

# Structure de données : Rapport sur les Arbres Binaires de Recherche

Rédigé et présenté par :

- Cédric Alonso
- Jade Hatoum

Année académique 2023 - 2024

# SOMMAIRE

## I. Arbre Binaire de Recherche

# Introduction

Structure Générale : Au cours de ce semestre, nous avons étudié et appliqué diverses structures permettant de faciliter la gestion et la manipulation des données. Cependant, presque aucunes ne fournissaient des méthodes d'organisation ordonnées à une telle échelle que celle des arbres, précisément les Arbres Binaires de Recherche (ABR).

Ces derniers permettent d'enregistrer des valeurs comparables (dans notre cas des entiers), en les allouant aux nœuds de l'arbre. Afin de comprendre comment ils se comportent, il suffit de connaître la composition de cet arbre. En effet, chaque nœud “source” possède deux nœuds “fils” : un à gauche stockant une valeur inférieure ou égale à celle de la source, et un à droite stockant une valeur strictement supérieur à celle de la source. De nouveaux nœuds fils sont créés et rattachés sous ces derniers, lors de chaque nouvelle insertion d'une valeur.

La particularité des ABR se repose sur son extensibilité. Cette insertion de valeurs est reproduite afin de pouvoir enregistrer un flux de données dans un arbre binaire de manière retracable. Il permet donc de retrouver un nœud portant la valeur recherchée, grâce à sa propriété de comparaison.

En bref, un ABR est un arbre binaire à hauteur variable comprenant des nœuds et leurs deux nœuds fils à valeur comparable.

Comme toujours, nous utiliserons le langage Java pour les différentes implémentations. De plus, un menu sur console est mis à disposition afin de manipuler l'ensemble des méthodes.

# I. Arbre Binaire de Recherche

## 1. Algorithme explicite

Dans ce TP, nous couvrons les méthodes fondamentales de la manipulation d'un ABR : la suppression d'un nœud portant une valeur précise, la recherche d'une valeur d'un nœud, l'ajout continu des nœuds à valeurs indiquées dans l'arbre, et le calcul de nombreuses statistiques liées à l'ABR tels que sa traversée en ordres préfixe, infixé et postfixe.

### La suppression :

Pour ce faire, il existe la méthode *removing*, qui après avoir récupéré la valeur entière *value*, utilise la méthode *remove* de BST.java. Ceci permet de rechercher puis supprimer un nœud portant *value*, tout en préservant la structure de l'ABR.

Elle utilise une approche récursive, similaire à celle de la méthode *search*, pour localiser le nœud à supprimer. Une fois le nœud trouvé, nous commençons d'abord par traiter la condition d'arrêt suivante. Si le nœud est une feuille, il est simplement retiré.

Et pour le reste, si le nœud a un seul sous-arbre, ce sous-arbre remplace le nœud à supprimer, afin de réunir les deux ABR en un arbre. Si le nœud a deux sous-arbres, sa valeur est remplacée par la valeur minimale du sous-arbre droit, et une récursion est effectuée pour supprimer cette valeur minimale. Cela assure que la propriété de l'ABR est maintenue.

### La recherche :

La recherche est effectuée à partir de la méthode *searching*, qui fait recours à la méthode *search* qui elle vérifie si une valeur spécifiée est présente dans l'ABR.

Elle aussi utilise une approche récursive pour parcourir l'arbre. Ses deux conditions d'arrêts couvrent les cas suivants. Si le nœud actuel *Node node* est nul, la valeur n'est pas présente. Et si la valeur actuelle du nœud (*node.getValue()*) correspond à la valeur recherchée (*value*). Sinon, elle poursuit la recherche en parcourant récursivement le sous-arbre gauche (*Node source.left*) ou droit (*Node source.right*), en fonction de la comparaison.

### L'ajout :

La méthode *adding* gère cet ajout depuis Main.java, en proposant à l'utilisateur de préciser la valeur entière à rajouter tant qu'il ne décide pas d'arrêter. La méthode *insert* dans BST.java est quant à elle responsable d'ajouter récursivement une valeur entière *value* à partir d'un *Node source*.

Cette approche récursive lui permet de parcourir l'arbre jusqu'à trouver l'emplacement approprié pour la nouvelle valeur. Si le nœud actuel est nul, un nouveau nœud est créé avec la valeur à insérer. En fonction de la comparaison entre la valeur (*value*) à insérer et la valeur du nœud actuel (*source.getValue()*), la récursion se poursuit dans le sous-arbre gauche (*Node source.left*) ou droit (*Node source.right*).

### Le calcul des statistiques :

Depuis la méthode *displayStatistics*, il est possible de calculer et obtenir diverses statistiques de l'instance de l'ABR (*BST bst*) modifié jusqu'à présent. Ci-dessous se retrouvent l'ensemble des méthodes permettant d'effectuer ces calculs.

- Traversées ordonnées (en ordre préfixe, infixé et postfixe) :

Les méthodes de traversée *inOrder*, *preOrder*, *postOrder* réalisent respectivement des traversées en ordre, préordonnée et postordonnée de l'ABR. Ces traversées sont utiles pour obtenir les valeurs de l'ABR dans différents ordres, comme trié, préordonné (pour la copie), et postordonné (pour la libération de mémoire).

Chaque méthode utilise une approche récursive pour explorer l'arbre dans l'ordre souhaité, ajoutant les valeurs des nœuds visités à la liste d'entiers *traversal*. Si le nœud actuel *node* n'est pas nul, on appellera la méthode appropriée sur le fils gauche puis le fils droit du nœud *node*, tout en rajoutant à la liste *traversal* la valeur du nœud actuel.

Ces trois actions sont toutes présentes dans les trois méthodes, la seule différence étant quand on ajoute la valeur à la liste. Les noms des méthodes sont auto-explicatifs ce qui nous aide énormément : avant le parcours des fils en ordre préfixe, entre le parcours du fils gauche et droit en ordre infixé, et enfin, après le parcours des fils en ordre postfixe.

- Parcours en largeur :

La méthode *breadthFirstSearch* retourne le parcours en largeur de l'ABR, en Ainsi, en traversant l'arbre horizontalement, explorant tous les nœuds du niveau actuel avant de passer au niveau suivant.

Pour ce faire, on fait recours à la queue *Queue* que nous avions défini au TP5. Initialement, on crée une instance de la queue et on effectue un *push* sur le nœud *source* d'abord. Ensuite, on entre dans une boucle while, où chaque itération retire le nœud à l'avant de la file avec *pop* et ajoute la valeur de ce nœud à la liste de traversée *traversal*. Ensuite, les enfants gauche et droit de ce nœud, s'ils existent, sont ajoutés à la file avec *push*.

Ce processus se répète jusqu'à ce que la file soit vide, garantissant ainsi que chaque niveau de l'ABR est visité avant de passer au niveau suivant.

- Hauteur :

La méthode *height* est responsable du calcul de l'hauteur d'un arbre à partir d'un nœud *node* donné, donc le nombre de niveaux sous ce nœud.

Elle accomplit cela en suivant une approche récursive. D'abord, la condition d'arrêt est vérifiée ; si le nœud actuel *node* est nul, nous retournons une valeur 0 à rajouter à la somme des niveaux. Sinon, elle fait appel à elle-même sur le fils gauche et droit, sélectionne la valeur maximale entre ces deux dernières et rajoute 1, afin d'incrémenter l'hauteur.

- Valeurs minimale et maximale :

Les méthodes *getMinValue* et *getMaxValue* récupèrent respectivement la valeur minimale et maximale de l'ABR.

Pour obtenir la valeur minimale, les deux méthodes exploitent la propriété de l'ABR. Ainsi, la méthode *getMinValue* parcourt l'ABR en se déplaçant constamment vers la gauche jusqu'à atteindre le nœud le plus à gauche, dont la valeur est renvoyée. D'autre part, la méthode *getMaxValue* se déplace constamment vers la droite jusqu'à atteindre le nœud le plus à droite, renvoyant ainsi la valeur maximale de l'ABR.

- Valeur moyenne :

La méthode *getMeanValue* calcule la valeur moyenne des nœuds dans l'ABR.

Pour ce faire, elle utilise une des traversées tel que *inOrder* pour obtenir toutes les valeurs stockées dans l'arbre. Ensuite, elle parcourt la liste des valeurs, les additionne et divise le total par le nombre de valeurs pour obtenir la moyenne.

- Nombre de nœuds :

La méthode *nbNodes* compte le nombre de nœuds non-nuls dans l'ABR.

Elle utilise une approche récursive pour explorer l'arbre. Si le nœud actuel *node* n'est pas nul, la méthode rappelle récursivement *nbNodes* sur le fils gauche et le fils droit de *node*. La somme de ces deux résultats, ajoutée à 1 (représentant le nœud actuel non-nul), donne le nombre total de nœuds non-nuls dans l'ABR.

## 2. Le code réalisé

```
package TP6;

/**
 * Exception levée lorsqu'une opération est effectuée sur une file.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
18 usages  ≈ cedric.alonso
public class BSTException extends Exception {

    /**
     * Construit une nouvelle exception avec le message spécifié concernant les arbres de recherches binaires.
     *
     * @param message Le message d'erreur associé à l'exception.
     */
    4 usages  ≈ cedric.alonso
    public BSTException(String message) { super("\u001B[31m" + message + "\u001B[0m"); }

}
```

```
package TP6;

/**
 * Représentation d'un nœud dans un arbre de recherche binaire (BST).
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
22 usages  ≈ mrredcoding +1
public class Node {

    5 usages
    private Integer value;
    4 usages
    private Node left;
    4 usages
    private Node right;

    /**
     * Constructeur d'un nœud sans valeur, gauche, ni droite.
     */
    1 usage  ≈ cedric.alonso
    public Node() {
        this.value = null;
        this.left = null;
        this.right = null;
    }

    /**
     * Constructeur d'un nœud avec une valeur donnée, sans nœuds gauche ni droit.
     *
     * @param valeur La valeur du nœud.
     */
    1 usage  ≈ cedric.alonso +1
    public Node(Integer valeur) {
        this.value = valeur;
        this.left = null;
        this.right = null;
    }
}
```

```
/*
 * Obtient la valeur du nœud.
 *
 * @return La valeur du nœud.
 */
± cedric.alonso
public Integer getValue() {
    return value;
}

/**
 * Définit la valeur du nœud.
 *
 * @param valeur La nouvelle valeur du nœud.
 */
± mrredcoding
public void setValue(Integer valeur) {
    this.value = valeur;
}

/**
 * Obtient le nœud gauche.
 *
 * @return Le nœud gauche.
 */
15 usages ± cedric.alonso
public Node getLeft() {
    return left;
}

/**
 * Définit le nœud gauche.
 *
 * @param noeud Le nœud gauche.
 */
2 usages ± mrredcoding
public void setLeft(Node noeud) {
    this.left = noeud;
}

/**
 * Obtient le nœud droit.
 *
 * @return Le nœud droit.
 */
17 usages ± cedric.alonso
public Node getRight() {
    return right;
}

/**
 * Définit le nœud droit.
 *
 * @param noeud Le nœud droit.
 */
3 usages ± mrredcoding
public void setRight(Node noeud) {
    this.right = noeud;
}

/**
 * Obtient une représentation sous forme de chaîne de caractères du nœud.
 *
 * @return La représentation du nœud en tant que chaîne de caractères.
 */
± cedric.alonso
@Override
public String toString() {
    return String.valueOf(this.value);
}
```

```

package TP6;

import TP2.stack.EmptyStackExceptions;

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

/**
 * Implementation of a Binary Search Tree (BST).
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hataoum
 */
7 usages  ± cedric.alonso +1
public class BST {

    /**
     * Maximum number of nodes for each level in the BST.
     */
1 usage
    public static final int MAX_NODES_PER_LEVEL = 2;

    16 usages
    private Node root;

    /**
     * Constructs an empty Binary Search Tree.
     */
1 usage  ± cedric.alonso
    public BST() {
        this.root = new Node();
    }

    /**
     * Inserts a value into the Binary Search Tree.
     *
     * @param value The value to insert.
     */
1 usage  ± cedric.alonso
    public void insert(Integer value) {
        this.root = insert(value, this.root);
    }

    3 usages  ± cedric.alonso
    private Node insert(Integer value, Node source) {
        if (source == null || source.getValue() == null)
            return new Node(value);
        else if (value > source.getValue())
            source.setRight(insert(value, source.getRight()));
        else if (value <= source.getValue())
            source.setLeft(insert(value, source.getLeft()));

        return source;
    }

    /**
     * Performs an in-order traversal of the Binary Search Tree.
     *
     * @return List of integers representing the in-order traversal.
     */
3 usages  ± mrredcoding
    public List<Integer> inOrder() {
        List<Integer> inOrderTraversal = new ArrayList<>();
        inOrder(root, inOrderTraversal);
        return inOrderTraversal;
    }
}

```

```

3 usages ± mrredcoding
private void inOrder(Node node, List<Integer> traversal) {
    if (node != null) {
        inOrder(node.getLeft(), traversal);
        traversal.add(node.getValue());
        inOrder(node.getRight(), traversal);
    }
}

/**
 * Performs a pre-order traversal of the Binary Search Tree.
 *
 * @return List of integers representing the pre-order traversal.
 */
1 usage ± cedric.alonso +1
public List<Integer> preOrder() {
    List<Integer> preOrderTraversal = new ArrayList<>();
    preOrder(root, preOrderTraversal);
    return preOrderTraversal;
}

3 usages ± cedric.alonso +1
private void preOrder(Node node, List<Integer> traversal) {
    if (node != null) {
        traversal.add(node.getValue());
        preOrder(node.getLeft(), traversal);
        preOrder(node.getRight(), traversal);
    }
}

/**
 * Performs a post-order traversal of the Binary Search Tree.
 *
 * @return List of integers representing the post-order traversal.
 */
1 usage ± cedric.alonso +1
public List<Integer> postOrder() {
    List<Integer> postOrderTraversal = new ArrayList<>();
    postOrder(root, postOrderTraversal);
    return postOrderTraversal;
}

private void postOrder(Node node, List<Integer> traversal) {
    if (node != null) {
        postOrder(node.getLeft(), traversal);
        postOrder(node.getRight(), traversal);
        traversal.add(node.getValue());
    }
}

/**
 * Performs a breadth-first search (BFS) traversal of the Binary Search Tree.
 *
 * @return List of integers representing the BFS traversal.
 * @throws QueueException if there is an issue with the queue.
 */
1 usage new
public List<Integer> breadthFirstSearch() {
    try {
        List<Integer> BfsTraversal = new ArrayList<>();
        breadthFirstSearch(root, BfsTraversal);
        return BfsTraversal;
    } catch (QueueException e) {
        System.err.println("There was an issue with the queue");
        return null;
    }
}

```

```

1 usage  new *
private void breadthFirstSearch(Node source, List<Integer> traversal) throws QueueException {
    if (source == null)
        return;

    Queue<Node> queue = new Queue<>( nbMaxElements: MAX_CHILDREN_PER_NODE * 2);
    queue.push(source);

    while (!queue.isEmpty()) {
        Node current = queue.pop();
        traversal.add(current.getValue());

        if (current.getLeft() != null)
            queue.push(current.getLeft());
        if (current.getRight() != null)
            queue.push(current.getRight());
    }
}

/** 
 * Gets the maximum value in the Binary Search Tree.
 *
 * @return The maximum value in the Binary Search Tree.
 * @throws BSTException if the Binary Search Tree is empty.
 */
1 usage  ± cedric.alonso
public Integer getMaxValue() throws BSTException {
    if (isEmpty())
        throw new BSTException("BST is empty");

    return maxValue(this.root);
}

1 usage  ± cedric.alonso +1
private Integer maxValue(Node node) {
    Integer maxValue = node.getValue();
    while (node.getRight() != null) {
        if (node.getRight().getValue() != null)
            maxValue = node.getRight().getValue();
        node = node.getRight();
    }
    return maxValue;
}

/** 
 * Gets the minimum value in the Binary Search Tree.
 *
 * @return The minimum value in the Binary Search Tree.
 * @throws BSTException if the Binary Search Tree is empty.
 */
1 usage  ± cedric.alonso
public Integer getMinValue() throws BSTException {
    if (isEmpty())
        throw new BSTException("BST is empty");

    return minValue(this.root);
}

private Integer minValue(Node node) {
    Integer minValue = node.getValue();
    while (node.getLeft() != null) {
        if (node.getLeft().getValue() != null)
            minValue = node.getLeft().getValue();
        node = node.getLeft();
    }
    return minValue;
}

```

```

/**
 * Gets the mean value of the Binary Search Tree.
 *
 * @return The mean value of the Binary Search Tree.
 * @throws BSTException if the Binary Search Tree is empty.
 * @throws EmptyStackExceptions if there is an empty stack while calculating the mean value.
 */
1usage  ± mrrredcoding
public Integer getMeanValue() throws BSTException, EmptyStackExceptions {
    List<Integer> allValues = inOrder();
    int sum = 0;
    for (Integer value : allValues)
        sum += value;

    return sum / allValues.size();
}

/**
 * Calculates the height of the Binary Search Tree.
 *
 * @return The height of the Binary Search Tree.
 */
1usage  ± cedric.alonso
public int height() {
    return height(root);
}

5 usages  ± cedric.alonso
private int height(Node node) {
    if (node == null)
        return 0;

    int leftHeight = height(node.getLeft());
    int rightHeight = height(node.getRight());

    return Math.max(leftHeight, rightHeight) + 1;
}

/**
 * Counts the number of non-null nodes in the Binary Search Tree.
 *
 * @return The number of non-null nodes in the Binary Search Tree.
 */
1usage  ± cedric.alonso
public int nbNodes() {
    return nbNodes(root);
}

3 usages  ± cedric.alonso
private int nbNodes(Node node) {
    if (node == null)
        return 0;

    int leftNodes = nbNodes(node.getLeft());
    int rightNodes = nbNodes(node.getRight());

    return leftNodes + rightNodes + 1;
}

 /**
 * Searches for a node with the specified value in the Binary Search Tree.
 *
 * @param value The value to search for.
 * @throws BSTException if the Binary Search Tree is empty.
 * @return True if the value to search for is present in the Binary Search Tree, false otherwise.
 */
3 usages  ± cedric.alonso
public boolean search(Integer value) throws BSTException {
    if (this.isEmpty())
        throw new BSTException("BST is empty");
    return search(value, root);
}

```

```

3 usages  ± cedric.alonso
> private boolean search(Integer value, Node node) {
|   if (node == null)
|       return false;
|   if (value.equals(node.getValue()))
|       return true;
|
|   if (value < node.getValue())
|       return search(value, node.getLeft());
|   else
|       return search(value, node.getRight());
}
*/
* Removes a node with the specified value from the Binary Search Tree if it exists.
*
* @param value The value to remove.
* @throws BSTException if the value to remove does not belong to the tree.
*/
1 usage  ± cedric.alonso
public void remove(Integer value) throws BSTException {
    if (!this.search(value))
        throw new BSTException("The value '" + value + "' can't be removed because it doesn't belong to the tree");
    root = remove(value, root);
}

4 usages  ± cedric.alonso
private Node remove(Integer value, Node node) {
    if (node == null)
        return null;
    |
    if (value < node.getValue()) {
        node.setLeft(remove(value, node.getLeft()));
    } else if (value > node.getValue()) {
        node.setRight(remove(value, node.getRight()));
    } else {
        if (node.getLeft() == null)
            return node.getRight();
        else if (node.getRight() == null)
            return node.getLeft();
        |
        node.setValue(minValue(node.getRight()));
        |
        node.setRight(remove(node.getValue(), node.getRight()));
    }
    |
    return node;
}

*/
* Checks if the Binary Search Tree is empty.
*
* @return True if the Binary Search Tree is empty, false otherwise.
*/
± cedric.alonso
public boolean isEmpty() {
    return this.root == null || this.root.getValue() == null;
}

*/
* Returns a string representation of the Binary Search Tree.
*
* @return The string representation of the Binary Search Tree.
*/
± cedric.alonso
@Override
public String toString() {
    if (this.isEmpty())
        return "BST is empty";
    |
    return "BST : " + inOrder();
}
}

```

### 3. Les tests et résultats de l'implémentation

```
package TP0;

import TP2.stack.EmptyStackExceptions;
import TP2.utils.Test;

import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * The main class provides a menu-driven interface for users to perform various operations on a BST, including
 * adding, searching, and removing nodes, as well as displaying statistics about the BST.
 * The user interacts with the program through the console by choosing options from the menu.
 * The program handles user input, performs the requested operations on the BST, and displays the results.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
public class Main {

    6 usages
    private static final Scanner scanner = new Scanner(System.in);

    ± cedric.alonso
    public static void main(String[] args) throws BSTException, EmptyStackExceptions {
        BST bst = new BST();
        menu(bst);
    }

    /**
     * The menu method handles the main interaction loop, allowing the user to choose various operations
     * on the Binary Search Tree until the program is exited.
     *
     * @param bst The Binary Search Tree instance
     */
    1 usage ± cedric.alonso
    private static void menu(BST bst) {
        int choice = 0;
        do {
            displayMenu();
            System.out.print("Enter your choice: ");
            try {
                choice = scanner.nextInt();
                switch (choice) {
                    case 1:
                        adding(bst);
                        break;
                    case 2:
                        searching(bst);
                        break;
                    case 3:
                        removing(bst);
                        break;
                    case 4:
                        displayStatistics(bst);
                        break;
                    case 5:
                        exit();
                    default:
                        System.err.println("Your choice must be in the range [1;5]");
                        scanner.nextLine();
                }
            } catch (InputMismatchException e) {
                System.err.println("Your choice must be an integer");
                scanner.nextLine();
            }
        } while (choice != 5);
    }
}
```

```

    /**
     * Displays the main menu with available options for the user.
     */
    1usage += cedric.alonso
    private static void displayMenu() {
        System.out.println(Test.titleBox("***** MENU *****"));
        System.out.println("1. Add new nodes to the Binary Search Tree");
        System.out.println("2. Search for nodes in the Binary Search Tree");
        System.out.println("3. Remove nodes from the Binary Search Tree");
        System.out.println("4. Display some statistics of the Binary Search Tree");
        System.out.println("5. Exit the program");
    }

    /**
     * Handles the removal of nodes from the Binary Search Tree based on user input.
     *
     * @param bst The Binary Search Tree instance
     */
    1usage += cedric.alonso+1
    private static void removing(BST bst) {
        if (bst.isEmpty()) {
            System.err.println("BST is empty");
            return;
        }

        String choice;
        do {
            System.out.println("Enter a number to remove in the binary search tree: (type 'stop' to continue)");
            choice = scanner.next();
            if (!"stop".equalsIgnoreCase(choice)) {
                try {
                    int number = Integer.parseInt(choice);
                    boolean found = bst.search(number);

                    if (found) {
                        bst.remove(number);
                        System.out.println("The number " + number + " has been removed from the binary search tree");
                    } else
                        System.out.println("The number " + number + " does not belong to the tree");

                } catch (NumberFormatException e) {
                    System.err.println("You need to type only numbers or 'stop'");
                    choice = "continue";
                } catch (BSTException e) {
                    System.out.println(e.getMessage());
                    choice = "stop";
                }
            }
        } while (!"stop".equalsIgnoreCase(choice));
    }

    /**
     * Handles the searching of nodes in the Binary Search Tree based on user input.
     *
     * @param bst The Binary Search Tree instance
     */
    1usage += cedric.alonso
    private static void searching(BST bst) {
        if (bst.isEmpty()) {
            System.err.println("BST is empty");
            return;
        }

        String choice;
        do {
            System.out.println("Enter a number to find in the binary search tree: (type 'stop' to continue)");
            choice = scanner.next();
            if (!"stop".equalsIgnoreCase(choice)) {
                try {
                    int number = Integer.parseInt(choice);
                    boolean found = bst.search(number);

                    System.out.println("The number " + number + " is " + (!found ? "not" : "") +
                        " in the binary search tree");
                } catch (NumberFormatException e) {
                    System.err.println("You need to type only numbers or 'stop'");
                    choice = "continue";
                } catch (BSTException e) {
                    System.out.println(e.getMessage());
                    choice = "stop";
                }
            }
        } while (!"stop".equalsIgnoreCase(choice));
    }
}

```

```

    /**
     * Handles the addition of nodes to the Binary Search Tree based on user input.
     *
     * @param bst The Binary Search Tree instance
     */
    1 usage  ± cedric.alonso
    private static void adding(BST bst) {
        String choice;
        do {
            System.out.println("Enter a number to add in the binary search tree: (type 'stop' when you are done)");
            choice = scanner.next();
            if (!"stop".equalsIgnoreCase(choice)) {
                try {
                    int number = Integer.parseInt(choice);
                    bst.insert(number);
                    System.out.println("The number " + number + " has been added to the binary search tree");
                } catch (NumberFormatException e) {
                    System.err.println("You need to type only numbers or 'stop'");
                    choice = "continue";
                }
            }
        } while (!"stop".equalsIgnoreCase(choice));
    }

    /**
     * Displays various statistics of the Binary Search Tree, such as in-order, post-order, and pre-order traversals,
     * as well as minimum, maximum, and mean values, height, breadth, and the number of nodes.
     *
     * @param bst The Binary Search Tree instance
     */
    1 usage  ± cedric.alonso
    private static void displayStatistics(BST bst) {
        try {
            System.out.println("In order: " + bst.inOrder());
            System.out.println("Post order: " + bst.postOrder());
            System.out.println("Pre order: " + bst.preOrder());

            System.out.println("Breadth: " + bst.breadthFirstSearch());

            System.out.println("Minimal value: " + bst.getMinValue());
            System.out.println("Maximal value: " + bst.getMaxValue());
            System.out.println("Mean value: " + bst.getMeanValue());

            System.out.println("Height: " + bst.height());
            System.out.println("Number of nodes: " + bst.nbNodes());
        } catch (BSTException | EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Exits the program, displaying a farewell message.
     */
    1 usage  ± cedric.alonso
    private static void exit() {
        System.out.println("You exited the program ^w^");
        System.exit(status: 0);
    }
}

```

Année académique 2023 - 2024