ALUNO: Gabriel Felipe Jess Meira.

RELATÓRIO SOBRE ÁRVORES

ÁRVORE BINÁRIA

Implementei a árvore binária usando três classes: Main, Node e ArvoreBinaria.

CLASSE Node

Na classe Node, temos a definição dos nós/ponteiros com sua informação.

Atributos:

```
//Atributo da informação do nó
3 usages
private int Informacao;

//Atributo do filho a esquerda do nó
3 usages
private Node FEsquerda;

//Atributo do filho a direita do nó
3 usages
private Node FDireita;
```

Atributo de informação para armazenar a informação do nó em questão.

Atributo de FEsquerda para se referenciar ao filho a esquerda do nó em questão.

Atributo de FDireita para se referenciar ao filho a direita do nó em questão.

Método Construtor:

```java
//Método Construtor
3 usages
public Node(int informacao) {
    this.Informacao = informacao;
    this.FEsquerda = null;
    this.FDireita = null;
}
```

O método construtor recebe apenas o parâmetro de informação para realizar a criação do nó, pois os filhos são definidos a partir do método set de cada um nas operações de inserção e remoção de um nó na árvore binária.

Getters e Setters:

```java
//Função que retorna a informação do nó em questão
20 usages
public int getInformacao() {
    return this.Informacao;
}

//Função que define a informação do nó em questão
no usages
public void setInformacao(Integer informacao) {
    this.Informacao = informacao;
}

//Função que retorna o filho a esquerda do nó em questão
15 usages
public Node getFEsquerda() {
    return this.FEsquerda;
}

//Função que define o filho a esquerda do nó em questão
7 usages
public void setFEsquerda(Node filhoEsquerda) {
    this.FEsquerda = filhoEsquerda;
}

//Função que retorna o filho a direita do nó em questão
15 usages
public Node getFDireita() { return this.FDireita; }

//Função que define o filho a direita do nó em questão
7 usages
public void setFDireita(Node filhoDireita) {
    this.FDireita = filhoDireita;
}
```

Métodos que retornam e definem atributos da classe Node.

CLASSE ArvoreBinaria

Na classe ArvoreBinaria ficam os métodos de inserção, remoção, busca e impressão da árvore binária e a definição da raiz da árvore.

Atributo:

```
//Atributo da raiz
8 usages
private Node Raiz;
```

Uma árvore binária só possuí um atributo do tipo Node, a Raiz, pois a partir dela, os outros nós são referenciados.

Método Construtor:

```
//Método construtor
1 usage
public ArvoreBinaria(){
    this.Raiz = null;
}
```

O método construtor da árvore binária é definir a Raiz da árvore como null, pois a partir dos métodos de inserção e remoção, a Raiz da árvore é manipulada.

Get e Set:

```
//Método que retorna a raiz da árvore binária
15 usages
public Node getRaiz() {
    return this.Raiz;
}

//Método que define a raiz da arvore binária
5 usages
public void setRaiz(Node raiz) {
    this.Raiz = raiz;
}
```

Método que retorna e que define a Raiz da árvore binária.

Método buscar:

```java
//Método que busca um nó na árvore
2 usages
public boolean buscar(int n){
    Node atual = this.Raiz;
    while (atual != null && atual.getInformacao() != n){
        if (n >= atual.getInformacao()){
            atual = atual.getFDireita();
        }else {
            atual = atual.getFEsquerda();
        }
    }

    if (atual != null){
        if (atual.getInformacao() == n){
            return true;
        }
    }

    return false;
}
```

O método buscar, retorna um valor booleano baseado se o método encontrou ou não o nó que possuí informação equivalente ao parâmetro de entrada do tipo inteiro n.

O método começa a partir da raíz da árvore binária, e basicamente vai iterando para a esquerda ou para a direita do deste nó.

A cada iteração o nó atual é atualizado baseado no valor do parâmetro de entrada n. Se o valor de n for maior ou igual a informação do nó atual, o próximo nó a ser verificado é o nó da direita do nó atual. Caso o valor de n for menor que a informação do nó atual, o próximo nó a ser verificado é o nó da esquerda do nó atual.

A iteração só para quando o valor do nó atual for diferente de null ou quando o valor do nó atual é equivalente ao valor do parâmetro de entrada n.

Por fim, retorna true caso o nó atual não for nulo e possuir o mesmo valor de n ou retorna false caso não achar o nó.

Método insertElementoArvore:

```java
//Método para inserir um elemento/novo nó na árvore binária
13 usages
public void insertElementoArvore(int n, Node no){
    //Verifica se é a raiz
    if (no != null){
        //O nó aux é um nó de auxílio do código
        Node atual = no;
        //Verifica o lado para manipular
        //Esquerda do nó auxiliar (informação n é menor q a informação do nó auxiliar)
        if (n < atual.getInformacao()) {
            //Com filho a esquerda disponível
            if (atual.getFEsquerda() == null) {
                //Define o filho da esquerda
                Node fe = new Node(n);
                atual.setFEsquerda(fe);
            //Sem filho a esquerda disponivel
            }else{
                //Atualiza o nó auxiliar para prosseguir a execução do loop
                insertElementoArvore(n, atual.getFEsquerda());
            }
            //Direita do nó auxiliar (informação n é maior ou igual a informação do nó auxiliar)
        } else {
            //Com filho a direita disponivel
            if (atual.getFDireita() == null) {
                Node fd = new Node(n);
                atual.setFDireita(fd);
            //Sem filho a direita disponivel
            }else{
                //Atualiza o nó auxiliar para prosseguir a execução do loop
                insertElementoArvore(n, atual.getFDireita());
            }
        }
    }else{
        //Caso o nó passado ser nulo, o mesmo é a raiz
        Node raiz = new Node(n);
        this.Raiz = raiz;
    }
}
```

Este método é recursivo e possuí dois parâmetros de entrada: o número a ser inserido (n) e o nó atual que irá ser usado na execução da função.

A primeira verificação da execução do método serve para identificar se o nó atual é a raiz da árvore binária. Então basta verificarmos se o no recebido é nulo ou não, se for nulo, irá atualizar a raiz da árvore com um novo nó do valor recebido.

Caso não for nulo, é definida a variável atual que é um nó que serve para apenas visualizar melhor o parâmetro de entrada nó na lógica de inserção.

Em seguida irá verificar para qual lado o valor n vai ser enviado, ou para a esquerda(se o valor de n for < que o valor do nó atual) ou para a direita(se o valor de n for >= que o valor do nó atual).

Assim ele seleciona o lado e verifica se o nó filho do elemento atual que pertence ao lado selecionado está vazio, ou seja, verifica se o nó atual possuí filho para o lado que foi escolhido.

Se o nó atual não possuir filho no lado escolhido, então o filho deste lado é definido baseado no valor de n e a iteração é interrompida.

Caso o nó atual possuir filho no lado escolhido, então a função de inserção é chamada novamente passando os parâmetros do valor de n, que se mantém, porém o nó a ser passado para a próxima iteração é o filho do nó atual do lado escolhido(esquerda ou direita).

Método removeElemento:

```
public void removeElemento(int n, Node no, Node noP){
```

O método removeElemento recebe três parâmetros: o valor do nó a ser retirado (n), o nó atual que irá ser tratado na lógica do método(no) e o nó pai do nó atual(noP) que irá ser utilizado para reposicionar os elementos na árvore.

```
Node atual = no;
Node noPai = noP;
```

No início da execução o são definidos os nós atual e noPai, que irão facilitar o entendimento da lógica da função de remoção.

Em seguida é realizado uma verificação se o nó atual possuí o mesmo valor que o valor do nó a ser removido, se não possuir então ou seleciona o nó filho a esquerda(quando o valor de n é menor que o valor do nó atual) ou para a direita(quando o valor de n é maior que o valor do nó atual) para seguir com a iteração.

Caso possuir o nó atual possuir a mesma informação do nó atual, então é este nó que deve ser removido da árvore binária.

```
//Busca o elemento na árvore binária, passando o próximo nó a ser verificado e o pai dele
//Elemento menor do que o nó atual(esquerda)
if (n < atual.getInformacao()){
    removeElemento(n, atual.getFEsquerda(), atual);
//Elemento maior do que o nó atual(direita)
}else if (n > atual.getInformacao()){
    removeElemento(n, atual.getFDireita(), atual);
//Achou o elemento
}else{
```

Assim temos três possíveis casos de remoção: o nó atual possuí filho a esquerda, o nó atual possuí filho a direita ou o nó atual é um nó folha.

CASO 1 (Caso o nó atual possuí filho a esquerda):

```
//Atual possuí filho a esquerda
if (atual.getFEsquerda() != null){
```

Primeiro é verificado se o nó atual possuí filho a esquerda, se o mesmo possuir, então são definidas as variáveis aux e noPaiAux, que auxiliam a localizar o último nó a direita do filho da esquerda do nó atual (maior valor dos nós a esquerda do nó atual).

Quando encontrar este valor a variável aux recebe o mesmo e o noPaiAux é definido também baseado no nó pai da variável aux.

```
//Pega o nó mais a direita do filho da esquerda para substituição
Node aux = atual.getFEsquerda();
Node noPaiAux = atual;
while (aux.getFDireita() != null){
    noPaiAux = aux;
    aux = aux.getFDireita();
}
```

Em seguida o noPaiAux tem o valor do seu filho a esquerda atualizado como null, caso o valor de aux for menor que o valor do noPaiAux. Caso o valor de aux for maior ou igual ao do noPaiAux, o mesmo tem seu filho a direita atualizado como null.

```
//Remove o nó folha do nó paiAuxiliar
if (aux.getInformacao() < noPaiAux.getInformacao()){
    noPaiAux.setFEsquerda(null);
    //Atualiza o filho a direita do nó pai
}else{
    noPaiAux.setFDireita(null);
}
```

O próximo passo é atualizar os novos filhos do no aux.

O filho a direita do no aux é definido baseado no nó a direita do nó atual. E o filho a esquerda é definido baseado no filho a esquerda do nó atual, caso o valor não for igual ao nó aux, se for igual, é setado como nulo.

```
//Atualiza o nó que irá mudar de posição com o filho a direita do nó q foi removido
if (atual.getFDireita() != null){
    aux.setFDireita(atual.getFDireita());
}else{
    aux.setFDireita(null);
}

//Atualiza o nó que irá mudar de posição com o filho a esquerda do nó q foi removido
if (atual.getFEsquerda()!= null){
    if (atual.getFEsquerda().getInformacao() != aux.getInformacao()){
        aux.setFEsquerda(atual.getFEsquerda());
    }else{
        aux.setFEsquerda(null);
    }
}else{
    aux.setFEsquerda(null);
}
```

Em seguida atualiza ou o filho a direita ou esquerda do nó noPai, que é o pai do nó atual. Se o valor de atual for menor que o valor do nó pai, o filho a esquerda do nó pai será o nó aux. Se o valor do nó atual for maior ou igual, então o filho a direita do nó pai será o nó aux.

Caso o nó pai for nulo, então o nó atual é a raiz, então o nó auxiliar irá ser a nova raiz.

```
//Verifica se o nó possuí pai (Verifica se o nó em questão é a raíz da arvore binária)
if (noPai != null){
    //Atualiza o filho a esquerda do nó pai da árvore binária
    if (atual.getInformacao() < noPai.getInformacao()){
        noPai.setFEsquerda(aux);
        //Atualiza o filho a direita do nó pai
    }else{
        noPai.setFDireita(aux);
    }
}else{ //Único caso em que um nó não possuí pai é a raiz
    this.Raiz = aux;
}
```

CASO 2 (Caso o nó atual possuí filho a direita):

```
//Atual possuí filho a direita
} else if (atual.getFDireita() != null){
```

Primeiro é verificado se o nó atual possuí filho a direita, se o mesmo possuir, então são definidas as variáveis aux e noPaiAux, que auxiliam a localizar o último nó a esquerda do filho da direita do nó atual (menor valor dos nós a direita do nó atual).

Quando encontrar este valor a variável aux recebe o mesmo e o noPaiAux é definido também baseado no nó pai da variável aux.

```
//Pega o nó mais a esquerda do filho da direita para substituição
Node aux = atual.getFDireita();
Node noPaiAux = atual;
while (aux.getFEsquerda() != null){
    noPaiAux = aux;
    aux = aux.getFEsquerda();
}
```

Em seguida o noPaiAux tem o valor do seu filho a esquerda atualizado como null, caso o valor de aux for menor que o valor do noPaiAux. Caso o valor de aux for maior ou igual ao do noPaiAux, o mesmo tem seu filho a direita atualizado como null.

```
//Remove o nó folha do nó paiAuxiliar
if (aux.getInformacao() < noPaiAux.getInformacao()){
    noPaiAux.setFEsquerda(null);
    //Atualiza o filho a direita do nó pai
}else{
    noPaiAux.setFDireita(null);
}
```

O próximo passo é atualizar os novos filhos do no aux.

O filho a esquerda do no aux é definido baseado no nó a esquerda do nó atual. E o filho a direita é definido baseado no filho a direita do nó atual, caso o valor não for igual ao nó aux, se for igual, é setado como nulo.

```
//Atualiza o nó que irá mudar de posição com o filho a esquerda do nó q foi removido
if (atual.getFEsquerda() != null){
    aux.setFEsquerda(atual.getFEsquerda());
}else{
    aux.setFEsquerda(null);
}

//Atualiza o nó que irá mudar de posição com o filho a direita do nó q foi removido
if (atual.getFDireita()!= null){
    if (atual.getFDireita().getInformacao() != aux.getInformacao()){
        aux.setFDireita(atual.getFDireita());
    }else{
        aux.setFDireita(null);
    }
}else{
    aux.setFDireita(null);
}
```

Em seguida atualiza ou o filho a direita ou esquerda do nó noPai, que é o pai do nó atual. Se o valor de atual for menor que o valor do nó pai, o filho a esquerda do nó pai será o nó aux. Se o valor do nó atual for maior ou igual, então o filho a direita do nó pai será o nó aux.

Caso o nó pai for nulo, então o nó atual é a raiz, então o nó auxiliar irá ser a nova raiz.

```
//Verifica se o nó possuí pai (Verifica se o nó em questão é a raíz da arvore binária)
if (noPai != null) {
    //Atualiza o filho a esquerda do nó pai da árvore binária
    if (atual.getInformacao() < noPai.getInformacao()) {
        noPai.setFEsquerda(aux);
        //Atualiza o filho a direita do nó pai
    } else {
        noPai.setFDireita(aux);
    }
}else{ //Único caso em que um nó não possuí pai é a raiz
    this.Raiz = aux;
}
```

CASO 3 (Caso o nó atual é um nó folha):

Atualiza ou o filho a esquerda(caso o valor do nó atual for menor que o valor do nó pai) ou o filho a direita(caso o valor do nó atual for maior ou igual ao valor do nó pai) do nó pai como valor nulo.

Caso o nó pai for nulo, então o nó atual é a raiz, então a nova raiz será nula.

```
//Verifica se o nó possui pai (Verifica se o nó em questão é a raiz da arvore binária)
if (noPai != null) {
    //Atualiza o filho a esquerda do nó pai
    if (atual.getInformacao() < noPai.getInformacao()) {
        noPai.setFEsquerda(null);
        //Atualiza o filho a direita do nó pai
    } else {
        noPai.setFDireita(null);
    }
}else{ //Único caso em que um nó não possuí pai é a raiz
    this.Raiz = null;
}
```

Método preOrdem:

```
//Impressão em Pré-Ordem
3 usages
public void preOrdem(Node n){
    if (n != null){
        //Imprime a informação da raíz
        System.out.print(n.getInformacao() + " ");
        //Se tiver filho a esquerda chama o mesmo
        if (n.getFEsquerda() != null){
            preOrdem(n.getFEsquerda());
        }
        //Se tiver filho a direita chama o mesmo
        if (n.getFDireita() != null){
            preOrdem(n.getFDireita());
        }
    }
}
```

Imprime a árvore binária no formato pré-ordem.

Método emOrdem:

```java
//Impressão em emOrdem
3 usages
public void emOrdem(Node n){
    if (n != null){
        //Se tiver filho a esquerda chama o mesmo
        if (n.getFEsquerda() != null){
            emOrdem(n.getFEsquerda());
        }
        //Imprime a informação da raiz
        System.out.print(n.getInformacao() + " ");
        //Se tiver filho a direita chama o mesmo
        if (n.getFDireita() != null){
            emOrdem(n.getFDireita());
        }
    }
}
```

Imprime a árvore binária no formato em ordem.

Método posOrdem:

```java
//Impressão em Pós-Ordem
3 usages
public void posOrdem(Node n){
    if (n != null){
        //Se tiver filho a esquerda chama o mesmo
        if (n.getFEsquerda() != null){
            posOrdem(n.getFEsquerda());
        }
        //Se tiver filho a direita chama o mesmo
        if (n.getFDireita() != null){
            posOrdem(n.getFDireita());
        }
        //Imprime a informação da raiz
        System.out.print(n.getInformacao() + " ");
    }
}
```

Imprime a árvore binária no formato pós-ordem.

CLASSE Main:

Na classe Main, são realizadas as chamadas de métodos da classe da árvore binária e é o lugar onde o menu da aplicação está implementado.

Bibliotecas:

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;
```

As bibliotecas que foram utilizadas serviram de auxílio para receber inputs na classe main, ler arquivos txt e gerar números aleatórios.

Variáveis globais:

```
//Criação da Árvore binária
ArvoreBinaria arvoreBinaria = new ArvoreBinaria();

//Criação do gerador de números aleatórios
Random gerador = new Random();

//Criação dos scanners
Scanner sc = new Scanner(System.in);
Scanner scF;
```

Menu:

```
//Menu
int option = -1;
while (option != 0){
    System.out.println("===== Árvore Binária =====");

    //Impressão da árvore AVL nos três modos
    System.out.println();
    System.out.println("Pré-Ordem: ");
    arvoreBinaria.preOrdem(arvoreBinaria.getRaiz());
    System.out.println();

    System.out.println();
    System.out.println("Em-Ordem: ");
    arvoreBinaria.emOrdem(arvoreBinaria.getRaiz());
    System.out.println();

    System.out.println();
    System.out.println("Pós-Ordem: ");
    arvoreBinaria.posOrdem(arvoreBinaria.getRaiz());
    System.out.println();
    System.out.println("========================");

    System.out.println("0. Sair");
    System.out.println("1. Inserção");
    System.out.println("2. Remoção");
    System.out.println("3. Busca");
    System.out.println("4. Geração de árvore binária com 100 números aleatórios");
    System.out.println("5. Geração de árvore binária com 500 números aleatórios");
    System.out.println("6. Geração de árvore binária com 1000 números aleatórios");
    System.out.println("7. Geração de árvore binária com 10000 números aleatórios");
    System.out.println("8. Geração de árvore binária com 20000 números aleatórios");
    System.out.println("9. Carregar txt");
    System.out.println("Escolha uma opção: ");
    option = sc.nextInt();
```

Criação da base do menu, onde estão listadas as operações e a leitura da opção escolhida do usuário.

Definição de variáveis auxiliares do menu:

```
//Variáveis de auxílio para execução
int numero = 0;
long comeco = 0;
long fim = 0;
long tempoTotal = 0;
```

Navegação do menu:

```
//Navegação do menu
switch (option){
```

A navegação no menu ocorre baseado num switch case de opções.

Menu (Parar execução do programa):

```
case 0:
    //Encerra o programa
    System.exit( status: 0);
    break;
```

Caso a opção selecionada for 0, o programa tem sua execução interrompida.

Menu (Inserção):

```
case 1:
    //Inserção
    boolean parou = false;
    while (parou == false) {
        System.out.println();
        System.out.println("Entre com o número (-1 para parar): ");
        numero = sc.nextInt();
        comeco = System.nanoTime();
        if (numero != -1){
            arvoreBinaria.insertElementoArvore(numero, arvoreBinaria.getRaiz());
        }else {
            parou = true;
        }
        fim = System.nanoTime();
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos
        System.out.println("Tempo total de execução da inserção: " + tempoTotal + " microssegundos");
    }
    break;
```

Caso a opção selecionada for 1, então o programa inicia um loop que pede para o usuário entrar com os números a serem inseridos na árvore binária.

A cada elemento inserido pelo usuário, a classe main chama o método de inserção da classe da árvore binária com os parâmetros do número inserido pelo usuário e com a raiz da árvore binária.

O tempo de cada inserção é medido e imprimido no final da inserção do elemento no console.

Se o usuário inserir o valor -1 o programa volta para o menu de opções.

Menu (Remoção):

```
case 2:
    //Remoção
    System.out.println();
    System.out.println("Entre com o número: ");
    numero = sc.nextInt();
    comeco = System.nanoTime();
    boolean noEncontradoR = arvoreBinaria.buscar(numero);
    if (noEncontradoR == true){
        arvoreBinaria.removeElemento(numero, arvoreBinaria.getRaiz(), noP: null);
    }else{
        System.out.println("Nó não está presente na árvore para ser removido");
    }
    fim = System.nanoTime();
    tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos
    System.out.println("Tempo total de execução da remoção: " + tempoTotal + " microssegundos");
    break;
```

Caso a opção selecionada for 2, então o programa solicita o valor do nó que deve ser removido.

Assim que o usuário inserir o valor, o programa busca por este nó, se o programa encontrar o nó, então chama o método de remoção da classe ArvoreBinaria com os parâmetros de número inserido pelo usuário, com a raiz da árvore binária e com o valor null (nó pai da raiz). Se o valor do número inserido pelo usuário não for encontrado na árvore binária, avisa isto ao usuário.

O tempo de cada remoção ou tentativa de remoção é medido e imprimido no final de cada execução desta opção no console.

Menu (Busca):

```
case 3:
    //Busca
    int numeroBusca = -1;
    System.out.println("Insira um número para ser realizado a busca: ");
    numeroBusca = sc.nextInt();
    comeco = System.nanoTime();
    boolean noEncontradoB = arvoreBinaria.buscar(numeroBusca);
    if (noEncontradoB == true){
        System.out.println("Nó " + numeroBusca + " foi encontrado!");
    }else{
        System.out.println("Nó " + numeroBusca + " não foi encontrado!");
    }
    fim = System.nanoTime();
    tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos
    System.out.println("Tempo total de execução da busca: " + tempoTotal + " microssegundos");
    break;
```

Caso a opção selecionada for 3, então o programa solicita o valor do nó a ser buscado.

Assim que o valor do nó a ser buscado for informado, é criada a variável noEncontradoB, que possuí o seu valor baseado no retorno do método buscar do valor informado pelo usuário.

Caso o noEncontradoB for true, o número foi encontrado na árvore binária. Caso contrário, o número não foi encontrado na mesma.

O tempo de cada busca é medido e imprimido no final de cada execução desta opção no console.

Menu (Criação de árvore binária com 100 elementos aleatórios):

```
case 4:
    //Inserção de 100 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 100 números
        FileWriter fw = new FileWriter( fileName: "CemElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore binária
        for (int i = 0; i < 100; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            arvoreBinaria.insertElementoArvore(numero, arvoreBinaria.getRaiz());
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 100 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore binária com 100 elementos!");
    }

    break;
```

Esta opção começa criando um txt que irá armazenar os 100 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 100 elementos aleatórios na árvore binária.

O loop que gera os elementos aleatórios é basicamente um for que é executado 100 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (CemElementos.txt) e também é inserido na árvore binária.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore binária com 500 elementos aleatórios):

```
case 5:
    //Inserção de 500 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 500 números
        FileWriter fw = new FileWriter( fileName: "QuinhentosElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore binária
        for (int i = 0; i < 500; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            arvoreBinaria.insertElementoArvore(numero, arvoreBinaria.getRaiz());
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 500 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore binária com 500 elementos!");
    }

    break;
```

Esta opção começa criando um txt que irá armazenar os 500 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 500 elementos aleatórios na árvore binária.

O loop que gera os elementos aleatórios é basicamente um for que é executado 500 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (QuinhentosElementos.txt) e também é inserido na árvore binária.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore binária com 1000 elementos aleatórios):

```
case 6:
    //Inserção de 1000 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 1000 números
        FileWriter fw = new FileWriter( fileName: "MilElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore binária
        for (int i = 0; i < 1000; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            arvoreBinaria.insertElementoArvore(numero, arvoreBinaria.getRaiz());
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 1000 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore binária com 1000 elementos!");
    }

    break;
```

Esta opção começa criando um txt que irá armazenar os 1000 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 1000 elementos aleatórios na árvore binária.

O loop que gera os elementos aleatórios é basicamente um for que é executado 1000 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (MilElementos.txt) e também é inserido na árvore binária.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore binária com 10000 elementos aleatórios):

```
case 7:
    //Inserção de 10000 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 10000 números
        FileWriter fw = new FileWriter( fileName: "DezMilElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore binária
        for (int i = 0; i < 10000; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            arvoreBinaria.insertElementoArvore(numero, arvoreBinaria.getRaiz());
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 10000 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore binária com 10000 elementos!");
    }

    break;
```

Esta opção começa criando um txt que irá armazenar os 10000 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 10000 elementos aleatórios na árvore binária.

O loop que gera os elementos aleatórios é basicamente um for que é executado 10000 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (DezMilElementos.txt) e também é inserido na árvore binária.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore binária com 20000 elementos aleatórios):

```
case 8:
    //Inserção de 20000 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 20000 números
        FileWriter fw = new FileWriter( fileName: "VinteMilElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore binária
        for (int i = 0; i < 20000; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            arvoreBinaria.insertElementoArvore(numero, arvoreBinaria.getRaiz());
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 20000 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore binária com 20000 elementos!");
    }

    break;
```

Esta opção começa criando um txt que irá armazenar os 20000 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 20000 elementos aleatórios na árvore binária.

O loop que gera os elementos aleatórios é basicamente um for que é executado 20000 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (VinteMilElementos.txt) e também é inserido na árvore binária.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Leitura de TXT):

```java
case 9:
    //Leitura de arquivo TXT com os dados
    Path caminho100 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreBinaria\\CemElementos.txt");
    Path caminho500 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreBinaria\\QuinhentosElementos.txt");
    Path caminho1000 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreBinaria\\MilElementos.txt");
    Path caminho10000 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreBinaria\\DezMilElementos.txt");
    Path caminho20000 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreBinaria\\VinteMilElementos.txt");

    //Opção de txt a ser lido:
    int opcaoTxt = -1;

    System.out.println();
    System.out.println("1 - 100 Elementos");
    System.out.println("2 - 500 Elementos");
    System.out.println("3 - 1000 Elementos");
    System.out.println("4 - 10000 Elementos");
    System.out.println("5 - 20000 Elementos");
    System.out.println("Escolha um txt a ser lido:");

    //Vetor para armazenar os valores
    ArrayList<Integer> numerosTxt = new ArrayList<>();

    //Leitura de opção TXT
    opcaoTxt = sc.nextInt();
```

Caso a opção selecionada for 9, o programa salva o caminho de cada documento TXT em variáveis separadas para cada um.

Em seguida é iniciado um novo menu de seleção de leitura de txt, baseado na escolha do usuário o txt escolhido é lido pelo programa.

```java
case 1:
    //100 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 100 elementos
        scF = new Scanner(caminho100);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreBinaria.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            arvoreBinaria.insertElementoArvore(i, arvoreBinaria.getRaiz());
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 100 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 100 números!");
    }
    break;
```

A lógica de leitura é basicamente a leitura até a última linha do arquivo do TXT selecionado pelo usuário dentro de um loop while que salva os valores lidos em um Arraylist de inteiros.

Em seguida limpa a árvore binária para criar uma nova baseado nos 100 elementos novos.

Assim que limpar que a árvore binária for “zerada”, a mesma recebe a inserção dos novos valores que estão no Arraylist de leitura do arquivo do txt.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

A lógica é a mesma para as outras opções de leitura ->

```
case 2:
    //500 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 500 elementos
        scF = new Scanner(caminho500);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreBinaria.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            arvoreBinaria.insertElementoArvore(i, arvoreBinaria.getRaiz());
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 500 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 500 números!");
    }
    break;
```

```java
case 3:
    //1000 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 1000 elementos
        scF = new Scanner(caminho1000);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreBinaria.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            arvoreBinaria.insertElementoArvore(i, arvoreBinaria.getRaiz());
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 1000 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 1000 números!");
    }
    break;
```

```java
case 4:
    //10000 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 10000 elementos
        scF = new Scanner(caminho10000);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreBinaria.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            arvoreBinaria.insertElementoArvore(i, arvoreBinaria.getRaiz());
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 10000 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 10000 números!");
    }
    break;
```

```java
case 5:
    //20000 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 20000 elementos
        scF = new Scanner(caminho20000);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreBinaria.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            arvoreBinaria.insertElementoArvore(i, arvoreBinaria.getRaiz());
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 20000 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 20000 números!");
    }
    break;
```

ÁRVORE AVL

Implementei a árvore binária usando três classes: Main, Node e ArvoreAVL.

CLASSE Node

Na classe Node, temos a definição dos nós/ponteiros com sua informação.

Atributos:

```
//Atributo da informação do nó
3 usages
private int Informacao;

//Atributo que se referencia ao nó pai
3 usages
private Node NoPai;

//Atributo do filho a esquerda do nó
3 usages
private Node FEsquerda;

//Atributo do filho a direita do nó
3 usages
private Node FDireita;
```

Atributo de informação para armazenar a informação do nó em questão.

Atributo de NoPai para se referenciar ao no pai do nó em questão.

Atributo de FEsquerda para se referenciar ao filho a esquerda do nó em questão.

Atributo de FDireita para se referenciar ao filho a direita do nó em questão.

Método Construtor:

```java
//Método Construtor
7 usages
public Node(int informacao) {
    this.Informacao = informacao;
    this.NoPai      = null;
    this.FEsquerda = null;
    this.FDireita = null;
}
```

O método construtor recebe apenas o parâmetro de informação para realizar a criação do nó, pois o pai e os filhos são definidos a partir do método set de cada um nas operações de inserção, rotações e remoção de um nó na árvore AVL.

Getters e Setters:

```java
//Função que retorna a informação do nó em questão
27 usages
public int getInformacao() {
    return this.Informacao;
}

//Função que define a informação do nó em questão
no usages
public void setInformacao(Integer informacao) {
    this.Informacao = informacao;
}

//Função que retorna o filho a esquerda do nó em questão
62 usages
public Node getFEsquerda() {
    return this.FEsquerda;
}

//Função que define o filho a esquerda do nó em questão
23 usages
public void setFEsquerda(Node filhoEsquerda) {
    this.FEsquerda = filhoEsquerda;
}

//Função que retorna o filho a direita do nó em questão
66 usages
public Node getFDireita() {
    return this.FDireita;
}

//Função que define o filho a direita do nó em questão
23 usages
public void setFDireita(Node filhoDireita) {
    this.FDireita = filhoDireita;
}
```

```java
//Função que retorna o nó pai do nó em questão
15 usages
public Node getNoPai() {
    return NoPai;
}

//Função que define o nó pai do nó em questão
32 usages
public void setNoPai(Node noPai) {
    NoPai = noPai;
}
```

Métodos que retornam e definem atributos da classe Node.

CLASSE ArvoreAVL

Na classe ArvoreAVL ficam os métodos de inserção, remoção, busca, verificação de balanceamento, rotações e impressão da árvore AVL e a definição da raiz da árvore.

Atributo:

```java
//Atributo da raiz
12 usages
private Node Raiz;
```

Uma árvore binária só possuí um atributo do tipo Node, a Raiz, pois a partir dela, os outros nós são referenciados.

Get e Set:

```java
//Método que retorna a raiz da árvore AVL
11 usages
public Node getRaiz() {
    return this.Raiz;
}

//Método para definir a raiz da árvore AVL
6 usages
public void setRaiz(Node raiz) {
    this.Raiz = raiz;
}
```

Método que retorna e que define a Raiz da árvore AVL.

Método altura:

```java
//Método que retorna a altura de um nó da árvore AVL
6 usages
public int altura(Node no){
    if(no == null){
        return -1;
    }
    int esquerda = altura(no.getFEsquerda());
    int direita = altura(no.getFDireita());
    if(esquerda > direita){
        return 1 + esquerda;
    }
    return 1 + direita;
}
```

Este método é recursivo e retorna a altura do nó em questão, percorrendo os filhos a esquerda e a direita para realizar o cálculo da altura.

Método buscar:

```java
//Método que busca um nó na árvore
2 usages
public boolean buscar(int n){
    Node atual = this.Raiz;
    while (atual != null && atual.getInformacao() != n){
        if (n >= atual.getInformacao()){
            atual = atual.getFDireita();
        }else {
            atual = atual.getFEsquerda();
        }
    }

    if (atual != null){
        if (atual.getInformacao() == n){
            return true;
        }
    }

    return false;
}
```

O método buscar, retorna um valor booleano baseado se o método encontrou ou não o nó que possuí informação equivalente ao parâmetro de entrada do tipo inteiro n.

O método começa a partir da raíz da árvore AVL, e basicamente vai iterando para a esquerda ou para a direita do deste nó.

A cada iteração o nó atual é atualizado baseado no valor do parâmetro de entrada n. Se o valor de n for maior ou igual a informação do nó atual, o próximo nó a ser verificado é o nó da direita do nó atual. Caso o valor de n for menor que a informação do nó atual, o próximo nó a ser verificado é o nó da esquerda do nó atual.

A iteração só para quando o valor do nó atual for diferente de null ou quando o valor do nó atual é equivalente ao valor do parâmetro de entrada n.

Por fim, retorna true caso o nó atual não for nulo e possuir o mesmo valor de n ou retorna false caso não achar o nó.

Método insertElementoArvore:

```java
//Método para inserir um elemento/novo nó na árvore AVL
9 usages
public void insertElementoArvore(Node noInserido, Node noAtual){
    //Verifica se é a raiz
    if (noAtual != null){
        //O nó aux é um nó de auxílio do código
        Node atual = noAtual;
        //Verifica o lado para manipular
        //Esquerda do nó auxiliar (informação n é menor q a informação do nó auxiliar)
        if (noInserido.getInformacao() < atual.getInformacao()) {
            //Com filho a esquerda disponivel
            if (atual.getFEsquerda() == null) {
                //Define o filho da esquerda
                atual.setFEsquerda(noInserido);
                noInserido.setNoPai(atual);
                //Sem filho a esquerda disponivel
            }else{
                //Atualiza o nó auxiliar para prosseguir a execução do loop
                insertElementoArvore(noInserido, atual.getFEsquerda());
            }
            //Direita do nó auxiliar (informação n é maior ou igual a informação do nó auxiliar)
        } else {
            //Com filho a direita disponivel
            if (atual.getFDireita() == null) {
                atual.setFDireita(noInserido);
                noInserido.setNoPai(atual);
                //Sem filho a direita disponivel
            }else{
                //Atualiza o nó auxiliar para prosseguir a execução do loop
                insertElementoArvore(noInserido, atual.getFDireita());
            }
        }
    }else{
        //Caso o nó passado ser nulo, o mesmo é a raiz
        this.Raiz = noInserido;
    }
}
```

Este método é recursivo e possuí dois parâmetros de entrada: o nó a ser inserido (noInserio) e o nó atual (noAtual) que irá ser usado na execução da função.

A primeira verificação da execução do método serve para identificar se o nó atual é a raiz da árvore binária. Então basta verificarmos se o parâmetro noAtual é nulo ou não, se for nulo, irá atualizar a raiz da árvore com o novo nó inserido.

Caso não for nulo, é definida a variável atual que é um nó que serve para apenas visualizar melhor o parâmetro de entrada noAtual na lógica de inserção.

Em seguida irá verificar para qual lado o nó a ser inserido vai ser enviado, ou para a esquerda(se o valor do nó inserido for < que o valor do nó atual) ou para a direita(se o valor do nó inserido for >= que o valor do nó atual).

Assim ele seleciona o lado e verifica se o nó filho do elemento atual que pertence ao lado selecionado está vazio, ou seja, verifica se o nó atual possuí filho para o lado que foi escolhido.

Se o nó atual não possuir filho no lado escolhido, então o filho deste lado é definido baseado no nó inserido e o pai do nó inserido é definido baseado no valor do nó atual, em seguida a iteração é interrompida.

Caso o nó atual possuir filho no lado escolhido, então a função de inserção é chamada novamente passando os parâmetros do nó a ser inserido, que se mantém, porém o nó atual a ser passado para a próxima iteração é o filho do nó atual do lado escolhido(esquerda ou direita).

Método removeElemento:

```
public void removeElemento(int n, Node no, Node noP){
    Node atual = no;
    Node noPai = noP;
    //Busca o elemento na árvore binária, passando o próximo nó a ser verificado e o pai dele
    //Elemento menor do que o nó atual(esquerda)
    if (n < atual.getInformacao()){
        removeElemento(n, atual.getFEsquerda(), atual);
        //Elemento maior do que o nó atual(direita)
    }else if (n > atual.getInformacao()){
        removeElemento(n, atual.getFDireita(), atual);
        //Achou o elemento
    }else{
```

Este método é recursivo e possuí três parâmetros de entrada: n (número a ser removido), no (nó atual em que a lógica irá trabalhar) e o noP (nó pai do atual).

Primeiramente a lógica realiza uma busca na árvore AVL até encontrar o nó que deve ser removido.

Quando se encontra o nó ocorrem três casos.

CASO 1 (Caso o nó a ser removido possuí filho a esquerda):

```
//Árvore possuí filho a esquerda
if (atual.getFEsquerda() != null){
```

Então a lógica busca o nó com maior valor da subárvore da esquerda para substituir o nó removido (aux) e também busca o nó pai do valor que irá substituir o nó a ser removido(noPaiAux).

```
//Pega o nó mais a direita do filho da esquerda para substituição
Node aux = atual.getFEsquerda();
Node noPaiAux = atual;
while (aux.getFDireita() != null){
    noPaiAux = aux;
    aux = aux.getFDireita();
}
```

Se o nó aux possuir filho a esquerda, o mesmo deve ser armazenado em uma variável (filhoAux) para ser reposicionado depois.

```
//Verifica se o aux possuí filho a esquerda
Node filhoAux = null;
if (aux.getFEsquerda() != null){
    filhoAux = aux.getFEsquerda();
}
```

Em seguida o programa salva os filhos do nó que irá ser removido, para que depois, sejam realocados.

```
//Filhos do nó atual
Node filhoEAtual = atual.getFEsquerda();
Node filhoDAtual = null;
if (atual.getFDireita() != null){
    filhoDAtual = atual.getFDireita();
}
```

Depois o programa atualiza o filho a direita do nó pai do novo nó que irá substituir o nó a ser removido como nulo, ou se o filho possuir filho a esquerda, o programa atualiza o filho a direita do nó noPaiAux como o filhoAux que é o nó filho a esquerda do nó que irá substituir o nó removido.

```
//Remove o nó folha do nó paiAuxiliar
if (aux.getFEsquerda() != null){
    noPaiAux.setFDireita(filhoAux);
    filhoAux.setNoPai(noPaiAux);
}else{
    noPaiAux.setFDireita(null);
}
```

Continuando a execução, o programa atualiza o filho a direita do nó aux com o valor do filho a direita do nó que foi removido.

```
//Atualiza o nó que irá mudar de posição com o filho a direita do nó q foi removido
if (filhoDAtual != null){
    aux.setFDireita(filhoDAtual);
    filhoDAtual.setNoPai(aux);
}else{
    aux.setFDireita(null);
}
```

Depois o filho a esquerda do nó atual é atualizado seguindo a seguinte lógica:

Se o filho a esquerda do nó que foi removido, possuir um valor diferente do nó aux, então o nó a esquerda do nó aux irá ser o nó a esquerda do nó que foi removido (filhoEAtual), e filhoEAtual tem o valor de seu pai atualizado para aux. Caso contrário, o filho a esquerda de aux se mantém.

```
//Atualiza o nó que irá mudar de posição com o filho a esquerda do nó q foi removido
if (filhoEAtual != aux){
    aux.setFEsquerda(filhoEAtual);
    filhoEAtual.setNoPai(aux);
}else{
    aux.setFEsquerda(filhoAux);
}
```

Depois o programa reposiciona o nó aux ou a esquerda ou a direita no pai do nó que foi removido, e o nó pai do nó que irá substituir o nó removido (aux) é atualizado.

```
//Verifica se o nó possuí pai (Verifica se o nó em questão é a raiz da arvore AVL)
if (noPai != null){
    //Atualiza o filho a esquerda do nó pai da árvore binária
    if (atual.getInformacao() < noPai.getInformacao()){
        noPai.setFEsquerda(aux);
    //Atualiza o filho a direita do nó pai
    }else{
        noPai.setFDireita(aux);
    }
    aux.setNoPai(noPai);
}else{ //Único caso em que um nó não possuí pai é a raiz
    this.Raiz = aux;
    aux.setNoPai(null);
}
```

Em seguida o balanceamento é verificado, utilizando a seguinte lógica:

O novo nó é ou não um nó folha, se o nó não for folha, verifica o balanceamento baseado nos nós folhas das subárvores da esquerda e da direita do nó aux.

```
//Verifica Balanceamento
Node noFolha = null;
if (aux.getFEsquerda() != null || aux.getFDireita()!= null){
    //Subárvore da esquerda
    if (aux.getFEsquerda() != null){
        noFolha = aux.getFEsquerda();
        while (true){
            if (noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else if(noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else{
                break;
            }
        }
        verificacaoBalanceamento(noFolha);
    }
    //Subárvore da direita
    if (aux.getFDireita() != null) {
        noFolha = aux.getFDireita();
        while (true) {
            if (noFolha.getFEsquerda() != null) {
                noFolha = noFolha.getFEsquerda();
            } else if (noFolha.getFDireita() != null) {
                noFolha = noFolha.getFDireita();
            } else {
                break;
            }
        }
        verificacaoBalanceamento(noFolha);
    }
//Nó Folha
}else{
```

Se for um nó folha, o balanceamento é verificado baseado na subárvore da direita ou da esquerda, ou no nó noPaiAux.

```
//Nó Folha
}else{
    //Subárvore a esquerda do nó pai
    if (noPaiAux.getFEsquerda() != null){
        noFolha = noPaiAux.getFEsquerda();
        while (true){
            if (noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else if(noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else{
                break;
            }
        }
    //Subárvore a direita do nó pai
    }else if (noPaiAux.getFDireita() != null){
        noFolha = noPaiAux.getFDireita();
        while (true){
            if (noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else if(noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else{
                break;
            }
        }
    //Nó pai é um nó folha
    }else{
        noFolha = noPaiAux;
    }

    verificacaoBalanceamento(noFolha);
}
```

CASO 2 (Caso o nó a ser removido possuí filho a direita):

```
//Árvore possuí filho a direita
} else if (atual.getFDireita() != null){
```

Então a lógica busca o nó com menor valor da subárvore da direita para substituir o nó removido (aux) e também busca o nó pai do valor que irá substituir o nó a ser removido(noPaiAux).

Então a lógica busca o nó com maior valor da subárvore da esquerda para substituir o nó removido (aux) e também busca o nó pai do valor que irá substituir o nó a ser removido(noPaiAux).

```
//Pega o nó mais a esquerda do filho da direita para substituição
Node aux = atual.getFDireita();
Node noPaiAux = atual;
while (aux.getFEsquerda() != null){
    noPaiAux = aux;
    aux = aux.getFEsquerda();
}
```

Se o nó aux possuir filho a direita, o mesmo deve ser armazenado em uma variável (filhoAux) para ser reposicionado depois.

```
//Verifica se o aux possuí filho a direita
Node filhoAux = null;
if (aux.getFDireita() != null){
    filhoAux = aux.getFDireita();
}
```

Em seguida o programa salva os filhos do nó que irá ser removido, para que depois, sejam realocados.

```
//Filhos do nó atual
Node filhoDAtual = atual.getFDireita();
Node filhoEAtual = null;
if (atual.getFEsquerda() != null){
    filhoEAtual = atual.getFEsquerda();
}
```

Depois o programa atualiza o filho a esquerda do nó pai do novo nó que irá substituir o nó a ser removido como nulo, ou se o filho possuir filho a esquerda, o programa atualiza o filho a esquerda do nó noPaiAux como o filhoAux que é o nó filho a esquerda do nó que irá substituir o nó removido.

```
//Remove o nó folha do nó paiAuxiliar
if (aux.getFDireita() != null){
    noPaiAux.setFEsquerda(filhoAux);
    filhoAux.setNoPai(noPaiAux);
}else{
    noPaiAux.setFEsquerda(null);
}
```

Continuando a execução, o programa atualiza o filho a esquerda do nó aux com o valor do filho a esquerda do nó que foi removido.

```
//Atualiza o nó que irá mudar de posição com o filho a esquerda do nó q foi removido
if (filhoEAtual != null){
    aux.setFEsquerda(filhoEAtual);
    filhoEAtual.setNoPai(aux);
}else{
    aux.setFEsquerda(null);
}
```

Depois o filho a direita do nó atual é atualizado seguindo a seguinte lógica:

Se o filho a direita do nó que foi removido, possuir um valor diferente do nó aux, então o nó a direita do nó aux irá ser o nó a direita do nó que foi removido (filhoDAtual), e filhoDAtual tem o valor de seu pai atualizado para aux. Caso contrário, o filho a direita de aux se mantém.

```
//Atualiza o nó que irá mudar de posição com o filho a direita do nó q foi removido
if (filhoDAtual != aux){
    aux.setFDireita(filhoDAtual);
    filhoDAtual.setNoPai(aux);
}else{
    aux.setFDireita(filhoAux);
}
```

Depois o programa reposiciona o nó aux ou a esquerda ou a direita no pai do nó que foi removido, e o nó pai do nó que irá substituir o nó removido (aux) é atualizado.

```
//Verifica se o nó possuí pai (Verifica se o nó em questão é a raiz da arvore AVL)
if (noPai != null) {
    //Atualiza o filho a esquerda do nó pai da árvore binária
    if (atual.getInformacao() < noPai.getInformacao()) {
        noPai.setFEsquerda(aux);
        //Atualiza o filho a direita do nó pai
    } else {
        noPai.setFDireita(aux);
    }
    aux.setNoPai(noPai);
}else{ //Único caso em que um nó não possuí pai é a raiz
    this.Raiz = aux;
    aux.setNoPai(null);
}
```

Em seguida o balanceamento é verificado, utilizando a seguinte lógica:

O novo nó é ou não um nó folha, se o nó não for folha, verifica o balanceamento baseado nos nós folhas das subárvores da esquerda e da direita do nó aux.

```
//Verifica balanceamento
Node noFolha = null;
if (aux.getFEsquerda() != null || aux.getFDireita() != null){
    //Subárvore da direita
    if (aux.getFDireita() != null){
        noFolha = aux.getFDireita();
        while (true){
            if (noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else if(noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else{
                break;
            }
        }
        verificacaoBalanceamento(noFolha);
    }

    //Subárvore da esquerda
    if (aux.getFEsquerda() != null){
        noFolha = aux.getFEsquerda();
        while (true){
            if (noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else if(noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else{
                break;
            }
        }
        verificacaoBalanceamento(noFolha);
    }
//Nó Folha
}else{
```

Se for um nó folha, o balanceamento é verificado baseado na subárvore da direita ou da esquerda, ou no nó noPaiAux.

```
//Nó Folha
}else{
    //Subárvore da esquerda do nó pai
    if (noPaiAux.getFEsquerda() != null){
        noFolha = noPaiAux.getFEsquerda();
        while (true){
            if (noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else if(noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else{
                break;
            }
        }
    //Subárvore da direita do nó pai
    }else if (noPaiAux.getFDireita() != null){
        noFolha = noPaiAux.getFDireita();
        while (true){
            if (noFolha.getFEsquerda() != null){
                noFolha = noFolha.getFEsquerda();
            }else if(noFolha.getFDireita() != null){
                noFolha = noFolha.getFDireita();
            }else{
                break;
            }
        }
    //Nó pai é um nó folha
    }else{
        noFolha = noPaiAux;
    }

    verificacaoBalanceamento(noFolha);
}
```

CASO 3 (Nó não tem filhos(nó folha)):

```
//Nó não tem filhos(nó folha)
}else{
```

O programa verifica se o nó pai da árvore AVL é diferente de null, se for null é a raíz da árvore e a mesma é atualizada

```
}else{ //Único caso em que um nó não possuí pai é a raiz
    this.Raiz = null;
}
```

Caso não for a raiz, remove ou o filho a esquerda ou o filho da direita do nó pai do nó que foi removido.

```
if (noPai != null) {
    //Atualiza o filho a esquerda do nó pai
    if (atual.getInformacao() < noPai.getInformacao()) {
        noPai.setFEsquerda(null);
        //Atualiza o filho a direita do nó pai
    } else {
        noPai.setFDireita(null);
    }
```

Em seguida verifica o balanceamento do nó folha do nó pai.

```
//Verifica balanceamento
Node noFolha = null;
//Subárvore da esquerda do nó pai
if (noPai.getFEsquerda() != null){
    noFolha = noPai.getFEsquerda();
    while (true){
        if (noFolha.getFDireita() != null){
            noFolha = noFolha.getFDireita();
        }else if(noFolha.getFEsquerda() != null){
            noFolha = noFolha.getFEsquerda();
        }else{
            break;
        }
    }
//Subárvore da direita do nó pai
}else if (noPai.getFDireita() != null){
    noFolha = noPai.getFDireita();
    while (true){
        if (noFolha.getFDireita() != null){
            noFolha = noFolha.getFDireita();
        }else if(noFolha.getFDireita() != null){
            noFolha = noFolha.getFDireita();
        }else{
            break;
        }
    }
//Nó pai é um nó folha
}else{
    noFolha = noPai;
}

verificacaoBalanceamento(noFolha);
```

Método verificacaoBalanceamento:

Método recursivo que chama as rotações para serem executadas nos nós de acordo com o desbalanceamento dos nós pai e vôs do nó que está sendo analisado.

```
//Verifica desbalanceamento dos nós
15 usages
public void verificacaoBalanceamento(Node no){
```

Recebe um nó como parâmetro de entrada que irá ser utilizado na função para verificar o desbalanceamento.

A função começa com uma verificação, verificando se o nó recebido é nulo, se for nulo a iteração é interrompida, senão ela continua.

```
if (no == null){
    return;
}
```

Em seguida a lógica verifica se o nó recebido possuí um nó pai, senão possuir a função é abortada.

```
//Verifica o desbalanceamento dos nós
if (no.getNoPai() != null){
```

Se possuir pai, então o nó pai é armazenado em uma variável e então a lógica verifica se o nó pai possuí pai (nó vô), senão possuir a lógica é abortada, se possuir continua.

```
Node noPai = no.getNoPai();

//Busca o nó vô e o armazena, se possuir
Node noVo = null;
if (noPai.getNoPai() != null){
```

Em seguida o nó vô é definido, e baseado na altura dos nós pai e vô, as operações de rotações são chamadas.

```
noVo = noPai.getNoPai();

//Calcula a altura dos nós
int alturaVo = 0;
int alturaPai = 0;

//Altura do Vo
if (noVo != null){
    alturaVo = altura(noVo.getFEsquerda()) - altura(noVo.getFDireita());
}

//Altura do nó pai
alturaPai = altura(noPai.getFEsquerda()) - altura(noPai.getFDireita());

//Caso de dupla rotação a direita no nó atual
if (alturaVo == 2 && alturaPai == -1) {
    duplaRotacaoDireita(noVo);
//Rotação a direita no nó Pai
} else if (alturaVo == 2) {
    rotacaoDireita(noPai);
}

//Caso de dupla rotação a esquerda no nó vô
if (alturaVo == -2 && alturaPai == 1) {
    duplaRotacaoEsquerda(noVo);
//Rotação a esquerda no nó pai
} else if(alturaVo == -2) {
    rotacaoEsquerda(noPai);
}

//Percorre a árvore verificando o balanceamento até a raiz
verificacaoBalanceamento(noPai);
```

No final de cada verificação, a função é chamada novamente até chegar ao nó raiz.

Método rotacaoDireita:

```
//Método da rotação direita
1 usage
public void rotacaoDireita(Node no){
```

Recebe o nó pai da função verificacaoBalanceamento.

Em seguida define os nós filhos, pai, vô e pai vô.

```
//Definição dos nós filhos, pai, vô e pai Vô
Node noPai = no;
Node noVo = noPai.getNoPai();
Node noPaiVo = noVo.getNoPai();
Node noFilho = noPai.getFEsquerda();
Node noFilhoD = null;
```

Depois verifica se a nova raiz possuí filho a direita para que seja realizada realocações se necessário.

```
//Verifica se a nova raiz possui filhos a direita
if (noPai.getFDireita() != null){
    noFilhoD = noPai.getFDireita();
}
```

Em seguida atualiza o filho a direita da nova raiz.

```
//Atualiza o filho a direita da nova raiz
noPai.setFDireita(noVo);
```

Caso a nova raiz tivesse um filho a direita este filho é reposicionado agora no código.

```
//Reposiciona o nó temporário
if (noFilhoD != null){
    noVo.setFEsquerda(noFilhoD);
    noFilhoD.setNoPai(noVo);
}else{
    noVo.setFEsquerda(null);
}
```

Atualiza o nó pai do filho da direita da nova raiz.

```
//Atualiza o nó pai do filho a direita da nova raiz
noVo.setNoPai(noPai);
```

Reposiciona a nova raiz no lugar do nó vô e atualiza o pai da nova raíz.

```
//Se o pai do nó vô não for nulo não é a raiz
if (noPaiVo != null){
    //Atualiza o filho a direita do nó pai do vô
    if (noPai.getInformacao() >= noPaiVo.getInformacao()){
        noPaiVo.setFDireita(noPai);
    //Atualiza o filho a esquerda do nó pai do vô
    }else{
        noPaiVo.setFEsquerda(noPai);
    }
    noPai.setNoPai(noPaiVo);
}else {
    this.Raiz = noPai;
    noPai.setNoPai(null);
}
```

Método duplaRotacaoDireita:

```
//Método da dupla rotação a direita
1 usage
public void duplaRotacaoDireita(Node noVo){
```

Recebe o nó vô da função verificacaoBalanceamento.

Em seguida define as variáveis de nó pai do vô, nó pai e nó filho.

```
//Definição de variáveis
Node noPaiVo = noVo.getNoPai();
Node noPai = noVo.getFEsquerda();
Node noFilho = noPai.getFDireita();
```

Depois salva os filhos do nó filho em variáveis auxiliares, se o filho possuir filhos.

```
//Manipulação de dados que podem ser filhos do filho
Node noFilhoE = null;
Node noFilhoD = null;

if (noFilho.getFEsquerda() != null){
    noFilhoE = noFilho.getFEsquerda();
}

if (noFilho.getFDireita() != null) {
    noFilhoD = noFilho.getFDireita();
}
```

Em seguida atualiza a nova raiz com os novos filhos.

```
//Atualização da nova raiz
noFilho.setFDireita(noVo);
noFilho.setFEsquerda(noPai);
```

Atualiza os filhos dos filhos da nova raiz.

```
//Vô "perdeu um filho"
if (noFilhoD != null) {
    noVo.setFEsquerda(noFilhoD);
    noFilhoD.setNoPai(noVo);
}else {
    noVo.setFEsquerda(null);
}

//Pai "perdeu seu filho"
if (noFilhoE != null){
    noPai.setFDireita(noFilhoE);
    noFilhoE.setNoPai(noPai);
}else{
    noPai.setFDireita(null);
}
```

Depois atualiza a nova raiz como pai do nó pai e do nó vô.

```
//Atualiza a nova raiz como pai do nó pai e nó vô
noVo.setNoPai(noFilho);
noPai.setNoPai(noFilho);
```

Reposiciona a nova raiz no lugar do nó vô e atualiza o pai da nova raíz.

```
//Se o pai do nó Vô for nulo, logo o nó vô é a raiz
if (noPaiVo != null){
    //Atualiza o filho a esquerda do nó pai do vô
    if (noFilho.getInformacao() >= noPaiVo.getInformacao()){
        noPaiVo.setFDireita(noFilho);
    //Atualiza o filho a direita do nó pai do vô
    }else{
        noPaiVo.setFEsquerda(noFilho);
    }
    noFilho.setNoPai(noPaiVo);
}else{
    this.Raiz = noFilho;
    noFilho.setNoPai(null);
}
```

Método rotacaoEsquerda:

```
//Método da rotação esquerda
1 usage
public void rotacaoEsquerda(Node no){
```

Recebe o nó pai da função verificacaoBalanceamento.

Em seguida define os nós filhos, pai, vô e pai vô.

```
//Definição dos nós filhos, pai, vô e pai do vô
Node noPai = no;
Node noVo = noPai.getNoPai();
Node noPaiVo = noVo.getNoPai();
Node noFilho = noPai.getFDireita();
Node noFilhoE = null;
```

Depois verifica se a nova raiz possuí filho a esquerda para que seja realizada realocações se necessário.

```
//Verifica se a nova raiz possuí filhos a esquerda
if (noPai.getFEsquerda() != null){
    noFilhoE = noPai.getFEsquerda();
}
```

Em seguida atualiza o filho a esquerda da nova raiz.

```
//Atualiza o filho a esquerda da nova raiz
noPai.setFEsquerda(noVo);
```

Caso a nova raiz tivesse um filho a esquerda este filho é reposicionado agora no código.

```
//Reposiciona o nó temporário
if (noFilhoE != null){
    noVo.setFDireita(noFilhoE);
    noFilhoE.setNoPai(noVo);
}else{
    noVo.setFDireita(null);
}
```

Atualiza o nó pai do filho da esquerda da nova raiz.

```
//Atualiza o nó pai do filho a esquerda da nova raiz
noVo.setNoPai(noPai);
```

Reposiciona a nova raiz no lugar do nó vô e atualiza o pai da nova raíz.

```
//Se o vô não for nulo não é a raiz
if (noPaiVo != null){
    //Atualiza o filho a direita do nó pai do vô
    if (noPai.getInformacao() >= noPaiVo.getInformacao()){
        noPaiVo.setFDireita(noPai);
    //Atualiza o filho a esquerda do nó pai do vô
    }else{
        noPaiVo.setFEsquerda(noPai);
    }
    noPai.setNoPai(noPaiVo);
}else{
    this.Raiz = noPai;
    noPai.setNoPai(null);
}
```

Método duplaRotacaoEsquerda:

```
//Método da dupla rotação a esquerda
1 usage
public void duplaRotacaoEsquerda(Node noVo){
```

Recebe o nó vô da função verificacaoBalanceamento.

Em seguida define as variáveis de nó pai do vô, nó pai e nó filho.

```
//Definição de variáveis
Node noPaiVo = noVo.getNoPai();
Node noPai = noVo.getFDireita();
Node noFilho = noPai.getFEsquerda();
```

Depois salva os filhos do nó filho em variáveis auxiliares, se o filho possuir filhos.

```
//Manipulação de dados que podem ser filhos do filho
Node noFilhoE = null;
Node noFilhoD = null;

if (noFilho.getFEsquerda() != null){
    noFilhoE = noFilho.getFEsquerda();
}

if (noFilho.getFDireita() != null) {
    noFilhoD = noFilho.getFDireita();
}
```

Em seguida atualiza a nova raiz com os novos filhos.

```
//Atualização da nova raiz
noFilho.setFEsquerda(noVo);
noFilho.setFDireita(noPai);
```

Atualiza os filhos dos filhos da nova raiz.

```
//Vô "perdeu um filho"
if (noFilhoE != null) {
    noVo.setFDireita(noFilhoE);
    noFilhoE.setNoPai(noVo);
}else{
    noVo.setFDireita(null);
}

//Pai "perdeu seu filho"
if (noFilhoD != null){
    noPai.setFEsquerda(noFilhoD);
    noFilhoD.setNoPai(noPai);
}else{
    noPai.setFEsquerda(null);
}
```

Depois atualiza a nova raiz como pai do nó pai e do nó vô.

```
//Atualiza a nova raiz como pai do nó pai e do nó vô
noVo.setNoPai(noFilho);
noPai.setNoPai(noFilho);
```

Reposiciona a nova raiz no lugar do nó vô e atualiza o pai da nova raíz.

```
//Se o pai do nó vô for nulo, logo o nó vô é a raiz
if (noPaiVo != null){
    //Atualiza o filho a direita do nó pai do vô
    if (noFilho.getInformacao() >= noPaiVo.getInformacao()){
        noPaiVo.setFDireita(noFilho);
        //Atualiza o filho a esquerda do nó pai do vô
    }else{
        noPaiVo.setFEsquerda(noFilho);
    }
    noFilho.setNoPai(noPaiVo);
}else{
    this.Raiz = noFilho;
    noFilho.setNoPai(null);
}
```

Método preOrdem:

```java
//Impressão em Pré-Ordem
3 usages
public void preOrdem(Node n){
    if (n != null){
        //Imprime a informação da raíz
        System.out.print(n.getInformacao() + " ");
        //Se tiver filho a esquerda chama o mesmo
        if (n.getFEsquerda() != null){
            preOrdem(n.getFEsquerda());
        }
        //Se tiver filho a direita chama o mesmo
        if (n.getFDireita() != null){
            preOrdem(n.getFDireita());
        }
    }
}
```

Imprime a árvore AVL no formato pré-ordem.

Método emOrdem:

```java
//Impressão em emOrdem
3 usages
public void emOrdem(Node n){
    if (n != null){
        //Se tiver filho a esquerda chama o mesmo
        if (n.getFEsquerda() != null){
            emOrdem(n.getFEsquerda());
        }
        //Imprime a informação da raíz
        System.out.print(n.getInformacao() + " ");
        //Se tiver filho a direita chama o mesmo
        if (n.getFDireita() != null){
            emOrdem(n.getFDireita());
        }
    }
}
```

Imprime a árvore AVL no formato em ordem.

Método posOrdem:

```java
//Impressão em Pós-Ordem
3 usages
public void posOrdem(Node n){
    if (n != null){
        //Se tiver filho a esquerda chama o mesmo
        if (n.getFEsquerda() != null){
            posOrdem(n.getFEsquerda());
        }
        //Se tiver filho a direita chama o mesmo
        if (n.getFDireita() != null){
            posOrdem(n.getFDireita());
        }
        //Imprime a informação da raiz
        System.out.print(n.getInformacao() + " ");
    }
}
```

Imprime a árvore AVL no formato pós-ordem.

CLASSE Main:

Na classe Main, são realizadas as chamadas de métodos da classe da árvore AVL e é o lugar onde o menu da aplicação está implementado.

Bibliotecas:

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;
```

As bibliotecas que foram utilizadas serviram de auxílio para receber inputs na classe main, ler arquivos txt e gerar números aleatórios.

Variáveis globais:

```
//Criação da Árvore AVL
ArvoreAVL arvoreAVL = new ArvoreAVL();

//Criação do gerador de números aleatórios
Random gerador = new Random();

//Criação dos scanners
Scanner sc = new Scanner(System.in);
Scanner scF;
```

Menu:

```
//Menu
int option = -1;
while (option != 0){
    System.out.println("===== Árvore AVL =====");

    //Impressão da árvore AVL nos três modos
    System.out.println();
    System.out.println("Pré-Ordem: ");
    arvoreAVL.preOrdem(arvoreAVL.getRaiz());
    System.out.println();

    System.out.println();
    System.out.println("Em-Ordem: ");
    arvoreAVL.emOrdem(arvoreAVL.getRaiz());
    System.out.println();

    System.out.println();
    System.out.println("Pós-Ordem: ");
    arvoreAVL.posOrdem(arvoreAVL.getRaiz());
    System.out.println();
    System.out.println("========================");

    System.out.println("0. Sair");
    System.out.println("1. Inserção");
    System.out.println("2. Remoção");
    System.out.println("3. Busca");
    System.out.println("4. Geração de árvore AVL com 100 números aleatórios");
    System.out.println("5. Geração de árvore AVL com 500 números aleatórios");
    System.out.println("6. Geração de árvore AVL com 1000 números aleatórios");
    System.out.println("7. Geração de árvore AVL com 10000 números aleatórios");
    System.out.println("8. Geração de árvore AVL com 20000 números aleatórios");
    System.out.println("9. Carregar txt");
    System.out.println("Escolha uma opção: ");
    option = sc.nextInt();
```

Criação da base do menu, onde estão listadas as operações e a leitura da opção escolhida do usuário.

Definição de variáveis auxiliares do menu:

```
//Variáveis de auxílio para execução
int numero = 0;
long comeco = 0;
long fim = 0;
long tempoTotal = 0;
```

Navegação do menu:

```
//Navegação do menu
switch (option){
```

A navegação no menu ocorre baseado num switch case de opções.

Menu (Parar execução do programa):

```
case 0:
    //Encerra o programa
    System.exit( status: 0);
    break;
```

Caso a opção selecionada for 0, o programa tem sua execução interrompida.

Menu (Inserção):

```
case 1:
    //Inserção
    boolean parou = false;
    while (parou == false) {
        System.out.println();
        System.out.println("Entre com o número (-1 para parar): ");
        numero = sc.nextInt();
        comeco = System.nanoTime();
        if (numero != -1){
            Node novoNo = new Node(numero);
            arvoreAVL.insertElementoArvore(novoNo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(novoNo);
        }else {
            parou = true;
        }
        fim = System.nanoTime();
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos
        System.out.println("Tempo total de execução da inserção: " + tempoTotal + " microssegundos");
    }
    break;
```

Caso a opção selecionada for 1, então o programa inicia um loop que pede para o usuário entrar com os números a serem inseridos na árvore binária.

A cada elemento inserido pelo usuário, a classe main cria um nó com o número inserido e chama o método de inserção da classe da árvore AVL com os parâmetros do novo nó a ser inserido na árvore AVL e a raiz da árvore AVL, em seguida é verificado o balanceamento do lado da subárvore em que o novo nó foi inserido.

O tempo de cada inserção é medido e imprimido no final da inserção do elemento no console.

Se o usuário inserir o valor -1 o programa volta para o menu de opções.

Menu (Remoção):

```
case 2:
    //Remoção
    System.out.println();
    System.out.println("Entre com o número: ");
    numero = sc.nextInt();
    comeco = System.nanoTime();
    boolean noEncontradoR = arvoreAVL.buscar(numero);
    if (noEncontradoR == true){
        arvoreAVL.removeElemento(numero, arvoreAVL.getRaiz(), noP: null);
    }else {
        System.out.println("Nó " + numero + " não foi encontrado para remoção!");
    }
    fim = System.nanoTime();
    tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos
    System.out.println("Tempo total de execução da remoção: " + tempoTotal + " microssegundos");
    break;
```

Caso a opção selecionada for 2, então o programa solicita o valor do nó que deve ser removido.

Assim que o usuário inserir o valor, o programa busca por este nó, se o programa encontrar o nó, então chama o método de remoção da classe ArvoreAVL com os parâmetros de número inserido pelo usuário, com a raiz da árvore AVL e com o valor null (nó pai da raiz). Se o valor do número inserido pelo usuário não for encontrado na árvore AVL, avisa isto ao usuário.

O tempo de cada remoção ou tentativa de remoção é medido e imprimido no final de cada execução desta opção no console.

Menu (Busca):

```
case 3:
    //Busca
    int numeroBusca = -1;
    System.out.println("Insira um número para ser realizado a busca: ");
    numeroBusca = sc.nextInt();
    comeco = System.nanoTime();
    boolean noEncontradoB = arvoreAVL.buscar(numeroBusca);
    if (noEncontradoB == true){
        System.out.println("Nó " + numeroBusca + " foi encontrado!");
    }else{
        System.out.println("Nó " + numeroBusca + " não foi encontrado!");
    }
    fim = System.nanoTime();
    tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos
    System.out.println("Tempo total de execução da busca: " + tempoTotal + " microssegundos");
    break;
```

Caso a opção selecionada for 3, então o programa solicita o valor do nó a ser buscado.

Assim que o valor do nó a ser buscado for informado, é criada a variável noEncontradoB, que possuí o seu valor baseado no retorno do método buscar do valor informado pelo usuário.

Caso o noEncontradoB for true, o número foi encontrado na árvore AVL. Caso contrário, o número não foi encontrado na mesma.

O tempo de cada busca é medido e imprimido no final de cada execução desta opção no console.

Menu (Criação de árvore AVL com 100 elementos aleatórios):

```
case 4:
    //Inserção de 100 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 100 números
        FileWriter fw = new FileWriter( fileName: "CemElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore AVL
        for (int i = 0; i < 100; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            Node novoNo = new Node(numero);
            arvoreAVL.insertElementoArvore(novoNo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(novoNo);
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 100 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore AVL com 100 elementos!");
    }
    break;
```

Esta opção começa criando um txt que irá armazenar os 100 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 100 elementos aleatórios na árvore AVL.

O loop que gera os elementos aleatórios é basicamente um for que é executado 100 vezes gerando um número aleatório de 0-99, onde cada elemento

gerado é salvo em uma linha do documento txt criado (CemElementos.txt) e também é inserido e tem seu balanceamento verificado na árvore AVL.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore AVL com 500 elementos aleatórios):

```
case 5:
    //Inserção de 500 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 500 números
        FileWriter fw = new FileWriter( fileName: "QuinhentosElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore AVL
        for (int i = 0; i < 500; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            Node novoNo = new Node(numero);
            arvoreAVL.insertElementoArvore(novoNo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(novoNo);
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 500 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore AVL com 500 elementos!");
    }
    break;
```

Esta opção começa criando um txt que irá armazenar os 500 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 500 elementos aleatórios na árvore AVL.

O loop que gera os elementos aleatórios é basicamente um for que é executado 500 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (QuinhentosElementos.txt) e também é inserido e tem seu balanceamento verificado na árvore AVL.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore AVL com 1000 elementos aleatórios):

```
case 6:
    //Inserção de 1000 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 1000 números
        FileWriter fw = new FileWriter( fileName: "MilElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore AVL
        for (int i = 0; i < 1000; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            Node novoNo = new Node(numero);
            arvoreAVL.insertElementoArvore(novoNo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(novoNo);
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 1000 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore AVL com 1000 elementos!");
    }
    break;
```

Esta opção começa criando um txt que irá armazenar os 1000 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 1000 elementos aleatórios na árvore AVL.

O loop que gera os elementos aleatórios é basicamente um for que é executado 1000 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (MilElementos.txt) e também é inserido e tem seu balanceamento verificado na árvore AVL.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore AVL com 10000 elementos aleatórios):

```
case 7:
    //Inserção de 10000 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 10000 números
        FileWriter fw = new FileWriter( fileName: "DezMilElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore AVL
        for (int i = 0; i < 10000; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            Node novoNo = new Node(numero);
            arvoreAVL.insertElementoArvore(novoNo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(novoNo);
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 10000 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore AVL com 10000 elementos!");
    }
    break;
```

Esta opção começa criando um txt que irá armazenar os 10000 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 10000 elementos aleatórios na árvore AVL.

O loop que gera os elementos aleatórios é basicamente um for que é executado 10000 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (DezMilElementos.txt) e também é inserido e tem seu balanceamento verificado na árvore AVL.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Criação de árvore AVL com 20000 elementos aleatórios):

```java
case 8:
    //Inserção de 20000 números aleatórios
    System.out.println();

    //Começa a contagem do tempo
    comeco = System.nanoTime();

    try {
        //Criação do arquivo TXT de 20000 números
        FileWriter fw = new FileWriter( fileName: "VinteMilElementos.txt");
        PrintWriter pw = new PrintWriter(fw);

        //Inserção dos elementos na arvore AVL
        for (int i = 0; i < 20000; i++) {
            numero = gerador.nextInt( bound: 100);
            pw.println(numero);
            Node novoNo = new Node(numero);
            arvoreAVL.insertElementoArvore(novoNo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(novoNo);
        }

        System.out.println();

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da inserção de 20000 números aleatórios: " + tempoTotal + " microssegundos");

        //Salva e fecha o txt
        pw.flush();
        pw.close();
        fw.close();

    }catch (Exception erro){
        System.out.println("Erro ao gerar árvore AVL com 20000 elementos!");
    }
    break;
```

Esta opção começa criando um txt que irá armazenar os 20000 elementos que foram gerados de forma aleatória na execução do loop de inserção dos 20000 elementos aleatórios na árvore AVL.

O loop que gera os elementos aleatórios é basicamente um for que é executado 20000 vezes gerando um número aleatório de 0-99, onde cada elemento gerado é salvo em uma linha do documento txt criado (VinteMilElementos.txt) e também é inserido e tem seu balanceamento verificado na árvore AVL.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

Antes de sair da opção, salva o documento TXT.

Menu (Leitura de TXT):

```
case 9:
    //Leitura de arquivo TXT com os dados
    Path caminho100 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreAVL\\CemElementos.txt");
    Path caminho500 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreAVL\\QuinhentosElementos.txt");
    Path caminho1000 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreAVL\\MilElementos.txt");
    Path caminho10000 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreAVL\\DezMilElementos.txt");
    Path caminho20000 = Paths.get( first: "C:\\Users\\Gabriel F\\Documents\\Faculdade\\4° Período\\Estrutura de Dados\\Árvores\\ArvoreAVL\\VinteMilElementos.txt");

    //Opção de txt a ser lido:
    int opcaoTxt = -1;

    System.out.println();
    System.out.println("1 - 100 Elementos");
    System.out.println("2 - 500 Elementos");
    System.out.println("3 - 1000 Elementos");
    System.out.println("4 - 10000 Elementos");
    System.out.println("5 - 20000 Elementos");
    System.out.println("Escolha um txt a ser lido:");

    //Vetor para armazenar os valores
    ArrayList<Integer> numerosTxt = new ArrayList<>();

    //Leitura de opção TXT
    opcaoTxt = sc.nextInt();
```

Caso a opção selecionada for 9, o programa salva o caminho de cada documento TXT em variáveis separadas para cada um.

Em seguida é iniciado um novo menu de seleção de leitura de txt, baseado na escolha do usuário o txt escolhido é lido pelo programa.

```
case 1:
    //100 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 100 elementos
        scF = new Scanner(caminho100);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreAVL.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            Node noNovo = new Node(i);
            arvoreAVL.insertElementoArvore(noNovo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(noNovo);
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 100 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 100 números!");
    }
    break;
```

A lógica de leitura é basicamente a leitura até a última linha do arquivo do TXT selecionado pelo usuário dentro de um loop while que salva os valores lidos em um Arraylist de inteiros.

Em seguida limpa a árvore AVL para criar uma nova baseado nos 100 elementos novos.

Assim que limpar que a árvore AVL for “zerada”, a mesma recebe a inserção dos novos valores que estão no Arraylist de leitura do arquivo do txt.

A cada inserção, também é verificado o balanceamento da subárvore do lado que sofreu alteração na árvore avl.

O tempo deste processo é medido e imprimido no final de cada execução desta opção no console.

A lógica é a mesma para as outras opções de leitura ->

```
case 2:
    //500 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 500 elementos
        scF = new Scanner(caminho500);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreAVL.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            Node noNovo = new Node(i);
            arvoreAVL.insertElementoArvore(noNovo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(noNovo);
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 500 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 500 números!");
    }
    break;
```

```java
case 3:
    //1000 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 1000 elementos
        scF = new Scanner(caminho1000);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreAVL.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            Node noNovo = new Node(i);
            arvoreAVL.insertElementoArvore(noNovo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(noNovo);
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 1000 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 1000 números!");
    }
    break;
```

```java
case 4:
    //10000 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 10000 elementos
        scF = new Scanner(caminho10000);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreAVL.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            Node noNovo = new Node(i);
            arvoreAVL.insertElementoArvore(noNovo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(noNovo);
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 10000 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 10000 números!");
    }
    break;
```

```java
case 5:
    //20000 Elementos
    try{
        //Começa a contagem do tempo
        comeco = System.nanoTime();

        //Percorre arquivo TXT de 20000 elementos
        scF = new Scanner(caminho20000);
        while(scF.hasNextLine()){
            numerosTxt.add(Integer.parseInt(scF.nextLine()));
        }

        //Apaga a árvore binária
        arvoreAVL.setRaiz(null);

        //Preenche com os elementos do txt
        for (Integer i : numerosTxt){
            Node noNovo = new Node(i);
            arvoreAVL.insertElementoArvore(noNovo, arvoreAVL.getRaiz());
            arvoreAVL.verificacaoBalanceamento(noNovo);
        }

        //Finaliza a contagem do tempo
        fim = System.nanoTime();

        //Calcula o tempo total da execução
        tempoTotal = (fim - comeco) / 1000; // Converter para microssegundos

        //Imprime o tempo total da execução
        System.out.println("Tempo total da leitura do txt de 20000 números: " + tempoTotal + " microssegundos");
    }catch (Exception erro){
        System.out.println("Erro ao ler o arquivo TXT com 20000 números!");
    }
    break;
```

PERFORMANCE DA ÁRVORE BINÁRIA

A performance da árvore binária foi medida no tempo de cada execução de cada ação na árvore binária em microssegundos.

Performance (Inserção 10, 59, 98 após 20000 elementos serem inseridos):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	18	15	16	16

Performance (Inserção 100 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	3.206	2.777	2.218	2.733

Performance (Inserção 500 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	5.565	4.875	6.271	5.570

Performance (Inserção 1000 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	6.319	6.974	7.625	6.792

Performance (Inserção 10000 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	22.542	24.758	23.370	23.556

Performance (Inserção 20000 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	56.365	55.234	50.292	53.963

Performance (Remoção 54 3x após 20000 elementos serem inseridos):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	29	28	28	28

Performance (Busca 55 após 20000 elementos serem inseridos):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	363	197	318	292

PERFORMANCE DA ÁRVORE AVL

A performance da árvore AVL foi medida no tempo de cada execução de cada ação na árvore binária em microsegundos

Performance (Inserção 10, 59, 98 após 20000 elementos serem inseridos):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	602	548	557	569

Performance (Inserção 100 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	2.547	3.279	3.138	2.988

Performance (Inserção 500 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	7.492	7.349	7.144	7.328

Performance (Inserção 1000 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	15.050	13.943	11.683	13.558

Performance (Inserção 10000 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	923.028	894.561	923.788	913.792

Performance (Inserção 20000 elementos aleatórios):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	3.899.776	3.706.116	3.854.795	3.820.229

Performance (Remoção 54 3x após 20000 elementos serem inseridos):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	911	863	380	718

Performance (Busca 10, 59, 98 após 20000 elementos serem inseridos):

-	TENTATIVA 1	TENTATIVA 2	TENTATIVA 3	MÉDIA
TEMPO (µs)	304	46	21	123

CONCLUSÃO

De acordo com minha implementação as operações de inserção e de exclusão de um elemento na árvore binária é mais rápida que na árvore AVL, pois não há a operação de balanceamento e de verificação de balanceamento de nós.

Contudo a operação de busca é cerca de 2,43x mais devagar na árvore binária do que na árvore AVL, devido a altura da árvore AVL ser inferior a árvore binária.

Assim, podemos ver que a estrutura de árvore AVL é mais eficiente quando se trata de busca em um conjunto de dados e é uma estrutura de dados mais complexa e inteligente que uma simples árvore binária.