PUCPR- Pontifícia Universitária Católica do Paraná
Curso:Bacharelado em Ciência da Computação
Discente:Victor Silva Camargo, Vinícius Silva Camargo
Docente:Andrey Cabral Meira

1 . Comparando o desempenho com inserção de 100, 500 ,1000, 10000 e 20000

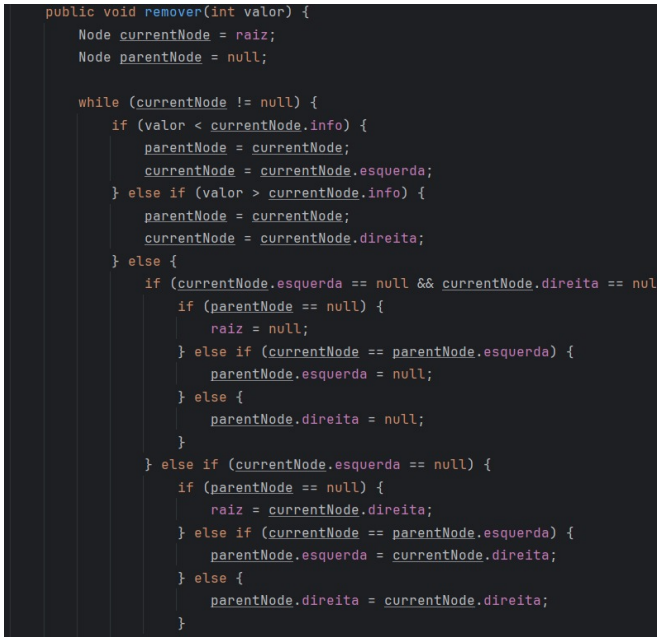
```
Tamanho da Árvore: 100
Tempo de Inserção AVL: 715400 ns
Tempo de Inserção Árvore Não Balanceada: 660000 ns
Tempo de Busca AVL: 4200 ns
Tempo de Busca Árvore Não Balanceada: 2200 ns
Tempo de Remoção AVL: 23100 ns
Tempo de Remoção Árvore Não Balanceada: 6300 ns

Tamanho da Árvore: 500
Tempo de Inserção AVL: 519600 ns
Tempo de Inserção Árvore Não Balanceada: 143100 ns
Tempo de Busca AVL: 800 ns
Tempo de Busca Árvore Não Balanceada: 700 ns
Tempo de Remoção AVL: 1500 ns
Tempo de Remoção Árvore Não Balanceada: 1000 ns

Tamanho da Árvore: 1000
Tempo de Inserção AVL: 291100 ns
Tempo de Inserção Árvore Não Balanceada: 114000 ns
Tempo de Busca AVL: 500 ns
Tempo de Busca Árvore Não Balanceada: 700 ns
Tempo de Remoção AVL: 1200 ns
Tempo de Remoção Árvore Não Balanceada: 900 ns
```

```
Tamanho da Árvore: 10000
Tempo de Inserção AVL: 2199400 ns
Tempo de Inserção Árvore Não Balanceada: 1666800 ns
Tempo de Busca AVL: 600 ns
Tempo de Busca Árvore Não Balanceada: 1100 ns
Tempo de Remoção AVL: 2700 ns
Tempo de Remoção Árvore Não Balanceada: 1000 ns

Tamanho da Árvore: 20000
Tempo de Inserção AVL: 2076300 ns
Tempo de Inserção Árvore Não Balanceada: 2633300 ns
Tempo de Busca AVL: 1100 ns
Tempo de Busca Árvore Não Balanceada: 1900 ns
Tempo de Remoção AVL: 2900 ns
Tempo de Remoção Árvore Não Balanceada: 2000 ns
```

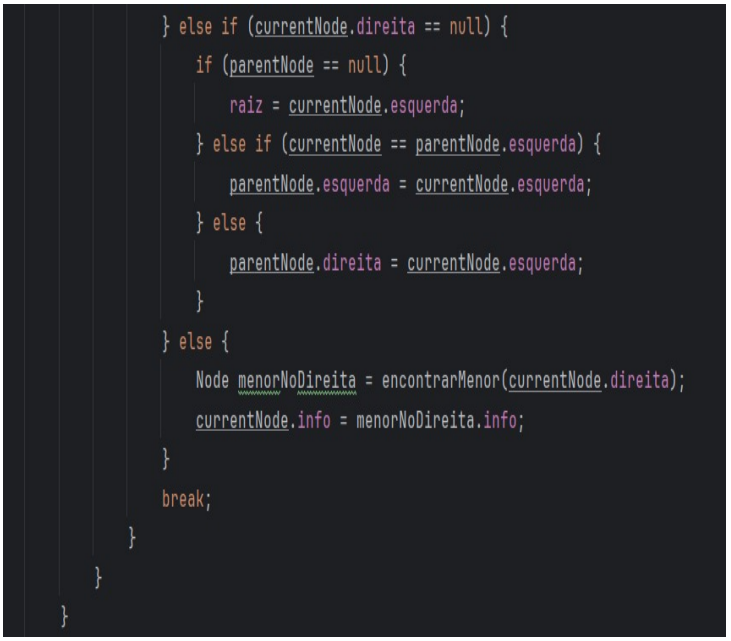
2 . Como o código funciona

```java
public void inserir(int valor) {
    if (raiz == null) {
        raiz = new Node(valor);
        return;
    }

    Node currentNode = raiz;

    while (true) {
        if (valor < currentNode.info) {
            if (currentNode.esquerda == null) {
                currentNode.esquerda = new Node(valor);
                break;
            }
            currentNode = currentNode.esquerda;
        } else if (valor > currentNode.info) {
            if (currentNode.direita == null) {
                currentNode.direita = new Node(valor);
                break;
            }
            currentNode = currentNode.direita;
        } else {

            break;
        }
    }
}
```

public void inserir(int valor): Este método é usado para inserir um novo valor na árvore binária de busca. Ele começa verificando se a raiz da árvore está vazia. Se a raiz estiver vazia, ele cria um novo nó com o valor especificado e define esse nó como a raiz da árvore. Se a raiz não estiver vazia, o método percorre a árvore da raiz até encontrar um local adequado para inserir o novo valor. Ele faz isso comparando o valor com o valor do nó atual e decidindo se deve ir para a esquerda ou para a direita até encontrar um local vazio para inserção.



```java
public void remover(int valor) {
    Node currentNode = raiz;
    Node parentNode = null;

    while (currentNode != null) {
        if (valor < currentNode.info) {
            parentNode = currentNode;
            currentNode = currentNode.esquerda;
        } else if (valor > currentNode.info) {
            parentNode = currentNode;
            currentNode = currentNode.direita;
        } else {
            if (currentNode.esquerda == null && currentNode.direita == nul
                if (parentNode == null) {
                    raiz = null;
                } else if (currentNode == parentNode.esquerda) {
                    parentNode.esquerda = null;
                } else {
                    parentNode.direita = null;
                }
            } else if (currentNode.esquerda == null) {
                if (parentNode == null) {
                    raiz = currentNode.direita;
                } else if (currentNode == parentNode.esquerda) {
                    parentNode.esquerda = currentNode.direita;
                } else {
                    parentNode.direita = currentNode.direita;
                }
```



```java
            } else if (currentNode.direita == null) {
                if (parentNode == null) {
                    raiz = currentNode.esquerda;
                } else if (currentNode == parentNode.esquerda) {
                    parentNode.esquerda = currentNode.esquerda;
                } else {
                    parentNode.direita = currentNode.esquerda;
                }
            } else {
                Node menorNoDireita = encontrarMenor(currentNode.direita);
                currentNode.info = menorNoDireita.info;
            }
            break;
        }
    }
}
```

public void remover(int valor): Este método é usado para remover um valor específico da árvore binária de busca. Ele começa na raiz da árvore e percorre a árvore enquanto compara o valor desejado com o valor do nó atual. Existem quatro cenários possíveis para a remoção:

Caso 1: O nó a ser removido não tem filhos. Nesse caso, o nó é removido diretamente.
Caso 2: O nó a ser removido tem apenas um filho (esquerdo ou direito). O nó é substituído por seu filho.
Caso 3: O nó a ser removido tem dois filhos. Nesse caso, o valor do nó é substituído pelo menor valor encontrado na subárvore direita do nó a ser removido, e o nó com o menor valor é removido.
Caso 4: O nó a ser removido é a raiz da árvore. Nesse caso, a raiz é substituída pela menor subárvore da direita e o nó com o menor valor é removido.

```
private Node encontrarMenor(Node node) {
    while (node.esquerda != null) {
        node = node.esquerda;
    }
    return node;
}
```

private Node encontrarMenor(Node node): Este é um método auxiliar que encontra o nó com o menor valor em uma árvore. Ele começa na raiz da árvore especificada (geralmente a subárvore direita do nó a ser removido) e percorre a árvore até encontrar o nó com o menor valor, que será o nó mais à esquerda da árvore.

```
public class Node {
    19 usages
    Node esquerda;
    9 usages
    int info;
    15 usages
    Node direita;

    3 usages  Victor S. Camargo
    public Node(int info) {
        this.info = info;
        esquerda = null;
        direita = null;
    }
```

Esse código representa a definição de uma classe chamada Node (nó em português) que é usada para construir uma árvore binária. Aqui está uma descrição dos elementos do código:

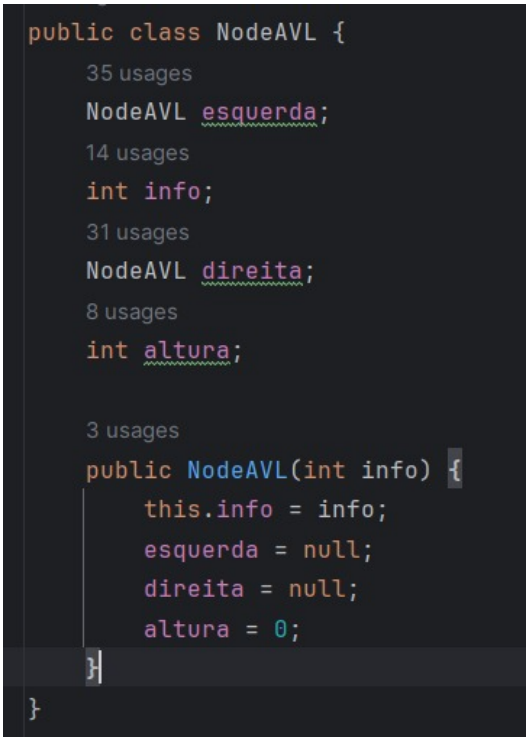
Node esquerda: Isso cria uma variável chamada esquerda que é do tipo Node, ou seja, é uma referência para outro nó na árvore que representa o filho esquerdo deste nó.

int info: Isso cria uma variável chamada info que armazena um valor inteiro. Esse valor inteiro representa o dado armazenado no nó da árvore.

Node direita: Isso cria uma variável chamada direita que é do tipo Node, e é uma referência para outro nó na árvore que representa o filho direito deste nó.

O construtor public Node(int info) é usado para criar um novo nó da árvore. Quando um nó é criado, ele recebe um valor inteiro (info) que é armazenado no nó. Inicialmente, as referências esquerda e direita são definidas como null, indicando que o novo nó não possui filhos.

Essa classe Node é fundamental para a construção de uma árvore binária, onde cada nó possui um valor inteiro e pode ter até dois filhos (um à esquerda e outro à direita). Os nós são organizados de tal forma que os valores menores do que o valor do nó atual são armazenados no filho esquerdo, e os valores maiores são armazenados no filho direito, criando assim uma estrutura de árvore.



NodeAVL esquerda: Isso cria uma variável chamada esquerda que é do tipo NodeAVL, ou seja, é uma referência para outro nó na árvore que representa o filho esquerdo deste nó.

int info: Isso cria uma variável chamada info que armazena um valor inteiro. Esse valor inteiro representa o dado armazenado no nó da árvore.

NodeAVL direita: Isso cria uma variável chamada direita que é do tipo NodeAVL, e é uma referência para outro nó na árvore que representa o filho direito deste nó.

int altura: Isso cria uma variável chamada altura que armazena a altura do nó na árvore AVL. A altura de um nó é o comprimento do caminho mais longo daquele nó até uma folha na árvore.

```
public void inserir(int valor) {
    if (raiz == null) {
        raiz = new NodeAVL(valor);
        return;
    }

    NodeAVL currentNode = raiz;
    NodeAVL parentNode = null;

    while (currentNode != null) {
        parentNode = currentNode;

        if (valor < currentNode.info) {
            currentNode = currentNode.esquerda;
        } else {
            currentNode = currentNode.direita;
        }
    }

    if (valor < parentNode.info) {
        parentNode.esquerda = new NodeAVL(valor);
    } else {
        parentNode.direita = new NodeAVL(valor);
    }
```

```
    currentNode = raiz;
    while (currentNode != null) {
        currentNode.altura = 1 + max(altura(currentNode.esquerda), altura(currentNode.direita));
        currentNode = balancear(currentNode, valor);

        if (valor < currentNode.info) {
            currentNode = currentNode.esquerda;
        } else {
            currentNode = currentNode.direita;
        }
    }
}
```

public void inserir(int valor): Este método é usado para inserir um novo valor na árvore AVL. Ele começa verificando se a raiz da árvore está vazia. Se a raiz estiver vazia, ele cria um novo nó com o valor especificado e define esse nó como a raiz da árvore. Caso contrário, ele percorre a árvore até encontrar um local adequado para inserir o novo valor. Ele mantém o equilíbrio da árvore durante a inserção, ajustando as alturas dos nós e realizando rotações quando necessário.

```
2 usages
public boolean buscar(int valor) {
    NodeAVL currentNode = raiz;
    while (currentNode != null) {
        if (currentNode.info == valor) {
            return true;
        } else if (valor < currentNode.info) {
            currentNode = currentNode.esquerda;
        } else {
            currentNode = currentNode.direita;
        }
    }
    return false;
}
```

public boolean buscar(int valor): Este método é usado para buscar um valor na árvore AVL. Ele começa na raiz da árvore e percorre a árvore enquanto compara o valor desejado com o valor do nó atual. Se encontrar o valor desejado, retorna true, indicando que o valor foi encontrado. Caso contrário, segue para a subárvore esquerda ou direita com base na comparação e continua a busca até encontrar o valor desejado ou atingir um nó nulo.

```java
public void remover(int valor) {
    if (raiz == null) {
        return;
    }

    NodeAVL currentNode = raiz;
    NodeAVL parentNode = null;

    while (currentNode != null) {
        if (valor < currentNode.info) {
            parentNode = currentNode;
            currentNode = currentNode.esquerda;
        } else if (valor > currentNode.info) {
            parentNode = currentNode;
            currentNode = currentNode.direita;
        } else {
            if (currentNode.esquerda == null || currentNode.direita == null) {
                NodeAVL temp;
                if (currentNode.esquerda != null) {
                    temp = currentNode.esquerda;
                } else {
                    temp = currentNode.direita;
                }

                if (parentNode == null) {
                    raiz = temp;
                } else if (parentNode.esquerda == currentNode) {
                    parentNode.esquerda = temp;
                } else {
                    parentNode.direita = temp;
                }
```

```java
            } else {
                NodeAVL temp = encontrarMenor(currentNode.direita);
                currentNode.info = temp.info;
                valor = temp.info;
                parentNode = currentNode;
                currentNode = currentNode.direita;
            }
        }
    }

    currentNode = raiz;
    while (currentNode != null) {
        int alturaEsquerda;
        int alturaDireita;

        if (currentNode.esquerda != null) {
            alturaEsquerda = currentNode.esquerda.altura;
        } else {
            alturaEsquerda = -1;
        }

        if (currentNode.direita != null) {
            alturaDireita = currentNode.direita.altura;
        } else {
            alturaDireita = -1;
        }

        if (alturaEsquerda > alturaDireita) {
            currentNode.altura = 1 + alturaEsquerda;
        } else {
            currentNode.altura = 1 + alturaDireita;
        }
```

```java
        currentNode = balancear(currentNode, valor);

        if (valor < currentNode.info) {
            currentNode = currentNode.esquerda;
        } else {
            currentNode = currentNode.direita;
        }
    }
```

public void remover(int valor): Este método é usado para remover um valor específico da árvore AVL. Ele começa a busca pelo valor a ser removido da mesma forma que o método de busca. Existem quatro cenários possíveis para a remoção, e o método executa as operações apropriadas para manter a árvore balanceada. Isso inclui a atualização das alturas dos nós e a realização de rotações quando necessário

```java
private int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

A função max compara dois números inteiros a e b. Se a for maior que b, ela retorna a; caso contrário, retorna b, retornando assim o valor máximo entre os dois números fornecidos como entrada.

```
1 usage
private NodeAVL encontrarMenor(NodeAVL node) {
    while (node.esquerda != null) {
        node = node.esquerda;
    }
    return node;
}
```

private NodeAVL encontrarMenor(NodeAVL node): Este é um método auxiliar que encontra o nó com o menor valor em uma árvore. Ele começa na raiz da árvore especificada (geralmente a subárvore direita do nó a ser removido) e percorre a árvore até encontrar o nó com o menor valor, que será o nó mais à esquerda da árvore.

```
private int altura(NodeAVL node) {
    if (node == null) {
        return -1;
    }
    return node.altura;
}
```

private int altura(NodeAVL node): Este método calcula a altura de um nó da árvore. A altura de um nó é a maior altura entre suas subárvores esquerda e direita

```
private int fatorBalanceamento(NodeAVL node) {
    if (node == null) {
        return 0;
    }
    return altura(node.esquerda) - altura(node.direita);
}
```

private int fatorBalanceamento(NodeAVL node): Este método calcula o fator de balanceamento de um nó, que é a diferença entre a altura da subárvore esquerda e a altura da subárvore direita. Um fator de balanceamento fora do intervalo [-1, 1] indica um desequilíbrio que precisa ser corrigido.

```java
private NodeAVL rotacaoEsquerda(NodeAVL x) {
    if (x == null || x.direita == null) {
        return x;
    }

    NodeAVL y = x.direita;
    NodeAVL T2 = y.esquerda;


    y.esquerda = x;
    x.direita = T2;


    x.altura = 1 + Math.max(altura(x.esquerda), altura(x.direita));
    y.altura = 1 + Math.max(altura(y.esquerda), altura(y.direita));

    return y;
}
```

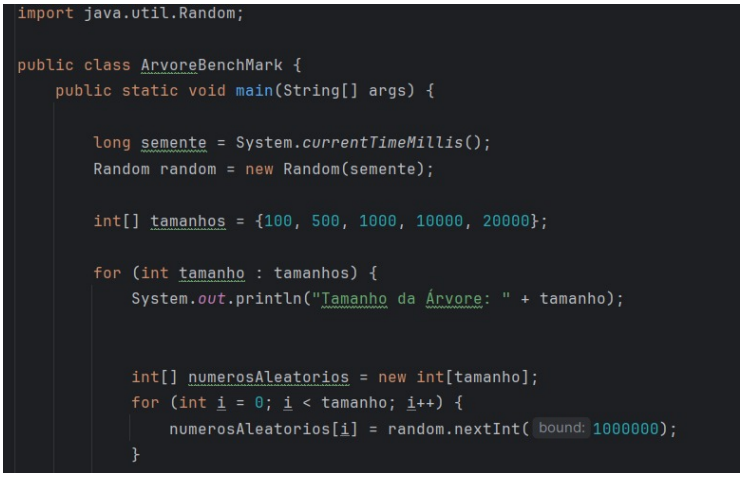
private NodeAVL rotacaoEsquerda(NodeAVL x): Este método realiza uma rotação à esquerda em torno de um nó x. Essa rotação é usada para corrigir desequilíbrios quando um nó tem um fator de balanceamento positivo e sua subárvore esquerda tem um fator de balanceamento positivo ou zero.

```java
private NodeAVL rotacaoDireita(NodeAVL x) {
    if (x == null || x.esquerda == null) {
        return x;
    }

    NodeAVL y = x.esquerda;
    NodeAVL T2 = y.direita;


    y.direita = x;
    x.esquerda = T2;


    x.altura = 1 + Math.max(altura(x.esquerda), altura(x.direita));
    y.altura = 1 + Math.max(altura(y.esquerda), altura(y.direita));

    return y;
}
```

private NodeAVL rotacaoDireita(NodeAVL x): Este método realiza uma rotação à direita em torno de um nó x. Essa rotação é usada para corrigir desequilíbrios quando um nó tem um fator de balanceamento negativo e sua subárvore direita tem um fator de balanceamento negativo ou zero.

```java
private NodeAVL balancear(NodeAVL node, int valor) {
    if (node == null) {
        return node;
    }

    int fb = fatorBalanceamento(node);


    if (fb > 1) {
        if (valor < node.esquerda.info) {
            return rotacaoDireita(node);
        } else {
            node.esquerda = rotacaoEsquerda(node.esquerda);
            return rotacaoDireita(node);
        }
    }


    if (fb < -1) {
        if (valor > node.direita.info) {
            return rotacaoEsquerda(node);
        } else {
            node.direita = rotacaoDireita(node.direita);
            return rotacaoEsquerda(node);
        }
    }

    return node;
}
```

private NodeAVL balancear(NodeAVL node, int valor): Este método é responsável por verificar e corrigir o balanceamento de um nó após uma inserção ou remoção. Ele verifica o fator de balanceamento do nó e, se necessário, realiza rotações para manter a árvore equilibrada.



```
import java.util.Random;

public class ArvoreBenchMark {
    public static void main(String[] args) {

        long semente = System.currentTimeMillis();
        Random random = new Random(semente);

        int[] tamanhos = {100, 500, 1000, 10000, 20000};

        for (int tamanho : tamanhos) {
            System.out.println("Tamanho da Árvore: " + tamanho);


            int[] numerosAleatorios = new int[tamanho];
            for (int i = 0; i < tamanho; i++) {
                numerosAleatorios[i] = random.nextInt( bound: 1000000);
            }
```

Nesta parte inicial do código, estamos importando a classe Random para gerar números aleatórios. Em seguida, obtemos uma semente para o gerador de números aleatórios com base no tempo atual, o que garante que os números aleatórios gerados em diferentes execuções sejam diferentes. Definimos um array tamanhos com os tamanhos das árvores que serão testadas (100, 500, 1000, 10000 e 20000). Em seguida, começamos um loop que percorre os diferentes tamanhos de árvore.

Dentro do loop, geramos um conjunto de números aleatórios usando a semente. Esses números aleatórios serão usados para inserir valores nas árvores a serem testadas.

```
ArvoreBinariaAVL arvoreAVL = new ArvoreBinariaAVL();
long startTime = System.nanoTime();

for (int numero : numerosAleatorios) {
    arvoreAVL.inserir(numero);
}

long endTime = System.nanoTime();
System.out.println("Tempo de Inserção AVL: " + (endTime - startTime) + " ns");
```

Nesta parte, criamos uma instância de ArvoreBinariaAVL que é uma árvore AVL e medimos o tempo necessário para inserir os números aleatórios gerados anteriormente nessa árvore. O tempo de início (startTime) é obtido antes da inserção, e o tempo de término (endTime) é obtido após a inserção. A diferença entre endTime e startTime nos dá o tempo de inserção em nanossegundos.

```
ArvoreBinaria arvoreNaoBalanceada = new ArvoreBinaria();
startTime = System.nanoTime();

for (int numero : numerosAleatorios) {
    arvoreNaoBalanceada.inserir(numero);
}

endTime = System.nanoTime();
System.out.println("Tempo de Inserção Árvore Não Balanceada: " + (endTime - startTime) + " ns");
```

Aqui, criamos uma instância de ArvoreBinaria, que é uma árvore binária não balanceada, e medimos o tempo necessário para inserir os mesmos números aleatórios nessa árvore. O processo é semelhante ao anterior, medindo o tempo de início antes da inserção e o tempo de término após a inserção.

Essas partes do código realizam a geração de números aleatórios, a criação de árvores AVL e árvores binárias não balanceadas, e a medição do tempo de inserção nessas árvores.

```
int chaveBusca = numerosAleatorios[random.nextInt(tamanho)];

startTime = System.nanoTime();
arvoreAVL.buscar(chaveBusca);
endTime = System.nanoTime();
System.out.println("Tempo de Busca AVL: " + (endTime - startTime) + " ns");

startTime = System.nanoTime();
arvoreNaoBalanceada.buscar(chaveBusca);
endTime = System.nanoTime();
System.out.println("Tempo de Busca Árvore Não Balanceada: " + (endTime - startTime) + " ns");
```

Nesta parte, escolhemos aleatoriamente uma chave de busca a partir dos números gerados e medimos o tempo necessário para realizar a busca dessa chave em ambas as árvores (AVL e não balanceada). Novamente, usamos System.nanoTime() para registrar o tempo de início e fim da operação de busca e exibimos os resultados.

```
        int chaveRemocao = numerosAleatorios[random.nextInt(tamanho)];

        startTime = System.nanoTime();
        arvoreAVL.remover(chaveRemocao);
        endTime = System.nanoTime();
        System.out.println("Tempo de Remoção AVL: " + (endTime - startTime) + " ns");

        startTime = System.nanoTime();
        arvoreNaoBalanceada.remover(chaveRemocao);
        endTime = System.nanoTime();
        System.out.println("Tempo de Remoção Árvore Não Balanceada: " + (endTime - startTime) + " ns");

        System.out.println();
    }
}
```

Nesta última parte do código, selecionamos aleatoriamente uma chave para remoção e medimos o tempo necessário para remover essa chave em ambas as árvores (AVL e não balanceada). O processo é semelhante ao das etapas anteriores, onde registramos os tempos de início e fim da operação de remoção e exibimos os resultados.

3 . Análise Crítica

Através da comparação do tempo de inserção, busca e remoção, vimos que a árvore balanceada no geral tem melhor desempenho para busca e árvore não balanceada é mais rápida na inserção e na remoção, isso se deve ao fato de que a árvore balanceada tem que realizar mais passos em comparação com a não balanceada, mas podem haver casos em que a árvore balanceada seja mais lenta para busca e mais rápida para remoção e inserção, o mesmo vale para a árvore não balanceada.