UFAM

UNIVERSIDADE FEDERAL DO AMAZONAS

INSTITUTO DE COMPUTAÇÃO

IEC013 – ALGORITMOS E ESTRUTURAS DE DADOS II

FT05 – Engenharia da Computação

21753594 – Josias Ben Ferreira Cavalcante

Árvore Rubro Negra

Manaus-AM

2019

21753594 – Josias Ben Ferreira Cavalcante

Árvore Rubro Negra

Relatório de trabalho final apresentado junto ao curso de Engenharia da Computação como requisito à obtenção de nota de prova final para o primeiro período de 2019 (2019/1).

Professor: Edson Nascimento

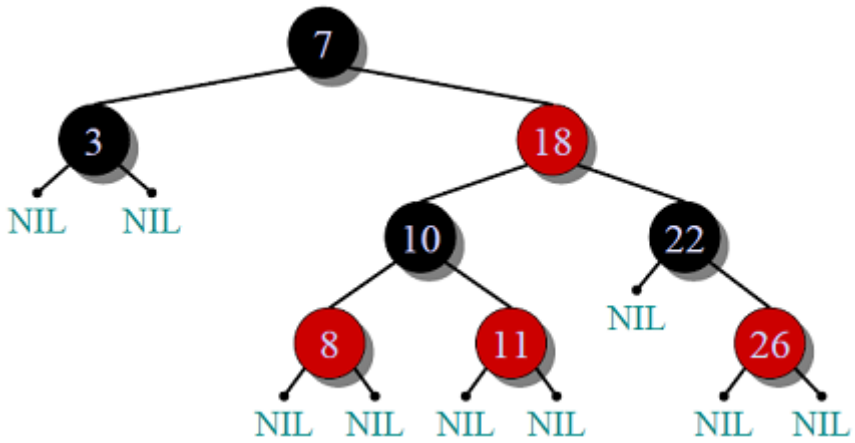
Manaus-AM

2019

INTRODUÇÃO

Neste relatório estarei apresentando a estrutura da árvore rubro negra, uma árvore binária de busca auto balanceada em que cada um de seus nós seguem regras rigorosamente para promover a proximidade do balanceamento da árvore.

Sobre a Estrutura



Como dito anteriormente, a árvore rubro negra é um tipo de árvore binária de busca auto balanceada. Foi inventada por Rudolf Bayer, em 1972, que a chamou de “Árvores Binárias B Simétrica”, mas passou a ser chamada de “Rubro Negra” (Red Black, do inglês) em um artigo escrito por Leonidas J. Guibas e Robert Sedgewick em 1978.

É uma estrutura complexa, porém possui um bom pior-caso de tempo de execução para suas operações e é eficiente na prática: pode buscar, inserir e remover em tempo O(log n), onde n é o número total de elementos da árvore.

Em resumo, a árvore rubro negra é uma árvore binária de busca que insere e remove de forma inteligente, para assegurar que a árvore permaneça aproximadamente balanceada.

São regras da rubro negra:

1) Todo nó possui uma cor: Preto ou Vermelho.
2) A Raiz da árvore sempre será da cor Preta.
3) Não pode haver dois nós Vermelhos adjacentes, ou seja, um nó vermelho não pode ter um pai vermelho ou um filho vermelho. Se um nó é vermelho, então todos os seus filhos são pretos
4) Toda folha null é preta.
5) Cada caminho de um nó (incluindo a raiz) para qualquer no descendente no null folha possui o mesmo número de nós pretos.

A maioria das operações em árvores binárias de busca tomam o tempo O(h), onde h é a altura da árvore. O custo dessas operações pode se tornar O(n) para uma árvore binária distorcida (tão distorcida a ponto de se tornar uma simples lista encadeada). Se nos certificarmos de que a altura da árvore permanece O(log n) após cada inserção e remoção, então podemos garantir um limite superior de O(log n) para todas as operações.

A altura de uma árvore rubro negra sempre é O(log n), onde n é o número de nós na árvore.

Algumas boas práticas sugerem a criação de um nó sentilena. Todos os nós folhas (nill) apontarão para este nó, caracterizando o NULL.

Objetivos

Implementar a Árvore Rubro Negra, bem como suas operações de inserção, remoção, busca e impressão dos dados.

Da Implementação

- Estrutura:

```
//_____________________________________________________________________________
//                          ESTRUTURA DA ARVORE RUBRO NEGRA                    |
//-----------------------------------------------------------------------------
typedef struct No{
    int chave;
    short int cor;
    struct No *pai;
    struct No *filhoEsq;
    struct No *filhoDir;
}No;
//_____________________________________________________________________________
```

A variável inteira chave indica o valor de uma chave. A variável cor (1 bit) indica se o nó é Preto ou Vermelho (1 para preto). A variável pai é um ponteiro que aponta para o pai do nó. A raiz tem seu pai apontado para nill. As variáveis filhoEsq e filhoDir apontam para a subárvore esquerda e direita, respectivamente.

- Variáveis Globais:

```
//_____________________________________________________________________________
//                                VARIAVEIS GLOBAIS                            |
//-----------------------------------------------------------------------------
enum Cor {Vermelho=0, Preto=1};
No *raiz;
No *null;
//_____________________________________________________________________________
```

O programa que estarei implementando tratará apenas uma árvore, apenas para fins didáticos e de explicação. Logo, haverá uma raiz global que será manipulada por opções dadas na função main. Aqui é criado o nó sentinela, o qual todos os nós NULL e as folhas apontarão.

Funções de Criação

- Função Criar Nó:

```
//_____________________________________________________________________________
//                    FUNCAO CRIAR NO (alocagem e setup inicial)               |
//-----------------------------------------------------------------------------
No *criarNo(int chave, No *filhoEsq, No *filhoDir, No *pai){
    No *no = malloc(sizeof(*no));
    no->chave = chave;
    no->pai = pai;
    no->filhoEsq = filhoEsq;
    no->filhoDir = filhoDir;
    //no->cor = Vermelho;
    return no;
}
//_____________________________________________________________________________
```

Função responsável por criar um nó, sendo passado os parâmetros chave, e os ponteiros. Não será atribuída imediatamente uma cor para o nó, pois este quem fará será a operação de inserção, quando o nó estiver na árvore.

- Função Inicializadora:

```
//_____________________________________________________________________________
//                 FUNCAO CRIAR ARVORE (alocagem e setup inicial)              |
//-----------------------------------------------------------------------------
void iniciarArvore(){
    // null e raiz globais
    null = criarNo(0, NULL, NULL, NULL);
    null->cor = Preto;
    raiz = null;
}
//_____________________________________________________________________________
```

Função responsável por caracterizar o nó sentinela. Quando uma árvore é criada, ela começa com um ponteiro apontando para o null global, logo, essa função se encarregará disso.

Funções Flag

- Função Qual cor:

```
// FUNCAO OBTER COR DO NO
int qualCor(No *no){
    if(no == null){
        return Preto;
    }
    else return no->cor;
}
```

Retorna a cor de dado nó

- Função Tem Filho:

```
// VERIFICA SE O NO TEM FILHO ESQUERDO
int temFilhoEsq(No *no){
    if(no != null && no->filhoEsq!=null){
        return 1;
    }
    return 0;
}

// VERIFICA SE O NO TEM FILHO DIREITO
int temFilhoDir(No *no){
    if(no != null && no->filhoDir!=null){
        return 1;
    }
    return 0;
}
```

Ambas funções de verificação de existência de filhos de um nó.

- Função Tem Pai:

```
// VERIFICA SE O NO TEM PAI (SE NAO EH RAIZ)
int temPai(No *no){
    if(no!=null && no->pai!=null){
        return 1;
    }
    return 0;
}
```

Com essa função é possível verificar se um nó possui um pai, logo, se não é raiz.

- Função Tem Irmão:

```
// VERIFICA SE O NO TEM IRMAO
int temIrmao(No *no){
    if((no != null)&&(temPai(no))&&(temFilhoDir(no->pai))&&(temFilhoEsq(no->pai))){
        return 1;
    }
    return 0;
}
```

Função que verifica se dado nó possui irmão, isto é, se o pai do nó possui ambos os filhos.

- Função Tem avô:

```
// VERIFICA SE O NO TEM AVO
int temAvo(No *no){
    if(no->pai != null && no->pai->pai != null){
        return 1;
    }
    return 0;
}
```

Função que verifica se o pai de um dado nó possui um pai.

- Função Tem Tio:

```
// VERIFICA SE O NO TEM TIO
int temTio(No *no){
    if(temPai(no) && temIrmao(no->pai)){
        return 1;
    }
    return 0;
}
```

Função que verifica se o pai de um dado nó possui irmão, isto é, se o avô do nó possui os dois filhos diferentes de null.

Funções de Manipulação

- Função troca cores:

```
// FUNCAO TROCA AS CORES ENTRE DOIS NOS
void trocarCores(No *a, No *b){
    int corDeA = a->cor;
    int corDeB = b->cor;
    a->cor = corDeB;
    b->cor = corDeA;
}
```

Função responsável por trocar as cores entre dois nós.

- Função Inverte cor:

```
// FUNCAO INVERTE COR
void inverterCor(No *no){
    if(no == null){
        return;
    }
    no->cor = !(no->cor);
}
```

Função responsável por retornar um nó com sua cor invertida, isto é, se é passado como parâmetro um nó da cor preta, a função retorna esse mesmo nó, só que da cor Vermelha, e vice-versa.

- Função Altera Cor:

```
//FUNCAO ALTERAR COR DO NO
void alterarCor(No *no, int cor){
    if(no == null) {
        return;
    }
    no->cor = cor;
}
```

Função responsável por alterar o nó para uma cor indicada no parâmetro.

- Função Pai do No:

```
// RETORNA O PAI DE UM DADO NO
No *paiDoNo(No *no){
    return no->pai;
}
```

Função por retornar o nó pai.

- Função Avô do Nó:

```
// RETORNA O AVO DE UM DADO NO
No *avoDoNo(No *no){
    return(paiDoNo(no))->pai;
}
```

Retorna o avô do Nó

- Função Maior dos Menores (MdM):

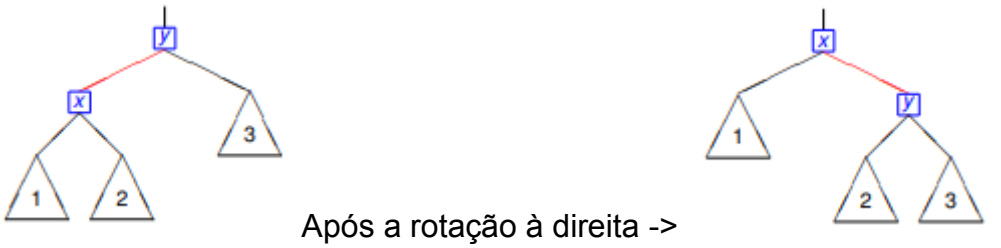
```
// RETORNA O MDM DO NO (MENOR DOS MAIORES)
No *menorDosMaiores(No *no){
    No *aux;
    if(no->filhoDir == null){
        return null;
    }
    aux = no->filhoDir;
    while(aux->filhoEsq != null){
        aux = aux->filhoEsq;
    }
    return aux;
}
```

Retorna o nó que possui a chave imediatamente maior que a do nó dado.

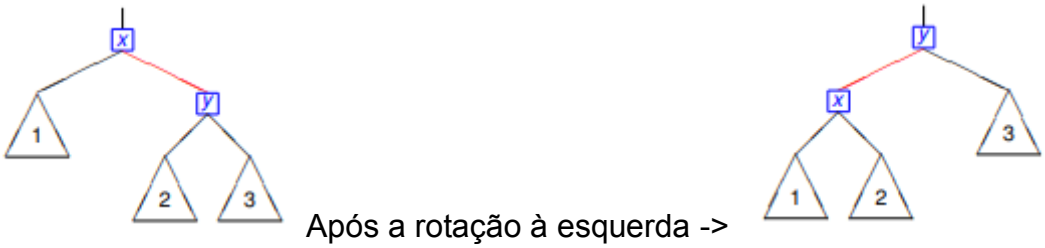
Funções de Rotação

Para que nenhuma das propriedades da Árvore rubro negra seja quebrada, é necessário fazer alterações na estrutura da árvore. Tais alterações são feitas por dois tipos de rotações: Rotação à Direita e Rotação à Esquerda.

O algoritmo de rotação (seja para direita ou para esquerda) muda alguns ponteiros da árvore, preservando a propriedade de Árvore Binária de Busca. Leva tempo constante O(1) para efetuar ser efetuada.

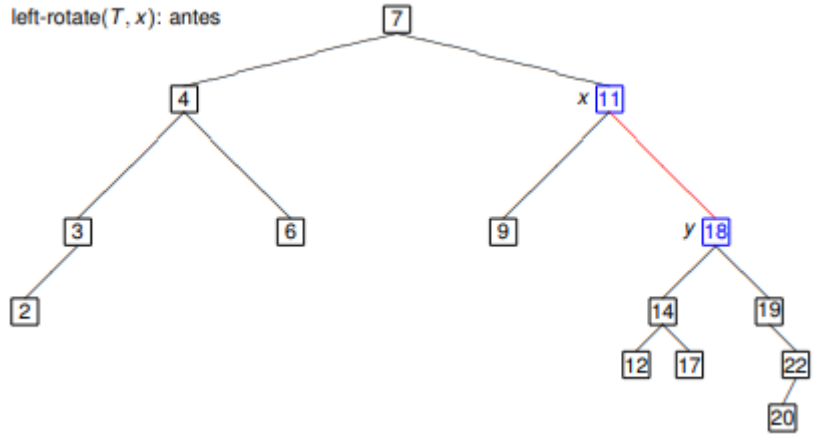


Após a rotação à direita ->
Rotaciona Y

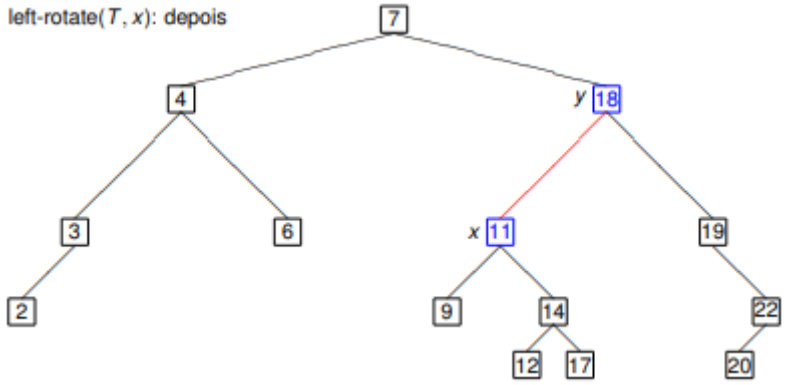


Após a rotação à esquerda ->
Rotaciona X

Rotacionando o nó X à esquerda (ANTES):



Rotacionando o nó X à esquerda (DEPOIS).



- Função Rotação à Direita

```
//ROTACIONAR A DIREITA
No *rotacionarDireita(No *raiz, No *no){
    No *auxiliar = no->filhoEsq;
    no->filhoEsq = auxiliar->filhoDir;
    if(auxiliar->filhoDir != null){
        auxiliar->filhoDir->pai = no;
    }
    auxiliar->pai = no->pai;
    if(paiDoNo(no) == null){
        raiz = auxiliar;
    }
    else if(no == paiDoNo(no)->filhoEsq){
        no->pai->filhoEsq = auxiliar;
    }
    else{
        paiDoNo(no)->filhoDir = auxiliar;
    }
    auxiliar->filhoDir = no;
    no->pai = auxiliar;
    return raiz;
}
```

Função responsável por operar a rotação à direita em determinado nó de uma árvore, quando necessário.

- Função Rotação à Esquerda

```
// ROTACIONAR A ESQUERDA
No *rotacionarEsquerda(No *raiz, No *no){
    No *auxiliar = no->filhoDir;
    no->filhoDir = auxiliar->filhoEsq;
    if(auxiliar->filhoEsq != null){
        auxiliar->filhoEsq->pai = no;
    }
    auxiliar->pai = no->pai;
    if(paiDoNo(no) == null){
        raiz = auxiliar;
    }
    else if(no == paiDoNo(no)->filhoEsq){
        paiDoNo(no)->filhoEsq = auxiliar;
    }
    else{
        paiDoNo(no)->filhoDir = auxiliar;
    }
    auxiliar->filhoEsq = no;
    no->pai = auxiliar;
    return raiz;
}
```

Função responsável por operar a rotação à esquerda em determinado nó de uma árvore, quando necessário.

Função de Correção após Inserção

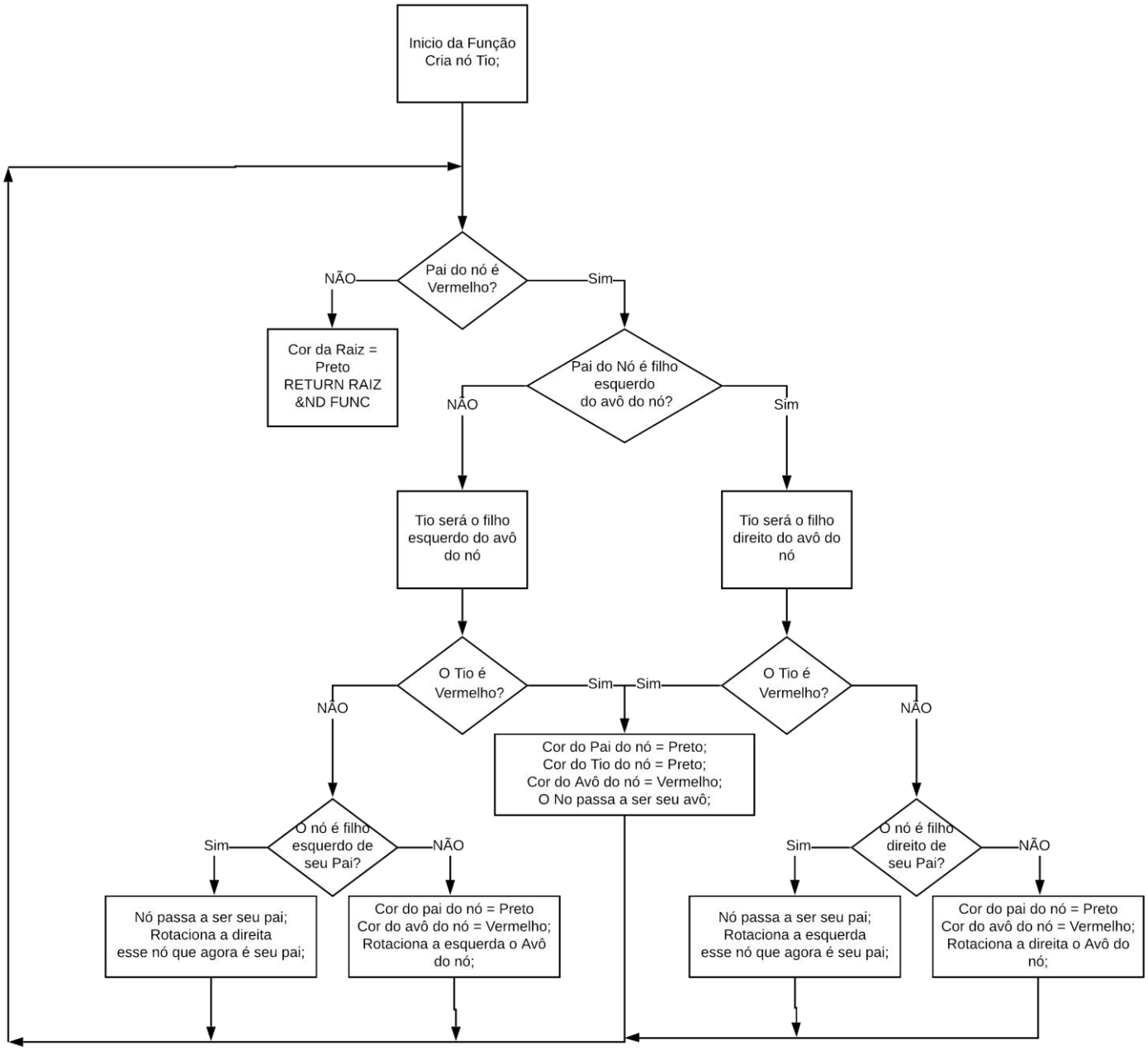
```
No *consertaInsercao(No *raiz, No *no){
    No *tio;
    while(paiDoNo(no)->cor == Vermelho){
        if(paiDoNo(no) == avoDoNo(no)->filhoEsq){
            tio = avoDoNo(no)->filhoDir;
            if(tio->cor == Vermelho){
                paiDoNo(no)->cor = Preto;
                tio->cor = Preto;
                avoDoNo(no)->cor = Vermelho;
                no = avoDoNo(no);
            }
            else{
                if(no == paiDoNo(no)->filhoDir){
                    no = paiDoNo(no);
                    raiz = rotacionarEsquerda(raiz, no);
                }
                else{
                    paiDoNo(no)->cor = Preto;
                    avoDoNo(no)->cor = Vermelho;
                    raiz = rotacionarDireita(raiz, avoDoNo(no));
                }
            }
        }

        else{
            tio = avoDoNo(no)->filhoEsq;
            if(tio->cor == Vermelho){
                paiDoNo(no)->cor = Preto;
                tio->cor = Preto;
                avoDoNo(no)->cor = Vermelho;
                no = avoDoNo(no);
            }
            else{
                if(no == paiDoNo(no)->filhoEsq){
                    no = paiDoNo(no);
                    raiz = rotacionarDireita(raiz, no);
                }
                else{
                    paiDoNo(no)->cor = Preto;
                    avoDoNo(no)->cor = Vermelho;
                    raiz = rotacionarEsquerda(raiz, avoDoNo(no));
                }
            }
        }
    }

    raiz->cor = Preto;
    return raiz;
}
```

Enquanto o pai do no recém inserido for vermelho, será feita uma verificação em seu tio.
Se o tio do for Vermelho, será necessário fazer uma recoloração dos nós. Mas se caso for preto, será necessário fazer rotações que dependem de que posição está o tio, o pai do nó e o próprio nó.

Se o Tio é filho direito do avô do nó (implicando que o pai do nó é filho esquerdo do avô do nó):
. Se o tio é Vermelho:
. . Realiza recoloração dos nós;
. Senão:
. . .Se o nó é filho direito de seu pai:
. . . Realiza rotação à esquerda;
. . .Senão:
. . . Recolore o pai e o avo, e rotaciona à direita;
.
Senão:
. Se o tio é Vermelho:
. . Realiza recoloração dos nós;
. Senão:
. . Se o nó é filho é filho esquerdo de seu pai:
. . . Realiza rotação à direita;
. . Senão: Recolore o pai e o avo, e rotaciona à esquerda;
. . .

Findado o while, a raiz em algum momento pode ter deixado de ser Preta. Nesse caso, atribuímos a cor Preta para ela.

Segue abaixo o Fluxograma de Processos desta função:



- Função Inserir:

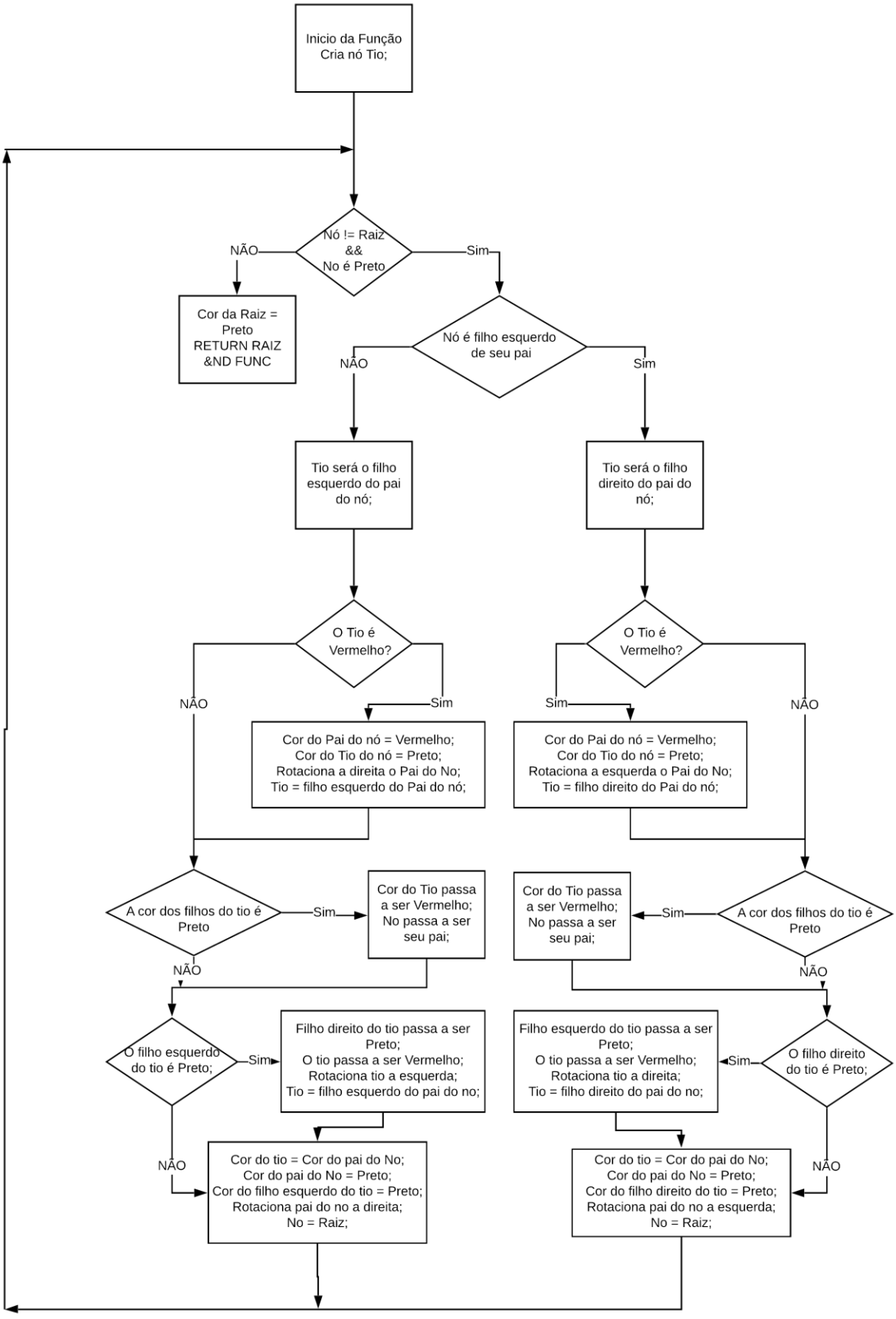
```
//_____________________________________________________________________________
//                            INSERCAO SIMPLES EM BST                          |
//-----------------------------------------------------------------------------
No *inserir(No *raiz, int chave){
    if(buscar(raiz,chave)== null){
        No *novo = criarNo(chave, null, null, null); //NO A SER INSERIDO
        No *caminhador = raiz; // A BUSCA COMECA PELA RAIZ
        No *paix = null; // DURANTE A CAMINHADA SALVAREI O PAI DO NOVO NO
        while(caminhador != null){ //busca o pai do nodo novo
            paix = caminhador;
            if(chave < caminhador->chave){
                caminhador = caminhador->filhoEsq;
            }
            else{
                caminhador = caminhador->filhoDir;
            }
        }
        novo->pai = paix;
        if(paix == null){ //arvore vazia
            raiz = novo;
        }
        else if(chave < paix->chave){
            paix->filhoEsq = novo;
        }
        else{
            paix->filhoDir = novo;
        }
        novo->cor = Vermelho;
        raiz = consertaInsercao(raiz, novo);
    }
    return raiz;
}
```

Função responsável for fazer a inserção simples em árvore binária de busca, preservando o pai do novo nó; e por fim, fazer a chamada da função que corrige o nó recém inserido.

Função de Correção após Remoção

```c
void consertaRemocao(No *raiz, No *no){
    No *tio;
    while(no!=raiz && no->cor==Preto){
        if(no == paiDoNo(no)->filhoEsq){
            tio = paiDoNo(no)->filhoDir;
            if(tio->cor == Vermelho){
                tio->cor = Preto;
                paiDoNo(no)->cor = Vermelho;
                rotacionarEsquerda(raiz, paiDoNo(no));
                tio = paiDoNo(no)->filhoDir;
            }
            if(tio->filhoEsq->cor == Preto && tio->filhoDir->cor == Preto){
                tio->cor = Vermelho;
                no = paiDoNo(no);
            }
            else if(tio->filhoDir->cor == Preto){
                tio->filhoEsq->cor = Preto;
                tio->cor = Vermelho;
                rotacionarDireita(raiz, tio);
                tio = paiDoNo(no)->filhoDir;
            }
            tio->cor = paiDoNo(no)->cor;
            paiDoNo(no)->cor = Preto;
            tio->filhoDir->cor = Preto;
            rotacionarEsquerda(raiz, paiDoNo(no));
            no = raiz;
        }

        else{
            tio = paiDoNo(no)->filhoEsq;
            if(tio->cor == Vermelho){
                tio->cor = Preto;
                paiDoNo(no)->cor = Vermelho;
                rotacionarDireita(raiz,paiDoNo(no));
                tio = paiDoNo(no)->filhoEsq;
            }
            if(tio->filhoDir->cor == Preto && tio->filhoEsq->cor == Preto){
                tio->cor = Vermelho;
                no = paiDoNo(no);
            }
            else if(tio->filhoEsq->cor == Preto){
                tio->filhoDir->cor = Preto;
                tio->cor = Vermelho;
                rotacionarEsquerda(raiz, tio);
                tio = paiDoNo(no)->filhoEsq;
            }
            tio->cor = paiDoNo(no)->cor;
            paiDoNo(no)->cor = Preto;
            tio->filhoEsq->cor = Preto;
            rotacionarDireita(raiz, paiDoNo(no));
            no = raiz;
        }
    }
    no->cor = Preto;
}
```

Segue abaixo o Fluxograma de Processos desta função:



- Função Remover:

```
void remover(No *raiz, No *noDaChave){
    No* noARemover;
    No* filhoDoNoRemovido;
    if(noDaChave == null)
        return;
    if(noDaChave->filhoEsq == null || noDaChave->filhoDir == null) // Nó sem ambos filhos
        noARemover = noDaChave;
    else
        noARemover = menorDosMaiores(noDaChave); // Caso tenha ambos filhos, pega mDm


    if(noARemover->filhoEsq != null)
        filhoDoNoRemovido = noARemover->filhoEsq;
    else
        filhoDoNoRemovido = noARemover->filhoDir;
    filhoDoNoRemovido->pai = paiDoNo(noARemover);


    if(paiDoNo(noARemover) == null)
        raiz = filhoDoNoRemovido;
    else if(noARemover == paiDoNo(noARemover)->filhoEsq)
        paiDoNo(noARemover)->filhoEsq = filhoDoNoRemovido;
    else
        paiDoNo(noARemover)->filhoDir = filhoDoNoRemovido;


    if(noDaChave != noARemover) // copia chave e dados do noARemover para noDaChave
        noDaChave->chave = noARemover->chave;
    if(noARemover == raiz){
        raiz = null;
        return;
    }
    free(noARemover);
    if(noARemover->cor == Preto)
        consertaRemocao(raiz,filhoDoNoRemovido);
}
```

Realiza uma remoção simples em árvore binária. Se necessário, faz a chamada da função corrige remoção.

- Função Buscar:

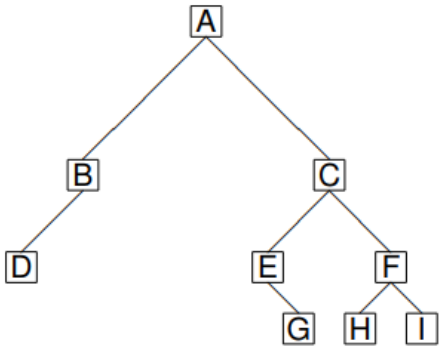
```
No *buscar(No *no, int chave){
    if(no == null){
        return null;
    }
    if(no->chave == chave){
        return no;
    }
    else if(chave < no->chave){
        return buscar(no->filhoEsq, chave);
    }
    else{
        return buscar(no->filhoDir, chave);
    }
}
```

Essa função é idêntica à função de busca em árvore binária de busca.

Impressão

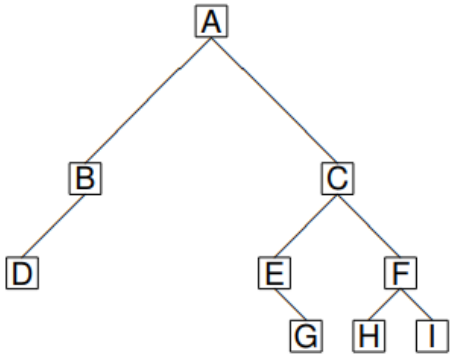
Diversas formas de percorrer uma árvore binária de busca, no entanto, estarei destacando três formas: Pré-Ordem, Em-Ordem e Pós-Ordem.

Pré-Ordem:



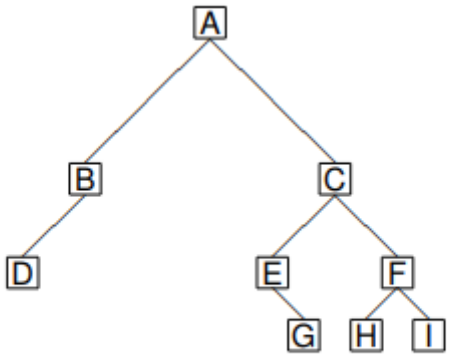
Percurso em pré-ordem: A B D C E G F H I

Em-Ordem:



Percurso em in-ordem: D B A E G C H F I

Pos-Ordem:



Percurso em pós-ordem: D B G E H I F C A

- Funções impressoras:

Impressão Simples no Nó:

```c
void imprimeNo(No *no){
    printf("CHAVE: %d   ", no->chave);
    printf("COR: ");
    if(no->cor){
        printf("Preto\n");
    }
    else{
        printf("Vermelho\n");
    }
}
```

Impressão detalhada no Nó:

```c
void statusNo(No *no){
    if(no!=NULL && no!=null){
        printf("CHAVE: %d\n", no->chave);
        printf("COR: ");
        if(no->cor){
            printf("Preto\n");
        }
        else{
            printf("Vermelho\n");
        }
        if(temAvo(no))
            printf("TEM AVO DE CHAVE: %d\n", no->pai->pai->chave);
        if(temTio(no)){
            if(no->pai == no->pai->pai->filhoEsq)
                printf("TEM TIO DE CHAVE: %d\n", no->pai->pai->filhoDir->chave);
            else
                printf("TEM TIO DE CHAVE: %d\n", no->pai->pai->filhoEsq->chave);
        }
        if(temPai(no))
            printf("TEM PAI DE CHAVE: %d\n", no->pai->chave);
        if(temIrmao(no)){
            if(no == no->pai->filhoEsq)
                printf("TEM IRMAO DE CHAVE: %d\n", no->pai->filhoDir->chave);
            else
                printf("TEM IRMAO DE CHAVE: %d\n", no->pai->filhoEsq->chave);
        }
        if(temFilhoEsq(no))
            printf("TEM FILHO ESQUERDO DE CHAVE: %d\n", no->filhoEsq->chave);
        if(temFilhoDir(no))
            printf("TEM FILHO DIREITO DE CHAVE: %d\n", no->filhoDir->chave);
        printf("\n");
    }
    else printf("O NO INFORMADO EH NULL ! ! !\n");
}
```

Impressão Simples em Árvore:

```
// IMPRIME CHAVE E COR DOS NOS DE UMA ARVORE EM ORDEM
void impressaoSimplesEmOrdem(No *no){
    if(no == null){
        return;
    }
    impressaoSimplesEmOrdem(no->filhoEsq);
    imprimeNo(no);
    impressaoSimplesEmOrdem(no->filhoDir);
}

// IMPRIME CHAVE E COR DOS NOS DE UMA ARVORE EM PRE ORDEM
void impressaoSimplesPreOrdem(No *no){
    if(no == null){
        return;
    }
    imprimeNo(no);
    impressaoSimplesPreOrdem(no->filhoEsq);
    impressaoSimplesPreOrdem(no->filhoDir);
}

// IMPRIME CHAVE E COR DOS NOS DE UMA ARVORE EM POS ORDEM
void impressaoSimplesPosOrdem(No *no){
    if(no == null){
        return;
    }
    impressaoSimplesPosOrdem(no->filhoEsq);
    impressaoSimplesPosOrdem(no->filhoDir);
    imprimeNo(no);
}
```

Impressão Detalhada em Árvore:

```
// IMPRIME O STATUS DOS NOS DE UMA ARVORE EM ORDEM
void impressaoCompletaEmOrdem(No *no){
    if(no==null){
        return; // retorno void
    }
    impressaoCompletaEmOrdem(no->filhoEsq);
    statusNo(no);
    impressaoCompletaEmOrdem(no->filhoDir);
}

// IMPRIME O STATUS DOS NOS DE UMA ARVORE EM PRE ORDEM
void impressaoCompletaPreOrdem(No *no){
    if(no==null){
        return; // retorno void
    }
    statusNo(no);
    impressaoCompletaPreOrdem(no->filhoEsq);
    impressaoCompletaPreOrdem(no->filhoDir);
}

// IMPRIME O STATUS DOS NOS DE UMA ARVORE EM POS ORDEM
void impressaoCompletaPosOrdem(No *no){
    if(no==null){
        return; // retorno void
    }
    impressaoCompletaPosOrdem(no->filhoEsq);
    impressaoCompletaPosOrdem(no->filhoDir);
    statusNo(no);
}
```

Funções Acessórias

- Função Destruir Árvore:

```
void destroir(No *no){
    if(no == null){
        return;
    }
    destroir(no->filhoEsq);
    destroir(no->filhoDir);
    free(no);
}
```

Dá Free em todos os nós da árvore.

- Gerar entradas Aleatórias:

```
No *inserirChavesAleatorias(No *raiz, int quantidade){
    int i;
    int chave=0;
    for(i=0;i<quantidade;i++){
        chave = (int)(1 + rand()%100);
        raiz = inserir(raiz,chave);
        printf("%.2lf%%  -  Inserindo: %d\n",(double)(((double)i/(double)quantidade)*100), chave);
    }
    printf("100.00%%\n");
    return raiz;
}
```

Para realizar simulações de teste.

- Logo do Programa:

```
void imprimirLogo(){
    printf("     /\\                          \n");
    printf("    /  \\    _ ____   _____  _ __ ___  \n");
    printf("   / /\\ \\ | '__\\ \\ / / _ \\| '_ ` _ \\\n");
    printf("  / ____ \\| |   \\ V / (_) | | | | | |\n");
    printf(" /_/    \\_\\_|    \\_/ \\___/|_| |_| |_|\n");

    printf("\n");

    printf("  _____                 _             \n");
    printf(" |  __ \\     | |               \n");
    printf(" | |__) |   _| |__  _ __ ___   \n");
    printf(" |  _  / | | | '_ \\| '__/ _ \\  \n");
    printf(" | | \\ \\ |_| | |_) | | | (_) | \n");
    printf(" |_|  \\_\\__,_|_.__/|_|  \\___/  \n");

    printf("\n");

    printf("  _    _                       \n");
    printf(" | \\ | |                      \n");
    printf(" |  \\| | ___  __ _ _ __ __ _  \n");
    printf(" | . ` |/ _ \\/ _` | '__/ _` | \n");
    printf(" | |\\  |  __/ (_| | | | (_| | \n");
    printf(" |_| \\_|\\___|\\__, |_|  \\__,_| \n");
    printf("              __/ |           \n");
    printf("             |___/            \n");
    printf("\n\tBy: JOSIAS BEN F. CAVALCANTE - 21753594\n");

    system("pause");
}
```

Exibe a tela de abertura e apresentação.

Funções de Menu

- Menu Inicial:

```c
int menuInicial(){
    int opcao;
    printf("_____________________________________________________________________________\n");
    printf("|                  A R V O R E   R U B R O   N E G R A                      |\n");
    printf("-----------------------------------------------------------------------------\n");
    printf("[1] Gerar e Inserir elementos Aleatorios (simulacao)\n");
    printf("[2] Inserir elemento\n");
    printf("[3] Remover elemento\n");
    printf("[4] Buscar elemento\n");
    printf("[5] Imprimir arvore\n");
    printf("[6] Esvaziar arvore\n");
    printf("[7] Sair\n");
    printf("   Por Favor, selecione uma opcao acima:\n");
    scanf("%d", &opcao);
    return opcao;
}
```

- Menu de Busca:

```c
int menuBusca(){
    int opcao;
    printf("_____________________________________________________________________________\n");
    printf("|                     B U S C A R   E L E M E N T O                         |\n");
    printf("-----------------------------------------------------------------------------\n");
    printf("\tPor Favor, selecione uma opcao de detalhamento abaixo:\n");
    printf("[1] Detalhamento Simples (chave e cor)\n");
    printf("[2] Detalhamento Completo (chave, cor e nos relacionais)\n");
    scanf("%d", &opcao);
    return opcao;
}
```

- Menu de Impressão:

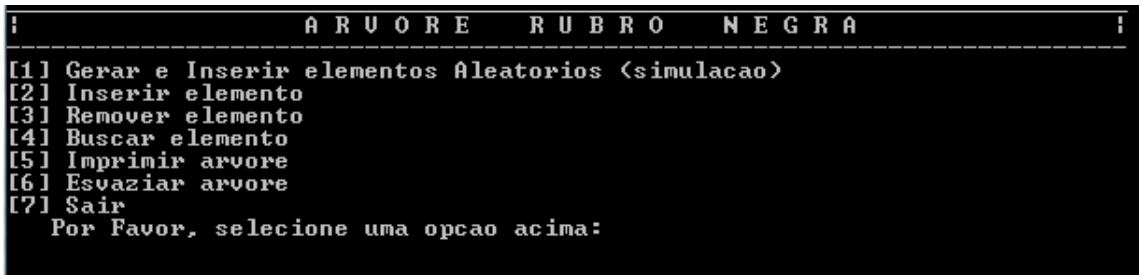
```c
int menuImpressoras(){
    int opcao;
    printf("_____________________________________________________________________________\n");
    printf("|                    I M P R I M I R   A R V O R E                          |\n");
    printf("-----------------------------------------------------------------------------\n");
    printf("\tPor Favor, selecione um metodo de impressao abaixo:\n");
    printf("[1] Pre-ordem SIMPLES\n");
    printf("[2] Em ordem SIMPLES\n");
    printf("[3] Pos-ordem SIMPLES \n");
    printf("[4] Pre-ordem COMPLETA\n");
    printf("[5] Em ordem COMPLETA\n");
    printf("[6] Pos-ordem COMPLETA\n");
    printf("\n[7] Voltar\n");
    scanf("%d", &opcao);
    return opcao;
}
```

Devido as funções Inserir Elemento e Remover Elemento não demandarem mais opções de escolhas, não foi dedicado um menu especial para elas.

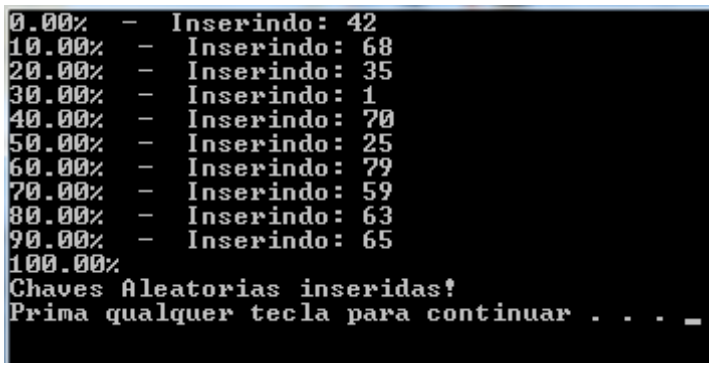
UTILIZANDO O PROGRAMA

Ao iniciar, o log mostra a logo do programa juntamente com o nome e a matricula do desenvolvedor:

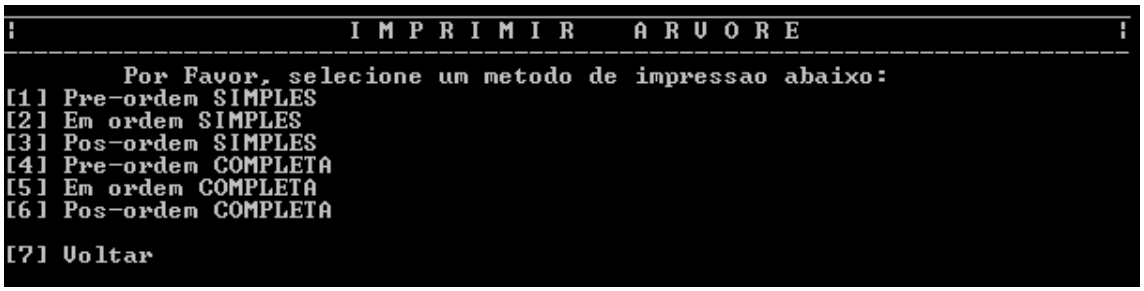
Basta Pressionar qualquer Tecla para continuar. Após pressionar, será levado ao menu principal:



A árvore inicialmente se encontra vazia. Tentar imprimir, buscar ou remover não terão efeito. Começando pela opção 1 vou gerar 10 números aleatórios. Essa primeira função foi desenvolvida para verificar casos pequenos, logo, não me pus a estender a variedade e/ou quantidade de números possíveis.



Após inserir aleatoriamente esses 10 números, vou imprimir a árvore de forma detalhada e Em-Order.

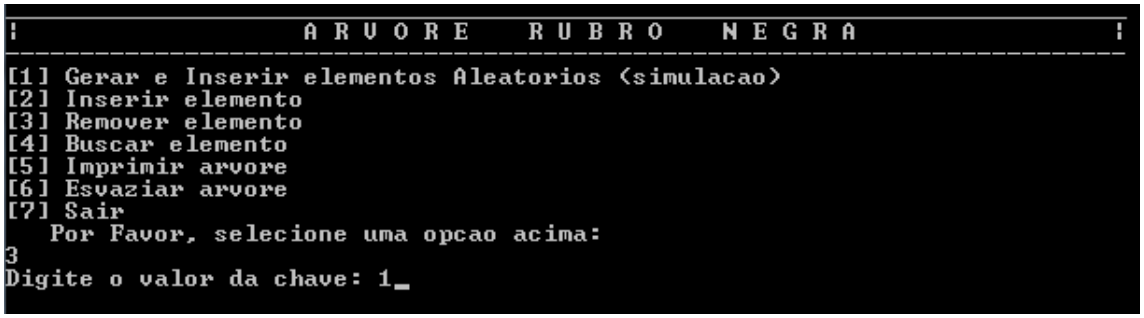


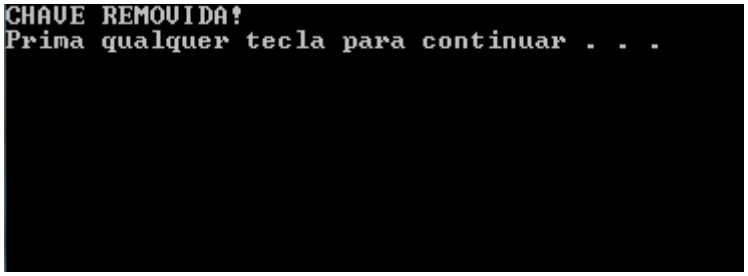
Seleciono a opção 5 e o resultado segue abaixo:

```
CHAVE: 1
COR: Vermelho
TEM AVO DE CHAVE: 42
TEM TIO DE CHAVE: 59
TEM PAI DE CHAVE: 25
TEM IRMAO DE CHAVE: 35

CHAVE: 25
COR: Preto
TEM AVO DE CHAVE: 63
TEM TIO DE CHAVE: 70
TEM PAI DE CHAVE: 42
TEM IRMAO DE CHAVE: 59
TEM FILHO ESQUERDO DE CHAVE: 1
TEM FILHO DIREITO DE CHAVE: 35

CHAVE: 35
COR: Vermelho
TEM AVO DE CHAVE: 42
TEM TIO DE CHAVE: 59
TEM PAI DE CHAVE: 25
TEM IRMAO DE CHAVE: 1

CHAVE: 42
COR: Vermelho
TEM PAI DE CHAVE: 63
TEM IRMAO DE CHAVE: 70
TEM FILHO ESQUERDO DE CHAVE: 25
TEM FILHO DIREITO DE CHAVE: 59

CHAVE: 59
COR: Preto
TEM AVO DE CHAVE: 63
TEM TIO DE CHAVE: 70
TEM PAI DE CHAVE: 42
TEM IRMAO DE CHAVE: 25

CHAVE: 63
COR: Preto
TEM FILHO ESQUERDO DE CHAVE: 42
TEM FILHO DIREITO DE CHAVE: 70

CHAVE: 65
COR: Vermelho
TEM AVO DE CHAVE: 70
TEM TIO DE CHAVE: 79
TEM PAI DE CHAVE: 68

CHAVE: 68
COR: Preto
TEM AVO DE CHAVE: 63
TEM TIO DE CHAVE: 42
TEM PAI DE CHAVE: 70
TEM IRMAO DE CHAVE: 79
TEM FILHO ESQUERDO DE CHAVE: 65

CHAVE: 70
COR: Vermelho
TEM PAI DE CHAVE: 63
TEM IRMAO DE CHAVE: 42
TEM FILHO ESQUERDO DE CHAVE: 68
TEM FILHO DIREITO DE CHAVE: 79

CHAVE: 79
COR: Preto
TEM AVO DE CHAVE: 63
TEM TIO DE CHAVE: 42
TEM PAI DE CHAVE: 70
TEM IRMAO DE CHAVE: 68
```

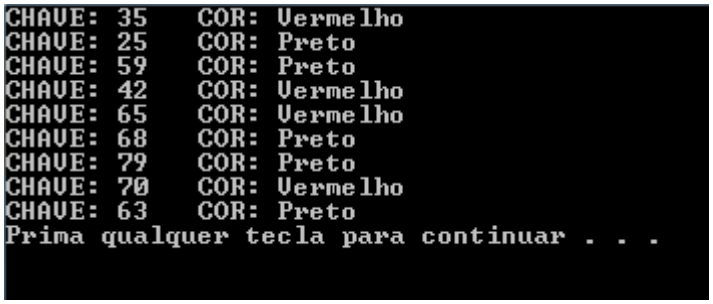
O interessante da impressão detalhada é que, além de saber a chave e a cor de certo nó, posso saber a relação dele com os outros nós.

Observando isso, irei remover o nó que possui a chave 1.



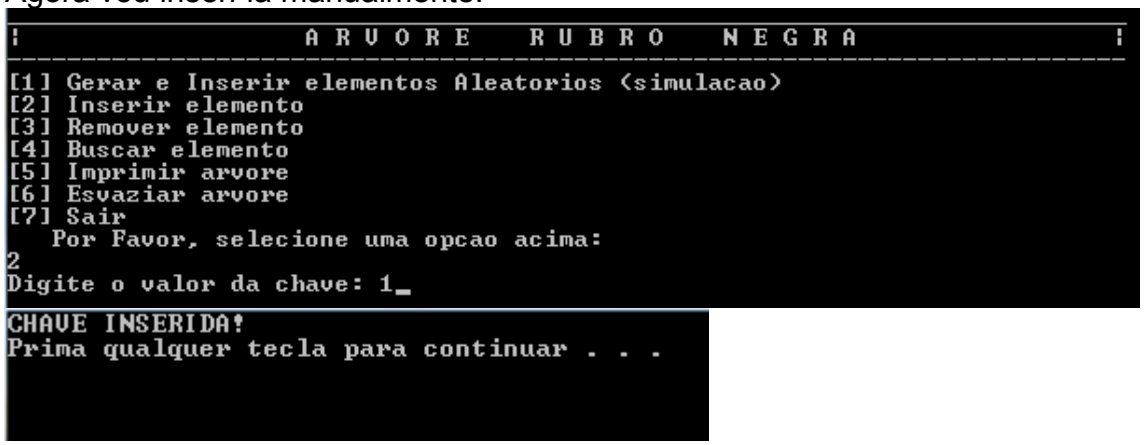


Agora vou reimprimir a chave, só que dessa vez de forma simples e em PosOrder:



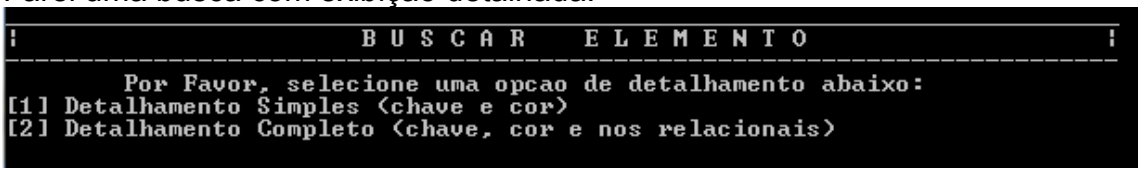
Percebe-se que o nó que possuía a chave de número 1 não está mais na árvore.

Agora vou inseri-la manualmente:

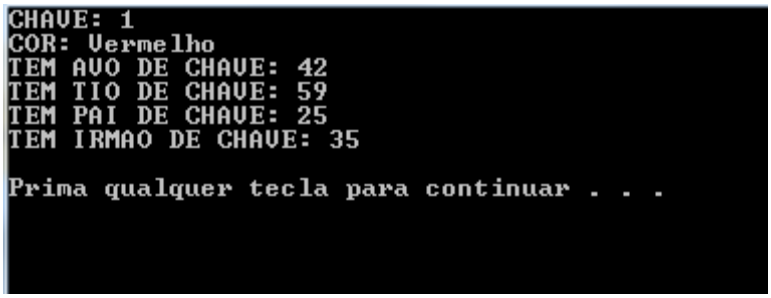


Mas agora, ao invés de imprimir toda a árvore, vou buscar por ela:

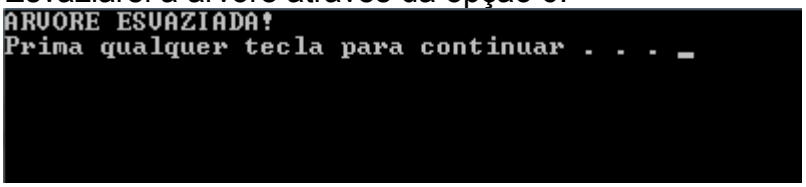
Farei uma busca com exibição detalhada:



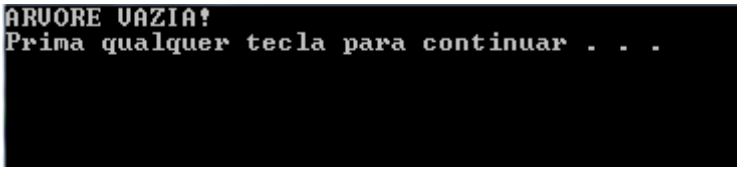
Opção 2



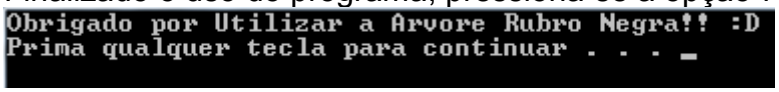
Esvaziarei a árvore através da opção 6:



Agora tentarei (esperando fracasso) imprimir toda a árvore seja qual for a forma de impressão:



Finalizado o uso do programa, pressiona-se a opção 7 do menu inicial:



Assim como há uma saudação, há uma despedida.

CONCLUSÃO

Por fim, em comparação com a árvore AVL, as árvores AVL são mais equilibradas que a árvore rubro negra, mas podem resultar em mais rotações durante a inserção e a remoção.
Portanto, se a aplicação desejada envolver muitas inserções e remoções frequentes, prefere-se o uso da árvore rubro negra. E se as inserções e remoções forem menos frequentes e a buscar for a operação mais frequente, a árvore AVL é preferível em relação à árvore rubro negra.
Com isso, a árvore rubro negra é mais utilizada em bibliotecas.