**Avaliação RA02 - Árvore AVL e Binária**

**Christiansen Taques Dias, Angelo Dela Vedova Brocca e Gustavo Vinicius Aymoto**

O código consiste em implementações de duas estruturas de árvores em Java: uma árvore binária e uma árvore AVL. Além disso, há uma classe Main que realiza testes de desempenho para ambas as árvores, medindo o tempo de execução para operações de inserção, busca e remoção em diferentes tamanhos de conjuntos de dados.

A estrutura básica de cada árvore (Binária e AVL) é composta por um conjunto de nós, onde cada nó possui um valor, uma referência para o nó à esquerda e outra para o nó à direita, além de uma altura (no caso da AVL). Cada árvore possui métodos para inserção, busca e remoção de elementos.

A classe “ArvoreBinaria” representa a árvore binária. Os métodos principais incluem:

* *inserirRecursivamente*: insere um valor na árvore de forma recursiva, mantendo a propriedade de árvore de busca binária.
* *buscarRecursivamente*: realiza uma busca recursiva na árvore para verificar se um valor está presente.
* *removerRecursivamente*: remove um valor da árvore de forma recursiva, mantendo a propriedade de árvore de busca binária.

Métodos auxiliares como altura, getBalanceamento, rotacaoDireita e rotacaoEsquerda são utilizados para manter a árvore balanceada.

A classe ArvoreAVL é semelhante à ArvoreBinaria, mas inclui lógica adicional para manter a árvore balanceada automaticamente após operações de inserção e remoção.

A classe Main é responsável por realizar testes de desempenho nas duas árvores. Ela cria instâncias de ambas as árvores, insere, busca e remove elementos aleatórios e mede o tempo de execução para cada operação em diferentes tamanhos de conjuntos de dados.

Os resultados dos testes de desempenho são impressos no console, mostrando o tempo de execução para cada operação em ambas as árvores em conjuntos de dados de tamanhos diferentes.

**Métodos do Código**

**Métodos da classe ArvoreBinaria:**

* *inserir(int valor):*Este método insere um novo nó com o valor especificado na árvore binária .  
  Utiliza o método inserirRecursivamente para realizar a inserção de forma recursiva.
* *inserirRecursivamente(No atual, int valor)*:  
  Realiza a inserção de um valor na árvore binária de forma recursiva.  
  Se o nó atual for nulo, cria um novo nó com o valor especificado.  
  Compara o valor a ser inserido com o valor do nó atual e decide se deve ser inserido à esquerda ou à direita.  
  Atualiza a altura do nó atual e realiza rotações se necessário para manter a propriedade de árvore de busca binária.
* *buscar(int valor)*:  
  Verifica se um determinado valor está presente na árvore binária de - forma recursiva.  
  Utiliza o método buscarRecursivamente.
* *buscarRecursivamente(No atual, int valor)*:  
  Realiza a busca por um valor na árvore binária de forma recursiva.  
  Se o nó atual for nulo, retorna false.  
  Compara o valor a ser buscado com o valor do nó atual e decide se deve procurar à esquerda ou à direita.
* *remover(int valor)*:  
  Remove um nó com o valor especificado da árvore binária .  
  Utiliza o método removerRecursivamente.
* *removerRecursivamente(No atual, int valor)*:  
  Realiza a remoção de um valor na árvore binária de forma recursiva.  
  Se o nó atual for nulo, retorna nulo.  
  Se o valor a ser removido for igual ao valor do nó atual, executa a lógica para remover o nó.  
  Atualiza a altura do nó atual e realiza rotações se necessário para manter a propriedade de árvore de busca binária.
* *rotacaoDireita(No y)*:  
  Realiza uma rotação à direita na árvore binária.  
  Utilizado para corrigir desbalanceamentos durante a inserção ou remoção.
* *rotacaoEsquerda(No x)*:  
  Realiza uma rotação à esquerda na árvore binária.  
  Utilizado para corrigir desbalanceamentos durante a inserção ou remoção.
* *altura(No no)*:  
  Retorna a altura de um nó (ou 0 se o nó for nulo).
* *getBalanceamento(No no)*:  
  Retorna o fator de balanceamento de um nó (diferença entre as alturas das subárvores esquerda e direita).
* *atualizarAlturaNo(No no)*:  
  Atualiza a altura de um nó com base nas alturas das subárvores esquerda e direita.

**Métodos adicionais da classe ArvoreAVL:**

* *atualizarAlturaNo(No no)*:  
  Atualiza a altura de um nó AVL após a inserção ou remoção.

**Prints, Resultado e Análise**

Os resultados apresentados pelo código indicam o tempo de execução, em milissegundos, para operações de inserção, busca e remoção em árvores binárias de busca e árvores AVL com diferentes tamanhos de conjuntos de dados. Vamos analisar cada parte:

1. **Testando com 100 elementos:**

**Inserção:** Ambas as árvores (Binária e AVL) levaram 1 ms para inserir 100 elementos.

**Busca:** Ambas as árvores não demoraram tempo perceptível (0 ms) para realizar buscas em 100 elementos.

**Remoção:** Ambas as árvores não demoraram tempo perceptível (0 ms) para realizar remoções em 100 elementos.

1. **Testando com 500 elementos:**

**Inserção:** Ambas as árvores levaram 1 ms para inserir 500 elementos.

**Busca:** Ambas as árvores não demoraram tempo perceptível (0 ms) para realizar buscas em 500 elementos.

**Remoção:** A árvore Binária não demorou tempo perceptível (0 ms) para remover elementos, enquanto a árvore AVL levou 1 ms.

1. **Testando com 1000 elementos:**

**Inserção:** Ambas as árvores não demoraram tempo perceptível (0 ms) para inserir 1000 elementos.

**Busca:** A árvore Binária não demorou tempo perceptível (0 ms) para buscar elementos, enquanto a árvore AVL levou 1 ms.

**Remoção:** A árvore Binária levou 1 ms para remover elementos, enquanto a árvore AVL não demorou tempo perceptível (0 ms).

1. **Testando com 10000 elementos:**

**Inserção:** A árvore Binária levou 3 ms, e a árvore AVL levou 4 ms para inserir 10000 elementos.

**Busca:** A árvore Binária levou 2 ms, e a árvore AVL levou 1 ms para buscar elementos em 10000 elementos.

**Remoção:** A árvore Binária levou 3 ms, e a árvore AVL levou 2 ms para remover elementos em 10000 elementos.

1. **Testando com 20000 elementos:**

**Inserção**: Ambas as árvores (Binária e AVL) levaram 3 ms para inserir 20000 elementos.

**Busca:** A árvore Binária levou 4 ms, e a árvore AVL levou 3 ms para buscar elementos em 20000 elementos.

**Remoção:** A árvore Binária levou 3 ms, e a árvore AVL levou 11 ms para remover elementos em 20000 elementos.

Com isso, podemos concluir que a árvore AVL, devido ao seu mecanismo de auto balanceamento, pode levar um pouco mais de tempo em algumas operações, especialmente em conjuntos de dados maiores. O tempo de execução pode variar de acordo com as características específicas dos dados inseridos, pois o desempenho das árvores depende da distribuição dos valores. Em geral, o tempo de execução para operações em árvores AVL é ligeiramente maior devido à necessidade de manter a árvore balanceada. No entanto, essa diferença pode ser compensada pelos benefícios de uma árvore balanceada em tempo de busca.

**Resumo:**

O código consiste na implementação de duas estruturas de árvores em Java: uma árvore binária, e uma árvore AVL. A classe Main realiza testes de desempenho nas duas árvores, medindo o tempo de execução para operações de inserção, busca e remoção em conjuntos de dados de diferentes tamanhos. Os resultados indicam que, em conjuntos menores, ambas as árvores têm desempenho semelhante, mas à medida que o tamanho dos conjuntos aumenta, a árvore AVL, devido ao seu mecanismo de auto balanceamento, pode ter um tempo ligeiramente maior em algumas operações. A análise dos prints mostra que a árvore AVL mantém a vantagem de busca balanceada, mesmo que isso implique em tempos de execução um pouco maiores em operações de inserção e remoção em comparação com a árvore Binária.