Governo Federal

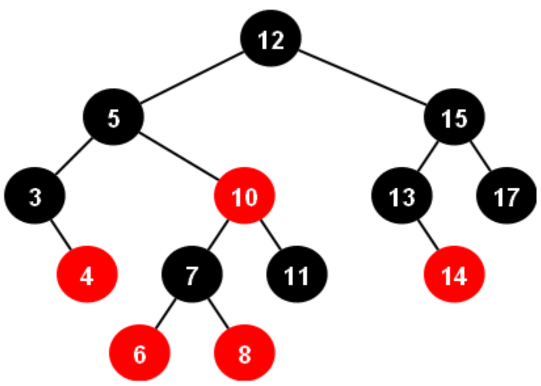
Ministério da Educação

deINF
departamentodeInformática

Universidade Federal do Maranhão

A Universidade que Cresce com Inovação e Inclusão Social

# Árvore Rubro-Negra cont...




## Estrutura de Dados II

Prof. João Dallyson

Email: Joao.dallyson@ufma.br

Referência: Notas de aula Prof. Rafael Fernandes DAI/IFMA

# Propriedades

1. Todo nó é vermelho ou **preto**
2. A **raiz** da árvore necessariamente é preta
3. Toda **folha** null é preta
4. Nós vermelhos que não seja folhas possuem apenas **filhos** pretos
5. Todos os caminhos a partir da raiz até suas folhas passam pela mesma quantidade de nós pretos
6. Não podem existir dois nós vermelhos consecutivos

# Algoritmo de Rotação

```
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
p[y] ← p[x]                       ▷ Liga o pai de x a y.
if p[x] = nil[T]
   then raiz[T] ← y
   else if x = esquerda[p[x]]
           then esquerda [p[x]] ← y
           else direita[p[x]] ← y
esquerda[y] ← x                   ▷ Coloca x à esquerda de y.
p[x] ← y
```

# Algoritmo de Inserção

- **Para inserir um elemento em uma árvore rubro-negra:**
  - Inicialmente, um nodo Z é inserido como se a árvore fosse uma árvore de pesquisa binária comum.
  - Colori-se Z de vermelho.
  - Invoca-se a função RB-INSERT-FIXUP para
  - recolorir e executar rotações na árvore

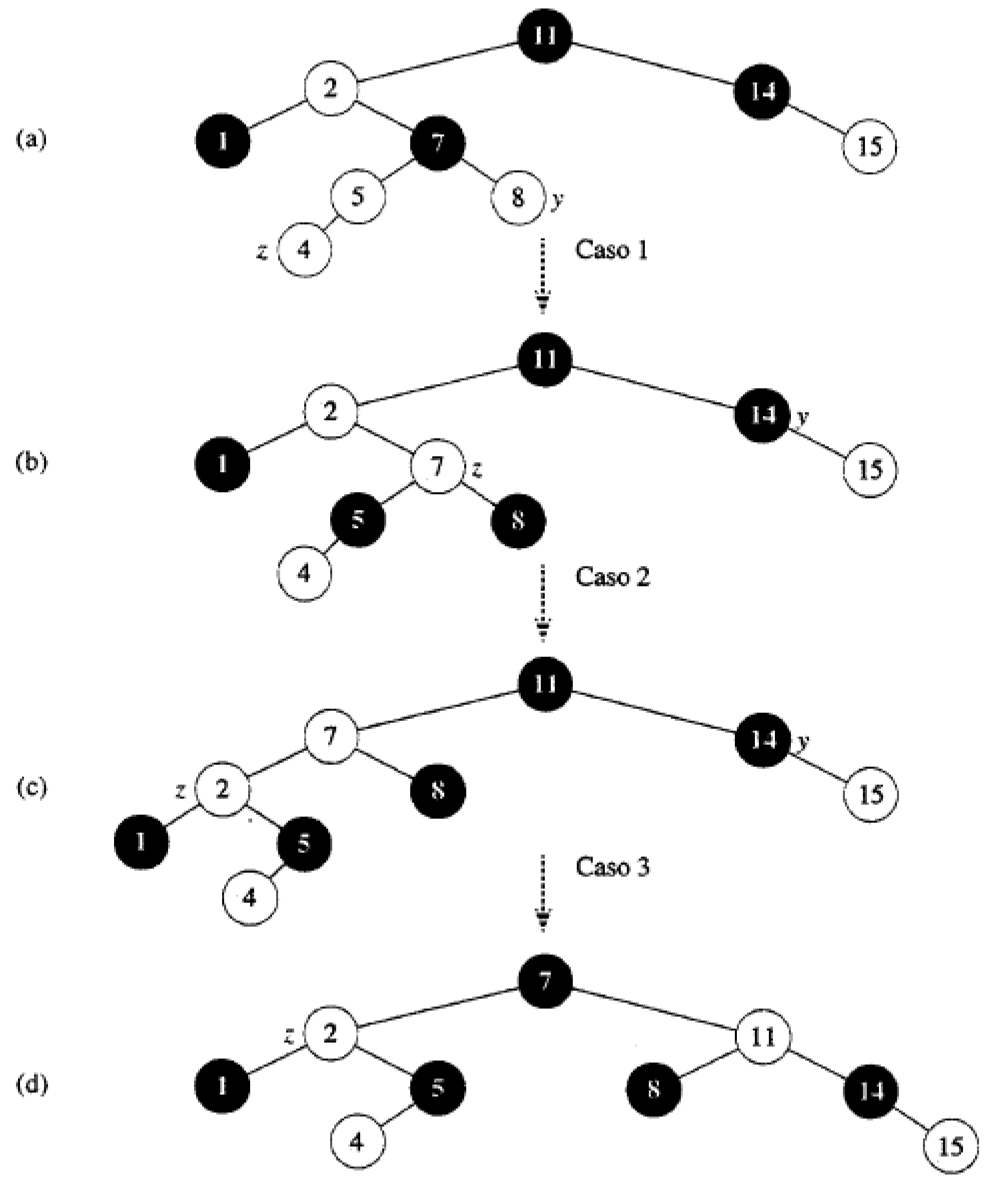
# Algoritmo de Inserção

```
RB-INSERT(T, z)
y ← nil[T]
x ← raiz[T]
while x ≠ nil[T]
  do y ← x
     if chave[z] < chave[x]
        then x ← esquerda[x]
        else x ← direita[x]
p[z] ← y
if y = nil[T]
  then raiz[T] ← z
  else if chave[z] < chave[y]
          then esquerda[y] ← z
          else direita[y] ← z
esquerda[z] ← nil[T]
direita[z] ← nil[T]
cor[z] ← VERMELHO
RB-INSERT-FIXUP(T, z)
```

# Algoritmo de Inserção

```
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
```

# Inserção


(a)
11
2
14
1
7
15
5
8
y
z
4
Caso 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Caso 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Caso 3
(d)
7
z
2
11
1
5
8
14
4
15

# Inserção – Entendo o algoritmo

- **Passos:**
  - Determinar as violações das regras da árvore vermelho-preto com a inserção e coloração do novo nodo z
  - Examinar a meta global do loop while, linhas 1 a 15
  - Examinar os três casos que o loop while se divide
- **Quais das propriedades da árvore rubro-negra podem ser violadas?**
  - As propriedade 2 e 4!
    - A raiz deve ser preta.
    - Um nó vermelho não pode ter filho vermelho

# Inserção

- **Inicialização:**
  - Começa-se com uma árvore vermelho-preto sem violações e começamos com a inserção de z.
    - Quando RB-Insert-FixUp é chamado z é o nodo vermelho que foi adicionado
    - Se P[z] é a raiz, este começou preto e não é modificado
    - Se houver violações, são da propriedade 2 ou da propriedade 4 (exclusivamente).

# Inserção

- **Término:**
  - Quando o loop termina, P[z] é preto (propriedade 4)
  - A propriedade 2 é garantida na linha 16.
- **Manutenção:**
  - Só entra-se no loop se P[z] for vermelho, caso contrário, P[z] é preto.

# Remoção

- É um pouco mais complexa do que a inserção
- A remoção em árvores rubro-negras pode ser realizada também com um número logarítmico de operações
- O procedimento é composto de uma etapa de remoção em árvore binária de busca seguido de uma etapa de balanceamento, se as propriedades rubro-negras forem destruídas durante a operação
- **Passos:**
  - Encontre o nó z a ser removido
  - Remova o nó z da mesma forma que em uma árvore binária de pesquisa
  - Ajuste os critérios da árvore rubro-negra

# Remoção

- **Remoção efetiva:**
  - Após as operações de rotação/alteração de cor necessárias, a remoção do nó é efetivamente realizada, restabelecendo-se as propriedades da árvore
- **Remoção preguiçosa**
  - Consiste em apenas marcar um determinado nó como removido, sem efetivamente retirá-lo da árvore

# Remoção

- Como em uma árvore binária, a remoção de um elemento de uma árvore vermelho-preto tem custo em tempo de O(lgn)
- Para Remover um elemento, o processo é bem semelhante a de uma árvore binária:

```
RB-DELETE(T, z)
if esquerda[z] = nil[T] or direita[z] = nil[T]
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if esquerda[y] ≠ nil[T]
    then x ← esquerda[y]
    else x ← direita[y]
p[x] ← p[y]
if p[y] = nil[T]
    then raiz[T] ← x
    else if y = esquerda[p[y]]
        then esquerda[p[y]] ← x
        else direita[p[y]] ← 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
```

# Remoção

- **Observações:**
  - A função RB-Delete-FixUp só é invocada caso Z (o nodo deletado) seja preto.
  - Caso o nodo deletado seja vermelho as propriedades vermelho-preto ficam inalteradas, visto que:
    - Nenhuma altura preta foi alterada
    - Nenhum nodo vermelho ficou adjacente
    - Se Z é vermelho, então Z não é Raiz: Assim a Raiz continua preto.

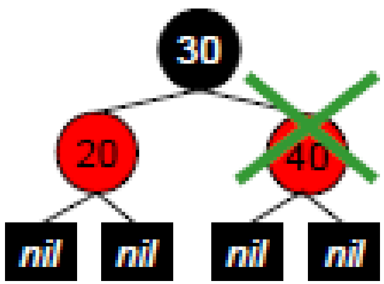
# Remoção

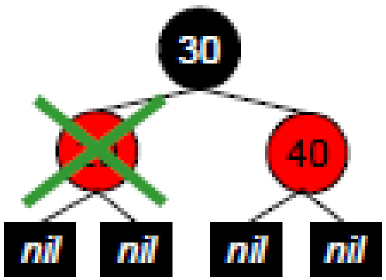
- **Remoção nó intermediário**
  - Não há problema porque as cores permanecem iguais
  - Existe apenas a troca de valores
- **Nó Vermelho**
  - Ok! Não altera o balanceamento da árvore
- **Nó Preto**

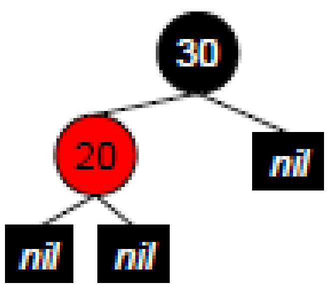
  PROBLEMA

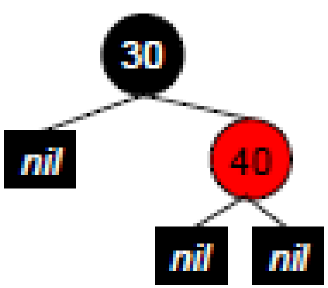
# Exemplo

- **Nó vermelho**



# Remoção

- **Se o nodo y deletado é preto, podem ocorrer três problemas:**
  - Se y era a raiz e um filho vermelho de y se torna a raiz (viola propriedade 2)
  - Se x e P[y] (que agora também é P[x]) eram vermelhos (viola a propriedade 4)
  - A remoção de y faz com que o caminho que continha y tenha agora um nodo preto a menos (viola a propriedade 5)

# Remoção

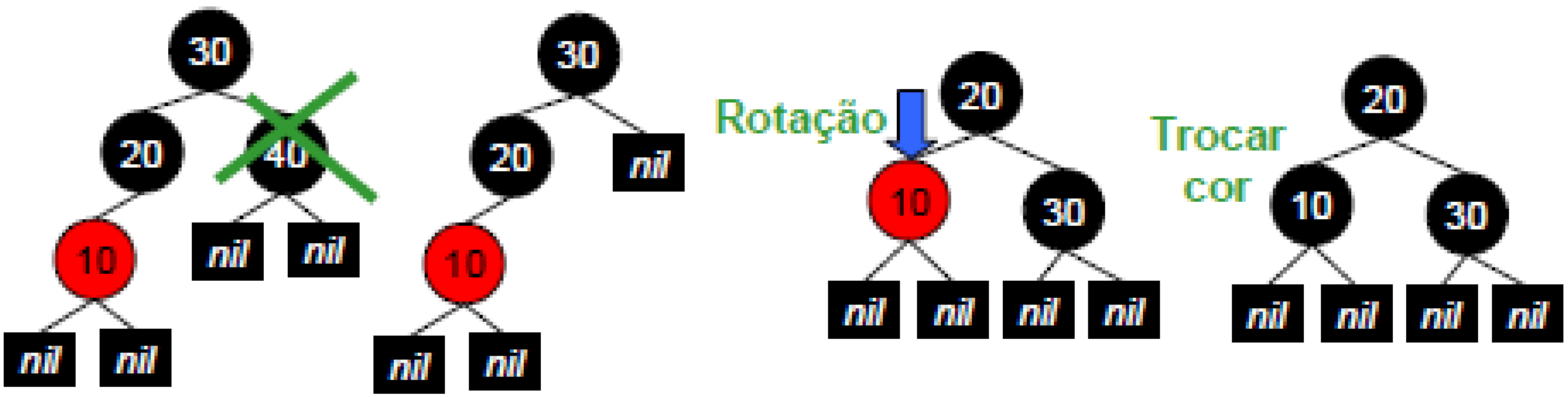
```
RB-DELETE-FIXUP(T, x)
while x ≠ raiz[T] e cor[x] = PRETO
    do if x = esquerda[p[x]]
          then w ← direita[p[x]]
               if cor[w] = VERMELHO
                  then cor[w] ← PRETO                              ▷ Caso 1
                       cor[p[x]] ← VERMELHO                        ▷ 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
```

# Casos

- **Caso 1:**
  - O irmão w de x é vermelho
- **Caso 2:**
  - O irmão w de x é preto e ambos os filhos de w são pretos
- **Caso 3:**
  - O irmão w de x é preto e o filho da esquerda de w é vermelho e o da direita é preto
- **Caso 4**
  - O irmão w de x é preto e o filho da direita de w é vermelho

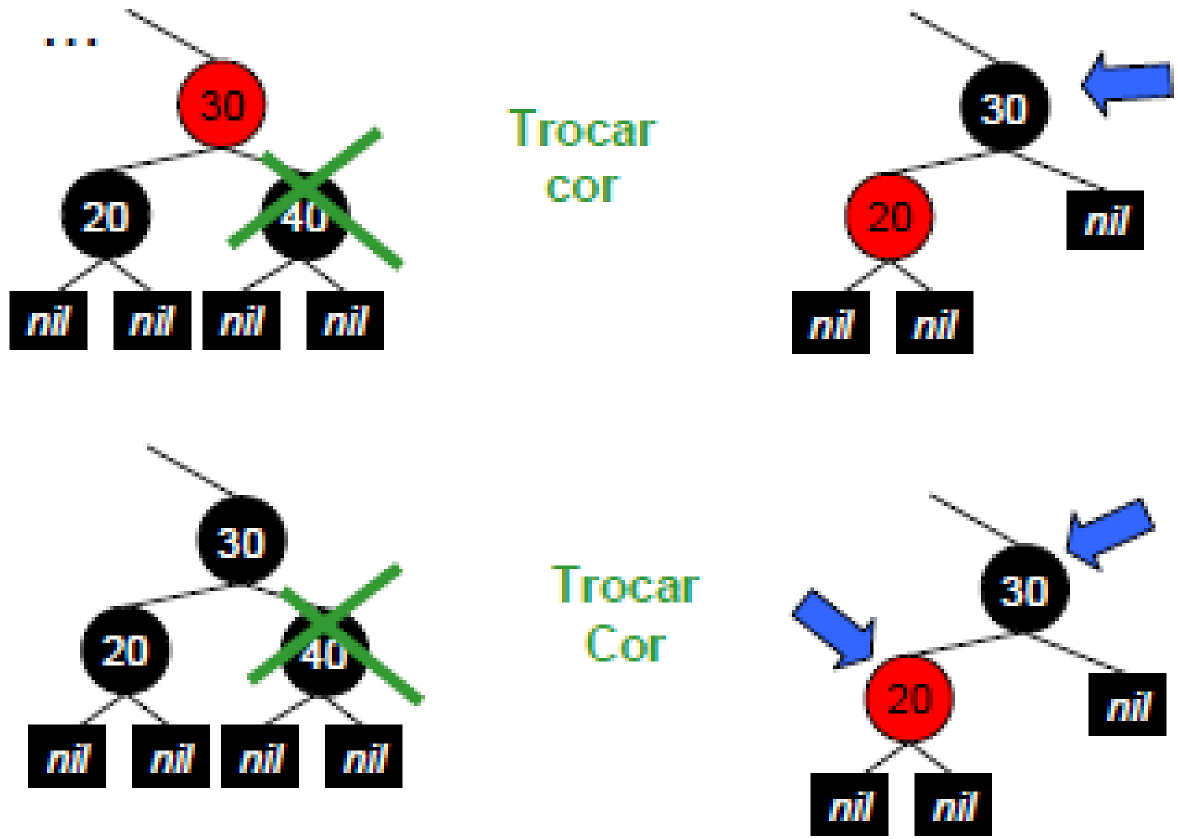
# Exemplos

## 1) Irmão preto – filho vermelho



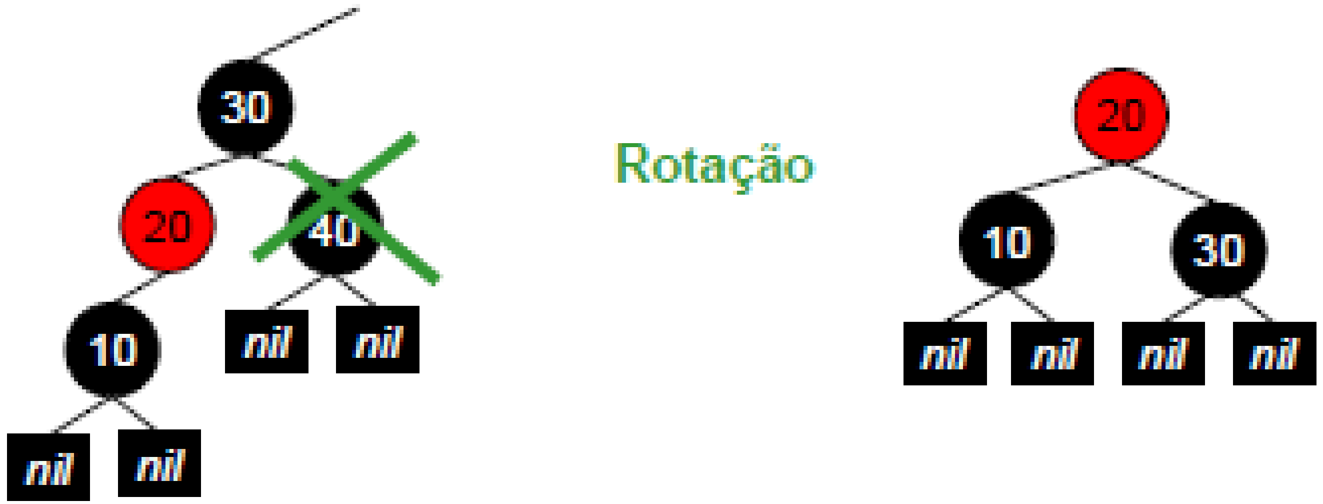
## Exemplos

### 2) Irmão preto – dois filhos pretos



# Exemplos

## 3) Irmão vermelho



# Resumo

- **Arvores Binárias de Busca com bit adicional para representar a cor (PRETA ou VERMELHA).**
- **Cinco propriedades basicas:**
  1. Todo nó é colorido PRETO ou VERMELHO.
  2. A raiz é PRETA.
  3. As folhas (NIL) são PRETAS.
  4. Se um nó é VERMELHO, seus filhos são PRETOS.
  5. Para cada nó, todos os caminhos até as folhas descendentes contem o mesmo número de nós.
- **Lema: Uma Arvore Rubro-Negra com n nós internos tem altura no máximo 2lg(n+1).**
- **Possui todas as operações que uma árvore binária balanceada comum possui.**

# Referências

**Básica**


- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Editora Campus, 2002
- Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Cengage Learning, 2004.
- Notas de aula. Prof. Rafael Fernandes DAI/IFMA
- Notas de aula. Prof. Tiago A. E. Ferreira. DEINFO/UFRPE


**Complementar**


- ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: Algoritmos, análise da complexidade e implementações em Java e C/C++. Pearson Prentice Hall, 2010
- DROZDEK, Adam. Adam Drozdek. Data Structures and Algorithms in Java. 2. Cengage Learning. 2004. 2. Cengage Learning. 2004
- GOODRICH, Michael T. Estruturas de dados e algoritmos em java. 4 ED. Porto Alegre: Bookman, 2007. 600.
- SKIENA, Steven S.. **The Algorithm Design Manual**. 2. Springer-Verlag. 2008

# Perguntas....