Faculdade de Educação Tecnológica do Estado do Rio de Janeiro

Análise e Desenvolvimento de Sistemas

Ezio Chifunga

Guilherme Lima

Lucas Sena

Sérgio da Silva

Estruturas Árvores

Árvore Rubro-negra

Rio de Janeiro

2023

Trabalho apresentado no curso

de graduação da Faculdade

de Educação Tecnológica do

Estado do Rio de Janeiro

Professor(a): Claudia Ferlin, Doutora.

Rio de Janeiro

2023

Disciplina: 3 ESD

Turno: Manhã

Período: 2023.1

Trabalho: Apresentar áreas de aplicação da árvore rubro-negra, sua estrutura e operações básicas com algumas simulações de inclusões e exclusões.

ÁRVORES

I - Introdução

Nesta apresentação, discutiremos a Árvore Binária Rubro-Negra, uma estrutura de dados especializada em manter árvores binárias balanceadas. Será explicado como ocorrem os processos de inserção, remoção e busca nessa estrutura. Contudo, faz-se necessário esclarecer alguns aspectos gerais da estrutura da árvore.

II - Definição

É uma estrutura de dados que se caracteriza por uma relação de hierarquia entre os elementos que a compõem.

É organizada como uma coleção não vazia de vértices e ramos que satisfazem certos requisitos, modela a hierarquia existente entre os elementos e é baseada em recursividade.

São utilizadas para representar estruturas hierárquicas, tais como: hierarquia de pastas, árvore genealógica, organograma de uma empresa e etc…

II.I - Representação

Cada nó da árvore possui: A informação e dois ponteiros para as sub-árvores, uma à esquerda e uma à direita.

Implementação em C:

```
struct arv {
    int info;
    struct arv* esq;
    struct arv* dir;
};
```

II.II - Considerações:

- Numa árvore binária de busca com n chaves e de altura h, as operações de busca, inserção e remoção têm complexidade de tempo O(h).
- No pior caso, a altura de uma árvore binária de busca pode ser O(n). No caso médio, vimos que a altura é O(log n).
- Árvores AVL, árvores 2-3 e árvores rubro-negras são alguns tipos de árvores binárias de busca ditas balanceadas com altura O(log n).
- Todas essas árvores são projetadas para busca de dados armazenados na memória principal (RAM).
- As árvores 2-3 são generalizadas para as chamadas B-árvores, para busca eficiente de dados armazenados em memória secundária (disco rígido).
- A árvore rubro-negra possui altura máxima igual a 2 log(n + 1).

ÁRVORE RUBRO NEGRA

I - Definição

É um tipo de árvore binária balanceada(ABB). Se diferencia da AVL uma vez que, cada nó possui uma cor (rubro ou negro) e é a partir dessas cores que irá ocorrer o balanceamento nesta árvore.

Originalmente criada por Rudolf Bayer em 1972, sendo nomeada como “Árvores Binárias Simétricas”, posteriormente, em um trabalho de Leonidas J. Guibas e Robert Sedgewick de 1978, ela adquiriu o seu nome atual.

I.II - Representação

Cada nó da árvore possui: A informação, uma cor, um ponteiro para o nó pai, um ponteiro para o nó filho esquerdo e um ponteiro para o nó filho direito.

Implementação em C:

```
typedef struct Node {

    int info;
    int cor;
    struct Node* pai;
    struct Node* esquerdo;
    struct Node* direito;

} Node;
```

II - Propriedades:

- Nó da árvore possui um atributo de cor que pode ser **vermelho** ou **preto**.
- A raiz é sempre preta.
- Todo nó folha (“NULL”) é preto.
- Se um nó é vermelho, então os seus filhos são pretos, ou seja, não existem nós vermelhos consecutivos.
- Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém o mesmo número de nós **pretos**.
- Como todo nó folha termina com dois ponteiros para “NULL”, eles podem ser ignorados na representação gráfica da árvore, para fins de didática.

II.I - Comparações com a Árvore AVL

- Ambas apresentam complexidade logarítmica (O (log n)) para qualquer operação (inserção, remoção ou busca);

- Enquanto na AVL, a altura de duas subárvores irmãs diferem no máximo em 1, na Rubro-Negra, a altura de uma subárvore pode ser até o dobro da altura da sua irmã.
- Enquanto na AVL, a diferença das alturas das subárvores irmãs é o fator de balanceamento dos nós, na Rubro-Negra, o fator do balanceamento é definido pelas cores dos nós.

II.II - Balanceamento

- Permite o rebalanceamento local.
  - ❖ Apenas a parte afetada pela **inserção** e/ou **remoção** é re-balanceada.
  - ❖ Uso de **rotações** e ajuste de cores na etapa de re-balanceamento.
  - ❖ Essas operações corrigem as propriedades que foram violadas.

- A árvore rubro-negra busca manter-se como uma árvore binária quase completa.
  - ❖ O custo de qualquer algoritmo é máximo “O log(n)”.

II.III - Performance:

- Rubro-negra versus AVL

  - ❖ Na teoria, possuem a mesma complexidade computacional (inserção, remoção e busca), “O log(n)”.
  - ❖ Na prática, a árvore AVL é mais rápida na operação de busca e mais lenta nas operações de inserção e remoção.
  - ❖ A árvore AVL é mais balanceada do que a rubro-negra, o que acelera a operação de busca.

- ❖ Maior custo nas operações de inserção e remoção, no pior caso, uma operação de remoção pode exigir “O log(n)” rotações na árvore AVL, mas apenas três rotações na árvore rubro-negra.

III - Operações

A árvore Rubro-negra, assim como as demais árvores, suportam algumas operações que são necessárias à manipulação de conjuntos de dados, tais como: inserção, remoção e busca, desde que sejam mantidas suas propriedades balanceadas.

III.I - Inserção

- Permite adicionar um novo elemento à árvore. Durante a inserção, o algoritmo realiza rotações e re-colorações para garantir que a árvore seja balanceada e atenda às propriedades da árvore rubro-negra.
- Abaixo o passo a passo para a inserção de um novo nó à árvore.

  - ❖ Se a raiz é nula, insira o nó.
  - ❖ Cada novo nó, por definição possui cor rubro;
  - ❖ Se o valor a ser inserido for menor que o valor da raiz, vá para a subárvore esquerda;
  - ❖ Se o valor a ser inserido for maior que o valor da raiz, vá para a subárvore direita (a inserção é feita exatamente igual em uma ABB).
  - ❖ Após a inserção, verifique se as propriedades da Rubro-Negra ainda se mantêm;
  - ❖ A raiz da árvore é sempre negra;

- **1º caso** - Se o pai do novo nó inserido for NEGRO, todas as propriedades se mantêm;

- **2º Caso** - Se o pai do novo nó inserido for RUBRO, rotações ou alterações de cor precisam ser feitas (2º Caso).
  - **Regra 1 -** O pai e o tio são rubros:

    1 - O pai e o tio ficam Negros.

    2 - o Avô fica Rubro. Se o avô for a raiz da árvore, ele fica preto.

    3 - Se o Avô não for a raiz e seu pai for rubro, trata-se o avô como novo nó e verifica-se toda a regra novamente.

  - **Regra 2 -** O pai é rubro e o tio é negro (Este caso se subdivide em quatro novos casos):
    - **sub-regra 1 (rotação simples para direita):** O pai do novo nó é filho esquerdo e novo nó é filho esquerdo.

      1 - Pai fica Negro;

      2 - Avô fica Rubro;

      3 - Rotaciona o avô para a direita.

    - **sub-regra 2 (rotação dupla para direita):** O pai do novo nó é filho esquerdo e o novo nó é filho direito.

      1 - Rotaciona o pai para a esquerda;

      2 - novo nó = filho esquerdo do novo nó (depois da rotação o antigo pai passa a ser filho do novo nó);

3 - Pai fica Negro;

4 - Avô fica Rubro;

5 - Rotaciona o avô para a direita;

- **sub-regra 3 (rotação simples para esquerda):** O pai do novo nó é filho direito e novo nó é filho direito.

  1 - Pai fica Negro;

  2 - Avô fica Rubro;

  3 - Rotaciona o avô para a esquerda.

- **sub-regra 4 (rotação dupla para esquerda):** O pai do novo nó é filho direito e o novo nó é filho esquerdo.

  1 - Rotaciona o pai para a direita;

  2 - novo nó = filho esquerdo do novo nó (depois da rotação o antigo pai passa a ser filho do novo nó);

  3 - Pai fica Negro;

  4 - Avô fica Rubro;

  5 - Rotaciona o avô para a esquerda;

III.I.i - Código de Inserção

```
#include <stdio.h>

#include <stdlib.h>
```

```
// Estrutura de um nó da árvore
typedef struct Node {
    int data;           // Dado do nó
    int color;          // Cor do nó: 0 para vermelho, 1 para preto
    struct Node *parent;  // Ponteiro para o nó pai
    struct Node *left;    // Ponteiro para o nó filho esquerdo
    struct Node *right;   // Ponteiro para o nó filho direito
} Node;

// Função auxiliar para criar um novo nó
Node *createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = data;
    newNode->color = 0;  // Novos nós sempre são vermelhos
    newNode->parent = NULL;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
// Função auxiliar para trocar a cor dos nós
void swapColors(Node **node1, Node **node2) {
    int tempColor = (*node1)->color;
    (*node1)->color = (*node2)->color;
    (*node2)->color = tempColor;
}

// Função auxiliar para rotacionar à esquerda
void rotateLeft(Node **root, Node *node) {
    Node *rightChild = node->right;
    node->right = rightChild->left;

    if (node->right != NULL)
        node->right->parent = node;

    rightChild->parent = node->parent;

    if (void rotateRight(Node **root, Node *node) {
    Node *leftChild = node->left;
    node->left = leftChild->right;
```

```
    if (node->left != NULL)
        node->left->parent = node;

    leftChild->parent = node->parent;

    if (node->parent == NULL)
        *root = leftChild;
    else if (node == node->parent->left)
        node->parent->left = leftChild;
    else
        node->parent->right = leftChild;

    leftChild->right = node;
    node->parent = leftChild;
}
node->parent == NULL)
        *root = rightChild;
    else if (node == node->parent->left)
        node->parent->left = rightChild;
    else
        node->parent->right = rightChild;
```

```
    rightChild->left = node;
    node->parent = rightChild;
}

// Função auxiliar para rotacionar à direita
void rotateRight(Node **root, Node *node) {
Node *leftChild = node->left;
node->left = leftChild->right;
if (node->left != NULL)
        node->left->parent = node;
leftChild->parent = node->parent;
if (node->parent == NULL)
        *root = leftChild;
 else if (node == node->parent->left)
        node->parent->left = leftChild;
        else node->parent->right = leftChild;

        leftChild->right = node;
        node->parent = leftChild; }
```

```
// Função auxiliar para corrigir a árvore após a inserção
void fixInsertion(Node **root, Node *node) {
    while (node != *root && node->parent->color == 0) {  // Enquanto o pai for vermelho
        if (node->parent == node->parent->parent->left) {
            Node *uncle = node->parent->parent->right;

            if (uncle != NULL && uncle->color == 0) {
                // Caso 1: Tio é vermelho
                node->parent->color = 1;
                uncle->color = 1;
                node->parent->parent->color = 0;
                node = node->parent->parent;
            } else {
                if (node == node->parent->right) {
                    // Caso 2: Tio é preto e o nó é filho direito
                    node = node->parent;
                    rotateLeft(root, node);
                }

                // Caso 3: Tio é preto e o nó é filho esquerdo
```

```
            node->parent->color = 1;
            node->parent->parent->color = 0;
            rotateRight(root, node->parent->parent);
        }
    } else {
        Node *uncle = node->parent->parent->left;

        if (uncle != NULL && uncle->color == 0) {
            // Caso 1: Tio é vermelho
            node->parent->color = 1;
            uncle->color = 1;
            node->parent->parent->color = 0;
            node = node->parent->parent;
        } else {
            if (node == node->parent->left) {
                // Caso 2: Tio é preto e o nó é filho esquerdo
                node = node->parent;
                rotateRight(root, node);
            }

            // Caso 3: Tio é preto e o nó é filho direito
```

```
                node->parent->color = 1;
                node->parent->parent->color = 0;
                rotateLeft(root, node->parent->parent);
            }
        }
    }
    (*root)->color = 1;  // A raiz sempre deve ser preta
}

// Função para inserir um nó na árvore rubro-negra
void insertNode(Node **root, int data) {
    Node *newNode = createNode(data);
    Node *current = *root;
    Node *parent = NULL;

    while (current != NULL) {
        parent = current;

        if (newNode->data < current->data)
            current = current->left;
        else
```

```
            current = current->right;
        }

        newNode->parent = parent;

        if (parent == NULL)
            *root = newNode;
        else if (newNode->data < parent->data)
            parent->left = newNode;
        else
            parent->right = newNode;

        fixInsertion(root, newNode);
    }

    // Função auxiliar para imprimir a árvore em ordem
    void inOrderTraversal(Node *root) {
        if (root != NULL) {
            inOrderTraversal(root->left);
            printf("%d ", root->data);
            inOrderTraversal(root->right);
```

```
    }
}

// Função principal
int main() {
    Node *root = NULL;

    insertNode(&root, 10);
    insertNode(&root, 20);
    insertNode(&root, 30);
    insertNode(&root, 15);
    insertNode(&root, 18);
    insertNode(&root, 25);

    printf("Árvore em ordem: ");
    inOrderTraversal(root);
    printf("\n");

    return 0;
}
```

III.II - Remoção (exclusão)

- Permite remover um elemento da árvore. Durante a exclusão, o algoritmo realiza rotações e re-colorações para garantir que a árvore seja balanceada e atenda às propriedades da árvore rubro-negra.
- O procedimento para remoção de um nó é o mesmo utilizado na árvore ABB, contudo após a remoção, deve-se verificar se as propriedades da árvore rubro-negra se mantêm.
- As situações que podem ocorrer na remoção de elementos são quatro.

Situação	Nó removido (dado)	Sucessor
1	RUBRO	RUBRO
2	NEGRO	RUBRO
3	NEGRO	NEGRO
4	RUBRO	NEGRO

- **1ª Situação - Remoção de um nó Rubro e o sucessor é Rubro** -
  - Basta colocar o sucessor no lugar do nó a ser removido.

- **2ª Situação - Remoção de um nó Negro e o sucessor é Rubro**
  - Substitui o sucessor pelo nó a ser removido e pinte o sucessor de negro.

- **3ª Situação - Remoção de um nó Negro e o sucessor é Negro**

- **Caso 1 - Se o nó a ser removido(v) é Negro e seu irmão(w) é Rubro e seu pai(p) é Negro. Marque ele com um duplo negro e :**
  - Faça uma rotação simples à esquerda.
  - Pinte o irmão(w) de negro.
  - Pinte o pai(p) de rubro.

- **Caso 2a - Se o nó a ser removido(v) é Negro e seu irmão(w) é Negro, com filhos Negros e seu pai(p) é Negro.**
  - Pinte o Irmão(w) de rubro.
- **Caso 2b - Se o nó a ser removido(v) é Negro e seu irmão(w) é Negro, com filhos Negros e seu pai(p) é Rubro.**
  - Pinte o irmão(w) de rubro e o pai(p) do nó a ser removido(v) de negro.
- **Caso 3 - Se o nó a ser removido(v) é Negro e seu irmão(w) é Negro, tem pai de qualquer cor (rubro ou negro), tem irmão(w) com filho esquerdo rubro e filho direito negro.**
  - Rotação simples à direita no irmão(w).
  - Trocar as cores do irmão(w) com seu filho esquerdo.
- **Caso 4 - Se o nó a ser removido(v) é Negro e seu irmão(w) é Negro, tem pai de qualquer cor (rubro ou negro), tem irmão(w) com filho esquerdo de qualquer cor(rubro ou negro) e filho direito rubro.**
  - Rotação simples à esquerda no irmão(w).
  - Pinte o pai(p) de negro, caso seja rubro.
  - Pinte o irmão(w) na cor anterior do pai(p).
  - Pinte o filho direito do irmão(w) de negro.

- **4ª Situação - Remoção de um nó Rubro e o sucessor é Negro**
  - Pinte o sucessor(x) de rubro.
  - Proceda conforme a 3ª situação.

III.II.i - Código de Remoção

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int key;
    int color; // 0 for black, 1 for red
    struct Node* parent;
    struct Node* left;
    struct Node* right;
} Node;

// Função auxiliar para criar um novo nó
Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->color = 1; // Novos nós são sempre vermelhos
    newNode->parent = NULL;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Função auxiliar para fazer a rotação à esquerda
```

```
void leftRotate(Node** root, Node* x) {
    Node* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        (*root) = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

// Função auxiliar para fazer a rotação à direita
void rightRotate(Node** root, Node* y) {
    Node* x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == NULL)
        (*root) = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}
```

```
// Função auxiliar para balancear a árvore após a remoção
void fixDelete(Node** root, Node* x) {
    while (x != (*root) && x->color == 0) {
        if (x == x->parent->left) {
            Node* w = x->parent->right;
            if (w->color == 1) {
                w->color = 0;
                x->parent->color = 1;
                leftRotate(root, x->parent);
                w = x->parent->right;
            }
            if (w->left->color == 0 && w->right->color == 0) {
                w->color = 1;
                x = x->parent;
            } else {
                if (w->right->color == 0) {
                    w->left->color = 0;
                    w->color = 1;
                    rightRotate(root, w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = 0;
                w->right->color = 0;
                leftRotate(root, x->parent);
                x = (*root);
            }
        } else {
            Node* w = x->parent->left;
            if (w->color == 1) {
```

```
            w->color = 0;
            x->parent->color = 1;
            rightRotate(root, x->parent);
            w = x->parent->left;
        }
        if (w->right->color == 0 && w->left->color == 0) {
            w->color = 1;
            x = x->parent;
        } else {
            if (w->left->color == 0) {
                w->right->color = 0;
                w->color = 1;
                leftRotate(root, w);
                w = x->parent->left;
            }
            w->color = x->parent->color;
            x->parent->color = 0;
            w->left->color = 0;
            rightRotate(root, x->parent);
            x = (*root);
        }
    }
  }
  x->color = 0;
}

// Função auxiliar para encontrar o nó mínimo
Node* minimumNode(Node* node) {
  while (node->left != NULL)
    node = node->left;
  return node;
}
```

```
// Função auxiliar para substituir um nó por outro
void replaceNode(Node** root, Node* x, Node* y) {
    if (x->parent == NULL)
        (*root) = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    if (y != NULL)
        y->parent = x->parent;
}

// Função auxiliar para remover um nó da árvore
void deleteNode(Node** root, int key) {
    Node* z = createNode(key);
    Node* x, *y;
    if (z == NULL)
        return;

    // Encontrar o nó a ser removido
    Node* node = (*root);
    while (node != NULL) {
        if (z->key < node->key)
            node = node->left;
        else if (z->key > node->key)
            node = node->right;
        else
            break;
    }
    if (node == NULL) {
```

```
    printf("O nó %d não existe na árvore.\n", key);
    return;
}

y = node;
int yOriginalColor = y->color;
if (node->left == NULL) {
    x = node->right;
    replaceNode(root, node, node->right);
} else if (node->right == NULL) {
    x = node->left;
    replaceNode(root, node, node->left);
} else {
    y = minimumNode(node->right);
    yOriginalColor = y->color;
    x = y->right;
    if (y->parent == node)
        x->parent = y;
    else {
        replaceNode(root, y, y->right);
        y->right = node->right;
        y->right->parent = y;
    }
    replaceNode(root, node, y);
    y->left = node->left;
    y->left->parent = y;
    y->color = node->color;
}
free(node);

if (yOriginalColor == 0)
    fixDelete(root, x);
```

```
}

// Função auxiliar para imprimir a árvore rubro-negra em ordem
void inOrderTraversal(Node* node) {
    if (node != NULL) {
        inOrderTraversal(node->left);
        printf("%d ", node->key);
        inOrderTraversal(node->right);
    }
}

// Função auxiliar para imprimir a árvore rubro-negra
void printTree(Node* root) {
    printf("Árvore Rubro-Negra (in-order): ");
    inOrderTraversal(root);
    printf("\n");
}

int main() {
    Node* root = NULL;

    // Exemplo de inserção de nós
    root = createNode(7);
    root->color = 0; // Raiz sempre é preta
    Node* node1 = createNode(3);
    Node* node2 = createNode(18);
    Node* node3 = createNode(10);
    Node* node4 = createNode(22);
    Node* node5 = createNode(8);
    Node* node6 = createNode(11);
    Node* node7 = createNode(26);
    Node* node8 = createNode(2);
```

```
Node* node9 = createNode(6);
Node* node10 = createNode(13);

root->left = node1;
root->right = node2;
node1->parent = root;
node2->parent = root;
node1->left = node3;
node1->right = node4;
node3->parent = node1;
node4->parent = node1;
node3->left = node5;
node3->right = node6;
node5->parent = node3;
node6->parent = node3;
node4->right = node7;
node7->parent = node4;
node5->left = node8;
node5->right = node9;
node8->parent = node5;
node9->parent = node5;
node6->right = node10;
node10->parent = node6;

printTree(root);

// Exemplo de remoção de nós
deleteNode(&root, 8);
deleteNode(&root, 18);

printTree(root);
```

```
    return 0;
}
```

III.III - Busca

- Permite procurar um elemento específico na árvore rubro-negra. A busca é feita comparando o elemento desejado com os elementos presentes nos nós da árvore, seguindo o caminho apropriado com base nas comparações até encontrar o elemento desejado ou chegar a um nó folha.
- A estrutura de uma Árvore Rubro-Negra não afeta o processo de busca, pois as propriedades de busca de uma árvore binária de busca são preservadas. A propriedade de ordenação da árvore (valores menores à esquerda e valores maiores à direita) ainda é válida.
- Existem várias maneiras de percorrer (ou ler) uma Árvore Rubro-Negra, assim como em qualquer outra árvore binária. As três principais formas são: pré-ordem (preorder), em ordem (inorder) e pós-ordem (postorder). Além dessas, também é possível percorrer em largura (ou nível a nível) utilizando uma abordagem de busca em largura.
- Observe como cada um desses métodos de funciona em uma Árvore Rubro-Negra:

    - 1- Pré-ordem (preorder):
        - Visite o nó atual.
        - Percorra recursivamente a subárvore esquerda.
        - Percorra recursivamente a subárvore direita.

    - 2- Em ordem (inorder):
        - Percorra recursivamente a subárvore esquerda.
        - Visite o nó atual.
        - Percorra recursivamente a subárvore direita.

    - 3- Pós-ordem (postorder):
        - Percorra recursivamente a subárvore esquerda.

        - Percorra recursivamente a subárvore direita.
        - Visite o nó atual.
- Essas três abordagens são chamadas de percorrimento em profundidade (depth-first traversal), pois percorrem a árvore de forma a explorar as subárvores o mais a fundo possível antes de voltar.

- Além desses métodos, também é possível percorrer a Árvore Rubro-Negra em largura (level order traversal), que segue uma abordagem de busca em largura, explorando os nós por níveis. Nesse caso, você visitaria os nós da árvore em cada nível antes de prosseguir para o próximo nível.

- Cada método de percorrimento oferece uma maneira diferente de visualizar a estrutura da árvore e pode ser escolhido com base nas necessidades específicas do problema em questão.

- Abaixo o passo a passo para a exclusão de um novo nó da árvore.

    - Comece na raiz da árvore.
    - Compare o valor que você está procurando com o valor do nó atual.
    - Se o valor for igual ao valor do nó atual, a busca foi bem-sucedida e o nó foi encontrado. Retorne o valor nó.
    - Se o valor for menor que o valor do nó atual, vá para o nó filho esquerdo e repita o passo 2.
    - Se o valor for maior que o valor do nó atual, vá para o nó filho direito e repita o passo 2.
    - Se chegar a um nó nulo (folha), isso significa que o valor procurado não está presente na árvore. Retorne um valor indicando a ausência.

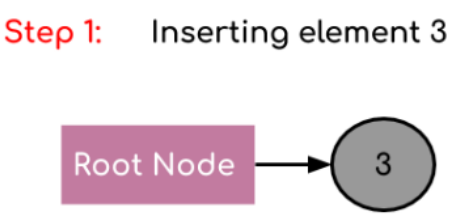
III.III.i -

```
// Função para buscar um nó na árvore rubro-negra
struct Node *searchNode(struct Node *root, int data) {
if (root == NULL || root->data == data)
        return root;

        if (root->data < data)
                return searchNode(root->right, data);

                return searchNode(root->left, data); }
```

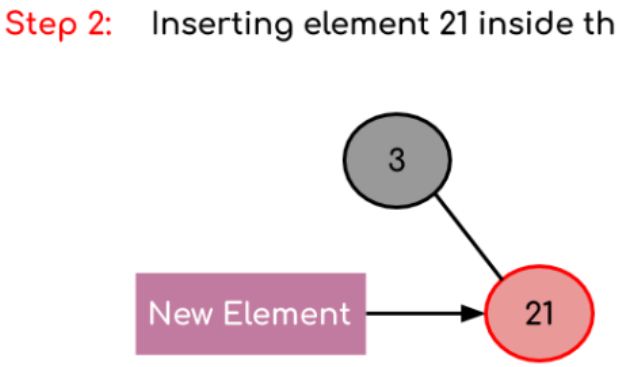
IV - Simulação

Exemplo: Criar uma árvore rubro-negra com os elementos 3, 21, 32 e 15 em uma árvore vazia.

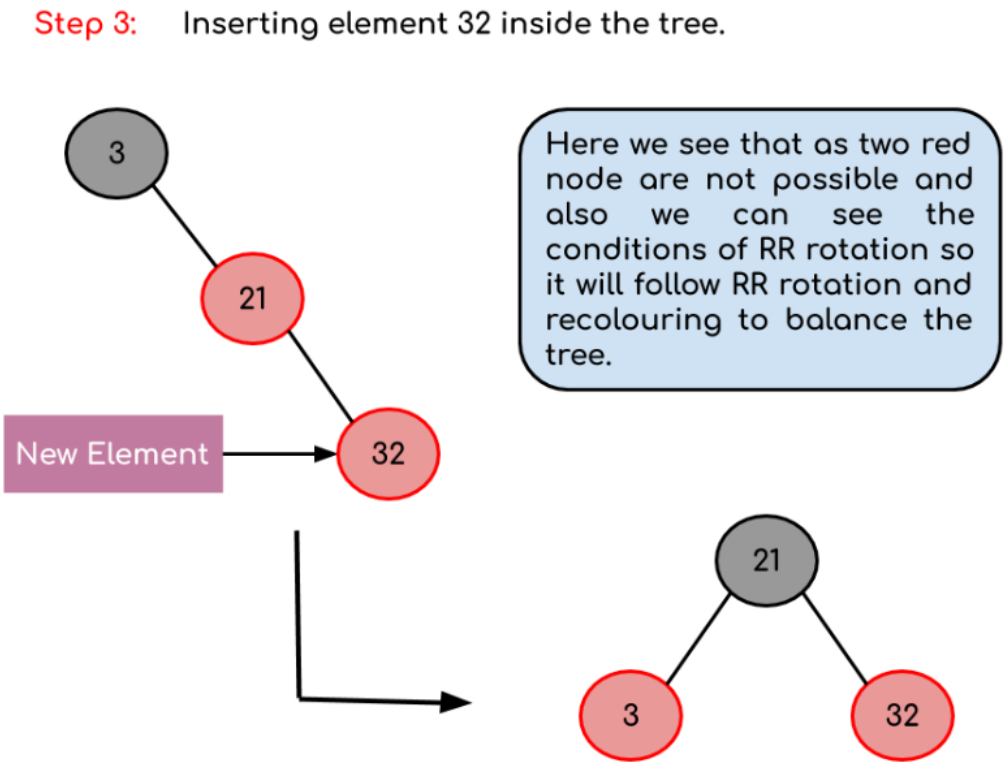
Solução:



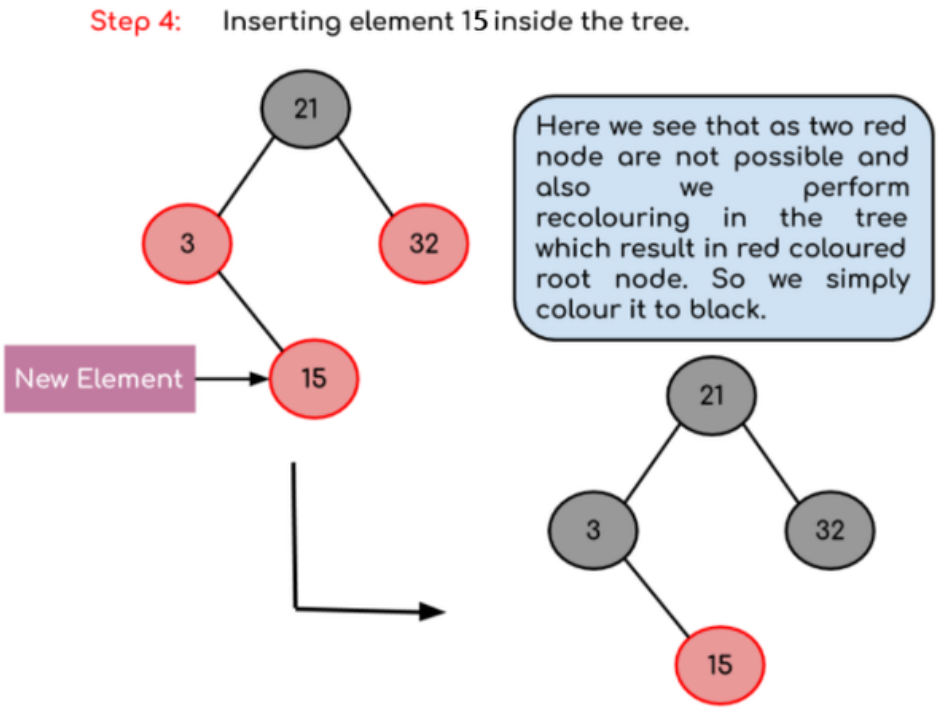
Quando o primeiro elemento é inserido ele é inserido como nó raiz e como nó raiz tem a cor preta então adquira a cor preta.



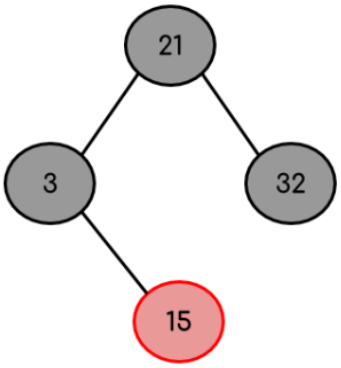
O novo elemento é sempre inserido com a cor vermelha e como 21 > 3 passa a fazer parte da subárvore direita do nó raiz.



Agora, ao inserirmos 32, vemos que há um par pai-filho vermelho que viola a regra da árvore rubro-negra, então temos que girá-lo. Além disso, vemos as condições de rotação RR (considerando o nó nulo do nó raiz como preto), então após a rotação como o nó raiz não pode ser vermelho, então temos que recolorir a árvore criada na árvore mostrada acima.



Estrutura final da árvore:



IV.I - Utilização:

- A estrutura de uma Árvore Rubro-Negra não afeta o processo de busca, pois as propriedades de busca de uma árvore binária de busca são preservadas. A propriedade de ordenação da árvore (valores menores à esquerda e valores maiores à direita) ainda é válida.
- As vantagens da Árvore Rubro-Negra em relação a uma árvore binária de busca padrão estão relacionadas ao balanceamento da árvore. Como a altura da árvore é mantida em um limite proporcional ao logaritmo do número de nós, a busca em uma Árvore Rubro-Negra é eficiente e tem complexidade de tempo O(log n), onde "n" é o número de nós na árvore.
- Se a aplicação realiza de forma intensa a operação de busca, é melhor usar uma árvore AVL.
- Se a operação mais usada é a de inserção ou de remoção, use a árvore rubro-negra.

V - Aplicações:

- Árvores rubro-negras são de uso mais geral do que as árvores AVL. Sendo utilizadas em diversas aplicações e bibliotecas de linguagens de programação:

    - Dicionários ordenados (úteis para armazenar pares de chave-valor).
    - Estruturas de dados persistentes.
    - Banco de dados.
    - Sistema de arquivos.
    - JAVA: java.util.TreeMap, java.util.TreeSet.
    - C++ STL: map, multmap, multset.
    - Linux Kernel: completely fair scheduler, linux/rbttree.h.

VI - Conclusão

A Árvore Binária Rubro-Negra é uma estrutura de dados eficiente para manter árvores binárias balanceadas, pois é uma estrutura de dados versátil que oferece um bom desempenho em operações de busca, inserção e remoção. Ela é amplamente utilizada em uma variedade de aplicações, como: Dicionários ordenados, bancos de dados e sistemas de arquivos, onde a eficiência é essencial.

Referências:

MC202 - Estrutura de Dados, Autor: Alexandre Xavier Falcão, Instituto de Computação - UNICAMP, Disponível em:

Algoritmos e estrutura de dados III, Autor: Carlos Oberdan Rolim, Ciência da Computação - ……., Disponível em:

Estruturas de dados: Árvores, Disponível em: http://www.inf.ufes.br/~pdcosta/ensino/2012-1-estruturas-de-dados/slides/Aula15%20%28arvores%29.pdf

Árvore Binária - wikilivros, Disponível em: https://pt.wikibooks.org/wiki/Programar_em_C/%C3%81rvores_bin%C3%A1rias

Árvore Rubro-negra, Autor: Siang Wun Song, Universidade de São Paulo - IME/USP, Disponível em: https://www.ime.usp.br/~song/mac5710/slides/08rb.pdf

Árvores Rubro-Negras, Autor

http://www.ulysseso.com/livros/ed2/ApF.pdf