

Trabalho de Estrutura de Dados _- _ Arvore AVL e Arvore Binaria

Aluno: Tiago Vieira Paulin – Eng. de Software Turma B

ARVORE BINARIA X ARVORE AVL

Em resumo, tanto a Arvore AVL quanto a Arvore Binária funcionam da mesma forma. Ambas consistem em uma estrutura de dados onde é possível manipular dados de maneira onde os elementos de maior valor se encontram mais a direita da árvore, e da mesma forma, os de menor valor se encontram mais a esquerda da árvore. A diferença entre as duas está no fato de que a binária não possui restrição quanto ao tamanho de suas sub-árvores a esquerda e a direita do nó e questão, já a Árvore AVL segue a seguinte regra: A diferença entre a sub-árvore a direita e a sub-árvore a esquerda de um determinado nó não pode ser maior do que 1 e nem menor do que -1. Caso isso venha acontecer, a árvore AVL deve realizar um balanceamento em seus nós de forma a reposicioná-los a fim de que eles atendam essa regra. Por meio dessa organização a árvore AVL possibilita uma estrutura mais simétrica e equilibrada, evitando que a árvore cresça mais para um lado que para o outro.

Qual funciona melhor? Em qual situação? Porque?

Vale ressaltar que cada árvore possui seu propósito, e o que define qual funciona melhor depende do caso e do projeto

- **Árvore Binária:** Uma das principais vantagens dela é sem dúvida a sua implementação simples e que não necessita muita manutenção, sua utilização é recomendada para casos onde não é necessária a inserção e remoção de elementos com muita frequência, e quando não é necessário comportar um volume muito grande de dados. Possível é bem possível que dessa forma ela “penda” mais para um lado que para o outro.
- **Árvore AVL:** Em contrapartida a árvore AVL possui uma implementação mais complexa, porém, ela é mais recomendada quando se deseja manipular, inserir e remover uma grande quantidade de dados. Pois por ser equilibrada pode possibilitar uma manipulação de dados mais otimizada em alguns casos. Exceto onde é necessária a inserção de uma grande quantidade de dados em pouco tempo.

IMPLEMENTAÇÃO DO CÓDIGO – CLASSE NodeTree

Atributos:

```
89 usages
public class NodeTree {
    // definindo atributos
    3 usages
    private Integer informacao;
    3 usages
    private NodeTree esquerdo;
    3 usages
    private NodeTree direito;
    // definindo métodos
```

Os atributos da classe são informação, esquerda e direita, que, respectivamente comportam a informação do nó instanciado, o nó a esquerda e o nó a direita

Construtor:

```
2 usages
7 public NodeTree() {
8     this.informacao = null;
9     this.esquerdo = null;
10    this.direito = null;
11 }
```

Ao instanciar um nó, inicialmente todos os seus atributos são definidos como null.

Métodos get:

```
12 // metodos get
53 usages
13 public Integer getInformacao() { return informacao; }
81 usages
16 public NodeTree getEsquerdo() { return esquerdo; }
67 usages
19 public NodeTree getDireito() { return direito; }
```

Retornam os valores dos atributos do nó para que eles possam ser manipulados na árvore

Métodos set:

```
// metodos set
2 usages
public void setInformacao(Integer informacao) { this.informacao = informacao; }
33 usages
public void setEsquerdo(NodeTree esquerdo) { this.esquerdo = esquerdo; }
25 usages
public void setDireito(NodeTree direito) { this.direito = direito; }
}
```

Possibilitam alterar os valores dos atributos dos nós para serem manipulados na árvore

IMPLEMENTAÇÃO DO CÓDIGO – CLASSE ArvoreBinaria

Atributos:

```
public class ArvoreBinaria {
    // definindo atributos
    26 usages
    private NodeTree raiz;
}
```

O único atributo que a classe Arvore Binária possui é do tipo NodeTree é a raiz da árvore, que vai armazenar o nó principal que dará início a árvore e será pai de todos os outros nós.

Método construtor:

```
// metodo construtor
no usages
public ArvoreBinaria(){
    this.raiz = null;
}
```

O método construtor inicializa o único atributo da classe como null, pois, quando instanciada a árvore ainda não possui nenhum nó.

Método de inserção:

```
// metodo para inserir dados na arvore
no usages
public void inserir(int dado){
    NodeTree no = new NodeTree(); // instancia o novo no
    no.setInformacao(dado); // seta o dado
    if(raiz == null){ // caso a raiz esteja vazia
        raiz = no;
    } else { // se nao
        NodeTree atual = raiz; // cria auxiliar
        while(true){
            // condicoes de insercao
            if(atual.getEsquerdo() == null){ // se o ultimo da esquerda for uma folha
                if(dado < atual.getInformacao()){ // caso o dado seja menor que o da folha
                    atual.setEsquerdo(no); // insere a esquerda da folha
                    break; // para o loop
                }
            }
            if(atual.getDireito() == null){ // se o ultimo da direita for folha
                if(dado >= atual.getInformacao()){ // caso o dado seja maior ou igual o da folha
                    atual.setDireito(no); // insere a direita da folha
                    break; // para o loop
                }
            }
            // condicoes de navegabilidade
            if(dado >= atual.getInformacao()){ // se o dado for maior que o do no
                atual = atual.getDireito(); // caminha pra direita
            } else { // se for menor
                atual = atual.getEsquerdo(); // caminha pra esquerda
            }
        }
    }
}
```

O método de inserção recebe o dado que se deseja inserir na árvore e o armazena em um novo nó instanciado no começo do método, após isso é feita uma verificação que verifica se a árvore está vazia, se sim, aquele nó assume o lugar da raiz principal da árvore, se não, vai entrar em um loop que temo intuito de encontrar a posição ideal para aquele nó, de acordo com o valor recebido, se for maior que o no que está comparando, se sim o dado é comparado com o nó a direita, se menor, ele é comparado com o valor a esquerda, quando é encontrada a posição para o novo nó ele é inserido e o loop para.

Métodos de travessia da Árvore:

```
52     private void preOrdem(NodeTree arvore){
53         if (arvore != null){ // se o no nao for nulo
54             System.out.print(arvore.getInformacao() + " "); // printa a informacao
55             preOrdem(arvore.getEsquerdo()); // chamada recursiva com o proximo da esquerda
56             preOrdem(arvore.getDireito()); // chamada recursiva com o proximo da direita
57         }
58     }
59     // metodo para imprimir in-ordem
60     3 usages
61     private void inOrdem(NodeTree arvore){
62         if(arvore != null){ // se o no nao for nulo
63             inOrdem(arvore.getEsquerdo()); // chamada recursiva com o proximo da esquerda
64             System.out.print(arvore.getInformacao() + " "); // printa a informacao
65             inOrdem(arvore.getDireito()); // chamada recursiva com o proximo da direita
66         }
67     }
68     // metodo para imprimir pos-ordem
69     3 usages
70     private void posOrdem(NodeTree arvore){
71         if(arvore != null){ // se o no nao for nulo
72             posOrdem(arvore.getEsquerdo()); // chamada recursiva com o proximo da esquerda
73             posOrdem(arvore.getDireito()); // chamada recursiva com o proximo da direita
74             System.out.print(arvore.getInformacao() + " "); // printa a informacao
75         }
76     }
77 }

40     public void imprimir(){
41         System.out.print("Pré-ordem: "); // tipo de organizacao
42         preOrdem(raiz); // metodo que imprime
43         System.out.println("\n"); // pula uma linha
44         System.out.print("In-ordem: "); // tipo de organizacao
45         inOrdem(raiz); // metodo que imprime
46         System.out.println("\n"); // pula uma linha
47         System.out.print("Pós-ordem: "); // tipo de organizacao
48         posOrdem(raiz); // metodo que imprime
49         System.out.println("\n"); // pula uma linha
50     }
```

Os métodos de travessia são o preOrdem, inOrdem e posOrdem que realizam chamadas recursivas a fim de percorrer e imprimir os dados da árvore no padrão desejado.

Pré-ordem: raiz , subárvore a esquerda e depois subárvore a direita

In-ordem: subárvore esquerda, raiz, subárvore direita

Pós-ordem: subárvore esquerda, subárvore direita, raiz

O método imprimir basicamente reúne esses métodos para imprimir as três travessias com uma única chamada e de forma organizada, facilitando a execução do print e entendimento da árvore.

Método de Remoção da Raiz:

```
109 // metodo para remover a raiz da arvore
110 3 usages
111 private void removerRaiz(){
112     if(isFolha(raiz)){ // se a raiz for folha
113         raiz = null; // iguala a null
114     } else { // se nao
115         if(temDireito(raiz) && temEsquerdo(raiz)){ // se o no tem direito e tem esquerdo
116             NodeTree novaRaiz = raiz.getDireito(); // nova raiz sera o no da direita
117             if(novaRaiz.getEsquerdo() == null){ // se a nova raiz nao tiver nenhum no a esquerda
118                 novaRaiz.setEsquerdo(raiz.getEsquerdo()); // posiciona a nova raiz da arvore setando no esquerdo o lado esquerdo da arvore
119             } else { // se a nova raiz tiver um no a esquerda
120                 NodeTree esq = novaRaiz.getEsquerdo(); // armazena
121                 while(esq.getEsquerdo() != null){ // enquanto nao estiver no ultimo no a esquerda da nova raiz
122                     esq = esq.getEsquerdo(); // percorre
123                 }
124                 esq.setEsquerdo(raiz.getEsquerdo()); // seta a esquerda desse no toda a subarvore esquerda da antiga raiz
125                 raiz = novaRaiz; // atualizo minha arvore
126             }
127         } else if(temEsquerdo(raiz) && !temDireito(raiz)){ // se a raiz tiver esquerdo e nao tiver direito
128             raiz = raiz.getEsquerdo(); // percorre um no
129         } else { // se tiver direito e nao tiver esquerdo
130             raiz = raiz.getDireito(); // percorre um no
131         }
132     }
133 }
```

Foi implementado um método que realiza a remoção da raiz da árvore para facilitar o processo no método remover caso o dado que o usuário queira remover corresponder a raiz da árvore, basicamente esse método verifica se a raiz é uma folha, ou seja, não possui nenhum filho, nesse caso a raiz vai ser igualada a null. Se não for uma folha os outros casos serão tratados, o caso em que a raiz tenha só filhos a direita, o caso em que a raiz tenha filhos só a esquerda e caso a raiz tenha filhos em ambos os lados. Basicamente em todos esses casos a raiz é removida e os nós restantes são reposicionados na árvore.

Método de Remoção:

```
133 // metodo para remover um elemento qualquer da arvore
134 no usages
135 public void remover(int dado){
136     NodeTree remover = raiz; // inicializa o no que sera removido
137     NodeTree pai = raiz; // inicializa o pai do removido
138     NodeTree newFilho; // inicializa o novo filho
139     if(dado == raiz.getInformacao()){ // se o dado coincidir com a raiz
140         removerRaiz(); // remove a raiz
141     } else { // se nao
142         while(dado != remover.getInformacao()){ // enquanto eu nao encontrar o dado na arvore
143             if(dado > remover.getInformacao()){ // se o dado for maior que o que esta no remover
144                 pai = remover; // atualiza o pai
145                 remover = remover.getDireito(); // caminha pra direita
146             } else { // se nao
147                 pai = remover; // atualiza o pai
148                 remover = remover.getEsquerdo(); // caminha pra esquerda
149             }
150         }
151         if(isFolha(remover)){ // se o remover for folha
152             if(pai.getInformacao() > remover.getInformacao()){ // se o pai for maior que o removido
153                 pai.setEsquerdo(null); // apaga o da esquerda
154             } else { // se nao
155                 pai.setDireito(null); // apaga o da direita
156             }
157         } else { // se nao for folha
158             newFilho = remover.getDireito();
159             pai.setDireito(newFilho);
160         }
161     }
162 }
```

```

156 } else { // se nao for folha
157     if (temEsquerdo(remover) && !temDireito(remover)){ // se o no tiver esquerdo e nao tiver direito
158         newFilho = remover.getEsquerdo(); // seta o novo filho
159         if(newFilho.getInformacao() >= pai.getInformacao()){ // se o novo filho for maior ou igual ao pai
160             pai.setDireito(newFilho); // seta o novo filho a direita do pai
161         } else { // se nao
162             pai.setEsquerdo(newFilho); // seta o novo filho a esquerda do pai
163         }
164     } else if (temDireito(remover) && !temEsquerdo(remover)){ // se tem direito e nao tem esquerdo
165         newFilho = remover.getDireito(); // seta o novo filho
166         if(newFilho.getInformacao() >= pai.getInformacao()){ // se o novo filho for maior ou igual ao pai
167             pai.setDireito(newFilho); // seta o novo filho a direita do pai
168         } else { // se nao
169             pai.setEsquerdo(newFilho); // seta o novo filho a esquerda do pai
170         }
171     } else { // se o removido tiver direito e esquerdo
172         newFilho = remover.getDireito(); // seta o novo filho
173         if(isFolha(newFilho)){ // se o novo filho for folha
174             if(newFilho.getInformacao() >= pai.getInformacao()){ // se o novo filho for maior ou igual ao pai
175                 pai.setDireito(newFilho); // seta o novo filho a direita do pai
176             } else { // se nao
177                 pai.setEsquerdo(newFilho); // seta o novo filho a esquerda do pai
178             }
179             if(remover.getEsquerdo() != null){ // se o removido tiver um esquerdo
180                 newFilho.setEsquerdo(remover.getEsquerdo()); //seta o esquerdo do removido a esquerda do novo filho
181             }
182         } else { // se o novo filho nao for folha
183             NodeTree menorPossivel = newFilho; // variavel para armazenar o menor possivel
184             NodeTree aux = newFilho; // auxiliar
185             while (true){
186                 if(menorPossivel.getEsquerdo() != null){ // enquanto nao for o menor possivel
187                     // se o novo filho for maior ou igual ao pai
188                     NodeTree menorPossivel = newFilho; // variavel para armazenar o menor possivel
189                     NodeTree aux = newFilho; // auxiliar
190                     while (true){
191                         if(menorPossivel.getEsquerdo() != null){ // enquanto nao for o menor possivel
192                             aux = menorPossivel; // armazena na auxiliar
193                             menorPossivel = menorPossivel.getEsquerdo(); // percorre
194                         }
195                         if(menorPossivel.getEsquerdo() == null || isFolha(menorPossivel)){ // se o menor possivel for a ultima a esquerda ou for uma folha
196                             break; // para o loop
197                         }
198                     }
199                     if(menorPossivel.getDireito() != null){ // se tiver algum no a direita do menor possivel
200                         aux.setEsquerdo(menorPossivel.getDireito()); // seta o no a esquerda do auxiliar
201                     }
202                     newFilho = menorPossivel; // novo filho vira o menor possivel
203                     if(newFilho.getInformacao() >= pai.getInformacao()){ // se o novo filho for maior ou igual ao pai
204                         pai.setDireito(newFilho); // insere a direita do pai
205                     } else { // se nao
206                         pai.setEsquerdo(newFilho); // seta a esquerda do pai
207                     }
208                     // atualiza a esquerda e a direita do novo filho
209                     newFilho.setEsquerdo(remover.getEsquerdo());
210                     newFilho.setDireito(remover.getDireito());
211                 }
212             }
213         }
214     }
215 }

```

O método de remoção que foi o mais complexo ele faz basicamente as mesmas verificações do remover raiz só que de forma a valer para qualquer posição da árvore, no geral eu vou ter três variáveis, o removido, que vai corresponder ao nó que eu quero remover, o pai, que corresponde ao pai do que será removido e o newFilho que irá assumir a posição do nó que foi removido, caso seja necessário. Basicamente os nós pai e removido percorrem a árvore até encontrar o valor que foi passado como parâmetro no método remover e quando encontram o nó removido é setado como null ou deixado de lado e o pai recebe um novo filho, de forma a rearranjar a árvore

Método de Busca:

```
235 // metodo para buscar um elemento na arvore
    no usages
236 public int buscar(int dado){
237     int resultado;
238     NodeTree percorre = raiz;
239     while(true){
240         if(percorre.getInformacao() == dado){
241             resultado = dado;
242             break;
243         }
244         if(dado >= percorre.getInformacao()){
245             percorre = percorre.getDireito();
246         } else if (dado < percorre.getInformacao()){
247             percorre = percorre.getEsquerdo();
248         }
249     }
250     return resultado;
251 }
```

O método de busca possui um retorno do tipo int e recebe um parâmetro que corresponde ao valor que será buscado na árvore, o parâmetro é comparado com os nós da árvore até encontrar um valor correspondente ao desejado, quando ele encontra esse valor o valor é atribuído ao atributo resultado e o mesmo é retornado pelo método.

Métodos para Realizar Verificações:

```
211 // método para verificar se o nó tem direito
212 // 4 usages
213 @ private boolean temDireito(NodeTree arvore){
214     if(arvore.getDireito() == null){ // se o nó a direita for null
215         return false; // não tem direito
216     } else { // se não
217         return true; // tem direito
218     }
219 }
220 // método para verificar se o nó tem esquerdo
221 // 4 usages
222 @ private boolean temEsquerdo(NodeTree arvore){
223     if(arvore.getEsquerdo() == null){ // se o nó a esquerda for null
224         return false; // não tem esquerdo
225     } else { // se não
226         return true; // tem esquerdo
227     }
228 }
229 // método para verificar se o nó é folha
230 // 6 usages
231 @ private boolean isFolha(NodeTree no){
232     if(no.getEsquerdo() == null && no.getDireito() == null){ // se o esquerdo e o direito for null
233         return true; // é folha
234     } else { // se não
235         return false; // não é folha
236     }
237 }
```

Para facilitar as operações feitas nos métodos de remoção foram feitos os seguintes métodos de verificação:

temDireito: esse método recebe um nó e verifica se o mesmo possui um filho a sua direita por meio de um retorno boolean

temEsquerdo: esse método recebe um nó e verifica se o mesmo possui um filho a sua esquerda por meio de um retorno boolean

isFolha: esse método recebe um nó e verifica se o mesmo é uma folha (não possui filhos) por meio de um retorno boolean

IMPLEMENTAÇÃO DE CÓDIGO – CLASSE ArvoreAvl

Atributos:

```
3 // 6 usages
4 public class ArvoreAvl {
5     // definindo atributos
6     // 40 usages
7     private NodeTree raiz;
```

O único atributo que a classe Arvore Binária possui é do tipo NodeTree é a raiz da árvore, que vai armazenar o nó principal que dará início a árvore e será pai de todos os outros nós.

Método construtor:

```
6      // metodo construtor
      3 usages
7      public ArvoreAvl(){
8          this.raiz = null;
9      }
```

O método construtor inicializa o único atributo da classe como null, pois, quando instanciada a árvore ainda não possui nenhum nó.

Método de inserção:

```
12      public void inserir(int dado){
13          NodeTree no = new NodeTree(); // instancia o novo no
14          no.setInformacao(dado); // seta o dado
15          if(raiz == null){ // caso a raiz esteja vazia
16              raiz = no;
17          } else { // se nao
18              NodeTree atual = raiz; // cria auxiliar
19              while(true){
20                  // condicoes de insercao
21                  if(atual.getEsquerdo() == null){ // se o ultimo da esquerda for uma folha
22                      if(dado < atual.getInformacao()){ // caso o dado seja menor que o da folha
23                          atual.setEsquerdo(no); // insere a esquerda da folha
24                          break; // para o loop
25                      }
26                  }
27                  if(atual.getDireito() == null){ // se o ultimo da direita for folha
28                      if(dado >= atual.getInformacao()){ // caso o dado seja maior ou igual o da folha
29                          atual.setDireito(no); // insere a direita da folha
30                          break; // para o loop
31                      }
32                  }
33                  // condicoes de navegabilidade
34                  if(dado >= atual.getInformacao()){ // se o dado for maior que o do no
35                      atual = atual.getDireito(); // caminha pra direita
36                  } else { // se for menor
37                      atual = atual.getEsquerdo(); // caminha pra esquerda
38                  }
39              }
40          }
41          verificarArvore(no); // verifica se a arvore precisa ser balanceada e atualiza a arvore
```

O método de inserção recebe o dado que se deseja inserir na árvore e o armazena em um novo nó instanciado no começo do método, após isso é feita uma verificação que verifica se a árvore está vazia, se sim, aquele nó assume o lugar da raiz principal da árvore, se não, vai entrar em um loop que temo intuito de encontrar a posição ideal para aquele nó, de acordo com o valor recebido, se for maior que o no que está comparando, se sim o dado é comparado com o nó a direita, se menor, ele é comparado com o valor a esquerda, quando é encontrada a posição para o novo nó ele é inserido e o loop para.

Após a parada do loop é chamado o método verificarArvore que recebe como parâmetro o nó que acaba de ser inserido na árvore, dessa forma esse método por meio de chamadas recursivas verifica o balanceamento de todos os nós que foram afetados no processo de inserção.

Métodos de travessia:

```
52     private void preOrdem(NodeTree arvore){
53         if (arvore != null){ // se o no nao for nulo
54             System.out.print(arvore.getInformacao() + " "); // printa a informacao
55             preOrdem(arvore.getEsquerdo()); // chamada recursiva com o proximo da esquerda
56             preOrdem(arvore.getDireito()); // chamada recursiva com o proximo da direita
57         }
58     }
59     // metodo para imprimir in-ordem
60     3 usages
61     private void inOrdem(NodeTree arvore){
62         if(arvore != null){ // se o no nao for nulo
63             inOrdem(arvore.getEsquerdo()); // chamada recursiva com o proximo da esquerda
64             System.out.print(arvore.getInformacao() + " "); // printa a informacao
65             inOrdem(arvore.getDireito()); // chamada recursiva com o proximo da direita
66         }
67     }
68     // metodo para imprimir pos-ordem
69     3 usages
70     private void posOrdem(NodeTree arvore){
71         if(arvore != null){ // se o no nao for nulo
72             posOrdem(arvore.getEsquerdo()); // chamada recursiva com o proximo da esquerda
73             posOrdem(arvore.getDireito()); // chamada recursiva com o proximo da direita
74             System.out.print(arvore.getInformacao() + " "); // printa a informacao
75         }
76     }
77 }

40     public void imprimir(){
41         System.out.print("Pré-ordem: "); // tipo de organizacao
42         preOrdem(raiz); // metodo que imprime
43         System.out.println("\n"); // pula uma linha
44         System.out.print("In-ordem: "); // tipo de organizacao
45         inOrdem(raiz); // metodo que imprime
46         System.out.println("\n"); // pula uma linha
47         System.out.print("Pós-ordem: "); // tipo de organizacao
48         posOrdem(raiz); // metodo que imprime
49         System.out.println("\n"); // pula uma linha
50     }
```

Os métodos de travessia são o preOrdem, inOrdem e posOrdem que realizam chamadas recursivas a fim de percorrer e imprimir os dados da árvore no padrão desejado.

Pré-ordem: raiz , subárvore a esquerda e depois subárvore a direita

In-ordem: subárvore esquerda, raiz, subárvore direita

Pós-ordem: subárvore esquerda, subárvore direita, raiz

O método imprimir basicamente reúne esses métodos para imprimir as três travessias com uma única chamada e de forma organizada, facilitando a execução do print e entendimento da árvore.

Métodos de rotação:

```
151 @ private NodeTree rotacaoEsquerda(NodeTree no){
152     NodeTree novaRaiz = no.getDireito(); // seta a nova raiz como sendo o no da direita
153     no.setDireito(novaRaiz.getEsquerdo()); // seta a direita da antiga raiz o esquerdo da nova raiz
154     novaRaiz.setEsquerdo(no); // atualiza o esquerdo da nova raiz como sendo a raiz antiga
155     return novaRaiz; // retorna a raiz rotacionada
156 }
```

```
158 @ private NodeTree rotacaoDireita(NodeTree no){
159     NodeTree novaRaiz = no.getEsquerdo(); // seta a nova raiz como sendo o no da esquerda
160     no.setEsquerdo(novaRaiz.getDireito()); // seta a esquerda da raiz antiga o no direito da nova raiz
161     novaRaiz.setDireito(no); // atualiza a direita da nova raiz como sendo o no antigo
162     return novaRaiz; // retorna a raiz rotacionada
163 }
```

```
165 @ private NodeTree duplaRotacaoEsquerda(NodeTree no){
166     NodeTree novaRaiz = no.getDireito().getEsquerdo(); // pega a nova raiz da arvore que vai ser o neto da raiz que entrou
167     NodeTree pai = no.getDireito(); // armazena o pai
168     NodeTree vo = no; // armazena o vo
169     pai.setEsquerdo(novaRaiz.getDireito());
170     vo.setDireito(novaRaiz.getEsquerdo());
171     novaRaiz.setEsquerdo(vo); // seta o vo a esquerda da nova raiz
172     novaRaiz.setDireito(pai); // seta o pai a direita da nova raiz
173     return novaRaiz; // retorna a raiz balanceada
174 }
```

```
176 @ private NodeTree duplaRotacaoDireita(NodeTree no){
177     NodeTree novaRaiz = no.getEsquerdo().getDireito(); // pega a nova raiz da arvore que vai ser o neto da raiz que entrou
178     NodeTree pai = no.getEsquerdo(); // armazena o pai
179     NodeTree vo = no; // armazena o vo
180     pai.setDireito(novaRaiz.getEsquerdo());
181     vo.setEsquerdo(novaRaiz.getDireito());
182     novaRaiz.setDireito(vo); // seta o vo a direita da nova raiz
183     novaRaiz.setEsquerdo(pai); // seta o pai a esquerda da nova raiz
184     return novaRaiz; // retorna a raiz balanceada
185 }
```

Os métodos de rotação são responsáveis pela realização do balanceamento na árvore, são eles que entram em ação quando um nó da árvore atinge um balanceamento de -2 ou de 2, o nó que atingiu esse balanceamento entra como parâmetro do método e de acordo com a disposição dos elementos os mesmos são reposicionados, no caso das rotações>

Rotação simples direita: ocorre quando o nó está desbalanceado para esquerda (2), nele o nó desbalanceado vira filho direito do seu filho esquerdo, e caso já houvesse algum filho direito na nova raiz, o mesmo vira filho esquerdo da antiga raiz.

Rotação simples esquerda: ocorre quando o nó está desbalanceado para direita (-2), nele o nó desbalanceado vira filho esquerdo do seu filho direito, e caso já houvesse algum filho esquerdo na nova raiz, o mesmo vira filho direito da antiga raiz.

Rotação dupla direita: esse caso ocorre quando o nó está desbalanceado para esquerda (2) e possui um filho com o balanceamento -1 quando isso ocorre o filho direito do filho com balanceamento -1 vira a nova raiz, onde seu pai vira filho esquerdo e seu avô vira filho direito, caso a nova raiz já tivesse 2 filhos, o nó que estava a direita é reposicionado a esquerda do avô, e o que estava a esquerda é reposicionado a direita do pai.

Rotação dupla esquerda: esse caso ocorre quando o nó está desbalanceado para direita (-2) e possui um filho com o balanceamento 1 quando isso ocorre o filho esquerdo do filho com balanceamento 1 vira a nova raiz, onde seu pai vira filho direito e seu avô vira filho

esquerdo, caso a nova raiz já tivesse 2 filhos, o nó que estava a esquerda é reposicionado a direita do avo, e o que estava a direita é reposicionado a esquerda do pai.

Método de Balanceamento:

```
187 private NodeTree balancear(NodeTree no, int balanceamento){
188     NodeTree balanceado = null; // inicializa o no que vai ser retornado
189     if(balanceamento == -2){ // verifica se o no esta desbalanceado para direita
190         NodeTree filho = no.getDireito(); // armazena o filho
191         int verificaFilho = (altura(filho.getEsquerdo()) - altura(filho.getDireito())); // verifica o balanceamento do filho
192         if(verificaFilho == 1){ // se o balanceamento do filho for 1
193             balanceado = duplaRotacaoEsquerda(no); // faz uma dupla rotacao
194         } else { // se nao
195             balanceado = rotacaoEsquerda(no); // faz uma rotacao simples
196         }
197     } else if(balanceamento == 2){ // verifica se o no esta desbalanceado para a esquerda
198         NodeTree filho = no.getEsquerdo(); // armazena o filho
199         int verificaFilho = (altura(filho.getEsquerdo()) - altura(filho.getDireito())); // verifica o balanceamento do filho
200         if(verificaFilho == -1){ // se o balanceamento do filho for -1
201             balanceado = duplaRotacaoDireita(no); // faz uma dupla rotacao
202         } else { // se nao
203             balanceado = rotacaoDireita(no); // faz uma rotacao simples
204         }
205     }
206     return balanceado; // retorna o no balanceado
207 }
```

O método de balanceamento basicamente vai chamar e executar os métodos de rotação de acordo com cada caso, ele recebe o nó desbalanceado como parâmetro, juntamente com seu balanceamento que sempre será -2 ou 2 quando esse método for chamado, baseado nisso ele vai verificar o balanceamento do filho desse nó pra ver se ele vai precisar fazer um balanceamento duplo ou simples, depois ele vai fazer a rotação e retornar o nó já balanceado.

Método de verificação de Balanceamento:

```
209 private void verificarArvore(NodeTree no){
210     if(no == null){ // se o no for null
211         return; // para o metodo
212     }
213     int balanceamento = (altura(no.getEsquerdo()) - altura(no.getDireito())); // calcula o balanceamento do no baseado na diferença de altura de suas duas subarvores
214     NodeTree pai = encontrarPai(no);
215     if(pai != null){
216         if((balanceamento == 2) || (balanceamento == -2)){ // se o balanceamento for igual a 2 ou -2
217             NodeTree balanceado = balancear(no, balanceamento); // balanceia o no
218             if(balanceado.getInformacao() >= pai.getInformacao()){
219                 pai.setDireito(balanceado);
220             } else {
221                 pai.setEsquerdo(balanceado);
222             }
223         }
224         verificarArvore(pai);
225     } else {
226         if((balanceamento == 2) || (balanceamento == -2)){
227             raiz = balancear(raiz, balanceamento);
228         }
229     }
230 }
```

Esse método é chamado toda vez que um nó é inserido ou removido da árvore, ele tem uma chamada recursiva que varre de baixo pra cima todo o ramo da árvore que poderia ser afetado por aquela inserção/remoção, a cada nó que ele passa ele verifica seu balanceamento, caso o balanceamento seja -2 ou 2 ele vai chamar o método balancear que foi explicado anteriormente que irá retornar o nó balanceado de acordo com o caso, depois de retornado, o novo nó balanceado é recolocado na árvore no mesmo lugar, pois o pai foi armazenado para que o nó balanceado pudesse ser colocado no mesmo lugar em que estava antes do balanceamento.

Método para encontrar pai:

```
268 public NodeTree encontrarPai(NodeTree no) {  
269     if (no == null || no == raiz) {  
270         return null; // O nó alvo não tem pai ou é a raiz.  
271     }  
272     NodeTree percorre = raiz;  
273     NodeTree pai = null;  
274     while (percorre != null) {  
275         if ((percorre.getEsquerdo() == no) || (percorre.getDireito() == no)) {  
276             pai = percorre;  
277             break; // Encontramos o pai do nó alvo.  
278         } else if (no.getInformacao() >= percorre.getInformacao()) {  
279             percorre = percorre.getDireito();  
280         } else {  
281             percorre = percorre.getEsquerdo();  
282         }  
283     }  
284     return pai;  
285 }  
286 }
```

Esse método recebe um nó como parâmetro e localiza o pai no mesmo na árvore AVL, ele faz isso comparando nó a nó e percorrendo a árvore. Esse método foi implementado para que no método VerificarArvore o pai do nó fosse armazenado antes do filho ser balanceado para que ele fosse setado novamente após o balanceamento. A implementação desse método se fez necessária pois os nós da árvore não são duplamente encadeadas. Apesar de existir a possibilidade de se fazer uma árvore duplamente encadeada foi preferível a implementação desse método.

Método de remoção de raiz:

```
114 private void removerRaiz(){
115     if(isFolha(raiz)){ // se a raiz for folha
116         raiz = null; // iguala a null
117     } else { // se nao
118         if(temDireito(raiz) && temEsquerdo(raiz)){ // se o no tem direito e tem esquerdo
119             NodeTree novaRaiz = raiz.getDireito(); // nova raiz sera o no da direita
120             if(novaRaiz.getEsquerdo() == null){ // se a nova raiz nao tiver nenhum no a esquerda
121                 novaRaiz.setEsquerdo(raiz.getEsquerdo()); // posiciona a nova raiz da arvore setando no esquerdo o lado esquerdo da arvore
122             } else { // se a nova raiz tiver um no a esquerda
123                 NodeTree esq = novaRaiz.getEsquerdo(); // armazena
124                 while(esq.getEsquerdo() != null){ // enquanto nao estiver no ultimo no a esquerda da nova raiz
125                     esq = esq.getEsquerdo(); // percorre
126                 }
127                 esq.setEsquerdo(raiz.getEsquerdo()); // seta a esquerda desse no toda a subarvore esquerda da antiga raiz
128                 raiz = novaRaiz; // atualizo minha arvore
129             }
130         } else if(temEsquerdo(raiz) && !temDireito(raiz)){ // se a raiz tiver esquerdo e nao tiver direito
131             raiz = raiz.getEsquerdo(); // percorre um no
132         } else { // se tiver direito e nao tiver esquerdo
133             raiz = raiz.getDireito(); // percorre um no
134         }
135         verificarArvore(raiz);
136     }
137 }
```

Foi implementado um método que realiza a remoção da raiz da árvore para facilitar o processo no método remover caso o dado que o usuário queira remover corresponder a raiz da árvore, basicamente esse método verifica se a raiz é uma folha, ou seja, não possui nenhum filho, nesse caso a raiz vai ser igualada a null. Se não for uma folha os outros casos serão tratados, o caso em que a raiz tenha só filhos a direita, o caso em que a raiz tenha filhos só a esquerda e caso a raiz tenha filhos em ambos os lados. Basicamente em todos esses casos a raiz é removida e os nós restantes são reposicionados na árvore.

Após isso é chamado o método verificarArvore recebendo a raiz como parâmetro, dessa forma ele verifica se a nova raiz precisa de um balanceamento.

Método de remoção:

```
5 usages
298 public void remover(int dado){
299     NodeTree remover = raiz;
300     NodeTree pai = raiz;
301     NodeTree newFilho;
302     if(dado == raiz.getInformacao()){
303         removerRaiz();
304     } else {
305         while(dado != remover.getInformacao()){
306             if(dado >= remover.getInformacao()){
307                 pai = remover;
308                 remover = remover.getDireito();
309             } else {
310                 pai = remover;
311                 remover = remover.getEsquerdo();
312             }
313         }
314     }
315     if(isFolha(remover)){
316         if(remover.getInformacao() >= pai.getInformacao()){ // se o pai for maior que o removido
317             pai.setDireito(null); // apaga o da esquerda
318         } else { // se nao
319             pai.setEsquerdo(null); // apaga o da direita
320         }
321     } else {
322         if(temDireito(remover) && !temEsquerdo(remover)){
323             newFilho = remover.getDireito();
324             if(newFilho.getInformacao() >= pai.getInformacao()){
325                 pai.setDireito(newFilho);
326             } else {
327                 pai.setEsquerdo(newFilho);
328             }
329         }
330     }
331 }
```

```
313 } else {
314     if(temDireito(remover) && !temEsquerdo(remover)){
315         newFilho = remover.getDireito();
316         if(newFilho.getInformacao() >= pai.getInformacao()){
317             pai.setDireito(newFilho);
318         } else {
319             pai.setEsquerdo(newFilho);
320         }
321         verificarArvore(pai);
322     } else if (!temDireito(remover) && temEsquerdo(remover)){
323         newFilho = remover.getEsquerdo();
324         if(newFilho.getInformacao() >= pai.getInformacao()){
325             pai.setDireito(newFilho);
326         } else {
327             pai.setEsquerdo(newFilho);
328         }
329         verificarArvore(pai);
330     } else {
331         newFilho = remover.getDireito();
332         if(isFolha(newFilho) || (!temEsquerdo(newFilho) && temDireito(newFilho))){
333             if(newFilho.getInformacao() >= pai.getInformacao()){
334                 pai.setDireito(newFilho);
335             } else {
336                 pai.setEsquerdo(newFilho);
337             }
338             newFilho.setEsquerdo(remover.getEsquerdo());
339             verificarArvore(newFilho);
340         } else {
341             NodeTree menorPossivel = newFilho;
342             NodeTree paiMenorPossivel = newFilho;
343             while(menorPossivel.getEsquerdo() != null){
```



```

339         verificarArvore(newFilho);
340     } else {
341         NodeTree menorPossivel = newFilho;
342         NodeTree paiMenorPossivel = newFilho;
343         while(menorPossivel.getEsquerdo() != null){
344             paiMenorPossivel = menorPossivel;
345             menorPossivel = menorPossivel.getEsquerdo();
346         }
347         paiMenorPossivel.setEsquerdo(menorPossivel.getDireito());
348         verificarArvore(paiMenorPossivel);
349         newFilho = menorPossivel;
350         if(newFilho.getInformacao() >= pai.getInformacao()){
351             pai.setDireito(newFilho);
352         } else {
353             pai.setEsquerdo(newFilho);
354         }
355         newFilho.setDireito(remover.getDireito());
356         newFilho.setEsquerdo(remover.getEsquerdo());
357         verificarArvore(newFilho);
358     }
359 }
360 }
361 }

```

O método de remoção que foi o mais complexo ele faz basicamente as mesmas verificações do remover raiz só que de forma a valer para qualquer posição da árvore, no geral eu vou ter três variáveis, o removido, que vai corresponder ao nó que eu quero remover, o pai, que corresponde ao pai do que será removido e o newFilho que irá assumir a posição do nó que foi removido, caso seja necessário. Basicamente os nós pai e removido percorrem a árvore até encontrar o valor que foi passado como parâmetro no método remover e quando encontram o nó removido é setado como null ou deixado de lado e o pai recebe um novo filho, de forma a rearranjar a árvore

Após isso é chamado o método verificarArvore recebendo os nós que sofreram mudanças, de acordo com a situação do removido como parâmetro que vai varrer o ramo afetado pela remoção de baixo para cima e identificar se a ação gerou a necessidade de um balanceamento na árvore.

Método de Busca:

```
235 // metodo para buscar um elemento na arvore
    no usages
236 public int buscar(int dado){
237     int resultado;
238     NodeTree percorre = raiz;
239     while(true){
240         if(percorre.getInformacao() == dado){
241             resultado = dado;
242             break;
243         }
244         if(dado >= percorre.getInformacao()){
245             percorre = percorre.getDireito();
246         } else if (dado < percorre.getInformacao()){
247             percorre = percorre.getEsquerdo();
248         }
249     }
250     return resultado;
251 }
```

O método de busca possui um retorno do tipo int e recebe um parâmetro que corresponde ao valor que será buscado na árvore, o parâmetro é comparado com os nós da árvore até encontrar um valor correspondente ao desejado, quando ele encontra esse valor o valor é atribuído ao atributo resultado e o mesmo é retornado pelo método.

Métodos para Realizar verificações:

```
211 // método para verificar se o nó tem direito
212 // 4 usages
213 @ private boolean temDireito(NodeTree arvore){
214     if(arvore.getDireito() == null){ // se o nó a direita for null
215         return false; // não tem direito
216     } else { // se não
217         return true; // tem direito
218     }
219 }
220 // método para verificar se o nó tem esquerdo
221 // 4 usages
222 @ private boolean temEsquerdo(NodeTree arvore){
223     if(arvore.getEsquerdo() == null){ // se o nó a esquerda for null
224         return false; // não tem esquerdo
225     } else { // se não
226         return true; // tem esquerdo
227     }
228 }
229 // método para verificar se o nó é folha
230 // 6 usages
231 @ private boolean isFolha(NodeTree no){
232     if(no.getEsquerdo() == null && no.getDireito() == null){ // se o esquerdo e o direito for null
233         return true; // é folha
234     } else { // se não
235         return false; // não é folha
236     }
237 }
```

Para facilitar as operações feitas nos métodos de remoção foram feitos os seguintes métodos de verificação:

temDireito: esse método recebe um nó e verifica se o mesmo possui um filho a sua direita por meio de um retorno boolean

temEsquerdo: esse método recebe um nó e verifica se o mesmo possui um filho a sua esquerda por meio de um retorno boolean

isFolha: esse método recebe um nó e verifica se o mesmo é uma folha (não possui filhos) por meio de um retorno boolean

COMPARANDO INSERCAO E REMOÇÃO – ARVORE BINÁRIA X ARVORE AVL

Inserindo dados: 34 32 90 31 91 58 35 14 26 79 84 77 61 85 28 98 62 60

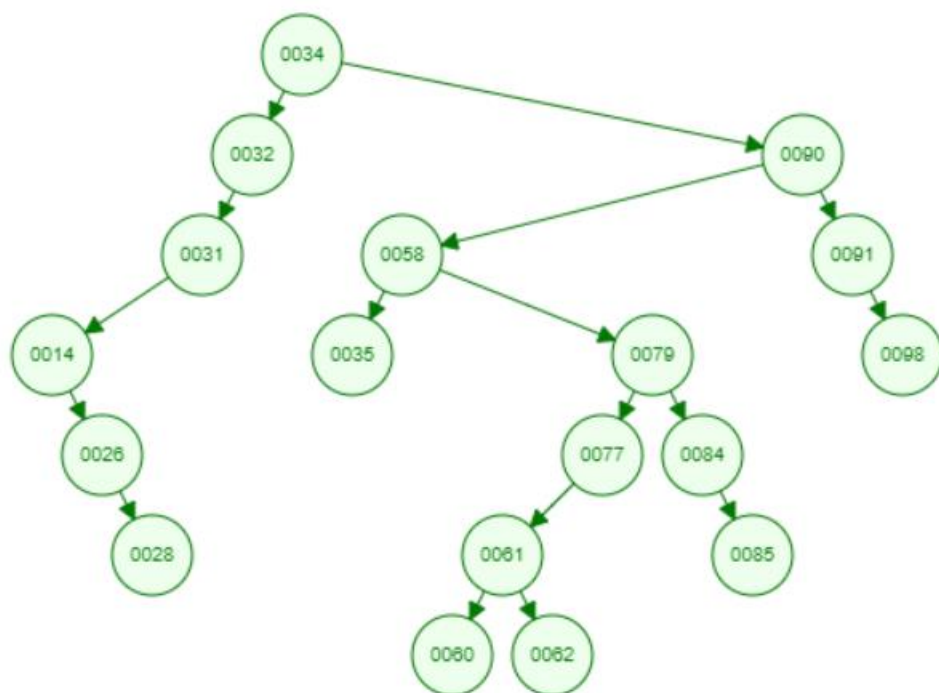
Árvore Binária:

```
6
7   ArvoreBinaria arvoreBinaria = new ArvoreBinaria();
8
9   arvoreBinaria.inserir( dado: 34);
10  arvoreBinaria.inserir( dado: 32);
11  arvoreBinaria.inserir( dado: 90);
12  arvoreBinaria.inserir( dado: 31);
13  arvoreBinaria.inserir( dado: 91);
14  arvoreBinaria.inserir( dado: 58);
15  arvoreBinaria.inserir( dado: 35);
16  arvoreBinaria.inserir( dado: 14);
17  arvoreBinaria.inserir( dado: 26);
18  arvoreBinaria.inserir( dado: 79);
19  arvoreBinaria.inserir( dado: 84);
20  arvoreBinaria.inserir( dado: 77);
21  arvoreBinaria.inserir( dado: 61);
22  arvoreBinaria.inserir( dado: 85);
23  arvoreBinaria.inserir( dado: 28);
24  arvoreBinaria.inserir( dado: 98);
25  arvoreBinaria.inserir( dado: 62);
26  arvoreBinaria.inserir( dado: 60);
27
28  arvoreBinaria.imprimir();
29
```

```
Run: Main x
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\Je
Árvore Binária:
Pré-ordem: 34 32 31 14 26 28 90 58 35 79 77 61 60 62 84 85 91 98

In-ordem: 14 26 28 31 32 34 35 58 60 61 62 77 79 84 85 90 91 98

Pós-ordem: 28 26 14 31 32 35 60 62 61 77 85 84 79 58 98 91 90 34
```



REPRESENTAÇÃO DA ÁRVORE BINÁRIA PELO SITE:

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

Árvore AVL:

```
29
30     ArvoreAvl arvoreAvl = new ArvoreAvl();
31
32     arvoreAvl.inserir( dado: 34);
33     arvoreAvl.inserir( dado: 32);
34     arvoreAvl.inserir( dado: 90);
35     arvoreAvl.inserir( dado: 31);
36     arvoreAvl.inserir( dado: 91);
37     arvoreAvl.inserir( dado: 58);
38     arvoreAvl.inserir( dado: 35);
39     arvoreAvl.inserir( dado: 14);
40     arvoreAvl.inserir( dado: 26);
41     arvoreAvl.inserir( dado: 79);
42     arvoreAvl.inserir( dado: 84);
43     arvoreAvl.inserir( dado: 77);
44     arvoreAvl.inserir( dado: 61);
45     arvoreAvl.inserir( dado: 85);
46     arvoreAvl.inserir( dado: 28);
47     arvoreAvl.inserir( dado: 98);
48     arvoreAvl.inserir( dado: 62);
49     arvoreAvl.inserir( dado: 60);
50
51     arvoreAvl.imprimir();
```

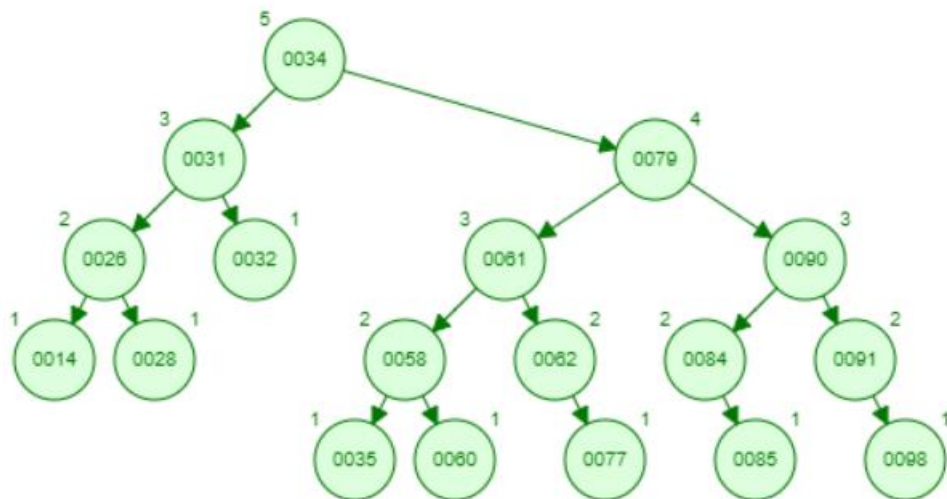
Run: ☐ Main x

Árvore AVL:

Pré-ordem: 34 31 26 14 28 32 79 61 58 35 60 62 77 90 84 85 91 98

In-ordem: 14 26 28 31 32 34 35 58 60 61 62 77 79 84 85 90 91 98

Pós-ordem: 14 28 26 32 31 35 60 58 77 62 61 85 84 98 91 90 79 34



REPRESENTAÇÃO DA ÁRVORE AVL PELO SITE:

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

Removendo dados: 14 28

Árvore Binária:

```

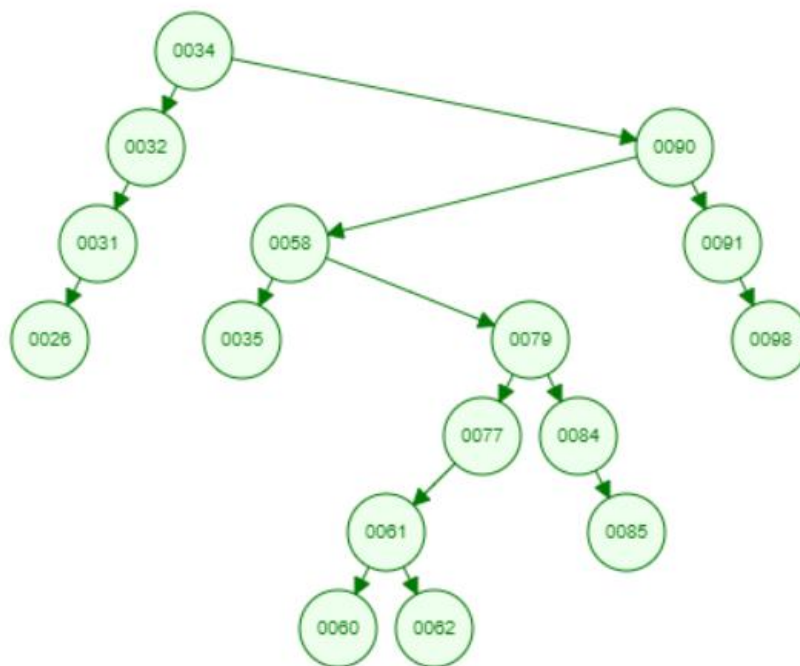
0   arvoreBinaria.remover( dado: 14);
1   arvoreBinaria.remover( dado: 28);
2
3   arvoreBinaria.imprimir();|

```

```

Run:  Main x
    "C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Pr
Árvore Binária:
Pré-ordem: 34 32 31 26 90 58 35 79 77 61 60 62 84 85 91 98
In-ordem: 26 31 32 34 35 58 60 61 62 77 79 84 85 90 91 98
Pós-ordem: 26 31 32 35 60 62 61 77 85 84 79 58 98 91 90 34

```



REPRESENTAÇÃO DA ÁRVORE BINÁRIA PELO SITE:

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

Árvore AVL:

```

arvoreAvl.remove( dado: 14);
arvoreAvl.remove( dado: 28);

arvoreAvl.imprimir();

```

```

"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program

```

```

Árvore AVL:

```

```

Pré-ordem: 79 34 31 26 32 61 58 35 60 62 77 90 84 85 91 98

```

```

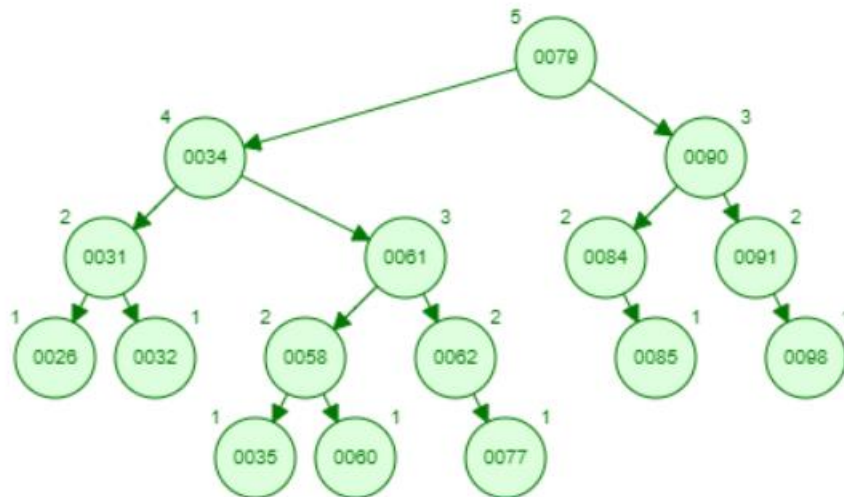
In-ordem: 26 31 32 34 35 58 60 61 62 77 79 84 85 90 91 98

```

```

Pós-ordem: 26 32 31 35 60 58 77 62 61 34 85 84 98 91 90 79

```

REPRESENTAÇÃO DA ÁRVORE AVL PELO SITE:

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

TESTES DE DESEMPENHO

Como será feito: Foram instanciados vetores com capacidades de 100, 500, 1000, 10000 e 20000 dados que serão preenchidos com números de 1 a 1000000, os dados desses vetores vão ser inseridos dentro de uma árvore AVL e uma árvore binária, o tempo será cronometrado em nanossegundos para ver qual das duas estruturas possui inserção mais eficiente

```
61
62 Random random = new Random();
63
64 int[] vetorDe100 = new int[100];
65 int[] vetorDe500 = new int[500];
66 int[] vetorDe1000 = new int[1000];
67 int[] vetorDe10000 = new int[10000];
68 int[] vetorDe20000 = new int[20000];
69
70 for(int i = 0; i < 100; i++){
71     int dado = random.nextInt( bound: 1000000) + 1;
72     vetorDe100[i] = dado;
73 }
74 for(int i = 0; i < 500; i++){
75     int dado = random.nextInt( bound: 1000000) + 1;
76     vetorDe500[i] = dado;
77 }
78 for(int i = 0; i < 1000; i++){
79     int dado = random.nextInt( bound: 1000000) + 1;
80     vetorDe1000[i] = dado;
81 }
82 for(int i = 0; i < 10000; i++){
83     int dado = random.nextInt( bound: 1000000) + 1;
84     vetorDe10000[i] = dado;
85 }
86 for(int i = 0; i < 20000; i++){
87     int dado = random.nextInt( bound: 1000000) + 1;
88     vetorDe20000[i] = dado;
89 }
90
```

Exemplo:

```
binariaInsercaoInicio = System.nanoTime();
for (int dado: vetorDe100) {
    arvoreBinaria.inserir(dado);
}
binariaInsercaoFim = System.nanoTime();

long binaria100 = binariaInsercaoFim - binariaInsercaoInicio;

avlInsercaoInicio = System.nanoTime();
for (int dado: vetorDe100) {
    arvoreAvl.inserir(dado);
}
avlInsercaoFim = System.nanoTime();

long avl100 = avlInsercaoFim - avlInsercaoInicio;
```

Resultados de inserção:

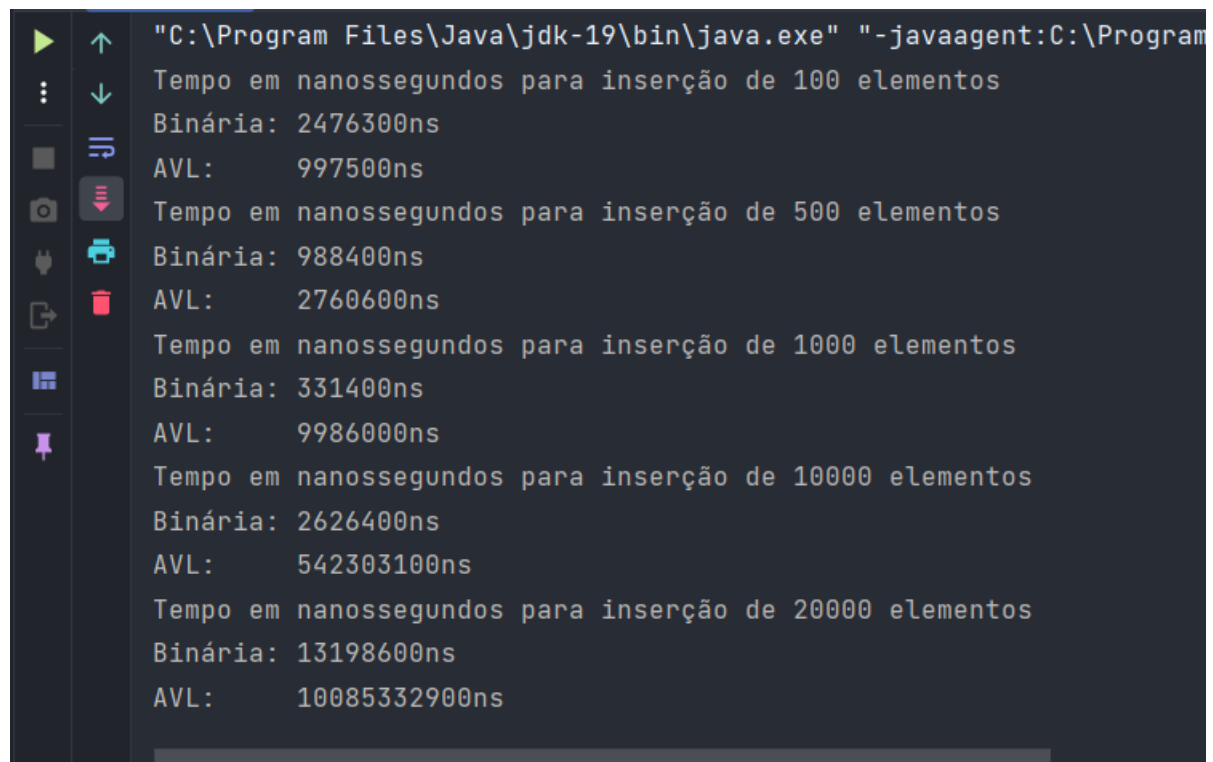
```
System.out.println("Tempo em nanossegundos para inserção de 100 elementos");
System.out.println("Binária: " + binaria100 + "ns");
System.out.println("AVL: " + avl100 + "ns");

System.out.println("Tempo em nanossegundos para inserção de 500 elementos");
System.out.println("Binária: " + binaria500 + "ns");
System.out.println("AVL: " + avl500 + "ns");

System.out.println("Tempo em nanossegundos para inserção de 1000 elementos");
System.out.println("Binária: " + binaria1000 + "ns");
System.out.println("AVL: " + avl1000 + "ns");

System.out.println("Tempo em nanossegundos para inserção de 10000 elementos");
System.out.println("Binária: " + binaria10000 + "ns");
System.out.println("AVL: " + avl10000 + "ns");

System.out.println("Tempo em nanossegundos para inserção de 20000 elementos");
System.out.println("Binária: " + binaria20000 + "ns");
System.out.println("AVL: " + avl20000 + "ns");
```



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program
Tempo em nanossegundos para inserção de 100 elementos
Binária: 2476300ns
AVL: 997500ns
Tempo em nanossegundos para inserção de 500 elementos
Binária: 988400ns
AVL: 2760600ns
Tempo em nanossegundos para inserção de 1000 elementos
Binária: 331400ns
AVL: 9986000ns
Tempo em nanossegundos para inserção de 10000 elementos
Binária: 2626400ns
AVL: 542303100ns
Tempo em nanossegundos para inserção de 20000 elementos
Binária: 13198600ns
AVL: 10085332900ns
```

Baseado nos resultados obtidos do teste de inserção é possível concluir que em casos onde é necessária a inserção de uma grande quantidade de dados a abordagem mais rápida é a estrutura de dados Binária, onde a mesma tem pouca variação de tempo entre a inserção de 100 elementos e 20000 elementos em relação a AVL. Isso ocorre pelo fato de que a árvore AVL a cada inserção executa uma série de tarefas a mais que a árvore binária, devido

as suas verificações e balanceamentos que consequentemente tornam o processo mais demorado

Teste de busca: da mesma forma que o teste de inserção, o teste de busca vai cronometrar em nanossegundos o tempo necessário para encontrar o mesmo elemento em cada árvore. Porém, para obter um resultado mais preciso a busca do elemento na árvore será feita repetidas vezes, e no final vai ser calculado a média do tempo onde:

$$\text{Tempo} = (\text{Tfinal} - \text{Tinicial}) / \text{numDeRepeticoes}$$

Exemplo:

```
197     int dado100 = vetorDe100[87];
198     int dado500 = vetorDe500[398];
199     int dado1000 = vetorDe1000[826];
200     int dado10000 = vetorDe10000[7692];
201     int dado20000 = vetorDe20000[15834];
202
203     int numRepeticoes = 50; // Número de repetições para obter uma média mais preci
204
205     // Para o elemento dado100
206     binariaInsercaoInicio = System.nanoTime();
207     for (int i = 0; i < numRepeticoes; i++) {
208         arvoreBinaria.buscar(dado100);
209     }
210     binariaInsercaoFim = System.nanoTime();
211     long bin100 = (binariaInsercaoFim - binariaInsercaoInicio) / numRepeticoes;
212
213     avlInsercaoInicio = System.nanoTime();
214     for (int i = 0; i < numRepeticoes; i++) {
215         arvoreAvl.buscar(dado100);
216     }
217     avlInsercaoFim = System.nanoTime();
218     long avl100 = (avlInsercaoFim - avlInsercaoInicio) / numRepeticoes;
219
220     binariaInsercaoInicio = System.nanoTime();
221     for (int i = 0; i < numRepeticoes; i++) {
222         arvoreBinaria.buscar(dado500);
223     }
```

```

System.out.println("Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes " + dado100);
System.out.println("Binaria: " + bin100 + "ns");
System.out.println("AVL: " + avl100 + "ns");

System.out.println("Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes " + dado500);
System.out.println("Binaria: " + bin500 + "ns");
System.out.println("AVL: " + avl500 + "ns");

System.out.println("Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes " + dado1000);
System.out.println("Binaria: " + bin1000 + "ns");
System.out.println("AVL: " + avl1000 + "ns");

System.out.println("Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes " + dado10000);
System.out.println("Binaria: " + bin10000 + "ns");
System.out.println("AVL: " + avl10000 + "ns");

System.out.println("Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes " + dado20000);
System.out.println("Binaria: " + bin20000 + "ns");
System.out.println("AVL: " + avl20000 + "ns");

```

Resultados:

```

Run: Main x
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Intel
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 818898
Binaria: 2066ns
AVL: 1168ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 360646
Binaria: 1866ns
AVL: 1722ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 263642
Binaria: 1884ns
AVL: 1664ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 209823
Binaria: 2960ns
AVL: 1696ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 823174
Binaria: 2526ns
AVL: 2210ns

```

Baseado nos resultados, é possível observar que a média do tempo de busca em nanossegundos com 50 iterações na árvore AVL é consideravelmente menor que na árvore binária, isso ocorre pela forma como é organizado os nós na árvore AVL que são todos balanceados, gerando uma árvore mais equilibrada e com mais facilidade de acessar algum elemento.

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ ID
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 400992
Binaria: 1800ns
AVL: 1153ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 970787
Binaria: 1004ns
AVL: 1542ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 863391
Binaria: 1861ns
AVL: 1515ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 525102
Binaria: 1905ns
AVL: 124ns
Media de tempo em nanossegundos para buscar o elemento com 50 repeticoes 973174
Binaria: 1359ns
AVL: 106ns
```

Outro teste feito, porém agora com 100 iterações em cada busca de cada árvore, os resultados seguem o mesmo padrão onde quanto mais elementos a árvore possui mais rápida é a busca da AVL em relação a Binária.

Teste de remoção: Assim como nos outros testes, será cronometrado em nanossegundos o tempo que cada uma das árvores leva para remover o mesmo elemento e os resultados serão comparados

```

295
296     binariaInsercaoInicio = System.nanoTime();
297     arvoreBinaria.remove(dado100);
298     binariaInsercaoFim = System.nanoTime();
299     long bin100 = binariaInsercaoFim - binariaInsercaoInicio;
300     avlInsercaoInicio = System.nanoTime();
301     arvoreAvl.remove(dado100);
302     avlInsercaoFim = System.nanoTime();
303     long avl100 = avlInsercaoFim - avlInsercaoInicio;
304
305     binariaInsercaoInicio = System.nanoTime();
306     arvoreBinaria.remove(dado500);
307     binariaInsercaoFim = System.nanoTime();
308     long bin500 = binariaInsercaoFim - binariaInsercaoInicio;
309     avlInsercaoInicio = System.nanoTime();
310     arvoreAvl.remove(dado500);
311     avlInsercaoFim = System.nanoTime();
312     long avl500 = avlInsercaoFim - avlInsercaoInicio;
313
314     binariaInsercaoInicio = System.nanoTime();
315     arvoreBinaria.remove(dado1000);
316     binariaInsercaoFim = System.nanoTime();
317     long bin1000 = binariaInsercaoFim - binariaInsercaoInicio;
318     avlInsercaoInicio = System.nanoTime();
319     arvoreAvl.remove(dado1000);
320     avlInsercaoFim = System.nanoTime();
321     long avl1000 = avlInsercaoFim - avlInsercaoInicio;
322

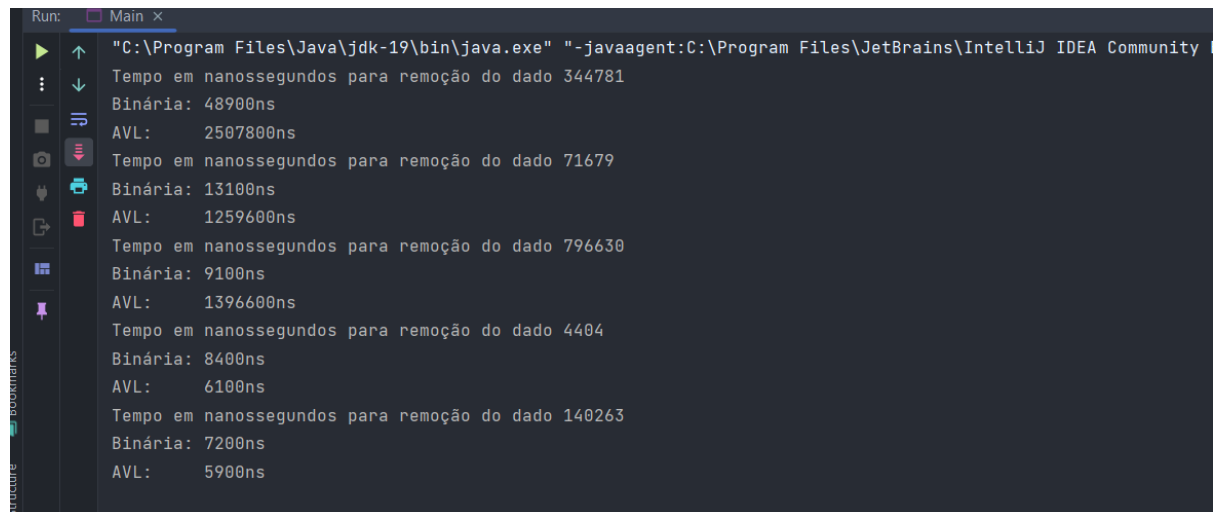
```

```

340
341     System.out.println("Tempo em nanossegundos para remoção do dado " + dado100);
342     System.out.println("Binária: " + bin100 + "ns");
343     System.out.println("AVL: " + avl100 + "ns");
344
345     System.out.println("Tempo em nanossegundos para remoção do dado " + dado500);
346     System.out.println("Binária: " + bin500 + "ns");
347     System.out.println("AVL: " + avl500 + "ns");
348
349     System.out.println("Tempo em nanossegundos para remoção do dado " + dado1000);
350     System.out.println("Binária: " + bin1000 + "ns");
351     System.out.println("AVL: " + avl1000 + "ns");
352
353     System.out.println("Tempo em nanossegundos para remoção do dado " + dado10000);
354     System.out.println("Binária: " + bin10000 + "ns");
355     System.out.println("AVL: " + avl10000 + "ns");
356
357     System.out.println("Tempo em nanossegundos para remoção do dado " + dado20000);
358     System.out.println("Binária: " + bin20000 + "ns");
359     System.out.println("AVL: " + avl20000 + "ns");
360 }
361 }

```


Resultados:



```
Run: Main x
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community B
Tempo em nanossegundos para remoção do dado 344781
Binária: 48900ns
AVL: 2507800ns
Tempo em nanossegundos para remoção do dado 71679
Binária: 13100ns
AVL: 1259600ns
Tempo em nanossegundos para remoção do dado 796630
Binária: 9100ns
AVL: 1396600ns
Tempo em nanossegundos para remoção do dado 4404
Binária: 8400ns
AVL: 6100ns
Tempo em nanossegundos para remoção do dado 140263
Binária: 7200ns
AVL: 5900ns
```

Com base nos resultados é possível observar que na minha implementação a árvore AVL possui desempenho na remoção superior a binária somente a partir de 10000 elementos, em teoria era esperado ela ter desempenho superior em todos os casos, por conta da disposição dos elementos. Porém, esse atraso nos outros casos pode ser por conta da implementação do método de remoção na árvore AVL que talvez pudesse ser implementado de forma mais otimizada.

CONCLUSÃO

O trabalho em questão tinha o seguinte propósito: a implementação de uma Árvore Binária de Busca com os métodos de inserção, busca e remoção; a implementação de uma Árvore AVL de Busca com os métodos de inserção, busca e remoção; uma análise crítica e comparativa de desempenho das duas estruturas de dados exercendo algumas funções.

No geral, todos os requisitos foram implementados e os resultados da experiência foram documentados no relatório.

No decorrer da implementação:

Dificuldades: entender com clareza como tirar da teoria as rotações e passar para implementação de forma eficiente; encontrar um meio de balancear minha árvore assim que necessário.

Todas as dificuldades foram superadas no decorrer do trabalho.

Bugs solucionados: bugs de rotação onde ocorria um desaparecimento de um nó ou parte da árvore, bugs de rotação onde os nós não eram reposicionados de maneira correta o que acarretava em um loop infinito de rotações causando StackOverflow Error no cálculo da altura, bugs de varredura da árvore.

Bug não solucionado: em alguns casos onde é feita uma tentativa de inserção na arvore AVL onde é estipulada uma quantidade x de elementos a serem inseridos em um range de y a z ocorre um erro na árvore, onde o código aponta como null toda a parte esquerda da árvore a partir da raiz principal, apesar dos elementos de fato estarem lá, os mesmos não são identificados

Exemplos:

Tentativa de inserir 1000 elementos aleatórios de 1 a 100

```
Random random = new Random();
for(int i = 0; i < 1000; i++){
    int dado = random.nextInt( bound: 100) + 1;
    arvoreAvl.inserir(dado);
}
```

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar=54114:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar" 54114
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "NodeTree.getEsquerdo()" because "filho" is null
    at ArvoreAvl.balancear(ArvoreAvl.java:201)
    at ArvoreAvl.verificarArvore(ArvoreAvl.java:229)
    at ArvoreAvl.verificarArvore(ArvoreAvl.java:226)
    at ArvoreAvl.verificarArvore(ArvoreAvl.java:226)
    at ArvoreAvl.inserir(ArvoreAvl.java:41)
    at Main.main(Main.java:65)
```

Ocorre o erro onde toda a subarvore esquerda da raiz principal não e identificada

Agora caso ao invés de 1000 eu insira 100 elementos de 1 a 100

```
62 Random random = new Random();
63 for(int i = 0; i < 100; i++){
64     int dado = random.nextInt( bound: 100) + 1;
65     arvoreAvl.inserir(dado);
66 }
```

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar=54114:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar" 54114
Árvore AVL:
Pré-ordem: 55 26 12 4 2 1 5 5 9 8 9 19 17 13 18 18 21 20 19 20 22 24 44 37 31 29 29 29 29 30 33 32 33 39 38 38 38 40 40 42 42 43 47 45 44 44 44 46 52 48 47 50 48 53 52 53 53 8
In-ordem: 1 2 4 5 5 8 9 9 12 13 17 18 18 19 19 20 20 21 22 24 26 29 29 29 29 30 31 32 33 33 37 38 38 38 39 40 40 42 42 43 44 44 44 44 45 45 46 47 47 48 48 50 52 52 53 53 53 55 55
Pós-ordem: 1 2 5 8 9 9 5 4 13 18 18 17 19 20 20 24 22 21 19 12 29 29 30 29 29 32 33 33 31 38 38 38 40 42 43 42 40 39 37 44 44 44 46 45 45 47 48 50 48 52 53 53 53 52 47 44 26 55 5
Process finished with exit code 0
```

Nesse caso a inserção dos elementos ocorre sem problema

Da mesma forma, se eu inserir 1000 valores de 1 a 1000

```
1
2 Random random = new Random();
3 for(int i = 0; i < 1000; i ++){
4     int dado = random.nextInt(bound: 1000) + 1;
5     arvoreAvl.inserir(dado);
6 }
7 arvoreAvl.imprimir();
8
```

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar=54152:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.3.2\bin" -Dfile.encoding=UTF-8
Arvore AVL:
Pré-ordem: 2 3 4 3 14 13 7 6 13 28 19 14 20 19 30 28 23 23 20 24 23 28 28 29 28 51 39 34 32 37 35 38 44 41 41 39 41 41 48 47 49 77 62 59 56 56 51 57 56 58 60 60 61 69 63 63
In-ordem: 2 3 3 4 6 7 13 13 14 14 19 19 20 20 20 23 23 24 24 28 28 28 28 28 29 30 32 34 35 37 38 39 39 41 41 41 41 44 47 48 49 51 51 56 56 56 57 58 59 60 60 60 61 62 62 63 63 63
Pós-ordem: 3 6 7 13 13 14 19 20 19 20 23 28 24 23 28 28 29 28 32 35 38 37 34 39 41 41 41 41 47 49 48 44 39 51 56 56 58 57 56 60 61 60 60 59 62 63 63 68 66 64 63 70 73 73 73

Process finished with exit code 0
```

A inserção também ocorre com sucesso

Essa anomalia ocorre na árvore em alguns casos como esse, e alguns outros, a variar entre o número de elementos e o intervalo dos números inseridos

Tentativas de solucionar o Bug: revisões do código implementado na árvore AVL sem sucesso na identificação do problema; Refatoração do código de implementação da árvore AVL feita mais de 1 vez e sem sucesso na resolução do bug; Dúvida tirada com o professor, onde o mesmo não identificou o problema que está causando o Bug; Dúvida tirada com um programador com alguns anos de experiência em java, onde o mesmo sugeriu que eu refatorasse o código utilizando outra lógica, não fui capaz de pensar em outra maneira de refazer o código sem ser a lógica que eu elaborei.