Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 16 –Árvore rubro-negra (Remoção e Implementação)

Agenda

- Remoção
- Estrutura (Implementação)
- Rotações (Implementação)
- Inserção (Implementação)

Remoção

- A remoção de um elemento em uma árvore rubro-negra é um processo que envolve duas etapas principais:
  i. Remoção como em uma árvore binária de busca. São três casos (os mesmos discutidos na aula 9):
     a. Remoção de um nó sem filhos (somente remove o nó)
     b. Remoção de um nó com um filho (substitui o nó pelo filho)
     c. Remoção de um nó com dois filhos (substitui pelo menor da subárvore direita)
  ii. Correção das propriedades rubro-negras

Remoção

- A remoção em árvores rubro-negras é mais complexa do que a inserção, pois pode quebrar regras importantes, especialmente a "**altura negra**".
  - Ou seja, a quantidade de nós **negros** da raiz até cada uma das folhas.

- A parte crítica vem após a remoção:
  - Reestabelecer as propriedades rubro-negras.

Remoção

- [**Relembrando**] As regras a serem seguidas da arvore rubro-negra são:
  1. Todo nó é **rubro** (rubro) ou **negro**.
  2. Raiz é sempre **negra**.
  3. Novo nó é sempre **rubro**.
  4. Todo caminho da raiz até algum nó folha terá o mesmo número de nós **negros**.
  5. Nenhum caminho pode ter dois nós **rubros** consecutivos.
  6. NULL é considerado **negro**.

Remoção

- Se um nó negro é removido, isso pode quebrar a quarta regra.
    4. Todo caminho da raiz até algum nó folha terá o mesmo número de nós **negros**.

- Para resolver, introduzimos o conceito de nó "duplamente negro" — um marcador temporário para indicar que um caminho está com um negro a menos.

Remoção

- O nó duplamente negro (ou double black) é um conceito utilizado durante a remoção de nós em uma árvore rubro-negra.

- Representa que existe temporariamente um desequilíbrio na quantidade de nós negros em um caminho da árvore.

Remoção

- Como a estrutura da árvore rubro-negra não permite diretamente um "nó com dois níveis de pretidão", o nó duplamente negro é tratado como um estado lógico.

- Ou seja, um nó comum com uma "marca" indicando que ele está com uma pretidão a mais.

Remoção

- Esse estado não é representado fisicamente como uma nova cor, mas sim como uma condição especial que exige ações corretivas, como recolorações e rotações, para restaurar as propriedades da árvore.

- Esse mecanismo é essencial para garantir que, mesmo após a remoção de um nó negro, a árvore continue válida e equilibrada conforme suas regras.

Remoção

Durante esse processo de correção, o nó duplamente negro pode:

- "Subir" na árvore,
- Transferir o excesso de pretidão para outro nó,
- Ou ser eliminado ao final da reestruturação.

- O caso de remoção da árvore rubro-negra é um dos mais complicados.
  - Como o intuito da disciplina é fornecer uma visão geral da disciplina, nós **NÃO** veremos a remoção em detalhes.

Estrutura

```c
typedef struct no {
    int info;
    int cor;
    struct no *esq;
    struct no *dir;
    struct no *pai;
} Node;
```

- Com relação a estrutura, utilizaremos a mesma estrutura padrão da árvore binária de busca acrescido de dois campos:
  - **cor**, para indicar se é rubro ou negro
  - **pai**, um ponteiro para o pai do nó

Estrutura

```c
void inicializar_NIL() {
    NIL = (Node *)malloc(sizeof(Node));
    NIL->cor = BLACK;
    NIL->esq = NULL;
    NIL->dir = NULL;
    NIL->pai = NULL;
}
```

- Para a árvore rubro-negra, introduziremos o conceito de NIL.
  - É chamado de NIL o nó “sentinela” em árvores rubro-negras
  - É um nó real na memória, ao contrário de NULL.
  - Substitui o uso de NULL para facilitar a implementação e manter as propriedades da árvore rubro-negra.

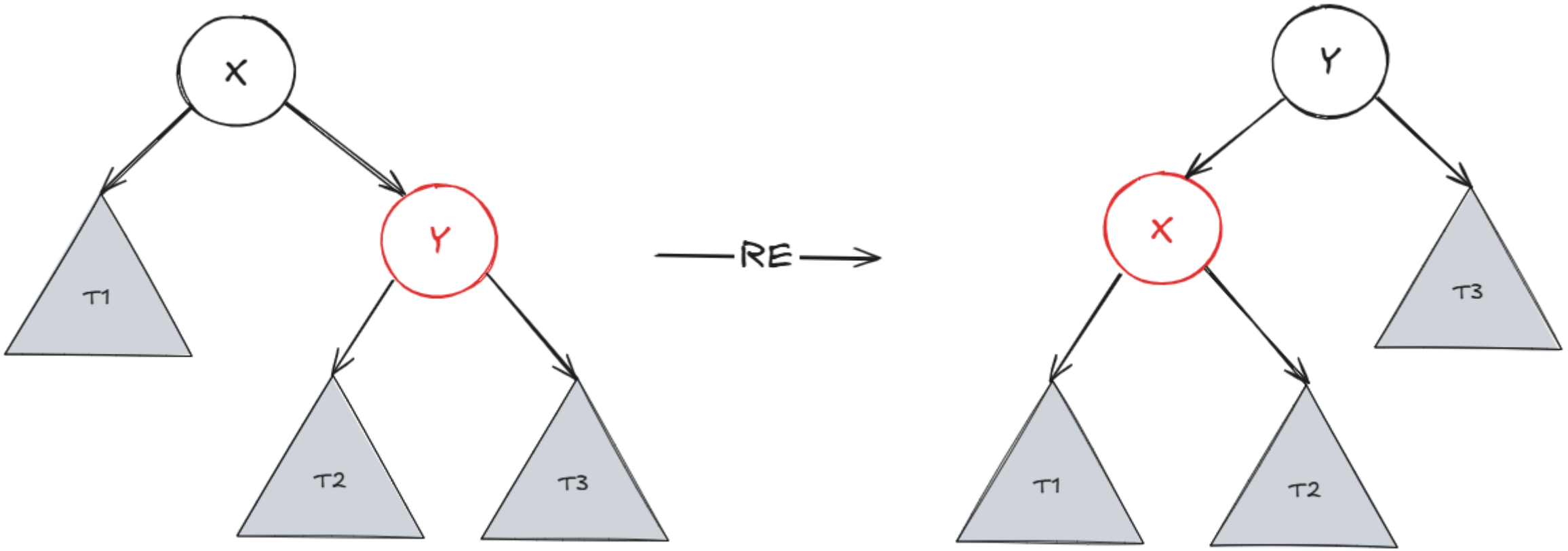
Estrutura

```c
void inicializar_NIL() {
    NIL = (Node *)malloc(sizeof(Node));
    NIL->cor = BLACK;
    NIL->esq = NULL;
    NIL->dir = NULL;
    NIL->pai = NULL;
}
```

- Nesta inicialização, vamos alocar a memória para NIL e definir a sua cor como sendo negra (BLACK).

- Posteriormente, vamos definir seu pai e seus filhos à esquerda e à direita como nulo.

Rotações



Rotações

- A função rotacao_esquerda() realiza uma rotação à esquerda em torno de um nó x em uma árvore rubro-negra, alterando a estrutura da árvore para manter seu balanceamento.

- O nó y, filho direito de x, sobe para a posição de x, e x passa a ser o filho esquerdo de y.

- Durante esse processo, o filho esquerdo de y (caso exista e não seja o nó sentinela NIL) se torna o novo filho direito de x, e os ponteiros de pais e filhos são atualizados adequadamente para manter a coerência da árvore.

```c
void rotacao_esquerda(Node **raiz, Node *x) {
    Node *y = x->dir;
    x->dir = y->esq;
    if (y->esq != NIL) {
        y->esq->pai = x;
    }
    y->pai = x->pai;
    if (x->pai == NULL) {
        *raiz = y;
    } else if (x == x->pai->esq) {
        x->pai->esq = y;
    } else {
        x->pai->dir = y;
    }
    y->esq = x;
    x->pai = y;
}
```

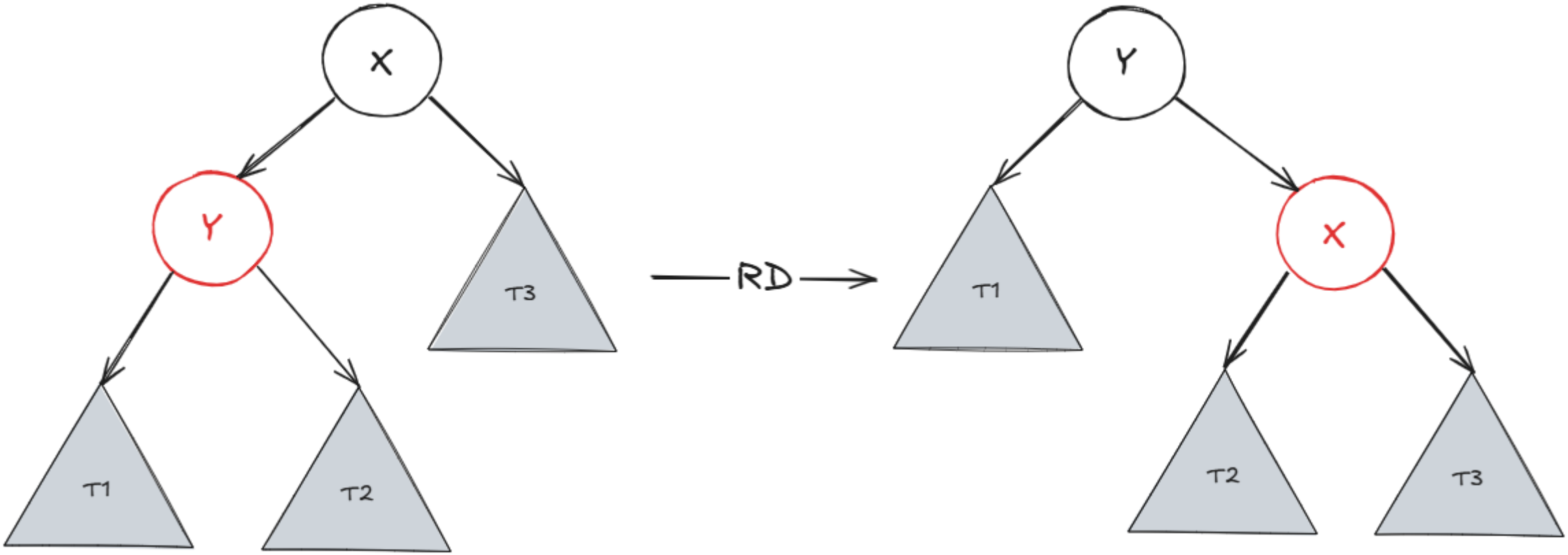
Rotações

- Se x for a raiz da árvore, o ponteiro da raiz é atualizado para apontar para y.

- Essa operação é essencial em árvores rubro-negras para corrigir desequilíbrios após inserções e remoções, especialmente para garantir as propriedades de ordenação e balanceamento da estrutura.

```c
void rotacao_esquerda(Node **raiz, Node *x) {
    Node *y = x->dir;
    x->dir = y->esq;
    if (y->esq != NIL) {
        y->esq->pai = x;
    }
    y->pai = x->pai;
    if (x->pai == NULL) {
        *raiz = y;
    } else if (x == x->pai->esq) {
        x->pai->esq = y;
    } else {
        x->pai->dir = y;
    }
    y->esq = x;
    x->pai = y;
}
```

Rotações



Rotações

- A função rotacao_direita() realiza uma rotação à direita em torno de um nó x em uma árvore rubro-negra, ajustando a estrutura da árvore para manter seu balanceamento.

- Nessa operação, o filho esquerdo de x (denominado y) sobe para ocupar o lugar de x, e x se torna o filho direito de y.

- O filho direito de y, se existir (isto é, se não for o nó sentinela NIL), passa a ser o novo filho esquerdo de x, e o ponteiro pai desse nó é atualizado para apontar para x.

```c
void rotacao_direita(Node **raiz, Node *x) {
    Node *y = x->esq;
    x->esq = y->dir;
    if (y->dir != NIL) {
        y->dir->pai = x;
    }
    y->pai = x->pai;
    if (x->pai == NULL) {
        *raiz = y;
    } else if (x == x->pai->dir) {
        x->pai->dir = y;
    } else {
        x->pai->esq = y;
    }
    y->dir = x;
    x->pai = y;
}
```

Rotações

- Em seguida, os ponteiros dos pais são ajustados: y passa a ter o mesmo pai que x tinha, e o ponteiro correspondente do pai de x (se existir) é atualizado para apontar para y.

- Por fim, y->dir aponta para x, e x->pai passa a ser y.

- Essa operação é usada para corrigir desequilíbrios à esquerda e preservar as propriedades de ordenação e altura da árvore.

```c
void rotacao_direita(Node **raiz, Node *x) {
    Node *y = x->esq;
    x->esq = y->dir;
    if (y->dir != NIL) {
        y->dir->pai = x;
    }
    y->pai = x->pai;
    if (x->pai == NULL) {
        *raiz = y;
    } else if (x == x->pai->dir) {
        x->pai->dir = y;
    } else {
        x->pai->esq = y;
    }
    y->dir = x;
    x->pai = y;
}
```

Inserção

- A função corrigir_insercao() é responsável por restaurar as propriedades de uma árvore rubro-negra após a inserção de um novo nó z, já que a simples inserção pode violar a regra de que um nó vermelho não pode ter um pai vermelho.

- O algoritmo verifica repetidamente se o pai de z é vermelho, o que indica violação da propriedade da árvore.

```c
void corrigir_insercao(Node **raiz, Node *z) {
    while (z->pai && z->pai->cor == RED) {
        if (z->pai == z->pai->pai->esq) {
            Node *tio = z->pai->pai->dir;
            if (tio->cor == RED) {
                z->pai->cor = BLACK;
                tio->cor = BLACK;
                z->pai->pai->cor = RED;
                z = z->pai->pai;
            } else {
                if (z == z->pai->dir) {
                    z = z->pai;
                    rotacao_esquerda(raiz, z);
                }
                z->pai->cor = BLACK;
                z->pai->pai->cor = RED;
                rotacao_direita(raiz, z->pai->pai);
            }
        } else {
            Node *tio = z->pai->pai->esq;
            if (tio->cor == RED) {
                z->pai->cor = BLACK;
                tio->cor = BLACK;
                z->pai->pai->cor = RED;
                z = z->pai->pai;
            } else {
                if (z == z->pai->esq) {
                    z = z->pai;
                    rotacao_direita(raiz, z);
                }
                z->pai->cor = BLACK;
                z->pai->pai->cor = RED;
                rotacao_esquerda(raiz, z->pai->pai);
            }
        }
    }
    (*raiz)->cor = BLACK;
}
```

Inserção

- Se for, ele identifica o tio de z (o outro filho do avô de z) e aplica uma das seguintes estratégias:
  - Se o tio for vermelho, é um caso de recoloração (o pai e o tio ficam negros, o avô fica vermelho, e a verificação continua a partir do avô);
  - Se o tio for negro, é um caso de rotação: primeiro uma rotação interna (esquerda ou direita, dependendo da posição de z), seguida por uma rotação externa no avô, com ajustes de cor para manter o balanceamento.

- O código trata simetricamente os casos em que o pai de z é filho esquerdo ou direito do avô. Ao final, garante-se que a raiz seja negra, conforme exigido pelas regras da árvore rubro-negra.

```c
void corrigir_insercao(Node **raiz, Node *z) {
    while (z->pai && z->pai->cor == RED) {
        if (z->pai == z->pai->pai->esq) {
            Node *tio = z->pai->pai->dir;
            if (tio->cor == RED) {
                z->pai->cor = BLACK;
                tio->cor = BLACK;
                z->pai->pai->cor = RED;
                z = z->pai->pai;
            } else {
                if (z == z->pai->dir) {
                    z = z->pai;
                    rotacao_esquerda(raiz, z);
                }
                z->pai->cor = BLACK;
                z->pai->pai->cor = RED;
                rotacao_direita(raiz, z->pai->pai);
            }
        } else {
            Node *tio = z->pai->pai->esq;
            if (tio->cor == RED) {
                z->pai->cor = BLACK;
                tio->cor = BLACK;
                z->pai->pai->cor = RED;
                z = z->pai->pai;
            } else {
                if (z == z->pai->esq) {
                    z = z->pai;
                    rotacao_direita(raiz, z);
                }
                z->pai->cor = BLACK;
                z->pai->pai->cor = RED;
                rotacao_esquerda(raiz, z->pai->pai);
            }
        }
    }
    (*raiz)->cor = BLACK;
}
```

Inserção

- A função inserir() implementa a inserção de um novo valor em uma árvore rubro-negra, respeitando as regras de uma árvore binária de busca e, posteriormente, corrigindo possíveis violações das propriedades rubro-negras.

- Primeiro, ela aloca e inicializa um novo nó z com o valor fornecido, cor vermelha, e ponteiros de filhos apontando para o nó sentinela NIL.

```
void inserir(Node **raiz, int valor) {
    Node *z = (Node *) malloc(sizeof(Node));
    z->info = valor;
    z->cor = RED;
    z->esq = NIL;
    z->dir = NIL;

    Node *y = NULL;
    Node *x = *raiz;

    while (x != NIL) {
        y = x;
        if (valor < x->info) {
            x = x->esq;
        } else {
            x = x->dir;
        }
    }

    z->pai = y;
    if (y == NULL) {
        *raiz = z;
    } else if (valor < y->info) {
        y->esq = z;
    } else {
        y->dir = z;
    }
    corrigir_insercao(raiz, z);
}
```

Inserção

- Em seguida, ela percorre a árvore a partir da raiz (*raiz) para encontrar a posição correta onde o novo nó deve ser inserido, atualizando os ponteiros x e y para manter o controle do caminho percorrido.

- Depois de encontrar a posição, z é ligado como filho esquerdo ou direito de y, dependendo da comparação entre os valores. Caso y seja NULL, o novo nó é a raiz da árvore.

- Por fim, a função chama corrigir_insercao() para aplicar as rotações e recolorações necessárias, garantindo que a estrutura da árvore rubro-negra continue válida após a inserção.

```c
void inserir(Node **raiz, int valor) {
    Node *z = (Node *) malloc(sizeof(Node));
    z->info = valor;
    z->cor = RED;
    z->esq = NIL;
    z->dir = NIL;

    Node *y = NULL;
    Node *x = *raiz;

    while (x != NIL) {
        y = x;
        if (valor < x->info) {
            x = x->esq;
        } else {
            x = x->dir;
        }
    }

    z->pai = y;
    if (y == NULL) {
        *raiz = z;
    } else if (valor < y->info) {
        y->esq = z;
    } else {
        y->dir = z;
    }
    corrigir_insercao(raiz, z);
}
```

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 16 –Árvore rubro-negra (Remoção e Implementação)