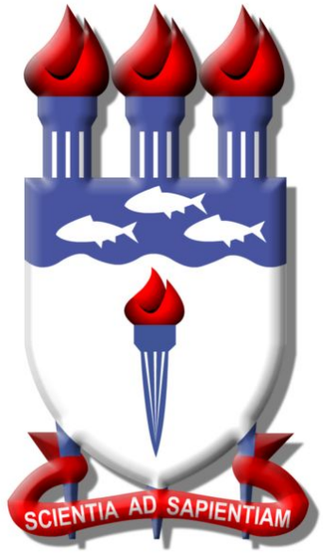


Árvores Red-Black

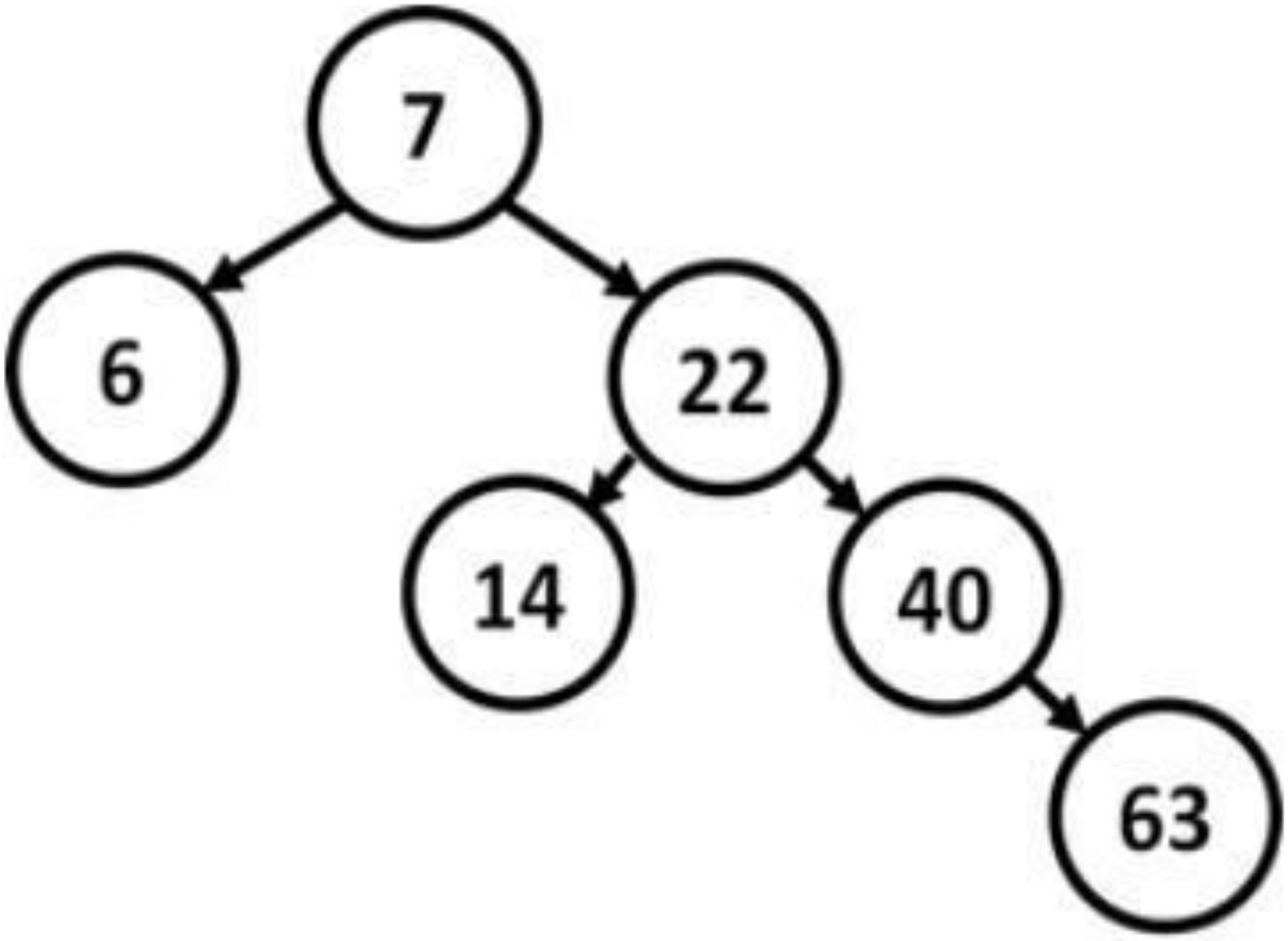
Eduardo Maciel | Eliezir Moreira | Josenilton Ferreira
Lucas Cassiano | Maria Letícia

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?

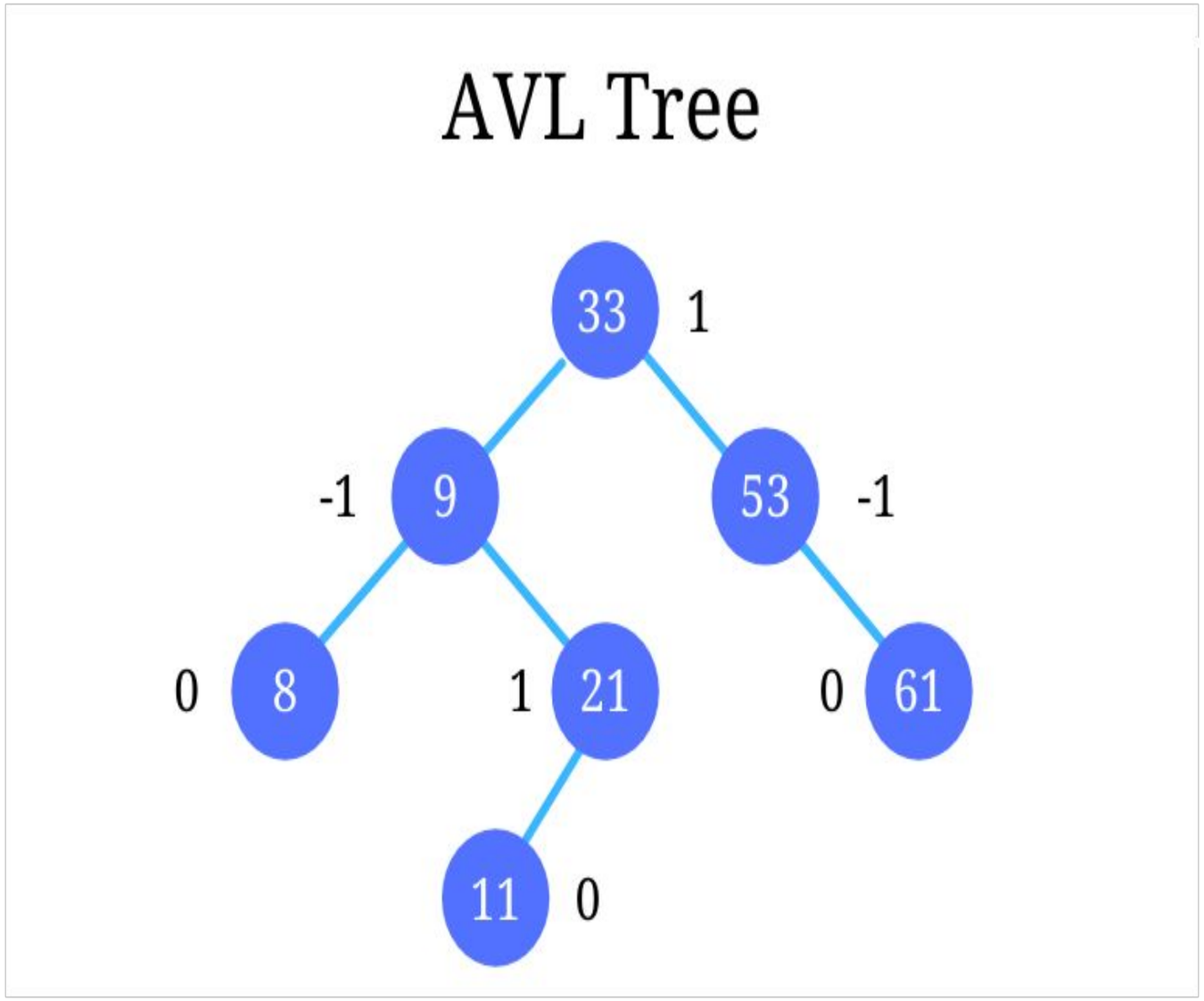


Problemas da BST:

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

- Não há garantia de balanceamento.

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?

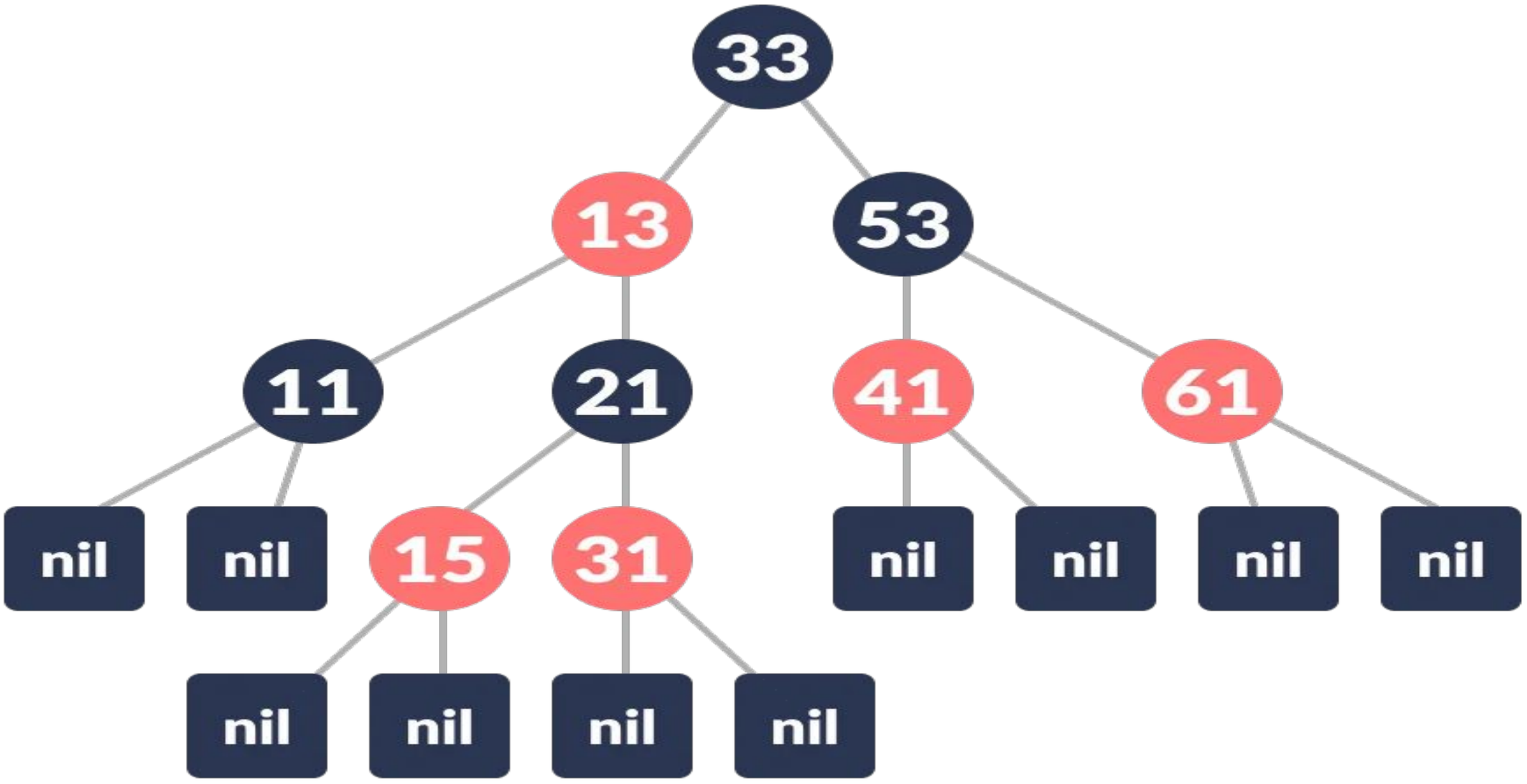
Árvores Red-Black

- 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)**

Árvores Red-Black



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"**

```c
void insertNode(int data)
{
    struct treeNode *ptr = root, *parent_ptr = NULL;
    while (ptr != NULL)
    {
        if (data == ptr->data)
        {
            printf("Valores duplicados nao sao permitidos!\n");
            return;
        }
        parent_ptr = ptr;
        ptr = (data < ptr->data) ? ptr->link[0] : ptr->link[1];
    }
    struct treeNode *newNode = createNode(data);
    newNode->parent = parent_ptr;
    if (parent_ptr == NULL)
        root = newNode;
    else if (data < parent_ptr->data)
        parent_ptr->link[0] = newNode;
    else
        parent_ptr->link[1] = newNode;
    insertionFixUp(newNode);
}
```

```c
void insertionFixUp(struct treeNode *ptr)
{
    while (ptr->parent != NULL && ptr->parent->color == RED)
    {
        struct treeNode *parent_ptr = ptr->parent;
        struct treeNode *grand_parent_ptr = ptr->parent->parent;

        // Se o pai for o filho da esquerda do avô
        if (parent_ptr == grand_parent_ptr->link[0])
        {
            struct treeNode *uncle_ptr = grand_parent_ptr->link[1];

            // Se o tio for vermelho (o pai também é vermelho), troca as cores
            // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
            if (uncle_ptr != NULL && uncle_ptr->color == RED)
            {
                grand_parent_ptr->color = RED;
                parent_ptr->color = BLACK;
                uncle_ptr->color = BLACK;
                ptr = grand_parent_ptr;
            }
```

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

```
else
        {
            struct treeNode *uncle_ptr = grand_parent_ptr->link[0];

            // Se o tio for vermelho (o pai também é vermelho), troca as cores
            // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
            if (uncle_ptr != NULL && uncle_ptr->color == RED)
            {
                grand_parent_ptr->color = RED;
                parent_ptr->color = BLACK;
                uncle_ptr->color = BLACK;
                ptr = grand_parent_ptr;
            }
            else
            {
                // Se o nó for o filho da esquerda do pai, faz uma rotação para a direita
                // Isso mantém a propriedade 4 (todos os caminhos de um nó para os nós folhas descendentes contém o mesmo número de nós "pretos")
                if (ptr == parent_ptr->link[0])
                {
                    rightRotate(parent_ptr);
                    ptr = parent_ptr;
                    parent_ptr = ptr->parent;
                }
                // Faz uma rotação para a esquerda no avô e troca as cores do pai e do avô
                // Isso mantém a propriedade 3 (se um nó é "vermelho", então os seus filhos são "pretos")
                leftRotate(grand_parent_ptr);
                swap(&(parent_ptr->color), &(grand_parent_ptr->color));
                ptr = parent_ptr;
            }
        }
    }
    // A raiz é sempre "preta" (propriedade 1)
    root->color = BLACK;
}
```

Animação de inserção

Animação de inserção

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.