**UFES / CEUNES**

**DEPARTAMENTO COMPUTAÇÃO E ELETRÔNICA**

**CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

CHRISTIAN JONAS OLIVEIRA

JOÃO VICTOR DO ROZÁRIO RECLA

**IMPLEMENTAÇÃO DA ÁRVORE ARN:**

**ESTRUTURA ARTIGO**

SÃO MATEUS – ES

2021

Sumário

1. Definição da estrutura ARN e Artigo 3

2. Inserção 3

2.1 Insert\_Fixup 4

3. Rotações 8

3.1 Rotação para a esquerda 8

3.2 Rotação para a direita 9

4. Busca 10

5. Remoção 10

5.1 Remove\_Fixup 12

5.2 Algoritmos auxiliares da remoção 15

6. Auxiliares Gerais 16

6.1 Cria Sentinela 16

6.2 Cria ARN 16

6.3 Imprime ARN 17

6.4 Destrói ARN 17

6.5 Compara id de Artigos 18

7. Cliente (Main) 18

7.1 Coleta e inserção de dados de um Artigo. 18

7.2 Imprime dados 19

7.3 Imprime Menu 20

Neste trabalho, o objetivo era implementar a estrutura ARN (Árvore Rubro Negra) e suas funções para uma estrutura que descreve o cadastro de um artigo na TAD. Listaremos a seguir o TAD e todas as suas funções e as devidas explicações necessárias para a compreensão daquilo que foi programado.

1. Definição da estrutura ARN e Artigo

**typedef** **struct** ARN{

artigo \*key;

**struct** ARN \*f;

**struct** ARN \*left;

**struct** ARN \*right;

**int** color;

}ARN;

**typedef** **struct** artigo{

**int** id;

**int** ano;

**char** autor[**200**];

**char** titulo[**200**];

**char** revista[**200**];

**char** DOI[**20**];

**char** palavraChave[**200**];

}artigo;

1. Inserção

**int insert\_ARN(ARN \*\*root, ARN \*\*node);**

A inserção recebe o endereço de um ponteiro do Nó-raiz da árvore e o endereço de um ponteiro para o elemento a ser inserido.

**while**(aux1 != Sentinela){

// verifica se o node já existe;

**if**(compare\_Artigo(aux1->key, (\*node)->key) == **0**) **return** **0**;

aux2 = aux1;

// buscando a posição no qual o novo no devera ser inserido;

**if**(compare\_Artigo((\*node)->key, aux1->key) < **0**)

aux1 = aux1->left;

**else** aux1 = aux1->right;

}

Primeiramente há uma sequência de operações que “moverá” o ponteiro auxiliar (aux1) para o local onde o novo nó deverá ser inserido. A cada iteração será verificado se o elemento já existe na ARN, se ele é maior que o no atual ou se é menor que o no atual.

**if**(aux2==Sentinela) (\*root) = (\*node);

**else**{

// caso seja filho a esquerda do no pai, node será o novo filho a esquerda;

**if**(compare\_Artigo((\*node)->key, aux2->key) < **0**)

aux2->left = (\*node);

// caso seja filho a direita do no pai, node será o novo filho a direita;

**else** aux2->right = (\*node);

}

// valores default de um novo no;

(\*node)->left = Sentinela;

(\*node)->right = Sentinela;

(\*node)->color = **1**;

insert\_Fixup(root, node);

Após encontrar o local onde o elemento deverá ser inserido, seu novo pai é nomeado. Em sequência verifica-se o caso do mesmo ser o primeiro elemento. Se não for o caso, trata-se a hereditariedade do novo elemento, dele ser filho a esquerda ou filho a direita do Nó-pai. Ao final, realiza-se o apontamento dos filhos do novo no para o Nó-Sentinela e logo após, ocorre a devida reestruturação e balanceamento da árvore e de suas cores através do algoritmo “*insert\_Fixup*”.

* 1. Insert\_Fixup

**void insert\_Fixup(ARN \*\*root, ARN \*\*node);**

Neste algoritmo será detalhado sobre as rotações e as mudanças de cor aplicadas na ARN. O algoritmo se inicia com um loop, afim de reorganizar a árvore a partir do Nó-crítico até a raiz. Ele será executado enquanto a cor do Nó-pai em questão for rubra.

**while**((\*node)->f->color==**1**){

aux1 = (\*node)->f;

aux2 = aux1->f;

// aux1 é filho a esquerda do seu pai?

**if**(aux1 == aux2->left){

aux3 = aux2->right;

Primeiramente, verifica-se se o pai do Nó-crítico é filho a esquerda de seu pai, afim de tratar os casos gerados no lado esquerdo da árvore. Em caso afirmativo, três situações deverão ser analisadas, todas elas relacionadas a uma propriedade, já que as outras não serão violadas ou serão facilmente resolvidas.

Nessas situações, há uma violação na propriedade 4, que afirma que não existem nós rubros consecutivos.

**Situação 1**

// caso em que o tio do Nó a ser inserido é rubro.

**if**(aux3->color==**1**){

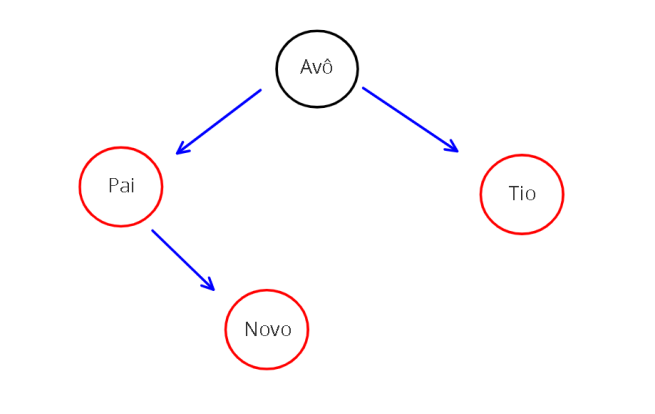
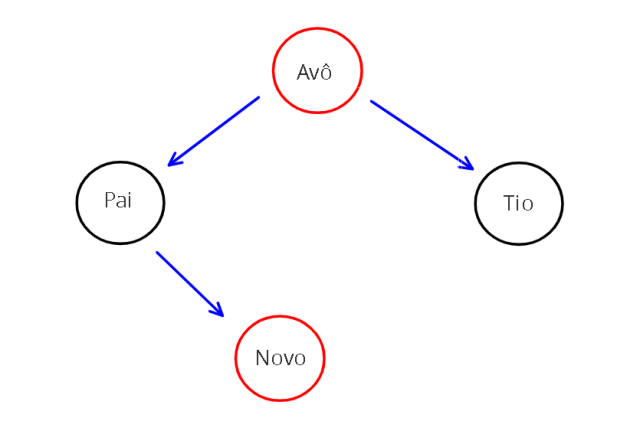
aux1->color = **0**;

aux3->color = **0**;

aux2->color = **1**;

(\*node) = aux2;

}

Para solucionar esse problema, o pai e o tio do Nó inserido recebem a cor negra enquanto seu avô recebe a cor rubra.

*Após a mudança*

*Antes da mudança*

**Situação 2**

**else**{

// caso onde tio do Nó a ser inserido é negro e o novo Nó é filho a direita de seu pai;

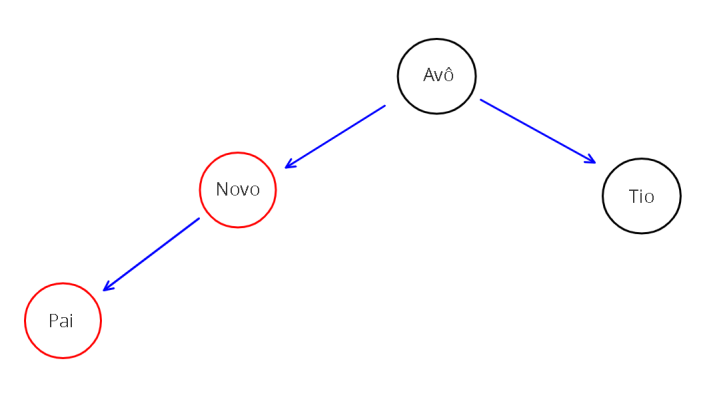
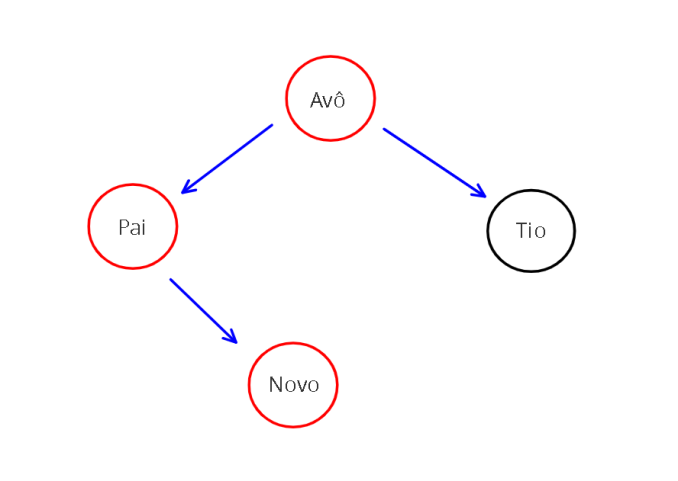
**if**((\*node) == aux1->right){

(\*node) = aux1;

left\_Rotation(root, node);

}

Neste caso, deverá ser realizado um reapontamento do Nó inserido seguido de uma rotação para a esquerda e outra para a direita. Basicamente, transformaremos a situação 2 de maneira que ela possa ser resolvida com o pedaço de código da próxima situação.



*Antes da mudança*

*Após a mudança (Caso 3)*

**Situação 3**

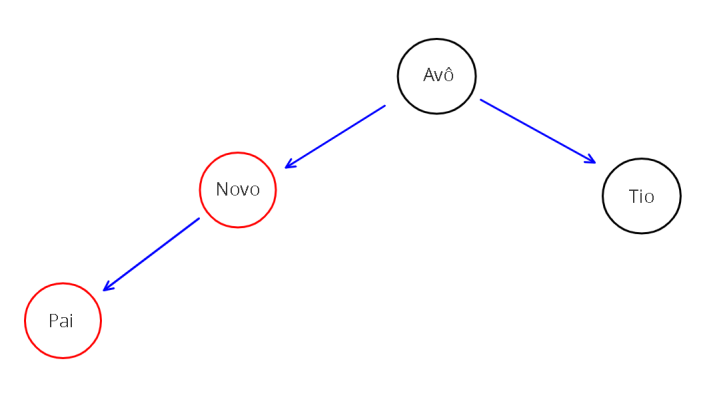
// caso onde tio do Nó a ser inserido é negro e o novo Nó é filho a esquerda de seu pai;

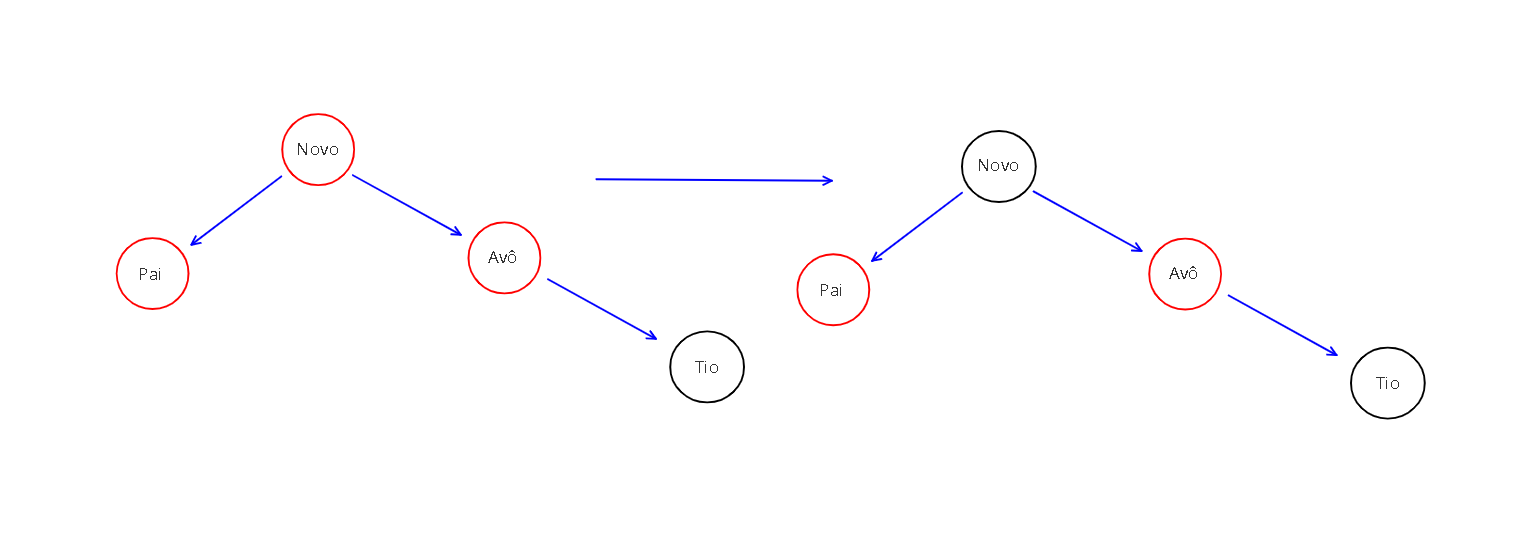
(\*node)->f->color = **0**;

aux2->color = **1**;

right\_Rotation(root, &aux2);

Esta situação além de ser um complemento da situação 2, corrige também o caso oposto ao caso 2, que é onde o filho foi inserido, ao invés de ser filho a direita, é filho a esquerda de seu pai. Daí, realiza-se a troca de cor do pai do nó a ser inserido, e do avô do nó a ser inserido, seguido de uma rotação para a direita. Ao final do algoritmo troca-se a cor da raiz para negra.



Vale ressaltar que há também uma sequência do código em questão, que tratará o caso em que *aux1* é filho a direita de seu pai. Neste caso, as rotações ocorrem de maneira semelhante, porém espelhada.

*Troca da cor da raiz*

*Após da rotação*

*Antes da rotação*

1. Rotações  
   1. Rotação para a esquerda

**void left\_Rotation(ARN \*\*root, ARN \*\*node);**

A rotação para a esquerda, definida pela função *left\_Rotation*, funcionará da seguinte forma:

ARN \*aux;

aux = (\*node)->right;

(\*node)->right = aux->left;

**if**(aux->left!=Sentinela){

aux->left->f = \*node;

}

aux->f = (\*node)->f;

**if**((\*node)->f==Sentinela) \*root = aux;

**else**{

// caso o no que sofreu rotação seja filho a esquerda do no pai, então aux será o novo filho a esquerda;

**if**((\*node) == (\*node)->f->left) (\*node)->f->left = aux;

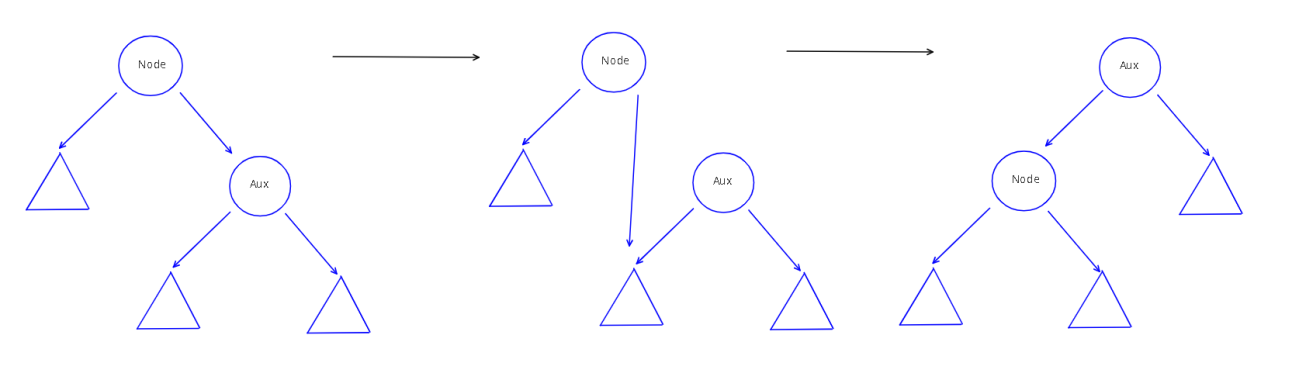
// caso o no que sofreu rotação seja filho a direita do no pai, então aux será o novo filho a direita;

**else** (\*node)->f->right = aux;

}

aux->left = (\*node);

(\*node)->f = aux;

O fluxo de reapontamentos ocorrerá de acordo com a ilustração a seguir:

* 1. Rotação para a direita

**void right\_Rotation(ARN \*\*root, ARN \*\*node);**

A rotação para a direta, definida pela função *right\_rotation*, funciona de forma análoga a função *left\_rotation*, porém de forma “espelhada”.

ARN \*aux;

aux = (\*node)->right;

(\*node)->right = aux->left;

**if**(aux->left!=Sentinela){

aux->left->f = \*node;

}

aux->f = (\*node)->f;

**if**((\*node)->f==Sentinela) \*root = aux;

**else**{

// caso o no que sofreu rotacao seja filho a esquerda do no pai, entao aux sera o novo filho a esquerda;

**if**((\*node) == (\*node)->f->left) (\*node)->f->left = aux;

// caso o no que sofreu rotacao seja filho a direita do no pai, entao aux sera o novo filho a direita;

**else** (\*node)->f->right = aux;

}

aux->left = (\*node);

(\*node)->f = aux;

1. Busca

Eae prof <3

**ARN \*search\_Artigo(ARN \*root, int idn);**

Nessa função a ideia é buscar o nó que contém a chave do artigo desejado à partir da análise das sub-árvores da raiz em questão. O loop continuará sendo executado enquanto o elemento não for encontrado. O caso de parada se dá quando todos os elementos até a possível posição do elemento buscado já foram analisados, isto é, quando o ponteiro chega até um nó Sentinela.

ARN \*aux = root;

**while**(aux!=Sentinela){

**if**(aux->key->id == idn) **break**;

// caso a chave do no atual seja menor que a chave procurada, vá para a direita;

**else** **if**(aux->key->id < idn) aux = aux->right;

// caso a chave do no atual seja maior que a chave procurada, vá para a esquerda;

**else** **if**(aux->key->id > idn) aux = aux->left;

}

**return** aux;

1. Remoção

**void remove\_ARN(ARN \*\*root, ARN \*\*node);**

O algoritmo de remoção foi implementado de modo a reapontar os nós ligados ao que será deletado de maneira que o “desprenda” da árvore. Na remoção, faz-se o uso de dois algoritmos auxiliares em seus casos: a *moreleft\_ARN* e o *transfer\_NodeARN*, que serão apresentados no tópico de “Auxiliares da Remoção”. Vale ressaltar, que a chave a ser deletada está sendo guardada, através da função de busca, na função main, onde a mesma terá seu espaço de memória liberado após a impressão de seus dados.

**int** caux;

ARN \*a1, \*a2;

a2 = \*node;

caux = a2->color;

// Nó a ser deletado nao possui filho a esquerda

**if**((\*node)->left == Sentinela){

a1 = (\*node)->right;

transfer\_NodeARN(root, node, &(\*node)->right);

}

**else**{

// Nó a ser deletado nao possui filho a direita

**if**((\*node)->right == Sentinela){

a1 = (\*node)->left;

transfer\_NodeARN(root, node, &(\*node)->left);

}

**else**{

// caso em que ele possui os dois filhos, onde precisamos pegar o seu sucessor

a2 = moreleft\_ARN((\*node)->right);

caux = a2->color;

a1 = a2->right;

// caso em o no a ser deletado eh pai do seu sucessor ou seja, o seu filho a direita;

// consequencia do more\_left retornar o próprio filho a direita;

**if**(a2->f == \*node) a1->f = a2;

// caso em que o sucessor do no a ser deletado é o filho mais a esquerda de seu filho a direita.

**else**{

transfer\_NodeARN(root, &a2, &a2->right);

a2->right = (\*node)->right;

a2->right->f = a2;

}

// reapontamentos necessários para manter a estrutura da ARN pós-caso em que o no a ser

// deletado possui dois filhos;

transfer\_NodeARN(root, node, &a2);

a2->left = (\*node)->left;

a2->left->f = a2;

a2->color = (\*node)->color;

}

}

// Fixup para as violacoes ocorridas

**if**(caux == **0**) remove\_Fixup(root, &a1);

O passo mais importante da remoção acontece na função *remove\_Fixup*. Nela ocorrerá o devido rebalanceamento da ARN, uma vez que o processo de reapontamento da remoção não garante que as propriedades da árvore permaneçam satisfeitas.

* 1. Remove\_Fixup

**void remove\_Fixup(ARN \*\*root, ARN \*\*node);**

A ideia deste algoritmo é corrigir os possíveis erros relacionados ao balanceamento e quebra de propriedades gerado durante a remoção. É aqui, onde são tratadas as rotações e as mudanças de cor da árvore.

**while**((\*node != \*root) && (\*node)->color==**0**){

// caso node seja filho a esquerda de seu pai;

**if**(\*node == (\*node)->f->left){

// aux recebe o seu irmao a direita;

aux = (\*node)->f->right;

O algoritmo inicia com um loop que percorrerá a árvore de baixo pra cima, a partir do Nó passado, enquanto a cor de “*node*” for negra. O tratamento do loop se dá pelo fato do Nó (node) em questão possuir dupla cor negra, uma vez que caso ele seja rubro, o problema facilmente é resolvido mudando sua cor de rubro para negro.

Como consequência da entrada no loop, há quatro possíveis casos a serem tratados. Vale ressaltar que um Nó não possui fisicamente duas cores, e sim é apenas uma abstração do que realmente acontece.

**CASO 1:**

Este é o caso onde *node* possui "duas cores iguais" (duplo negro), e seu irmão possui cor rubra. Há uma adaptação da árvore de maneira que ela entre em algum dos próximos casos.

**if**(aux->color==**1**){

aux->color = **0**;

(\*node)->f->color = **1**;

left\_Rotation(root, &(\*node)->f);

aux = (\*node)->f->right;

}

**CASO 2:**

Este é o caso onde o *node* possui "duas cores iguais (duplo negro)", e seu irmão possui dois filhos negros. Daí, "remove-se" uma cor negra de cada sub-árvore da raiz (pai de *node*), reparando o que antes "feria" a propriedade de que um nó tem somente uma cor.

**if**((aux->left->color==**0**) && (aux->right->color==**0**)){

aux->color = **1**;

(\*node) = (\*node)->f;

}

Caso não entre no segundo caso, sabemos que os filhos do irmão de *node* não são ambos negros. Portanto, trata-se “separadamente” esse caso.

**CASO 3:**

Este é o caso onde o *node* possui "duas cores iguais (duplo negro)", e seu irmão filhos de cores diferentes, negro pela direita e rubro pela esquerda (exatamente nesta ordem). Este caso pode desbalancear as sub-árvores de *root*, já que a quantidade de nos negros é diferente. Aqui, há uma adaptação no caso 3 para que entre no 4.

**else**{

**if**(aux->right->color == **0**){

aux->left->color = **0**;

aux->color = **1**;

right\_Rotation(root, &aux);

aux = (\*node)->f->right;

}

**CASO 4:**

Este é o caso onde *node* é duplo negro, e seu irmão possui filho a direita rubro. Neste caso, não importa a cor do filho da esquerda, contanto que o da direita seja rubro. Aqui, realiza-se reapontamentos com o objetivo de remover uma das cores de *node,* tornando-o um nó com apenas uma cor. O laço termina com a atribuição *node* = *root*.

aux->color = (\*node)->f->color;

(\*node)->f->color = **0**;

aux->right->color = **0**;

left\_Rotation(root, &(\*node)->f);

\*node = \*root;

}

}

... Parte espelhada do código, referente a quando *node* é filho a direita de seu pai.

Em sua última linha, trata-se o caso para que a raiz seja sempre negra.

(\*node)->color = 0;

* 1. Algoritmos auxiliares da remoção

**moreleft\_ARN:**

A função auxiliar *moreleft\_ARN*, irá procurar o nó mais a esquerda do Nó passado (que será o elemento sucessor da contagem).

**ARN \*moreleft\_ARN(ARN \*node);**

// caso onde o elemento mais a esquerda ja eh o node

**if**(node == Sentinela || node->left == Sentinela) **return** node;

**else** **return** moreleft\_ARN(node->left);

**transfer\_NodeARN:**

A função auxiliar *transfer\_NodeARN,* realizará a transferência de pais de dois *nodes* passados, onde o filho (n2) do Nó a ser deletado (n1) recebe seu avô como pai.

**void transfer\_NodeARN(ARN \*\*root, ARN \*\*n1, ARN \*\*n2);**

**if**((\*n1)->f==Sentinela) (\*root) = \*n2;

**else**{

// caso n1 seja filho a esquerda do pai dele;

**if**(\*n1 == (\*n1)->f->left) (\*n1)->f->left = \*n2;

// caso n1 seja filho a direita do pai dele;

**else** (\*n1)->f->right = \*n2;

}

// troca de "pais" entre os nos;

(\*n2)->f = (\*n1)->f;

1. Auxiliares Gerais
   1. Cria Sentinela

Função responsável por reservar um espaço de memória dedicado a um ponteiro do tipo “ARN” que será um Nó personalizado, retornando-o caso a alocação seja bem sucedida.

ARN \***create\_Sentinela**(){

ARN \*newNode = (ARN \*)malloc(**sizeof**(ARN));

**if**(newNode){

newNode->key = NULL;

newNode->f = NULL;

newNode->left = NULL;

newNode->right = NULL;

newNode->color = **0**;

}

**return** newNode;

}

* 1. Cria ARN

Função responsável por reservar um espaço de memória dedicado a um ponteiro do tipo “ARN”, retornando-o caso a alocação seja bem sucedida.

ARN \***create\_ARN**(artigo \*element){

ARN \*newNode = (ARN \*)malloc(**sizeof**(ARN));

**if**(newNode){

newNode->key = element;

newNode->f = Sentinela;

newNode->left = Sentinela;

newNode->right = Sentinela;

newNode->color = **1**;

}

**return** newNode;

}

* 1. Imprime ARN

Função responsável por imprimir todos os elementos de uma ARN.

**void** **print\_ARN**(ARN \*root, **int** level){

**int** i;

// impressão da ARN por nível, começando da raiz ate a folha

**if**(root != Sentinela){

// chamada da impressão para a sub-arvore a direita

print\_ARN(root->right, level+**1**);

**for**(i = **0**; i<level; i++) printf("**\t\t**");

// impressão de um elemento da ARN

**if**(root->color==**0**)

printf("[%d: N]**\n\n**", root->key->id);

**else** printf("[%d: R]**\n\n**", root->key->id);

// chamada da impressão para a sub-arvore a esquerda

print\_ARN(root->left, level+**1**);

}

}

* 1. Destrói ARN

Função responsável por liberar espaço de memória antes dedicado aos nós da ARN.

**void** **destroy\_ARN**(ARN \*\*root){

**if**(\*root == Sentinela) **return**;

destroy\_ARN(&(\*root)->left);

destroy\_ARN(&(\*root)->right);

// destroi efetivamente a key elemento em questão

free((\*root)->key);

free(\*root);

**return**;

}

* 1. Compara id de Artigos

Função responsável por comparar a chave de dois artigos. Retorna 1 caso o primeiro seja maior que o segundo, 0 se forem iguais e -1 caso o primeiro for menor que o segundo.

**int** **compare\_Artigo**(artigo \*a1, artigo \*a2){

**if**(a1->id > a2->id) **return** **1**;

**else** **if**(a1->id < a2->id) **return** -**1**;

**return** **0**;

}

1. Cliente (Main)
   1. Coleta e inserção de dados de um Artigo.

Função responsável pela leitura e criação de um espaço do tipo artigo.

artigo \***putInformation\_artigo**(){

artigo \*newArtigo = (artigo \*)malloc(**sizeof**(artigo));

**char** a[**200**];

**if**(newArtigo){

**int** len;

printf("**\n\t** Digite as informacoes do artigo: ");

printf("**\n\t** ID: ");

scanf("%d", &newArtigo->id);

printf("**\n\t** Ano de publicacao: ");

scanf("%d", &newArtigo->ano);

printf("**\n\t** Autor: ");

getchar();

fgets(a, **200**, stdin);

len = strlen(a);

a[--len] = **0**;

strcpy(newArtigo->autor, a);

printf("**\n\t** Titulo: ");

fgets(a, **200**, stdin);

len = strlen(a);

a[--len] = **0**;

strcpy(newArtigo->titulo, a);

printf("**\n\t** Revista: ");

fgets(a, **200**, stdin);

len = strlen(a);

a[--len] = **0**;

strcpy(newArtigo->revista, a);

printf("**\n\t** DOI: ");

fgets(a, **200**, stdin);

len = strlen(a);

a[--len] = **0**;

strcpy(newArtigo->DOI, a);

printf("**\n\t** Palavra Chave: ");

fgets(a, **200**, stdin);

len = strlen(a);

a[--len] = **0**;

strcpy(newArtigo->palavraChave, a);

}

**return** newArtigo;

}

* 1. Imprime dados

Função responsável por imprimir os dados de uma Comanda.

**void** **print\_artigo**(artigo \*art){

printf("**\n**");

printf("**\n\t** ID: %d", art->id);

printf("**\n\t** Ano: %d", art->ano);

printf("**\n\t** Autor: %s", art->autor);

printf("**\n\t** Titulo: %s", art->revista);

printf("**\n\t** Revista: %s", art->revista);

printf("**\n\t** DOI: %s", art->DOI);

printf("**\n\t** Palavra Chave: %s", art->palavraChave);

printf("**\n**");

}

* 1. Imprime Menu

Função responsável por imprimir o Menu formatado.

**void** **print\_menu**(){

**int** i;

printf("**\n\n\t** ");

**for** (i = **0**; i<**40**; i++) printf("-");

printf("**\n\t** | 1 - Inserir um novo elemento na ARN**\t**|");

printf("**\n\t** | 2 - Remover um elemento da ARN **\t**|");

printf("**\n\t** | 3 - Buscar um elemento na ARN **\t**|");

printf("**\n\t** | 4 - Imprimir ARN **\t\t\t**|");

printf("**\n\t** | 5 - Sair do programa**\t\t\t**|");

printf("**\n\t** ");

**for** (i = **0**; i<**40**; i++) printf("-");

}