Pedro Guimarães Lopes Martins

Gabriela Apetz Lima

Relatório sobre o Funcionamento do Código de Árvores Binárias e AVL

***Funcionamento**

Árvores Binárias: No código BinaryTree.java, temos uma classe chamada **BinaryTree**, que representa uma árvore binária simples.

- **Inserção:** Quando você insere um número, o código decide se ele vai para a esquerda ou para a direita, dependendo do valor. Por exemplo, números menores vão para a esquerda, maiores vão para a direita.
- **Remoção:** Para remover um número, o código encontra o número na árvore e faz a remoção. Se o número a ser removido tiver dois filhos, ele faz uma substituição inteligente.
    - **Problema**:** O método remover parece estar removendo todas as instâncias do número passado como parâmetro ☹️. Outra possível explicação é que após a remoção a substituição não está sendo feita de forma correta. Esse problema é evidenciado ao remover um elemento, e logo após, efetuar a busca por uma outra possível ocorrência deste número (algo muito provável, dado em conta o número de elementos inseridos em um intervalo de números aleatórios controlado).
- **Busca:** Para buscar um número na árvore, o código começa na raiz e vai navegando até encontrar o número desejado ou chegar a uma folha, implementado no código de forma recursiva.

Árvores AVL: No código AVLTree.java, temos também uma classe chamada **AVLTree**, que representa uma árvore AVL, que se mantém balanceada após inserção e remoção de elementos.

- **Inserção:** Quando você insere um número em uma árvore AVL, o código faz as rotações necessárias para manter o equilíbrio. Se a árvore começar a ficar torta, ele a endireita automaticamente.
- **Remoção:** A remoção em uma árvore AVL também é inteligente. Ele encontra o número a ser removido e, novamente, faz as rotações necessárias para manter o equilíbrio.
    - **Problema**:** Mesmo problema de remoção, explicado na parte de Árvore Binária.
- **Busca:** A busca em uma árvore AVL é semelhante à busca em uma árvore binária.

Medindo o Desempenho: O código também mede o tempo que cada operação leva. Ele usa **System.nanoTime()** para fazer isso. Essas medições nos ajudam a entender como as duas árvores se saem em termos de velocidade em diferentes operações.

Main.java: é a classe principal do programa e coordena as operações das árvores binárias e AVL. Aqui está uma visão geral do que o código faz:

- Cria instâncias de **BinaryTree** e **AVLTree** para representar as duas árvores.
- Gera números aleatórios e insere-os nas duas árvores, medindo o tempo de inserção.
- Entra em um loop para permitir que o usuário realize operações como consulta e remoção.
- Mede o tempo de busca e remoção para ambas as árvores.
- Exibe os resultados no console, incluindo os tempos de execução das operações.

***Análise Crítica**

Comparando as arvores analisando seu desempenho (análise de tempo) com a inserção de 100,500,1000 e 10.000 e 20.000 elementos gerados aleatoriamente, remoção e busca, obtivemos os seguintes resultados (em questão de nanosegundos):

-Inserção: Sempre mais demorada na árvore AVL.

-Busca: Resultado variável, sendo dependente da posição do número buscado

-Remoção: Quase Sempre mais demorada na árvore AVL, possivelmente pois esta utiliza de recursão para realizar o método.

***"Qual funciona melhor? Em qual situação? Por quê?"**

A escolha entre árvores binárias e árvores AVL depende da frequência das operações de inserção e remoção, bem como das necessidades de desempenho. Resumidamente, em um caso em que o desempenho é um fator chave, e as operações de remoção e inserção se dão de forma constante, a árvore AVL é a melhor escolha devido ao seu mecanismo de balanceamento automático. No entanto, se o que mais é valorizado é a simplicidade de implementação, com inserção e remoção menos frequentes, a árvore Binária simples é a melhor opção.