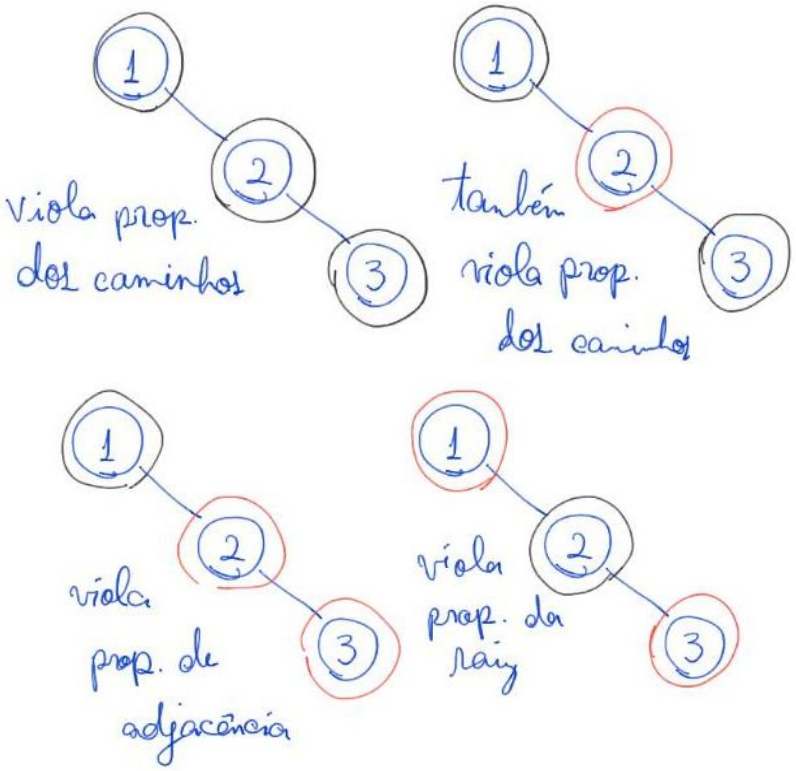
# AED2 - Aula 05
# Árvores rubro-negras

Definição:

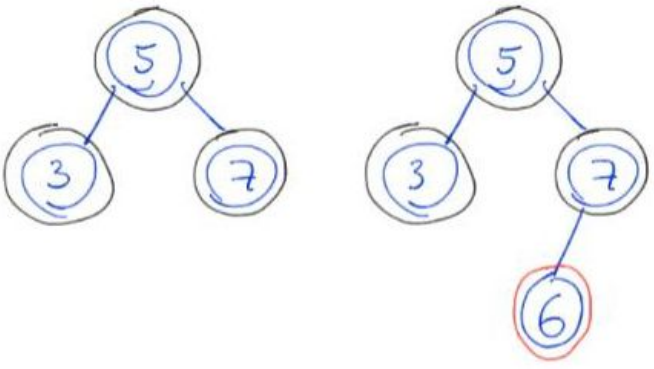
1. cada nó é vermelho ou preto.
2. raiz é sempre preta.
3. dois nós vermelhos não podem ser adjacentes,
    - ou seja, um nó vermelho só pode ter filhos pretos.
4. todo caminho da raiz até um apontador NULL (caminho raiz-NULL)
    - tem o mesmo número de nós pretos.
    - Pense nesses caminhos como buscas mal sucedidas.

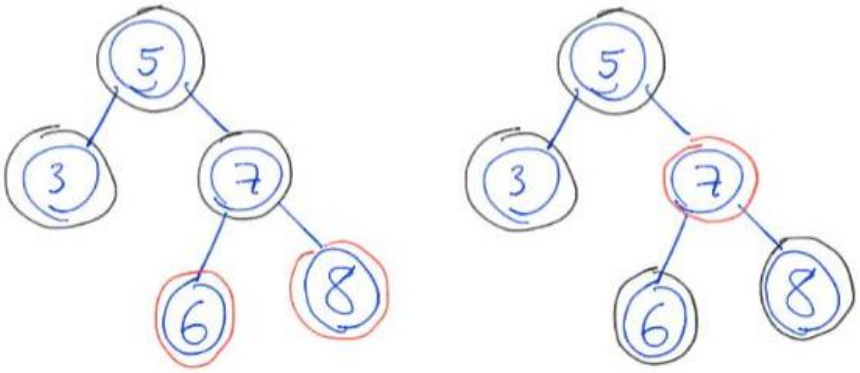
Para ganhar intuição de que essas propriedades levam a uma árvore balanceada

- note que uma lista com três nós já não pode ser uma árvore rubro-negra.




Exemplos:






A seguir o código para a estrutura de um nó de árvore rubro-negra:

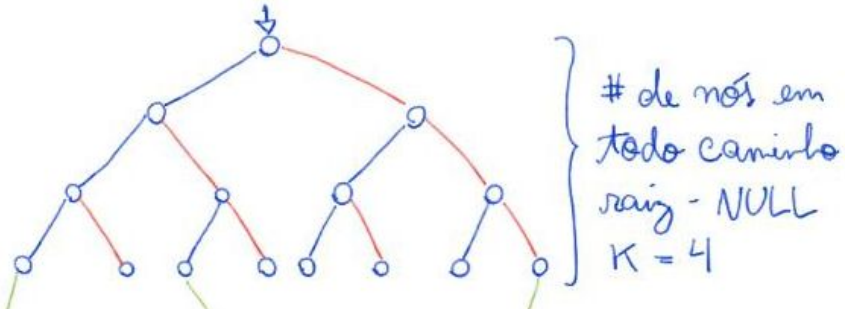
```
typedef int Item;
typedef int Chave;

typedef struct noh
{
   int vermelho;
   Chave chave;
   Item conteudo;
   struct noh *pai;
   struct noh *esq;
   struct noh *dir;
} Noh;

typedef Noh *Arvore;
```

**Altura máxima de árvores rubro-negras**

Vamos demonstrar que, toda árvore rubro-negra tem altura <= 2 lg (n+1).

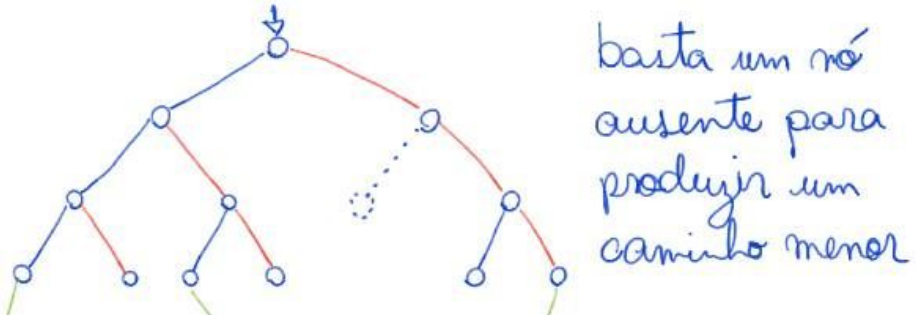
Observe que, se todo caminho raiz-NULL de uma árvore tem pelo menos k nós,

- então os primeiros k níveis da árvore devem estar completos.



Caso contrário, haveria um caminho da raiz até o nó ausente,

- resultando em um caminho raiz-NULL de comprimento menor que k - 1,
    - lembrando que comprimento de um caminho,
        - é um a menos que o número de nós deste.




Como os primeiros k níveis da árvore estão completos,

- o número de nós n desta árvore é maior ou igual que
    - o número de nós numa árvore binária completa de altura k - 1,
        - ou seja, n >= 2^k - 1.
- Portanto, k <= lg (n + 1).

Note que, numa árvore rubro-negra, pela propriedade 4 da definição,

- todo caminho raiz-NULL tem um mesmo número de nós pretos.
    - Digamos que este número é k.

Portanto, temos um limitante superior para

- o número de nós pretos em qualquer caminho raiz-NULL,
    - i.e., k <= lg (n + 1).
- No entanto, queremos um limitante para
    - o número total de nós em qualquer caminho,
        - já que isso limita a altura da árvore.

Pelas propriedades 1 e 3 da definição, todo caminho tem no máximo

- um nó vermelho para cada nó preto,
    - já que os caminhos começam na raiz,
        - que é preta,
    - e não são permitidos dois nós vermelhos consecutivos.

Assim, o número total de nós em qualquer caminho raiz-NULL da árvore é

- <= 2 * k  <= 2 lg (n + 1).

Como a altura da árvore é igual ao comprimento do maior caminho raiz-NULL,

- e o comprimento de um caminho é menor que o seu número de nós,
    - o resultado segue.

**Inserção em árvores rubro-negras**

A ideia geral é

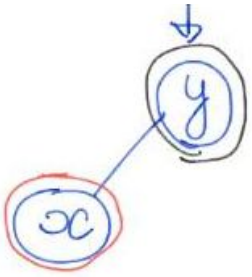
- inserir o novo nó como uma folha vermelha,
    - após percorrer um caminho descendente na árvore,
- e então usar recoloração e rotações,
    - ao percorrer o caminho no sentido ascendente,
        - para restabelecer as propriedades da definição.

A seguir vamos analisar alguns casos

- que podem ocorrer na volta da recursão,
    - quando percorremos o caminho ascendente,
- sempre considerando que o nó corrente é x.

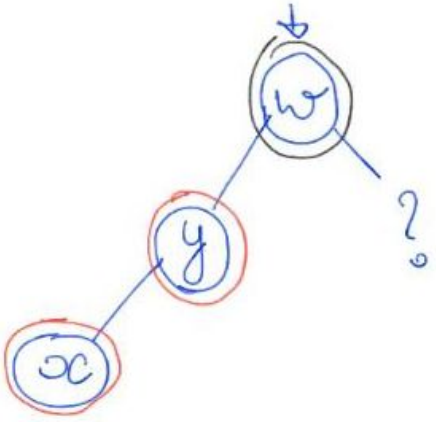
Caso 1: se y, o pai de x, não for vermelho,

- as propriedades da definição estão mantidas
    - e nada precisa ser feito.



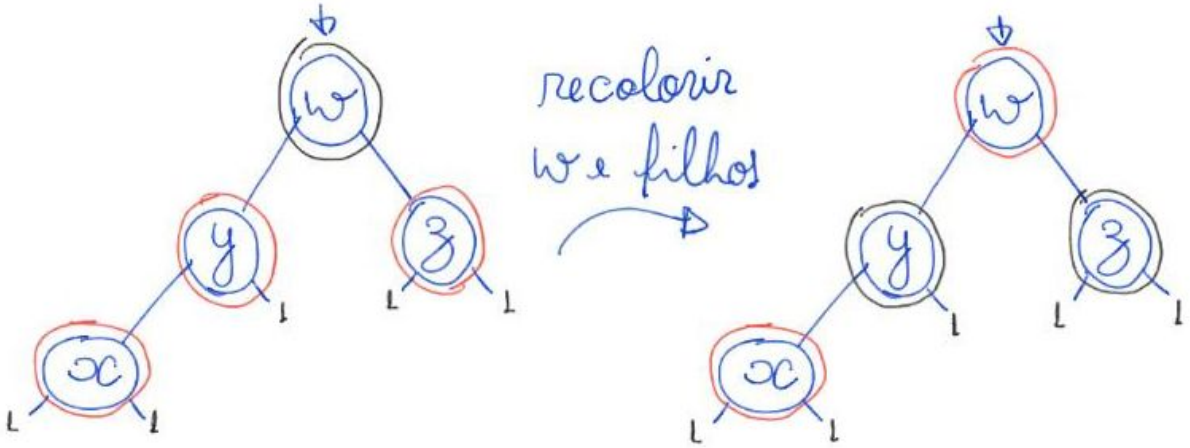

Caso 2: se y é vermelho,

- temos que restaurar a propriedade 3,
    - e sabemos que y não é a raiz,
        - por conta da propriedade 2,
    - e que w, o pai de y, é preto,
        - por conta da propriedade 3.

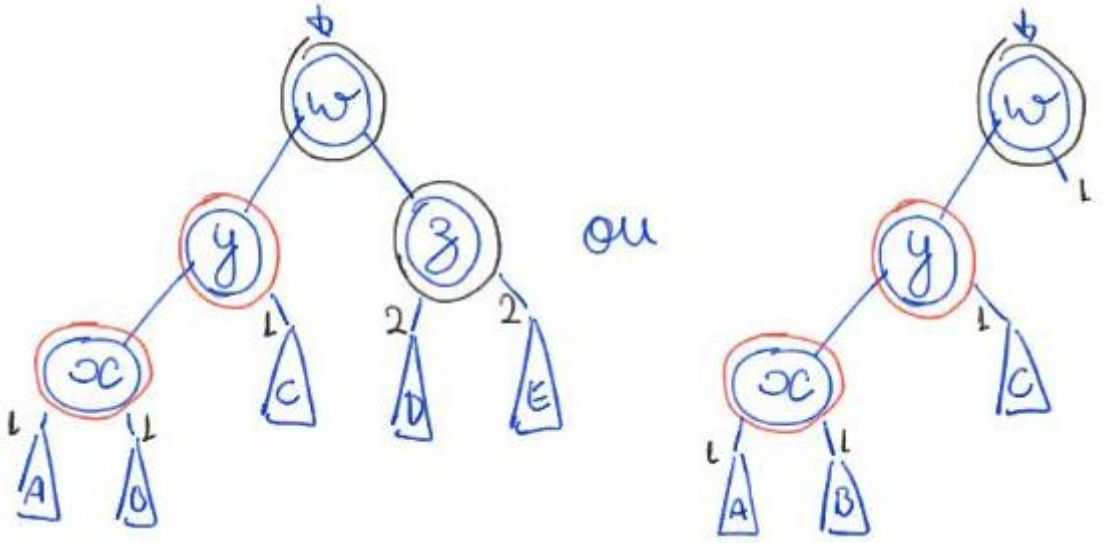



- Caso 2.1: w tem outro filho z que tem cor vermelha.

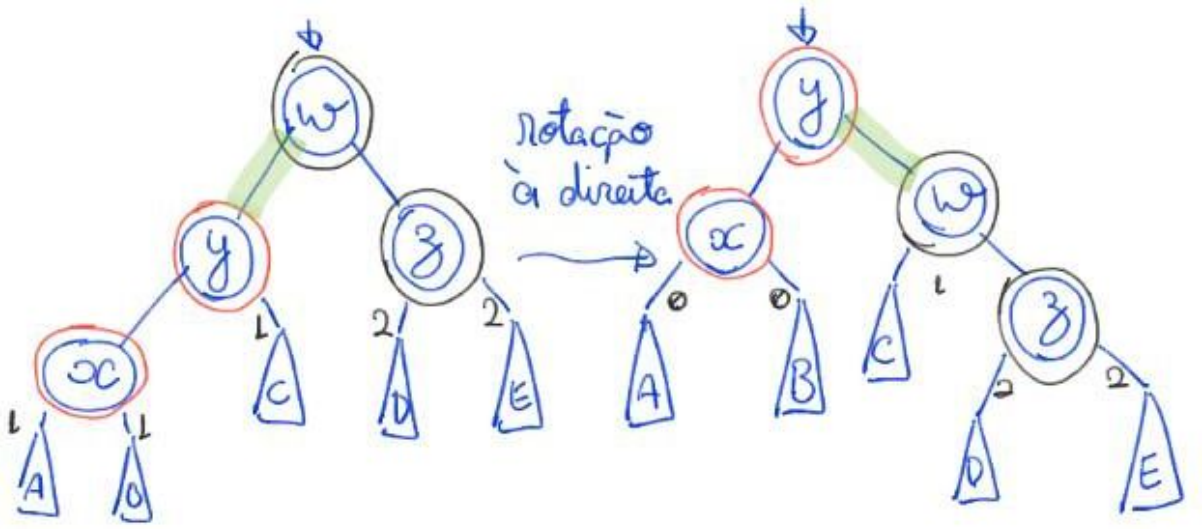
- Neste caso vamos recolorir z e y para preto e w para vermelho.



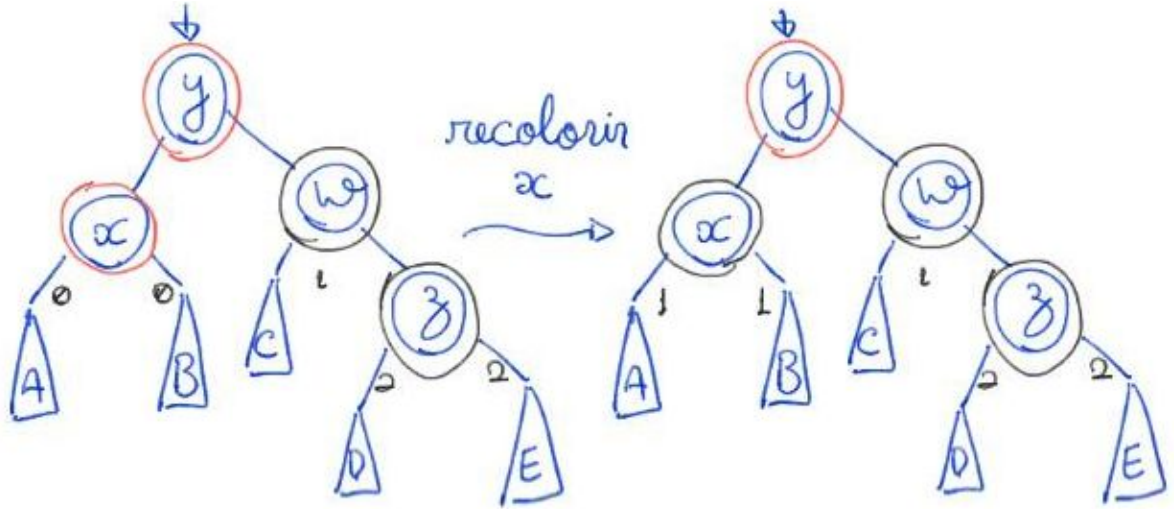

  - Note que, dessa forma preservamos a propriedade 4,
    - pois todo caminho que passa por w passa por y ou por z,
    - e resolvemos a propriedade 3 entre x e y.
  - Ao continuar a subida na árvore, será necessário
    - analisar a situação de w, que ficou vermelho, pois
      - ele pode violar a propriedade 3 com relação a seu pai.
  - Se w for a raiz da árvore, basta mudar a cor dele para preto.
- Caso 2.2: w tem não tem outro filho vermelho.
  - Neste caso, w pode ter outro filho preto ou não ter outro filho.



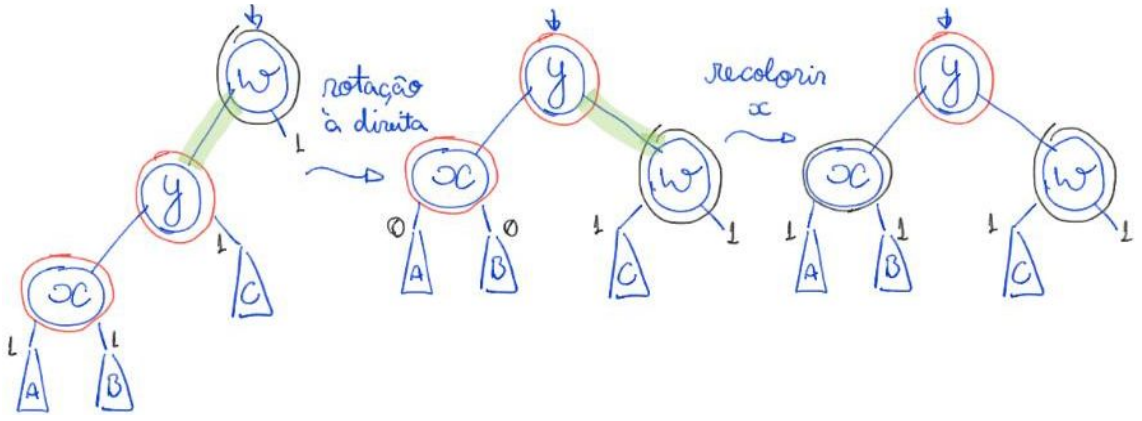

    - Primeiro, vamos considerar a presença de um nó preto z.
  - Fazemos uma rotação à direita a partir de w,
    - que inverte a relação entre w e y.
    - Note que, esta rotação reduz o número de nós pretos
      - nos caminhos que vão da raiz até os filhos de x.




- Então mudamos a cor de x para preto,
  - resolvendo tanto o problema dos vermelhos adjacentes
    - quanto do número de pretos nos caminhos raiz-NULL.




- Ao continuar a subida na árvore, será necessário analisar
  - a situação de y, que se tornou a raiz da subárvore e é vermelho,
    - pois ele pode violar propriedade 3 com relação a seu pai.
  - Se y for a raiz da árvore, basta mudar a cor dele para preto.
- Note que, se w não tiver outro filho, a mesma solução funciona.



Embora nossos tratamentos dos casos anteriores estejam corretos,

- uma vez que eles restauram as propriedades da árvore rubro-negra,
    - sua implementação é um tanto complicada, por ser necessário
        - manipular antecessores do nó corrente.
- Por isso, vamos estudar a implementação da inserção
    - em um tipo especial de árvore rubro-negra.

**Inserção em árvores rubro-negras esquerdistas**

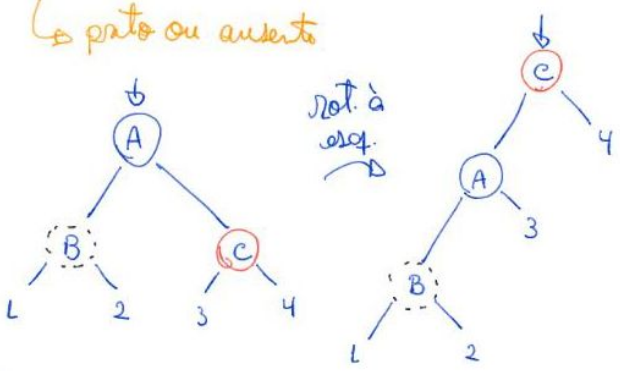
A seguir apresentamos a implementação recursiva utilizada

- na versão da árvore rubro-negra introduzida por Sedgewick,
    - que é conhecida como árvore rubro-negra esquerdista,
        - pois os nós vermelhos sempre são filhos esquerdos.

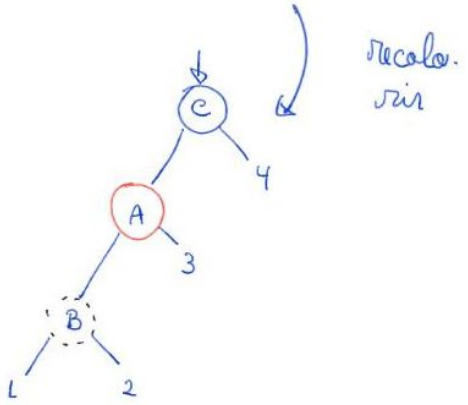
```
Noh *novoNoh(Chave chave, Item conteudo)
{
   Noh *novo;
   novo = (Noh *)malloc(sizeof(Noh));
   novo->vermelho = 1;
   novo->chave = chave;
   novo->conteudo = conteudo;
   novo->esq = NULL;
   novo->dir = NULL;
   //    novo->pai = ??
   return novo;
}

Arvore insereRN(Noh *r, Noh *novo)
{
   if (r == NULL) // subárvore era vazia
   {
       novo->pai = NULL;
       return novo;
   }
   if (novo->chave <= r->chave) // desce à esquerda
   {
       r->esq = insereRN(r->esq, novo);
```

```
        r->esq->pai = r;
    }
    else // desce à direita
    {
        r->dir = insereRN(r->dir, novo);
        r->dir->pai = r;
    }
```

- se f.d. é vermelho e f.e. não é vermelho
  - preto ou ausente
  - rotação à esq.

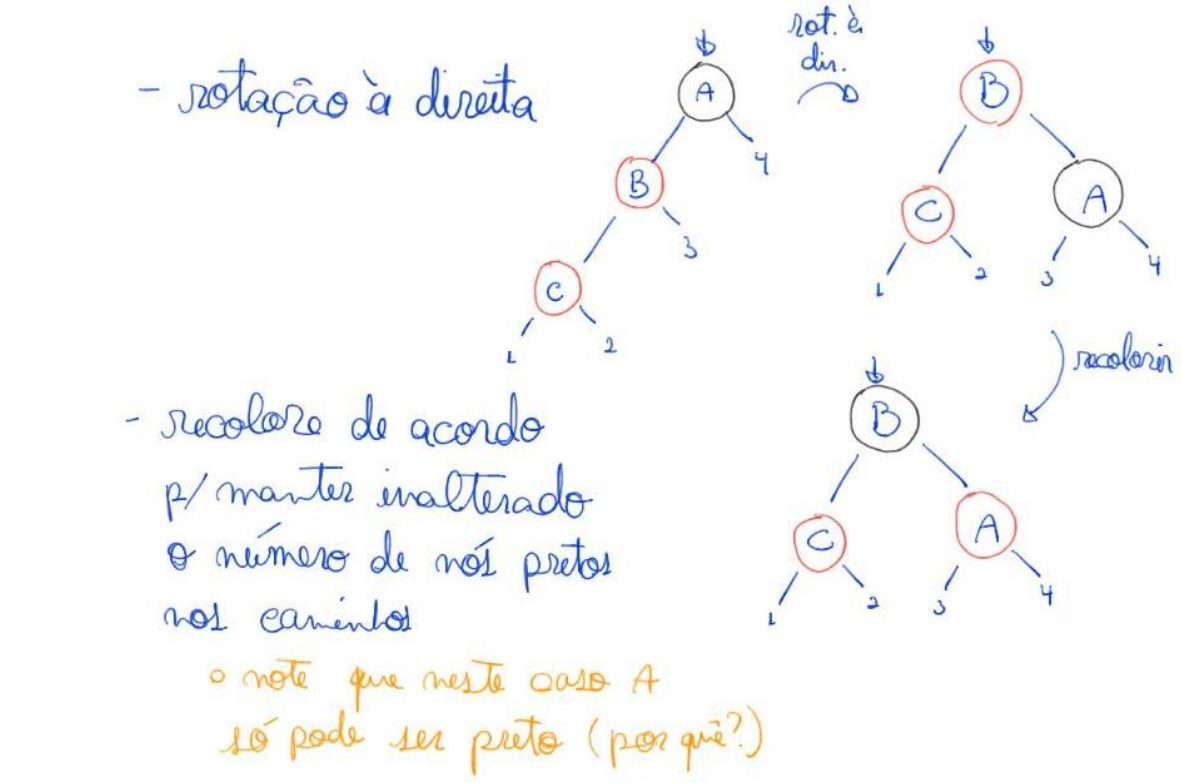



  - recolore de acordo p/ manter inalterado o número de nós pretos nos caminhos
    - note que A pode ter cor preta ou vermelha e que sua cor é herdada por C

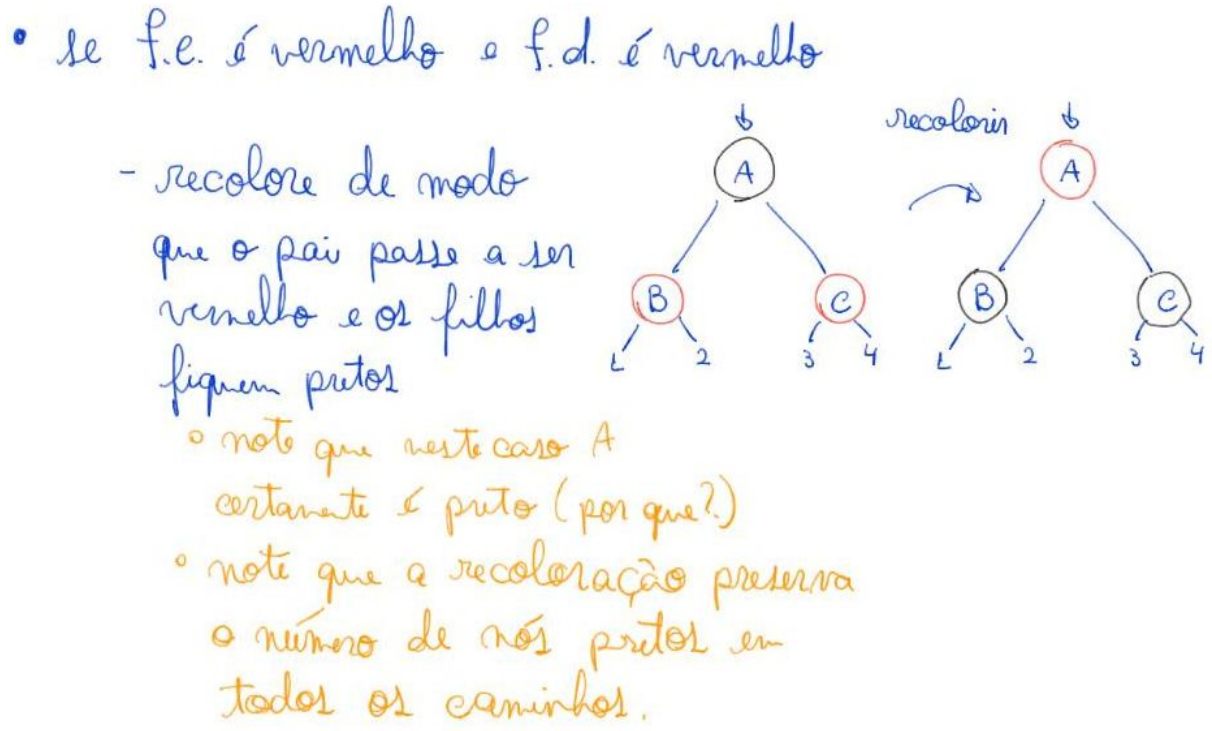



```
    if (r->dir != NULL && r->dir->vermelho == 1 && (r->esq == NULL ||
r->esq->vermelho == 0))
    {
        r = rotacaoEsq(r);
        r->vermelho = r->esq->vermelho;
        r->esq->vermelho = 1;
    }
```

- se f.e. é vermelho e f.e. do f.e. é vermelho




```
   if (r->esq != NULL && r->esq->vermelho == 1 && r->esq->esq != NULL && r->esq->esq->vermelho == 1)
   {
       // se filho e neto esquerdos forem vermelhos faz rotação a direita e recolore de acordo
       r = rotacaoDir(r);
       r->vermelho = 0;
       r->dir->vermelho = 1;
   }
```




```
   if (r->esq != NULL && r->esq->vermelho == 1 && r->dir != NULL &&
r->dir->vermelho == 1)
   {
       // se os dois filhos são vermelhos, troque a cor com o pai
preto
       r->esq->vermelho = 0;
       r->dir->vermelho = 0;
       r->vermelho = 1;
   }
   return r;
}

Arvore inserir(Arvore r, Chave chave, Item conteudo)
{
   Noh *novo = novoNoh(chave, conteudo);
   return insereRN(r, novo);
}
```

É interessante observar como as operações anteriores

- garantem a propriedade invariante da árvore rubro-negra esquerdista,
    - i.e., os nós vermelhos sempre são filhos esquerdos.