CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

ANTONIELLY BERGAMI RIBEIRO
WILLIAN PACHECO SILVA

RELATÓRIO: ÁRVORE RUBRO-NEGRA

SÃO MATEUS
2021

ANTONIELLY BERGAMI RIBEIRO
WILLIAN PACHECO SILVA

RELATÓRIO: ÁRVORE RUBRO-NEGRA

Relatório apresentado à Disciplina de Estrutura de dados II dos cursos de bacharelado em Ciência e Engenharia da Computação 2020/2 do Centro Universitário Norte do Espírito Santo, como requisito parcial para avaliação .
Orientador: Luciana Lee.

SÃO MATEUS
2021

RESUMO

Este relatório faz parte da avaliação parcial da disciplina de Estrutura de dados II dos cursos de Bacharelado em Ciência e Engenharia da Computação do Centro Universitário Norte do Espírito Santo.

Sua finalidade é apresentar os resultados das implementações das funções de uma árvore rubro-negra que armazena estruturas do tipo “Artigo” já definido pela professora.

Na árvore foram implementados as funções padrões: inserir, remover e buscar, e também foram implementadas funções auxiliares para realizar as rotações. Além disso, foi criado um arquivo main em que foi definida uma estrutura para aplicar o código da biblioteca ARN na estrutura do tipo “Artigo” e um menu para auxiliar o usuário a escolher a opção que ele deseja realizar na árvore. Também foi criado o arquivo .h, para conter o cabeçalho das funções, e o makefile para que seja possível realizar a compilação do programa por meio do make.

SUMÁRIO

1 INTRODUÇÃO

Uma árvore rubro-negra é um tipo de árvore binária de busca balanceada (estrutura de dados baseada em nós) com um bit extra de armazenamento em cada nó. Esse bit serve para armazenar a cor do nó, que pode ser vermelha ou preta. Essa árvore insere e remove de forma inteligente assegurando que a árvore permaneça balanceada, que cada nó seja da cor certa e que nenhuma propriedade da árvore rubro-negra seja violada. Essa árvore restringe as cores dos nós no caminho da raiz até uma folha para assegurar que o comprimento de nenhum dos caminhos que ligam a raiz a uma folha seja mais do que duas unidades maior do que qualquer outro caminho que liga a raiz até uma folha.

Vale ressaltar que as árvores rubro-negras obedecem algumas propriedades, são elas:

1. Todo nó é vermelho ou preto
2. A raiz é preta
3. Toda folha é preta
4. Se um nó é vermelho, então seus filhos são pretos
5. Para cada nó, todos os caminhos simples do nó até as folhas descendentes contêm o mesmo número de nós pretos.

Este relatório apresenta os resultados encontrados após a implementação da árvore rubro-negra com as funções de busca, inserção e remoção, além de outras funções como, por exemplo, a função de impressão da árvore e a função para liberar a memória alocada.

2 OBJETIVOS

O objetivo do trabalho é apresentar os conhecimentos adquiridos nas aulas de estruturas de dados sendo aplicados na implementação da árvore rubro negra de acordo com o conteúdo aprendido em sala de aula e com pesquisas complementares demonstrando os métodos utilizados e a ordem de serviço para a perfeita execução da atividade proposta.

3 METODOLOGIA

Os resultados encontrados no procedimento da realização do estudo sobre uma árvore rubro-negra foram feitos através de um programa desenvolvido pelos alunos na linguagem de programação C.

3.1 LINGUAGEM C

C é uma linguagem de programação compilada de propósito geral, estruturada, imperativa, procedural, padronizada pela Organização Internacional para Padronização (ISO), criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs para desenvolvimento do sistema operacional Unix (originalmente escrito em Assembly). C é uma das linguagens de programação mais populares e existem poucas arquiteturas para as quais não existem compiladores para C. C tem influenciado muitas outras linguagens de programação (por exemplo, a linguagem Java), mais notavelmente C++, que originalmente começou como uma extensão para C.

4 APRESENTAÇÃO DOS RESULTADOS E DISCUSSÕES

Para alcançarmos o objetivo proposto, que é a implementação da árvore rubro-negra armazenando dados do tipo “Artigo”, criamos o arquivo de cabeçalho, o arquivo do cliente e o makefile para melhor organização e execução do programa.

4.1 ARN.h

Começando pelo arquivo de cabeçalho, “ARN.h”. Nesse arquivo é possível que o cliente tenha uma ideia geral das funções que foram implementadas na biblioteca. Ele contém a declaração da estrutura, inclusão das bibliotecas, definição dos protótipos das funções, a descrição de cada uma delas e o seu retorno. Após sua criação, o arquivo ARN.h passa a ser uma biblioteca que depois será incluída no arquivo ARN.c e no arquivo do cliente que for utilizar a biblioteca criada. No caso deste trabalho, o arquivo cliente é o “main.c”. As imagens abaixo demonstram como foi implementado o arquivo ARN.h

```c
// Este arquivo se trata de uma biblioteca para uma árvore rubro negra com tipo de dados genérico.
#ifndef __ARN_H__
#define __ARN_H__

//Inclusão das bibliotecas necessárias.
#include <stdio.h>
#include <stdlib.h>

/*A estrutura abaixo é a estrutura da arvore rubro negra. Cada nó da árvore será declarado com esse tipo de dado.
Na estrutura dessa árvore, cada nó tem os campos de esq, dir, pai, chave e cor.
esq: campo do tipo struct arn * que armazena um ponteiro para o filho da esquerda do nó.
dir: campo do tipo struct arn * que armazena um ponteiro para o filho da direita do nó.
pai: campo do tipo struct arn * que armazena o nó pai de um nó.
chave: campo do tipo void * que armazena um dado do tipo genérico.
cor: campo do tipo int que armazena a cor daquele nó, sendo 0 = preto e 1 = vermelho.*/
typedef struct arn
{
    struct arn *esq;
    struct arn *dir;
    struct arn *pai;
    void *chave;
    int cor;
} ARN;

//Definição da variavel global que será o nó sentinela
ARN externo;

/*Função auxiliar utilizada para inicializar a variavel externo. O retorno dela será o endereço da variavel
já inicializada.*/
ARN *criaArvore();

/* Função que cria um nó da árvore. Ela recebe como parâmetro a chave do tipo void * e retorna
o nó da árvore que possui como chave o elemento recebido.*/
ARN *criaNo(void *chave);

/*Função auxiliar utilizada para realizar as rotações para a esquerda. Essa função recebe como parâmetro
um ponteiro para o nó raiz da árvore e um ponteiro para o nó da árvore sobre o qual ocorrerá a rotação.
Com base nisso, ela realiza a rotação para a esquerda e depois retorna um ponteiro para o novo nó raiz da
árvore.*/
ARN *rotacaoEsquerda(ARN *raiz, ARN *x);

/*Função auxiliar utilizada para realizar as rotações para a direita. Essa função recebe como parâmetro
um ponteiro para o nó raiz da árvore e um ponteiro para o nó da árvore sobre o qual ocorrerá a rotação.
Com base nisso, ela realiza a rotação para a direita e depois retorna um ponteiro para o novo nó raiz da
árvore.*/
ARN *rotacaoDireita(ARN *raiz, ARN *x);
```

IMAGEM 1.1: Implementação do arquivo ARN.h

```c
/*Função auxiliar utilizada para realizar as rotações e definições de cores dos nós após a inserção.
Recebe como parâmetro um ponteiro para o nó raiz da árvore e um ponteiro para o nó a ser inserido.
O retorno será um ponteiro para o novo nó raiz.*/
ARN *balanceamentoInsercao(ARN *raiz, ARN *z);

/*Função de inserção de chaves em uma árvore rubro negra. Recebe como parâmetro um ponteiro para o nó raiz da
árvore, a chave a ser inserida, uma função de callback comparaChave que realiza as comparações entre as chaves
e uma função de callback liberaChave que libera a memória alocada caso a chave recebida já exista na árvore.
Essa função retorna um ponteiro para a nova raiz da árvore.*/
ARN *insercao(ARN *raiz, void *chave, int (*comparaChave)(void *, void *), void (*liberaChave)(void *));

/*Função utilizada para transferir um pai de um nó de origem para outro nó de destino. Ela recebe um ponteiro
para a raiz da árvore, um ponteiro para o nó de origem e um ponteiro para o nó de destino. Depois de transferir
o pai, ela retorna um ponteiro para a nova raiz da árvore.*/
ARN *transferePai(ARN *raiz, ARN *origem, ARN *destino);

/*Função auxiliar utilizada para realizar as rotações de definições de cores dos nós após a remoção.
Recebe como parâmetro um ponteiro para o nó raiz da árvore e um ponteiro para o nó a ser removido.
O retorno será um ponteiro para o novo nó raiz.*/
ARN *balanceamentoRemocao(ARN *raiz, ARN *x);

/*Função auxiliar utilizada para encontrar o sucessor de um nó. Essa função recebe como parâmetro
um ponteiro para um nó da árvore e percorre essa árvore até encontrar o sucessor desse nó. O retorno
será o nó sucessor.*/
ARN *sucessor(ARN *a);

/*Função de remoção de chaves em uma árvore rubro negra. Recebe como parâmetro um ponteiro para o nó raiz da
árvore, um ponteiro para o nó a ser removido, uma função de callback comparaChave para realizar as comparações
entre as chaves e uma função de callback liberaChave que libera a memória alocada caso a chave recebida já
exista na árvore. Essa função retorna um ponteiro para a nova raiz da árvore.*/
ARN *remocao(ARN *raiz, ARN *noRemovido, void (*liberaChave)(void *));

/*Função recursiva utilizada para buscar uma chave na árvore. Essa função recebe como parâmetro
um ponteiro para o nó raiz da árvore, um ponteiro para a chave a ser buscada na árvore e uma função
de callback para realizar a comparação entre as chaves para que seja possível encontrar o nó. O retorno
dela é a chave buscada caso o elemento seja encontrado e um ponteiro para o nó "externo" caso o elemento
não esteja na árvore.*/
ARN *buscaChave(ARN *a, void *chave, int (*comparaChave)(void *, void *));
```

IMAGEM 1.2: Implementação do arquivo ARN.h.

```c

/*Função auxiliar utilizada para retornar o campo chave de um nó da árvore. Ela recebe um ponteiro para
um nó da árvore e retorna um ponteiro do tipo void * que contém a chave daquele nó.*/
void *retornaChave(ARN *no);

/*Função recursiva que libera a memória alocada para cada nó da árvore e das respectivas chaves deles.
Ela recebe como parâmetro um ponteiro para o nó raiz da árvore e uma função de callback que realiza
a liberação da memória alocada para cada chave. Essa função não possui retorno.*/
void liberaArvore(ARN *a, void (*liberaChave)(void *));

/*Função recursiva utilizada para imprimir os nós da árvore evidenciando o nível e a cor de cada nó.
Ela recebe como parâmetro um ponteiro para o nó raiz da árvore, uma variável para controlar o nível da
árvore em que o nó se encontra e uma função de callback que realiza a impressão das chaves do tipo de
dado do código do cliente. Essa função não possui retorno.*/
void imprimeArvore(ARN *a, int cont, void (*imprimeChave)(void *));

#endif
```

IMAGEM 1.3: Implementação do arquivo ARN.h.

No ARN.h é possível notar que foram incluídas as bibliotecas necessárias para que o programa funcione. Logo após isso, a estrutura utilizada para a árvore ARN foi implementada e descrita. Depois, os cabeçalhos das funções foram listados. Vale ressaltar que esse arquivo foi bem comentado e possui descrição do que será necessário para o funcionamento de cada função. Isso é importante pois são esses comentários que irão orientar o cliente sobre como utilizar a biblioteca da forma correta.

4.1.1 Estrutura do nó

Nesse arquivo foi feita a implementação da estrutura que será utilizada nos nós da árvore rubro-negra. Essa estrutura é a seguinte:

```
typedef struct arn
{
    struct arn *esq;
    struct arn *dir;
    struct arn *pai;
    void *chave;
    int cor; // 0 negro, 1 rubro
} ARN;
```

IMAGEM 2: Declaração da estrutura do nó.

Essa estrutura possui alguns campos:

Os três primeiros campos da estrutura são campos do tipo ponteiro para ela mesma. Isso quer dizer que esses campos são ponteiros e que irão apontar para outras estruturas do tipo ARN. Isso é importante pois eles são os ponteiros que irão apontar para as subárvores esquerdas e direitas de cada nó e também para o nó pai desse nó.

O quarto campo é o campo que vai armazenar a chave do nó. Como a implementação da árvore ARN foi para um tipo de dados genérico, a chave deve ser do tipo void * para que o nó possa receber como chave um ponteiro para qualquer tipo de dado. Como a atividade pediu uma árvore que armazene Artigos, no arquivo main.c foi criada a estrutura Artigo que será armazenada na árvore. Essa estrutura será explicada mais adiante.

Por fim, o último campo é um campo do tipo inteiro que armazena a cor de cada nó sendo 0 para preto e 1 para vermelho.

4.2 MAIN.C

Também foi implementado o arquivo de cliente, também chamado de main. Nele foram definidas as funções de callback, o menu e a função principal do programa. Além disso, também foi implementada uma estrutura “Artigo” para que todas as chaves armazenadas na árvore sejam desse tipo.

4.2.1 Estrutura do artigo

A estrutura criada no arquivo main.c foi denominada artigo e foi previamente definida pela professora. Ela foi implementada da seguinte forma:

IMAGEM 3: Declaração da estrutura do Artigo.

A partir da implementação da estrutura acima os nós inseridos na árvore irão armazenar chaves do tipo “Artigo”, pois no campo chave do nó da árvore será passado um ponteiro para essa estrutura.

4.2.2 Funções de Callback

No arquivo main.c também foram implementadas as funções de callback. Essas funções são passadas por parâmetro nas chamadas de algumas funções do arquivo ARN.c.

4.2.2.1 Imprime Chave

A função “imprimeChave” é a função de callback utilizada para imprimir o id do artigo. Ela recebe um ponteiro do tipo void * e não retorna nada. Nela é feita uma verificação e, se o nó a ser impresso não for nulo, é feito um casting do tipo void * para o tipo Artigo * e imprime o id da chave . Caso o nó seja nulo, a função imprime uma mensagem de erro.

Essa função é importante pois ela é passada como parâmetro para a função do arquivo ARN.c que realiza a impressão da árvore.

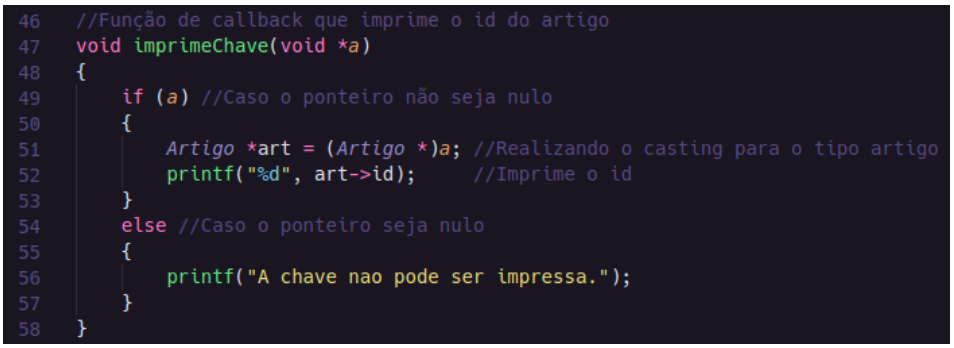


IMAGEM 4: Implementação da função imprimeChave

4.2.2.2 Libera Chave

A função “liberaChave” é a função de callback utilizada para liberar a memória alocada para uma chave do tipo Artigo. Essa função verifica se o nó for

nulo e, se for, imprime uma mensagem de erro. Caso contrário, é necessário realizar um casting de void * para Artigo * e depois executar o comando free( ). Essa função recebe como parâmetro uma chave do tipo void * e não retorna nada. Além disso, ela é passada como parâmetro para as funções de inserção e para a “liberaArvore”. A implementação dessa função está na imagem abaixo:

```
83  //Função de callback que libera a memória alocada para o artigo
84  void liberaChave(void *a)
85  {
86      if (a) //Caso o ponteiro não seja nulo
87      {
88          Artigo *art = (Artigo *)a; //Casting para o tipo artigo
89          free(art);                 //Libera a memória
90      }
91      else //Caso o ponteiro seja nulo
92      {
93          printf("A chave nao pode ser liberada.");
94      }
95  }
```

IMAGEM 5: Implementação da função liberaChave.

4.2.2.3 Compara Chave

A função “comparaChave” é a função de callback usada para comparar dois artigos através do id de cada um deles.

Essa função recebe como parâmetro dois ponteiros para o tipo void para serem comparados. A função verifica se um dos ponteiros é NULL, caso pelo menos um deles seja, não será possível realizar a comparação, então a função imprime uma mensagem de erro e retorna -2 para indicar que a comparação não foi bem sucedida. Caso não ocorram problemas, a função irá realizar o casting de void * para Artigo * e comparar os ids.

Através dessa comparação, a função terá como retorno “1” caso o id do primeiro artigo seja maior que o do segundo, “-1” caso o id do segundo artigo seja maior e “0” caso os ids sejam iguais. A partir desses retornos, a função inserir (do arquivo ARN.c), por exemplo, pode percorrer os nós da árvore e inserir o novo nó no lugar certo, a função de busca também utiliza essa função para realizar as comparações e percorrer a árvore. A implementação dessa função está na imagem abaixo:

```c
//Função de callback utilizada para comparar dois artigos
int comparaChave(void *a1, void *a2)
{
    if (a1 && a2) //Caso os dois artigos não sejam nulos
    {
        //Realizando o casting para o tipo artigo
        Artigo *art1 = (Artigo *)a1;
        Artigo *art2 = (Artigo *)a2;

        if (art1->id > art2->id) //Caso o id do primeiro artigo seja maior do que o do segundo
            return 1;
        else if (art1->id < art2->id) //Caso o id do primeiro artigo seja menor do que o do segundo
            return -1;
        else //Caso os ids sejam iguais
            return 0;
    }
    else //Caso algum dos ponteiros seja nulo
    {
        printf("A chave nao pode ser comparada.");
        return -2;
    }
}
```

IMAGEM 6: Implementação da função comparaChave.

4.2.3 Cria artigo

Também foi implementada a função para alocar memória para uma estrutura do tipo “Artigo”, a função “criaArtigo”. Caso ocorra algum erro de alocação a função irá imprimir uma mensagem de erro e, caso ocorra tudo certo, a função inicializa os campos e retorna o novo nó já com seus atributos. Essa função recebe uma variável do tipo artigo e retorna um ponteiro para a memória alocada para o artigo. A imagem a seguir demonstra como foi implementada essa função.

```c
Artigo *criaArtigo(Artigo a)
{
    //aloca memória
    Artigo *novo = (Artigo *)malloc(sizeof(Artigo));
    if (!novo) //caso ocrra algum erro
    {
        printf("Erro de alocacao de memoria.");
        exit(1);
    }
    //caso não haja erros os campos são inicializados.
    novo->id = a.id;
    novo->ano = a.ano;
    strcpy(novo->autor, a.autor);
    strcpy(novo->titulo, a.titulo);
    strcpy(novo->revista, a.revista);
    strcpy(novo->DOI, a.DOI);
    strcpy(novo->palavraChave, a.palavraChave);
    return novo;
}
```

IMAGEM 7: Implementação da função para alocação de memória de um Artigo.

4.2.4 Menu

Outra função implementada no arquivo “main.c” é a função de menu. Que serve para que o usuário tenha acesso a um menu para escolher a operação que deseja realizar.

Essa função recebe como parâmetro um ponteiro para o nó raiz da árvore e uma variável que indica a opção escolhida pelo usuário. O retorno será um ponteiro que aponta para a raiz da árvore.

As operações a serem realizadas na árvore são escolhidas através desse menu que, no programa feito, está na seguinte ordem:

1 - Inserir Artigo
2 - Remover Artigo
3 - Procurar Artigo
4 - Imprimir Árvore
5 - Sair

O programa é executado e então as opções aparecem na tela para que o usuário escolha a que ele deseja. Essa opção será passada para a função de menu e a partir de um switch case será chamada a função referente a ação desejada.

O usuário não deve escolher uma opção que não existe no menu, mas caso isso ocorra, é exibida uma mensagem de erro e ocorre um looping das opções até que ele escolha uma opção válida. A imagem abaixo mostra o que acontece quando o usuário escolhe uma opção que não estava presente no menu.

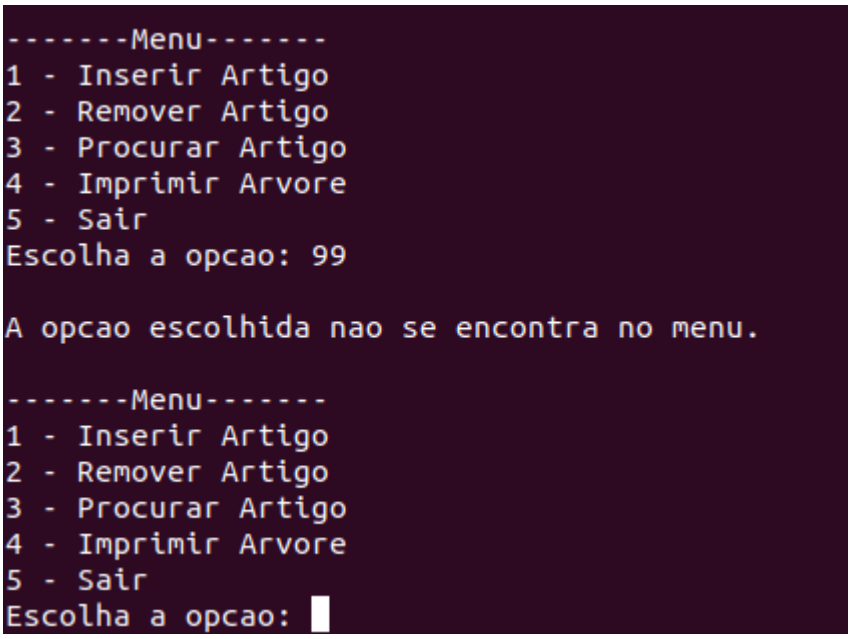


IMAGEM 8: Saída do menu caso a opção escolhida seja inválida.

O caso da opção escolhida não estar disponível no menu foi tratado no caso default do switch case. O código desse caso está na imagem abaixo.

```
default: //Caso a opção escolhida não esteja no menu
    printf("\nA opcao escolhida nao se encontra no menu.\n");
    break;
```

IMAGEM 9: Trecho de código que trata o caso da opção escolhida não estar no menu.

Outra opção é a de sair, e quando isso ocorre, toda a memória alocada durante o programa é liberada por meio da função “liberaArvore” e o programa é encerrado como mostra a figura abaixo.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 5
antonielly@antonielly-pc:~/Documentos/UFES/ED2/T1/T1$
```

IMAGEM 10: Saída do programa caso a opção escolhida seja sair.

4.2.4.1 Inserir

Quando a opção escolhida é “Inserir Artigo” será feita uma leitura dos dados do novo artigo da seguinte forma:

```
-------Menu-------
1 - Inserir Artigo
2 - Remover Artigo
3 - Procurar Artigo
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 1

A opcao selecionada foi: Inserir Artigo.

Digite o id do novo Artigo: 12
Digite o ano do novo Artigo: 2021
Digite o autor do novo Artigo: Willian e Antonielly
Digite o titulo do novo Artigo: Trabalho de EDII
Digite a revista do novo Artigo: Revista
Digite o DOI do novo Artigo: DOI
Digite a palavra chave do novo Artigo: rubro-negra

O seguinte Artigo foi cadastrado:

Id: 12
Ano: 2021
Autor: Willian e Antonielly
Titulo: Trabalho de EDII
Revista: Revista
DOI: DOI
Palavra chave: rubro-negra
```

IMAGEM 11: Saída do programa caso a opção seja inserir.

A partir disso, o programa vai buscar na árvore se a chave a ser inserida já existe, garantindo assim que não se possa inserir chaves repetidas na árvore. Caso seja encontrado um artigo com o mesmo id, uma mensagem de erro é mostrada na tela. Caso contrário, podemos então cadastrar o elemento através da função “insercao”.

A função de inserir recebe como parâmetro um ponteiro para a chave a ser inserida. Nesse caso, a chave é do tipo “Artigo”, então a função “criaArtigo” foi usada para fazer a alocação de memória e inicializar os dados. Depois disso, a memória alocada por essa função foi passada como chave para a função de inserção. Após a inserção, os dados do artigo são impressos na tela. A implementação desse caso está na imagem abaixo:

```
107     case 1: //Inserir elemento
108         //Leitura dos dados
109         printf("\nA opcao selecionada foi: Inserir Artigo.\n\n");
110         printf("Digite o id do novo Artigo: ");
111         scanf("%d", &artigoAuxiliar.id);
112         printf("Digite o ano do novo Artigo: ");
113         scanf("%d", &artigoAuxiliar.ano);
114 
115         //Lê o enter para limpar o buffer do teclado e permitir a leitura da string a seguir
116         scanf("%c", &enter);
117 
118         printf("Digite o autor do novo Artigo: ");
119         //Esse tipo de scanf permite a leitura de uma string com mais de uma palavra
120         scanf("%[^\n]", artigoAuxiliar.autor);
121 
122         //Lendo o buffer do teclado
123         scanf("%c", &enter);
124 
125         printf("Digite o titulo do novo Artigo: ");
126         scanf("%[^\n]", artigoAuxiliar.titulo);
127 
128         //Lendo o buffer do teclado
129         scanf("%c", &enter);
130 
131         printf("Digite a revista do novo Artigo: ");
132         scanf("%[^\n]", artigoAuxiliar.revista);
133 
134         //Lendo o buffer do teclado
135         scanf("%c", &enter);
136 
137         printf("Digite o DOI do novo Artigo: ");
138         scanf("%[^\n]", artigoAuxiliar.DOI);
139 
140         //Lendo o buffer do teclado
141         scanf("%c", &enter);
142 
143         printf("Digite a palavra chave do novo Artigo: ");
144         scanf("%[^\n]", artigoAuxiliar.palavraChave);
145 
146         //Realizando a busca da chave a ser inserida
147         noBuscado = buscaChave(a, &artigoAuxiliar, comparaChave);
```

IMAGEM 12.1: Implementação do código para a chamada da opção inserir.

```
149         if (noBuscado == &externo) //Caso a chave ainda não exista na árvore
150         {
151             //Realiza a inserção do elemento na árvore
152             a = insercao(a, criaArtigo(artigoAuxiliar), comparaChave, liberaChave);
153 
154             //Imprime o elemento inserido
155             printf("\nO seguinte Artigo foi cadastrado:\n");
156             printf("\nId: %d\n", artigoAuxiliar.id);
157             printf("Ano: %d\n", artigoAuxiliar.ano);
158             printf("Autor: %s\n", artigoAuxiliar.autor);
159             printf("Titulo: %s\n", artigoAuxiliar.titulo);
160             printf("Revista: %s\n", artigoAuxiliar.revista);
161             printf("DOI: %s\n", artigoAuxiliar.DOI);
162             printf("Palavra chave: %s\n", artigoAuxiliar.palavraChave);
163         }
164         else //Caso a chave já exista na árvore
165             printf("\nNao foi possivel cadastrar. O Artigo com chave %d ja foi cadastrado.\n", artigoAuxiliar.id);
166         break;
```

IMAGEM 12.2: Implementação do código para a chamada da opção inserir.

4.2.4.2 Remover

Quando a opção escolhida é “Remover Artigo” o usuário precisa digitar o id do artigo que será removido, como mostra a imagem abaixo:

```
-------Menu-------
1 - Inserir Artigo
2 - Remover Artigo
3 - Procurar Artigo
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 2

A opcao selecionada foi: Remover Artigo.

Digite a chave da Artigo a ser removida:
```

IMAGEM 13: Saída do programa caso a opção escolhida seja remover.

O usuário digita a chave e então é feita uma busca na árvore para verificar se a chave está na árvore ou não. Caso a chave não seja encontrada, o programa exibe uma mensagem de erro. Caso contrário, é chamada a função de remover. Vale ressaltar que a função recebe um ponteiro do tipo void * (ponteiro para tipo genérico de dados) que aponta para um artigo que tem o id que o usuário deseja remover então, não é necessário alocar memória para o artigo a ser buscado, basta criar uma variável do tipo Artigo, definir o campo id dela como sendo o id a ser removido e passar o endereço de memória dessa variável para a função. Quando a remoção é realizada, as informações do nó que foi removido são impressas na tela. A implementação desse caso está apresentada na imagem abaixo.

```c
case 2: //Remover elemento
    //Realizando a leitura dos dados
    printf("\nA opcao selecionada foi: Remover Artigo.\n\n");
    printf("Digite a chave do Artigo a ser removido: ");
    scanf("%d", &artigoAuxiliar.id);

    //Buscando pela chave a ser removida
    noBuscado = buscaChave(a, &artigoAuxiliar, comparaChave);

    if (noBuscado != &externo) //Caso a chave exista na árvore
    {
        // Armazena o artigo a ser removido
        artigoBuscado = (Artigo *)retornaChave(noBuscado);
        artigoAuxiliar.id = artigoBuscado->id;
        artigoAuxiliar.ano = artigoBuscado->ano;
        strcpy(artigoAuxiliar.autor, artigoBuscado->autor);
        strcpy(artigoAuxiliar.titulo, artigoBuscado->titulo);
        strcpy(artigoAuxiliar.revista, artigoBuscado->revista);
        strcpy(artigoAuxiliar.DOI, artigoBuscado->DOI);
        strcpy(artigoAuxiliar.palavraChave, artigoBuscado->palavraChave);

        //Realiza a remoção
        a = remocao(a, noBuscado, liberaChave);

        //Imprime a Artigo removido
        printf("\nO seguinte Artigo foi removido:\n");
        printf("\nId: %d\n", artigoAuxiliar.id);
        printf("Ano: %d\n", artigoAuxiliar.ano);
        printf("Autor: %s\n", artigoAuxiliar.autor);
        printf("Titulo: %s\n", artigoAuxiliar.titulo);
        printf("Revista: %s\n", artigoAuxiliar.revista);
        printf("DOI: %s\n", artigoAuxiliar.DOI);
        printf("Palavra chave: %s\n", artigoAuxiliar.palavraChave);
    }
    else //Caso a chave não esteja na árvore
        printf("\nNao foi possivel remover. O Artigo com chave %d nao foi encontrado.\n", artigoAuxiliar.id);
    break;
```

IMAGEM 14: Implementação do código para o caso de o menu chamar a opção de remover

4.2.4.3 Procurar

Quando a opção escolhida é “Procurar artigo”, de maneira semelhante a remoção, também é solicitado que o usuário informe a chave da mesa que está sendo buscada como é mostrado na imagem abaixo:

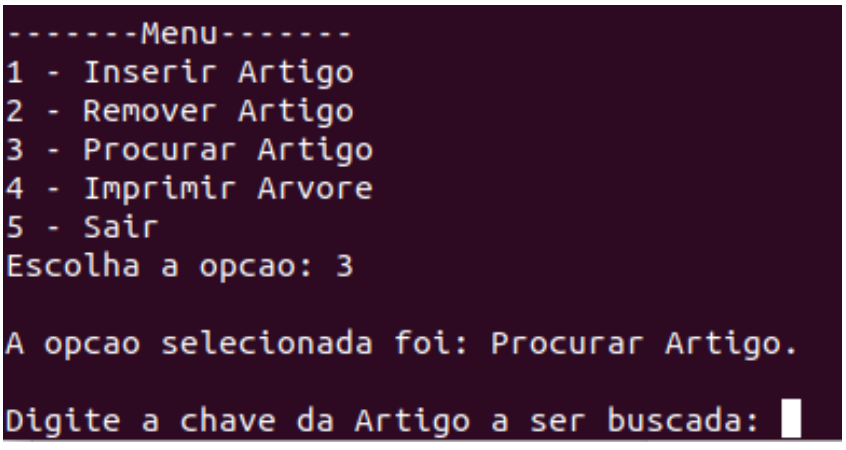


IMAGEM 15: Saída do programa caso a opção escolhida seja procurar.

Após pedir que o usuário digite a chave da mesa a ser buscada, a função de busca será chamada para buscar a chave na árvore. Caso a chave seja encontrada, as informações dela serão impressas. Caso ela não seja encontrada, será impressa

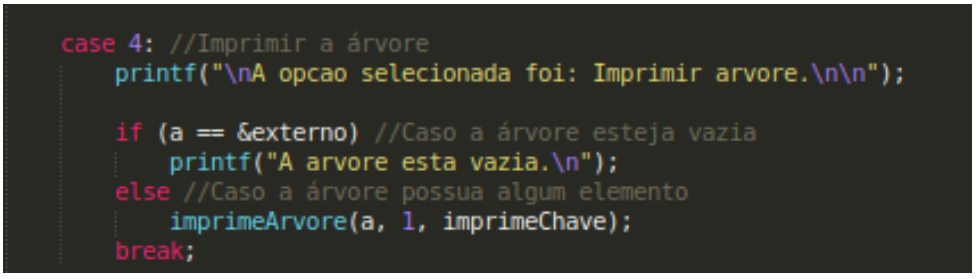
uma mensagem de erro indicando que não foi possível encontrar a chave buscada. A implementação desse caso está na imagem a seguir.

```
206     case 3: //Buscar elemento
207         //Leitura dos dados
208         printf("\nA opcao selecionada foi: Procurar Artigo.\n\n");
209         printf("Digite a chave do Artigo a ser buscado: ");
210         scanf("%d", &artigoAuxiliar.id);
211 
212         //Realizando a busca da chave
213         noBuscado = buscaChave(a, &artigoAuxiliar, comparaChave);
214 
215         if (noBuscado != &externo) //Caso a chave seja encontrada
216         {
217             //Obtém a chave do nó
218             artigoBuscado = (Artigo *)retornaChave(noBuscado);
219             //Imprime as informações do artigo
220             printf("\nId: %d\n", artigoBuscado->id);
221             printf("Ano: %d\n", artigoBuscado->ano);
222             printf("Autor: %s\n", artigoBuscado->autor);
223             printf("Titulo: %s\n", artigoBuscado->titulo);
224             printf("Revista: %s\n", artigoBuscado->revista);
225             printf("DOI: %s\n", artigoBuscado->DOI);
226             printf("Palavra chave: %s\n", artigoBuscado->palavraChave);
227         }
228         else //Caso a chave não esteja na árvore
229             printf("\nO Artigo buscado nao foi encontrado.\n");
230         break;
```

IMAGEM 16: Implementação do código para o caso de a opção escolhida no menu for procurar.

4.2.4.4 Imprimir

Quando a opção escolhida é “Imprimir árvore”, é verificado se a árvore está vazia e, caso esteja, uma mensagem é impressa dizendo que ela está vazia. Caso contrário, a função de impressão da árvore é chamada. A implementação desse caso está na imagem a seguir.



```
case 4: //Imprimir a árvore
    printf("\nA opcao selecionada foi: Imprimir arvore.\n\n");

    if (a == &externo) //Caso a árvore esteja vazia
        printf("A arvore esta vazia.\n");
    else //Caso a árvore possua algum elemento
        imprimeArvore(a, 1, imprimeChave);
    break;
```

IMAGEM 17: Implementação do código para o caso de a opção escolhida no menu for imprimir.

4.2.5 Main

Além das funções citadas existe, no arquivo main.c, a função principal que é justamente a “main”. Essa função inicializa uma árvore vazia para realização das operações disponíveis no menu. A função “main” também é utilizada para a realização do laço de repetição (do-while) das opções do menu, funcionando da seguinte forma: Assim que o usuário escolher a opção desejada, a função “menu” será chamada recebendo como parâmetros a árvore que foi inicializada e o número referente a opção escolhida para assim realizar a operação. Além disso, a função de menu só é chamada caso a opção escolhida não seja a opção de sair. No fim, ao escolher a opção de sair, o programa irá sair do loop e a função “liberaArvore” irá liberar a memória que foi alocada para cada nó da árvore e suas respectivas chaves. A implementação da função main está na imagem abaixo.

```c
int main()
{
    ARN *a = criaArvore();
    Artigo art;

    int op = 0;
    do
    {
        //Opções do menu
        printf("\n-------Menu-------\n");
        printf("1 - Inserir Artigo\n");
        printf("2 - Remover Artigo\n");
        printf("3 - Procurar Artigo\n");
        printf("4 - Imprimir Arvore\n");
        printf("5 - Sair\n");

        //Leitura da opção
        printf("Escolha a opcao: ");
        scanf("%d", &op);

        if (op != 5) //Se a opção selecionada não for a de sair
            a = menu(a, op);

    } while (op != 5); //Permanece no menu enquanto não for selecionada a opção de sair

    //Libera a memória alocada para a árvore
    liberaArvore(a, liberaChave);

    return 0;
}
```

IMAGEM 18: Implementação da função main.

4.3 MAKEFILE

Também foi implementado o arquivo “makefile” contendo um conjunto de diretivas usadas para a compilação dos arquivos. Esse arquivo permite que todo o código seja compilado apenas com o comando make, e permite que todos os arquivos gerados pela compilação sejam removidos com o comando make clean. Esse arquivo torna a compilação do programa bem mais simples e prática. A implementação do arquivo makefile se encontra na imagem abaixo.

```
all: ARN

ARN: main.o ARN.o
    gcc -o ARN main.o ARN.o

main.o: main.c ARN.h
    gcc -c main.c -Wall

ARN.o: ARN.c ARN.h
    gcc -c ARN.c -Wall

clean:
    rm ARN main.o ARN.o
```

IMAGEM 19: Arquivo makefile

4.4 ARN.C

Também foi implementado um arquivo contendo todas as implementações das funções relativas à árvore rubro-negra. Esse arquivo é o “ARN.c” e as funções presentes nesse arquivo serão explicadas a seguir:

4.4.1 Cria nó

A função “criaNo”, é a função usada para alocar memória para um novo nó na árvore. Ela imprime uma mensagem de erro caso ocorra algum erro durante a alocação e, caso não haja nenhum erro, os campos desse nó são inicializados. Por fim a função retorna o novo nó alocado.

```
// Função que aloca a memória para um novo nó da árvore
ARN *criaNo(void *chave)
{
    //Alocação de memória
    ARN *novo = (ARN *)malloc(sizeof(ARN));
    if (!novo) //Caso ocorra algum erro de alocação
    {
        printf("Erro de alocacao de memoria!");
        exit(1);
    }
    //inicialização dos campos do nó
    novo->chave = chave;
    novo->cor = 1;
    novo->dir = &externo;
    novo->esq = &externo;
    novo->pai = &externo;

    return novo; //retorna o nó inicializado.
}
```

IMAGEM 20: Implementação da função criaNo.

4.4.2 Cria Arvore

Na implementação feita, optou-se por utilizar a abordagem com o nó sentinela. Esse nó foi chamado de nó externo e foi criado como uma variável global.

Desse modo, em vez de um ponteiro apontar para NULL quando ele não possui subárvores ou pai, basta fazer ele apontar para o nó sentinela para que o código seja mais simples e que seja possível evitar algumas falhas de segmentação. Dessa forma, a árvore ficaria da forma ilustrada abaixo, sendo “E” o nó externo:

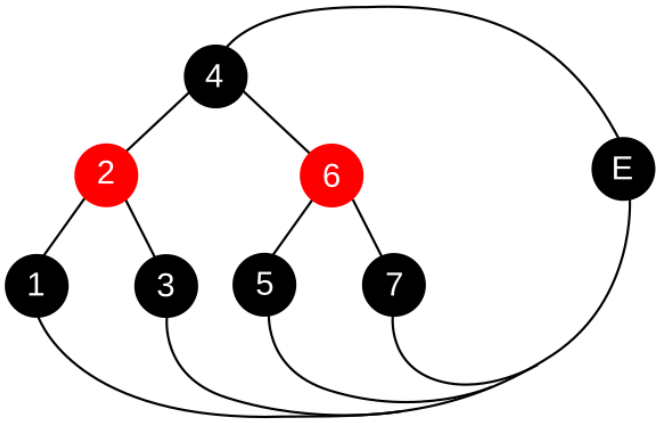


IMAGEM 21: Implementação com nó sentinela.

Dessa forma, a função “criaArvore” é a função usada para inicializar a variável “externo” e retornar o endereço dessa variável. A implementação dessa função se encontra na imagem abaixo:

```
//Função que incializa a variavel externo.
ARN *criaArvore()
{
    externo.chave = 0;
    externo.cor = 0;
    externo.pai = NULL;
    externo.esq = NULL;
    externo.dir = NULL;
    return &externo; //retorna o campo inicializado.
}
```

IMAGEM 22: Implementação da função criaArvore.

Vale ressaltar que, como em uma árvore rubro-negra as folhas são sempre negras, a cor do nó externo também é preta.

4.4.3 Retorna Chave

A função "retornaChave" é uma função auxiliar utilizada somente para retornar o campo chave de um nó para que o programa cliente não precise ter acesso aos campos da árvore rubro negra. Dessa forma, ele apenas usa essa função para obter a chave de um nó que foi buscado. A implementação dessa função se encontra abaixo:

```c
//função auxiliar que retorna o campo chave de um nó da árvore.
void *retornaChave(ARN *a)
{
    return a->chave;
}
```

IMAGEM 23: Implementação da função retornaChave.

4.4.4 Busca chave

A função “buscaChave” é utilizada para buscar uma chave na árvore e funciona da seguinte forma: primeiro é verificado se a árvore está vazia ou não. Caso esteja ou a chave não seja encontrada, a função irá retornar o endereço de “externo”. Caso não esteja vazia, a função começa a realizar as comparações que ocorrem através da recursão e do auxílio da função de callback “compara”. O primeiro condicional é usado para comparar se o id da chave a ser buscada é menor do que o da chave atual. Se for, chama novamente a função “buscaChave” mas agora percorrendo a subárvore da esquerda. O mesmo acontece com o condicional para percorrer a subárvore da direita, porém agora, na função “compara”, a chave deverá ser maior do que a chave atual. Com isso, a função irá percorrer a árvore até encontrar a chave, ou chegar ao nó externo. Quando essa chave for encontrada, o retorno será um ponteiro para o nó da árvore que contém a chave buscada. A imagem abaixo demonstra como foi implementada a função.

```c
//Função recursiva que busca uma chave na árvore
ARN *buscaChave(ARN *a, void *chave, int (*comparaChave)(void *, void *))
{
    if (a != &externo) //Caso a árvore não esteja vazia
    {
        if (comparaChave(chave, a->chave) == -1) //Se a chave buscada for menor do que a chave atual
            //Realiza a busca na subárvore da esquerda
            return buscaChave(a->esq, chave, comparaChave);
        else if (comparaChave(chave, a->chave) == 1) //Se a chave buscada for maior do que a chave atual
            //Realiza a busca na subárvore da direita
            return buscaChave(a->dir, chave, comparaChave);
        else //Caso a chave for encontrada
            //Retorna o nó que contém a chave buscada
            return a;
    }
    else //Se a árvore está vazia ou o elemento não está na árvore
        return &externo;
}
```

IMAGEM 24: Implementação da função buscaChave.

Essa função de busca foi utilizada no menu para procurar um artigo por meio da chave que ele possui. Quando se realiza a busca de uma chave, ela pode ser encontrada na árvore ou não. Os resultados para cada caso estão apresentados nas imagens a seguir:

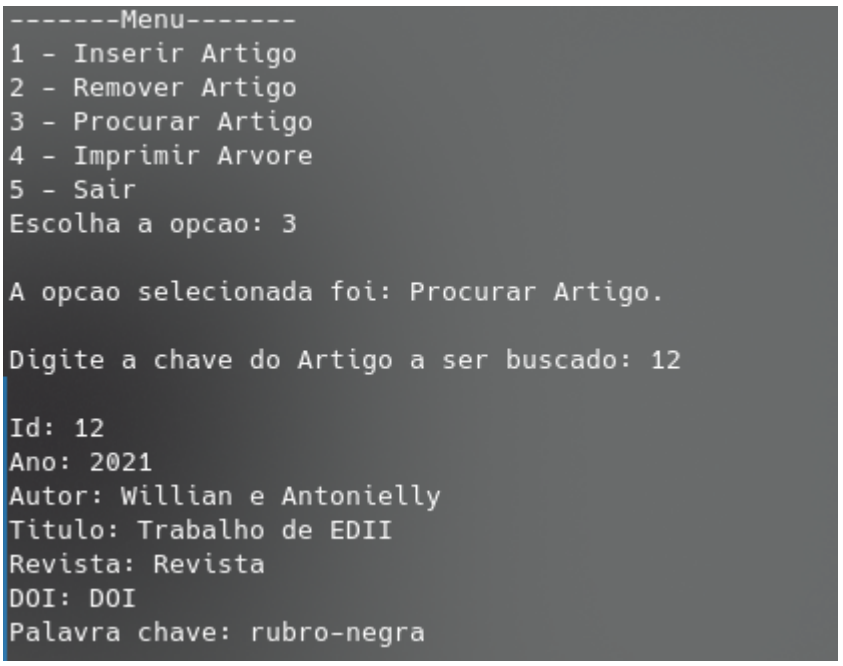


IMAGEM 25: Saída do programa caso a chave seja encontrada.

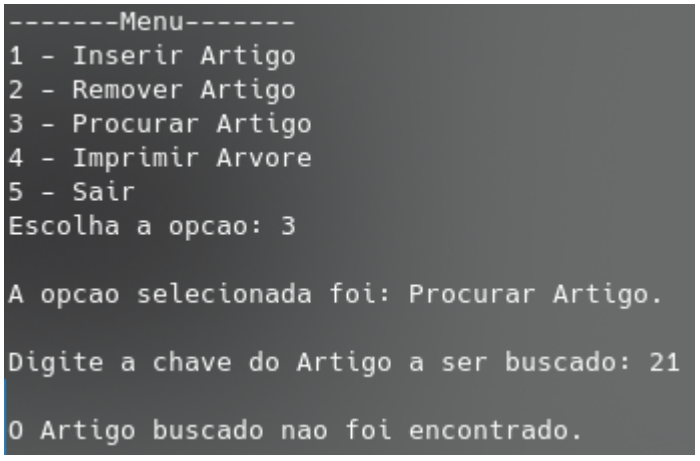


IMAGEM 26: Saída do programa caso a chave não seja encontrada.

4.4.5 Libera Árvore

Essa função é utilizada para liberar a memória alocada para cada nó da árvore e de suas respectivas chaves. Ela funciona assim: a primeira coisa a ser feita é verificar se a árvore não está vazia, se isso for verdade, a função vai funcionar através de recursão para percorrer toda árvore liberando nó por nó. Por fim, é utilizada a função de callback “liberaChave” para liberar a memória alocada para cada chave de cada nó da árvore. O comando utilizado para liberar a memória alocada para o nó é o comando free( ). A implementação dessa função se encontra na imagem abaixo.

```c
//Função que libera a memória alocada para o nó da árvore e suas respectivas chaves
void liberaArvore(ARN *a, void (*liberaChave)(void *))
{
    if (a != &externo) //Caso a árvore não esteja vazia
    {
        //Chama a função recursivamente para liberar a memória na subárvore esquerda
        liberaArvore(a->esq, liberaChave);
        //Chama a função recursivamente para liberar a memória na subárvore direita
        liberaArvore(a->dir, liberaChave);
        //Utiliza a função de callback para liberar a memória alocada para a chave
        liberaChave(a->chave);
        //Libera a memória alocada para o nó da árvore.
        free(a);
    }
}
```

IMAGEM 27: Implementação da função liberaArvore.

4.4.6 Imprime Árvore

A função “imprimeArvore” é usada para imprimir cada nó de uma árvore e a sua respectiva cor. A função funciona da seguinte forma: Caso a árvore não esteja vazia, a função de impressão é chamada recursivamente para a subárvore da direita, ocorre a tabulação do nível da árvore através de um for e um print com “\t” e depois a recursão para a subárvore da esquerda. Além disso, a impressão da chave é feita através da função de callback “imprimeChave”. Como também é necessário realizar a impressão da cor, existe um if que define uma variável cor com “V” ou “P” dependendo do conteúdo do nó: “0” ou “1”. A impressão da árvore ocorre de forma simétrica. A implementação dessa função está na imagem abaixo.

```c
/*Função utilizada para imprimir cada nó da árvore e sua cor  aplicando a tabulação
de acordo com o nível de cada nó */
void imprimeArvore(ARN *a, int cont, void (*imprimeChave)(void *))
{
    if (a != &externo) //Caso a árvore não esteja vazia
    {
        //Chama a função recursivamente para imprimir a subarvore da direita
        imprimeArvore(a->dir, cont + 1, imprimeChave);

        //Realiza a tabulação por nível
        for (int i = 0; i < cont; i++)
        {
            printf("\t");
        }

        //define uma variavel cor do tipo char que será utilizada para imprimir a cor do nó
        char cor;
        if (a->cor == 0) //se o campo cor for 0
            cor = 'P'; // cor recebe P de preto
        else // caso contrário
            cor = 'V'; //cor recebe V de vermelho

        printf("[");
        //Utiliza a função de callback para imprimir a chave
        imprimeChave(a->chave);
        printf(": %c]\n", cor);

        //Chama a função recursivamente para imprimir a subarvore da esquerda
        imprimeArvore(a->esq, cont + 1, imprimeChave);
    }
}
```

IMAGEM 28: Implementação da função imprimeArvore.

A função imprimeArvore é utilizada pelo menu para realizar a impressão da árvore. As imagens a seguir mostram como foi feito o tratamento dos casos da árvore estar vazia ou não.

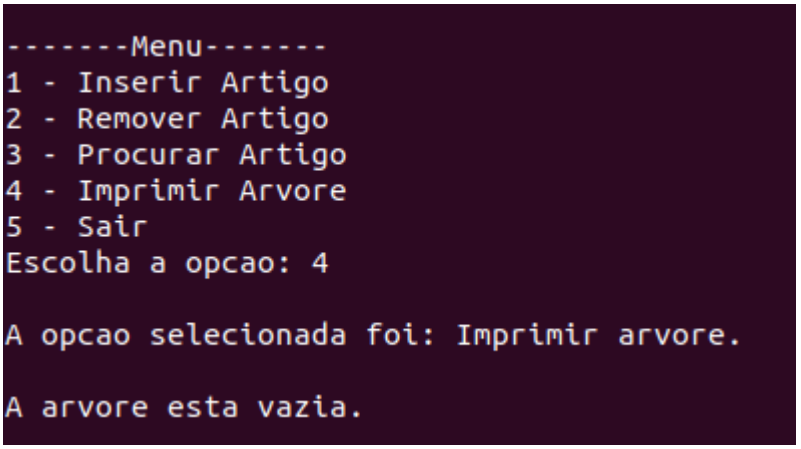


IMAGEM 29: Saída do programa para a opção imprimir caso a árvore esteja vazia.

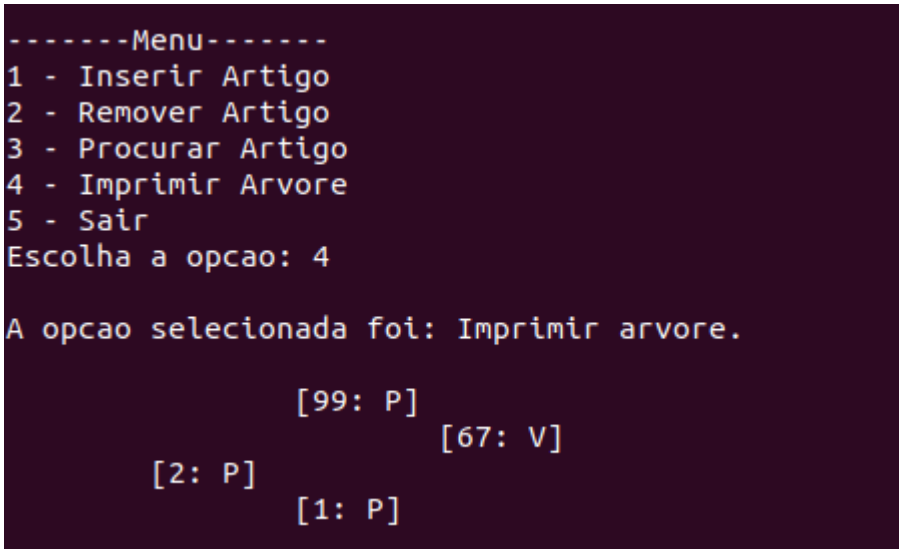


IMAGEM 30: Saída do programa para a opção imprimir para a árvore não vazia

4.4.7 Rotação direita

Após realizar os processos de inserção e remoção na árvore, pode ser que ocorra um desbalanceamento ou violação de alguma propriedade da árvore. Para resolver esse problema, são utilizados os processos de rotação. Esses processos consistem basicamente na reorganização dos ponteiros da árvore com o intuito de restabelecer as propriedades da árvore.

O processo de rotação à direita busca realizar uma rotação para a direita, mantendo as propriedades de árvore de busca binária. Ele geralmente é utilizado quando ocorre algum desbalanceamento da subárvore esquerda do nó.

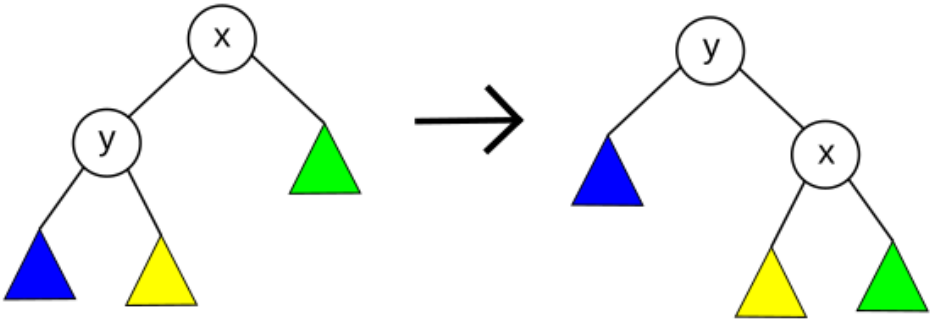


IMAGEM 31: Rotação à direita

A imagem acima ilustra como ocorre o processo de rotação à direita. Vale ressaltar que na imagem os nós estão pintados de branco apenas para representar um caso genérico, mas na árvore rubro-negra esses devem ser ou vermelhos ou pretos para que não violem a propriedade 1 da árvore. Inicialmente a árvore se encontra à disposição à esquerda, em que cada nó possui subárvores que foram representadas pelos triângulos. Após a rotação, a árvore passa a ter a disposição representada à direita. O código dessa rotação se encontra na imagem abaixo:

```
77  //Função que realiza a rotação simples à direita
78  ARN *rotacaoDireita(ARN *raiz, ARN *x)
79  {
80      //Declaração da variavel
81      ARN *filhoEsquerda;
82
83      //filhoEsquerda recebe o filho a esquerda do nó
84      filhoEsquerda = x->esq;
85
86      //O filho a esquerda de x recebe o filho a direita do filho à esquerda
87      x->esq = filhoEsquerda->dir;
88
89      if (filhoEsquerda->dir != &externo) //Se o filho a esquerda tiver filho a direita
90          filhoEsquerda->dir->pai = x;    //O pai do filho a direita do filho a esquerda recebe x
91
92      filhoEsquerda->pai = x->pai; //O pai do filho à esquerda recebe o pai de x
93
94      if (x->pai == &externo)   //Caso x seja a raiz
95          raiz = filhoEsquerda; //A nova raiz se torna o filho à esquerda
96      else                      //Caso x não seja a raiz
97      {
98          if (x == x->pai->dir)           //Se x for o filho à direita do pai dele
99              x->pai->dir = filhoEsquerda; //O filho a direita do pai de x recebe o filho à esquerda
100         else                            //Se x for o filho à esquerda do pai dele
101             x->pai->esq = filhoEsquerda; //O filho a esquerda do pai de x recebe o filho à esquerda
102     }
103     filhoEsquerda->dir = x; //O filho à direita de filhoEsquerda passa a ser x
104     x->pai = filhoEsquerda; //O pai de x passa a ser filhoEsquerda
105
106     return raiz; //Retorna o novo nó raiz
107 }
```

IMAGEM 32: Implementação da Rotação à direita

Considerando que a rotação aconteceu sobre o nó “x”, e que “y” é o filho à esquerda de “x”, para realizar a rotação à direita é necessário realizar o seguinte procedimento:

1. O filho à esquerda de “x“ passa a ser o filho à direita de “y” (linha 87)
2. Se a subárvore amarela não for nula, o pai dela passa a ser “x” (linha 89)
3. O pai de “y” passa a ser o pai de “x” (mesmo que seja o nó externo) (linha 92)
4. Se “x” for a raiz da árvore, a nova raiz passa a ser y (linha 94)
5. Se “x” não for a raiz da árvore, o pai de “x” recebe “y” como filho (linha 98)
6. O filho à direita de “y” passa a ser “x” (linha 103)
7. O pai de “y” passa a ser x (linha 104)

O procedimento descrito acima deve ser realizado nessa ordem para que não se perca a referência para nenhum nó da árvore durante o processo. Além disso, no passo 5, como o pai de “x” pode ter filho à esquerda ou à direita, é feita uma verificação nas linhas 98 e 100 para saber se “x” é o filho à direita ou à esquerda do pai dele e depois o pai recebe “y” como o filho que “x” era antes da rotação.

4.4.8 Rotação esquerda

Outro caso analisado será o caso de rotação à esquerda. Esse processo busca realizar a rotação, mantendo as propriedades de árvore de busca binária. Ele geralmente é utilizado quando ocorre algum desbalanceamento da subárvore direita do nó.

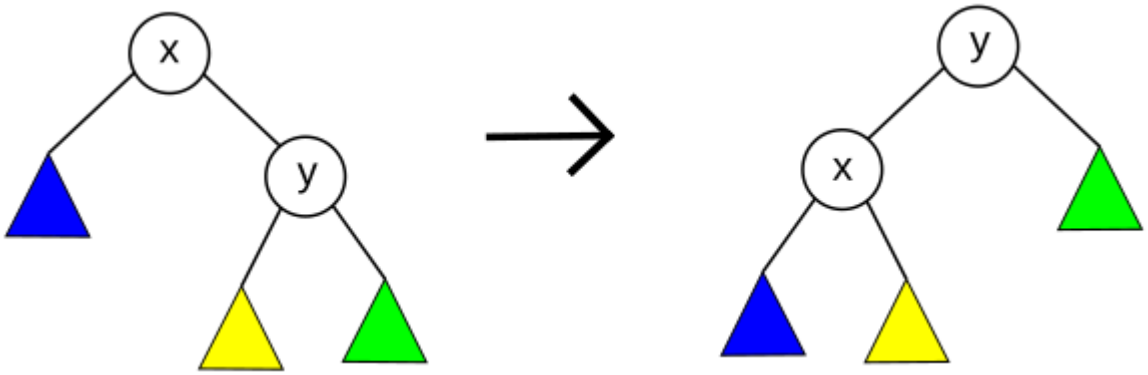
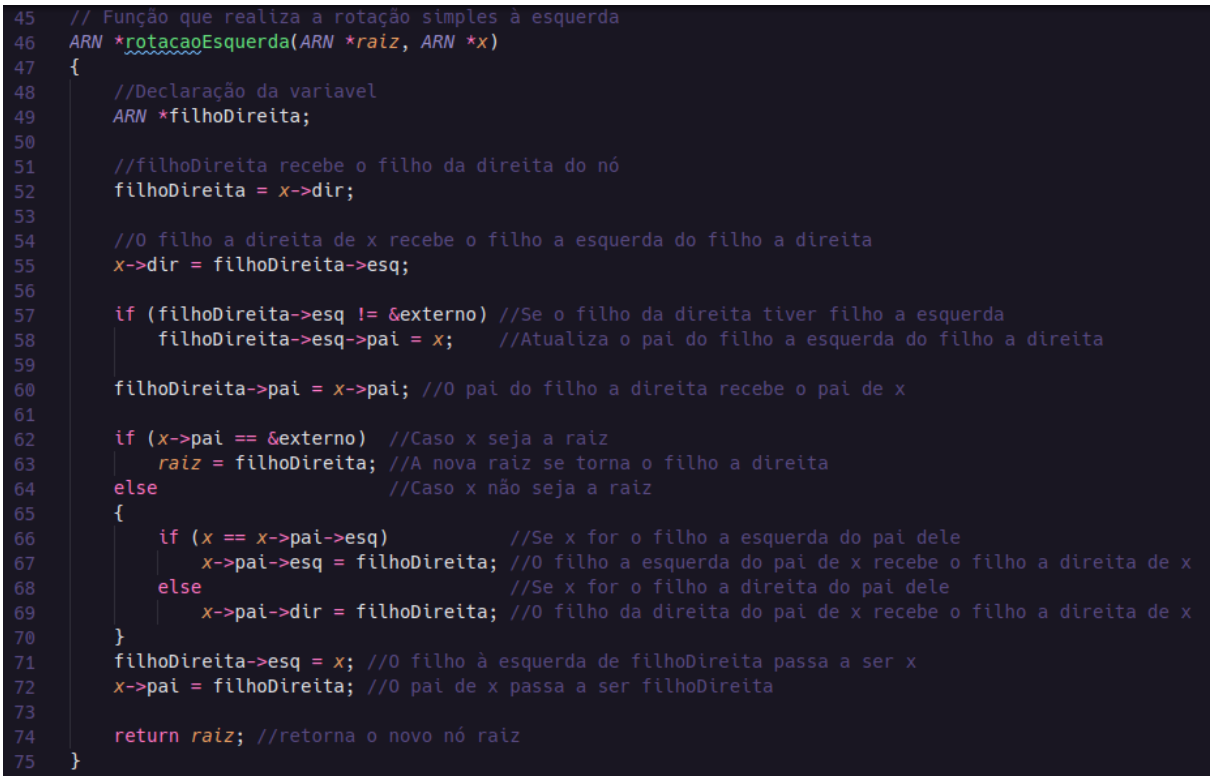


IMAGEM 33: Rotação à esquerda

A imagem acima ilustra como ocorre o processo de rotação à esquerda. Vale ressaltar que na imagem os nós estão pintados de branco apenas para representar um caso genérico, mas na árvore rubro-negra eles devem ser ou vermelhos ou pretos para que não violem a propriedade 1 da árvore. Inicialmente a árvore se encontra à disposição à esquerda, em que cada nó possui subárvores que foram

representadas pelos triângulos. Após a rotação, a árvore passa a ter a disposição à direita. O código dessa rotação se encontra na imagem abaixo:



```
45  // Função que realiza a rotação simples à esquerda
46  ARN *rotacaoEsquerda(ARN *raiz, ARN *x)
47  {
48      //Declaração da variavel
49      ARN *filhoDireita;
50
51      //filhoDireita recebe o filho da direita do nó
52      filhoDireita = x->dir;
53
54      //O filho a direita de x recebe o filho a esquerda do filho a direita
55      x->dir = filhoDireita->esq;
56
57      if (filhoDireita->esq != &externo) //Se o filho da direita tiver filho a esquerda
58          filhoDireita->esq->pai = x;    //Atualiza o pai do filho a esquerda do filho a direita
59
60      filhoDireita->pai = x->pai; //O pai do filho a direita recebe o pai de x
61
62      if (x->pai == &externo)  //Caso x seja a raiz
63          raiz = filhoDireita; //A nova raiz se torna o filho a direita
64      else                     //Caso x não seja a raiz
65      {
66          if (x == x->pai->esq)          //Se x for o filho a esquerda do pai dele
67              x->pai->esq = filhoDireita; //O filho a esquerda do pai de x recebe o filho a direita de x
68          else                           //Se x for o filho a direita do pai dele
69              x->pai->dir = filhoDireita; //O filho da direita do pai de x recebe o filho a direita de x
70      }
71      filhoDireita->esq = x; //O filho à esquerda de filhoDireita passa a ser x
72      x->pai = filhoDireita; //O pai de x passa a ser filhoDireita
73
74      return raiz; //retorna o novo nó raiz
75  }
```

IMAGEM 34: Implementação da rotação à esquerda

Considerando que a rotação aconteceu sobre o nó “x”, e que “y” é o filho à direita de “x”, para realizar a rotação à esquerda é necessário realizar o seguinte procedimento:

1. O filho à direita de “x“ passa a ser o filho à esquerda de “y” (linha 55)
2. Se a subárvore amarela não for nula, o pai dela passa a ser “x” (linha 57)
3. O pai de “y” passa a ser o pai de “x” (mesmo que seja o nó externo) (linha 60)
4. Se “x” for a raiz da árvore, a nova raiz passa a ser y (linha 63)
5. Se “x” não for a raiz da árvore, o pai de “x” recebe “y” como filho (linha 66)
6. O filho à esquerda de “y” passa a ser “x” (linha 71)
7. O pai de “y” passa a ser x (linha 72)

O procedimento descrito acima deve ser realizado nessa ordem para que não se perca a referência para nenhum nó da árvore durante o processo. Além disso, no passo 5, como o pai de “x” pode ter filho à esquerda ou à direita, é feita uma verificação nas linhas 66 e 68 para saber se “x” é o filho à direita ou à esquerda do pai dele e depois o pai recebe “y” como o filho que “x” era antes da rotação.

4.4.9 Inserção

A função “insercao”, como o nome sugere, é uma função utilizada para inserir um novo nó na árvore. A implementação dessa função ficou da seguinte forma:

```
171  //Função de inserção na árvore rubro negra
172  ARN *insercao(ARN *raiz, void *chave, int (*comparaChave)(void *, void *), void (*liberaChave)(void *))
173  {
174      //Criação do novo nó
175      ARN *novoNo = criaNo(chave);
176
177      //Declaração de veriaveis
178      ARN *x, *pai;
179
180      pai = &externo;
181      x = raiz;
182
183      while (x != &externo)
184      {
185          if (comparaChave(x->chave, novoNo->chave) == 0) //Caso a chave já exista na árvore
186          {
187              //Libera a memória alocada para a chave e para o nó
188              liberaChave(novoNo->chave);
189              free(novoNo);
190              return raiz; //Retorna o nó raiz
191          }
192
193          pai = x;
194
195          if (comparaChave(novoNo->chave, x->chave) == -1) //Caso a chave a ser inserida seja menor do que o elemento atual
196              x = x->esq;
197          else //Caso a chave a ser inserida seja maior do que o elemento atual
198              x = x->dir;
199      }
200
201      novoNo->pai = pai;
202
203      if (pai == &externo) //Se a árvore estava vazia
204          raiz = novoNo;
205      else
206      {
207          if (comparaChave(novoNo->chave, pai->chave) == -1) //Se o novo nó for o filho a esquerda do pai
208              pai->esq = novoNo;
209          else //Se o novo nó for o filho a direita do pai
210              pai->dir = novoNo;
211      }
212
213      //O novo nó não tem filhos e tem cor vermelha
214      novoNo->esq = &externo;
215      novoNo->dir = &externo;
216      novoNo->cor = 1;
```

IMAGEM 35.1: Implementação da inserção

```
217
218      raiz = balanceamentoInsercao(raiz, novoNo); //Chama a função para consertar possíveis desbalanceamentos
219
220      return raiz; //Retorna o nó raiz
221  }
```

IMAGEM 35.2: Implementação da inserção

A função recebe a chave a ser inserida e, a partir dela, cria um novo nó que será inserido na árvore (linha 175). Além disso, a função também recebe o ponteiro para a raiz da árvore e as funções de callback “comparaChave” e “liberaChave”. Depois de criar o nó com a chave, é criado então um laço que irá se repetir enquanto x, que inicialmente é o nó raiz da árvore, ainda não tiver chegado a uma folha (linha 183). Com isso, iniciam-se as verificações necessárias através da função de callback “comparaChave” para que ocorra a inserção no devido lugar.

Inicialmente, é feita uma busca na árvore para encontrar a posição em que o novo nó deve ser inserido (linha 183 a 199). Nesse processo, é verificado se a chave do nó atual e a chave a ser inserida são iguais (linha 185). Se forem iguais, a

função “liberaChave” é chamada para liberar a memória alocada para aquela chave, em seguida é utilizado o comando free para liberar também a memória alocada para o nó, pois a árvore não aceita chaves repetidas.

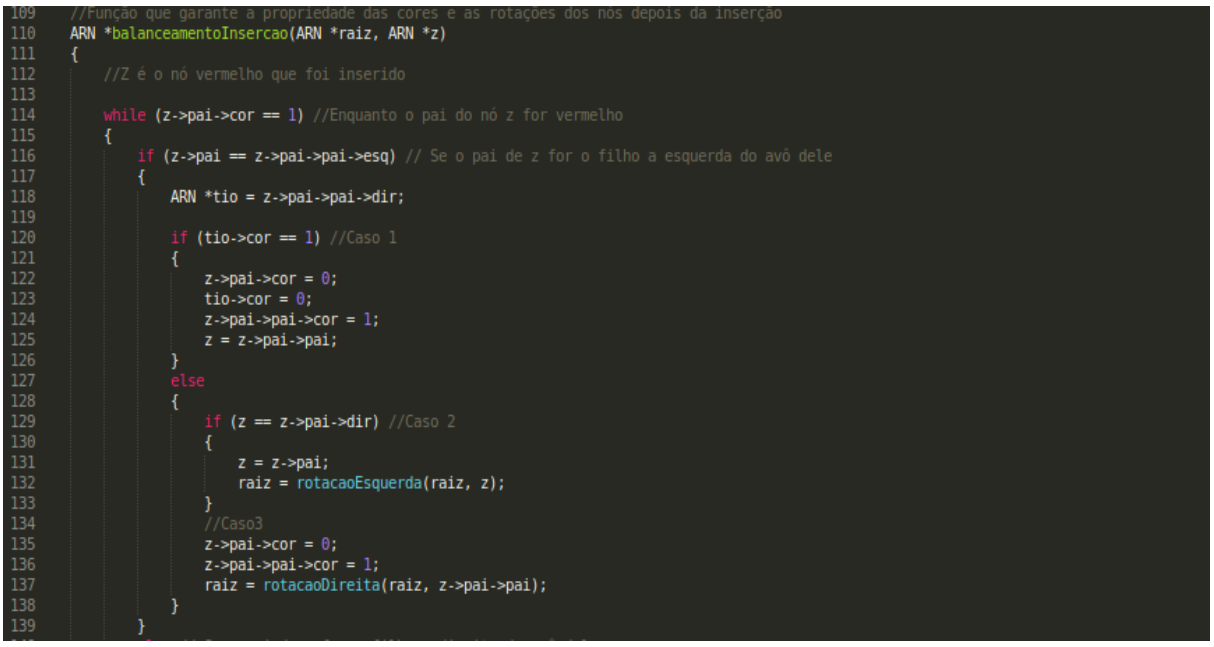
Em seguida, a verificação é feita para saber se a chave a ser inserida é menor do que a chave do nó atual (linha 195), se for, x passará a ser x->esq e se não for, x passa a ser x->dir.

Essas verificações terminam quando “x” chegar ao nó externo. Durante todo esse processo, a variável “pai” armazena o pai do nó x. Ao fim do laço, é feita uma verificação para o caso da árvore estar vazia (linha 203), caso esteja, a raiz simplesmente irá receber o novo nó que agora passa a ser o nó raiz (linha 204). Caso a árvore não esteja vazia, novamente será utilizada a função de callback para verificar se o novo nó é filho à direita ou à esquerda do nó pai. Assim, o nó pai irá receber o novo nó à sua esquerda ou direita dependendo do retorno dessa comparação.

Por fim, os atributos do novo nó são atualizados para garantir que os filhos dele sejam o nó sentinela e que a cor dele seja vermelha (linhas 214 a 216), já que todo novo nó é vermelho para não correr o risco de alterar a altura preta da árvore. Depois disso, é chamada a função de balanceamento da inserção com o nó raiz e o novo nó (linha 218) pois, ao realizar a inserção de um novo nó, optou-se por sempre inserir o nó com cor vermelha para não correr o risco de violar a altura de nós negros na árvore. Entretanto, ao inserir um nó vermelho na árvore, pode ser que ele seja filho de um nó vermelho e isso viola a propriedade 4 da árvore rubro-negra.

4.4.10 Balanceamento inserção

A função “balanceamentoInsercao” serve para que as propriedades da árvore rubro-negra sejam restabelecidas após a inserção. A implementação dessa função se encontra nas imagens abaixo:



```
109   //Função que garante a propriedade das cores e as rotações dos nós depois da inserção
110   ARN *balanceamentoInsercao(ARN *raiz, ARN *z)
111   {
112       //Z é o nó vermelho que foi inserido
113
114       while (z->pai->cor == 1) //Enquanto o pai do nó z for vermelho
115       {
116           if (z->pai == z->pai->pai->esq) // Se o pai de z for o filho a esquerda do avô dele
117           {
118               ARN *tio = z->pai->pai->dir;
119
120               if (tio->cor == 1) //Caso 1
121               {
122                   z->pai->cor = 0;
123                   tio->cor = 0;
124                   z->pai->pai->cor = 1;
125                   z = z->pai->pai;
126               }
127               else
128               {
129                   if (z == z->pai->dir) //Caso 2
130                   {
131                       z = z->pai;
132                       raiz = rotacaoEsquerda(raiz, z);
133                   }
134                   //Caso3
135                   z->pai->cor = 0;
136                   z->pai->pai->cor = 1;
137                   raiz = rotacaoDireita(raiz, z->pai->pai);
138               }
139           }
```

IMAGEM 36.1: Implementação da função balanceamentoInsercao

```
140         else // Se o pai de z for o filho a direita do avô dele
141         {
142             ARN *tio = z->pai->pai->esq;
143 
144             if (tio->cor == 1) //Caso 1
145             {
146                 z->pai->cor = 0;
147                 tio->cor = 0;
148                 z->pai->pai->cor = 1;
149                 z = z->pai->pai;
150             }
151             else
152             {
153                 if (z == z->pai->esq) //Caso 2
154                 {
155                     z = z->pai;
156                     raiz = rotacaoDireita(raiz, z);
157                 }
158 
159                 //Caso 3
160                 z->pai->cor = 0;
161                 z->pai->pai->cor = 1;
162                 raiz = rotacaoEsquerda(raiz, z->pai->pai);
163             }
164         }
165     }
166 
167     raiz->cor = 0; // Define a raiz como preta
168     return raiz;   //Retorna o nó raiz
169 }
170 
```

IMAGEM 36.2: Implementação da função balanceamentoInsercao

No código, a variável “z” representa o novo nó que foi inserido na árvore. Um novo nó sempre tem por padrão a cor vermelha. Com isso, é definido um laço que percorrerá os nós da árvore enquanto a cor do pai de z for vermelha (linha 114). A partir desse while foram tratados os casos de violação das propriedades. Existem 3 casos para o caso de pai de “z” ser o filho à direita do avô dele e 3 casos para caso o pai de Z ser o filho à esquerda do avô dele. Exemplificamos os casos em que o pai de “z” é o filho à esquerda do avô dele e considera-se que os casos em que o pai de “z” é o filho da direita do avô dele são simétricos aos casos apresentados.

Caso 1- Pai e tio do novo nó são vermelhos (linhas 120-126):

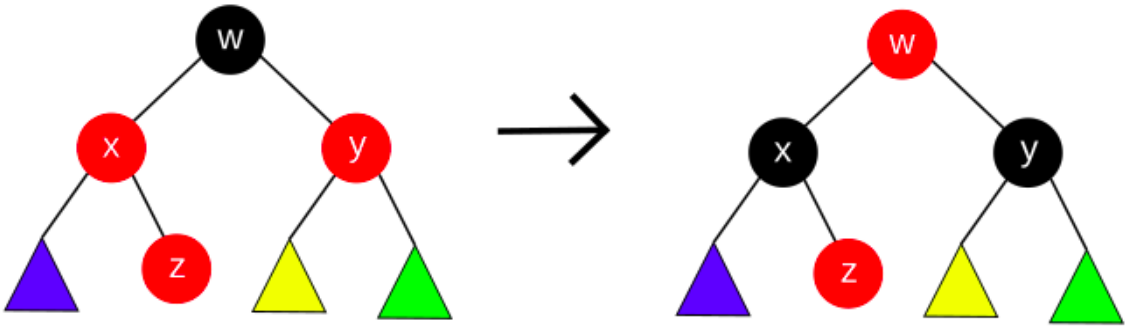


IMAGEM 37: Caso 1a - balanceamentoInsercao

Nesse caso, Z representa o nó que foi inserido, X é o pai de Z e Y é o tio de Z. Ao inserir esse nó à direita de x (que obrigatoriamente será vermelho), o pai e o

tio de Z também estão vermelhos e isso viola a propriedade que diz que o filho de um nó vermelho deve ser preto. Assim sendo, é necessário tratar esse caso.

No caso 1, o tratamento é bem simples pois não é necessário realizar rotações, apenas mudanças de cores, então o que irá acontecer é que o pai e o tio do novo nó ficarão pretos (linha 122 e 123) e o avô desse nó ficará vermelho (linha 124). Entretanto, se o avô for a raiz da árvore, ele não pode ser vermelho pois desobedece a propriedade de que a raiz de uma árvore rubro-negra deve ser negra, mas não podemos simplesmente defini-lo como preto pois ele pode não ser raiz, então faremos o novo nó apontar para o seu avô (linha 149) e o laço de repetição irá realizar as devidas verificações no novo nó na próxima iteração.

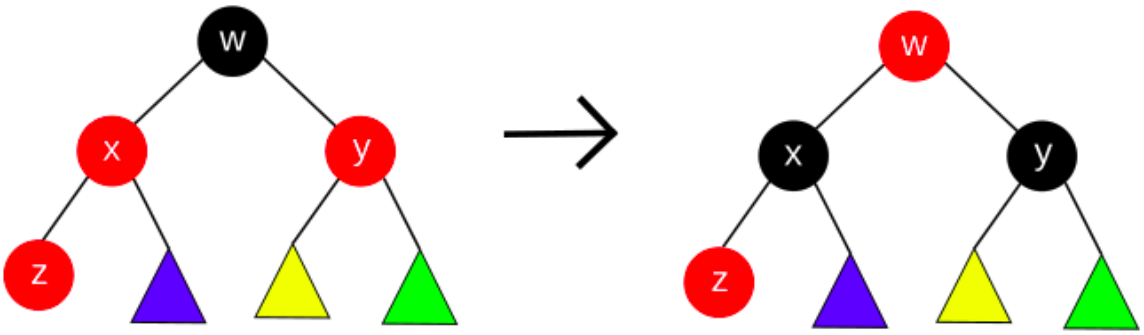


IMAGEM 38: Caso 1b - balanceamentoInsercao

Também pode acontecer de z ser inserido à esquerda de x. O tratamento para esse caso será o mesmo, pois as mudanças de cores envolvem apenas o pai, tio e o avô, logo, a posição do nó “z” não interfere no processo abordado no caso 1.

Caso 2 (linhas 129-133):

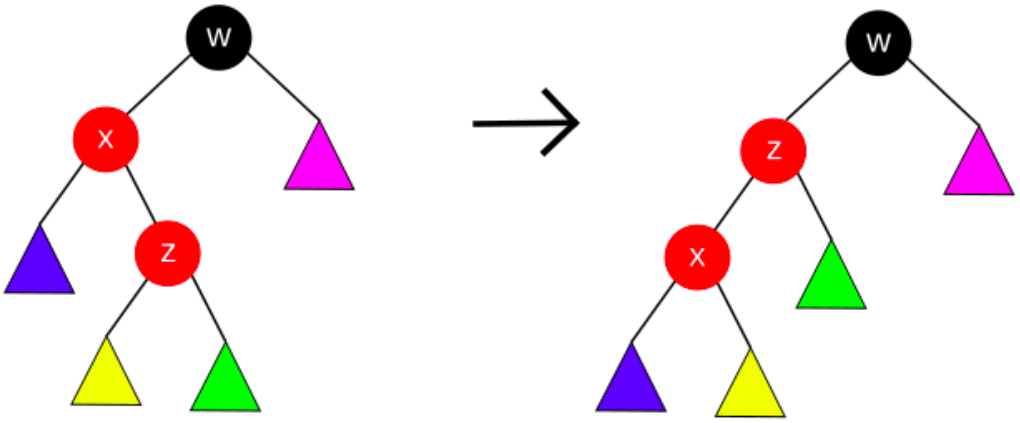


IMAGEM 39: Caso 2 - balanceamentoInsercao

O segundo caso acontece quando o tio do nó Z é preto e Z é o filho à direita do seu pai.

Nesse caso Z, que é o novo nó, irá apontar para o seu pai (linha 131) e na sequência realizaremos uma rotação à esquerda com o novo nó e o nó raiz da árvore. Nota-se que nesse caso não foi necessário realizar mudança nas cores dos nós.

Após a rotação precisaremos entrar no caso 3 só que agora com z tendo o valor de seu pai. Como é possível perceber, o caso 2 é apenas um caso para deixar a árvore no caso 3, e depois tratar como se fosse o caso 3.

Caso 3 (linhas 134-137):

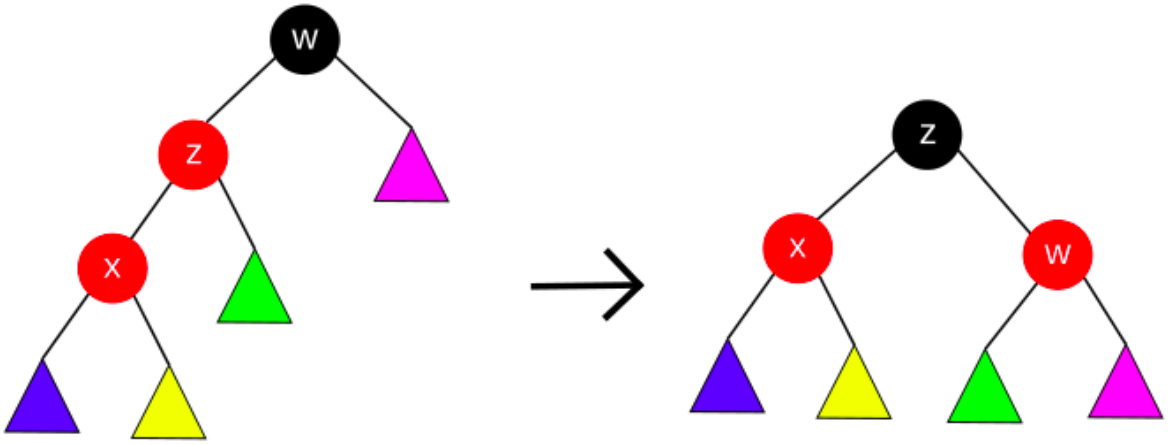


IMAGEM 40: Caso 3 - balanceamentoInsercao

Nesse caso, Z é o pai do novo nó (X) então, podemos dizer que o novo nó e também o seu pai são vermelhos, não respeitando assim a propriedade de que o pai de um nó vermelho deve ser preto.

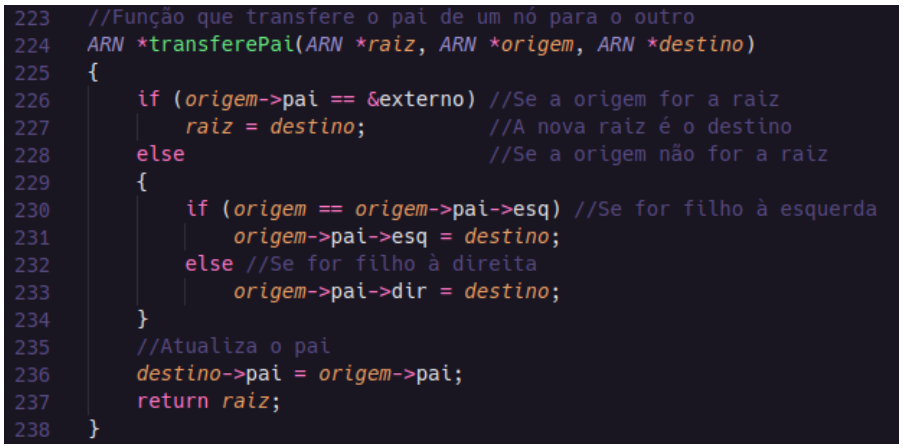
Para manter as propriedades da árvore, precisaremos executar mudança das cores dos nós: o pai do novo nó será preto, o avô vermelho e por fim será realizada uma rotação à direita com o nó raiz e seu avô.

Como o pai do novo nó agora é preto, o while não terá uma nova iteração e assim, basta que a raiz da árvore receba a cor preta já que essa é uma propriedade da árvore rubro-negra: o nó raiz sempre será preto. Depois de realizar esse procedimento, a função retorna o novo nó raiz da árvore.

4.4.11 Transfere pai

A função “transferePai” é uma função bem simples que foi utilizada apenas para evitar repetição de código. Como o próprio nome dela já diz, ela transfere o pai

de um nó de origem para um nó de destino. A implementação dessa função se encontra na imagem abaixo:



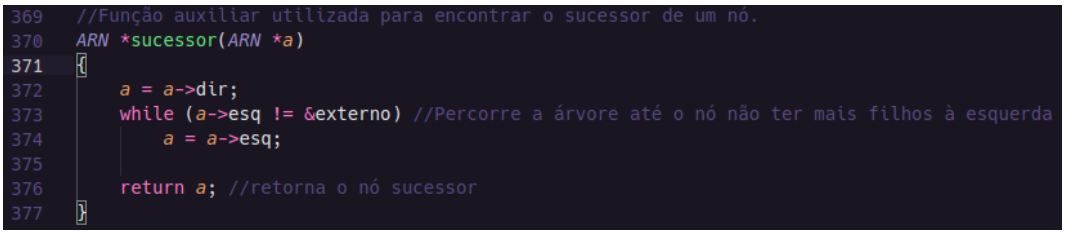
```
223  //Função que transfere o pai de um nó para o outro
224  ARN *transferePai(ARN *raiz, ARN *origem, ARN *destino)
225  {
226      if (origem->pai == &externo) //Se a origem for a raiz
227          raiz = destino;          //A nova raiz é o destino
228      else                         //Se a origem não for a raiz
229      {
230          if (origem == origem->pai->esq) //Se for filho à esquerda
231              origem->pai->esq = destino;
232          else //Se for filho à direita
233              origem->pai->dir = destino;
234      }
235      //Atualiza o pai
236      destino->pai = origem->pai;
237      return raiz;
238  }
```

IMAGEM 41: Implementação da função transferePai

Como é possível observar na implementação acima, algumas verificações são feitas para analisar o que é melhor a ser feito em cada caso. Inicialmente, é verificado se a origem é a raiz da árvore (linha 226). Caso ela seja, a raiz se torna o destino (linha 227). Caso a origem não seja a raiz da árvore, outra verificação é feita. Como o pai da origem pode ter filhos à direita ou à esquerda, é necessário verificar qual desses filhos é a origem (linha 230). Depois dessa verificação, o pai da origem recebe “destino” como o filho que “origem” era (linha 231 ou 233). Por fim, depois que o pai da origem já está apontando para o destino, o pai do destino passa a ser o pai da origem (linha 236).

4.4.12 Sucessor

A função “sucessor” é uma função auxiliar utilizada para encontrar o sucessor de um nó da árvore. Pela definição, o sucessor de um nó é o nó que possui a menor chave que é maior do que a chave atual. A implementação dessa função se encontra na imagem abaixo.



```
369  //Função auxiliar utilizada para encontrar o sucessor de um nó.
370  ARN *sucessor(ARN *a)
371  {
372      a = a->dir;
373      while (a->esq != &externo) //Percorre a árvore até o nó não ter mais filhos à esquerda
374          a = a->esq;
375
376      return a; //retorna o nó sucessor
377  }
```

IMAGEM 42: Implementação da função sucessor

Como é possível perceber, a implementação dessa função é bem simples. Ela recebe um nó da árvore e se desloca para o filho à direita dele (linha 372), que é o primeiro nó maior do que o nó atual. Depois disso, ela se desloca para os filhos à esquerda, que são menores do que o primeiro maior nó, enquanto o nó possui filhos

à esquerda (linhas 373 e 374). Por fim, a função retorna o sucessor do nó que foi recebido por parâmetro.

A imagem abaixo demonstra como ocorre o processo de encontrar o sucessor de um nó.

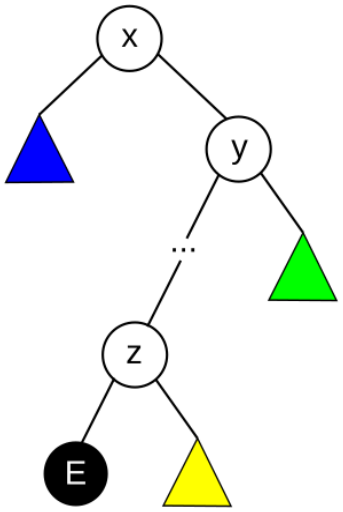


IMAGEM 43: Processo para encontrar o sucessor

Caso se queira achar o sucessor do nó “x”, basta se deslocar para o filho à direita dele (nó “y”) e depois se deslocar para os filhos à esquerda enquanto os nós tiverem filhos à esquerda. De acordo com a imagem, o nó sucessor de “x” seria o nó “z”, pois ele não tem mais filhos à esquerda já que o filho à esquerda dele é o nó externo.

4.4.13 Remoção

A função de remoção da árvore rubro-negra possui os mesmos princípios da remoção na árvore de busca binária, mas ela é acrescida de outros tratamentos para que o balanceamento seja mantido e para que nenhuma propriedade da árvore seja violada.

Na implementação feita da função de remoção, ela não realiza uma busca na árvore para encontrar o nó a ser removido. Ela deve receber por parâmetro um ponteiro para o nó da árvore que contém a chave que se deseja remover. Por isso, antes de chamar a função de remoção é necessário chamar a função de busca para buscar o nó com aquela chave e verificar se tal chave existe na chave antes de tentar removê-la.

Partindo do princípio de que a chave existe na árvore, e o nó que contém aquela chave já foi buscado e foi passado para a função de remoção, será feita a explicação do processo de remoção de uma chave na árvore rubro-negra.

Ao remover um nó da árvore, podem ocorrer alguns casos. Eles serão explicados a seguir.

O primeiro caso ocorre quando o nó a ser removido só possui filho à direita, ou não possui nenhum filho. Esse caso é ilustrado na imagem abaixo.

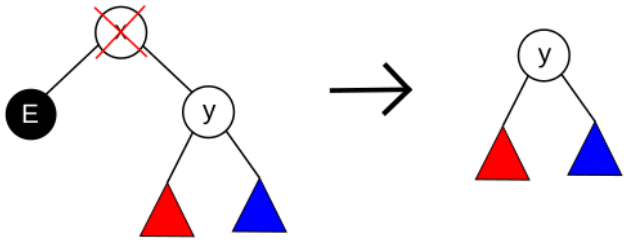


IMAGEM 44: Remoção quando possui apenas subárvore à direita

A imagem ilustra a remoção do nó “x” que possui apenas o filho à direita “y”, já que o filho à esquerda é o nó externo. Antes de realizar a remoção, a subárvore com raiz em “x” estava do jeito ilustrado na imagem à esquerda e depois de realizar a remoção, a árvore se encontra à disposição à direita. Vale ressaltar que os nós estão na cor branca para representar um caso genérico, mas na implementação real os nós “x” e “y” devem possuir a cor preta ou vermelha para que a propriedade 1 da árvore não seja violada. O código desse caso de remoção é apresentado a seguir:

```
325     if (noRemovido->esq == &externo) //Se o nó removido só tiver filho à direita ou não tiver filhos
326     {
327         x = noRemovido->dir; //x aponta pra o filho do nó removido da direita
328         raiz = transferePai(raiz, noRemovido, noRemovido->dir);
329     }
```

IMAGEM 45: Implementação da remoção com subárvore à direita

Ao realizar esse processo de remoção, basta fazer o filho à direita do nó removido ser a nova raiz da subárvore (linha 327) e garantir que o pai do nó removido aponte para o filho à direita do nó removido (linha 328). No código, a variável “x” é apenas uma variável para armazenar a nova raiz da subárvore (mesmo que seja o nó externo) e posteriormente serão feitos os devidos apontamentos nela.

Vale ressaltar que esse trecho de código também funciona para o caso do nó não possuir filhos, pois se ele não possui nenhum filho, ele também não possui filho à esquerda, e o procedimento a ser realizado é o mesmo. A variável “x” irá armazenar o nó sentinela, que é o filho à direita do nó removido (linha 327), e o pai do nó removido será transferido para o nó sentinela (linha 328).

O segundo caso ocorre quando o nó a ser removido só possui filho à esquerda. Esse caso é ilustrado na imagem abaixo.

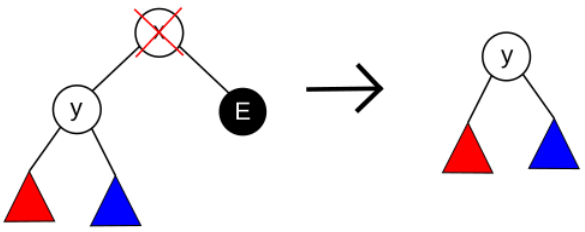


IMAGEM 46: Remoção apenas com subárvore esquerda

A imagem ilustra a remoção do nó “x” que possui apenas o filho à esquerda “y”, já que o filho à direita é o nó externo. Antes de realizar a remoção, a subárvore com raiz em “x” estava da forma ilustrada na imagem à esquerda e depois de realizar a remoção, a árvore se encontra à disposição à direita. Vale ressaltar que os nós estão na cor branca para representar um caso genérico, mas na implementação real os nós “x” e “y” devem possuir a cor preta ou vermelha para que a propriedade 1 da árvore não seja violada. O código desse caso de remoção é apresentado a seguir:

```
332         if (noRemovido->dir == &externo) //Se o nó removido só tiver filho à esquerda
333         {
334             x = noRemovido->esq; //x aponta para o filho do nó removido da esquerda
335             raiz = transferePai(raiz, noRemovido, noRemovido->esq);
336         }
```

IMAGEM 47: Implementação da remoção com subárvore esquerda

Ao realizar esse processo de remoção, basta fazer o filho à esquerda do nó removido ser a nova raiz da subárvore (linha 334) e garantir que o pai do nó removido aponte para o filho à esquerda do nó removido (linha 335). No código, a variável “x” é apenas uma variável para armazenar a nova raiz da subárvore (mesmo que seja o nó externo) e posteriormente serão feitos os devidos apontamentos nela.

O último caso ocorre quando o nó a ser removido possui tanto o filho à esquerda quanto o filho à direita. Nesse caso é necessário encontrar o sucessor do nó a ser removido e podem acontecer duas situações que serão abordadas abaixo.

A primeira situação é quando o sucessor é o filho à direita do nó a ser removido. Essa situação é ilustrada abaixo:

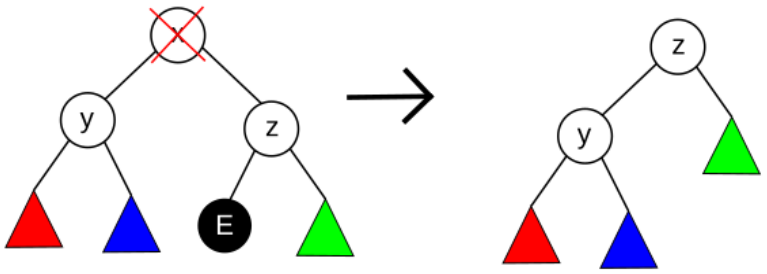


IMAGEM 48: Remoção com sucessor filho do removido

Nessa situação, basta fazer o sucessor ser a raiz da subárvore, e o filho à esquerda dele ser o filho à esquerda do nó removido.

Já o segundo caso é quando o sucessor não é o filho à direita do nó a ser removido. Esse caso se encontra na imagem abaixo.

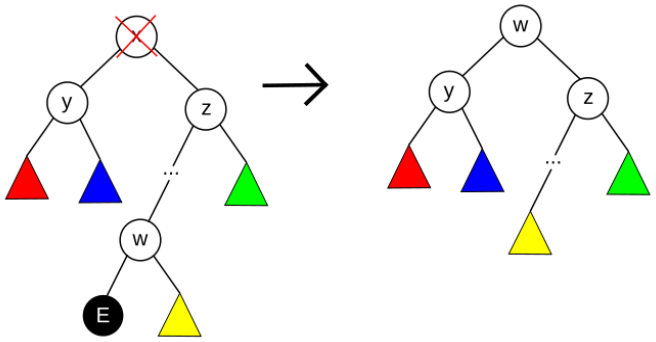


IMAGEM 49: Remoção quando o sucessor não é filho do removido

Nesse caso, é necessário que o sucessor saia da posição original dele e assuma a posição do nó a ser removido. O código que trata o caso do nó a ser removido possuir os dois filhos está na imagem abaixo.

```
337         else //Se o nó removido tiver os dois filhos
338         {
339             y = sucessor(noRemovido); //Encontra o sucessor
340             corDeY = y->cor;          //Salvando a cor de y antes que possa haver alguma mudança
341             x = y->dir;
342             if (y->pai == noRemovido) //Se o nó removido for pai do sucessor
343             {
344                 x->pai = y;
345             }
346             else //Se o nó removido não for pai do sucessor
347             {
348                 raiz = transferePai(raiz, y, y->dir);
349                 y->dir = noRemovido->dir;
350                 y->dir->pai = y;
351             }
352             raiz = transferePai(raiz, noRemovido, y);
353             y->esq = noRemovido->esq;
354             y->esq->pai = y;
355             y->cor = noRemovido->cor;
356         }
```

IMAGEM 50: Implementação da remoção com as duas subárvores

Ao analisar as duas situações possíveis para esse caso, é possível perceber que os apontamentos entre o sucessor e a subárvore esquerda são os mesmos para os dois casos. Em ambos os casos o pai do nó removido deve ser transferido para o sucessor (linha 352), a subárvore esquerda do sucessor passa a ser a subárvore à esquerda do nó removido (linha 353), o pai da subárvore esquerda passa a ser o sucessor (linha 354), e a cor do sucessor deve ser a mesma cor do nó removido (linha 355), já que esse nó vai assumir o lugar do removido. Além disso, é criada uma variável para armazenar a cor do sucessor antes que ele mude de cor (linha 340).

Além desse trecho em comum para as duas situações, é possível perceber que cada situação deve ser tratada de um jeito específico.

A primeira situação apenas requer que o pai do filho à direita do sucessor seja realmente o sucessor pois, se o filho à direita do sucessor for o nó sentinela, pode ser que o ponteiro para o pai desse nó esteja apontando para outro lugar, e

isso irá atrapalhar o processo de balanceamento da remoção que será tratado adiante. Por isso, foi feita a atribuição da linha 344.

Por fim, a segunda situação requer que o sucessor seja retirado do lugar dele e que ele assuma o lugar do nó removido. Para isso, o pai do sucessor deve ser transferido para o filho à direita dele, já que ele deve ser removido daquela posição e ele não tem filhos à esquerda (linha 348), a subárvore à direita do sucessor passa a ser a subárvore à direita do nó removido (linha 349) e o pai da nova subárvore à direita do sucessor passa a ser o sucessor (linha 350).

Depois de tratar cada caso da remoção, o nó já foi removido com sucesso, mas dependendo da cor do sucessor, pode ser que tenha acontecido algum desbalanceamento ou alguma violação das propriedades da árvore rubro-negra. Entretanto, essas propriedades só são violadas para o caso da subárvore só ter um filho e a cor do nó removido era preta, ou quando a subárvore tem dois filhos e a cor do sucessor era preta.

Quando o nó a ser removido é preto e só tem um filho, ele é removido e pode ser que a altura de nós pretos da árvore seja alterada. Já quando o nó a ser removido possui dois filhos, se o sucessor for preto, ao trocar ele de posição com o nó removido pode ser que haja alteração da altura preta caso o nó removido fosse vermelho. Por isso, a função “balanceamentoRemocao” deve ser chamada para balancear a árvore novamente de forma que nenhuma propriedade dela seja violada. Depois disso, a memória alocada para a chave do nó removido é liberada por meio da função de callback “liberaChave”, o nó da árvore é liberado por meio do “free” e a nova raiz da árvore é retornada. O trecho de código que trata a chamada da função “balanceamentoRemocao” e a liberação da memória do nó se encontra abaixo:

```
358     if (corDeY == 0)
359     {
360         raiz = balanceamentoRemocao(raiz, x); //Restaurar as propriedades de balanceamento e cor
361     }
362     //Libera a memória alocada para noRemovido
363     liberaChave(noRemovido->chave); //Libera a memória alocada para noRemovido
364     free(noRemovido);
365     return raiz;
366 }
```

IMAGEM 51: Chamada do balanceamento da remoção e liberação de memória

Vale ressaltar que a variável corDeY é a variável que armazena a cor do nó removido se ele possuir apenas um filho, ou a cor do sucessor caso o nó removido tenha os dois filhos. Ela é usada para verificar se essa cor é preta e realizar o balanceamento caso a cor seja preta.

4.4.14 Balanceamento remoção

Após realizar a remoção, pode ser que ocorra a violação de alguma propriedade da árvore rubro-negra. Por isso, a função “balanceamentoRemocao” é chamada ao fim da função de remoção como foi explicado anteriormente. Essa função garante que após a remoção a árvore continue balanceada e a altura de nós pretos de todas as subárvores seja a mesma. Para isso, ela altera as cores de alguns nós e realiza algumas rotações em alguns casos. Vale ressaltar que a função só é chamada quando a altura de nós pretos pode ter sido alterada na árvore. Assim sendo, essa cor preta deve ser compensada de alguma forma. Por isso, assume-se que o nó “x” possui uma cor preta extra e a cada iteração do laço, são tratados cada caso até chegar a um ponto em que se está na raiz, ou se chegou em um nó vermelho e basta colorir esse nó de preto. A seguir serão explicados cada caso tratado nesta função. Nas imagens, o nó “x” é o nó que possui a cor preta “a mais”.

O primeiro caso tratado é o caso em que o irmão de “x” é vermelho. Esse caso está ilustrado abaixo:

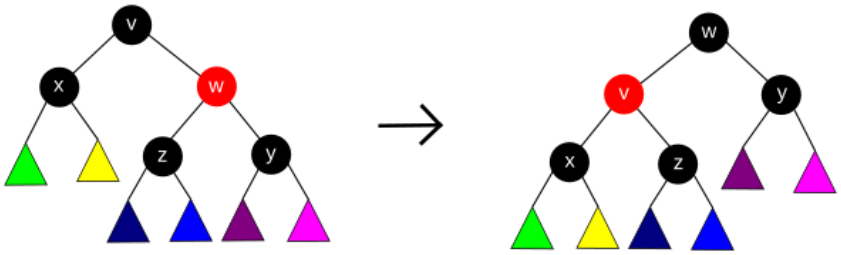


IMAGEM 52: Caso 1 - balanceamentoRemocao

Esse caso é um caso de transição que irá converter esse caso no caso 2. O código que trata esse caso foi apresentado abaixo:

```
249             if (irmao->cor == 1) //Caso 1
250             {
251                 irmao->cor = 0;
252                 x->pai->cor = 1;
253                 raiz = rotacaoEsquerda(raiz, x->pai); //Necessário rotação
254                 irmao = x->pai->dir;
255             }
```

IMAGEM 53: Implementação do caso 1 da função balanceamentoRemocao

Ele consiste basicamente em trocar a cor do irmão de “x” com a cor do pai dele (linhas 251 e 252) e depois realizar uma rotação à esquerda no pai de “x” (linha 253). Isso fará com que o novo irmão de “x” seja o filho à esquerda do antigo irmão dele, e isso irá levar aos outros casos. No código (linha 254) o novo irmão passa a ser o filho à direita do pai de “x” pois já ocorreu a rotação, e a árvore já se encontra na sua nova organização. Assim sendo, o novo irmão de “x” após a rotação será o

filho à direita do pai dele, já que “x” é o filho à esquerda do pai dele. Os outros casos ocorrem quando o irmão de “x” for preto.

O segundo caso acontece quando o irmão de “x” é preto e possui filhos pretos. Esse caso está ilustrado na imagem abaixo.

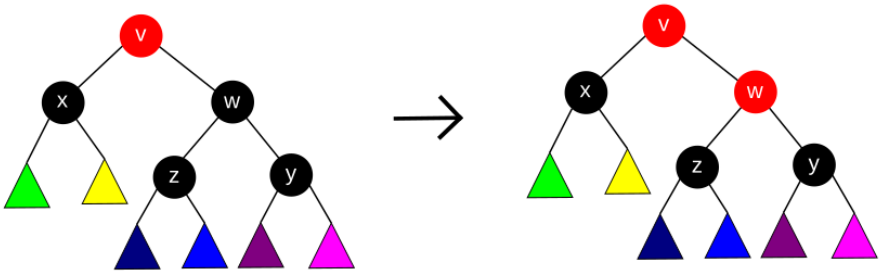


IMAGEM 54: Caso 2 - balanceamentoRemocao

Esse tipo de caso pode acontecer naturalmente ou pode ser um resultado do caso 1. De qualquer forma, nesse caso, para suprir o nó que foi retirado de “x” e consertar a altura preta da árvore basta colorir o irmão de “x” de vermelho. Isso já resolve o problema. O código para esse caso se encontra na imagem abaixo:

```
256             if (irmao->esq->cor == 0 && irmao->dir->cor == 0) //Caso 2
257             {
258                 irmao->cor = 1;
259                 x = x->pai;
260             }
```

IMAGEM 55: Implementação do caso 2 da função balanceamentoRemocao

Como é possível perceber, no código foi verificado se o irmão possui filhos pretos (linha 256). Em caso afirmativo, a cor do irmão se torna vermelha (linha 258). Como os casos estão sendo tratados em um laço de repetição, o “x” é alterado para o pai dele para que na próxima iteração do loop possa ser verificado se o pai de “x” quebra alguma propriedade da árvore.

O terceiro caso também é um caso de transição que prepara a árvore para o caso 4. Ele acontece quando o irmão de “x” é preto e possui filho à esquerda vermelho e filho à direita preto. Esse caso é ilustrado na imagem abaixo.

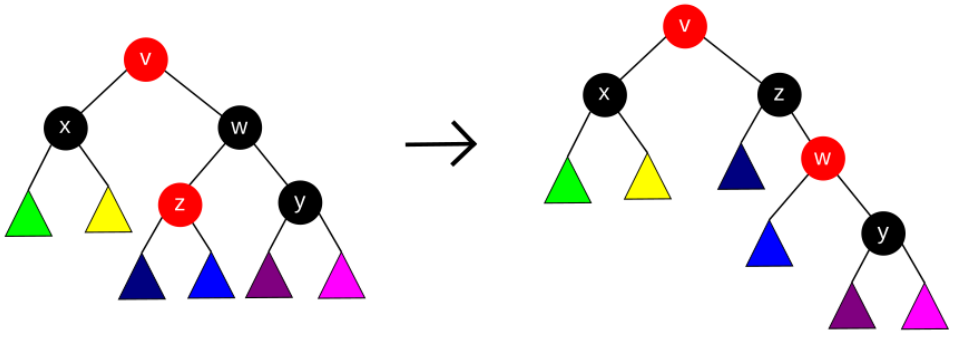


IMAGEM 56: Caso 3 - balanceamentoRemocao

Como é possível perceber na imagem, o irmão de “x” é preto e ele possui filho à esquerda vermelho e o filho à direita preto. Para preparar esse caso para o caso 4, é necessário realizar uma rotação à esquerda no irmão de “x”. O código para esse caso se encontra na imagem abaixo:

```
263             if (irmao->dir->cor == 0) //Caso 3
264             {
265                 irmao->esq->cor = 0;
266                 irmao->cor = 1;
267                 raiz = rotacaoDireita(raiz, irmao); //Necessário rotação
268                 irmao = x->pai->dir;
269             }
```

IMAGEM 57: Implementação do caso 3 da função balanceamentoRemocao

Como é possível perceber no código, inicialmente é verificado se o filho à direita do irmão de “x” é preto (linha 263). Depois é realizada a troca de cores do filho à esquerda do irmão com o irmão (linha 265 e 266) e é realizada uma rotação à direita no irmão (linha 267). Por fim, depois de realizar a rotação, o irmão de “x” se torna o filho à esquerda do antigo irmão. Como a rotação já aconteceu, o novo irmão de “x” é o filho à direita do pai dele, já que o “x” é o filho à esquerda do pai dele (linha 268).

Por fim, o último caso ocorre quando o irmão de “x” é preto e o filho à direita dele é vermelho. Esse caso é ilustrado na imagem abaixo.

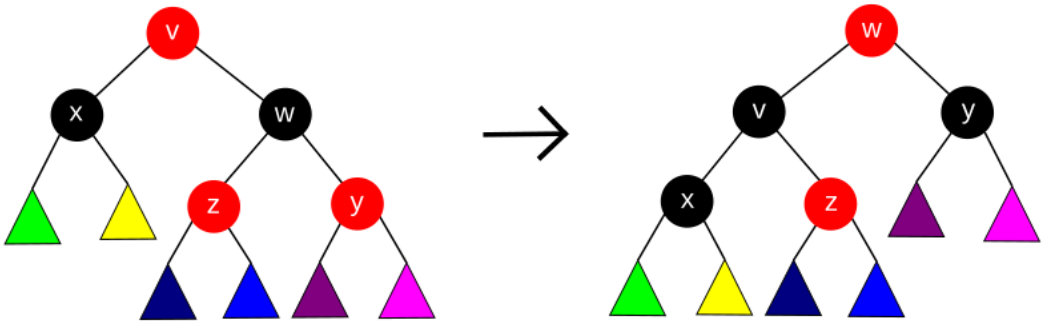


IMAGEM 58: Caso 4 - balanceamentoRemocao

É possível chegar nesse caso de forma natural ou por meio da aplicação do caso 3 na árvore. Para que esse caso ocorra, basta que o filho à direita do irmão de “x” seja vermelho, o filho à esquerda pode ser de qualquer cor. A implementação desse caso se encontra na imagem abaixo:

```
270             //Caso 4
271             irmao->cor = x->pai->cor;
272             x->pai->cor = 0;
273             irmao->dir->cor = 0;
274             raiz = rotacaoEsquerda(raiz, x->pai); //Necessário rotação
275             x = raiz;
```

IMAGEM 59: Implementação do caso 4 da função balanceamentoRemocao

Nesse caso, o irmão recebe a cor do pai (linha 271), o pai recebe a cor preta (linha 272), o filho à direita do irmão recebe a cor preta (linha 273), ocorre uma rotação à esquerda no pai de “x” (linha 274), e “x” passa a apontar para a raiz (linha 275) para indicar que foi finalizado o processo de restabelecimento das propriedades da árvore rubro-negra e encerra o loop.

Os algoritmos explicados acima servem para o caso do nó “x” ser o filho à esquerda do pai dele, mas quando “x” é o filho à direita do pai dele, acontecem exatamente os mesmos casos, mas de forma simétrica. Basicamente basta trocar a palavra “esquerda” pela palavra “direita” e vice versa na explicação e é possível obter todos os casos e explicações para quando “x” é o filho à direita do pai dele. O código para esse caso está na imagem abaixo.

```
280         ARN *irmao = x->pai->esq; //Inicializa o irmão
281
282         if (irmao->cor == 1) //Caso 1
283         {
284             irmao->cor = 0;
285             x->pai->cor = 1;
286             raiz = rotacaoDireita(raiz, x->pai); //Necessário rotação
287             irmao = x->pai->esq;
288         }
289         if (irmao->dir->cor == 0 && irmao->esq->cor == 0) //Caso 2
290         {
291             irmao->cor = 1;
292             x = x->pai;
293         }
294         else
295         {
296             if (irmao->esq->cor == 0) //Caso 3
297             {
298                 irmao->dir->cor = 0;
299                 irmao->cor = 1;
300                 raiz = rotacaoEsquerda(raiz, irmao); //Necessário rotação
301                 irmao = x->pai->esq;
302             }
303             //Caso 4
304             irmao->cor = x->pai->cor;
305             x->pai->cor = 0;
306             irmao->esq->cor = 0;
307             raiz = rotacaoDireita(raiz, x->pai); //Necessário rotação
308             x = raiz;
309         }
```

IMAGEM 60: Implementação da função balanceamentoInsercao quando “x” é o filho à direita do pai dele

Vale ressaltar que ao fim do processo, a cor da raiz da árvore passa a ser preta e a nova raiz da árvore é retornada.

5 CONCLUSÃO

Portanto, é possível concluir que o que permite que a árvore rubro-negra apresente um bom desempenho constante nas operações de busca, inserção e remoção é o fato dela ser balanceada por meio das propriedades dela e das cores dos nós. Entretanto, para garantir o balanceamento dela é preciso realizar algumas operações de rotação e modificar as cores em cada caso como foi descrito no decorrer do trabalho. Além disso, foi possível perceber que, mesmo que os algoritmos sejam mais complexos de se implementar, o fato da árvore ser balanceada e um pouco regrada acelera o processo de busca. Além disso, como a árvore rubro-negra é um pouco mais flexível quanto a diferença de altura das folhas como a árvore AVL, ela é um pouco mais rápida para processos como inserção e remoção. Por fim, o uso do nó sentinela, chamado de “nó externo” durante o trabalho foi de grande importância para a implementação pois evitou que várias verificações fossem feitas, reduziu consideravelmente o tamanho do código das funções e impediu que ocorram possíveis falhas de segmentação que deveriam ser tratadas caso não fosse implementado com o nó sentinela.

6 REFERÊNCIAS

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Algoritmos**: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

https://pt.wikipedia.org/wiki/%C3%81rvore_rubro-negra

https://pt.wikipedia.org/wiki/C_(linguagem_de_programa%C3%A7%C3%A3o)