UNIVERSIDADE FEDERAL DO ESPIRITO SANTO

CIÊNCIA DA COMPUTAÇÃO

UFES

IASMIN MARQUES PEREIRA

RAFAEL MENDES MERLO

RELATÓRIO TRABALHO – CONTROLE DE ESTOQUE

(ÁRVORE RUBRO NEGRA – RECURSIVA)

ESTRUTURA DE DADOS II

SÃO MATEUS
2023

IASMIN MARQUES PEREIRA

RAFAEL MENDES MERLO

RELATÓRIO TRABALHO – CONTROLE DE ESTOQUE (ÁRVORE RUBRO NEGRA – RECURSIVA)

ESTRUTURA DE DADOS II

Relatório do trabalho de estrutura de dados II sobre controle de estoque através da estrutura da Árvore Rubro apresentado ao professora Luciana Lee, como requisito para obtenção de nota da Universidade Federal do Espírito Santo – Campus São Mateus.

PROFESSORA: **Luciana Lee**

SÃO MATEUS
2023

SUMÁRIO

1. Justificativa

Este trabalho possui como intuito implementar a árvore rubro-negra de forma que os nós não tenham ponteiro para o nó pai, atendendo as seguintes operações: cadastro de um novo produto, exclusão um produto cadastrado, atualização da quantidade de um produto no estoque, listar todos os produtos cadastrados, listar todos os produtos em estoque, impressão da árvore Rubro-Negra.

2. Objetivos

- ✓ O programa deverá ser implementado em Linguagem C e totalmente comentado.
- ✓ O relatório deverá conter a explicação detalhada do programa (forma de compilação do programa, estruturas, funções, formatos de entrada e saída de dados).
- ✓ O programa deverá ter as operações de cadastro de novo produto, exclusão de um produto cadastrado, atualização da quantidade de um produto no estoque, listar todos os produtos cadastrados, listar todos os produtos disponíveis no estoque e por fim a impressão da árvore Rubro-Negra.
- ✓ A entrega da implementação e do relatório é obrigatória. A falta de qualquer uma dessas partes leva à anulação do trabalho (nota zero).
- ✓ A estrutura de um nó da árvore não pode ter ponteiro para o nó pai (e nem para nenhum ancestral na árvore

3. Introdução

Visando o objetivo geral e os específicos decidimos dividir o desenvolvimento do trabalho em alguns tópicos, sendo eles: a implementação da árvore rubro negra (sem o ponteiro para o pai), operações de inserção e exclusão do nó da arvore rubro negra e por fim as implementações e adições no código das informações relacionadas ao produto a ser controlado pelo estoque. A organização da implementação foi feita através de um tipo abstrato de dados (TAD), em três arquivos:

- **main.c:** contém o menu que sera exibido para o usuário com as opções disponíveis de controle de estoque, produtos e impressão da árvore rubro negra;
- **rbt.c:** nesse arquivo contém todas as implementações relacionadas a estrutura da árvore rubro negra, inserção, exclusão, balanceamento, rotações, função de listagem de produtos cadastrados, impressão de produtos no estoque, impressão da árvore rubro negra, entre outras funções auxiliares que nos ajudaram nas operações/ funções principais;
- **rbt.h:** onde contém todos os headers/ cabeçalho das funções;

4. Metodologia

4.1 Implementação da Árvore Rubro Negra

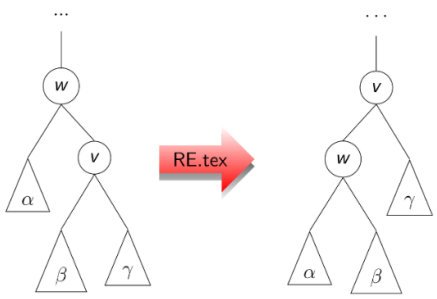
Implementação da Árvore Rubro negra recursiva por conta da limitação de não pode ter um ponteiro para seu pai; Para uma árvore ser considerada uma árvore rubro negra ela precisa preencher alguns requisitos, sendo:

1. Todo nó da árvore é vermelho ou preto;
2. A raiz é sempre preta;
3. Todo nó folha (nó externo/ NULL) é preto;
4. Se um nó é vermelho, então os seus filhos são pretos (Não existem nós vermelhos consecutivos);
5. Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém;

O balanceamento da Árvore Rubro Negra é feito através da altura preta da árvore, da quantidade de NOS pretos presentes na árvore. Durante o balanceamento da árvore faremos através das rotações e da mudança de cor dos nós caso necessário.

Sobre as rotações, existem apenas duas funções de rotação: rotação à esquerda e a rotação à direita;

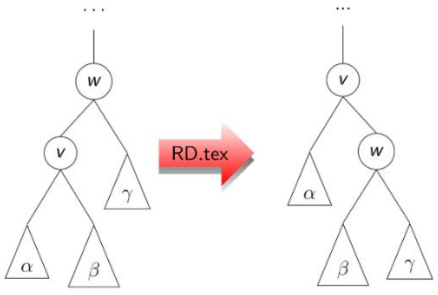
- Rotação a Esquerda




```c
struct NO *rotateLeftRb(struct NO *root){ // Rotação a esquerda
    struct NO *newRoot = root->right;
    struct NO *newRootLeft = newRoot != NULL ? newRoot->left : NULL;

    root->right = newRootLeft;
    newRoot->left = root;

    newRoot->color = root->color; // Atualizar as cores corretamente
    root->color = RED;

    if (newRootLeft){ // Atualizar as cores dos filhos
        newRootLeft->color = BLACK;
    }

    return newRoot;
}
```

Figura 1 - Rotação a esquerda

- Rotação à Direita

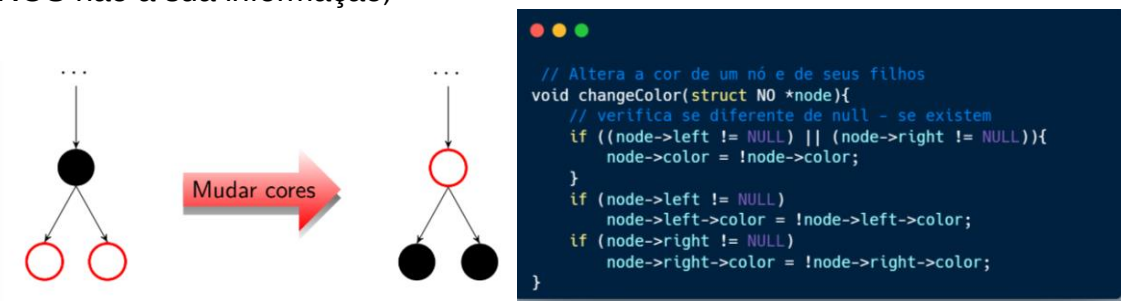



```c
struct NO *rotateRightRb(struct NO *root){ // Rotação a Direita
    struct NO *newRoot = root->left;
    struct NO *newRootRight = newRoot != NULL ? newRoot->right : NULL;

    root->left = newRootRight;
    newRoot->right = root;

    newRoot->color = root->color; // Atualizar as cores corretamente
    root->color = RED;

    if (newRootRight){ // Atualizar as cores dos filhos
        newRootRight->color = BLACK;
    }

    return newRoot;
}
```

Figura 2- Rotação a direita

- Mudança de Cor da Árvore

Podemos ter a necessidade de mudar a cor de um nó e de seus filhos de vermelho para preto e vice-versa. A função implementada só troca a cor dos NOS não a sua informação;




```c
// Altera a cor de um nó e de seus filhos
void changeColor(struct NO *node){
    // verifica se diferente de null - se existem
    if ((node->left != NULL) || (node->right != NULL)){
        node->color = !node->color;
    }
    if (node->left != NULL)
        node->left->color = !node->left->color;
    if (node->right != NULL)
        node->right->color = !node->right->color;
}
```

Figura 3 - Mudar Cores

4.1.1 Inserção

Na implementação da inserção dividimos o codigo em duas funções principais, a “inserRb” e a “insertNodeRb”. Sendo a “inserRb” uma função auxiliar da inserção, que sera chamada no arquivo “main.c” quando o usuário digitar a opção número um, de inserir um novo produto; já na “insertNodeRb” é feito o tratamento das possiveis violações de uma arvore rubro negra.

```c
int insertRb(RbTree *root, int current, int qtd, const char *name, char **name_prod){
//função auxiliar a inserção do produto na árvore
    int ans;
    free(*name_prod);// Libera a memória anterior, se necessário
    // Aloca memória suficiente para a nova string
    *name_prod = (char *)malloc((strlen(name) + 1) * sizeof(char));

    int i = 0;
    // tratamento de inserção de um tipo char/ string
    while (name[i] != '\0'){
        if (name[i] == ' '){
            (*name_prod)[i] = '\0';
        }
        (*name_prod)[i] = name[i];
        i++;
    }
    printf("%s", *name_prod);

    *root = insertNodeRb(*root, current, qtd, *name_prod, &ans); // Faz o tratamento
das possíveis violações da árvore rubro-negra

    if ((*root) != NULL) // Seta a cor da raiz para preta, se necessário
        (*root)->color = BLACK;

    return ans;
}
```

Figura 4 - Função auxiliar de Inserção

Existem três casos possíveis que podem violar as propriedades da árvore rubro negra, são eles:

Caso 1: o tio de q é Rubro.

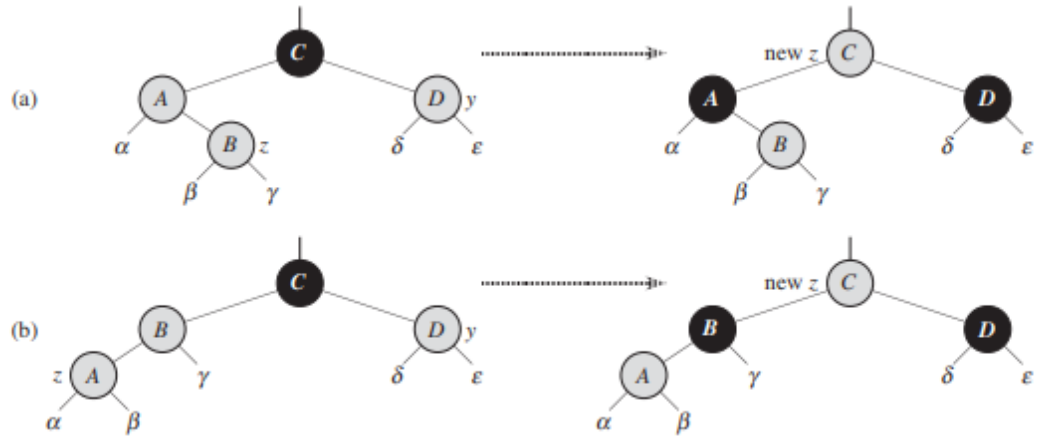


Figura 5 - Violação Caso 1

Caso 2: O tio de q é Negro e q é filho a direita de seu pai

Caso 3: o tio de q é Negro e q é é filho a esquerda de seu pai.

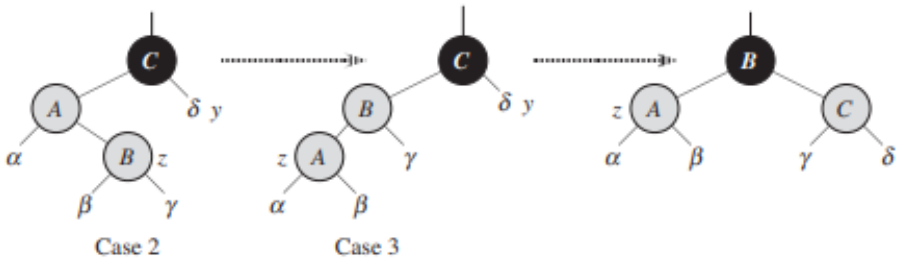


Figura 6 - Casos 2 e 3 de violação

```c
struct NO *insertNodeRb(struct NO *root, int key, int qtd, const char *name, int *ans){
  //função responsavel pela verificação de violações e inserção do no na arvore
    if (!root){
        struct NO *insert = (struct NO *)malloc(sizeof(struct NO));
        if (!insert){ // No não foi inserido
            *ans = 0;
            return NULL;
        }
        insert->info = key; //info recebe o numero da chave do produto
        insert->color = RED; // no inicia com a cor Vermelha
        insert->left = insert->right = NULL; //filhos nulos
        insert->qtd_prod = qtd; //quantidade do produto recebe a quatidade inserida pelo usuário
       // Adicionar o nome do produto
        insert->name_prod = (char *)malloc((strlen(name) + 1) * sizeof(char));
        strcpy(insert->name_prod, name);
        *ans = 1;
        return insert;
    }
    if (key == root->info){ // Se a chave já existir, retorna NULL
        *ans = 0;
        return NULL;
    }
    if (key < root->info){ //Chave do produto menor do que a atual, desce para esquerda
        root->left = insertNodeRb(root->left, key, qtd, name, ans);
    }
    else{  //Chave do produto maior do que a atual, desce para direita
        root->right = insertNodeRb(root->right, key, qtd, name, ans);
    }
    // tratamento de casos de violação da inserção
    if (*ans){ //checa o balanceamento da arvore
        if (key < root->info){ //Chave do produto menor do que a atual, desce para esquerda
            if (root->left && getColor(root->left) == RED && getColor(root->left->left) == RED) {
            // no adicionado à esquerda e irmão com cor vermelha
              if (root->right && getColor(root->right) == RED){
                    root->color = RED;
                    root->left->color = BLACK;
                    root->right->color = BLACK;
                }else{
                    root = rotateRightRb(root);
                    root->right->color = RED;
                    root->color = BLACK;
                }
            }else if (root->left && getColor(root->left) == RED && getColor(root->left->right) == RED){
                if (root->right && getColor(root->right) == RED){// pai e tio vermelhos
                    root->color = RED;
                    root->left->color = BLACK;
                    root->right->color = BLACK;
                }else{
                    root->left = rotateLeftRb(root->left);
                    root = rotateRightRb(root);
                    root->color = BLACK;
                    root->right->color = RED;
                }
            }
        }else{ //Chave do produto maior do que a atual, desce para direita
            if (root->right && getColor(root->right) == RED && getColor(root->right->right) == RED) {
            //no adicionado à direita e irmão com cor vermelha
              if (root->left && getColor(root->left) == RED){
                    root->color = RED;
                    root->left->color = BLACK;
                    root->right->color = BLACK;
                }else{
                    root = rotateLeftRb(root);
                    root->left->color = RED;
                    root->color = BLACK;
                }
            }else if (root->right && getColor(root->right) == RED && getColor(root->right->left) == RED){
                if (root->left && getColor(root->left) == RED){//pai e tio vermelhos
                    root->color = RED;
                    root->left->color = BLACK;
                    root->right->color = BLACK;
                }else{
                    root->right = rotateRightRb(root->right);
                    root = rotateLeftRb(root);
                    root->color = BLACK;
                    root->left->color = RED;
                }
            }
        }
    }
    return root;
}
```

Figura 7 - Função que lida com a Inserção e possíveis violações

4.1.2 Exclusão

Na implementação da exclusão dividimos o codigo em duas funções principais, a “removeRb” e a “removeElementRb”. Sendo a “removeRb” uma função auxiliar da remoção, que sera chamada no arquivo “main.c” quando o usuário digitar a opção número dois, de excluir um produto; já na “removeElementRb”, e na função secundária auxiliar “dellBalanceNodes”, é feito o tratamento das possiveis violações de uma arvore rubro negra quando uma chave é excluida.

```c
int removeRb(RbTree *root, int current){ //função auxiliar da remoção de elementos da árvore
    if (searchElement(*root, current)){ //elemento encontrado
        *root = removeElementRb(*root, current); // Atualiza o ponteiro raiz após a remoção

        if (*root != NULL) // Define a cor da nova raiz como preta
            (*root)->color = BLACK;
        return 1;
    }else{ //elemento não foi encontrado
        return 0;
    }
}
```

Figura 8 - Função auxiliar de remoção

```c
struct NO *removeElementRb(struct NO *node, int current){
  //função que faz a exclusão do No e faz as verificações de violação da árvore
    if (current < node->info){//O valor é menor do que o valor do nó atual
        if (getColor(node->left) == BLACK && getColor(node->left->left) == BLACK)
            node = moveRedToLeft(node);
        node->left = removeElementRb(node->left, current);
    }else if (current > node->info){ //O valor é maior do que o valor do nó atual
        if (getColor(node->left) == RED)
            node = rotateRightRb(node);
        if (getColor(node->right) == BLACK && getColor(node->right->left) == BLACK)
            node = moveRedToRight(node);
        node->right = removeElementRb(node->right, current);
    }else{ // valor == atual
        if (current == node->info && node->left == NULL && node->right == NULL){
          // Verifica se é uma folha, remove o no
            free(node);
            return NULL;
        }
        if (node->right != NULL && getColor(node->right) == BLACK && node->right->left != NULL && getColor(node-
>right->left) == BLACK)
            node = moveRedToRight(node);
        if (current == node->info){
            if (node->left != NULL){ //Substituindo o valor do nó atual pelo
                                     //maior elemento da subárvore esquerda
                struct NO *x = searchLargest(node->left);
                node->info = x->info;
                node->left = removeElementRb(node->left, x->info);
            }else if (node->right != NULL){ //O nó atual não tem filho esquerdo, substituindo
                                            //o valor pelo menor elemento da subárvore direita
                struct NO *x = searchSmallest(node->right);
                node->info = x->info;
                node->right = removeElementRb(node->right, x->info);
            }else{                          //O nó atual não tem filhos, removendo o nó
                free(node);
                return NULL;
            }
        }
    }
    if (node == NULL){ // Verifica se o nó atual é NULL
        return NULL;
    }
    node = dellBalanceNodes(node); // Ajusta o balanceamento do nó após a remoção (função
                                   //auxiliar de verificação de violações apos remoção)
    return node;
}
```

Figura 9 - Função que lida com a Exclusão e possíveis violações

A remoção é um pouco mais complicada de lidar, comparando a com a inserção, os casos de violação da remoção são:

Caso 1: No removido preto com filho vermelho

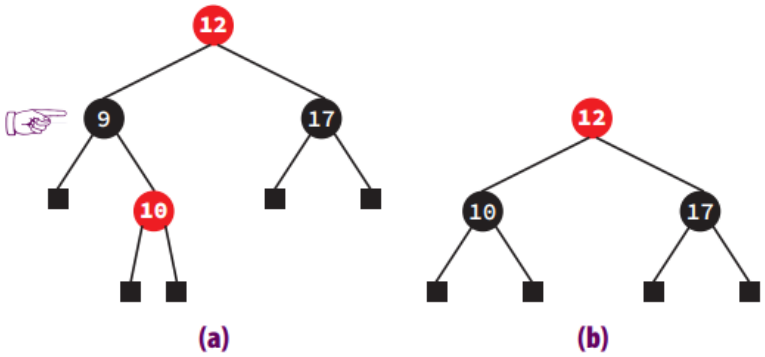


Figura 10 – Exemplo de Caso 1 da remoção

Caso 2: Irmão Preto e Sobrinho Preto

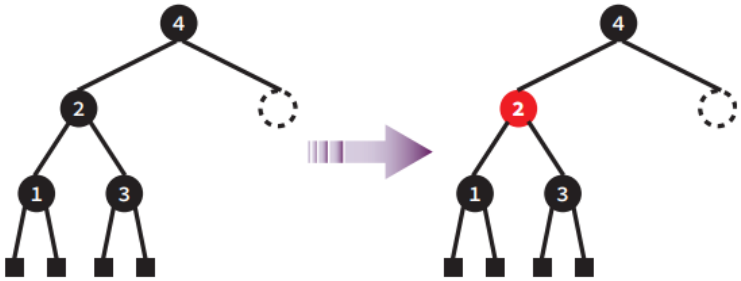


Figura 11 - Exemplo de Caso 2 da remoção

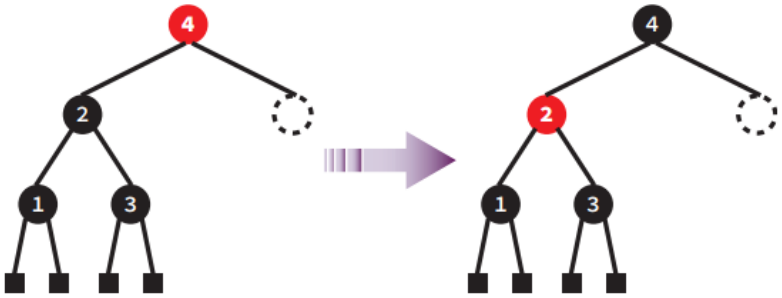


Figura 12 - Exemplo de Caso 2 da Remoção

Caso 3: Irmão Preto e Sobrinho(s) Vermelho(s)

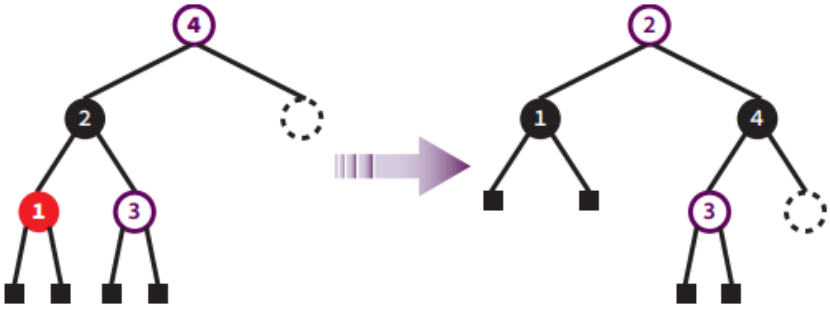


Figura 13 - Exemplo do caso 3 de remoção

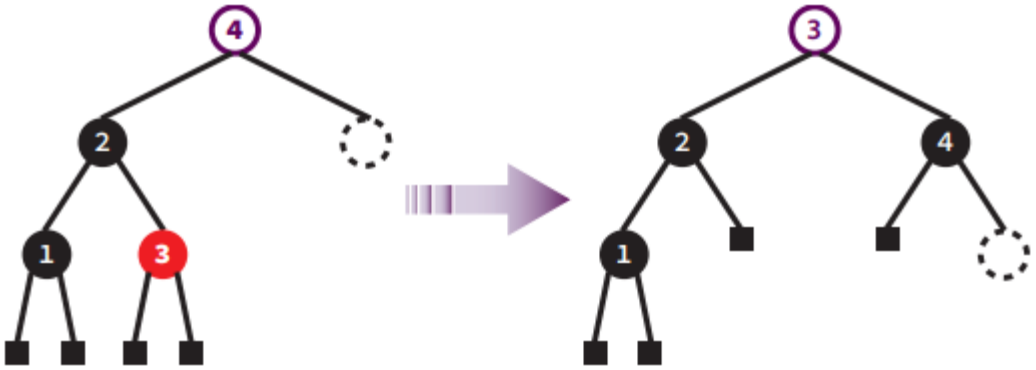


Figura 14 - Exemplo de caso 3 de remoção

Caso 4: O Irmão do Nó Removido É Vermelho

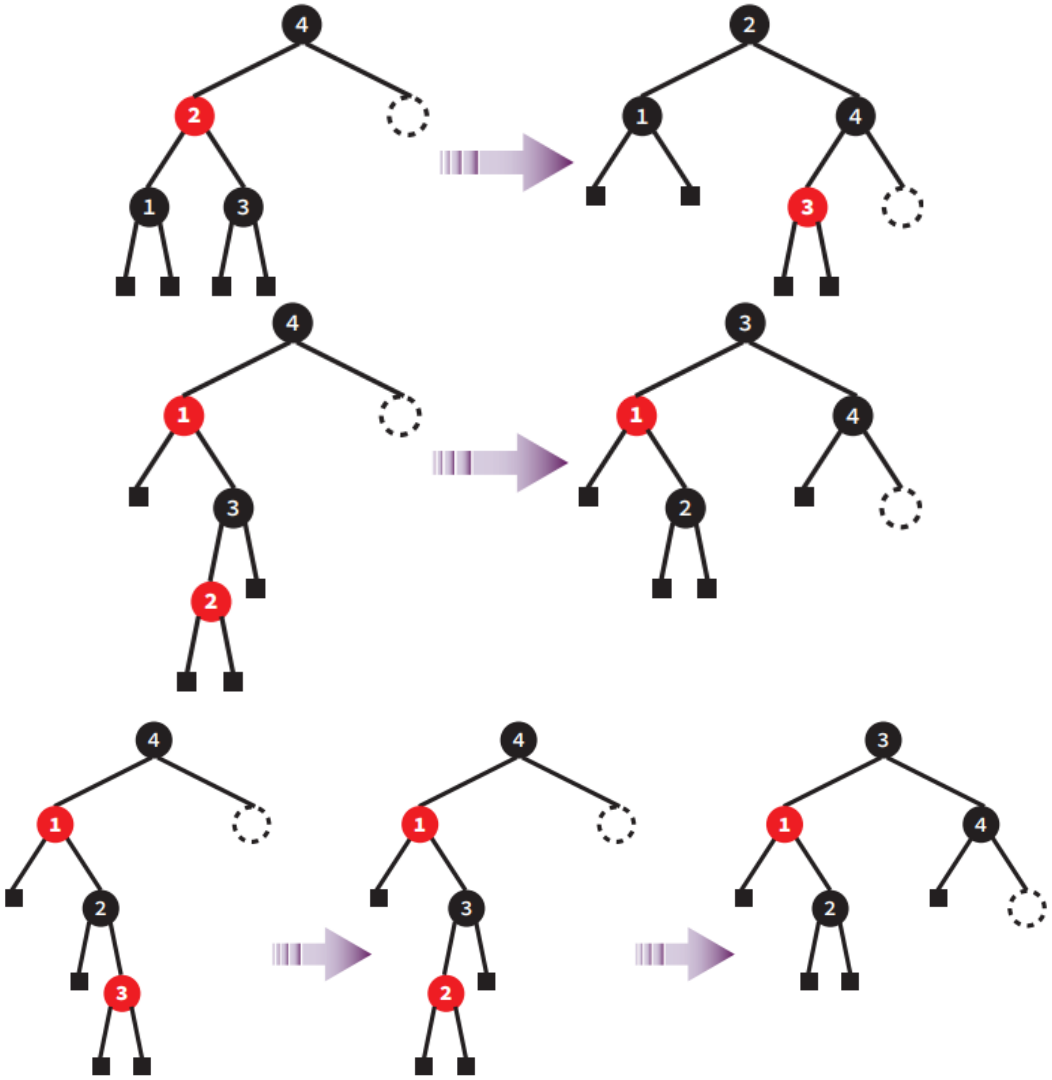


Figura 15 - Exemplo de caso 4 de remoção

```c
struct NO *dellBalanceNodes(struct NO *node){ //ultima verificação de balanceamento apos a remoção
    if (node->left == NULL && node->right != NULL && getColor(node->right) == BLACK){
      //trocou a cor direita
        node->right->color = RED;
    }
    if (node->right == NULL && node->left != NULL && getColor(node->left) == BLACK){
      //trocou a cor esquerda
        node->left->color = RED;
    }
    if (node->right != NULL && getColor(node->right) == RED && getColor(node->right->right) == RED){
       // O filho direito e o neto à direita são vermelhos
        node = rotateLeftRb(node);
        node->color = RED;
    }

    // Nó Vermelho com dois filhos Vermelhos: trocar as cores
    if (node->left != NULL && node->right != NULL && getColor(node) == RED && getColor(node->left)
== RED && getColor(node->right) == RED && node->left->right != NULL){
        node->color = BLACK;        // Cor do nó atual é definida como preto
        node->left->color = BLACK;  // Cor do filho da esquerda é definida como preto
        node->right->color = BLACK; // Cor do filho da direita é definida como preto
        node->left->right->color = RED;
    }
    return node;
}
```

Figura 16 - Função auxiliar de balanceamento

Também é utilizado para auxiliar no balanceamento da árvore rubro negra após a exclusão de um produto as funções de mover um no vermelho para esquerda e para direita.

```c
struct NO *moveRedToLeft(struct NO *node){ //move o no vermelho para esquerda
    changeColor(node);
    if (node->right != NULL && getColor(node->right->left) == RED){
        node->right = rotateRightRb(node->right);
        node = rotateLeftRb(node);
        changeColor(node);
    }
    return node;
}
```

Figura 17 - Implementação da função moveRedToLeft

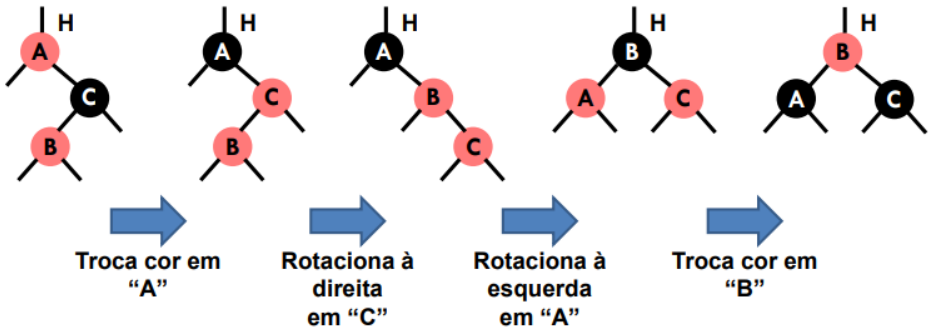


Figura 18 - Exemplo de utilização da função moveRedToLeft

```c
struct NO *moveRedToRight(struct NO *node){ //move o no vermelho para direita
    changeColor(node);
    if (node->left != NULL && getColor(node->left->left) == RED){
        node = rotateRightRb(node);
        changeColor(node);
    }
    return node;
}
```

Figura 19 - Implementação da função moveRedToRight

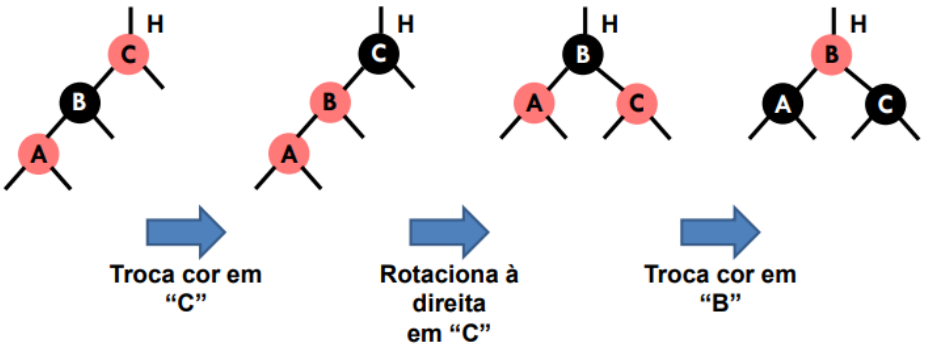


Figura 20 - Exemplo de utilização da função moveRedToRight

E para encontrar com mais facilidade o maior e menor no de uma determinada arvore, criamos duas funções que percorrerm a arvore e suas sub-árvores para encontrar o maior e menor elemento; sendo a "searchSmallest" para encontrar o menor elemento e a "searchLargest" para encontrar o maior elemento.

```c
struct NO *searchSmallest(struct NO *node){ //pesquisa o menor elemento (subarvore esquerda)
    if (node == NULL)
        return NULL;

    if (node->left == NULL)
        return node;

    return searchSmallest(node->left);
}
```

Figura 21- Implementação da função searchSmallest

```c
struct NO *searchLargest(struct NO *node){ //pesquisa o maior elemento (subarvore direita)
    if (node == NULL)
        return NULL;

    if (node->right == NULL)
        return node;

    return searchLargest(node->right);
}
```

Figura 22- Implementação da função searchLargest

4.1.3 Funções Auxiliares (geral)

```c
RbTree *createRbTree(){ // Função auxiliar para criar um novo nó
    RbTree *root = (RbTree *)malloc(sizeof(RbTree));
    if (root != NULL){
        *root = NULL;
    }
    return root;
}
```

Figura 23- Implementação da função createRbTree

```c
void freeNode(struct NO *no){ // Liberar o No
    if (no == NULL)
        return;
    freeNode(no->left);
    freeNode(no->right);
    free(no);
    no = NULL;
}
```

Figura 24 - Implementação da função freeNode

```c
void freeRbTree(RbTree *root){ // Liberar a árvore
    if (root == NULL)
        return;
    freeNode(*root);
    free(root);
}
```

Figura 25- Implementação da função freeRbTree

```c
struct NO *searchElement(struct NO *root, int current){
  // Encontrar Elemento - Busca
    if (root == NULL || root->info == current){
        return root; //retorna o valor encontrado ou a raiz (se for nula)
    }
    if (current < root->info)   { // Se o valor for menor que o valor
                //do nó atual, realiza a busca na subárvore esquerda
        return searchElement(root->left, current);
    }else{  // Se o valor for maior que o valor do nó atual,
            //realiza a busca na subárvore direita
        return searchElement(root->right, current);
    }
}
```

Figura 26 - Implementação da função searchElement

```c
int returnQuant(struct NO *root, int current, int qnt){
  // retorna a quantidade atual do produto;
    if (root == NULL || root->info == current){
        qnt = root->qtd_prod;
        // retorna a quantidade do produto atual se existir
        return qnt;
    }
    if (current < root->info){ // Se o valor for menor que o valor do nó
                               //atual, realiza a busca na subárvore esquerda
        return returnQuant(root->left, current, qnt);
    }else{ // Se o valor for maior que o valor do nó atual, realiza a busca
           //na subárvore direita
        return returnQuant(root->right, current, qnt);
    }
}
```

Figura 27- Implementação da função returnQuant

```c
int getColor(struct NO *node){ // pega a Cor do No
    if (node)
        return node->color;
    else
        return BLACK;
}
```

Figura 28- Implementação da função getColor

```c
// muda a quantidade do produto
void changeInfo(struct NO *root, int oldValue, int newValue){
  //verifica se o elemento existe
    struct NO *no = searchElement(root, oldValue);
    if (no == NULL){
        return;
    }
    no->qtd_prod = newValue;
}
```

Figura 29- Implementação da função changeInfo

4.1.4 Funções de Impressão

```c
void prinTree(RbTree *root){ // auxiliar da impressao da arvore rubro negra
    if (root == NULL){
        printf("Árvore vazia.\n");
        return;
    }
    printTreeHelper(root, 0);
}

void printTreeHelper(RbTree *root, int indentLevel){ // impressao arvore rubro negra
    if (root == NULL){
        printf("Árvore vazia.\n");
    }
    if (*root == NULL){
        return;
    }
    RbTree currentNode = *root;
    printTreeHelper(&(currentNode->right), indentLevel + 1);
    for (int i = 0; i < indentLevel; i++){
        printf("    ");
    }
    if (currentNode != NULL){
        printf("%d - %d\n", currentNode->info, getColor(currentNode));
        printTreeHelper(&(currentNode->left), indentLevel + 1);
    }
}
```

Figura 30- Implementação da função de impressão da árvore

```c
void printProd(RbTree *root){ // auxiliar na impressao do produto
    if (root == NULL){
        printf("Árvore vazia.\n");
        return;
    }
    printProdHelper(root, 0);
}

void printProdHelper(RbTree *root, int indentLevel){ // impressao produto
    if (root == NULL){
        printf("Árvore vazia.\n");
    }
    if (*root == NULL){
        return;
    }
    if (*root != NULL){
        printProdHelper(&((*root)->right), indentLevel + 1);
        for (int i = 0; i < indentLevel; i++)
            printf("    ");
        printf("%d (%d) - [ %s: %d ] \n", (*root)->info, (*root)->color, (*root)->name_prod, (*root)->qtd_prod);
        printProdHelper(&((*root)->left), indentLevel + 1);
    }
}
```

Figura 31- Implementação da função de impressão de um produto

```c
void printProdEstoqueAux(RbTree *root){ // auxiliar na impressao do produto em estoque
    if (root == NULL){
        printf("Árvore vazia.\n");
        return;
    }
    printProdEstoque(root, 0);
}

void printProdEstoque(RbTree *root, int indentLevel){ // impressao produto em estoque
    if (root == NULL){
        printf("Árvore vazia.\n");
    }
    if (*root == NULL){
        return;
    }
    if (*root != NULL){
        if ((*root)->qtd_prod != 0 && (*root)->qtd_prod > 0){
            printf("%s : %d \n", (*root)->name_prod, (*root)->qtd_prod);
        }
        printProdEstoque(&((*root)->right), indentLevel + 1);
        printProdEstoque(&((*root)->left), indentLevel + 1);
    }
}
```

Figura 32- Implementação da função de impressão de produtos disponíveis no estoque

```c
void printProdCastrados(RbTree *root){ // auxiliar na impressao do produtos cadastrados
    if (root == NULL){
        printf("Árvore vazia.\n");
        return;
    }
    printProdCastradosHelper(root, 0);
}

void printProdCastradosHelper(RbTree *root, int indentLevel){ // impressao produtos cadastrados
    if (root == NULL){
        printf("Árvore vazia.\n");
    }
    if (*root == NULL){
        return;
    }
    if (*root != NULL){
        printf("%d : %s  \n", (*root)->qtd_prod, (*root)->name_prod);
        printProdCastradosHelper(&((*root)->right), indentLevel + 1);
        printProdCastradosHelper(&((*root)->left), indentLevel + 1);
    }
}
```

Figura 33- Implementação da função de impressão dos produtos cadastrados

4.2 Arquivos

4.2.1 Rbt.h

```c
// ED2_TrabPratico_Grupo5_[IasminMarquesPereira][RafaelMendesMerlo]
#include <stdio.h>
#include <stdlib.h>

struct NO{ //estrutura do NO
    int info; //chave do produto = chave da arvore
    int color;//cor do no (red or black)
    struct NO *left; //no filho esquerdo
    struct NO *right; //no filho direito
    char* name_prod; //nome do produto
    int qtd_prod; //quantidade do produto
};

typedef struct NO *RbTree;

//funções utilizadas no codigo;
RbTree *createRbTree();
void freeNode(struct NO *no);
void freeRbTree(RbTree *root);
struct NO *searchElement(struct NO *root, int current);
int returnQuant(struct NO *root, int current, int qnt);
struct NO *rotateLeftRb(struct NO *root);
struct NO *rotateRightRb(struct NO *root);
int getColor(struct NO *node);
void changeColor(struct NO *node);
int insertRb(RbTree *root, int current, int qtd, const char *name, char **name_prod);
void changeInfo(struct NO *root, int oldValue, int newValue);
struct NO *dellBalanceNodes(struct NO *node);
struct NO *moveRedToLeft(struct NO *node);
struct NO *moveRedToRight(struct NO *node);
struct NO *searchSmallest(struct NO *node);
struct NO *searchLargest(struct NO *node);
struct NO *removeElementRb(struct NO *node, int current);
int removeRb(RbTree *root, int current);
void prinTree(RbTree *root);
void printTreeHelper(RbTree *root, int indentLevel);
void printProd(RbTree *root);
void printProdHelper(RbTree *root, int indentLevel);
void printProdEstoqueAux(RbTree *root);
void printProdEstoque(RbTree *root, int indentLevel);
void printProdCastrados(RbTree *root);
void printProdCastradosHelper(RbTree *root, int indentLevel);
```

Figura 34 - arquivo rbt.h

4.2.2 Main.c

```c
// ED2_TrabPratico_Grupo5_[IasminMarquesPereira][RafaelMendesMerlo]
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "RBT.c"

int main(){
    //variaveis de controle e armazenamento de informações para exibição
    int choice, novo; //controle do menu
    int qtd=0;//quantidade do produto
    int item =0; // controle de codigo do item a ser inserido
    char name[100]; // nome produto - string
    char *name_prod = NULL; //controle nome do produto - string
    RbTree *root = createRbTree();//arvore rubro negra
    int item_dois=0; // controle - atualização de quantidade de produto
    int quant_atual=0; // quantidade atual do produto

    do{
        printf("\n ---------------------------------------------- \n1. Cadastrar um novo produto\n");
        printf("2. Excluir um produto cadastrado\n");
        printf("3. Atualizar a quantidade de um produto no estoque\n");
        printf("4. Listar produtos cadastrados\n");
        printf("5. Listar produtos em estoque\n");
        printf("6. Imprimir a árvore Rubro-Negra\n");
        printf("0. Sair\n ---------------------------------------------- \n");
        printf("\nEscolha uma opção: ");
        scanf("%d", &choice);

        switch (choice){
        case 1: //inserir um produto
            item =0;
            printf("Digite o codigo produto a ser inserido: ");
            scanf("%d", &item);
            if (item<=0){
                printf("\n Codigo invalido \n");
            }else if (searchElement(*root, item)){ //codigo inserido já existe
                printf("\n Já possui um produto cadastrado com essa chave, tente outra chave \n");
            }else{
                fflush(stdin);
                printf("Digite o nome do produto (sem espaços): ");
                scanf(" %99[^\n]", name);
                printf("Digite a qtd do produto: ");
                scanf("%d", &qtd);
                if(item<0){
                    printf("\n - quantidade inválida - \n");
                }else{
                    item = insertRb(root, item, qtd, name, &name_prod);
                    printf("\n");
                    printProd(root);
                }
            }
            break;
        case 2: //remover um produto
            printf("Digite o produto a ser removido: ");
            scanf("%d", &item);
            if (searchElement(*root, item)){    //se o produto existir: remove
                removeRb(root, item);           //remove o produto
                prinTree(root);                 //printa a arvore
            }else{                              //produto nao existe
                printf("Produto não Existe! \n");
            }
            break;
        case 3: //atualizar quantidade do produto
            printf("Digite o codigo do produto a ser atualizado: ");
            scanf("%d", &item_dois);
            if (searchElement(*root, item_dois)) { //verifica se o produto existe
                quant_atual = returnQuant(*root,item_dois,quant_atual);
                printf("\n> O produto %d existe! Quantidade Atual: %d \n", item_dois, quant_atual);
                printf("\nInforme o novo valor da quantidade: ");
                scanf("%d", &novo);
                if(novo<0){ //se o valor da quantidade menor que zero, INVALIDO
                    printf("Numero inserido invalido;\n");
                }else{
                    changeInfo(*root, item_dois, novo);
                    printProd(root);
                }
            }else
                printf("O produto não existe!\n");
            break;
        case 4: //Exibe os produtos cadastrados

            if (*root == NULL){ // caso a árvore esteja vazia, não possui produtos cadastrados
                printf("\n [Sem produtos Cadastrados] \n");
            }else{ // se a arvore possuir produtos cadastrados
                printf("\n --- Produtos Cadastrados --- \n");
                printProdCastrados(root);
            }

            break;
        case 5://Exibe os produtos disponiveis em estoque
            if (*root == NULL){ // caso a árvore esteja vazia, não possui produtos cadastrados
                printf("\n [Sem produtos Cadastrados] \n");
            }else{ //caso não possua produtos: não tera informações, se possuir disponivel em estoque: exibira
                printf("\n --- Produtos Disponiveis no Estoque Atualmente --- \n");
                printProdEstoqueAux(root);
            }
            break;
        case 6: //impressao da arvore
            if (*root == NULL){  // caso a árvore esteja vazia
                printf("\n [Árvore Vazia] \n");
            }else{//exibe os elementos cadastrados na arvore
                prinTree(root);
            }
            break;
        case 0://encerra o programa
            printf("Encerrando...\n");
            break;
        default: //lida com as opções invalidas
            printf("Opção inválida! Tente novamente.\n");
        }
        printf("\n");
    } while (choice != 0);

    return 0;
}
```

Figura 35 - arquivo main.c

4.2.3 RBT.c

Figura 36 - arquivo rbtc.c

5. Resultados e Discussão

Fizemos alguns testes e algumas observações, quando usuário insere alguma String com espaço, na hora da impressão pode aparecer alguns caracteres aleatórios, mesmo tendo feito o tratamento de String no programa; também tivemos alguns problemas com o controle de Nós nulos, como por exemplo rotações com os nos nulos, rotações com Nós folhas, a solução que

encontramos para isso foi verificar a existência do nó antes de qualquer operação. Por exemplo:

```
(root!=NULL)

(root->left!=NULL)

(root->right!=NULL)

//exemplos de verificação da existencia desse no
```

Figura 37 - Exemplo de implementação de verificação

5.1 Exemplo 1 – Inserção Invalida

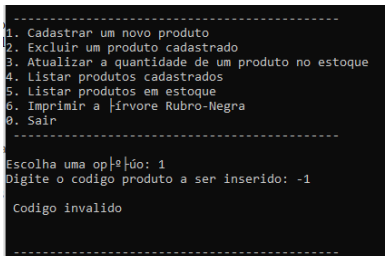


Figura 38 - Exemplo de inserção invalida

5.2 Exemplo 2 – Não há elementos na árvore

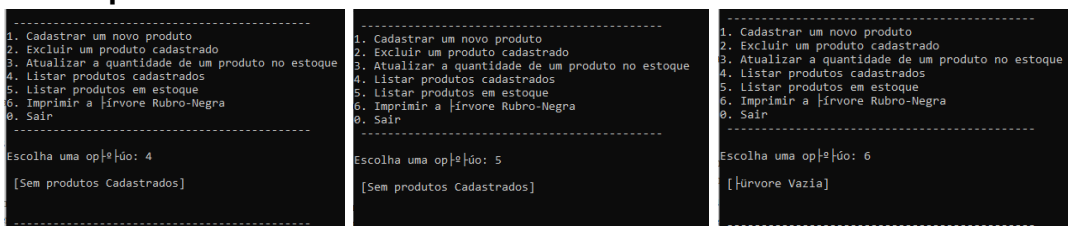


Figura 39 – Exemplo de Operações com uma Árvore Vazia

5.3 Exemplo 3 - Código digitado do produto invalido/ inexistente

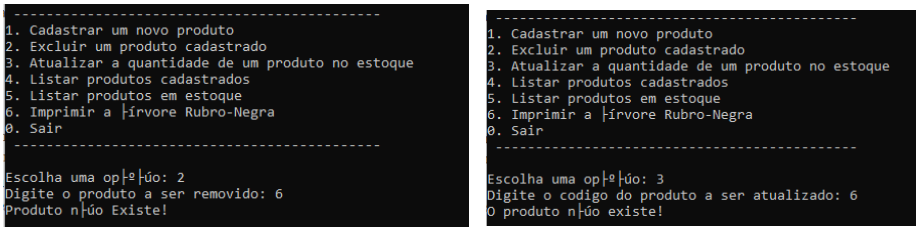


Figura 40 - Exemplo de código invalido

5.4 Exemplo 4 – Inserção válida

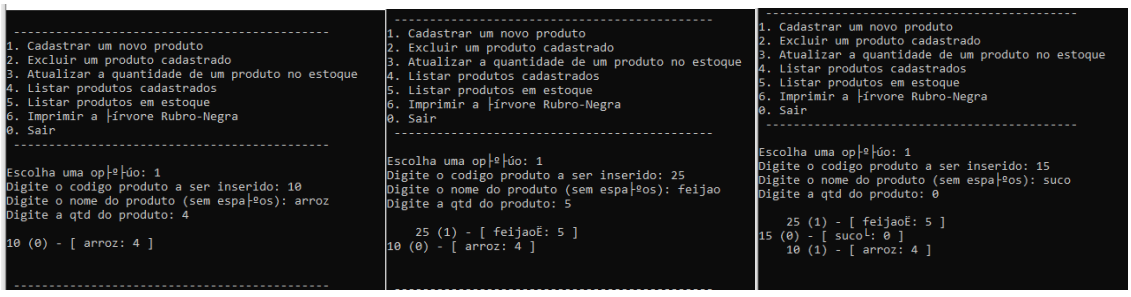


Figura 41 - Exemplo de Inserção válida

5.5 Exemplo 5 – Impressão produtos e produtos em estoque

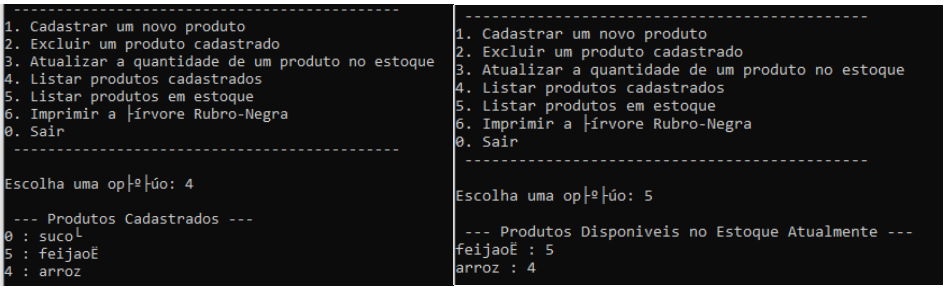


Figura 42 - Exemplo de Impressão de produtos cadastrados e em estoque

5.6 Exemplo 6 - Alterar quantidade de um produto

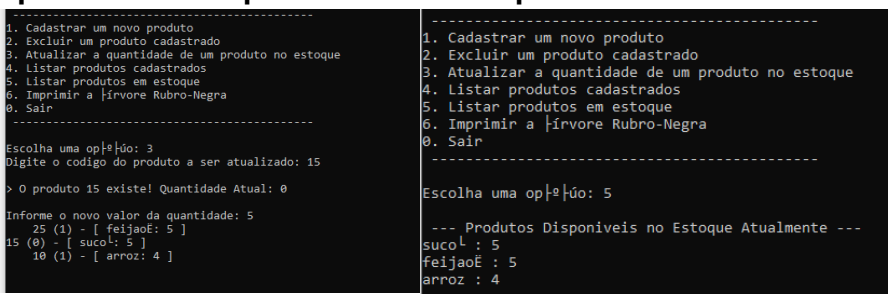


Figura 43 - Exemplo de alteração de quantidade de um produto

5.7 Exemplo 7 – Excluir produto

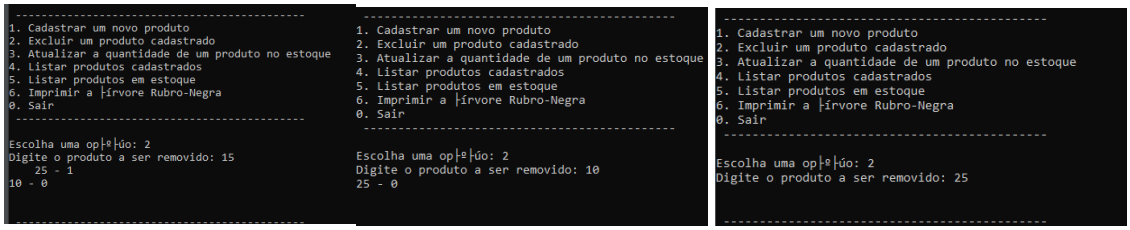


Figura 44 – Exemplo 1 de exclusão de produto

5.8 Exemplo 8 – Excluir produto (com mais de três produtos cadastrados)

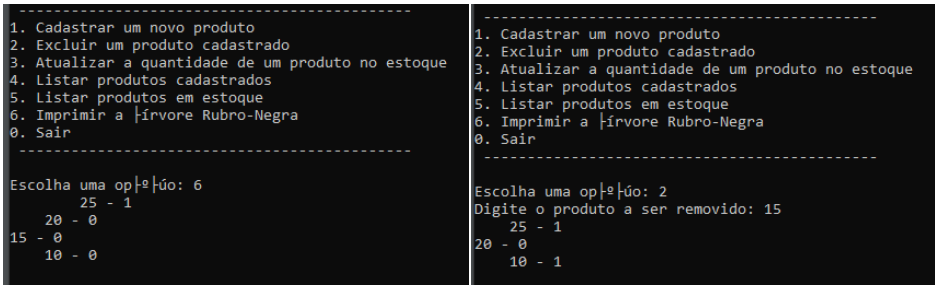


Figura 45 – Exemplo 2 de exclusão de produto

Apesar de não violar nenhuma das regras rubro negra, na exclusão de uma chave (exemplo, uma arvore com quatro chaves) o balanceamento correto de acordo com simulações que fizemos, a árvore deveria ficar com todos os nós pretos restantes, na arvore em que desenvolvemos a arvore retorna com a raiz

preta e os dois filhos vermelhos já pronta para outra inserção ou remoção de chave, não violando nenhuma das regras que torna a árvore rubro negra.

5.9 Exemplo 9 – Impressão de arvore rubro negra

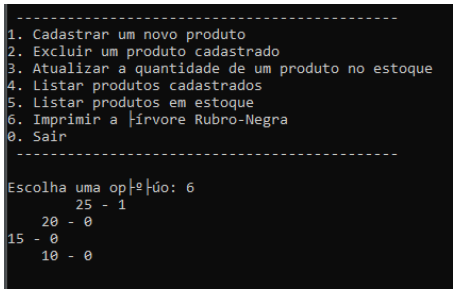


Figura 46 - Exemplo de impressão de árvore rubro negra

6. Conclusão

Concluiu-se que todos os objetivos foram alcançados com sucesso, o desenvolvimento das funções de cadastro de novo produto, exclusão de um produto cadastrado, atualização da quantidade de um produto no estoque e mostra-lo quando ele está disponível, listar todos os produtos cadastrados, listar todos os produtos disponíveis no estoque e a impressão da árvore Rubro-Negra, todas as funções foram implementadas sem a utilização de um ponteiro para o nó pai, nem para nenhum ancestral da árvore, sendo assim o desenvolvimento da arvore ficou todo recursivo, assim como a maioria das funções que auxiliam a construção da nossa árvore Rubro-Negra.

7. Referências Bibliográficas

https://www.inf.ufsc.br/~aldo.vw/estruturas/simulador/RB.html (Acessado em: 25/06/2023 às 20:50)

https://carbon.now.sh/ (Acessado em: 05/07/2023 às 21:00)

https://www.facom.ufu.br/~backes/gsi011/Aula12-ArvoreRB.pdf (Acessado em: 23/06/2023 às 22:10)

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-redblack.html#Node (Acessado em: 27/06/2023 às 15:57)

http://www.ulysseso.com/livros/ed2/ApF.pdf (Acessado em: 27/06/2023 às 16:45)

http://wiki.foz.ifpr.edu.br/wiki/index.php/Caracteres_e_String_em_C (Acessado em: 01/07/2023 às 19:23)