```java
package com.clerton.leal;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        boolean isRunning = true;
        Tree tree = new Tree();
        Scanner scanner = new Scanner(System.in);
        while(isRunning){
            String option = getUserOption(scanner);
            InputType inputType = validateUserOption(option);
            switch (inputType){
                case INSERT:
                    try{
                        int value = getUserValue(scanner);
                        tree.addNode(value);
                        tree.printTree();
                    } catch (NumberFormatException e){
                        System.out.println("Valor digitado não é um numero");
                    }

                    break;

                case REMOVE:
                    try{
                        int value = getUserValue(scanner);
                        tree.removeNode(value);
                        tree.printTree();
                    } catch (NumberFormatException e){
                        System.out.println("Valor digitado não é um numero");
                    }

                    break;

                case REPORT:
                    tree.report();
                    break;

                case QUIT:
                    isRunning = false;
                    System.out.println("Programa finalizado.");
                    break;

                case INVALID:
                    System.out.println("Opção invalida. Tente novamente.");
                    break;
            }

        }
    }

    private static Integer getUserValue(Scanner scanner) throws NumberFormatExceptio
n{
        System.out.println("Insira o valor do nó:");
        final String value = scanner.nextLine();
```

```
59          return Integer.parseInt(value);
60      }
61
62      private static String getUserOption(Scanner scanner){
63          System.out.println("Digite I para inserir um nó");
64          System.out.println("Digite R para remover um nó");
65          System.out.println("Digite E para ver as estatisticas da arvore");
66          System.out.println("Digite S para sair do programa");
67          return scanner.nextLine();
68      }
69
70      private static InputType validateUserOption(String input){
71          if(input.equalsIgnoreCase("I")){
72              return InputType.INSERT;
73          } else if(input.equalsIgnoreCase("R")){
74              return InputType.REMOVE;
75          } else if(input.equalsIgnoreCase("E")){
76              return InputType.REPORT;
77          } else if(input.equalsIgnoreCase("S")){
78              return InputType.QUIT;
79          } else{
80              return InputType.INVALID;
81
82          }
83      }
84
85      static enum InputType{
86          INSERT,
87          REMOVE,
88          REPORT,
89          QUIT,
90          INVALID;
91      }
92
93      public static class Tree {
94          Node root;
95
96          public Node getRoot() {
97              return root;
98          }
99
100         public void addNode(int key) {
101             addNode(new Node(key));
102             System.out.println("Adicionado no " + key);
103         }
104
105         public void addNode(Node node) {
106             root = addNodeRecursive(root, node);
107         }
108
109         public Node addNodeRecursive(Node root, Node node) {
110             if (root == null) {
111                 root = node;
112             } else if (node.getKey() == root.getKey()) {
113                 node.setLeft(root.getLeft());
114                 node.setRight(root.getRight());
115                 root = node;
116             } else if (node.getKey() < root.getKey()) {
117                 root.setLeft(addNodeRecursive(root.getLeft(), node));
```

```
118             } else {
119                 root.setRight(addNodeRecursive(root.getRight(), node));
120             }
121 
122             return root;
123         }
124 
125         public Node get(int key) {
126             return getNodeRecursive(root, key);
127         }
128 
129         private Node getNodeRecursive(Node root, int key) {
130             if (root == null || key == root.getKey()){
131                 return root;
132             } else if (key < root.getKey()) {
133                 return getNodeRecursive(root.getLeft(), key);
134             } else{
135                 return getNodeRecursive(root.getRight(), key);
136             }
137         }
138 
139         public void removeNode(int key) {
140             root = removeNodeRecursive(root, key);
141             System.out.println("Removido no " + key);
142         }
143 
144         private Node removeNodeRecursive(Node root, int key) {
145             if (root == null){
146                 System.out.println("O no " + key + " Não existe na arvore");
147                 return null;
148             } else if (key == root.getKey()) {
149                 Node newRoot = root.getLeft();
150                 if(newRoot == null){
151                     newRoot = root.getRight();
152                 } else if (root.getRight() != null) {
153                     Node smallestRight = getSmallest(root.getRight());
154                     smallestRight.setLeft(newRoot.getRight());
155                     newRoot.setRight(root.getRight());
156                 }
157                 root = newRoot;
158             } else if (key < root.getKey()){
159                 root.setLeft(removeNodeRecursive(root.getLeft(), key));
160             } else{
161                 root.setRight(removeNodeRecursive(root.getRight(), key));
162             }
163 
164             return root;
165         }
166 
167         private Node getSmallest(Node root) {
168             while (root.getLeft() != null) {
169                 root = root.getLeft();
170             }
171 
172             return root;
173         }
174 
175         public List<Node> list() {
176             List<Node> arrayList = new ArrayList<Node>();
```

```
177         listRecursive(arrayList, root);
178         return arrayList;
179     }
180
181     private void listRecursive(List<Node> nodeList, Node root) {
182         if (root == null){
183             return;
184         }
185
186         listRecursive(nodeList, root.getLeft());
187         nodeList.add(root);
188         listRecursive(nodeList, root.getRight());
189     }
190
191     public void printTree() {
192         for(PrintType printType : PrintType.values()){
193             printTreeByType(printType);
194         }
195         System.out.print("\n");
196     }
197
198     public void printTreeByType(PrintType printType) {
199         switch (printType) {
200             case PRE_ORDER:
201                 printPreOrderTree();
202                 break;
203             case POS_ORDER:
204                 printPosOrderTree();
205                 break;
206             case HEIGHT:
207                 printInLevelOrder();
208                 break;
209             case IN_ORDER:
210                 printInOrderTree();
211                 break;
212             default:
213                 break;
214         }
215     }
216
217     private void printPosOrderTree() {
218         System.out.print("Pos-order: ");
219         posOrderTree(root);
220         System.out.print("\n");
221     }
222
223     private void posOrderTree(Node node) {
224         if(node != null) {
225             posOrderTree(node.left);
226             posOrderTree(node.right);
227             System.out.print( node.getKey() + " " );
228         }
229     }
230
231     public void printPreOrderTree() {
232         System.out.print("Pre-order: ");
233         preOrderTree(root);
234         System.out.print("\n");
235     }
```

```
236
237         public void preOrderTree(Node node) {
238             if(node != null) {
239                 System.out.print( node.getKey() + " " );
240                 preOrderTree(node.left);
241                 preOrderTree(node.right);
242             }
243         }
244
245
246         private void printInOrderTree() {
247             System.out.print("In-order: ");
248             inOrderTree(root);
249             System.out.print("\n");
250         }
251
252         public void inOrderTree(Node node) {
253             if( node != null ) {
254                 inOrderTree(node.left);
255                 System.out.print( node.getKey() + " " );
256                 inOrderTree(node.right);
257             }
258         }
259
260         public void printInLevelOrder(){
261             System.out.print("Altura: ");
262             int h = treeDepth(root);
263             for(int i=1; i<=h; i++){
264                 printInLevelOrder(root, i); //print every level
265             }
266
267             System.out.print("\n");
268         }
269
270         public void printInLevelOrder(Node node, int level){
271             if(node ==null){
272                 return;
273             }
274             if(level == 1){
275                 System.out.print(node.getKey() + " ");
276             } else if (level > 1){
277                 printInLevelOrder(node.left, level - 1);
278                 printInLevelOrder(node.right, level - 1);
279             }
280         }
281
282         private int treeSize() {
283             return treeSize(root);
284         }
285
286         private int treeSize(Node node) {
287             if( node == null ) {
288                 return 0;
289             }
290
291             int leftSize = treeSize(node.left);
292             int rightSize = treeSize(node.right);
293
294             return leftSize + rightSize + 1;
```

```
295         }
296 
297         private int treeDepth() {
298             return treeDepth(root);
299         }
300 
301         private int treeDepth(Node node) {
302             if( node == null ) {
303                 return 0;
304             }
305 
306             int leftDepth = treeDepth(node.left);
307             int rightDepth = treeDepth(node.right);
308 
309             if(leftDepth < rightDepth){
310                 return rightDepth + 1;
311             } else{
312                 return leftDepth + 1;
313             }
314         }
315 
316         private int minimumNodeTree() {
317             return minimumNodeTree(root);
318         }
319 
320         private int minimumNodeTree(Node node) {
321             while( node.left != null ) {
322                 node = node.left;
323             }
324 
325             return node.getKey();
326         }
327 
328         private int maximumNodeTree() {
329             return maximumNodeTree(root);
330         }
331 
332         public int maximumNodeTree(Node node) {
333             while( node.right != null ) {
334                 node = node.right;
335             }
336 
337             return node.getKey();
338         }
339 
340         public void report(){
341             System.out.println("Quantidade de nós: " + treeSize());
342             System.out.println("Altura da arvore: " + treeDepth());
343             System.out.println("Nó mais alto: " + maximumNodeTree());
344             System.out.println("Nó mais baixo: " + minimumNodeTree());
345         }
346 
347         public class Node  {
348             int key;
349             Node left;
350             Node right;
351 
352             Node(int key) {
353                 this.key = key;
```

```
354         }
355
356         public int getKey() {
357             return key;
358         }
359
360         public Node getLeft() {
361             return left;
362         }
363
364         public void setLeft(Node left) {
365             this.left = left;
366         }
367
368         public Node getRight() {
369             return right;
370         }
371
372         public void setRight(Node right) {
373             this.right = right;
374         }
375     }
376
377     public enum PrintType{
378         PRE_ORDER,
379         POS_ORDER,
380         IN_ORDER,
381         HEIGHT;
382     }
383
384 }
385 }
386
```