FGV EMAp
ESCOLA DE MATEMÁTICA APLICADA

RELATÓRIO DE AVALIAÇÃO SUBSTITUTIVA

Natália Brandão de Sousa

Rio de Janeiro
2024

Natália Brandão de Sousa

RELATÓRIO DE AVALIAÇÃO SUBSTITUTIVA

Este relatório tem como objetivo detalhar a implementação de árvores rubro-negras em C++, bem como os testes realizados e uma análise crítica da solução.

. Professor: Rafael de Pinho

Rio de Janeiro
2024

Sumário

1 Introdução **1**
1.1 Árvore Rubro-Negra 1
1.2 Lógica da inserção e remoção 1

2 Funções requisitadas **2**
2.1 Insert 2
2.2 Remove 3
2.3 Search 5
2.4 Inorder 7
2.5 isValid Tree RB 8
2.6 findMax 9
2.7 findMin 10
2.8 height 11

3 Funções auxiliares **12**
3.1 createTree 12
3.2 createNode 13
3.3 blackHeight 14
3.4 rotateLeft 15
3.5 rotateRight 17
3.6 insertFixup 19
3.7 printTreeRB 20
3.8 transplant 22
3.9 removeFixup 23
3.10 verifyBlackNodes 25
3.11 verifyRedNodes 26

4 Resultado de Testes **27**

5 Conclusão: **29**
5.1 Pontos críticos 29
5.2 Vantagens e desvantagens 29

1 Introdução

1.1 Árvore Rubro-Negra

Uma árvore rubro-negra é uma árvore binária de busca balanceada com propriedades adicionais que garantem que o caminho mais longo da raiz até uma folha não seja mais do que o dobro do caminho mais curto, assegurando uma altura $O(\log n)$.[2] As propriedades da árvore rubro-negra são:

- Cada nó é vermelho ou preto.
- A raiz é preta.
- Todas as folhas (NIL) são pretas.
- Se um nó é vermelho, seus filhos são pretos.
- Para cada nó, todos os caminhos simples do nó até suas folhas descendentes contêm o mesmo número de nós pretos.

Estas propriedades garantem que a árvore se mantenha balanceada, assegurando um tempo de operação $O(\log n)$ para inserção, remoção e busca. [1]

1.2 Lógica da inserção e remoção

Inserção: Primeiro, é necessário buscar a posição correta para inserir o novo nó. A busca começa pela raiz da árvore e, a cada comparação, o algoritmo se move para a subárvore esquerda se o valor do novo nó for menor, ou para a subárvore direita se for maior. Esse processo se repete até encontrar um nó folha (null), onde o novo nó será inserido.

Após encontrar a posição correta, o novo nó é criado com a cor vermelha e inserido na árvore na posição determinada (esquerda ou direita do nó folha). Em seguida, a árvore precisa ser corrigida para manter suas propriedades.

Se o novo nó for a raiz, ele é pintado de preto. Se o pai do novo nó for preto, a inserção está concluída, pois não há violações das propriedades. No entanto, se o pai do novo nó for vermelho, ocorrerão violações que precisam ser corrigidas.

Há três casos principais para corrigir a árvore. No Caso 1, se o tio do novo nó também for vermelho, tanto o pai quanto o tio são pintados de preto, e o avô é pintado de vermelho. O ponteiro é então movido para o avô, e o processo se repete. No Caso 2, se o novo nó e seu pai formarem um triângulo, é feita uma rotação que transforma o triângulo em uma linha. No Caso 3, quando o novo nó e seu pai formam uma linha, é realizada uma rotação no avô, trocando as cores do pai e do avô, e o novo nó é pintado de preto.

Remoção: Primeiramente, busca-se o nó a ser removido, começando pela raiz e seguindo a lógica de uma árvore binária de busca, movendo-se para a esquerda para valores menores e para a direita para valores maiores.

Após encontrar o nó a ser removido, se ele tiver menos de dois filhos (nenhum ou um filho), é substituído por seu filho (ou por nulo, se não tiver filhos). Se o nó

tiver dois filhos, encontra-se o menor nó na subárvore direita (ou o maior na subárvore esquerda), substitui-se o valor do nó a ser removido pelo valor desse sucessor e remove-se o sucessor.

A correção da árvore é necessária se o nó removido ou o sucessor substituído era preto, pois pode ocorrer uma violação das propriedades da árvore rubro-negra. Existem quatro casos principais a serem considerados para corrigir a árvore. No Caso 1, se o irmão do nó removido for vermelho, o irmão é pintado de preto e o pai de vermelho, seguido de uma rotação que transforma o irmão em preto. No Caso 2, se os filhos do irmão forem pretos, o irmão é pintado de vermelho e o ponteiro é movido para o pai. No Caso 3, se o filho esquerdo do irmão for vermelho e o direito for preto, o filho esquerdo do irmão é pintado de preto e o irmão de vermelho, seguido de uma rotação no irmão. No Caso 4, se o filho direito do irmão for vermelho, o irmão é pintado com a cor do pai, o pai de preto e o filho direito do irmão de preto, seguido de uma rotação no pai. Após esses ajustes, o ponteiro é agora a raiz.

2 Funções requisitadas

Nesta seção irei detalhar a implementação das funções principais, conforme foi requisitada na avaliação substitutiva.

2.1 Insert

O objetivo da função insert é inserir um novo nó na árvore rubro-negra.

```
void insert(Tree_RB* ptrTree, int iValue)
```

Parâmetros:

- `ptrTree`: Ponteiro para a estrutura `Tree_RB`, que representa a árvore rubro-negra.
- `iValue`: O valor inteiro a ser inserido na árvore.

Verificação da raiz:

```
if (ptrTree->ptrRoot == nullptr)
{
    ptrTree->ptrRoot = createNode(iValue, BLACK);
    return;
}
```

Se a raiz da árvore (ptrRoot) é nula, a árvore está vazia. Daí cria o primeiro nó da árvore com o valor iValue e com a cor preta. Por fim, retorna.

Percorrendo a árvore para encontrar a posição de inserção:

```
Node_RB* ptrCurrent = ptrTree->ptrRoot;
Node_RB* ptrParent = nullptr;
```

```
while (ptrCurrent != nullptr)
{
    ptrParent = ptrCurrent;

    if (iValue < ptrCurrent->iValue)
    {
        ptrCurrent = ptrCurrent->ptrLeft;
    }
    else
    {
        ptrCurrent = ptrCurrent->ptrRight;
    }
}
```

Aqui, ptrCurrent é inicializado como a raiz da árvore (ptrRoot). Ele será usado para percorrer a árvore. E ptrParent é inicializado como nullptr. Ele armazenará o nó pai do novo nó que será inserido.O loop while continua até ptrCurrent se tornar nullptr. Isso significa que o loop percorre a árvore até encontrar uma posição nula onde o novo nó deve ser inserido. Dentro do loop: ptrParent é atualizado para o valor atual de ptrCurrent. Isso mantém o registro do nó pai do nó onde eventualmente ptrCurrent se tornará nullptr. Depois, há uma comparação dos valores: Se iValue é menor que o valor do nó atual (ptrCurrent->iValue), o ponteiro ptrCurrent é movido para o filho esquerdo do nó atual (ptrCurrent->ptrLeft). Caso contrário (se iValue é maior ou igual ao valor do nó atual), o ponteiro ptrCurrent é movido para o filho direito do nó atual (ptrCurrent->ptrRight).

Ajuste da árvore:

```
insertFixup(ptrTree, ptrNew);
```

Chama a função insertFixup para ajustar a árvore, garantindo que todas as propriedades da árvore rubro-negra sejam mantidas após a inserção. (Tal função será melhor explicada na seção de funções auxiliares)

2.2 Remove

O objetivo da função remove é remover um nó da árvore rubro-negra.

```
void remove(Tree_RB* ptrTree, int iValue)
```

Parâmetros:

- `ptrTree`: Ponteiro para a estrutura `Tree_RB`, que representa a árvore rubro-negra.
- `iValue`: O valor inteiro do nó a ser removido da árvore.

Busca do nó a ser removido:

```
void removeNode_RB* ptrRemove = search(ptrTree, iValue);
if (ptrTree->ptrRoot == nullptr || ptrRemove == nullptr) { return; }
```

Utiliza a função search para encontrar o nó a ser removido. Se a árvore está vazia ou o nó não é encontrado, retorna.

Armazenamento da cor original e inicialização do ponteiro do filho:

```
bool bOriginalColor = ptrRemove->bColor;
Node_RB* ptrChild = nullptr;
```

Aqui, bOriginalColor é uma variável booleana que armazena a cor original do nó que será removido (ptrRemove). Durante o processo de remoção em uma árvore rubro-negra, se o nó removido for preto, pode ser necessário fazer ajustes na árvore para manter as propriedades da árvore rubro-negra. Daí, ptrChild é inicializado como nullptr e será usado para representar o filho do nó que está sendo removido (ptrRemove). Durante a remoção, ptrChild será ajustado dependendo dos casos de remoção.

Determinação do caso de remoção:

```
bool bOriginalColor = ptrRemove->bColor;
Node_RB* ptrChild = nullptr;
```

- **Caso 1:** Filho à esquerda é nulo:

```
if (ptrRemove->ptrLeft == nullptr)
{
    ptrChild = ptrRemove->ptrRight;
    transplant(ptrTree, ptrRemove, ptrChild);
}
```

Define ptrChild como o filho direito do nó a ser removido. E chama a função transplant para substituir o nó removido por ptrChild.

- **Caso 2:** Filho à direita é nulo:

```
else if (ptrRemove->ptrRight == nullptr)
{
    ptrChild = ptrRemove->ptrLeft;
    transplant(ptrTree, ptrRemove, ptrChild);
}
```

Define ptrChild como o filho esquerdo do nó a ser removido.E chama a função transplant para substituir o nó removido por ptrChild.

- **Caso 3:** Ambos os filhos não são nulos:

```
else
{
    Node_RB* ptrSubMin = findMin(ptrRemove->ptrRight);
    bOriginalColor = ptrSubMin->bColor;
    ptrChild = ptrSubMin->ptrRight;

    if (ptrSubMin->ptrParent == ptrRemove)
    {
        ptrChild->ptrParent = ptrSubMin;
    }
    else
    {
        transplant(ptrTree, ptrSubMin, ptrChild);
        ptrSubMin->ptrRight = ptrRemove->ptrRight;
        ptrRemove->ptrRight->ptrParent = ptrSubMin;
    }

    transplant(ptrTree, ptrRemove, ptrSubMin);
    ptrSubMin->ptrLeft = ptrRemove->ptrLeft;
    ptrRemove->ptrLeft->ptrParent = ptrSubMin;
    ptrSubMin->bColor = ptrRemove->bColor;
}
```

Aqui, encontra o sucessor do nó a ser removido ptrSubMin. Armazena a cor original do sucessor e define ptrChild como o filho direito do sucessor. Se o sucessor não é filho direto do nó a ser removido, realiza a substituição com transplant e ajusta os ponteiros.Depois, substitui o nó a ser removido pelo sucessor e ajusta os ponteiros do sucessor. Por fim, transfere a cor do nó removido para o sucessor.

- Deleção do nó removido:

```
delete ptrRemove; // Deleta efetivamente o nó
```

- Ajuste da árvore:

```
if (bOriginalColor == BLACK) {
    removeFixup(ptrTree, ptrChild);
}
```

Se a cor original do nó removido era preta, chama a função removeFixup para ajustar a árvore.

2.3 Search

O objetivo da função search é por procurar um valor específico em uma árvore rubro-negra.

```
Node_RB* search(Tree_RB* ptrTree, int value)
```

Parâmetros:

- `ptrTree`: É um ponteiro para a estrutura `Tree_RB`, que contém a raiz da árvore rubro-negra que queremos pesquisar.
- `value`: É o valor inteiro que estamos procurando dentro da árvore.

```
Node_RB* ptrCurrent = ptrTree->ptrRoot;
```

Aqui, ptrCurrent é inicializado com o ponteiro para a raiz da árvore . Isso permite que comecemos a busca a partir da raiz da árvore rubro-negra.

Loop de Busca:

```
    while (ptrCurrent != nullptr)
    {
        if (value == ptrCurrent->iValue)
        {
            return ptrCurrent; // Nó encontrado
        }
        else if (value < ptrCurrent->iValue) {
            // Procura na subárvore esquerda
            ptrCurrent = ptrCurrent->ptrLeft;
        } else {
            // Procura na subárvore direita
            ptrCurrent = ptrCurrent->ptrRight;
        }
    }
```

Bem, aqui o loop while continua enquanto ptrCurrent não for nullptr, o que significa que ainda há nós para explorar na árvore. Dentro do loop, a gente verifica se value é igual a ptrCurrent->iValue. Se forem iguais, encontramos o nó que contém o valor procurado, então retornamos ptrCurrent. Se value for menor que ptrCurrent->iValue, atualizamos ptrCurrent para ptrCurrent->ptrLeft, movendo para a subárvore esquerda. E se value for maior que ptrCurrent->iValue, atualizamos ptrCurrent para ptrCurrent->ptrRight, movendo para a subárvore direita.

Condição de Saída do Loop:

```
return nullptr; // Nó não encontrado
```

Se o loop terminar sem encontrar o valor (ptrCurrent se torna `nullptr`), significa que o valor não está presente na árvore. Daí, retornamos nullptr.

2.4 Inorder

O objetivo da função é percorrer uma árvore rubro-negra em ordem, ou seja, em uma sequência ordenada de valores.

```
void inorder(Node_RB* ptrNode)
```

Parâmetros:

- **ptrNode:** Ponteiro para o nó atual da árvore.

Verificação de Nulidade:

```
if (ptrNode == nullptr) { return; }
```

Verifica se o nó atual é nulo. Se for, retorna imediatamente, pois não há nós para processar.

Percorrendo a Subárvore Esquerda:

```
inorder(ptrNode->ptrLeft);
```

Chama recursivamente a função inorder para percorrer a subárvore esquerda do nó atual. Isso garante que todos os nós da subárvore esquerda sejam visitados antes do próprio nó atual.

Impressão do Valor do Nó:

```
if (ptrNode->bColor == RED) {
    cout << red << ptrNode->iValue << reset << " ";
} else {
    cout << ptrNode->iValue << " ";
}
```

Imprime o valor do nó atual. Se o nó for vermelho , o valor é impresso em vermelho.

Percorrendo a Subárvore Direita:

```
inorder(ptrNode->ptrRight);
```

Chama recursivamente a função inorder para percorrer a subárvore direita do nó atual. Isso garante que todos os nós da subárvore direita sejam visitados após o próprio nó atual e sua subárvore esquerda.

2.5 isValid Tree RB

O objetivo da função isValid Tree RB é verificar se uma árvore rubro-negra satisfaz todas as propriedades fundamentais dessa estrutura de dados.

Parâmetros:

- **ptrTree:** Ponteiro para a estrutura que representa a árvore rubro-negra.

Verificação de Nulidade da Árvore:

```
if(ptrTree == nullptr) return true;
```

Verifica se o ponteiro ptrTree aponta para nullptr, ou seja, se a árvore está vazia. Se estiver vazia, considera-se que uma árvore vazia é válida por definição.

Inicialização do Ponteiro para a Raiz:

```
Node_RB* ptrNode = ptrTree->ptrRoot;
```

Obtém o ponteiro para a raiz da árvore rubro-negra ptrTree.

Verificação da Cor da Raiz:

```
if(ptrNode->bColor != BLACK) {
    cerr << "Violação: a raíz não é preta." << endl;
    return false;
}
```

Verifica se a raiz da árvore não é preta. De acordo com as propriedades da árvore rubro-negra, a raiz deve ser sempre preta.

Verificação dos Nós Vermelhos:

```
if(!verifyRedNodes(ptrNode)) {
    cerr << "Violação: os nós vermelhos têm filhos pretos." << endl;
    return false;
}
```

Chama a função verifyRedNodes para verificar se todos os nós vermelhos da árvore têm apenas filhos pretos. Isso é crucial para garantir que a propriedade de que nenhum nó vermelho pode ter um filho vermelho seja mantida.

Verificação da Contagem de Nós Pretos em um Caminho:

```
int iCountBlack = 0;
Node_RB* temp = ptrNode;
while(temp != nullptr) {
    if (temp->bColor == BLACK) iCountBlack++;
    temp = temp->ptrLeft;
```

```
}
```

Calcula o número de nós pretos em qualquer caminho, percorrendo a subárvore esquerda a partir da raiz até um nó externo.

Verificação do Número de Nós Pretos em Todos os Caminhos:

```
if(!verifyBlackNodes(ptrNode, iCountBlack, 0)) {
    cerr << "Violação: Os caminhos até as folhas
    (NIL) não têm o mesmo número de nós pretos."
    << endl;
    return false;
}
```

Chama a função verifyBlackNodes para verificar se todos os caminhos da raiz até os nós NIL têm o mesmo número de nós pretos. Isso é essencial para garantir a propriedade de que todos os caminhos de uma raiz até as folhas externas têm o mesmo número de nós pretos.

Retorno de Validação:

```
return true;
```

Se todas as verificações acima forem bem-sucedidas, a função retorna true, indicando que a árvore rubro-negra é válida de acordo com todas as suas propriedades.

2.6 findMax

O objtivo da função findMax é encontrar e retornar o nó com o valor máximo na árvore rubro-negra, começando a busca a partir do nó passado como argumento (ptrRoot).

```
Node_RB* findMax(Node_RB* ptrRoot)
```

Parâmetros:

- **Node_RB*** Ponteiro para o nó a partir do qual começará a busca pelo valor máximo na árvore rubro-negra.

Verificação de Nulidade do Nó Raiz:

```
if (ptrRoot == nullptr) { return nullptr; }
```

Verifica se o nó raiz passado como argumento é nulo. Se for, retorna imediatamentenullptr, indicando que a árvore está vazia e não há nenhum valor máximo a ser encontrado.

Inicialização do Ponteiro Atual:

```
Node_RB* ptrCurrent = ptrRoot;
```

Inicializa um ponteiro ptrCurrent apontando para o nó raiz fornecido. Este ponteiro será usado para percorrer a árvore durante a busca pelo valor máximo.

Loop de Busca pelo Valor Máximo:

```
while(ptrCurrent->ptrRight != nullptr)
{
    ptrCurrent = ptrCurrent->ptrRight;
}
```

O loop continua enquanto o ponteiro ptrCurrent tiver um filho direito diferente de nullptr. Isso significa que ainda não alcançamos o nó mais à direita na árvore (o maior valor).

Dentro do loop, o ponteiro ptrCurrent é atualizado para apontar para seu filho direito. Isso é feito repetidamente até que ptrCurrent não tenha mais um filho direito, indicando que encontramos o nó com o valor máximo na árvore rubro-negra.

Retorno do Nó com o Valor Máximo:

```
return ptrCurrent;
```

Após sair do loop, ptrCurrent aponta para o nó que contém o valor máximo na árvore rubro-negra. O ponteiro ptrCurrent é então retornado como resultado da função.

2.7 findMin

O objetivo da função findMin é encontrar e retornar o nó com o valor mínimo na árvore rubro-negra, começando a busca a partir do nó passado como argumento (ptrRoot).

```
Node_RB* findMin(Node_RB* ptrRoot)
```

Parâmetros:

- **ptrRoot:** Ponteiro para o nó a partir do qual começará a busca pelo valor mínimo na árvore rubro-negra.

Verificação de Nulidade do Nó Raiz:

```
if (ptrRoot == nullptr) { return nullptr; }
```

Verifica se o nó raiz passado como argumento é nulo. Se for, retorna imediatamente nullptr, indicando que a árvore está vazia e não há nenhum valor mínimo a ser encontrado.

Inicialização do Ponteiro Atual:

```
Node_RB* ptrCurrent = ptrRoot;
```

Inicializa um ponteiro ptrCurrent apontando para o nó raiz fornecido (ptrRoot). Este ponteiro será usado para percorrer a árvore durante a busca pelo valor mínimo.

Loop de Busca pelo Valor Mínimo:

```
while(ptrCurrent->ptrLeft != nullptr)
{
    ptrCurrent = ptrCurrent->ptrLeft;
}
```

O loop continua enquanto o ponteiro ptrCurrent tiver um filho esquerdo diferente de nullptr. Isso significa que ainda não alcançamos o nó mais à esquerda na árvore (o menor valor).

Dentro do loop, o ponteiro ptrCurrent é atualizado para apontar para seu filho esquerdo . Isso é feito repetidamente até que ptrCurrent não tenha mais um filho esquerdo, indicando que encontramos o nó com o valor mínimo na árvore rubro-negra.

Retorno do Nó com o Valor Mínimo:

```
return ptrCurrent;
```

Após sair do loop, ptrCurrent aponta para o nó que contém o valor mínimo na árvore rubro-negra. O ponteiro ptrCurrent é então retornado como resultado da função.

2.8 height

O objetivo da função height é calcular a altura de um nó na árvore rubro-negra, começando a partir do nó passado como argumento (ptrNode).

```
int height(Node_RB* ptrNode)
```

Parâmetros:

- **ptrNode:** Ponteiro para o nó a partir do qual a altura será calculada na árvore rubro-negra.

Verificação de Nulidade do Nó:

```
if (ptrNode == nullptr) { return -1; }
```

Verifica se o nó passado como argumento (ptrNode) é nulo. Se for, retorna imediatamente -1, indicando que a altura desse nó é -1 .

Cálculo da Altura dos Filhos Esquerdo e Direito:

```
int iLeftHeight = height(ptrNode->ptrLeft);
int iRightHeight = height(ptrNode->ptrRight);
```

Chama recursivamente a função height para calcular a altura dos filhos esquerdo e direito do nó atual . Essas chamadas recursivas irão calcular a altura de cada subárvore.

Determinação da Altura do Nó Atual:

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

Calcula a altura do nó atual somando 1 ao máximo entre as alturas do filho esquerdo e do filho direito . Isso garante que a altura do nó atual leve em consideração a altura de sua subárvore mais alta.

3 Funções auxiliares

Nesta seção irei detalhar a implementação das funções auxiliares, conforme foi requisitada na avaliação substitutiva.

3.1 createTree

O objetivo da função createTree é criar e retornar um ponteiro para uma nova estrutura de árvore rubro-negra.

```
Tree_RB* createTree()
```

Retorna um ponteiro para uma nova estrutura de árvore rubro-negra (Tree_RB).

Alocação de Memória para a Estrutura da Árvore:

```
Tree_RB* ptrTree = new Tree_RB;
```

Usa o operador new para alocar dinamicamente memória para uma nova estrutura de árvore rubro-negra (Tree_RB). Isso cria um novo objeto Tree_RB na memória e retorna um ponteiro para ele.

Retorno do Ponteiro da Árvore Criada:

```
return ptrTree;
```

Retorna o ponteiro recém-alocado que aponta para a estrutura de árvore rubro-negra criada.

3.2 createNode

As funções createNode são responsáveis por criar e inicializar nós para a árvore rubro-negra.

```
Node_RB* createNode(int iValue)
```

Parâmetros:

- **iValue:** Valor inteiro que será armazenado no nó.

Alocação de Memória:

```
Node_RB* ptrNode = new Node_RB;
```

Inicialização dos Campos do Nó:

```
ptrNode->iValue = iValue;
ptrNode->bColor = RED;
```

Define o valor iValue fornecido como argumento para o campo iValue do nó. Daí, Inicializa o campo bColor com vermelho, que é a cor padrão para um novo nó na inserção inicial.

Retorno do Ponteiro do Nó Criado:

```
return ptrNode;
```

Retorna o ponteiro recém-alocado que aponta para o nó criado e inicializado.

A mesma função agora com um parâmetro a mais:

```
Node_RB* createNode(int iValue, Color color)
```

Parâmetros:

- **iValue:** Valor inteiro que será armazenado no nó.
- **color:** Enumeração `Color` que especifica a cor do nó (`RED` ou `BLACK`).

Alocação de Memória:

```
Node_RB* ptrNode = new Node_RB;
```

Similar à primeira função, aloca dinamicamente memória para um novo nó da árvore rubro-negra (Node_RB).

Inicialização dos Campos do Nó:

```
ptrNode->iValue = iValue;
ptrNode->bColor = color;
```

Define o valor iValue fornecido como argumento para o campo iValue do nó. Inicializa o campo bColor com a cor especificada pelo parâmetro color, que pode ser RED ou BLACK.

Retorno do Ponteiro do Nó Criado:

```
return ptrNode;
```

Retorna o ponteiro recém-alocado que aponta para o nó criado e inicializado com a cor especificada.

3.3 blackHeight

O objetivo da função blackHeight é calcular a altura preta de um nó na árvore rubro-negra, que é definida como o número de nós pretos ao longo de qualquer caminho da raiz até uma folha.

```
int blackHeight(Node_RB* ptrNode)
```

Parâmetros:

- **ptrNode:** Ponteiro para o nó atual que está sendo verificado.

Caso Base:

```
int blackHeightif (ptrNode == nullptr) { return 0; }
```

Verifica se o ponteiro ptrNode é nulo. Se for, isso significa que alcançamos uma folha (NIL) na árvore, então a altura preta é 0.

Calcula a Altura Preta para o Filho Esquerdo:

```
int iLeftHeight = blackHeight(ptrNode->ptrLeft);
```

Chama recursivamente a função blackHeight para calcular a altura preta do filho esquerdo do nó atual (ptrNode->ptrLeft).

Calcula a Altura Preta para o Filho Direito:

```
int iRightHeight = blackHeight(ptrNode->ptrRight);
```

Chama recursivamente a função blackHeight para calcular a altura preta do filho direito do nó atual (ptrNode->ptrRight).

Determinação da Altura Preta do Nó Atual:

```
if (ptrNode->bColor == BLACK)
{
    return 1 + max(iLeftHeight, iRightHeight);
}
```

Se o nó atual for preto, a altura preta é calculada como 1 (para o próprio nó preto) mais o máximo entre as alturas pretas dos seus filhos esquerdo e direito.

Retorno :

```
return max(iLeftHeight, iRightHeight);
```

3.4 rotateLeft

O objetivo da função rotateLeft é realizar uma rotação à esquerda em um nó específico de uma árvore rubro-negra, ajustando os ponteiros dos nós envolvidos.

```
Node_RB* rotateLeft(Tree_RB* ptrTree, Node_RB* ptrNode)
```

Parâmetros:

- **ptrTree:** Ponteiro para a estrutura `Tree_RB`, representando a árvore rubro-negra.
- **ptrNode:** Ponteiro para o nó onde a rotação à esquerda deve ser realizada.

Inicialização de Variáveis:

```
Node_RB* ptrGrandparent = ptrNode->ptrParent;
Node_RB* ptrPivot = ptrNode->ptrRight;
Node_RB* ptrChild = nullptr;
```

- **ptrGrandparent:** Ponteiro para o nó pai de `ptrNode`.
- **ptrPivot:** Ponteiro para o nó direito de `ptrNode` (o pivô da rotação).
- **ptrChild:** Inicialmente `nullptr`, este ponteiro será usado para o nó esquerdo de `ptrPivot`.

Verificação do Nó Pivô:

```
if (ptrPivot != nullptr)
{
    ptrChild = ptrPivot->ptrLeft;
```

Verifica se ptrPivot não é nulo. Se for nulo, a rotação não pode ser realizada. Se não for nulo, ptrChild é atualizado para apontar para o nó esquerdo de ptrPivot.

Realização da Rotação:

```
// Realizar a rotação
ptrNode->ptrRight = ptrChild;
ptrPivot->ptrLeft = ptrNode;
```

- `ptrNode->ptrRight` é atualizado para `ptrChild`.
- `ptrPivot->ptrLeft` é atualizado para `ptrNode`.

Atualização dos Ponteiros de Parentesco:

```
    // Atualizar os ponteiros de parentesco
ptrNode->ptrParent = ptrPivot;
ptrPivot->ptrParent = ptrGrandparent;
if (ptrChild != nullptr) {
    ptrChild->ptrParent = ptrNode;
}
```

- `ptrNode->ptrParent` é atualizado para `ptrPivot`.
- `ptrPivot->ptrParent` é atualizado para `ptrGrandparent`.
- Se `ptrChild` não for nulo, `ptrChild->ptrParent` é atualizado para `ptrNode`.

Atualização do Ponteiro do Ancestral:

```
       // Atualizar o ponteiro do ancestral
if (ptrGrandparent != nullptr) {
    if (ptrGrandparent->ptrLeft == ptrNode) {
        ptrGrandparent->ptrLeft = ptrPivot;
    } else {
        ptrGrandparent->ptrRight = ptrPivot;
    }
} else {
    // Caso a subárvore seja a árvore inteira
    ptrTree->ptrRoot = ptrPivot;
}

return ptrPivot;
```

- Se `ptrGrandparent` não for nulo:
  - Se `ptrNode` é o filho esquerdo de `ptrGrandparent`, `ptrGrandparent->ptrLeft` é atualizado para `ptrPivot`.
  - Caso contrário, `ptrGrandparent->ptrRight` é atualizado para `ptrPivot`.
- Se `ptrGrandparent` for nulo (indicando que `ptrNode` era a raiz da árvore):
  - `ptrTree->ptrRoot` é atualizado para `ptrPivot`.

Retorno

```
        // Retorno de um ponteiro nulo, caso a rotação
        não seja possível
return nullptr;
```

Se ptrPivot é nulo, a rotação não pode ser realizada e a função retorna nullptr.

3.5 rotateRight

O objetivo da função rotateRight é realizar uma rotação à direita em um nó específico de uma árvore rubro-negra, ajustando os ponteiros dos nós envolvidos.

```
Node_RB* rotateRight(Tree_RB* ptrTree, Node_RB* ptrNode)
```

Parâmetros:

- **ptrTree**: Ponteiro para a estrutura `Tree_RB`, representando a árvore rubro-negra.
- **ptrNode**: Ponteiro para o nó onde a rotação à direita deve ser realizada.

Inicialização de Variáveis:

```
Node_RB* ptrGrandparent = ptrNode->ptrParent;
Node_RB* ptrPivot = ptrNode->ptrLeft;
Node_RB* ptrChild = nullptr;
```

- **ptrGrandparent**: Ponteiro para o nó pai de `ptrNode`.
- **ptrPivot**: Ponteiro para o nó esquerdo de `ptrNode` (o pivô da rotação).
- **ptrChild**: Inicialmente `nullptr`, este ponteiro será usado para o nó direito de `ptrPivot`.

Verificação do Nó Pivô:

```
if (ptrPivot != nullptr)
{
    ptrChild = ptrPivot->ptrRight;
```

Verifica se ptrPivot não é nulo. Se for nulo, a rotação não pode ser realizada. Se não for nulo, ptrChild é atualizado para apontar para o nó direito de ptrPivot.

Realização da Rotação:

```
    // Realizar a rotação
    ptrNode->ptrLeft = ptrChild;
    ptrPivot->ptrRight = ptrNode;
```

ptrNode->ptrLeft é atualizado para ptrChild. ptrPivot->ptrRight é atualizado para ptrNode.

Atualização dos Ponteiros de Parentesco:

```
// Atualizar os ponteiros de parentesco
ptrNode->ptrParent = ptrPivot;
ptrPivot->ptrParent = ptrGrandparent;
if (ptrChild != nullptr) {
    ptrChild->ptrParent = ptrNode;
}
```

Aqui, ptrNode->ptrParent é atualizado para ptrPivot, ptrPivot->ptrParent é atualizado para ptrGrandparent. Se ptrChild não for nulo, ptrChild->ptrParent é atualizado paraptrNode.

Atualização do Ponteiro do Ancestral:

```
// Atualizar o ponteiro do ancestral
if (ptrGrandparent != nullptr) {
    if (ptrGrandparent->ptrLeft == ptrNode) {
        ptrGrandparent->ptrLeft = ptrPivot;
    } else {
        ptrGrandparent->ptrRight = ptrPivot;
    }
} else {
    // Caso a subárvore seja a árvore inteira
    ptrTree->ptrRoot = ptrPivot;
}

return ptrPivot;
```

- Se `ptrGrandparent` não for nulo:
  - Se `ptrNode` é o filho esquerdo de `ptrGrandparent`, `ptrGrandparent->ptrLeft` é atualizado para `ptrPivot`.
  - Caso contrário, `ptrGrandparent->ptrRight` é atualizado para `ptrPivot`.
- Se `ptrGrandparent` for nulo (indicando que `ptrNode` era a raiz da árvore): `ptrTree->ptrRoot` é atualizado para `ptrPivot`.

Retorno

```
// Retorno de um ponteiro nulo, caso a rotação não seja possível
return nullptr;
```

Se ptrPivot é nulo, a rotação não pode ser realizada e a função retorna nullptr.

3.6 insertFixup

O objetivo da função insertFixup é corrigir quaisquer violações das propriedades da árvore rubro-negra que possam ocorrer após a inserção de um novo nó na árvore. Ela é chamada após a inserção de um novo nó (ptrNew) e garante que a estrutura da árvore mantenha suas propriedades, principalmente a propriedade de cores e a estrutura balanceada da árvore.

```
void insertFixup(Tree_RB* ptrTree, Node_RB* ptrNew)
```

Parâmetros:

- `ptrTree`: Ponteiro para a estrutura `Tree_RB`, representando a árvore rubro-negra.
- `ptrNew`: Ponteiro para o nó recém-inserido na árvore, cujas propriedades estão sendo ajustadas.

Laço Principal :

```
while (ptrNew != ptrTree->ptrRoot && ptrNew->ptrParent->bColor == RED)
```

Este laço executa enquanto o nó ptrNew não é a raiz da árvore e seu pai (ptrNew->ptrParent) é vermelho. Isso indica que há uma possível violação das propriedades da árvore que precisa ser corrigida.

Verificação do Lado do Pai :

```
if (ptrNew->ptrParent == ptrNew->ptrParent->ptrParent->ptrLeft)
```

Verifica se o pai de ptrNew é o filho esquerdo de seu avô (ptrNew->ptrParent->ptrParent). Isso determina em qual lado do avô o novo nó foi inserido na árvore.

Caso A1: Tio é Vermelho (ptrUncle != nullptr ptrUncle->bColor == RED):

```
ptrNew->ptrParent->bColor = BLACK;
ptrUncle->bColor = BLACK;
ptrNew->ptrParent->ptrParent->bColor = RED;
ptrNew = ptrNew->ptrParent->ptrParent;
```

Se o tio de ptrNew (irmão do seu pai) é vermelho, realiza uma rotação de cores para corrigir a árvore: O pai de ptrNew e o tio são coloridos de preto, o avô de ptrNew é colorido de vermelho. Daí ptrNew é movido para o avô, para verificar se mais correções são necessárias.

Caso A2: Tio é Preto e ptrNew é um Filho Direito:

```
ptrNew = ptrNew->ptrParent;
rotateLeft(ptrTree, ptrNew);
```

Se o tio é preto e ptrNew é o filho direito, realiza uma rotação à esquerda com relação ao pai de ptrNew para alinhar ptrNew como filho esquerdo: Realiza uma rotação à esquerda com relação ao pai de ptrNew.

Caso A3: Tio é Preto e ptrNew é um Filho Esquerdo:

```
ptrNew->ptrParent->bColor = BLACK;
ptrNew->ptrParent->ptrParent->bColor = RED;
rotateRight(ptrTree, ptrNew->ptrParent->ptrParent);
```

Se o tio é preto e ptrNew é o filho esquerdo, ajusta as cores do pai de ptrNew e do avô, e realiza uma rotação à direita com relação ao avô para manter a árvore balanceada: Ajustando as cores do pai de ptrNew para preto e do avô para vermelho. Daí, realiza uma rotação à direita com relação ao avô.

Caso Simétrico (else):

```
else
{
    // Código simétrico ao anterior, trocando
    esquerda por direita e vice-versa
}
```

Se o tio é preto e ptrNew é o filho direito, realiza uma rotação à esquerda com relação ao pai de ptrNew para alinhar ptrNew como filho esquerdo.

else

- Código simétrico ao anterior, trocando esquerda por direita e vice-versa.

Finalização da Função

```
// Garante que a raiz da árvore seja preta
ptrTree->ptrRoot->bColor = BLACK;
```

Após corrigir todas as violações, garante que a raiz da árvore seja sempre preta.

3.7 printTreeRB

O objetivo das funções printTreeRB é imprimir uma árvore rubro-negra.

```
void printTreeRB(string sPrefix, Node_RB* ptrNode, bool isLeft)
```

Parâmetros:

- **sPrefix:** Uma string que contém o prefixo a ser adicionado à impressão atual, controlando a indentação dos nós na árvore.
- **ptrNode:** Ponteiro para o nó atual que está sendo impresso.
- **isLeft:** Indica se o nó atual é o filho esquerdo (`true`) ou direito (`false`) de seu pai.

Verificação de Nó Nulo:

```
if (ptrNode != nullptr)
```

Verifica se o nó atual não é nulo, pois a função recursiva continua até todos os nós da árvore serem impressos.

Impressão do Nó:

```
cout << sPrefix;
cout << (isLeft ? "" : "");
```

Imprime o prefixo (sPrefix) que controla a indentação. Decide se imprime se o nó for esquerdo ou se for direito.

Impressão do Valor do Nó:

```
if (ptrNode->bColor == RED)
{
    cout << red << ptrNode->iValue << reset << " ";
}
else
{
    cout << ptrNode->iValue << " ";
}
```

Verifica a cor do nó (`ptrNode->bColor`): se for vermelho, imprime o valor com a formatação em red. Caso contrário, imprime o valor normalmente.

Quebra de Linha:

```
cout << endl;
```

Insere uma quebra de linha após imprimir o valor do nó e sua cor, se aplicável.

Recursão para os Filhos:

```
printTreeRB(sPrefix + (isLeft ? "| " : "    "),
ptrNode->ptrLeft, true);
printTreeRB(sPrefix + (isLeft ? "|  " : "    "),
ptrNode->ptrRight, false);
```

Chama recursivamente a função printTreeRB para imprimir os filhos esquerdo e direito do nó atual. Atualiza o prefixo (sPrefix) para ajustar a indentação, adicionando '|' se for filho esquerdo ou se for filho direito.

Função printTreeRB(Node_RB* ptrNode)

Esta função é uma sobrecarga que chama a função `printTreeRB` principal com um prefixo vazio ('') e define `isLeft` como 'false', indicando que o nó não é o filho esquerdo do pai (pois não tem pai).

```
void printTreeRB(Node_RB* ptrNode)
{
    printTreeRB("", ptrNode, false);
}
```

Parâmetros:

- **ptrNode:** Ponteiro para o nó raiz da árvore a ser impressa.

Chama printTreeRB(, ptrNode, false), iniciando a impressão da árvore com um prefixo vazio e considerando que o nó raiz não é um filho esquerdo.

3.8 transplant

O objetivo da função transplant é substituir um nó por outro em uma árvore rubro-negra ptrTree.

```
void transplant(Tree_RB* ptrTree, Node_RB* ptrNode, Node_RB* ptrSubRoot)
```

Parâmetros:

- **ptrTree:** Ponteiro para a estrutura da árvore rubro-negra.
- **ptrNode:** Nó que será substituído.
- **ptrSubRoot:** Nó que substituirá ptrNode.

Verificação se ptrNode é a raiz:

```
if (ptrNode->ptrParent == nullptr)
{
    ptrTree->ptrRoot = ptrSubRoot;
```

```
}
```

Verifica se ptrNode não possui pai (ptrParent == nullptr). Nesse caso, atualiza a raiz da árvore (ptrTree->ptrRoot) para ptrSubRoot.

Verificação do lado do nó em relação ao pai:

```
else if (ptrNode == ptrNode->ptrParent->ptrLeft)
{
    ptrNode->ptrParent->ptrLeft = ptrSubRoot;
}
else {
    ptrNode->ptrParent->ptrRight = ptrSubRoot;
}
```

Se ptrNode possui um pai e é o filho esquerdo desse pai (ptrNode == ptrNode->ptrParent->ptrLeft), então ptrSubRoot será atribuído como o novo filho esquerdo do pai de ptrNode. Caso contrário, ptrSubRoot será atribuído como o novo filho direito do pai de ptrNode.

Atualização do ponteiro pai de ptrSubRoot:

```
if (ptrSubRoot != nullptr) {
    ptrSubRoot->ptrParent = ptrNode->ptrParent;
}
```

Se ptrSubRoot não for nulo, define seu pai como o pai de ptrNode.

3.9 removeFixup

O objetivo da função removeFixup é corrigir violações das propriedades da árvore rubro-negra após a remoção de um nó.

```
void removeFixup(Tree_RB* ptrTree, Node_RB* ptrChild)
```

Parâmetros:

- `ptrTree`: Ponteiro para a estrutura da árvore rubro-negra.
- `ptrChild`: Nó que precisa ser corrigido após a remoção.

Verificação de Nó Nulo:

```
if (ptrChild == nullptr) { return; }
```

Caso o nó a ser corrigido (ptrChild) seja nulo, a função retorna imediatamente, pois não há correções a serem feitas.

Loop de Correção:

```
while (ptrChild != ptrTree->ptrRoot && ptrChild->bColor == BLACK)
{
    // Verifica se correções são necessárias
    if (ptrChild != ptrChild->ptrParent->ptrLeft) { break;}
```

O loop continua enquanto ptrChild não é a raiz e a cor de ptrChild é preta.

Obtenção do Irmão do Nó:

```
Node_RB* ptrSibling = ptrChild->ptrParent->ptrRight;
```

Obtém o irmão de ptrChild, que é o nó à direita do pai de ptrChild.

Caso 1: Irmão Vermelho:

```
    if (ptrSibling->bColor == RED)
    {
        ptrSibling->bColor = BLACK;
        ptrChild->ptrParent->bColor = RED;
        rotateLeft(ptrTree, ptrChild->ptrParent);
        ptrSibling = ptrChild->ptrParent->ptrRight;
    }
```

Se o irmão ptrSibling é vermelho, realiza uma rotação para a esquerda no pai de ptrChild (rotateLeft(ptrTree, ptrChild->ptrParent)). A cor do irmão ptrSibling é ajustada para preto e a cor do pai de ptrChild é ajustada para vermelho. Daí, Atualiza ptrSibling para ser o irmão direito atualizado após a rotação.

Caso 2: Irmão Preto e Filhos Pretos:

```
    if (ptrSibling->ptrLeft->bColor ==
    BLACK && ptrSibling->ptrRight->bColor == BLACK)
    {
        ptrSibling->bColor = RED;
        ptrChild = ptrChild->ptrParent;
    }
```

Se o irmão ptrSibling tem ambos os filhos pretos, ajusta a cor de ptrSibling para vermelho. Move ptrChild para o pai de ptrChild, continuando a verificação por possíveis violações.

Caso 3: Irmão Preto, Filho Esquerdo Vermelho e Filho Direito Preto:

```
    if (ptrSibling->ptrRight->bColor == BLACK)
    {
        ptrSibling->ptrLeft->bColor = BLACK;
        ptrSibling->bColor = RED;
        rotateRight(ptrTree, ptrSibling);
        ptrSibling = ptrChild->ptrParent->ptrRight;
    }
```

Se o irmão ptrSibling tem o filho esquerdo vermelho e o filho direito preto, ajusta a cor do filho esquerdo de ptrSibling para preto. Ajusta a cor de ptrSibling para vermelho e realiza uma rotação para a direita em ptrSibling. Deppis, atualiza ptrSibling para ser o irmão direito atualizado após a rotação.

Caso 4: Irmão Preto e Filho Direito Vermelho:

```
    ptrSibling->bColor = ptrChild->ptrParent->bColor;
    ptrChild->ptrParent->bColor = BLACK;
    ptrSibling->ptrRight->bColor = BLACK;
    rotateLeft(ptrTree, ptrChild->ptrParent);
    ptrChild = ptrTree->ptrRoot;
```

Se o irmão ptrSibling tem o filho direito vermelho, ajusta a cor de ptrSibling para a cor do pai de ptrChild. Ajusta a cor do pai de ptrChild para preto e a cor do filho direito de ptrSibling para preto. Também, realiza uma rotação para a esquerda no pai de ptrChild. Depois, define ptrChild como a raiz da árvore (ptrTree->ptrRoot).

Finalização:

```
ptrChild->bColor = BLACK;
```

Define a cor de ptrChild como preto, garantindo que a raiz da árvore seja preta após as correções.

3.10 verifyBlackNodes

O objetivo da função verifyBlackNodes é verificar se todos os caminhos da árvore rubro-negra, do nó dado até suas folhas nulas (NIL), contêm o mesmo número de nós pretos.

```
bool verifyBlackNodes(Node_RB* ptrNode, int
iCountBlack, int currentCount)
```

Parâmetros:

- `ptrNode`: Nó atual da árvore que está sendo verificado.
- `iCountBlack`: Número esperado de nós pretos em um caminho da raiz até uma folha NIL.
- `currentCount`: Número atual de nós pretos ao longo do caminho atual.

Verificação de Nó Nulo:

```
if (ptrNode == nullptr)
{
    return currentCount == iCountBlack;
}
```

Se ptrNode for nulo, significa que chegamos a uma folha NIL. Nesse caso, a função verifica se currentCount (número de nós pretos ao longo deste caminho) é igual a iCountBlack (número esperado de nós pretos). E aí, retorna true se forem iguais, indicando que este caminho tem o número correto de nós pretos.

Verificação da Cor do Nó Atual:

```
if (ptrNode->bColor == BLACK)
{
    currentCount++;
}
```

Se o nó atual ptrNode for preto, incrementa o currentCount. Isso ajuda a contar o número de nós pretos ao longo do caminho atual da raiz até o nó atual.

Chamadas Recursivas:

```
return verifyBlackNodes(ptrNode->ptrLeft,
iCountBlack,
currentCount) && verifyBlackNodes(ptrNode->ptrRight, iCountBlack,
currentCount);
```

Recursivamente, a função chama verifyBlackNodes para o filho esquerdo e para o filho direito do nó atual ptrNode. Ela passa o mesmo iCountBlack e o currentCount atualizado para cada chamada recursiva. A função retorna verdadeir se os dois caminhos da árvore, a partir do nó atual, satisfazem a condição de ter o mesmo número de nós pretos até suas folhas NIL. MAs, se em algum ponto a condição não for satisfeita (por exemplo, se um caminho tem um número diferente de nós pretos do que o outro), a função retorna falso.

3.11 verifyRedNodes

O objetivo da função verifyRedNodes é verificar se todos os nós vermelhos em uma árvore rubro-negra satisfazem a propriedade de não terem filhos vermelhos.

```
bool verifyRedNodes(Node_RB* ptrNode)
```

Parâmetros:

- `ptrNode`: Nó atual da árvore que está sendo verificado.

Verificação de Nó Nulo:

```
if (ptrNode == nullptr) return true;
```

Se `ptrNode` for nulo, a função retorna verdadeiro. Isso é importante para o caso base da recursão e para lidar com subárvores vazias.

Verificação da Cor do Nó Atual:

```
if (ptrNode->bColor == RED)
{
    if ((ptrNode->ptrLeft != nullptr &&
    ptrNode->ptrLeft->bColor == RED) ||
        (ptrNode->ptrRight != nullptr &&
        ptrNode->ptrRight->bColor == RED)) {
        return false;
    }
}
```

Se o ptrNode atual for vermelho, a função verifica se ele tem algum filho também vermelho. Isso é feito verificando se ptrNode->ptrLeft ou ptrNode->ptrRight é não nulo e se seus respectivos nós têm a cor vermelha. Se encontrar um filho vermelho, a função retorna falso, indicando que a propriedade da árvore rubro-negra foi violada.

Chamadas Recursivas:

```
return verifyRedNodes(ptrNode->ptrLeft)
&& verifyRedNodes(ptrNode->ptrRight);
```

A função chama recursivamente verifyRedNodes para o filho esquerdo (ptrLeft) e para o filho direito (ptrRight) do nó atual ptrNode. Daí, ela retorna verdadeiro se todos os nós vermelhos em toda a árvore, a partir do nó atual, satisfazem a condição de não terem filhos vermelhos. Se em algum nó vermelho tiver um filho vermelho, a função retorna falso.

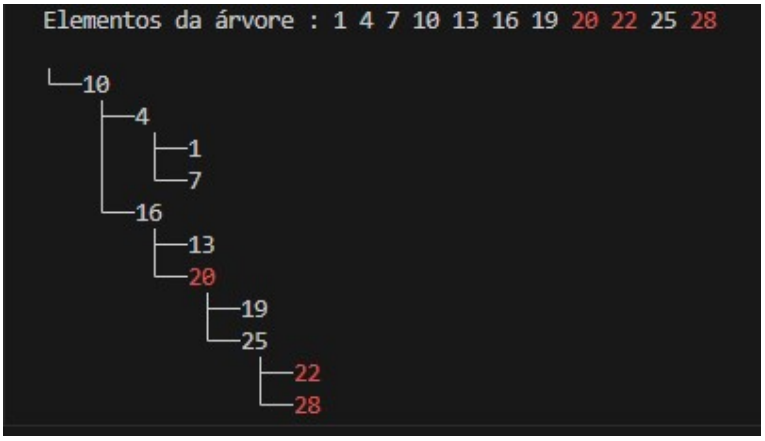
4 Resultado de Testes

Inicializa-se uma árvore Red-Black utilizando a função createTree(). Após a inserção dos elementos, é executado um percurso inorder para listar os elementos da árvore. Em seguida, a estrutura da árvore é impressa utilizando a função printTreeRB().

Para testar a remoção de elementos, o código tenta remover o elemento com valor 60, que não existe, e depois remove o elemento com valor 20, que existe. A estrutura da árvore é novamente impressa após as remoções. A busca de elementos é realizada pelo valor 10 que é encontrado, e pelo valor 80 que não é encontrado. Para verificar a validade da árvore, a função isValid Tree RB() é utilizada . Em seguida, encontra-se o valor mínimo na árvore e verifica-se se este valor é 1. De forma semelhante, o valor máximo na árvore é encontrado e verifica-se se este valor é 28. Finalmente, a altura da árvore é calculada e exibida.

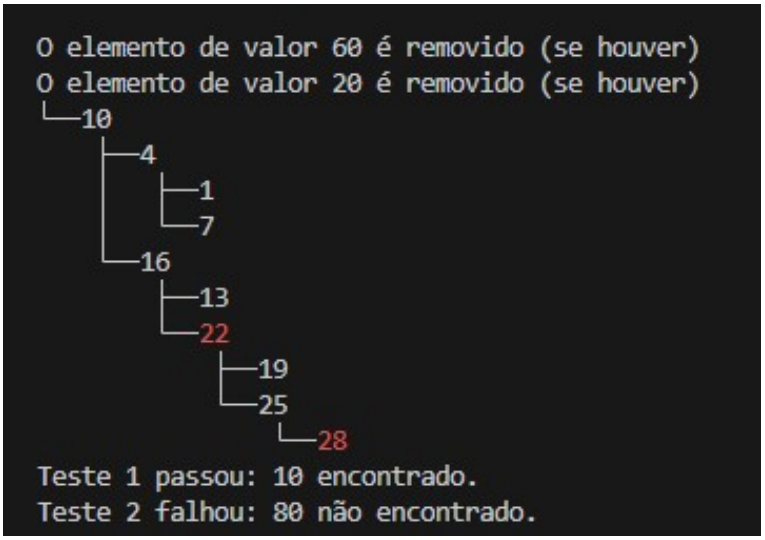
Criando a árvore, inserindo elementos nela e testando a inorder

- **Console:**



Testando as funções remove e search

- **Console:**



Testando isValid Tree RB, findMin, findMax e height

- **Console:**

```
A árvore é válida: 1 - (1 = sim, 0 = não)
Teste passou: valor mínimo é 1.
Teste passou: valor máximo é 28.
Altura da árvore: 4
```

5 Conclusão:

5.1 Pontos críticos

O principal ponto positivo da minha implementação das árvores rubro-negras é a possibilidade de visualizar o estado da árvore após cada operação. Embora não tenha sido um requisito obrigatório do projeto, a inclusão dessa funcionalidade facilita a compreensão da árvore, permitindo uma análise visual da estrutura e dos valores armazenados.

No entanto, um ponto de melhoria seria diminuir a complexidade de algumas funções que ficaram muito grandes, como a de insertFixup e removeFixup que eram funções auxiliares. Considerando que funções longas dificultam a clareza e a organização do código.

5.2 Vantagens e desvantagens

Se a gente considerar por exemplo, uma árvore binária de busca padrão na qual os elementos são inseridos em ordem crescente (1, 2, 3, ..., n), cada novo elemento é inserido como o filho direito do último nó inserido, resultando em uma estrutura em forma de lista ligada com altura O(n). É notório que árvores Rubro-Negras, evitam esse tipo de perda de eficiência.

Por outro lado, garantir a correta aplicação das rotações e recolorações necessárias após inserções e remoções necessitou de um cuidado grande para evitar bugs, por exemplo, um nó vermelho não pode ter filhos vermelhos. Além disso, garantir que os ponteiros para os nós da árvore fossem atualizados corretamente, me trouxe problemas com falhas que levaram a acessos inválidos à memória (segmentation faults), depois de alguns ajustes tais problemas foram resolvidos.

Referências

[1] André Backes. *Estrutura de Dados em C | Aula 105 - Árvore Rubro Negra - Definição*. 2015. URL: https://youtu.be/DaWNuijRRFY?si=M14ZCo1x-nRm9LUi.

[2] John Morris. *8.2 Red-Black Trees*. https://www.eecs.umich.edu/courses/eecs380/ALG/red_black.html. Acessado em 1 de julho de 2024.