Estruturas de Dados

aula 18: Á arvore rubro-negra

1 Introdução

Na aula passada nós resolvemos o nosso problema de eficiência.

E agora nós tmeos uma estrutura de dados que suporta as operações de buscas, inserção e remoção em tempo $O(\log n)$ — (*no pior caso*)

Mas, ao resolver um problema nós acabamos criando outro.

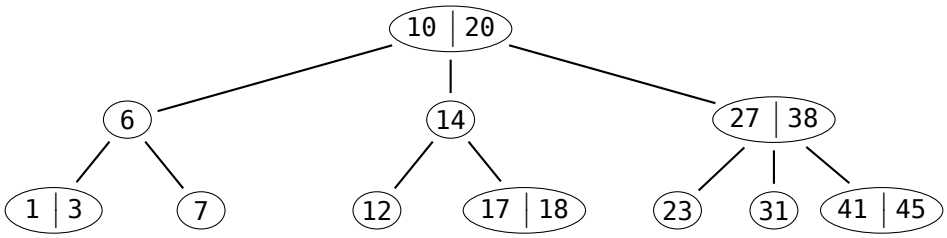
Porque a implementação das árvores 2-3 é bem complicada — (*são ponteiros demais, e casos demais*)

Hoje nós vamos simplificar as coisas por meio de uma esperteza.

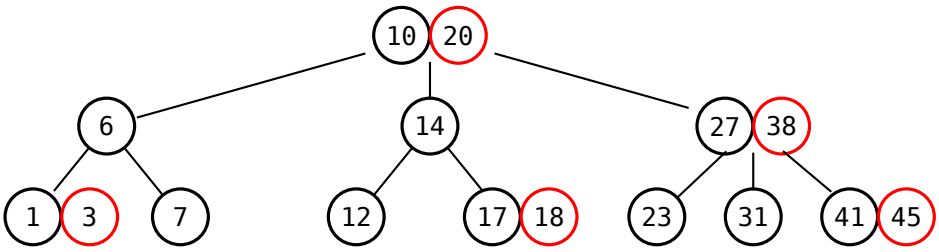
2 Desmontando a árvore 2-3

A esperteza consiste em desmontar a árvore 2-3.

Quer dizer, considere a árvore 2-3 abaixo

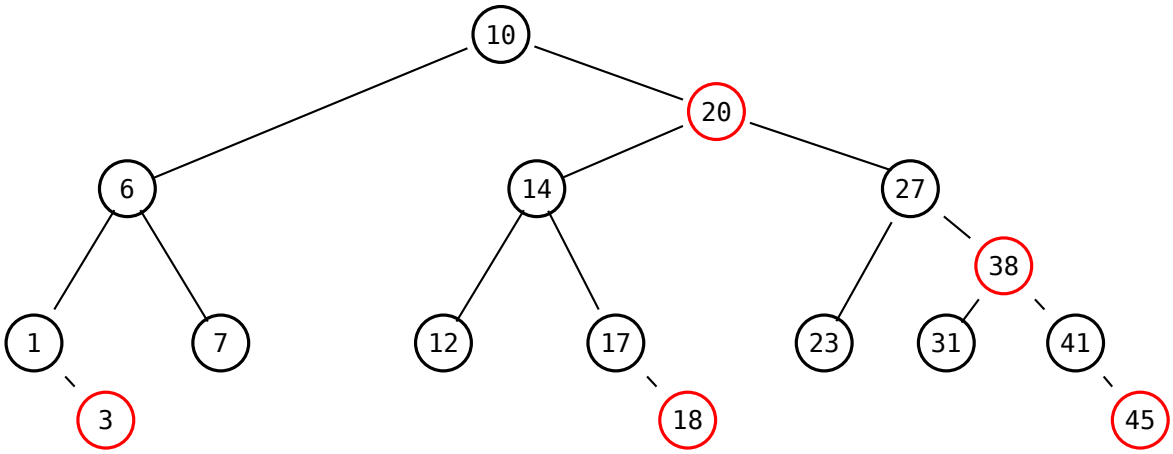


A ideia é visualizar os elementos do tipo 3 como um par de elementos



— (*onde o segundo elemento do par ganha a cor vermelha*)

E daí, nós empurramos os elementos vermelhos um pouquinho mais para baixo — (*i.e., 1/2 nível para baixo*)



Pronto, aqui nós já temos duas boas notícias

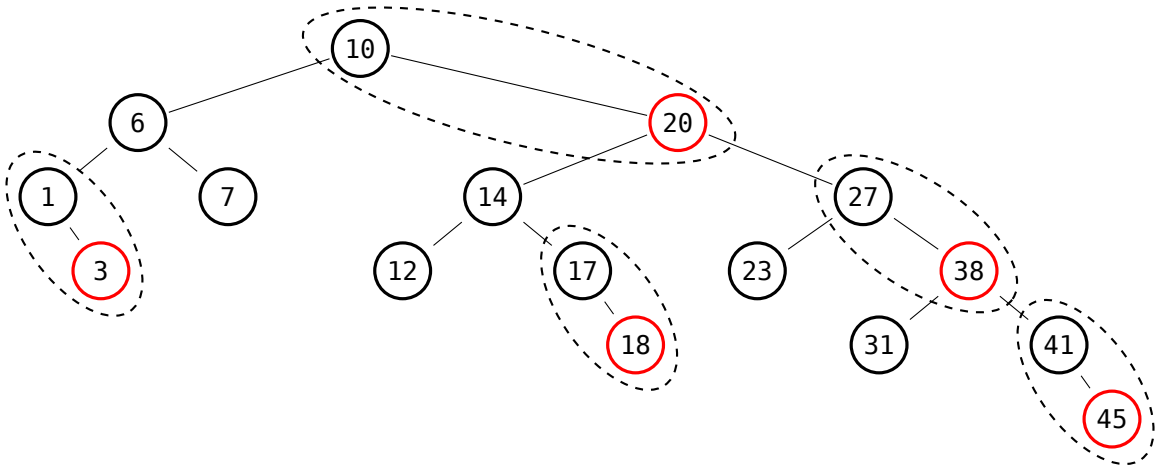
1. A primeira é que agora nós temos uma árvore binária de busca
2. E a segunda é que essa operação no máximo dobra a altura da árvore — (*porque?*)

Daí, como a árvore 2-3 tem altura $O(\log n)$, a nossa árvore binária também vai ter altura $O(\log n)$.

Mas, para a coisa funcionar direito, é preciso que a árvore continue assim depois das operações de inserção e remoção.

E aqui aparece a segunda esperteza.

Quer dizer, a ideia é olhar para a árvore binária tentando ver a estrutura da árvore 2-3



Fazendo isso, a gente vê o seguinte

- Os elementos vermelhos são sempre o filho direito de um elemento preto[1]
- Um elemento preto que tem um filho vermelho corresponde a um elemento do tipo 3 — (*na árvore 2-3*)
- Um elemento preto que não tem filho vermelho corresponde a um elemento do tipo 2 — (*na árvore 2-3*)

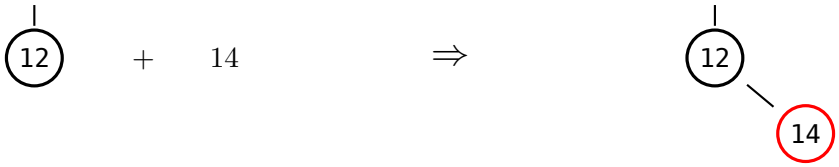
[1] O vermelho devia ser o filho esquerdo, o filho comunista, mas aqui as coisas são assim ...

E agora a gente pode raciocinar sobre a operação de inserção.

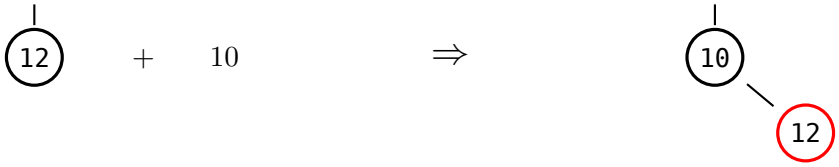
Tudo começa com a descida até o local onde vai acontecer a inserção

A situação mais simples de todas é o caso da folha preta

Daí, se a inserção acontece do lado direito, basta criar o filho vermelho



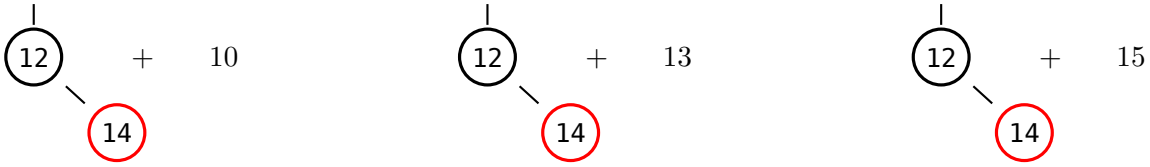
E se a inserção acontece do lado esquerdo, o novo elemento toma o lugar da folha



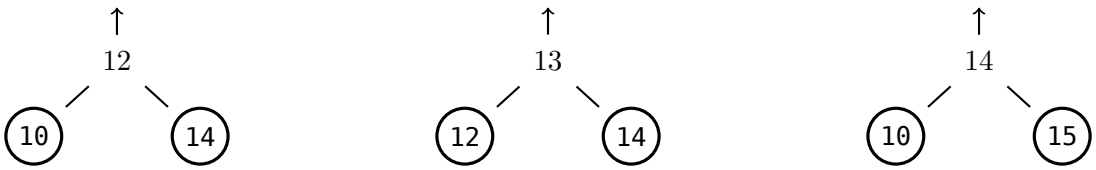
Esses dois casos correspondem à situação onde uma folha do tipo 2 vira uma folha do tipo 3 — (*na árvore 2-3*)

E os próximos casos correspondem à explosão da folha do tipo 3

- a inserção à esquerda de um elemento preto que tem um filho vermelho
- a inserção embaixo de uma folha vermelha



Em todos esses casos, a explosão cria dois elementos pretos, o elemento do meio sobe, e a sua cor vai ser decidida em cima



E em cima, acontece a mesma coisa outra vez.

3 A árvore rubro-negra

Legal.

Agora a gente tem a lógica da árvore 2-3 funcionando em uma árvore binária de busca.

Mas isso ainda não simplificou as coisas — (*porque ainda existem casos demais*)

Para conseguir isso, a gente vai permitir que os elementos pretos também tenham um filho esquerdo vermelho.

Porque daí, a inserção à esquerda de um elemento preto sempre pode ser resolvida criando o filho vermelho.

Certo.

Isso simplifica as coisas.

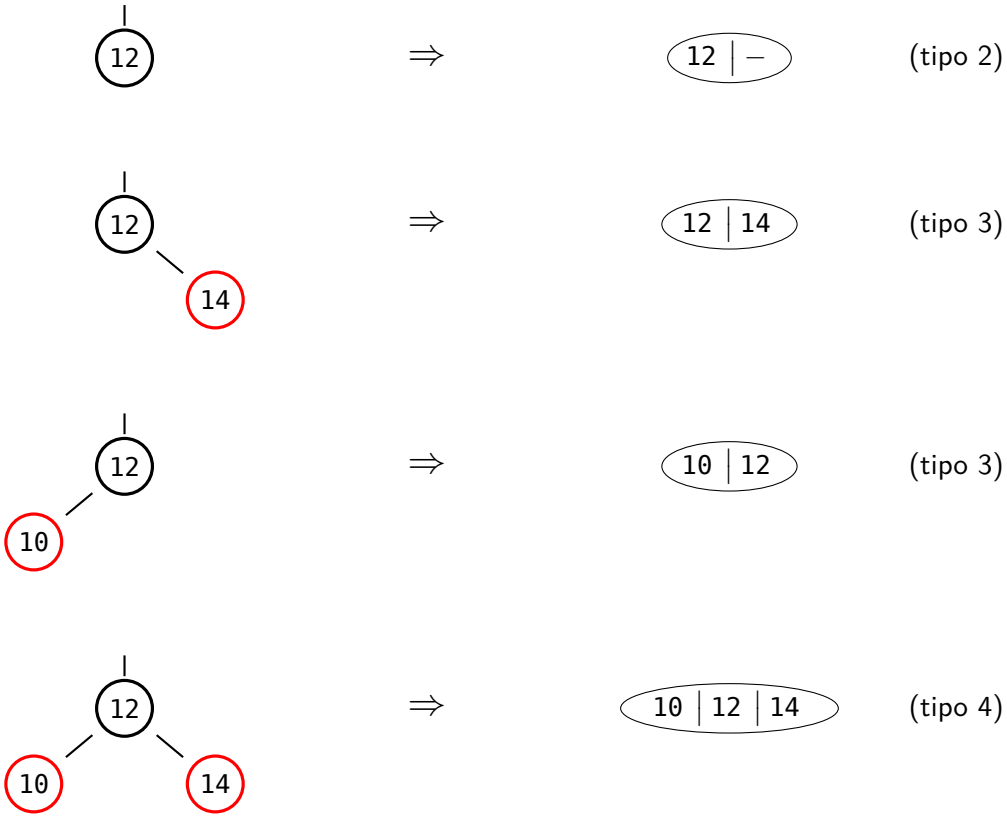
Mas a gente perde a interpretação da árvore 2-3.

E ao perder a interpretação, a gente perde a garantia da altura $O(\log n)$.

Só que, é bem fácil recuperar essa garantia.

Quer dizer, basta encontrar outra interpretação.

E a interpretação é que agora a nossa árvore é uma árvore 2-3-4 desmontada.



Abaixo nós temos um exemplo com a estrutura da árvore 2-3-4 em destaque

(. . .)

Olhando para esse exemplo, a gente vê que

- Os elementos vermelhos são sempre o filho de um elemento preto
- Os elementos pretos formam uma árvore completa — (*porque a árvore 2-3-4 é uma árvore completa*)

Como a subárvore preta é completa, ela tem altura $O(\log n)$.

Daí a gente nota que os elementos vermelhos no máximo dobram a altura da árvore — (*porque?*)

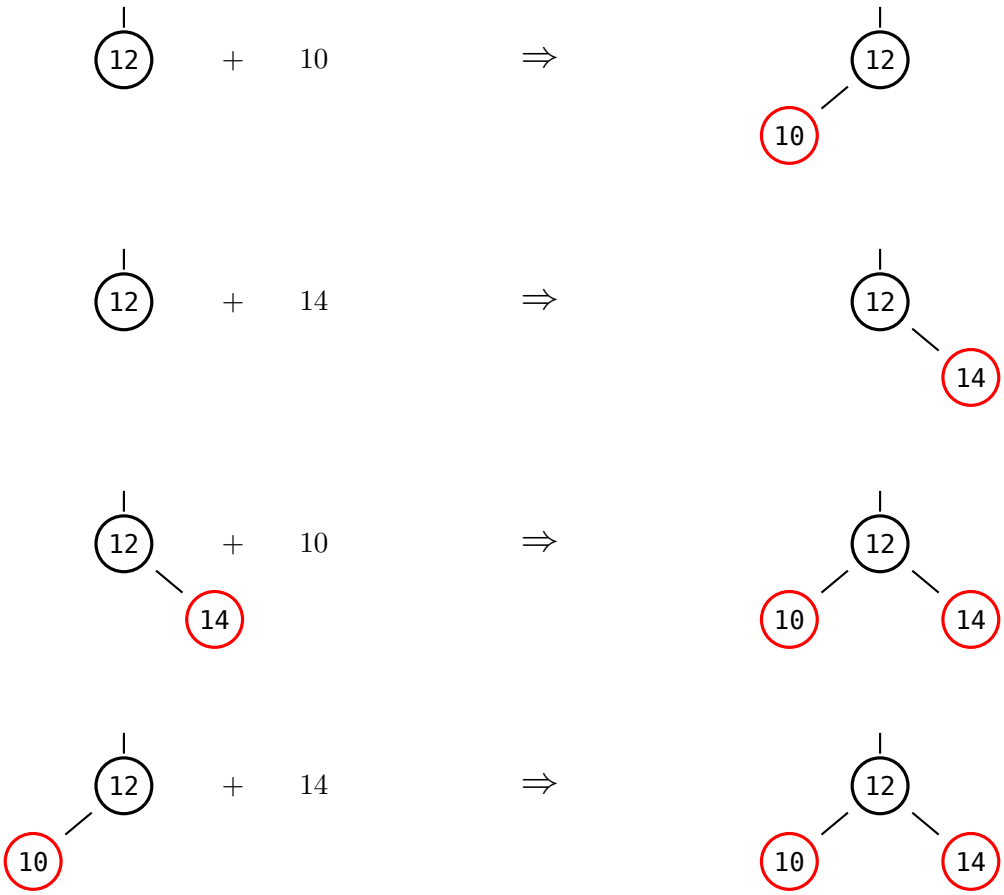
E isso já nos dá a garantia de que a nossa árvore tem altura $O(\log n)$.

Essa árvore é conhecida como a *árvore rubro-negra*.

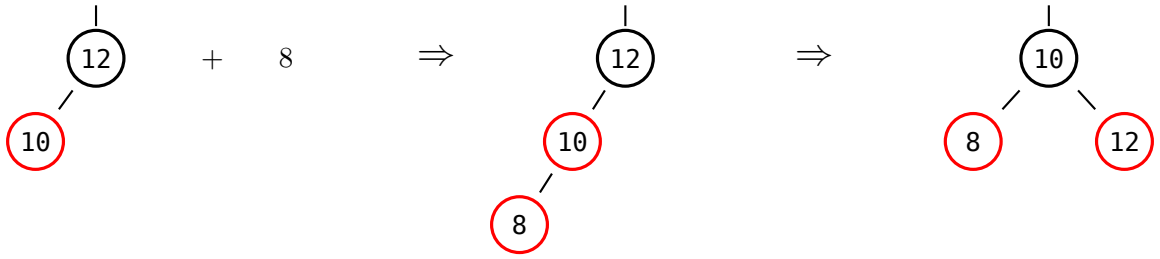
4 Inserção na árvore rubro-negra

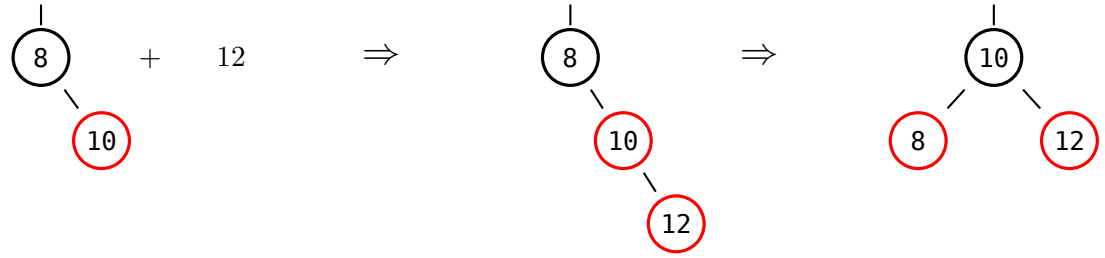
Agora nós precisamos ver como a inserção de um novo elemento na árvore rubro-negra preserva a sua altura $O(\log n)$.

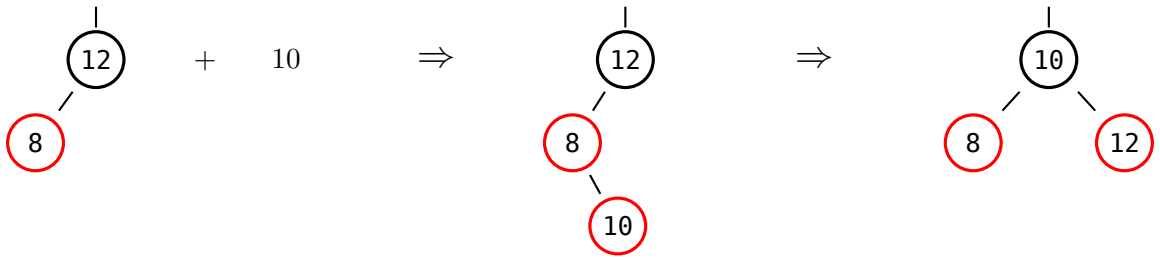
A inserção embaixo de um elemento preto é bem fácil

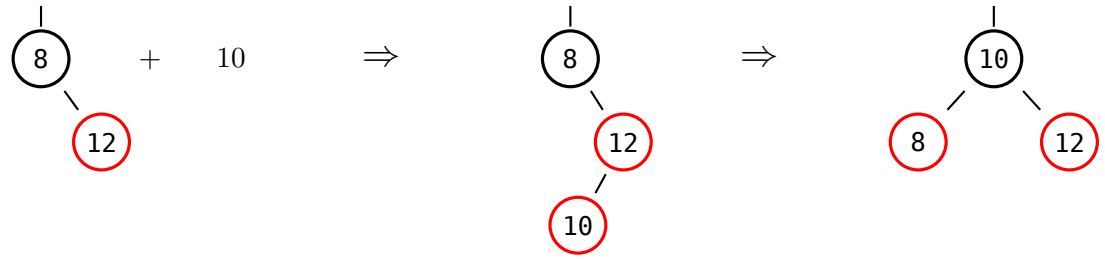


E existe um caso onde a inserção embaixo de um elemento vermelho também é fácil: quando o pai desse elemento vermelho não tem um outro filho vermelho









Quer dizer, em todos esses casos acontece um rearranjo entre o novo elemento, o seu pai vermelho e o seu avô preto.

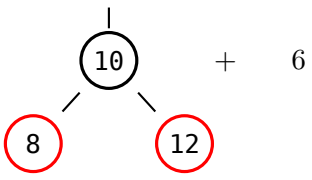
E a coisa acaba aí.

Em todos os casos, nós mantemos um elemento preto no topo com elementos vermelhos embaixo.

Logo, a propriedade da subárvore preta completa é mantida.

E com ela nós temos a garantia de altura $O(\log n)$.

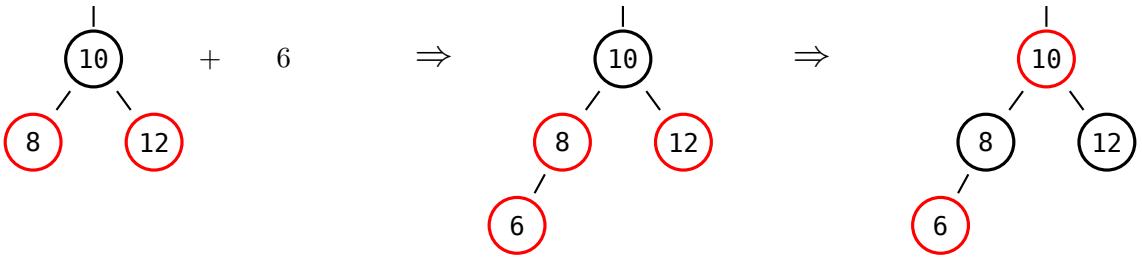
Mas ainda falta um caso.



— (*o caso da explosão na árvore 2-3-4*)

Só que esse caso também é fácil.

Quer dizer, o truque sujo aqui é resolver as coisas só por meio de uma troca de cores



Claro, a troca da cor do avô (10) de preto para vermelho pode ter efeitos colaterais — (*porque o bisavô também pode ser vermelho*)

Mas a gente trata essa situação como a inserção do 10 no bisavô.

E daí, nós temos os mesmos casos outra vez.

Legal, não é?

4.1 Implementação

Sim, mas agora é preciso ver como as coisas funcionam na prática.

O primeiro passo é definir o novo registro

```
struct RegIntRN
    int                num
    int                cor
    struct RegIntRN    *esq, *dir
```

Daí, a gente nota que existem 3 casos para a operação de inserção

1. filho vermelho de um pai preto
2. filho vermelho de um pai vermelho, que é o único filho vermelho do avô
3. filho vermelho de um pai vermelho, onde o avô tem outro filho vermelho

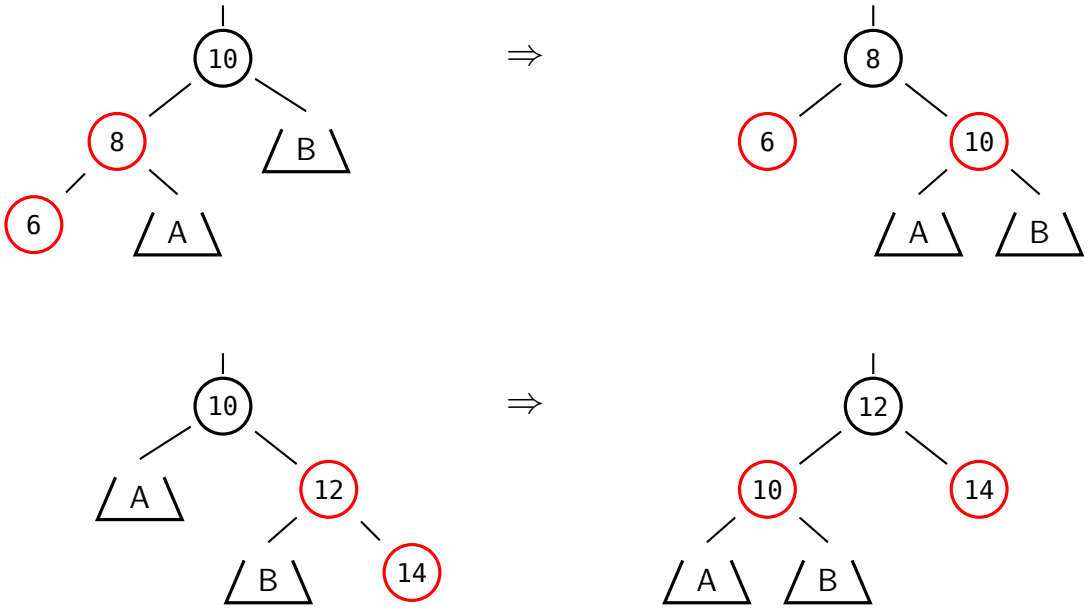
No primeiro caso não há nada a fazer.

E no terceiro caso só o que acontece é uma troca de cores — (*o que pode propagar a ação para cima*)

Então, o único caso que envolve uma pouco mais de trabalho é o segundo.

E no segundo caso, existe um subcaso simples e um subcaso mais complicado.

O subcaso simples é o seguinte



Quer dizer, nos dois casos o pai toma o lugar do avô.

E nós vamos escrever funções auxiliares que implementam essas operações.

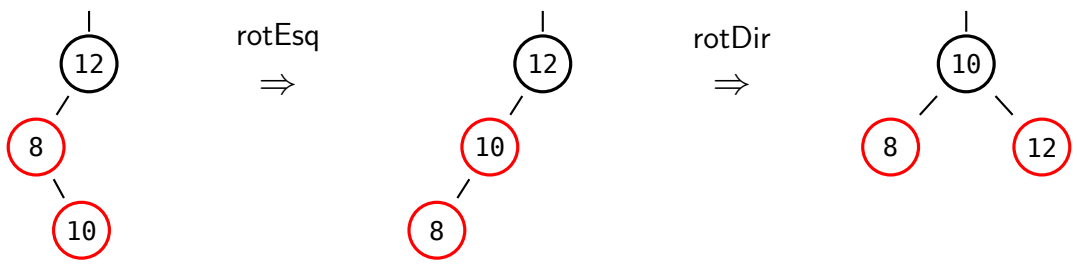
Nos livros, essas operações são chamadas de *rotação*.

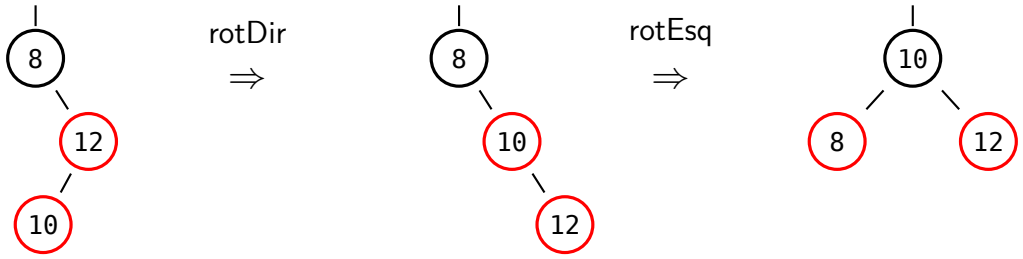
E nós vamos utilizar o mesmo nome aqui

```
func rotDir (pai,avo)

     m = pai->dir
     pai->dir = avo                  // o pai toma o lugar do avô
     avo->esq = m

     Retorna (pai)


func rotEsq (pai,avo)

     m = pai->esq
     pai->esq = avo                  // o pai toma o lugar do avô
     avo->dir = m

     Retorna (pai)
```

Legal.

Agora nós podemos ver que o outro caso se resolve com duas rotações





Agora, a gente pode esboçar o procedimento de inserção em três etapas

1. primeiro nós fazemos a descida na árvore até o local da inserção
2. daí, o novo elemento é inserido na árvore — (*com a cor vermelha*)
3. e depois, nós subimos ajustando as coisas para que não exista um pai vermelho com um filho vermelho

E observando como os ajustes são feitos, a gente nota que é mais fácil implementá-los a partir da posição do avô — (*porque?*)

A coisa fica assim

```
func inserçãoRN (k,q)

     Se (q == Nulo)                                  // cria o novo registro
          r = Novo (RegIntRN)
          r->num,cor,esq,dir = k, Verm, Nulo, Nulo
          Retorna (r)

     Sse (q->num > k)                                // descida pela esquerda

          q->esq = inserçãoRN (k,q->esq)
          Se ((q->esq)->cor == Preto)    Retorna (q)

          neto1  = Nulo;   neto2 = Nulo              // verificação e ajuste
          Se (q->esq->esq != Nulo  E  (q->esq->esq)->cor == Verm)
               neto1 = q->esq->esq
          Se (q->esq->dir != Nulo  E  (q->esq->dir)->cor == Verm)
               neto2 = q->esq->dir
          Se (neto1 == Nulo  E  neto2 == Nulo)   Retorna (q)

          Sse ((q->dir)->cor == Verm)                // caso 3
               q->cor = Verm
               (q->esq)->cor = Preto;   (q->dir)->cor = Preto
               Retorna (q)

          Sse (neto1 != Nulo)                        // caso 2.1
               q = rotDir (q->esq,q)
               q->cor = Preto;   (q->dir)->cor = Verm
               Retorna (q)
```

```
Senão
      q->esq = rotEsq ( neto2, q->esq )
      q = rotDir (q->esq,q)
      q->cor = Preto;   (q->dir)->cor = Verm
      Retorna ( q )
```