Relatório de Avaliação

RB tree

CAIO DE LANNA CALIXTO

july 3, 2024

Objetivo

O relatório terá como objetivo explicar as particularidades da árvore rubro-negra, bem como sua implementação, utilidade e eficiência comparada a outros modelos de estrutura de dados. Para isso uma série de testes serão realizados com o algoritmo e dados como a eficiência do tempo de inserção, remoção e busca de nós será medido.

Node struct.

Diferindo do que foi visto em árvores binária de busca, as RB tree (ou Árvore Rubro Negra) tem uma diferente estrutura de nó, pois essa conta com dois parâmetros a mais do que a Árvore binária convencional. Uma variável ‘Color’, que por sua vez pode assumir o valor de duas constantes nomeadas (enum), que pode ser ‘BLACK’ ou ‘RED’. Além de que a estrutura também conta com um ponteiro a mais que a Árvore binária que vimos anteriormente, este que deve apontar para o nó ‘pai’ do nó em questão.

```
enum Color {RED, BLACK};

//Criando a estrutura para a árvore rubro-negra.
typedef struct Node
{
    int iPayload;
    Node* ptrLeft;
    Node* ptrRight;
    Node* ptrParent;

    Color color;
} Node;
```

Assim fica a estrutura.

Criação do nó

Agora que já definimos a ‘struct’ podemos elaborar uma função para a criação de um nó.

```
//Criando um novo nó
Node* createNode(int iValue)
{
    //usando malloc para alocar memória
    Node* tmp = (Node*) malloc(sizeof(Node));

    //codigo de erro caso nao consigo alocar memória.
    if (tmp == nullptr)
    {
        cerr << "Erro em createNode: malloc" << endl;
        exit(1);
    }

    tmp->iPayload = iValue;
    tmp->ptrLeft = nullptr;
    tmp->ptrRight = nullptr;
    tmp->ptrParent=nullptr;

    tmp->color = RED; // sempre começa em RED.

    return tmp;
}
    Color color;
} Node;
```

usando a função malloc(); “pedimos” ao sistema para reservar a quantidade de memória suficiente para alocar nossa estrutura (o nó criado). Quando conseguimos criar esse nó, o fazemos com seus ponteiros nulos e um iPayload que definimos, além de que a cor do nó que criamos é sempre vermelha de início, e assim que o integramos na estrutura esse nó passa a ter a cor ideal para sua posição na árvore. A função é do tipo (Node*) então ao terminar de criar o nó o retornamos.

A partir desse momento precisamos discutir como um nó criado é inserido em uma RBtree , e como isso se difere da árvore binária de busca.

Regras da RBtree

Para discutir como será a inserção do nó precisamos saber antes as regras da Árvore Rubro Negra, dessa forma entenderemos o código de inserção.

- A cada nó é atribuída a cor **RED** ou **BLACK**.
- O nó raiz é sempre **BLACK**.
- Os nós **RED** só podem ter filhos **BLACK**.
- As folhas das árvores são sempre **BLACK**
- Todos os caminhos de um nó até suas folhas descendentes tem sempre o mesmo número de nós **BLACK**.

Bem, todas essas características têm importância direta na eficiência do algoritmo, pois montar uma árvore nesses moldes faz com que ela esteja sempre balanceada, ou seja, sua subárvore à esquerda nunca tem um tamanho muito maior do que a subárvore à direita. Uma árvore balanceada tem vantagens na hora de fazermos uma busca, pois se a altura de cada subárvore de mesmo nível é parecida, então nosso algoritmo tem um tempo de busca O(log2N), sendo N o número de nós da árvore.

Árvore Binária não balanceada.

Pensando em uma árvore binária não balanceada, simplesmente incluímos os nós apenas ‘caminhando’ pela árvore e verificando se o nó atual é maior ou menor

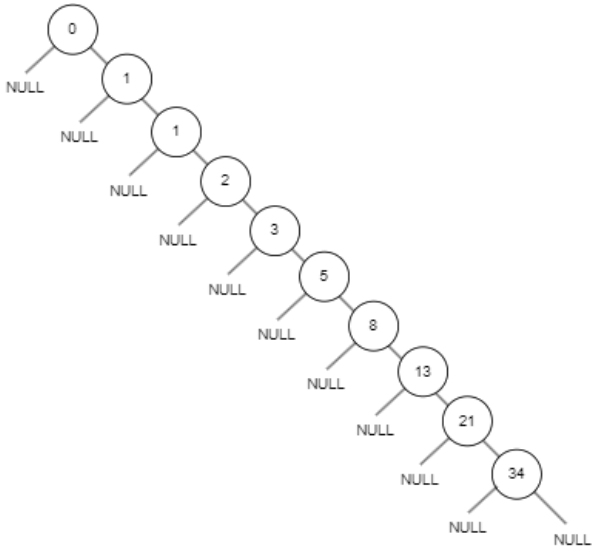
que o nó que vamos inserir, e assim que encontramos o seu lugar exato o inserimos no mesmo. É o caso da função abaixo.

```
Node* insertNode(Node* startingNode, int iData)
{
    if(startingNode == nullptr)
    {
        return createNode(iData);
    }

    if(iData < startingNode->iPayload)
    {
        startingNode->ptrLeft = insertNode(startingNode->ptrLeft, iData);
    }
    else
    {
        startingNode->ptrRight = insertNode(startingNode->ptrRight, iData);
    }

    return startingNode;
}
```

Fazer a inserção dessa maneira pode levar ao desbalanceamento da árvore, por exemplo se fizermos uma inserção crescente como a sequência de Fibonacci por exemplo.

```
int main()
{
    Node* root = nullptr;

    root = insertNode(root, 0);
    root = insertNode(root, 1);
    root = insertNode(root, 1);
    root = insertNode(root, 2);
    root = insertNode(root, 3);
    root = insertNode(root, 5);
    root = insertNode(root, 8);
    root = insertNode(root, 13);
    root = insertNode(root, 21);
    root = insertNode(root, 34);
    return 0;
}
```

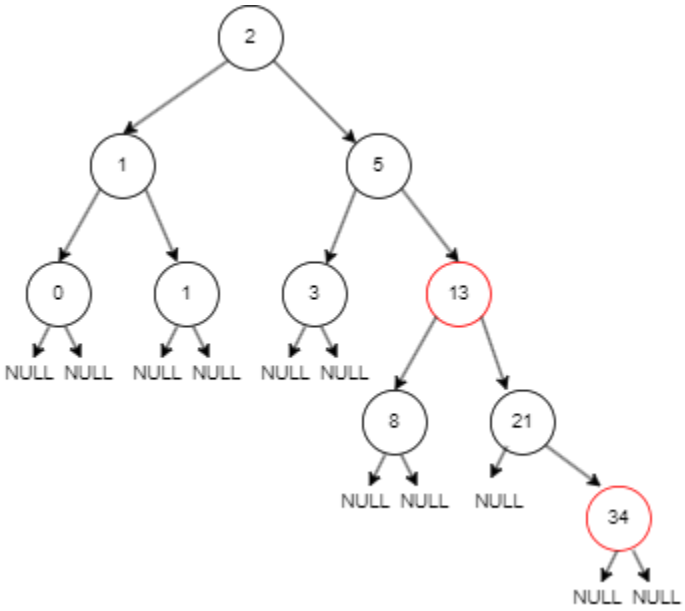
Dessa forma teremos uma árvore que se comporta mais parecido com uma lista encadeada do que com uma árvore, e isso é péssimo pois seu tempo de busca e inserção passa a ser O(N) o que para um N muito grande passa a ser muito maior que O(Log2N). A imagem abaixo representa a árvore gerada.



Ou seja, temos uma árvore binária em que o tempo de busca é tão longo quanto de uma lista. Isso se deve pelo fato de que a cada inserção de nó a ‘altura’ da árvore aumenta. Se continuarmos inserindo termos da sequência de fibonacci nessa árvore ela continuará aumentando e sua altura será igual a N.

RBtree

Agora, como a RBtree tem sistema de balanceamento, a mesma inserção dos termos de fibonacci daria um resultado bem diferente que a anterior. Dessa forma fazer buscas na árvore levaria um tempo O(Log2N) pois agora a altura é aproximadamente Log2N. Segue abaixo a imagem da nova árvore.



Inserção.

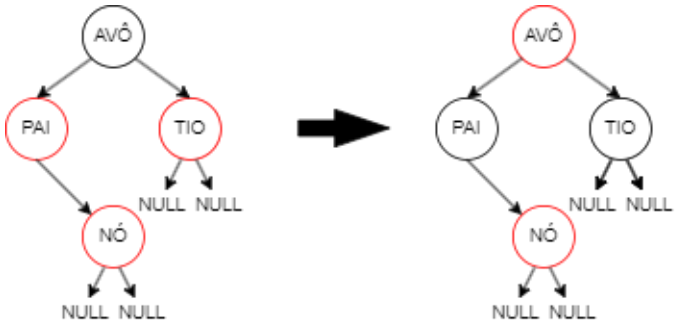
A estratégia utilizada para inserção vai ser primeiro alocar o nó como em uma árvore de busca normal. Porém, após a inserção é possível que uma das regras da árvore rubro-negra citadas no capítulo ‘Regras da RBtree’ sejam quebradas, se esse for o caso só então que corrigimos a isenção. Em nosso código essa função é a `void corrigeInsert(Node* root, Node* newNode);` Essa função de tipo ‘void’ recebe como argumento a raiz da árvore e o nó que deve ser verificado, no caso é o nó que acabou de ser inserido. Porém para entender de fato como “consertar” a árvore é necessário entender o conceito de rotações do nós.

```
Node* insertNode(Node* root, int iData)
{
    Node* newNode = createNode(iData); //cria o no que vai ser inserido.
    if (root == nullptr)
    {
        newNode->color = BLACK; //A raiz é sempre Preta.
        return newNode;
    }

    Node* ParentTemp = nullptr;
    Node* temp = root;

    while(temp != nullptr)
    {
        //dessa forma sempre salvamos o ponteiro anterior a temp como ParentTemp.
        ParentTemp = temp;
        if (iData < temp->iPayload)
        {
            temp = temp->ptrLeft;
        } else {temp = temp->ptrRight;}

    }

    //Agora apontamos pro ponteiro anterior
    newNode->ptrParent = ParentTemp;

    if (iData < ParentTemp->iPayload)
        ParentTemp->ptrLeft = newNode;
    else
        ParentTemp->ptrRight = newNode;


    corrigeInsert(root, newNode);  // FUNÇÃO QUE CORRIGE A INSERÇÃO PARA NAO QUEBRAR AS REGRAS DA
RBtree.
    return root;

}
```

corrrigeInser();

➔ PRIMEIRO CASO

Se o pai e o tio do nó inserido forem **RED**, mudamos sua cor para **BLACK**, e então mudamos a cor do avô do no para **RED**. e então verificamos para o avô do nó se as regras da RBtree foram quebradas, isso impede que exista o caso em que um nó vermelho tem um filho vermelho.



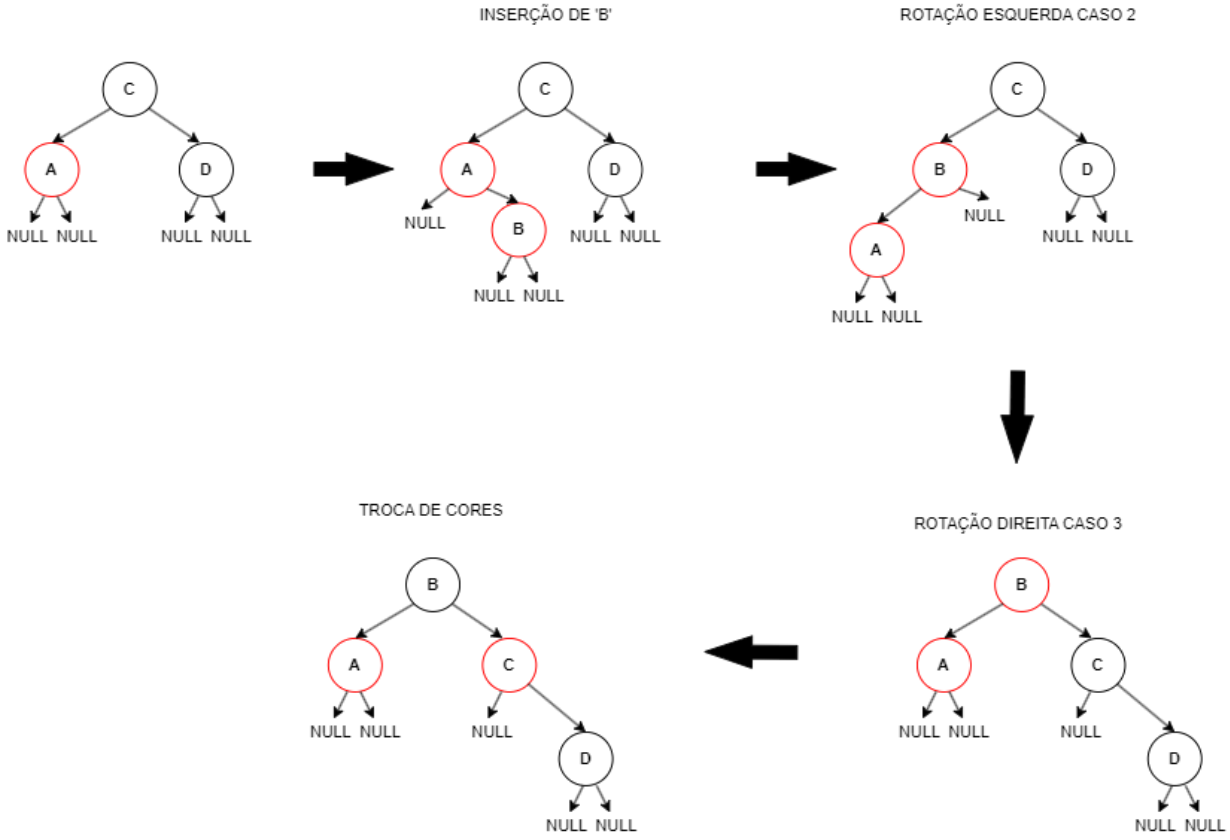
➔ SEGUNDO CASO

Se o nó inserido é filho **ESQUERDO** de um pai **DIREITO**, ou filho **DIREITO** de um pai **ESQUERDO**, então deve haver uma rotação.

➔ TERCEIRO CASO

Se o nó inserido é filho **ESQUERDO** de um pai **ESQUERDO**, ou filho **DIREITO** de um pai **DIREITO**, então deve haver uma rotação.

Abaixo segue imagem que mostra o algoritmo funcionando em uma árvore reduzida.


INSERÇÃO DE 'B'
ROTAÇÃO ESQUERDA CASO 2
ROTAÇÃO DIREITA CASO 3
TROCA DE CORES
C
A
D
B
NULL

rotateLeft

```
void rotateLeft(Node*& root, Node*& modifiedNode)
{
    Node* modifiedNodeRight = modifiedNode->ptrRight; //Armazena o filho direito do nó
    modifiedNode->ptrRight = modifiedNodeRight->ptrLeft;

    if (modifiedNodeRight->ptrLeft != nullptr)
    {
        modifiedNodeRight->ptrLeft->ptrParent = modifiedNode;
    }

    modifiedNodeRight->ptrParent = modifiedNode->ptrParent;

    //ve se o nó que estamos modificando era a raiz, ou seja ptrParent == nullptr;
    if (modifiedNode->ptrParent == nullptr)
    {
        root = modifiedNodeRight;
    } else if (modifiedNode == modifiedNode->ptrParent->ptrLeft) //Vê se ele era o filho esquerdo do pai
    {
        modifiedNode->ptrParent->ptrLeft = modifiedNodeRight;
    } else
    {
        modifiedNode->ptrParent->ptrRight = modifiedNodeRight;
    }

    modifiedNodeRight->ptrLeft = modifiedNode;
    modifiedNode->ptrParent = modifiedNodeRight;
}
```

rotateRight

```
void rotateRight(Node*& root, Node*& modifiedNode)
{
    Node* modifiedNodeLeft = modifiedNode->ptrLeft; // Armazena o filho esquerdo de modifiedNode.
    modifiedNode->ptrLeft = modifiedNodeLeft->ptrRight;

    if (modifiedNodeLeft->ptrRight != nullptr)
    {
        modifiedNodeLeft->ptrRight->ptrParent = modifiedNode;
    }

    modifiedNodeLeft->ptrParent = modifiedNode->ptrParent;

    //ve se o nó que estamos modificando era a raiz, ou seja ptrParent == nullptr;
    if (modifiedNode->ptrParent == nullptr)
    {
        root = modifiedNodeLeft;
    } else if (modifiedNode == modifiedNode->ptrParent->ptrLeft) //se modifiedNode era o filho esquerdo
atualiza o ponteiro esquerdo de seu pai.
    {
        modifiedNode->ptrParent->ptrLeft = modifiedNodeLeft;
    } else
    {
        modifiedNode->ptrParent->ptrRight = modifiedNodeLeft;
    }

    modifiedNodeLeft->ptrRight = modifiedNode;
    modifiedNode->ptrParent = modifiedNodeLeft;
}
```

Esses são os códigos responsáveis pelas rotações

OUTRAS FUNÇÕES

As seguintes funções foram implementadas de forma recursiva, como vimos em sala de aula, elas também servem para árvores binárias não são RBtree. A forma como elas foram feitas permite que chamamos a função dentro dela mesma e então começa um looping onde um problema maior é subdividido em problemas menores com passos iguais.

traverseInOrder

```
//travessia inorder com recursão.
void traverseInOrder(Node* ptrStartingNode)
{
    if (ptrStartingNode != nullptr)
    {
        traverseInOrder(ptrStartingNode->ptrLeft);
        cout << " " << ptrStartingNode->iPayload;
        traverseInOrder(ptrStartingNode->ptrRight);
    }
}
```

lesserLeaf

```cpp
//Encontrar o nó mínimo.
Node* lesserLeaf(Node* startingNode)
{
    Node* ptrCurrent = startingNode;

    //Caminha na árvore indo pra esquerda
    while (ptrCurrent && ptrCurrent->ptrLeft != nullptr) ptrCurrent = ptrCurrent->ptrLeft;

    cout << "O menor nó é : " << ptrCurrent->iPayload << endl;
    return ptrCurrent;
}
```

biggerLeaf

```cpp
//Encontrar o nó maximo.
Node* biggerLeaf(Node* startingNode)
{
    Node* ptrCurrent = startingNode;

    //Caminha na árvore indo pra direita.
    while (ptrCurrent && ptrCurrent->ptrRight != nullptr) ptrCurrent = ptrCurrent->ptrRight;

    cout << "O maior nó é : " << ptrCurrent->iPayload << endl;
    return ptrCurrent;
}
```

treeHeight

```cpp
//Econtrar altura de forma recursiva.
int treeHeight(Node* startingNode)
{
    if (startingNode == nullptr) return 0;
    else
    {
        int iLeftHeight = treeHeight(startingNode->ptrLeft);
        int iRightHeight = treeHeight(startingNode->ptrRight);

        return max(iLeftHeight, iRightHeight) + 1;
    }
}
```

searchNode

```
//Encontrar um nó específico de forma recursiva
Node* searchNode(Node* startingNode, int iData)
{
    if(startingNode == nullptr) return nullptr;
    else if(iData == startingNode->iPayload) return startingNode;
    else if(iData < startingNode->iPayload) return searchNode(startingNode->ptrLeft, iData);
    else return searchNode(startingNode->ptrRight, iData);
}
```

Testes.

Vamos fazer um teste usando novamente uma função não decrescente. Porém, dessa vez vamos cronometrar o tempo de inserção e busca de i-ésimo termos de da função para um ‘i’ relativamente grande. vamos também cronometrar o mesmo para uma árvore binária que não se preocupa com balanceamento.

➜ Primeiro veremos o código para a inserção desbalanceada.

```
Node* insertNodeDESBALANCEADO(Node* startingNode, int iData)
{
    if(startingNode == nullptr)
    {
        return createNode(iData);
    }

    if(iData < startingNode->iPayload)
    {
        startingNode->ptrLeft = insertNodeDESBALANCEADO(startingNode->ptrLeft, iData);
    }
    else
    {
        startingNode->ptrRight = insertNodeDESBALANCEADO(startingNode->ptrRight, iData);
    }

    return startingNode;
}
```

➜ Agora usaremos essa função para adicionar um número elevado de nós e medir o tempo de inserção e busca. Além de que também vamos observar a altura da árvore, o maior nó e o menor.

```cpp
//Exemplo 1, com insertNodeDESBALANCEADO
    Node* root = nullptr;

    //utilizanto a biblioteca chrono para medir tempo.
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100000; i++)
    {
        root = insertNodeDESBALANCEADO(root, i);
    }

    cout << "================================================================" << endl;

    auto end = std::chrono::high_resolution_clock::now() - start;
    long long resultadoNano = std::chrono::duration_cast<std::chrono::nanoseconds>(end).count();
    cout << "tempo de inserção sem balanceamento: " << resultadoNano << endl;

    //utilizanto a biblioteca chrono para medir tempo.
    auto start_2 = std::chrono::high_resolution_clock::now();

    Node* Testing_searchNode = searchNode(root, 99999);
    cout << "endereço do nó procurado pela função: " << Testing_searchNode << endl;

    auto end_2 = std::chrono::high_resolution_clock::now() - start_2;
    long long resultadoNano_2 = std::chrono::duration_cast<std::chrono::nanoseconds>(end_2).count();
    cout << "tempo de busca sem balanceamento: " << resultadoNano_2 << endl;

    int AlturaArvore = treeHeight(root);
    cout << "A altura ad árvore é: " << AlturaArvore << endl;

    cout << "================================================================" << endl;
```

```
================================================================
tempo de inserção sem balanceamento: 55074819908
endereço do nó procurado pela função: 0x555ef14d3080
tempo de busca sem balanceamento: 1099104
A altura ad árvore é: 100000
================================================================
```

É claro que dessa forma estamos evidenciando o pior dos casos da inserção, pois não há ganho de eficiência algum se comparado com uma linked-list por exemplo, porém esse caso nos mostra como é importante balancear a árvore para fazer buscas

➜ Agora usaremos o código da inserção balanceada e veremos que tanto a altura da árvore (e consequentemente o tempo de busca e inserção) diminuem exponencialmente, passando a ser O(log2N) ao invés de O(N).

```
Node* root_2 = nullptr;

    //utilizanto a biblioteca chrono para medir tempo.
    auto start_3 = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100000; i++)
    {
        root_2 = insertNode(root_2, i);
    }

    cout << "================================================================" << endl;

    auto end_3 = std::chrono::high_resolution_clock::now() - start_3;
    long long resultadoNano_3 = std::chrono::duration_cast<std::chrono::nanoseconds>(end_3).count();
    cout << "tempo de inserção com balanceamento: " << resultadoNano_3 << endl;

    //utilizanto a biblioteca chrono para medir tempo.
    auto start_4 = std::chrono::high_resolution_clock::now();

    Node* Testing_searchNode_2 = searchNode(root_2, 99999);
    cout << "endereço do nó procurado pela função: " << Testing_searchNode_2 << endl;

    auto end_4 = std::chrono::high_resolution_clock::now() - start_4;
    long long resultadoNano_4 = std::chrono::duration_cast<std::chrono::nanoseconds>(end_4).count();
    cout << "tempo de busca com balanceamento: " << resultadoNano_4 << endl;

    int AlturaArvore_2 = treeHeight(root_2);
    cout << "A altura ad árvore é: " << AlturaArvore_2 << endl;

    cout << "================================================================" << endl;
```

```
================================================================
tempo de inserção com balanceamento: 29365681
endereço do nó procurado pela função: 0x55d4293dc080
tempo de busca com balanceamento: 5414
A altura ad árvore é: 22
================================================================
```

Como podemos observar a altura da árvore é 22, então se formos adicionar um novo nó ou então buscar um nó, vamos precisar de no máximo 22 iterações para chegar até as folhas da árvore. Esse é o poder de uma árvore balanceada se comparada a uma sem balanceamento.

Teste da função remote

```
    cout << "================================================================" << endl;

    Node* root_3 = nullptr;

    root_3 = insertNode(root_3, 40);
    root_3 = insertNode(root_3, 10);
    root_3 = insertNode(root_3, 0);
    root_3 = insertNode(root_3, 5);
    root_3 = insertNode(root_3, 3);
    root_3 = insertNode(root_3, 4);
    root_3 = insertNode(root_3, 1);


    cout << "traverseInOrder antes de remover o no 40" << endl;
    traverseInOrder(root_3);
    cout << endl;
    removeNode(root_3, 40);

    cout << "traverseInOrder depois de remover o no 40" << endl;
    traverseInOrder(root_3);
    cout << endl;
    cout << "================================================================" << endl;
    return 0;
```

```
================================================================
traverseInOrder antes de remover o no 40
 0 1 3 4 5 10 40
traverseInOrder depois de remover o no 40
 0 1 3 4 5 10
================================================================
```