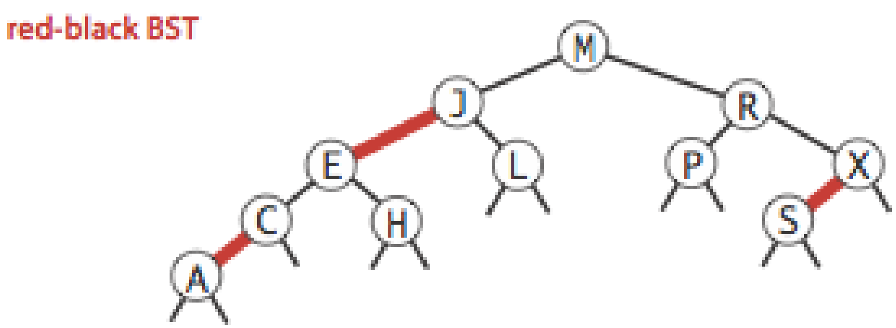
- ***Def:*** Uma Árvore Red-Black (enviesada para a esquerda) é uma
  - *Árvore de Pesquisa Binária com ligações vermelhas e negras com as seguintes restrições*:

    Ligações vermelhas são sempre para a esquerda

    Nenhum nó tem duas ligações vermelhas (ex: pai e filho)

    Qualquer caminho da raiz para um nó vazio tem o <u>mesmo número de ligações negras</u>

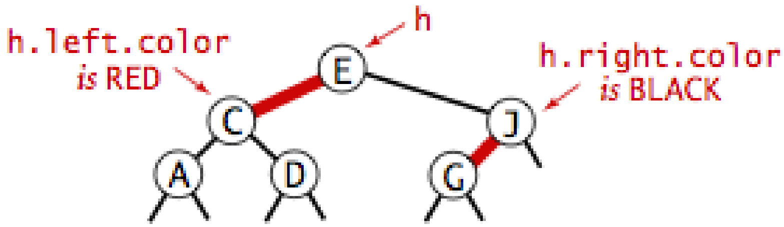
balanceamento negro perfeito



Observação: existem árvores red-black enviesadas para a direita, em que as ligações vermelhas são sempre para a direita. A direção do viés altera ligeiramente a implementação, mas não altera as propriedades mais importantes relativas à complexidade temporal e espacial desta estrutura de dados.

Implementação Árvores Red-Black

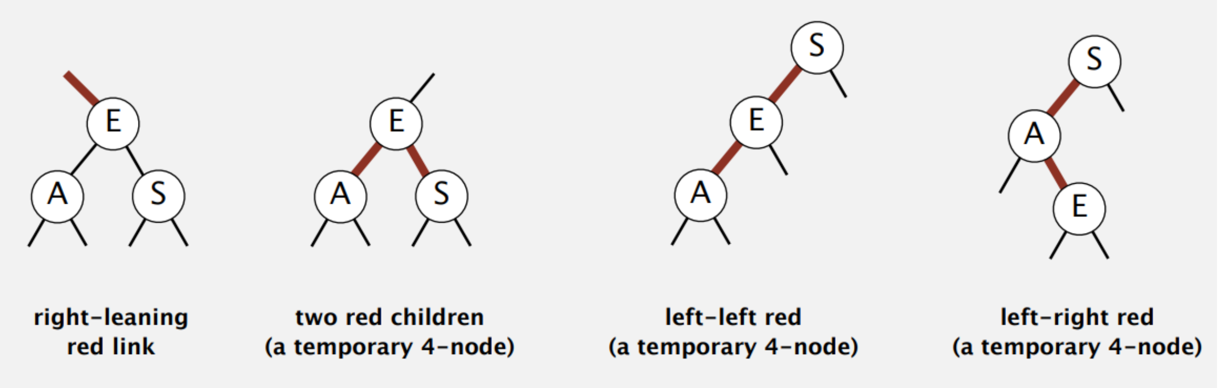
```java
public class RedBlackBST<Key extends Comparable<Key>,Value> {
    private static final boolean RED = true;
    private static final boolean BLACK = false;
    private Node root;

    private class Node
    {
        Key key;
        Value value;
        Node left, right;
        int size;
        boolean color;

        Node(Key key, Value val, int size, boolean color)
        {
            this.key = key;
            this.value = val;
            this.size = size;
            this.color = color;
        }
    }

    private boolean isRed(Node n)
    {
        if (n == null) return false;
        else return n.color == RED;
    }
}
```



Cor de uma ligação é guardada no nó filho

Operação de rotação

- Para implementar operações de inserção na árvore Red-Black
  - Podemos ter temporáriamente ligações que não respeitam as restrições
  - Ex:

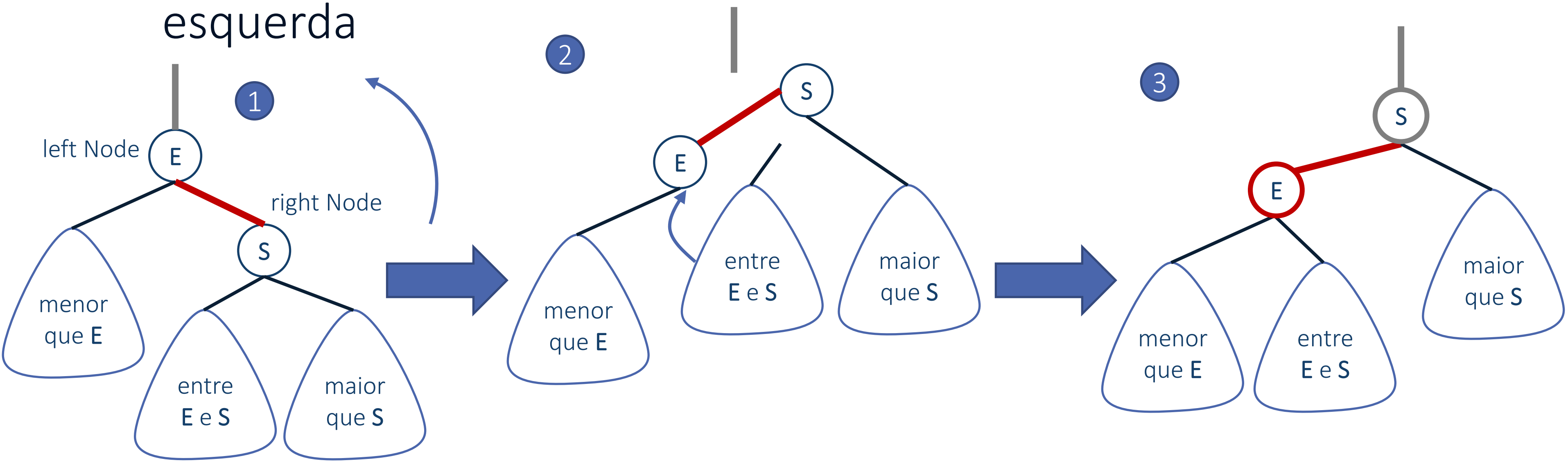


  - Precisamos de definir operações de rotação e mudança de cor

    Para corrigir estes problemas

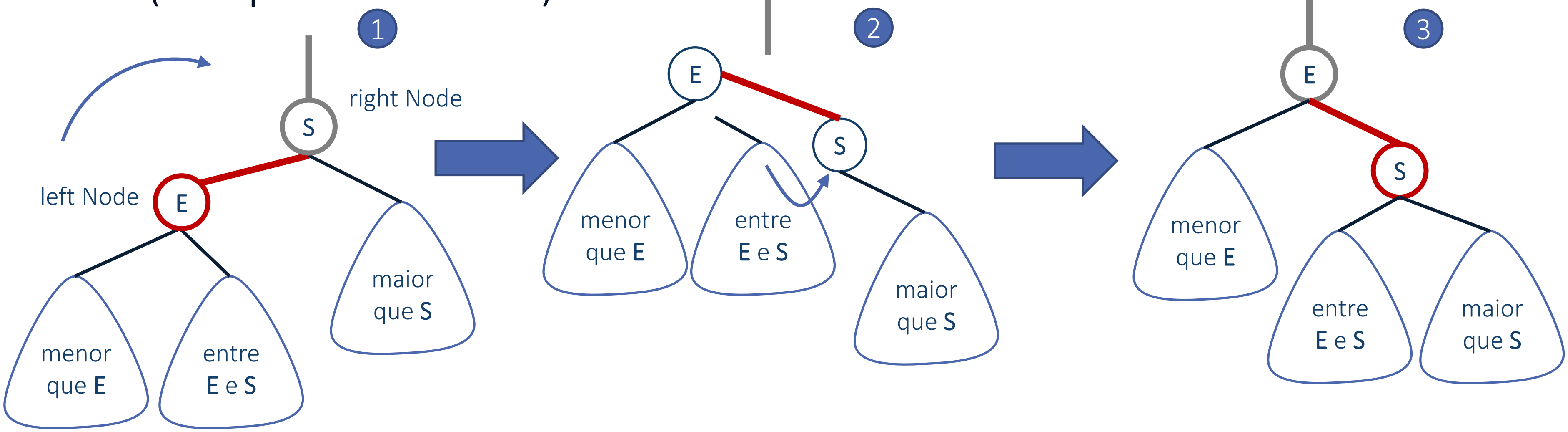
Operação de rotação esquerda

- Rotação para a esquerda
  - Rotação de uma ligação vermelha direita (temporária) para a esquerda



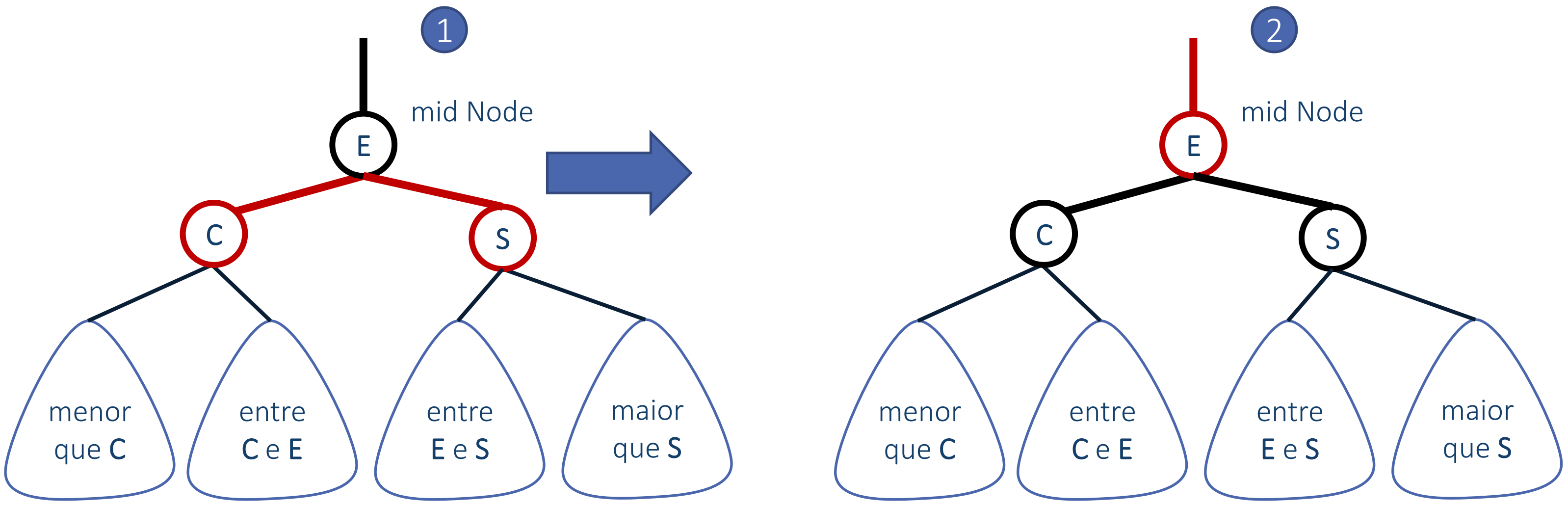
Operação de rotação direita

- Rotação para a direita
  - Rotação de uma ligação vermelha esquerda para uma direita (temporáriamente)



Operação de troca de cor

- Pai com 2 ligações vermelhas para dois filhos

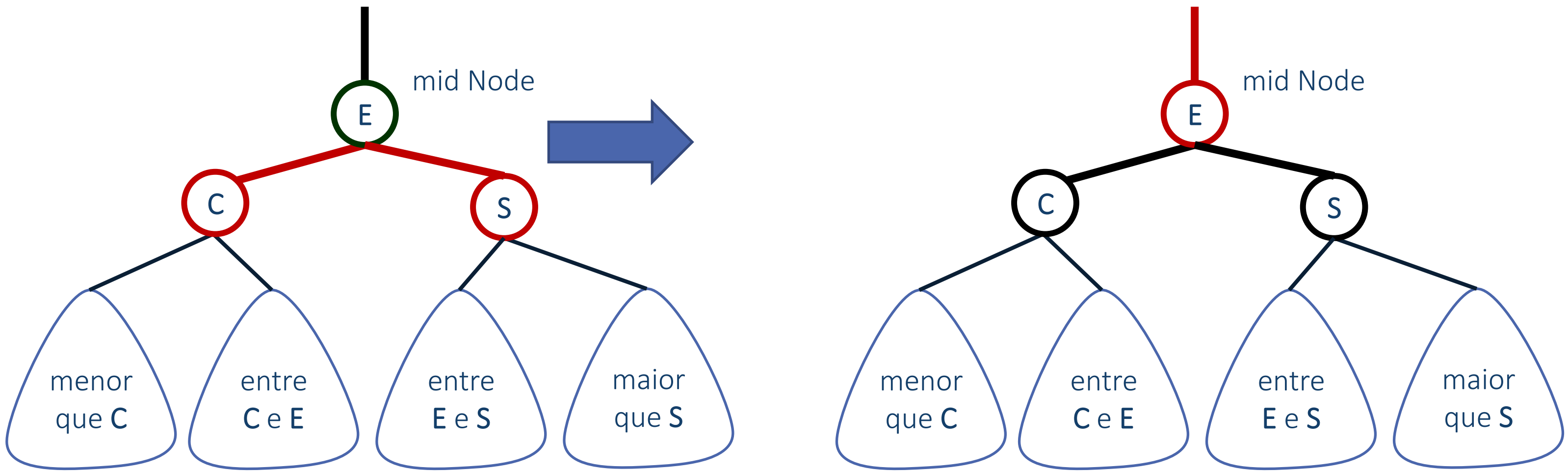


Propriedades Operações

- Operações de rotação e troca de cor
  - Podem ser vistas como tranformações locais

    Preservam ordem

    Preservam balanceamento de ligações negras



Operação de Pesquisa

- get(Key k)
  - Operação de pesquisa implementada exactamente como pesquisa em árvore binária de pesquisa

    Comparar chave k com chave do nó no.chave

    Se k == no.chave

    Encontrámos a chave desejada, retornar o valor associado

    Se k < no.chave

    Procurar no filho esquerdo

    Se k > no.chave

    Procurar no filho direito

    Se chegarmos a um nó vazio

    Não encontramos a chave, retornar null

Operação de Inserção

- *put(Key k, Value v)*
- 3 casos possíveis
  - Inserção em árvore vazia
  - Inserção em nó sem ligações vermelhas
  - Inserção em nó com uma ligação vermelha

Inserção em árvore vazia

- Criação de um novo nó
  - Chave e valor recebidos
  - Sem filhos
  - Nó sem ligação para pai

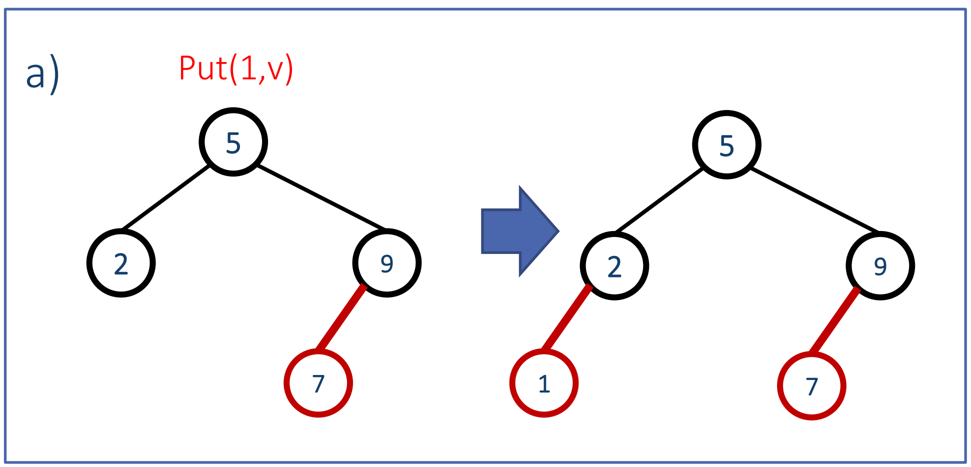
    Por definição, a cor do nó raiz é negra

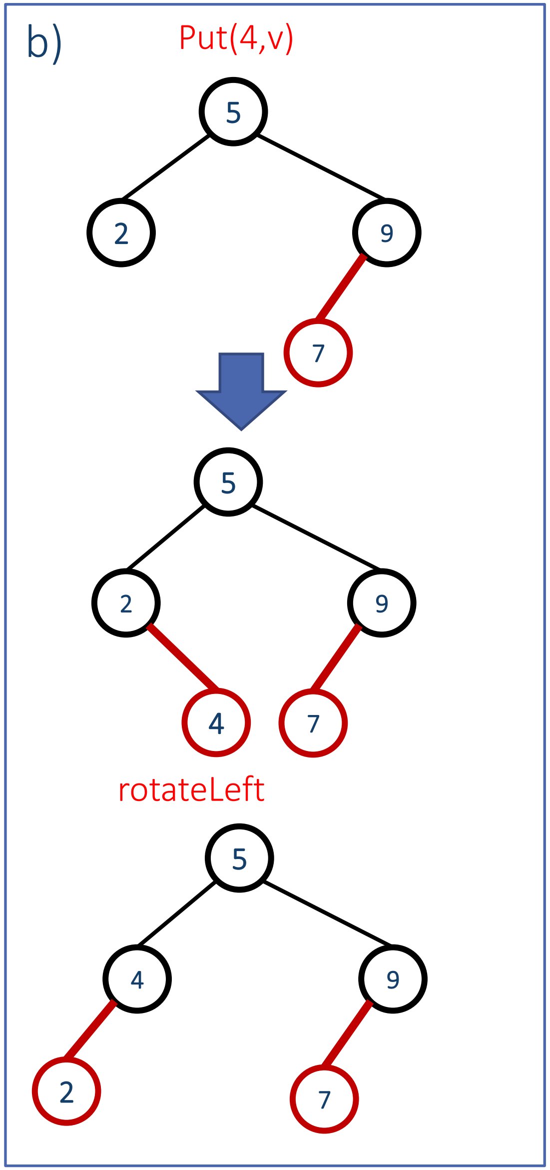
Inserção em nó s/ ligações vermelhas

- Inserção como numa BST
  - Encontrar lugar de inserção
  - Criar nó (ligação vermelha)
  - a) se nó estiver do lado esquerdo

    Não é preciso fazer mais nada
  - b) se nó estiver do lado direito

    Rotate Left





Inserção em nó c/ ligação vermelha

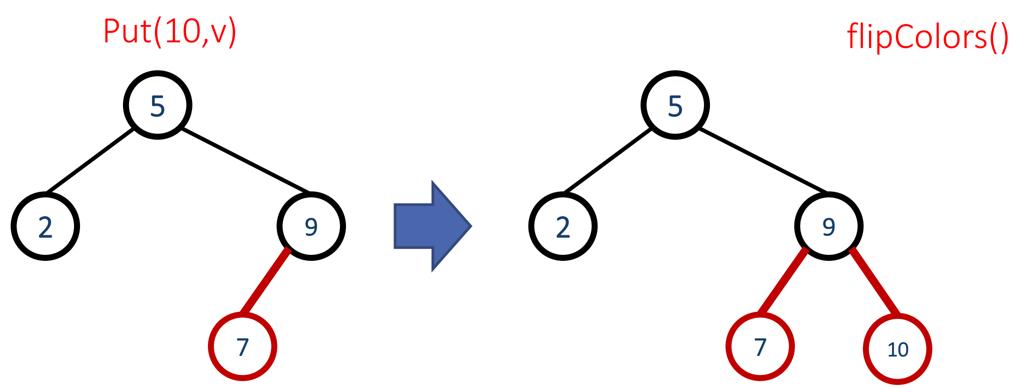
- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

    a) nova chave > chaves existentes na ligação

    Dá origem a um nó com duas ligações vermelhas

    Basta trocar a cor

a)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

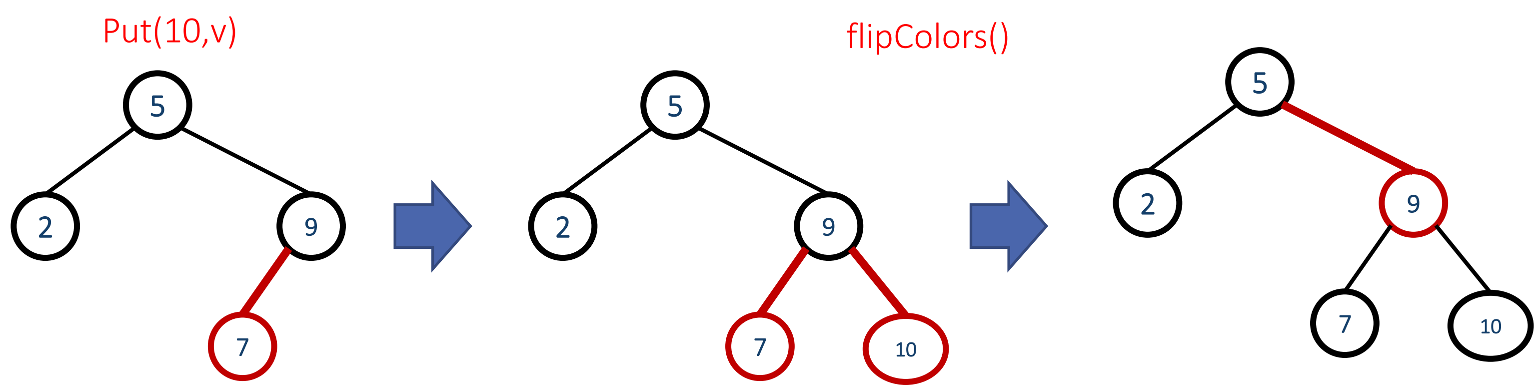
    a) nova chave > chaves existentes na ligação

    Dá origem a um nó com duas ligações vermelhas

    Basta trocar a cor

    Fazer correções recursivamente de baixo para cima

a)



Inserção em nó c/ ligação vermelha

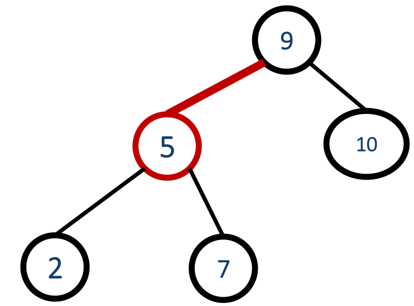
- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

    a) nova chave > chaves existentes na ligação

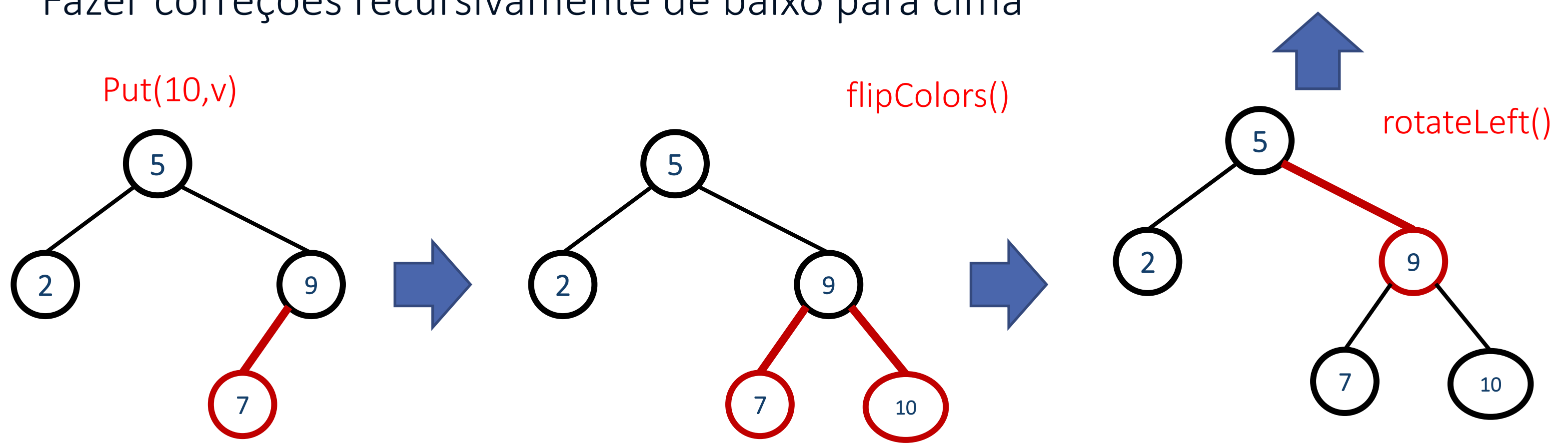
    Dá origem a um nó com duas ligações vermelhas

    Basta trocar a cor

    Fazer correções recursivamente de baixo para cima



a)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

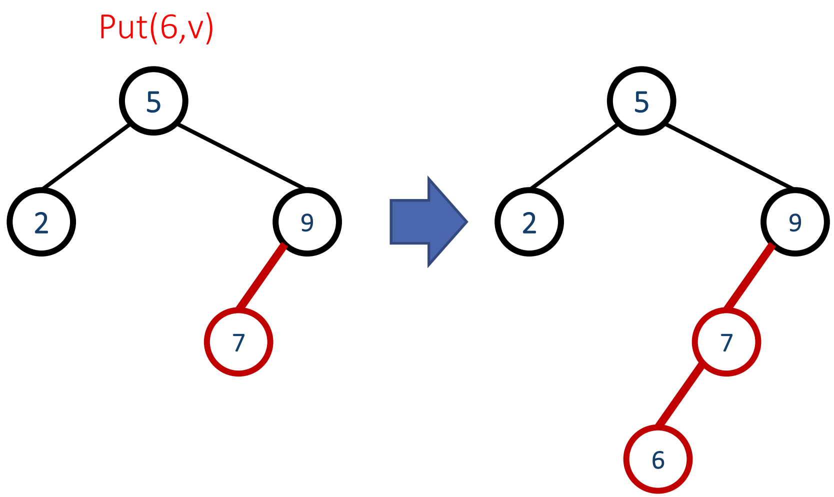
    b) nova chave < chaves existentes na ligação

    Dá origem a 2 ligações vermelhas esquerdas seguidas

    Rodar a ligação superior para a direita

    Trocar a cor

b)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

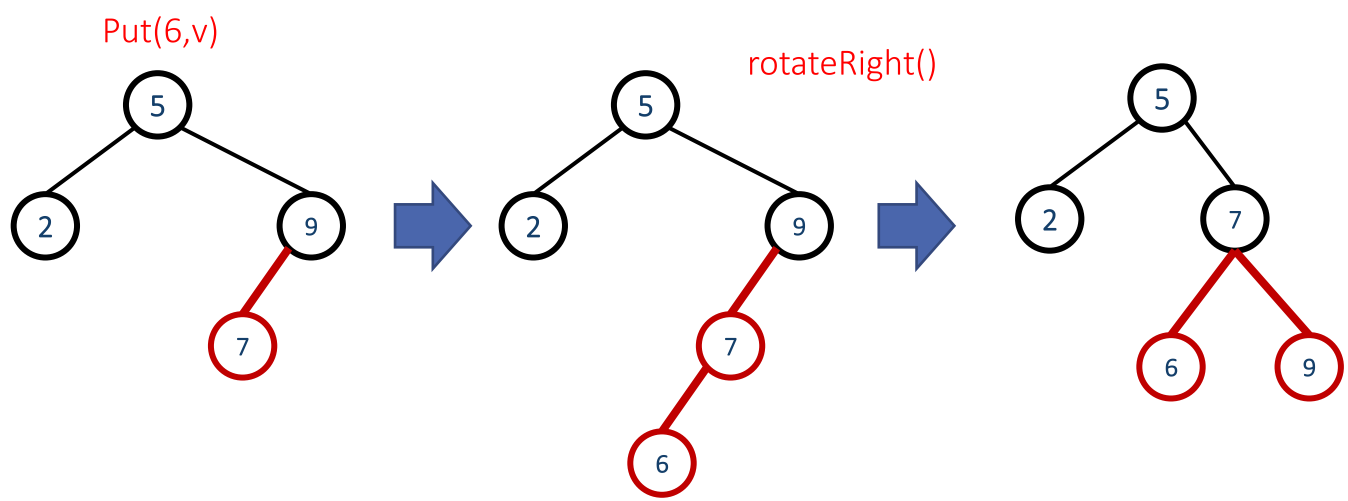
    b) nova chave < chaves existentes na ligação

    Dá origem a 2 ligações vermelhas esquerdas seguidas

    Rodar a ligação superior para a direita

    Trocar a cor

b)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

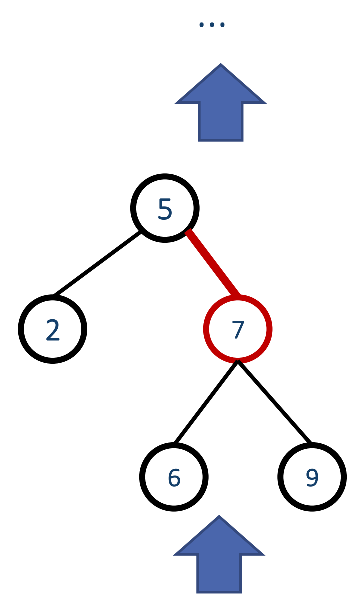
    b) nova chave < chaves existentes na ligação

    Dá origem a 2 ligações vermelhas esquerdas seguidas

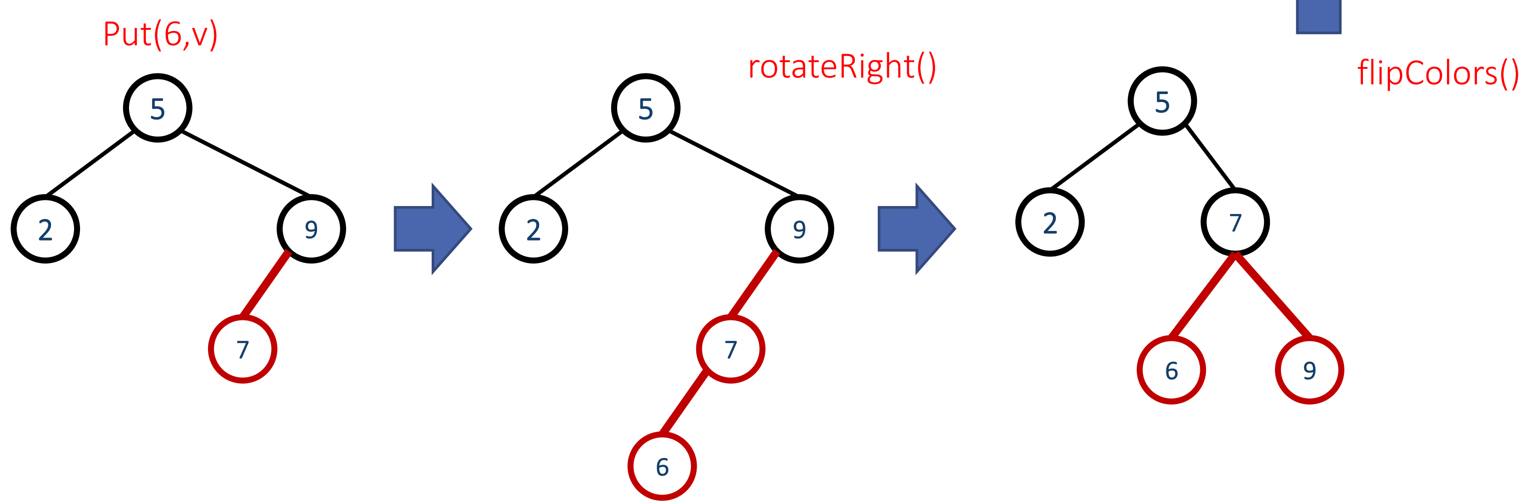
    Rodar a ligação superior para a direita

    Trocar a cor

    Fazer correções recursivamente de baixo para cima



b)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

    c) chave$_{min}$ < nova chave < chave$_{max}$

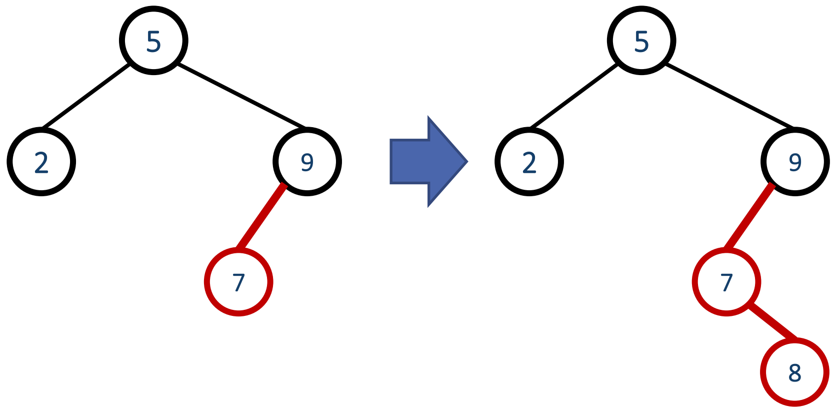
    Dá origem a 2 ligações vermelhas esquerda-direita

    Rodar a ligação inferior para a esquerda (obtem-se o caso b)

    Rodar a ligação superior para a direita

    Trocar a cor

c) Put(8,v)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

    c) $chave_{min}$ < nova chave < $chave_{max}$

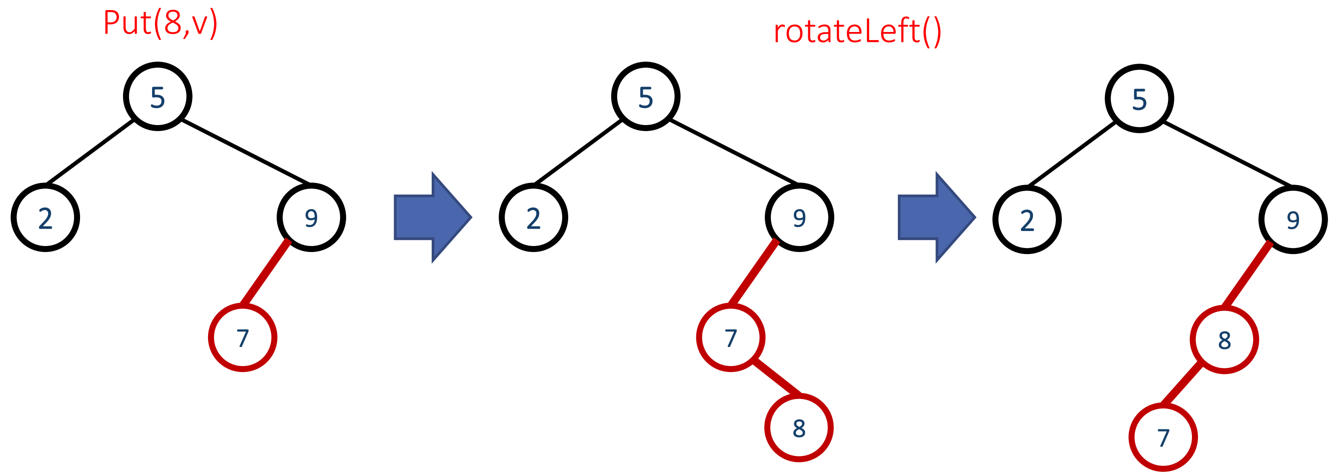
    Dá origem a 2 ligações vermelhas esquerda-direita

    Rodar a ligação inferior para a esquerda (obtem-se o caso b)

    Rodar a ligação superior para a direita

    Trocar a cor

c)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

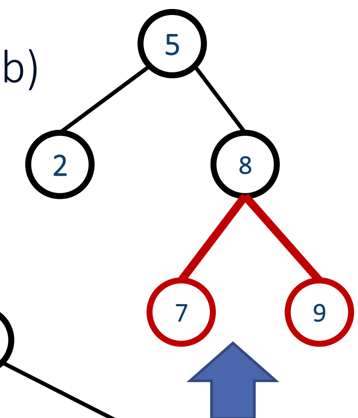
    c) chave$_{min}$ < nova chave < chave$_{max}$

    Dá origem a 2 ligações vermelhas esquerda-direita

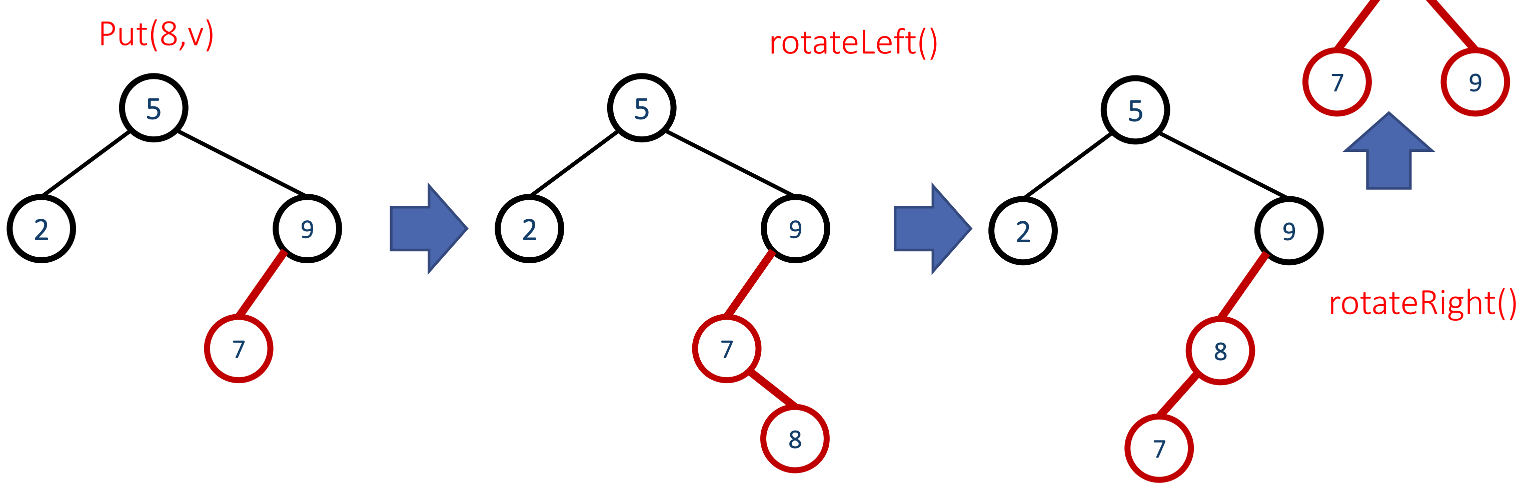
    Rodar a ligação inferior para a esquerda (obtem-se o caso b)

    Rodar a ligação superior para a direita

    Trocar a cor



c)



Inserção em nó c/ ligação vermelha

- Inserção num nó com uma ligação vermelha
  - Novo nó criado com cor vermelha

    c) chave$_{min}$ < nova chave < chave$_{max}$

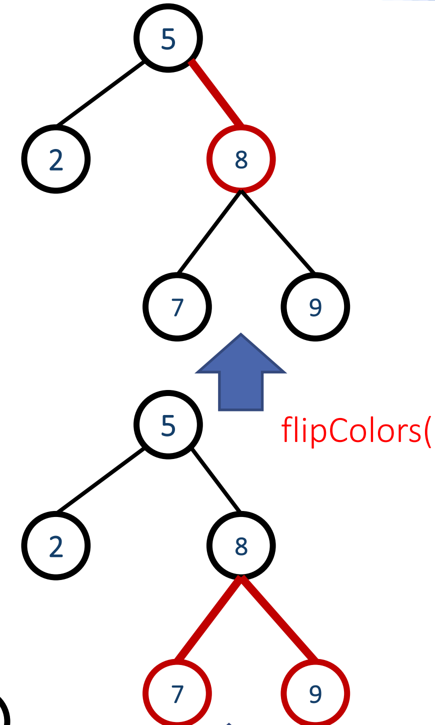
    Dá origem a 2 ligações vermelhas esquerda-direita

    Rodar a ligação inferior para a esquerda (obtem-se o caso b)

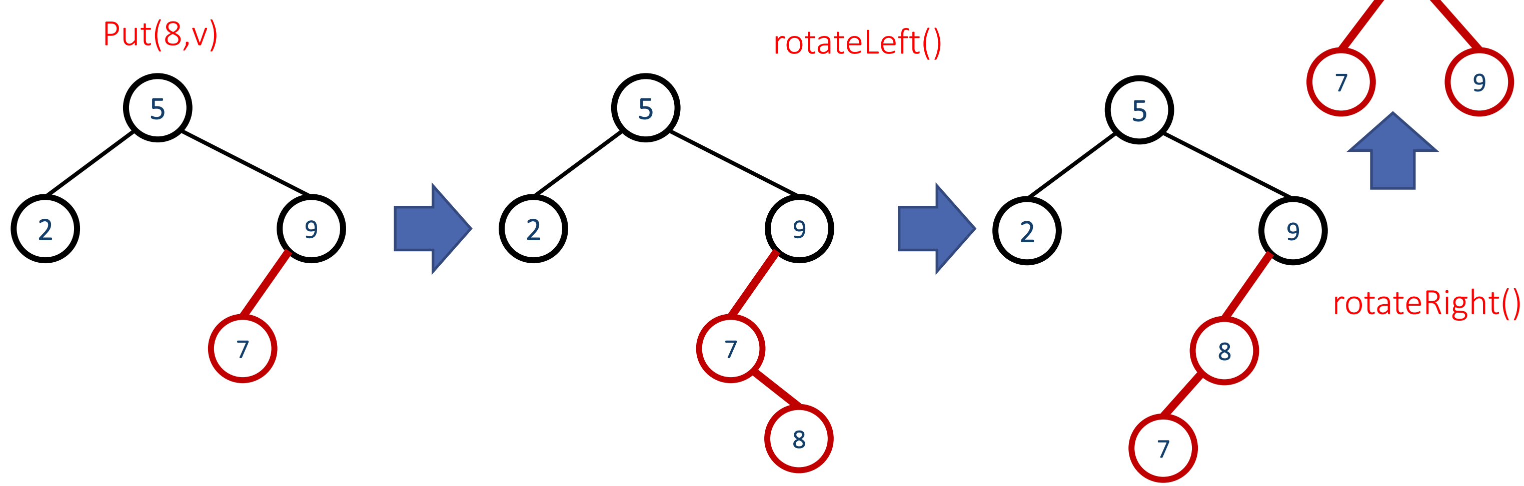
    Rodar a ligação superior para a direita

    Trocar a cor

    Fazer correções recursivamente de baixo para cima



c)



Inserção

- Inserção nó pode ser simplificado para
  - Inserção como numa BST
  - Nó criado com ligação vermelha
  - Resolver problemas com ligações vermelhas de baixo para cima (pela seguinte ordem)

    Única ligação vermelha à direita – rotate left

    Duas ligações vermelhas esquerdas seguidas – rotate right

    Ligação vermelha à esquerda e direita – flip colors

Implementação Inserção

```java
public void put(Key k, Value v)
{
    this.root = put(this.root, k, v);
    this.root.color = BLACK;
}

private Node put(Node n, Key k, Value v)
{
    if(n == null) return new Node(k,v, 1, RED);

    int cmp = k.compareTo(n.key);
    if(cmp == 0) //key already exists, update
    {
        n.value = v;
        return n;
    }
    else if(cmp < 0) n.left = put(n.left, k, v);
    else n.right = put(n.right, k, v);

    //after inserting a node we might need to correct red links
    // a single right red link, rotate left
    if(isRed(n.right) && !isRed(n.left)) n = rotateLeft(n);
    // two left red links, rotate right
    if(isRed(n.left) && isRed(n.left.left)) n = rotateRight(n);
    //two red childs, flip colors
    if(isRed(n.left) && isRed(n.right)) flipColors(n);

    n.size = size(n.left) + size(n.right) + 1;

    return n;
}
```

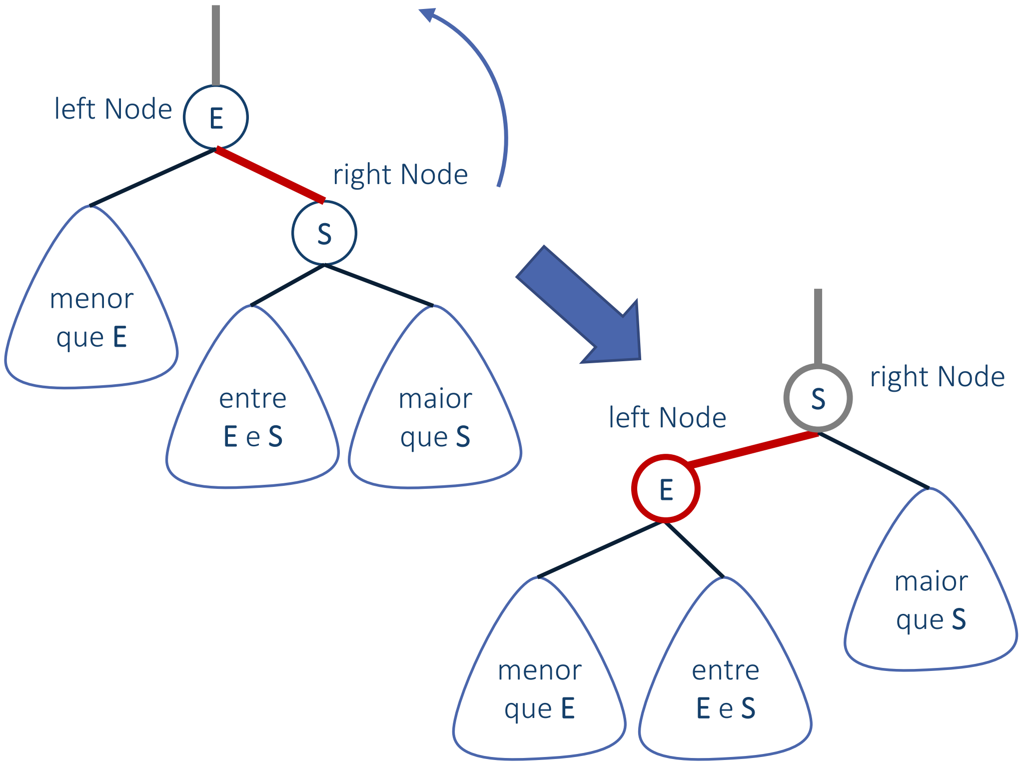
Operação troca de cor

Complexidade temporal assimptótica
O(1)

```
private void flipColors(Node parentNode)
{
    parentNode.color = RED;
    parentNode.left.color = BLACK;
    parentNode.right.color = BLACK;
}
```

Operação rotação esquerda

```
//returns the node that will become
// the new parent,
// after the rotation operation
private Node rotateLeft(Node leftN)
{
    Node rightN = leftN.right;
    //update pointers
    leftN.right = rightN.left;
    rightN.left = leftN;
    //update colors
    rightN.color = leftN.color;
    leftN.color = RED;

    return rightN;
}
```



Operação rotação esquerda

```
//returns the node that will become
// the new parent,
// after the rotation operation
private Node rotateLeft(Node leftN)
{
    Node rightN = leftN.right;
    //update pointers
    leftN.right = rightN.left;
    rightN.left = leftN;
    //update colors
    rightN.color = leftN.color;
    leftN.color = RED;

    return rightN;
}
```

Observação:
A implementação de uma operação de rotação é extremamente eficiente devido a à utilização de nós com ponteiros.

Para mudar uma subárvore de um nó para o outro, basta mudar um par de ponteiros.

Se tivéssemos usado uma representação baseada em arrays para esta árvore red-black (usando o truque que usámos para heaps), teríamos de pagar um preço muito mais elevado, pois teríamos que mover explicitamente todos os elementos que estão na subárvore para outras posições do array.

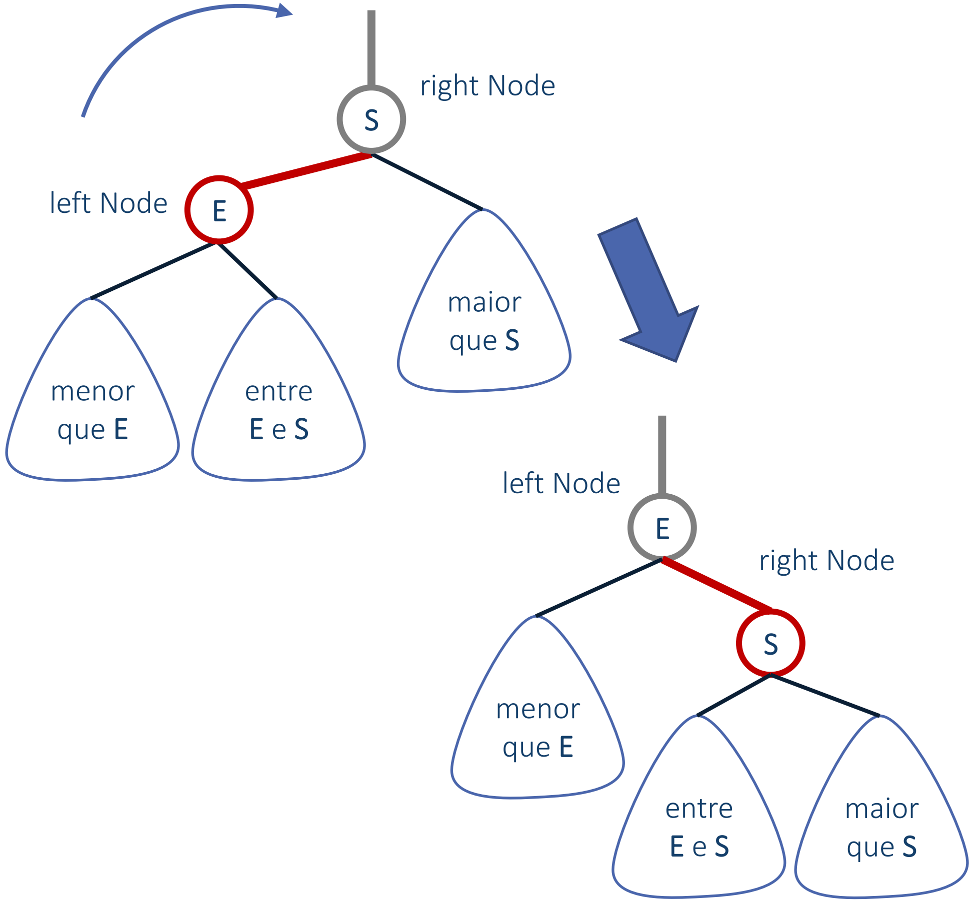
Operação rotação esquerda

```java
//returns the node that will become
// the new parent,
// after the rotation operation
private Node rotateLeft(Node leftN)
{
    Node rightN = leftN.right;
    //update pointers
    leftN.right = rightN.left;
    rightN.left = leftN;
    //update colors
    rightN.color = leftN.color;
    leftN.color = RED;

    return rightN;
}
```

Complexidade temporal assimptótica
O(1)

Operação rotação direita

```java
//returns the node that will become
// the new parent,
// after the rotation operation
private Node rotateRight(Node rightN)
{
    Node leftN = rightN.left;
    //update pointers
    rightN.left = leftN.right;
    leftN.right = rightN;
    //update colors
    leftN.color = rightN.color;
    rightN.color = RED;

    return leftN;
}
```



Operação rotação direita

```
//returns the node that will become
// the new parent,
// after the rotation operation
private Node rotateRight(Node rightN)
{
    Node leftN = rightN.left;
    //update pointers
    rightN.left = leftN.right;
    leftN.right = rightN;
    //update colors
    leftN.color = rightN.color;
    rightN.color = RED;

    return leftN;
}
```

Complexidade temporal assimptótica
O(1)

Complexidade temporal put

```
private Node put(Node n, Key k, Value v)
{
    if(n == null) return new Node(k,v, 1, RED);

    int cmp = k.compareTo(n.key);
    if(cmp == 0) //key already exists, update
    {
        n.value = v;
        return n;
    }
    else if(cmp < 0) n.left = put(n.left, k, v);
    else n.right = put(n.right, k, v);

    //after inserting a node we might need to correct red links
    // a single right red link, rotate left
    if(isRed(n.right) && !isRed(n.left)) n = rotateLeft(n);
    // two left red links, rotate right
    if(isRed(n.left) && isRed(n.left.left)) n = rotateRight(n);
    //two red childs, flip colors
    if(isRed(n.left) && isRed(n.right)) flipColors(n);

    n.size = size(n.left) + size(n.right) + 1;

    return n;
}
```

O(1)

O(1)

O(1)

Chamada recursiva

O(1)

O(1)

O(1)

O(1)

Complexidade temporal put

```
private Node put(Node n, Key k, Value v)
{
    if(n == null) return new Node(k,v, 1, RED);

    int cmp = k.compareTo(n.key);
    if(cmp == 0) //key already exists, update
    {
        n.value = v;
        return n;
    }
    else if(cmp < 0) n.left = put(n.left, k, v);
    else n.right = put(n.right, k, v);

    //after inserting a node we might need to correct red links
}
```

Chamada recursiva

Observação:
Mesmo que tenhamos de fazer as 3 operações de transformação, o custo de processar um nó é um valor constante.

Portanto a complexidade temporal vai depender do número de nós visitados.

Caso a chave que estamos a tentar inserir não exista, vamos percorrer todos os nós de um caminho, a partir da raiz, até a um nó vazio.

Ou seja, a complexidade temporal do método *put* é dada pela **profundidade do caminho.**

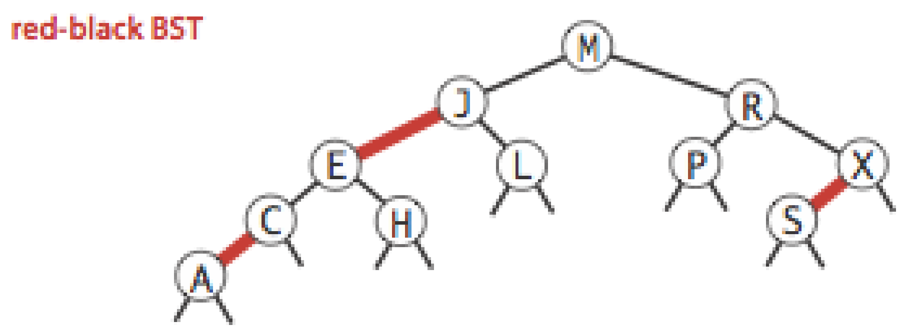
Complexidade temporal

- ***Def:*** Uma Árvore Red-Black é uma
  - *Árvore de Pesquisa Binária com ligações vermelhas e negras com as seguintes restrições*:

    Ligações vermelhas são sempre para a esquerda

    Nenhum nó tem duas ligações vermelhas (ex: pai e filho)

    Qualquer caminho da raiz para um nó vazio tem o <u>mesmo número de ligações negras</u>



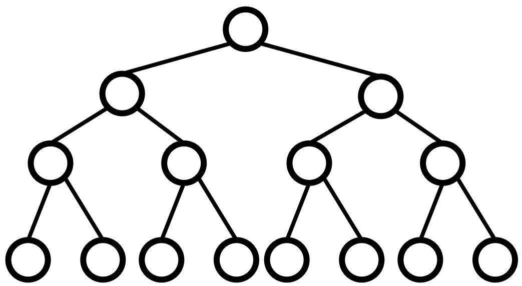
balanceamento negro perfeito

Complexidade Temporal

- Profundidade de Árvore Red-Black
  - Qualquer caminho tem o mesmo número de ligações Negras

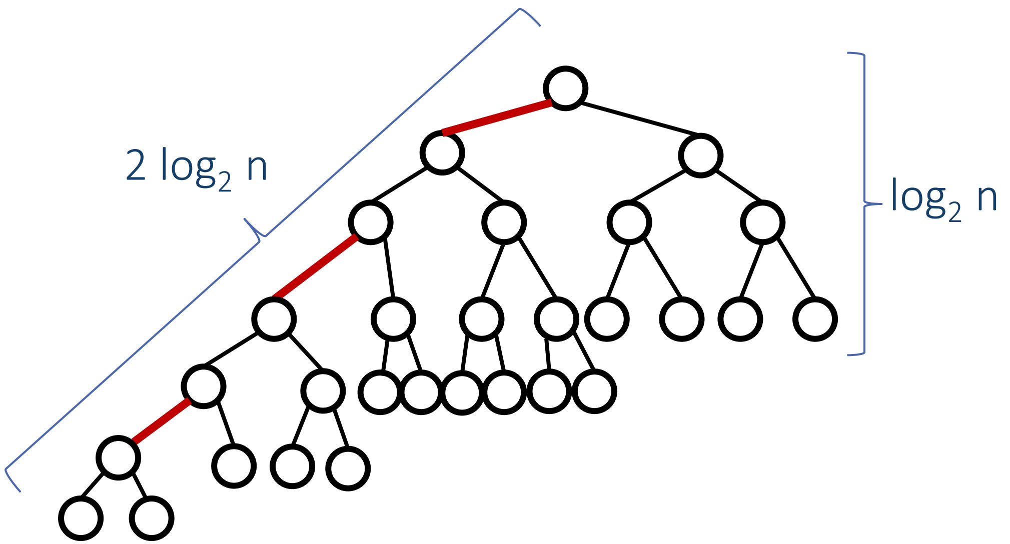
- Melhor caso
  - Todos os nós têm ligações negras

  - Profundidade = $\log_2 n$



Complexidade Temporal

- Profundidade de Árvore Red-Black
  - Qualquer caminho tem o mesmo número de ligações Negras



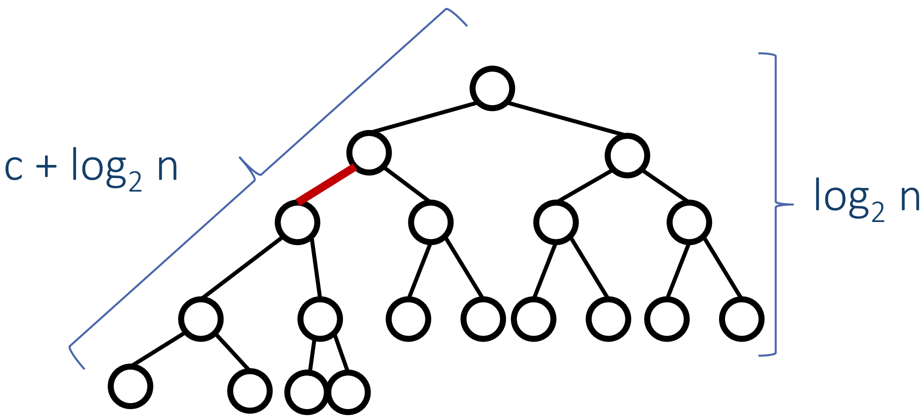
- Pior caso:
  - O caminho da esquerda é todo formado pelo máximo de ligações vermelhas possíveis

    No entanto não podemos ter 2 ligações vermelhas seguidas

  - Profundidade ~ **$2 \log_2 n$**

Complexidade Temporal

- Profundidade de Árvore Red-Black
  - Qualquer caminho tem o mesmo número de ligações Negras



- Caso médio:
  - O número de ligações vermelhas num caminho é substancialmente inferior ao número de ligações negras

  - Profundidade ~ $\mathbf{\log_2 n}$

Complexidade Temporal

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
Array Pesquisa Binária	n	$\log_2 n$	$2n$	$\log_2 n$
Árvore Pesquisa Binária	$1.39 \log_2 n$	$1.39 \log_2 n$	n	n
Árvore 2-3	$c \log_2 n$	$c \log_2 n$	$c \log_2 n$	$c \log_2 n$
Árvore Red-Black	$\mathbf{\log_2 n}$	$\mathbf{\log_2 n}$	$\mathbf{2 \log_2 n}$	$\mathbf{2 \log_2 n}$