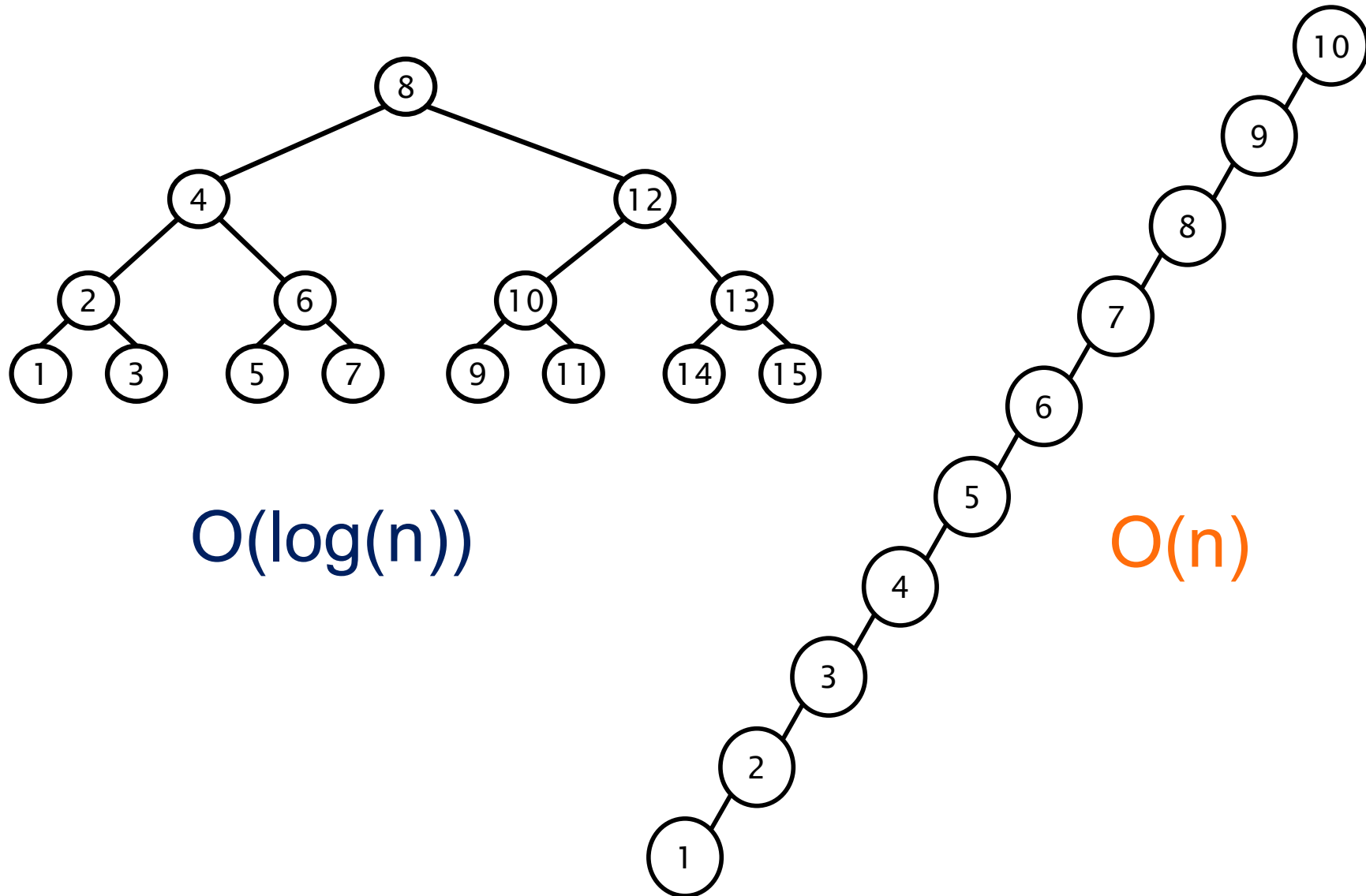
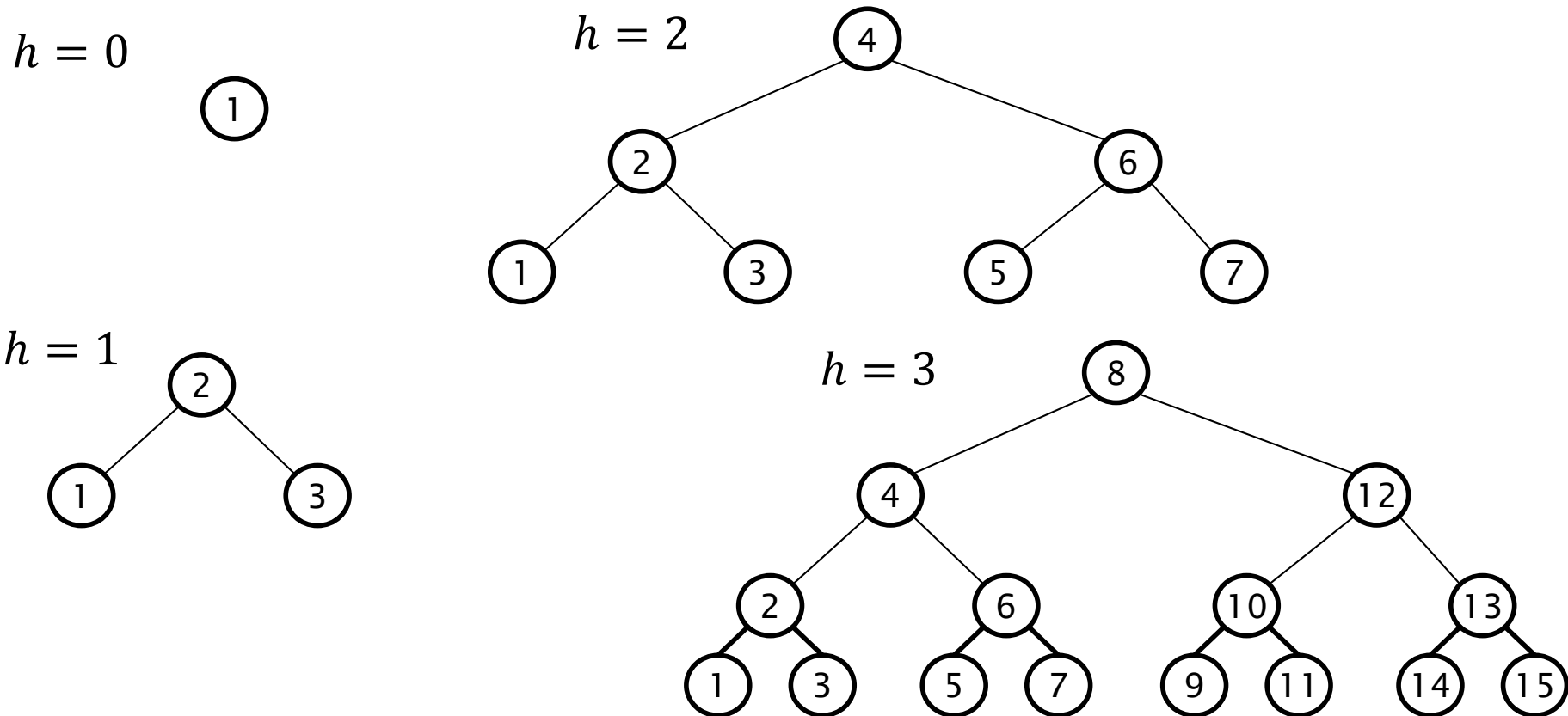


Árvores Binárias de Busca Balanceadas



Número mínimo de nós numa árvore **cheia** de altura h



$$n_c(h) = n_c(h-1) + 2^h$$

$$n_c(h) = 1 + 2^1 + 2^2 + 2^3 + \dots + 2^h = 2^{h+1} - 1$$

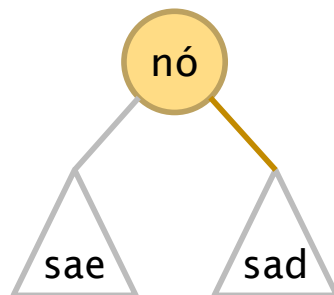
Balanceamento de Árvores Binárias de Busca

motivação:

fazer busca em $O(\log(n))$

objetivo:

- diminuir a altura da árvore mantendo o número de nós.
- diminuir a diferença de altura entre a sub-árvore à esquerda (h_e) e a sub-árvore à direita (h_d)



$$|h_d - h_e| = \text{pequeno}$$

Árvores Rubronegras

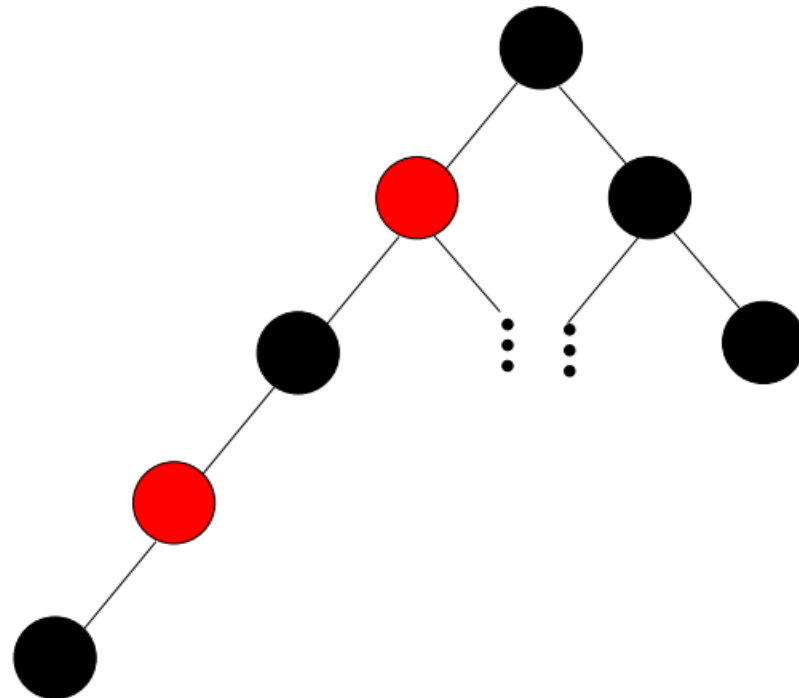
São árvores auto-ajustáveis, que procuram manter o balanceamento de forma automática mediante as seguintes restrições:

1. Todo nó é **vermelho** ou **preto**
2. A raiz é **preta**
3. As folhas são os nós NULL e são **pretas**
4. Nós **vermelhos** só tem **filhos pretos**
5. Todos os caminhos a partir da raiz da árvore até suas folhas passa pelo mesmo número de nós **pretos**

Árvores Rubronegras

Estas restrições garantem que o maior caminho entre a raiz e as folhas é no máximo o dobro do menor caminho.

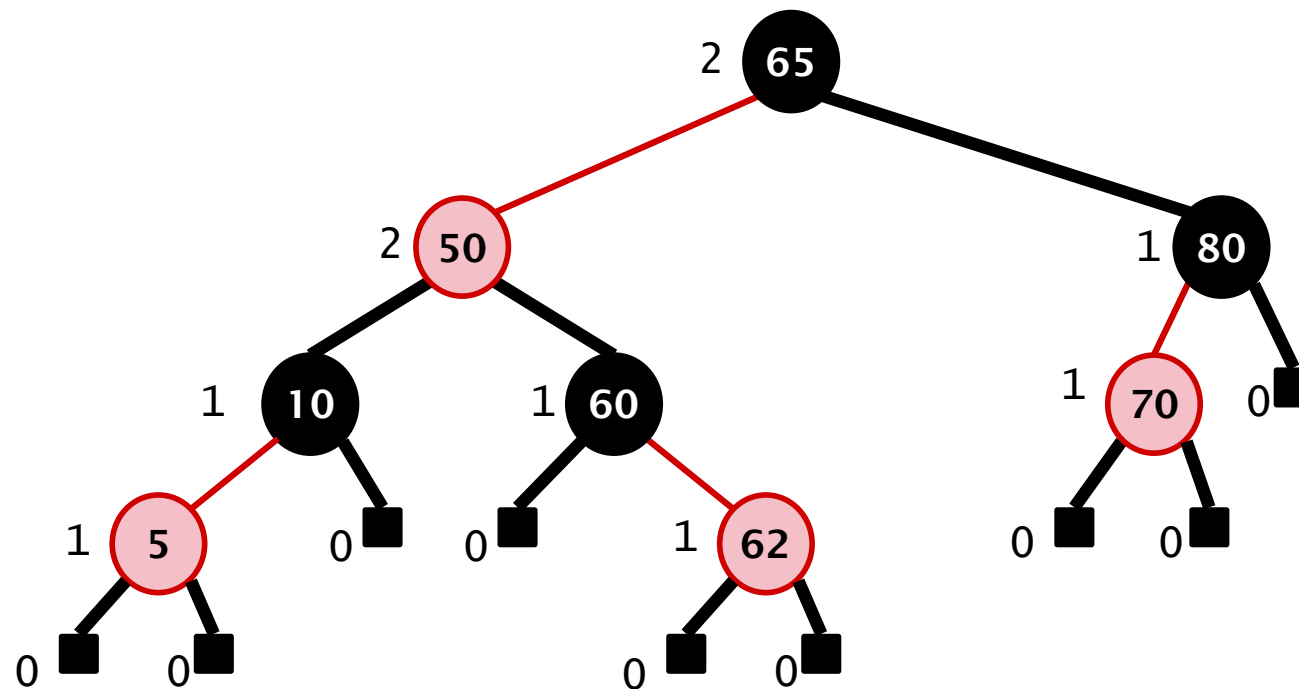
Elas garantem um balanceamento razoável



Árvore Red-Black - definição

Altura negra (rank) de um nó:

número de **arestas pretas** de qualquer caminho desde o nó até um nó externo



Árvore Red-Black - conceitos

$r = \textit{rank}$ da raiz (2)

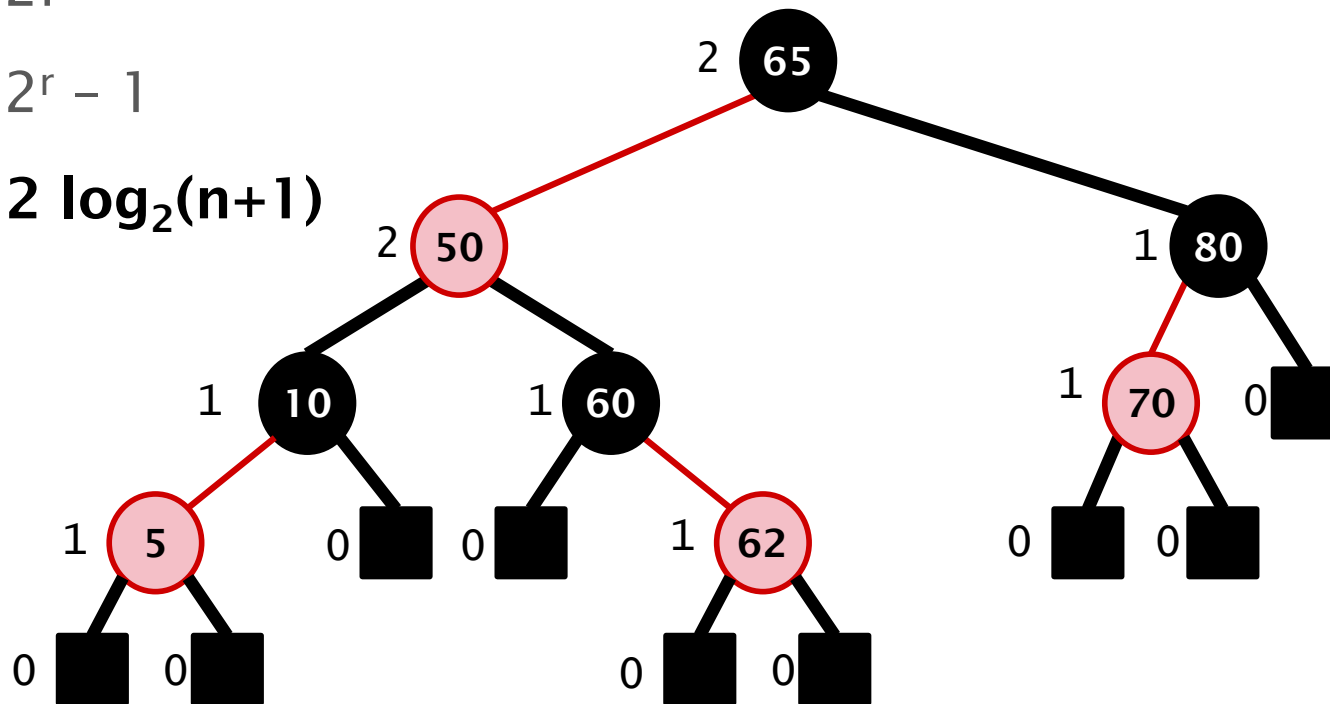
$h =$ altura da árvore sem nós externos ($3 = 2r-1$)

$n =$ número de nós (8)

$$h \leq 2r$$

$$n \geq 2^r - 1$$

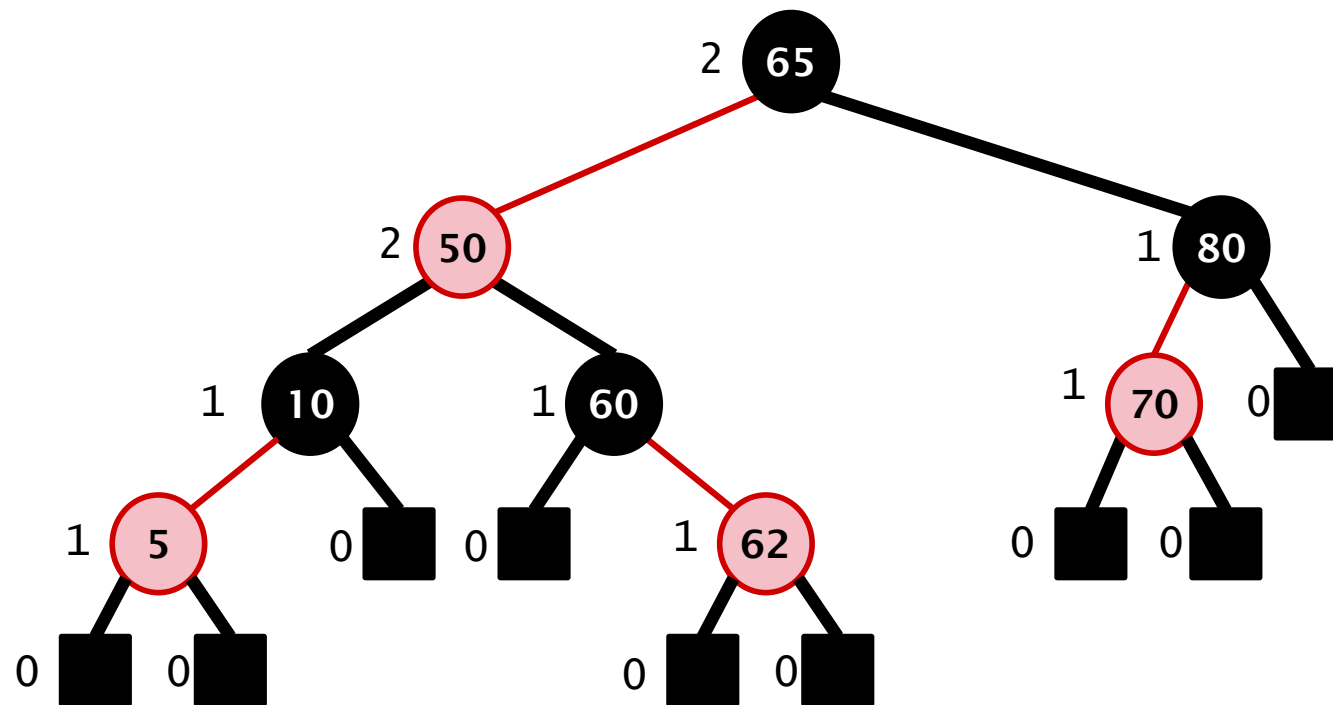
$$h \leq 2 \log_2(n+1)$$



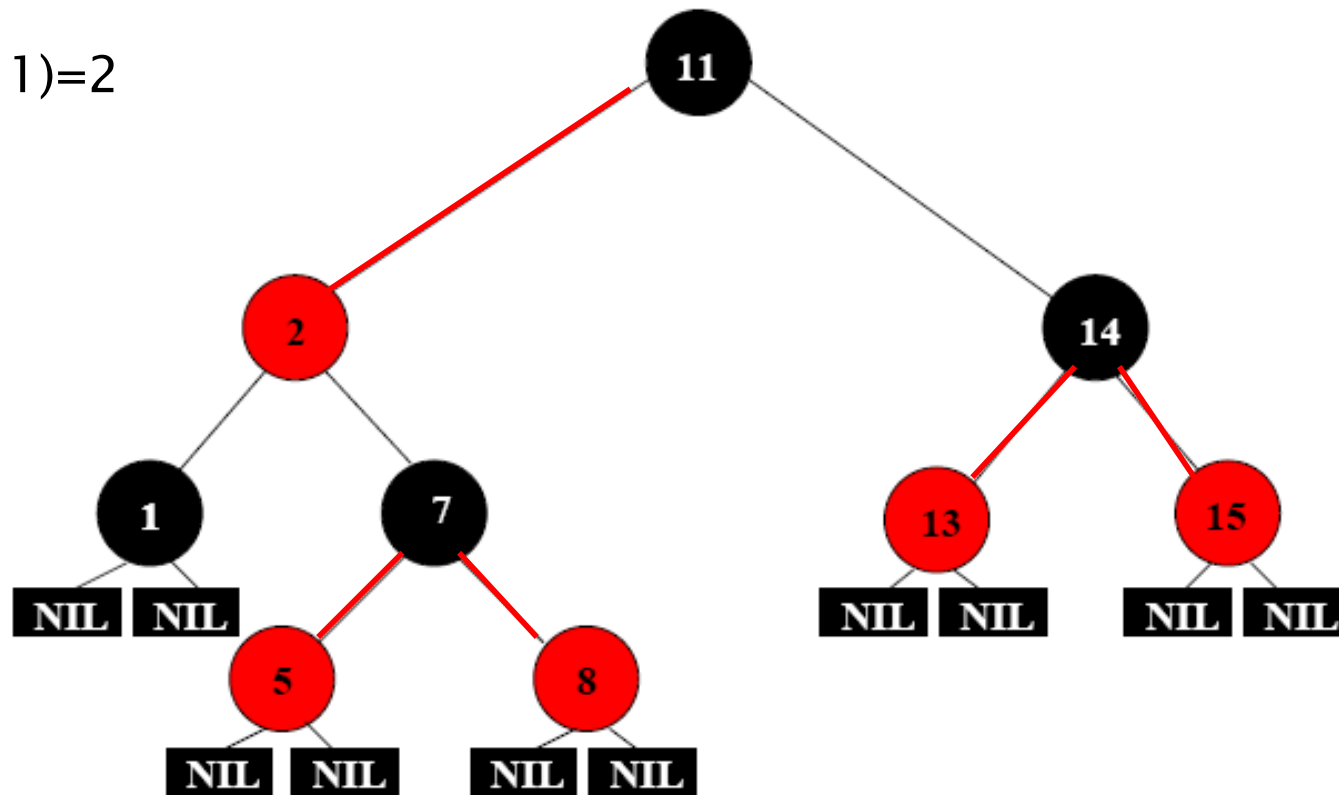
Árvore Red-Black - conceitos

Sejam P e Q dois caminhos da raiz até nós externos:

$$\text{comprimento}(P) \leq 2 \times \text{comprimento}(Q)$$



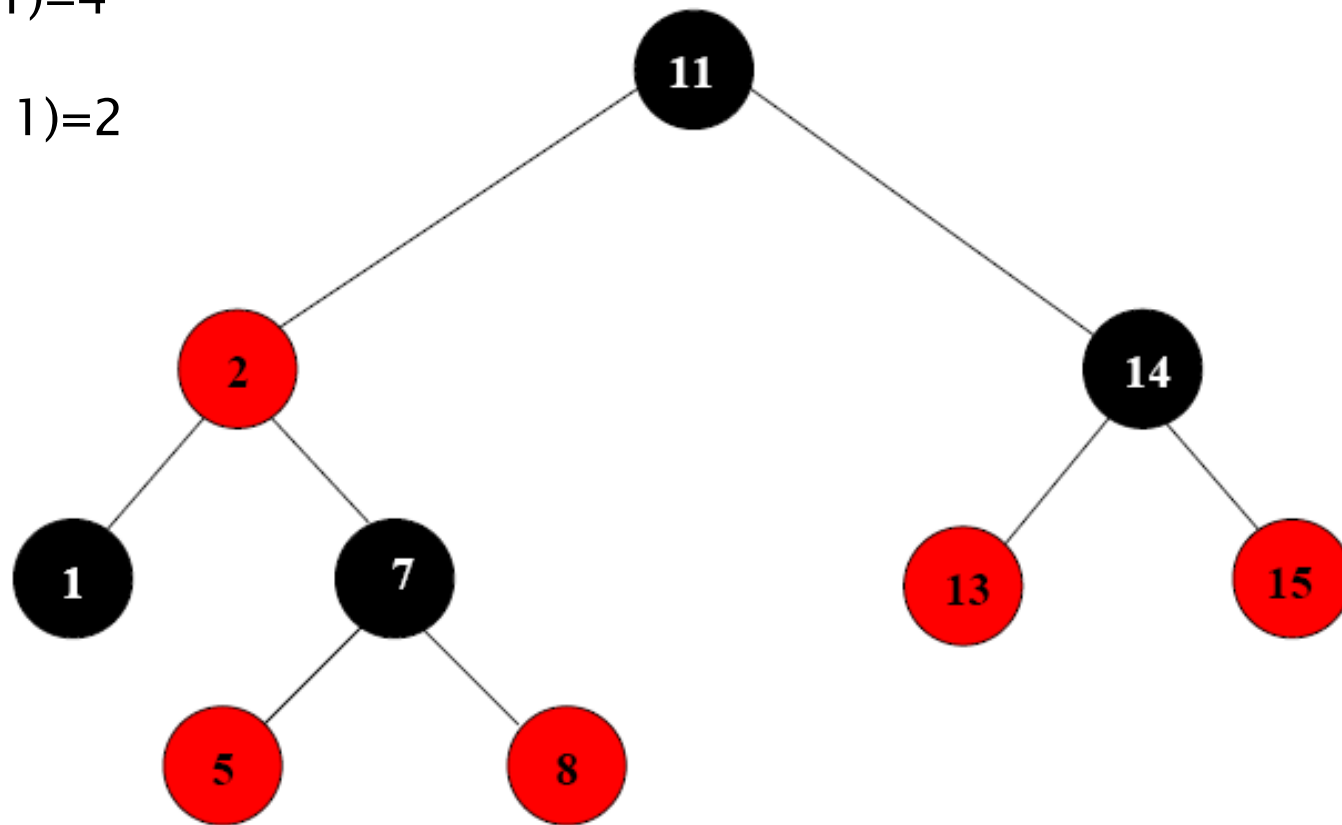
Formas de representação

$$h(1,1)=4$$
$$\text{bh}(1\ 1)=2$$


Formas de representação

$h(11)=4$

$bh(11)=2$

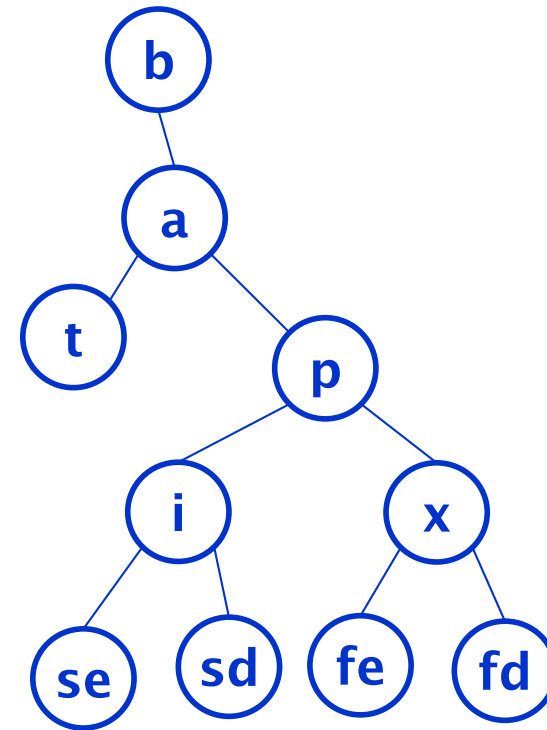


Árvore Red-Black - busca

igual à busca numa árvore binária de busca

Notação

```
x // nó corrente  
p = x->pai // pai  
a = p->pai // avô  
b = a->pai // bisavô  
t = a->esq ou a->dir // tio  
fe = x->esq // filho à esquerda  
fd = x->dir // filho à direita  
i = p->esq ou p->dir // irmão  
se = i->esq // sobrinho a esquerda  
sd = i->dir //sobrinho a direita
```



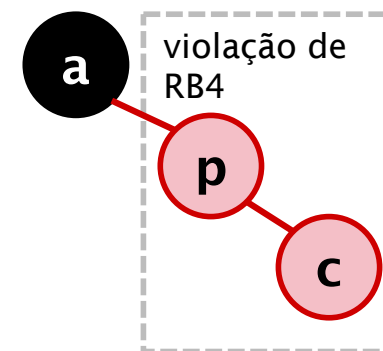
Inserção

Ideia geral da inserção é seguir o processo de uma ABB colorindo o novo nó **vermelho**, assim se o pai for **preto** a inserção não viola as regras da Red Black.

Possíveis violações:

[RB2] a raiz é preta

[RB4] **p** é vermelho



Vamos “consertar” a árvore de baixo para cima.

Correção da inserção

Duas maneiras de corrigir a violação das regras RB: re-colorir os nós envolvidos e/ou fazer rotações dos nós.

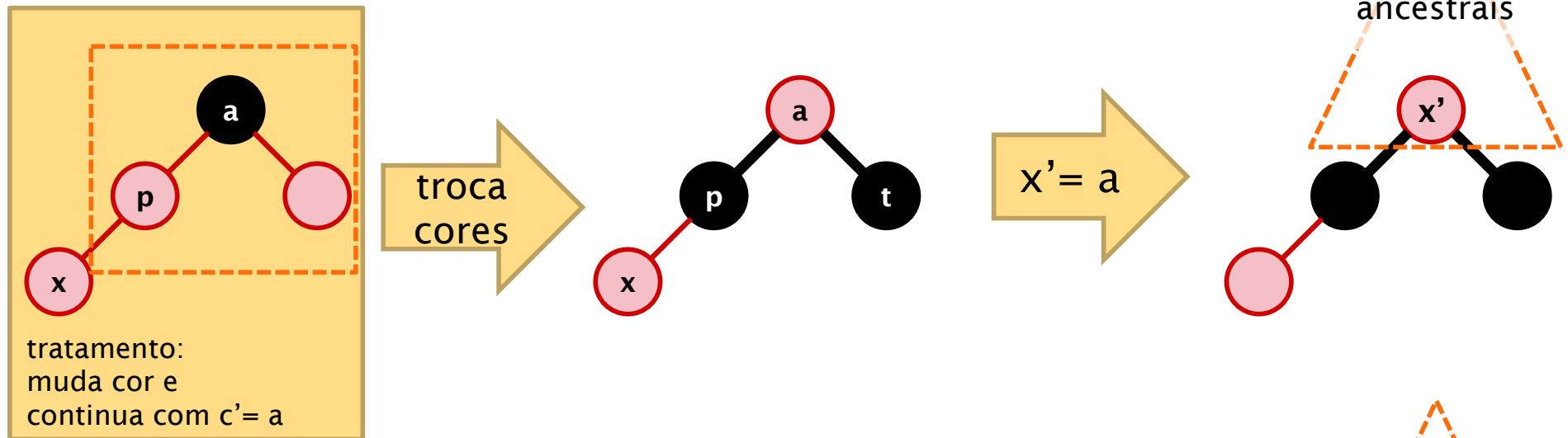
- **Correção 1:** Se o nó vermelho for a **raiz**, simplesmente troque a cor. A **raiz** vira preta e altura preta da árvore aumenta de uma unidade. [recolorir]

Correção 2: tio vermelho

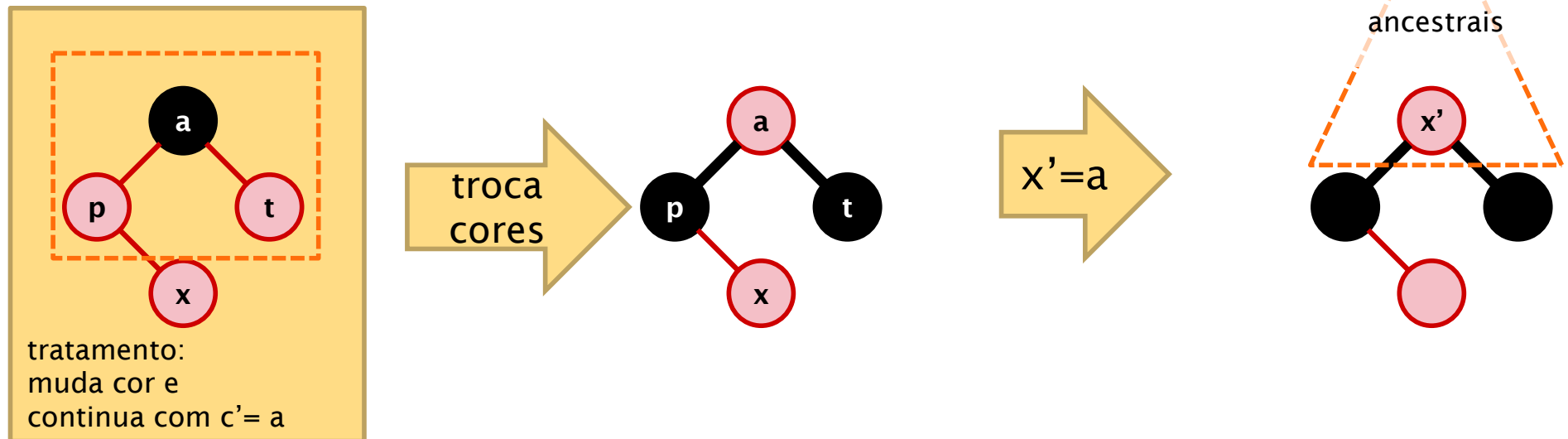
Se o pai e o tio forem **vermelhos**, troque a cor de ambos para **preto**. Como isso aumenta a altura negra da sub-árvore em questão, isso pode violar a RB4, por isso mude a cor do **avo** para vermelho. Continue a correção a partir do avo. A correção vai de baixo para cima.

notação XY: p é filho $X \{L,R\}$ de a
 x é filho $Y \{L,R\}$ de p

Tratamento por mudança de cor



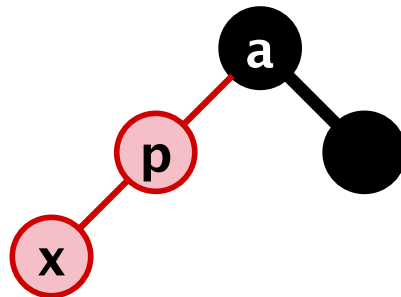
Note que, neste caso, não importa se c é um filho a esquerda ou a direita



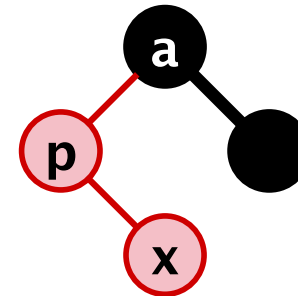
Correção 3: tio preto

Se o tio for preto, precisamos saber se **x** e **p** são filhos à direita ou à esquerda. Cada um dos quatro sub-casos tem um tratamento diferente.

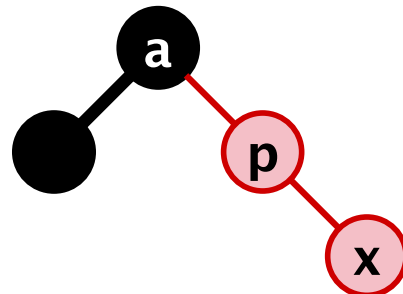
LL



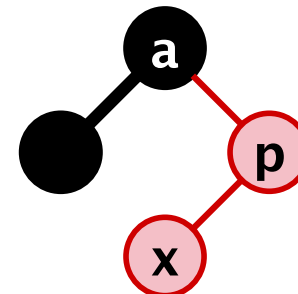
LR



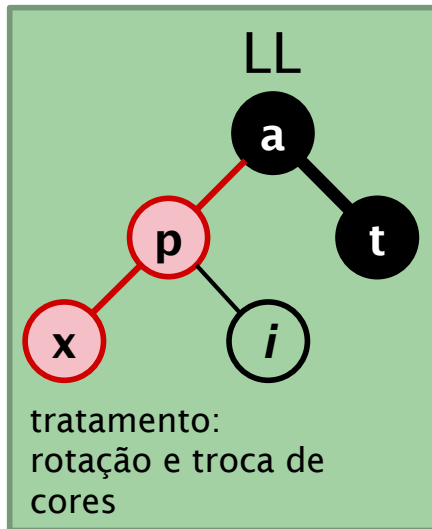
RR



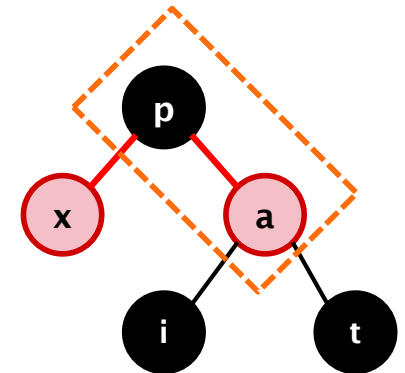
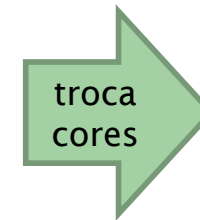
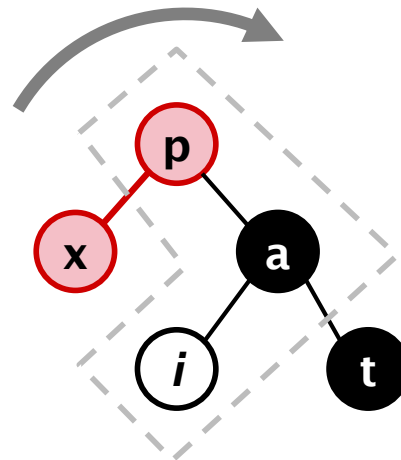
RL



Tratamento LL rotação e troca de cor



```
p = x->pai;
a = p->pai;
b = a->pai;
i = p->dir;
```



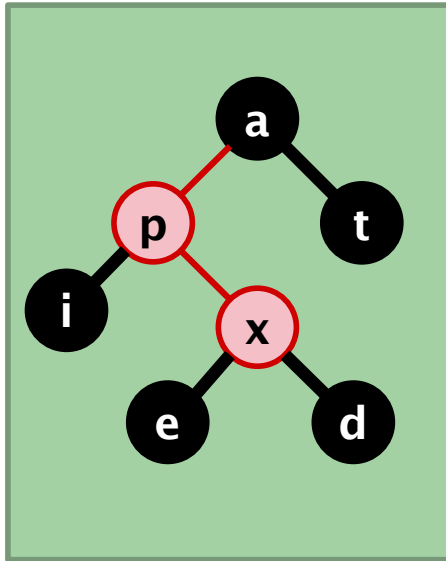
```
p->cor = BLACK;
a->cor = RED;
```

```
p->pai = b;
if (b!=nil) {
    if (a==b->esq)
        b->esq = p;
    else
        b->dir = p;
}
```

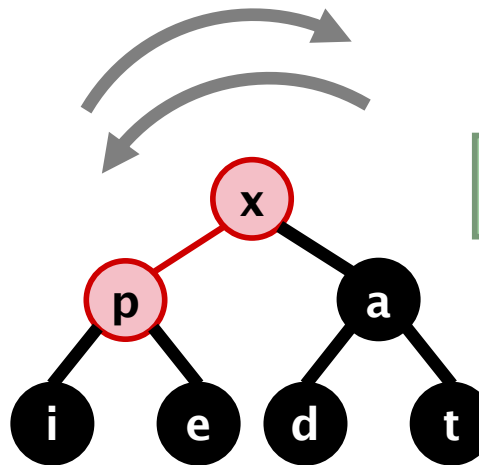
```
p->dir = a;  a->pai = p;
a->esq = i;  i->pai = a;
```

Tratamento LR rotação dupla e troca de cor

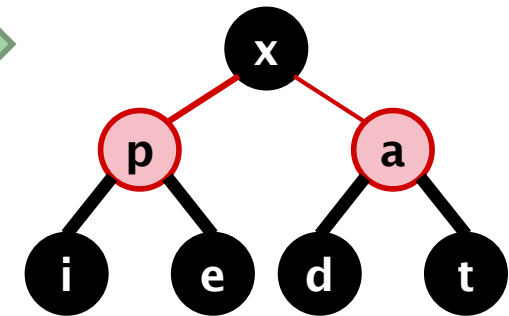
LR



```
p = x->pai;
a = p->pai;
b = a->pai;
i = p->esq;
t = a->dir;
e = x->esq;
d = x->dir;
```



troca
cores



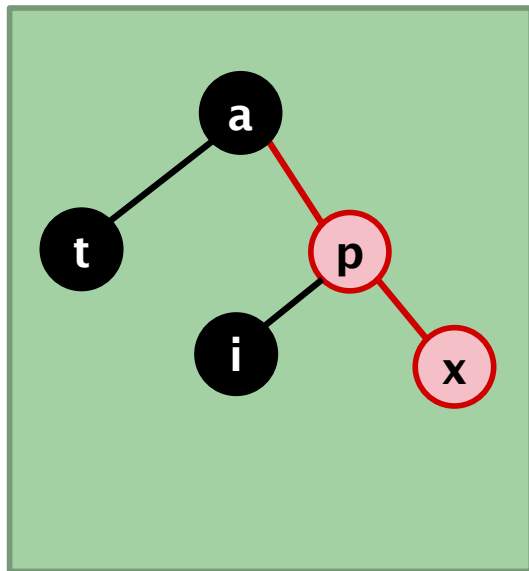
```
c->pai = b;
if (b!=nil) {
    if (a==b->esq)
        b->esq = p;
    else
        b->dir = p;
}
```

```
c->esq = p; p->pai = c;
c->dir = a; a->pai = c;
p->dir = e; e->pai = p;
a->esq = d; d->pai = a;
```

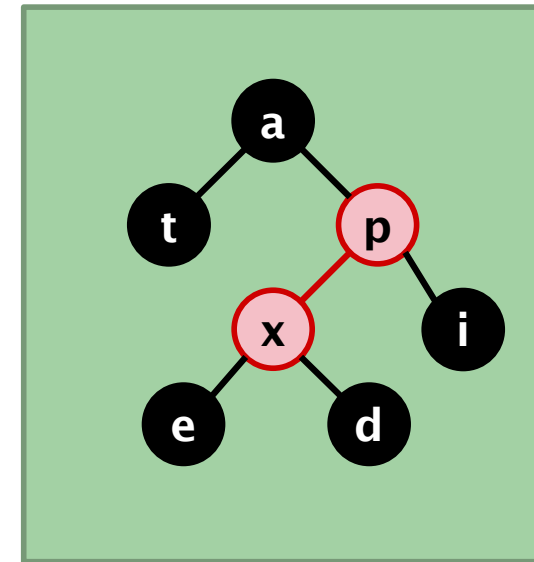
```
c->cor = BLACK;
a->cor = RED;
```

Tratamento LR rotação dupla e troca de cor

RR

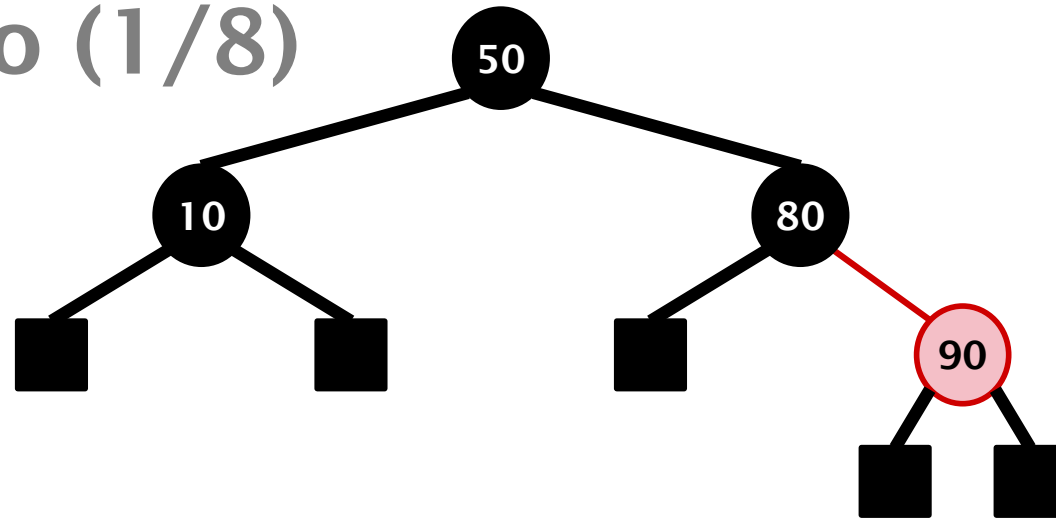


RL

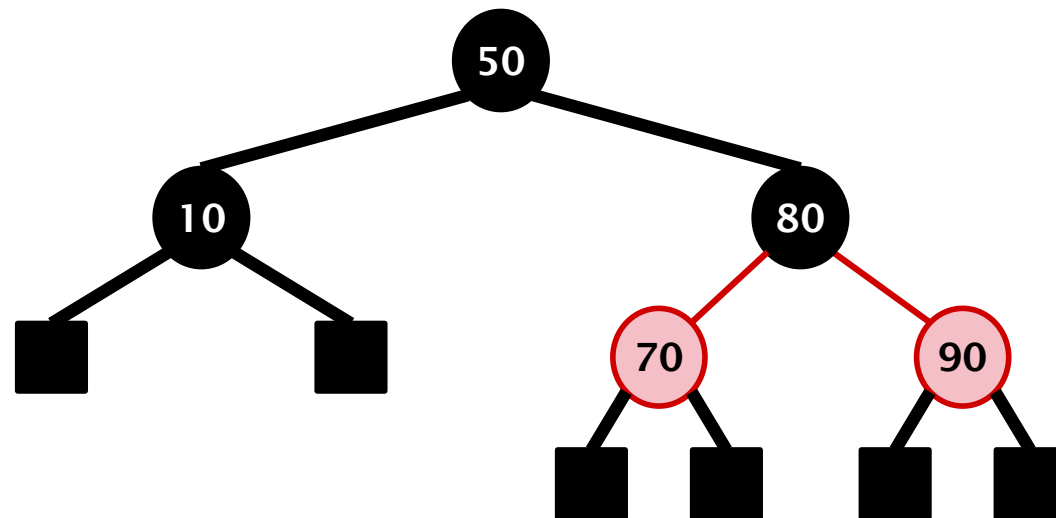


Exercício: Faça os casos simétricos

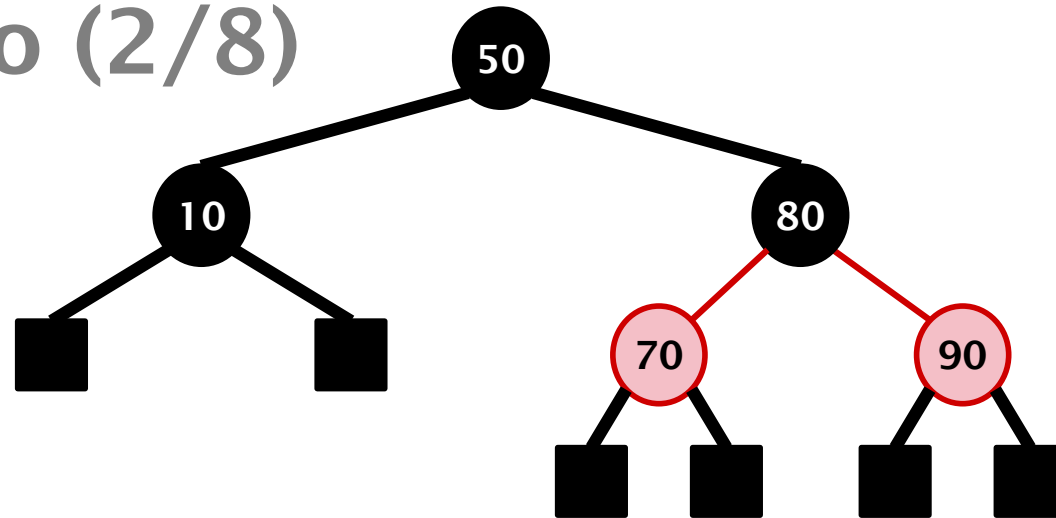
Exemplo (1/8)



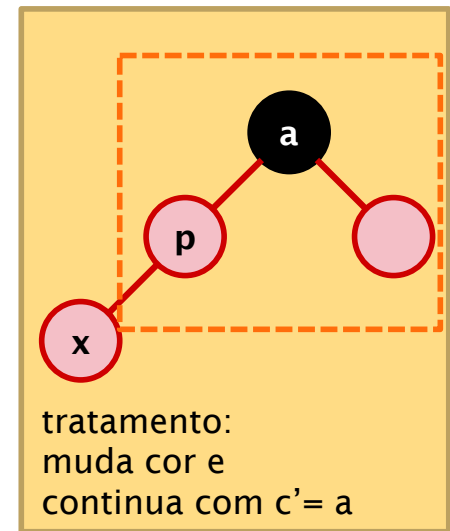
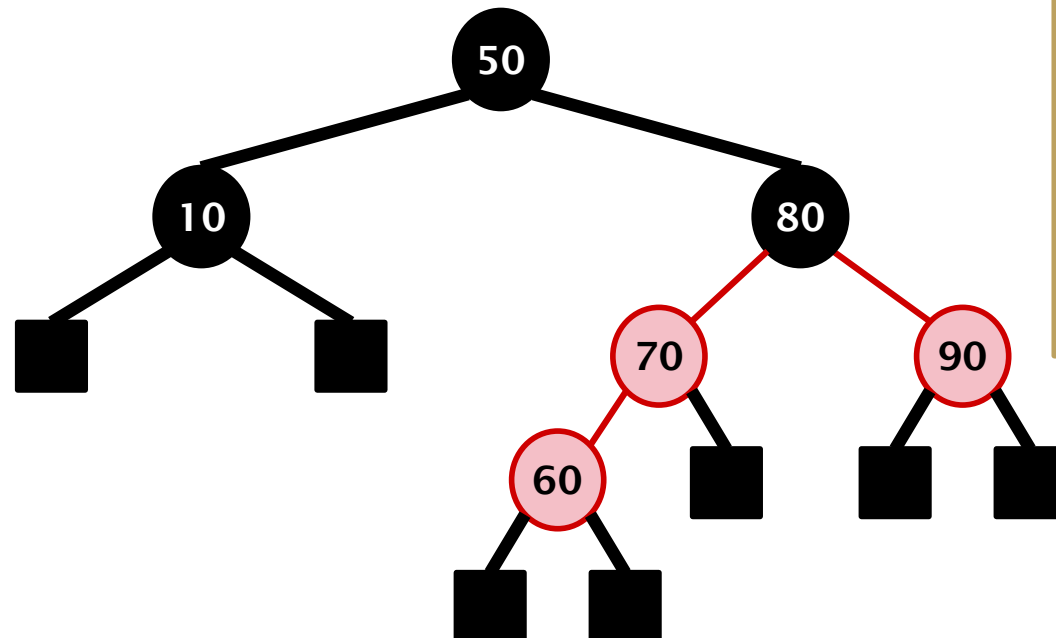
insere 70



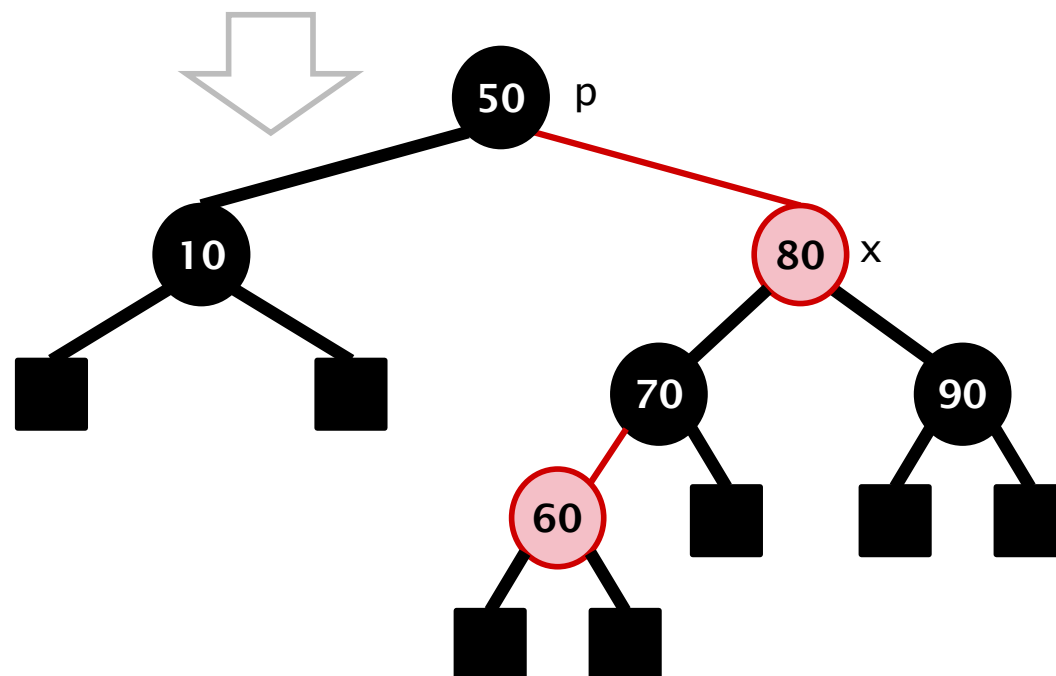
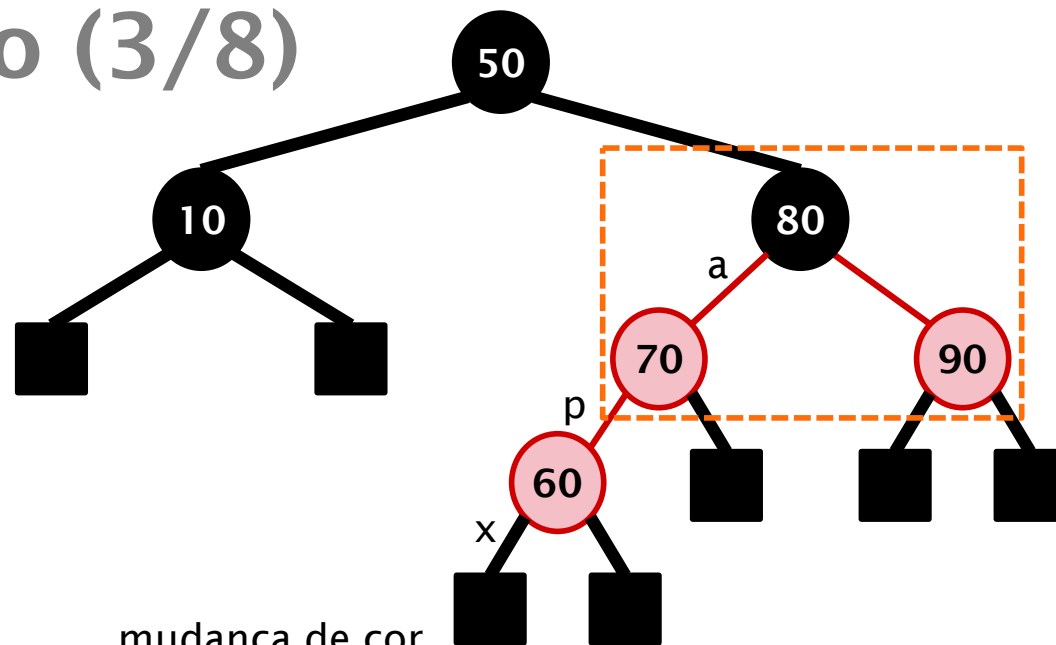
Exemplo (2/8)



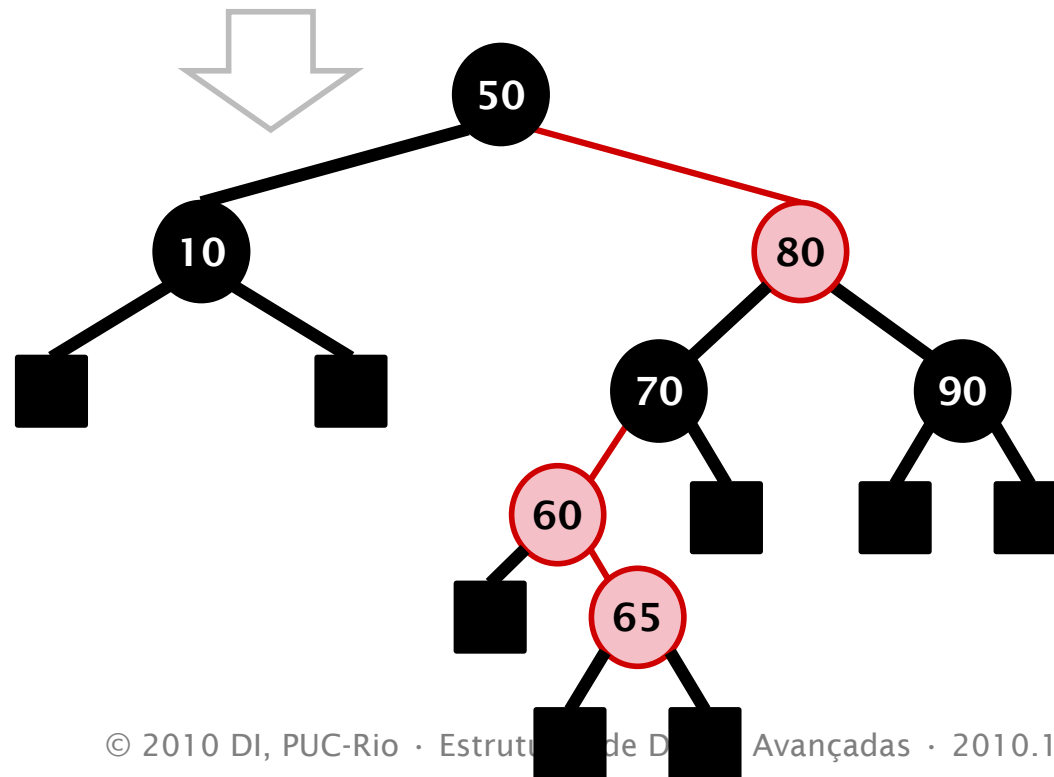
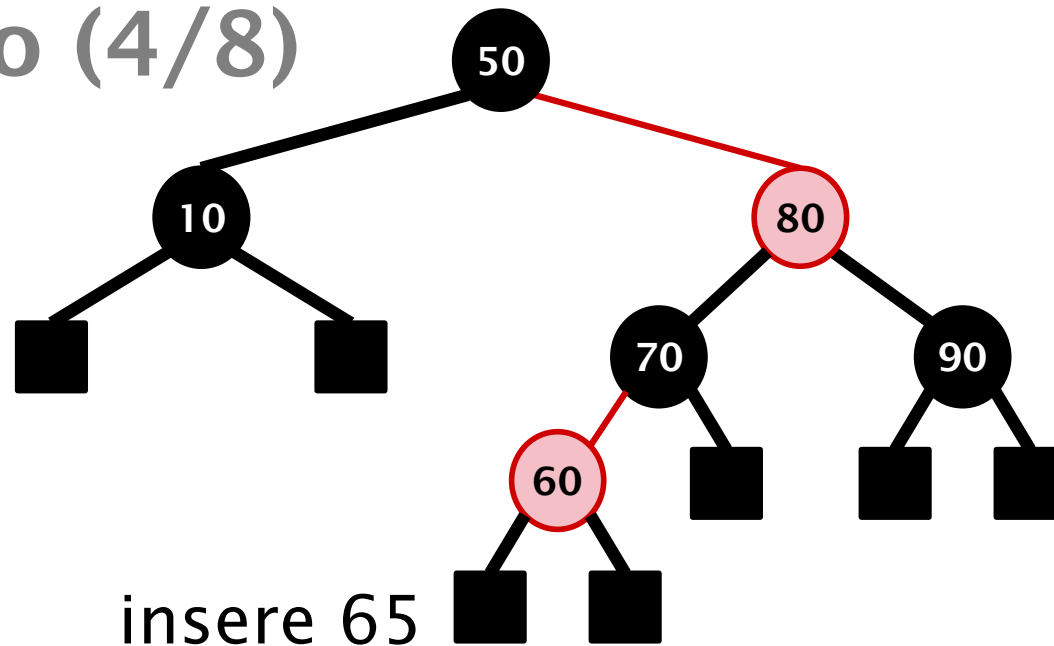
insere 60



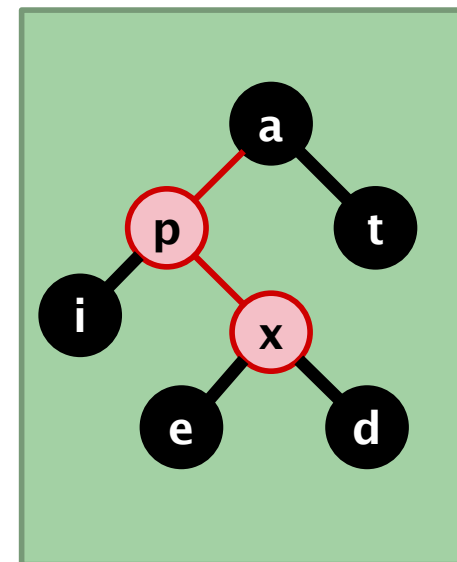
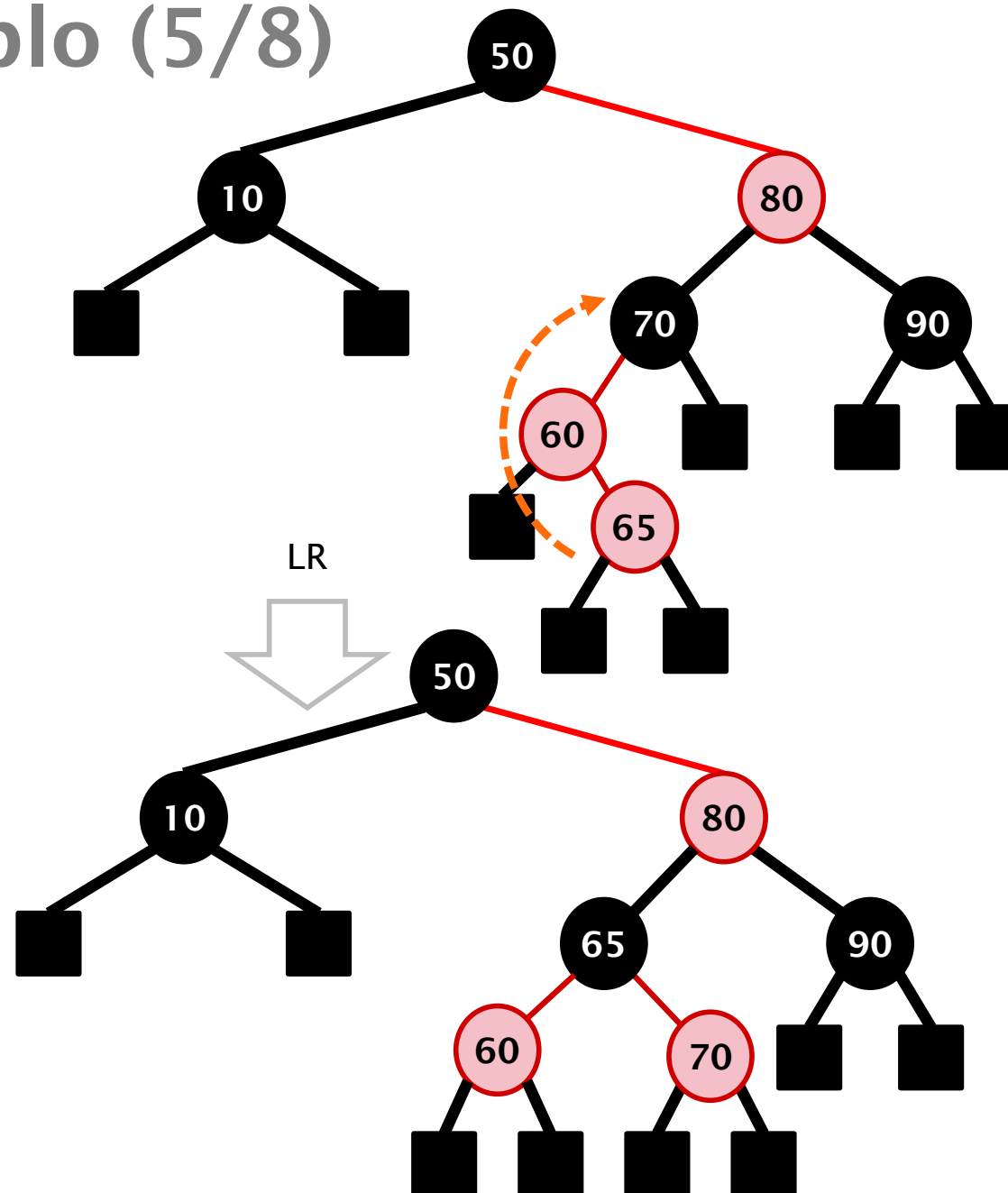
Exemplo (3/8)



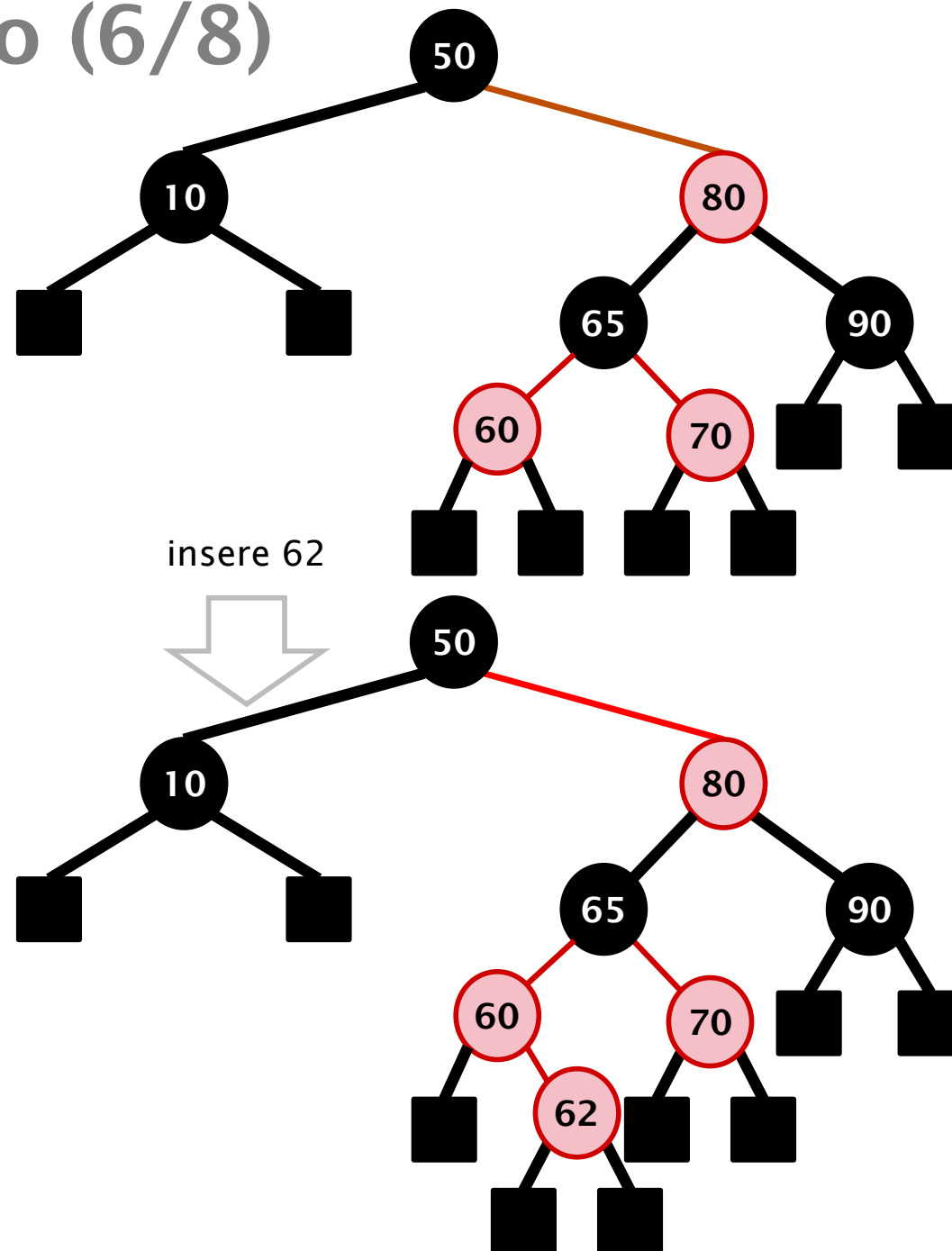
Exemplo (4/8)



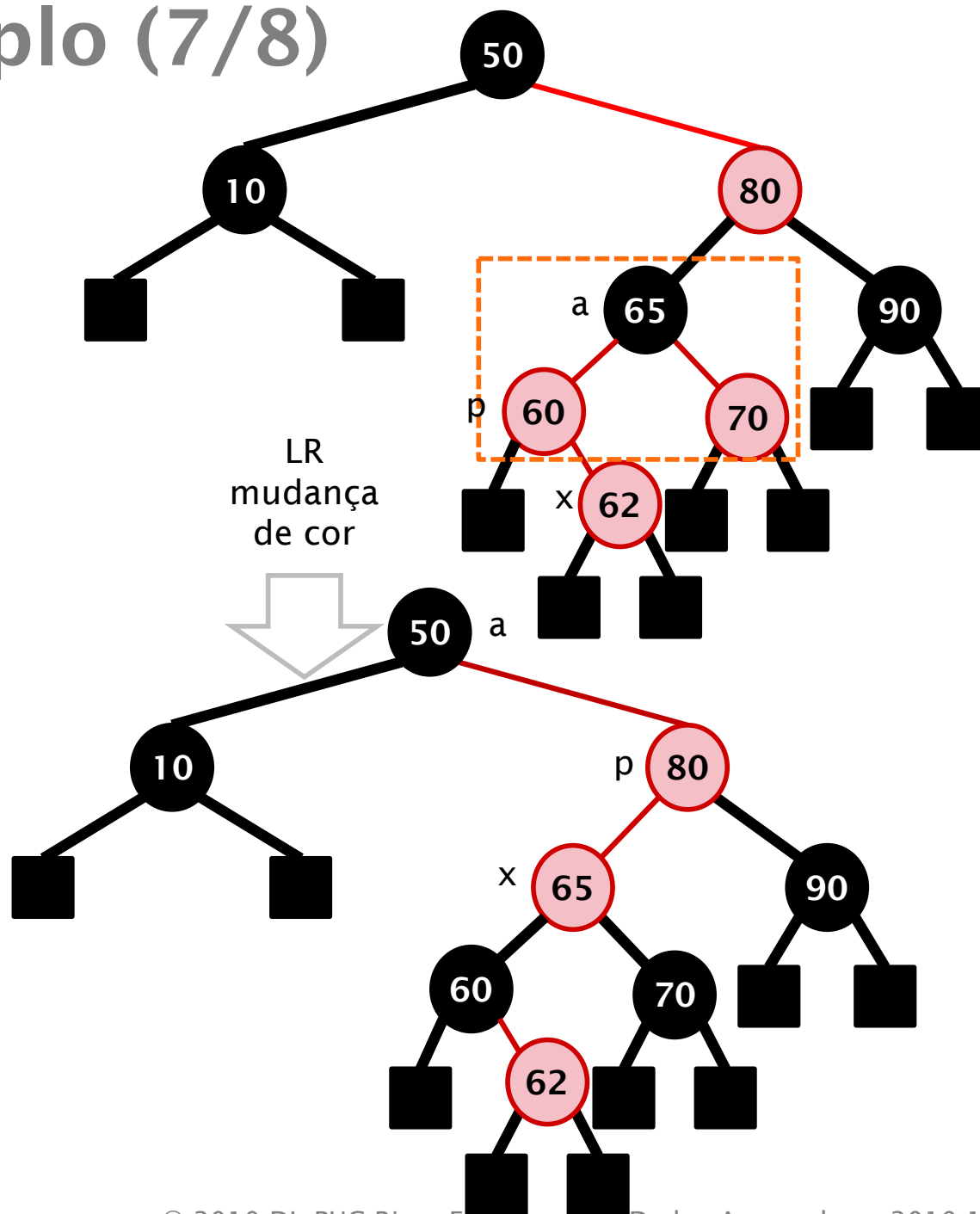
Exemplo (5/8)



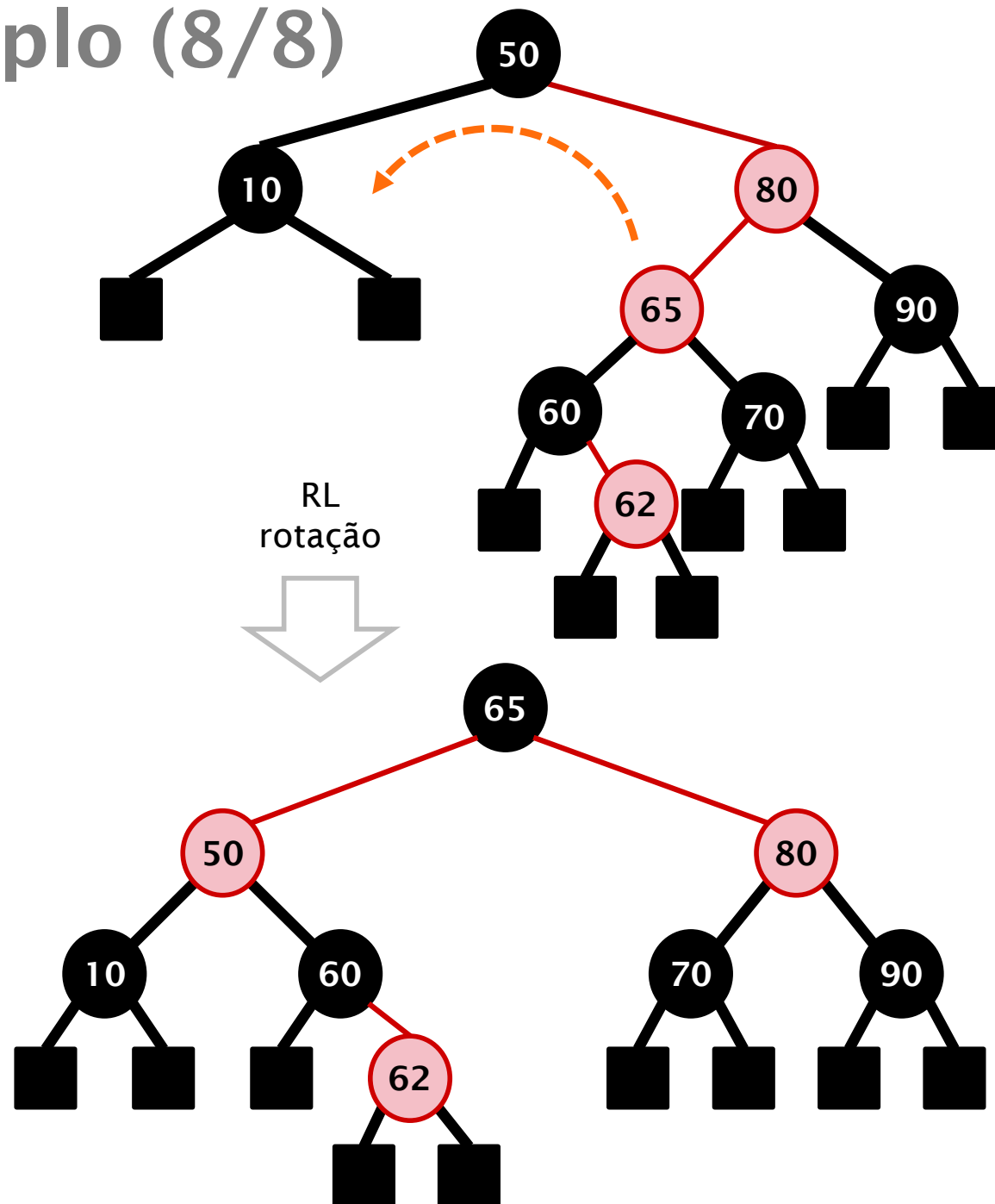
Exemplo (6/8)



Exemplo (7/8)



Exemplo (8/8)



Árvore Red-Black – remoção

Como na inserção o processo segue o mesmos passos da árvore binária de busca convencional. Temos apenas que cuidar para manter as 4 regras RB.

Na remoção a esperança é que o nó x , que vai ser retirado seja vermelho para não alterarmos as alturas pretas. Quando o nó retirado for preto isso reduz o número de nós pretos nos caminhos da raiz para as folhas que ele participa. Caso isso ocorra, precisamos corrigir a árvore com mudanças de cor e/ou estrutura (rotações), como no caso das inserções.

Árvore Red-Black – remoção

Como a remoção segue o processo de uma ABB, existem 3 casos para o nó x , que vai ser removido:

1. é nó folha
2. é um nó que possui **uma** sub-árvore
3. é um nó que possui **duas** sub-árvores

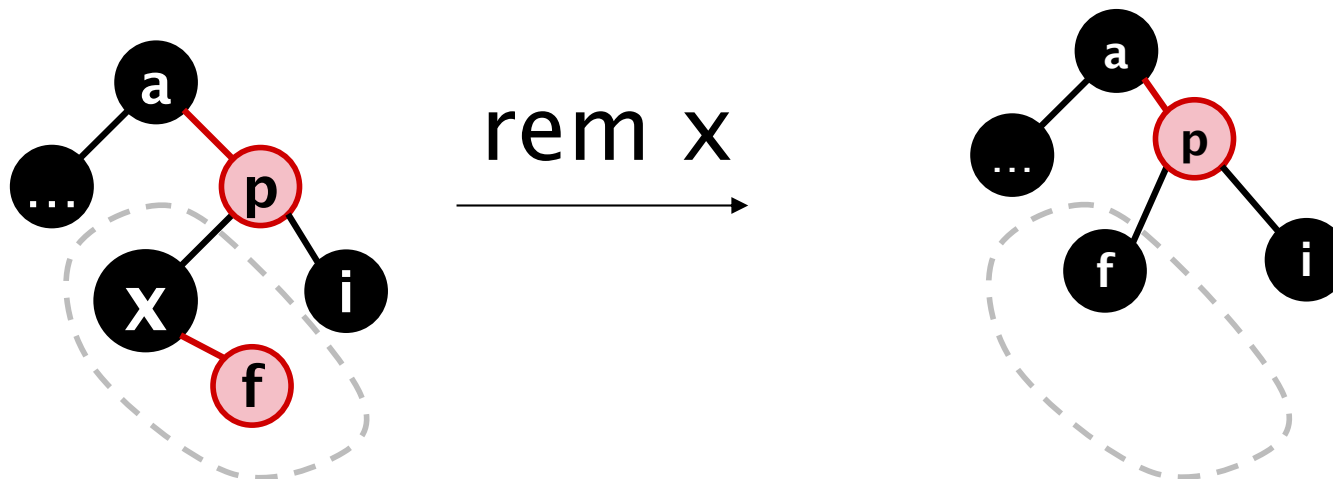
O caso 3 é tratado substituindo o nó que vai ser removido pelo seu sucessor na árvore. E isso não muda a estrutura nesse nó. A mudança ocorre no nó onde estava esse sucessor que necessariamente está nos casos 1 ou 2. O sucessor é o nó mais a esquerda da sub-árvore da direita.

Casos 1 e 2 são os que temos que tratar na correção da remoção.

Remoção com no máximo um filho

Seja f o filho do nó x que deve ser removido (pode ser esquerdo ou direito).

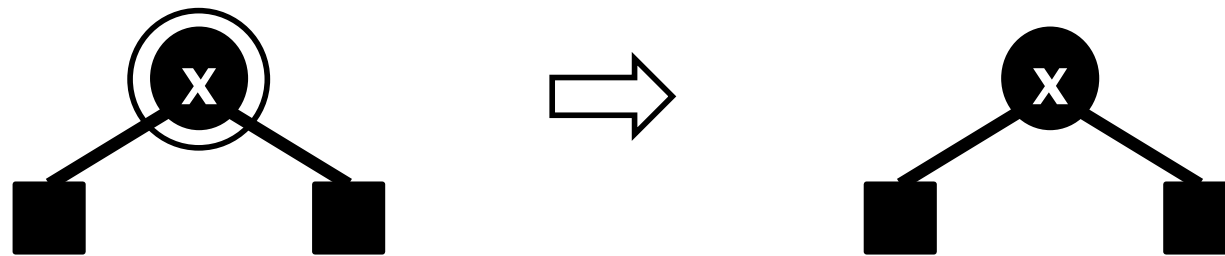
1. Se f ou x for vermelho, f substitui x e é marcado como preto.



Remoção com **x** e **p** pretos: **duplo-preto**

2. Se **f** e **x** forem ambos pretos, **f** substitui **x** e é marcado como **duplo-preto** para indicar um problema que precisa ser corrigido. Essa correção deve acontecer com um nó vermelho absorvendo esse preto ou quando chegamos na raiz.

Caso 0: o nó duplo-preto, **x**, é raiz



Solução do caso 0: torne o nó duplo-preto apenas preto. A altura preta da árvore é reduzida de um, mas as 4 propriedades RB são preservadas.

Laço de correção de um duplo-preto

Enquanto o nó x é duplo-preto faça:

Caso 2.0: raiz

Caso 2.1: tem um irmão vermelho.

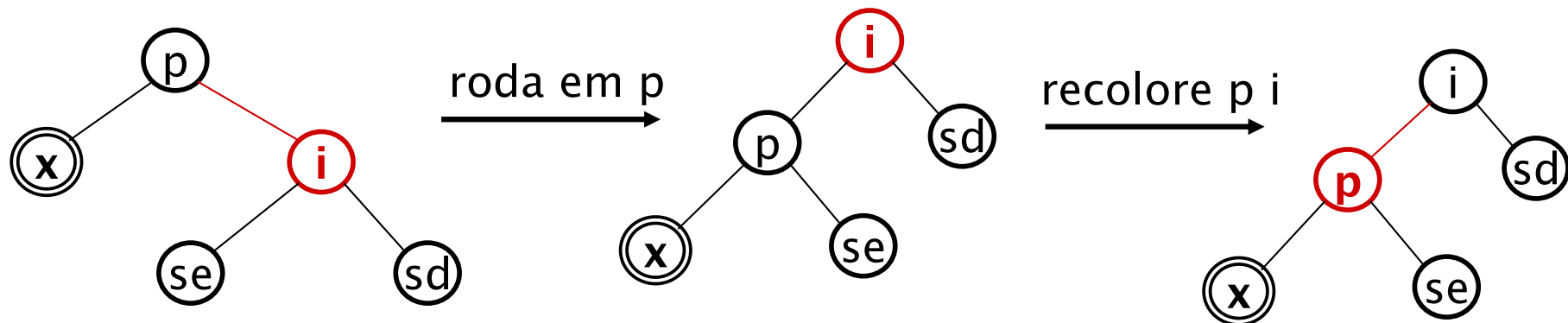
Caso 2.2: irmão preto e dois sobrinhos pretos.

Caso 2.3: irmão preto e sobrinho a esquerda é vermelho e o a direita é preto.

Caso 2.4: irmão preto e sobrinho a direita é vermelho (o a esquerda pode ser preto ou vermelho).

Duplo preto

Caso 2.1: Irmão vermelho

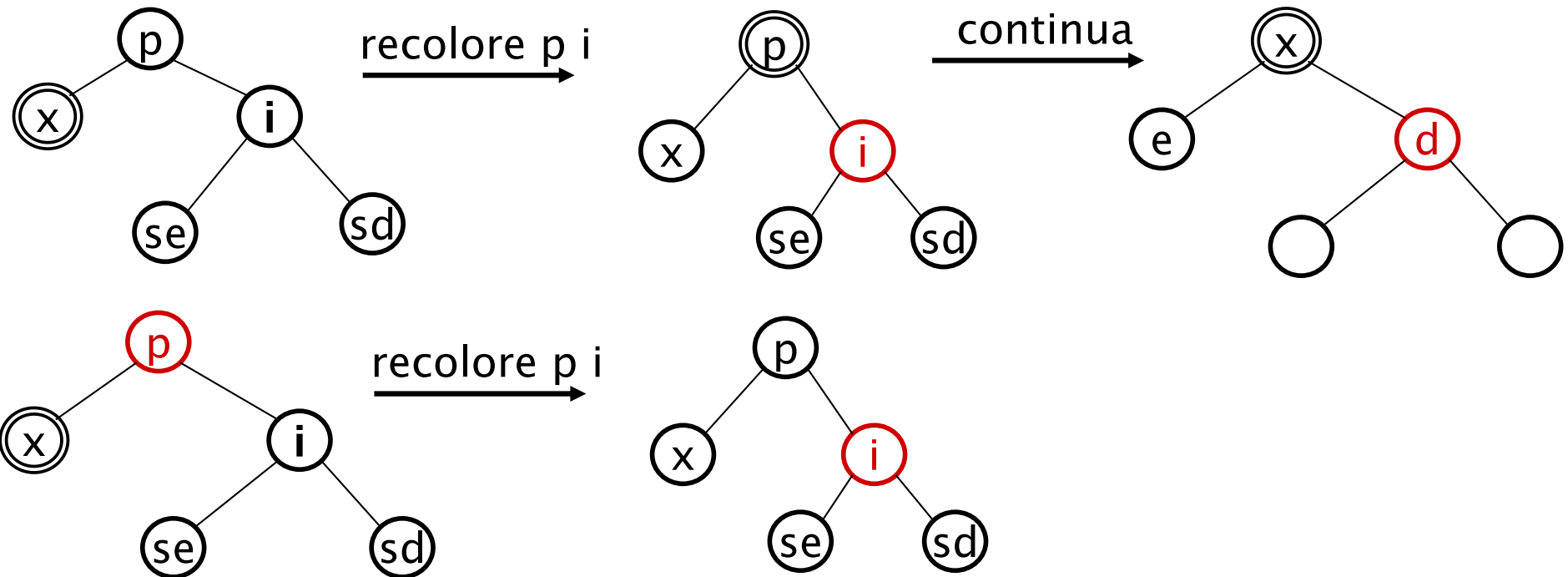


Solução: rode através do pai e troque a cor do pai com o irmão. **Não é um caso terminal.**

Exercício: Faça o caso em que x é o filho a direita.

Duplo preto

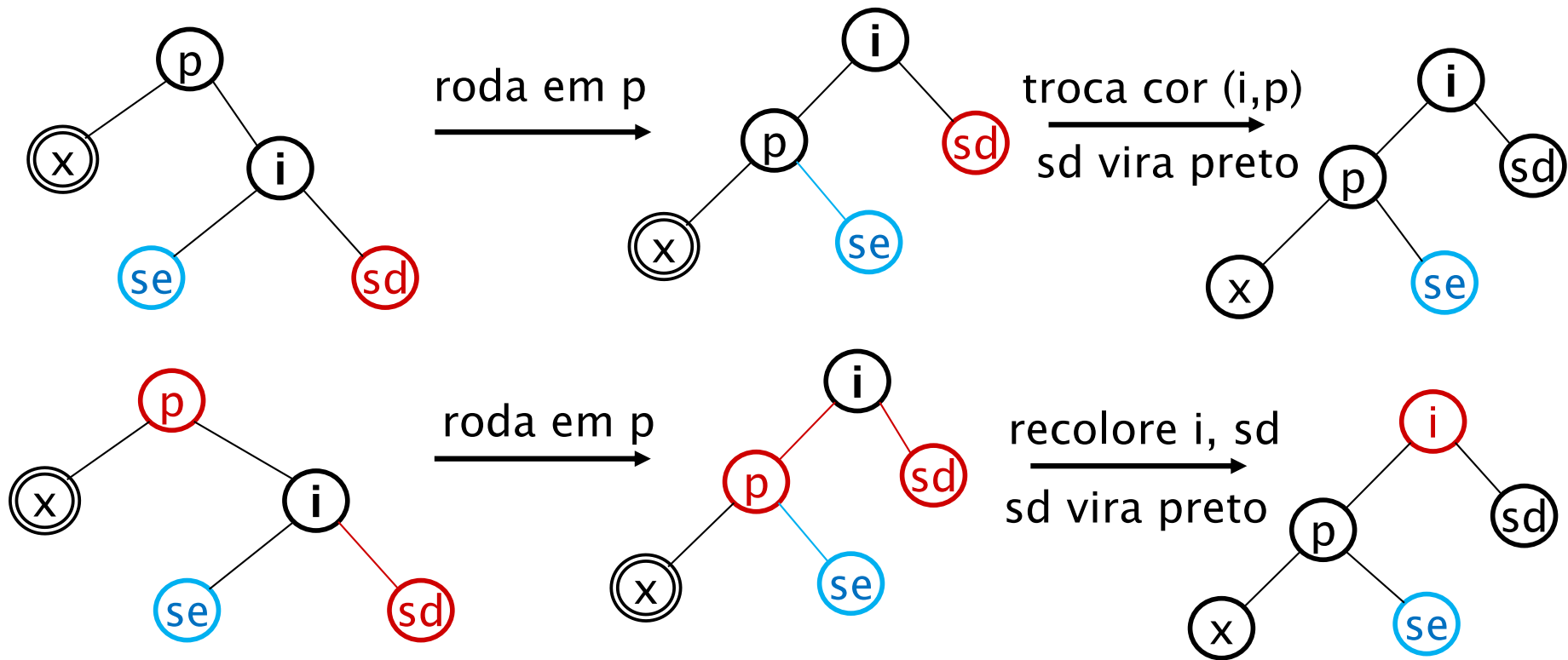
Caso 2.2: Irmão **i** e seus dois filhos também são **pretos**.



Torne o irmão vermelho, **x** vira um preto normal e seu pai ganha um nível de preto. Se **p** for vermelho, vira preto e o algoritmo termina. Se **p** for preto ele vira duplo-preto e o algoritmo continua.

Duplo preto

Caso 2.3: Irmão i é **preto**. O filho “zig” é **vermelho**.

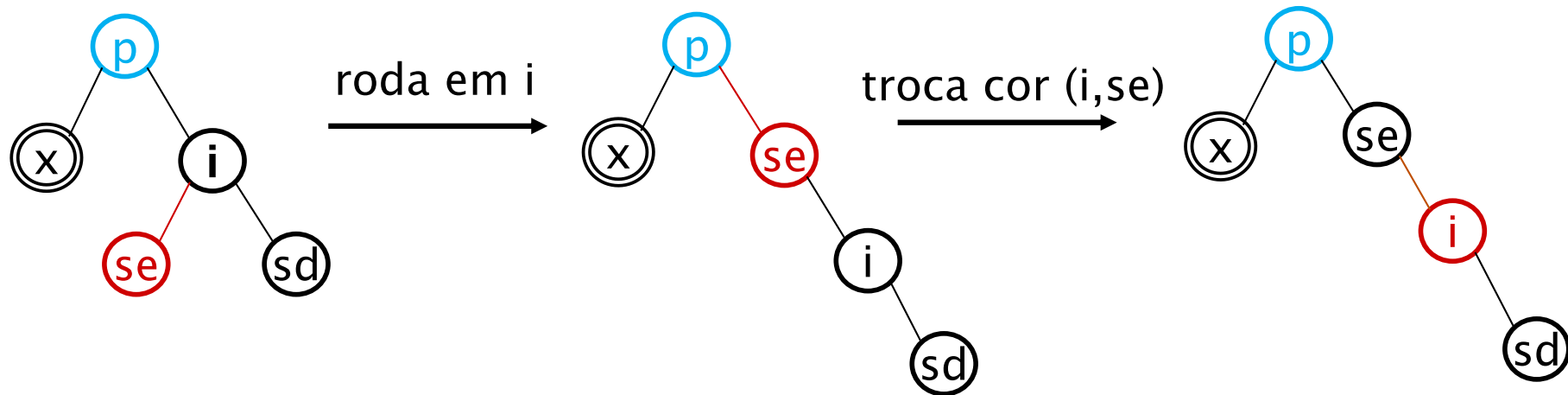


Rode em p , troque a cor de p com i , pinte de preto sd e passe x de duplo-preto para preto.

Caso terminal.

Duplo preto

Caso 2.4: Irmão i é **preto**. O filho “zag” é **vermelho**.



Rode em i , troque a cor de i com se .

Não terminou. Continue no caso 2.3

Resumo visual

