UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO - UFES
CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO - CEUNES
CIÊNCIA DA COMPUTAÇÃO

JOÃO PAULO SOUZA FERRETE
RAMON PEZZIN TON

RELATÓRIO

IMPLEMENTAÇÃO DE ÁRVORE RUBRO-NEGRA

SÃO MATEUS
27 DE ABRIL DE 2021

INTRODUÇÃO

Neste relatório temos como objetivo a implementação do tipo abstrato de dados (TAD) árvore rubro-negra, que é uma variação da árvore binária de busca. Desta forma, serão apresentadas as funções feitas e as lógicas utilizadas para a implementação e funcionamento do programa, bem como os conceitos aplicados e como funcionam cada função. A codificação foi feita na linguagem C, com a compilação feita a partir de um arquivo Makefile.

IMPLEMENTAÇÕES

Arquivo ‘rubro.h’:

O arquivo ‘*rubro.h*’ conta com a biblioteca das funções implementadas e todas as estruturas utilizadas para implementação da TAD. Este arquivo será incluído em todos os outros, para que seja possível a manipulação do tipo abstrato de dados em todo o programa.

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

typedef enum cor{
   preto,
   vermelho
}cor;

typedef enum boolean{
   false,
   true
}bool;

typedef struct rubro{
   void * obj;
   cor Cor;
   struct rubro * esq, *dir, *pai;
}rubro;

rubro * externo;

void apagaArvore(rubro * pt);
rubro * criarRubro();
rubro * buscaRubro(rubro * pt, void* obj, int (*compara)(void*,
void*));
void rubroIFixUp(rubro **pt, rubro* q);
void insereRubro(rubro** pt, void* obj, int (*compara)(void*, void*));
void rotacaoEsquerda(rubro ** pt, rubro *w);
void rotacaoDireita(rubro ** pt, rubro *w);
int alturaRubro (rubro * ptr);
void* getObj(rubro * pt);
cor getCor(rubro* pt);
void transferePaiRubro(rubro ** pt, rubro* u, rubro*v);
void rubroRFixUp(rubro **pt, rubro* x);
void removeRubro(rubro **pt, rubro *z);
```

Neste arquivo temos, primeiramente, a importação das bibliotecas a serem utilizadas na execução do programa, e então a definição das estruturas utilizadas. A primeira estrutura é

um enumerador, que contém os dois tipos de cores possíveis (preta e vermelha), e será utilizada para informar ao programa qual a cor do nó na árvore. Como segunda estrutura temos outro enumerador, contendo os elementos ‘true’ e ‘false’, que serão utilizados para facilitar o entendimento de verdadeiro e falso dentro do programa.

Como principal estrutura, temos uma *struct* que define o tipo ‘rubro’, representando um nó para uma árvore rubro-negra. Essa estrutura conta com um ponteiro para um tipo genérico de dado, onde irá guardar os elementos de informação da árvore, uma variável do tipo ‘cor’, que irá guardar a cor daquele nó na árvore, e também um ponteiro para para elementos do tipo rubro, que irá guardar os ponteiros para os filhos da direita e da esquerda, além de um ponteiro para o pai do nó em questão.

Ao fim da definição de estruturas, temos a declaração de uma variável global chamada ‘externo’ que irá guardar as informações dos ‘nós-folha’ da árvore. E, então, temos os protótipos de todas as funções que serão implementadas no arquivo ‘*rubro.c*’ e serão descritos a seguir.

Arquivo ‘rubro.c’:

Este arquivo é o que contém a implementação da TAD propriamente dita, contando com a codificação de todas as funções prototipadas no arquivo anterior. Este arquivo inclui ‘*rubro.h*’, para que consiga manipular os elementos definidos lá.

```
rubro * criarRubro(){

   rubro* pt = (rubro *) malloc(sizeof(rubro));
   if(!pt) return NULL;

   pt->dir = pt->esq = pt->pai = externo;
   pt->Cor = vermelho;

   return pt;
}
```

A primeira função, implementada neste arquivo, é a ‘*criaRubro()*’, que é responsável por criar um novo nó de árvore rubro-negra. Ela não recebe nenhum parâmetro, e retorna um ponteiro para um novo nó. Primeiramente, na sua execução, é alocado um espaço de memória para a árvore e então é verificado se o elemento foi alocado com sucesso. Em caso negativo a função para sua execução retornando *NULL*, mas, em caso positivo, a função continua a execução, fazendo com que todos os ponteiros para árvore apontem para o nó externo, já que

esse elemento será inserido como nó-folha. A cor, então, é definida como vermelha, já que todo novo nó inserido é rubro. Ao fim desse processo, o ponteiro para esse elemento é retornado.

```
void apagaArvore(rubro * pt){

   if(!pt || pt == externo) return;

   apagaArvore(pt->dir);
   apagaArvore(pt->esq);

   free(pt->obj);
   free(pt);
}
```

A função '*apagaArvore*' é responsável por desalocar todo o espaço utilizado por uma árvore. É do tipo *void,* e recebe por parâmetro um ponteiro para a raiz da árvore. Caso a raiz seja nula ou seja o elemento externo, então nada deverá ser feito, interrompendo a função com o return. Caso contrário, a função é chamada recursivamente para que sejam apagadas as subárvores direita e esquerda, e então é liberado o espaço utilizado pelo elemento de informação e pelo nó.

```
rubro * encontraMaiorRubro(rubro* pt){

   if(pt == externo) return externo;
   if(pt->dir == externo) return pt;
   return encontraMaiorRubro(pt->dir);
}
```

Essa função é responsável por encontrar o maior elemento de uma árvore. Ela é utilizada, na função de remoção, para encontrar o sucessor do nó a ser removido. Ela retorna um ponteiro para o maior elemento encontrado e recebe a raiz da árvore por parâmetro. Seu funcionamento é recursivo, e depende da característica das árvores binárias de busca, em que cada elemento maior que o elemento da raiz atual está na sua subárvore direita. Desta forma, primeiramente, verificamos se o nó atual é o nó externo. Em caso positivo a árvore não possui elementos e o próprio nó externo é retornado. Caso contrário, verificaremos se o nó da árvore possui filho direito. Caso não possua, então ele já é o maior elemento, retornando, assim, um ponteiro para ele. Mas caso ele possua filhos à direita, então chamamos a função, recursivamente, passando a analisar, agora, a subárvore direita.

```
int alturaRubro (rubro * ptr){

   int r, l;
   if(!ptr || ptr == externo) return 0;

   r=1+alturaRubro(ptr->dir);
   l=1+alturaRubro(ptr->esq);
   if(r>l) return r;
   return l;
}
```

Esta função é responsável por calcular a altura de uma árvore. Ela recebe por parâmetro um ponteiro para uma raiz e, inicialmente, verifica se esse ponteiro é válido. Em caso negativo, é retornado 0, visto que ela não possui altura, caso contrário a variável '*int*' 'r' recebe 1 (a altura de uma nó sozinho é 1) mais a altura da subárvore direita, que é calculada com uma chamada recursiva da função, e a variável 'l' recebe a mesma chamada, mas com a sub-árvore esquerda. Ao fim, corresponde à altura da árvore o maior valor, então caso 'r' seja maior que 'l', 'r' é retornado, caso contrário 'l' é retornado.

```
void* getObj(rubro * pt){

   if(!pt || pt == externo) return externo;
   return pt->obj;
}
```

A função *getObj()* é responsável por retornar um ponteiro para o elemento de informação do nó passado por parâmetro. Ela recebe um ponteiro para a raiz da árvore e verifica se é um ponteiro válido. Em caso positivo, é retornado um ponteiro para o elemento de informação de tipo genérico. E, em caso negativo, é retornado um ponteiro para o nó externo.

```
cor getCor(rubro* pt){

   return pt->Cor;
}
```

Essa função retorna a cor de um nó da árvore passado por parâmetro. Neste caso, não foi verificado se o nó passado por parâmetro é o *externo*, visto que o nó externo também possui uma cor.

```
rubro* buscaRubro(rubro * pt, void* obj, int (*compara)(void*, void*)){

   if (!pt || !obj || pt == externo) return NULL;

   if (compara(obj, pt->obj) == 0) return pt;
   if(compara(obj, pt->obj)<0) return buscaRubro(pt->dir, obj,
compara);
   return buscaRubro(pt->esq, obj, compara);
}
```

A função de busca em uma árvore rubro-negra funciona exatamente como uma função de busca em árvore *AVL*, visto que também é uma árvore binária de busca. Ela retorna um ponteiro para o nó encontrado e recebe por parâmetro um ponteiro para a raiz da árvore, um ponteiro para um elemento de informação que contém a chave de busca e um ponteiro para uma função de comparação.

Inicialmente, verifica-se os ponteiros recebidos por parâmetro. Caso alguma das condições seja satisfeita, é retornado *NULL*, pois significa, também, que o nó buscado não está na árvore. Em caso negativo, comparamos os objetos. Caso o objeto de informação do nó atual da árvore seja igual ao elemento passado por parâmetro, retornamos o ponteiro para este nó. Caso o objeto passado por parâmetro seja maior que o elemento do nó atual, chamamos a função recursivamente passando por parâmetro o ponteiro para a subárvore direita. Caso contrário, o elemento buscado é menor que o elemento da raiz atual. Então a função é chamada passando por parâmetro a subárvore esquerda.

```
void rotacaoEsquerda(rubro ** pt, rubro *w){

   if(!*pt || !w || *pt == externo || w == externo) return;

   rubro *v;

   v = w->dir;
   w->dir = v->esq;

   if(v->esq!=externo) v->esq->pai = w;
   if(v!=externo) v->pai = w->pai;
   if(w->pai == externo) *pt = v;
   else{
       if(w == w->pai->esq) w->pai->esq = v;
       else w->pai->dir = v;
   }
   v->esq = w;
   w->pai = v;
}
```

A função *'rotacaoEsquerda'* é responsável por efetuar a rotação para a esquerda a partir de um elemento *w*, que é passado por parâmetro, além de um ponteiro para a raiz da árvore. Após as verificações para realização da rotação, fazemos com que *v* aponte para o filho direito de *w*, e então o filho a direito de w passa a ser o filho a esquerda de v. Caso o filho esquerdo de v não seja o nó externo, seu pai passa a ser w. Caso *v* não seja o nó externo, então o pai de v passa a ser o pai de *w*. Caso o pai de *w* seja o nó externo, então ele é a raiz da árvore, por isso fazemos com que o ponteiro para a raiz da árvore aponte para v. Caso contrário, verificamos se w é filho direito ou esquerdo do seu pai, e fazemos com que ele passe a apontar para v. Assim, efetuamos a rotação propriamente dita, que é quando fazemos w ser filho à esquerda de *v*.

```
void rotacaoDireita(rubro ** pt, rubro *w){

   if(!*pt || !w || *pt == externo || w == externo) return;

   rubro *v;

   v = w->esq;
   w->esq = v->dir;
   if(v->dir!=externo) v->dir->pai = w;
   if(v!=externo) v->pai = w->pai;
   if(w->pai == externo) *pt = v;
   else{
       if(w == w->pai->dir) w->pai->dir = v;
       else w->pai->esq = v;
   }
   v->dir = w;
   w->pai = v;
}
```

A rotação à direita acontece de forma similar e simétrica à rotação à esquerda. Também recebe por parâmetro um ponteiro para a raiz da árvore e um ponteiro para o elemento que se deseja rotacionar. Mas, neste caso, o elemento *w* passa a ser o filho direito do elemento *v*, que era seu filho à esquerda.

```
void transferePaiRubro(rubro ** pt, rubro* u, rubro*v){

   if(!pt || *pt == externo ) return;

   if(u->pai == externo) *pt = v;
   else {
       if(u == u->pai->esq) u->pai->esq = v;
       else u->pai->dir = v;
   }
   v->pai = u->pai;
}
```

A função '*transferePaiRubro*' é responsável por transferir o elemento *v* para a posição do elemento *u*, transferindo, assim, seu respectivo pai. Além dos ponteiros *u* e *v*, ela recebe por parâmetro, também, um ponteiro para a raiz da árvore binária. Primeiro verificamos se a árvore é válida, e então verificamos se o pai de *u* é o nó externo. Neste caso, *u* é a raiz da árvore, e então fazemos com que a raiz da árvore seja *v*. Caso contrário, então verificamos se u é filho à esquerda ou à direita de seu pai, fazendo com que essa posição passe a apontar para v. E então o pai de *v* passa a ser o mesmo pai de *u*.

Operação de Inserção

A operação de inserção, na árvore rubro-negra, utiliza duas funções. A primeira função é a que insere um elemento, propriamente dito, e a segunda função é responsável por corrigir os possíveis erros e problemas causados por essa inserção nas propriedades de uma árvore rubro-negra. Será descrito, agora, em trechos de códigos, como foi implementada cada uma dessas funções.

Função '*insereRubro*'

```
void insereRubro(rubro** pt, void* elem, int (*compara)(void*, void*)){

   if(!elem) return;

   rubro *x, *v, *q;
   q = criarRubro();
   q->obj = elem;
   v = externo;
   x = *pt;
```

Esta função é responsável por inserir um novo elemento na árvore rubro-negra. Ela recebe por parâmetro um ponteiro para a raiz da árvore, um elemento de informação, que será inserido na árvore, e uma função de comparação entre elementos de informação de tipo

genérico de dados. No início da execução utilizamos a função *criarRubro* para alocar um novo elemento e fazemos com que a variável *q* receba um ponteiro para esse local alocado. E, então, atribuímos a *q* o elemento de informação que desejamos inserir na árvore.

```
while(x != externo){
       if(compara(elem, x->obj) == 0) return;
       v = x;
       if(compara(elem, x->obj) < 0) x = x->dir;
       else x = x->esq;
   }
```

Então, é feito um laço para descobrir o pai do elemento que será inserido. Caso um elemento da árvore seja igual ao elemento que será inserido, a função é encerrada, visto que a árvore não possui elementos iguais. Caso o elemento a ser inserido seja maior que o elemento do nó atual, fazemos com que o laço seja chamado novamente para a subárvore direita, caso contrário com a subárvore esquerda. Ao fim do laço, a variável '*v*' irá guardar um endereço de memória para o pai do nó que será inserido.

```
q->pai = v;
   if(v == externo) *pt = q;
   else{
       if(compara(elem, v->obj) < 0) v->dir = q;
       else v->esq = q;
   }
   rubroIFixUp(pt, q);
}
```

Após o laço, então, fazemos a inserção do elemento. O ponteiro para o pai do elemento a ser adicionado passa a apontar para v. Caso *v* seja o elemento externo, então é a inserção do primeiro elemento, então fazemos com que o ponteiro para a raiz da árvore aponte para *q*. Caso contrário, verificamos se o elemento de *v* é maior ou menor que o elemento inserido. Caso seja maior, o elemento passa a ser filho à direita de v, e caso seja menor ele passa a ser filho à esquerda de v. Após o fim da inserção, é chamada a função *rubroIFixUp* que irá corrigir possíveis violações nas propriedades da rubro-negra.

Função '*rubroIFixUp*'

```
void rubroIFixUp(rubro **pt, rubro* q){

    if(!*pt || !q || *pt == externo || q == externo) return;

    rubro *pai, *avo, *t;
    while(q->pai->Cor == vermelho){
        pai = q->pai;
        avo = q->pai->pai;
```

Essa função, como dito anteriormente, é responsável por fazer com que a árvore mantenha as características de uma árvore rubro-negra após a inserção de um novo nó. Ela recebe por parâmetro um ponteiro para o ponteiro da raiz da árvore e um ponteiro para o novo elemento inserido. Inicialmente, a função possui um laço que ocorrerá enquanto o pai do último elemento inserido seja vermelho, pois, já que cada novo nó inserido é vermelho, viola a regra de que um nó vermelho não pode possuir pai vermelho. Assim, dentro do laço, fazemos com que o ponteiro *pai* aponte para o pai de *q*, e o ponteiro *avo* aponte para o pai do pai de *q*.

```
        if(pai == avo->esq){
            t = avo->dir;
            if(t->Cor == vermelho){
                pai->Cor = t->Cor = preto;
                avo->Cor = vermelho;
                q = avo;
            }
```

Então, já que o problema após a inserção está no pai de *q*, verificamos se ele é filho à esquerda de seu pai. Em caso afirmativo, fazemos com que o ponteiro *t* aponte para o irmão de *pai*, e verificamos se a cor de *t* é vermelho, já que nesse caso a violação de regras pode ser resolvida com a mudança de cores de *pai* e de seu irmão *t* para a cor preta, e a cor de *avo* para vermelho. Assim, como *q* é o novo nó vermelho, passamos a verificar ele como novo nó rubro.

```
        else{
            if(q == pai->dir){
                q = pai;
                rotacaoEsquerda(pt, q);
            }
            q->pai->Cor = preto;
            avo->Cor = vermelho;
            rotacaoDireita(pt, avo);
        }
```

Já, caso o tio de *q* não seja vermelho também, temos outra violação, fazendo-se necessário uma rotação. Caso q seja filho à direita de seu pai, é necessário fazer uma rotação dupla, fazendo com que o nó *q* seja o pai e então fazendo uma rotação à esquerda, antes de continuar a execução.

Continuando a execução do código, pintamos a cor do pai de *q* para preto, e a cor de seu avô para vermelho, resolvendo assim a violação de cores, e então é feita uma rotação à direita.

```
        }else{
               t = avo->esq;
               if(t->Cor == vermelho){
                   pai->Cor = t->Cor = preto;
                   avo->Cor = vermelho;
                   q = avo;
               }
               else{
                   if(q == pai->esq){
                       q = pai;
                       rotacaoDireita(pt, q);
                   }
                   q->pai->Cor = preto;
                   avo->Cor = vermelho;
                   rotacaoEsquerda(pt, avo);
               }
           }
       }
       (*pt)->Cor = preto;
   }
```

Já, caso o pai do nó inserido seja filho à direita do pai que *q*, o procedimento é feito de maneira similar à anterior, porém com o processo espelhado, trocando os lados direitos pelos esquerdos e vice-versa. Ao fim da execução do laço, fazemos com que a cor da raiz da árvore continue preta, para corrigir qualquer equívoco.

Operação de Remoção

A operação de remoção, assim como a de inserção, utiliza duas funções. A primeira é a função que remove o elemento, propriamente dita, e a segunda é a função que garante a continuidade das regras e propriedades de uma árvore rubro-negra.

Função '*removeRubro*'

```
void removeRubro(rubro **pt, rubro *z){
   rubro *x, *y;
   cor yColor;

   y = z;
   yColor = y->Cor;
```

Essa função é responsável por remover um elemento da árvore rubro-negra, e recebe por parâmetro um ponteiro para a raiz da árvore e um ponteiro para o elemento que deverá ser removido. Ao se iniciar a execução da função, *y* também passa a apontar para *z,* que é o elemento que será removido e *yColor*, que representa a cor do nó que será removido, guarda a cor de *y*.

```
   if(z->esq == externo){
       x = z->dir;
       transferePaiRubro(pt, z, z->dir);
       free(z->obj);
       free(z);
   }
```

Caso o elemento a ser removido não possua filho à esquerda, então fazemos com que *x* aponte para o filho a direita que *z,* que pode existir, e então fazemos a transferência do pai de *z* para seu filho à direita *x*,que tomará o lugar de *z*. Após isso liberamos todo o espaço utilizado pelo nó z.

```
else if (z->dir == externo){
       x = z->esq;
       transferePaiRubro(pt, z, z->esq);
       free(z->obj);
       free(z);
   }
```

Caso o elemento não possua filho à direita, o procedimento é exatamente o mesmo para caso não possua filho à esquerda, com exceção de que *x*, agora, aponta para o filho à esquerda de *z*.

```
else{
       y = encontraMaiorRubro(z->esq);
       yColor=y->Cor;
       x = y->esq;
       if(y->pai == z) x->pai = y;
       else{
           transferePaiRubro(pt, y, x);
           y->esq = z->esq;
           y->esq->pai = y;
       }
```

Já, caso *z* possua os dois filhos, então precisamos encontrar o elemento que irá ser o seu sucessor. Neste caso, o sucessor de *z* é o maior elemento da subárvore esquerda. Após a determinação de *y,* fazemos com que *yColor* receba a cor de *y*, já que ele será o elemento que sairá de sua posição, para tomar a posição de *z.* Desta forma, x passa a apontar para o filho à esquerda de *y*, visto que, se *y* é o maior elemento da subárvore esquerda de *z*, então ele não possui filhos à direita.

Assim, para a atribuição dos novos pais de *x*, caso *y* seja filho direto de z, então x continua como filho de y. Mas, caso contrário, então é feita a transferência do pai de y para x, e fazemos com que *y* receba o filho à esquerda de *z*.

```
        transferePaiRubro(pt, z, y);
        y->dir = z->dir;
        y->dir->pai = y;
        y->Cor = z->Cor;
        free(z->obj);
        free(z);
    }
```

Após o tratamento para os filhos de *y*, fazemos com que ele passe a substituir z, fazendo a transferência do pai, dos filhos à direita e da cor de *z* para *y*. Após essas transferências é liberado o espaço utilizado pelo nó *z*.

```
    if(yColor == preto) rubroRFixUp(pt, x);

    (*pt)->Cor = preto;
    externo->pai=externo->dir=externo->esq=externo;
}
```

Ao fim da remoção, verificamos se a cor do elemento movido ou removido é preta. Em caso afirmativo é chamada a função de correção, visto que a altura de nós negros foi alterada na árvore. Então, garantimos q o nó raiz da árvore continue negro e que todos os ponteiros do nó externo apontem para o nó externo.

Função ‘rubroRFixUP’

```
void rubroRFixUp(rubro **pt, rubro* x){

    if(!pt || *pt == externo) return;
    rubro *w;

    while(x!=*pt && x->Cor == preto){
```

Essa função é a responsável pela correção das características da rubro-negra e recebe um ponteiro para a raiz da árvore e um ponteiro para o sucessor do nó removido. A função

consiste, basicamente, em um laço, que irá acontecer enquanto x for diferente da raiz da árvore e a cor de x for preto, indicando assim que há violação.

```
if(x == x->pai->esq){
    w = x->pai->dir;
    if(w->Cor == vermelho){
        w->Cor = preto;
        x->pai->Cor = vermelho;
        rotacaoEsquerda(pt, x->pai);
        w = x->pai->dir;
    }
```

Em cada iteração do laço, primeiramente, verificamos se x é filho à esquerda de seu pai. Em caso positivo, é feito com que *w* passe a apontar para o irmão de *x*. Caso a cor *w* seja vermelho, então temos o primeiro caso de violação. Para resolvê-lo, fazemos com que a cor de w, assim como x, seja preto e a de seu pai seja vermelho, e então fazemos uma rotação à esquerda, com foco no seu pai. Assim, w passa a apontar ao novo irmão de x após a rotação.

```
    if(w->esq->Cor == preto && w->dir->Cor == preto){
        w->Cor = vermelho;
        x = x->pai;
    }
```

Caso, mesmo após a rotação, os dois filhos de *w* sejam de cor negra, então a cor de *w* passa a ser vermelho e a referência de *x* passa a ser seu pai, já que as inconsistências no nível atual já foram consertadas.

```
    else{

        if (w->dir->Cor == preto){
            w->esq->Cor = preto;
            w->Cor = vermelho;
            rotacaoDireita(pt, w);
            w = x->pai->dir;
        }
        w->Cor = x->pai->Cor;
        x->pai->Cor = preto;
        w->dir->Cor = preto;
        rotacaoEsquerda(pt, x->pai);
        x = *pt;
    }
}
```

Caso contrário, ou seja, se pelo menos um dos filhos de *w* for vermelho, então verificamos se o filho direito é preto. Em caso positivo, então o filho à esquerda de *w* também

passa a ser preto, e *w* passa a ser vermelho. É feita uma rotação à direita e *w* passa a apontar para o novo irmão de *x* após a rotação.

Após a verificação, fazemos com que w receba a cor de seu pai, e a cor de seu pai e de seu filho à direita passa a ser preto. Fazemos, então, uma rotação à esquerda e fazemos *x* receber um ponteiro para a raiz da árvore, para interromper o laço, já que, neste ponto, todos os problemas já foram solucionados.

```
        else{
            w = x->pai->esq;
            if(w->Cor == vermelho){
                w->Cor = preto;
                x->pai->Cor = vermelho;
                rotacaoDireita(pt, x->pai);
                w = x->pai->esq;
            }
            if(w->dir->Cor == preto && w->esq->Cor == preto){
                w->Cor = vermelho;
                x = x->pai;
            }
            else{
                if (w->esq->Cor == preto){
                    w->dir->Cor = preto;
                    w->Cor = vermelho;
                    rotacaoEsquerda(pt, w);
                    w = x->pai->esq;
                }
                w->Cor = x->pai->Cor;
                x->pai->Cor = preto;
                w->esq->Cor = preto;
                rotacaoDireita(pt, x->pai);
                x = *pt;
            }
        }

    }
    x->Cor = preto;
```

Já, caso *x* seja filho à direita de seu pai, então o procedimento é feito de maneira similar ao anterior, porém trocando os lados esquerdo e direito. Ao fim do laço fazemos com que a cor de x, que aponta para a raiz da árvore, seja preta.

Arquivo '*main.c*'

O arquivo *main.c* é o arquivo que contém o programa cliente, que usará a árvore rubro-negra. Ela inclui os arquivos anteriores, e define a estrutura do elemento de informação, bem como a função de impressão e a função de comparação entre esses elementos.

```
typedef struct artigo{
   int id;
   int ano;
   char autor[200];
   char titulo[200];
   char revista[200];
   char DOI[200];
   char palavraChave[200];
}Artigo;
```

Primeiramente temos a definição da estrutura do elemento de informação da árvore, que corresponde à informações de um artigo, contendo um número de identificação (*id*), que será utilizado como chave na árvore, o ano de publicação, o nome do ator, o título do artigo, o nome da revista em que foi publicado, a DOI do artigo e as palavras-chave dele.

```
int comparaArt(void* ob1, void* ob2){

   int a = ((Artigo*)ob1)->id;
   int b = ((Artigo*)ob2)->id;

   if(a>b) return -1;
   if(a<b) return 1;
   return 0;

}
```

A função *comparaArt* é a função passada por parâmetro para as funções de inclusão e de busca em uma árvore rubro-negra, e ela é responsável por comparar dois artigos passados por parâmetro. Primeiramente fazemos com que as váriáveis *a* e *b* receba o *id* dos objetos 1 e 2, respectivamente. Caso o *id* a seja maior que o *b*, retornamos -1. Caso o *id b* seja maior que o *a*, retornamos 1. Já caso eles sejam iguais, retornamos 0.

```
Artigo * insereArtigo(){

   Artigo *a = (Artigo *) malloc(sizeof(Artigo));

   printf("Insira o ID do artigo: ");
   scanf("%d", &a->id);
   printf("Insira o Título do artigo: ");
   scanf("%s", a->titulo);
   printf("Insira o nome do autor: ");
   scanf("%s", a->autor);
   printf("Insira o ano de publicação: ");
   scanf("%d", &a->ano);
   printf("Insira o nome da revista: ");
   scanf("%s", a->revista);
   printf("Insira o DOI: ");
   scanf("%s", a->DOI);
   printf("Insira as palavras-chave: ");
   scanf("%s", a->palavraChave);

   return a;
}
```

Essa função é responsável por criar um novo elemento de informação do tipo Artigo. Primeiramente é alocado o espaço de memória necessário, e então solicita ao usuário que insira as informações do artigo. Ao fim da inserção a função retorna um ponteiro para o elemento criado.

```
Artigo * pesquisaArt(){

   Artigo *a = (Artigo *) malloc(sizeof(Artigo));

   printf("Insira o ID do artigo: ");
   scanf("%d", &a->id);

   return a;
}
```

De forma similar à função *insereArtigo*, a função *pesquisaArt* é responsável por criar um elemento de informação, mas, desta vez, para busca. O espaço é alocado e o usuário deve inserir apenas o ID do artigo, para que possa ser feita a busca. Ao fim da inserção, a função retorna um ponteiro para o elemento alocado.

```
void imprimeArtigo(void *obj){

   Artigo *a = (Artigo *) obj;
   printf("ID: %d\n", a->id);
   printf("Titulo: %s\n", a->titulo);
   printf("Autor: %s\n", a->autor);
   printf("Ano: %d\n", a->ano);
   printf("Revista: %s\n", a->revista);
   printf("DOI: %s\n", a->DOI);
   printf("Palavra-chave: %s\n", a->palavraChave);
}
```

Já, a função *imprimeArtigo* é responsável por imprimir todos os atributos de um elemento de tipo genérico passado por parâmetro. Primeiramente, é feito um casting para o tipo arquivo, e então são impressos todos os atributos daquele artigo.

```
void imprimeArvore(rubro * ptr){

   if(!ptr || ptr == externo) return;
   int i, nivel = alturaRubro(ptr);

   if(ptr){
       imprimeArvore(ptr->esq);
       for(i=0; i<nivel; i++) printf("\t");
       Artigo *a = (Artigo *) getObj(ptr);
       if(getCor(ptr) == vermelho) printf("[%d : rubro]\n\n", a->id);
       else printf("[%d : negro]\n\n", a->id);
       imprimeArvore(ptr->dir);
   }
}
```

A função de impressão de árvore é responsável por imprimir toda a estrutura da árvore e recebe por parâmetro um ponteiro para a raiz da mesma. A impressão é feita a partir do número de tabulações correspondentes ao nível. Primeiramente, a variável '*nivel*' recebe a altura da árvore. Então, chamamos a função para que, primeiramente, seja impresso a subárvore esquerda, para que a impressão seja simétrica, e então imprimimos, de acordo com o nível da árvore, a tabulação, para facilitar a visualização, então é feito um casting do elemento de informação genérico para o tipo Artigo. Então, caso a cor do nó atual seja vermelho, imprimimos o *id* dele após a cor. Já, caso seja preto, imprimimos o *id* seguido da palavra 'negro'. Então, se chama a função de impressão para a subárvore direita.

Função '*main*'

Essa é a função principal do programa, responsável pelo gerenciamento de toda a estrutura de dados, inclusão, exclusão, busca e impressão, do sistema.

```
int main(){

   externo = (rubro *) malloc(sizeof(rubro));
   externo->Cor = preto;
   externo->pai = externo->dir = externo->esq = externo;

   bool ba=true;

   int op;
   rubro *aux, *pt = externo;
   Artigo * art;
   char a;
```

No início da função, alocamos o espaço de memória para o nó externo, e aplicamos suas características, onde sua cor passa a ser preta e todos os seus ponteiros passam a apontar para ele mesmo. E criamos a variável do tipo bool *'ba'* que é responsável pela continuidade da execução do laço.

```
while(ba){
   printf("\n\n--------MENU--------\n");
   printf("1 - Inserir artigo\n");
   printf("2 - Remover artigo\n");
   printf("3 - Pesquisar artigo\n");
   printf("4 - Imprimir Árvore\n");
   printf("0 - SAIR\n");
   printf("--------------------\n");
   scanf("%d", &op);
```

Então, enquanto '*ba'* for verdadeiro, o faço continuará executando. No início do laço é mostrado o menu de opções ao usuário, onde ele deverá escolher uma opção.

```
switch (op){
   case 1:
       art = insereArtigo();
       insereRubro(&pt, art, &comparaArt);
       printf("\n\nELEMENTO INSERIDO!\n");
       break;
```

Caso o usuário insira a opção 1, então um novo elemento deverá ser inserido na árvore. Então é criado um novo elemento de informação, e chamamos a função '*insereRubro*' para inserir esse elemento na árvore. Após a inserção uma mensagem de 'Elemento Inserido' é mostrada e o programa volta ao menu.

```
case 2:

    art = pesquisaArt();
    aux = buscaRubro(pt, art, &comparaArt);

    if(!aux) printf("\nElemento não encontrado!\n");
    else{
        printf("\n\nElemento Encontrado: \n");
        imprimeArtigo(getObj(aux));

        printf("Deseja apaga-lo? (s/n):  ");
        getchar();
        scanf("%c", &a);
        if(a=='s' || a=='S'){
            removeRubro(&pt, aux);
            printf("\n\nElemento apagado!\n");
        }
    }
    free(art);
    break;
```

Caso o usuário escolha a opção 2, então um elemento deverá ser deletado. Então, é criado um elemento de informação contendo apenas o *ID* para que seja feita a busca. Então a busca é feita a partir do *id* inserido, e o retorno é guardado na variável *aux*. Caso *aux* seja igual a *NULL*, então o elemento não existe na árvore, imprimindo uma mensagem de que o elemento não foi encontrado.

Mas, caso *aux* seja diferente de *NULL*, então o elemento foi encontrado. Então é impresso ao usuário todas as informações sobre o artigo encontrado e solicita a ele a confirmação para que o elemento seja apagado. Caso o usuário confirme, então é chamada a função *removeRubro*, passando o elemento encontrado por parâmetro. Então é impresso uma mensagem de '*Elemento apagado!*'. Ao fim é liberado o espaço utilizado pelo elemento de informação para busca e então o sistema volta ao menu.

```
case 3:
    art = pesquisaArt();

    aux = buscaRubro(pt, art, &comparaArt);
    if(!aux) printf("Elemento não encontrado!\n");
    else{

        printf("\n\nElemento Encontrado: \n");
        imprimeArtigo(getObj(aux));/
    }
    free(art);
    break;
```

Caso o usuário insira a opção 3, então um elemento deverá ser buscado. Desta forma, criamos um elemento de informação contendo apenas o *id*, para que seja buscado. O retorno da função de busca é guardado na variável *aux*. Caso ela seja nula, então é impresso que o elemento não foi encontrado. Já, caso ela seja diferente de nulo, todos os seus atributos são impressos. É liberado o espaço utilizado pelo elemento de busca e a função volta ao menu.

```
            case 4:

                printf("Imprimindo árvore: \n\n");
                imprimeArvore(pt);
                break;
```

Já, se o usuário inserir a opção de impressão estrutura de árvore, o sistema exibe uma mensagem de impressão e chama a função *imprimeArvore*, passando por parâmetro o ponteiro para a raiz da árvore.

```
            case 0:

                apagaArvore(pt);
                free(externo);
                ba=false;
        }
```

Por último, caso o usuário insira a opção de finalizar o programa, a função '*apagaArvore'* é chamada, para liberar todo o espaço utilizado pela árvore. Então liberamos o nó externo e atribuímos *false* a variável *ba* para que a execução do laço seja interrompida, encerrando, assim, o programa.

CONCLUSÃO

Neste relatório foi possível descrever todas as funções implementadas, aplicando os conceitos aprendidos em aula e exercitando os conhecimentos adquiridos anteriormente, o que ocasionou uma melhora nas práticas de programação e de entendimento de códigos.