Fundação Getúlio Vargas
Escola de Matemática Aplicada

Ciência de Dados e Inteligência Artificial

Relatório de Estruturas de Dados

Leonardo Alexandre da Silva Ferreira
João Felipe Vilas Boas

Rio de Janeiro
Julho / 2024

Leonardo Alexandre da Silva Ferreira
João Felipe Vilas Boas

Relatório da Atividade Avaliativa de Estrutura de Dados

Relatório da Atividade Avaliativa da disciplina Estrutura de Dados do curso Ciência de Dados e Inteligência Artificial

Professor:
Rafael de Pinho André

Rio de Janeiro
Julho / 2024

Sumário

1 Introdução

As árvores rubro-negras são um tipo de árvore binária de busca autobalanceada. Elas são utilizadas em várias aplicações de ciência da computação devido às suas operações de inserção, exclusão e busca eficientes, que garantem complexidade de tempo $O(\log n)$. Uma árvore rubro-negra é uma árvore binária de busca que segue estas propriedades:

1. Cada nó é vermelho ou preto.
2. A raiz é preta.
3. Todas as folhas são pretas.
4. Se um nó é vermelho, então ambos os filhos são pretos.
5. Para cada nó, todos os caminhos simples do nó até as folhas descendentes contêm o mesmo número de nós pretos.

Essas propriedades garantem que o caminho mais longo da raiz até uma folha não seja mais do que o dobro do caminho mais curto, o que assegura o balanceamento da árvore.

2 Implementação

A implementação da árvore rubro-negra foi dividida em três arquivos principais:

RBTree.h: Contém a definição da estrutura do nó e da classe da árvore rubro-negra, incluindo todas as funções necessárias.
RBTree.cpp: Implementa as funções declaradas em **RBTree.h**.
main.cpp: Realiza os testes de inserção, exclusão e busca na árvore rubro-negra.

2.1 Estrutura do Arquivo RBTree.h

Estrutura do Nó (`Node`): Define um nó da árvore, incluindo seus atributos (dados, cor, ponteiros para os filhos e pai).
Classe da Árvore (`RBTree`): Define a estrutura da árvore e declara as funções utilizadas para manipulação da árvore, como inserção, exclusão, rotação, balanceamento e percursos.

2.2 Implementação das Funções em RBTree.cpp

Construtor do Nó: Inicializa os atributos do nó.
Função: `Node<T>::Node(T data)`

Construtor da Árvore: Inicializa a árvore com um nó `TNULL` que representa as folhas nulas.
Função: `RBTree<T>::RBTree()`

Inicialização do Nó Nulo (TNULL): Inicializa um nó TNULL com dados nulos e cor preta.
Função: `initializeNULLNode(Node<T>* node, Node<T>* parent)`

Funções de Percurso: Implementam percursos preOrder, inOrder e postOrder.
Funções: `preOrderHelper(Node<T>* node)`, `inOrderHelper(Node<T>* node)`, `postOrderHelper(Node<T>* node)`

Busca: Busca um nó com uma chave específica na árvore.

Função: `searchTreeHelper(Node<T>* node, T key)`, `searchTree(T k)`

Inserção e Balanceamento: Insere um nó na árvore e balanceia a árvore após a inserção, o que garante as propriedades da árvore rubro-negra.
Funções: `insert(T key)`, `balanceInsert(Node<T>* k)`

Impressão da Árvore: Função para imprimir a árvore de forma visualmente agradável.
Função: `printHelper(Node<T>* root, std::string indent, bool last)`, `printTree()`

Nó com a Menor Chave: Encontra o nó com a menor chave a partir de um nó dado.
Função: `minimum(Node<T>* node)`

Nó com a Maior Chave: Encontra o nó com a maior chave a partir de um nó dado.
Função: `maximum(Node<T>* node)`

Sucessor de um Nó: Encontra o sucessor de um nó dado.
Função: `successor(Node<T>* x)`

Antecessor de um Nó: Encontra o antecessor de um nó dado.
Função: `predecessor(Node<T>* x)`

Raiz da Árvore: Retorna a raiz da árvore.
Função: `getRoot()`

Substituição de um Nó por Outro: Substitui um nó `u` por um nó `v` e ajusta os ponteiros dos pais e filhos conforme necessário.
Função: `rbTransplant(Node<T>* u, Node<T>* v)`

Rotações: Implementam rotações à esquerda e à direita para manter o balanceamento da árvore.
Funções: `leftRotate(Node<T>* x)`, `rightRotate(Node<T>* x)`

Exclusão e Balanceamento: Exclui um nó da árvore e realiza o balanceamento subsequente.
Funções: `deleteNode(T data)`, `deleteNodeHelper(Node<T>* node, T key)`, `balanceDelete(Node<T>* x)`

Validação da Árvore Rubro-Negra: Valida as propriedades específicas de uma árvore rubro-negra.
Funções: `validateRBTree()`, `validateHelper(Node<T>* node)`, `checkRBProperties(Node<T>* node, int& blackCount, int pathBlackCount)`

Funções de Altura: Calcula a altura da árvore.
Funções: `height()`, `calculateHeight(Node<T>* node)`

Função TNULL: Retorna o nó TNULL, que representa as folhas nulas na árvore.
Função: `getTNULL()`

2.3 Estrutura do Arquivo main.cpp

O arquivo **main.cpp** contém funções específicas para realizar testes na árvore rubro-negra.

Teste de Inserção: Insere várias chaves na árvore e imprime o estado da árvore após as inserções.
Função: `testInsert()`

Teste de Exclusão: Insere algumas chaves e realiza exclusões específicas, verificando o comportamento da árvore após cada exclusão.
Função: `testDelete()`

Teste de Busca: Insere chaves na árvore e realiza buscas para verificar a presença das chaves inseridas e a ausência de chaves não inseridas.
Função: `testSearch()`

Teste de Validação: Valida a árvore rubro-negra para garantir que todas as propriedades da árvore sejam mantidas.
Função: `testValidation()`

Teste de Altura: Calcula e imprime a altura da árvore antes e após inserções adicionais.

Função: `testHeight()`

Teste de Nó Mínimo: Encontra e imprime o nó com a menor chave na árvore antes e após inserções adicionais.

Função: `testMinimum()`

Teste de Percurso em Ordem Simétrica (Inorder): Realiza e imprime o percurso em ordem simétrica (inorder) da árvore.

Função: `testInorder()`

Teste de Nó Máximo: Encontra e imprime o nó com a maior chave na árvore antes e após inserções adicionais.

Função: `testMaximum()`

Esses testes verificam se a árvore mantém suas propriedades e se comporta conforme o esperado nas operações de inserção, exclusão, busca, validação, cálculo de altura, percurso inOrder e identificação dos nós mínimo e máximo.

3 Testes Realizados

Os testes foram implementados no arquivo `main.cpp`, com funções dedicadas para verificar a integridade e a funcionalidade da árvore rubro-negra. Por meio de três funções principais, `testInsert`, `testDelete`, e `testSearch`, foram realizadas diversas operações para garantir que a árvore cumpra com suas propriedades e requisitos. A seguir, é detalhado cada tipo de teste realizado, o que inclui os objetivos e as abordagens utilizadas.

3.1 Teste de Inserção

Objetivo: Verificar se a árvore rubro-negra mantém suas propriedades de balanceamento e estrutura correta após a inserção de múltiplas chaves.

Implementação: A função `testInsert` é responsável por adicionar uma série de chaves à árvore e imprimir o estado da árvore após diferentes etapas de inserção. Este processo é dividido em três etapas principais:

1. **Inserções Iniciais:**
    - A função insere um conjunto inicial de chaves: 55, 40, 65, 60, 75 e 57.
    - Após as inserções, a árvore é impressa para verificar a manutenção da estrutura correta e o balanceamento dos nós.
    - O objetivo é observar se a árvore continua a seguir as propriedades da árvore rubro-negra, como a alternância de cores e a restrição de que cada caminho da raiz para uma folha preta tenha o mesmo número de nós pretos.
2. **Inserções Adicionais:**
    - São inseridas mais chaves: 20, 30, 10, 90 e 80.
    - A árvore é impressa novamente após as inserções para verificar se o balanceamento e as propriedades da árvore são preservados.
3. **Inserções Mais Complexas:**
    - São adicionadas outras chaves: 85, 50 e 35, além de 5.

- A impressão da árvore após essas inserções ajuda a verificar o comportamento da árvore em cenários mais complexos e se o balanceamento ainda é mantido de forma adequada.

Este teste assegura que a árvore rubro-negra consegue lidar com múltiplas inserções e manter seu balanceamento, fundamental para a eficiência das operações subsequentes.

3.2 Teste de Exclusão

Objetivo: Garantir que a árvore rubro-negra mantenha suas propriedades e estrutura correta após a exclusão de nós, o que inclui a tentativa de excluir um nó inexistente.

Implementação: A função `testDelete` realiza inserções seguidas de exclusões e imprime o estado da árvore após cada exclusão para avaliar a eficácia do processo de exclusão e balanceamento. Os testes são divididos em:

1. **Exclusões Específicas:**
   - A função começa com a inserção de chaves: 55, 40, 65, 60, 75 e 57.
   - Em seguida, exclui nós específicos: 40, 65 e 55.
   - Após cada exclusão, a árvore é impressa para verificar se as propriedades da árvore rubro-negra são respeitadas e se a árvore continua balanceada.
2. **Exclusão de Nó Inexistente:**
   - A função tenta excluir um nó com a chave 999, que não está presente na árvore.
   - Este teste valida que a função de exclusão lida corretamente com a tentativa de remover um nó inexistente sem afetar a estrutura da árvore.
3. **Exclusões Adicionais:**

- Após as exclusões iniciais, são realizadas novas inserções (20, 30, 10, 90 e 80) seguidas de exclusões: 20, 30, 10, 90 e 80.
- O estado da árvore é verificado após cada exclusão para assegurar que a árvore permanece balanceada e suas propriedades são mantidas.

Este teste garante que a árvore rubro-negra pode lidar com a exclusão de nós de forma eficiente e correta, mantendo a integridade das suas propriedades após cada operação.

3.3 Teste de Busca

Objetivo: Verificar a capacidade da árvore rubro-negra de localizar chaves específicas e assegurar que a busca de chaves inexistentes e a busca em uma árvore vazia sejam tratadas corretamente.

Implementação: A função `testSearch` insere um conjunto de chaves e realiza várias buscas para verificar a precisão das operações de busca:

1. **Buscas em Chaves Existentes:**
   - A função começa com a inserção de chaves: 55, 40, 65, 60, 75 e 57.
   - São realizadas buscas para verificar a presença das chaves 65, 40 e 75, confirmando que estas chaves podem ser encontradas na árvore.
2. **Busca de Chaves Inexistentes:**
   - A função realiza buscas para chaves que não estão presentes na árvore, como 999.
   - Este teste assegura que a função de busca retorna o resultado correto para chaves inexistentes, confirmando o comportamento esperado da função de busca.
3. **Buscas em Chaves Adicionais:**

- Novas chaves são inseridas (20, 30, 10, 90 e 80) e são realizadas buscas para verificar se estas chaves podem ser encontradas na árvore.

4. **Teste de Busca em Árvore Vazia:**

   - É realizada uma busca em uma árvore que não possui chaves (uma árvore recém-criada).
   - Este teste verifica se a função de busca lida corretamente com o cenário de uma árvore vazia, confirmando que a função de busca não retorna resultados incorretos.

Este teste verifica a precisão e robustez da função de busca, garantindo que a árvore possa localizar chaves de forma correta e que a função lida com casos de chaves inexistentes e árvores vazias de maneira apropriada.

3.4 Teste de Validação

Objetivo: Garantir que a árvore rubro-negra mantenha suas propriedades após diversas inserções.

Implementação: A função `testValidation` insere várias chaves na árvore e verifica se a árvore continua a ser uma árvore rubro-negra válida após as inserções:

- Inserções de chaves: 55, 40, 65, 60, 75 e 57.
- A validação é realizada e o resultado é impresso para confirmar se a árvore é válida.
- Inserções adicionais de chaves: 20, 30, 10, 90 e 80.
- A validação é realizada novamente para garantir que a árvore continua a ser válida após mais inserções.

Este teste assegura que a árvore rubro-negra mantém suas propriedades fundamentais após inserções consecutivas.

3.5 Teste de Altura

Objetivo: Verificar se a altura da árvore é calculada corretamente e se a altura reflete a estrutura balanceada da árvore após várias inserções.

Implementação: A função `testHeight` insere várias chaves e calcula a altura da árvore em dois momentos:

- Após inserções iniciais: 55, 40, 65, 60, 75 e 57.
- A altura é calculada e impressa para verificar a altura da árvore.
- Inserções adicionais: 20, 30, 10, 90 e 80.
- A altura é calculada novamente para verificar a altura da árvore após mais inserções.

Este teste verifica se a altura da árvore é adequada e reflete a estrutura balanceada da árvore rubro-negra.

3.6 Teste de Nó Mínimo

Objetivo: Encontrar e verificar o nó com a menor chave na árvore.

Implementação: A função `testMinimum` insere várias chaves e encontra o nó mínimo em dois momentos:

- Após inserções iniciais: 55, 40, 65, 60, 75 e 57.
- O nó mínimo é encontrado e impresso.
- Inserções adicionais: 20, 30 e 10.
- O nó mínimo é encontrado novamente e impresso.

Este teste garante que a função de encontrar o nó mínimo funciona corretamente.

3.7 Teste de Percurso em Ordem Simétrica (Inorder)

Objetivo: Realizar e imprimir o percurso em ordem simétrica (inorder) da árvore.

Implementação: A função `testInorder` insere várias chaves e realiza um percurso em ordem simétrica.

- Após inserções iniciais: 55, 40, 65, 60, 75 e 57.
- O percurso em ordem simétrica é realizado e impresso.

Este teste assegura que a função de percurso em ordem simétrica funciona corretamente e imprime a árvore na ordem correta.

3.8 Teste de Nó Máximo

Objetivo: Encontrar e verificar o nó com a maior chave na árvore.

Implementação: A função `testMaximum` insere várias chaves e encontra o nó máximo em dois momentos:

- Após inserções iniciais: 55, 40, 65, 60, 75 e 57.
- O nó máximo é encontrado e impresso.
- Inserções adicionais: 20, 30, 10, 90 e 80.
- O nó máximo é encontrado novamente e impresso.

Este teste garante que a função de encontrar o nó máximo funciona de forma correta.

4 Otimização do Código

A implementação da árvore rubro-negra é eficiente por natureza devido às suas características e estratégias de design específicas. Abaixo, têm-se as principais otimizações aplicadas na implementação, explicando como cada uma contribui para a eficiência e robustez da árvore.

4.1 Folhas Nulas (TNULL)

A introdução do conceito de folhas nulas (TNULL) é uma das otimizações mais eficazes na implementação da árvore rubro-negra.

- Simplificação do Código: O uso do nó TNULL para representar as folhas da árvore reduz a complexidade do código ao tratar todos os nós da árvore de forma uniforme. Em vez de lidar com casos especiais para nós nulos, TNULL simplifica a lógica de operações como inserções, exclusões e balanceamentos.

- Consistência Operacional: A presença de TNULL permite que operações como inserções e exclusões tratem todos os nós de maneira consistente, sem a necessidade de verificar se um filho é nulo, pois todas as folhas são representadas por TNULL. Isso facilita a implementação de métodos de manipulação da árvore, como inserções e balanceamentos, tornando o código mais limpo e menos propenso a erros.

4.2 Rotações e Balanceamento

As operações de rotação e balanceamento foram otimizadas para garantir que a árvore mantenha suas propriedades de balanceamento de maneira eficiente.

- Complexidade $O(\log n)$: A rotação e o balanceamento são operações cruciais que garantem que a árvore permaneça balanceada, com uma complexidade de tempo $O(\log n)$ para as operações de inserção e exclusão. As rotações são realizadas em tempo constante $O(1)$, e o balanceamento pode exigir um número fixo de rotações e ajustes.

- Estratégias de Balanceamento: Após cada inserção ou exclusão, métodos específicos ajustam as cores dos nós e realizam rotações para manter

as propriedades da árvore rubro-negra. Esses métodos asseguram que as regras da árvore rubro-negra sejam seguidas, o que é fundamental para a eficiência das operações de busca, inserção e exclusão.

4.3 Eficiência dos Métodos de Busca e Manipulação

O design dos métodos de busca e manipulação da árvore é otimizado para garantir eficiência e robustez.

- Busca e Manipulação de Nó: Métodos como `searchTree`, `minimum`, `maximum`, `successor` e `predecessor` são implementados para garantir que a busca e manipulação de nós sejam realizadas de forma eficiente. A busca por um nó é feita em tempo $O(\log n)$, aproveitando a estrutura balanceada da árvore.

- Mínimo e Máximo: Os métodos para encontrar o nó com a menor e maior chave a partir de um dado nó são otimizados para navegar diretamente para as folhas mais à esquerda ou à direita.

- Sucessor e Antecessor: Estes métodos determinam rapidamente o próximo e o nó anterior em ordem de chave, utilizando a estrutura da árvore para realizar buscas e ajustes de forma eficiente.

4.4 Substituição de Nós (`rbTransplant`)

A função `rbTransplant` foi otimizada para substituir um nó por outro com eficiência.

- Eficiência na Substituição de Nós: A função `rbTransplant` substitui um nó u por um nó v de forma eficiente, ajustando os ponteiros dos pais e dos filhos de acordo com a substituição. Esta função é crucial durante a exclusão de nós e é projetada para ser executada em tempo constante $O(1)$.

- Manutenção da Complexidade: A função `rbTransplant` permite que a estrutura da árvore seja ajustada sem a necessidade de rebalançar toda a árvore, mantendo a complexidade da operação de exclusão de nós em $O(\log n)$.

4.5 Manutenção da Propriedade da Árvore Rubro-Negra

As operações de inserção e exclusão garantem que as propriedades da árvore rubro-negra sejam mantidas.

- Garantia das Propriedades: Durante a inserção e exclusão de nós, métodos de balanceamento são aplicados para garantir que todas as propriedades da árvore rubro-negra, como a alternância de cores e a restrição de que cada caminho da raiz para uma folha preta tenha o mesmo número de nós pretos, sejam respeitadas.

- Preservação da Eficiência: O balanceamento pós-inserção e pós-exclusão garante que a árvore continue a cumprir as propriedades rubro-negras, o que assegura a manutenção da eficiência das operações de busca e modificação.

5 Análise Crítica

Primeiramente, a eficiência da implementação é um dos seus pontos fortes mais evidentes. A árvore rubro-negra é projetada para garantir que as operações de inserção, exclusão e busca sejam realizadas em tempo $O(\log n)$. A implementação segue rigorosamente essas garantias de complexidade, o que a torna adequada para aplicações que exigem desempenho consistente e previsível, mesmo com um grande número de elementos.

A implementação faz uso eficaz do conceito de folhas nulas (TNULL), uma prática que simplifica a lógica de inserção e exclusão ao tratar todas as folhas como nós com um valor padrão e uma cor fixa. Esta abordagem reduz a necessidade de verificações adicionais e ajuda a manter o código mais claro e mais fácil de entender.

A estrutura clara e o design organizado do código também são pontos fortes. As funções são bem segmentadas e seguem uma estrutura lógica que facilita a leitura e a manutenção. Funções como `leftRotate`, `rightRotate`, `balanceInsert` e `balanceDelete` são implementadas de forma a garantir que as operações de balanceamento sejam feitas de acordo com as regras da árvore rubro-negra, preservando suas propriedades fundamentais.

Além disso, a implementação inclui uma função para imprimir a árvore (`printTree`), que fornece uma visualização útil da estrutura da árvore. Esta função é uma ferramenta valiosa para a depuração e verificação visual do estado da árvore após várias operações, ajudando a garantir que a árvore está se comportando conforme o esperado.

Os testes realizados no arquivo `main.cpp` confirmam que a árvore rubro-negra mantém suas propriedades após a inserção e a exclusão de nós e desempenha as operações de maneira correta, demonstrando que a implementação atende aos requisitos funcionais.

Em resumo, a implementação da árvore rubro-negra apresentada é eficiente e flexível, com uma estrutura de código clara e uma abordagem inteligente para a gestão de nós e operações. A implementação de funções de balanceamento e visualização são bem feitas, e os testes realizados asseguram que as operações básicas são executadas de maneira correta.

6 Conclusão

A implementação da árvore rubro-negra em C++ demonstrou ser tanto eficiente quanto robusta, atendendo efetivamente às exigências de uma estrutura de dados balanceada e de alta performance. A árvore rubro-negra é conhecida por sua capacidade de manter o equilíbrio das operações de busca, inserção e exclusão com uma complexidade de tempo garantida de $O(\log n)$, e a implementação apresentada cumpre essas expectativas com excelência.