Assignment III: AVL Tree Implementation

Student Name: Orlando Alvarado Vargas
Student ID: 2226968

Instructions:

You are required to implement an AVL Tree to store double values using Object-Oriented Programming (OOP) principles in C++.
Your program should:

- Define a class for **AVLTree** that supports insertion, deletion, and searching of double values.
- Implement balancing operations to maintain the AVL property after insertions and deletions.
- Study and discuss the effect of floating-point precision on the correctness and balancing of the AVL tree (e.g., how comparing double values may affect insertions, deletions, and tree structure). Provide examples or test cases illustrating potential issues and how to handle them.
- Implement traversal of the tree in-order, pre-order, and post-order.
- Implement a method to merge two AVL trees without losing the AVL property:
  1. Convert both trees to sorted arrays using in-order traversal.
  2. Merge the two sorted arrays into one efficiently.
  3. Construct a new AVL tree from the merged sorted array.
  4. Include several test cases to demonstrate the merging process and the correctness of the resulting AVL tree.
- Demonstrate encapsulation and other OOP concepts such as method overloading and operator overloading where applicable.

Submission Requirements:

- Include your well-commented source code in the report.
- Provide sample test cases and their outputs.
- Briefly explain your design choices and how OOP principles are applied.
- Ensure your code is well-structured and follows best practices.

A Introduction

In this report, I will explain how I implemented the AVL Tree for the assignment in C++. With this system, the user can insert, search, and delete double values, and the Tree manages to keep its balance automatically through rotations. The program also allows the Tree to be shown in different orders (inorder, preorder, postorder) and offers a function to merge two AVL Trees, keeping the balance.

B Class Definitions

First of all, for the implementation of the AVL Tree, I defined a structure and a class with their own attributes and functions:

- **Node:** Represents each individual node in the AVL Tree. This structure has a `double` value, its height, and left and right pointers that point to the children of the node.
- **Class:** Manages the complete AVL Tree and its main root. It implements the system's main functions: inserting, searching, and deleting values, displaying the Tree in inorder, preorder, and postorder, and offers the function to merge two AVL trees. Also, the Tree has private functions to rotate and update heights in order to keep the Tree balanced.

B.1 Node Structure

```
// ---------- Definition of the Node structure ----------
struct Node{
    //---------- Atributes ----------
    double value;  // The value stored in the node
    int height;    // The height of the subtree rooted at this node
    Node* left;    // Pointer to the left child
    Node* right;   // Pointer to the right child

    // Constructor
    Node(double val){
        value = val;
        height = 1;
        left = nullptr;
        right = nullptr;
    }
};
```

B.2 Tree class

```
// ---------- Definition of the Tree class ----------
class Tree{
    private:
        Node* root; // The main root of the tree

        int getHeight(Node* node);  // Get height of a node
        int getFE(Node* node);  // Get Balance Factor (FE)

        Node* rightRotate(Node* y); // Performs a right rotation
        Node* leftRotate(Node* x);  // Performs a left rotation

        Node* insert(Node* node, double value); // Recursive function
           to insert a node
        Node* search(Node* node, double value); // Recursive function
           to search a node
        Node* erase(Node* node, double value); // Recursive function
           to erase a node

        void inorder(Node* node);   // Inorder Traversal (L, Root, R)
        void preorder(Node*node);   // Preorder Traversal (Root, L, R)
        void postorder(Node* node); // Postorder Traversal (L, R,
           Root)

        void destroy(Node* node);   // Destroy tree (free memory)

        void to_vector(Node* node, vector<double>& current_tree);
           // Convert the Tree to a vector
```

```
        Node* from_vector(const vector<double>& vec, int start, int
            end);   // Build a tree from a vector

    // --- Public Interface ---
    public:
        Tree(); // Constructor
        ~Tree();    // Destructor

        void insert(double value);  // Public insert
        void inorder(); // Public inorder
        void preorder();    // Public preorder
        void postorder();   // Public postorder
        bool search(double value);  //  Public search
        void erase(double value);  // Public erase
        void merge(Tree& otherTree);    // Merge two trees
};
```

C Method Implementations

This section describes the implementation of the constructors, methods, and functions of the system's classes. A key implementation detail is the use of a global constant `e` (epsilon), which is used to compare `double` values. This approach helps manage the imprecision inherent in floating-point arithmetic, ensuring that operations like searching or checking for duplicates are reliable.

```
 // Beginning of the Code.
using namespace std;

//Defined a global constant "e" (epsilon) to compare double values
//This helps to get rid of the imprecision
const double e = 1e-9;
```

C.1 Tree class

In order of maintaining the AVL characteristics of being balanced, I implemented some methods like the self-balancing mechanism which involves the private insert, erase, left rotate, and right rotate functions. This self-balancing logic is the most complex part of the implementation, as it must correctly identify and resolve imbalances by checking the Balance Factor and updating the height of the nodes after every operation. This method is so important that it repeats in the insert and erase functions.

```
    node->height = max(getHeight(node->left), getHeight(node->right))
       + 1;
    int balance = getFE(node);

    if (balance > 1 && getFE(node->left) >= 0){
        return rightRotate(node);
    }
    if (balance < -1 && getFE(node->right) <= 0){
        return leftRotate(node);
    }
    if (balance > 1 && getFE(node->left) < 0){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && getFE(node->right) > 0){
        node->right = rightRotate(node->right);
```

```
        return leftRotate(node);
    }

    return node;
```

D Main Function

In the main function, an instance of the Tree class is created, and a simple menu is built to interact with the system. This menu allows the user to access the functions implemented in the Tree class.

D.1 Structure

The menu operates within a do-while loop, which repeats until the user selects the exit option. First, the menu options are presented, and the user's selection is read into a selection variable. A switch statement then handles the user's choice. If the user selects 0, the system closes. If the user enters a value outside the expected range, the system displays a warning message.

```
int main(){
    Tree tree;  // Create tree instance

    // Menu variables
    int selection;
    double val;

    do{ // Menu loop
        // Print menu options
        cout << "\n␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Little␣Garden\n"<<
                "----------------------WELCOME--------------------\n"
                   <<
                "\nPick␣a␣number␣to␣select␣any␣of␣this␣options:\n" <<
                "1)␣Insert␣a␣new␣number.\n" <<
                "2)␣Search␣a␣number.\n" <<
                "3)␣Delete␣a␣number.\n" <<
                "4)␣See␣the␣tree␣in␣order.\n" <<
                "5)␣Merge␣to␣another␣tree.\n" <<
                "0)␣Exit␣the␣system\n";
        cout << "\nSelection:␣";
        cin >> selection;

        // . . . Menu . . .

        switch (selection){     // Handle selection
            case 0:{            // Exit
                cout << "\nHave␣a␣good␣day!!!\n";
                cout << "\n====================␣␣␣␣CLOSING␣SYSTEM␣␣␣␣
                   ====================\n";
                break;
            }

            default:{       // Invalid input guard for numbers
               outside the expected range
                cout << "\n############␣␣␣␣␣␣Please␣select␣a␣number␣
                   between␣0␣and␣5␣␣␣>:(␣␣␣␣␣␣############\n";
            }
        }
```

```
    }while(selection != 0); // Repeat until user selects 0

    return 0;
}
```

D.2 Insert Value

This option asks for a double value. The insert function is then called with this value, which adds the new element to the tree and automatically balances it if necessary.

```
case 1:{            // Insert
    cout << "\n----------------------NEW␣
       VALUE----------------------\n";
    cout << "\nPlease␣enter␣the␣value:␣";
    cin >> val;
    tree.insert(val);
    cout <<
       "\n-----------------------------------------------------\n";
    break;
}
```

D.3 Search a Value

The system asks for a value and then uses the search function in the Tree class. If the value exists in the Tree, the system prints a confirmation message. Otherwise, it informs the user that the value does not exist in the tree.

```
case 2:{            // Search
    cout <<
       "\n------------------------SEARCH----------------------\n";
    cout << "Enter␣the␣value␣you␣want␣to␣search:␣";
    cin >> val;
    cout << "\n...Searching...\n";
    if (tree.search(val)){
        cout << val << "␣exists␣in␣the␣Tree␣:D\n";
    }
    else{
        cout << val << "␣does␣not␣exist␣in␣the␣Tree␣D:\n";
    }
    cout <<
       "\n-----------------------------------------------------\n";
    break;
}
```

D.4 Delete Values

This function works the opposite way of the insert function. First of all, it verifies if the value exists in the Tree using the search function. If it exists, the system calls the erase method to delete the corresponding node and balance the Tree if it is needed.

```
case 3:{            // Delete
    cout << "\n--------------------DELETE␣
       VALUES--------------------\n";
    cout << "Enter␣the␣value␣you␣want␣to␣errase:␣";
    cin >> val;
    if (!tree.search(val)){     // Check if value exists
        cout << val << "␣does␣not␣exist␣in␣the␣Tree␣D:\n";
    }
```

```
    else {
        tree.erase(val);
        cout << "\n...Deleting value...\n";
    }
    cout <<
       "\n-----------------------------------------------------\n";
    break;
}
```

D.5 See Tree

This option calls the inorder, preorder, and postorder functions. It simply displays the tree's contents according to each of these three traversal orders.

```
case 4:{            // Display
    cout << "\n----------------------INSIDE
       TREE----------------------\n";
    cout << "Tree inorder: " << endl;
    cout << "   ";
    tree.inorder();
    cout << "\nTree preorder: " << endl;
    cout << "   ";
    tree.preorder();
    cout << "\nTree postorder: " << endl;
    cout << "   ";
    tree.postorder();
    cout <<
       "\n-----------------------------------------------------\n";
    break;
}
```

D.6 Merge Two Trees

This creates a new temporary instance for other Tree and asks the user for double values in the new tree. All the values are captured with a for loop, inserting them directly into the other Tree instance. The system then displays this new tree and asks for a confirmation to follow the process. If confirmed, the merge function is called. This merge method then handles the process of converting both trees to arrays, merging them, and rebuilding the main tree from the final sorted array.

```
case 5:{            // Merge
    cout <<
       "\n-----------------------MERGING----------------------\n";
    Tree otherTree; // Create a temporary tree
    int amount;
    int merge_selection;
    cout << "\nPlease enter the amount of values: ";
    cin >> amount;
    // Fill the temp tree
    for (int i = 0; i < amount; i++){
        cout << "Value " << i + 1 << " : ";
        cin >> val;
        otherTree.insert(val);
    }
    // Confirm the temp tree
    cout << "\nThanks , this is the Tree inorder about to merge: " <<
       endl;
```

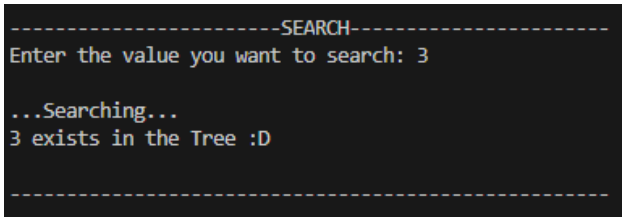
```
    cout << "␣␣␣";
    otherTree.inorder();
    cout << "\nIs␣it␣okay␣to␣continue?\n" <<
            "1)␣Yes\n" <<
            "2)␣No\n";
    cout << "Selection:␣";
    cin >> merge_selection;
    // Merge Trees
    if (merge_selection == 1){
        cout << "\n...Merging␣Trees...\n";
        tree.merge(otherTree);
        cout << "\nTrees␣merged␣:D\n";
        cout <<
           "\n-------------------------------------------------------\n";
        break;
    }
    else if(merge_selection == 2){
        cout << "\nOkay,␣try␣again.\n";
        cout <<
           "\n-------------------------------------------------------\n";
        break;
    }
    else{
        cout << "\n############␣␣␣␣␣Please␣select␣a␣number␣between␣1␣
           and␣2␣␣>:(␣␣␣␣␣############\n";
        cout << "\nTry␣again.\n";
        cout <<
           "\n-------------------------------------------------------\n";
        break;
    }
    cout <<
       "\n-------------------------------------------------------\n";
    break;
}
```

E Test Cases

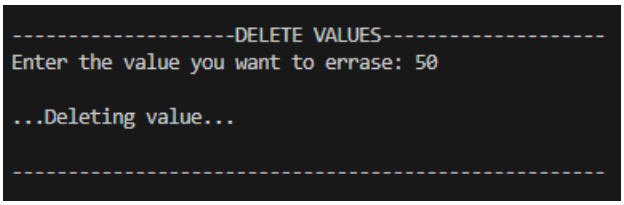
Here are some examples of how the menu looks.

Test Case: Main Menu

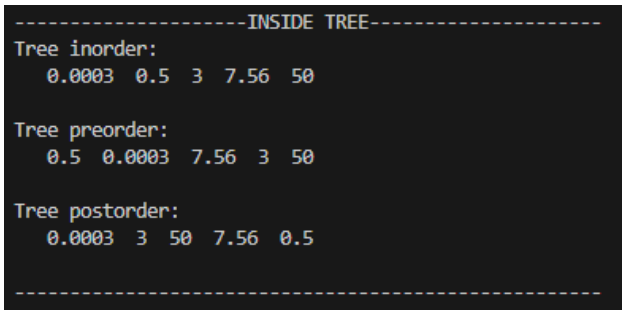
```
                    Little Garden
----------------------WELCOME-----------------------

Pick a number to select any of this options:
1) Insert a new number.
2) Search a number.
3) Delete a number.
4) See the tree in order.
5) Merge to another tree.
0) Exit the system

Selection:
```



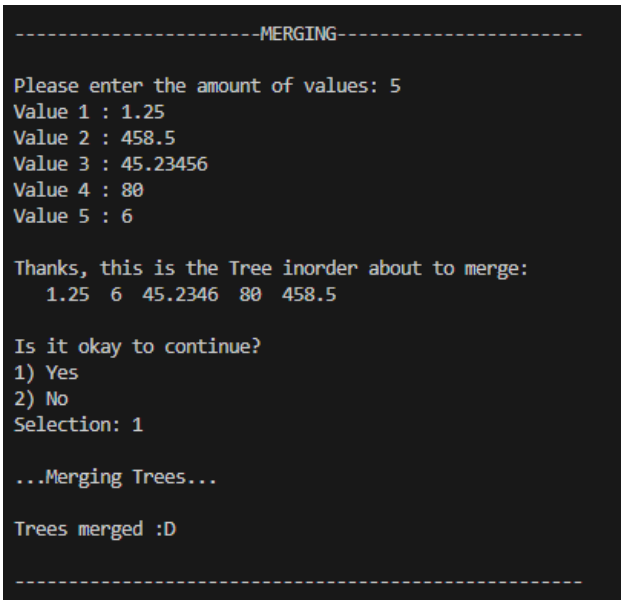
Search Value



Delete Value



See the tree



Merge Trees

Figure 1: Screenshots of different library operations.