

Árvores Red-Black

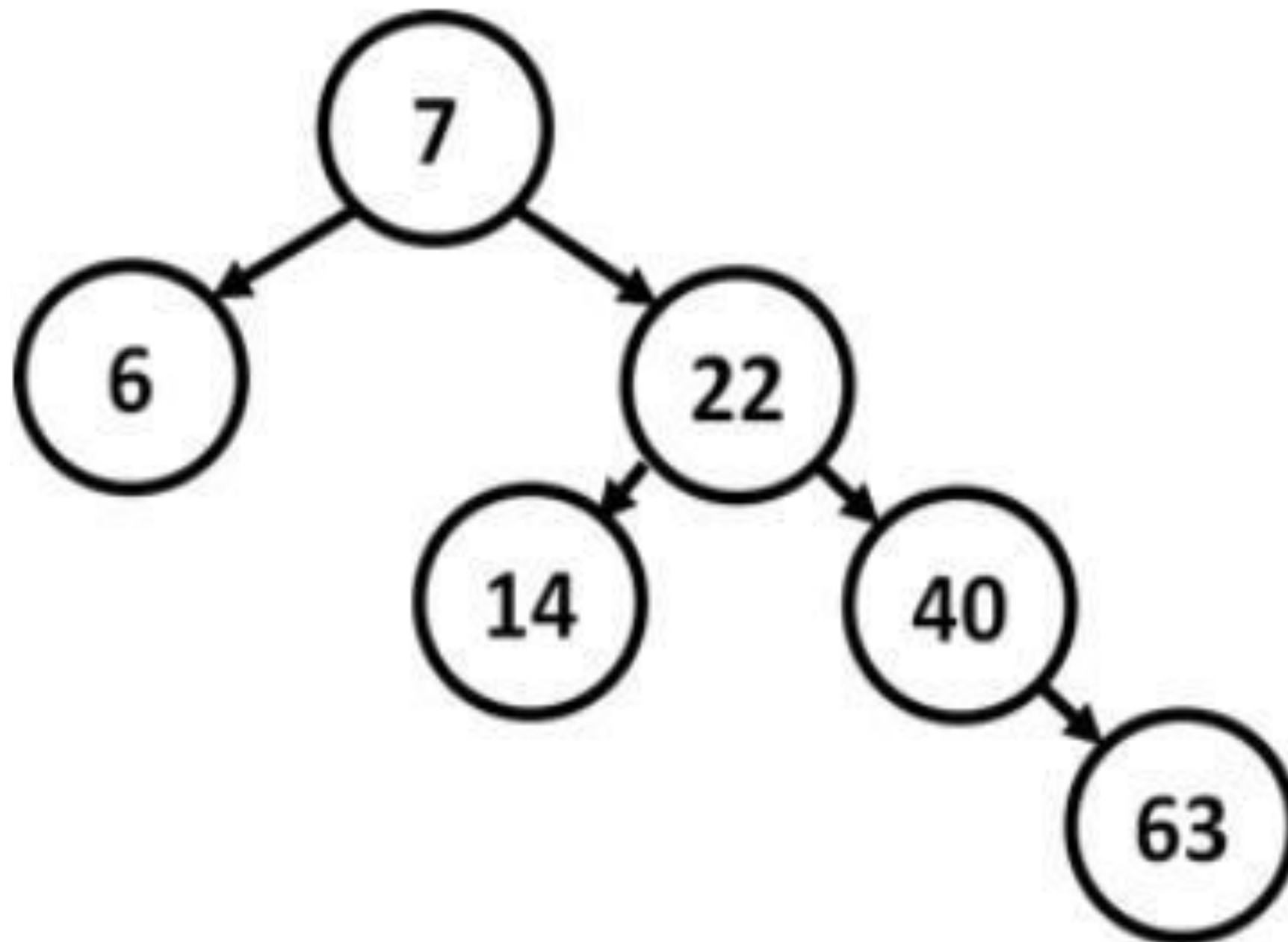
**Eduardo Maciel | Eliezir Moreira | Josenilton Ferreira
Lucas Cassiano | Maria Letícia**

<https://github.com/theduardomaciel/projeto-ed>

Motivação

- Imagine que temos um conjunto de elementos e precisamos realizar operações de busca, inserção e remoção eficientes dentro desse conjunto.
- Queremos garantir que a árvore resultante seja balanceada para evitar cenários extremos de desempenho e exija poucas reorganizações da árvore.
- Como podemos fazer isso?

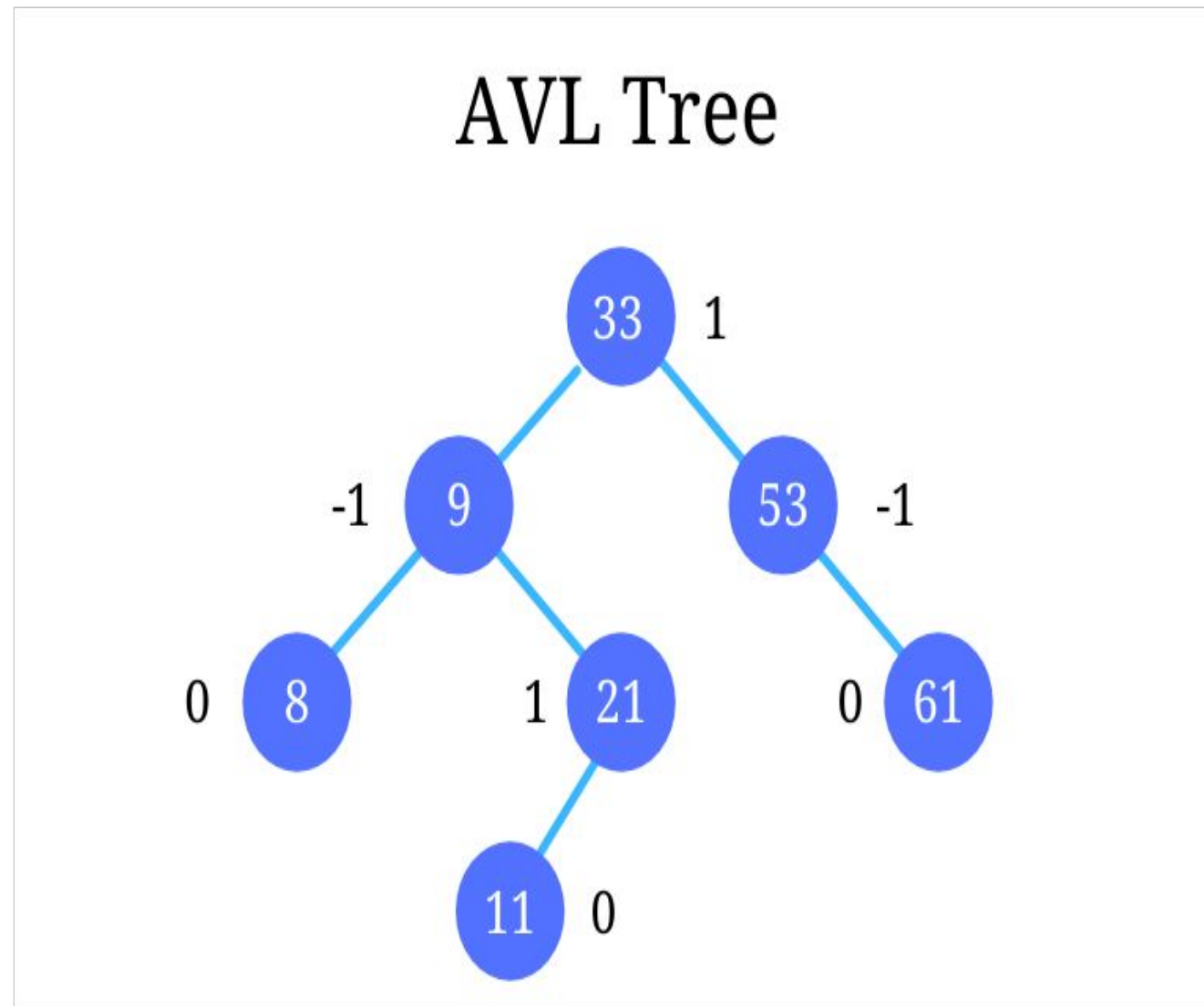
Árvores de Busca Binária?



Problemas da BST:

- . Não há garantia de balanceamento.
- . A altura da árvore pode se tornar desbalanceada, levando a operações ineficientes (pior caso de busca em **$O(n)$**).

Árvores AVL?



Problemas da AVL:

- As operações de balanceamento podem ser custosas.
- Requer mais espaço de armazenamento para manter os fatores de balanceamento.

E qual seria a solução?

Árvores Red-Black

- É uma estrutura de dados que combina as propriedades de árvores binárias de busca com regras adicionais para manter o equilíbrio.

As árvores Red-Black têm várias aplicações, incluindo:

- Estruturas de Dados Internas: São usadas em implementações de dicionários, mapas e conjuntos.
- Árvores AVL Alternativas: As árvores Red-Black são mais simples de implementar do que as árvores AVL, mas ainda mantêm um bom equilíbrio.
- Algoritmos de Busca e Inserção Eficientes: As operações de busca, inserção e remoção em árvores Red-Black têm complexidade **$O(\log n)$**

Definições

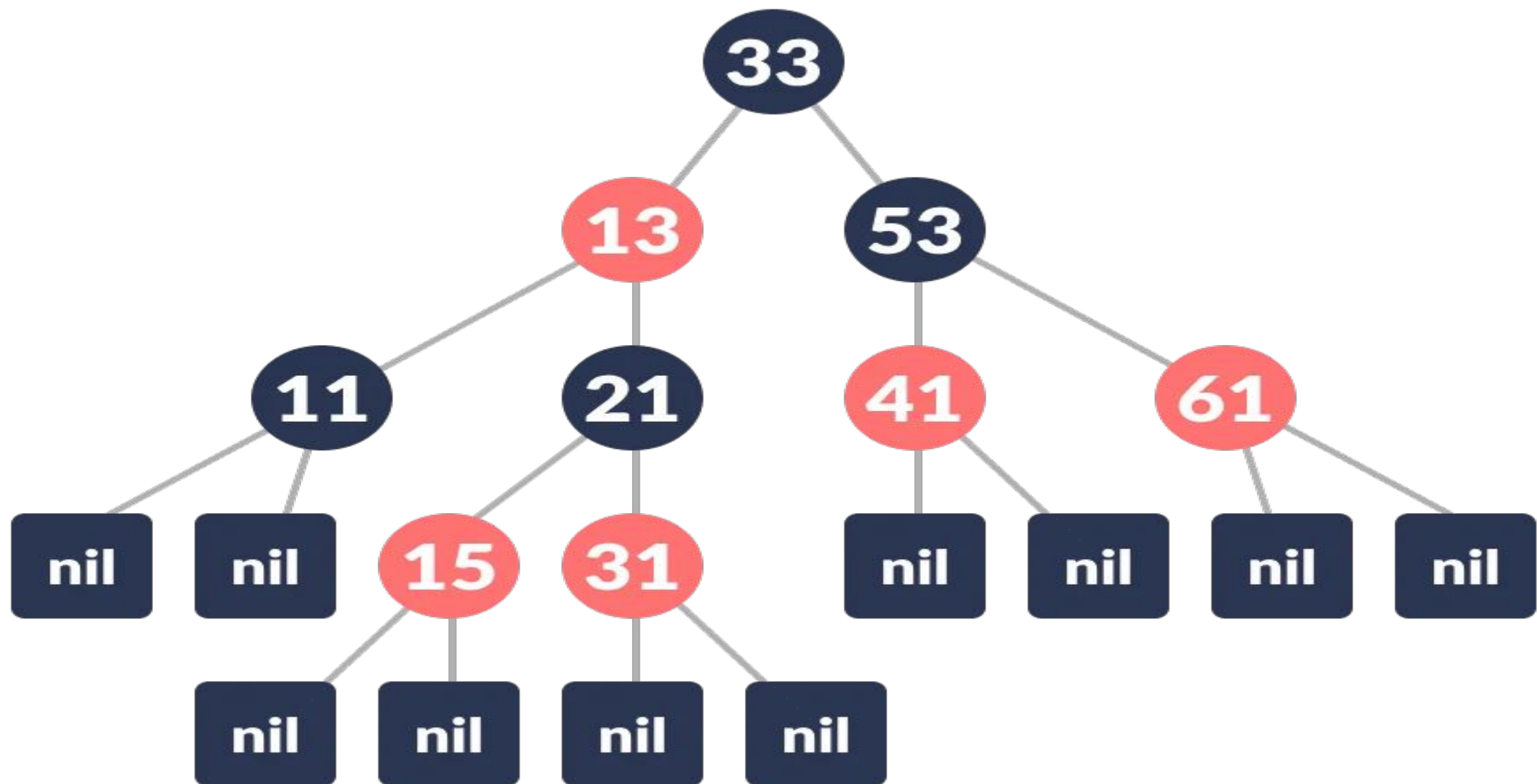
Árvore Red-Black

Cada nó da árvore possui um atributo de cor que pode ser "vermelho" ou "preto"

Propriedades da árvore Rubro-Negra:

- A raiz é sempre "preta"
- Todo nó folha ("NULL") é "preto"
- Se um nó é "vermelho", então os seus filhos são "pretos"
- Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém o mesmo número de nós "pretos"

Árvores Red-Black





```
1 void insertNode(int data)
2 {
3     struct treeNode *ptr = root, *parent_ptr = NULL;
4     while (ptr != NULL)
5     {
6         if (data == ptr->data)
7         {
8             printf("Valores duplicados nao sao permitidos!\n");
9             return;
10        }
11        parent_ptr = ptr;
12        ptr = (data < ptr->data) ? ptr->link[0] : ptr->link[1];
13    }
14    struct treeNode *newNode = createNode(data);
15    newNode->parent = parent_ptr;
16    if (parent_ptr == NULL)
17        root = newNode;
18    else if (data < parent_ptr->data)
19        parent_ptr->link[0] = newNode;
20    else
21        parent_ptr->link[1] = newNode;
22    insertionFixUp(newNode);
23 }
```

```
1 struct treeNode
2 {
3     int data;
4     int color;
5     struct treeNode *link[2];
6     struct treeNode *parent;
7 };
```



```
1 void insertionFixUp(struct treeNode *ptr)
2 {
3     while (ptr->parent != NULL && ptr->parent->color == RED)
4     {
5         struct treeNode *parent_ptr = ptr->parent;
6         struct treeNode *grand_parent_ptr = ptr->parent->parent;
7
8         // Se o pai for o filho da esquerda do avô
9         if (parent_ptr == grand_parent_ptr->link[0])
10        {
11            struct treeNode *uncle_ptr = grand_parent_ptr->link[1];
12
13            // Se o tio for vermelho (o pai também é vermelho), troca as cores
14            // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
15            if (uncle_ptr != NULL && uncle_ptr->color == RED)
16            {
17                grand_parent_ptr->color = RED;
18                parent_ptr->color = BLACK;
19                uncle_ptr->color = BLACK;
20                ptr = grand_parent_ptr;
21            }
```

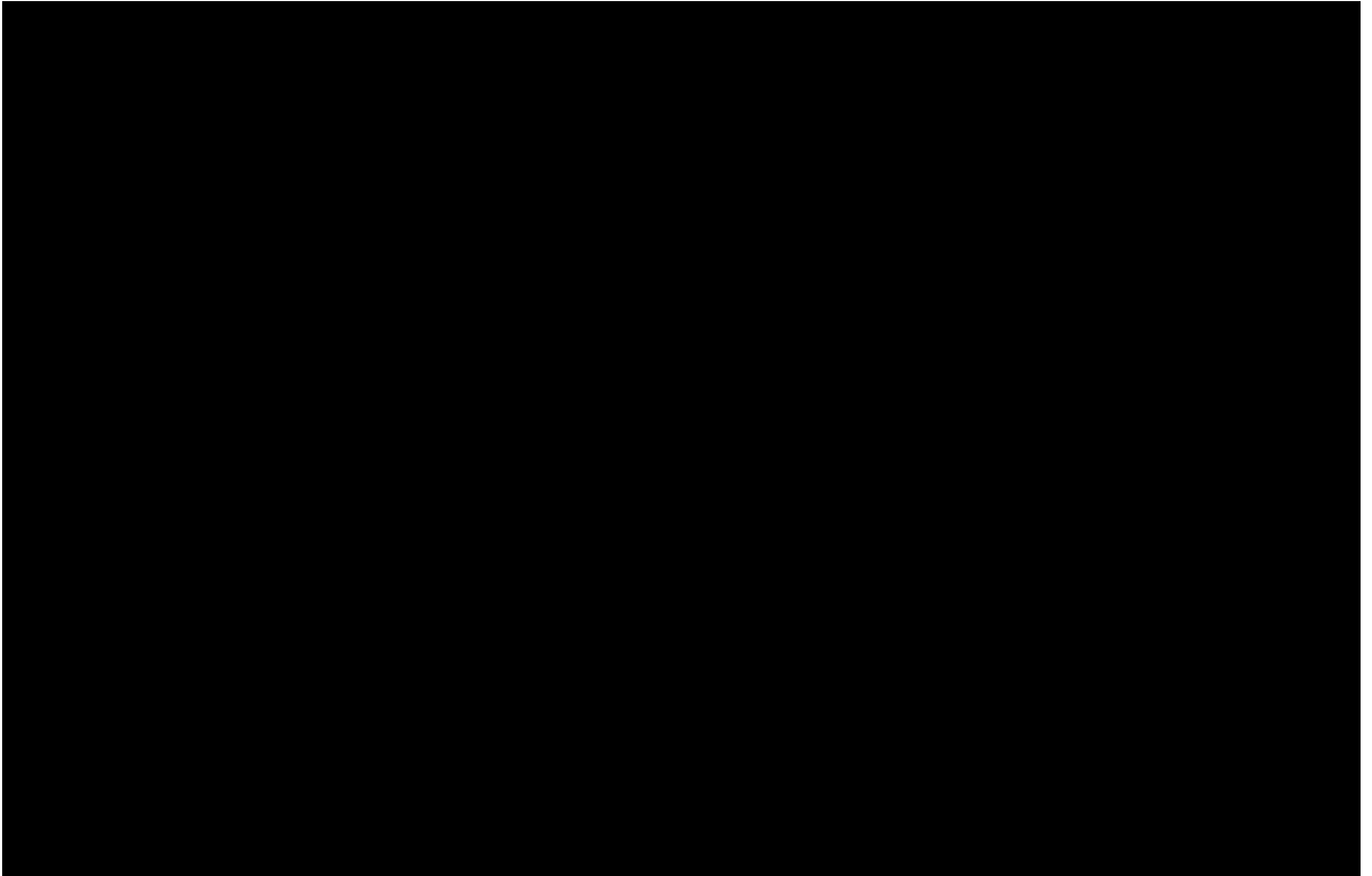



```
1     else
2     {
3         // Se o nó for o filho da direita do pai, faz uma rotação para a esquerda
4         // Isso mantém a propriedade 4 (todos os caminhos de um nó para os nós folhas
5         // descendentes contém o mesmo número de nós "pretos")
6         if (ptr == parent_ptr->link[1])
7         {
8             leftRotate(parent_ptr);
9             ptr = parent_ptr;
10            parent_ptr = ptr->parent;
11        }
12        // Faz uma rotação para a direita no avô e troca as cores do pai e do avô
13        // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
14        rightRotate(grand_parent_ptr);
15        swap(&(parent_ptr->color), &(grand_parent_ptr->color));
16        ptr = parent_ptr;
17    }
18 }
```

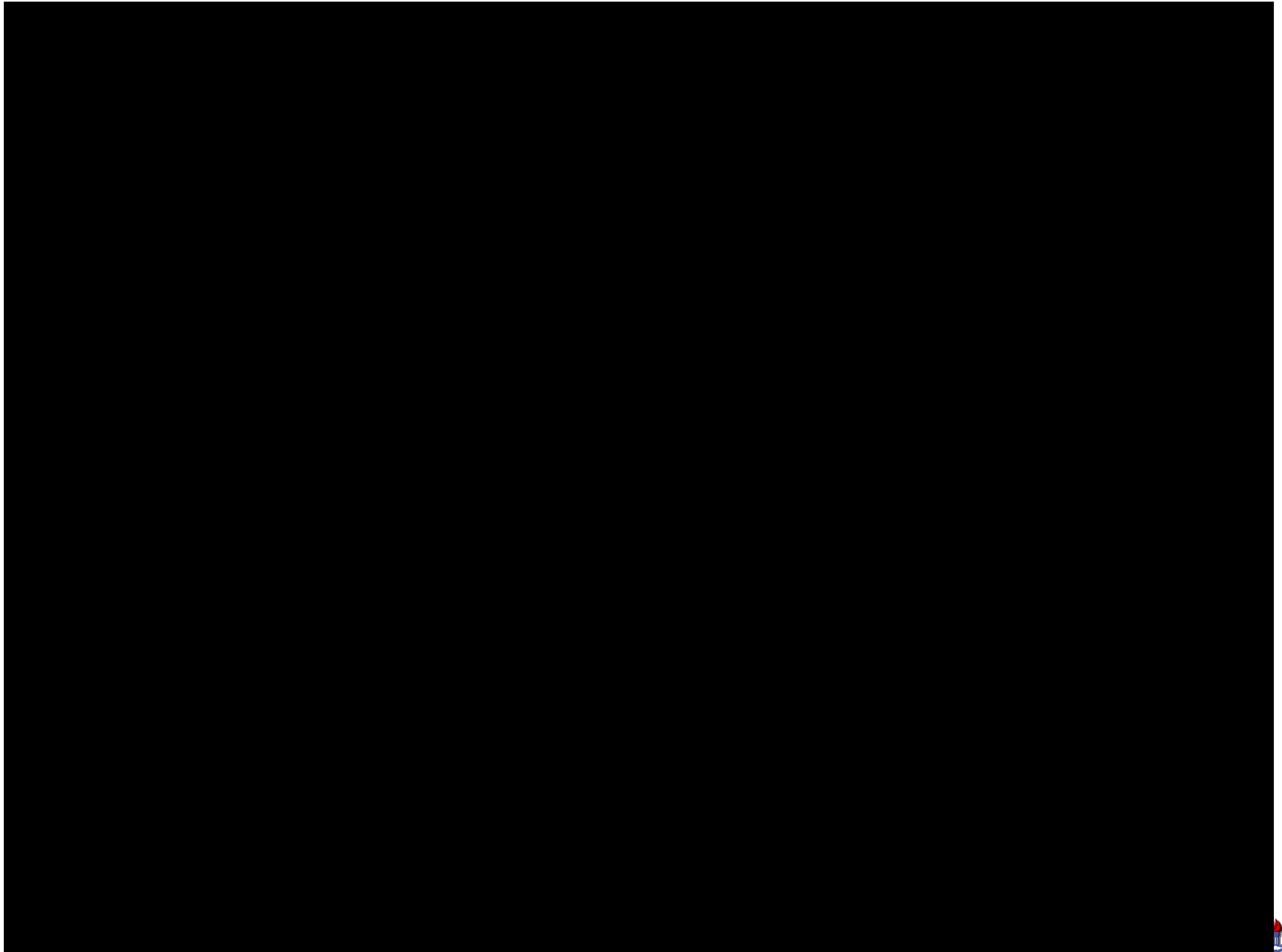


```
1  else
2  {
3      struct treeNode *uncle_ptr = grand_parent_ptr->link[0];
4
5      // Se o tio for vermelho (o pai também é vermelho), troca as cores
6      // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
7      if (uncle_ptr != NULL && uncle_ptr->color == RED)
8      {
9          grand_parent_ptr->color = RED;
10         parent_ptr->color = BLACK;
11         uncle_ptr->color = BLACK;
12         ptr = grand_parent_ptr;
13     }
14     else
15     {
16         // Se o nó for o filho da esquerda do pai, faz uma rotação para a direita
17         // Isso mantém a propriedade 4 (todos os caminhos de um nó para os nós folhas descendentes contém o mesmo número de nós "pretos")
18         if (ptr == parent_ptr->link[0])
19         {
20             rightRotate(parent_ptr);
21             ptr = parent_ptr;
22             parent_ptr = ptr->parent;
23         }
24         // Faz uma rotação para a esquerda no avô e troca as cores do pai e do avô
25         // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
26         leftRotate(grand_parent_ptr);
27         swap(&(parent_ptr->color), &(grand_parent_ptr->color));
28         ptr = parent_ptr;
29     }
30 }
31 }
32 // A raiz é sempre "preta" (propriedade 1)
33 root->color = BLACK;
34 }
```

Animação de inserção



Animação de remoção



Vantagens das Árvores Red-Black:

1. Balanceamento Eficiente:

- As árvores Red-Black mantêm o equilíbrio de forma eficiente.
- Garantem que a altura da árvore seja proporcional ao logaritmo do número de nós.
- Isso resulta em operações de busca, inserção e remoção com tempo médio $O(\log n)$.

2. Acesso a Prefixos:

- As árvores Red-Black permitem acesso a prefixos, o que é crucial para a busca eficiente de palavras.
- Essa característica é especialmente útil em aplicações como dicionários, autocorreção e indexação de palavras.

3. Ampla Utilização:

- São amplamente utilizadas em sistemas e tecnologias para organizar dados de forma eficiente.
- Encontram aplicação em bancos de dados, compiladores, sistemas de arquivos e muito mais.

Desvantagens das Árvores Red-Black:

1. Complexidade de Implementação:

- Embora não sejam tão complexas quanto algumas outras estruturas, a implementação correta das árvores Red-Black ainda requer atenção aos detalhes.
- O cumprimento das propriedades (como cores e balanceamento) pode ser desafiador.

2. Não Tão Rígidas Quanto as AVL Trees:

- As árvores Red-Black são menos rígidas em manter o balanceamento perfeito do que as árvores AVL.
- Isso pode ser uma desvantagem em cenários onde o balanceamento extremamente preciso é necessário.