Árvores

Árvores *Red-Black* – Parte I: Definição e Inserção

Prof. Edson Alves - UnB/FGA

2018

Sumário

Definição

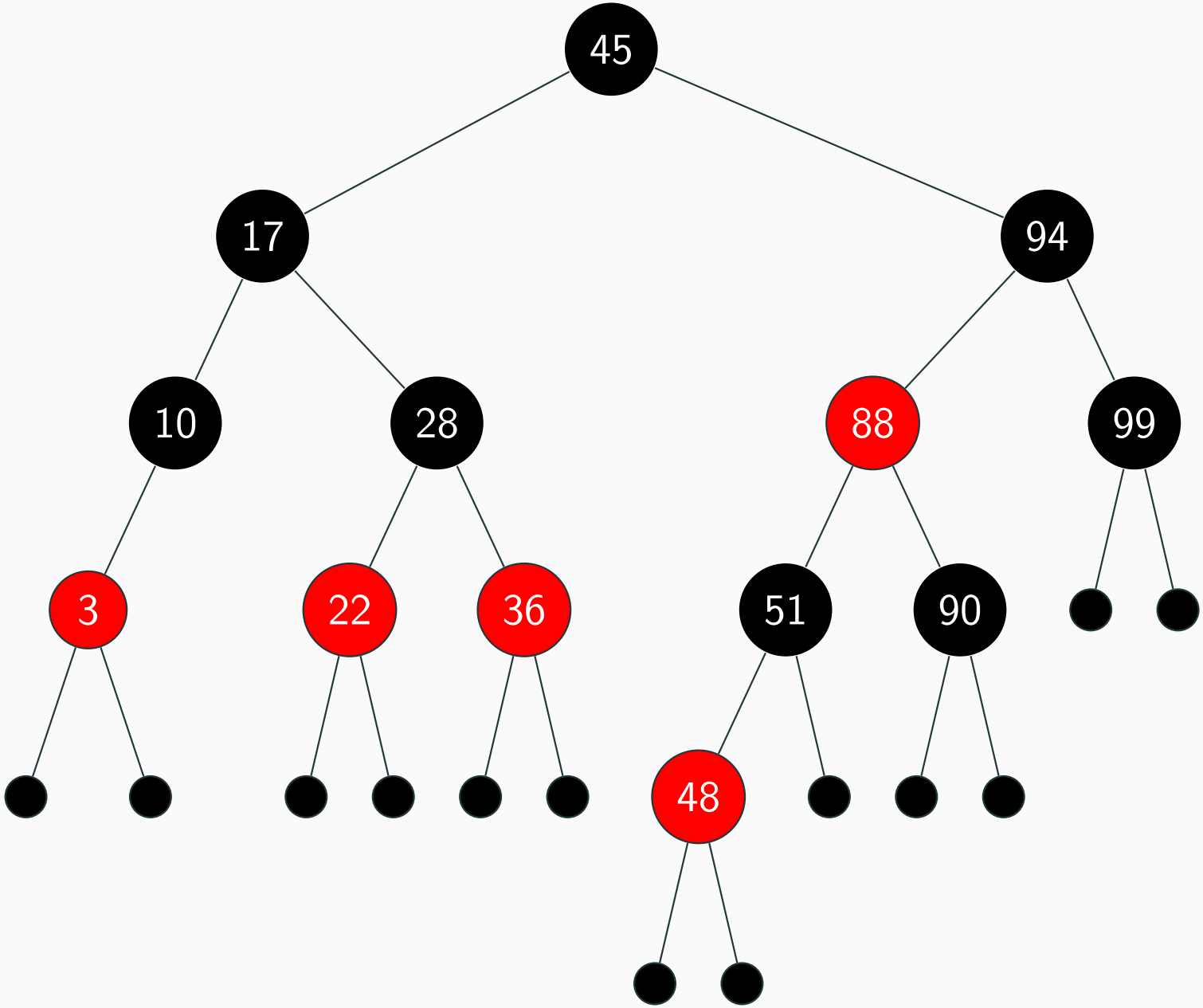
Árvores Red-Black

- Uma árvore *red-black* é uma árvore binária de busca auto-balanceável
- Foi proposta em 1978 por Leonidas J. Guibas e Robert Sedgewick
- A cada um de seus nós é atribuída uma cor: vermelha ou preta
- São estabelecidas 5 propriedades que relacionam os nós e suas cores
- Estas propriedades garantem que a altura h da árvore seja proporcional a $\log N$, onde N é o tamanho da árvore
- As rotinas de inserção e remoção devem preservar as propriedades das árvores *red-black* e, consequentemente, o balanceamento de sua altura

Propriedades de uma árvore red-black

1. Cada nó ou é vermelho ou é preto
2. A raiz é tem a cor preta
3. Todas as folhas são nulas e tem a cor preta
4. Se um nó é vermelho, então todos os seus filhos são pretos
5. Dado um nó n, todos os caminhos de n até um de seus descendentes nulos tem o mesmo número de nós pretos

Exemplo de árvore red-black



Observações sobre árvores red-black

- Cormen et. al propõem e demonstram o seguinte lema: "*A red-black tree with N internal nodes has height at most* $2\log(N+1)$"
- Este lema garante complexidade $O(\log N)$ para as operações de busca, inserção e remoção
- Como uma árvore *red-black* é uma árvore binária de busca, o algoritmo de busca é idêntico ao utilizado em árvores binárias de busca
- As inserções e remoções devem tratar as possíveis violações às propriedades das árvores *red-black*, de modo que as árvores resultantes sejam efetivamente árvores *red-black*
- Para implementar tais operações, é útil manter um ponteiro para o pai de cada nó
- É útil também implementar funções auxiliares que permitam acessar os ponteiros do avó, do tio e do irmão de um dado nó

Definição de uma árvore red-black em C++

```cpp
#include <bits/stdc++.h>

template<typename T>
class RBTree {
private:
    struct Node {
        T info;
        enum { RED, BLACK } color;
        Node *left, *right, *parent;
    };

    Node *root;

public:
    RBTree() : root(nullptr) {}

```

Funções auxiliares

```
17 private:
18     Node * parent(Node *node)
19     {
20         return node ? node->parent : nullptr;
21     }
22
23     Node * grandparent(Node* node)
24     {
25         return parent(parent(node));
26     }
27
28     Node* sibling(Node* node)
29     {
30         auto p = parent(node);
31         return p ? (node == p->left ? p->right : p->left) : nullptr;
32     }
33
34     Node * uncle(Node* node)
35     {
36         return sibling(parent(node));
37     }
```

Funções auxiliares

```
38
39      void rotate_left(Node *G, Node *P, Node *C)
40      {
41          if (G != nullptr)
42              G->left == P ? G->left = C : G->right = C;
43
44          P->right = C->left;
45          C->left = P;
46      }
47
48      void rotate_right(Node *G, Node *P, Node *C)
49      {
50          if (G != nullptr)
51              G->left == P ? G->left = C : G->right = C;
52
53          P->left = C->right;
54          C->right = P;
55      }
56
```

Inserção

Inserção em árvores red-black

- A inserção em uma árvore *red-black* consiste em duas etapas
- A primeira é a inserção de um nó vermelho, nos mesmos moldes da inserção em uma árvore binária de busca
- A segunda etapa consiste em corrigir possíveis violações às propriedades de uma árvore *red-black*
- Observe que as propriedade 1 e 3 sempre serão verdadeiras
- As demais propriedades podem ser violadas, a depender do local onde a inserção foi realizada
- São quatro cenários possíveis

Rotinas básicas de inserção

```
57 public:
58     void insert(const T& info) {
59         auto node = insert(&root, nullptr, info);
60         restore_properties(node);
61     }
62
63 private:
64     Node * insert(Node **node, Node *parent, const T& info)
65     {
66         if (*node == nullptr) {
67             *node = new Node { info, Node::RED, nullptr, nullptr, parent };
68             return *node;
69         }
70
71         if ((*node)->info == info)
72             return *node;
73         else if (info < (*node)->info)
74             return insert(&(*node)->left, *node, info);
75         else
76             return insert(&(*node)->right, *node, info);
77     }
```

Cenário A: inserção em árvore vazia

- Neste caso, a inserção é trivial, mas a propriedade 1 é violada
- A correção consiste em pintar a raiz com a cor preta
- Este processo adicionará uma unidade ao tamanho de todos os caminhos que partem da raiz às folhas
- Assim, a propriedade 5 ficará preservada
- Segue abaixo uma visualização da inserção da informação 40 em uma árvore vazia:

Restauração das propriedades no cenário A

```
78
79      void restore_properties(Node *node)
80      {
81          if (parent(node) == nullptr)        // Cenário A: node é a raiz
82              node->color = Node::BLACK;
```

Cenário B: o pai do nó inserido é preto

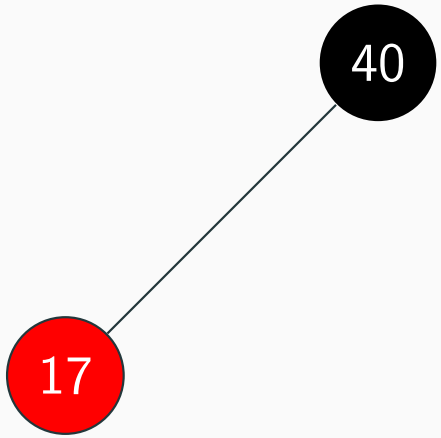
- Como o pai do nó inserido n é preto, não há violação da propriedade 4
- Além disso, como n é vermelho, o caminho da raiz até um de seus filhos mantém o mesmo número de nós pretos que haviam até a posição onde n foi inserido
- Deste modo, não há violação da propriedade 5
- Como n tem um pai preto, ele não é a raiz (pois a raiz não tem pai)
- Assim, não há violação da propriedade 2
- De fato, neste cenário não há necessidade de nenhuma correção após a inserção

Exemplo de inserções no cenário B

Informação a ser inserida: 17

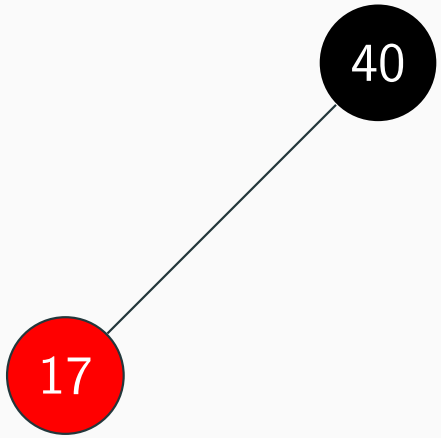
Exemplo de inserções no cenário B

Informação a ser inserida: 17



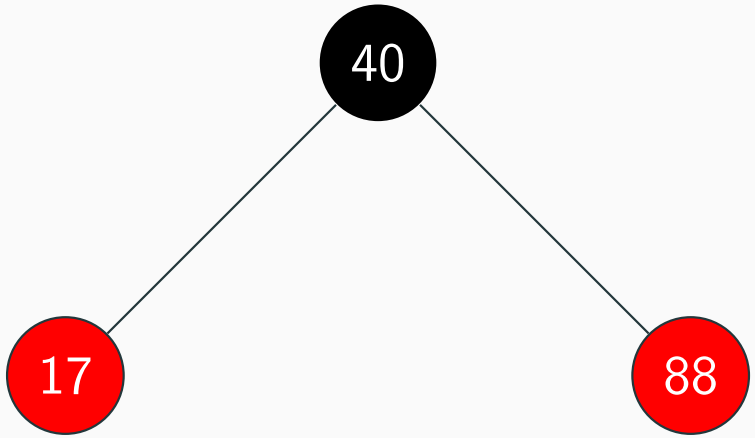
Exemplo de inserções no cenário B

Informação a ser inserida: 88



Exemplo de inserções no cenário B

Informação a ser inserida: 88

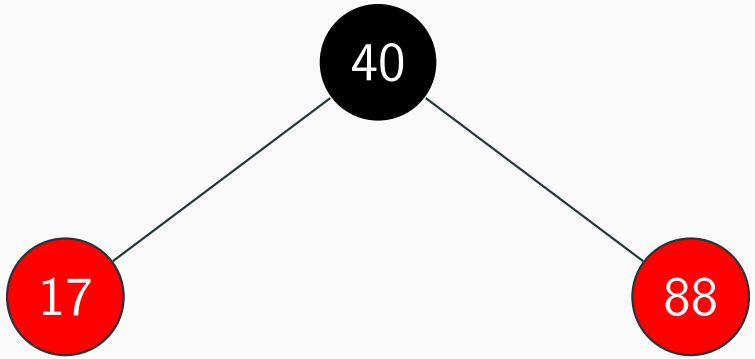


Cenário C: o pai e o tio do nó inserido são vermelhos

- Neste cenário, o pai e o filho do nó inserido n são ambos vermelhos
- Como n também é vermelho, há uma violação da propriedade 4
- Se o avô se tornar vermelho, e o pai e o tio se tornarem pretos, a violação da propriedade 4 é corrigida
- Esta mudança também não viola a propriedade 5, pois o número de nós pretos nos caminhos muda
- Contudo, se o avô for a raiz, a propriedade 2 passa a ser violada
- Caso contrário, pode existir uma violação da propriedade 4, se o bisavô for vermelho
- Para evitar tais violações, é preciso reiniciar a rotina de correção no avô

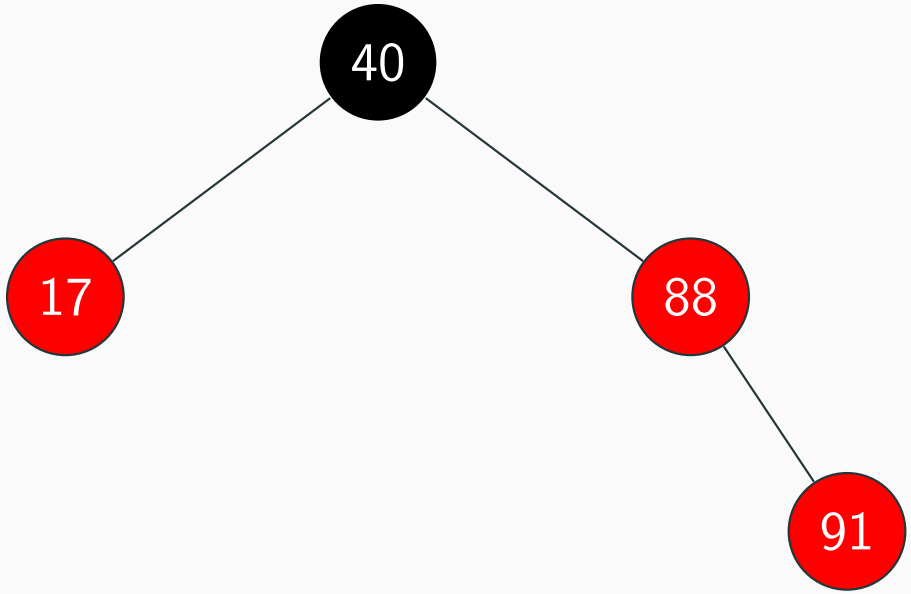
Exemplo de inserções no cenário C

Informação a ser inserida: 91



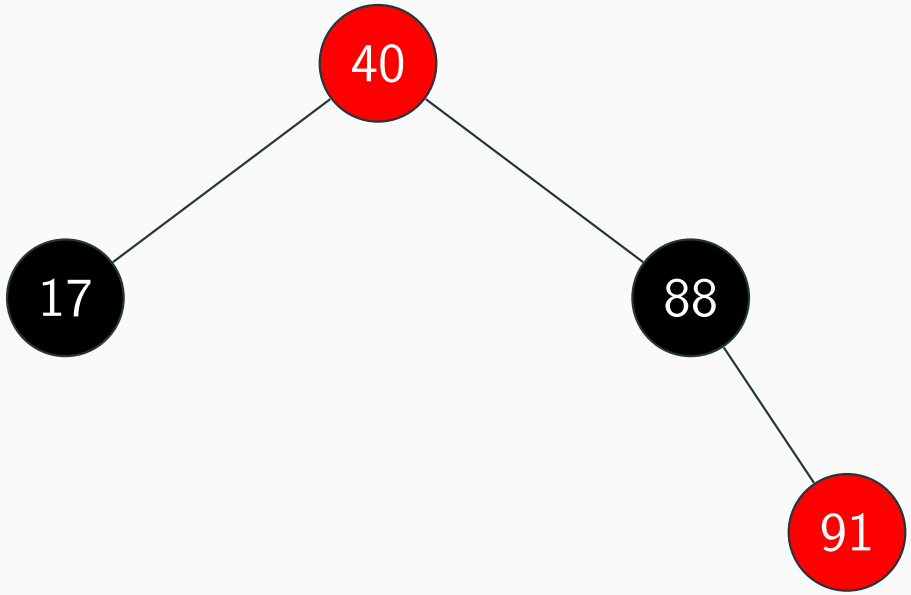
Exemplo de inserções no cenário C

Informação a ser inserida: 91



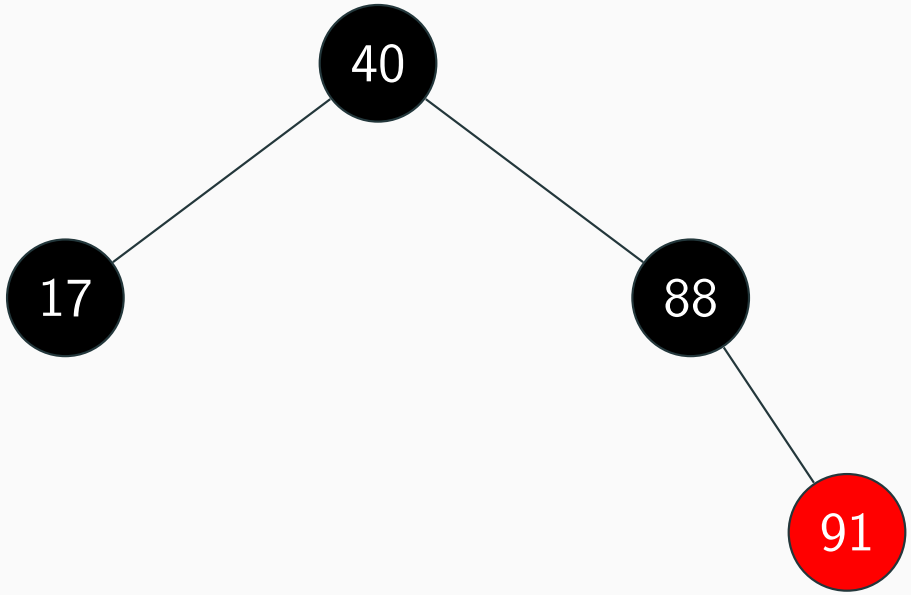
Exemplo de inserções no cenário C

Informação a ser inserida: 91



Exemplo de inserções no cenário C

Informação a ser inserida: 91



Restauração das propriedades nos cenários B e C

```
83          else if (parent(node)->color == Node::BLACK)    // Cenário B
84              return;
85          else if (uncle(node) and uncle(node)->color == Node::RED)
86          {
87              // Cenário C: pai e tio vermelhos
88              parent(node)->color = Node::BLACK;
89              uncle(node)->color = Node::BLACK;
90              grandparent(node)->color = Node::BLACK;
91
92              // Como o pai é vermelho, o avô não é nulo
93              restore_properties(grandparent(node));
```

Referências

1. Red-Black Trees, acesso em 27/03/2019.
2. 8.2 Red-Black Trees, acesso em 27/03/2019.
3. Wikipédia. *Red-Black Tree*, acesso em 27/03/2019.[1]

[1] https://en.wikipedia.org/wiki/Red-black_tree