Técnicas e Análise de Algoritmos

Árvores Binárias e de Busca - Parte 02

Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

Árvores Binária de Busca
Árvores Red-Black

Árvores Red-Black

- Uma árvore red-black (ou árvore rubro-negra) é uma árvore binária de busca **auto-balanceável**
- Cada um de seus nós é atribuído uma cor: vermelho ou preto
- 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
- Inserção e remoção devem preservar as propriedades da árvore

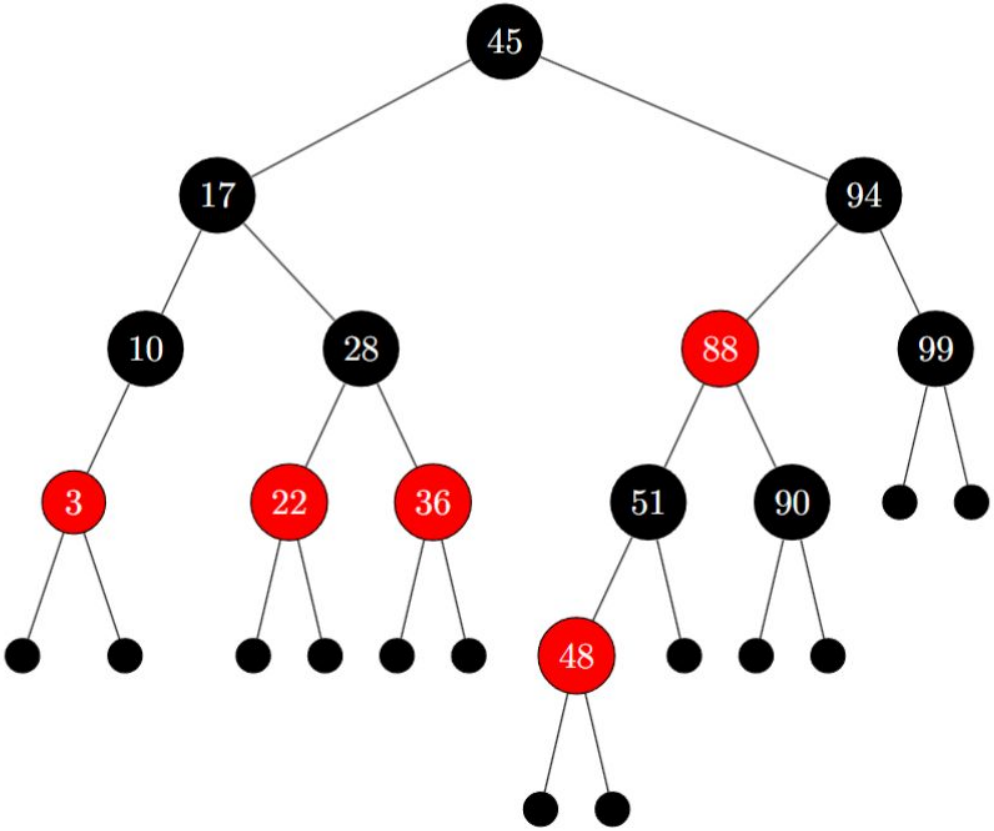
Árvores Red-Black

- Propriedades de uma árvore red-black:
  1. Cada nó ou é vermelho ou é preto
  2. A raiz 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 têm o mesmo número de nós pretos

Árvores Red-Black

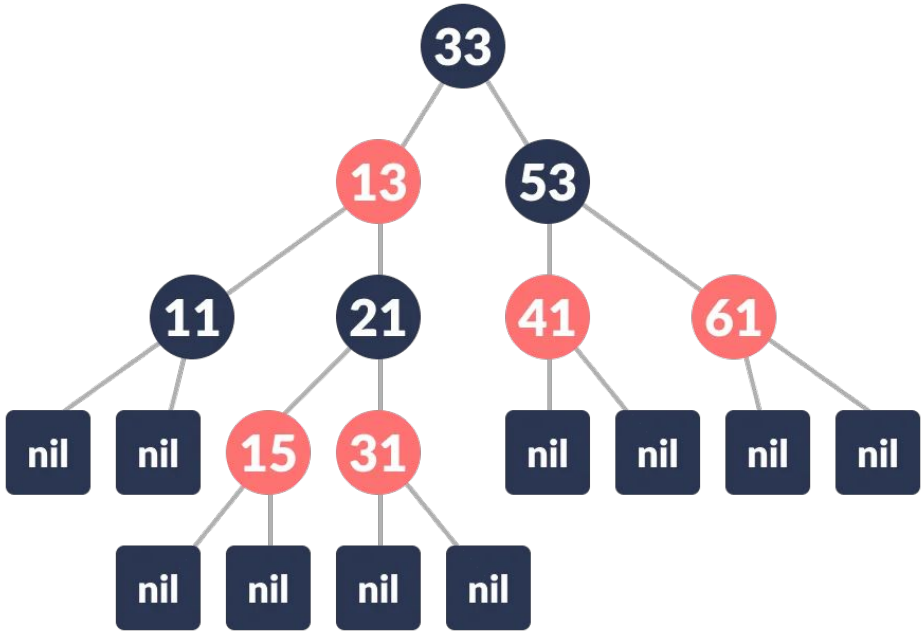
```
class ArvoreRB {
private:
    struct No {
        int info;
        enum { RED, BLACK } cor;
        No *esquerda, *direita, *pai;
    };

    No *raiz;

public:
    ArvoreRB() : raiz(nullptr) {}
}
```

Árvores Red-Black

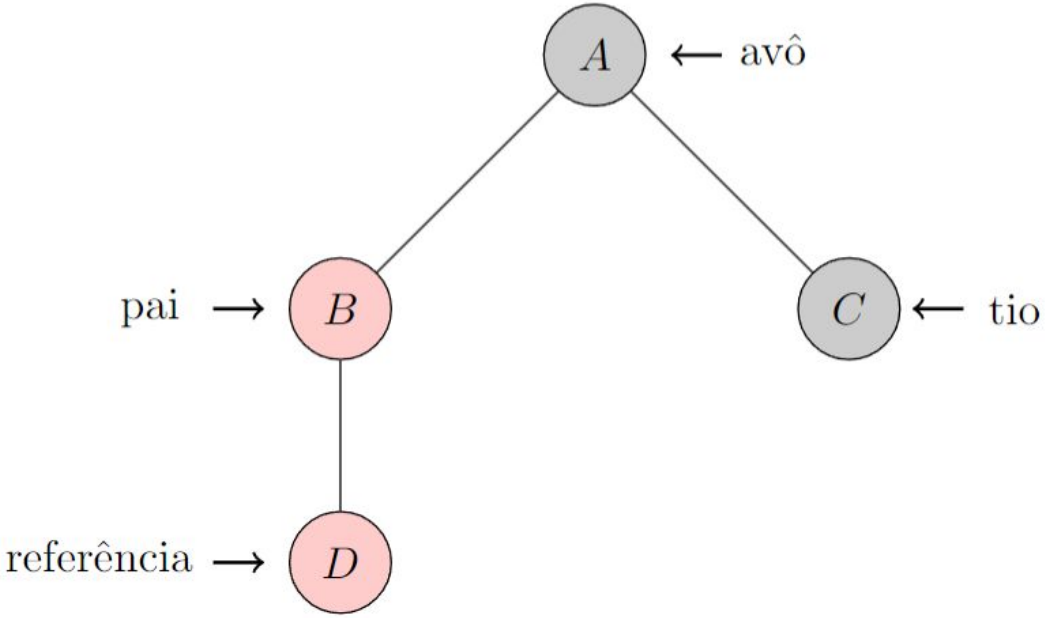


Árvores Red-Black

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 têm o mesmo número de nós pretos



Árvores Red-Black - Elementos Importantes



Árvores Red-Black - Elementos Importantes

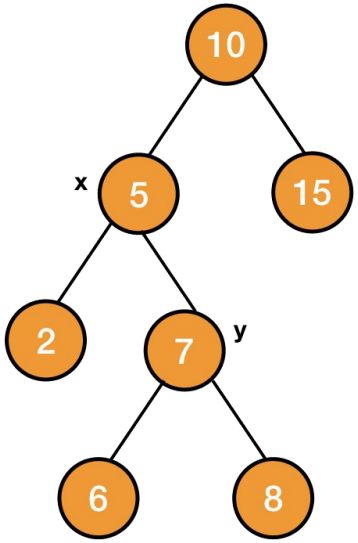
```
No * pai(No *no) {
    return no ? no->pai : nullptr;
}

No * avo(No* no)
{
    return pai(pai(no));
}

No* irmao(No* no)
{
    auto p = pai(no);
    return p ? (no == p->esquerda ? p->direita : p->esquerda) : nullptr;
}

No * tio(No* no)
{
    return irmao(pai(no));
}
```

Árvores Red-Black - Rotações

- As operações mais importantes de uma árvore balanceada, como no exemplo da red-black, são a inserção e a deleção, que é o momento onde o “auto-balanceamento” deverá ocorrer
- O balanceamento ocorre observando se alguma propriedade da árvore foi violada, e aplicando-se **rotações** nos elementos da árvore:
  a. Rotações à esquerda
  b. Rotações à direita

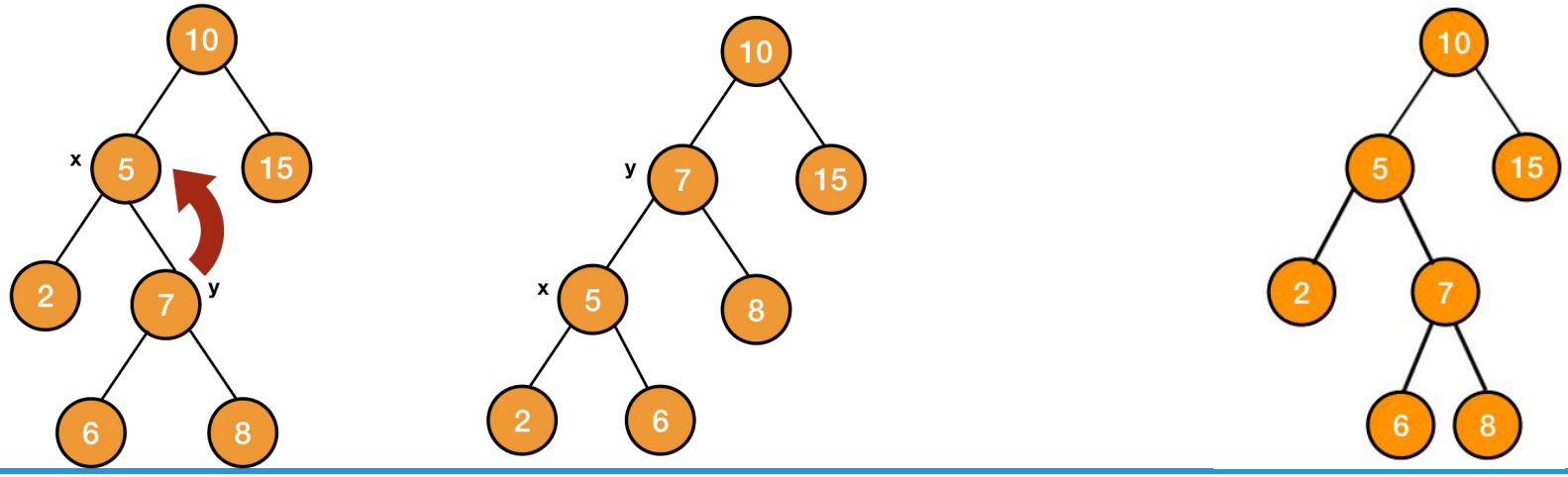
Árvores Red-Black - Rotações

- **Rotação à esquerda**
  - Nela, assumimos que o filho direito não é nulo



Árvores Red-Black - Rotações

- **Rotação à esquerda**
  - Depois de aplicar a rotação à esquerda no nó x, o nó y se tornará a nova raiz da subárvore e seu filho esquerdo será x
  - Já o filho esquerdo anterior de y agora se tornará o filho direito de x.

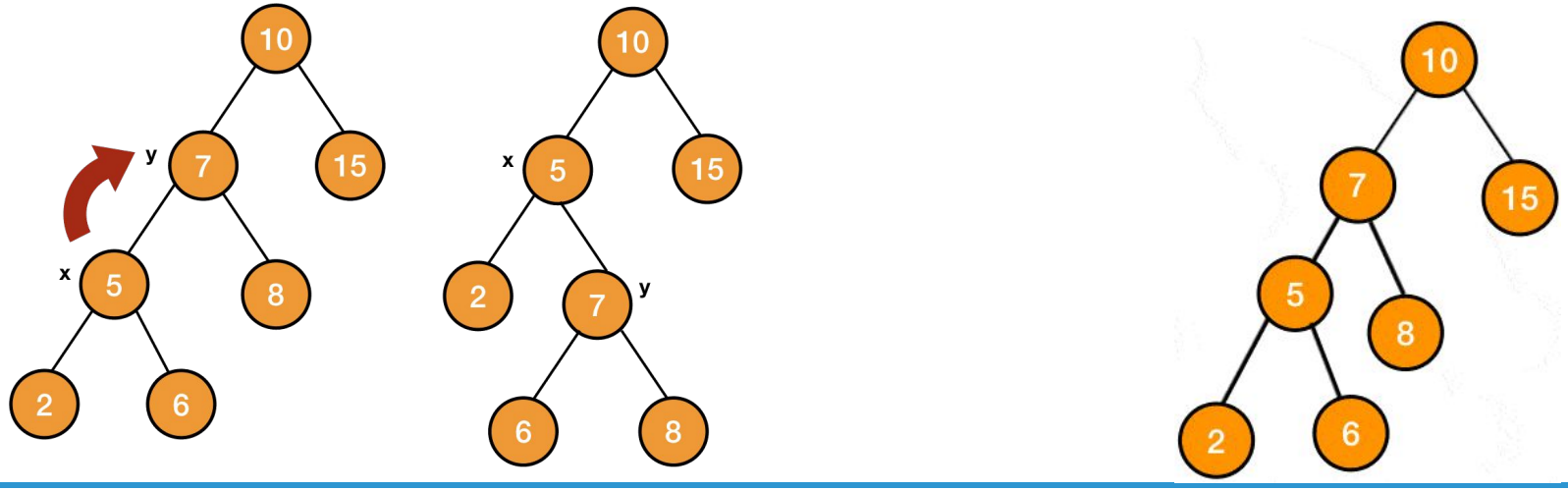


Árvores Red-Black - Rotações

```
void rotacao_esquerda(No *G, No *P, No *C) {
    if (G != nullptr)
        G->esquerda == P ? G->esquerda = C : G->direita = C;

    P->direita = C->esquerda;
    C->esquerda = P;

    C->pai = G;
    P->pai = C;

    if (P->direita)
        P->direita->pai = P;
}
```

Árvores Red-Black - Rotações

- Rotação à direita (de maneira análoga)
  - Então, a rotação para a direita no nó y fará de x a raiz da árvore, y se tornará o filho direito de x
  - E o filho direito anterior de x agora se tornará o filho esquerdo de y



Árvores Red-Black - Rotações

```
void rotacao_direita(No *G, No *P, No *C) {
    if (G != nullptr)
        G->esquerda == P ? G->esquerda = C : G->direita = C;

    P->esquerda = C->direita;
    C->direita = P;

    C->pai = G;
    P->pai = C;

    if (P->esquerda)
        P->esquerda->pai = P;
}
```

Árvores Red-Black
Operações de Inserção

Árvores Red-Black - Inserções

- A inserção em uma árvore red-black consiste em duas etapas
  a. A primeira é a inserção de um nó vermelho, nos mesmos moldes da inserção em uma árvore binária de busca
  b. A segunda etapa consiste em corrigir possíveis violações às propriedades de uma árvore red-black

Árvores Red-Black - Inserções

- Um detalhe é que as propriedade 1 e 3 sempre serão verdadeiras
- As demais propriedades podem ser violadas, a depender do local onde a inserção foi realizada
    - A partir da inserção, é possível ocorrer quatro cenários diferentes

Árvores Red-Black - Inserção (Cenário A) - árvore vazia

- Neste caso, a inserção é trivial, mas a propriedade 2 é violada
- A correção consiste em pintar a raiz com a cor preta
  - Este processo adiciona uma unidade ao tamanho de todos os caminhos que partem da raiz às folhas
  - Assim, a propriedade 5 ficará preservada

Árvores Red-Black - Inserção (Cenário A) - árvore vazia

- Segue abaixo uma visualização da inserção da informação 40 em uma árvore vazia:

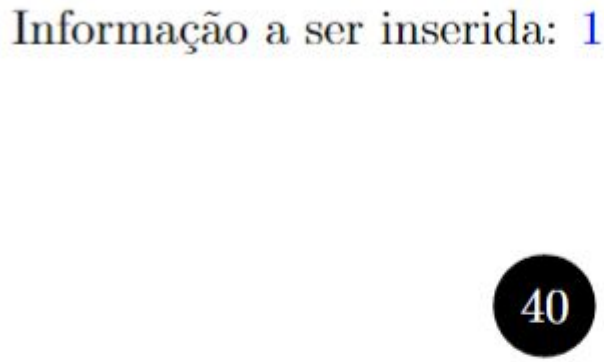
Árvores Red-Black - Inserção (Cenário B)
O pai do nó é preto (1/2)

- 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

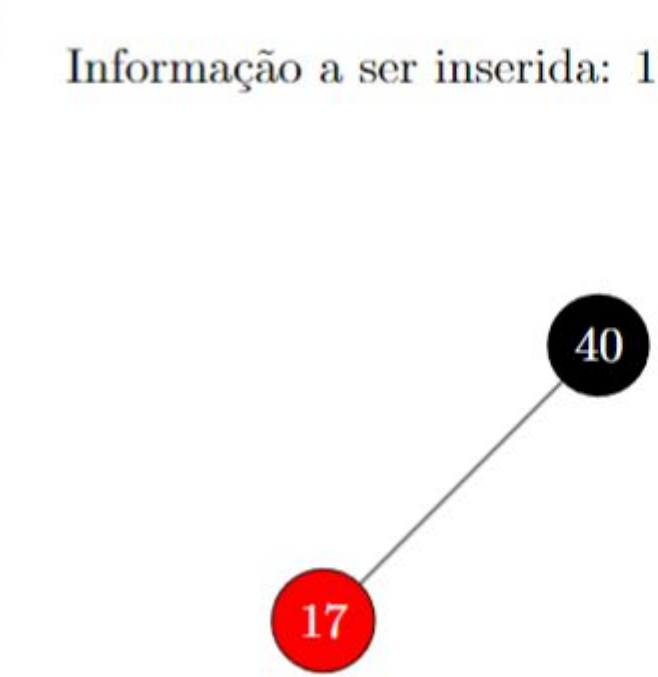
Árvores Red-Black - Inserção (Cenário B)
O pai do nó é preto (2/2)

- 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

Árvores Red-Black - Inserção (Cenário B)
O pai do nó é preto

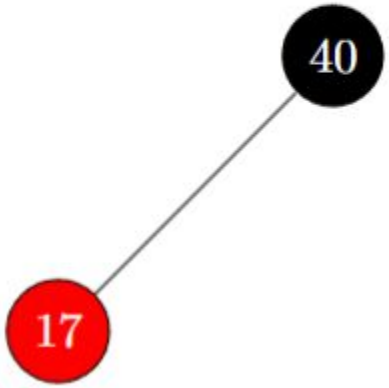


Árvores Red-Black - Inserção (Cenário B)
O pai do nó é preto



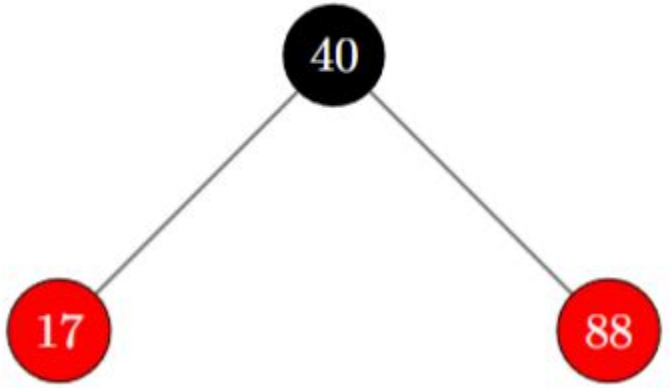
Árvores Red-Black - Inserção (Cenário B)
O pai do nó é preto

Informação a ser inserida: 88



Árvores Red-Black - Inserção (Cenário B)
O pai do nó é preto

Informação a ser inserida: 88



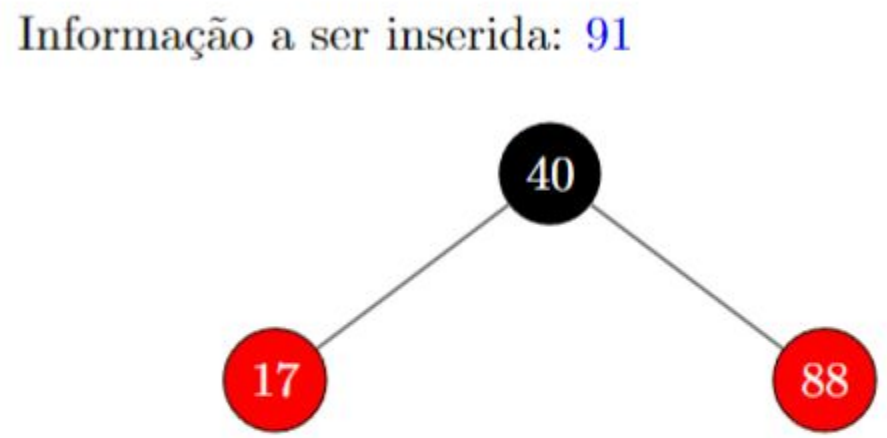
Árvores Red-Black - Inserção (Cenário C)
O pai e o tio do nó são vermelhos (1/2)

- Nesse cenário, o pai e o tio 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**

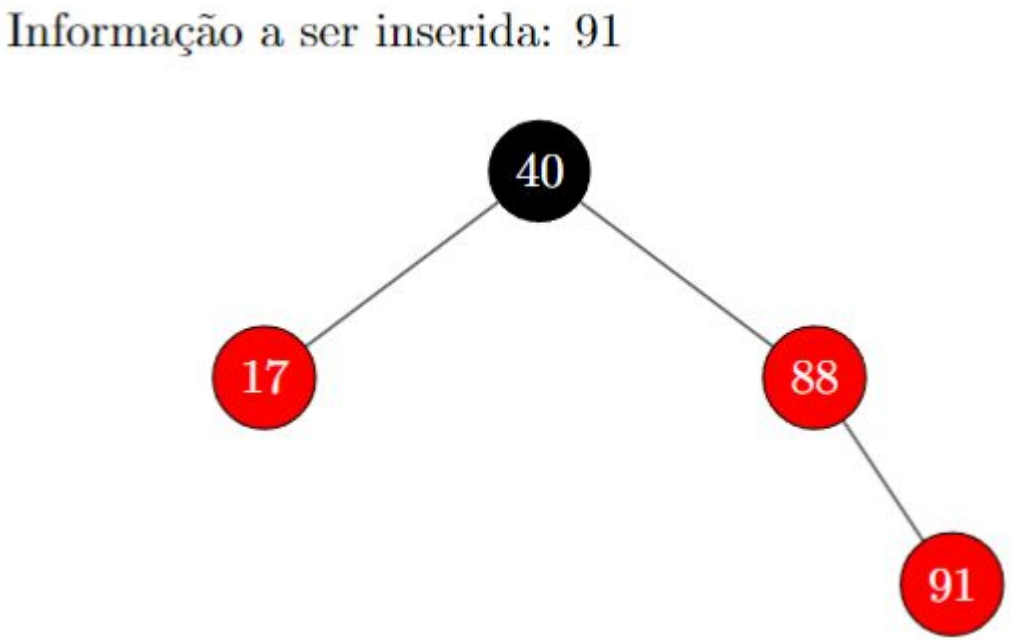
Árvores Red-Black - Inserção (Cenário C)
O pai e o tio do nó são vermelhos (2/2)

- Esta mudança também não viola a propriedade 5, pois o número de nós pretos nos caminhos não 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

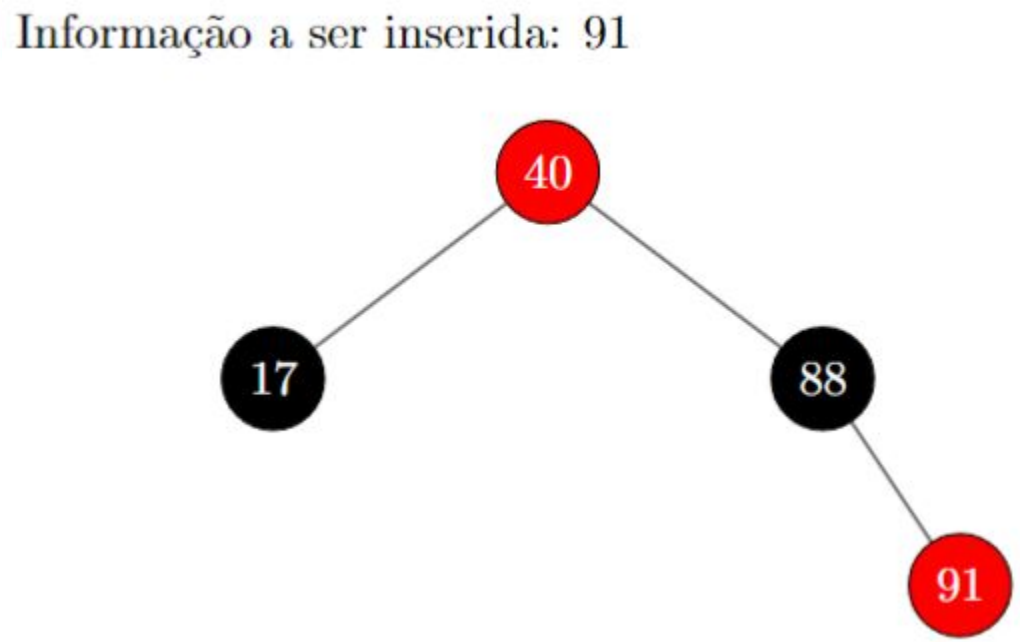
Árvores Red-Black - Inserção (Cenário C)
O pai e o tio do nó são vermelhos



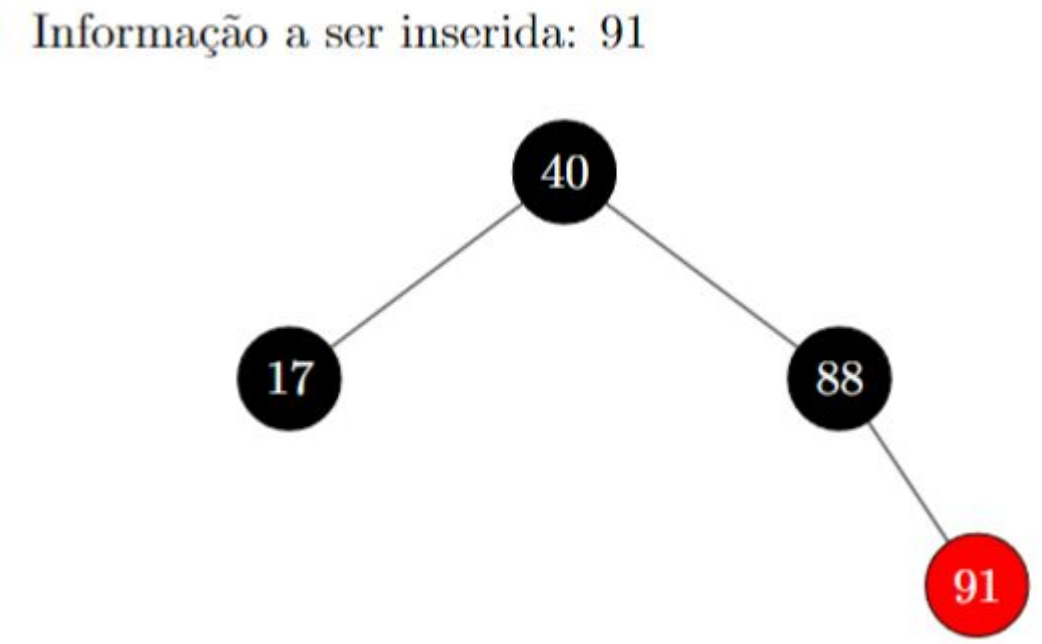
Árvores Red-Black - Inserção (Cenário C)
O pai e o tio do nó são vermelhos



Árvores Red-Black - Inserção (Cenário C)
O pai e o tio do nó são vermelhos



Árvores Red-Black - Inserção (Cenário C)
O pai e o tio do nó são vermelhos



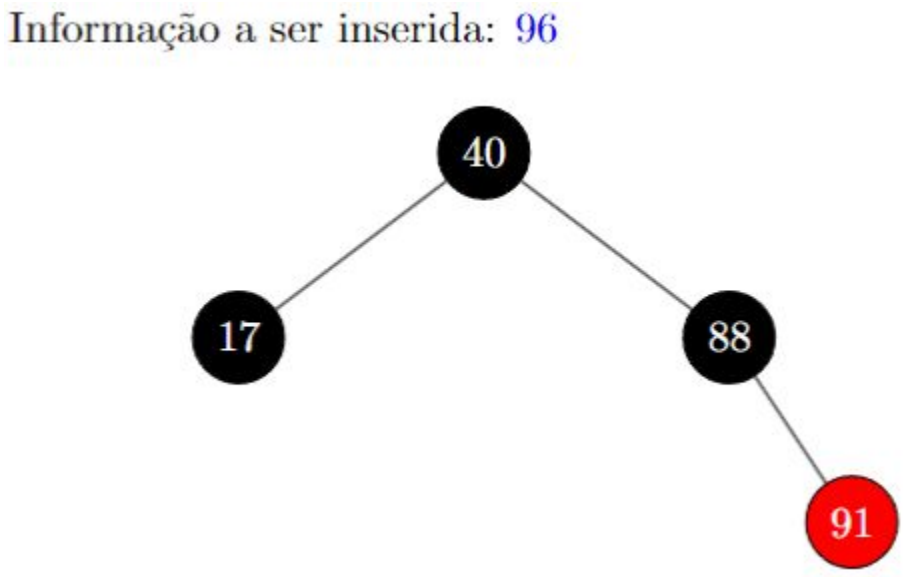
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto (1/2)

- Neste último cenário, é necessário utilizar rotações de modo a reposicionar o pai na posição do avô
- A direção da rotação é definida de acordo com a posição do pai em relação ao avô
  - Se o filho é à esquerda, a rotação é para a direita
  - Se o filho é à direita, a rotação é para a esquerda

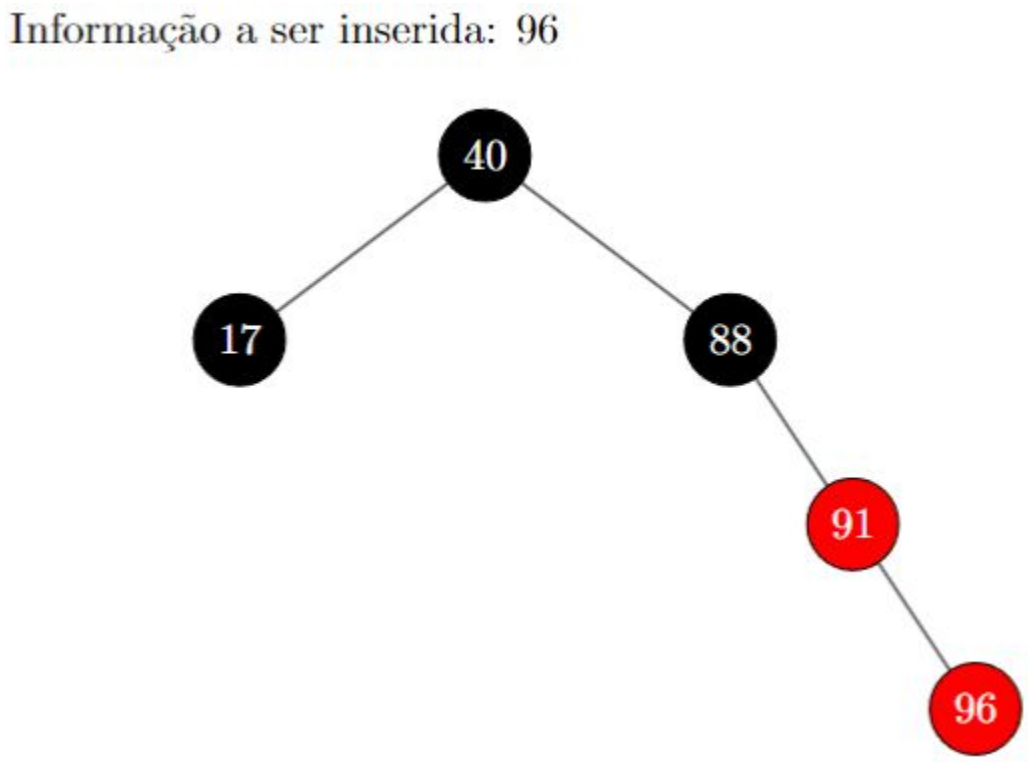
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto (2/2)

- Também é necessária a troca de cores entre o pai e o avô após a rotação
- Esta troca restaura a violação à propriedade 4 e não viola a propriedade 5
- Há, porém, um caso especial se o filho estiver em direção oposta a que o pai ocupa em relação ao avô
    - Neste caso, é necessária uma rotação para colocar o filho na posição do pai
- Deste modo, ambos passaram a ter a mesma direção em relação ao avô

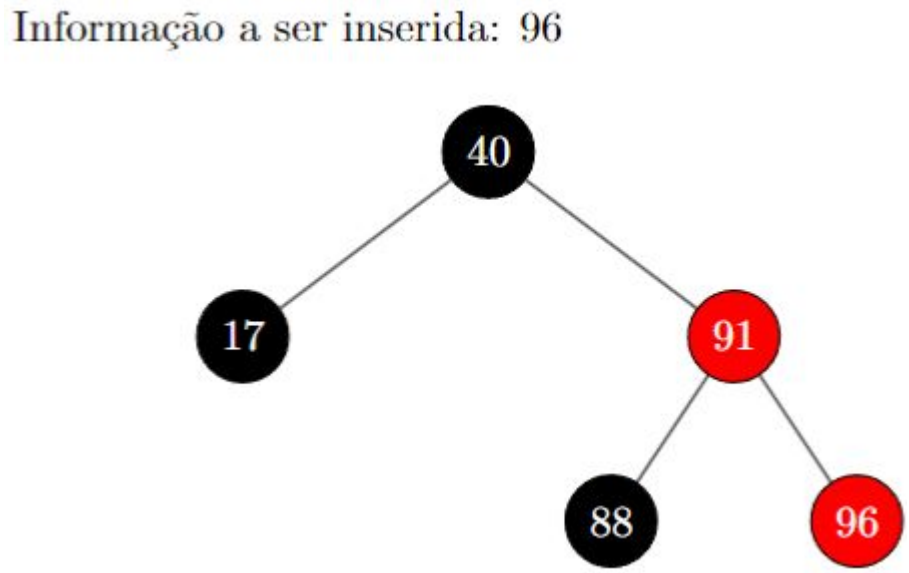
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



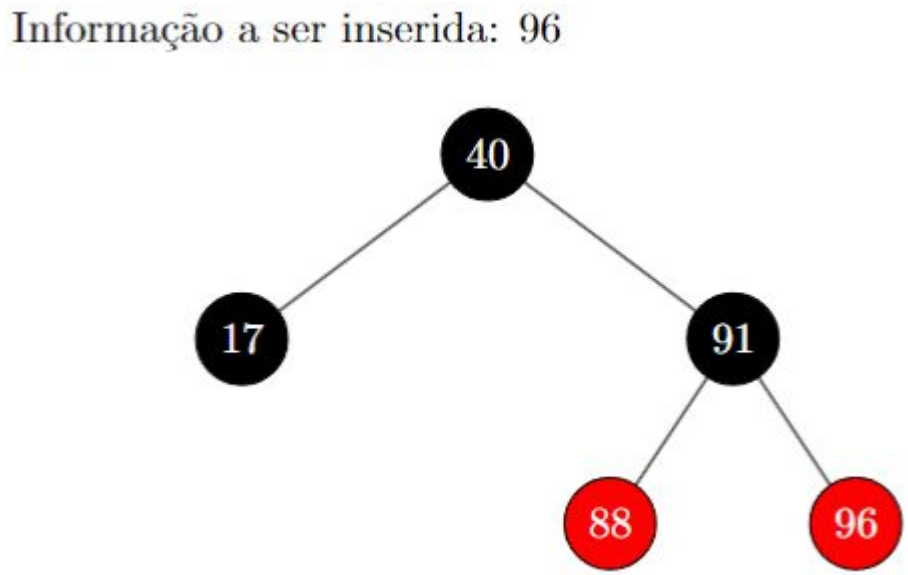
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



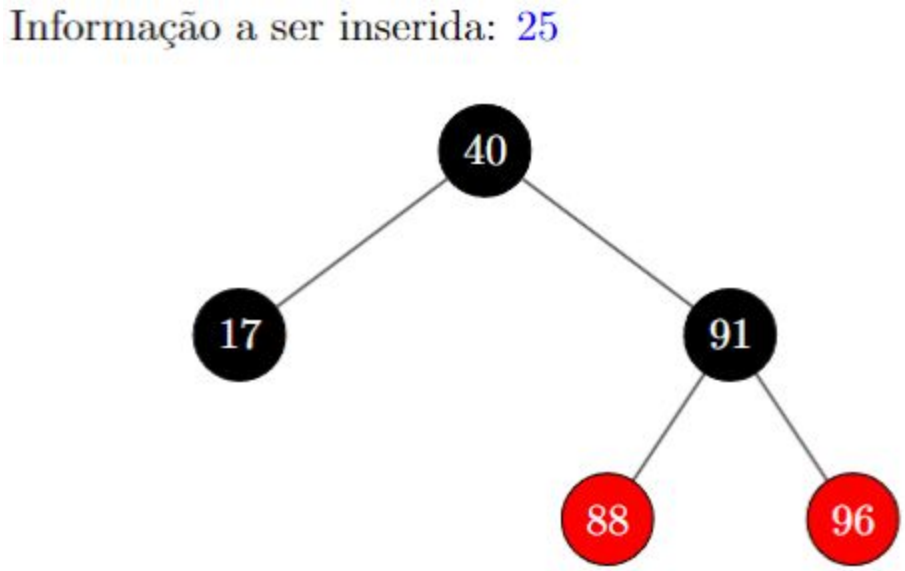
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



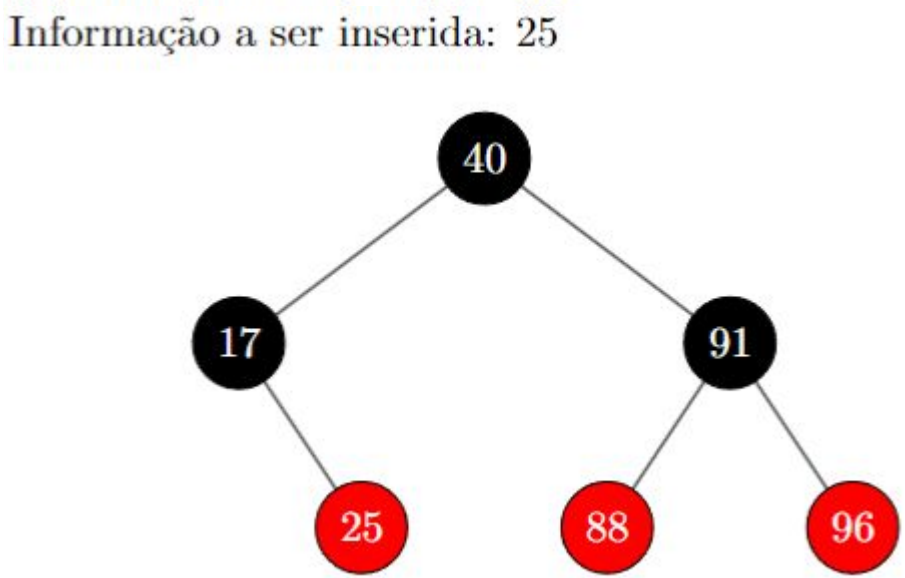
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



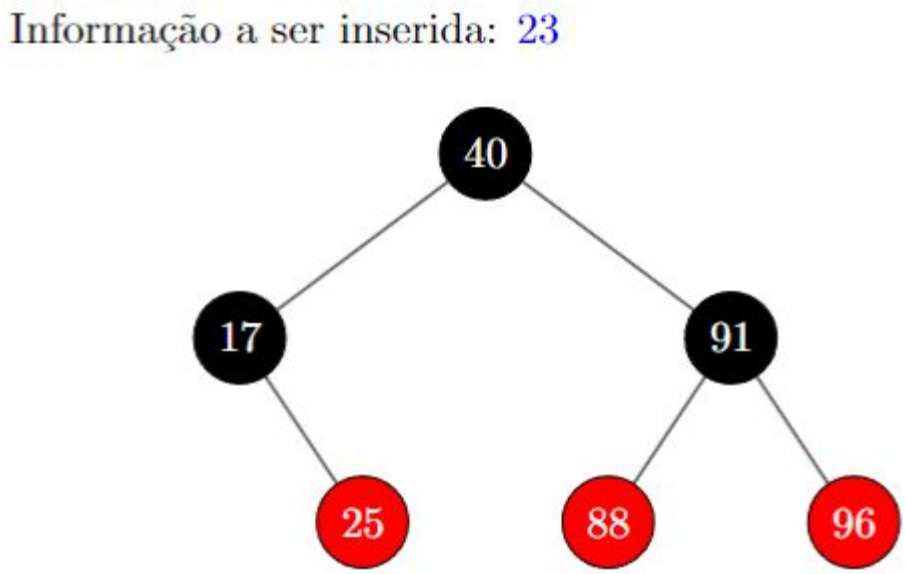
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



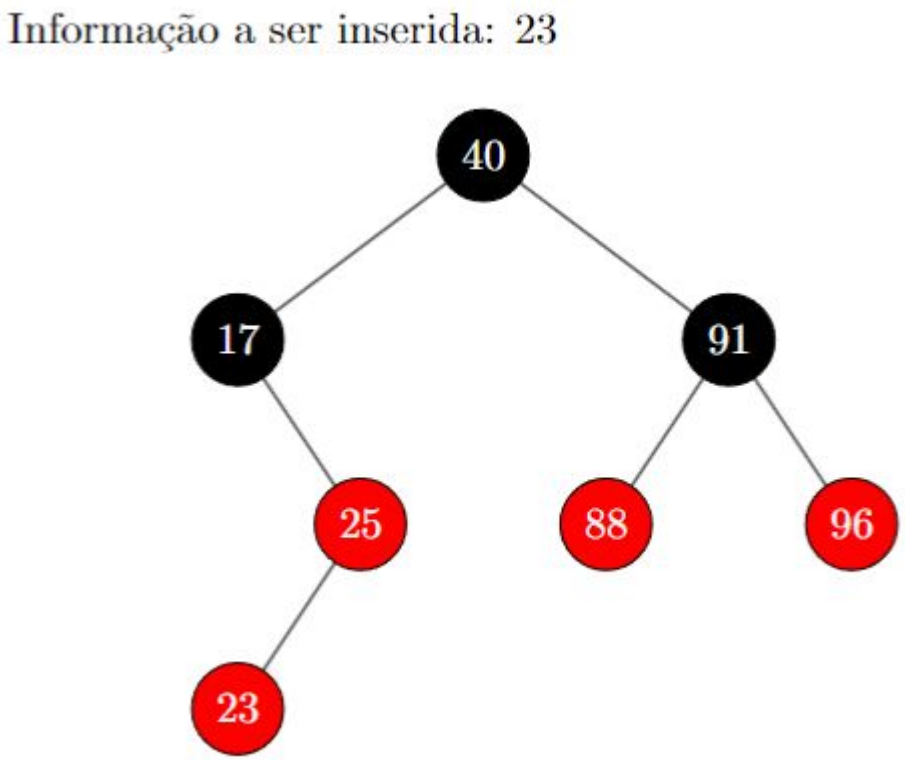
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



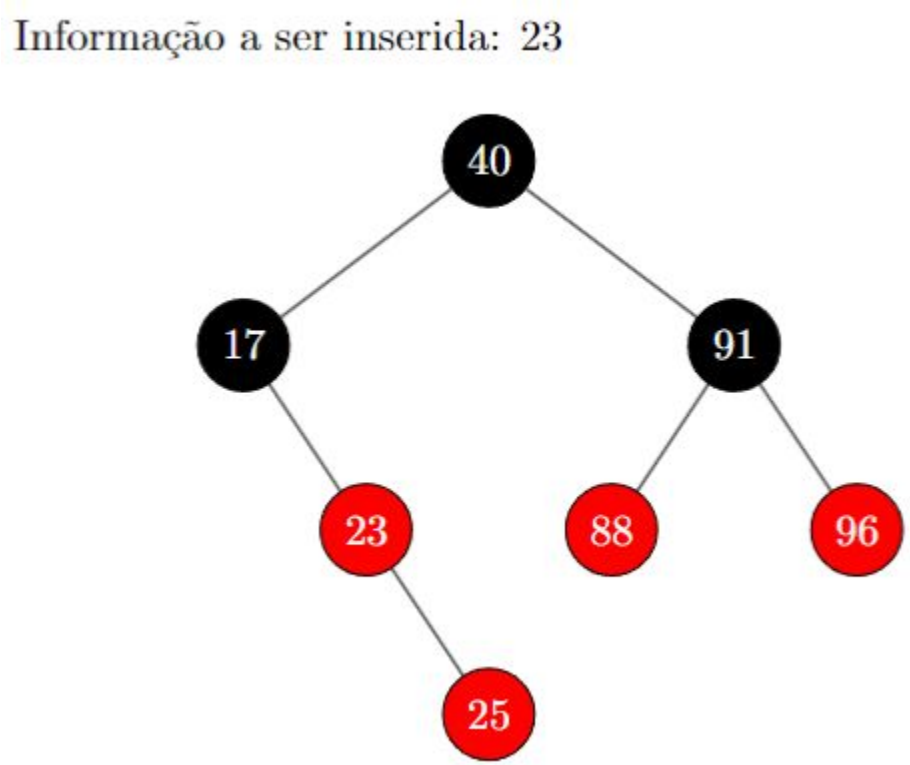
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



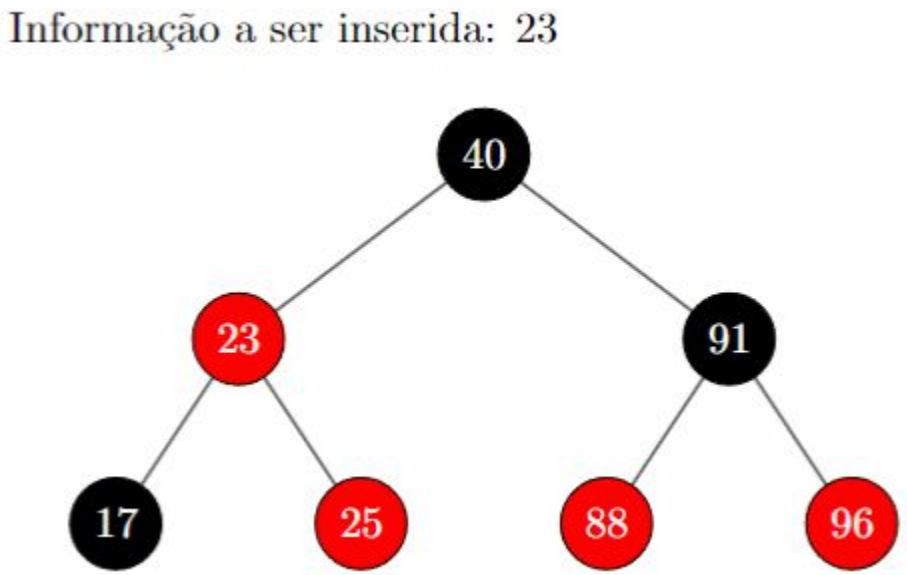
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



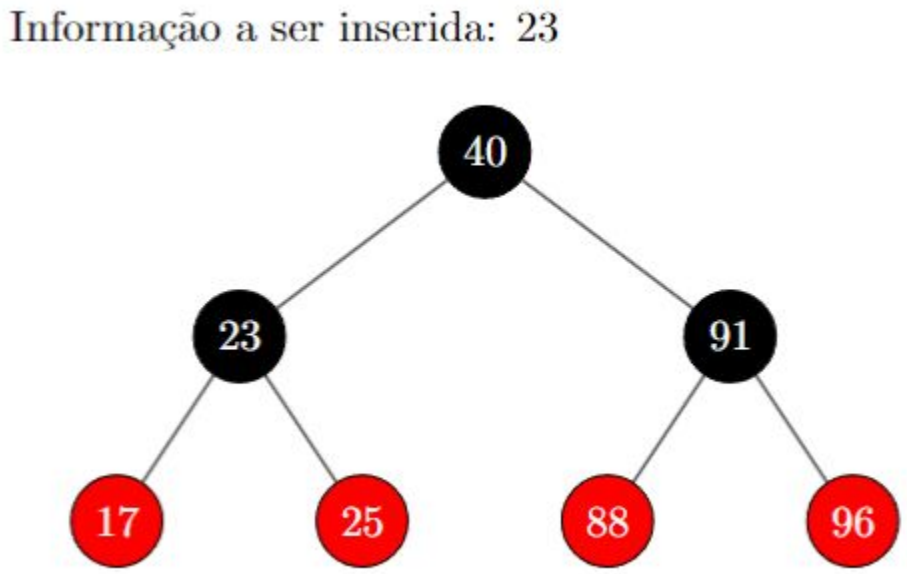
Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



Árvores Red-Black - Inserção (Cenário D)
O pai é vermelho e o tio é preto



Árvores Red-Black
Operações de Remoção

Árvores Red-Black - Remoções

- De forma semelhante ao que acontece com as árvores binárias de busca, a remoção deve ser tratada em três casos distintos
- Os casos **dependem do número de filhos que não sejam folhas** do nó N a ser removido:
  - Nenhum
  - Um
  - Dois

Árvores Red-Black - Remoções

- O caso de dois filhos pode ser removido a um dos dois casos anteriores, usando a mesma estratégia da remoção por cópia: basta substituir a informação do nó N pela informação do nó mais à direita D da sub-árvore à esquerda e proceder com a remoção fazendo N = D
- Assim, a partir deste ponto, será considerado que o nó N a ser removido tem, no máximo, um filho C que não seja folha

Árvores Red-Black - Caso trivial de remoção (1/2)

- O caso trivial da remoção ocorre quando **a informação a ser removida não consta na árvore**
- Neste caso **não há o que remover**, e a rotina deve ser encerrada
- Para identificar tal caso, é necessário o auxílio de uma função auxiliar, que localiza o ponteiro do nó a ser removido

Árvores Red-Black - Caso trivial de remoção (2/2)

- Caso esta rotina não localize o nó, o ponteiro retornado será nulo e a remoção será encerrada
- Outra rotina auxiliar útil é a que reduz o caso 3 (o nó N a ser removido tem dois filhos que não são folhas) para o caso 1 ou o caso 2
- Esta rotina deve retornar o ponteiro do novo nó a ser removido

Árvores Red-Black - Remoção de nó vermelho (1/2)

- Se N é vermelho, pela propriedade 2 seu filho C deve ser preto
- A substituição de N por C mantém as propriedades da árvore red-black
- Primeiramente, se C vir a ocupar a raiz da árvore, esta será preta, mantendo a propriedade 2
- O pai P de N é necessariamente preto, uma vez que N é vermelho, de modo que a propriedade 4 não é violada

Árvores Red-Black - Remoção de nó vermelho (2/2)

- Por fim, como o nó removido é vermelho, o número de nós pretos nos caminhos não se altera, mantendo a propriedade 5
- Observe que este caso só ocorre se ambos filhos de N são folhas
    - O caso de apenas uma folha violaria a propriedade 5 de uma árvore red-black

Árvores Red-Black
Remoção de nó preto com filho vermelho (1/2)

- Se N é preto e seu filho C é vermelho, a remoção de N viola a propriedade 5, devido a redução no número de nós pretos
- Além disso, a promoção de um nó vermelho pode levar a violação da propriedade 4
- Uma forma de preservar ambas propriedades é recolorir C como um nó preto

Árvores Red-Black
Remoção de nó preto com filho vermelho (2/2)

- Isto restaura a violação da propriedade 5, pois a perda de N agora é compensada com a adição de um novo nó preto
- O fato de C assumir a cor preta evita que a propriedade 4 seja violada

Árvores Red-Black
Remoção de nó preto com filho preto (1/2)

- O caso onde ambos N e C são pretos é o mais complexo dentre todos os que envolvem um nó com, no máximo, um filho não-folha
- A remoção de um nó preto viola a propriedade 5 das árvores Red-black
- Há múltiplos cenários possíveis, cada um sendo necessário ser tratado adequadamente, por meio do rebalanceamento da árvore

Árvores Red-Black
Remoção de nó preto com filho preto (2/2)

- Observe que este caso ocorre apenas quando ambos filhos de N são folhas
- Isto porque se N tivesse apenas uma folha preta, o fato do outro filho não ser folha violaria a propriedade 5

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário A - após a troca de N e C, C é a raiz

- Este é o cenário **mais simples**
- A remoção de um nó preto da posição raiz subtrai igualmente uma unidade de todos os caminhos da raiz às folhas
- Assim a propriedade 5 fica preservada
- As demais propriedades também se mantém: como as folhas são pretas, a raiz também será preta

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário B - Irmão vermelho (1/2)

- Se o nó a ser removido tem um irmão S vermelho, é preciso promover um reposicionamento dos nós, além de uma troca de cores entre o pai P e o irmão S
- Este cenário não pode ser resolvido diretamente: o resultado deste reposicionamento levará a um dos próximos cenários
- Primeiramente as cores de P e S devem ser trocadas

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário B - Irmão vermelho (2/2)

- Uma rotação de S em torno de P , o torna o novo avô de N
- Ao final deste, uma das duas subárvores terá caminho da raiz até uma folha uma unidade menor do que a outra, violando a propriedade 5
- Porém o caso a ser tratado mudou
    - Agora N tem pai vermelho, com irmão preto
- Este cenário será abordado mais adiante

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário C - Pai, irmão e sobrinhos pretos (1/2)

- Neste caso é necessário recolorir S como vermelho
- Isto faz com que todos os caminhos que passem por S tenham um nó preto a menos
- Porém a remoção de N também reduz um nó preto dos caminhos que passavam por N

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário C - Pai, irmão e sobrinhos pretos (2/2)

- Contudo, os caminhos que passam por P agora tem um nó preto a menos do que os caminhos que não passam, violando a propriedade 5
- Assim é preciso reiniciar o rebalanceamento em P

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário D - Irmão e sobrinhos pretos, pai vermelho (1/2)

- Este cenário é semelhante ao cenário C
    - Pai, irmão e sobrinhos preto
- Contudo, a troca de cores em P e S, neste caso, não viola a propriedade 5, pois não há mudança no número de nós pretos nos caminhos que passam por S

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário D - Irmão e sobrinhos pretos, pai vermelho (2/2)

- O número de nós pretos dos caminhos que passam por N aumenta em um, mas este número volta ao original após a remoção de N , o qual é preto
- Assim, este caso se encerra com esta troca de cores

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário E - Irmão preto, sobrinho à esquerda vermelho (1/2)

- Se N é o filho à esquerda de P , é necessário rotacionar o sobrinho à direita em torno de S
- Esta transformação torna o sobrinho à esquerda o novo irmão de N
- Antes da rotação, as cores de S e do sobrinho vermelho devem ser trocadas
- Estas modificações não violam a propriedade 5

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário E - Irmão preto, sobrinho à esquerda vermelho (2/2)

- Agora, N tem irmão preto com sobrinho à direita vermelho, o que configura o sexto e último cenário (F - que é consequência do E)
    - F: Irmão preto, sobrinho à direita vermelho
- Se N é o filho à direita de P e o sobrinho vermelho está à direita de S, a situação é simétrica
- Contudo, a rotação deve ser à esquerda em torno de S

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário F - Irmão preto, sobrinho à direita vermelho (1/2)

- Neste cenaŕio, N é o filho à esquerda de P , S é preto e o sobrinho à direita é vermelho
- O rebalanceamento consiste em rotacionar S à esquerda em torno de P , trocar as cores de P e S e tornar o sobrinho preto
- A subárvore manterá a cor de sua raiz, preservando a propriedade 4

Árvores Red-Black
Remoção de nó preto com filho preto
Cenário F - Irmão preto, sobrinho à direita vermelho (2/2)

- A mudança de cores não viola a propriedade 5, mas é preciso observar que, após a rotação, N tem um ancestral preto a mais, e N será removido, fazendo com que a contagem de nós pretos original se mantenha
- É possível observar que os caminhos que não passam por N preservam o mesmo número de nós pretos que tinham antes da rotação
- Este caso se encerra restaurando as propriedades da árvore red-black

Árvores Red-Black

- Árvores Red-Black possuem uma implementação complexa que provê um semi-balanceamento e todas as operações essenciais em O(lg N)
- Possui diversos usos na prática, desde o Kernel do Linux a implentações de estruturas abstratas em linguagens de programação

Árvores Red-Black

- Existem outras estruturas (árvores) auto balanceadas que provêem a mesma complexidade
  - Árvores AVL
    - Utiliza como base as operações de rotação aprendidas mais o controle da altura da árvore para manter o balanceamento
    - Mais teórica (pouca ou nenhuma aplicação)
  - Árvores B
    - N-ária (nós com dois ou mais filhos)
    - Usada no controle de armazenamento de arquivos

Árvores Binária de Busca STL

Árvores Binárias de Busca na STL

- A STL (Standard Template Library) da linguagem C++ não oferece uma implementação básica de árvores binárias de busca que permita o acesso direto aos nós e seus ponteiros
- Entretanto, ela oferece tipos de dados abstratos cujas implementações utilizam árvores binárias de busca auto-balanceável

Árvores Binárias de Busca na STL

- O padrão da linguagem não especifica qual árvore deve ser utilizada na implementação, e sim as complexidades assintóticas esperadas para cada operação
- Segundo o site https://en.cppreference.com/, em geral são utilizadas árvores red-black
- Os principais tipos de dados abstratos implementados são os conjuntos (sets) e os dicionários (maps)

Set

Set

- O conjunto (set) é um tipo de dado abstrato que representa um conjunto de elementos únicos
- Estes elementos são mantidos em ordem crescente, de acordo com a implementação do operador < do tipo de elemento a ser armazenado
- O tipo de dado a ser armazenado é paramétrico, e deve ser definido na instanciação do conjunto

Set

- A principal característica dos conjuntos é a eficiência nas operações de inserção, remoção e busca
- Todas as três tem complexidade O(log N), onde N é o número de elementos no conjunto

Construção de um Set

- O padrão C++11 oferece cinco construtores distintos para um set (1/2)
  a. O primeiro é o padrão e constrói um conjunto vazio
  b. Range constructor permite a construção de um conjunto a partir de dois iteradores, first e last, que determinam um intervalo de valores
     - Também permite a definição de um alocador customizado
  c. O terceiro, copy constructor, cria uma cópia exata do set passado como parâmetro

Construção de um Set

- O padrão C++11 oferece cinco construtores distintos para um set (2/2)
  d. O quarto, move constructor, move o conteúdo do set passado como parâmetro para o novo conjunto
  e. O quinto, initializer-list constructor, cria um novo set com os elementos passados na lista de inicialização

Construção de um Set

```
set<int> s1;                          // Conjunto vazio

string s = "IDP";

set<char> s2(s.begin(), s.end());     // Conjunto com os caracteres de s

set<char> s3(s2);                     // Conjunto cópia de s2

set<char> s4(move(s3));               // s4 == s2 e s3 torna-se vazio

set<double> s5 {1.1, 2.2, 3.3};       // Conjunto com 3 elementos
```

Set - Principais Operações

- As principais operações em um conjunto são a inserção, remoção e busca, todas com complexidade O(log N), onde N é o número de elementos armazenados no conjunto
- A inserção é feita através do método insert(), que pode receber ou o valor a ser inserido ou uma lista de inicialização com os elementos a serem inseridos

Set - Principais Operações

- **A inserção de um valor que já existe no conjunto não tem efeito**
- O método erase() remove o nó que contém o valor passado como parâmetro, se existir tal valor no conjunto
  - O retorno do método pode ser utilizado para se determinar quantos elementos foram removidos

Set - Principais Operações

- Para se determinar se um elemento está ou não no conjunto há duas alternativas:
  1. Utilizar o método count(), que retorna o número de ocorrências do valor passado como parâmetro
  2. Utilizar o método find() que retorna o iterador para o elemento que contém o valor, ou o iterador end(), caso contrário

Set - Principais Operações

```
set<int> c;

c.insert(7);                    // c == {7}
c.insert(3);                    // c == {3, 7}
c.insert(7);                    // c == {3, 7}
c.insert({4, 8});               // c == {3, 4, 7, 8}

cout << c.size() << endl;       // 4

auto n = c.erase(3);            // n == 1 e c == {4, 7, 8}
n = c.erase(12);                // n == 0 e c == {4, 7, 8}

n = c.count(4);                 // n == 1

auto it = c.find(33);           // it == c.end()
```

Set - Operações Relevantes

- O método empty() verifica se o conjunto está ou não vazio
- O método size() determina o número de elementos
- Tanto empty() quando size() tem complexidade constante

Set - Operações Relevantes

- O método lower_bound() retorna um iterator para o primeiro elemento do conjunto cuja informação é maior ou igual ao valor passado
- O método upper_bound() tem comportamento semelhante, retornando um iterador para o elemento cuja informação é estritamente maior do que valor passado como parâmetro
- Ambos métodos tem complexidade O(log N)

Set - Operações Relevantes

```
set<int> s {1, 2, 4, 8, 15, 16, 3, 4, 1, 2};

cout << s.size() << endl;           // 7

cout << s.empty() << endl;          // 0

auto it1 = s.lower_bound(15);       // it1 == 15
if (it1 != s.end())
    cout << *it1 << endl;           // 15

auto it2 = s.upper_bound(15);       // it2 == 16
if (it2 != s.end())
    cout << *it2 << endl;           // 16

for (auto x: s)
    cout << x << " ";               // 1 2 3 4 8 15 16
```

Multiset

Multiset

- A STL também oferece a implementação de um conjunto que permite a inserção de elementos repetidos, denominado multiset
- O retorno do método count() corresponde ao número de ocorrências de um mesmo valor
- **O método erase() apaga todas as ocorrências do valor passado**

Multiset

- Para remover somente uma ocorrência, esta ocorrência deve ser localizada com o método find() e o iterador de retorno deve ser passado como parâmetro para o método erase()
- Uma travessia usando range for passa uma vez em cada ocorrência de cada elemento
- O método equal_range() retorna um par de iteradores que delimitam o intervalo de valores idênticos ao valor passado como parâmetro

Multiset

```
multiset<int> ms = {1, 5, 5, 5, 8, 4, 12, 5, 1, 1};

cout << ms.size() << endl;          // 10
cout << ms.count(5) << endl;        // 4
cout << ms.count(1) << endl;        // 3

ms.erase(1);                       // ms == {4, 5, 5, 5, 5, 8, 12}
ms.erase(ms.find(5));              // ms == {4, 5, 5, 5, 8, 12}

auto [a, b] = ms.equal_range(5);   // a == iterator para o primeiro 5
                                   // b == iterator para 8 (primeiro elemento maior que 5)

for (auto it = a; it != b; it++)
    cout << *it << " ";            // 5 5 5
cout << endl;
```

Map

Map

- map é um tipo abstrato de dados da STL do C++ que abstrai o conceito de dicionário
- Cada elemento do map é composto de uma chave (key) e um valor associado (value)
- Tanto o tipo da chave quanto do valor são paramétricos, e podem ser distintos

Map

- Os elementos são ordenados por meio de suas chaves
  - As chaves são únicas
- A inserção de um par (key, value) para uma chave já inserida modifica o valor da chave existente
- As operações de inserção, remoção e busca são eficientes, com complexidade O(log N), onde N é o número de elementos inseridos no map

Map

- Embora sejam tipos abstratos de dados distintos, as interfaces do map e do set contém inúmeras interseções
  - Todos os métodos apresentados para o set estão também disponíveis para o map
- A principal diferença reside no fato de que os iteradores do map são pares

Map

- O primeiro elemento de um iterador é a chave e o segundo elemento é o valor
- Além do map, a STL também oferece o multimap, o qual suporta chaves repetidas

Map

```
map<int, int> m;                    // Mapa vazio

m[5] = 10;                          // m == {{5, 10}}
m[3] = 10;                          // m == {{3, 10}, {5, 10}}

m[5] = 20;                          // m == {{3, 10}, {5, 20}}

m.insert({12, 12});                 // m == {{3, 10}, {5, 20}, {12, 12}}

m.erase(5);                         // m == {{3, 10}, {12, 12}}

for (auto [k, v] : m)
    cout << k << " " << v << endl;
```

Conclusão