UNIFESP

UNIVERSIDADE FEDERAL DE SÃO PAULO

1933

Árvores vermelho-preto

Prof. Lilian Berton

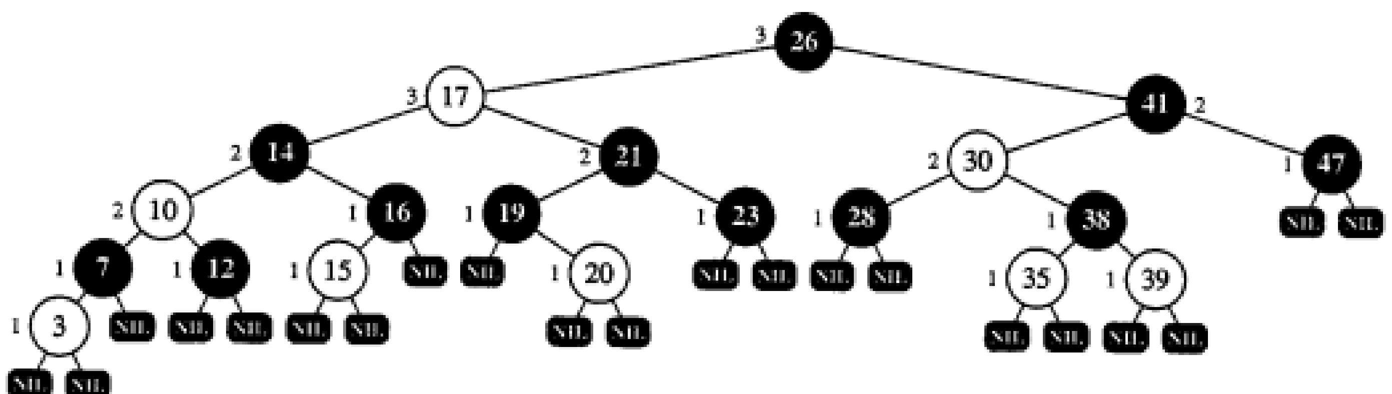
São José dos Campos, 2018

Aula totalmente baseada no material de Thomas Cormen.

Árvores vermelho-preto

- São árvores de pesquisa binária com um bit extra de armazenamento por nó: sua cor, que pode ser vermelho ou preto. Satisfazem as seguintes propriedades:

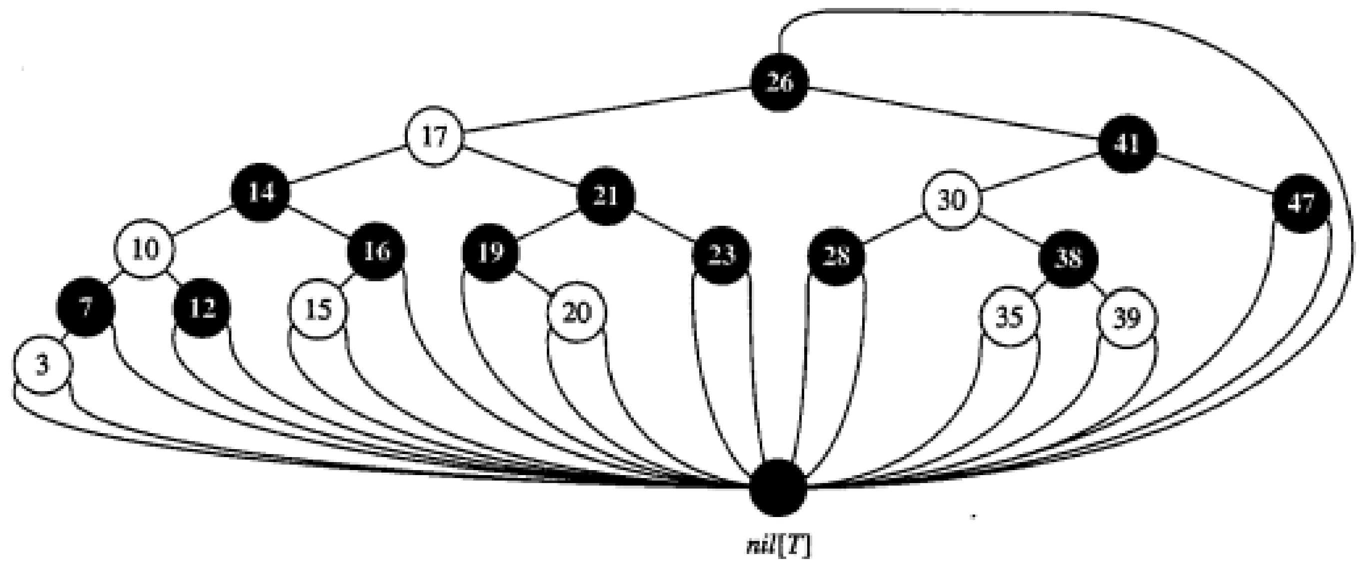
1. Todo nó é **vermelho** ou **preto**.
2. A raiz é **preta**.
3. Toda folha é **preta (NULL)**.
4. Se um nó é **vermelho**, então ambos os seus **filhos são pretos**.
5. Para cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o **mesmo número de nós pretos**.



(a)

Árvores vermelho-preto

- Em uma árvore vermelho-preto podemos substituir cada folha Null por uma única sentinela Null que é sempre preta, e também é o pai da raiz.

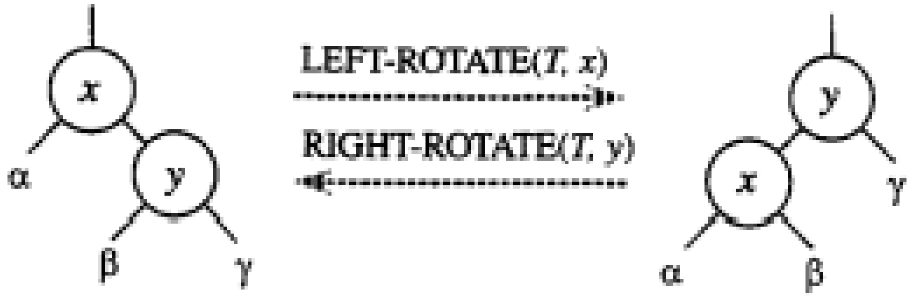


- A estrutura original foi inventada em 1972 por Rudolf Bayer que a chamou de "Árvores Binárias B Simétricas", mas adquiriu este nome moderno em um artigo de 1978 escrito por Leonidas J. Guibas e Robert Sedgewick.

Rotações

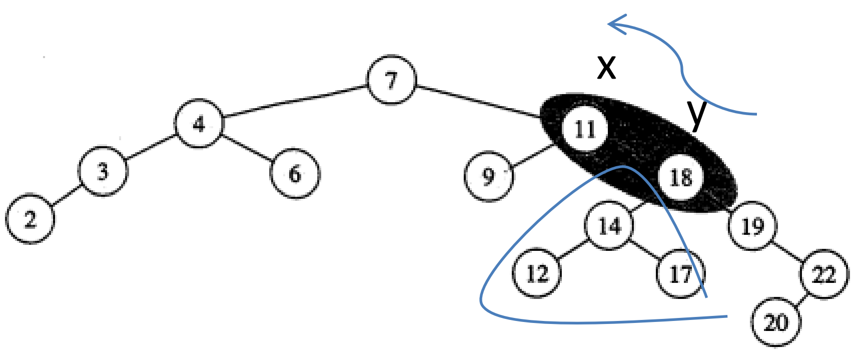
- As operações de inserção e remoção demoram tempo O (log n).
- Elas **podem violar as propriedades vermelho-preto** enumeradas anteriormente. Pode ser necessário mudar as cores de alguns nós e a estrutura de ponteiros.

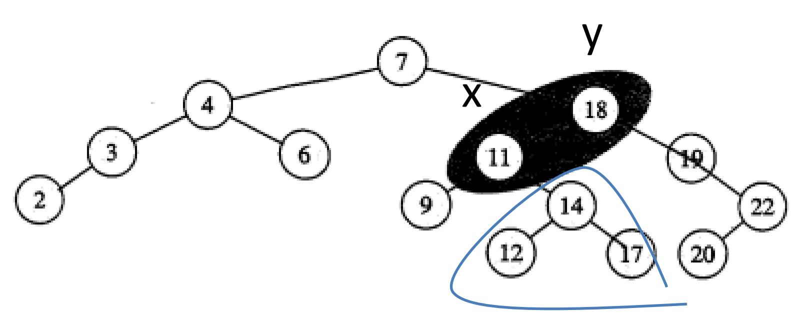
- **Rotação a esquerda**: faz de y a nova raiz da subarvore, tendo x como filho da esquerda de y e o filho da esquerda de y como filho da direita de x.
- **Rotação a direita**: faz de x a nova raiz da subarvore, tendo y como filho da direita de x e o filho da direita de x como filho da esquerda de y.



Rotação a esquerda com ponteiro para o pai

```
LEFT-ROTATE(T, x)
y ← direita[x]                          ▷ Define y.
direita[x] ← esquerda[y]                ▷ Faz da subárvore esquerda de y a subárvore direita de x.
p[esquerda[y]] ← x                      - Atualiza o pai da esq de y
p[y] ← p[x]                             ▷ Liga o pai de x a y.
if p[x] = nil[T]
    then raiz[T] ← y                    - Se x era raiz, torna y a nova raiz
    else if x = esquerda[p[x]]
            then esquerda [p[x]] ← y    - Senão atualiza o ponteiro do pai de x apontar para y
            else direita[p[x]] ← y
esquerda[y] ← x                         ▷ Coloca x à esquerda de y.
p[x] ← y                                - Atualiza o pai de x como y
```





Rotação a direita segue a mesma ideia.

Inserção com ponteiro para o pai

```
RB-INSERT(T, z)
y ← nil[T]                          - penultimo
x ← raiz[T]                         - ultimo
while x ≠ nil[T]                    - Inicia na raiz e percorre
  do y ← x                          a árvore até chegar na folha
     if chave[z] < chave[x]
        then x ← esquerda[x]
        else x ← direita[x]
p[z] ← y                            - Atualiza o pai de z
if y = nil[T]                       - Se a arvore era vazia
  then raiz[T] ← z                  z se torna a nova raiz
  else if chave[z] < chave[y]
          then esquerda[y] ← z
          else direita[y] ← z
esquerda[z] ← nil[T]
direita[z] ← nil[T]                 - os filhos de z são Null
cor[z] ← VERMELHO                   - z é vermelho
RB-INSERT-FIXUP(T, z)
```

Verifica se alguma propriedade da árvore foi violada e concerta

- Insere-se um elemento z como se fosse uma árvore binária comum e colore **z de vermelho**.
- Para garantir que as propriedades vermelho-preto serão preservadas chama-se outro procedimento para re-colorir os nós e fazer rotações.

Inserção: quais propriedades podem ser violadas?

1. Todo nó é **vermelho** ou **preto**.

2. A raiz é **preta**.

3. Toda folha é **preta (NULL)**.

4. Se um nó é **vermelho**, então ambos os seus **filhos são pretos**.

5. Para cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o **mesmo número de nós pretos**.

Inserção: quais propriedades podem ser violadas?

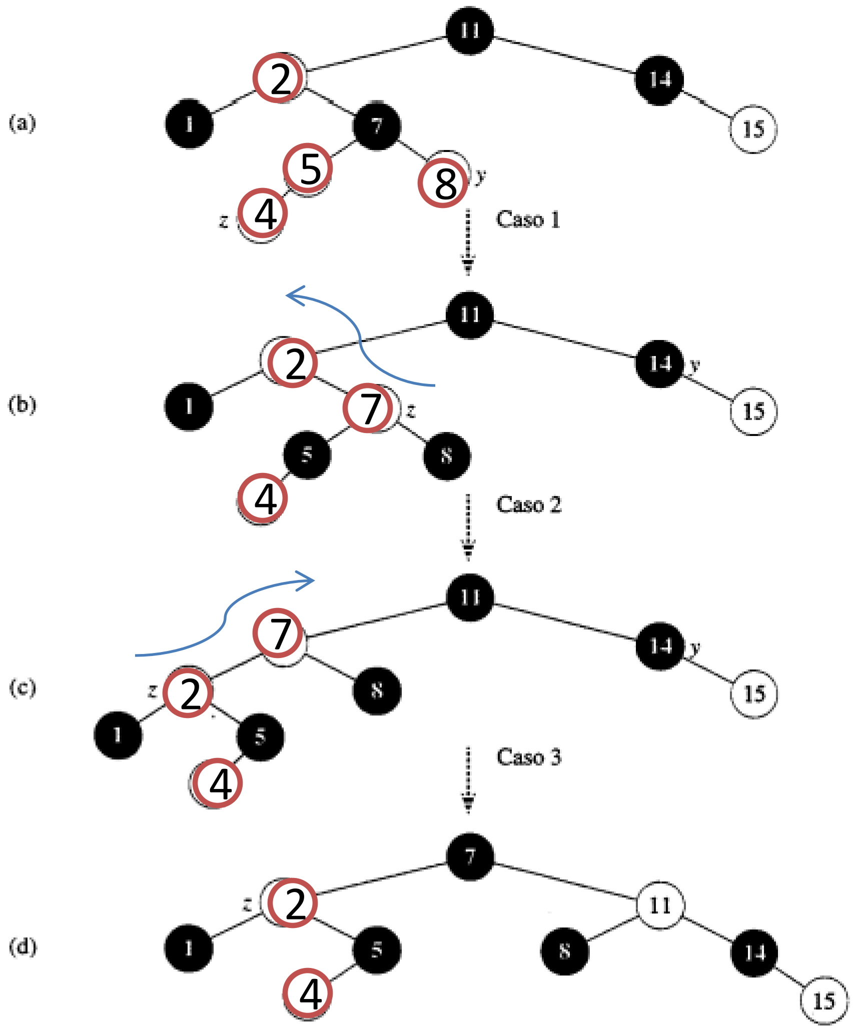
- **1 e 3** -> continuam válidas, pois z é vermelho e ambos os filhos do nó recém inserido são Null.

- **5** -> diz que o número de nós pretos é igual em todo caminho a partir de um nó, continua válida, pois o nó z substitui o Null (preto) e o nó z é vermelho com filhos Null (pretos).

- **2** -> diz que raiz seja preta, **pode ser violada** se z é raiz!
- Nesse caso troca a cor da raiz para **preto**.

- **4** -> diz que um nó vermelho não pode ter um filho vermelho, **pode ser violada** se p[z] é vermelho! Pois z também é colorido de vermelho.
- Nesse caso precisa aplicar os tratamentos a seguir.

Inserção: 3 casos para tratar

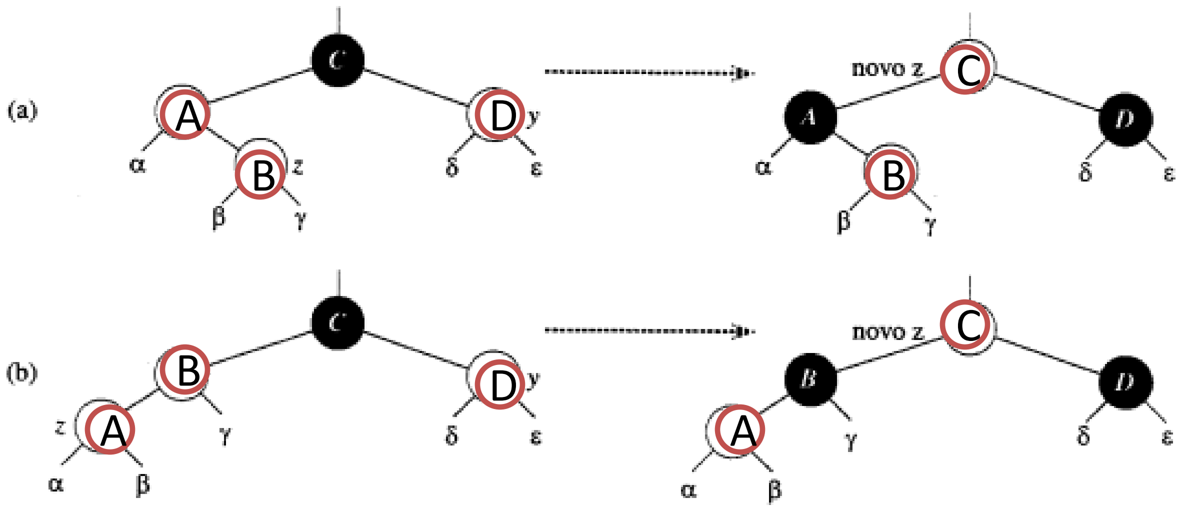


- Como z e seu pai p[z] são vermelhos, viola-se a propriedade 4. Como o tio y de z é vermelho, aplica-se o **caso 1**. Os nós são re-coloridos, o ponteiro de z é movido para cima na árvore, resultando na árvore da Fig. (b).

- Novamente, z e seu pai são vermelhos, mas o tio y de z é preto. Como z é o filho da direita de p[z], o **caso 2** pode ser aplicado. Uma rotação a esquerda é executada, a árvore resultante é mostrada em (c).

- Agora z é o filho da esquerda de seu pai, o **caso 3** pode ser aplicado. Uma rotação a direita produz a árvore em (d), que é uma árvore vermelho-preto válida.

Caso 1: o tio y de z é vermelho



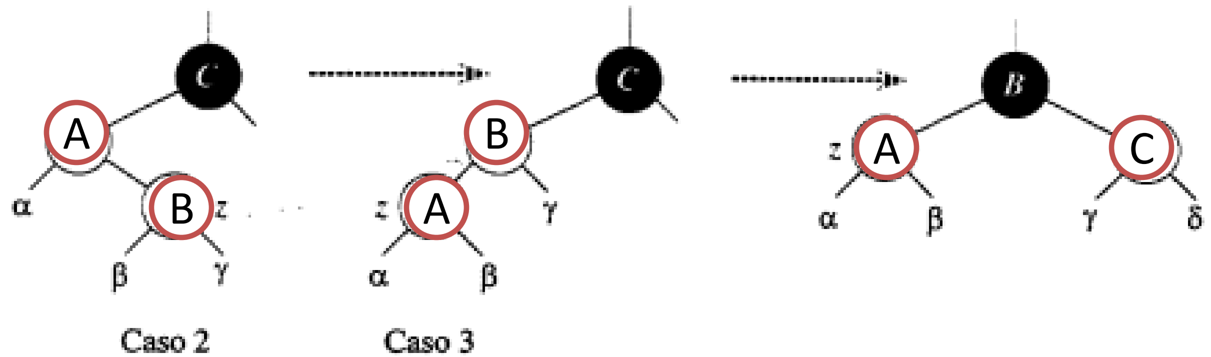
• Nesse caso a **propriedade 4 é violada, pois z e seu pai p[z] são vermelhos.**

• Se z é um filho da direita (a) ou da esquerda (b) a mesma ação é tomada.

• O código para o **caso 1** altera as cores de alguns nós, preservando a propriedade 5: todos os caminhos descendentes desde um nó até uma folha têm o mesmo número de pretos.

• O loop while continua com o avô p[p[z]] de z como o novo z. Uma nova violação da propriedade 4 só pode ocorrer entre o novo z e seu pai que são vermelhos.

Caso 2 e 3: **o tio y de z é preto** e z é filho da direita (2) ou da esquerda (3)



- A propriedade 4 é violada pois z e seu pai p[z] são vermelhos.
- O **caso 2** é transformado no **caso 3** por uma rotação a esquerda, preservando a propriedade 5 que todos os caminhos de um nó até uma folha tem o mesmo número de pretos.
- O **caso 3** faz algumas mudanças de cores e uma rotação a direita, preservando a propriedade 5.

Inserção – procedimento para tratar os 3 casos

```
RB-INSERT-FIXUP(T, z)
while cor[p[z]] = VERMELHO
      do if p[z] = esquerda[p[p[z]]]
            then y ← direita[p[p[z]]]
                 if cor[y] ← VERMELHO
                    then cor[p[z]] ← PRETO                ▷ Caso 1
                         cor[y] ← PRETO                   ▷ Caso 1
                         cor[p[p[x]]] ← VERMELHO          ▷ Caso 1
                         z ← p[p[z]]                      ▷ Caso 1
                    else  if z = direita[p[z]]
                            then z ← p[z]                 ▷ Caso 2
                                 LEFT-ROTATE(T, z)        ▷ Caso 2
                          cor[p[z]] ← PRETO               ▷ Caso 3
                          cor[p[p[z]]] ← VERMELHO         ▷ Caso 3
                          RIGHT-ROTATE(T, p[p[z]])        ▷ Caso 3
            else (igual a cláusula then
                  com "direita" e "esquerda" trocadas)
cor[raiz[T]] ← PRETO
```

- z é o novo elemento inserido

Enquanto o pai de z for vermelho repete:

- Encontra y o tio de z

- Se a cor do tio é vermelho

Pinta o pai de z de preto
Pinta o tio de z de preto
Pinta o avô de vermelho
Passa o ponteiro z para o avô

- Se a cor do tio preta e z é filho a direita

Faz rotação a esquerda sobre o pai de z

Pinta o pai de z de preto
O avô de z de vermelho
Faz rotação a esquerda sobre o avô de z

- finaliza pintando a raiz de preto

Análise de complexidade

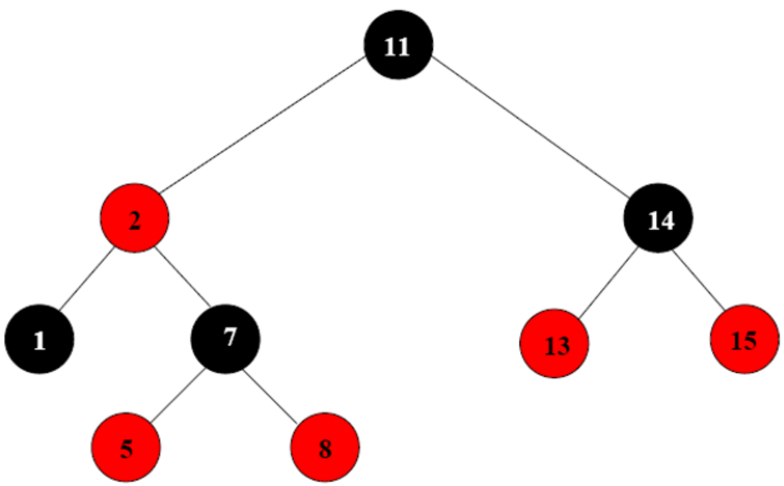
- Como a altura de uma árvore vermelho-preto sobre n nós é O(log n) as linhas 1 a 16 de RB-INSERT levam o tempo O(log n).

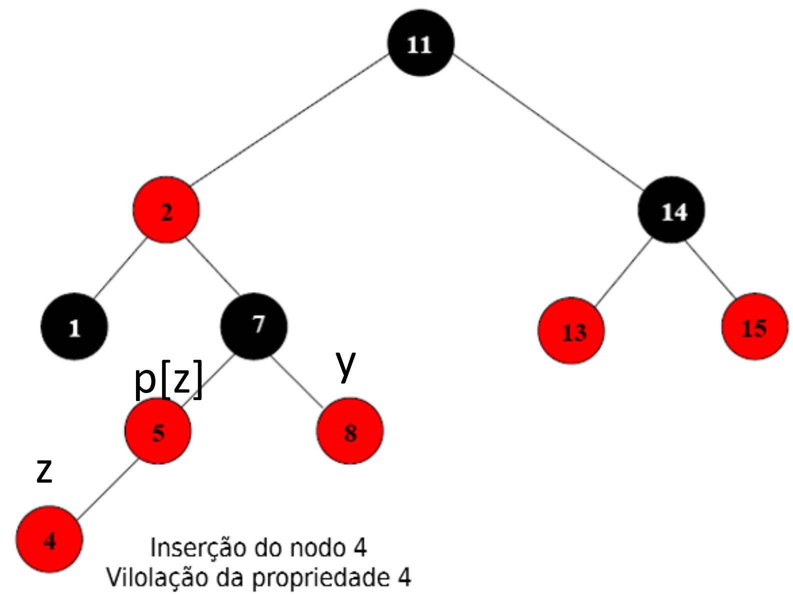
- Em RB-INSERT-FIXUP, o loop while só se repete no caso 1, onde o ponteiro z sobe dois níveis na árvore. O número total que o loop while pode executar é O(log n). Nunca é executado mais de duas rotações, pois o loop while termina se o caso 2 ou 3 é executado.

- Assim RB-INSERT demora um tempo total O(log n).

Exemplo inserção

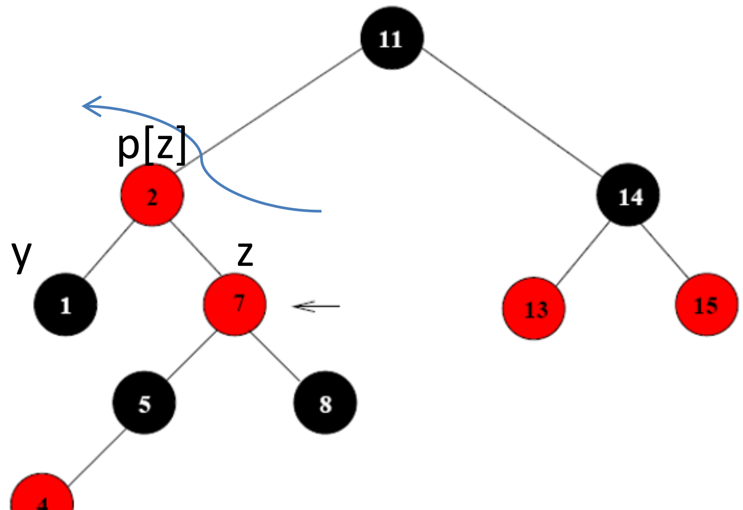
- Inserção do valor 4
- Aplicar caso 1 – re-colorir alguns nós



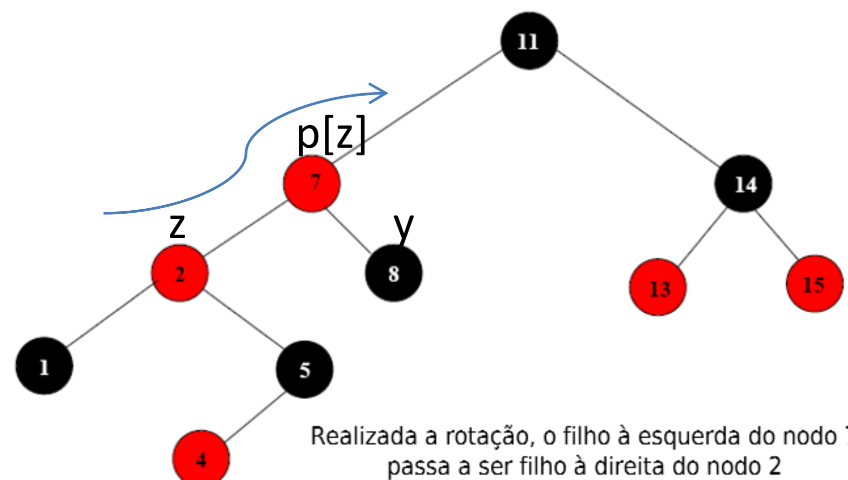


Exemplo inserção

- Aplicar caso 2 – rotação a esquerda
- Aplicar caso 3 – rotação a direita

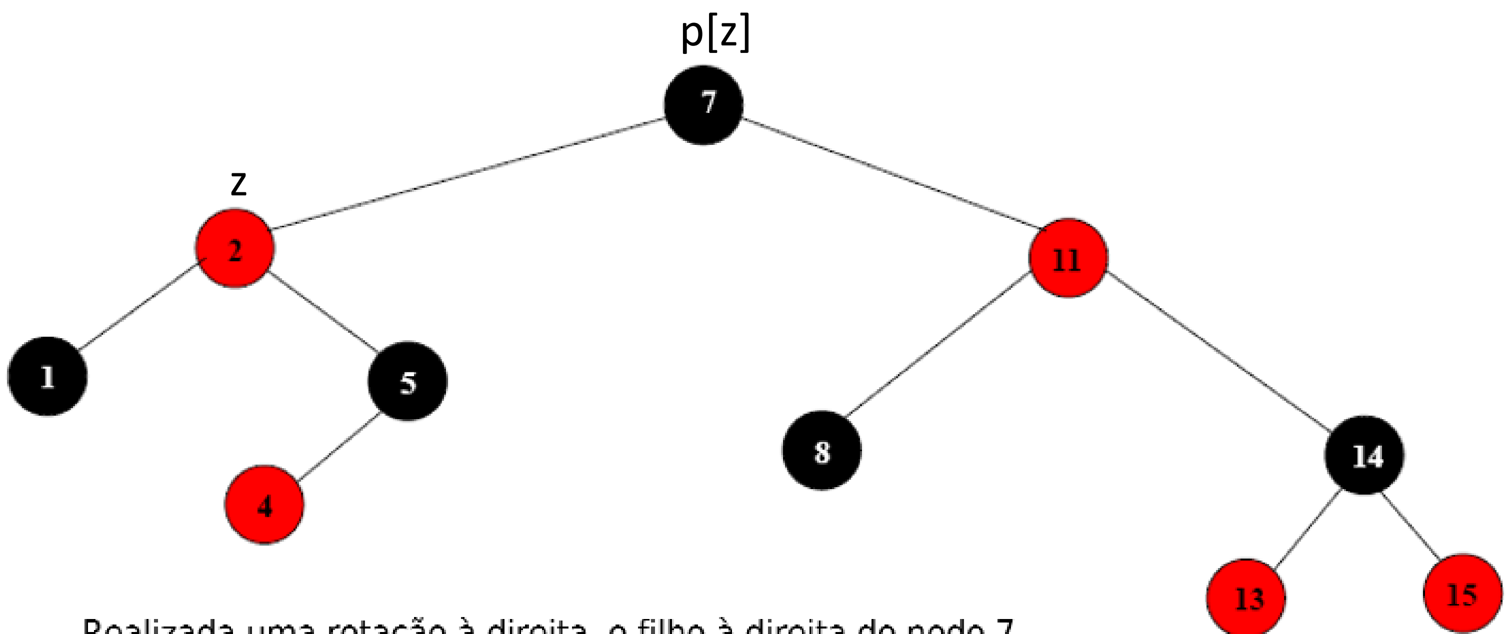


Nodos 5 e 8 passam a ser pretos
Outra violação da propriedade 4 entre os nodos 2 e 7
Necessária uma rotação à esquerda



Realizada a rotação, o filho à esquerda do nodo 7
passa a ser filho à direita do nodo 2
Necessária rotação à direita.

Exemplo inserção



Realizada uma rotação à direita, o filho à direita do nodo 7
passa a ser filho à esquerda do nodo 11
O nodo 7 é colorido de preto, é restaurada a propriedade 4
e nenhuma outra é violada

Remoção com ponteiro para o pai

```
RB-DELETE(T, z)
if esquerda[z] = nil[T] or direita[z] = nil[T]
    then y ← z                          - se z só tem um filho ou nenhum, y = z
    else y ← TREE-SUCCESSOR(z)          - senão procura filho de z
if esquerda[y] ≠ nil[T]                   para substituição
    then x ← esquerda[y]                - se y tem um filho a esq.
    else x ← direita[y]                   ou a dir, armazena em x
p[x] ← p[y]                             - atualiza pai de x como pai de y
if p[y] = nil[T]
    then raiz[T] ← x                    Se pai de y é null, x vira raiz
    else if y = esquerda[p[y]]
        then esquerda[p[y]] ← x         - atualiza filho do pai y
        else direita[p[y]] ← x            como sendo x
if y ≠ z
    then chave[z] ← chave[y]
        copia dados satélite de y em z
if cor[y] = PRETO
    then RB-DELETE-FIXUP(T, x)
return y
```

Verifica se alguma propriedade da árvore foi violada e concerta

- Remoção semelhante a árvore binária! Verifica se z é “folha” (aponta para Null), se z tem apenas um filho ou se z tem dois filhos. Nesse caso, substitui o nó removido pelo nó mais a direita na subarvore a esq. ou nó mais a esq. na subarvore a direita.

- Após extrair um nó, **se ele era preto**, chama um procedimento auxiliar que muda as cores e executa rotações para restaurar as propriedades vermelho-preto.

Remoção: quais propriedades podem ser violadas?

1. Todo nó é **vermelho** ou **preto**.
2. A raiz é **preta**.
3. Toda folha é **preta (NULL)**.
4. Se um nó é **vermelho**, então ambos os seus **filhos são pretos**.
5. Para cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o **mesmo número de nós pretos**.

Remoção: quais propriedades podem ser violadas?

- **Se o nó y extraído era preto**:
- 2 -> se y era raiz e um filho vermelho x de y se torna a nova raiz.
- 4 -> se tanto x quanto p[y] (que agora é p[x]) eram vermelhos
- 5 -> a remoção de y faz qualquer caminho que o continha ter um nó **preto a menos**. Podemos corrigir isso dizendo que **x** tem um "preto extra" , isto é, adicionar 1 a contagem de nós pretos em qualquer caminho que contenha x
- 1 -> quando extraímos o nó preto y, "empurramos" essa característica para seu filho, porém agora **x** não é vermelho nem preto

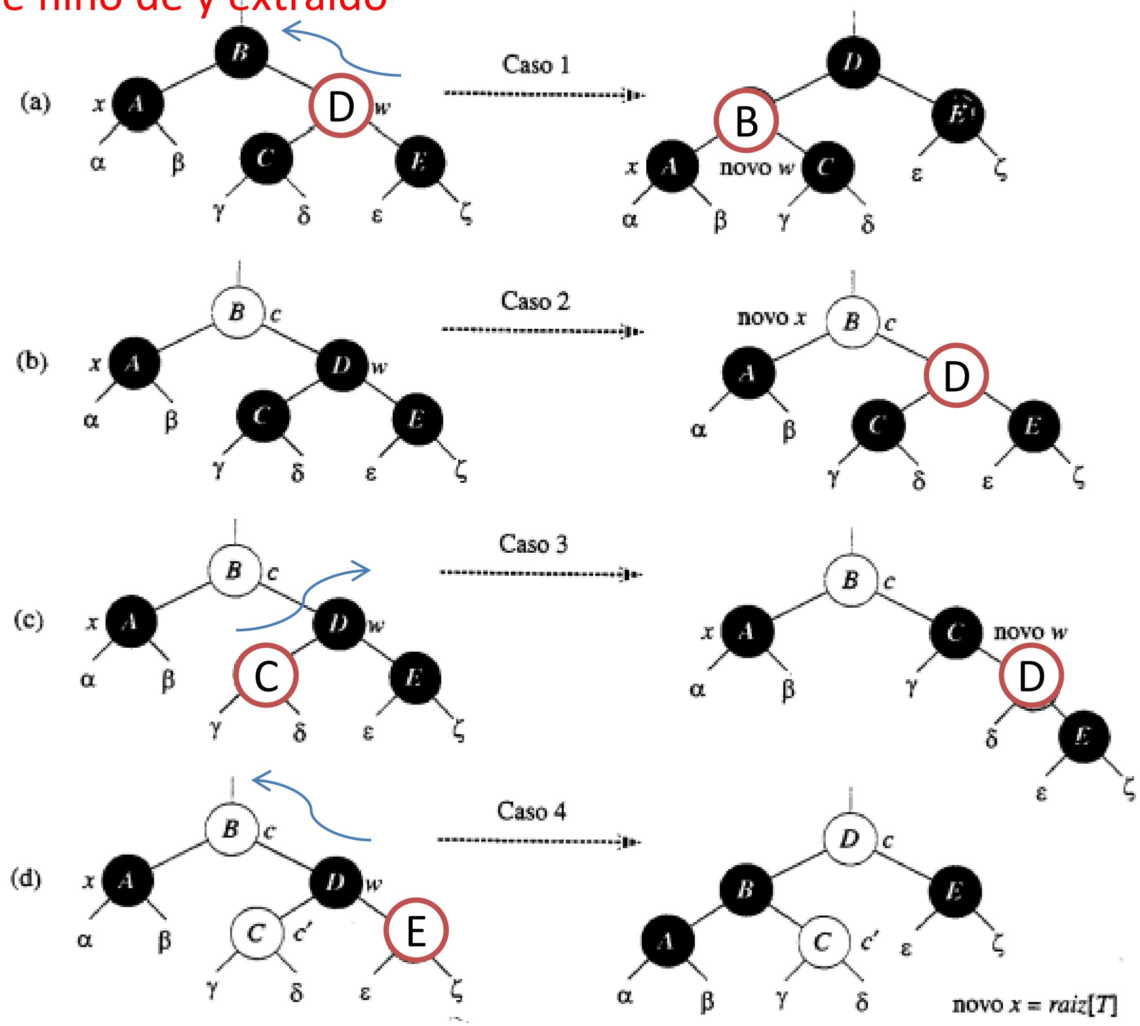
Remoção

- O procedimento de remoção deve restaurar as propriedades 1, 2 e 4.

- O objetivo é mover o preto extra para cima na árvore até:
  - **1: x apontar para um nó vermelho e preto, em cujo caso colore-se x de preto.**
  - **2: x apontar para a raiz, e nesse caso o preto extra pode ser removido**
  - **3: executar operações adequadas de rotação e de nova coloração.**

Remoção: 4 casos para tratar

x é filho de y extraído



- O **caso 1** é transformado no caso 2, 3 ou 4 pela troca das cores dos nós B e D e uma rotação a esquerda

- No **caso 2**, o “preto extra” (ponteiro x) é movido para cima na árvore, colore-se o nó D de vermelho e define-se x apontando para B. Colore-se x de preto.

- O **caso 3** é transformado no caso 4 pela troca das cores dos nós C e D e uma rotação a direita.

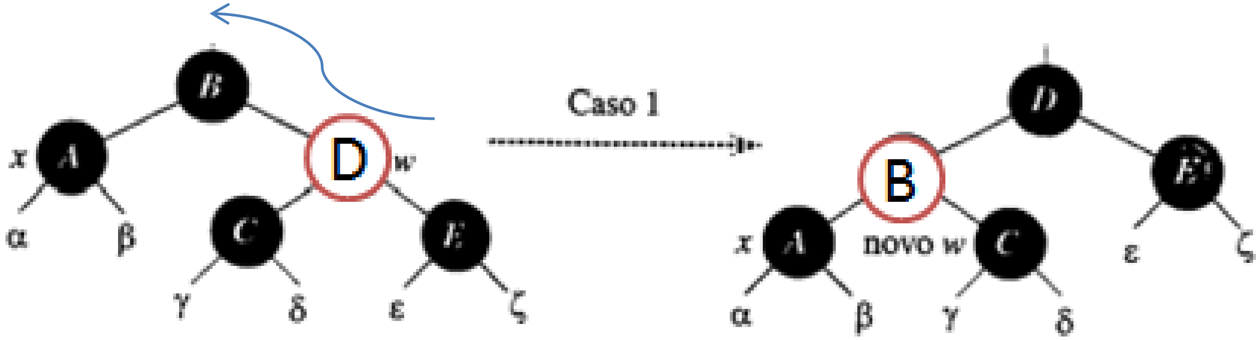
- O **caso 4**, o “preto extra” (ponteiro x) pode ser removido mudando-se algumas cores e uma rotação a esquerda.

Caso 1: o irmão w de x é vermelho

- Como w deve ter filhos pretos, podemos trocar as cores de w e p[x], depois executar uma **rotação a esquerda sobre p[x]** sem violar as propriedades vermelho-preto.

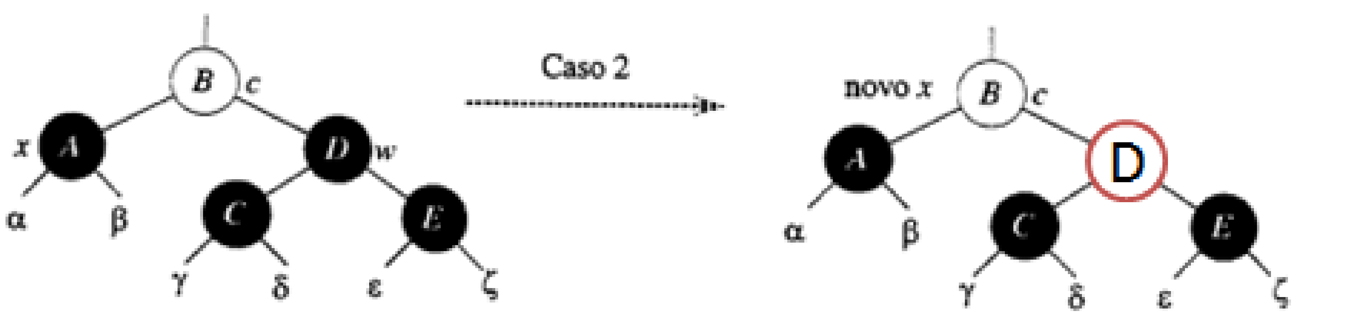
- O novo irmão de x, um dos filhos de w antes da rotação, agora é preto e, convertemos o caso no caso 2, 3 ou 4.

- Os casos 2, 3 e 4 ocorrem quando o nó w é preto, se distinguem pelas cores dos filhos de w.



Caso 2: o irmão w de x é **preto**, e ambos os filhos de w são **pretos**

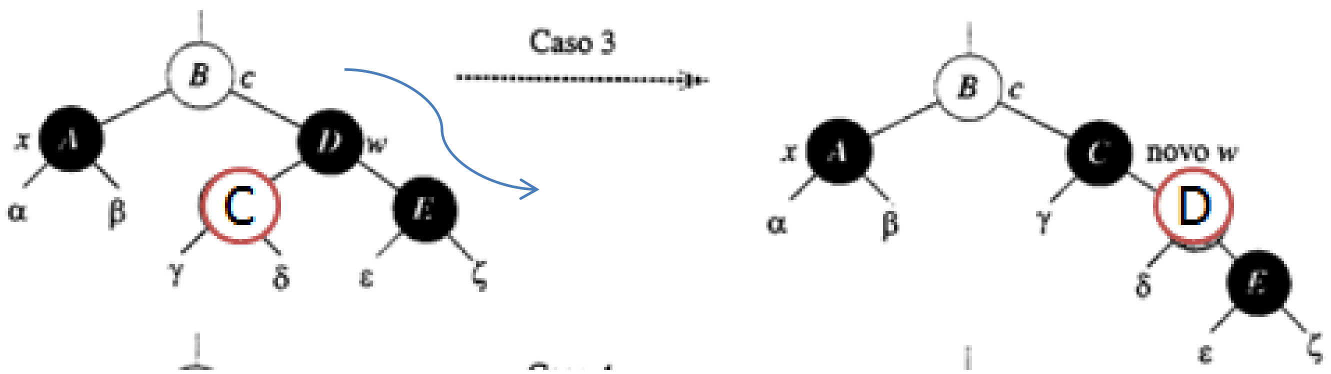
- Como w e seus filhos são pretos, retiramos um preto de x e de w, deixando x com apenas um preto e w vermelho.
- Para compensar, a remoção de um preto de x e de w, adiciona-se um preto extra a p[x], que era originalmente vermelho ou preto.
- Fazemos isso repetindo o loop while com p[x] como o novo nó x.



Caso 3: o irmão w de x é **preto**, o filho da esquerda de w é vermelho e o filho da direita de w é **preto**

- Podemos alterar as cores de w e de seu filho da esquerda, depois de executar uma **rotação a direita sobre w** sem violar as propriedades vermelho-preto.

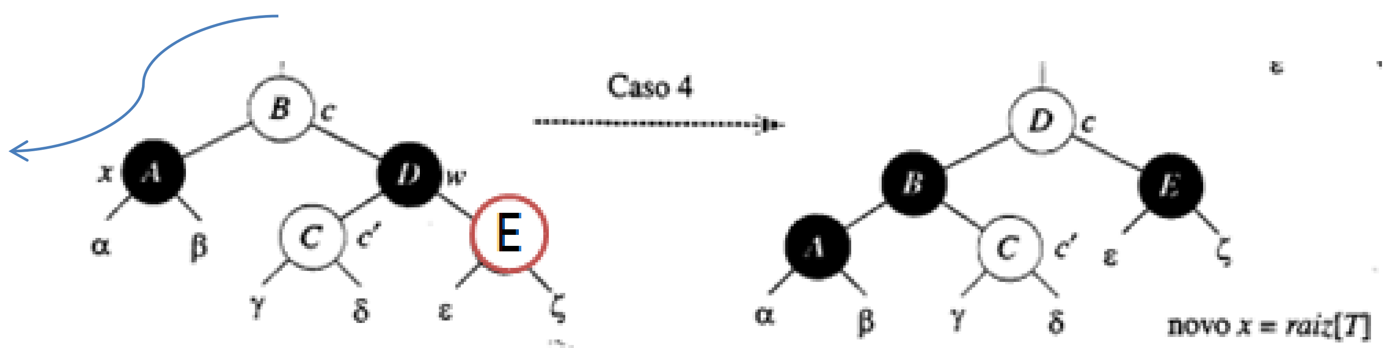
- O novo irmão w de x é agora um nó preto com um filho da direita vermelho, e assim transforma-se o caso 3 no 4.



Caso 4: o irmão w de x é **preto**, e o filho da direita de w é vermelho

- Fazendo algumas mudanças de cores e executando uma **rotação a esquerda sobre p[x]**, podemos remover o preto extra em x, tornando-o unicamente preto, sem violar as propriedades vermelho-preto.

- A **definição de x como raiz faz o loop while se encerrar** ao testar a condição de loop.



Remoção

```
RB-DELETE-FIXUP(T, x)
while x ≠ raiz[T] e cor[x] = PRETO
    do if x = esquerda[p[x]]
        then w ← direita[p[x]]
            if cor[w] = VERMELHO
              then cor[w] ← PRETO                          ▷ Caso 1
                   cor[p[x]] ← VERMELHO                     ▷ Caso 1
                   LEFT-ROTATE(T, p[x])                     ▷ Caso 1
                   w ← direita[p[x]]                        ▷ Caso 1
            if cor[esquerda[w]] = PRETO e cor[direita[w]] = PRETO
              then cor[w] ← VERMELHO                        ▷ Caso 2
                   x ← p[x]                                 ▷ Caso 2
              else if cor[direita[w]] = PRETO
                     then cor[esquerda[w]] ← PRETO          ▷ Caso 3
                          cor[w] ← VERMELHO                 ▷ Caso 3
                          RIGHT-ROTATE(T, w)                ▷ Caso 3
                          w ← direita[p[x]]                 ▷ Caso 3
                     cor[w] ← cor[p[x]]                     ▷ Caso 4
                     cor[p[x]] ← PRETO                      ▷ Caso 4
                     cor[direita[w]] ← PRETO                ▷ Caso 4
                     LEFT-ROTATE(T, p[x])                   ▷ Caso 4
                     x ← raiz[T]                            ▷ Caso 4
        else (igual à cláusula then com "direita" e "esquerda" trocadas)
cor[x] ← PRETO
```

- enquanto x não é raiz e tem cor preta

- encontra w, o irmão de x

- se o irmão w é vermelho

- pinta w de preto
- pinta pai de x de vermelho
-faz rotação a esq. sobre pai de x
- filho a dir de w passa a ser p[x]

-Se ambos os filhos de w são pretos
-Pinta w de vermelho
-Passa o ponteiro de x para o pai

-Se o filho dir de w é preto
-Pinta filho esq de preto
-Pinta w de vermelho
-Faz rotação a dir. sobre w
-Novo w passa ser dir. de p[x]

-w recebe cor do pai de x
-Pai x recebe cor preto
-Filho dir de w recebe preto
- rotação a esq. sobre p[x]

- finaliza pintando x de preto

Análise de complexidade

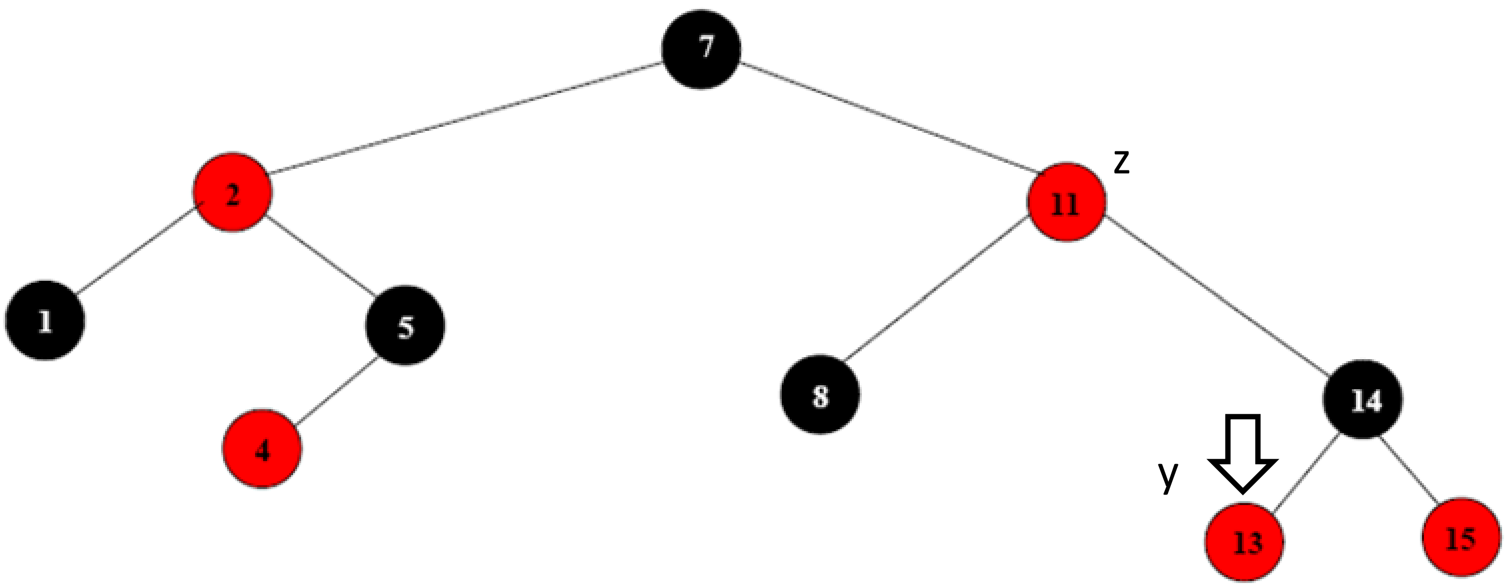
- Como a altura da árvore vermelho-preto de n nós é O(log n), o custo da remoção sem o procedimento RB-DELETE-FIXUP é O(log n).

- Dentro do RB-DELETE-FIXUP cada um dos casos 1, 3 e 4 termina depois de fazer mudanças de cores e no máximo 3 rotações.
- O caso 2 é o único em que o loop while pode ser repetido e o ponteiro x se move para cima na árvore no máximo log(n) vezes sem executar nenhuma rotação.

- Assim RB-DELETE-FIXUP demora O(log n) e faz no max 3 rotações. Sendo seu custo total O(log n).

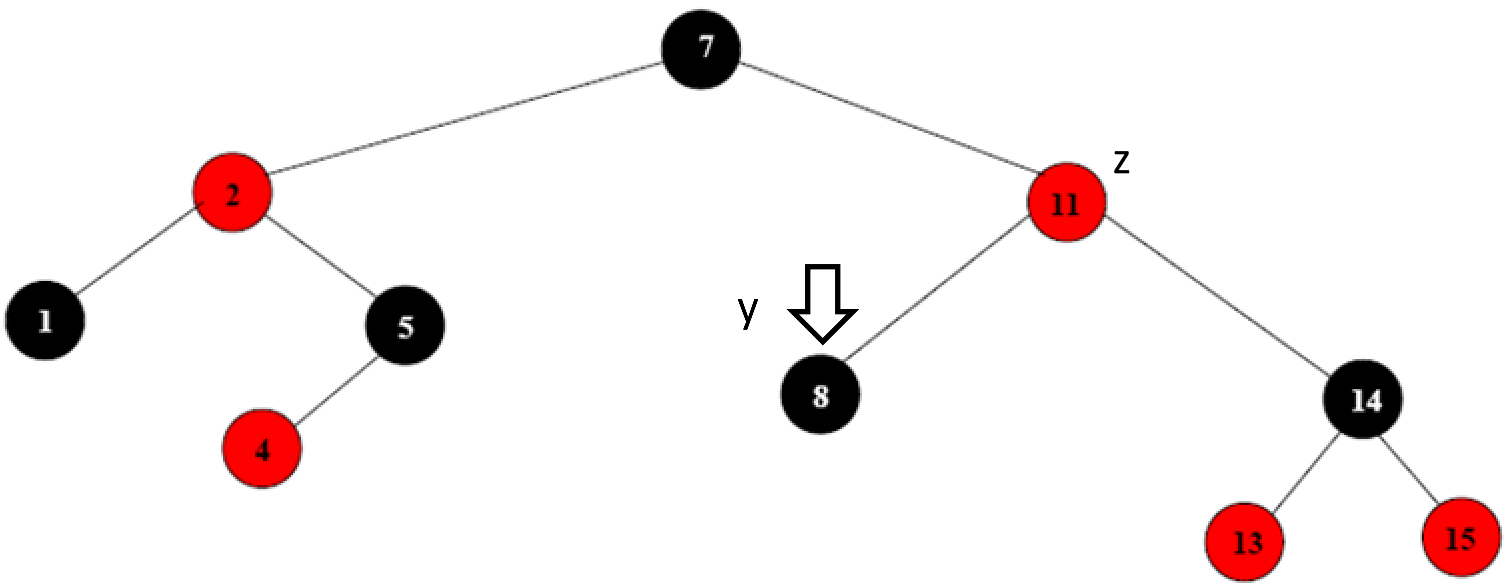
Exemplo remoção

- Remoção de z = 11, busca elemento na árvore para substituir. Se y é vermelho, não precisa fazer nenhuma alteração.



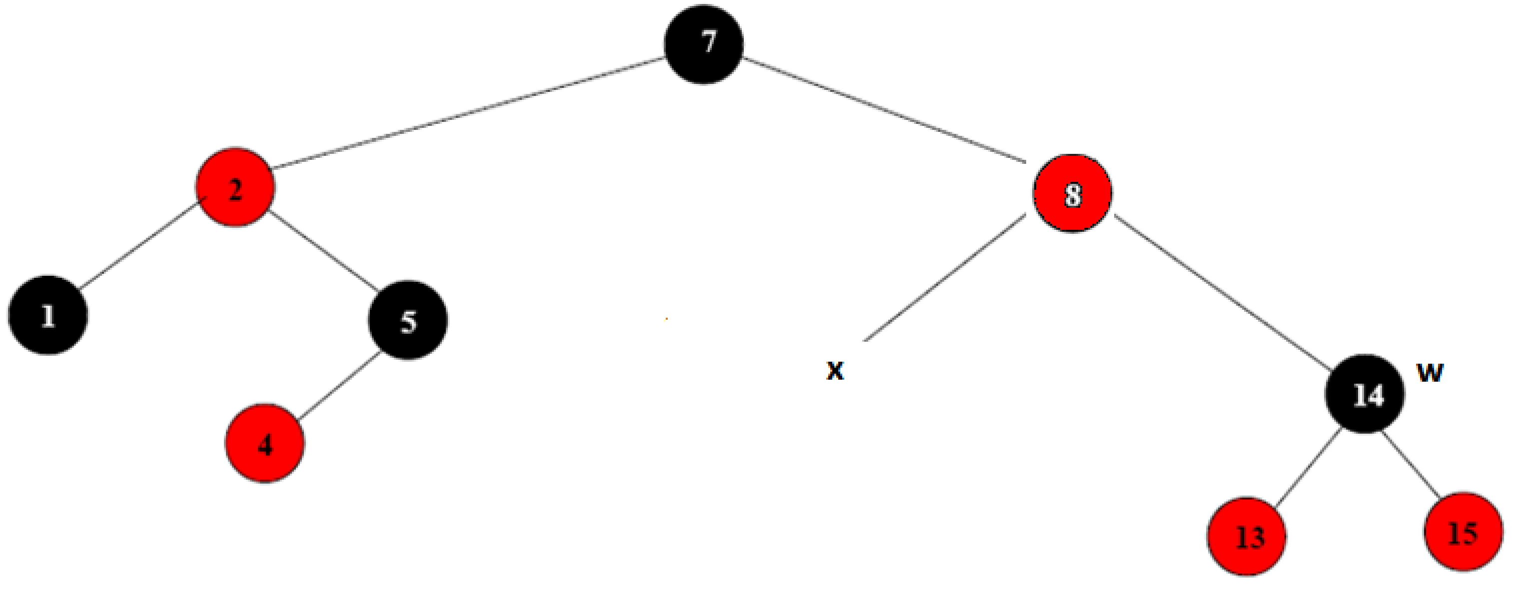
Exemplo remoção

- Remoção de z = 11, busca elemento na árvore para substituir. Se y é preto, precisa restaurar as propriedades da árvore.



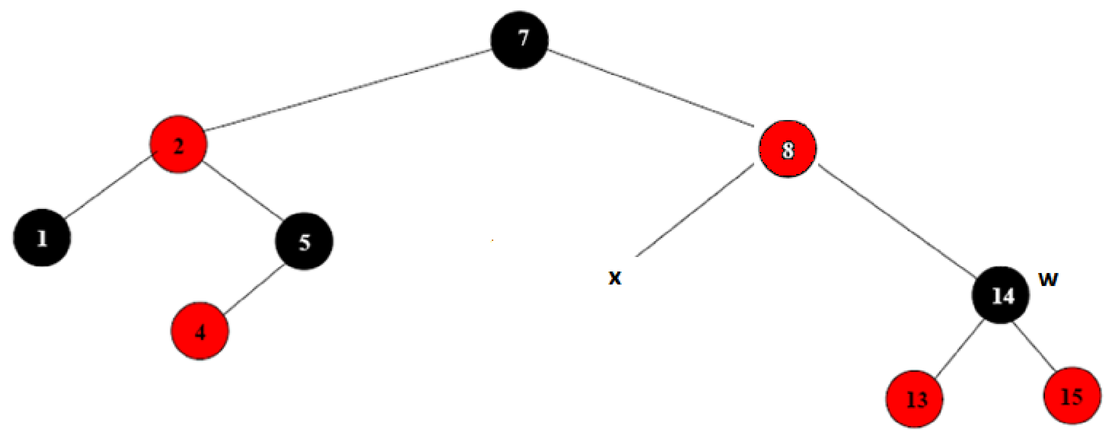
Exemplo remoção

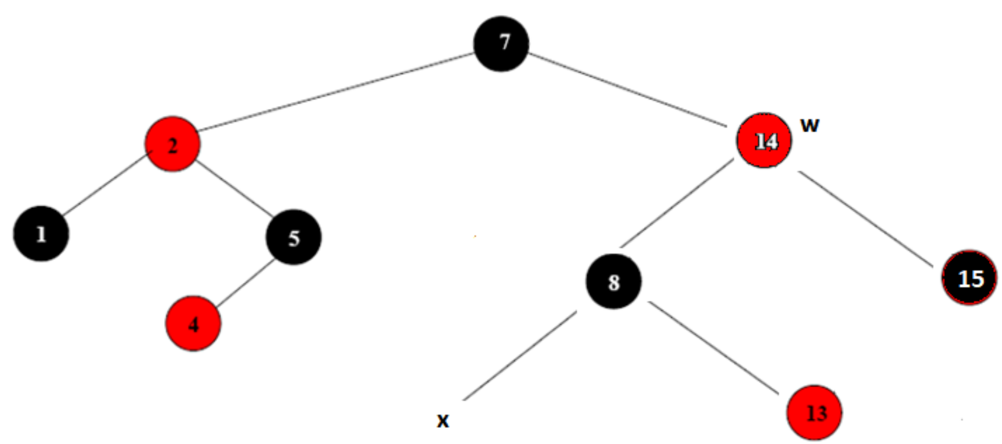
- Caso 4: o irmão w de x é preto e o filho da direita de w é vermelho.



Exemplo remoção

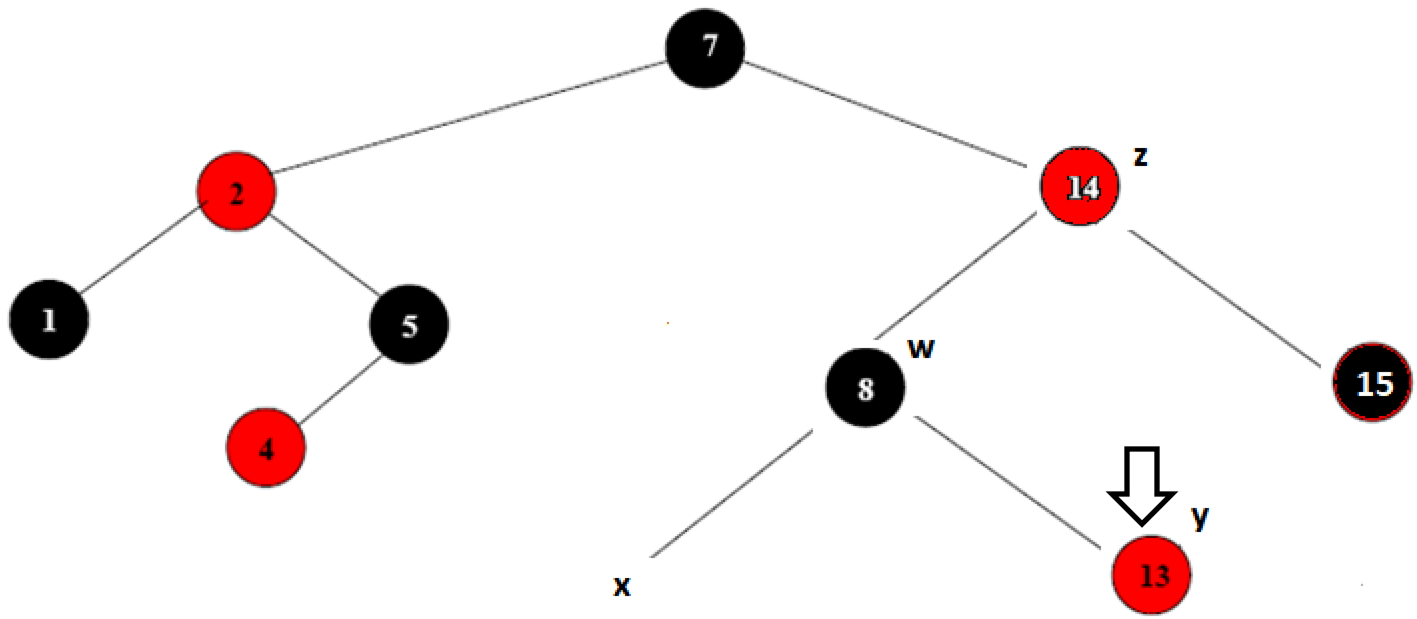
- Caso 4:
- cor[w] = cor [p[x]]
- cor[p[x]] = preto
- cor[dir[w]] = preto
- Rotação a esquerda





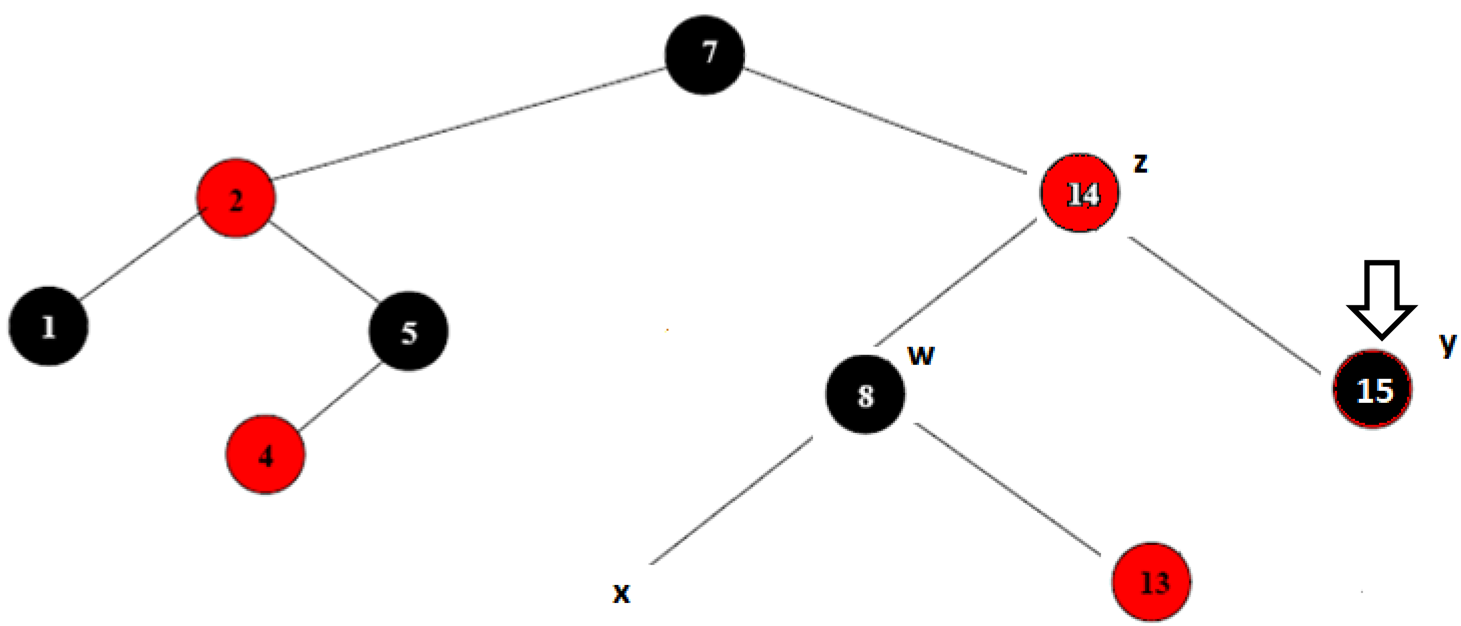
Exemplo remoção

- Remoção de z = 14, busca elemento na árvore para substituir. Se y é vermelho, não precisa fazer nenhuma alteração.



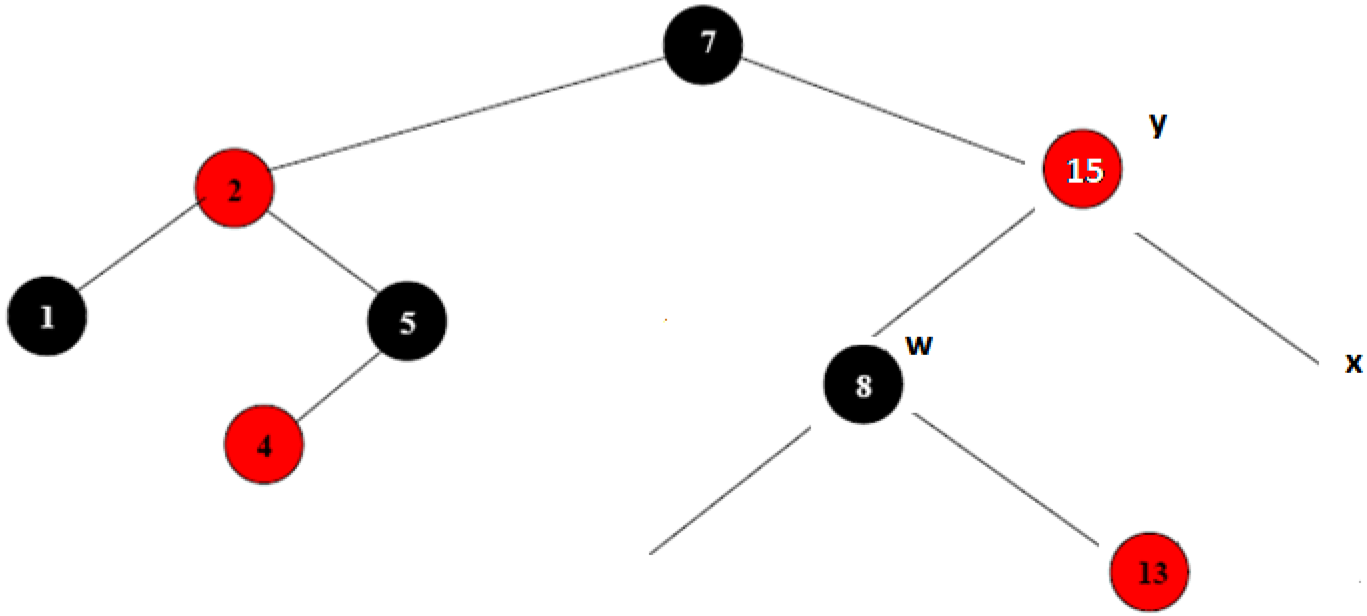
Exemplo remoção

- Remoção de z = 14, busca elemento na árvore para substituir. Se y é preto, precisa restaurar as propriedades da árvore.



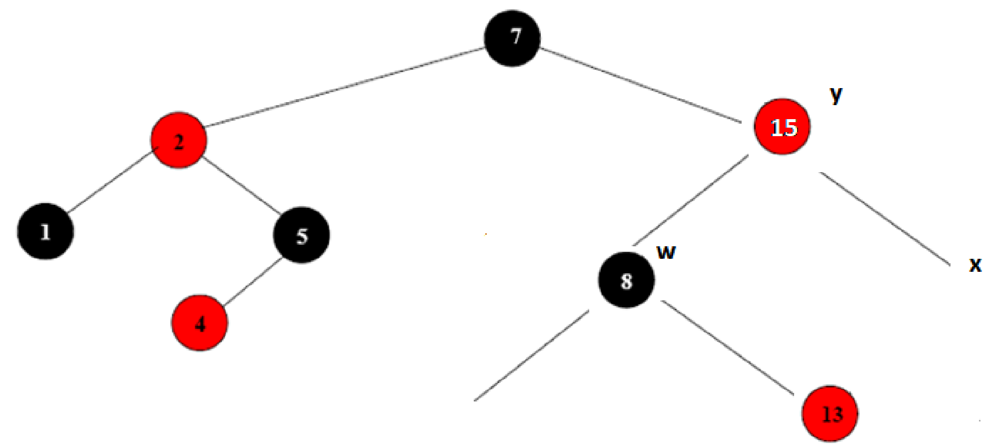
Exemplo remoção

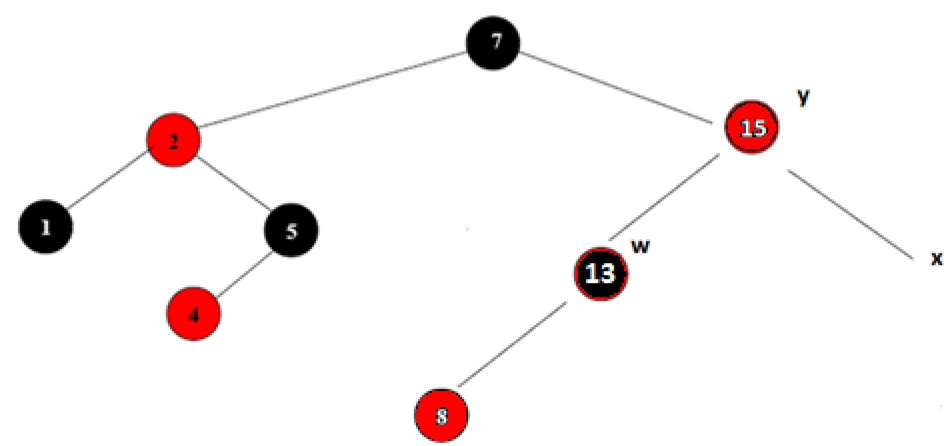
- Caso 3 e 4: o irmão w de x é preto e o filho da direita de w é vermelho.



Exemplo remoção

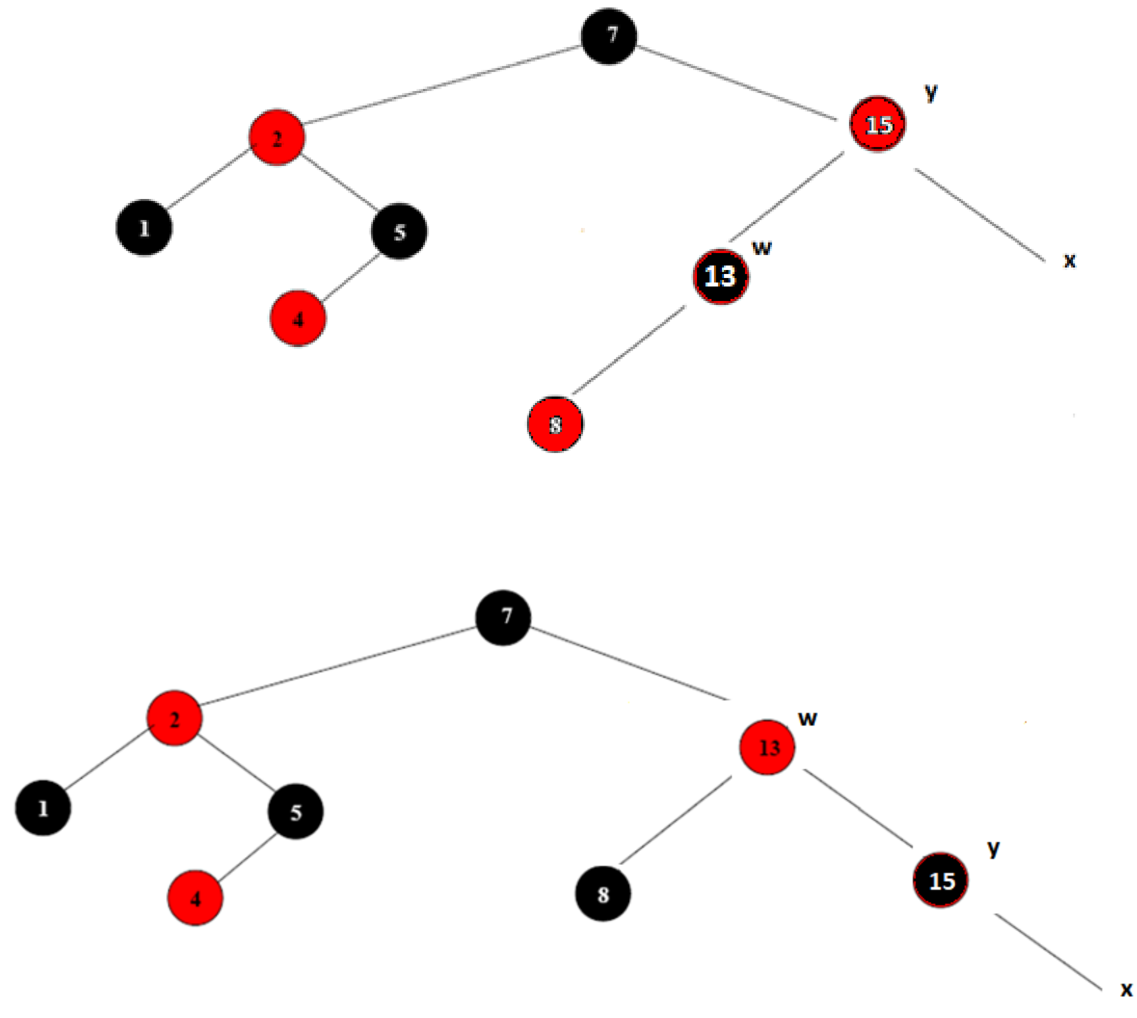
- Caso 3:
- Cor[dir[w]] = preto
- Cor [w] = vermelho
- Rotação a esquerda





Exemplo remoção

- Caso 4:
- cor[w] = cor [p[x]]
- cor[p[x]] = preto
- cor[dir[w]] = preto
- Rotação a esquerda



Exercícios

1. Responda:
   a) Sempre que inserimos um nó em uma árvore vermelho-preto sua cor é vermelho (linha 16 do algoritmo RB-INSERT). Porque não optar por definir sua cor como preta?

   b) Após a execução do RB-DELETE-FIXUP a raiz sempre será preta? Por que?

   c) Desenhe o passo-a-passo com inserção numa árvore de pesquisa **vermelho-preto** sobre as chaves {41 – 38 – 31 – 12 – 19 – 8 – 50 – 1 – 100 – 101 }.

   d) Desenhe o passo-a-passo com remoção dos elementos 50 e 8 na árvore anterior.

   e) Pesquise/implemente um algoritmo de inserção em árvore vermelho-preto e compare o tempo de inserir 10,000 elementos e buscar o menor/maior elemento com os tempos computados na aula anterior para vetor, arv. Binária, AVL.

2. Resolver URI 1861, 1229

3. Fazer um resumo sobre o andamento do seu trabalho proposto na disciplina.

Nunca é tarde demais para ser aquilo que se desejou ser.

Mary Ann Evans